Intel® 80303 I/O Processor Developer's Manual

developer.intel.com

May 2000
Order Number: 273353-001

Intel® 80303 I/O Processor

Developer's Manual

Revision 0.5

May 2000

Order Number: 273353-001

Contents

1 Introduction 1-1

1.1 Intel® 80303 I/O Processor 1-1
1.2 Intel® 80303 I/O Processor Features 1-2
1.2.1 PCI-to-PCI Bridge Unit 1-2
1.2.2 Internal Bus 1-2
1.2.3 Private PCI Device Support 1-3
1.2.4 DMA Controller 1-3
1.2.5 Address Translation Unit 1-3
1.2.6 Messaging Unit 1-3
1.2.7 Memory Controller 1-3
1.2.8 Application Accelerator Unit 1-3
1.2.9 Performance Monitoring Unit 1-4
1.2.10 I^2C Bus Interface Unit 1-4
1.2.11 GPIO Interface Unit 1-4
1.2.12 Secondary PCI Arbitration Unit 1-4
1.3 Terminology and Conventions 1-5
1.3.1 Representing Numbers 1-5
1.3.2 Fields 1-5
1.3.3 Specifying Bit and Signal Values 1-5
1.3.4 Signal Name Conventions 1-5
1.3.5 Terminology 1-6

2 Data Types and Memory Addressing Modes 2-1

2.1 Data Types 2-1
2.1.1 Word/Dword Notation 2-2
2.1.2 Integers 2-2
2.1.3 Ordinals 2-3
2.1.4 Bits and Bit Fields 2-3
2.1.5 Triple and Quad Words 2-3
2.1.6 Register Data Alignment 2-3
2.1.7 Literals 2-4
2.2 Bit and Byte Ordering in Memory 2-4
2.3 Memory Addressing Modes 2-4
2.3.1 Absolute 2-5
2.3.2 Register Indirect 2-5
2.3.3 Index with Displacement 2-5
2.3.4 IP with Displacement 2-6
2.3.5 Addressing Mode Examples 2-6

3 Programming Environment 3-1

3.1 Overview 3-1
3.2 Registers and Literals as Instruction Operands 3-2
3.2.1 Global Registers 3-3
3.2.2 Local Registers 3-3
3.2.3 Register Scoreboarding 3-4
3.2.4 Literals 3-4

3.2.5 Register and Literal Addressing and Alignment 3-4
3.3 Memory-Mapped Control Registers (MMRs) 3-6
3.3.1 Intel® i960® Core Processor Function Memory-Mapped Registers 3-6
3.3.1.1 Restrictions on Instructions that Access the Intel® i960® Core Processor Memory-Mapped Registers 3-6
3.3.1.2 Access Faults for Intel® i960® Core Processor MMRs 3-7
3.3.2 Intel® 80303 I/O Processor Peripheral Memory-Mapped Registers 3-7
3.3.2.1 Accessing The Peripheral Memory-Mapped Registers 3-8
3.4 Architecturally Defined Data Structures 3-9
3.5 Memory Address Space 3-10
3.5.1 Memory Requirements 3-11
3.5.2 Data and Instruction Alignment in the Address Space 3-12
3.5.3 Byte, Word and Bit Addressing 3-12
3.5.4 Internal Data RAM 3-13
3.5.5 Instruction Cache 3-13
3.5.6 Data Cache 3-13
3.6 Processor-State Registers 3-13
3.6.1 Instruction Pointer (IP) Register 3-13
3.6.2 Arithmetic Controls Register – AC 3-14
3.6.2.1 Initializing and Modifying the AC Register 3-14
3.6.2.2 Condition Code (AC.cc) 3-15
3.6.3 Process Controls Register – PC 3-16
3.6.3.1 Initializing and Modifying the PC Register 3-17
3.6.4 Trace Controls (TC) Register 3-17
3.7 User-Supervisor Protection Model 3-18
3.7.1 Supervisor Mode Resources 3-18
3.7.2 Using the User-Supervisor Protection Model 3-19

4 Cache and On-Chip Data RAM 4-1

4.1 Internal Data RAM 4-1
4.2 Local Register Cache 4-2
4.3 Instruction Cache 4-3
4.3.1 Enabling and Disabling the Instruction Cache 4-4
4.3.2 Operation While the Instruction Cache Is Disabled 4-4
4.3.3 Loading and Locking Instructions in the Instruction Cache 4-4
4.3.4 Instruction Cache Visibility 4-5
4.3.5 Instruction Cache Coherency 4-5
4.4 Data Cache 4-5
4.4.1 Enabling and Disabling the Data Cache 4-5
4.4.2 Multi-Word Data Accesses that Partially Hit the Data Cache 4-6
4.4.3 Data Cache Fill Policy 4-6
4.4.4 Data Cache Write Policy 4-7
4.4.5 Data Cache Coherency and Non-Cacheable Accesses 4-8
4.4.6 External I/O Bus Masters and Cache Coherency 4-8
4.4.7 Data Cache Visibility 4-8

5 Instruction Set Overview 5-1

5.1 Instruction Formats 5-1
5.1.1 Assembly Language Format 5-1
5.1.2 Instruction Encoding Formats 5-2
5.1.3 Instruction Operands 5-3

5.2 Instruction Groups 5-4
5.2.1 Data Movement 5-5
5.2.1.1 Load and Store Instructions 5-5
5.2.1.2 Move 5-6
5.2.1.3 Load Address 5-6
5.2.2 Select Conditional 5-6
5.2.3 Arithmetic 5-7
5.2.3.1 Add, Subtract, Multiply, Divide, Conditional Add, Conditional Subtract 5-7
5.2.3.2 Remainder and Modulo 5-8
5.2.3.3 Shift, Rotate and Extended Shift 5-8
5.2.3.4 Extended Arithmetic 5-9
5.2.4 Logical 5-9
5.2.5 Bit, Bit Field and Byte Operations 5-10
5.2.5.1 Bit Operations 5-10
5.2.5.2 Bit Field Operations 5-10
5.2.5.3 Byte Operations 5-10
5.2.6 Comparison 5-11
5.2.6.1 Compare and Conditional Compare 5-11
5.2.6.2 Compare and Increment or Decrement 5-12
5.2.6.3 Test Condition Codes 5-12
5.2.7 Branch 5-12
5.2.7.1 Unconditional Branch 5-13
5.2.7.2 Conditional Branch 5-13
5.2.7.3 Compare and Branch 5-14
5.2.8 Call/Return 5-15
5.2.9 Faults 5-16
5.2.10 Debug 5-16
5.2.11 Atomic Instructions 5-17
5.2.12 Processor Management 5-17
5.3 Performance Optimization 5-18
5.3.1 Instruction Optimizations 5-18
5.3.1.1 Load / Store Execution Model 5-18
5.3.1.2 Compare Operations 5-18
5.3.1.3 Microcoded Instructions 5-18
5.3.1.4 Multiply-Divide Unit Instructions 5-19
5.3.1.5 Multi-Cycle Register Operations 5-19
5.3.1.6 Simple Control Transfer 5-19
5.3.1.7 Memory Instructions 5-20
5.3.1.8 Unaligned Memory Accesses 5-20
5.3.2 Miscellaneous Optimizations 5-21
5.3.2.1 Masking of Integer Overflow 5-21
5.3.2.2 Avoid Using PFP, SP, R3 As Destinations for MDU Instructions 5-21
5.3.2.3 Use Global Registers (g0 - g14) As Destinations for MDU Instructions 5-21
5.3.2.4 Execute in Imprecise Fault Mode 5-21
5.3.3 Cache Control 5-21

6 Instruction Set Reference 6-1

6.1 Notation 6-2
6.1.1 Alphabetic Reference 6-2
6.1.2 Mnemonic 6-2
6.1.3 Format 6-2
6.1.4 Description 6-3

6.1.5 Action 6-3
6.1.6 Faults 6-5
6.1.7 Example 6-5
6.1.8 Opcode and Instruction Format 6-5
6.1.9 See Also 6-5
6.1.10 Side Effects 6-5
6.1.11 Notes 6-5
6.2 Instructions 6-6
6.2.1 ADD<cc> 6-6
6.2.2 addc 6-9
6.2.3 addi, addo 6-10
6.2.4 alterbit 6-11
6.2.5 and, andnot 6-12
6.2.6 atadd 6-13
6.2.7 atmod 6-14
6.2.8 b, bx 6-15
6.2.9 bal, balx 6-16
6.2.10 bbc, bbs 6-17
6.2.11 BRANCH<cc> 6-18
6.2.12 bswap 6-20
6.2.13 call 6-21
6.2.14 calls 6-22
6.2.15 callx 6-24
6.2.16 chkbit 6-25
6.2.17 clrbit 6-26
6.2.18 cmpdeci, cmpdeco 6-27
6.2.19 cmpinci, cmpinco 6-28
6.2.20 COMPARE 6-29
6.2.21 COMPARE AND BRANCH<cc> 6-30
6.2.22 concmpi, concmpo 6-32
6.2.23 dcctl 6-33
6.2.24 divi, divo 6-38
6.2.25 ediv 6-39
6.2.26 emul 6-40
6.2.27 eshro 6-41
6.2.28 extract 6-42
6.2.29 FAULT<cc> 6-43
6.2.30 flushreg 6-44
6.2.31 fmark 6-45
6.2.32 halt 6-46
6.2.33 icctl 6-47
6.2.34 intctl 6-53
6.2.35 intdis 6-54
6.2.36 inten 6-55
6.2.37 LOAD 6-56
6.2.38 lda 6-59
6.2.39 mark 6-60
6.2.40 modac 6-61
6.2.41 modi 6-62
6.2.42 modify 6-63

6.2.43 modpc 6-64
6.2.44 modtc 6-65
6.2.45 MOVE 6-66
6.2.46 muli, mulo 6-68
6.2.47 nand 6-69
6.2.48 nor 6-70
6.2.49 not, notand 6-71
6.2.50 notbit 6-72
6.2.51 notor 6-73
6.2.52 or, ornot 6-74
6.2.53 remi, remo 6-75
6.2.54 ret 6-76
6.2.55 rotate 6-78
6.2.56 scanbit 6-79
6.2.57 scanbyte 6-80
6.2.58 SEL<cc> 6-81
6.2.59 setbit 6-82
6.2.60 SHIFT 6-83
6.2.61 spanbit 6-85
6.2.62 STORE 6-86
6.2.63 subc 6-90
6.2.64 SUB<cc> 6-91
6.2.65 subi, subo 6-93
6.2.66 syncf 6-94
6.2.67 sysctl 6-95
6.2.68 TEST<cc> 6-99
6.2.69 xnor, xor 6-100

7 Procedure Calls 7-1

7.1 Call and Return Mechanism 7-2
7.1.1 Local Registers and the Procedure Stack 7-2
7.1.2 Local Register and Stack Management 7-3
7.1.2.1 Frame Pointer 7-3
7.1.2.2 Stack Pointer 7-4
7.1.2.3 Considerations When Pushing Data onto the Stack 7-4
7.1.2.4 Considerations When Popping Data off the Stack 7-4
7.1.2.5 Previous Frame Pointer 7-4
7.1.2.6 Return Type Field 7-5
7.1.2.7 Return Instruction Pointer 7-5
7.1.3 Call and Return Action 7-5
7.1.3.1 Call Operation 7-6
7.1.3.2 Return Operation 7-6
7.1.4 Caching Local Register Sets 7-7
7.1.4.1 Reserving Local Register Sets for High Priority Interrupts 7-8
7.1.5 Mapping Local Registers to the Procedure Stack 7-11
7.2 Modifying the PFP Register 7-12
7.3 Parameter Passing 7-13
7.4 Local Calls 7-14
7.5 System Calls 7-15
7.5.1 System Procedure Table 7-15
7.5.1.1 Procedure Entries 7-16

7.5.1.2 Supervisor Stack Pointer 7-17
7.5.1.3 Trace Control Bit 7-17
7.5.2 System Call to a Local Procedure 7-17
7.5.3 System Call to a Supervisor Procedure 7-17
7.6 User and Supervisor Stacks 7-18
7.7 Interrupt and Fault Calls 7-18
7.8 Returns 7-19
7.9 Branch-and-Link 7-20

8 PCI and Peripheral Interrupt Controller Unit 8-1

8.1 Overview 8-1
8.1.1 The Intel® 80303 I/O Processor Core Interrupt Architecture 8-2
8.1.2 Software Requirements For Interrupt Handling 8-2
8.1.3 Interrupt Priority 8-3
8.1.4 Interrupt Table 8-4
8.1.4.1 Vector Entries 8-5
8.1.4.2 Pending Interrupts 8-5
8.1.4.3 Caching Portions of the Interrupt Table 8-5
8.1.5 Interrupt Stack And Interrupt Record 8-6
8.1.6 Posting Interrupts 8-7
8.1.6.1 Posting Software Interrupts via sysctl 8-7
8.1.6.2 Posting Software Interrupts Directly in the Interrupt Table 8-8
8.1.6.3 Posting External Interrupts 8-8
8.1.6.4 Posting Hardware Interrupts 8-8
8.1.7 Resolving Interrupt Priority 8-9
8.1.8 Sampling Pending Interrupts in the Interrupt Table 8-10
8.1.9 Saving the Interrupt Mask 8-11
8.2 The Intel® i960® Core Processor Interrupt Controller 8-12
8.2.1 Interrupt Controller Dedicated Mode 8-14
8.2.2 Interrupt Detection 8-15
8.2.3 Non-Maskable Interrupt (**NMI#**) 8-16
8.2.4 Timer Interrupts 8-16
8.2.5 Software Interrupts 8-16
8.2.6 Interrupt Operation Sequence 8-16
8.2.7 Setting Up the Interrupt Controller 8-17
8.2.8 Interrupt Service Routines 8-18
8.2.9 Interrupt Context Switch 8-18
8.2.9.1 Servicing An Interrupt From Executing State 8-19
8.2.9.2 Servicing An Interrupt From Interrupted State 8-19
8.3 Theory of Operation 8-20
8.4 Intel® 80303 I/O Processor Interrupts 8-21
8.4.1 PCI Interrupt Routing 8-23
8.4.2 Intel® 80303 I/O Processor: External Interrupt Interface 8-23
8.4.3 Intel® 80303 I/O Processor: Internal Peripheral Interrupt Routing 8-24
8.4.3.1 XINT6# Interrupt Sources 8-24
8.4.3.2 **XINT7#** Interrupt Sources 8-25
8.4.3.3 NMI# Interrupt Sources 8-25
8.4.4 PCI Outbound Doorbell Interrupts 8-27
8.5 Default Status 8-28
8.5.1 Interrupt Controller Register Access Requirements 8-28
8.6 Performance Requirements 8-28

8.6.1 Optimizing Interrupt Performance 8-29
8.6.2 Interrupt Service Latency 8-30
8.6.3 Features to Improve Interrupt Performance 8-30
8.6.3.1 Vector Caching Option 8-30
8.6.3.2 Caching Interrupt Routines and Reserving Register Frames 8-31
8.6.3.3 Caching the Interrupt Stack 8-31
8.6.4 Base Interrupt Latency 8-32
8.6.5 Maximum Interrupt Latency 8-33
8.6.6 Avoiding Certain Destinations for MDU Operations 8-34
8.6.7 Secondary PCI to Upstream Interrupt Routing Latency 8-34
8.7 Register Definitions 8-35
8.7.1 Interrupt Control Register (ICON) 8-36
8.7.2 Interrupt Mapping Registers (IMAP0-IMAP2) 8-37
8.7.3 Interrupt Pending (IPND) and Interrupt Mask (IMSK) Registers 8-39
8.7.4 PCI Interrupt Routing Select Register - PIRSR 8-41
8.7.5 XINT6 Interrupt Status Register - X6ISR 8-42
8.7.6 XINT7 Interrupt Status Register- X7ISR 8-43
8.7.7 NMI Interrupt Status Register - NISR 8-44

9 Faults 9-1

9.1 Fault Handling Overview 9-1
9.2 Fault Types 9-3
9.3 Fault Table 9-5
9.4 Stack Used in Fault Handling 9-7
9.5 Fault Record 9-7
9.5.1 Fault Record Description 9-8
9.5.2 Fault Record Location 9-9
9.6 Multiple and Parallel Faults 9-11
9.6.1 Multiple Non-Trace Faults on the Same Instruction 9-11
9.6.2 Multiple Trace Fault Conditions on the Same Instruction 9-11
9.6.3 Multiple Trace and Non-Trace Fault Conditions on the Same Instruction 9-11
9.6.4 Parallel Faults 9-11
9.6.4.1 Faults on Multiple Instructions Executed in Parallel 9-12
9.6.4.2 Fault Record for Parallel Faults 9-13
9.6.5 Override Faults 9-14
9.6.6 System Error 9-14
9.7 Fault Handling Procedures 9-15
9.7.1 Possible Fault Handling Procedure Actions 9-15
9.7.2 Program Resumption Following a Fault 9-15
9.7.2.1 Faults Happening Before Instruction Execution 9-15
9.7.2.2 Faults Happening During Instruction Execution 9-16
9.7.2.3 Faults Happening After Instruction Execution 9-16
9.7.3 Return Instruction Pointer (RIP) 9-16
9.7.4 Returning to Point in Program Where Fault Occurred 9-16
9.7.5 Program Return Point Other than Occurred Fault 9-17
9.7.6 Fault Controls 9-17
9.8 Fault Handling Action 9-18
9.8.1 Local Fault Call 9-18
9.8.2 System-Local Fault Call 9-18
9.8.3 System-Supervisor Fault Call 9-19
9.8.4 Faults and Interrupts 9-19

9.9 Precise and Imprecise Faults 9-20
9.9.1 Precise Faults 9-20
9.9.2 Imprecise Faults 9-20
9.9.3 Asynchronous Faults 9-20
9.9.4 No Imprecise Faults (AC.nif) Bit 9-21
9.9.5 Controlling Fault Precision 9-21
9.10 Fault Reference 9-22
9.10.1 ARITHMETIC Faults 9-23
9.10.2 CONSTRAINT Faults 9-24
9.10.3 OPERATION Faults 9-25
9.10.4 OVERRIDE Faults 9-26
9.10.5 PARALLEL Faults 9-27
9.10.6 PROTECTION Faults 9-28
9.10.7 TRACE Faults 9-29
9.10.8 TYPE Faults 9-30

10 Tracing and Debugging 10-1

10.1 Trace Controls 10-1
10.1.1 Trace Controls Register – TC 10-2
10.1.2 PC Trace Enable Bit and Trace-Fault-Pending Flag 10-3
10.2 Trace Modes 10-3
10.2.1 Instruction Trace 10-3
10.2.2 Branch Trace 10-4
10.2.3 Call Trace 10-4
10.2.4 Return Trace 10-4
10.2.5 Prereturn Trace 10-4
10.2.6 Supervisor Trace 10-5
10.2.7 Mark Trace 10-5
10.2.7.1 Software Breakpoints 10-5
10.2.7.2 Hardware Breakpoints 10-5
10.2.7.3 Requesting Modification Rights to Hardware Breakpoint Resources 10-6
10.2.7.4 Breakpoint Control Register – BPCON 10-7
10.2.7.5 Data Address Breakpoint Registers – DABx 10-8
10.2.7.6 Instruction Breakpoint Registers – IPBx 10-9
10.3 Generating a Trace Fault 10-10
10.4 Handling Multiple Trace Events 10-10
10.5 Trace Fault Handling Procedure 10-10
10.5.1 Tracing and Interrupt Procedures 10-11
10.5.2 Tracing on Calls and Returns 10-11
10.5.2.1 Tracing on Explicit Call 10-11
10.5.2.2 Tracing on Implicit Call 10-12
10.5.2.3 Tracing on Return from Explicit Call 10-13
10.5.2.4 Tracing on Return from Implicit Call: Fault Case 10-13
10.5.2.5 Tracing on Return from Implicit Call: Interrupt Case 10-13

11 Initialization and System Requirements 11-1

11.1 Overview 11-1
11.1.1 Core Initialization 11-1
11.1.2 General Initialization 11-2
11.2 Intel® 80303 I/O Processor Initialization 11-2
11.2.1 Initialization Modes 11-2

11.2.2 Mode 0 Initialization 11-3
11.2.3 Mode 1 Initialization 11-3
11.2.4 Mode 2 Initialization 11-3
11.2.5 Mode 3 (Default Mode) 11-3
11.2.6 Secondary PCI Bus Arbitration Unit 11-5
11.2.7 Internal Bus Arbitration Unit 11-5
11.2.8 Reset State Operation 11-5
11.3 Intel® i960® Core Processor Initialization 11-6
11.3.1 Self Test Function (**STEST**, **FAIL#**) 11-7
11.3.1.1 The **STEST** Signal 11-7
11.3.1.2 Intel® i960® Local Bus Confidence Test 11-7
11.3.1.3 The Fail Signal (**FAIL#**) 11-8
11.3.1.4 IMI Alignment Check and Core Processor Error 11-8
11.3.1.5 **FAIL#** Code 11-9
11.4 Initial Memory Image (IMI) 11-10
11.4.1 Initialization Boot Record (IBR) 11-12
11.4.2 Process Control Block – PRCB 11-15
11.4.3 Process PRCB Flow 11-17
11.4.3.1 AC Initial Image 11-18
11.4.3.2 Fault Configuration Word 11-18
11.4.3.3 Instruction Cache Configuration Word 11-18
11.4.3.4 Register Cache Configuration Word 11-18
11.4.4 Control Table 11-19
11.5 Device Identification on Reset 11-20
11.6 Reinitializing and Relocating Data Structures 11-21
11.6.1 Output Clocks 11-21

12 Core Processor and Internal Operation 12-1

12.1 Core Processor Memory Attributes 12-1
12.2 Physical Memory Attributes 12-2
12.2.1 PMCON Registers 12-2
12.2.2 Bus Control Register – BCON 12-4
12.3 Programming the Logical Memory Attributes 12-5
12.3.1 Logical Memory Attributes 12-5
12.3.2 Logical Memory Address Registers - LMADR0:1 12-6
12.3.3 Defining the Effective Range of a Logical Data Template 12-8
12.3.4 Data Caching Enable 12-8
12.3.5 Enabling the Logical Memory Template 12-8
12.3.6 Initialization 12-9
12.3.7 Boundary Conditions for Logical Memory Templates 12-9
12.3.7.1 Internal Memory Locations and Peripheral MMRs 12-9
12.3.7.2 Overlapping Logical Data Template Ranges 12-9
12.3.7.3 Accesses Across LMT Boundaries 12-9
12.3.8 Modifying the LMT Registers 12-9
12.4 Bus Interface Unit 12-10
12.4.1 Overview 12-10
12.4.2 Addressing 12-12
12.4.2.1 Bus Width 12-12
12.4.3 Multi-Transaction Timer 12-12
12.4.4 Features 12-12
12.4.4.1 Write Buffering 12-12

12.4.4.2 Instruction Fetch Bypass 12-13
12.4.4.3 Instruction Prefetch 12-13
12.4.4.4 Write Merging 12-13
12.4.4.5 Atomic Accesses 12-14
12.4.5 Interrupts and Error Conditions 12-15
12.4.5.1 Master-Abort 12-15
12.4.5.2 PCI Target-Abort 12-15
12.4.5.3 Internal Bus Target-Abort 12-16
12.4.6 Register Definitions 12-17
12.4.6.1 BIU Control Register - BIUCR 12-17
12.4.6.2 BIU Interrupt Status Register - BIUISR 12-18

13 Memory Controller 13-1

13.1 Overview 13-1
13.1.1 Glossary 13-2
13.2 Theory of Operation 13-3
13.2.1 Functional Blocks 13-3
13.2.1.1 Internal Bus Interface 13-4
13.2.1.2 Address Decode 13-4
13.2.1.3 Configuration Registers 13-4
13.2.1.4 SDRAM State Machine 13-4
13.2.1.5 Flash State Machine 13-4
13.2.1.6 Refresh Counter 13-5
13.2.1.7 Pipeline Queues and Error Correction Logic 13-5
13.2.2 Flash Memory Support 13-5
13.2.2.1 Flash Memory Addressing 13-6
13.2.2.2 Flash Read Cycle 13-7
13.2.2.3 Flash Write Cycle 13-10
13.2.3 SDRAM Memory Support 13-11
13.2.3.1 SDRAM Sizes and Configurations 13-13
13.2.3.2 SDRAM Addressing 13-15
13.2.3.3 Page Hit/Miss Determination 13-16
13.2.3.4 SDRAM Commands 13-18
13.2.3.5 SDRAM Initialization 13-19
13.2.3.6 SDRAM Mode Programming 13-20
13.2.3.7 SDRAM Read Cycle 13-21
13.2.3.8 SDRAM Write Cycle 13-24
13.2.3.9 SDRAM Refresh Cycle 13-27
13.2.4 Error Correction and Detection 13-28
13.2.4.1 ECC Generation 13-28
13.2.4.2 ECC Generation for Partial Writes 13-28
13.2.4.3 ECC Checking 13-30
13.2.4.4 Scrubbing 13-32
13.2.4.5 ECC Testing 13-34
13.2.5 Overlapping Memory Regions 13-34
13.2.6 SDRAM Clocking 13-35
13.3 Power Failure Mode 13-36
13.3.1 Theory of Operation 13-36
13.3.2 Power Failure Sequence 13-37
13.3.2.1 Power Failure Impact on the System 13-37
13.3.2.2 System Assumptions 13-38
13.3.3 Memory Controller Response to **I_RST#** 13-38
13.3.3.1 External Logic Required for Power Failure 13-40

13.4 Interrupts/Error Conditions 13-42
13.4.1 Single-Bit Error Detection 13-42
13.4.2 Double-Bit/Nibble Error Detection 13-43
13.5 Reset Conditions 13-43
13.6 Register Definitions 13-44
13.6.1 SDRAM Initialization Register - SDIR 13-45
13.6.2 SDRAM Control Register - SDCR 13-46
13.6.3 SDRAM Base Register - SDBR 13-47
13.6.4 SDRAM Boundary Register 0 - SBR0 13-48
13.6.5 SDRAM Boundary Registers 1 - SBR1 13-49
13.6.6 ECC Control Register - ECCR 13-50
13.6.7 ECC Log Registers - ELOG0, ELOG1 13-51
13.6.8 ECC Address Registers - ECAR0, ECAR1 13-52
13.6.9 ECC Test Register - ECTST 13-53
13.6.10 Flash Base Register 0 - FEBR0 13-54
13.6.11 Flash Base Register 1 - FEBR1 13-55
13.6.12 Flash Bank Size Register 0 - FBSR0 13-56
13.6.13 Flash Bank Size Register 1 - FBSR1 13-57
13.6.14 Flash Wait States Registers - FWSR0, FWSR1 13-58
13.6.15 Memory Controller Interrupt Status Register - MCISR 13-59
13.6.16 Refresh Frequency Register - RFR 13-60

14 PCI-to-PCI Bridge Unit 14-1

14.1 Overview 14-1
14.2 Theory of Operation 14-3
14.3 Architectural Description 14-4
14.3.1 Primary PCI Interface 14-4
14.3.2 Secondary PCI Interface 14-4
14.3.3 Upstream/Downstream Queues 14-5
14.3.4 Configuration Registers 14-6
14.4 Configuration Accesses 14-7
14.4.1 Type 0 Commands 14-9
14.4.2 Type 1 Commands and Type 1 to Type 0 Conversions 14-9
14.4.3 Type 1 to Type 1 Forwarding 14-11
14.4.4 Type 1 to Special Cycle Conversion 14-12
14.4.5 Private Type 0 Commands on the Secondary Interface 14-13
14.4.6 Special Cycles 14-15
14.4.7 Extended Configuration Space 14-15
14.5 Address Decoding 14-15
14.5.1 I/O Address Space 14-16
14.5.1.1 Disabling the I/O Address Range 14-17
14.5.1.2 ISA Mode 14-17
14.5.2 Memory Address Space 14-18
14.5.2.1 Burst Order 14-19
14.5.2.2 Disabling the Memory Address Range 14-19
14.5.3 VGA Address Support 14-20
14.5.3.1 VGA Compatible Addressing 14-20
14.5.4 64-Bit Address Decoding - Dual Address Cycles 14-21
14.5.5 Private Address Space 14-23
14.5.6 Secondary PCI to Messaging Unit Access 14-23

14.5.7 Address Decode Summary 14-24
14.6 Bridge Operation 14-27
14.6.1 PCI Interfaces 14-27
14.6.1.1 Primary Interface 14-27
14.6.1.2 Secondary Interface 14-28
14.6.2 Claiming a PCI Transaction 14-29
14.6.2.1 Latency Timers 14-29
14.6.2.2 Delayed Transactions 14-30
14.6.2.3 Posted Transactions 14-31
14.6.3 64-Bit Operation 14-31
14.6.3.1 64-Bit Protocol 14-32
14.6.3.2 64-Bit Operation with 32-Bit Targets 14-34
14.6.4 66 MHz Operation 14-36
14.6.5 PCI Read Transactions 14-37
14.6.5.1 Read Streaming 14-41
14.6.5.2 Read Boundary 14-41
14.6.6 PCI Write Transactions 14-42
14.6.6.1 Delayed Write Transactions 14-42
14.6.6.2 Posted Write Transactions 14-43
14.6.6.3 Memory Write Command 14-44
14.6.6.4 Memory Write and Invalidate Command 14-45
14.6.6.5 I/O Write Command 14-46
14.6.6.6 Write Boundary 14-46
14.6.6.7 Qword Unaligned Memory Write Transactions 14-46
14.6.6.8 Fast Back-to-Back Transactions 14-46
14.7 Queue Architecture 14-47
14.7.1 Queue Operation 14-48
14.7.1.1 Upstream/Downstream Posted Memory Write Queue Structures 14-49
14.7.1.2 Upstream/Downstream Delayed Read Completion Queues 14-50
14.7.1.3 Upstream/Downstream Delayed Write Completion Queue 14-51
14.7.1.4 Upstream/Downstream Transaction Queues 14-52
14.7.2 Transaction Ordering 14-52
14.8 Bridge Data Flow 14-55
14.8.1 Delayed Read Transaction 14-55
14.8.2 Delayed Write Transaction 14-57
14.8.3 Posted Write Transaction 14-59
14.9 Exclusive Access 14-61
14.9.1 Secondary Interface Error Handling 14-62
14.10 PCI Transaction Termination 14-63
14.10.1 Termination as a Master (Initiator) 14-63
14.10.1.1 Completion 14-63
14.10.1.2 Time-out 14-63
14.10.1.3 Time-out During Memory Write and Invalidate 14-63
14.10.1.4 Master-Abort 14-64
14.10.2 Termination as a Slave (Target) 14-64
14.10.2.1 Retry 14-64
14.10.2.2 Disconnect 14-65
14.10.2.3 Target-Abort 14-65
14.11 Error Conditions 14-66
14.11.1 Address Parity Errors 14-66
14.11.1.1 Address Parity Errors on Primary Interface 14-66
14.11.1.2 Address Parity Errors on Secondary Interface 14-67

14.11.2 Data Parity Errors 14-68
14.11.2.1 Read Data Parity 14-68
14.11.2.2 Delayed Write Data Parity 14-69
14.11.2.3 Posted Write Data Parity 14-71
14.11.3 **SERR#** Assertion 14-72
14.11.4 Discard Timers 14-73
14.11.5 PCI-to-PCI Bridge Error Summary 14-74
14.12 Initialization and Reset Requirements 14-79
14.12.1 Bridge Reset 14-79
14.12.2 Configuring the PCI-to-PCI Bridge 14-79
14.12.3 64-Bit Bus Configuration 14-80
14.13 Power-up/Default States 14-81
14.14 Performance Considerations 14-81
14.15 Register Definitions 14-82
14.15.1 Vendor Identification Register - VIDR 14-85
14.15.2 Device ID Register - DIDR 14-86
14.15.3 Primary Command Register - PCR 14-87
14.15.4 Primary Status Register - PSR 14-88
14.15.5 Revision ID Register - RID 14-90
14.15.6 Class Code Register - CCR 14-91
14.15.7 Cacheline Size Register - CLSR 14-92
14.15.8 Primary Latency Timer Register - PLTR 14-93
14.15.9 Header Type Register - HTR 14-94
14.15.10 Primary Bus Number Register - PBNR 14-95
14.15.11 Secondary Bus Number Register - SBNR 14-96
14.15.12 Subordinate Bus Number Register - SubBNR 14-97
14.15.13 Secondary Latency Timer Register - SLTR 14-98
14.15.14 I/O Base Register - IOBR 14-99
14.15.15 I/O Limit Register - IOLR 14-100
14.15.16 Secondary Status Register - SSR 14-101
14.15.17 Memory Base Register - MBR 14-102
14.15.18 Memory Limit Register - MLR 14-103
14.15.19 Prefetchable Memory Base Register - PMBR 14-104
14.15.20 Prefetchable Memory Limit Register - PMLR 14-105
14.15.21 Capabilities Pointer Register - Cap_Ptr 14-106
14.15.22 Bridge Control Register - BCR 14-107
14.15.23 Extended Bridge Control Register - EBCR 14-109
14.15.24 Secondary IDSEL Select Register - SISR 14-111
14.15.25 Primary Bridge Interrupt Status Register - PBISR 14-113
14.15.26 Secondary Bridge Interrupt Status Register - SBISR 14-114
14.15.27 Secondary Arbitration Control Register - SACR 14-115
14.15.28 PCI Interrupt Routing Select Register - PIRSR 14-115
14.15.29 Secondary I/O Base Register - SIOBR 14-116
14.15.30 Secondary I/O Limit Register - SIOLR 14-117
14.15.31 Secondary Clock Disable Register - SCDR 14-118
14.15.32 Secondary Memory Base Register - SMBR 14-119
14.15.33 Secondary Memory Limit Register - SMLR 14-120
14.15.34 Secondary Decode Enable Register - SDER 14-121
14.15.35 Queue Control Register - QCR 14-123
14.15.36 Capability Identifier Register - Cap_ID 14-124

14.15.37 Next Item Pointer Register - Next_Item_Ptr 14-125
14.15.38 Power Management Capabilities Register - PMCR 14-126
14.15.39 Power Management Control/Status Register - PMCSR 14-127
14.15.40 PMCSR PCI-to-PCI Bridge Support - PMCSR_BSE 14-128

15 PCI Address Translation Unit 15-1

15.1 Overview 15-1
15.2 ATU Address Translation 15-4
15.2.1 Inbound Transactions 15-5
15.2.1.1 Inbound Address Translation 15-6
15.2.1.2 Inbound Write Transaction 15-9
15.2.1.3 Inbound Read Transaction 15-11
15.2.1.4 Inbound Configuration Cycle Translation 15-13
15.2.1.5 Discard Timers 15-14
15.2.2 Outbound Transactions 15-14
15.2.2.1 Outbound Address Translation 15-15
15.2.2.2 Outbound Address Translation Windows 15-15
15.2.2.3 Direct Addressing Window 15-18
15.2.2.4 Outbound Write Transaction 15-19
15.2.2.5 Outbound Read Transaction 15-20
15.2.3 Private PCI Address Space / Outbound Configuration Cycle Translation 15-22
15.2.4 PCI Multi-Function Device Swapping/Disabling 15-23
15.2.5 64-Bit PCI Operation 15-24
15.2.5.1 64-Bit Protocol 15-24
15.2.5.2 64-Bit Operation with 32-Bit Targets 15-26
15.2.6 66 MHz Operation 15-28
15.3 Messaging Unit 15-29
15.4 Expansion ROM Translation Unit 15-30
15.5 ATU Queue Architecture 15-31
15.5.1 Inbound Queues 15-31
15.5.1.1 Inbound Write Queue Structure 15-32
15.5.1.2 Inbound Read Queues and Inbound Transaction Queues 15-33
15.5.1.3 Inbound Delayed Write Queue 15-34
15.5.2 Outbound Queues 15-34
15.5.3 Transaction Ordering 15-35
15.6 ATU Error Conditions 15-38
15.6.1 Address Parity Errors on the PCI Interface 15-39
15.6.2 Data Parity Errors on the PCI Interface 15-40
15.6.2.1 Outbound Read Data Parity Errors - Master 15-40
15.6.2.2 Outbound Write Data Parity Errors - Master 15-41
15.6.2.3 Inbound Read Data Parity Errors - Slave 15-41
15.6.2.4 Inbound Write Data Parity Errors - Slave 15-41
15.6.2.5 Inbound Configuration Write Data Parity Errors - Slave 15-42
15.6.3 Master Aborts on the PCI Interface 15-43
15.6.4 Target Aborts on the PCI Interface 15-44
15.6.5 SERR# Assertion and Detection 15-45
15.6.6 Internal Bus Error Conditions 15-47
15.6.6.1 Master Abort on the Internal Bus 15-47
15.6.6.2 Target Abort on the Internal Bus 15-49
15.6.7 ATU Error Summary 15-50
15.7 Register Definitions 15-54
15.7.1 ATU Vendor ID Register - ATUVID 15-60

15.7.2 ATU Device ID Register - ATUDID 15-61
15.7.3 Primary ATU Command Register - PATUCMD 15-62
15.7.4 Primary ATU Status Register - PATUSR 15-63
15.7.5 ATU Revision ID Register - ATURID 15-65
15.7.6 ATU Class Code Register - ATUCCR 15-66
15.7.7 ATU Cacheline Size Register - ATUCLSR 15-67
15.7.8 ATU Latency Timer Register - ATULT 15-68
15.7.9 ATU Header Type Register - ATUHTR 15-69
15.7.10 ATU BIST Register - ATUBISTR 15-70
15.7.11 Primary Inbound ATU Base Address Register - PIABAR 15-71
15.7.12 ATU Subsystem Vendor ID Register - ASVIR 15-72
15.7.13 ATU Subsystem ID Register - ASIR 15-73
15.7.14 Expansion ROM Base Address Register - ERBAR 15-74
15.7.15 ATU Capabilities Pointer Register - ATU_Cap_Ptr 15-75
15.7.16 Determining Block Sizes for Base Address Registers 15-76
15.7.17 ATU Interrupt Line Register - ATUILR 15-77
15.7.18 ATU Interrupt Pin Register - ATUIPR 15-78
15.7.19 ATU Minimum Grant Register - ATUMGNT 15-79
15.7.20 ATU Maximum Latency Register - ATUMLAT 15-80
15.7.21 Primary Inbound ATU Limit Register - PIALR 15-81
15.7.22 Primary Inbound ATU Translate Value Register - PIATVR 15-82
15.7.23 Secondary Inbound ATU Base Address Register - SIABAR 15-83
15.7.24 Secondary Inbound ATU Limit Register - SIALR 15-84
15.7.25 Secondary Inbound ATU Translate Value Register - SIATVR 15-85
15.7.26 Primary Outbound Memory Window Value Register - POMWVR 15-86
15.7.27 Primary Outbound I/O Window Value Register - POIOWVR 15-87
15.7.28 Primary Outbound DAC Window Value Register - PODWVR 15-88
15.7.29 Primary Outbound Upper 64-bit DAC Register - POUDR 15-89
15.7.30 Secondary Outbound Memory Window Value Register - SOMWVR 15-90
15.7.31 Secondary Outbound I/O Window Value Register - SOIOWVR 15-91
15.7.32 Expansion ROM Limit Register - ERLR 15-92
15.7.33 Expansion ROM Translate Value Register - ERTVR 15-93
15.7.34 ATU_Capability Identifier Register - ATU_Cap_ID 15-94
15.7.35 ATU Next Item Pointer Register - ATU_Next_Item_Ptr 15-95
15.7.36 ATU Power Management Capabilities Register - APMCR 15-96
15.7.37 ATU Power Management Control/Status Register - APMCSR 15-97
15.7.38 ATU Configuration Register - ATUCR 15-98
15.7.39 Primary ATU Interrupt Status Register - PATUISR 15-100
15.7.40 Secondary ATU Interrupt Status Register - SATUISR 15-102
15.7.41 Secondary ATU Command Register - SATUCMD 15-104
15.7.42 Secondary ATU Status Register - SATUSR 15-105
15.7.43 Secondary Outbound DAC Window Value Register - SODWVR 15-106
15.7.44 Secondary Outbound Upper 64-bit DAC Register - SOUDR 15-107
15.7.45 Primary Outbound Configuration Cycle Address Register - POCCAR 15-108
15.7.46 Secondary Outbound Configuration Cycle Address Register - SOCCAR 15-109
15.7.47 Primary Outbound Configuration Cycle Data Register - POCCDR 15-110
15.7.48 Secondary Outbound Configuration Cycle Data Register - SOCCDR 15-111
15.7.49 Primary ATU Queue Control Register - PAQCR 15-112
15.7.50 Secondary ATU Queue Control Register - SAQCR 15-113
15.7.51 Primary ATU Interrupt Mask Register - PATUIMR 15-114

15.7.52 Secondary ATU Interrupt Mask Register - SATUIMR 15-115

16 Messaging Unit 16-1

16.1 Overview 16-1
16.2 Theory of Operation 16-2
16.2.1 Transaction Ordering 16-4
16.3 Message Registers 16-5
16.3.1 Outbound Messages 16-5
16.3.2 Inbound Messages 16-5
16.4 Doorbell Registers 16-6
16.4.1 Outbound Doorbells 16-6
16.4.2 Inbound Doorbells 16-6
16.5 Circular Queues 16-7
16.5.1 Inbound Free Queue 16-11
16.5.2 Inbound Post Queue 16-12
16.5.3 Outbound Post Queue 16-13
16.5.4 Outbound Free Queue 16-14
16.6 Index Registers 16-15
16.7 Messaging Unit Error Conditions 16-15
16.8 Register Definitions 16-16
16.8.1 Inbound Message Register - IMRx 16-18
16.8.2 Outbound Message Register - OMRx 16-19
16.8.3 Inbound Doorbell Register - IDR 16-20
16.8.4 Inbound Interrupt Status Register - IISR 16-21
16.8.5 Inbound Interrupt Mask Register - IIMR 16-22
16.8.6 Outbound Doorbell Register - ODR 16-23
16.8.7 Outbound Interrupt Status Register - OISR 16-24
16.8.8 Outbound Interrupt Mask Register - OIMR 16-25
16.8.9 MU Configuration Register - MUCR 16-26
16.8.10 Queue Base Address Register - QBAR 16-27
16.8.11 Inbound Free Head Pointer Register - IFHPR 16-28
16.8.12 Inbound Free Tail Pointer Register - IFTPR 16-29
16.8.13 Inbound Post Head Pointer Register - IPHPR 16-30
16.8.14 Inbound Post Tail Pointer Register - IPTPR 16-31
16.8.15 Outbound Free Head Pointer Register - OFHPR 16-32
16.8.16 Outbound Free Tail Pointer Register - OFTPR 16-33
16.8.17 Outbound Post Head Pointer Register - OPHPR 16-34
16.8.18 Outbound Post Tail Pointer Register - OPTPR 16-35
16.8.19 Index Address Register - IAR 16-36
16.9 Power/Default Status 16-36

17 Intel® 80303 I/O Processor Arbitration 17-1

17.1 Arbitration Overview 17-1
17.2 PCI Arbiter Overview 17-2
17.2.1 Theory of Operation 17-3
17.2.1.1 Priority Mechanism 17-3
17.2.1.2 Priority Example with Three Bus Masters 17-4
17.2.1.3 Priority Example with Six Bus Masters 17-5
17.2.1.4 Arbitration Signalling Protocol 17-6
17.2.1.5 Secondary PCI Bus Arbitration Parking 17-8
17.2.2 Atomic Accesses 17-8

17.2.3 Internal and Secondary PCI Arbiter Differences 17-9
17.2.3.1 Multi-Transaction Timer 17-9
17.3 PCI Selector Operation 17-11
17.3.1 Primary PCI Bus Arbitration Parking 17-11
17.4 Master Latency Timer Operation 17-11
17.4.1 Primary and Secondary PCI Master Latency Timers 17-11
17.4.2 Internal Master Latency Timer 17-11
17.5 Reset Conditions 17-12
17.5.1 **S_REQ64#** Control 17-12
17.6 Register Definitions 17-13
17.6.1 Secondary Arbitration Control Register - SACR 17-14
17.6.2 Internal Arbitration Control Register - IACR 17-15
17.6.3 Master Latency Timer Register - MLTR 17-16
17.6.4 Multi-Transaction Timer Register - MTTR 17-17

18 Timers 18-1

18.1 Timer Registers 18-2
18.1.1 Timer Mode Registers – TMR0:1 18-3
18.1.1.1 Bit 0 - Terminal Count Status Bit (TMRx.tc) 18-3
18.1.1.2 Bit 1 - Timer Enable (TMRx.enable) 18-4
18.1.1.3 Bit 2 - Timer Auto Reload Enable (TMRx.reload) 18-4
18.1.1.4 Bit 3 - Timer Register Supervisor Read/Write Control (TMRx.sup) 18-5
18.1.1.5 Bits 4, 5 - Timer Input Clock Select (TMRx.csel1:0) 18-5
18.1.2 Timer Count Register – TCR0:1 18-6
18.1.3 Timer Reload Register – TRR0:1 18-7
18.2 Timer Operation 18-8
18.2.1 Basic Timer Operation 18-8
18.2.2 Load/Store Access Latency for Timer Registers 18-9
18.3 Timer Interrupts 18-10
18.4 Powerup/Reset Initialization 18-10
18.5 Uncommon TCRx and TRRx Conditions 18-10
18.6 Timer State Diagram 18-11

19 DMA Controller Unit 19-1

19.1 Overview 19-1
19.2 Theory of Operation 19-3
19.3 DMA Transfer 19-4
19.3.1 Chain Descriptors 19-5
19.3.2 Initiating DMA Transfers 19-7
19.3.3 Scatter Gather DMA Transfers 19-8
19.3.4 Synchronizing a Program to Chained Transfers 19-9
19.3.5 Appending to The End of a Chain 19-11
19.4 64-bit Transfers on a 64-bit PCI Bus 19-12
19.4.1 64-bit Operation with 64-bit Targets 19-12
19.4.2 64-bit Operation with 32-bit Targets 19-12
19.4.3 64-bit Addressing 19-13
19.4.4 66 MHz Operation 19-13
19.5 Data Transfers 19-14
19.5.1 PCI-to-Local Memory Transfers 19-14
19.5.2 Local Memory to PCI Transfers: Memory Write Command 19-14
19.5.3 Local Memory to PCI Transfers: Memory Write and Invalidate Command 19-15

19.5.4 Exclusive Access 19-15
19.6 Data Queues 19-16
19.7 Data Alignment 19-17
19.7.1 64-bit Unaligned Data Transfers 19-17
19.7.2 64/32-bit Unaligned Data Transfers 19-18
19.8 Channel Priority 19-19
19.9 Programming Model State Diagram 19-20
19.10 DMA Channel Programming Examples 19-21
19.10.1 Software DMA Controller Initialization 19-21
19.10.2 Software Start DMA Transfer 19-21
19.10.3 Software Suspend Channel 19-22
19.11 Interrupts 19-23
19.12 Error Conditions 19-24
19.12.1 PCI Errors 19-24
19.12.2 Internal Bus Errors 19-24
19.13 Power-up/Default Status 19-26
19.14 Register Definitions 19-26
19.14.1 Channel Control Register - CCR 19-27
19.14.2 Channel Status Register - CSR 19-28
19.14.3 Next Descriptor Address Register - NDAR 19-30
19.14.4 Descriptor Address Register - DAR 19-31
19.14.5 Byte Count Register - BCR 19-32
19.14.6 PCI Address Register - PADR 19-33
19.14.7 PCI Upper Address Register - PUADR 19-34
19.14.8 Intel® i960® Local Address Register - LADR 19-35
19.14.9 Descriptor Control Register - DCR 19-36
19.14.9.1 PCI Commands Support 19-37

20 Application Accelerator Unit 20-1

20.1 Overview 20-1
20.2 Theory of Operation 20-2
20.3 Hardware-Assist XOR Unit 20-3
20.3.1 Data Transfer 20-3
20.3.2 Chain Descriptor Format (Four Source Addresses) 20-4
20.3.3 Chain Descriptor Format (Eight Source Addresses) 20-7
20.3.4 The Bitwise-XOR Algorithm 20-10
20.3.5 Initiating the XOR Operation 20-13
20.3.6 Scatter Gather Transfers 20-14
20.3.7 Synchronizing a Program to Chained Operation 20-14
20.3.8 Appending to The End of a Chain 20-16
20.4 Packing and Unpacking 20-17
20.4.1 64-bit Unaligned Data Transfers 20-17
20.5 Application Accelerator Priority 20-19
20.6 Programming Model State Diagram 20-20
20.7 Programming the Application Accelerator 20-21
20.7.1 Application Accelerator Initialization 20-21
20.7.2 Start XOR Transfer 20-21
20.7.3 Suspend Application Accelerator 20-22
20.8 Interrupts 20-23
20.9 Error Conditions 20-24

20.10 Power-up/Default Status 20-25
20.11 Register Definitions 20-25
20.11.1 Accelerator Control Register - ACR 20-26
20.11.2 Accelerator Status Register - ASR 20-27
20.11.3 Accelerator Descriptor Address Register - ADAR 20-28
20.11.4 Accelerator Next Descriptor Address Register - ANDAR 20-29
20.11.5 80960 Source Address Register - SAR 20-30
20.11.6 80960 Destination Address Register - DAR 20-31
20.11.7 Accelerator Byte Count Register - ABCR 20-32
20.11.8 Accelerator Descriptor Control Register - ADCR 20-33

21 Performance Monitoring Unit 21-1

21.1 Overview 21-1
21.2 Theory of Operation 21-2
21.2.1 Global Time Stamp 21-2
21.2.2 Programmable Event Counters 21-2
21.2.2.1 Occurrence Events 21-3
21.2.2.2 Duration Events 21-4
21.2.3 Performance Monitoring 21-5
21.3 Event Description 21-6
21.3.1 Mode0: Performance Monitoring Disabled 21-6
21.3.2 Mode1: Primary PCI bus and Internal Agents 21-6
21.3.2.1 M1_PPCIBus_idle 21-6
21.3.2.2 M1_PPCIBus_busy 21-6
21.3.2.3 M1_bridge_acq 21-7
21.3.2.4 M1_bridge_own 21-7
21.3.2.5 M1_DMA0_acq 21-7
21.3.2.6 M1_DMA0_own 21-7
21.3.2.7 M1_DMA1_acq 21-7
21.3.2.8 M1_DMA1_own 21-7
21.3.2.9 M1_PATU_acq 21-7
21.3.2.10 M1_PATU_own 21-8
21.3.2.11 M1_DMA0_gnt 21-8
21.3.2.12 M1_DMA1_gnt 21-8
21.3.2.13 M1_PATU_gnt 21-8
21.3.2.14 M1_bridge_gnt 21-8
21.3.3 Mode 2: Secondary PCI Bus and Internal Agents 21-9
21.3.3.1 M2_SPCIBus_idle 21-9
21.3.3.2 M2_SPCIBus_busy 21-9
21.3.3.3 M2_SATU_acq 21-9
21.3.3.4 M2_SATU_own 21-9
21.3.3.5 M2_bridge_acq 21-9
21.3.3.6 M2_bridge_own 21-9
21.3.3.7 M2_DMA2_acq 21-10
21.3.3.8 M2_DMA2_own 21-10
21.3.3.9 M2_bridge_gnt 21-10
21.3.3.10 M2_SATU_gnt 21-10
21.3.3.11 M2_DMA2_gnt 21-10
21.3.3.12 M2_PPCIBus_idle 21-10
21.3.3.13 M2_PPCIBus_busy 21-10
21.3.3.14 M2_IBus_busy 21-10
21.3.4 Mode 3: Secondary PCI Bus and External Agents 21-11

21.3.4.1 M3_SPCIbus_idle 21-11
21.3.4.2 M3_SPCIbus_busy 21-11
21.3.4.3 M3_SPCI_IOP_acq 21-11
21.3.4.4 M3_SPCI_IOP_own 21-11
21.3.4.5 M3_D0_acq 21-11
21.3.4.6 M3_D0_own 21-12
21.3.4.7 M3_D1_acq 21-12
21.3.4.8 M3_D1_own 21-12
21.3.4.9 M3_D2_acq 21-12
21.3.4.10 M3_D2_own 21-12
21.3.4.11 M3_SPCI_IOP_gnt 21-12
21.3.4.12 M3_D0_gnt 21-12
21.3.4.13 M3_D1_gnt 21-13
21.3.4.14 M3_D2_gnt 21-13
21.3.5 Mode 4: Secondary PCI Bus and External Agents 21-13
21.3.5.1 M4_SPCIbus_idle 21-13
21.3.5.2 M4_SPCIbus_busy 21-13
21.3.5.3 M4_D3_acq 21-13
21.3.5.4 M4_D3_own 21-14
21.3.5.5 M4_D4_acq 21-14
21.3.5.6 M4_D4_own 21-14
21.3.5.7 M4_D5_acq 21-14
21.3.5.8 M4_D5_own 21-14
21.3.5.9 M4_D3_gnt 21-14
21.3.5.10 M4_D4_gnt 21-14
21.3.5.11 M4_D5_gnt 21-14
21.3.5.12 M4_SPCI_IOP_gnt 21-15
21.3.5.13 M4_SPCI_IOP_acq 21-15
21.3.5.14 M4_SPCI_IOP_own 21-15
21.3.6 Mode 5: Intel® 80303 I/O Processor Bus and Agents Events 21-15
21.3.6.1 M5_IBus_idle 21-15
21.3.6.2 M5_IBus_busy 21-15
21.3.6.3 M5_AA_acq 21-15
21.3.6.4 M5_AA_own 21-16
21.3.6.5 M5_DMA0_acq 21-16
21.3.6.6 M5_DMA0_own 21-16
21.3.6.7 M5_DMA1_acq 21-16
21.3.6.8 M5_DMA1_own 21-16
21.3.6.9 M5_DMA2_acq 21-16
21.3.6.10 M5_DMA2_own 21-16
21.3.6.11 M5_AA_gnt 21-17
21.3.6.12 M5_DMA0_gnt 21-17
21.3.6.13 M5_DMA1_gnt 21-17
21.3.6.14 M5_DMA2_gnt 21-17
21.3.7 Mode 6: Intel® 80303 I/O Processor Bus and Agents Events 21-17
21.3.7.1 M6_core_acq 21-17
21.3.7.2 M6_core_own 21-17
21.3.7.3 M6_PATU_acq 21-18
21.3.7.4 M6_PATU_own 21-18
21.3.7.5 M6_SATU_acq 21-18
21.3.7.6 M6_SATU_own 21-18
21.3.7.7 M6_PBOFF_time 21-18
21.3.7.8 M6_PBOFF_cnt 21-18
21.3.7.9 M6_SBOFF_time 21-18

21.3.7.10 M6_SBOFF_cnt 21-19
21.3.7.11 M6_PATU_gnt 21-19
21.3.7.12 M6_SATU_gnt 21-19
21.3.7.13 M6_core_gnt 21-19
0.0.0.1. M6_ATU_retry 21-19
21.3.8 Mode 7: Intel® 80303 I/O Processor Internal Bus, Secondary PCI Bus and Primary PCI Bus Events 21-19
21.3.8.1 M7_IBus_idle 21-19
21.3.8.2 M7_IBus_busy 21-20
21.3.8.3 M7_SPCIbus_idle 21-20
21.3.8.4 M7_SPCIbus_busy 21-20
21.3.8.5 M7_SPCI_IOP_own 21-20
21.3.8.6 M7_D0_own 21-20
21.3.8.7 M7_D1_own 21-20
21.3.8.8 M7_D2_own 21-20
21.3.8.9 M7_D3_own 21-20
21.3.8.10 M7_D4_own 21-20
21.3.8.11 M7_D5_own 21-21
21.3.8.12 M7_PPCI_IOP_own 21-21
21.3.8.13 M7_PPCIbus_idle 21-21
21.3.8.14 M7_PPCIbus_busy 21-21
21.4 Interrupts 21-21
21.5 Reset Conditions 21-21
21.6 Register Definitions 21-22
21.6.1 Global Timer Mode Register (GTMR) 21-23
21.6.2 Event Select Register (ESR) 21-24
21.6.3 Event Monitoring Interrupt Status Register (EMISR) 21-25
21.6.4 Global Time Stamp Register (GTSR) 21-27
21.6.5 Programmable Event Counter Register (PECRx) 21-27

22 I^2C Bus Interface Unit 22-1

22.1 Overview 22-1
22.2 Theory of Operation 22-2
22.2.1 Operational Blocks 22-3
22.2.2 I^2C Bus Interface Modes 22-4
22.2.3 Start and Stop Bus States 22-5
22.2.3.1 START Condition 22-6
22.2.3.2 No START or STOP Condition 22-6
22.2.3.3 STOP Condition 22-7
22.3 I^2C Bus Operation 22-8
22.3.1 Serial Clock Line (**SCL**) Generation 22-8
22.3.2 Data and Addressing Management 22-9
22.3.2.1 Addressing a Slave Device 22-10
22.3.3 I^2C Acknowledge 22-11
22.3.4 Arbitration 22-12
22.3.4.1 **SCL** Arbitration 22-12
22.3.4.2 **SDA** Arbitration 22-13
22.3.5 Master Operations 22-14
22.3.6 Slave Operations 22-18
22.3.7 General Call Address 22-20
22.4 Slave Mode Programming Examples 22-22
22.4.1 Initialize Unit 22-22

22.4.2 Write 1 Byte as a Slave 22-22
22.4.3 Read 2 Bytes as a Slave 22-22
22.5 Master Programming Examples 22-23
22.5.1 Initialize Unit 22-23
22.5.2 Write 1 Byte as a Master 22-23
22.5.3 Read 1 Byte as a Master 22-23
22.5.4 Write 2 Bytes and Repeated Start Read 1 Byte as a Master 22-24
22.5.5 Read 2 Bytes as a Master - Send STOP Using the Abort 22-24
22.6 Glitch Suppression Logic 22-26
22.7 Reset Conditions 22-27
22.8 Register Definitions 22-28
22.8.1 I^2C Control Register- ICR 22-29
22.8.2 I^2C Status Register- ISR 22-32
22.8.3 I^2C Slave Address Register- ISAR 22-34
22.8.4 I^2C Data Buffer Register- IDBR 22-35
22.8.5 I^2C Clock Count Register- ICCR 22-36
22.8.6 I^2C Bus Monitor Register- IBMR 22-37

23 General Purpose Input Output (GPIO) 23-1

23.1 General Purpose Input Output Support 23-1
23.1.1 General Purpose Inputs 23-1
23.1.2 General Purpose Outputs 23-1
23.1.3 Reset Initialization of General Purpose Input Output Function 23-1
23.2 Register Definitions 23-2
23.2.1 GPIO Output Enable Register - GPOE 23-2
23.2.2 GPIO Input Data Register - GPID 23-4
23.2.3 GPIO Output Data Register - GPOD 23-5

24 Test Features 24-1

24.1 On-Circuit Emulation (ONCE) 24-1
24.1.1 Entering/Exiting ONCE Mode 24-1
24.1.2 ONCE Mode and Boundary-Scan (JTAG) are Incompatible 24-2
24.1.2.1 **DEN#** Alternatives 24-2
24.2 Boundary-Scan (JTAG) 24-2
24.2.1 Boundary-Scan Architecture 24-3
24.2.2 TAP Pins 24-4
24.2.3 Instruction Register 24-4
24.2.3.1 Boundary-Scan Instruction Set 24-5
24.2.4 TAP Test Data Registers 24-7
24.2.4.1 Device Identification Register 24-7
24.2.4.2 Bypass Register 24-7
24.2.4.3 RUNBIST Register 24-7
24.2.4.4 Boundary-Scan Register 24-8
24.2.5 TAP Controller 24-17
24.2.5.1 Test Logic Reset State 24-18
24.2.5.2 Run-Test/Idle State 24-18
24.2.5.3 Select-DR-Scan State 24-18
24.2.5.4 Capture-DR State 24-18
24.2.5.5 Shift-DR State 24-18
24.2.5.6 Exit1-DR State 24-19
24.2.5.7 Pause-DR State 24-19

24.2.5.8 Exit2-DR State 24-19
24.2.5.9 Update-DR State 24-19
24.2.5.10 Select-IR Scan State 24-19
24.2.5.11 Capture-IR State 24-20
24.2.5.12 Shift-IR State 24-20
24.2.5.13 Exit1-IR State 24-20
24.2.5.14 Pause-IR State 24-20
24.2.5.15 Exit2-IR State 24-20
24.2.5.16 Update-IR State 24-20
24.2.6 Boundary-Scan Example 24-21

25 Clocking and Reset 25-1

25.1 Clocking Overview 25-1
25.1.1 Clocking Theory of Operation 25-1
25.1.2 Clocking Region 1 25-2
25.1.3 Clocking Region 2 25-2
25.1.4 Clocking Region 3 25-3
25.1.5 Clocking Region Summary 25-3
25.2 Reset Overview 25-4
25.2.1 Primary PCI Reset 25-5
25.2.2 Secondary PCI Reset 25-6
25.2.3 Internal Bus Reset 25-6
25.3 Reset Strapping Options 25-7

A Machine-Level Instruction Formats A-1

A.1 General Instruction Format A-1
A.2 REG Format A-3
A.3 COBR Format A-4
A.4 CTRL Format A-4
A.5 MEM Format A-5
A.5.1 MEMA Format Addressing A-6
A.5.2 MEMB Format Addressing A-6

B Opcodes and Execution Times B-1

B.1 Instruction Reference by Opcode B-1

C Peripheral Memory-Mapped Registers C-1

C.1 Overview C-1
C.2 Accessing the Peripheral Memory-Mapped Registers C-2
C.3 Architecturally Reserved Memory Space C-3
C.4 Peripheral Memory-Mapped Register Address Space C-5

Figures

1-1 Intel® 80303 I/O Processor Functional Block Diagram 1-1
2-1 Data Types and Ranges 2-1
3-1 Intel® 80303 I/O Processor Programming Environment 3-2
3-2 Local Memory Address Space 3-10
3-3 Arithmetic Controls Register – AC 3-14
4-1 Internal Data RAM and Register Cache 4-1
5-1 Machine-Level Instruction Formats 5-2
6-1 dcctl src1 and src/dst Formats 6-34
6-2 Store Data Cache to Memory Output Format 6-35
6-3 D-Cache Tag and Valid Bit Formats 6-35
6-4 icctl src1 and src/dst Formats 6-48
6-5 Store Instruction Cache to Memory Output Format 6-49
6-6 I-Cache Set Data, Tag and Valid Bit Formats 6-50
6-7 Src1 Operand Interpretation 6-95
6-8 src/dst Interpretation for Breakpoint Resource Request 6-96
7-1 Procedure Stack Structure and Local Registers 7-3
7-2 Frame Spill 7-9
7-3 Frame Fill 7-10
7-4 System Procedure Table 7-16
7-5 Previous Frame Pointer Register – PFP 7-19
8-1 Interrupt Handling Data Structures 8-1
8-2 Interrupt Table 8-4
8-3 Storage of an Interrupt Record on the Interrupt Stack 8-6
8-4 Interrupt Controller 8-13
8-5 Interrupt Pin Vector Assignment 8-14
8-6 Interrupt Fast Sampling 8-15
8-7 Interrupt Controller Connections 8-22
8-8 Interrupt Service Flowchart 8-29
9-1 Fault-Handling Data Structures 9-1
9-2 Fault Table and Fault Table Entries 9-5
9-3 Fault Record 9-9
9-4 Storage of the Fault Record on the Stack 9-10
11-1 Initialization Flow Chart 11-4
11-2 Processor Initialization Flow 11-6
11-3 FAIL# Timing 11-8
11-4 Initial Memory Image (IMI) and Process Control Block (PRCB) 11-11
11-5 Control Table 11-19
12-1 LMCON Example 12-5
12-2 Core Processor/BIU Interface Block Diagram 12-10
12-3 Internal Block Diagram 12-11
13-1 Memory Controller Block Diagram 13-3
13-2 Four Mbyte Flash Memory System 13-6
13-3 90 ns Flash Read Cycle 13-8
13-4 60 ns Flash Burst Read Cycle 13-9
13-5 90ns Flash Write Cycle 13-10
13-6 Dual-Bank SDRAM Memory Subsystem 13-12
13-7 Logical Memory Image of a 64/128/256 Mbit SDRAM Memory Subsystem 13-17
13-8 Supported SDRAM Mode Register Settings 13-19

13-9 SDRAM Initialization Sequence (controlled with software) 13-20
13-10 SDRAM Read, 40 bytes, ECC Enabled, BL=4, Page Hit 13-21
13-11 SDRAM Read, 40 bytes, ECC Enabled, BL=4, Page Miss 13-23
13-12 SDRAM Write, 40 bytes, ECC Enabled, BL=4, Page Hit 13-24
13-13 SDRAM Write, 40 bytes, ECC Enabled, BL=4, Page Miss 13-25
13-14 Refresh Following a Read Cycle 13-27
13-15 Sub 64-bit SDRAM Write (D_1) 13-29
13-16 H-Matrix 13-31
13-17 Scubbing Routine Flow Chart 13-33
13-18 SDRAM Clocking 13-35
13-19 Power Failure Sequence 13-37
13-20 Power Failure State Machine 13-38
13-21 Power Failure Sequence 13-39
13-22 External Power Failure State Machine 13-40
13-23 External Power Failure Logic in the System 13-41
14-1 PCI-to-PCI Bridge Unit Functional Block Diagram 14-2
14-2 Bridge Operation 14-3
14-3 PCI Configuration Access Formats 14-7
14-4 Secondary IDSEL Example 14-14
14-5 ISA Mode Address Decode 14-17
14-6 Overlapping Memory Address Ranges 14-18
14-7 VGA Compatible Addressing 14-20
14-8 64-bit Dual Address Read Cycle 14-22
14-9 PCI 64-Bit Transfer to a 64-Bit Target 14-33
14-10 64-Bit Write Request with 32-Bit Transfer 14-35
14-11 Bridge Configuration Header Format 14-83
14-12 Extended Bridge Configuration Header Format 14-83
15-1 ATU Block Diagram 15-2
15-2 ATU Queue Architecture Block Diagram 15-3
15-3 Inbound Address Detection 15-7
15-4 Inbound Translation Example 15-8
15-5 Intel® i960® Memory Map - Outbound Translation Window 15-16
15-6 Outbound Address Translation Windows 15-17
15-7 Direct Addressing Window 15-18
15-8 PCI 64-Bit Transfer from a 64-Bit Target 15-25
15-9 64-Bit Write Request with 32-Bit Transfer 15-27
15-10 ATU Interface Configuration Header Format 15-54
15-11 ATU Interface Extended Configuration Header Format 15-55
16-1 PCI Memory Map 16-3
16-2 Overview of Circular Queue Operation 16-8
16-3 Circular Queue Operation 16-10
17-1 Intel® 80303 I/O Processor Arbitration Block Diagram 17-1
17-2 Secondary PCI Arbitration Example 17-3
17-3 Arbitration Between Two Masters 17-6
17-4 BIU Back-to-Back Transactions with MTT enabled 17-9
18-1 Timer Functional Diagram 18-1
18-2 Timer Unit State Diagram 18-11
19-1 DMA Controller 19-2
19-2 DMA Channel Block Diagram 19-3
19-3 DMA Chain Descriptor 19-5

19-4 DMA Chaining Operation 19-6
19-5 Example of Gather Chaining 19-8
19-6 Synchronizing to Chained Transfers 19-10
19-7 Optimization of an Unaligned DMA 19-17
19-8 Optimization of an Unaligned DMA 19-18
19-9 DMA Programming Model State Diagram 19-20
19-10 Software Example for Channel Initialization 19-21
19-11 Software Example for DMA Transfer 19-21
19-12 Software Example for Channel Suspend 19-22
20-1 Application Accelerator Unit 20-1
20-2 Application Accelerator Block Diagram 20-2
20-3 Chain Descriptor Format 20-4
20-4 XOR Chaining Operation 20-6
20-5 Chain Descriptor Format for Eight Source Addresses (XOR Function) 20-7
20-6 XOR Chaining Operation 20-9
20-7 The Bit-wise XOR Algorithm 20-10
20-8 Hardware Assist XOR Unit 20-11
20-9 Example of Gather Chaining for Four Source Blocks 20-13
20-10 Synchronizing to Chained XOR Operation 20-15
20-11 Optimization of an Unaligned Data Transfer 20-18
20-12 Application Accelerator Programming Model State Diagram 20-20
20-13 Pseudo Code: Application Accelerator Initialization 20-21
20-14 Pseudo Code: XOR Transfer Operation 20-21
20-15 Pseudo Code: Suspend Application Accelerator 20-22
22-1 I^2C Bus Configuration Example 22-2
22-2 I^2C Bus Interface Unit Block Diagram 22-3
22-3 Start and Stop Conditions 22-6
22-4 START and STOP Conditions 22-7
22-5 Data Format of First Byte in Master Transaction 22-10
22-6 Acknowledge on the I^2C Bus 22-11
22-7 Clock Synchronization During the Arbitration Procedure 22-12
22-8 Arbitration Procedure of Two Masters 22-13
22-9 Master-Receiver Read from Slave-Transmitter 22-17
22-10 Master-Receiver Read from Slave-Transmitter / Repeated Start / Master-Transmitter Write to Slave-Receiver 22-17
22-11 A Complete Data Transfer 22-17
22-12 Master-Transmitter Write to Slave-Receiver 22-19
22-13 Master-Receiver Read to Slave-Transmitter 22-19
22-14 Master-Receiver Read to Slave-Transmitter, Repeated START, Master-Transmitter Write to Slave-Receiver 22-19
22-15 General Call Address 22-20
24-1 Test Access Port Block Diagram 24-3
24-2 TAP Controller State Diagram 24-17
24-3 Example Showing Typical JTAG Operations 24-22
24-4 Timing Diagram Illustrating the Loading of Instruction Register 24-23
24-5 Timing Diagram Illustrating the Loading of Data Register 24-24
25-1 Clocking Regions Diagram 25-1
25-2 SDRAM Clocking Diagram 25-2
25-3 Reset Block Diagram 25-4
A-1 Instruction Formats A-1
C-2 Intel® 80303 I/O Processor Address Space C-4

Tables

2-1 Intel® 80303 I/O Processor and PCI Architecture Data Word Notation Differences 2-2
2-2 Memory Addressing Modes 2-4
3-1 Registers and Literals Used as Instruction Operands 3-3
3-2 Allowable Register Operands 3-5
3-3 Data Structure Descriptions 3-9
3-4 Alignment of Data Structures in the Address Space 3-12
3-5 Condition Codes for True or False Conditions 3-15
3-6 Condition Codes for Equality and Inequality Conditions 3-15
3-7 Condition Codes for Carry Out and Overflow 3-15
3-8 Process Controls Register – PC 3-16
4-1 Load Instruction Updates 4-6
5-1 Instruction Encoding Formats (REG, COBR, CRTL, MEM) 5-2
5-2 Operands and Instruction Formats 5-3
5-3 Intel® 80303 I/O Processor Instruction Set 5-4
5-4 Load and Store Instructions 5-5
5-5 Move Instructions 5-6
5-6 Select Condition Instructions 5-6
5-7 Arithmetic Operations 5-7
5-8 Add, Subtract, Multiply, Divide, Conditional Add, Conditional Subtract Instructions 5-7
5-9 Remainder and Modulo Instructions 5-8
5-10 Shift, Rotate and Extended Shift Instructions 5-8
5-11 Extended Arithmetic Instructions 5-9
5-12 Logical Instructions 5-9
5-13 Bit Operations Instructions 5-10
5-14 ompare and Conditional Compare Instructions 5-11
5-15 Compare and Increment or Decrement Instructions 5-12
5-16 Test Condition Code Instructions 5-12
5-17 Unconditional Branch Instructions 5-13
5-18 Conditional Branch Instructions 5-13
5-19 Compare and Branch Instructions 5-14
5-20 Call/Return Instructions 5-15
5-21 Faults Instructions 5-16
5-22 Debug Instructions 5-16
5-23 Processor Management Instructions 5-17
5-24 Multi-Cycle Register Operations Microcode Instructions 5-19
5-25 Simple Control Transfer Instructions 5-19
5-26 Memory Instructions 5-20
5-27 Memory Instructions 5-20
5-28 Cache Control Instructions 5-21
6-1 Pseudo-Code Symbol Definitions 6-4
6-2 Faults Applicable to All Instructions 6-4
6-3 Common Faulting Conditions 6-4
6-4 ADD<cc> 6-6
6-5 addc 6-9
6-6 addi, addo 6-10
6-7 alterbit 6-11
6-8 and, andnot 6-12
6-9 atadd 6-13

6-10 atmod 6-14
6-11 b, bx 6-15
6-12 bal, balx 6-16
6-13 bbc, bbs 6-17
6-14 BRANCH<cc> 6-18
6-15 bswap 6-20
6-16 call 6-21
6-17 calls 6-22
6-18 callx 6-24
6-19 chkbit 6-25
6-20 clrbit 6-26
6-21 cmpdeci, cmpdeco 6-27
6-22 cmpinci, cmpinco 6-28
6-23 COMPARE 6-29
6-24 COMPARE AND BRANCH<cc> 6-30
6-25 concmpi, concmpo 6-32
6-26 dcctl 6-33
6-27 dcctl Status Values and D-Cache Parameters 6-34
6-28 divi, divo 6-38
6-29 ediv 6-39
6-30 emul 6-40
6-31 eshro 6-41
6-32 extract 6-42
6-33 FAULT<cc> 6-43
6-34 flushreg 6-44
6-35 fmark 6-45
6-36 halt 6-46
6-37 icctl 6-47
6-38 icctl Operand Fields 6-47
6-39 icctl Status Values and I-Cache Parameters 6-49
6-40 intctl 6-53
6-41 intdis 6-54
6-42 inten 6-55
6-43 LOAD 6-56
6-44 lda 6-59
6-45 mark 6-60
6-46 modac 6-61
6-47 modi 6-62
6-48 modify 6-63
6-49 modpc 6-64
6-50 modtc 6-65
6-51 MOVE 6-66
6-52 muli, mulo 6-68
6-53 hand 6-69
6-54 nor 6-70
6-55 not, notand 6-71
6-56 notbit 6-72
6-57 notor 6-73
6-58 or, ornot 6-74
6-59 remi, remo 6-75

6-60 ret 6-76
6-61 rotate 6-78
6-62 scanbit 6-79
6-63 scanbyte 6-80
6-64 SEL<cc> 6-81
6-65 setbit 6-82
6-66 SHIFT 6-83
6-67 spanbit 6-85
6-68 STORE 6-86
6-69 subc 6-90
6-70 SUB<cc> 6-91
6-71 subi, subo 6-93
6-72 syncf 6-94
6-73 sysctl 6-95
6-74 sysctl Field Definitions 6-95
6-75 Cache Mode Configuration 6-95
6-76 TEST<cc> 6-99
6-77 xnor, xor 6-100
7-1 Encodings of Entry Type Field in System Procedure Table 7-16
7-2 Encoding of Return Status Field 7-20
8-1 PCI Interrupt Routing Summary 8-23
8-2 Interrupt Input Pin Descriptions 8-23
8-3 XINT6# Interrupt Sources 8-24
8-4 XINT7# Interrupt Sources 8-25
8-5 NMI# Interrupt Sources 8-26
8-6 Default Interrupt Routing and Status Values 8-28
8-7 Location of Cached Vectors in Internal RAM 8-31
8-8 Base Interrupt Latency 8-32
8-9 Worst-Case Interrupt Latency Controlled by divo to Destination r15 8-33
8-10 Worst-Case Interrupt Latency Controlled by divo to Destination r3 8-33
8-11 Worst-Case Interrupt Latency Controlled by Calls 8-33
8-12 Worst-Case Interrupt Latency When Delivering a Software Interrupt 8-34
8-13 Worst-Case Interrupt Latency Controlled by flushreg of One Stack Frame 8-34
8-14 Interrupt Control Registers Addresses 8-35
8-15 Interrupt Control (ICON) Register 8-36
8-16 Interrupt Map Register 0 (IMAP0) 8-37
8-17 Interrupt Map Register 1 (IMAP1) 8-38
8-18 Interrupt Map Register 2 (IMAP2) 8-38
8-19 Interrupt Pending (IPND) Register 8-40
8-20 Interrupt Mask (IMSK) Register 8-40
8-21 PCI Interrupt Routing Select Register - PIRSR 8-41
8-22 XINT6 Interrupt Status Register- X6ISR 8-42
8-23 XINT7 Interrupt Status Register- X7ISR 8-43
8-24 NMI Interrupt Status Register- NISR 8-44
9-1 Fault Types and Subtypes 9-3
9-2 Fault Control Bits and Masks 9-17
10-1 Intel® 80303 I/O Processor Trace Controls Register – TC 10-2
10-2 src/dst Encoding 10-6
10-3 Breakpoint Control Register – BPCON 10-7
10-4 Configuring Data Address Breakpoint Registers – DABx 10-7

10-5 Programming Data Address Breakpoint Modes – DABx 10-7
10-6 Data Address Breakpoint Register – DABx 10-8
10-7 Instruction Breakpoint Register – IPBx 10-9
10-8 Instruction Breakpoint Modes 10-9
10-9 Tracing on Explicit Call 10-11
10-10 Tracing on Implicit Call 10-12
10-11 Tracing on Return from Explicit Call 10-13
11-1 Initialization Modes 11-2
11-2 Reset Values 11-5
11-3 BIST Failure Codes 11-9
11-4 Non-BIST Failure Codes 11-9
11-5 Initialization Boot Record 11-12
11-6 PMCON14_15 Register Bit Description in IBR 11-14
11-7 PRCB Configuration 11-15
11-8 Process Control Block Configuration Words 11-16
11-9 Processor Device ID Register - PDIDR 11-20
11-10 Intel® i960® Core Processor Device ID Register - DEVICEID 11-20
12-1 PMCON Address Mapping 12-2
12-2 Physical Memory Control Registers – PMCON0:15 12-3
12-3 Bus Control Register – BCON 12-4
12-4 Logical Memory Address Registers – LMADR0:1 12-6
12-5 Logical Memory Mask Registers – LMMR0:1 12-6
12-6 Default Logical Memory Configuration Register – DLMCON 12-7
12-7 Bus Interface Unit Register Table 12-17
12-8 BIU Control Register - BIUCR 12-17
12-9 BIU Interrupt Status Register - BIUISR 12-18
13-1 Commonly Used Terms 13-2
13-2 Flash Interface Signals 13-5
13-3 Address Decoding for Flash Memory Space 13-7
13-4 Flash Wait State Profile Programming 13-9
13-5 SDRAM Interface Signals 13-11
13-6 Supported SDRAM Configurations 13-13
13-7 SDRAM Address Register Definitions 13-13
13-8 Address Decoding for SDRAM Memory Space 13-14
13-9 Programming Values for the SDRAM Boundary Registers (SBRx[7:3]) 13-14
13-10 SDRAM Address Translation for 64/128/256 Mbit Devices using SA[12:0] 13-15
13-11 SDRAM Address Translation for 256 Mbit Devices 13-15
13-12 SDRAM Commands 13-18
13-13 Syndrome Decoding 13-30
13-14 Overlapping Address Priorities 13-34
13-15 MCU Error Response 13-42
13-16 Memory Controller Register Table 13-44
13-17 SDRAM Initialization Register - SDIR 13-45
13-18 SDRAM Control Register - SDCR 13-46
13-19 Drive Strength Programmability Options 13-47
13-20 SDRAM Base Register - SDBR 13-47
13-21 SDRAM Boundary Register 0 - SBR0 13-48
13-22 SDRAM Boundary Registers - SBR1 13-49
13-23 ECC Control Register - ECCR 13-50
13-24 ECC Log Registers - ELOG0, ELOG1 13-51

13-25 ECC Address Registers - ECAR0, ECAR1 13-52
13-26 ECC Test Register - ECTST 13-53
13-27 Flash Base Register 0 - FEBR0 13-54
13-28 Flash Base Register 1 - FEBR1 13-55
13-29 Flash Bank Size Register 0 - FBSR0 13-56
13-30 Flash Bank Size Register 1 - FBSR1 13-57
13-31 Flash Wait State Registers - FWSR0, FWSR1 13-58
13-32 Memory Controller Interrupt Status Register - MCISR 13-59
13-33 Refresh Frequency Register - RFR 13-60
14-1 PCI Configuration Command Access Formats 14-7
14-2 Bridge Configuration Cycle Handling Summary 14-8
14-3 IDSEL Mapping for Type 1 to Type 0 Conversions 14-10
14-4 Public/Private PCI Memory IDSEL Select Configurations 14-13
14-5 Primary to Secondary Memory Address Decoding Summary 14-24
14-6 Primary to Secondary I/O Address Decoding Summary 14-24
14-7 Secondary to Primary Memory Address Decoding Summary 14-25
14-8 Secondary to Primary I/O Address Decoding Summary 14-26
14-9 PCI Commands 14-27
14-10 Delayed Transactions vs. Posted Transactions 14-31
14-11 Prefetchable and Non-Prefetchable Memory Summary 14-38
14-12 Downstream Memory Read Prefetch Size 14-39
14-13 Upstream Memory Read Prefetch Size 14-39
14-14 Bridge Unit Queue 14-48
14-15 D_DRC Assignments 14-50
14-16 U_DRC Assignments 14-50
14-17 Bridge Transaction Ordering Rules 14-52
14-18 Bridge Transaction Ordering and Priority Mechanism 14-54
14-19 LOCK# Operation State Definitions 14-61
14-20 PSR Error Reporting Summary 14-74
14-21 SSR Error Reporting Summary 14-77
14-22 64-Bit Configuration Options at Reset 14-80
14-23 PCI-to-PCI Bridge Register Table 14-84
14-24 Vendor Identification Register - VIDR 14-85
14-25 Device Identification Register - DIDR (80303 I/O processor) 14-86
14-26 Primary Command Register - PCR 14-87
14-27 Primary Status Register - PSR 14-88
14-28 Revision Identification Register - RID 14-90
14-29 Class Code Register - CCR 14-91
14-30 Cacheline Size Register - CLSR 14-92
14-31 Primary Latency Timer Register- PLTR 14-93
14-32 Header Type Register- HTR 14-94
14-33 Primary Bus Number Register- PBNR 14-95
14-34 Secondary Bus Number Register - SBNR 14-96
14-35 Subordinate Bus Number Register - SubBNR 14-97
14-36 Secondary Latency Timer Register - SLTR 14-98
14-37 I/O Base Register - IOBR 14-99
14-38 I/O Limit Register - IOLR 14-100
14-39 Secondary Status Register - SSR 14-101
14-40 Memory Base Register - MBR 14-102
14-41 Memory Limit Register - MLR 14-103

14-42 Prefetchable Memory Base Register - PMBR 14-104
14-43 Prefetchable Memory Limit Register - PMLR 14-105
14-44 Capabilities Pointer Register - Cap_Ptr 14-106
14-45 Bridge Control Register - BCR 14-107
14-46 Extended Bridge Control Register - EBCR 14-109
14-47 Secondary IDSEL Select Register - SISR 14-111
14-48 Primary Bridge Interrupt Status Register - PBISR 14-113
14-49 Secondary Bridge Interrupt Status Register - SBISR 14-114
14-50 Secondary I/O Base Register - SIOBR 14-116
14-51 Secondary I/O Limit Register - SIOLR 14-117
14-52 Secondary Clock Disable Register - SCDR 14-118
14-53 Secondary Memory Base Register - SMBR 14-119
14-54 Secondary Memory Limit Register - SMLR 14-120
14-55 Secondary Decode Enable Register - SDER 14-121
14-56 Queue Control Register- QCR 14-123
14-57 Capability Identifier Register - Cap_ID 14-124
14-58 Next Item Pointer Register - Next_Item_Ptr 14-125
14-59 Power Management Capabilities Register - PMCR 14-126
14-60 Power Management Control/Status Register - PMCSR 14-127
14-61 PMCSR PCI-to-PCI Bridge Support - PMCSR_BSE 14-128
15-1 ATU Command Support 15-5
15-2 Outbound Read Fetch Sizes 15-20
15-3 PCI Multi-Function Device Swapping/Disabling Summary 15-23
15-4 Inbound Queues 15-31
15-5 Inbound Read Prefetch Data Sizes 15-33
15-6 Outbound Queues 15-34
15-7 ATU Inbound Data Flow Ordering Rules 15-35
15-8 ATU Outbound Data Flow Ordering Rules 15-35
15-9 Address Parity Errors on the PCI Interface 15-39
15-10 Outbound Read Data Parity Errors - Master 15-40
15-11 Outbound Write Data Parity Errors - Master 15-41
15-12 Inbound Write Data Parity Errors - Slave 15-41
15-13 Master Aborts on the PCI Interface 15-43
15-14 Target Aborts on the PCI Interface 15-44
15-15 Target Aborts on the Primary and Secondary ATUs 15-44
15-16 SERR# Assertion on the Primary and Secondary ATUs 15-45
15-17 SERR# Detection On the Primary and Secondary ATUs 15-46
15-18 Master Abort During an Inbound Write Transaction 15-47
15-19 Master Abort During an Inbound Read Transaction 15-48
15-20 Target Abort During an Inbound Write Transaction 15-49
15-21 Target Abort During an Inbound Read Transaction 15-49
15-22 Primary ATU Error Reporting Summary - PCI Interface 15-50
15-23 Secondary ATU Error Reporting Summary - PCI Interface 15-51
15-24 Primary ATU Error Reporting Summary - Internal Bus Interface 15-52
15-25 Secondary ATU Error Reporting Summary - Internal Bus Interface 15-53
15-26 Address Translation Unit Registers 15-56
15-27 ATU PCI Configuration Register Space 15-58
15-28 ATU Vendor ID Register - ATUVID 15-60
15-29 Device ID Register - DID (80303 I/O processor) 15-61
15-30 Primary ATU Command Register - PATUCMD 15-62

15-31 Primary ATU Status Register - PATUSR 15-63
15-32 ATU Revision ID Register - ATURID 15-65
15-33 ATU Class Code Register - ATUCCR 15-66
15-34 ATU Cacheline Size Register - ATUCLSR 15-67
15-35 ATU Latency Timer Register - ATULT 15-68
15-36 ATU Header Type Register - ATUHTR 15-69
15-37 ATU BIST Register - ATUBISTR 15-70
15-38 Primary Inbound ATU Base Address Register - PIABAR 15-71
15-39 ATU Subsystem Vendor ID Register - ASVIR 15-72
15-40 ATU Subsystem ID Register - ASIR 15-73
15-41 Expansion ROM Base Address Register -ERBAR 15-74
15-42 ATU Capabilities Pointer Register - ATU_Cap_Ptr 15-75
15-43 Memory Block Size Read Response Table 15-76
15-44 ATU Base Registers and Associated Limit Registers 15-76
15-45 ATU Interrupt Line Register - ATUILR 15-77
15-46 ATU Interrupt Pin Register - ATUIPR 15-78
15-47 ATU Minimum Grant Register - ATUMGNT 15-79
15-48 ATU Maximum Latency Register - ATUMLAT 15-80
15-49 Primary Inbound ATU Limit Register - PIALR 15-81
15-50 Primary Inbound ATU Translate Value Register - PIATVR 15-82
15-51 Secondary Inbound ATU Base Address Register - SIABAR 15-83
15-52 Secondary Inbound ATU Limit Register - SIALR 15-84
15-53 Secondary Inbound Translate ATU Value Register - SIATVR 15-85
15-54 Primary Outbound Memory Window Value Register - POMWVR 15-86
15-55 Primary Outbound I/O Window Value Register - POIOWVR 15-87
15-56 Primary Outbound DAC Window Value Register - PODWVR 15-88
15-57 Primary Outbound Upper 64-bit DAC Register - POUDR 15-89
15-58 Secondary Outbound Memory Window Value Register - SOMWVR 15-90
15-59 Secondary Outbound I/O Window Value Register - SOIOWVR 15-91
15-60 Expansion ROM Limit Register - ERLR 15-92
15-61 Expansion ROM Translate Value Register - ERTVR 15-93
15-62 ATU_Capability Identifier Register - ATU_Cap_ID 15-94
15-63 ATU Next Item Pointer Register - ATU_Next_Item_Ptr 15-95
15-64 ATU Power Management Capabilities Register - APMCR 15-96
15-65 ATU Power Management Control/Status Register - APMCSR 15-97
15-66 ATU Configuration Register - ATUCR 15-98
15-67 Primary ATU Interrupt Status Register - PATUISR 15-100
15-68 Secondary ATU Interrupt Status Register - SATUISR 15-102
15-69 Secondary ATU Command Register - SATUCMD 15-104
15-70 Secondary ATU Status Register - SATUSR 15-105
15-71 Secondary Outbound DAC Window Value Register - SODWVR 15-106
15-72 Secondary Outbound Upper 64-bit DAC Register - SOUDR 15-107
15-73 Primary Outbound Configuration Cycle Address Register - POCCAR 15-108
15-74 Secondary Outbound Configuration Cycle Address Register - SOCCAR 15-109
15-75 Primary Outbound Configuration Cycle Data Register - POCCDR 15-110
15-76 Secondary Outbound Configuration Cycle Data Register - SOCCDR 15-111
15-77 Primary ATU Queue Control Register - PAQCR 15-112
15-78 Secondary ATU Queue Control Register - SAQCR 15-113
15-79 Primary ATU Interrupt Mask Register - PATUIMR 15-114
15-80 Secondary ATU Interrupt Mask Register - SATUIMR 15-115

16-1 MU Summary 16-3
16-2 Circular Queue Ordering Requirements 16-4
16-3 Circular Queue Summary 16-7
16-4 Queue Starting Addresses 16-9
16-5 Circular Queue Summary 16-14
16-6 Message Unit Register Table 16-17
16-7 Inbound Message Register - IMRx 16-18
16-8 Outbound Message Register - OMRx 16-19
16-9 Inbound Doorbell Register - IDR 16-20
16-10 Inbound Interrupt Status Register - IISR 16-21
16-11 Inbound Interrupt Mask Register - IIMR 16-22
16-12 Outbound Doorbell Register - ODR 16-23
16-13 Outbound Interrupt Status Register - OISR 16-24
16-14 Outbound Interrupt Mask Register - OIMR 16-25
16-15 MU Configuration Register - MUCR 16-26
16-16 Queue Base Address Register - QBAR 16-27
16-17 Inbound Free Head Pointer Register - IFHPR 16-28
16-18 Inbound Free Tail Pointer Register - IFTPR 16-29
16-19 Inbound Post Head Pointer Register - IPHPR 16-30
16-20 Inbound Post Tail Pointer Register - IPTPR 16-31
16-21 Outbound Free Head Pointer Register - OFHPR 16-32
16-22 Outbound Free Tail Pointer Register - OFTPR 16-33
16-23 Outbound Post Head Pointer Register - OPHPR 16-34
16-24 Outbound Post Tail Pointer Register - OPTPR 16-35
16-25 Index Address Register - IAR 16-36
17-1 Bus Master / Programmed Priorities 17-4
17-2 Bus Arbitration Example – Three Bus Masters 17-4
17-3 Bus Arbitration Example – Six Bus Masters 17-5
17-4 Arbitration Flow 17-7
17-5 Arbitration Block and Reset Signals 17-12
17-6 Secondary Arbiter Register Table 17-13
17-7 Secondary Arbitration Control Register - SACR 17-14
17-8 2-Bit Priorities 17-14
17-9 Internal Arbitration Control Register - IACR 17-15
17-10 Master Latency Timer Register - MLTR 17-16
17-11 Multi-Transaction Timer Register - MTTR 17-17
18-1 Timer Performance Ranges 18-1
18-2 Timer Registers 18-2
18-3 Timer Mode Register – TMRx 18-3
18-4 Timer Input Clock (TCLOCK) Frequency Selection 18-5
18-5 Timer Count Register – TCRx 18-6
18-6 Timer Reload Register – TRRx 18-7
18-7 Timer Mode Register Control Bit Summary 18-8
18-8 Timer Responses to Register Bit Settings 18-9
18-9 Timer Powerup Mode Settings 18-10
18-10 Uncommon TMRx Control Bit Settings 18-10
19-1 DMA Registers 19-4
19-2 DMA Interrupt Summary 19-23
19-3 DMA Controller Unit Registers 19-26
19-4 Channel Control Register - CCR 19-27

19-5 Channel Status Register - CSR 19-28
19-6 Next Descriptor Address Register - NDAR 19-30
19-7 Descriptor Address Register - DAR 19-31
19-8 Byte Count Register - BCR 19-32
19-9 PCI Address Register - PADR 19-33
19-10 PCI Upper Address Register - PUADR 19-34
19-11 Intel® i960® Local Address Register - LADR 19-35
19-12 Descriptor Control Register - DCR 19-36
19-13 PCI Commands 19-37
20-1 Register Description 20-3
20-2 AAU Interrupts 20-23
20-3 Application Accelerator Unit Registers 20-25
20-4 Accelerator Control Register - ACR 20-26
20-5 Accelerator Status Register - ASR 20-27
20-6 Accelerator Descriptor Address Register - ADAR 20-28
20-7 Accelerator Next Descriptor Address Register - ANDAR 20-29
20-8 80960 Source Address Register - SARx 20-30
20-9 80960 Destination Address Register - DAR 20-31
20-10 Accelerator Byte Count Register - ABCR 20-32
20-11 Accelerator Descriptor Control Register - ADCR 20-33
21-1 Occurrence Events 21-3
21-2 Duration Events 21-4
21-3 Relationship between the Monitored mode and Monitored Interface 21-5
21-4 Event Monitor Register Table 21-22
21-5 Global Timer Mode Register (GTMR) 21-23
21-6 Event Select Register (ESR) - External Version 21-24
21-7 Event Monitoring Interrupt Status Register (EMISR) 21-25
21-8 Global Time Stamp Register - GTSR 21-27
21-9 Programmable Event Counter Register - PECRx 21-28
22-1 I^2C Bus Definitions 22-2
22-2 Modes of Operation 22-4
22-3 START and STOP Bit Definitions 22-5
22-4 ICCR Programming Values 22-8
22-5 Master Transactions 22-14
22-6 Slave Transactions 22-18
22-7 General Call Address Second Byte Definitions 22-20
22-8 I^2C Register Summary Table 22-28
22-9 I^2C Control Register - ICR 22-29
22-10 I^2C Status Register - ISR 22-32
22-11 I^2C Slave Address Register - ISAR 22-34
22-12 I^2C Data Buffer Register - IDBR 22-35
22-13 I^2C Clock Count Register - ICCR 22-36
22-14 I^2C Bus Monitor Register - IBMR 22-37
23-1 GPIO Output Enable Register - GPOE 23-3
23-2 GPIO Input Data Register - GPID 23-4
23-3 Output Data Register - GPOD 23-5
24-1 TAP Controller Pin Definitions 24-4
24-2 Boundary-Scan Instruction Set 24-5
24-3 IEEE Instructions 24-6
24-4 Intel® 80303 I/O Processor Boundary Scan Register Bit Order 24-8

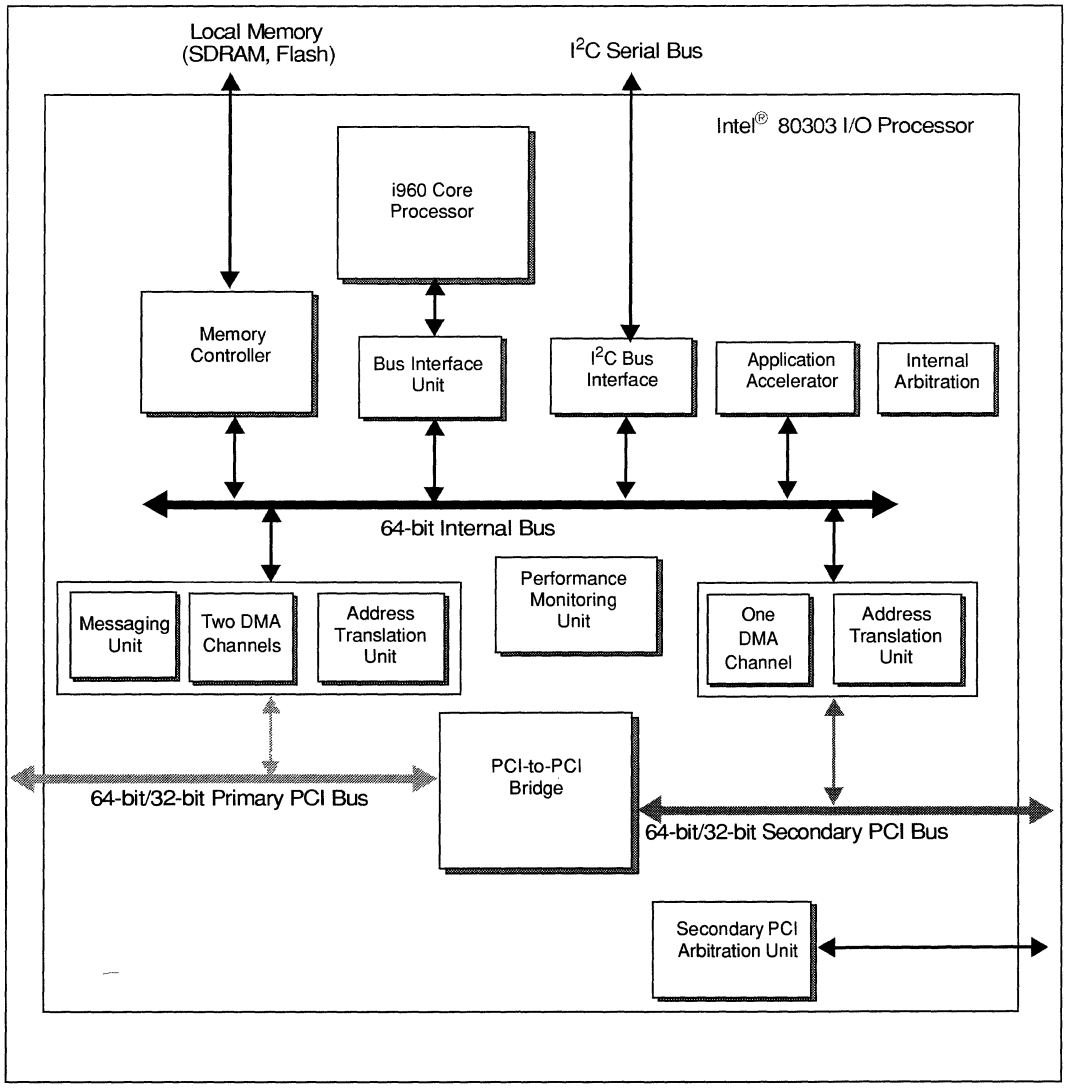
25-1 Clock Pin Summary 25-3
25-2 Clock Region Summary 25-3
25-3 Configuration Modes 25-7
A-1 Instruction Field Descriptions A-2
A-2 Encoding of *src1* and *src2* in REG Format A-3
A-3 Encoding of src/dst in REG Format A-3
A-4 Encoding of src1 in COBR Format A-4
A-5 Encoding of src2 in COBR Format A-4
A-6 Addressing Modes for MEM Format Instructions A-5
A-7 Encoding of Scale Field A-6
B-8 Miscellaneous Instruction Encoding Bits B-1
B-9 REG Format Instruction Encodings B-2
B-10 COBR Format Instruction Encodings B-6
B-11 CTRL Format Instruction Encodings B-7
B-12 Cycle Counts for sysctl Operations B-7
B-13 Cycle Counts for icctl Operations B-8
B-14 Cycle Counts for dcctl Operations B-8
B-15 Cycle Counts for intctl Operations B-8
B-16 MEM Format Instruction Encodings B-9
B-17 Addressing Mode Performance B-10
C-18 Intel® 80960 Local Addresses Assigned to Integrated Peripherals C-5
C-19 Peripheral Memory-Mapped Register Locations C-6

Introduction 1

1.1 Intel® 80303 I/O Processor

The Intel® 80303 I/O processor integrates a high-performance Intel® i960® core processor into a Peripheral Components Interconnect (PCI) functionality. This integrated processor addresses the needs of intelligent I/O applications and helps reduce intelligent I/O system costs. As indicated in Figure 1-1, the primary functional units include an i960 core processor, PCI to PCI bus bridge, Address Translation Unit, Messaging Unit, Direct Memory Access (DMA) Controller, Memory Controller, Secondary PCI bus Arbitration Unit, I^2C Bus Interface Unit, GPIO Interface Unit, Application Accelerator Unit, Performance Monitoring Unit and Bus Interface Unit.

Figure 1-1. Intel® 80303 I/O Processor Functional Block Diagram



1.2 Intel® 80303 I/O Processor Features

The 80303 I/O processor combines an enhanced Intel® i960® JT processor with powerful new features to create two intelligent I/O processors. This multi-function PCI device is fully compliant with the *PCI Local Bus Specification*, Revision 2.2. 80303 I/O processor-specific features include:

- PCI-to-PCI Bridge Unit
- Private PCI Device Support
- DMA Controller
- Address Translation Unit
- Memory Controller
- Application Accelerator Unit
- I2C Bus Interface Unit
- GPIO Interface Unit
- Performance Monitoring Unit
- Secondary PCI Arbitration Unit
- Messaging Unit
- I_2O Compatibility

The subsections that follow briefly overview each feature. Refer to the appropriate chapter for full technical descriptions.

The 80303 I/O processor's core processor is based upon an enhanced version of the i960 JT processor. The core processor operates at a maximum frequency of 100 MHz. The instruction cache is 16 Kbytes in size and is two-way set associative. The data cache is 16K bytes in size and is direct-mapped.

1.2.1 PCI-to-PCI Bridge Unit

The PCI-to-PCI bridge unit (referred to as "bridge") connects two independent PCI buses. It is fully compliant with the *PCI-to-PCI Bridge Architecture Specification*, Revision 1.0 published by the PCI Special Interest Group. It allows certain bus transactions on one PCI bus to be forwarded to the other PCI bus. It allows fully independent PCI bus operation and fully supports 32-bit or 64-bit bus widths. Dedicated data queues support high performance bandwidth on the PCI buses. The primary and secondary PCI buses may independently be configured as 32-bit or 64-bit wide. The 80303 I/O processor supports PCI 64-bit Dual Address Cycle (DAC) addressing.

The bridge has dedicated PCI configuration space that is accessible through the primary PCI bus. The bridge also supports the power management extended capability configuration header as defined by the *PCI Bus Power Management Interface Specification*, Revision 1.1 Compliance to this specification provides the hardware support required by the software initiative defined by the *Advanced Configuration and Power Interface Specification*, Revision 1.0 (ACPI).

1.2.2 Internal Bus

The Internal Bus is a high-speed interconnect between all internal units and controllers. The Internal Bus operates at 66 MHz and is 64 bits wide.

1.2.3 Private PCI Device Support

A key 80303 I/O processor feature is that it explicitly supports private PCI devices on the secondary PCI bus without being detected by PCI configuration software. The bridge and Address Translation Unit work together to hide private devices from PCI configuration cycles and allow these devices to use a private PCI address space. The Address Translation Unit uses normal PCI configuration cycles to configure these devices.

1.2.4 DMA Controller

The DMA Controller allows low-latency, high-throughput data transfers between PCI bus agents and 80303 local memory. Three separate DMA channels accommodate data transfers: two for primary PCI bus, one for the secondary PCI bus. The DMA Controller supports chaining and unaligned data transfers. It is programmable through the i960 core processor only.

1.2.5 Address Translation Unit

The Address Translation Unit (ATU) allows PCI transactions direct access to the 80303 I/O processor local memory. The ATU supports transactions between PCI address space and 80303 I/O processor address space. Address translation is controlled through programmable registers accessible from both the PCI interface and the i960 core processor. Dual access to registers allows flexibility in mapping the two address spaces. The ATU also supports the power management extended capability configuration header that as defined by the *PCI Bus Power Management Interface Specification*, Revision 1.1

1.2.6 Messaging Unit

The Messaging Unit (MU) provides data transfer between the PCI system and the 80303 I/O processor. It uses interrupts to notify each system when new data arrives. The MU has four messaging mechanisms: Message Registers, Doorbell Registers, Circular Queues and Index Registers. Each allows a host processor or external PCI device and the 80303 I/O processor to communicate through message passing and interrupt generation.

1.2.7 Memory Controller

The Memory Controller allows direct control of external memory systems, including SDRAM, SRAM, ROM and flash. It provides a direct connect interface to memory that typically does not require external logic. It features programmable chip selects, a wait state generator, and support for error correction codes (ECC). External memory can be configured as PCI addressable memory or private 80303 I/O processor memory.

1.2.8 Application Accelerator Unit

The Application Accelerator Unit transfers blocks of data to and from the local memory and performs boolean operations, such as XOR, on the data.

1.2.9 Performance Monitoring Unit

The Performance Monitoring Unit (PMON) allows various events on the 80303 I/O processor to be monitored. 14 Event Counters can be programmed to observe events selected from a pre-defined set of events.

1.2.10 I^2C Bus Interface Unit

The I^2C (Inter-Integrated Circuit) Bus Interface Unit allows the i960 core processor to serve as a master and slave device residing on the I^2C bus. The I^2C unit uses a serial bus developed by Philips Semiconductor consisting of a two-pin interface. The bus allows the 80303 I/O processor to interface to other I^2C peripherals and microcontrollers for system management functions. It requires a minimum of hardware for an economical system to relay status and reliability information on the I/O subsystem to an external device. Also refer to *I^2C Peripherals for Microcontrollers* (Philips Semiconductor).

1.2.11 GPIO Interface Unit

Eight pins are provided as General Purpose Input Output (GPIO) pins. These pins can be used by the Intel® i960® JT processor to control or monitor external devices in the I/O subsystem.

1.2.12 Secondary PCI Arbitration Unit

The Secondary PCI Arbitration Unit is the arbiter for the secondary PCI bus. It includes a fairness algorithm with programmable priorities and six PCI request and grant signal pairs. This arbitration unit can also be disabled to allow for external arbitration.

1.3 Terminology and Conventions

1.3.1 Representing Numbers

All numbers in this document can be assumed to be base 10 unless designated otherwise. In text, numbers in base 16 are represented as "nnnH", where the "H" signifies hexadecimal. In pseudo code descriptions, hexadecimal numbers are represented in the form 0x1234ABCD. Binary numbers are not explicitly identified but are assumed when bit operations or bit ranges are used.

1.3.2 Fields

A *reserved* field is a field that may be used by an implementation. If the initial value of a reserved field is supplied by software, this value must be zero. Software should not modify reserved fields or depend on any values in reserved fields.

A *read/write* field can written to a new value following initialization. This field can always be read to return the current value.

A *read only* field can be read to return the current value. Writes to *read only* fields are treated as no-op operations and will not change the current value nor result in an error condition.

A *read/clear* field can also be read to return the current value. A write to a *read/clear* field with the data value of 0 will cause no change to the field. A write to a *read/clear* field with a data value of 1 will cause the field to be cleared (reset to the value of 0). For example, if a *read/clear* field has a value of F0H, and a data value of 55H is written, the resultant field will be A0H.

A *read/set* field can also be read to return the current value. A write to a *read/set* field with the data value of 0 will cause no change to the field. A write to a *read/set* field with a data value of 1 will cause the field to be set (set to the value of 1). For example, if a *read/set* field has a value of F0H, and a data value of 55H is written, the resultant field will be F5H.

A *writeonce/readonly* field can be written to a new value **once** following initialization. After the this write has occurred, the *writeonce/readonly* field will treat all subsequent writes as no-op operations and will not change the current value or result in an error condition. The field can always be read to return the current value.

1.3.3 Specifying Bit and Signal Values

The terms *set* and *clear* in this specification refer to bit values in register and data structures. If a bit is set, its value is 1; if the bit is clear, its value is 0. Likewise, *setting* a bit means giving it a value of 1 and *clearing* a bit means giving it a value of 0.

The terms *assert* and *deassert* refer to the logically active or inactive value of a signal or bit, respectively.

1.3.4 Signal Name Conventions

All signal names use the PCI signal name convention of using the "#" symbol at the end of a signal name to indicate that the signal's active state occurs when it is at a low voltage. This includes 80303 processor related signal names that normally use an overline. The absence of the "#" symbol indicates that the signal's active state occurs when it is at a high voltage.

1.3.5 Terminology

To aid the discussion of the 80303 I/O processor architecture, the following terminology is used:

Downstream	At or toward a PCI bus with a higher number (after configuration)
DWORD	32-bit data word
QWORD	64-bit data word
Host processor	Processor located upstream from the 80303 I/O processor
Local processor	i960 core processor within the 80303 I/O processor
Local bus	80303 I/O processor Internal Bus
Local memory	Memory subsystem on the 80303 processor local bus
Upstream	At or toward a PCI bus with a lower number (after configuration)

Data Types and Memory Addressing Modes 2

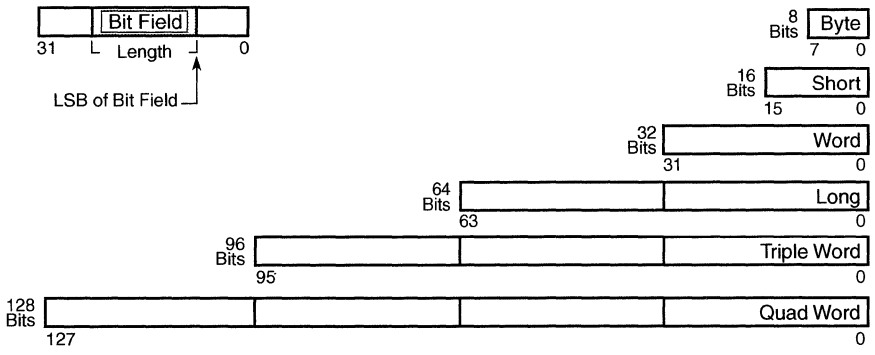
2.1 Data Types

The instruction set references or produces several data lengths and formats. The Intel® 80303 I/O processor supports the following data types:

- Integer (signed 8, 16 and 32 bits)
- Ordinal (unsigned integer 8, 16, and 32 bits)
- Long Word (64 bits)
- Triple Word (96 bits)
- Quad Word (128 bits)
- Bit Field
- Bit

Figure 2-1 illustrates the class, data type and length of each type supported by the Intel® i960® processor.

Figure 2-1. Data Types and Ranges



Class	Data Type	Length	Range
Numeric (Interger)	Byte Integer Short Integer Integer	8 Bits 16 Bits 32 Bits	-2^7 to 2^7-1 -2^{15} to $2^{15}-1$ -2^{31} to $2^{31}-1$
Numeric (Ordinal)	Byte Ordinal Short Ordinal Ordinal Long Ordinal	8 Bits 16 Bits 32 Bits 64 Bits	0 to 2^8-1 0 to $2^{16}-1$ 0 to $2^{32}-1$ 0 to $2^{64}-1$
Numeric (Ordinal)	Bit Bit Field Long Word Triple Word Quad Word	1 Bit 1-32 Bits 64 Bits 96 Bits 128 Bits	N/A

2.1.1 Word/Dword Notation

Data lengths, as described in the 80303 I/O processor, differ from the conventions used for the 80303 architecture. See also Table 2-1:

- In the PCI specification the term *word* refers to a 16-bit block of data.
- In this manual and other documentation relating to the 80303 I/O processor, the term *word* refers to a 32-bit block of data.

Table 2-1. Intel® 80303 I/O Processor and PCI Architecture Data Word Notation Differences

No. of Bits	PCI Architecture	80303 Architecture
16	**word**	short word or half word
32	doubleword or dword	**word**

2.1.2 Integers

Integers are signed whole numbers that are stored and operated on in two's complement format by the integer instructions. Most integer instructions operate on 32-bit integers. Byte and short integers are referenced by the byte and short classes of the load, store and compare instructions only.

Integer load or store size (byte, short or word) determines how sign extension or data truncation is performed when data is moved between registers and memory.

For instructions **ldib** (load integer byte) and **ldis** (load integer short), a byte or short word in memory is considered a two's complement value. The value is sign-extended and placed in the 32-bit register that is the destination for the load.

Example 2-1. Sign Extensions on Load Byte and Load Short

```
ldib
    7AH is loaded into a register as 0000 007AH
    FAH is loaded into a register as FFFF FFFAH

ldis
    05A5H is loaded into a register as 0000 05A5H
    85A5H is loaded into a register as FFFF 85A5H
```

For instructions **stib** (store integer byte) and **stis** (store integer short), a 32-bit two's complement number in a register is stored to memory as a byte or short word. When register data is too large to be stored as a byte or short word, the value is truncated and the integer overflow condition is signalled. When an overflow occurs, either an AC register flag is set or the ARITHMETIC.INTEGER_OVERFLOW fault is generated, depending on the Integer Overflow Mask bit (AC.om) in the AC register. Chapter 9, "Faults" describes the integer overflow fault.

For instructions **ld** (load word) and **st** (store word), data is moved directly between memory and a register with no sign extension or data truncation.

2.1.3 Ordinals

Ordinals or unsigned integer data types are stored and treated as positive binary values. Figure 2-1 shows the supported ordinal sizes.

The large number of instructions that perform logical, bit manipulation and unsigned arithmetic operations reference 32-bit ordinal operands. When ordinals are used to represent Boolean values, 1 = TRUE and 0 = FALSE. Most extended arithmetic instructions reference the long ordinal data type. Only load (**ldob** and **ldos**), store (**stob** and **stos**), and compare ordinal instructions reference the byte and short ordinal data types.

Sign and sign extension are not considered when ordinal loads and stores are performed; however, the values may be zero-extended or truncated. A short word or byte load to a register causes the value loaded to be zero-extended to 32 bits. A short word or byte store to memory truncates an ordinal value in a register to fit the destination memory. No overflow condition is signalled in this case.

2.1.4 Bits and Bit Fields

The processor provides several instructions that perform operations on individual bits or bit fields within register operands. An individual bit is specified for a bit operation by giving its bit number and register. Internal registers always follow little endian byte order; the least significant bit is bit 0 and the most significant bit is bit 31.

A bit field is any contiguous group of bits (up to 32 bits long) in a 32-bit register. Bit fields do not span register boundaries. A bit field is defined by giving its length in bits (1-32) and the bit number of its lowest numbered bit (0-31).

Loading and storing bit and bit-field data is normally performed using the ordinal load (**ldo**) and store (**sto**) instructions. When an **ldi** instruction loads a bit or bit field value into a 32-bit register, the processor appends sign extension bits. A byte or short store can signal an integer overflow condition.

2.1.5 Triple and Quad Words

Triple and quad words refer to consecutive words in memory or in registers. Triple- and quad-word load, store and move instructions use these data types to accomplish block movements. No data manipulation (sign extension, zero extension or truncation) is performed in these instructions.

Triple- and quad-word data types can be considered a superset of the other data types described. Data in each word subset of a quad word is likely to be the operand or result of an ordinal, integer, bit or bit field instruction.

2.1.6 Register Data Alignment

Several instructions operate on multiple-word operands. For example, the load-long instruction (**ldl**) loads two words from memory into two consecutive registers. Here the register number for the least significant word is automatically loaded into the next higher-numbered register.

In cases where an instruction specifies a register number, and multiple, consecutive registers are implied, the register number must be even if two registers are accessed (e.g., g0, g2) and an integral multiple of four if three or four registers are accessed (e.g., g0, g4). When a register reference for a source value is not properly aligned, the registers that the processor writes to are undefined.

The 80303 I/O processor does not require data alignment in external memory; the processor hardware handles unaligned memory accesses automatically. Optionally, user software can configure the processor to generate a fault on unaligned memory accesses.

2.1.7 Literals

The architecture defines a set of 32 literals that can be used as operands in many instructions. These literals are ordinal (unsigned) values that range from 0 to 31 (5 bits). When a literal is used as an operand, the processor expands it to 32 bits by adding leading zeros. If the instruction requires an operand larger than 32 bits, the processor zero-extends the value to the operand size. If a literal is used in an instruction that requires integer operands, the processor treats the literal as a positive integer value.

2.2 Bit and Byte Ordering in Memory

All occurrences of numeric and non-numeric data types, except bits and bit fields, must start on a byte boundary. Any data item occupying multiple bytes is stored as little endian.

2.3 Memory Addressing Modes

Nine modes are available for addressing operands in memory. Each addressing mode is used to reference a byte location in the processor's address space. Table 2-2 shows the memory addressing modes and a brief description of each mode's address elements and assembly code syntax.

Table 2-2. Memory Addressing Modes

Mode	Description	Assembler Syntax	Inst. Type
Absolute*offset*	offset (smaller than 4096)	exp	MEMA
displacement	displacement (larger than 4095)	exp	MEMB
Register Indirect	abase	(reg)	MEMB
with offset	abase + offset	exp (reg)	MEMA
with displacement	abase + displacement	exp (reg)	MEMB
with index	abase + (index*scale)	(reg) [reg*scale]	MEMB
with index and displacement	abase + (index*scale) + displacement	exp (reg) [reg*scale]	MEMB
Index with displacement	(index*scale) + displacement	exp [reg*scale]	MEMB
instruction pointer (IP) with displacement	IP + displacement + 8	exp (IP)	MEMB

NOTE: *reg* is register, *exp* is an expression or symbolic label, and IP is the Instruction Pointer.

See Table B-8, "Miscellaneous Instruction Encoding Bits" on page B-1 for more on addressing modes. For purposes of this memory addressing modes description, MEMA format instructions require one word of memory and MEMB usually require two words and therefore consume twice the bus bandwidth to read. Otherwise, both formats perform the same functions.

2.3.1 Absolute

Absolute addressing modes allow a memory location to be referenced directly as an offset from address 0H. At the instruction encoding level, two absolute addressing modes are provided: absolute offset and absolute displacement, depending on offset size.

- For the absolute offset addressing mode, the offset is an ordinal number ranging from 0 to 4095. The absolute offset addressing mode is encoded in the MEMA machine instruction format.
- For the absolute displacement addressing mode, the offset value ranges from 0 to 2^{32}-1. The absolute displacement addressing mode is encoded in the MEMB format.

Addressing modes and encoding instruction formats are described in Chapter 6, "Instruction Set Reference".

At the assembly language level, the two absolute addressing modes use the same syntax. Typically, development tools allow absolute addresses to be specified through arithmetic expressions (e.g., x + 44) or symbolic labels. After evaluating an address specified with the absolute addressing mode, the assembler converts the address into an offset or displacement and selects the appropriate instruction encoding format and addressing mode.

2.3.2 Register Indirect

Register indirect addressing modes use a register's 32-bit value as a base for address calculation. The register value is referred to as the address base (designated "abase" in Table 2-2). Depending on the addressing mode, an optional scaled index and offset can be added to this address base.

Register indirect addressing modes are useful for addressing elements of an array or record structure. When addressing array elements, the abase value provides the address of the first array element. An offset (or displacement) selects a particular array element.

In register-indirect-with-index addressing mode, the index is specified using a value contained in a register. This index value is multiplied by a scale factor. Allowable factors are 1, 2, 4, 8 and 16. The register-indirect-with-index addressing mode is encoded in the MEMA format.

The two versions of register-indirect-with-offset addressing mode at the instruction encoding level are register-indirect-with-offset and register-indirect-with-displacement. As with absolute addressing modes, the mode selected depends on the size of the offset from the base address.

At the assembly language level, the assembler allows the offset to be specified with an expression or symbolic label, then evaluates the address to determine whether to use register-indirect-with-offset (MEMA format) or register-indirect-with-displacement (MEMB format) addressing mode.

Register-indirect-with-index-and-displacement addressing mode adds both a scaled index and a displacement to the address base. There is only one version of this addressing mode at the instruction encoding level, and it is encoded in the MEMB instruction format.

2.3.3 Index with Displacement

A scaled index can also be used with a displacement alone. Again, the index is contained in a register and multiplied by a scaling constant before displacement is added. This mode uses MEMB format.

2.3.4 IP with Displacement

This addressing mode is used with load and store instructions to make them instruction pointer (IP) relative. IP-with-displacement addressing mode references the next instruction's address plus the displacement plus a constant of 8. The constant is added because, in a typical processor implementation, the address has incremented beyond the next instruction address at the time of address calculation. The constant simplifies IP-with-displacement addressing mode implementation. This mode uses MEMB format.

2.3.5 Addressing Mode Examples

The following examples show how i960 processor addressing modes are encoded in assembly language. Example 2-2 shows addressing mode mnemonics. Example 2-3 illustrates the usefulness of scaled index and scaled index plus displacement addressing modes. In this example, a procedure named array_op uses these addressing modes to fill two contiguous memory blocks separated by a constant offset. A pointer to the top of the block is passed to the procedure in g0, the block size is passed in g1 and the fill data in g2. Refer to Appendix A, "Machine-Level Instruction Formats".

Example 2-2. Addressing Mode Mnemonics

```
st      g4,xyz              # Absolute; word from g4 stored at memory
                            # location designated with label xyz.
ldob    (r3),r4             # Register indirect; ordinal byte from
                            # memory location given in r3 loaded
                            # into register r4 and zero extended.
stl     g6,xyz(g5)          # Register indirect with displacement;
                            # double word from g6,g7 stored at memory
                            # location xyz + g5.
ldq     (r8)[r9*4],r4       # Register indirect with index; quad-word
                            # beginning at memory location r8 + (r9
                            # scaled by 4) loaded into r4 through r7.
st      g3,xyz(g4)[g5*2]    # Register indirect with index and
                            # displacement; word in g3 stored to mem
                            # location g4 + xyz + (g5 scaled by 2).
ldis    xyz[r12*2],r13      # Index with displacement; load short
                            # integer at memory location xyz + r12
                            # into r13 and sign extended.
st      r4,xyz(IP)          # IP with displacement; store word in r4
                            # at memory location IP + xyz + 8.
```

Example 2-3. Scaled Index and Scaled Index Plus Displacement Addressing Modes

```
array_op:
  mov         g0,r4             # Pointer to array is copied to r4.
  subi        1,g1,r3           # Calculate index for the last array
  b           .I33              # element to be filled
.I34:
  st          g2,(r4)[r3*4]     # Fill element at index
  st          g2,0x30(r4)[r3*4  # Fill element at index+constant offset
              ]
  subi        1,r3,r3           # Decrement index
.I33:
  cmpible     0,r3,.I34         # Store next array elements if
  ret                           # index is not 0
```

intel®

Programming Environment 3

This chapter describes the Intel® 80303 I/O processor programming environment including global and local registers, control registers, literals, processor-state registers and address space.

3.1 Overview

The Intel® i960® architecture defines a programming environment for program execution, data storage and data manipulation. Figure 3-1 shows the programming environment elements that include a 4 Gbyte (2^{32} byte) flat address space, an instruction cache, a data cache, global and local general-purpose registers, a register cache, a set of literals, control registers and a set of processor state registers.

The processor includes several architecturally-defined data structures located in memory as part of the programming environment. These data structures handle procedure calls, interrupts and faults and provide configuration information at initialization. These data structures are:

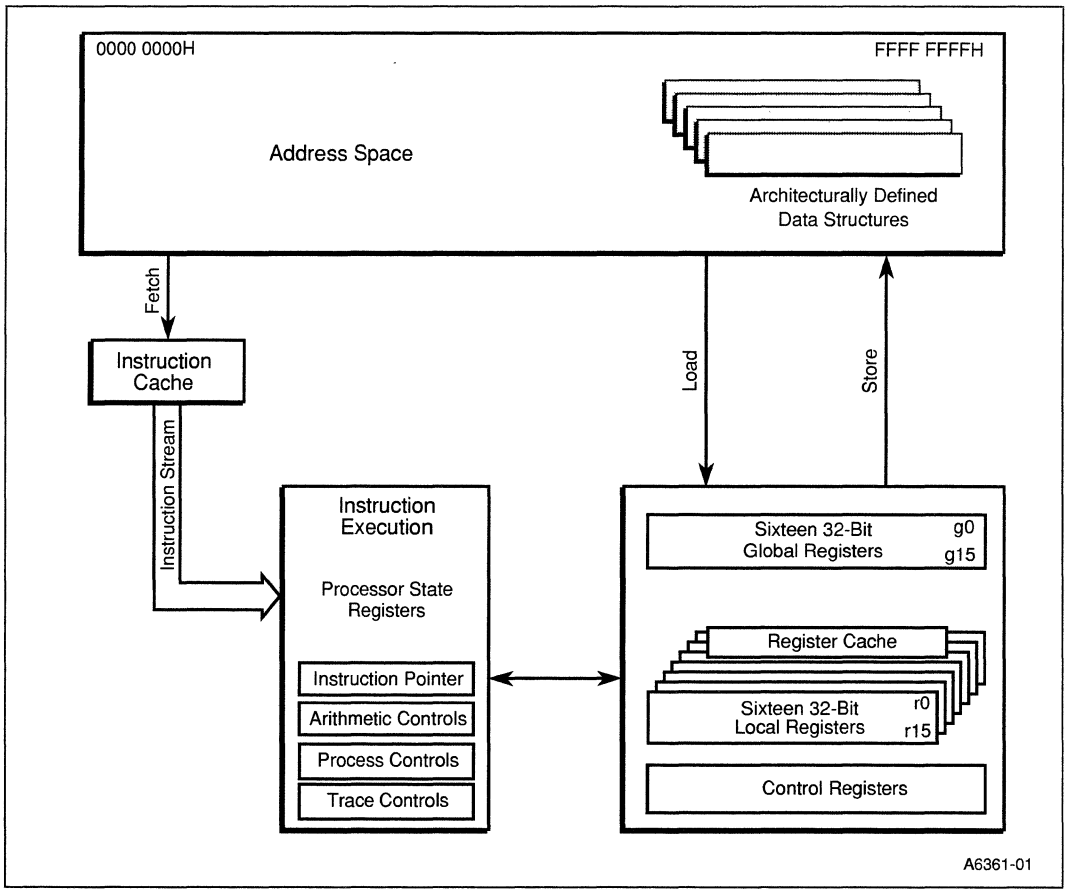
- interrupt stack
- control table
- system procedure table
- local stack
- fault table
- process control block
- supervisor stack
- interrupt table
- initialization boot record

3.2 Registers and Literals as Instruction Operands

With the exception of a few special instructions, the 80303 I/O processor uses only simple load and store instructions to access memory. All operations take place at the register level. The processor uses 16 global registers, 16 local registers and 32 literals (constants 0-31) as instruction operands.

The global register numbers are g0 through g15; local register numbers are r0 through r15. Several of these registers are used for dedicated functions. For example, register r0 is the previous frame pointer, often referred to as *pfp*. i960 processor compilers and assemblers recognize only the instruction operands listed in Table 3-1. Throughout this manual, the registers' descriptive names, numbers, operands and acronyms are used interchangeably, as dictated by context.

Figure 3-1. Intel® 80303 I/O Processor Programming Environment



3.2.1 Global Registers

Global registers are general-purpose 32-bit data registers that provide temporary storage for a program's computational operands. These registers retain their contents across procedure boundaries. As such, they provide a fast and efficient means of passing parameters between procedures.

Table 3-1. Registers and Literals Used as Instruction Operands

Instruction Operand	Register Name (number)	Function	Acronym
g0 - g14	global (g0-g14)	general purpose	
fp	global (g15)	frame pointer	FP
pfp	local (r0)	previous frame pointer	PFP
sp	local (r1)	stack pointer	SP
rip	local (r2)	return instruction pointer	RIP
r3 - r15	local (r3-r15)	general purpose	
0-31		literals	

The i960 architecture supplies 16 global registers, designated g0 through g15. Register g15 is reserved for the current Frame Pointer (FP), which contains the address of the first byte in the current (topmost) stack frame in internal memory. See Section 7.1, "Call and Return Mechanism" on page 7-2) for a description of the FP and procedure stack.

After the processor is reset, register g0 contains the 80303 I/O processor device identification and stepping information. g0 retains this information until it is written over by the user program. The 80303 I/O processor device identification and stepping information is also stored in the memory-mapped DEVICEID register located at FF00 8710H. In addition, the 80303 I/O processor device identification and stepping information is stored in the memory-mapped register located at 0000 1710H.

3.2.2 Local Registers

The i960 architecture provides a separate set of 32-bit local data registers (r0 through r15) for each active procedure. These registers provide storage for variables that are local to a procedure. Each time a procedure is called, the processor allocates a new set of local registers and saves the calling procedure's local registers. When the application returns from the procedure, the local registers are released for the next procedure call. The processor performs local register management; a program need not explicitly save and restore these registers.

r3 through r15 are general purpose registers; r0 through r2 are reserved for special functions; r0 contains the Previous Frame Pointer (PFP); r1 contains the Stack Pointer (SP); r2 contains the Return Instruction Pointer (RIP). These are discussed in Chapter 7, "Procedure Calls".

The processor does not always clear or initialize the set of local registers assigned to a new procedure. Also, the processor does not initialize the local register save area in the newly created stack frame for the procedure. User software should not rely on the initial values of local registers.

3.2.3 Register Scoreboarding

Register scoreboarding maintains register coherency by preventing parallel execution units from accessing registers for which there is an outstanding operation. When an instruction that targets a destination register or group of registers executes, the processor sets a register-scoreboard bit to indicate that this register or group of registers is being used in an operation. If the instructions that follow do not require data from registers already in use, the processor can execute those instructions before the prior instruction execution completes.

Software can use this feature to execute one or more single-cycle instructions concurrently with a multi-cycle instruction (e.g., multiply or divide). Example 3-1 shows a case where register scoreboarding prevents a subsequent instruction from executing. It also illustrates overlapping instructions that do not have register dependencies.

Example 3-1. Register Scoreboarding

```
muli      r4,r5,r6      # r6 is scoreboarded
addi      r6,r7,r8      # addi must wait for the previous multiply
          .             # to complete
          .
muli      r4,r5,r10     # r10 is scoreboarded
and       r6,r7,r8      # and instruction is executed concurrently with multiply
```

3.2.4 Literals

Architecture defines a set of 32 literals that can be used as operands in many instructions. These literals are ordinal (unsigned) values ranging from 0 to 31 (5 bits). When a literal is used as an operand, the processor expands it to 32 bits by adding leading zeros. If the instruction requires an operand larger than 32 bits, the processor zero-extends the value to operand size. If a literal is used in an instruction requiring integer operands, the processor treats the literal as a positive integer value.

3.2.5 Register and Literal Addressing and Alignment

Several instructions operate on multiple-word operands. For example, the load long instruction (**ldl**) loads two words from memory into two consecutive registers. The register for the less significant word is specified in the instruction. The more significant word is automatically loaded into the next higher-numbered register.

In cases where an instruction specifies a register number and multiple consecutive registers are implied, the register number must be even if two registers are accessed (e.g., g0, g2) and an integral multiple of 4 if three or four registers are accessed (e.g., g0, g4). If a register reference for a source value is not properly aligned, the source value is undefined and an OPERATION.INVALID_OPERAND fault is generated. If a register reference for a destination value is not properly aligned, the registers to which the processor writes and the values written are undefined. The processor then generates an OPERATION.INVALID_OPERAND fault. The assembly language code in Example 3-2 shows an example of correct and incorrect register alignment.

Example 3-2. Register Alignment

```
movl      g3,g8         # Incorrect alignment - resulting value
          .             # in registers g8 and g9 is
          .             # unpredictable (non-aligned source)
          .
movl      g4,g8         # Correct alignment
```

Global registers, local registers and literals are used directly as instruction operands. Table 3-2 lists instruction operands for each machine-level instruction format and the positions that can be filled by each register or literal.

Table 3-2. Allowable Register Operands

		Operand[1]		
Instruction Encoding	**Operand Field**	**Local Register**	**Global Register**	**Literal**
REG	*src1*	X	X	X
	src2	X	X	X
	src/dst (as *src*)	X	X	X
	src/dst (as *dst*)	X	X	
	src/dst (as both)	X	X	
MEM	*src/dst*	X	X	
	abase	X	X	
	index	X	X	
COBR	src1	X	X	
	src2	X	X	X
	dst	X[2]	X[2]	X[2]

NOTES:
1. "X" denotes the register can be used as an operand in a particular instruction field.
2. The **COBR** destination operands apply only to **TEST** instructions.

3.3 Memory-Mapped Control Registers (MMRs)

The 80303 I/O processor gives software the interface to easily read and modify internal control registers. Each of these registers is accessed as a memory-mapped register with a unique memory address. There are two distinct sets of memory-mapped registers on the 80303 I/O processor. The first set exists in the FF00 0000H through FFFF FFFFH address range and is used to control the 80303 I/O processor functions. The second set exists in the 0000 1000H through 0000 18FFH address range and is used to control the 80303 I/O processor integrated peripherals. The processor ensures that accesses to MMRs do not generate external bus cycles.

3.3.1 Intel® i960® Core Processor Function Memory-Mapped Registers

Portions of the 80303 I/O processor address space (addresses FF00 0000H through FFFF FFFFH) are reserved for memory-mapped registers. These memory-mapped registers are accessed through word-operand memory instructions (**atmod**, **atadd**, **sysctl**, **ld** and **st** instructions) only. Accesses to this address space do not generate external bus cycles. The latency in accessing each of these registers is one cycle.

Each register has an associated access mode (user and supervisor modes) and access type (read and write accesses). Table C-18 and Table C-19 show all the memory-mapped registers.

The registers are partitioned into user and supervisor spaces based on their addresses. Addresses FF00 0000H through FF00 7FFFH are allocated to user space memory-mapped registers; Addresses FF00 8000H to FFFF FFFFH are allocated to supervisor space registers.

3.3.1.1 Restrictions on Instructions that Access the Intel® i960® Core Processor Memory-Mapped Registers

The majority of memory-mapped registers can be accessed by both load (**ld**) and store (**st**) instructions. However some registers have restrictions on the types of accesses they allow. To ensure correct operation, the access type restrictions for each register should be followed. The access type columns of Table C-18 and Table C-19 indicate the allowed access types for each register.

Unless otherwise indicated by its access type, the modification of a memory-mapped register by a **st** instruction takes effect completely before the next instruction starts execution.

Some operations require an atomic-read-modify-write sequence to a register, most notably IPND and IMSK. The **atmod** and **atadd** instructions provide a special mechanism to quickly modify the IPND and IMSK registers in an atomic manner on the 80303 I/O processor. Do not use this instruction on any other memory-mapped registers.

The **sysctl** instruction can also modify the contents of a memory-mapped register atomically; in addition, **sysctl** is the only method to read the breakpoint registers on the 80303 I/O processor; the breakpoints cannot be read using a **ld** instruction.

At initialization the control table is automatically loaded into the on-chip control registers. This action simplifies the user's start-up code by providing a transparent setup of the processor's peripherals. See Chapter 11, "Initialization and System Requirements".

3.3.1.2 Access Faults for Intel® i960® Core Processor MMRs

Memory-mapped registers are meant to be accessed only as aligned, word-size registers with adherence to the appropriate access mode. Accessing these registers in any other way results in faults or undefined operation. An access is performed using the following fault model:

1. The access must be a word-sized, word-aligned access; otherwise, the processor generates an OPERATION.UNIMPLEMENTED fault.
2. If the access is a store in user mode to an implemented supervisor location, a TYPE.MISMATCH fault occurs. It is unpredictable whether a store to an unimplemented supervisor location causes a fault.
3. If the access is neither of the above, the access is attempted. Note that an MMR may generate faults based on conditions specific to that MMR. (Example: trying to write the timer registers in user mode when they have been allocated to supervisor mode only.)
4. When a store access to an MMR faults, the processor ensures that the store does not take effect.
5. A load access of a reserved location returns an unpredictable value.
6. Avoid any store accesses to reserved locations. Such a store can result in undefined operation of the processor if the location is in supervisor space.

Instruction fetches from the memory-mapped register space are not allowed and result in an OPERATION.UNIMPLEMENTED fault.

3.3.2 Intel® 80303 I/O Processor Peripheral Memory-Mapped Registers

The Peripheral Memory-Mapped Register (PMMR) interface gives software the ability to read and modify internal control registers. Each of these 32-bit registers is accessed as a memory-mapped register with a unique memory address, using regular memory-format instructions from the 80303 I/O processor. See Appendix C, "Peripheral Memory-Mapped Registers".

The memory-mapped registers discussed in this chapter are specific to the 80303 I/O processor only. They support the DMA controller, memory controller, PCI and peripheral interrupt controller, messaging unit, internal arbitration unit, PCI to PCI bridge unit, PCI address translation unit, I^2C bus interface unit, performance monitoring unit, and the application accelerator unit. This manual provides chapters that fully describe each of these peripherals.

The PMMR interface (addresses 0000 1000H through 0000 18FFH) provides full accessibility from the primary ATU, secondary ATU, and the 80303 I/O processor.

3.3.2.1 Accessing The Peripheral Memory-Mapped Registers

The PMMR interface is a slave device connected to the 80303 internal bus. This interface accepts data transactions that appear on the 80303 internal bus from the Primary ATU, Secondary ATU, and the 80303 I/O processor. The PMMR interface allows these devices to perform read, write, or read-modify-write transactions.

The PMMR interface does not support multi-word burst accesses from any bus master. The PMMR interface supports 32-bit bus width transactions only. Because of this, PMCON0:1 must be configured as a 32-bit memory region for accesses that originate from the 80303 I/O processor.

The PMMR interface is byte addressable. For PMMR reads, all accesses are promoted to word accesses and all data bytes are returned. The byte enables generated by the bus masters when performing PMMR write cycles indicate which data bytes are valid on the 80303 internal bus. However, there may be requirements from the individual units that interface to the PMMR. For example, when configuring the DMA channel's control register, a full 32-bit write must be performed to configure and restart the DMA channel. These restrictions are highlighted in the chapters describing the integrated peripheral units.

The PMMR interface supports the 80303 internal bus atomic operations from the 80303 I/O processor. The 80303 I/O processor provides **atmod** (atomic modify) and **atadd** (atomic add) instructions for atomic accesses to memory. When the 80303 processor executes an **atmod** or **atadd** instruction, the LOCK# signal is asserted. The 80303 internal bus is not granted to any other bus master until the LOCK# signal is deasserted. This prevents other bus masters from accessing the PMMR interface during a locked operation.

All PMMR transactions are allowed from 80303 I/O processor operating in either user mode or supervisor mode. In addition, the PMMR does not provide any access fault to the 80303 I/O processor.

The following PMMR registers have read/write access from the 80303 internal bus (for both the PCI Bridge and ATU):

- Vendor ID register
- Device ID register
- Revision ID register
- Class Code register
- Header Type register
- Bridge Subsystem ID register
- Bridge Subsystem Vendor ID register

For accesses through PCI configuration cycles, access is specified in the register definition located in the appropriate chapter.

For PCI configuration read transactions, the PMMR returns a zero value for reserved registers. For PCI configuration write transactions, the PMMR discards the data. For all other accesses, reading or writing a reserved register is undefined. See Table C-18 and Table C-19 for register memory locations.

3.4 Architecturally Defined Data Structures

The architecture defines a set of data structures including stacks, interfaces to system procedures, interrupt handling procedures and fault handling procedures. Table 3-3 defines the data structures and references other sections of this manual where detailed information can be found.

The 80303 I/O processor defines two initialization data structures: the Initialization Boot Record (IBR) and the Process Control Block (PRCB). These structures provide initialization data and pointers to other data structures in memory. When the processor is initialized, these pointers are read from the initialization data structures and cached for internal use.

Pointers to the system procedure table, interrupt table, interrupt stack, fault table and control table are specified in the processor control block. Supervisor stack location is specified in the system procedure table. User stack location is specified in the user's startup code. Of these structures, only the system procedure table, fault table, control table and initialization data structures may be in ROM; the interrupt table and stacks must be in RAM. The interrupt table must be located in RAM to allow posting of software interrupts.

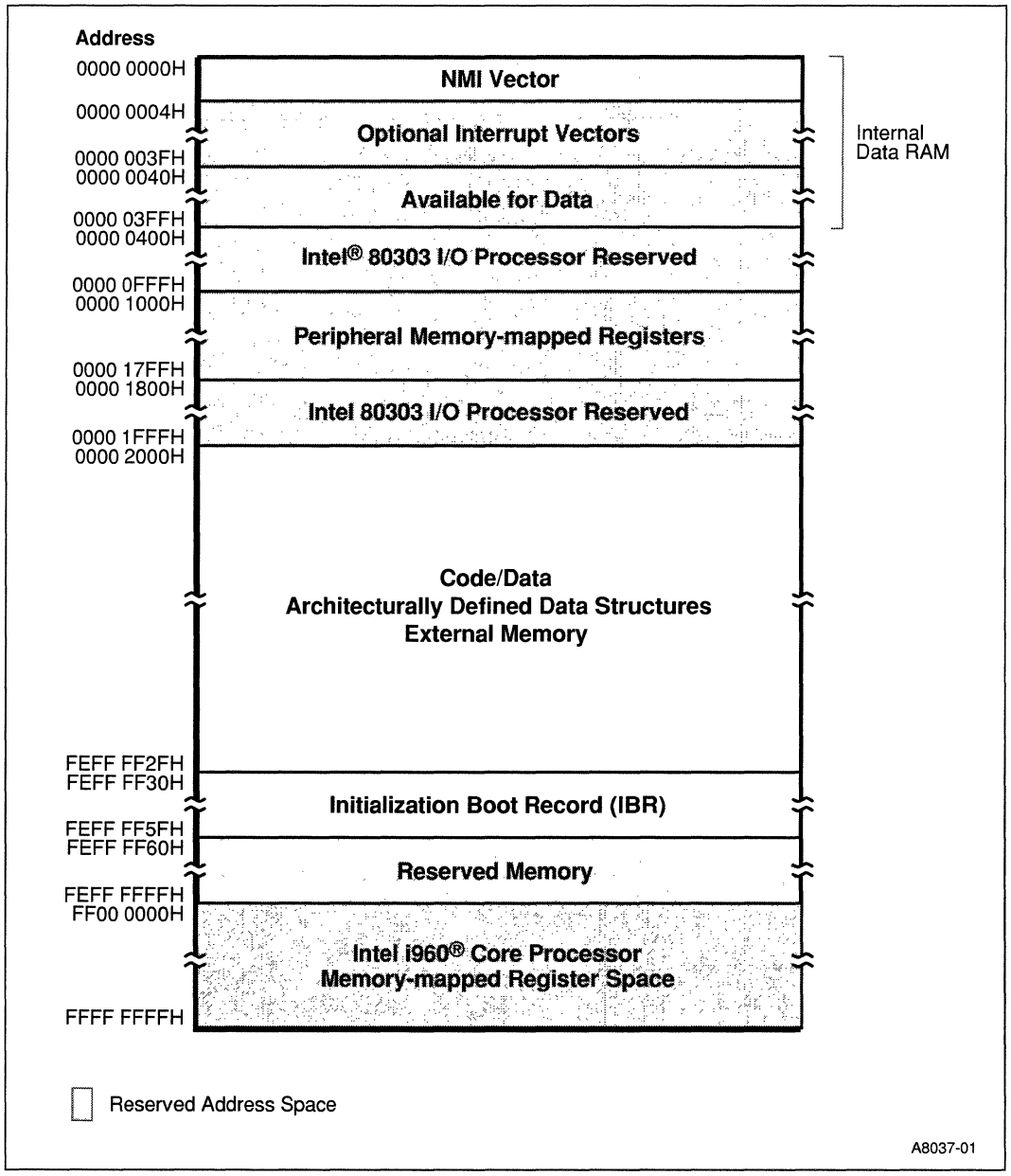
Table 3-3. Data Structure Descriptions

Structure	Description
User and Supervisor Stacks Section 7.6, "User and Supervisor Stacks" on page 7-18	The processor uses these stacks when executing application code.
Interrupt Stack Section 8.1.5, "Interrupt Stack And Interrupt Record" on page 8-6	A separate interrupt stack is provided to ensure that interrupt handling does not interfere with application programs.
System Procedure Table Section 3.7, "User-Supervisor Protection Model" on page 3-18 Section 7.5, "System Calls" on page 7-15	Contains pointers to system procedures. Application code uses the system call instruction (**calls**) to access system procedures through this table. A system supervisor call switches execution mode from user mode to supervisor mode. When the processor switches modes, it also switches to the supervisor stack.
Interrupt Table Section 8.1.4, "Interrupt Table" on page 8-4	The interrupt table contains vectors (pointers) to interrupt handling procedures. When an interrupt is serviced, a particular interrupt table entry is specified.
Fault Table Section 9.3, "Fault Table" on page 9-5	Contains pointers to fault handling procedures. When the processor detects a fault, it selects a particular entry in the fault table. The architecture does not require a separate fault handling stack. Instead, a fault handling procedure uses the supervisor stack, user stack or interrupt stack, depending on the processor execution mode in which the fault occurred and the type of call made to the fault handling procedure.
Control Table Section 11.4.4, "Control Table" on page 11-19	Contains on-chip control register values. Control table values are moved to on-chip registers at initialization or with **sysctl**.

3.5 Memory Address Space

The 80303 I/O processor's local address space is byte-addressable with addresses running contiguously from 0 to 2^{32}-1. Some memory space is reserved or assigned special functions as shown in Figure 3-2.

Figure 3-2. Local Memory Address Space



Physical addresses can be mapped to read-write memory, read-only memory and memory-mapped I/O. The architecture does not define a dedicated, addressable I/O space. There are no subdivisions of the address space such as segments. For memory management, an external memory management unit (MMU) may subdivide memory into pages or restrict access to certain areas of memory to protect a kernel's code, data and stack. However, the processor views this address space as linear.

An address in memory is a 32-bit value in the range 0H to FFFF FFFFH. Depending on the instruction, an address can reference in memory a single byte, short word (2 bytes), word (4 bytes), double word (8 bytes), triple word (12 bytes) or quad word (16 bytes). Refer to load and store instruction descriptions in Chapter 6, "Instruction Set Reference" for multiple-byte addressing information.

3.5.1 Memory Requirements

The architecture requires that external memory have the following properties:

- Memory must be byte-addressable.
- Physical memory must not be mapped to reserved addresses that are specifically used by the processor implementation.
- Memory must guarantee indivisible access (read or write) for addresses that fall within 16-byte boundaries.
- Memory must guarantee atomic access for addresses that fall within 16-byte boundaries.

The latter two capabilities, *indivisible* and *atomic* access, are required only when multiple processors or other external agents, such as DMA or graphics controllers, share a common memory.

indivisible access — Guarantees that a processor, reading or writing a set of memory locations, complete the operation before another processor or external agent can read or write the same location. The processor requires indivisible access within an aligned 16-byte block of memory.

atomic access — A read-modify-write operation. Here the external memory system must guarantee that once a processor begins a read-modify-write operation on an aligned, 16-byte block of memory it is allowed to complete the operation before another processor or external agent can access to the same location. An atomic memory system can be implemented by using the 80303 I/O processor signal to qualify hold requests from external bus agents. The processor asserts 80303 I/O processor for the duration of an atomic memory operation.

The upper 16 Mbytes of the address space (addresses FF00 0000H through FFFF FFFFH and 0000 1000H through 0000 018FFH) are reserved for implementation-specific functions. 80303 I/O processor programs cannot use this address space except for accesses to memory-mapped registers. The processor does not generate any external bus cycles to this memory. As shown in Figure 3-2, part of the initialization boot record is located just below the 80303 I/O processor's reserved memory.

The 80303 I/O processor requires some special consideration when using the lower 1 Kbyte of address space (addresses 0000H 03FFH). Loads and stores directed to these addresses access internal memory; instruction fetches from these addresses are not allowed by the processor. See Section 4.1, "Internal Data RAM" on page 4-1. No external bus cycles are generated to this address space.

3.5.2 Data and Instruction Alignment in the Address Space

Instructions, program data and architecturally defined data structures can be placed anywhere in non-reserved address space while adhering to these alignment requirements:

- Align instructions on word boundaries.
- Align all architecturally defined data structures on the boundaries specified in Table 3-4.
- Align instruction operands for the atomic instructions (**atadd**, **atmod**) to word boundaries in memory.

The 80303 I/O processor can perform unaligned load or store accesses. The processor handles a non-aligned load or store request by:

- After the access completes, the processor can generate an OPERATION.UNALIGNED fault, if directed to do so.

The method of handling faults is selected at initialization based on the value of the Fault Configuration Word in the Process Control Block. See Section 11.4.2, "Process Control Block – PRCB" on page 11-15.

Table 3-4. Alignment of Data Structures in the Address Space

Data Structure	Alignment Boundary
System Procedure Table	4 byte
Interrupt Table	4 byte
Fault Table	4 byte
Control Table	16 byte
User Stack	16 byte
Supervisor Stack	16 byte
Interrupt Stack	16 byte
Process Control Block	16 byte
Initialization Boot Record	Fixed at FEFF FF30H

3.5.3 Byte, Word and Bit Addressing

The processor provides instructions for moving data blocks of various lengths from memory to registers (**ld**) and from registers to memory (**st**). Supported sizes for blocks are bytes, short words (2 bytes), words (4 bytes), double words, triple words and quad words. For example, **stl** (store long) stores an 8-byte (double word) data block in memory.

The most efficient way to move data blocks longer than 16 bytes is to move them in quad-word increments, using quad-word instructions **ldq** and **stq**.

When a data block is stored in memory, the block's least significant byte is stored at a base memory address and the more significant bytes are stored at successively higher byte addresses. This method of ordering bytes in memory is referred to as "little endian" ordering.

When loading a byte, short word or word from memory to a register, the block's least significant bit is always loaded in register bit 0. When loading double words, triple words and quad words, the least significant word is stored in the base register. The more significant words are then stored at successively higher-numbered registers. Individual bits can be addressed only in data that resides in a register: bit 0 in a register is the least significant bit, bit 31 is the most significant bit.

3.5.4 Internal Data RAM

The 80303 I/O processor has 1 Kbyte of on-chip data RAM. Only data accesses are allowed in this region. Portions of the data RAM can also be reserved for functions such as caching interrupt vectors. The internal RAM is fully described in Chapter 4, "Cache and On-Chip Data RAM".

3.5.5 Instruction Cache

The instruction cache enhances performance by reducing the number of instruction fetches from external memory. The cache provides fast execution of cached code and loops of code in the cache and also provides more bus bandwidth for data operations in external memory. The 80303 I/O processor instruction cache is a 16-Kbyte, two-way set associative cache, organized in two sets of four-word lines.

3.5.6 Data Cache

The data cache on the 80303 I/O processor is a write-through 4-Kbyte direct-mapped cache. For more information, see Chapter 4, "Cache and On-Chip Data RAM".

3.6 Processor-State Registers

The architecture defines four 32-bit registers that contain status and control information:

- Instruction Pointer (IP) register
- Arithmetic Controls (AC) register
- Process Controls (PC) register
- Trace Controls (TC) register

3.6.1 Instruction Pointer (IP) Register

The IP register contains the address of the instruction currently being executed. This address is 32 bits long; however, since instructions are required to be aligned on word boundaries in memory, the IP's two least-significant bits are always 0 (zero).

All i960 processor instructions are either one or two words long. The IP gives the address of the lowest-order byte of the first word of the instruction.

The IP register cannot be read directly. However, the IP-with-displacement addressing mode lets software use the IP as an offset into the address space. This addressing mode can also be used with the **lda** (load address) instruction to read the current IP value.

When a break occurs in the instruction stream due to an interrupt, procedure call or fault, the processor stores the IP of the next instruction to be executed in local register r2, which is usually referred to as the return IP or RIP register. Refer to Chapter 7, "Procedure Calls" for further discussion.

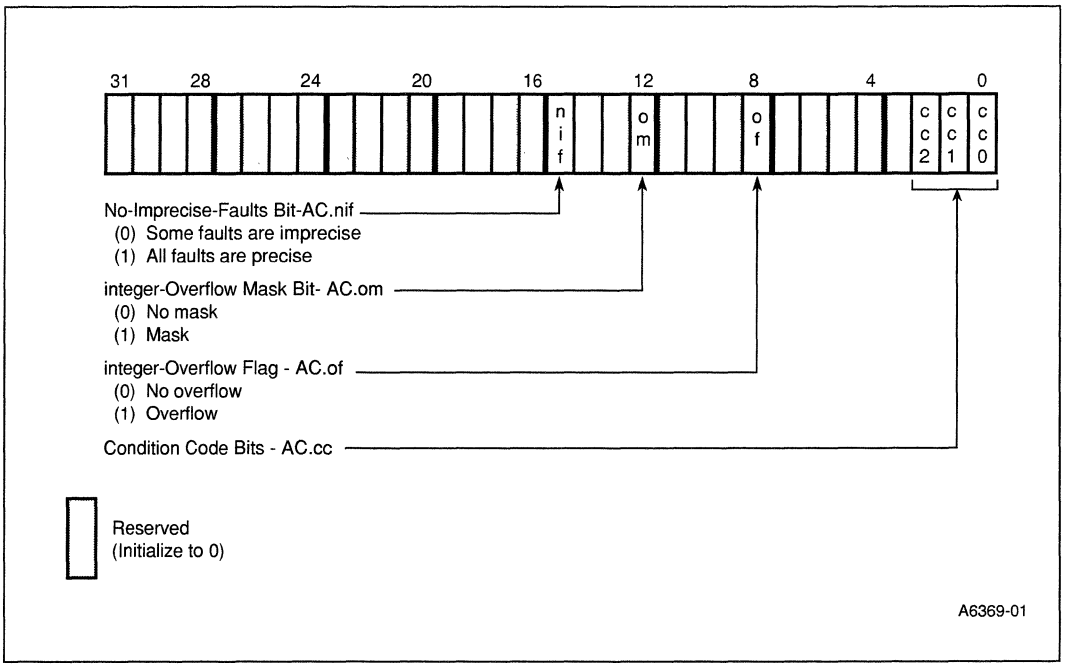
3.6.2 Arithmetic Controls Register – AC

The AC register (Table 3-3) contains:

- condition code flags
- integer overflow flag
- mask bit
- a bit that controls faulting on imprecise faults

Unused AC register bits are reserved.

Figure 3-3. Arithmetic Controls Register – AC



3.6.2.1 Initializing and Modifying the AC Register

At initialization, the AC register is loaded from the Initial AC image field in the Process Control Block. Set reserved bits to 0 in the AC Register Initial Image. Refer to Chapter 11, "Initialization and System Requirements".

After initialization, software must not modify or depend on the AC register's initial image in the PRCB. Software can use the modify arithmetic controls (**modac**) instruction to examine and/or modify any of the register bits. This instruction provides a mask operand that lets user software limit access to the register's specific bits or groups of bits, such as the reserved bits.

The processor automatically saves and restores the AC register when it services an interrupt or handles a fault. The processor saves the current AC register state in an interrupt record or fault record, then restores the register upon returning from the interrupt or fault handler.

3.6.2.2 Condition Code (AC.cc)

The processor sets the AC register *condition code flags* (bits 0-2) to indicate results of certain instructions, such as compare instructions. Other instructions, such as conditional branch instructions, examine these flags and perform functions as dictated by the state of the condition code flags. Once the processor sets the condition code flags, the flags remain unchanged until another instruction executes that modifies the field.

Condition code flags show true/false conditions, inequalities (greater than, equal or less than conditions) or carry and overflow conditions for extended arithmetic instructions. To show true or false conditions, the processor sets the flags as shown in Table 3-5. To show equality and inequalities, the processor sets the condition code flags as shown in Table 3-6.

Table 3-5. Condition Codes for True or False Conditions

Condition Code	Condition
010_2	true
000_2	false

Table 3-6. Condition Codes for Equality and Inequality Conditions

Condition Code	Condition
000_2	unordered
001_2	greater than
010_2	equal
100_2	less than

The term *unordered* is used when comparing floating point numbers. The 80303 I/O processor does not implement on-chip floating point processing.

To show carry out and overflow, the processor sets the condition code flags as shown in Table 3-7.

Table 3-7. Condition Codes for Carry Out and Overflow

Condition Code	Condition
$01X_2$	carry out
$0X1_2$	overflow

Certain instructions, such as the branch-if instructions, use a 3-bit mask to evaluate condition code flags. For example, branch-if-greater-or-equal instruction (**bge**) uses a mask of 011_2 to determine if condition code is set to either greater-than or equal. Conditional instructions use similar masks for remaining conditions (e.g., greater-or-equal [011_2], less-or-equal [110_2] and not-equal [101_2]). The mask is part of the instruction opcode; the instruction performs a bitwise AND of mask and condition code.

The AC register *integer overflow flag* (bit 8) and *integer overflow mask bit* (bit 12) are used in conjunction with the ARITHMETIC.INTEGER_OVERFLOW fault. The mask bit disables fault generation. When the fault is masked and integer overflow is encountered, the processor sets the integer overflow flag instead of generating a fault. If the fault is not masked, the fault is allowed to occur and the flag is not set.

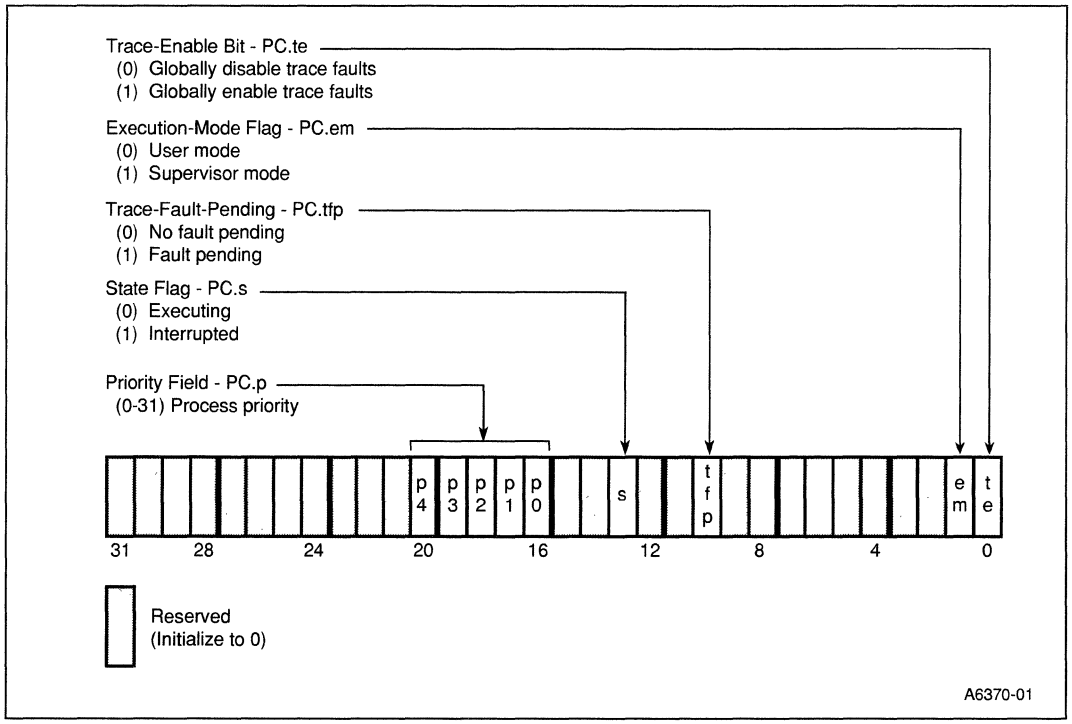
Once the processor sets this flag, the flag remains set until the application software clears it. Refer to the discussion of the ARITHMETIC.INTEGER_OVERFLOW fault in Chapter 9, "Faults" for more information about the integer overflow mask bit and flag.

The *no imprecise faults (AC.nif) bit* (bit 15) determines whether or not faults are allowed to be imprecise. If set, all faults are required to be precise; if clear, certain faults can be imprecise. See Section 9.9, "Precise and Imprecise Faults" on page 9-20 for more information.

3.6.3 Process Controls Register – PC

The PC register (Table 3-8) is used to control processor activity and show the processor's current state. The PC register *execution mode flag* (bit 1) indicates that the processor is operating in either user mode (0) or supervisor mode (1). The processor automatically sets this flag on a system call when a switch from user mode to supervisor mode occurs and it clears the flag on a return from supervisor mode. (User and supervisor modes are described in Section 3.7, "User-Supervisor Protection Model" on page 3-18.

Table 3-8. Process Controls Register – PC



PC register *state flag* (bit 13) indicates the processor state: executing (0) or interrupted (1). If the processor is servicing an interrupt, its state is interrupted. Otherwise, the processor's state is executing.

While in the interrupted state, the processor can receive and handle additional interrupts. When nested interrupts occur, the processor remains in the interrupted state until all interrupts are handled, then switches back to the executing state on the return from the initial interrupt procedure.

The PC register *priority field* (bits 16 through 20) indicates the processor's current executing or interrupted priority. The architecture defines a mechanism for prioritizing execution of code, servicing interrupts and servicing other implementation-dependent tasks or events. This mechanism defines 32 priority levels, ranging from 0 (the lowest priority level) to 31 (the highest). The priority field always reflects the current priority of the processor. Software can change this priority by use of the **modpc** instruction.

The processor uses the priority field to determine whether to service an interrupt immediately or to post the interrupt. The processor compares the priority of a requested interrupt with the current process priority. When the interrupt priority is greater than the current process priority or equal to 31, the interrupt is serviced; otherwise it is posted. When an interrupt is serviced, the process priority field is automatically changed to reflect interrupt priority. See Chapter 8, "PCI and Peripheral Interrupt Controller Unit".

The PC register *trace enable bit* (bit 0) and *trace fault pending flag* (bit 10) control the tracing function. The trace enable bit determines whether trace faults are globally enabled (1) or globally disabled (0). The trace fault pending flag indicates that a trace event has been detected (1) or not detected (0). The tracing functions are further described in Chapter 10, "Tracing and Debugging".

3.6.3.1 Initializing and Modifying the PC Register

Any of the following three methods can be used to change bits in the PC register:

- Modify process controls instruction (**modpc**)
- Alter the saved process controls prior to a return from an interrupt handler or fault handler

The **modpc** instruction reads and modifies the PC register directly. A TYPE.MISMATCH fault results if software executes **modpc** in user mode with a non-zero mask. As with **modac**, **modpc** provides a mask operand that can be used to limit access to specific bits or groups of bits in the register. In user mode, software can use **modpc** to read the current PC register.

In the latter two methods, the interrupt or fault handler changes process controls in the interrupt or fault record that is saved on the stack. Upon return from the interrupt or fault handler, the modified process controls are copied into the PC register. The processor must be in supervisor mode prior to return for modified process controls to be copied into the PC register.

When process controls are changed as described above, the processor recognizes the changes immediately except for one situation: if **modpc** is used to change the trace enable bit, the processor may not recognize the change before the next four non-branch instructions are executed.

After initialization (hardware reset), the process controls reflect the following conditions:

- priority = 31
- execution mode = supervisor
- trace enable = disabled
- state = interrupted
- no trace fault pending

When the processor is reinitialized with a **sysctl** reinitialize message, the PC register is not changed.

Software should not use **modpc** to modify execution mode or trace fault state flags except under special circumstances, such as in initialization code. Normally, execution mode is changed through the call and return mechanism. See Section 6.2.43, "modpc" on page 6-64 for more details.

3.6.4 Trace Controls (TC) Register

The TC register, in conjunction with the PC register, controls processor tracing facilities. It contains trace mode enable bits and trace event flags that are used to enable specific tracing modes and record trace events, respectively. Trace controls are described in Chapter 10, "Tracing and Debugging".

3.7 User-Supervisor Protection Model

The processor can be in either of two execution modes: user or supervisor. The capability of a separate user and supervisor execution mode creates a code and data protection mechanism referred to as the user-supervisor protection model. This mechanism allows code, data and stack for a kernel (or system executive) to reside in the same address space as code, data and stack for the application. The mechanism restricts access to all or parts of the kernel by the application code. This protection mechanism prevents application software from inadvertently altering the kernel.

3.7.1 Supervisor Mode Resources

Supervisor mode is a privileged mode that provides several additional capabilities over user mode.

- When the processor switches to supervisor mode, it also switches to the supervisor stack. Switching to the supervisor stack helps maintain a kernel's integrity. For example, it allows access to system debugging software or a system monitor, even if an application's program destroys its own stack.
- In supervisor mode, the processor is allowed access to a set of supervisor-only functions and instructions. For example, the processor uses supervisor mode to handle interrupts and trace faults. Operations that can modify interrupt controller behavior or reconfigure bus controller characteristics can be performed only in supervisor mode. These functions include modification of control registers and internal data RAM that is dedicated to interrupt controllers. A fault is generated if supervisor-only operations are attempted while the processor is in user mode.

The PC register execution mode flag specifies processor execution mode. The processor automatically sets and clears this flag when it switches between the two execution modes.

- **dcctl** (data cache control)
- **inten** (global interrupt enable)
- Protected timer unit registers
- **modpc** (modify process controls w/ non-zero mask)
- **icctl** (instruction cache control)
- **sysctl** (system control)
- **intctl** (global interrupt enable and disable)
- Protected internal data RAM or Supervisor MMR space write
- **intdis** (global interrupt disable)

Note that all of these instructions return a TYPE.MISMATCH fault if executed in user mode.

3.7.2 Using the User-Supervisor Protection Model

A program switches from user mode to supervisor mode by making a system-supervisor call (also referred to as a supervisor call). A system-supervisor call is a call executed with the call-system instruction (**calls**). With **calls**, the IP for the called procedure comes from the system procedure table. An entry in the system procedure table can specify an execution mode switch to supervisor mode when the called procedure is executed. **calls** and the system procedure table thus provide a tightly controlled interface to procedures that can execute in supervisor mode. Once the processor switches to supervisor mode, it remains in that mode until a return is performed to the procedure that caused the original mode switch.

Interrupts and faults can cause the processor to switch from user to supervisor mode. When the processor handles an interrupt, it automatically switches to supervisor mode. However, it does not switch to the supervisor stack. Instead, it switches to the interrupt stack. Fault table entries determine if a particular fault transitions the processor from user to supervisor mode.

If an application does not require a user-supervisor protection mechanism, the processor can always execute in supervisor mode. At initialization, the processor is placed in supervisor mode prior to executing the first instruction of the application code. The processor then remains in supervisor mode indefinitely, as long as no action is taken to change execution mode to user mode. The processor does not need a user stack in this case.

Cache and On-Chip Data RAM 4

This chapter describes the structure and user configuration of all forms of on-chip storage, including caches (data, local register and instruction) and data RAM.

4.1 Internal Data RAM

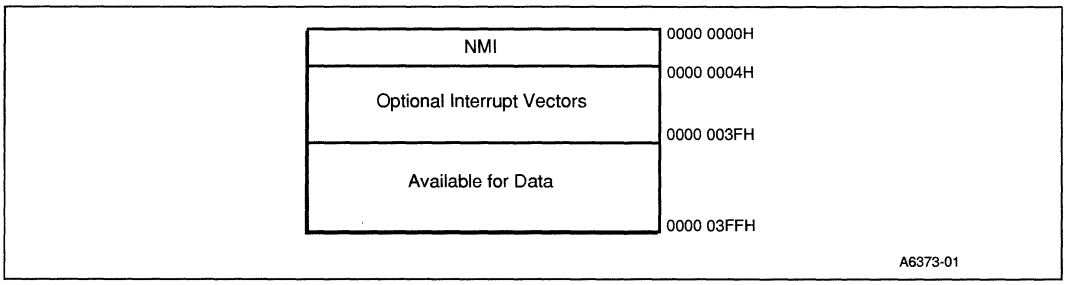
Internal data RAM is mapped to the lower 1 Kbyte (0 to 03FFH) address space. Loads and stores with target addresses in internal data RAM operate directly on internal data RAM; no external bus activity is generated. Data RAM allows time-critical data storage and retrieval without dependence on external bus performance. Only data accesses are allowed to internal data RAM; instructions cannot be fetched from internal data RAM. Instruction fetches directed to data RAM cause an OPERATION.UNIMPLEMENTED fault to occur.

Internal data RAM locations are never cached in the data cache. Logical Memory Template bits controlling caching are ignored for data RAM accesses.

Some internal data RAM locations are reserved for functions other than general data storage. The first 64 bytes of data RAM may be used to cache interrupt vectors, which reduces interrupt latency. The word at 0000H is always reserved for cached NMI vector. With the exception of the cached NMI vector, other reserved portions of data RAM can be used for data storage when the alternate function is not used. All locations of internal data RAM can be read in both supervisor and user mode.

The first 64 bytes (0000H to 003FH) of internal RAM are always user-mode write-protected. This portion of data RAM can be read while executing in user or supervisor mode; however, it can be only modified in supervisor mode. This area can also be write-protected from supervisor mode writes by setting the BCON.sirp bit. See Section 12.2.2, "Bus Control Register – BCON" on page 12-4. Protecting this portion of the data RAM from user and supervisor rights preserves the interrupt vectors that may be cached there. See Section 8.6.3.1, "Vector Caching Option" on page 8-30.

Figure 4-1. Internal Data RAM and Register Cache



The remainder of the internal data RAM can always be written from supervisor mode. User mode write protection is optionally selected for the rest of the data RAM (40H to 3FFH) by setting the Bus Control Register RAM protection bit (BCON.irp). Writes to internal data RAM locations while they are protected generate a TYPE.MISMATCH fault. See Section 12.2.2, "Bus Control Register – BCON" on page 12-4 for the format of the BCON register.

New versions of the Intel® i960® processor compilers take advantage of internal data RAM. Profiling compilers, such as those offered by Intel, can allocate the most frequently used variables into this RAM.

4.2 Local Register Cache

The Intel® 80303 I/O processor provides fast storage of local registers for call and return operations by using an internal local register cache (also known as a *stack frame cache*). Up to eight local register sets can be contained in the cache before sets must be saved in external memory. The register set is all the local registers (i.e., r0 through r15). The processor uses a 128-bit wide bus to store local register sets quickly to the register cache. An integrated procedure call mechanism saves the current local register set when a call is executed. A local register set is saved into a frame in the local register cache, one frame per register set. When the eighth frame is saved, the oldest set of local registers is flushed to the procedure stack in external memory, which frees one frame.

Section 7.1.4, "Caching Local Register Sets" on page 7-7 and Section 7.1.5, "Mapping Local Registers to the Procedure Stack" on page 7-11 further discuss the relationship between the internal register cache and the external procedure stack.

The branch-and-link (**bal** and **balx**) instructions do not cause the local registers to be stored.

The entire internal register cache contents can be copied to the external procedure stack through the flushreg instruction. Section 6.2.30, "flushreg" on page 6-44 explains the instruction itself and Section 7.2, "Modifying the PFP Register" on page 7-12 offers a practical example when flushreg must be used.

To decrease interrupt latency, software can reserve a number of frames in the local register cache solely for high priority interrupts (interrupted state and process priority greater than or equal to 28). The remaining frames in the cache can be used by all code, including high-priority interrupts. When a frame is reserved for high-priority interrupts, the local registers of the code interrupted by a high-priority interrupt can be saved to the local register cache without causing a frame flush to memory, providing the local register cache is not already full. Thus, the register allocation for the implicit interrupt call does not incur the latency of a frame flush.

Software can reserve frames for high-priority interrupt code by writing bits 10 through 8 of the register cache configuration word in the PRCB. This value indicates the number of free frames within the register cache that can be used by high-priority interrupts only. Any attempt by non-critical code to reduce the number of free frames below this value results in a frame flush to external memory. The free frame check is performed only when a frame is pushed, which occurs only for an implicit or explicit call. The following pseudo-code illustrates the operation of the register cache when a frame is pushed.

Example 4-1. Register Cache Operation

```
frames_for_non_critical = 7- RCW[11:8];
if (interrupt_request)
    set_interrupt_handler_PC;
push_frame;
number_of_frames = number_of_frames + 1;
if (number_of_frames = 8) {
         flush_register_frame(oldest_frame);
         number_of_frames = number_of_frames - 1; }
else if (number_of_frames = (frames_for_non_critical + 1) &&
         (PC.priority < 28 || PC.state != interrupted) ) {
         flush_register_frame(oldest_frame);
         number_of_frames = number_of_frames - 1; }
```

The valid range for the number of reserved free frames is 0 to 7. Setting the value to 0 reserves no frames for exclusive use by high-priority interrupts. Setting the value to 1 reserves 1 frame for high-priority interrupts and 6 frames to be shared by all code. Setting the value to 7 causes the register cache to become disabled for non-critical code. If the number of reserved high-priority frames exceeds the allocated size of the register cache, the entire cache is reserved for high-priority interrupts. In that case, all low-priority interrupts and procedure calls cause frame spills to external memory.

4.3 Instruction Cache

The 80303 I/O processor features a 16-Kbyte, 2-way set-associative instruction cache (I-cache) organized in lines of four 32-bit words. The cache provides fast execution of cached code and loops of code and provides more bus bandwidth for data operations in external memory. To optimize cache updates when branches or interrupts are executed, each word in the line has a separate valid bit. When requested instructions are found in the cache, the instruction fetch time is one cycle for up to four words. A mechanism to load and lock critical code within a way of the cache is provided along with a mechanism to disable the cache. The cache is managed through the **icctl** or **sysctl** instruction. The **sysctl** instruction supports the instruction cache to maintain compatibility with other i960 processor software. Using **icctl** is the preferred and more versatile method for controlling the instruction cache on the 80303 I/O processor.

Cache misses cause the processor to issue a double-word or a quad-word fetch, based on the location of the Instruction Pointer:

- If the IP is at word 0 or word 1 of a 16-byte block, a four-word fetch is initiated.
- If the IP is at word 2 or word 3 of a 16-byte block, a two-word fetch is initiated.

4.3.1 Enabling and Disabling the Instruction Cache

Enabling the instruction cache is controlled on reset or initialization by the instruction cache configuration word in the Process Control Block (PRCB); see Table 11-8, "Process Control Block Configuration Words" on page 11-16. When bit 16 in the instruction cache configuration word is set, the instruction cache is disabled and all instruction fetches are directed to external memory. Disabling the instruction cache is useful for tracing execution in a software debug environment.

The instruction cache remains disabled until one of three operations is performed:

- **icctl** is issued with the enable instruction cache operation (preferred method)
- **sysctl** is issued with the configure-instruction-cache message type and cache configuration mode other than disable cache (provides compatibility with other i960 processors; not the preferred method for the 80303 I/O processor).
- The processor is reinitialized with a new value in the instruction cache configuration word

4.3.2 Operation While the Instruction Cache Is Disabled

Disabling the instruction cache *does not* disable instruction buffering that may occur in the instruction fetch unit. A four-word instruction buffer is always enabled, even when the cache is disabled.

There is one tag and four word-valid bits associated with the buffer. Because there is only one tag for the buffer, any "miss" within the buffer causes the following:

- All four words of the buffer are invalidated.
- A new tag value for the required instruction is loaded.
- The required instruction(s) are fetched from external memory.

Depending on the alignment of the "missed" instruction, either two or four words of instructions are fetched and only the valid bits corresponding to the fetched words are set in the buffer. No external instruction fetches are generated until there is a "miss" within the buffer, even in the presence of forward and backward branches.

4.3.3 Loading and Locking Instructions in the Instruction Cache

The processor can be directed to load a block of instructions into the cache and then lock out all normal updates to the cache. This cache load-and-lock mechanism is provided to minimize latency on program control transfers to key operations such as interrupt service routines. The block size that can be loaded and locked on the 80303 I/O processor is one way of the cache.

An **icctl** or **sysctl** instruction is issued with a configure-instruction-cache message type to select the load-and-lock mechanism. When the lock option is selected, the processor loads the cache starting at an address specified as an operand to the instruction.

4.3.4 Instruction Cache Visibility

Instruction cache status can be determined by issuing **icctl** with an instruction-cache status message. To facilitate debugging, the instruction cache contents, instructions, tags and valid bits can be written to memory. This is done by issuing **icctl** with the store cache operation.

4.3.5 Instruction Cache Coherency

The 80303 I/O processor does not snoop the bus to prevent instruction cache incoherency. The cache does not detect modification to program memory by loads, stores or actions of other bus masters. Several situations may require program memory modification, such as uploading code at initialization or loading from a backplane bus or a disk drive.

The application program is responsible for synchronizing its own code modification and cache invalidation. In general, a program must ensure that modified code space is not accessed until modification and cache-invalidate are completed. To achieve cache coherency, instruction cache contents should be invalidated after code modification is complete. **icctl** invalidates the instruction cache for the 80303 I/O processor. Alternatively, i960 processor legacy software can use **sysctl**.

4.4 Data Cache

The 80303 I/O processor features a 4-Kbyte, direct-mapped cache that enhances performance by reducing the number of data load and store accesses to external memory. The cache is write-through and write-allocate. It has a line size of 4 words and each line in the cache has a valid bit. To reduce fetch latency on cache misses, each word within a line also has a valid bit. Caches are managed through the **dcctl** instruction.

User settings in the memory region configuration registers LMCON0-1 and DLMCON determine the data accesses that are cacheable or non-cacheable based on memory region.

4.4.1 Enabling and Disabling the Data Cache

To cache data, two conditions must be met:

1. The data cache must be enabled. A **dcctl** instruction issued with an enable data cache message enables the cache. On reset or initialization, the data cache is always disabled and all valid bits are set to zero.
2. Data caching for a location must be enabled by the corresponding logical memory template, or by the default logical memory template if no other template applies. See Section 12.2.2, "Bus Control Register – BCON" on page 12-4 for more details on logical memory templates.

When the data cache is disabled, all data fetches are directed to external memory. Disabling the data cache is useful for debugging or monitoring a system. To disable the data cache, issue a **dcctl** with a disable data cache message. The enable and disable status of the data cache and various attributes of the cache can be determined by a **dcctl** issued with a data-cache status message.

4.4.2 Multi-Word Data Accesses that Partially Hit the Data Cache

The following applies only when data caching is enabled for an access.

For a multi-word load access (**ldl**, **ldt**, **ldq**) in which none of the requested words hit the data cache, an external bus transaction is started to acquire all the words of the access.

For a multi-word load access that partially hits the data cache, the processor may either:

- Load or reload all words of the access (even those that hit) from the external bus.
- Load only missing words from the external bus and interleave them with words found in the data cache.

The multi-word alignment determines which of the above methods is used:

- Naturally aligned multi-word accesses cause all words to be reloaded.
- An unaligned multi-word access causes only missing words to be loaded.

When any words (Table 4-1) accessed with **ldl**, **ldt**, or **ldq** miss the data cache, every word accessed by that load instruction is updated in the cache.

Table 4-1. Load Instruction Updates

Load Instruction	Number of Updated Words
ldq	4 words
ldt	3 words
ldl	2 words

In each case, the external bus accesses used to acquire the data may consist of none, one, or several burst accesses based on the alignment of the data and the bus-width of the memory region that contains the data. See Chapter 12, "Core Processor and Internal Operation" for more details.

A multi-word load access that completely hits in the data cache does not cause external bus accesses.

For a multi-word store access (**stl**, **stt**, **stq**) an external bus transaction is started to write all words of the access regardless if any or all words of the access hit the data cache. External bus accesses used to write the data may consist of either one or several burst accesses based on data alignment and the bus-width of the memory region that receives the data. The cache is also updated accordingly as described earlier in this chapter.

4.4.3 Data Cache Fill Policy

The 80303 I/O processor always uses a "natural" fill policy for cacheable loads. The processor fetches only the amount of data that is requested by a load (i.e., a word, long word, etc.) on a data cache miss. Exceptions are byte and short-word accesses, which are always promoted to words. This allows a complete word to be brought into the cache and marked valid. When the data cache is disabled and loads are done from a cacheable region, promotions from bytes and short words still take place.

4.4.4 Data Cache Write Policy

The write policy determines the action taken on cacheable writes (stores). The 80303 I/O processor always uses a write-through policy. Stores are always seen on the external bus, thus maintaining coherency between the data cache and external memory.

The 80303 I/O processor always uses a write-allocate policy for data. For a cacheable location, data is always written to the data cache regardless of whether the access is a hit or miss. The following cases are relevant to consider:

1. In the case of a hit for a word or multi-word store, the appropriate line and word(s) are updated with the data.
2. In the case of a miss for a word or multi-word store, a tag and cache line are allocated, if needed, and the appropriate valid bits, line, and word(s) are updated.
3. In the case of byte or short-word data that hits a valid word in the cache, both the word in cache and external memory are updated with the data; the cache word remains valid.
4. In the case of byte or short-word data that falls within a valid line but misses because the appropriate word is invalid, both the word and external memory are updated with the data; however, the cache word remains invalid.
5. In the case of byte or short-word data that does not fall within a valid line, the external memory is updated with the data. For data writes less than a word, the data cache is not updated; the tags and valid bits are not changed.

A byte or short word is always invalid in the data cache since valid bits only apply to words.

For cacheable stores that are equal to or greater than a word in length, cache tags and appropriate valid bits are updated whenever data is written into the cache. Consider a word store that misses as an example. The tag is always updated and its valid bit is set. The appropriate valid bit for that word is always set and the other three valid bits are always cleared. If the word store hits the cache, the tag bits remain unchanged. The valid bit for the stored word is set; all other valid bits are unchanged.

Cacheable stores that are less than a word in length are handled differently. Byte and short-word stores that hit the cache (i.e., are contained in valid words within valid cache lines) do not change the tag and valid bits. The processor writes the data into the cache and external memory as usual. A byte or short-word store to an invalid word within a valid cache line leaves the word valid bit cleared because the rest of the word is still invalid. In these two cases the processor simultaneously writes the data into the cache and the external memory.

4.4.5 Data Cache Coherency and Non-Cacheable Accesses

The 80303 I/O processor ensures that the data cache is always kept coherent with accesses that it initiates and performs. The most visible application of this requirement concerns non-cacheable accesses discussed below. However, the processor does not provide data cache coherency for accesses on the external bus that it did not initiate. Software is responsible for maintaining coherency in a multi-processor environment.

An access is defined as non-cacheable when any of the following is true:

1. The access falls into an address range mapped by an enabled LMCON or DLMCON and the data-caching enabled bit in the matching LMCON is clear.
2. The entire data cache is disabled.
3. The access is a read operation of the read-modify-write sequence performed by an **atmod** or **atadd** instruction.
4. The access is an implicit read access to the interrupt table to post or deliver a software interrupt.

If the memory location targeted by an **atmod** or **atadd** instruction is currently in the data cache, it is invalidated.

If the address for a non-cacheable store matches a tag ("tag hit"), the corresponding cache line is marked invalid. This is because the word is not actually updated with the value of the store. This behavior ensures that the data cache never contains stale data in a single-processor system. A simple case illustrates the necessity of this behavior: a read of data previously stored by a non-cacheable access must return the new value of the data, not the value in the cache. Because the processor invalidates the appropriate word in the cache line on a store hit when the cache is disabled, coherency can be maintained when the data cache is enabled and disabled dynamically.

Data loads or stores invalidate the corresponding lines of the cache even when data caching is disabled. This behavior further ensures that the cache does not contain stale data.

4.4.6 External I/O Bus Masters and Cache Coherency

The 80303 I/O processor implements a single processor coherency mechanism. There is no hardware mechanism, such as bus snooping, to support multiprocessing. If another bus master can change shared memory, there is no guarantee that the data cache contains the most recent data. The user must manage such data coherency issues in software.

A suggested practice is to program the LMCON0-1 registers such that I/O regions are non-cacheable. Partitioning the system in this fashion eliminates I/O as a source of coherency problems. See Section 12.2.1, "PMCON Registers" on page 12-2 for more information on this subject.

4.4.7 Data Cache Visibility

Data cache status can be determined by a **dcctl** instruction issued with a data-cache status message. Data cache contents, data, tags and valid bits can be written to memory as an aid for debugging. This operation is accomplished by a **dcctl** instruction issued with the dump cache operand. See Section 6.2.23, "dcctl" on page 6-33 for more information.

Instruction Set Overview 5

This chapter provides an overview of the Intel® i960® microprocessor family instruction set and Intel® 80303 I/O processor-specific instruction set extensions. Also discussed are the assembly-language and instruction-encoding formats, various instruction groups and each group's instructions.

Chapter 6, "Instruction Set Reference" describes each instruction, including assembly language syntax, and the action taken when the instruction executes and examples of how to use the instruction.

5.1 Instruction Formats

80303 I/O processor instructions may be described in two formats: assembly language and instruction encoding. The following subsections briefly describe these formats.

5.1.1 Assembly Language Format

Throughout this manual, instructions are referred to by their assembly language mnemonics. For example, the add ordinal instruction is referred to as **addo**. Examples use Intel 80303 assembly language syntax which consists of the instruction mnemonic followed by zero to three operands, separated by commas. In the following assembly language statement example for **addo**, ordinal operands in global registers g5 and g9 are added together, and the result is stored in g7:

```
addo g5, g9, g7# g7 = g9 + g5
```

In the assembly language listings in this chapter, registers are denoted as:

g	global register
r	local register
#	pound sign precedes a comment

All numbers used as literals or in address expressions are assumed to be decimal. Hexadecimal numbers are denoted with a "0x" prefix (e.g., 0xffff0012). Several assembly language instruction statement examples follow. Additional assembly language examples are given in Section 2.3.5, "Addressing Mode Examples" on page 2-6.

Example 5-1. Assembly Language Instruction Statement Examples

```
subi r3, r5, r6              #r6 = r5 - r3
setbit 13, g4, g5            #g5 = g4 with bit 13 set
lda 0xfab3, r12              #r12 = 0xfab3
ld (r4), g3                  #g3 = memory location that r4 points to
st g10, (r6)[r7*2]           #g10 = memory location that r6+2*r7 points to
```

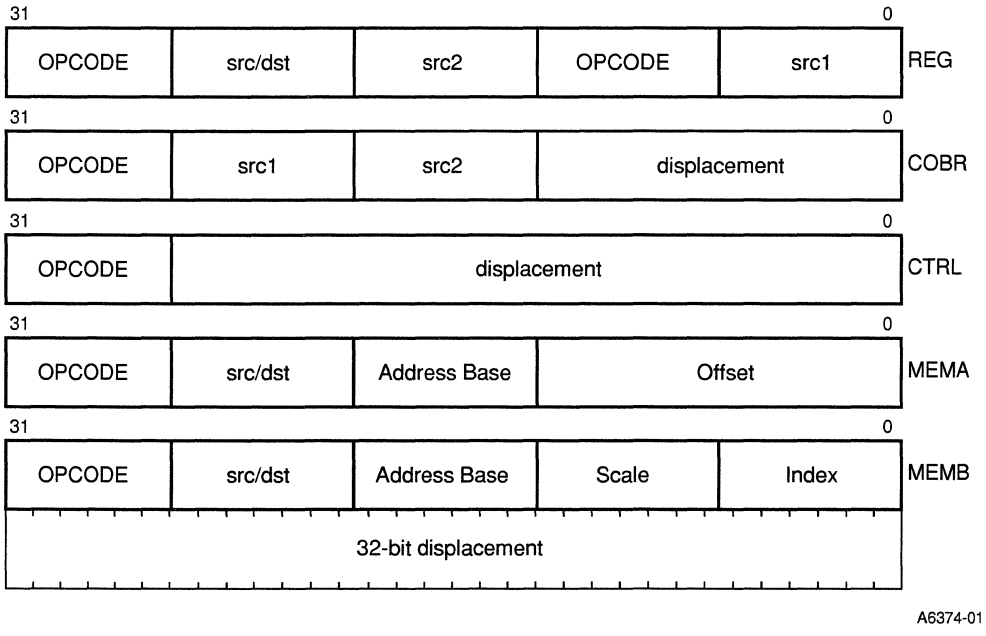
5.1.2 Instruction Encoding Formats

All instructions are encoded in one 32-bit machine language instruction — an *opword* — which must be word aligned in memory. An opword's most significant eight bits contain the opcode field. The opcode field determines the instruction to be performed and how the remainder of the machine language instruction is interpreted. Instructions are encoded in opwords in one of four formats (see Figure 5-1). For more information on instruction formats, see Appendix A, "Machine-Level Instruction Formats".

Table 5-1. Instruction Encoding Formats (REG, COBR, CRTL, MEM)

Instruction Type	Format	Description
register	REG	Most instructions are encoded in this format. Used primarily for instructions which perform register-to-register operations.
compare and branch	COBR	An encoding optimization which combines compare and branch operations into one opword. Other compare and branch operations are also provided as REG and CTRL format instructions.
control	CTRL	For branches and calls that do not depend on registers for address calculation.
memory	MEM	Used for referencing an operand which is a memory address. Load and store instructions — and some branch and call instructions — use this format. MEM format has two encodings: MEMA or MEMB. Usage depends upon the addressing mode selected. MEMB-formatted addressing modes use the word in memory immediately following the instruction opword as a 32-bit constant. MEMA format uses one word and MEMB uses two words.

Figure 5-1. Machine-Level Instruction Formats



5.1.3 Instruction Operands

This section identifies and describes operands that can be used with the instruction formats.

Table 5-2. Operands and Instruction Formats

Format	Operand(s)	Description
REG	*src1*, *src2*, *src/dst*	*src1* and *src2* can be global registers, local registers or literals. *src/dst* is either a global or a local register.
CTRL	*displacement*	CTRL format is used for branch and call instructions. *displacement* value indicates the target instruction of the branch or call.
COBR	*src1*, *src2*, *displacement*	*src1*, *src2* indicate values to be compared; *displacement* indicates branch target. *src1* can specify a global register, local register or a literal. *src2* can specify a global or local register.
MEM	*src/dst*, *efa*	Specifies source or destination register and an effective address (*efa*) formed by using the processor's addressing modes as described in Section 2.3, "Memory Addressing Modes" on page 2-4. Registers specified in a MEM format instruction must be either a global or local register.

5.2 Instruction Groups

The i960 processor instruction set can be categorized into the following functional groups shown in Table 5-3. The actual number of instructions is greater than those shown in this list because, for some operations, several unique instructions are provided to handle various operand sizes, data types or branch conditions. The following sections provide an overview of the instructions in each group. For detailed information about each instruction, refer to Chapter 6, "Instruction Set Reference".

Table 5-3. Intel® 80303 I/O Processor Instruction Set

Data Movement	Arithmetic	Logical	Bit, Bit Field and Byte
Load Store Move *Conditional Select Load Address	Add Subtract Multiply Divide Remainder Modulo Shift Extended Shift Extended Multiply Extended Divide Add with Carry Subtract with Carry *Conditional Add *Conditional Subtract Rotate	And Not And And Not Or Exclusive Or Not Or Or Not Nor Exclusive Nor Not Nand	Set Bit Clear Bit Not Bit Alter Bit Scan For Bit Span Over Bit Extract Modify Scan Byte for Equal *Byte Swap
Comparison	**Branch**	**Call/Return**	**Fault**
Compare Conditional Compare Compare and Increment Compare and Decrement Test Condition Code Check Bit	Unconditional Branch Conditional Branch Compare and Branch	Call Call Extended Call System Return Branch and Link	Conditional Fault Synchronize Faults
Debug	**Processor Management**	**Atomic**	
Modify Trace Controls Mark Force Mark	Flush Local Registers Modify Arithmetic Controls Modify Process Controls *Halt System Control *Cache Control *Interrupt Control	Atomic Add Atomic Modify	

* Denotes newer instructions that are NOT available on 80960CA/CF, 80960KA/KB and 80960SA/SB implementations.

5.2.1 Data Movement

These instructions are used to move data from memory to global and local registers, from global and local registers to memory, and between local and global registers.

Rules for register alignment must be followed when using load, store and move instructions that move 8, 12 or 16 bytes at a time. See Section 3.5, "Memory Address Space" on page 3-10 for alignment requirements for code portability across implementations.

5.2.1.1 Load and Store Instructions

Load instructions copy bytes or words from memory to local or global registers or to a group of registers. Each load instruction has a corresponding store instruction to memory bytes or words to copy from a selected local or global register or group of registers. All load and store instructions use the MEM format.

Table 5-4. Load and Store Instructions

Instruction	Description	Instruction	Description
ld	load word	**st**	store word
ldob	load ordinal byte	**stob**	store ordinal byte
ldos	load ordinal short	**stos**	store ordinal short
ldib	load integer byte	**stib**	store integer byte
ldis	load integer short	**stis**	store integer short
ldl	load long	**stl**	store long
ldt	load triple	**stt**	store triple
ldq	load quad	**stq**	store quad

ld copies 4 bytes from memory into a register; **ldl** copies 8 bytes; **ldt** copies 12 bytes into successive registers; **ldq** copies 16 bytes into successive registers.

st copies 4 bytes from a register into memory; **stl** copies 8 bytes; **stt** copies 12 bytes from successive registers; **stq** copies 16 bytes from successive registers.

For **ld**, **ldob**, **ldos**, **ldib** and **ldis**, the instruction specifies a memory address and register; the memory address value is copied into the register. The processor automatically extends byte and short (half-word) operands to 32 bits according to data type. Ordinals are zero-extended; integers are sign-extended.

For **st**, **stob**, **stos**, **stib** and **stis**, the instruction specifies a memory address and register; the register value is copied into memory. For byte and short instructions, the processor automatically reformats the source register's 32-bit value for the shorter memory location. For **stib** and **stis**, this reformatting can cause integer overflow when the register value is too large for the shorter memory location. When integer overflow occurs, either an integer-overflow fault is generated or the integer-overflow flag in the AC register is set, depending on the integer-overflow mask bit setting in the AC register.

For **stob** and **stos**, the processor truncates the register value and does not create a fault when truncation resulted in the loss of significant bits.

5.2.1.2 Move

Move instructions copy data from a local or global register or group of registers to another register or group of registers. These instructions use the REG format.

Table 5-5. Move Instructions

Instriction	Description	Instriction	Description
mov	move word	**movt**	move triple word
movl	move long word	**movq**	move quad word

5.2.1.3 Load Address

The Load Address instruction (**lda**) computes an effective address in the address space from an operand presented in one of the addressing modes. **lda** is commonly used to load a constant into a register. This instruction uses the MEM format and can operate upon local or global registers.

On the 80303 I/O processor, **lda** is useful for performing simple arithmetic operations. The processor's parallelism allows **lda** to execute in the same clock as another arithmetic or logical operation.

5.2.2 Select Conditional

Given the proper condition code bit settings in the Arithmetic Controls register, these instructions move one of two pieces of data from its source to the specified destination.

Table 5-6. Select Condition Instructions

Instruction	Description	Instruction	Description
selno	Select Based on Unordered	**sell**	Select Based on Less
selg	Select Based on Greater	**selne**	Select Based on Not Equal
sele	Select Based on Equal	**selle**	Select Based on Less or Equal
selge	Select Based on Greater or Equal	**selo**	Select Based on Ordered

5.2.3 Arithmetic

Table 5-7 lists arithmetic operations and data types for which the 80303 I/O processor provides instructions. "X" in this table indicates that the microprocessor provides an instruction for the specified operation and data type. All arithmetic operations are carried out on operands in registers or literals. Refer to Section 5.2.11, "Atomic Instructions" on page 5-17 for instructions which handle specific requirements for in-place memory operations.

All arithmetic instructions use the REG format and can operate on local or global registers. The following subsections describe arithmetic instructions for ordinal and integer data types.

Table 5-7. Arithmetic Operations

Arithmetic Operations	Data Types	
	Integer	Ordinal
Add	X	X
Add with Carry	X	X
Conditional Add	X	X
Subtract	X	X
Subtract with Carry	X	X
Conditional Subtract	X	X
Multiply	X	X
Extended Multiply		X
Divide	X	X
Extended Divide		X
Remainder	X	X
Modulo	X	
Shift Left	X	X
Shift Right	X	X
Extended Shift Right		X
Shift Right Dividing Integer	X	

NOTE: "X" indicates that an instruction is available for the specified operation and data type.

5.2.3.1 Add, Subtract, Multiply, Divide, Conditional Add, Conditional Subtract

These instructions perform add, subtract, multiply or divide operations on integers and ordinals:

Table 5-8. Add, Subtract, Multiply, Divide, Conditional Add, Conditional Subtract Instructions

Instruction	Description	Instruction	Description
addi	Add Integer	**muli**	Multiply Integer
addo	Add Ordinal	**mulo**	Multiply Ordinal
subi	Subtract Integer	**divi**	Divide Integer
subo	Subtract Ordinal	**divo**	Divide Ordinal
SUB<cc>	Conditional Subtract		

addi, **ADDI<cc>**, **subi**, **SUBI<cc>**, **muli** and **divi** generate an integer-overflow fault when the result is too large to fit in the 32-bit destination. **divi** and **divo** generate a zero-divide fault when the divisor is zero.

5.2.3.2 Remainder and Modulo

These instructions divide one operand by another and retain the remainder of the operation:

Table 5-9. Remainder and Modulo Instructions

Instruction	Description	Instruction	Description
remi	remainder integer	**modi**	modulo integer
remo	remainder ordinal		

The difference between the remainder and modulo instructions lies in the sign of the result. For **remi** and **remo**, the result has the same sign as the dividend; for **modi**, the result has the same sign as the divisor.

5.2.3.3 Shift, Rotate and Extended Shift

These shift instructions shift an operand a specified number of bits left or right:

Table 5-10. Shift, Rotate and Extended Shift Instructions

Instruction	Description	Instruction	Description
shlo	shift left ordinal	**shrdi**	shift right dividing integer
shro	shift right ordinal	**rotate**	rotate left
shli	shift left integer	**eshro**	extended shift right ordinal
shri	shift right integer		

Except for **rotate**, these instructions discard bits shifted beyond the register boundary.

shlo shifts zeros in from the least significant bit; **shro** shifts zeros in from the most significant bit. These instructions are equivalent to **mulo** and **divo** by the power of 2, respectively.

shli shifts zeros in from the least significant bit. When the shift operation results in an overflow, an integer-overflow fault is generated (when enabled). The destination register is written with the source shifted as much as possible without overflow and an integer-overflow fault is signaled.

shri performs a conventional arithmetic shift right operation by extending the sign bit. However, when this instruction is used to divide a negative integer operand by the power of 2, it may produce an incorrect quotient. (Discarding the bits shifted out has the effect of rounding the result toward negative.)

shrdi is provided for dividing integers by the power of 2. With this instruction, 1 is added to the result when the bits shifted out are non-zero and the operand is negative, which produces the correct result for negative operands. **shli** and **shrdi** are equivalent to **muli** and **divi** by the power of 2, respectively, except in cases where an overflow error occurs.

rotate rotates operand bits to the left (toward higher significance) by a specified number of bits. Bits shifted beyond the register's left boundary (bit 31) appear at the right boundary (bit 0).

The **eshro** instruction performs an ordinal right shift of a source register pair (64 bits) by as much as 32 bits and stores the result in a single (32-bit) register. This instruction is equivalent to an extended divide by a power of 2, which produces no remainder. The instruction is also the equivalent of a 64-bit extract of 32 bits.

5.2.3.4 Extended Arithmetic

These instructions support extended-precision arithmetic; (i.e., arithmetic operations on operands greater than one word in length):

Table 5-11. Extended Arithmetic Instructions

Instruction	Description	Instruction	Description
addc	add ordinal with carry	**emul**	extended multiply
subc	subtract ordinal with carry	**ediv**	extended divide

addc adds two word operands (literals or contained in registers) plus the AC Register condition code bit 1 (used here as a carry bit). When the result has a carry, bit 1 of the condition code is set; otherwise, it is cleared. This instruction's description in Chapter 6, "Instruction Set Reference" gives an example of how this instruction can be used to add two long-word (64-bit) operands together.

subc is similar to **addc**, except it is used to subtract extended-precision values. Although **addc** and **subc** treat their operands as ordinals, the instructions also set bit 0 of the condition codes when the operation would have resulted in an integer overflow condition. This facilitates a software implementation of extended integer arithmetic.

emul multiplies two ordinals (each contained in a register), producing a long ordinal result (stored in two registers). **ediv** divides a long ordinal by an ordinal, producing an ordinal quotient and an ordinal remainder (stored in two adjacent registers).

5.2.4 Logical

These instructions perform bitwise Boolean operations on the specified operands:

Table 5-12. Logical Instructions

Instruction	Description	Instruction	Description
and	*src2* AND *src1*	**xnor**	*src2* XNOR *src1*
notand	(NOT *src2*) AND *src1*	**not**	NOT *src1*
andnot	*src2* AND (NOT *src1*)	**notor**	(NOT *src2*) or *src1*
xor	*src2* XOR *src1*	**ornot**	*src2* **or** (NOT *src1*)
or	*src2* OR *src1*	**nand**	NOT (*src2* AND *src1*)
nor	NOT (*src2* OR *src1*)		

All logical instructions use the REG format and can operate on literals or local or global registers.

5.2.5 Bit, Bit Field and Byte Operations

These perform operations on a specified bit or bit field in an ordinal operand. All Bit, Bit Field and Byte instructions use the REG format and can operate on literals or local or global registers.

5.2.5.1 Bit Operations

These instructions operate on a specified bit:

Table 5-13. Bit Operations Instructions

Instruction	Description	Instruction	Description
setbit	set bit	**alterbit**	alter bit
clrbit	clear bit	**scanbit**	scan for bit
notbit	invert bit	**spanbit**	span over bit

setbit, **clrbit** and **notbit** set, clear or complement (toggle) a specified bit in an ordinal.

alterbit alters the state of a specified bit in an ordinal according to the condition code. When the condition code is 010_2, the bit is set; when the condition code is 000_2, the bit is cleared.

chkbit, described in Section 5.2.6, "Comparison" on page 5-11, can be used to check the value of an individual bit in an ordinal.

scanbit and **spanbit** find the most significant set bit or clear bit, respectively, in an ordinal.

5.2.5.2 Bit Field Operations

The two bit field instructions are **extract** and **modify**.

extract converts a specified bit field, taken from an ordinal value, into an ordinal value. In essence, this instruction shifts right a bit field in a register and fills in the bits to the left of the bit field with zeros. (**eshro** also provides the equivalent of a 64-bit extract of 32 bits).

modify copies bits from one register into another register. Only masked bits in the destination register are modified. **modify** is equivalent to a bit field move.

5.2.5.3 Byte Operations

scanbyte performs a byte-by-byte comparison of two ordinals to determine when any two corresponding bytes are equal. The condition code is set based on the results of the comparison. **scanbyte** uses the REG format and can specify literals or local or global registers as arguments.

bswap alters the order of bytes in a word, reversing its "endianess."

5.2.6 Comparison

The processor provides several types of instructions for comparing two operands, as described in the following subsections.

5.2.6.1 Compare and Conditional Compare

These instructions compare two operands then set the condition code bits in the AC register according to the results of the comparison:

Table 5-14. ompare and Conditional Compare Instructions

Instruction	Description	Instruction	Description
cmpi	Compare Integer	**concmpi**	Conditional Compare Integer
cmpib	Compare Integer Byte	**concmpo**	Conditional Compare Ordinal
cmpis	Compare Integer Short	**chkbit**	Check Bit
cmpo	Compare Ordinal		

These all use the REG format and can specify literals or local or global registers. The condition code bits are set to indicate whether one operand is less than, equal to, or greater than the other operand. See Section 3.6.2, "Arithmetic Controls Register – AC" on page 3-14 for a description of the condition codes for conditional operations.

cmpi and **cmpo** simply compare the two operands and set the condition code bits accordingly. **concmpi** and **concmpo** first check the status of condition code bit 2:

- When not set, the operands are compared as with **cmpi** and **cmpo**.
- When set, no comparison is performed and the condition code flags are not changed.

The conditional-compare instructions are provided specifically to optimize two-sided range comparisons to check for the condition when A is between B and C ($B \leq A \leq C$). Here, a compare instruction (**cmpi** or **cmpo**) checks one side of the range ($A \geq B$) and a conditional compare instruction (**concmpi** or **concmpo**) checks the other side ($A \leq C$) according to the result of the first comparison. The condition codes following the conditional comparison directly reflect the results of both comparison operations. Therefore, only one conditional branch instruction is required to act upon the range check; otherwise, two branches would be needed.

chkbit checks a specified bit in a register and sets the condition code flags according to the bit state. The condition code is set to 010_2 when the bit is set and 000_2 otherwise.

5.2.6.2 Compare and Increment or Decrement

These instructions compare two operands, set the condition code bits according to the compare results, then increment or decrement one of the operands:

Table 5-15. Compare and Increment or Decrement Instructions

Instruction	Description	Instruction	Description
cmpinci	compare and increment integer	**cmpdeci**	compare and decrement integer
cmpinco	compare and increment ordinal	**cmpdeco**	compare and decrement ordinal

These all use the REG format and can specify literals or local or global registers. They are an architectural performance optimization which allows two register operations (e.g., compare and add) to execute in a single cycle. The intended use of these instructions is at the end of iterative loops.

5.2.6.3 Test Condition Codes

These test instructions allow the state of the condition code flags to be tested:

Table 5-16. Test Condition Code Instructions

Instruction	Description	Instruction	Description
teste	test for equal	**testg**	test for greater
testne	test for not equal	**testge**	test for greater or equal
testl	test for less	**testo**	test for ordered
testle	test for less or equal	**testno**	test for unordered

When the condition code matches the instruction-specified condition, a TRUE (0000 0001H) is stored in a destination register; otherwise, a FALSE (0000 0000H) is stored. All use the COBR format and can operate on local and global registers.

5.2.7 Branch

Branch instructions allow program flow direction to be changed by explicitly modifying the IP. The processor provides three branch instruction types:

- unconditional branch
- conditional branch
- compare and branch

Most branch instructions specify the target IP by specifying a signed *displacement* to be added to the current IP. Other branch instructions specify the target IP's memory address, using one of the processor's addressing modes. This latter group of instructions is called extended addressing instructions (e.g., branch extended, branch-and-link extended).

5.2.7.1 Unconditional Branch

These instructions are used for unconditional branching:

Table 5-17. Unconditional Branch Instructions

Instruction	Description	Instruction	Description
b	Branch	**bal**	Branch and Link
bx	Branch Extended	**balx**	Branch and Link Extended

b and **bal** use the CTRL format. **bx** and **balx** use the MEM format and can specify local or global registers as operands. **b** and **bx** cause program execution to jump to the specified target IP. These two instructions perform the same function; however, their determination of the target IP differs. The target IP of a **b** instruction is specified at link time as a relative *displacement* from the current IP. The target IP of the **bx** instruction is the absolute address resulting from the instruction's use of a memory-addressing mode during execution.

bal and **balx** store the next instruction's address in a specified register, then jump to the specified target IP. (For **bal**, the RIP is automatically stored in register g14; for **balx**, the RIP location is specified with an instruction operand.) As described in Section 7.9, "Branch-and-Link" on page 7-20, branch and link instructions provide a method of performing procedure calls that do not use the processor's integrated call/return mechanism. Here, the saved instruction address is used as a return IP. Branch and link is generally used to call leaf procedures (that is, procedures that do not call other procedures).

bx and **balx** can make use of any memory-addressing mode.

5.2.7.2 Conditional Branch

With conditional branch (**BRANCH IF**) instructions, the processor checks the AC register condition code flags. When these flags match the value specified with the instruction, the processor jumps to the target IP. These instructions use the *displacement-plus-ip* method of specifying the target IP:

Table 5-18. Conditional Branch Instructions

Instruction	Description	Instruction	Description
be	branch if equal/true	**bg**	branch if greater
bne	branch if not equal	**bge**	branch if greater or equal
bl	branch if less	**bo**	branch if ordered
ble	branch if less or equal	**bno**	branch if unordered/false

All use the CTRL format. **bo** and **bno** are used with real numbers. **bno** can also be used with the result of a **chkbit** or **scanbit** instruction. Refer to Section 3.6.2.2, "Condition Code (AC.cc)" on page 3-15 for a discussion of the condition code for conditional operations.

5.2.7.3 Compare and Branch

These instructions compare two operands then branch according to the comparison result. Three instruction subtypes are compare integer, compare ordinal and branch on bit:

Table 5-19. Compare and Branch Instructions

Instruction	Description	Instruction	Description
cmpibe	compare integer and branch if equal	**cmpobe**	compare ordinal and branch if equal
cmpibne	compare integer and branch if not equal	**cmpobne**	compare ordinal and branch if not equal
cmpibl	compare integer and branch if less	**cmpobl**	compare ordinal and branch if less
cmpible	compare integer and branch if less or equal	**cmpoble**	compare ordinal and branch if less or equal
cmpibg	compare integer and branch if greater	**cmpobg**	compare ordinal and branch if greater
cmpibge	compare integer and branch if greater or equal	**cmpobge**	compare ordinal and branch if greater or equal
cmpibo	compare integer and branch if ordered	**bbs**	check bit and branch if set
cmpibno	compare integer and branch if unordered	**bbc**	check bit and branch if clear

All use the COBR machine instruction format and can specify literals, local or global registers as operands. With compare ordinal and branch (**compob***) and compare integer and branch (**compib***) instructions, two operands are compared and the condition code bits are set as described in Section 5.2.6, "Comparison" on page 5-11. A conditional branch is then executed as with the conditional branch (**BRANCH IF**) instructions.

With check bit and branch instructions (**bbs**, **bbc**), one operand specifies a bit to be checked in the second operand. The condition code flags are set according to the state of the specified bit: 010_2 (true) when the bit is set and 000_2 (false) when the bit is clear. A conditional branch is then executed according to condition code bit settings.

These instructions can be used to optimize execution performance time. When it is not possible to separate adjacent compare and branch instructions from other unrelated instructions, replacing two instructions with a single compare and branch instruction increases performance.

5.2.8 Call/Return

The 80303 I/O processor offers an on-chip call/return mechanism for making procedure calls. Refer to Section 7.1, "Call and Return Mechanism" on page 7-2. The following instructions support this mechanism:

Table 5-20. Call/Return Instructions

Instruction	Description	Instruction	Description
call	call	**calls**	call system
callx	call extended	**ret**	return

call and **ret** use the CTRL machine-instruction format. **callx** uses the MEM format and can specify local or global registers. **calls** uses the REG format and can specify local or global registers.

call and **callx** make local calls to procedures. A local call is a call that does not require a switch to another stack. **call** and **callx** differ only in the method of specifying the target procedure's address. The target procedure of a call is determined at link time and is encoded in the opword as a signed *displacement* relative to the call IP. **callx** specifies the target procedure as an absolute 32-bit address calculated at run time using any one of the addressing modes. For both instructions, a new set of local registers and a new stack frame are allocated for the called procedure.

calls is used to make calls to system procedures — procedures that provide a kernel or system-executive service. This instruction operates similarly to **call** and **callx**, except that it gets its target-procedure address from the system procedure table. An index number included as an operand in the instruction provides an entry point into the procedure table.

Depending on the type of entry being pointed to in the system procedure table, **calls** can cause either a system-supervisor call or a system-local call to be executed. A system-supervisor call is a call to a system procedure that switches the processor to supervisor mode and switches to the supervisor stack. A system-local call is a call to a system procedure that does not cause an execution mode or stack change. Supervisor mode is described throughout Chapter 7, "Procedure Calls".

ret performs a return from a called procedure to the calling procedure (the procedure that made the call). **ret** obtains its target IP (return IP) from linkage information that was saved for the calling procedure. **ret** is used to return from all calls — including local and supervisor calls — and from implicit calls to interrupt and fault handlers.

5.2.9 Faults

Generally, the processor generates faults automatically as the result of certain operations. Fault handling procedures are then invoked to handle various fault types without explicit intervention by the currently running program. These conditional fault instructions permit a program to explicitly generate a fault according to the state of the condition code flags. All use the CTRL format.

Table 5-21. Faults Instructions

Instruction	Description	Instruction	Description
faulte	fault if equal	**faultg**	fault if greater
faultne	fault if not equal	**faultge**	fault if greater or equal
faultl	fault if less	**faulto**	fault if ordered
faultle	fault if less or equal	**faultno**	fault if unordered

syncf ensures that any faults that occur during the execution of prior instructions occur before the instruction that follows the **syncf**. **syncf** uses the REG format and requires no operands.

5.2.10 Debug

The processor supports debugging and monitoring of program activity through the use of trace events. The following instructions support these debugging and monitoring tools:

Table 5-22. Debug Instructions

Instruction	Description	Instruction	Description
modpc	modify process controls	**mark**	mark
modtc	modify trace controls	**fmark**	force mark

These all use the REG format. Trace functions are controlled with bits in the Trace Control (TC) register which enable or disable various types of tracing. Other TC register flags indicate when an enabled trace event is detected. Refer to Chapter 10, "Tracing and Debugging".

modtc permits trace controls to be modified. **mark** causes a breakpoint trace event to be generated when breakpoint trace mode is enabled. **fmark** generates a breakpoint trace independent of the state of the breakpoint trace mode bits.

Other instructions that are helpful in debugging include **modpc** and **sysctl**. **modpc** can enable/disable trace fault generation. The **sysctl** instruction also provides control over breakpoint trace event generation. This instruction is used, in part, to load and control the 80303 I/O processor's breakpoint registers.

5.2.11 Atomic Instructions

Atomic instructions perform an atomic read-modify-write operation on operands in memory. An atomic operation is one in which other memory operations are forced to occur before or after, but not during, the accesses that comprise the atomic operation. These instructions are required to enable synchronization between interrupt handlers and background tasks in any system. They are also particularly useful in systems where several agents — processors, coprocessors or external logic — have access to the same system memory for communication.

The atomic instructions are atomic add (**atadd**) and atomic modify (**atmod**). **atadd** causes an operand to be added to the value in the specified memory location. **atmod** causes bits in the specified memory location to be modified under control of a mask. Both instructions use the REG format and can specify literals or local or global registers as operands.

5.2.12 Processor Management

These instructions control processor-related functions:

Table 5-23. Processor Management Instructions

Instruction	Description	Instruction	Description
modpc	Modify the Process Controls register	**modac**	Modify the Arithmetic Controls register
flushreg	Flush cached local register sets to memory		

All use the REG format and can specify literals or local or global registers.

modpc provides a method of reading and modifying PC register contents. Only programs operating in supervisor mode may modify the PC register; however, any program may read it.

The processor provides a flush local registers instruction (**flushreg**) to save the contents of the cached local registers to the stack. The flush local registers instruction automatically stores the contents of all the local register sets — except the current set — in the register save area of their associated stack frames.

The modify arithmetic controls instruction (**modac**) allows the AC register contents to be copied to a register and/or modified under the control of a mask. The AC register cannot be explicitly addressed with any other instruction; however, it is implicitly accessed by instructions that use the condition codes or set the integer overflow flag.

sysctl is used to configure the interrupt controller, breakpoint registers and instruction cache. It also permits software to signal an interrupt or cause a processor reset and reinitialization. **sysctl** may be executed only by programs operating in supervisor mode.

intctl, **inten** and **intdis** are used to enable and disable interrupts and to determine current interrupt enable status.

5.3 Performance Optimization

Performance optimization is categorized into two sections: instructions optimizations and miscellaneous optimizations.

5.3.1 Instruction Optimizations

Instruction optimizations are broken down by the instruction classification.

5.3.1.1 Load / Store Execution Model

Because the 80303 I/O processor has a 32-bit external data bus, multiple word accesses require multiple cycles. The processor uses microcode to sequence the multi-word accesses. Because the microcode can ensure that aligned multi-words are bursted together on the external bus, software should not substitute multiple single-word instructions for one multi-word instruction for data that is not likely to be in cache; (i.e., one **ldq** provides better bus performance than four **ld** instructions).

Once a load is issued, the processor attempts to execute other instructions while the load is outstanding. It is important to note that when the load misses the data cache, the processor does not stall the issuing of subsequent instructions (other than stores) that do not depend on the load.

Software should avoid following a load with an instruction that depends on the result of the load. For a load that hits the data cache, a one-cycle stall occurs when the instruction immediately after the load requires the data. When the load fails to hit the data cache, the instruction depending on the load is stalled until the outstanding load request is resolved.

Multiple, back-to-back load instructions do not stall the processor until the bus queue becomes full.

The processor delays issuing a store instruction until all previously-issued load instructions complete. This happens regardless of whether the store is dependent on the load. This ordering between loads and stores ensures that the return data from a previous cache-read miss does not overwrite the cache line updated by a subsequent store.

5.3.1.2 Compare Operations

Byte and short word data is more efficiently compared using the new byte and short compare instructions (**cmpob, cmpib, cmpos, cmpis**), rather than shifting the data and using a word compare instruction.

5.3.1.3 Microcoded Instructions

While the majority of instructions on the 80303 I/O processor are single cycle and are executed directly by processor hardware, some require microcode emulation. Entry into a microcode routine requires two cycles. Exit from microcode typically requires two cycles. For some routines, one cycle of the exit process can execute in parallel with another instruction, thus saving one cycle of execution time.

5.3.1.4 Multiply-Divide Unit Instructions

The Multiply-Divide Unit (MDU) performs a number of multi-cycle arithmetic operations. These can range from 2 cycles for a 16-bitx32-bit **mulo**, 4 cycles for a 32-bitx32-bit **mulo**, to 30+ cycles for an **ediv**.

Once issued, these MDU instructions are executed in parallel with other non-MDU instructions that do not depend on the result of the MDU operation. Attempting to issue another MDU instruction while a current MDU instruction is executing, stalls the processor until the first one completes.

5.3.1.5 Multi-Cycle Register Operations

A few register operations can also take multiple cycles. The following instructions are performed in microcode:

Table 5-24. Multi-Cycle Register Operations Microcode Instructions

• **bswap**	• **extract**	• **eshro**	• **modify**	• **movl**	• **movt**
• **movq**	• **shrdi**	• **scanbit**	• **spanbit**	• **testno**	• **testo**
• **testl**	• **testle**	• **teste**	• **testne**	• **testg**	• **testge**

On the 80303 I/O processor, **test<cc>** *dst* is microcoded and takes many more cycles than **SEL<cc>** 0,1,*dst*, which is executed in one cycle directly by processor hardware.

Multi-register move operation execution time can be decreased at the expense of cache utilization and code density by using **mov** the appropriate number of times instead of **movl**, **movt** and **movq**.

5.3.1.6 Simple Control Transfer

There is no branch look-ahead or branch prediction mechanism on the 80303 I/O processor. Simple branch instructions take one cycle to execute, and one more cycle is needed to fetch the target instruction if the branch is actually taken.

Table 5-25. Simple Control Transfer Instructions

• **b**	• **bno**	• **bl**	• **be**	• **bg**
• **bal**	• **bo**	• **ble**	• **bne**	• **bge**

b, bal, bno, bo, bl, ble, be, bne, bg, bge

One mode of the **bx** (branch-extended) instruction, **bx** (base), is also a simple branch and takes one cycle to execute and one cycle to fetch the target.

As a result, a **bal(g14)** or **bx (g14)** sequence provides a two-cycle call and return mechanism for efficient leaf procedure implementation.

Compare-and-branch instructions have been optimized on the 80303 I/O processor. They require two cycles to execute, and one more cycle to fetch the target instruction if the branch is actually taken (see Table 5-19 on page 5-14).

5.3.1.7 Memory Instructions

The 80303 I/O processor provides efficient support for naturally aligned byte, short, and word accesses that use one of six optimized addressing modes. These accesses require only one to two cycles to execute; additional cycles are needed for a load to return its data.

The byte, short and word memory instructions are:

Table 5-26. Memory Instructions

• ldob	• ldis	• stob	• stis
• ldib	• ld	• stib	• st
• ldos	• lda	• stos	•

The remainder of accesses require multiple cycles to execute. These include:

- Unaligned short, and word accesses
- Byte, short, and word accesses that do not use one of the 6 optimized addressing modes
- Multi-word accesses

The multi-word accesses are:

Table 5-27. Memory Instructions

• ldl	• ldt	• ldq	• stl	• stt	• stq

5.3.1.8 Unaligned Memory Accesses

Unaligned memory accesses are performed by microcode. Microcode sequences the access into smaller aligned pieces and does any merging of data that is needed. As a result, these accesses are not as efficient as aligned accesses. In addition, no bursting on the external bus is performed for these accesses. Whenever possible, unaligned accesses should be avoided.

5.3.2 Miscellaneous Optimizations

5.3.2.1 Masking of Integer Overflow

The i960 core architecture inserts an implicit **syncf** before performing a call operation or delivering an interrupt so that a fault handler can be dispatched first, when necessary. **syncf** can require a number of cycles to complete when a multi-cycle integer-multiply (**muli**) or integer-divide (**divi**) instruction is issued previously and integer-overflow faults are unmasked (allowed to occur). Call performance and interrupt latency can be improved by masking integer-overflow faults (AC.om = 1), which allows the implicit **syncf** to complete more quickly.

5.3.2.2 Avoid Using PFP, SP, R3 As Destinations for MDU Instructions

When performing a call operation or delivering an interrupt, the processor typically attempts to push the first four local registers (pfp, sp, rip, and r3) onto the local register cache as early as possible. Because of register-interlock, this operation is stalled until previous instructions return their results to these registers. In most cases, this is not a problem; however, in the case of multi-cycle instructions (**divo, divi, ediv, modi, remo, and remi**), the processor could be stalled for many cycles waiting for the result and unable to proceed to the next step of call processing or interrupt delivery.

Call performance and interrupt latency can be improved by avoiding the first four registers as the destination for a MDU instruction. Generally, registers pfp, sp, and rip should be avoided; they are used for procedure linking.

5.3.2.3 Use Global Registers (g0 - g14) As Destinations for MDU Instructions

Using the same rationale as in the previous item, call processing and interrupt performance are improved even further by using global registers (g0-g14) as the destination for multi-cycle MDU instructions. This is because there is no dependency between g0-g14 and implicit or explicit call operations (i.e., global registers are not pushed onto the local register cache).

5.3.2.4 Execute in Imprecise Fault Mode

Significant performance improvement is possible by allowing imprecise faults (AC.nif = 0). In precise fault mode (AC.nif = 1), the processor does not issue a new instruction until the previous one completes. This ensures that a fault from the previous instruction is delivered before the next instruction can begin execution. Imprecise fault mode allows new instructions to be issued before previous ones complete, thus increasing the instruction issue rate. Many applications can tolerate the imprecise fault reporting for the performance gain. A **syncf** can be used in imprecise fault mode to isolate faults at desired points of execution when necessary.

5.3.3 Cache Control

The following instructions provide instruction and data cache control functions.

Table 5-28. Cache Control Instructions

Instruction	Description	Instruction	Description
icctl	Instruction cache control	**dcctl**	Data cache control

icctl and **dcctl** provide cache control functions including: enabling, disabling, loading and locking (instruction cache only), invalidating, getting status and storing cache information out to memory.

Instruction Set Reference 6

This chapter provides detailed information about each instruction available to the Intel® 80303 I/O processor. Instructions are listed alphabetically by assembly language mnemonic. Format and notation used in this chapter are defined in Section 6.1, "Notation" on page 6-2.

Information in this chapter is oriented toward programmers who write assembly language code for the 80303 I/O processor. Information provided for each instruction includes:

- Alphabetic listing of all instructions
- Faults that can occur during execution
- Assembly language mnemonic, name and format
- Action (or algorithm) and other side effects of executing an instruction
- Description of the instruction's operation
- Assembly language example
- Related instructions
- Opcode and instruction encoding format

Additional information about the instruction set can be found in the following chapters and appendices in this manual:

- Chapter 5, "Instruction Set Overview" - Summarizes the instruction set by group and describes the assembly language instruction format.
- Appendix A, "Machine-Level Instruction Formats" - Describes instruction set opword encodings.
- Appendix B, "Opcodes and Execution Times" - A quick-reference listing of instruction encodings assists debugging with a logic analyzer.

6.1 Notation

In general, notation in this chapter is consistent with usage throughout the manual; however, there are a few exceptions. Read the following subsections to understand notations that are specific to this chapter.

6.1.1 Alphabetic Reference

Instructions are listed alphabetically by assembly language mnemonic. When several instructions are related and fall together alphabetically, they are described as a group on a single page.

The instruction's assembly language mnemonic is shown in bold at the top of the page (e.g., **subc**). Occasionally, it is not practical to list all mnemonics at the page top. In these cases, the name of the instruction group is shown in capital letters (e.g., **BRANCH<cc>** or **FAULT<cc>**).

The 80303 I/O processor-specific extensions to the Intel® i960® microprocessor instruction set are indicated in the header text for each such instruction. This type of notation is also used to indicate new core architecture instructions. Sections describing new core instructions provide notes as to which i960-series processors do not implement these instructions.

Generally, instruction set extensions are not portable to other i960 processor implementations. Further, new core instructions are not typically portable to earlier i960 processor family implementations such as the Intel® i960® Kx microprocessors.

6.1.2 Mnemonic

The *Mnemonic* section gives the mnemonic (in boldface type) and instruction name for each instruction covered on the page, for example:

Example 6-1. Mnemonic Instruction

Instruction	Description
subi	Subtract Integer

This name is the actual assembly language instruction name recognized by assemblers.

6.1.3 Format

The *Format* section gives the instruction's assembly language format and allowable operand types. Format is given in two or three lines. The following is a two-line format example:

Example 6-2. Two-line Format

sub*	*src1*	*src2*	*dst*
	reg/lit	reg/lit	reg

The first line gives the assembly language mnemonic (boldface type) and operands (italics). When the format is used for two or more instructions, an abbreviated form of the mnemonic is used. An * (asterisk) at the end of the mnemonic indicates a variable: in the above example, **sub*** is either **subi** or **subo**. Capital letters indicate an instruction class. For example, **ADD<cc>** refers to the class of conditional add instructions (e.g., **addio**, **addig**, **addoo**, **addog**).

Operand names are designed to describe operand function (e.g., *src*, *len*, *mask*).

The second line shows allowable entries for each operand. Notation is as follows:

Example 6-3. Operand Allowable Entries (second line)

Entry	Description
reg	Global (g0 ... g15) or local (r0 ... r15) register
lit	Literal of the range 0 ... 31
disp	Signed displacement of range (-2^{22} ... $2^{22} - 1$)
mem	Address defined with the full range of addressing modes

In some cases, a third line is added to show register or memory location contents. For example, it may be useful to know that a register is to contain an address. The notation used in this line is as follows:

Example 6-4. Register or Memory Location Contents (third line)

Contents	Description
addr	Address
efa	Effective Address

6.1.4 Description

The *Description* section is a narrative description of the instruction's function and operands. It also gives programming hints when appropriate.

6.1.5 Action

The *Action* section gives an algorithm written in a "C-like" pseudo-code that describes direct effects and possible side effects of executing an instruction. Algorithms document the instruction's net effect on the programming environment; they do not necessarily describe how the processor actually implements the instruction. The following is an example of the action algorithm for the **alterbit** instruction:

Example 6-5. Action Algorithm for the alterbit Instruction

```
if ((AC.cc & 010₂)==0)
        dst = src2 & ~(2**(src1%32));
else
        dst = src2 | 2**(src1%32);
```

Table 6-1 defines each abbreviation used in the instruction reference pseudo-code. The pseudo-code has been written to comply as closely as possible with standard C programming language notation. Table 6-1 lists the pseudocode symbol definitions.

Table 6-1. Pseudo-Code Symbol Definitions

Symbol	Description
==, !=	Comparison: equal, not equal
<, >	less than, greater than
<=, >=	less than or equal to, greater than or equal to
<<, >>	Logical Shift
**	Exponentiation
&, &&	Bitwise AND, logical AND
|, ||	Bitwise OR, logical OR
^	Bitwise XOR
~	One's Complement
%	Modulo
+, -	Addition, Subtraction
*	Multiplication (Integer or Ordinal)
/	Division (Integer or Ordinal)
#	Comment delimiter

Table 6-2. Faults Applicable to All Instructions

Fault Type	Subtype	Description
OPERATION	UNIMPLEMENTED	An attempt to execute any instruction fetched from internal data RAM or a memory-mapped region causes an operation unimplemented fault.
TRACE	MARK	A Mark Trace Event is signaled after completion of an instruction for which there is a hardware breakpoint condition match. A Trace fault is generated when PC.mk is set.
	INSTRUCTION	An Instruction Trace Event is signaled after instruction completion. A Trace fault is generated when both PC.te and TC.i=1.

Table 6-3. Common Faulting Conditions

Fault Type	Subtype	Description
OPERATION	UNALIGNED	Any instruction that causes an unaligned memory access causes an operation aligned fault when unaligned faults are not masked in the fault configuration word in the Processor Control Block (PRCB).
	INVALID_OPCODE	This fault is generated when the processor attempts to execute an instruction containing an undefined opcode or addressing mode.
	INVALID_OPERAND	This fault is caused by a non-defined operand in a supervisor mode only instruction or by an operand reference to an unaligned long-, triple- or quad-register group.
	UNIMPLEMENTED	This fault can occur due to an attempt to perform a non-word or unaligned access to a memory-mapped region or when attempting to fetch instructions from MMR space or internal data RAM.
TYPE	MISMATCH	Any instruction that attempts to write to supervisor protected internal data RAM or a memory-mapped register in supervisor space while not in supervisor mode causes a TYPE.MISMATCH fault. This fault is also generated for any non-supervisor mode reference to an SFR.

6.1.6 Faults

The *Faults* section lists faults that can be signaled as a direct result of instruction execution. Table 6-2 shows the possible faulting conditions that are common to the entire instruction set and could directly result from any instruction. These fault types are not included in the instruction reference. Table 6-3 shows the possible faulting conditions that are common to large subsets of the instruction set. When an instruction can generate a fault, it is noted in that instruction's *Faults* section. In these sections, "Standard" refers to the faults shown in Table 6-2 and Table 6-3.

6.1.7 Example

The *Example* section gives an assembly language example of an application of the instruction.

6.1.8 Opcode and Instruction Format

The *Opcode and Instruction Format* section gives the opcode and instruction format for each instruction, for example:

Example 6-6. Opcode and Instruction Format

Instruction	Opcode	Format
subi	593H	REG

The opcode is given in hexadecimal format. The format is one of four possible formats: REG, COBR, CTRL and MEM. Refer to Appendix A, "Machine-Level Instruction Formats" for more information on the formats.

6.1.9 See Also

The *See Also* section gives the mnemonics of related instructions which are also alphabetically listed in this chapter.

6.1.10 Side Effects

This section indicates whether the instruction causes changes to the condition code bits in the Arithmetic Controls.

6.1.11 Notes

This section provides additional information about an instruction such as whether it is implemented in other i960 processor families.

6.2 Instructions

The processor's instructions are arranged alphabetically by instruction or instruction group.

6.2.1 ADD<cc>

Table 6-4. ADD<cc>

Mnemonic:	**addono**	Add Ordinal if Unordered		
	addog	Add Ordinal if Greater		
	addoe	Add Ordinal if Equal		
	addoge	Add Ordinal if Greater or Equal		
	addol	Add Ordinal if Less		
	addone	Add Ordinal if Not Equal		
	addole	Add Ordinal if Less or Equal		
	addoo	Add Ordinal if Ordered		
	addino	Add Integer if Unordered		
	addig	Add Integer if Greater		
	addie	Add Integer if Equal		
	addige	Add Integer if Greater or Equal		
	addil	Add Integer if Less		
	addine	Add Integer if Not Equal		
	addile	Add Integer if Less or Equal		
	addio	Add Integer if Ordered		
Format:	**add***	*src1*,	*src2*,	dst
		reg/lit	reg/lit	reg

Description: Conditionally adds *src2* and *src1*, values and stores the result in *dst* based on the AC register condition code. If for Unordered the condition code is 0, or if for all other cases the logical AND of the condition code and mask part of the opcode is not 0, then the values are added and placed in the destination. Otherwise the destination is left unchanged. The table below shows the condition code mask for each instruction. The mask is in opcode bits 4-6.

Instruction	Mask	Condition
addono	000_2	Unordered
addino		
addog	001_2	Greater
addig		
addoe	010_2	Equal
addie		
addoge	011_2	Greater or equal
addige		
addol	100_2	Less
addil		
addone	101_2	Not equal
addine		
addole	110_2	Less or equal
addile		
addoo	111_2	Ordered
addio		

Action:

addo<cc>:

```
if((mask & AC.cc) || (mask == AC.cc))
    dst = (src1 + src2)[31:0];
```

addi<cc>:

```
if((mask & AC.cc) || (mask == AC.cc))
{
    {       true_result = (src1 + src2);
            dst = true_result[31:0];
    }
    if((true_result > (2**31) - 1) || (true_result < -2**31))
                                    #Check for overflow
    {       if(AC.om == 1)
              AC.of = 1;
            else
              generate_fault(ARITHMETIC.OVERFLOW);
    }
}
```

Faults:

STANDARD	Refer to Section 6.1.6, "Faults" on page 6-5.
ARITHMETIC.OVERFLOW	Occurs only with **addi<cc>**.

Example:

```
# Assume (AC.cc AND 001₂) ≠ 0.
addig r4, r8, r10            # r10 = r8 + r4

# Assume (AC.cc AND 101₂) = 0.
addone r4, r8, r10           # r10 is not changed.
```

Opcode:	**addono**	780H	REG
	addog	790H	REG
	addoge	7A0H	REG
	addoe	7B0H	REG
	addol	7C0H	REG
	addone	7D0H	REG
	addole	7E0H	REG
	addoo	7F0H	REG
	addino	781H	REG
	addig	791H	REG
	addie	7A1H	REG
	addige	7B1H	REG
	addil	7C1H	REG
	addine	7D1H	REG
	addile	7E1H	REG
	addio	7F1H	REG

See Also: **addc, SUB<cc>, addi, addo**

Notes: This class of core instructions is not implemented on 80960Cx, Kx and Sx processors.

6.2.2 addc

Table 6-5. addc

Mnemonic: **addc** Add Ordinal With Carry

Format:

addc	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg

Description:

Adds *src2* and *src1* values and condition code bit 1 (used here as a carry-in) and stores the result in *dst*. If ordinal addition results in a carry out, condition code bit 1 is set; otherwise, bit 1 is cleared. If integer addition results in an overflow, condition code bit 0 is set; otherwise, bit 0 is cleared. Regardless of addition results, condition code bit 2 is always set to 0.

addc can be used for ordinal or integer arithmetic. **addc** does not distinguish between ordinal and integer source operands. Instead, the processor evaluates the result for both data types and sets condition code bits 0 and 1 accordingly.

An integer overflow fault is never signaled with this instruction.

Action:

```
dst = (src1 + src2 + AC.cc[1])[31:0];
AC.cc[2:0] = 000₂;
if((src2[31] == src1[31]) && (src2[31] != dst[31]))
        AC.cc[0] = 1;                    # Set overflow bit.
AC.cc[1] = (src2 + src1 + AC.cc[1])[32]; # Carry out.
```

Faults: STANDARD Refer to Section 6.1.6, "Faults" on page 6-5.

Example:

```
# Example of double-precision arithmetic.
# Assume 64-bit source operands
# in g0,g1 and g2,g3
cmpo 1, 0           # Clears Bit 1 (carry bit) of
                    # the AC.cc.
addc g0, g2, g0     # Add low-order 32 bits:
                    # g0 = g2 + g0 + carry bit
addc g1, g3, g1     # Add high-order 32 bits:
                    # g1 = g3 + g1 + carry bit
                    # 64-bit result is in g0, g1.
```

Opcode: **addc** 5B0H REG

See Also: **ADD<cc>, SUB<cc>**

Side Effects: Sets the condition code in the arithmetic controls.

6.2.3 addi, addo

Table 6-6. addi, addo

Mnemonic:	**addo**	Add Ordinal		
	addi	Add Integer		
Format:	**add***	*src1,*	*src2,*	*dst*
		reg/lit	reg/lit	reg

Description: Adds *src2* and *src1* values and stores the result in *dst*. The binary results from these two instructions are identical. The only difference is that **addi** can signal an integer overflow.

Action:

addo:

```
dst = (src2 +src1)[31:0];
```

addi:

```
true_result = (src1 + src2);
dst = true_result[31:0];
if((true_result > (2**31) - 1) || (true_result < -2**31)) # Check for overflow
{       if(AC.om == 1)
                AC.of = 1;
        else
                generate_fault(ARITHMETIC.OVERFLOW);
}
```

Faults:	STANDARD	Refer to Section 6.1.6, "Faults" on page 6-5.
	ARITHMETIC.OVERFLOW	Occurs only with addi.
Example:	`addi r4, g5, r9`	`# r9 = g5 + r4`
Opcode:	**addo** 590H	REG
	addi 591H	REG

See Also: **addc, subi, subo, subc, ADD<cc>**

6.2.4 alterbit

Table 6-7. alterbit

Mnemonic:	**alterbit**	Alter Bit		
Format:	**alterbit**	*bitpos*,	*src*,	*dst*
		reg/lit	reg/lit	reg

Description: Copies *src* value to *dst* with one bit altered. *bitpos* operand specifies bit to be changed; condition code determines the value to which the bit is set. If condition code is $X1X_2$, bit 1 = 1, the selected bit is set; otherwise, it is cleared. Typically this instruction is used to set the *bitpos* bit in the *targ* register if the result of a compare instruction is the equal condition code (010_2).

Action:

```
if((AC.cc & 010₂)==0)
        dst = src & ~(2**(bitpos%32));
else
        dst = src | 2**(bitpos%32);
```

Faults: STANDARD — Refer to Section 6.1.6, "Faults" on page 6-5.

Example:

```
# Assume AC.cc = 010₂.
alterbit 24, g4,g9    # g9 = g4, with bit 24 set.
```

Opcode:	**alterbit**	58FH	REG

See Also: **chkbit, clrbit, notbit, setbit**

6.2.5 and, andnot

Table 6-8. and, andnot

Mnemonic:	**and**	And		
	andnot	And Not		
Format:	**and**	*src1,*	*src2,*	*dst*
		reg/lit	reg/lit	reg
	andnot	*src1,*	*src2,*	*dst*
		reg/lit	reg/lit	reg

Description: Performs a bitwise AND (**and**) or AND NOT (**andnot**) operation on *src2* and *src1* values and stores result in *dst*. Note in the action expressions below, *src2* operand comes first, so that with **andnot** the expression is evaluated as:

{*src2* and not (*src1*)}

rather than

{*src1* and not (*src2*)}

Action:

and:

dst = src2 & src1;

andnot:

dst = src2 & ~src1;

Faults: STANDARD — Refer to Section 6.1.6, "Faults" on page 6-5.

Example:

```
and 0x7, g8, g2          # Put lower 3 bits of g8 in g2.
andnot 0x7, r12, r9      # Copy r12 to r9 with lower
                         # three bits cleared.
```

Opcode:	**and**	581H	REG
	andnot	582H	REG

See Also: **nand, nor, not, notand, notor, or, ornot, xnor, xor**

6.2.6 atadd

Table 6-9. atadd

Mnemonic:	**atadd**	Atomic Add		
Format:	**atadd**	*addr,*	*src,*	*dst*
		reg	reg/lit	reg

Description: Adds *src* value (full word) to value in the memory location specified with *addr* operand. This read-modify-write operation is performed on the actual data in memory and never on a cached value on chip. Initial value from memory is stored in *dst*.

Memory read and write are done atomically (i.e., other bus masters must be prevented from accessing the word of memory containing the word specified by *src/dst* operand until operation completes). See Section 3.5.1, "Memory Requirements" on page 3-11 or more information on atomic accesses.

Memory location in *addr* is the word's first byte (LSB) address. Address is automatically aligned to a word boundary. (Note that *addr* operand maps to *src1* operand of the REG format.)

Action:

```
implicit_syncf();
tempa = addr & 0xFFFFFFFC;
temp = atomic_read(tempa);
atomic_write(tempa, temp+src);
dst = temp;
```

Faults: STANDARD — Refer to Section 6.1.6, "Faults" on page 6-5.

Example:

```
atadd r8, r3, r11          # r8 contains the address of
                           # memory location.
                           # r11 = (r8)
                           # (r8) = r11 + r3.
```

Opcode:	**atadd**	612H	REG

See Also: **atmod**

6.2.7 atmod

Table 6-10. atmod

Mnemonic:	**atmod**	Atomic Modify		
Format:	**atmod**	*addr,*	*mask,*	*src/dst*
		reg	reg/lit	reg

Description: Copies the selected bits of *src/dst* value into memory location specified in *addr*. The read-modify-write operation is performed on the actual data in memory and never on a cached value on chip. Bits set in *mask* operand select bits to be modified in memory. Initial value from memory is stored in *src/dst*. See Section 3.5.1, "Memory Requirements" on page 3-11 for information on atomic accesses.

Memory read and write are done atomically (i.e., other bus masters must be prevented from accessing the word of memory containing the word specified with the *src/dst* operand until operation completes).

Memory location in *addr* is the modified word's first byte (LSB) address. Address is automatically aligned to a word boundary.

Action:

```
implicit_syncf();
tempa = addr & 0xFFFFFFFC;
tempb = atomic_read(tempa);
temp = (tempb &~ mask) | (src_dst & mask);
atomic_write(tempa, temp);
src_dst = tempb;
```

Faults: STANDARD — Refer to Section 6.1.6, "Faults" on page 6-5.

Example:

```
atmod g5, g7, g10          # tempa = (g5)
                           # temp = (tempa andnot g7) or
                           # (g10 and g7)
                           # (g5) = temp
                           # g10 = tempa
```

Opcode:	**atmod**	610H	REG

See Also: **atadd**

6.2.8 b, bx

Table 6-11. b, bx

Mnemonic:	**b**	Branch	
	bx	Branch Extended	
Format:	**b**	*targ*	
		disp	
	bx	*targ*	
		mem	
Description:	Branches to the specified target.		
	With the **b** instruction, IP specified with *targ* operand can be no farther than -2^{23} to (2^{23}- 4) bytes from current IP. When using the Intel i960 processor assembler, *targ* operand must be a label which specifies target instruction's IP.		
	bx performs the same operation as **b** except the target instruction can be farther than -2^{23} to (2^{23}- 4) bytes from current IP. Here, the target operand is an effective address, which allows the full range of addressing modes to be used to specify target instruction's IP. The "IP + displacement" addressing mode allows the instruction to be IP-relative. Indirect branching can be performed by placing target address in a register then using a register-indirect addressing mode.		
	Refer to Section 2.3, "Memory Addressing Modes" on page 2-4 for information on this subject.		
Action:	**b, bx**:		
	IP[31:2] = effective_address(targ[31:2]);		
	IP[1:0] = 0;		
Faults:	STANDARD	Refer to Section 6.1.6, "Faults" on page 6-5.	
Example:	`b xyz`	`# IP = xyz;`	
	`bx 1332 (ip)`	`# IP = IP + 8 + 1332;`	
	`# this example uses IP-relative addressing`		
Opcode:	**b**	08H	CTRL
	bx	84H	MEM
See Also:	**bal, balx, BRANCH<cc>, COMPARE AND BRANCH<cc>, bbc, bbs**		

6.2.9 bal, balx

Table 6-12. bal, balx

Mnemonic:

bal	Branch and Link	
balx	Branch and Link Extended	

Format:

bal	*targ*	
	disp	
balx	*targ*,	*dst*
	mem	reg

Description:

Stores address of instruction following **bal** or **balx** in a register then branches to the instruction specified with the *targ* operand.

The **bal** and **balx** instructions are used to call leaf procedures (procedures that do not call other procedures). The IP saved in the register provides a return IP that the leaf procedure can branch to (using a **b** or **bx** instruction) to perform a return from the procedure. Note that these instructions do not use the processor's call-and-return mechanism, so the calling procedure shares its local-register set with the called (leaf) procedure.

With **bal**, address of next instruction is stored in register g14. *targ* operand value can be no farther than -2^{23} to $(2^{23}- 4)$ bytes from current IP. When using the Intel i960 processor assembler, *targ* must be a label which specifies the target instruction's IP.

balx performs same operation as **bal** except next instruction address is stored in *dst* (allowing the return IP to be stored in any available register). With **balx**, the full address space can be accessed. Here, the target operand is an effective address, which allows full range of addressing modes to be used to specify target IP. "IP + displacement" addressing mode allows instruction to be IP-relative. Indirect branching can be performed by placing target address in a register and then using a register-indirect addressing mode.

See Section 2.3, "Memory Addressing Modes" on page 2-4 for a complete discussion of addressing modes available with memory-type operands.

Action:

bal:

```
g14 = IP + 4;
IP[31:2] = effective_address(targ[31:2]);
IP[1:0] = 0;
```

balx:

```
dst = IP + instruction_length;
# Instruction_length = 4 or 8 depending on the addressing mode used.
IP[31:2] = effective_address(targ[31:2]);      # Resume execution at new IP.
IP[1:0] = 0;
```

Faults: STANDARD — Refer to Section 6.1.6, "Faults" on page 6-5.

Example:

```
bal xyz                                 # g14 = IP + 4
                                        # IP = xyz

balx (g2), g4                           # g4 = IP + 4
                                        # IP = (g2)
```

Opcode:

bal	0BH	CTRL
balx	85H	MEM

See Also: **b, bx, BRANCH<cc>, COMPARE AND BRANCH<cc>, bbc, bbs**

6.2.10 bbc, bbs

Table 6-13. bbc, bbs

Mnemonic: **bbc** Check Bit and Branch If Clear
bbs Check Bit and Branch If Set

Format:

bb*	*bitpos*,	*src*,	*targ*
	reg/lit	reg	disp

Description: Checks bit (designated by *bitpos*) in *src* and sets AC register condition code according to *src* value. The processor then performs conditional branch to instruction specified with *targ*, based on condition code state.

For **bbc**, if selected bit in *src* is clear, the processor sets condition code to 000_2 and branches to instruction specified by *targ*; otherwise, it sets condition code to 010_2 and goes to next instruction.

For **bbs**, if selected bit is set, the processor sets condition code to 010_2 and branches to *targ*; otherwise, it sets condition code to 000_2 and goes to next instruction.

targ can be no farther than -2^{12} to $(2^{12} - 4)$ bytes from current IP. When using the Intel i960 processor assembler, *targ* must be a label which specifies target instruction's IP.

Action:

bbs:

```
if((src & 2**(bitpos%32)) == 1)
{       AC.cc = 010₂;
        temp[31:2] = sign_extension(targ[12:2]);
        IP[31:2] = IP[31:2] + temp[31:2];
        IP[1:0] = 0;
}
else
        AC.cc = 000₂;
```

bbc:

```
if((src & 2**(bitpos%32)) == 0)
{       AC.cc = 000₂;
        temp[31:2] = sign_extension(targ[12:2]);
        IP[31:2] = IP[31:2] + temp[31:2];
        IP[1:0] = 0;
}
else
        AC.cc = 010₂;
```

Faults: STANDARD Refer to Section 6.1.6, "Faults" on page 6-5.

Example:

```
# Assume bit 10 of r6 is clear.
bbc 10, r6, xyz                 # Bit 10 of r6 is checked
                                # and found clear:
                                # AC.cc = 000
                                # IP = xyz;
```

Opcode:

bbc	30H	COBR
bbs	37H	COBR

See Also: **chkbit, COMPARE AND BRANCH<cc>, BRANCH<cc>**

Side Effects: Sets the condition code in the arithmetic controls.

6.2.11 BRANCH<cc>

Table 6-14. BRANCH<cc>

Mnemonic:	**be**	Branch If Equal
	bne	Branch If Not Equal
	bl	Branch If Less
	ble	Branch If Less Or Equal
	bg	Branch If Greater
	bge	Branch If Greater Or Equal
	bo	Branch If Ordered
	bno	Branch If Unordered
Format:	**b***	*targ*
		disp

Description:

Branches to instruction specified with *targ* operand according to AC register condition code state.

For all branch<cc> instructions except **bno**, the processor branches to instruction specified with *targ*, if the logical AND of condition code and mask part of opcode is not zero. Otherwise, it goes to next instruction.

For **bno**, the processor branches to instruction specified with *targ* if the condition code is zero. Otherwise, it goes to next instruction.

For instance, **bno** (unordered) can be used as a branch if false instruction when coupled with **chkbit**. For **bno**, branch is taken if condition code equals 000_2. **be** can be used as branch-if true instruction.

The *targ* operand value can be no farther than -2^{23} to $(2^{23}- 4)$ bytes from current IP.

The following table shows condition code mask for each instruction. The mask is in opcode bits 0-2.

Instruction	Mask	Condition
bno	000_2	Unordered
bg	001_2	Greater
be	010_2	Equal
bge	011_2	Greater or equal
bl	100_2	Less
bne	101_2	Not equal
ble	110_2	Less or equal
bo	111_2	Ordered

Action:

```
if((mask & AC.cc) || (mask == AC.cc))
{       temp[31:2] = sign_extension(targ[23:2]);
        IP[31:2] = IP[31:2] + temp[31:2];
        IP[1:0] = 0;
}
```

Faults: STANDARD — Refer to Section 6.1.6, "Faults" on page 6-5.

Example:

```
# Assume (AC.cc AND 100(2)) ≠ 0
bl xyz          # IP = xyz;
```

Opcode:

be	12H	CTRL
bne	15H	CTRL
bl	14H	CTRL
ble	16H	CTRL
bg	11H	CTRL
bge	13H	CTRL
bo	17H	CTRL
bno	10H	CTRL

See Also: **b, bx, bbc, bbs, COMPARE AND BRANCH<cc>, bal, balx, BRANCH<cc>**

6.2.12 bswap

Table 6-15. bswap

Mnemonic:	**bswap**	Byte Swap	
Format:	**bswap**	*src1:src,*	*src2:dst*
		reg/lit	reg
Description:	Alters the order of bytes in a word, reversing its "endianess." Copies bytes 3:0 of *src1* to *src2* reversing order of the bytes. Byte 0 of *src1* becomes byte 3 of *src2*, byte 1 of *src1* becomes byte 2 of *src2*, etc.		
Action:	dst = (rotate_left(src 8) & 0x00FF00FF) +(rotate_left(src 24) & 0xFF00FF00);		
Faults:	STANDARD	Refer to Section 6.1.6, "Faults" on page 6-5.	
Example:		`# g8 = 0x89ABCDEF`	
	`bswap g8, g10`	`# Reverse byte order`	
		`# g10 now 0xEFCDAB89`	
Opcode:	**bswap**	5ADH	REG
See Also:	**scanbyte, rotate**		
Notes:	This core instruction is not implemented on 80960Cx, Kx and Sx processors.		

6.2.13 call

Table 6-16. call

Mnemonic: **call** Call

Format: **call** *targ*

disp

Description: Calls a new procedure. *targ* operand specifies the IP of called procedure's first instruction. When using the Intel i960 processor assembler, *targ* must be a label.

In executing this instruction, the processor performs a local call operation as described in Section 7.1.3.1, "Call Operation" on page 7-6. As part of this operation, the processor saves the set of local registers associated with the calling procedure and allocates a new set of local registers and a new stack frame for the called procedure. Processor then goes to the instruction specified with *targ* and begins execution.

targ can be no farther than -2^{23} to $(2^{23} - 4)$ bytes from current IP.

Action:

```
# Wait for any uncompleted instructions to finish.
implicit_syncf();
temp = (SP + (SALIGN*16 - 1)) & ~(SALIGN*16 - 1)
        # Round stack pointer to next boundary.
        # SALIGN=1 on 80303.
RIP = IP;
if   (register_set_available)
        allocate_new_frame( );
else
        {       save_register_set( );     # Save register set in memory at its FP.
                allocate_new_frame( );
        }
        #       Local register references now refer to new frame.
IP[31:2] = effective_address(targ[31:2]);
IP[1:0] = 0;
PFP = FP;
FP = temp;
SP = temp + 64;
```

Faults: STANDARD Refer to Section 6.1.6, "Faults" on page 6-5.

Example:

```
call xyz                # IP = xyz
```

Opcode: **call** 09H CTRL

See Also: **bal, calls, callx**

6.2.14 calls

Table 6-17. calls (Sheet 1 of 2)

Mnemonic: **calls** Call System

Format: **calls** *targ*
reg/lit

Description: Calls a system procedure. The *targ* operand gives the number of the procedure being called. For **calls**, the processor performs system call operation described in Section 7.5, "System Calls" on page 7-15. *targ* provides an index to a system procedure table entry from which the processor gets the called procedure's IP.

The called procedure can be a local or supervisor procedure, depending on system procedure table entry type. If it is a supervisor procedure, the processor switches to supervisor mode (if not already in this mode).

As part of this operation, processor also allocates a new set of local registers and a new stack frame for called procedure. If the processor switches to supervisor mode, the new stack frame is created on the supervisor stack.

Action:

```
# Wait for any uncompleted instructions to finish.
implicit_syncf();
If (targ > 259)
        generate_fault(PROTECTION.LENGTH);
temp = get_sys_proc_entry(sptbase + 48 + 4*targ);
        # sptbase is address of supervisor procedure table.


 if (register_set_available)
        allocate_new_frame( );
         else
        {       save_register_set( );       Save a frame in memory at its FP
                allocate_new_frame( );
                # Local register references now refer to new frame.
        }
RIP = IP;
IP[31:2] = effective_address(temp[31:2]);
IP[1:0] = 0;
if ((temp.type == local) ||  (PC.em == supervisor))
        {               # Local call or supervisor call from supervisor mode.
                tempa =  (SP + (SALIGN*16 - 1)) & ~(SALIGN*16 - 1)
                # Round stack pointer to next boundary.
                # SALIGN=1 on 80303.
                temp.RRR = 000₂;
        }
        else            # Supervisor call from user mode.
        {       tempa = SSP;            # Get Supervisor Stack pointer.
                temp.RRR = 010₂ | PC.te;
                PC.em = supervisor;
```

Table 6-17. calls (Sheet 2 of 2)

```
                                        PC.te = temp.te;
                                }
        PFP = FP;
        PFP.rrr = temp.RRR;
        FP = tempa;
        SP = tempa + 64;
```

Faults:	STANDARD		Refer to Section 6.1.6, "Faults" on page 6-5.
	PROTECTION.LENGTH		Specifies a procedure number greater than 259.

Example:

```
calls r12                   # IP = value obtained from
                            # procedure table for procedure
                            # number given in r12.
calls 3                     # Call procedure 3.
```

Opcode:	**calls**	660H	REG

See Also: **bal, call, callx, ret**

6.2.15 callx

Table 6-18. callx

Mnemonic: **callx** Call Extended

Format: **callx** *targ*

mem

Description: Calls new procedure. *targ* specifies IP of called procedure's first instruction.

In executing **callx**, the processor performs a local call as described in Section 7.1.3.1, "Call Operation" on page 7-6. As part of this operation, the processor allocates a new set of local registers and a new stack frame for the called procedure. Processor then goes to the instruction specified with *targ* and begins execution of new procedure.

callx performs the same operation as call except the target instruction can be farther than -2^{23} to $(2^{23} - 4)$ bytes from current IP.

The *targ* operand is a memory type, which allows the full range of addressing modes to be used to specify the IP of the target instruction. The "IP + displacement" addressing mode allows the instruction to be IP-relative. Indirect calls can be performed by placing the target address in a register and then using one of the register-indirect addressing modes.

Refer to Chapter 2, "Data Types and Memory Addressing Modes" for more information.

Action:

```
#       Wait for any uncompleted instructions to finish;
implicit_syncf();
        temp = (SP + (SALIGN*16 - 1)) & ~(SALIGN*16 - 1)
                # Round stack pointer to next boundary.
                # SALIGN=1 on 80303.
RIP = IP;
if (register_set_available)
        allocate_new_frame( );
else
        {       save_register_set( );  # Save register set in memory at its FP;
                allocate_new_frame( );
        }
        #       Local register references now refer to new frame.
IP[31:2] = effective_address(targ[31:2]);
IP[1:0] = 0;
PFP = FP;
FP = temp;
SP = temp + 64;
```

Faults: STANDARD Refer to Section 6.1.6, "Faults" on page 6-5.

Example:

```
callx (g5)              # IP = (g5), where the address in g5
                        # is the address of the new procedure.
```

Opcode: **callx** 86H MEM

See Also: **bal, call, calls, ret**

6.2.16 chkbit

Table 6-19. chkbit

Mnemonic:	**chkbit**	Check Bit	
Format:	**chkbit**	*bitpos*,	*src2*
		reg/lit	reg/lit
Description:	Checks bit in *src2* designated by *bitpos* and sets condition code according to value found. If bit is set, condition code is set to 010_2; if bit is clear, condition code is set to 000_2.		

Action:	if (((src2 & 2**(bitpos % 32)) == 0)
	AC.cc = 000_2;
	else
	AC.cc = 010_2;		

Faults:	STANDARD		Refer to Section 6.1.6, "Faults" on page 6-5.
Example:	`chkbit 13, g8`		`# Checks bit 13 in g8 and sets` `# AC.cc according to the result.`
Opcode:	**chkbit**	5AEH	REG
See Also:	**alterbit, clrbit, notbit, setbit, cmpi, cmpo**		
Side Effects:	Sets the condition code in the arithmetic controls.		

6.2.17 clrbit

Table 6-20. clrbit

Mnemonic:	**clrbit**	Clear Bit		
Format:	**clrbit**	*bitpos*,	*src*,	*dst*
		reg/lit	reg/lit	reg
Description:	Copies *src* value to *dst* with one bit cleared. *bitpos* operand specifies bit to be cleared.			
Action:	dst = src & ~(2**(bitpos%32));			
Faults:	STANDARD		Refer to Section 6.1.6, "Faults" on page 6-5.	
Example:	`clrbit 23, g3, g6`		`# g6 = g3 with bit 23 cleared.`	
Opcode:	**clrbit**	58CH	REG	
See Also:	**alterbit, chkbit, notbit, setbit**			

6.2.18 cmpdeci, cmpdeco

Table 6-21. cmpdeci, cmpdeco

Mnemonic: **cmpdeci** Compare and Decrement Integer

cmpdeco Compare and Decrement Ordinal

Format:

cmpdec*	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg

Description: Compares *src2* and *src1* values and sets the condition code according to comparison results. *src2* is then decremented by one and result is stored in *dst*. The following table shows condition code setting for the three possible results of the comparison.

Condition Code	Comparison
100_2	*src1* < *src2*
010_2	*src1* = *src2*
001_2	*src1* > *src2*

These instructions are intended for use in ending iterative loops. For **cmpdeci**, integer overflow is ignored to allow looping down through the minimum integer values.

Action:

```
if(src1 < src2)
        AC.cc = 100₂;
else if(src1 == src2)
        AC.cc = 010₂;
else
        AC.cc = 001₂;
dst = src2 -1;          # Overflow suppressed for cmpdeci.
```

Faults: STANDARD — Refer to Section 6.1.6, "Faults" on page 6-5.

Example:

```
cmpdeci 12, g7, g1      # Compares g7 with 12 and sets
                        # AC.cc to indicate the result
                        # g1 = g7 - 1.
```

Opcode:

cmpdeci	5A7H	REG
cmpdeco	5A6H	REG

See Also: **cmpinco, cmpo, cmpi, cmpinci, COMPARE AND BRANCH<cc>**

Side Effects: Sets the condition code in the arithmetic controls.

6.2.19 cmpinci, cmpinco

Table 6-22. cmpinci, cmpinco

Mnemonic: **cmpinci** Compare and Increment Integer

cmpinco Compare and Increment Ordinal

Format:

cmpinc*	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg

Description: Compares *src2* and *src1* values and sets the condition code according to comparison results. *src2* is then incremented by one and result is stored in *dst*. The following table shows condition code settings for the three possible comparison results.

Condition Code	Comparison
100_2	*src1 < src2*
010_2	*src1 = src2*
001_2	*src1 > src2*

These instructions are intended for use in ending iterative loops. For **cmpinci**, integer overflow is ignored to allow looping up through the maximum integer values.

Action:

```
if (src1 < src2)
        AC.cc = 100₂;
else if (src1 == src2)
        AC.cc = 010₂;
else
        AC.cc = 001₂;
dst = src2 + 1;                 # Overflow suppressed for cmpinci.
```

Faults: STANDARD — Refer to Section 6.1.6, "Faults" on page 6-5.

Example:

```
cmpinco r8, g2, g9      # Compares the values in g2
                        # and r8 and sets AC.cc to
                        # indicate the result:
                        # g9 = g2 + 1
```

Opcode:

cmpinci	5A5H	REG
cmpinco	5A4H	REG

See Also: **cmpdeco, cmpo, cmpi, cmpdeci, COMPARE AND BRANCH<cc>**

Side Effects: Sets the condition code in the arithmetic controls.

6.2.20 COMPARE

Table 6-23. COMPARE

Mnemonic:

cmpi	Compare Integer
cmpib	Compare Integer Byte
cmpis	Compare Integer Short
cmpo	Compare Ordinal
cmpob	Compare Ordinal Byte
cmpos	Compare Ordinal Short

Format:

cmp*	*src1*,	*src2*
	reg/lit	reg/lit

Description:

Compares *src2* and *src1* values and sets condition code according to comparison results. The table below shows condition code settings for the three possible comparison results.

Condition Code	Comparison
100_2	*src1* < *src2*
010_2	*src1* = *src2*
001_2	*src1* > *src2*

cmpi* followed by a branch-if instruction is equivalent to a compare-integer-and-branch instruction. The latter method of comparing and branching produces more compact code; however, the former method can execute byte and short compares without masking. The same is true for **cmpo*** and the compare-ordinal-and-branch instructions.

Action:

```
# For cmpo, cmpi, N = 31.
# For cmpos, cmpis, N = 15
# For cmpob, cmpib, N = 7.

if   (src1[N:0] < src2[N:0])
          AC.cc = 100₂;
else if (src1[N:0] == src2[N:0])
          AC.cc = 010₂;
else if (src1[N:0] > src2[N:0])
          AC.cc = 001₂;
```

Faults: STANDARD — Refer to Section 6.1.6, "Faults" on page 6-5.

Example:

```
cmpo r9, 0x10          # Compares the value in r9 with
                       # 0x10 and sets AC.cc to indicate
                       # the result
bg xyz                 # Branches to xyz if the value of
                       # r9 was greater than 0x10.
```

Opcode:

cmpi	5A1H	REG
cmpib	595H	REG
cmpis	597H	REG
cmpo	5A0H	REG
cmpob	594H	REG
cmpos	596H	REG

See Also: **COMPARE AND BRANCH<cc>, cmpdeci, cmpdeco, cmpinci, cmpinco, concmpi, concmpo**

Side Effects: Sets the condition code in the arithmetic controls.

Notes: The core instructions **cmpib**, **cmpis**, **compob** and **compos** are not implemented on 80960Cx, Kx and Sx processors.

6.2.21 COMPARE AND BRANCH<cc>

Table 6-24. COMPARE AND BRANCH<cc>

Mnemonic:	**cmpibe**	Compare Integer and Branch If Equal		
	cmpibne	Compare Integer and Branch If Not Equal		
	cmpibl	Compare Integer and Branch If Less		
	cmpible	Compare Integer and Branch If Less Or Equal		
	cmpibg	Compare Integer and Branch If Greater		
	cmpibge	Compare Integer and Branch If Greater Or Equal		
	cmpibo	Compare Integer and Branch If Ordered		
	cmpibno	Compare Integer and Branch If Not Ordered		
	cmpobe	Compare Ordinal and Branch If Equal		
	cmpobne	Compare Ordinal and Branch If Not Equal		
	cmpobl	Compare Ordinal and Branch If Less		
	cmpoble	Compare Ordinal and Branch If Less Or Equal		
	cmpobg	Compare Ordinal and Branch If Greater		
	cmpobge	Compare Ordinal and Branch If Greater Or Equal		
Format:	**cmpib***	*src1*,	*src2*,	*targ*
		reg/lit	reg	disp
	cmpob*	*src1*,	*src2*,	*targ*
		reg/lit	reg	disp

Description:

Compares *src2* and *src1* values and sets AC register condition code according to comparison results. If logical AND of condition code and mask part of opcode is not zero, the processor branches to instruction specified with *targ*; otherwise, the processor goes to next instruction.

targ can be no farther than -2^{12} to $(2^{12} - 4)$ bytes from current IP. When using the Intel i960 processor assembler, *targ* must be a label that specifies target instruction's IP.

Functions these instructions perform can be duplicated with a **cmpi** or **cmpo** followed by a branch-if instruction, as described in Section 6.2.20, "COMPARE" on page 6-29.

The following table shows the condition-code mask for each instruction. The mask is in bits 0-2 of the opcode.

Instruction	Mask	Branch Condition
cmpibno	000_2	No Condition
cmpibg	001_2	*src1* > *src2*
cmpibe	010_2	*src1* = *src2*
cmpibge	011_2	*src1* ≥ *src2*
cmpibl	100_2	*src1* < *src2*
cmpibne	101_2	*src1* ≠ *src2*
cmpible	110_2	*src1* ≤ *src2*
cmpibo	111_2	Any Condition
cmpobg	001_2	*src1* > *src2*
cmpobe	010_2	*src1* = *src2*
cmpobge	011_2	*src1* ≥ *src2*
cmpobl	100_2	*src1* < *src2*
cmpobne	101_2	*src1* ≠ *src2*
cmpoble	110_2	*src1* ≤ *src2*

cmpibo always branches; **cmpibno** never branches.

Action:

```
if(src1 < src2
        AC.cc = 100₂;
else if(src1 == src2)
        AC.cc = 010₂;
else
        AC.cc = 001₂;
if((mask && AC.cc) != 000₂)
        IP[31:2] = efa[31:2];        # Resume execution at the new IP.
        IP[1:0] = 0;
```

Faults: STANDARD — Refer to Section 6.1.6, "Faults" on page 6-5.

Example:

```
# Assume g3 < g9
cmpibl g3, g9, xyz        # g9 is compared with g3;
                          # IP = xyz.

# assume 19 ≥ r7          # 19 is compared with r7;
cmpobge 19, r7, xyz       # IP = xyz.
```

Opcode:

cmpibe	3AH	COBR
cmpibne	3DH	COBR
cmpibl	3CH	COBR
cmpible	3EH	COBR
cmpibg	39H	COBR
cmpibge	3BH	COBR
cmpibo	3FH	COBR
cmpibno	38H	COBR
cmpobe	32H	COBR
cmpobne	35H	COBR
cmpobl	34H	COBR
cmpoble	36H	COBR
cmpobg	31H	COBR
cmpobge	33H	COBR

See Also: **BRANCH<cc>, cmpi, cmpo, bal, balx**

Side Effects: Sets the condition code in the arithmetic controls.

6.2.22 concmpi, concmpo

Table 6-25. concmpi, concmpo

Mnemonic: **concmpi** Conditional Compare Integer
concmpo Conditional Compare Ordinal

Format:

concmp*	*src1*,	*src2*
	reg/lit	reg/lit

Description: Compares *src2* and *src1* values if condition code bit 2 is not set. If comparison is performed, condition code is set according to comparison results. Otherwise, condition codes are not altered.

These instructions are provided to facilitate bounds checking by means of two-sided range comparisons (e.g., is A between B and C?). They are generally used after a compare instruction to test whether a value is inclusively between two other values.

The example below illustrates this application by testing whether g3 value is between g5 and g6 values, where g5 is assumed to be less than g6. First a comparison (**cmpo**) of g3 and g6 is performed. If g3 is less than or equal to g6 (i.e., condition code is either 010_2 or 001_2), a conditional comparison (**concmpo**) of g3 and g5 is then performed. If g3 is greater than or equal to g5 (indicating that g3 is within the bounds of g5 and g6), condition code is set to 010_2; otherwise, it is set to 001_2.

Action:

```
if (AC.cc != 1XX2)
{       if(src1 <= src2)
                AC.cc = 0102;
        else
                AC.cc  = 0012;
}
```

Faults: STANDARD Refer to Section 6.1.6, "Faults" on page 6-5.

Example:

```
cmpo g6, g3             # Compares g6 and g3
                        # and sets AC.cc.
concmpo g5, g3          # If AC.cc < 1002 (g6 Š g3)
                        # g5 is compared with g3.
```

At this point, depending on the register ordering, the condition code is one of those listed in the table below.

Order	CC
g5 < g6 < g3	100_2
g5 < g6 = g3	010_2
g5 < g3 < g6	010_2
g5 = g3 < g6	010_2
g3 < g5 < g6	001_2

Opcode:

concmpi	5A3H	REG
concmpo	5A2H	REG

See Also: **cmpo, cmpi, cmpdeci, cmpdeco, cmpinci, cmpinco, COMPARE AND BRANCH<cc>**

Side Effects: Sets the condition code in the arithmetic controls.

6.2.23 dcctl

Table 6-26. dcctl

Mnemonic: **dcctl** Data-cache Control

Format:	*src1,*	*src2,*	*src/dst*
	reg/lit	reg/lit	reg

Description: Performs management and control of the data cache including disabling, enabling, invalidating, ensuring coherency, getting status, and storing cache contents to memory. Operations are indicated by the value of *src1*. *src2* and *src/dst* are also used by some operations. When needed by the operation, the processor orders the effects of the operation with previous and subsequent operations to ensure correct behavior. The table below shows dcctl operand fields.

Function	src1	src2	src/dst
Disable D-cache	0	NA	NA
Enable D-cache	1	NA	NA
Global invalidate D-cache	2	NA	NA
Ensure cache coherency1	3	NA	NA
Get D-cache status	4	NA	*src:* NA *dst:* Receives D-cache status (Figure 6-1).
Reserved	5	NA	NA
Store D-cache to memory	6	Destination address for cache sets	*src*: D-cache set #'s to be stored (Figure 6-1).
Reserved	7	NA	NA
Quick invalidate	8	1	NA
Reserved	9	NA	NA

1. Invalidates data cache on 80303.

Figure 6-1. dcctl *src1* and *src/dst* Formats

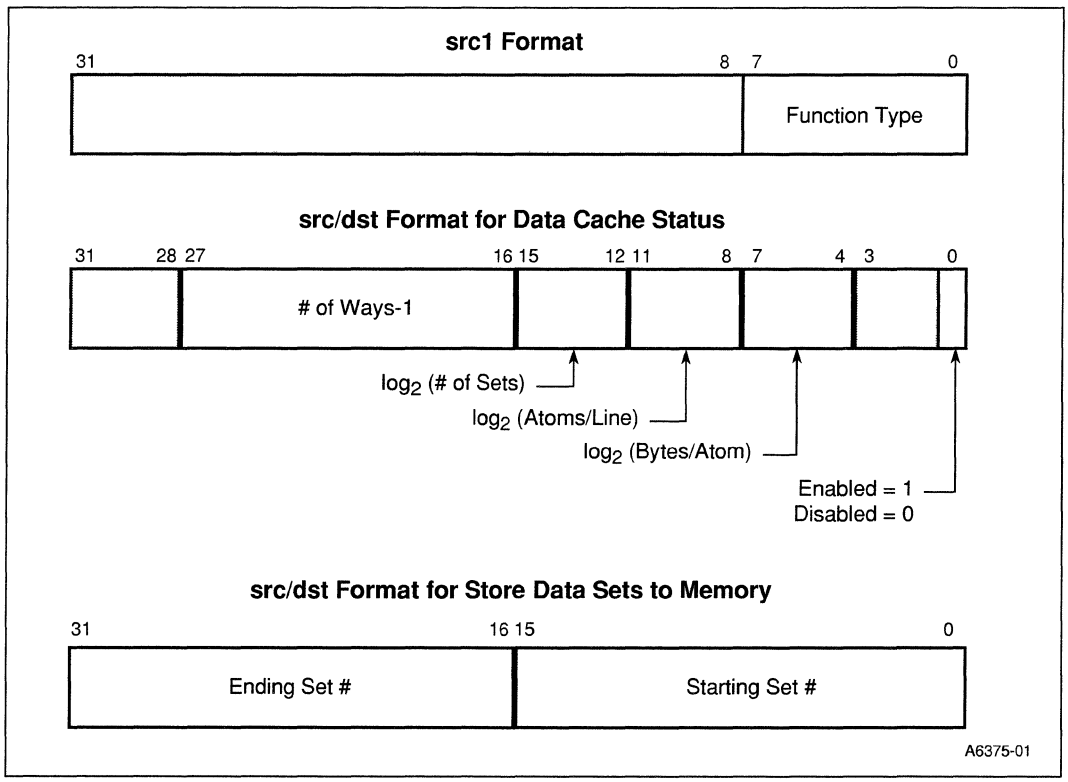


Table 6-27. dcctl Status Values and D-Cache Parameters

Value	Value on Intel® 80303 I/O Processor
bytes per atom	4
atoms per line	4
number of sets	256 (full)
number of ways	1 (Direct)
cache size	4-Kbytes (full)
Status[0] (enable / disable)	0 or 1
Status[1:3] (reserved)	0
Status[7:4] ($\log_2$(bytes per atom))	2
Status[11:8] ($\log_2$(atoms per line))	2
Status[15:12] ($\log_2$(number of sets))	8 (full)
Status[27:16] ($\log_2$(number of ways - 1))	0

Figure 6-2. Store Data Cache to Memory Output Format

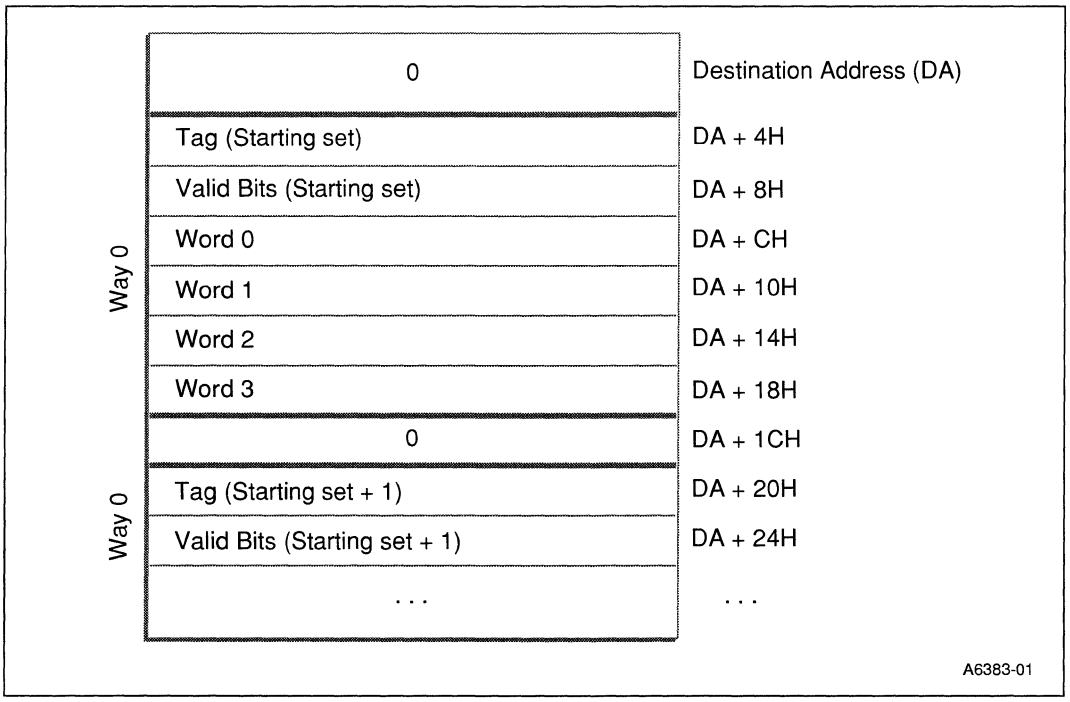
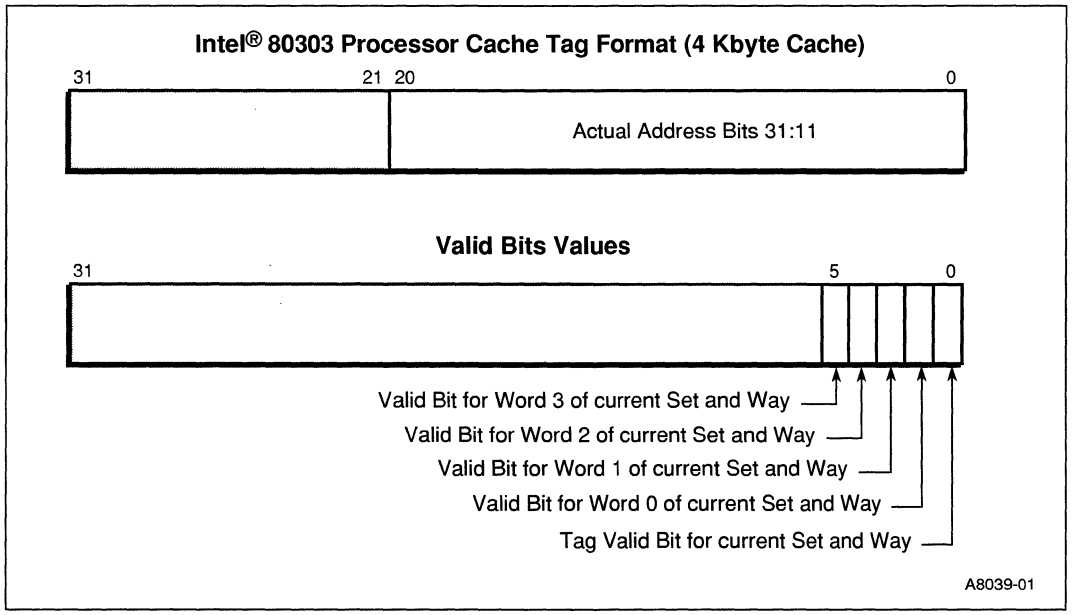


Figure 6-3. D-Cache Tag and Valid Bit Formats



Action:

```
if (PC.em != supervisor)
    generate_fault(TYPE.MISMATCH);
order_wrt(previous_operations);
switch (src1[7:0]) {
    case 0:     # Disable data cache
                disable_Dcache( );
                break;
    case 1:     # Enable data cache.
                enable_Dcache( );
                break;
    case 2:     # Global invalidate data cache.
                invalidate_Dcache( );
                break;
    case 3:     # Ensure coherency of data cache with memory.
                # Causes data cache to be invalidated on this processor.
                ensure_Dcache_coherency( );
                break;
    case 4:     # Get data cache status into src_dst.
                if (Dcache_enabled) src_dst[0] = 1;
                else src_dst[0] = 0;
                # Atom is 4 bytes.
                src_dst[7:4] = log2(bytes per atom);
                # 4 atoms per line.
                src_dst[11:8] = log2(atoms per line);
                src_dst[15:12] = log2(number of sets);
                src_dst[27:16] = number of ways-1; # in lines per set
                # cache size = ([27:16]+1) << ([7:4] + [11:8] + [15:12]).
                break;
    case 6:  # Store data cache sets to memory pointed to by src2.
                start = src_dst[15:0]    #    Starting set number.
                end  = src_dst[31:16]    #    Ending set number.
                                         #    (zero-origin)
                if (end >= Dcache_max_sets) end = Dcache_max_sets - 1;
                if (start > end) generate_fault
                        (OPERATION.INVALID_OPERAND);
                memadr = src2;           #    Must be word-aligned.
                if (0x3 & memadr! = 0)
                generate_fault(OPERATION.INVALID_OPERAND)
                for (set = start; set <= end; set++){
                        # Set_Data is described at end of this code flow.
```

```
                                        memory[memadr] = Set_Data[set];
                                        memadr += 4;
                                        for (way = 0; way < numb_ways; way++)
                                                {memory[memadr] = tags[set][way];
                                                 memadr += 4;
                                                memory[memadr] = valid_bits[set][way];
                                                memadr += 4;
                                                for (word = 0; word < words_in_line;
                                                word++)
                                                { memory[memadr] =
                                                        Dcache_line[set][way][word];
                                                 memadr += 4;
                                                }
                                                }
                        }
                        break;
        default:        # Reserved.
                        generate_fault(OPERATION.INVALID_OPERAND);
                        break;
        }
order_wrt(subsequent_operations);
```

Faults:

STANDARD	Refer to Section 6.1.6, "Faults" on page 6-5.
TYPE.MISMATCH	Attempt to execute instruction while not in supervisor mode.
OPERATION.INVALID_OPERAND	

Example:

```
                                # g0 = 6, g1 = 0x10000000,
                                # g2 = 0x001F0001
dcctl g0,g1,g2                  # Store the status of D-cache
                                # sets 1-0x1F to memory starting
                                # at 0x10000000.
```

Opcode:

dcctl	65CH	REG

See Also: **sysctl**

Notes: DCCTL function 6 stores data-cache sets to a target range in external memory. For any memory location that is cached and also within the target range for function 6, the corresponding word-valid bit is cleared after function 6 completes to ensure data-cache coherency. Thus, **dcctl** function 6 can alter the state of the cache after it completes, but only the word-valid bits. In all cases, even when the cache sets to store to external memory overlap the cache sets that map the target range in external memory, DCCTL function 6 always returns the state of the cache as it existed when the DCCTL was issued.

This instruction is implemented on the 80303, 80960RM/RN, 80960RP/RD, 80960Hx, and 80960Jx processor families only, and may or may not be implemented on future i960 processors.

6.2.24 divi, divo

Table 6-28. divi, divo

Mnemonic:	**divi**	Divide Integer		
	divo	Divide Ordinal		
Format:	**div***	*src1*,	*src2*,	*dst*
		reg/lit	reg/lit	reg

Description: Divides *src2* value by *src1* value and stores the result in *dst*. Remainder is discarded.

For **divi**, an integer-overflow fault can be signaled.

Action:

```
divo:
if (src1 == 0)
{       dst = undefined_value;
        generate_fault (ARITHMETIC.ZERO_DIVIDE);
else
        dst = src2/src1;
divi:
if (src1 == 0)
{       dst = undefined_value;
        generate_fault (ARITHMETIC.ZERO_DIVIDE);}
else    if ((src2 == -2**31) && (src1 == -1)
        {       dst = -2**31
        if (AC.om == 1
                AC.of  = 1;
        else
                generate_fault (ARITHMETIC.OVERFLOW);
        }
else
        dst  = src2 / src1;
```

Faults:	STANDARD	Refer to Section 6.1.6, "Faults" on page 6-5.
	ARITHMETIC.ZERO_DIVIDE	The *src1* operand is 0.
	ARITHMETIC.OVERFLOW	Result too large for destination register (**divi** only). If overflow occurs and AC.om=1, fault is suppressed and AC.of is set to 1. Result's least significant 32 bits are stored in *dst*.

Example:

```
divo r3, r8, r13        # r13 = r8/r3
```

Opcode:	**divi**	74BH	REG
	divo	70BH	REG

See Also: **ediv, mulo, muli, emul**

6.2.25 ediv

Table 6-29. ediv

Mnemonic:	**ediv**	Extended Divide		
Format:	**ediv**	*src1*,	*src2*,	*dst*
		reg/lit	reg/lit	reg

Description:

Divides *src2* by *src1* and stores result in *dst*. The *src2* value is a long ordinal (64 bits) contained in two adjacent registers. *src2* specifies the lower numbered register which contains operand's least significant bits. *src2* must be an even numbered register (i.e., g0, g2, ... or r4, r6, r8...). *src1* value is a normal ordinal (i.e., 32 bits).

The result consists of a one-word remainder and a one-word quotient. Remainder is stored in the register designated by *dst*; quotient is stored in the next highest numbered register. *dst* must be an even numbered register (i.e., g0, g2, ... r4, r6, r8, ...).

This instruction performs ordinal arithmetic.

If this operation overflows (quotient or remainder do not fit in 32 bits), no fault is raised and the result is undefined.

Action:

```
if((reg_number(src2)%2 != 0) || (reg_number(dst)%2 != 0))
{       dst[0] = undefined_value;
        dst[1] = undefined_value;
        generate_fault (OPERATION.INVALID_OPERAND);
}
else if(src1 == 0)
{       dst[0] = undefined_value;
        dst[1] = undefined_value;
        generate_fault(ARITHMETIC.DIVIDE_ZERO);
}
else                    # Quotient
{       dst[1] = ((src2 + reg_value(src2[1]) * 2**32) / src1)[31:0];
        #Remainder
        dst[0] = (src2 + reg_value(src2[1]) * 2**32
                - ((src2 + reg_value(src2[1]) * 2**32 / src1) * src1);
}
```

Faults:	STANDARD	Refer to Section 6.1.6, "Faults" on page 6-5.
	ARITHMETIC.ZERO_DIVIDE	The *src1* operand is 0.

Example:

```
ediv g3, g4, g10        # g10 = remainder of g4,g5/g3
                        # g11 = quotient of g4,g5/g3
```

Opcode:	**ediv**	671H	REG

See Also: **emul, divi, divo**

6.2.26 emul

Table 6-30. emul

Mnemonic:	**emul**	Extended Multiply		
Format:	**emul**	*src1*,	*src2*,	*dst*
		reg/lit	reg/lit	reg

Description: Multiplies *src2* by *src1* and stores the result in *dst*. Result is a long ordinal (64 bits) stored in two adjacent registers. *dst* specifies lower numbered register, which receives the result's least significant bits. *dst* must be an even numbered register (i.e., g0, g2, ... r4, r6, r8, ...).

This instruction performs ordinal arithmetic.

Action:

```
if(reg_number(dst)%2 != 0)
{       dst[0] = undefined_value;
        dst[1] = undefined_value;
        generate_fault(OPERATION.INVALID_OPERAND);
}
else
{
        dst[0] = (src1 * src2)[31:0];
        dst[1] = (src1 * src2)[63:32];
}
```

Faults: STANDARD — Refer to Section 6.1.6, "Faults" on page 6-5.

Example:

```
emul r4, r5, g2          # g2,g3 = r4 * r5.
```

Opcode: **emul** 670H REG

See Also: **ediv, muli, mulo**

6.2.27 eshro

Table 6-31. eshro

Mnemonic:	**eshro**	Extended Shift Right Ordinal		
Format:	**eshro**	*src1,*	*src2,*	*dst*
		reg/lit	reg/lit	reg

Description: Shifts *src2* right by (*src1* **mod** 32) places and stores the result in *dst*. Bits shifted beyond the least-significant bit are discarded.

src2 value is a long ordinal (i.e., 64 bits) contained in two adjacent registers. *src2* operand specifies the lower numbered register, which contains operand's least significant bits. *src2* operand must be an even numbered register (i.e., r4, r6, r8, ... or g0, g2).

src1 operand is a single 32-bit register or literal where the lower 5 bits specify the number of places that the *src2* operand is to be shifted.

The least significant 32 bits of the shift operation result are stored in *dst*.

Action:

```
if(reg_number(src2)%2 != 0)
{
        dst[0] = undefined_value;
        dst[1] = undefined_value;
        generate_fault(OPERATION.INVALID_OPERAND);
}
else
        dst = shift_right((src2 + reg_value(src2[1]) * 2**32),(src1%32))[31:0];
```

Faults: STANDARD — Refer to Section 6.1.6, "Faults" on page 6-5.

Example:

```
eshro g3, g4, g11       # g11 = g4,5 shifted right by
                        # (g3 MOD 32).
```

Opcode:	**eshro**	5D8H	REG

See Also: **SHIFT, extract**

Notes: This core instruction is not implemented on the 80960Kx and Sx processors.

6.2.28 extract

Table 6-32. extract

Mnemonic:	**extract**	Extract		
Format:	**extract**	*bitpos*	*len*	*src/dst*
		reg/lit	reg/lit	reg
Description:	Shifts a specified bit field in *src/dst* right and zero fills bits to left of shifted bit field. *bitpos* value specifies the least significant bit of the bit field to be shifted; *len* value specifies bit field length.			
Action:	src_dst = (src_dst >> min(bitpos, 32)) & ~ (0xFFFFFFFF << len);			
Faults:	STANDARD	Refer to Section 6.1.6, "Faults" on page 6-5.		
Example:	`extract 5, 12, g4`	`# g4 = g4 with bits 5 through 16 shifted right.`		
Opcode:	**extract**	651H	REG	
See Also:	**modify**			

6.2.29 FAULT<cc>

Table 6-33. FAULT<cc>

Mnemonic:

faulte	Fault If Equal
faultne	Fault If Not Equal
faultl	Fault If Less
faultle	Fault If Less Or Equal
faultg	Fault If Greater
faultge	Fault If Greater Or Equal
faulto	Fault If Ordered
faultno	Fault If Not Ordered

Format: **fault***

Description: Raises a constraint-range fault if the logical AND of the condition code and opcode's mask part is not zero. For **faultno** (unordered), fault is raised if condition code is equal to 000_2.

faulto and **faultno** are provided for use by implementations with a floating point coprocessor. They are used for compare and branch (or fault) operations involving real numbers.

The table below shows condition-code mask for each instruction (mask is opcode bits 0-2).

Instruction	Mask	Condition
faultno	000_2	Unordered
faultg	001_2	Greater
faulte	010_2	Equal
faultge	011_2	Greater or equal
faultl	100_2	Less
faultne	101_2	Not equal
faultle	110_2	Less or equal
faulto	111_2	Ordered

Action:

For all except **faultno**:

if(mask && AC.cc != 000_2

    generate_fault(CONSTRAINT.RANGE);

faultno:

if(AC.cc == 000_2

    generate_fault(CONSTRAINT.RANGE);

Faults: STANDARD — Refer to Section 6.1.6, "Faults" on page 6-5.

Faults: CONSTRAINT.RANGE — If condition being tested is true.

Example:

```
# Assume (AC.cc AND 110₂)≠000₂
faultle                    # Generate CONSTRAINT_RANGE fault
```

Opcode:

faulte	1AH	CTRL
faultne	1DH	CTRL
faultl	1CH	CTRL
faultle	1EH	CTRL
faultg	19H	CTRL
faultge	1BH	CTRL
faulto	1FH	CTRL
faultno	18H	CTRL

See Also: **BRANCH<cc>, TEST<cc>**

6.2.30 flushreg

Table 6-34. flushreg

Mnemonic:	**flushreg**	Flush Local Registers	
Format:	**flushreg**		
Description:	Copies the contents of every cached register set, except the current set, to its associated stack frame in memory. The entire register cache is then marked as purged (or invalid). On a return to a stack frame for which the local registers are not cached, the processor reloads the locals from memory. **flushreg** is provided to allow a debugger or application program to circumvent the processor's normal call/return mechanism. For example, a debugger may need to go back several frames in the stack on the next return, rather than using the normal return mechanism that returns one frame at a time. Since the local registers of an unknown number of previous stack frames may be cached, a **flushreg** must be executed prior to modifying the PFP to return to a frame other than the one directly below the current frame. To reduce interrupt latency, **flushreg** is abortable. If an interrupt of higher priority than the current process is detected while **flushreg** is executing, **flushreg** flushes at least one frame and aborts. After executing the interrupt handler, the processor returns to the **flushreg** instruction and re-executes it. **flushreg** does not reflush any frames that were flushed before the interrupt occurred. **flushreg** is not aborted by high priority interrupts if tracing is enabled in the PC or if any faults are pending at the time of the interrupt.		
Action:	Each local cached register set except the current one is flushed to its associated stack frame in memory and marked as purged, meaning that they are reloaded from memory if and when they become the current local register set.		
Faults:	STANDARD	Refer to Section 6.1.6, "Faults" on page 6-5.	
Example:	`flushreg`		
Opcode:	**flushreg**	66DH	REG

6.2.31 fmark

Table 6-35. fmark

Mnemonic:	**fmark**	Force Mark	
Format:	**fmark**		
Description:	Generates a mark trace event. Causes a mark trace event to be generated, regardless of mark trace mode flag setting, providing the trace enable bit, bit 0 in the Process Controls, is set. For more information on trace fault generation, refer to Chapter 10, "Tracing and Debugging".		
Action:	A mark trace event is generated, independent of the setting of the mark-trace-mode flag.		
Faults:	STANDARD		Refer to Section 6.1.6, "Faults" on page 6-5.
Faults:	TRACE.MARK		A TRACE.MARK fault is generated if PC.te=1.

Example:	`# Assume PC.te = 1`
	`fmark`
	`# Mark trace event is generated at this point in the`
	`# instruction stream.`		

Opcode:	**fmark**	66CH	REG
See Also:	**mark**		

6.2.32 halt

Table 6-36. halt

Mnemonic: **halt** Halt CPU

Format: **halt** *src1*

reg/lit

Description: Causes the i960 core processor to enter HALT mode. Entry into Halt mode allows the interrupt enable state to be conditionally changed based on the value of *src1*.

The processor exits Halt mode on a hardware reset or upon receipt of an interrupt that should be delivered based on the current process priority. After executing the interrupt that forced the processor out of Halt mode, execution resumes at the instruction immediately after the **halt** instruction. The processor must be in supervisor mode to use this instruction.

src1	Operation
0	Disable interrupts and halt
1	Enable interrupts and halt
2	Use current interrupt enable state and halt

Action:

```
implicit_syncf;
if (PC.em != supervisor
        generate_fault(TYPE.MISMATCH);
switch(src1) {
        case 0:     # Disable interrupts. set ICON.gie.
                    global_interrupt_enable = true;          break;
        case 1:     # Enable interrupts. clear ICON.gie.
                    global_interrupt_enable = false;         break;
        case 2:     # Use the current interrupt enable state.
                    break;
        default:
                    generate_fault(OPERATION.INVALID_OPERAND);
                    break;
}

ensure_bus_is_quiescient;
enter_HALT_mode;
```

Faults:

STANDARD	Refer to Section 6.1.6, "Faults" on page 6-5.
TYPE.MISMATCH	Attempt to execute instruction while not in supervisor mode.

Example:

```
                    # ICON.gie = 1, g0 = 1, Interrupts
                    disabled.
halt g0             # Enable interrupts and halt.
```

Opcode: **halt** 65DH REG

Notes: This instruction is implemented on the 80303, 80960RM/RN, 80960RP/RD, and 80960Jx processor families only, and may or may not be implemented on future i960 processors.

6.2.33 icctl

Table 6-37. icctl

Mnemonic:	**icctl**	Instruction-cache Control		
Format:	**icctl**	*src1,*	*src2,*	*src/dst*
		reg/lit	reg/lit	reg

Description: Performs management and control of the instruction cache including disabling, enabling, invalidating, loading and locking, getting status, and storing cache sets to memory. Operations are indicated by the value of *src1*. Some operations also use *src2* and *src/dst*. When needed by the operation, the processor orders the effects of the operation with previous and subsequent operations to ensure correct behavior. For specific function setup, see the following tables and diagrams:

Table 6-38. icctl Operand Fields

Function	src1	src2	src/dst
Disable I-cache	0	NA	NA
Enable I-cache	1	NA	NA
Invalidate I-cache	2	NA	NA
Load and lock I-cache	3	*src*: Starting address of code to lock.	Number of blocks to lock.
Get I-cache status	4	NA	*dst:* Receives status (Figure 6-4).
Get I-cache locking status	5	NA	*dst:* Receives status (Figure 6-4)
Store I-cache sets to memory	6	Destination address for cache sets	*src*: I-cache set #'s to be stored (Figure 6-4).

Figure 6-4. icctl *src1* and *src/dst* Formats

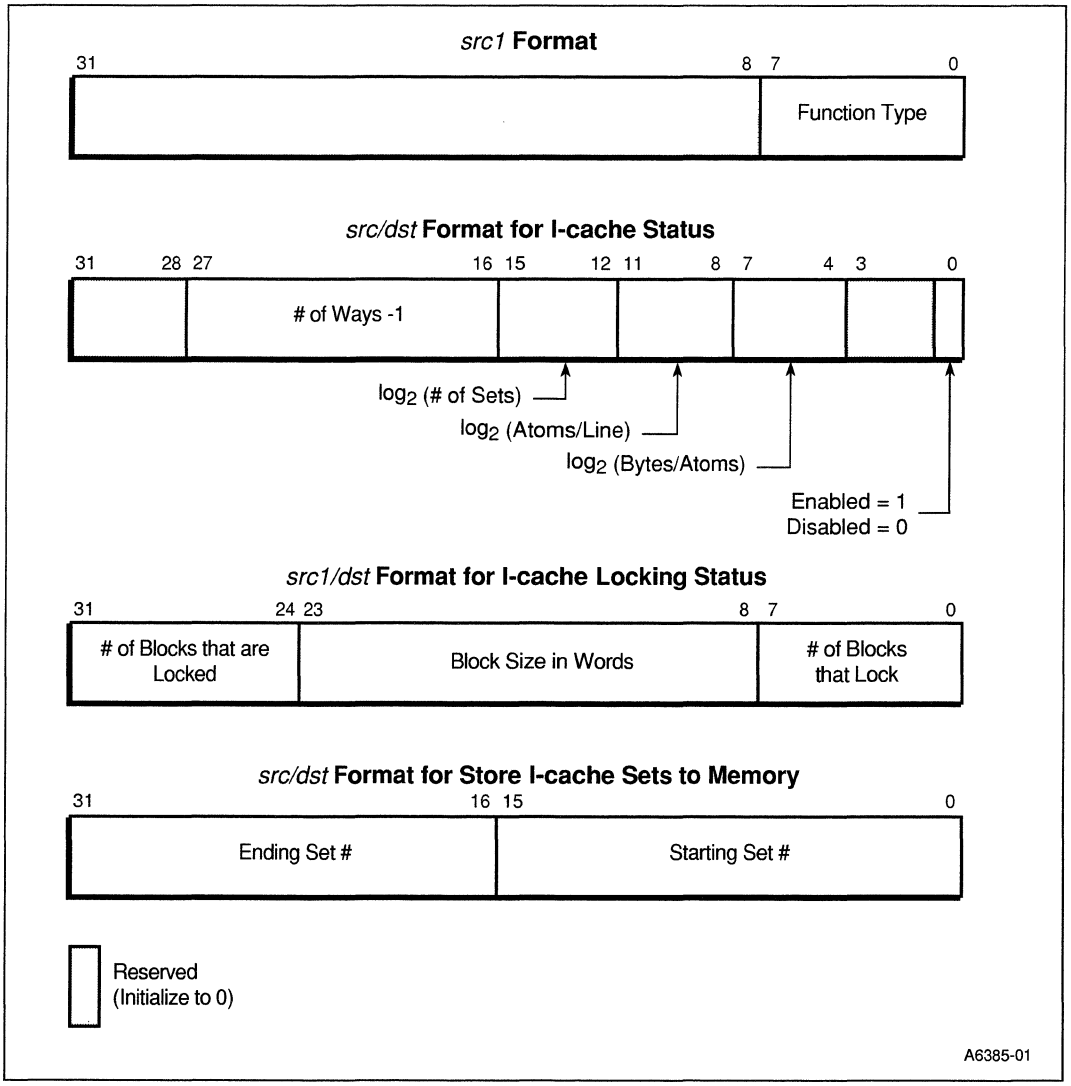


Table 6-39. icctl Status Values and I-Cache Parameters

Value	Value on 80960JN CPU
bytes per atom	4
atoms per line	4
number of sets	512
number of ways	2
cache size	16-Kbytes
Status[0] (enable / disable)	0 or 1
Status[1:3] (reserved)	0
Status[7:4] (log_2(bytes per atom))	2
Status[11:8] (log_2(atoms per line))	2
Status[15:12] (log_2(number of sets))	9
Status[27:16] (number of ways - 1)	1
Lock Status[23:8] (block size in words)	1
Lock Status[23:8] (block size in words)	2048
Lock Status[31:24] (number of blocks that are locked)	0 or 1

Figure 6-5. Store Instruction Cache to Memory Output Format

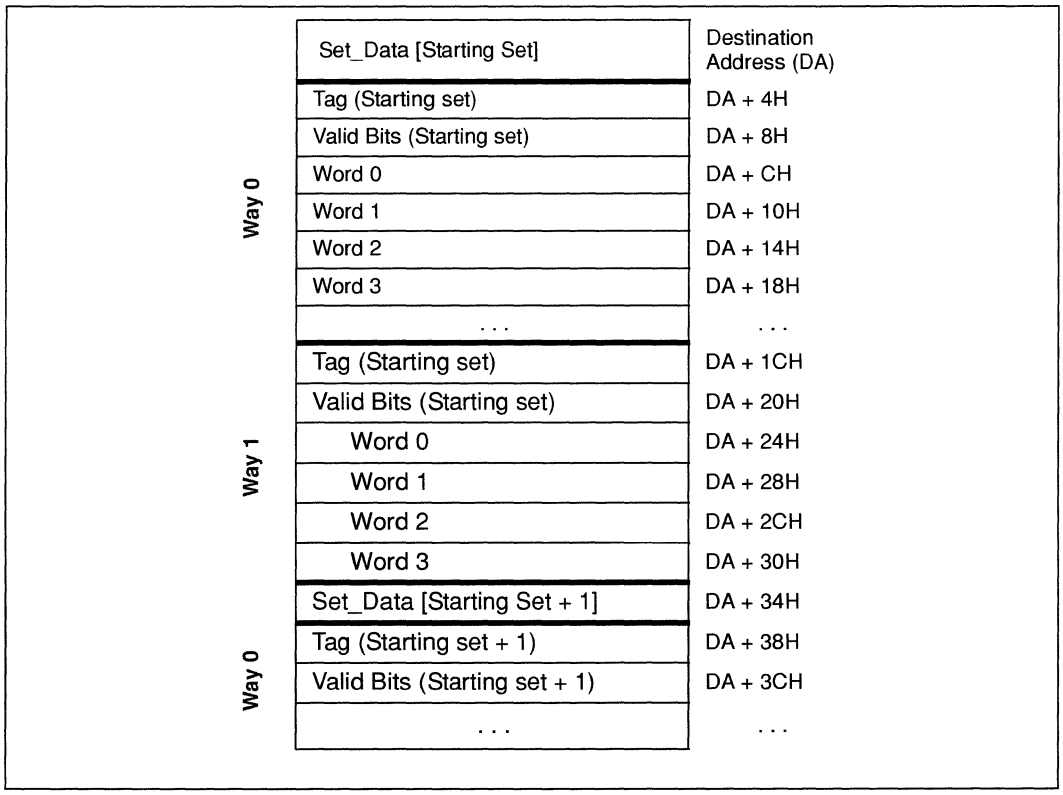
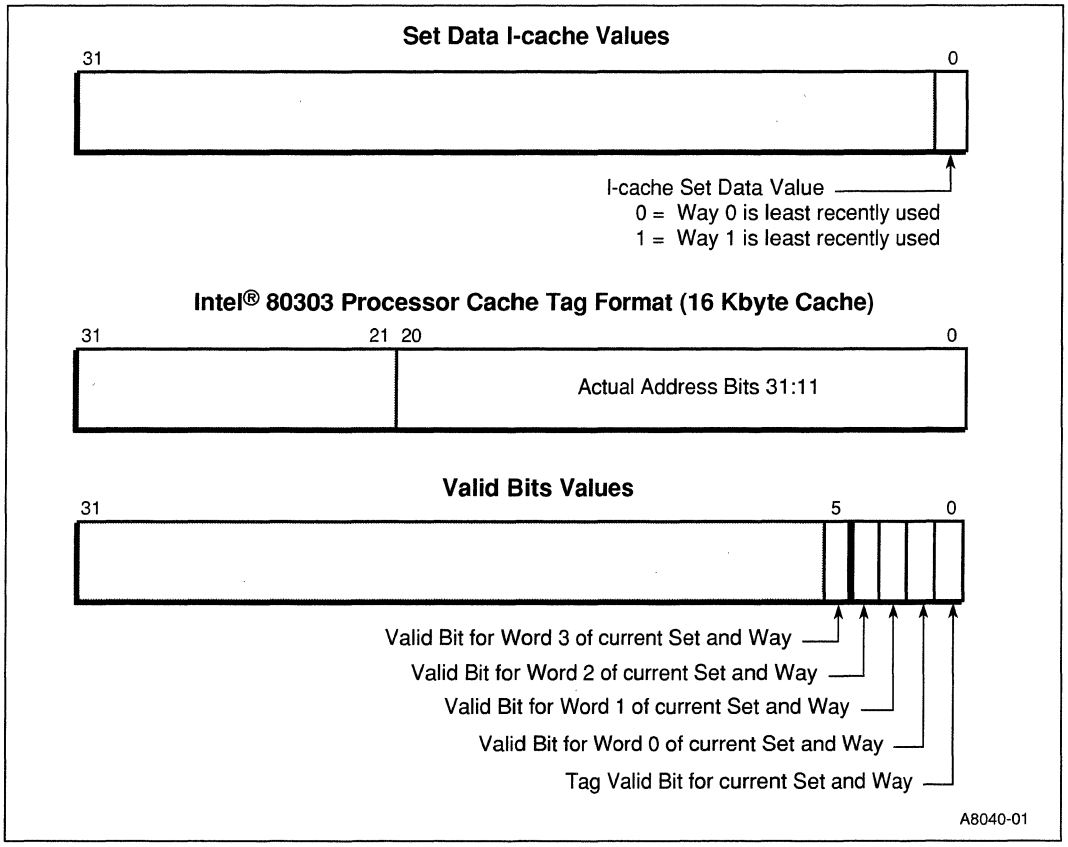


Figure 6-6. I-Cache Set Data, Tag and Valid Bit Formats



Action:

```
if (PC.em != supervisor)
        generate_fault(TYPE.MISMATCH);
switch (src1[7:0]) {
        case 0:  # Disable instruction cache.
                 disable_instruction_cache( );
                 break;
        case 1:  # Enable instruction cache.
                 enable_instruction_cache( );
                 break;
        case 2:  # Globally invalidate instruction cache.
                 # Includes locked lines also.
                 invalidate_instruction_cache( );
                 unlock_icache( );
                 break;
        case 3:  # Load & Lock code into Instruction-Cache
                 # src_dst has number of contiguous blocks to lock.
                 # src2 has starting address of code to lock.
                 # On the i960 RP, src2 is aligned to a quad word boundary
                        aligned_addr = src2 & 0xFFFFFFF0;
                        invalidate(I-cache); unlock(I-cache);
                        for (j = 0; j < src_dst; j++)
                        {       way = way_associated_with_block(j);
                                start = src2 + j*block_size;
                                end = start + block_size;
                                for (i = start; i < end; i=i+4)
                                {       set = set_associated_with(i);
                                        word = word_associated_with(i);
                                        Icache_line[set][way][word] =
                                                memory[i];
                                update_tag_n_valid_bits(set,way,word)
                                        lock_icache(set,way,word);
                                } } break;
        case 4:  # Get instruction cache status into src_dst.
                 if (Icache_enabled) src_dst[0] = 1;
                        else src_dst[0] = 0
                 # Atom is 4 bytes.
                  src_dst[7:4] = log2(bytes per atom);
                 # 4 atoms per line.
                  src_dst[11:8] = log2(atoms per line);
                 src_dst[15:12] = log2(number of sets);
                 src_dst[27:16] = number of ways-1; #in lines per set
                 # cache size = ([27:16]+1) << ([7:4] + [11:8] + [15:12])
                 break;
        case 5:  # Get instruction cache locking status into dst.
                 src_dst[7:0] = number_of_blocks_that_lock;
                 src_dst[23:8] = block_size_in_words;
                 src_dst[31:24] = number_of_blocks_that_are_locked;
                 break;
        case 6:  # Store instr cache sets to memory pointed to by src2.
                 start = src_dst[15:0]          # Starting set number
```

```
                        end  = src_dst[31:16]          # Ending set number
                                                       # (zero-origin)
                        if (end >= lcache_max_sets)
                                end = lcache_max_sets - 1;
                        if (start > end)
                                generate_fault(OPERATION.INVALID_OPERAND);
                        memadr = src2;          # Must be word-aligned.
                        if(0x3 & memadr != 0)
generate_fault(OPERATION.INVALID_OPERAND);
                        for (set = start; set <= end; set++){
                                # Set_Data is described at end of this code flow.
                                memory[memadr] = Set_Data[set];
                                memadr += 4;
                                for (way = 0; way < numb_ways; way++)
                                        {memory[memadr] = tags[set][way];
                                         memadr += 4;
                                         memory[memadr] = valid_bits[set][way];
                                         memadr += 4;
                                        for (word = 0; word < words_in_line;
                                                                        word++)
                                                {memory[memadr] =
                                                   lcache_line[set][way][word];
                                                 memadr += 4;
                                                }
                                        } break;
                        default:  # Reserved.
                                generate_fault(OPERATION.INVALID_OPERAND);
                                break;}
```

Faults:

STANDARD		Refer to Section 6.1.6, "Faults" on page 6-5.
TYPE.MISMATCH		Attempt to execute instruction while not in supervisor mode.

Example:

```
                                # g0 = 3, g1=0x10000000, g2=1
icctl g0,g1,g2                  # Load and lock 1 block of cache
                                # (one way) with
                                # location of code at starting
                                # 0x10000000.
```

Opcode:	icctl	65BH	REG

See Also: **sysctl**

Notes: This instruction is implemented on the 80303, 80960RM/RN, 80960RP/RD, 80960Hx, and 80960Jx processor families only, and may or may not be implemented on future i960 processors.

6.2.34 intctl

Table 6-40. intctl

Mnemonic: **intctl** Global Enable and Disable of Interrupts

Format:

intctl	*src1*	*dst*
	reg/lit	reg

Description: Globally enables, disables or returns the current status of interrupts depending on the value of *src1*. Returns the previous interrupt enable state (1 for enabled or 0 for disabled) in *dst*. When the state of the global interrupt enable is changed, the processor ensures that the new state is in full effect before the instruction completes. (This instruction is implemented by manipulating ICON.gie.)

***src1* Value**	**Operation**
0	Disables interrupts
1	Enables interrupts
2	Returns current interrupt enable status

Action:

```
if (PC.em != supervisor)
        generate_fault(TYPE.MISMATCH);
old_interrupt_enable = global_interrupt_enable;
switch(src1) {
        case 0:  # Disable. Set ICON.gie to one.
                 globally_disable_interrupts;
                 global_interrupt_enable = false;
                 order_wrt(subsequent_instructions);
                 break;
        case 1:  # Enable.  Clear ICON.gie to zero.
                 globally_enable_interrupts;
                 global_interrupt_enable = true;
                 order_wrt(subsequent_instructions);
                 break;
        case 2:  # Return status. Return ICON.gie
                 break;
        default: generate_fault(OPERATION.INVALID_OPERAND);
                 break;
        }
        if(old_interrupt_enable)s
                 dst = 1;
        else
                 dst = 0;
```

Faults:

STANDARD	Refer to Section 6.1.6, "Faults" on page 6-5.
TYPE.MISMATCH	Attempt to execute instruction while not in supervisor mode.

Example:

```
                        # ICON.gie = 0, interrupts enabled
intctl 0, g4            # Disable interrupts (ICON.gie = 1)
                        # g4 = 1
```

Opcode:

intctl	658H	REG

See Also: **intdis, inten**

Notes: This instruction is implemented on the 80303, 80960RM/RN, 80960RP/RD, 80960Hx, and 80960Jx processor families only, and may or may not be implemented on future i960 processors.

6.2.35 intdis

Table 6-41. intdis

Mnemonic:	**intdis**	Global Interrupt Disable	
Format:	**intdis**		
Description:	Globally disables interrupts and ensures that the change takes effect before the instruction completes. This operation is implemented by setting ICON.gie to one.		

Action:	if (PC.em != supervisor)
	generate_fault(TYPE.MISMATCH);
	# Implemented by setting ICON.gie to one.
	globally_disable_interrupts;
	interrupt_enable = false;
	order_wrt(subsequent_instructions);		

Faults:	STANDARD		Refer to Section 6.1.6, "Faults" on page 6-5.
	TYPE.MISMATCH		Attempt to execute instruction while not in supervisor mode.
Example:			`# ICON.gie = 0, interrupts enabled`
	`intdis`		`# Disable interrupts.`
			`# ICON.gie = 1`
Opcode:	**intdis**	5B4H	REG
See Also:	**intctl, inten**		
Notes:	This instruction is implemented on the 80303, 80960RM/RN, 80960RP/RD, 80960Hx, and 80960Jx processor families only, and may or may not be implemented on future i960 processors.		

6.2.36 inten

Table 6-42. inten

Mnemonic:	**inten**	global interrupt enable	
Format:	**inten**		
Description:	Globally enables interrupts and ensures that the change takes effect before the instruction completes. This operation is implemented by clearing ICON.gie to zero.		

Action:	if (PC.em != supervisor)
	generate_fault(TYPE.MISMATCH);
	# Implemented by clearing ICON.gie to zero.
	globally_enable_interrupts;
	interrupt_enable = true;
	order_wrt(subsequent_instructions);		

Faults:	STANDARD		Refer to Section 6.1.6, "Faults" on page 6-5.
	TYPE.MISMATCH		Attempt to execute instruction while not in supervisor mode.
Example:			`# ICON.gie = 1, interrupts disabled.`
	`inten`		`# Enable interrupts.`
			`# ICON.gie = 0`
Opcode:	**inten**	5B5H	REG
See Also:	**intctl, intdis**		
Notes:	This instruction is implemented on the 80303, 80960RM/RN, 80960RP/RD, 80960Hx, and 80960Jx processor families only, and may or may not be implemented on future i960 processors.		

6.2.37 LOAD

Table 6-43. LOAD (Sheet 1 of 3)

Mnemonic:	**ld**	Load	
	ldob	Load Ordinal Byte	
	ldos	Load Ordinal Short	
	ldib	Load Integer Byte	
	ldis	Load Integer Short	
	ldl	Load Long	
	ldt	Load Triple	
	ldq	Load Quad	
Format:	**ld***	*src,*	*dst*
		mem	reg

Description:

Copies byte or byte string from memory into a register or group of successive registers.

The *src* operand specifies the address of first byte to be loaded. The full range of addressing modes may be used in specifying *src*. Refer to Chapter 2, "Data Types and Memory Addressing Modes" for more information.

dst specifies a register or the first (lowest numbered) register of successive registers.

ldob and **ldib** load a byte and **ldos** and **ldis** load a half word and convert it to a full 32-bit word. Data being loaded is sign-extended during integer loads and zero-extended during ordinal loads.

ld, ldl, ldt and **ldq** instructions copy 4, 8, 12 and 16 bytes, respectively, from memory into successive registers.

For **ldl**, *dst* must specify an even numbered register (i.e., g0, g2...). For **ldt** and **ldq**, *dst* must specify a register number that is a multiple of four (i.e., g0, g4, g8, g12, r4, r8, r12). Results are unpredictable if registers are not aligned on the required boundary or if data extends beyond register g15 or r15 for **ldl**, **ldt** or **ldq**.

Action:

ld:

```
dst = read_memory(effective_address)[31:0];
if((effective_address[1:0] != 00₂ ) && unaligned _fault_enabled)
        generate_fault(OPERATION.UNALIGNED);
```

ldob:

```
dst[7:0] = read_memory(effective_address)[7:0];
dst[31:8] = 0x000000;
```

ldib:

```
dst[7:0] = read_memory(effective_address)[7:0];
if(dst[7] == 0)
        dst[31:8] = 0x000000;
else
        dst[31:8] = 0xFFFFFF;
```

ldis:

```
dst = read_memory(effective_address)[15:0];
                                # Order depends on endianism.
dst[31:16] = 0x0000;
if((effective_address[0] != 0₂) && unaligned_fault_enabled)
        generate_fault(OPERATION.UNALIGNED);
```

Table 6-43. LOAD (Sheet 2 of 3)

```
ldos:
dst = read_memory(effective_address)[15:0];
                                        # Order depends on endianism.
dst[31:16] = 0x0000;
if((effective_address[0] != 0₂) && unaligned_fault_enabled)
        generate_fault(OPERATION.UNALIGNED);
ldis:
dst[15:0] = read_memory(effective_address)[15:0];
                                        # Order depends on endianism.
if(dst[15] == 0₂
        dst[31:16] = 0x0000;
else
        dst[31:16] = 0xFFFF;
if((effective_address[0] != 0₂) && unaligned_fault_enabled)
        generate_fault(OPERATION.UNALIGNED);
ldl:
if((reg_number(dst) % 2) != 0)
        generate_fault(OPERATION.INVALID_OPERAND);
        # dst not modified.
else
{       dst = read_memory(effective_address)[31:0];
        dst_+_1 = read_memory(effective_address_+_4)[31:0];
        if((effective_address[2:0] != 000₂) && unaligned_fault_enabled)
                generate_fault(OPERATION.UNALIGNED);
}
ldt:
if((reg_number(dst) % 4) != 0)
        generate_fault(OPERATION.INVALID_OPERAND);
        # dst not modified.
else
{       dst = read_memory(effective_adddress)[31:0];
        dst_+_1 = read_memory(effective_adddress_+_4)[31:0];
        dst_+_2 = read_memory(effective_adddress_+_8)[31:0];
        if((effective_address[3:0] != 0000₂) && unaligned_fault_enabled)
                generate_fault(OPERATION.UNALIGNED);
}
ldq:
if((reg_number(dst) % 4) != 0)
        generate_fault(OPERATION.INVALID_OPERAND);
        # dst not modified.
else
{       dst = read_memory(effective_adddress)[31:0];
                                        # Order depends on endianism.
        dst_+_1 = read_memory(effective_adddress_+_4)[31:0];
        dst_+_2 = read_memory(effective_adddress_+_8)[31:0];
```

Table 6-43. LOAD (Sheet 3 of 3)

```
        dst_+_3 = read_memory(effective_adddress_+_12)[31:0];
        if((effective_address[3:0] != 0000₂) && unaligned_fault_enabled)
        generate_fault(OPERATION.UNALIGNED);
    }
```

Faults: STANDARD — Refer to Section 6.1.6, "Faults" on page 6-5.
OPERATION.UNALIGNED
OPERATION.INVALID_OPERAND

Example:
```
ldl 2450 (r3), r10      # r10, r11 = r3 + 2450 in
                        # memory
```

Opcode:

ld	90H	MEM
ldob	80H	MEM
ldos	88H	MEM
ldib	C0H	MEM
ldis	C8H	MEM
ldl	98H	MEM
ldt	A0H	MEM
ldq	B0H	MEM

See Also: **MOVE, STORE**

6.2.38 lda

Table 6-44. lda

Mnemonic:	**lda**	Load Address	
Format:	**lda**	*src*,	*dst*
		mem	reg
		efa	
Description:	Computes the effective address specified with *src* and stores it in *dst*. The *src* address is not checked for validity. Any addressing mode may be used to calculate *efa*. An important application of this instruction is to load a constant longer than 5 bits into a register. (To load a register with a constant of 5 bits or less, **mov** can be used with a literal as the *src* operand.)		
Action:	dst = effective_address;		
Faults:	STANDARD	Refer to Section 6.1.6, "Faults" on page 6-5.	
Example:	`lda 58 (g9), g1`	`# g1 = g9+58`	
	`lda 0x749, r8`	`# r8 = 0x749`	
Opcode:	**lda**	8CH	MEM

6.2.39 mark

Table 6-45. mark

Mnemonic:	**mark**	Mark	
Format:	**mark**		
Description:	Generates mark trace fault if mark trace mode is enabled. Mark trace mode is enabled if the PC register trace enable bit (bit 0) and the TC register mark trace mode bit (bit 7) are set. If mark trace mode is not enabled, **mark** behaves like a no-op. For more information on trace fault generation, refer to Chapter 10, "Tracing and Debugging".		
Action:	if(PC.te && TC.mk) generate_fault(TRACE.MARK)		
Faults:	STANDARD	Refer to Section 6.1.6, "Faults" on page 6-5.	
	TRACE.MARK	Trace fault is generated if PC.te=1 and TC.mk=1.	

Example:	`# Assume that the mark trace mode is enabled.`
	`ld xyz, r4`
	`addi r4, r5, r6`
	`mark`
	`# Mark trace event is generated at this point in the`
	`# instruction stream.`		

Opcode:	**mark**	66BH	REG
See Also:	**fmark, modpc, modtc**		

6.2.40 modac

Table 6-46. modac

Mnemonic:	**modac**	Modify AC		
Format:	**modac**	*mask*,	*src*,	*dst*
		reg/lit	reg/lit	reg
Description:	Reads and modifies the AC register. *src* contains the value to be placed in the AC register; *mask* specifies bits that may be changed. Only bits set in *mask* are modified. Once the AC register is changed, its initial state is copied into *dst.*			

Action:	temp = AC;
	AC = (src & mask) | (AC & ~mask);
	dst = temp;			

Faults:	STANDARD		Refer to Section 6.1.6, "Faults" on page 6-5.	
Example:	`modac g1, g9, g12`		`# AC = g9, masked by g1.` `# g12 = initial value of AC.`	
Opcode:	**modac**	645H	REG	
See Also:	**modpc, modtc**			
Side Effects:	Sets the condition code in the arithmetic controls.			

6.2.41 modi

Table 6-47. modi

Mnemonic:	**modi**	Modulo Integer		
Format:	**modi**	*src1*,	*src2*,	*dst*
		reg/lit	reg/lit	reg

Description: Divides *src2* by *src1*, where both are integers and stores the modulo remainder of the result in *dst*. If the result is nonzero, *dst* has the same sign as *src1*.

Action:

```
if(src1 == 0)
        {       dst = undefined_value;
                generate_fault(ARITHMETIC.ZERO_DIVIDE);
        }
dst = src2 - (src2/src1) * src1;
if((src2 *src1 < 0) && (dst != 0))
        dst = dst + src1;
```

Faults:	STANDARD	See Section 6.1.6, "Faults" on page 6-5.
	ARITHMETIC.ZERO_DIVIDE	The *src1* operand is zero.

Example:

```
modi r9, r2, r5          # r5 = modulo (r2/r9)
```

Opcode:	**modi**	749H	REG

See Also: **divi, divo, remi, remo**

Notes: **modi** generates the correct result (0) when computing -2^{31} mod -1, although the corresponding 32-bit division does overflow, it does not generate a fault.

6.2.42 modify

Table 6-48. modify

Mnemonic:	**modify**	Modify		
Format:	**modify**	*mask*,	*src*,	*src/dst*
		reg/lit	reg/lit	reg
Description:	Modifies selected bits in *src/dst* with bits from *src*. The *mask* operand selects the bits to be modified: only bits set in the *mask* operand are modified in *src/dst*.			
Action:	src_dst = (src & mask) | (src_dst & ~mask);			
Faults:	STANDARD		Refer to Section 6.1.6, "Faults" on page 6-5.	
Example:	`modify g8, g10, r4`		`# r4 = g10 masked by g8.`	
Opcode:	**modify**	650H	REG	
See Also:	**alterbit, extract**			

6.2.43 modpc

Table 6-49. modpc

Mnemonic:	**modpc**	Modify Process Controls		
Format:	**modpc**	*src,*	*mask,*	*src/dst*
		reg/lit	reg/lit	reg

Description:

Reads and modifies the PC register as specified with mask and *src/dst*. *src/dst* operand contains the value to be placed in the PC register; *mask* operand specifies bits that may be changed. Only bits set in the *mask* are modified. Once the PC register is changed, its initial value is copied into *src/dst*. The *src* operand is a dummy operand that should specify a literal or the same register as the mask operand.

The processor must be in supervisor mode to use this instruction with a non-zero mask value. If mask=0, this instruction can be used to read the process controls, without the processor being in supervisor mode.

If the action of this instruction lowers the processor priority, the processor checks the interrupt table for pending interrupts.

When process controls are changed, the processor recognizes the changes immediately except in one situation: if **modpc** is used to change the trace enable bit, the processor may not recognize the change before the next four non-branch instructions are executed. For more information see Section 3.6.3, "Process Controls Register – PC" on page 3-16.

Action:

```
if(mask != 0)
{       if(PC.em != supervisor)
                generate_fault(TYPE.MISMATCH);
        temp = PC;
        PC = (mask & src_dst) | (PC & ~mask);
        src_dst = temp;
        if(temp.priority > PC.priority)
                check_pending_interrupts;
else
        src_dst = PC;
```

Faults:

STANDARD	Refer to Section 6.1.6, "Faults" on page 6-5.
TYPE.MISMATCH	

Example:

```
modpc g9, g9, g8        # process controls = g8
                        # masked by g9.
```

Opcode:	**modpc**	655H	REG

See Also: **modac, modtc**

Notes:

Since **modpc** does not switch stacks, it should not be used to switch the mode of execution from supervisor to user (the supervisor stack can get corrupted in this case). The call and return mechanism should be used instead.

6.2.44 modtc

Table 6-50. modtc

Mnemonic:	**modtc**	Modify Trace Controls		
Format:	**modtc**	*mask*,	*src2*,	*dst*
		reg/lit	reg/lit	reg

Description: Reads and modifies TC register as specified with *mask* and *src2*. The *src2* operand contains the value to be placed in the TC register; *mask* operand specifies bits that may be changed. Only bits set in *mask* are modified. *mask* must not enable modification of reserved bits. Once the TC register is changed, its initial state is copied into *dst*.

The changed trace controls may take effect immediately or may be delayed. If delayed, the changed trace controls may not take effect until after the first non-branching instruction is fetched from memory or after four non-branching instructions are executed.

For more information on the trace controls, refer to Chapter 9, "Faults" and Chapter 10, "Tracing and Debugging".

Action:

```
mode_bits = 0x000000FE;
event_flags = 0X0F000000
temp = TC;
tempa = (event_flags & TC & mask) | (mode_bits & mask);
TC = (tempa & src2) | (TC & ~tempa);
dst = temp;
```

Faults: STANDARD — Refer to Section 6.1.6, "Faults" on page 6-5.

Example:

```
modtc g12, g10, g2      # trace controls = g10 masked
                        # by g12; previous trace
                        # controls stored in g2.
```

Opcode:	**modtc**	654H	REG

See Also: **modac, modpc**

6.2.45 MOVE

Table 6-51. MOVE (Sheet 1 of 2)

Mnemonic:	**mov**	Move	
	movl	Move Long	
	movt	Move Triple	
	movq	Move Quad	
Format:	**mov***	*src1*,	*dst*
		reg/lit	reg

Description: Copies the contents of one or more source registers (specified with *src*) to one or more destination registers (specified with *dst*).

For **movl**, **movt** and **movq**, *src1* and *dst* specify the first (lowest numbered) register of several successive registers. *src1* and *dst* registers must be even numbered (e.g., g0, g2, ... or r4, r6, ...) for **movl** and an integral multiple of four (e.g., g0, g4, ... or r4, r8, ...) for **movt** and **movq**.

The moved register values are unpredictable when: 1) the *src* and *dst* operands overlap; 2) registers are not properly aligned.

Action:

mov:

```
if(is_reg(src1))
        dst = src1;
else
{       dst[4:0] = src1;                #src1 is a 5-bit literal.
        dst[31:5] = 0;
}
```

movl:

```
if((reg_num(src1)%2 != 0) || (reg_num(dst)%2 != 0))
{       dst = undefined_value;
        dst_+_1 = undefined_value;
        generate_fault(OPERATION.INVALID_OPERAND);
}
else if(is_reg(src1))
{       dst = src1;
        dst_+_1 = src1_+_1;
}
else
{       dst[4:0] = src1;                #src1 is a 5-bit literal.
        dst[31:5] = 0;
        dst_+_1[31:0] = 0;
}
```

movt:

```
if((reg_num(src1)%4 != 0) || (reg_num(dst)%4 != 0))
{       dst = undefined_value;
        dst_+_1 = undefined_value;
```

Table 6-51. MOVE (Sheet 2 of 2)

```
                dst_+_2 = undefined_value;
                generate_fault(OPERATION.INVALID_OPERAND);
        }
        else if(is_reg(src1))
        {       dst = src1;
                dst_+_1 = src1_+_1;
                dst_+_2 = src1_+_2;
        }
        else
        {       dst[4:0] = src1;                #src1 is a 5-bit literal.
                dst[31:5] = 0;                  dst_+_1[31:0] = 0;
                dst_+_2[31:0] = 0;
        }
        movq:
        if((reg_num(src1)%4 != 0) || (reg_num(dst)%4 != 0))
        {       dst = undefined_value;
                dst_+_1 = undefined_value;
                dst_+_2 = undefined_value;
                dst_+_3 = undefined_value;
                generate_fault(OPERATION.INVALID_OPERAND);
        }
        else if(is_reg(src1))
        {       dst = src1;
                dst_+_1 = src1_+_1;
                dst_+_2 = src1_+_2;
                dst_+_3 = src1_+_3;
        }
        else
        {       dst[4:0] = src1;                #src1 is a 5 bit literal.
                dst[31:5] = 0;
                dst_+_1[31:0] = 0;
                dst_+_2[31:0] = 0;
                dst_+_3[31:0] = 0;
        }
```

Faults: STANDARD — Refer to Section 6.1.6, "Faults" on page 6-5.

Example:
```
movt g8, r4          # r4, r5, r6 = g8, g9, g10
```

Opcode:		
mov	5CCH	REG
movl	5DCH	REG
movt	5ECH	REG
movq	5FCH	REG

See Also: **LOAD, STORE, lda**

6.2.46 muli, mulo

Table 6-52. muli, mulo

Mnemonic:	**muli**	Multiply Integer		
	mulo	Multiply Ordinal		
Format:	**mul***	*src1*,	*src2*,	*dst*
		reg/lit	reg/lit	reg

Description: Multiplies the *src2* value by the *src1* value and stores the result in *dst*. The binary results from these two instructions are identical. The only difference is that muli can signal an integer overflow.

Action:

mulo:

```
dst = (src2 * src1)[31:0];
```

muli:

```
true_result = (src1 * src2);
dst = true_result[31:0];
if((true_result > (2**31) - 1) || (true_result < -2**31)) # Check for overflow
{       if(AC.om == 1)
                AC.of = 1;
        else
                generate_fault(ARITHMETIC.OVERFLOW);
}
```

Faults:	STANDARD	Refer to Section 6.1.6, "Faults" on page 6-5.
	ARITHMETIC.OVERFLOW	Result is too large for destination register (**muli** only). If a condition of overflow occurs, the least significant 32 bits of the result are stored in the destination register.

Example:

```
muli r3, r4, r9          # r9 = r4 * r3
```

Opcode:	**muli**	741H	REG
	mulo	701H	REG

See Also: **emul, ediv, divi, divo**

6.2.47 nand

Table 6-53. hand

Mnemonic:	**nand**	Nand		
Format:	**nand**	*src1*,	*src2*,	*dst*
		reg/lit	reg/lit	reg
Description:	Performs a bitwise NAND operation on *src2* and *src1* values and stores the result in *dst*.			
Action:	dst = ~src2 | ~src1;			
Faults:	STANDARD		Refer to Section 6.1.6, "Faults" on page 6-5.	
Example:	`nand g5, r3, r7`		`# r7 = r3 NAND g5`	
Opcode:	**nand**	58EH	REG	
See Also:	**and, andnot, nor, not, notand, notor, or, ornot, xnor, xor**			

6.2.48 nor

Table 6-54. nor

Mnemonic:	**nor**	Nor		
Format:	**nor**	*src1*,	*src2*,	*dst*
		reg/lit	reg/lit	reg
Description:	Performs a bitwise NOR operation on the *src2* and *src1* values and stores the result in *dst*.			
Action:	dst = ~src2 & ~src1;			
Faults:	STANDARD	Refer to Section 6.1.6, "Faults" on page 6-5.		
Example:	`nor g8, 28, r5`	`# r5 = 28 NOR g8`		
Opcode:	**nor**	588H	REG	
See Also:	**and, andnot, nand, not, notand, notor, or, ornot, xnor, xor**			

6.2.49 not, notand

Table 6-55. not, notand

Mnemonic:	**not**	Not	
	notand	Not And	
Format:	**not**	*src1*,	*dst*
		reg/lit	reg
	notand	*src1*,	*dst*
		reg/lit	reg
Description:	Performs a bitwise NOT (**not** instruction) or NOT AND (**notand** instruction) operation on the *src2* and *src1* values and stores the result in *dst*.		
Action:	**not:**		
	dst = ~src1;		
	notand:		
	dst = ~src2 & src1;		
Faults:	STANDARD	Refer to Section 6.1.6, "Faults" on page 6-5.	
Example:	`not g2, g4`	`# g4 = NOT g2`	
	`notand r5, r6, r7`	`# r7 = NOT r6 AND r5`	
Opcode:	**not**	58AH	REG
	notand	584H	REG
See Also:	**and, andnot, nand, nor, notor, or, ornot, xnor, xor**		

6.2.50 notbit

Table 6-56. notbit

Mnemonic:	**notbit**	Not Bit		
Format:	**notbit**	*bitpos*,	*src2*,	*dst*
		reg/lit	reg/lit	reg
Description:	Copies the *src2* value to *dst* with one bit toggled. The *bitpos* operand specifies the bit to be toggled.			
Action:	dst = src2 ^ 2**(src1%32);			
Faults:	STANDARD		Refer to Section 6.1.6, "Faults" on page 6-5.	
Example:	`notbit r3, r12, r7`		`# r7 = r12 with the bit`	
			`# specified in r3 toggled.`	
Opcode:	**notbit**	580H	REG	
See Also:	**alterbit, chkbit, clrbit, setbit**			

6.2.51 notor

Table 6-57. notor

Mnemonic:	**notor**	Not Or		
Format:	**notor**	*src1*,	*src2*,	*dst*
		reg/lit	reg/lit	reg
Description:	Performs a bitwise NOTOR operation on *src2* and *src1* values and stores result in *dst*.			
Action:	dst = ~src2 | src1;			
Faults:	STANDARD		Refer to Section 6.1.6, "Faults" on page 6-5.	
Example:	`notor g12, g3, g6`		`# g6 = NOT g3 OR g12`	
Opcode:	**notor**	58DH	REG	
See Also:	**and, andnot, nand, nor, not, notand, or, ornot, xnor, xor**			

6.2.52 or, ornot

Table 6-58. or, ornot

Mnemonic:	**or**	Or		
	ornot	Or Not		
Format:	**or**	*src1*,	*src2*,	*dst*
		reg/lit	reg/lit	reg
	ornot	*src1*,	*src2*,	*dst*
		reg/lit	reg/lit	reg
Description:	Performs a bitwise OR (**or** instruction) or ORNOT (**ornot** instruction) operation on the *src2* and *src1* values and stores the result in *dst*.			
Action:	**or:**			
	dst = src2 | src1;			
	ornot:			
	dst = src2 | ~src1;			
Faults:	STANDARD		Refer to Section 6.1.6, "Faults" on page 6-5.	
Example:	`or 14, g9, g3`		`# g3 = g9 OR 14`	
	`ornot r3, r8, r11`		`# r11 = r8 OR NOT r3`	
Opcode:	**or**	587H	REG	
	ornot	58BH	REG	
See Also:	**and, andnot, nand, nor, not, notand, notor, xnor, xor**			

6.2.53 remi, remo

Table 6-59. remi, remo

Mnemonic:	**remi**	Remainder Integer		
	remo	Remainder Ordinal		
Format:	**rem***	*src1*,	*src2*,	*dst*
		reg/lit	reg/lit	reg

Description: Divides *src2* by *src1* and stores the remainder in *dst*. The sign of the result (if nonzero) is the same as the sign of *src2*.

Action:

remi, remo:

```
if(src1 == 0)
        generate_fault(ARITHMETIC.ZERO_DIVIDE);
dst = src2 - (src2/src1)*src1;
```

Faults:	STANDARD	Refer to Section 6.1.6, "Faults" on page 6-5.
	ARITHMETIC.ZERO_DIVIDE	The *src1* operand is 0.

Example:

```
remo r4, r5, r6          # r6 = r5 rem r4
```

Opcode:	**remi**	748H	REG
	remo	708H	REG

See Also: **modi**

Notes: **remi** produces the correct result (0) even when computing -2^{31} **remi** -1, which would cause the corresponding division to overflow, although no fault is generated.

6.2.54 ret

Table 6-60. ret (Sheet 1 of 2)

Mnemonic: **ret** Return

Format: **ret**

Description: Returns program control to the calling procedure. The current stack frame (i.e., that of the called procedure) is deallocated and the FP is changed to point to the calling procedure's stack frame. Instruction execution is continued at the instruction pointed to by the RIP in the calling procedure's stack frame, which is the instruction immediately following the call instruction.

As shown in the action statement below, the return-status field and prereturn-trace flag determine the action that the processor takes on the return. These fields are contained in bits 0 through 3 of register r0 of the called procedure's local registers.

See Chapter 7, "Procedure Calls" for more on **ret**.

Action:

```
implicit_syncf();
if(pfp.p && PC.te && TC.p)
{       pfp.p = 0;
        generate_fault(TRACE.PRERETURN);
}
switch(return_status_field)
{
        case 000₂:      #local return
                get_FP_and_IP();
                break;
        case 001₂:      #fault return
                tempa = memory(FP-16);
                tempb = memory(FP-12);
                get_FP_and_IP();
                AC = tempb;
                if(execution_mode == supervisor)
                        PC = tempa;
                break;
        case 010₂       #supervisor return, trace on return disabled
                if(execution_mode != supervisor)
                        get_FP_and_IP();
                else
                {       PC.te = 0;
                        execution_mode = user;
                        get_FP_and_IP();
                }
                break;
        case 011₂:      # supervisor return, trace on return enabled
                if(execution_mode != supervisor)
                get_FP_and_IP();
```

Table 6-60. ret (Sheet 2 of 2)

```
                    else
                    {       PC.te = 1;
                            execution_mode = user;
                            get_FP_and_IP();
                    }
                    break;
                    case 100₂:      #reserved - unpredictable behavior
                            break;
                    case 101₂:      #reserved - unpredictable behavior
                            break;
                    case 110₂:      #reserved - unpredictable behavior
                            break;
                    case 111₂       #interrupt return
                            tempa = memory(FP-16);
                            tempb = memory(FP-12);
                            get_FP_and_IP();
                            AC = tempb;
                            if(execution_mode == supervisor)
                                    PC = tempa;
                                    check_pending_interrupts();
                                    break;
                    }
                    get_FP_and_IP()
                    {       FP =PFP;
                            free(current_register_set);
                            if(not_allocated(FP))
                                    retrieve_from_memory(FP);
                            IP = RIP;
                    }
```

Faults:	STANDARD		Refer to Section 6.1.6, "Faults" on page 6-5.
Example:	`ret`		`# Program control returns to` `# context of calling procedure.`
Opcode:	**ret**	0AH	CTRL
See Also:	**call, calls, callx**		

6.2.55 rotate

Table 6-61. rotate

Mnemonic:	**rotate**	Rotate		
Format:	**rotate**	*len*,	*src2*,	*dst*
		reg/lit	reg/lit	reg

Description: Copies *src2* to *dst* and rotates the bits in the resulting *dst* operand to the left (toward higher significance). Bits shifted off left end of word are inserted at right end of word. The *len* operand specifies number of bits that the *dst* operand is rotated.

This instruction can also be used to rotate bits to the right. The number of bits the word is to be rotated right should be subtracted from 32 and the result used as the *len* operand.

Action: *src2* is rotated by *len* mod 32. This value is stored in *dst*.

Faults: STANDARD — Refer to Section 6.1.6, "Faults" on page 6-5.

Example:
```
rotate 13, r8, r12      # r12 = r8 with bits rotated
                        # 13 bits to left.
```

Opcode:	**rotate**	59DH	REG

See Also: **SHIFT, eshro**

6.2.56 scanbit

Table 6-62. scanbit

Mnemonic:	**scanbit**	Scan For Bit	
Format:	**scanbit**	*src1*,	*dst*
		reg/lit	reg

Description: Searches *src1* for a set bit (1 bit). If a set bit is found, the bit number of the most significant set bit is stored in the *dst* and the condition code is set to 010_2. If *src* value is zero, all 1's are stored in *dst* and condition code is set to 000_2.

Action:

```
dst = 0xFFFFFFFF;
AC.cc = 000₂;
for(i = 31; i >= 0; i--)
{       if((src1 & 2**i) != 0)
{       dst = i;
        AC.cc = 010₂;
        break;
}
}
```

Faults: STANDARD — Refer to Section 6.1.6, "Faults" on page 6-5.

Example:

```
# assume g8 is nonzero
scanbit g8, g10          # g10 = bit number of most-
                         # significant set bit in g8;
                         # AC.cc = 010₂.
```

Opcode:	**scanbit**	641H	REG

See Also: **spanbit, setbit**

Side Effects: Sets the condition code in the arithmetic controls.

6.2.57 scanbyte

Table 6-63. scanbyte

Mnemonic:	**scanbyte**	Scan Byte Equal	
Format:	**scanbyte**	*src1*,	*src2*
		reg/lit	reg/lit
Description:	Performs byte-by-byte comparison of *src1* and *src2* and sets condition code to 010_2 if any two corresponding bytes are equal. If no corresponding bytes are equal, condition code is set to 000_2.		

Action:	if((src1 & 0x000000FF) == (src2 & 0x000000FF)
	|| (src1 & 0x0000FF00) == (src2 & 0x0000FF00)
	|| (src1 & 0x00FF0000) == (src2 & 0x00FF0000)
	|| (src1 & 0xFF000000) == (src2 & 0xFF000000))
	AC.cc = 010_2;
	else
	AC.cc = 000_2;		

Faults:	STANDARD	Refer to Section 6.1.6, "Faults" on page 6-5.	
Example:	`# Assume r9 = 0x11AB1100 # AC.cc = 010`$_2$ `scanbyte 0x00AB0011, r9`		
Opcode:	**scanbyte**	5ACH	REG
See Also:	**bswap**		
Side Effects:	Sets the condition code in the arithmetic controls.		

6.2.58 SEL<cc>

Table 6-64. SEL<cc>

Mnemonic:

selno	Select Based on Unordered
selg	Select Based on Greater
sele	Select Based on Equal
selge	Select Based on Greater or Equal
sell	Select Based on Less
selne	Select Based on Not Equal
selle	Select Based on Less or Equal
selo	Select Based on Ordered

Format:

sel*	*src1,*	*src2,*	*dst*
	reg/lit	reg/lit	reg

Description:

Selects either *src1* or *src2* to be stored in *dst* based on the condition code bits in the arithmetic controls. If for Unordered the condition code is 0, or if for the other cases the logical AND of the condition code and the mask part of the opcode is not zero, then the value of *src2* is stored in the destination. Else, the value of *src1* is stored in the destination.

Instruction	Mask	Condition
selno	000_2	Unordered
selg	001_2	Greater
sele	010_2	Equal
selge	011_2	Greater or equal
sell	100_2	Less
selne	101_2	Not equal
selle	110_2	Less or equal
selo	111_2	Ordered

Action:

```
if ((mask & AC.cc) || (mask == AC.cc))
        dst = src2;
else
        dst = src1;
```

Faults: STANDARD — Refer to Section 6.1.6, "Faults" on page 6-5.

Example:

```
                              # AC.cc = 010₂
sele g0,g1,g2                 # g2 = g1
                              # AC.cc = 001₂
sell g0,g1,g2                 # g2 = g0
```

Opcode:

selno	784H	REG
selg	794H	REG
sele	7A4H	REG
selge	7B4H	REG
sell	7C4H	REG
selne	7D4H	REG
selle	7E4H	REG
selo	7F4H	REG

See Also: **MOVE, TEST<cc>, cmpi, cmpo, SUB<cc>**

Notes: These core instructions are not implemented on i960 Cx, Kx and Sx processors.

6.2.59 setbit

Table 6-65. setbit

Mnemonic:	**setbit**	Set Bit		
Format:	**setbit**	*bitpos*,	*src*,	*dst*
		reg/lit	reg/lit	reg
Description:	Copies *src* value to *dst* with one bit set. *bitpos* specifies bit to be set.			
Action:	dst = src | (2**(bitpos%32));			
Faults:	STANDARD		Refer to Section 6.1.6, "Faults" on page 6-5.	
Example:	`setbit 15, r9, r1`		`# r1 = r9 with bit 15 set.`	
Opcode:	**setbit**	583H	REG	
See Also:	**alterbit, chkbit, clrbit, notbit**			

6.2.60 SHIFT

Table 6-66. SHIFT (Sheet 1 of 2)

Mnemonic:	**shlo**	Shift Left Ordinal		
	shro	Shift Right Ordinal		
	shli	Shift Left Integer		
	shri	Shift Right Integer		
	shrdi	Shift Right Dividing Integer		
Format:	**sh***	*len*,	*src*,	*dst*
		reg/lit	reg/lit	reg

Description:

Shifts *src* left or right by the number of bits indicated with the *len* operand and stores the result in *dst*. Bits shifted beyond register boundary are discarded. For values of *len* > 32, the processor interprets the value as 32.

shlo shifts zeros in from the least significant bit; **shro** shifts zeros in from the most significant bit. These instructions are equivalent to **mulo** and **divo** by the power of 2, respectively.

shli shifts zeros in from the least significant bit. An overflow fault is generated if the bits shifted out are not the same as the most significant bit (bit 31). If overflow occurs, *dst* equals *src* shifted left as much as possible without overflowing.

shri performs a conventional arithmetic shift-right operation by shifting in the most significant bit (bit 31). When this instruction is used to divide a negative integer operand by the power of 2, it produces an incorrect quotient (discarding the bits shifted out has the effect of rounding the result toward negative).

shrdi is provided for dividing integers by the power of 2. With this instruction, 1 is added to the result if the bits shifted out are non-zero and the *src* operand was negative, which produces the correct result for negative operands.

shli and **shrdi** are equivalent to **muli** and **divi** by the power of 2.

Action:

shlo:

```
if(src1 < 32)
        dst = src * (2**len);
else
        dst = 0;
```

shro:

```
if(src1 < 32)
        dst = src / (2**len);
else
        dst = 0;
```

shli:

```
if(len > 32)
        count = 32;
else
        count = src1;
temp = src;
while((temp[31] == temp[30]) && (count > 0))
{       temp = (temp * 2)[31:0];
```

Table 6-66. SHIFT (Sheet 2 of 2)

```
                count = count - 1;
}
dst = temp;
if(count > 0)
{       if(AC.om == 1)
        AC.of = 1;
else
        generate_fault(ARITHMETIC.OVERFLOW);
}
shri:
if(len > 32)
        count = 32;
else
        count = src1;
temp = src;
while(count > 0)
{       temp = (temp >> 1)[31:0];
        temp[31] = src[31];
        count = count - 1;
}
        dst = temp;
shrdi:
dst = src / (2**len);
```

Faults: STANDARD — Refer to Section 6.1.6, "Faults" on page 6-5.
ARITHMETIC.OVERFLOW — For **shli.**

Example:
```
shli 13, g4, r6          # g6 = g4 shifted left 13 bits.
```

Opcode:

shlo	59CH	REG
shro	598H	REG
shli	59EH	REG
shri	59BH	REG
shrdi	59AH	REG

See Also: **divi, muli, rotate, eshro**

Notes: **shli** and **shrdi** are identical to multiplications and divisions for all positive and negative values of *src2*. **shri** is the conventional arithmetic right shift that does not produce a correct quotient when *src2* is negative.

6.2.61 spanbit

Table 6-67. spanbit

Mnemonic:	**spanbit**	Span Over Bit	
Format:	**spanbit**	*src*,	*dst*
		reg/lit	reg

Description: Searches *src* value for the most significant clear bit (0 bit). If a most significant 0 bit is found, its bit number is stored in *dst* and condition code is set to 010_2. If *src* value is all 1's, all 1's are stored in *dst* and condition code is set to 000_2.

Action:

```
dst = 0xFFFFFFFF;
AC.cc = 000₂;
for(i = 31; i > = 0; i--)
{       if((src1 & 2**i) == 0)
            dst = i;
            AC.cc = 010₂;
            break;
    }
}
```

Faults: STANDARD — Refer to Section 6.1.6, "Faults" on page 6-5.

Example:

```
# Assume r2 is not 0xffffffff
spanbit r2, r9          # r9 = bit number of most-
                        # significant clear bit in r2;
                        # AC.cc = 010₂
```

Opcode:	**spanbit**	640H	REG

See Also: **scanbit**

Side Effects: Sets the condition code in the arithmetic controls.

6.2.62 STORE

Table 6-68. STORE (Sheet 1 of 4)

Mnemonic:	**st**	Store	
	stob	Store Ordinal Byte	
	stos	Store Ordinal Short	
	stib	Store Integer Byte	
	stis	Store Integer Short	
	stl	Store Long	
	stt	Store Triple	
	stq	Store Quad	
Format:	**st***	*src1*,	*dst*
		reg	mem

Description: Copies a byte or group of bytes from a register or group of registers to memory. *src* specifies a register or the first (lowest numbered) register of successive registers.

dst specifies the address of the memory location where the byte or first byte or a group of bytes is to be stored. The full range of addressing modes may be used in specifying *dst*. Refer to Section 2.3, "Memory Addressing Modes" on page 2-4 for a complete discussion.

stob and **stib** store a byte and **stos** and **stis** store a half word from the *src* register's low order bytes. Data for ordinal stores is truncated to fit the destination width. If the data for integer stores cannot be represented correctly in the destination width, an Arithmetic Integer Overflow fault is signaled.

st, **stl**, **stt** and **stq** copy 4, 8, 12 and 16 bytes, respectively, from successive registers to memory.

For **stl**, *src* must specify an even numbered register (e.g., g0, g2, ... or r0, r2, ...). For **stt** and **stq**, *src* must specify a register number that is a multiple of four (e.g., g0, g4, g8, ... or r0, r4, r8, ...).

Action:

st:

```
if (illegal_write_to_on_chip_RAM)
        generate_fault(TYPE.MISMATCH);
else if ((effective_address[1:0] != 00₂) && unaligned_fault_enabled)
        {store_to_memory(effective_address)[31:0]  = src1;
        generate_fault(OPERATION.UNALIGNED);}
else
        store_to_memory(effective_address)[31:0] = src1;
```

stob:

```
if (illegal_write_to_on_chip_RAM_or_MMR)
        generate_fault(TYPE.MISMATCH);
else
        store_to_memory(effective_address)[7:0] = src1[7:0];
```

Table 6-68. STORE (Sheet 2 of 4)

```
stib:
if (illegal_write_to_on_chip_RAM_or_MMR)
        generate_fault(TYPE.MISMATCH);
else if ((src1[31:8] != 0) && (src1[31:8] != 0xFFFFFF))
        {       store_to_memory(effective_address)[7:0] = src1[7:0];
                if (AC.om == 1)
                        AC.of = 1;
                else
                        generate_fault(ARITHMETIC.OVERFLOW);
        }
else
        store_to_memory(effective_address)[7:0] = src1[7:0];
end if;
stos:
if (illegal_write_to_on_chip_RAM_or_MMR)
        generate_fault(TYPE.MISMATCH);
else if ((effective_address[0] != 0₂) && unaligned_fault_enabled)
        {       store_to_memory(effective_address)[15:0] = src1[15:0];
                generate_fault(OPERATION.UNALIGNED);
        }
else
        store_to_memory(effective_address)[15:0] = src1[15:0];

stis:
if (illegal_write_to_on_chip_RAM_or_MMR)
        generate_fault(TYPE.MISMATCH);
else if ((effective_address[0] != 0₂) && unaligned_fault_enabled)
        {       store_to_memory(effective_address)[15:0] = src1[15:0];
                generate_fault(OPERATION.UNALIGNED);
        }
else if ((src1[31:16] != 0) && (src1[31:16] != 0xFFFF))
        {       store_to_memory(effective_address)[15:0] = src1[15:0];
                if (AC.om == 1)
                AC.of = 1;
        else
                generate_fault(ARITHMETIC.OVERFLOW);
        }
else
        store_to_memory(effective_address)[15:0] = src1[15:0];
```

Table 6-68. STORE (Sheet 3 of 4)

```
stl:
if (illegal_write_to_on_chip_RAM_or_MMR)
        generate_fault(TYPE.MISMATCH);
else if (reg_number(src1) % 2 != 0)
        generate_fault(OPERATION.INVALID_OPERAND);
else if ((effective_address[2:0] != 0002) && unaligned_fault_enabled)
        {       store_to_memory(effective_address)[31:0] = src1;
                store_to_memory(effective_address + 4)[31:0] = src1_+_1;
                generate_fault (OPERATION.UNALIGNED);
        }
else
        {       store_to_memory(effective_address)[31:0] = src1;
                store_to_memory(effective_address + 4)[31:0] = src1_+_1;
        }
stt:
if (illegal_write_to_on_chip_RAM_or_MMR)
        generate_fault(TYPE.MISMATCH);
else if (reg_number(src1) % 4 != 0)
        generate_fault(OPERATION.INVALID_OPERAND);
else if ((effective_address[3:0] != 0000) && unaligned_fault_enabled)
        {       store_to_memory(effective_address)[31:0] = src1;
                store_to_memory(effective_address + 4)[31:0] = src1_+_1;
                store_to_memory(effective_address + 8)[31:0] = src1_+_2;
                generate_fault (OPERATION.UNALIGNED);
        }
else
        {       store_to_memory(effective_address)[31:0] = src1;
                store_to_memory(effective_address + 4)[31:0] = src1_+_1;
                store_to_memory(effective_address + 8)[31:0] = src1_+_2;
        }
stq:
if (illegal_write_to_on_chip_RAM_or_MMR)
        generate_fault(TYPE.MISMATCH);
else if (reg_number(src1) % 4 != 0)
        generate_fault(OPERATION.INVALID_OPERAND);
else if ((effective_address[3:0] != 0000) && unaligned_fault_enabled)
        {       store_to_memory(effective_address)[31:0] = src1;
                store_to_memory(effective_address + 4)[31:0] = src1_+_1;
                store_to_memory(effective_address + 8)[31:0] = src1_+_2;
                store_to_memory(effective_address + 12)[31:0] = src1_+_3;
                generate_fault (OPERATION.UNALIGNED);
        }
```

Table 6-68. STORE (Sheet 4 of 4)

```
else
        {       store_to_memory(effective_address)[31:0] = src1;
                store_to_memory(effective_address + 4)[31:0] = src1_+_1;
                store_to_memory(effective_address + 8)[31:0] = src1_+_2;
                store_to_memory(effective_address + 12)[31:0] = src1_+_3;
        }
```

Faults:

STANDARD	Refer to Section 6.1.6, "Faults" on page 6-5.
ARITHMETIC.OVERFLOW	For **stib**, **stis**.

Example:

```
st g2, 1254 (g6)        # Word beginning at offset
                        # 1254 + (g6) = g2.
```

Opcode:

st	92H	MEM
stob	82H	MEM
stos	8AH	MEM
stib	C2H	MEM
stis	CAH	MEM
stl	9AH	MEM
stt	A2H	MEM
stq	B2H	MEM

See Also: **LOAD, MOVE**

Notes: illegal_write_to_on_chip_RAM is an implementation-dependent mechanism. The mapping of register bits to memory(*efa*) depends on the endianism of the memory region and is implementation-dependent.

6.2.63 subc

Table 6-69. subc

Mnemonic:	**subc**	Subtract Ordinal With Carry		
Format:	**subc**	*src1*,	*src2*,	*dst*
		reg/lit	reg/lit	reg

Description: Subtracts *src1* from *src2*, then subtracts the opposite of condition code bit 1 (used here as the carry bit) and stores the result in *dst*. If the ordinal subtraction results in a carry, condition code bit 1 is set to 1, otherwise it is set to 0.

This instruction can also be used for integer subtraction. Here, if integer subtraction results in an overflow, condition code bit 0 is set.

subc does not distinguish between ordinals and integers: it sets condition code bits 0 and 1 regardless of data type.

Action:

```
dst = (src2 - src1 -1 + AC.cc[1])[31:0];
AC.cc[2:0] = 000₂;
if((src2[31] == src1[31]) && (src2[31] != dst[31]))
        AC.cc[0] = 1;                    # Overflow bit.
AC.cc[1] = (src2 - src1 -1 + AC.cc[1])[32];          # Carry out.
```

Faults: STANDARD — Refer to Section 6.1.6, "Faults" on page 6-5.

Example:

```
subc g5, g6, g7
#g7 = g6 - g5 - not(condition code bit 1)
```

Opcode:	**subc**	5B2H	REG

See Also: **addc, addi, addo, subi, subo**

Side Effects: Sets the condition code in the arithmetic controls.

6.2.64 SUB<cc>

Table 6-70. SUB<cc>

Mnemonic:	**subono**	Subtract Ordinal if Unordered		
	subog	Subtract Ordinal if Greater		
	suboe	Subtract Ordinal if Equal		
	suboge	Subtract Ordinal if Greater or Equal		
	subol	Subtract Ordinal if Less		
	subone	Subtract Ordinal if Not Equal		
	subole	Subtract Ordinal if Less or Equal		
	suboo	Subtract Ordinal if Ordered		
	subino	Subtract Integer if Unordered		
	subig	Subtract Integer if Greater		
	subie	Subtract Integer if Equal		
	subige	Subtract Integer if Greater or Equal		
	subil	Subtract Integer if Less		
	subine	Subtract Integer if Not Equal		
	subile	Subtract Integer if Less or Equal		
	subio	Subtract Integer if Ordered		
Format:	**sub***	*src1,*	*src2,*	*dst*
		reg/lit	reg/lit	reg

Description: Subtracts *src1* from *src2* conditionally based on the condition code bits in the arithmetic controls.

If for Unordered the condition code is 0, or if for the other cases the logical AND of the condition code and the mask part of the opcode is not zero; then *src1* is subtracted from *src2* and the result stored in the destination

Instruction	Mask	Condition
subono, subino	000_2	Unordered
subog, subig	001_2	Greater
suboe, subie	010_2	Equal
suboge, subige	011_2	Greater or equal
subol, subil	100_2	Less
subone, subine	101_2	Not equal
subole, subile	110_2	Less or equal
suboo, subio	111_2	Ordered

Action:

SUBO<cc>:

```
if ((mask & AC.cc) || (mask == AC.cc))
dst = (src2 - src1)[31:0];
```

SUBI<cc>:

```
if ((mask & AC.cc) || (mask == AC.cc))
{
```

```
{       true_result = (src2 - src1);
        dst = true_result[31:0];
}
if((true_result > (2**31) - 1) || (true_result < -2**31))
                                                # Check for overflow
        {       if (AC.om == 1)
                AC.of  = 1;
        else
                generate_fault (ARITHMETIC.OVERFLOW);
        }
}
```

Faults: STANDARD — Refer to Section 6.1.6, "Faults" on page 6-5.

ARITHMETIC.OVERFLOW — For the **SUBI<cc>** class.

Example:

```
                                # AC.cc = 010₂
suboge g0,g1,g2                 # g2 = g1 - g0
                                # AC.cc = 001₂
subile g0,g1,g2                 # g2 not modified
```

Opcode:

subono	782H	REG
subog	792H	REG
suboe	7A2H	REG
suboge	7B2H	REG
subol	7C2H	REG
subone	7D2H	REG
subole	7E2H	REG
suboo	7F2H	REG
subino	783H	REG
subig	793H	REG
subie	7A3H	REG
subige	7B3H	REG
subil	7C3H	REG
subine	7D3H	REG
subile	7E3H	REG
subio	7F3H	REG

See Also: **subc, subi, subo, SEL<cc>, TEST<cc>**

Notes: These core instructions are not implemented on 80960Cx, Kx and Sx processors.

6.2.65 subi, subo

Table 6-71. subi, subo

Mnemonic:	**subi**	Subtract Integer		
	subo	Subtract Ordinal		
Format:	**sub***	*src1*,	*src2*,	*dst*
		reg/lit	reg/lit	reg

Description: Subtracts *src1* from *src2* and stores the result in *dst*. The binary results from these two instructions are identical. The only difference is that **subi** can signal an integer overflow.

Action:

subo:

```
dst = (src2 - src1)[31:0];
```

subi:

```
true_result = (src2 - src1);
dst = true_result[31:0];
if((true_result > (2**31) - 1) || (true_result < -2**31))     # Check for overflow
{       if(AC.om == 1)
        AC.of = 1;
else
        generate_fault(ARITHMETIC.OVERFLOW);
}
```

Faults:	STANDARD	Refer to Section 6.1.6, "Faults" on page 6-5.
	ARITHMETIC.OVERFLOW	For **subi.**

Example:

```
subi g6, g9, g12          # g12 = g9 - g6
```

Opcode:	**subi**	593H	REG
	subo	592H	REG

See Also: **addi, addo, subc, addc**

6.2.66 syncf

Table 6-72. syncf

Mnemonic: **syncf** Synchronize Faults

Format: **syncf**

Description: Waits for all faults to be generated that are associated with any prior uncompleted instructions.

Action:

```
if(AC.nif == 1)
        break;
else
        wait_until_all_previous_instructions_in_flow_have_completed();
        #       This also means that all of the faults on these instructions have
        #       been reported.
```

Faults: STANDARD Refer to Section 6.1.6, "Faults" on page 6-5.

Example:

```
ld xyz, g6
addi r6, r8, r8
syncf
and g6, 0xFFFF, g8
# The syncf instruction ensures that any faults
# that may occur during the execution of the
# ld and addi instructions occur before the
# and instruction is executed.
```

Opcode: **syncf** 66FH REG

See Also: **mark, fmark**

6.2.67 sysctl

Table 6-73. sysctl

Mnemonic:	**sysctl**	System Control		
Format:	**sysctl**	*src1,*	*src2,*	*src/dst*
		reg/lit	reg/lit	reg

Description: Performs system management and control operations including requesting software interrupts, invalidating the instruction cache, configuring the instruction cache, processor reinitialization, modifying memory-mapped registers, and acquiring breakpoint resource information.

Processor control function specified by the message field of *src1* is executed. The type field of *src1* is interpreted depending upon the command. Remaining *src1* bits are reserved. The *src2* and *src3* operands are also interpreted depending upon the command.

Figure 6-7. Src1 Operand Interpretation

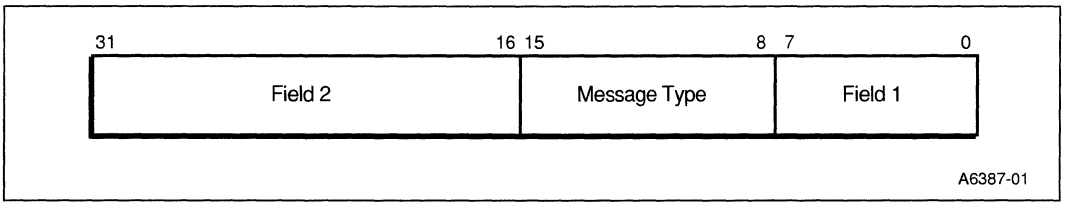


Table 6-74. sysctl Field Definitions

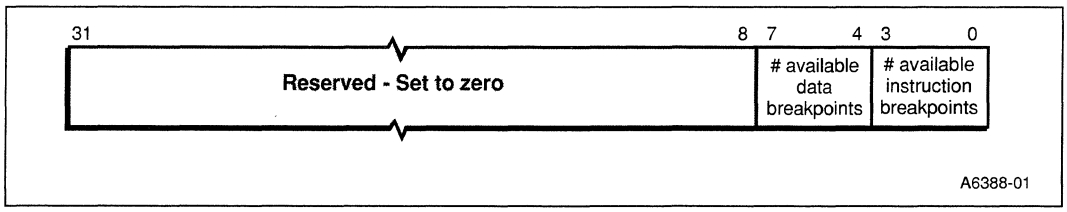
Message	Src1			Src2	Src/Dst
	Type	Field 1	Field 2	Field 3	Field 4
Request Interrupt	0x0	Vector Number	N/U	N/U	N/U
Invalidate Cache	0x1	N/U	N/U	N/U	N/U
Configure Instruction Cache	0x2	Cache Mode Configuration (Table 6-75)	N/U	Cache load address	N/U
Reinitialize	0x3	N/U	N/U	Starting IP	PRCB Pointer
Modify Memory-Mapped Control Register (MMR)	0x5	N/U	Lower 2 bytes of MMR address	Value to write	Mask
Breakpoint Resource Request	0x6	N/U	N/U	N/U	Breakpoint info (Figure 6-8)

NOTE: Sources and fields that are not used (designated N/U) are ignored.

Table 6-75. Cache Mode Configuration

Mode Field	Mode Description	80303
000_2	Normal cache enabled	16 Kbyte
$XX1_2$	Full cache disabled	16 Kbyte
100_2 or 110_2	Load and lock one way of the cache	4 Kbyte

Figure 6-8. *src/dst* Interpretation for Breakpoint Resource Request



Action:

```
if (PC.em != supervisor)
        generate_fault(TYPE.MISMATCH);
order_wrt(previous_operations);
OPtype = (src1 & 0xff00) >> 8;
switch (OPtype) {
case 0:         # Signal Software Interrupt
        vector_to_post = 0xff & src1;
        priority_to_post = vector_to_post >> 3;
        pend_ints_addr = interrupt_table_base + 4 + priority_to_post;
        pend_priority = memory_read(interrupt_table_base,atomic_lock);
        # Priority zero just recans Interrupt Table
        if (priority_to_post != 0)
                {pend_ints = memory_read(pend_ints_addr, non-cacheable)
                pend_ints[7 & vector] = 1;
                 pend_priority[priority_to_post] = 1;
                memory_write(pend_ints_addr, pend_ints); }
        memory_write(interrupt_table_base,pend_priority,atomic_unlock);
        # Update internal software priority with highest priority interrupt
        # from newly adjusted Pending Priorities word. The current internal
        # software priority is always replaced by the new, computed one. (If
        # there is no bit set in pending_priorities word for the current
        # internal one, then it is discarded by this action.)
        if (pend_priority == 0)
                SW_Int_Priority = 0;
        else {msb_set = scan_bit(pend_priority);
                SW_Int_Priority = msb_set;  }
        # Make sure change to internal software priority takes full effect
        # before next instruction.
        order_wrt(subsequent_operations);
                        break;
case 1:  # Global Invalidate Instruction Cache
        invalidate_instruction_cache( );
        unlock_instruction_cache( );
```

```
                break;
case 2:         # Configure Instruction-Cache
                mode = src1 & 0xff;
                if (mode & 1) disable_instruction_cache;
                else switch (mode) {
                        case 0:         enable_instruction_cache; break;
                        case 4,6:       # Load & Lock code into I-Cache
                                        # All contiguous blocks are locked.
                                        # Note:  block = way on 80303.
                                        # src2 has starting address of code to lock.
                                        # src2 is aligned to a quad word
                                        # boundary
                                        aligned_addr = src2 & 0xfffffff0;
                                        invalidate(I-cache); unlock(I-cache);
                                        for (j = 0; j < number_of_blocks_that_lock; j++)
                                        {way = block_associated_with_block(j);
                                        start = src2 + j*block_size;
                                        end = start + block_size;
                                        for (i = start; i < end; i=i+4)
                                        {
                                                set = set_associated_with(i);
                                                word = word_associated_with(i);
                                                Icache_line[set][way][word] =
                                                        memory[i];
                                                update_tag_n_valid_bits(set,way,word)
                                                lock_icache(set,way,word);
                                        } } break;
                        default:                generate_operation_invalid_operand_fault;

                        } break;
case 3:         # Software Re-init
                disable(I_cache); invalidate(I_cache);
                disable(D_cache); invalidate(D_cache);
                Process_PRCB(dst);  # dst has ptr to new PRCB
                IP = src2;
                break;
case 5:         # Modify One Memory-Mapped Control Register (MMR)
                # src1[31:16] has lower 2 bytes of MMR address
                # src2 has value to write; dst has mask.
                # After operation, dst has old value of MMR
```

```
                addr = (0xff00 << 16) | (src1 >> 16);
                temp = memory[addr];
                memory[addr] = (src2 & dst) | (temp & ~dst);
                dst = temp;
                break;
        case 6: # Breakpoint Resource Request
                acquire_available_instr_breakpoints( );
                dst[3:0] = number_of_available_instr_breakpoints;
                acquire_available_data_breakpoints( );
                dst[7:4] = number_of_available_data_breakpoints;
                dst[31:8] = 0;
                break;
        default: # Reserved, fault occurs
                generate_fault(OPERATION.INVALID_OPERAND);
                break;
        }
        order_wrt(subsequent_operations);
```

Faults: STANDARD Refer to Section 6.1.6, "Faults" on page 6-5.

Example:

```
ldconst 0x100,r6                # Set up message.
sysctl r6,r7,r8                 # Invalidate I-cache.
                                # r7, r8 are not used.
ldconst 0x204, g0               # Set up message type and
                                # cache configuration mode.
                                # Lock half cache.
ldconst 0x20000000,g2           # Starting address of code.
sysctl g0,g2,g2                 # Execute Load and Lock.
```

Opcode: **sysctl** 659H REG

See Also: **Sdcctl, icctl**

Notes: This instruction is implemented on 80303, 80960RP/RD, 80960Hx, 80960Jx and 80960Cx processors, and may or may not be implemented on future i960 processors.

6.2.68 TEST<cc>

Table 6-76. TEST<cc>

Mnemonic:	**teste**	Test For Equal
	testne	Test For Not Equal
	testl	Test For Less
	testle	Test For Less Or Equal
	testg	Test For Greater
	testge	Test For Greater Or Equal
	testo	Test For Ordered
	testno	Test For Not Ordered
Format:	**test***	dst:src1
		reg

Description: Stores a true (01H) in *dst* if the logical AND of the condition code and opcode mask part is not zero. Otherwise, the instruction stores a false (00H) in *dst*. For **testno** (Unordered), a true is stored if the condition code is 000_2, otherwise a false is stored.

The following table shows the condition-code mask for each instruction. The mask is in bits 0-2 of the opcode.

Instruction	Mask	Condition
testno	000_2	Unordered
testg	001_2	Greater
teste	010_2	Equal
testge	011_2	Greater or equal
testl	100_2	Less
testne	101_2	Not equal
testle	110_2	Less or equal
testo	111_2	Ordered

Action:

For all **TEST<cc>** except **testno**:

```
if((mask & AC.cc) != 000₂)
        src1 = 1;          #true value
else
        src1 = 0;          #false value
```

testno:

```
if(AC.cc == 000₂)
        src1 = 1;          #true value
else
        src1 = 0;          #false value
```

Faults: STANDARD — Refer to Section 6.1.6, "Faults" on page 6-5.

Example:

```
# Assume AC.cc = 100₂
testl g9                   # g9 = 0x00000001
xor g1, g7, g4             # g4 = g7 XOR g1
```

Opcode:	**teste** 22H	COBR
	testne 25H	COBR
	testl 24H	COBR
	testle	26H
	testg	21H
	testge	23H
	testo	27H
	testno	20H

See Also: **cmpi, cmpdeci, cmpinci**

See Also: **cmpi, cmpdeci, cmpinci**

6.2.69 xnor, xor

Table 6-77. xnor, xor

Mnemonic:	**xnor**	Exclusive Nor		
	xor	Exclusive Or		
Format:	**xnor**	*src1*,	*src2*,	dst
		reg/lit	reg/lit	reg
	xor	*src1*,	*src2*,	dst
		reg/lit	reg/lit	reg
Description:	Performs a bitwise XNOR (xnor instruction) or XOR (xor instruction) operation on the src2 and src1 values and stores the result in dst.			
Action:	**xnor:**			
	dst = ~(src2 | src1) | (src2 & src1);			
	xor:			
	dst = (src2 | src1) & ~(src2 & src1);			
Faults:	STANDARD	Refer to Section 6.1.6, "Faults" on page 6-5.		
Example:	`xnor r3, r9, r12`		`# r12 = r9 XNOR r3`	
	`xor g1, g7, g4`		`# g4 = g7 XOR g1`	
Opcode:	**xnor**	589H	REG	
	xor	586H	REG	
See Also:	**and, andnot, nand, nor, not, notand, notor, or, ornot**			

intel®

Procedure Calls 7

This chapter describes mechanisms for making procedure calls, which include branch-and-link instructions, built-in call and return mechanism, call instructions (**call**, **callx**, **calls**), return instruction (**ret**) and call actions caused by interrupts and faults.

The Intel® i960® architecture supports two methods for making procedure calls:

- A RISC-style branch-and-link: fast call best suited for calling procedures not calling other procedures.
- An integrated call and return mechanism: a more versatile method for making procedure calls, providing a highly efficient means for managing a large number of registers and the program stack.

On a branch-and-link (**bal**, **balx**), the processor branches and saves a return IP in a register. The called procedure uses the same set of registers and same stack as the calling procedure. On a call (**call**, **callx**, **calls**) or when an interrupt or fault occurs, the processor also branches to a target instruction and saves a return IP. Additionally, the processor saves local registers and allocates a new set of local registers and a new stack for the called procedure. The saved context is restored when the return instruction (**ret**) executes.

In many RISC architectures, a branch-and-link instruction is used as the base instruction for coding a procedure call. The user program handles register and stack management for the call. Since i960 architecture provides a fully integrated call and return mechanism, branch-and-link coding calls are not necessary. Additionally, the integrated call is much faster than typical RISC-coded calls.

The branch-and-link instruction in the i960 processor family, therefore, is used primarily for calling leaf procedures. Leaf procedures call no other procedures; they reside at the "leaves" of the call tree.

In the i960 architecture the integrated call and return mechanism is used in two ways:

- explicit calls to procedures in a user's program
- implicit calls to interrupt and fault handlers

The remainder of this chapter explains the generalized call mechanism used for explicit and implicit calls and call and return instructions.

The processor performs two call actions:

local When a local call is made, execution mode remains unchanged and the stack frame for called procedure is placed on *local stack*. Local stack refers to stack of calling procedure.

supervisor When a supervisor call is made from user mode, execution mode is switched to supervisor and the stack frame for the called procedure is placed on the *supervisor stack*.

When a supervisor call is issued from supervisor mode, the call degenerates into a local call (i.e., no mode nor stack switch).

Explicit procedure calls can be made using several instructions. Local call instructions **call** and **callx** perform a local call action. With **call** and **callx**, the called procedure IP is included as an operand in the instruction.

A system call is made with **calls**. This instruction is similar to **call** and **callx**, except the processor obtains the called procedure IP from the *system procedure table*. A system call, when executed, is directed to perform either local or supervisor call action. These calls are referred to as *system-local* and *system-supervisor* calls, respectively. A system-supervisor call is also referred to as *supervisor call*.

7.1 Call and Return Mechanism

At any point in a program, the i960 processor has access to the global registers, a local register set and the procedure stack. A subset of the stack allocated to the procedure is called the stack frame.

- When a call executes, a new stack frame is allocated for the called procedure. The processor also saves the current local register set, freeing these registers for use by the newly called procedure. In this way, every procedure has a unique stack and a unique set of local registers.
- When a return executes, the current local register set and current stack frame are deallocated. The previous local register set and previous stack frame are restored.

7.1.1 Local Registers and the Procedure Stack

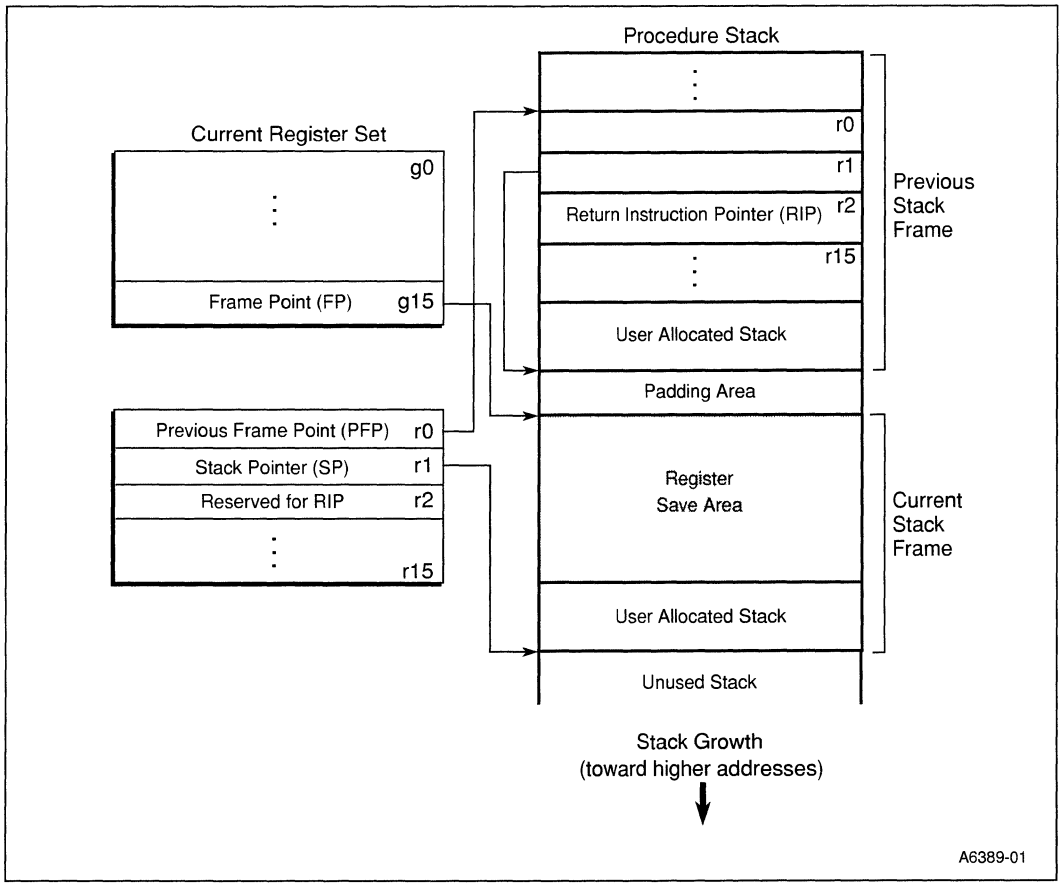
The processor automatically allocates a set of 16 local registers for each procedure. Since local registers are on-chip, they provide fast access storage for local variables. Of the 16 local registers, 13 are available for general use; r0, r1 and r2 are reserved for linkage information to tie procedures together.

The processor does not always clear or initialize the set of local registers assigned to a new procedure. Therefore, initial register contents are unpredictable. Also, because the processor does not initialize the local register save area in the newly created stack frame for the procedure, its contents are equally unpredictable.

The procedure stack can be located anywhere in the address space and grows from low addresses to high addresses. It consists of contiguous frames, one frame for each active procedure. Local registers for a procedure are assigned a save area in each stack frame (Figure 7-1). The procedure stack, available to the user, begins after this save area.

To increase procedure call speed, the architecture allows an implementation to cache the saved local register sets on-chip. Thus, when a procedure call is made, the contents of the current set of local registers often do not have to be written out to the save area in the stack frame in memory. Refer to *Section 7.1.4, "Caching Local Register Sets" on page 7-7* and Section 7.1.4.1, "Reserving Local Register Sets for High Priority Interrupts" on page 7-8 for more about local registers and procedure stack interrelations.

Figure 7-1. Procedure Stack Structure and Local Registers



7.1.2 Local Register and Stack Management

Global register g15 (FP) and local registers r0 (PFP), r1 (SP) and r2 (RIP) contain information to link procedures together and link local registers to the procedure stack (Figure 7-1). The following subsections describe this linkage information.

7.1.2.1 Frame Pointer

The frame pointer is the current stack frame's first byte address. It is stored in global register g15, the frame pointer (FP) register. The FP register is always reserved for the frame pointer; do not use g15 for general storage.

Stack frame alignment is defined for each implementation of the i960 processor family, according to an SALIGN parameter. In the 80303 I/O processor, stacks are aligned on 16-byte boundaries (Figure 7-1). When the processor needs to create a new frame on a procedure call, it adds a padding area to the stack so that the new frame starts on a 16-byte boundary.

7.1.2.2 Stack Pointer

The stack pointer is the byte-aligned address of the stack frame's next unused byte. The stack pointer value is stored in local register r1, the stack pointer (SP) register. The procedure stack grows upward (i.e., toward higher addresses). When a stack frame is created, the processor automatically adds 64 to the frame pointer value and stores the result in the SP register. This action creates the register save area in the stack frame for the local registers.

The program must modify the SP register value when data is stored or removed from the stack. The i960 architecture does not provide an explicit push or pop instruction to perform this action. This is typically done by adding the size of all pushes to the stack in one operation.

7.1.2.3 Considerations When Pushing Data onto the Stack

Care should be taken in writing to stack in the presence of unforeseen faults and interrupts. In the general case, to ensure that the data written to the stack is not corrupted by a fault or interrupt record, the SP should be incremented first to allocate the space, and then the data should be written to the allocated space:

```
mov     sp,r4
addo    24,sp,sp
st      data,(r4)
        ...
st      data,20(r4)
```

7.1.2.4 Considerations When Popping Data off the Stack

For reasons similar to those discussed in the previous section, care should be taken in reading the stack in the presence of unforeseen faults and interrupts. In the general case, to ensure that data about to be popped off the stack is not corrupted by a fault or interrupt record, the data should be read first and then the sp should be decremented:

```
subo    24,sp,r4
ld      20(r4),rn
        ...
ld      (r4),rn
mov     r4,sp
```

7.1.2.5 Previous Frame Pointer

The previous frame pointer is the previous stack frame's first byte address. This address's upper 28 bits are stored in local register r0, the previous frame pointer (PFP) register. The four least-significant bits of the PFP are used to store the return type field. See Figure 7-5 and Table 7-2 for more information on the PFP and the return-type field.

7.1.2.6 Return Type Field

PFP register bits 0 through 3 contain return type information for the calling procedure. When a procedure call is made — either explicit or implicit — the processor records the call type in the return type field. The processor then uses this information to select the proper return mechanism when returning to the calling procedure. The use of this information is described in Section 7.8, "Returns" on page 7-19.

7.1.2.7 Return Instruction Pointer

The actual RIP register (r2) is reserved by the processor to support the call and return mechanism and must not be used by software; the actual value of RIP is unpredictable at all times. For example, an implicit procedure call (fault or interrupt) can occur at any time and modify the RIP. An OPERATION.INVALID_OPERAND fault is generated when attempting to write the RIP.

The image of the RIP register in the stack frame is used by the processor to determine that frame's return instruction address. When a call is made, the processor saves the address of the instruction after the call in the image of the RIP register in the calling frame.

7.1.3 Call and Return Action

To clarify how procedures are linked and how the local registers and stack are managed, the following sections describe a general call and return operation and the operations performed with the FP, SP, PFP and RIP registers.

The events for call and return operations are given in a logical order of operation. The 80303 I/O processor can execute independent operations in parallel; therefore, many of these events execute simultaneously. For example, to improve performance, the processor often begins prefetching of the target instruction for the call or return before the operation is complete.

7.1.3.1 Call Operation

When a **call, calls** or **callx** instruction is executed or an implicit call is triggered:

1. The processor stores the instruction pointer for the instruction following the call in the current stack's RIP register (r2).
2. The current local registers — including the PFP, SP and RIP registers — are saved, freeing these for use by the called procedure. The local registers are saved in the on-chip local register cache if space is available.
3. The frame pointer (g15) for the calling procedure is stored in the current stack's PFP register (r0). The return type field in the PFP register is set according to the call type which is performed. See Section 7.8, "Returns" on page 7-19.
4. For a local or system-local call, a new stack frame is allocated by using the old stack pointer value saved in step 2. This value is first rounded to the next 16-byte boundary to create a new frame pointer, then stored in the FP register. Next, 64 bytes are added to create the new frame's register save area. This value is stored in the SP register.

   For an interrupt call from user mode, the current interrupt stack pointer value is used instead of the value saved in step 2.

   For a system-supervisor call from user mode, the current Supervisor Stack Pointer (SSP) value is used instead of the value saved in step 2.
5. The instruction pointer is loaded with the address of the first instruction in the called procedure. The processor gets the new instruction pointer from the **call**, the system procedure table, the interrupt table or the fault table, depending on the type of call executed.

Upon completion of these steps, the processor begins executing the called procedure. Sometime before a return or nested call, the local register set is bound to the allocated stack frame.

7.1.3.2 Return Operation

A return from any call type — explicit or implicit — is always initiated with a return (**ret**) instruction. On a return, the processor performs these operations:

1. The current stack frame and local registers are deallocated by loading the FP register with the value of the PFP register.
2. The local registers for the return target procedure are retrieved. The registers are usually read from the local register cache; however, in some cases, these registers have been flushed from register cache to memory and must be read directly from the save area in the stack frame.
3. The processor sets the instruction pointer to the value of the RIP register.

Upon completion of these steps, the processor executes the instruction to which it returns. The frames created before the **ret** instruction was executed is overwritten by later implicit or explicit call operations.

7.1.4 Caching Local Register Sets

Actual implementations of the i960 architecture may cache some number of local register sets within the processor to improve performance. Local registers are typically saved and restored from the local register cache when calls and returns are executed. Other overhead associated with a call or return is performed in parallel with this data movement.

When the number of nested procedures exceeds local register cache size, local register sets must at times be saved to (and restored from) their associated save areas in the procedure stack. Because these operations require access to external memory, this local cache miss affects call and return performance.

When a call is made and no frames are available in the register cache, a register set in the cache must be saved to external memory to make room for the current set of local registers in the cache. See Section 4.2, "Local Register Cache" on page 4-2. This action is referred to as a frame spill. The oldest set of local registers stored in the cache is spilled to the associated local register save area in the procedure stack. Figure 7-2 illustrates a call operation with and without a frame spill.

Similarly, when a return is made and the local register set for the target procedure is not available in the cache, these local registers must be retrieved from the procedure stack in memory. This operation is referred to as a frame fill. Figure 7-3 illustrates return operations with and without frame fills.

The **flushreg** instruction, described in Section 6.2.30, "flushreg" on page 6-44, writes all local register sets (except the current one) to their associated stack frames in memory. The register cache is then invalidated, meaning that all flushed register sets are restored from their save areas in memory.

For most programs, the existence of the multiple local register sets and their saving/restoring in the stack frames should be transparent. However, there are some special cases:

- A store to the register save area in memory does not necessarily update a local register set, unless user software executes **flushreg** first.
- Reading from the register save area in memory does not necessarily return the current value of a local register set, unless user software executes **flushreg** first.
- There is no mechanism, including **flushreg**, to access the current local register set with a read or write to memory.
- **flushreg** must be executed sometime before returning from the current frame if the current procedure modifies the PFP in register r0, or else the behavior of the **ret** instruction is not predictable.
- The values of the local registers r2 to r15 in a new frame are undefined.

flushreg is commonly used in debuggers or fault handlers to gain access to all saved local registers. In this way, call history may be traced back through nested procedures.

7.1.4.1 Reserving Local Register Sets for High Priority Interrupts

To decrease interrupt latency for high priority interrupts, software can limit the number of frames available to all remaining code. This includes code that is either in the executing state (non-interrupted) or code that is in the interrupted state but has a process priority less than 28. For the purposes of discussion here, this remaining code is referred to as *non-critical code*. Specifying a limit for non-critical code ensures that some number of free frames are available to high-priority interrupt service routines. Software can specify the limit for non-critical code by writing bits 10 through 8 of the register cache configuration word in the PRCB (Table 11-8, "Process Control Block Configuration Words" on page 11-16). The value indicates how many frames within the register cache may be used by non-critical code before a frame needs to be flushed to external memory. The programmed limit is used only when a frame is pushed, which occurs only for an implicit or explicit call.

Allowed values of the programmed limit range from 0 to 7. Setting the value to 0 reserves no frames for high-priority interrupts. Setting the value to 7 causes the register cache to become disabled for non-critical code. See Section 11.4.2, "Process Control Block – PRCB" on page 11-15.

Figure 7-2. Frame Spill

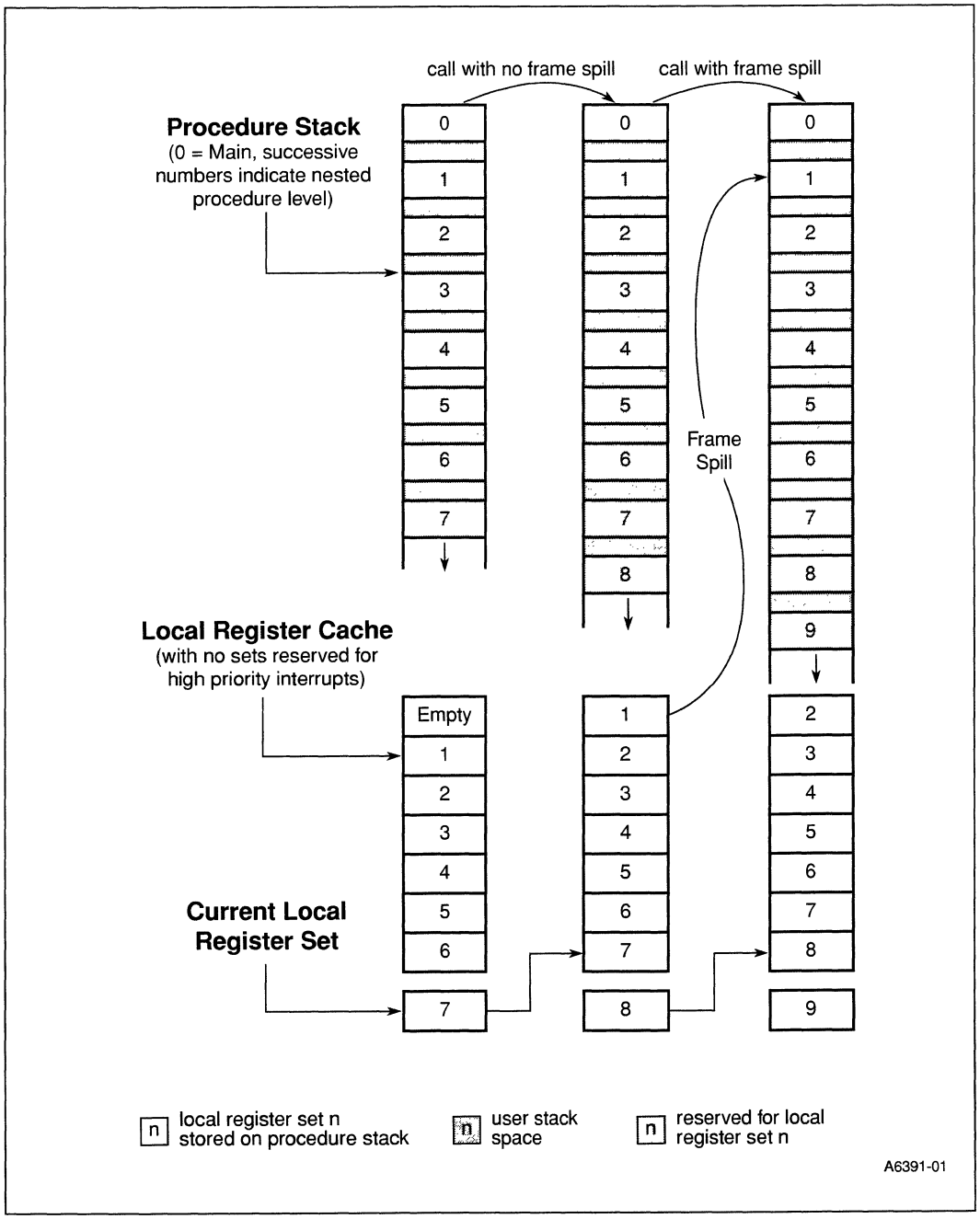
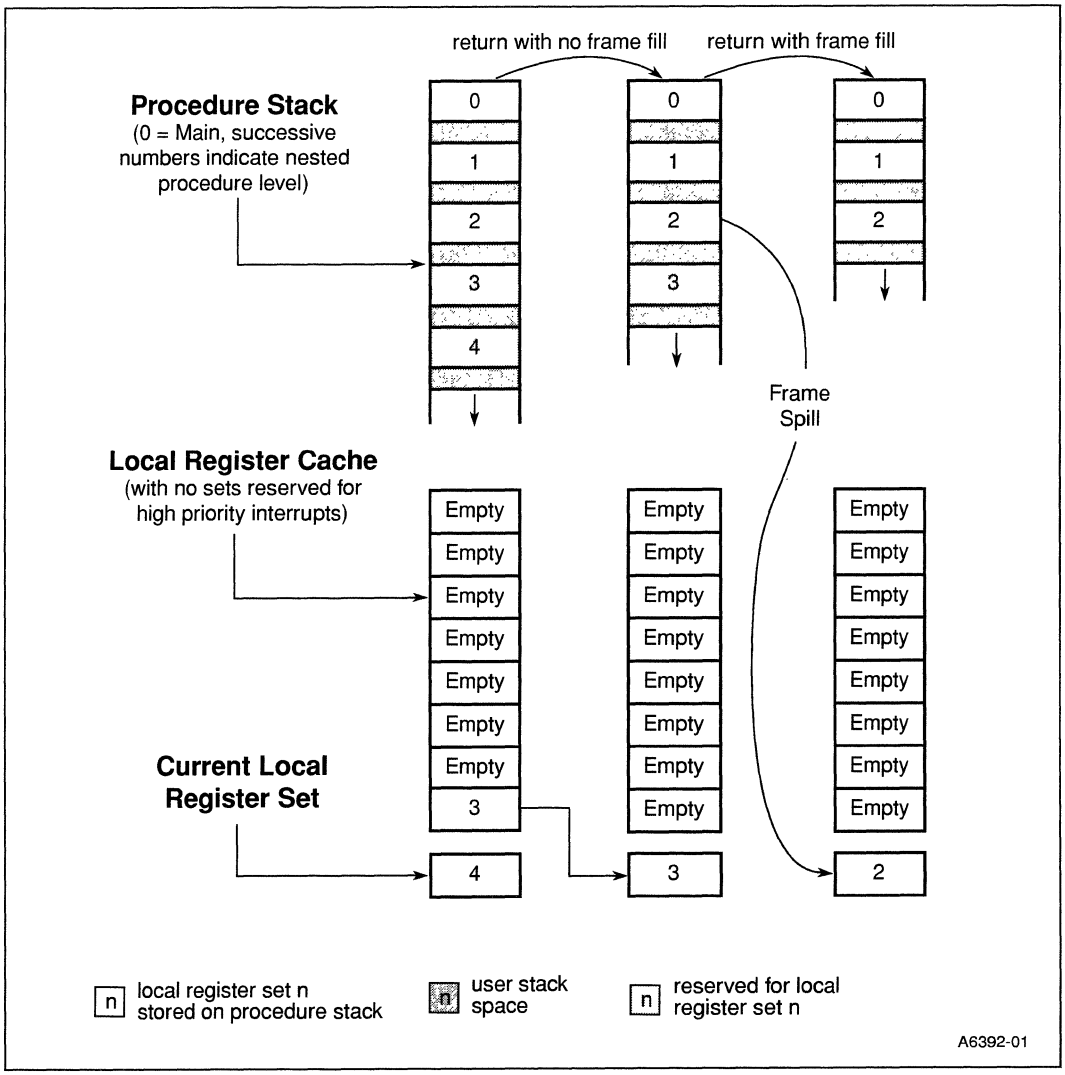


Figure 7-3. Frame Fill



7.1.5 Mapping Local Registers to the Procedure Stack

Each local register set is mapped to a register save area of its respective frame in the procedure stack (Figure 7-1). Saved local register sets are frequently cached on-chip rather than saved to memory. This is not a write-through cache. Local register set contents are not saved automatically to the save area in memory when the register set is cached. This would cause a significant performance loss for call operations.

Also, no automatic update policy is implemented for the register cache. If the register save area in memory for a cached register set is modified, there is no guarantee that the modification is reflected when the register set is restored. For a frame spill, the set must be flushed to memory prior to the modification for the modification to be valid.

The **flushreg** instruction causes the contents of all cached local register sets to be written (flushed) to their associated stack frames in memory. The register cache is then invalidated, meaning that all flushed register sets are restored from their save areas in memory. The current set of local registers is not written to memory. **flushreg** is commonly used in debuggers or fault handlers to gain access to all saved local registers. In this way, call history may be traced back through nested procedures. **flushreg** is also used when implementing task switches in multitasking kernels. The procedure stack is changed as part of the task switch. To change the procedure stack, **flushreg** is executed to update the current procedure stack and invalidate all entries in the local register cache. Next, the procedure stack is changed by directly modifying the FP and SP registers and executing a call operation. After **flushreg** executes, the procedure stack may also be changed by modifying the previous frame in memory and executing a return operation.

When a set of local registers is assigned to a new procedure, the processor may or may not clear or initialize these registers. Therefore, initial register contents are unpredictable. Also, the processor does not initialize the local register save area in the newly created stack frame for the procedure; its contents are equally unpredictable.

7.2 Modifying the PFP Register

The FP must not be directly modified by user software or risk corrupting the local registers. Instead, implement context switches by modifying the PFP.

Modification of the PFP is typically for context switches; as part of the switch, the active procedure changes the pointer to the frame that it returns to (previous frame pointer — PFP). Great care should be taken in modifying the PFP. In the general case, a **flushreg** must be issued before and after modifying the PFP when the local register cache is enabled (Example 7-1). This requirement ensures the correct operation of a context switch on all i960 processors in all situations.

Example 7-1. flushreg

```
# Do a context switch.
# Assume PFP = 0x5000.
flushreg      # Flush Frames to correct address.
lda 0x8000,pfp
flushreg      # Ensure that "ret" gets updated PFP.
ret
```

The **flushreg** before the modification is necessary to ensure that the frame of the previous context (mapped to 0x5000 in the example) is "spilled" to the proper external memory address and removed from the local register cache. If the **flushreg** before the modification was omitted, a **flushreg** (or implicit frame spill due to an interrupt) after the modification of PFP would cause the frame of the previous context to be written to the wrong location in external memory.

The **flushreg** after the modification ensures that outstanding results are completely written to the PFP before a subsequent **ret** instruction can be executed. Recall that the **ret** instruction uses the low-order 4 bits of the PFP to select which **ret** function to perform. Requiring the **flushreg** after the PFP modification allows an i960 implementation to implement a simple mechanism that quickly selects the **ret** function at the time the **ret** instruction is issued and provides a faster return operation.

Note the **flushreg** after the modification executes very quickly because the local register cache has already been flushed by the **flushreg** before; only synchronization of the PFP is performed. i960 processor implementations may provide other mechanisms to ensure PFP synchronization in addition to **flushreg**, but a **flushreg** after a PFP modification is ensured to work on all i960 processors.

7.3 Parameter Passing

Parameters are passed between procedures in two ways:

value Parameters are passed directly to the calling procedure as part of the call and return mechanism. This is the fastest method of passing parameters.

reference Parameters are stored in an argument list in memory and a pointer to the argument list is passed in a global register.

When passing parameters by value, the calling procedure stores the parameters to be passed in global registers. Since the calling procedure and the called procedure share the global registers, the called procedure has direct access to the parameters after the call.

When a procedure needs to pass more parameters than fits in the global registers, they can be passed by reference. Here, parameters are placed in an argument list and a pointer to the argument list is placed in a global register.

The argument list can be stored anywhere in memory; however, a convenient place to store an argument list is in the stack for a calling procedure. Space for the argument list is created by incrementing the SP register value. If the argument list is stored in the current stack, the argument list is automatically deallocated when no longer needed.

A procedure receives parameters from — and returns values to — other calling procedures. To do this successfully and consistently, all procedures must agree on the use of the global registers.

Parameter registers pass values into a function. Up to 12 parameters can be passed by value using the global registers. If the number of parameters exceeds 12, additional parameters are passed using the calling procedure's stack; a pointer to the argument list is passed in a pre-designated register. Similarly, several registers are set aside for return arguments and a return argument block pointer is defined to point to additional parameters. If the number of return arguments exceeds the available number of return argument registers, the calling procedure passes a pointer to an argument list on its stack where the remaining return values is placed. Example 7-2 illustrates parameter passing by value and by reference.

Local registers are automatically saved when a call is made. Because of the local register cache, they are saved quickly and with no external bus traffic. The efficiency of the local register mechanism plays an important role in two cases when calls are made:

1. When a procedure is called which contains other calls, global parameter registers should be moved to working local registers at the beginning of the procedure. In this way, parameter registers are freed and nested calls are easily managed. The register move instruction necessary to perform this action is very fast; the working parameters — now in local registers — are saved efficiently when nested calls are made.
2. When other procedures are nested within an interrupt or fault procedure, the procedure must preserve all normally non-preserved parameter registers, such as the global registers. This is necessary because the interrupt or fault occurs at any point in the user's program and a return from an interrupt or fault must restore the exact processor state. The interrupt or fault procedure can move non-preserved global registers to local registers before the nested call.

Example 7-2. Parameter Passing Code Example

```
# Example of parameter passing . . .
# C-source:             int a,b[10];
#                       a = proc1(a,1,'x',&b[0]);
#                       assembles to ...
    mov                 r3,g0               # value of a
    ldconst             1,g1                # value of 1
    ldconst             120,g2              # value of "x"
    lda                 0x40(fp),g3         # reference to b[10]
    call                _proc1
    mov                 g0,r3               # save return value in "a"
                        .
                        .
_proc1:
    movq                g0,r4               # save parameters
                        .
                        .                   # other instructions in procedure
                        .                   # and nested calls
    mov                 r3,g0               # load return parameter
    ret
```

7.4 Local Calls

A local call does not cause a stack switch. A local call can be made two ways:

- with the **call** and **callx** instructions; or
- with a system-local call as described in Section 7.5, "System Calls" on page 7-15.

call specifies the address of the called procedures as the IP plus a signed, 24-bit displacement (i.e., -2^{23} to $2^{23} - 4$). **callx** allows any of the addressing modes to be used to specify the procedure address. The IP-with-displacement addressing mode allows full 32-bit IP-relative addressing.

When a local call is made with a **call** or **callx**, the processor performs the same operation as described in Section 7.1.3.1, "Call Operation" on page 7-6. The target IP for the call is derived from the instruction's operands and the new stack frame is allocated on the current stack.

7.5 System Calls

A system call is a call made via the system procedure table. It can be used to make a system-local call — similar to a local call made with **call** and **callx** in the sense that there is no stack nor mode switch — or a system supervisor call. A system call is initiated with **calls**, which requires a procedure number operand. The procedure number provides an index into the system procedure table, where the processor finds IPs for specific procedures.

Using an i960 processor language assembler, a system procedure is directly declared using the .sysproc directive. At link time, the optimized call directive, callj, is replaced with a **calls** when a system procedure target is specified. (Refer to current i960 processor assembler documentation for a description of the .sysproc and callj directives.)

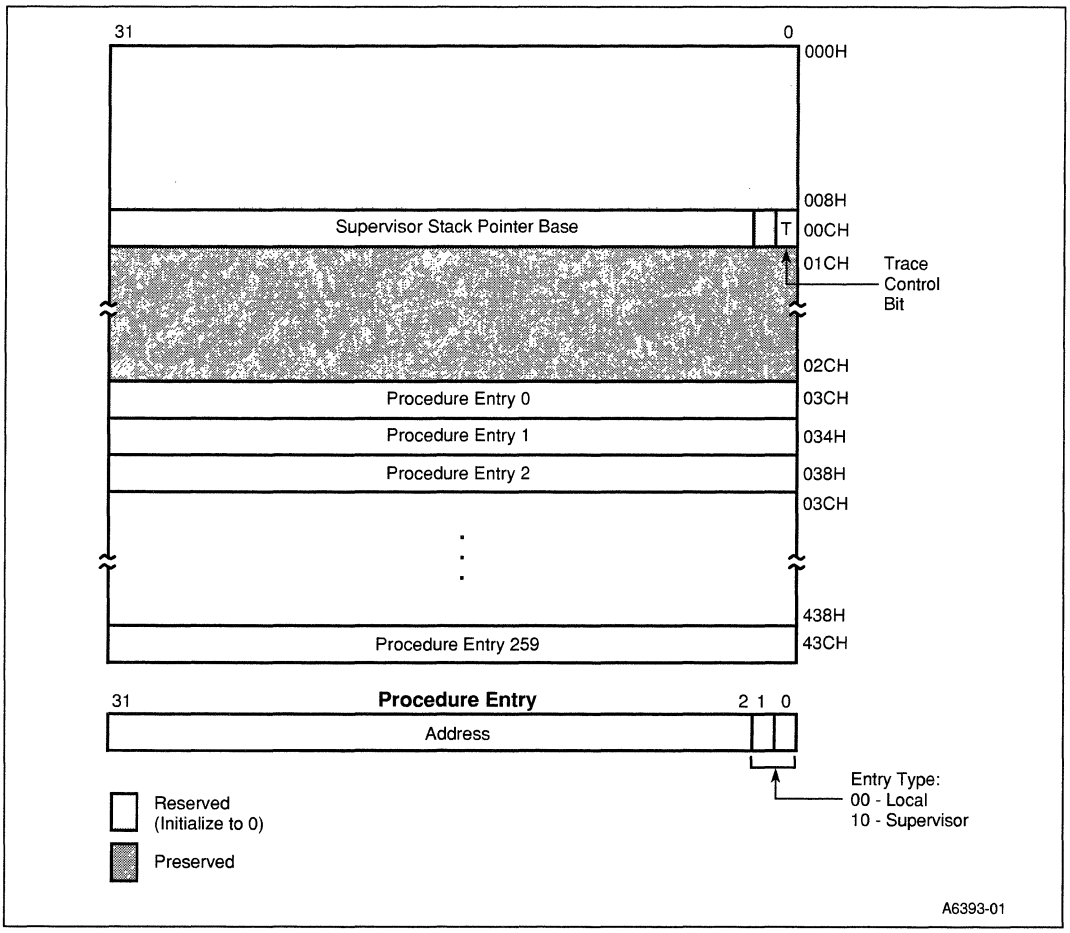
The system call mechanism offers two benefits. First, it supports application software portability. System calls are commonly used to call kernel services. By calling these services with a procedure number rather than a specific IP, applications software does not need to be changed each time the implementation of the kernel services is modified. Only the entries in the system procedure table must be changed. Second, the ability to switch to a different execution mode and stack with a system supervisor call allows kernel procedures and data to be insulated from applications code. This benefit is further described in Section 3.7, "User-Supervisor Protection Model" on page 3-18.

7.5.1 System Procedure Table

The system procedure table is a data structure for storing IPs to system procedures. These can be procedures which software can access through (1) a system call or (2) the fault handling mechanism. Using the system procedure table to store IPs for fault handling is described in Section 9.1, "Fault Handling Overview" on page 9-1.

Figure 7-4 shows the system procedure table structure. It is 1088 bytes in length and can have up to 260 procedure entries. At initialization, the processor caches a pointer to the system procedure table. This pointer is located in the PRCB. The following subsections describe this table's fields.

Figure 7-4. System Procedure Table



7.5.1.1 Procedure Entries

A procedure entry in the system procedure table specifies a procedure's location and type. Each entry is one word in length and consists of an address (IP) field and a type field. The address field gives the address of the first instruction of the target procedure. Since all instructions are word aligned, only the entry's 30 most significant bits are used for the address. The entry's two least-significant bits specify entry type. The procedure entry type field indicates call type: system-local call or system-supervisor call (Table 7-1). On a system call, the processor performs different actions depending on the type of call selected.

Table 7-1. Encodings of Entry Type Field in System Procedure Table

Encoding	Call Type
00	System-Local Call
01	Reserved[1]
10	System-Supervisor Call
11	Reserved[1]

1. Calls with reserved entry types have unpredictable behavior.

7.5.1.2 Supervisor Stack Pointer

When a system-supervisor call is made, the processor switches to a new stack called the *supervisor stack*, if not already in supervisor mode. The processor gets a pointer to this stack from the supervisor stack pointer field in the system procedure table (Figure 7-4) during the reset initialization sequence and caches the pointer internally. Only the 30 most significant bits of the supervisor stack pointer are given. The processor aligns this value to the next 16-byte boundary to determine the first byte of the new stack frame.

7.5.1.3 Trace Control Bit

The trace control bit (byte 12, bit 0) specifies the new value of the trace enable bit in the PC register (PC.te) when a system-supervisor call causes a switch from user mode to supervisor mode. Setting this bit to 1 enables tracing in the supervisor mode; setting it to 0 disables tracing. The use of this bit is described in Section 10.1.2, "PC Trace Enable Bit and Trace-Fault-Pending Flag" on page 10-3.

7.5.2 System Call to a Local Procedure

When a **calls** instruction references an entry in the system procedure table with an entry type of 00, the processor executes a system-local call to the selected procedure. The action that the processor performs is the same as described in Section 7.1.3.1, "Call Operation" on page 7-6. The call's target IP is taken from the system procedure table and the new stack frame is allocated on the current stack, and the processor does not switch to supervisor mode. The **calls** algorithm is described in Section 6.2.14, "calls" on page 6-22.

7.5.3 System Call to a Supervisor Procedure

When a **calls** instruction references an entry in the system procedure table with an entry type of 10_2, the processor executes a system-supervisor call to the selected procedure. The call's target IP is taken from the system procedure table.

The processor performs the same action as described in Section 7.1.3.1, "Call Operation" on page 7-6, with the following exceptions:

- If the processor is in user mode, it switches to supervisor mode.
- If a mode switch occurs, SP is read from the Supervisor Stack Pointer (SSP) base. A new frame for the called procedure is placed at the location pointed to after alignment of SP.
- If no mode switch occurs, the new frame is allocated on the current stack.
- If a mode switch occurs, the state of the trace enable bit in the PC register is saved in the return type field in the PFP register. The trace enable bit is then loaded from the trace control bit in the system procedure table.
- If no mode switch occurs, the value 000_2 (**calls** instruction) or 001_2 (fault call) is saved in the return type field of the pfp register.

When the processor switches to supervisor mode, it remains in that mode and creates new frames on the supervisor stack until a return is performed from the procedure that caused the original switch to supervisor mode. While in supervisor mode, either the local call instructions (**call** and **callx**) or **calls** can be used to call procedures.

The user-supervisor protection model and its relationship to the supervisor call are described in Section 3.7, "User-Supervisor Protection Model" on page 3-18.

7.6 User and Supervisor Stacks

When using the user-supervisor protection mechanism, the processor maintains separate stacks in the address space. One of these stacks — the user stack — is for procedures executed in user mode; the other stack — the supervisor stack — is for procedures executed in supervisor mode.

The user and supervisor stacks are identical in structure (Figure 7-1). The base stack pointer for the supervisor stack is automatically read from the system procedure table and cached internally during initialization. Each time a user-to-supervisor mode switch occurs, the cached supervisor stack pointer base is used for the starting point of the new supervisor stack. The base stack pointer for the user stack is usually created in the initialization code. See Section 11.2, "Intel® 80303 I/O Processor Initialization" on page 11-2. The base stack pointers must be aligned to a 16-byte boundary; otherwise, the first frame pointer on the interrupt stack is rounded up to the previous 16-byte boundary.

7.7 Interrupt and Fault Calls

The architecture defines two types of implicit calls that make use of the call and return mechanism: interrupt-handling procedure calls and fault-handling procedure calls. A call to an interrupt procedure is similar to a system-supervisor call. Here, the processor obtains pointers to the interrupt procedures through the interrupt table. The processor always switches to supervisor mode on an interrupt procedure call.

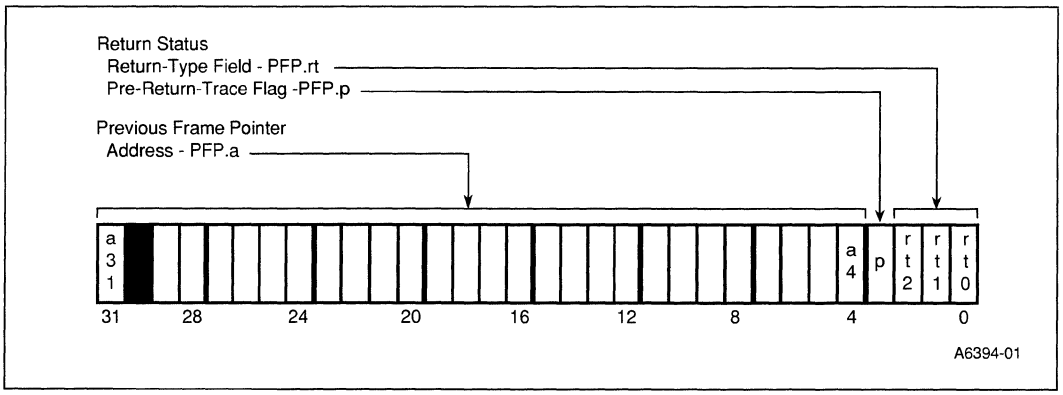
A call to a fault procedure is similar to a system call. Fault procedure calls can be local calls or supervisor calls. The processor obtains pointers to fault procedures through the fault table and (optionally) through the system procedure table.

When a fault call or interrupt call is made, a fault record or interrupt record is placed in the newly generated stack frame for the call. These records hold the machine state and information to identify the fault or interrupt. When a return from an interrupt or fault is executed, machine state is restored from these records. See Chapter 8, "PCI and Peripheral Interrupt Controller Unit" and Chapter 9, "Faults" for more information on the structure of the fault and interrupt records.

7.8 Returns

The return (**ret**) instruction provides a generalized return mechanism that can be used to return from any procedure that was entered by **call**, **calls**, **callx**, an interrupt call or a fault call. When **ret** executes, the processor uses the information from the return-type field in the PFP register (Figure 7-5) to determine the type of return action to take.

Figure 7-5. Previous Frame Pointer Register – PFP



return-type field indicates the type of call which was made. Table 7-2 shows the return-type field encoding for the various calls: local, supervisor, interrupt and fault.

trace-on-return flag (PFP.rt0 or bit 0 of the return-type field) stores the trace enable bit value when an explicit system-supervisor call is made from user mode. When the call is made, the PC register trace enable bit is saved as the trace-on-return flag and then replaced by the trace controls bit in the system procedure table. On a return, the trace enable bit's original value is restored. This mechanism allows instruction tracing to be turned on or off when a supervisor mode switch occurs. See Section 10.5.2.1, "Tracing on Explicit Call" on page 10-11.

prereturn-trace flag (PFP.p) is used in conjunction with call-trace and prereturn-trace modes. If call-trace mode is enabled when a call is made, the processor sets the prereturn-trace flag; otherwise it clears the flag. Then, if this flag is set and prereturn-trace mode is enabled, a prereturn trace event is generated on a return, before any actions associated with the return operation are performed. See Section 10.2, "Trace Modes" on page 10-3 for a discussion of interaction between call-trace and prereturn-trace modes with the prereturn-trace flag.

Table 7-2. Encoding of Return Status Field

Return Status Field	Call Type	Return Action
000	Local call (system-local call or system-supervisor call made from supervisor mode)	Local return (return to local stack; no mode switch)
001	Fault call	Fault return
01t	System-supervisor from user mode	Supervisor return (return to user stack, mode switch to user mode, trace enable bit is replaced with the t[1] bit stored in the PFP register on the call)
100	reserved [2]	
101	reserved[2]	
110	reserved[2]	
111	Interrupt call	Interrupt return

NOTES:
1. "t" denotes the trace-on-return flag; used only for system supervisor calls which cause a user-to-supervisor mode switch.
2. This return type results in unpredictable behavior.

7.9 Branch-and-Link

A branch-and-link is executed using either the branch-and-link instruction (**bal**) or branch-and-link-extended instruction (**balx**). When either instruction executes, the processor branches to the first instruction of the called procedure (the target instruction), while saving a return IP for the calling procedure in a register. The called procedure uses the same set of local registers and stack frame as the calling procedure:

- For **bal**, the return IP is automatically saved in global register g14
- For **balx**, the return IP instruction is saved in a register specified by one of the instruction's operands

A return from a branch-and-link is generally carried out with a **bx** (branch extended) instruction, where the branch target is the address saved with the branch-and-link instruction. The branch-and-link method of making procedure calls is recommended for calls to leaf procedures. Leaf procedures typically call no other procedures. Branch-and-link is the fastest way to make a call, providing the calling procedure does not require its own registers or stack frame.

PCI and Peripheral Interrupt Controller Unit 8

This chapter describes the Intel® 80303 I/O processor Interrupt Controller Unit. The operation modes, setup, external memory interface, and implementation of the interrupts are described in this chapter.

8.1 Overview

An interrupt is an event that causes a temporary break in program execution so the processor can handle another task. Interrupts commonly request I/O services or synchronize the processor with some external hardware activity. For interrupt handler portability across the Intel® i960® processor family, the architecture defines a consistent interrupt state and interrupt-priority-handling mechanism. To manage and prioritize interrupt requests in parallel with processor execution, the 80303 I/O processor provides an on-chip programmable interrupt controller.

When the processor is redirected to service an interrupt, it uses a vector number that accompanies the interrupt request to locate the vector entry in the interrupt table. From that entry, it gets an address to the first instruction of the selected interrupt procedure. The processor then makes an implicit call to that procedure.

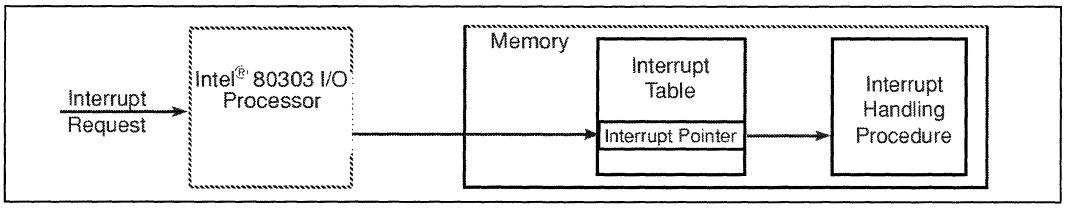
When the interrupt call is made, the processor uses a dedicated interrupt stack. The processor creates a new frame for the interrupt on this stack and a new set of local registers is allocated to the interrupt procedure. The interrupted program's current state is also saved.

Upon return from the interrupt procedure, the processor restores the interrupted program's state, switches back to the stack that the processor was using prior to the interrupt and resumes program execution.

Since interrupts are handled based on priority, requested interrupts are often saved for later service rather than handled immediately. The mechanism for saving the interrupt is referred to as interrupt posting. Interrupt posting is described in Section 8.1.6, "Posting Interrupts" on page 8-7.

The i960 processor defines two data structures to support interrupt processing: the interrupt table (Figure 8-1) and interrupt stack. The interrupt table contains 248 vectors for interrupt handling procedures (eight of which are reserved) and an area for posting software requested interrupts. The interrupt stack prevents interrupt handling procedures from using the stack in use by the application program. It also locates the interrupt stack in a different area of memory than the user and supervisor stack (e.g., fast SRAM).

Figure 8-1. Interrupt Handling Data Structures



Requests for interrupt service come from many sources and are prioritized such that instruction execution is redirected only when an interrupt request is of higher priority than that of the executing task. On the 80303 I/O processor, interrupt requests may originate from external hardware sources, internal peripherals or software. The 80303 I/O processor contains a number of integrated peripherals which may generate interrupts, including:

- DMA Channel 0
- DMA Channel 1
- DMA Channel 2
- Primary and Secondary Bridge Interface
- Performance Monitoring Unit
- Timers 0 and 1
- Primary ATU
- Secondary ATU
- I^2C Bus Interface Unit
- Application Accelerator Unit
- Messaging Unit
- Memory Controller Unit

The interrupt controller can also intercept external secondary PCI interrupts and forward them to the primary PCI interrupt pins.

Interrupts are detected with the chip's 6-bit interrupt port and with a dedicated Non-Maskable Interrupt (**NMI#**) input in the Intel® i960® core processor interrupt controller. Interrupt requests originate from software by the **sysctl** instruction. To manage and prioritize all possible interrupts, the processor integrates an on-chip programmable interrupt controller.

8.1.1 The Intel® 80303 I/O Processor Core Interrupt Architecture

The 80303 I/O processor contains the same core interrupt architecture as many other 80960 family members. Some of the core features include the interrupt record and stack, the way interrupts are posted, and the way interrupt priorities are resolved. These basic architectural features are detailed in the following sections.

8.1.2 Software Requirements For Interrupt Handling

To use the processor's interrupt handling facilities, user software must provide the following items in memory:

- Interrupt Table
- Interrupt Handler Routines
- Interrupt Stack

These items are established in memory as part of the initialization procedure. Once these items are present in memory and pointers to them have been entered in the appropriate system data structures, the processor handles interrupts automatically and independently from software.

8.1.3 Interrupt Priority

Each procedure pointer's priority is defined by dividing the procedure pointer number by eight. Thus, at each priority level, there are eight possible procedure pointers (e.g., procedure pointers 8-15 have a priority of 1 and procedure pointers 246-255 have a priority of 31). Procedure pointers 0-7 cannot be used because a priority-0 interrupt would never successfully stop execution of a program of any priority. In addition, procedure pointers 244-247 and 249-251 are reserved; therefore, 241 procedure pointers are available to the user.

The processor compares its current priority with the interrupt request priority to determine whether to service the interrupt immediately or to delay service:

- The interrupt is serviced immediately when its priority is higher than the priority of the program or interrupt the processor is currently executing.
- The interrupt is posted as a pending interrupt (not serviced immediately) when the interrupt priority is less than or equal to the processor's current priority.

See Section 8.1.4.2, "Pending Interrupts" on page 8-5. When multiple interrupt requests are pending at the same priority level, the request with the highest vector number is serviced first.

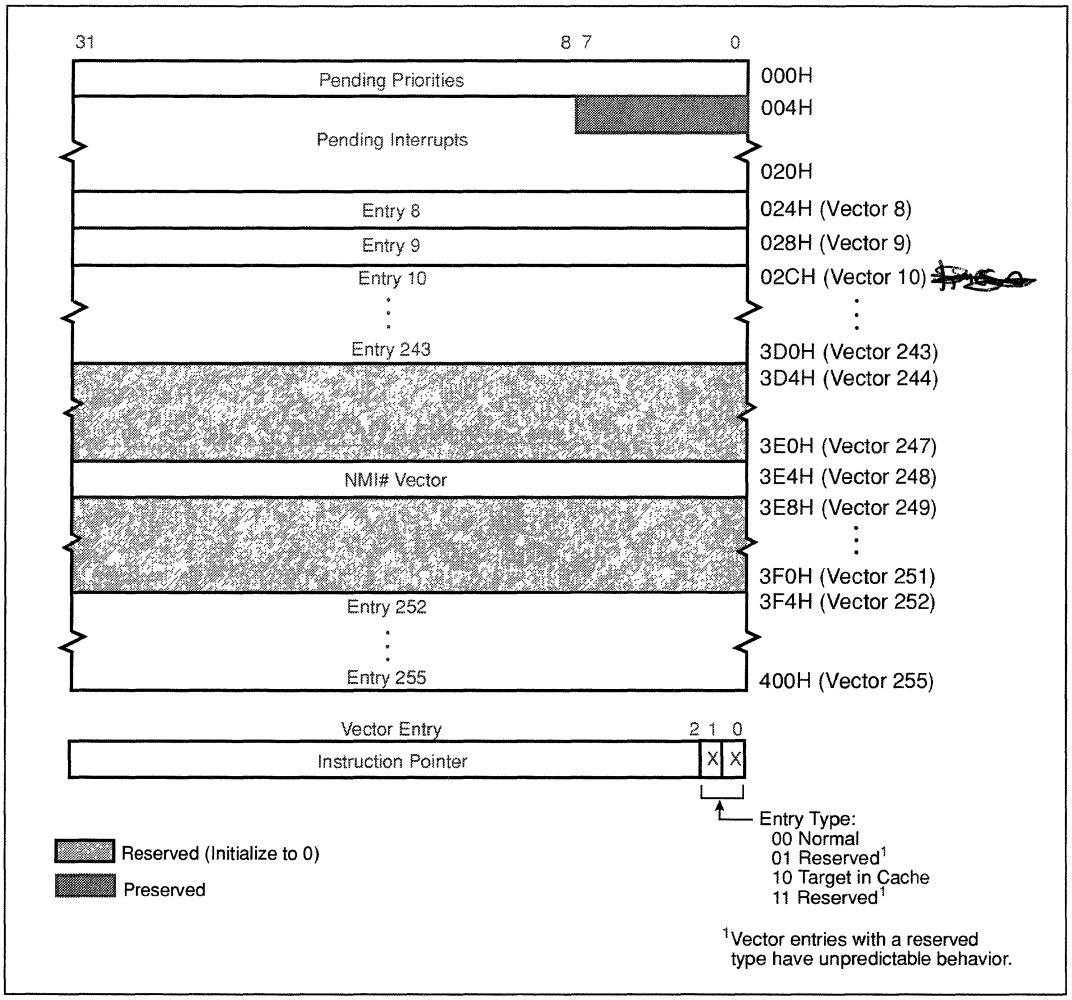
Priority-31 interrupts are handled as a special case. Even when the processor is executing at priority level 31, a priority-31 interrupt will interrupt the processor. On the 80303 I/O processor, the non-maskable interrupt (**NMI#**) interrupts priority-31 execution; no interrupt can interrupt an **NMI#** handler.

8.1.4 Interrupt Table

The interrupt table (Figure 8-2) is 1028 bytes in length and can be located anywhere in the non-reserved address space. It must be aligned on a word boundary. The processor reads a pointer to the interrupt table byte 0 during initialization. The interrupt table must be located in RAM so the processor can read and write the table's pending interrupt section for software or externally generated interrupts.

The interrupt table is divided into two sections: *vector entries* and *pending interrupts*. Each are described in the subsections that follow.

Figure 8-2. Interrupt Table



8.1.4.1 Vector Entries

A vector entry contains a specific interrupt handler's address. When an interrupt is serviced, the processor branches to the address specified by the vector entry.

Each interrupt is associated with an 8-bit vector number pointing to an interrupt table vector entry. The vector entry section contains 248 word-length entries. Vector numbers 8-243 and 252-255, and their associated vector entries, are used for conventional interrupts. Vector number 248 is the **NMI#** vector. Vector numbers 244-247 and 249-251 are reserved. Vector number 248 and its associated vector entry, is for the non-maskable interrupt (**NMI#**). Vector numbers 0-7 cannot be used.

Vector entry 248 contains the **NMI#** handler address. When the processor is initialized, the interrupt table **NMI#** vector automatically read and stored in internal data RAM location 0H. The **NMI#** vector is subsequently fetched from internal data RAM to improve interrupt performance.

The vector entry structure is given at the bottom of Figure 8-2. Each interrupt procedure must begin on a word boundary, so the processor assumes the two least significant vector bits are 0. Bits 0 and 1 of an entry indicate entry type: type 00 indicates the interrupt procedure should be fetched normally; type 10 indicates the interrupt procedure should be fetched from the instruction cache locked partition. Refer to Section 8.6.3.2, "Caching Interrupt Routines and Reserving Register Frames" on page 8-31. The other possible entry types are reserved and must not be used.

8.1.4.2 Pending Interrupts

The pending interrupts section comprises the first 36 bytes of thr interrupt table, divided into two fields: pending priorities (byte offset 0 through 3) and pending interrupts (4 through 35).

Each of the 32 bits in the pending priorities field indicate an interrupt priority. When the processor posts a pending interrupt in the interrupt table, the bit corresponding to the interrupt's priority is set (e.g., when an interrupt with a priority of 10 is posted in the interrupt table, bit 10 is set).

Each of the pending interrupts field 256 bits represent an interrupt procedure pointer. Byte offset 5 is for vectors 8 through 15, byte offset 6 is for vectors 16 through 23, and so on. Byte offset 4, the first byte of the pending interrupts field, is reserved. When an interrupt is posted, its corresponding pending interrupt field bit is set.

This encoding of the pending priority and pending interrupt fields permits the processor to first check for any pending interrupts with a priority greater than the current program and then determine the vector number of the interrupt with the highest priority.

8.1.4.3 Caching Portions of the Interrupt Table

The architecture allows all or part of the interrupt table to be cached internally to the processor. The purpose of caching these fields is to reduce interrupt latency by allowing the processor to access certain interrupt procedure pointers and the pending interrupt information without having to make external memory accesses. The 80303 I/O processor caches the following:

- The value of the highest priority posted in the pending priorities field.
- A predefined subset of interrupt procedure pointers (entries from the interrupt table).
- Pending interrupts received from external interrupt pins.

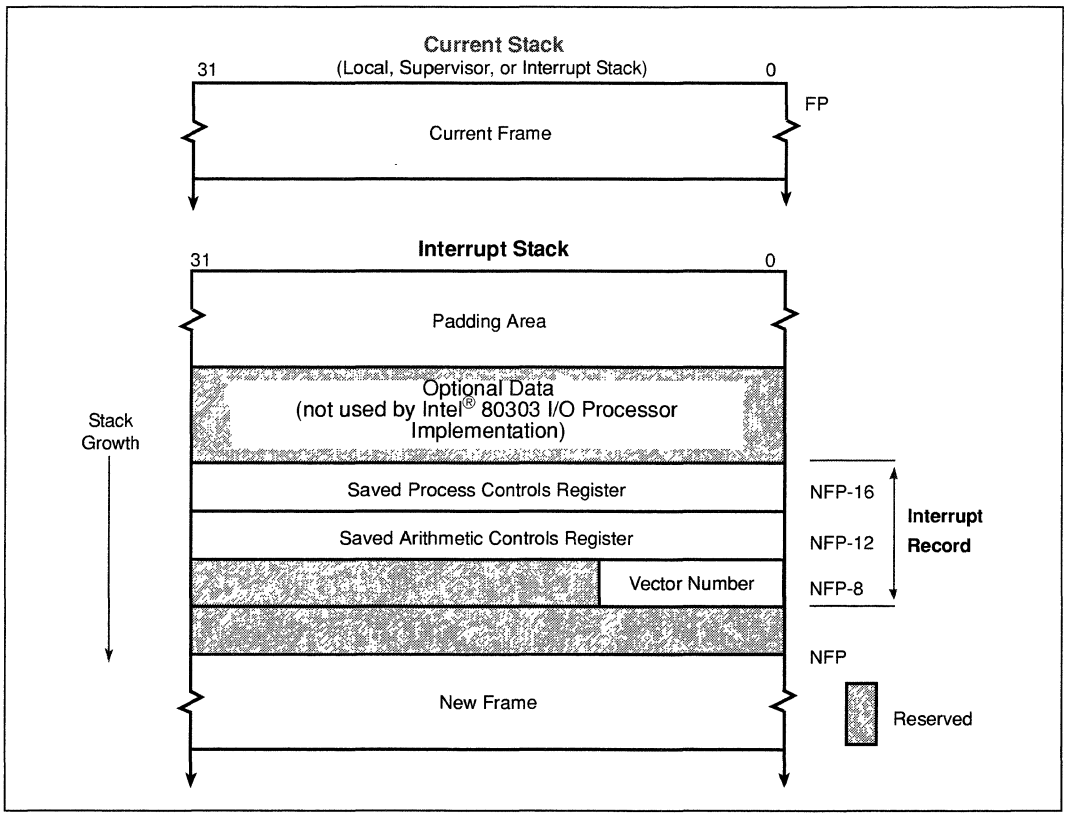
This caching mechanism is non-transparent; the processor may modify fields in a cached interrupt table without modifying the same fields in the interrupt table itself. Vector caching is described in Section 8.6.3.1, "Vector Caching Option" on page 8-30.

8.1.5 Interrupt Stack And Interrupt Record

The interrupt stack can be located anywhere in the non-reserved address space. The processor obtains a pointer to the base of the stack during initialization. As with the local stack, the interrupt stack grows from lower addresses to higher addresses.

The processor saves the state of an interrupted program, or an interrupted interrupt procedure, in a record on the interrupt stack. Figure 8-3 shows the structure of this interrupt record.

Figure 8-3. Storage of an Interrupt Record on the Interrupt Stack



The interrupt record is always stored on the interrupt stack adjacent to the new frame that is created for the interrupt handling procedure. It includes the state of the AC and PC registers at the time the interrupt was serviced and the interrupt procedure pointer number used. Relative to the new frame pointer (NFP), the saved AC register is located at address NFP-12, the saved PC register is located at address NFP-16.

In the 80303 I/O processor, the stack is aligned to a 16-byte boundary. When the processor needs to create a new frame on an interrupt call, it adds a padding area to the stack so that the new frame starts on a 16-byte boundary.

8.1.6 Posting Interrupts

Interrupts are posted to the processor by a number of different mechanisms; these are described in the following sections.

- Software interrupts: interrupts posted through the interrupt table, by software running on the 80303 I/O processor.
- External Interrupts: interrupts posted through the interrupt table, by an external agent to the 80303 I/O processor.
- Hardware interrupts: interrupts posted directly to the 80303 I/O processor through an implementation-dependent mechanism that may avoid using the interrupt table.

8.1.6.1 Posting Software Interrupts via sysctl

In the 80303 I/O processor, **sysctl** is typically used to request an interrupt in a program (see Example 8-1). The request interrupt message type (00H) is selected and the interrupt procedure pointer number is specified in the least significant byte of the instruction operand. See *i960® RP Microprocessor User's Manual* (272736), "sysctl" for a complete discussion of **sysctl**.

Example 8-1. Using sysctl to Request an Interrupt

```
ldconst 0x53,g5# Vector number 53H is loaded
               # into byte 0 of register g5 and
               # the value is zero extended into
               # byte 1 of the register
sysctl g5, g5, g5# Vector number 53H is posted
```

A literal can be used to post an interrupt with a vector number from 8 to 31. Here, the required value of 00H in the second byte of a register operand is implied.

The action of the processor when it executes the **sysctl** instruction is as follows:

1. The processor performs an atomic write to the interrupt table and sets the bits in the pending-interrupts and pending-priorities fields that correspond to the requested interrupt.
2. The processor updates the internal software priority register with the highest pending priority value from the interrupt table. This may be the priority of the interrupt that was just posted.

The interrupt controller continuously compares the following three values: software priority register, current process priority, priority of the highest pending hardware-generated interrupt. When the software priority register value is the highest of the three, the following actions occur:

1. The interrupt controller signals the core that a software-generated interrupt is to be serviced.
2. The core checks the interrupt table in memory, determines the vector number of the highest priority pending interrupt and clears the pending-interrupts and pending-priorities bits in the table that correspond to that interrupt.
3. The core detects the interrupt with the next highest priority that is posted in the interrupt table (if any) and writes that value into the software priority register.
4. The core services the highest priority interrupt.

When more than one pending interrupt is posted in the interrupt table at the same interrupt priority, the core handles the interrupt with the highest vector number first. The software priority register is an internal register and, as such, is not visible to the user. The core only updates this register's value when **sysctl** requests an interrupt or when a software-generated interrupt is serviced.

8.1.6.2 Posting Software Interrupts Directly in the Interrupt Table

In special cases within a single processor system, software can post interrupts by setting the desired pending-interrupt and pending-priorities bits directly. Direct posting requires that software ensure that no external I/O agents post a pending interrupt simultaneously, and that an interrupt cannot occur after one bit is set but before the other is set. Note, however, that this method is not recommended.

8.1.6.3 Posting External Interrupts

An external agent posts (sets) a pending interrupt with vector "v" to the i960 processor through the interrupt table by executing the following algorithm:

```
External_Agent_Posting:

x = atomic_read(pending_priorities); #synchronize;
z = read(pending_interrupts[v/8]);
x[v/8] = 1;
z[v mod 8] = 1;
write(pending_interrupts[v/8]) = z;
atomic_write(pending_priorities) = x;
```

Generally, software cannot use this algorithm to post interrupts because there is no way for software to have an atomic (locking) read/write span multiple instructions.

8.1.6.4 Posting Hardware Interrupts

Certain interrupts are posted directly to the processor by an implementation-dependent mechanism that can bypass the interrupt table. This is often done for performance reasons.

8.1.7 Resolving Interrupt Priority

The interrupt controller continuously compares the processor's priority to the priorities of the highest-posted software interrupt and the highest-pending hardware interrupt. The core is interrupted when a pending interrupt request is higher than the processor priority or has a priority of 31. (Note that a priority-31 interrupt handler can be interrupted by another priority-31 interrupt.) There are no priority-0 interrupts, since such an interrupt would never have a priority higher than the current process, and would therefore never be serviced.

In the event that both hardware and software requested interrupts are posted at the same level, the hardware interrupt is delivered first while the software interrupt is left pending. As a result, when both priority-31 hardware- and software-requested interrupts are pending, control is first transferred to the interrupt handler for the hardware-requested interrupt. However, before the first instruction of that handler can be executed, the pending software-requested interrupt is delivered and control is transferred to the corresponding interrupt handler.

Example 8-2. Interrupt Resolution

```
/* Model used to resolve interrupts between execution of all macro instructions */
if (NMI#_pending && !block_NMI)
    { block_NMI = true;  /* Reset on return from NMI INTR handler */
        vecnum = 248; vector_addr = 0;
        PC.priority = 31;
        push_local_register_set();
        goto common_interrupt_process; }
if (ICON.gie == enabled) {
    expand_HW_int();
    temp = max(HW_Int_Priority, SW_Int_Priority);
    if (temp == 31 || temp > PC.priority)
        { PC.priority = temp;
            if (SW_Int_Priority > HW_Int_Priority) goto Deliver_SW_Int;
            else{ vecnum = HW_vecnum; goto Deliver_HW_Int;}
        }
    }
```

8.1.8 Sampling Pending Interrupts in the Interrupt Table

At specific points, the processor checks the interrupt table for pending interrupts posted. When one is found, it is handled as if the interrupt occurred at that time. In the 80303 I/O processor, a check for pending interrupts in the interrupt table is made when requesting a software interrupt with **sysctl** or when servicing a software interrupt.

When a check of the interrupt table is made, the following algorithm is used. Since the pending interrupts may be cached, the check for pending interrupt operation may not involve any memory operations. The algorithm uses synchronization because there may be multiple agents posting and unposting interrupts. In the algorithm, w, x, y, and z are temporary registers within the processor.

Example 8-3. Pending Interrupts

```
Check_For_Pending_Interrupts:

x = read(pending_priorities);
if(x == 0) return(); #nothing to do
y = most_significant_bit(x);
if(y != 31 && y <= current_priority) return();
x = atomic_read(pending_priorities); #synchronize
if(x == 0)
    {atomic_write(pending_priorities) = x;
    return();} #interrupts disappeared
        # (e.g., handled by another processor)
y = most_significant_bit(x); #must be repeated
if(y != 31 && y <= current_priority)
    {atomic_write(pending_priorities) = x;
    return();} #interrupt disappeared
z = read(pending_interrupts[y]); #z is a byte
if(z == 0)
    {x[y] = 0; #false alarm, should not happen
    atomic_write(pending_priorities) = x;
    return();}
else
    {w = most_significant_bit[z];
    z[w] = 0;
    write(pending_interrupts[y]) = z;
    if(z == 0) x[y] = 0; #no others at this level
    atomic_write(pending_priorities) = x;
    take_interrupt();}
```

The algorithm shows that the pending interrupts are marked by a bit in the Pending Interrupts Field, and that the Pending Priorities Field is an optimization. The processor examines Pending Interrupts only when the corresponding bit in Pending Priorities is set.

The steps prior to the `atomic_read` are another optimization. Note that these steps must be repeated within the synchronized critical section, since another processor could have spotted and accepted the same pending interrupt(s).

Use **sysctl** with a vector in the range 0 to 7 to force the core to check the interrupt table for pending interrupts. When an external agent is posting interrupts to a shared interrupt table, use **sysctl** periodically to guarantee recognition of pending interrupts posted in the table by the external agent.

8.1.9 Saving the Interrupt Mask

Whenever an interrupt requested by the external interrupt pins or by the internal timers is serviced, the IMSK register is automatically saved in register r3 of the new local register set allocated for the interrupt handler. After the mask is saved, the IMSK register is optionally cleared. This masks all interrupts except **NMI#**s while an interrupt is serviced. Since the IMSK register value is saved, the interrupt procedure can restore the value before returning. The option of clearing the mask is selected by programming the ICON register as described in Section 8.7.1, "Interrupt Control Register (ICON)" on page 8-36.

Priority-31 interrupts are interrupted by other priority-31 interrupts. For level-activated interrupt inputs, instructions within the interrupt handler are typically responsible for causing the source to deactivate. If these priority-31 interrupts are not masked, another priority-31 interrupt is signaled and serviced before the handler can deactivate the source. The first instruction of the interrupt handling procedure is never reached, unless the option is selected to clear the IMSK register on entry to the interrupt.

Another use of the mask is to lock out other interrupts when executing time-critical portions of an interrupt handling procedure. All hardware-generated interrupts are masked until software explicitly replaces the mask.

The processor does not restore r3 to the IMSK register when the interrupt return is executed. When the IMSK register is cleared, the interrupt handler must restore the IMSK register to enable interrupts after return from the handler.

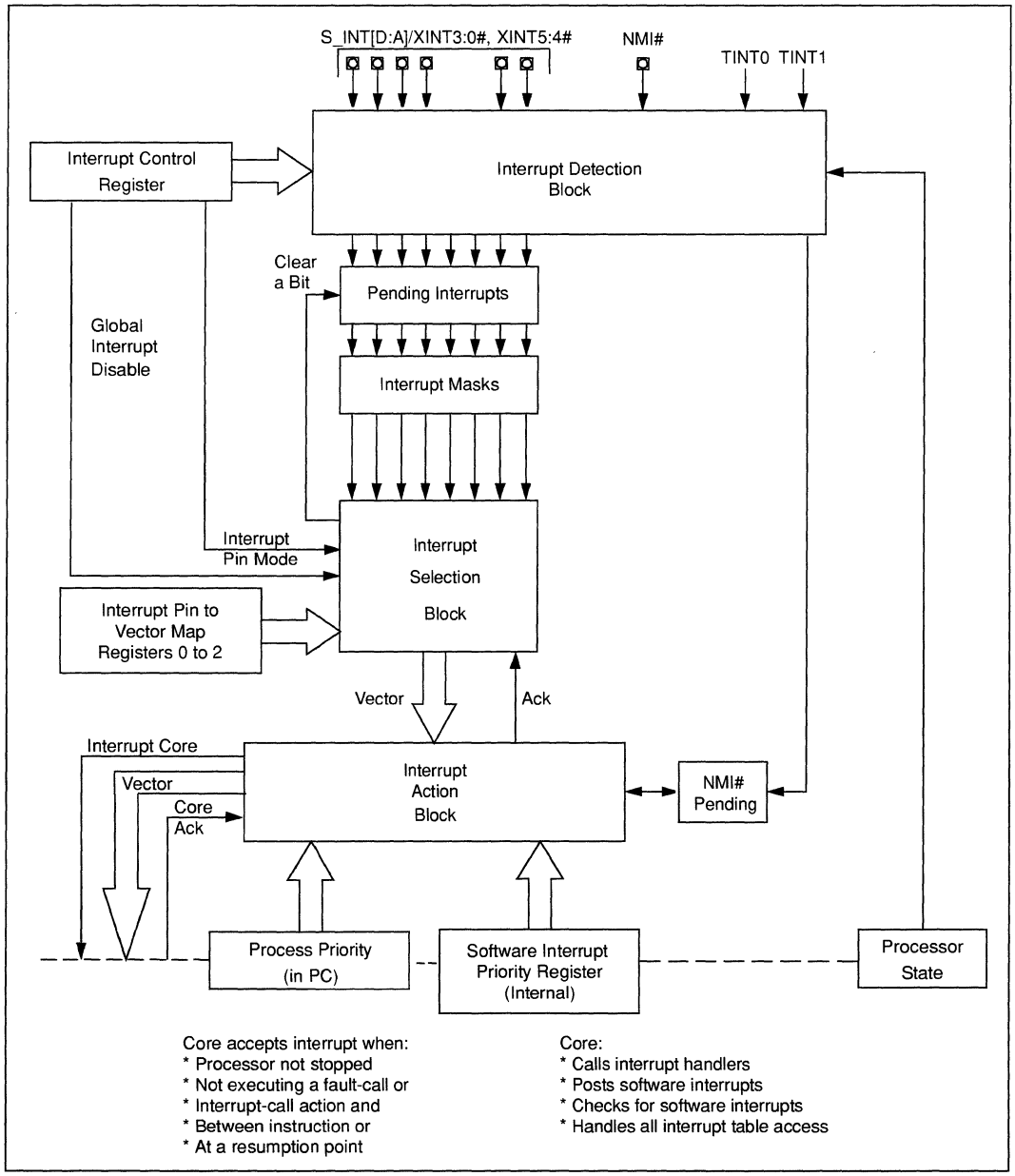
8.2 The Intel® i960® Core Processor Interrupt Controller

The 80303 I/O processor Interrupt Controller Unit (ICU) provides a flexible, low-latency means for requesting and posting interrupts and minimizing the core's interrupt handling burden. Acting independently from the core, the interrupt controller posts interrupts requested by hardware and software sources and compares the priorities of posted interrupts with the current process priority.

The interrupt controller provides the following features for managing hardware-requested interrupts:

- Low latency, high throughput handling.
- Six external interrupt pins.
- One non-maskable interrupt pin.
- Two internal timers sources.
- Peripheral interrupt sources.

Figure 8-4. Interrupt Controller



The user program interfaces to the interrupt controller with ten memory-mapped control registers. The Interrupt Control Register (ICON) and Interrupt Map Control Registers (IMAP0-IMAP2) provide configuration information. The Interrupt Pending Register (IPND) posts hardware-requested interrupts. The Interrupt Mask Register (IMSK) selectively masks hardware-requested interrupts.

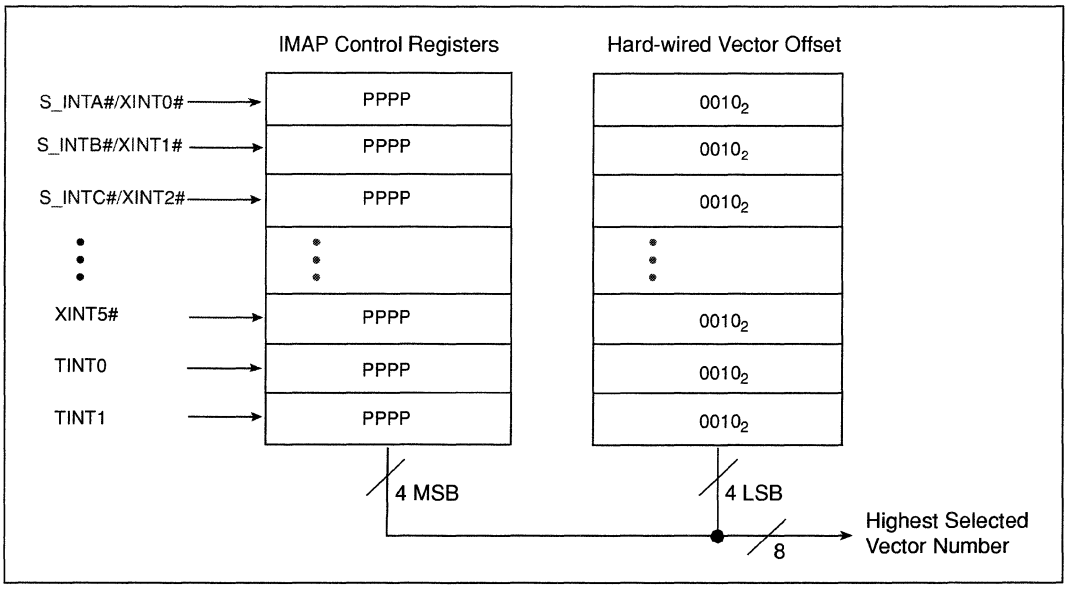
8.2.1 Interrupt Controller Dedicated Mode

The 80303 I/O processor interrupt controller external pins are set up for dedicated mode operation, where each external interrupt pin is assigned a vector number. Vector numbers that may be assigned to a pin are those with the encoding PPPP 0010_2 (Figure 8-5), where bits marked P are programmed with bits in the interrupt map (IMAP) registers. This encoding of programmable bits and preset bits can designate 15 unique vector numbers, each with a unique, even-numbered priority. (Vector 0000 0010_2 is undefined; it has a priority of 0.)

Interrupts are posted in the interrupt pending (IPND) register. Single bits in the IPND register correspond to each of the eight dedicated external interrupt inputs, or the two timer inputs to the interrupt controller. The interrupt mask (IMSK) register selectively masks each of the interrupts. Optionally, the IMSK register can be saved and cleared when an interrupt is serviced. This locks out other hardware-generated interrupts until the mask is restored. See Section 8.7, "Register Definitions" on page 8-35 for a further description of the IMSK, IPND and IMAP registers.

Interrupt vectors are assigned to timer inputs in the same way external pins are assigned vectors.

Figure 8-5. Interrupt Pin Vector Assignment



8.2.2 Interrupt Detection

The **XINT5:0#** pins and the NMI# pin use level-low detection. All of the interrupt pins use fast sampling.

For low-level detection, the pin's bit in the IPND register remains set as long as the pin is asserted (low). The processor attempts to clear the IPND bit on entry into the interrupt handler. However, if the active level on the pin is not removed at this time, the bit in the IPND register remains set until the source of the interrupt is deactivated and the IPND bit is explicitly cleared by software. Software may attempt to clear an interrupt pending bit before the active level on the corresponding pin is removed. In this case, the active level on the interrupt pin causes the pending bit to remain asserted.

After the interrupt signal is deasserted, the handler then clears the interrupt pending bit for that source before return from handler is executed. If the pending bit is not cleared, the interrupt is re-entered after the return is executed.

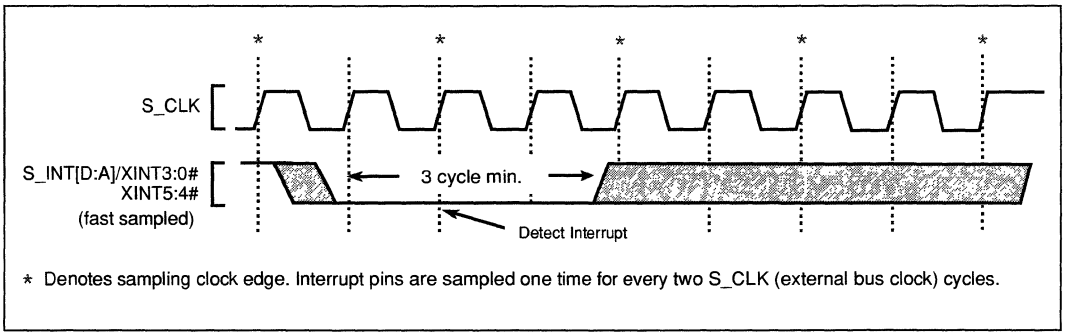
Example 8-4 demonstrates how a level detect interrupt is typically handled. The example assumes that the **ld** from address "timer_0," deactivates the interrupt input.

Example 8-4. Return from a Level-detect Interrupt

```
# Clear level-detect interrupts before return from handler
  lda  IPND_MMR, g1  # Get address of IPND Memory-Mapped Register
  ld   timer_0, g0   # Get timer value and clear TMRO
  lda  0x1000, g2
wait:
  mov   0, g3
  atmod g1, g2, g3
  bbs   0xC, g3, wait
  ret               # Return from handler
```

Interrupt pins are asynchronous inputs. Setup or hold times relative to S_CLK are not needed to ensure proper pin detection. Note in Figure 8-6, which shows how a signal is sampled using fast sampling, that interrupt inputs are sampled once every two S_CLK cycles. For practical purposes, this means that asynchronous interrupting devices must generate an interrupt signal that is asserted for at least three S_CLK cycles. See the <Emphasis>80960RP Intelligent I/O Microprocessor Data Sheet for setup and hold specifications that guarantee detection of the interrupt on particular edges of S_CLK. These specifications are useful in designs that use synchronous logic to generate interrupt signals to the processor. These specification must also be used to calculate the minimum signal width, as shown in Figure 8-6.

Figure 8-6. Interrupt Fast Sampling



* Denotes sampling clock edge. Interrupt pins are sampled one time for every two S_CLK (external bus clock) cycles.

8.2.3 Non-Maskable Interrupt (NMI#)

The **NMI#** pin generates an interrupt for implementation of critical interrupt routines. Error interrupts from the internal peripheral units also come into the 80303 Core through the **NMI#** pin. **NMI#** provides an interrupt that cannot be masked and that has a priority of 31. The interrupt vector for **NMI#** resides in the interrupt table as vector number 248. During initialization, the core caches the vector for **NMI#** on-chip, to reduce **NMI#** latency. The **NMI#** vector is cached in location 0H of internal data RAM.

The core immediately services **NMI#** requests. While servicing an **NMI#**, the core does not respond to any other interrupt requests, even another **NMI#** request. The processor remains in this non-interruptible state until any return-from-interrupt (in supervisor mode) occurs. An interrupt request on the **NMI#** pin is always falling-edge detected. (Note that a return-from-interrupt in user mode does not unblock **NMI#** events and should be avoided by software.)

8.2.4 Timer Interrupts

Each of the two timer units has an associated interrupt to allow the application to accept or post the interrupt request. The timer interrupts are connected directly to the 80303 I/O processor interrupt controller and are posted in the IPND register. These interrupts are set up through the timer control registers described in the *i960® RP Microprocessor User's Manual* (272736), Chapter 19, "Timers".

8.2.5 Software Interrupts

The application program may use the **sysctl** instruction to request interrupt service. The vector that **sysctl** requests is serviced immediately or posted in the interrupt table's pending interrupts section, depending upon the current processor priority and the request's priority. The interrupt controller caches the priority of the highest priority interrupt posted in the interrupt table. The processor cannot request vector 248 (**NMI#**) as a software interrupt.

8.2.6 Interrupt Operation Sequence

The interrupt controller, microcode and core resources handle all stages of interrupt service. Interrupt service is handled in the following stages:

Requesting Interrupt — In the 80303 I/O processor, the programmable on-chip interrupt controller transparently manages all interrupt requests. Interrupts are generated by hardware (external events) or software (the application program). Hardware requests are signaled on the 6-bit external interrupt port (S_INT[D:A]/XINT3:0#, XINT5:4#), the non-maskable interrupt pin (**NMI#**) or the two timer channels. Software interrupts are signaled with the sysctl instruction with post-interrupt message type.

Posting Interrupts — When an interrupt is requested, the interrupt is either serviced immediately or saved for later service, depending on the interrupt's priority. Saving the interrupt for later service is referred to as posting. Once posted, an interrupt becomes a pending interrupt. Hardware and software interrupts are posted differently:

- Hardware interrupts are posted by setting the interrupt's assigned bit in the interrupt pending (IPND) memory mapped register
- Software interrupts are posted by setting the interrupt's assigned bit in the interrupt table's pending priorities and pending interrupts fields

Checking Pending Interrupts — The interrupt controller compares each pending interrupt's priority with the current process priority. When process priority changes, posted interrupts of higher priority are then serviced. Comparing the process priority to posted interrupt priority is handled differently for hardware and software interrupts. Each hardware interrupt is assigned a specific priority when the processor is configured. The priority of all posted hardware interrupts is continually compared to the current process priority. Software interrupts are posted in the interrupt table in external memory. The highest priority posted in this table is also saved in an on-chip software priority register; this register is continually compared to the current process priority.

Servicing Interrupts — When the process priority falls below that of any posted interrupt, the interrupt is serviced. The comparator signals the core to begin a microcode sequence to perform the interrupt context switch and branch to the first instruction of the interrupt routine.

Figure 8-4 illustrates interrupt controller function. For best performance, the interrupt flow for hardware interrupt sources is implemented entirely in hardware.

The comparator only signals the core when a posted interrupt is a higher priority than the process priority. Because the comparator function is implemented in hardware, microcode cycles are never consumed unless an interrupt is serviced.

8.2.7 Setting Up the Interrupt Controller

This section provides an example of setting up the interrupt controller. The following example describes how the interrupt controller can be dynamically configured after initialization.

Example 8-5. sets up the interrupt controller to fetch interrupt vectors from internal data RAM rather than external memory. Initially the IMSK register is masked to allow for setup. A value that selects vector caching is loaded into the ICON register and the IMSK is unmasked.

Example 8-5. Programming the Interrupt Controller for Vector Caching

```
# Example vector caching setup . . .
mov 0x0, g0
mov 0x00006000, g1
st  g0,IMSK# mask, IMSK MMR at 0XFF008504
st  g1,ICON
st  g1,IMSK# fetch vectors from internal RAM
```

8.2.8 Interrupt Service Routines

An interrupt handling procedure performs a specific action that is associated with a particular interrupt procedure pointer. For example, one interrupt handler task might initiate a timer unit request. The interrupt handler procedures can be located anywhere in the non-reserved address space. Since instructions in the i960 processor architecture must be word-aligned, each procedure must begin on a word boundary.

When an interrupt handling procedure is called, the processor allocates a new frame on the interrupt stack and a set of local registers for the procedure. If not already in supervisor mode, the processor always switches to supervisor mode while an interrupt is handled. It also saves the states of the AC and PC registers for the interrupted program.

The interrupt procedure shares the remainder of the execution environment resources (namely the global registers and the address space) with the interrupted program. Thus, interrupt procedures must preserve and restore the state of any resources shared with a non-cooperating program. For example, an interrupt procedure that uses a global register that is not permanently allocated to it should save the register's contents before using the register and restore the contents before returning from the interrupt handler.

To reduce interrupt latency to critical interrupt routines, interrupt handlers may be locked into the instruction cache. See Section 8.6.3.2, "Caching Interrupt Routines and Reserving Register Frames" on page 8-31 for a complete description.

8.2.9 Interrupt Context Switch

When the processor services an interrupt, it automatically saves the interrupted program state or interrupt procedure and calls the interrupt handling procedure associated with the new interrupt request. When the interrupt handler completes, the processor automatically restores the interrupted program state. The method used to service an interrupt depends on the processor state when the interrupt is received.

An *executing-state* interrupt — When the processor is executing a background task and an interrupt request is posted, the interrupt context switch must change stacks to the interrupt stack.

An *interrupted-state* interrupt — When the processor is already executing an interrupt handler, no stack switch is required since the interrupt stack is already in use.

The following subsections describe interrupt handling actions for executing-state and interrupted-state interrupts. In both cases, it is assumed that the interrupt priority is higher than that of the processor and thus is serviced immediately when the processor receives it.

8.2.9.1 Servicing An Interrupt From Executing State

When the processor receives an interrupt while in the executing state (i.e., executing a program, PC.s = 0), it performs the following actions to service the interrupt. This procedure is the same regardless of whether the processor is in user or supervisor mode when the interrupt occurs. The processor:

1. Switches to the interrupt stack (Figure 8-3). The interrupt stack pointer becomes the new stack pointer for the processor.
2. Saves the current PC and AC in an interrupt record on the interrupt stack. The processor also saves the interrupt procedure pointer number.
3. Allocates a new frame on the interrupt stack and loads the new frame pointer (NFP) in global register g15.
4. Sets the state flag in PC to interrupted (PC.s = 1), its execution mode to supervisor and its priority to the priority of the interrupt. Setting the processor's priority to that of the interrupt ensures that lower priority interrupts cannot interrupt the servicing of the current interrupt.
5. Clears the trace enable bit in PC. The interrupt is handled without raising trace faults.
6. Sets the frame return status field pfp[2:0] to 111_2.
7. Performs a call operation as described in the *i960® RP Microprocessor User's Manual* (272736), Chapter 7, "Procedure Calls". The address for the called procedure is specified in the interrupt table for the specified interrupt procedure pointer.

After completing the interrupt procedure, the processor:

1. Copies the arithmetic controls field and the process controls field from the interrupt record into the AC and PC, respectively. It therefore switches to the executing state and restores the trace-enable bit to its value before the interrupt occurred.
2. Deallocates the current stack frame and interrupt record from the interrupt stack and switches to the stack it was using before servicing the interrupt.
3. Performs a return operation as described in the *i960® RP Microprocessor User's Manual* (272736), Chapter 7, "Procedure Calls".
4. Resumes work on the program when all pending interrupts and trace faults are serviced.

8.2.9.2 Servicing An Interrupt From Interrupted State

When the processor receives an interrupt while servicing another interrupt, and the new interrupt has a higher priority than one being serviced, the current interrupt-handler routine is interrupted. Here, the processor performs the same interrupt-servicing action as described in Section 8.2.9.1 to save the state of the interrupted interrupt-handler routine. The interrupt record is saved on the top of the interrupt stack prior to the new frame that is created for use in servicing the new interrupt. See Figure 8-3.

On the return from the current interrupt handler to the previous interrupt handler, the processor de-allocates the current stack frame and interrupt record, and stays on the interrupt stack.

8.3 Theory of Operation

The PCI and Peripheral Interrupt Controller (PPIC) provides the ability to generate interrupts to both the i960 core processor and the PCI bus. The 80303 I/O processor contains a number of peripherals which may generate an interrupt to the i960 core processor. These are:

- DMA Channel 0
- DMA Channel 1
- DMA Channel 2
- Bridge Primary Interface
- Bridge Secondary Interface
- Performance Monitoring Unit
- Primary ATU
- Secondary ATU
- I^2C Bus Interface Unit
- Application Accelerator Unit
- Messaging Unit
- Memory Controller Unit

In addition to the internal peripherals, external devices may also generate interrupts to the i960 core processor. External devices can generate interrupts via the **XINT5:0#** pins and the **NMI#** pin. The PCI and Peripheral Interrupt Controller provides the ability to direct PCI interrupts. The routing logic enables, under software control, the ability to intercept the external secondary PCI interrupts and forward to the primary PCI interrupt pins.

The PCI and Peripheral Interrupt Controller has two functions:

- Internal Peripheral Interrupt Control
- PCI Interrupt Routing

The internal peripheral interrupt control mechanism consolidates a number of interrupt sources for a given peripheral into a single interrupt driven to the i960 core processor. In order to provide the executing software with the knowledge of interrupt source, memory-mapped status registers describe the source of the interrupts. All of the peripheral interrupts are individually enabled from the respective peripheral control registers.

The PCI interrupt routing mechanism allows the host software (or 80303 software) to route PCI interrupts to either the i960 core processor or the **P_INTA#**, **P_INTB#**, **P_INTC#**, and **P_INTD#** output pins. This routing mechanism is controlled through a memory-mapped register accessible from the primary PCI bridge configuration space or the 80303 I/O processor.

8.4 Intel® 80303 I/O Processor Interrupts

The interrupt controller on the i960 core processor has six external interrupt pins, two interrupt lines dedicated to inter-chip communication, and one non-maskable interrupt pin for detecting external interrupt requests. The six external interrupts may be individually mapped to interrupt vectors.

The nine interrupt signals of the i960 core processor have the following definitions and programming options:

XINT7:6# **Internal Interrupt** - These signals generate interrupts to communicate with internal devices on the **80303 I/O processor.**

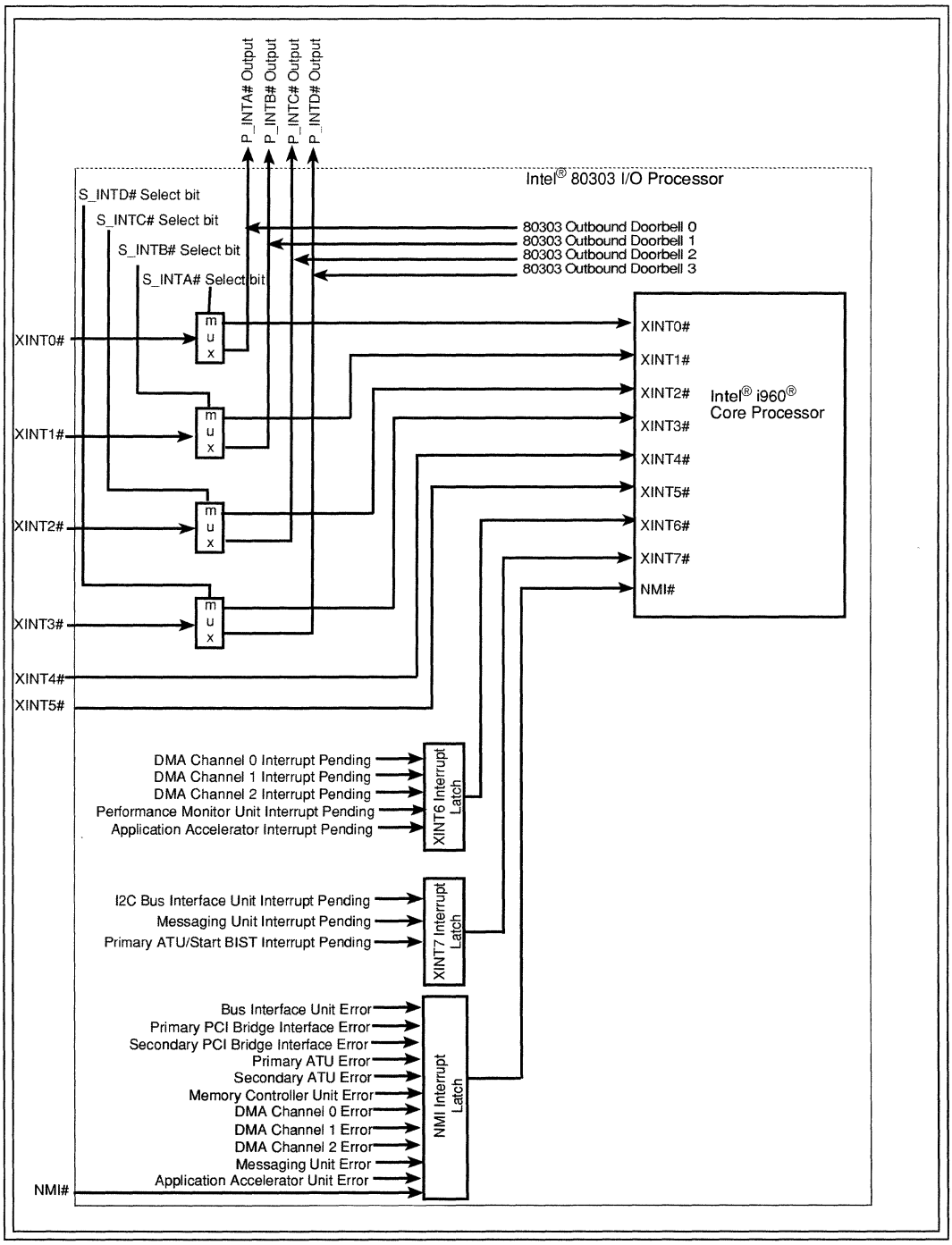
XINT5:0# **External Interrupt (Input)** - These pins cause interrupts to be requested. Interrupts are software configurable for three modes: dedicated, expanded, mixed. Each interrupt can be programmed as an edge-detect input or as a level-detect input. Additionally, a debouncing mode for these interrupts can be selected under program control.

NMI# **Non-Maskable Interrupt (Input)** - Causes a non-maskable interrupt event to occur. NMI is the highest priority interrupt recognized. The **NMI#** input of the i960 core processor is edge-triggered. The external **NMI#** input of the 80303 I/O processor requires a level input. The interrupt latch drives an active low input to the processor as long as a valid interrupt condition exists. A debouncing mode for **NMI#** can be selected under program control. This pin is internally synchronized.

Utilization of the 80303 I/O processor interrupt mechanism relies on the configuration of the i960 core processor interrupt controller and XINT Select bit in the PCI Interrupt Routing Select Register (PIRSR).

The PCI and Peripheral Interrupt Controller provides the connections to the i960 core processor. These connections are shown in Figure 8-7.

Figure 8-7. Interrupt Controller Connections



8.4.1 PCI Interrupt Routing

The four secondary PCI interrupt inputs **S_INT[A:D]#** can be routed to either i960 core processor interrupt inputs or to the primary PCI interrupt output pins **P_INT[A:D]#**.

Routing of interrupt inputs is controlled by the PCI Interrupt Routing Select Register (PIRSR). Table 8-1 summarizes the usage of the bits in the PIRSR.

Table 8-1. PCI Interrupt Routing Summary

PIRSR Select Bit	Bit Value	Description
bit 0	1	**S_INTA#/XINT0#** Input Pin routed to i960 core processor **XINT0#** Input Pin
	0	**S_INTA#/XINT0#** Input Pin routed to **P_INTA#** Output Pin
bit 1	1	**S_INTB#/XINT1#** Input Pin routed to i960 core processor **XINT1#** Input Pin
	0	**S_INTB#/XINT1#** Input Pin routed to **P_INTB#** Output Pin
bit 2	1	**S_INTC#/XINT2#** Input Pin routed to i960 core processor **XINT2#** Input Pin
	0	**S_INTC#/XINT2#** Input Pin routed to **P_INTC#** Output Pin
bit 3	1	**S_INTD#/XINT3#** Input Pin routed to i960 core processor **XINT3#** Input Pin
	0	**S_INTD#/XINT3#** Input Pin routed to **P_INTD#** Output Pin

Note: **XINT0#** through **XINT3#** of the i960 core processor must be programmed to be level sensitive to accommodate PCI interrupts. When any Select bit is set, the logic external to the i960 core processor input must drive an inactive level ('1') to the corresponding interrupt input of the i960 core processor.

8.4.2 Intel® 80303 I/O Processor: External Interrupt Interface

The external interrupt input interface for the 80303 I/O processor consists of the pins shown in Table 8-2.

Table 8-2. Interrupt Input Pin Descriptions

Signal	Description
S_INTA#/XINT0#	Directed to the **P_INTA#** output or the i960 core processor interrupt input **XINT0#**.
S_INTB#/XINT1#	Directed to the **P_INTB#** output or the i960 core processor interrupt input **XINT1#**.
S_INTC#/XINT2#	Directed to the **P_INTC#** output or the i960 core processor interrupt input **XINT2#**.
S_INTD#/XINT3#	Directed to the **P_INTD#** output or the i960 core processor interrupt input **XINT3#**.
XINT4#	Always connected to the i960 core processor interrupt input **XINT4#**.
XINT5#	Always connected to the i960 core processor interrupt input **XINT5#**.
NMI#	Shared with ten internal interrupts. These include all collected error interrupts from the local processor, primary PCI bridge interface, secondary PCI bridge interface, primary ATU, secondary ATU, three DMA channels, application accelerator and the messaging unit. All interrupts are directed to the i960 core processor **NMI#** input. Software reads the NMI Interrupt Status Register to determine the exact source of the interrupt.

8.4.3 Intel® 80303 I/O Processor: Internal Peripheral Interrupt Routing

The **XINT6#**, **XINT7#**, and **NMI#** interrupt inputs on the i960 core processor receive inputs from multiple internal interrupt sources. There is one internal latch before each of these three inputs that provides the necessary muxing of the different interrupt sources. More detail about the exact cause of the interrupt can be determined by reading the status register of the respective peripheral unit.

8.4.3.1 XINT6# Interrupt Sources

The **XINT6#** interrupt of the i960 core processor receives interrupts from the three DMA channels, Performance Monitoring Unit and the Application Accelerator (AA) Unit. Each DMA channel interrupt is either for DMA End of Transfer interrupt or DMA End of Chain interrupt. A Performance Monitoring Unit interrupt implies that at least one of the fourteen programmable event counters and/or the Global Time Stamp Counter has a pending interrupt condition. An AA interrupt implies an End of Chain interrupt or an End of Transfer interrupt.

A valid interrupt from any of these sources sets the bit in the latch and outputs a *level-sensitive* interrupt to the i960 core processor **XINT6#** input. The interrupt latch should continue driving an active low input to the processor interrupt input as long as a one is present in the latch. The **XINT6#** Interrupt Latch is read through the **XINT6#** Interrupt Status Register. The **XINT6#** Interrupt Latch is cleared by clearing the source of the interrupt at the internal peripheral.

The interrupt sources which drive the inputs to the **XINT6#** Interrupt Latch are detailed in Table 8-3.

Table 8-3. XINT6# Interrupt Sources

Unit	Interrupt Condition	Register
DMA Channel 0	End of Chain	Section 19.14.2, "Channel Status Register - CSR" on page 19-28
	End of Transfer	Section 19.14.2, "Channel Status Register - CSR" on page 19-28
DMA Channel 1	End of Chain	Section 19.14.2, "Channel Status Register - CSR" on page 19-28
	End of Transfer	Section 19.14.2, "Channel Status Register - CSR" on page 19-28
DMA Channel 2	End of Chain	Section 19.14.2, "Channel Status Register - CSR" on page 19-28
	End of Transfer	Section 19.14.2, "Channel Status Register - CSR" on page 19-28
Application Accelerator	End of Chain	Section 20.11.2, "Accelerator Status Register - ASR" on page 20-27
	End of Transfer	Section 20.11.2, "Accelerator Status Register - ASR" on page 20-27
Performance Monitor	Counter Overflow	Section 21.6.3, "Event Monitoring Interrupt Status Register (EMISR)" on page 21-25

8.4.3.2 XINT7# Interrupt Sources

The **XINT7#** interrupt on the i960 core processor receives interrupts from the I^2C Bus Interface Unit, the Primary ATU, and the Messaging Unit. A valid interrupt from any of these sources sets the bit in the latch and outputs a *level-sensitive* interrupt to the i960 core processor **XINT7#** input. The interrupt latch should continue driving an active low input to the processor interrupt input as long as a one is present in the latch. The **XINT7#** Interrupt Latch is read through the **XINT7#** Interrupt Status Register. The **XINT7#** Interrupt Latch is cleared by clearing the source of the interrupt at the internal peripheral.

The unit interrupt sources which drive the inputs to the **XINT7#** interrupt latch are detailed in Table 8-4.

Table 8-4. XINT7# Interrupt Sources

Unit	Register	Interrupt Condition
I^2C Bus Interface Unit	I^2C Status Register	Receive Buffer Full
		Transmit Buffer Empty
		Slave Address Detect
		STOP Detected
		Bus Error Detected
		Arbitration Lost Detected
Messaging Unit	Inbound Interrupt Status Register	Index Register Interrupt
		Inbound Post Queue Interrupt
		Inbound Doorbell Interrupt
		Inbound Message 1 Interrupt
		Inbound Message 0 Interrupt
Primary ATU	Primary ATU Interrupt Status Register	ATU BIST Start

8.4.3.3 NMI# Interrupt Sources

The Non-Maskable Interrupt (**NMI#**) on the i960 core processor receives interrupts from the external pin, the primary and secondary ATUs, the primary and secondary bridge interfaces, the local processor, the Messaging Unit, three DMA channels and the application accelerator. Each of these interrupts represent an error condition in the peripheral unit. Refer to the appropriate units for more details.

The NMI Interrupt Latch accepts one interrupt input from each source and the external **NMI#** pin. A valid interrupt from any of these sources sets the bit in the latch and outputs an *edge-triggered* interrupt to the i960 core processor **NMI#** input. The NMI Interrupt Latch is read through the NMI Interrupt Status Register. The NMI interrupt latch is cleared by clearing the source of the interrupt at the internal peripheral.

Note: The i960 core processor **NMI#** input is edge-triggered. The external **NMI#** input of the 80303 I/O processor requires a level input. The interrupt latch drives an active low input to the processor as long as a valid interrupt condition exists. When there are multiple interrupt sources (e.g.: DMA Channel 0 and DMA Channel 1), the NMI latch output transitions from active low to high to account for the interrupt condition that has been cleared. It then outputs another edge-triggered input to the i960 core processor to identify the second interrupt condition that still exists.

Table 8-5 details the unit interrupt sources, which drive the inputs to the NMI interrupt latch.

Table 8-5. NMI# Interrupt Sources

Unit	Register	Error Condition
Primary PCI Bridge Interface	Primary Bridge Interrupt Status Register	PCI Master Parity Error
		PCI Target Abort (target)
		PCI Target Abort (master)
		PCI Master Abort
		P_SERR# Asserted
Secondary PCI Bridge Interface	Secondary Bridge Interrupt Status Register	PCI Master Parity Error
		PCI Target Abort (target)
		PCI Target Abort (master)
		PCI Master Abort
		S_SERR# Asserted
Primary ATU Primary ATU Interrupt Status Register		PCI Master Parity Error
		PCI Target Abort (target)
		PCI Target Abort (master)
		PCI Master Abort
		P_SERR# Detected
		IB Master Abort
		ATU BIST Interrupt
Messaging Unit	Inbound Interrupt Status Register	Outbound Free Queue Full Interrupt
		NMI Doorbell Interrupt
Secondary ATU Secondary ATU Interrupt Status Register		PCI Master Parity Error
		PCI Target Abort (target)
		PCI Target Abort (master)
		PCI Master Abort
		S_SERR# Detected
		IB Master Abort
Bus Interface Unit	BIU Interrupt Status Register	IB Master Abort
DMA Channel 0	Channel Status Register 0	PCI Master Parity Error
		PCI Target Abort (master)
		PCI Master Abort
		IB Master Abort
DMA Channel 1	Channel Status Register 1	PCI Master Parity Error
		PCI Target Abort (master)
		PCI Master Abort
		IB Master Abort
DMA Channel 2	Channel Status Register 2	PCI Master Parity Error
		PCI Target Abort (master)
		PCI Master Abort
		IB Master Abort
Application Accelerator	Accelerator Status Register	IB Master Abort
Memory Controller	Memory Controller Interrupt Status Register	Target-Abort (Single or Multi-bit ECC Errors)

The PCI Interrupt Routing Select Register, XINT6 Interrupt Status Register, XINT7 Interrupt Status Register, and NMI Interrupt Status Register are described in Section 8.7.

8.4.4 PCI Outbound Doorbell Interrupts

The 80303 I/O processor has the capability of generating interrupts on the primary PCI interrupt pins. This is done by setting a bit in the Outbound Doorbell Register within the Messaging Unit. Bits 28 through 31 correspond to PCI interrupts **P_INTA#** through **P_INTD#** respectively. Setting a bit within the register will generate the corresponding PCI interrupt.

Bits 27 through 0 in the Outbound Doorbell Register are all cleared in the default state. When any bit is set, a PCI interrupt is generated. The bit-field in the Address Translation Unit Interrupt Pin Register (ATUIPR) will determine which PCI interrupt (**P_INTA#** through **P_INTD#**) is generated. Refer to *PCI Local Bus Specification*, Revision 2.2 for complete details on the bit-field definition of the ATUIPR.

8.5 Default Status

The interrupt logic is reset by the primary PCI reset signal or through software. Table 8-6 shows the power-up and reset values.

Table 8-6. Default Interrupt Routing and Status Values

Register	Default Value	Description
IPND	Undefined	Software responsible for clearing this register before unmasking any interrupts.
IMSK	0000 0000H	All interrupts masked.
IMAP2:0	Initial Image in Control Table	Set to user's values.
ICON	Initial Image in Control Table	Set to user's values.
PIRSR	000000H	All interrupts **IRQ[23:0]** routed to the Intel® i960® core processor.
NMI Interrupt Latch	0000 00000_2	All interrupts cleared
XINT7 Interrupt Latch	00000_2	All interrupts cleared
XINT6 Interrupt Latch	00H	All interrupts cleared
NMI Interrupt Status Register	0000 0000H	No interrupts set
XINT7 Interrupt Status Register	0000 0000H	No interrupts set
XINT6 Interrupt Status Register	0000 0000H	No interrupts set

8.5.1 Interrupt Controller Register Access Requirements

A load instruction that accesses the IPND, IMSK, IMAP2:0 or ICON register has a latency of one internal processor cycle. A store access to an interrupt register is synchronous with respect to the next instruction; that is, the operation completes fully and all state changes take effect before the next instruction begins execution.

Interrupts can be enabled and disabled quickly by the **intdis** and **inten** instructions, which take four cycles each to execute. **intctl** takes a few cycles longer because it returns the previous interrupt enable value. See *i960® RM/RN I/O Processor Developer's Manual* (273158) Chapter 6, "Instruction Set References" for more information on these instructions.

8.6 Performance Requirements

The interrupt routing logic shall accept the routing control value written to the PIRSR register and accept the changes within one clock after the write has completed. When the processor reads the PIRSR register, the value shall be returned immediately, effectively zero wait states.

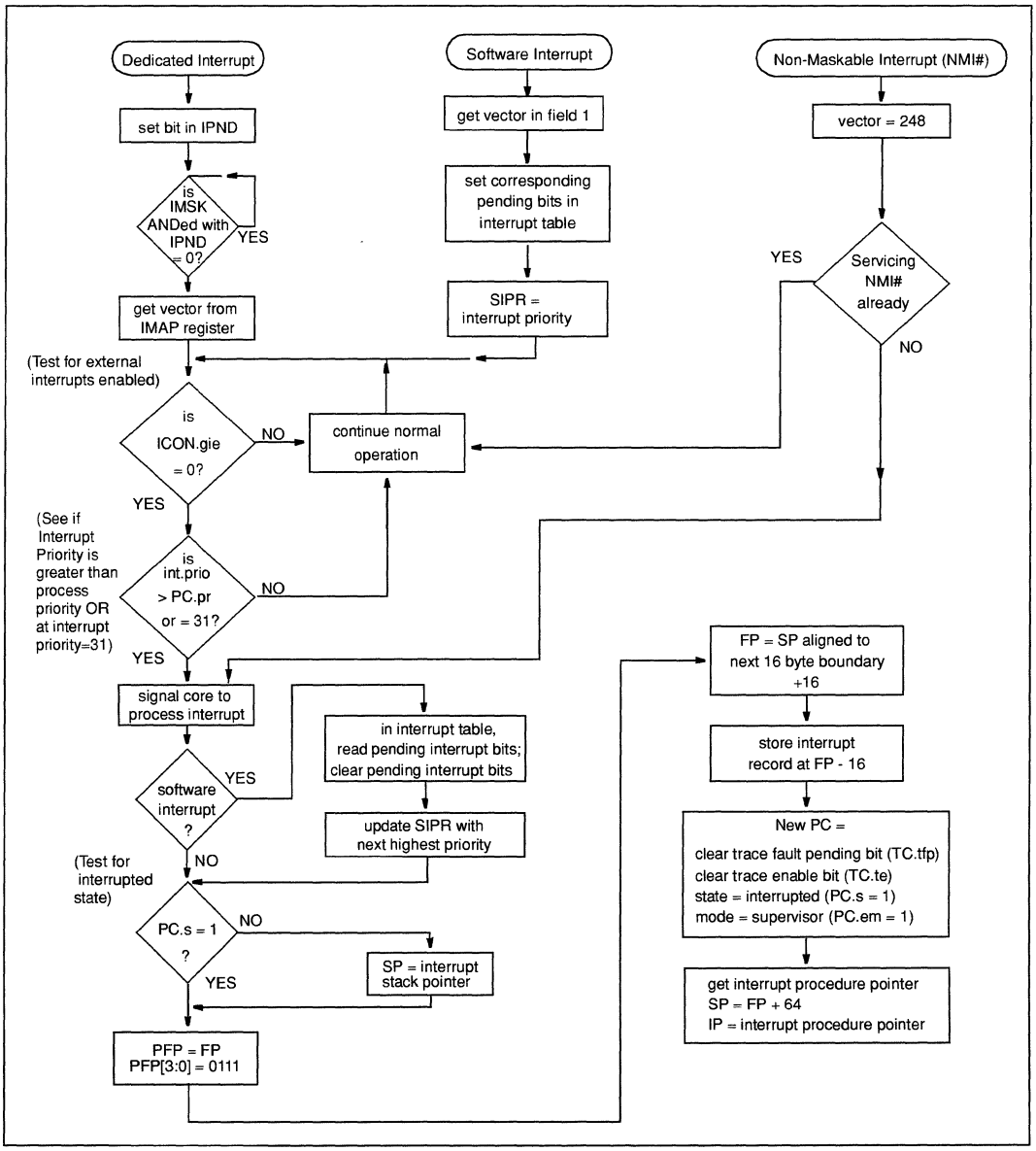
When the processor reads the NISR, X4ISR, X5ISR, X6ISR and X7ISR registers, the value shall be returned immediately, effectively zero wait states.

The logic shall introduce no more than one clock delay when the interrupt is recognized on the input of the logic until the signal is driven either to the i960 core processor interrupt controller or the PCI output interrupt pins.

8.6.1 Optimizing Interrupt Performance

Figure 8-8 depicts the path from interrupt source to interrupt service routine. This section discusses interrupt performance in general and suggests techniques the application can use to get the best interrupt performance.

Figure 8-8. Interrupt Service Flowchart



8.6.2 Interrupt Service Latency

The established measure of interrupt performance is the time required to perform an interrupt task switch, which is known as *interrupt service latency*. Latency is the time measured between interrupt source activation and execution of the first instruction for the accompanying interrupt-handling procedure.

Interrupt latency depends on interrupt controller configuration and the instruction being executed at the time of the interrupt. The processor also has a number of cache options that reduce interrupt latency. In the discussion that follows, interrupt latency is expressed as a number of bus clock cycles.

8.6.3 Features to Improve Interrupt Performance

The 80303 I/O processor employs four methods to reduce interrupt latency:

- Caching interrupt vectors on-chip
- Caching of interrupt handling procedure code
- Reserving register frames in the local register cache
- Caching the interrupt stack in the data cache

8.6.3.1 Vector Caching Option

To reduce interrupt latency, the 80303 I/O processor caches some interrupt table vector entries in internal data RAM. When the vector cache option is enabled and an interrupt request has a cached vector to be serviced, the controller fetches the associated vector from internal RAM rather than from the interrupt table in memory.

Interrupts with a vector number with the four least-significant bits equal to 0010_2 can be cached. Vectors that can be cached coincide with the vector numbers selected with the mapping registers and assigned to dedicated-mode inputs. The vector caching option is selected when programming the ICON register; software must explicitly store the vector entries in internal RAM.

Since the internal RAM is mapped to the address space directly, this operation can be performed using the core's store instructions. Table 8-7 shows the required vector mapping to specific locations in internal RAM. For example, the vector entry for vector number 18 must be stored at RAM location 04H, and so on.

The **NMI#** vector is also shown in Table 8-7. This vector is always cached in internal data RAM at location 0000H. The processor automatically loads this location at initialization with the value of vector number 248 in the interrupt table.

Table 8-7. Location of Cached Vectors in Internal RAM

Vector Number (Binary)	Vector Number (Decimal)	Internal RAM Address
(NMI#)	248	0000H
$0001\ 0010_2$	18	0004H
$0010\ 0010_2$	34	0008H
$0011\ 0010_2$	50	000CH
$0100\ 0010_2$	66	0010H
$0101\ 0010_2$	82	0014H
$0110\ 0010_2$	98	0018H
$0111\ 0010_2$	114	001CH
$1000\ 0010_2$	130	0020H
$1001\ 0010_2$	146	0024H
$1010\ 0010_2$	162	0028H
$1011\ 0010_2$	178	002CH
$1100\ 0010_2$	194	0030H
$1101\ 0010_2$	210	0034H
$1110\ 0010_2$	226	0038H
$1111\ 0010_2$	242	003CH

8.6.3.2 Caching Interrupt Routines and Reserving Register Frames

The time required to fetch the first instructions of an interrupt-handling procedure affects interrupt response time and throughput. The controller reduces this fetch time by caching interrupt procedures or portions of procedures in the 80303 I/O processor's instruction cache.

To decrease interrupt latency for high priority interrupts (priority 28 and above), software can limit the number of frames in the local register cache available to code running at a lower priority (priority 27 and below). This ensures that some number of free frames are available to high-priority interrupt service routines. See *i960® RM/RN I/O Processor Developer's Manual* (273158) "Local Register Cache, for more details.

8.6.3.3 Caching the Interrupt Stack

By locating the interrupt stack in memory that can be cached by the data cache, the performance of interrupt returns can be improved. This is because accesses to the interrupt record by the interrupt return can be satisfied by the data cache. See *i960® RM/RN I/O Processor Developer's Manual* (273158), "Programming the Physical Memory Attributes (PMCON Registers)" for details on how to enable data caching for portions of memory.

8.6.4 Base Interrupt Latency

In many applications, the processor's instruction mix and cache configuration are known sufficiently well to use typical interrupt latency in calculations of overall system performance. For example, a timer interrupt may frequently trigger a task switch in a multi-tasking kernel. Base interrupt latency assumes the following:

- Single-cycle RISC instruction is interrupted
- Frame flush does not occur
- Bus queue is empty
- Cached interrupt handler
- No interaction of faults and interrupts (i.e., a stable system)

Table 8-13 shows the base latencies for all interrupt types, with varying vector caching options.

Table 8-8. Base Interrupt Latency

Interrupt Type	Vector Caching Enabled	Typical Latency (Bus Clocks)
NMI#	Yes	30
XINT[5:4]#, TINT1:0[1]	Yes	34
	No	40+a[2]
XINT[7:6]#, XINT[3:0]#	Yes	35
	No	41+a[2]
Software	Yes	68
	No	69+a[2]

NOTES:

1. TINT - Timer interrupts
2. a = MAX (0,N - 7), where "N" is the number of bus cycles needed to perform a word load.

8.6.5 Maximum Interrupt Latency

In real-time applications, worst-case interrupt latency must be considered for critical handling of external events. For example, an interrupt from a mechanical subsystem may need service to calculate servo loop parameters to maintain directional control. Determining worst-case latency depends on knowledge of the processor's instruction mix and operating environment as well as the interrupt controller configuration. Excluding certain very long, uninterruptible instructions from critical sections of code reduces worst-case interrupt latency to levels approaching the base latency.

The following tables present worst case interrupt latencies based on possible execution of **divo** (r15 destination), **divo** (r3 destination), **calls** or **flushreg** instructions or software interrupt detection. The assumptions for these tables are the same as for Table 8-13, except for instruction execution.

Table 8-9. Worst-Case Interrupt Latency Controlled by divo to Destination r15

Interrupt Type	Vector Caching Enabled	Worst Latency (Bus Clocks)
NMI#	Yes	43
XINT[5:4]#, TINT1:0	Yes	45
	No	45+a[1]
XINT[7:6]# XINT[3:0]#	Yes	46
	No	46+a[1]

NOTES:

1. a = MAX (0,N - 11), where "N" is the number of bus cycles needed to perform a word load.

Table 8-10. Worst-Case Interrupt Latency Controlled by divo to Destination r3

Interrupt Type	Vector Caching Enabled	Worst Latency (Bus Clocks)
NMI#	Yes	60
XINT[5:4]#, TINT1:0	Yes	65
	No	72+a[1]
XINT[7:6]# XINT[3:0]#	Yes	66
	No	73+a[1]

NOTES:

1. a = MAX (0,N - 7), where "N" is the number of bus cycles needed to perform a word load.

Table 8-11. Worst-Case Interrupt Latency Controlled by Calls

Interrupt Type	Vector Caching Enabled	Worst Latency (Bus Clocks)
NMI#	Yes	54+a[1]
XINT[5:4]#, TINT1:0	Yes	58+a[1]
	No	66+a+b[1]
XINT[7:6]# XINT[3:0]#	Yes	59+a[1]
	No	67+a+b[1]

NOTES:

1. a = MAX (0,N - 4)
   b = MAX (0,N - 7)
   where "N" is the number of bus cycles needed to perform a word load.

Table 8-12. Worst-Case Interrupt Latency When Delivering a Software Interrupt

Interrupt Type	Vector Caching Enabled	Worst Latency (Bus Clocks)
NMI#	Yes	97
XINT[5:4]#, TINT1:0	Yes	99
	No	107+a[1]
XINT[7:6]# XINT[3:0]#	Yes	100
	No	108+a[1]

NOTES:

1. a = MAX (0,N - 7), where "N" is the number of bus cycles needed to perform a word load.

Table 8-13. Worst-Case Interrupt Latency Controlled by flushreg of One Stack Frame

Interrupt Type	Vector Caching Enabled	Worst Latency (Bus Clocks)
NMI#	Yes	78+a+b[1]
XINT[5:4]#, TINT1:0	Yes	82+a+b[1]
	No	89+a+b+c[1]
XINT[7:6]# XINT[3:0]#	Yes	83+a+b[1]
	No	90+a+b+c[1]

NOTES:

1. a = MAX (0, M - 15)
   b = MAX (0, M - 28)
   c = MAX (0, N - 7)
   where "M" is the number of bus cycles needed to perform a quad word store and "N" is the number of bus cycles needed to perform a word load. Interrupt latency increases rapidly as the number of flushed stack frames increases.

8.6.6 Avoiding Certain Destinations for MDU Operations

Typically, when delivering an interrupt, the processor attempts to push the first four local registers (pfp, sp, rip, and r3) onto the local register cache as early as possible. Because of register-interlock, this operation is stalled until previous instructions return their results to these registers. In most cases, this is not a problem; however, in the case of instructions performed by the Multiply/Divide Unit (**divo**, **divi**, **ediv**, **modi**, **remo**, and **remi**), the processor could be stalled for many cycles waiting for the result and unable to proceed to the next step of interrupt delivery.

Interrupt latency can be improved by avoiding the first four local registers as the destination for a Multiply/Divide Unit operation. (Registers pfp, sp, and rip should be avoided anyway for general operations as these are used for procedure linking.)

8.6.7 Secondary PCI to Upstream Interrupt Routing Latency

The interrupt routing logic accepts the changes to the routing control value written to the PIRSR register one clock after the write has completed. There is a one clock delay from the time that the interrupt is recognized on the input of the MUX until the signal is driven either to the 80303 core interrupt controller or directly upstream.

8.7 Register Definitions

All ten registers are visible as 80303 I/O processor memory mapped registers and can be accessed through the internal memory bus. Each is a 32-bit register and is memory-mapped in the 80303 I/O processor memory space. The memory-mapped addresses of the interrupt control registers are found in Appendix C, "Peripheral Memory-Mapped Registers."

There are four control and status registers for the PCI and Peripheral Interrupt Controller:

- PCI Interrupt Routing Select Register
- XINT6 Interrupt Status Register
- XINT7 Interrupt Status Register
- NMI Interrupt Status Register

The PCI Interrupt Routing Select Register is accessible from the internal memory bus and also during PCI configuration cycles through the PCI configuration register space (function #0). See Chapter 15, "PCI Address Translation Unit" for additional information regarding the PCI configuration cycles that access the PCI Interrupt Routing Select Register. The programmer's interface to the interrupt controller is through ten memory-mapped control registers. Table 8-14 describes these registers.

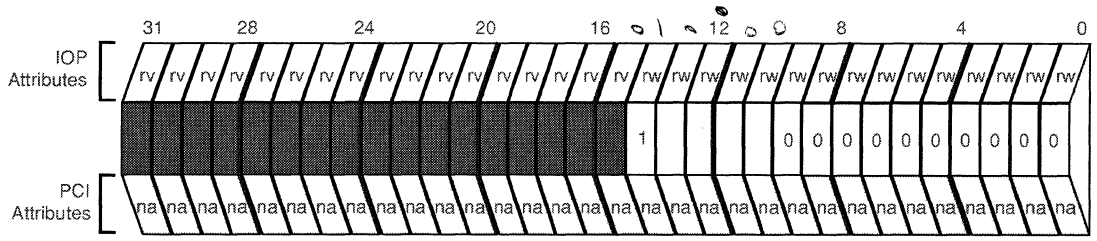
Table 8-14. Interrupt Control Registers Addresses

Register Name	Description	Address
ICON	Interrupt Control Register	FF00 8510H
IMAP0	Interrupt Map Register 0	FF00 8520H
IMAP1	Interrupt Map Register 1	FF00 8524H
IMAP2	Interrupt Map Register 2	FF00 8528H
IPND	Interrupt Pending Register	FF00 8500H
IMSK	Interrupt Mask Register	FF00 8504H
PIRSR	PCI Interrupt Routing Select Register	0000 1050H
X6ISR	XINT6 Interrupt Status Register	0000 1708H
X7ISR	XINT7 Interrupt Status Register	0000 1704H
NISR	NMI Interrupt Status Register	0000 1700H

8.7.1 Interrupt Control Register (ICON)

The ICON register is a 32-bit memory-mapped control register, that sets up the interrupt controller. Software can manipulate this register using the load/store type instructions. The ICON register is also automatically loaded at initialization from the control table in external memory. Table 8-15 describes the ICON register.

Table 8-15. Interrupt Control (ICON) Register



Intel® i960® Core internal bus address
FF00 8510H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set

RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:15	Default Value loaded from image in Control Table	Reserved.
14		This bit must be set (1).
13		Vector Cache Enable - determines whether interrupt table vector entries are fetched from the interrupt table (bit clear) or from internal data RAM (bit set). Only vectors with the four least-significant bits equal to 0010_2 may be cached in internal data RAM.
12:11		Mask Operation Field - determines the operation the core performs on the mask register when a hardware-generated interrupt is serviced. On an interrupt, the value in IMSK is copied to r3. IMSK is then either left unchanged (00) or cleared (01). IMSK is never cleared for **NMI#** or software interrupts.
10		Global Interrupts Enable - globally enables or disables the external interrupt pins and timer unit inputs. It does not affect the **NMI#** pin. This bit performs the same function as clearing the IMSK register. This bit is also changed indirectly by the instructions **inten**, **intdis**, **intctl**.
9:0		These bits must be cleared (0).

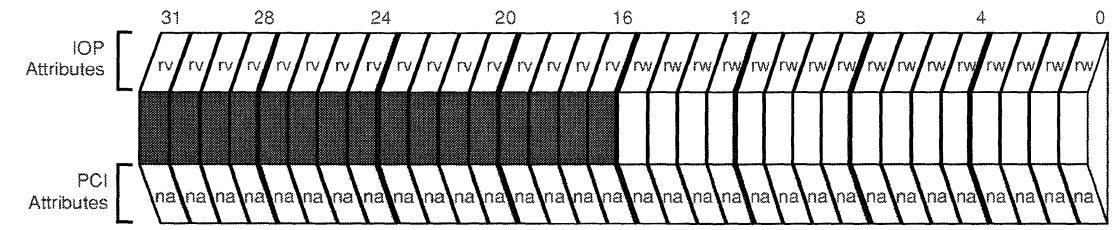
8.7.2 Interrupt Mapping Registers (IMAP0-IMAP2)

The IMAP registers (Table 8-16, Table 8-17 and Table 8-18) are three 32-bit registers (IMAP0 through IMAP2). These registers are used to program the vector number associated with the interrupt source. IMAP0 and IMAP1 contain mapping information for the external interrupt pins (four bits per pin). IMAP2 contains mapping information for the timer-interrupt inputs (four bits per interrupt).

Each set of four bits contains a vector number's four most-significant bits; the four least-significant bits are always 0010_2. In other words; each source can be programmed for a vector number of PPPP 0010_2, where "P" indicates a programmable bit. For example, IMAP0 bits 4 through 7 contain mapping information for the XINT1 pin. When these bits are set to 0110_2, the pin is mapped to vector number 0110 0010_2 (or vector number 98).

Software can access the mapping registers using load/store type instructions. The mapping registers are also automatically loaded at initialization from the control table in external memory.

Table 8-16. Interrupt Map Register 0 (IMAP0)



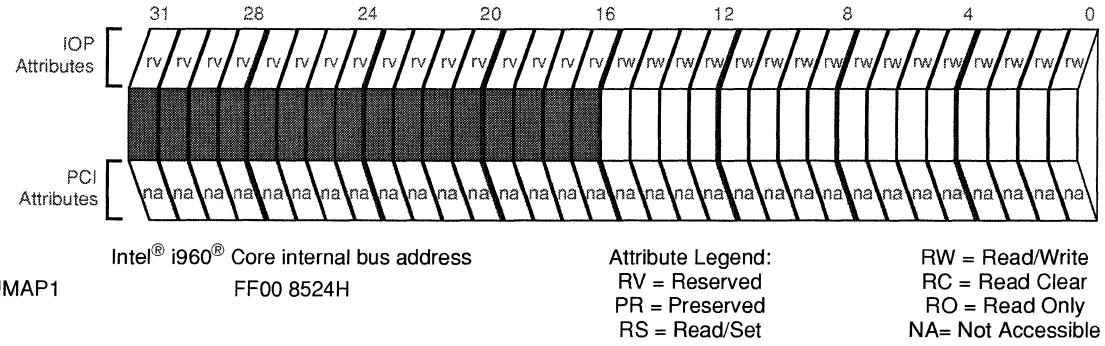
Intel® i960® Core internal bus address
IMAP0 FF00 8520H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set

RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:16	Default Value loaded from image in Control Table	Reserved (initialize to 0)
15:12		External Interrupt 3 Field - IMAP0.x3
11:8		External Interrupt 2 Field - IMAP0.x2
7:4		External Interrupt 1 Field - IMAP0.x1
3:0		External Interrupt 0 Field - IMAP0.x0

Table 8-17. Interrupt Map Register 1 (IMAP1)



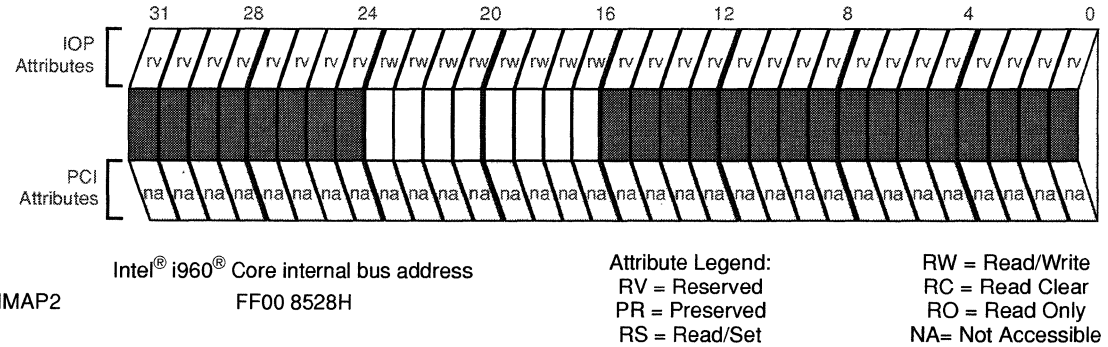
Intel® i960® Core internal bus address
IMAP1 FF00 8524H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set

RW = Read/Write
RC = Read Clear
RO = Read Only
NA= Not Accessible

Bit	Default	Description
31:16	Default Value loaded from image in Control Table	Reserved (initialize to 0)
15:12		External Interrupt 7 Field - IMAP1.x7
11:8		External Interrupt 6 Field - IMAP1.x6
9:4		External Interrupt 5 Field - IMAP1.x5
3:0		External Interrupt 4 Field - IMAP1.x4

Table 8-18. Interrupt Map Register 2 (IMAP2)



Intel® i960® Core internal bus address
IMAP2 FF00 8528H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set

RW = Read/Write
RC = Read Clear
RO = Read Only
NA= Not Accessible

Bit	Default	Description
31:24	Default Value loaded from image in Control Table	Reserved (initialize to 0)
23:20		Timer Interrupt 1 Field - IMAP2.t1
19:16		Timer Interrupt 0 Field - IMAP2.t0
15:0		Reserved (initialize to 0)

8.7.3 Interrupt Pending (IPND) and Interrupt Mask (IMSK) Registers

The IPND and IMSK (Table 8-19 and Table 8-20) registers are both memory-mapped registers. Bits 0 through 7 of these registers are associated with the external interrupt pins (**XINT0#** - **XINT7#**) and bits 12 and 13 are associated with the timer-interrupt inputs (TMR0 and TMR1). All other bits are reserved and should be cleared at initialization.

The IPND register posts interrupts originating from the eight external dedicated sources and the two timer sources. Asserting one of these inputs latches a 1 into its associated bit in the IPND register. The mask register provides a mechanism for masking individual bits in the IPND register. An interrupt source is disabled when its associated mask bit is cleared (0).

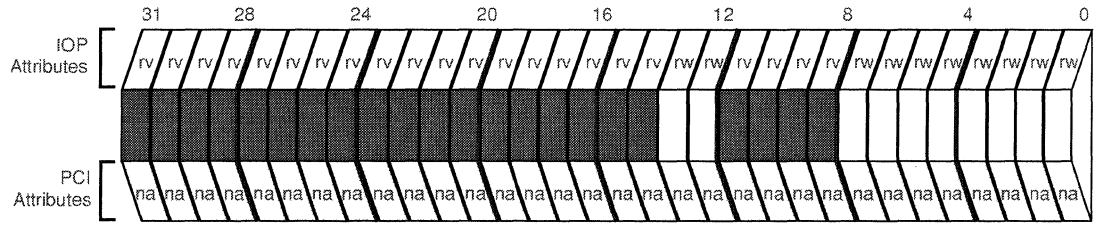
When delivering a hardware interrupt, the interrupt controller conditionally clears IMSK based on the value of the ICON.mo bit. Note that IMSK is never cleared for **NMI#** or software interrupt.

Although software can read and write IPND and IMSK using any memory-format instruction, it is recommended that a read-modify-write operation using the atomic-modify instruction (**atmod**) be used for reading and writing these registers. Executing an **atmod** on one of these registers causes the interrupt controller to perform regular interrupt processing (including using or automatically updating IPND and IMSK) either before or after, but, not during the read-modify-write operation on that register. This requirement ensures that modifications to IPND and IMSK take effect cleanly, completely, and at a well-defined point. Note that the processor does not assert the LOCK# pin externally when executing an atomic instruction to IPND and IMSK.

When the processor core handles a pending interrupt, it attempts to clear the bit that is latched for that interrupt in the IPND register before it begins servicing the interrupt. If that bit is associated with an interrupt source that is programmed for level detection and the true level is still present, the bit remains set. Because of this, the interrupt routine for a level-detected interrupt should clear the external interrupt source and explicitly clear the IPND bit before return from the handler is executed.

An alternative method of posting interrupts in the IPND register, other than through the external interrupt pins, is to set bits in the register directly using an **atmod** instruction. This operation has the same effect as requesting an interrupt through the external interrupt pins.

Table 8-19. Interrupt Pending (IPND) Register



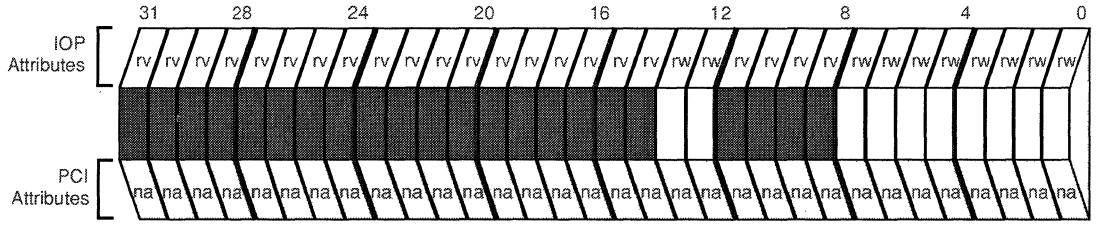
Intel® i960® Core internal bus address
FF00 8500H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set

RW = Read/Write
RC = Read Clear
RO = Read Only
NA= Not Accessible

Bit	Default	Description
31:14	0	Reserved
13:12	x	Timer Interrupt Pending Bits - IPND.tip 0 = No Interrupt 1 = Pending Interrupt
11:8	0000_2	Reserved (initialize to 0)
7:0	x	External Interrupt Pending Bits - IPND.xip 0 = No Interrupt 1 = Pending Interrupt

Table 8-20. Interrupt Mask (IMSK) Register



Intel® i960® Core internal bus address
FF00 8504H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set

RW = Read/Write
RC = Read Clear
RO = Read Only
NA= Not Accessible

Bit	Default	Description
31:14	0	Reserved
13:12	00_2	Timer Interrupt Pending Bits - IMSK.xim 0 = Masked 1 = Not Masked
11:8	0000_2	Reserved (initialize to 0)
7:0	00H	External Interrupt Mask Bits - IMSK.tim 0 = Masked 1 = Not Masked

8.7.4 PCI Interrupt Routing Select Register - PIRSR

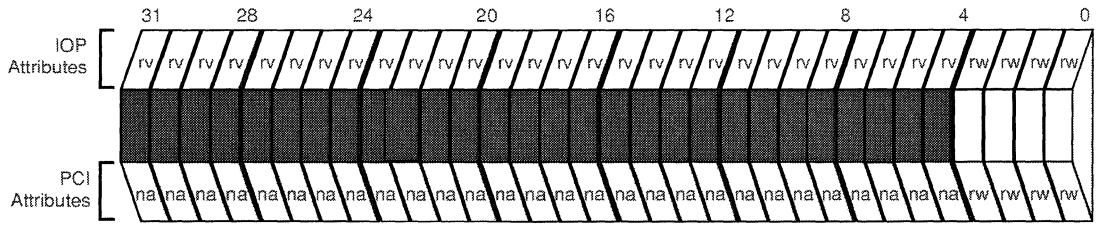
The PCI Interrupt Routing Select Register (PIRSR) determines the routing of the external input pins. The input pins consist of four secondary PCI interrupt inputs which are routed to either the primary PCI interrupts or i960 core processor interrupts. The PCI interrupt pins are defined as "level sensitive," asserted low. The assertion and deassertion of the interrupt pins are synchronous to the PCI or processor clock.

When any of **S_INT[A:D]#** are routed upstream the corresponding i960 core processor inputs (**XINT3:0#**) must be set inactive (level '1').

When any of the **XINT4#** or **XINT5#** latch interrupts are steered away from the i960 core processor, the logic external to the latch input must drive an inactive level ('1') to the corresponding interrupt latch input.

Table 8-21 shows the bit definitions for programming the PCI Interrupt Routing Select Register.

Table 8-21. PCI Interrupt Routing Select Register - PIRSR



Intel® i960® Core internal bus address
0000 1050H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set

RW = Read/Write
RC = Read Clear
RO = Read Only
NA= Not Accessible

Bit	Default	Description
31:4	0	Reserved
3	0	**S_INTD#** Select Bit - PIRSR.seld 0 = Interrupt routed to P_INTD# pin 1 = Interrupt routed to i960 core processor interrupt controller input (**XINT3#**)
2	0	**S_INTC#** Select Bit - PIRSR.selc 0 = Interrupt routed to P_INTC# pin 1 = Interrupt routed to i960 core processor interrupt controller input (**XINT2#**)
1	0	**S_INTB#** Select Bit - PIRSR.selb 0 = Interrupt routed to P_INTB# pin 1 = Interrupt routed to i960 core processor interrupt controller input (**XINT1#**)
0	0	**S_INTA#** Select Bit - PIRSR.sela 0 = Interrupt routed to P_INTA# pin 1 = Interrupt routed to i960 core processor interrupt controller input (**XINT0#**)

8.7.5 XINT6 Interrupt Status Register - X6ISR

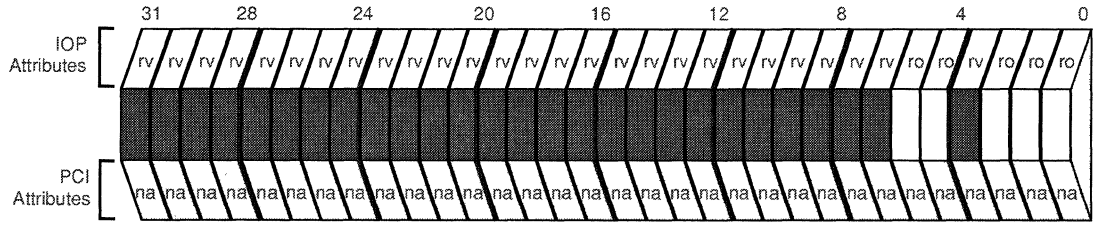
The XINT6 Interrupt Status Register (X6ISR) contains the current pending **XINT6#** interrupts. The source of the **XINT6#** interrupt can be the internal peripheral devices connected through the **XINT6#** Interrupt Latch. The interrupts which can be generated on the **XINT6#** input are detailed in Section 8.4.3, "Intel® 80303 I/O Processor: Internal Peripheral Interrupt Routing" on page 8-24.

The X6ISR is used by application software to determine the source of an interrupt on the **XINT6#** input and to clear that interrupt. All bits within this register are defined as read only. The bits within this register are cleared when the source of the interrupt (status register source shown in Table 8-3) are cleared. The X6ISR reflects the current state of the input to the **XINT6#** Interrupt Latch.

Due to the asynchronous nature of the 80303 I/O processor peripheral units, multiple interrupts can be active when application software reads the X6ISR register. Application software must handle such conditions appropriately. In addition, application software may subsequently read the X6ISR register to determine if additional interrupts have occurred during interrupt processing for the prior interrupts. All interrupts from the X6ISR register will be at the same priority level within the i960 core processor.

Table 8-22 details the bit definition of the X6ISR.

Table 8-22. XINT6 Interrupt Status Register- X6ISR



Intel® i960® Core internal bus address
0000 1708H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set

RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:06	0	Reserved
05	0	Application Accelerator Interrupt Pending - when set, an end of chain condition has been signaled by the Application Accelerator. When clear, no interrupt condition exists.
04	0_2	Performance Monitor Interrupt Pending - when set, at least one of the programmable event counters and/or the Global Time Stamp Counter contains an overflow condition. Application software identifies the counter by reading the Event Monitoring Interrupt Status register (EMISR). When clear, no interrupt condition exists.
03	0_2	Reserved.
02	0_2	DMA Channel 2 Interrupt Pending - when set, an end of chain of channel active condition has been signaled by DMA channel 2. When clear, no interrupt condition exists.
01	0_2	DMA Channel 1 Interrupt Pending - when set, an end of chain of channel active condition has been signaled by DMA channel 1. When clear, no interrupt condition exists.
00	0_2	DMA Channel 0 Interrupt Pending - when set, an end of chain of channel active condition has been signaled by DMA channel 0. When clear, no interrupt condition exists.

8.7.6 XINT7 Interrupt Status Register- X7ISR

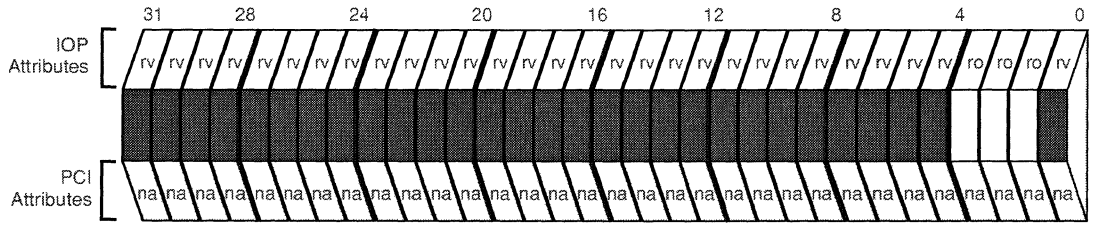
The XINT7 Interrupt Status Register (X7ISR) contains the current pending XINT7 interrupts. The source of the XINT7 interrupt can be the internal peripheral devices connected through the XINT7 Interrupt Latch. The interrupts which can be generated on the **XINT7#** input are detailed in Section 8.4.3, "Intel® 80303 I/O Processor: Internal Peripheral Interrupt Routing" on page 8-24.

The X7ISR is used by application software to determine the source of an interrupt on the **XINT7#** input and to clear that interrupt. All bits within this register are defined as read only. The bits within this register are cleared when the source of the interrupt (status register source shown in Table 8-4) are cleared. The X7ISR reflects the current state of the input to the XINT7 Interrupt Latch.

Due to the asynchronous nature of the 80303 I/O processor peripheral units, multiple interrupts can be active when application software reads the X7ISR register. Application software must handle these multiple interrupt conditions appropriately. In addition, application software may subsequently read the X7ISR register to determine if additional interrupts have occurred during interrupt processing for the prior interrupts. All interrupts from the X7ISR register will be at the same priority level within the i960 core processor.

Table 8-23 details the bit definition of the X7ISR.

Table 8-23. XINT7 Interrupt Status Register- X7ISR



Intel® i960® Core internal bus address
0000 1704H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set

RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:04	0	Reserved
03	0_2	Primary ATU/Start BIST Interrupt Pending - when set, the host processor has set the start BIST request in the ATUBISTR register. When clear, no start BIST interrupt is pending.
02	0_2	Messaging Unit Interrupt Pending - when set, an interrupt from the Messaging Unit is pending. When clear, no interrupt is pending.
01	0_2	I^2C Interrupt Pending - when set, an interrupt is from the I^2C Bus Interface Unit is pending. When clear, no interrupt is pending.
00	0_2	Reserved

8.7.7 NMI Interrupt Status Register - NISR

The NMI Interrupt Status Register (NISR) contains the current pending NMI interrupts. The source of the NMI interrupt can be the internal peripheral devices connected through the NMI Interrupt Latch or the external **NMI#** input pin. The interrupts which can be generated on the **NMI#** input are detailed in Section 8.4.3, "Intel® 80303 I/O Processor: Internal Peripheral Interrupt Routing" on page 8-24.

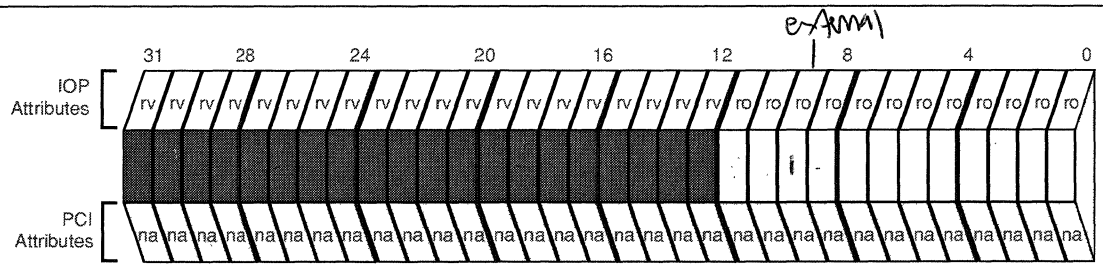
The NMI Interrupt Status Register is used by application software to determine the source of an interrupt on the **NMI#** input and to clear that interrupt. All of the bits within the NISR are read only. The bits within this register are cleared when the source of the interrupt (status register source shown in Table 8-5) are cleared. The NISR reflects the current state of the input to the NMI Interrupt Latch.

Due to the asynchronous nature of the 80303 I/O processor peripheral units, multiple interrupts can be active when the application software reads the NISR register. Application software must handle these multiple interrupt conditions appropriately. In addition, application software may subsequently read the NISR register to determine if additional interrupts have occurred during interrupt processing for the prior interrupts. All interrupts from the NISR register will be at the same priority level within the i960 core processor.

Note: Although the **NMI#** input of the i960 core processor is edge triggered, the external **NMI#** input of the 80303 I/O processor requires a level input and must be latched external to the 80303 I/O processor.

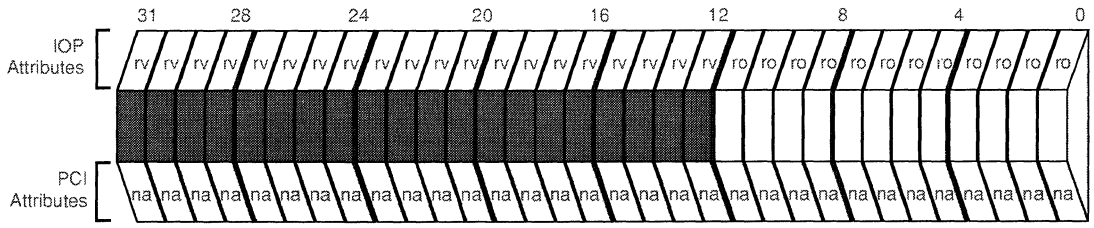
Table 8-24 details the bit definitions for the NMI interrupt status register.

Table 8-24. NMI Interrupt Status Register- NISR (Sheet 1 of 2)



Bit	Default	Description
31:12	0	Reserved
11	0	Bus Interface Unit Error - when set, a PCI or internal bus error condition exists within the BIU. When clear, no error condition exists.
10	0_2	Application Accelerator Unit Error - when set, an internal bus error condition exists within AA unit. When clear, no error exists.
09	0_2	External **NMI#** Interrupt - when set, an interrupt is pending on the external **NMI#** input. When clear, no interrupt exists.
08	0_2	Messaging Unit Interrupt - when set, an NMI interrupt or error exists in the Messaging Unit. When clear, no error exits.

Table 8-24. NMI Interrupt Status Register- NISR (Sheet 2 of 2)



Intel® i960® Core internal bus address
0000 1700H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set

RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
07	0_2	DMA Channel 2 Error - when set, a PCI or internal bus error condition exists within DMA channel. When clear, no error exists.
06	0_2	DMA Channel 1 Error - when set, a PCI or internal bus error condition exists within DMA channel. When clear, no error exists.
05	0_2	DMA Channel 0 Error - when set, a PCI or internal bus error condition exists within DMA channel. When clear, no error exists.
04	0_2	Secondary Bridge Error - when set, a PCI error condition exists within the secondary interface of the bridge. When clear, no error exists.
03	0_2	Primary Bridge Interface Error - when set, a PCI error condition exists within the primary interface of the bridge. When clear, no error exists.
02	0_2	Secondary ATU Error - when set, a PCI or internal bus error condition exists within the secondary ATU. When clear, no error exists.
01	0_2	Primary ATU Error - when set, a PCI or internal bus error condition exists within the primary ATU. When clear, no error exists.
00	0_2	Memory Controller Error - when set, an error condition exists within the MCU. The bit indicates one of the following conditions: • A single-bit correctable or uncorrectable ECC error. • A multi-bit correctable or uncorrectable ECC error. When clear, no error exists.

Faults 9

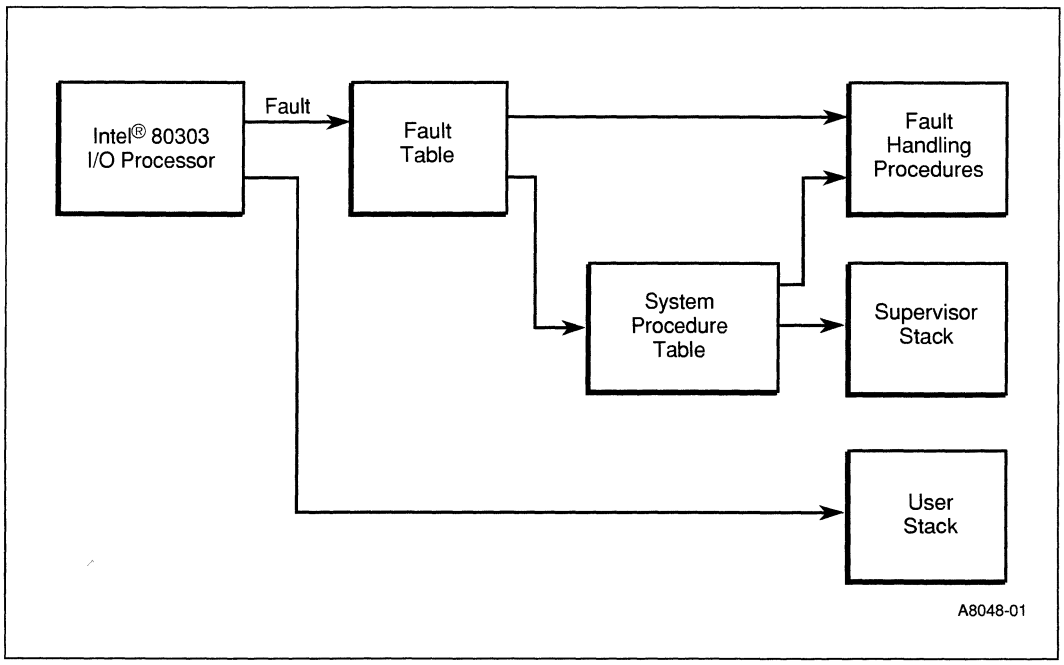
This chapter describes the Intel® 80303 I/O processor fault handling facilities. Subjects covered include the fault handling data structures and fault handling mechanisms. See Section 9.10, "Fault Reference" on page 9-22 for detailed information on each fault type.

9.1 Fault Handling Overview

The Intel® i960® architecture defines various conditions in code and/or the processor's internal state that could cause the processor to deliver incorrect or inappropriate results or that could cause it to choose an undesirable control path. These are called *fault conditions*. For example, the architecture defines faults for divide-by-zero and overflow conditions on integer calculations with an inappropriate operand value.

As shown in Figure 9-1, the architecture defines a fault table, a system procedure table, a set of fault handling procedures and stacks (user stack, supervisor stack and interrupt stack) to handle processor-generated faults.

Figure 9-1. Fault-Handling Data Structures



The fault table contains pointers to fault handling procedures. The system procedure table optionally provides an interface to any fault handling procedure and allows faults to be handled in supervisor mode. Stack frames for fault handling procedures are created on either the user or supervisor stack, depending on the mode in which the fault is handled. When the processor is in the interrupted state, the processor uses the interrupt stack.

Once these data structures and the code for the fault procedures are established in memory, the processor handles faults automatically and independently from application software.

The processor can detect a fault at any time while executing instructions, whether from a program, interrupt handling procedure or fault handling procedure. When a fault occurs, the processor determines the fault type and selects a corresponding fault handling procedure from the fault table. It then invokes the fault handling procedure by means of an implicit call. As described later in this chapter, the fault handler call can be:

- A local call (call-extended operation)
- A system-local call (local call through the system procedure table)
- A system-supervisor call (supervisor call through the system procedure table)

A normal fault condition is handled by the processor in the following manner:

- The current local registers are saved and cached on-chip.
- PFP = FP and the value 001 is written to the Return Type Field (Fault Call). Refer to Section 7.8, "Returns" on page 7-19 for more information.
- When the fault call is a system-supervisor call from user mode, the processor switches to the supervisor stack; otherwise, SP is re-aligned on the current stack.
- The processor writes the fault record on the new stack. This record includes information on the fault and the processor's state when the fault was generated.
- The Instruction Pointer (IP) of the first instruction of the fault handler is accessed through the fault table or through the system procedure table (for system fault calls).

After the fault record is created, the processor executes the selected fault handling procedure. When a fault is recoverable (i.e., the program can be resumed after handling the fault) the Return Instruction Pointer (RIP) is defined for the fault being serviced (Section 9.10, "Fault Reference" on page 9-22, and the processor resumes execution at the RIP upon return from the fault handler. When the RIP is undefined, the fault handling procedure can create one by using the **flushreg** instruction followed by a modification of the RIP in the previous frame. The fault handler can also call a debug monitor or reset the processor instead of resuming prior execution.

This procedure call mechanism also handles faults that occur:

- While the processor is servicing an interrupt
- While the processor is servicing another fault

9.2 Fault Types

The i960 architecture defines a basic set of faults that are categorized by type and subtype. Each fault has a unique type and subtype number. When the processor detects a fault, it records the fault type and subtype numbers in the fault record. It then uses the type number to select the fault handling procedure.

The fault handling procedure can optionally use the subtype number to select a specific fault handling action. The 80303 I/O processor recognizes i960 architecture-defined faults and a new fault subtype for detecting unaligned memory accesses. Table 9-1 lists all faults that the 80303 I/O processor detects, arranged by type and subtype. Text that follows the table gives column definitions.

Table 9-1. Fault Types and Subtypes

Fault Type		Fault Subtype		Fault Record
Number	**Name**	**Number or Bit Position**	**Name**	
0H	OVERRIDE	NA	NA	See section 8.10.1, "Overrides" (pg. 8-23)
0H	PARALLEL	NA	NA	see Section 9.6.4, "Parallel Faults" on page 9-11
1H	TRACE	Bit 1 Bit 2 Bit 3 Bit 4 Bit 5 Bit 6 Bit 7	INSTRUCTION BRANCH CALL RETURN PRERETURN SUPERVISOR MARK/BREAKPOINT	0001 0002H 0001 0004H 0001 0008H 0001 0010H 0001 0020H 0001 0040H 0001 0080H
2H	OPERATION	1H 2H 3H 4H	INVALID_OPCODE UNIMPLEMENTED UNALIGNED INVALID_OPERAND	0002 0001H 0002 0002H 0002 0003H 0002 0004H
3H	ARITHMETIC	1H 2H	INTEGER_OVERFLOW ZERO-DIVIDE	0003 0001H 0003 0002H
4H	Reserved			
5H	CONSTRAINT	1H	RANGE	0005 0001H
6H	Reserved			
7H	PROTECTION	Bit 1	LENGTH	0007 0002H
8H - 9H	Reserved			
AH	TYPE	1H	MISMATCH	000A 0001H
BH - FH	Reserved			

In Table 9-1:

- The first (left-most) column contains the fault type numbers in hexadecimal.
- The second column shows the fault type name.
- The third column gives the fault subtype number as either: (1) a hexadecimal number or (2) as a bit position in the fault record's 8-bit fault subtype field. The bit position method of indicating a fault subtype is used for certain faults (such as trace faults) in which two or more fault subtypes may occur simultaneously.
- The fourth column gives the fault subtype name. For convenience, individual faults are referenced by their fault-subtype names. Thus an OPERATION.INVALID_OPERAND fault is referred to as an INVALID_OPERAND fault; an ARITHMETIC.INTEGER_OVERFLOW fault is referred to as an INTEGER_OVERFLOW fault.
- The fifth column shows the encoding of the word in the fault record that contains the fault type and fault subtype numbers.

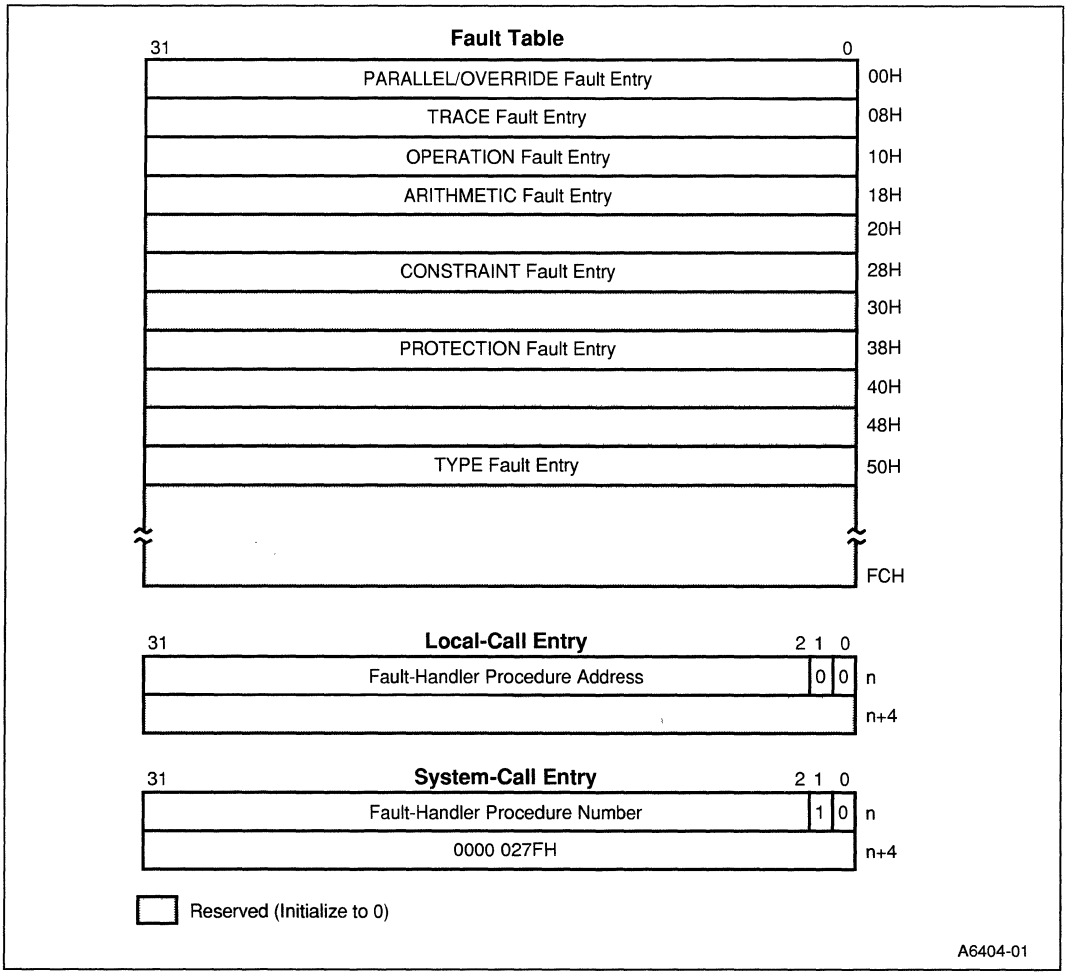
Other i960 processor family members may provide extensions that recognize additional fault conditions. Fault type and subtype encoding allows all faults to be included in the fault table: those that are common to all i960 processors and those that are specific to one or more family members. The fault types are used consistently for all family members. For example, Fault Type 4H is reserved for floating point faults. Any i960 processor with floating point operations uses Entry 4H to store the pointer to the floating point fault handling procedure.

9.3 Fault Table

The fault table (Figure 9-3) is the processor's pathway to the fault handling procedures. It can be located anywhere in the address space. From the Process Control Block, the processor obtains a pointer to the fault table during initialization.

The fault table contains one entry for each fault type. When a fault occurs, the processor uses the fault type to select an entry in the fault table. From this entry, the processor obtains a pointer to the fault handling procedure for the type of fault that occurred. Once called, a fault handling procedure has the option of reading the fault subtype or subtypes from the fault record when determining the appropriate fault recovery action.

Figure 9-2. Fault Table and Fault Table Entries



As indicated in Figure 9-3, two fault table entry types are allowed: local-call entry and system-call entry. Each is two words in length. The entry type field (bits 0 and 1 of the entry's first word) and the value in the entry's second word determine the entry type.

local-call entry (type 00_2)	Provides an instruction pointer for the fault handling procedure. The processor uses this entry to invoke the specified procedure by means of an implicit local-call operation. The second word of a local procedure entry is reserved. It must be set to zero when the fault table is created and not accessed after that.
system-call entry (type 10_2)	Provides a procedure number in the system procedure table. This entry must have an entry type of 10_2 and a value in the second word of 0000 027FH. Using this entry, the processor invokes the specified fault handling procedure by means of an implicit call-system operation similar to that performed for the **calls** instruction. A fault handling procedure in the system procedure table can be called with a system-local call or a system-supervisor call, depending on the entry type in the system-procedure table.

Other entry types (01_2 and 11_2) are reserved and have unpredictable behavior.

To summarize, a fault handling procedure can be invoked through the fault table in any of three ways: a local call, a system-local call or a system-supervisor call.

9.4 Stack Used in Fault Handling

The i960 architecture does not define a dedicated fault handling stack. Instead, to handle a fault, the processor uses either the user, interrupt or supervisor stack, whichever is active when the fault is generated. There is, however, one exception: if the user stack is active when a fault is generated and the fault handling procedure is called with an implicit system supervisor call, the processor switches to the supervisor stack to handle the fault.

9.5 Fault Record

When a fault occurs, the processor records information about the fault in a fault record in memory. The fault handling procedure uses the information in the fault record to correct or recover from the fault condition and, if possible, resume program execution. The fault record is stored on the same stack that the fault handling procedure uses to handle the fault.

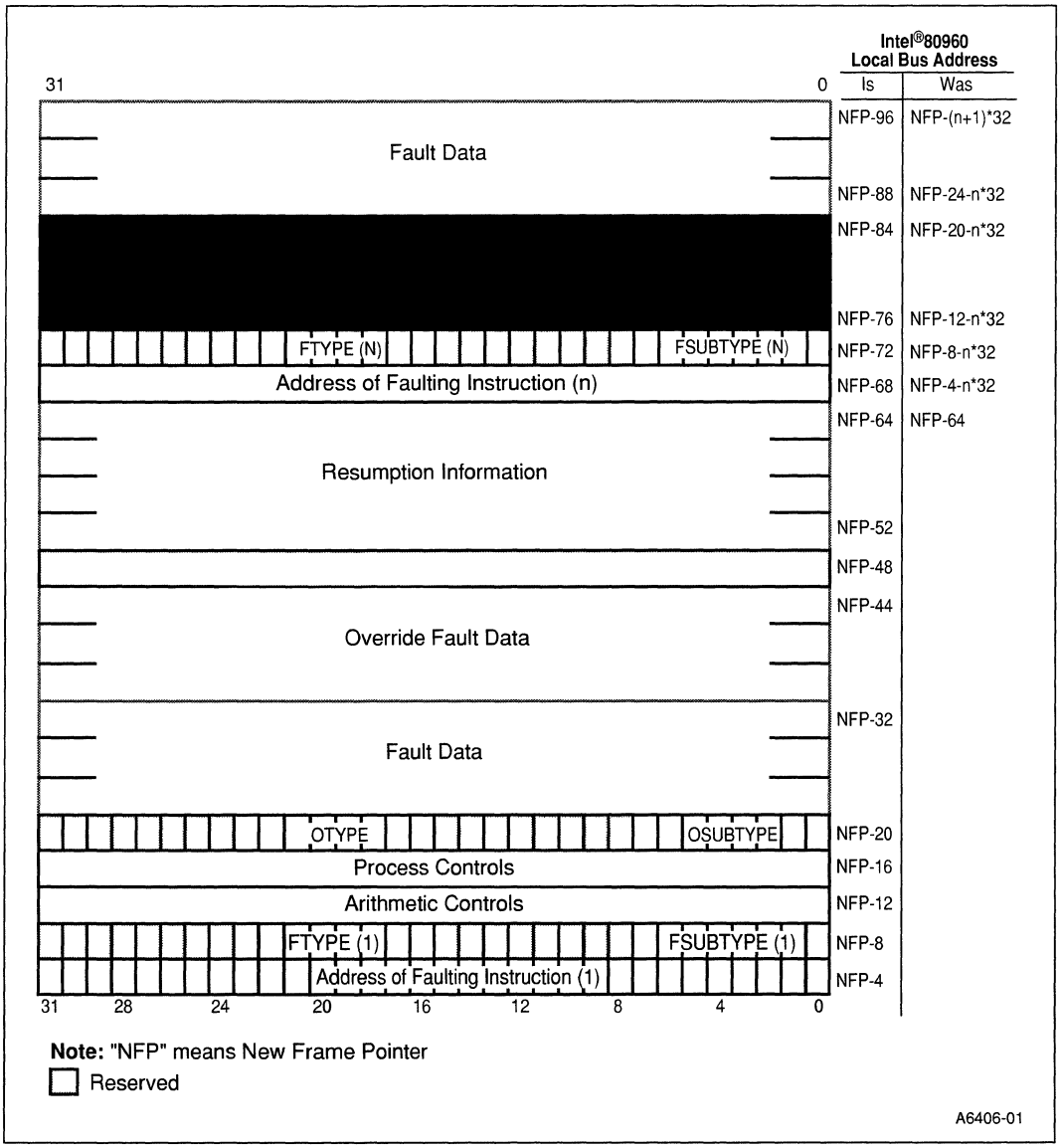
9.5.1 Fault Record Description

Figure 9-3 shows the fault record's structure. In this record, the fault's type number and subtype number (or bit positions for multiple subtypes) are stored in the fault type and subtype fields, respectively. The Address of Faulting Instruction Field contains the IP of the instruction that caused the processor to fault.

When a fault is generated, the existing PC and AC register contents are stored in their respective fault record fields. The processor uses this information to resume program execution after the fault is handled.

The Resumption Field is used to store information about a pending trace fault. When a trace fault and a non-trace fault occur simultaneously, the non-trace fault is serviced first and the pending trace may be lost depending on the non-trace fault encountered. The Trace Reporting paragraph for each fault specifies whether the pending trace is kept or lost.

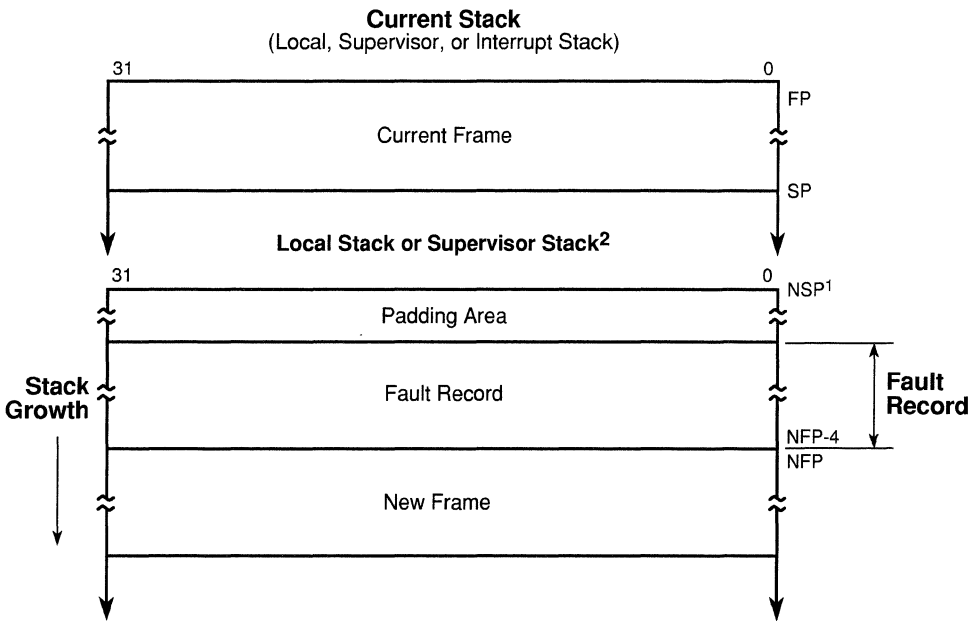
Figure 9-3. Fault Record



9.5.2 Fault Record Location

The fault record is stored on the stack that the processor uses to execute the fault handling procedure. As shown in Figure 9-4, this stack can be the user stack, supervisor stack or interrupt stack. The fault record begins at byte address NFP-1. NFP refers to the new frame pointer that is computed by adding the memory size allocated for padding and the fault record to the new stack pointer (NSP). The processor rounds the FP to the next 16-byte boundary and then allocates 80 bytes for the fault record.

Figure 9-4. Storage of the Fault Record on the Stack



Notes:
1. If the call to the fault handler procedure does not require a stack switch, the new stack pointer (NSP) is the same as SP.
2. If the processor is in user mode and the fault handler procedure is called with a system supervisor call, the processor switches to the supervisor stack.

A6405-01

9.6 Multiple and Parallel Faults

Multiple fault conditions can occur during a single instruction execution and during multiple instruction execution when the instructions are executed by different units within the processor. The following sections describe how faults are handled under these conditions.

9.6.1 Multiple Non-Trace Faults on the Same Instruction

Multiple fault conditions can occur during a single instruction execution. For example, an instruction can have an invalid operand and unaligned address. When this situation occurs, the processor is required to recognize and generate at least one of the fault conditions. The processor may not detect all fault conditions and reports only one detected non-trace fault on a single instruction.

In a multiple fault situation, the reported fault condition is left to the implementation.

9.6.2 Multiple Trace Fault Conditions on the Same Instruction

Trace faults on different instructions cannot happen concurrently, because trace faults are precise (Section 9.9, "Precise and Imprecise Faults" on page 9-20). Multiple trace fault conditions on the same instruction are reported in a single trace fault record (with the exception of prereturn trace, which always happens alone). To support multiple fault reporting, the trace fault uses bit positions in the fault-subtype field to indicate occurrences of multiple faults of the same type (Table 9-1).

9.6.3 Multiple Trace and Non-Trace Fault Conditions on the Same Instruction

The execution of a single instruction can create one or more trace fault conditions in addition to multiple non-trace fault conditions. When this occurs:

- The pending trace is dismissed if any of the non trace faults dismisses it, as mentioned in the "Trace Reporting" paragraph for that fault in Section 9.10, "Fault Reference" on page 9-22.
- The processor services one of the non trace faults.
- Finally, the trace is serviced upon return from the non-trace fault handler if it was not dismissed in step 1.

9.6.4 Parallel Faults

The 80303 I/O processor exploits the architecture tolerance of out-of-order instruction execution, by issuing instructions to independent execution units within the processor. The following subsections describe how the processor handles faults in this environment.

9.6.4.1 Faults on Multiple Instructions Executed in Parallel

When AC.nif=0, imprecise faults relative to different instructions executing in parallel may be reported in a single parallel fault record. For these conditions, the processor calls a unique fault handler, the PARALLEL fault handler (Section 9.9.4, "No Imprecise Faults (AC.nif) Bit" on page 9-21). This mechanism allows instructions that can fault to be executed in parallel with other instructions or out of order.

In parallel fault situations, the processor saves the fault type and subtype of the second and subsequent faults detected in the optional section of the fault record. The optional section is the area below NFP-64 where the fault records for each of the parallel faults that occurred are stored. The fault handling procedure for parallel faults can then analyze the fault record and handle the faults. The fault record for parallel faults is described in the next section.

When the RIP is undefined for at least one of the faults found in the parallel fault record, then the RIP of the parallel fault handler is undefined. In this case, the parallel fault handling procedure can either create a RIP and return or call a debug monitor to analyze the faults.

When the RIP is defined for all faults found in the fault record, then it points to the next instruction not yet executed. The parallel fault handler can simply return to the next instruction not yet executed with a **ret** instruction.

Consider the following code example, where the **muli** and the **addi** instructions both have overflow conditions. AC.om=0, AC.nif = 0, and both instructions are in the instruction cache at the time of their execution. The **addi** and **muli** are allowed to execute in parallel because AC.nif = 0 and the faults that these instructions can generate (ARITHMETIC) are imprecise.

```
muli        g2, g4, g6;
addi        g8, g9, g10;          # results in integer overflow
```

The fault on the **addi** is detected before the fault on the **muli** because the **muli** takes longer to execute. The fault call synchronizes faults on the way to the overflow fault handler for the **addi** instruction (Section 9.9.5, "Controlling Fault Precision" on page 9-21), which is when the **muli** fault is detected. The processor builds a parallel fault record with information relative to both faults and calls the parallel fault handler. In the fault handler, ARITHMETIC faults may be recovered by storing the desired result of the instruction in the proper destination register and setting the AC.of flag (optional) to indicate that an overflow occurred. A **ret** at the end of the parallel fault handler routine then returns to the next instruction not yet executed in the program flow.

On the 80303 I/O processor, the **muli** overflow fault is the only fault that can happen with a delay. Therefore, parallel fault records can report a maximum of two faults, one of which must be a **muli** ARITHMETIC.INTEGER_OVERFLOW fault.

A parallel fault handler must be accessed through a system-supervisor call. Local and system-local parallel fault handlers are not supported by the architecture and have unpredictable behavior. Tracing is disabled upon entry into the parallel fault handler (PC.te is cleared). It is restored upon return from the handler. To prevent infinite internal loops, the parallel fault handler should not set PC.te.

9.6.4.2 Fault Record for Parallel Faults

When parallel faults occur, the processor selects one of the faults and records it in the first 16 bytes of the fault record as described in Section 9.5.1, "Fault Record Description" on page 9-8. The remaining parallel faults are written to the fault record's optional section, and the fault handling procedure for parallel faults is invoked. Figure 9-3 shows the structure of the fault record for parallel faults.

The OType/OSubtype word at NFP - 20 contains the number of parallel faults. The optional section also contains a 32-byte parallel fault record for each additional parallel fault. These parallel fault records are stored incrementally in the fault record starting at byte offset NFP-68. The fault record for each additional fault contains only the fault type, fault subtype, address-of-faulting-instruction and the optional fault section. (For example, when two parallel faults occur, the fault record for the second fault is located from NFP-96 to NFP-65.)

For the second fault recorded (n=2), the relationship (NFP-8-(n * 32)) reduces to NFP-72. For the 80303 I/O processor, a maximum of two faults are reported in the parallel fault record, and one of them must be the ARITHMETIC.INTEGER_OVERFLOW fault on a **muli** instruction.

9.6.5 Override Faults

The 80303 I/O processor can detect a fault condition while the processor is preparing to service a previously detected fault. When this occurs, it is called an *override condition*. This section describes this condition and how the processor handles it.

A normal fault condition is handled by the processor in the following manner:

- The current local registers are saved and cached on-chip.
- PFP = FP and the value 001 is written to the Return Type Field (Fault Call). Refer to Section 7.8, "Returns" on page 7-19 for more information.
- When the fault call is a system-supervisor call from user mode, the processor switches to the supervisor stack; otherwise, SP is re-aligned on the current stack.
- The processor writes the fault record on the new stack.
- The IP of the first instruction of the fault handler is accessed through the fault table or through the system procedure table (for system fault calls).

A fault that occurs during any of the above actions is called an override fault. In response to this condition, the processor does the following:

- Switches the execution mode to supervisor.
- Selects the override condition that shows that the writing of the fault record was unsuccessful. If no such fault exists, the processor selects one of the other fault conditions. This method ensures that the fault handler has information regarding the fault record write.
- Saves information pertaining to the override condition selected. The fault record describes the first fault as described previously. Field OType contains the fault type of the second fault, field OSubtype contains the fault subtype of the second fault and field override-fault-data contains what would normally be the fault data field for the second fault type.
- Attempts to access the IP of the first instruction in the override fault handler through the system procedure table.

It should be noted that a fault that occurs while the processor is actually executing a fault handling procedure is not an override fault. The override fault entry is entry 0. When the override fault entry in the fault table points to a location beyond the system procedure table, the processor enters system error mode. Override fault conditions include: PROTECTION and OPERATION.UNIMPLEMENTED faults.

An override fault handler must be accessed through a system-supervisor call. Local and system-local override fault handlers are not supported by the architecture and have an unpredictable behavior. Tracing is disabled upon entry into the override fault handler (PC.te is cleared). It is restored upon return from the handler. To prevent infinite internal loops, the override fault handler should not set PC.te.

9.6.6 System Error

When a fault is detected while the processor is in the process of servicing an override or parallel fault, the processor enters the system error state. Note that "servicing" indicates that the processor has detected the override or parallel fault, but has not begun executing the fault handling procedure. This type of error causes the processor to enter a system error state. In this state, the processor uses only one read bus transaction to signal the fail code message; the address of the bus transaction is the fail code itself. See Section 11.3.1.5, "FAIL# Code" on page 11-9.

9.7 Fault Handling Procedures

The fault handling procedures can be located anywhere in the address space except within the on-chip data RAM or MMR space. Each procedure must begin on a word boundary. The processor can execute the procedure in user or supervisor mode, depending on the fault table entry type.

9.7.1 Possible Fault Handling Procedure Actions

The processor allows easy recovery from many faults that occur. When fault recovery is possible, the processor's fault handling mechanism allows the processor to automatically resume work on the program or pending interrupt when the fault occurred. Resumption is initiated with a **ret** instruction in the fault handling procedure.

When recovery from the fault is not possible or not desirable, the fault handling procedure can take one of the following actions, depending on the nature and severity of the fault condition (or conditions, in the case of multiple faults):

- Return to a point in the program or interrupt code other than the point of the fault.
- Call a debug monitor.
- Perform processor or system shutdown with or without explicitly saving the processor state and fault information.

When working with the processor at the development level, a common fault handling strategy is to save the fault and processor state information and call a debugging tool such as a monitor.

9.7.2 Program Resumption Following a Fault

Because of the wide variety of faults, they can occur at different times with respect to the faulting instruction:

- Before execution of the faulting instruction (e.g., fetch from on-chip RAM)
- During instruction execution (e.g., integer overflow)
- Immediately following execution (e.g., trace)

9.7.2.1 Faults Happening Before Instruction Execution

The following fault types occur before instruction execution:

- ARITHMETIC.ZERO_DIVIDE
- TYPE.MISMATCH
- PROTECTION.LENGTH
- All OPERATION subtypes except UNALIGNED

For these faults, the contents of a destination register are lost, and memory is not updated. The RIP is defined for the ARITHMETIC.ZERO_DIVIDE fault only. In some cases the fault occurs before the faulting instruction is executed, the faulting instruction may be fixed and re-executed upon return from the fault handling procedure.

9.7.2.2 Faults Happening During Instruction Execution

The following fault types occur during instruction execution:

- CONSTRAINT.RANGE
- OPERATION.UNALIGNED
- ARITHMETIC.INTEGER_OVERFLOW

For these faults, the fault handler must explicitly modify the RIP to return to the faulting application (except for ARITHMETIC.INTEGER_OVERFLOW).

When a fault occurs during or after execution of the faulting instruction, the fault may be accompanied by a program state change such that program execution cannot be resumed after the fault is handled. For example, when an integer overflow fault occurs, the overflow value is stored in the destination. When the destination register is the same as one of the source registers, the source value is lost, making it impossible to re-execute the faulting instruction.

9.7.2.3 Faults Happening After Instruction Execution

For these faults, the Return Instruction Pointer (RIP) is defined and the fault handler can return to the next instruction in the flow:

- TRACE
- ARITHMETIC.INTEGER_OVERFLOW

In general, resumption of program execution with no changes in the program's control flow is possible with the following fault types or subtypes:

- All TRACE Subtypes

The effect of specific fault types on a program is defined in Section 9.10, "Fault Reference" on page 9-22 under the heading Program State Changes.

9.7.3 Return Instruction Pointer (RIP)

When a fault handling procedure is called, a Return Instruction Pointer (RIP) is saved in the image of the RIP in the faulting frame. The RIP can be accessed at address PFP+8 while executing the fault handler after a **flushreg**. The RIP in the previous frame points to an instruction where program execution can be resumed with no break in the program's control flow. It generally points to the faulting instruction or to the next instruction to be executed. In some instances, however, the RIP is undefined. RIP content for each fault is described in Section 9.10, "Fault Reference" on page 9-22.

9.7.4 Returning to Point in Program Where Fault Occurred

As described in Section 9.7.2, "Program Resumption Following a Fault" on page 9-15, most faults can be handled such that program control flow is not affected. In this case, the processor allows a program to be resumed at the point where the fault occurred, following a return from a fault handling procedure (initiated with a **ret** instruction). The resumption mechanism used here is similar to that provided for returning from an interrupt handler.

Also, to restore the PC register from the fault record upon return from the fault handler, the fault handling procedure must be executed in supervisor mode either by using a supervisor call or by running the program in supervisor mode. See the pseudocode in Section 6.2.54, "ret" on page 6-76.

9.7.5 Program Return Point Other than Occurred Fault

A fault handling procedure can also return to a point in the program other than where the fault occurred. To do this, the fault procedure must alter the RIP. To do this reliably, the fault handling procedure should perform the following steps:

1. Flush the local register sets to the stack with a **flushreg** instruction.
2. Modify the RIP in the previous frame.
3. Clear the trace-fault-pending flag in the fault record's process controls field before the return (optional).
4. Execute a return with the **ret** instruction.

Use this technique carefully and only in situations where the fault handling procedure is closely coupled with the application program.

9.7.6 Fault Controls

For certain fault types and subtypes, the processor employs register mask bits or flags that determine whether or not a fault is generated when a fault condition occurs. Table 9-2 summarizes these flags and masks, the data structures in which they are located, and the fault subtypes they affect.

The integer overflow mask bit inhibits the generation of integer overflow faults. The use of this mask is discussed in Section 9.10, "Fault Reference" on page 9-22.

The Arithmetic Controls no imprecise faults (AC.nif) bit controls the synchronizing of faults for a category of faults called imprecise faults. The function of this bit is described in Section 9.9, "Precise and Imprecise Faults" on page 9-20.

TC register trace mode bits and the PC register trace enable bit support trace faults. Trace mode bits enable trace modes; the trace enable bit (PC.te) enables trace fault generation. The use of these bits is described in the trace faults description in Section 9.10, "Fault Reference" on page 9-22. Further discussion of these flags is provided in Chapter 10, "Tracing and Debugging".

Table 9-2. Fault Control Bits and Masks

Flag or Mask Name	Location	Faults Affected
Integer Overflow Mask Bit	Arithmetic Controls (AC) Register	INTEGER_OVERFLOW
No Imprecise Faults Bit	Arithmetic Controls (AC) Register	All Imprecise Faults
Trace Enable Bit	Process Controls (PC) Register	All TRACE Faults
Trace Mode	Trace Controls (TC) Register	All TRACE Faults except hardware breakpoint traces and **fmark**
Unaligned Fault Mask	Process Control Block (PRCB)	UNALIGNED Fault

The unaligned fault mask bit is located in the process control block (PRCB), which is read from the fault configuration word (located at address PRCB pointer + 0CH) during initialization. It controls whether unaligned memory accesses generate a fault.

9.8 Fault Handling Action

Once a fault occurs, the processor saves the program state, calls the fault handling procedure and, if possible, restores the program state when the fault recovery action completes. No software other than the fault handling procedures is required to support this activity.

Three types of implicit procedure calls can be used to invoke the fault handling procedure: a local call, a system-local call and a system-supervisor call.

The following subsections describe actions the processor takes while handling faults. It is not necessary to read these sections to use the fault handling mechanism or to write a fault handling procedure. This discussion is provided for those readers who wish to know the details of the fault handling mechanism.

9.8.1 Local Fault Call

When the selected fault handler entry in the fault table is an entry type 000_2 (a local procedure), the processor operates as described in Section 7.1.3.1, "Call Operation" on page 7-6, with the following exceptions:

- A new frame is created on the stack that the processor is currently using. The stack can be the user stack, supervisor stack or interrupt stack.
- The fault record is copied into the area allocated for it in the stack, beginning at NFP-1 (Figure 9-4).
- The processor gets the IP for the first instruction in the called fault handling procedure from the fault table.
- The processor stores the fault return code (001_2) in the PFP return type field.

When the fault handling procedure is not able to perform a recovery action, it performs one of the actions described in Section 9.7.2, "Program Resumption Following a Fault" on page 9-15.

When the handler action results in recovery from the fault, a **ret** instruction in the fault handling procedure allows processor control to return to the program that was executing when the fault occurred. Upon return, the processor performs the action described in Section 7.1.3.2, "Return Operation" on page 7-6, except that the arithmetic controls field from the fault record is copied into the AC register. When the processor is in user mode before execution of the return, the process controls field from the fault record is not copied back to the PC register.

9.8.2 System-Local Fault Call

When the fault handler selects an entry for a local procedure in the system procedure table (entry type 10_2), the processor performs the same action as is described in the previous section for a local fault call or return. The only difference is that the processor gets the fault handling procedure's address from the system procedure table rather than from the fault table.

9.8.3 System-Supervisor Fault Call

When the fault handler selects an entry for a supervisor procedure in the system procedure table, the processor performs the same action described in Section 7.1.3.1, "Call Operation" on page 7-6, with the following exceptions:

- When the fault occurs while in user mode, the processor switches to supervisor mode, reads the supervisor stack pointer from the system procedure table and switches to the supervisor stack. A new frame is then created on the supervisor stack.
- When the fault occurs while in supervisor mode, the processor creates a new frame on the current stack. When the processor is executing a supervisor procedure when the fault occurred, the current stack is the supervisor stack; when it is executing an interrupt handler procedure, the current stack is the interrupt stack. (The processor switches to supervisor mode when handling interrupts.)
- The fault record is copied into the area allocated for it in the new stack frame, beginning at NFP-1 (Figure 9-4).
- The processor gets the IP for the first instruction of the fault handling procedure from the system procedure table (using the index provided in the fault table entry).
- The processor stores the fault return code (001_2) in the PFP register return type field. When the fault is not a trace, parallel or override fault, it copies the state of the system procedure table trace control flag (byte 12, bit 0) into the PC register trace enable bit. When the fault is a trace, parallel or override fault, the trace enable bit is cleared.

On a return from the fault handling procedure, the processor performs the action described in Section 7.1.3.2, "Return Operation" on page 7-6 with the addition of the following:

- The fault record arithmetic controls field is copied into the AC register.
- When the processor is in supervisor mode prior to the return from the fault handling procedure (which it should be), the fault record process controls field is copied into the PC register. The mode is then switched back to user, if it was in user mode before the call.
- The processor switches back to the stack it was using when the fault occurred. (When the processor was in user mode when the fault occurred, this operation causes a switch from the supervisor stack to the user stack.)
- When the trace-fault-pending flag and trace enable bits are set in the PC field of the fault record, the trace fault on the instruction at the origin of the supervisor fault call is handled at this time.

The user should note that PC register restoration causes any changes to the process controls done by the fault handling procedure to be lost.

9.8.4 Faults and Interrupts

When an interrupt occurs during an instruction that faults, an instruction that has already faulted, or fault handling procedure selection, the processor:

1. Completes the selection of the fault handling procedure.
2. Creates the fault record.
3. Services the interrupt just prior to executing the first instruction of the fault handling procedure.
4. Handles the fault upon return from the interrupt.

Handling the interrupt before the fault reduces interrupt latency.

9.9 Precise and Imprecise Faults

As described in Section 9.10.5, "PARALLEL Faults" on page 9-27, the i960 architecture — to support parallel and out-of-order instruction execution — allows some faults to be generated together.

The processor provides two mechanisms for controlling the circumstances under which faults are generated: the AC register no-imprecise-faults bit (AC.nif) and the instructions that synchronize faults. See Section 9.9.5, "Controlling Fault Precision" on page 9-21 for more information. Faults are categorized as precise, imprecise and asynchronous. The following subsections describe each.

9.9.1 Precise Faults

A fault is precise if it meets all of the following conditions:

- The faulting instruction is the earliest instruction in the instruction issue order to generate a fault.
- All instructions after the faulting instruction, in instruction issue order, are guaranteed not to have executed.

TRACE and PROTECTION.LENGTH faults are always precise. Precise faults cannot be found in parallel records with other precise or imprecise faults.

9.9.2 Imprecise Faults

Faults that do not meet all of the requirements for precise faults are considered imprecise. For imprecise faults, the state of execution of instructions surrounding the faulting instruction may be unpredictable. When instructions are executed out of order and an imprecise fault occurs, it may not be possible to access the source operands of the instruction. This is because they may have been modified by subsequent instructions executed out of order. However, the RIP of some imprecise faults (e.g., ARITHMETIC) points to the next instruction that has not yet executed and guarantees the return from the fault handler to the original flow of execution. Faults that the architecture allows to be imprecise are OPERATION, CONSTRAINT, ARITHMETIC and TYPE.

9.9.3 Asynchronous Faults

Asynchronous faults are those whose occurrence has no direct relationship to the instruction pointer. This group includes MACHINE faults, which are not implemented on the 80303 I/O processor.

9.9.4 No Imprecise Faults (AC.nif) Bit

The Arithmetic Controls no imprecise faults (AC.nif) bit controls imprecise fault generation. When AC.nif is set, out of order instruction execution is disabled and all faults generated are precise. Therefore, setting this bit reduces processor performance. When AC.nif is clear, several imprecise faults may be reported together in a parallel fault record. Precise faults can never be found in parallel fault records, thus only more than one imprecise fault occurring concurrently with AC.nif = 0 can produce a parallel fault.

Compiled code should execute with the AC.nif bit clear, using **syncf** where necessary to ensure that faults occur in order. In this mode, imprecise faults are considered to be catastrophic errors from which recovery is not needed. This also allows the processor to take advantage of internal pipelining, which can speed up processing time. When only precise faults are allowed, the processor must restrict the use of pipelining to prevent imprecise faults.

The AC.nif bit should be set if recovery from one or more imprecise faults is required. For example, the AC.nif bit should be set if a program needs to handle and recover from unmasked integer-overflow faults and the fault handling procedure cannot be closely coupled with the application to perform imprecise fault recovery.

9.9.5 Controlling Fault Precision

The **syncf** instruction forces the processor to complete execution of all instructions that occur prior to **syncf** and to generate all faults before it begins work on instructions that occur after **syncf**. This instruction has two uses:

- It forces faults to be precise when the AC.nif bit is clear.
- It ensures that all instructions are complete and all faults are generated in one block of code before executing another block of code.

The implicit fault call operation synchronizes all faults. In addition, the following instructions or operations perform synchronization of all faults except MACHINE.PARITY:

- Call and return operations including **call**, **callx**, **calls** and **ret** instructions, plus the implicit interrupt and fault call operations.
- Atomic operations including **atadd** and **atmod**.

9.10 Fault Reference

This section describes each fault type and subtype and gives detailed information about what is stored in the various fields of the fault record. The section is organized alphabetically by fault type. The following paragraphs describe the information that is provided for each fault type.

Fault Type: Gives the number that appears in the fault record fault-type field when the fault is generated.

Fault Subtype: Lists the fault subtypes and the number associated with each fault subtype.

Function: Describes the purpose and handling of the fault type and each subtype.

RIP: Describes the value saved in the image of the RIP register in the stack frame that the processor was using when the fault occurred. In the RIP definitions, "next instruction" refers to the instruction directly after the faulting instruction or to an instruction to which the processor can logically return when resuming program execution.

Note: Note that the discussions of many fault types specify that the RIP contains the address of the instruction that would have executed next had the fault not occurred.

Fault IP: Describes the contents of the fault record's fault instruction pointer field, typically the faulting instruction's IP.

Fault Data: Describes any values stored in the fault record's fault data field.

Class: Indicates if a fault is precise or imprecise.

Program State Changes: Describes the process state changes that would prevent re-executing the faulting instruction if applicable.

Trace Reporting: Relates whether a trace fault (other than PRERET) can be detected on the faulting instruction, also if and when the fault is serviced.

Note: Additional information specific to particular implementations of the i960 architecture.

9.10.1 ARITHMETIC Faults

Fault Type: 3H

Fault Subtype:

Number	Name
0H	Reserved
1H	INTEGER_OVERFLOW
2H	ZERO_DIVIDE
3H-FH	Reserved

Function:

Indicates a problem with an operand or the result of an arithmetic instruction. An INTEGER_OVERFLOW fault is generated when the result of an integer instruction overflows its destination and the AC register integer overflow mask is cleared. Here, the result's *n* least significant bits are stored in the destination, where *n* is destination size. Instructions that generate this fault are:

addi	**subi**	**stis**
stib	**shli**	**ADDI<cc>**
muli	**divi**	**SUBI<cc>**
divo	**divi**	
ediv	**remi**	
remo	**modi**	

RIP: IP of the instruction that would have executed next if the fault had not occurred.

Fault IP: IP of the faulting instruction.

Class: Imprecise.

Program State Changes: Faults may be imprecise when executing with the AC.nif bit cleared. INTEGER_OVERFLOW and ZERO_DIVIDE faults may not be recoverable because the result is stored in the destination before the fault is generated (e.g., the faulting instruction cannot be re-executed if the destination register was also a source register for the instruction).

Trace Reporting: The trace is reported upon return from the arithmetic fault handler.

9.10.2 CONSTRAINT Faults

Fault Type: 5H

Fault Subtype:

Number	Name
0H	Reserved
1H	RANGE
2H-FH	Reserved

Function: Indicates the program or procedure violated an architectural constraint.

A CONSTRAINT.RANGE fault is generated when a FAULT<cc> instruction is executed and the AC register condition code field matches the condition required by the instruction.

RIP: No defined value.

Fault IP: Faulting instruction.

Class: Imprecise.

Program State Changes: These faults may be imprecise when executing with the AC.nif bit cleared. No changes in the program's control flow accompany these faults. A CONSTRAINT.RANGE fault is generated after the **FAULT<cc>** instruction executes. The program state is not affected.

Trace Reporting: Serviced upon return from the Constraint fault handler.

9.10.3 OPERATION Faults

Fault Type: 2H

Fault Subtype:

Number	Name
0H	Reserved
1H	INVALID_OPCODE
2H	UNIMPLEMENTED
3H	UNALIGNED
4H	INVALID_OPERAND
5H - FH	Reserved

Function: Indicates the processor cannot execute the current instruction because of invalid instruction syntax or operand semantics.

An INVALID_OPCODE fault is generated when the processor attempts to execute an instruction containing an undefined opcode or addressing mode.

An UNIMPLEMENTED fault is generated when the processor attempts to execute an instruction fetched from on-chip data RAM, or when a non-word or unaligned access to a memory-mapped region is performed, or when attempting to write memory-mapped region 0xFF0084XX when rights have not been granted.

An UNALIGNED fault is generated when the following conditions are present: (1) the processor attempts to access an unaligned word or group of words in non-MMR memory; and (2) the fault is enabled by the unaligned-fault mask bit in the PRCB fault configuration word.

An INVALID_OPERAND fault is generated when the processor attempts to execute an instruction that has one or more operands having special requirements that are not satisfied. This fault is generated when specifying a non-existent SFR or non-defined **sysctl, icctl, dcctl or intctl** command, or referencing an unaligned long-, triple- or quad-register group, or by referencing an undefined register, or by writing to the RIP register (r2).

RIP: No defined value.

Fault IP: Address of the faulting instruction.

Fault Data: When an UNALIGNED fault is signaled, the effective address of the unaligned access is placed in the fault record's optional data section, beginning at address NFP-24. This address is useful to debug a program that is making unintentional unaligned accesses.\

Class: Imprecise.

Program State Changes: For the INVALID_OPCODE and UNIMPLEMENTED faults (case: store to MMR), the destination of the faulting instruction is not modified. (For the UNALIGNED fault, the memory operation completes correctly before the fault is reported.) In all other cases, the destination is undefined.

Trace Reporting: OPERATION.UNALIGNED fault: the trace is reported upon return from the OPERATION fault handler. All other subtypes: the trace event is lost.

Notes: OPERATION.UNALIGNED fault is not implemented on i960 Kx and Sx CPUs.

9.10.4 OVERRIDE Faults

Fault Type:
Fault table entry = 10H
The fault type in the fault record on the stack equals the fault type of the initial fault. The fault type in the internal registers equals the fault type of the additional fault detected while attempting to service the initial fault.

Fault Subtype:
The fault subtype in the fault record on the stack equals the fault subtype of the initial fault. The fault subtype in the internal registers equals the fault subtype of the additional fault detected while attempting to service the initial fault.

Fault OType:
The fault type of the additional fault detected while attempting to deliver the program fault.

Fault OSubtype:
The fault subtype of the additional fault detected while attempting to deliver the program fault.

Function:
The override fault handler must be accessed through a system-supervisor call. Local and system-local override fault handlers are not supported and have an unpredictable behavior. Tracing is disabled upon entry into the override fault handler (PC.te is cleared). It is restored upon return from the handler. To prevent infinite internal loops, the override fault handler should not set PC.te.

Trace Reporting:
Same behavior as if the override condition had not existed. Refer to the description of the original program fault.

Note:
Fault handlers must not be placed in a GMU-protected area.

9.10.5 PARALLEL Faults

Fault Type: Fault table entry = 0H
Fault type in fault record = fault type of one of the parallel faults.

Fault Subtype: Fault subtype of one of the parallel faults.

Fault OType: 0H

Fault OSubtype: Number of parallel faults.

Number of Faults: Number of parallel faults.

Function: See Section 9.6.4, "Parallel Faults" on page 9-11 for a complete description of parallel faults. When the AC.nif=0, the architecture permits the processor to execute instructions in parallel and out-of-order by different execution units. When an imprecise fault occurs in any of these units, it is not possible to stop the execution of those instructions after the faulting instruction. It is also possible that more than one fault is detected from different instructions almost at the same time.

When there is more than one outstanding fault at the point when all execution units terminate, a parallel fault situation arises. The fault record of parallel faults contains the fault information of all faults that occurred in parallel. The number of parallel faults is indicated in the Parallel Faults Field (NFP-20). See Figure 9-3. The maximum size of the fault record is implementation dependent and depends on the number of parallel and pipeline execution units in the specific implementation.

The parallel fault handler must be accessed through a system-supervisor call. Local and system-local parallel fault handlers are not supported by the i960 processor and have an unpredictable behavior. Tracing is disabled upon entry into the parallel fault handler (PC.te is cleared). It is restored upon return from the handler. To prevent infinite internal loops, the parallel fault handler should not set PC.te.

RIP: When all parallel fault types allow a RIP to be defined, the **RIP** is the next instruction in the flow of execution, otherwise it is undefined.

Fault IP: IP of one of the faulting instructions.

Class: Imprecise.

Program State Changes: State changes associated with all the parallel faults.

Trace Reporting: If all parallel fault types allow for a resumption trace, then a trace is reported upon return from the parallel fault handler, or else it is lost.

9.10.6 PROTECTION Faults

Fault Type:	7H	
Fault Subtype:	**Number**	**Name**
	Bit 0	Reserved
	Bit 1	LENGTH
Function:	Indicates that a program or procedure is attempting to perform an illegal operation that the architecture protects against. A PROTECTION.LENGTH fault is generated when the index operand, used in a **calls** instruction, points to an entry beyond the extent of the system procedure table. Reserved	
RIP:	IP of the faulting instruction.	
Fault IP:	LENGTH: IP of the faulting instruction.	
Class:	Imprecise. (PROTECTION.LENGTH is precise even though the PROTECTION fault class is imprecise.)	
Program State Changes:	LENGTH: The instruction does not execute.	
Trace Reporting:	PROTECTION.LENGTH: The trace event is lost.	

9.10.7 TRACE Faults

Fault Type: 1H

Fault Subtype:

Number	Name
Bit 0	Reserved
Bit 1	INSTRUCTION
Bit 2	BRANCH
Bit 3	CALL
Bit 4	RETURN
Bit 5	PRERETURN
Bit 6	SUPERVISOR
Bit 7	MARK/BREAKPOINT

Function: Indicates the processor detected one or more trace events. The event tracing mechanism is described in Chapter 10, "Tracing and Debugging".

A trace event is the occurrence of a particular instruction or instruction type in the instruction stream. The processor recognizes seven different trace events: instruction, branch, call, return, prereturn, supervisor, mark. It detects these events only if the TC register mode bit is set for the event. If the PC register trace enable bit is also set, the processor generates a fault when a trace event is detected.

A TRACE fault is generated following the instruction that causes a trace event (or prior to the instruction for the prereturn trace event). The following trace modes are available:

INSTRUCTION	Generates a trace event following every instruction.
BRANCH	Generates a trace event following any branch instruction when branch is taken (a branch trace event does not occur on branch-and-link or call instructions).
CALL	Generates a trace event following any call or branch-and-link instruction or an implicit fault call.
RETURN	Generates a trace event following a ret.
PRERETURN	Generates a trace event prior to any ret instruction, provided the PFP register prereturn trace flag is set (the processor sets the flag automatically when a call trace is serviced). A prereturn trace fault is always generated alone.
SUPERVISOR	Generates a trace event following any calls instruction that references a supervisor procedure entry in the system procedure table and on a return from a supervisor procedure where the return status type in the PFP register is 0102 or 0112.
MARK/BREAKPOINT	Generates a trace event following the mark instruction. The MARK fault subtype bit, however, is used to indicate a match of the instruction-address breakpoint register or the data-address breakpoint register as well as the fmark and mark instructions.

RIP: Instruction immediately following the instruction traced, in instruction issue order, except for PRERETURN. For PRERETURN, the RIP is the return instruction traced.

Fault IP: IP of the faulting instruction for all except prereturn trace and call trace (on implicit fault calls), for which the fault IP field is undefined.

Class: Precise.

Fault IP: IP of the faulting instruction.

9.10.8 TYPE Faults

Fault Type: AH

Fault Subtype:

Number	Name
0H	Reserved
1H	MISMATCH
2H-FH	Reserved

Function:

Indicates a program or procedure attempted to perform an illegal operation on an architecture-defined data type or a typed data structure.

A TYPE.MISMATCH fault is generated when attempts are made to:

Execute a privileged (supervisor-mode only) instruction while the processor is in user mode.

modpc **intctl**
sysctl **inten**
icctl **intdis**
dcctl

- Write to on-chip data RAM while the processor is in supervisor-only write mode and BCON.irp is set.
- Write to the first 64 bytes of on-chip data RAM while the processor is in either user or supervisor mode and BCON.sirp is set.
- Write to memory-mapped registers in supervisor space from user mode.
- Write to timer registers while in user mode, when timer registers are protected against user-mode writes.

RIP: No defined value.

Class: Imprecise.

Program State Changes: The fault happens before execution of the instruction. Machine state is not changed.

Trace Reporting: The trace event is lost.

intel®

Tracing and Debugging 10

This chapter describes the Intel® 80303 I/O processor facilities for runtime activity monitoring. The Intel® i960® architecture provides facilities for monitoring processor activity through trace event generation. A trace event indicates a condition where the processor has just completed executing a particular instruction or a type of instruction or where the processor is about to execute a particular instruction. When the processor detects a trace event, it generates a trace fault and makes an implicit call to the fault handling procedure for trace faults. This procedure can, in turn, call debugging software to display or analyze the processor state when the trace event occurred. This analysis can be used to locate software or hardware bugs or for general system monitoring during program development.

Tracing is enabled by the process controls (PC) register trace enable bit and a set of trace mode bits in the trace controls (TC) register. Alternatively, the **mark** and **fmark** instructions can be used to generate trace events explicitly in the instruction stream.

The 80303 I/O processor also provides four hardware breakpoint registers that generate trace events and trace faults. Two registers are dedicated to trapping on instruction execution addresses, while the remaining two registers can trap on the addresses of various types of data accesses.

10.1 Trace Controls

To use the architecture's tracing facilities, software must provide trace fault handling procedures, perhaps interfaced with a debugging monitor. Software must also manipulate the following registers and control bits to enable the various tracing modes and enable or disable tracing in general.

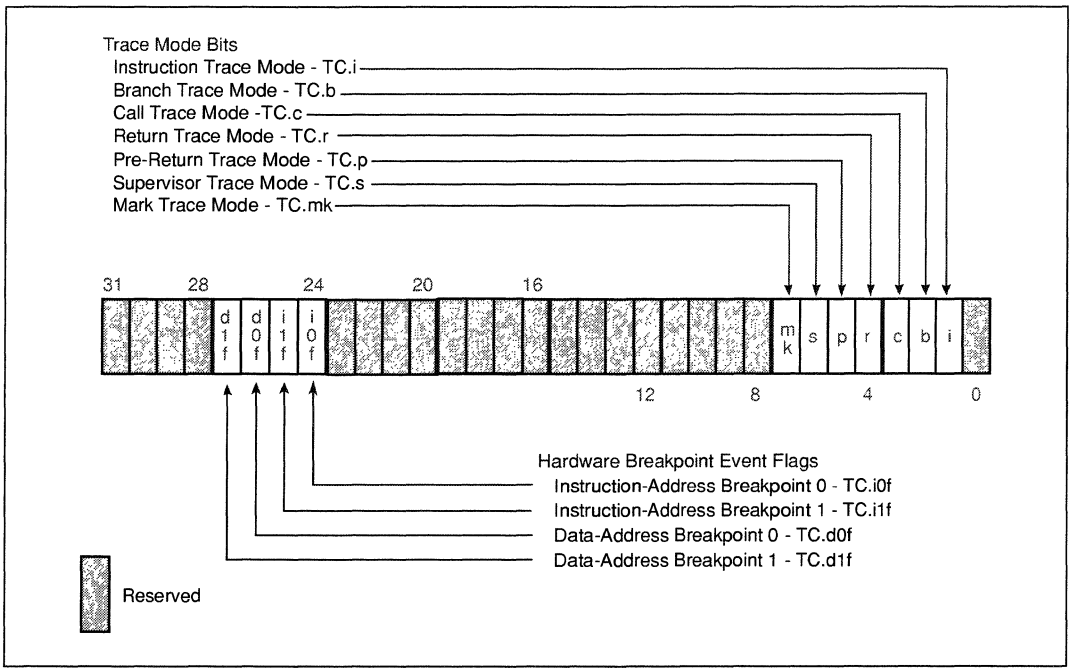
- TC register mode bits
- DAB0-DAB1 registers' address field and enable bit (in the control table)
- System procedure table supervisor-stack-pointer field trace control bit
- IPB0-IPB1 registers' address field (in the control table)
- PC register trace enable bit
- PFP register return status field prereturn trace flag (bit 3)
- BPCON register breakpoint mode bits and enable bits (in the control table)

These controls are described in the following subsections.

10.1.1 Trace Controls Register – TC

The TC register (Table 10-1) allows software to define conditions that generate trace events.

Table 10-1. Intel® 80303 I/O Processor Trace Controls Register – TC



The TC register contains mode bits and event flags. Mode bits define a set of tracing conditions that the processor can detect. For example, when the call-trace mode bit is set, the processor generates a trace event when a call or branch-and-link operation executes. See Section 10.2, "Trace Modes" on page 10-3. The processor uses event flags to monitor which breakpoint trace events are generated.

A special instruction, modify-trace-controls (**modtc**), allows software to modify the TC register. On initialization, the TC register is read from the Control Table. **modtc** can then be used to set or clear trace mode bits as required. Updating TC mode bits may take up to four non-branching instructions to take effect. Software can access the breakpoint event flags using **modtc**. The processor automatically sets and clears these flags as part of its trace handling mechanism: the breakpoint event flag corresponding to the trace being serviced is set in the TC while servicing a breakpoint trace fault; the TC event flags are cleared upon return from the trace fault handler. When the program is not in a trace fault handler, or when the trace is not for breakpoints, the TC event bits are clear. On the 80303 I/O processor, TC register bits 0, 8 through 23 and 28 through 31 are reserved. Software must initialize these bits to zero and cannot modify them afterwards.

10.1.2 PC Trace Enable Bit and Trace-Fault-Pending Flag

The Process Controls (PC) register trace enable bit and the trace-fault-pending flag in the PC field of the fault record control tracing (Section 3.6.3, "Process Controls Register – PC" on page 3-16). The trace enable bit enables the processor's tracing facilities; when set, the processor generates trace faults on all trace events.

Typically, software selects the trace modes to be used through the TC register. It then sets the trace enable bit to begin tracing. This bit is also altered as part of some call and return operations that the processor performs as described in Section 10.5.2, "Tracing on Calls and Returns" on page 10-11.

The update of PC.te through **modpc** may take up to four non-branching instructions to take effect. The update of PC.te through call and return operations is immediate.

The trace-fault-pending flag, in the PC field of the fault record, allows the processor to remember to service a trace fault when a trace event is detected at the same time as another event (e.g., non-trace fault, interrupt). The non-trace fault event is serviced before the trace fault, and depending on the event type and execution mode, the trace-fault-pending flag in the PC field of the fault record may be used to generate a fault upon return from the non-trace fault event (Section 10.5.2.4, "Tracing on Return from Implicit Call: Fault Case" on page 10-13).

10.2 Trace Modes

This section defines trace modes enabled through the TC register. These modes can be enabled individually or several modes can be enabled at once. Some modes overlap, such as call-trace mode and supervisor-trace mode.

- Instruction trace
- Branch trace
- Mark trace
- Prereturn trace
- Call trace
- Return trace
- Supervisor trace

See Section 10.4, "Handling Multiple Trace Events" on page 10-10 for a description of processor function when multiple trace events occur.

10.2.1 Instruction Trace

When the instruction-trace mode is enabled in TC (TC.i = 1) and tracing is enabled in PC (PC.te = 1), the processor generates an instruction-trace fault immediately after an instruction is executed. A debug monitor can use this mode (TC.i = 1, PC.te = 1) to single-step the processor.

10.2.2 Branch Trace

When the branch-trace mode is enabled in TC (TC.b = 1) and PC.te is set, the processor generates a branch-trace fault immediately after a branch instruction executes, if the branch is taken. A branch-trace event is not generated for conditional-branch instructions that do not branch, branch-and-link instructions, and call-and-return instructions.

10.2.3 Call Trace

When the call-trace mode is enabled in TC (TC.c = 1) and PC.te is set after the call operation, the processor generates a call-trace fault when a call instruction (**call**, **callx** or **calls**) or a branch-and-link instruction (**bal** or **balx**) executes. See Section 10.5.2.1, "Tracing on Explicit Call" on page 10-11 for a detailed description of call tracing on explicit instructions. Interrupt calls are never traced.

An implicit call to a fault handler also generates a call trace if TC.c and PC.te are set after the call. Refer to Section 10.5.2.2, "Tracing on Implicit Call" on page 10-12 for a complete description of this case.

When the processor services a trace fault, it sets the prereturn-trace flag (PFP register bit 3) in the new frame created by the call operation or in the current frame if a branch-and-link operation was performed. The processor uses this flag to determine whether or not to signal a prereturn-trace event on a **ret** instruction.

10.2.4 Return Trace

When the return-trace mode is enabled in TC and PC.te is set after the return instruction, the processor generates a return-trace fault for a return from explicit call (PFP.rrr = 000 or PFP.rrr = 01x). See Section 10.5.2.3, "Tracing on Return from Explicit Call" on page 10-13.

A return from fault may be traced and a return from interrupt cannot. See Section 10.5.2.4, "Tracing on Return from Implicit Call: Fault Case" on page 10-13 and Section 10.5.2.5, "Tracing on Return from Implicit Call: Interrupt Case" on page 10-13 for details.

10.2.5 Prereturn Trace

When the TC prereturn-trace mode, the PC.te, and the PFP prereturn-trace flag (PFP.p) are set, the processor generates a prereturn-trace fault prior to executing a **ret** execution. The dependence on PFP.p implies that prereturn tracing cannot be used without enabling call tracing. The processor sets PFP.p whenever it services a call-trace fault (as described above) for call-trace mode.

If another trace event occurs at the same time as the prereturn-trace event, the processor generates a fault on the non-prereturn-trace event first. Then, on a return from that fault handler, it generates a fault on the prereturn-trace event. The prereturn trace is the only trace event that can cause two successive trace faults to be generated between instruction boundaries.

10.2.6 Supervisor Trace

When supervisor-trace mode is enabled in TC, and PC.te is set, the processor generates a supervisor-trace fault after either of the following:

- A call-system instruction (**calls**) executes from user mode and the procedure table entry is for a system-supervisor call.
- A **ret** instruction executes from supervisor mode and the return-type field is set to 010_2 or 011_2 (i.e., return from **calls**).

This trace mode allows a debugging program to determine kernel-procedure call boundaries within the instruction stream.

10.2.7 Mark Trace

Mark trace mode allows trace faults to be generated at places other than those specified with the other trace modes, using the **mark** instruction. It should be noted that the MARK fault subtype bit in the fault record is used to indicate a match of the instruction-address breakpoint registers or the data-address breakpoint registers as well as the **fmark** and **mark** instructions.

10.2.7.1 Software Breakpoints

mark and **fmark** allow breakpoint trace faults to be generated at specific points in the instruction stream. When mark trace mode is enabled and PC.te is set, the processor generates a mark trace fault any time it encounters a **mark** instruction. **fmark** causes the processor to generate a mark trace fault regardless of whether or not mark trace mode is enabled, provided PC.te is set. If PC.te is clear, **mark** and **fmark** behave like no-ops.

10.2.7.2 Hardware Breakpoints

The hardware breakpoint registers are provided to enable generation of trace faults on instruction execution and data access.

The 80303 I/O processor implements two instruction and two data address breakpoint registers, denoted IPB0, IPB1, DAB0 and DAB1. The instruction and data address breakpoint registers are 32-bit registers. The instruction breakpoint registers cause a break *after* execution of the target instruction. The DABx registers cause a break *after* memory access has been issued to the bus controller.

Hardware breakpoint registers may be armed or disarmed. When registers are armed, hardware breakpoints can generate an architectural trace fault. When registers are disarmed, no action occurs, and execution continues normally. Since instructions are always word aligned, the two low-order bits of IPBx registers act as control bits. Control bits for DABx registers reside in the Breakpoint Control (BPCON) register. BPCON enables data address breakpoint registers, and sets specific modes of these registers. Hardware breakpoints are globally enabled by the process controls trace enable bit (PC.te).

The IPBx, DABx, and BPCON registers may be accessed using normal load and store instructions (except for loads from IPBx register). The application must be in supervisor mode for a legal access to occur. See Section 3.3, "Memory-Mapped Control Registers (MMRs)" on page 3-6 for more information on the address for each register.

Applications must request modification rights to the hardware breakpoint resources, before attempting to modify these resources. Rights are requested by executing the **sysctl** instruction, as described in the following section.

10.2.7.3 Requesting Modification Rights to Hardware Breakpoint Resources

Application code must always first request and acquire modification rights to the hardware breakpoint resources before any attempt is made to modify them. This mechanism is employed to eliminate simultaneous usage of breakpoint resources by emulation tools and application code. An emulation tool exercises supervisor control over breakpoint resource allocation. If the emulator retains control of breakpoint resources, none are available for application code. If an emulation tool is not being used in conjunction with the device, modification rights to breakpoint resources are granted to the application. The emulation tool may relinquish control of breakpoint resources to the application.

If the application attempts to modify the breakpoint or breakpoint control (BPCON) registers without first obtaining rights, an OPERATION.UNIMPLEMENTED fault is generated. In this case, the breakpoint resource are not modified, whether accessed through a **sysctl** instruction or as a memory-mapped register.

Application code requests modification rights by executing the **sysctl** instruction and issuing the Breakpoint Resource Request message (*src1*.Message_Type = 06H). In response, the current available breakpoint resources are returned as the *src/dst* parameter (*src/dst* must be a register). The *src2* parameter is not used. Results returned in the *src/dst* parameter must be interpreted as shown in Table 10-2.

Table 10-2. *src/dst* Encoding

***src/dst* 7:4**	***src/dst* 3:0**
Number of Available Data Address Breakpoints	Number of Available Instruction Breakpoints

NOTE: src/dst 31:8 are reserved and always return zeroes.

The following code sample illustrates the execution of the breakpoint resource request.

Example 10-1. Execution of Breakpoint Resource Reques

```
ldconst 0x600, r4          # Load the Breakpoint Resource
                           # Request message type into r4.
sysctl r4, r4, r4          # Issue the request.
```

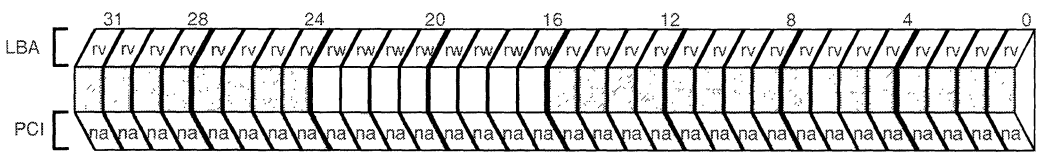
Assume in this example that after execution of the **sysctl** instruction, the value of r4 is 0000 0022H. This indicates that the application has gained modification rights to both instruction and both data address breakpoint registers. If the value returned is zero, the application has not gained the rights to the breakpoint resources.

Because the 80303 I/O processor does not initialize the breakpoint registers from the control table during initialization (as i960 Cx processors do), the application must explicitly initialize the breakpoint registers in order to use them once modification rights have been granted by the **sysctl** instruction.

10.2.7.4 Breakpoint Control Register – BPCON

The format of the BPCON registers are shown in Table 10-3 and Table 10-6. Each breakpoint has four control bits associated with it: two mode and two enable bits. The enable bits (DABx.e0, DABx.e1) in BPCON act to enable or disable the data address breakpoints, while the mode bits (DABx.m0, DABx.m1) dictate which type of access generates a break event.

Table 10-3. Breakpoint Control Register – BPCON



LBA: 8440H **PCI:** NA		**Legend:** NA = Not Accessible, RO = Read Only, RV = Reserved, PR = Preserved, RW = Read/Write, RS = Read/Set, RC = Read Clear, LBA = 80303 local bus address, PCI = PCI Configuration Address Offset
31:24	00H	Reserved. Initialize to 0.
21	0_2	DAB1 Breakpoint Enable Control Bit: DAB1.e1
20	0_2	DAB1 Breakpoint Enable Control Bit: DAB1.e0
19	0_2	DAB0 Breakpoint Mode Control Bit: DAB0.m1
18	0_2	DAB0 Breakpoint Mode Control Bit: DAB0.m0
17	0_2	DAB0 Breakpoint Enable Control Bit: DAB0.e1
16	0_2	DAB0 Breakpoint Enable Control Bit: DAB0.e0

Programming the BPCON register is summarized in Table 10-4 and Table 10-5.

Table 10-4. Configuring Data Address Breakpoint Registers – DABx

PC.te	DABx.e1	DABx.e0	Description
0	X	X	No action. With PC.te clear, breakpoints are globally disabled.
X	0	0	No action. DABx is disabled.
1	0	1	Reserved.
1	1	0	Reserved.
1	1	1	Generate a Trace Fault.

NOTE: "X" = don't care. Reserved combinations must not be used.

The mode bits of BPCON control the type of access that generates a fault, trace message, or break event, as summarized in Table 10-5.

Table 10-5. Programming Data Address Breakpoint Modes – DABx

DABx.m1	DABx.m0	Mode
0	0	Break on Data Write Access Only.
0	1	Break on Data Read or Data Write Access.
1	0	Break on Data Read Access.
1	1	Break on Data Read or Data Write Access.

10.2.7.5 Data Address Breakpoint Registers – DABx

The format for the Data Address Breakpoint (DAB) registers is shown in Table 10-6. Each breakpoint register contains a 32-bit address of a byte to match on.

A breakpoint is triggered when both a data access's type and address matches that specified by BPCON and the appropriate DAB register. The mode bits for each DAB register, which are contained in BPCON (Section 10.2.7.4, "Breakpoint Control Register – BPCON" on page 10-7), qualify the access types that DAB matches. An access-type match selects that DAB register to perform address checking. An address match occurs when the byte address of any of the bytes referenced by the data access matches the byte address contained within a selected DAB.

Consider the following example. DAB0 is enabled to break on any data read access and has a value of 100FH. Any of the following instructions causes the DAB0 breakpoint to be triggered:

Example 10-2. DAB0 Breakpoint Trigger Instructions

```
ldob        0x100f,r8
ldos        0x100e,r8
ld          0x100c,r8
ld          0x100d,r8          /* even unaligned accesses */
ldl         0x1008,r8
ldq         0x1000,r8
```

Note that the instruction:

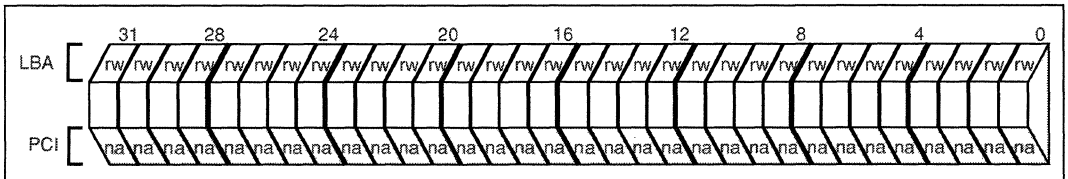
```
ldt 0x1000,r8
```

does not cause the breakpoint to be triggered because byte 100FH is not referenced by the triple word access.

Data address breakpoints can be set to break on any data read, any data write, or any data read or data write access. All accesses qualify for checking. These include explicit load and store instructions, and implicit data accesses performed by other instructions and normal processor operations.

For data accesses to the memory-mapped control register space, it is unpredictable whether breakpoint traces are generated when the access matches the breakpoints and also results in an OPERATION fault or TYPE.MISMATCH fault. The OPERATION or TYPE.MISMATCH fault is always reported in this case.

Table 10-6. Data Address Breakpoint Register – DABx

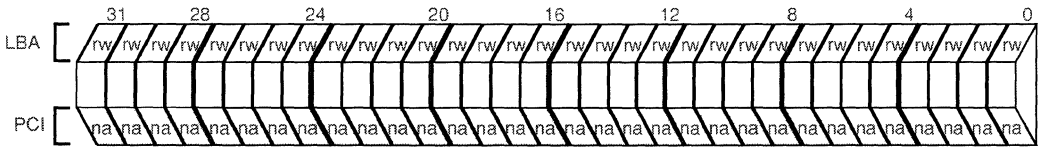


LBA: Ch 0-8420H Ch 1-8424H **PCI:** NA		**Legend:** NA = Not Accessible, RO = Read Only, RV = Reserved, PR = Preserved, RW = Read/Write, RS = Read/Set, RC = Read Clear, LBA = 80303 local bus address, PCI = PCI Configuration Address Offset
31:00	0000 0000H	Data Address.

10.2.7.6 Instruction Breakpoint Registers – IPBx

The format for the instruction breakpoint registers is given in Table 10-7. The upper 30 bits of the IPBx register contain the word-aligned instruction address on which to break. The two low-order bits indicate the action to take upon an address match.

Table 10-7. Instruction Breakpoint Register – IPBx



LBA: Ch 0-8400H Ch 1-8404H **PCI:** NA		**Legend:** NA = Not Accessible, RO = Read Only, RV = Reserved, PR = Preserved, RW = Read/Write, RS = Read/Set, RC = Read Clear, LBA = 80303 local bus address, PCI = PCI Configuration Address Offset
31:02	0000 0000H	Instruction Address.
01	0_2	IPBX Mode: IPB1
00	0_2	IPBX Mode: IPB0

Programming the instruction breakpoint register modes is shown in Table 10-8.

On the 80303 I/O processor, the instruction breakpoint memory-mapped registers can be read by using the **sysctl** instruction only. They can be modified by **sysctl** or by a word-length store instruction.

Storing directly to an IP breakpoint register may cause unexpected results if tracing is enabled. Any instructions in the superscalar template of a store operation that updates an IPB and any instructions in the subsequent superscalar template may trigger on the new or old value of the breakpoint register. The IP in the fault record may be that of the instruction that caused the breakpoint or may be the new value of the IPB register. The return IP in the fault record is always correct.

If it is necessary to avoid this condition, use the modify memory-mapped control register operation of the **sysctl** instruction to update the IPB registers.

Table 10-8. Instruction Breakpoint Modes

PC.te	IPBx.m1	IPBx.m0	Action
0	X	X	No action. Globally disabled.
X	0	0	No action. IPBx disabled.
1	0	1	Reserved.
1	1	0	Reserved.
1	1	1	Generate a Trace Fault.

NOTE: "X" = don't care. Reserved combinations must not be used.

10.3 Generating a Trace Fault

To summarize the information presented in the previous sections, the processor services a trace fault when PC.te is set and the processor detects any of the following conditions:

- An instruction included in a trace mode group executes or is about to execute (in the case of a prereturn trace event) and the trace mode for that instruction is enabled.
- A fault call operation executes and the call-trace mode is enabled.
- A **mark** instruction executes and the breakpoint-trace mode is enabled.
- An **fmark** instruction executes.
- The processor executes an instruction at an IP matching an enabled instruction address breakpoint (IPB) register.
- The processor issues a memory access matching the conditions of an enabled data address breakpoint (DAB) register.

10.4 Handling Multiple Trace Events

With the exception of a prereturn trace event, which is always reported alone, it is possible for a combination of trace events to be reported in the same fault record. The processor may not report all events; however, it always reports a supervisor event and it always signals at least one event.

If the processor reports prereturn trace and other trace types at the same time, it reports the other trace types in a single trace fault record first, and then services the prereturn trace fault upon return from the other trace fault.

10.5 Trace Fault Handling Procedure

The processor calls the trace fault handling procedure when it detects a trace event. See Section 9.7, "Fault Handling Procedures" on page 9-15 for general requirements for fault handling procedures. A trace fault handler must be invoked with an implicit system-supervisor call, this differs from other fault handling procedures. When the call is made, the processor clears the PC register trace enable bit (PC.te), disabling trace faults in the trace fault handler. Recall that for all other implicit or explicit system-supervisor calls, the processor replaces the trace enable bit with the system procedure table trace control bit. Clearing PC.te ensures that tracing is turned off when a trace fault handling procedure is being executed, thus preventing an endless loop of trace fault handling calls.

The processor calls the trace fault handling procedure when it detects a trace event. See Section 9.7, "Fault Handling Procedures" on page 9-15 for general requirements for fault handling procedures.

The trace fault handling procedure is involved in a specific way and is handled differently than other faults. A trace fault handler must be invoked with an implicit system-supervisor call. When the call is made, the PC register trace enable bit is cleared. This disables trace faults in the trace fault handler. Recall that for all other implicit or explicit system-supervisor calls the trace enable bit is replaced with the system procedure table trace control bit. The exception handling of trace enable for trace faults ensures that tracing is turned off when a trace fault handling procedure is being executed. This is necessary to prevent an endless loop of trace fault handling calls.

10.5.1 Tracing and Interrupt Procedures

When the processor invokes an interrupt handling procedure to service an interrupt, it disables tracing. It does this by saving the PC register's current state in the interrupt record, then clearing the PC register trace enable bit.

On returning from the interrupt handling procedure, the processor restores the PC register to the state it was in prior to handling the interrupt, which restores the trace enable bit. See Section 10.5.2.2, "Tracing on Implicit Call" on page 10-12 and Section 10.5.2.5, "Tracing on Return from Implicit Call: Interrupt Case" on page 10-13 for detailed descriptions of tracing on calls and returns from interrupts.

10.5.2 Tracing on Calls and Returns

During call and return operations, the trace enable flag (PC.te) may be altered. This section discusses how tracing is handled on explicit and implicit calls and returns.

Since all trace faults (except prereturn) are serviced after execution of the traced instruction, tracing on calls and returns is controlled by the PC.te in effect after the call or the return.

10.5.2.1 Tracing on Explicit Call

Tracing an explicit call happens before execution of the first instruction of the procedure called.

Tracing is not modified by using a **call** or **callx** instruction. Further, tracing is not modified by using a **calls** instruction from supervisor mode. When **calls** is issued from user mode, PC.te is read from the supervisor stack pointer trace enable bit (SSP.te) of the system procedure table, which is cached on chip during initialization. The trace enable bit in effect before the **calls** is stored in the new PFP[0] bit and is restored upon return from the routine (Section 10.5.2.3, "Tracing on Return from Explicit Call" on page 10-13). The **calls** instruction and all instructions of the procedure called are traced according to the new PC.te.

Table 10-9. Tracing on Explicit Call

Call Type	Calling Procedure Trace Enable	Calling Procedure Mode	Saved PFP.rt2:0	Called Procedure Trace Enable Bit
call, callx	PC.te	user or supervisor	000_2	PC.te
calls	PC.te	supervisor	000_2	PC.te
calls	PC.te	user	$01t_2$ Stores PC.te into bit 0 of PFP.rt2:0	SSP.te

Refer to Table 7-2, "Encoding of Return Status Field" on page 7-20).

10.5.2.2 Tracing on Implicit Call

Tracing on an implicit call happens before execution of the first instruction of the non-trace fault handler called. Table 10-10 summarizes all cases of tracing on implicit call. In the table, "a" is a bit variable that symbolizes the trace enable bit in PC.

Table 10-10 summarizes all cases.

Table 10-10. Tracing on Implicit Call

Call Type	System Procedure Table Entry	Previous Frame Pointer Return Status (PFP.rt2:0)	Source PC.te	Target PC.te	PC.te Value Used for Traces on Implicit Call
00-Fault[1]	N.A.	001	a[2]	a	a
10-Fault[1]	00	001	a	a	a
10-Fault[1]	10	001	a	SSP.te	SSP.te
00-Parallel/Override Fault 00-Trace Fault	x[2]	Type of trace fault not supported			
10-Parallel/Override Fault 10-Trace Fault	00	Type of trace fault not supported			
10-Parallel/Override Fault 10-Trace Fault	10	001	a	0	0
Interrupt	N.A.	111	a	0	0

1. On 80303 I/O processor, all faults except parallel/override and trace faults.
2. "a" and "x" are bit variables.

Tracing is not altered on the way to a local or a system-local fault handler, so the call is traced if PC.te and TC.c are set before the call. For an implicit system-supervisor call, PC.te is read from the Supervisor Stack Pointer enable bit (SSP.te). The trace on the call is serviced before execution of the first instruction of the non-trace fault handler (tracing is disabled on the way to a trace fault handler).

On the 80303 I/O processor, the parallel/override fault handler must be accessed through a system-supervisor call. Tracing is disabled on the way to the parallel/override fault handler.

The only type of trace fault handler supported is the system-supervisor type. Tracing is disabled on the way to the trace fault handler.

Tracing is disabled by the processor on the way to an interrupt handler, so an interrupt call is never traced.

Note that the Fault IP field of the fault record is not defined when tracing a fault call, because there is no instruction pointer associated with an implicit call.

10.5.2.3 Tracing on Return from Explicit Call

Table 10-11 shows all cases.

Table 10-11. Tracing on Return from Explicit Call

PFP.rt2:0	Execution Mode PC.em	Trace Enable Used for Trace on Return
000_2	user or supervisor	PC.te
$01a_2$	user	PC.te
$01a_2$	supervisor	t_2 (from PFP.r2:0)

Refer to Table 7-2, "Encoding of Return Status Field" on page 7-20.

For a return from local call (return type 000), tracing is not modified. For a return from system call (return type 01a, with PC.te equal to "a" before the call), tracing of the return and subsequent instructions is controlled by "a", which is restored in the PC.te during execution of the return.

10.5.2.4 Tracing on Return from Implicit Call: Fault Case

When the processor detects several fault conditions on the same instruction (referred to as the "target"), the non-trace fault is serviced first. Upon return from the non-trace fault handler, the processor services a trace fault on the target if in supervisor mode before the return and if the trace enable and trace-fault-pending flags are set in the PC field of the non-trace fault record (at FP-16).

If the processor is in user mode before the return, tracing is not altered. The pending trace on the target instruction is lost, and the return is traced according to the current PC.te.

10.5.2.5 Tracing on Return from Implicit Call: Interrupt Case

When an interrupt and a trace fault are reported on the same instruction, the instruction completes and then the interrupt is serviced. Upon return from the interrupt, the trace fault is serviced if the interrupt handler did not switch to user mode. On the 80303 I/O processor, the interrupt handler returns directly to the trace fault handler.

If the interrupt return is executed from user mode, the PC register is not restored and tracing of the return occurs according to the PC.te and TC.modes bit fields.

Initialization and System Requirements 11

This chapter describes the steps that the Intel® 80303 I/O processor performs during initialization. Discussed are the reset modes, the reset state and built-in self test (BIST) features. This chapter also describes the processor's basic system requirements — including power, ground and clock — and concludes with some general guidelines for high-speed circuit board design.

11.1 Overview

The 80303 I/O processor initialization can basically be separated into two steps: initialization of the Intel® i960® core processor and initialization of all of the other units. Four initialization modes are available; the selected mode is determined by the values of the **RST_MODE#** and RETRY signals when **P_RST#** is asserted. These modes dictate when the i960 core processor initializes and when the primary PCI interface accepts transactions.

Many of the 80303 I/O processor functional units require initialization before system operation. The order in which they are initialized is important and is dependent on the system design. There is no one single initialization process for the 80303 I/O processor. Instead, there are several options that may be considered.

Note: Sample initialization code, technical notes and other developer resources are available on the Intel World Wide Web site at: http://www.intel.com.

11.1.1 Core Initialization

When the i960 core processor initialization begins, the processor uses an Initial Memory Image (IMI) to establish its state. The IMI includes:

- Initialization Boot Record (IBR) – contains the addresses of the first instruction of the user's code and the PRCB.
- Process Control Block (PRCB) – contains pointers to system data structures; also contains information used to configure the processor at initialization.
- System data structures – the processor caches several data structure pointers internally at initialization.

Software can reinitialize the processor. When a reinitialization takes place, a new PRCB and reinitialization instruction pointer are specified. Reinitialization is useful for relocating data structures from ROM to RAM after initialization.

11.1.2 General Initialization

The 80303 I/O processor supports several facilities to assist in system testing and start-up diagnostics. ONCE mode electrically removes the processor from a system. This feature is useful for system-level testing where a remote tester exercises the processor system. The 80303 I/O processor also supports JTAG boundary scan (Chapter 24, "Test Features"). During initialization, the processor performs an internal functional self test and local bus self test. These features are useful for system diagnostics to ensure basic CPU and system bus functionality.

The processor is designed to minimize the requirements of its external system. It requires an input clock and clean power and ground connections (V_{SS} and V_{CC}). Since the processor can operate at a high frequency, the external system must be designed with considerations to reduce induced noise on signals, power and ground.

11.2 Intel® 80303 I/O Processor Initialization

Several functional units within the 80303 I/O processor must be initialized before system operation. These are the PCI-to-PCI Bridge, Address Translation Unit (ATU), i960 core processor, Memory Controller, and Secondary PCI Bus Arbiter. The order in which they are initialized is dependent on how the 80303 I/O processor is used in the system. The initialization process begins when the Primary PCI Bus Reset signal (**P_RST#**) is asserted.

11.2.1 Initialization Modes

The initialization process is generally controlled through either an external host processor or the i960 core processor. Based on this assumption, there are four initialization modes.

The mode is determined by the value of the **RST_MODE#** and **RETRY** signals, described in the next sections. Table 11-1 describes the relationship between the **RST_MODE#** and **RETRY** signal values and the initialization mode.

Table 11-1. Initialization Modes

RST_MODE#	RETRY	Initialization Mode	Primary PCI Interface	Intel® i960® Core Processor
0	0	Mode 0	Accepts Transactions	Held in Reset
0	1	Mode 1	Retries All Configuration Transactions	Held in Reset
1	0	Mode 2	Accepts Transactions	Initializes
1	1	Mode 3 (default)	Retries All Configuration Transactions	Initializes

The **RST_MODE#** signal is sampled on the rising edge of **P_RST#**. The inverse value of this signal is then written to the Core Processor Reset bit in the Extended Bridge Control Register (EBCR). See Chapter 14, "PCI-to-PCI Bridge Unit". When **RST_MODE#** is active and **P_RST#** is asserted, the i960 core processor is held in reset until **P_RST#** is deasserted. The i960 core processor reset is released when the reset bit in EBCR is cleared. When **RST_MODE#** is inactive and **P_RST#** is asserted, the i960 core processor is reset. The i960 core processor then begins its normal initialization sequence when **P_RST#** is deasserted.

The RETRY signal is sampled on the rising edge of **P_RST#**. The value of this signal is written to the Configuration Cycle Disable bit in the EBCR. When RETRY is active and **P_RST#** is de-asserted, the 80303 I/O processor signals a Retry on all PCI configuration cycles it receives on the primary PCI bus. When RETRY is inactive and **P_RST#** is de-asserted, the 80303 I/O processor accepts PCI configuration cycles on the primary PCI bus.

Figure 11-1 shows a flow chart of the initialization process.

11.2.2 Mode 0 Initialization

Mode 0 allows a host processor to configure the 80303 I/O processor peripherals while the i960 core processor is held in reset. The host processor configures the PCI-to-PCI Bridge by assigning bus numbers, allocating PCI address space, and assigning IRQ numbers. The memory controller and ATU can also be initialized by the host processor. Program code for the i960 core processor may be downloaded into local memory by the host processor.

The host processor clears the 80303 reset signal by clearing the Core Processor Reset bit in the EBCR. This deasserts the internal reset signal on the i960 core processor and the processor begins its initialization process.

11.2.3 Mode 1 Initialization

Intel does not recommend the use of Mode 1 initialization.

11.2.4 Mode 2 Initialization

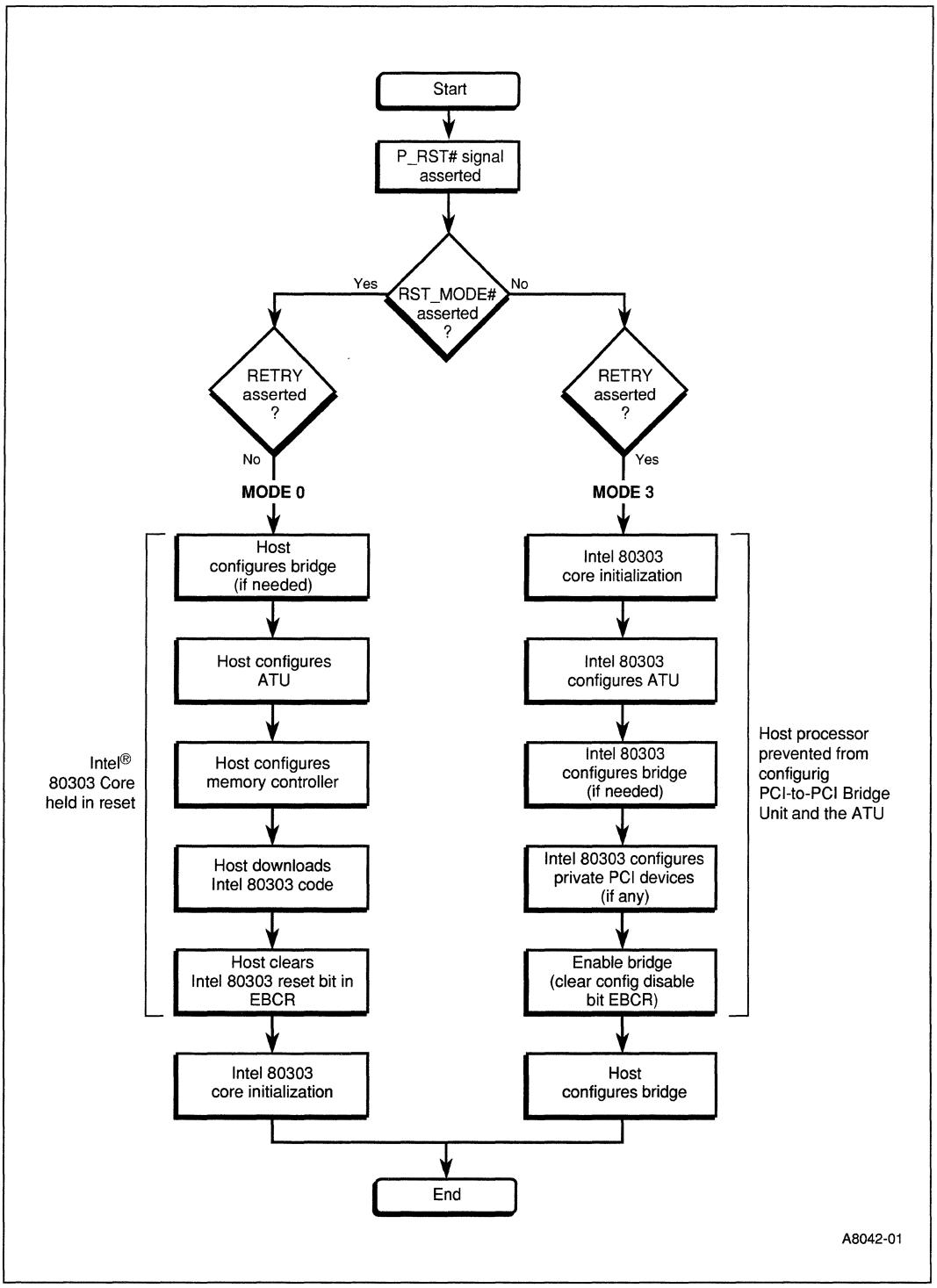
Intel does not recommend the use of Mode 2 initialization.

11.2.5 Mode 3 (Default Mode)

Mode 3 allows the i960 core processor to initialize and control the initialization process before the host processor is allowed to configure the 80303 I/O processor peripherals. During this time, the primary PCI interface signals a Retry on all configuration cycles it receives until the i960 core processor clears the Configuration Cycle Disable bit in the EBCR. This option is only available when an initialization ROM is used.

By allowing the i960 core processor to control the initialization process, it is possible to initialize the PCI configuration registers to values other than the default power-up values. Certain PCI configuration registers that are read only through PCI configuration cycles are read/write from the i960 core processor. This allows the programmer to customize the way the 80303 I/O processor appears to the PCI configuration software.

Figure 11-1. Initialization Flow Chart



11.2.6 Secondary PCI Bus Arbitration Unit

After reset, all devices controlled by the secondary PCI Bus Arbiter are set to low priority, except for the secondary PCI interface of the 80303, which is set to high priority.

The secondary bus arbiter is reset by the S_RST# signal on the secondary interface. Whenever the secondary bus is reset, the secondary arbiter is reset moving all devices to their programmed priority levels and starting the round robin arbitration sequence on the lowest number device at each priority level.

11.2.7 Internal Bus Arbitration Unit

The internal bus arbitration logic is reset by the **P_RST#** signal. The reset values of the registers are shown in Table 11-2. All bus masters are initialized to the highest priority. None of the devices are disabled at powerup.

Table 11-2. Reset Values

Internal Arbitration Register	Reset Value	Note
Internal Arbitration Control Register (IACR)	0000 0000H	All Bus Masters Enabled
Master Latency Timer Register (MLTR)	0000 0FFFH	Maximum Count Value
Multi-Transaction Timer Register (MTTR)	0000 0000H	Disabled

11.2.8 Reset State Operation

The **P_RST#** signal, when asserted, causes the 80303 I/O processor to enter the reset state. All external signals go to a defined state, internal logic is initialized, and certain registers are set to defined values.

P_RST# must be asserted when power is applied to the processor. The processor then stabilizes in the reset state. This power-up reset is referred to as *cold reset.* To ensure that all internal logic has stabilized in the reset state, a valid input clock (**S_CLK**) and V_{CC} must be present and stable for a specified time before **P_RST#** can be deasserted.

The processor may also be cycled through the reset state after execution has started. This is referred to as *warm reset*. For a warm reset, **P_RST#** must be asserted for a minimum number of clock cycles. Specifications for a cold and warm reset can be found in the *80960RM I/O Processor Data Sheet* and the *80960RN I/O Processor Data Sheet.*

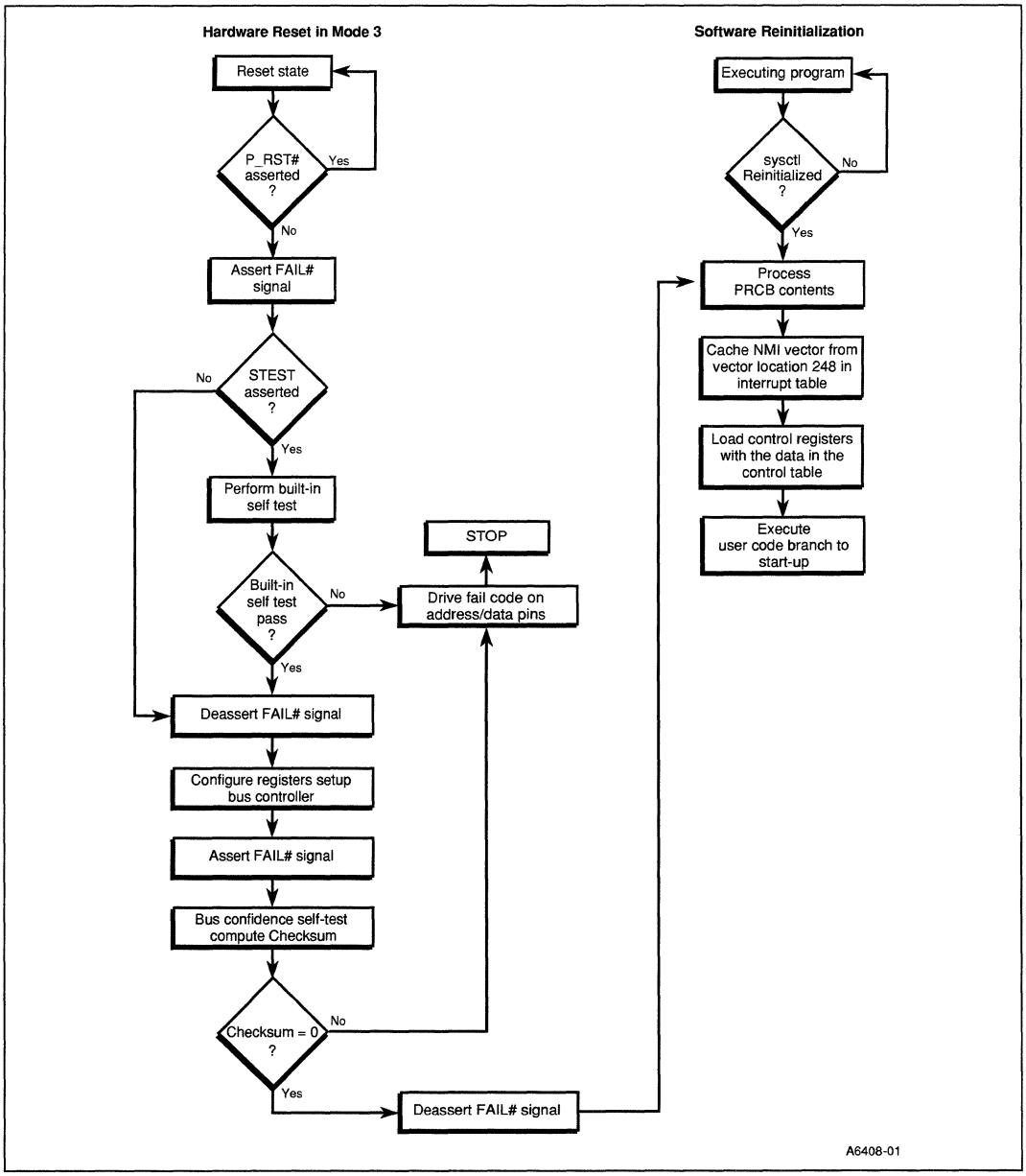
User software cannot reset the entire 80303 I/O processor; however, the **sysctl** instruction can reset the i960 core processor. The **P_RST#** signal must be asserted to enter the reset state. See Section 11.6, "Reinitializing and Relocating Data Structures" on page 11-21.

11.3 Intel® i960® Core Processor Initialization

Initialization describes the mechanism that the processor uses to establish its initial state and begin instruction execution. When i960 core processor initialization begins, the processor automatically configures itself with information specified in the IMI and performs its built-in self test based on the sampling of the **STEST** signal. The processor then branches to the first instruction of user code. See Figure 11-2 for a flow chart of i960 core processor initialization.

The objective of the initialization sequence is to provide a complete, working initial state when the first user instruction executes. The user's start-up code needs only to perform several basic functions to place the processor in a configuration for executing application code.

Figure 11-2. Processor Initialization Flow



11.3.1 Self Test Function (STEST, FAIL#)

As part of initialization, the 80303 I/O processor executes a local bus confidence self test, an alignment check for data structures within the initial memory image (IMI), and optionally, a built-in self test program. The self test (**STEST**) signal enables or disables built-in self test. The **FAIL#** signal indicates that the self tests failed by asserting **FAIL#**. During normal operations the **FAIL#** signal can be asserted when a core processor error is detected. The following subsections further describe these signal functions.

Built-in self test checks basic functionality of internal data paths, registers and memory arrays on-chip. Built-in self test is not intended to be a full validation of processor functionality; it is intended to detect catastrophic internal failures and complement a user's system diagnostics by ensuring a confidence level in the processor before any system diagnostics are executed.

Note: BIST applies only to the i960 core processor.

11.3.1.1 The STEST Signal

The **STEST** signal enables and disables Built-In Self Test (BIST). BIST can be disabled when the initialization time needs to be minimized or when diagnostics are simply not necessary. The **STEST** signal is sampled under the following conditions:

- On the rising edge **P_RST#**
- On the rising edge of reset mode (**RST_MODE#**), if used.
- On the rising edge of na internal bus reset (initiated after the Reset Internal Bus bit in the Extended Bridge Control Register (EBCR) is set).

When **STEST** is asserted, the i960 core processor executes the built-in self test. When **STEST** is deasserted, the i960 core processor bypasses built-in self test.

11.3.1.2 Intel® i960® Local Bus Confidence Test

The local bus confidence test is always performed regardless of **STEST** signal value. The local bus confidence test reads eight words from the Initialization Boot Record (IBR) and performs a checksum on the words and the constant FFF FFFFH. The test passes only when the processor calculates a sum of zero (0). The test can detect catastrophic bus failures such as external address, data or control lines that are stuck, shorted or open.

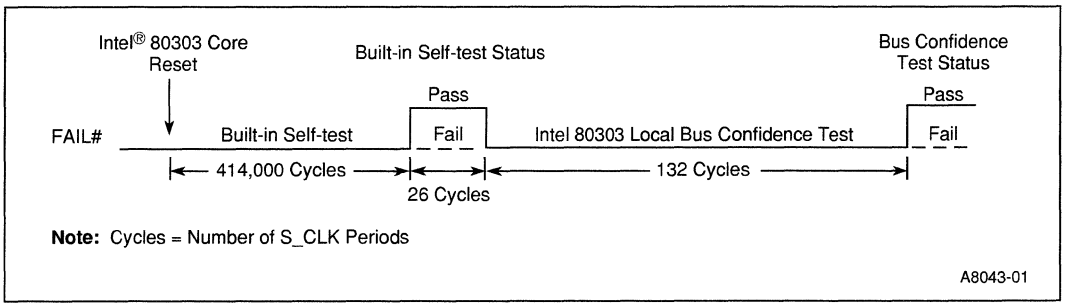
11.3.1.3 The Fail Signal (FAIL#)

The **FAIL#** signal signals errors in either the built-in self test or the bus confidence self test. **FAIL#** is asserted (low) for each self test (Figure 11-3):

- When any test fails, the **FAIL#** signal remains asserted, a fail code message is driven onto the address bus, and the processor stops execution at the point of failure.
- When a core processor error occurs, **FAIL#** is also asserted. See Section 11.3.1.4, "IMI Alignment Check and Core Processor Error" on page 11-8 for details.
- When the test passes, **FAIL#** is deasserted.

When **FAIL#** stays asserted, the only way to resume normal operation is to perform a reset operation. When the **STEST** signal is used to disable the built-in self test, the test does not execute; however, **FAIL#** still asserts at the point where the built-in self test would occur. **FAIL#** is deasserted after the bus confidence test passes. In Figure 11-3, all transitions on the **FAIL#** signal are relative to **S_CLK** as described in the *Intel®* 80960RM I/O Processor Datasheet and the *Intel®* 80960RN I/O Processor Datasheet.

Figure 11-3. FAIL# Timing



11.3.1.4 IMI Alignment Check and Core Processor Error

The alignment check during initialization for data structures within the IMI ensures that the PRCB, control table, interrupt table, system-procedure table, and fault table are aligned to word boundaries. Normal processor operation is not possible without the alignment of these key data structures. The alignment check is one case where a core processor error could occur.

The other case of core processor error can occur during regular operation when generation of an override fault incurs a fault. The sequence of events leading up to this case is quite uncommon.

When a core processor error is detected, the **FAIL#** signal is asserted, a fail code message is driven onto the address bus, and the processor stops execution at the point of failure. The only way to resume normal operation of the processor is to perform a reset operation. Because core processor error generation can occur sometime after the Bus confidence test and even after initialization during normal processor operation, the **FAIL#** signal is a logic one before the detection of a Core Processor Error.

11.3.1.5 FAIL# Code

The processor uses only one read bus transaction to signal the fail code message; the address of the bus transaction is the fail code itself. The fail code is of the form: 0xFEFFFFnn; bits 6 to 0 contain a mask recording the possible failures. Bit 7, when one, indicates the mask contains failures from Built-In Self-Test (BIST); when zero, the mask indicates other failures. The fail codes are shown in Table 11-3 and Table 11-4.

Table 11-3. BIST Failure Codes

Bit	When Set
7	Set to one for BIST failure
6	On-chip Data-RAM failure detected by BIST
5	Internal Microcode ROM failure detected by BIST
4	I-cache failure detected by BIST
3	D-cache failure detected by BIST
2	Local-register cache or processor core failure detected by BIST
1	Always Zero
0	Always Zero

Table 11-4. Non-BIST Failure Codes

Bit	When Set
7	Set to zero for non-BIST failure
6	Always One; this bit does not indicate a failure
5	Always One; this bit does not indicate a failure
4	A data structure within the IMI is not aligned to a word boundary
3	A core processor error during normal operation has occurred
2	The Bus Confidence test has failed
1	Always Zero
0	Always Zero

11.4 Initial Memory Image (IMI)

The IMI comprises the minimum set of data structures that the processor needs to initialize. As shown in Figure 11-4, these structures are: the initialization boot record (IBR), process control block (PRCB) and system data structures. The IBR is located at a fixed address in memory. The other components are referenced directly or indirectly by pointers in the IBR and the PRCB. The IMI performs three functions for the processor:

- Provides initial configuration information for the core.
- Provides pointers to the system data structures and the first instruction to be executed after processor initialization.
- Provides checksum words that the processor uses in its self test routine at startup.

Several data structures are typically included as part of the IMI because values in these data structures are accessed by the processor during initialization. These data structures are usually programmed in the systems's boot ROM, located in memory region 14_15 of the address space. The required data structures are:

- PRCB
- IBR
- System procedure table
- Control table
- Interrupt table
- Fault table

To ensure proper processor operation, the PRCB, system procedure table, control table, interrupt table, and fault table must not be located in architecturally reserved memory – addresses reserved for on-chip Data RAM and addresses at and above FEFF FF60H. In addition, each of these structures must start at a word-aligned address; a core processor error occurs when any of these structures are not word-aligned. See Section 11.3.1.3, "The Fail Signal (FAIL#)" on page 11-8.

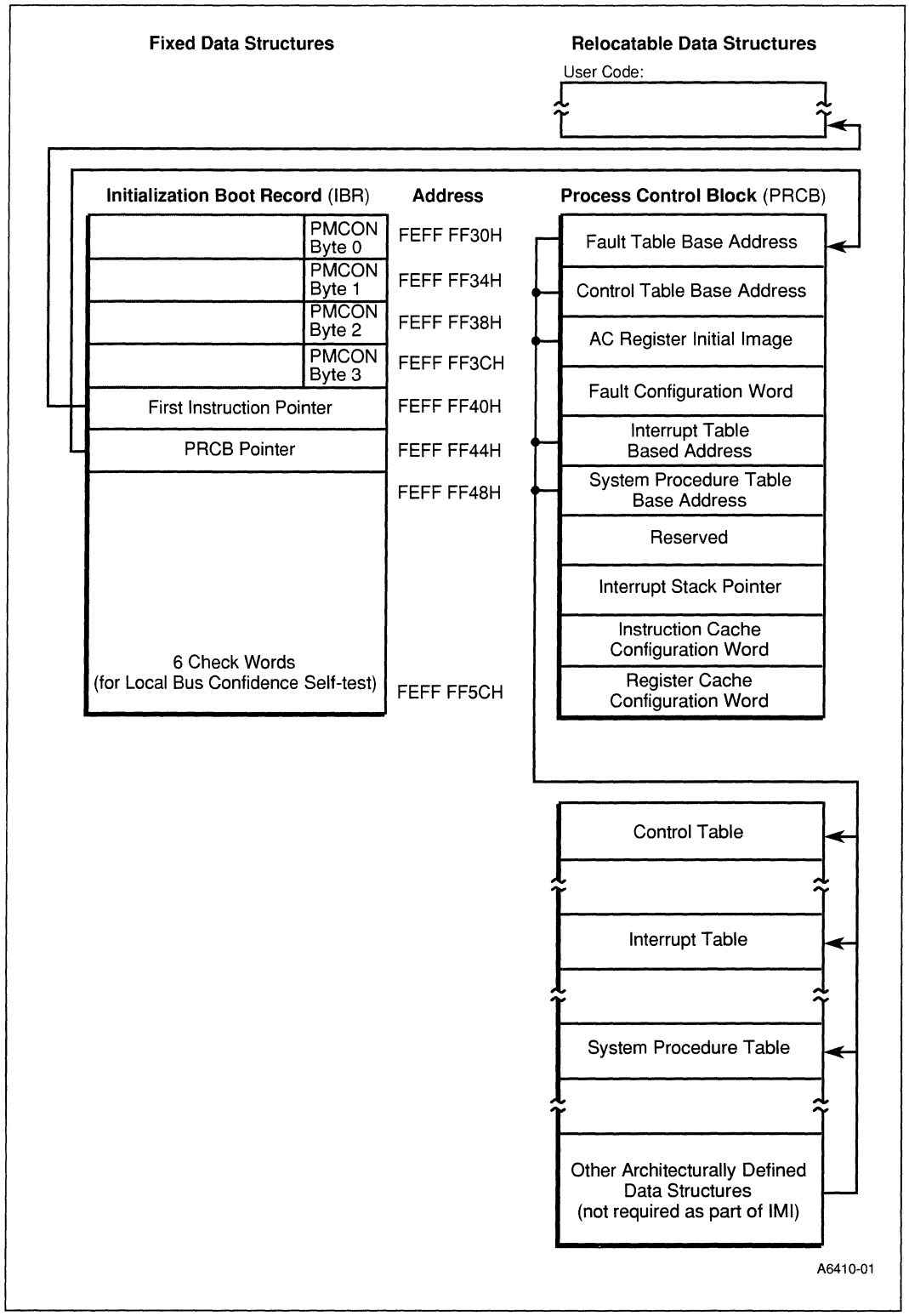
At initialization, the processor loads the Supervisor Stack Pointer (SSP) from the system procedure table, aligns it to a 16-byte boundary, and caches the pointer in the SSP memory-mapped control register. Recall that the supervisor stack pointer is located in the preamble of the system procedure table at byte offset 12 from the base address. The system procedure table base address is programmed in the PRCB. Consult Section 7.5.1, "System Procedure Table" on page 7-15 for the format of the system procedure table.

At initialization, the NMI vector is loaded from the interrupt table and saved at location 0000 0000H of the internal data RAM. The interrupt table is typically programmed in the boot ROM and then relocated to internal RAM by reinitializing the processor.

The fault table is typically located in boot ROM. When it is necessary to locate the fault table in RAM, the processor must be reinitialized.

The remaining data structures that an application may need are the user stack, supervisor stack and interrupt stack. These stacks must be located in the 80303 I/O processor's local bus RAM.

Figure 11-4. Initial Memory Image (IMI) and Process Control Block (PRCB)



11.4.1 Initialization Boot Record (IBR)

The initialization boot record (IBR) is the primary data structure required to initialize the 80303 I/O processor processor. The IBR is a 12-word structure which must be located at address FEFF FF30H (Table 11-5). The IBR is made up of four components: the initial bus configuration data, the first instruction pointer, the PRCB pointer and the bus confidence test checksum data.

Table 11-5. Initialization Boot Record

Byte Physical Address	Description
FEFF FF30H	PMCON14_15, byte 0 (Program to 0000 0000H) for 80303 I/O processor
FEFF FF31H to FEFF FF33H	Reserved
FEFF FF34H	PMCON14_15, byte 1 (Program to 0000 0000H) for 80303 I/O processor
FEFF FF35H to FEFF FF37H	Reserved
FEFF FF38H	PMCON14_15, byte 2 (Program to 0000 0080H) for 80303 I/O processor
FEFF FF39H to FEFF FF3BH	Reserved
FEFF FF3CH	PMCON14_15, byte 3 (Program to 0000 0000H) for 80303 I/O processor
FEFF FF3DH to FEFF FF3FH	Reserved
FEFF FF40H to FEFF FF43H	First Instruction Pointer
FEFF FF44H to FEFF FF47H	PRCB Pointer
FEFF FF48H to FEFF FF4BH	Bus Confidence Self-Test Check Word 0
FEFF FF4CH to FEFF FF4FH	Bus Confidence Self-Test Check Word 1
FEFF FF50H to FEFF FF53H	Bus Confidence Self-Test Check Word 2
FEFF FF54H to FEFF FF57H	Bus Confidence Self-Test Check Word 3
FEFF FF58H to FEFF FF5BH	Bus Confidence Self-Test Check Word 4
FEFF FF5CH to FEFF FF5FH	Bus Confidence Self-Test Check Word 5

When the processor reads the IMI during initialization, it must know the bus characteristics of external memory where the IMI is located. Specifically, it must know the bus width and endianism for the remainder of the IMI. At initialization, the processor sets the PMCON register to an 8-bit bus width. The processor then needs to form the initial DLMCON and PMCON14_15 registers so that the memory containing the IBR can be accessed correctly. The lowest-order byte of each of the IBRs first 4 words are used to form the register values. On the 80303 I/O processor, the bytes at FEFF FF30H and FEFF FF34H are not needed, so the processor starts fetching at address FEFF FF38. The loading of these registers is shown in the pseudo-code flow in Example 11-1.

Note: The 80303 I/O processor requires that all PMCON registers be programmed for 32-bit bus widths.

Example 11-1. Processor Initialization Pseudocode Flow

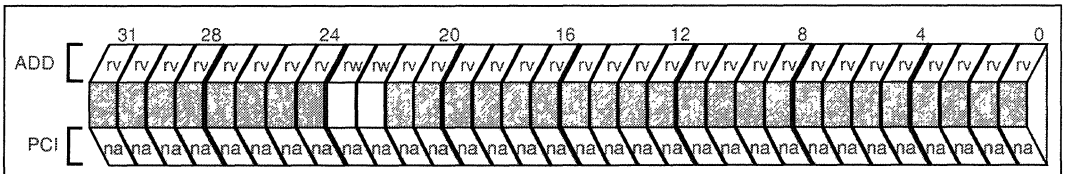
```
Processor_Initialization_flow ( )

{
FAIL_pin = true;
restore_full_cache_mode; disable(I_cache); invalidate(I_cache);
disable(D_cache); invalidate(D_cache);
BCON.ctv = 0; /* Selects PMCON14_15 to control all accesses */
PMCON14_15 = 0; /* Selects 8-bit bus width */
/** Exit Reset State & Start_Init **/
        if (STEST_ON_RISING_EDGE_OF_RESET)
                                    status = BIST();  /* BIST does not return if it fails */
        FAIL_pin = false;
        PC = 0x001f2002;            /* PC.Priority = 31, PC.em = Supervisor,*/
                                    /* PC.te = 0; PC.State = Interrupted    */
        ibr_ptr = 0xfeffff30;       /* ibr_ptr used to fetch IBR words  */

/* Read PMCON14_15 image in IBR */
FAIL_pin = true;                                          IMSK = 0;
DLMCON.dcen = 0;                    LMMR0.lmte = 0; LMMR1.lmte = 0;

PMCON14_15[byte2] = 0xc0 & memory[ibr_ptr +8];
/*Compute CheckSum on Boot Record */
carry = 0;                          CheckSum = 0xffffffff;
for( i = 6; i>0; i--)               /*carry is carry out from previous add*/
        CheckSum = memory[ibr_ptr + 24 + i*4] + CheckSum + carry;
prcb_ptr = memory[ibr_ptr + 0x14];
IP = memory[prcb_ptr + 4];
CheckSum = prcb_ptr + IP + CheckSum + carry;
if(CheckSum != 0)
        {fail_msg = 0xfeffff64;     /* Fail BUS Confidence Test */
        dummy = memory[fail_msg];                         /* Do load with address = fail_msg
                                                          */
        for(;;);                    /* loop forever with FAIL pin true */
        }
else    FAIL_pin = false;

/* Process PRCB and Control Table */
prcb_ptr = memory[ibr_ptr + 0x14];
Process_PRCB(prcb_ptr);             /* See Process PRCB Section for Details */

Destroy_Global_&_Local_Register_Values();                 /*Previous values of Global
                                                          and Local Registers are
                                                          Destroyed during
                                                          initialization and software re-
                                                           initialization*/
g0 = 80303core_device_ID;
return;                             /* Execute First Instruction */

}
```

The processor initializes the DLMCON.dcen bit to 0 to disable data caching. The remainder of the assembled word is used to initialize PMCON14_15. In conjunction with this step, the processor clears the bus control table valid bit (BCON.ctv), to ensure for the remainder of initialization that every bus request issued takes configuration information from the PMCON14_15 register, regardless of the memory region associated with the request. At a later point in initialization, the processor loads the remainder of the memory region configuration table from the external control table. The Bus Configuration (BCON) register is also loaded at this time. The control table valid (BCON.ctv) bit is then set in the control table to validate the PMCON registers after they are loaded. In this way, the bus controller is completely configured during initialization. (Chapter 12, "Core Processor and Internal Operation" for a complete discussion of memory regions and configuring the bus controller.)

After the bus configuration data is loaded and the new bus configuration is in place, the processor loads the remainder of the IBR which consists of the first instruction pointer, the PRCB pointer and six checksum words. The PRCB pointer and the first instruction pointer are internally cached. The six checksum words — along with the PRCB pointer and the first instruction pointer — are used in a checksum calculation which implements a confidence test of the local bus. The checksum calculation is shown in the pseudo-code flow in Example 11-2. When the checksum calculation equals zero, then the bus confidence test passes.

Table 11-6 further describes the IBR organization.

Table 11-6. PMCON14_15 Register Bit Description in IBR



ADD: 8638H **PCI:** NA		**Legend:** NA = Not Accessible, RO = Read Only, RV = Reserved, PR = Preserved, RW = Read/Write, RS = Read/Set, RC = Read Clear, ADD = 80303 internal bus address, PCI = PCI Configuration Address Offset
31:24	00H	Reserved. Initialize to 0.
23:22	00_2	Local Bus Width (BW) (00) Reserved (01) Reserved (10) 32-bit (11) Reserved
21:00	00 0000H	Reserved. Initialize to 0.

11.4.2 Process Control Block – PRCB

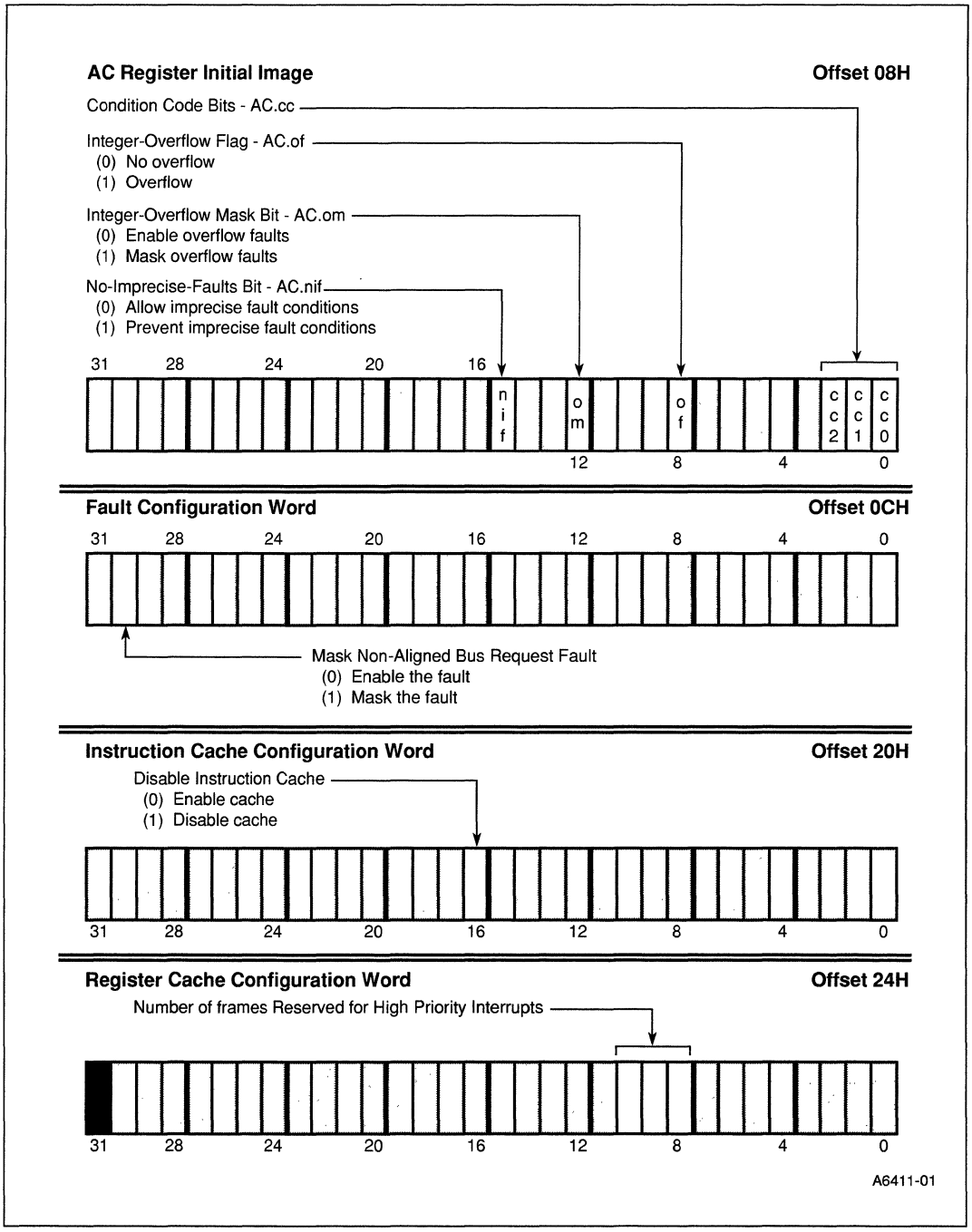
The PRCB contains base addresses for system data structures and initial configuration information for the i960 core processor. The base addresses are accessed from these internal registers. The registers are accessible to the users through the memory mapped interface. Upon reset or reinitialization, the registers are initialized. The PRCB format is shown in Table 11-7.

Table 11-7. PRCB Configuration

Physical Address	Description
PRCB POINTER + 00H	Fault Table Base Address
PRCB POINTER + 04H	Control Table Base Address
PRCB POINTER + 08H	AC Register Initial Image
PRCB POINTER + 0CH	Fault Configuration Word
PRCB POINTER + 10H	Interrupt Table Base Address
PRCB POINTER + 14H	System Procedure Table Base Address
PRCB POINTER + 18H	Reserved
PRCB POINTER + 1CH	Interrupt Stack Pointer
PRCB POINTER + 20H	Instruction Cache Configuration Word
PRCB POINTER + 24H	Register Cache Configuration Word

The initial configuration information is programmed in the arithmetic controls register (AC) initial image, the fault configuration word, the instruction cache configuration word, and the register cache configuration word. Table 11-8 show these configuration words.

Table 11-8. Process Control Block Configuration Words



11.4.3 Process PRCB Flow

The following pseudo-code flow illustrates the processing of the PRCB. Note that this flow is used for both initialization and reinitialization (through **sysctl**).

Example 11-2. PRCB Processing Pseudo-code Flow

```
Process_PRCB(prcb_ptr)
{       PRCB_mmr = prcb_ptr;
        reset_state(data_ram);              /* It is unpredictable whether the   */
                                            /* Data RAM keeps its prior contents */
        fault_table  =  memory[PRCB_mmr];
        ctrl_table   =  memory[PRCB_mmr+0x4];
        AC           =  memory[PRCB_mmr+0x8];
        fault_config =  memory[PRCB_mmr+0xc];
        if (1 & (fault_config >> 30))
generate_fault_on_unaligned_access = false;
        else    generate_fault_on_unaligned_access = true;

/** Load Interrupt Table Pointer **/
        Reset_block_NMI;
        interrupt_table =  memory[PRCB_mmr+0x10];

/** Load System Procedure Table Pointer **/
        sysproc = memory[PRCB_mmr+0x14];

/** Initialize ISP, FP, SP, and PFP **/
        ISP_mmr =  memory[PRCB_mmr+0x1c];
        FP      = ISP_mmr;
        SP      = FP + 64;
        PFP     = FP;

/** Initialize Instruction Cache **/
        ICCW = memory[PRCB_mmr+0x20];
        if (1 & (ICCW >> 16) ) enable(I_cache);

/** Cache NMI Vector Entry in Data RAM**/
        memory[0] = memory[interrupt_table + (248*4) + 4];

/** Process System Procedure Table **/
        temp    = memory[sysproc+0xc];
        SSP_mmr = (~0x3) & temp;
        SSP.te  = 1 & temp;

/** Configure Local Register Cache **/
        programmed_limit = (7 & (memory[PRCB_mmr+0x24] >> 8) );
        config_reg_cache(programmed_limit);

/** Load_control_table. Note breakpoints and BPCON are excluded here **/
        load_control_table(ctrl_table+0x10, ctrl_table+0x58);
                /* Load ctrl_table+0x10 through ctrl_table+0x58 */
        load_control_table(ctrl_table+0x68, ctrl_table+0x6c);
                /* Load ctrl_table+0x68 through ctrl_table+0x6C */
        IBP0 = 0x0; IBP1 = 0x0; DAB0 = 0x0; DAB1 = 0x0;

/** Initialize Timers **/
        TMR0.tc   = 0; TMR1.tc   = 0; TMR0.enable = 0; TMR1.enable = 0;
        TMR0.sup  = 0; TMR1.sup  = 0; TMR0.reload = 0; TMR1.reload = 0;
        TMR0.csel = 0; TMR1.csel = 0;

        return;
```

11.4.3.1 AC Initial Image

The AC initial image is loaded into the on-chip AC register during initialization. The AC initial image allows the initial value of the overflow mask, no imprecise faults bit and condition code bits to be selected at initialization.

The AC initial image condition code bits can be used to specify the source of an initialization or reinitialization when a single instruction entry point to the user start-up code is desirable. This is accomplished by programming the condition code in the AC initial image to a different value for each different entry point. The user start-up code can detect the condition code values — and thus the source of the reinitialization — by using the compare or compare-and-branch instructions.

11.4.3.2 Fault Configuration Word

The fault configuration word allows the operation-unaligned fault to be masked when an unaligned memory request is issued. When an unaligned access is encountered, the processor *always* performs the access. After performing the access, the processor determines whether it should generate a fault. When bit 30 in the fault configuration word is set, a fault is not generated after an unaligned memory request is performed. When bit 30 is clear, a fault is generated after an unaligned memory request is performed.

11.4.3.3 Instruction Cache Configuration Word

The instruction cache configuration word allows the instruction cache to be enabled or disabled at initialization. When bit 16 in the instruction cache configuration word is set, the instruction cache is disabled and all instruction fetches are directed to external memory. Disabling the instruction cache is useful for tracing execution in a software debug environment.

The instruction cache remains disabled until the following operations:

- The processor is reinitialized with a new value in the instruction cache configuration word
- **icctl** is issued with the enable instruction cache operation
- **sysctl** is issued with the configure instruction cache message type and a cache configuration mode other than disable cache.

11.4.3.4 Register Cache Configuration Word

The register cache configuration word specifies the number of free frames in the local register cache that can be used by critical code (i.e., code that is in the interrupted state and has a process priority greater than or equal to 28).

The register cache and the configuration word are explained further in Section 4.2, "Local Register Cache" on page 4-2.

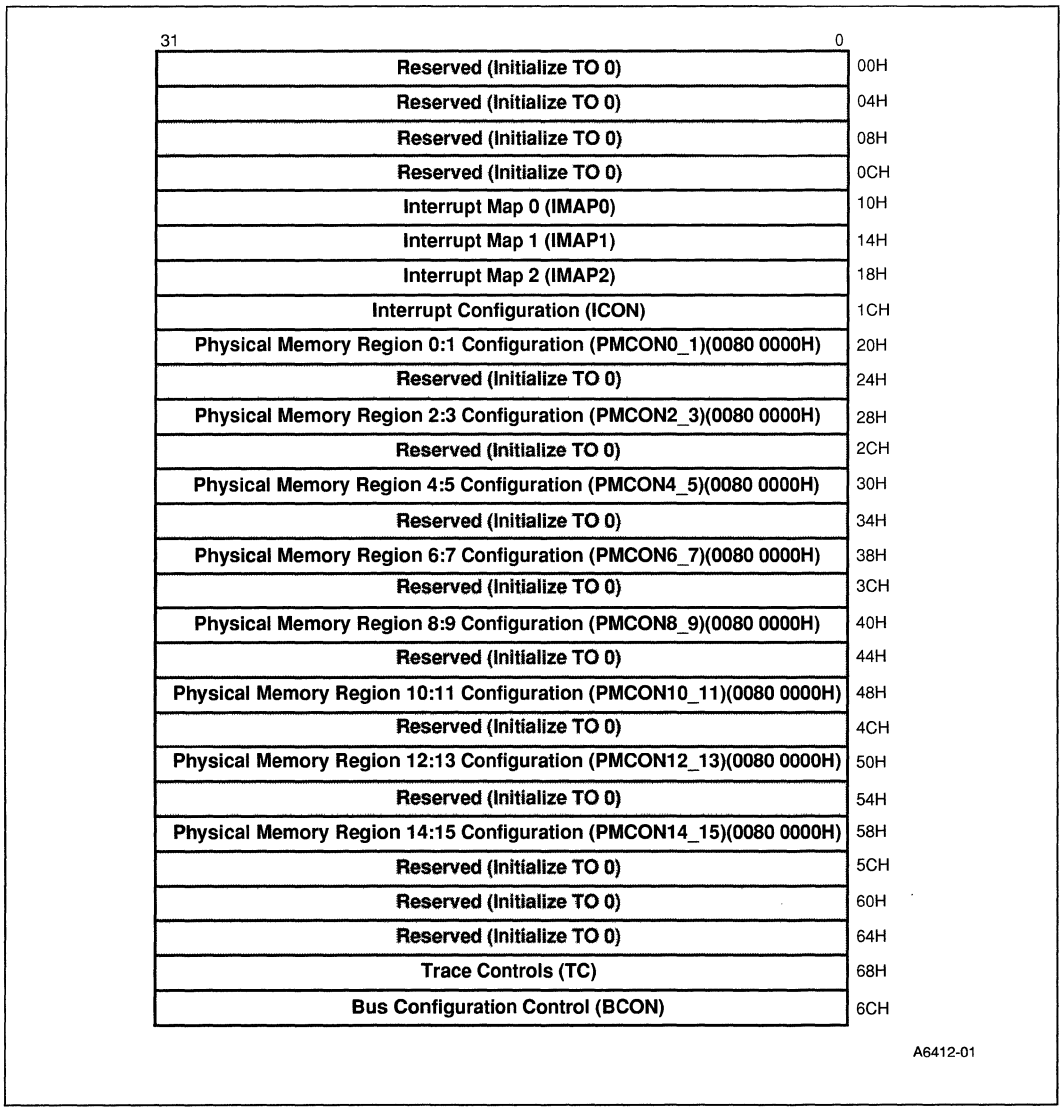
11.4.4 Control Table

The control table is the data structure that contains the on-chip control registers values. It is automatically loaded during initialization and must be completely constructed in the IMI. Figure 11-5 shows the Control Table format.

For register bit definitions of the on-chip control table registers, see the following:

- IMAP — Table 8-16 through Table 8-18, "Interrupt Map Register 2 (IMAP2)" on page 8-38
- ICON — Table 8-15, "Interrupt Control (ICON) Register" on page 8-36
- PMCON — Table 12-2, "Physical Memory Control Registers – PMCON0:15" on page 12-3
- TC — Table 10-1, "Intel® 80303 I/O Processor Trace Controls Register – TC" on page 10-2
- BCON —Table 12-3, "Bus Control Register – BCON" on page 12-4

Figure 11-5. Control Table

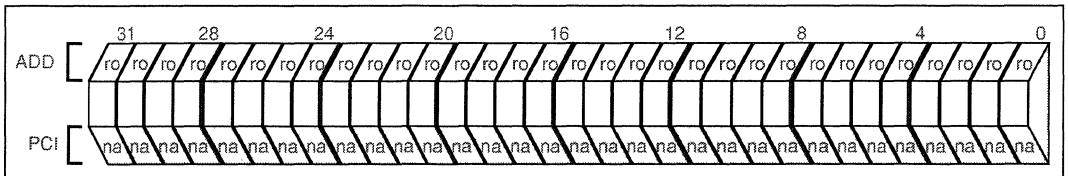


11.5 Device Identification on Reset

During the manufacturing process, values characterizing the 80303 I/O processor type and stepping are programmed into the memory-mapped registers. The 80303 I/O processor contains two read-only device ID MMRs. One holds the Processor Device ID (PDIDR) and the other holds the i960 Core Processor Device ID (DEVICEID).

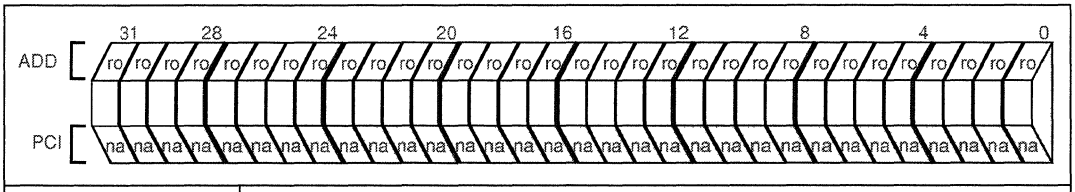
The device identification values are compliant with the IEEE 1149.1 specification and Intel standards. Table 11-9 and Table 11-10 describe the fields of the two Device IDs. During initialization, the PDIDR is placed in g0.

Table 11-9. Processor Device ID Register - PDIDR



ADD: 1710H **PCI:** NA		**Legend:**NA = Not AccessibleRO = Read Only RV = ReservedPR = PreservedRW = Read/Write RS = Read/SetRC = Read Clear ADD = 80303 internal bus addressPCI = PCI Configuration Address Offset
31:0	X	The values programmed into this register vary with stepping. Refer to the *80303 I/O Processor Specification Update (273355)* for the correct value.

Table 11-10. Intel® i960® Core Processor Device ID Register - DEVICEID



ADD: FF00 8710H **PCI:** NA		**Legend:** NA = Not Accessible, RO = Read Only, RV = Reserved, PR = Preserved, RW = Read/Write, RS = Read/Set, RC = Read Clear, ADD = 80303 internal bus address, PCI = PCI Configuration Address Offset
31:0	X	The values programmed into this register vary with stepping. Refer to the *80303 I/O Processor Specification Update (273355)* for the correct value.

11.6 Reinitializing and Relocating Data Structures

Reinitialization can reconfigure the processor and change pointers to data structures. The processor is reinitialized by issuing the **sysctl** instruction with the reinitialize processor message type. See Section 6.2.67, "sysctl" on page 6-95 for a description of **sysctl**.) The reinitialization instruction pointer and a new PRCB pointer are specified as operands to the **sysctl** instruction. When the processor is reinitialized, the fields in the newly specified PRCB are loaded as described in Section 11.4.2, "Process Control Block – PRCB" on page 11-15.

Reinitialization is useful for relocating data structures to RAM after initialization. The interrupt table must be located in RAM: to post software-generated interrupts, the processor writes to the pending priorities and pending interrupts fields in this table. It may also be necessary to relocate the control table to RAM: it must be in RAM when the control register values are to be changed by user code. In some systems, it is necessary to relocate other data structures (fault table and system procedure table) to RAM because of unsatisfactory load performance from ROM.

After initialization, the software is responsible for copying data structures from ROM into RAM. The processor is then reinitialized with a new PRCB which contains the base addresses of the new data structures in RAM.

The processor caches the following pointers during its initialization. To modify these data structures, a software re-initialization is needed.

- Interrupt Table Address
- Fault Table Address
- System Procedure Table Address
- Control Table Address

11.6.1 Output Clocks

The 80303 I/O processor supports an I^2C bus interface. The output clock frequency for I^2C operation is 100 KHz or 400 KHz. This clock is generated from the i960 core processor clock. To use the I^2C interface, a clock divider value must be written into the I^2C Clock Count Register. See Section 22.8.5, "I2C Clock Count Register- ICCR" on page 22-36.

intel®

Core Processor and Internal Operation 12

This chapter provides information on setting the Core Processor memory-mapped registers that configure the local memory bus. Topics include enabling/disabling data caching for a memory region, setting Intel® i960® core local bus width, the Bus Interface Unit (BIU), and the *Intel*® 80960RM/RN internal bus.

12.1 Core Processor Memory Attributes

Every location in memory has associated physical and logical attributes. For example, a specific location may have the following attributes:

- ***Logical:*** Data is non-cacheable
- ***Physical:*** 80303 I/O processor requires all to be 32-bit wide physical regions

In the example above, physical attributes correspond to those parameters that indicate *how to physically access the data*. The BCU uses physical attributes to determine the local bus protocol and signal pins to use when controlling the memory subsystem. The logical attributes tell the BCU how to interpret, format and control interaction of on-chip data caches. The physical and logical attributes for an individual location are independently programmable.

12.2 Physical Memory Attributes

The physical memory attributes of the Intel® 80303 I/O processor are controlled through the PMCON registers and the BCON.

12.2.1 PMCON Registers

Physical Memory Configuration registers, PMCON0_1 to PMCON14_15, are shown in Table 12-2. The PMCON registers reside within memory-mapped control register space. Each PMCON register controls one 512-Mbyte region of memory according to mapping in Table 12-1.

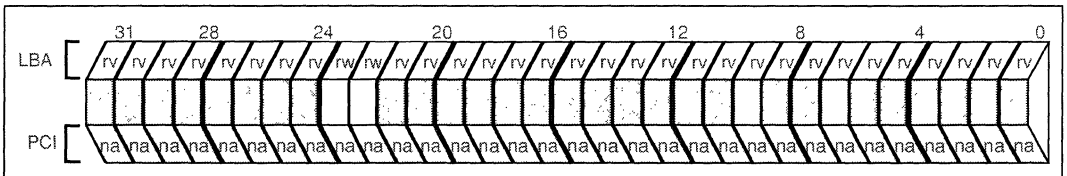
Table 12-1. PMCON Address Mapping

Register (Control Table Entry)	Region Controlled	Required Bus Width
Physical Memory Control Register 0 – PMCON0_1	0000 0000H to 0FFF FFFFH and 1000 0000H to 1FFF FFFFH	32 bits - 80960RM/RN Peripheral Memory-Mapped Registers
Physical Memory Control Register 1 – PMCON2_3	2000 0000H to 2FFF FFFFH and 3000 0000H to 3FFF FFFFH	32 Bits
Physical Memory Control Register 2 – PMCON4_5	4000 0000H to 4FFF FFFFH and 5000 0000H to 5FFF FFFFH	32 Bits
Physical Memory Control Register 3 – PMCON6_7	6000 0000H to 6FFF FFFFH and 7000 0000H to 7FFF FFFFH	32 Bits
Physical Memory Control Register 4 – PMCON8_9	8000 0000H to 8FFF FFFFH and 9000 0000H to 9FFF FFFFH	32 bits - 80960RM/RN outbound ATU translation windows
Physical Memory Control Register 5 – PMCON10_11	A000 0000H to AFFF FFFFH and B000 0000H to BFFF FFFFH	32 Bits
Physical Memory Control Register 6 – PMCON12_13	C000 0000H to CFFF FFFFH and D000 0000H to DFFF FFFFH	32 Bits
Physical Memory Control Register 7 – PMCON14_15	E000 0000H to EFFF FFFFH and F000 0000H to FFFF FFFFH	32 Bits

The Bus Interface Unit expects all accesses coming out of the i960® core processor to be targeted for a 32-bit region. The PMCON registers should all be programmed to support a 32-bit bus width during initialization and then left alone.

All eight PMCON registers are loaded automatically during system initialization. The initial values are stored in the Control Table in the Initialization Boot Record [Section 11.4, "Initial Memory Image (IMI)" on page 11-10].

Table 12-2. Physical Memory Control Registers – PMCON0:15

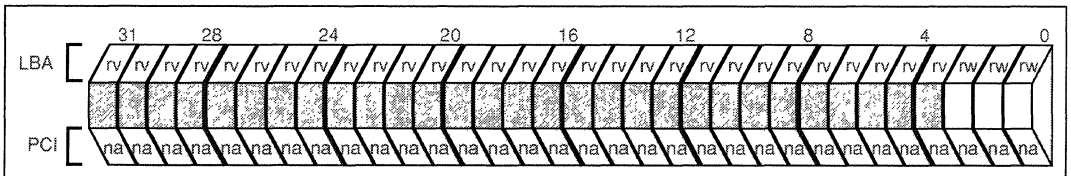


LBA: Table 12-1 **PCI:** NA		**Legend:** NA = Not Accessible, RO = Read Only, RV = Reserved, PR = Preserved, RW = Read/Write, RS = Read/Set, RC = Read Clear, LBA = 80303 Local Bus Address, PCI = PCI Configuration Address Offset
Bit	**Default**	**Description**
31:24	00H	Reserved. Initialize to 0.
23:22	00_2	Bus Width Selects the local bus width for a region: (00) = reserved (do not use) (01) = reserved (do not use) (10) = 32-bit bus (11) = reserved (do not use)
21:00	00 0000H	Reserved. Initialize to 0.

12.2.2 Bus Control Register – BCON

Immediately after a hardware reset, the PMCON register contents are marked invalid in the Bus Control (BCON) register. When the PMCON entries are marked invalid in BCON, the BCU uses the parameters in PMCON14_15 for *all* regions. On a hardware reset, PMCON14_15 is automatically cleared. This operation configures all regions to an 8-bit bus width. Subsequently, the processor loads all PMCON registers from the Control Table. The processor then loads BCON from the Control Table. When bit 0 of BCON is clear, PMCON14_15 remains in use for all local bus accesses. When bit 0 of BCON is set, the region table is valid and the BCU uses the programmed PMCON values for each region.

Table 12-3. Bus Control Register – BCON



LBA: 86FCH **PCI:** NA	**Legend:** NA = Not Accessible, RO = Read Only, RV = Reserved, PR = Preserved, RW = Read/Write, RS = Read/Set, RC = Read Clear, LBA = 80303 Local Bus Address, PCI = PCI Configuration Address Offset	
Bit	**Default**	**Description**
31:03	0000 0000H	Reserved.
02	0_2	Supervisor Internal RAM Protection (0) = First 64 bytes not protected from supervisor mode write (1) = First 64 bytes protected from supervisor mode writes
01	0_2	Internal RAM Protection (0) = Internal data RAM not protected from user mode writes (1) = Internal data RAM protected from user mode write
00	0_2	Configuration Entries in Control Table Valid) (0) = PMCON entries not valid, default to PMCON14_15 setting (1) = PMCON entries valid

12.3 Programming the Logical Memory Attributes

Bit field definitions for *Logical Memory Address Registers* - LMADR1:0 and LMMR1:0 registers are shown in Table 12-4. LMCON registers reside within the i960 core processor memory-mapped control register space. (Appendix C, "Peripheral Memory-Mapped Registers".)

12.3.1 Logical Memory Attributes

The 80303 I/O processor provides a mechanism for defining two *Logical Memory Templates (LMTs)*. An LMT may be used to specify whether a section (or subset) of a physical memory subsystem connected to the BCU (e.g., DRAM, SRAM) is cacheable or non-cacheable in the on-chip data cache.

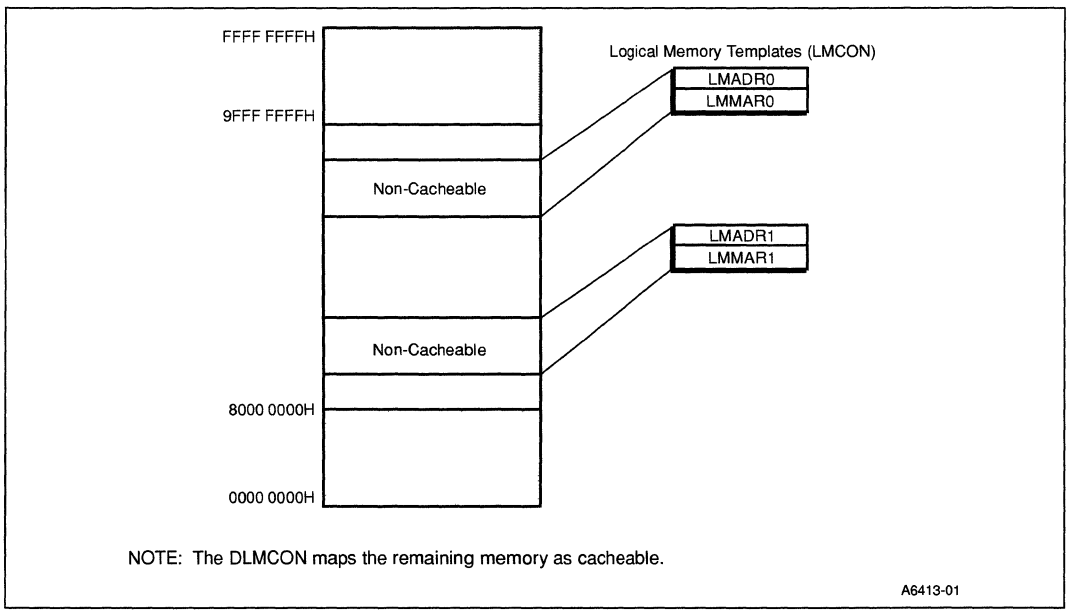
There are typically several different LMTs defined within a single memory subsystem. For example, data within one area of DRAM may be non-cacheable while data in another area is cacheable. Figure 12-1 shows the use of the *Control Table registers (PMCON)* with logical memory templates for a single DRAM region in a typical application.

Each logical memory template is defined by programming the *Logical Memory Configuration registers (LMCON)*. An LMCON register pair defines a data template for areas of memory that have common logical attributes. The 80303 I/O processor has two pairs of LMCON registers — defining two separate templates. The extent of each data template is described by an address (on 4 Kbyte boundaries) and an address mask. The address is programmed in the *Logical Memory Address register (LMADR)*. The mask is programmed in the *Logical Memory Mask register (LMMR)*. These two registers constitute the LMCON register pair.

The *Default Logical Memory Configuration (DLMCON)* register provides configuration data for areas of memory that do not fall within one of the two logical data templates.

The LMCON registers and their programming are described in Section 12.3, "Programming the Logical Memory Attributes" on page 12-5.

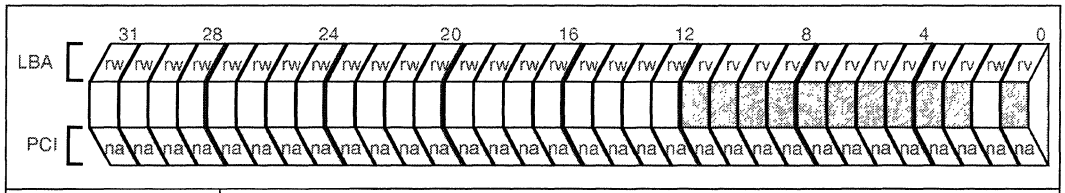
Figure 12-1. LMCON Example



12.3.2 Logical Memory Address Registers - LMADR0:1

The LMADR1:0 registers define the address for the logical data templates and template caching.

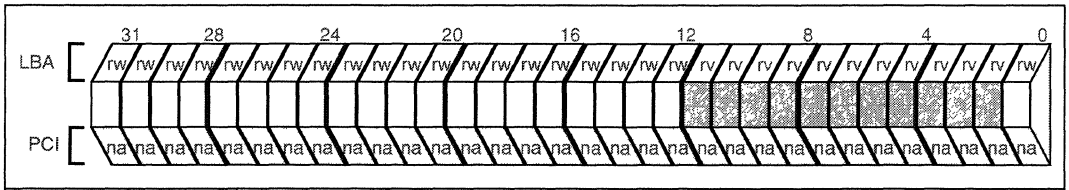
Table 12-4. Logical Memory Address Registers – LMADR0:1



LBA: CH0-8108H CH1-8110H **PCI:** NA	**Legend:** NA = Not Accessible, RO = Read Only, RV = Reserved, PR = Preserved, RW = Read/Write, RS = Read/Set, RC = Read Clear, LBA = 80303 Local Bus Address, PCI = PCI Configuration Address Offset

Bit	Default	Description
31:12	0000 0H	Template Starting Address - Defines upper 20 bits for the address of a logical data template. The lower 12 bits are fixed at zero. The starting address is modulo 4 Kbytes.
11:02	000H	Reserved.
01	0_2	Data Cache Enable - Controls data caching for the template. (0) = Data caching disabled (1) = Data caching enabled Instruction caching is never affected by this bit.
00	0_2	Reserved.

Table 12-5. Logical Memory Mask Registers – LMMR0:1

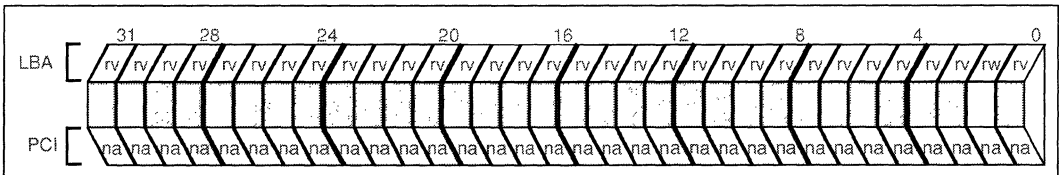


LBA: CH0-810CH CH1-8114H **PCI:** NA	**Legend:** NA = Not Accessible, RO = Read Only, RV = Reserved, PR = Preserved, RW = Read/Write, RS = Read/Set, RC = Read Clear, LBA = 80303 Local Bus Address, PCI = PCI Configuration Address Offset

Bit	Default	Description
31:12	0000 0H	Template Address Mask - Defines upper 20 bits for the address mask for a logical memory template. The lower 12 bits are fixed at zero (MA). (0) = Mask (1) = Do not mask
11:01	000H	Reserved.
00	0_2	Logical Memory Template Enabled - Enables/disables logical memory template. (0) = LMT disable (1) = LMT enabled

The Default Logical Memory Configuration (DLMCON) register is shown in Table 12-6. The BCU uses the parameters in the DLMCON register when the current access does not fall within one of the two logical memory templates (LMTs).

Table 12-6. Default Logical Memory Configuration Register – DLMCON



LBA: 8100H **PCI:** NA	**Legend:** NA = Not Accessible, RO = Read Only, RV = Reserved, PR = Preserved, RW = Read/Write, RS = Read/Set, RC = Read Clear, LBA = 80303 Local Bus Address, PCI = PCI Configuration Address Offset	
Bit	**Default**	**Description**
31:02	0000 0000H	Reserved.
01	0_2	Data Cache Enable - Controls data caching for areas not within other logical memory templates. (0) = Data caching disabled (1) = Write-through caching enabled Instruction caching is never affected by this bit.
00	0_2	Reserved.

12.3.3 Defining the Effective Range of a Logical Data Template

For each logical data template, an LMADRx register sets the base address using the bits 31:12. The LMMR register sets the address mask using the bits 31:12. The effective address range for a logical data template is defined by using bits 31:12 in the LMADRx register and bits 31:12 in the LMMRx register.

For each access, only those address bits in the range 31:12 marked as unmasked (defined by bits MA31:12 in the LMMRx register), are compared against bits 31:12 in the LMMRx register. When all of the unmasked bits of the address match bits 31:12 of the LMMRx register, then the address falls within the memory region governed by "x" logical memory template. The lower 12 address bits are not compared and are thus considered masked bits or "don't care" bits. This forces a minimum 4 Kbyte boundary on a memory region governed by a logical memory template. Logically, the operation is as follows:

(EFA31:12 xnor LMADRx31:12) or (not LMMRx31:12)

Where EFA31:12 is the effective address for a bus access. Only when all compared address bits match is the logical data template used for the current access. Two examples help clarify the operation of the address comparators.

- Create a template 64 Kbytes in length beginning at address 0010 0000H and ending at address 0010 FFFFH. Determine the form of the candidate address to match and then program the LMADR and LMMR registers:

  Candidate Address is of form:`0010 XXXX`
  LMADR <31:12> should be:`0010 0...`
  LMMR <31:12> should be:`FFFF 0...`

- Multiple data templates can be created from a single LMADRxLMMRx register pair by aliasing effective addresses. For example, to create sixteen 64 Kbyte templates, each beginning on modulo 1 Mbyte boundaries starting at 0000 0000H and ending with 00F0 0000H, the registers are programmed as follows:

  Candidate Address is of form:`00X0 XXXX`
  LMADR <31:12> should be:`0000 0...`
  LMMR <31:12> should be:`FF0F 0...`

12.3.4 Data Caching Enable

Enabling and disabling data caching for an LMT is controlled via the bit 0 in the LMADR register. Likewise, the bit 1 in the DLMCON enables and disables data-caching for regions of memory that are not covered by the LMCON registers.

Disabling a memory range does not exclude an address range from being cacheable. For cacheable ranges, the BCU promotes all sub-word accesses to word accesses.

12.3.5 Enabling the Logical Memory Template

LMMRx bit 0 activates the logical data template in the LMMR register for the programmed range.

12.3.6 Initialization

Immediately following a hardware reset, all LMTs are disabled. The bit 0 in each of the LMMR registers is cleared (0) and all other bits are undefined. Also the Default Logical Memory Control register Data Caching Enable (LMADRx bit 1) is cleared (Data Caching Disabled). Application software may initialize and enable the logical memory template after hardware reset. The registers are not modified by software initialization.

12.3.7 Boundary Conditions for Logical Memory Templates

The following sections describe the operation of the LMT registers during conditions other than "normal" accesses. See Chapter 4, "Cache and On-Chip Data RAM" for a treatment of data cache coherency when modifying an LMT.

12.3.7.1 Internal Memory Locations and Peripheral MMRs

The LMT registers are not used during accesses to i960 core processor memory-mapped registers. Internal data RAM locations are never cached; LMT bits controlling caching are ignored for data RAM accesses. The 80303 I/O processor peripheral MMRs, (addresses 0000 1000H through 0000 17FFH) and the ATU windows (8000 0000H through 9001 FFFFH) should be defined as non-cacheable. Further, if direct addressing is enabled (bit 8 of the ATUCR) addresses 0000 0000H through 7FFF FFFFH should be defined as non-cacheable.

12.3.7.2 Overlapping Logical Data Template Ranges

Logical data templates that specify overlapping ranges are not allowed. When an access is attempted that matches more than one enabled LMT range, the operation of the access becomes undefined.

To establish different logical memory attributes for the same address range, program non-overlapping logical ranges, then use partial physical address decoding.

12.3.7.3 Accesses Across LMT Boundaries

Accesses that cross LMT boundaries should be avoided. These accesses are unaligned and broken into a number of smaller aligned accesses, which reside in one or the other LMT, but not both. Each smaller access is completed using the parameters of the LMT in which it resides.

12.3.8 Modifying the LMT Registers

An LMT register can be modified using **st** or **sysctl** instructions. Both instructions ensure data cache coherency and order the modification with previous and subsequent data accesses.

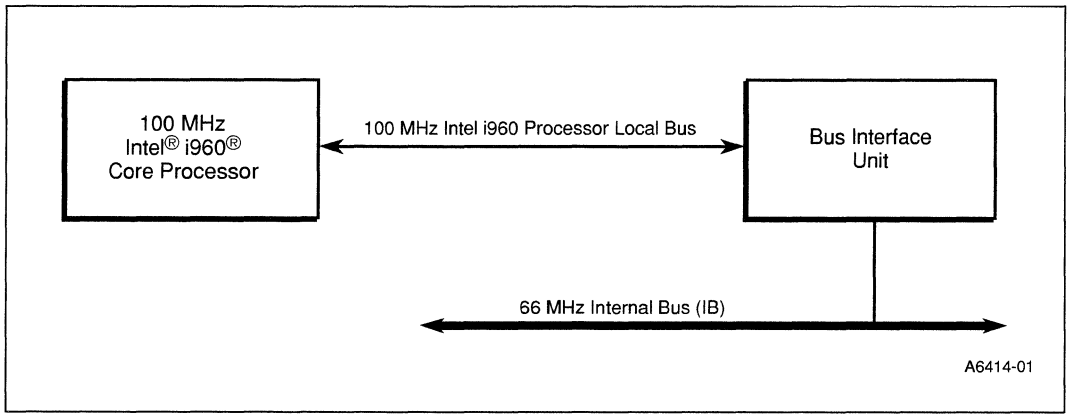
12.4 Bus Interface Unit

The BIU connects the i960 core processor to the Internal Bus. The BIU has two bus interfaces:

- 32-bit i960 core processor bus
- 64-bit Internal Bus (IB).

The BIU is the only agent on the i960 core processor bus. The BIU also separates the core processor clock domain from the Internal Bus clock domain. See Figure 12-2.

Figure 12-2. Core Processor/BIU Interface Block Diagram



The BIU forwards i960 core processor bus accesses to the Internal Bus and is responsible for their completion. No address translation is performed by the BIU.

12.4.1 Overview

The BIU accepts i960 core processor bus accesses. It forwards read accesses to the IB and returns the read data to the i960 core processor. It completes write accesses for the i960 core processor.

All accesses received by the BIU from the i960 core processor are processed in order, except that instruction fetches may bypass write accesses. See Section 12.4.4.2, "Instruction Fetch Bypass" on page 12-13.

The BIU may address any target on the Internal Bus. Instruction fetch accesses by the i960 core processor to either ATU is not supported.

The BIU divides the core processor clock domain (100 MHz) from the Internal Bus clock domain (66 MHz). All data moving through the BIU is buffered.

The BIU has several address/data buffers:

- Write Buffer
- Read Buffer
- Prefetch Buffer

The Write Buffer temporarily stores write accesses that are destined for the IB. The Write Buffer is 2 entries deep and each entry can store one address and up to 16 bytes of data. The BIU is responsible for forwarding all write accesses to the IB and ensuring their completion.

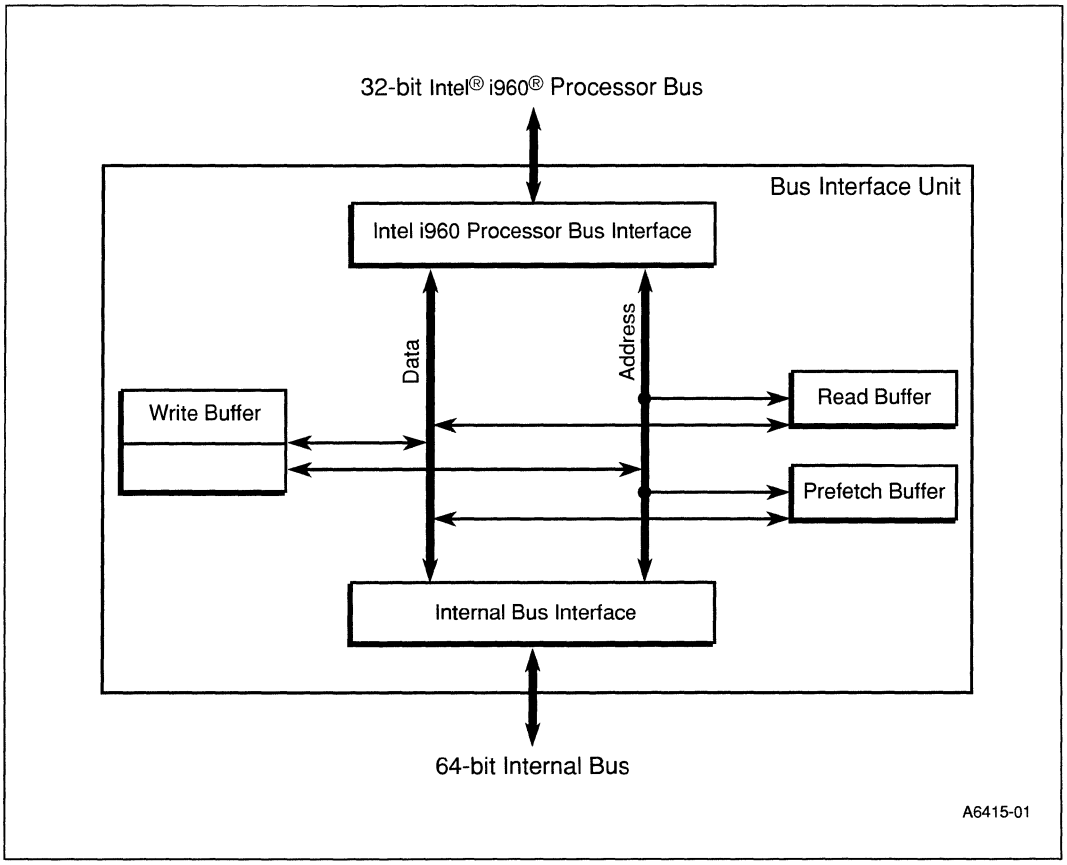
The Read Buffer temporarily stores read data for read accesses returning from the IB to the i960 core processor. The Read Buffer is one entry deep and each entry can store up to 16 bytes of data.

The Prefetch Buffer temporarily stores additional instructions prefetched by the BIU from the Memory Controller.

The BIU has two optional features that are intended to increase overall performance. The BIU can extend i960 core processor fetches by 16 bytes and then store the additional 16 bytes in the Prefetch Buffer. If a subsequent instruction fetch hits the Prefetch Buffer, the instructions are returned to the processor and an IB bus access is avoided. Under special conditions, the BIU also can merge two sequential write accesses into one IB bus access. Both of these features can be independently enabled or disabled.

The BIU does not perform byte merging (merging byte writes together) or write collapsing (collapsing multiple writes to one location).

Figure 12-3. Internal Block Diagram



12.4.2 Addressing

The BIU converts 32-bit DWORD addresses from the i960 core processor bus into 64-bit QWORD addresses on the Internal Bus. The BIU does not translate addresses or otherwise alter addresses.

The BIU only reads and writes data on the Internal Bus as indicated by the Byte Enables generated by the i960 core processor. The BIU assumes that all accesses are to non-prefetchable memory. It does not promote i960 byte or i960 short accesses to DWORD accesses.

12.4.2.1 Bus Width

The BIU only supports i960 core processor data bus width of 32-bits. The Bus Width field (BW1:0) in the Physical Memory Region Configuration Registers (PMCON) should set to 10_2 (32-bit bus).

Note: Setting the i960 core processor data bus width in the PMCON Registers to 16-bits or 8-bits results in undefined behavior.

The BIU, however, does supports the 8-bit bus width when the Initial Boot Record (IBR) is read during core processor initialization.

12.4.3 Multi-Transaction Timer

The BIU has an associated Multi-Transaction Timer (MTT) in the Internal Bus Arbiter. When programmed properly, the MTT allows for a guaranteed quantum of time for the BIU. See Chapter 17, "Intel® 80303 I/O Processor Arbitration".

12.4.4 Features

Additional features of the BIU are:

- Write Buffering
- Instruction Fetch Bypass
- Instruction Prefetch (optional)
- Write Merging (optional)

12.4.4.1 Write Buffering

The Write Buffer temporarily stores multiple i960 core processor write accesses waiting for completion on the Internal Bus. The Write Buffer has two entries. Each entry contains one 32-bit address and 16 bytes of data storage.

Write buffering allows the BIU to handle up to 2 outstanding write accesses from the i960 core processor. If Write Merging is enabled, up to 4 outstanding write accesses are allowed.

An instruction fetch by the i960 core processor may bypass write accesses in the Write (see Section 12.4.4.2, "Instruction Fetch Bypass").

12.4.4.2 Instruction Fetch Bypass

With instruction fetch bypass, instruction fetches by the i960 core processor bypass any write accesses in the Write Buffer. The instruction fetch has priority over all write accesses in the Write Buffer for the next IB access performed by the BIU.

If the write access in the Write Buffer was attempted on the IB but has not completed (e.g., received a Retry), the instruction fetch does not bypass the write access. The instruction fetch bypasses the write access only if the write access has not started on the IB.

There is no address checking between the address of the instruction fetch and the address of any write accesses in the Write Buffer.

12.4.4.3 Instruction Prefetch

With instruction prefetch, instruction fetches by the i960 core processor cause the BIU to extend the IB bus access by an additional 16 bytes. An 8-byte fetch is extended to a 24-byte fetch and a 16-byte fetch is extended to a 32-byte fetch. The 8-byte or 16-byte fetch data originally requested by the processor is returned to the processor. The additional 16 bytes of instructions are stored in the Prefetch Buffer and marked valid.

If, for any reason, the Memory Controller is not able to deliver the complete extra 16 bytes of instructions (e.g., the Memory Controller disconnects after the original 8-byte or 16-byte fetch but before the complete 24-byte or 32-byte fetch is returned), the prefetch is aborted. The Prefetch Buffer is not loaded and is marked invalid.

When instruction prefetch is enabled, the address of all instruction fetches is compared with the address stored in the Prefetch Buffer. If the buffer is valid and the addresses match, the BIU returns the contents of the Prefetch Buffer to the i960 core processor and the BIU does not begin an IB access for the fetch.

Because the Prefetch Buffer contains 16 bytes and the i960 core processor may make an 8-byte instruction fetch, it is possible that the desired 8 bytes is in the Prefetch Buffer but the addresses do not match. In this case, the BIU does not return the 8 bytes from the Prefetch Buffer but must instead make an IB bus request. For example, if the Prefetch Buffer is marked valid and contains the address A000.0000H and the subsequent instruction fetch is an 8-byte fetch with an address of A000.0002H, the BIU does not match the addresses even though the instructions are in the Prefetch Buffer.

Instruction Prefetch can be enabled or disabled by the Instruction Prefetch Enable bit in the BIU Control Register.

12.4.4.4 Write Merging

With write merging, the BIU may merge two sequential write accesses by the i960 core processor into one IB bus access. Write merging is controlled by the Write Merging Enable bit in the BIU Control Register.

There is only one type of sequential write accesses that may be merged: one DWORD write followed another one DWORD write.

For write merging, the addresses must be sequential and incrementing. Bits 31:03 of the addresses of both accesses must match exactly. Bits 02:00 of the address must be 000_2 for the first access and 100_2 for the second access. The resulting QWORD must be naturally aligned. No other pairings of store accesses are merged by the BIU.

For example, if the first access is a DWORD write and the address is xxxx.xxx0H, the next access must be a DWORD write and the address must be xxxx.xxx4H. If the first access is a DWORD write and the address is xxxx.xxx8H, the next access must be a DWORD write and the address must be xxxx.xxxCH.

DWORDs are merged at the input to the Write Buffer. Both DWORDs are written to the same entry in the Write Buffer to form a QWORD in the entry.

Write accesses are never deliberately held in the Write Buffer and not completed on the Internal Bus just to enable Write Merging. Write Merging only occurs if the first DWORD write has not started on the IB before the second DWORD write occurs.

Instruction fetches that bypass write accesses in the Write Buffer do not affect write merging. An instruction fetch that occurs between two DWORD writes may bypass the first write in the Write Buffer. The second write may then merge with the first write.

Example 12-1. Code Examples of Write Merging

```
; Merge Example
st g0, 0xA0000000
st g7, 0xA0000004

; Merge Example
st g5, 0xA0001008
st g4, 0xA000100C

; Non-Example
;(not merged due to non-sequential addresses)
st g5, 0xA0002000
st g6, 0xA0002010
```

12.4.4.5 Atomic Accesses

The BIU supports atomic bus accesses from the i960 core processor to local memory and the Peripheral Memory-Mapped Registers (PMMR) only. Atomic instructions (atmod, atadd) from the i960 core processor require that the BIU perform the memory read-modify-write operation atomically.

12.4.5 Interrupts and Error Conditions

The BIU records error conditions that result from accesses initiated by the BIU on the Internal Bus. The errors are recorded in the BIU Interrupt Status Register (BIUISR).

12.4.5.1 Master-Abort

There are two ways that the BIU can receive a Master-Abort from the Internal Bus:

- IB Master-Abort: No target on the Internal Bus claims the transaction
- PCI Master-Abort: The ATU, as a PCI master on behalf of the BIU, received a Master-Abort on the PCI bus and is returning the Master-Abort to the BIU during the read completion

When an Internal Bus access initiated by the BIU receives a Master-Abort, the BIU records the Master-Abort condition in the BIU Interrupt Status Register and signals an NMI interrupt to the i960 core processor. Note that a Master-Abort received from the ATU is not recorded as an IB Master-Abort in the BIU Interrupt Status Register.

The PCI Master Abort is recorded in the PATUISR or SATUISR depending on which outbound ATU window is accessed. The ATU generates an interrupt to the i960 core processor. When the ATU detects a Master-Abort on the PCI bus for a read access and the ATU returns the Master-Abort to the BIU during the read completion.

For read accesses, the BIU returns FFH to the i960 core processor for each byte read. For write accesses, the BIU clears the access from the Write Buffer.

12.4.5.2 PCI Target-Abort

There are two ways that the BIU can receive a Target-Abort from the Internal Bus:

- PCI Target-Abort: The ATU, as a PCI master on behalf of the BIU, received a Target-Abort on the PCI bus and is returning the Target-Abort to the BIU
- IB Target-Abort: The Memory Controller returned a Target-Abort to the BIU

The PCI Target Abort is recorded in the PATUISR or SATUISR depending on which outbound ATU window is accessed. The ATU generates an interrupt to the i960 core processor. When the ATU detects a Target-Abort on the PCI bus for a read access and the ATU returns the Target-Abort to the BIU during the read completion.

The BIU does not need to distinguish the difference between an MCU access that resulted in a target abort and a outbound ATU access that results in a target abort. Both the ATUs and the MCUs record the target aborts and generate its respective interrupt to the core. The only requirement for the BIU during a target abort, is to return any valid data received from the target and then returns FFH to the i960 core processor for each unread byte. For write accesses, the BIU clears the access from the Write Buffer.

The IB Target Abort is discussed in Section 12.4.5.3, "Internal Bus Target-Abort" on page 12-16 and When an Internal Bus access initiated by the BIU receives a Target-Abort, the Memory Controller Unit (MCU) records the Target-Abort condition in the MCU Interrupt Status Register and signal an NMI interrupt to the i960 core processor.

12.4.5.3 Internal Bus Target-Abort

There are four ways that the BIU can receive a Target-Abort from the Internal Bus Memory Controller Unit (MCU):

- Target Abort during BIU write to MCU
- Target Abort during BIU read from MCU
- Target Abort during BIU instruction fetch from MCU
- Target Abort during BIU instruction prefetch from MCU

The MCU generates a target abort to initiating masters only when the access hits the SDRAM, ECC is enabled, and the MCU detected a multi-bit ECC error.

All four of the Target Abort cases are described in the following bullets:

- **Target-Abort During BIU Write**

  When BIU Internal Bus access receives a Target-Abort from the MCU, the MCU records the Target-Abort in ELOGx registers and generates an NMI interrupt to the i960 core processor.

  Since the BIU burst a maximum of 2 data cycles (for 64-bit SDRAM), and 4 data cycles for (32-bit SDRAM), the target abort can occur on any of the data cycles. If the target abort occurs on any data cycle, except the last data cycle, the BIU discards with the remaining data.

  If target abort occurs on the last data cycle of a burst, the BIU completes the transaction before the MCU had determined there is a multi-bit ECC error. Therefore, to the BIU, the transaction appears completed and there is no remaining data in the write queue for the transaction. In this case, the MCU is the only unit that can notify the core processor of the error condition.

- **Target-Abort During BIU Read**

  When BIU Internal Bus access receives a Target-Abort from the MCU, the MCU records the Target-Abort in the ELOGx registers and generate an NMI interrupt to the i960 core processor.

  For read accesses, the BIU returns any data received from the target and then returns FFH to the i960 core processor for each unread byte.

- **Target-Abort During BIU Instruction Fetch**

  When BIU Internal Bus access receives a Target-Abort from the MCU, the MCU records the Target-Abort in the ELOGx registers and generate an NMI interrupt to the i960 core processor.

  For read accesses, the BIU returns any data received from the target and then returns FFH to the i960 core processor for each unread byte.

- **Target-Abort During BIU Instruction Prefetch**

  When BIU Internal Bus access receives a Target-Abort from the MCU, the MCU records the Target-Abort in the ELOGx registers and generate an NMI interrupt to the i960 core processor.

  For read accesses, the BIU returns any data received from the target and then returns FFH to the i960 core processor for each unread byte. If the target abort occurs during the BIU instruction prefetch, the prefetch buffer is not loaded and is marked invalid.

Note: The MCU records errors and generates an interrupt to the i960 core processor during an instruction prefetch. The MCU may log the same error condition multiple times under the following condition. If, during the instruction prefetch, the MCU generates a target abort, the error is recorded. If the instruction flow is such that the next instruction fetch hits the addresses which the BIU prefetch just occurred, the BIU generates an IB cycle to fetch the instructions as instructed to by the i960 core processor. The second access by the BIU, to the same address as the previous prefetch (which marked the prefetch buffer invalid because of the target abort), results in another target abort by the MCU to the BIU. The MCU logs the same error condition again if there are available ELOGx registers. However, because the interrupt from the MCU to the NMI latch is level sensitive, the MCU generates only one interrupt to the i960 core processor. The software processing the MCU errors, must ensure that all error conditions are processed each interrupt to the i960 core.

12.4.6 Register Definitions

There are two peripheral memory-mapped registers (PMMR) for the BIU. Table 12-7 lists the PMMR registers for the BIU.

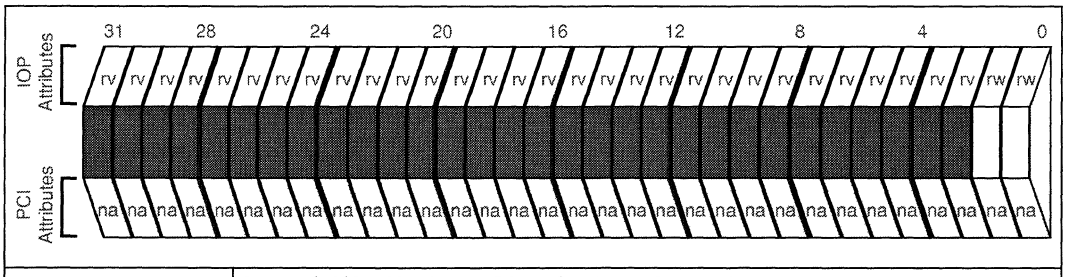
Table 12-7. Bus Interface Unit Register Table

Section, Register Name - Acronym (Page)
Section 12.4.6.1, "BIU Control Register - BIUCR" on page 12-17
Section 12.4.6.2, "BIU Interrupt Status Register - BIUISR" on page 12-18

12.4.6.1 BIU Control Register - BIUCR

The BIU Control Register (BIUCR) allows software to control the BIU.

Table 12-8. BIU Control Register - BIUCR

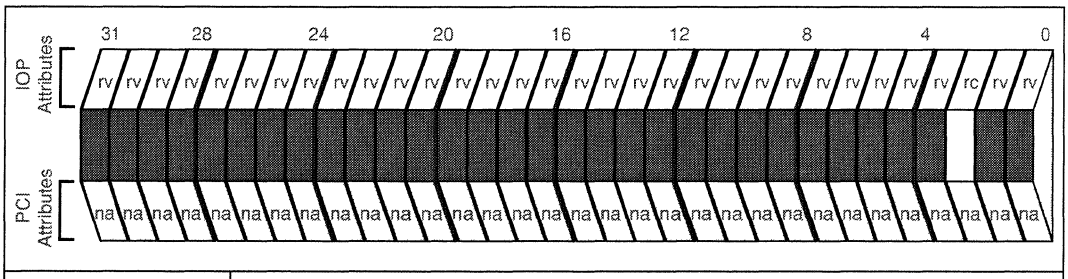


80303 Core Internal Address: 1640H		**Legend:** NA = Not Accessible, RO = Read Only, RV = Reserved, PR = Preserved, RW = Read/Write, RS = Read/Set, RC = Read Clear, LBA = 80303 Local Bus Address, PCI = PCI Configuration Address Offset
Bit	**Default**	**Description**
31:02	0000 0000H	Reserved
01	1_2	Write Merging Enable - When set, the BIU merges sequential DWORD Write accesses. When clear, the BIU does not merge any accesses.
00	1_2	Instruction Prefetch Enable - When set, the BIU extends i960 core processor instruction fetches by 16 bytes. The 16-byte prefetch data is stored in the Prefetch Buffer. When clear, the BIU does not prefetch additional instructions and the Prefetch Buffer is invalid.

12.4.6.2 BIU Interrupt Status Register - BIUISR

The BIU Interrupt Status Register (BIUISR) records the assertion of interrupts to the i960 core processor.

Table 12-9. BIU Interrupt Status Register - BIUISR



80303 Core Internal Address 1644H		**Legend:** NA = Not Accessible, RO = Read Only, RV = Reserved, PR = Preserved, RW = Read/Write, RS = Read/Set, RC = Read Clear, LBA = 80303 Local Bus Address, PCI = PCI Configuration Address Offset
Bit	**Default**	**Description**
31:03	00000000H	Reserved
02	0_2	IB Master-Abort - When set, the BIU has detected a Master-Abort on the Internal Bus and has signalled an NMI interrupt to the i960 core processor. This bit is cleared by software. Note that a Master-Abort received from the ATU is not recorded as an IB Master-Abort in this bit.
01:00	0_2	Reserved

Memory Controller 13

This chapter describes the integrated Memory Controller Unit (MCU). The operating modes, initialization, external interfaces, and implementation are detailed in this chapter.

13.1 Overview

The Intel® 80303 I/O processor integrates a Memory Controller to provide a direct interface between the 80303 I/O processor and its local memory subsystem. The Memory Controller supports[1]:

- Up to 16 Mbytes of 8-bit Flash (8Mbyte/Bank)
- Between 32 and 512 Mbytes of 64-bit Synchronous DRAM (SDRAM)
- Single-bit error correction, double-bit and nibble detection support (ECC[2])

The Flash interface provides an 8-bit data bus, 23-bit address bus, and control to support up to two 64 Mbit Bulk-Erase or Boot-Block Flash devices. The Flash devices provide storage for the 80303 I/O processor initialization code.

The MCU provides a separate SDRAM interface from the Flash interface. The SDRAM interface provides a direct connection to a high bandwidth and reliable memory subsystem. The SDRAM interface consists of a 100 MHz, 64-bit wide data path to support 800 Mbytes/sec throughput. In addition, the SDRAM interface is designed to be compatible with 133 MHz technologies. An 8-bit Error Correction Code (ECC) across each 64-bit word improves system reliability. The ECC is stored into the SDRAM array along with the data and is checked when the data is read. If the code is incorrect, the MCU corrects the data (if possible) before reaching the initiator of the read. User-defined fault correction software is responsible for scrubbing the memory array.

- The MCU supports two banks of SDRAM in the form of one unbuffered two-bank dual inline memory module (DIMM[1]) or two unbuffered single-bank DIMMs.
- The MCU responds to internal bus memory accesses within its programmed address range and issues the memory request to either the Flash or SDRAM interface.
- The MCU provides four chip enables to the memory subsystem. Two chip enables service the SDRAM subsystem (one per bank) and two service the Flash devices.

1. The MCU does NOT support registered DIMMs.
2. The MCU does NOT support non-ECC memory subsystems.

13.1.1 Glossary

This section lists commonly used terms throughout this chapter:

Table 13-1. Commonly Used Terms

Term	Definition
Bank	A bank is defined as a memory region defined with a base register and a bank size register. Physically, a bank of memory is controlled by a single chip select. A DIMM could comprise of a single or dual banks.
Column	A column refers to a portion of memory within an SDRAM device. An SDRAM device can be thought of as a grid with rows and columns. Once a row is activated, any column within that row can be accessed multiple times without reactivating the row. Columns are activated with **SCAS#**.
DIMM	A DIMM is an acronym for Dual Inline Memory Module. A DIMM is a physical card comprising multiple SDRAM devices. The card could be populated on one or both sides. A DIMM can be single or dual-bank.
Leaf	SDRAM devices use multiple banks within the device operating in an interleaved mode. 16 Mbit SDRAM devices contains two internal banks and the MCU supports 64 Mbit devices containing four internal banks. An internal bank is defined as a leaf (to avoid confusion with a memory bank).
Page	A page is a row of memory. Once a row is activated, any column within that row can be accessed multiple times without reactivating the row. This is referred to as "keeping the page open." While it depends on the SDRAM device configuration, the MCU supports only the smallest possible page size (2 Kbytes for 64-bit wide memory). Therefore, if an SDRAM physical configuration supports a larger page size, the MCU breaks it up into smaller 2 Kbyte pages.
Row	A row refers to a portion of memory within an SDRAM device. An SDRAM device can be thought of as a grid with rows and columns. Once a row is activated, any column within that row can be accessed multiple times without reactivating the row. Rows are activated with **SRAS#**.
Scrubbing	Once an error is detected within the memory array, the MCU must correct the error (if possible) while delivering the data to the initiator. Correcting the memory location is referred to as "scrubbing the array." The MCU relies on software to scrub any errors.
Syndrome	A syndrome is a value which indicates an error in the data read from the memory array. The MCU computes the syndrome with every memory read. Decoding the syndrome indicates: the bit in error for a single-bit error, a double-bit error, or a nibble-error. Table 13-13 defines the syndrome decoding.

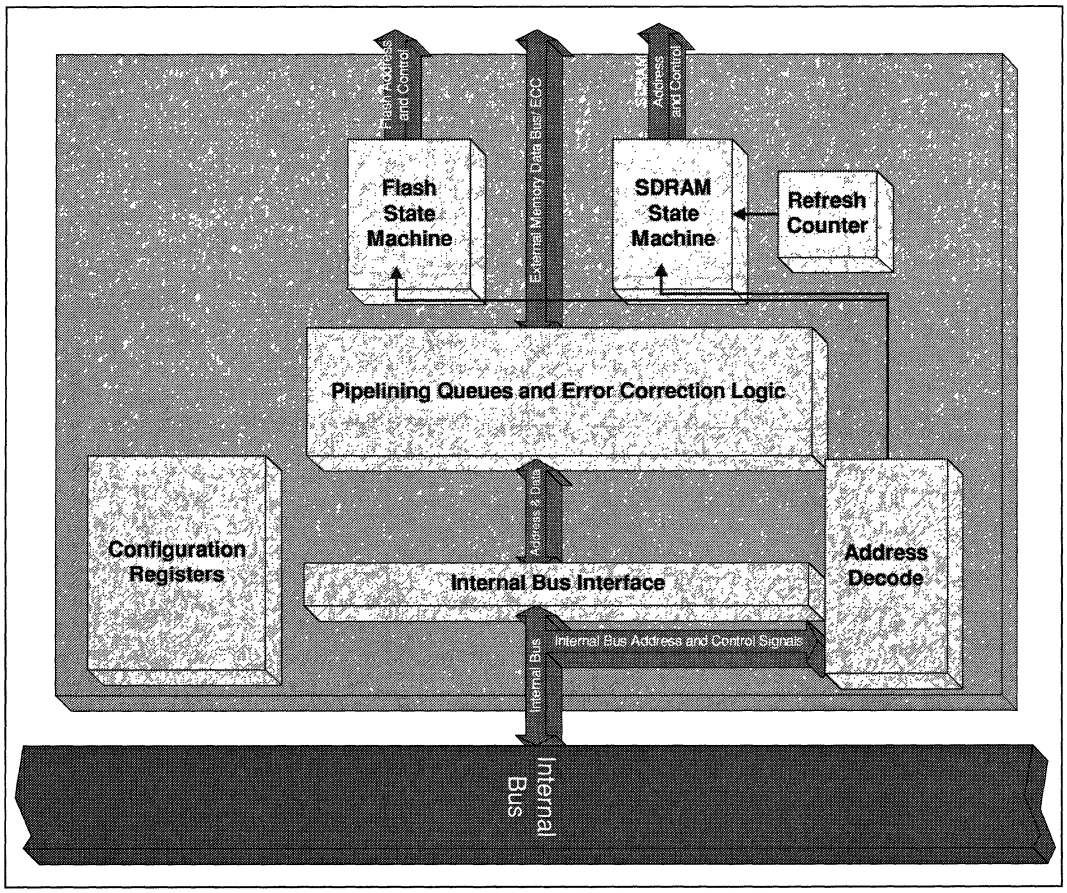
13.2 Theory of Operation

The 80303 I/O processor memory controller translates the internal bus transactions into the protocol supported by the connected memory subsystem. The supported memory components consist of SDRAM and/or Flash memory devices.

13.2.1 Functional Blocks

The memory controller logically comprises the blocks illustrated in Figure 13-1:

Figure 13-1. Memory Controller Block Diagram



13.2.1.1 Internal Bus Interface

The Internal Bus Interface block supports the internal bus protocol detailed in Appendix A, "Internal Bus Signaling and Protocol". The internal bus protocol is a subset of PCI protocol.

Since the MCU is only a slave for internal bus transactions, the internal bus interface monitors **I_FRAME#**. When **I_FRAME#** is asserted, the address decode block checks if the address falls within the Flash memory ranges, SDRAM memory ranges, or the MCU MMR register space. If the address falls within any of these ranges, the MCU claims the transaction by asserting **I_DEVSEL#**.

13.2.1.2 Address Decode

The Address Decode block is responsible for decoding the internal bus address and determining if the MCU should claim the internal bus transaction. There are three address ranges that the MCU responds to.

Flash Memory Space
: The Flash memory space is defined with the Flash Base Address Registers (FEBR0, FEBR1) and the Flash Bank Size Registers (FBSR0, FBSR1). The transaction is intended for a Flash bank if the address falls between the base register (FEBRx) and the base plus the Flash size register (FBSRx).

SDRAM Memory Space
: The SDRAM memory space is defined with the SDRAM Base Address Register (SDBR) and the SDRAM Boundary Registers (SBR0, SBR1). The transaction is intended for an SDRAM bank if the address is between the base register (SDBR) and between the boundaries programmed with SBR0 and SBR1.

  The Address Decode block also records and maintains the open SDRAM pages. The MCU can keep a maximum of eight pages open simultaneously. This block keeps track of these pages and determines if the internal bus transaction hits an open page. For more details about the page hit/miss determination, see Section 13.2.3.3, "Page Hit/Miss Determination" on page 13-16.

Memory-Mapped Register Space
: The MCU MMR memory space is 1500H to 15FFH. The registers are detailed in Section 13.6, "Register Definitions" on page 13-44.

13.2.1.3 Configuration Registers

The Configuration Registers block contains all of the memory-mapped registers listed in Section 13.6, "Register Definitions" on page 13-44. These registers define the memory subsystem connected to the 80303 I/O processor. The status registers indicate the current MCU status.

13.2.1.4 SDRAM State Machine

The SDRAM State Machine controls the protocol for SDRAM transactions.

13.2.1.5 Flash State Machine

The Flash State Machine controls the protocol for Flash transactions.

13.2.1.6 Refresh Counter

The Refresh Counter block keeps track of when the SDRAM devices need to be refreshed. The refresh interval is programmed in the RFR. Once the 10-bit refresh counter reaches the programmed interval, the SDRAM state machine issues a refresh command to the SDRAM devices. If a transaction is currently in progress, the SDRAM State Machine waits for the completion of the transaction to issue the refresh cycle. See Section 13.2.3.9, "SDRAM Refresh Cycle" on page 13-27 for more details.

13.2.1.7 Pipeline Queues and Error Correction Logic

Since the MCU generates error correction codes based on the data, the MCU is a pipelined architecture. Pipelining also ensures acceptable AC timings to the memory interfaces. The SDRAM state machine pipelines SDRAM memory operations for two clocks. The Flash state machine pipelines memory operations for one clock. The Pipeline Queues keep up to two stages of 64-bit data and 32-bit address.

The Error Correction Logic generates the ECC code for SDRAM reads and writes. For reads, this logic compares the ECC codes read with the locally generated ECC code. If the codes mismatch then the Error Correction Logic determines the error type. For a single-bit error, this block determines which bit is in error and corrects the error. For a double-bit or nibble error, the Error Correction Logic logs the error in ELOG0 and ELOG1. See Section 13.2.4, "Error Correction and Detection" on page 13-28 for more details.

13.2.2 Flash Memory Support

The 80303 I/O processor memory controller supports one or two 8-bit Flash devices. The second Flash bank may be used to interface a UART device. Flash devices typically store initialization code.

The MCU supports read bursting up to 8 bytes of data from the Flash device for a single read transaction. Any write transactions the core issues to the Flash address space must always be single byte transfers (`stob`).

The MCU separates the Flash interface from the SDRAM interface to isolate the electrical loading on the SDRAM interface. The MCU implements twenty three address pins multiplexed on **RAD[16:0]** to address Flash devices up to 64 Mbit. Refer to the timing diagrams in Figure 13-3 and Figure 13-4 for details about how the pins are multiplexed.

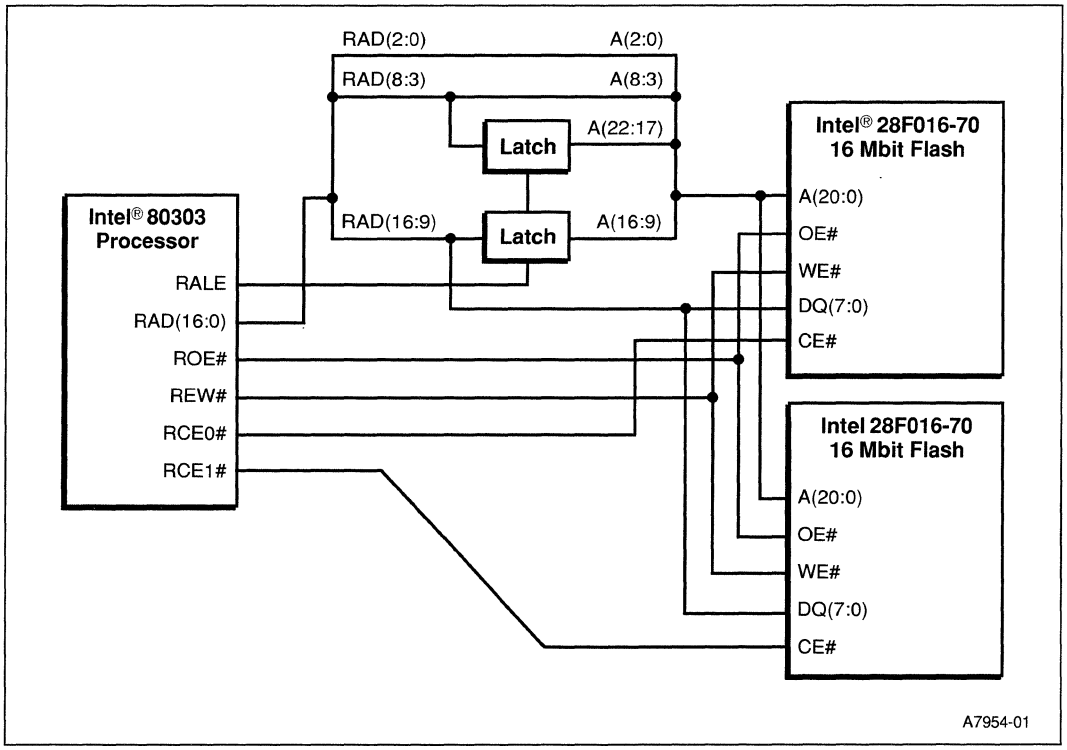
Flash memory space is separate from the SDRAM space. The Flash chip enables activate the appropriate Flash bank when the address falls within one of the Flash address ranges. Table 13-2 shows the Flash interface signals.

Table 13-2. Flash Interface Signals

Pin Name	Description
RCE[1:0]#	Chip Enable. Must be asserted for all transactions to the Flash device.
RWE#	Write Enable. Controls the Flash data input buffers.
ROE#	Output Enable. Asserted for reads, deasserted for writes. Controls the Flash output data buffers for write transactions.
RAD[16:0]	Address/Data bus capable of supporting 16 Mbit of Flash (2Mx8). The data bus is multiplexed on **RAD[16:9]**.
RALE	Address Latch Enable. Indicates the transfer of a physical address. **RALE** is asserted during a Flash address cycle and deasserted before the beginning of the data cycle.

Figure 13-2 illustrates how two Flash devices would interface with the 80303 I/O processor through the MCU.

Figure 13-2. Four Mbyte Flash Memory System



13.2.2.1 Flash Memory Addressing

Since the internal bus comprises a 64-bit data bus, it is possible that an internal bus master requests up to 8 bytes for a read transaction. The Flash interface utilizes an externally multiplexed address/data bus. The address bus is effectively 23 bits. The memory controller pipelines the address in two cycles. During the first address phase, address bits [22:9] are presented on **RAD[16:3]**. An external 14-bit latch must preserve **RAD[16:3]** using **RALE**. During the second address phase, address bits [8:0] are presented on **RAD[8:0].** The data bus is multiplexed on **RAD[16:9]**. The MCU increments the lower 3 address bits (**RAD[2:0]**) throughout the subsequent data phases.

The two Flash chip enables (**RCE[1:0]#**) support a Flash memory subsystem consisting of two devices. The base addresses for the two Flash devices are programmed in FEBR0 and FEBR1. The size of each Flash memory region is programmed in FBSR0 and FBSR1.

To determine if the internal bus address is within Flash memory space, Table 13-3 indicates which bits of the FEBRx are compared with the corresponding bits of **I_AD[31:0]**.

Table 13-3. Address Decoding for Flash Memory Space

Flash Bank Size	Bits Compared for Decoding the Flash Address Range
64 Kbytes	FEBRx[31:16]
128 Kbytes	FEBRx[31:17]
256 Kbytes	FEBRx[31:18]
512 Kbytes	FEBRx[31:19]
1 Mbyte	FEBRx[31:20]
2 Mbytes	FEBRx[31:21]
4 Mbytes	FEBRx[31:22]
8 Mbytes	FEBRx[31:23]

A valid address range for **RCE[0]#** is defined by the start address programmed in the base address register (FEBR0) with the ending address determined by the size programmed into the bank size register (FBSR0). The programmed value in the base address register requires alignment based on the size programmed into the FBSR0 register. The base address logic ignores the lower address bits based on the programmed block size. For example, if the memory size is 2 Mbytes, the software programs the base address value aligned with a 2 Mbyte boundary.

RCE[1]# follows the same logic but uses FEBR1 and FBSR1.

Refer to Section 13.2.5, "Overlapping Memory Regions" on page 13-34 for prioritization details if the two Flash memory regions overlap.

13.2.2.2 Flash Read Cycle

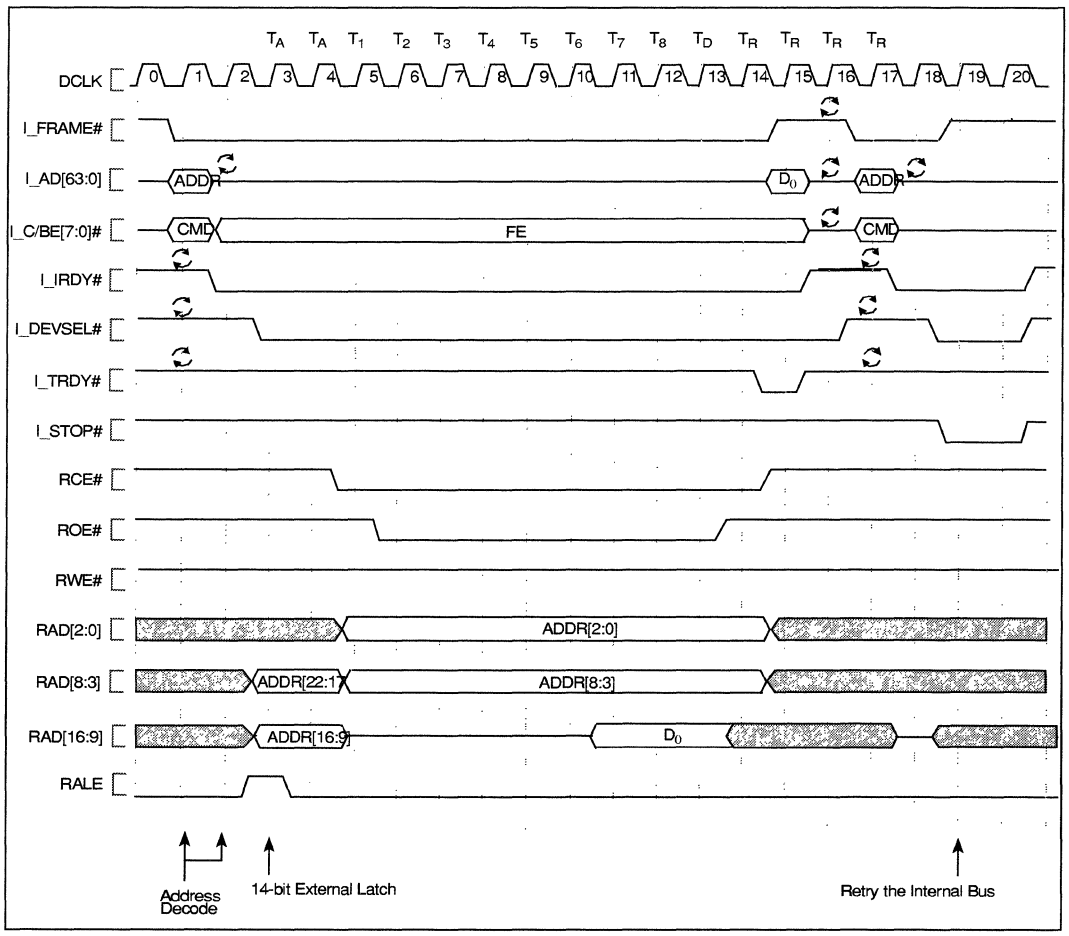
Reading a Flash device involves driving the address, output enable, and chip enable. Depending on the speed of the Flash device, the data returns several cycles later.

The definition of address-to-data wait states are the number of cycles between the assertion of **RCE[1:0]#** or **ROE#** (whichever is last), and the arrival of data from the Flash or UART device on **RAD[16:9]**. The definition of recovery wait states are the number of cycles between the data arrival on **RAD[16:9]** and the address for the next Flash transaction.

Address-to-data and recovery wait states programmed in FWSR0 and FWSR1 are identical for reads and writes. Since the read wait state requirement is typically greater, the write wait state requirement is guaranteed to be met. Refer to Table 13-4 for the programmable address-to data and recovery wait states.

Figure 13-3 illustrates a read cycle from a 90 ns Flash device.

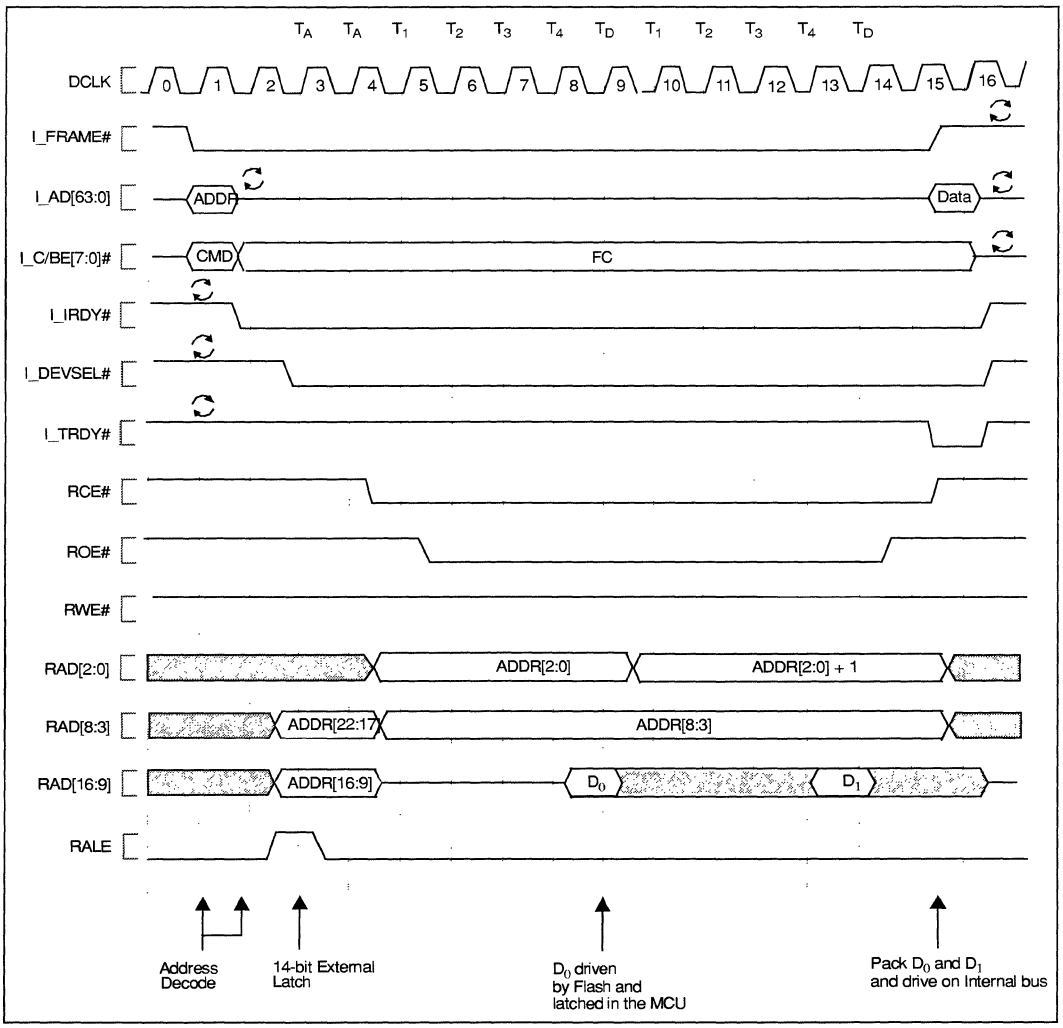
Figure 13-3. 90 ns Flash Read Cycle



When an internal bus master requests data from the Flash memory region, the MCU decodes the internal bus byte enables (**I_C/BE[7:0]#)** for the initial **RAD[2:0]**. The read request could result in multiple 8-bit reads (burst) on the Flash interface depending on **I_C/BE[7:0]#**. The Flash state machine increments **RAD[2:0]** for each read. The MCU is responsible for packing the multiple bytes and placing them on the appropriate byte lanes before driving the data on the internal bus. Due to the typically long time for Flash reads, the master reading data will always get disconnected after the first data phase.

Figure 13-4 illustrates a bursted read cycle from a 60 ns Flash device.

Figure 13-4. 60 ns Flash Burst Read Cycle



Refer to Table 13-4 for the programmable address-to data wait states.

Table 13-4. Flash Wait State Profile Programming

Flash Speed	Address-to-Data Wait States	Recovery Wait States
<= 55 ns	8	0
<= 115 ns	16	4
<= 175 ns	20	4

13.2.2.3 Flash Write Cycle

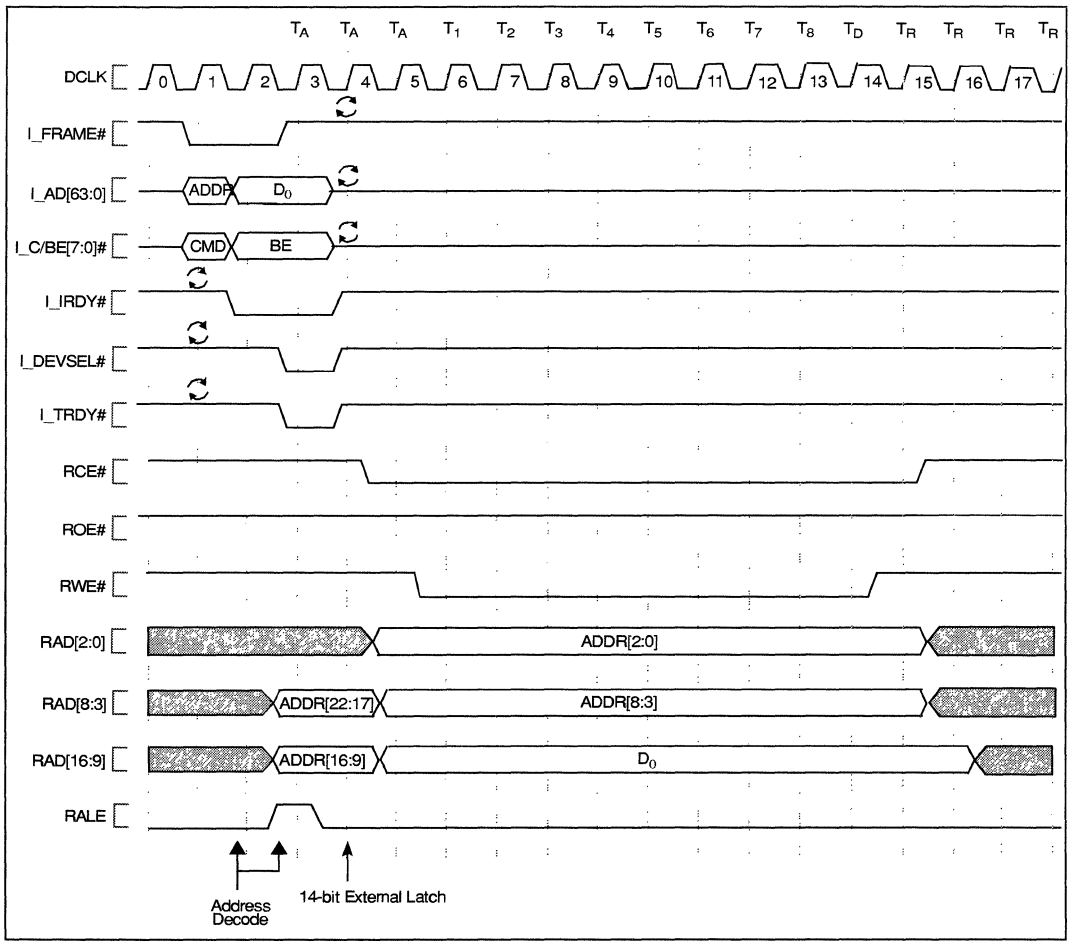
Address-to-data and recovery wait states for reads and writes are identical and programmed in FWSR0 and FWSR1. Refer to Table 13-4 for the programmable address-to data wait states.

The MCU claims internal bus transactions and accepts the data with zero wait-states, thus freeing the internal bus. However, the MCU remains busy until the cycle completes on the Flash interface. Subsequent MCU cycles are retried on the internal bus during this period.

The MCU does not support bursting data to a Flash device since the Flash device has no write buffers to support bursting data.

Figure 13-5 illustrates a write cycle to a 90 ns Flash device.

Figure 13-5. 90ns Flash Write Cycle



13.2.3 SDRAM Memory Support

The 80303 I/O processor memory controller supports one or two banks of SDRAM. SDRAM allows zero data-to-data wait-state operation at 100 MHz. SDRAM offers an extremely wide range of configuration options emerging from the SDRAMs internal interleaving and bursting capabilities.

The MCU supports SDRAM burst lengths of four. A burst length of four enables seamless read/write bursting of long data streams as long as the MCU master does not cross the page boundary. Page boundaries are at naturally aligned 2 Kbyte blocks for the 64-bit data bus. The MCU ensures that the page boundary is not crossed within a single transaction by initiating a disconnect with data on the Internal Bus prior to the page boundary.

The MCU SDRAM interface provides a flexible mix of combinations including:

Table 13-5 shows the SDRAM interface signals.

Table 13-5. SDRAM Interface Signals

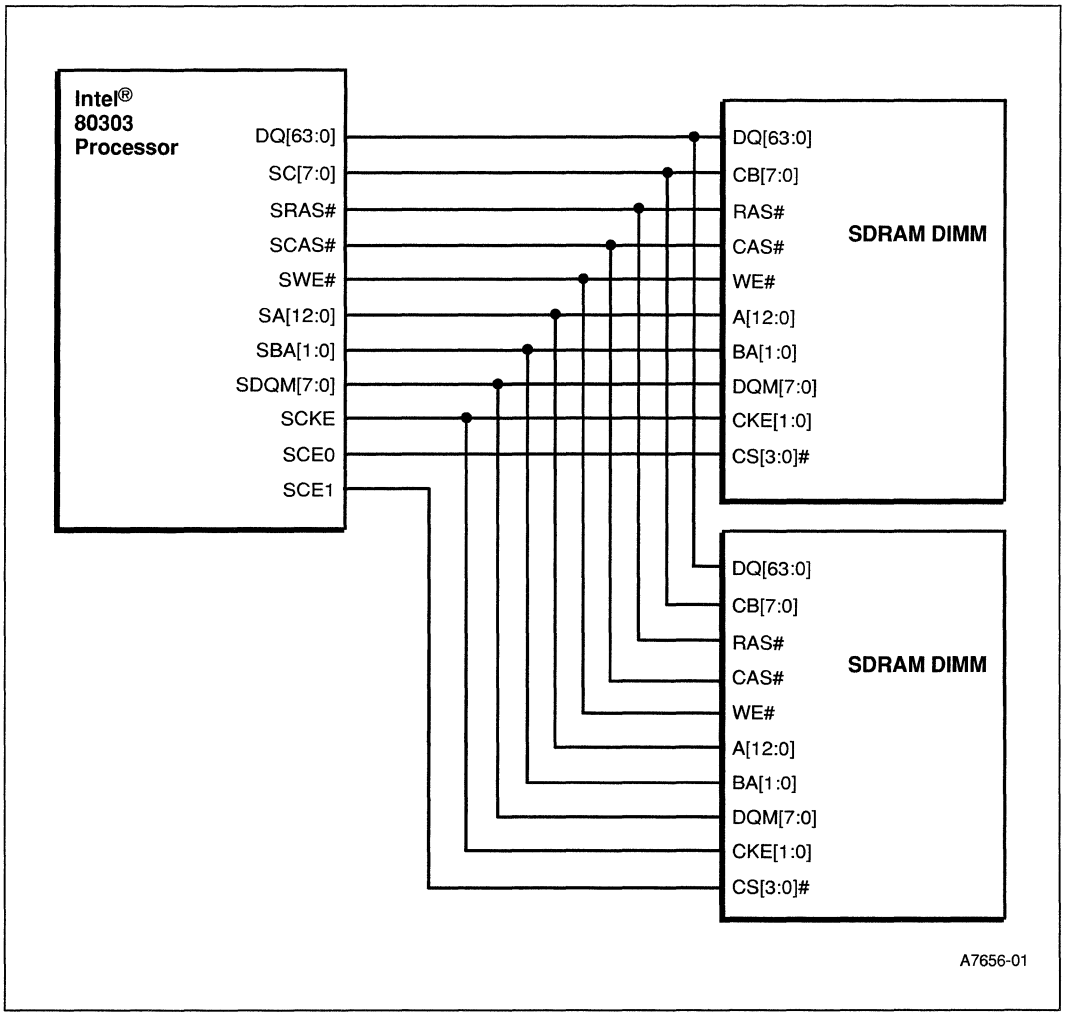
Pin Name	Description
DCLK[3:0]	*SDRAM Output Clocks* - These are the four output clocks driven to the Unbuffered DIMMs supported by the Intel® 80303 I/O processor. Section 13.2.6, "SDRAM Clocking" on page 13-35 describes the SDRAM clocking strategy.
DCLKOUT	*SDRAM Feedback clock* - This clock is driven in to the memory subsystem so that **DCLK[3:0]** may be skewed back to accommodate for the clocks' flight time and be compatible with 100/133 MHz SDRAM technologies
DCLKIN	*SDRAM Clock In* - This is the **DCLKOUT** clock returning from the memory subsystem. Section 13.2.6, "SDRAM Clocking" on page 13-35 describes the SDRAM clocking strategy.
SCKE[1:0]	*Clock enables* - One clock after **SCKE[1:0]** is deasserted, the data is latched on **DQ[63:0]** and **SCB[7:0]**. The burst counters within the SDRAM device are not incremented. Deasserting this signal places the SDRAM in self-refresh mode. For normal operation, **SCKE[1:0]** must be asserted.
SDQM[7:0]	*Data Mask* - On a write, these signals enable the eight output data bytes required by the 64-bit data bus. On a read, two clocks after asserting **SDQM[7:0]** the output data bytes are disabled.
SCE[1:0]#	*Chip Select* - Must be asserted for all transactions to the SDRAM device. One per bank.
SWE#	*Write Enable* - Controls the SDRAM data input buffers. Asserting **SWE#** causes the data on **DQ[63:0]** and **SCB[7:0]** to be written into the SDRAM devices.
SBA[1:0]	*SDRAM Bank Selects* - Controls which of the internal SDRAM banks to read or write. For 16 Mbit devices (2 banks), only **SBA[0]** is used while 64 Mbit devices use **SBA[1:0]**.
SA[10]	*Address bit 10* - If high during a read or write command, auto-precharge occurs after the command. During a **row-activate** command, this bit is part of the address (see Table 13-6).
SA[12:0]	*Address bits 12 through 0* - Indicates the row or column to access depending on the state of **SRAS# and SCAS#** (see Table 13-6).
SA[13]	*Address bit 13* -
SRAS#	*Row Address Strobe* - Indicates that the current address on **SA[12:0]** is the row.
SCAS#	*Column Address Strobe* - Indicates that the current address on **SA[12:0]** is the column.
DQ[63:0]	*Data Bus* - 64-bit wide data bus.
SCB[7:0]	*ECC Bus* - 8-bit error correction code which accompanies the data on **DQ[63:0].**

Utilizing the SDRAM chip enables **SCE[1:0]#** and internal bank selects **SBA[1:0]**, the MCU keeps a maximum of eight pages open simultaneously with 64/128/256 Mbit devices. The number of available pages depends on the memory subsystem population. A single 64/128/256 Mbit SDRAM bank allows four pages and two banks allow eight pages.

Open pages allow optimal performance when a read or write occurs to an open page. Multiple open pages allow multiple memory segments to be open simultaneously and is well-suited for the 80303 I/O processor's system environment. The MCUs paging algorithm is detailed in Section 13.2.3.3, "Page Hit/Miss Determination" on page 13-16. The waveforms illustrating the performance issues are in Section 13.2.3.7, "SDRAM Read Cycle" on page 13-21 and Section 13.2.3.8, "SDRAM Write Cycle" on page 13-24.

Figure 13-6 illustrates how two banks of SDRAM would interface with the 80303 I/O processor through the MCU.

Figure 13-6. Dual-Bank SDRAM Memory Subsystem



13.2.3.1 SDRAM Sizes and Configurations

The MCU supports an ECC only memory subsystem ranging from 32 to 528 Mbytes. An ECC system may be implemented using x8, or x16 devices. This allows flexibility and offers between 32 and 512 Mbytes. Table 13-6 illustrates the supported SDRAM configurations.

Table 13-6. Supported SDRAM Configurations

SDRAM Technology	SDRAM Arrangement	# Banks	Address Size		Leaf Select		Total Memory Size
			Row	Column	SBA[1]	SBA[0]	
64 Mbit	8M x 8	1	12	9	I_AD[25]	I_AD[24]	64M
		2					128M
	4M x 16	1	12	8	I_AD[24]	I_AD[23]	32M
		2					64M
128 Mbit	16M x 8	1	12	10	I_AD[26]	I_AD[25]	128M
		2					256M
	8M x 16	1	12	9	I_AD[25]	I_AD[24]	64M
		2					128M
256 Mbit	32M x 8	1	13	10	I_AD[27]	I_AD[26]	256M
		2					512M
	16M x 16	1	13	9	I_AD[26]	I_AD[25]	128M
		2					256M

64/128/256 Mbit SDRAM devices comprise four internal leaves. The MCU controls the leaf selects within 64/128/256 Mbit SDRAM by toggling **SBA[0]** and **SBA[1]**.

The two SDRAM chip enables (**SCE[1:0]#**) support an SDRAM memory subsystem consisting of two banks. The base address for the two contiguous banks are programmed in the SDRAM Base Register (SDBR) and must be aligned to a 32Mbyte boundary. The size of each SDRAM bank is programmed with the SDRAM boundary registers (SBR0 and SBR1).

Table 13-7. SDRAM Address Register Definitions

SDRAM Address Register	Definition
SDRAM Base Register (SDBR)	The lowest address for SDRAM memory space aligned to a 32 Mbyte boundary.
SDRAM Boundary Register 0 (SBR0)	The upper address offset to the SDBR for bank 0 of SDRAM memory space. Also, the lower address offset to the SDBR for bank 1 of SDRAM memory space.
SDRAM Boundary Register 1 (SBR1)	The upper address offset to the SDBR for bank 1 of SDRAM memory space. SBR1 must be greater than or equal to SBR0.

Note: SDRAM memory space must be aligned to a 32Mbyte boundary and must ***never*** cross a 512 Mbyte boundary.

The base register defines the upper seven address bits of the SDRAM memory space. The boundary registers define the address limits for each SDRAM bank in 32 Mbyte granularity. Table 13-8 defines the conditions which must be satisfied to activate an SDRAM memory bank.

Table 13-8. Address Decoding for SDRAM Memory Space

Condition	SDRAM Bank Selected
I_AD[31:29] is not equal to the SDBR[31:29]	None
I_AD[31:25] is greater than or equal to the SDBR[31:25] **I_AD[28:25]** is less than the value in SBR0[7:3]	Bank 0
I_AD[31:25] is greater than or equal to the SDBR[31:25] **I_AD[28:25]** is greater than or equal to the value in SBR0[7:3] **I_AD[28:25]** is less than the value in SBR1[7:3]	Bank 1

Example 13-1. Address Register Programming Example 1

The user wants to program the SDRAM memory space to begin at B000 0000H. Bank 0 is 32 Mbytes and Bank 1 is 64 Mbytes yielding in a total memory of 96 Mbytes. The registers would be programmed as follows:

SDBR = B000 0000H
SBR0[7:3] = 00001_2 = 01H
SBR1[7:3] = 00011_2 = 03H

Example 13-2. Address Register Programming Example 2

The user wants to program the SDRAM memory space to begin at B000 0000H. Bank 0 is 32 Mbytes and Bank 1 is unpopulated. The registers would be programmed as follows:

SDBR = B000 0000H
SBR0[7:3] = 00001_2 = 01H
SBR1[7:3] = 00001_2 = 01H

Example 13-3. Address Register Programming Example 3

The user wants to program the SDRAM memory space to begin at B000 0000H. Bank 0 is 32 Mbytes and Bank 1 is 32 Mbytes yielding in a total memory of 64 Mbytes. The registers would be programmed as follows:

SDBR = B000 0000H
SBR0[7:3] = 00001_2 = 01H
SBR1[7:3] = 00010_2 = 02H

Table 13-9 shows the programming for SDRAM memory space

Table 13-9. Programming Values for the SDRAM Boundary Registers (SBRx[7:3]) (Sheet 1 of 2)

Bank Size	Bank 0 (SBR0)	Bank 1 (SBR1)
Empty	SBR0 = 0x00 + SDBR[28:25]	SBR1 = 0x00 + SBR0[7:3]
32M	SBR0 = 0x01 + SDBR[28:25]	SBR1 = 0x01 + SBR0[7:3]
64M	SBR0 = 0x02 + SDBR[28:25]	SBR1 = 0x02 + SBR0[7:3]

Table 13-9. Programming Values for the SDRAM Boundary Registers (SBRx[7:3]) (Sheet 2 of 2)

Bank Size	Bank 0 (SBR0)	Bank 1 (SBR1)
128M	SBR0 = 0x04 + SDBR[28:25]	SBR1 = 0x04 + SBR0[7:3]
256M	SBR0 = 0x08 + SDBR[28:25]	SBR1 = 0x08 + SBR0[7:3]

13.2.3.2 SDRAM Addressing

SDRAM addressing for 64/128/256 Mbit Devices using SA[12:0]

Table 13-10 illustrates how the internal address is mapped to the **SA[12:0]** lines for 64/128/256 Mbit SDRAM devices.

Table 13-10. SDRAM Address Translation for 64/128/256 Mbit Devices using SA[12:0]

SA[12:0]	12	11	10	9	8	7	6	5	4	3	2	1	0
Row	I_AD[25]	I_AD[22]	I_AD[21]	I_AD[20]	I_AD[19]	I_AD[18]	I_AD[17]	I_AD[16]	I_AD[15]	I_AD[14]	I_AD[13]	I_AD[12]	I_AD[11]
Col	-	-	V[1]	I_AD[24]	I_AD[23]	I_AD[10]	I_AD[9]	I_AD[8]	I_AD[7]	I_AD[6]	I_AD[5]	I_AD[4]	I_AD[3]

NOTES:
1. **SA[10]** is used for precharge variations on the read or write command. See Table 13-12 for more details.
2. For the Leaf Selects, see Table 13-6.

SDRAM addressing for 256 Mbit Devices using SA[13,11:0]

The **SA[13]** signal can provide the appropriate address signal (**I_AD[24]**) for 256Mb x 16 SDRAM devices. Table 13-11 illustrates how the internal address is mapped to the **SA[13,11:0]** lines. Instead of connecting **SA[12]** to the 256Mb x 16 SDRAM, replace it with **SA[13]**. With this configuration the 13 rows and 9 columns will now provide the correct addressing to the 256Mb x 16 SDRAM.

Table 13-11. SDRAM Address Translation for 256 Mbit Devices

SA[13,11:0]	13	11	10	9	8	7	6	5	4	3	2	1	0
Row	I_AD[24]	I_AD[22]	I_AD[21]	I_AD[20]	I_AD[19]	I_AD[18]	I_AD[17]	I_AD[16]	I_AD[15]	I_AD[14]	I_AD[13]	I_AD[12]	I_AD[11]
Col	-	-	V[1]		I_AD[23]	I_AD[10]	I_AD[9]	I_AD[8]	I_AD[7]	I_AD[6]	I_AD[5]	I_AD[4]	I_AD[3]

NOTES:
1. **SA[10]** is used for precharge variations on the read or write command. See Table 13-12 for more details.
2. For the Leaf Selects, see Table 13-6.

Since the MCU supports SDRAM bursting, the MCU increments the column address by four for each SDRAM read or write burst.

The MCU supports a sequential burst type (Figure 13-8). Sequential bursting means that the address issued to the SDRAM is incremented by the SDRAM device in a linear fashion during the burst cycle.

13.2.3.3 Page Hit/Miss Determination

The MCU address translation assumes a 2 Kbyte page even if the physical addressing allows a greater page size. For 64/128/256 Mbit devices, the MCU keeps four pages per bank (8 maximum) open simultaneously.

For 64/128/256 Mbit devices, the MCU keeps only one page each of Bank0/Leaf0, Bank0/Leaf1, Bank0/Leaf2, Bank0/Leaf3, Bank1/Leaf0, Bank1/Leaf1, Bank1/Leaf2, and Bank1/Leaf3 open simultaneously. This rule implies that one 2 Kbyte page per eighth of the memory can be open. See Figure 13-7 for an example organization using 64 Mbit devices.

The MCU paging logic determines the hit/miss status for reads and writes. For a new SDRAM transaction, the MCU compares the address of the current transaction with the address stored in the appropriate page address register. Assuming 64/128/256 Mbit SDRAM devices and two banks, there are eight pages kept open simultaneously. The SDRAM chip enables (**SCE[1:0]#**) and leaf selects (**SBA[1:0]**) determine which page address to compare.

If the current transaction misses the open page selected then the MCU closes the open page pointed to by **SCE[1:0]#** and **SBA[1:0]** by issuing a **precharge** command. The MCU opens the current page with a **row-activate** command and the transaction completes with a **read** or **write** command. When the MCU opens the current page, **I_AD[31:11]** is stored in the page address register pointed to by **SCE[1:0]#** and **SBA[1:0]** so it may be compared for future transactions.

If the current transaction hits the open page, then the page is already active and the **read** or **write** command may be issued without a **row-activate** command. If the refresh timer expires and the MCU issues an **auto-refresh** command, all pages are closed.

Figure 13-10 illustrates the performance benefit of a read hit versus a read miss in Figure 13-11. Figure 13-12 illustrates the performance benefit of a write hit versus a write miss in Figure 13-13.

Figure 13-7. Logical Memory Image of a 64/128/256 Mbit SDRAM Memory Subsystem

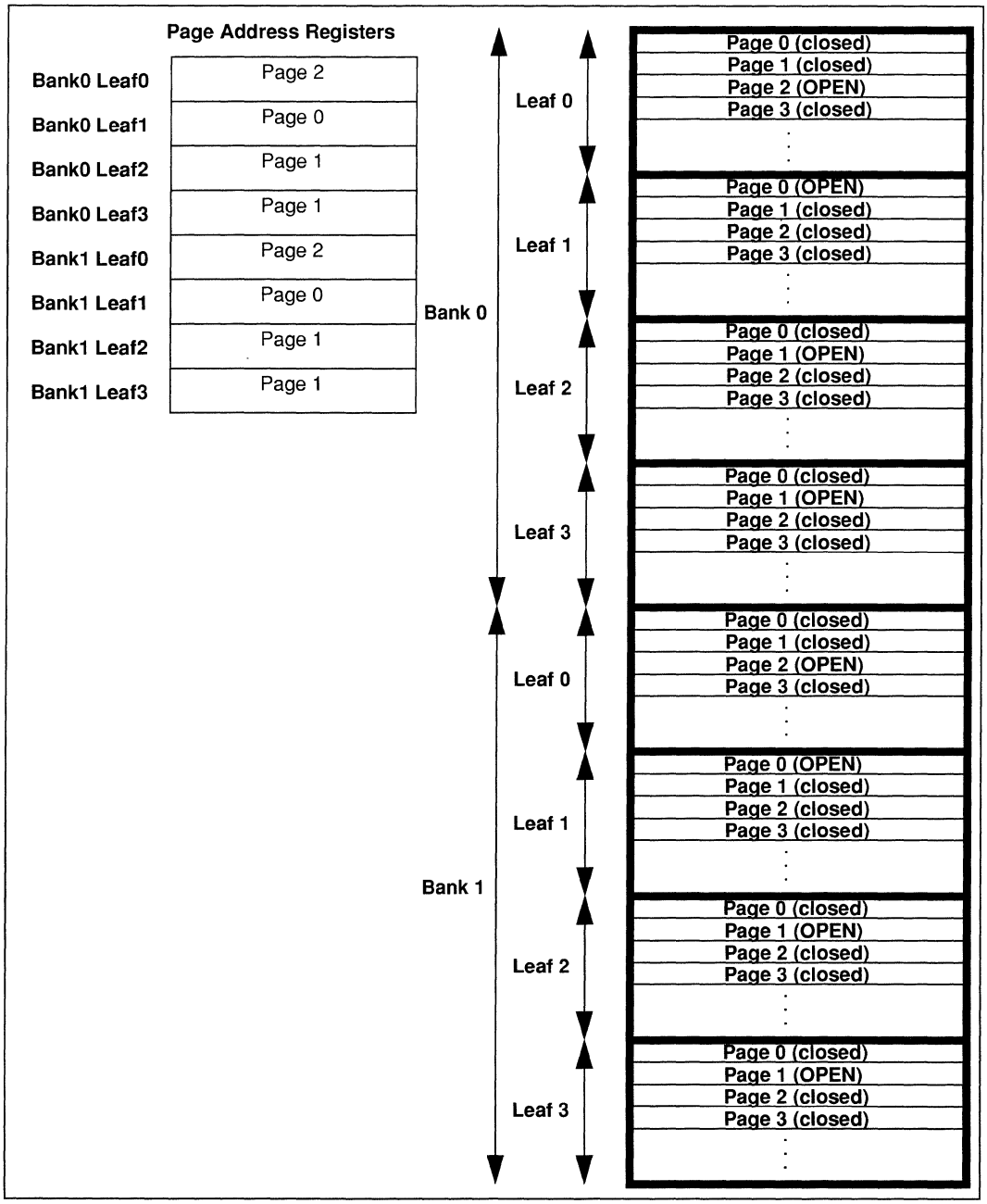


Figure 13-7 illustrates how the logical memory image is partitioned with respect to open and closed pages. If the above image represents a 64 Mbyte SDRAM memory size, each bank is 32 Mbytes and each leaf is 8 Mbytes.

Only one page may be open within each of the leaf blocks. The block sizes depend on the memory sizes implemented in the SDRAM memory subsystem. The page size is always 2 Kbytes. The programmer can optimize SDRAM transactions by partitioning code and data across the leaf boundaries to maximize the number of page hits.

13.2.3.4 SDRAM Commands

The MCU issues specific commands to the SDRAM devices by encoding them on the **SCE[1:0]#**, **SRAS#**, **SCAS#**, and **SWE#** inputs. Table 13-12 lists all of the SDRAM commands understood by SDRAM devices. The MCU supports a subset of these commands.

Table 13-12. SDRAM Commands

Command	Conditions					Comments
	SCE#	SRAS#	SCAS#	SWE#	Other	
NOP	0	1	1	1		No Operation
Mode Register Set	0	0	0	0		Load the Mode Register from **SA[12:0]**
Row Activate	0	0	1	1	**SBA[0]** = Leaf	Activate a row specified on **SA[12:0]**
Read	0	1	0	1	**SBA[0]** = Leaf **SA[10]** = 0	Column burst read
Read w/ Auto-Precharge	0	1	0	1	**SBA[0]** = Leaf **SA[10]** = 1	Column burst read with row precharge at the end of the transfer
Write	0	1	0	0	**SBA[0]** = Leaf **SA[10]** = 0	Column burst write
Write w/ Auto-Precharge	0	1	0	0	**SBA[0]** = Leaf **SA[10]** = 1	Column burst write with row precharge at the end of the transfer
Precharge	0	0	1	0	**SBA[0]** = Leaf **SA[10]** = 0	Precharge a single leaf
Precharge All	0	0	1	0	**SA[10]** = 1	Precharge both leaves
Auto-Refresh	0	0	0	1		Refresh both banks from on-chip refresh counter
Self-Refresh	0	0	0	1	**SCKE** = 0	Refresh autonomously while **SCKE** = 0
Power Down	X	X	X	X	**SCKE** = 0	Power down if both banks precharged when **SCKE** = 0
Stop	0	1	1	0		Interrupt a read or write burst.

NOTES:
1. This table copied from New DRAM Technologies by Steven Przybylski.
2. Shaded boxes indicate commands not supported by 80303 I/O processor. They are included for completeness.

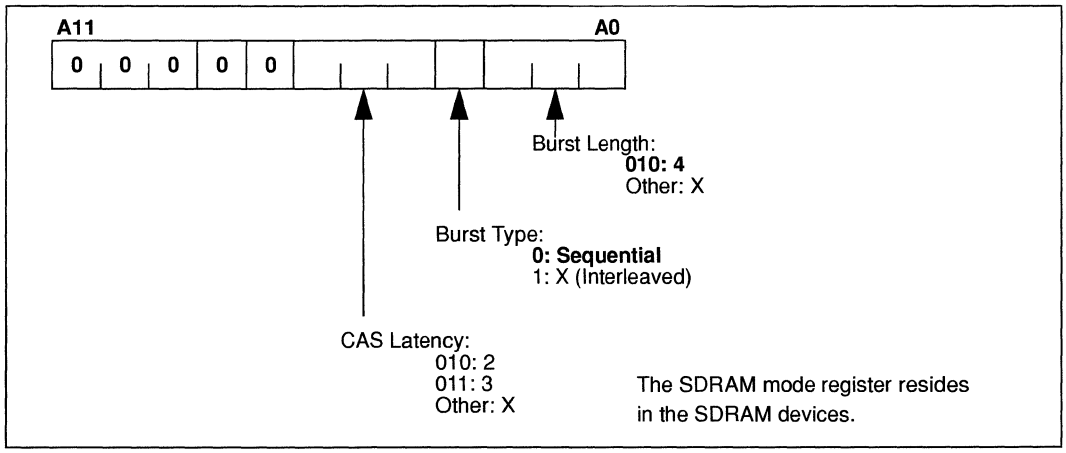
SDRAM commands are synchronous to the clock so the MCU sets up the above conditions prior to the **DCLKOUT[3:0]** rising edge.

13.2.3.5 SDRAM Initialization

Since SDRAM devices contain a controller within the device, the MCU must initialize them specifically. Upon the deassertion of **I_RST#**, software initializes the SDRAM devices with the sequence illustrated with Figure 13-9:

1. The MCU applies the clock (**DCLKOUT[3:0]**) at power up along with system power (clock frequency unknown).
2. The MCU must stabilize **DCLKOUT[3:0]** within 100 µs after power stabilizes.
3. The MCU holds all the control inputs inactive (**SRAS#**, **SCAS#**, **SWE#**, **SCE[1:0]# = 1**) and deasserts **SCKE[1:0]** for a minimum of 1 ms after supply voltage reaches the desired level. Asserting P_RST# achieves this state.
4. Software disables the refresh counter by setting the RFR to zero.
5. Software issues one **NOP** cycle after the 1 ms device deselect. A **NOP** is accomplished by setting the SDIR to 011_2. The MCU asserts **SCKE[1:0]** with the NOP.
6. Software pauses 200 µsec after the **NOP**.
7. Software re-enables the refresh counter by setting the RFR to the required value.
8. Software issues a **precharge-all** command to the SDRAM interface by setting the SDIR to 010_2.
9. Software provides eight **auto-refresh** cycles. An **auto-refresh** cycle is accomplished by setting the SDIR to 100_2. Software must ensure at least T_{rc} cycles between each **auto-refresh** command.
10. Software issues a **mode-register-select** command by writing to the SDIR to program the SDRAM parameters. Setting the SDIR to 000_2 programs the MCU for CAS Latency of two while setting the SDIR to 001_2 programs the MCU for CAS Latency of three. The MCU supports the following SDRAM mode parameters:
    a. CAS Latency (CL) = three or two
    b. Wrap Type (WT) = Sequential
    c. Burst Length (BL) = four

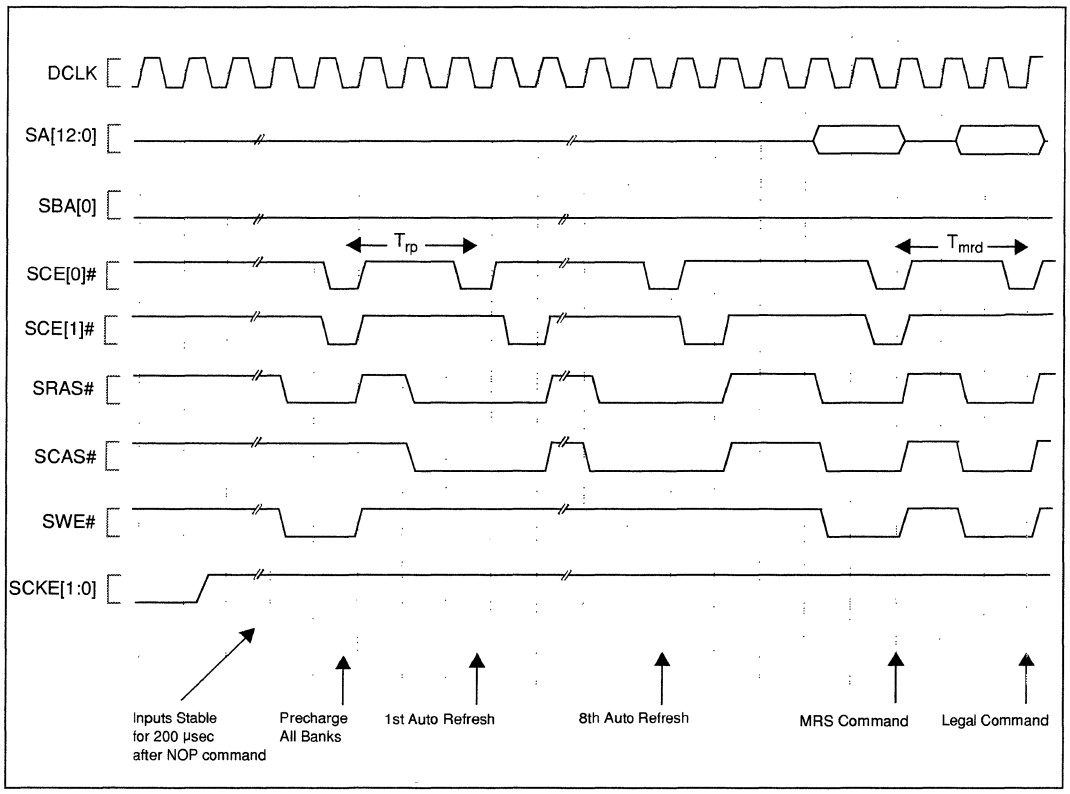
Figure 13-8. Supported SDRAM Mode Register Settings



11. The MCU may issue a **row-activate** command three clocks after the **mode-register-set** command (T_{mrd}).

The waveform in Figure 13-9 illustrates the SDRAM initialization sequence.

Figure 13-9. SDRAM Initialization Sequence (controlled with software)



Since the SDRAM subsystem implements ECC (see Section 13.2.4, "Error Correction and Detection" on page 13-28), initialization software must initialize the entire memory array with the 80303 I/O processor. It is important that every memory location has a valid ECC byte. The BIU optimizes SDRAM accesses by supporting instruction and read prefetching. If the memory array is not initialized, the BIU may attempt to read memory locations beyond the specified word(s). In this case, the MCU will report an ECC error even though software did not specifically request the uninitialized data.

13.2.3.6 SDRAM Mode Programming

The MCU programs the SDRAM devices through a **mode-register-set** command. During the initialization sequence this command sets the SDRAM mode register (see Section 13.2.3.5, "SDRAM Initialization" on page 13-19) by programming the SDIR.

The SDRAM state machine ensures that a **row-activate** command is issued no sooner than T_{mrd} (3) cycles after the **mode-register-set** command.

13.2.3.7 SDRAM Read Cycle

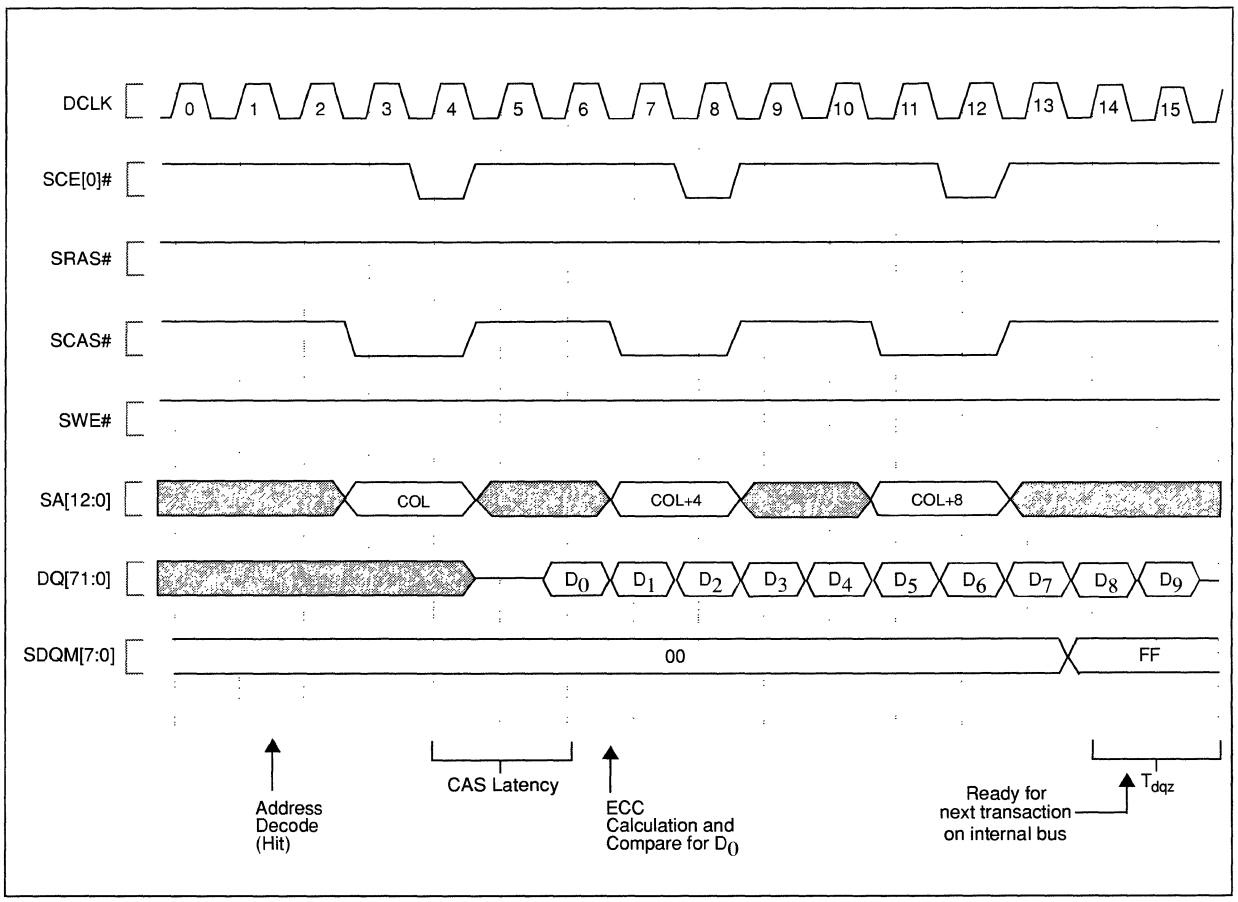
Read performance is optimized for page hits and the MCUs behavior is different for the hit and miss scenario.

Note: To accommodate a heavily loaded memory subsystem (>= 18 SDRAM devices), the MCU drives **SA[12:0]**, **SBA[1:0]**, **SCAS#**, **SRAS#**, and **SWE#** for two clocks in the following SDRAM timing diagrams. The MCU drives **SCE[1:0]#** for one clock since it maintains half the loading of the above signals, **SDQM[7:0]** is unaffected.

Example 13-4. Read Page Hit

A page hit occurs when the current address falls within a row that is currently open. For a page hit, the MCU does not need to open the page (assert **SRAS#**) and avoids the RAS-to-CAS delay achieving greater performance. The waveform for a read that hits a page in bank 0 is illustrated in Figure 13-10.

Figure 13-10. SDRAM Read, 40 bytes, ECC Enabled, BL=4, Page Hit



- The MCU decodes the address to determine if the transaction should be claimed.
  - — If the address falls in the SDRAM address range indicated by the SDBR, SBR0, and SBR1, the MCU claims the transaction.

— During the same cycle, the MCU determines whether or not any of the open pages are hit. If so, then the SDRAM state machine activates the appropriate bank by asserting its chip select for the next cycle.

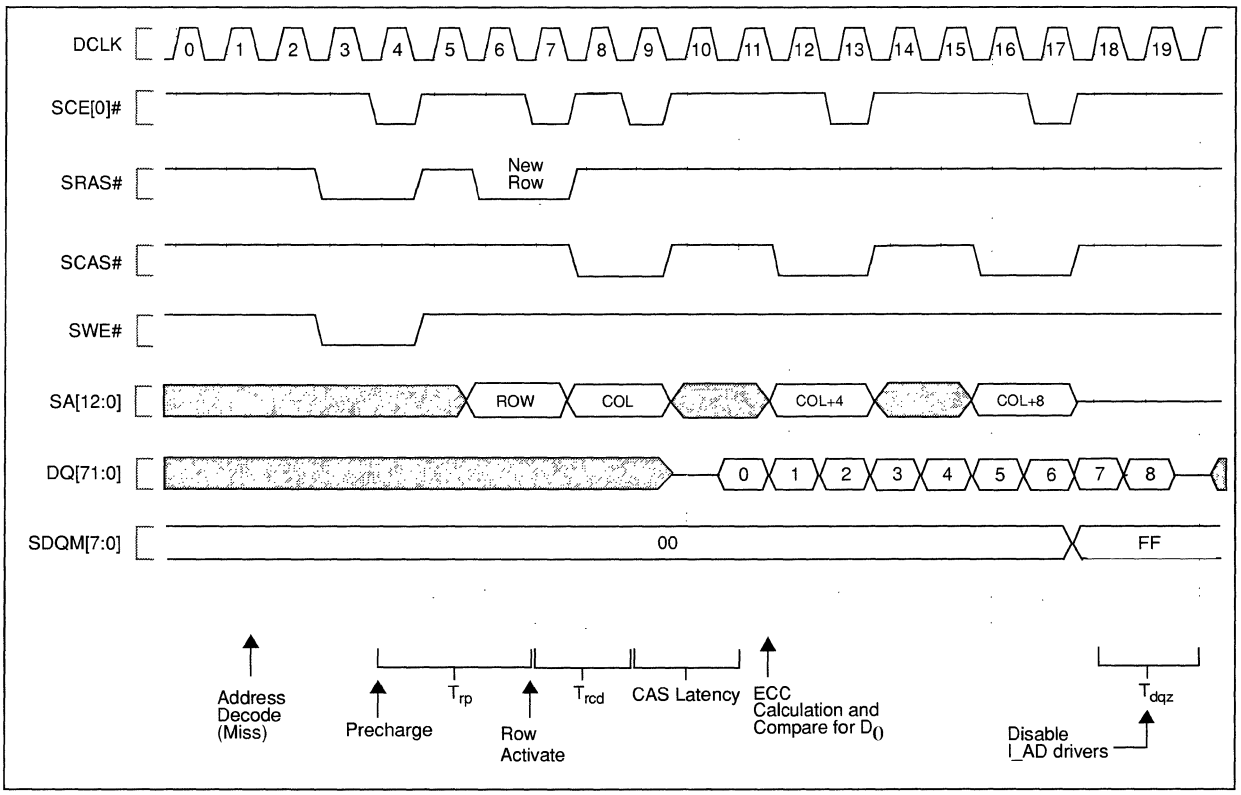
- In the following cycle, the MCU asserts **SCAS#**, deasserts **SWE#**, and places the column address on **SA[12:0]**. This initiates the burst read cycle.
- After the CAS latency expires, the SDRAM device drives data to the MCU.
- Upon receipt of the data, the MCU calculates the ECC code from the data and compares it with the ECC returned by the SDRAM array. Section 13.2.4, "Error Correction and Detection" on page 13-28 explains the ECC algorithm in more detail.
- Assuming the calculated ECC matches the read ECC, the MCU drives the data onto the internal bus.
- For each burst read issued, the memory controller increments the column address by four.

The MCU continues to return data until the master initiating the transaction is satisfied. Once the master terminates the transaction, the MCU ceases issuing read cycles and asserts **SDQM[7:0]** preventing the SDRAM devices from driving the additional data. The additional data returned from the SDRAM devices is discarded.

Example 13-5. Read Page Miss

A read that misses the open pages encounters a miss penalty because the currently open page needs to be closed before the read can be issued to the new page. Refer to Section 13.2.3.3, "Page Hit/Miss Determination" on page 13-16 for the paging algorithm details. Closing a page means issuing a **precharge** command to the row that needs to be closed. Figure 13-11 illustrates a read miss. The new page and the old page are in bank 0.

Figure 13-11. SDRAM Read, 40 bytes, ECC Enabled, BL=4, Page Miss



- The MCU decodes the address to determine if the transaction should be claimed.
  - — If the address falls in the SDRAM address range indicated by the SDBR, SBR0, and SBR1, the MCU claims the transaction.
  - — During the same cycle, the MCU determines whether or not any of the open pages are hit.
- In the following cycle, the MCU closes the currently open page by issuing a **precharge** command to the currently open row.
  - — The MCU waits T_{rp} (3) cycles after the precharge before issuing the **row-activate** command for the new read transaction.
- The **row-activate** command enables the appropriate row.
  - — The MCU asserts **SRAS#**, deasserts **SWE#**, and drives the row address on **SA[12:0]**.
- After T_{rcd} (2) cycles, the MCU issues the **read** command by asserting **SCAS#** while driving the column address on **SA[12:0]**.

The remainder of the read transaction is identical to a "Read Page Hit" on page 13-21 beginning at clock cycle 8.

13.2.3.8 SDRAM Write Cycle

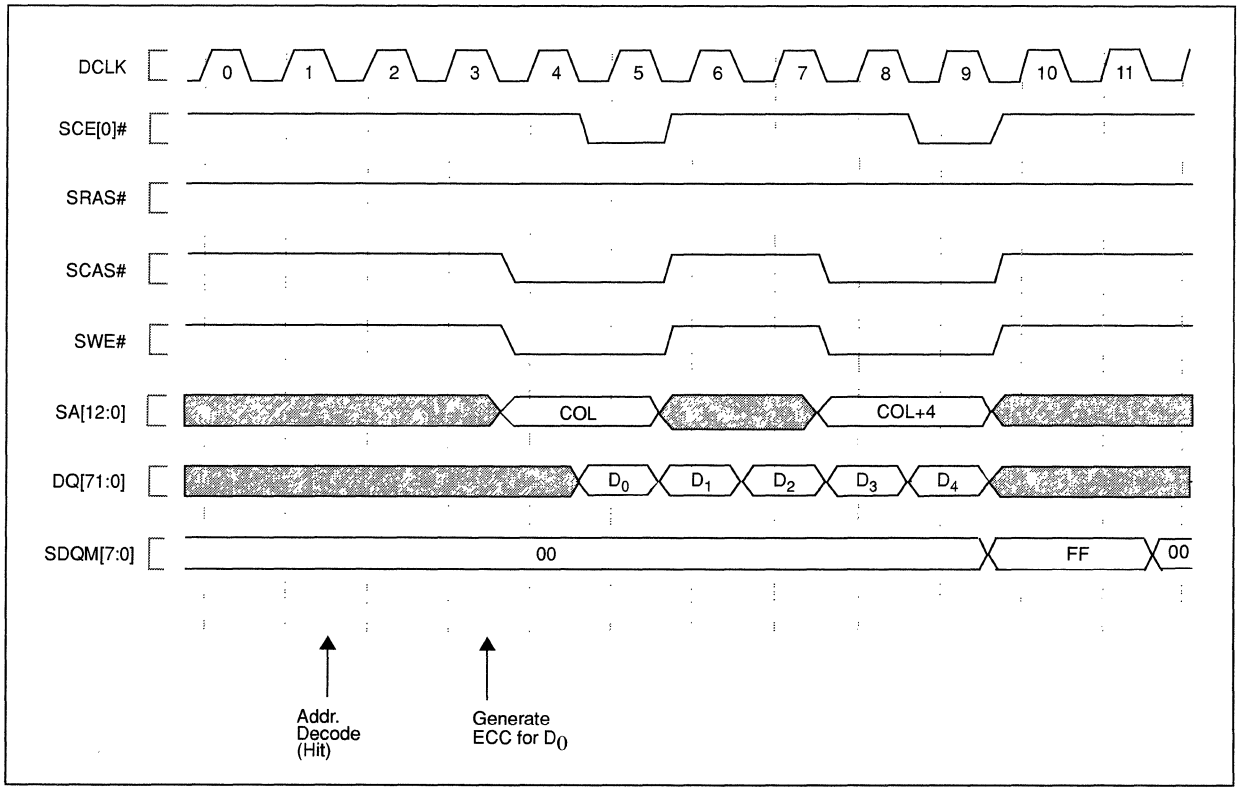
The performance is best for page hits and therefore the MCUs behavior is different for the hit and miss scenario. Section 13.2.4, "Error Correction and Detection" on page 13-28 explains the ECC algorithm in more detail.

Note: To accommodate a heavily loaded memory subsystem (>= 18 SDRAM devices), the MCU drives **SA[12:0]**, **SBA[1:0]**, **SCAS#**, **SRAS#**, and **SWE#** for two clocks in the following SDRAM timing diagrams. The MCU drives **SCE[1:0]#** for one clock since it maintains half the loading of the above signals, **SDQM[7:0]** is unaffected.

Example 13-6. Write Page Hit

For a page hit, the MCU does not need to open the page (assert **SRAS#**) and avoids the RAS-to-CAS delay achieving greater performance. The waveform for a write that hits a page in bank 0 is illustrated in Figure 13-10.

Figure 13-12. SDRAM Write, 40 bytes, ECC Enabled, BL=4, Page Hit



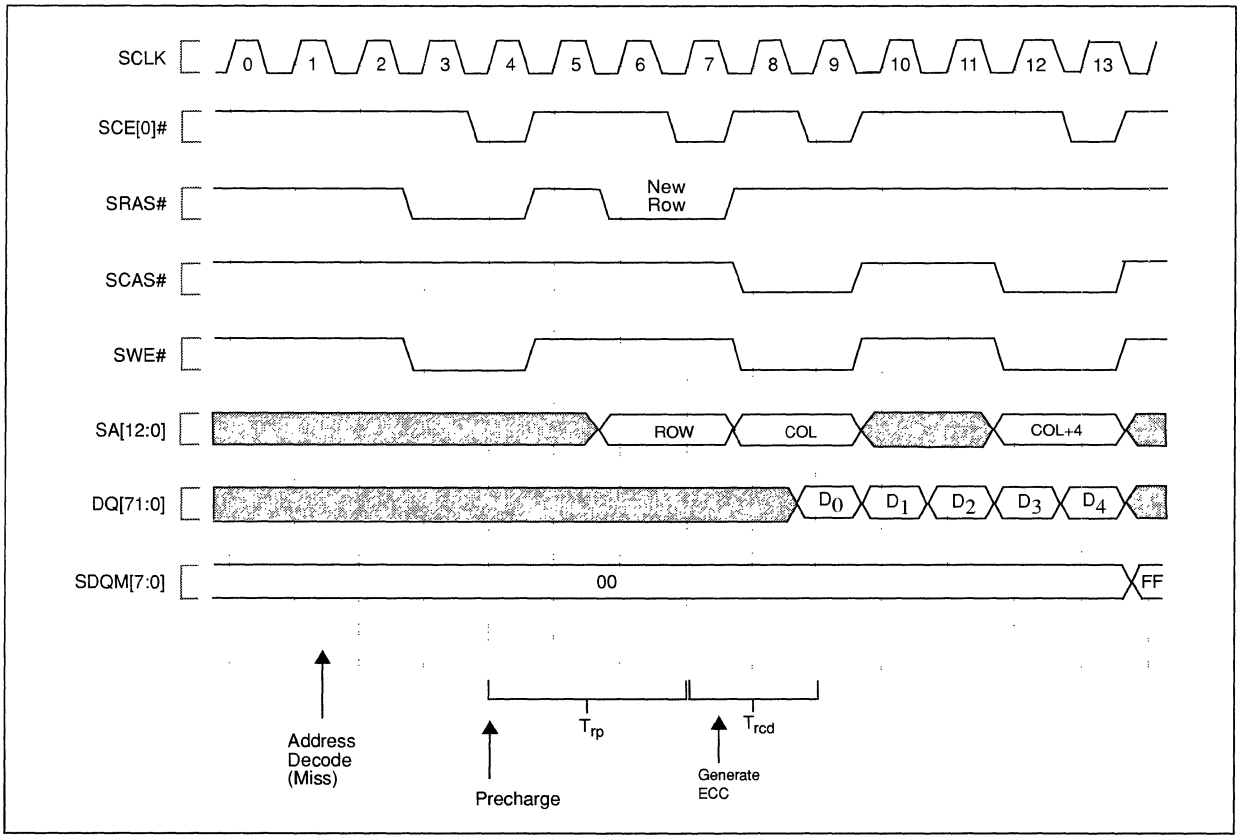
- The MCU decodes the address to determine if the transaction should be claimed.
  - — If the address falls in the MCU address range, the MCU claims the transaction.
  - — During the same cycle, the MCU determines whether or not any of the open pages are hit. If so, then the SDRAM state machine activates the appropriate bank by asserting its chip select for the next cycle.
- The ECC logic generates the ECC code for the data to be written.

- In the following cycle, the MCU asserts **SCAS#**, asserts **SWE#**, and places the column address on **SA[12:0]**. This initiates the burst write cycle. The MCU drives the data to be written and its ECC code to the SDRAM devices.
- The MCU drives the new data on the data bus each cycle until the transaction is completed with the master deasserting **I_FRAME#**.
- If the data to write is not aligned on an 8 byte boundary, the MCU will perform a read-modify-write of the entire 8 byte aligned quad-word and incorporate the new data while regenerating ECC.

Example 13-7. Write Page Miss

A write that misses the open pages encounters a miss penalty because the currently open page needs to be closed before the MCU can issue the write to the SDRAM. Closing a page means issuing a **precharge** command to the row that needs to be closed. Figure 13-13 illustrates a write miss. The new page and the old page are in bank 0.

Figure 13-13. SDRAM Write, 40 bytes, ECC Enabled, BL=4, Page Miss



- Once an external master asserts **I_FRAME#**, the MCU decodes the address to determine if the transaction should be claimed.
  - — If the address falls in the MCU address range, the MCU claims the transaction by asserting **I_DEVSEL#**.
- In the following cycle, the MCU closes the currently open page by issuing a **precharge** command to the currently open row.

— The MCU waits T_{rp} (3) cycles after the precharge command before the MCU issues the **row-activate** command for the new write transaction.

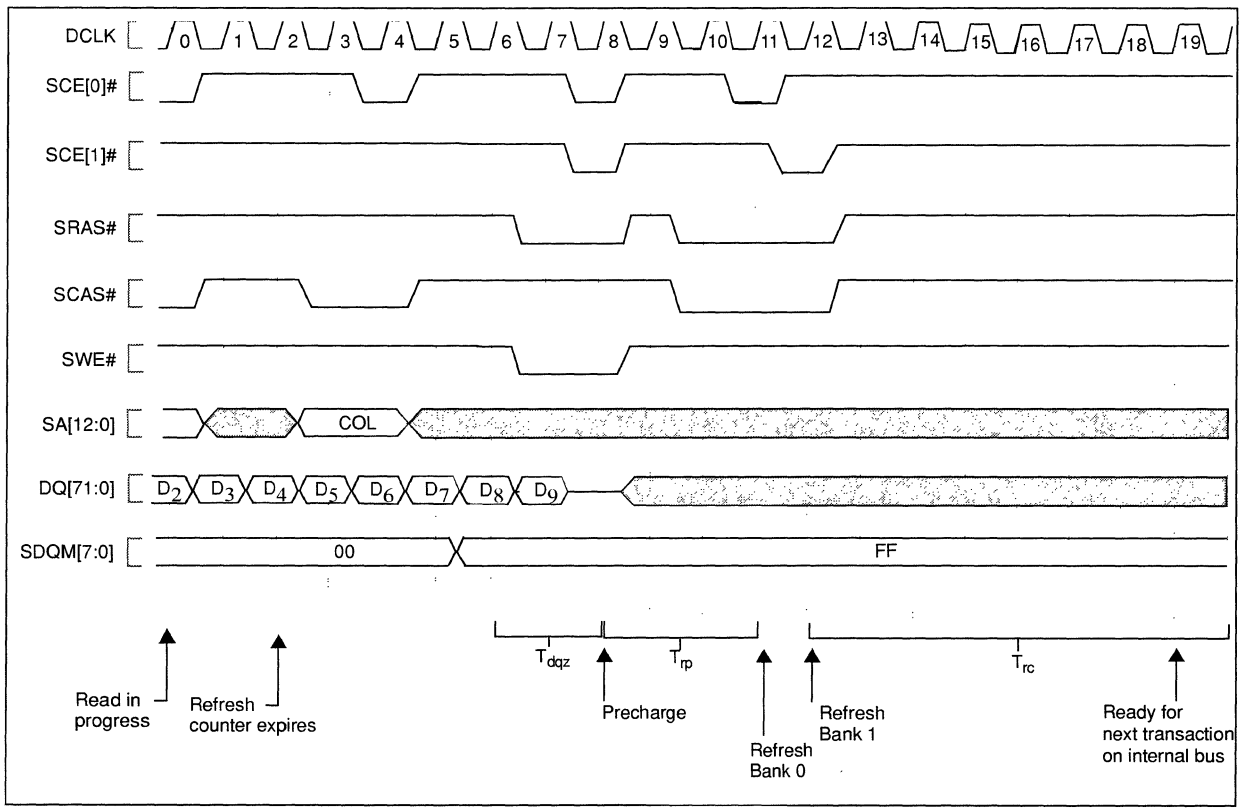
- The MCU issues the **row-activate** command enabling the appropriate row.
  — The MCU asserts **SRAS#** while driving the row address on **SA[12:0]**.
- After T_{rcd} (2) cycles, the MCU issues the **write** command by asserting **SCAS#** and driving the column address on **SA[12:0]**.

The remainder of the write transaction is identical to "Write Page Hit" on page 13-24.

13.2.3.9 SDRAM Refresh Cycle

Since the SDRAM is a dynamic memory, the MCU issues a refresh cycle periodically. The interval of these refresh cycles is programmable in the RFR register. The SDRAM device generates the refresh address internally. The MCU initiates two sequential refresh cycles (one per bank) after the MCUs refresh timer expires and any current transaction is complete. The waveform in Figure 13-14 illustrates the case where the refresh timer expires in the middle of an incomplete read cycle.

Figure 13-14. Refresh Following a Read Cycle



- Once the refresh timer expires, the MCU knows that a refresh cycle is necessary.
  - — The refresh timer continues to count for the next refresh cycle.
- The MCU allows the current read transaction to complete.
  - — Since the MCU is currently reading from the SDRAM array, the refresh cycle is queued until the transaction is complete.
- The MCU closes all open pages with a **precharge-all** command to all the populated SDRAM banks.
  - — The MCU resets the page register valid bits.
- The MCU issues an **auto-refresh** command to SDRAM bank 0.
  - — This command affects all internal leaves.
- In the next cycle, the MCU issues an **auto-refresh** command to SDRAM bank 1.
- After T_{rc} cycles, the MCU can service a new transaction or another refresh cycle.

The refresh timer value is programmed with the RFR depending on the internal bus frequency. The RFR should be programmed to 600H. The longest possible internal bus transaction is writing a 2Kbyte page where each data cycle results in a read-modify-write due to partial writes (see Section 13.2.4.2, "ECC Generation for Partial Writes" on page 13-28). Such a transaction could potentially require queueing two refresh cycles.

13.2.4 Error Correction and Detection

The MCU is capable of correcting any single bit errors and detecting any double bit errors in the 80303 I/O processor's SDRAM memory subsystem. In addition, the ECC logic detects any three or four bit errors which occur in the same nibble. ECC enhances the reliability of a memory subsystem by correcting single bit errors caused by electrical noise or occasional alpha particle hits on the SDRAM devices.

Similar to parity, which simply detects single bit errors, error correction requires an additional 8-bit code word for the 64-bit datum. This means that a memory must have the additional 8-bit error correction code (**SCB[7:0]**) per 64-bit datum (**DQ[63:0]**) resulting in a 72-bit wide memory subsystem. During SDRAM read cycles, the MCU detects single bit errors and corrects the data prior to returning the data on the internal bus. SDRAM write cycles generate the ECC and sends it with the data to the memories.

Scrubbing is the process of correcting an error in the memory array. The chance of an unrecoverable multi-bit error increases if the software does not correct a single-bit error in the array. For the 80303 I/O processor, scrubbing is handled by software. When an error occurs, the MCU logs the error type in ELOG0 or ELOG1 and the address in ECAR0 or ECAR1.

13.2.4.1 ECC Generation

For write operations, the MCU generates the error correction code which is written along with the data. The algorithm for a write transaction is:

```
if data to write is 64 bits wide
    Generate the ECC
    Write the new data and ECC
else {Partial Write}
    Read entire 64-bit data word from memory
    Merge the new data portion with the data from memory
    Generate the new ECC
    Write new data and ECC
```

13.2.4.2 ECC Generation for Partial Writes

If the internal bus master writes less than the data bus width programmed in the SDCR, then the MCU translates the write transaction into a read-modify-write transaction. For a partial write, the MCU calculates the ECC for the modified datum and writes it back. So, if an internal bus master issues a write cycle with partial data, the MCU:

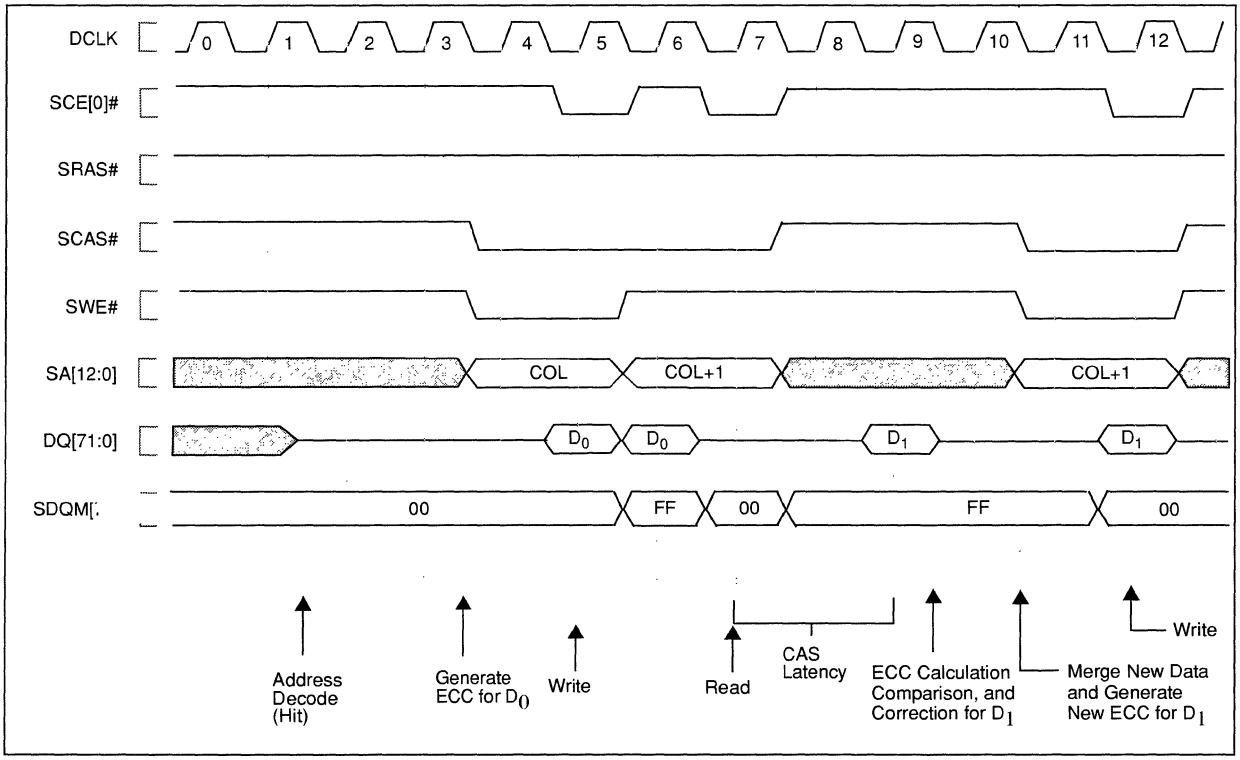
1. Issues a 64-bit read.
2. Modifies the value with the new portion to be written.
3. Calculates the ECC on the modified value.

4. Writes the 64-bit value and ECC.

Figure 13-15 shows an example where the second data of a burst write to bank 0 is less than 64-bits wide. The waveform illustrates how the MCU issues a read-modify-write cycle for the second data (D_1).

Even though the internal bus transaction completes, the MCU may still be busy due to the read-modify-write. Subsequent MCU transactions are retried on the internal bus during this period.

Figure 13-15. Sub 64-bit SDRAM Write (D_1)



13.2.4.3 ECC Checking

If enabled, the ECC logic uses the following ECC read algorithm. This algorithm corrects the data before it's driven onto the internal bus. The ECC algorithm for a read transaction is:

```
Read 64-bit data and 8-bit ECC
Compute the syndrome
if the syndrome <> 0 {ECC Error}
    determine error type
    Register the address where the error occurred
    if error is correctable {single bit}
        Correct data
        Send corrected data to internal bus
        Interrupt core for software scrubbing
    else {uncorrectable}
        if the read cycle is a part of a RMW cycle
            Interrupt the core for uncorrectable error (MCISR[2])
        else
            Target-Abort the transaction
```

When the MCU reads the ECC code from the memory subsystem, it is compared (XORed) with an ECC that the MCU generates from the data read from the memory. The result is called the syndrome. Table 13-13 shows how the MCU decodes the syndrome for SDRAM read cycles.

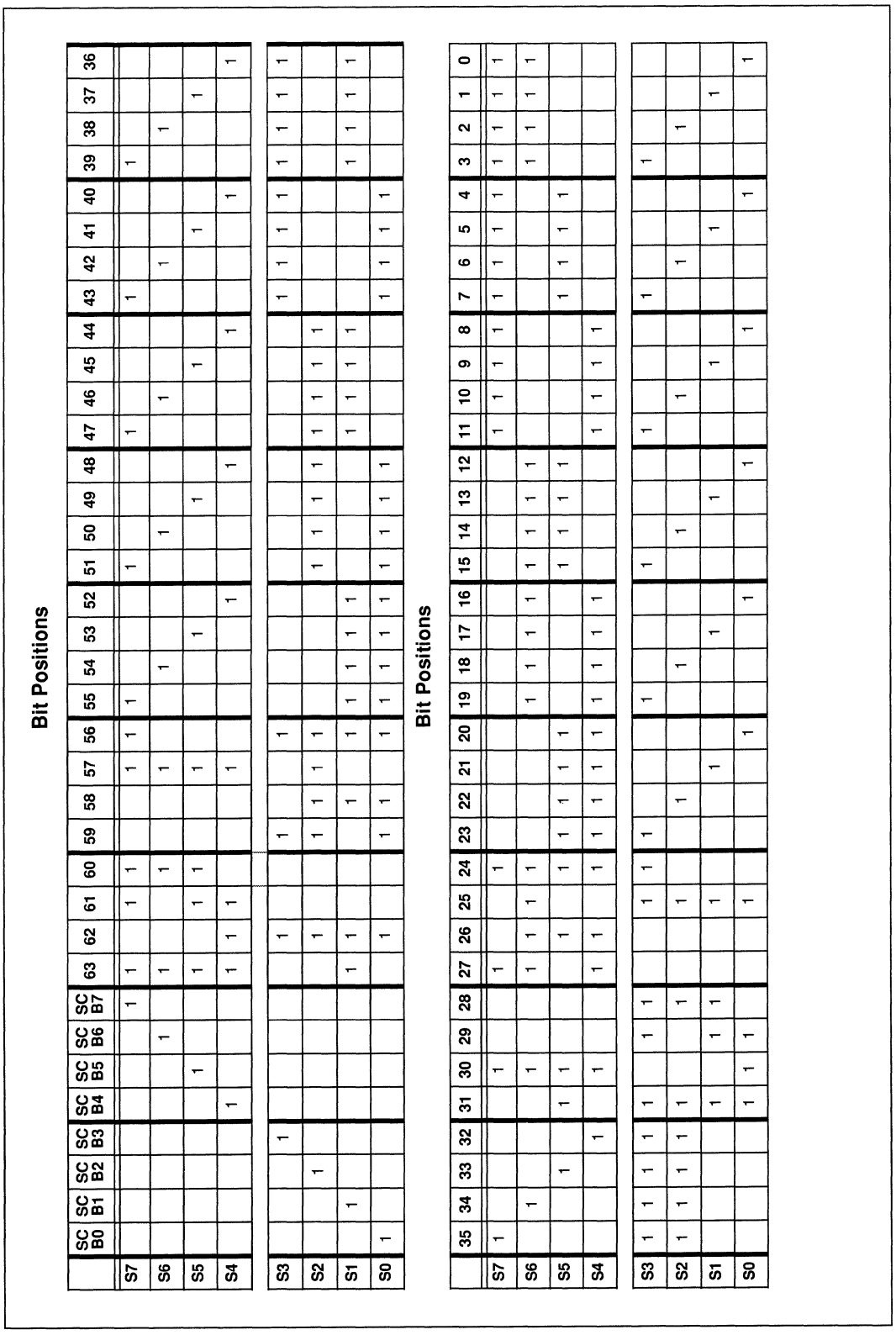
Table 13-13. Syndrome Decoding

Error Type	Symptom
None	The syndrome is 0000 0000.
Single-Bit	The syndrome contains an odd number of ones.
Nibble	One nibble of the syndrome contains 3 bits that are a "1" and the other nibble contains all zeroes. This error is uncorrectable.
Double-Bit	All other syndrome values. This error is uncorrectable.

If decoding the syndrome indicates a double-bit or nibble error (see Table 13-13), the transaction results in a target-abort. If an internal bus master detects a target-abort, the master asserts an NMI to the core. If during a write cycle, the internal bus master has already released the bus, the MCU sets bit 2 in the MCISR and the MCU interrupts the core with an NMI.

If the syndrome indicates a single-bit error, the H-Matrix (Figure 13-16) is used to determine the bit error. For example if the syndrome was 1100 0001 (**S[7:0]**), the error is with bit 0 of **DQ[63:0]**. To correct the error, the MCU inverts bit 0 before driving the data on the internal bus.

Figure 13-16. H-Matrix



If error reporting is enabled in the ECCR and the MCU detects a nibble, single-bit, or double-bit error, the MCU stores the address in ECARx and the syndrome in ELOGx. Software decides how to proceed through an interrupt handler. By registering the address in ECARx, software can identify the faulty DIMM.

For details about the MCU error conditions and how the MMR registers are affected, refer to Section 13.4, "Interrupts/Error Conditions" on page 13-42.

13.2.4.4 Scrubbing

Fixing the data error in memory is called scrubbing. The 80303 I/O processor relies on software scrubbing. Once the MCU detects an error during a read, the MCU writes the address where the error occurred in the ECARx register, updates the ELOGx register with information regarding the error and interrupts the core with an NMI. The core decides how to fix the error through an interrupt handler. Software could decide to perform the scrubbing on:

- the data location that failed
- the entire row of the data that failed
- the entire memory

The interrupt handler should perform the following actions.

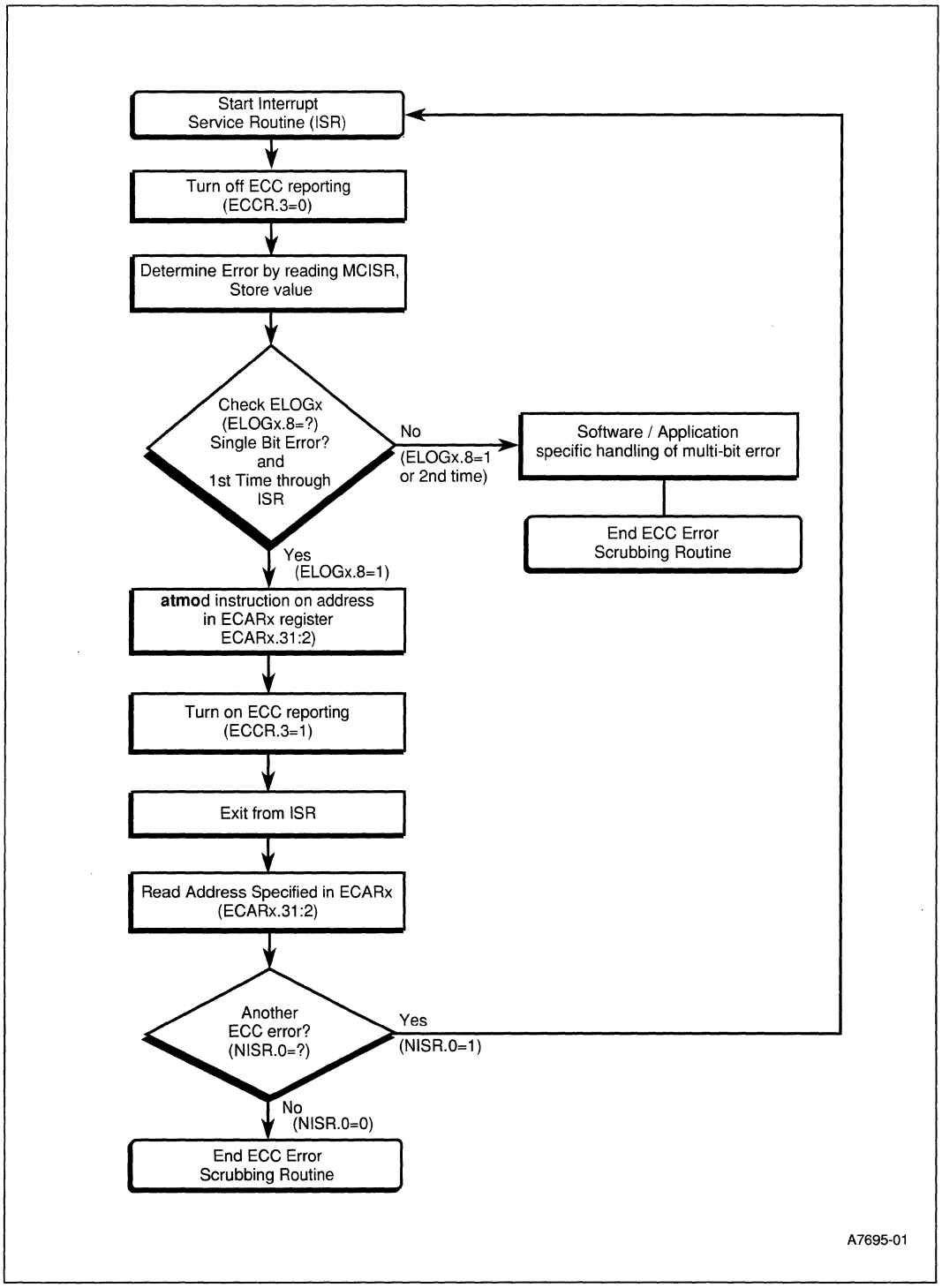
1. Turn off ECC error reporting.

Note: Since the scrubbing routine reads the failed location in order to fix the single-bit error, a second error is reported. Therefore, software should disable single-bit ECC reporting (ECCR[0]) during the scrubbing routine.

2. Read the MCISR register and store the results. Determine which ECAR/ELOG register the MCU used to record the ECC error. Note: This is a read clear register.
3. Check the ELOGx register for information about the ECC error. Determine if the error is a single bit or multi-bit error. Double-bit or nibble errors cannot be fixed. If it's a single-bit error, continue.
4. Do an **atmod** (read, modify, write) to the address specified in the ECARx register. Even though an atmod instruction is only a 32 bit instruction, this will work for either 32 bit memory mode or 64 bit memory mode. The error is fixed by the MCU when it reads the memory location and stores the value in the register, then the corrected data is written back to the memory location.
5. Turn on ECC error reporting.
6. Exit from ISR.
7. Read from the address specified in the ECARx register again.
8. If no errors are reported (through NMI interrupt), the scrubbing procedure worked.
9. If another error is generated, treat this as a multi-bit error.

The diagram below is a flowchart version of the scrubbing routine.

Figure 13-17. Scubbing Routine Flow Chart



13.2.4.5 ECC Testing

The MCU implements the ECTST register providing the programmer the ability to test error handling software. For write transactions, the ECTST register value is XORed with the generated ECC. This inverts the bits where the mask is set prior to writing the ECC to memory. When the MCU reads the address later, the ECC mismatches and the error condition occurs (see Section 13.4, "Interrupts/Error Conditions" on page 13-42).

13.2.5 Overlapping Memory Regions

The MCU supports four independent memory regions:

- MMR Memory Space
- SDRAM Memory Space
- Two Flash Memory Spaces

The MMR memory space is fixed at 1500H to 15FFH. Software programs the SDRAM memory region by providing a base address in SDBR and each of the two bank boundaries in SBR0 and SBR1. The first Flash address range is programmed with a base register in FEBR0 and the bank size in FBSR0. FEBR1 and FBSR1 defines the second address range.

While it is not recommended, the four ranges could overlap. In the case of a memory region overlap, refer to Table 13-14 for the priority rules.

Table 13-14. Overlapping Address Priorities

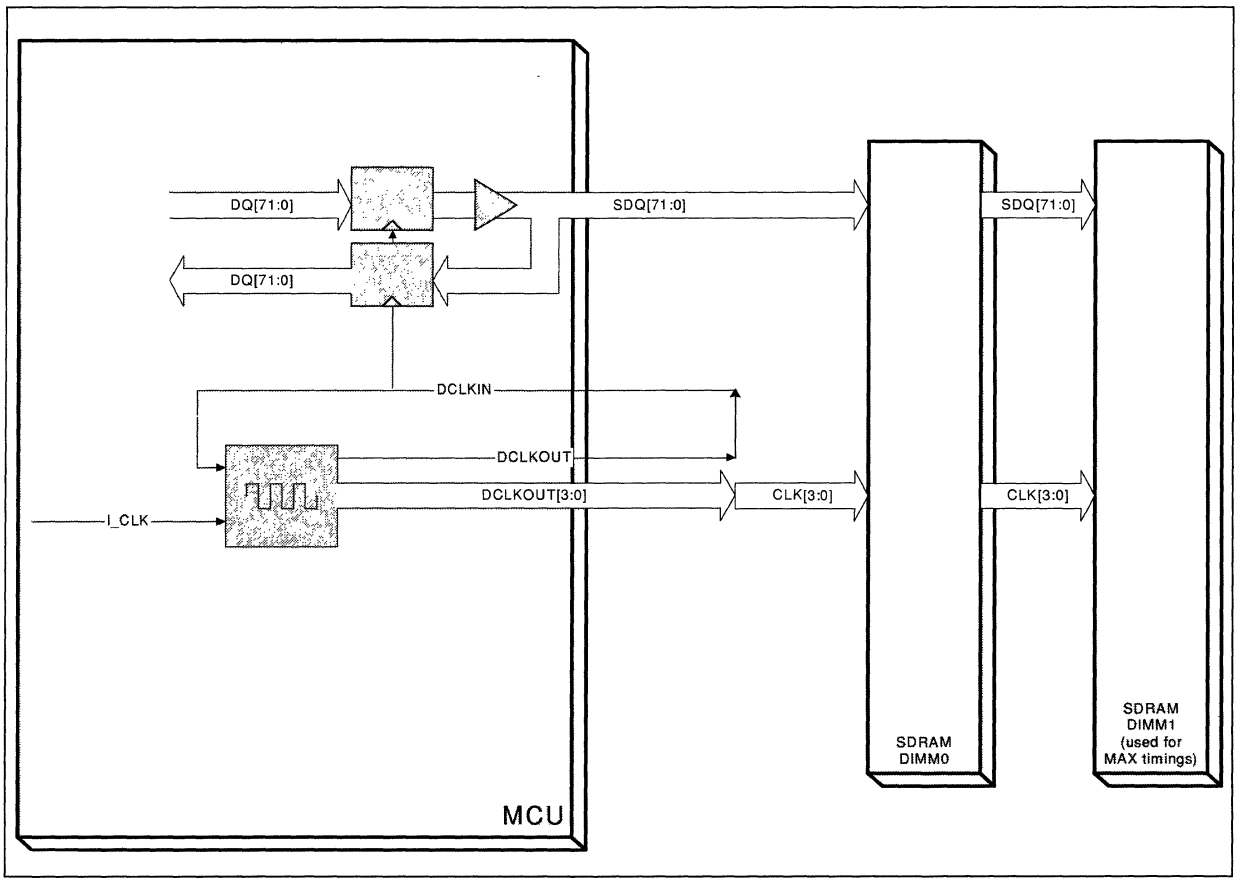
Priority	Address Region
Highest	Memory Mapped Register Address Space
	Flash Bank 0 Address Space
	Flash Bank 1 Address Space
Lowest	SDRAM Address Space

13.2.6 SDRAM Clocking

The MCU provides 4 clocks (**DCLKOUT[3:0]**) to the SDRAM memory subsystem at 100 MHz. The 72-bit 2-bank SDRAM DIMM specification requires 4 clocks to distribute the loading across eighteen x8 SDRAM components.

DCLKOUT is driven back into the 80303 I/O processor as **DCLKIN** so that **DCLK[3:0]** may be skewed back to accommodate for the clocks' flight time and be compatible with 133 MHz SDRAM technologies. The amount of skew is determined by the board trace length. Refer to Figure 13-18 for the layout diagram. SDRAM layout details as well as the clocking strategy are recommended in the 80303 I/O processor Design Guide.

Figure 13-18. SDRAM Clocking



13.3 Power Failure Mode

The 80303 I/O processor is an I/O processor used in server applications including networking and storage. Specifically, the storage applications supported utilize the 80303 I/O processor as the IOP for a SCSI RAID disk subsystem. The integrated memory controller supports up to 512 Mbytes of local memory used for disk caching. The local memory is used for the temporary storage of disk writes which greatly improves disk performance.

For the 80303 I/O processor memory controller, up to 512 Mbytes may be stored within the disk cache. While the host assumes all written data is stored on the non-volatile disk subsystem, the IOP must ensure that eventually all the data in the disk cache is actually stored onto disk.

The power supply could fail to provide power to the I/O subsystem in the case of a power outage or a failed power supply. It is imperative that the cached data within the IOP's local memory is not lost. If power fails, the local memory subsystem must remain powered with a battery backup and some agent must continue to refresh at the appropriate interval specified by the memory component datasheet.

This proposal defines the mechanism which the 80303 I/O processor's memory controller ensures that the data within local memory is not lost during a power failure.

13.3.1 Theory of Operation

SDRAM technology provides a simple way of enabling data preservation through the **self-refresh** command. This command is issued by the memory controller and the SDRAM will refresh itself autonomously with internal logic and timers. The **self-refresh** command is defined in Table 13-12.

The SDRAM device will remain in self-refresh mode as long as:

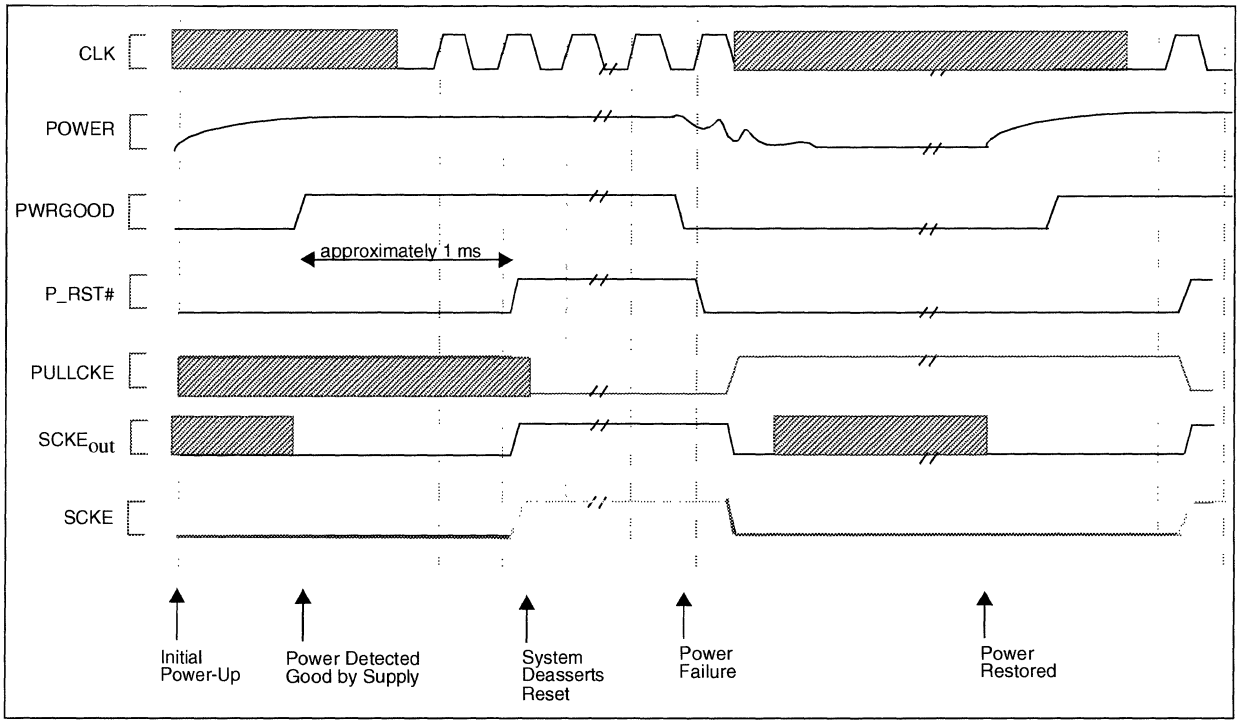
- The device continues to be powered.
- **SCKE** is held low until the memory controller is ready to control the SDRAM once again.

Power to the SDRAM subsystem is ensured with an adequate battery backup and a reliable method for switching between system power and battery power. The memory controller is responsible for deasserting **SCKE[1:0]** when issuing the **self-refresh** command but while power gradually drops, **SCKE[1:0] MUST** remain deasserted regardless of the state of V_{cc} powering the 80303 I/O processor.

13.3.2 Power Failure Sequence

Figure 13-19 illustrates the sequence of events during a power failure as defined by *PCI Local Bus Specification*, Revision 2.2.

Figure 13-19. Power Failure Sequence



13.3.2.1 Power Failure Impact on the System

Upon initial power-up a power supply provides the appropriate voltage to the system. The voltage level will increase at a rate that is dependent on the type of power supply used and the components in the system. These variables are not certain, so the power supply often provides a signal called **PWRGOOD** which indicates the time when the voltage has reached a reliable level. The power supply deasserts **PWRGOOD** if the voltage level drops below a certain minimum threshold.

PCI Local Bus Specification, Revision 2.2 indicates that once **PWRGOOD** is deasserted, the PCI reset pin (P_RST#) is asserted in order to float the output buffers. In the specification T_{fail} is defined as the time when P_RST# is asserted in response to the power rail going out of specification. T_{fail} is the minimum of:

- 500 ns from either power rail going out of specification (exceeding specified tolerances by more than 500mV)
- 100 ns from the 5V rail falling below the 3.3V rail by more than 300mV

13.3.2.2 System Assumptions

This proposal makes specific assumptions about the system's behavior during a power failure. If the below assumptions are not guaranteed, it is the vendor's responsibility to ensure them.

1. P_RST# is asserted to the 80303 I/O processor when there is at least 1 us of reliable power remaining and stays asserted as long as reliable power is available. This is required so that the memory controller can execute it's power-failure state machine in response to the assertion of P_RST#.
2. The PCI clock will continue to run for at least 20 clock cycles after P_RST# is asserted. The 80303 I/O processor requires a PCI clock operating in the specified range of 16-66 Mhz in order to complete the power fail sequence and put the SDRAM in self-refresh mode.

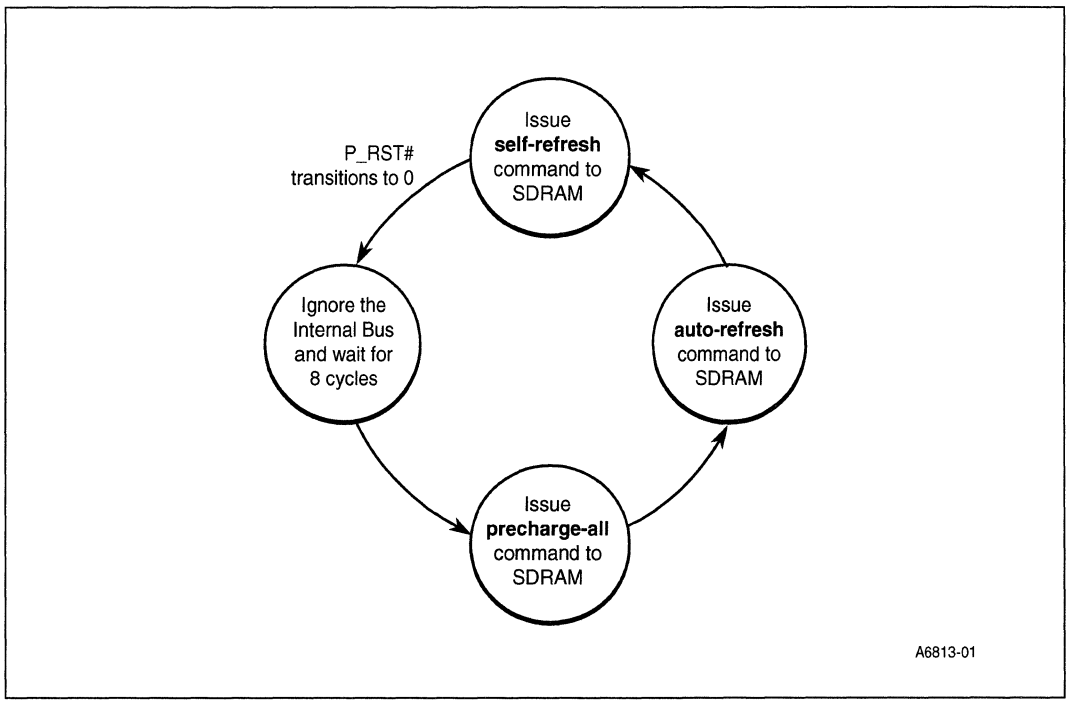
13.3.3 Memory Controller Response to I_RST#

The memory controller assumes a power failure condition whenever **I_RST#** is asserted. Note that **I_RST#** is asserted in response to the assertion of P_RST#. If P_RST# indicates a true power failure, then battery-backup power is supplied to the SDRAM array. If P_RST# indicates any condition other than a power failure, the SDRAM array will be powered down and any attempt to issue the **self-refresh** command is ignored by the memory.

Due to the high loading on **SCKE** and the requirement of 100 MHz operation, the memory controller must drive two copies to the SDRAM DIMM. The board layout will distribute the two **SCKE[1:0]** signals between the two SDRAM banks equally.

Refer to Figure 13-20 for a high-level state machine representation illustrating the memory controller's behavior during a power failure condition.

Figure 13-20. Power Failure State Machine

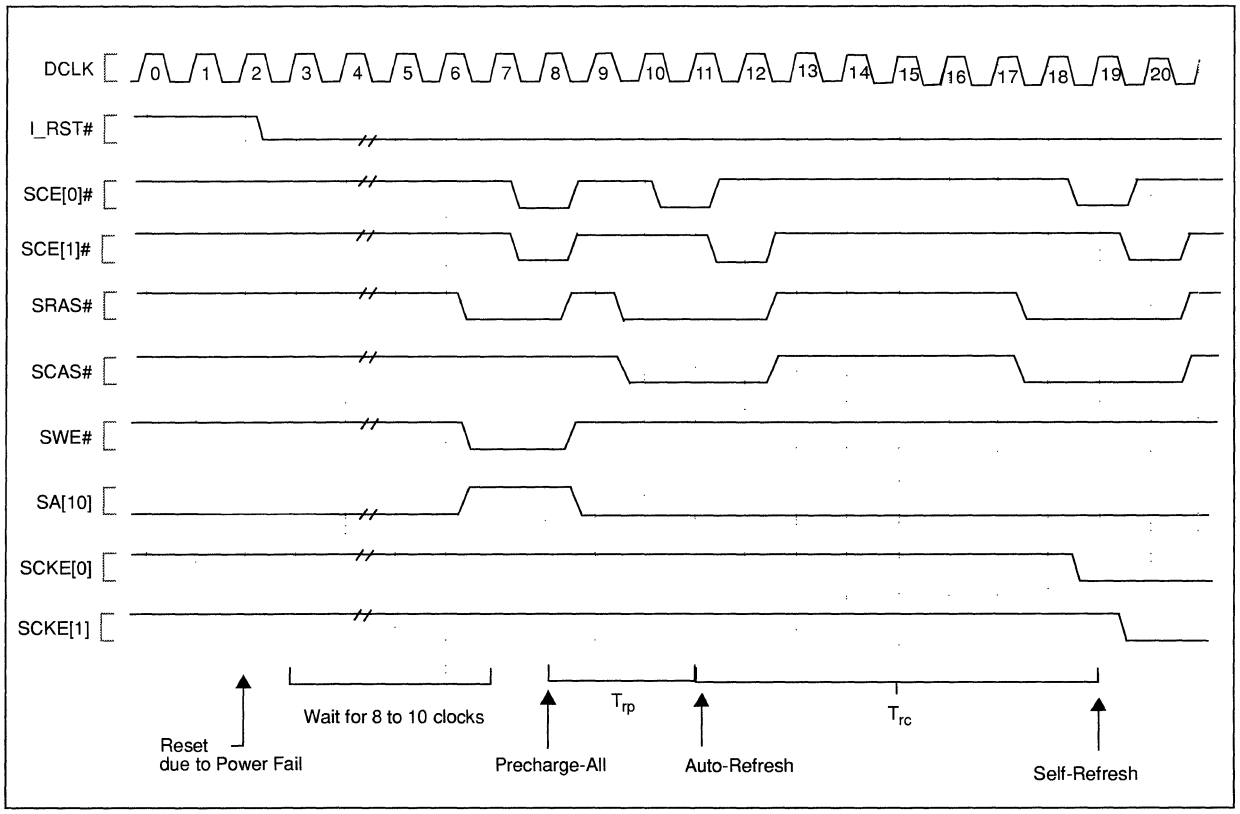


Once the memory controller detects the assertion of **I_RST#**, the memory controller will:

- Ignore the internal bus. The internal bus is invalid during **I_RST#** so the memory controller should not assume any valid state.
- Wait for eight clocks to allow any previous SDRAM bus activity (i.e., read burst) to complete before the memory controller issues the **precharge-all** command.
- Deactivate all SDRAM leaves with the **precharge-all** command.
- Issue an **auto-refresh** command and wait T_{rc} (8) clocks.
- Issue a **self-refresh** command to the SDRAM devices and continue to deassert **SCKE[1:0].**

Figure 13-21 illustrates the SDRAM waveforms upon the assertion of **I_RST#.**

Figure 13-21. Power Failure Sequence



SCKE[1:0] must be held low throughout the power-down period. The memory controller drives it low initially with the **self-refresh** command, but an external pull-down is required to continually drive it low when the 80303 I/O processor loses power. External logic ensures that **SCKE[1:0]** is held low after the memory controller initially deasserts it. Likewise, the external logic must stop driving **SCKE[1:0]** low once P_RST# is deasserted by the system. Figure 13-22 shows one example of the external logic required for power failure mode.

As long as the SDRAM memory subsystem is powered with a battery source and **SCKE[1:0]** is held low, the SDRAM preserves its memory image.

When power is restored, the system asserts P_RST# to the 80303 I/O processor. While the 80303 I/O processor is reset, **SCKE[1:0]** is held low by the memory controller. After P_RST# is deasserted (and subsequently, **I_RST#** is deasserted), the 80303 I/O processor must be re-initialized to reset the CAS Latency parameter. The MRS command issued to the SDRAM subsystem re-asserts **SCKE[1:0]** to ones and the memory controller resumes refreshing. The SDRAM initialization sequence does not affect the memory contents. For more details about the SDRAM initialization sequence, refer to Section 13.2.3.5, "SDRAM Initialization" on page 13-19.

Note: The power failure mechanism in the memory controller is not responsible for maintaining the 80303 I/O processor state. The purpose of this mechanism is to maintain the memory so that any data cached in the local memory can be flushed once power is restored. Any data queued within the 80303 I/O processor's components (ATUs, BIU, etc.) will be lost.

13.3.3.1 External Logic Required for Power Failure

Assertion of P_RST# During Power Failure

Unfortunately, the 80303 I/O processor will not function down to the minimum voltage level of 2.5 volts after which the PCI specification guarantees P_RST# will be asserted during a power failure. As a result, if the power failure mechanism is to be used, the P_RST# input pin must be an OR function of the system's P_RST# signal and the output of a voltage detection circuit that will trigger below the 3.0 volts guaranteed by the PCI specification during normal operations but above the 2.5 volts where the device is not functional.

SCKE Control

Refer to Figure 13-22 for a state machine of the external logic required to control **SCKE[1:0]** for power failure mode. Actual implementations may vary. This state machine can be implemented in a programmable logic device illustrated in Figure 13-23.

Figure 13-22. External Power Failure State Machine

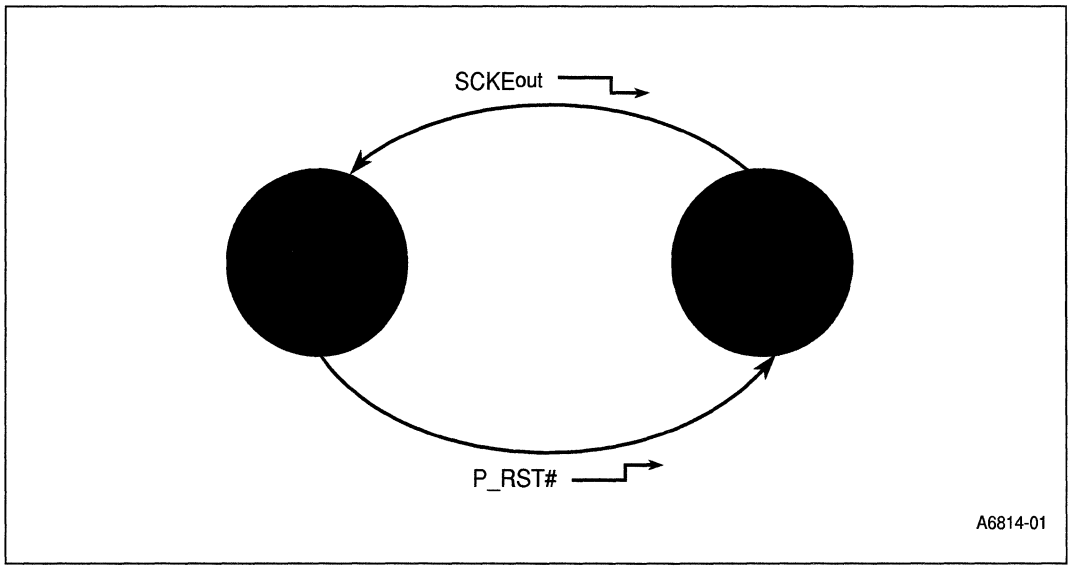
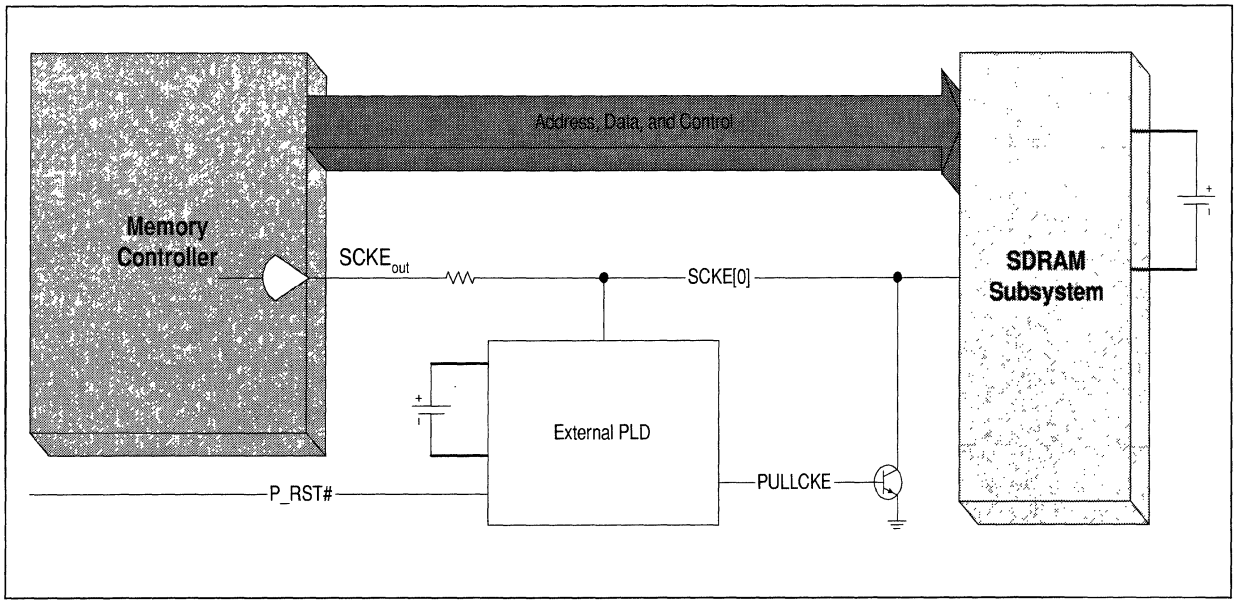


Figure 13-23. External Power Failure Logic in the System



The implementation illustrated in Figure 13-23 requires that all external logic is powered by V_{batt}. The edge detect state machine will turn on the pull-down when the MCU deasserts **SCKE[1:0]**. As long as V_{batt} is active, **SCKE[1:0]** is held low. Once the memory controller is reset, the rising edge of P_RST# deactivates the pull-down. The memory controller will reliably control **SCKE[1:0]** at this point driving it low.

Note: Figure 13-23 shows logic for one of the **SCKE** signals. The loading of this signal is large enough that two signals are required (one per SDRAM bank). The above logic will need to be replicated for each **SCKE[1:0]**.

Initialization of the Power Failure State Machine

The 80303 I/O processor's power failure state machine can not be initialized by the P_RST# pin like the rest of the device's state machines since the power failure state machine is operational during P_RST# assertion.

The 80303 I/O processor provides a dedicated input pin, **PWRDELAY** that will be used to initialize the MCUs power failure state machine when **deasserted**. This signal should be driven by external circuitry that will assert **PWRDELAY** following the initial deassertion of P_RST#. **PWRDELAY** will not be deasserted until power has truly failed.

This circuitry could consist of a capacitor that is charged up through a FET enabled via the deassertion of P_RST#. The only discharge path available to the capacitor would be through a Zener diode connected to VCC, thus **PWRDELAY** would be deasserted only when power has truly failed.

13.4 Interrupts/Error Conditions

The MCU has two conditions which require intervention from the 80303 I/O processor core. If a single-bit error is detected during a read cycle, the MCU can fix the error but software needs to fix the error in the memory array. If a double-bit or nibble error is detected, the core decides how to handle the condition. For all ECC errors, the MCU records the master of the transaction resulting in the error in ELOGx[18:16] and interrupts the core.

If the MCU detects an ECC error during a read or write cycle[1], MCISR[0] or MCISR[1] is set to 1. Whenever the MCU toggles one of the MCISR bits from 0 to 1, an NMI is generated to the core.

Table 13-15 shows how the MCU responds to error conditions.

Table 13-15. MCU Error Response

Error Type	MCU Action
Single-Bit during a read or write	Fix Error
Double-Bit/Nibble during a read	Target Abort the transaction
Double-Bit/Nibble during a write • Internal Bus write cycle not yet complete	Target Abort the transaction Do not write the data in error to SDRAM array
Double-Bit/Nibble during a write • Internal Bus write cycle completed	Do not write the data in error to SDRAM array

Note: If ECC reporting is enabled with ECCR[1] or ECCR[0] and an ECC error occurs, MCISR[1] or MCISR[0] is set and ELOGx/ECARx logs the error in addition to the above table actions.

13.4.1 Single-Bit Error Detection

When enabled, the MCU interrupts the core when the ECC logic detects a single-bit error by setting the appropriate bit in the MCISR register. The core knows the interrupt was caused by a single-bit error by polling the ELOG0 or ELOG1 register. The MCU ensures that correct data is transferred onto the internal bus but the interrupt handler is responsible for scrubbing the error in the array (refer to Section 13.2.4.4, "Scrubbing" on page 13-32).

An example flow for a single-bit error with error detection and reporting enabled is:

- A single-bit ECC error is detected on the data bus (**DQ[63:0]**) by the MCU.
- The MCU fixes the error prior to sending the data onto the internal bus.
- The MCU clears ELOG0[8] indicating a single-bit error.
- The MCU records the master of the transaction that resulted in an error in ELOG0[18:16]
- The MCU loads ELOG0[7:0] with the syndrome that indicated the error.
- The MCU loads ECAR0[31:2] with address where the error occurred.
- Since the core needs to scrub the error in the array, the MCU sets MCISR[0] to 1 (assuming it is not already set).
  - — Setting any bit in the MCISR causes an NMI to the core.

1. Any error condition during a write cycle actually occurs while performing the read portion of a read-modify-write on a sub-64-bit write. See Section 13.2.4.1, "ECC Generation" on page 13-28 for details.

- Software polls the interrupt status register. Bit 0 set to 1 indicates that the first error has occurred.
- Software polls ELOG0 and ECAR0 and scrubs the error at the location specified by ECAR0.
- Software writes a 1 to MCISR[0] thereby clearing it.

If software does not perform error scrubbing, the probability of an unrecoverable double-bit error increases for the memory location containing the single-bit error.

ECARx and ELOGx remain registered until software explicitly clears them.

If a second error occurs before software clears the first by resetting MCISR[0] or MCISR[1], the error is recorded in the remaining ELOGx/ECARx register. If none are available, the error is not logged but the MCU carries out the action described in Table 13-15.

13.4.2 Double-Bit/Nibble Error Detection

If a multi-bit error occurs during a read or write transaction and error reporting is enabled, the MCU sets MCISR[0] or MCISR[1] which asserts an NMI to the core. Upon receiving an NMI, the core knows the interrupt was caused by a double-bit or nibble error by polling the ELOGx registers.

When the MCU detects a double-bit or nibble error during a read cycle and error reporting is enabled in the ECCR, the MCU target aborts the transaction indicating to the internal bus masters that an unrecoverable error has been detected. The MCU records the error type in ELOGx and the address in ECARx.

When the MCU detects a double-bit or nibble error during a write cycle and error reporting is enabled in the ECCR, the MCU records the first nibble or double-bit error by programming ELOGx and ECARx. The MCU cannot correct the data before sending it on **DQ[63:0]** so the MCU aborts the read-modify-write cycle.

If a second error occurs before software clears the first by resetting MCISR[0] or MCISR[1], the error is recorded in the remaining ELOGx/ECARx register. If none are available, the error is not logged but the MCU carries out the action described in Table 13-15.

It is the interrupt handler's responsibility to decide how to handle this error condition and clear the MCISR.

13.5 Reset Conditions

Once P_RST# is deasserted and 200 us have passed, software must issue the initialization sequence defined in Section 13.2.3.5, "SDRAM Initialization" on page 13-19. After initialization, the SDRAM devices are ready to be written to or read from. Reads issued prior to a write to the same address results in an ECC error (if enabled) and is not recommended.

While P_RST# is asserted, the MCU initializes its MMR registers to the states defined in Section 13.6, "Register Definitions" on page 13-44.

13.6 Register Definitions

A series of configuration registers control the MCU. Software can determine the status of the MCU by reading the status registers. Table 13-16 lists all of the MCU registers which are detailed further in proceeding sections.

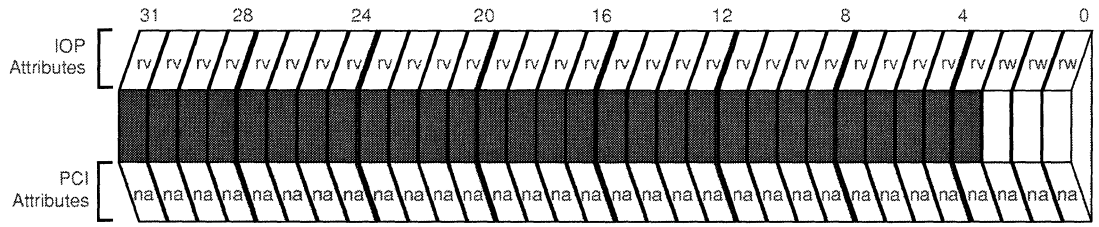
Table 13-16. Memory Controller Register Table

Section, Register Name - Acronym (Page)
Section 13.6.1, "SDRAM Initialization Register - SDIR" on page 13-45
Section 13.6.2, "SDRAM Control Register - SDCR" on page 13-46
Section 13.6.3, "SDRAM Base Register - SDBR" on page 13-47
Section 13.6.4, "SDRAM Boundary Register 0 - SBR0" on page 13-48
Section 13.6.5, "SDRAM Boundary Registers 1 - SBR1" on page 13-49
Section 13.6.6, "ECC Control Register - ECCR" on page 13-50
Section 13.6.7, "ECC Log Registers - ELOG0, ELOG1" on page 13-51
Section 13.6.8, "ECC Address Registers - ECAR0, ECAR1" on page 13-52
Section 13.6.9, "ECC Test Register - ECTST" on page 13-53
Section 13.6.10, "Flash Base Register 0 - FEBR0" on page 13-54
Section 13.6.11, "Flash Base Register 1 - FEBR1" on page 13-55
Section 13.6.12, "Flash Bank Size Register 0 - FBSR0" on page 13-56
Section 13.6.13, "Flash Bank Size Register 1 - FBSR1" on page 13-57
Section 13.6.14, "Flash Wait States Registers - FWSR0, FWSR1" on page 13-58
Section 13.6.15, "Memory Controller Interrupt Status Register - MCISR" on page 13-59
Section 13.6.16, "Refresh Frequency Register - RFR" on page 13-60

13.6.1 SDRAM Initialization Register - SDIR

The SDRAM Initialization Register (SDIR) is responsible for programming the operation of the SDRAM device state machines. The SDIR provides a method for software to execute the SDRAM initialization sequence (see Section 13.2.3.5, "SDRAM Initialization" on page 13-19).

Table 13-17. SDRAM Initialization Register - SDIR



80960 Core Local Bus Address
1500H

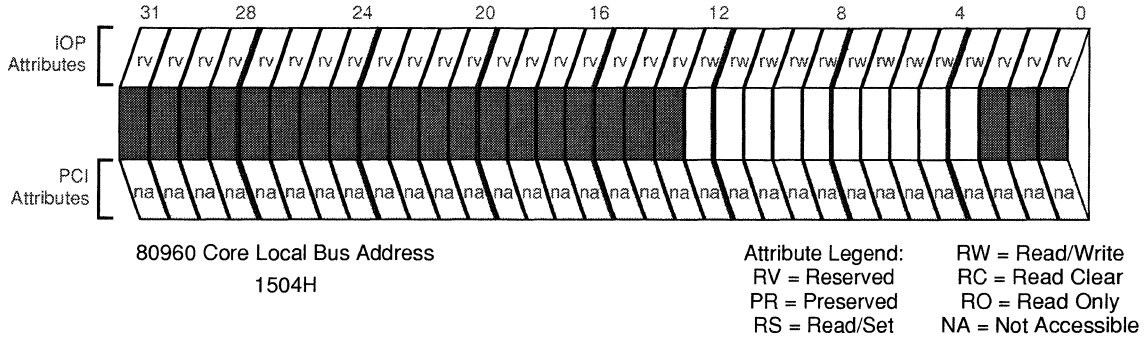
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:03	0	Reserved
02:00	111_2	**Special SDRAM Command**: These bits are used for SDRAM initialization. See Section 13.2.3.5, "SDRAM Initialization" on page 13-19 for details. While not in the initialization sequence, these bits should be set to $11x_2$. For details on the exact SDRAM commands, refer to Table 13-12, "SDRAM Commands" on page 13-18. • 000_2 - **Mode-Register-Set Command** where CAS# Latency = 2. • 001_2 - **Mode-Register-Set Command** where CAS# Latency = 3. • 010_2 - **Precharge-All Command**: The MCU issues one **precharge-all** command to the SDRAM devices. • 011_2 - **NOP Command**: The MCU issues one **NOP** command to the SDRAM devices. • 100_2 - **Auto-Refresh Command**: The MCU issues one **auto-refresh** command to the SDRAM devices. • $11x_2$ - **Normal SDRAM Operation**

13.6.2 SDRAM Control Register - SDCR

The SDRAM Control Register (SDCR) is responsible for programming the operation of the SDRAM state machines. The SDCR specifies the drive strength for the MCU pins, the bus width, and power failure handling. Refer to Table 13-19 for the recommended output buffer drive programmability.

Table 13-18. SDRAM Control Register - SDCR



Bit	Default	Description
31:13	0	Reserved
12:11	00_2	**Address and Control Drive Strength:** Controls the strength of the **SA[12:0]**, **SBA[1:0]**, **SRAS#**, **SCAS#**, **SWE# SDRAM** output buffers. Register can be programmed with four distinct drive strengths ranging from low to high: (low drive strength) "00" => "01" => "10" => "11" (high drive strength)
10:09	00_2	**Data Mask Drive Strength:** Controls the strength of the **SDQM[7:0] SDRAM** output buffers. Register can be programmed with four distinct drive strengths ranging from low to high: (low drive strength) "00" => "01" => "10" => "11" (high drive strength)
08:07	00_2	**Chip Enable 1 Drive Strength:** Controls the strength of the **SCE[1]# and SCKE[1] SDRAM** output buffers. Register can be programmed with four distinct drive strengths ranging from low to high: (low drive strength) "00" => "01" => "10" => "11" (high drive strength)
06:05	00_2	**Chip Enable 0 Drive Strength:** Controls the strength of the **SCE[0]# and SCKE[0] SDRAM** output buffers. Register can be programmed with four distinct drive strengths ranging from low to high: (low drive strength) "00" => "01" => "10" => "11" (high drive strength)
04:03	00_2	**Data Bus Drive Strength:** Controls the strength of the **DQ[63:0] and SCB[7:0] SDRAM** output Register can be programmed with four distinct drive strengths ranging from low to high: (low drive strength) "00" => "01" => "10" => "11" (high drive strength)
02:00	000_2	Reserved

Table 13-19. Drive Strength Programmability Options

Form Factor	Bank 0	Bank 1	SDCR[4:3] (SDQ)	SDCR[6:5] (SCE0#, SCKE0)	SDCR[8:7] (SCE1#, SCKE1)	SDCR[10:9] (SDQM)	SDCR[12:11] (SA[12:0], Controls)
1 single-sided DIMM	9[x8]	None	00	01	10	10	10
2 single-sided DIMMs	9[x8]	9[x8]	01	10	10	10	10
1 double-sided DIMM	9[x8]	9[x8]	01	10	10	10	10
1 single-sided DIMM	5[x16]	None	00	01	01	01	01
2 single-sided DIMMs	5[x16]	5[x16]	01	10	01	01	10
1 double-sided DIMM	5[x16]	5[x16]	01	10	01	01	10

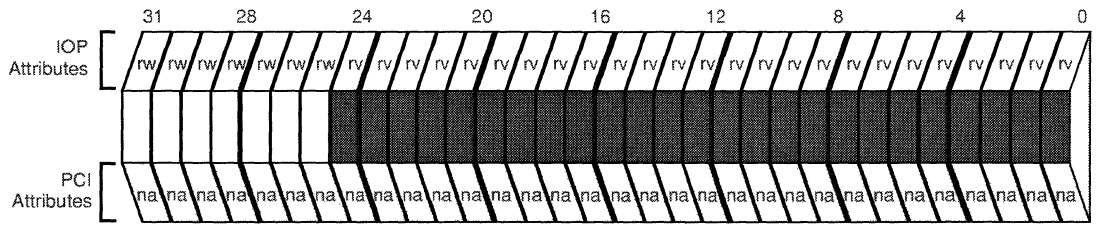
1. The values in this column represent the number of SDRAM devices in the bank of a certain SDRAM device width. For instance, 5x16 means that the bank contains 5 SDRAM devices with a 16-bit data bus width.
2. In the 5x16 configuration, there are 8 unused bits of one of the SDRAM devices used to create the 72-bit wide bus.

13.6.3 SDRAM Base Register - SDBR

This register indicates the beginning of SDRAM space. See Section 13.2.3.1, "SDRAM Sizes and Configurations" on page 13-13 for usage details. There can be two contiguous physical banks defined by SBR0 and SBR1 in the SDRAM subsystem starting at this address.

Note: SDRAM space must ***never*** cross a 512Mbyte boundary.

Table 13-20. SDRAM Base Register - SDBR

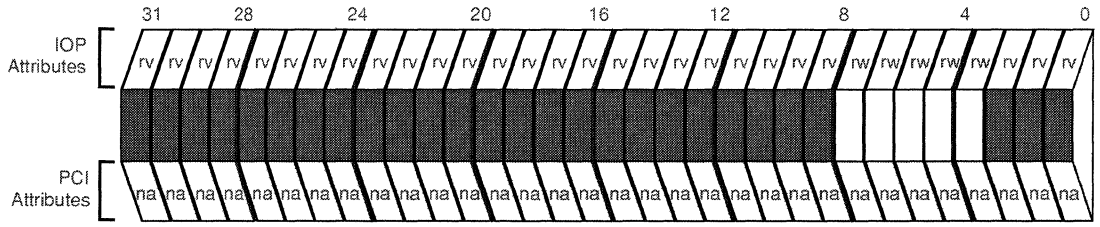


Bit	Default	Description
31:25	0	**SDRAM Base Address:** These bits define the upper seven bits of the SDRAM base address.
24:00	0	Reserved

13.6.4 SDRAM Boundary Register 0 - SBR0

This register indicates the upper boundary of SDRAM bank 0. If bank 0 is unpopulated, SBR0[7:3] is programmed with same value as SBR1[7:3]. See Section 13.2.3.1, "SDRAM Sizes and Configurations" on page 13-13 for more details and programming examples.

Table 13-21. SDRAM Boundary Register 0 - SBR0



80960 Core Local Bus Address
150CH

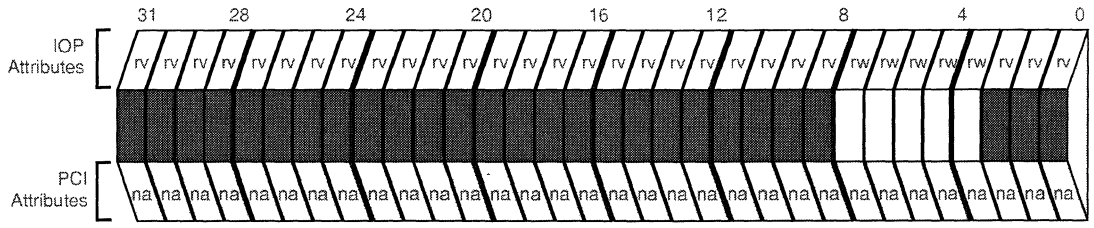
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:08	000000H	Reserved
07:03	000000_2	**SDRAM Boundary:** Defines the upper limit of SDRAM bank 0.
02:00	0H	Reserved

13.6.5 SDRAM Boundary Registers 1 - SBR1

This register indicates the upper boundary of SDRAM bank 1. If bank 1 is unpopulated, SBR1[7:3] is programmed with same value as SBR0[7:3]. If bank 1 is populated, SBR1[7:3] must be programmed greater than or equal to SBR0[7:3]. See Section 13.2.3.1, "SDRAM Sizes and Configurations" on page 13-13 for more details and programming examples.

Table 13-22. SDRAM Boundary Registers - SBR1



80960 Core Local Bus Address
1510H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:08	000000H	Reserved
07:03	000000_2	**SDRAM Boundary:** Defines the upper limit of SDRAM bank 1.
02:00	0H	Reserved

13.6.6 ECC Control Register - ECCR

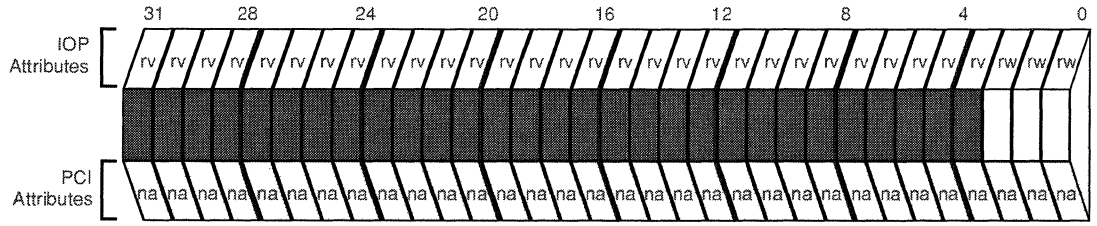
This register programs the MCU error correction and detection capabilities. The configuration depends on the application's needs but a typical configuration is:

- Enable multi-bit error reporting
- Enable single-bit error reporting
- Enable single-bit error correcting

While there are separate bits for single-bit and multi-bit error reporting, if either is enabled, both are enabled.

For more details, see Section 13.2.4, "Error Correction and Detection" on page 13-28 and Section 13.4, "Interrupts/Error Conditions" on page 13-42.

Table 13-23. ECC Control Register - ECCR



80960 Core Local Bus Address
1534H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:03	000 0000H	Reserved
02	0_2	**Single Bit Error Correction Enable:** Enables or disables the correction of a single bit error. 0 = Disable single bit error correction 1 = Enable single bit error correction
01	0_2	**Multi-Bit Error Reporting Enable:** Enables or disables the reporting of a multi-bit error condition. 0 = Disable multi-bit error reporting 1 = Enable multi-bit error reporting
00	0_2	**Single Bit Error Reporting Enable:** Enables or disables the reporting of a single bit error condition. 0 = Disable single bit error reporting 1 = Enable single bit error reporting

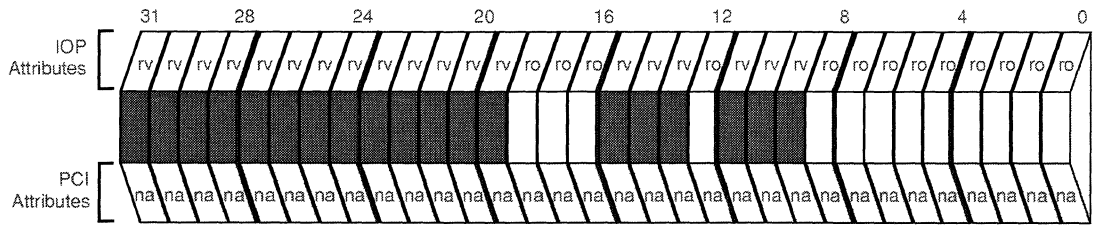
13.6.7 ECC Log Registers - ELOG0, ELOG1

The ECC Log Registers are responsible for logging the error types detected on the local memory bus. Two errors can be detected and logged. The error type is logged (single-bit or multi-bit) along with the syndrome that indicated the error. For a single-bit error, software can read this syndrome and determine which bit had the error in order to perform scrubbing. For a multi-bit error, software can read the syndrome and determine if the error is a nibble error (see Table 13-13, "Syndrome Decoding" on page 13-30).

The error recorded in ELOG0 corresponds to the address in ECAR0. ELOG1 corresponds to ECAR1.

The ELOGx registers comprise read-only bits and only have meaning if MCISR[0] or MCISR[1] is non-zero. For more details on error handling, see Section 13.2.4, "Error Correction and Detection" on page 13-28.

Table 13-24. ECC Log Registers - ELOG0, ELOG1



Error #	80960 Core Local Bus Address
0	1538H
1	153CH

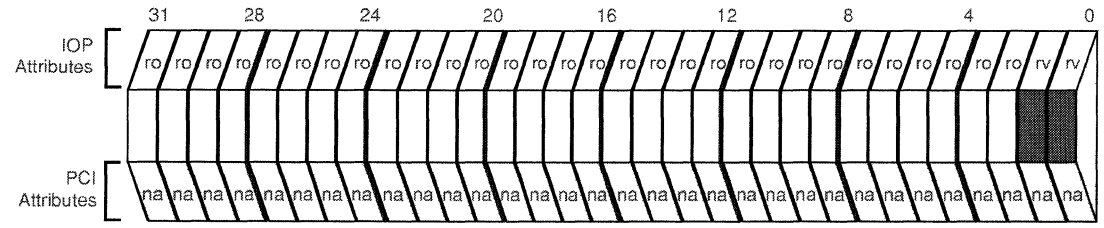
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:19	0	Reserved
18:16	000_2	**ECC Error Master: Indicates the master of the logged error.** • 000 - Primary ATU / Expansion ROM / Messaging Unit • 001 - DMA Channel 0 • 010 - DMA Channel 1 • 011 - Secondary ATU • 100 - DMA Channel 2 • 101 - Core/Bus Interface Unit • 110 - Application Accelerator • 111 - *Reserved*
15:13	000_2	Reserved
12	0_2	**Read or Write**: Indicates if the error occurred during a read or write transaction. 0 = Read error 1 = Write Error
11:09	000_2	Reserved
08	0_2	**ECC Error Type:** Indicates the type of error that occurred at this address. 0 = Single Bit Error 1 = Multi-Bit Error
07:00	00H	**Syndrome:** Holds the syndrome value that indicated the error.

13.6.8 ECC Address Registers - ECAR0, ECAR1

These registers are responsible for logging the addresses where the errors were detected on the local memory bus. Two errors can be detected and logged. The software knows exactly which SDRAM device had the error by reading these registers and decoding the syndrome in the log registers. For error details, see Section 13.2.4, "Error Correction and Detection" on page 13-28).

Table 13-25. ECC Address Registers - ECAR0, ECAR1



Error #	80960 Core Local Bus Address
0	1540H
1	1544H

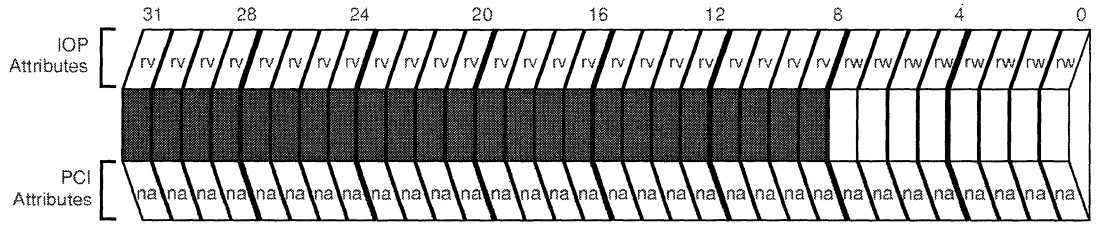
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:02	0	Error Address: Stores the upper 30 bits of the address that resulted in a single bit or multi-bit error.
01:00	00_2	Reserved

13.6.9 ECC Test Register - ECTST

This register allows testing between the ECC logic and the memory subsystem (Section 13.2.4.5, "ECC Testing" on page 13-34). To test error handling software, the programmer writes this register with a non-zero masking function. Any subsequent writes to memory stores a masked version of the computed ECC. Therefore, any subsequent reads to these locations result in an ECC error.

Table 13-26. ECC Test Register - ECTST



80960 Core Local Bus Address
1548H

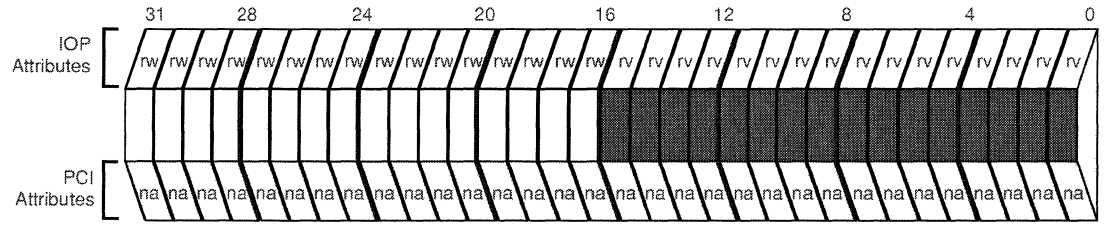
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:08	00 0000H	Reserved
07:00	00H	ECC Mask: 8-bit ECC mask. Each bit of the generated ECC is XORed with the appropriate bit in this mask field before the ECC is stored into memory. See Section 13.2.4.5, "ECC Testing" on page 13-34.

13.6.10 Flash Base Register 0 - FEBR0

This register indicates the beginning of the first Flash memory bank. The starting location must be boundary equal to the granularity of the Flash device. The upper 16 bits are used for a 64 Kbyte bank, 15 for a 128 Kbyte bank, etc. There can be two non-contiguous physical banks in the Flash subsystem starting with this address. For more details, see Section 13.2.2.1, "Flash Memory Addressing" on page 13-6.

Table 13-27. Flash Base Register 0 - FEBR0



80960 Core Local Bus Address
154CH

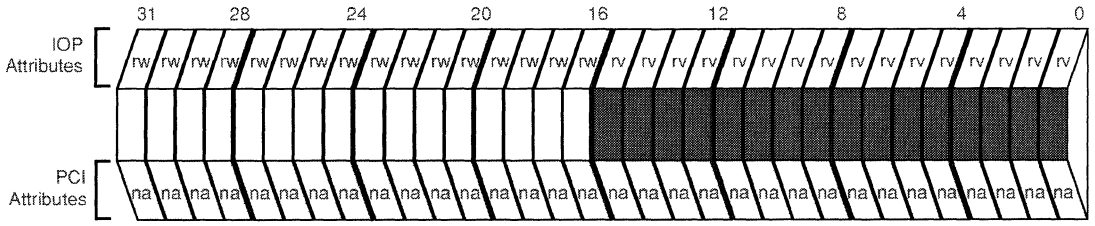
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:16	FE80H	**Flash Base Address:** These bits define the upper 16 bits of the Flash base address.
15:00	0000H	Reserved

13.6.11 Flash Base Register 1 - FEBR1

This register indicates the beginning of the second Flash memory bank. The starting location must be boundary equal to the granularity of the Flash device. The upper 16 bits are used for a 64 Kbyte bank, 15 for a 128 Kbyte bank, etc. There can be two non-contiguous physical banks in the Flash subsystem. For more details, see Section 13.2.2.1, "Flash Memory Addressing" on page 13-6.

Table 13-28. Flash Base Register 1 - FEBR1

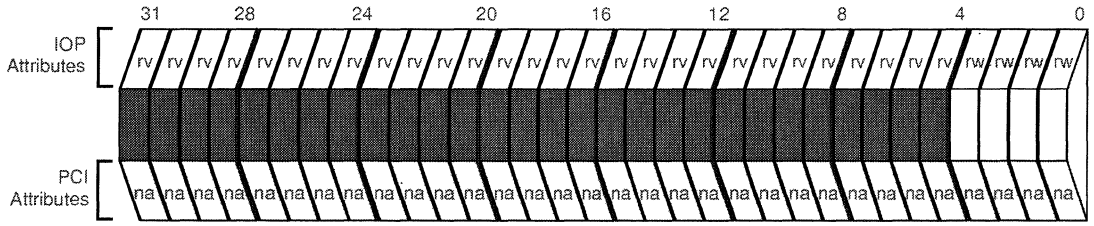


Bit	Default	Description
31:16	0000H	**Flash Base Address:** These bits define the upper 16 bits of the Flash base address.
15:00	0000H	Reserved

13.6.12 Flash Bank Size Register 0 - FBSR0

This register indicates the size of Flash bank 0. The two Flash banks do not have to be equal in size. If the bank is unpopulated, a value of zero is programmed. See Section 13.2.2.1, "Flash Memory Addressing" on page 13-6 for more details.

Table 13-29. Flash Bank Size Register 0 - FBSR0



80960 Core Local Bus Address
1554H

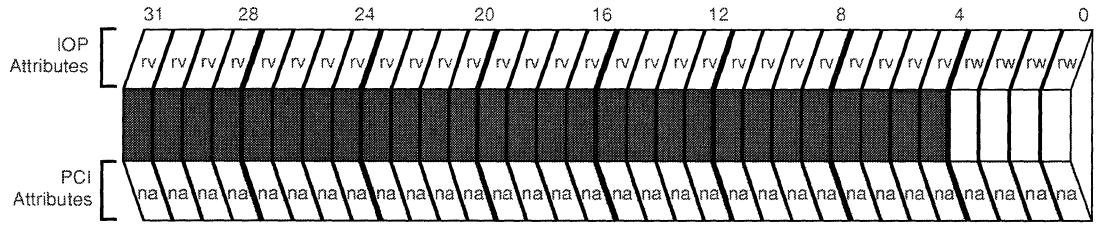
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:04	0	Reserved
03:00	1000_2	**Flash Bank Size:** Defines the size for the Flash bank. • 0000 - Bank disabled • 0001 - 64 Kbytes • 0010 - 128 Kbytes • 0011 - 256 Kbytes • 0100 - 512 Kbytes • 0101 - 1 Mbytes • 0110 - 2 Mbytes • 0111 - 4 Mbytes • 1XXX - 8 Mbytes

13.6.13 Flash Bank Size Register 1 - FBSR1

These registers indicate the size of Flash bank 1. The two Flash banks do not have to be equal in size. If the bank is unpopulated, a value of zero is programmed. See Section 13.2.2.1, "Flash Memory Addressing" on page 13-6 for more details.

Table 13-30. Flash Bank Size Register 1 - FBSR1



80960 Core Local Bus Address
1558H

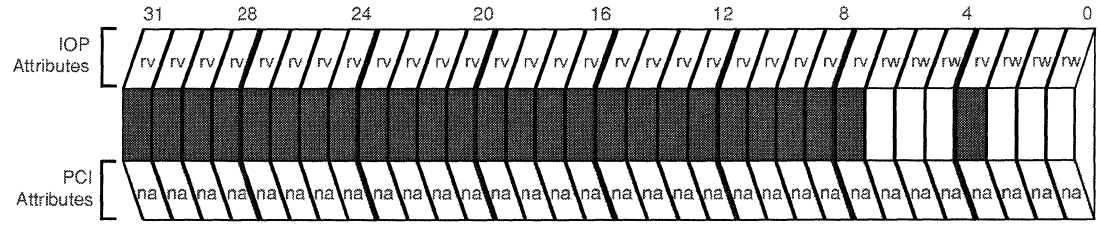
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:04	0	Reserved
03:00	0000_2	**Flash Bank Size:** Defines the size for the Flash bank. • 0000 - Bank disabled • 0001 - 64 Kbytes • 0010 - 128 Kbytes • 0011 - 256 Kbytes • 0100 - 512 Kbytes • 0101 - 1 Mbytes • 0110 - 2 Mbytes • 0111 - 4 Mbytes • 1XXX - 8 Mbytes

13.6.14 Flash Wait States Registers - FWSR0, FWSR1

These registers indicate the wait state and recovery cycle profile of each physical Flash bank. Programmability for up to 20 address-to-data wait states is included to accommodate UART devices. For more details, see Section 13.2.2.2, "Flash Read Cycle" on page 13-7 and Section 13.2.2.3, "Flash Write Cycle" on page 13-10.

Table 13-31. Flash Wait State Registers - FWSR0, FWSR1



Bank #	80960 Core Local Bus Address
0	155CH
1	1560H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:07	0	Reserved
06:04	111_2	**Recovery Cycle Wait States:** Defines the number of recovery cycle wait states for the Flash bank. • 000 - 1 Recovery wait state • 001 - 4 Recovery wait states • 010 - 8 Recovery wait states • 011 - 12 Recovery wait states • 100 - 16 Recovery wait states • Others (Default) - 20 Recovery wait states
03	0_2	Reserved
02:00	111_2	**Address-to-Data Wait States:** Defines the number of address-to-data wait states for the Flash bank during a read or write transaction. • 000 - 4 Address-to-Data wait states • 001 - 8 Address-to-Data wait states • 010 - 12 Address-to-Data wait states • 011 - 16 Address-to-Data wait states • Others (Default) - 20 Address-to-Data wait states

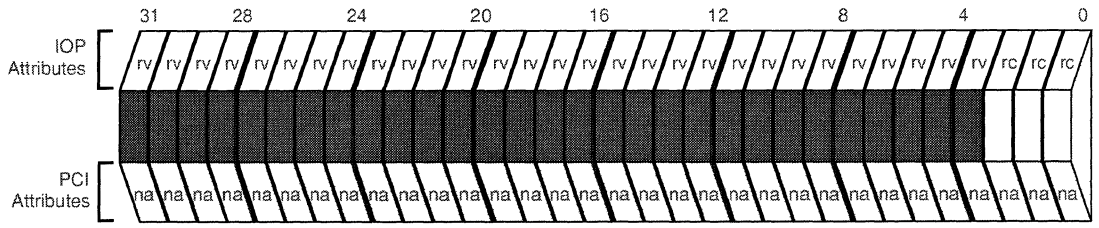
13.6.15 Memory Controller Interrupt Status Register - MCISR

Setting the MCISR asserts an NMI to the core. Upon an interrupt, the JX core polls the interrupt status register for each unit. The interrupt status register tells the core the reason for the interrupt. The MCU has three interrupt conditions: first ECC error (MCISR[0]), second ECC error (MCISR[1]), and more than two ECC errors (MCISR[2]).

If the MCU detects an ECC error and both MCISR[0] and MCISR[1] are cleared, the error is logged in ELOG0 and MCISR[0] is set to 1. If one of the MCISR bits are not clear and the MCU detects an error, the error is logged in the unused ELOGx register and the appropriate MCISR bit is set to 1. If both MCISR[0] and MCISR[1] are not clear, any additional ECC errors are not logged and MCISR[2] is set.

Bits 2:0 are read/clear bits which means that to clear them, software must write a one to these bits.

Table 13-32. Memory Controller Interrupt Status Register - MCISR



80960 Core Local Bus Address
1564H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

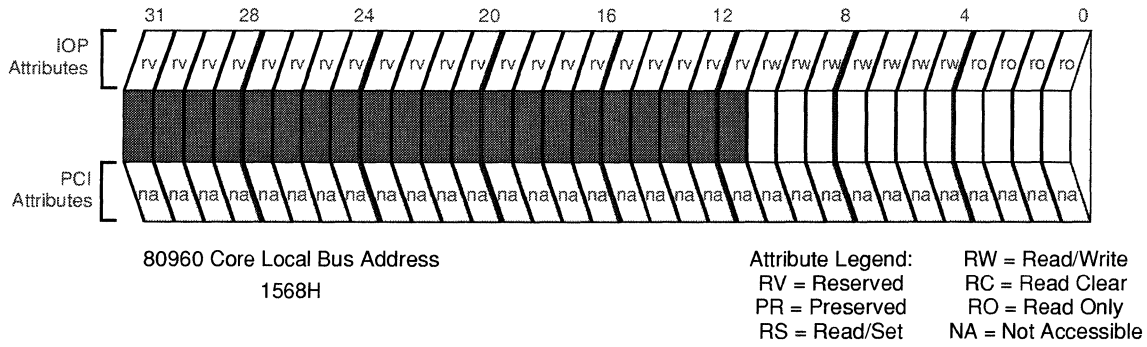
Bit	Default	Description
31:03	0	Reserved
02	0_2	**ECC Error N:** Indicates that the MCU detected an ECC error while MCISR[1] and MCISR[0] are both set. 0 = No error detected 1 = Error detected
01	0_2	**ECC Error 1:** Indicates that the MCU detected an ECC error and recorded the error in ELOG1. 0 = No error detected 1 = Error detected and recorded in ELOG1
00	0_2	**ECC Error 0:** Indicates that the MCU detected an ECC error and recorded the error in ELOG0. 0 = No error detected 1 = Error detected and recorded in ELOG0

13.6.16 Refresh Frequency Register - RFR

The Refresh Frequency Register is programmed for refreshing the SDRAM subsystem at the specified interval. Writing to the RFR programs the refresh counter with the number of clocks between refresh cycles. Reading from the RFR results in the value currently within the refresh counter.

For 100MHz operation, the RFR should be programmed with a value of 600H. For frequencies below 100MHz, the RFR should be programmed with 400H.

Table 13-33. Refresh Frequency Register - RFR



Bit	Default	Description
31:11	0	Reserved
10:00	000H	**Refresh Interval:** Programs the number of clocks that triggers a refresh cycle to the SDRAM interface. If all zeroes, refresh cycles are disabled. See Section 13.2.1.6, "Refresh Counter" on page 13-5.

PCI-to-PCI Bridge Unit 14

This chapter describes the PCI-to-PCI Bridge including functionality, modes of operation, configuration, and integration into the Intel® 80303 I/O processor system architecture.

14.1 Overview

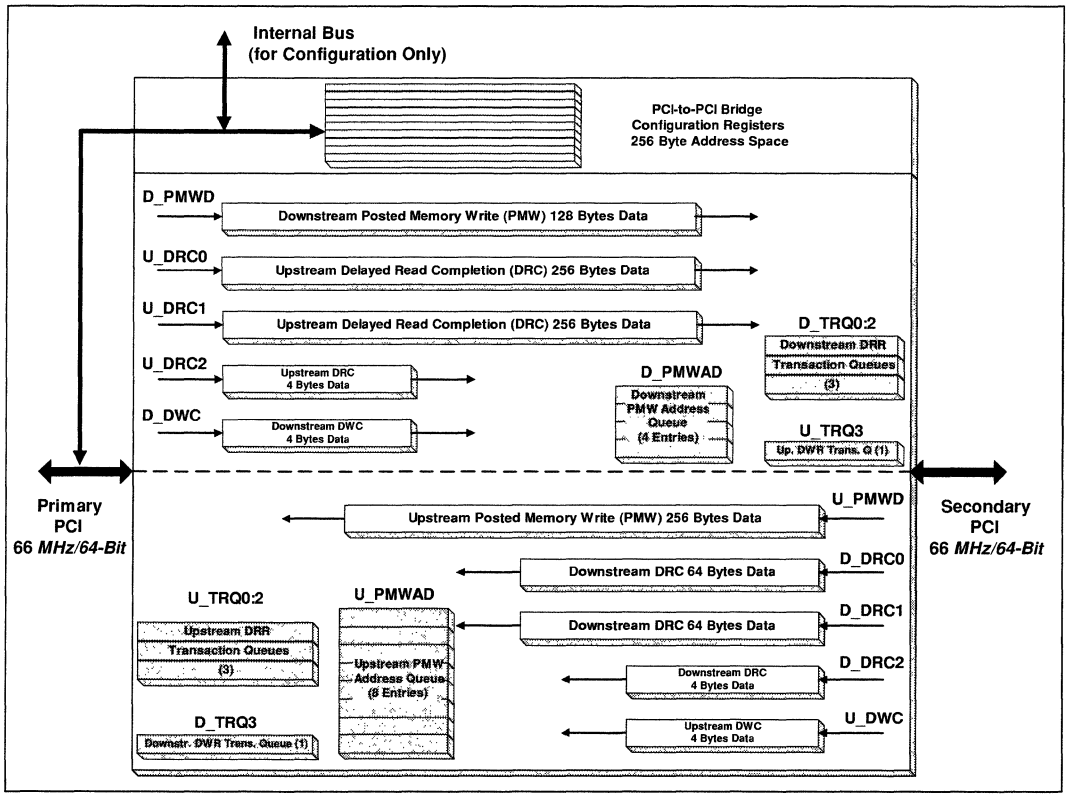
The PCI-to-PCI bridge unit extends a PCI Bus beyond its physical constraint of ten electrical PCI loads at 33 MHz or five electrical PCI loads at 66 MHz. The bridge unit uses the concept of hierarchical buses; each hierarchybus is a separate electrical entity, but all hierarchy buses are logically one bus. The PCI-to-PCI bridge unit does not increase the PCI bus bandwidth, it only allows that bus to be extended for applications requiring more I/O components than PCI electrical specifications allow.

PCI-to-PCI bridge unit features include:

- Full compliance to the *PCI Local Bus Specification*, Revision 2.2.
- Full compliance to the *PCI-to-PCI Bridge Architecture Specification*, Revision 1.1.
- Full Compliance to the *PCI Bus Power Management Interface Specification*, Revision 1.1.
  - — Defines the PCI hardware support required by the *Advanced Configuration and Power Interface Specification*, Revision 1.0 initiative.
- Support for 66 MHz PCI operation.
- 528 MBytes/sec. PCI bandwidth for Primary and Secondary buses through 64-bit/66 MHz operation.
- Synchronous operation between Primary and Secondary PCI busses.
- Provides six Secondary PCI output clocks (**S_CLKOUT[5:0]**).
- Support for 32-bit PCI masters and targets on both busses.
  - — Additional support for independent 32-bit only bus configurations on Primary and Secondary busses.
- Independent Primary and Secondary PCI buses allowing concurrent operations in either direction.
- Multiple *Memory Write* and *Memory Write and Invalidate* operations posted within the upstream and downstream bridge queues concurrently.
  - — Up to four PMW transactions with a total of 128 Bytes of write data on downstream transactions.
  - — Up to eight PMW transactions with a total of 256 Bytes of write data on upstream transactions.
- Support for up to three delayed read cycles initiated from the Primary bus and three delayed read cycles initiated from the Secondary bus.
  - — 516 bytes dedicated for delayed read completion data for upstream reads.
  - — 132 bytes dedicated for delayed read completion data for downstream reads.
- Separate memory and I/O address spaces on the Secondary side of the bridge.
- VGA Compatible Addressing Mode.
- 64-bit addressing mode (Dual Address Command) for upstream cycles initiated from Secondary PCI interface.
- Private device configuration and address space for private PCI devices on Secondary PCI bus.

Figure 14-1 shows a block diagram of the 80303 I/O processor PCI-to-PCI Bridge unit.

Figure 14-1. PCI-to-PCI Bridge Unit Functional Block Diagram



14.2 Theory of Operation

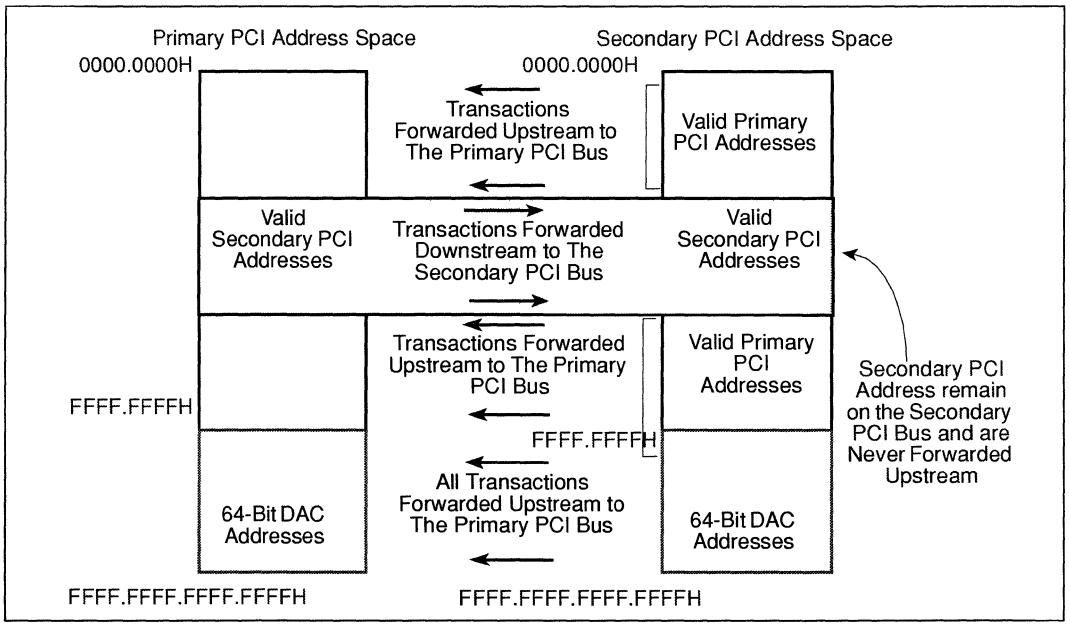
The bridge unit operates as an address filter unit between the Primary and the Secondary PCI buses. PCI supports three separate address spaces:

- 32-Bit address space with Single Address Cycle (SAC)
- 64-Bit address space with Dual Address Cycle (DAC)
- 64 Kbyte I/O address space (with 16-bit addressing)
- Separate configuration space

A PCI-to-PCI bridge is programmed with a contiguous range of addresses within the memory and I/O address spaces, which then become the Secondary PCI address space. Any address present on the Primary side of the bridge which falls within the programmed Secondary space is forwarded from the Primary to the Secondary side while addresses outside the Secondary space are ignored by the Primary interface. The Secondary side of the bridge works in reverse of the Primary side, ignoring any addresses within the programmed Secondary address space and forwarding any addresses outside the Secondary space to the Primary side. See Figure 14-2.

The Primary and Secondary interfaces of the PCI bridge each implement PCI *PCI-to-PCI Bridge Architecture Specification*, Revision 1.1 compliant master and target devices. A PCI transaction initiated on one side of the bridge will address the initiating bus bridge interface as a target and the transaction will be completed by the target bus interface operating as a master device. The bridge is software transparent to PCI devices on either side.

Figure 14-2. Bridge Operation



The PCI-to-PCI bridge unit of the 80303 I/O processor adheres, at a minimum, to the required features found in the *PCI-to-PCI Bridge Architecture Specification*, Revision 1.1 and the *PCI Local Bus Specification*, Revision 2.2. This chapter will describe bridge functionality and will refer to the *PCI-to-PCI Bridge Architecture Specification*, Revision 1.1 and the *PCI Local Bus Specification*, Revision 2.2 where appropriate.

14.3 Architectural Description

The PCI-to-PCI bridge unit can be logically separated into four major components. They are:

- Primary PCI Interface
- Secondary PCI Interface
- Upstream/Downstream Queues
- Configuration Registers

14.3.1 Primary PCI Interface

The Primary PCI interface of the PCI-to-PCI bridge unit can act either as a target or an initiator of a PCI bus transaction. For most systems, the Primary interface will be connected to the PCI side of a Host/PCI bridge which is typically the lowest numbered PCI bus in a system hierarchy. The Primary interface consists of the mandatory 50 signal pins defined within the *PCI-to-PCI Bridge Architecture Specification*, Revision 1.1, four optional interrupt pins, one 66 MHz enable pin, and the 39 pins required by the PCI 64-bit extension. Refer to the *PCI Local Bus Specification*, Revision 2.2 for a complete description of individual pin functionality.

The Primary PCI interface implements both an initiator (master) and a target (slave) PCI device. When a PCI transaction is initiated on the Secondary bus, the Primary master state machine completes the transaction (write or read) as if it was the initiating device. The Primary PCI interface, as a PCI target for transactions that need to complete on the Secondary bus, accepts the transaction and forwards the request to the Secondary side. As a target, the Primary PCI interface uses positive decoding to claim the PCI transaction addressed below the bridge and then forward the transaction onto the Secondary master interface.

The Primary PCI interface is responsible for all PCI command interpretation, address decoding and error handling for transactions initiated on the PCI-to-PCI bridge Primary bus.

The Primary interface of the 80303 I/O processor supports enhanced PCI bandwidth of 528 MBytes/sec. through the use of the 64-bit PCI extension at a frequency of up to 66 MHz.

The additional bandwidth that the 80303 I/O processor Primary PCI interface provides is used to support additional I/O bandwidth from the Secondary PCI bus as well as providing a faster/wider pipe to the host processor memory bus.

14.3.2 Secondary PCI Interface

The Secondary PCI interface of the PCI-to-PCI bridge unit functions in almost the same manner as the Primary interface. It consists of both a PCI master and a PCI slave device and implements the "second" PCI bus with a new set of PCI electrical loads for use by the system. The Secondary PCI interface consists of the mandatory 49 pins defined in the *PCI-to-PCI Bridge Architecture Specification*, Revision 1.1, one 66 MHz enable pin, and the 39 pins required for the 64-bit extension. Four additional PCI interrupt pins are provided for use by Secondary PCI devices.

As a slave (target), the Secondary PCI interface is responsible for claiming PCI transactions that do not fit within the bridge Secondary memory or I/O address space and forwarding them through the bridge to the addressed target on the Primary side. As a master (initiator), the Secondary PCI

interface is responsible for completing transactions initiated on the Primary side of the bridge. The Secondary PCI interface uses inverse decoding of the bridge address registers and only forwards addresses within the Primary address space across the bridge.

The Secondary PCI interface also implements a separate address space for private PCI devices on the Secondary bus where it ignores and does not forward a range of Primary addresses defined at configuration time by the i960 core processor. Support for private PCI devices is discussed in Section 14.4.5 and Section 14.5.5.

The Secondary PCI interface supports the use of PCI dual address cycles (DAC) for memory transactions targeted at the Primary bus and main system memory. The Secondary interface will claim *all* DAC memory cycles present on the Secondary bus with subtractive (default) or medium decode timing decode timing.

14.3.3 Upstream/Downstream Queues

The 80303 I/O processor implements an extensive queueing architecture to improve the PCI bandwidth for all write transactions and to reduce the latency of read transactions from both sides of the PCI-to-PCI bridge unit. As a *PCI Local Bus Specification*, Revision 2.2 compliant device, the bridge unit supports both posted and delayed transactions.

In a Delayed transaction, the information required to complete the transaction is latched and the transaction is terminated with a Retry. The bridge then performs the transaction on behalf of the initiator. The initiator is required to repeat the original transaction that was terminated with a Retry in order to complete the transaction.

In a Posted transaction, the transaction is allowed to complete on the initiating bus before completing on the target bus.

Delayed and Posted transactions are discussed in detail in Section 14.6.

The bridge has an asymmetric queue architecture supporting the data flow requirements of intelligent I/O applications. For downstream transactions (initiated on the Primary PCI bus interface) the PCI-to-PCI bridge unit supports the following number and types of a queues:

- Up to four transactions with 128 bytes of data for posted memory write transactions
  - — FIFO implementation supporting variable length write transactions within the same queue. Any combination of burst sizes from one to four transactions.
  - — Supports *Memory Write* and *Memory Write and Invalidate* transactions
- 132 bytes of delayed read completion (DRC) data queue with three separate Transaction Address Queues
  - — Two 64-byte DRC queues
  - — One 4-Byte DRC queue
  - — Transaction Queue holds delayed read addresses during PCI delayed transactions
  - — Supports *Memory Read, Memory Read Line, Memory Read Multiple, Configuration Read* and *I/O Read* transactions
- Separate 4-byte queue for I/O and Configuration Write Cycles
  - — Performed as Delayed Write Cycles

For upstream transactions (initiated on the Secondary PCI interface), the bridge supports a larger set of queues to accommodate high PCI bandwidth targeted at the Primary PCI bus:

- Up to eight transactions with 256 bytes of data for posted memory write transactions.
  - — FIFO implementation supporting variable length write transactions within the same queue. Any combination of burst sizes from one to 8 transactions.
  - — Supports *Memory Write* and *Memory Write and Invalidate* transactions.
- 516 bytes of delayed read completion (DRC) data queue with three separate Transaction Address Queues.
  - — Two 256-byte DRC queues.
  - — One 4-byte DRC queue.
  - — Transaction Queue holds delayed read addresses during PCI delayed transactions.
  - — Supports *Memory Read, Memory Read Line, Memory Read Multiple,* and *I/O Read* transactions.
- Separate 4-byte queue for Delayed Write Cycles.
  - — I/O Writes and Configuration Writes.

The asymmetric, multi-transaction queue architecture enforces all *PCI Local Bus Specification*, Revision 2.2 ordering rules. Priority mechanisms with additional prefetch rules assign larger read queues (if available) for Memory Read Line and Memory Read Multiple transactions. See Section 14.6.5 and Section 14.7.2 for additional details.

14.3.4 Configuration Registers

Every PCI device implements a separate configuration address space and configuration registers. The *PCI Local Bus Specification*, Revision 2.2 requires that the configuration space be 256 bytes long with the first 64 bytes adhering to a predefined header format. The PCI-to-PCI Bridge in the 80303 I/O processor contains the predefined 64 byte header registers plus additional configuration registers for device dependent operation (see Section 14.15).

The first 16 bytes of the bridge configuration header format implement the common configuration registers required by all PCI devices. The value in the read-only Header Type Register defines the format for the remaining 48 bytes within the header and returns a 81H for a PCI-to-PCI bridge that integrates other functions.

The bridge has a seven byte extended configuration space residing at configuration offset 68H that supports the *Advanced Configuration and Power Interface Specification,* Revision 1.0 by providing the Power Management registers defined by the *PCI Bus Power Management Interface Specification*, Revision 1.1.

Devices on the Primary bus can only access the PCI-to-PCI bridge configuration space with Type 0 configuration commands. Devices on the Secondary PCI bus can *not* access bridge configuration space with PCI configuration cycles. The configuration registers hold all the necessary address decode, error condition and status information for both sides of the bridge.

14.4 Configuration Accesses

This section describes how the bridge handles PCI configuration read and write commands.

There are two classes of targets for PCI configuration commands:

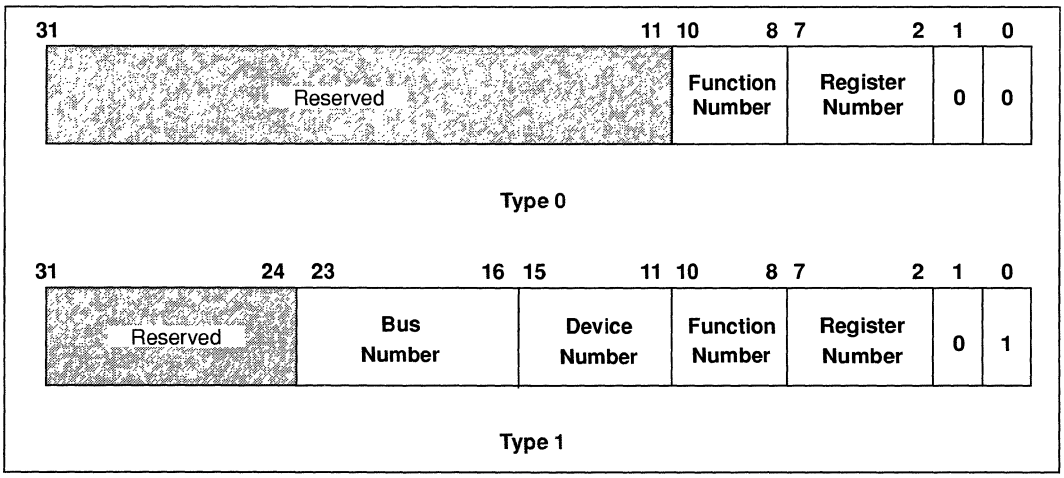
- devices that reside on the Primary PCI bus
- devices that reside on hierarchical (Secondary) PCI buses that are accessed via PCI-to-PCI bridge chips

The encoding of the address during a configuration command distinguishes the target of the command. Figure 14-3 and Table 14-1 show the different address encodings associated with each PCI configuration command type. Type 0 and Type 1 commands are distinguished by address bits **AD[1:0]**.

Table 14-1. PCI Configuration Command Access Formats

Bit Function	Type 0 Commands Bit Position (# of Bits)	Type 1 Commands Bit Position (# of Bits)
Command Type	1:0 (2)	1:0 (2)
Register Number	7:2 (6)	7:2 (6)
Function Number	10:8 (3)	10:8 (3)
Device Number	N/A	15:11 (5)
Bus Number	N/A	23:16 (8)
Reserved	31:11 (20)	31:24 (8)

Figure 14-3. PCI Configuration Access Formats



A Type 0 configuration command on the Primary interface may be accepted or ignored by the bridge depending on the value of the **P_IDSEL** input. A Type 1 configuration command on the Primary interface may be ignored, forwarded downstream unaltered, converted to a Type 0 command on the Secondary interface, or converted to a Special Cycle on the Secondary interface.

A Type 1 configuration write command on the Secondary interface of the bridge may be ignored, forwarded upstream under certain conditions, or converted to a Special Cycle on the Primary interface. The bridge cannot convert a Type 1 configuration command on the Secondary side to a Type 0 on the Primary side. The bridge will ignore all configuration reads and Type 0 configuration writes on the Secondary interface.

Configuration commands will only be accepted on the Primary interface if the Configuration Cycle Retry bit within the Extended Bridge Command Register (EBCR, see Section 14.15) is cleared. If the Configuration Cycle Retry bit is set, the Primary PCI interface will signal a Retry on all Type 1 and Type 0 configuration commands.

All configuration commands are 32-bit only and therefore will not use the 64-bit extensions of both the Primary and Secondary PCI bus interfaces. See Section 14.6.3 for complete details of 64-bit operation. In addition, the 80303 I/O processor does not support bursting during Type 0 or Type 1 configuration cycles. Type 0 and Type 1 configuration writes will be disconnected after the first 32-bit data phase. Type 1 configuration reads (handled as delayed transactions) will only read up at a maximum 1 Dword (actual data read depends on the byte enables during the data phase).

Table 14-2. Bridge Configuration Cycle Handling Summary

Primary Interface	Secondary Interface
NOTE: Type 0 - Bridge Accepts	Type 0 - Bridge Ignores
Type 1 forwarded to Type 1 on Secondary Side if: Bus number between SBNR and SubBNR (including SubBNR)	Type 1 forwarded to Type 1 on Primary Side if: Command = Config Write and Bus number does not equal PBNR and Bus Number is outside SBNR and SubBNR and Address = 0xXXXXFF01H
Type 1 converted to Type 0 on Secondary Side if: Command = Config Write and Bus number = SBNR and Address not equal to 0xXXXXFF01H or Command = Config Read and Bus number = SBNR	Type 1 **never** converted to Type 0 on Primary Side
Type 1 converted to a Special Cycle on Secondary Side if: Command = Config Write and Bus number is equal to SBNR and Address equals 0xXXXXFF01H	Type 1 converted to a Special Cycle on Primary Side if: Command = Config Write and Bus number = PBNR and Address = 0xXXXXFF01H

14.4.1 Type 0 Commands

If address bits **P_AD[1:0]** are 00_2, then the transaction present on the PCI bus is a Type 0 configuration read or write command. Type 0 configuration transactions configure PCI devices connected to the bus where the transaction originated. The PCI-to-PCI bridge responds to Type 0 commands on the Primary PCI interface only. Type 0 configuration commands present on the Secondary bus are ignored by the bridge.

The bridge is selected by a PCI configuration command and will claim it (by asserting **P_DEVSEL#**) if the **P_IDSEL** pin is asserted, the PCI command indicates a configuration read or write, and address bits **P_AD[1:0]** are 00_2 all during the address phase. The Primary interface will ignore any configuration command (**P_IDSEL** active) where **P_AD[1:0]** are not 00_2 (see Section 14.4.2 for the case of 01_2). During the configuration access address phase, the PCI address is divided into a number of fields to determine the actual configuration register access. These fields, in combination with the byte enables during the data phase create the unique encoding necessary to access the individual registers of the configuration address space:

- **P_AD[7:2]** - Register Number. Selects one of 64 DWORD registers in the bridge PCI configuration address space.
- **P_C/BE[3:0]#** - Used during the data phase. Selects which actual configuration register is used within the DWORD address. Creates byte addressability of the register space.
- **P_AD[10:8]** - Function Number. Used to select which function of a multi-function device is being accessed. The PCI-to-PCI bridge unit is function 0 and therefore it will only respond to 000_2 in this bit field and ignore all other bit combinations. (Refer to Section 15.2.4, "PCI Multi-Function Device Swapping/Disabling" on page 15-23 for exceptions to this statement.)

Address bits **P_AD[31:11]** are used to drive the bridge unit **P_IDSEL** input. Typically, the **IDSEL** input of each PCI device on a PCI bus is connected to a unique address bit in this range. This mapping requires that only one address bit from **P_AD[31:11]** be asserted during the address phase of a configuration access.

14.4.2 Type 1 Commands and Type 1 to Type 0 Conversions

If **P_AD[1:0]** are 01_2, a Type 1 configuration command is present. Type 1 commands can be forwarded by the bridge to any level in the PCI hierarchy (up to 255 levels). Eventually, a Type 1 command is converted to a Type 0 command by a PCI bridge to configure a device on its Secondary interface. Configuration registers in the bridge itself (PBNR, SBNR, and SubBNR) identify the bridge Primary bus number, Secondary bus number and a subordinate bus number (highest numbered PCI bus beneath the bridge). These parameters, along with the information embedded in the PCI Type 1 command determine whether a Type 1 transaction is ignored, forwarded, or converted to a Type 0 command. Type 1 commands are also used as a means for generating PCI Special Cycles on a hierarchical bus.

Address bits **P_AD[10:2]** in a Type 1 command have the same function as in a Type 0 command. **P_AD[15:11]** and **P_AD[23:16]** are used to determine a unique IDSEL encoding and to determine whether or not to convert the Type 1 command to a Type 0, forward it unmodified, or ignore it completely. The bit fields within a Type 1 PCI configuration command are as follows:

- **P_AD[7:2]** - Register Number. Selects one of 64 DWORD registers in the bridge PCI configuration address space.
- **P_C/BE[3:0]#** - Used during the data phase. Selects which actual configuration register is used within the DWORD address. Creates byte addressability of the register space.

- **P_AD[10:8]** - Function Number. Used to select which function of a multi-function device is being accessed.
- **P_AD[15:11]** - Device Number. Used during Type 1 to Type 0 conversion. Decoded by the bridge and used to select a unique address bit to drive an **IDSEL** input of a PCI device on the Secondary bus during the Type 0 transaction that occurs after a Type 1 to Type 0 conversion. The value in **P_AD[15:11]** is decoded and used to drive **S_AD[31:11]**. See Table 14-3.
- **P_AD[23:16]** - Bus Number. Used to identify the hierarchical bus number for which the configuration transaction is intended and where the Type 0 conversion needs to occur. The bridge uses this information in conjunction with the *Primary, Secondary, and Subordinate Bus Number* registers to make the decision to forward unaltered or to convert to a Type 0 on its Secondary interface. If the bus number bit field (bits 23:16 of Type 1 command) matches the value in the Secondary bus number register (Section 14.15.11), the transaction is converted to a Type 0 on the Secondary bus.

Table 14-3 shows the address mapping for driving **S_AD[31:11]** on the Secondary bus based on the encoding of the device number in **P_AD[15:11]** of a Type 1 transaction. Note that when **P_AD[15]** = 1_2 on the Primary interface, bits 31:11 are not asserted on the Secondary interface.

In addition, the Secondary IDSEL Select Register (SISR, see Section 14.15) can cause any of the Secondary address bits **S_AD[25:16]** to be zero regardless of the Primary address **P_AD[15:11]**. This register is needed for implementing private PCI devices on the Secondary PCI bus. Refer to Section 14.4.5 for details.

Table 14-3. IDSEL Mapping for Type 1 to Type 0 Conversions

Primary Address P_AD[15:11]	Secondary Address Bits S_AD[31:11]
00000	0000 0000 0000 0001 0000 0
00001	0000 0000 0000 0010 0000 0
00010	0000 0000 0000 0100 0000 0
00011	0000 0000 0000 1000 0000 0
00100	0000 0000 0001 0000 0000 0
00101	0000 0000 0010 0000 0000 0
00110	0000 0000 0100 0000 0000 0
00111	0000 0000 1000 0000 0000 0
01000	0000 0001 0000 0000 0000 0
01001	0000 0010 0000 0000 0000 0
01010	0000 0100 0000 0000 0000 0
01011	0000 1000 0000 0000 0000 0
01100	0001 0000 0000 0000 0000 0
01101	0010 0000 0000 0000 0000 0
01110	0100 0000 0000 0000 0000 0
01111	1000 0000 0000 0000 0000 0
10000 - 11111	0000 0000 0000 0000 0000 0

14.4.3 Type 1 to Type 1 Forwarding

A Type 1 write transaction on the Primary bus is converted to a Type 0 write transaction and forwarded to the Secondary interface provided the following condition is met:

- Bus number in the Type 1 command is equal to the Secondary Bus Number Register (SBNR, see Section 14.15).

A Type 1 write transaction on the Primary bus is forwarded unmodified to the Secondary interface provided the following condition is met:

- Bus number in the Type 1 command is greater than the SBNR but less than or equal to the Subordinate Bus Number Register (SubBNR).

In this instance, the Secondary interface will generate a Type 1 command address cycle with exactly the same address information that was contained within the Type 1 command on the Primary interface. The Type 1 command on the Secondary interface will be intercepted and decoded by a downstream bridge.

A Type 1 write transaction on the Secondary bus is forwarded unmodified to the Primary interface provided all of the following conditions are met:

- Device Number is all ones - **S_AD[15:11]** = 11111_2
- Function Number is all ones - **S_AD[10:8]** = 111_2
- Register Number is all zeros - **S_AD[7:2]** = 00000_2
- Bus Number does not match the Primary Bus Number of the bridge
- Bus Number is outside the range of bus numbers specified by the Secondary Bus Number (inclusive) and the Subordinate Bus Number (inclusive) of the bridge.

The bridge will generate a Type 1 on the Primary side with exactly the same information as on the Secondary side. This Type 1 command will be intercepted by an upstream bridge and converted to a Special Cycle transaction.

Note that Type 1 to Type 1 forwarding is for Configuration Write commands only. Type 1 Configuration Read commands are not forwarded upstream through the bridge.

14.4.4 Type 1 to Special Cycle Conversion

A Type 1 configuration write command on the Primary interface will be converted to a Special Cycle command on the Secondary interface provided all of the following conditions are met:

- Device Number is all ones - **P_AD[15:11]** = 11111_2
- Function Number is all ones - **P_AD[10:8]** = 111_2
- Register Number is all zeros - **P_AD[7:2]** = 00000_2
- Bus Number matches the Secondary Bus Number of the bridge

All PCI devices ignore the address during a Special Cycle and a Master-Abort occurs on the PCI bus. However, the Master Abort error status bit (bit 13 of the Secondary Status Register) is *not* set. The data for the Special Cycle on the Secondary interface will be the write data from the Type 1 command on the Primary interface. Converted cycles are restricted to a burst length of one PCI 32-bit data phase.

A Type 1 configuration write command on the Secondary interface will be converted to a Special Cycle command on the Primary interface provided all of the following conditions are met:

- Device Number is all ones - **S_AD[15:11]** = 11111_2
- Function Number is all ones - **S_AD[10:8]** = 111_2
- Register Number is all zeros - **S_AD[7:2]** = 00000_2
- Bus Number matches the Primary Bus Number of the bridge

The address during a Special Cycle is ignored by all PCI devices and a Master-Abort occurs on the PCI bus. However, the Master Abort error status bit (bit 13 of the Primary Status Register) is *not* set. The data for the Special Cycle on the Primary interface will be the write data from the Type 1 command on the Secondary interface. Converted cycles are restricted to a burst length of one 32-bit PCI data phase.

14.4.5 Private Type 0 Commands on the Secondary Interface

Type 0 configuration reads and write commands can be generated by the Secondary Address Translation Unit of the 80303 I/O processor. These Type 0 configuration commands are required to configure private PCI devices on the Secondary bus which are in private PCI address space. These commands are initiated by the Address Translation Unit and not by Type 1 commands on the Primary bus. Any device mapped into this private address space *will not* be part of the standard Secondary PCI address space and therefore will not be configured by the system host processor. These devices are hidden from PCI configuration software but are accessible from the 80303 I/O processor Secondary Address Translation Unit. See Chapter 15, "PCI Address Translation Unit" for a complete description of the private PCI address space implementation.

In Type 0 Secondary interface commands, **S_AD[31:11]** are used to select the target device **IDSEL** input. In Type 1 to Type 0 conversions, **P_AD[15:11]** are decoded to assert a unique address line from **S_AD[31:16]** on the Secondary interface as described in Section 14.4.2, "Type 1 Commands and Type 1 to Type 0 Conversions" on page 14-9. This leaves **S_AD[15:11]** on the Secondary interface open for a possibility of up to 5 address lines for **IDSEL** assertion of private PCI devices. These 5 address lines shall be reserved for private PCI devices on the Secondary PCI bus.

If more than five unique address lines are required, the Secondary IDSEL Select Register (SISR) can be programmed to block an additional 10 address lines during Type 1 to Type 0 conversions from the Primary interface. Secondary addresses **S_AD[25:16]** are the addresses that can be masked by the SISR register. By setting bits 0 through 9 (corresponding to **S_AD[25] - S_AD[16]**) in the SISR, the associated address line can be forced to remain deasserted for the **P_AD[15:11]** encodings of 00000_2 - 01001_2 and therefore are free to be used as an IDSEL select line for private Secondary PCI devices. Table 14-4 shows the possible configurations of **S_AD[31:11]** for public/private Type 0 commands on the Secondary interface. For example, if SISR Bit 9 is set, **S_AD[16]** will never be asserted during a Type 1 to Type 0 conversion from the Primary PCI bus. It can only be asserted by the Secondary Address Translation Unit.

If Primary interface receives a Type 1 command that intends to use an **S_AD** address lines reserved for private PCI devices, the bridge will perform the Type 1 to Type 0 conversion but not assert the reserved **S_AD** address line. The Type 0 command will then be ignored on the Secondary PCI bus.

By using the SISR register and the five reserved address lines, a total of 15 **IDSEL** signals are available for private PCI devices.

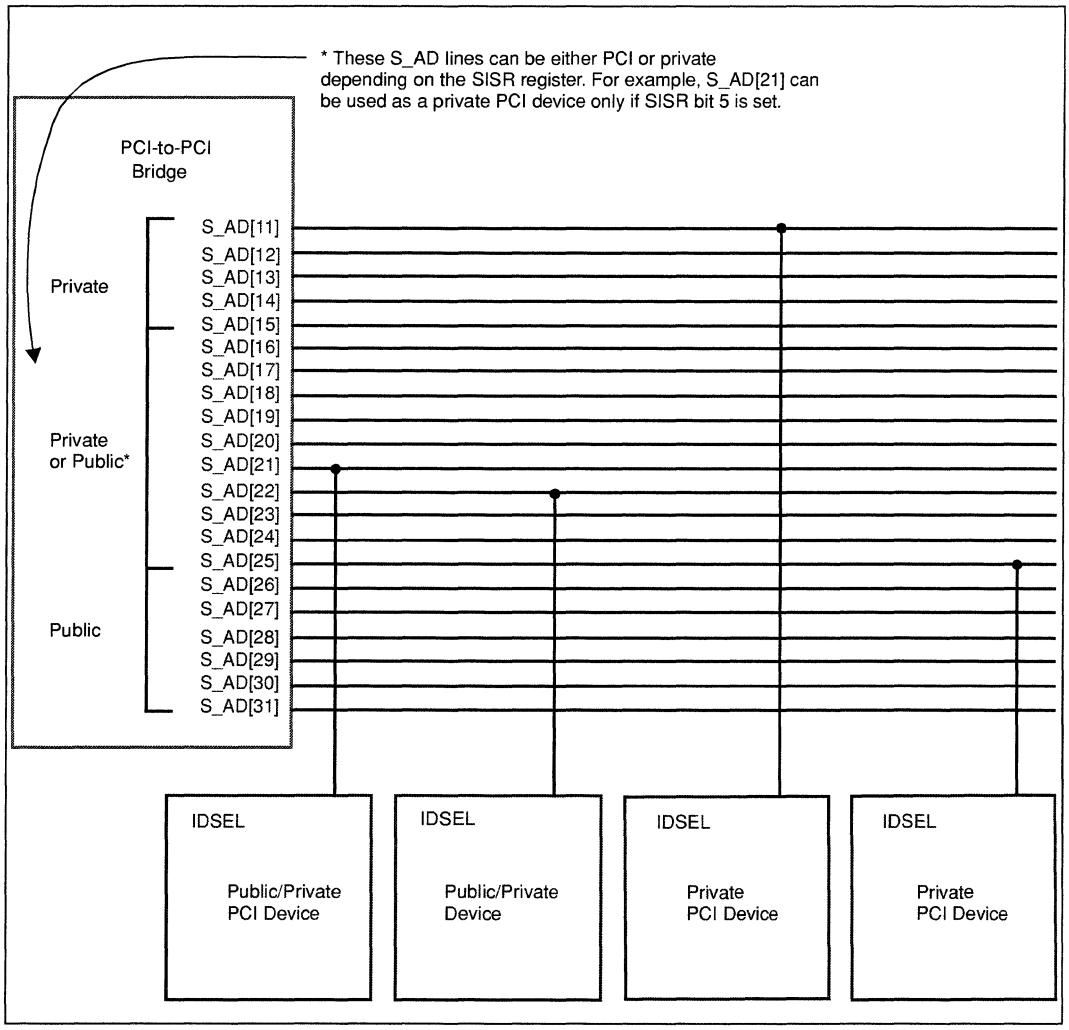
Table 14-4. Public/Private PCI Memory IDSEL Select Configurations

Primary Address P_AD[15:11]	Secondary Addresses S_AD[31:11] with All SISR Bits = 0	Secondary IDSEL Select Register Bits 9-0	Secondary Addresses S_AD[31:11] with SISR Bits Programmed
00000	0000 0000 0000 0001 0000 0_2	1XXXXXXXXX$_2$	0000 0000 0000 0000 0000 0_2
00001	0000 0000 0000 0010 0000 0_2	X1XXXXXXXX$_2$	0000 0000 0000 0000 0000 0_2
00010	0000 0000 0000 0100 0000 0_2	XX1XXXXXXX$_2$	0000 0000 0000 0000 0000 0_2
00011	0000 0000 0000 1000 0000 0_2	XXX1XXXXXX$_2$	0000 0000 0000 0000 0000 0_2
00100	0000 0000 0001 0000 0000 0_2	XXXX1XXXXX$_2$	0000 0000 0000 0000 0000 0_2
00101	0000 0000 0010 0000 0000 0_2	XXXXX1XXXX$_2$	0000 0000 0000 0000 0000 0_2
00110	0000 0000 0100 0000 0000 0_2	XXXXXX1XXX$_2$	0000 0000 0000 0000 0000 0_2
00111	0000 0000 1000 0000 0000 0_2	XXXXXXX1XX$_2$	0000 0000 0000 0000 0000 0_2
01000_2	0000 0001 0000 0000 0000 0_2	XXXXXXXX1X$_2$	0000 0000 0000 0000 0000 0_2
01001_2	0000 0010 0000 0000 0000 0_2	XXXXXXXXX1$_2$	0000 0000 0000 0000 0000 0_2

X = Don't Care

Figure 14-4 shows an example of connecting **S_AD** lines to **IDSEL** inputs of PCI devices and private PCI devices.

Figure 14-4. Secondary IDSEL Example



14.4.6 Special Cycles

The bridge unit will neither initiate nor accept PCI Special Cycle commands on either the Primary or the Secondary interface, except as a conversion. A mechanism is provided for converting Type 1 write commands to Special Cycles on either interface. See Section 14.4.4 for details.

14.4.7 Extended Configuration Space

The bridge unit includes an 8-byte extended configuration space. The extended configuration space can be accessed by a device on the Primary interface through a mechanism defined in the *PCI Local Bus Specification*, Revision 2.2.

In the bridge Primary Status Register (Section 14.15.4) the appropriate bit is set indicating that the Extended Capability Configuration space is supported. When this bit is read, the device can then read the Capabilities Pointer register (Section 14.15.21) to determine the configuration offset of the Extended Capabilities Configuration Space.

The first byte at the Extended Configuration Offset is the Capability Identifier Register (Section 14.15.36). This will identify this Extended Configuration Header space as the type defined by the *PCI Bus Power Management Interface Specification*, Revision 1.1.

Following the Capability Identifier Register will be the single byte Next Item Pointer Register (Section 14.15.37) which will indicate the configuration offset of an additional Extended Capabilities Header, if supported. In the bridge, the Next Item Pointer Register is set to 00H indicating that there are no additional Extended Capabilities Headers supported in the bridge configuration space.

14.5 Address Decoding

The 80303 I/O processor provides three separate address ranges that are used to determine which memory and I/O addresses are forwarded in either direction across the bridge portion of the 80303 I/O processor. There are two address ranges provided for memory transactions and one address range provided for I/O transactions. The bridge uses a base address register and limit register to implement an address range. The address ranges are positively decoded on the Primary interface with any address within the range considered a Secondary address and therefore capable of being forwarded downstream across the bridge. On the Secondary interface, the address ranges are inversely decoded.

In addition to the memory and I/O space, the bridge unit implements support for an ISA compatibility mode to support downstream expansion bridges and support for VGA graphics devices on the secondary interface of the bridge.

Standard bridge unit address decoding can also be modified by the Secondary Decode Enable Register (SDER). The bits within this register enable private address space on the Secondary side of the bridge.

The bridge will not accept PCI transactions generated by the Address Translation Units or the DMA Controller from the Secondary PCI interface. The bridge is capable of mastering transactions on the Primary interface that can be accepted by the Primary Address Translation Unit. (see Chapter 15, "PCI Address Translation Unit".)

14.5.1 I/O Address Space

The PCI-to-PCI bridge unit implements one programmable address range for PCI I/O transactions. A continuous I/O address space is defined by the I/O Base Register (IOBR) and the I/O Limit Register (IOLR) in the bridge configuration space. The upper four bits of the IOBR correspond to **AD[15:12]** of the I/O address and the lower twelve bits are always 000H forcing a 4 Kbyte alignment for the I/O address space. The upper four bits of the IOLR also correspond to **AD[15:12]** and the lower twelve bits are FFFH forcing a granularity of 4 Kbytes.

The bridge unit will forward from the Primary to Secondary interface an I/O transaction that has an address within the address range defined (inclusively) by the IOBR and the IOLR. In this instance the Primary interface acts as a PCI target and the Secondary interface acts as a PCI initiator for the bridged I/O transaction.

If an I/O read or write transaction is present on the Secondary bus, the bridge unit forwards it to the Primary interface if the address is outside the address range defined by IOBR and IOLR. In this instance the Secondary interface acts as a PCI target and the Primary interface serves as a PCI initiator.

The 80303 I/O processor only supports 16-bit addresses for I/O transactions and therefore any I/O transaction with an address greater than 64 Kbytes will not be forwarded over either interface. The bridge assumes **AD[31:16]** = 0000H even though these bits are not implemented in the IOBR and the IOLR. The bridge unit must still perform a full 32-bit decode during an I/O transaction to check for **AD[31:16]** = 0000H per the *PCI Local Bus Specification*, Revision 2.2.

I/O Read and I/O Write transactions with invalid byte enables (those that are inconsistent with the byte address) will be transparently passed by the bridge. In this case, it is expected that the target will target-abort, and the bridge will pass the target-abort back to the master.

For all PCI I/O transactions (I/O Read/Write Commands), the bridge does not use the PCI 64-bit extensions. I/O cycles are performed as 32-bit transactions only (**REQ64#** is never asserted).

The bridge response to I/O transactions can be modified by the following configuration bits:

- Master Enable bit in the Primary Command Register (PCR)
- I/O Enable bit in the Primary Command Register (PCR)
- ISA Enable bit in the Bridge Control Register (BCR)
- VGA Enable bit in the Bridge Control Register (BCR

The Master Enable bit needs to be set to allow the Primary interface to function as a PCI initiator (master) on behalf of transactions initiated on the Secondary bus. The I/O Enable bit must be set to allow the bridge to accept I/O transactions on the Primary interface. The VGA Enable bit in the BCR will cause I/O accesses where **AD[9:0]** are in the ranges 3B0H - 3BBH and 3C0H - 3DFH (inclusive of ISA addresses - **AD[15:10]** are not decoded) to be forwarded from primary to secondary and blocked from secondary to primary. See Section 14.5.3, "VGA Address Support" on page 14-20 for more details on VGA Compatible addressing.

The ISA Enable bit is discussed in the following section.

14.5.1.1 Disabling the I/O Address Range

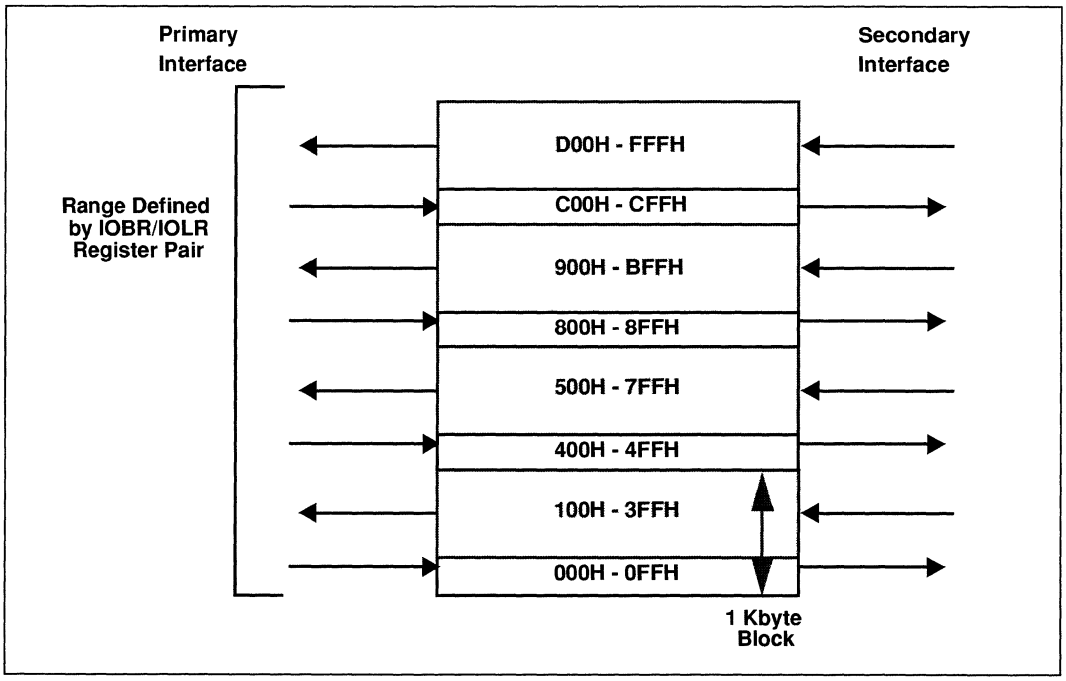
The I/O address range can be disabled for Primary to Secondary transactions by using either the I/O Enable bit or by using the I/O Base and Limit Registers. If the I/O Limit Register (IOLR) is programmed to a value less than the I/O Base Register (IOBR), the 80303 I/O processor will not forward any transactions from the Primary PCI interface to the Secondary PCI interface. In this case, *all* I/O transactions from the Secondary to the Primary will be forwarded upstream through the bridge.

14.5.1.2 ISA Mode

The PCI-to-PCI bridge unit of the 80303 I/O processor implements an ISA Enable bit in the Bridge Control Register (BCR) to provide ISA-awareness for ISA I/O cards on downstream PCI buses. ISA Mode only affects I/O addresses within the address range defined by the IOBR and IOLR registers. When ISA Mode is enabled by setting the ISA Enable bit in the Bridge Control Register (BCR), the bridge will filter out and *not* forward from Primary to Secondary I/O transactions with addresses in the upper 768 bytes (300H) of each naturally aligned 1 Kbyte block. Conversely, I/O transactions on the Secondary bus will inversely decode the ISA addresses and therefore forward I/O transactions with addresses in the upper 768 bytes of each naturally aligned 1 Kbyte block from Secondary to Primary.

ISA I/O cards only decode the lower 10 bits of the address (1 Kbyte). The upper 768 bytes of the 1 Kbyte is assigned for general I/O. Because these cards do not decode the upper 6 bits of the 16-bit I/O address, the ISA address is aliased 64 times in the 64 Kbyte I/O address space. The combination of ISA addressing modes and the 4 Kbyte I/O address granularity results in an address decode that is similar to EISA slot decoding. Devices on the Secondary interface may be mapped to the first 256 bytes of each 1 Kbyte block. ISA addressing and the ISA Enable bit do not affect ordering, posting or error handling behavior of the PCI-to-PCI bridge unit. See Figure 14-5 for an ISA address decoding diagram.

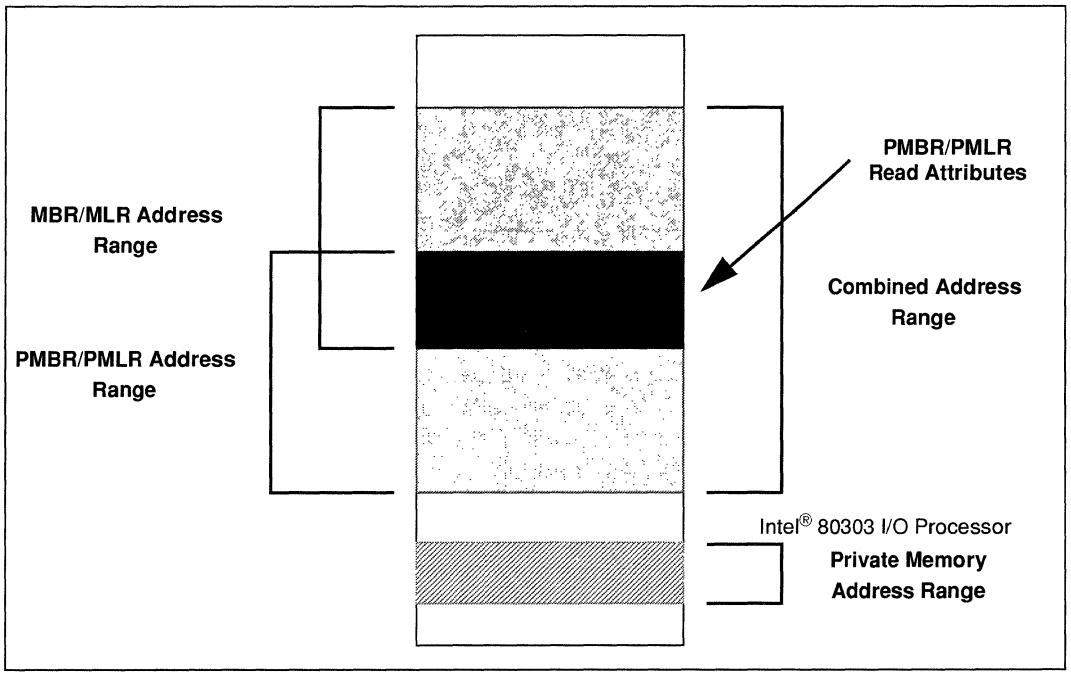
Figure 14-5. ISA Mode Address Decode



14.5.2 Memory Address Space

The bridge supports two separate address ranges for forwarding PCI memory accesses downstream from the Primary to Secondary interfaces. The Memory Base Register (MBR) and the Memory Limit Register (MLR) define one address range (often referred to as the Memory Mapped I/O Range) and the Prefetchable Memory Base Register (PMBR) and the Prefetchable Limit Register (PMLR) define the other address range. The prefetchable address range is used in determining which memory spaces are capable of prefetching without side effects. Both register pairs determine when the bridge will forward *Memory Read, Memory Read Line, Memory Read Multiple, Memory Write, and Memory Write and Invalidate* transactions across the bridge. In the case where the two register pairs overlap, one address range results that is the summation of both registers combined (Figure 14-6) with the prefetchable range having priority over bridge read transaction response.

Figure 14-6. Overlapping Memory Address Ranges



The upper twelve bits of the MBR, MLR, PMBR, PMLR (see Section 14.15 for all register definitions) registers correspond to address bits **AD[31:20]** of a Primary or a Secondary Single Address Cycle (SAC) memory address. For decoding purposes, the bridge assumes that **AD[19:0]** of both memory base registers are 00000H and that **AD[19:0]** of both memory limit registers are FFFFFH. This forces the memory address ranges supported by the bridge unit to be aligned on 1 Mbyte boundaries and to have a size granularity of 1 Mbyte. The lower four bits in all four registers are read only from the configuration address space and return zero when read.

Any PCI memory transaction present on the Primary bus that falls *inside* the address ranges defined by the two register pairs (MBR-MLR and PMBR-PMLR) will be forwarded downstream across the bridge from the Primary to Secondary interface. The command used on the Secondary interface may or may not match the command used on the Primary interface. Under certain conditions *Memory Write and Invalidate* commands can be converted to *Memory Write* commands (see Section 14.6.6.4) and within the non-prefetchable address space, *Memory Read Multiple* and *Memory Read Line* commands can be converted to *Memory Read* commands (see Section 14.6.5).

Any PCI memory transaction present on the Secondary bus that falls *outside* the address range defined by the two register pairs (MBR-MLR and PMBR-PMLR) is forwarded upstream across the bridge from the Secondary to Primary interface. These transactions default to prefetchable unless programmed to non-prefetchable in the EBCR. The Secondary interface will forward all Dual Address Cycles from the Secondary bus to the Primary bus. Dual address cycles are constrained to the upper 4 Gbytes of the 64-bit address space (see Section 14.5.4). Under certain conditions *Memory Write and Invalidate* commands can be converted to *Memory Write* commands (see Section 14.6.6.4) and within the non-prefetchable address space, *Memory Read Multiple* and *Memory Read Line* commands can be converted to *Memory Read* commands (see Section 14.6.5).

The 64-bit PCI extensions can be used by PCI memory commands for transactions initiated from either bridge PCI interface. See Section 14.6.3 for details on PCI 64-bit extensions. As a side note, the addition of a 64-bit PCI datapath still requires the use of DAC mode for 64-bit addressing. See Section 14.5.4 for details.

The bridge response to memory transactions on either interface may be modified by the following register bits from the bridge configuration space:

- Master Enable bit in the Primary Command Register (PCR)
- Memory Enable bit in the Primary Command Register (PCR)
- VGA Enable bit in the Bridge Control Register (BCR)

The Memory Enable bit in the PCR register must be set to allow the bridge to accept memory transactions on the Primary bus. The Master Enable bit in the PCR must be set to allow the Primary interface to master PCI transactions.

The VGA Enable bit in the BCR register forces the bridge to forward memory accesses in the address range from 0A0000H to 0BFFFFH from the Primary to Secondary and blocks addresses in the same range from Secondary to Primary. See Section 14.5.3, "VGA Address Support" on page 14-20 for more details on VGA Compatible addressing.

14.5.2.1 Burst Order

The bridge only supports linear incrementing addresses for burst order (**AD**[1:0] = 00_2). For any other burst order, the Bridge will disconnect the transaction after the first 32-bit data phase. See Section 14.8.1, "Delayed Read Transaction" on page 14-55 for information on non-linear MRLs and MRMs.

14.5.2.2 Disabling the Memory Address Range

The Memory address range can be disabled for Primary to Secondary transactions by using either the Memory Enable bit or by using the MBR-MLR and PMBR-PMLR register pairs. If the Memory Enable bit in the Primary Command Register (PCR) is cleared, the Primary interface of the bridge will not respond to any PCI memory transaction that falls within the MBR-MLR or PMBR-PMLR register pair address ranges. The Secondary interface is unaffected by Memory Enable bit in the PCR. In this case, all Memory transactions from the Secondary to the Primary will be forwarded upstream through the bridge.

If the Memory Limit Register (MLR) is programmed to a value less than the Memory Base Register (MBR) and the Prefetchable Memory Limit Register (PMLR) is programmed to a value less than the Prefetchable Memory Base Register (PMBR), the 80303 I/O processor will not forward any transactions from the Primary to the Secondary. In this case, all Memory transactions from the Secondary to the Primary will be forwarded upstream through the bridge.

14.5.3 VGA Address Support

Of the issues related to the use of VGA compatible devices in systems with 80303 I/O processor devices are VGA-ISA compatible addressing and VGA palette snooping. To support a VGA device on a downstream bus from the 80303 I/O processor, the PCI-to-PCI bridge is able to recognize and forward VGA addresses on the Primary interface to the Secondary interface. The PCI-to-PCI bridge unit of the 80303 I/O processor device does **not** support VGA palette snooping.

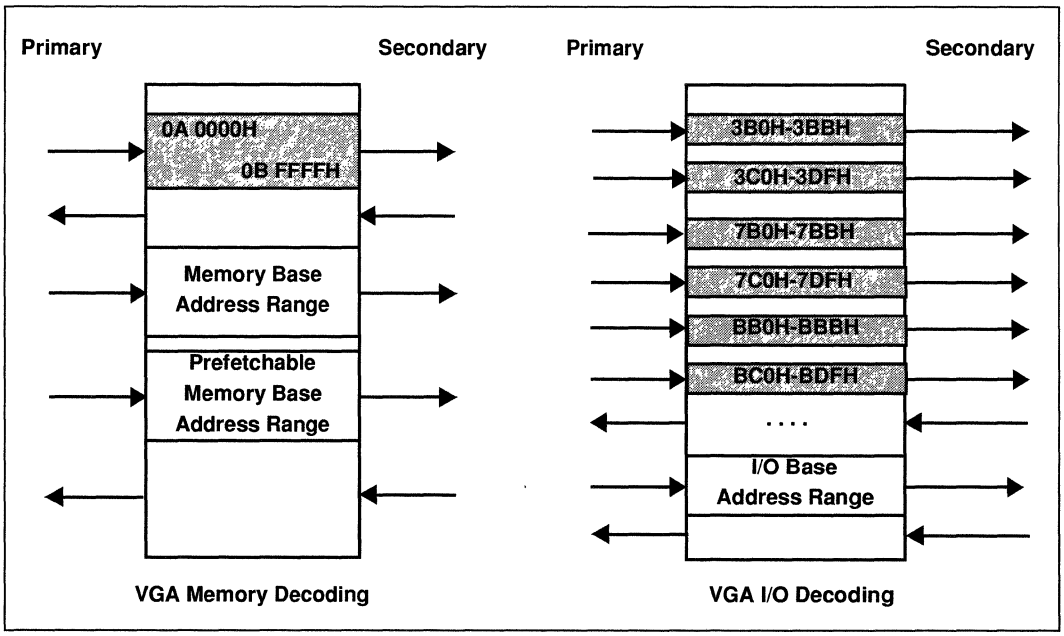
14.5.3.1 VGA Compatible Addressing

VGA addressing is used to allow the PCI-to-PCI bridge to support VGA frame buffer addressing and VGA register addressing. When the VGA Enable bit is set in the Bridge Command Register, the PCI-to-PCI bridge will positively decode memory accesses to a VGA frame buffer and I/O accesses to VGA registers on a Secondary bus. The following addresses are positively decoded on the Primary interface when the VGA Enable bit is set:

- VGA memory accesses - 0A0000H - 0BFFFFH
- VGA I/O accesses - **AD[9:0]** = 3B0H - 3BBH and 3C0H - 3DFH. These addresses are inclusive of ISA aliasing since **AD[15:10]** are not decoded for VGA I/O accesses These I/O addresses are aliased every 1KB throughout the first 64KB of I/O space. This means that **AD[31:16]** = 0000H.

VGA compatible addressing, when the VGA Enable bit is set, is not dependent on the address ranges programmed into the MBR/MLR and PMBR/PMLR register pairs for memory or the IOBR/IOLR register pair. The stated addresses will be forwarded from primary to secondary and blocked from secondary to primary regardless of the defined address ranges. In addition, VGA compatible addressing is not dependent on the ISA enable bit. VGA compatible addressing is dependent upon the I/O Enable bit and the Memory Enable bit and the respective memory and I/O transactions will be disabled if the appropriate I/O and memory bits are not set.

Figure 14-7. VGA Compatible Addressing



14.5.4 64-Bit Address Decoding - Dual Address Cycles

The bridge unit supports the dual address cycle (DAC) command for 64-bit addressing on the Secondary interface of the bridge unit only. Dual address cycle commands allow 64-bit addressing by using two PCI address phases; the first one for the lower 32 bits and the second one for the higher 32 bits. The DAC command is also used by the bridge Primary PCI interface to forward DAC cycles that appear on the secondary bus upstream.

The bridge unit decodes and forwards all dual address cycle commands from the Secondary to the Primary interface regardless of the address ranges defined in the MBR/MLR and PMBR/PMLR register pairs. DAC cycles are restricted to PCI memory commands only. I/O and configuration cycles are not supported in the greater than 4 GB address space. All DAC transactions are treated as prefetchable and adhere to the prefetch data amounts defined in Table 14-13.

The bridge unit defaults to Subtractive Decode timing for claiming dual address cycle commands. Subtractive Decode timing is defined as the assertion of **DEVSEL#** on the fourth clock after the address phase, the fifth clock after **FRAME#**, for DAC cycles. If the Secondary DAC Medium Decode Enable bit is set in the EBCR, the Secondary interface of the bridge claims all DAC transactions with medium decode timing.

The Primary interface will not forward dual address cycle commands to the Secondary interface.

The operation of DAC mode addressing for 32-bit or 64-bit buses, as defined by the *PCI Local Bus Specification*, Revision 2.2, is shown in Figure 14-8. For 32-bit bus operation or for a DAC request initiated from a 32-bit device on a 64-bit bus, **AD[63:32]** and **C/BE[7:4]#** are ignored. As a master on the Primary PCI bus, the bridge unit extends the address phase to two clock cycles. In the first cycle, the bridge drives the low order 32-bits of address on **AD[31:0]** and the DAC PCI command (1101_2) on **C/BE[3:0]#**. In the second address cycle, the bridge drives the high order 32-bit of address on **AD[31:0]** and the actual PCI read/write command on **C/BE[3:0]#**.

For 64-bit bus operation as a target on the Secondary bus, the bridge unit does not decode the high order address bits driven on **S_AD[63:32]** during the first address phase of the DAC cycle. The Secondary bridge interface waits for the second address phase to capture the complete 64-bit address and the actual PCI command for the transaction. As a master on the Primary PCI bus interface, the bridge operates as defined in Figure 14-8 and drives the high order 32 bits on **P_AD[63:32]** and the actual PCI command on **P_C/BE[7:4]#** during the first address phase of the DAC cycle. Both address phases as defined for a 32-bit bus are still performed.

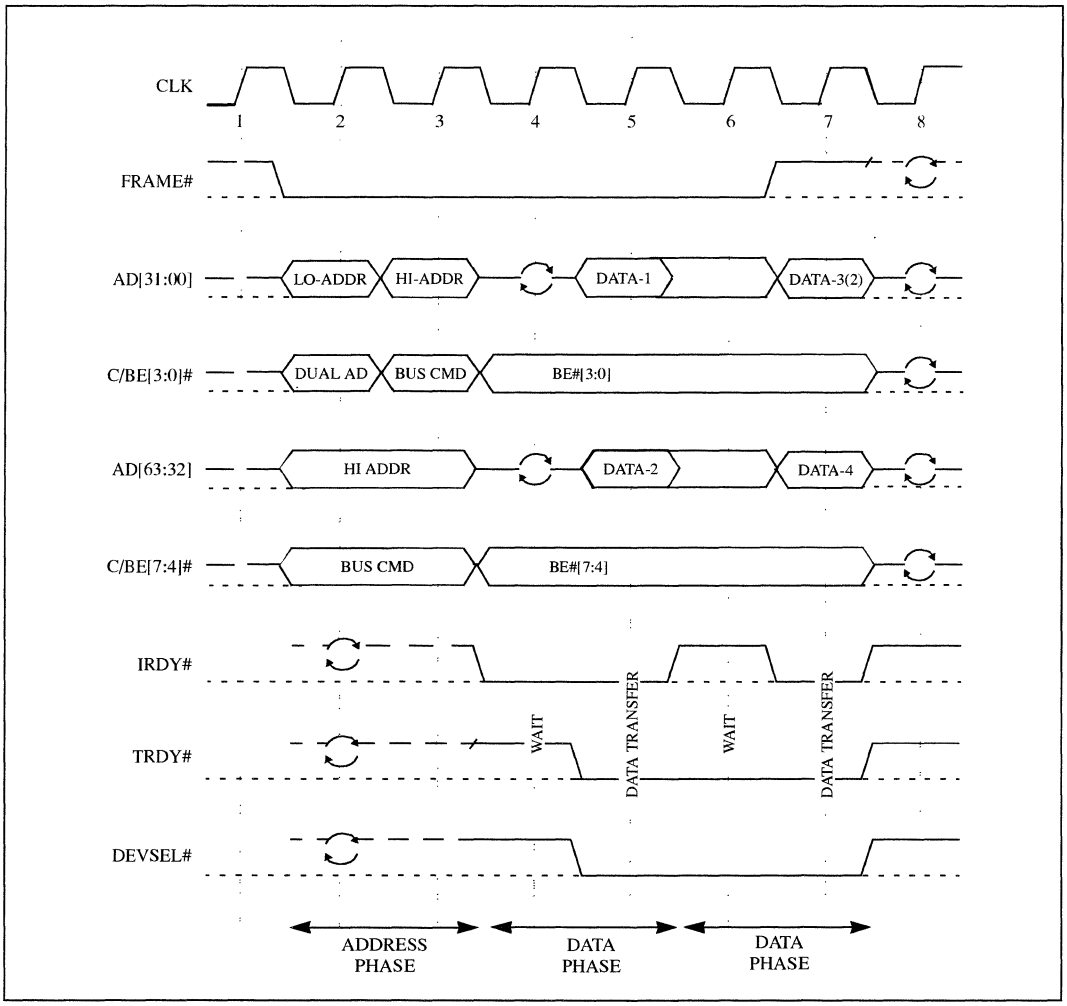
The response to DAC commands on the Secondary interface may be modified by the following register bit from the bridge configuration space:

- the Master Enable bit in the Primary Command Register (PCR)
- the Posting Disable bit in the Extended Bridge Control Register (EBCR)

The Master Enable bit in the PCR must be set to allow the Primary interface to master PCI transactions.

If the Posting Disable bit is set, the Secondary interface of the bridge unit will not accept ***any DAC write transactions at all.***

Figure 14-8. 64-bit Dual Address Read Cycle



14.5.5 Private Address Space

The bridge supports private address space by not claiming and forwarding upstream private Memory and I/O addresses on the Secondary PCI bus. These private addresses will appear to the bridge as Primary PCI addresses because they fall outside the Secondary PCI address space. Private addresses are only supported on the Secondary PCI bus and may be used for transactions from private devices to the Secondary ATU, transactions from the 80303 I/O processor to private devices, or transactions from private device to private device. The bridge will not claim transactions with these three types of private addresses if private addressing has been enabled:

1. Inbound transactions from private devices to the Secondary ATU.
2. Outbound transactions from the Secondary ATU or DMA channel 2 to private devices.
3. Peer transactions from Secondary devices.

For inbound private transactions, the Secondary ATU is responsible for claiming these transactions. If the Secondary ATU claims the transaction, the bridge will not claim or try to forward the transaction. The inbound ATU address space takes precedence over the inverse decoding performed by the bridge on the Secondary PCI interface.

For outbound transactions from the Secondary ATU or DMA channel 2, or peer transactions from Secondary device to private device, the programmer must use the Secondary Memory Base Register and Secondary Memory Limit Register (SMBR/SMLR) to define a private memory address range and the Secondary I/O Base Register and the Secondary I/O Limit Register (SIOBR/SIOLR) to define a private I/O address range. To enable this feature, the Private Memory Space Enable bit in the Secondary Decode Enable Register must be set. See Section 14.15.34. The bridge will not claim any Secondary PCI address that falls within a valid SMBR/SMLR and SIOBR/SIOLR address ranges if the Private Memory Space Enable or the Private I/O Space Enable bits are set.

14.5.6 Secondary PCI to Messaging Unit Access

The PCI-to-PCI bridge unit is responsible for providing the data path for access to the Messaging Unit (part of the Primary ATU). The bridge, in conjunction with the SATU, allows Secondary PCI masters to read and write the first 4 KB of the PATU inbound address space (the MU). The following statements apply to accessing the MU from the Secondary PCI bus through the bridge:

- The Secondary Bus - Messaging Unit Access Enable bit must be set. When set, the SATU will not claim the first 4 KB of its inbound address space, allowing the bridge the opportunity. This bit is contained in the ATUCR in the ATU configuration space (see Chapter 15, "PCI Address Translation Unit").
- The PCI memory read or write transaction (I/O or configuration cycles are not supported) must have a valid bridge address *outside* the PMBR/PMLR and MBR/MLR address ranges.
- The bridge unit Primary interface takes no other action to allow Secondary access to the MU. The application programmer is responsible for guaranteeing that the MU address is accessible from the Secondary PCI interface as an upstream bridge transaction. If the upstream transaction, meant for the Messaging Unit, is not at the correct address, a master abort will occur or the transaction will be claimed by the incorrect target.
- Normal Upstream read prefetch behavior applies. The Messaging Unit will disconnect (as a 32-bit device) after delivering one 32-bit Dword.

14.5.7 Address Decode Summary

Table 14-5 through Table 14-8 contain a summary of the address decode options. Table 14-5 and Table 14-6 summarize how addresses are decoded for Primary to Secondary transactions. Table 14-5 and Table 14-8 summarize how addresses are decoded for Secondary to Primary transactions. Each pair of tables is divided into one Memory transaction table and one I/O transaction table. The tables list the various control bits and the potential address ranges.

The response for the address is noted in each table entry. The response is determined by the control bits and by the address range the address falls into. The response may be one of following three:

- Forward the transaction across the Bridge (denoted as "Forward").
- Ignore the transaction and do not forward across the Bridge (denoted as "Ignore").
- This particular range is not valid and the response is dictated by another address range (denoted as "Not Valid").

The tables assume that the Memory and I/O Base and Limit address ranges are only valid when the Limit is greater than or equal to the Base. Table 14-5 is a summary of the Memory address decoding rules for Primary to Secondary Memory transactions.

Table 14-5. Primary to Secondary Memory Address Decoding Summary

Memory Enable bit	VGA Enable Bit	Special Memory Window Enable	Primary to Secondary				
			In MBR/MLR range	In PMBR/PM LR range	In VGA Memory Range	In Special Memory Range	Outside all valid ranges
0	0	0	Ignore	Ignore	Not Valid	Ignore	Ignore
0	1	0	Ignore	Ignore	Not Valid	Ignore	Ignore
1	0	0	Forward	Forward	Ignore	Ignore	Ignore
1	1	0	Forward	Forward	Forward	Ignore	Ignore
0	0	1	Ignore	Ignore	Not Valid	Forward	Ignore
0	1	1	Ignore	Ignore	Not Valid	Forward	Ignore
1	0	1	Forward	Forward	Ignore	Forward	Ignore
1	1	1	Forward	Forward	Forward	Forward	Ignore

Table 14-6 is a summary of the I/O address decoding rules for Primary to Secondary I/O transactions. The I/O Enable bit must be set to forward any I/O transactions. To be in the ISA range, the address must also fall in the IOBR/IOLR range. Also, the ISA range covers the complete IOBR/IOLR range.

Table 14-6. Primary to Secondary I/O Address Decoding Summary

I/O Enable bit	ISA Mode bit	VGA Enable Bit	Primary to Secondary				
			In IOBR/IOLR range	In VGA I/O Range	In ISA range (Lower 256 bytes)	In ISA range (Upper 768 bytes)	Outside all valid ranges
0	X	X	Ignore	Not Valid			Ignore
1	0	0	Forward	Ignore	Ignore	Not Valid	Ignore
1	0	1	Forward	Forward	Ignore	Not Valid	Ignore
1	1	0	Forward	Ignore	Forward	Ignore	Ignore
1	1	1	Forward	Forward	Forward	Ignore	Ignore

Note: When ISA is enabled, not all of the I/O addresses defined by the IOBR/IOLR range are forwarded downstream.

Table 14-5 is a summary of the Memory address decoding rules for Secondary to Primary Memory transactions.

The Private Address Space Enable bit in the SDER can disable forwarding of the SMBR/SMLR range.

Table 14-7. Secondary to Primary Memory Address Decoding Summary

Master Enable bit	Private Address Space Enable bit	Memory Enable bit	VGA Enable Bit	Special Memory Window Enable	Secondary to Primary						
					In MBR/MLR range	In PMBR/ PMLR range	In SMBR/ SMLR range	In ATU Inbound	In VGA Memory Range	In Special Memory Range	Outside all valid ranges
0	X	X	X	X	Ignore						
1	0	0	0	0	Forward	Forward	Not Valid	Ignore	Not Valid	Not Valid	Forward
1	0	0	0	1	Forward	Forward	Not Valid	Ignore	Not Valid	Ignore	Forward
1	0	0	1	0	Forward	Forward	Not Valid	Ignore	Not Valid	Not Valid	Forward
1	0	0	1	1	Forward	Forward	Not Valid	Ignore	Not Valid	Ignore	Forward
1	0	1	0	0	Ignore	Ignore	Not Valid	Ignore	Not Valid	Not Valid	Forward
1	0	1	0	1	Ignore	Ignore	Not Valid	Ignore	Not Valid	Ignore	Forward
1	0	1	1	0	Ignore	Ignore	Not Valid	Ignore	Ignore	Not Valid	Forward
1	0	1	1	1	Ignore	Ignore	Not Valid	Ignore	Ignore	Ignore	Forward
1	1	0	0	0	Forward	Forward	Ignore	Ignore	Not Valid	Not Valid	Forward
1	1	0	0	1	Forward	Forward	Ignore	Ignore	Not Valid	Ignore	Forward
1	1	0	1	0	Forward	Forward	Ignore	Ignore	Not Valid	Not Valid	Forward
1	1	0	1	1	Forward	Forward	Ignore	Ignore	Not Valid	Ignore	Forward
1	1	1	0	0	Ignore	Ignore	Ignore	Ignore	Not Valid	Not Valid	Forward
1	1	1	0	1	Ignore	Ignore	Ignore	Ignore	Not Valid	Ignore	Forward
1	1	1	1	0	Ignore	Ignore	Ignore	Ignore	Ignore	Not Valid	Forward
1	1	1	1	1	Ignore	Ignore	Ignore	Ignore	Ignore	Ignore	Forward

Table 14-8 is a summary of the I/O address decoding rules for Secondary to Primary I/O transactions. The ISA Enable pertains to the IOBR/IOLR range.

Table 14-8. Secondary to Primary I/O Address Decoding Summary

Master Enable bit	I/O Enable bit	ISA Mode bit	Private Memory Space Enable	VGA Enable	Secondary to Primary					
					In IOBR/IOLR range	In SIOBR/SIOLR range	In ISA range (Lower 256 bytes)	In ISA range (Upper 768 bytes)	In VGA I/O Range	Outside all valid ranges
0	X	X	X	X	Ignore					
1	0	0	0	0	Forward	Not Valid	Not Valid	Not Valid	Not Valid	Forward
1	0	0	0	1	Forward	Not Valid	Not Valid	Not Valid	Not Valid	Forward
1	0	1	0	0	Forward	Not Valid	Not Valid	Not Valid	Not Valid	Forward
1	0	1	0	1	Forward	Not Valid	Not Valid	Not Valid	Not Valid	Forward
1	1	0	0	0	Ignore	Not Valid	Not Valid	Not Valid	Not Valid	Forward
1	1	0	0	1	Ignore	Not Valid	Not Valid	Not Valid	Ignore	Forward
1	1	1	0	0	Forward	Not Valid	Ignore	Forward	Not Valid	Forward
1	1	1	0	1	Forward	Not Valid	Ignore	Forward	Ignore	Forward
1	0	0	1	0	Forward	Ignore	Not Valid	Not Valid	Not Valid	Forward
1	0	0	1	1	Forward	Ignore	Not Valid	Not Valid	Not Valid	Forward
1	0	1	1	0	Forward	Ignore	Not Valid	Not Valid	Not Valid	Forward
1	0	1	1	1	Forward	Ignore	Not Valid	Not Valid	Not Valid	Forward
1	1	0	1	0	Ignore	Ignore	Not Valid	Not Valid	Not Valid	Forward
1	1	0	1	1	Ignore	Ignore	Not Valid	Not Valid	Ignore	Forward
1	1	1	1	0	Forward	Ignore	Ignore	Forward	Not Valid	Forward
1	1	1	1	1	Forward	Ignore	Ignore	Forward	Ignore	Forward

Note: When ISA is enabled, not all of the I/O addresses defined by the IOBR/IOLR range are forwarded upstream.

14.6 Bridge Operation

The bridge unit of the 80303 I/O processor is capable of forwarding all types of memory, I/O and configuration commands from one PCI interface to the other PCI interface. Table 14-9 defines the PCI commands supported and not supported by the PCI-to-PCI bridge unit and its two PCI interfaces. PCI commands are encoded within the **C/BE[3:0]#** pins on either interface during the address phase of any PCI transaction (excluding DAC cycles which encode the DAC command in the first address phase and the read or write command in the second address phase).

Table 14-9. PCI Commands

C/BE#	PCI Command	Initiator: Primary Bus Target: Secondary Bus	Initiator: Secondary Bus Target: Primary Bus
0000_2	Interrupt Acknowledge	Ignore	Ignore
0001_2	Special Cycle	Ignore	Ignore
0010_2	I/O Read	Forward	Forward
0011_2	I/O Write	Forward	Forward
0100_2	Reserved	Ignore	Ignore
0101_2	Reserved	Ignore	Ignore
0110_2	Memory Read	Forward	Forward
0111_2	Memory Write	Forward	Forward
1000_2	Reserved	Ignore	Ignore
1001_2	Reserved	Ignore	Ignore
1010_2	Configuration Read	Forward	Ignore
1011_2	Configuration Write	Forward	Forward (Type 1 Only)
1100_2	Memory Read Multiple	Forward	Forward
1101_2	Dual Address Cycle	Ignore	Forward
1110_2	Memory Read Line	Forward	Forward
1111_2	Memory Write and Invalidate	Forward	Forward

14.6.1 PCI Interfaces

The 80303 I/O processor bridge unit consists of a Primary PCI interface and a Secondary PCI interface. When transactions are initiated on the Primary bus and claimed by the bridge, the Primary interface serves as a PCI target device and the Secondary interface serves as an initiating device for the true PCI target on the Secondary bus. The Primary bus is the initiating bus and the Secondary bus is the target bus. The sequence is reversed for transactions initiated on the Secondary bus. The interfaces are defined in the following sections.

14.6.1.1 Primary Interface

The Primary PCI interface of the bridge unit will be the interface connected to the lower numbered PCI bus between the two PCI buses that the 80303 I/O processor bridges. The Primary PCI interface must adhere to the definition of a PCI master and slave device as defined within the *PCI Local Bus Specification*, Revision 2.2 and the *PCI-to-PCI Bridge Architecture Specification*, Revision 1.1.

14.6.1.2 Secondary Interface

The Secondary PCI interface of the bridge unit will be the interface connected to the higher numbered PCI bus between the two PCI buses that the 80303 I/O processor bridges. The Secondary PCI interface must adhere to the definition of a PCI master and slave device as defined within the *PCI-to-PCI Bridge Architecture Specification*, Revision 1.1 and the *PCI Local Bus Specification*, Revision 2.2.

14.6.2 Claiming a PCI Transaction

In general, the PCI-to-PCI bridge unit, as a target on the initiating bus, uses medium timing to assert **DEVSEL#** to claim a bus transaction.

With the exception of Primary Interface Slow Decode under certain, special conditions, the performance of the bridge will **not** be affected. Both Slow and Medium Decode timings will meet the PCI specification of claiming a transaction within five clocks of the assertion of **FRAME#** by the initiating PCI device. The bridge target interface will claim the transaction depending on the transaction type and address. See the rules for address decoding for memory and I/O transactions in Section 14.5 and for configuration transactions in Section 14.4.

The bridge unit, as a master on the target bus, expects **DEVSEL#** to be asserted from the target device within five PCI clocks of asserting **FRAME#**. If the target interface does not receive **DEVSEL#** within the required amount of time, it will signal a Master-Abort on the target bus if the function is enabled (see Section 14.10.1 for Master-Abort information).

See the *PCI Local Bus Specification*, Revision 2.2 for full details on transaction claiming.

14.6.2.1 Latency Timers

A latency timer (LT) is used to create a mechanism that limits one masters ownership of a PCI bus in the presence of other bus masters. There are two latency timers in the bridge, one for each of the PCI interface masters.

The function of each latency timer is defined as:

- A LT is initialized and suspended (not counting) whenever a master interface (Primary or Secondary) is not asserting **FRAME#.**
- When a master interface asserts **FRAME#**, the LT will start counting down one for every PCI clock cycle that **FRAME#** is asserted.
- If the master interface deasserts **FRAME#** before the LT has expired (reached zero), the LT is meaningless to the transaction. The LT is initialized when **FRAME#** is deasserted.
- If the LT expires before the transaction completes, the interface must relinquish the bus and terminate the transaction (see Section 14.10.1) as soon as the master interface **GNT#** signal is deasserted. If the LT expires and the master interface **GNT#** signal is still asserted, the transaction is allowed to continue until it is complete or the master **GNT#** signal is deasserted. The exception to this rule is when a master is currently performing a Memory Write and Invalidate command on the bus. Refer to Section 14.10.1.3 for details.

In essence, the LT creates a minimum time slice that each master is allowed to own the PCI bus. Two registers exist within the bridge unit configuration space which define the maximum count and granularity of both the Primary and Secondary latency timers; the Primary Latency Timer Register (PLTR) and the Secondary Latency Timer Register (SLTR). Each register is 8 bits wide resulting in a time slice of up to 248 PCI clocks that each interface can own its respective PCI bus. The lower three bits (02 through 00) of the PLTR and the SLTR are hardwired to 000_2 which forces a minimum granularity for the timer of eight PCI clocks. The upper five bits of the register are programmable to allow the timer value for each PCI interface to be independently programmed to a value between 11111000_2 and 00000000_2 resulting in timer count of anywhere from 0 to 248.

14.6.2.2 Delayed Transactions

Delayed transactions are a method for processing PCI transactions that may exceed the *PCI Local Bus Specification*, Revision 2.2 requirement for no more than 16 clocks of latency between the PCI address and the first data word.

The bridge will process all transactions as Delayed transactions, except for Memory Write and Memory Write and Invalidate transactions. These two transactions can be processed as Posted transactions or as delayed write transactions. If the Posting Disable bit in the Extended Bridge Control Register is clear, Memory Write and Memory Write and Invalidate transactions will be processed as Posted transactions (default state). If Posting Disable bit is set, Memory Write commands will be processed as Delayed transactions and Memory Write and Invalidate commands will be processed as delayed Memory Write commands.

In a Delayed transactions performed by the bridge, the address, command, **REQ64#** and byte enable information required to complete the transaction is latched by the bridge in a transaction queue and the initiator is signaled a retry. For writes, the information includes the data to be written as well. The bridge performs the request on the target bus on behalf of the initiator. For reads, the returned data and the target response is stored in the bridge delayed read completion (DRC) queues. For writes, only the target response is recorded. The retried initiator must then repeat the original request on the initiating bus in order complete the full transaction.

A Delayed transaction consists of three parts:

- Request phase on the initiating bus
- Completion phase on the target bus
- Completion phase on the initiating bus

The request phase is when the transaction information is latched by the bridge and the bridge terminates the transaction with a Retry. This is referred to as a Delayed Request phase.

Once a Delayed Request transaction is accepted by the bridge, the bridge will initiate a completion phase on the target bus using the same transaction type as on the initiating bus. Data that accompanies the request transactions for delayed writes is held in the bridge delayed write request (DWR) queues. Data being returned for reads is written into the DRC queues.

The completion phase on the initiating bus is when the initiator repeats the original request and the bridge signals a termination other than Retry. This is referred to as a Delayed Completion transaction. The Delayed Completion transaction will terminate with the same termination as the target bus transaction. For example, if the target bus transaction terminated with Disconnect, the Delayed Completion transaction will terminate with Disconnect.

The bridge has a discard timer associated with each data buffer used for Delayed transactions (see Section 14.11.4). If the discard timer expires before the initiator repeats the original request, the data and associated request information is discarded.

14.6.2.3 Posted Transactions

In a Posted transaction performed by the bridge, the bridge stores the data in a write queue and signals a termination other than Retry. Once the bridge acquires the target bus, it completes the request.

Table 14-10 summarizes the difference between Delayed transactions and Posted transactions.

Table 14-10. Delayed Transactions vs. Posted Transactions

Delayed Transaction	Posted Transaction
For all PCI commands (except Special Cycle)	For Memory Write, Memory Write and Invalidate commands only
Requires repeated request	Does not require repeated request
Completes on target bus before initiating bus	Completes on initiating bus before target bus
Less efficient for writes	More efficient for writes

14.6.3 64-Bit Operation

Both the Primary and Secondary interfaces of the 80303 I/O processor are capable of PCI 64-bit operation to support data transfer rates of up to 264 MBytes/sec. at 33 MHz or 528 Mbytes/sec. at 66 MHz. The 64-bit PCI extensions add 39 additional signals to each bridge PCI interface. These signals and their functions are

- **AD[63:32]** - high order address/data bus
- **C/BE[7:4]#** - byte enables covering high order four bytes of data
- **PAR64** - even parity signal covering **AD[63:32]** and **C/BE[7:4]#**. Same timing as **PAR**
- **REQ64#** - used by a 64-bit master to request a 64-bit operation. Same timing as **FRAME#**
- **ACK64#** - used by a 64-bit capable target in response to **REQ64#** being asserted. Signifies to the master that the transaction can be completed with 64-bit transfers. Same timing as **DEVSEL#**.

At PCI bus reset, each individual PCI bus (Primary and Secondary) will independently sample their respective **REQ64#** signals. If this signal is low, the bus is 64-bit capable and the respective master state machines will attempt to complete all memory transactions as 64-bit cycles. See Section 14.12.3 for complete details of **REQ64#** detection by each PCI interface at power-up. Once a PCI bus interface is known to be 64-bit, the interface may attempt the following transaction types as 64-bit; *Memory Read, Memory Read Line, Memory Read Multiple, Memory Write, and Memory Write and Invalidate*. Configuration and I/O transactions are 32-bit only.

The bridge attempts a transaction on the target interface according to the size of the initiating interface. For example, a downstream write from a 32-bit master on the Primary bus will typically be attempted as a 32-bit transaction on the Secondary PCI interface. This approach improves the possibility of streaming between the initiator and its target. Another possibility occurs if a 32-bit write transaction is fully posted and meets the criteria of a 64-bit transaction, the write will be attempted as a 64-bit. transaction. The previous statements also apply to read transactions.

14.6.3.1 64-Bit Protocol

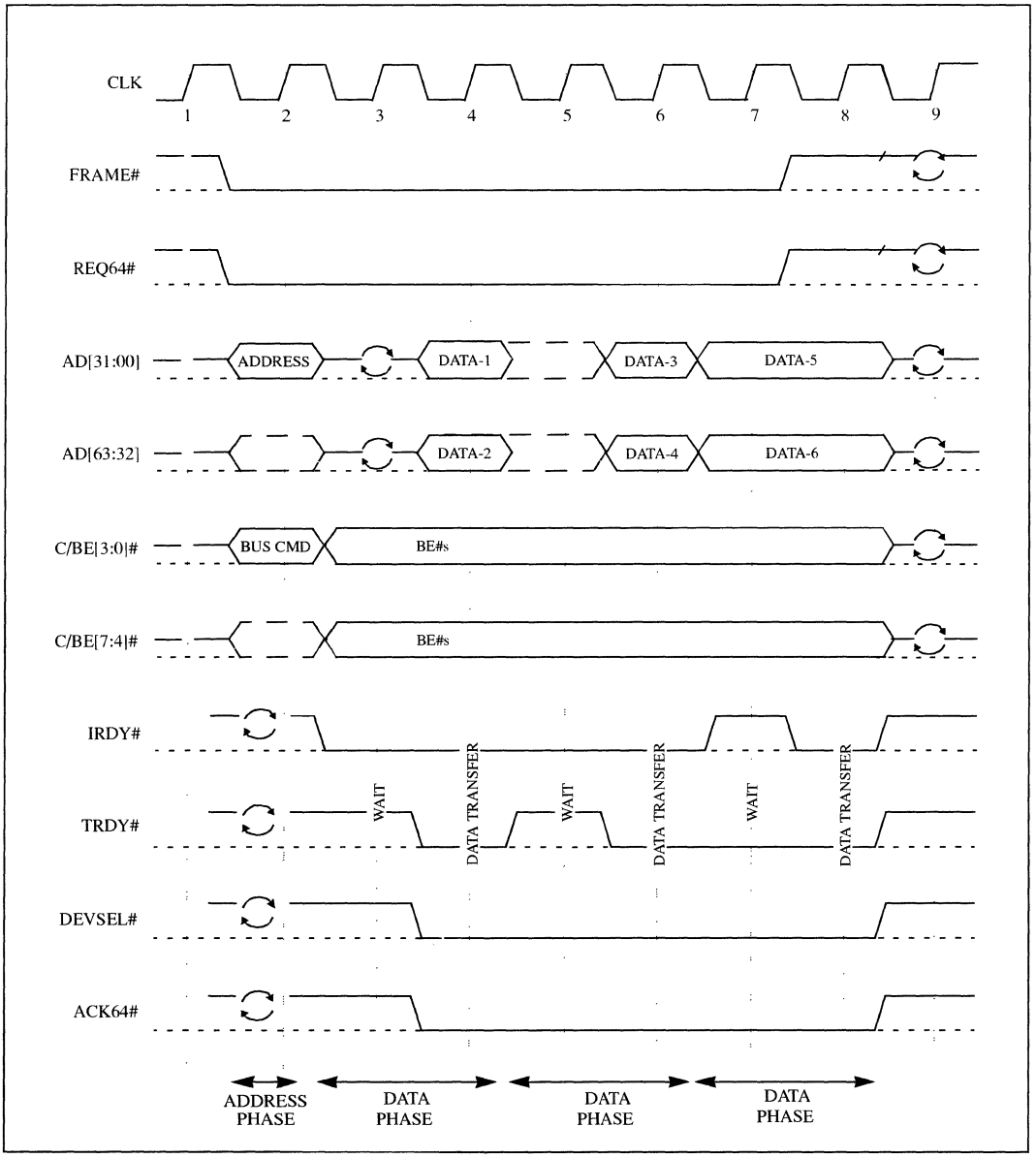
The 64-bit PCI extensions have been developed to coincide with the existing 32-bit protocol. The additional 32 bits of address/data require an additional four byte enables and a parity signal to cover them. The bus timing, protocol, and turn-around cycles behave exactly the same for the 64-bit signals as they do for the standard PCI interface signals with the exception of the 64-bit handshake signals referenced below.

The 64-bit handshake signals used by the 80303 I/O processor are **P_REQ64#** and **P_ACK64#** on the Primary interface and **S_REQ64#** and **S_ACK64#** on the Secondary interface. As a master, a PCI interface of the bridge will assert **REQ64#** with **FRAME#** to indicate to the target that a 64-bit transaction is being requested. **REQ64#** is asserted and deasserted with the exact timing as **FRAME#** for the master state machines. When **REQ64#** is asserted, the target of the memory operation is required to assert **ACK64#** with the same timing as **DEVSEL#** to allow a 64-bit transaction to proceed. If **ACK64#** is not asserted with **DEVSEL#**, the master interface must revert to a 32-bit transaction. See Section 14.6.3.2 for details on 64-bit operation with 32-bit targets.

A 64-bit transaction is required to have a 64-bit aligned address (**AD2** = 0). A master that is starting a request on an odd boundary (**AD2** = 1) must use a 32-bit transaction and not assert **REQ64#**. This is true for an initial request or as the result of a disconnect from a 32-bit target (see the next section). The bridge as a master on the target interface for reads will use 32-bit transactions when the initiator starts a 32-bit read transaction on an odd boundary

When **ACK64#** is asserted by the target of the transaction, a 64-bit transfer may proceed. As stated, a 64-bit transfer behaves exactly the same as a 32-bit transfer except that up to eight bytes of data are transferred during each PCI data phase. For the 64-bit transfer, the **AD[63:32]** and **C/BE[7:4]#** are reserved during the address phase. (assuming a SAC transfer). During the data phases, the master interface transfers up to eight bytes of data on each of the 8-byte lanes defined by **AD[63:00]**. As in a 32-bit transfer the master is capable of asserting any (or none) of the byte enables during each of the data phases within a burst transfer. Refer to Figure 14-9 for a diagram of a 64-bit transfer to a 64-bit target. **PAR64** for a 64-bit transfer has the same function and timing as **PAR** for a 32-bit transfer. **PAR64** must be asserted one clock after each address and data phase. 64-bit targets will qualify address parity checking using **PAR64** with the assertion of **REQ64#**. Although **AD[63:32]** and **C/BE[7:4]#** are reserved for SAC 64-bit transfers, **PAR64** must still be preserved and therefore stable values must be driven.

Figure 14-9. PCI 64-Bit Transfer to a 64-Bit Target



As a target, the slave state machines of both bridge PCI interfaces are capable of responding as a 64-bit target. When a PCI memory transaction is claimed by a bridge interface and the initiating master has requested a 64-bit transfer by asserting **REQ64#** with **FRAME#**, the bridge slave interface will assert and deassert **ACK64#** with the same timing and protocol as **DEVSEL#**. Further 64-bit slave operation is exactly like 32-bit operation with data being written or returned on both **AD[31:00]** and **AD[63:32]** using **C/BE[3:0]#** and **C/BE[7:4]#** respectively. **PAR64** must be driven with the same timing as **PAR** for read operations.

14.6.3.2 64-Bit Operation with 32-Bit Targets

When a 64-bit transfer is requested by the PCI master interfaces by the assertion of **REQ64#**, it is not guaranteed that the target of the transaction is capable of performing the 64-bit request. If the target is not 64-bit capable, **ACK64#** will remain deasserted when the target asserts **DEVSEL#** to claim the transaction. When a target signals that it cannot complete the transaction using 64-bit transfers, the bridge master interfaces are responsible for completing the transactions as a 32-bit master. Two possible conditions arise from a 32-bit target which does not respond with **ACK64#**:

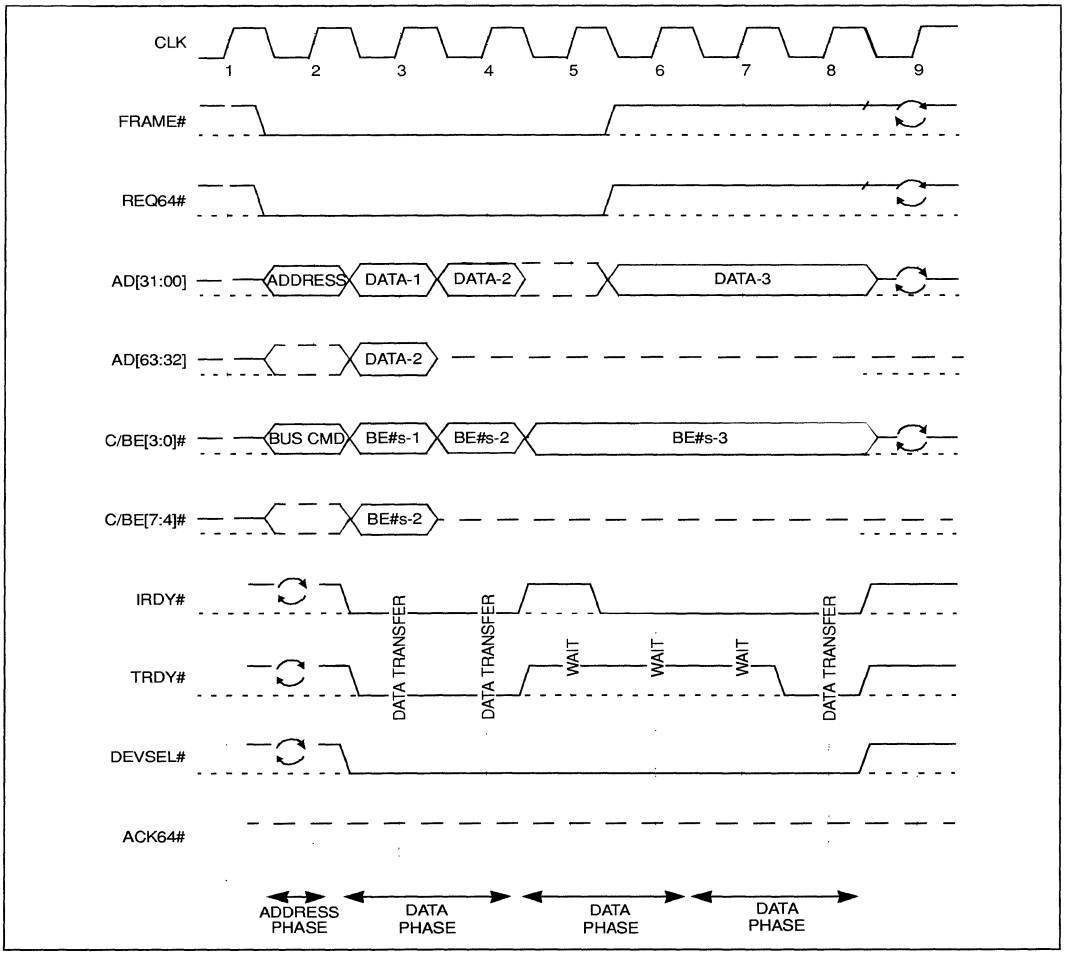
1. **ACK64#** deasserted but a burst can be sustained
2. **ACK64#** deasserted but a burst can not be sustained

If a 32-bit target does not respond with **ACK64#** and **STOP#**, it is capable of continuing a burst as a 32-bit target. For memory read requests, the bridge master interfaces changes to 32-bit operation by only expecting read data on the lower byte lanes, **AD[31:0]**. The master interfaces continue requesting read data (by the continued asserting of **IRDY#**) as 32-bit masters. No master completions are prematurely signaled due to 32-bit target response. For memory write operations, the master interface may already have the first data phase on the bus by the time it is detected that **ACK64#** has not been asserted. The bridge Primary/Secondary master interface discontinues driving data on the upper 4 bytes during the second data phase. The second data phase of the burst now contains the data from the high 4 bytes of the first data phase. The master interface stops driving the **AD[63:32]** and **C/BE[7:4]#** during data phase 2 and all subsequent data phases of the burst write transfer. See Figure 14-10 for a diagram of this transaction. As a note, a disconnect after the first data phase of the burst transfer write will result in the continuation of the write transaction as a 32-bit master only (no **REQ64#**). This works similar to the write transfer disconnected in the first data phase described in the next paragraph.

If a 32-bit target does not respond with **ACK64#** but asserts **STOP#**, the target will not continue the burst. If a read or write request is made and **STOP#** without **TRDY#** is signaled (Retry), the master interface must repeat the original read or write request as a 64-bit transaction. If the target signals a Disconnect with data (**STOP#** and **TRDY#**) on a write transaction, then only the lower four bytes of the 8-byte transfer have been delivered. The master state machines of the bridge unit repeat the request as a 32-bit master (no **REQ64#** assertion) using the upper four bytes of data from the disconnected transaction on **AD[31:00]** and the next address (i.e., if address 00H was used in the first 64-bit request, address 04H is used in the next 32-bit request). The bridge unit completes the memory write transaction as a 32-bit master until the data transferred from the initiating interface is exhausted (data from the posted memory write being completed on the target bus) regardless of the number of times the target disconnects the master or the address boundary on which it occurs. This occurs for 64-bit requests which are disconnected with no **ACK64#**. 64-bit requests disconnected with an **ACK64#** are continued as 64-bit requests. If the target signals a Disconnect with data on a read transaction (during the first data phase), then data has only been returned on **AD[31:00]**. No additional read requests are initiated due to delayed read transaction usage (See Section 14.6.5 for details).

Note that 32-bit targets create special circumstances for **FRAME#** signaling. For 32-bit, single Dword transfers, **FRAME#** is driven low and then high immediately in the next clock signaling last data phase. Due to the potential of requiring two 32-bit data phases to complete what was originally intended as one 64-bit data phase, this is not possible. **FRAME#** must not be deasserted until after **ACK64#** is returned.

Figure 14-10. 64-Bit Write Request with 32-Bit Transfer



As a PCI-to-PCI bridge, the 80303 I/O processor may be in a system environment with 64-bit devices on one bridge side and 32-bit devices on the other bridge side. This creates potential problems when a 64-bit master performs a delayed read to a 32-bit target with a prefetching *PCI Local Bus Specification*, Revision 2.2 bridge in the data path within non-prefetching address space. To account for this, the following rules apply to bridge read behavior:

- For a non-prefetchable read, the bridge will never return **ACK64#** and will always perform a 32-bit read (**REQ64#** not asserted) on the target interface.
- As is the case in all delayed reads, a disconnect during the delayed completion cycle on the target bus *does not* result in any additional reads.
- For all prefetchable reads, if the initiator starts a transaction with A2=0, the target interface will assert **REQ64#**. If the initiator starts a transaction with A2=1, the target interface will not assert **REQ64#**. This means that a 32-bit requestor can transfer data from a 64-bit target on a 64-bit target bus. If the read completion data is completely buffered and QWORD aligned at the tail end, the bridge will return the data as a 64-bit target.
- Delayed reads in prefetchable address space can return 64-bit data to a 64-bit master on the initiating bus even if the read on the target bus was from a 32-bit target.
- **REQ64#** is required to be asserted on a Retry sequence by the master but the target is under no obligation to assert **ACK64#** during the completion cycle even if it was asserted during the original transaction when the delayed read was enqueued and initiated.

14.6.4 66 MHz Operation

The PCI interfaces of the 80303 I/O processor are capable of PCI 66 MHz operation to support data transfer rates of up to 264 MBytes/sec. with a 32-bit bus or 528 Mbytes/sec. with a 64-bit bus. Differences between 33 MHz PCI and 66 MHz PCI are minimal. Both share the same protocol, and signal definitions. The 66 MHz PCI extension adds one additional signal to each bridge PCI interface. The signal and its function is

- **P_M66EN** - when asserted , indicates that the Primary PCI bus will run at 66 MHz
- **S_M66EN** - when asserted , indicates that the Secondary PCI bus will run at 66 MHz

Additionally, bit 5 of the "Primary Status Register - PSR" and bit 5 of the "Secondary Status Register - SSR" are set to a 1_2 indicating that the bridge is capable of 66 MHz operation on both interfaces.

At PCI bus reset, each individual PCI bus (Primary and Secondary) will independently sample their respective **M66EN** signals. If this signal is high, the bus is 66 MHz capable, and in the case of the Primary bus, the clock unit will be configured to accept a 66 MHz Primary PCI Bus clock.

The 66 MHz capable 80303 I/O processor supports the following Primary and Secondary bus frequency combinations:

- 66 MHz Primary bus, 66 MHz Secondary bus
- 66 MHz Primary bus, 33 MHz Secondary bus
- 33 MHz Primary bus, 33 MHz Secondary bus

The 80303 I/O processor does not support 33 MHz Primary/66 MHz Secondary bus operation, where the Secondary bus is operating at twice the frequency of the Primary bus. If **P_M66EN** is low (Primary bus at 33 MHz), then the 80303 I/O processor pulls down **S_M66EN** to indicate that the Secondary PCI bus is operating at 33 MHz.

The 80303 I/O processor generates clock signals **S_CLKOUT[5:0]** for the Secondary bus devices. The 80303 I/O processor divides the Primary bus clock **P_CLK** by two to generate the Secondary bus clock outputs whenever the Primary bus is operating at 66 MHz and the Secondary bus is operating at 33 MHz. The 80303 I/O processor detects this condition when **P_M66EN** is high and **S_M66EN** is low. The "Secondary Clock Disable Register - SCDR" provides the ability to selectively disable unused secondary output clocks following the deassertion of **P_RST#**.

For the 80303 I/O processor, the **P_M66EN** signal is routed from an external pin. While, for the 80303 I/O processor the **P_M66EN** signal is hardwired internally to be driven high since **P_CLK** will always be running at 66 MHz.

For more details on the 80303 I/O processor 66 MHz PCI clock scheme and the operation of the Secondary PCI bus clocks, please see Chapter 25, "Clocking and Reset".

14.6.5 PCI Read Transactions

The 80303 I/O processor supports memory read and I/O read transactions from both sides of the bridge unit. Memory read transactions are claimed if they are within the MBR/MLR or PMBR/PMLR address pairs on the Primary bus and outside the register pairs on the Secondary bus. I/O read transactions are claimed if they are within the IOBR/IOLR write transactions on the Primary bus and outside the address pair on the Secondary bus. Refer to the *PCI Local Bus Specification*, Revision 2.2 for full details on memory and I/O read transactions. Prefetchable Memory read commands will be attempted as 64-bit transactions (see Section 14.6.3). I/O, non-prefetchable reads and configuration reads are always performed as 32-bit operations. Refer to Section 14.7.1 for information on bridge queue operation during PCI read operations.

The bridge implements Delayed Read transactions in order to meet initial transaction latency requirements (from initiating bus **IRDY#** active to target bus **TRDY#** active). Delayed Transaction operation is described in Section 14.6.2.2.

The Delayed Read Request (DRR) transaction is the initial memory read or I/O read transaction that the bridge claims. The address, command, and byte enables of this transaction will be latched by the bridge and retained in the Transaction Queues. Once the bridge interface latches the address, command (including **REQ64#** for 64-bit transfers), and byte enables, it will signal a Retry to the initiator who is then required to re-issue the now delayed request.

If the DRR is accepted by the bridge, the bridge will then initiate the transaction on the target bus. Delayed Requests are accepted as new requests if all of the following conditions apply:

- The DRR does not match any DRRs currently held by the bridge in the initiating bus Transaction Queues.
- The request does not match up with the Delayed Completion currently held by the bridge. This new request must be checked against possible Delayed Completions to see if this is a repeated request that can be completed.
- The bridge has the ability to hold a Delayed Read Request in an available Transaction Queue and Delayed Read Completion Queue. In the situation where no queues are available, the bridge will signal a Retry without latching any information.

Two requests will match only if they have the exact same address, command, byte enables, and **REQ64#**. For the purposes of matching a delayed request with a delayed completion, the bridge unit will not compare byte enables for all prefetchable transactions that have linear addresses. Byte enables will be compared for prefetchable transactions that are non-linear.

If the request is accepted as a delayed transaction, the bridge retrys the master on the initiating bus and performs the same memory or I/O read command on the target bus. If the request is not accepted, the bridge signals a Retry to the master on the initiating bus with no action on the target bus.

When the target returns data on the target bus, the bridge will store the data in a Delayed Read Completion (DRC) Queue along with the associated Delayed Request information (address, command, **REQ64#**, and byte enables) that already exists in the Transaction Queue. The bridge will accept 1 or more data bytes to be stored in the DRC Queues. If additional queue space becomes unavailable (either from physically full or due to reserved space) and more data words are available from the target, the bridge will signal a Disconnect on the target bus.

The amount of data the bridge reads on the target bus and store in the DRC queues depends on:

- PCI command type
- whether the memory address space is prefetchable or not
- Size of Delayed Read Completion Queues available for data

Whether or not the read command prefetches depends on the address space, whether the transaction is upstream or downstream and the Upstream Prefetchable Enable bit in the EBCR. See Table 14-11 for a summary.

For downstream *Memory Read* commands, which address range (MBR-MBLR or PMBR-PMLR) is used to claim the address determines whether the memory is prefetchable or not. See Section 14.5.2.

For upstream *Memory Read* commands on the Secondary PCI bus, the bridge treats the memory as prefetchable or non-prefetchable depending on the Upstream Prefetchable Memory Enable bit in the Extended Bridge Control Register. If this bit is set, upstream memory is prefetchable. If this bit is clear, upstream memory is non-prefetchable.

For all *Memory Read Line* and *Memory Read Multiple* transactions in the non-prefetchable address space (either upstream or downstream), the bridge unit will alias the command to *Memory Read* on the target interface. For the purposes of matching MRL/MRM in the non-prefetchable address space, the bridge will match on the original command issued from the PCI master on the initiating interface. Non-prefetchable commands are always claimed as a 32-bit target (**ACK64#** deasserted) and attempted as 32-bit requests (**REQ64#** deasserted) on the target bus.

Table 14-11. Prefetchable and Non-Prefetchable Memory Summary

<table>
<tr><th rowspan="2">PCI Command</th><th colspan="2">Prefetchable</th><th colspan="2">Non-Prefetchable</th></tr>
<tr><th>Downstream</th><th>Upstream</th><th>Downstream</th><th>Upstream</th></tr>
<tr><td>Memory Read</td><td rowspan="3">In PMBR/PMLR Address Range</td><td rowspan="4">Upstream Prefetch Enable bit = 1 in EBCR</td><td>In MBR/MLR Address Range</td><td>Upstream Prefetch Enable bit = 0 in EBCR</td></tr>
<tr><td>Memory Read Line</td><td rowspan="2">In MBR/MLR Address Range[1]</td><td rowspan="2">Upstream Prefetch Enable bit = 0 in EBCR[2]</td></tr>
<tr><td>Memory Read Multiple</td></tr>
<tr><td>DAC Read</td><td>N/A</td><td>N/A</td><td>Upstream Prefetch Enable bit = 0 in EBCR[3]</td></tr>
</table>

1. MRL and MRM commands are aliased to Memory Read command on Secondary PCI Bus
2. MRL and MRM commands are aliased to Memory Read command on Primary PCI Bus
3. DAC MRL and MRM are aliased to Memory Read on the Primary PCI Bus

For DAC read commands on the Secondary PCI bus, the bridge will treat the memory the same as SAC transactions. See Table 14-11 for details.

The rules for the amount of data attempted to be read during a delayed transaction depend on the command used, the direction of the request (upstream or downstream) and whether or not prefetchable or non-prefetchable address space is used. Table 14-12 and Table 14-13 summarize the rules for downstream and upstream read transactions respectively. Note that the actual amount of data read will depend upon the DRC queue available at the time the DRR is enqueued in the Transaction Queue (refer to Section 14.7.1 for queue selection criteria), the amount of data delivered by the target on the actual target bus and the starting address of the read command.

The starting address of the read transaction must be on a cacheline boundary to prefetch the full data size determined in Table 14-12 and Table 14-13. For example a MRM read with a 32 byte cacheline configuration that wants to prefetch 128 bytes and start on address XXXXXX24H would only read a maximum of 124 bytes. If the same read had a starting address of XXXXXX20H, the maximum of 128 bytes could be read (assuming the target returned that much data).

Table 14-12. Downstream Memory Read Prefetch Size

Read Command	Prefetchable Memory Address Space		Non-Prefetchable Memory Address Space	
	CLS[1]= 8 (32 bytes)	CLS = 16 (64 bytes)	CLS = 8 (32 bytes)	CLS = 16 (64 bytes)
Memory Read	Up to 32 Bytes	Up to 64 Bytes	Up to 4 Bytes	Up to 4 Bytes
Memory Read Line	Up to 32 Bytes[2]	Up to 64 Bytes	Up to 4 Bytes	Up to 4 Bytes
Memory Read Multiple	Up to 64 Bytes	Up to 64 Bytes	Up to 4 Bytes	Up to 4 Bytes

1. CLS - Cache Line Size Defined by the Cache Line Size Register within the bridge configuration space
2. Up to 64 Bytes if the Downstream MRL Prefetch Size Bit is set in the Queue Control Register (See Section 14.15.35)

Table 14-13. Upstream Memory Read Prefetch Size

Read Command	Prefetchable Memory Address Space		Non-Prefetchable Memory Address Space	
	CLS[1] = 8 (32 bytes)	CLS = 16 (64 bytes)	CLS = 8 (32 bytes)	CLS = 16 (64 bytes)
Memory Read	Up to 32 Bytes	Up to 64 Bytes	Up to 4 Bytes	Up to 4 Bytes
Memory Read Line	Up to 32 Bytes[2]	Up to 64 Bytes[3]	Up to 4 Bytes	Up to 4 Bytes
Memory Read Multiple	Up to 256 Bytes	Up to 256 Bytes	Up to 4 Bytes	Up to 4 Bytes

1. CLS - Cache Line Size Defined by the Cache Line Size Register within the bridge configuration space
2. Up to 64 Bytes if the Upstream MRL Prefetch Size Bit is set in the Queue Control Register (See Section 14.15.35)
3. Up to 128 Bytes if the Upstream MRL Prefetch Size Bit is set in the Queue Control Register (See Section 14.15.35)

If the value in the CLS is anything other than eight or 16, the read prefetch behavior will be that of a CLS value of eight (32 bytes).

If selected read queue is large enough, MRL control bits within the Queue Control Register are capable of promoting the *Memory Read Line* Command prefetch size to 2x the amount in previous tables, if the command is in the prefetchable address space. Refer to Section 14.15.35 for details. The MRL prefetch bits increase the maximum prefetch size attempted during an MRL transaction.

I/O Read commands, Configuration Read, and all non-prefetchable read commands are limited to one 32-bit PCI data phase. The bridge reads and stores up to four bytes for these transaction types. The bridge will signal a Disconnect to the initiator if the master requests more than one DWORD.

The Delayed Completion transaction is the repeated memory read, I/O read, or configuration read transaction from the original initiator. The bridge matches the address, command, **REQ64#,** and byte enables of repeated transaction with those in the Transaction Queue and retrieves the data from the DRC queues. The bridge provides the requested data to the initiator and signals the termination (other than Retry) that matches what was used on the target bus.

The bridge will terminate the Delayed Completion transaction with:

- Completion termination if the transaction on the target bus terminated normally.
- Master-Abort termination or 1s (the number of 1s passed back, either 32-bit or 64-bit, is based on PCI bus size of initiating master, and in 64-bit bus size case, **REQ64#/ACK64#**) if the transaction on the target bus terminated with Master-Abort. See Section 14.10.1 for more information.
- Target-Abort termination if the transaction on the target bus terminated with Target-Abort.
- Disconnect termination if the transaction on the target bus terminated with Disconnect before the prefetch data amount was reached.

Any additional data words read from the target by the bridge but not ultimately requested by the initiator will be discarded upon transaction completion from the DRC queue. The bridge will *not* follow the termination rules above when it reads more data than is requested. The bridge will terminate with Completion termination if the initiator requests less data words than the bridge read from the target. For example, if the bridge reads eight Dwords from the target and is terminated with a Disconnect while the initiator only reads four DWORDs, the bridge will terminate with Completion termination.

If the expected number of prefetch data transfers are not received from the target for a *Memory Read Line* or a *Memory Read Multiple* command, the bridge will perform the same number of data transfers to the initiator during the Delayed Read Completion transaction as it receives from the target. For example, if the bridge, as a master, is disconnected by the target before reading the prefetch amount and only receives 16 DWORDs from the target, the bridge, as a target will only return 16 DWORDs to the initiator during the Delayed Read Completion transaction. An additional read transaction on the target bus is not issued to read the full prefetch amount as defined in Table 14-12 and Table 14-13.

If the initiator does not repeat the read transaction, the data and associated information may be discarded (see Section 14.11.4).

Under *PCI Local Bus Specification*, Revision 2.2, the initiator must repeat the read transaction exactly with the same address, byte enables, **REQ64#**, and command or the bridge will treat the transaction as a new request which will result in a deadlock condition. To support *PCI Local Bus Specification*, Revision 2.0 devices, the bridge can be programmed to ignore the memory read command (*Memory Read, Memory Read Line, and Memory Read Multiple*) when trying to match the current read transaction on an initiating interface with data in a DRC queue which was read previously (DRC on target bus). If the Delayed Read Command (DRC) Alias Bit in the QCR register is set, the bridge will not distinguish the read commands on transactions which where read through prefetchable address space only. For example, the bridge enqueues a DRR with a *Memory Read Multiple* command and performs the read on the target bus. Some time later, a PCI master attempts a *Memory Read* with the same address as the previous *Memory Read Multiple*. If the DRC Alias Bit is set and the transaction was in prefetchable address space, the bridge initiating interface would return the read data from the DRC queue and consider the Delayed Read transaction complete. If the DRC Alias bit in the QCR was clear, or if the transaction was in non-prefetchable address space, the bridge would not return data since the PCI read commands didn't match, only the address (and of course byte enables).

The bridge only supports the linear incrementing burst mode for Memory commands (**AD**[1:0] = 00_2). For a non-linear (**AD**[1:0] do not equal 00_2) Memory Read transaction, the bridge will fetch and transfer one Dword of data and then signal a Disconnect to the initiator. For a non-linear MRL or MRM (prefetchable or non-prefetchable) transaction, the bridge will convert the transaction to an MR on the target bus, fetch and transfer one Dword of data, and then signal a Disconnect to the initiator.

14.6.5.1 Read Streaming

Once the target interface of the bridge starts reading memory data, the initiating interface of the bridge allows the retried transaction access to the data in the DRC queue if there are at least 4 Dwords already in the queue. However, a target termination by the PCI slave on the target bus allows the retrying master on the initiating bus access to the data in the DRC queue even if less than 4 Dwords are already in the queue.

If the PCI master on the initiating interface is granted access to the DRC queue on a retried transaction, and the target interface of the bridge is filling, read streaming can occur. During read streaming, the bridge unit is filling on the target interface and draining on the initiating interface simultaneously. The following rules apply for read streaming

- Read streaming will only occur for the following master/target transaction sizes:
  - — 32-bit request: 32-bit target
  - — 64-bit request: 64-bit target
  - — 32-bit request: 64-bit target
- Read streaming only occurs for prefetchable transactions (see Table 14-11).
- The bridge unit will read beyond the prefetch read sizes to accommodate read streaming.
- Read streaming will stop if the target performs a disconnect, the master terminates, or the 4 KB read boundary is reached
- The bridge unit will never insert target or master D-D wait states on the initiating or target busses to accommodate read streaming except as defined below.

To provide the maximum window of opportunity to stream read data, the bridge target interface will insert up to 16 target waitstates (from the master assertion of FRAME#) when the master attempts to complete a read during a delayed completion transaction on the initiating interface. The following rules apply to this situation:

- The initiating interface only inserts waitstates if the target bus has **GNT#** and has asserted **FRAME#** for the read transaction matching the master on the initiating bus.
- Wait states are inserted until read low watermark in the DRC queue. The read low watermark is two Qwords.
- Once the read low watermark is reached the initiating interface will assert **TRDY#** for the first time and start delivering data to the target
- If the full 16 clocks has expired and the read low watermark has not been reached, the initiating interface will assert **STOP#** signaling a Retry to the initiator.
- This mechanism is used for all prefetchable read transactions crossing the bridge.

14.6.5.2 Read Boundary

The bridge is required not to read past a 4 Kbyte read address boundary. This prevents a prefetchable read access from crossing the boundary from a prefetchable range into a non-prefetchable range. When the 4 Kbyte read address boundary is reached, the bridge will signal a Disconnect on the target bus.

14.6.6 PCI Write Transactions

The 80303 I/O processor supports memory write and I/O write transactions from both sides of the bridge unit. Memory write transactions are claimed if they are within the MBR/MLR or PMBR/PMLR address pairs on the Primary bus and outside the register pairs on the Secondary bus. I/O Write transactions are claimed if they are within the IOBR/IOLR write transactions on the Primary bus and outside the address pair on the Secondary bus. Refer to the *PCI Local Bus Specification*, Revision 2.2 for full details on memory and I/O write transactions. Memory write commands will be attempted as 64-bit transactions (see Section 14.6.3). I/O and configuration write commands are always performed as 32-bit operations.

The bridge supports both posted and delayed write transactions for memory transactions. I/O write and configuration write transactions are always delayed transactions. The Posting Disable bit must be clear in the Extended Bridge Control Register (EBCR) to allow posting to occur from either interface of the bridge. If this bit is set, all write transactions are processed as delayed transactions.

14.6.6.1 Delayed Write Transactions

A Delayed write transaction is very similar to a Delayed read transaction. The bridge will claim the transaction on the initiating bus by asserting **DEVSEL#** and latch the address, command, byte enables into a Transaction Queue, and data into a Delayed Write Completion (DWC) Queue. It will then signal a Retry to the initiator.

Delayed write transactions are limited to one data cycle of 4 bytes for I/O writes, configuration writes, and for memory writes performed with posting disabled.

Delayed write transactions are used for:

- I/O Writes
- Configuration Writes
- All memory writes when the Posting Disable bit in the Extended Bridge Control Register (EBCR) is set. This means the bridge limits all write commands to one PCI data phase (4 bytes) when this bit is set.

The bridge then initiates the same command on the target bus. Once the target bus has been obtained, the bridge propagates write data from the initiating bus to the target bus. The bridge will keep the request information in a Transaction Queue and a DWC queue. The request information is the address, command (including **REQ64#**), byte enables, parity (if enabled), and data.

Once the write data has been successfully transferred to the target by assertion of **IRDY#** and **TRDY#** on the target bus, the bridge can now accept the repeated write command from the original initiator. At this time, the bridge will accept the request and attempt to match it with the transaction information in a Transaction Queue. The bridge must match the address, command, **REQ64#**, byte enables, parity (if parity is enabled), and data in order to signal a termination other than Retry to the initiator. The bridge unit will use the following terminations for delayed write cycles:

- Completion termination if the transaction on the target bus terminated normally.
- Master-Abort termination if the transaction on the target bus terminated with Master-Abort or normal termination (see Section 14.10.1.4).
- Target-Abort termination if the transaction on the target bus terminated with Target-Abort.

The initiator must repeat the write transaction exactly with the same address, **REQ64#**, byte enables, command, parity, and data or the bridge will treat the transaction as a new request. If the initiator does not repeat the write transaction, the data and associated information may be discarded (see Section 14.11.4).

14.6.6.2 Posted Write Transactions

In a posted write transaction, the bridge will accept the write data and assert **TRDY#** to the initiating bus before the data has been transferred to target interface for writing to the target bus. Once the bridge has acquired the target bus, it will transfer the write data to the target to complete the PCI transaction (both **IRDY#** and **TRDY#** asserted on the target bus).

For downstream posted write transactions, the bridge contains a 128-byte FIFO queue (Posted Memory Write Queue) for holding PMW data and separate address queue capable of holding up to 4 PMW transaction addresses (entries). For upstream posted write transactions, the bridge contains a 256-byte FIFO queue and a separate address queue capable of holding up to 8 PMW transaction addresses. This queue implementation will hold any number of posted memory writes up to the data queue depth and the size address queue. For example, the downstream queue could maintain two posted write transactions:

1. PMW 1
    a. 20 bytes of data in data queue
    b. 4 bytes of address (one entry) in address queue
2. PMW 2
    a. 74 bytes of data in data queue
    b. 4 bytes for address (one entry) in address queue

For a total of 94 bytes of data and two transaction entries. Another example of a possible upstream PMW Queue state could have the queue holding four transactions:

1. PMW 1
    a. 4 bytes for address (one entry) in address queue
    b. 4 bytes of data in data queue
2. PMW 2
    a. 4 bytes for address(one entry) in address queue
    b. 8 bytes of data in data queue
3. PMW 3
    a. 4 bytes for address in address queue
    b. 64 bytes of data in data queue
4. PMW 4
    a. 4 bytes for address (one entry) in address queue
    b. 128 bytes of data in data queue

For a total of 204 bytes of data in the data queue and four entries in the address queue. New posted memory write transactions are accepted in the PMW queues as long as there is enough data queue space to hold the data (8 bytes) and one transaction entry.

If the bridge PMW queues reach one less than the full state (defined as 8 bytes free) and the bridge has not acquired the target bus to transfer out data, the bridge will signal a Disconnect to the initiator on the initiating bus on the last transfer that would fill the PMW queue. A state may exist where a 64-bit master is filling the PMW queue and the bridge unit is transferring to a 32-bit target on the target interface. For this or any other bus configuration where the Primary and Secondary

buses don't maintain the same PCI bandwidths, the bridge PMW Queues maintain data integrity and guarantee no data is lost by disconnecting a filling master on an initiating bus before an overflow condition exists.

The PMW Queues are capable of streaming write data from an initiating bus to a target bus assuming no prior PMW transactions exist in the PMW Queue (by the time the queue fills) being accessed and the target interface is capable of acquiring the bus and the addressed target device. This can continue until the target initiates termination (Disconnect or Target-Abort), the bridge initiates termination on the target bus (Time-out), the PMW Queue fills, the transaction is an MWI and a full cacheline is not free in the PMW Queue, or the initiator completes the required number of write transfers. In the situation where the target bus transaction terminates while the initiator is still transferring data, the PMW Queue will fill and a Disconnect would occur in the same situation as when the initiator started the original transaction (see previous paragraph). In addition, neither the Primary or Secondary interface of the bridge will insert target wait states (deassertion of **TRDY#**) or master wait states (deassertion of **IRDY#**) to support the sustaining of a streaming transaction through the bridge. The target interface of the bridge may insert master wait states to guarantee the transfer of an entire cacheline during Memory Write and Invalidate transfers. See Section 14.6.6.4, "Memory Write and Invalidate Command" on page 14-45.

The bridge unit is capable of supporting simultaneous write posting in both directions across the bridge.

When a memory write transaction is accepted on the initiating bus interface, and transaction ordering supports the immediate draining of the current transaction (see Section 14.7.2), the default is for the target bridge interface to assert **REQ#** once the first PCI dataphase has been entered into the posted write queue. In this case, the PCI dataphase is a 32-bit word if the master is driving 32-bits or a 64-bit word if the master is driving 64-bits.

The bridge only supports the linear incrementing burst mode for Memory write commands. The bridge will signal a Disconnect to the initiator after the transfer of the first data phase if the burst mode is *not* linear incrementing.

See Section 14.7.1.1 for complete details on posted memory write queues.

14.6.6.3 Memory Write Command

A PCI initiator will use the memory write command for transferring data to one of the memory address spaces defined in one or both of the MBR/MLR and the PMBR/PMLR register pairs. Memory write transactions can be either posted or delayed transactions. This is determined by the Posting Disable bit in the Extended Bridge Control Register. If clear, posted transactions are used. If set, delayed transactions are used.

Delayed Memory Write commands will only transfer one 32-bit PCI data phase. This means **FRAME#** will only be asserted for one clock on the target interface and that the initiating interface will signal a target Disconnect after the first data transfer. Refer to the *PCI Local Bus Specification*, Revision 2.2 for more details.

14.6.6.4 Memory Write and Invalidate Command

The Memory Write and Invalidate (MWI) command is essentially identical to the Memory Write command except it guarantees a minimum transfer of at least one cacheline as defined by the Cacheline Size Register (CLSR). The initiating PCI master will only allow the transaction to cross the predefined cacheline boundary if it intends to transfer the entire next cacheline.

The target interface of the bridge must guarantee that there is enough free queue space in the PMW data queue to accept an MWI transaction. If this is not true, the MWI is retried. Once a full cacheline is accepted and the master continues bursting into the next cacheline, this decision needs to be made again and so on. For example, if in the upstream queue, an MWI is active on the Secondary bus, a full cacheline (32 bytes in this case) has already been transferred, and the U_PMWD queue only has 24 free bytes available, the Secondary interface of the bridge will perform a disconnect without data on the first data phase of the next cacheline. If U_PMWD queue, in this case, had 32 bytes or more of free queue space available, the Secondary interface would continue accepting the next cacheline.

When the bridge accepts an MWI command which is terminated with a Master Abort on the target bus, the bridge may disconnect the transaction before transferring an entire cacheline into the queue.

When the bridge accepts an MWI command which is terminated by the master before the entire cacheline is transferred, the bridge will complete the transaction using a Memory Write Invalidate command to transfer the partial cacheline.

When the bridge accepts an MWI command which is disconnected by the target (on the target interface) before the entire cacheline is transferred, the bridge will complete the transaction using a Memory Write command to transfer the partial cacheline. If the transaction is still in progress (streaming), the bridge is free to disconnect the initiator with a target disconnect on the initiating bus in the middle of a cacheline. No other action is taken by the bridge unit; no error is reported.

To satisfy the MWI command protocol, the target interface of the bridge will deassert IRDY# (master wait state) when a stream is occurring (data transferred on initiating and target interface simultaneously) and the target interface is capable of a faster transfer rate than the initiating interface. This can occur due to varying bus or target widths or bus speeds or master wait states from the initiator on the initiating bus. (Master wait states may cause the bridge to insert more than 8 IRDY# wait-states between data phases on the target bus.)

The bridge unit converts a MWI command to a Memory Write command if the CLSR is programmed to a value of zero or if the Cacheline Size Register is programmed to a value other than 8 or 16. Refer to the *PCI Local Bus Specification*, Revision 2.2 for the full details of a Memory Write and Invalidate command.

If posting is disabled, the bridge will not allow the MWI command to appear on the target bus. The bridge will convert the MWI to a Memory Write and only allow one PCI data phase on the target bus.

If the MWI Alias bit is set in the Queue Control Register, the bridge will accept an MWI command as long as the PMW queue is not in a full state. This means that there does not need to be at least a cacheline of queue space free to accept the MWI. When the MWI Alias bit is set, the bridge target interface will alias the MWI command to a *Memory Write* command for transfer to the PCI target. MWI master rules do not apply. In addition, MWI transactions which start on a non-cacheline boundary are treated as if the MWI alias bit is set, i.e. they are aliased to an *Memory Write*.

14.6.6.5 I/O Write Command

All I/O Write transactions will be processed as Delayed transactions. The 80303 I/O processor is restricted to 16-bit addressing for I/O transactions although it still must decode the full 32-bits of address and verify that **AD[31:16]** = 0000H. The bridge will claim any transaction inside the 16-bit address range defined by the I/O Base and I/O Limit registers on the Primary bus and outside the address range on the Secondary address bus.

14.6.6.6 Write Boundary

The bridge of the 80303 I/O processor imposes a naturally-aligned 4096-byte write boundary for posted write transactions only. When the bridge unit detects a write boundary, the initiating interface will signal a Disconnect to the initiator and complete delivery of the write data within the PMW Queue to the target interface. The write boundary can be considered an address counter which is incremented by one for every byte of a burst transaction. The write boundary is imposed when the lower 12 bits of the counter reach zero.

14.6.6.7 Qword Unaligned Memory Write Transactions

To minimize the number of null write transactions on the PCI bus, the bridge has the following behavior for Qword (8-byte) unaligned write transactions:

- If the memory write transaction is completely posted within the bridge posted memory write queue (upstream or downstream), and the transaction is QWORD aligned at both the head and tail of the transaction, then the bridge will attempt this as a 64-bit transaction (assuming the bus is defined as 64-bit).
- If the memory write transaction is in a streaming mode (active on initiating bus while the target interface is acquired), the bridge will attempt the transaction on the target bus based on the initiating bus transaction width:
  - — 64-bit memory write transaction on initiating bus is attempted as a 64-bit transaction on the target bus
  - — 32-bit memory write transaction on initiating bus is attempted as a 32-bit transaction on the target bus

14.6.6.8 Fast Back-to-Back Transactions

The 80303 I/O processor bridge unit does not generate fast back to back transactions. The Fast Back to Back Enable bits in the Primary Command Register (PCR) and in the Bridge Control Register (BCR) are ignored.

The bridge unit is capable of accepting fast back to back transactions from the same PCI master.

14.7 Queue Architecture

The extensive queueing architecture in the 80303 I/O processor allows the bridge to achieve maximum PCI throughput between buses while keeping read latency to a minimum. The bridge unit queues are responsible for transferring all transactions from an initiating bus to a target bus. The queues are classified under the following five categories:

- **Posted Memory Write Queue** - used to forward posted memory write operations (*Memory Write, Memory Write and Invalidate*) from an initiating bus to a target bus. By definition, the data within a posted memory write queue has already completed on its source bus.
- **Delayed Read Completion Queue** - used to forward memory read (*Memory Read, Memory Read Line, Memory Read Multiple*), configuration read, and I/O read data from the target bus back to the initiating bus. Read data within a DRC Queue is the result of a Delayed Read Completion transaction and is not considered "read" until delivered to the requesting master on the initiating bus.
- **Delayed Write Completion Queue** - used to forward I/O write and configuration write data from the initiating bus to the target bus. Write data in a DWC queue is the result of a Delayed Write Request transaction and is not considered written until delivered to the addressed PCI slave on the target bus. The DWC Queue also returns the status of the write operation on the target bus back to the master on the initiating bus.
- **Transaction Queue** - used to hold the address, **REQ64#,** and command of a delayed request cycle. This includes all memory and I/O reads as well as all delayed write operations.
- **Address Queue** - used to hold the address and command of a posted memory write operation.

The bridge is capable of holding multiple posted memory writes and delayed reads in either direction simultaneously. This high performance architecture requires strict adherence to PCI-to-PCI bridge transaction ordering rules. The ordering requirements for the 80303 I/O processor bridge architecture is defined in Section 14.7.2. Refer to the *PCI Local Bus Specification*, Revision 2.2, Appendix E for complete details on PCI transaction ordering.

14.7.1 Queue Operation

Table 14-14 details a summary of the different queues present in the 80303 I/O processor bridge unit.

Table 14-14. Bridge Unit Queue

Queue Mnemonic	Queue Name	Queue Size	Transactions Possible in Queue[1]
U_PMWD	Upstream Posted Memory Write	256 Bytes	MWI, MW
U_PMWAD	Upstream Posted Memory Write Address	8 Entries Address/Command	MWI, MW
U_DRC0	Upstream Delayed Read Completion 0	256 Bytes	MR, MRL, MRM
U_DRC1	Upstream Delayed Read Completion 1	256 Bytes	MR, MRL, MRM
U_DRC2	Upstream Delayed Read Completion 2	4 Bytes	Non-prefetchable Read, IOR
U_DWC	Upstream Delayed Write Completion	4 Bytes	CW, IOW
U_TRQ0:2	Upstream Transaction Queues 0:2	Address / Command	DRR address
U_TRQ3	Upstream Transaction Queue 3	Address / Command	DWR address
D_PMWD	Downstream Posted Memory Write	128 Bytes	MWI, MW
D_PMWAD	Downstream Posted Memory Write Address	4 Entries Address/Command	MWI, MW
D_DRC0	Downstream Delayed Read Completion 0	64 Bytes	MR, MRL, MRM
D_DRC1	Downstream Delayed Read Completion 1	64 Bytes	MR, MRL, MRM
D_DRC2	Downstream Delayed Read Completion 2	4 Bytes	Non-Pref MR, CR, IOR
D_DWC	Downstream Delayed Write Completion	4 Bytes	CW, IOW
D_TRQ0:2	Downstream Transaction Queues 0:2	Address / Command	DRR address
D_TRQ3	Downstream Transaction Queue 3	Address / Command	DWR address

1. MRL - Memory Read Line, MRM - Memory Read Multiple, MR - Memory Read (Non-Prefetchable is noted, Prefetchable otherwise), MW - Memory Write, MWI - Memory Write & Invalidate, IOR - I/O Read, IOW - I/O Write, CD - Configuration Read, CW - Configuration Write, DRR - Delayed Read Request, DWR - Delayed Write Request

Figure 14-1 contains a block diagram of the queues defined in Table 14-14. There are five different types of queues in the bridge. Each queue type has a specific responsibility for either upstream or downstream transactions. Detailed explanations of the queue types follow.

14.7.1.1 Upstream/Downstream Posted Memory Write Queue Structures

Each upstream and downstream PMW Queue structure consists of two separate queues: one for data, one for address. The upstream data queue, U_PMWD, has a queue depth of 256 bytes and moves write transactions from the Secondary bus to the Primary bus. The corresponding address queue, U_PMWAD, holds the address of the posted memory write transactions. There are a total of 8 entries in the U_PMWAD for holding up to eight SACs. DAC addresses are handled in a separate address queue sitting beside the U_PMWAD for holding up to 8 DAC memory writes. If the write transaction is a DAC cycle, the upper 32-bits of address is entered into this piece of the U_PMWAD. Each entry is the address for the write data which exists in the U_PMWD queue. The size of the data (transaction burst size) attached to each address queue entry is variable.

The downstream queue, D_PMWD, has a depth of 128 bytes and moves write transactions from the Primary bus to the Secondary bus. The address queue, D_PMWAD, contains four address entries for up to four SAC addresses. Downstream DACs are not supported. The queue operation is the same as the upstream description.

Memory write transactions fill the tail of the queue on the initiating bus and are drained from the head of the queue on the target bus. The following rules apply to the initiating bus interface and govern the acceptance of data into the tail of the PMW Queue:

- A memory write operation claimed by the slave PCI interface on the initiating bus is accepted into the queue if the data queue is in a non-full state and can accept at least one data phase and the address queue has free space for the address. A non-full state for the data queues are defined as:
  - — 1/4 empty or 1/2 empty based on the setting of the QCR. See Section 14.15.35.
  - — The length of a cacheline (determined by the Cacheline Size Register, see Section 14.15.7) for *Memory Write and Invalidate* transactions.

  A non-full state for the address queues are defined as one address entry (SAC or DAC).

  A Retry is signaled if these conditions are not true when a transaction is first claimed by the slave interface.
- If the PMW data queue reaches a full state while filling, a disconnect with data is signaled to the master of the transaction on the data phase that fills the queue to a completely full state (no queue bytes remaining).

Error conditions on the initiating bus take precedence over the previous rules. See Section 14.11 for error condition responses.

Memory write transactions are drained from the head of the queue, when the master interface has acquired bus ownership, and transaction ordering and priority have been satisfied (Section 14.7.2). A memory write transaction is considered drained from the queue when the entire amount of data entered on the initiating bus has been accepted by the target. Error conditions resulting in the cancellation of a write transaction (master-abort and target-abort) only flush the transaction at the head of both the address and data queues. All other transactions within the queues are considered still valid. When draining *Memory Write and Invalidate* transactions, the master interface may only complete on cacheline boundaries (regardless of **GNT#** and the master latency timer).

Transactions entering the tail of an empty queue (no previous write transactions reside in queue) are forwarded immediately to the head of the queue. A queue entry (4 bytes for 32-bit data and 8 bytes for 64-bit data) is immediately added to the tail of the queue when drained from the head of the queue on the target bus. As a note, both the upstream and downstream PMW Queues do not operate if the Posting Disable bit is set in the EBCR (see Section 14.15.23). All write operations are delayed and use the DWC Queues.

14.7.1.2 Upstream/Downstream Delayed Read Completion Queues

The Delayed Read Completion Queues (DRC) in the bridge hold the read data obtained during a read completion cycle on the target bus of a Delayed Read Request transaction. The bridge unit has three DRC Queues for each direction of data through the bridge unit. These DRC Queues are different sizes allowing for larger read prefetch sizes when *Memory Read Line* and *Memory Read Multiple* commands are used. I/O Reads and Configuration Reads are constrained to the 4-byte queues on each side of the bridge (see Table 14-14).

Only the data from a delayed read completion cycle is stored in the DRC queue. The address latched from the delayed read request cycle on the initiating bus is stored in the dedicated transaction queues. U_DRC0 through U_DRC2 use U_TRQ0 through U_TRQ2 respectively and D_DRC0 through D_DRC2 use D_TRQ0 through D_TRQ2 respectively.

Transaction queues U_TRQ0, U_TRQ1, and U_TRQ2 have an additional 32-bits of address space for holding the upper 32-bits of an upstream DAC read transaction. Upstream DACs are constrained to U_DRC0, U_DRC1, and U_DRC2.

To maximize read throughput, the larger DRC queues are assigned to the memory read hint commands to maximize the amount of data read on the target bus interface. I/O Reads, Configuration Reads, and non-prefetchable Memory Reads are assigned to the dedicated 4-byte queues. The assignment schemes are in Table 14-15 for the downstream read queues and Table 14-16 for the upstream read queues. Refer to Table 14-12 for downstream read prefetch data sizes and Table 14-13 for upstream read prefetch sizes.

Table 14-15. D_DRC Assignments

PCI Command	Queue Assignment	
	Prefetch	Non-Prefetch
Memory Read Multiple	64 Byte Queue	4 Byte Queue
Memory Read Line	64 Byte Queue	4 Byte Queue
Memory Read	64 Byte Queue	4 Byte Queue
I/O Read	N/A	4 Byte Queue
Configuration Read	N/A	4 Byte Queue

Table 14-16. U_DRC Assignments

PCI Command	Queue Assignment	
	Prefetch	Non-Prefetch
Memory Read Multiple	128 Byte Queue	4 Byte Queue
Memory Read Line	128 Byte Queue	4 Byte Queue
Memory Read	128 Byte Queue	4 Byte Queue
I/O Read	N/A	4 Byte Queue

The exact amount of data read by the master state machine on the target interface depends upon the size of the queue assigned to the request cycle, read command used, prefetchable or non-prefetchable, and how much data the PCI target device delivers. Table 14-12 and Table 14-13 show the amounts of data attempted to be read for the different memory read commands in prefetchable and non-prefetchable address spaces. If an entry in Table 14-12 and Table 14-13 states a prefetch size of 128 bytes and the target PCI device on the target bus disconnects the bridge master interface before reaching the prefetch size, the DRC is complete on the target bus and is allowed to be returned to the initiator. Additional cycles are not initiated to fill the DRC queue to the predefined prefetch data size. PCI error conditions override all prefetch amounts (i.e., a master-abort and target-abort conditions).

Filling the DRC Queues on the target bus only occurs when the DRR cycle in the dedicated Transaction Queue has satisfied priority and transaction ordering. Once the DRC cycle is complete on the target bus, it remains in the DRC Queue until the master on the initiating bus performs a Retry cycle with the same address and command as the initial read request cycle. The DRC transaction will remain in the DRC Queue until retrieved by the master or until the discard timer attached to the queue has expired. Section 14.11.4 explains discard timer operation. When the master does retrieve the data from the DRC queue, the only amount returned is what the master asks for. Any data left in a DRC queue after the master has performed a master completion is invalidated. The bridge unit slave state machine on the initiating bus will only disconnect the read in response to a disconnect on the target bus during the DRC completion cycle of an error condition. See Section 14.11 for all bridge error states.

14.7.1.3 Upstream/Downstream Delayed Write Completion Queue

The upstream and downstream Delayed Write Completion Queues hold the data from a delayed write cycle moving from the initiating bus to the target bus. The upstream and downstream DWC Queues are each 4 bytes in length with one per side of the bridge unit. When a delayed write cycle is claimed by the initiating side of the bridge, the write data is entered into the DWC queue and the initiator is issued a Retry. The address and command information for the delayed write cycle is held in a Transaction Queue. Each DWC Queue has a dedicated Transaction Queue. D_DWC used D_TRQ3 and U_DWC uses U_TRQ3. The DWC queue will hold all *I/O Write* and *Configuration Write* data. In addition, if write posting is disabled all *Memory Write* and *Memory Write and Invalidate* commands will be delayed and will use the initiating bus DWC Queue.

During the write request cycle on the initiating bus, the slave interface of the bridge will claim the delayed write cycle. If the DWC Queue is busy (and the transaction in x_TRQ3 does not match) then the cycle is retried immediately since only one DWC queue exists per side of the bridge and each one is only capable of holding the data from one transaction at a time. If the DWC Queue is empty, the data from the write request cycle is latched into the DWC for delivery to the target bus. The queue is 4 bytes only since all delayed write cycles are a maximum of 32-bits in length.

The DWC Queue is drained on the target bus of the delayed write cycle during the Delayed Write Completion phase after transaction ordering and priority are satisfied. The address from the x_TRQ3 queue is presented and the data for the write is drained from the DWC Queue. Once the data is accepted the DWC Queue is responsible for returning the completion status of the cycle from the target bus back to the initiating bus. This completion is then delivered back to the original master. This completion will be either normal master completion, target disconnect, or one of the error conditions discussed in Section 14.11.

When the retry cycle occurs on the initiating bus, the DWC Queue is used to match the data and byte enables from the retry cycle to the request cycle. This is different than delayed reads which only use the address, byte enables, **REQ64#**, LOCK#(for downstream transactions only), and command to determine if there is a cycle match.

14.7.1.4 Upstream/Downstream Transaction Queues

The upstream and downstream Transaction Queues are used to hold the address and command information from a delayed read or delayed write request cycle. The address within the Transaction Queue is latched on the initiating bus and is presented on the target bus during the delayed completion cycle. Once the delayed completion cycle is enqueued in the completion queue (data for reads, status for writes), the Transaction Queue is used in determining which PCI transaction on the initiating bus is the retried transaction of the original request cycle.

The choice of which Transaction Queue for reads (U_TRQ0 - U_TRQ2 and D_TRQ0 - D_TRQ2) is determined from the information in Section 14.7.1.2. The Transaction Queues for write (U_TRQ3 and D_TRQ3) are dedicated.

The Transaction Queues are loaded during the request cycle on the initiating bus and are only invalidated when a PCI master retries the original request transaction on the initiating interface or when a discard timer attached to the associated data queue expires.

For Dual Address Cycles initiated on the Secondary interface, the Transaction Queues are capable of holding the upper 32-bits of address in a separate set of queues.

14.7.2 Transaction Ordering

Because the bridge can process multiple transactions simultaneously, it must maintain proper ordering to avoid deadlock conditions and improve throughput. The PCI-to-PCI Bridge transaction ordering rules used by the 80303 I/O processor are listed in Table 14-17.

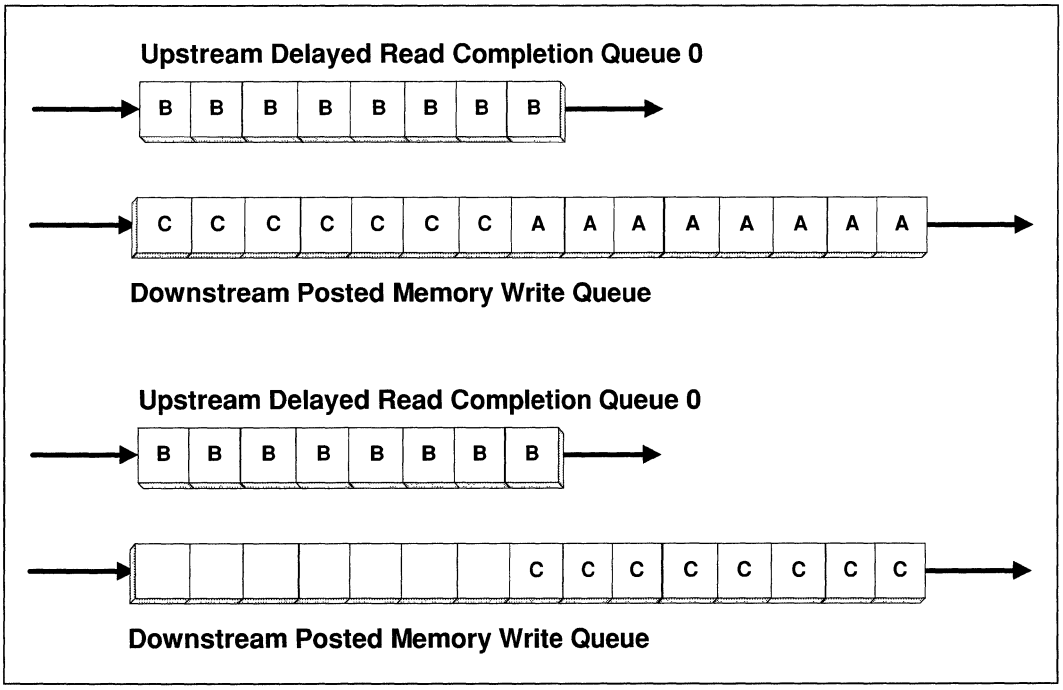
Table 14-17. Bridge Transaction Ordering Rules

Row Pass Column?	Posted Memory Write (PMW)	Delayed Read Request (DRR)	Delayed Write Request (DWR)	Delayed Read Completion (DRC)	Delayed Write Completion (DWC)
Posted Memory Write (PMW)	No	Yes	Yes	Yes	Yes
Delayed Read Request (DRR)	No	Yes	Yes	Yes	Yes
Delayed Write Request (DWR)	No	Yes	No	Yes	Yes
Delayed Read Completion (DRC)	No	Yes	Yes	Yes	Yes
Delayed Write Completion (DWC)	Yes	Yes	Yes	Yes	No

These transaction ordering rules define base line operation for the way data moves in both directions through the PCI-to-PCI Bridge. In Table 14-17 a **NO** response in a box means, based on ordering rules, the current transaction (the row) can not pass the previous transaction (the column) under any circumstance. A **Yes** response in the box means the current transaction is *allowed* to pass the previous transaction, but is not required to do so. This table is derived from Appendix E of the *PCI Local Bus Specification*, Revision 2.2. The rules for *when* a current transaction will pass a previous transaction (based on a **YES** in Table 14-17) are defined in Section 14.7.2.

In the case of bridge posted memory write operations, multiple transactions may exist within the PMW Queue at any point in time. The ordering of these transactions is based on a time stamp basis. Transactions entering the queue are stamped with a relative time in relation to all other transactions moving in a similar direction.

Example 14-1. Downstream Data Path Queue Completion



In Example 14-1., the downstream write data queue (D_PMWD) and an upstream read completion queue (U_DRC0) of the bridge are shown. In this example, transaction A entered the write queue at **Time 0**. Next, the bridge entered read completion data into the upstream read queue at **Time 1** (Transaction B). Finally, before the previous transactions could be cleared, another downstream write, Transaction C, was entered into the downstream write data queue. The ordering in Table 14-17 states that nothing can pass a PMW and therefore Transaction A must complete on the Secondary bus before Transaction B is allowed to complete since an upstream read completion can not pass a downstream posted memory write. Also, Transaction A must complete before Transaction C since a PMW can not pass another PMW. Once Transaction A completes, Transaction C moves to the head of the downstream posted memory write queue. The two transactions at the head of the queues moving data in downstream direction are now Transaction C, a downstream posted memory write, and Transaction B, an upstream read completion. Ordering states that a PMW may pass a read completion. This means that the priority mechanism now takes over to decide which will complete since a **YES** condition from Table 14-17 is now present. In this case, if the PCI master on the Secondary bus acquires the Secondary bus first, Transaction B will complete. If the Secondary interface of the bridge acquires the Secondary bus first, Transaction C will complete. Note that ordering enforced the completion of Transaction A but priority dictated the completion of Transactions B and C.

The first action performed to determine which transaction is allowed to proceed (either upstream or downstream) is to apply the rules of ordering as defined in Table 14-17. Any box marked **No** must be satisfied first. For example if a downstream read request is in the D_TRQx queue and it was latched *after* the data in the D_PMW arrived, then ordering states that a Read Request may not pass a Posted Memory Write; therefore the Posted Memory Write must be cleared out of the D_PMW before the Read Request is attempted on the Secondary bus. Once transaction ordering is satisfied, the boxes marked **Yes** are now resolved based on the priority mechanism in Section 14.7.2.

Table 14-18 summarizes the transaction ordering tables in relation to token assignment of the priority mechanism. This table is read as follow:

1. As the transaction reaches the respective queue head, the question in column two is asked.
2. Based on the answer in column three, either a token is assigned or no token is assigned signifying that transaction ordering must first be satisfied. Note that if the answer is Yes/No in column three, the Action in column four is for either a Yes or a No.

Table 14-18. Bridge Transaction Ordering and Priority Mechanism

Transaction at Head of Queue	Question	Answer	Action
Posted Memory Write in PMWD	Is there a DRR in a TRQ0:2 queue with an earlier time stamp?	Yes/No	Assign Token
	Is there a DRC in DRC0:2 with an earlier time stamp?	Yes/No	Assign Token
	Is there a DWR in TRQ3 with an earlier time stamp?	Yes/No	Assign Token
	Is there a DWC in DWC queue with an earlier time stamp?	Yes/No	Assign Token
Delayed Read Request in TRQ0:2	Is there a PMW in the PMWD queue with an earlier time stamp?	Yes	Do Not Assign Token Allow previous Transaction to Complete
	Is there a PMW in the PMWD queue with an earlier time stamp?	No	Assign Token
	Is there a DRR in a TRQ0:2 queue with an earlier time stamp?	Yes/No	Assign Token
	Is there a DRC in DRC0:2 with an earlier time stamp?	Yes/No	Assign Token
	Is there a DWR in TRQ3 with an earlier time stamp?	Yes/No	Assign Token
	Is there a DWC in DWC queue with an earlier time stamp?	Yes/No	Assign Token
Delayed Write Request in TRQ3	Is there a PMW in the PMWD queue with an earlier time stamp?	Yes	Do Not Assign Token Allow previous Transaction to Complete
	Is there a PMW in the PMWD queue with an earlier time stamp?	No	Assign Token
	Is there a DRR in a TRQ0:2 queue with an earlier time stamp?	Yes/No	Assign Token
	Is there a DRC in DRC0:2 with an earlier time stamp?	Yes/No	Assign Token
	Is there a DWC in DWC queue with an earlier time stamp?	Yes/No	Assign Token
Delayed Read Completion in DRC0:2	Is there a PMW in the PMWD queue with an earlier time stamp?	Yes	Do Not Assign Token Allow previous Transaction to Complete
	Is there a PMW in the PMWD queue with an earlier time stamp?	No	Assign Token
	Is there a DRR in a TRQ0:2 queue with an earlier time stamp?	Yes/No	Assign Token
	Is there a DRC in DRC0:2 with an earlier time stamp?	Yes/No	Assign Token
	Is there a DWR in TRQ3 with an earlier time stamp?	Yes/No	Assign Token
	Is there a DWC in DWC queue with an earlier time stamp?	Yes/No	Assign Token
Delayed Write Completion in DWC	Is there a PMW in the PMWD queue with an earlier time stamp?	Yes/No	Assign Token
	Is there a DRR in a TRQ0:2 queue with an earlier time stamp?	Yes/No	Assign Token
	Is there a DRC in DRC0:2 with an earlier time stamp?	Yes/No	Assign Token
	Is there a DWR in TRQ3 with an earlier time stamp?	Yes/No	Assign Token

14.8 Bridge Data Flow

The bridge allows transactions to cross both PCI buses through the 80303 I/O processor. PCI transactions initiated on the Primary PCI bus and targeted at an agent on the Secondary PCI bus are referred to as *downstream* transactions and PCI transactions initiated on the Secondary PCI bus and targeted at an agent on the Primary PCI bus are referred to as *upstream* transactions.

Upstream and downstream bridge transactions are best described by the data flows used on the initiating and target bus during read and write operations. The following sections describe:

- Delayed Read transactions
- Delayed Write transactions
- Posted Write transactions

Separate upstream and downstream transactions are not shown but have identical data flows except that the references to initiating interface and target interface are reversed.

14.8.1 Delayed Read Transaction

A delayed read transaction is initiated by a PCI master on the initiating PCI bus and is targeted at a PCI agent on the target PCI bus. The read transaction is propagated through the bridge and read data is returned through a Delayed Read Completion Queue (DRC).

All read transactions are processed as delayed read transactions. The PCI slave interface on the initiating bus of the bridge claims the read transaction and store the read address and control information in a bridge Transaction Queue. This read request is then forwarded to the target bus. The target bus master interface then performs the read from a PCI target and store returning read data in a Delayed Read Completion Queue. The original PCI master on the initiating bus continuously retries the read transaction until slave interface on initiating bus claims the transaction and returns read data present in the DRC Queue. The data flow for a delayed read transaction is summarized as followings:

- The Bridge claims the PCI read transaction if the PCI address is within the address window defined by a Base/Limit register pair and a Transaction Queue is available to retain the address/control information to forward to the target bus.
- If there is currently an available Transaction Queue, then latch the PCI address into Transaction Queue and then signal a Retry to the initiator.
- If an address parity error is detected, allow transaction to master-abort and assert **SERR#**. This assumes parity checking is enabled and **SERR#** assertion is enabled (Note: **SERR#** is not asserted on Secondary bus interface, only on the Primary bus interface, see Section 14.11.1).
- If transaction inside an address window and a Transaction Queue not available or inside an address window, a cycle match occurs, but read data not ready (!DRC_Ready), then retry transaction.
- If transaction is inside an address window, a cycle match occurs, read data is ready in the DRC (DRC_Ready), then return the read data to the master device. Data is returned 64-bits wide if the master used **REQ64#** during the request or 32-bits at a time if **REQ64#** was not asserted.
- Once read data has started to be driven onto the initiating PCI bus from a DRC Queue, it will continue to be driven until one of the following is true:
  - — The initiator completes the PCI transaction, master-completion.
  - — The DRC Queue becomes empty.
  - — A target-abort condition is driven out from the DRC Queue.
- If a data parity error is detected by the master and **PERR#** is asserted, set the appropriate error response bits (if enabled, see Section 14.11.2).

The target PCI interface will initiate the read transaction with the PCI address and command used on the initiating bus and then put the return data into a DRC Queue to return it to the initiating PCI bus. The data flow is summarized in the following statements:

- The bridge PCI master interface on the target bus will request the PCI bus when an read request address is written to a Transaction Queue.
- Once the bus is granted to the bridge interface for the read transaction, the target bus master interface will initiate a read transaction with the same address and command used on the initiating interface. If the read is a memory read (and a 64-bit bus is enabled) the bridge will assert **REQ64#** attempting to use 64-bit data phases. If the bus is not 64-bit enabled or it is an I/O or Configuration Read, **REQ64#** is not asserted.
- If the transaction is claimed and retried, the bus interface will re-attempt the transaction. If the master interface receives a master-abort, a master-abort condition is loaded into the DRC Queue for return to the master on the initiating bus. The condition loaded is dependent on the Master Abort bit in the BCR. See Section 14.10.1.4 for details.
- Once the read transaction is claimed, the bridge master interface will read data from the PCI target. If **ACK64#** is asserted data is read 64-bits at a time. If **ACK64#** is not asserted, data is read 32-bits at a time.
- The master interface will continue to read data until one of the following is true:
  - — The prefetch amount of DWORDs specified in Table 14-12 is reached.
  - — The target disconnects the transaction.
  - — A PCI time-out occurs.
  - — The target performs a target-abort (this condition is returned and loaded into the DRC Queue).
  - — The bridge master interface encounters a 4 Kbyte boundary.
- If parity checking is enabled and a data parity error is detected, the master interface will assert **PERR#** and continue reading data until one of the previous conditions is true.

14.8.2 Delayed Write Transaction

A delayed write transaction is initiated by an agent on the initiating PCI bus and is targeted at a PCI agent on the target PCI buses. I/O and Configuration writes as well as memory writes with posting disabled are processed as delayed writes. The delayed write request address is propagated from the initiating PCI bus to the target PCI bus through a Transaction Queue. The delayed write request data is propagated to the target bus in a Delayed Write Completion (DWC) Queue. Completion status is returned to the master on the initiating bus during a retry cycle.

The data flow for a delayed write transaction on the PCI bus is summarized in the following statements:

- The Bridge claims the PCI write transaction if the PCI address is within an address window defined by a Base/Limit register pair, the Delayed Write Completion Queue is available (if DWC is free, the associated Transaction Queue is free by architectural definition), and it is a delayed write PCI transaction (I/O writes, Configuration Writes, Memory Writes with posting disabled).
- The address is written to the Transaction Queue in anticipation of capturing the data for the delayed write request.
- If an address parity error is detected (if enabled), the Transaction Queue is cleared and the transaction is allowed to master-abort. **SERR#** is asserted if enabled and on the Primary bus.
- The assertion of **STOP#**, to Retry the transactions, by the bridge unit is delayed until the assertion of **PAR** by the master so parity can be calculated. If parity is good, the transaction is retried and the delayed write request can proceed to the target interface.
- If a data parity error is detected and parity response is enabled, the transaction is claimed (not retried) with the assertion of **TRDY#** and **PERR#** is asserted if enabled. The delayed write request is cleared from the queues and is not forwarded to the target interface.
- If subsequent transactions from the master on the initiating bus are inside the address window, the DWC is not cleared, and there is no cycle match or the write has not completed on the target interface (!Write_Complete), the transaction is retried immediately.
- If subsequent transactions are inside the address window, the DWC is not cleared, there is a cycle match, and the transaction has already completed on the target bus, then the bridge unit slave interface will compare the write data from the master with the write data that was used in the delayed request cycle. If the data matches (with no parity error detected) then the status seen on the target bus is returned and a disconnect is performed. Note that the status returned is exactly what was seen from the target on the target bus i.e. target-abort, parity error or normal completion.
- If a parity error is detected from the data being written from the initiating master, the bridge slave interface will claim the transaction and assert **PERR#** if enabled. Since the data has not matched with the data from the write request cycle, the transaction remains enqueued.

The PCI master target interface is responsible for issuing the write completion transaction to a PCI agent using the data in the DWC Queue and the address/command from the Transaction Queue. The data flow for a delayed write transaction on the target bus is summarized in the following statements:

- The master interface on the target PCI bus will request the PCI bus when after address and data from the request cycle have been received and ordering has been satisfied (see Section 14.7.2). Once the bus is granted, the target PCI interface will write the PCI address to the PCI bus and wait for the transaction to be claimed.
- If an address parity error is detected by an agent on the bus, **SERR#** may be asserted.
- If the transaction on the bus receives a master-abort, the appropriate master abort condition is loaded into the DWC queue for return to the PCI master on the initiating bus. Refer to Section 14.10.1.4 for master-abort conditions.
- If the PCI target signals a Retry or Disconnect, the master interface will return to idle. If the PCI target signals claims the transaction, the 32-bit data is transferred to the target in a single data phase (all delayed writes are performed as 32-bit operations.). The master/target response is captured for return to the master on the initiating bus. The following conditions are possible:
  - — Master Completion - Normal Completion
  - — Target Abort
  - — Data Parity Error

  The Write_Complete flag is set indicating to the initiating interface that the write completion cycle is complete.

14.8.3 Posted Write Transaction

A posted write transaction is initiated by a PCI master on the Primary PCI bus and is targeted at a PCI agent on the Secondary PCI bus. The address and data from the master on the initiating bus is written to the Posted Memory Write (PMW) Queue for delivery to the target bus. Posted memory write operations complete on the initiating bus before they complete on the target bus.

The data flow for a posted write transaction on the PCI bus is summarized in the following statements:

- The Bridge claims the PCI write transaction if the PCI address within the address window defined by a Base/Limit register pair. If the PMW Queue is full (defined as not enough buffer space to hold the address and at least one data phase) the transaction is retried. If the memory write transaction is inside the address window and the PMW Queue is not full (and posting is enabled) the transaction is claimed and the address is entered into the PMW Queue.
- If an address parity error is detected from the master. **SERR#** is asserted (if enabled and the initiating bus is the Primary bus) and the transaction is allowed to master-abort.
- Once the PCI address is in the PMW Queue, the PCI interface can start accepting write data and store it in the PMW Queue. If **REQ64#** was asserted by the master, 64-bit data is received. If **REQ64#** was not asserted by the master, 32-bit data is received. The PCI interface will continue accepting write data until one of the following is true:
  - — The initiator completes the transaction - master completion.
  - — The PMW Queue becomes full. In this case, the PCI slave interface will signal a disconnect to the master and return to idle.
- If a data parity error is detected, the slave interface will assert **PERR#** (if enabled). No other action is taken and the reception of write data continues.

The PCI interface is responsible for completing the posted write transaction to a PCI agent using the address/data in the PMW Queue. The data flow for the posted write transaction on the target PCI bus is summarized in the following statements:

- The master interface on the target PCI bus will request the PCI bus when posted write transaction address is at the head of the PMW Queue and transaction ordering has been satisfied (see Section 14.7.2). Once the bus is granted, the target PCI interface will write the PCI address from the PMW Queue to the PCI bus and wait for the transaction to be claimed.
- If an address parity error is detected by the assertion of **SERR#** on the target interface, the error is recorded. The action taken by the interface depends on the targets response to the parity error. If a master abort is used, see the following section.
- If a master-abort is signaled, the master-abort condition is used. This could be either flushing the write data, asserting **P_SERR#**, or signaling a disconnect. Refer to Section 14.10.1.4 for details.
- Once the PCI write transaction is claimed, the PCI interface will transfer data from the PMW Queue to the PCI bus. If **ACK64#** is asserted, data is transferred 64-bits at a time. If **ACK64#** is deasserted, data is transferred 32-bits at a time. Data is transferred to the bus until one of the following is true:
  - — The PCI target signals Disconnect.
  - — The PCI target signals a Target-Abort. In this case, the PMW Queue is flushed and the transaction is aborted.
  - — The PMW Queue become empty signifying that the transaction is finished. This results in a master completion.
- If the transfer is an MWI, the bridge target bus interface may have to insert master waitstates to guarantee to transfer of an entire cacheline of data (defined by the value in the CLS register).
- If a data parity error is detected by the target (**PERR#** driven) and it is not a result of an error propagated from the initiating interface, the bridge master interface logs the error and asserts **P_SERR#**, if enabled, on the Primary bus. Refer to Section 14.11.2.3 for details.

14.9 Exclusive Access

The bridge supports the PCI exclusive access mechanism using the **P_LOCK#** signal for downstream accesses only. The bridge ignores the **S_LOCK#** signal for upstream accesses.

Note: PCI Masters on the Secondary bus should not attempt to perform upstream locked transactions. Doing so may cause the PCI system to enter a state which prevents the Secondary locking master from completing the locked request enqueued in the bridge, resulting in a system livelock.

The bridge establishes itself as a locked target during a Delayed Read Request on the Primary PCI bus when **P_LOCK#** is deasserted in the address phase. When the bridge detects a downstream locked read request, a bridge lock sequence is started. This sequence is a series of state transitions which, when completed, will leave the bridge "locked" from all masters except for the master which owns the **P_LOCK#** resource.

These states, and their transition criteria, are described in detail in Table 14-19.

Table 14-19. LOCK# Operation State Definitions

State	Primary Interface		Secondary Interface		Transition
	Operation	Definition	Operation	Definition	Transition
Unlocked	Unrestricted	Accept all transactions on interface'	Unrestricted	Accept all transactions on interface	Move to **Secondary Locking** when locked DRR received on Primary interface
Secondary Locking	Restricted	All new downstream transactions retried.	Unrestricted	Accept all transactions on interface	Move to **Primary Locking** when bridge has finished mastering the locked request on the Secondary interface. If an abort occurs while mastering this transaction the bridge will transition to directly to the **Unlocking** state.
Primary Locking	Restricted	All new downstream transactions retried.	Restricted	All new upstream transactions retried	Upon completion of the locked request the bridge transitions to **Locked** state. If the DRCs discard timer expires before the transaction is completed, the bridge will transition directly to the **Unlocking** state.
Locked	Restricted	Only accepts downstream transactions from lock master	Restricted	All upstream transactions retried	Moves to Unlocking when Lock master on Primary interface releases **P_LOCK#** signal. If an abort occurs while mastering a locked transaction, the bridge will release the **S_LOCK#** signal and transition to the **Unlocking** state.
Unlocking	Restricted	All new downstream transactions retried	Restricted	All upstream transactions retried	Moves to the **Unlocked** state as soon as the bridge has completely emptied all of its transaction queues. Note: In certain error situations this may require that the discard timers be used to remove transactions which cannot otherwise be completed.

If an abort (see Section 14.9.1) occurs, either while locked or while attempting to lock, or the discard timer associated with a locked transaction expires, the bridge will transition to the **Unlocking** state. If this occurs due to an abort, any remaining enqueued requests are forwarded as unlocked transactions. This ensures that the bridge makes every attempt to pass the lost lock status back to the initiating master before accepting more transactions.

Downstream lock transactions must begin with a delayed read request (DRR) from the lock master. If a PCI master tries to establish a lock with a write transaction using any PCI write command, the Primary interface accepts the write transaction as an unlocked transaction and forwards it to the Secondary interface without asserting **S_LOCK#**.

Systems which implement multiple 80303 I/O processors must be aware that there are situations which can cause lockup conditions if downstream masters are allowed to generate locked transactions. These lockup conditions can arise due to interactions between the PCI transaction ordering rules and the requirement to restrict transaction forwarding through the bridge while locking or locked. This should not present a problem as the only master which typically generates locked transactions is the system host, located on the uppermost PCI bus.

Several other precautions are worth noting for systems that forward transactions to downstream targets. Since the PCI specification requires that the lock resource be released when the locked master transaction aborts, situations may arise in which the bridge loses its lock on the Secondary bus before the initial locked master is ready to release its lock. The system designer must ensure that the bridge is configured to allow error or abort status to be reflected upstream to handle these situations, otherwise data integrity may be compromised in the shared resource.

Finally, since the bridge is required to transition through the **Unlocking** state before returning to **Unlocked**, the bridge must be able to empty all of its transaction queues. Discard timers should never be disabled, since timing out may be the only means of emptying the bridge in the event a master is unable to complete an enqueued transaction.

14.9.1 Secondary Interface Error Handling

The *PCI Local Bus Specification*, Revision 2.2 states that a master which has lost its bus lock, must deassert LOCK#. An issue arises because the specification also states that a master must retry (with LOCK#) all locked outstanding transactions already attempted on the bus.

The Secondary interface of the bridge will use the following rules when an abort condition (master or target) is encountered from a target on the Secondary interface:

- Deassert **S_LOCK#** as masters are required to do when an abort condition is encountered.
- Put the bridge into the **Unlocked** state (see Table 14-19).
- Convert all remaining requests within the bridge into non-locked transactions and let them complete (locked completions will complete as locked). This includes any outstanding delayed requests.

The converted transactions (on the Secondary interface) may be accepted as new transactions by any downstream bridges. The original master on the initiating bus will release **P_LOCK#** when the abort condition is reported. This may leave delayed locked requests within downstream bridge queues which can only be removed by the action of the Discard Timers. In the case of posted write transactions, it is up to software to use **P_SERR#** to determine that the transaction aborted on the target interface.

14.10 PCI Transaction Termination

PCI creates a mechanism for both PCI initiators and PCI targets to prematurely terminate a transaction. As a PCI master (initiator), a device can terminate a transaction when it is complete or when an error condition occurs. As a slave (target), a PCI device can only terminate when an error condition occurs. While transaction termination can be initiated by either a master or a target, ultimately it is up to the master to bring a PCI transaction to an orderly conclusion. As a bridge device, the target interface is responsible for this on the target bus. A PCI transaction is considered concluded when both **FRAME#** and **IRDY#** are both deasserted indicating a PCI IDLE cycle.

14.10.1 Termination as a Master (Initiator)

The target interface of the bridge unit, acting as a PCI master, will terminate a PCI transaction under the different situations described in the following sections.

14.10.1.1 Completion

The target interface will complete the transaction in response to a completion on the initiating interface. Completion termination occurs when the initiator bus has **FRAME#** and **IRDY#** deasserted. **FRAME#** will always be deasserted during the second to last data transfer of a transaction. Refer to the *PCI Local Bus Specification*, Revision 2.2.

14.10.1.2 Time-out

A time-out occurs when the **GNT#** signal is deasserted on the target bus and the associated master latency timer has expired (see Section 14.6.2.1). A normal termination will occur on the target interface (except when Memory Write and Invalidate is in progress). See the next section for the Memory Write and Invalidate case.

14.10.1.3 Time-out During Memory Write and Invalidate

If target interface time-out occurs during a Memory Write and Invalidate transaction, the bridge will retain ownership until an entire cacheline has been transferred from the PMW Queues. Refer to the *PCI Local Bus Specification*, Revision 2.2.

14.10.1.4 Master-Abort

A Master-Abort is used when no target responds with a **DEVSEL#** within 5 clocks after the assertion of **FRAME#.** The 80303 I/O processor bridge unit has two mechanisms for handling Master-Aborts. The mechanism depends on the Master Abort Mode bit in the Bridge Control Register (BCR).

When the Master Abort Mode bit is cleared, the bridge is operating in a PC compatibility mode. When a read transaction crosses the bridge in this mode and the target interface signals a Master-Abort, the bridge returns all 1s (32-bits wide or 64-bits wide based on the size of the initiating PCI bus) to the initiator during the repeated transaction and terminates normally (with **TRDY#**) on the initiating interface. When a write transaction crosses the bridge in this mode and the target interface signals a Master-Abort, the bridge completes the transaction normally on the initiating interface and discards the write data on the target interface. In both cases, the bridge sets the Received Master Abort bit in the Primary Status Register (PSR) if the Master-Abort occurred on the Primary interface and in the Secondary Status Register (SSR) if the Master-Abort occurred on the Secondary interface.

When the Master Abort Mode bit is set, the bridge signals a Master-Abort to the initiator of a delayed read or write transaction when that transaction causes a Master-Abort on the target bus. The bridge sets the corresponding Received Master Abort bit as in the previous case. If the transaction that caused the Master-Abort on the target interface was a posted write transaction, the bridge asserts **P_SERR#** on the Primary interface (if enabled). The bridge terminates the posted write transaction on the initiating interface with a disconnect with or without data. The Received Master Abort bit is set in the appropriate status register corresponding to the Master-Abort interface and the Signaled System Error bit in the PSR.

A Master-Abort is not signaled during a Special Cycle transaction from either interface.

14.10.2 Termination as a Slave (Target)

The method for a target termination on either PCI interface is the assertion of the **STOP#** signal. A PCI target asserts **STOP#** to request that the master terminate a transaction. The target will hold **STOP#** asserted until the master deasserts **FRAME#. IRDY#** and **TRDY#** are independent of target termination so data may or may not be transferred. The only rule is that if **STOP#** is asserted when **TRDY#** is deasserted, the master will not wait for the final data transfer. The following sections summarize the bridge actions as a PCI target for termination situations. See the *PCI Local Bus Specification*, Revision 2.2 for details.

14.10.2.1 Retry

Retry refers to a termination request to the initiator where data has not been transferred. The bridge uses the Retry mechanism when the bridge:

- is unable to provide resources for propagating the transaction to its destination.
- accepts a delayed request.
- receives a delayed request and it does not match any delayed completions held by the bridge.
- is locked and the initiator does not own the **LOCK#** signal.

A Retry is signaled when **STOP#** and **DEVSEL#** are asserted and **TRDY#** is deasserted on the initiating interface.

14.10.2.2 Disconnect

A Disconnect is used when the initiating interface is unable to respond to the initiator due to a condition like the posting buffer has become full. A Disconnect is used when data has been already been transferred to the bridge. Refer to the PCI Local Bus Specification for details on Disconnect.

A Disconnect is signaled using two sequences. When **STOP#**, **TRDY#**, and **DEVSEL#** are all asserted, it indicates that this transfer is the last and at least one data word is transferred. When **STOP#** and **DEVSEL#** are asserted and **TRDY#** is deasserted after previous data transfers, it indicates that the most recent transfer was the last.

14.10.2.3 Target-Abort

A Target-Abort differs from a Retry or a Disconnect when **STOP#** is asserted and **DEVSEL#** has been deasserted.

During all transactions crossing the bridge, except posted writes, the bridge will signal a Target-Abort to the initiator on the initiating bus when a Target-Abort is received by the bridge on the target bus. The bridge will set the Target Abort (master) bit in the target bus status register (PSR or SSR) and the Target Abort (target) bit in the initiating bus status register. (An exception to this rule can occur in the case where a target inserts data-to-data wait states after the initial Qword of data. If the bridge is forced to disconnect with data on the initiating side, due to the fact that the bridge does not insert data-to-data wait states as a slave, and a target-abort is then signalled by the target after the bridge has disconnected with the master, the target-abort will not be reflected back to the master and the Target Abort (target) bit in the initiating bus status register will not be set.)

If the bridge detects a Target-Abort during a posted write transaction on the target bus and the write is still in progress on the initiating bus, the bridge will signal a Target-Abort to the initiator on the initiating bus. The bridge will set the Target Abort (master) bit in the target bus status register (PSR or SSR) and the Target Abort (target) bit in the initiating bus status register. The error must be signaled on the originating bus in the same data phase in which it occurred on the destination bus.

If the posted write transaction is complete on the initiating interface, the bridge will assert **P_SERR#** (if enabled) on the Primary interface indicating a system error. The bridge will also set the Target Abort (master) bit in the target bus status register (PSR or SSR).

14.11 Error Conditions

PCI provides an extensive error reporting mechanism. The PCI-to-PCI bridge implements parity generation and parity error detection on both the Primary and Secondary PCI interfaces and passes that information to the Primary interface. This enables the parity error recovery mechanisms outlined on the *PCI Local Bus Specification*, Revision 2.2 without special considerations for a bridge.

The following sections detail the bridge response to parity errors on the Primary and Secondary PCI interfaces.

14.11.1 Address Parity Errors

The bridge must detect and report address parity errors for transactions on both interfaces. When the bridge, as a device on the initiating interface, detects an address parity error before claiming a cycle, the bridge will not claim the cycle (not assert **DEVSEL#**) and allow the transaction to terminate with the Master-Abort mechanism.

When the bridge detects an address parity error during a transaction the Primary and Secondary interfaces will handle the error in different manners.

14.11.1.1 Address Parity Errors on Primary Interface

If an address parity error occurs on the Primary interface of the bridge unit, the 80303 I/O processor performs the following actions based on the constraints specified:

- If the Primary Parity Error Response Enable bit in the PCR is set, the Primary bridge interface will *not* claim the transaction by *not* asserting **P_DEVSEL#**, allowing a master abort to occur.

  If the Primary Parity Error Response Enable bit in the PCR is cleared, the Primary bridge interface takes normal action and allows the transaction to proceed (claim the transaction if the address is within the bridge address space).

- Assert **P_SERR#** on the Primary interface if the **SERR#** Enable bit and Primary Parity Error Response Enable bit in the PCR are both set.

- Set the Signaled System Error bit in the PSR if the **SERR#** Enable bit and Primary Parity Error Response Enable bit in the PCR are both set. If the Signaled System Error bit in the PSR is set and the **P_SERR#** Asserted Interrupt Mask is clear in the SDER, set the **P_SERR#** Asserted bit in the PBISR

- Set the Detected Parity Error bit in the PSR. If the Detected Parity Error bit in the PSR is set and the Primary Detected Parity Error Interrupt Mask bit in the SDER is clear, set Detected Parity Error bit in the PBISR.

14.11.1.2 Address Parity Errors on Secondary Interface

If an address parity error occurs on the Secondary interface of the bridge unit, the 80303 I/O processor performs the following actions based on the constraints specified:

- If the Secondary Parity Error Response Enable bit in the Bridge Control Register (BCR) is set, the Secondary bridge interface will *not* claim the transaction by *not* asserting **S_DEVSEL#**, allowing a master abort to occur.

  If the Secondary Parity Error Response Enable bit in the BCR is cleared, the Secondary bridge interface takes normal action and allows the transaction to proceed.

- Assert **P_SERR#** on the Primary interface if the **SERR#** Enable bit in the PCR is set and Secondary Parity Error Response Enable bit and the Secondary **SERR#** enable bit in the BCR are set.

- Set the Signaled System Error bit in the PSR if the **SERR#** Enable bit in the PCR is set and Secondary Parity Error Response Enable bit in the BCR is set. If the Signaled System Error bit in the PSR is set and the **P_SERR#** Asserted Interrupt Mask is clear in the SDER, set the **P_SERR#** Asserted bit in the PBISR

- Set the Detected Parity Error bit in the SSR. If the Detected Parity Error bit in the SSR is set and the Secondary Detected Parity Error Interrupt Mask bit in the SDER is clear, set Detected Parity Error bit in the SBISR.

While forwarding a DAC cycle upstream, the bridge may detect an address parity error in any of the four different parts of the DAC address phase in which parity information is encoded. If an address parity error is detected in these cases, the bridge will forward the DAC using bad address parity for all possible parts of the forwarded transaction. This eliminates the possibility of an address parity being filtered out by the bridge in the event it is converted from a 64-bit transaction to a 32-bit transaction.

14.11.2 Data Parity Errors

When the bridge unit detects a data parity error, the bad data and bad parity will be passed to the opposite interface whenever possible. This will enable the parity error recovery mechanisms outlined in the *PCI Local Bus Specification*, Revision 2.2 without special consideration for the bridge in the datapath.

14.11.2.1 Read Data Parity

When a data parity error is detected during a read transaction that crosses the bridge unit, it will assert **PERR#** on the target interface. The bridge will pass the bad data and the bad parity to the initiating interface and bus where the initiator will also detect the bad parity and data and assert **PERR#** on the initiating bus. The bridge will set the Detected Parity Error bit and set the Data Parity Detected bit (when enabled) in the PSR if the Primary interface is the target bus or the SSR if the Secondary interface is the target bus. When data parity is detected by the master on the initiating bus, it will assert **PERR#**. No other action is taken by the bridge unit.

Specifically for downstream reads (initiated by a master on the Primary bus interface), the 80303 I/O processor performs the following actions with the given constraints:

- **S_PERR#** is asserted two clock cycles following the data phase in which the data parity error is detected on the Secondary bus. This is only done if the Secondary Parity Error Response Enable bit in the Bridge Control Register (BCR) is set.
- The Data Parity Detected bit in the Secondary Status Register (SSR) is set if the Secondary Parity Error Response Enable bit in the BCR is set. If the Secondary PCI Master Parity Error Interrupt Mask bit in the SDER is clear, set the PCI Master Parity Error bit in the SBISR.
- The Detected Parity Error bit in the SSR is set. If the Secondary Detected Parity Error Interrupt Mask bit is clear in the SDER, set the Detected Parity Error bit in the SBISR.
- The data and the bad parity are stored in a DRC queue and returned to the master on the Primary bus during Delayed Read Completion cycle. If the data word with the bad parity is not read from DRC queue by the initiator (i.e., delayed cycle read 32 bytes with an error in byte 30 and the master only wanted 16 bytes) due to the prefetch algorithm (see Section 14.6.5), the data is discarded when the queue is invalidated and no other action is taken.

Specifically for upstream reads (initiated by a master on the Secondary bus interface), the 80303 I/O processor performs the following actions with the given constraints:

- **P_PERR#** is asserted two clocks cycles following the data phase in which the data parity error is detected on the Primary bus. This is only done if the Primary Parity Error Response Enable bit in the Primary Command Register (PCR) is set.
- The Data Parity Detected bit in the Primary Status Register (PSR) is set if the Primary Parity Error Response Enable bit in the PCR is set. If the Primary PCI Master Parity Error Interrupt Mask bit in the SDER is clear, set the PCI Master Parity Error bit in the PBISR.
- The Detected Parity Error bit in the PSR is set. If the Primary Detected Parity Error Interrupt Mask bit is clear in the SDER, set the Detected Parity Error bit in the PBISR.
- The data and the bad parity are stored in a DRC queue and returned to the master on the Secondary bus during Delayed Read Completion cycle. If the data word with the bad parity is not read from DRC queue by the initiator (i.e., delayed cycle read 32 bytes with an error in byte 30 and the master only wanted 16 bytes) due to prefetch algorithm (see Section 14.6.5), the data is discarded when the queue is invalidated and no other action is taken.

In both cases, the initiator of the Delayed Read transaction is responsible for asserting **PERR#** on the initiating bus (if enabled) in response to bridge unit delivering data along with the bad parity.

14.11.2.2 Delayed Write Data Parity

To allow for correct data parity calculations for delayed write transactions, the bridge will delay the assertion of **STOP#** (signalling a Retry) until **PAR** is driven by the master. A parity error during a delayed write transaction can occur in any of the following parts of the transactions:

- During the Delayed Write Request cycle on the initiating bus when the transaction is enqueued by the bridge unit.
- During the Delayed Write Completion cycle on the target bus when the write data is delivered to the target and write status is capture for delivery to the initiator
- During the Delayed Write Completion cycle on the initiating bus when write status is to be delivered to the initiator who has retried the transaction.

Depending on where and when the parity error occurs, different responses are required.

The 80303 I/O processor Primary bridge interface has the following responses to a delayed write parity error for downstream transactions during Delayed Write Request cycles on the initiating bus with the given constraints:

- If the Primary Parity Error Response bit in the PCR is set, the Primary bridge interface asserts **P_TRDY#** (disconnects with data) and two clock cycles later asserts **P_PERR#** notifying the initiator of the parity error. The delayed write cycle in not enqueued and not forwarded to the Secondary interface.

  The Detected Parity Error bit is set in the Primary Status Register (PSR) only if data has been transferred. This scenario would occur if a request is seen with bad parity. In this case the request is immediately completed and discarded. Because of the completion, data has been transferred on the initiating interface. If the Primary Detected Parity Error Interrupt Mask bit is clear in the SDER, set the Detected Parity Error bit in the PBISR.

- If the Primary Parity Error Response bit in the PCR is cleared, the Primary bridge interface retries the transaction by asserting **P_STOP#** and enqueues the Delayed Write Request cycle to be forwarded to the Secondary bridge interface. **P_PERR#** is not asserted.

On the Secondary bridge interface, the following responses to a delayed write parity error for upstream transactions during Delayed Write Request cycles on the initiating bus with the given constraints:

- If the Secondary Parity Error Response bit in the BCR is set, the Secondary bridge interface asserts **S_TRDY#** (disconnecting with data) and two clock cycles later asserts **S_PERR#** notifying the initiator of the parity error. The delayed write cycle in not enqueued and not forwarded to the Primary interface.

  The Detected Parity Error bit is set in the Secondary Status Register (SSR) only if data has been transferred. This scenario would occur if a request is seen with bad parity. In this case the request is immediately completed and discarded. Because of the completion, data has been transferred on the initiating interface. If the Secondary Detected Parity Error Interrupt Mask bit is clear in the SDER, set the Detected Parity Error bit in the SBISR.

- If the Secondary Parity Error Response bit in the BCR is cleared, the Secondary bridge interface retries the transaction by asserting **S_STOP#** and enqueues the Delayed Write Request cycle to be forwarded to the Primary bridge interface. **S_PERR#** is not asserted.

During a downstream Write Completion Cycle on the target bus when the bridge is trying to deliver enqueued write data, the Secondary bridge interface has the following actions with the given constraints when **S_PERR#** is detected during the transaction:

- The Data Parity Error Detected bit is set in the Secondary Status Register (SSR) if the Secondary Parity Error Response Bit is set in the BCR. If the Data Parity Error Detected bit in the SSR is set and the Secondary PCI Master Parity Error Interrupt Mask in the SDER is clear, set the PCI Master Parity Error bit in the SBISR.
- The Secondary interface of the bridge captures the error completion status for delivery back to the initiator on the Primary interface.

During a upstream Write Completion Cycle on the target bus when the bridge is trying to deliver enqueued write data, the Primary bridge interface has the following actions with the given constraints when **P_PERR#** is detected during the transaction:

- The Data Parity Error Detected bit is set in the Primary Status Register (PSR) if the Primary Parity Error Response Bit is set in the PCR. If the Data Parity Error Detected bit in the PSR is set and the Primary PCI Master Parity Error Interrupt Mask in the SDER is clear, set the PCI Master Parity Error bit in the PBISR.
- The Primary interface of the bridge captures the error completion status for delivery back to the initiator on the Secondary interface.

For the original write transaction to be completed, the initiator will retry the transaction on the initiating bus and the bridge will return the status from the target bus, completing the transaction. A data parity error can occur in this scenario on the initiating bus that was not detected during the write completion cycle on the target bus or a parity error can occur in response to a parity error that did occur during the write completion cycle on the target bus (contained with the status returned by the bridge).

For downstream delayed completion transaction on the initiating bus where a data parity error occurs that did not occur on the target bus (i.e. status being returned is normal completion) the Primary bridge interface performs the following actions with the given constraints:

- If the Primary Parity Error Response Bit is set in the PCR, the Primary interface claims the transaction by asserting **P_TRDY#** and two clocks later asserts **P_PERR#**. The Delayed Completion cycle in the DWC Queue remains since the data of retried command did not match the data within the queue.

  If the Primary Parity Error Response Bit is clear in the PCR, the Primary interface will retry the transaction with no other response. A new transaction is not enqueued due to queue architecture constraints (see Section 14.7.1).

- The Detected Parity Error bit is set in the Primary Status Register (PSR) in the following scenario only: a transaction with bad parity was forwarded to the Secondary bus, which means that the Parity Response Enable Bit (PCR) associated with the Primary bus is not set. If the Primary Detected Parity Error Interrupt Mask bit is clear in the SDER, set the Detected Parity Error bit in the PBISR.

  Note that if the parity of the original request does not match the parity of the transaction the Primary master sends for a completion, the bridge will not detect a match for the completion attempt and will retry the transaction. In this case the transaction will most likely never be completed, and the enqueued data will eventually be discarded.

For upstream delayed completion transaction on the initiating bus where a data parity error occurs that did not occur on the target bus (i.e. status being returned is normal completion) the Secondary bridge interface performs the following actions with the given constraints:

- If the Secondary Parity Error Response Bit is set in the BCR, the Primary interface claims the transaction by asserting **S_TRDY#** and two clocks later asserts **S_PERR#**. The Delayed Completion cycle in the DWC Queue remains since the data of the retried command did not match the data within the queue.

  If the Secondary Parity Error Response Bit is clear in the BCR, the Secondary interface will retry the transaction with no other response. A new transaction is not enqueued due to queue architecture constraints (see Section 14.7.1).

- The Detected Parity Error bit is set in the Secondary Status Register (SSR) in the following scenario only: a transaction with bad parity was forwarded to the Primary bus, which means that the Parity Response Enable Bit (BCR) associated with the Secondary bus is not set. If the Secondary Detected Parity Error Interrupt Mask bit is clear in the SDER, set the Detected Parity Error bit in the SBISR.

  Note that if the parity of the original request does not match the parity of the transaction the Secondary master sends for a completion, the bridge will not detect a match for the completion attempt and will retry the transaction. In this case the transaction will most likely never be completed, and the enqueued data will eventually be discarded.

When returning status on the initiating bus from an error that occurred on the target and did not originally occur on the initiating bus, the bridge unit will assert **PERR#** (on the initiating bus interface) two clocks after the data is written assuming the bridge unit has parity enabled on both interfaces by the setting of the Primary and Secondary Parity Error Response bits in the PCR and BCR. The status is delivered and the transaction is cleared from the DWC queue. The appropriate Detected Parity Error bit is *not* set since the error did not actually occur on that bus.

14.11.2.3 Posted Write Data Parity

When a data parity error is detected by the bridge initiating interface during a posted write transactions that crosses the bridge, the bridge asserts **PERR#** on the initiating bus two clocks after the error is detected and retains the bad data and parity in the PMW Queue. The bridge will always perform the following on the initiating bus interface

- The Detected Parity Error bit in the PSR is set if the initiating bus is the Primary bus. If the Primary Detected Parity Error Interrupt Mask bit is clear in the SDER, set the Detected Parity Error bit in the PBISR.
- The Detected Parity Error bit in the SSR is set if the initiating bus is the Secondary bus. If the Secondary Detected Parity Error Interrupt Mask bit is clear in the SDER, set the Detected Parity Error bit in the SBISR.

When the write data is transferred on the target bus, the target of the transaction should assert **PERR#** on the target bus. If **PERR#** is asserted by the target, then the bridge sets:

- The Data Parity Detected bit in the PSR if the target bus is the Primary and the Primary Parity Error Response bit is set in the PCR. If the Primary Detected Parity Error Interrupt Mask bit is clear in the SDER, set the Detected Parity Error bit in the PBISR.
- The Data Parity Detected bit in the SSR if the target bus is the Secondary and the Secondary Parity Error Response bit is set in the BCR. If the Secondary Detected Parity Error Interrupt Mask bit is clear in the SDER, set the Detected Parity Error bit in the SBISR.

When a data parity error is detected on the target bus by the target of a posted write transaction when the bridge did not detect a data parity error on the initiating bus, the master of the transaction has no way of knowing that the data parity error has occurred (since **PERR#** cannot be forwarded back to the master due to the two clock cycle restriction for **PERR#** and data transferred). **P_SERR#** is used to notify the Primary interface of an error of this type. **P_SERR#** is only used when the data parity error occurs on the target bus and was not detected on the initiating bus.

For a downstream transaction, where no data parity error was detected on the Primary interface, which is completing on the Secondary bus and **S_PERR#** is detected by the Secondary interface the following actions are performed with the given constraints:

- The Data Parity Detected bit in the SSR is set if the Secondary Parity Error Enable bit is set in the BCR. If the Data Parity Detected bit is set in the SSR and the Secondary PCI Master Parity Error Interrupt Mask bit is clear in the SDER, set the Data Parity Detected bit in the SBISR.
- **P_SERR#** is asserted if:
  - — the Secondary Parity Error Enable bit is set in the BCR
  - — the Primary Parity Error Enable bit is set in the PCR
  - — the **SERR#** Enable bit is set in the PCR

For an upstream transaction, where no data parity error was detected on the Secondary interface, which is completing on the Primary bus and **P_PERR#** is detected by the Primary interface the following actions are performed with the given constraints:

- The Data Parity Detected bit in the PSR is set if the Primary Parity Error Enable bit is set in the PCR. If the Data Parity Detected bit is set in the PSR and the Primary PCI Master Parity Error Interrupt Mask bit is clear in the SDER, set the Data Parity Detected bit in the PBISR.
- **P_SERR#** is asserted if:
  - — the Secondary Parity Error Enable bit is set in the BCR
  - — the Primary Parity Error Enable bit is set in the PCR
  - — the **SERR#** Enable bit is set in the PCR

14.11.3 SERR# Assertion

Whenever **S_SERR#** is asserted on the Secondary interface of the bridge, the bridge must assert **P_SERR#** on the Primary interface if the following is true:

- The **SERR#** Enable bit is set in the PCR.
- The Secondary **SERR#** Enable bit is set in the BCR

The bridge must also set the Received System Error bit in the SSR. This function propagates the error upstream to the Primary interface. If the Received System Error bit in the SSR is set and the S_SERR# Detected Interrupt Mask bit in the SDER is clear, the S_SERR# Detected bit is set in the SBISR.

14.11.4 Discard Timers

The discard timers are responsible for preventing deadlocks when the initiator of a retried transaction fails to complete the transaction within 2^{10} or 2^{15} PCI clock cycles on the initiating bus. The timers start counting when the delayed request becomes a delayed completion by completing on the destination bus. If the originating master on the initiating bus has not retried the transaction before the associated timer expires, the completion transaction is discarded and **P_SERR#** is optionally asserted on the Primary bus.

There are eight discard timers in the bridge unit. Each PCI interface of the bridge unit has separate discard timers for a the DRC and DWC Queues in each direction. When the discard timer attached to a particular queue expires, the queue is invalidated, freeing the queue for use with a new transaction.

The discard timers are controlled through configuration bits in the Bridge Control Register. Delayed cycles initiated from the Primary bus interface have a programmable discard value of 2^{10} (enabled by setting bit 8 in the BCR) or 2^{15} (enabled by clearing bit 8 in the BCR). Delayed cycles initiated from the Secondary bus interface have a programmable discard value of 2^{10} (enabled by setting bit 9 in the BCR) or 2^{15} (enabled by clearing bit 9 in the BCR).

When a discard timer expires, the bridge sets (unconditionally) the Discard Timer Status bit in the BCR and optionally asserts **P_SERR#** if the following is true:

- The **SERR#** Enable bit is set in the PCR
- The Discard Timer **SERR#** Enable bit is set in the BCR

The Primary and Secondary Discard Timers can be disabled by setting Discard Timer Disable bit (bit 07) in the Extended Bridge Control Register (EBCR). When disabled, the timers will not count and delayed completion transactions will remain in their respective queues until retrieved by a PCI master.

14.11.5 PCI-to-PCI Bridge Error Summary

Table 14-20 and Table 14-21summarize the bridges error reporting for PCI bus errors (parity and transaction termination). The tables are in relation to the Primary and Secondary Status Registers. Each table details the corresponding registers error bit and the conditions that set the bit. The Primary and Secondary Bridge Interrupt Status Registers are also shown. These registers record i960 core processor interrupt status information.

Note: When an external agent violates PCI protocol, PCI-to-PCI Bridge behavior may be unpredictable/undefined.

Table 14-20. PSR Error Reporting Summary (Sheet 1 of 3)

Error Bit in Primary Status Register (PSR)	Error Condition	Qualifying Bit in Primary Command Register (PCR Unless Otherwise Noted)
Detected Parity Error - Bit 15	Address Parity Error on Primary Interface	N/A
	Upstream Read Data Parity Error on Primary Bus	N/A
	Downstream Posted Memory Write Data Parity Error on Primary Bus	N/A
	Downstream Delayed Write Data Parity Error During Request Cycle on Primary Bus if Request is Seen with Bad Parity & Immediately Completed	Primary Parity Error Response Enable must be SET- bit 6
	Downstream Delayed Write Data Parity Error During Completion Cycle on Primary Bus, Only if Error Occurred During Request Phase of Transaction on Primary Bus and Completion Request from Master Matches Initial Request Error	Primary Parity Error Response Enable must be CLEAR- bit 6
Data Parity Error Detected - Bit 8	Upstream Delayed Read Data Parity Error on Primary Bus	Primary Parity Error Response Enable - bit 6
	Upstream Delayed Write Data Parity Error During Completion Cycle on Primary Bus	Primary Parity Error Response Enable - bit 6
	Upstream Posted Memory Write Data Parity Error on the Primary Bus	Primary Parity Error Response Enable - bit 6

Table 14-20. PSR Error Reporting Summary (Sheet 2 of 3)

Error Bit in Primary Status Register (PSR)	Error Condition	Qualifying Bit in Primary Command Register (PCR Unless Otherwise Noted)
Signaled System Error - bit 14	Address Parity Error on Primary Interface	Primary Parity Error Response Enable - bit 6 **SERR#** Enable - bit 8
	Address Parity Error on Secondary Interface	Secondary Parity Error Response Enable - bit 0 (BCR) **SERR#** Enable - bit 8 **SERR#** Forwarding - bit 1 (BCR)
	Downstream Posted Memory Write Data Parity Error Which Occurs on Secondary Bus and Did Not Occur on Primary Bus	Primary Parity Error Response Enable - bit 6 Secondary Parity Error Response Enable - bit 0 (BCR) **SERR#** Enable - bit 8
	Upstream Posted Memory Write Data Parity Error Which Occurs on Primary Bus and Did Not Occur on Secondary Bus	Primary Parity Error Response Enable - bit 6 Secondary Parity Error Response Enable - bit 0 (BCR) **SERR#** Enable - bit 8
	Downstream Posted Memory Write that Receives a Target Abort on Secondary Bus and the Transaction is Not Currently Active on the Primary Interface	**SERR#** Enable - bit 8
	Upstream Posted Memory Write that Receives a Target Abort on Primary Bus and the Transaction is Not Currently Active on the Secondary Interface	**SERR#** Enable - bit 8
	Downstream Posted Memory Write that Ends in a Master Abort on the Secondary Bus	Master Abort Mode - bit 5 (BCR) **SERR#** Enable - bit 8
	Upstream Posted Memory Write that Ends in a Master Abort on the Primary Bus	Master Abort Mode - bit 5 (BCR) **SERR#** Enable - bit 8
	One of 8 Discard Timers Expire	Discard Timer **SERR#** Enable - bit 11 (BCR) **SERR#** Enable - bit 8
Master Abort - bit 13	Upstream Delayed Read (memory or I/O) Which Received a Master Abort on the Primary Bus	N/A
	Upstream Write (posted or delayed) Which Received a Master Abort on the Primary Bus	N/A
Target Abort (master) - bit 12	Upstream Delayed Read (memory or I/O) Which Received a Target Abort on the Primary Bus	N/A
	Upstream Write (posted or delayed) Which Received a Target Abort on the Primary Bus	N/A

Table 14-20. PSR Error Reporting Summary (Sheet 3 of 3)

Error Bit in Primary Status Register (PSR)	Error Condition	Qualifying Bit in Primary Command Register (PCR Unless Otherwise Noted)
Target Abort (target) - bit 11	Downstream Delayed Read Which Receives a Target Abort on the Secondary Bus. Set During Completion Cycle on Primary Bus.	N/A
	Downstream Delayed Write Which Received a Target Abort on the Secondary Bus. Set During Completion Cycle on Primary Bus.	N/A
	Downstream Posted Write Which Received a Target Abort on the Secondary Bus and the Transaction Was Still Active on the Primary Bus	N/A

Table 14-21. SSR Error Reporting Summary (Sheet 1 of 2)

Error Bit in Primary Status Register (SSR)	Error Condition	Qualifying Bit in Primary Command Register (BCR)
Detected Parity Error - Bit 15	Address Parity Error on Secondary Interface	N/A
	Downstream Read Data Parity Error on Secondary Bus	N/A
	Upstream Posted Memory Write Data Parity Error on Secondary Bus	N/A
	Upstream Delayed Write Data Parity Error During Request Cycle on Secondary Bus if Request is Seen with Bad Parity & Immediately Completed	Secondary Parity Error Response Enable must be SET- bit 0
	Upstream Delayed Write Data Parity Error During Completion Cycle on Secondary Bus, Only if Error Occurred During Request Phase of Transaction on Secondary Bus and Completion Request from Master Matches Initial Request Error	Secondary Parity Error Response Enable must be CLEAR- bit 0
Data Parity Error Detected - Bit 8	Downstream Delayed Read Data Parity Error on Secondary Bus	Secondary Parity Error Response Enable - bit 0
	Downstream Delayed Write Data Parity Error During Completion Cycle on Secondary Bus	Secondary Parity Error Response Enable - bit 0
	Downstream Posted Memory Write Data Parity Error on the Secondary Bus	Secondary Parity Error Response Enable - bit 0
Received System Error - bit 14	S_SERR# Detected on Secondary Interface	N/A
	Address Parity Error on Secondary Interface	Secondary Parity Error Response Enable - bit 0 (BCR)
Master Abort - bit 13	Downstream Delayed Read (memory or I/O) Which Received a Master Abort on the Secondary Bus	N/A
	Downstream Write (posted or delayed) Which Received a Master Abort on the Secondary Bus	N/A
Target Abort (master) - bit 12	Downstream Delayed Read (memory or I/O) Which Received a Target Abort on the Secondary Bus	N/A
	Downstream Write (posted or delayed) Which Received a Target Abort on the Secondary Bus	N/A

Table 14-21. SSR Error Reporting Summary (Sheet 2 of 2)

Error Bit in Primary Status Register (SSR)	Error Condition	Qualifying Bit in Primary Command Register (BCR)
Target Abort (target) - bit 11	Upstream Delayed Read Which Received a Target Abort on the Primary Bus. Set During Completion Cycle on Secondary Bus.	N/A
	Upstream Delayed Write Which Received a Target Abort on the Primary Bus. Set During Completion Cycle on Secondary Bus.	N/A
	Upstream Posted Write Which Received a Target Abort on the Primary Bus and the Transaction Was Still Active on the Secondary Bus	N/A

14.12 Initialization and Reset Requirements

When the Primary bus **P_RST#** is removed from the Primary interface, the PCI-to-PCI bridge unit is in an inactive mode. The bridge responds only to Type 0 configuration cycles with the Primary **IDSEL** input active. System configuration software is responsible for setting up the bridge unit for correct operation. Refer to the *PCI Local Bus Specification*, Revision 2.2 and the *PCI-to-PCI Bridge Architecture Specification*, Revision 1.1.

14.12.1 Bridge Reset

The PCI-to-PCI bridge unit has two independent reset states; one for the Primary interface and one for the Secondary interface. The Secondary **S_RST#** signal is the logical OR of the Primary interface **P_RST#** and the Secondary Bus Reset bit in the BCR. The *assertion* of the Secondary interface **S_RST#** output is asynchronous with respect to the Secondary output clocks (**S_CLKOUT[5:0]**); to support this, there exists a combinatorial path from the Primary **P_RST#** input and the Secondary Bus Reset bit in the BCR to the Secondary PCI bus **S_RST#** output. The *deassertion* of the **S_RST#** output is synchronous with the Secondary output clocks.

The bridge will not take any action if Secondary bus **S_RST#** is driven active by another device.

When the Secondary Bus Reset Bit in the BCR is set and subsequently cleared by software, the 80303 I/O processor can also be programmed to send an interrupt to the core processor. This is done using the Secondary Decode Enable Register - SDER.

During the reset sequence (no more than three clocks from the assertion of **P_RST#** on the Primary interface) the Secondary PCI Bus Arbitration Unit must park the Secondary bus on the Secondary bus interface. Refer to Section 17.2.1.3, "Priority Example with Six Bus Masters" on page 17-5 for details on bridge parking.

14.12.2 Configuring the PCI-to-PCI Bridge

For the bridge unit to operate in a system environment, several things must be properly initialized. The procedure below is required for all PCI-to-PCI bridges and is included here for completeness.

1. The Primary Bus Number, Secondary Bus Number and Subordinate Bus Number must be programmed with valid bus numbers. This must be done to allow the configuration software to probe the configuration space of downstream buses.
2. If I/O accesses must be forwarded downstream, the IOBR and IOLR register pair must be programmed to proper values and then the I/O Space Enable bit set in the PCR register. The ISA Enable bit of the BCR register should be set if the system includes ISA or EISA buses.
3. If Memory accesses must be forwarded downstream then both the Memory Mapped I/O range and the Prefetchable Memory range must be defined by programming the MBR/MLR and PMBR/PMLR register pairs. If only one address range is required then the PMBR/PMLR register pair can be programmed with the same values as the MBR/MLR register pair. After all four memory registers are valid, the Memory Enable bit in the PCR register can be set.
4. If VGA Compatible address forwarding is required, the VGA Enable bit (bit 3 of the BCR) must be set. Also, the Memory Space Enable and the I/O Space Enable bits must be set in order for VGA Compatible addressing to be operational. This mode forwards VGA Compatible Memory and I/O accesses downstream independent of the MBR/MLR, PMBR/PMLR, and IOBR/IOLR register pairs. This mode should be enabled when a VGA compatible device exists on the Secondary PCI bus.

5. If bus masters are to be supported on the downstream buses, the Bus Master Enable bit in the PCR register must be set. Note that once this bit is set, all I/O and Memory accesses on the Secondary bus that do not fall into the ranges defined by the bridge will be forwarded to the Primary interface and bus. This means that if the I/O Space Enable bit of the PCR register is not set, all I/O accesses on the Secondary bus will be passed to the Primary bus. Likewise, if the Memory Enable bit in the PCR is not set, all memory accesses on the Secondary bus will be forwarded to the Primary bus.
6. The CLSR, PLTR, and SLTR must be set to appropriate values before the bridge is fully functional. Most systems will want the **SERR#** Enable bits in the PCR and the BCR registers set as well.
7. To enable the *PCI Bus Power Management Interface Specification*, Revision 1.1 compliance support, the Power State Transition interrupt mask in bit 1 of the SDER needs to be cleared.
8. The Configuration Cycle Retry Bit in Section 14.15.23, "Extended Bridge Control Register - EBCR" on page 14-109 must then be cleared to allow the host to configure the bridge.

The previous list is the base minimum required to initialize the bridge unit. It is the configuration software responsibility to enable or disable the additional base and 80303 I/O processor specific features found in the bridge.

Note: If the bridge is using private PCI devices on the Secondary bus and their **IDSEL** inputs are using **S_AD[25:16]**, then the Secondary IDSEL Select Register must be programmed before the system configuration software probes the Secondary bus.

14.12.3 64-Bit Bus Configuration

At 80303 I/O processor reset time, it is the responsibility of the bus arbitration resource to configure the bus for 64-bit operation. If the bus is configured for 64-bit operation, the PCI master interfaces of the bridge will attempt memory transactions as 64-bit cycles. 64-bit bus operation is defined by the state of the **REQ64#** pin on the rising edge of the bus reset signal (**P_RST#** for the Primary bus). Table 14-22 details the bus configuration for the different states of each bus **REQ64#** at reset. The results of bus configuration operation are latched into the EBCR register (bit 8 for Primary bus and bit 9 for Secondary bus).

Table 14-22. 64-Bit Configuration Options at Reset

Pin	State at the Rising Edge of Reset	Bus Configuration
P_REQ64#	Asserted (logic 0)	64-bit Capable Bus
P_REQ64#	Deasserted (logic 1)	32-bit Only Bus
S_REQ64#	Asserted (logic 0)	64-bit Capable Bus
S_REQ64#	Deasserted (logic 1)	32-bit Only Bus

14.13 Power-up/Default States

Upon power-up and before **P_RST#** is asserted on the Primary interface, the bridge is in an inactive mode of operation. After reset, all internal registers associated with the bridge configuration address space are set to the default values defined in Section 14.15. The posting buffers are marked invalid. Refer to Section 14.12.1 for details on resetting the PCI-to-PCI bridge unit.

14.14 Performance Considerations

As a bridge device, the 80303 I/O processor will have a negative impact on the performance of a transaction initiated on one bus and completed on another bus as compared to a transaction that is initiated and completed on the same bus.

The following are a list of performance requirements for PCI-to-PCI bridge unit operation:

- The bridge unit uses Medium Decoding on both the Primary and Secondary interfaces. This means that as a target, the bridge will claim the transaction by asserting **DEVSEL#** (if decoded and accepted) within three PCI clocks of the assertion of **FRAME#** on the initiating interface.
- The bridge unit will not take any additional clocks to forward data through an non-full queue. The bridge unit is capable of emptying the data queues at the rate of one 64-bit word per PCI clock from the second through the nth word of a delayed or posted transaction.

14.15 Register Definitions

The following sections describe the PCI-to-PCI bridge configuration registers. The configuration space consists of 8-, 16-, 24-, and 32-bit registers arranged in a predefined format. The configuration registers are accessed through Type 0 Configuration Reads and Writes on the Primary side of the bridge and through i960 processor local operations.

Each register is detailed in functionality, access type (read/write, read/clear, read only) and reset default condition. As stated, a Type 0 configuration command on the Primary side with an active **IDSEL** or a memory-mapped i960 processor access is required to read or write these registers. The format for the registers with offsets up to 3EH are defined with the *PCI-to-PCI Bridge Architecture Specification*, Revision 1.1. Registers with offsets greater than 3EH are implementation specific to the 80303 I/O processor.

See Chapter 1, "Introduction" for definitions of *reserved, read only, writeonce/readonly,* and *read/clear.* All registers adhere to the definitions found in the *PCI Local Bus Specification*, Revision 2.2 and the *PCI-to-PCI Bridge Architecture Specification*, Revision 1.1 unless otherwise noted.

An additional requirement exists to allow the i960 core processor to access the bridge configuration space. Some registers that are *read only* from Type 0 Configuration Read and Write commands may be writable from the i960 core processor. This allows certain configuration registers to be initialized before PCI configuration begins. See Appendix C, "Peripheral Memory-Mapped Registers".

The i960 core processor will read and write the bridge configuration space as memory-mapped registers. Table 14-23 shows the register and its associated offset used in a PCI configuration command and its memory-mapped address in the i960 processor address space.

The assertion of the **P_RST#** signal on the Primary side of the bridge affects the state of **most** of the registers contained within the bridge configuration space. Unless otherwise noted, all bits and registers will return to their stated default state value upon Primary reset. The only bit affected by **I_RST#** is bit 5 of the EBCR, Reset Internal Bus.

Figure 14-11. Bridge Configuration Header Format

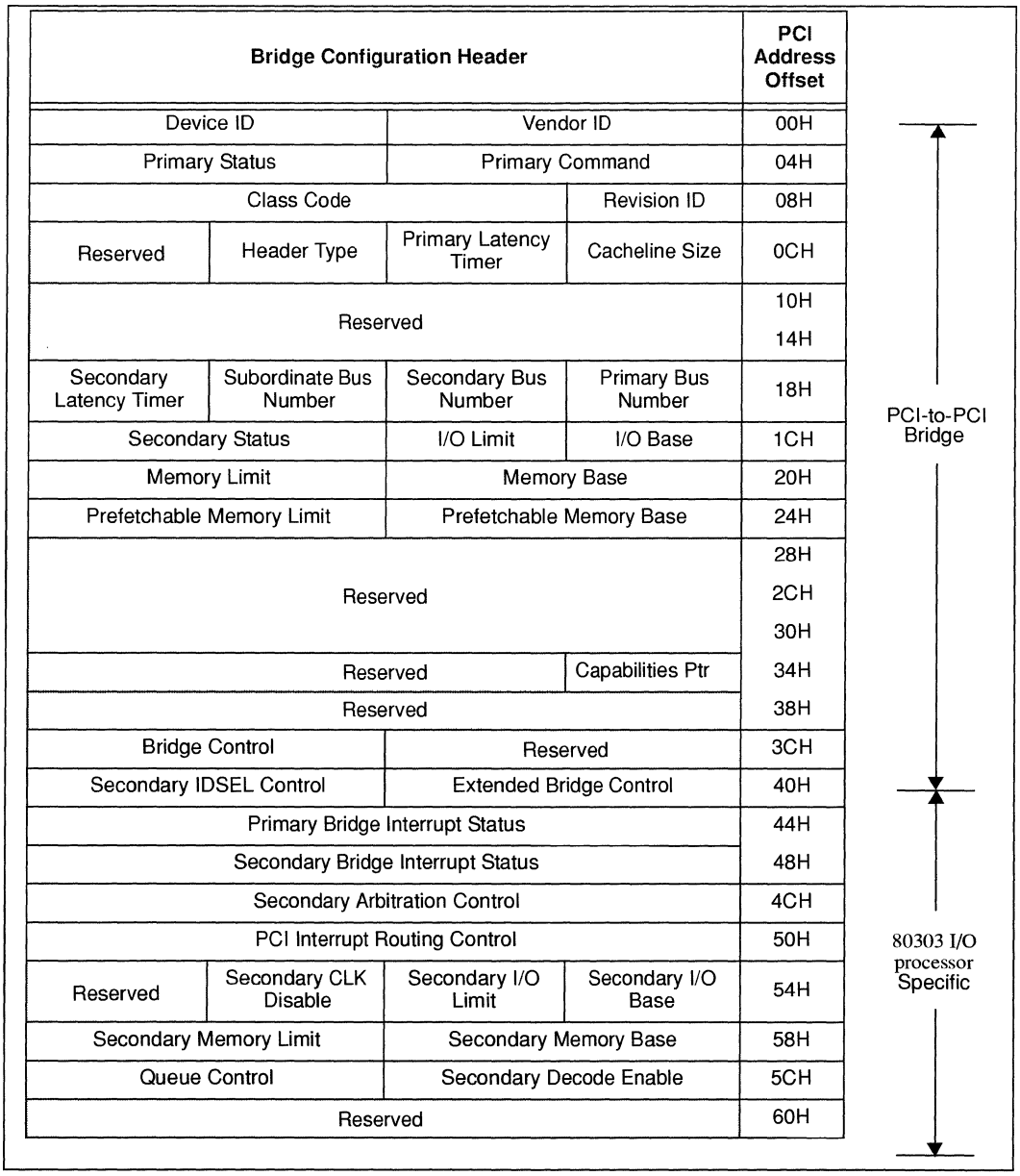


Figure 14-12. Extended Bridge Configuration Header Format

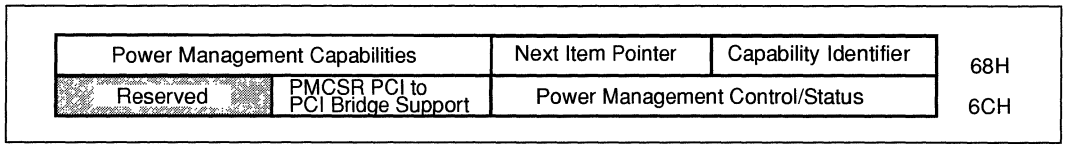


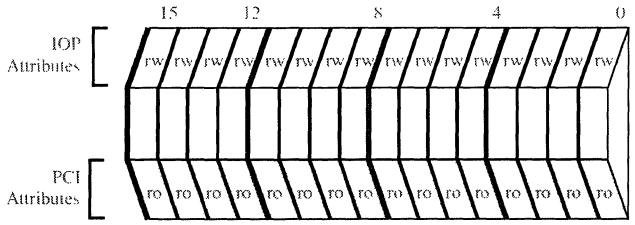
Table 14-23. PCI-to-PCI Bridge Register Table

Internal Bus Address	Section, Register Name - Acronym (Page)
1000H	Section 14.15.1, "Vendor Identification Register - VIDR" on page 14-85
1002H	Section 14.15.2, "Device ID Register - DIDR" on page 14-86
1004H	Section 14.15.3, "Primary Command Register - PCR" on page 14-87
1006H	Section 14.15.4, "Primary Status Register - PSR" on page 14-88
1008H	Section 14.15.5, "Revision ID Register - RID" on page 14-90
1009H	Section 14.15.6, "Class Code Register - CCR" on page 14-91
100CH	Section 14.15.7, "Cacheline Size Register - CLSR" on page 14-92
100DH	Section 14.15.8, "Primary Latency Timer Register - PLTR" on page 14-93
100EH	Section 14.15.9, "Header Type Register - HTR" on page 14-94
1018H	Section 14.15.10, "Primary Bus Number Register - PBNR" on page 14-95
1019H	Section 14.15.11, "Secondary Bus Number Register - SBNR" on page 14-96
101AH	Section 14.15.12, "Subordinate Bus Number Register - SubBNR" on page 14-97
101BH	Section 14.15.13, "Secondary Latency Timer Register - SLTR" on page 14-98
101CH	Section 14.15.14, "I/O Base Register - IOBR" on page 14-99
101DH	Section 14.15.15, "I/O Limit Register - IOLR" on page 14-100
101EH	Section 14.15.16, "Secondary Status Register - SSR" on page 14-101
1020H	Section 14.15.17, "Memory Base Register - MBR" on page 14-102
1022H	Section 14.15.18, "Memory Limit Register - MLR" on page 14-103
1024H	Section 14.15.19, "Prefetchable Memory Base Register - PMBR" on page 14-104
1026H	Section 14.15.20, "Prefetchable Memory Limit Register - PMLR" on page 14-105
1034H	Section 14.15.21, "Capabilities Pointer Register - Cap_Ptr" on page 14-106
103EH	Section 14.15.22, "Bridge Control Register - BCR" on page 14-107
1040H	Section 14.15.23, "Extended Bridge Control Register - EBCR" on page 14-109
1042H	Section 14.15.24, "Secondary IDSEL Select Register - SISR" on page 14-111
1044H	Section 14.15.25, "Primary Bridge Interrupt Status Register - PBISR" on page 14-113
1048H	Section 14.15.26, "Secondary Bridge Interrupt Status Register - SBISR" on page 14-114
104CH	Section 14.15.27, "Secondary Arbitration Control Register - SACR" on page 14-115
1050H	Section 14.15.28, "PCI Interrupt Routing Select Register - PIRSR" on page 14-115
1054H	Section 14.15.29, "Secondary I/O Base Register - SIOBR" on page 14-116
1055H	Section 14.15.30, "Secondary I/O Limit Register - SIOLR" on page 14-117
1056H	Section 14.15.31, "Secondary Clock Disable Register - SCDR" on page 14-118
1058H	Section 14.15.32, "Secondary Memory Base Register - SMBR" on page 14-119
105AH	Section 14.15.33, "Secondary Memory Limit Register - SMLR" on page 14-120
105CH	Section 14.15.34, "Secondary Decode Enable Register - SDER" on page 14-121
105EH	Section 14.15.35, "Queue Control Register - QCR" on page 14-123
1060H	**Reserved**
1064H	**Reserved**
1068H	Section 14.15.36, "Capability Identifier Register - Cap_ID" on page 14-124
1069H	Section 14.15.37, "Next Item Pointer Register - Next_Item_Ptr" on page 14-125
106AH	Section 14.15.38, "Power Management Capabilities Register - PMCR" on page 14-126
106CH	Section 14.15.39, "Power Management Control/Status Register - PMCSR" on page 14-127
106EH	Section 14.15.40, "PMCSR PCI-to-PCI Bridge Support - PMCSR_BSE" on page 14-128

14.15.1 Vendor Identification Register - VIDR

Vendor ID Register bits adhere to the definitions in the *PCI Local Bus Specification*, Revision 2.2.

Table 14-24. Vendor Identification Register - VIDR



PCI Configuration Offset
00 - 01H

Intel® i960® Core Local Bus Address
0000 1000H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set

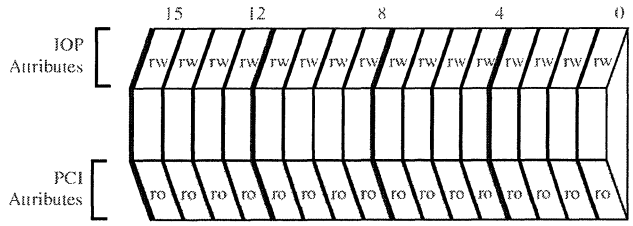
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
15:00	8086H	Vendor ID - A unique identifier indicating the manufacturer of a PCI device

14.15.2 Device ID Register - DIDR

Device ID Register bits adhere to the definitions in the *PCI Local Bus Specification*, Revision 2.2.

Table 14-25. Device Identification Register - DIDR (80303 I/O processor)



PCI Configuration Offset
02 - 03H

Intel® i960® Core Local Bus Address
0000 1002H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set

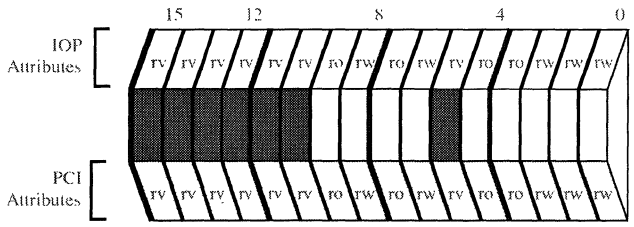
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
15:00	0309H	Device ID - This is a 16-bit value assigned to the 80303 I/O processor. This register, combined with the VID, uniquely identify any PCI device.

14.15.3 Primary Command Register - PCR

Primary Command Register bits adhere to the definitions in the *PCI Local Bus Specification*, Revision 2.2 and in most cases affect the behavior of the Primary interface of the PCI-to-PCI bridge.

Table 14-26. Primary Command Register - PCR



PCI Configuration Offset
04 - 05H

Intel® i960® Core Local Bus Address
0000 1004H

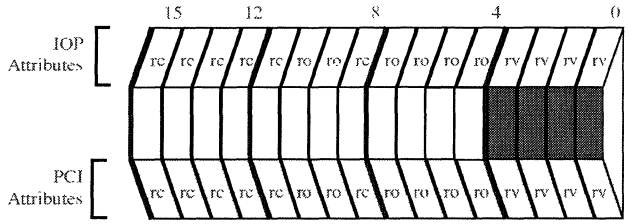
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
15:10	00000_2	Reserved.
09	0_2	Fast Back to Back Enable - This Primary interface does not perform fast back to back transactions.
08	0_2	**SERR#** Enable - If this bit is cleared, the 80303 I/O processor is not allowed to assert **P_SERR#** on its Primary interface.
07	0_2	Wait Cycle Control - controls address/data stepping. Not implemented and a reserved bit field
06	0_2	Primary Parity Error Response Enable - If this bit is set, then the bridge must take normal action when a parity error is detected. If it is cleared, then parity checking is disabled.
05	0_2	VGA Palette Snoop Enable - VGA Palette Snooping is not supported.
04	0_2	Memory Write and Invalidate Enable - Not applicable. A PCI-to-PCI bridge does not initiate MWI commands, only forwards them on behalf of another master. The initiator has the control to determine which type of write command to use.
03	0_2	Special Cycle Enable - The bridge cannot respond as the target of a Special Cycle so this bit field is defined as read only.
02	0_2	Bus Master Enable - Controls the bridge ability to operate as a master on the Primary interface for memory and I/O transactions. This bit does not affect the bridge ability to forward or convert Type 1 configuration commands. When this bit is set, the bridge is enabled to act as a master on the Primary interface. When this bit is clear, the bridge will not claim any memory or I/O transactions on the Secondary PCI interface.
01	0_2	Memory Enable - Controls the bridge response to both memory and prefetchable memory accesses. If this bit is cleared, the bridge will not respond to any memory access on the Primary PCI interface.
00	0_2	I/O Space Enable - Controls the bridges response to I/O transactions on the Primary PCI interface. If this bit is cleared, the bridge will not respond to any I/O transaction on the Primary side.

14.15.4 Primary Status Register - PSR

Primary Status Register bits adhere to the definitions in the *PCI Local Bus Specification*, Revision 2.2 but only apply to the Primary interface of the bridge. The *read/clear* bits can only be set by the internal hardware and are cleared by either a reset condition of by writing a 1_2 to the register.

Table 14-27. Primary Status Register - PSR (Sheet 1 of 2)



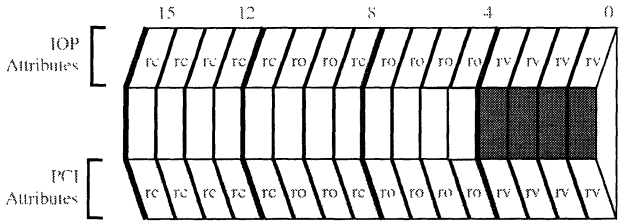
PCI Configuration Offset
06 - 07H

Intel® i960® Core Local Bus Address
0000 1006H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
15	0_2	Detected Parity Error - This bit is set when a parity error is detected during a data transfer on the Primary bus even if parity handling is disabled. Set under the following conditions: • Write Data Parity Error when the Primary interface of the Bridge is a slave (downstream write). • Read Data Parity Error when the Primary interface of the Bridge is a master (upstream read). • Any Address Parity Error on the Primary Bus (including one generated by the Primary interface of the Bridge).
14	0_2	Signaled System Error - This bit is set if **P_SERR#** is asserted on the Primary bus.
13	0_2	Master Abort - This bit is set whenever a transaction initiated by the bridge on the Primary bus (except Special Cycles) ends in a Master-Abort.
12	0_2	Target Abort (master) - This bit is set whenever a transaction initiated by the Primary interface ends in a Target-Abort.
11	0_2	Target Abort (target) - This bit is set whenever the bridge, acting as a target, terminates the transaction on the Primary bus with a Target-Abort.
10:09	10_2	DEVSEL# Timing - These bits are read-only and define the slowest DEVSEL# timing for a target device (except configuration accesses). 00_2 = Fast 01_2 = Medium 10_2 = Slow 11_2 = Reserved In general, the Primary interface uses Medium timing.Though this occurrence is not part of the normal bridge operation, the *PCI Local Bus Specification*, Revision 2.2 requires that this field of the status register indicate the slowest DEVSEL# possible by a target device. In the event that a subtractive decode agent is present on the bridge primary bus interface, the indication that the bridge could claim positively with Slow decode will prevent the users of this bus from programming the subtractive decode agent to claim with Slow decode timing. With the exception of Primary Interface Slow Decode under certain, special conditions, the performance of the bridge will **not** be affected.
08	0_2	Data Parity Error Detected - This bit is set when the bridge: • asserted **P_PERR#** (or saw asserted) on the Primary bus. • and was the master of the transaction when it occurred. • and the Primary Parity Error Response bit is set in the PCR.

Table 14-27. Primary Status Register - PSR (Sheet 2 of 2)



PCI Configuration Offset
06 - 07H

Intel® i960® Core Local Bus Address
0000 1006H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set

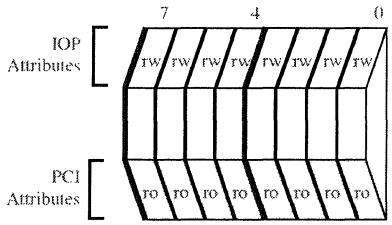
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
07	1_2	Fast Back-to-Back Capable - Indicates that the Primary interface capable of accepting Fast Back-to-Back transactions as a target.
06	0_2	UDF Supported - User Definable Features are not supported.
05	1_2	66 MHz. Capable - 66 MHz. operation is supported.
04	1_2	**Capabilities** - This function implements extended capabilities.
03:00	0000_2	Reserved.

14.15.5 Revision ID Register - RID

Revision ID Register bits adhere to definitions in the *PCI Local Bus Specification*, Revision 2.2.

Table 14-28. Revision Identification Register - RID



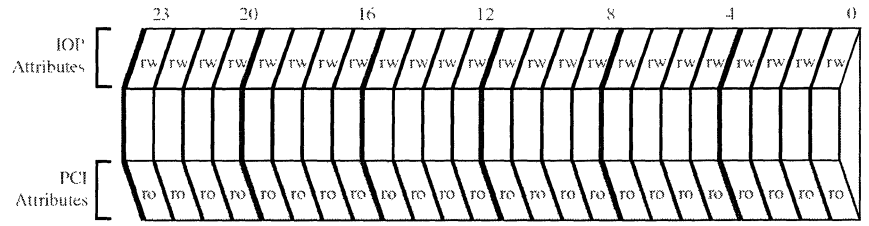
PCI Configuration Offset	Intel® i960® Core Local Bus Address	Attribute Legend:	
08H	0000 1008H	RV = Reserved	RW = Read/Write
		PR = Preserved	RC = Read Clear
		RS = Read/Set	RO = Read Only
			NA = Not Accessible

Bit	Default	Description
07:00	00H	Revision ID - This value identifies the revision number of the 80303 I/O processor.

14.15.6 Class Code Register - CCR

Class Code Register bits adhere to the definitions in the *PCI Local Bus Specification*, Revision 2.2. It tells the auto configuration software the type of function present in the PCI device.

Table 14-29. Class Code Register - CCR



PCI Configuration Offset
09 - 0BH

Intel® i960® Core Local Bus Address
0000 1009H

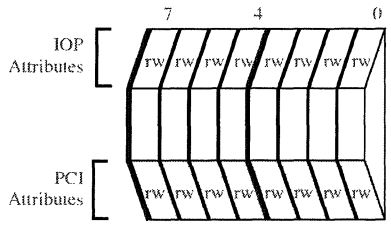
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
23:16	06H	Base Class - Bridge Device
15:08	04H	Sub Class - PCI-to-PCI Bridge Device
07:00	00H	Programming Interface - Consistent with PCI-to-PCI Bridge Architecture Specification Revision 1.0.

14.15.7 Cacheline Size Register - CLSR

Cacheline Size Register bits adhere to the definitions in the *PCI Local Bus Specification*, Revision 2.2 and apply to both sides of the bridge. It is programmed with the system cacheline size in DWORDs (32-bit quantities). The Cacheline Size is restricted to either 32 or 64 bytes. If a value other than 8 or 16 is written to the Cacheline Size Register, the Bridge will behave as if a value of zero was written.

Table 14-30. Cacheline Size Register - CLSR



PCI Configuration Offset: 0CH

Intel® i960® Core Local Bus Address: 0000 100CH

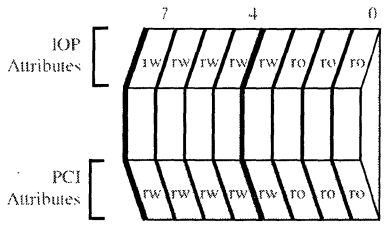
Attribute Legend:
RW = Read/Write
RV = Reserved
RC = Read Clear
PR = Preserved
RO = Read Only
RS = Read/Set
NA = Not Accessible

Bit	Default	Description
07:00	00H	Cacheline Size - Cacheline size in DWORDs. Cacheline size is restricted to a register value of 8 or 16 for 32- or 64-byte cachelines, respectively.

14.15.8 Primary Latency Timer Register - PLTR

Primary Latency Timer Register bits adhere to the definitions in the *PCI Local Bus Specification*, Revision 2.2 and apply to the Primary side of the bridge only. It loads a timer at the beginning of each PCI transaction initiated by the bridge on the Primary bus. If the timer counts down to zero, the bridge must terminate the transaction as soon as the **GNT#** signal is deasserted.

Table 14-31. Primary Latency Timer Register- PLTR



PCI Configuration Offset
0DH

Intel® i960® Core Local Bus Address
0000 100DH

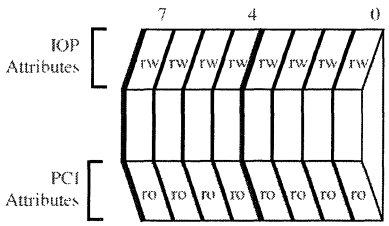
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
07:03	00000_2	Programmable Latency Timer - This portion of the register varies the latency timer for the Primary interface from a minimum of zero clocks to a maximum of 248 clocks.
02:00	000_2	Latency Timer Granularity - These bits are read only giving a programmable granularity of eight clocks for the Latency Timer.

14.15.9 Header Type Register - HTR

Header Type Register bits adhere to the definitions in the *PCI Local Bus Specification*, Revision 2.2. This register indicates the layout of bytes 10H to 3FH of the bridge configuration space. The most significant bit indicates whether or not the device is multi-function and is defined as a 1 for multi-function device in the *PCI-to-PCI Bridge Architecture Specification*, Revision 1.1. (Refer to Section 15.2.4, "PCI Multi-Function Device Swapping/Disabling" on page 15-23 for exceptions to this statement.)

Table 14-32. Header Type Register- HTR



PCI Configuration Offset
0EH

Intel® i960® Core Local Bus Address
0000 100EH

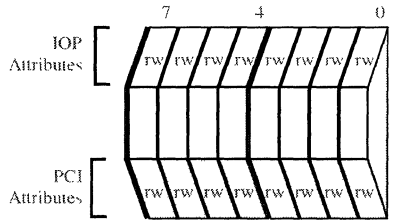
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
7	1_2	Single Function/Multi-Function Device - This bit identifies whether or not the 80303 I/O processor is a single or multi-function PCI device. The 80303 I/O processor is considered a multi-function device.
06:00	000001_2	PCI Header Type - This bit field tells the system initialization code what type of PCI header is implemented. The 80303 I/O processor has a PCI-to-PCI bridge header as defined in *PCI-to-PCI Bridge Architecture Specification*, Revision 1.0.

14.15.10 Primary Bus Number Register - PBNR

Primary Bus Number Register bits adhere to the definitions in the *PCI Local Bus Specification*, Revision 2.2. This register records the bus number of the bridge Primary interface. This register decodes Type 1 configuration transactions on the Secondary interface that should be converted to Special Cycle transactions on the Primary interface.

Table 14-33. Primary Bus Number Register- PBNR



PCI Configuration Offset: 18H
Intel® i960® Core Local Bus Address: 0000 1018H

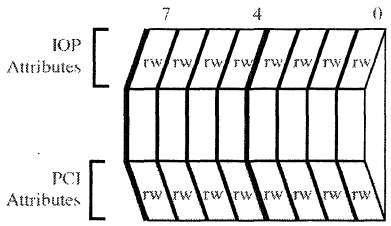
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
07:00	00H	Primary Bus Number - This field is programmed with the PCI bus number of the bridge Primary interface.

14.15.11 Secondary Bus Number Register - SBNR

Secondary Bus Number Register bits adhere to the definitions in the PCI Local Bus Specification.This register records the bus number of the bridge Secondary interface. This register determines when to respond to Type 1 configuration commands on the Primary interface and convert them to Type 0 commands on the Secondary interface.

Table 14-34. Secondary Bus Number Register - SBNR



PCI Configuration Offset: 19H

Intel® i960® Core Local Bus Address: 0000 1019H

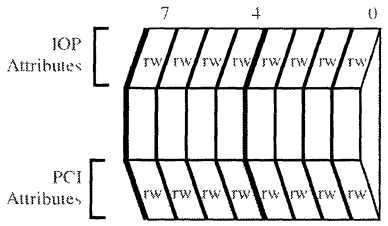
Attribute Legend:
RW = Read/Write
RC = Read Clear
RV = Reserved
RO = Read Only
PR = Preserved
NA = Not Accessible
RS = Read/Set

Bit	Default	Description
07:00	00H	Secondary Bus Number - This field is programmed with the PCI bus number of the bridge Secondary interface.

14.15.12 Subordinate Bus Number Register - SubBNR

Subordinate Bus Number Register bits adhere to the definitions in the *PCI Local Bus Specification*, Revision 2.2. This register records the highest numbered PCI bus behind the bridge. This register is used in conjunction with the Secondary bus number to determine when to respond to Type 1 configuration commands on the Primary bus and pass them on to the Secondary interface.

Table 14-35. Subordinate Bus Number Register - SubBNR



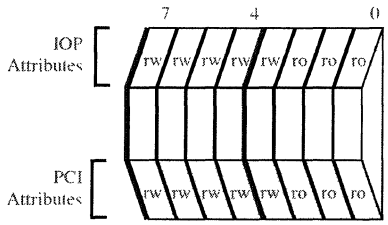
PCI Configuration Offset	Intel® i960® Core Local Bus Address	Attribute Legend:	
1AH	0000 101AH	RV = Reserved	RW = Read/Write
		PR = Preserved	RC = Read Clear
		RS = Read/Set	RO = Read Only
			NA = Not Accessible

Bit	Default	Description
07:00	00H	Subordinate Bus Number - This field is programmed with the highest numbered PCI bus which exists behind the bridge.

14.15.13 Secondary Latency Timer Register - SLTR

Secondary Latency Timer Register bits adhere to the definitions in the *PCI Local Bus Specification*, Revision 2.2 and apply to the Secondary interface of the bridge only. It loads a timer at the beginning of each PCI transaction initiated by the bridge on the Secondary bus. If the timer counts down to zero, the bridge must terminate the transaction as soon as the **GNT#** signal is deasserted.

Table 14-36. Secondary Latency Timer Register - SLTR



PCI Configuration Offset
1BH

Intel® i960® Core Local Bus Address
0000 101BH

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
07:03	00000_2	Programmable Latency Timer - This portion of the register varies the latency timer for the Secondary interface from a minimum of zero clocks to a maximum of 248 clocks.
02:00	000_2	Latency Timer Granularity - These bits are read only giving a programmable granularity of eight clocks for the Latency Timer.

14.15.14 I/O Base Register - IOBR

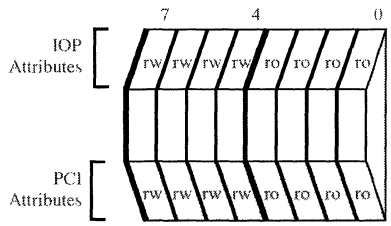
I/O Base Register bits adhere to the definitions in the *PCI Local Bus Specification*, Revision 2.2. The I/O Base Register defines the bottom address (inclusive) of an address range that is used to determine when to forward I/O transactions from one side of the bridge to the other. It must be programmed with a valid value before the *I/O Space Enable* bit in the Primary Command Register (PCR) is set. The bridge only supports 16-bit addressing which is indicated by a value of 0H in the four least significant bits of the register. The upper 4 bits are programmed with **AD[15:12]** for the bottom of the address range. **AD[11:0]** of the base address is always 000H forcing the I/O address range to be 4 Kbyte aligned.

For the purposes of address decoding, the bridge assumes that **AD[31:16]**, the upper 16 address bits of the I/O address, are zero. The bridge must still perform the address decode on the full 32 bits of address per PCI Local Bus Specification and check that the upper 16 bits are equal to 0000H.

The I/O address range (defined by the IOBR in conjunction with the IOLR) is modified by the ISA Enable bit of the Bridge Control Register (BCR). If this bit is set then I/O addresses in the range X400H - XFFFH will not be accepted by the Primary side of the bridge, even if the address falls within the defined I/O address range.

The VGA Enable bit in the BCR will cause I/O accesses where **AD[9:0]** are in the ranges 3B0H - 3BBH and 3C0H - 3DFH (inclusive of ISA addresses - **AD[15:10]** are not decoded) to be forwarded from primary to secondary and blocked from secondary to primary.

Table 14-37. I/O Base Register - IOBR



PCI Configuration Offset
1CH

Intel® i960® Core Local Bus Address
0000 101CH

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
07:04	0H	I/O Base Address - This field is programmed with **AD[15:12]** of the bottom of the I/O address range to be passed down the hierarchy by the bridge.
03:00	0H	I/O Addressing Capability - The value of 0H signifies that the bridge only supports 16-bit I/O addressing.

14.15.15 I/O Limit Register - IOLR

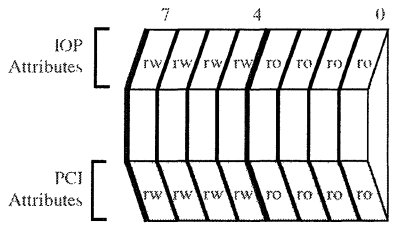
I/O Limit Register bits adhere to the definitions in the *PCI Local Bus Specification*, Revision 2.2. The I/O Limit Register defines the upper address (inclusive) of an address range that is used to determine when to forward I/O transactions from one side of the bridge to the other. It must be programmed with a valid value greater than or equal to the IOBR before the *I/O Space Enable* bit in the Primary Command Register (PCR) is set. The bridge only supports 16-bit addressing which is indicated by a value of 0H in the four least significant bits of the register. The upper 4 bits are programmed with **AD[15:12]** for the top of the address range. **AD[11:0]** of the limit address is always FFFH forcing a 4 Kbyte I/O range granularity.

For the purposes of address decoding, the bridge assumes that **AD[31:16]**, the upper 16 address bits of the I/O address, are zero. The bridge must still perform the address decode on the full 32 bits of address per *PCI Local Bus Specification*, Revision 2.2 and check that the upper 16 bits are equal to 0000H.

The I/O address range (defined by the IOBR in conjunction with the IOLR) is modified by the *ISA Enable* bit of the Bridge Control Register. If this bit is set then I/O addresses in the range X400H - XFFFH will not be accepted by the Primary side of the bridge, even if the address falls within the defined I/O address range.

The VGA Enable bit in the BCR will cause I/O accesses where **AD[9:0]** are in the ranges 3B0H - 3BBH and 3C0H - 3DFH (inclusive of ISA addresses - **AD[15:10]** are not decoded) to be forwarded from primary to secondary and blocked from secondary to primary.

Table 14-38. I/O Limit Register - IOLR



PCI Configuration Offset: 1DH

Intel® i960® Core Local Bus Address: 0000 101DH

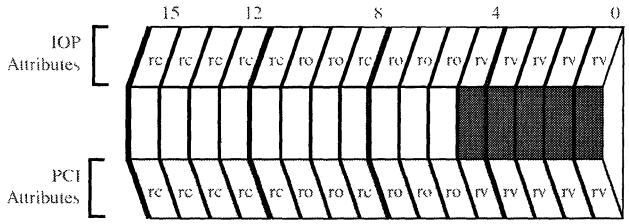
Attribute Legend:
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible
RV = Reserved
PR = Preserved
RS = Read/Set

Bit	Default	Description
07:04	0H	I/O Limit Address - This field is programmed with **AD[15:12]** of the top of the I/O address range to be passed down the hierarchy by the bridge.
03:00	0H	I/O Addressing Capability - The value of 0H signifies that the bridge only supports 16-bit I/O addressing.

14.15.16 Secondary Status Register - SSR

Secondary Status Register bits adhere to the definitions in the *PCI Local Bus Specification*, Revision 2.2 (with modifications made to bit 14 by the *PCI-to-PCI Bridge Architecture Specification*, Revision 1.1) and apply to the Secondary interface of the bridge only. The Read/Clear bits can only be set by the hardware. They are cleared when **S_RST#** is asserted or by writing a 1_2 to the bit location.

Table 14-39. Secondary Status Register - SSR



PCI Configuration Offset
1E - 1FH

Intel® i960® Core Local Bus Address
0000 101EH

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

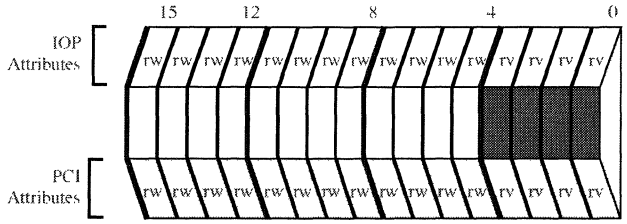
Bit	Default	Description
15	0_2	Detected Parity Error - This bit is set when a parity error is detected during a data transfer on the Secondary bus even if parity handling is disabled. Set under the following conditions: • Write Data Parity Error when the Secondary interface of the Bridge is a slave (upstream write). • Read Data Parity Error when the Secondary interface of the Bridge is a master (downstream read). • Any Address Parity Error on the Secondary Bus (including one generated by the Secondary interface of the Bridge).
14	0_2	Received System Error - When set indicates that **S_SERR#** was detected by the bridge on the Secondary interface.
13	0_2	Master Abort - This bit is set whenever a transaction initiated by the Secondary interface (except Special Cycles) ends in Master-Abort
12	0_2	Target Abort (master) - This bit is set whenever a transaction initiated by the Secondary interface ends in a Target-Abort.
11	0_2	Target Abort (target) - This bit is set whenever the Secondary interface, acting as a target, terminates a transaction with a Target-Abort
10:09	01_2	**DEVSEL#** Timing - Medium Decode Timing for the Secondary interface
08	0_2	Data Parity Error Detected - This bit is set when the bridge: • asserted **S_PERR#** (or saw asserted) on the Secondary bus • and was the master of the transaction when it occurred • and the Secondary Parity Error Response bit is set in the BCR.
07	1_2	Fast Back-to-Back Capable - Indicates that the Secondary interface is capable of accepting Fast Back-to-Back transactions as a target on the Secondary interface
06	0_2	UDF Supported - This indicates that User Definable Features is not supported
05	1_2	66 MHz. Capable - 66 MHz. operation is supported.
04:00	00000_2	Reserved.

14.15.17 Memory Base Register - MBR

Memory Base Register bits adhere to the definitions in the *PCI Local Bus Specification*, Revision 2.2. The Memory Base Register defines the bottom address (inclusive) of a memory-mapped I/O address range (non-prefetchable) that is used to determine when to forward memory transactions from one side of the bridge to the other. The Memory Base Register must be programmed before the *Memory Space Enable* bit of the Primary Command Register (PCR) is set. The upper 12 bits correspond to **AD[31:20]** of 32-bit addresses. For the purposes of address decoding, the bridge assumes that **AD[19:0]**, the lower 20 address bits of the memory base address, are zero. This means that the bottom of the defined address range will be aligned on a 1 Mbyte boundary.

The VGA Enable bit in the BCR register forces the bridge to forward memory accesses in the address range from 0A0000H to 0BFFFFH from the Primary to Secondary and blocks addresses in the same range from Secondary to Primary.

Table 14-40. Memory Base Register - MBR



PCI Configuration Offset
20 - 21H

Intel® i960® Core Local Bus Address
0000 1020H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set

RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

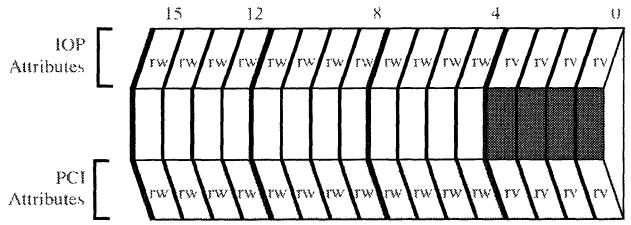
Bit	Default	Description
15:04	000H	Memory Base Address - This field is programmed with **AD[31:20]** of the bottom of the memory address range to be passed down the hierarchy by the bridge.
03:00	0H	Reserved.

14.15.18 Memory Limit Register - MLR

Memory Limit Register bits adhere to the definitions in the *PCI Local Bus Specification*, Revision 2.2. The Memory Limit Register defines the upper address (inclusive) of the memory-mapped I/O address range (non-prefetchable) that is used to determine when to forward memory transactions from one side of the bridge to the other. The Memory Limit Register must be programmed to a value greater than or equal to the MBR before the *Memory Space Enable* bit of the Primary Command Register is set. The upper 12 bits correspond to **AD[31:20]** of 32-bit addresses. For the purposes of address decoding, the bridge assumes that **AD[19:0]**, the lower 20 bits of the memory limit address, are FFFFFH. This forces a 1 Mbyte granularity on the memory address range.

The VGA Enable bit in the BCR register forces the bridge to forward memory accesses in the address range from 0A0000H to 0BFFFFH from the Primary to Secondary and blocks addresses in the same range from Secondary to Primary.

Table 14-41. Memory Limit Register - MLR



PCI Configuration Offset
22 - 23H

Intel® i960® Core Local Bus Address
0000 1022H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

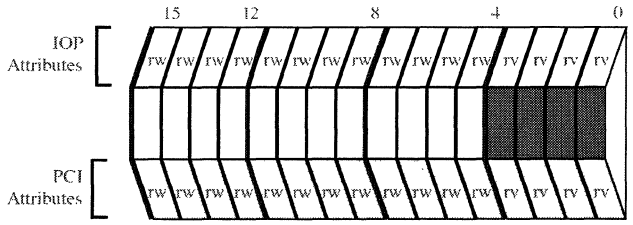
Bit	Default	Description
15:04	000H	Memory Limit Address - This field is programmed with **AD[31:20]** of the top of the memory address range to be passed down the hierarchy by the bridge.
03:00	0H	Reserved.

14.15.19 Prefetchable Memory Base Register - PMBR

The Prefetchable Memory Base Register defines the bottom address (inclusive) of a prefetchable memory address range that is used to determine when to forward memory transactions from one side of the bridge to the other. The Prefetchable Memory Base Register must be programmed before the *Memory Space Enable* bit of the Primary Command Register (PCR) is set. The upper 12 bits correspond to **AD[31:20]** of 32-bit addresses. For the purposes of address decoding, the bridge assumes that **AD[19:0]**, the lower 20 address bits of the memory base address, are zero. This means that the bottom of the defined address range will be aligned on a 1 Mbyte boundary.

The VGA Enable bit in the BCR register forces the bridge to forward memory accesses in the address range from 0A0000H to 0BFFFFH from the Primary to Secondary and blocks addresses in the same range from Secondary to Primary.

Table 14-42. Prefetchable Memory Base Register - PMBR



PCI Configuration Offset
24 - 25H

Intel® i960® Core Local Bus Address
0000 1024H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set

RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

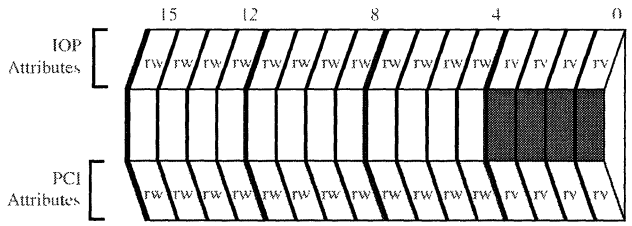
Bit	Default	Description
15:04	000H	Prefetchable Memory Base Address - This field is programmed with **AD[31:20]** of the bottom of the memory address range to be passed down the hierarchy by the bridge.
03:00	0H	Reserved.

14.15.20 Prefetchable Memory Limit Register - PMLR

The Prefetchable Memory Limit Register defines the upper address (inclusive) of a prefetchable memory address range that is used to determine when to forward memory transactions from one side of the bridge to the other. The Prefetchable Memory Limit Register must be programmed to a value greater than or equal to the PMBR before the *Memory Space Enable* bit of the Primary Command Register is set. If the value in the PMLR is not greater than or equal to the value of the PMBR once the *Memory Space Enable* bit is set, memory transactions on either side of the bridge will be indeterminate. The upper 12 bits correspond to **AD[31:20]** of 32-bit addresses. For the purposes of address decoding, the bridge assumes that **AD[19:0]**, the lower 20 bits of the memory limit address, are FFFFFH. This forces a 1 Mbyte granularity on the memory address range.

The VGA Enable bit in the BCR register forces the bridge to forward memory accesses in the address range from 0A0000H to 0BFFFFH from the Primary to Secondary and blocks addresses in the same range from Secondary to Primary.

Table 14-43. Prefetchable Memory Limit Register - PMLR



PCI Configuration Offset
26 - 27H

Intel® i960® Core Local Bus Address
0000 1026H

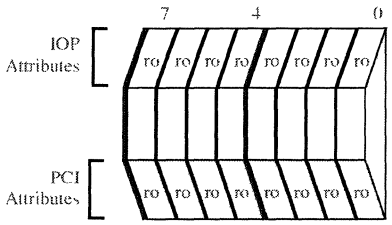
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
15:04	000H	Prefetchable Memory Limit Address - This field is programmed with **AD[31:20]** of the top of the memory address range to be passed down the hierarchy by the bridge.
03:00	0H	Reserved.

14.15.21 Capabilities Pointer Register - Cap_Ptr

The Capabilities Pointer Register bits adhere to the definitions in the *PCI Local Bus Specification*, Revision 2.2. This register provides an offset for the PCI Configuration Space in this function, for the location of the first item in the Capabilities linked list. In the case of the 80303 I/O processor, this is the PCI Bus Power Management extended capability as defined by the *PCI Bus Power Management Interface Specification*, Revision 1.1.

Table 14-44. Capabilities Pointer Register - Cap_Ptr



PCI Configuration Offset
34H

Intel® i960® Core Local Bus Address
0000 1034H

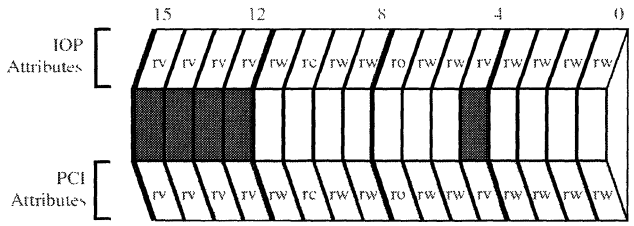
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
07:00	68H	Capability List Pointer - This provides an offset for the configuration space in this function, that points to the 80303 I/O processor PCI Bus Power Management extended capability.

14.15.22 Bridge Control Register - BCR

Bridge Control Register bits provide extensions to the Command Register that are specific to PCI-to-PCI bridges. The Bridge Control Register provides many of the same controls for the Secondary interface that are provided by the Command register for the Primary interface. Some bits affect the operation of both interfaces of the bridge.

Table 14-45. Bridge Control Register - BCR (Sheet 1 of 2)



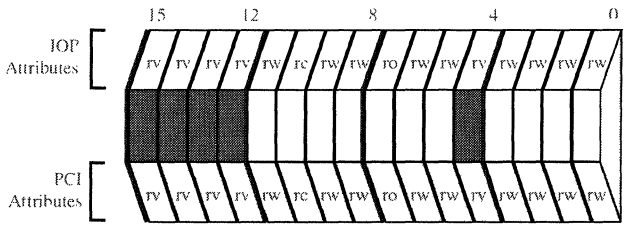
PCI Configuration Offset
3E - 3FH

Intel® i960® Core Local Bus Address
0000 103EH

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
15:12	0000_2	Reserved.
11	0_2	Discard Timer **SERR#** Enable - This bit enables the assertion of **P_SERR#** for all discard timers. A value of 0 indicates that **P_SERR#** is not asserted when any discard timer expires. A value of 1 indicates that **P_SERR#** is asserted (if enabled in the PCR) when a discard timer expires.
10	0_2	Discard Timer Status - This bit indicates the status of the discard timers. A value of zero indicates that no discard timers have expired. A value of 1 indicates that at least one of the eight discard timers has expired.
09	0_2	Secondary Discard Timer Value - This bit controls the time-out value for the four discard timers attached to the queues holding data for transactions initiated on the Secondary bus. A value of zero indicates the time-out value is 2^{15} clocks. A value of 1 indicates the time-out value is 2^{10} clocks.
08	0_2	Primary Discard Timer Value - This bit controls the time-out value for the four discard timers attached to the queues holding data for transactions initiated on the Primary bus. A value of 0 indicates the time-out value is 2^{15} clocks. A value of 1 indicates the time-out value is 2^{10} clocks.
07	0_2	Fast Back to Back Enable - This Secondary interface does not perform fast back to back transactions.
06	0_2	Secondary Bus Reset - This bit controls the Secondary bus **S_RST#** signal. When set: • The PCI-to-PCI Bridge Unit will reset all upstream and downstream Transaction Queues and Data Queues as well as the Secondary PCI bus interface. The Bridge PCI configuration registers are not reset. The Primary PCI bus interface will retry all transactions, except Type 0 configuration transactions, until this bit is cleared. • DMA Channel 2 will immediately halt any PCI transactions and gracefully complete any local bus transactions. It will then return to an idle state. DMA Channel 2 will not begin any new transfers until the Secondary Bus Reset bit is cleared. • Secondary ATU will immediately halt any PCI transactions and gracefully complete any local bus transactions. The i960 core processor will be released from back-off, if necessary. The Secondary ATU will not accept any new i960 core processor requests until the Secondary Bus Reset bit is cleared. The Secondary ATU configuration registers are reset. • An interrupt may be sent to the core processor based upon the setting of bit 3 in the SDER. When this bit is cleared, the **S_RST#** signal is deasserted. The software must clear this bit.

Table 14-45. Bridge Control Register - BCR (Sheet 2 of 2)



PCI Configuration Offset
3E - 3FH

Intel® i960® Core Local Bus Address
0000 103EH

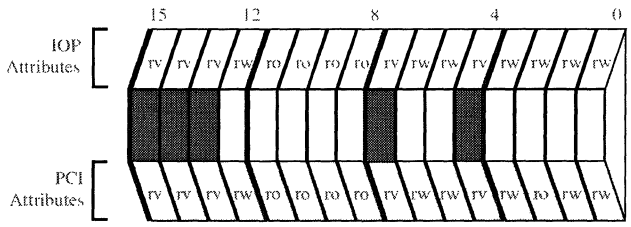
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
05	0_2	Master Abort Mode - This bit controls the bridge functionality whenever a Master-Abort termination occurs on either interface for transactions in which the bridge is the slave. When clear, reads will return all ones (32-bit or 64-bit depending on the PCI bus size of the initiating master and in the 64-bit bus case on **REQ64#/ACK64#**) and write data will be accepted by the bridge and discarded. When set, the bridge will signal a Master-Abort to the requesting master when the corresponding transaction on the other side of the bridge terminates with a Master-Abort and the transaction has not yet been concluded (reads and non-posted writes). When the bit is set and the transaction on the requesting interface has completed (posted writes) then the bridge must assert **P_SERR#** on the Primary interface (providing enabled in the PCR).
04	0_2	Reserved.
03	0_2	VGA Enable - Modifies the bridge response to VGA compatible addresses. When set to a '1', this bit indicates that a VGA device is on the Secondary PCI bus. Therefore, the bridge positively decodes and forwards the following transactions downstream regardless of the value of the MBR/MLR, PMBR/PMLR, and IOBR/IOLR registers. Memory transactions addressing: 000A0000h-000BFFFFh I/O transactions addressing: 3B0h-3BBh and 3C0h-3DFh In addition, the Secondary address decoder will block the forwarding of these I/O and Memory transactions from Secondary to Primary. When set to '0' the bridge will not forward VGA compatible memory and I/O addresses from the Secondary to the Primary PCI interface unless they are enabled for forwarding by the defined I/O and memory address ranges.
02	0_2	ISA Enable - This bit modifies the bridges response to ISA I/O addresses. This only applies to I/O addresses that are defined by the bridge in IOBR and IOLR and are also in the first 64 Kbytes of PCI address space (0000.0000H - 0000.FFFFH) When set, the bridge will not forward from Primary to Secondary and I/O transactions addressing the last 768 bytes in each 1 Kbyte block. In the opposite direction, I/O transactions will be forwarded up the bridge if the address the last 768 bytes in each 1 Kbyte block.
01	0_2	Secondary **SERR#** Enable - This bit controls the forwarding of Secondary interface **S_SERR#** assertions to the Primary interface. When this bit is set, If the **SERR#** Enable bit in the PCR register is set and the bridge detects the assertion of **S_SERR#** on the Secondary bus, it will then assert **P_SERR#** on the Primary interface. When clear, **S_SERR#** assertions are not forwarded to the Primary interface.
00	0_2	Secondary Parity Error Response Enable - This bit controls the response to parity errors on the Secondary interface. If this bit is clear, all address and data parity errors on the Secondary interface will be ignored. If this bit is set, detection and reporting of all parity errors on the Secondary interface is enabled. Correct parity must be generated even when parity error reporting is disabled.

14.15.23 Extended Bridge Control Register - EBCR

The Extended Bridge Control Register controls the extended functionality the bridge implements over the base *PCI-to-PCI Bridge Architecture Specification*, Revision 1.1.

Table 14-46. Extended Bridge Control Register - EBCR (Sheet 1 of 2)



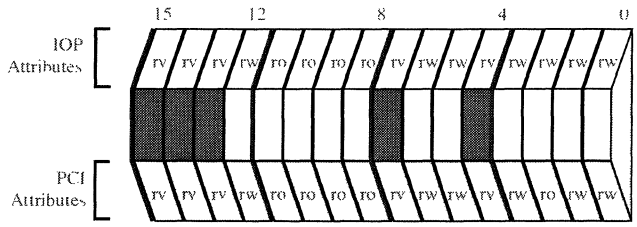
PCI Configuration Offset
40 - 41H

Intel® i960® Core Local Bus Address
0000 1040H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
15:13	0000_2	Reserved.
12		
11	Varies with external state of **S_M66EN** at Secondary PCI bus reset	Secondary Bus Operating at 66 MHz - When set, the Secondary interface has been initialized to function at 66 MHz by the assertion of **S_M66EN** during bus initialization. When clear, the Secondary interface has been initialized as a 33 MHz bus.
10	Varies with external state of **P_M66EN** at Primary PCI bus reset	Primary Bus Operating at 66 MHz - When set, the Primary interface has been initialized to function at 66 MHz by the assertion of **P_M66EN** during bus initialization. When clear, the Primary interface has been initialized as a 33 MHz bus.
09	Varies with external state of **S_REQ64#** at Secondary PCI bus reset	Secondary PCI Bus 64-Bit Capable - When clear, the Secondary PCI bus interface has been configured as 64-bit capable by the assertion of **S_REQ64#** on the rising edge of **S_RST#**. When set, the Secondary PCI interface is configured as 32-bit only.
08	Varies with external state of **P_REQ64#** at Primary PCI bus reset	Primary PCI Bus 64-Bit Capable - When clear, the Primary PCI bus interface has been configured as 64-bit capable by the assertion of **P_REQ64#** on the rising edge of **P_RST#**. When set, the Primary PCI interface is configured as 32-bit only.
07	0_2	Reserved.
06	0_2	Secondary DAC Medium Decode Enable - When set, DAC cycles on the Secondary PCI interface of the bridge will be claimed by the bridge and forwarded to the Primary PCI interface with medium decode timing. When clear, all DAC cycles on the Secondary PCI interface will be claimed with subtractive decode timing and forwarded to the Primary PCI interface.

Table 14-46. Extended Bridge Control Register - EBCR (Sheet 2 of 2)



PCI Configuration Offset
40 - 41H

Intel® i960® Core Local Bus Address
0000 1040H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

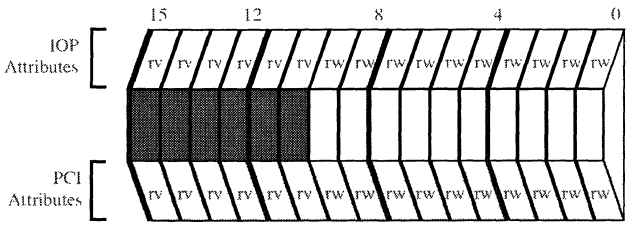
Bit	Default	Description
05	0_2	Reset Internal Bus - This bit controls the reset of the i960 core processor and all units on the internal bus. When set: • The PCI-to-PCI Bridge Unit is not reset. Upstream and downstream bridge I/O and memory transactions are unaffected during the IB reset. • All current PCI transactions being mastered by the ATU and DMA will complete, and the ATU and DMA master interfaces will proceed to an idle state. No additional transactions will be mastered by these units until the IB reset is complete. • All current transactions being slaved by the ATU on either the PCI bus or the internal bus will complete, and the ATU slave interfaces will proceed to an idle state. All future slave transactions will master abort, with the exception of the completion cycle for the transaction that set the Reset Internal Bus bit in the EBCR. • If the value of the Core Processor Reset bit in the EBCR (upon normal reset) is set, the i960 core processor will be held in reset when the IB reset is complete. • The Bridge and the ATU will ignore configuration cycles, and they will appear as master aborts for: 32 Internal Bus clocks + the number of Internal Bus clocks needed to finish all ATU and DMA transactions that will complete before the IB reset (as described in the above text). • The 80303 I/O processor hardware will clear this bit after the reset operation completes.
04	0_2	Reserved.
03	1_2	Upstream Prefetchable Memory Enable - When this bit is set, the Bridge assumes that upstream Memory Read commands are to prefetchable memory. When this bit is clear, the Bridge assumes that upstream Memory Read commands are to non-prefetchable memory. (Modifying this bit, while the bridge is enabled may cause unknown behavior.)
02	Varies with external state of **RETRY** pin at Primary PCI bus reset	Configuration Cycle Retry - When this bit is set, the Primary PCI interface of the 80303 I/O processor will respond to all configuration cycles with a Retry condition. When clear, the 80303 I/O processor will respond to the appropriate configuration cycles. The default condition for this bit is based on the external state of the **RETRY** pin at the rising edge of **P_RST#**. If the external state of the pin is high, the bit is set. If the external state of the pin is low, the bit is cleared.
01	Varies with external state of **RST_MODE#** pin at Primary PCI bus reset	Core Processor Reset - This bit is set to its default value by the hardware when either P_RST# is asserted or the Reset Local Bus bit in the EBCR is set. When this bit is set, the i960 core processor is being held in reset. Software cannot set this bit. Software will be required to clear this bit to deassert i960 processor reset. The default condition for this bit is based on the external state of the **RST_MODE#** pin at the rising edge of **P_RST#**. If the external state of the pin is low, the default value of this bit is set. If the external state of the pin is high, the default value of this bit is clear.
00	0_2	Posting Disable - If this bit is set, the bridge is not allowed to post write transactions from either bridge interface. All memory write transactions are processed as Delayed Write transactions. If this bit is clear, the bridge is allowed to post write transactions. (Modifying this bit, while the bridge is enabled may cause unknown behavior.)

14.15.24 Secondary IDSEL Select Register - SISR

The Secondary IDSEL Select Register controls the usage of **S_AD[25:16]** in Type 1 to Type 0 conversions from the Primary to Secondary interface. In default operation, a unique encoding on Primary addresses **P_AD[15:11]** results in the assertion of one bit on the Secondary address bus **S_AD[31:16]** during a Type 1 to Type 0 conversion (See Section 14.4.2.). This is used for the assertion of **IDSEL** on the device being targeted by the Type 0 configuration command. This register allows Secondary address bits **S_AD[25:16]** to be used to configure private PCI devices by forcing Secondary address bits **S_AD[25:16]** to all zeros during Type 1 to Type 0 conversions, regardless of the state of Primary addresses **P_AD[15:11]** (device number in Type 1 configuration command).

If any address bit within **S_AD[25:16]** is to be used for private Secondary PCI devices, the i960 core processor must guarantee that the corresponding bit in the SISR register is set before the host tries to configure the hierarchical PCI buses.

Table 14-47. Secondary IDSEL Select Register - SISR (Sheet 1 of 2)



PCI Configuration Offset
42 - 43H

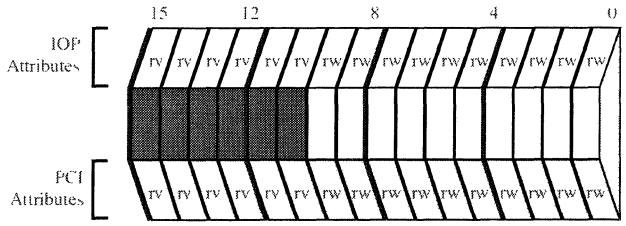
Intel® i960® Core Local Bus Address
0000 1042H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set

RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
15:10	000000_2	Reserved.
09	0_2	**AD16 - IDSEL** Disable - When this bit is set, **AD16** will be deasserted for any possible Type 1 to Type 0 conversion. When clear, **AD16** will be asserted when Primary addresses **AD[15:11]** = 00000_2 during a Type 1 to Type 0 conversion.
08	0_2	**AD17 - IDSEL** Disable - When this bit is set, **AD17** will be deasserted for any possible Type 1 to Type 0 conversion. When clear, **AD17** will be asserted when Primary addresses **AD[15:11]** = 00001_2 during a Type 1 to Type 0 conversion.
07	0_2	**AD18 - IDSEL** Disable - When this bit is set, **AD18** will be deasserted for any possible Type 1 to Type 0 conversion. When clear, **AD18** will be asserted when Primary addresses **AD[15:11]** = 00010_2 during a Type 1 to Type 0 conversion.
06	0_2	**AD19 - IDSEL** Disable - When this bit is set, **AD19** will be deasserted for any possible Type 1 to Type 0 conversion. When clear, **AD19** will be asserted when Primary addresses **AD[15:11]** = 00011_2 during a Type 1 to Type 0 conversion.
05	0_2	**AD20 - IDSEL** Disable - When this bit is set, **AD20** will be deasserted for any possible Type 1 to Type 0 conversion. When clear, **AD20** will be asserted when Primary addresses **AD[15:11]** = 00100_2 during a Type 1 to Type 0 conversion.
04	0_2	**AD21 - IDSEL** Disable - When this bit is set, **AD21** will be deasserted for any possible Type 1 to Type 0 conversion. When clear, **AD21** will be asserted when Primary addresses **AD[15:11]** = 00101_2 during a Type 1 to Type 0 conversion.
03	0_2	**AD22 - IDSEL** Disable - When this bit is set, **AD22** will be deasserted for any possible Type 1 to Type 0 conversion. When clear, **AD22** will be asserted when Primary addresses **AD[15:11]** = 00110_2 during a Type 1 to Type 0 conversion.

Table 14-47. Secondary IDSEL Select Register - SISR (Sheet 2 of 2)



PCI Configuration Offset
42 - 43H

Intel® i960® Core Local Bus Address
0000 1042H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set

RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
02	0_2	**AD23** - **IDSEL** Disable - When this bit is set, **AD23** will be deasserted for any possible Type 1 to Type 0 conversion. When clear, **AD23** will be asserted when Primary addresses **AD[15:11]** = 00111_2 during a Type 1 to Type 0 conversion.
01	0_2	**AD24**- **IDSEL** Disable - When this bit is set, **AD24** will be deasserted for any possible Type 1 to Type 0 conversion. When clear, **AD24** will be asserted when Primary addresses **AD[15:11]** = 01000_2 during a Type 1 to Type 0 conversion.
00	0_2	**AD25**- **IDSEL** Disable - When this bit is set, **AD25** will be deasserted for any possible Type 1 to Type 0 conversion. When clear, **AD25** will be asserted when Primary addresses **AD[15:11]** = 01001_2 during a Type 1 to Type 0 conversion.

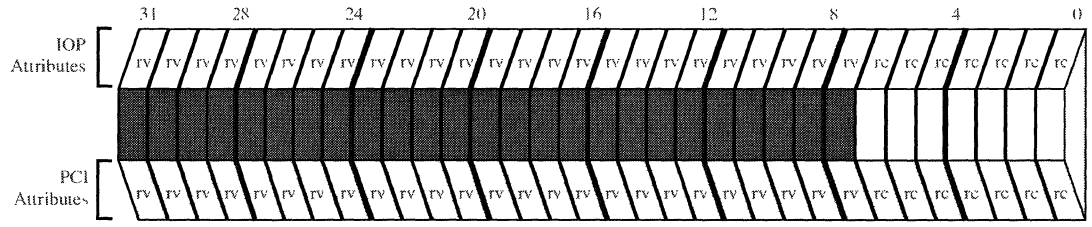
14.15.25 Primary Bridge Interrupt Status Register - PBISR

The Primary Bridge Interrupt Status Register notifies the i960 core processor of the source of a Primary Bridge interface interrupt. In addition, this register is written to clear the source of the interrupt to the interrupt unit of the 80303 I/O processor (see Section 8.4, "Intel® 80303 I/O Processor Interrupts" on page 8-21). All bits in this register are Read Only from PCI and Read/Clear from the local bus.

Bits 5:0 are a direct reflection of bits 15:11 and bit 8 (respectively) of the Primary Status Register (these bits are set at the same time by hardware but need to be cleared independently). The conditions that result in a Primary Bridge interrupt to the core processor are cleared by writing a "1" to the appropriate bits in this register.

The individual setting of the bits within the PBISR can be masked through the bits 10:6, bit 4 and bit 1 of the SDER. Refer to Section 14.15.34 for details.

Table 14-48. Primary Bridge Interrupt Status Register - PBISR



PCI Configuration Offset
44 - 47H

Intel® i960® Core Local Bus Address
0000 1044H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:07	0000000H	Reserved.
06	0_2	Power State Transition - When the Power State Field of the Power Management Control/Status Register is written to transition the Bridge function Power State from either D3 to D0 or D0 to D3 and the Power State Transition Interrupt mask is cleared, this bit is set.
05	0_2	Detected Parity Error - This bit is set when a parity error is detected during a data transfer on the Primary bus even if parity handling is disabled. Set under the following conditions: • Write Data Parity Error when the Primary interface of the Bridge is a slave (downstream write). • Read Data Parity Error when the Primary interface of the Bridge is a master (upstream read). • Any Address Parity Error on the Primary Bus (including one generated by the Primary interface of the Bridge).
04	0_2	**P_SERR#** Asserted - This bit is set if **P_SERR#** is asserted on the Primary PCI bus.
03	0_2	PCI Master Abort - This bit is set whenever a transaction initiated by the Primary master interface ends in a Master-Abort.
02	0_2	PCI Target Abort (Master) - This bit is set whenever a transaction initiated by the Primary master interface ends in a Target-Abort.
01	0_2	PCI Target Abort (Target) - This bit is set whenever the Primary interface, acting as a target, terminates the transaction on the PCI bus with a Target-Abort.
00	0_2	PCI Master Parity Error - The Primary interface sets this bit when three conditions are met: 1. the bus agent asserted **P_PERR#** itself or observed **P_PERR#** asserted 2. the agent setting the bit acted as the bus master for the operation in which the error occurred 3. the Parity Checking Enable bit (PCR Register) is set

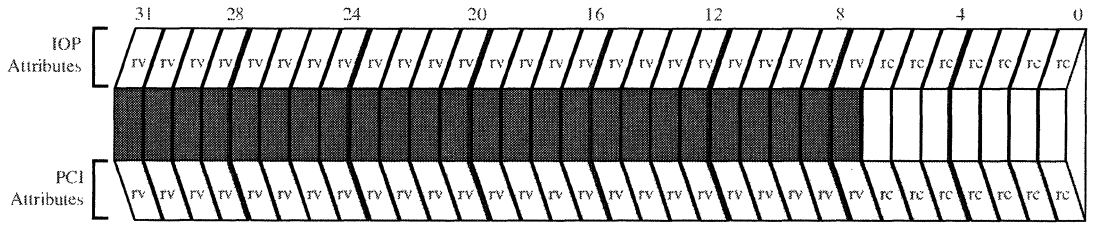
14.15.26 Secondary Bridge Interrupt Status Register - SBISR

The Secondary Bridge Interrupt Status Register notifies the i960 core processor of the source of a Secondary Bridge interface interrupt. In addition, this register is written to clear the source of the interrupt to the interrupt unit of the 80303 I/O processor (see Section 8.4, "Intel® 80303 I/O Processor Interrupts" on page 8-21). All bits in this register are Read/Clear from the PCI bus and the local bus.

Bits 5:0 are a direct reflection of bits 15:11 and bit 8 (respectively) of the Secondary Status Register (these bits are set at the same time by hardware but need to be cleared independently). Bit 6 is set when software sets and subsequently clears the Secondary Bus Reset bit in the BCR. The conditions that result in a Secondary Bridge interrupt are cleared by writing a "1" to the appropriate bits in this register.

The individual setting of the bits within the SBISR can be masked through the bits 3, 15:11, and 5 of the SDER. Refer to Section 14.15.34 for details.

Table 14-49. Secondary Bridge Interrupt Status Register - SBISR



PCI Configuration Offset: 48 - 4BH

Intel® i960® Core Local Bus Address: 0000 1048H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:07	0000000H	Reserved.
06	0_2	Secondary Bus Reset Occurred - This bit is set when the bridge senses the deassertion (by software only) of bit 6, Secondary Bus Reset, in the BCR.
05	0_2	Detected Parity Error - This bit is set when a parity error is detected during a data transfer on the Secondary bus even if parity handling is disabled. Set under the following conditions: • Write Data Parity Error when the Secondary interface of the Bridge is a slave (upstream write). • Read Data Parity Error when the Secondary interface of the Bridge is a master (downstream read). • Any Address Parity Error on the Secondary Bus (including one generated by the Secondary interface of the Bridge).
04	0_2	Received System Error - This bit is set if **S_SERR#** is detected on the Secondary PCI bus.
03	0_2	PCI Master Abort - This bit is set whenever a transaction initiated by the Secondary master interface ends in a Master-Abort.
02	0_2	PCI Target Abort (Master) - This bit is set whenever a transaction initiated by the Secondary master interface ends in a Target-Abort.
01	0_2	PCI Target Abort (Target) - This bit is set whenever the Secondary interface, acting as a target, terminates the transaction on the PCI bus with a Target-Abort.
00	0_2	PCI Master Parity Error - The Secondary interface sets this bit when three conditions are met: 1. the bus agent asserted **S_PERR#** itself or observed **S_PERR#** asserted 2. the agent setting the bit acted as the bus master for the operation in which the error occurred 3. the Secondary Parity Error Response bit (BCR Register) is set

14.15.27 Secondary Arbitration Control Register - SACR

Refer to Section 17.6.1, "Secondary Arbitration Control Register - SACR" on page 17-14 for a description of the Secondary Arbitration Control Register.

14.15.28 PCI Interrupt Routing Select Register - PIRSR

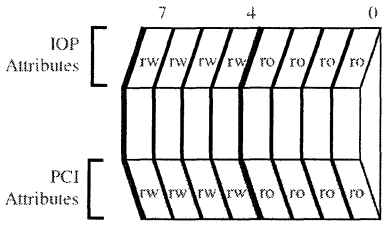
Refer to Section 8.7.4, "PCI Interrupt Routing Select Register - PIRSR" on page 8-41 for a description of the PCI Interrupt Routing Select Register.

14.15.29 Secondary I/O Base Register - SIOBR

Secondary I/O Base Register bits are used when the Secondary PCI interface is enabled for private addressing. The Secondary I/O Base Register defines the bottom address (inclusive) of a positively decoded address range that is used to determine when to not forward I/O transactions from the Secondary interface to the Primary interface of the bridge. It must be programmed with a valid value before the Private Address Space Enable bit is set. The bridge only supports 16-bit addressing which is indicated by a value of 0H in the four least significant bits of the register. The upper 4 bits are programmed with **S_AD[15:12]** for the bottom of the address range. **S_AD[11:0]** of the base address is always 000H forcing the Secondary I/O address range to be 4 Kbyte aligned.

For the purposes of address decoding, the bridge assumes that **S_AD[31:16]**, the upper 16 address bits of the I/O address, are zero. The bridge must still perform the address decode on the full 32 bits of address per PCI Local Bus Specification and check that the upper 16 bits are equal to 0000H.

Table 14-50. Secondary I/O Base Register - SIOBR



PCI Configuration Offset
54H

Intel® i960® Core Local Bus Address
0000 1054H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

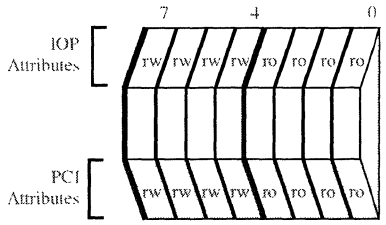
Bit	Default	Description
07:04	0H	Secondary I/O Base Address - This field is programmed with **S_AD[15:12]** of the bottom of the private Secondary I/O address range not passed from the Secondary to the Primary side of the bridge due to a private I/O range.
03:00	0H	I/O Addressing Capability - The value of 0H signifies that the bridge only supports 16-bit I/O addressing.

14.15.30 Secondary I/O Limit Register - SIOLR

Secondary I/O Limit Register bits are used when the Secondary PCI interface is enabled for private address decoding. The Secondary I/O Limit Register defines the upper address (inclusive) of a decoded Secondary address range that is used to determine when to not forward I/O transactions from the Secondary to Primary interface of the bridge. The bridge only supports 16-bit addressing which is indicated by a value of 0H in the four least significant bits of the register. The upper 4 bits are programmed with **S_AD[15:12]** for the top of the address range. **S_AD[11:0]** of the base address is always FFFH forcing a 4 Kbyte I/O range granularity.

For the purposes of address decoding, the bridge assumes that **S_AD[31:16]**, the upper 16 address bits of the I/O address, are zero. The bridge must still perform the address decode on the full 32 bits of address per PCI Local Bus Specification and check that the upper 16 bits are equal to 0000H.

Table 14-51. Secondary I/O Limit Register - SIOLR



PCI Configuration Offset
55H

Intel® i960® Core Local Bus Address
0000 1055H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

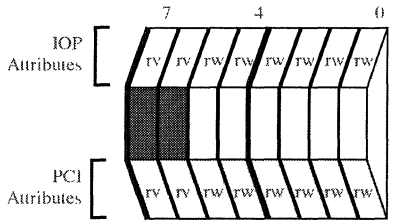
Bit	Default	Description
07:04	0H	Secondary I/O Limit Address - This field is programmed with **S_AD[15:12]** of the top of the private I/O address range not passed from the Secondary to Primary interface.
03:00	0H	Secondary I/O Addressing Capability - The value of 0H signifies that the bridge only supports 16-bit I/O addressing.

14.15.31 Secondary Clock Disable Register - SCDR

The BDG unit provides six Secondary PCI output clocks (**S_CLKOUT[5:0]**). Upon assertion of **P_RST#,** all of the six secondary output clocks are enabled.

The Secondary Clock Disable Register provides the ability to selectively disable unused secondary output clocks following deassertion of **P_RST#.**

Table 14-52. Secondary Clock Disable Register - SCDR



PCI Configuration Offset
56H

Intel® i960® Core Local Bus Address
0000 1056H

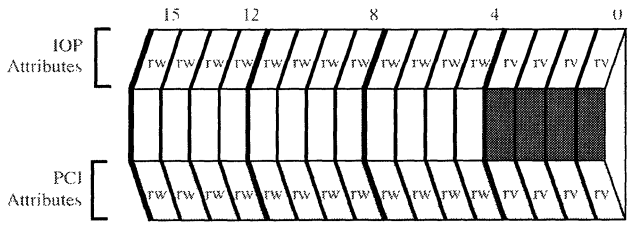
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
07:06	00_2	Reserved.
05	0_2	Disable **S_CLKOUT5** -- When set, this bit disables the operation of the **S_CLKOUT5** secondary clock output. Instead of toggling, the output will be driven to logic 0.
04	0_2	Disable **S_CLKOUT4** -- When set, this bit disables the operation of the **S_CLKOUT4** secondary clock output. Instead of toggling, the output will be driven to logic 0.
03	0_2	Disable **S_CLKOUT3** -- When set, this bit disables the operation of the **S_CLKOUT3** secondary clock output. Instead of toggling, the output will be driven to logic 0.
02	0_2	Disable **S_CLKOUT2** -- When set, this bit disables the operation of the **S_CLKOUT2** secondary clock output. Instead of toggling, the output will be driven to logic 0.
01	0_2	Disable **S_CLKOUT1** -- When set, this bit disables the operation of the **S_CLKOUT1** secondary clock output. Instead of toggling, the output will be driven to logic 0.
00	0_2	Disable **S_CLKOUT0** -- When set, this bit disables the operation of the **S_CLKOUT0** secondary clock output. Instead of toggling, the output will be driven to logic 0.

14.15.32 Secondary Memory Base Register - SMBR

Secondary Memory Base Register bits are used to define a private address space on the Secondary PCI bus if the Private Address Space Enable bit in the SDER is set. The Secondary Memory Base Register defines the bottom address (inclusive) of a memory-mapped address range that is used to determine when to not forward transactions from the Secondary to Primary interface. The Secondary Memory Base Register must be programmed with a valid value before the Private Address Space Enable bit in the SDER is set. The upper 12 bits correspond to **S_AD[31:20]** of 32-bit addresses. For the purposes of address decoding, the bridge assumes that **S_AD[19:0]**, the lower 20 address bits of the memory base address, are zero. This means that the bottom of the defined address range will be aligned on a 1 Mbyte boundary.

Table 14-53. Secondary Memory Base Register - SMBR



PCI Configuration Offset
58 - 59H

Intel® i960® Core Local Bus Address
0000 1058H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set

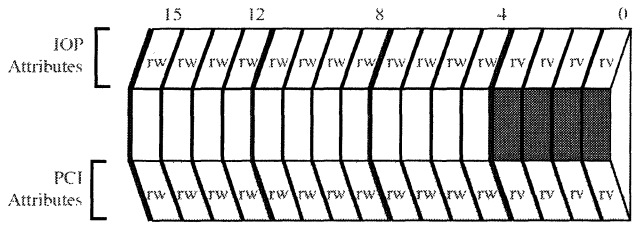
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
15:04	000H	Secondary Memory Base Address - This field is programmed with **S_AD[31:20]** of the bottom of the Secondary memory address range that is not passed from the Secondary to Primary interface when private address space is enabled.
03:00	0H	Reserved.

14.15.33 Secondary Memory Limit Register - SMLR

Secondary Memory Limit Register bits are used when the Secondary interface of the bridge unit is enabled for private address decoding. The Secondary Memory Limit Register defines the upper address (inclusive) of a memory-mapped address range that is used to determine when to not forward transactions from the Secondary to Primary interface. The Secondary Memory Limit Register must be programmed to a value greater than or equal to the SMBR before private address space is enabled. If the value in the SMLR is not greater than or equal to the value of the SMBR once the Private Address Enable bit is set, the private address range is indeterminate and will not function. The upper 12 bits correspond to **S_AD[31:20]** of 32-bit addresses. For the purposes of address decoding, the bridge assumes that **S_AD[19:0]**, the lower 20 address bits of the Secondary memory base address, are FFFFFH. This forces a 1 Mbyte granularity on the memory address range.

Table 14-54. Secondary Memory Limit Register - SMLR



PCI Configuration Offset
5A - 5BH

Intel® i960® Core Local Bus Address
0000 105AH

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
15:04	000H	Secondary Memory Limit Address - This field is programmed with **S_AD[31:20]** of the top of the Secondary memory address range that is not forwarded from Secondary to Primary side due to a private address space.
03:00	0H	Reserved.

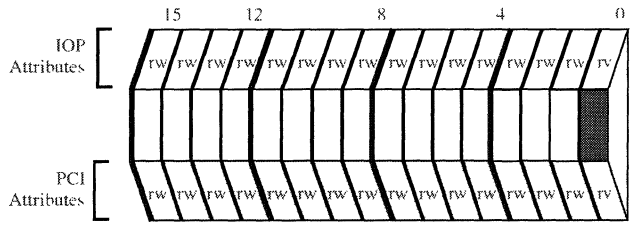
14.15.34 Secondary Decode Enable Register - SDER

The Secondary Decode Enable Register has two separate functions. It is used to control the address decode functions on the Secondary PCI interface of the bridge unit and in addition contains the control bits capable of masking the Primary and Secondary bridge interface interrupt sources to the i960 core processor.

The Private Memory Space Enable bit allows a private memory and I/O space to be created on the Secondary PCI bus. This bit is used in conjunction with the SMBR/SMLR and the SIOBR/SIOLR registers. If this bit is set, transactions with addresses within the memory and I/O address ranges are ignored by the bridge. It also disables Secondary positive decode.

The interrupt mask bits are responsible for masking interrupt conditions to the Primary and Secondary Bridge Interrupt Status Registers (PBISR and SBISR). Masking the bits in the PBISR and SBISR prevents the setting of the Primary Bridge PCI Interface Error Interrupt Bit and the Secondary Bridge PCI Interface Error Bit in the NMI Interrupt Status Register (see Chapter 8, "PCI and Peripheral Interrupt Controller Unit". The setting of a mask bit means that an error condition which results in the setting of an error response bit in the PSR or SSR does not set the corresponding bit in the PBISR or SBISR.

Table 14-55. Secondary Decode Enable Register - SDER (Sheet 1 of 2)



PCI Configuration Offset
5C - 5DH

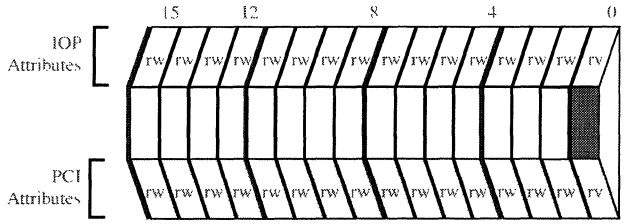
Intel® i960® Core Local Bus Address
0000 105CH

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set

RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
15	1_2	**S_SERR#** Detected Interrupt Mask - When set, detecting **S_SERR#** on the Secondary interface resulting in bit 14 of the SSR being set will *not* result in bit 4 of the SBISR being set. When clear, an error that sets bit 14 of the SSR will cause bit 4 of the SBISR to be set
14	1_2	Secondary PCI Master Abort Interrupt Mask - When set, a master abort error resulting in bit 13 of the SSR being set will *not* result in bit 3 of the SBISR being set. When clear, an error that sets bit 13 of the SSR will cause bit 3 of the SBISR to be set.
13	1_2	Secondary PCI Target Abort (Master) Interrupt Mask- When set, a target abort error resulting in bit 12 of the SSR being set will *not* result in bit 2 of the SBISR being set. When clear, an error that sets bit 12 of the SSR will cause bit 2 of the SBISR to be set.
12	1_2	Secondary PCI Target Abort (Target) Interrupt Mask - When set, a target abort error resulting in bit 11 of the SSR being set will *not* result in bit 1 of the SBISR being set. When clear, an error that sets bit 11 of the SSR will cause bit 1 of the SBISR to be set.
11	1_2	Secondary PCI Master Parity Error Interrupt Mask - When set a parity error resulting in bit 8 of the SSR being set will *not* result in bit 0 of the SBISR being set. When clear, an error that sets bit 8 of the SSR will cause bit 0 of the SBISR to be set.
10	1_2	**P_SERR#** Asserted Interrupt Mask - When set, detecting or asserting **P_SERR#** on the Primary interface resulting in bit 14 of the PSR being set will *not* result in bit 4 of the PBISR being set. When clear, an error that sets bit 14 of the PSR will cause bit 4 of the PBISR to be set

Table 14-55. Secondary Decode Enable Register - SDER (Sheet 2 of 2)



PCI Configuration Offset
5C - 5DH

Intel® i960® Core Local Bus Address
0000 105CH

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set

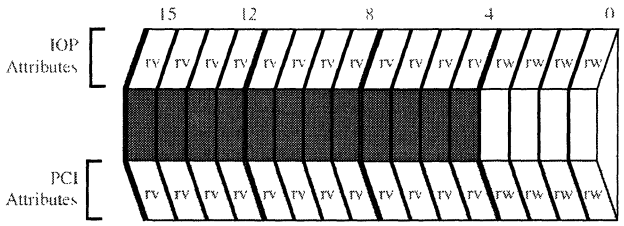
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
09	1_2	Primary PCI Master Abort Interrupt Mask - When set, a master abort error resulting in bit 13 of the PSR being set will *not* result in bit 3 of the PBISR being set. When clear, an error that sets bit 13 of the PSR will cause bit 3 of the PBISR to be set.
08	1_2	Primary PCI Target Abort (Master) Interrupt Mask- When set, a target abort error resulting in bit 12 of the PSR being set will *not* result in bit 2 of the PBISR being set. When clear, an error that sets bit 12 of the PSR will cause bit 2 of the PBISR to be set.
07	1_2	Primary PCI Target Abort (Target) Interrupt Mask - When set, a target abort error resulting in bit 11 of the PSR being set will *not* result in bit 1 of the PBISR being set. When clear, an error that sets bit 11 of the PSR will cause bit 1 of the PBISR to be set.
06	1_2	Primary PCI Master Parity Error Interrupt Mask - When set a parity error resulting in bit 8 of the PSR being set will *not* result in bit 0 of the PBISR being set. When clear, an error that sets bit 8 of the PSR will cause bit 0 of the PBISR to be set.
05	1_2	Secondary Detected Parity Error Bit Interrupt Mask - When set a parity error resulting in bit 15 of the SSR being set will not result in bit 5 of the SBISR being set.
04	1_2	Primary Detected Parity Error Bit Interrupt Mask - When set a parity error resulting in bit 15 of the PSR being set will not result in bit 5 of the PBISR being set.
03	1_2	Secondary Bus Reset Occurred Interrupt Mask - When this bit is set, and the bridge senses the deassertion (by software only) of bit 6, Secondary Bus Reset, in the BCR, bit 6 of the SBISR will not be set.
02	0_2	Private Memory Space Enable - when set, this bit disables Bridge forwarding of addresses in the SMBR/SMLR and SIOBR/SIOLR address ranges. This creates a private memory space on the Secondary PCI bus that allows peer to peer transactions.
01	1_2	Power State Transition Interrupt Mask - When this bit is set and the Power Management Control/Status Register is written to transition the Bridge Function Power State from either D0 to D3 or D3 to D0, bit 6 of the PBISR is not set.
00	0_2	Reserved.

14.15.35 Queue Control Register - QCR

The Queue Control Register contains programmable parameters affecting operation of the PCI-to-PCI Bridge Queues.

Table 14-56. Queue Control Register- QCR



PCI Configuration Offset
5E - 5FH

Intel® i960® Core Local Bus Address
0000 105EH

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set

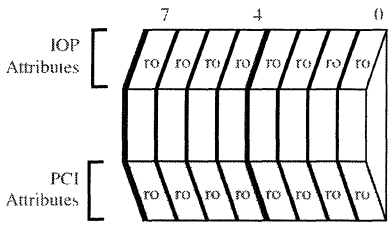
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
15:04	000H	Reserved.
03	0_2	DRC Alias - when set, the bridge does not distinguish read commands in prefetchable address space when attempting to match a current PCI read transaction with read data enqueued within a DRC buffer. When clear, a current read transaction must have the exact same read command as the DRR for the bridge to deliver DRC data. (Modifying this bit, while the bridge is enabled may cause unknown behavior.)
02	0_2	MWI Alias - when set, the target interface of the bridge treats an MWI as a Memory Write and aliases the MWI to a Memory Write on the target bus.
01	0_2	Upstream MRL Prefetch Size - when set, the maximum prefetch data size attempted by the Primary interface of the bridge, as a master, is 128 bytes. When clear, the maximum prefetch size is 32 bytes. Refer to Table 14-13 for the exact prefetch data sizes during MRL transactions.
00	0_2	Downstream MRL Prefetch Size - when set, the maximum prefetch data size attempted by the Secondary interface of the bridge, as a master, is 64 bytes. When clear, the maximum prefetch size is 32 bytes. Refer to Table 14-12 for the exact prefetch data sizes during MRL transactions.

14.15.36 Capability Identifier Register - Cap_ID

The Capability Identifier Register bits adhere to the definitions in the *PCI Local Bus Specification*, Revision 2.2. This register in the PCI Extended Capability header identifies the type of Extended Capability contained in that header. In the case of the 80303 I/O processor, this is the PCI Bus Power Management extended capability with an ID of 01H as defined by the *PCI Bus Power Management Interface Specification*, Revision 1.1.

Table 14-57. Capability Identifier Register - Cap_ID



PCI Configuration Offset: 68H

Intel® i960® Core Local Bus Address: 0000 1068H

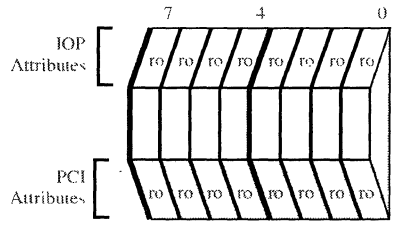
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
07:00	01H	Cap_Id - This field with its 01H value identifies this item in the linked list of Extended Capability Headers as being the PCI Power Management Registers.

14.15.37 Next Item Pointer Register - Next_Item_Ptr

The Next Item Pointer Register bits adhere to the definitions in the *PCI Local Bus Specification*, Revision 2.2. This register describes the location of the next item in the function capability list. For the 80303 I/O processor, the Power Management Registers are the only Extended Capability Supported, thus the Next_Item_Ptr is set to 00H, indicating the end of the Capabilities List.

Table 14-58. Next Item Pointer Register - Next_Item_Ptr



PCI Configuration Offset
69H

Intel® i960® Core Local Bus Address
0000 1069H

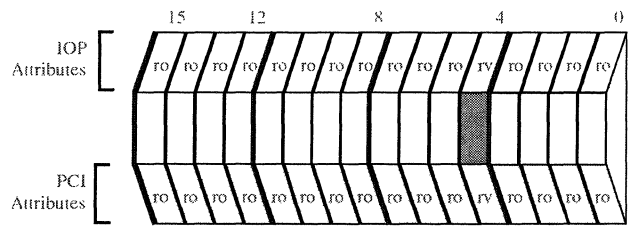
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
07:00	00H	Next_ Item_ Pointer - This field provides an offset into the function configuration space pointing to the next item in the function capability list. Since there are no other Extended Capabilities besides PCI Power Management in the 80303 I/O processor, the register is set to 00H.

14.15.38 Power Management Capabilities Register - PMCR

Power Management Capabilities bits adhere to the definitions in the *PCI Bus Power Management Interface Specification*, Revision 1.1. This register is a 16-bit read-only register which provides information on the capabilities of the bridge function related to power management.

Table 14-59. Power Management Capabilities Register - PMCR



PCI Configuration Offset
6A- 6BH

Intel® i960® Core Local Bus Address
0000 106AH

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set

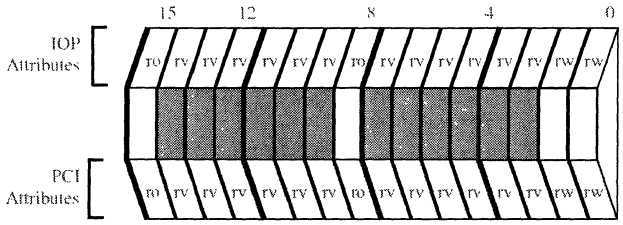
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
15:11	00000_2	PME_Support - This function is not capable of asserting the PME# signal in any state, since PME# is not supported by the 80303 I/O processor, thus the value of PME_Support is set to 00000B.
10	0_2	Support - This bit is set to 0_2 indicating that the 80303 I/O processor does not support the D2 Power Management State
9	0_2	Support - This bit is set to 0_2 indicating that the 80303 I/O processor does not support the D1 Power Management State
8:6	000_2	Aux_Current - This field is set to 000_2 indicating that the 80303 I/O processor has no current requirements for the 3.3Vaux signal as defined in the *PCI Bus Power Management Interface Specification*, Revision 1.1
5	0_2	DSI - This field is set to 0_2 meaning that this function will not require a device specific initialization sequence following the transition to the D0 uninitialized state.
4	0_2	Reserved.
3	0_2	PME Clock - Since the 80303 I/O processor does not support PME# signal generation, this value is set to 0_2
2:0	010_2	Version - Setting these bits to 010_2 means that this function complies with *PCI Bus Power Management Interface Specification*, Revision 1.1

14.15.39 Power Management Control/Status Register - PMCSR

Power Management Control/Status bits adhere to the definitions in the *PCI Bus Power Management Interface Specification*, Revision 1.1. This 16-bit register is the primary control and status interface for the power management extended capability.

Table 14-60. Power Management Control/Status Register - PMCSR



PCI Configuration Offset
6C- 6DH

Intel® i960® Core Local Bus Address
0000 106CH

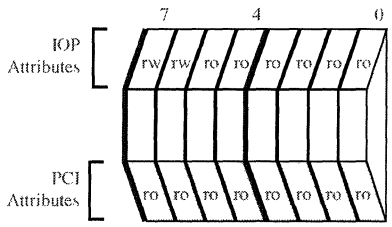
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
15	0_2	PME_Status - This function is not capable of asserting the PME# signal in any state, since PME# is not supported by the 80303 I/O processor, thus the value of PME_Status is set to 0_2 and is read-only.
14:9	00H	Reserved.
8	0_2	PME_En - This bit is hardwired to read-only 0_2 since this function does not support PME# generation from any power state.
7:2	000000_2	Reserved.
1:0	00_2	Power State - This 2-bit field is used both to determine the current power state of a function and to set the function into a new power state. The definition of the values is: 00_2 - D0 01_2 - D1 (Unsupported) 10_2 - D2 (Unsupported) 11_2 - $D3_{hot}$ The 80303 I/O processor supports only the D0 and $D3_{hot}$ states.

14.15.40 PMCSR PCI-to-PCI Bridge Support - PMCSR_BSE

This register supports Bridge specific Power Management Control/Status functionality and is required for all PCI-to-PCI bridges.

Table 14-61. PMCSR PCI-to-PCI Bridge Support - PMCSR_BSE



PCI Configuration Offset
6EH

Intel® i960® Core Local Bus Address
0000 106EH

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
07	0_2	BPCC_En (Bus Power/Clock Control Enable) - If this bit is set the Power State field in the PMCSR register can be used to control the Secondary bus Power/Clock.
06	0_2	B2_B3# (B2/B3 support for $D3_{hot}$) - If BPCC_EN (bit 07) is set, the state of this bit determines the action that is to occur as a direct result of programming the Power State field of the PMCSR from D0 to $D3_{hot}$. If this bit is set, the Secondary bus PCI clock will be stopped (B2) when the Power State field of the PMCSR is programmed to $D3_{hot}$ and BPCC_EN is set. If this bit is cleared, the Secondary bus will have its power removed (B3) when the Power State field of the PMCSR is programmed to $D3_{hot}$ and BPCC_EN is set.
5:0	000000_2	Reserved.

PCI Address Translation Unit 15

This chapter describes the operation modes, setup, and implementation of the mechanism which interfaces between the primary and secondary PCI busses and the Intel® 80303 I/O processor internal bus.

15.1 Overview

As indicated in Figure 15-1, the ATU — the interface between the PCI bus and the on-chip internal bus — consists of two address translation units, the Expansion ROM Unit and the Messaging Unit (MU) described in Chapter 16, "Messaging Unit"

The ATUs support both inbound and outbound address translation. The ATUs are:

- Primary ATU (PATU) — provides access between the primary PCI bus and the 80303 I/O Processor Internal Bus. The primary ATU and Messaging Unit share PCI address space.
- Secondary ATU (SATU) — provides access between the secondary PCI bus and the 80303 I/O Processor Internal Bus.

Transactions initiated on a PCI bus and targeted at the 80303 I/O Processor Internal Bus are referred to as *inbound transactions* (PCI to internal bus); transactions initiated on the 80303 I/O Processor Internal Bus and targeted at a PCI bus are referred to as *outbound transactions* (internal bus to PCI). The ATU handles multiple inbound PCI transactions; it can simultaneously process PCI read and write transactions.

During inbound transactions, the ATU converts PCI addresses (initiated by a PCI bus master) to internal bus addresses and initiates the data transfer on the 80303 I/O Processor Internal Bus. During outbound transactions, the ATU converts internal bus addresses to PCI addresses and initiates the data transfer on the respective PCI bus.

The Messaging Unit provides a mechanism for the system processor and the 80303 I/O processor to transfer control information. The Messaging Unit occupies the first 4 Kbytes of the Primary ATU address space. PCI masters on the primary interface of the 80303 I/O processor access the MU by addressing the PATU anywhere in the first 4 KB offset from the PATU Base Address Register. When the Secondary Bus Messaging Unit Access mode is enabled, secondary PCI masters can access the MU by forwarding transactions through the PCI-to-PCI Bridge Unit.

The Expansion ROM provides the PCI mechanism for downloading device/board driver code during system boot sequence. It consists of a separate inbound address range which accesses a Flash EPROM device connected through the 80303 I/O processor memory controller. Refer to the *PCI Local Bus Specification,* Revision 2.2 for details of Expansion ROM usage.

The Primary and Secondary Address Translation Units, the Expansion ROM Translation Unit, and the Messaging Unit appear as a single PCI device on the primary PCI bus. These units collectively are the second PCI function in the multi-function 80303 I/O processor. (Refer to Section 15.2.4 for exceptions to this statement.) The block diagram for the ATUs and the Messaging Unit is shown in Figure 15-1.

Both the Primary ATU and the Secondary ATU support the PCI 64-bit and 66 MHz extensions providing up to 528 Mbytes/sec of PCI bandwidth. On the internal interface, the Primary and Secondary ATU implement the 80303 I/O processor internal bus protocol which provides for a maximum of 528 Mbytes/sec using 64-bit/66 MHz signaling.

The Primary ATU includes an 8 byte extended capability header that implements Power Management registers as defined by the *PCI Bus Power Management Interface Specification,* Revision 1.1.

The functionality of the ATUs is described in the following sections. The Primary and Secondary ATUs (and the Messaging Unit) have a memory-mapped register interface that is visible from either the PCI interface, the internal bus interface, or both.

Figure 15-1. ATU Block Diagram

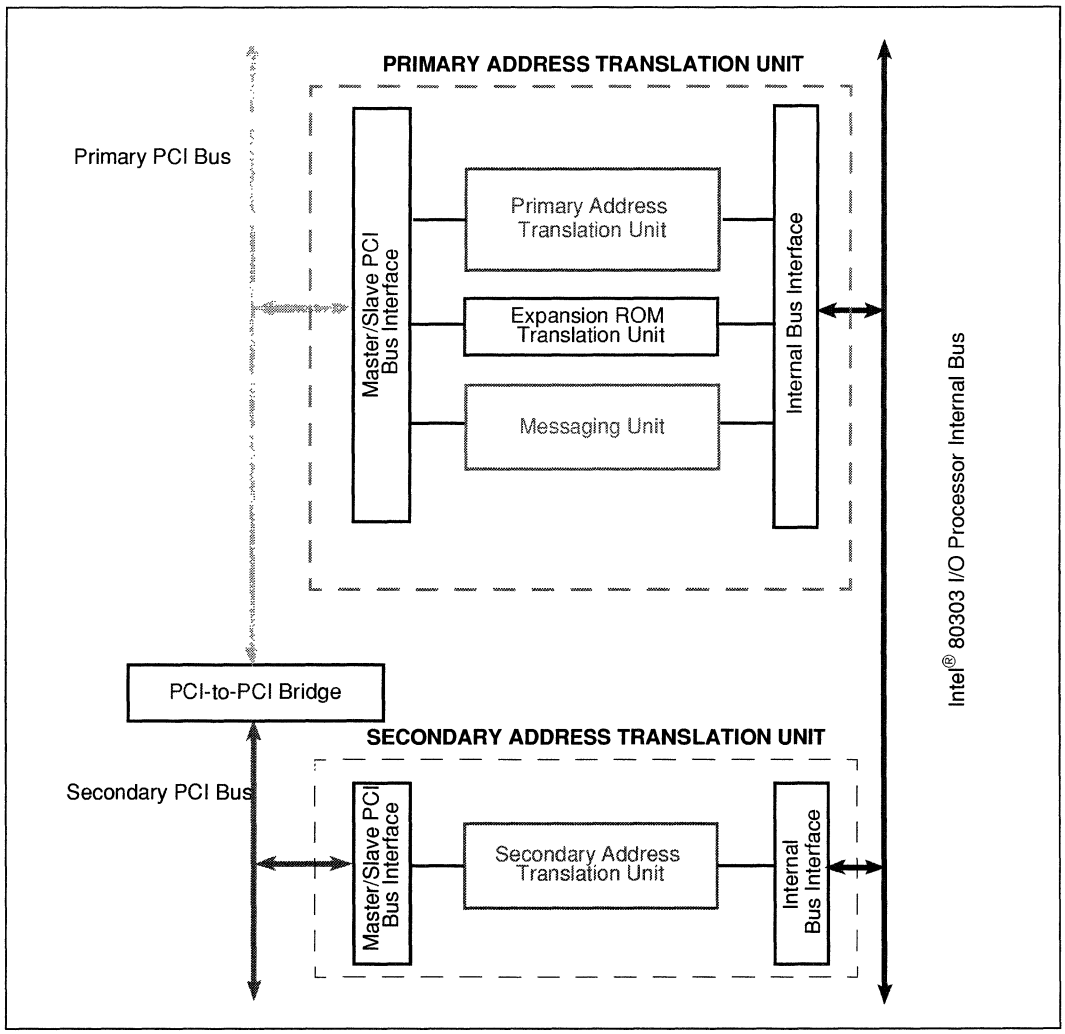
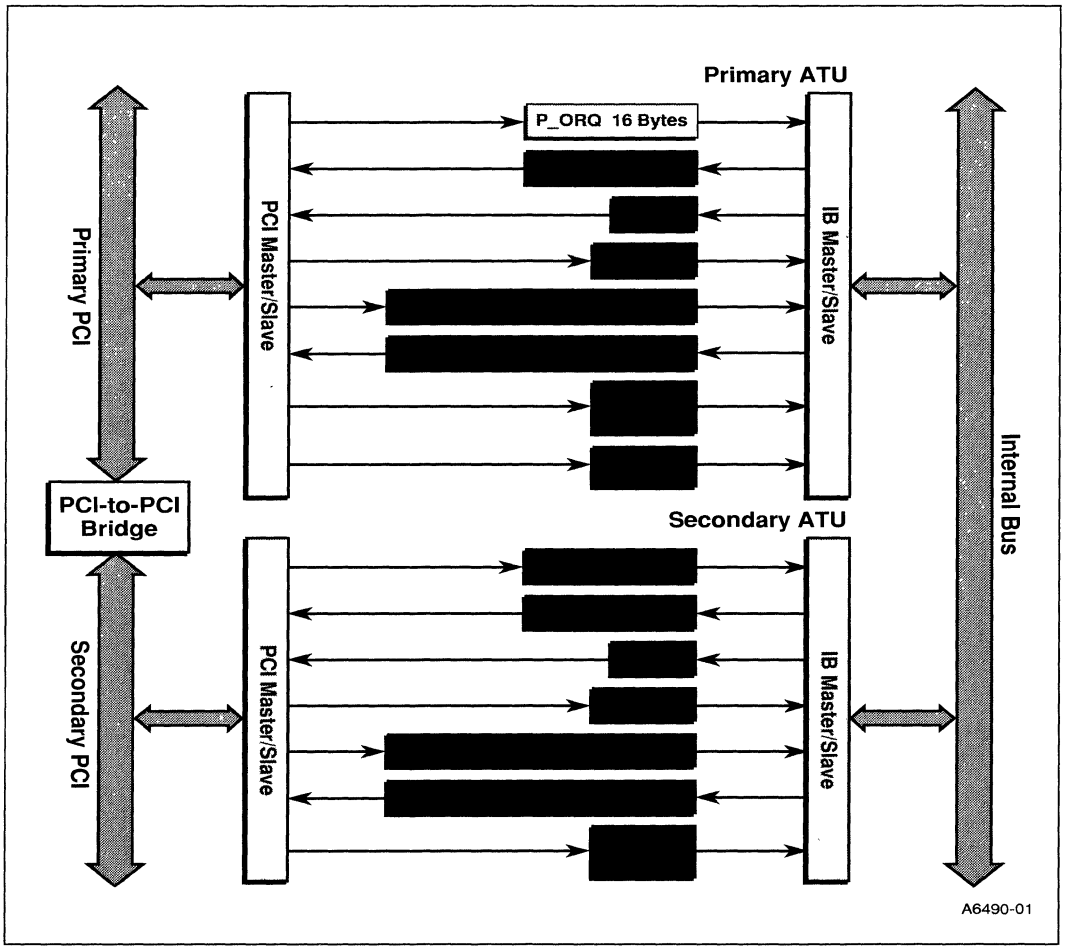


Figure 15-2. ATU Queue Architecture Block Diagram



15.2 ATU Address Translation

The primary ATU and the secondary ATU support transactions from both directions through the 80303 I/O processor. The primary ATU allows PCI masters on the primary PCI bus to initiate transactions to the 80303 I/O Processor Internal Bus and allows the i960® core processor to initiate transactions to the primary PCI bus. The secondary ATU performs the same function, but on the secondary PCI bus and for secondary PCI bus masters.

The ATUs implement an address windowing scheme to determine which addresses to claim and translate to the appropriate bus.

- The address windowing mechanism for inbound translation is described in Section 15.2.1.1, "Inbound Address Translation" on page 15-6
- The address windowing mechanism for outbound translation is described in Section 15.2.2.1, "Outbound Address Translation" on page 15-15

The ATU has the ability to handle multiple inbound PCI transactions simultaneously. The ATU may contain up to four PCI memory writes up to the data queue size of the ATU (PATU or SATU). Each ATU is also capable of handling two outstanding delayed read transactions. Refer to Figure 15-2 and Section 15.5 for details of the ATU queue architecture.

The primary ATU contains a data path between the primary PCI bus and the internal bus. Connecting the primary ATU in this manner enables data transfers to occur without requiring any resources on the secondary PCI bus. The secondary ATU contains a data path between the secondary PCI bus and the internal bus. The secondary ATU allows secondary PCI bus masters to access the internal bus and 80303 I/O processor local memory. These transactions are initiated by a secondary bus master and do not require any bandwidth on the primary PCI bus.

The ATU units allow for recognition and generation of multiple PCI cycle types. Table 15-1 shows the PCI commands supported for both inbound and outbound ATU transactions. The type of operation seen by the ATU on inbound transactions is determined by the PCI master (on either primary or secondary bus) who initiates the transaction. Claiming an inbound transaction depends on the address range programmed within the inbound translation window. The type of transaction used by the ATU on outbound transactions is determined by the 80303 local address and the fixed outbound windowing scheme. See Section 15.2.2.1, "Outbound Address Translation" on page 15-15 for the full details on outbound PCI cycle selection.

Both ATUs support the 64-bit addressing specified by the *PCI Local Bus Specification,* Revision 2.2. This 64-bit addressing extension is for outbound data transactions only (i.e., data transfers initiated by the i960 core processor). This is in addition to the 64-bit data extensions supported by the 80303 I/O processor. Refer to Section 15.2.5 for details of 64-bit PCI operation.

Neither ATU supports exclusive access using the PCI LOCK# signal. Also, the ATUs do not guarantee atomicity for outbound transactions when performing atomic accesses using i960 core processor atomic instructions (**atmod**, **atadd**).

Table 15-1. ATU Command Support

PCI Command Type	Claimed on Inbound Transactions on PCI Bus	Generated by Outbound Transactions on PCI Bus	Valid Internal Bus Command
Interrupt Acknowledge	No	No	No
Special Cycle	No	No	No
I/O Read	No	**Yes**	No
I/O Write	No	**Yes**	No
Memory Read	**Yes**	**Yes**	Yes
Memory Write	**Yes**	**Yes**	Yes
Memory Write and Invalidate	**Yes**	No	No
Memory Read Line	**Yes**	No	Yes
Memory Read Multiple	**Yes**	No	Yes
Configuration Read	**Yes**	**Yes**	Yes
Configuration Write	**Yes**	**Yes**	Yes
Dual Address Cycle	No	**Yes**	No

Inbound and outbound ATU transactions are best described by the data flows used on the PCI bus and the 80303 I/O processor internal bus during read and write operations. The following sections describe read and write operations for inbound ATU transactions (PCI to internal bus) and outbound transactions (internal bus to PCI). For the purposes of data flows, there are no distinctions between primary ATU transactions and secondary ATU transactions.

15.2.1 Inbound Transactions

Inbound transactions which target the PATU or the SATU are translated and performed on the 80303 I/O processor internal bus. As a PCI target, the ATUs are capable of accepting all PCI memory read and write operations as either a 32-bit or a 64-bit target PCI target. Refer to Section 15.2.5 for details on 64-bit PCI operation. *Memory Writes* and *Memory Write and Invalidate* operations are performed as posted operations and all memory read operations are performed as delayed reads. The PATU is capable of accepting configuration read and write cycles. For *Configuration Writes*, the cycles are performed as delayed memory write operations. *Configuration Reads* are performed as delayed read operations.

Inbound write transactions have their address entered into the inbound write address queue (IWQAD) and data entered into the inbound write data queue (IWQ). The IWQ/IWQAD pair are capable of holding up to 4 write operations up to the size of the data queue. Inbound read operations (memory and configuration) have their address entered into the inbound transaction queue (ITQ) and the data is returned to the PCI master in the inbound read queue (IRQ). Inbound configuration writes use the inbound delayed write queue (IDWQ) for address and data. Refer to Section 15.5 for details of queue operation.

For inbound transactions, the ATUs are slaves on the PCI bus and are masters on the internal bus. PCI slave operation is defined in the *PCI Local Bus Specification,* Revision 2.2..

15.2.1.1 Inbound Address Translation

The ATUs allow PCI bus masters to directly access the internal bus. These PCI bus masters can read or write 80303 I/O processor memory-mapped registers or 80303 I/O processor local memory space. The process of inbound address translation involves two steps:

1. Address Detection.
   — Determine when the 32-bit PCI address is within the address window defined for the inbound ATU (primary or secondary).
   — Claim the PCI transaction with medium DEVSEL# timing.
2. Address Translation.
   — Translate the 32-bit PCI address to a 32-bit 80303 I/O Processor Internal Bus address.

The ATUs use the following registers in inbound address translation:

- Inbound ATU Base Address Register
- Inbound ATU Limit Register
- Inbound ATU Translate Value Register

See Section 15.7, "Register Definitions" on page 15-54 for details on inbound translation register definition and programming constraints.

By convention, primary inbound ATU addresses are primary PCI addresses; secondary inbound ATU addresses are secondary PCI addresses. For the SATU, inbound addresses beyond the first 4 KB of the SATU inbound address space which are capable of being claimed by the secondary interface of bridge unit and the SATU slave interface, will be claimed by the SATU.

The first 4 KB of the SATU inbound address space is dependent on the value of bit 12 (Secondary Bus-Messaging Unit Access Enable bit) of the ATUCR. If set, these addresses are claimed by the secondary interface of the Bridge Unit (if a valid bridge address). If clear, these addresses are claimed by the SATU for forwarding to the internal bus. See Section 15.3 for details.

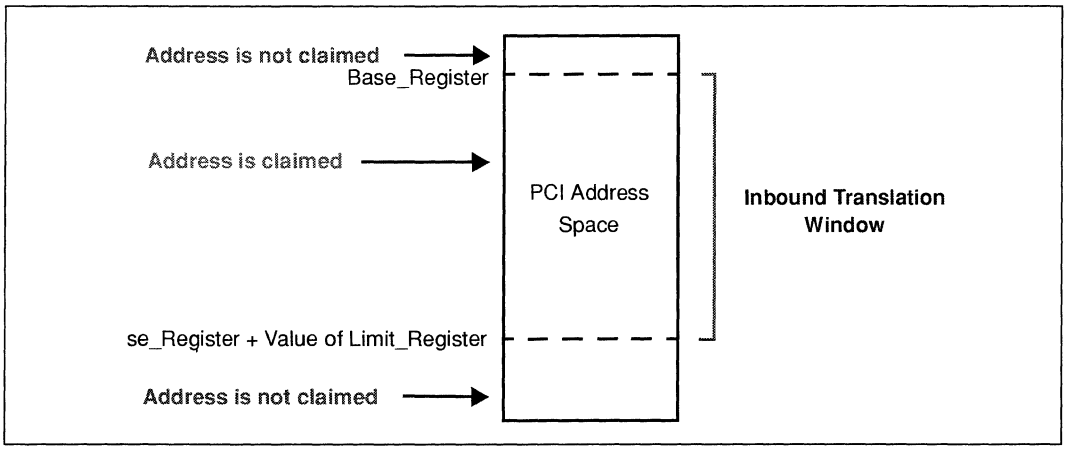
Inbound address detection is determined from the 32-bit PCI address, the base address register and the limit register. The algorithm for detection is:

Equation 15-1. Inbound Address Detection

```
When (PCI_Address & Limit_Register == Base_Register) the PCI Address is claimed by the Inbound ATU.
```

Figure 15-3 shows an example of inbound address detection.

Figure 15-3. Inbound Address Detection



The incoming 32-bit PCI address is bitwise ANDed with the associated inbound limit register. When the result matches the base register, the inbound PCI address is detected as being within the inbound translation window and is claimed by the ATU.

Note: The first 4 Kbytes of the primary ATUs inbound address translation window are reserved for the Messaging Unit. PCI addresses in this 4 Kbyte area are not translated and forwarded to the local bus as inbound transactions. See Section 15.3, "Messaging Unit" on page 15-29.

Once the transaction is claimed, the address must be translated from a 32-bit PCI address to a 32-bit internal bus address. The algorithm is:

Equation 15-2. Inbound Translation

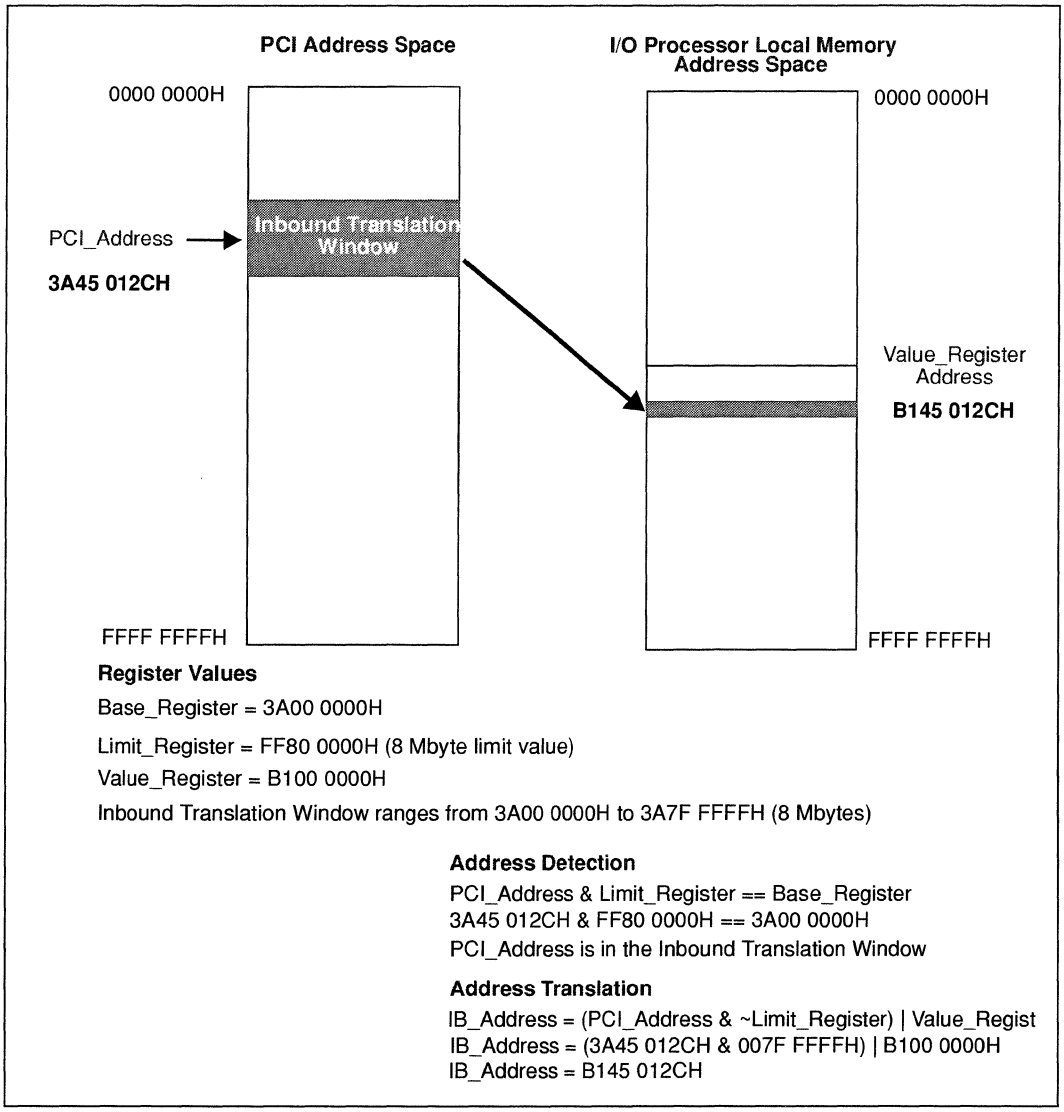
```
80303 I/O Processor Internal Bus Address = (PCI_Address & ~Limit_Register) |
ATU_Translate_Value_Register.
```

The incoming 32-bit PCI address is first bitwise ANDed with the bitwise inverse of the limit register. This result is bitwise ORed with the ATU Translate Value and the result is the internal bus address. This translation mechanism is used for all inbound memory read and write commands excluding inbound configuration read and writes. Inbound configuration cycle translation is described in Section 15.2.1.4, "Inbound Configuration Cycle Translation" on page 15-13. Address aliasing of multiple PCI addresses to the same physical 80303 I/O Processor Internal Bus address can be prevented by programming the inbound translate value register on boundaries matching the associated limit register, but this is only enforced through application programming.

For inbound memory transactions, the only burst order supported is Linear Incrementing. For any other burst order, the ATU signals a Disconnect after the first data phase.

Figure 15-4 shows an inbound translation example. This example would hold true for an inbound transaction from either the primary or secondary PCI bus.

Figure 15-4. Inbound Translation Example



15.2.1.2 Inbound Write Transaction

An inbound write transaction is initiated by a PCI master (on either the primary or secondary PCI bus) and is targeted at either 80303 I/O processor local memory or a 80303 I/O processor memory-mapped register.

Data flow for an inbound write transaction on the PCI bus is summarized as:

- The ATU claims the PCI write transaction when the PCI address is within the inbound translation window defined by the ATU Inbound Base Register and Inbound Limit Register.
- If the IWQAD has at least one address entry available and the IWQ is not full and is capable of accepting data (dependent upon the Memory Write Non-Full State Bits in Section 15.7.49, "Primary ATU Queue Control Register - PAQCR" on page 15-112 and Section 15.7.50, "Secondary ATU Queue Control Register - SAQCR" on page 15-113, the address is latched and the first data phase is accepted. If additional queue space is available, the slave interface continues accepting data until the IWQ reaches a full state. If **REQ64#** was driven by the initiator, data is accepted as 64-bit, otherwise a 32-bit transactions is used.
- If an address parity error is detected during the address phase of the transaction, the address parity error mechanisms are used. Refer to Section 15.6.1 for details of the address parity error response. If a data parity error is detected while accepting data, the slave interface sets the appropriate bits based on PCI specification. No other action is taken. Refer to Section 15.6.2.4 for details of the inbound write data parity error response.
- The PCI interface continues to accept write data until one of the following is true:
  - — The initiator performs a master completion.
  - — The IWQ becomes full. In this case, the PCI interface signals a Disconnect to the initiator and returns to idle.
- If a master abort or a memory controller multi-bit ECC error (target abort), occurs during the inbound transaction on the internal bus and the transaction is still active on the PCI interface, the slave interface will perform a disconnect, and **SERR#** is asserted based upon the setting of the PATUIMR or SATUIMR, see Section 15.7.51, "Primary ATU Interrupt Mask Register - PATUIMR" on page 15-114 and Section 15.7.52, "Secondary ATU Interrupt Mask Register - SATUIMR" on page 15-115.

Once the PCI interface places a PCI address in the IWQAD and a 2 QWORD boundary is crossed or if the master disconnects on the PCI bus, the ATUs internal bus interface becomes aware of the inbound write. If there are additional write transactions ahead in the IWQ/IWQAD, the current transaction remains posted until ordering and priority have been satisfied (Refer to Section 15.5.3) and the transaction is attempted on the internal bus by the ATU internal master interface. If there are no other write operations in the queue and ordering and the priority mechanism supports it, the ATU will attempt to immediately acquire the internal bus and allow write streaming to occur. If the queue fills or the master completes before the first data phase is accepted (by the assertion of **I_TRDY#**) on the internal bus, streaming can not occur. The ATU will not insert target wait states nor do data merging on the PCI interface to allow for streaming.

Data flow for the inbound write transaction on the internal bus is summarized as:

- The ATU internal bus master requests the internal bus when the IWQAD/IWQ contains the PCI address and data for the current transaction which has crossed at least a 2 QWORD boundary or a PCI address from an earlier posted PCI transaction has moved to the head of the IWQAD.
- When the internal bus is granted, the internal bus master interface initiates the write transaction by driving the translated address onto the internal bus. For details on inbound address translation, see Section 15.2, "ATU Address Translation" on page 15-4. If **I_DEVSEL#** is not returned, a master abort condition is signaled on the internal bus. The current transaction is flushed from the queue and **SERR#** on the PCI interface is asserted based upon the setting of the ATUCR, see Section 15.7.38, "ATU Configuration Register - ATUCR" on page 15-98.
- Write data is transferred from the IWQ to the internal bus when data is available and the internal bus interface retains internal bus ownership. The ATU master interface will assert **I_REQ64#** to attempt a 64-bit transfer. If **I_ACK64#** is not returned, a 32-bit transfer is used. Transfers of less than 64-bits use the **I_C/BE[7:0]#** to mask the bytes not written in the 64-bit data phase.
- The internal bus interface stops transferring data from the current transaction to the internal bus when one of the following conditions becomes true:
  - — The internal bus master interface loses bus ownership and the master latency timer has expired. The ATU internal master will perform a master completion and attempt to reacquire the bus to complete delivery of the data.
  - — A Disconnect with Data is signaled on the internal bus from the internal slave. If the transaction in the IWQ is complete, the master returns to idle. If the transaction in the IWQ is not complete, the master attempts to reacquire the internal bus.
  - — The data from the current transaction has completed. A master completion is performed and the bus returns to idle.
  - — A Target Abort is signaled from the internal bus slave. This is in response to an ECC error from the memory controller. **SERR#** is asserted based upon the setting of the PATUIMR or the SATUIMR, see Section 15.7.51, "Primary ATU Interrupt Mask Register - PATUIMR" on page 15-114 and Section 15.7.52, "Secondary ATU Interrupt Mask Register - SATUIMR" on page 15-115. A disconnect is signaled on PCI if the transaction is active. If the transaction in the IWQ is complete, the master returns to idle. If the transaction in the IWQ is not complete, the master attempts to reacquire the internal bus. Refer to Section 15.6.6.2, for full details.
  - — A Master Abort is signaled on the internal bus. **SERR#** is asserted based upon the setting of the PATUIMR or the SATUIMR, see Section 15.7.51, "Primary ATU Interrupt Mask Register - PATUIMR" on page 15-114 and Section 15.7.52, "Secondary ATU Interrupt Mask Register - SATUIMR" on page 15-115. Data is flushed from the IWQ.
- When the ATU attempts to transfer data in the IWQ to the IB and is stopped during a burst for any reason other than a Master Abort, the ATU will attempt to reacquire the IB only after one of the following conditions is met:
  - — The transactions has disconnected on the PCI bus.
  - — At least 4 Dwords are in the IWQ.
  - — The next IB address to attempt is not Qword aligned.

15.2.1.3 Inbound Read Transaction

An inbound read transaction is initiated by a PCI master (on either the primary or secondary PCI bus) and is targeted at either 80303 I/O processor local memory or a 80303 I/O processor memory-mapped register. The read transaction is propagated through the inbound transaction queues (ITQ1 and ITQ2) and read data is returned through the inbound read queue (IRQ).

All inbound read transactions are processed as delayed read transactions. The ATUs PCI interface claims the read transaction and forwards the read request through to the internal bus and returns the read data to the PCI bus. Data flow for an inbound read transaction on the PCI bus is summarized in the following statements:

- The ATU claims the PCI read transaction when the PCI address is within the inbound translation window defined by ATU Inbound Base Register and Inbound Limit Register.
- When one of the ITQs is empty, the PCI address and command are latched into the available ITQ and a Retry is signalled to the initiator.
- If an ITQ is currently holding transaction information from a previous delayed read, the current transaction information is compared to the previous transaction information (based on the setting of the DRC Alias bit in Section 15.7.38, "ATU Configuration Register - ATUCR" on page 15-98). If there is a match and the data is in the IRQ, return the data to the master on the PCI bus. If there is a match or the data is not available, a Retry is signaled with no other action taken. If there is not a match and there is an ITQ available, latch the transaction information, signal a Retry and initiate a delayed transaction. If there is not a match and there is not an ITQ available, signal a Retry with no other action taken.
  - — For the case where there is a match on the transaction information and the IRQ is currently being filled, memory read streaming is possible.
  - — If an address parity error is detected, address parity response defined in Section 15.6 is used.
- Once read data is driven onto the PCI bus from the IRQ, it continues until one of the following is true:
  - — The initiator completes PCI transaction. Unread data left in the IRQ is flushed.
  - — An internal bus Target Abort was detected. In this case, the Q-word associated with the Target Abort is never entered into the IRQ, and therefore is never returned.
  - — The IRQ becomes empty. In this case, the PCI interface signals a Disconnect with data to the initiator on the last data word available.

  The slave ATU interface delivers 64-bit read data if **REQ64#** was asserted and 32-bit read data if **REQ64#** was deasserted.
- If the master inserts waitstates on the PCI bus, the ATU PCI slave interface will wait with no premature disconnects.
- If a data parity error occurs signified by **PERR#** asserted from the master, no action is taken by the slave interface. Refer to Section 15.6.2.3.
- If the transaction on the internal bus resulted in a master abort, the completion cycle is allowed to master abort on the PCI interface. The ITQ for this transaction is flushed (Section 15.6.1.).
- When the first Q-word read on the internal bus is target-aborted, either a target-abort or a disconnect with data is signaled to the initiator. This is based on the ATU ECC Target Abort Enable bit (bit 0 of the PATUIMR for PATU and bit 0 of the SATUIMR for the SATU). If set, a target abort is used, if clear, a disconnect is used.

The data flow for an inbound read transaction on the internal bus is summarized in the following statements:

- The ATU internal bus master interface will request the internal bus when a PCI address appears in an ITQ and transaction ordering has been satisfied.
- Once the internal bus is granted, the internal bus master interface will drive the translated address onto the bus and wait for **I_DEVSEL#**. If a Retry is signaled, the request will be repeated. If a master abort occurs, the transaction is considered complete and a master abort is loaded into the associated IRQ for return to the PCI initiator (transaction is flushed once the PCI master has been delivered the master abort).
- Once the translated address is on the bus and the transaction has been accepted, the internal bus slave will start returning data with the assertion of **I_TRDY#**. Read data is continuously received by the IRQ until one of the following is true:
  - — The predetermined prefetch data amount is received. This is detailed in Section 15.5.1.2. The ATU internal bus master interface performs a master completion in this case.
  - — A Target Abort is received on the internal bus from the internal bus slave. In this case, the transaction is aborted. If a Target Abort occurs before 64 bits are ready, notify the PCI side; otherwise, discard the Target Aborted Q-word and take no further action.
  - — The IRQ becomes full. In this case, the ATU master performs a master completion.
  - — The ATU loses ownership of the internal bus and the master latency timer has expired. A master completion is performed on the internal bus. If less than 64-bits of data has been fetched, the ATU IB master interface attempts to reacquire the bus. If not, the bus returns to idle.
  - — A disconnect with data is received from the internal bus slave. If less than 64-bits of data has been fetched, the ATU internal bus master interface attempts to reacquire the bus. If not, the bus returns to idle.

  If the prefetch amount of data has been read and the PCI bus is actively draining the data on the PCI interface, the ATU will continue to read data and latch it into the IRQ to support inbound read streaming. If the IRQ fills and the PCI interface is active, IB master wait states are not inserted to support streaming.
- Since all inbound reads are promoted to 64-bit internal bus transactions, a disconnect from the internal bus target with less than 8 bytes returned to the IRQ creates a problem for 64-bit PCI requestors. To guarantee a minimum of 64-bits of data prefetched for the PCI initiator, the ATU will reacquire the internal bus.

To support *PCI Local Bus Specification*, Revision 2.0 devices, the ATUs can be programmed to ignore the memory read command (Memory Read, Memory Read Line, and Memory Read Multiple) when trying to match the current inbound read transaction with data in a DRC queue which was read previously (DRC on target bus). If the Read Command Alias Bit in the ATUCR register is set, the ATUs will not distinguish the read commands on transactions. For example, the ATU enqueues a DRR with a Memory Read Multiple command and performs the read on the internal bus. Some time later, a PCI master attempts a Memory Read with the same address as the previous Memory Read Multiple. If the Read Command Bit is set, the ATU would return the read data from the DRC queue and consider the Delayed Read transaction complete. If the Read Command bit in the ATUCR was clear, the ATU would not return data since the PCI read commands did not match, only the address.

15.2.1.4 Inbound Configuration Cycle Translation

The ATU only accepts Type 0 configuration cycles with a function number of one (the bridge is function 0 in the 80303 I/O processor). (Refer to Section 15.2.4, "PCI Multi-Function Device Swapping/Disabling" on page 15-23 for exceptions to this statement.)

Both primary and secondary ATUs are configured through the primary ATU. This means that only one configuration space exists for both PCI buses. All inbound configuration cycles are processed as delayed transactions. The translation mechanism for inbound configuration cycles is defined by the *PCI Local Bus Specification,* Revision 2.1.

The ATU configuration space is selected by a PCI configuration command and will claim the access (by asserting **P_DEVSEL#**) if the **P_IDSEL** pin is asserted, the PCI command indicates a configuration read or write, and address bits **P_AD[1:0]** are 00_2 all during the address phase. The ATU primary interface will ignore any configuration command (**P_IDSEL** active) where **P_AD[1:0]** are not 00_2 (e.g., Type 1 commands). During the configuration access address phase, the PCI address is divided into a number of fields to determine the actual configuration register access. These fields, in combination with the byte enables during the data phase create the unique encoding necessary to access the individual registers of the configuration address space:

- **P_AD[7:2]** - Register Number. Selects one of 64 DWORD registers in the ATU PCI configuration address space.
- **P_C/BE[3:0]#** - Used during the data phase. Selects which actual configuration register is used within the DWORD address. Creates byte addressability of the register space.
- **P_AD[10:8]** - Function Number. Used to select which function of a multi-function device is being accessed. The ATUs are function 1 and therefore it will only respond to 001_2 in this bit field and ignore all other bit combinations. (Refer to Section 15.2.4, "PCI Multi-Function Device Swapping/Disabling" on page 15-23 for exceptions to this statement.)

The ATU configuration address space starts at internal address 0000.1200H. Therefore **P_AD[7:2]** equal to 000000_2 equates to address 0000.1200H and **P_AD[7:2]** equal to 000001_2 results in address 0000.1204H and so on.

For inbound configuration reads, the IRQ and ITQ are used in the same manner as inbound memory read operations. The internal bus cycle that results will be a 32-bit transaction where **I_REQ64#** is not asserted. For inbound configuration writes, the PATU adds a delayed write data queue, IDWQ, which holds data in the same manner as the IWQ. The transaction information from the configuration write operation on the primary PCI interface is latched into the IDWQ (if full, a Retry is signaled). The data from the delayed write request cycle is latched into the IDWQ and forwarded to the internal bus interface. Once transaction ordering and priority have been satisfied, the internal bus master interface will request the internal bus and deliver the write data to the target as defined in Section 15.2.1.2.

The status of the transaction on the internal bus is returned to the PCI initiator on the primary PCI bus. The retry cycle from the initiator is accepted once the write has been completed on the internal bus and the status has been latched for return to the PCI master. Since Master Aborts and Target Aborts cannot occur during configuration cycles on the internal bus, normal completion status is returned. The data from PCI completion transaction is discarded.

15.2.1.5 Discard Timers

The ATUs implement discard timers for inbound delayed transactions. These timers prevent deadlocks when the initiator of a retried delayed transaction fails to complete the transaction within 2^{10} or 2^{15} PCI clock cycles on the initiating bus. The timer starts counting when the delayed request becomes a delayed completion by completing on the internal bus. When the originating master on the PCI bus has not retried the transaction before the timer expires, the completion transaction is discarded.

Discard timer values are controlled by the Bridge Control Register's Primary Discard Timer Value bit (for the primary ATU) and the Secondary Discard Timer Value bit (for the secondary ATU). The PATU queues covered by discard timers are the P_IRQ and the P_IDWQ. The SATU queue covered by discard timer is the S_IRQ. After discarding a transaction, the ATUs must set the Discard Timer Status bit in the ATU Control Register. However, unlike the PCI to PCI Bridge Unit, the ATUs do *not* assert the P_**SERR#** signal after discarding a transaction.

15.2.2 Outbound Transactions

Outbound transactions initiated by the 80303 I/O processor core processor are either to the primary PCI interface through the PATU or the secondary PCI interface through the SATU. As a PCI master, the ATUs are capable of PCI I/O transactions, PCI memory reads (excluding the read hint commands MRL and MRM), PCI memory writes (excluding MWI), configuration reads and writes, and DAC cycles. Outbound transactions are performed as either 32-bit or a 64-bit PCI transactions. Refer to Section 15.2.5 for details on 64-bit operation. Outbound memory write operations are performed as posted operations and outbound memory read operations are all performed as delayed read operations.

Outbound transactions use a separate set of queues from inbound transactions. Outbound write operations have their address entered into the outbound transaction queue (OTQ) and their data into the outbound write queue (OWQ). Outbound read transactions, performed as delayed operations, use the same address queue, the OTQ, and get data returned into the outbound read queue (ORQ). Refer to Section 15.5.2 for details of outbound queue architecture. Outbound configuration transactions use a special outbound port structure. Refer to Section 15.2.3 for details.

For outbound transactions, the ATUs are slaves on the internal bus and masters on the PCI bus. PCI master operation is defined in the *PCI Local Bus Specification,* Revision 2.2.

15.2.2.1 Outbound Address Translation

In addition to providing the mechanism for inbound translation, the ATUs translate i960 core processor-initiated cycles to the PCI bus. This is known as *outbound address translation*. Outbound transactions are processor reads or writes targeted at the PCI primary or secondary bus. The ATU internal bus slave interface claims internal bus cycles and completes the cycle on the PCI bus on behalf of the i960 core processor. The primary and secondary ATUs support two different outbound windowing modes:

- Address Translation Windowing
- Direct Address Windowing (No translation)

Figure 15-5 shows a 80303 I/O processor memory map with all reserved address locations highlighted. The outbound translation windows exist from 8000.0000H to 9001.FFFFH. This is a 256 Mbyte window and a 128 Kbyte window which are equally divided between the primary and secondary ATUs. The direct addressing window is from 0000.2000H to 7FFF.FFFFH. Both outbound schemes are described in the following subsections.

Outbound address translation is disabled for the Primary ATU when the Bus Master Enable bit in the Primary ATU Command Register is clear and is disabled for the Secondary ATU when the Bus Master Enable bit in the Secondary ATU Command Register is clear. When the Bus Master Enable bit is clear, the ATU does not claim any i960 core processor accesses. These unclaimed accesses may result in an internal bus Master Abort. For outbound Memory transactions, the only burst order supported is Linear Incrementing.

15.2.2.2 Outbound Address Translation Windows

Inbound translation involves a programmable inbound translation window consisting of a base and limit register and a value register for PCI to internal bus translation. The outbound address translation windows use a similar methodology except that the outbound translation window base addresses and limit sizes are fixed in 80303 I/O Processor Internal Bus local address space; this removes the need for separate base and limit registers.

Figure 15-6 illustrates the outbound address translation windows. Each ATU has three windows: two are 64 Mbyte and one is 64 Kbyte. The primary outbound memory and DAC translation windows range from 8000.0000H to 87FF.FFFFH and the secondary outbound memory and DAC translation windows range from 8800.0000H to 8FFF.FFFFH. After these four windows, the primary and secondary outbound I/O windows range from 9000.0000H to 9001.FFFFH.

Each memory and DAC window is 64 Mbytes and each I/O window is 64 Kbytes. An internal bus cycle with an address within one outbound window initiates a read or write cycle on the targeted PCI bus. The PCI cycle type depends on which translation window the local bus cycle "hits". The read or write decision is based on the internal bus cycle type.

Each ATU has a window dedicated to the following outbound PCI transaction types:

- Memory reads and writes - Memory Window
- I/O reads and writes - I/O Window
- Dual Address Cycle reads and writes - DAC Window

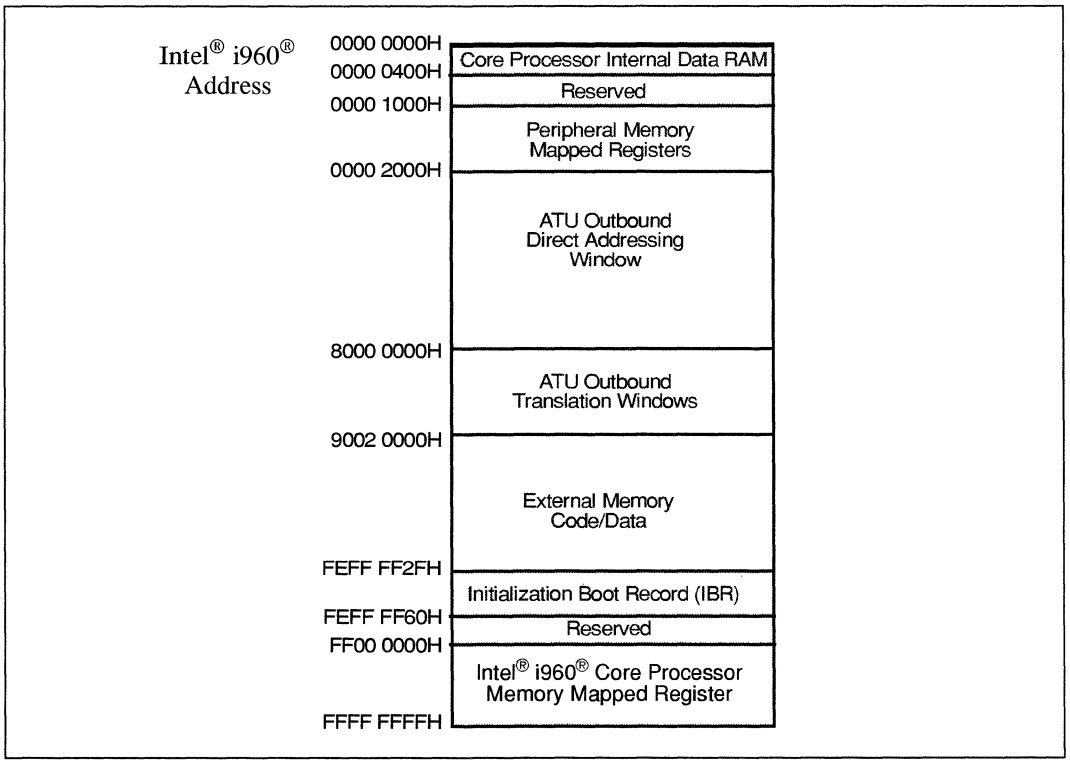
Refer to Figure 15-6 for the sub-window addresses involved in primary and secondary outbound translation.

The windowing scheme refers to:

- a core processor read cycle that addresses a Memory Window is translated to a Memory Read on the PCI bus
- a core processor write cycle that addresses a Memory Window is translated to a Memory Write on the PCI bus
- a core processor read cycle that addresses the I/O Window is an I/O Read on the PCI bus
- a core processor write cycle that addresses the I/O Window is an I/O Write on the PCI bus
- a core processor read cycle that addresses a DAC Window is translated to a DAC Memory Read on the PCI bus
- a core processor write cycle that addresses a DAC Window is translated to a DAC Memory Write on the PCI bus

Memory Write and Invalidate (MWI), Memory Read Line, and Memory Read Multiple commands are not supported in outbound ATU transactions on the PCI interface.

Figure 15-5. Intel® i960® Memory Map - Outbound Translation Window



The translation portion of outbound ATU transactions is accomplished with a value register in the same manner as inbound translations. The POUDR and SOUDR contain the high order 32-bits of a dual-cycle 64-bit address. Each ATU uses the registers shown below during outbound address translation:

- Outbound Memory Window Value Register
- Outbound I/O Window Value Register
- Outbound DAC Window Value Register
- Outbound Upper 64-Bit DAC Register
- Outbound Configuration Cycle Address Register

See Section 15.7 for details on outbound translation register definition and programming constraints. The translation algorithm used, as stated, is very similar to inbound translation. For memory and DAC transactions, the algorithm is:

Equation 15-3. Outbound Address Translation

```
PCI Address = (Internal_Bus_Address & 03FF.FFFFH) | Window_Value_Register
```

For memory and DAC transactions, the internal bus address is bitwise ANDed with the inverse of 64 Mbytes which clears the upper 6 bits of address. The result is bitwise ORed with the outbound window value register to create the lower 32-bits of the primary or secondary PCI address. For I/O transactions, the algorithm is:

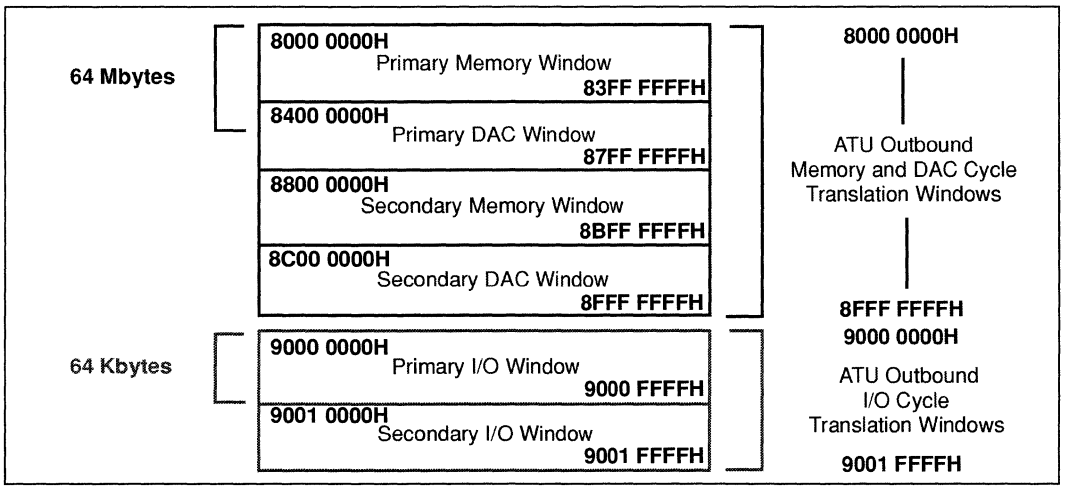
Equation 15-4. I/O Transactions

```
PCI Address = (Internal_Bus_Address & 0000.FFFFH) | Window_Value_Register
```

For I/O transactions, the internal bus address is bitwise ANDed with the inverse of 64 Kbytes which clears the upper 16 bits of address. Address aliasing can be prevented by programming the outbound window value registers on boundaries equivalent to the window's length, but this is only enforced through application programming. PCI I/O addresses are byte addresses and not word addresses. The PCI I/O address's two least significant bits are determined by byte enables that the processor issues. For example, when the i960 core processor performs a 2-byte write and generates byte enables of 0011_2, the ATU sets the two least significant bits of PCI I/O address to 10_2.

Note: When the i960 core processor data cache is enabled for accesses to the Outbound I/O Window, the byte enables generated by the i960 core processor are always 00_2 for Byte and Short accesses.

Figure 15-6. Outbound Address Translation Windows

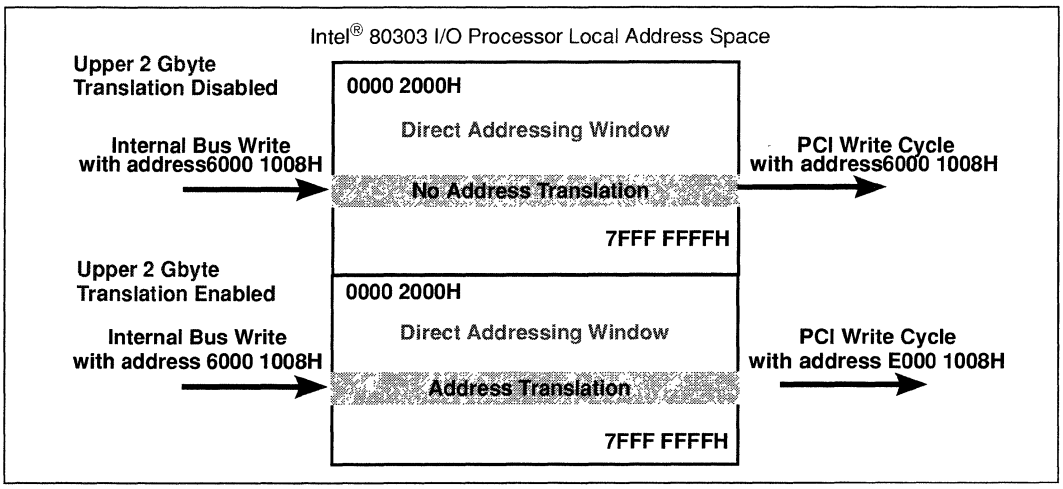


15.2.2.3 Direct Addressing Window

The second method used by outbound cycles from the 80303 I/O processor to the PCI bus is the direct addressing window. This is a window of addresses in 80303 I/O processor address space that act in the same manner as the outbound translation windows either without any translation or with the translation of address bit 31 only. This allows the Direct Addressing window to translate to different address ranges on the PCI bus (0000.2000H to 7FFF.FFFFH or 8000.2000H to FFFF.FFFFH). A 80303 I/O processor read or write to a local bus address within the direct addressing window initiates a read or write on the PCI bus with the same address (with the possible exception of address bit 31) as used on the internal bus. Figure 15-7 shows two examples of outbound writes that are through the direct addressing window.

Direct Addressing is limited to PCI memory read commands and writes only. I/O cycles, DAC cycles, and MWI commands are not supported with direct addressing.

Figure 15-7. Direct Addressing Window



The internal bus side of the direct addressing window address range is fixed in the lower 2 Gbytes of the 80303 I/O processor local address space (except for the first 8 Kbytes which is reserved for the i960 core processor internal data RAM and 80303 I/O processor memory-mapped registers). Internal bus cycles with an address from 0000.2000H to 7FFF.FFFCH are forwarded to a PCI bus, when enabled. The primary PCI bus is the default bus for direct addressing. The following bits within the ATUCR affect direct addressing operation:

- ATUCR Direct Addressing Enable bit - when set, enables the direct addressing window. When clear, addresses within the direct addressing window are not forwarded to the PCI bus.
- ATUCR Secondary Direct Addressing Select bit - when clear, all transactions through the direct addressing window are to the primary ATU and primary PCI bus. When set, all transactions through the direct addressing window are to the secondary ATU and secondary PCI bus.
- ATUCR Direct Addressing Upper 2G Translation Enable - when set, the ATU will forward internal bus cycles with an address between 0000.2000H and 7FFF.FFFFH to the PCI bus with bit 31 of the address set (8000.2000H - FFFF.FFFFH). When clear, no translation will occur.

15.2.2.4 Outbound Write Transaction

An outbound write transaction is initiated by the i960 core processor and is targeted at a PCI slave on either the primary or secondary PCI buses. The outbound write address and write data are propagated from the 80303 I/O Processor Internal Bus to a PCI bus through the OTQ and OWQ, respectively.

The ATUs internal bus slave interface claims the write transaction and forwards the write data through to the targeted PCI bus. The data flow for an outbound write transaction on the internal bus is summarized in the following statements:

- The ATU internal bus slave interface will latch the address from the internal bus into the OTQ when that address is inside one of the outbound translate windows (see Section 15.5) and the OTQ is empty.
- Once the outbound address is latched, the internal bus slave interface will store the write data into the OWQ until the internal bus transaction completes. The initiator of the transaction will perform a master completion when done writing data. The OWQ is capable of holding 16 bytes of data which is the maximum amount written by the core processor.
- When the OTQ is not available, the slave interface will signal a Retry on the internal bus to the outbound cycle initiator. The ATU will signal the BIU at the time of the Retry that it should not request the internal bus until the ATU has notified it that the OTQ is available.
- When the OTQ latches the address, the outbound cycle is enabled for transmission on the PCI Bus and the PCI master requests the PCI bus.

The PCI interface is responsible for completing the outbound write transaction to a PCI address translated from the OTQ and the data in the OWQ. The data flow for an outbound write transaction on the PCI bus is summarized in the following statements:

- The ATU PCI interface will request the PCI bus when the completed internal bus transaction is written to the OTQ (a write request). Once the bus is granted, the PCI master interface will write the PCI translated address from the OTQ to the PCI bus and wait for the transaction to be claimed.
- If a Master Abort is seen during the address phase, the transaction is flushed and the OTQ and OWQ are cleared. Refer to Section 15.6.3 for full details on PCI master abort conditions during outbound transactions.
- Once the PCI write transaction is claimed, the PCI interface will transfer data from the OWQ to the PCI bus until one of the following is true:
  - — The PCI target signals a Retry or Disconnect. The ATU PCI master will attempt to reacquire the PCI bus to complete the write transaction.
  - — The **GNT#** signal is deasserted and the master latency timer has expired. In this case, the master interface will attempt to reacquire the PCI bus and complete the write transaction.
  - — The PCI target signals a Target-Abort. In this case, the OWQ and OTQ are cleared and the transaction is aborted. The appropriate error bits are set defined in Section 15.6.4.
  - — The OWQ become empty signifying that the transaction is finished. The write address is removed from the OTQ and the interface returns to idle.

  If a data parity error is encountered (**PERR#** detected), the master interface will continue writing data to clear the queue.

  If the PCI target deasserts **TRDY#**, no action is taken by the ATU master other than inserting waitstates.

15.2.2.5 Outbound Read Transaction

An outbound read transaction is initiated by the i960 core processor and is targeted at a PCI slave on either the primary or secondary PCI buses. The read transaction is propagated through the outbound transaction queue (OTQ) and read data is returned through the outbound read queue (ORQ).

The ATUs internal bus slave interface claims the read transaction and forwards the read request through to the PCI bus and returns the read data to the internal bus. The byte enables for the first word only of the transaction are also passed by the ATU (to cover the case of less than 1 Dword being requested). The fetch data amount used by the PCI side is determined by the read command used on the internal bus by the IB master. Table 15-2 are the fetch data sizes used during outbound ATU read transactions:

Table 15-2. Outbound Read Fetch Sizes

Internal Bus Command	Outbound Fetch Size
Memory Read	4 Bytes (1 Dword)
Memory Read Line	8 Bytes (2 Dwords)
Memory Read Multiple	16 Bytes (4 Dwords)

The data flow for an outbound read transaction on the local bus is summarized in the following statements:

- The ATU internal bus interface latches the internal bus address when the address is inside an outbound address translation window (or the direct addressing window, if enabled) and the OTQ is empty. When the OTQ is not empty (previous outbound transaction in progress), the internal bus interface signals a Retry to the transaction initiator.
- Once the outbound internal address is latched into the OTQ, a Retry is signaled to the internal bus master and a delayed read transaction is initiated. The ATU will signal the BIU at the time of the Retry that a delayed cycle has started and that it should not request the internal bus until the ATU has notified it that the data to be read is now available.
- If during the completion cycle on the PCI interface, a master abort is encountered, a flag is set and the ATU notifies the BIU that it may now request the internal bus to complete the retried transaction. A master abort condition is returned once the IB master has acquired the bus and asserted the address of the delayed read completion cycle. The OTQ is cleared of the transaction.
- Once the transaction completes on the PCI bus, the ATU notifies the BIU that it may now request the internal bus to complete the retried transaction. The outbound read was deterministic with no prefetching and data read is the data that was required per the command used on the internal bus (see Table 15-2).
- A target abort encountered on the PCI bus is returned as a target abort to the IB master on the first data phase. If a data parity error is signaled on PCI, the bad data is still passed through to the IB master.

The data flow for an outbound read transaction on the PCI bus is summarized in the following statements:

- The ATU PCI interface will request the PCI bus when an address is written to the OTQ (a read request). Once the bus is granted, the PCI interface will transfer the PCI translated address from the OTQ to the PCI bus and wait for the transaction to be claimed.
- If no **DEVSEL#** is asserted, a master abort is signaled. This is passed through to the internal bus slave interface.
- Once the transaction is claimed and data is provided by the target, the PCI interface continue reading until the fetch data amount is satisfied. The master interface will stop reading under the following circumstances:
  - — A disconnect is signaled from the PCI target. The master interface will attempt to reacquire the bus and continue reading until the fetch data size is satisfied.
  - — The master interface loses **GNT#** and the interface MLT has expired. A master completion is performed and the interface attempts to reacquire the bus and continues reading until the fetch data size is satisfied.
  - — A target abort is signaled from the PCI target. The target abort is returned to the internal bus and the PCI interface returns to idle. The appropriate error bits are set as defined in Section 15.6.4.
  - — The fetch data size has been reached. The master interface performs a master completion and the interface returns to idle.

  If the PCI target inserts waitstates at any point, the PCI master interface halts until **TRDY#** is asserted. No other action is taken.

15.2.3 Private PCI Address Space / Outbound Configuration Cycle Translation

The secondary ATU contains special support for private address spaces on the secondary PCI bus. A private address space is defined as a range of secondary PCI bus addresses which are not part of the secondary PCI address space as defined by the bridge and are also not part of the primary PCI address space. Private address space can be considered a "hole" in the PCI address space that is only supported on the secondary PCI bus. Private address space generally falls within the primary PCI address space and requires special bridge support so that it does not forward these addresses. The 80303 I/O processor has several mechanisms to support private address space:

- Inbound transactions from private devices through the secondary ATU.
- Outbound transactions from the secondary ATU and DMA channel 2 to private devices.
- Outbound configuration cycles to private devices.
- Hiding private devices from PCI Type 0 configuration cycles. (See Chapter 14, "PCI-to-PCI Bridge Unit" for more details.)

For inbound transactions from private devices, the secondary ATU can be configured outside the valid secondary PCI address space; this creates private address space. The secondary ATU claims private addresses and prevents the bridge from forwarding them upstream to the primary PCI bus.

For outbound transactions from the secondary ATU and DMAs, the programmer needs to define a private memory address range (see Section 14.5.5, "Private Address Space" on page 14-23) to prevent the bridge from forwarding these transactions upstream to the primary PCI bus.

Outbound configuration cycles — secondary and primary — can support private PCI devices. Outbound ATUs provide a port programming model for outbound configuration cycles. Performing an outbound configuration cycle to either the primary or secondary PCI bus involves up to two internal bus cycles:

1) Writing the Outbound Configuration Cycle Address Register (primary or secondary) with the PCI address used during the configuration cycle. See the *PCI Local Bus Specification,* Revision 2.2 for information regarding configuration address cycle formats. This IB bus cycle enables the transaction.
2) Writing or reading the Outbound Configuration Cycle Data Register (primary or secondary). The i960 core processor cycle initiates the transaction. A read causes a configuration cycle read to the primary or secondary PCI bus with the address in the outbound configuration cycle address register. Similarly, a write initiates a configuration cycle write to PCI with the write data from the second processor cycle. Configuration cycles are non-burst and restricted to a single 32-bit word cycle. Internal bus burst writes and reads to the Outbound Configuration Cycle Data Register are disconnected after the first data phase.

Master aborts during outbound configuration reads result in master aborts being returned on the internal bus.

When the Configuration Cycle Data Register is written, the data is latched and forwarded to the PCI bus with the internal master issuing a disconnect with data for 32-bits only. This cycle will not receive an **I_ACK64#** from the ATU and therefore is defined as 32-bit only.

When the Configuration Cycle Data Register is read, the internal bus master is retried and the delayed cycle is issued. Refer to Section 15.2.2.5 for details on outbound read behavior.

Note that both the Configuration Cycle Address and Data registers are non-burstable. Software should only access these 4 registers with the single Dword read or write load/store operations. A burst attempt to these registers may result in incorrect or unexpected behavior.

Section 15.7, "Register Definitions" on page 15-54 describes an outbound configuration cycle address and data register definition and programming constraints. Note that while the programming model uses the register interface for outbound configuration cycles, from a hardware standpoint, the address is entered into the OTQ, configuration write data goes through the OWQ and configuration read data is returned in the ORQ.

Note: Outbound configuration cycle data registers are not physical registers. They are a 80303 I/O processor memory mapped addresses used to initiate a transaction with the address in the associated address register. Reads/writes to these registers return data from the PCI bus — not from the register.

15.2.4 PCI Multi-Function Device Swapping/Disabling

The 80303 I/O processor, in its default state, appears on the PCI bus as a multi-function device, with the Bridge as function 0 and the ATU as function 1. If necessary, these function numbers can be swapped, or the 80303 I/O processor can appear as a single function device, with either the ATU or the Bridge designated as the single function. The swapping is accomplished by setting or clearing bit 21 of the ATU Configuration Register - ATUCR and setting the value in the ATU Header Type Register - ATUHTR and the ATU Header Type Register - ATUHTR in the bridge from the i960 core processor. The 80303 I/O processor must be in mode 3 (core executing and configuration cycles retried) when executing the changes to these registers. The register settings are summarized in Table 15-3.

Table 15-3. PCI Multi-Function Device Swapping/Disabling Summary

Bridge Header Type Register (HTR)	ATU Header Type Register (ATUHTR)	ATU Configuration Register (ATUCR), Bit 21	80303 I/O processor Device Type	Bridge Function Number	ATU Function Number
1	1	0	Multi-Function (Default)	Function 0	Function 1
1	1	1	Multi-Function	Function 1	Function 0
0	0	0	Single Function	Function 0	Master-Aborts
0	0	1	Single Function	Master-Aborts	Function 0

Note: Configuring the 80303 I/O processor as a single function device is only recommended in situations where the host BIOS does not recognize multi-function devices and/or PCI-to-PCI Bridge configuration headers. It is up to the user to handle/disable error reporting for the disabled unit.

15.2.5 64-Bit PCI Operation

Both the PATU and the SATU are capable of PCI 64-bit operation to support data transfer rates of up to 528 MBytes/sec. The 64-bit PCI extensions add 39 additional signals to each ATU PCI interface. These signals and there functions are

- **AD[63:32]** - high order address/data bus
- **C/BE[7:4]#** - byte enables covering high order 4 bytes of data
- **PAR64** - even parity signal covering **AD[63:32]** and **C/BE[7:4]#**. Same timing as **PAR**
- **REQ64#** - used by a 64-bit master to request a 64-bit operation. Same timing as **FRAME#**
- **ACK64#** - used by a 64-bit capable target in response to **REQ64#** being asserted. Signifies to the master that the transaction can be completed with 64-bit transfers. Same timing as **DEVSEL#**.

At PCI bus reset, each individual PCI bus (primary and secondary) will independently sample their respective **REQ64#** signals. If this signal is low, the bus is 64-bit capable. The PCI to PCI Bridge Unit holds the information about 64-bit bus capability latched at the de-assertion of reset. The Primary Bus 64-Bit Capable bit (bit 8) of the Extended Bridge Control Register (EBCR) tells the PATU whether or not the bus it is connected to is 64-bit capable. The Secondary Bus 64-Bit Capable bit (Bit 9) of the Extended Bridge Control Register (EBCR) tells the SATU if the secondary bus is 64-bit capable. Refer to the Chapter 14, "PCI-to-PCI Bridge Unit" for details.

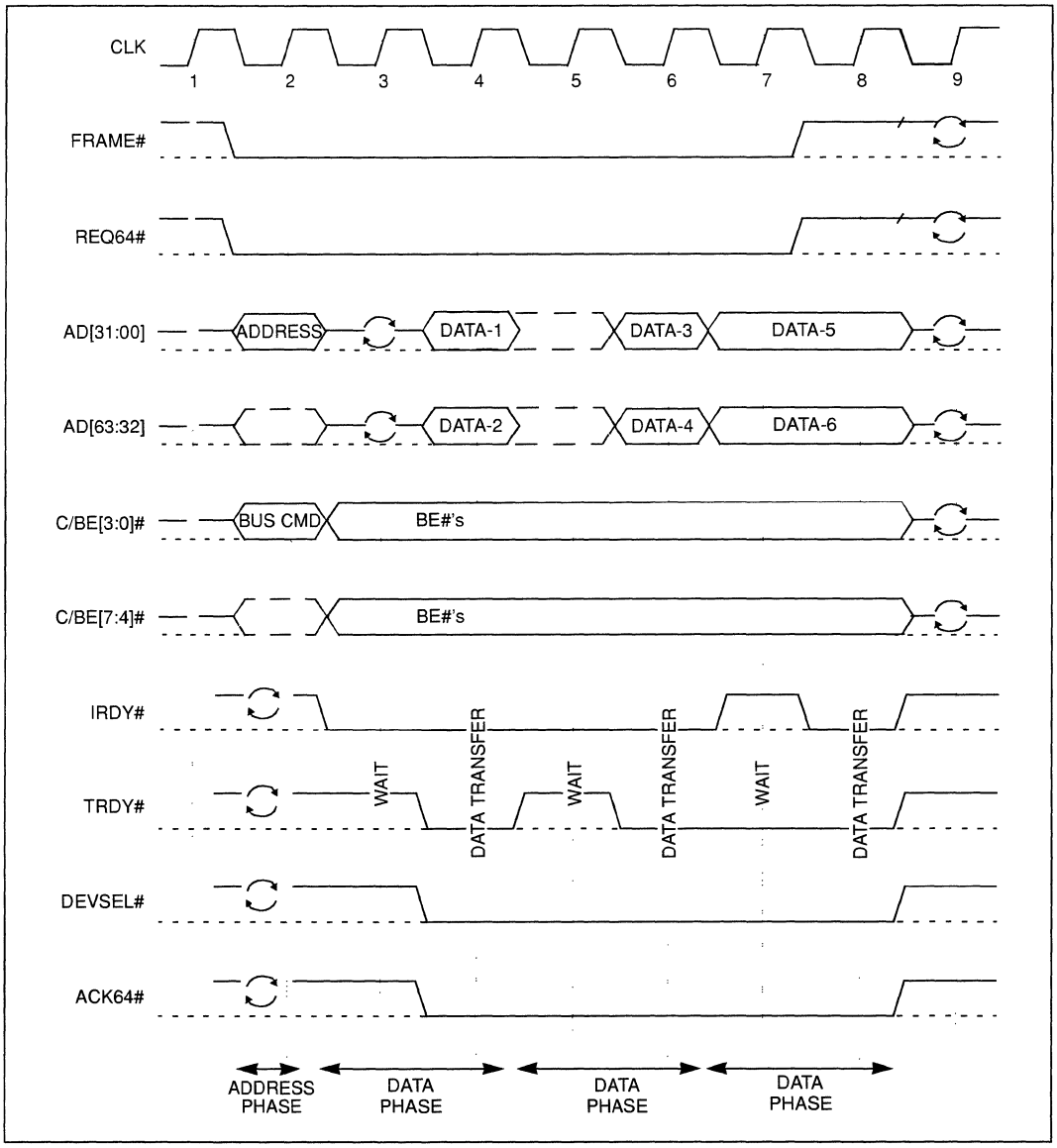
15.2.5.1 64-Bit Protocol

The 64-bit PCI extensions have been developed to coincide with the existing 32-bit protocol. The additional 32 bits of address/data require an additional 4 byte enables and a parity signal to cover them. The bus timing, protocol, and turn-around cycles behave exactly the same for the 64-bit signals as they do for the standard PCI interface signals with the exception of the 64-bit handshake signals referenced below.

The 64-bit handshake signals used by the 80303 I/O processor are **P_REQ64#** and **P_ACK64#** on the primary interface and **S_REQ64#** and **S_ACK64#** on the secondary interface. As a master, a PCI interface of the ATUs will assert **REQ64#** with **FRAME#** to indicate to the target that a 64-bit transaction is being requested. **REQ64#** is asserted and deasserted with the exact timing as **FRAME#** for the master state machines. When **REQ64#** is asserted, the target of the memory operation is required to assert **ACK64#** with the same timing as **DEVSEL#** to allow a 64-bit transaction to proceed. If **ACK64#** is not asserted with **DEVSEL#**, the master interface must revert to a 32-bit transaction. See Section 15.2.5.2 for details on 64-bit operation with 32-bit targets.

When **ACK64#** is asserted by the target of the transaction, a 64-bit transfer must proceed. As stated, a 64-bit transfer behaves exactly the same as a 32-bit transfer except that up to 8 bytes of data are transferred during each PCI data phase. For the 64-bit transfer, the **AD[63:32]** and **C/BE[7:4]#** are reserved during the address phase (assuming a SAC transfer). During the data phases, the master interface transfers up to 8 bytes of data on each of the 8 byte lanes defined by **C/BE[7:4]#**. As in a 32-bit transfer the master is capable of asserting any (or none) of the byte enables during each of the data phases within a burst transfer. Refer to Figure 15-8 for a diagram of a 64-bit transfer from a 64-bit target. **PAR64** for a 64-bit transfer has the same function and timing as **PAR** for a 32-bit transfer. **PAR64** must be asserted one clock after each address and data phase. 64-bit targets will qualify address parity checking using **PAR64** with the assertion of **REQ64#**. Although **AD[63:32]** and **C/BE[7:4]#** are reserved for SAC 64-bit transfers, parity must still be preserved and therefore stable values must be driven.

Figure 15-8. PCI 64-Bit Transfer from a 64-Bit Target



As a target, the slave state machines of both ATU PCI interfaces are capable of responding as a 64-bit target. When a PCI memory transaction is claimed by an ATU interface and the initiating master has requested a 64-bit transfer by asserting **REQ64#** with **FRAME#**, the ATU slave interface will assert and deassert **ACK64#** with the same timing and protocol as **DEVSEL#**. Furthermore, 64-bit slave operation is exactly like 32-bit operation with data being written or returned on both **AD[31:00]** and **AD[63:32]** using **C/BE[3:0]#** and **C/BE[7:4]#,** respectively. **PAR64** must be driven with the same timing as **PAR** for read operations.

As a target during write operations, the ATUs must make sure they contain enough data queue space (i.e. 8 bytes) to complete the next data transfer. Otherwise for less than 8 bytes of queue space, a Target Disconnect with Data must be signaled in the data phase prior to the data phase where the queue space will become full (defined as 7 bytes or less of queue space available).

15.2.5.2 64-Bit Operation with 32-Bit Targets

When a 64-bit transfer is requested by the PCI master interfaces by the assertion of **REQ64#**, it is not guaranteed that the target of the transaction is capable of performing the 64-bit request. In this case, **ACK64#** will remain deasserted when the target asserts **DEVSEL#** to claim the transaction. When a target signals that is cannot complete the transaction using 64-bit transfers, the ATU master interfaces are responsible for completing the transactions as a 32-bit master. Two possible conditions arise from a 32-bit target which does not respond with **ACK64#**:

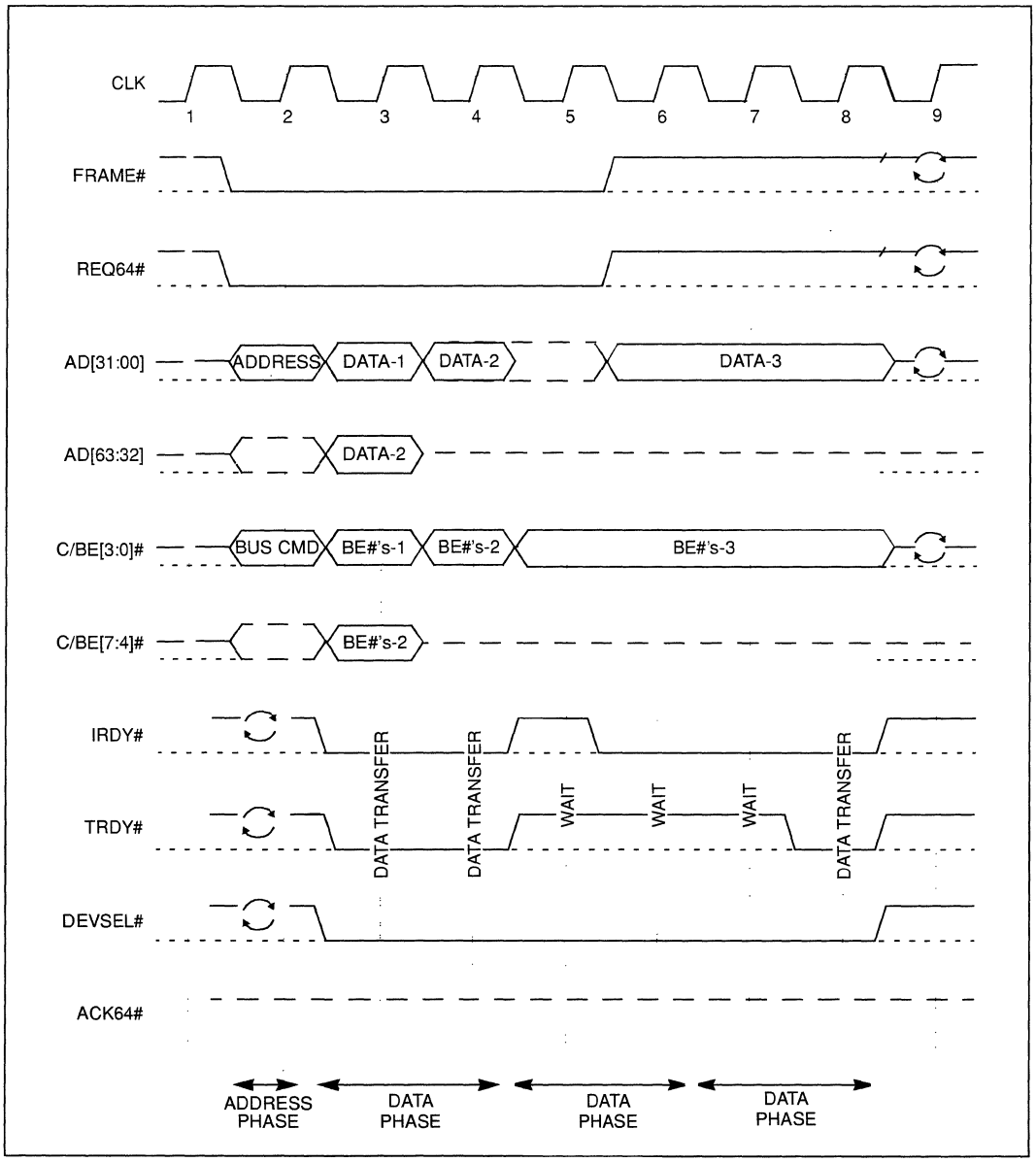
1. **ACK64#** deasserted but a burst can be sustained
2. **ACK64#** deasserted but a burst can not be sustained

If a 32-bit target does not respond with **ACK64#** and **STOP#**, it is capable of continuing a burst as a 32-bit target. For memory read requests, the ATU interfaces changes to 32-bit operation by only expecting read data on the lower byte lanes, **AD[31:0]**. The master interfaces continue requesting read data (by the continued asserting of **IRDY#**) as 32-bit masters. No master completions are prematurely signaled due to 32-bit target response. For memory write operations, the master interface may already have the first data phase on the bus by the time it is detected that **ACK64#** has not been asserted. The PATU and SATU master interfaces discontinue driving data on the upper 4 bytes during the second data phase. The second data phase of the burst now contains the data from the high 4 bytes of the first data phase. The master interface stops driving the **AD[63:32]** and **C/BE[7:4]#** during data phase 2 and all subsequent data phases of the burst write transfer. See Figure 15-9 for a diagram of this transaction. As a note, a disconnect after the first data phase of the burst transfer write will result in the continuation of the write transaction as a 32-bit master only (no **REQ64#**). This works similar to the write transfer disconnected in the first data phase described in the next paragraph.

If a 32-bit target does not respond with **ACK64#** but asserts **STOP#**, the target will not continue the burst. If a read or write request is made and **STOP#** without **TRDY#** is signaled (Retry), the master interface must repeat the original read or write request as a 64-bit transaction. If the target signals a disconnect with data (**STOP#** and **TRDY#**) on a write transaction, then only the lower 4 bytes of the 8 byte transfer have been delivered. The master state machines of the ATUs repeat the request as a 32-bit master (no **REQ64#** assertion) using the upper 4 bytes of data from the disconnected transaction on **AD[31:00]** and the next address (i.e. if address 00H was used in the first 64-bit request, address 04H is used in the next 32-bit request). A disconnect from a 32-bit target before an odd address results in a new transaction (if required) as a 32-bit master. A disconnect from a 32-bit target before an even address results in a new transaction as a 64-bit master (if required).

Note that 32-bit targets create special circumstances for **FRAME#** signaling. For 64-bit, single Qword transfers, **FRAME#** is driven low and then high immediately in the next clock signaling last data phase. Due to the potential of requiring two 32-bit data phases to complete what was originally intended as one 64-bit data phase, this is not possible. FRAME# must not be deasserted until after **ACK64#** is returned or not.

Figure 15-9. 64-Bit Write Request with 32-Bit Transfer



15.2.6 66 MHz Operation

Both the PATU and the SATU of the 80303 I/O processor are capable of PCI 66 MHz operation to support data transfer rates of up to 264 MBytes/sec with a 32-bit bus or 528 Mbytes/sec with a 64-bit bus. Differences between 33 MHz PCI and 66 MHz PCI are minimal. Both share the same protocol, and signal definitions. The 66 MHz PCI extension adds one additional signal to each PCI interface. The signal and its function is

- **M66EN** - when asserted for a PCI interface, indicates that interface will run at 66 MHz

Additionally, bit 5 of the Primary ATU Status Register - PATUSR is set to a 1_2 indicating that the PATU is capable of 66 MHz operation.

At PCI bus reset, each individual PCI bus (primary and secondary) will independently sample their respective **M66EN** signals. If this signal is high, the bus is 66 MHz capable, and in the case of the primary bus, the clock unit will be configured to accept a 66 MHz Primary PCI Bus clock.

The 66 MHz capable 80303 I/O processor supports the following primary and secondary bus frequency combinations:

- 66 MHz primary bus, 66 MHz secondary bus
- 66 MHz primary bus, 33 MHz secondary bus
- 33 MHz primary bus, 33 MHz secondary bus

The 80303 I/O processor does not support 33 MHz primary/66 MHz secondary bus operation, where the secondary bus is operating at twice the frequency of the primary bus. If **P_M66EN** is low (primary bus at 33 MHz), then the 80303 I/O processor pulls down **S_M66EN** to indicate that the secondary PCI bus is operating at 33 MHz.

The 80303 I/O processor generates clock signals **S_CLKOUT[6:0]** for the secondary bus devices and its own secondary interface (**S_CLKIN**). The 80303 I/O processor divides the primary bus clock **P_CLK** by two to generate the secondary bus clock outputs whenever the primary bus is operating at 66 MHz and the secondary bus is operating at 33 MHz. The 80303 I/O processor detects this condition when **P_M66EN** is high and **S_M66EN** is low.

For more details on the 80303 I/O processor's 66 MHz PCI clock scheme and the operation of the secondary PCI bus clocks, please see Chapter 25, "Clocking and Reset".

15.3 Messaging Unit

The Messaging Unit (MU) transfers data between the PCI system and the 80303 I/O processor and notifies the respective system when new data arrives. The MU is described in Chapter 16, "Messaging Unit".

The primary PCI window for messaging transactions is always the *first* 4 Kbytes of the inbound translation window defined by the Primary Inbound ATU Base Address Register (PIABAR) and the Primary Inbound ATU Limit Register (PIALR).

Access to the Messaging Unit from the secondary PCI interface is supported through a combination of the PCI-to-PCI Bridge Unit and the Secondary ATU. If bit 12 of the ATUCR is set, the first 4 KB of the SATU address is not claimed by the SATU but is allowed to be claimed by the secondary interface of the bridge. This address space must be within the range that is normally decoded and forwarded from secondary to primary by the bridge. Once the transaction is forwarded through the bridge, the setting of bit 12 allows the 80303 I/O processor to act as a master (bridge) and slave (ATU/Messaging Unit) at the same time on the primary interface. Refer to Section 15.7.38, "ATU Configuration Register - ATUCR" on page 15-98 for details of bit 12. The bridge unit does not perform any special "steering" of transactions from the secondary interface to the primary ATU/MU. The upstream bridge transaction must have a valid MU address to access the MU (first 4 KB of primary ATU address space).

All of the Messaging Unit errors are reported in the same manner as PATU errors. Error conditions and status can be found in the PATUSR and the PATUISR, see Section 15.6, "ATU Error Conditions" on page 15-38.

15.4 Expansion ROM Translation Unit

The primary inbound ATU supports one address range (defined by a base/limit register pair) used for the Expansion ROM. Refer to the *PCI Local Bus Specification,* Revision 2.2 for details on Expansion ROM format and usage.

During a powerup sequence, initialization code from Expansion ROM is executed once by the host processor to initialize the associated device. The code can be discarded once executed. Expansion ROM registers are described inSection 15.7.14, Section 15.7.32, and Section 15.7.33.

The inbound primary ATU supports an inbound Expansion ROM window which works like the inbound translation window. A read from the expansion ROM windows is forwarded to the internal bus and to the Memory Controller. The address translation algorithm is the same as the inbound translation; see Section 15.2.1.1, "Inbound Address Translation" on page 15-6. The only width Expansion ROM supported by the 80303 I/O processor Memory Controller is an 8-bit non-volatile device (FLASH/EPROM/ROM). The PATU uses standard 64-bit accesses on the internal bus and the responsibility for packing the data from the 8-bit device resides with the Memory Controller.

The Expansion ROM unit uses the primary ATU inbound transaction queue and the inbound read data queue. The address of the inbound delayed read cycle is entered into the P_ITQx queue and the delayed read completion data is returned in the P_IRQ. Expansion ROM writes are not supported and result in a Target Abort. The internal bus master interface will fill the P_IRQ read queue with a minimum of 8-bytes in response to a read on the PCI bus. As a PCI target, the Expansion ROM interface behaves as a standard ATU interface and is capable of returning a 64-byte access by the assertion of **P_ACK64#** in response to a 64-bit request.

15.5 ATU Queue Architecture

ATU operation and performance depends on the queueing mechanism implemented between the internal bus interface and PCI bus interface. As indicated in Figure 15-2, the ATU queue architecture consists of separate inbound and outbound queues for ATU. The function of each queue is described in the following sections.

15.5.1 Inbound Queues

The inbound data queues of the ATUs support transactions initiated on a PCI bus and targeted at either 80303 I/O processor local memory or a 80303 I/O processor memory mapped register. Table 15-4 details the name and sizes of the PATU and SATU inbound data queues.

Table 15-4. Inbound Queues

ATU	Queue Mnemonic	Queue Name	Queue Size (Bytes)
PATU	P_IWQ	Primary Inbound Write Data Queue	256
	P_IWQAD	Primary Inbound Write Address Queue	4 Transaction Addresses
	P_IRQ	Primary Inbound Read Data Queue	256
	P_IDWQ	Primary Inbound Delayed Write Queue	8
	P_ITQ1	Primary Inbound Transaction Queue 1	Address/Command
	P_ITQ2	Primary Inbound Transaction Queue 2	Address/Command
SATU	S_IWQ	Secondary Inbound Write Data Queue	256
	S_IWQAD	Secondary Inbound Write Address Queue	4 Transaction Addresses
	S_IRQ	Secondary Inbound Read Data Queue	256
	S_ITQ1	Secondary Inbound Transaction Queue 1	Address/Command
	S_ITQ2	Secondary Inbound Transaction Queue 2	Address/Command

15.5.1.1 Inbound Write Queue Structure

The PATU and SATU Inbound Write Queues consist of the inbound write data queues and the inbound write address queues. The inbound write data queue hold the data for memory write transactions moving from a PCI Bus to the internal bus and the address queues hold the corresponding address of the transactions in the data queues. The primary inbound write queue, P_IWQ, has a queue depth of 256 bytes and moves write transactions from the primary PCI bus to the internal bus. The corresponding address queue, P_IWQAD, is capable of holding 4 address entries. The queue pair is capable of holding up to 4 memory write (or MWI) transactions up to the size of the queue in a manner similar to the bridge unit write queues.

The secondary inbound write queue (S_IWQ) has a depth of 256 bytes and moves write transactions from the secondary PCI bus to the internal bus. The corresponding address queue, S_IWQAD, is capable of holding 4 address entries. This queue pair functions the same as the primary queue pair, holding up to 4 transactions of variable length up to the size of the data queue.

Memory write transactions fill the tail of the queue on the PCI bus and are drained from the head of the queue on the internal bus. The following rules apply to the PCI bus interface and govern the acceptance of data into the tail of IWQ and address into the tail of the IWQAD:

- A memory write operation claimed by the slave PCI interface on the PCI bus is accepted into the address and data queues if the queues are in a non-full state (see Section 15.7.49, "Primary ATU Queue Control Register - PAQCR" on page 15-112 and Section 15.7.50, "Secondary ATU Queue Control Register - SAQCR" on page 15-113). A Retry is signaled if this condition is not true when a transaction is first claimed by the slave interface.
- If the IWQ reaches a full state while filling, a disconnect with data is signaled to the master of the transaction on the data phase that fills the queue to a completely full state (no queue bytes remaining).

Memory write transactions are drained from the head of the queue when the master interface has acquired bus ownership and transaction ordering and priority have been satisfied (see Section 15.5.3). A memory write transaction is considered drained from the queue when the entire amount of data entered on the PCI bus has been accepted by the internal bus target. Error conditions resulting in the cancellation of a write transaction (master-abort) only flush the transaction at the head of the data and address queue. All other transactions within the queues are considered still valid. *Memory Write and Invalidate* transactions are treated like *Memory Write* transactions on the PCI interface and use the *Memory Write* command on the internal bus.

Transactions entering the tail of an empty queue (no previous write transactions reside in queue) are forwarded immediately to the head of the queue. A queue entry (8 bytes for either 64-bit or 32-bit data) is immediately added to the tail of the data queue when drained from the head of the queue on the target bus.

15.5.1.2 Inbound Read Queues and Inbound Transaction Queues

The inbound read queues are responsible for retrieving data from local memory and returning it to the PCI busses in response to a delayed read transaction initiated from a PCI master. The ATUs each have one IRQ for data only. The address of the transaction is held in a dedicated ITQ. P_ITQ1 and P_ITQ2 are dedicated to P_IRQ with a similar arrangement for the secondary ATU queues. Each IRQ holds the data from only one read transaction from the PCI bus. The read request cycle on PCI latches the read command and the address into the ITQ when the cycle is first initiated by the PCI master. The ATU IB master interface takes the translated address and the command and performs a read on the internal bus. Reads can be any of the PCI memory read command types using the ATU inbound translation or an inbound configuration read using the specific configuration cycle translation. The data from the read on the IB is stored in the IRQ until the PCI master initiates a read cycle that matches the initial request cycle in both command and address. Any data left in an IRQ after the delivery of a completion cycle on PCI is flushed. This is possible since all internal bus memory is considered prefetchable with no read side effects.

The exact amount of data read by the master state machine on the IB interface depends upon the read command used and how much data the PCI target device delivers. Table 15-5 shows the amount of data attempted to be read for the different memory read commands for both the primary and secondary ATUs. In addition, memory read streaming is used. This means that if an IRQ is currently being drained while it is being filled and the prefetch size is reached, the ATU internal bus master maintains the transaction and continues filling read data into the IRQ until it fills up. If the IRQ reaches a full state while being drained, the ATU internal bus master relinquishes the bus. No master wait states are inserted. If additional read prefetch data is entered into the queue after the draining master gives up the PCI bus, the data is flushed.

The function of the two transaction queues for each data queue is to allow the acceptance of up to two delayed read requests. While only 1 read completion can be occurring at any one time, the second DRR can be accepted to reduce the latency of accepting another DRR after the previous DRC has completed. For example, a DRR can be accepted into P_ITQ1. After the DRR has been accepted and the read starts on the internal bus, data will start filling P_IRQ from the internal bus side. While this is occurring the PCI slave interface is capable of accepting another independent read request into P_ITQ2. This read will only begin on the internal bus after a PCI master has performed a read completion cycle on PCI and has drained the read data associated with P_ITQ1 from P_IRQ. Under no circumstances will the read data queue hold read data from more than one transaction queue at a time.

Internal bus error conditions override all prefetch amounts. i.e. a master-abort and target-abort conditions.

Table 15-5. Inbound Read Prefetch Data Sizes

ATU	PCI Read Command	Prefetch Size (Bytes)
PATU	Memory Read	32
	Memory Read Line	128
	Memory Read Multiple	256
SATU	Memory Read	32
	Memory Read Line	128
	Memory Read Multiple	256

15.5.1.3 Inbound Delayed Write Queue

The IDWQ is present only in the primary ATU and is used specifically for inbound configuration write cycles to the ATUs. I/O Write transactions are not accepted by the PATU or SATU, and result in a Master Abort.

The IDWQ contains both address and data of a configuration write cycle. When the delayed write cycle is initiated on the PCI bus, the address and data are entered into the 8 byte queue, forwarded to the IB bus. The address translation, is specific configuration translation (Section 15.2.1.4). The transaction is forwarded to the IB bus once transaction ordering is satisfied and the translated write cycle is performed on the internal bus IB memory write command. The status of the transaction (normal completion) is maintained in the IDWQ for return to the PCI master on the initiating bus.

The IDWQ can only hold 32-bit data and should never be accessed from PCI with **P_REQ64#** active. In addition, the cycle should always return only 32-bits of data on the internal bus and should never receive an **I_ACK64#**.

15.5.2 Outbound Queues

The outbound ATU queues are used to hold read and write transactions from the core processor, directed at the PCI busses. Each ATU outbound queue structure has a separate read, write, and address queue. Table 15-6 contains information about both PATU and SATU outbound queues.

Table 15-6. Outbound Queues

ATU	Queue Mnemonic	Queue Name	Queue Size (Bytes)
PATU	P_OWQ	Primary Outbound Write Queue	16
	P_ORQ	Primary Outbound Read Queue	16
	P_OTQ	Primary Outbound Transaction Queue	Address/Command
SATU	S_OWQ	Secondary Outbound Write Queue	16
	S_ORQ	Secondary Outbound Read Queue	16
	S_OTQ	Secondary Outbound Transaction Queue	Address/Command

The outbound queues are capable of holding outbound memory read and write, I/O read, I/O write, and DAC transactions. The type of transaction used is defined by the internal bus address and the command used on the internal bus (memory write, memory read, memory read line, memory read multiple). See Section 15.2.2.1 and Section 15.2.2.2 for details on outbound address translation. For DAC cycles, each outbound transaction queue contains a separate register which contains the upper 32-bits of a 64-bit outbound transactions (see Section 15.7.29 and Section 15.7.44).

When the core processor (BIU) initiates an outbound write transaction, the address is entered into the OTQ (if empty). The data from the internal bus write is then entered into the OWQ and the transaction is forwarded to the PCI bus. When the write completes (or an error occurs), the address is flushed from the OTQ. Data is flushed only from master abort or target abort cases.

For outbound reads, the address is entered into the OTQ (if empty) and a retry is signaled to the master on the internal bus. Read data is prefetched (amounts based on Table 15-2) into the ORQ and once the full prefetch amount (or a target abort or master abort error) is reached, the data is allowed to be returned to the master on the internal bus.

The amount of data read during an outbound read cycle, depends on the internal bus read command presented during the address phase. Since all outbound reads are deterministic and not speculative prefetching, the ATU must complete the read before allowing the internal bus master access to the data. Table 15-2 shows read sizes used by the primary and secondary ATUs during outbound reads.

15.5.3 Transaction Ordering

Because the ATUs can process multiple transactions, they must maintain proper ordering to avoid deadlock conditions and improve throughput. The ATU transaction ordering rules used by the 80303 I/O processor are listed in Table 15-7 for the inbound direction and Table 15-8 for the outbound direction. The tables are based on the direction the transaction is moving, (i.e., the data for outbound delayed read moves in the same direction as the data for an inbound write or the address/command for an inbound read).

Table 15-7. ATU Inbound Data Flow Ordering Rules

Row Pass Column?		Inbound Write		Inbound Read Request	Inbound Configuration Write Request	Outbound Read Completion
		ATU Inbound Writes	MU Inbound Writes			
Inbound Write	ATU[1] Inbound Writes	No	No	No/Yes[2]	No	Yes
	MU[3] Inbound Writes	No	No	No	No	Yes
Inbound Read Request		No	No	No	No	Yes
Inbound Configuration Write Request[4]		No	No	No	No	Yes
Outbound Read Completion		No	No	Yes	No	No

1. ATU Primary and Secondary Inbound Write Queues.
2. The only situation where an ATU inbound write can pass an inbound read request is if there is both a delayed read completion and an inbound read request pending.
3. Messaging Unit Inbound Queue (Primary Only).
4. Not valid in Secondary ATU.

Table 15-8. ATU Outbound Data Flow Ordering Rules

Row Pass Column?	Outbound Write	Outbound Read Request	Inbound Read Completion	Inbound Delayed Write Completion
Outbound Write	No	No	Yes	Yes
Outbound Read Request	No	No	Yes	Yes
Inbound Read Completion	No	Yes	No	Yes
Inbound Delayed Write Completion[1]	Yes	Yes	Yes	No

1. Not valid for Secondary ATU.

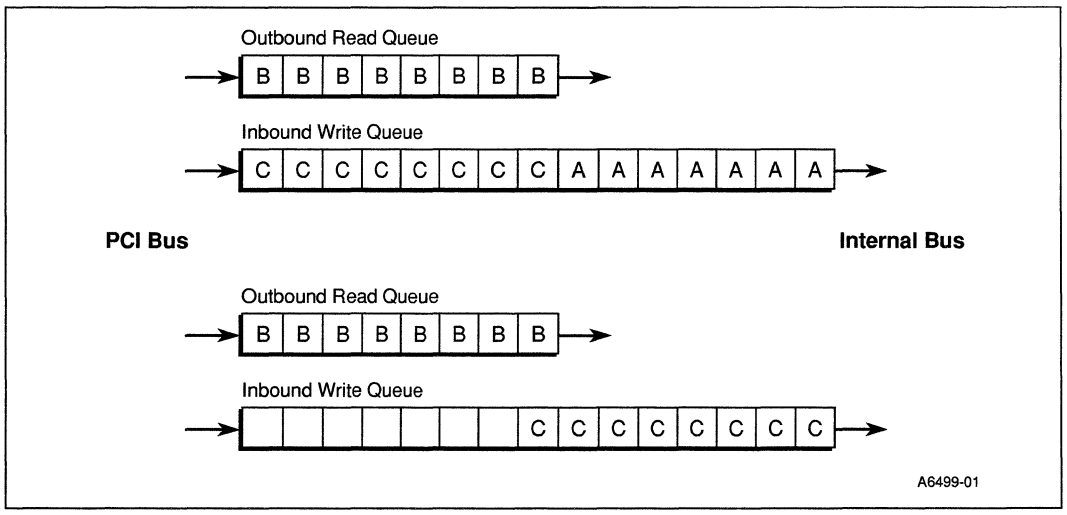
Term definitions used in Table 15-7 and Table 15-8 (PCI terms noted in parenthesis) are as follows:

- Inbound Write (PMW) - Data from a write cycle initiated on PCI and targeted at the internal bus. Note that the address is in a separate transaction queue and is not referenced. Inbound writes can also come in through the Messaging Unit which is part of the primary ATU.
- Inbound Read Request (DRR) - Address information from read transactions retried and delayed on PCI bus. Mastered on internal bus to retrieve data for Inbound Read Completion.
- Inbound Configuration Write Request - (DWR) - Address and data associated with a configuration write transaction from primary PCI and targeted at ATU PCI configuration address space. Once completed on internal bus, creates an Inbound Configuration Write Completion. Only available in the PATU.
- Outbound Read Completion (DRC) - Data read on PCI in process of being returned to the BIU on the internal bus. This data is completion cycle resulting from Outbound Read Request.
- Outbound Write (PMW) - Address and data from a write initiated on the internal bus and eventually completing on the PCI bus.
- Outbound Read Request (DRR) - Address/command of a delayed read cycle initiated on the internal bus. The read data is returned in the Outbound Read Completion cycle.
- Inbound Read Completion (DRC) - Data read on the internal bus in the process of being returned to the PCI bus. This data is the completion cycle for an Inbound Read Request.
- Inbound Configuration Write Completion (DWC) - Status of an inbound write configuration cycle traveling from internal bus back towards primary PCI bus. Only present in the PATU.

These transaction ordering rules define the way data moves in both directions through the ATUs. In Table 15-7 and Table 15-8 a **NO** response in a box means that based on ordering rules, the current transaction (row) can not pass previous transaction (column) under any circumstance. A **Yes** response in box means the current transaction is *allowed* to pass the previous transaction, but is not required to, based on whether a consistent view of data or prevention of deadlocks is needed.

In the case of inbound write operations, multiple transactions may exist within the x_IWQ and the corresponding x_IWQAD at any point in time. The ordering of these transactions is based on a time stamp basis. Transactions entering the queue are stamped with a relative time in relation to all other transactions moving in a similar direction.

Example 15-1. Inbound Queue Completion



In Example 15-1 on page 15-36, the inbound write and outbound read queues of an ATU are shown. In this example, transaction A entered the write queue at **Time 0**. Next, the ATU entered read data into the outbound read queue at **Time 1** (Transaction B). Finally, before the previous transactions could be cleared, another inbound write, Transaction C, was entered into the IWQ. The ordering in Table 15-7 states that nothing can pass an inbound write and therefore Transaction A must complete on the internal bus before Transaction B since an outbound read completion can not pass an inbound write. Also, Transaction A must complete before Transaction C since an inbound write can not pass another inbound write. Once Transaction A completes, Transaction C moves to the head of the IWQ. The two transactions at the head of the queues moving data in an inbound direction are now Transaction C, an inbound write, and Transaction B, an outbound read completion. Ordering states that an inbound write may pass an outbound read completion. This means that the priority mechanism now takes over to decide which will complete (defined in the next section). In this case, if the BIU acquires the internal bus first, Transaction B will complete. If the ATU acquires the internal bus first, Transaction C will complete. Note that ordering enforced the completion of Transaction A but priority dictated the completion of Transactions B and C.

The first action performed to determine which transaction is allowed to proceed (either inbound or outbound) is to apply the rules of ordering as defined in Table 15-7 and Table 15-8. Any box marked **No** must be satisfied first. For example, if an inbound read request is in P_ITQ1 and it was latched *after* the data in the P_IDWQ arrived (this is a configuration write), then ordering states that an Inbound Read Request may not pass an Inbound Configuration Write Request. Therefore, the Inbound Configuration Write Request must be cleared out of P_IDWQ before the Inbound Read Request is attempted on the internal bus. Once transaction ordering is satisfied, the boxes marked **Yes** are now resolved.

15.6 ATU Error Conditions

PCI and internal bus error conditions cause the ATU state machines to exit normal operation and return to idle states. In addition, status bits are set to inform error handling code of the exact cause of the error condition. The 80303 I/O processor ATUs use a similar error handling scheme for PCI interrupts as the PCI to PCI Bridge Unit. All of the Messaging Unit errors are reported in the same manner as PATU errors. Error conditions and status can be found in the PATUSR and the PATUISR. The basic flow for a PCI error is as follows:

- Set the bit in the ATU Status Register which corresponds to the error condition (master abort, target abort, etc.)
- Set the bit in the ATU Interrupt Status Register which corresponds to the error condition (master abort, target abort, etc.). This function is maskable for all PCI error conditions.
- The setting of the bit in the ATU Interrupt Status Register results in a **NMI#** interrupt being driven to the i960 core processor

Error conditions on one side of the ATU are generally propagated to the other side of the ATU and have different effects depending on the error. Error conditions and their effects are described in the following sections.

PCI bus error conditions and the action taken on the bus are defined within the *PCI Local Bus Specification,* Revision 2.2. The ATU adheres to the error conditions defined within the PCI specification for both master and slave operation. Error conditions on the internal bus are caused by an ECC error from the Memory Controller (see Section 13.4, "Interrupts/Error Conditions" on page 13-42 for details on memory controller error conditions) or by incorrect addressing resulting in an internal master abort. All actions on the PCI Bus for error situations are dependent on the error control bits found in the Primary ATU and Secondary ATU Control Registers. See Section 15.7, "Register Definitions" on page 15-54.

The following sections detail all ATU error conditions on the PCI bus and the 80303 I/O processor internal bus, action taken on these conditions, and the status and control bits associated with error handling.

15.6.1 Address Parity Errors on the PCI Interface

The ATUs must detect and report address parity errors for transactions on both PCI buses. If an address parity error occurs on the PCI interface of either ATU, the 80303 I/O processor performs the following actions based on the constraints specified:

Table 15-9. Address Parity Errors on the PCI Interface

Primary ATU	Secondary ATU
If the Parity Error Response bit in the PATUCMD is set, the PATU will *not* claim the transaction by *not* asserting **P_DEVSEL#**, allowing a master abort to occur. If the Parity Error Response Enable bit in the PATUCMD is cleared, the PATU takes normal action and allows the transaction to proceed.	If the Parity Error Response bit in the SATUCMD is set, the SATU will *not* claim the transaction by *not* asserting **S_DEVSEL#**, allowing a master abort to occur. If the Parity Error Response Enable bit in the SATUCMD is cleared, the SATU takes normal action and allows the transaction to proceed.
Assert **P_SERR#** if the **P_SERR#** Enable bit and Parity Error Response bit in the PATUCMD are both set.	Assert **S_SERR#** if the **S_SERR#** Enable bit and Parity Error Response bit in the SATUCMD are both set.
Set the **P_SERR#** Asserted bit in the PATUSR if the **P_SERR#** Enable bit and Parity Error Response bit in the PATUCMD are both set.	Set the **S_SERR#** Asserted bit in the SATUSR if the **S_SERR#** Enable bit and Parity Error Response bit in the SATUCMD are both set.
Set the Detected Parity Error bit in the PATUSR	Set the Detected Parity Error bit in the SATUSR
If the PATU **P_SERR#** Asserted Interrupt Mask Bit in the PATUIMR is clear, set the **P_SERR#** Asserted bit in the PATUISR, if set, no action.	If the SATU **S_SERR#** Asserted Interrupt Mask Bit in the SATUIMR is clear, set the **S_SERR#** Asserted bit in the SATUISR, if set, no action.
If the PATU **P_SERR#** Detected Interrupt Mask Bit in the ATUCR is clear, set the **P_SERR#** Detected bit in the PATUISR, if set, no action.	If the SATU **S_SERR#** Detected Interrupt Mask Bit in the ATUCR is clear, set the **S_SERR#** Detected bit in the SATUISR, if set, no action.
If the PATU Detected Parity Error Interrupt Mask bit in the PATUIMR is clear, set the Detected Parity Error bit in the PATUISR. If set, no action	If the SATU Detected Parity Error Interrupt Mask bit in the SATUIMR is clear, set the Detected Parity Error bit in the SATUISR. If set, no action

15.6.2 Data Parity Errors on the PCI Interface

Two kinds of data parity errors can occur on the PCI interface; errors as a master and errors as a slave. For errors as a master (outbound transactions), the ATUs will detect data parity errors on reads and record data parity errors occurring at the target for writes. For errors as a slave device (inbound transactions), the ATUs will detect data parity errors during write transactions and take no action for data parity errors during read transactions.

15.6.2.1 Outbound Read Data Parity Errors - Master

Data parity errors occurring during read operations initiated by the ATU are recorded, **PERR#** is asserted (if enabled) and the data is returned to the initiator on the internal bus. The entire prefetch amount of data is read and the transaction is never terminated with master completion in response to the data parity error. Specifically, the following actions with the given constraints are taken on both the primary and secondary ATUs:

Table 15-10. Outbound Read Data Parity Errors - Master

Primary ATU	Secondary ATU
P_PERR# is asserted two clocks cycles following the data phase in which the data parity error is detected on the primary bus. This is only done if the Parity Error Response bit in the PATUCMD is set.	S_PERR# is asserted two clocks cycles following the data phase in which the data parity error is detected on the secondary bus. This is only done if the Parity Error Response bit in the SATUCMD is set.
The Master Parity Error bit in the PATUSR is set if the Parity Error Response bit in the PATUCMD is set.	The Master Parity Error bit in the SATUSR is set if the Parity Error Response bit in the SATUCMD is set.
The Detected Parity Error bit in the PATUSR is set	The Detected Parity Error bit in the SATUSR is set
If the PATU PCI Master Parity Error Interrupt Mask Bit in the PATUIMR is clear, set the PCI Master Parity Error bit in the PATUISR, if set, no action.	If the SATU PCI Master Parity Error Interrupt Mask Bit in the SATUIMR is clear, set the PCI Master Parity Error bit in the SATUISR, if set, no action.
If the PATU Detected Parity Error Interrupt Mask bit in the PATUIMR is clear, set the Detected Parity Error bit in the PATUISR. If set, no action	If the SATU Detected Parity Error Interrupt Mask bit in the SATUIMR is clear, set the Detected Parity Error bit in the SATUISR. If set, no action

Outbound read parity errors, as stated, will result in the bad data being delivered back to the initiator on the internal bus of the 80303 I/O processor.

15.6.2.2 Outbound Write Data Parity Errors - Master

Data parity errors occurring during write operations initiated by the ATU may record the assertion of **PERR#** from the target on the PCI Bus. When an error occurs, the ATUs will continue writing data to the target to clear the OWQ of the current outbound write transaction. Specifically, the following actions with the given constraints are taken on both the primary and secondary ATUs:

Table 15-11. Outbound Write Data Parity Errors - Master

Primary ATU	Secondary ATU
If **P_PERR#** is sampled active and the Parity Error Response bit in the PATUCMD is set, set the Master Parity Error bit in the PATUSR. If the Parity Error Response bit in the PATUCMD is clear, no action is taken.	If **S_PERR#** is sampled active and the Parity Error Response bit in the SATUCMD is set, set the Master Parity Error bit in the SATUSR. If the Parity Error Response bit in the SATUCMD is clear, no action is taken.
If the PATU PCI Master Parity Error Interrupt Mask Bit in the PATUIMR is clear, set the PCI Master Parity Error bit in the PATUISR, if set, no action.	If the SATU PCI Master Parity Error Interrupt Mask Bit in the SATUIMR is clear, set the PCI Master Parity Error bit in the SATUISR, if set, no action.

Outbound write parity errors, as stated, will not result in a master completion. In addition, if the target terminates the transaction (disconnect), the ATU master must reinitiate the transaction to clear the data from the OWQ.

15.6.2.3 Inbound Read Data Parity Errors - Slave

Inbound read data parity errors occur when read data delivered from the IRQ is detected as having bad parity by the master of the transaction who is receiving the data. The master may optionally report the error to the system by asserting **PERR#**. As a slave device in this scenario, no action is required and no error bits are set.

15.6.2.4 Inbound Write Data Parity Errors - Slave

Data parity errors occurring during write operations received by the ATU may assert **PERR#** on the PCI Bus. When an error occurs, the ATUs will continue accepting data until the master of the write transaction completes or a queue fill condition is reached. Specifically, the following actions with the given constraints are taken on both the primary and secondary ATUs:

Table 15-12. Inbound Write Data Parity Errors - Slave

Primary ATU	Secondary ATU
P_PERR# is asserted two clocks cycles following the data phase in which the data parity error is detected on the primary bus. This is only done if the Parity Error Response bit in the PATUCMD is set.	**S_PERR#** is asserted two clocks cycles following the data phase in which the data parity error is detected on the secondary bus. This is only done if the Parity Error Response bit in the SATUCMD is set.
The Detected Parity Error bit in the PATUSR is set	The Detected Parity Error bit in the SATUSR is set
If the PATU Detected Parity Error Interrupt Mask bit in the PATUIMR is clear, set the Detected Parity Error bit in the PATUISR. If set, no action	If the SATU Detected Parity Error Interrupt Mask bit in the SATUIMR is clear, set the Detected Parity Error bit in the SATUISR. If set, no action

15.6.2.5 Inbound Configuration Write Data Parity Errors - Slave

To allow for correct data parity calculations for delayed write transactions, the primary ATU will delay the assertion of **P_STOP#** (signalling a Retry) until **P_PAR** is driven by the master. A parity error during a delayed write transaction (inbound configuration write cycle) can occur in any of the following parts of the transactions:

- During the initial Delayed Write Request cycle on the primary PCI bus when the PATU latches the address/command and data for delayed delivery to the internal configuration register.
- During the Delayed Write Completion cycle on the primary PCI bus when the ATU will deliver the status of the operation back to the original master.

The 80303 I/O processor's primary ATU PCI interface has the following responses to a delayed write parity error for inbound transactions during Delayed Write Request cycles with the given constraints:

- If the Parity Error Response bit in the PATUCMD is set, the primary ATU asserts **P_TRDY#** (disconnects with data) and two clock cycles later asserts **P_PERR#** notifying the initiator of the parity error. The delayed write cycle in not enqueued and forwarded to the internal bus.

  If the Primary Parity Error Response bit in the PATUCMD is cleared, the primary ATU retries the transaction by asserting **P_STOP#** and enqueues the Delayed Write Request cycle to be forwarded to the internal bus. **P_PERR#** is not asserted.
- The Detected Parity Error bit is set in the Primary ATU Status Register (PATUSR).
- If the PATU Detected Parity Error Interrupt Mask bit in the PATUIMR is clear, set the Detected Parity Error bit in the PATUISR. If set, no action

For the original write transaction to be completed, the initiator will retry the transaction on the PCI bus and the PATU will return the status from the internal bus, completing the transaction.

For the Delayed Write Completion transaction on the primary PCI bus where a data parity error occurs and therefore does not agree with the status being returned from the internal bus (i.e. status being returned is normal completion) the primary ATU performs the following actions with the given constraints:

- If the Parity Error Response Bit is set in the PATUCMD, the primary ATU asserts **P_TRDY#** (disconnects with data) and two clocks later asserts S_PERR#. The Delayed Completion cycle in the IDWQ remains since the data of retried command did not match the data within the queue.

  If the Parity Error Response Bit is clear in the PATUCMD, the primary ATU will retry the transaction with no other response. A new transaction is not enqueued due to queue architecture constraints (see Section 15.5.1.1).
- The Detected Parity Error bit is set in the Primary ATU Status Register (PATUSR).
- If the PATU Detected Parity Error Interrupt Mask bit in the PATUIMR is clear, set the Detected Parity Error bit in the PATUISR. If set, no action.

15.6.3 Master Aborts on the PCI Interface

As a master on the PCI bus, the ATUs can encounter master abort conditions during outbound read and write transactions. A master abort is signaled when the target of the transaction does not assert **DEVSEL#** within 5 clocks of the assertion of **FRAME#**. The following action with the given constraints are performed by the primary and secondary ATUs when a master abort is detected by the PCI master interface during an outbound read or write transaction:

Table 15-13. Master Aborts on the PCI Interface

Primary ATU	Secondary ATU
Set the Master Abort bit (bit 13) in the PATUSR	Set the Master Abort bit (bit 13) in the SATUSR
If the PATU PCI Master Abort Interrupt Mask bit in the PATUIMR is clear, set the PCI Master Abort bit in the PATUISR. If set, no action	If the SATU PCI Master Abort Interrupt Mask bit in the SATUIMR is clear, set the PCI Master Abort bit in the SATUISR. If set, no action
If an outbound write, flush the write data in the P_OWQ and the address in the P_OTQ	If an outbound write, flush the write data in the S_OWQ and the address in the S_OTQ
If an outbound read, return the master abort condition to the internal master when the completion cycle is allowed to proceed on the internal bus. Flush the address from the P_OTQ.	If an outbound read, return the master abort condition to the internal master when the completion cycle is allowed to proceed on the internal bus. Flush the address from the S_OTQ.

For the read case, the BIU is responsible for completing a transaction to the core processor (see Chapter 22, "I2C Bus Interface Unit").

As a target, the ATU PCI interface is capable of creating a master abort case during inbound reads. If the inbound read transaction is master aborted on the internal bus, the ATU holding the Delayed Request Cycle will purposely master abort the PCI initiator during a Delayed Completion retry cycle on the PCI bus. After this has occurred, the read transaction is flushed from the Inbound Transaction Queue (ITQ).

15.6.4 Target Aborts on the PCI Interface

Target abort can be signaled to the ATUs by PCI targets during outbound transactions and a target abort can be initiated by the ATUs during inbound transactions where the internal bus cycle resulted in a Target Abort from the memory controller due to an ECC error.

An inbound read transaction will result in a PCI bus target abort when an ECC error was received from the internal bus memory controller, the ATU ECC Target Abort Enable bit is set, and the Qword aligned data phase that received a target abort on the internal bus is requested on the PCI bus.

The following actions with the given constraints are performed by the primary and secondary ATUs when a target abort is signaled by the PCI slave interface during an inbound read or write transaction:

Table 15-14. Target Aborts on the PCI Interface

Primary ATU	Secondary ATU
Set the Target Abort (target) bit (bit 11) in the PATUSR	Set the Target Abort (target) bit (bit 11) in the SATUSR
If the PATU PCI Target Abort (target) Interrupt Mask bit in the PATUIMR is clear, set the PCI Target Abort (target) bit in the PATUISR. If set, no action	If the SATU PCI Target Abort (target) Interrupt Mask bit in the SATUIMR is clear, set the PCI Target Abort (target) bit in the SATUISR. If set, no action
If an inbound read, the P_IRQ is flushed after the completion cycle is performed on the primary PCI bus.	If an inbound read, the S_IRQ is flushed after the completion cycle is performed on the secondary PCI bus.

As a master during outbound transactions, the ATUs can receive target aborts from their PCI targets. For outbound writes, the transaction in the OWQ is flushed and for outbound reads, the target abort is delivered back to the initiator on the internal bus. The following actions with the given constraints are performed by the primary and secondary ATUs when a target abort is detected by the PCI master interface during an outbound read or write transaction:

Table 15-15. Target Aborts on the Primary and Secondary ATUs

Primary ATU	Secondary ATU
Set the Target Abort (master) bit (bit 12) in the PATUSR	Set the Target Abort (target) bit (bit 12) in the SATUSR
If the PATU PCI Target Abort (master) Interrupt Mask bit in the PATUIMR is clear, set the PCI Target Abort (master) bit in the PATUISR. If set, no action	If the SATU PCI Target Abort (master) Interrupt Mask bit in the SATUIMR is clear, set the PCI Target Abort (master) bit in the SATUISR. If set, no action
If an outbound write transaction, flush the P_OWQ and the P_OTQ.	If an outbound write transaction, flush the S_OWQ and the S_OTQ.
If an outbound read transaction, return the target abort condition to the initiator on the internal bus through the P_ORQ.	If an outbound read transaction, return the target abort condition to the initiator on the internal bus through the S_ORQ.

15.6.5 SERR# Assertion and Detection

The primary and secondary ATUs are capable of reporting error conditions through the use of the **P_SERR#** output and the S_SERR# output respectively.

The following conditions may result in the assertion of **P_SERR#** by the primary ATU:

- An address parity error is detected by the PATU PCI interface and the Parity Error Response bit and the **P_SERR#** Enable are set in the PATUCMD.
- An inbound write transaction is target aborted when the transaction is attempted on the internal bus, the Primary ATU Inbound Error **P_SERR#** Enable bit in the PATUIMR is set by the memory controller, and the **P_SERR#** Enable bit is set in the PATUCMD.
- An inbound write transaction is master aborted on the internal bus, the Primary ATU Inbound Error **P_SERR#** Enable bit in the PATUIMR is set, and the **P_SERR#** Enable bit is set in the PATUCMD.
- The **P_SERR#** Manual Assertion bit in the ATUCR has been set by the core processor and the **P_SERR#** Enable bit is set in the PATUCMD.

The following conditions may result in the assertion S_SERR# by the secondary ATU:

- An address parity error is detected by the SATU PCI interface and the Parity Error Response bit and the S_SERR# Enable are set in the SATUCMD.
- An inbound write transaction is target aborted by the memory controller when the transaction is attempted on the internal bus, the Secondary ATU Inbound Error S_SERR# Enable bit in the SATUIMR is set, and the S_SERR# Enable bit is set in the SATUCMD.
- An inbound write transaction is master aborted on the internal bus, the Secondary ATU Inbound Error S_SERR# Enable bit in the SATUIMR is set, and the S_SERR# Enable bit is set in the SATUCMD.
- The S_SERR# Manual Assertion bit in the ATUCR has been set by the core processor and the S_SERR# Enable bit is set in the SATUCMD.

Note that the **SERR#** manual assertion bits must be cleared manually before they can be set again resulting in **SERR#** asserted. Refer to Section 15.7.38, "ATU Configuration Register - ATUCR" on page 15-98 for details. For S_SERR# assertions by the SATU, the bridge must be programmed to pass the error upstream for detection by a host processor.

The following actions with the given constraints are performed by the primary and secondary ATUs when **SERR#** is asserted by the PCI interface:

Table 15-16. SERR# Assertion on the Primary and Secondary ATUs

Primary ATU	Secondary ATU
Set the **P_SERR#** Asserted bit in the PATUSR	Set the S_SERR# Asserted bit in the SATUSR
If the PATU **P_SERR#** Asserted Interrupt Mask bit in the PATUIMR is clear, set the **P_SERR#** Asserted bit in the PATUISR. If set, no action	If the SATU S_SERR# Asserted Interrupt Mask bit in the SATUIMR is clear, set the S_SERR# Asserted bit in the SATUISR. If set, no action
If the PATU **P_SERR#** Detected Interrupt Mask bit in the ATUCR is clear, set the **P_SERR#** Detected bit in the PATUISR. If set, no action	If the SATU S_SERR# Detected Interrupt Mask bit in the ATUCR is clear, set the S_SERR# Detected bit in the SATUISR. If set, no action

The following actions with the given constraints are performed by the primary and secondary ATUs when **SERR#** is detected by the PCI interface:

Table 15-17. SERR# Detection On the Primary and Secondary ATUs

Primary ATU	Secondary ATU
If the PATU **P_SERR#** Detected Interrupt Mask bit in the ATUCR is clear, set the **P_SERR#** Detected bit in the PATUISR. If set, no action	If the SATU S_SERR# Detected Interrupt Mask bit in the ATUCR is clear, set the S_SERR# Detected bit in the SATUISR. If set, no action

Note: Whenever the ATU asserts **SERR#**, both the asserted and detected status bits are set in the corresponding ISR. To mask an **NMI#** interrupt to the core when either the PATU or SATU asserts **SERR#**, both the **SERR#** asserted mask bit and the **SERR#** detected mask bit must be set. To mask and NMI# when either of the ATUs have detected **SERR#** (from some other device on one of the PCI interfaces), just the corresponding **SERR#** detected mask bit must be set.

15.6.6 Internal Bus Error Conditions

The 80303 I/O processor internal bus uses a protocol similar to the PCI specification. As such, master abort and target abort conditions are valid error states on the bus. The error handling protocol for internal bus conditions is similar to the PCI bus error protocol. An internal bus error will result in a bit being set in the Primary or Secondary Interrupt Status Registers at which time an interrupt is driven to the core processor. Unlike PCI errors, internal bus error conditions are not maskable.

The following sections detail internal bus error conditions for the primary and secondary ATUs.

15.6.6.1 Master Abort on the Internal Bus

A master abort on the internal bus is seen by the ATUs when the inbound translated address presented on the internal bus is not claimed by the assertion of **I_DEVSEL#**. As a slave device, the ATUs will return a master abort (by not asserting **I_DEVSEL#**) during an outbound read DRC cycle on the internal bus in response to a master abort on the PCI interface.

The following action with the given constraints are performed by the primary and secondary ATUs when a master abort is detected by the internal master interface during an inbound write transaction:

Table 15-18. Master Abort During an Inbound Write Transaction

Primary ATU	Secondary ATU
Set the Internal Bus Master Abort bit (bit 7) in the PATUISR	Set the Internal Bus Master Abort bit (bit 7) in the SATUISR
If the inbound write transaction is still active on the primary PCI interface, notify the primary PCI slave interface to disconnect the transaction.	If the inbound write transaction is still active on the secondary PCI interface, notify the secondary PCI slave interface to disconnect the transaction.
If the Primary Inbound Error **P_SERR#** Enable bit is set and the **P_SERR#** Enable bit is set in the PATUCMD, assert **P_SERR#** on the primary interface. If both bits are not set, take no action.	If the Secondary Inbound Error S_SERR# Enable bit is set and the S_SERR# Enable bit is set in the SATUCMD, assert S_SERR# on the secondary interface. If both bits are not set, take no action.
If **P_SERR#** is asserted, set the **P_SERR#** Asserted bit in the PATUSR	If S_SERR# is asserted, set the S_SERR# Asserted bit in the SATUSR
If **P_SERR#** is asserted and the PATU **P_SERR#** Asserted Interrupt Mask bit in the PATUIMR is clear, set the **P_SERR#** Asserted bit in the PATUISR. If set, no action	If S_SERR# is asserted and the SATU S_SERR# Asserted Interrupt Mask bit in the SATUIMR is clear, set the S_SERR# Asserted bit in the SATUISR. If set, no action
If **P_SERR#** is asserted and the PATU **P_SERR#** Detected Interrupt Mask bit in the ATUCR is clear, set the **P_SERR#** Detected bit in the PATUISR. If set, no action	If S_SERR# is asserted and the SATU S_SERR# Detected Interrupt Mask bit in the ATUCR is clear, set the S_SERR# Detected bit in the SATUISR. If set, no action
Flush the transaction that was master aborted from the P_IWQ.	Flush the transaction that was master aborted from the S_IWQ.

The Internal Bus Master Abort bit is non-maskable and always results in an **NMI#** interrupt being driven to the core processor.

The following action with the given constraints are performed by the primary and secondary ATUs when a master abort is detected by the internal master interface during an inbound read transaction:

Table 15-19. Master Abort During an Inbound Read Transaction

Primary ATU	Secondary ATU
Set the Internal Bus Master Abort bit (bit 7) in the PATUISR	Set the Internal Bus Master Abort bit (bit 7) in the SATUISR
Return a master abort condition to the initiating master during the delayed completion cycle on the primary PCI bus. No data is ever read from the internal bus and returned to the primary PCI bus.	Return a master abort condition to the initiating master during the delayed completion cycle on the secondary PCI bus. No data is ever read from the internal bus and returned to the secondary PCI bus.
Flush the transaction that was master aborted from the P_ITQ after the master abort is delivered on the PCI interface.	Flush the transaction that was master aborted from the S_ITQ after the master abort is delivered on the PCI interface.

The Internal Bus Master Abort bit is non-maskable and will always result in an **NMI#** interrupt being driven to the core processor.

As a slave device on the internal bus, the ATUs can return a master abort to the BIU in response to a master abort seen on the PCI interface during a delayed read cycle. In this scenario, the master abort is detected on the PCI interface during the read. Once this occurs, the ATU notifies the internal bus arbiter to allow the BIU to acquire the bus and after the assertion of **I_FRAME#**, the ATU fails to return an **I_DEVSEL#** signalling a master abort to the internal bus master (the BIU). No error conditions are recorded by the slave interface of the ATUs during master abort operations, since they are already recorded by the PCI interface.

15.6.6.2 Target Abort on the Internal Bus

Target Aborts can be seen by the internal bus master interface during inbound read and write operations to the memory controller. They are signaled as a slave device in response to a target abort encountered during outbound read. During inbound operations, the memory controller is capable of signalling a target abort when a multi-bit, unrecoverable ECC error is encountered. This can occur during writes of less than 64-bits and during any read operation. During outbound read operations, a delayed read cycle, that is target aborted on the PCI bus, results in a target abort being driven back to the BIU on the internal bus. Outbound writes will not see target aborts because they are always fully posted.

The following action with the given constraints are performed by the primary and secondary ATUs when a target abort is detected by the internal master interface during an inbound write transaction:

Table 15-20. Target Abort During an Inbound Write Transaction

Primary ATU	Secondary ATU
If the Primary Inbound Error **P_SERR#** Enable bit is set and the **P_SERR#** Enable bit is set in the PATUCMD, assert **P_SERR#** on the primary interface. If both bits aren't set, take no action.	If the Secondary Inbound Error S_SERR# Enable bit is set, and the S_SERR# Enable bit is set in the SATUCMD, assert S_SERR# on the secondary interface. If both bits aren't set, take no action.
If **P_SERR#** is asserted, set the **P_SERR#** Asserted bit in the PATUSR.	If S_SERR# is asserted, set the S_SERR# Asserted bit in the SATUSR.
If **P_SERR#** is asserted and the PATU **P_SERR#** Asserted Interrupt Mask bit in the PATUIMR is clear, set the **P_SERR#** Asserted bit in the PATUISR. If set, no action.	If S_SERR# is asserted and the SATU S_SERR# Asserted Interrupt Mask bit in the SATUIMR is clear, set the S_SERR# Asserted bit in the SATUISR. If set, no action.
If **P_SERR#** is asserted and the PATU **P_SERR#** Detected Interrupt Mask bit in the ATUCR is clear, set the **P_SERR#** Detected bit in the PATUISR. If set, no action	If S_SERR# is asserted and the SATU S_SERR# Detected Interrupt Mask bit in the ATUCR is clear, set the S_SERR# Detected bit in the SATUISR. If set, no action
If the inbound write transaction is still active on the primary PCI interface, notify the primary PCI slave interface to disconnect the transaction.	If the inbound write transaction is still active on the primary PCI interface, notify the secondary PCI slave interface to disconnect the transaction.

The Memory Controller is responsible for creating the core processor **NMI#** interrupt. The inbound write queue (IWQ) is not cleared and the ATU internal bus master interface will re-arbitrate for the internal bus and eventually drain the transaction which was target aborted from the queue.

The following action with the given constraints are performed by the primary and secondary ATUs when a target abort is detected by the internal master interface during an inbound read transaction:

Table 15-21. Target Abort During an Inbound Read Transaction

Primary ATU	Secondary ATU
If the data word which was target aborted on the internal bus is actually requested and delivered on the primary PCI Bus, and the Primary ATU ECC Target Abort Enable bit is set in the PATUIMR, a target abort is returned to the PCI initiator on that data word. If the Primary ATU ECC Target Abort Enable bit is clear in the PATUIMR, a disconnect with data is returned to the PCI initiator during the data word that was target aborted on the internal bus.	If the data word which was target aborted on the internal bus is actually requested and delivered on the secondary PCI Bus, and the Secondary ATU ECC Target Abort Enable bit is set in the SATUIMR, a target abort is returned to the PCI initiator on that data word. If the Secondary ATU ECC Target Abort Enable bit is clear in the SATUIMR, a disconnect with data is returned to the PCI initiator during the data word that was target aborted on the internal bus.

The Memory Controller is responsible for creating the **NMI#** interrupt to the core processor. Note target aborts are signalled on a Qword basis. If either Dword of a Qword target aborts, both will be considered to have target aborted.

15.6.7 ATU Error Summary

Table 15-22 through Table 15-25 summarize the ATU error reporting for PCI bus errors (Table 15-22 and Table 15-23) and internal bus errors (Table 15-24 and Table 15-25). The tables assume that all error reporting is enabled through the appropriate command registers (unless otherwise noted). The Primary and Secondary ATU Status Registers record PCI bus errors. Note that the **SERR#** Asserted bit in the Status Register is set only when the **SERR#** Enable bit in the Command Register is set. The Primary and Secondary ATU Interrupt Status Registers record i960 core processor interrupt status information.

Note: When an external agent violates PCI protocol, Primary and Secondary ATU behavior may be unpredictable/undefined.

Table 15-22. Primary ATU Error Reporting Summary - PCI Interface

Error Condition	Bits Set in Primary ATU Status Register (PATUSR)	Bits Set in Primary ATU Interrupt Status Register (PATUISR)	Interrupt Mask Bit in PATUIMR or ATUCR
Inbound Write Address Parity Error	Detected Parity Error - bit 15	Detected Parity Error - bit 9	PATUIMR bit 7
	P_SERR# Asserted - bit 14	**P_SERR#** Asserted - bit 10	PATUIMR bit 6
	N/A	**P_SERR#** Detected - bit 4	ATUCR bit 9
Inbound Write Data Parity Error	Detected Parity Error - bit 15	Detected Parity Error - bit 9	PATUIMR bit 7
Inbound Write Master or Target Abort	**P_SERR#** Asserted - bit 14	**P_SERR#** Asserted - bit 10	PATUIMR bit 6
	N/A	**P_SERR#** Detected - bit 4	ATUCR bit 9
Inbound Read Address Parity Error	Detected Parity Error - bit 15	Detected Parity Error - bit 9	PATUIMR bit 7
	P_SERR# Asserted - bit 14	**P_SERR#** Asserted - bit 10	PATUIMR bit 6
	N/A	**P_SERR#** Detected - bit 4	ATUCR bit 9
Inbound Read Data Parity Error	N/A	N/A	N/A
Inbound Read Target Abort	Target Abort (target) - bit 11	PCI Target Abort (target) - bit 1	PATUIMR bit 3
Outbound Write Master Abort	Master Abort - bit 13	PCI Master Abort - bit 3	PATUIMR bit 5
Outbound Write Data Parity Error	Master Parity Error - bit 8	PCI Master Parity Error - bit 0	PATUIMR bit 2
Outbound Write Target Abort	Target Abort (master) - bit 12	PCI Target Abort (master) - bit 2	PATUIMR bit 4
Outbound Read Master Abort	Master Abort - bit 13	PCI Master Abort - bit 3	PATUIMR bit 5
Outbound Read Data Parity Error	Detected Parity Error - bit 15	Detected Parity Error - bit 9	PATUIMR bit 7
	Master Parity Error - bit 8	PCI Master Parity Error - bit 0	PATUIMR bit 2
Outbound Read Target Abort	Target Abort (master) - bit 12	PCI Target Abort (master) - bit 2	PATUIMR bit 4
P_SERR# Detected	N/A	**P_SERR#** Detected - bit 4	ATUCR bit 9

Table 15-23. Secondary ATU Error Reporting Summary - PCI Interface

Error Condition	Bits Set in Secondary ATU Status Register (SATUSR)	Bits Set in Secondary ATU Interrupt Status Register (SATUISR)	Interrupt Mask Bit in SATUIMR or ATUCR
Inbound Write Address Parity Error	Detected Parity Error - bit 15	Detected Parity Error - bit 9	SATUIMR bit 7
	S_SERR# Asserted - bit 14	S_SERR# Asserted - bit 10	SATUIMR bit 6
	N/A	S_SERR# Detected - bit 4	ATUCR Bit 10
Inbound Write Data Parity Error	Detected Parity Error - bit 15	Detected Parity Error - bit 9	SATUIMR bit 7
Inbound Write Master or Target Abort	S_SERR# Asserted - bit 14	S_SERR# Asserted - bit 10	SATUIMR bit 6
	N/A	S_SERR# Detected - bit 4	ATUCR Bit 10
Inbound Read Address Parity Error	Detected Parity Error - bit 15	Detected Parity Error - bit 9	SATUIMR bit 7
	S_SERR# Asserted - bit 14	S_SERR# Asserted - bit 10	SATUIMR bit 6
	N/A	S_SERR# Detected - bit 4	ATUCR Bit 10
Inbound Read Data Parity Error	N/A	N/A	N/A
Inbound Read Target Abort	Target Abort (target) - bit 11	PCI Target Abort (target) - bit 1	SATUIMR bit 3
Outbound Write Master Abort	Master Abort - bit 13	PCI Master Abort - bit 3	SATUIMR bit 5
Outbound Write Data Parity Error	Master Parity Error - bit 8	PCI Master Parity Error - bit 0	SATUIMR bit 2
Outbound Write Target Abort	Target Abort (master) - bit 12	PCI Target Abort (master) - bit 2	SATUIMR bit 4
Outbound Read Master Abort	Master Abort - bit 13	PCI Master Abort - bit 3	SATUIMR bit 5
Outbound Read Data Parity Error	Detected Parity Error - bit 15	Detected Parity Error - bit 9	SATUIMR bit 7
	Master Parity Error - bit 8	PCI Master Parity Error - bit 0	SATUIMR bit 2
Outbound Read Target Abort	Target Abort (master) - bit 12	PCI Target Abort (master) - bit 2	SATUIMR bit 4
S_SERR# Detected	N/A	S_SERR# Detected - bit 4	ATUCR Bit 10

Table 15-24. Primary ATU Error Reporting Summary - Internal Bus Interface

Error Condition	Bits Set in Primary ATU Status Register (PATUSR)	Bits Set in Primary ATU Interrupt Status Register (PATUISR)	Interrupt Mask Bit in PATUIMR or ATUCR
Inbound Write Master Abort	N/A	Internal Bus Master Abort - bit 7	N/A
	P_SERR# Asserted - bit 14	**P_SERR#** Asserted - bit 10	PATUIMR bit 6
	N/A	**P_SERR#** Detected - bit 4	ATUCR bit 9
Inbound Write Target Abort	**P_SERR#** Asserted - bit 14	**P_SERR#** Asserted - bit 10	PATUIMR bit 6
	N/A	**P_SERR#** Detected - bit 4	ATUCR bit 9
Inbound Read Master Abort	N/A	Internal Bus Master Abort - bit 7	N/A
Inbound Read Target Abort	N/A	N/A	N/A
Outbound Write Master Abort[1]	N/A	N/A	N/A
Outbound Write Target Abort[2]	N/A	N/A	N/A
Outbound Read Master Abort[3]	N/A	N/A	N/A
Outbound Read Target Abort	N/A	N/A	N/A

1. Never occurs since outbound writes are always completely posted.
2. Never occurs since outbound writes are always completely posted.
3. In response to a master abort during the DRC on the primary PCI bus. No errors posted in the PATU, only in the BIU.

Table 15-25. Secondary ATU Error Reporting Summary - Internal Bus Interface

Error Condition	Bits Set in Secondary ATU Status Register (SATUSR)	Bits Set in Secondary ATU Interrupt Status Register (SATUISR)	Interrupt Mask Bit in SATUIMR or ATUCR
Inbound Write Master Abort	N/A	Internal Bus Master Abort - bit 7	N/A
	S_SERR# Asserted - bit 14	S_SERR# Asserted - bit 10	SATUIMR bit 6
	N/A	S_SERR# Detected - bit 4	ATUCR Bit 10
Inbound Write Target Abort	S_SERR# Asserted - bit 14	S_SERR# Asserted - bit 10	SATUIMR bit 6
	N/A	S_SERR# Detected - bit 4	ATUCR Bit 10
Inbound Read Master Abort	N/A	Internal Bus Master Abort - bit 7	N/A
Inbound Read Target Abort	N/A	N/A	N/A
Outbound Write Master Abort[1]	N/A	N/A	N/A
Outbound Write Target Abort[2]	N/A	N/A	N/A
Outbound Read Master Abort[3]	N/A	N/A	N/A
Outbound Read Target Abort	N/A	N/A	N/A

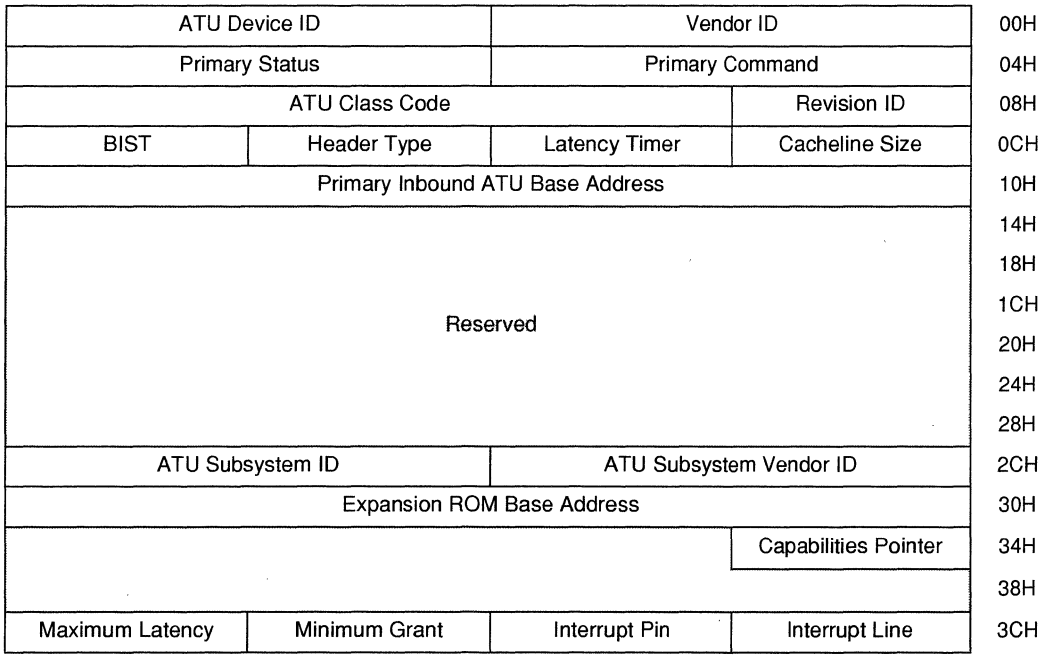
1. Never occurs since outbound writes are always completely posted.
2. Never occurs since outbound writes are always completely posted.
3. In response to a master abort during the DRC on the secondary PCI bus. No errors posted in the SATU, only in the BIU.

15.7 Register Definitions

Every PCI device implements its own separate configuration address space and configuration registers. The *PCI Local Bus Specification*, Revision 2.2 requires that configuration space be 256 bytes, and the first 64 bytes must adhere to a predefined header format.

Figure 15-10 defines the header format. Table 15-13 shows the PCI configuration registers, listed by internal bus address offset. Table 15-14 shows the entire ATU configuration space (including header and extended registers) and the corresponding section that describes each register. Note that all configuration read and write transactions will be accepted on the internal bus as 32-bit transactions. Refer to Appendix C, "Peripheral Memory-Mapped Registers".

Figure 15-10. ATU Interface Configuration Header Format



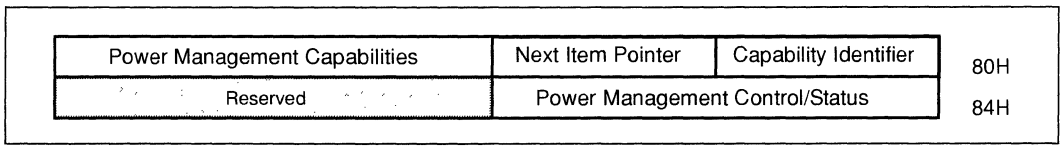
Primary and secondary ATUs are programmed via a Type 0 configuration command on the primary interface. See Section 15.2.1.4, "Inbound Configuration Cycle Translation" on page 15-13. ATU configuration space is function number one of the 80303 I/O processor multi-function PCI device. (Refer to Section 15.2.4, "PCI Multi-Function Device Swapping/Disabling" on page 15-23 for exceptions to this statement.)

Beyond the required 64 byte header format, ATU configuration space implements extended register space in support of the units functionality. Refer to the *PCI Local Bus Specification*, Revision 2.2 for details on accessing and programming configuration register space.

The ATU unit includes an 8 byte extended capability configuration space beginning at configuration offset 80H. The extended configuration space can be accessed by a device on the primary interface through a mechanism defined in the *PCI Local Bus Specification*, Revision 2.2.

In the Primary ATU Status Register (Section 15.7.4) the appropriate bit is set indicating that the Extended Capability Configuration space is supported. When this bit is read, the device can then read the Capabilities Pointer register (Section 15.7.15) to determine the configuration offset of the Extended Capabilities Configuration Header. The format of this header is depicted in Figure 15-11.

Figure 15-11. ATU Interface Extended Configuration Header Format



The first byte at the Extended Configuration Offset is the ATU Capability Identifier Register (Section 15.7.34). This will identify this Extended Configuration Header space as the type defined by the *PCI Bus Power Management Interface Specification,* Revision 1.1.

Following the Capability Identifier Register will be the single byte Next Item Pointer Register (Section 15.7.35) which will indicate the configuration offset of an additional Extended Capabilities Header, if supported. In the ATU, the Next Item Pointer Register is set to 00H indicating that there are no additional Extended Capabilities Headers supported in the ATUs configuration space.

To enable the *PCI Bus Power Management Interface Specification,* Revision 1.1 compliance support, the Power State Transition interrupt mask in bit 8 of the PATUIMR needs to be cleared and the PCI extended Capabilities enable in bit 4 of the PATUSR needs to be set. It is the configuration software responsibility to properly enable and initialize the ATUs Power Management Interface before the Configuration Cycle Retry Bit in the Chapter 14, "Extended Bridge Control Register - EBCR" is cleared in order for the ATU to be *Advanced Configuration and Power Interface Specification,* Revision 1.0 compliant.

The following sections describe the ATU and Expansion ROM configuration registers. Configuration space consists of 8, 16, 24, and 32-bit registers arranged in a predefined format. Each register is described in functionality, access type (read/write, read/clear, read only) and reset default condition.

See Section 1.3, "Terminology and Conventions" on page 1-5 for a description of *reserved, read only,* and *read/clear.* All registers adhere to the definitions found in the *PCI Local Bus Specification,* Revision 2.2 unless otherwise noted.

The PCI register number for each register is given in Table 15-13. As stated, a Type 0 configuration command on the primary bus with an active P_IDSEL or a memory-mapped internal bus access is required to read or write these registers.

Note: Each configuration register's access type is individually defined for PCI configuration accesses. Some PCI read-only configuration registers have read/write capability from the 80303 I/O processor core CPU. See also Appendix C, "Peripheral Memory-Mapped Registers".

Table 15-26. Address Translation Unit Registers (Sheet 1 of 2)

Register Name	Bits	PCI Configuration Cycle Register Number	Internal Bus Address
ATU Vendor ID Register - ATUVID	16	0	0000.1200H
Device ID Register - DID (80303 I/O processor)	16	0	0000.1202H
Primary ATU Command Register - PATUCMD	16	1	0000.1204H
Primary ATU Status Register - PATUSR	16	1	0000.1206H
ATU Revision ID Register - ATURID	8	2	0000.1208H
ATU Class Code Register - ATUCCR	24	2	0000.1209H
ATU Cacheline Size Register - ATUCLSR	8	3	0000.120CH
ATU Latency Timer Register - ATULT	8	3	0000.120DH
ATU Header Type Register - ATUHTR	8	3	0000.120EH
ATU BIST Register - ATUBISTR	8	3	0000.120FH
Primary Inbound ATU Base Address Register - PIABAR	32	4	0000.1210H
Reserved	32	5	0000.1214H
Reserved	32	6	0000.1218H
Reserved	32	7	0000.121CH
Reserved	32	8	0000.1220H
Reserved	32	9	0000.1224H
Reserved	32	10	0000.1228H
ATU Subsystem Vendor ID Register - ASVIR	16	11	0000.122CH
ATU Subsystem ID Register - ASIR	16	11	0000.122EH
Expansion ROM Base Address Register -ERBAR	32	12	0000.1230H
ATU Capabilities Pointer Register - ATU_Cap_Ptr	8	13	0000.1234H
Reserved	8	13	0000.1235H
Reserved	16	13	0000.1236H
Reserved	32	14	0000.1238H
ATU Interrupt Line Register - ATUILR	8	15	0000.123CH
ATU Interrupt Pin Register - ATUIPR	8	15	0000.123DH
ATU Minimum Grant Register - ATUMGNT	8	15	0000.123EH
ATU Maximum Latency Register - ATUMLAT	8	15	0000.123FH
Primary Inbound ATU Limit Register - PIALR	32	16	0000.1240H
Primary Inbound ATU Translate Value Register - PIATVR	32	17	0000.1244H
Secondary Inbound ATU Base Address Register - SIABAR	32	18	0000.1248H
Secondary Inbound ATU Limit Register - SIALR	32	19	0000.124CH
Secondary Inbound Translate ATU Value Register - SIATVR	32	20	0000.1250H
Primary Outbound Memory Window Value Register - POMWVR	32	21	0000.1254H
Reserved	32	22	0000.1258H

Table 15-26. Address Translation Unit Registers (Sheet 2 of 2)

Register Name	Bits	PCI Configuration Cycle Register Number	Internal Bus Address
Primary Outbound I/O Window Value Register - POIOWVR	32	23	0000.125CH
Primary Outbound DAC Window Value Register - PODWVR	32	24	0000.1260H
Primary Outbound Upper 64-bit DAC Register - POUDR	32	25	0000.1264H
Secondary Outbound Memory Window Value Register - SOMWVR	32	26	0000.1268H
Secondary Outbound I/O Window Value Register - SOIOWVR	32	27	0000.126CH
Reserved	32	28	0000.1270H
Expansion ROM Limit Register - ERLR	32	29	0000.1274H
Expansion ROM Translate Value Register - ERTVR	32	30	0000.1278H
Reserved	32	31	0000.127CH
ATU_Capability Identifier Register - ATU_Cap_ID	8	32	0000.1280H
ATU Next Item Pointer Register - ATU_Next_Item_Ptr	8	32	0000.1281H
ATU Power Management Capabilities Register - APMCR	16	32	0000.1282H
ATU Power Management Control/Status Register - APMCSR	16	33	0000.1284H
Reserved	16	33	0000.1286H
ATU Configuration Register - ATUCR	32	34	0000.1288H
Reserved	32	35	0000.128CH
Primary ATU Interrupt Status Register - PATUISR	32	36	0000.1290H
Secondary ATU Interrupt Status Register - SATUISR	32	37	0000.1294H
Secondary ATU Command Register - SATUCMD	16	38	0000.1298H
Secondary ATU Status Register - SATUSR	16	38	0000.129AH
Secondary Outbound DAC Window Value Register - SODWVR	32	39	0000.129CH
Secondary Outbound Upper 64-bit DAC Register - SOUDR	32	40	0000.12A0H
Primary Outbound Configuration Cycle Address Register - POCCAR	32	41	0000.12A4H
Secondary Outbound Configuration Cycle Address Register - SOCCAR	32	42	0000.12A8H
Primary Outbound Configuration Cycle Data Register - POCCDR	32	Not Available in PCI Configuration Space	0000.12ACH
Secondary Outbound Configuration Cycle Data Register - SOCCDR	32	Not Available in PCI Configuration Space	0000.12B0H
Primary ATU Queue Control Register - PAQCR	32	45	0000.12B4H
Secondary ATU Queue Control Register - SAQCR	32	46	0000.12B8H
Primary ATU Interrupt Mask Register - PATUIMR	32	47	0000.12BCH
Secondary ATU Interrupt Mask Register - SATUIMR	32	48	0000.12C0H

Table 15-27. ATU PCI Configuration Register Space (Sheet 1 of 2)

Internal Bus Address Offset	ATU PCI Configuration Register Section, Name, Page
00H	Section 15.7.1, "ATU Vendor ID Register - ATUVID" on page 15-60
02H	Section 15.7.2, "ATU Device ID Register - ATUDID" on page 15-61
04H	Section 15.7.3, "Primary ATU Command Register - PATUCMD" on page 15-62
06H	Section 15.7.4, "Primary ATU Status Register - PATUSR" on page 15-63
08H	Section 15.7.5, "ATU Revision ID Register - ATURID" on page 15-65
09H	Section 15.7.6, "ATU Class Code Register - ATUCCR" on page 15-66
0CH	Section 15.7.7, "ATU Cacheline Size Register - ATUCLSR" on page 15-67
0DH	Section 15.7.8, "ATU Latency Timer Register - ATULT" on page 15-68
0EH	Section 15.7.9, "ATU Header Type Register - ATUHTR" on page 15-69
0FH	Section 15.7.10, "ATU BIST Register - ATUBISTR" on page 15-70
10H	Section 15.7.11, "Primary Inbound ATU Base Address Register - PIABAR" on page 15-71
2CH	Section 15.7.12, "ATU Subsystem Vendor ID Register - ASVIR" on page 15-72
2EH	Section 15.7.13, "ATU Subsystem ID Register - ASIR" on page 15-73
30H	Section 15.7.14, "Expansion ROM Base Address Register - ERBAR" on page 15-74
34H	Section 15.7.15, "ATU Capabilities Pointer Register - ATU_Cap_Ptr" on page 15-75
3CH	Section 15.7.17, "ATU Interrupt Line Register - ATUILR" on page 15-77
3DH	Section 15.7.18, "ATU Interrupt Pin Register - ATUIPR" on page 15-78
3EH	Section 15.7.19, "ATU Minimum Grant Register - ATUMGNT" on page 15-79
3FH	Section 15.7.20, "ATU Maximum Latency Register - ATUMLAT" on page 15-80
40H	Section 15.7.21, "Primary Inbound ATU Limit Register - PIALR" on page 15-81
44H	Section 15.7.22, "Primary Inbound ATU Translate Value Register - PIATVR" on page 15-82
48H	Section 15.7.23, "Secondary Inbound ATU Base Address Register - SIABAR" on page 15-83
4CH	Section 15.7.24, "Secondary Inbound ATU Limit Register - SIALR" on page 15-84
50H	Section 15.7.25, "Secondary Inbound ATU Translate Value Register - SIATVR" on page 15-85
54H	Section 15.7.26, "Primary Outbound Memory Window Value Register - POMWVR" on page 15-86
5CH	Section 15.7.27, "Primary Outbound I/O Window Value Register - POIOWVR" on page 15-87
60H	Section 15.7.28, "Primary Outbound DAC Window Value Register - PODWVR" on page 15-88
64H	Section 15.7.29, "Primary Outbound Upper 64-bit DAC Register - POUDR" on page 15-89
68H	Section 15.7.30, "Secondary Outbound Memory Window Value Register - SOMWVR" on page 15-90
6CH	Section 15.7.31, "Secondary Outbound I/O Window Value Register - SOIOWVR" on page 15-91
74H	Section 15.7.32, "Expansion ROM Limit Register - ERLR" on page 15-92
78H	Section 15.7.33, "Expansion ROM Translate Value Register - ERTVR" on page 15-93
80H	Section 15.7.34, "ATU_Capability Identifier Register - ATU_Cap_ID" on page 15-94
81H	Section 15.7.35, "ATU Next Item Pointer Register - ATU_Next_Item_Ptr" on page 15-95
82H	Section 15.7.36, "ATU Power Management Capabilities Register - APMCR" on page 15-96
84H	Section 15.7.37, "ATU Power Management Control/Status Register - APMCSR" on page 15-97

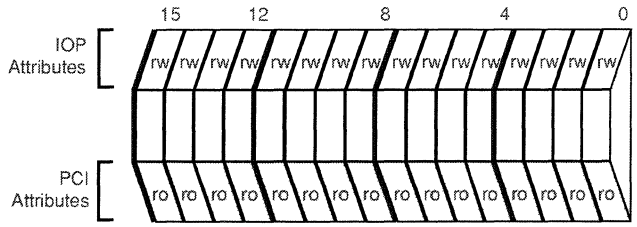
Table 15-27. ATU PCI Configuration Register Space (Sheet 2 of 2)

Internal Bus Address Offset	ATU PCI Configuration Register Section, Name, Page
88H	Section 15.7.38, "ATU Configuration Register - ATUCR" on page 15-98
90H	Section 15.7.39, "Primary ATU Interrupt Status Register - PATUISR" on page 15-100
94H	Section 15.7.40, "Secondary ATU Interrupt Status Register - SATUISR" on page 15-102
98H	Section 15.7.41, "Secondary ATU Command Register - SATUCMD" on page 15-104
9AH	Section 15.7.42, "Secondary ATU Status Register - SATUSR" on page 15-105
9CH	Section 15.7.43, "Secondary Outbound DAC Window Value Register - SODWVR" on page 15-106
A0H	Section 15.7.44, "Secondary Outbound Upper 64-bit DAC Register - SOUDR" on page 15-107
A4H	Section 15.7.45, "Primary Outbound Configuration Cycle Address Register - POCCAR" on page 15-108
A8H	Section 15.7.46, "Secondary Outbound Configuration Cycle Address Register - SOCCAR" on page 15-109
ACH	Section 15.7.47, "Primary Outbound Configuration Cycle Data Register - POCCDR" on page 15-110
B0H	Section 15.7.48, "Secondary Outbound Configuration Cycle Data Register - SOCCDR" on page 15-111
B4H	Section 15.7.49, "Primary ATU Queue Control Register - PAQCR" on page 15-112
B8H	Section 15.7.50, "Secondary ATU Queue Control Register - SAQCR" on page 15-113
BCH	Section 15.7.51, "Primary ATU Interrupt Mask Register - PATUIMR" on page 15-114
C0H	Section 15.7.52, "Secondary ATU Interrupt Mask Register - SATUIMR" on page 15-115

15.7.1 ATU Vendor ID Register - ATUVID

ATU Vendor ID Register bits adhere to the definitions in the *PCI Local Bus Specification*, Revision 2.2.

Table 15-28. ATU Vendor ID Register - ATUVID



Intel® i960® Core Local Bus Address
1200H

PCI Configuration Address Offset
00H - 01H

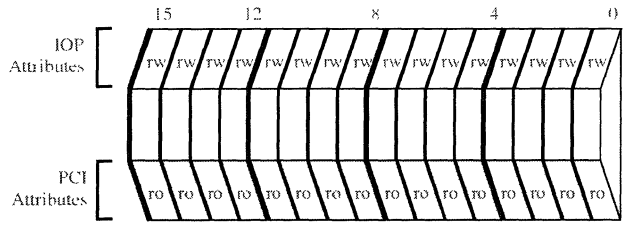
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
15:00	0530C	ATU Vendor ID - This is a 16-bit value assigned to Intel. This register, combined with the DID, uniquely identify the PCI device. Access type is Read/Write to allow the 80303 I/O processor to configure the register as a different vendor ID to simulate the interface of a standard mechanism currently used by existing application software.

15.7.2 ATU Device ID Register - ATUDID

ATU Device ID Register bits adhere to the definitions in the *PCI Local Bus Specification*, Revision 2.2.

Table 15-29. Device ID Register - DID (80303 I/O processor)



Intel® i960® Core Local Bus Address
1202H

PCI Configuration Address Offset
02H - 03H

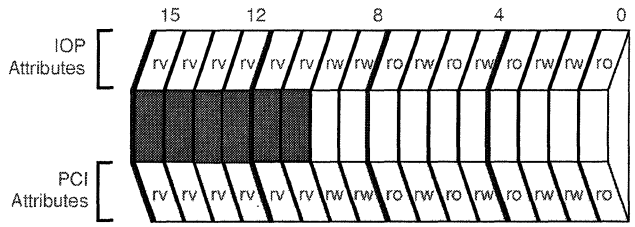
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
15:00	5309H	ATU Device ID - This is a 16-bit value assigned to the ATU. This ID, combined with the VID, uniquely identify any PCI device.

15.7.3 Primary ATU Command Register - PATUCMD

ATU Command Register bits adhere to the definitions in the *PCI Local Bus Specification,* Revision 2.2 and in most cases, affect the behavior of the primary PCI ATU and devices on the primary PCI bus.

Table 15-30. Primary ATU Command Register - PATUCMD



Intel® i960® Core Local Bus Address
1204H

PCI Configuration Address Offset
04H - 05H

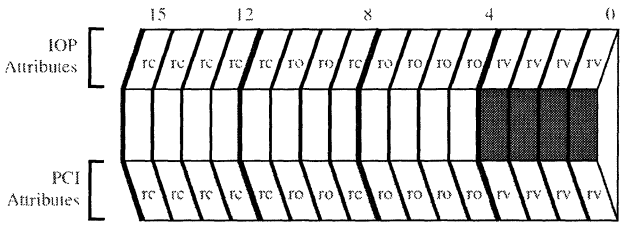
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
15:10	000000_2	Reserved
09	0_2	Fast Back to Back Enable - When cleared, the ATU primary interface is not allowed to generate fast back-to-back cycles on its bus.
08	0_2	**P_SERR#** Enable - When cleared, the ATU primary interface is not allowed to assert **P_SERR#** on the primary PCI interface.
07	0_2	Wait Cycle Control - controls address/data stepping. Not implemented and a reserved bit field.
06	0_2	Parity Error Response - When set, the primary ATU and DMA channels 0 and 1 take normal action when a parity error is detected. When cleared, parity checking is disabled.
05	0_2	VGA Palette Snoop Enable - The primary ATU interface does not support I/O writes and therefore, does not perform VGA palette snooping.
04	0_2	Memory Write and Invalidate Enable - When set, DMA channels 0 and 1 may generate MWI commands. When clear, DMA channels 0 and 1 use Memory Write commands instead of MWI.
03	0_2	Special Cycle Enable - The ATU interface does not respond to special cycle commands in any way. Not implemented and a reserved bit field.
02	0_2	Bus Master Enable - The primary ATU interface can act as a master on the PCI bus. When cleared, disables the device from generating PCI accesses. When set, allows the device to behave as a PCI bus master. This enable bit also controls DMA channels 0 and 1 master interface. The bit must be set before initiating a DMA transfer on the PCI bus.
01	0_2	Memory Enable - Controls the primary ATU interface's response to PCI memory addresses. When cleared, the ATU interface does not respond to any memory access on the PCI bus.
00	0_2	I/O Space Enable - Controls the ATU interface response to I/O transactions on the primary side. Not implemented and a reserved bit field.

15.7.4 Primary ATU Status Register - PATUSR

The Primary ATU Status Register bits adhere to the *PCI Local Bus Specification*, Revision 2.2 definitions. The *read/clear* bits can only be set by internal hardware and cleared by either a reset condition or by writing a 1_2 to the register.

Table 15-31. Primary ATU Status Register - PATUSR (Sheet 1 of 2)



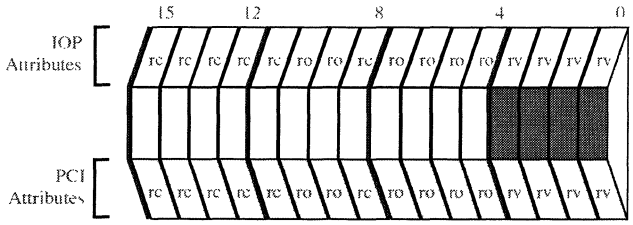
Intel® i960® Core Local Bus Address
1206H

PCI Configuration Address Offset
06H - 07H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
15	0_2	Detected Parity Error - set when a parity error is detected on the primary PCI bus even when the PATUCMD register's Parity Error Response bit is cleared. Set under the following conditions: • Write Data Parity Error when the PATU is a slave (inbound write). • Read Data Parity Error when the PATU, DMA Channel 0, or DMA Channel1 is a master (outbound read). • Any Address Parity Error on the Primary Bus (including one generated by the PATU or DMA Channels 0 &1).
14	0_2	**P_SERR#** Asserted - set when **P_SERR#** is asserted on the PCI bus by the primary ATU.
13	0_2	Master Abort - set when a transaction initiated by the primary ATU PCI master interface, DMA Channel 0, or DMA Channel 1 ends in a Master-Abort. Setting of this bit due to an error condition from either DMA Channel will not cause an ATU interrupt to the core.
12	0_2	Target Abort (master) - set when a transaction initiated by the primary ATU PCI master interface, DMA Channel 0 master interface or DMA Channel 1 master interface ends in a target abort. Setting of this bit due to an error condition from either DMA Channel will not cause an ATU interrupt to the core.
11	0_2	Target Abort (target) - set when the primary ATU interface, acting as a target, terminates the transaction on the primary PCI bus with a target abort.
10:09	10_2	DEVSEL# Timing - These bits are read-only and define the slowest DEVSEL# timing for a target device (except configuration accesses). 00_2 = Fast 01_2 = Medium 10_2 = Slow 11_2 = Reserved In general, the primary and secondary ATU interfaces use Medium timing. However, if the Messaging Unit is enabled and the Inbound/Outbound Message Ports (Primary ATU Inbound Window offset 40h) are hit with a 64-bit access (**P_REQ64#** asserted), the need to pre-decode the **P_C/BE[3:0]#** bus forces the PATU to claim the access using Slow decode timing. Though this occurrence is rather unlikely, the *PCI Local Bus Specification,* Revision 2.2 requires that this field of the status register indicate the slowest DEVSEL# possible by a target device. In the event that a subtractive decode agent is present on the PATU bus segment, the indication that the PATU could claim positively with Slow decode will prevent the users of this bus from programming the subtractive decode agent to claim with Slow decode timing.

Table 15-31. Primary ATU Status Register - PATUSR (Sheet 2 of 2)



Intel® i960® Core Local Bus Address
1206H

PCI Configuration Address Offset
06H - 07H

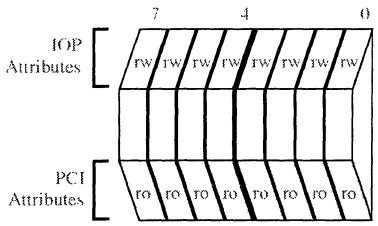
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
08	0_2	Master Parity Error - The primary ATU interface sets this bit under the following conditions: • The PATU, DMA Channel 0, or DMA Channel 1 asserted S_PERR# itself or the PATU observed S_PERR# asserted. • And the PATU, DMA Channel 0, or DMA Channel 1 acted as the bus master for the operation in which the error occurred. • And the PATUCMD register's Parity Error Response bit is set Setting of this bit due to an error condition from either DMA Channel will not cause an ATU interrupt to the core.
07	1_2	Fast Back-to-Back - The ATU/Messaging Unit interface is capable of accepting fast back-to-back transactions when the transactions are not to the same target.
06	0_2	UDF Supported - User Definable Features are not supported
05	1_2	66 MHz. Capable - 66 MHz operation is supported.
04	1_2	**Capabilities** - When set, this function implements extended capabilities.
03:00	00000_2	Reserved

15.7.5 ATU Revision ID Register - ATURID

Revision ID Register bit definitions adhere to *PCI Local Bus Specification*, Revision 2.2.

Table 15-32. ATU Revision ID Register - ATURID



Intel® i960® Core Local Bus Address
1208H

PCI Configuration Address Offset
08H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set

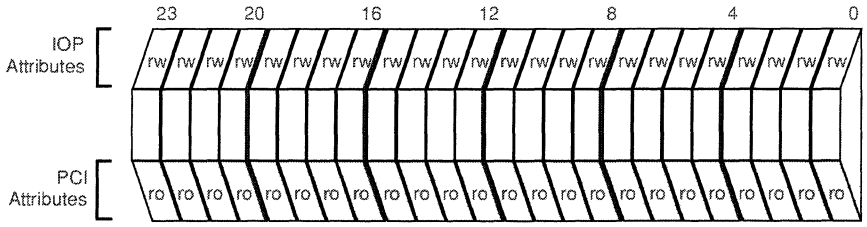
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
07:00	00H	ATU Revision - identifies the 80303 I/O processor's revision number.

15.7.6 ATU Class Code Register - ATUCCR

Class Code Register bit definitions adhere to *PCI Local Bus Specification*, Revision 2.2. Auto configuration software reads this register to determine the PCI device function.

Table 15-33. ATU Class Code Register - ATUCCR



Intel® i960® Core Local Bus Address
1209H

PCI Configuration Address Offset
09H - 0BH

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set

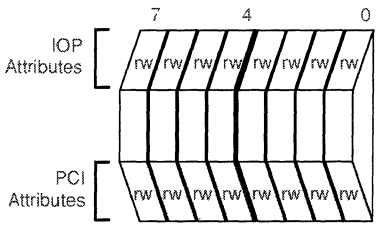
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
23:16	05H	Base Class - Memory Controller
15:08	80H	Sub Class - Other Memory Controller
07:00	00H	Programming Interface - None defined

15.7.7 ATU Cacheline Size Register - ATUCLSR

Cacheline Size Register bit definitions adhere to *PCI Local Bus Specification*, Revision 2.2. This register is programmed with the system cacheline size in DWORDs (32-bit words). Cacheline Size is restricted to either 0, 8 or 16 DWORDs; the ATU interprets any other value as "0".

Table 15-34. ATU Cacheline Size Register - ATUCLSR



Intel® i960® Core Local Bus Address
120CH

PCI Configuration Address Offset
0CH

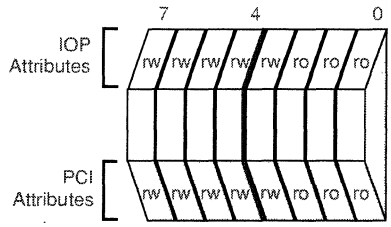
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear

Bit	Default	Description
07:00	00H	ATU Cacheline Size - specifies the system cacheline size in DWORDs. Cacheline size is restricted to either 0, 8 or 16 DWORDs.

15.7.8 ATU Latency Timer Register - ATULT

ATU Latency Timer Register bit definitions apply to both the primary and secondary PCI interfaces.

Table 15-35. ATU Latency Timer Register - ATULT



Intel® i960® Core Local Bus Address
120DH

PCI Configuration Address Offset
0DH

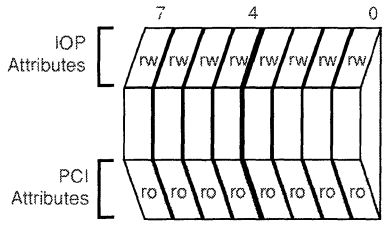
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
07:03	00000_2	Programmable Latency Timer - This field varies the latency timer for the primary interface from 0 to 248 clocks.
02:00	000_2	Latency Timer Granularity - These Bits are read only giving a programmable granularity of 8 clocks for the latency timer.

15.7.9 ATU Header Type Register - ATUHTR

Header Type Register bit definitions adhere to *PCI Local Bus Specification*, Revision 2.2. This register indicates the layout of ATU and Messaging Unit register configuration space bytes 10H to 3FH. The MSB indicates whether or not the device is multi-function. (Refer to Section 15.2.4, "PCI Multi-Function Device Swapping/Disabling" on page 15-23 for using this register in other than its default state.)

Table 15-36. ATU Header Type Register - ATUHTR



Intel® i960® Core Local Bus Address
120EH

PCI Configuration Address Offset
0EH

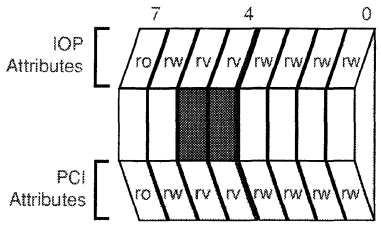
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
07	1_2	Single Function/Multi-Function Device - Identifies the ATU as a multi-function PCI device.
06:00	000000_2	PCI Header Type - This bit field indicates the type of PCI header implemented. The ATU interface header conforms to *PCI Local Bus Specification*, Revision 2.2.

15.7.10 ATU BIST Register - ATUBISTR

The ATU BIST Register controls the functions the i960 core processor performs when BIST is initiated. This register is the interface between the host processor requesting BIST functions and the 80303 I/O processor replying with the results from the software implementation of the BIST functionality.

Table 15-37. ATU BIST Register - ATUBISTR



Intel® i960® Core Local Bus Address
120FH

PCI Configuration Address Offset
0FH

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
07	0_2	BIST Capable - This bit value is always equal to the ATUCR ATU BIST Interrupt Enable bit. See Section 15.7.38, "ATU Configuration Register - ATUCR" on page 15-98.
06	0_2	Start BIST - When the ATUCR BIST Interrupt Enable bit is set: Setting this bit generates an interrupt to the i960 core processor to perform a software BIST function. The i960 core processor clears this bit when the BIST software has completed with the BIST results found in ATUBISTR register bits [3:0]. When the ATUCR BIST Interrupt Enable bit is clear: Setting this bit does not generate an interrupt to the i960 core processor and no BIST functions is performed. The i960 core processor does not clear this bit.
05:04	00_2	Reserved
03:00	0000_2	BIST Completion Code - when the ATUCR BIST Interrupt Enable bit is set and the ATUBISTR Start BIST bit is set (bit 6): The i960 core processor places the results of the software BIST in these bits. A nonzero value indicates a device-specific error.

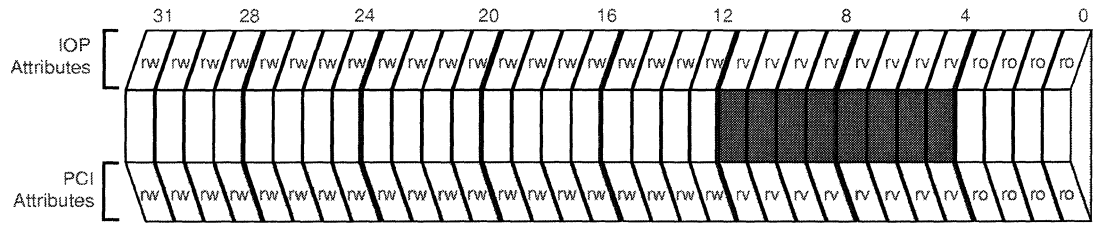
15.7.11 Primary Inbound ATU Base Address Register - PIABAR

The Primary Inbound ATU Base Address Register (PIABAR) defines the block of memory addresses where the primary inbound translation window begins. The inbound ATU decodes and forwards the bus request to the 80303 I/O processor internal bus with a translated address to map into 80303 I/O processor local memory. The PIABAR defines the base address and describes the required memory block size; see Section 15.7.16, "Determining Block Sizes for Base Address Registers" on page 15-76. Bits 31 through 12 of the PIABAR will be either read/write bits or read only with a value of 0 depending on the value located within the PIALR. This configuration allows the PIABAR to be programmed per *PCI Local Bus Specification,* Revision 2.2.

The first 4 Kbytes of memory defined by the PIABAR and the PIALR is reserved for the Messaging Unit.

The programmed value within the base address register must comply with the PCI programming requirements for address alignment. Refer to the *PCI Local Bus Specification,* Revision 2.2 for additional information on programming base address registers.

Table 15-38. Primary Inbound ATU Base Address Register - PIABAR



Intel® i960® Core Local Bus Address
1210H

PCI Configuration Address Offset
10H - 13H

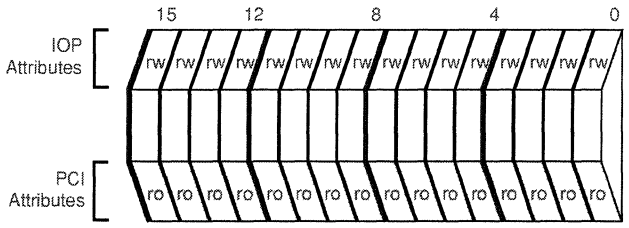
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:12	00000H	Primary Translation Base Address - These bits define the actual location the Primary translation function is to respond to when addressed from the PCI bus. The default base address is undefined.
11:04	00H	Reserved.
03	1_2	Prefetchable Indicator - Defines the memory spaces as prefetchable.
02:01	00_2	Address Type - These bits define where the block of memory can be located. The base address must be located anywhere in the first 4 Gbyte of address space (lower 32 bits of address).
00	0_2	Memory Space Indicator - This bit field describes memory or I/O space base address. The primary ATU does not occupy I/O space, thus this bit must be zero.

15.7.12 ATU Subsystem Vendor ID Register - ASVIR

ATU Subsystem Vendor ID Register bit definitions adhere to *PCI Local Bus Specification,* Revision 2.2.

Table 15-39. ATU Subsystem Vendor ID Register - ASVIR



Intel® i960® Core Local Bus Address
122CH

PCI Configuration Address Offset
2CH - 2DH

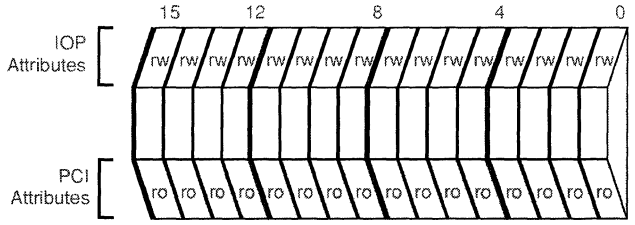
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
15:0	0000H	Subsystem Vendor ID - This register uniquely identifies the add-in board or subsystem vendor.

15.7.13 ATU Subsystem ID Register - ASIR

ATU Subsystem ID Register bit definitions adhere to *PCI Local Bus Specification*, Revision 2.2.

Table 15-40. ATU Subsystem ID Register - ASIR



Intel® i960® Core Local Bus Address
122EH

PCI Configuration Address Offset
2EH - 2FH

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

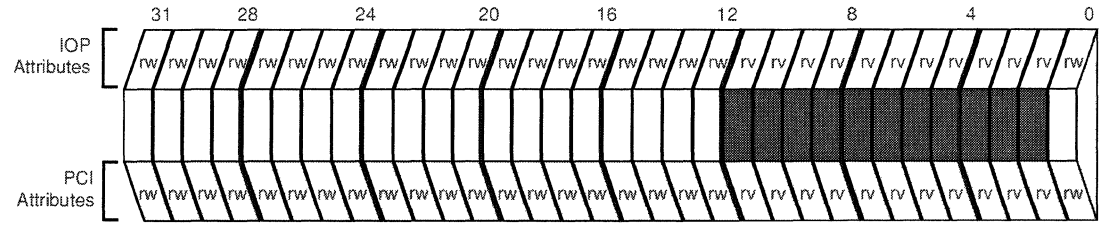
Bit	Default	Description
15:0	0000H	Subsystem ID - uniquely identifies the add-in board or subsystem.

15.7.14 Expansion ROM Base Address Register - ERBAR

The Expansion ROM Base Address Register defines the block of memory addresses used for containing the Expansion ROM. It permits the inclusion of multiple code images, allowing the device to be initialized. The code image supplied consists of either executable code or an interpreted code. Each code image must start on a 512 byte boundary and each must contain the PCI Expansion ROM header. Image placement in ROM space depends on the length of code images which precede it within ROM. ERBAR defines the base address and describes the required memory block size; see Section 15.7.16. Expansion ROM address space (limit size) can be a maximum of 16 MBytes. Bits 31 through 12 of the ERBAR will be either read/write bits or read only with a value of 0 depending on the value located within the ERLR. This configuration allows the ERBAR to be programmed per *PCI Local Bus Specification,* Revision 2.2.

The Expansion ROM Base Address Register's programmed value must comply with the PCI programming requirements for address alignment. Refer to the *PCI Local Bus Specification,* Revision 2.2 for additional information on programming Expansion ROM base address registers.

Table 15-41. Expansion ROM Base Address Register -ERBAR



Intel® i960® Core Local Bus Address
1230H

PCI Configuration Address Offset
30H - 33H

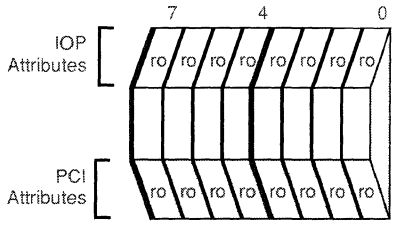
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:12	00000H	Expansion ROM Base Address - These bits define the actual location where the Expansion ROM address window resides when addressed from the primary PCI bus on any 4 Kbyte boundary.
11:01	000H	Reserved
00	0_2	Address Decode Enable - This bit field shows the ROM address decoder is enabled or disabled. When cleared, indicates the address decoder is disabled.

15.7.15 ATU Capabilities Pointer Register - ATU_Cap_Ptr

The Capabilities Pointer Register bits adhere to the definitions in the *PCI Local Bus Specification*, Revision 2.2. This register provides an offset in this function's PCI Configuration Space for the location of the first item in the Capabilities linked list. In the case of the 80303 I/O processor, this is the PCI Bus Power Management extended capability as defined by the *PCI Bus Power Management Interface Specification*, Revision 1.1.

Table 15-42. ATU Capabilities Pointer Register - ATU_Cap_Ptr



Intel® i960® Core Local Bus Address
0000 1234H

PCI Configuration Address Offset
34H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

36 Bit	Default	Description
07:00	80H	Capability List Pointer - This provides an offset in this function's configuration space that points to the 80303 I/O processor's PCI Bus Power Management extended capability.

15.7.16 Determining Block Sizes for Base Address Registers

The required address size and type can be determined by writing ones to a base address register and reading from the registers. By scanning the returned value from the least-significant bit of the base address registers upwards, the programmer can determine the required address space size. The binary-weighted value of the first non-zero bit found indicates the required amount of space. Section 15-43 describes the relationship between the values read back and the byte sizes the base address register requires.

Table 15-43. Memory Block Size Read Response Table

Response After Writing all 1s to the Base Address Register	Size (in Bytes)	Response After Writing all 1s to the Base Address Register	Size (in Bytes)
FFFFFFF0H	16	FFF00000H	1 M
FFFFFFE0H	32	FFE00000H	2 M
FFFFFFC0H	64	FFC00000H	4 M
FFFFFF80H	128	FF800000H	8 M
FFFFFF00H	256	FF000000H	16 M
FFFFFE00H	512	FE000000H	32 M
FFFFFC00H	1K	FC000000H	64 M
FFFFF800H	2K	F8000000H	128 M
FFFFF000H	4K	F0000000H	256 M
FFFFE000H	8K	E0000000H	512 M
FFFFC000H	16K	C0000000H	1 G
FFFF8000H	32K	80000000H	2 G
FFFF0000H	64K	00000000H	Register not implemented, no address space required.
FFFE0000H	128K		
FFFC0000H	256K		
FFF80000H	512K		

As an example, assume that FFFF.FFFFH is written to the ATU Primary Inbound Base Address Register (PIABAR) and the value read back is FFF0.0004H. Bit zero is a zero, so the device requires memory address space. Bits 2:1 are 00_2, so the memory can be located anywhere within 32-bit address space (4 Gbytes). Bit three is one, so the memory does supports prefetching. Scanning upwards starting at bit four, bit twenty is the first one bit found. The binary-weighted value of this bit is 1,048,576, indicated that the device requires 1 Mbyte of memory space.

Both the Primary and Secondary Inbound ATU Base Address Registers and the Expansion ROM Base Address Register use their associated limit registers to enable which bits within the base address register are read/write and which bits are read only (0). This allows the programming of these registers in a manner similar to other PCI devices even though the limit is variable.

Table 15-44. ATU Base Registers and Associated Limit Registers

Base Address Register	Limit Register	Description
Primary Inbound ATU Base Address Register	Primary Inbound ATU Limit Register	Defines the inbound translation window from the primary PCI bus.
Secondary Inbound ATU Base Address Register	Secondary Inbound ATU Limit Register	Defines the inbound translation window from the secondary PCI bus.
Expansion ROM Base Address Register	Expansion ROM Limit Register	Defines the window of addresses used by a primary bus master for reading from an Expansion ROM.

15.7.17 ATU Interrupt Line Register - ATUILR

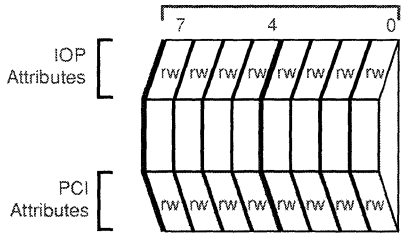
ATU Interrupt Line Register bit definitions adhere to *PCI Local Bus Specification*, Revision 2.2. This register identifies the system interrupt controller's interrupt request lines which connect to the device's PCI interrupt request lines (as specified in the interrupt pin register).

In a PC environment, for example, the register values and corresponding connections are:

- 0 (00H) through 15 (0FH) correspond to IRQ0 through IRQ15
- 16 (10H) through 254 (FEH) are reserved
- 255 (FFH) indicates "unknown" or "no connection"

The operating system or device driver can examine each device's interrupt pin and interrupt line register to determine which system interrupt request line the device uses to issue requests for service.

Table 15-45. ATU Interrupt Line Register - ATUILR



Intel® i960® Core Local Bus Address: 123CH

PCI Configuration Address Offset: 3CH

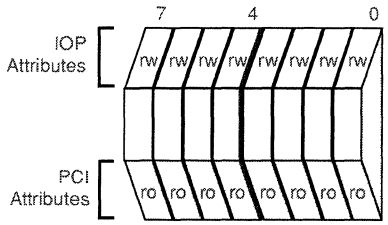
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
07:00	FFH	Interrupt Assigned - system-assigned value identifies which system interrupt controller's interrupt request line connects to the device's PCI interrupt request lines (as specified in the interrupt pin register). A value of FFH signifies "no connection" or "unknown".

15.7.18 ATU Interrupt Pin Register - ATUIPR

ATU Interrupt Pin Register bit definitions adhere to *PCI Local Bus Specification,* Revision 2.2. This register identifies the interrupt pin the ATU and Messaging Unit interface uses. The 80303 I/O processor is, by default, a PCI multi-function device and, as such, can generate more than one interrupt output. The interrupt output is for the Messaging Unit on **P_INTA#**, **P_INTB#**, **P_INTC#**, or **P_INTD#**. The i960 core processor modifies the pin register to match the PCI interrupts which the Messaging Unit generates.

Table 15-46. ATU Interrupt Pin Register - ATUIPR



Intel® i960® Core Local Bus Address: 123DH

PCI Configuration Address Offset: 3DH

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

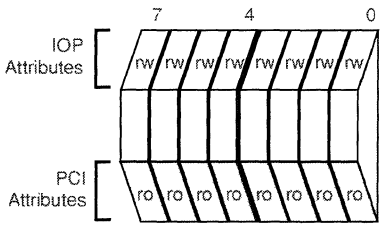
Bit	Default	Description
07:00	01H	Interrupt Used - A value of 01H signifies that the ATU interface unit uses **INTA#** as the interrupt pin.

15.7.19 ATU Minimum Grant Register - ATUMGNT

ATU Minimum Grant Register bit definitions adhere to *PCI Local Bus Specification*, Revision 2.2. This register specifies the burst period the device requires in increments of 8 PCI clocks.

This register and the ATU Maximum Latency register are information-only registers which the configuration uses to determine how often a bus master typically requires access to the PCI bus and the duration of a typical transfer when it does acquire the bus. This information is useful in determining the values to be programmed into the bus master latency timers and in programming the algorithm to be used by the PCI bus arbiter.

Table 15-47. ATU Minimum Grant Register - ATUMGNT



Intel® i960® Core Local Bus Address
123EH

PCI Configuration Address Offset
3EH

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

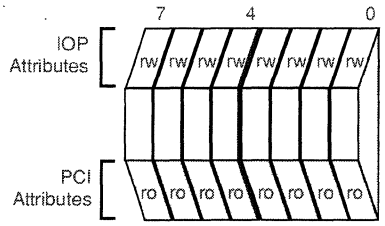
Bit	Default	Description
07:00	00H	This register specifies how long a burst period the device needs in increments of 8 PCI clocks. A zero value indicates the device has no stringent requirement.

15.7.20 ATU Maximum Latency Register - ATUMLAT

ATU Maximum Latency Register bit definitions adhere to *PCI Local Bus Specification,* Revision 2.2. This register specifies how often the device needs to access the PCI bus in increments of 8 PCI clocks.

This register and the Minimum Grant Register are information-only registers which the configuration uses to determine how often a bus master typically requires access to the PCI bus and the duration of a typical transfer when it does acquire the bus. This information is useful in determining the values to be programmed into the bus master latency timers and in programming the algorithm to be used by the PCI bus arbiter.

Table 15-48. ATU Maximum Latency Register - ATUMLAT



Intel® i960® Core Local Bus Address
123FH

PCI Configuration Address Offset
3FH

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set

RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
07:00	00H	Specifies frequency (how often) the device needs to access the PCI bus in increments of 8 PCI clocks. A zero value indicates the device has no stringent requirement.

15.7.21 Primary Inbound ATU Limit Register - PIALR

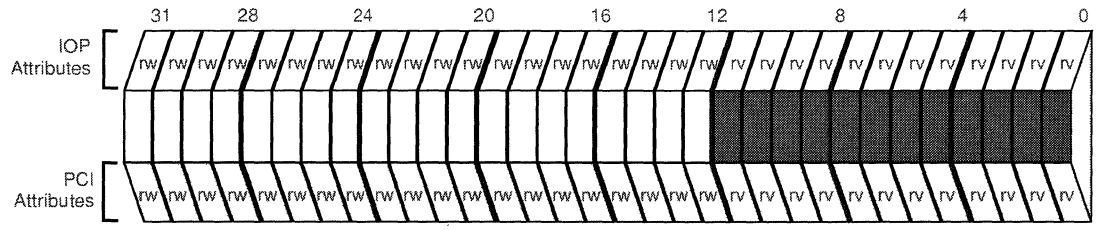
Primary inbound address translation occurs for data transfers occurring from the PCI bus (originated from the primary PCI bus) to the 80303 I/O processor internal bus. The address translation block converts PCI addresses to internal bus addresses.

The primary inbound translation base address is specified in Section 15.7.11. When determining block size requirements — as described in Section 15.7.16 — the primary translation limit register provides the block size requirements for the primary base address register. The remaining registers used for performing address translation are discussed in Section 15.2.1.1.

The 80303 I/O processor value register's programmed value must be naturally aligned with the base address register's programmed value. The limit register is used as a mask; thus, the lower address bits programmed into the 80303 I/O processor value register are invalid. Refer to the *PCI Local Bus Specification,* Revision 2.2 for additional information on programming base address registers.

Bits 31 to 12 within the PIALR have a direct effect on the PIABAR register, bits 31 to 12, with a one to one correspondence. A value of 0 in a bit within the PIALR makes the corresponding bit within the PIABAR a read only bit which always returns 0. A value of 1 in a bit within the PIALR makes the corresponding bit within the PIABAR read/write from PCI. Note that a consequence of this programming scheme is that unless a valid value exists within the PIALR, all writes to the PIABAR will have no effect since a value of all zeros within the PIALR makes the PIABAR a read only register.

Table 15-49. Primary Inbound ATU Limit Register - PIALR



Intel® i960® Core Local Bus Address: 1240H

PCI Configuration Address Offset: 40H - 43H

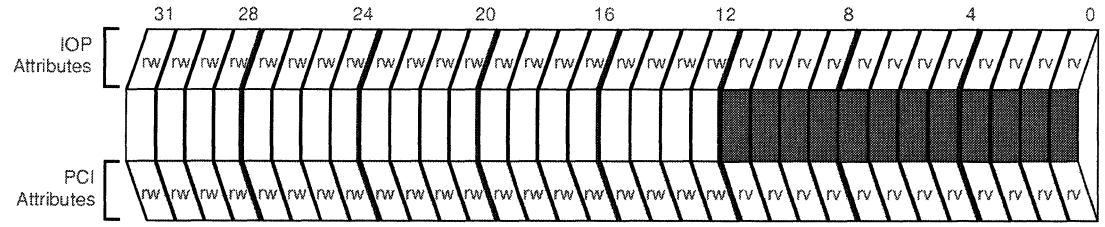
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:12	FF000H	Primary Inbound Translation Limit - This readback value determines the memory block size required for the primary ATU translation unit. This defaults to an inbound window of 16MB.
11:00	000H	Reserved

15.7.22 Primary Inbound ATU Translate Value Register - PIATVR

The Primary Inbound ATU Translate Value Register (PIATVR) contains the internal bus address used to convert primary PCI bus addresses. The converted address is driven on the internal bus as a result of the primary inbound ATU address translation.

Table 15-50. Primary Inbound ATU Translate Value Register - PIATVR



Intel® i960® Core Local Bus Address: 1244H

PCI Configuration Address Offset: 44H - 47H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:12	00001H	Primary Inbound ATU Translation Value - This value is used to convert the primary PCI address to internal bus addresses. This value must be 64-bit aligned on the internal bus. The default address allows the ATU to access the internal 80303 I/O processor memory-mapped registers.
11:00	000H	Reserved

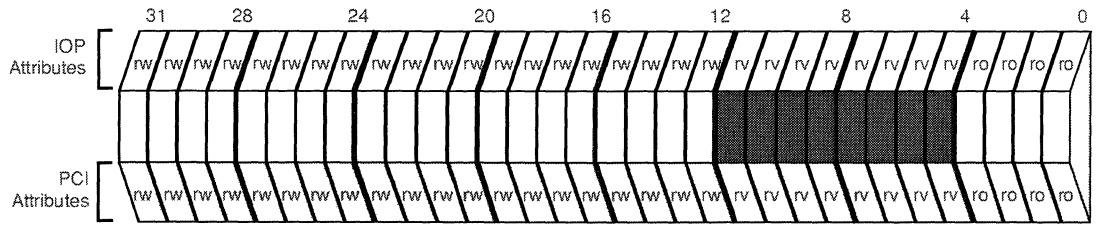
15.7.23 Secondary Inbound ATU Base Address Register - SIABAR

The Secondary Inbound ATU Base Address Register (SIABAR) defines the block of memory addresses where the secondary inbound translation window begins. The inbound ATU decodes and forwards the bus request to the 80303 I/O processor internal bus with a translated address to map into the 80303 I/O processor internal memory. The SIABAR defines the base address and describes the required memory block size; see Section 15.7.16, "Determining Block Sizes for Base Address Registers" on page 15-76. Bits 31 through 12 of the SIABAR will be either read/write bits or read only with a value of 0 depending on the value located within the SIALR. This configuration allows the SIABAR to be programmed per *PCI Local Bus Specification*, Revision 2.2.

The base address register's programmed value must comply with the PCI programming requirements for address alignment. Refer to the *PCI Local Bus Specification*, Revision 2.2 for additional information on programming base address registers.

Note: When trying to access the Messaging Unit from the Secondary PCI bus through the Bridge (see Section 15.7.38, "ATU Configuration Register - ATUCR" on page 15-98, Secondary Bus Messaging Unit Access Enable Bit), the SIABAR must be programmed the same as the PIABAR.

Table 15-51. Secondary Inbound ATU Base Address Register - SIABAR



Intel® i960® Core Local Bus Address
1248H

PCI Configuration Address Offset
48H - 4BH

Attribute Legend:
RW = Read/Write
RV = Reserved
RC = Read Clear
PR = Preserved
RO = Read Only
RS = Read/Set
NA = Not Accessible

Bit	Default	Description
31:12	00000H	Secondary Translation Base Address - These bits define the actual location to which the Secondary Translation function responds when addressed from the secondary PCI bus. The default block size is indeterminate.
11:04	00H	Reserved.
03	1_2	Prefetchable Indicator - This bit defines the memory spaces as prefetchable.
02:01	00_2	Address Type - These bits define where the block of memory can be located. The base address must be located anywhere in the first 4 Gbyte of address space (lower 32-bits of address).
00	0_2	Memory Space Indicator - This bit shows the register contents describes memory or I/O space base address. The ATU does not occupy I/O space; thus, this bit must be zero.

15.7.24 Secondary Inbound ATU Limit Register - SIALR

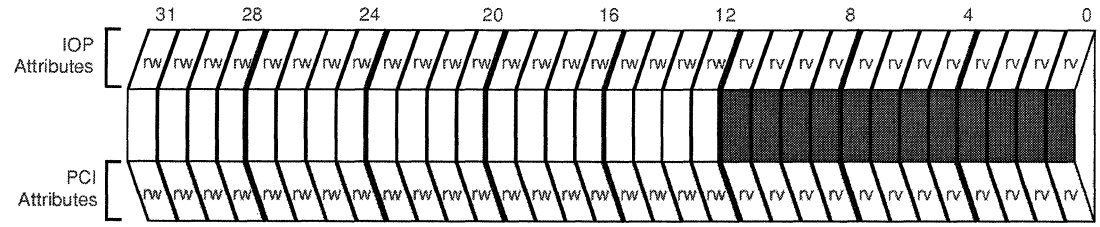
Secondary inbound address translation occurs for data transfers occurring from the secondary PCI bus to the 80303 I/O processor internal bus. The address translation block converts the PCI addresses to internal bus addresses.

The secondary translation base address is specified in Section 15.7.23, "Secondary Inbound ATU Base Address Register - SIABAR" on page 15-83. When determining the block size requirements as described in Section 15.7.16, "Determining Block Sizes for Base Address Registers" on page 15-76, the secondary limit register provides the block size requirements for the secondary base address register. The remaining registers used for performing address translation are discussed in Section 15.2.1.1, "Inbound Address Translation" on page 15-6.

The programmed value within the 80303 I/O processor value register must be naturally aligned with the programmed value found in the base address register. The limit register is used as a mask thus the lower address bits programmed into the 80303 I/O processor value register is invalid. The default value for the limit register is FFFFE000H, which is a 8 KByte limit. Refer to the *PCI Local Bus Specification,* Revision 2.2 for additional information on programming base address registers.

Bits 31 to 12 within the SIALR have a direct effect on the SIABAR register, bits 31 to 12, with a one to one correspondence. A value of 0 in a bit within the SIALR makes the corresponding bit within the SIABAR a read only bit which always returns 0. A value of 1 in a bit within the SIALR makes the corresponding bit within the SIABAR read/write from PCI. Note that a consequence of this programming scheme is that unless a valid value exists within the SIALR, all writes to the SIABAR will have no effect since a value of all zeros within the SIALR makes the SIABAR a read only register.

Table 15-52. Secondary Inbound ATU Limit Register - SIALR



Intel® i960® Core Local Bus Address: 124CH

PCI Configuration Address Offset: 4CH - 4FH

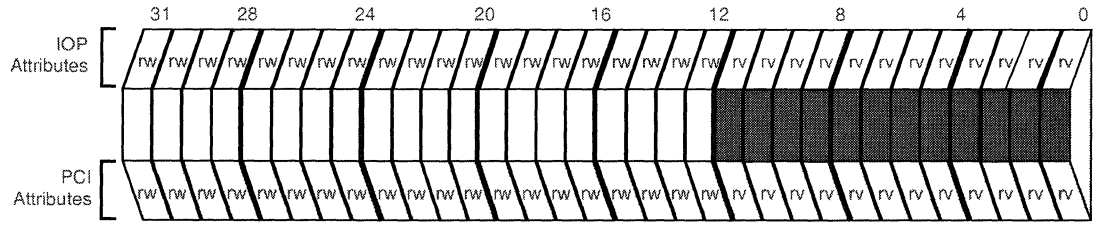
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:12	FFFFEH	Secondary Inbound ATU Limit - This is the read back value that determines the block size of memory required for the secondary ATU translation unit. Default size is 8 KB.
11:00	000H	Reserved

15.7.25 Secondary Inbound ATU Translate Value Register - SIATVR

The Secondary Inbound ATU Translate Value Register (SIATVR) contains the 80303 I/O processor internal bus address used to convert the secondary PCI bus address to a internal bus address. This address is driven on the 80303 I/O processor internal bus as a result of the secondary inbound ATU address translation.

Table 15-53. Secondary Inbound Translate ATU Value Register - SIATVR



Intel® i960® Core Local Bus Address
1250H

PCI Configuration Address Offset
50H - 53H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

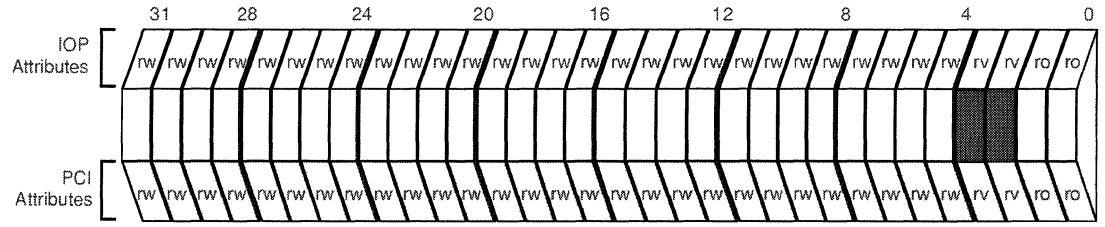
Bit	Default	Description
31:12	00001H	Secondary Inbound ATU Translate Value - Used to convert the secondary PCI address to a internal bus address. The secondary inbound address translation value must be 64-bit aligned on the 80303 I/O processor internal bus. (The default value of the entire register is 0000 1000H.)
11:00	000H	Reserved

15.7.26 Primary Outbound Memory Window Value Register - POMWVR

The Primary Outbound Memory Window Value Register (POMWVR) contains the primary PCI address used to convert 80303 I/O processor internal bus addresses for outbound transactions. This address is driven on the primary PCI bus as a result of the primary outbound ATU address translation. See Section 15.2.2.1, "Outbound Address Translation" on page 15-15 for details on outbound address translation.

The primary memory window is from internal bus address 8000 000H to 83FF FFFFH with the fixed length of 64 Mbytes.

Table 15-54. Primary Outbound Memory Window Value Register - POMWVR



Intel® i960® Core Local Bus Address: 1254H

PCI Configuration Address Offset: 54H - 57H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

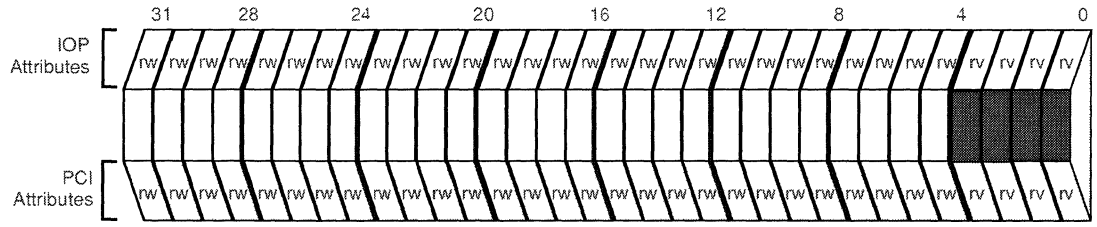
Bit	Default	Description
31:04	0000 000H	Primary Outbound MW Value - Used to convert 80303 I/O processor internal bus addresses to PCI addresses.
03:02	00_2	Reserved
01:00	00_2	Burst Order - This bit field shows the address sequence during a memory burst. Only linear incrementing mode is supported.

15.7.27 Primary Outbound I/O Window Value Register - POIOWVR

The Primary Outbound I/O Window Value Register (POIOWVR) contains the primary PCI I/O address used to convert the internal bus access to a PCI address. This address is driven on the primary PCI bus as a result of the primary outbound ATU address translation. See Section 15.2.2.1, "Outbound Address Translation" on page 15-15 for details on outbound address translation.

The primary I/O window is from 80303 I/O processor internal bus address 9000 000H to 9000 FFFFH with the fixed length of 64 Kbytes.

Table 15-55. Primary Outbound I/O Window Value Register - POIOWVR



Intel® i960® Core Local Bus Address: 125CH

PCI Configuration Address Offset: 5CH - 5FH

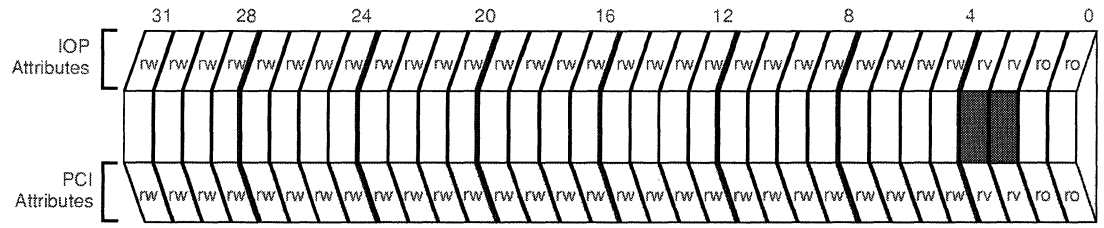
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:04	0000 000H	Primary Outbound I/O Window Value - Used to convert internal bus addresses to PCI addresses.
03:00	0H	Reserved

15.7.28 Primary Outbound DAC Window Value Register - PODWVR

The Primary Outbound DAC Window Value Register (PODWVR) contains the primary PCI DAC address used to convert a 80303 I/O processor internal bus address. This address is driven on the primary PCI bus as a result of the primary outbound ATU address translation. See Section 15.2.2.1, "Outbound Address Translation" on page 15-15 for details on outbound address translation. This register is used in conjunction with the Primary Outbound Upper 64-Bit DAC Register. The primary DAC window is from 80303 I/O processor internal bus address 8400 000H to 87FF FFFFH with the fixed length of 64 Mbytes.

Table 15-56. Primary Outbound DAC Window Value Register - PODWVR



Intel® i960® Core Local Bus Address: 1260H

PCI Configuration Address Offset: 60H - 63H

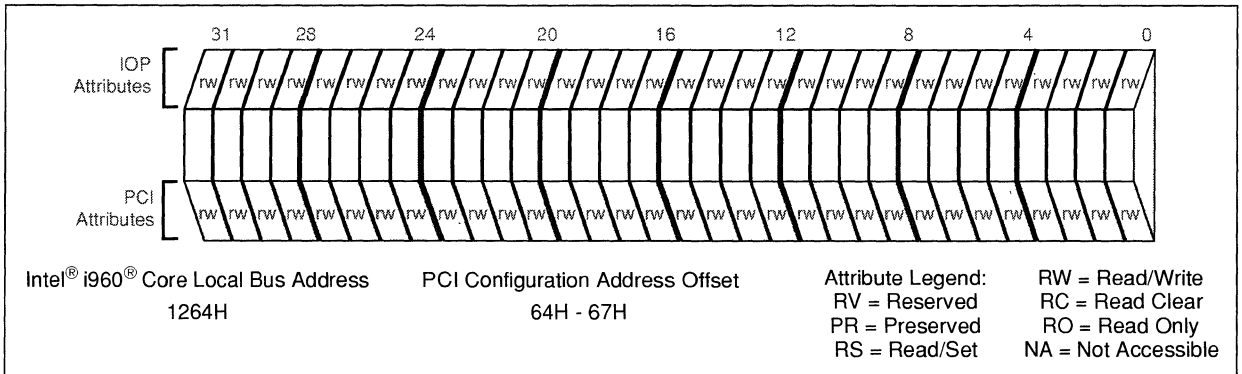
Attribute Legend:
RW = Read/Write
RV = Reserved
RC = Read Clear
PR = Preserved
RO = Read Only
RS = Read/Set
NA = Not Accessible

Bit	Default	Description
31:04	0000 000H	Primary Outbound DAC Window Value - This value the primary ATU uses to convert 80303 I/O processor internal bus addresses to PCI addresses.
03:02	00_2	Reserved
01:00	00_2	Burst Order - This bit field shows the address sequence during a memory burst. Only linear incrementing mode is supported.

15.7.29 Primary Outbound Upper 64-bit DAC Register - POUDR

The Primary Outbound Upper 64-bit DAC Register (POUDR) defines the upper 32-bits of address used during a dual address cycle. This enables the primary outbound ATU to directly address anywhere within the 64-bit host address space.

Table 15-57. Primary Outbound Upper 64-bit DAC Register - POUDR



Intel® i960® Core Local Bus Address
1264H

PCI Configuration Address Offset
64H - 67H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

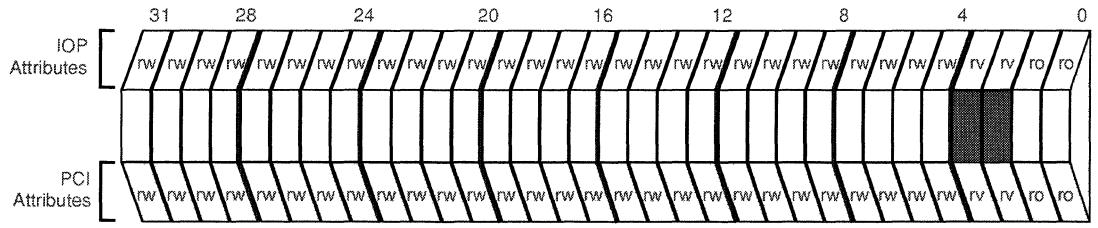
Bit	Default	Description
31:00	0000 0000H	These bits define the upper 32-bits of address driven during the dual address cycle (DAC).

15.7.30 Secondary Outbound Memory Window Value Register - SOMWVR

The Secondary Outbound Memory Window Value Register (SOMWVR) contains the secondary PCI address used to convert 80303 I/O processor internal bus addresses for outbound transactions. This address is driven on the secondary PCI bus as a result of the secondary outbound ATU address translation. See Section 15.2.2.1, "Outbound Address Translation" on page 15-15 for details on outbound address translation.

The secondary memory window is from 80303 I/O processor internal bus address 8800 000H to 8BFF FFFFH with the fixed length of 64 Mbytes.

Table 15-58. Secondary Outbound Memory Window Value Register - SOMWVR



Intel® i960® Core Local Bus Address: 1268H

PCI Configuration Address Offset: 68H - 6BH

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

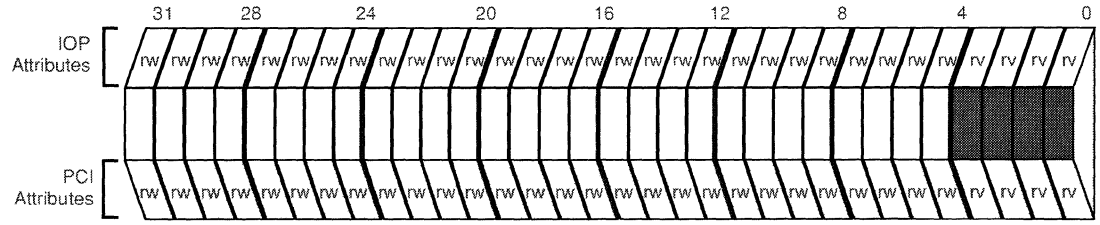
Bit	Default	Description
31:04	0000 000H	Secondary Outbound Memory Window Value - Used to convert 80303 I/O processor internal bus addresses to PCI addresses.
03:02	00_2	Reserved
01:00	00_2	Burst Order - This bit field shows the address sequence during a memory burst. Only linear incrementing mode is supported.

15.7.31 Secondary Outbound I/O Window Value Register - SOIOWVR

The Secondary Outbound I/O Window Value Register (SOIOWVR) contains the secondary PCI I/O address used to convert 80303 I/O processor internal bus addresses. This address is driven on the secondary PCI bus as a result of the secondary outbound ATU address translation. See Section 15.2.2.1, "Outbound Address Translation" on page 15-15 for details on outbound address translation.

The secondary I/O window is from 80303 I/O processor internal bus address 9001 0000H to 9001 FFFFH with the fixed length of 64 Kbytes.

Table 15-59. Secondary Outbound I/O Window Value Register - SOIOWVR



Intel® i960® Core Local Bus Address: 126CH

PCI Configuration Address Offset: 6CH - 6FH

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

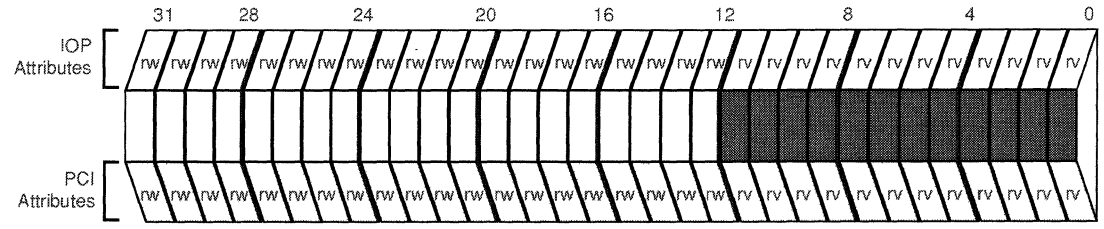
Bit	Default	Description
31:04	0000 000H	Secondary Outbound I/O Window Value - Used to convert internal bus addresses to PCI addresses.
03:00	0H	Reserved

15.7.32 Expansion ROM Limit Register - ERLR

The Expansion ROM Limit Register (ERLR) defines the block size of addresses the primary ATU will define as Expansion ROM address space. The block size is programmed by writing a value into the ERLR from the i960 core processor.

Bits 31 to 12 within the ERLR have a direct effect on the ERBAR register, bits 31 to 12, with a one to one correspondence. A value of 0 in a bit within the ERLR makes the corresponding bit within the ERBAR a read only bit which always returns 0. A value of 1 in a bit within the ERLR makes the corresponding bit within the ERBAR read/write from PCI.

Table 15-60. Expansion ROM Limit Register - ERLR



Intel® i960® Core Local Bus Address
1274H

PCI Configuration Address Offset
74H - 77H

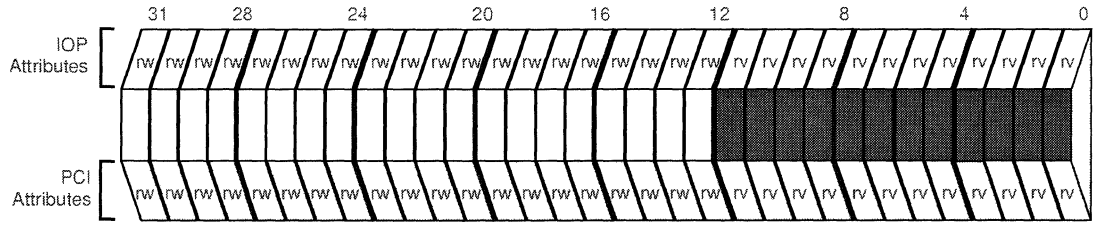
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:12	000000H	Expansion ROM Limit - Block size of memory required for the Expansion ROM translation unit. Default value is 0, which indicates no Expansion ROM address space and all bits within the ERBAR are read only with a value of 0.
11:00	000H	Reserved

15.7.33 Expansion ROM Translate Value Register - ERTVR

The Expansion ROM Translate Value Register contains the 80303 I/O processor internal bus address which the primary ATU will convert the primary PCI bus access. This address is driven on the internal bus as a result of the primary Expansion ROM address translation.

Table 15-61. Expansion ROM Translate Value Register - ERTVR



Intel® i960® Core Local Bus Address
1278H

PCI Configuration Address Offset
78H - 7BH

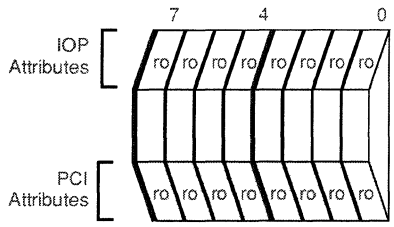
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:12	00000H	Expansion ROM Translation Value - Used to convert PCI addresses to 80303 I/O processor internal bus addresses for Expansion ROM accesses. The Expansion ROM address translation value must be word aligned on the internal bus.
11:00	000H	Reserved

15.7.34 ATU_Capability Identifier Register - ATU_Cap_ID

The Capability Identifier Register bits adhere to the definitions in the *PCI Local Bus Specification,* Revision 2.2. This register in the PCI Extended Capability header identifies the type of Extended Capability contained in that header. In the case of the 80303 I/O processor, this is the PCI Bus Power Management extended capability with an ID of 01H as defined by the *PCI Bus Power Management Interface Specification,* Revision 1.1.

Table 15-62. ATU_Capability Identifier Register - ATU_Cap_ID



Intel® i960® Core Local Bus Address
0000 1280H

PCI Configuration Offset
80H

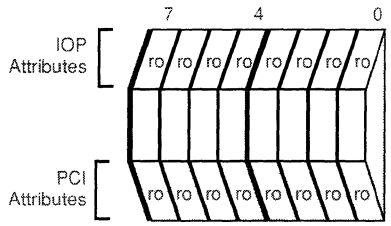
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
07:00	01H	Cap_Id - This field with its' 01H value identifies this item in the linked list of Extended Capability Headers as being the PCI Power Management Registers.

15.7.35 ATU Next Item Pointer Register - ATU_Next_Item_Ptr

The Next Item Pointer Register bits adhere to the definitions in the *PCI Local Bus Specification*, Revision 2.2. This register describes the location of the next item in the function's capability list. For the 80303 I/O processor, the Power Management Registers are the only Extended Capability Supported, thus the Next_Item_Ptr is set to 00H, indicating the end of the Capabilities List.

Table 15-63. ATU Next Item Pointer Register - ATU_Next_Item_Ptr



Intel® i960® Core Local Bus Address
0000 1281H

PCI Configuration Offset
81H

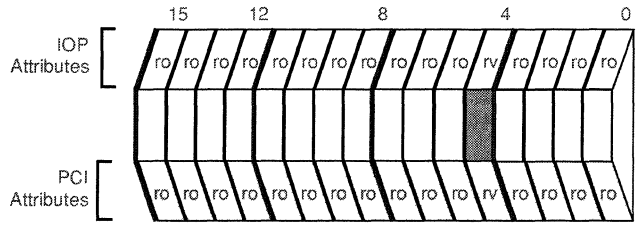
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
07:00	00H	Next_ Item_ Pointer - This field provides an offset into the function's configuration space pointing to the next item in the function's capability list. Since there are no other Extended Capabilities besides PCI Power Management in the 80303 I/O processor, the register is set to 00H.

15.7.36 ATU Power Management Capabilities Register - APMCR

Power Management Capabilities bits adhere to the definitions in the *PCI Bus Power Management Interface Specification,* Revision 1.1. This register is a 16-bit read-only register which provides information on the capabilities of the ATU function related to power management.

Table 15-64. ATU Power Management Capabilities Register - APMCR



Intel® i960® Core Local Bus Address
0000 1282H

PCI Configuration Offset
82- 83H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set

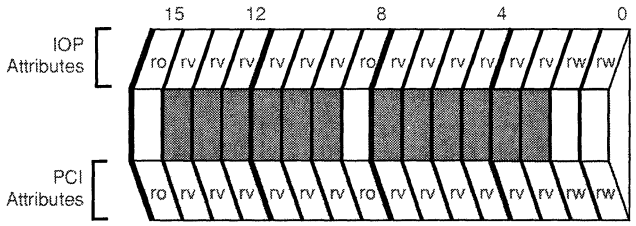
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
15:11	00000_2	PME_Support - This function is not capable of asserting the PME# signal in any state, since PME# is not supported by the 80303 I/O processor, thus the value of PME_Support is set to 00000B.
10	0_2	D2_Support - This bit is set to 0_2 indicating that the 80303 I/O processor does not support the D2 Power Management State
9	0_2	D1_Support - This bit is set to 0_2 indicating that the 80303 I/O processor does not support the D1 Power Management State
8:6	000_2	Aux_Current - This field is set to 000_2 indicating that the 80303 I/O processor has no current requirements for the 3.3Vaux signal as defined in the *PCI Bus Power Management Interface Specification,* Revision 1.1.
5	0_2	DSI - This field is set to 0_2 meaning that this function will require a device specific initialization sequence following the transition to the D0 uninitialized state.
4	0_2	Reserved.
3	0_2	PME Clock - Since the 80303 I/O processor does not support PME# signal generation, this value is set to 0_2
2:0	010_2	Version - Setting these bits to 010_2 means that this function complies with *PCI Bus Power Management Interface Specification,* Revision 1.1.

15.7.37 ATU Power Management Control/Status Register - APMCSR

Power Management Control/Status bits adhere to the definitions in the *PCI Bus Power Management Interface Specification*, Revision 1.1. This 16-bit register is the primary control and status interface for the power management extended capability.

Table 15-65. ATU Power Management Control/Status Register - APMCSR



Intel® i960® Core Local Bus Address
0000 1284H

PCI Configuration Offset
84- 85H

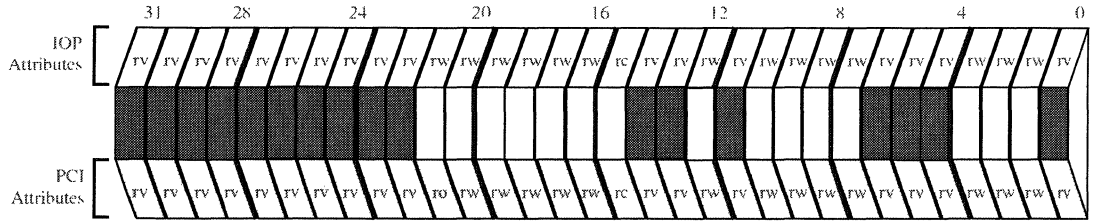
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
15	0_2	PME_Status - This function is not capable of asserting the PME# signal in any state, since PME# is not supported by the 80303 I/O processor, thus the value of PME_Status is set to 0_2 and is read-only.
14:9	00H	Reserved
8	0_2	PME_En - This bit is hardwired to read-only 0_2 since this function does not support PME# generation from any power state.
7:2	000000_2	Reserved
1:0	00_2	Power State - This 2-bit field is used both to determine the current power state of a function and to set the function into a new power state. The definition of the values is: 00_2 - D0 01_2 - D1 (Unsupported) 10_2 - D2 (Unsupported) 11_2 - $D3_{hot}$ The 80303 I/O processor supports only the D0 and $D3_{hot}$ states.i960 core processor

15.7.38 ATU Configuration Register - ATUCR

The ATU Configuration Register controls the outbound address translation for both the primary and secondary outbound translation units. It also contains bits for DRC aliasing, discard timer status, P_SERR# and S_SERR# manual assertion, access to the messaging unit from the secondary PCI bus, P_SERR# and S_SERR# detection interrupt masking, and ATU Bist interrupt enabling.

Table 15-66. ATU Configuration Register - ATUCR (Sheet 1 of 2)



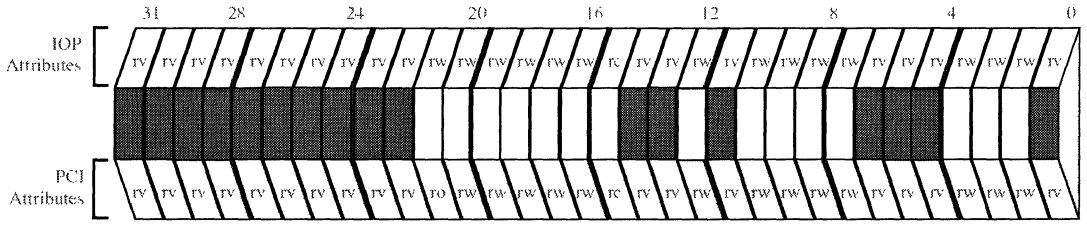
Intel® i960® Core Local Bus Address: 1288H

PCI Configuration Address Offset: 88H - 8BH

Attribute Legend: RV = Reserved, PR = Preserved, RS = Read/Set, RW = Read/Write, RC = Read Clear, RO = Read Only, NA = Not Accessible

Bit	Default	Description
31:22	00H	Reserved
21	0_2	Bridge Function Number - this bit in conjunction with the ATU Header Type Register (ATUHTR) and the Bridge Header Type Register (HTR), can swap the Bridge and the ATU device numbers as they appear to the PCI bus, or it can set the 80303 I/O processor as a single function device with either the ATU or the Bridge as the single function. (Refer to Section 15.2.4, "PCI Multi-Function Device Swapping/Disabling" on page 15-23 for programming information.)
20	0_2	SATU DRC Alias - when set, the secondary ATU does not distinguish read commands when attempting to match a current PCI read transaction with read data enqueued within the DRC buffer. When clear, a current read transaction must have the exact same read command as the DRR for the secondary ATU to deliver DRC data.
19	0_2	PATU DRC Alias - when set, the primary ATU does not distinguish read commands when attempting to match a current PCI read transaction with read data enqueued within the DRC buffer. When clear, a current read transaction must have the exact same read command as the DRR for the primary ATU to deliver DRC data.
18	0_2	Direct Addressing Upper 2Gbytes Translation Enable - When set, with Direct Addressing enabled (bit 7 of the ATUCR set), the ATU will forward internal bus cycles with an address between 0000.2000H and 7FFF.FFFFH to the PCI bus with bit 31 of the address set (8000.2000H - FFFF.FFFFH). When clear, no translation will occur.
17	0_2	S_SERR# Manual Assertion - when set, the SATU will assert **S_SERR#** for one clock on the secondary PCI interface. Until cleared, **S_SERR#** may not be manually asserted again. Once cleared, operation will proceed as specified.
16	0_2	**P_SERR#** Manual Assertion - when set, the PATU will assert **P_SERR#** for one clock on the primary PCI interface. Until cleared, **P_SERR#** may not be manually asserted again. Once cleared, operation will proceed as specified.
15	0_2	ATU Discard Timer Status - when set, one of the 3 discard timers within the PATU and SATU has expired and discarded the delayed completion transaction within the queue. When clear, no timer has expired.
14:13	00_2	Reserved

Table 15-66. ATU Configuration Register - ATUCR (Sheet 2 of 2)



Intel® i960® Core Local Bus Address: 1288H

PCI Configuration Address Offset: 88H - 8BH

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
12	0_2	Secondary Bus Messaging Unit Access Enable - If set, the secondary addresses which fall within the first 4 KB of the SATU inbound address space and are also capable of being claimed by the secondary interface of the bridge will default to the bridge for forwarding to the MU on the primary interface. If clear, the SATU has priority and will claim addresses that are both within the first 4 KB of the SATU and/or are capable of being claimed by the bridge unit. Setting this bit will simultaneously allow the Messaging Unit to claim target cycles which are mastered by the primary interface of the Bridge Unit.
11	0_2	Reserved
10	0_2	**S_SERR#** Interrupt Enable - When set, the i960 core processor will be signalled an **NMI#** interrupt if the SATU detects that **S_SERR#** was asserted on the secondary interface. When clear, the i960 core processor will not be interrupted when **S_SERR#** is detected as asserted on the secondary interface.
09	0_2	**P_SERR#** Interrupt Enable - When set, the i960 core processor will be signalled an **NMI#** interrupt if the PATU detects that **P_SERR#** was asserted on the primary interface. When clear, the i960 core processor will not be interrupted when **P_SERR#** is detected as asserted on the primary interface.
08	0_2	Direct Addressing Enable - Setting this bit will enable direct outbound addressing through the ATUs. Internal bus cycles with an address between 0000.2000H and 7FFF.FFFFH will automatically be forwarded to the PCI bus with or without translation of address bit 31 based on the setting of bit 18 of the ATUCR.
07	0_2	Secondary Direct Addressing Select - When set, results in direct addressing outbound transactions to be forwarded through the secondary ATU to the secondary PCI bus. When clear, direct addressing uses the primary ATU and the primary PCI bus. The Direct Addressing Enable bit must be set to enable direct addressing.
06:04	000_2	Reserved
03	0_2	ATU BIST Interrupt Enable - When set, enables an interrupt to the i960 core processor when the start BIST bit is set in the ATUBISTR register. This bit is also reflected as the BIST Capable bit 7 in the ATUBISTR register.
02	0_2	Secondary Outbound ATU Enable - When set, enables the secondary outbound address translation unit. When cleared, disables the secondary outbound ATU.
01	0_2	Primary Outbound ATU Enable - When set, enables the primary outbound address translation unit. When cleared, disables the primary outbound ATU.
00	0_2	Reserved

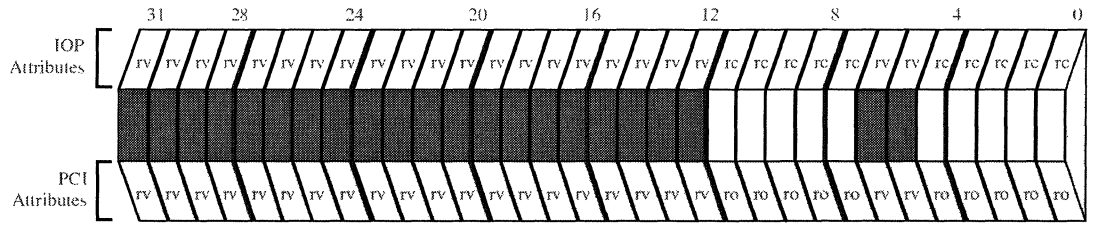
15.7.39 Primary ATU Interrupt Status Register - PATUISR

The Primary ATU Interrupt Status Register is used to notify the core processor of the source of a Primary ATU interrupt. In addition, this register is written to clear the source of the interrupt to the interrupt unit of the 80303 I/O processor. All bits in this register are Read/Clear.

Bits 4:0 are a direct reflection of bits 14:11 and bit 8 (respectively) of the Primary ATU Status Register (these bits are set at the same time by hardware but need to be cleared independently). Bit 7 is set by an error associated with the internal bus of the 80303 I/O processor. Bit 8 is for software BIST. The conditions that result in a Primary ATU interrupt are cleared by writing a 1 to the appropriate bits in this register.

Note that bits 4:0, bits 12:11, bit 9 and bit 7 can result in an **NMI#** interrupt being driven to the i960 core processor.

Table 15-67. Primary ATU Interrupt Status Register - PATUISR (Sheet 1 of 2)



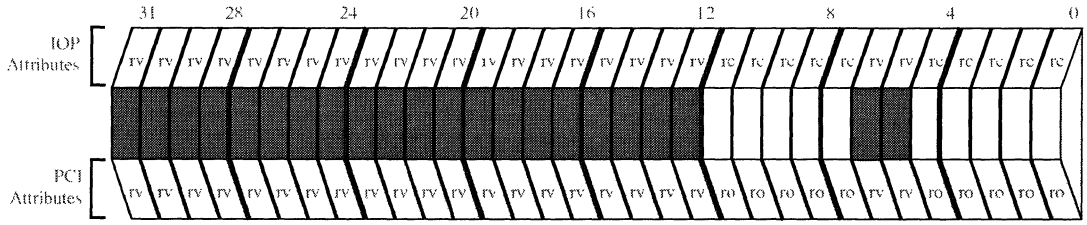
Intel® i960® Core Local Bus Address: 1290H

PCI Configuration Address Offset: 90H - 93H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:12	00000H	Reserved
11	0_2	Power State Transition - When the Power State Field of the ATU Power Management Control/Status Register is written to transition the ATU function Power State from either D3 to D0 or D0 to D3 and the ATU Power State Transition Interrupt mask bit is cleared, this bit is set.
10	0_2	**P_SERR#** Asserted - set when **P_SERR#** is asserted on the PCI bus by the primary ATU.
09	0_2	Detected Parity Error - set when a parity error is detected on the primary PCI bus even when the PATUCMD register's Parity Error Response bit is cleared. Set under the following conditions: • Write Data Parity Error when the PATU is a slave (inbound write). • Read Data Parity Error when the PATU is a master (outbound read). Read Data Parity Errors when DMA Channel 0 or DMA Channel 1 is the master ARE NOT logged here, and instead are logged in the appropriate DMA CSR. • Any Address Parity Error on the Primary Bus (including one generated by the PATU or DMA Channels 0 & 1 when loopback is enabled).
08	0_2	ATU BIST Interrupt - When set, the host processor has set the start BIST, ATUBISTR register bit 6, and the ATU BIST interrupt enable (ATUCR register bit 12) is enabled. The i960 core processor can initiate the software BIST and store the result in ATUBISTR register bits 3:0.
07	0_2	Internal Bus Master Abort - set when a transaction initiated by the ATU internal bus master interface ends in a Master-abort.
06:05	00_2	Reserved.
04	0_2	**P_SERR#** Detected - set when **P_SERR#** is detected on the PCI bus by the primary ATU.
03	0_2	PCI Master Abort - set when a transaction initiated by the ATU PCI master interface ends in a Master-abort.

Table 15-67. Primary ATU Interrupt Status Register - PATUISR (Sheet 2 of 2)



Intel® i960® Core Local Bus Address
1290H

PCI Configuration Address Offset
90H - 93H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
02	0_2	PCI Target Abort (master) - set when a transaction initiated by the ATU PCI master interface ends in a Target-abort.
01	0_2	PCI Target Abort (target) - set when the ATU interface, acting as a target, terminates the transaction on the PCI bus with a target abort.
00	0_2	PCI Master Parity Error - The ATU interface sets this bit when three conditions are met: • the PATU asserted S_PERR# or observed S_PERR# asserted • the PATU acted as the bus master for the operation in which the error occurred • Parity Error Response bit is set (in the Primary ATU Command Register)

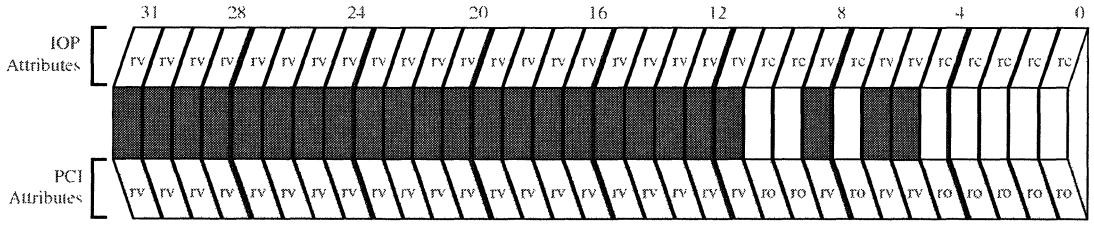
15.7.40 Secondary ATU Interrupt Status Register - SATUISR

The Secondary ATU Interrupt Status Register is used to notify the core processor of the source of a Secondary ATU interrupt. In addition, this register is written to clear the source of the interrupt to the interrupt unit of the 80303 I/O processor. All bits in this register are Read/Clear.

Bits 4:0 are a direct reflection of bits 14:11 and bit 8 (respectively) of the Secondary ATU Status Register (these bits are set at the same time by hardware but need to be cleared independently). Bit 7 is set by an error associated with the internal bus of the 80303 I/O processor. The conditions that result in a Secondary ATU interrupt are cleared by writing a 1 to the appropriate bits in this register.

Note that bits 4:0, bit 7, and bit 9 can result in an **NMI#** interrupt being driven to the i960 core processor.

Table 15-68. Secondary ATU Interrupt Status Register - SATUISR (Sheet 1 of 2)



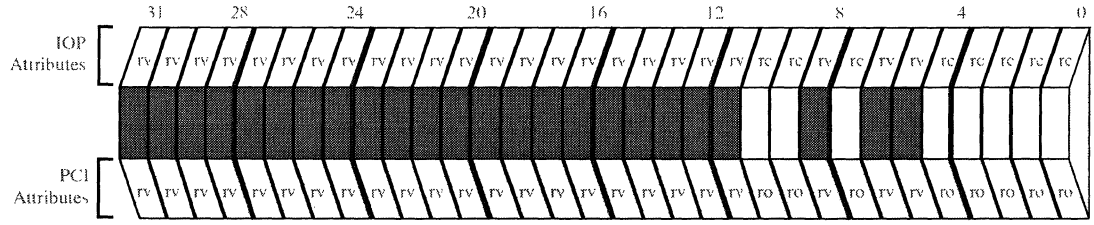
Intel® i960® Core Local Bus Address: 1294H

PCI Configuration Address Offset: 94H - 97H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:11	000000H	Reserved
10	0_2	**S_SERR#** Asserted - set when **S_SERR#** is asserted on the PCI bus by the secondary ATU.
09	0_2	Detected Parity Error - set when a parity error is detected on the secondary PCI bus even when the SATUCMD register's Parity Error Response bit is cleared. Set under the following conditions: • Write Data Parity Error when the SATU is a slave (inbound write). • Read Data Parity Error when the SATU is Master (outbound read). Read Data Parity Errors when DMA Channel 2 is a master ARE NOT logged here, and instead are logged in the DMA Channel 2 CSR. • Any Address Parity Error on the Secondary Bus (including one generated by the SATU or DMA Channel 2 when loopback is enabled).
08	0_2	Reserved
07	0_2	Internal Bus Master Abort - set when a transaction initiated by the ATU internal bus master interface ends in a Master-abort.
06:05	00_2	Reserved
04	0_2	**S_SERR#** Detected - set when **S_SERR#** is detected on the PCI bus by the secondary ATU.
03	0_2	PCI Master Abort - set when a transaction initiated by the ATU PCI master interface ends in a Master-abort.
02	0_2	PCI Target Abort (master) - set when a transaction initiated by the ATU PCI master interface ends in a Target-abort.
01	0_2	PCI Target Abort (target) - set when the ATU interface, acting as a target, terminates the transaction on the PCI bus with a target abort.

Table 15-68. Secondary ATU Interrupt Status Register - SATUISR (Sheet 2 of 2)



Intel® i960® Core Local Bus Address
1294H

PCI Configuration Address Offset
94H - 97H

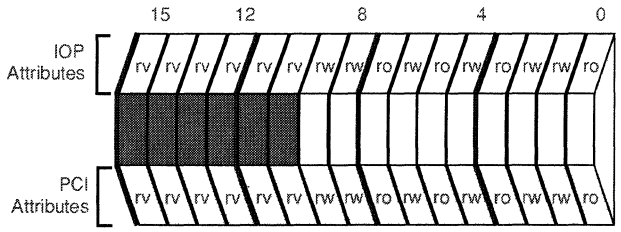
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
00	0_2	PCI Master Parity Error - The secondary ATU interface sets this bit when three conditions are met: • the SATU asserted S_PERR# or observed S_PERR# asserted • the SATU acted as the bus master for the operation in which the error occurred • Parity Error Response bit is set (in the Secondary ATU Command Register)

15.7.41 Secondary ATU Command Register - SATUCMD

Secondary ATU Command Register bits adhere to the definitions in the *PCI Local Bus Specification*, Revision 2.2 and in most cases affect the behavior of the device on the secondary PCI bus.

Table 15-69. Secondary ATU Command Register - SATUCMD



Intel® i960® Core Local Bus Address: 1298H

PCI Configuration Address Offset: 98H - 99H

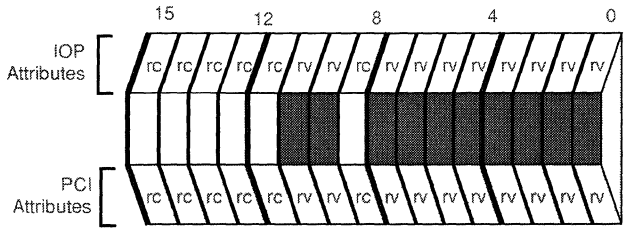
Attribute Legend: RV = Reserved, PR = Preserved, RS = Read/Set, RW = Read/Write, RC = Read Clear, RO = Read Only, NA = Not Accessible

Bit	Default	Description
15:10	00H	Reserved
09	0_2	Fast Back to Back Enable - When this bit is cleared, the secondary ATU interface is not allowed to generate fast back-to-back cycles on its bus.
08	0_2	S_SERR# Enable - When this bit is cleared, the secondary ATU interface is not allowed to assert S_SERR# on the PCI interface.
07	0_2	Wait Cycle Control - controls address/data stepping. Not implemented and a reserved bit field
06	0_2	Parity Error Response - When this bit is set, the secondary ATU and DMA channel 2 must take normal action when a parity error is detected. When it is cleared, parity checking is disabled.
05	0_2	VGA Palette Snoop Enable - The secondary ATU interface does not support I/O writes and therefore, does not perform VGA palette snooping.
04	0_2	Memory Write and Invalidate Enable - When this bit is set, DMA channel 2 may generate MWI commands. When this bit is clear, DMA channel 2 must use Memory Write commands instead of MWI.
03	0_2	Special Cycle Enable - The ATU interface does not respond to special cycle commands in any way. Not implemented and a reserved bit field
02	0_2	Bus Master Enable - The secondary ATU interface has the ability to act as a master on the PCI bus. A value of 0 disables the secondary ATU from claiming i960 core processor accesses and from generating PCI accesses. A value of 1 allows the secondary ATU to claim i960 core processor accesses and to behave as a PCI bus master. This enable bit also controls the master interface of the DMA channel 2. The bit must be set before initiating an DMA transfer on the PCI bus.
01	0_2	Memory Enable - Controls the secondary ATU interface's response to PCI memory addresses. When this bit is cleared, the ATU interface will not respond to any memory access on the PCI bus.
00	0_2	I/O Space Enable - Controls the ATU interface response to I/O transactions on the primary side. Not implemented and a reserved bit field.

15.7.42 Secondary ATU Status Register - SATUSR

Secondary ATU Status Register bits adhere to the definitions in the *PCI Local Bus Specification*, Revision 2.2. The *read/clear* bits can only be set by the internal hardware and are cleared by either a reset condition or by writing a 1_2 to the register.

Table 15-70. Secondary ATU Status Register - SATUSR



Intel® i960® Core Local Bus Address: 129AH

PCI Configuration Address Offset: 9AH - 9BH

Attribute Legend: RW = Read/Write; RV = Reserved; RC = Read Clear; PR = Preserved; RO = Read Only; RS = Read/Set; NA = Not Accessible

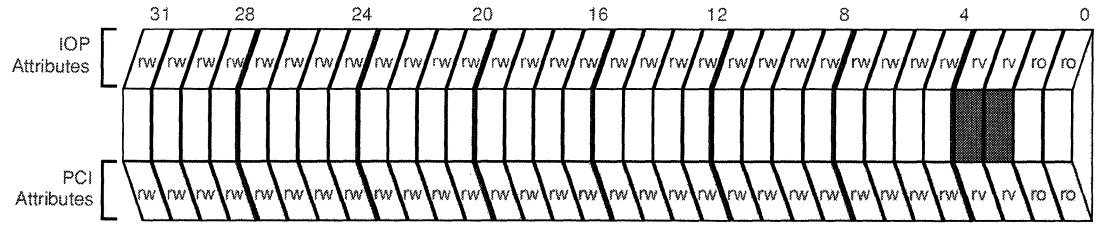
Bit	Default	Description
15	0_2	Detected Parity Error - set when a parity error is detected on the secondary PCI bus even when the SATUCMD register's Parity Error Response bit is cleared. Set under the following conditions: • Write Data Parity Error when the SATU is a slave (inbound write). • Read Data Parity Error when the SATU or DMA Channel 2 is a master (outbound read). • Any Address Parity Error on the Secondary Bus (including one generated by the SATU or DMA Channel 2).
14	0_2	S_SERR# Asserted - set when S_SERR# is asserted by the secondary ATU.
13	0_2	Master Abort - set when a transaction initiated by the secondary ATU PCI master interface, or DMA Channel 2 ends in a Master-abort. Setting of this bit due to an error condition from a DMA Channel will not cause an ATU interrupt to the core.
12	0_2	Target Abort (master) - set when a transaction initiated by the secondary ATU PCI master interface or the DMA Channel 2 master interface ends in a Target-abort. Setting of this bit due to an error condition from a DMA Channel will not cause an ATU interrupt to the core.
11	0_2	Target Abort (target) - set when the secondary ATU PCI interface, acting as a target, terminates the transaction on the secondary PCI bus with a Target-abort.
10:09	00_2	Reserved
08	0_2	Master Parity Error - The secondary ATU interface sets this bit under the following conditions: • The SATU or DMA Channel 2 asserted S_PERR# itself or the SATU observed S_PERR# asserted. • And the SATU or DMA Channel 2 acted as the bus master for the operation in which the error occurred. • And the SATUCMD register's Parity Error Response bit is set Setting of this bit due to an error condition from a DMA Channel will not cause an ATU interrupt to the core.
07:00	00H	Reserved

15.7.43 Secondary Outbound DAC Window Value Register - SODWVR

The Secondary Outbound DAC Window Value Register (SODWVR) contains the secondary PCI DAC address used to convert an 80303 I/O processor internal bus address. This address is driven on the secondary PCI bus as a result of the secondary outbound ATU address translation. See Section 15.2.2.1, "Outbound Address Translation" on page 15-15 for details on outbound address translation. This register is used in conjunction with the Secondary Outbound Upper 64-Bit DAC Register.

The secondary DAC window is from 80303 I/O processor internal bus address 8C00 000H to 8FFF FFFFH with the fixed length of 64 Mbytes.

Table 15-71. Secondary Outbound DAC Window Value Register - SODWVR



Intel® i960® Core Local Bus Address: 129CH

PCI Configuration Address Offset: 9CH - 9FH

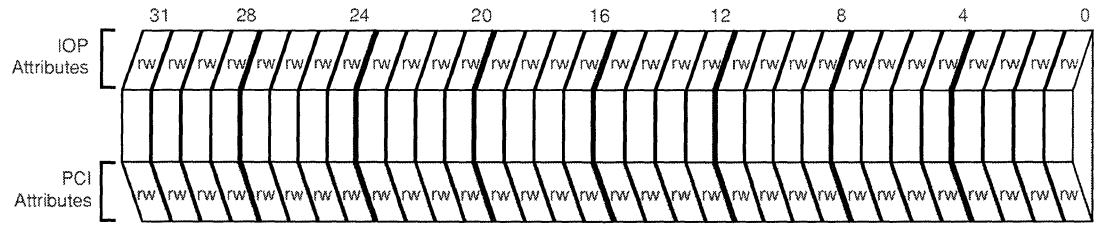
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:04	0000 000H	Secondary Outbound DAC Window Value - The secondary ATU uses this value to convert 80303 I/O processor internal bus addresses to PCI addresses.
03:02	00_2	Reserved
01:00	00_2	Burst Order - This bit field shows the address sequence during a memory burst. Only linear incrementing mode is supported.

15.7.44 Secondary Outbound Upper 64-bit DAC Register - SOUDR

The Secondary Outbound Upper 64-bit DAC Register (SOUDR) defines the upper 32-bits of address used during a dual address cycle. This enables the secondary outbound ATU to directly address anywhere within the 64-bit host address space.

Table 15-72. Secondary Outbound Upper 64-bit DAC Register - SOUDR



Intel® i960® Core Local Bus Address: 12A0H

PCI Configuration Address Offset: A0H - A3H

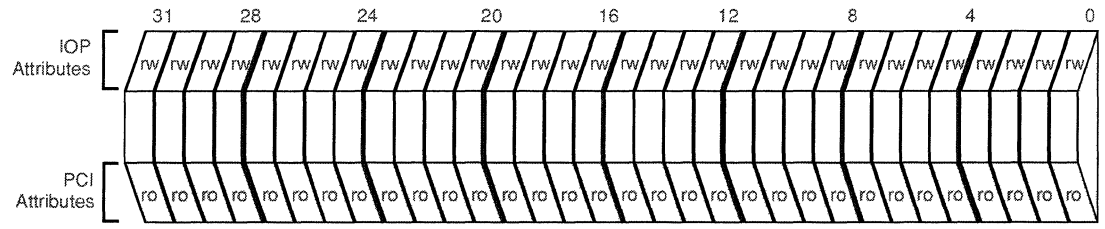
Attribute Legend:
RW = Read/Write
RV = Reserved
RC = Read Clear
PR = Preserved
RO = Read Only
RS = Read/Set
NA = Not Accessible

Bit	Default	Description
31:00	0000 0000H	Secondary Outbound Upper 64-bit DAC Address - These bits define the upper 32-bits of address driven during the dual address cycle (DAC).

15.7.45 Primary Outbound Configuration Cycle Address Register - POCCAR

The Primary Outbound Configuration Cycle Address Register is used to hold the 32-bit PCI configuration cycle address. The i960 core processor writes the PCI configuration cycles address which then enables the primary outbound configuration read or write. The i960 core processor then performs a read or write to the Primary Outbound Configuration Cycle Data Register to initiate the configuration cycle on the primary PCI bus.

Table 15-73. Primary Outbound Configuration Cycle Address Register - POCCAR



Intel® i960® Core Local Bus Address
12A4H

PCI Configuration Address Offset
A4H - A7H

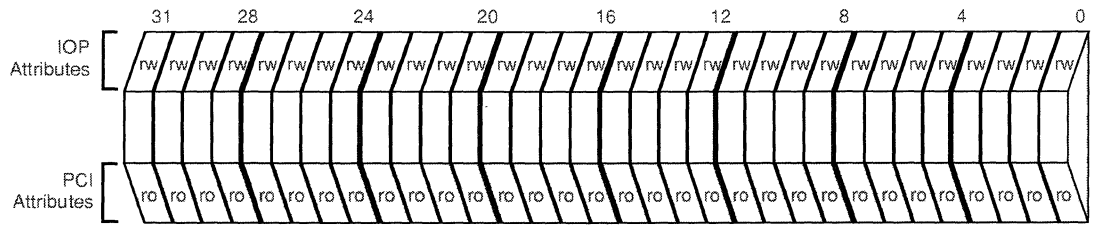
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:00	0000 0000H	Primary Configuration Cycle Address - These bits define the 32-bit PCI address used during an outbound configuration read or write cycle.

15.7.46 Secondary Outbound Configuration Cycle Address Register - SOCCAR

The Secondary Outbound Configuration Cycle Address Register is used to hold the 32-bit PCI configuration cycle address. The i960 core processor writes the PCI configuration cycles address which then enables the secondary outbound configuration read or write. The i960 core processor then performs a read or write to the Secondary Outbound Configuration Cycle Data Register to initiate the configuration cycle on the secondary PCI bus.

Table 15-74. Secondary Outbound Configuration Cycle Address Register - SOCCAR



Intel® i960® Core Local Bus Address
12A8H

PCI Configuration Address Offset
A8H - ABH

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

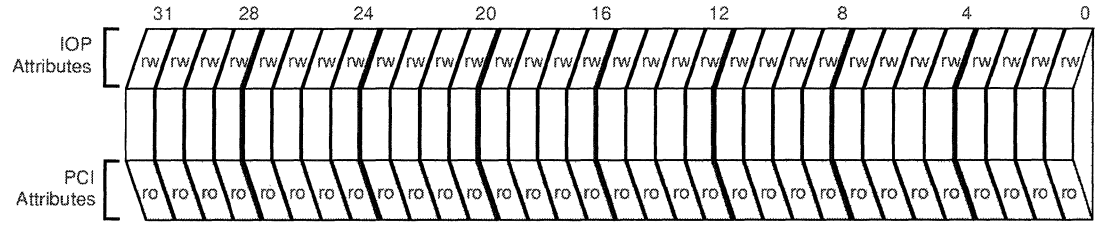
Bit	Default	Description
31:00	0000 0000H	Secondary Configuration Cycle Address - These bits define the 32-bit PCI address used during an outbound configuration read or write cycle.

15.7.47 Primary Outbound Configuration Cycle Data Register - POCCDR

The Primary Outbound Configuration Cycle Data Register is used to initiate a configuration read or write on the primary PCI bus. The register is logical rather than physical meaning that it is an address not a register. The i960 core processor will read or write the data registers memory-mapped address to initiate the configuration cycle on the PCI bus with the address found in the POCCAR. For a configuration write, the data is latched from the internal bus and forwarded directly to the P_OWQ. For a read, the data is returned directly from the P_ORQ to the i960 core processor and is never actually entered into the data register (which does not physically exist).

The POCCDR is only visible from 80303 I/O processor internal bus address space and appears as a reserved value within the ATU configuration space.

Table 15-75. Primary Outbound Configuration Cycle Data Register - POCCDR



Intel® i960® Core Local Bus Address
12ACH

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

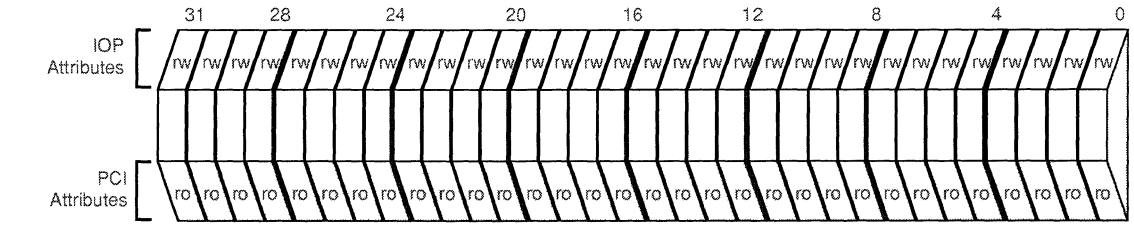
Bit	Default	Description
31:00	0000 0000H	Primary Configuration Cycle Data - These bits define the data used during an outbound configuration read or write cycle.

15.7.48 Secondary Outbound Configuration Cycle Data Register - SOCCDR

The Secondary Outbound Configuration Cycle Data Register is used to initiate a configuration read or write on the secondary PCI bus. The register is logical rather than physical meaning that it is an address not a register. The i960 core processor will read or write the data registers memory-mapped address to initiate the configuration cycle on the PCI bus with the address found in the SOCCAR. For a configuration write, the data is latched from the internal bus and forwarded directly to the S_OWQ. For a read, the data is returned directly from the S_ORQ to the i960 core processor and is never actually entered into the data register (which does not physically exist).

The SOCCDR is only visible from 80303 I/O processor internal bus address space and appears as a reserved value within the ATU configuration space.

Table 15-76. Secondary Outbound Configuration Cycle Data Register - SOCCDR



Intel® i960® Core Local Bus Address
12B0H

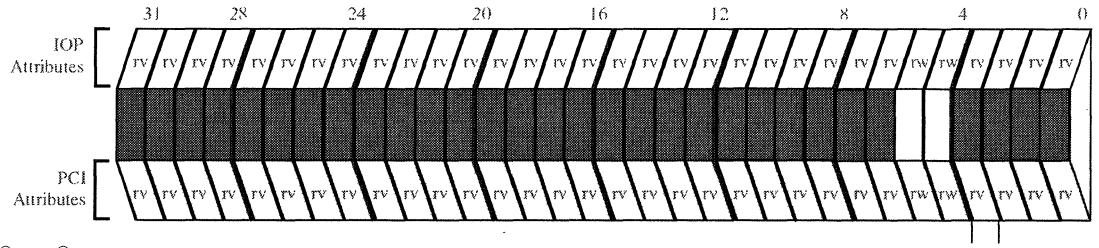
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:00	0000 0000H	Secondary Configuration Cycle Data - These bits define the data used during an outbound configuration read or write cycle.

15.7.49 Primary ATU Queue Control Register - PAQCR

The Primary ATU Queue Control Register contains programmable parameters affecting operation of the primary ATU queues.

Table 15-77. Primary ATU Queue Control Register - PAQCR



Intel® i960® Core Local Bus Address: 12B4H

PCI Configuration Address Offset: B4H - B7H

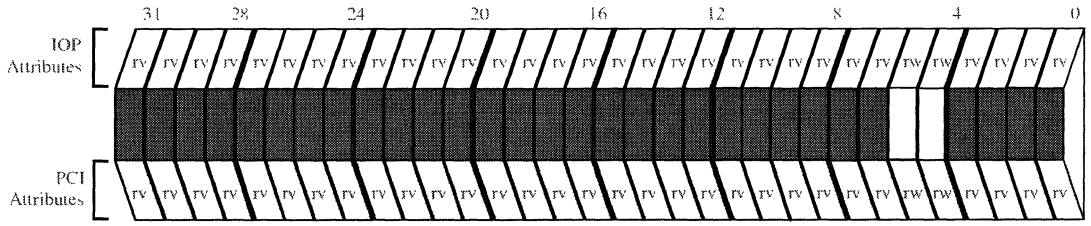
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:06	00H	Reserved
05:04	00_2	Memory Write Non-Full State - these bits define the definition of what a non-full state is for the primary ATU inbound posted memory write queue (P_IWQ). 00 - A non-full queue has 8 bytes or more free 01 - A non-full queue has 16 bytes or more free 10 - A non-full queue has 32 bytes or more free 11 - Reserved (treated like 00) These bits define when the PCI interface of the primary ATU will accept a posted memory write operation based on the number of bytes in a queue.
03:00	0H	Reserved

15.7.50 Secondary ATU Queue Control Register - SAQCR

The Secondary ATU Queue Control Register contains programmable parameters affecting operation of the secondary ATU queues.

Table 15-78. Secondary ATU Queue Control Register - SAQCR



Intel® i960® Core Local Bus Address
12B8H

PCI Configuration Address Offset
B8H - BBH

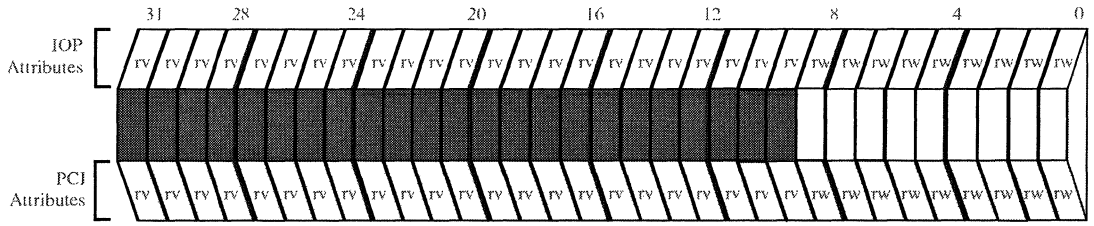
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:06	00H	Reserved
05:04	00_2	Memory Write Non-Full State - these bits define the definition of what a non-full state is for the secondary ATU inbound posted memory write queue (S_IWQ). 00 - A non-full queue has 8 bytes or more free 01 - A non-full queue has 16 bytes or more free 10 - A non-full queue has 32 bytes or more free 11 - Reserved (treated like 00) These bits define when the PCI interface of the secondary ATU will accept a posted memory write operation based on the number of bytes in a queue.
03:00	0H	Reserved

15.7.51 Primary ATU Interrupt Mask Register - PATUIMR

The Primary ATU Interrupt Mask Register contains the control bit to enable and disable interrupts generated by the primary ATU.

Table 15-79. Primary ATU Interrupt Mask Register - PATUIMR



Intel® i960® Core Local Bus Address: 12BCH

PCI Configuration Address Offset: BCH - BFH

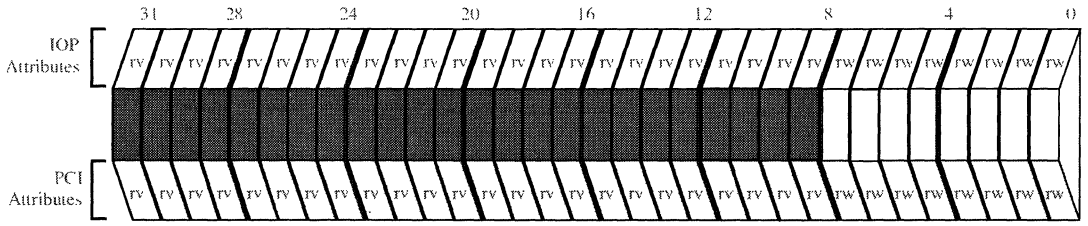
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:09	0000 00H	Reserved
08	1_2	Power State Transition Interrupt Mask - When this bit is set and the ATU Power Management Control/Status Register is written to transition the ATU Function Power State from either D0 to D3 or D3 to D0, bit 11of the PATUISR is not set.
07	0_2	PATU Detected Parity Error Interrupt Mask - When set, a parity error detected on the primary PCI bus that sets bit 15 of the PATUSR will *not* result in bit 9 of the PATUISR being set. When clear, an error that sets bit 15 of the PATUSR will result in bit 9 of the PATUISR being set.
06	0_2	PATU **P_SERR#** Asserted Interrupt Mask - When set, asserting **P_SERR#** on the primary interface resulting in bit 14 of the PATUSR being set will *not* result in bit 10 of the PATUISR being set. When clear, an error that sets bit 14 of the PATUSR will cause bit 10 of the PATUISR to be set. Note that this bit is specific to the PATU asserting **P_SERR#** and not detecting **P_SERR#** from another master.
05	0_2	PATU PCI Master Abort Interrupt Mask - When set, a master abort error resulting in bit 13 of the PATUSR being set will *not* result in bit 3 of the PATUISR being set. When clear, an error that sets bit 13 of the PATUSR will cause bit 3 of the PATUISR to be set.
04	0_2	PATU PCI Target Abort (Master) Interrupt Mask- When set, a target abort error resulting in bit 12 of the PATUSR being set will *not* result in bit 2 of the PATUISR being set. When clear, an error that sets bit 12 of the PATUSR will cause bit 2 of the PATUISR to be set.
03	0_2	PATU PCI Target Abort (Target) Interrupt Mask- When set, a target abort error resulting in bit 11 of the PATUSR being set will *not* result in bit 1 of the PATUISR being set. When clear, an error that sets bit 11 of the PATUSR will cause bit 1 of the PATUISR to be set.
02	0_2	PATU PCI Master Parity Error Interrupt Mask - When set, a parity error resulting in bit 8 of the PATUSR being set will *not* result in bit 0 of the PATUISR being set. When clear, an error that sets bit 8 of the PATUSR will cause bit 0 of the PATUISR to be set.
01	0_2	Primary ATU Inbound Error **P_SERR#** Enable - When set, the PATU will assert (if enabled through the PATUCMD) **P_SERR#** on the primary interface in response to a master abort on the internal bus during an inbound write transaction or a target abort from the memory controller (ECC Error) during an inbound write transaction. When clear, **P_SERR#** will not be asserted under the previous conditions.
00	0_2	Primary ATU ECC Target Abort Enable - When set, the PATU will perform a target abort on the primary PCI interface in response to a target abort (ECC error) from the memory controller on the internal bus. This action only occurs when during an inbound read transaction where the data phase that was target aborted on the internal bus is actually requested from the inbound read queue. When clear, the response under the same conditions is a disconnect with data (the data being up to 64 bits of 1's) on the PCI bus instead of a target abort.

15.7.52 Secondary ATU Interrupt Mask Register - SATUIMR

The Secondary ATU Interrupt Mask Register contains the control bit to enable and disable interrupts generated by the secondary ATU.

Table 15-80. Secondary ATU Interrupt Mask Register - SATUIMR



Intel® i960® Core Local Bus Address: 12C0H

PCI Configuration Address Offset: C0H - C3H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:08	0000 00H	Reserved
07	0_2	SATU Detected Parity Error Interrupt Mask - When set, a parity error detected on the secondary PCI bus that sets bit 15 of the SATUSR will *not* result in bit 9 of the SATUISR being set. When clear, an error that sets bit 15 of the SATUSR will result in bit 9 of the SATUISR being set.
06	0_2	SATU **S_SERR#** Asserted Interrupt Mask - When set, asserting **S_SERR#** on the secondary interface resulting in bit 14 of the SATUSR being set will *not* result in bit 10 of the SATUISR being set. When clear, an error that sets bit 14 of the SATUSR will cause bit 10 of the SATUISR to be set. Note that this bit is specific to the SATU asserting **S_SERR#** and not detecting **S_SERR#** from another master.
05	0_2	SATU PCI Master Abort Interrupt Mask - When set, a master abort error resulting in bit 13 of the SATUSR being set will *not* result in bit 3 of the SATUISR being set. When clear, an error that sets bit 13 of the SATUSR will cause bit 3 of the SATUISR to be set.
04	0_2	SATU PCI Target Abort (Master) Interrupt Mask- When set, a target abort error resulting in bit 12 of the SATUSR being set will *not* result in bit 2 of the SATUISR being set. When clear, an error that sets bit 12 of the SATUSR will cause bit 2 of the SATUISR to be set.
03	0_2	SATU PCI Target Abort (Target) Interrupt Mask - When set, a target abort error resulting in bit 11 of the SATUSR being set will *not* result in bit 1 of the SATUISR being set. When clear, an error that sets bit 11 of the SATUSR will cause bit 1 of the SATUISR to be set.
02	0_2	SATU PCI Master Parity Error Interrupt Mask - When set, a parity error resulting in bit 8 of the SATUSR being set will *not* result in bit 0 of the PATUISR being set. When clear, an error that sets bit 8 of the SATUSR will cause bit 0 of the SATUISR to be set.
01	0_2	Secondary ATU Inbound Error S_SERR# Enable - When set, the SATU will assert (if enabled through the SATUCMD) **S_SERR#** on the secondary interface in response to a master abort on the internal bus during an inbound write transaction or a target abort from the memory controller (ECC Error) during an inbound write transaction. When clear, **S_SERR#** will not be asserted under the previous conditions.
00	0_2	Secondary ATU ECC Target Abort Enable - When set, the SATU will perform a target abort on the secondary PCI interface in response to a target abort (ECC error) from the memory controller on the internal bus. This action only occurs during an inbound read transaction where the data phase that was target aborted on the internal bus is actually requested from the inbound read queue. When clear, the response under the same conditions is a disconnect with data (the data being up to 64 bits of 1's) on the PCI bus instead of a target abort.

Messaging Unit 16

This chapter describes the Messaging Unit (MU) of the Intel® 80303 I/O processor. The MU is closely related to the Primary Address Translation Unit (PATU) described in Chapter 15, "PCI Address Translation Unit".

16.1 Overview

The Messaging Unit provides a mechanism for data to be transferred between the PCI system and the Intel® i960® core processor and notifies the respective system of the arrival of new data through an interrupt. The MU can be used to send and receive messages.

The MU has four distinct messaging mechanisms. Each allows a host processor or external PCI agent and the i960 core processor to communicate through message passing and interrupt generation. The four mechanisms are:

- **Message Registers** — allow the i960 core processor and external PCI agents to communicate by passing messages in one of four 32-bit Message Registers. In this context, a message is any 32-bit data value. Message registers combine aspects of mailbox registers and doorbell registers. Writes to the message registers may optionally cause interrupts.
- **Doorbell Registers** — allow the i960 core processor to assert the PCI interrupt signals and allow external PCI agents to generate an interrupt to the i960 core processor.
- **Circular Queues** — support a message passing scheme that uses four circular queues.
- **Index Registers** — support a message passing scheme that uses a portion of the 80303 I/O processor local memory to implement a large set of message registers.

Each of the above is available to the system designer at the same time. No special mode selection is needed.

16.2 Theory of Operation

The MU has four independent messaging mechanisms. The four Message Registers are similar to a combination of mailbox and doorbell registers. Each holds a 32-bit value and generates an interrupt when written.

The two Doorbell Registers support software interrupts. When a bit is set in a Doorbell Register, an interrupt is generated.

The Circular Queues support a message passing scheme that uses four circular queues. The four circular queues are implemented in 80303 I/O processor local memory. Two queues are used for inbound messages and two are used for outbound messages. Interrupts may be generated when the queue is written.

The Index Registers use a portion of the 80303 I/O processor local memory to implement a large set of message registers. When one of the Index Registers is written, an interrupt is generated and the address of the register written is captured.

Interrupt status for all interrupts is recorded in the Inbound Interrupt Status Register and the Outbound Interrupt Status Register. Each interrupt generated by the Messaging Unit can be masked.

Multi-word PCI burst accesses are not supported by the Messaging Unit, with the exception of multi-word reads to the index registers. The MU terminates multi-word PCI transactions (other than index register reads) with a disconnect after the next Qword boundary, with the exception of queue ports.

All registers needed to configure and control the Messaging Unit are memory-mapped registers.

The MU uses the first 4 Kbytes of the primary inbound translation window in the Primary Address Translation Unit (PATU). This PCI address window is used for PCI transactions that access the 80303 I/O processor local memory. The PCI address of the primary inbound translation window is contained in the Primary Inbound ATU Base Address Register. See Chapter 15, "PCI Address Translation Unit" for more details on inbound ATU addressing and the PATU.

From the PCI perspective, the Messaging Unit is part of the Primary Address Translation Unit. The Messaging Unit uses the PCI configuration registers of the Primary ATU for control and status information. The Messaging Unit must observe all PCI control bits in the Primary ATU Command Register and ATU Configuration Register. The Messaging Unit reports all PCI errors in the Primary ATU Status Register. The Messaging Unit can be accessed from the 80303 I/O processor secondary PCI bus by sending the cycle through the PCI-to-PCI Bridge Unit. Refer to Chapter 14, "PCI-to-PCI Bridge Unit" for details of the correct configuration options to support this feature.

Parts of the Messaging Unit can be accessed as a 64-bit PCI device. The register interface, message registers, doorbell registers, and index registers return a P_ACK64# in response to a P_REQ64# on the primary interface. Up to 1 Qword of data can be read or written per transaction (except Index Register reads, see Section 16.6, on page 16-15). The Inbound and Outbound Queue Ports are always 32-bit addresses and the MU never asserts P_ACK64# to offsets 40H and 44H.

Figure 16-1. PCI Memory Map

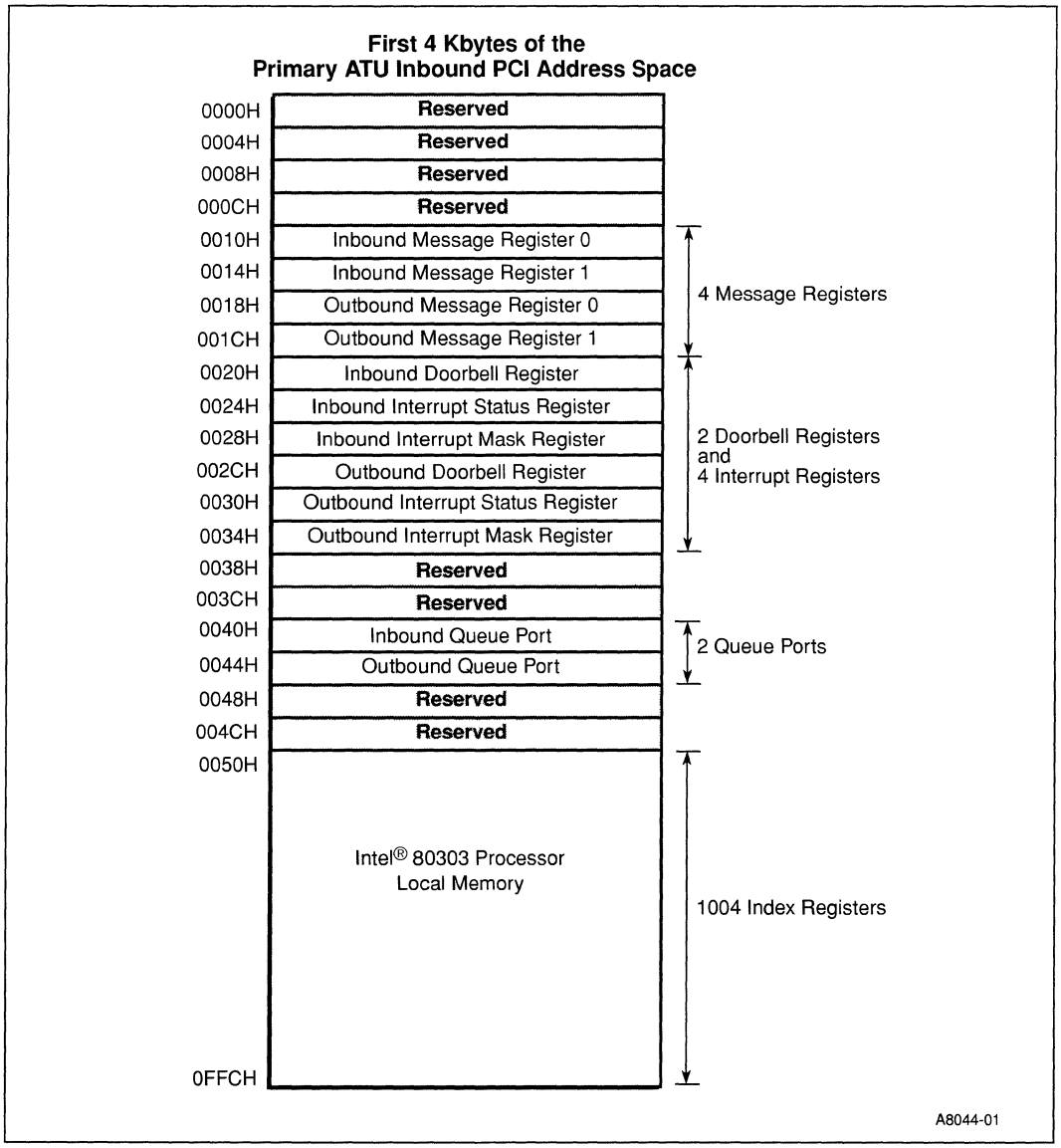


Table 16-1 provides a summary of the four messaging mechanisms used in the Messaging Unit.

Table 16-1. MU Summary

Mechanism	Quantity	Assert PCI Interrupt Signals?	Generate Intel® i960® Core Processor Interrupt?
Message Registers	2 Inbound and 2 Outbound	Optional	Optional
Doorbell Registers	1 Inbound and 1 Outbound	Optional	Optional
Circular Queues	4 Circular Queues	Under certain conditions	Under certain conditions
Index Registers	1004 32-bit Memory Locations	No	Optional

16.2.1 Transaction Ordering

From a PCI standpoint, the Messaging Unit is a piece of the primary ATU and therefore must maintain some ordering requirements against PATU transactions. Transaction ordering is achieved for the Index Registers, the Doorbell Register, and the Message Registers since these transactions are routed through the standard set of PATU read/write queues.

The Circular Queues (Inbound/Outbound Queue Port) are separate queue structures and therefore require ordering. The Inbound Post Queue (contains PCI writes) must be ordered against the inbound write queue of the PATU to allow the data that is represented by the Inbound Post interrupt to be written to local memory before the interrupt is delivered. See Table 16-2 for a summary of Messaging Unit transaction ordering.

Table 16-2. Circular Queue Ordering Requirements

Messaging Unit Feature		Transaction Ordering Mechanism
Message Registers		Through PATU Queues
Doorbell Registers		
Index Registers		
Circular Queues	Inbound Post	Ordered Against PATU Inbound Write Queue (PMW Can Not Pass Another PMW)
	Inbound Free	No Specific Hardware Ordering
	Outbound Post	
	Outbound Free	

16.3 Message Registers

Messages can be sent and received by the i960 core processor through the use of the Message Registers. When written, the Message Registers may cause an interrupt to be generated to either the i960 core processor or the PCI interrupt signals. Inbound messages are sent by the host processor and received by the 80303 I/O processor. Outbound messages are sent by the i960 core processor and received by the host processor.

The interrupt status for outbound messages is recorded in the Outbound Interrupt Status Register. Interrupt status for inbound messages is recorded in the Inbound Interrupt Status Register.

16.3.1 Outbound Messages

When an outbound message register is written by the i960 core processor, an interrupt may be generated on the P_INTA#, P_INTB#, P_INTC#, or P_INTD# interrupt pins. Which interrupt pin used is determined by the value of the ATU Interrupt Pin Register (Chapter 15, "PCI Address Translation Unit").

The PCI interrupt is recorded in the Outbound Interrupt Status Register. The interrupt causes the Outbound Message Interrupt bit to be set in the Outbound Interrupt Status Register. This is a Read/Clear bit that is set by the MU hardware and cleared by software.

The interrupt is cleared when an external PCI agent writes a value of "1" to the Outbound Message Interrupt bit in the Outbound Interrupt Status Register to clear the bit.

The interrupt may be masked by the Mask bits in the Outbound Interrupt Mask Register.

16.3.2 Inbound Messages

When an inbound message register is written by an external PCI agent, an interrupt may be generated to the i960 core processor. The interrupt may be masked by the Mask bits in the Inbound Interrupt Mask Register.

The i960 core processor interrupt is recorded in the Inbound Interrupt Status Register. The interrupt causes the Inbound Message Interrupt bit to be set in the Inbound Interrupt Status Register. This is a Read/Clear bit that is set by the MU hardware and cleared by software.

The interrupt is cleared when the i960 core processor writes a value of "1" to the Inbound Message Interrupt bit in the Inbound Interrupt Status Register.

16.4 Doorbell Registers

There are two Doorbell Registers: the Inbound Doorbell Register and the Outbound Doorbell Register. The Inbound Doorbell Register allows external PCI agents to generate interrupts to the i960 core processor. The Outbound Doorbell Register allows the i960 core processor to generate a PCI interrupt. Both Doorbell Registers may generate interrupts whenever a bit in the register is set.

16.4.1 Outbound Doorbells

When the Outbound Doorbell Register is written by the i960 core processor, an interrupt may be generated on the P_INTA#, P_INTB#, P_INTC#, or P_INTD# interrupt pins. An interrupt is generated when any of the bits in the doorbell register is written to a value of "1". Writing a value of "0" to any bit does not change the value of that bit and does not cause an interrupt to be generated. Once a bit is set in the Outbound Doorbell Register, it cannot be cleared by i960 core processor.

Which PCI interrupt pin used is determined by the value of the ATU Interrupt Pin Register (Chapter 15, "PCI Address Translation Unit").

The interrupt is recorded in the Outbound Interrupt Status Register.

The interrupt may be masked by the Mask bits in the Outbound Interrupt Mask Register. When the Mask bit is set for a particular bit, no interrupt is generated for that bit. The Outbound Interrupt Mask Register affects only the generation of the interrupt and not the values written to the Outbound Doorbell Register.

The interrupt is cleared when an external PCI agent writes a value of "1" to the bits in the Outbound Doorbell Register that are set. Writing a value of "0" to any bit does not change the value of that bit and does not clear the interrupt.

In summary, the i960 core processor generates an interrupt by setting bits in the Outbound Doorbell Register and external PCI agents clear the interrupt by also setting bits in the same register.

16.4.2 Inbound Doorbells

When the Inbound Doorbell Register is written by an external PCI agent, an interrupt may be generated to the i960 core processor. An interrupt is generated when any of the bits in the doorbell register is written to a value of "1". Writing a value of "0" to any bit does not change the value of that bit and does not cause an interrupt to be generated. Once a bit is set in the Inbound Doorbell Register, it cannot be cleared by any external PCI agent.

The interrupt is recorded in the Inbound Interrupt Status Register.

The interrupt may be masked by the Inbound Doorbell Interrupt Mask bit in the Inbound Interrupt Mask Register. When the mask bit is set for a particular bit, no interrupt is generated for that bit. The Inbound Interrupt Mask Register affects only the generation of the interrupt and not the values written to the Inbound Doorbell Register.

One bit in the Inbound Doorbell Register is reserved for an NMI interrupt.

The interrupt is cleared when the i960 core processor writes a value of "1" to the bits in the Inbound Doorbell Register that are set. Writing a value of "0" to any bit does not change the value of that bit and does not clear the interrupt.

16.5 Circular Queues

The MU implements four circular queues. There are two inbound queues and two outbound queues. In this case, inbound and outbound refer to the direction of the flow of posted messages.

Inbound messages are either:

- *posted* messages by other processors for the i960 core processor to process or
- *free* (or empty) messages that can be reused by other processors.

Outbound messages are either:

- *posted* messages by the i960 core processor for other processors to process or
- *free* (or empty) messages that can be reused by the i960 core processor.

Therefore, free inbound messages flow away from the i960 core processor and free outbound messages flow toward the i960 core processor.

The four Circular Queues are used to pass messages in the following manner. The two inbound queues are used to handle inbound messages and the two outbound queues are used to handle outbound messages. One of the inbound queues is designated the Free queue and it contains inbound free messages. The other inbound queue is designated the Post queue and it contains inbound posted messages. Similarly, one of the outbound queues is designated the Free queue and the other outbound queue is designated the Post queue. Table 16-3 contains a summary of the queues.

Table 16-3. Circular Queue Summary

Queue Name	Purpose	Action on PCI Interface
Inbound Post Queue	Queue for inbound messages from other processors waiting to be processed by the Intel® i960® core processor	Written
Inbound Free Queue	Queue for empty inbound messages from the i960 core processor available for use by other processors	Read
Outbound Post Queue	Queue for outbound messages from the i960 core processor that are being posted to the other processors	Read
Outbound Free Queue	Queue for empty outbound messages from other processors available for use by the i960 core processor	Written

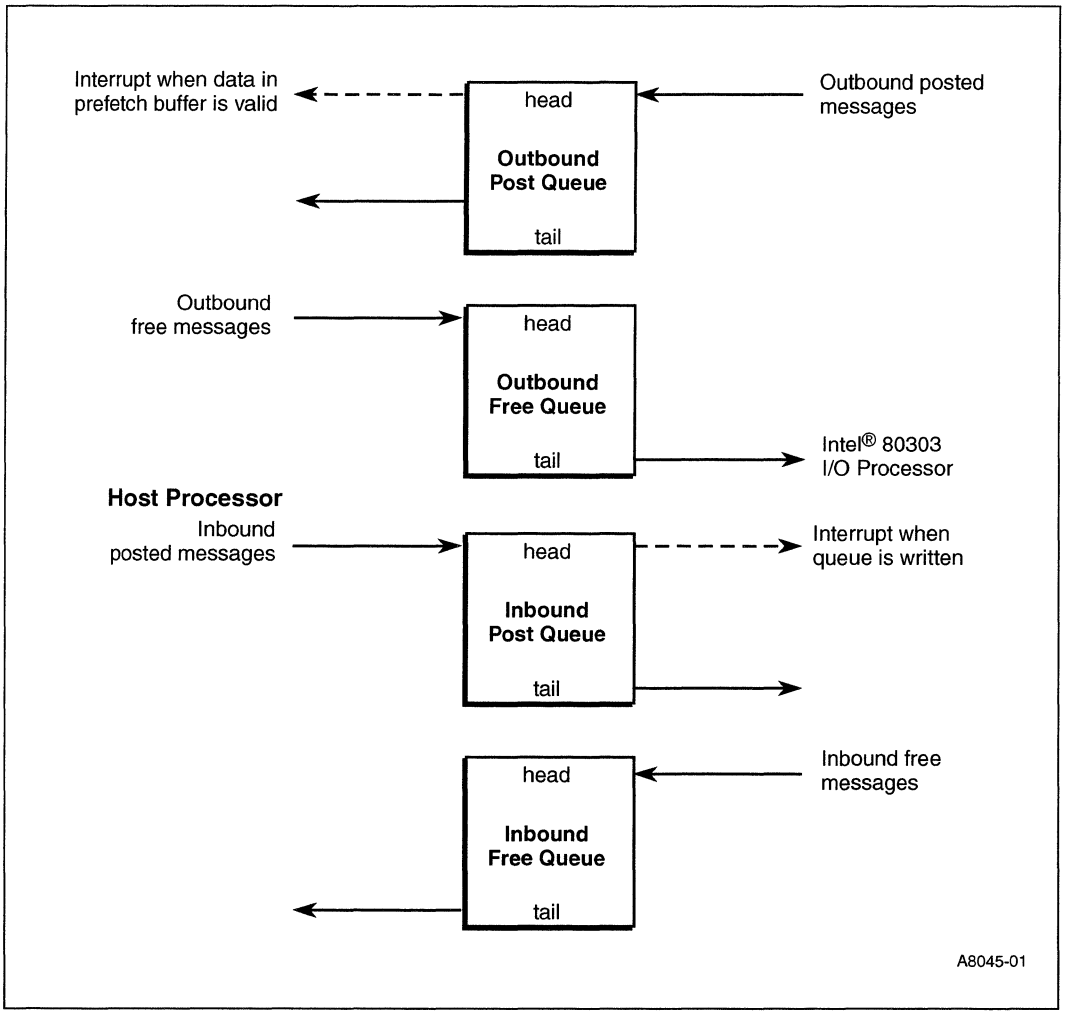
The two outbound queues allow the i960 core processor to post outbound messages in one queue and to receive free messages returning from the host processor. The i960 core processor posts outbound messages, the host processor receives the posted message and when it is finished with the message, places it back on the outbound free queue for reuse by the i960 core processor.

The two inbound queues allow the host processor to post inbound messages for the i960 core processor in one queue and to receive free messages returning from the i960 core processor. The host processor posts inbound messages, the i960 core processor receives the posted message and when it is finished with the message, places it back on the inbound free queue for reuse by the host processor.

Figure 16-2 provides an overview of the Circular Queue operation.

The circular queues are accessed by external PCI agents through two port locations in the PCI address space: Inbound Queue Port and Outbound Queue Port. The Inbound Queue Port is used by external PCI agents to read the Inbound Free Queue and write the Inbound Post Queue. The Outbound Queue Port is used by external PCI agents to read the Outbound Post Queue and write the Outbound Free Queue. Note that a PCI transaction to the inbound or outbound queue ports with null byte enables (P_C/BE[3:0]# = 1111_2) does not cause the MU hardware to increment the queue pointers. This is treated as if the PCI transaction did not occur. The Inbound and Outbound Queue Ports never respond with **P_ACK64#** on the primary PCI interface.

Figure 16-2. Overview of Circular Queue Operation



The data storage for the circular queues must be provided by the 80303 I/O processor local memory. The base address of the circular queues is contained in the Queue Base Address Register (Section 16.8.10, "Queue Base Address Register - QBAR" on page 16-27). Each entry in the queue is a 32-bit data value. Each read from or write to the queue may access only one queue entry. Multi-word accesses to the circular queues are not allowed. Sub-word accesses are promoted to 32-bit word accesses.

Each circular queue has a head pointer and a tail pointer. The pointers are offsets from the Queue Base Address. Writes to a queue occur at the head of the queue and reads occur from the tail. The head and tail pointers are incremented by either the i960 core processor or the Messaging Unit hardware. Which unit maintains the pointer is determined by the writer of the queue. More details about the pointers are given in the queue descriptions below. The pointers are incremented after the queue access. Both pointers wrap around to the first address of the circular queue when they reach the circular queue size.

The Messaging Unit generates an interrupt to the i960 core processor or generate a PCI interrupt under certain conditions. In general, when a Post queue is written, an interrupt is generated to notify the receiver that a message was posted.

The size of each circular queue can range from 4K entries (16 Kbytes) to 64K entries (256 Kbytes). All four queues must be the same size and may be contiguous. Therefore, the total amount of local memory needed by the circular queues ranges from 64 Kbytes to 1 Mbyte. The Queue size is determined by the Queue Size field in the MU Configuration Register.

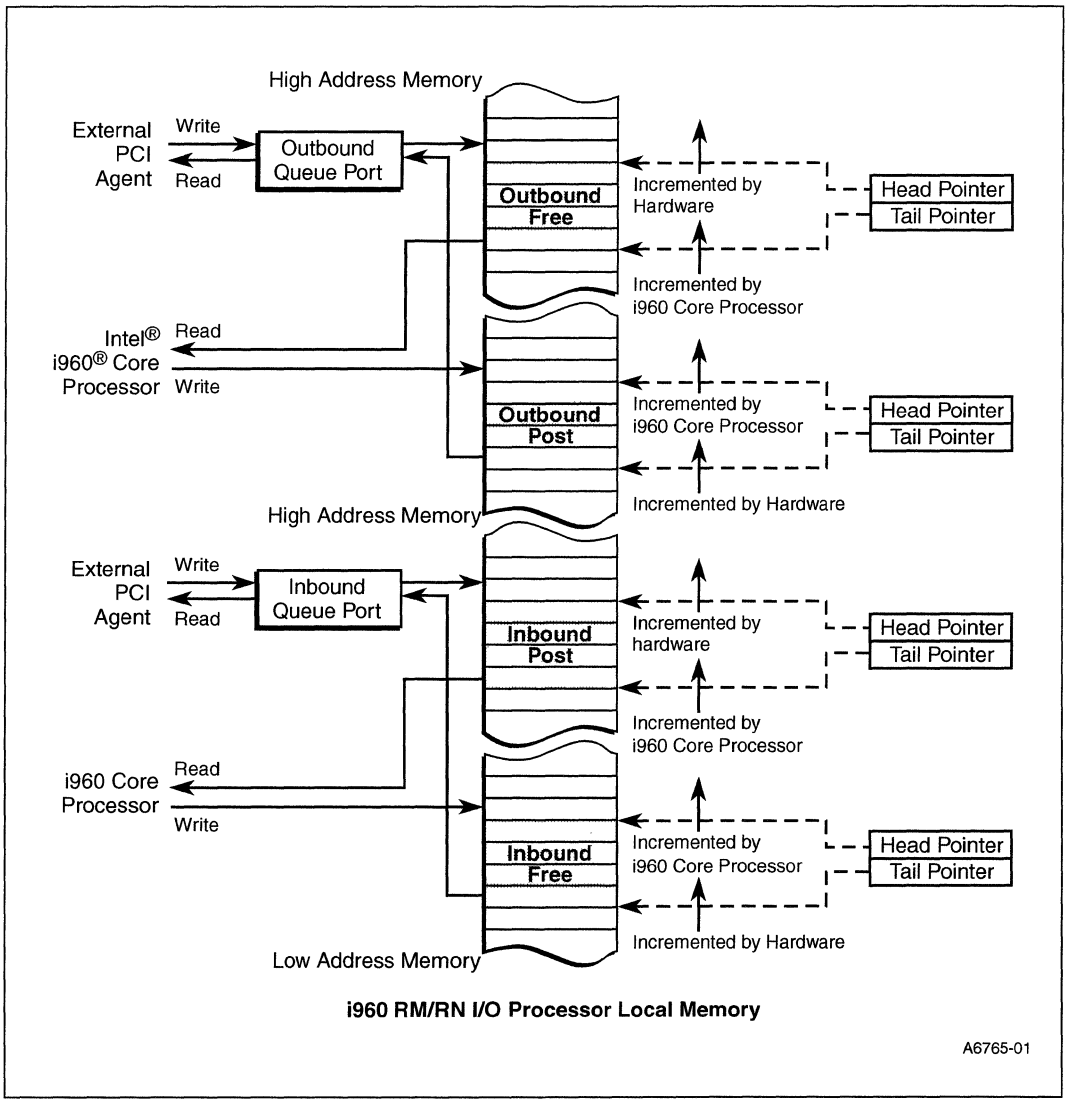
There is one base address for all four queues. It is stored in the Queue Base Address Register (QBAR). The starting addresses of each queue is based on the Queue Base Address and the Queue Size field. Table 16-4 shows an example of how the circular queues should be set up based on the *Intelligent I/O (I_2O) Architecture Specification*. Other ordering of the circular queues is possible, however.

Table 16-4. Queue Starting Addresses

Queue	Starting Address
Inbound Free Queue	QBAR
Inbound Post Queue	QBAR + Queue Size
Outbound Post Queue	QBAR + 2 * Queue Size
Outbound Free Queue	QBAR + 3 * Queue Size

Figure 16-3 provides a more detailed diagram of the usage of the Circular Queues.

Figure 16-3. Circular Queue Operation



16.5.1 Inbound Free Queue

The Inbound Free Queue holds free inbound messages placed there by the i960 core processor for other processors to use. This queue is read from the queue tail by external PCI agents. It is written to the queue head by the i960 core processor. The tail pointer is maintained by the MU hardware. The head pointer is maintained by the i960 core processor.

For a PCI read transaction that accesses the Inbound Queue Port, the MU attempts to read the data at the local memory address in the Inbound Free Tail Pointer:

- When the queue is not empty (head and tail pointers are not equal), or full (head and tail pointers are equal but the head pointer was last written by software), the data is returned.
- When the queue is empty (head and tail pointers are equal), the value of -1 (FFFF.FFFFH) is returned.
- When the queue was not empty and the MU succeeded in returning the data at the tail, the MU hardware must increment the value in the Inbound Free Tail Pointer Register.

To reduce latency for the PCI read access, the MU implements a prefetch mechanism to anticipate accesses to the Inbound Free Queue. The MU hardware prefetches the data at the tail of the Inbound Free Queue and loads it into an internal prefetch register. When the PCI read access occurs, the data is read directly from the prefetch register.

The prefetch mechanism loads a value of -1 (FFFF.FFFFH) into the prefetch register when the head and tail pointers are equal and the queue is empty. To update the prefetch register when messages are added to the queue and it becomes non-empty, the prefetch mechanism automatically starts a prefetch when the prefetch register contains FFFF.FFFFH and the Inbound Free Head Pointer Register is written. The i960 core processor needs to update the Inbound Free Head Pointer Register when it adds messages to the queue.

A prefetch must appear atomic from the perspective of the external PCI agent. When a prefetch is started, any PCI transaction that attempts to access the Inbound Free Queue is signalled a Retry until the prefetch is completed.

The i960 core processor may place messages in the Inbound Free Queue by writing the data to the local memory location pointed to by the Inbound Free Head Pointer Register. The processor must then increment the Inbound Free Head Pointer Register.

16.5.2 Inbound Post Queue

The Inbound Post Queue holds posted messages placed there by other processors for the i960 core processor to process. This queue is read from the queue tail by the i960 core processor. It is written to the queue head by external PCI agents. The tail pointer is maintained by the i960 core processor. The head pointer is maintained by the MU hardware.

For a PCI write transaction that accesses the Inbound Queue Port, the MU writes the data to the local memory location address in the Inbound Post Head Pointer Register.

When the data written to the Inbound Queue Port is written to local memory, the MU hardware increments the Inbound Post Head Pointer Register.

An i960 core processor interrupt may be generated when the Inbound Post Queue is written. The Inbound Post Queue Interrupt bit in the Inbound Interrupt Status Register indicates the interrupt status. The interrupt is cleared when the Inbound Post Queue Interrupt bit is cleared. The interrupt can be masked by the Inbound Interrupt Mask Register. When the Inbound Post Queue reaches a full state (head pointer equals tail pointer), the PCI interface retries all further writes until software increments the tail pointer or the Inbound Post Queue Interrupt bit is cleared. To prevent an indefinite retry, software must be aware of the state of the Inbound Post Queue Interrupt Mask bit to guarantee that the full condition is recognized by the core processor. In addition, to guarantee that the queue is not overwritten, software must remove data from the tail of the queue before clearing the interrupt (and incrementing the tail pointer).

From the time that the PCI write transaction is received until the data is written in local memory and the Inbound Post Head Pointer Register is incremented, any PCI transaction that attempts to access the Inbound Post Queue Port is signalled a Retry.

The i960 core processor may read messages from the Inbound Post Queue by reading the data from the local memory location pointed to by the Inbound Post Tail Pointer Register. The i960 core processor must then increment the Inbound Post Tail Pointer Register. When the Inbound Post Queue is full, the hardware retries any PCI writes until a slot in the queue becomes available, by the i960 core processor either clearing the inbound post queue interrupt or incrementing the tail pointer. When the head pointer and tail pointer become equal, software must clear the inbound post queue interrupt bit to avoid indefinite retries by the MU.

16.5.3 Outbound Post Queue

The Outbound Post Queue holds outbound posted messages placed there by the i960 core processor for other processors to process. This queue is read from the queue tail by external PCI agents. It is written to the queue head by the i960 core processor. The tail pointer is maintained by the MU hardware. The head pointer is maintained by the i960 core processor.

For a PCI read transaction that accesses the Outbound Queue Port, the MU attempts to read the data at the local memory address in the Outbound Post Tail Pointer Register:

- When the queue is not empty (head and tail pointers are not equal), or full (head and tail pointers are equal but the head pointer was last written by software), the data is returned.
- When the queue is empty (head and tail pointers are equal), the value of -1 (FFFF.FFFFH) is returned.
- When the queue was not empty and the MU succeeded in returning the data at the tail, the MU hardware must increment the value in the Outbound Post Tail Pointer Register.

To reduce latency for the PCI read access, the MU implements a prefetch mechanism to anticipate accesses to the Outbound Post Queue. The MU hardware prefetches the data at the tail of the Outbound Post Queue and load it into an internal prefetch register. When the PCI read access occurs, the data is read directly from the prefetch register.

The prefetch mechanism loads a value of -1 (FFFF.FFFFH) into the prefetch register when the head and tail pointers are equal and the queue is empty. To update the prefetch register when messages are added to the queue and it becomes non-empty, the prefetch mechanism automatically starts a prefetch when the prefetch register contains FFFF.FFFFH and the Outbound Post Head Pointer Register is written. The i960 core processor needs to update the Outbound Post Head Pointer Register when it adds messages to the queue.

A prefetch must appear atomic from the perspective of the external PCI agent. When a prefetch is started, any PCI transaction that attempts to access the Outbound Post Queue is signalled a Retry until the prefetch is completed.

A PCI interrupt may be generated when data in the prefetch buffer is valid. When the prefetch queue is clear, no interrupt is generated. The Outbound Post Queue Interrupt bit in the Outbound Interrupt Status Register indicates the status of the prefetch buffer data and therefore the interrupt status. The interrupt is cleared when any prefetched data is read from the Outbound Queue Port. The interrupt can be masked by the Outbound Interrupt Mask Register.

The i960 core processor may place messages in the Outbound Post Queue by writing the data to the local memory address in the Outbound Post Head Pointer Register. The processor must then increment the Outbound Post Head Pointer Register.

16.5.4 Outbound Free Queue

The Outbound Free Queue holds free messages placed there by other processors for the i960 core processor to use. This queue is read from the queue tail by the i960 core processor. It is written to the queue head by external PCI agents. The tail pointer is maintained by the i960 core processor. The head pointer is maintained by the MU hardware.

For a PCI write transaction that accesses the Outbound Queue Port, the MU writes the data to the local memory address in the Outbound Free Head Pointer Register. When the data written to the Outbound Queue Port is written to local memory, the MU hardware increments the Outbound Free Head Pointer Register.

When the head pointer and the tail pointer become equal and the queue is full, the MU may signal an NMI interrupt to the i960 core processor to register the queue full condition. This interrupt is recorded in the Inbound Interrupt Status Register. The **NMI#** interrupt is cleared and the Outbound Free Queue accepts writes when the Outbound Free Queue Full Interrupt bit is cleared and not by writing to the head or tail pointers. The interrupt can be masked by the Inbound Interrupt Mask Register. To prevent an indefinite retry, software must be aware of the state of the Outbound Free Queue Interrupt Mask bit to guarantee that the full condition is recognized by the core processor.

From the time that a PCI write transaction is received until the data is written in local memory and the Outbound Free Head Pointer Register is incremented, any PCI transaction that attempts to access the Outbound Free Queue Port is signalled a retry.

The i960 core processor may read messages from the Outbound Free Queue by reading the data from the local memory address in the Outbound Free Tail Pointer Register. The processor must then increment the Outbound Free Tail Pointer Register. When the Outbound Free Queue is full, the hardware must retry any PCI writes until a slot in the queue becomes available.

Table 16-5. Circular Queue Summary

Queue Name	PCI Port	Generate PCI Interrupt?	Generate i960® Core Processor Interrupt?	Head Pointer maintained by	Tail Pointer maintained by
Inbound Post Queue	Inbound Queue Port	No	Yes, when queue is written	MU hardware	i960 core processor
Inbound Free Queue		No	No	i960 core processor	MU hardware
Outbound Post Queue	Outbound Queue Port	Yes, when data in prefetch buffer is valid	No	i960 core processor	MU hardware
Outbound Free Queue		No	Yes, when the queue is full	MU hardware	i960 core processor

16.6 Index Registers

The Index Registers are a set of 1004 registers that, when written by an external PCI agent, can generate an interrupt to the i960 core processor. These registers are for inbound messages only. The interrupt is recorded in the Inbound Interrupt Status Register.

The storage for the Index Registers is allocated from the 80303 I/O processor local memory. PCI write accesses to the Index Registers write the data to local memory. PCI read accesses to the Index Registers read the data from local memory. The local memory used for the Index Registers ranges from Primary Inbound ATU Translate Value Register + 050H to Primary Inbound ATU Translate Value Register + FFFH. Chapter 15, "PCI Address Translation Unit" describes how PCI addresses are translated to local memory addresses.

The address of the first write access is stored in the Index Address Register. This register is written during the earliest write access and provides a means to determine which Index Register was written. Once updated by the MU, the Index Address Register is not updated until the Index Register Interrupt bit in the Inbound Interrupt Status Register is cleared. When the interrupt is cleared, the Index Address Register is re-enabled and stores the address of the next Index Register write access.

Writes by the i960 core processor to the local memory used by the Index Registers does not cause an interrupt and does not update the Index Address Register.

The index registers can be accessed with multi-word reads and single quad-word aligned writes.

16.7 Messaging Unit Error Conditions

The Messaging Unit, like the Primary ATU, encounters error conditions on the PCI interface as well as the internal bus interface. As a PCI target, all PCI errors (parity and aborts) are captured and recorded in the Primary ATU Status Register and can be masked using the PATU mechanisms. Refer to Chapter 15, "PCI Address Translation Unit" for further details.

16.8 Register Definitions

The following registers are located in the primary PCI address space and in the Peripheral Memory-Mapped Register (PMMR) address space. They are accessible through primary PCI bus transactions and through i960 core processor internal bus accesses. In the primary PCI address space, they are mapped into the first 80 bytes of the primary inbound address window of the Primary ATU.

- Inbound Message 0 Register
- Inbound Message 1 Register
- Outbound Message 0 Register
- Outbound Message 1 Register
- Inbound Doorbell Register
- Inbound Interrupt Status Register
- Inbound Interrupt Mask Register
- Outbound Doorbell Register
- Outbound Interrupt Status Register
- Outbound Interrupt Mask Register

The following registers are located in the Peripheral Memory-Mapped Register (PMMR) address space as described in Appendix C, "Peripheral Memory-Mapped Registers".

- MU Configuration Register
- Queue Base Address Register
- Inbound Free Head Pointer Register
- Inbound Free Tail Pointer Register
- Inbound Post Head Pointer Register
- Inbound Post Tail Pointer Register
- Outbound Free Head Pointer Register
- Outbound Free Tail Pointer Register
- Outbound Post Head Pointer Register
- Outbound Post Tail Pointer Register
- Index Address Register

Reading or writing a register that is reserved is undefined.

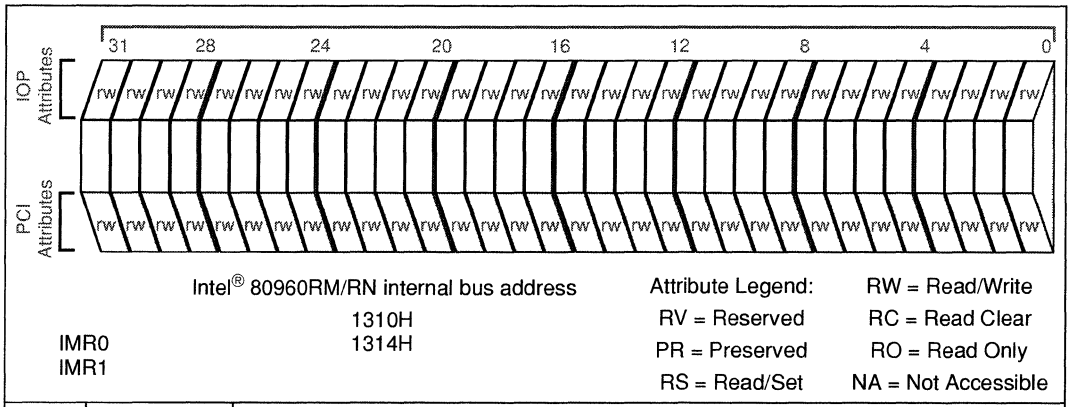
Table 16-6. Message Unit Register Table

Internal Bus Address	Section, Register Name - Acronym (Page)
1310H	Section 16.8.1, "Inbound Message Register - IMRx" on page 16-18
1314H	Section 16.8.1, "Inbound Message Register - IMRx" on page 16-18
1318H	Section 16.8.2, "Outbound Message Register - OMRx" on page 16-19
131CH	Section 16.8.2, "Outbound Message Register - OMRx" on page 16-19
1320H	Section 16.8.3, "Inbound Doorbell Register - IDR" on page 16-20
1324H	Section 16.8.4, "Inbound Interrupt Status Register - IISR" on page 16-21
1328H	Section 16.8.5, "Inbound Interrupt Mask Register - IIMR" on page 16-22
132CH	Section 16.8.6, "Outbound Doorbell Register - ODR" on page 16-23
1330H	Section 16.8.7, "Outbound Interrupt Status Register - OISR" on page 16-24
1334H	Section 16.8.8, "Outbound Interrupt Mask Register - OIMR" on page 16-25
1350H	Section 16.8.9, "MU Configuration Register - MUCR" on page 16-26
1354H	Section 16.8.10, "Queue Base Address Register - QBAR" on page 16-27
1360H	Section 16.8.11, "Inbound Free Head Pointer Register - IFHPR" on page 16-28
1364H	Section 16.8.12, "Inbound Free Tail Pointer Register - IFTPR" on page 16-29
1368H	Section 16.8.13, "Inbound Post Head Pointer Register - IPHPR" on page 16-30
136CH	Section 16.8.14, "Inbound Post Tail Pointer Register - IPTPR" on page 16-31
1370H	Section 16.8.15, "Outbound Free Head Pointer Register - OFHPR" on page 16-32
1374H	Section 16.8.16, "Outbound Free Tail Pointer Register - OFTPR" on page 16-33
1378H	Section 16.8.17, "Outbound Post Head Pointer Register - OPHPR" on page 16-34
137CH	Section 16.8.18, "Outbound Post Tail Pointer Register - OPTPR" on page 16-35
1380H	Section 16.8.19, "Index Address Register - IAR" on page 16-36

16.8.1 Inbound Message Register - IMRx

There are two Inbound Message Registers: IMR0 and IMR1. When the IMR register is written, an interrupt to the i960 core processor may be generated. The interrupt is recorded in the Inbound Interrupt Status Register and may be masked by the Inbound Message Interrupt Mask bit in the Inbound Interrupt Mask Register.

Table 16-7. Inbound Message Register - IMRx

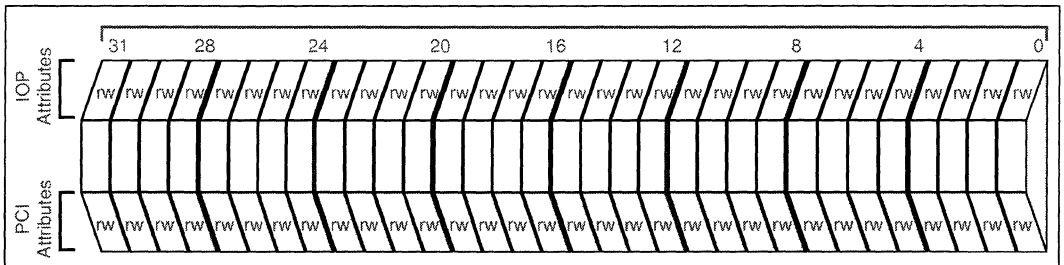


Bit	Default	Description
31:00	00000000H	Inbound Message - This is a 32-bit message written by an external PCI agent. When written, an interrupt to the i960 core processor may be generated.

16.8.2 Outbound Message Register - OMRx

There are two Outbound Message Registers: OMR0 and OMR1. When the OMR register is written, a PCI interrupt may be generated. The interrupt is recorded in the Outbound Interrupt Status Register and may be masked by the Outbound Message Interrupt Mask bit in the Outbound Interrupt Mask Register.

Table 16-8. Outbound Message Register - OMRx



Intel® 80960RM/RN internal bus address

OMR0 1318H
OMR1 131CH

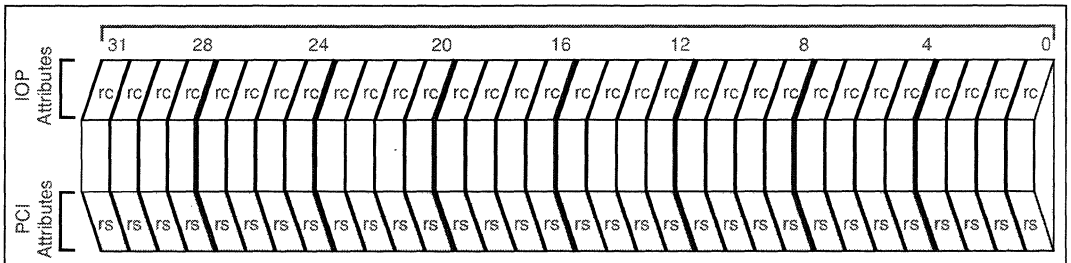
Attribute Legend:
RW = Read/Write
RV = Reserved
RC = Read Clear
PR = Preserved
RO = Read Only
RS = Read/Set
NA = Not Accessible

Bit	Default	Description
31:00	00000000H	Outbound Message - This is 32-bit message written by the i960 core processor. When written, an interrupt may be generated on the PCI Interrupt pin determined by the ATU Interrupt Pin Register.

16.8.3 Inbound Doorbell Register - IDR

The Inbound Doorbell Register (IDR) is used to generate interrupts to the i960 core processor. Bit 31 is reserved for generating an NMI interrupt. When bit 31 is set, an NMI interrupt may be generated to the i960 core processor. All other bits, when set, cause the XINT7 interrupt line of the i960 core processor to be asserted, when the interrupt is not masked by the Inbound Doorbell Interrupt Mask bit in the Inbound Interrupt Mask Register. The bits in the IDR register can only be set by an external PCI agent and can only be cleared by the i960 core processor.

Table 16-9. Inbound Doorbell Register - IDR



IDR

Intel® 80960RM/RN internal bus address
1320H

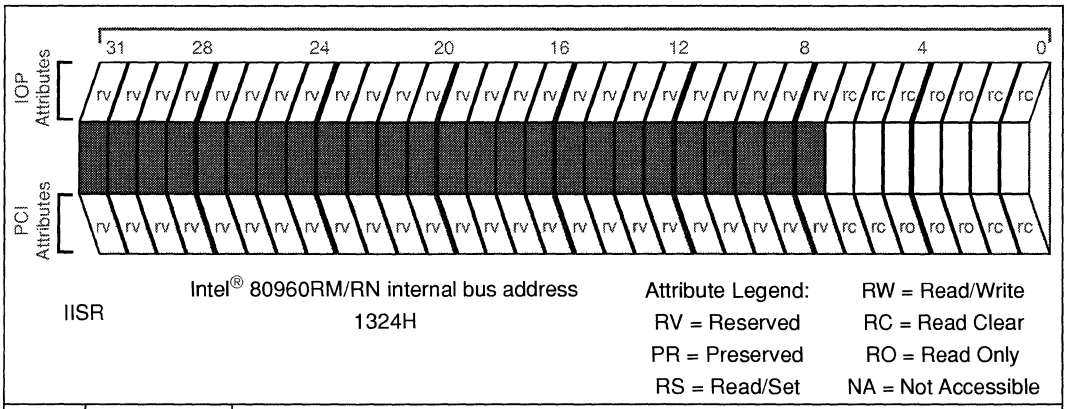
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31	0_2	NMI Interrupt - Generate an NMI Interrupt to the i960 core processor.
30:00	00000000H	XINT7 Interrupt - When any bit is set, generate an XINT7 interrupt to the i960 core processor. When all bits are clear, do not generate an XINT7 interrupt.

16.8.4 Inbound Interrupt Status Register - IISR

The Inbound Interrupt Status Register (IISR) contains hardware interrupt status. It records the status of i960 core processor interrupts generated by the Message Registers, Doorbell Registers, and the Circular Queues. All interrupts are routed to the XINT7 interrupt input of the i960 core processor, except for the NMI Doorbell Interrupt and the Outbound Free Queue Full interrupt; these two are routed to the NMI interrupt input. The generation of interrupts recorded in the Inbound Interrupt Status Register may be masked by setting the corresponding bit in the Inbound Interrupt Mask Register. Some bits in this register are Read Only. For those bits, the interrupt must be cleared through another register.

Table 16-10. Inbound Interrupt Status Register - IISR



IISR

Intel® 80960RM/RN internal bus address
1324H

Attribute Legend:
RW = Read/Write
RV = Reserved
RC = Read Clear
PR = Preserved
RO = Read Only
RS = Read/Set
NA = Not Accessible

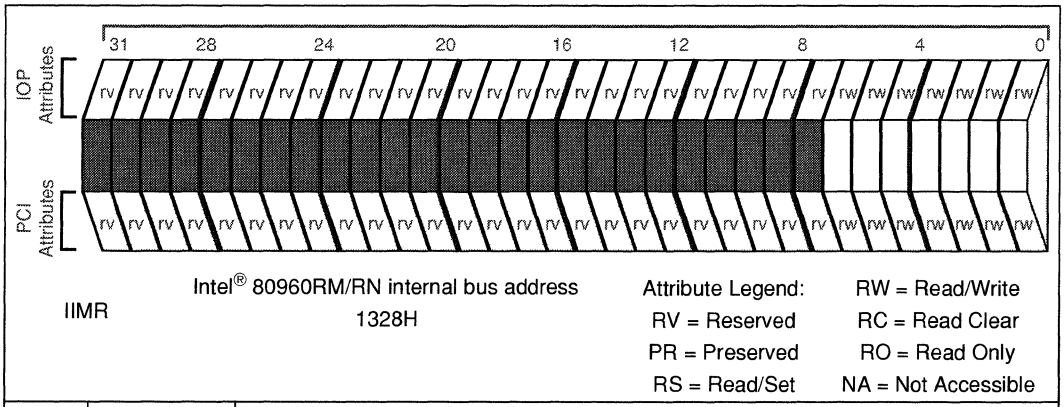
Bit	Default	Description
31:07	0000000H 0_2	Reserved
06	0_2	Index Register Interrupt - This bit is set by the MU hardware when an Index Register is written after a PCI transaction.
05	0_2	Outbound Free Queue Full Interrupt - This bit is set when the Outbound Free Head Pointer becomes equal to the Tail Pointer and the queue is full. An NMI interrupt is generated for this condition.
04	0_2	Inbound Post Queue Interrupt - This bit is set by the MU hardware when the Inbound Post Queue has been written.
03	0_2	NMI Doorbell Interrupt - This bit is set when the NMI Interrupt of the Inbound Doorbell Register is set. To clear this bit (and the interrupt), the NMI Interrupt bit of the Inbound Doorbell Register must be clear.
02	0_2	Inbound Doorbell Interrupt - This bit is set when at least one XINT7 Interrupt bit in the Inbound Doorbell Register is set. To clear this bit (and the interrupt), the XINT7 Interrupt bits in the Inbound Doorbell Register must all be clear.
01	0_2	Inbound Message 1 Interrupt - This bit is set by the MU hardware when the Inbound Message 1 Register has been written.
00	0_2	Inbound Message 0 Interrupt - This bit is set by the MU hardware when the Inbound Message 0 Register has been written.

16.8.5 Inbound Interrupt Mask Register - IIMR

The Inbound Interrupt Mask Register (IIMR) provides the ability to mask i960 core processor interrupts generated by the Messaging Unit. Each bit in the Mask register corresponds to an interrupt bit in the Inbound Interrupt Status Register.

Setting or clearing bits in this register does not affect the Inbound Interrupt Status Register. They only affect the generation of the i960 core processor interrupt.

Table 16-11. Inbound Interrupt Mask Register - IIMR



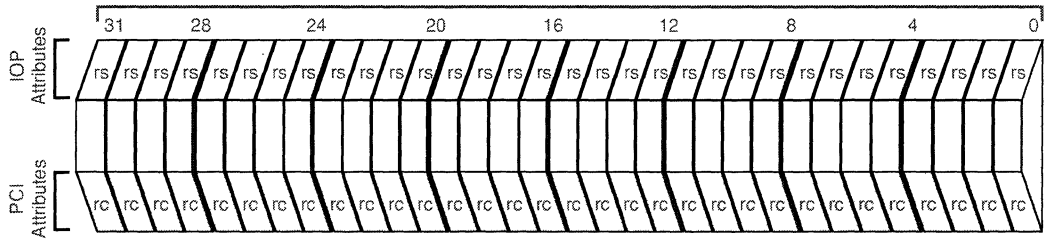
Bit	Default	Description
31:07	000000H 0_2	Reserved
06	0_2	Index Register Interrupt Mask - When set, this bit masks the interrupt generated by the MU hardware when an Index Register has been written after a PCI transaction.
05	0_2	Outbound Free Queue Full Interrupt Mask - When set, this bit masks the NMI interrupt generated when the Outbound Free Head Pointer becomes equal to the Tail Pointer and the queue is full.
04	0_2	Inbound Post Queue Interrupt Mask - When set, this bit masks the interrupt generated by the MU hardware when the Inbound Post Queue has been written.
03	0_2	NMI Doorbell Interrupt Mask - When set, this bit masks the NMI Interrupt when the NMI Interrupt bit of the Inbound Doorbell Register is set.
02	0_2	Inbound Doorbell Interrupt Mask - When set, this bit masks the interrupt generated when at least one XINT7 Interrupt bit in the Inbound Doorbell Register is set.
01	0_2	Inbound Message 1 Interrupt Mask - When set, this bit masks the Inbound Message 1 Interrupt generated by a write to the Inbound Message 1 Register.
00	0_2	Inbound Message 0 Interrupt Mask - When set, this bit masks the Inbound Message 0 Interrupt generated by a write to the Inbound Message 0 Register.

16.8.6 Outbound Doorbell Register - ODR

The Outbound Doorbell Register (ODR) allows software interrupt generation. It allows the i960 core processor to generate PCI interrupts to the host processor by writing to the Software Interrupt bits or to a specific PCI interrupt bit. The generation of PCI interrupts through the Outbound Doorbell Register may be masked by setting the Outbound Doorbell Interrupt Mask bit in the Outbound Interrupt Mask Register.

The Software Interrupt bits in this register can only be set by the i960 core processor and can only be cleared by an external PCI agent.

Table 16-12. Outbound Doorbell Register - ODR



ODR

Intel® 80960RM/RN internal bus address
132CH

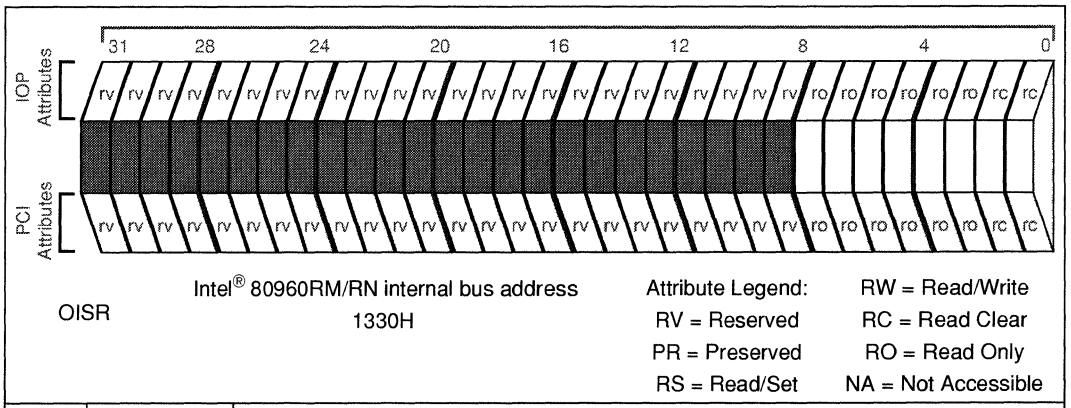
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31	0_2	PCI Interrupt D - When set, this bit causes the P_INTD# signal to be asserted. When this bit is cleared, the P_INTD# signal is deasserted.
30	0_2	PCI Interrupt C- When set, this bit causes the P_INTC# signal to be asserted. When this bit is cleared, the P_INTC# signal is deasserted.
29	0_2	PCI Interrupt B- When set, this bit causes the P_INTB# signal to be asserted. When this bit is cleared, the P_INTB# signal is deasserted.
28	0_2	PCI Interrupt A- When set, this bit causes the P_INTA# signal to be asserted. When this bit is cleared, the P_INTA# signal is deasserted.
27:00	000000H	Software Interrupt - When any bit is set, generate a PCI interrupt. The PCI interrupt pin used is determined by the ATU Interrupt Pin Register. When all bits are clear, do not generate a PCI interrupt.

16.8.7 Outbound Interrupt Status Register - OISR

The Outbound Interrupt Status Register (OISR) contains hardware interrupt status. It records the status of PCI interrupts generated by the Message Registers, Doorbell Registers, and the Circular Queues. The generation of PCI interrupts recorded in the Outbound Interrupt Status Register may be masked by setting the corresponding bit in the Outbound Interrupt Mask Register. Some bits in this register are Read Only. For those bits, the interrupt must be cleared through another register.

Table 16-13. Outbound Interrupt Status Register - OISR



OISR

Intel® 80960RM/RN internal bus address
1330H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

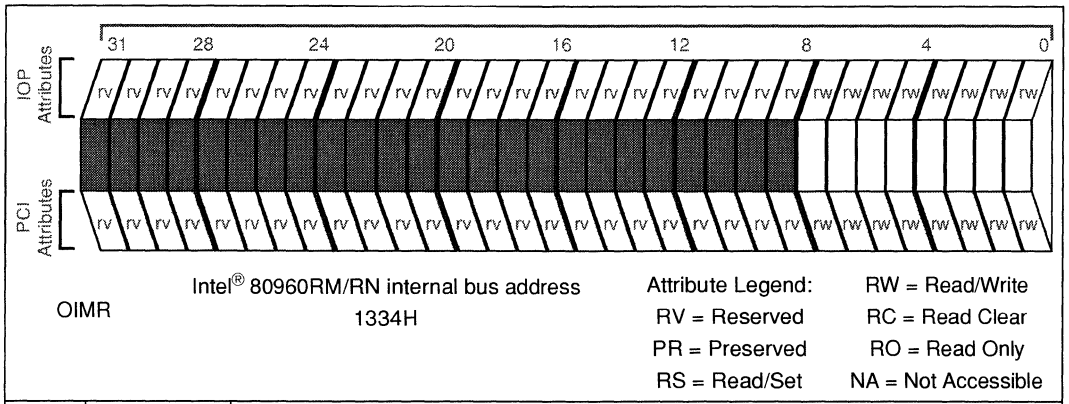
Bit	Default	Description
31:08	000000H	Reserved
07	0_2	PCI Interrupt D - This bit is set when the PCI Interrupt D bit is set in the Outbound Doorbell Register. To clear this bit (and the interrupt), the PCI Interrupt D bit must be cleared.
06	0_2	PCI Interrupt C - This bit is set when the PCI Interrupt C bit is set in the Outbound Doorbell Register. To clear this bit (and the interrupt), the PCI Interrupt C bit must be cleared.
05	0_2	PCI Interrupt B - This bit is set when the PCI Interrupt B bit is set in the Outbound Doorbell Register. To clear this bit (and the interrupt), the PCI Interrupt B bit must be cleared.
04	0_2	PCI Interrupt A - This bit is set when the PCI Interrupt A bit is set in the Outbound Doorbell Register. To clear this bit (and the interrupt), the PCI Interrupt A bit must be cleared.
03	0_2	Outbound Post Queue Interrupt - This bit is set when data in the prefetch buffer is valid. This bit is cleared when any prefetch data has been read from the Outbound Queue Port.
02	0_2	Outbound Doorbell Interrupt - This bit is set when at least one Software Interrupt bit in the Outbound Doorbell Register is set. To clear this bit (and the interrupt), the Software Interrupt bits in the Outbound Doorbell Register must all be clear.
01	0_2	Outbound Message 1 Interrupt - This bit is set by the MU when the Outbound Message 1 Register is written. Clearing this bit clears the interrupt.
00	0_2	Outbound Message 0 Interrupt - This bit is set by the MU when the Outbound Message 0 Register is written. Clearing this bit clears the interrupt.

16.8.8 Outbound Interrupt Mask Register - OIMR

The Outbound Interrupt Mask Register (OIMR) provides the ability to mask outbound PCI interrupts generated by the Messaging Unit. Each bit in the mask register corresponds to a hardware interrupt bit in the Outbound Interrupt Status Register. When the bit is set, the PCI interrupt is not generated. When the bit is clear, the interrupt is allowed to be generated.

Setting or clearing bits in this register does not affect the Outbound Interrupt Status Register. They only affect the generation of the PCI interrupt.

Table 16-14. Outbound Interrupt Mask Register - OIMR

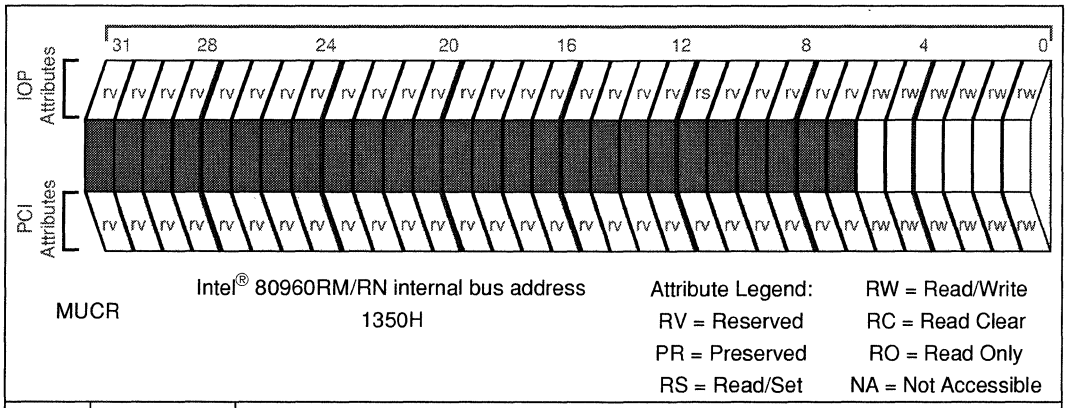


Bit	Default	Description
31:08	000000H	Reserved
07	0_2	PCI Interrupt D Mask - When set, this bit masks the PCI Interrupt D signal when the PCI Interrupt D bit in the in the Outbound Doorbell Register is set.
06	0_2	PCI Interrupt C Mask - When set, this bit masks the PCI Interrupt C signal when the PCI Interrupt C bit in the in the Outbound Doorbell Register is set.
05	0_2	PCI Interrupt B Mask - When set, this bit masks the PCI Interrupt B signal when the PCI Interrupt B bit in the in the Outbound Doorbell Register is set.
04	0_2	PCI Interrupt A Mask - When set, this bit masks the PCI Interrupt A signal when the PCI Interrupt A bit in the in the Outbound Doorbell Register is set.
03	0_2	Outbound Post Queue Interrupt Mask - When set, this bit masks the PCI interrupt generated when data in the prefetch buffer is valid.
02	0_2	Outbound Doorbell Interrupt Mask - When set, this bit masks the Software Interrupt generated by the Outbound Doorbell Register.
01	0_2	Outbound Message 1 Interrupt Mask - When set, this bit masks the Outbound Message 1 Interrupt generated by a write to the Outbound Message 1 Register.
00	0_2	Outbound Message 0 Interrupt Mask- When set, this bit masks the Outbound Message 0 Interrupt generated by a write to the Outbound Message 0 Register.

16.8.9 MU Configuration Register - MUCR

The MU Configuration Register (MUCR) contains the Circular Queue Enable bit and the size of one Circular Queue. The Circular Queue Enable bit enables or disables the Circular Queues. The Circular Queues are disabled at reset to allow the software to initialize the head and tail pointer registers before any PCI accesses to the Queue Ports. Each of the four Circular Queues may range from 4K entries (16 Kbytes) to 64K entries (256 Kbytes).

Table 16-15. MU Configuration Register - MUCR



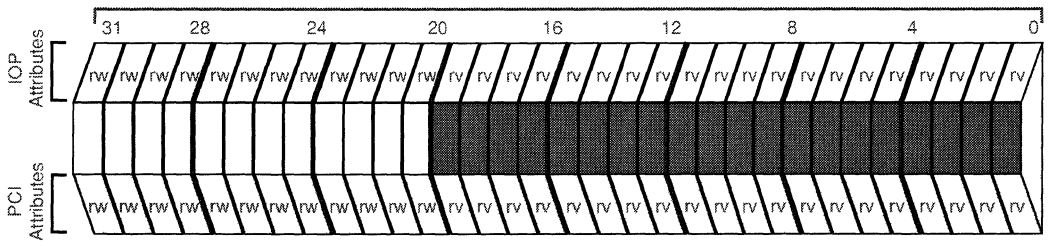
Bit	Default	Description
31:06	0000000H 0_2	Reserved
05:01	00001_2	Circular Queue Size - This field determines the size of each Circular Queue. All four queues are the same size. • 00001_2 - 4K Entries (16 Kbytes) • 00010_2 - 8K Entries (32 Kbytes) • 00100_2 - 16K Entries (64 Kbytes) • 01000_2 - 32K Entries (128 Kbytes) • 10000_2 - 64K Entries (256 Kbytes)
00	0_2	Circular Queue Enable - This bit enables or disables the Circular Queues. When clear the Circular Queues are disabled; however, the MU accepts PCI accesses to the Circular Queue Ports but ignores the data for Writes and returns FFFF.FFFFH for Reads. Interrupts is not generated to the core when disabled. When set, the Circular Queues are fully enabled.

16.8.10 Queue Base Address Register - QBAR

The Queue Base Address Register (QBAR) contains the local memory address of the Circular Queues. The base address must be located on a 1 Mbyte address boundary.

All Circular Queue head and tail pointers are based on the QBAR. When the head and tail pointer registers are read, the Queue Base Address is returned in the upper 12 bits. Writing to the upper 12 bits of the head and tail pointer registers does not affect the Queue Base Address or Queue Base Address Register.

Table 16-16. Queue Base Address Register - QBAR



QBAR

Intel® 80960RM/RN internal bus address
1354H

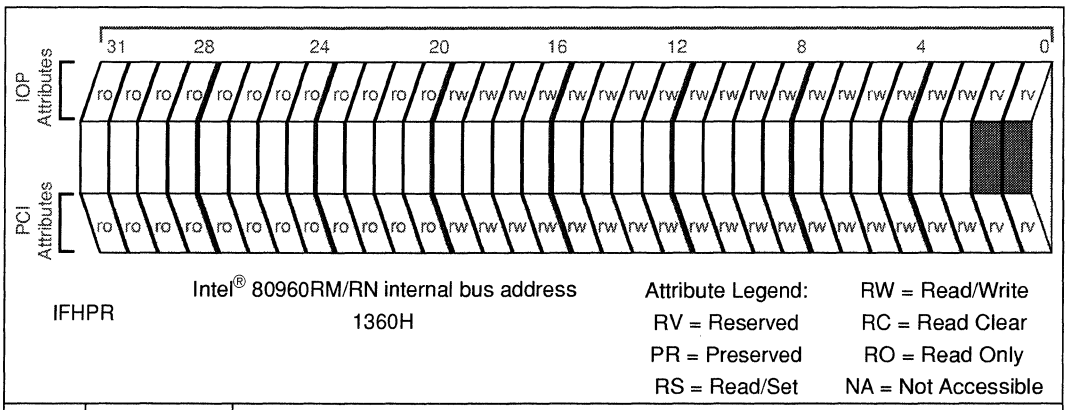
Attribute Legend:
RW = Read/Write
RV = Reserved
RC = Read Clear
PR = Preserved
RO = Read Only
RS = Read/Set
NA = Not Accessible

Bit	Default	Description
31:20	000H	Queue Base Address - Local memory address of the circular queues.
19:00	00000H	Reserved

16.8.11 Inbound Free Head Pointer Register - IFHPR

The Inbound Free Head Pointer Register (IFHPR) contains the local memory offset from the Queue Base Address of the head pointer for the Inbound Free Queue. The Head Pointer must be aligned on a word address boundary. When read, the Queue Base Address is provided in the upper 12 bits of the register. Writes to the upper 12 bits of the register are ignored. This register is maintained by software.

Table 16-17. Inbound Free Head Pointer Register - IFHPR

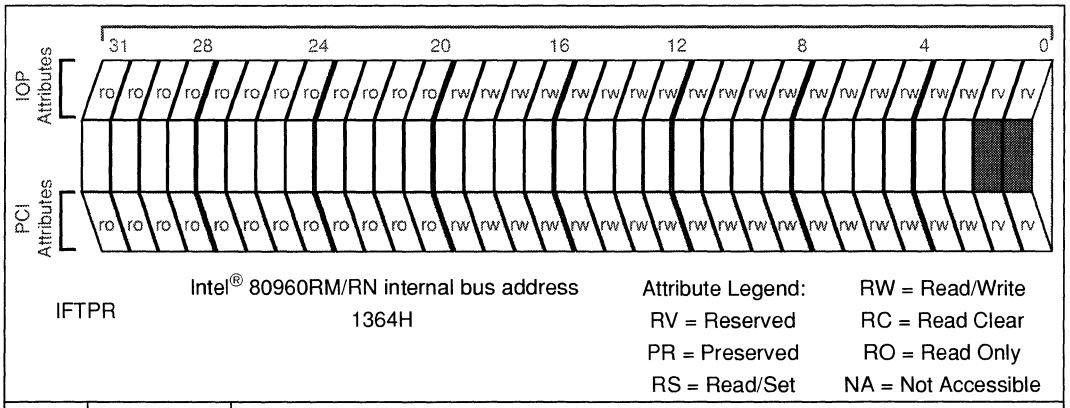


Bit	Default	Description
31:20	000H	Queue Base Address - Local memory address of the circular queues.
19:02	0000H 00_2	Inbound Free Head Pointer - Local memory offset of the head pointer for the Inbound Free Queue.
01:00	00_2	Reserved

16.8.12 Inbound Free Tail Pointer Register - IFTPR

The Inbound Free Tail Pointer Register (IFTPR) contains the local memory offset from the Queue Base Address of the tail pointer for the Inbound Free Queue. The Tail Pointer must be aligned on a word address boundary. When read, the Queue Base Address is provided in the upper 12 bits of the register. Writes to the upper 12 bits of the register are ignored.

Table 16-18. Inbound Free Tail Pointer Register - IFTPR

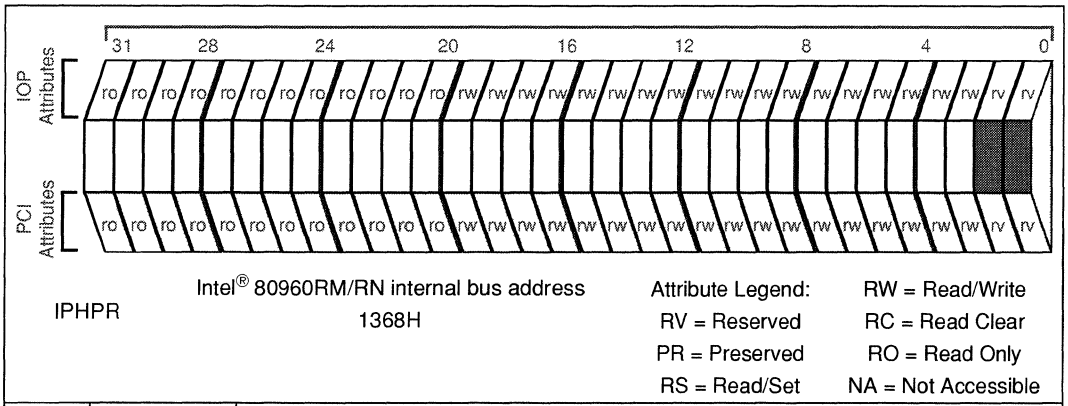


Bit	Default	Description
31:20	000H	Queue Base Address - Local memory address of the circular queues.
19:02	0000H 00_2	Inbound Free Tail Pointer - Local memory offset of the tail pointer for the Inbound Free Queue.
01:00	00_2	Reserved

16.8.13 Inbound Post Head Pointer Register - IPHPR

The Inbound Post Head Pointer Register (IPHPR) contains the local memory offset from the Queue Base Address of the head pointer for the Inbound Post Queue. The Head Pointer must be aligned on a word address boundary. When read, the Queue Base Address is provided in the upper 12 bits of the register. Writes to the upper 12 bits of the register are ignored.

Table 16-19. Inbound Post Head Pointer Register - IPHPR

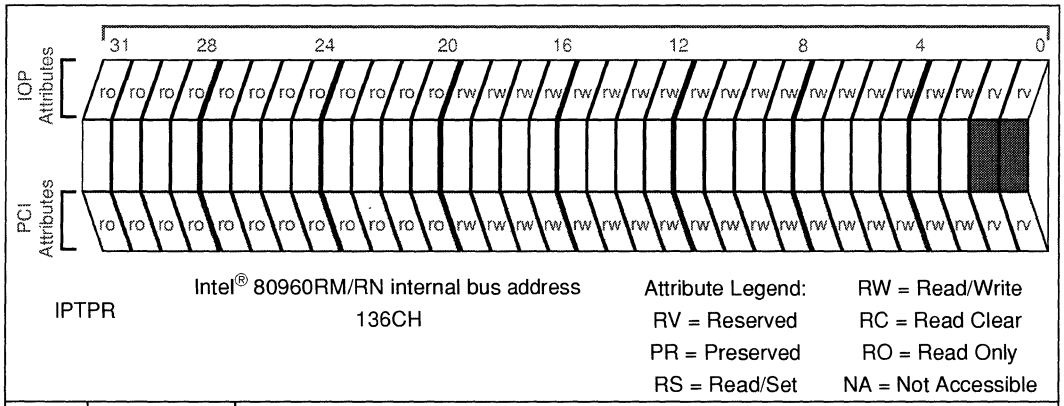


Bit	Default	Description
31:20	000H	Queue Base Address - Local memory address of the circular queues.
19:02	0000H 00_2	Inbound Post Head Pointer - Local memory offset of the head pointer for the Inbound Post Queue.
01:00	00_2	Reserved

16.8.14 Inbound Post Tail Pointer Register - IPTPR

The Inbound Post Tail Pointer Register (IPTPR) contains the local memory offset from the Queue Base Address of the tail pointer for the Inbound Post Queue. The Tail Pointer must be aligned on a word address boundary. When read, the Queue Base Address is provided in the upper 12 bits of the register. Writes to the upper 12 bits of the register are ignored.

Table 16-20. Inbound Post Tail Pointer Register - IPTPR

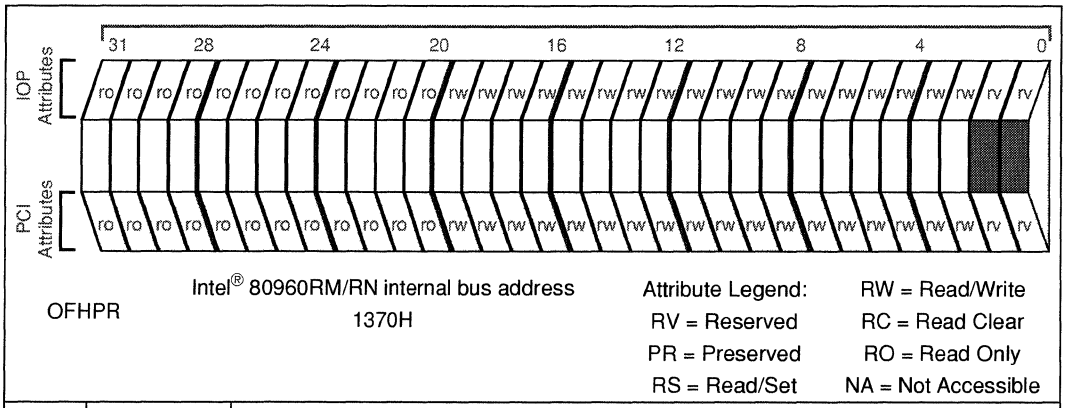


Bit	Default	Description
31:20	000H	Queue Base Address - Local memory address of the circular queues.
19:02	0000H 00_2	Inbound Post Tail Pointer - Local memory offset of the tail pointer for the Inbound Post Queue.
01:00	00_2	Reserved

16.8.15 Outbound Free Head Pointer Register - OFHPR

The Outbound Free Head Pointer Register (OFHPR) contains the local memory offset from the Queue Base Address of the head pointer for the Outbound Free Queue. The Head Pointer must be aligned on a word address boundary. This register is maintained by software. When read, the Queue Base Address is provided in the upper 12 bits of the register. Writes to the upper 12 bits of the register are ignored.

Table 16-21. Outbound Free Head Pointer Register - OFHPR

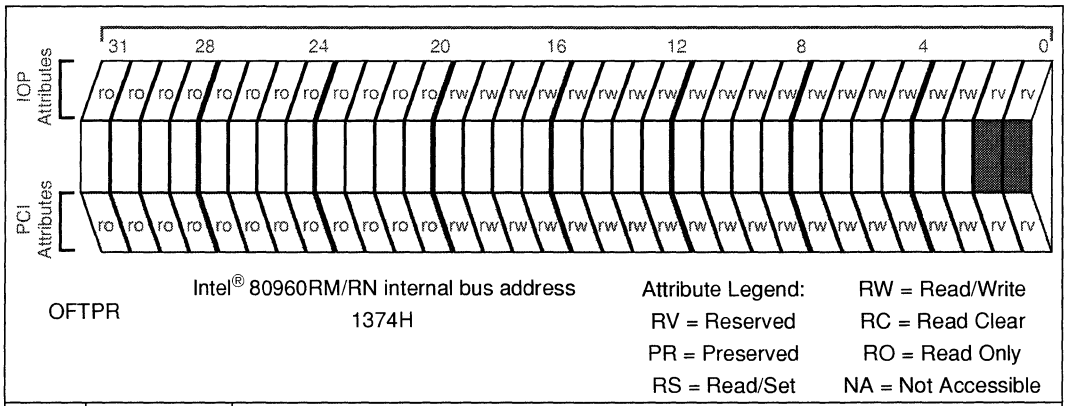


Bit	Default	Description
31:20	000H	Queue Base Address - Local memory address of the circular queues.
19:02	0000H 00_2	Outbound Free Head Pointer - Local memory offset of the head pointer for the Outbound Free Queue.
01:00	00_2	Reserved

16.8.16 Outbound Free Tail Pointer Register - OFTPR

The Outbound Free Tail Pointer Register (OFTPR) contains the local memory offset from the Queue Base Address of the tail pointer for the Outbound Free Queue. The Tail Pointer must be aligned on a word address boundary. When read, the Queue Base Address is provided in the upper 12 bits of the register. Writes to the upper 12 bits of the register are ignored.

Table 16-22. Outbound Free Tail Pointer Register - OFTPR

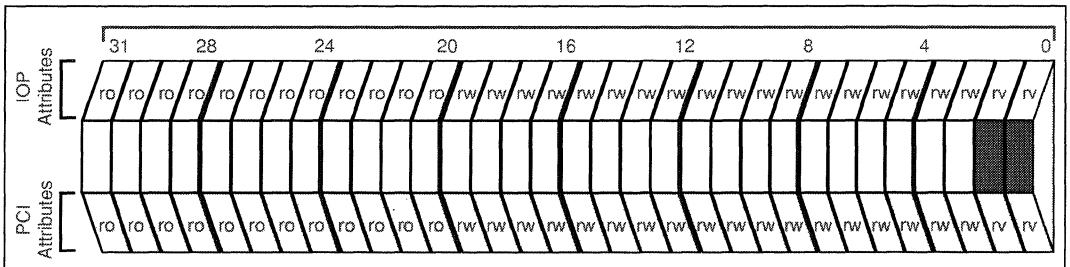


Bit	Default	Description
31:20	000H	Queue Base Address - Local memory address of the circular queues.
19:02	0000H 00_2	Outbound Free Tail Pointer - Local memory offset of the tail pointer for the Outbound Free Queue.
01:00	00_2	Reserved

16.8.17 Outbound Post Head Pointer Register - OPHPR

The Outbound Post Head Pointer Register (OPHPR) contains the local memory offset from the Queue Base Address of the head pointer for the Outbound Post Queue. The Head Pointer must be aligned on a word address boundary. When read, the Queue Base Address is provided in the upper 12 bits of the register. Writes to the upper 12 bits of the register are ignored.

Table 16-23. Outbound Post Head Pointer Register - OPHPR



OPHPR

Intel® 80960RM/RN internal bus address
1378H

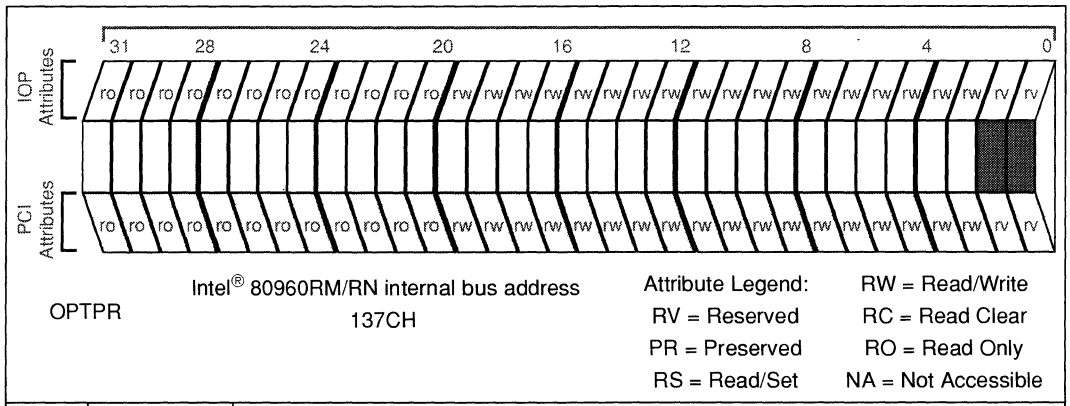
Attribute Legend:
RW = Read/Write
RV = Reserved
RC = Read Clear
PR = Preserved
RO = Read Only
RS = Read/Set
NA = Not Accessible

Bit	Default	Description
31:20	000H	Queue Base Address - Local memory address of the circular queues.
19:02	0000H 00_2	Outbound Post Head Pointer - Local memory offset of the head pointer for the Outbound Post Queue.
01:00	00_2	Reserved

16.8.18 Outbound Post Tail Pointer Register - OPTPR

The Outbound Post Tail Pointer Register (OPTPR) contains the local memory offset from the Queue Base Address of the tail pointer for the Outbound Post Queue. The Tail Pointer must be aligned on a word address boundary. When read, the Queue Base Address is provided in the upper 12 bits of the register. Writes to the upper 12 bits of the register are ignored.

Table 16-24. Outbound Post Tail Pointer Register - OPTPR



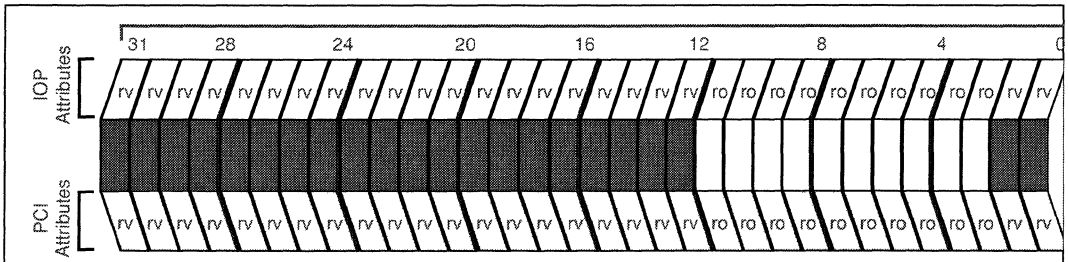
Bit	Default	Description
31:20	000H	Queue Base Address - Local memory address of the circular queues.
19:02	0000H 00_2	Outbound Post Tail Pointer - Local memory offset of the tail pointer for the Outbound Post Queue.
01:00	00_2	Reserved

16.8.19 Index Address Register - IAR

The Index Address Register (IAR) contains the offset of the least recently accessed Index Register. It is written by the MU when the Index Registers are written by a PCI agent. The register is not updated until the Index Interrupt bit in the Inbound Interrupt Status Register is cleared.

The local memory address of the Index Register least recently accessed is computed by adding the Index Address Register to the Primary Inbound ATU Translate Value Register.

Table 16-25. Index Address Register - IAR



IAR

Intel® 80960RM/RN internal bus address
1380H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:12	000000H	Reserved
11:02	00H 00_2	Index Address - is the local memory offset of the Index Register written (050H to FFCH)
01:00	00_2	Reserved

16.9 Power/Default Status

Software is responsible for initializing the Circular Queue Size in the MU Configuration Register and all head and tail pointer registers before setting the Circular Queue Enable bit.

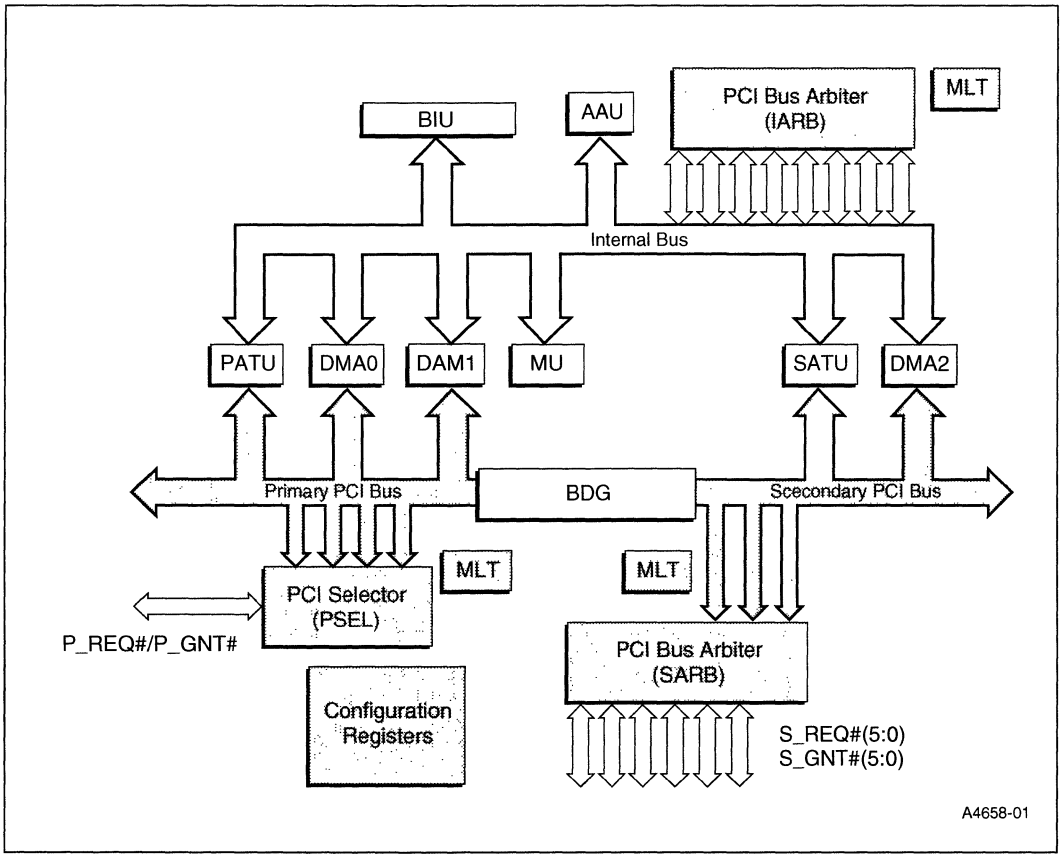
Intel® 80303 I/O Processor Arbitration17

This chapter describes the components which comprise Intel® 80303 I/O processor arbitration, which include two PCI Bus Arbiters, one PCI Selector, and two Latency Timers. The operation modes, setup, and implementation of these components are described in this chapter.

17.1 Arbitration Overview

The 80303 I/O processor interfaces two PCI buses and contains an internal PCI-like bus. Therefore, there are three PCI buses which need an arbitration mechanism. In addition, the 80303 I/O processor contains a secondary PCI arbiter for arbitrating multiple agents on the secondary PCI bus. Figure 17-1 illustrates all the potential PCI bus masters and which arbitration components are responsible for them.

Figure 17-1. Intel® 80303 I/O Processor Arbitration Block Diagram



The four components which comprise 80303 I/O processor arbitration are:

- **PCI Arbiter** (page 17-2) - The PCI Arbiter arbitrates between multiple PCI masters. The arbitration scheme is a round-robin with priority/promotion capabilities. The 80303 I/O processor contains two PCI arbiters: the Secondary PCI Arbiter and the Internal Bus Arbiter.
  - — The Secondary Arbiter (SARB) arbitrates between six potential off-chip secondary PCI bus masters and the three 80303 I/O processor secondary bus masters (Bridge, Secondary ATU, and DMA Channel 2).
  - — The Internal Bus Arbiter (IARB) arbitrates between the eight potential internal bus masters (Primary and Secondary ATUs, three DMA Channels, Messaging Unit, Application Accelerator, and the Bus Interface Unit for the core).
- **PCI Selector (page 17-11)** - The PCI selector arbitrates between the 80303 I/O processor PCI masters for a single **REQ#**/GNT# pair. The selector uses a simple round-robin arbitration scheme.
  - — The Primary PCI Selector (PSEL) selects one of the four primary PCI masters (Primary ATU, DMA Channels 0 and 1, and the Bridge). This selector arbitrates for **P_REQ#/P_GNT#** on the primary PCI bus.
- **Master Latency Timer (page 17-11)** - PCI protocol requires each PCI master to use a master latency timer (MLT). This timer counts the number of PCI cycles a master uses in a single transaction. Once the timer expires, the master must relinquish the PCI bus. The 80303 I/O processor implements three MLTs: one each for the Primary PCI bus, the Secondary PCI bus, and the Internal bus. Once the timer expires, a signal indicates to the current PCI bus master that its time has expired and must relinquish the bus if it no longer maintains its GNT#.
- **Arbitration Configuration Registers (page 17-13)** - Priorities and latency timer values for the arbitration mechanism are programmable, as defined in the Arbitration Configuration Registers.

17.2 PCI Arbiter Overview

The *PCI Local Bus Specification*, Revision 2.2 requires a central arbitration resource for each PCI bus within a system environment. This section details the operation of the PCI Arbiter block.

The PCI Arbiter supports:

- Up to nine PCI bus masters three priority levels for each bus master
- A "fairness" algorithm which ensures that each potential bus master is granted access to the PCI bus independent of other requests
- Hidden, access-based arbitration

PCI uses the concept of access-based arbitration rather than the traditional time slot approach. If a bus master requires the PCI bus for a transaction, the device requests the arbitration logic for the PCI bus. PCI arbitration consists of a simple **REQ#** and GNT# handshake protocol. When a device requires the secondary PCI bus, it asserts its **REQ#** output. The arbitration unit allows the requesting agent access to the bus by asserting that agent's GNT# input.

PCI arbitration is a hidden arbitration scheme where the arbitration sequence occurs in the background while another bus master may currently control the bus. Hidden arbitration has the advantage of not consuming any PCI bandwidth for arbitration overhead.

The arbiter is required by the *PCI Local Bus Specification*, Revision 2.2 to implement a "fair" arbitration algorithm. The PCI Arbiter's algorithm guarantees there is only one GNT# active on the PCI bus at any one time.

17.2.1 Theory of Operation

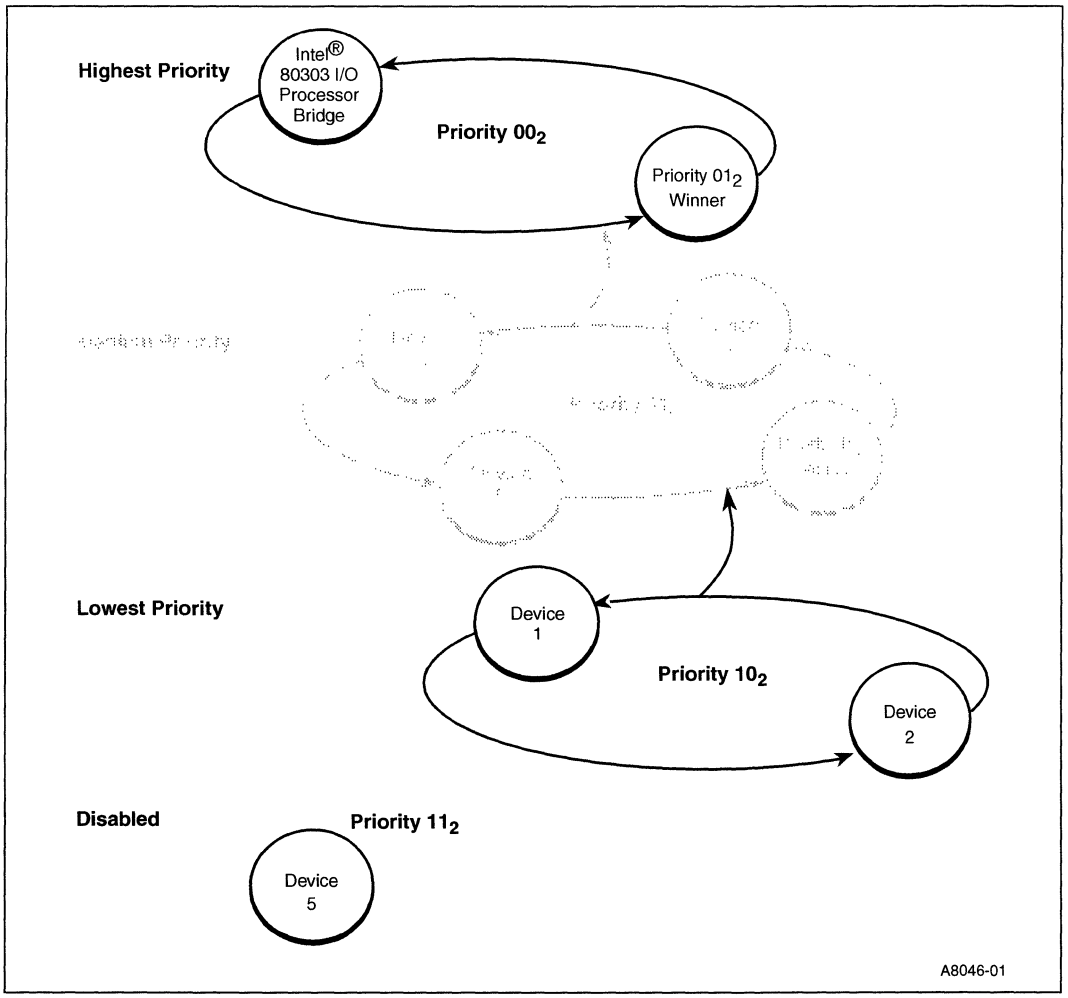
The purpose of the PCI Arbiter is to provide a fair arbitration scheme for all masters on the PCI bus. The PCI Arbiter adheres to all the requirements of the *PCI Local Bus Specification*, Revision 2.2.

17.2.1.1 Priority Mechanism

The PCI Arbiter supports up to nine bus masters. Each request can be programmed to one of three priority levels or be disabled. Application software programs the Secondary Arbiter Control Register (SACR) and the Internal Arbiter Control Register (IACR) to set the initial priority for each bus master. The arbiter promotes the bus master priority levels using a round-robin scheme.

Figure 17-2 is an example showing the three priority levels and reserved slots for the promoted requester.

Figure 17-2. Secondary PCI Arbitration Example



In Figure 17-2, the bus masters are initially programmed to the priorities shown in Table 17-1. The SACR register defines the initial priority levels for the SARB while the IACR defines the initial priority levels for the IARB.

Table 17-1. Bus Master / Programmed Priorities

Bus Master	Programmed Priority
80303 I/O Processor Bridge	High - 00_2
Device 0	Medium - 01_2
Device 3	Medium - 01_2
Device 4	Medium - 01_2
Device 1	Low - 10_2
Device 2	Low - 10_2
Device 5	Disabled - 11_2

Table 17-8 shows the 2-bit values that correspond to each priority level. A priority level of 11_2 effectively disables the associated device by removing it from the arbitration sequence. A device programmed with a 11_2 priority never receives a grant to gain access to the bus.

The priority of the individual bus master determines the level to which the device is placed in the round-robin scheme. The programmed priority determines the starting priority or the lowest priority the device is. If the application programs the device for low priority, the device may be promoted up to medium and then high priority until it is granted the local bus. Once the SARB grants the bus and the device asserts **S_FRAME#**, the device is reset to its initially programmed priority.

Note: If a low priority master requests the bus and there is no other higher priority agent requesting the bus, that master is granted the bus the following clock. The promotion mechanism does not consume bus cycles.

The round-robin arbitration scheme supports three levels of round-robin arbitration: low, medium, and high priority. Using a round-robin mechanism ensures there is a winner for each priority level. To enforce the concept of fairness, a slot is reserved for the winner of each priority level (except the highest) in the next higher priority level. When the winner of a priority level is not granted the bus during that particular arbitration sequence, it is promoted to the next higher level of priority.

17.2.1.2 Priority Example with Three Bus Masters

Table 17-2 presents an example of bus arbitration with three bus masters:

Table 17-2. Bus Arbitration Example – Three Bus Masters

Priority Level	Initial State	Winning Bus Master							
		A	B	A	C	A	B	A	C
High	A	B	A	C	A	B	A	C	A
Medium	B	C	C	B	B	–	C	B	B
Low	C	–	–	–	–	C	–	–	–

NOTE: In this example, all bus masters are continually requesting the bus.

Each of the bus masters (A, B, and C) are constantly requesting the bus and each is at a different priority level. The top row of Table 17-2 lists the current bus master/winner of the highest priority group. The three rows labelled as high, medium and low represent the actual priority levels that devices are currently at based on either their initial programmed priority or promotion through the levels. For example, device C starts out at low priority. Because it is the only device at this priority, it is the winner at low priority and is promoted to medium priority. Later, it wins at the medium priority level (against device B) and is promoted to high priority where it wins the level (against device A) and the bus. Device C is then put back at its programmed priority of low and starts the cycle over.

Continuing with Table 17-2, the winning bus master pattern follows as:

ABACABACABACABAC

17.2.1.3 Priority Example with Six Bus Masters

Table 17-3 illustrates an example of bus arbitration with six bus masters:

Table 17-3. Bus Arbitration Example – Six Bus Masters

Priority Level	Initial State	Winning Bus Master								
		A	B	C	A	B	D	A	B	E
High	AB	BC	AC	AB	BD	AD	AB	BE	AE	AB
Medium	CD	DF	CF	DE	CE	CF	CDF	CDF	CDE	
Low	EF	F	F	F	F	F	F	–	–	–

NOTE: In this example, all bus masters are continually requesting the bus.

Each of the six bus masters (A through F) are constantly requesting the bus. There are two masters programmed at each priority level. The top row of Table 17-3 lists the current bus master/winner of the highest priority group. The three rows labelled as high, medium and low represent the actual priority levels that devices are currently at based on either their initial programmed priority or promotion through the levels.

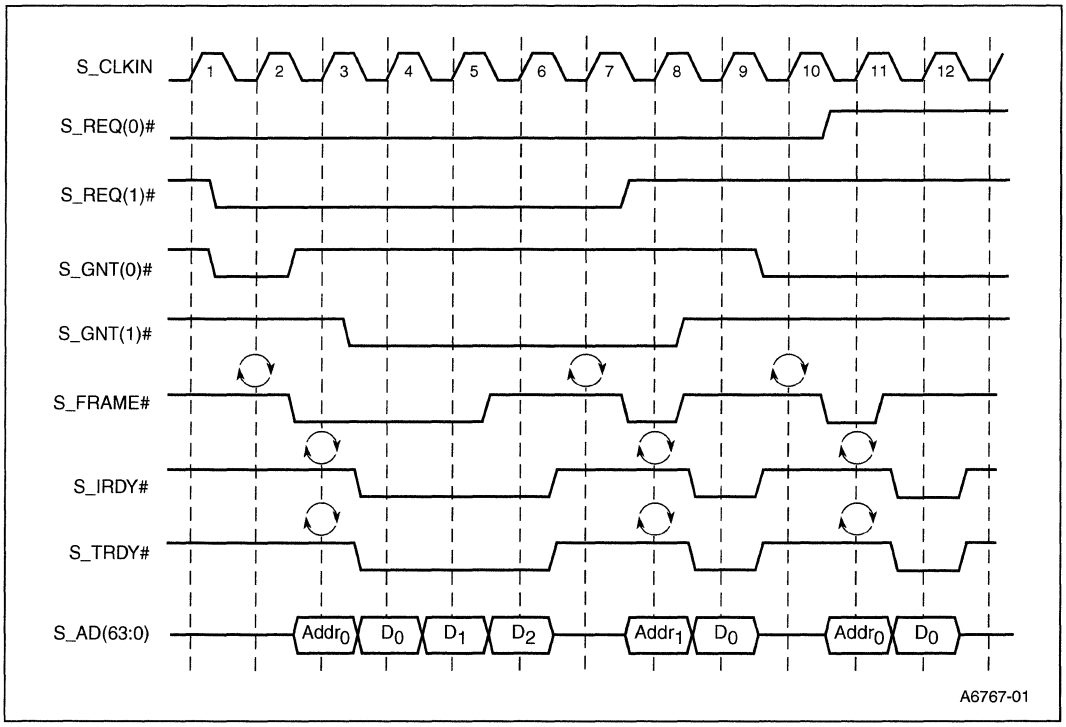
Continuing with Table 17-3, the winning bus master pattern follows as:

ABCABDABFABCABDABEABCABDABF

17.2.1.4 Arbitration Signalling Protocol

The PCI Arbiter interfaces to all requesting agents on the bus through the **REQ#**/GNT# handshaking protocol. A bus master asserts its **REQ#** to request ownership of the PCI bus. When the arbiter determines an agent may use the bus, it asserts the agent's GNT# input. Agents must only assert its **REQ#** to signal a true need for the bus and not to *reserve* the bus. Figure 17-3 illustrates secondary arbitration between masters of equal priority.

Figure 17-3. Arbitration Between Two Masters



An agent can be granted the bus while a previous bus owner still has control of the PCI bus (hidden arbitration). The arbiter is responsible for deciding which PCI device is granted the bus next while each master is responsible for determining when the PCI bus actually becomes free and is allowed to initiate its transaction by asserting FRAME#.

The *PCI Local Bus Specification*, Revision 2.2 indicates that a master may deassert its **REQ#** pin before the arbiter grants the PCI bus to that master. If a master deasserts its **REQ#** pin, the PCI Arbiter re-arbitrates and give bus ownership to the next master based on the priority algorithm defined in Section 17.2.1.1, "Priority Mechanism" on page 17-3.

The PCI Arbiter may deassert an agent's GNT# on any clock. An agent must ensure its GNT# is asserted on the clock edge where it initiates a transaction by asserting FRAME#. If GNT# is deasserted, the transaction may not proceed.

If any of the below three rules are satisfied, the arbiter may deassert one master's GNT# in order to service a higher priority master:

Rule 1: When GNT# is deasserted and FRAME# is asserted, the bus transaction is valid and continues.

Once a master initiates a transaction by asserting FRAME# because the arbiter has granted that master the PCI bus, the arbiter may deassert its GNT# to service the next master.

If the bus master asserts FRAME# and the PCI Arbiter removes its grant on the same cycle, the master assumes ownership of the bus and the arbiter behaves as if the bus was granted and claimed by the original master.

Rule 2: One GNT# can be deasserted coincident with another GNT# being asserted if the bus is not in the idle state. Otherwise, a one clock delay is added between the deassertion of a GNT# and the assertion of the next GNT#. This prevents contention on the **AD[63:0]** bus.

An idle state is defined as a cycle where FRAME# and IRDY# are deasserted. If the PCI bus appears to be idle, a master may actually be using "stepping" to drive the PCI bus. Stepping requires the master to drive **AD[63:0]** one cycle prior to the master's assertion of FRAME#. Refer to the *PCI Local Bus Specification*, Revision 2.2 for more details on address/data stepping.

The PCI Arbiter always satisfies this rule since the arbiter always asserts a master's GNT# one cycle after deasserting another master's GNT#.

Rule 3: While FRAME# is deasserted, GNT# may be deasserted any time in order to service another master, or in response to the associated **REQ#** being deasserted.

The PCI Arbiter continually updates the bus owner for the next transaction. For example, assume the arbiter grants the next transaction to a device of medium priority (Master_A). If a high priority device (Master_B) requests the PCI bus prior to Master_A claiming the bus by asserting FRAME#, the arbiter deasserts Master_A's GNT# and assert Master_B's GNT# one clock later.

Note: **The PCI Arbiter arbitrates the PCI bus by checking REQ[8:0]# on every cycle independent of any transactions on the bus[1].**

By monitoring **REQ[8:0]#**, the arbiter can control the arbitration algorithm described in Section 17.2.1.1, "Priority Mechanism" on page 17-3. The arbiter asserts GNT# two clocks after **REQ#** is asserted if the agent has won the bus. An example of arbitration flow is shown below in Table 17-4.

Table 17-4. Arbitration Flow

Cycle	Event
0	The arbiter is currently driving Master_A's GNT#. The arbitration flow is independent of whether or not Master_A is involved with a transaction. For example, the PCI bus could be parked with Master_A.
1	Master_B asserts its **REQ#** for PCI bus ownership. The arbitration logic calculates that Master_B has a higher priority than Master_A.
2	The arbiter deasserts GNT# for Master_A since Master_B is higher priority.
3	The arbiter asserts GNT# for Master_B.
4	When Master_B drives FRAME#, any of the priority winners that were not granted the bus are promoted to a higher priority level if the reserved promotion slot is unoccupied (Section 17.2.1.1, "Priority Mechanism" on page 17-3).

1. Rule 2 above, indicates that the idle state must be monitored on the PCI bus. The arbiter always asserts a master's GNT# one cycle after deasserting another master's GNT# so the idle state is unimportant.

17.2.1.5 Secondary PCI Bus Arbitration Parking

Arbitration parking occurs when the arbiter asserts GNT# to a selected PCI bus agent and no agent is currently using or requesting the bus.

Upon reset, the IARB parks the internal bus with the BIU and the SARB parks the secondary PCI bus with the bridge. After a master requests, and is granted the bus, the arbiter parks the bus with that master. In other words, the last master that was granted the bus is responsible for parking.

When the secondary PCI bus is parked, the last master continues to assert **S_AD[31:0]**, **S_C/BE[3:0]#**, and **S_PAR**. This prevents the PCI bus from floating.

Note: The 64-bit extension signals (**S_AD[63:32]**, **S_C/BE[7:4]#**, and **S_PAR64**) are not actively driven when the secondary PCI bus is parked on the 80303 I/O processor. Per the *PCI Local Bus Specification*, Revision 2.2, pull-ups provided on the motherboard ensure that these signals are stable.

When a PCI bus is parked during an idle state, the parked agent loses the bus when the arbiter asserts another agent's GNT#. The parked agent relinquishes the bus and stops driving the address and command signals in one clock and parity one clock after that (for the secondary PCI bus). When the arbiter removes GNT# and simultaneously an agent drives FRAME# on the bus, the agent completes the initiated bus transaction.

17.2.2 Atomic Accesses

The 80303 I/O processor core is capable of performing atomic operations to the memory subsystem. Since the BIU (Chapter 12, "Core Processor and Internal Operation") and MCU (Chapter 13, "Memory Controller") reside on the internal bus, the arbiter provides a mechanism for guaranteeing that no other master may access local memory while the core is performing an atomic operation.

17.2.3 Internal and Secondary PCI Arbiter Differences

There is one difference between the secondary arbiter (SARB) and the internal bus arbiter (IARB):

- The IARB maintains a Multi-Transaction Timer (MTT) for the BIU

The 80303 I/O processor core has an inherently small burst size. For this reason, a busy internal bus could inhibit data traffic for the core. To address this issue, the IARB implements a Multi-Transaction Timer (MTT) which allocates a minimum timeslice where the IARB keeps **GNT[8]#** asserted. Refer to Section 17.2.3.1, "Multi-Transaction Timer" on page 17-9 for details.

17.2.3.1 Multi-Transaction Timer

The Internal Arbiter incorporates a Multi-Transaction Timer (MTT) allowing the BIU more internal bus utilization regardless of its inherently small burst size. PCI is a transaction based protocol. In a system with long bursting agents, an agent such as the BIU with a small burst size could get starved.

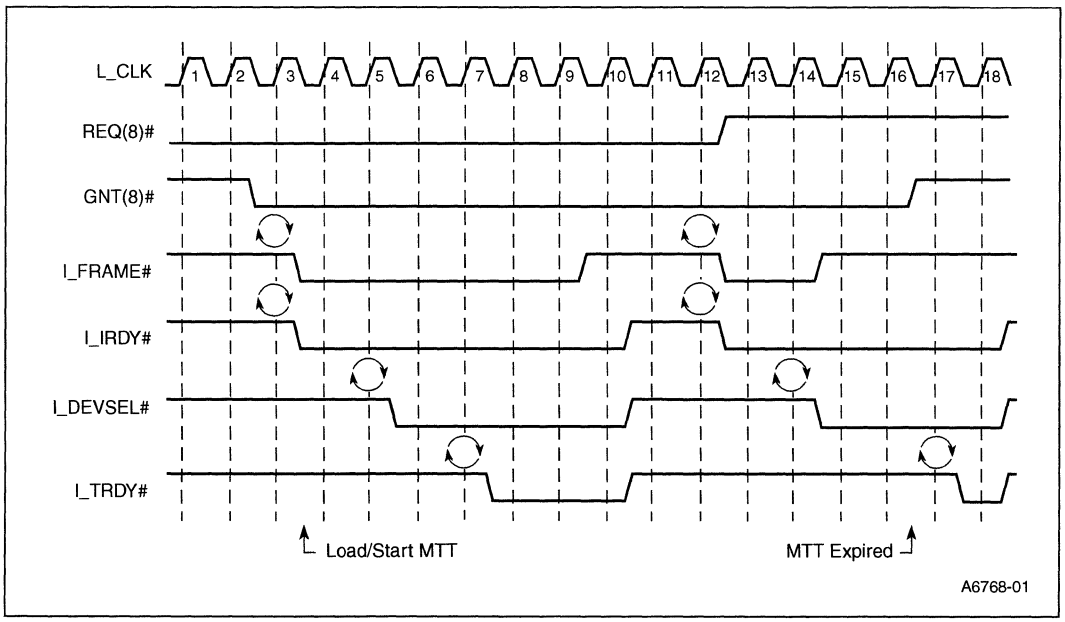
The MTT overcomes this potential bottleneck by guaranteeing a programmed timeslice during which the BIU is granted the internal bus. Once the IARB grants the internal bus to the BIU and the BIU initially asserts **I_FRAME#**, the MTT is loaded with the value programmed in the Multi-Transaction Timer Register (MTTR) and begins to decrement. The arbiter does not remove the BIU's grant (**GNT[8]#**) unless:

- The BIU no longer requests the bus by deasserting **REQ[8]#**.
- The BIU continues to drive **REQ[8]#** and the MTT expires.

Note: Even if a higher-priority master requests the internal bus, the arbiter does not deassert the BIU's grant (**GNT[8]#**) unless any of the above conditions occur.

Figure 17-4 illustrates an example of how the BIU uses the MTT for efficient back-to-back transactions. For this example, the MTTR is programmed for 13 cycles.

Figure 17-4. BIU Back-to-Back Transactions with MTT enabled



Note: The MTT is tightly coupled with the Master Latency Timer (MLT). The Master Latency Timer keeps track of the maximum time a **single** transaction may keep the bus. The MLT governs the PCI master and is detailed in Section 17.4, "Master Latency Timer Operation" on page 17-11. The Multi-Transaction Timer keeps track of the minimum time that **multiple** BIU transactions may keep the internal bus. The MTT governs the internal arbiter.

If the MTTR is programmed with zero, the MTT is effectively disabled.

17.3 PCI Selector Operation

Figure 17-1 shows the block diagram of all the arbitration components in the 80303 I/O processor. 80303 I/O processor arbitration includes one PCI selector block. The responsibility of the PCI selector is to assert an external **REQ#** on behalf of one of the internal masters. The PCI selector also routes the external GNT# to the requesting internal agent.

The Primary PCI bus has four potential masters from the 80303 I/O processor: DMA0, DMA1, PATU, and BDG. If one of the 80303 I/O processor masters needs the primary PCI bus and asserts its **REQ#**, the Primary PCI selector (PSEL) asserts **P_REQ#**. When the Primary PCI slave asserts **P_GNT#**, the PSEL asserts the GNT# for the master requesting the bus.

When a primary master asserts one of the four **REQ#** signals, the PSEL asserts **P_REQ#**. For the case of multiple requests, the selector must arbitrate between the requesting agents. The arbitration is a simple round-robin algorithm.

17.3.1 Primary PCI Bus Arbitration Parking

When the primary PCI bus is parked on the 80303 I/O processor, the last master continues to assert **P_AD[31:0]**, **P_C/BE[3:0]#**, and **P_PAR**. This prevents the PCI bus from floating.

Note: The 64-bit extension signals (**P_AD[63:32]**, **P_C/BE[7:4]#**, and **P_PAR64**) are not actively driven when the secondary PCI bus is parked on the 80303 I/O processor. Per the *PCI Local Bus Specification*, Revision 2.2, pull-ups provided on the motherboard ensure that these signals are stable.

17.4 Master Latency Timer Operation

Each PCI device must contain a Master Latency Timer (MLT). This timer defines the minimum time a PCI master may own the PCI bus. If no other agent is requesting the bus once the MLT expires, the master may continue to use the bus. Once another agent requests the PCI bus and the current bus master's latency timer has expired, the current master must release the bus as soon as possible to allow the requesting agent bus ownership.

17.4.1 Primary and Secondary PCI Master Latency Timers

Each PCI interface of the 80303 I/O processor (primary and secondary) contains a master latency timer (MLT) for use by the internal resources when they are acting as PCI bus masters. Both ATUs, the DMA channels, and the bridge interfaces use an MLT. MLT usage is explained in the *PCI Local Bus Specification*, Revision 2.2.

As defined by the PCI specification, a PCI bus master must release bus ownership as soon as possible when it has lost its GNT# and the MLT has expired. After the MLT expires, the bus master must relinquish the bus when an external device or one of the internal resources requests the bus.

17.4.2 Internal Master Latency Timer

All the internal bus masters use a common Internal Master Latency Timer (IMLT). After the IMLT expires, the current internal bus master must relinquish the bus if the arbiter deasserts its GNT#. The 12-bit IMLT is preloaded with the value programmed into the MLTR.

17.5 Reset Conditions

Table 17-5 shows all the arbitration blocks and the signal responsible for resetting its logic:

Table 17-5. Arbitration Block and Reset Signals

Arbitration Block	Reset With:
Secondary Arbiter (SARB)	S_RST#
Internal Arbiter (IARB)	P_RST#
Primary PCI Selector (PSEL)	P_RST#
Primary Master Latency Timer	P_RST#
Secondary Master Latency Timer	S_RST#

When the secondary bus is reset with S_RST#, the SARB logic is reset which effectively moves all secondary PCI devices to their programmed priority levels and starts the round robin arbitration sequence on the lowest number device at each priority level. Similarly, **I_RST#** moves all the internal agents to their programmed priority levels and starts the round robin arbitration sequence on the lowest number device at each priority level.

Because the SACR is located in the bridge configuration register space, it is reset when P_RST# is asserted. Refer to Section 17.6.1, "Secondary Arbitration Control Register - SACR" on page 17-14 for its value during reset.

17.5.1 S_REQ64# Control

While P_RST# is asserted, the SARB samples the **32BITPCI_EN#** pin. The SARB uses the sampled value to drive **S_REQ64#** while S_RST# is asserted.

- If **32BITPCI_EN#** is deasserted while P_RST# is asserted, **S_REQ64#** is asserted during the assertion of S_RST#. After the deassertion of S_RST#, **S_REQ64#** is driven high (deasserted) for one to two clocks before floating the **S_REQ64#** pin.
- If **32BITPCI_EN#** is asserted while P_RST# is asserted, **S_REQ64#** floats to allow the motherboard to pull-up.

S_REQ64# remains valid for one clock (**P_CLK**) after S_RST# deasserts.

17.6 Register Definitions

Table 17-6 lists Arbitration configuration registers which are detailed further in proceeding sections.

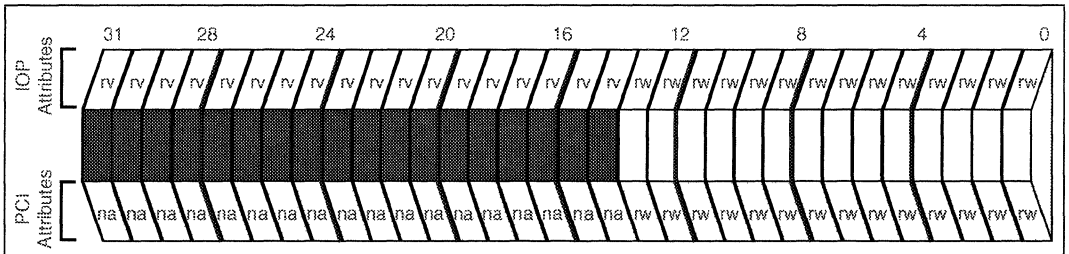
Table 17-6. Secondary Arbiter Register Table

Section, Register Name - Acronym (Page)
Section 17.6.1, "Secondary Arbitration Control Register - SACR" on page 17-14
Section 17.6.2, "Internal Arbitration Control Register - IACR" on page 17-15
Section 17.6.3, "Master Latency Timer Register - MLTR" on page 17-16
Section 17.6.4, "Multi-Transaction Timer Register - MTTR" on page 17-17

17.6.1 Secondary Arbitration Control Register - SACR

The Secondary Arbitration Control Register (SACR) sets the arbitration priority of each device that uses the secondary PCI bus. This register is part of the bridge configuration register space and is accessible from both the primary PCI bus and the 80303 I/O processor core.

Table 17-7. Secondary Arbitration Control Register - SACR



Internal Bus Address
104CH

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:14	0	Reserved
13:12	10_2	Device 5 Priority
11:10	10_2	Device 4 Priority
9:8	10_2	Device 3 Priority
7:6	10_2	Device 2 Priority
5:4	10_2	Device 1 Priority
3:2	10_2	Device 0 Priority
1:0	00_2	Secondary PCI Interface Priority (Bridge, DMA Channel 2, or Secondary ATU)

Each device is given a 2-bit priority shown in Table 17-8. The default values for the SACR give all external secondary PCI devices the lowest priority level and the highest priority to the 80303 I/O processor.

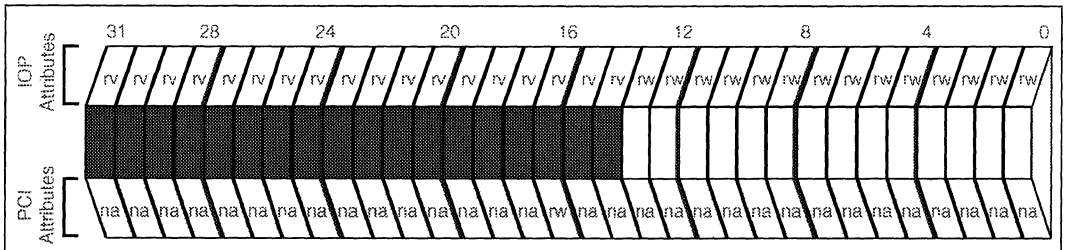
Table 17-8. 2-Bit Priorities

2-Bit Programmed Value	Priority Level
00_2	High Priority
01_2	Medium Priority
10_2	Low Priority
11_2	Disabled

17.6.2 Internal Arbitration Control Register - IACR

The Internal Arbitration Control Register (IACR) sets the arbitration priority of each device that uses the internal bus. This register is part of the local arbitration configuration register space and is accessible from the 80303 I/O processor core.

Table 17-9. Internal Arbitration Control Register - IACR



Internal Bus Address
1600H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RC = Read Clear
RO = Read Only
NA = Not Accessible

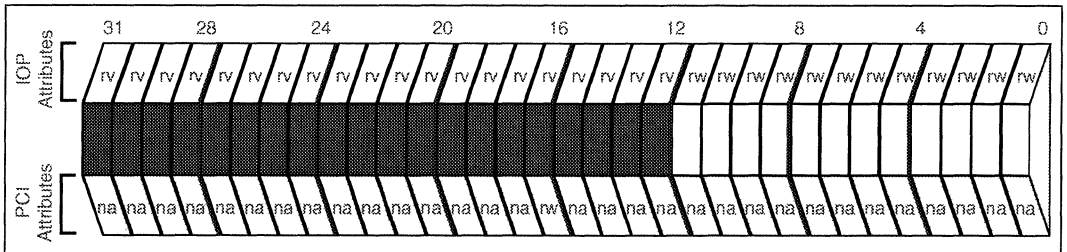
Bit	Default	Description
31:14	0	Reserved
13:12	00_2	Application Accelerator Priority
11:10	00_2	BIU Priority
9:8	00_2	DMA Channel 2 Priority
7:6	00_2	DMA Channel 1 Priority
5:4	00_2	DMA Channel 0 Priority
3:2	00_2	Secondary ATU Priority
1:0	00_2	Primary ATU and Messaging Unit Priority

Each device is given a 2-bit priority shown in Table 17-8. The default values for the IACR give all the internal bus masters the highest priority.

17.6.3 Master Latency Timer Register - MLTR

The Master Latency Timer Register defines preload value for the Internal Bus Master Latency Timer. The preload is a 12-bit value. This register is part of the local arbitration configuration register space and is accessible from the 80303 I/O processor core.

Table 17-10. Master Latency Timer Register - MLTR



Internal Bus Address
1604H

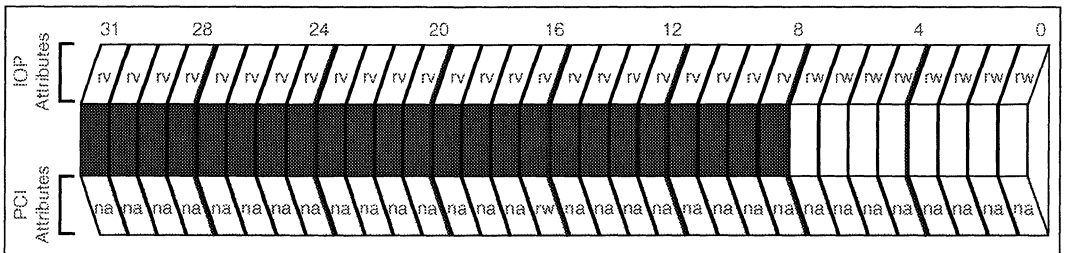
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:12	0	Reserved
11:0	FFFH	Master Latency Timer Preload Value - Indicates the minimum number of clocks a master is allowed to hold the PCI bus for a single transaction.

17.6.4 Multi-Transaction Timer Register - MTTR

The Multi-Transaction Timer Register defines the duration, which the i960 core access, through the BIU, retains **GNT[8]#** across back-to-back transactions. This is an 8-bit value allowing up to 255 dedicated internal bus cycles for as long as **REQ[8]#** is asserted. A value of zero effectively disables the MTT. This register is part of the local arbitration configuration register space and is accessible from the 80303 I/O processor core.

Table 17-11. Multi-Transaction Timer Register - MTTR



Internal Bus Address
1608H

Attribute Legend:
RW = Read/Write
RV = Reserved
RC = Read Clear
PR = Preserved
RO = Read Only
RS = Read/Set
NA = Not Accessible

Bit	Default	Description
31:8	000000H	Reserved
7:0	00H	Multi-Transaction Timer Preload Value - Indicates the minimum number of clocks a master is allowed to hold the PCI bus for a single transaction.

Timers 18

This chapter describes the Intel® 80303 I/O processors dual-independent 32-bit timers. Topics include timer registers (TMRx, TCRx and TRRx), timer operation, timer interrupts, and timer register values at initialization.

Each timer is programmed by the timer registers. These registers are memory-mapped within the processor, addressable on 32-bit boundaries. When enabled, a timer decrements the user-defined count value with each Timer Clock (TCLOCK) cycle. The countdown rate is also user-configurable to be equal to the bus clock frequency, or the bus clock rate divided by 2, 4 or 8. The timers can be programmed to either stop when the count value reaches zero (single-shot mode) or run continuously (auto-reload mode). When a timer's count reaches zero, the timer's interrupt unit signals the processor's interrupt controller. Figure 18-1 shows a diagram of the timer functions. See also Figure 18-2 for the Timer Unit state diagram.

Figure 18-1. Timer Functional Diagram

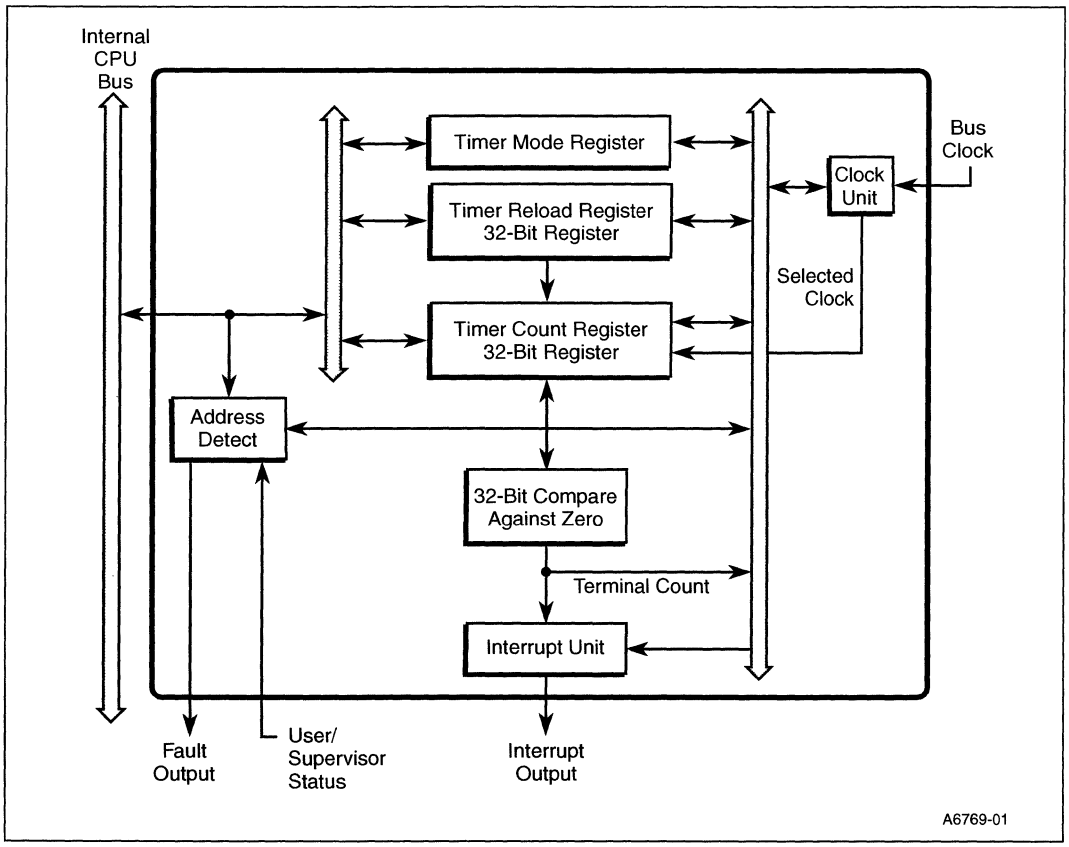


Table 18-1. Timer Performance Ranges

Bus Frequency (MHz)	Max Resolution (ns)	Max Range (mins)
100	10	5.73

18.1 Timer Registers

As shown in Table 18-2, each timer has three memory-mapped registers:

- Timer Mode Register - programs the specific mode of operation or indicates the current programmed status of the timer. This register is described in Section 18.1.1, "Timer Mode Registers – TMR0:1" on page 18-3.
- Timer Count Register - contains the timer's current count. See Section 18.1.2, "Timer Count Register – TCR0:1" on page 18-6.
- Timer Reload Register - contains the timer's reload count. See Section 18.1.3, "Timer Reload Register – TRR0:1" on page 18-7.

Table 18-2. Timer Registers

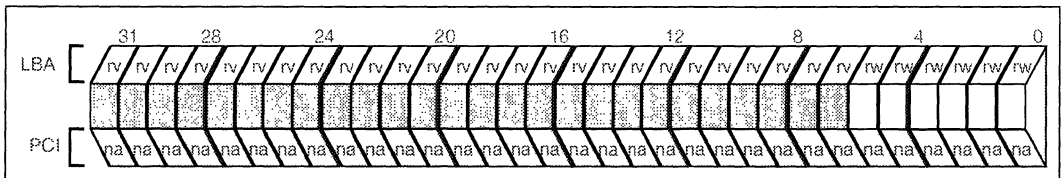
Timer Unit	Register Acronym	Register Name
Timer 0	TMR0	Timer Mode Register 0
	TCR0	Timer Count Register 0
	TRR0	Timer Reload Register 0
Timer 1	TMR1	Timer Mode Register 1
	TCR1	Timer Count Register 1
	TRR1	Timer Reload Register 1

For register memory locations, see Table 18-2, "Timer Registers" on page 18-2.

18.1.1 Timer Mode Registers – TMR0:1

The Timer Mode Register (TMRx) lets the user program the mode of operation and determine the current status of the timer. TMRx bits are described in the subsections following Table 18-3 and are summarized in Table 18-7.

Table 18-3. Timer Mode Register – TMRx



LBA: CH 0-0308H CH 1-0318H **PCI:** NA		**Legend:**NA = Not AccessibleRO = Read Only RV = ReservedPR = PreservedRW = Read/Write RS = Read/SetRC = Read Clear LBA = 80303 local bus addressPCI = PCI Configuration Address Offset
31:06	0000 000H	Reserved. Initialize to 0.
05:04	00_2	Timer Input Clock Selects - TMRx.csel1:0 (00) 1:1 Timer Clock = Bus Clock (01) 2:1 Timer Clock = Bus Clock / 2 (10) 4:1 Timer Clock = Bus Clock / 4 (11) 8:1 Timer Clock = Bus Clock / 8
03	0_2	Timer Register Supervisor Write Control - TMRx.sup (0) Supervisor and User Mode Write Enabled (1) Supervisor Mode Only Write Enabled
02	0_2	Timer Auto Reload Enable - TMRx.reload (0) Auto Reload Disabled (1) Auto Reload Enabled
01	0_2	Timer Enable - TMRx.enable (0) Disabled (1) Enabled
00	0_2	Terminal Count Status - TMRx.tc (0) No Terminal Count (1) Terminal Count

18.1.1.1 Bit 0 - Terminal Count Status Bit (TMRx.tc)

The TMRx.tc bit is set when the Timer Count Register (TCRx) decrements to 0 and bit 2 (TMRx.reload) is not set for a timer. The TMRx.tc bit allows applications to monitor timer status through software instead of interrupts. TMRx.tc remains set until software accesses (reads or writes) the TMRx. The access clears TMRx.tc. The timer ignores any value specified for TMRx.tc in a write request.

When auto-reload is selected for a timer and the timer is enabled, the TMRx.tc bit status is unpredictable. Software should not rely on the value of the TMRx.tc bit when auto-reload is enabled.

The processor also clears the TMRx.tc bit upon hardware or software reset. Refer to Section 11.2, "Intel® 80303 I/O Processor Initialization" on page 11-2.

18.1.1.2 Bit 1 - Timer Enable (TMRx.enable)

The TMRx.enable bit allows user software to control the timer's RUN/STOP status. When:

TMRx.enable = 1	The Timer Count Register (TCRx) value decrements every Timer Clock (TCLOCK) cycle. TCLOCK is determined by the Timer Input Clock Select (TMRx.csel bits 0-1). See Section 18.1.1.5. When TMRx.reload=0, the timer automatically clears TMRx.enable when the count reaches zero. When TMRx.reload=1, the bit remains set. See Section 18.1.1.3.
TMRx.enable = 0	The timer is disabled and ignores all input transitions.

User software sets this bit. Once started, the timer continues to run, regardless of other processor activity. Three events can stop the timer:

- User software explicitly clearing this bit (i.e., TMRx.enable = 0).
- TCRx value decrements to 0, and the Timer Auto Reload Enable (TMRx.reload) bit = 0.
- Hardware or software reset. Refer to Section 11.2, "Intel® 80303 I/O Processor Initialization" on page 11-2.

18.1.1.3 Bit 2 - Timer Auto Reload Enable (TMRx.reload)

The TMRx.reload bit determines whether the timer runs continuously or in single-shot mode. When TCRx = 0 and TMRx.enable = 1 and:

TMRx.reload = 1	The timer runs continuously.

The processor:

1. Automatically loads TCRx with the value in the Timer Reload Register (TRRx), when TCRx value decrements to 0.
2. Decrements TCRx until it equals 0 again.

Steps 1 and 2 repeat until software clears TMRx bits 1 or 2.

TMRx.reload = 0	The timer runs until the Timer Count Register = 0. TRRx has no effect on the timer.

User software sets this bit. When TMRx.enable and TMRx.reload are set and TRRx does not equal 0, the timer continues to run in auto-reload mode, regardless of other processor activity. Two events can stop the timer:

- User software explicitly clearing either TMRx.enable or TMRx.reload.
- Hardware or software reset.

The processor clears this bit upon hardware or software reset.

18.1.1.4 Bit 3 - Timer Register Supervisor Read/Write Control (TMRx.sup)

The TMRx.sup bit enables or disables user mode writes to the timer registers (TMRx, TCRx, TRRx). Supervisor mode writes are allowed regardless of this bit's condition. Software can read these registers from either mode.

When:

TMRx.sup = 1	The timer generates a TYPE.MISMATCH fault when a user mode task attempts a write to any of the timer registers; however, supervisor mode writes are allowed.
TMRx.sup = 0	The timer registers can be written from either user or supervisor mode.

The processor clears TMRx.sup upon hardware or software reset. Refer to Section 11.2, "Intel® 80303 I/O Processor Initialization" on page 11-2.

18.1.1.5 Bits 4, 5 - Timer Input Clock Select (TMRx.csel1:0)

User software programs the TMRx.csel bits to select the Timer Clock (TCLOCK) frequency. See Table 18-4. As shown in Figure 18-1, the bus clock is an input to the timer clock unit. These bits allow the application to specify whether TCLOCK runs at or slower than the bus clock frequency.

Table 18-4. Timer Input Clock (TCLOCK) Frequency Selection

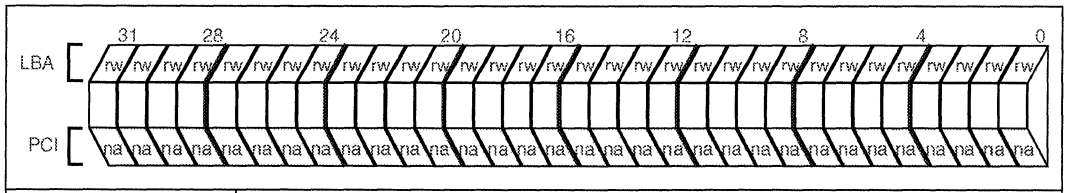
Bit 5 TMRx.csel1	Bit 4 TMRx.csel0	Timer Clock (TCLOCK)
0	0	Timer Clock = Bus Clock
0	1	Timer Clock = Bus Clock / 2
1	0	Timer Clock = Bus Clock / 4
1	1	Timer Clock = Bus Clock / 8

The processor clears these bits upon hardware or software reset (TCLOCK = Bus Clock).

18.1.2 Timer Count Register – TCR0:1

The Timer Count Register (TCRx) is a 32-bit register that contains the timer's current count. The register value decrements with each timer clock tick. When this register value decrements to zero (terminal count), a timer interrupt is generated. When TMRx.reload is not set for the timer, the status bit in the timer mode register (TMRx.tc) is set and remains set until the TMRx register is accessed. Table 18-5 shows the timer count register.

Table 18-5. Timer Count Register – TCRx



LBA: CH 0-0304H CH 1-0314H **PCI:** na		**Legend:**NA = Not Accessible,RO = Read Only, RV = Reserved,PR = Preserved, RW = Read/Write, RS = Read/Set,RC = Read Clear, LBA = 80303 local bus address, PCI = PCI Configuration Address Offset
31:00	0000 0000H	Timer Count Value - TCRx.d31:0

The valid programmable range is from 1H to FFFF FFFFH. Avoid programming TCRx to 0 as it will have varying results as described in Section 18.5, "Uncommon TCRx and TRRx Conditions" on page 18-10.

User software can read or write TCRx whether the timer is running or stopped. Bit 3 of TMRx determines user read/write control (Section 18.1.1.4). The TCRx value is undefined after hardware or software reset.

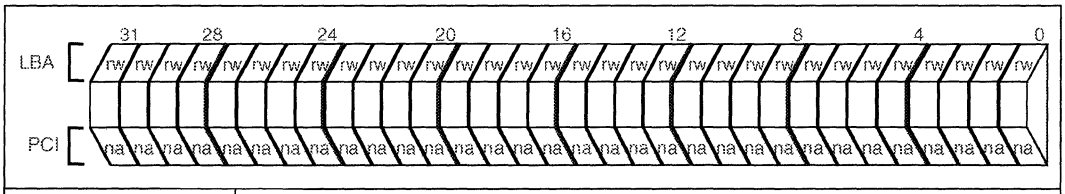
18.1.3 Timer Reload Register – TRR0:1

The Timer Reload Register (TRRx; Table 18-6) is a 32-bit register that contains the timer's reload count. The timer loads the reload count value into TCRx when TMRx.reload is set (1), TMRx.enable is set (1) and TCRx equals zero.

As with TCRx, the valid programmable range is from 1H to FFFF FFFFH. Avoid programming a value of 0, as it may prevent TINTx from asserting continuously. (See Section 18.5, "Uncommon TCRx and TRRx Conditions" on page 18-10 for more information.)

User software can access TRRx whether the timer is running or stopped. Bit 3 of TMRx determines read/write control (Section 18.1.1.4, "Bit 3 - Timer Register Supervisor Read/Write Control (TMRx.sup)" on page 18-5). TRRx value is undefined after hardware or software reset.

Table 18-6. Timer Reload Register – TRRx



LBA: CH 0-0300H CH 1-0310H **PCI:** NA		**Legend:**NA = Not Accessible,RO = Read Only, RV = Reserved,PR = Preserved, RW = Read/Write, RS = Read/Set,RC = Read Clear, LBA = 80303 local bus address, PCI = PCI Configuration Address Offset
31:00	0000 0000H	Timer Auto-Reload Value - TRRx.d31:0

18.2 Timer Operation

This section summarizes timer operation and describes load/store access latency for the timer registers.

18.2.1 Basic Timer Operation

Each timer has a programmable enable bit in its control register (TMRx.enable) to start and stop counting. The supervisor (TMRx.sup) bit controls write access to the enable bit. This allows the programmer to prevent user mode tasks from enabling or disabling the timer. Once the timer is enabled, the value stored in the Timer Count Register (TCRx) decrements every Timer Clock (TCLOCK) cycle. TCLOCK is determined by the Timer Input Clock Select (TMRx.csel) bit setting. The countdown rate can be set to equal the bus clock frequency, or the bus clock rate divided by 2, 4 or 8. Setting TCLOCK to a slower rate lets the user specify a longer count period with the same 32-bit TCRx value.

Software can read or write the TCRx value whether the timer is running or stopped. This lets the user monitor the count without using hardware interrupts. The TMRx.sup bit lets the programmer allow or prevent user mode writes to TCRx, TMRx and TRRx.

When the TCRx value decrements to zero, the unit's interrupt request signals the processor's interrupt controller. See Section 18.3, "Timer Interrupts" on page 18-10 for more information. The timer checks the value of the timer reload bit (TMRx.reload) setting. When TMRx.reload. = 1, the processor:

- Automatically reloads TCRx with the value in the Timer Reload Register (TRRx).
- Decrements TCRx until it equals 0 again.

This process repeats until software clears TMRx.reload or TMR.enable.

When TMRx.reload = 0, the timer stops running and sets the terminal count bit (TMRx.tc). This bit remains set until user software reads or writes the TMRx register. Either access type clears the bit. The timer ignores any value specified for TMRx.tc in a write request.

Table 18-7. Timer Mode Register Control Bit Summary

Bit 3 (TMRx.sup)	TRRx	TCRx	Bit 2 (TMRx.reload)	Bit 1 (TMRx.enable)	Action
X	X	X	X	0	Timer disabled.

NOTE: X = don't care
N = a number between 1H and FFFF FFFFH

18.2.2 Load/Store Access Latency for Timer Registers

As with all other load accesses from internal memory-mapped registers, a load instruction that accesses a timer register has a latency of one internal processor cycle. With one exception, a store access to a timer register completes and all state changes take effect before the next instruction begins execution. The exception to this is when disabling a timer. Latency associated with the disabling action is such that a timer interrupt may be posted immediately after the disabling instruction completes. This can occur when the timer is near zero as the store to TMRx occurs. In this case, the timer interrupt is posted immediately after the store to TMRx completes and before the next instruction can execute. Table 18-8 summarizes the timer access and response timings. Refer also to the individual register descriptions for details.

Note that the processor may delay the actual issuing of the load or store operation due to previous instruction activity and resource availability of processor functional units.

The processor ensures that the TMRx.tc bit is cleared within one bus clock after a load or store instruction accesses TMRx.

Table 18-8. Timer Responses to Register Bit Settings

Name	Status	Action
(TMRx.tc) Terminal Count Bit 0	READ	Timer clears this bit when user software accesses TMRx. This bit can be set 1 bus clock later. The timer sets this bit within 1 bus clock of TCRx reaching zero when TMRx.reload=0.
	WRITE	Timer clears this bit within 1 bus clock after the software accesses TMRx. The timer ignores any value specified for TMRx.tc in a write request.
(TMRx.enable) Timer Enable Bit 1	READ	Bit is available 1 bus clock after executing a read instruction from TMRx.
	WRITE	Writing a '1' enables the bus clock to decrement TCRx within 1 bus clock after executing a store instruction to TMRx.
(TMRx.reload) Timer Auto Reload Enable Bit 2	READ	Bit is available 1 bus clock after executing a read instruction from TMRx.
	WRITE	Writing a '1' enables the reload capability within 1 bus clock after the store instruction to TMRx has executed. The timer loads TRRx data into TCRx and decrements this value during the next bus clock cycle.
(TMRx.sup) Timer Register Supervisor Write Control Bit 3	READ	Bit is available 1 bus clock after executing a read instruction from TMRx.
	WRITE	Writing a '1' locks out user mode writes within 1 bus clock after the store instruction executes to TMRx. Upon detecting a user mode write the timer generates a TYPE.MISMATCH fault.
(TMRx.csel1:0) Timer Input Clock Select Bits 4-5	READ	Bits are available 1 bus clock after executing a read instruction from TMRx.csel1:0 bit(s).
	WRITE	The timer re-synchronizes the clock cycle used to decrement TCRx within one bus clock cycle after executing a store instruction to TMRx.csel1:0 bit(s).
(TCRx.d31:0) Timer Count Register	READ	The current TCRx count value is available within 1 bus clock cycle after executing a read instruction from TCRx. When the timer is running, the pre-decremented value is returned as the current value.
	WRITE	The value written to TCRx becomes the active value within 1 bus clock cycle. When the timer is running, the value written is decremented in the current clock cycle.
(TRRx.d31:0) Timer Reload Register	READ	The current TRRx count value is available within 1 bus clock after executing a read instruction from TRRx. When the timer is transferring the TRRx count into TCRx in the current count cycle, the timer returns the new TCRx count value to the executing read instruction.
	WRITE	The value written to TRRx becomes the active value stored in TRRx within 1 bus clock cycle. When the timer is transferring the TRRx value into the TCRx, data written to TRRx is also transferred into TCRx.

18.3 Timer Interrupts

Each timer is the source for one interrupt. When a timer detects a zero count in its TCRx, the timer generates an internal edge-detected Timer Interrupt signal (TINTx) to the interrupt controller, and the interrupt-pending (IPND.tipx) bit is set in the interrupt controller. Each timer interrupt can be selectively masked in the Interrupt Mask (IMSK) register or handled as a dedicated hardware-requested interrupt. Refer to Chapter 8, "PCI and Peripheral Interrupt Controller Unit" for a description of hardware-requested interrupts.

When the interrupt is disabled after a request is generated, but before a pending interrupt is serviced, the interrupt request is still active (the Interrupt Controller latches the request). When a timer generates a second interrupt request before the CPU services the first interrupt request, the second request may be lost.

When auto-reload is enabled for a timer, the timer continues to decrement the value in TCRx even after entry into the timer interrupt handler.

18.4 Powerup/Reset Initialization

Upon power up, external hardware reset or software reset (**sysctl**), the timer registers are initialized to the values shown in Table 18-9.

Table 18-9. Timer Powerup Mode Settings

Mode/Control Bit	Notes
TMRx.tc = 0	No terminal count
TMRx.enable = 0	Prevents counting and assertion of TINTx
TMRx.reload = 0	Single terminal count mode
TMRx.sup = 0	Supervisor or user mode access
TMRx.csel1:0 = 0	Timer Clock = Bus Clock
TCRx.d31:0 = 0	Undefined
TRRx.d31:0 = 0	Undefined
TINTx output	Deasserted

18.5 Uncommon TCRx and TRRx Conditions

Table 18-7 summarizes the most common settings for programming the timer registers. Under certain conditions, however, it may be useful to set the Timer Count Register or the Timer Reload Register to zero before enabling the timer. Table 18-10 details the conditions and results when these conditions are set.

Table 18-10. Uncommon TMRx Control Bit Settings

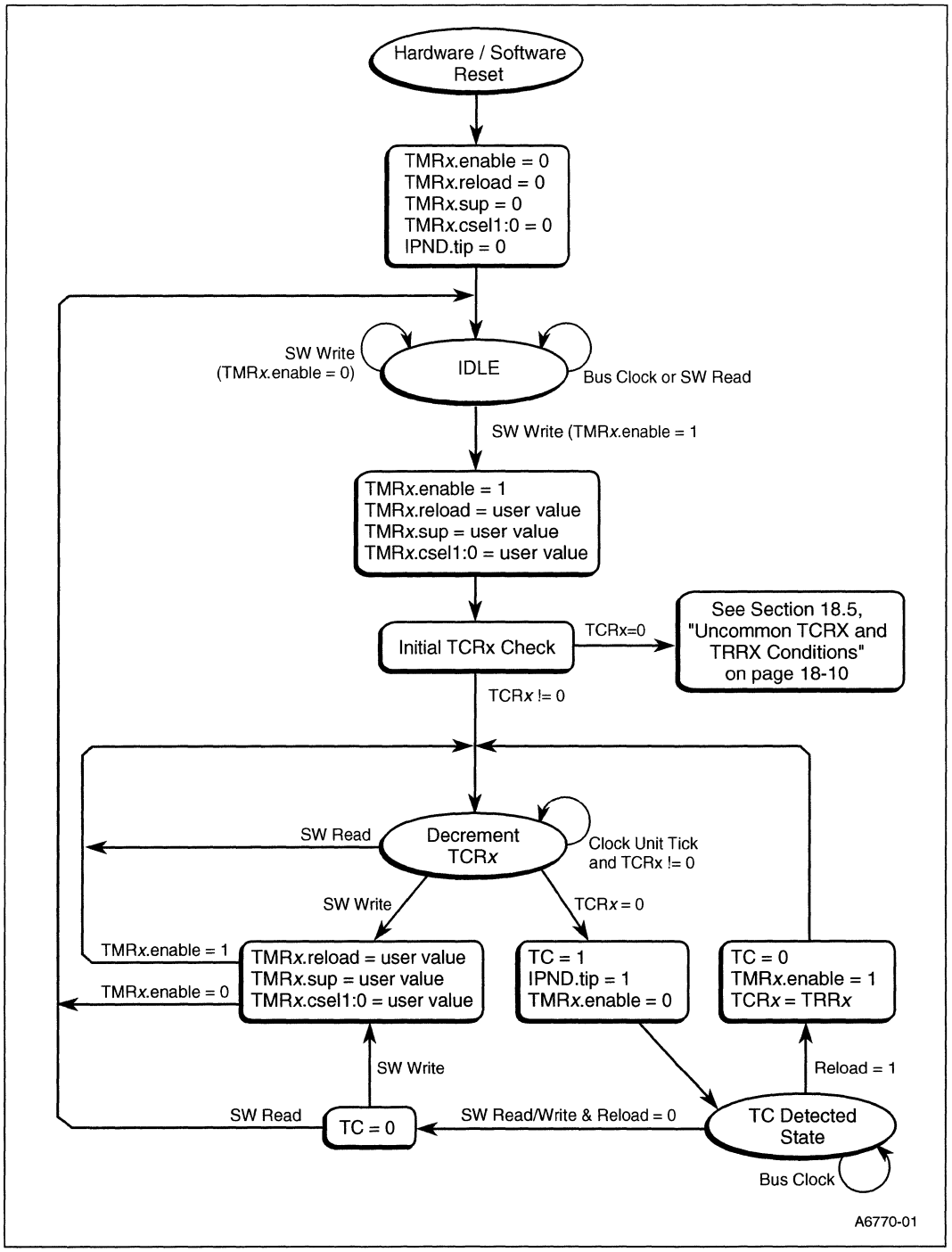
TRRx	TCRx	Bit 2 (TMRx.reload)	Bit 1 (TMRx.enable)	Action
X	0	0	1	TMRx.tc and TINTx set, TMR.enable cleared
0	0	1	1	Timer and auto reload enabled, TINTx not generated and timer enable remains set.
0	N	1	1	Timer and auto reload enabled. TINT.x set when TCRx=0. The timer remains enabled but further TINTx's are not generated.

NOTE: X = don't care
N = a number between 1H and FFFF FFFFH

18.6 Timer State Diagram

Figure 18-2 shows the common states of the Timer Unit. For uncommon conditions see Section 18.5, "Uncommon TCRx and TRRx Conditions" on page 18-10.

Figure 18-2. Timer Unit State Diagram



DMA Controller Unit 19

This chapter describes the integrated Direct Memory Access (DMA) Controller Unit. The operation modes, setup, external interface, and implementation of the DMA Controller are detailed in this chapter.

19.1 Overview

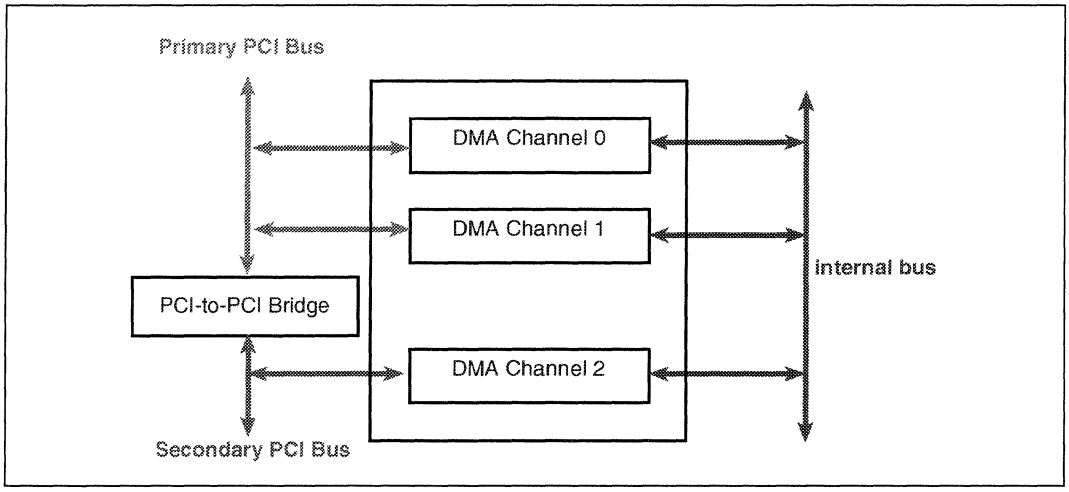
The DMA Controller provides low-latency, high-throughput data transfer capability. The DMA Controller optimizes block transfers of data between the PCI bus and the local processor memory. The DMA is an initiator on the PCI bus with burst capabilities providing a maximum throughput of 528 Mbytes/sec when the PCI bus is operating in 64-bit/66 MHz mode.

The DMA Controller hardware is responsible for executing data transfers and for providing the programming interface. The DMA Controller features:

- Three Independent Channels
- 256-byte queues in Ch-0 and Ch-1
- 64-byte queue in Ch-2
- Utilization of the Intel® 80303 I/O processor Memory Controller Interface
- 2^{32} addressing range on the Internal Bus interface
- 2^{64} addressing range on the primary and secondary PCI interfaces by using PCI Dual Address Cycle (DAC)
- Independent PCI interfaces to the primary and secondary PCI buses
- Hardware support for unaligned data transfers for both the PCI bus and the Internal Bus
- Up to 528 Mbytes/sec burst support for both the PCI bus and the 80303 I/O processor internal bus
- Direct addressing to and from the PCI bus
- Fully programmable directly from the Internal Bus
- Support for automatic data chaining for gathering and scattering of data blocks
- 64-bit/66 MHz PCI and 80303 I/O processor Internal Bus interface.

Figure 19-1 shows the connections of the DMA channels to the PCI busses.

Figure 19-1. DMA Controller

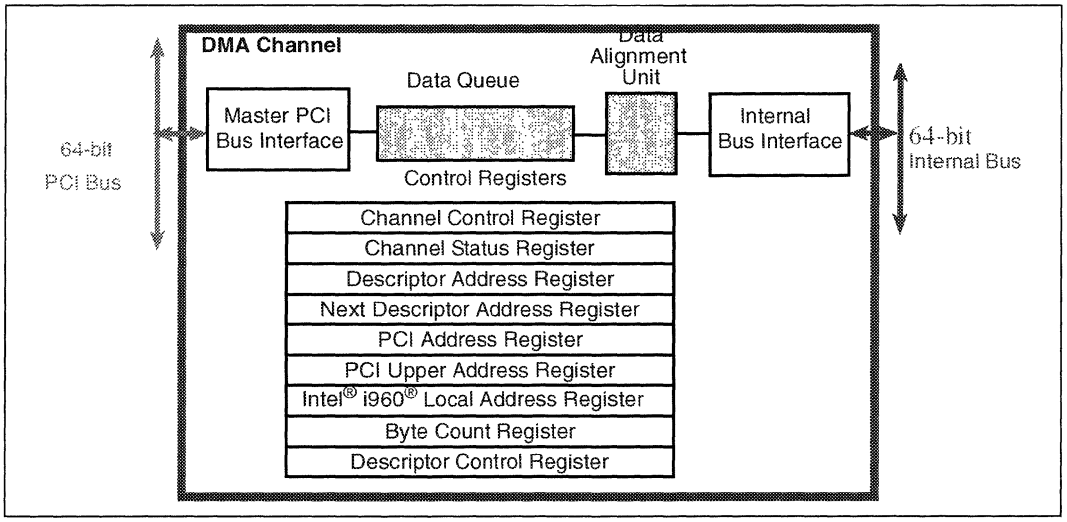


19.2 Theory of Operation

The DMA Controller provides three channels of high throughput PCI-to-Memory transfers. Channels 0 and 1 transfer blocks of data between the primary PCI bus and I/O processor local memory.

Channel 2 transfers blocks of data between the secondary PCI bus and I/O processor local memory. All channels operate identically. Each channel has a PCI bus interface and an internal bus interface. Figure 19-2 shows the block diagram for one channel of the DMA Controller.

Figure 19-2. DMA Channel Block Diagram



Each DMA channel uses direct addressing for both the PCI bus and the internal bus. It supports data transfers to and from the full 64-bit address range of the PCI bus. This includes 64-bit addressing using PCI DAC command. The channel provides a special register which contains the upper 32 address bits for the 64-bit address. The DMA channels do not support data transfers that cross a 32-bit address boundary.

Both the PCI interface and the internal bus interface support large burst lengths up to 4 KBytes.

The channel programming interface is accessible from the internal bus through a memory-mapped register interface. Each channel is programmed independently and has its own set of registers. A DMA transfer is configured by writing the source address, destination address, number of bytes to transfer, and various control information into a chain descriptor in I/O processor local memory. Chain descriptors are described in detail in Section 19.3.

Each channel supports chaining. Chain descriptors that describe one DMA transfer each can be linked together in I/O processor local memory to form a linked list. Each chain descriptor contains all the necessary information for transferring a block of data in addition to a pointer to the next chain descriptor. The end of the chain is indicated when the pointer is zero.

Each channel contains a hardware data alignment unit. This unit enables data transfers from or to unaligned addresses in either the PCI address space or the I/O processor local address space. All combinations of unaligned data are supported with the data alignment unit.

The DMA Controller supports 64-bit and 32-bit wide PCI bus widths. Refer to Section 19.4 for additional information on various PCI bus width transfer mechanisms.

19.3 DMA Transfer

A DMA transfer is a block move of data from one memory address space to another. DMA transfers are configured and initiated through a set of memory-mapped registers and one or more chain descriptors located in local memory. A DMA transfer is defined by the source address, destination address, number of bytes to transfer, and control values. These values are loaded into the chain descriptor before a DMA transfer begins. On the 80303 I/O processor internal bus, the DMA controller will attempt all transactions as 64-bit transfers.

Table 19-1. DMA Registers

Register	Abbreviation	Description
Channel Control Register	CCR	Channel Control Word
Channel Status Register	CSR	Channel Status Word
Descriptor Address Register	DAR	Address of Current Chain Descriptor
Next Descriptor Address Register	NDAR	Address of Next Chain Descriptor
PCI Address Register	PADR	Lower 32-bit PCI Address of Source/Destination
PCI Upper Address Register	PUADR	Upper 32-bit PCI Address of Source/Destination
Intel® 80303 I/O processor Local Address Register	LADR	80303 Address of Source/Destination
Byte Count Register	BCR	Number of Bytes to transfer
Descriptor Control Register	DCR	Chain Descriptor Control Word

19.3.1 Chain Descriptors

All DMA transfers are controlled by chain descriptors located in local memory. A chain descriptor contains the necessary information to complete one data transfer. A single DMA transfer has only one chain descriptor in memory. Chain descriptors can be linked together to form more complex DMA operations.

To perform a DMA transfer, one or more chain descriptors must first be written to 80303 local memory. Figure 19-3 shows the format of an individual chain descriptor. Every descriptor requires six contiguous words in 80303 memory and is required to be aligned on an 8-word boundary. All six words are required.

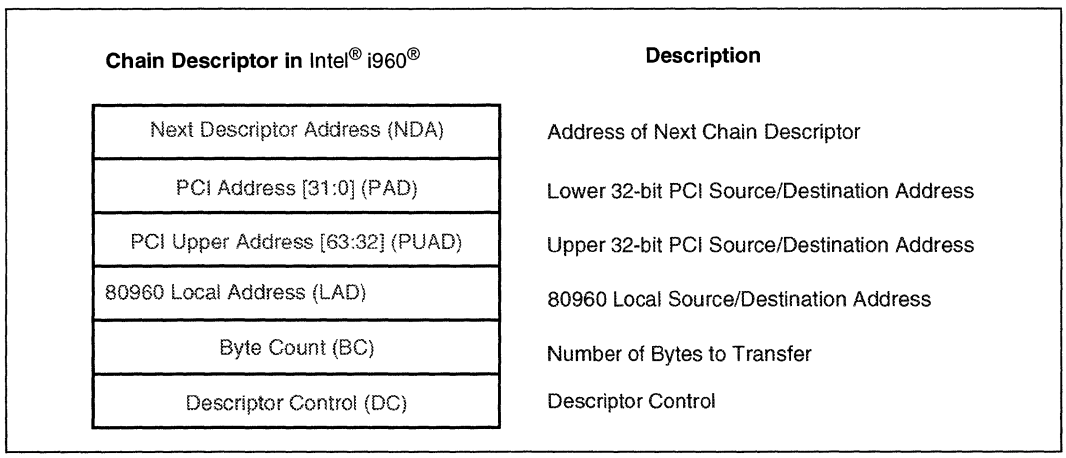
Each word in the chain descriptor is analogous to control register values. Bit definitions for the words in the chain descriptor are the same as for the DMA control registers.

- The first word is the 80303 memory address of the next chain descriptor. A value of zero specifies the end of chain. This value is loaded into the Next Descriptor Address Register. Because chain descriptors must be aligned on an 8-word boundary, the channel may ignore bits 04:00 of this address.
- The second word is the lower 32-bit PCI source/destination address. This address will be generated on the PCI bus. This value is loaded into the PCI Address Register.
- The third word is the upper 32-bit PCI source/destination address, if needed. This address will be used during Dual Address Cycles for driving 64-bit PCI addresses. The address will be ignored if DAC is disabled. This value will be loaded into the PCI Upper Address Register.
- The fourth word is the Intel® i960® source/destination address. This address will be driven on the internal bus. This value will be loaded into the 80960 Local Address Register.
- The fifth word is the Byte Count value. This value determines the number of bytes to transfer. This value will be loaded into the Byte Count Register.
- The sixth word is the Descriptor Control word. This word configures the DMA channel for one DMA transfer. It contains the PCI command type, which determines the direction of the data transfer. This value will be loaded into the Descriptor Control Register.

There are no data alignment requirements for either the PCI address or the 80960 address.

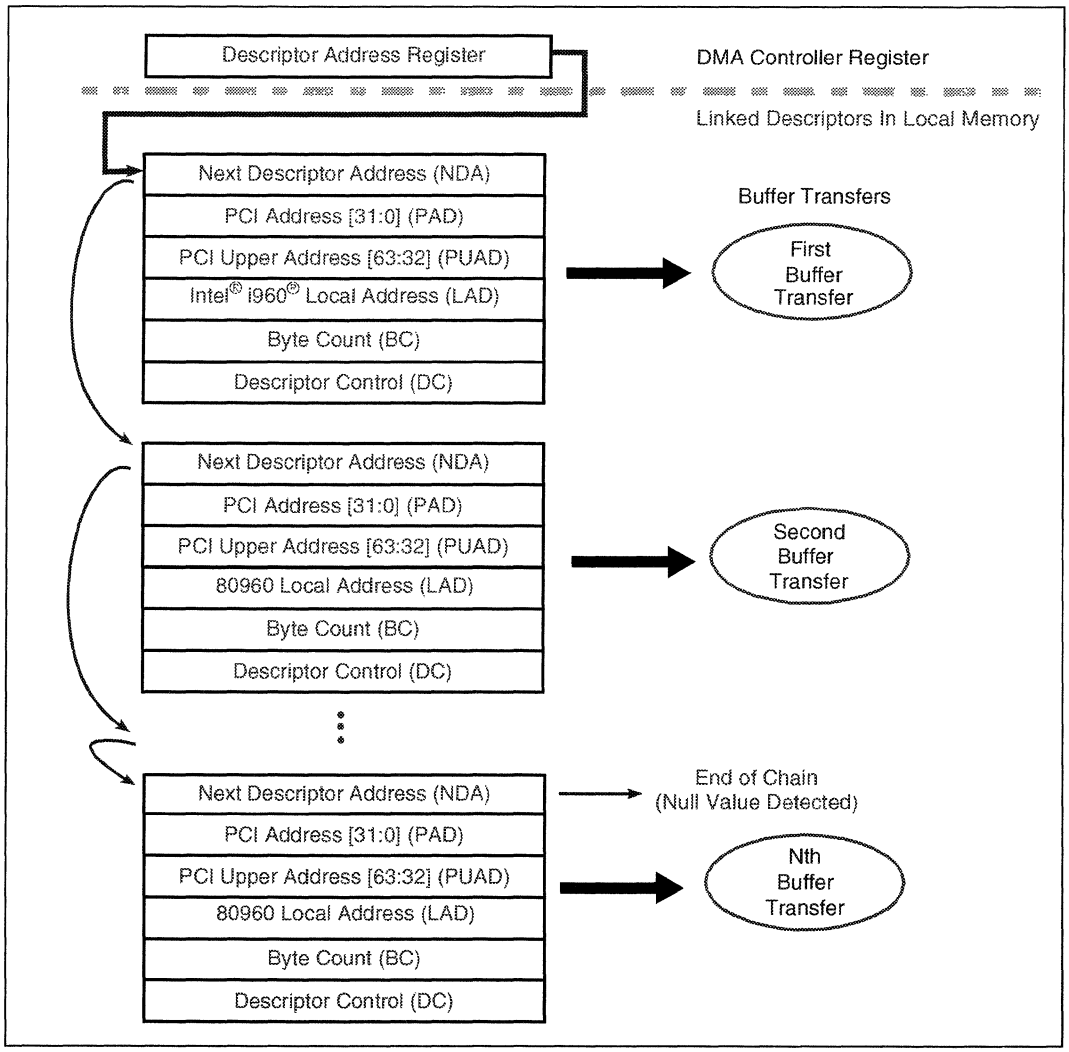
Refer to Section 19.14 for additional descriptions about the DMA Controller registers.

Figure 19-3. DMA Chain Descriptor



A series of chain descriptors can be built in local memory to transfer data between the PCI buses and the internal bus. For example, the application can build multiple chain descriptors to transfer many blocks of data which have different source addresses within the local memory. When multiple chain descriptors are built in 80960 memory, the application can link each of these chain descriptors using the Next Descriptor Address in the chain descriptor. This address logically links the chain descriptors together. This allows the application to build a list of DMA transfers which may not require the 80960 processor until all of the DMA transfers are complete. Figure 19-4 shows a list of DMA transfers built in external memory and how they are linked together.

Figure 19-4. DMA Chaining Operation



19.3.2 Initiating DMA Transfers

A DMA transfer is started by building one or more chain descriptors in 80960 local memory. Each chain descriptor takes the form shown in Figure 19-3. The chain descriptors are required to be aligned on an 8-word boundary in the 80960 local memory.

The following describes the steps for initiating a new DMA transfer:

1. The channel must be inactive prior to starting a DMA transfer. This can be checked by software by reading the *Channel Active* bit in the Channel Status Register (CSR). If this bit is clear, the channel is inactive. If this bit is set, the channel is currently active with a DMA transfer.
2. The CSR must be cleared of all error conditions.
3. The software writes the address of the first chain descriptor to the Next Descriptor Address Register.
4. The software sets the *Channel Enable* bit in the Channel Control Register (CCR). Since this is the start of a new DMA transfer and not the resumption of a previous transfer, the *Chain Resume* bit in the CCR should be clear.
5. The channel starts the DMA transfer by reading the chain descriptor at the address contained in the Next Descriptor Address Register. The channel loads the chain descriptor values into the channel control registers and begins data transfer. The Descriptor Address Register will now contain the address of the chain descriptor just read and the Next Descriptor Address Register will now contain the Next Descriptor Address from the chain descriptor just read.

The last descriptor in the DMA chain list will have zero in the next descriptor address field specifying the last chain descriptor. The NULL value notifies the DMA channel not to read additional chain descriptors from memory.

Once a DMA transfer is active, it may be temporarily suspended by clearing the *Channel Enable* bit in the CCR. Note that this does not abort the DMA transfer. The channel resumes the DMA transfer when the *Channel Enable* bit is set.

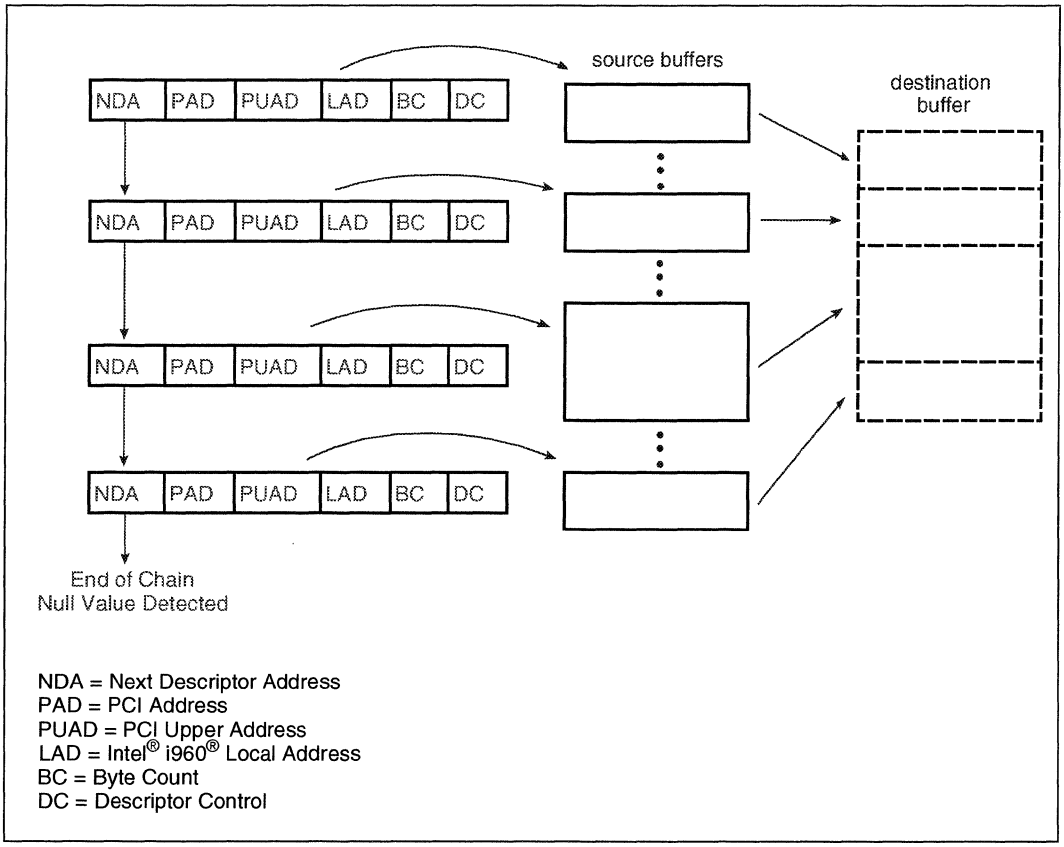
When descriptors are read from external memory, bus latency and memory speed affect chaining latency. Chaining latency is defined as the time required for the channel to access the next chain descriptor plus the time required to set up for the next DMA transfer.

See Section 19.9 for a state diagram of the channel programming model.

19.3.3 Scatter Gather DMA Transfers

The DMA Controller can be used to perform typical scatter gather data transfers. This consists of programming the chain descriptors to gather the data which may be located in non-contiguous blocks of memory. The chain descriptor specifies the destination location such that once all data has been transferred, the data will be contiguous in memory. Figure 19-5 shows how the destination pointers can gather data.

Figure 19-5. Example of Gather Chaining



19.3.4 Synchronizing a Program to Chained Transfers

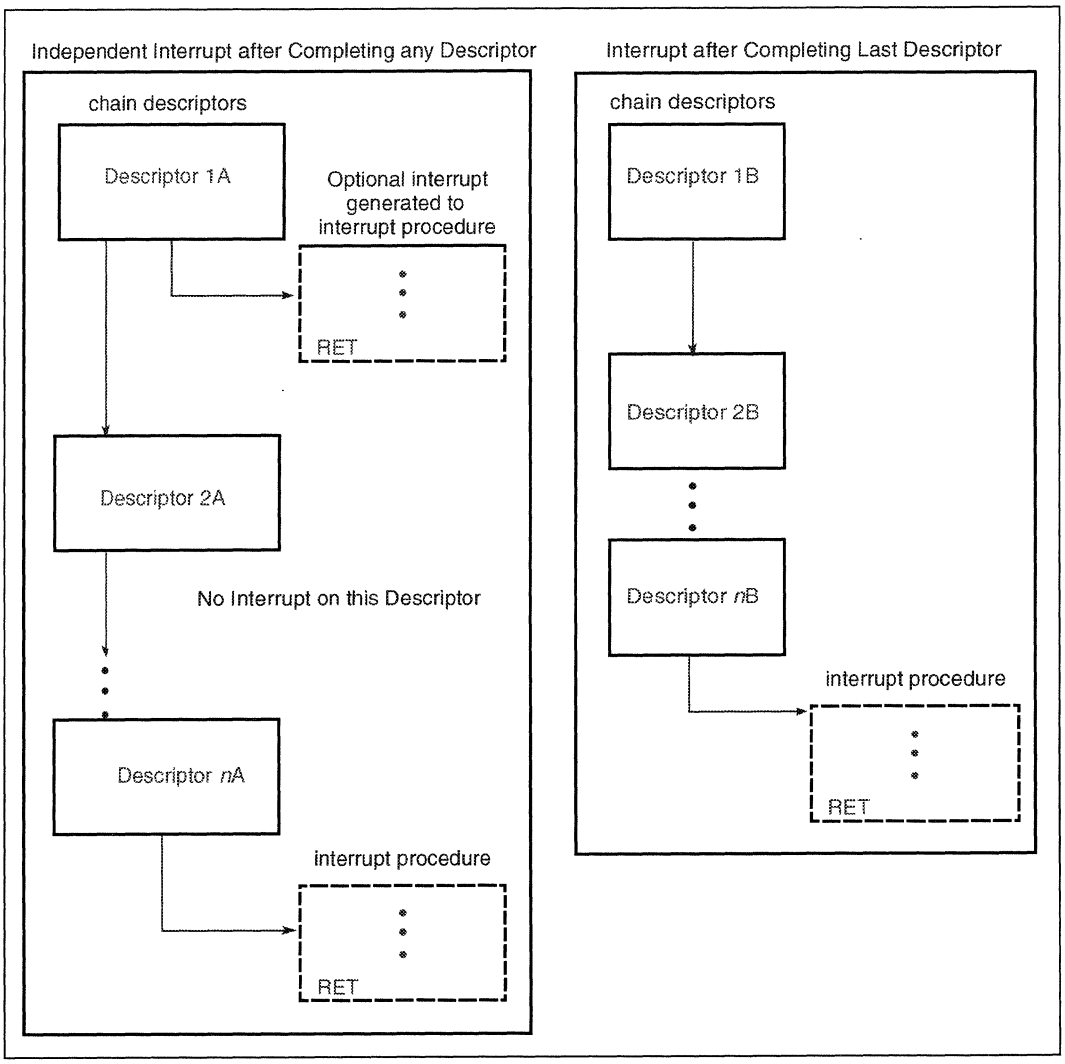
Chained DMA transfers can be synchronized to a program executing on the Intel® i960® core processor through the use of processor interrupts. The channel will generate an interrupt to the i960 core processor under certain conditions. They are:

1. [Interrupt & Continue] The channel completes the data transfer for a chain descriptor and the Next Descriptor Address Register is non-zero. If the *Interrupt Enable* bit within the Descriptor Control Register is set, an interrupt will be generated to the i960 core processor. This interrupt is for synchronization purposes only. The channel will set the *End Of Transfer Interrupt* flag in the Channel Status Register. Since it is not the last chain descriptor in the list, the DMA channel will start to process the next chain descriptor without requiring any processor interaction.
2. [End of Chain] The DMA channel completes the data transfer for a DMA chain descriptor and the Next Descriptor Address Register is zero specifying the end of the chain. If the *Interrupt Enable* bit within the Descriptor Control Register is set, an interrupt will be generated to the i960 core processor. The channel will set the *End Of Chain Interrupt* flag in the Channel Status Register.
3. [Error] An error condition occurs (refer to Section 19.12 for DMA error conditions) during a DMA transfer. The channel will halt operation on the current chain descriptor and not proceed to the next chain descriptor.

Each chain descriptor can independently set the *Interrupt Enable* bit in the Descriptor Control word. This bit enables an independent channel interrupt upon completion of the data transfer for the chain descriptor. This bit can be set or clear within each chain descriptor. Control of interrupt generation within each descriptor aids in synchronization of the executing software with DMA transfers.

Figure 19-6 shows two examples of program synchronization. The left column shows program synchronization based on individual chain descriptors. Descriptor 1A generated an interrupt to the processor, while descriptor 2A did not because the *Interrupt Enable* bit was clear. The last descriptor *n*A, generated an interrupt to signify the end of the chain has been reached. The right column in Figure 19-6 shows an example where the interrupt was generated only on the last descriptor signifying the end of chain.

Figure 19-6. Synchronizing to Chained Transfers



19.3.5 Appending to The End of a Chain

Once a channel has started processing a chain of DMA descriptors, the application software may need to append a chain descriptor to the current chain without interrupting the transfer in progress. The mechanism used for performing this action is controlled by the *Chain Resume* bit in the Channel Control Register.

The channel reads the subsequent chain descriptor each time the channel completes the current chain descriptor and the Next Descriptor Address Register is non-zero. The Next Descriptor Address Register will always contain the address of the next chain descriptor to be read and the Descriptor Address Register will always contain the address of the current chain descriptor.

The procedure for appending chains requires the software to find the last chain descriptor in the current chain and change the Next Descriptor Address in that descriptor to the address of the new chain to be appended. The software then sets the *Chain Resume* bit in the Channel Control Register for the channel. It does not matter if the channel is active or not.

The channel examines the *Chain Resume* bit of the CCR when the channel is idle or upon completion of a chain of DMA transfers. If this bit is set, the channel will re-read the Next Descriptor Address of the current chain descriptor and load it into the Next Descriptor Address Register. The address of the current chain descriptor is contained in the Descriptor Address Register. The channel will clear the *Chain Resume* bit and then examine the Next Descriptor Address Register. If the Next Descriptor Address Register is not zero, the channel will read the chain descriptor using this new address and begin a new DMA transfer. If the Next Descriptor Address Register is zero, the channel will remain or return to idle.

There are three cases to consider:

1. The channel completes a DMA transfer and it is not the last descriptor in the chain. In this case, the channel clears the *Chain Resume* bit and reads the next chain descriptor. The appended descriptor will be read when the channel reaches the end of the original chain.
2. The channel completes a DMA transfer and it is the last descriptor in the chain. In this case, the channel examines the state of the *Chain Resume* bit. If the bit is set, the channel re-reads the current descriptor to get the address of the appended chain descriptor. If the bit is clear, the channel returns to idle.
3. The channel is idle. In this case, the channel examines the state of the *Chain Resume* bit when the CCR is written. If the bit is set, the channel re-reads the last descriptor from the most-recent chain to get the appended chain descriptor.

19.4 64-bit Transfers on a 64-bit PCI Bus

The *PCI Local Bus Specification*, Revision 2.2 provides a mechanism that permits a 64-bit bus master to perform data transfers with a 64-bit target. 64-bit transactions on PCI are dynamically negotiated between the master and the target. The 64-bit PCI extensions add an additional 39 signal pins. The signal definitions and functions are detailed below:

- **AD[63:32]**: High order address/data bus.
- **C/BE[7:4]#**: Byte enables for the high order 4 bytes of data.
- **PAR64#**: Even parity for the upper double word.
- **REQ64#.**: Request 64-bit transfer. This signal is generated by the current 64-bit master to initiate a 64-bit operation. It has the same timing as the **FRAME#** signal.
- **ACK64#**: Acknowledge 64-bit transfer. This signal is generated by the currently addressed target in response to a **REQ64#.** assertion by the initiator. It has the same timing as the DEVSEL# signal.

If either master or target, or both, do not support 64-bit data transfers, 32-bit data transfers are used instead. For 64-bit transfers, all timings during data transfers are identical to that used for 32-bit transfers. Refer to the *PCI Local Bus Specification*, Revision 2.2 for details on the 64-bit Extension.

19.4.1 64-bit Operation with 64-bit Targets

The 64-bit protocol is implemented uniformly for the various internal masters (PCI-to-PCI Bridge Unit, DMA Ch-0, Ch-1, Ch-2 and Address Translation Units) on the 80303 I/O processor.

A 64-bit transfer is initiated by the DMA controller by the assertion of **REQ64#.** If the target device can perform 64-bit transfers, **ACK64#** will be asserted when the target asserts DEVSEL# to claim the transaction. When a target signals the ability to complete a transaction as a 64-bit transaction, the master interface of the DMA controller completes the transaction as a 64-bit master. In this instance, up to eight bytes are transferred in each data phase. The DMA channel will decrement the byte count by 8 for every successful data transfer cycle.

Refer to Section 14.6.3.1, "64-Bit Protocol" on page 14-32 for complete details on 64-bit Initiator and 64-bit Target Operation.

19.4.2 64-bit Operation with 32-bit Targets

A 64-bit transfer is initiated by the DMA controller by the assertion of **REQ64#.**. If the target device cannot perform 64-bit transfers, **ACK64#** will remain deasserted when the target asserts DEVSEL# to claim the transaction. When a target signals its inability to complete a transaction as a 64-bit transaction, the master interface of the DMA controller completes the transaction as a 32-bit master. In this instance, up to four bytes are transferred in each data phase. The DMA channel will decrement the byte count by 4 for every successful data transfer cycle.

Should a slave disconnect on an even word boundary, then all future transfers will be carried out as 32-bit transfers for the current chain descriptor transaction.

Refer to Section 14.6.3.2, "64-Bit Operation with 32-Bit Targets" on page 14-34 for complete details on 64-bit Initiator and 32-bit Target Operation.

19.4.3 64-bit Addressing

The standard PCI bus transactions support a 32-bit address. 64-bit addressing generated by any DMA channel on the PCI bus using the PCI DAC command allows for an extension to the 32-bit addressing space. During DAC cycles on a 32-bit bus, none of the signals listed as a 64-bit extension are used. During DAC cycles on a 64-bit bus, the upper 32-bits of the PCI address bus(**AD[63:32]**) are driven during both address phases. Also, the associated data command for the transaction (**C/BE 7:4]**) is driven during both address phases.

Refer to Section 14.5.4, "64-Bit Address Decoding - Dual Address Cycles" on page 14-21 for complete details on the 64-bit addressing protocol.

19.4.4 66 MHz Operation

All of the 80303 I/O processor DMA channels are capable of PCI 66 MHz operation to support data transfer rates of up to 264 MBytes/sec with a 32-bit bus or 528 Mbytes/sec with a 64-bit bus. Differences between 33 MHz PCI and 66 MHz PCI are minimal. Both share the same protocol, and signal definitions. The 66 MHz PCI extension adds one additional signal to each PCI interface. The signal and its function is

- **P_M66EN** - when asserted , indicates that the Primary PCI bus will run at 66 MHz
- **S_M66EN** - when asserted , indicates that the Secondary PCI bus will run at 66 MHz

At PCI bus reset, each individual PCI bus (primary and secondary) will independently sample their respective **M66EN** signals. If this signal is high, the bus is 66 MHz capable, and in the case of the primary bus, the clock unit will be configured to accept a 66 MHz Primary PCI Bus clock.

The 66 MHz capable 80303 I/O processor supports the following primary and secondary bus frequency combinations:

- 66 MHz primary bus, 66 MHz secondary bus
- 66 MHz primary bus, 33 MHz secondary bus
- 33 MHz primary bus, 33 MHz secondary bus

The 80303 I/O processor does not support 33 MHz primary/66 MHz secondary bus operation, where the secondary bus is operating at twice the frequency of the primary bus. If **P_M66EN** is low (primary bus at 33 MHz), then the 80303 I/O processor pulls down **S_M66EN** to indicate that the secondary PCI bus is operating at 33 MHz.

For more details on the 80303 I/O processor's 66 MHz PCI clock scheme and the operation of the secondary PCI bus clocks, please see Chapter 25, "Clocking and Reset".

19.5 Data Transfers

The 80303 I/O processor's DMA controller is optimized to perform data transfers between the PCI bus and local memory. These transfers are summarized in the following sections. The DMA Controller does not support *Master-Initiated* wait states on either interface.

19.5.1 PCI-to-Local Memory Transfers

PCI-to-Local memory transfers perform read cycles on the PCI bus and place the data into the DMA channel queues. Once data is placed into the queue, the internal bus interface of the DMA channel will request the internal bus and drain the queue by writing the data to the local memory.

The application software can use the various PCI command types to improve system performance for these transfers. The three defined PCI read commands include: Memory Read, Memory Read Line, and Memory Read Multiple. Refer to the *PCI Local Bus Specification*, Revision 2.2 for full description of these PCI commands.

For example, a Memory Read Multiple command can be programmed if the block size is larger than a cache line. This is used to notify the PCI target that the DMA channel intends to transfer a large block of data and the target should try to read ahead and anticipate the DMA controller read requests.

The application software determines which command type best meets the needs of the system.

19.5.2 Local Memory to PCI Transfers: Memory Write Command

Local memory to PCI transfers perform read cycles on the internal bus and place the data into the DMA channel queues. Once data is placed into the queue, the PCI bus interface of the DMA channel will request the PCI bus and drain the queue by writing the data to the PCI bus. Memory Write commands can be used for all data transfers to the PCI bus.

Local memory to PCI transfers generate two PCI write command types: Memory Write and Memory Write and Invalidate. The application software can use the appropriate PCI command type. However, the PCI target may provide better system performance by using the Memory Write and Invalidate command.

19.5.3 Local Memory to PCI Transfers: Memory Write and Invalidate Command

The second mechanism for performing local memory to PCI transfers may improve system performance based on the PCI target capabilities. The Memory Write and Invalidate (MWI) command improves system performance when the target is cacheable memory.

The DMA channel will attempt to use the Memory Write and Invalidate command on the PCI bus whenever programmed by the application software. The DMA channel will request the PCI bus once a complete cache line is available in the DMA queue. However, there are a number of circumstances which may prevent the DMA channel from actually initiating the MWI command. It is the responsibility of the application software to meet the requirements for the MWI command.

If any of the following three conditions is *not* met, the channel will convert the MWI command to a Memory Write command for the complete DMA transfer:

1. The ATU Cacheline Size Register (ATUCLSR), located in the ATU configuration space, must have a valid value other than zero. This register is programmed by host software.
2. The ATUCLSR must have a legal value which is less than or equal to the number of queue entries in the DMA channel queue. (The channel must guarantee an entire cache line can be transferred during an MWI bus transaction).
3. The Memory Write and Invalidate Enable bit in the Primary ATU Command Register (for channels 0 and 1) or the Secondary ATU Command Register (for channel 2) must be set.

If the above conditions are met, the DMA channel provides full MWI support. For example, to transfer an 80 byte block to a PCI address of 8001CH while the ATUCLSR is 8 DWORDs, the DMA channel will perform three PCI transactions:

1. Transfer of 4 bytes at address 8001CH using the Memory Write command.
2. Transfer of 64 bytes at address 80020H using the MWI command.
3. Transfer of 12 bytes at address 80060H using the Memory Write command.

19.5.4 Exclusive Access

The DMA Controller does not support exclusive access through the PCI LOCK# signal.

19.6 Data Queues

DMA Ch-0 and Ch-1 each contain a 256-byte, bidirectional data queue. DMA Ch-2 on the secondary side contains a 64-byte, bidirectional data queue. These queues temporarily hold data to increase performance of data transfers in both directions.

19.7 Data Alignment

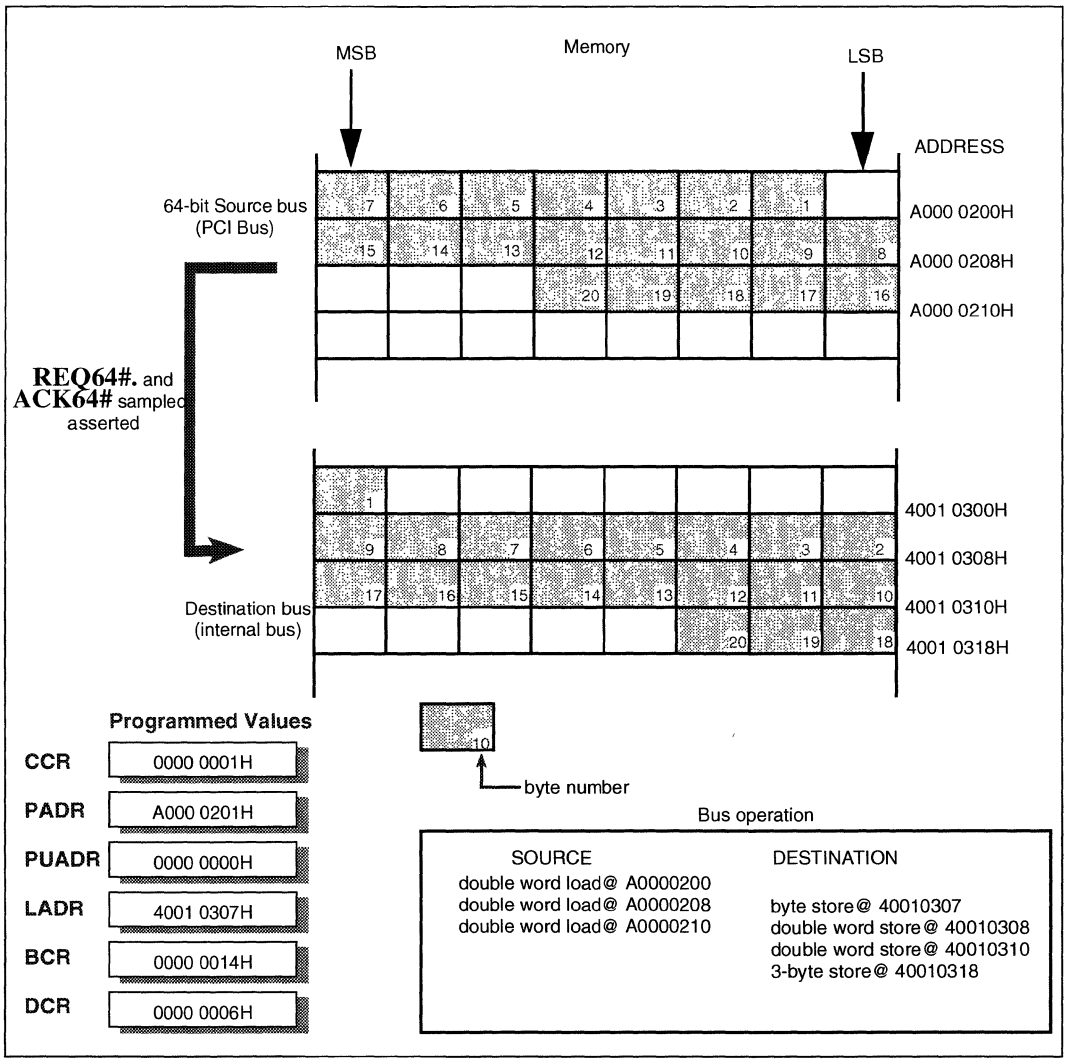
Each channel contains a hardware data alignment unit to support unaligned data transfers between the source and destination busses. The data alignment unit optimizes data transfers to and from 32 and 64-bit memory. The channel will reformat data words for the correct bus data width.

Aligned data transfers involve data accesses that fall on natural boundaries. For example; double words are aligned on 8-byte boundaries and words are aligned on 4-byte boundaries. DMA transfers can occur with both the source and destination addresses unaligned.

19.7.1 64-bit Unaligned Data Transfers

Figure 19-7 illustrates a DMA transfer between unaligned 64-bit source and destination addresses.

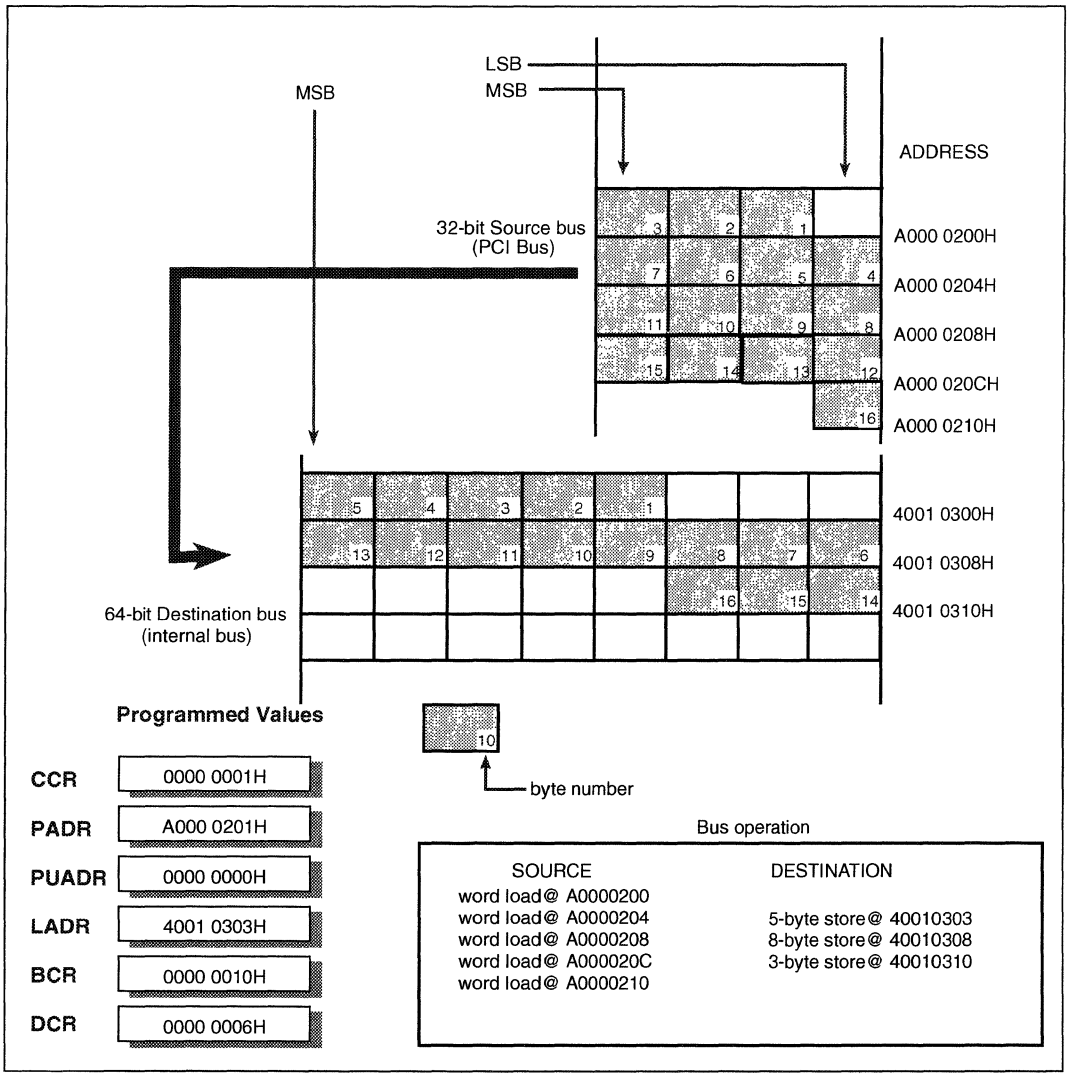
Figure 19-7. Optimization of an Unaligned DMA



19.7.2 64/32-bit Unaligned Data Transfers

Figure 19-8 illustrates a DMA transfer between an unaligned 32-bit source address and an unaligned 64-bit destination address.

Figure 19-8. Optimization of an Unaligned DMA



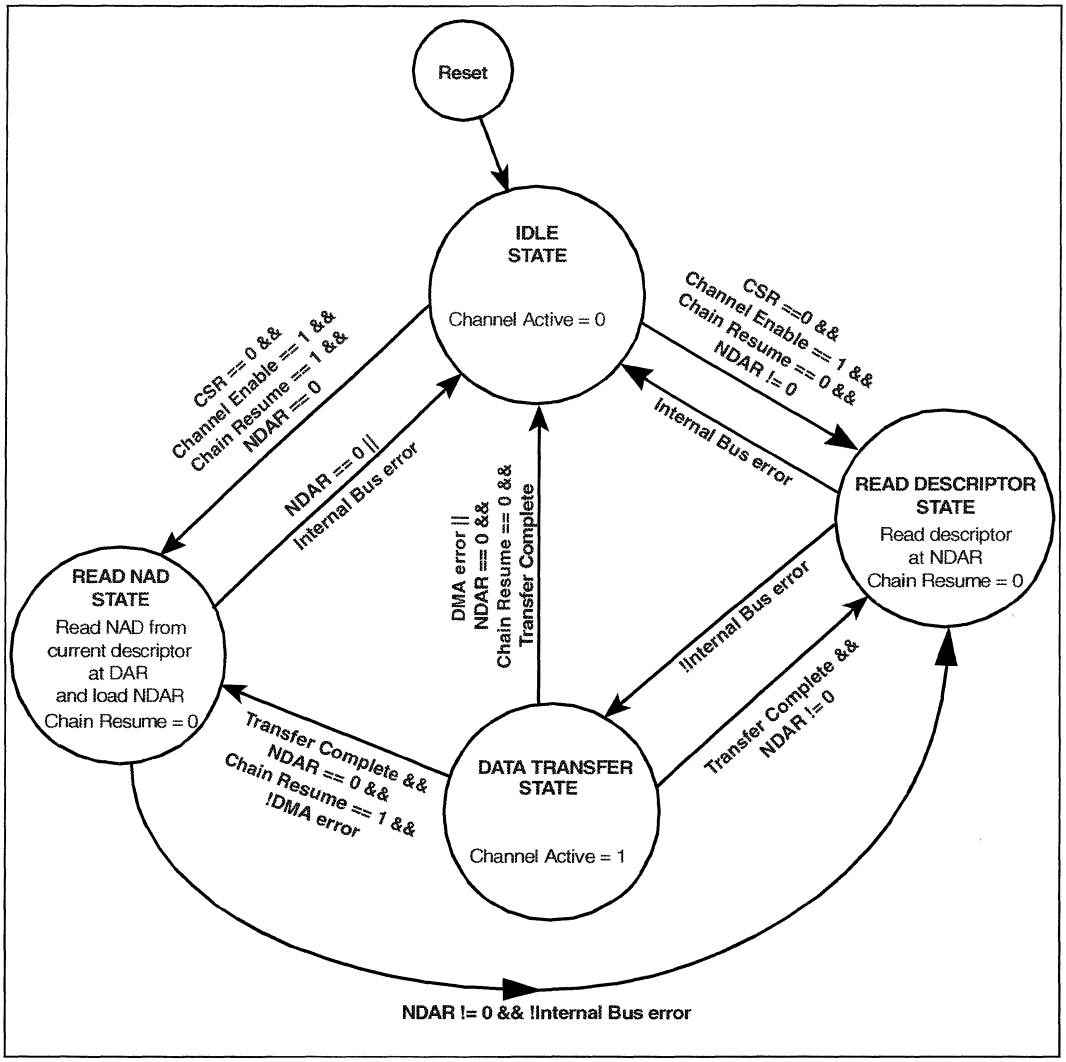
19.8 Channel Priority

The 80960 internal bus arbitration logic determines which internal bus master has access to the internal bus. Each DMA channel has an independent bus Request/Grant signal pair to the internal bus arbitration. Chapter 17, "Intel® 80303 I/O Processor Arbitration" further describes the priority scheme between all the bus masters on the internal bus. Also described is the priority mechanism used between the three DMA channels.

19.9 Programming Model State Diagram

The channel programming model diagram is shown in Figure 19-9. Error condition states are not shown.

Figure 19-9. DMA Programming Model State Diagram



19.10 DMA Channel Programming Examples

The software for the DMA channels falls into the following categories:

- Channel initialization
- Start DMA transfer
- Suspend channel

Examples for each of the software is shown in the following sections as pseudo code flow.

19.10.1 Software DMA Controller Initialization

The DMA Controller is designed to have independent control of the interrupts, enables, and control. The initialization consists of virtually no overhead as shown in Figure 19-10.

Figure 19-10. Software Example for Channel Initialization

```
CCR0 = 0x0000 0000 ; Disable channel
Call setup_channel
```

19.10.2 Software Start DMA Transfer

The DMA channel control register provides independent control per channel based on each time the DMA channel is configured. This provides the greatest flexibility to the applications programmer. The example shown in Figure 19-11 describes the pseudo code for starting a DMA transfer on channel 0.

Figure 19-11. Software Example for DMA Transfer

```
; Set up descriptor in local memory at address d
d.nad = 0           ; No chaining
d.pad = 0x0000F000 ; Source address of 0000F000H
d.puad = 0          ; DAC is not used
d.lad = 0xB0000000 ; Destination address B0000000H
d.bc = 64           ; byte count of 64
d.dc = 0x00000016  ; PCI Memory Read command, DAC disabled,
                    ; Interrupt processor after transfer
; Check for inactive channel & no interrupts pending
if (CSR0 != 0) exit; If channel is not ready, exit
; Start transfer
NDAR0 = &d          ; Set up descriptor address
CCR0 = 0x00000001  ; Set Channel Enable bit to start
```

19.10.3 Software Suspend Channel

The channel may need to be suspended for various reasons. The channel provides the ability to suspend the state of the channel without losing the current status. The channel will resume DMA operation without requiring the software to save the channel configuration. The example shown in Figure 19-12 describes the pseudo code for suspending channel 0.

Figure 19-12. Software Example for Channel Suspend

```
CCR0 = 0x0000 0000 ; Suspend Channel 0

    Channel suspended.....


CCR0 = 0x0000 0001 ; Resume Channel 0
```

19.11 Interrupts

Each channel can generate an interrupt to the i960 core processor. The *Interrupt Enable* bit in the Descriptor Control Register (DCRx.ie) determines whether the channel generates an interrupt upon successful error-free completion of a DMA transfer. Error conditions described in Section 19.12 also generate an interrupt. Each channel has one interrupt output connected to the PCI and Peripheral Interrupt Controller described in Chapter 8, "PCI and Peripheral Interrupt Controller Unit" summarizes the status flags and conditions when interrupts are generated in the Channel Status Register (CSRx).

Table 19-2. DMA Interrupt Summary

Interrupt Condition	Channel Status Register (CSR) Flags							Interrupt Generated?	
	Active	End of Transfer	End of Chain	PCI Master Abort	PCI Target Abort	PCI Parity Error	Internal Bus Error	DCR.ie Set	DCR.ie Clear
Byte count == 0 && NDARx != NULL (End of Transfer)	1	1	0	0	0	0	0	Y	N
Byte Count == 0 && NDARx == NULL (End of Chain)	0	0	1	0	0	0	0	Y	N
PCI Master-Abort	0	0	0	1	0	0	0	Y	Y
PCI Target-Abort	0	0	0	0	1	0	0	Y	Y
PCI Parity Error	0	0	0	0	0	1	0	Y	Y
Internal Bus Error	0	0	0	0	0	0	1	Y	Y

Note: End-of-Transfer and End-of-Chain flags will be set only when DCR.ie = 1. If DCR.ie = 0, then the above flags are always set to 0. End-of-Transfer Interrupt and End-of-Chain Interrupt can only be reported in the CSR if the DMA transfer completed without any reportable errors.The channel shall never report an End-of-Transfer interrupt or End-of-Chain interrupt along with any PCI error conditions. Multiple error conditions may occur and be reported together. Also, because the channel does not stop after reporting the End-of- Transfer Interrupt, internal bus errors may occur before the End-of-Transfer interrupt is acknowledged and cleared.

19.12 Error Conditions

There are four error conditions that may occur during a DMA transfer that are recorded by the channel. All error conditions are reported by setting the appropriate bit in the Channel Status Register (CSR). The DMA controller must satisfy all "retries" even when an error condition occurs on the opposite bus.

The possible error conditions are:

- PCI Master-Abort
- PCI Target-Abort
- PCI Data Parity Error
- Internal Bus Errors

19.12.1 PCI Errors

- If a PCI Master-Abort occurs during a DMA transfer, the channel will set bit 3 in the CSR. The channel will also reflect the error to the Address Translation Units (PATU or SATU depending on which channel was the master while the error occurred). The ATU will in turn, record this error condition by setting the appropriate bit in its status register (PATUSR or SATUSR). Refer to Chapter 15, "PCI Address Translation Unit" for complete details.
- If a PCI Target-Abort (Master) occurs during a DMA transfer, the channel will set bit 2 in the CSR. The channel will also reflect the error to the Address Translation Units (PATU or SATU depending on which channel was the master while the error occurred). The ATU will in turn, record this error condition by setting the appropriate bit in its status register (PATUSR or SATUSR). Refer to Chapter 15, "PCI Address Translation Unit" for complete details.
- If a PCI data parity error occurs during a DMA transfer, the channel will set bit 0 in the CSR. The channel will also reflect the error to the Address Translation Units (PATU or SATU depending on which channel was the master while the error occurred). The ATU will in turn, record this error condition by setting the appropriate bit in its status register (PATUSR or SATUSR). For PCI parity errors, data with incorrect parity is never transferred to local memory. Refer to Chapter 15, "PCI Address Translation Unit" for complete details.

19.12.2 Internal Bus Errors

- If an error occurs during a read of the Chain Descriptor or Next Descriptor Address, the channel may set the Internal Bus Master-abort error flag in the CSR. Then, the channel will load the registers (if possible) and stop.
- If an error occurs when the DMA channel is mastering a transaction on the PCI bus, the channel will prematurely end the transaction and stop transferring data as soon as possible.
- If the channel has asserted its PCI request signal, but not yet started the transaction, the channel will deassert its request.
- If the channel has not yet asserted a request for the PCI bus, the channel will never assert a request for the bus.

When an error condition occurs, the actions taken are detailed below:

- The channel shall cease data transfers for the current chain descriptor and clear the *Channel Active* flag in the CSR.
- The channel will invalidate any data held in the queue and not read any new chain descriptors.
- The channel will set the appropriate error flag in the Channel Status Register. For example; if a PCI Master-Abort occurred during a DMA transfer, the channel will set bit 3 in the CSR. During an MWI transaction, the channel will complete the transfer of the cache line before stopping.
- The channel also signals an interrupt to the i960 core processor.
- The channel will not restart a DMA transfer after any error condition. It is the responsibility of the application software to configure the channel to complete any remaining transfers.

Note: IB errors (**Target-abort only**) that occur while a DMA channel is the master on the internal bus are recorded by the MCU and interrupt the core. For correct operation of the DMA channel, user software has to disable the channel before clearing the error condition. Further, the channel needs to be re-enabled by writing a 1 to CCR.ce before initiating a new operation.

Accesses to the MCU can be 64-bits or smaller. In both cases, there are three possible scenarios for multi-bit ECC errors on reads or writes. These errors conditions are handled as detailed below:

- **Multi-bit ECC error on MCU Data Read**: Refer to Chapter 13, "Memory Controller" for complete details regarding error handling.
- **Multi-bit ECC error on MCU Data Write**: This instance covers the case where the first data write is less than a 64-bit value forcing the MCU to execute a *read-modify-write* operation. Refer to Chapter 13, "Memory Controller" for complete details regarding error handling.
- **Multi-bit ECC error on MCU Data Write**: This instance covers the case where the last data write is less than a 64-bit value forcing the MCU to execute a *read-modify-write* operation. Refer to Chapter 13, "Memory Controller" for complete details regarding error handling.

19.13 Power-up/Default Status

Upon power-up, an external hardware reset, the DMA Registers will be initialized to their default values.

19.14 Register Definitions

The DMA controller contains registers for controlling each channel. Each channel has nine memory-mapped control registers for independent operation. In register titles, x is 0, 1, or 2 for channel 0, 1, or 2 respectively.

There is read/write access only to the Channel Control Register, Channel Status Register, and the Next Descriptor Address Register. The remaining registers are read-only and are loaded with new values from the chain descriptor whenever the channel reads a chain descriptor from memory.

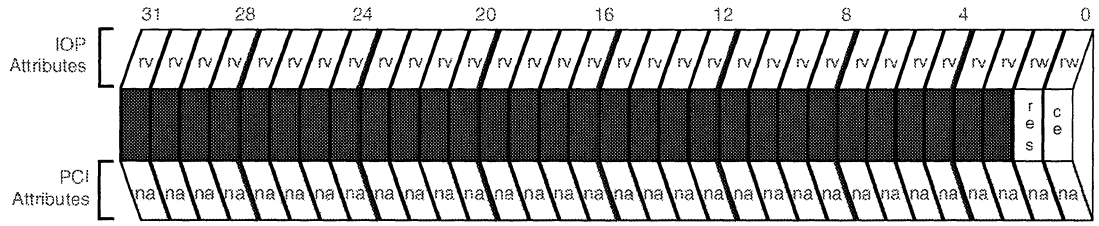
Table 19-3. DMA Controller Unit Registers

Section, Register Name, Acronym (page)
Section 19.14.1, "Channel Control Register - CCR" on page 19-27
Section 19.14.2, "Channel Status Register - CSR" on page 19-28
Section 19.14.3, "Next Descriptor Address Register - NDAR" on page 19-30
Section 19.14.4, "Descriptor Address Register - DAR" on page 19-31
Section 19.14.5, "Byte Count Register - BCR" on page 19-32
Section 19.14.6, "PCI Address Register - PADR" on page 19-33
Section 19.14.7, "PCI Upper Address Register - PUADR" on page 19-34
Section 19.14.8, "Intel® i960® Local Address Register - LADR" on page 19-35
Section 19.14.9, "Descriptor Control Register - DCR" on page 19-36

19.14.1 Channel Control Register - CCR

The Channel Control Register (CCR) specifies parameters that dictate the overall channel operating environment. The CCR should be initialized prior to any other DMA register following a system reset. Figure 19-4 shows the register format for the CCR. This register can be read or written while the DMA channel is active.

Table 19-4. Channel Control Register - CCR



Channel #	Intel® i960® Core internal bus address
0	1400H
1	1440H
2	1480H

Attribute Legend:	
RV = Reserved	RW = Read/Write
PR = Preserved	RC = Read Clear
RS = Read/Set	RO = Read Only
	NA = Not Accessible

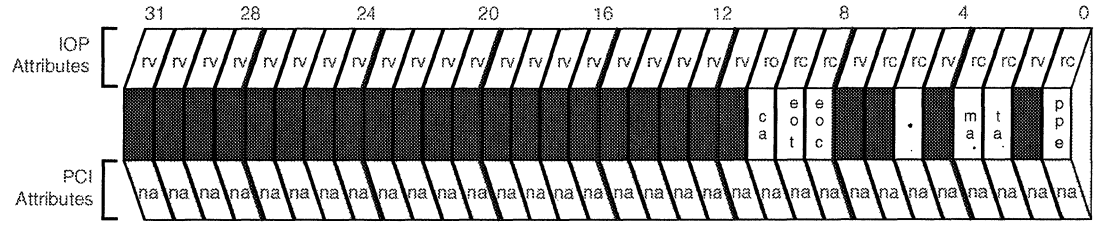
Bit	Default	Description
31:02	0000 0000H	Reserved
01	0_2	Chain Resume - when set, causes the channel to resume chaining by re-reading the current descriptor located at the address in the Descriptor Address Register when the channel is idle (CA bit in the CSR is clear) or when the channel completes a DMA transfer. This bit is cleared by the hardware when either: • The channel completes a DMA transfer and the Next Descriptor Address Register is zero. In this case, the channel proceeds to the next descriptor in the chain. • The channel re-reads the chain descriptor located at the address in the Descriptor Address Register and loads the Next Descriptor Address of that descriptor into the Next Descriptor Address Register
00	0_2	Channel Enable - When set, the channel enables DMA transfers. When clear, the channel disables DMA transfers. Clearing this bit once the channel is active suspends the current DMA transfer at the earliest opportunity by halting all internal bus transactions. The PCI interface may continue with the current transfer until the data queue either fills or empties. The channel does not initiate any new DMA transfers when this bit is cleared. Data held in queues remains valid. Setting this bit after the channel is suspended causes the channel to resume the DMA transfer. The Channel Enable bit works in conjunction with the Bus Master Enable bit of the Primary ATU Command Register for DMA Channel 0 and 1 and with the Bus Master Enable bit of the Secondary ATU Command Register for DMA Channel 2. The respective Bus Master Enable bit must be set for the DMA channel to start a transaction on the PCI bus.

19.14.2 Channel Status Register - CSR

The Channel Status Register (CSR) contains status flags that indicate the channel status. This register is typically read by software to examine the source of an interrupt. See Section 19.12 for a description of the error conditions that are reported in the CSR. See Section 19.11 for a description of interrupts caused by the DMA channel.

If a DMA error occurs, application software must check the status of the Channel Active flag before processing the interrupt. It is possible that the channel may still be active completing outstanding PCI transactions.

Table 19-5. Channel Status Register - CSR (Sheet 1 of 2)

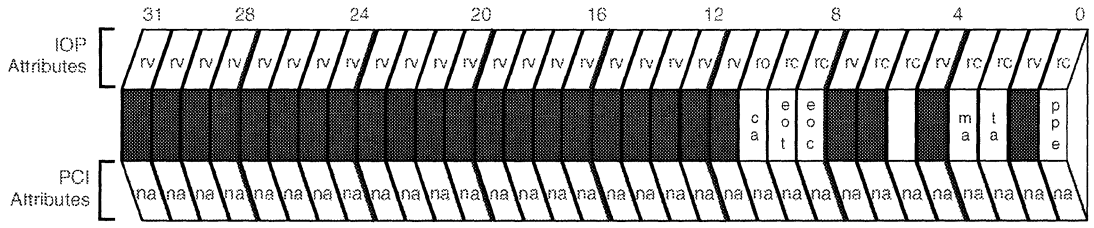


Channel #	Intel® i960® Core internal bus address
0	1404H
1	1444H
2	1484H

Attribute Legend:	
RV = Reserved	RW = Read/Write
PR = Preserved	RC = Read Clear
RS = Read/Set	RO = Read Only
	NA = Not Accessible

Bit	Default	Description
31:11	000000H	Reserved
10	0_2	Channel Active Flag - indicates the channel is either active (in use) or inactive (available). When set, indicates the channel is in use and actively performing DMA data transfers. When clear, indicates the channel is inactive and available to be configured to transfer data. The channel clears the Channel Active flag when the previously configured DMA transfer completes as a result of: • byte count reached zero and last chain descriptor is encountered (NULL value detected for Next Descriptor Address in chain descriptor) • PCI Master-abort occurred on the PCI interface • PCI Target-abort occurred on the PCI interface • PCI parity error occurred on the PCI interface • Internal Bus Errors The Channel Active flag is set when a Chain Descriptor is read from memory.
09	0_2	End of Transfer Interrupt Flag - set when the channel has signalled an interrupt to the Intel® i960® core processor after successfully completing an error-free DMA transfer but it is not the last descriptor in a chain.
08	0_2	End of Chain Interrupt Flag - set when the channel has signalled an interrupt to the i960 core processor after successfully completing an error-free DMA transfer that is the last of a chain.
07:06	0_2	Reserved
05	0_2	Internal Bus Master-Abort Flag - All Master-aborts when the channel is the master on the internal bus will be reflected by setting this bit.
04	0_2	Reserved
03	0_2	PCI Master Abort Flag - set when the channel has initiated a transaction on the PCI bus and has detected a Master-abort.
02	0_2	PCI Target Abort Flag - set when the channel has initiated a transaction on the PCI bus and has detected a Target-abort.

Table 19-5. Channel Status Register - CSR (Sheet 2 of 2)



Channel #	Intel® i960® Core internal bus address
0	1404H
1	1444H
2	1484H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
01	0_2	Reserved
00	0_2	PCI Parity Error Flag - is set when the following conditions are met: • DMA channel asserted PERR# or has observed PERR# asserted • DMA channel was the master for the transaction during which the error occurred

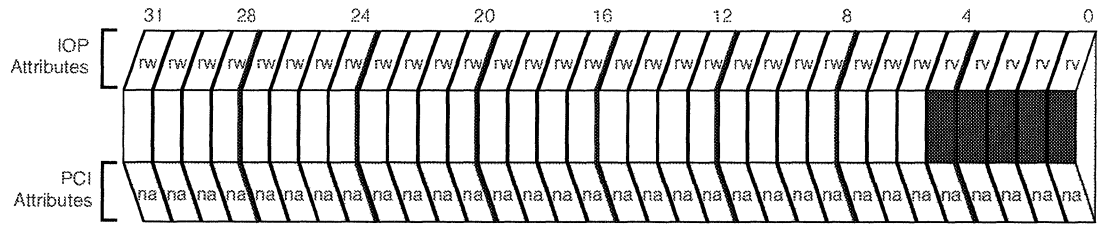
19.14.3 Next Descriptor Address Register - NDAR

The Next Descriptor Address Register (NDARx) contains the address of the next chain descriptor in 80960 local memory for a DMA transfer. When starting a DMA transfer, this register contains the address of the first chain descriptor. Table 19-6 depicts the Next Descriptor Address Register.

All chain descriptors are required to be aligned on an eight 32-bit word boundary. The channel may set bits 04:00 to zero when loading this register.

Note: The *Channel Enable* bit in the CCR and the *Channel Active* bit in the CSR must both be clear prior to writing the Next Descriptor Address Register. Writing a value to this register while the channel is active may result in undefined behavior.

Table 19-6. Next Descriptor Address Register - NDAR



Channel #	Intel® i960® Core internal bus address
0	1410H
1	1450H
2	1490H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

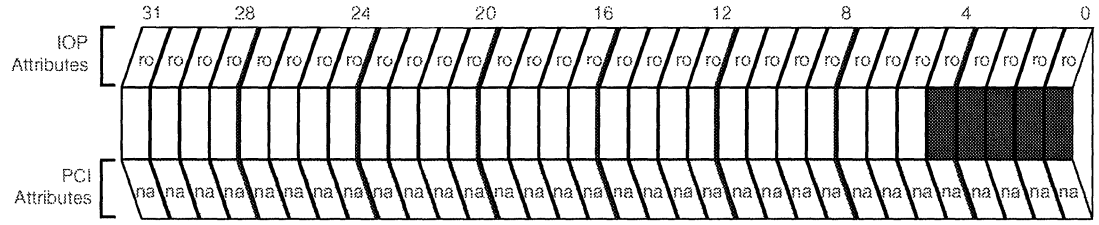
Bit	Default	Description
31:05	00000000000 00000000000 00000_2	Next Descriptor Address - local memory address of the next chain descriptor to be read by the channel.
04:00	00000_2	Reserved

19.14.4 Descriptor Address Register - DAR

The Descriptor Address Register (DARx) contains the address of the current chain descriptor in 80960 local memory for a DMA transfer. This register read-only and is loaded when a new chain descriptor is read. Table 19-7 depicts the Descriptor Address Register.

All chain descriptors are aligned on an eight 32-bit word boundary

Table 19-7. Descriptor Address Register - DAR



Channel #	Intel® i960® Core internal bus address
0	140CH
1	144CH
2	148CH

Attribute Legend:	
RV = Reserved	RW = Read/Write
PR = Preserved	RC = Read Clear
RS = Read/Set	RO = Read Only
	NA = Not Accessible

Bit	Default	Description
31:05	00000000000 00000000000 00000_2	Current Descriptor Address - local memory address of the current chain descriptor that was read by the channel.
04:00	00000_2	Reserved

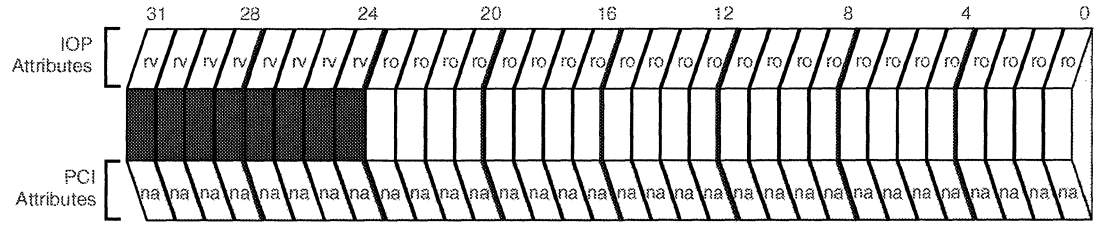
19.14.5 Byte Count Register - BCR

The Byte Count Register (BCRx) contains the number of bytes to transfer for a DMA transfer. This is a read-only register that is loaded from the Byte Count word in a chain descriptor. It allows for a maximum DMA transfer of 16 Mbytes. A value of zero is a valid byte count and results in no data words being transferred and no cycles generated on either the PCI bus or the internal bus.

Anytime this register is read by the i960 core processor, it contains the number of bytes left to transfer on the internal bus. Note that valid data may be in the channel's data queue. This register is decremented by 1 through 8 bytes for every successful transfer from the source to destination locations. Table 19-8 shows the Byte Count Register

Note: The byte count value is not required to be aligned to a 32-bit word boundary.

Table 19-8. Byte Count Register - BCR



Channel #	Intel® i960® Core internal bus address
0	1420H
1	1460H
2	14A0H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:24	00H	Reserved
23:00	000000H	Byte Count - is the number of bytes to transfer for a DMA transfer.

19.14.6 PCI Address Register - PADR

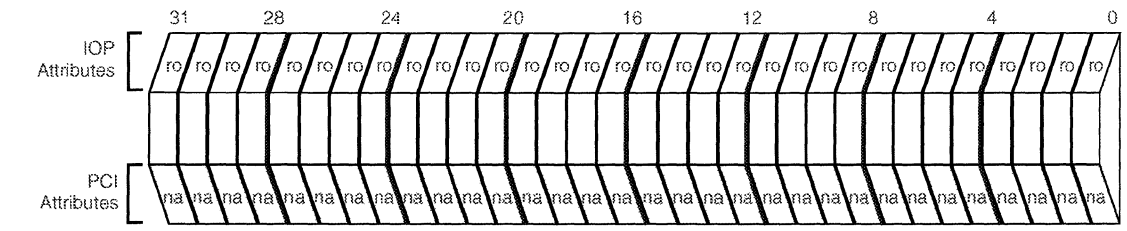
The PCI Address Register (PADR) contains the 32-bit PCI address for SAC cycles or the lower 32-bit PCI address of a 64-bit PCI address for DAC cycles. This address is the source or destination of the DMA transfer. This register is read-only and is loaded when a chain descriptor is read from memory.

Table 19-9 shows the PCI Address Register.

The channel will drive the **P_AD[1:0]** or **S_AD[1:0]** to a value of 00_2 indicating linear addressing. Refer to the PCI internal bus specification for additional information.

Note: The application programmer must not program the channel to transfer data across a 4 Gbyte boundary (i.e., the lower 32-bit address must not increment past the maximum address of FFFF.FFFFH). The channel will not notify the application of this condition.

Table 19-9. PCI Address Register - PADR



Channel #	Intel® i960® Core internal bus address
0	1414H
1	1454H
2	1494H

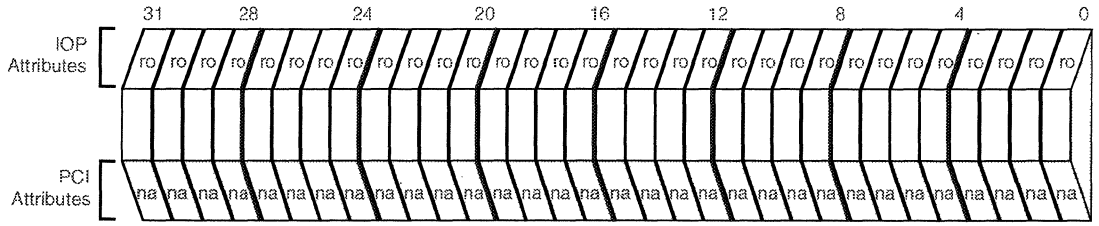
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:00	00000000H	PCI Address - is the PCI source/destination address.

19.14.7 PCI Upper Address Register - PUADR

The PCI Upper Address Register (PUADRx) contains the upper 32-bit address of a 64-bit address. Table 19-10 shows the register. This register is read-only and is loaded when a chain descriptor is read from memory

Table 19-10. PCI Upper Address Register - PUADR



Channel #	Intel® i960® Core internal bus address
0	1418H
1	1458H
2	1498H

Attribute Legend:	
RV = Reserved	RW = Read/Write
PR = Preserved	RC = Read Clear
RS = Read/Set	RO = Read Only
	NA = Not Accessible

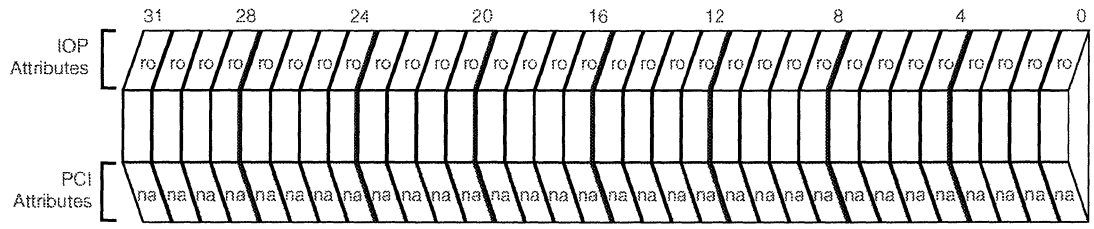
Bit	Default	Description
31:00	00000000H	PCI Upper Address - is the PCI source/destination upper address.

19.14.8 Intel® i960® Local Address Register - LADR

The 80960 Local Address Register (LADRx) contains the 32-bit 80960 local address. The 80960 address space is a 32-bit, byte addressable address space. Table 19-11 shows the 80960 Local Address Register. This read-only register is loaded when a chain descriptor is read from memory.

Note: Access to the Peripheral Memory-Mapped Registers through a DMA transfer is not allowed. The LADRx should not be programmed with values less than 0000 2000H, as this address space is reserved. The hardware must ensure that all internal bus accesses to this address space are properly terminated

Table 19-11. Intel® i960® Local Address Register - LADR



Channel #	Intel® i960® Core internal bus address
0	141CH
1	145CH
2	149CH

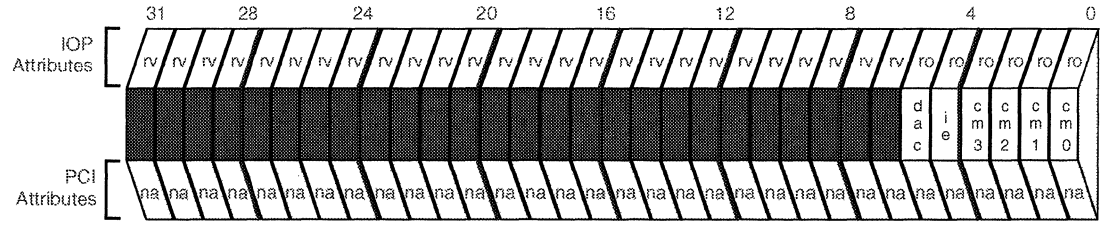
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:00	00000000H	Intel® i960® Local Address - the 80960 local source/destination address.

19.14.9 Descriptor Control Register - DCR

The Descriptor Control Register contains control values for the DMA transfer on a per-chain descriptor basis. This read-only register is loaded whenever a chain descriptor is read from memory. These values may vary from chain descriptor to chain descriptor. Table 19-12 shows the definition of the Descriptor Control Register.

Table 19-12. Descriptor Control Register - DCR



Channel #	Intel® i960® Core internal bus address
0	1424H
1	1464H
2	14A4H

Attribute Legend:	
RV = Reserved	RW = Read/Write
PR = Preserved	RC = Read Clear
RS = Read/Set	RO = Read Only
	NA = Not Accessible

Bit	Default	Description
31:06	000000H	Reserved
05	0_2	Dual Address Cycle Enable - determines the address cycle type generated on the PCI bus. When set, the channel uses Dual Address Cycle (DAC) to transfer a 64-bit address. When clear, the channel uses Single Address Cycle (SAC) to transfer a 32-bit address. For DAC, the PCI Address Register (PADRx) contains the lower 32-bit address used on the first address cycle. The PCI Upper Address Register (PUADRx) contains the upper 32 bits address cycle used on the second address cycle. The upper 32 bit address of a DAC transaction is required to be non-zero. Refer to Section 19.4 for details on 64-bit addressing.
04	0_2	Interrupt Enable - when set, the channel generates an interrupt to the Intel® 80303 I/O processor upon completion of a DMA transfer. When clear, no interrupt is generated.
03:00	0000_2	PCI Command - determines PCI bus command type for this DMA transfer. This value is used directly for the PCI bus command; e.g., when PCI Command is 0000_2, the PCI Command is 0000_2, a reserved command type. See Table 19-13. Hardware does not check for reserved or unsupported command types.

19.14.9.1 PCI Commands Support

The Memory Write and Invalidate command is fully supported by all channels of the DMA controller. Refer to Section 19.5.3, "Local Memory to PCI Transfers: Memory Write and Invalidate Command" on page 19-15 for a complete description of the behavior of the DMA channel during this PCI bus cycle.

Table 19-13. PCI Commands

C/BE[3:0]#	PCI Command Type	Description
0000_2	Intack	Not Supported
0001_2	SpCyc	Not Supported
0010_2	I/O Read	Not Supported
0011_2	I/O Write	Not Supported
0100_2	reserved	Not Supported
0101_2	reserved	Not Supported
0110_2	Memory Read	Memory Read of less than one cacheline
0111_2	Memory Write	Memory Write
1000_2	reserved	Not Supported
1001_2	reserved	Not Supported
1010_2	Configuration Read	Not Supported
1011_2	Configuration Write	Not Supported
1100_2	Memory Read Multiple	Memory Read of more than one cacheline
1101_2	reserved	Not Supported
1110_2	Memory Read Line	Memory Read of one cacheline
1111_2	Memory Write and Invalidate	Memory Write which guarantees the transfer of complete cache line (s) during the current transaction

Application Accelerator Unit 20

This chapter describes the integrated Application Accelerator (AAU) Unit. The operation modes, setup, external interface, and implementation of the AAU are detailed in this chapter.

20.1 Overview

The Application Accelerator provides low-latency, high-throughput data transfer capability between the AAU and Intel® 80303 I/O processor local memory. It executes data transfers to and from 80303 local memory and also provides the necessary programming interface. The Application Accelerator performs the following functions:

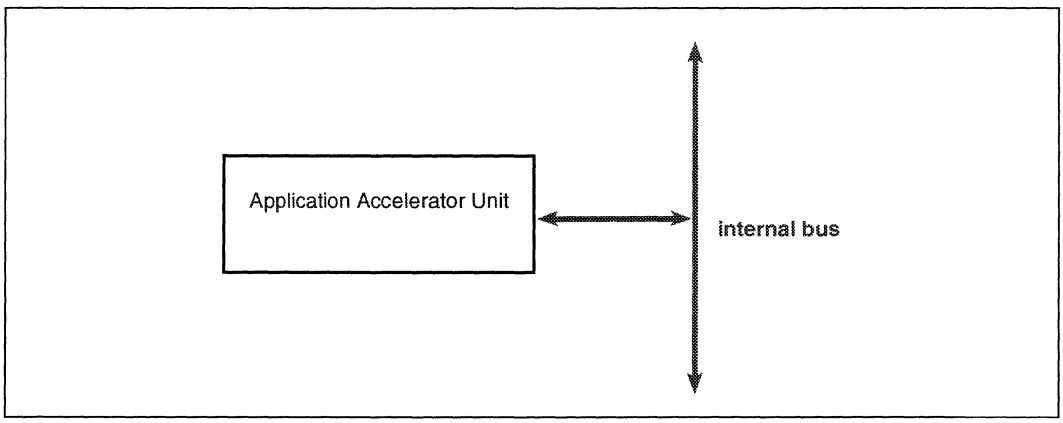
- Transfers data (read) from memory controller.
- Performs an optional boolean operation (XOR) on read data.
- Transfers data (write) to memory controller.

The AAU features:

- 1K-byte, arranged as 8-byte x 128-deep store queue.
  - — Configurable to a 512-byte, arranged as 8-byte x 64-deep store queue.
- Utilization of the 80303 I/O processor memory controller Interface.
- 2^{32} addressing range on the Intel® 80960 local memory interface.
- Hardware support for unaligned data transfers for the internal bus.
- Fully programmable from the Intel® i960® core processor.
- Support for automatic data chaining for gathering and scattering of data blocks.

Figure 20-1 shows a simplified connection of the Application Accelerator to the 80303 I/O processor internal bus.

Figure 20-1. Application Accelerator Unit

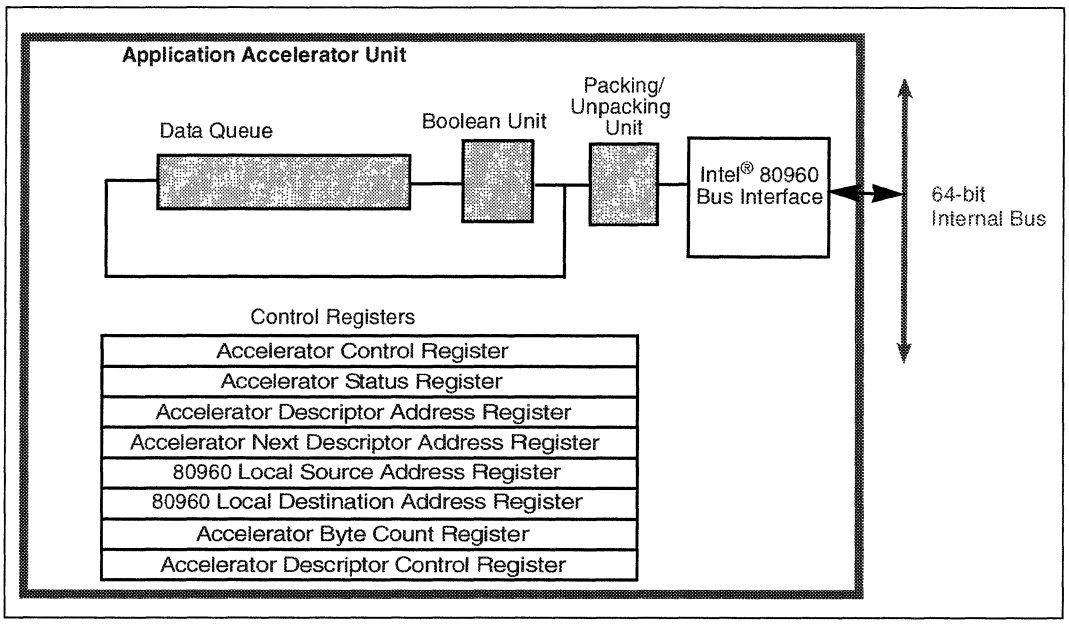


20.2 Theory of Operation

The Application Accelerator is a master on the internal bus and performs data transfers to and from local memory. It does not interface to either the primary PCI or secondary PCI bus. The AAU uses direct addressing for the memory controller.

The Application Accelerator Unit implements the XOR algorithm in hardware. It performs the XOR operation on multiple blocks of source (incoming) data and stores the result back in 80960 local memory. The source and destination addresses are specified through chain descriptors resident in 80960 local memory. Figure 20-2 shows the block diagram of the AAU. The AAU can also be used to perform memory-to-memory transfers of data blocks controlled by the 80303 I/O processor memory controller unit.

Figure 20-2. Application Accelerator Block Diagram



The Application Accelerator programming interface is accessible from the internal bus through a memory-mapped register interface. Data for the XOR operation is configured by writing the source addresses, destination address, number of bytes to transfer, and various control information into a chain descriptor in local memory. Chain descriptors are described in detail in Section 20.3.2, "Chain Descriptor Format (Four Source Addresses)" on page 20-4.

The Application Accelerator unit contains a hardware data packing and unpacking unit. This unit enables data transfers from and to unaligned addresses in 80960 local memory. All combinations of unaligned data are supported with the packing and unpacking unit. Data is held internally in the Application Accelerator until ready to be stored back to local memory. This is done using a 1K-byte holding queue (arranged as an 8-byte x 128-deep queue). Data to be written back to 80960 local memory can either be aligned or unaligned.

Each chain descriptor contains the necessary information for initiating an XOR operation on blocks of data specified by the source addresses. The Application Accelerator unit supports chaining. Chain descriptors that specify the source data to be XORed can be linked together in 80960 local memory to form a linked list.

20.3 Hardware-Assist XOR Unit

The Application Accelerator Unit implements the XOR algorithm in hardware. It performs the XOR operation on multiple blocks of source (incoming) data and stores the result back in 80960 local memory.

- The process of reading source data, executing the XOR algorithm, and storing the XOR data will hereafter be referred to as *XOR-transfer*.
- The process of reading or writing data will hereafter be referred to as *data transfer*.

The source and destination addresses are specified through chain descriptors resident in 80960 local memory.

20.3.1 Data Transfer

All transfers are configured and initiated through a set of memory-mapped registers and one or more chain descriptors located in local memory. A transfer is defined by the source address, destination address, number of bytes to transfer, and control values. These values are loaded in the chain descriptor before a transfer begins. Table 20-1 describes the registers that need to be configured for any operation.

Table 20-1. Register Description

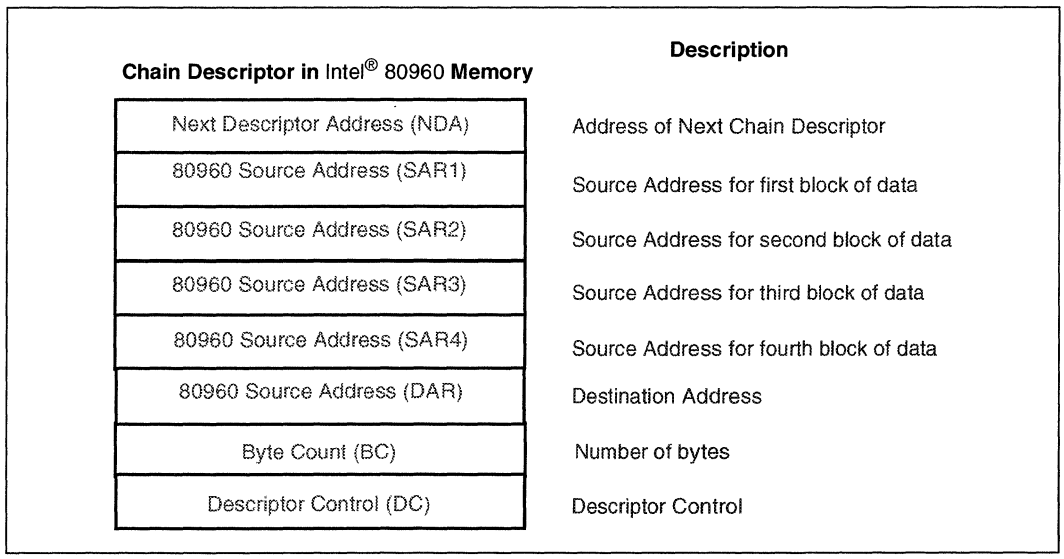
Register	Abbreviation	Description
Accelerator Control Register	ACR	Application Accelerator Control Word
Accelerator Status Register	ASR	Application Accelerator Status Word
Accelerator Descriptor Address Register	ADAR	Address of Current Chain Descriptor
Accelerator Next Descriptor Address Register	ANDAR	Address of Next Chain Descriptor
Source Address Register	SAR1.. SAR8	Local memory addresses of source data
Destination Address Register	DAR	Local memory address of destination data
Accelerator Byte Count Register	ABCR	Number of Bytes to transfer
Accelerator Descriptor Control Register	ADCR	Chain Descriptor Control Word

20.3.2 Chain Descriptor Format (Four Source Addresses)

All transfers are controlled by chain descriptors located in local memory. A chain descriptor contains the necessary information to complete one transfer. A single transfer has only one chain descriptor in memory. Chain descriptors can be linked together to form more complex operations.

To perform a transfer, one or more chain descriptors must first be written to 80960 local memory. Figure 20-3 shows the format of an individual chain descriptor. Every descriptor requires eight contiguous words in 80960 local memory and is required to be aligned on an 8-word boundary. All eight words are required.

Figure 20-3. Chain Descriptor Format



Each word in the chain descriptor is analogous to control register values. Bit definitions for the words in the chain descriptor are the same as for the control registers.

- The first word is the 80960 local memory address of the next chain descriptor. A value of zero specifies the end of the chain. This value is loaded into the Accelerator Next Descriptor Address Register. Because chain descriptors must be aligned on an 8-word boundary, the unit may ignore bits 04:00 of this address.
- The second word is the address of the first block of data resident in 80960 local memory. This address will be driven on the internal bus. This value is loaded into the Source Address Register 1.
- The third word is the address of the second block of data resident in 80960 local memory. This address will be driven on the internal bus. This value is loaded into the Source Address Register 2.
- The fourth word is the address of the third block of data resident in 80960 local memory. This address will be driven on the internal bus. This value is loaded into the Source Address Register 3.
- The fifth word is the address of the fourth block of data resident in 80960 local memory. This address will be driven on the internal bus. This value is loaded into the Source Address Register 4.

- The sixth word is the destination address where data will be stored in 80960 local memory. This address will be driven on the internal bus. This value is loaded into the Destination Address Register.
- The seventh word is the Byte Count value. This value specifies the number of bytes of data in the current chain descriptor. This value is loaded into the Accelerator Byte Count Register.
- The eighth word is the Descriptor Control Word. This word configures the Application Accelerator for one operation. This value is loaded into the Accelerator Descriptor Control Register.

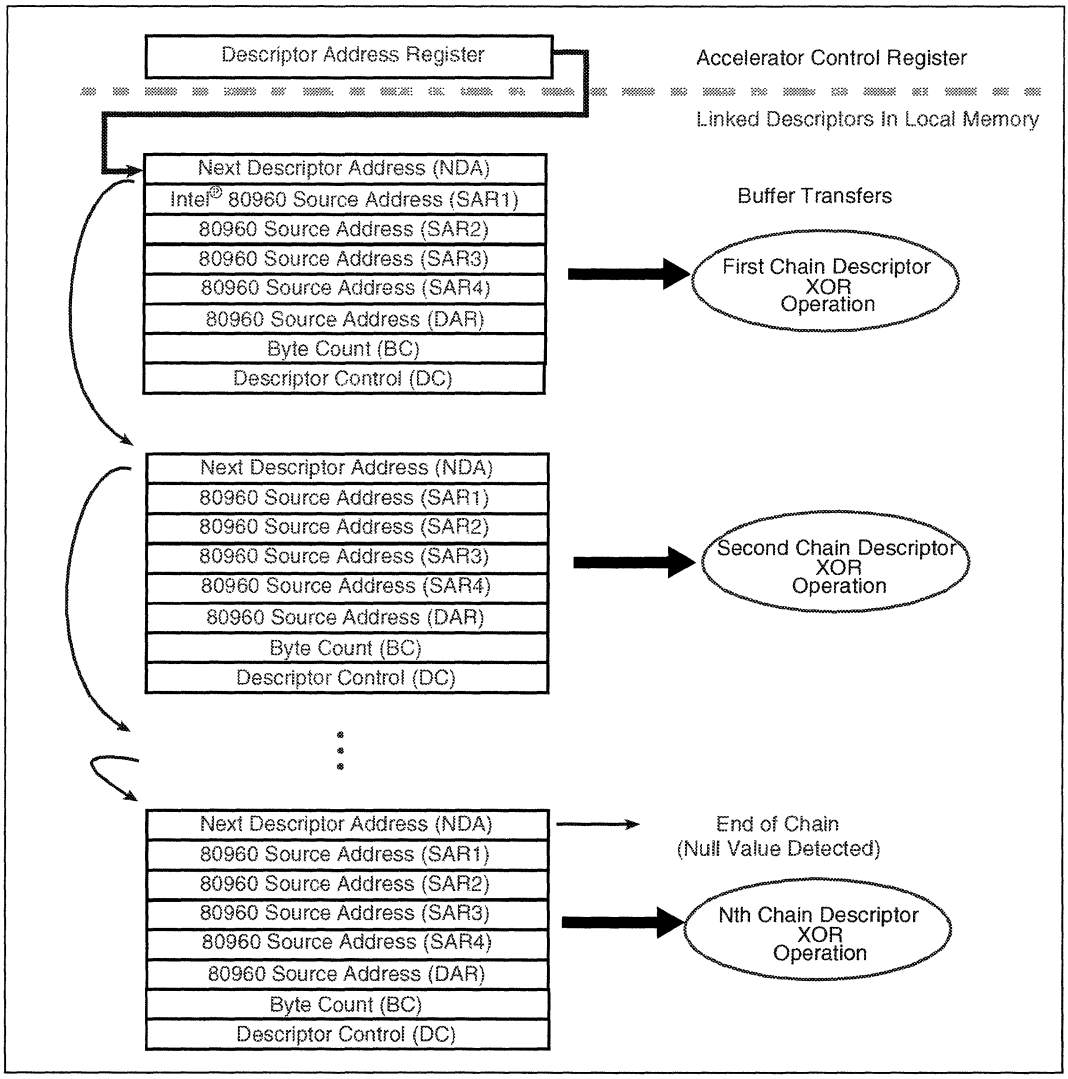
There are no data alignment requirements for any of the source addresses or the destination address. However, maximum performance is obtained from aligned transfers, especially small transfers. See Section 20.4.

Refer to Section 20.11 for additional description on the control registers.

To perform an *XOR-transfer*, a series of chain descriptors can be built in local memory to XOR multiple blocks of source data resident in 80960 local memory. The XOR-ed result is then stored back in 80960 local memory. An application can build multiple chain descriptors to XOR many blocks of data which have different source addresses within the local memory.

When multiple chain descriptors are built in 80960 local memory, the application can link each of these chain descriptors using the Next Descriptor Address in the chain descriptor. This address logically links the chain descriptors together. This allows the application to build a list of transfers which may not require the processor until all transfers are complete. Figure 20-4 shows an example of a linked-list of transfers specified in external memory.

Figure 20-4. XOR Chaining Operation



20.3.3 Chain Descriptor Format (Eight Source Addresses)

To perform an *XOR-transfer* with more than 4 source blocks of data (up to 8), a special chain descriptor needs to be configured:

- The first part (***principal-descriptor***) contains the address of the first 4 source data blocks along with other information.
- The second part (***mini-descriptor***) contains 4, 32-bit words containing the address of the additional four (SAR5 - SAR8) source data blocks. The mini-descriptor is written to a contiguous address immediately following the principal descriptor.

To perform a transfer, both parts (principal and mini-descriptor) must be written to 80960 local memory. Figure 20-5 shows the format of this configuration. Every descriptor requires twelve contiguous words in 80960 local memory and is required to be aligned on an 8-word boundary. All twelve words are required.

Figure 20-5. Chain Descriptor Format for Eight Source Addresses (XOR Function)

Chain Descriptor in Intel® 80960	Description
Next Descriptor Address (NDA)	Address of Next Chain Descriptor
80960 Source Address (SAR1)	Source Address for first block of data
80960 Source Address (SAR2)	Source Address for second block of data
80960 Source Address (SAR3)	Source Address for third block of data
80960 Source Address (SAR4)	Source Address for fourth block of data
80960 Source Address (DAR)	Destination Address of XOR-ed data
Byte Count (BC)	Number of bytes to XOR
Descriptor Control (DC)	Descriptor Control
80960 Source Address (SAR5)	Source Address for fifth data block
80960 Source Address (SAR6)	Source Address for sixth data block
80960 Source Address (SAR7)	Source Address for seventh data block
80960 Source Address (SAR8)	Source Address for eighth data block

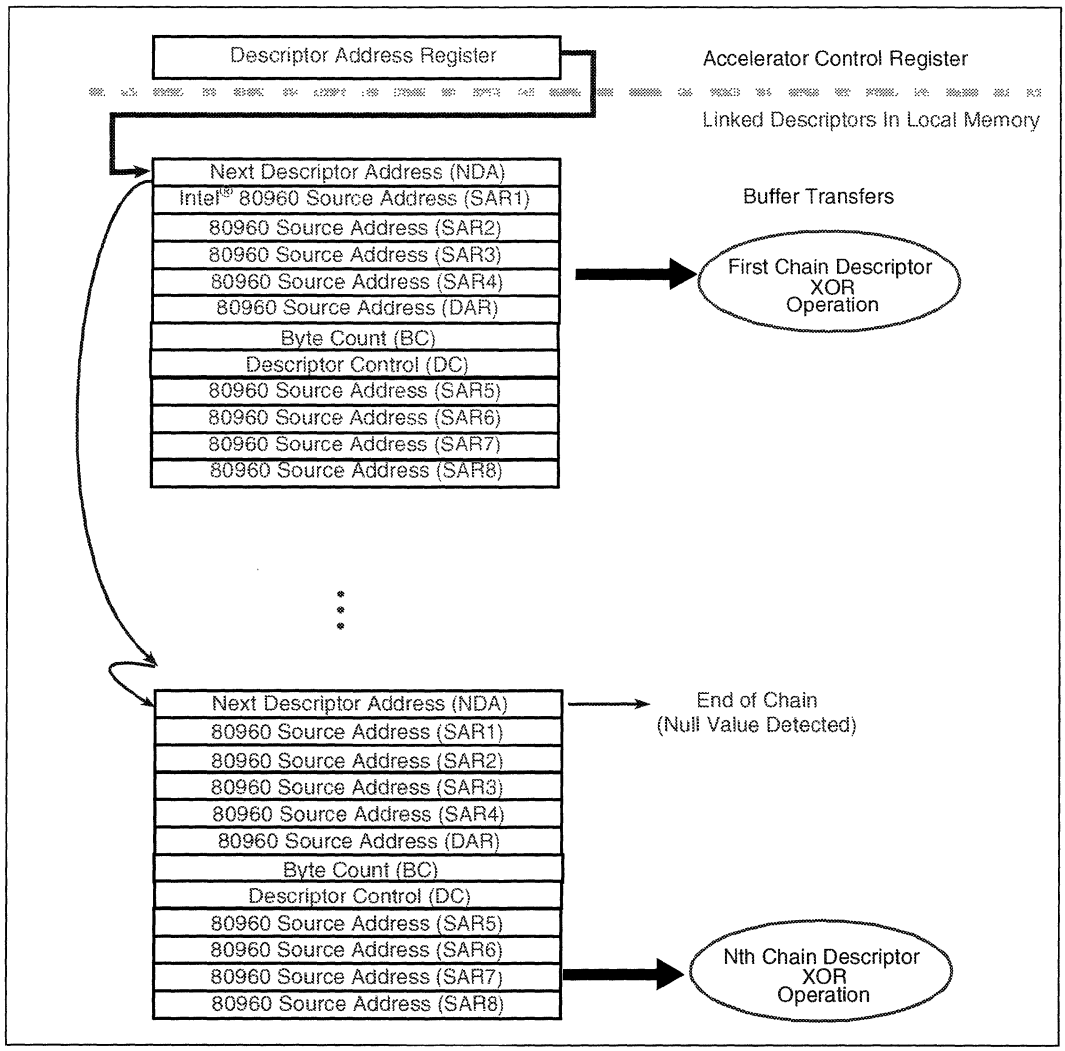
- The first word is the 80960 local memory address of the next chain descriptor. A value of zero specifies the end of the chain. This value is loaded into the Accelerator Next Descriptor Address Register. Because chain descriptors must be aligned on an 8-word boundary, the unit may ignore bits 04:00 of this address.
- The second word is the address of the first block of data resident in 80960 local memory. This address will be driven on the internal bus. This value is loaded into SAR1.
- The third word is the address of the second block of data resident in 80960 local memory. This address will be driven on the internal bus. This value is loaded into SAR2.
- The fourth word is the address of the third block of data resident in 80960 local memory. This address will be driven on the internal bus. This value is loaded into SAR3.
- The fifth word is the address of the fourth block of data resident in 80960 local memory. This address will be driven on the internal bus. This value is loaded into SAR4.

- The sixth word is the destination address where the XOR-ed data will be stored in 80960 local memory. This address will be driven on the internal bus. This value is loaded into the Destination Address Register.
- The seventh word is the Byte Count value. This value determines the total number of bytes of data to XOR in the current chain descriptor. This value is loaded into the Accelerator Byte Count Register.
- The eighth word is the Descriptor Control Word. This word configures the Application Accelerator for one operation. This value is loaded into the Accelerator Descriptor Control Register.
- The ninth word (1st word of mini-descriptor) is the address of the fifth block of data resident in 80960 local memory. This address will be driven on the internal bus. This value is loaded into SAR5.
- The tenth word (2nd word of mini-descriptor) is the address of the sixth block of data resident in 80960 local memory. This address will be driven on the internal bus. This value is loaded into SAR6.
- The eleventh word (3rd word of mini-descriptor) is the address of the seventh block of data resident in 80960 local memory. This address will be driven on the internal bus. This value is loaded SAR7.
- The twelfth word (4th word of mini-descriptor) is the address of the eighth block of data resident in 80960 local memory. This address will be driven on the internal bus. This value is loaded into SAR 8.

A series of chain descriptors can be built in local memory to XOR multiple blocks of source data resident in 80960 local memory. The XOR-ed result is then stored back in 80960 local memory. An application can build multiple chain descriptors to XOR many blocks of data which have different source addresses within the local memory.

When multiple chain descriptors are built in 80960 local memory, the application can link each of these chain descriptors using the Next Descriptor Address in the chain descriptor. This address logically links the chain descriptors together. This allows the application to build a list of transfers which may not require the processor until all transfers are complete. Figure 20-6 shows an example of a linked-list of transfers specified in external memory.

Figure 20-6. XOR Chaining Operation



20.3.4 The Bitwise-XOR Algorithm

Figure 20-7 describes the XOR algorithm implementation. In this illustrative example, there are four blocks of source data to be XOR-ed. The intermediate result is kept by the store queue in the AAU before being written back to 80960 local memory. The source data is located at addresses A000 0400H, A000 0800H, A000 0C00H and A000 1000H respectively.

All data transfers needed for this operation are controlled by chain descriptors located in local memory. The Application Accelerator as a master on the internal bus initiates data transfer. The algorithm is implemented such that as data is read from local memory, the boolean unit executes the XOR operation on incoming data.

Figure 20-7. The Bit-wise XOR Algorithm

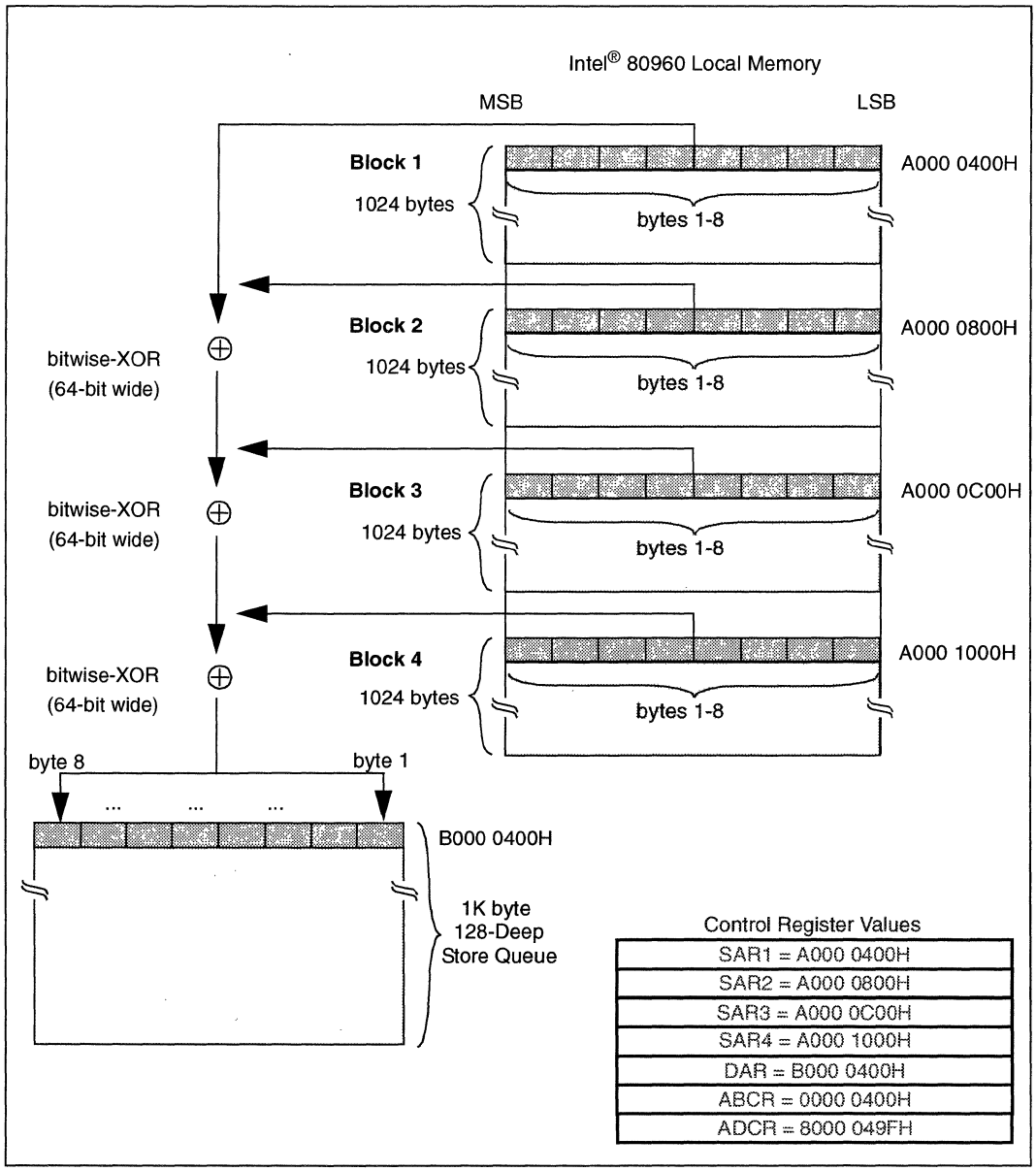
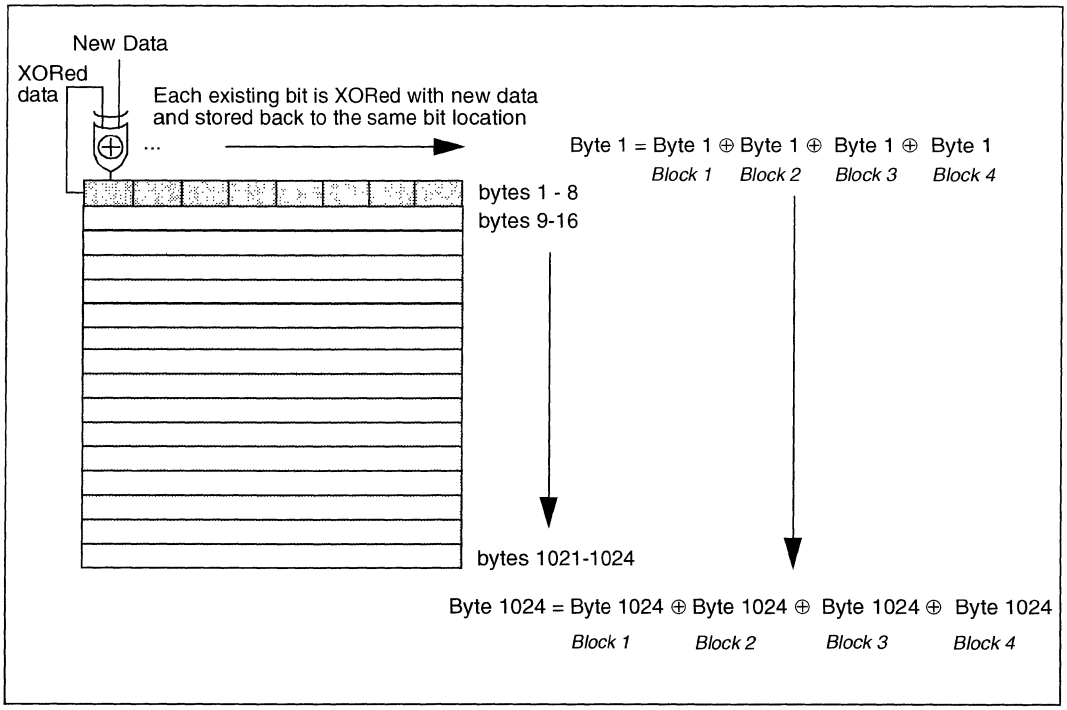


Figure 20-8. Hardware Assist XOR Unit



The XOR algorithm and methodology followed once a chain descriptor has been configured is detailed below:

1. The Application Accelerator as a master on the bus initiates data transfer from the address pointed at by the First Source Address Register (SAR1). As this is the first transfer in the current chain descriptor, the data is transferred directly to the store queue. The number of bytes transferred to the store queue is 1K (or 512 Bytes if the 512 Byte Buffer Enable Bit in the ACR register is set). The total number of bytes to *XOR-transfer* is specified by the Byte Count (BC) field in the chain descriptor.

Note: The Application Accelerator operates on a buffer of 1K or 512 bytes of data at a time depending on the value of the 512 Byte Buffer Enable bit. If the Byte Count Register contains a value greater than the buffer size, the AAU completes the *XOR-transfer* operation on the first buffer of data obtained from each Source Register (SAR1 - SAR4), then proceeds with the next buffer of data. This process is repeated until the BCR contains a zero value.

2. The Application Accelerator transfers the first eight bytes of data from the address pointed at by the Second Source Address Register (SAR2).
3. The boolean unit performs the bit-wise XOR algorithm on the input operands. The input operands are the first eight bytes of data read from SAR1 (bytes 1-8) which are stored in the queue and the first eight bytes of data just read from SAR2 (bytes 1-8).
4. The XOR-ed result is transferred to the store queue and stored in the first eight bytes (bytes 1-8) overwriting previously stored data.
5. The Application Accelerator transfers the next eight bytes of data (bytes 9-16) from address pointed at by the Second Source Address Register (SAR2).

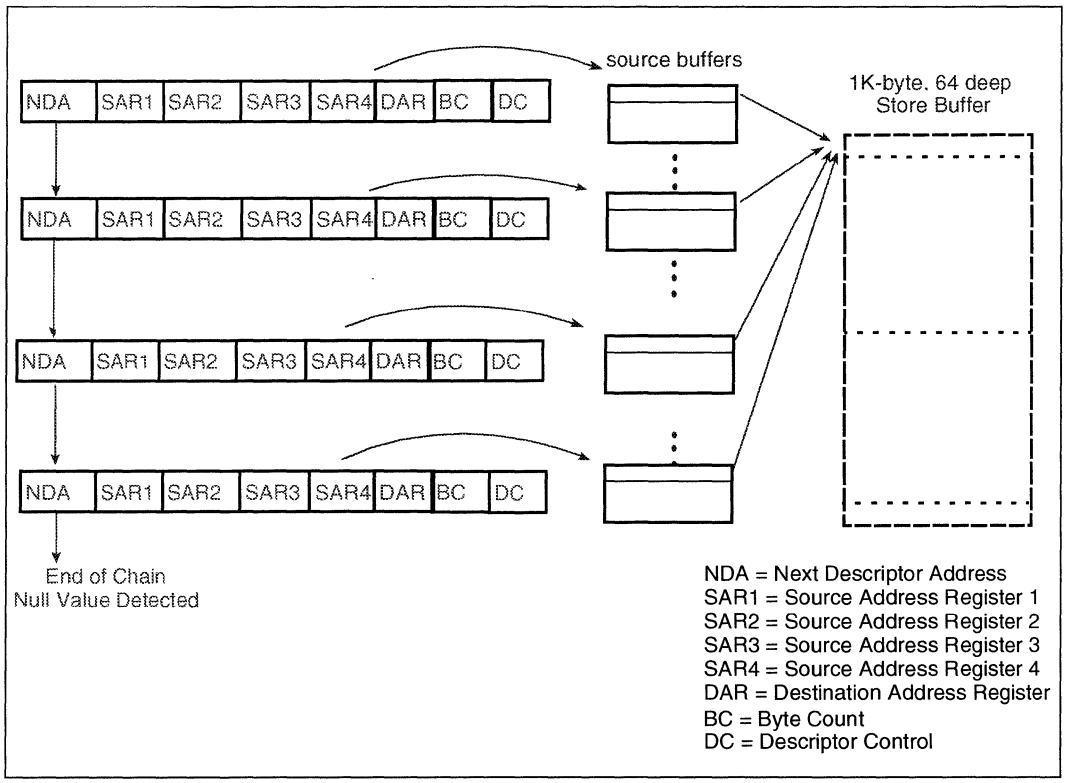
6. The boolean unit performs the bit-wise XOR algorithm on the input operands. The input operands are the next eight bytes of data read from SAR1 (bytes 9-16 stored in the queue) and the eight bytes of data read from SAR2 in Step-5.
7. Step-5 and Step-6 (Data transfer & XOR) are repeated until all data pointed at by SAR1 is XOR-ed with the corresponding data pointed at by SAR2. The store queue now contains a buffer full of XOR-ed data, the source addresses for which were specified in SAR1 and SAR2.
8. Steps 1-7 are repeated once again. The first input to the XOR unit is the data held in the store queue and the second input is the data pointed at by SAR3.
9. The above steps are repeated once more. The first input to the XOR unit is the data held in the store queue and the second input is the data pointed at by SAR4.
10. Once Steps 1-9 are completed, the XOR operation is complete for the first full buffer of the current chain descriptor. If the Destination Write Enable Bit in the Accelerator Descriptor Control Register (ADCR) is set, the data in the store queue is written to local memory at the address pointed to by the Destination Address Register (DAR). If the Destination Write Enable Bit in the ADCR is not set, the data is not written to local memory and is held in the queue. Steps 1-9 are repeated until all the bytes of data have undergone the *XOR-transfer* operation.

Note: If the ABCR register contains a value greater than the buffer size and the ADCR.dwe bit is cleared, the AAU will only read the first buffer of data and perform the specified function. It will not read the remaining bytes specified in the ABCR. Further, the AAU will proceed to process the next chain descriptor if it is specified.

20.3.5 Initiating the XOR Operation

An XOR operation is initiated by building one or more chain descriptors in 80960 local memory. Figure 20-9 shows the format of a principal descriptor.

Figure 20-9. Example of Gather Chaining for Four Source Blocks



The following describes the steps for initiating a new XOR operation:

1. The AAU must be inactive prior to starting an XOR operation. This can be checked by software by reading the *Accelerator Active* bit in the Accelerator Status Register. If this bit is clear, the unit is inactive. If this bit is set, the unit is currently active.
2. The ASR must be cleared of all error conditions.
3. The software writes the address of the first chain descriptor to the Accelerator Next Descriptor Address Register (ANDAR).
4. The software sets the *Accelerator Enable* bit in the Accelerator Control Register (ACR). Because this is the start of a new XOR operation and not the resumption of a previous operation, the *XOR Resume* bit in the ACR should be clear.
5. The AAU starts the XOR operation by reading the chain descriptor at the address contained in ANDAR. The AAU loads the chain descriptor values into the ADAR and begins data transfer. The Accelerator Descriptor Address Register (ADAR) contains the address of the chain descriptor just read and ANDAR will now contain the Next Descriptor Address from the chain descriptor just read.

The last descriptor in the XOR chain list will have zero in the next descriptor address field specifying the last chain descriptor. A NULL value notifies the AAU not to read additional chain descriptors from memory.

Once an XOR operation is active, it can be temporarily suspended by clearing the *Accelerator Enable* bit in the ACR. Note that this does not abort the XOR operation. The unit resumes the process when the *Accelerator Enable* bit is set.

When descriptors are read from external memory, bus latency and memory speed affect chaining latency. Chaining latency is defined as the time required for the AAU to access the next chain descriptor plus the time required to set up the next *XOR-transfer*.

20.3.6 Scatter Gather Transfers

The Application Accelerator can be used to perform typical scatter gather transfers. This consists of programming the chain descriptors to gather data which may be located in non-contiguous blocks of memory. The chain descriptor will specify the destination location such that once all data has been processed, the data will be contiguous in memory. Figure 20-9 shows how the destination pointers can gather data.

20.3.7 Synchronizing a Program to Chained Operation

Any operation involving the AAU can be synchronized to a program executing on the i960® core processor through the use of processor interrupts. The AAU will generate an interrupt to the i960® core processor under certain conditions. They are:

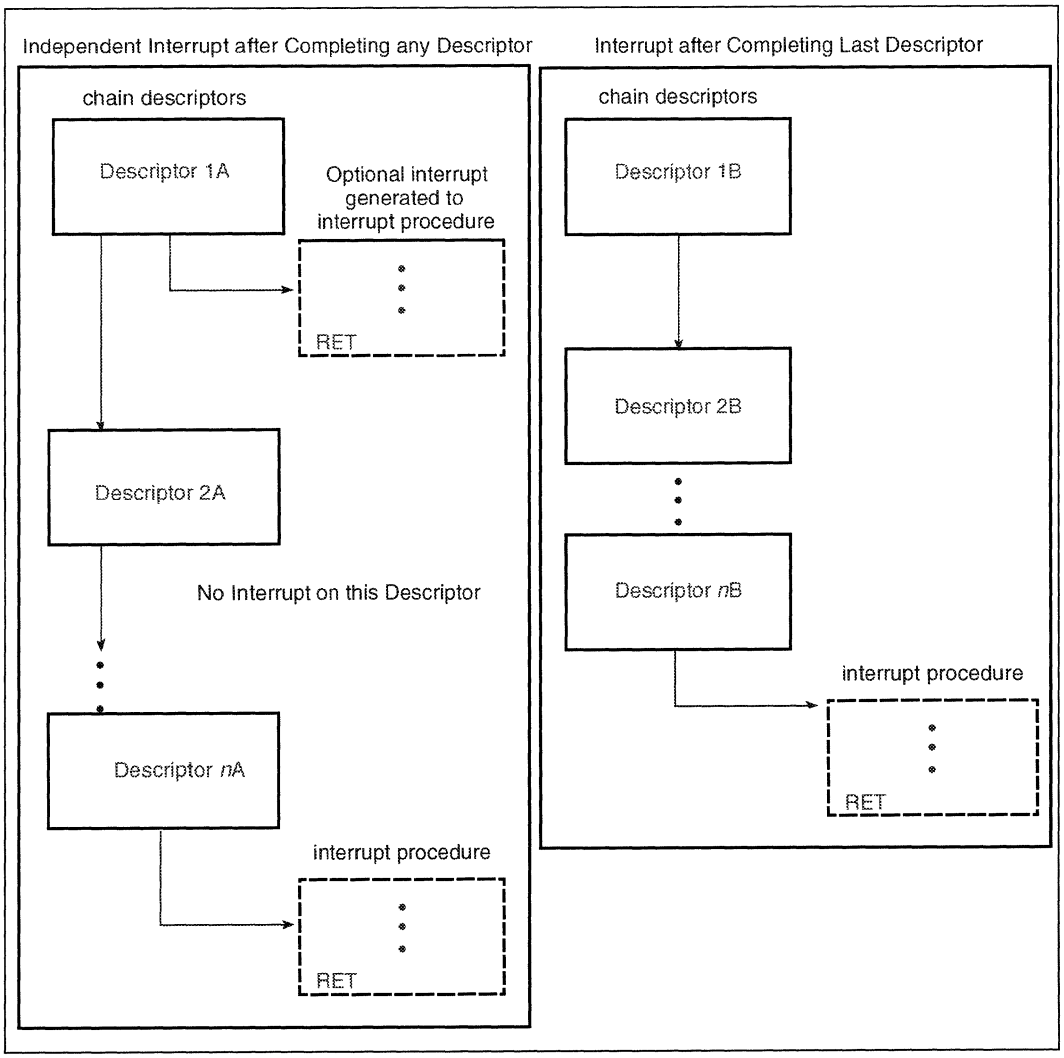
1. [Interrupt & Continue] The AAU completes processing a chain descriptor and the Accelerator Next Descriptor Address Register (ANDAR) is non-zero. If the *Interrupt Enable* bit within the Accelerator Descriptor Control Register (ADCR) is set, an interrupt will be generated to the i960® core processor. This interrupt is for synchronization purposes. The AAU will set the *End Of Transfer Interrupt* flag in the Accelerator Status Register (ASR). Since it is not the last chain descriptor in the list, the AAU will start to process the next chain descriptor without requiring any processor interaction.
2. [End of Chain] The AAU completes processing a chain descriptor and the Accelerator Next Descriptor Address Register is zero specifying the end of the chain. If the *Interrupt Enable* bit within the ADCR is set, an interrupt will be generated to the i960® core processor. The AAU will set the *End Of Chain Interrupt* flag in the ASR.
3. [Error] An error condition occurs (refer to Section 20.9, "Error Conditions" on page 20-24 for Application Accelerator error conditions) during a transfer. The AAU will halt operation on the current chain descriptor and not proceed to the next chain descriptor.

Each chain descriptor can independently set the *Interrupt Enable* bit in the Descriptor Control word. This bit enables an independent interrupt once a chain descriptor is processed. This bit can be set or clear within each chain descriptor. Control of interrupt generation within each descriptor aids in synchronization of the executing software with XOR operation.

Figure 20-10 shows two examples of program synchronization. The left column shows program synchronization based on individual chain descriptors. Descriptor 1A generated an interrupt to the processor, while descriptor 2A did not because the *Interrupt Enable* bit was clear. The last

descriptor *n*A, generated an interrupt to signify the end of the chain has been reached. The right column in Figure 20-10 shows an example where the interrupt was generated only on the last descriptor signifying the end of chain.

Figure 20-10. Synchronizing to Chained XOR Operation



20.3.8 Appending to The End of a Chain

Once the AAU has started processing a chain of descriptors, application software may need to append a chain descriptor to the current chain without interrupting the transfer in progress. The mechanism used for performing this action is controlled by the *Chain Resume* bit in the Accelerator Control Register (ACR).

The AAU reads the subsequent chain descriptor each time it completes the current chain descriptor and the Accelerator Next Descriptor Address Register (ANDAR) is non-zero. ANDAR always contains the address of the next chain descriptor to be read and the Accelerator Descriptor Address Register (ADAR) always contains the address of the current chain descriptor.

The procedure for appending chains requires the software to find the last chain descriptor in the current chain and change the Next Descriptor Address in that descriptor to the address of the new chain to be appended. The software then sets the *Chain Resume* bit in the ACR. It does not matter if the unit is active or not.

The AAU examines the *Chain Resume* bit of the ACR when the unit is idle or upon completion of a chain of transfers. If this bit is set, the AAU will re-read the Next Descriptor Address of the current chain descriptor and load it into ANDAR. The address of the current chain descriptor is contained in ADAR. The AAU clears the *Chain Resume* bit and then examines ANDAR. If ANDAR is not zero, the AAU will read the chain descriptor using this new address and begin a new operation. If ANDAR is zero, the AAU will remain or return to idle.

There are three cases to consider:

1. The AAU completes an *XOR-transfer* and it is not the last descriptor in the chain. In this case, the AAU clears the *Chain Resume* bit and reads the next chain descriptor. The appended descriptor will be read when the AAU reaches the end of the original chain.
2. The channel completes an *XOR-transfer* and it is the last descriptor in the chain. In this case, the AAU examines the state of the *Chain Resume* bit. If the bit is set, the AAU re-reads the current descriptor to get the address of the appended chain descriptor. If the bit is clear, the AAU returns to idle.
3. The AAU is idle. In this case, the AAU examines the state of the *Chain Resume* bit when the ACR is written. If the bit is set, the AAU re-reads the last descriptor from the most-recent chain to get the appended chain descriptor.

20.4 Packing and Unpacking

The Application Accelerator contains a hardware data packing and unpacking unit to support data transfers between unaligned source and destination addresses. Source and destination addresses can either be unaligned or aligned on natural boundaries. The packing unit optimizes data transfers to and from 32 and 64-bit memory. It reformats data words for the correct bus data width. When the read data needs to be packed or unpacked, the data is held internally and does not need to be re-read.

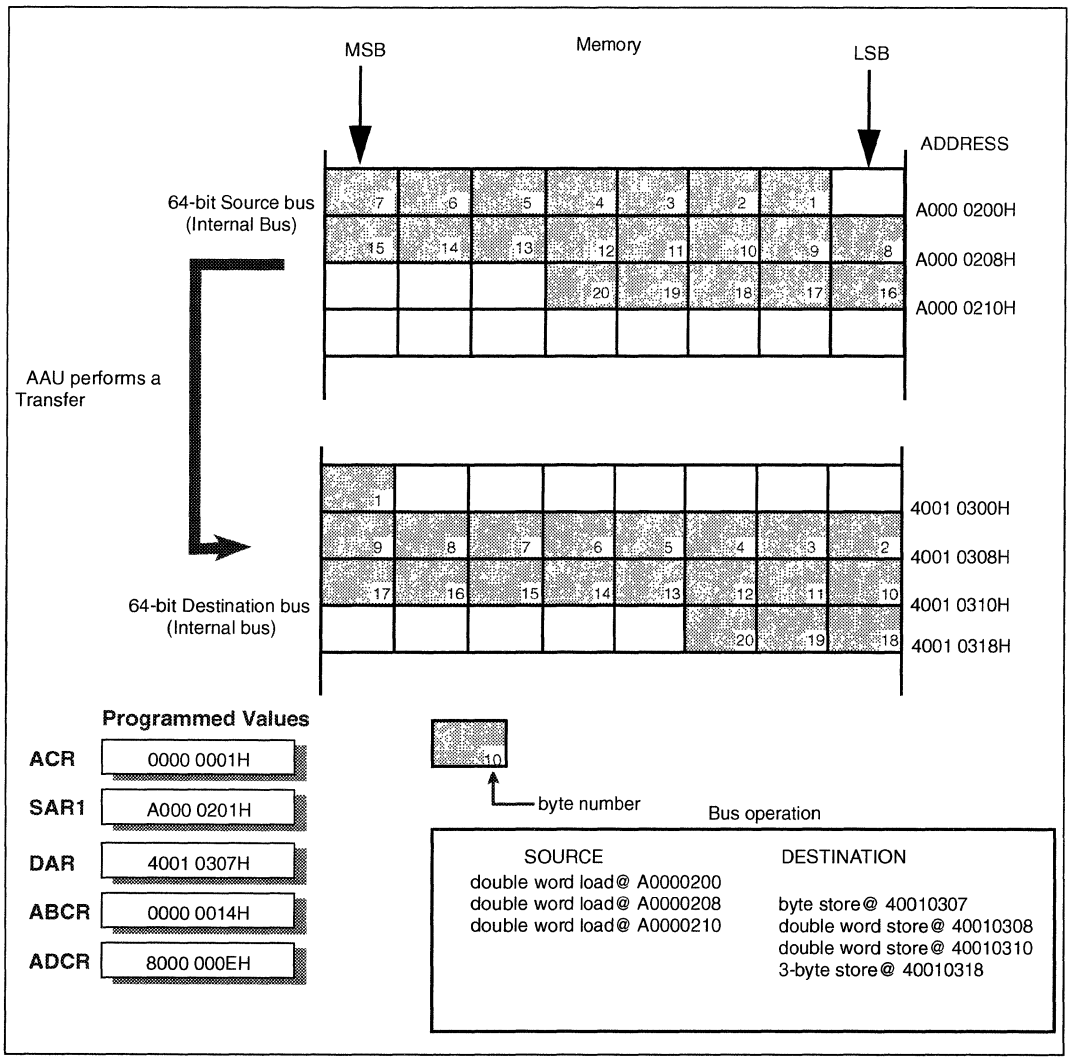
Aligned data transfers are those that fall on natural boundaries. For example; double words are aligned on 8-byte boundaries and words are aligned on 4-byte boundaries. Data transfers can take place in two instances:

- The source and destination addresses are both aligned.
- All or some source addresses are unaligned and the destination address is aligned or unaligned.

20.4.1 64-bit Unaligned Data Transfers

Figure 20-11 illustrates a data transfer between unaligned 64-bit, source and destination addresses.

Figure 20-11. Optimization of an Unaligned Data Transfer



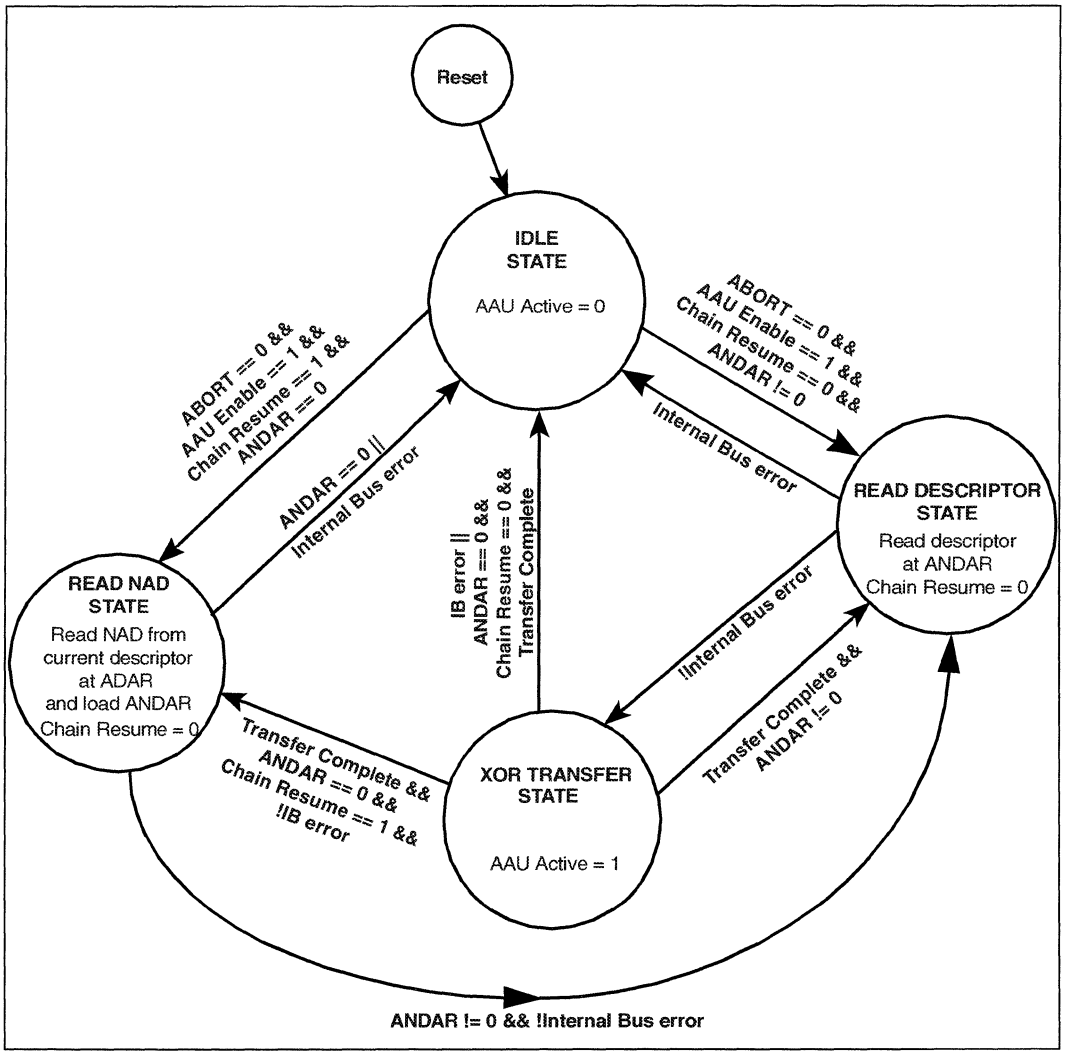
20.5 Application Accelerator Priority

The internal bus arbitration logic determines which internal bus master has access to the 80303 I/O processor internal bus. The Application Accelerator has an independent Bus Request/Grant signal pair to the internal bus arbitration logic. Chapter 17, "Intel® 80303 I/O Processor Arbitration" describes in detail the priority scheme between all of the bus masters on the internal bus.

20.6 Programming Model State Diagram

The AAU programming model diagram is shown in Figure 20-12. Error condition states are not shown.

Figure 20-12. Application Accelerator Programming Model State Diagram



20.7 Programming the Application Accelerator

The software for initiating an *XOR-transfer* using the Application Accelerator falls into the following categories:

- AAU initialization
- Start XOR transfer
- Suspend AAU

An example for each category is shown in the following sections as pseudo code flow.

20.7.1 Application Accelerator Initialization

The AAU is designed to have independent control of the interrupts, enables, and control. The initialization consists of virtually no overhead as shown in Figure 20-13.

Figure 20-13. Pseudo Code: Application Accelerator Initialization

```
ACR = 0x0000 0000 ; Disable the application accelerator
Call setup_accelerator
```

20.7.2 Start XOR Transfer

The AAU control register provides independent control each time the AAU is configured. This provides the greatest flexibility to the applications programmer. The example shown in Figure 20-14 describes the pseudo code for initiating an XOR operation with the AAU.

Figure 20-14. Pseudo Code: XOR Transfer Operation

```
; Set up descriptor in Intel® 80960 local memory at address d
d.nda  = 0          /* No chaining */
d.SAR1 = 0xA000 0400/* Source address of Data Block 1     */
d.SAR2 = 0xA000 0800/* Source address of Data Block 2     */
d.SAR3 = 0xA000 0C00/* Source address of Data Block 3     */
d.SAR4 = 0xA000 1000/* Source address of Data Block 4     */
d.DAR  = 0xB000 0100/* Destination address of XOR-ed data */
d.ABCR = 1024       /* Byte Count of 1024 */
d.ADCR = 0x8000 049F/* Direct fill data from Block 1      */
                    /* XOR with data from Block 2,Block 3 and
                       Block 4                            */
                    /* Store the result & interrupt processor */

; Check for Inactive AAU & no pending interrupts
if (ASR != 0) exit /* If AAU is not ready, exit */

; Start Operation
ANDAR = &d          ; Set up descriptor address
ACR = 0x00000001    ; Set AAU Enable bit
```

20.7.3 Suspend Application Accelerator

The Application Accelerator unit provides the ability to suspend the current state without losing status information. The AAU will resume without requiring application software to save the current configuration. The example shown in Figure 20-15 describes pseudo-code for suspending the ongoing operation and then restarting.

Figure 20-15. Pseudo Code: Suspend Application Accelerator

```
ACR = 0x0000 0000  ; Suspend ongoing AAU transfer

;Suspend Application Accelerator
ACR = 0x0000 0001  ; Resume AAU transfer
```

20.8 Interrupts

The Application Accelerator can generate an interrupt to the i960 core processor. The *Interrupt Enable* bit in the Accelerator Descriptor Control Register (ADCR.ie) determines whether the AAU generates an interrupt upon successful, error-free completion. Error conditions described in Section 20.9 also generate an interrupt. The AAU has one interrupt output connected to the PCI and Peripheral Interrupt Controller described in Chapter 8, "PCI and Peripheral Interrupt Controller Unit".

Once the AAU is enabled, the AAU loads the chain descriptor fields into the respective registers. A special case exists if ADCR.dwe = 0, then an interrupt will be generated (if enabled) after the descriptor is fetched and processed as defined by the block control fields in the ADCR. Table 20-2 summarizes the status flags and conditions when interrupts are generated in the Accelerator Status Register (ASR).

Table 20-2. AAU Interrupts

Interrupt Condition	Accelerator Status Register (ASR) Flags				Interrupt Generated?	
	Active	End of Transfer	End of Chain	IB Master Abort	ADCR.ie Set	ADCR.ie Clear
(ADCR.dwe == 0 || byte count == 0) && ANDAR != NULL (End of Transfer)	1	1	0	0	Y	N
(ADCR.dwe == 0 || byte count == 0) && ANDAR == NULL (End of Chain)	0	0	1	0	Y	N
IB Master Abort	0	0	0	1	Y	Y
IB Target Abort	0	0	0	0	N	N

Note: End-of-Transfer and End-of-Chain flags will be set only when ADCR.ie =1. If ADCR.ie = 0, then the above flags are always set to 0. End-of-Transfer Interrupt and End of Chain Interrupt can only be reported in the ASR if the descriptor fetch and processing completed without any reportable errors. However, multiple error conditions may occur and be reported together. Also, because the AAU does not stop after reporting the End-of-Transfer interrupt, an IB master-abort error may occur before the End-of-Transfer interrupt is serviced and cleared.

20.9 Error Conditions

Master Aborts that occur during a transfer are recorded by the Application Accelerator.

When an error occurs, the actions taken are detailed below:

- The AAU shall cease the ongoing transfer for the current chain descriptor and clear the *Application Accelerator Active* flag in the ASR.
- The AAU will not read any new chain descriptors.
- The AAU will set the error flag in the Accelerator Status Register. For example; if an IB master-abort occurred during a transfer, the channel will set bit 5 in the ASR.
- The AAU signals an interrupt to the i960® core processor.
- The Application Accelerator will not restart the transfer after an error condition. It is the responsibility of the application software to reconfigure the AAU to complete any remaining transfers.

Note: Target-aborts that occur while the AAU is the master on the internal bus are recorded by the MCU and interrupt the core. For correct operation of the AAU, user software has to disable the AAU before clearing the error condition. Further, the AAU needs to be re-enabled by writing a 1 to ACR.ae before initiating a new operation.

There are three possible scenarios for multi-bit ECC errors on reads or writes. These errors conditions are handled as detailed below:

- **Multi-bit ECC error on MCU Data Read**: Refer to Chapter 13, "Memory Controller" for details on error handling in this instance.
- **Multi-bit ECC error on MCU Data Write**: This instance covers the case where the first data write is less than a 64-bit value forcing the MCU to execute a *read-modify-write* operation. Refer to Chapter 13, "Memory Controller" for complete details.
- **Multi-bit ECC error on MCU Data Write**: This instance covers the case where the last data write is less than a 64-bit value forcing the MCU to execute a *read-modify-write* operation. Refer to Chapter 13, "Memory Controller" for complete details.

20.10 Power-up/Default Status

Upon power-up, an external hardware reset, the Application Accelerator Registers will be initialized to their default values.

20.11 Register Definitions

The Application Accelerator Unit contains fifteen memory-mapped registers for controlling its operation. There is read/write access only to the Accelerator Control Register, Accelerator Status Register, and the Accelerator Next Descriptor Address Register. All other registers are read-only and are loaded with new values from the chain descriptor whenever the AAU reads a chain descriptor from memory.

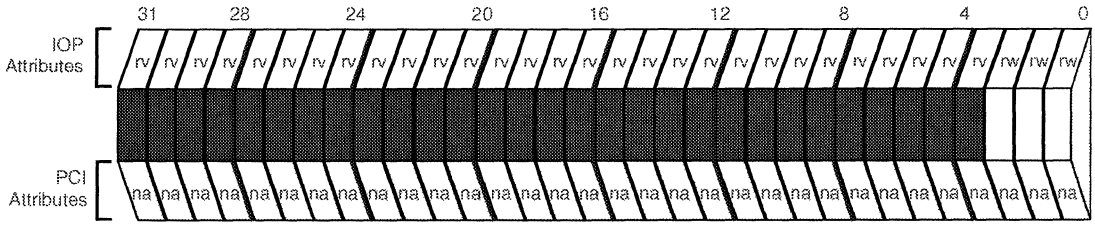
Table 20-3. Application Accelerator Unit Registers

Section, Register Name - Acronym (page)
Section 20.11.1, "Accelerator Control Register - ACR" on page 20-26
Section 20.11.2, "Accelerator Status Register - ASR" on page 20-27
Section 20.11.3, "Accelerator Descriptor Address Register - ADAR" on page 20-28
Section 20.11.4, "Accelerator Next Descriptor Address Register - ANDAR" on page 20-29
Section 20.11.5, "80960 Source Address Register - SAR" on page 20-30
Section 20.11.6, "80960 Destination Address Register - DAR" on page 20-31
Section 20.11.7, "Accelerator Byte Count Register - ABCR" on page 20-32
Section 20.11.8, "Accelerator Descriptor Control Register - ADCR" on page 20-33

20.11.1 Accelerator Control Register - ACR

The Accelerator Control Register (ACR) specifies parameters that dictate the overall operating environment. The ACR should be initialized prior to all other AAU registers following a system reset. Table 20-4 shows the register format. This register can be read or written while the AAU is active.

Table 20-4. Accelerator Control Register - ACR



Intel® 80960 Core Internal bus address
1800H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

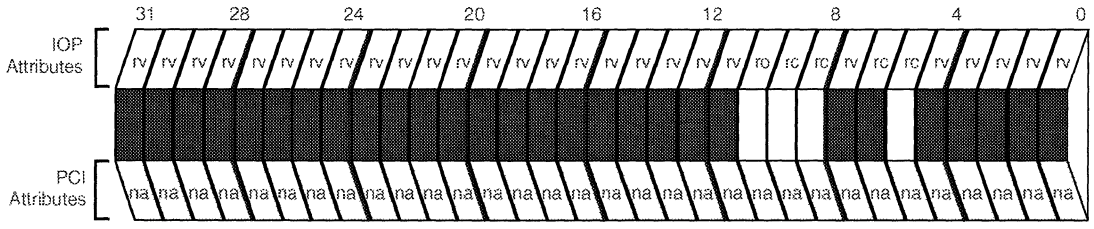
Bit	Default	Description
31:03	0	Reserved
02	0_2	512 Byte Buffer Enable - when set, causes the AAU to use only 512 bytes of the 1KB data buffer while processing **all descriptors.** **NOTES:** The 1KB data buffer consumes 128 data cycles on the Internal bus every time the AAU is granted the Internal Bus (assuming BC remains > 1K). Depending on the application, overall performance may be improved by throttling down the IB usage to 64 data cycles per Internal Bus grant.
01	0_2	Chain Resume - when set, causes the AAU to resume chaining by re-reading the current descriptor located at the address in the Accelerator Descriptor Address Register when the AAU is idle (AAU Active bit in the ASR is clear) or when the AAU completes a transfer. This bit is cleared by hardware when either: • The AAU completes a transfer and the Accelerator Next Descriptor Address Register is non-zero. In this case, the AAU proceeds to the next descriptor in the chain. • The AAU re-reads the chain descriptor located at the address in the Accelerator Descriptor Address Register and loads the Next Descriptor Address of that descriptor into the Accelerator Next Descriptor Address Register
00	0_2	AAU Enable - When set, the AAU enables transfers. When clear, the AAU disables any transfer. Clearing this bit when the AAU is active suspends the current transfer at the earliest opportunity by halting all internal bus transactions. The AAU does not initiate any new transfers when this bit is cleared. Data held in queues remains valid. Setting the bit after the AAU is suspended causes the AAU to resume the previously ongoing transfer.

20.11.2 Accelerator Status Register - ASR

The Accelerator Status Register (ASR) contains status flags that indicate status. This register is typically read by software to examine the source of an interrupt. See Section 20.9 for a description of the error conditions that are reported in the ASR. See Section 20.8 for a description of interrupts caused by the Application Accelerator.

If an AAU error occurs, application software should check the status of Accelerator Active flag before processing the interrupt.

Table 20-5. Accelerator Status Register - ASR



Intel® 80960 Core Internal bus address
1804H

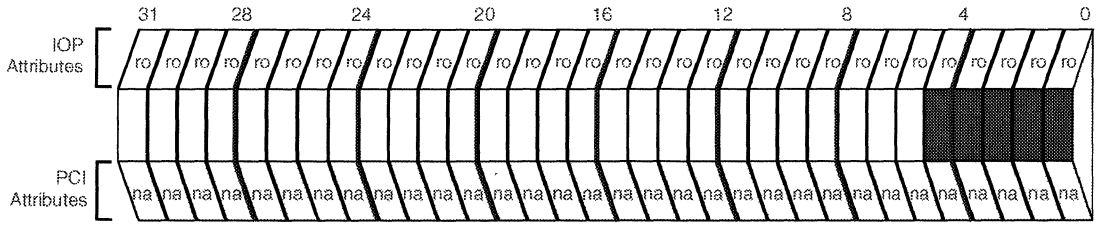
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:11	000000H	Reserved
10	0_2	Accelerator Active Flag - indicates the AAU is either active (in use) or idle (available). When set, indicates the AAU is in use and actively performing an operation. When clear, indicates the channel is idle and available to be configured for a new operation. The AAU clears the Accelerator Active flag when the previously configured transfer completes as a result of: • byte count reached zero and last chain descriptor is encountered (NULL value detected for Next Descriptor Address in chain descriptor) • Internal Bus Errors • Last chain descriptor is processed (NULL value detected for Next Descriptor Address in chain descriptor) and ADCR.dwe = 0. The Accelerator Active flag is set once a Chain Descriptor is read from memory.
09	0_2	End of Transfer Interrupt Flag - set when the AAU has signalled an interrupt to the 80960 processor after processing a descriptor but it is not the last descriptor in a chain.
08	0_2	End of Chain Interrupt Flag - set when the channel has signalled an interrupt to the 80960 processor after processing a descriptor that is the last in a chain.
07:06	0_2	Reserved
05	0_2	This bit is set if a Master-abort occurs during a transaction when the AAU is the master on the internal bus.
04:00	0_2	Reserved

20.11.3 Accelerator Descriptor Address Register - ADAR

The Accelerator Descriptor Address Register (ADAR) contains the address of the current chain descriptor in 80960 local memory for an *XOR-transfer*. This read-only register is loaded when a new chain descriptor is read. Table 20-6 depicts the Accelerator Descriptor Address Register. All chain descriptors are aligned on an eight, 32-bit word boundary.

Table 20-6. Accelerator Descriptor Address Register - ADAR



Intel® 80960 Core Internal bus address
1808H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:05	00000000000 00000000000 00000_2	Current Descriptor Address - local memory address of the current chain descriptor read by the Application Accelerator.
04:00	00000_2	Reserved

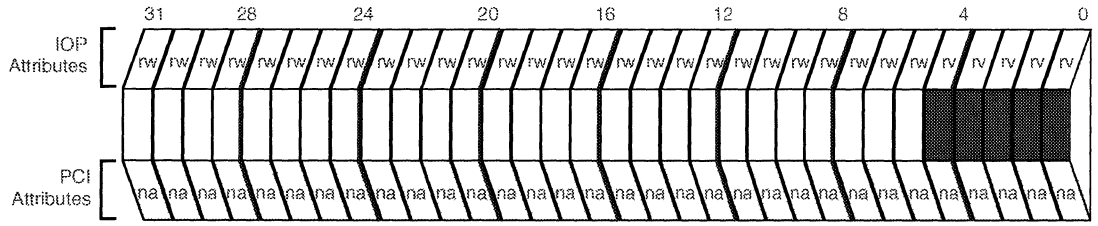
20.11.4 Accelerator Next Descriptor Address Register - ANDAR

The Accelerator Next Descriptor Address Register (ANDAR) contains the address of the next chain descriptor in 80960 local memory for an *XOR-transfer*. When starting a transfer, this register contains the address of the first chain descriptor. Table 20-7 depicts the Accelerator Next Descriptor Address Register.

All chain descriptors are aligned on an eight 32-bit word boundary. The AAU may set bits 04:00 to zero when loading this register.

Note: The *Accelerator Enable* bit in the ACR and the *Accelerator Active* bit in the ASR must both be clear prior to writing the ANDAR. Writing a value to this register while the AAU is active may result in undefined behavior.

Table 20-7. Accelerator Next Descriptor Address Register - ANDAR



Intel® 80960 Core Internal bus address
180CH

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:05	00000000000 00000000000 00000_2	Next Descriptor Address - local memory address of the next chain descriptor to be read by the Application Accelerator.
04:00	00000_2	Reserved

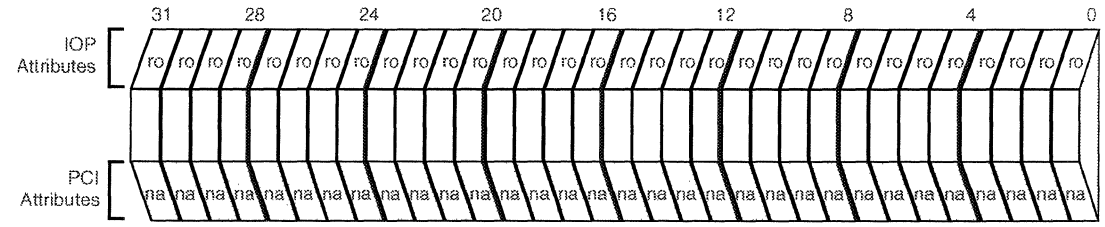
20.11.5 80960 Source Address Register - SAR

The 80960 Source Address Register (SARx) contains a 32-bit, 80960 local memory address. There are eight Source Address Registers (SAR1 - SAR8). Each of these registers is loaded with the address of the blocks of data to be operated upon by the Application Accelerator. The ADCR register (Table 20-11) controls the operation performed on each data block referenced by the registers (SAR1 - SAR8). The 80960 local memory address space is a 32-bit, byte addressable address space.

Reading the SARx registers once the AAU has started a chain descriptor will return the current source addresses. Once an XOR operation is initiated, these registers contain the current source addresses. For example; if the Byte Count is initially 4096 bytes and the AAU has completed the *XOR-transfer* operation on the first three 1K-byte data blocks, the value in register SAR1 will be the equal to the programmed descriptor value + 3072 (SAR1 + 3072).

Table 20-8 shows the 80960 Source Address Register. These read-only registers are loaded when a chain descriptor is read from memory.

Table 20-8. 80960 Source Address Register - SARx



Intel® 80960 Core Internal bus address

SAR1	1810H
SAR2	1814H
SAR3	1818H
SAR4	181CH
SAR5	182CH
SAR6	1830H
SAR7	1834H
SAR8	1838H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:00	00000000H	80960 Local Address - The 80960 local source address.

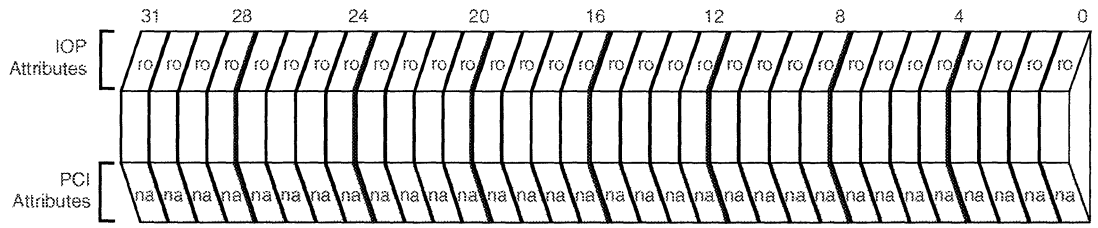
20.11.6 80960 Destination Address Register - DAR

The 80960 Destination Address Register (DAR) contains a 32-bit, 80960 local memory address. This address is the destination of the *XOR-transfer* - 80960 local memory address where XOR-ed data will be stored. The 80960 local memory address space is a 32-bit, byte addressable address space.

Reading the DAR once the AAU has started a chain descriptor will return the current destination address. For example; if the Byte Count is initially 4096 bytes and the AAU has completed the *XOR-transfer* operation on the first three 1K-byte data blocks, the value in the Destination Address Register (DAR) will be the equal to the programmed descriptor value + 3072 (DAR + 3072).

Table 20-9 shows the 80960 Destination Address Register. This read-only register is loaded when a chain descriptor is read from memory

Table 20-9. 80960 Destination Address Register - DAR



Intel® 80960 Core Internal bus address
1820H

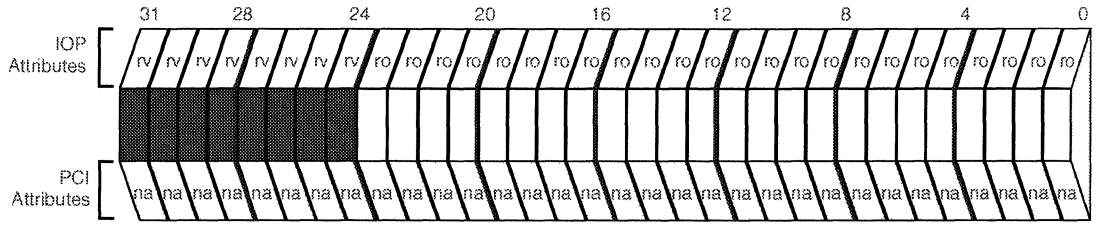
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:00	00000000H	80960 Local Address - The 80960 local destination address.

20.11.7 Accelerator Byte Count Register - ABCR

The Accelerator Byte Count Register (ABCR) contains the number of bytes to transfer for an *XOR-transfer* operation. This is a read-only register that is loaded from the Byte Count word in a chain descriptor. It allows for a maximum *XOR-transfer* of 16 Mbytes. A value of zero is a valid byte count and results in no read or write cycles being generated to the Memory Controller Unit. No cycles are generated on the 80303 I/O processor internal bus.

Table 20-10. Accelerator Byte Count Register - ABCR



Intel® 80960 Core Internal bus address
1824H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

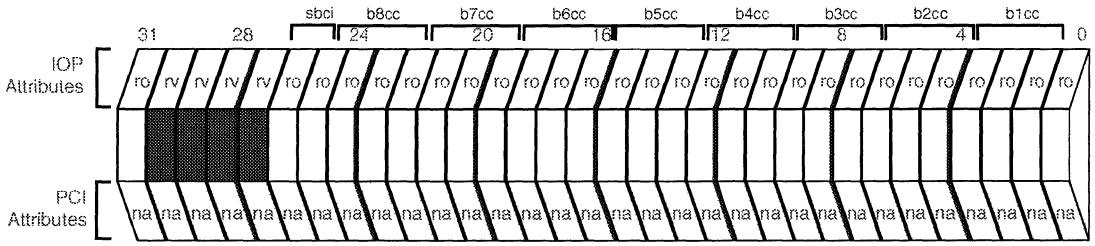
Bit	Default	Description
31:24	00H	Reserved
23:00	000000H	Byte Count - is the number of bytes to transfer for an *XOR-transfer* operation.

Note: Anytime this register is read by the i960® core processor, it contains the number of bytes left to *XOR-transfer* on the 80303 I/O processor internal bus. Note that valid data may be present in the Application Accelerator store queue. This register is decremented by 1 through 8 for every successful transfer from the store queue to the destination location. Table 20-10 shows the Accelerator Byte Count Register. The byte count value is not required to be aligned to a 32-bit word boundary (i.e., the byte count value can be a double word aligned, word aligned, short aligned, or byte aligned).

20.11.8 Accelerator Descriptor Control Register - ADCR

The Accelerator Descriptor Control Register contains control values for data transfer on a per-chain descriptor basis. This read-only register is loaded when a chain descriptor is read from memory. These values may vary from chain descriptor to chain descriptor. The AAU determines whether a mini-descriptor is appended to the end of the current chain descriptor by examining bits 26:25. Table 20-11 shows the definition of the Accelerator Descriptor Control Register.

Table 20-11. Accelerator Descriptor Control Register - ADCR (Sheet 1 of 3)

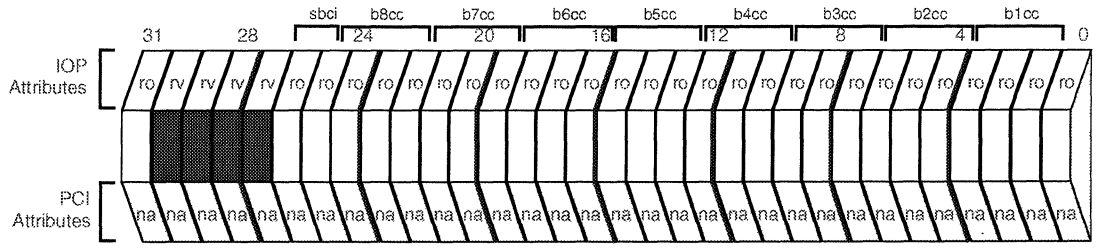


Intel® 80960 Core Internal bus address
1828H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31	0_2	Destination Write Enable - Determines whether data present in the store queue will be written out to 80960 local memory. When set, data in the queue will be written to the address specified in the Destination Address Register (DAR) after performing the specified operation on data referenced by the four SARx registers. When clear, data will be held in the queue. **NOTE:** If the ABCR register contains a value greater than the buffer size and this bit is cleared, the AAU will only read the first buffer of bytes and perform the specified function. It will not read the remaining bytes specified in the ABCR. Further, the AAU will proceed to process the next chain descriptor if it is specified.
30:27	0H	Reserved
26:25	00	Supplemental Block Control Interpreter - This bit field specifies the number of data blocks on which the *XOR-transfer* operation is executed. 00 0 Blocks - This specifies that no additional data blocks exist. The AAU will not read the mini-descriptor to initialize registers SAR5 - SAR8. 01 4 Blocks - This specifies that there are up to 4 additional data blocks. The AAU will therefore read the mini-descriptor to initialize registers SAR5 - SAR8. 10 Reserved 11 Reserved
24:22	0	**Block 8 Command Control** - This bit field specifies the type of operation to be carried out on the data pointed at by SAR8 register. 000 Null command - This implies that Block 8 Data can be disregarded for the current chain descriptor. The Application Accelerator will not transfer data from this block while processing the current chain descriptor. 001 XOR command - This implies that Block 8 Data will be transferred to the Application Accelerator to execute the XOR function. 010 Reserved 011 Reserved 100 Reserved 101 Reserved 110 Reserved 111 Reserved

Table 20-11. Accelerator Descriptor Control Register - ADCR (Sheet 2 of 3)

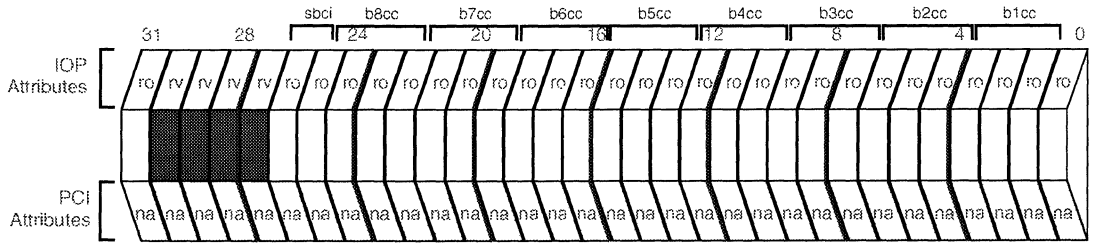


Intel® 80960 Core Internal bus address
1828H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
21:19	0	**Block 7 Command Control** - This bit field specifies the type of operation to be carried out on the data pointed at by SAR7 register. 000 Null command - This implies that Block 7 Data can be disregarded for the current chain descriptor. The Application Accelerator will not transfer data from this block while processing the current chain descriptor. 001 XOR command - This implies that Block 7 Data will be transferred to the Application Accelerator to execute the XOR function. 010 Reserved 011 Reserved 100 Reserved 101 Reserved 110 Reserved 111 Reserved
18:16	0	**Block 6 Command Control** - This bit field specifies the type of operation to be carried out on the data pointed at by SAR6 register. 000 Null command - This implies that Block 6 Data can be disregarded for the current chain descriptor. The Application Accelerator will not transfer data from this block while processing the current chain descriptor. 001 XOR command - This implies that Block 6 Data will be transferred to the Application Accelerator to execute the XOR function. 010 Reserved 011 Reserved 100 Reserved 101 Reserved 110 Reserved 111 Reserved
15:13	0	**Block 5 Command Control** - This bit field specifies the type of operation to be carried out on the data pointed at by SAR5 register. 000 Null command - This implies that Block 5 Data can be disregarded for the current chain descriptor. The Application Accelerator will not transfer data from this block while processing the current chain descriptor. 001 XOR command - This implies that Block 5 Data will be transferred to the Application Accelerator to execute the XOR function. 010 Reserved 011 Reserved 100 Reserved 101 Reserved 110 Reserved 111 Reserved
12:10	0	**Block 4 Command Control** - This bit field specifies the type of operation to be carried out on the data pointed at by SAR4 register. 000 Null command - This implies that Block 4 Data can be disregarded for the current chain descriptor. The Application Accelerator will not transfer data from this block while processing the current chain descriptor. 001 XOR command - This implies that Block 4 Data will be transferred to the Application Accelerator to execute the XOR function. 010 Reserved 011 Reserved 100 Reserved 101 Reserved 110 Reserved 111 Reserved

Table 20-11. Accelerator Descriptor Control Register - ADCR (Sheet 3 of 3)



Bit	Default	Description
09:07	0	**Block 3 Command Control** - This bit field specifies the type of operation to be carried out on the data pointed at by SAR3 register. 000 Null command - This implies that Block 3 Data can be disregarded for the current chain descriptor. The Application Accelerator will not transfer data from this block while processing the current chain descriptor. 001 XOR command - This implies that Block 3 Data will be transferred to the Application Accelerator to execute the XOR function. 010 Reserved 011 Reserved 100 Reserved 101 Reserved 110 Reserved 111 Reserved
06:04	0	**Block 2 Command Control** - This bit field specifies the type of operation to be carried out on the data pointed at by SAR2 register. 000 Null command - This implies that Block 2 Data can be disregarded for the current chain descriptor. The Application Accelerator will not transfer data from this block while processing the current chain descriptor. 001 XOR command - This implies that Block 2 Data will be transferred to the Application Accelerator to execute the XOR function. 010 Reserved 011 Reserved 100 Reserved 101 Reserved 110 Reserved 111 Reserved
03:01	0	**Block 1 Command Control** - This bit field specifies the type of operation to be carried out on the data pointed at by SAR1 register. 000 Null command - This implies that Block 1 Data can be disregarded for the current chain descriptor. The Application Accelerator will not transfer data from this block while processing the current chain descriptor. 001 XOR command - This implies that Block 1 Data will be transferred to the Application Accelerator to execute the XOR function. 010 Reserved 011 Reserved 100 Reserved 101 Reserved 110 Reserved 111 Direct Fill - This implies that Block 1 Data will be transferred directly from 80960 local memory to the store queue. In this instance, the data will bypass the Boolean Execution Unit within the Application Accelerator.
00	0	**Interrupt Enable** - When set, the Application Accelerator generates an interrupt to the 80303 I/O processor upon completion of a transfer. When clear, no interrupt is generated.

Performance Monitoring Unit 21

This chapter describes the Performance Monitoring features integrated on the Intel® 80303 I/O processor. These features aid in measuring and monitoring various system parameters that contribute to the overall performance of the processor. Also described are the operation modes, setup mechanisms, registers and interrupts.

The monitoring facility is generically referred to as PMON — Performance Monitoring. The facility is model specific, not architectural; its intended use is to gather performance measurements that can be used to retune/refine code for better system level performance.

21.1 Overview

The PMON facility provided on the 80303 I/O processor comprises:

- One dedicated Global Time Stamp counter, and
- Fourteen (14) Programmable Event Counters.

The global time stamp counter is a dedicated, free running 32-bit counter.

The programmable event counters are 32-bits wide. Each counter can be programmed to observe an event from a defined set of events. An event consists of a set of parameters which define a start condition and a stop condition. The monitored events are selected by programming an event select register (ESR).

21.2 Theory of Operation

The PMON facility provided on the 80303 I/O processor comprises:

- One dedicated Global Time Stamp counter, and
- Fourteen (14) programmable event counters.

The global time stamp counter is a dedicated, free running 32-bit counter clocked at one quarter the internal bus frequency. It provides a time base for monitoring all events on the 80303 I/O processor. Event counters are used to monitor events across different interfaces of the processor.

21.2.1 Global Time Stamp

The Global Time Stamp Counter is a dedicated, 32-bit counter provided on-chip. It contains a divisor which provides the input clock to the global time stamp counter. Typically the counter is clocked at one quarter the internal bus frequency. The counter is cleared upon the deassertion of the Internal Bus Reset signal. The counter interrupts the processor core if the interrupt bit (bit 0) in the Global Timer Mode Register (GTMR) is set. With the bit set, the counter sets bit 0 in the Event Monitoring Interrupt Status Register (EMISR) when it overflows. When this bit is set, an interrupt is generated to the core processor on the XINT6# interrupt pin. An overflow condition is defined as a counter rolling over from FFFF FFFFH to 0000 0000H.

Once the counter reaches the maximum value, it rolls over to zero and increments at the clock frequency. The value in the counter is accessible at all times by reading the memory mapped, Global Time Stamp Register (GTSR). The GTSR is a read-only register.

21.2.2 Programmable Event Counters

There are fourteen (14) general-purpose, 32-bit wide Programmable Event Counters (PECx). Each counter is programmed to monitor an event from a predetermined list of events. Depending on the monitored interface, the event tracked in any counter varies. Each counter is accessible through a memory-mapped, read-only register.

The programmable event counters provide real-time monitoring capability. The current count value contained in any counter is obtained by accessing the corresponding memory-mapped register.

Any counter that overflows sets the corresponding bit in the Event Monitoring Interrupt Status Register (EMISR). Once a counter reaches the maximum value, it rolls over to zero and starts incrementing. For example, when PEC1 overflows, it will set bit1 in the EMISR. Similarly, when any other counter (PEC2 - PEC14) overflows, the corresponding bit in the EMISR (bit2:14) is set. Once a bit in the EMISR is set, an interrupt is generated to the core processor on the XINT6# interrupt pin.

All event counters and the Global Time Stamp Counter are disabled after RESET and the values contained are undefined. All counters including the Global Time Stamp Counter are initialized to zero when a specific monitoring mode is chosen by writing a value to the Event Select Register (ESR) during performance monitoring. The fourteen programmable event counters monitor both kinds of events: occurrence events and duration events.

21.2.2.1 Occurrence Events

An occurrence event is counted each time the event occurs. Table 21-1 lists the various occurrence events that are monitored on the 80303 I/O processor.

Table 21-1. Occurrence Events

Observed Interface	Monitored Event
Secondary PCI bus	Number of grants to the Bridge
	Number of grants to Secondary Address Translation Unit
	Number of grants to DMA Ch-2
	Number of grants to the Intel® 80303 I/O processor
	Number of grants to external PCI masters 0..5
Primary PCI bus	Number of grants to the Bridge
	Number of grants to Primary Address Translation Unit (PATU)
	Number of grants to DMA Ch-0 and Ch-1
80303 I/O processor internal bus	Number of grants to Intel® i960® core processor.
	Number of grants to DMA Ch-0, Ch-1 and Ch-2
	Number of grants to Application Accelerator
	Number of times backoff (BOFF) asserted by Primary Address Translation Unit (PATU)
	Number of times backoff (BOFF) asserted by Secondary Address Translation Unit (SATU)
	Number of grants to PATU and SATU

21.2.2.2 Duration Events

For a duration event, the counter counts the number of clocks during which a particular condition or set of conditions is true. Acquisition latency measurements comprise:

- **Arbitration Latency**: This represents the elapsed time between the bus master's request to use the bus until the requesting master is granted the bus.
- **Bus Acquisition Latency**: This represents the elapsed time between the requesting bus master being granted the bus and the current bus master surrendering the bus allowing the requesting bus master to initiate the next transaction.

Table 21-2 lists the various duration events that are monitored on the 80303 I/O processor.

Table 21-2. Duration Events

Observed Interface	Monitored Event
Primary and Secondary PCI buses	Number clocks the PCI bus is busy
	Number of clocks the PCI bus is idle
	Acquisition latency and ownership metrics for the PATU and SATU
	Acquisition latency and ownership metrics for DMA Ch-0, Ch-1 and Ch-2
	Acquisition latency and ownership metrics for the Bridge
	Acquisition latency and ownership metrics for external masters 0..5 and the Intel® 80303 I/O processor (summation of all internal masters on secondary interface) on the secondary PCI bus
	Acquisition latency and ownership metrics for the 80303 I/O processor on the primary PCI bus
80303 I/O processor internal bus	Acquisition latency and ownership metrics for the Intel® i960® core processor
	Acquisition latency and ownership metrics for DMA Channels 0,1 and 2
	Acquisition latency and ownership metrics for Application Accelerator
	Acquisition latency and ownership metrics for the PATU
	Acquisition latency and ownership metrics for the SATU

21.2.3 Performance Monitoring

The Event Select Register (ESR) determines the interface to be monitored. Table 21-3 shows the relationship between the monitored mode specified in the ESR and the monitored interface. Performance Monitoring consists of a collection of event primitives which can then be used by the user for statistical calculations.

Table 21-3. Relationship between the Monitored mode and Monitored Interface

Monitoring Mode	Monitored Interface
M0	Performance Monitoring Disabled
M1	Primary PCI bus and internal agents (bridge, dma Ch0, dma Ch1, patu)
M2	Secondary PCI bus and internal agents (bridge, dma Ch2, satu)
M3	Secondary PCI bus and PCI agents (external masters 0..2 and Intel® 80303 I/O processor)
M4	Secondary PCI bus and PCI agents (external masters 3..5 and 80303 I/O processor)
M5	80303 I/O processor internal bus, DMA Channels and Application Accelerator
M6	80303 I/O processor internal bus, PATU, SATU and Intel® i960® processor
M7	80303 I/O processor internal bus, Primary PCI bus, Secondary PCI bus and Secondary PCI agents (external masters 0..5 & 80303 I/O processor)

Events across various interfaces are monitored by programming the event select register (ESR). The various interfaces that can be monitored on the 80303 I/O processor are:

- **Primary PCI bus and internal agents**: The different events monitored in this mode provide information about the primary PCI bus and the internal agents. The internal agents monitored are: Bridge, PATU, DMA Ch-0 and DMA Ch-1.
- **Secondary PCI bus and internal agents**: The different events monitored in this mode provide information about the secondary PCI bus and the internal agents. The internal agents monitored are: Bridge, SATU and DMA Ch-2.
- **Secondary PCI bus interface and external agents**: The different events monitored in this mode provide information about the secondary PCI bus and agents. There are seven PCI agents monitored: six external agents and the 80303 I/O processor.
- 80303 I/O processor **internal bus and bus masters**: The different events monitored in this mode provide information about the internal bus and the internal bus masters. The internal bus masters are: i960 core processor, Three DMA channels, Two Address Translation Units and the Application Accelerator.
- 80303 I/O processor **internal bus and Secondary PCI bus**: The different events monitored in this mode provide information about the internal bus and the PCI bus.

21.3 Event Description

Events monitored on the 80303 I/O processor can either be duration events or occurrence events. There are 98 events monitored on the 80303 I/O processor. A maximum of fourteen (14) events can be monitored concurrently. There are ten monitoring modes implemented as described below:

- **Mode 0:** Performance Monitoring disabled on the 80303 I/O processor.
- **Mode 1:** Monitors events on the Primary PCI bus.
- **Mode 2:** Monitors events on the Secondary PCI bus.
- **Mode 3:** Monitors events on the Secondary PCI bus.
- **Mode 4:** Monitors events on the Secondary PCI bus.
- **Mode 5:** Monitors events on the 80303 I/O processor internal bus.
- **Mode 6:** Monitors events on the 80303 I/O processor internal bus.
- **Mode 7:** Monitors events on the Primary PCI bus, Secondary PCI bus and 80303 I/O processor internal bus.

21.3.1 Mode0: Performance Monitoring Disabled

Programming Mode0 (M0) in the ESR disables performance monitoring on the 80303 I/O processor. Reading any counter including the GTSR in Mode 0 returns undefined results.

21.3.2 Mode1: Primary PCI bus and Internal Agents

Programming Mode1 (M1) in the ESR enables performance monitoring on the primary PCI bus. All counters are clocked at the primary PCI bus frequency. There are four internal agents monitored: PCI bridge, DMA Ch-0, DMA Ch-1 and PATU. The following sections describe the monitored events in Mode 1.

21.3.2.1 M1_PPCIBus_idle

This duration event increments the counter every PCI idle cycle. An idle cycle occurs when there is no activity on the bus due to data being transferred and/or the bus is not in an overhead cycle. An overhead cycle is a cycle when a master owns the bus, however the master is unable to send data or the target is unable to receive data - hence no data is transferred.

21.3.2.2 M1_PPCIBus_busy

This duration event increments the counter every PCI data cycle. Data cycles comprise of two instances:

- The 80303 I/O processor as a master on the bus is involved in data transfers to other masters.
- External masters initiate data transfers to either the 80303 I/O processor or to other masters on the bus.

21.3.2.3 M1_bridge_acq

This duration event counts the number of clocks spent by the bridge acquiring the PCI interface. The counter increments on every clock cycle after the bridge requests the PCI bus but has not actively driven the PCI bus as a master. The counter will also increment for all clock cycles when this agent's *Request Signal* is asserted but bus ownership currently belongs to another master. This is an event primitive, used in conjunction with another event primitive (number of grants granted to bridge) to calculate the average acquisition latency for the bridge.

21.3.2.4 M1_bridge_own

This duration event counts the duration for which the bridge is the master on the PCI interface. The counter increments on every clock cycle during which the bridge is the bus master.

21.3.2.5 M1_DMA0_acq

This duration event counts the number of clocks spent by the DMA Ch-0 acquiring the PCI interface. The counter increments on every clock cycle after the channel requests the PCI bus but has not actively driven the PCI bus as a master. The counter will also increment for all clock cycles when this agent's *Request Signal* is asserted but bus ownership currently belongs to another master. This is an event primitive, used in conjunction with another event primitive (number of grants granted to DMA Ch-0) to calculate the average acquisition latency for the channel.

21.3.2.6 M1_DMA0_own

This duration event counts the duration for which DMA Ch-0 is the master on the PCI interface. The counter increments on every clock cycle during which the channel is the bus master.

21.3.2.7 M1_DMA1_acq

This duration event counts the number of clocks spent by the DMA Ch-1 acquiring the PCI interface. The counter increments on every clock cycle after the channel requests the PCI bus but has not actively driven the PCI bus as a master. The counter will also increment for all clock cycles when this agent's *Request Signal* is asserted but bus ownership currently belongs to another master. This is an event primitive, used in conjunction with another event primitive (number of grants granted to DMA Ch-1) to calculate the average acquisition latency for the channel.

21.3.2.8 M1_DMA1_own

This duration event counts the duration for which DMA Ch-1 is the master on the PCI interface. The counter increments on every clock cycle during which the channel is the bus master.

21.3.2.9 M1_PATU_acq

This duration event counts the number of clocks spent by the PATU acquiring the PCI interface. The counter increments on every clock cycle after the unit requests the PCI bus but has not actively driven the PCI bus as a master. The counter will also increment for all clock cycles when this agent's *Request Signal* is asserted but bus ownership currently belongs to another master. This is an event primitive, used in conjunction with another event primitive (number of grants granted to PATU) to calculate the average acquisition latency for the unit.

21.3.2.10 M1_PATU_own

This duration event counts the duration for which PATU is the master on the PCI interface. The counter increments on every clock cycle during which the unit is the bus master.

21.3.2.11 M1_DMA0_gnt

This occurrence event monitors the number of times DMA Ch-0 is granted the PCI bus. This event increments the counter when the channel is the PCI bus master. The counter is incremented once for every new transaction. For multi-cycle transactions, the counter increments once on the first cycle.

21.3.2.12 M1_DMA1_gnt

This occurrence event monitors the number of times DMA Ch-1 is granted the PCI bus. This event increments the counter when the channel is the PCI bus master. The counter is incremented once for every new transaction. For multi-cycle transactions, the counter increments once on the first cycle.

21.3.2.13 M1_PATU_gnt

This occurrence event monitors the number of times the PATU is granted the PCI bus. This event increments the counter when the unit is the PCI bus master. The counter is incremented once for every new transaction. For multi-cycle transactions, the counter increments once on the first cycle.

21.3.2.14 M1_bridge_gnt

This occurrence event monitors the number of times the bridge is granted the PCI bus. This event increments the counter when the bridge is the PCI bus master. The counter is incremented once for every new transaction. For multi-cycle transactions, the counter increments once on the first cycle.

21.3.3 Mode 2: Secondary PCI Bus and Internal Agents

Programming Mode2 (M2) in the ESR enables performance monitoring on the secondary PCI bus. All counters are clocked at the secondary PCI bus frequency. There are three internal agents monitored: PCI Bridge, DMA Ch-2 and the Secondary Address Translation Unit (SATU). The following sections describe the monitored events in Mode 2.

21.3.3.1 M2_SPCIBus_idle

This duration event increments the counter every secondary PCI idle cycle. An idle cycle occurs when there is no activity on the bus due to data being transferred and/or the bus is not in an overhead cycle. An overhead cycle is a cycle when a master owns the bus, however the master is unable to send data or the target is unable to receive data - hence no data is transferred.

21.3.3.2 M2_SPCIBus_busy

This duration event increments the counter every secondary PCI data cycle. Data cycles comprise of two instances:

- The 80303 I/O processor as a master on the bus is involved in data transfers to other masters.
- External masters initiate data transfers to either the 80303 I/O processor or to other masters on the bus.

21.3.3.3 M2_SATU_acq

This duration event counts the number of clocks spent by the SATU acquiring the PCI interface. The counter increments on every clock cycle after the unit requests the PCI bus but has not actively driven the PCI bus as a master. The counter will also increment for all clock cycles when this agent's *Request Signal* is asserted but bus ownership currently belongs to another master. This is an event primitive, used in conjunction with another event primitive (number of grants granted to SATU) to calculate the average acquisition latency for the unit.

21.3.3.4 M2_SATU_own

This duration event counts the duration for which SATU is the master on the PCI interface. The counter increments on every clock cycle during which the unit is the bus master.

21.3.3.5 M2_bridge_acq

This duration event counts the number of clocks spent by the bridge acquiring the secondary PCI interface. The counter increments on every clock cycle after the bridge requests the PCI bus but has not actively driven the PCI bus as a master. The counter will also increment for all clock cycles when this agent's *Request Signal* is asserted but bus ownership currently belongs to another master. This is an event primitive, used in conjunction with another event primitive (number of grants granted to bridge) to calculate the average acquisition latency for the bridge.

21.3.3.6 M2_bridge_own

This duration event counts the duration for which the bridge is the master on the secondary PCI interface. The counter increments on every clock cycle during which the bridge is the bus master.

21.3.3.7 M2_DMA2_acq

This duration event counts number of clocks spent by DMA Ch-2 acquiring PCI interface. The counter increments every clock cycle after the channel requests the PCI bus, but has not actively driven the PCI bus as master. The counter also increments for all clock cycles when agent *Request Signal* is asserted, but bus ownership currently belongs to another master. This is an event primitive, used in conjunction with another event primitive (number of grants granted to DMA Ch-2) to calculate average acquisition latency for the channel.

21.3.3.8 M2_DMA2_own

This duration event counts the duration for which DMA Ch-2 is the master on the PCI interface. The counter increments on every clock cycle during which the channel is the bus master.

21.3.3.9 M2_bridge_gnt

This occurrence event monitors number of times the bridge is granted the secondary PCI bus. This event increments the counter when the bridge is the PCI bus master. The counter is incremented once for every new transaction. For multi-cycle transactions, the counter increments once on the first cycle.

21.3.3.10 M2_SATU_gnt

This occurrence event monitors the number of times the SATU is granted the PCI bus. This event increments the counter when the unit is the PCI bus master. The counter is incremented once for every new transaction. For multi-cycle transactions, the counter increments once on the first cycle.

21.3.3.11 M2_DMA2_gnt

This occurrence event monitors the number of times DMA Ch-2 is granted the PCI bus. This event increments the counter when the channel is the PCI bus master. The counter is incremented once for every new transaction. For multi-cycle transactions, the counter increments once on the first cycle.

21.3.3.12 M2_PPCIBus_idle

This duration event increments the counter every primary PCI idle cycle. An idle cycle occurs when there is no activity on the bus due to data being transferred and/or the bus is not in an overhead cycle. An overhead cycle is a cycle when a master owns the bus, however the master is unable to send data or the target is unable to receive data - hence no data is transferred.

21.3.3.13 M2_PPCIBus_busy

This duration event increments the counter every primary PCI data cycle. Data cycles comprise of two instances:

- The 80303 I/O processor as a master on the bus is involved in data transfers to other masters.
- External masters initiate data transfers to either the 80303 I/O processor or other masters on the bus.

21.3.3.14 M2_IBus_busy

This duration event increments the counter on every internal bus data cycle. This enables calculation of data utilization of the bus.

21.3.4 Mode 3: Secondary PCI Bus and External Agents

Programming Mode3 (M3) in the ESR enables performance monitoring on the secondary PCI bus. In addition, performance monitoring is done for external agents (80303 I/O processor, Master0, Master1, Master2) on the secondary PCI bus. Master0 indicates the external PCI device that is connected to the REQ0 and GNT0 signals of the internal arbiter in the 80303 I/O processor. The nomenclature is similar for all other external PCI masters; Master 1 through Master 5. There are four external agents monitored including the 80303 I/O processor.

All counters are clocked at the secondary PCI bus frequency. The following sections describe the monitored events in Mode 3.

21.3.4.1 M3_SPCIbus_idle

This duration event increments the counter every PCI idle cycle. An idle cycle occurs when there is no activity on the bus due to data being transferred and/or the bus is not in an overhead cycle. An overhead cycle is a cycle when a master owns the bus, however the master is unable to send data or the target is unable to receive data - hence no data is transferred.

21.3.4.2 M3_SPCIbus_busy

This duration event increments the counter every PCI data cycle. Data cycles comprise of two instances:

- The 80303 I/O processor as a master on the bus is involved in data transfers to other masters.
- External masters initiate data transfers to either the 80303 I/O processor or to other masters on the bus.

21.3.4.3 M3_SPCI_IOP_acq

This duration event counts number of clocks spent by the 80303 I/O processor (includes the bridge, dma Ch-2, and satu) acquiring the secondary PCI interface. The counter increments on every clock cycle after the processor has requested use of the secondary PCI bus but has not actively driven the secondary PCI bus as a master. The counter will also increment for all clock cycles when this agent's *Request Signal* is asserted but bus ownership currently belongs to another master. This is an event primitive, used in conjunction with another event primitive (number of grants granted to 80303 I/O processor) to calculate the average acquisition latency for the processor.

21.3.4.4 M3_SPCI_IOP_own

This duration event counts the duration for which the 80303 I/O processor is the master on the secondary PCI interface. The counter increments on every clock cycle during which the processor is the bus master.

21.3.4.5 M3_D0_acq

This duration event counts the number of clocks spent by PCI Master 0 acquiring the PCI interface. The counter increments on every clock cycle after the device has requested use of the PCI bus but has not actively driven the PCI bus as a master. The counter will also increment for all clock cycles when this agent's *Request Signal* is asserted but bus ownership currently belongs to another master. This is an event primitive, used in conjunction with another event primitive (number of grants granted to PCI Master 0) to calculate the average acquisition latency for the device.

21.3.4.6 M3_D0_own

This duration event counts the duration for which PCI Master 0 is the master on the PCI interface. The counter increments on every clock cycle during which PCI Master 0 is the bus master.

21.3.4.7 M3_D1_acq

This duration event counts the number of clocks spent by PCI Master 1 acquiring the PCI interface. The counter increments on every clock cycle after the device has requested use of the PCI bus but has not actively driven the PCI bus as a master. The counter will also increment for all clock cycles when this agent's *Request Signal* is asserted but bus ownership currently belongs to another master. This is an event primitive, used in conjunction with another event primitive (number of grants granted to PCI Master 1) to calculate the average acquisition latency for the device.

21.3.4.8 M3_D1_own

This duration event counts the duration for which PCI Master 1 is the master on the PCI interface. The counter increments on every clock cycle during which PCI Master 1 is the bus master.

21.3.4.9 M3_D2_acq

This duration event counts the number of clocks spent by PCI Master 2 acquiring the PCI interface. The counter increments on every clock cycle after the device has requested use of the PCI bus but has not actively driven the PCI bus as a master. The counter will also increment for all clock cycles when this agent's *Request Signal* is asserted but bus ownership currently belongs to another master. This is an event primitive, used in conjunction with another event primitive (number of grants granted to PCI Master 2) to calculate the average acquisition latency for the device.

21.3.4.10 M3_D2_own

This duration event counts the duration for which PCI Master 2 is the master on the PCI interface. The counter increments on every clock cycle during which PCI Master 2 is the bus master.

21.3.4.11 M3_SPCI_IOP_gnt

This occurrence event monitors the number of times the 80303 I/O processor is granted the secondary PCI bus. It increments the counter when the processor is the secondary PCI bus master. The counter is incremented once for every new transaction. For multi-cycle transactions, the counter increments once on the first cycle. The count value is a summation of the individual grants received by the bridge, satu and dma Ch-2.

21.3.4.12 M3_D0_gnt

This occurrence event monitors the number of times PCI Master 0 is granted the PCI bus. It increments the counter when the device is the PCI bus master. The counter is incremented once for every new transaction. For multi-cycle transactions, the counter increments once on the first cycle.

21.3.4.13 M3_D1_gnt

This occurrence event monitors the number of times PCI Master 1 is granted the PCI bus. It increments the counter when the device is the PCI bus master. The counter is incremented once for every new transaction. For multi-cycle transactions, the counter increments once on the first cycle.

21.3.4.14 M3_D2_gnt

This occurrence event monitors the number of times PCI Master 2 is granted the PCI bus. It increments the counter when the device is the PCI bus master. The counter is incremented once for every new transaction. For multi-cycle transactions, the counter increments once on the first cycle.

21.3.5 Mode 4: Secondary PCI Bus and External Agents

Programming Mode4 (M4) in the ESR enables performance monitoring on the secondary PCI bus. In addition, performance monitoring is done for external agents (80303 I/O processor, Master3, Master4 and Master5) on the PCI bus. Master3 indicates the external PCI device that is connected to the REQ3 and GNT3 signals of the internal arbiter in the 80303 I/O processor. The nomenclature is similar for all other external PCI masters; Master 1 through Master 5.

All counters are clocked at the secondary PCI bus frequency. The following sections describe the monitored events in Mode 4.

21.3.5.1 M4_SPCIbus_idle

This duration event increments the counter every PCI idle cycle. An idle cycle occurs when there is no activity on the bus due to data being transferred and/or the bus is not in an overhead cycle. An overhead cycle is a cycle when a master owns the bus, however the master is unable to send data or the target is unable to receive data - hence no data is transferred.

21.3.5.2 M4_SPCIbus_busy

This duration event increments the counter every PCI data cycle. Data cycles comprise of two instances:

- The 80303 I/O processor as a master on the bus is involved in data transfers to other masters.
- External masters initiate data transfers to either the 80303 I/O processor or to other masters on the bus.

21.3.5.3 M4_D3_acq

This duration event counts the number of clocks spent by PCI Master 3 acquiring the PCI interface. The counter increments on every clock cycle after the device has requested use of the PCI bus but has not actively driven the PCI bus as a master. The counter will also increment for all clock cycles when this agent's *Request Signal* is asserted but bus ownership currently belongs to another master. This is an event primitive, used in conjunction with another event primitive (number of grants granted to PCI Master 3) to calculate the average acquisition latency for the device.

21.3.5.4 M4_D3_own

This duration event counts the duration for which PCI Master 3 is the master on the PCI interface. The counter increments on every clock cycle during which PCI Master 3 is the bus master.

21.3.5.5 M4_D4_acq

This duration event counts the number of clocks spent by PCI Master 4 acquiring the PCI interface. The counter increments on every clock cycle after the device has requested use of the PCI bus but has not actively driven the PCI bus as a master. The counter will also increment for all clock cycles when this agent's *Request Signal* is asserted but bus ownership currently belongs to another master. This is an event primitive, used in conjunction with another event primitive (number of grants granted to PCI Master 4) to calculate the average acquisition latency for the device.

21.3.5.6 M4_D4_own

This duration event counts the duration for which PCI Master 4 is the master on the PCI interface. The counter increments on every clock cycle during which PCI Master 4 is the bus master.

21.3.5.7 M4_D5_acq

This duration event counts the number of clocks spent by PCI Master 5 acquiring the PCI interface. The counter increments on every clock cycle after the device has requested use of the PCI bus but has not actively driven the PCI bus as a master. The counter will also increment for all clock cycles when this agent's *Request Signal* is asserted but bus ownership currently belongs to another master. This is an event primitive, used in conjunction with another event primitive (number of grants granted to PCI Master 5) to calculate the average acquisition latency for the device.

21.3.5.8 M4_D5_own

This duration event counts the duration for which PCI Master 5 is the master on the PCI interface. The counter increments on every clock cycle during which PCI Master 5 is the bus master.

21.3.5.9 M4_D3_gnt

This occurrence event monitors the number of times PCI Master 3 is granted the PCI bus. It increments the counter when the device is the PCI bus master. The counter is incremented once for every new transaction. For multi-cycle transactions, the counter increments once on the first cycle.

21.3.5.10 M4_D4_gnt

This occurrence event monitors the number of times PCI Master 4 is granted the PCI bus. It increments the counter when the device is the PCI bus master. The counter is incremented once for every new transaction. For multi-cycle transactions, the counter increments once on the first cycle.

21.3.5.11 M4_D5_gnt

This occurrence event monitors the number of times PCI Master 5 is granted the PCI bus. It increments the counter when the device is the PCI bus master. The counter is incremented once for every new transaction. For multi-cycle transactions, the counter increments once on the first cycle.

21.3.5.12 M4_SPCI_IOP_gnt

This occurrence event monitors the number of times the 80303 I/O processor is granted the secondary PCI bus. It increments the counter when the processor is the secondary PCI bus master. The counter is incremented once for every new transaction. For multi-cycle transactions, the counter increments once on the first cycle. The count value is a summation of the individual grants received by the bridge, satu and dma Ch-2.

21.3.5.13 M4_SPCI_IOP_acq

This duration event counts number of clocks spent by the 80303 I/O processor (includes the bridge, dma Ch-2, and satu) acquiring the secondary PCI interface. The counter increments on every clock cycle after the processor has requested use of the secondary PCI bus but has not actively driven the secondary PCI bus as a master. The counter will also increment for all clock cycles when this agent's *Request Signal* is asserted but bus ownership currently belongs to another master. This is an event primitive, used in conjunction with another event primitive (number of grants granted to the 80303 I/O processor) to calculate the average acquisition latency for the processor.

21.3.5.14 M4_SPCI_IOP_own

This duration event counts the duration for which the 80303 I/O processor is the master on the secondary PCI interface. The counter increments on every clock cycle during which the processor is the bus master.

21.3.6 Mode 5: Intel® 80303 I/O Processor Bus and Agents Events

Programming Mode 5 (M5) in the ESR, enables 80303 I/O processor performance monitoring on the internal bus. In addition, performance monitoring is done for selected agents. In this mode, monitored agents are: DMA channels (Ch-0, Ch-1 and Ch-2) and the Application Accelerator. All counters are clocked at internal bus frequency. The following sections describe monitored events in Mode 5.

21.3.6.1 M5_IBus_idle

This duration event increments the counter every internal bus idle cycle. An idle cycle occurs when there is no activity on the bus due to data being transferred and/or the bus is not in an overhead cycle. An overhead cycle is a cycle when a master owns the bus, however the master is unable to send data or the target is unable to receive data - hence no data is transferred.

21.3.6.2 M5_IBus_busy

This duration event increments the counter on every internal bus data cycle. This enables calculation of data utilization of the bus.

21.3.6.3 M5_AA_acq

This duration event counts the number of clocks spent by the Application Accelerator (AA) acquiring the internal bus interface. The counter increments on every clock cycle after the AA has requested use of the bus but has not actively driven the bus as a master. The counter will also increment for all clock cycles when this agent's *Request Signal* is asserted but bus ownership currently belongs to another master. This is an event primitive, used in conjunction with another event primitive (number of grants granted to AA) to calculate the average acquisition latency.

21.3.6.4 M5_AA_own

This duration event counts the duration for which the AA is the master on the internal bus. The counter increments on every clock cycle during which the AA is the bus master.

21.3.6.5 M5_DMA0_acq

This duration event counts the number of clocks spent by DMA Ch-0 acquiring the internal bus interface. The counter increments on every clock cycle after Ch-0 has requested use of the bus but has not actively driven the internal bus as a master. The counter will also increment for all clock cycles when this agent's *Request Signal* is asserted but bus ownership currently belongs to another master. This is an event primitive, used in conjunction with another event primitive (number of grants granted to Ch-0) to calculate the average acquisition latency.

21.3.6.6 M5_DMA0_own

This duration event counts the duration for which DMA Ch-0 is the master on the internal bus. The counter increments on every clock cycle during which Ch-0 is the bus master.

21.3.6.7 M5_DMA1_acq

This duration event counts the number of clocks spent by DMA Ch-1 acquiring the internal bus interface. The counter increments on every clock cycle after Ch-1 has requested use of the bus but has not actively driven the internal bus as a master. The counter will also increment for all clock cycles when this agent's *Request Signal* is asserted but bus ownership currently belongs to another master. This is an event primitive, used in conjunction with another event primitive (number of grants granted to Ch-1) to calculate the average acquisition latency.

21.3.6.8 M5_DMA1_own

This duration event counts the duration for which DMA Ch-1 is the master on the internal bus. The counter increments on every clock cycle during which Ch-1 is the bus master.

21.3.6.9 M5_DMA2_acq

This duration event counts the number of clocks spent by DMA Ch-2 acquiring the internal bus interface. The counter increments on every clock cycle after Ch-2 has requested use of the bus but has not actively driven the internal bus as a master. The counter will also increment for all clock cycles when this agent's *Request Signal* is asserted but bus ownership currently belongs to another master. This is an event primitive, used in conjunction with another event primitive (number of grants granted to Ch-2) to calculate the average acquisition latency.

21.3.6.10 M5_DMA2_own

This duration event counts the duration for which DMA Ch-2 is the master on the internal bus. The counter increments on every clock cycle during which Ch-2 is the bus master.

21.3.6.11 M5_AA_gnt

This occurrence event monitors the number of times the AA is granted the bus. It increments the counter when the AA is the bus master. The counter is incremented once for every new transaction. For multi-cycle transactions, the counter increments once on the first cycle.

21.3.6.12 M5_DMA0_gnt

This occurrence event monitors the number of times DMA Ch-0 is granted the bus. It increments the counter when DMA Ch-0 is the bus master. The counter is incremented once for every new transaction. For multi-cycle transactions, the counter increments once on the first cycle.

21.3.6.13 M5_DMA1_gnt

This occurrence event monitors the number of times DMA Ch-1 is granted the bus. It increments the counter when DMA Ch-1 is the bus master. The counter is incremented once for every new transaction. For multi-cycle transactions, the counter increments once on the first cycle.

21.3.6.14 M5_DMA2_gnt

This occurrence event monitors the number of times DMA Ch-2 is granted the bus. It increments the counter when DMA Ch-2 is the bus master. The counter is incremented once for every new transaction. For multi-cycle transactions, the counter increments once on the first cycle.

21.3.7 Mode 6: Intel® 80303 I/O Processor Bus and Agents Events

Programming Mode6 (M6) in the ESR enables performance monitoring on the 80303 I/O processor internal bus. In addition, performance monitoring is also done for selected agents. In this mode, the monitored agents are Primary Address Translation Unit (PATU), Secondary Address Translation Unit (SATU) and i960 processor. All counters are clocked at the internal bus frequency. The following sections describe the monitored events in Mode 6.

21.3.7.1 M6_core_acq

This duration event counts number of clocks spent by i960 processor acquiring the internal bus interface. The counter increments every clock cycle after the i960 processor has requested use of the bus, but has not actively driven the internal bus as a master. The counter also increments for all clock cycles when the agent *Request Signal* is asserted, but bus ownership currently belongs to another master.

21.3.7.2 M6_core_own

This duration event counts the duration for which the i960 processor is the master on the internal bus. The counter increments on every clock cycle when the i960 processor is the bus master.

21.3.7.3 M6_PATU_acq

This duration event counts the number of clocks spent by PATU acquiring the internal bus interface. The counter increments on every clock cycle after the PATU has requested use of the bus but has not actively driven the internal bus as a master. The counter will also increment for all clock cycles when this agent's *Request Signal* is asserted but bus ownership currently belongs to another master. This is an event primitive, used in conjunction with another event primitive (number of grants granted to PATU) to calculate the average acquisition latency for the unit.

21.3.7.4 M6_PATU_own

This duration event counts the duration for which the PATU is the master on the internal bus. The counter increments on every clock cycle during which the PATU is the bus master.

21.3.7.5 M6_SATU_acq

This duration event counts the number of clocks spent by SATU acquiring the internal bus interface. The counter increments on every clock cycle after the SATU has requested use of the bus but has not actively driven the internal bus as a master. The counter will also increment for all clock cycles when this agent's *Request Signal* is asserted but bus ownership currently belongs to another master. This is an event primitive, used in conjunction with another event primitive (number of grants granted to SATU) to calculate the average acquisition latency for the unit.

21.3.7.6 M6_SATU_own

This duration event counts the duration for which the SATU is the master on the internal bus. The counter increments on every clock cycle during which the SATU is the bus master.

21.3.7.7 M6_PBOFF_time

This duration event counts the duration for which the backoff (PBOFF) signal is asserted by the PATU. This is an event primitive, used in conjunction with another event primitive (PBOFF_cnt) to calculate the average duration. The backoff signal is asserted by the PATU when it is busy with an outbound read transaction and the Bus Interface Unit (BIU) attempts to perform another transaction before the read transaction completes.

21.3.7.8 M6_PBOFF_cnt

This occurrence event counts the number of times the PATU asserts the PBOFF signal. This occurrence event increments the counter on every instance of PBOFF assertion.

21.3.7.9 M6_SBOFF_time

This duration event counts the duration for which the backoff (SBOFF) signal is asserted by the SATU. This is an event primitive, used in conjunction with another event primitive (SBOFF_cnt) to calculate the average duration. The backoff signal is asserted by the SATU when it is busy with an outbound read transaction and the Bus Interface Unit (BIU) attempts to perform another transaction before the read transaction completes.

21.3.7.10 M6_SBOFF_cnt

This occurrence event counts the number of times the SATU asserts the SBOFF signal. This occurrence event increments the counter on every instance of SBOFF assertion.

21.3.7.11 M6_PATU_gnt

This occurrence event monitors the number of times the PATU is granted the bus. This event increments the counter when the PATU is the bus master. The counter is incremented once for every new transaction. For multi-cycle transactions, the counter increments once on the first cycle.

21.3.7.12 M6_SATU_gnt

This occurrence event monitors the number of times the SATU is granted the bus. This event increments the counter when the SATU is the bus master. The counter is incremented once for every new transaction. For multi-cycle transactions, the counter increments once on the first cycle.

21.3.7.13 M6_core_gnt

This occurrence event monitors the number of times the core is granted the bus. This event increments the counter when the core is the bus master. The counter is incremented once for every new transaction. For multi-cycle transactions, the counter increments once on the first cycle.

0.0.0.1. M6_ATU_retry

This occurrence event counts the number of retries issued by the Primary Address Translation Unit (PATU) on the primary PCI bus due to the inbound write queue being unable to accept a new transaction. Retries issued by the PATU in response to configuration writes will not be included in this metric.

21.3.8 Mode 7: Intel® 80303 I/O Processor Internal Bus, Secondary PCI Bus and Primary PCI Bus Events

Programming Mode 7 (M7) in the ESR enables performance monitoring on the internal bus, secondary PCI bus and primary PCI bus. In addition, performance monitoring is done for external agents (80303 I/O processor and external masters 0.5) on the secondary bus and for 80303 I/O processor on the primary bus. Master0 designates the external secondary PCI device that is connected to the REQ0 and GNT0 signals of the internal arbiter in the 80303 I/O processor. The nomenclature is similar for all other external PCI masters; Master 1 through Master 5.

In this mode, counters monitoring events on the internal bus are clocked at the internal bus frequency and counters monitoring PCI events are clocked at the respective PCI bus frequencies. The following sections describe the monitored events in Mode 7.

21.3.8.1 M7_IBus_idle

This duration event increments the counter every internal bus idle cycle. An idle cycle occurs when there is no activity on the bus due to data being transferred and/or the bus is not in an overhead cycle. An overhead cycle is a cycle when a master owns the bus, however the master is unable to send data or the target is unable to receive data - hence no data is transferred.

21.3.8.2 M7_IBus_busy

This duration event increments the counter every internal bus data cycle. This enables calculation of data utilization of the bus.

21.3.8.3 M7_SPCIbus_idle

This duration event increments the counter every secondary PCI bus idle cycle. An idle cycle occurs when there is no activity on the bus due to data being transferred and/or the bus is not in an overhead cycle. An overhead cycle is a cycle when a master owns the bus, however the master is unable to send data or the target is unable to receive data - hence no data is transferred.

21.3.8.4 M7_SPCIbus_busy

This duration event increments the counter every secondary PCI data cycle. Data cycles comprise of two instances:

- The 80303 I/O processor as a master on the bus is involved in data transfers to other masters.
- External masters initiate data transfers to either the 80303 I/O processor or to other masters on the bus.

21.3.8.5 M7_SPCI_IOP_own

This duration event counts the duration for which the 80303 I/O processor is the master on the secondary PCI bus. The counter increments on every clock cycle during which the processor is the bus master.

21.3.8.6 M7_D0_own

This duration event counts the duration for which PCI Master 0 is the master on the secondary PCI bus. The counter increments on every clock cycle during which PCI Master 0 is the bus master.

21.3.8.7 M7_D1_own

This duration event counts the duration for which PCI Master 1 is the master on the secondary PCI bus. The counter increments on every clock cycle during which PCI Master 1 is the bus master.

21.3.8.8 M7_D2_own

This duration event counts the duration for which PCI Master 2 is the master on the secondary PCI bus. The counter increments on every clock cycle during which PCI Master 2 is the bus master.

21.3.8.9 M7_D3_own

This duration event counts the duration for which PCI Master 3 is the master on the secondary PCI bus. The counter increments on every clock cycle during which PCI Master 3 is the bus master.

21.3.8.10 M7_D4_own

This duration event counts the duration for which PCI Master 4 is the master on the secondary PCI bus. The counter increments on every clock cycle during which PCI Master 4 is the bus master.

21.3.8.11 M7_D5_own

This duration event counts the duration for which PCI Master 5 is the master on the secondary PCI bus. The counter increments on every clock cycle during which PCI Master 5 is the bus master.

21.3.8.12 M7_PPCI_IOP_own

This duration event counts the duration for which the 80303 I/O processor is the master on the primary PCI bus. The counter increments on every clock cycle during which the processor is the bus master.

21.3.8.13 M7_PPCIbus_idle

This duration event increments the counter ever y primary PCI bus idle cycle. An idle cycle occurs when there is no activity on the bus due to data being transferred and/or the bus is not in an overhead cycle. An overhead cycle is a cycle when a master owns the bus, however the master is unable to send data or the target is unable to receive data - hence no data is transferred.

21.3.8.14 M7_PPCIbus_busy

This duration event increments counter every PCI data cycle. Data cycles comprise two instances:

- The 80303 I/O processor as a master on the bus is involved in data transfers to other masters.
- External masters initiate data transfers to either the 80303 I/O processor or to other masters on the bus.

21.4 Interrupts

The Programmable Event Counters and the Global Time Stamp Counter generate interrupts to the 80303 I/O processor. When bit 0 (enable/disable bit) in the Global Timer Mode Register (GTMR) is set, the Global Time Stamp Counter interrupts the core processor on an overflow. Any Programmable Event Counter interrupts the processor on an overflow by setting the corresponding bit in the Event Monitoring Interrupt Status Register (EMISR). Setting a bit in this register generates an interrupt to the XINT6# interrupt pin of the core processor. When multiple counters overflow, each counter that overflows sets the corresponding bit in the EMISR.

The XINT6# pin of the core processor receives interrupts from multiple sources through the XINT6 interrupt latch. A valid interrupt from any source sets the bit in the latch and outputs a level-sensitive interrupt to the core processor XINT6# pin.

21.5 Reset Conditions

The Global Time Stamp Counter is cleared upon deassertion of the Internal Bus Reset signal. The Global Timer Mode Register (GTMR) is cleared on reset. The Event Select Register (ESR) defaults to Mode 0 upon reset: performance monitoring is disabled and all counters are disabled in this mode. The Programmable Event Counters (PECRx) values are undefined upon reset.

21.6 Register Definitions

The performance monitoring facility on 80303 I/O processor consists of eighteen (18) memory-mapped registers for controlling operation and monitoring various events. Each register is 32-bits wide. Each of these registers is accessed as a memory-mapped 32-bit register with a unique memory address. Access is accomplished through regular memory-format instructions from the i960 core processor.

Three registers control the mode of operation. They are; Global Timer Mode Register (GTMR), Event Monitoring Interrupt Status Register (EMISR), and the Event Select Register (ESR). The GTMR controls operation of the Global Time Stamp Counter. The EMISR is used to indicate an overflow condition in any counter during performance monitoring. An overflow condition in the Global Time Stamp Counter is also indicated in the EMISR when the mode is enabled. The value programmed into the Event Select Register (ESR) determines the monitored interface.

Fourteen(14) registers(PECR1 - PECR14) contain the current count value from the programmable event counters (PEC1 - PEC14). The Global Time Stamp Register (GTSR) contains the current count value of the Time Stamp Counter. The event registers (PECR1 - PECR14) and the GTSR are read-only registers.

Table 21-4 identifies the registers used for performance monitoring. Each register is described in the subsections following Table 21-4.

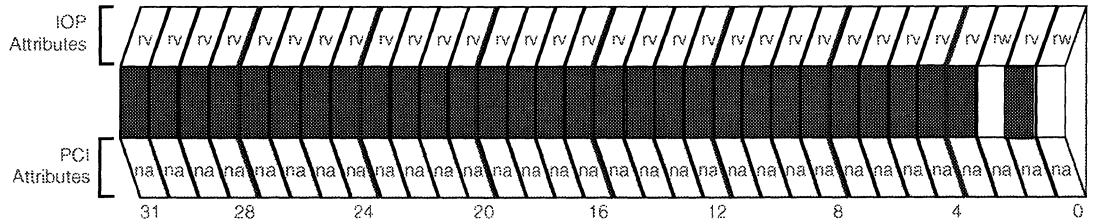
Table 21-4. Event Monitor Register Table

Section, Register Name - Acronym (Page)
Section 21.6.1, "Global Timer Mode Register (GTMR)" on page 21-23
Section 21.6.2, "Event Select Register (ESR)" on page 21-24
Section 21.6.3, "Event Monitoring Interrupt Status Register (EMISR)" on page 21-25
Section 21.6.4, "Global Time Stamp Register (GTSR)" on page 21-27
Section 21.6.5, "Programmable Event Counter Register (PECRx)" on page 21-27

21.6.1 Global Timer Mode Register (GTMR)

The Global Timer Mode Register (GTMR) programs the mode of operation or indicates the current mode of the Global Time Stamp Counter as shown in Table 21-5. This is a 32-bit, read-write register. Bit 0 controls the interrupt capability of the Global Time Stamp Counter. When enabled, an interrupt is generated to the i960 core processor processor on the XINT6# interrupt pin when the Global Time Stamp Counter overflows. Bit 2 is an enable/disable bit. When set (1), the Programmable Event Counters and the Global Time Stamp Counter are disabled and retain their previous values. This bit needs to be rewritten to enable all counters.

Table 21-5. Global Timer Mode Register (GTMR)



Intel® 80303 Core Internal Bus Address
0000 1100H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

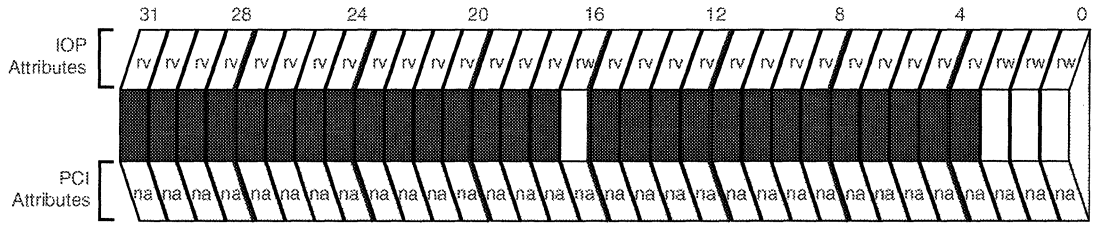
Bit	Default	Description
31:03	0	Reserved
2	0_2	Bit value determines if the Global Time Stamp Counter and the Programmable Event Counters are enabled or disabled. 0 = All counters enabled (enable counting) 1 = All counters disabled (disable counting)
1	0_2	Reserved
0	0_2	Bit value determines whether the Global Time Stamp Counter interrupts the processor on an overflow condition. 0 = Interrupt disabled 1 = Interrupt enabled

21.6.2 Event Select Register (ESR)

The Event Select Register (ESR) controls the specific mode of operation or indicates the current mode of performance monitoring. There are eight (8) modes supported. To change the monitored mode, it is necessary to write the entire ESR. The Programmable Event Counters and the Global Time Stamp Counter are reset when a new value is written to the ESR. Performance monitoring is disabled in the default mode.

Table 21-6 describes the various monitoring modes and the programmed values for those modes.

Table 21-6. Event Select Register (ESR) - External Version



Intel® 80303 Core Internal Bus Address
0000 1104H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:17	0	Reserved
16	0_2	PECRx Master Interrupt Enable: When set (1), any/all the programmable event counters will interrupt the Intel® i960® processor on an overflow. When clear (0), none of the programmable event counters will interrupt the processor on an overflow. In this mode, any counter that has an overflow condition will roll over to zero and start incrementing.
15:3	0	Reserved
2:0	0	Value in this bit field determines the monitored interface on the Intel® 80303 I/O processor. 000 Mode 0Performance Monitoring Disabled 001 Mode 1Primary PCI Bus & Internal Agents 010 Mode 2Secondary PCI Bus & Internal Agents 011 Mode 3Secondary PCI Bus & PCI Agents 100 Mode 4Secondary PCI Bus & PCI Agents (external masters 3..5) 101 Mode 5 80303 I/O processor internal bus, DMA Channels & AA 110 Mode 6 80303 I/O processor internal bus, PATU, SATU & i960 processor 111 Mode 7 80303 I/O processor internal bus, PCI buses (primary & secondary)

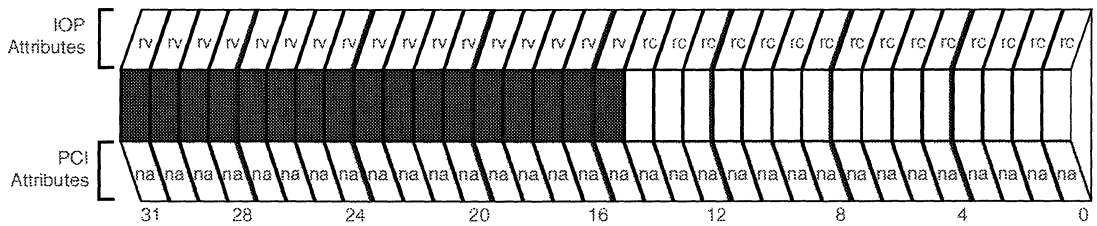
21.6.3 Event Monitoring Interrupt Status Register (EMISR)

The Event Monitoring Interrupt Status Register (EMISR) generates interrupts to the 80303 I/O processor. Bits 14:0 when set indicate an overflow condition in either the Global Time Stamp Counter or the Programmable Event Counters. This generates an interrupt on the XINT6# pin of the core processor. Bits 14:0 can only be set by the Event Counters and/or the Global Time Stamp Counter and can only be cleared by the core processor.

When this register is read by the core processor and multiple bits are set, it is the responsibility of the application software to record the value and prioritize the sequence of actions. Any bit (bits 14:0) once set is cleared by writing a 1 to the specific bit field.

Note: It is the responsibility of the application software to clear the individual bit fields in the register once a new mode is programmed into the ESR.

Table 21-7. Event Monitoring Interrupt Status Register (EMISR) (Sheet 1 of 2)

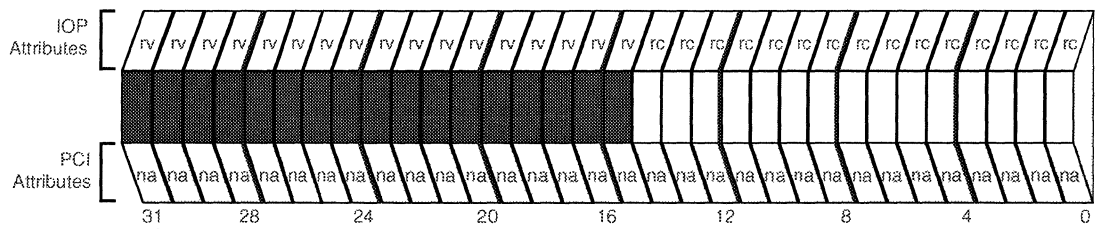


Intel® 80303 Core Internal Bus Address
0000 1108H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:15	0	Reserved
14	0_2	Bit value indicates status of the Programmable Event Counter 14 (PEC14) during event monitoring. When clear (0), no PEC14 overflow interrupt is pending. When set (1), a PEC14 overflow interrupt is pending.
13	0_2	Bit value indicates the status of the Programmable Event Counter 13 (PEC13) during event monitoring. When clear (0), no PEC13 overflow interrupt is pending. When set (1), a PEC13 overflow interrupt is pending.
12	0_2	Bit value indicates the status of the Programmable Event Counter 12 (PEC12) during event monitoring. When clear (0), no PEC12 overflow interrupt is pending. When set (1), a PEC12 overflow interrupt is pending.
11	0_2	Bit value indicates the status of the Programmable Event Counter 11 (PEC11) during event monitoring. When clear (0), no PEC11 overflow interrupt is pending. When set (1), a PEC11 overflow interrupt is pending.
10	0_2	Bit value indicates the status of the Programmable Event Counter 10 (PEC10) during event monitoring. When clear (0), no PEC10 overflow interrupt is pending. When set (1), a PEC10 overflow interrupt is pending.
9	0_2	Bit value indicates the status of the Programmable Event Counter 9 (PEC9) during event monitoring. When clear (0), no PEC9 overflow interrupt is pending. When set (1), a PEC9 overflow interrupt is pending.
8	0_2	Bit value indicates the status of the Programmable Event Counter 8 (PEC8) during event monitoring. When clear (0), no PEC8 overflow interrupt is pending. When set (1), a PEC8 overflow interrupt is pending.

Table 21-7. Event Monitoring Interrupt Status Register (EMISR) (Sheet 2 of 2)



Intel® 80303 Core Internal Bus Address
0000 1108H

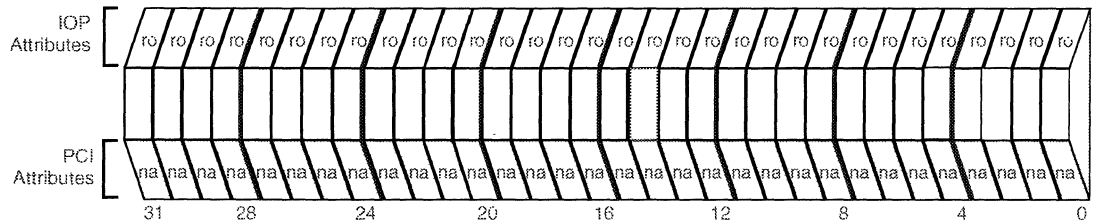
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
7	0_2	Bit value indicates the status of the Programmable Event Counter 7 (PEC7) during event monitoring. When clear (0), no PEC7 overflow interrupt is pending. When set (1), a PEC7 overflow interrupt is pending.
6	0_2	Bit value indicates the status of the Programmable Event Counter 6 (PEC6) during event monitoring. When clear (0), no PEC6 overflow interrupt is pending. When set (1), a PEC6 overflow interrupt is pending.
5	0_2	Bit value indicates the status of the Programmable Event Counter5 (PEC5) during event monitoring. When clear (0), no PEC5 overflow interrupt is pending. When set (1), a PEC5 overflow interrupt is pending.
4	0_2	Bit value indicates the status of the Programmable Event Counter4 (PEC4) during event monitoring. When clear (0), no PEC4 overflow interrupt is pending. When set (1), a PEC4 overflow interrupt is pending.
3	0_2	Bit value indicates Programmable Event Counter3 (PEC3) status during event monitoring. When clear (0), no PEC3 overflow interrupt is pending. When set (1), a PEC3 overflow interrupt is pending.
2	0_2	Bit value indicates the status of the Programmable Event Counter2 (PEC2) during event monitoring. When clear (0), no PEC2 overflow interrupt is pending. When set (1), a PEC2 overflow interrupt is pending.
1	0_2	Bit value indicates the status of the Programmable Event Counter1 (PEC1) during event monitoring. When clear (0), no PEC1 overflow interrupt is pending. When set (1), a PEC1 overflow interrupt is pending.
0	0_2	Bit value indicates the status of the Global Time Stamp Counter (GTS) during event monitoring. When clear (0), no GTS overflow interrupt is pending. When set (1), a GTS overflow interrupt is pending.

21.6.4 Global Time Stamp Register (GTSR)

The Global Time Stamp register (GTSR) is a 32-bit, read-only register. Writes to the GTSR have no effect. The GTSR contains the current count value of the Global Time Stamp Counter. The counter frequency is one-quarter the Internal Bus clock frequency. When a new mode is chosen by writing a value to the ESR, this register is reset to zero. This register can be read at any time and will return the current count value.

Table 21-8. Global Time Stamp Register - GTSR



Bit	Default	Description
31:00	X	This is a 32-bit, read-only register. When accessed, it returns the current count value in the Global Time Stamp Counter.

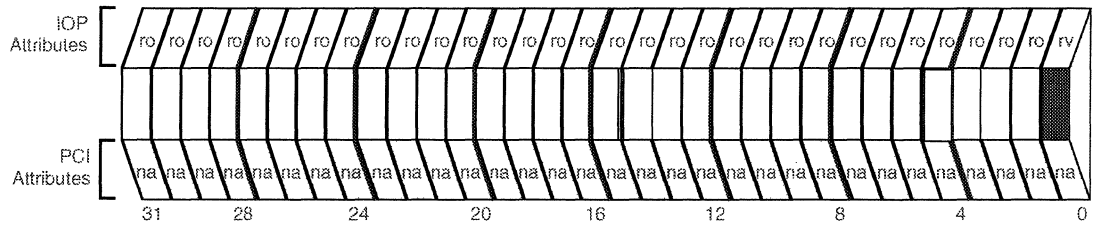
21.6.5 Programmable Event Counter Register (PECRx)

There are 14 programmable event counter registers (PECR1 - PECR14) that contain the current count value in the 14 event counters (PEC1 - PEC14). Each register is a 32-bit, read-only register. Writing to the Programmable Event Counter Registers (PECR1 - PECR14) has no effect.

The value in any register is incremented based on the current programmed ESR value and the descriptions shown in Table 21-1 through Table 21-4. When a new mode is chosen by writing a value to the ESR, these registers are reset to zero. Each of these registers can be read at any time, and return the current count value with a two event granularity (see note).

Note: These counters will increment from bits 31:1 with a two event granularity. For instance, after one event has occurred the counter will read 0000 0000H, while after two events has occurred the counter will read 0000 0002H.

Table 21-9. Programmable Event Counter Register - PECRx



Intel® 80303 Core Internal Bus Address

PECR1 0000 1114H
PECR2 0000 1118H
PECR3 0000 111CH
PECR4 0000 1120H
PECR5 0000 1124H
PECR6 0000 1128H
PECR7 0000 112CH
PECR8 0000 1130H
PECR9 0000 1134H
PECR10 0000 1138H
PECR11 0000 113CH
PECR12 0000 1140H
PECR13 0000 1144H
PECR14 0000 1148H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:01	X	This is a 32-bit, read-only register. When accessed, it returns the current count value in the respective event counter.
00	0_2	Reserved

I²C Bus Interface Unit 22

This chapter describes the I²C (Inter-Integrated Circuit) bus interface unit of the Intel® 80303 I/O processor, including the operation modes and setup. Throughout this manual, this peripheral is referred to as the I²C unit.

22.1 Overview

The I²C Bus Interface Unit allows the 80303 I/O processor to serve as a master and slave device residing on the I²C bus. The I²C bus is a serial bus developed by Philips Corporation consisting of a two-pin interface. **SDA** is the data pin for input and output functions and **SCL** is the clock pin for reference and control of the I²C bus.

The I²C bus allows the 80303 I/O processor to interface to other I²C peripherals and microcontrollers for system management functions. The serial bus requires a minimum of hardware for an economical system to relay status and reliability information on the 80303 I/O processor subsystem to an external device.

The I²C Bus Interface Unit is a peripheral device that resides on the 80303 I/O processor internal bus. Data is transmitted to and received from the I²C bus via a buffered interface. Control and status information is relayed through a set of memory-mapped registers. Refer to the I²C Bus Specification for complete details on I²C bus operation.

22.2 Theory of Operation

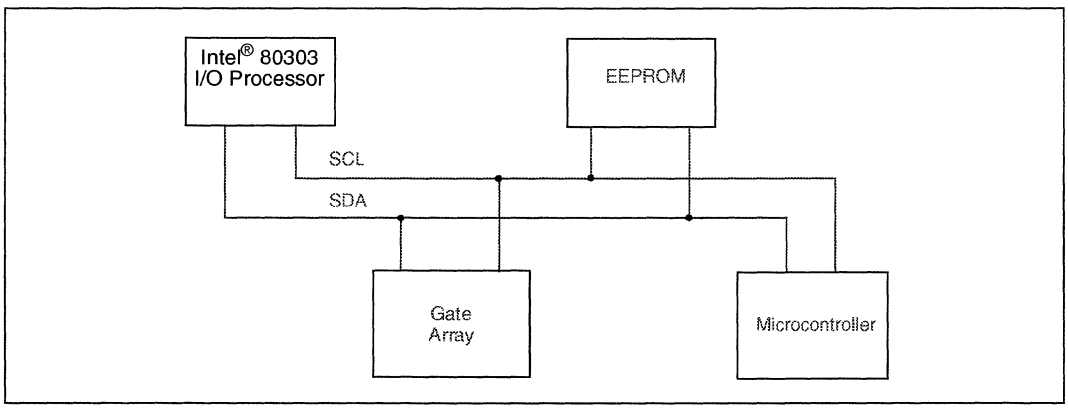
The I²C bus defines a serial protocol for passing information between agents on the I²C bus using only a two pin interface. The interface consists of a Serial Data/Address (**SDA**) line and a Serial Clock Line (**SCL**). Each device on the I²C bus is recognized by a unique 7-bit address and can operate as a transmitter or as a receiver. In addition to transmitter and receiver, the I²C bus uses the concept of master and slave. Table 22-1 lists the I²C device types.

Table 22-1. I²C Bus Definitions

I²C Device	Definition
Transmitter	Sends data to the I²C bus.
Receiver	Receives data from the I²C bus.
Master	Initiates a transfer, generates the clock signal, and terminates the transactions.
Slave	The device addressed by a master.
Multi-master	More than one master can attempt to control the bus at the same time without corrupting the message.
Arbitration	Procedure to ensure that, when more than one master simultaneously tries to control the bus, only one is allowed. This procedure ensures that messages are not corrupted.

As an example of I²C bus operation, consider the case of the 80303 I/O processor acting as a master on the bus (see Figure 22-1). The 80303 I/O processor, as a master, addresses an EEPROM as a slave to receive data. The 80303 I/O processor is a master-transmitter and the EEPROM is a slave-receiver. When the 80303 I/O processor reads data, the 80303 I/O processor is a master-receiver and the EEPROM is a slave-transmitter. In both cases, the master generates the clock, initiates the transaction and terminates it.

Figure 22-1. I²C Bus Configuration Example



The I²C bus allows for a multi-master system, which means more than one device can initiate data transfers at the same time. To support this feature, the I²C bus arbitration relies on the wired-AND connection of all I²C interfaces to the I²C bus. Two masters can drive the bus simultaneously provided they are driving identical data. The first master to drive **SDA** high while another master drives **SDA** loses the arbitration. The **SCL** line consists of a synchronized combination of clocks generated by the masters using the wired-AND connection to the **SCL** line.

The I²C bus serial operation uses an open-drain wired-AND bus structure, which allows multiple devices to drive the bus lines and to communicate status about events such as arbitration, wait states, error conditions and so on. For example, when a master drives the clock (**SCL**) line during a data transfer, it transfers a bit on every instance that the clock is high. When the slave is unable to accept or drive data at the rate that the master is requesting, the slave can hold the clock line low between the high states to insert a wait interval. The master's clock can only be altered by a slow slave peripheral keeping the clock line low or by another master during arbitration.

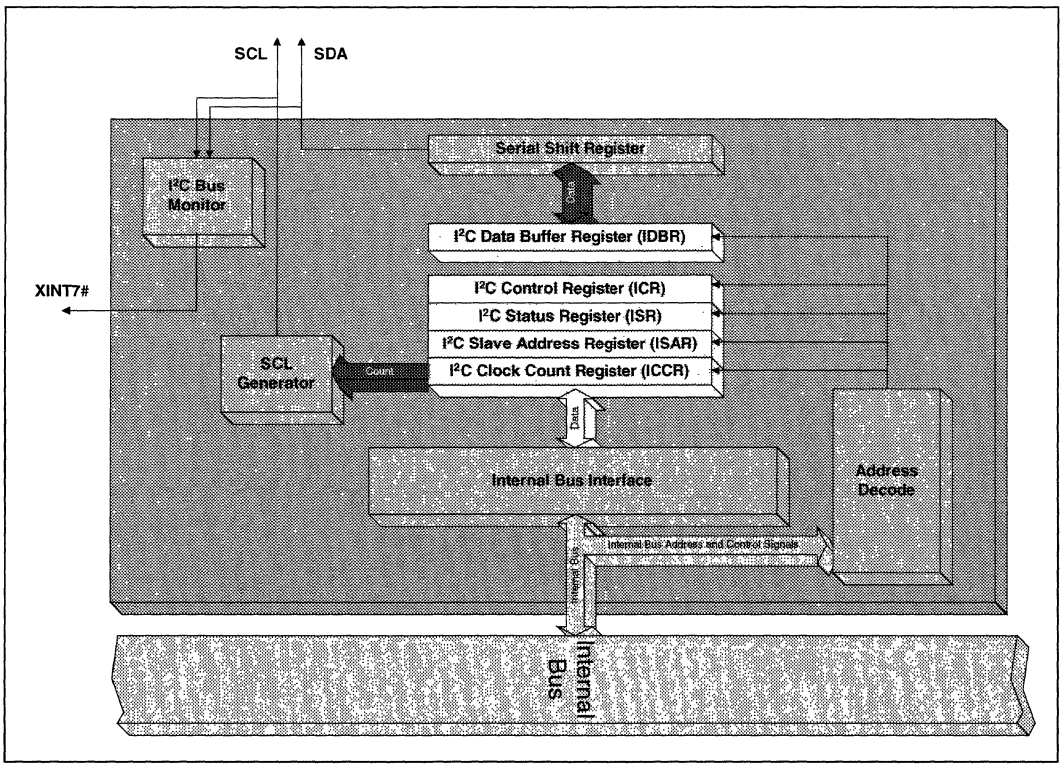
I²C transactions are either initiated by the 80303 I/O processor as a master or are received by the processor as a slave. Both conditions may result in the processor doing reads, writes, or both to the I²C bus.

22.2.1 Operational Blocks

The I²C Bus Interface Unit is a slave peripheral device that is connected to the internal bus. The 80303 I/O processor interrupt mechanism can be used for notifying the 80303 I/O processor that there is activity on the I²C bus. Polling can be also be used instead of interrupts, although it would be very cumbersome. Figure 22-2 shows a block diagram of the I²C Bus Interface Unit and its interface to the internal bus.

The I²C Bus Interface Unit consists of the two wire interface to the I²C bus, an 8-bit buffer for passing data to and from the 80303 I/O processor, a set of control and status registers, and a shift register for parallel/serial conversions.

Figure 22-2. I²C Bus Interface Unit Block Diagram



The I²C interrupts are signalled through 80303 I/O processor interrupt **XINT7#** and the XINT7 Interrupt Status Register (X7ISR) in the PCI and Peripheral Interrupt Controller (See Chapter 8, "PCI and Peripheral Interrupt Controller Unit"). The I²C Bus Interface Unit can set a bit within the X7ISR register when a buffer is full, buffer empty, slave address detected, arbitration lost, or bus error condition occurs. All interrupt conditions must be cleared explicitly by software. See Section 22.8.2, "I2C Status Register- ISR" on page 22-32) for details.

The I²C Data Buffer Register (IDBR) is an 8-bit data buffer that receives a byte of data from the shift register interface of the I²C bus on one side and parallel data from the 80303 I/O processor's internal bus on the other side. The serial shift register is not user accessible.

The control and status registers are located in the I²C memory-mapped address space (1680H to 1690H). The registers and their function are defined in Section 22.8.

The I²C Bus Interface Unit supports fast mode operation of 400 Kbits/sec. Fast mode logic levels, formats, and capacitive loading, and protocols are exactly the same as the 100 Kbits/sec standard mode. Because the data setup and hold times differ between the fast and standard mode, the I²C will be designed to meet the slower, standard mode requirements for these two specifications. Refer to the I²C Bus Specification for details.

22.2.2 I²C Bus Interface Modes

The I²C Bus Interface Unit can be in different modes of operation to accomplish a transfer. Table 22-2 summarizes the different modes.

Table 22-2. Modes of Operation

Mode	Definition
Master - Transmit	• I²C Bus Interface Unit acts as a master. • Used for a write operation. • I²C Bus Interface Unit sends the data. • I²C Bus Interface Unit is responsible for clocking. • Slave device will be in slave-receive mode
Master - Receive	• I²C Bus Interface Unit acts as a master. • Used for a read operation. • I²C Bus Interface Unit receives the data. • I²C Bus Interface Unit is responsible for clocking. • Slave device will be in slave-transmit mode
Slave - Transmit	• I²C Bus Interface Unit acts as a slave. • Used for a read (master) operation. • I²C Bus Interface Unit sends the data. • Master device will be in master-receive mode.
Slave - Receive (default)	• I²C Bus Interface Unit acts as a slave. • Used for a write (master) operation. • I²C Bus Interface Unit receives the data. • Master device will be in master-transmit mode.

While the I²C Bus Interface Unit is in idle mode (neither receiving or transmitting serial data), the unit defaults to Slave-Receive mode. This allows the interface to monitor the bus and receive any slave addresses that might be intended for the 80303 I/O processor.

When the I²C Bus Interface Unit receives an address that matches the 7-bit address found in the I²C Slave Address Register (ISAR) or the General Call Address (00H), the interface will either remain in Slave-Receive mode or transition to Slave-Transmit mode. This is determined by the Read/Write (R/W#) bit (the least significant bit of the byte containing the slave address). If the R/W# bit is low, the master initiating the transaction intends to do a write and the I²C Bus Interface Unit will remain in Slave-Receive mode. If the R/W# is high, the initiating master wants to read data and the slave transitions to Slave-Transmit mode. Slave operation is further defined in Section 22.3.6, "Slave Operations" on page 22-18.

When the 80303 I/O processor wants to initiate a read or write on the I²C bus, the I²C Bus Interface Unit will transition from the default Slave-Receive mode to Master-Transmit mode. If the 80303 I/O processor wants to write data, the interface remains in Master-Transmit mode after the address transfer has completed. (see Section 22.2.3.1, "START Condition" on page 22-6) for START information). If the 80303 I/O processor wants to read data, the I²C Bus Interface Unit will transmit the start address, then transition to Master-Receive mode. Master operation is further defined in Section 22.3.5, "Master Operations" on page 22-14.

22.2.3 Start and Stop Bus States

The I²C bus defines a transaction START and a transaction STOP bus state that are used at the beginning and end of the transfer of one to an unlimited number of bytes on the bus.

The 80303 I/O processor uses the START and STOP bits in the I²C Control Register (ICR) to:

- initiate an additional byte transfer
- initiate a START condition on the I²C bus
- enable Data Chaining (repeated START)
- initiate a STOP condition on the I²C bus

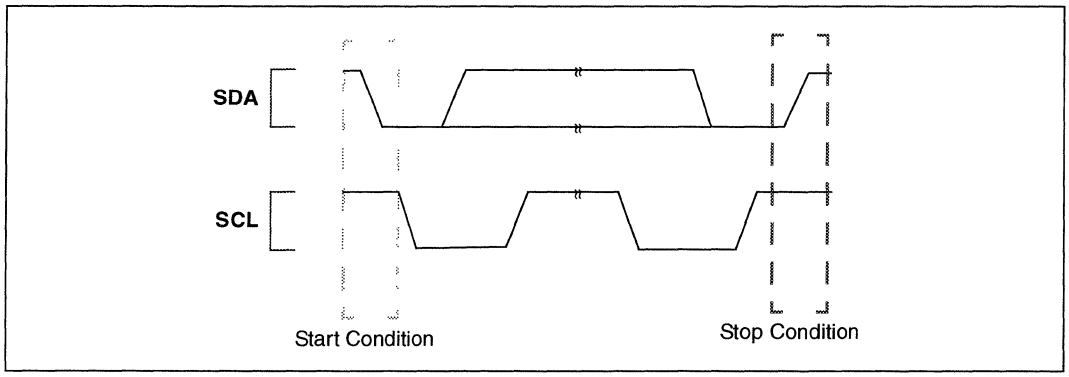
Table 22-3 summarizes the definition of the START and STOP bits in the ICR.

Table 22-3. START and STOP Bit Definitions

STOP bit	STAR T bit	Condition	Notes
0	0	No START or STOP	• No START or STOP condition is sent by the I²C Bus Interface Unit. This is used when multiple data bytes need to be transferred.
0	1	START Condition and Repeated START	• The I²C Bus Interface Unit will send a START condition and transmit the contents of the 8 bit IDBR after the START. The IDBR must contain the 7-bit address and the R/W# bit before a START is initiated. • For a repeated start, the IDBR contents will contain the target slave address and the R/W# bit. This enables multiple transfers to different slaves without giving up the bus. • The interface will stay in Master-Transmit mode if a write is used or transition to master-receive mode if a read is requested.
1	X	STOP Condition	• In Master-Transmit mode, the I²C Bus Interface Unit will transmit the 8-bit IDBR and then send a STOP on the I²C bus. • In Master-Receive mode, the Ack/Nack Control bit in the ICR must be changed to a negative Ack (see Section 22.3.3). The I²C Bus Interface Unit will write the Nack bit (Ack/Nack Control bit must be 1), receive the data byte in the IDBR, then send a STOP on the I²C bus.

Figure 22-3 shows the relationship between the **SDA** and **SCL** lines for a START and STOP condition.

Figure 22-3. Start and Stop Conditions



22.2.3.1 START Condition

The START condition (bits 1:0 of the ICR set to 01_2) initiates a master transaction or repeated START. Software must load the target slave address and the R/W# bit in the IDBR (see Section 22.8.4, "I2C Data Buffer Register- IDBR" on page 22-35) before setting the START ICR bit. The START and the IDBR contents are transmitted on the I^2C bus when the ICR Transfer Byte bit is set. The I^2C bus stays in master-transmit mode when a write is requested or enters master-receive mode when a read is requested. For a repeated start (a change in read or write or a change in the target slave address), the IDBR contains the updated target slave address and the R/W# bit. A repeated start enables multiple transfers to different slaves without giving up the bus.

The START condition is not cleared by the I^2C unit. When arbitration is lost while initiating a START, the I^2C unit may re-attempt the START when the bus becomes free. See Section 22.3.4, "Arbitration" on page 22-12 for details on how the I^2C unit functions under those circumstances.

22.2.3.2 No START or STOP Condition

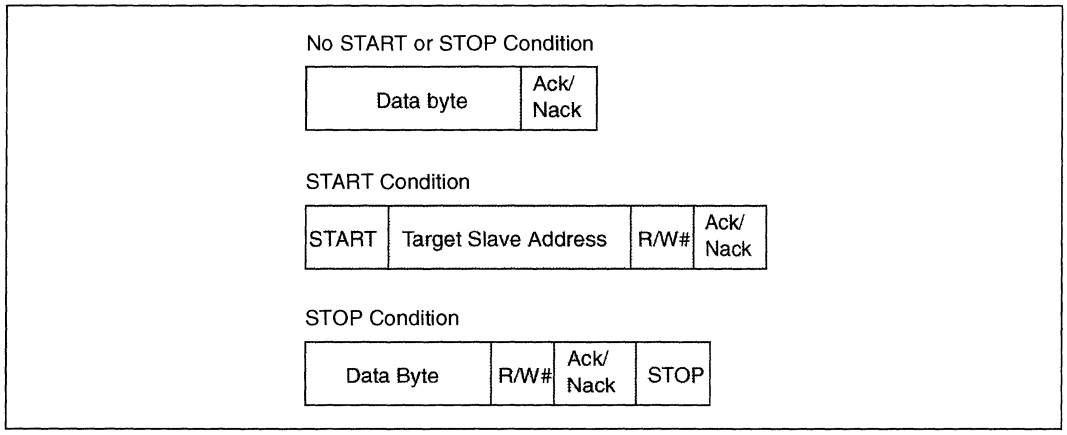
No START or STOP condition (bits 1:0 of the ICR set to 00_2) is used in master-transmit mode while the 80303 I/O processor is transmitting multiple data bytes (see Figure 22-3). Software writes the data byte, sets the IDBR Transmit Empty bit in the ISR (and interrupt when enabled), and clears the Transfer Byte bit in the ICR. The software then writes a new byte to the IDBR and sets the Transfer Byte ICR bit, which initiates the new byte transmission. This continues until the software sets the START or STOP bit. The START and STOP bits in the ICR are not automatically cleared by the I^2C unit after the transmission of a START, STOP or repeated START.

After each byte transfer (including the Ack/Nack bit) the I^2C unit holds the **SCL** line low (inserting wait states) until the Transfer Byte bit in the ICR is set. This action notifies the I^2C unit to release the **SCL** line and allow the next information transfer to proceed.

22.2.3.3 STOP Condition

The STOP condition (bits 1:0 of the ICR set to 10_2) terminates a data transfer. In master-transmit mode, the STOP bit and the Transfer Byte bit in the ICR must be set to initiate the last byte transfer (see Figure 22-3). In master-receive mode, to initiate the last transfer the 80303 I/O processor must set the Ack/Nack bit, the STOP bit, and the Transfer Byte bit in the ICR. Software must clear the STOP condition after it is transmitted.

Figure 22-4. START and STOP Conditions



22.3 I²C Bus Operation

The I²C Bus Interface Unit transfers in 1 byte increments. A data transfer on the I²C bus always follows the sequence:

1) START
2) 7-bit Slave Address
3) R/W# Bit
4) Acknowledge Pulse
5) 8 Bits of Data
6) Ack/Nack Pulse
7) Repeat of Step 5 and 6 for Required Number of Bytes
8) Repeated START (Repeat Step 1) or STOP

22.3.1 Serial Clock Line (SCL) Generation

The 80303 I/O processor's I²C unit is required to generate the I²C clock output when in master mode (either receive or transmit). **SCL** clock generation is accomplished through the use of the ICCR value, which is programmed at initialization. The ICCR value is used in the following equation to determine the **SCL** transition period:

Equation 22-1. SCL Transition Period

```
SCL Transition Period = ICCR Decimal Value * 80303 I/O processor Internal Bus
Clock Period
```

The **SCL** transition period is the amount of time the clock spends in the high or low state. When wait states are inserted or synchronization with another master is necessary, the I²C unit performs the necessary clock synchronization. The ICCR provides a simple method for determining I²C clock frequencies. Table 22-4 details sample programming values for the ICCR.

Table 22-4. ICCR Programming Values

Intel® 80303 I/O Processor Internal Bus Frequency	ICCR Value			SCL Transition Period	I²C Clock Frequency = [1/(SCL Transition Per. * 2)]
50 MHz	000111111_2	03FH	63	1.26 µs	396.8 KHz
	011111010_2	0FAH	250	5.00 µs	100.0 KHz
66 MHz	001010100_2	054H	84	1.26 µs	396.8 KHz
	101001101_2	14DH	333	5.00 µs	100.1 KHz
100 MHz	001111110_2	07DH	125	1.25 µs	400.0 KHz
	111110100_2	1F4H	500	5.00 µs	100.0 KHz

Programming a value less than 30H results in undefined behavior.

22.3.2 Data and Addressing Management

Data and slave addressing is managed via the I²C Data Buffer Register (IDBR) and the I²C Slave Address Register (ISAR). The IDBR (see Section 22.8.4, "I2C Data Buffer Register- IDBR" on page 22-35) contains data or a slave address and R/W# bit. The ISAR contains the 80303 I/O processor's programmable slave address. Data coming into the I²C unit is received into the IDBR after a full byte is received and acknowledged. To transmit data, the processor writes to the IDBR, and the I²C unit passes this onto the serial bus when the Transfer Byte bit in the ICR is set. See Section 22.8.1, "I2C Control Register- ICR" on page 22-29.

When the I²C unit is in transmit mode (master or slave):

1. Software writes data to the IDBR over the internal bus. This initiates a master transaction or sends the next data byte, after the IDBR Transmit Empty bit is sent.
2. The I²C unit transmits the data from the IDBR when the Transmit Empty bit in the ICR is set.
3. When enabled, an IDBR Transmit Empty interrupt is signalled when a byte is transferred on the I²C bus and the acknowledge cycle is complete.
4. When the I²C bus is ready to transfer the next byte before the processor has written the IDBR (and a STOP condition is not in place), the I²C unit inserts wait states until the processor writes a new value into the IDBR and sets the ICR Transfer Byte bit.

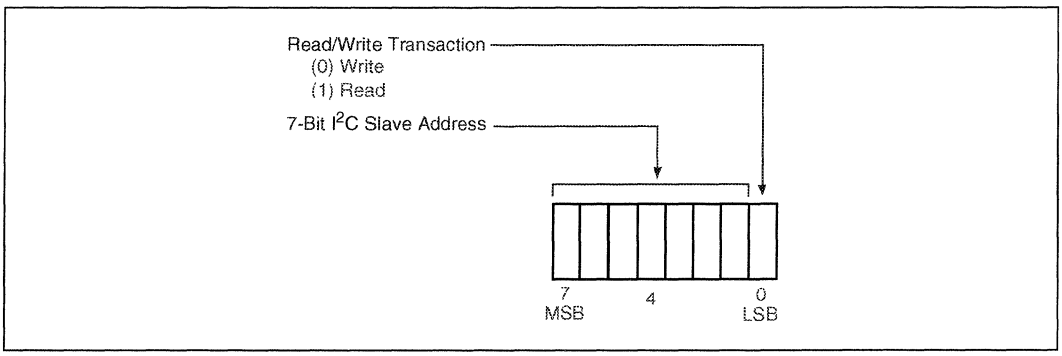
When the I²C unit is in receive mode (master or slave):

1. The processor reads the IDBR data over the internal bus after the IDBR Receive Full interrupt is signalled.
2. The I²C unit transfers data from the shift register to the IDBR after the Ack cycle completes.
3. The I²C unit inserts wait states until the IDBR is read. Refer to Section 22.3.3, "I2C Acknowledge" on page 22-11 for acknowledge pulse information in receiver mode.
4. After the processor reads the IDBR, the I²C unit writes the ICR's Ack/Nack Control bit and the Transfer Byte bit, allowing the next byte transfer to proceed.

22.3.2.1 Addressing a Slave Device

As a master device, the I²C unit must compose and send the first byte of a transaction. This byte consists of the slave address for the intended device and a R/W# bit for transaction definition. The slave address and the R/W# bit are written to the IDBR (see Figure 22-5).

Figure 22-5. Data Format of First Byte in Master Transaction



The first byte transmission must be followed by an Ack pulse from the addressed slave. When the transaction is a write, the I²C unit remains in master-transmit mode and the addressed slave device stays in slave-receive mode. When the transaction is a read, the I²C unit transitions to master-receive mode immediately following the Ack and the addressed slave device transitions to slave-transmit mode. When a Nack is returned, the I²C unit aborts the transaction by automatically sending a STOP and setting the ISR bus error bit.

When the I²C unit is enabled and idle (no bus activity), it stays in slave-receive mode and monitors the I²C bus for a START signal. Upon detecting a START pulse, the I²C unit reads the first seven bits and compares them to those in the I²C Slave Address Register (ISAR) and the general call address (00H). When the bits match those of the ISAR register, the I²C unit reads the eighth bit (R/W# bit) and transmits an Ack pulse. The I²C unit either remains in slave-receive mode (R/W# = 0) or transitions to slave-transmit mode (R/W# = 1). See Section 22.3.7, "General Call Address" on page 22-20 for actions when a general call address is detected.

22.3.3 I²C Acknowledge

Every I²C byte transfer must be accompanied by an acknowledge pulse, which is always generated by the receiver (master or slave). The transmitter must release the **SDA** line for the receiver to transmit the acknowledge pulse (see Figure 22-6).

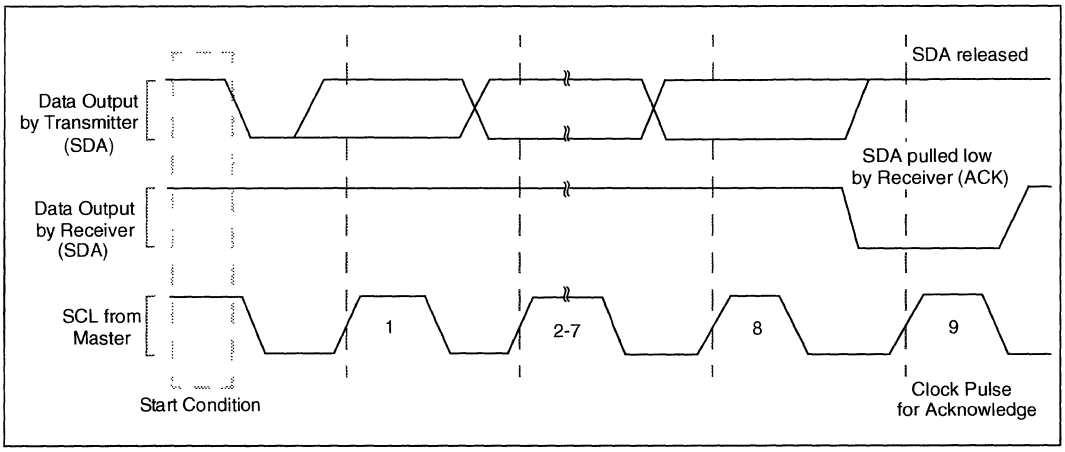
In master-transmit mode, when the target slave receiver device cannot generate the acknowledge pulse, the **SDA** line remains high. This lack of acknowledge (Nack) causes the I²C unit to set the bus error detected bit in the ISR and generate the associated interrupt (when enabled). The I²C unit aborts the transaction by generating a STOP automatically.

In master-receive mode, the I²C unit signals the slave-transmitter to stop sending data by using the negative acknowledge (Nack). The Ack/Nack bit value driven by the I²C bus is controlled by the Ack/Nack bit in the ICR. The bus error detected bit in the ISR is not set for a master-receive mode Nack (as required by the I²C bus protocol). The I²C unit automatically transmits the Ack pulse, based on the Ack/Nack ICR bit, after receiving each byte from the serial bus. Before receiving the last byte, software must set the Ack/Nack Control bit to Nack. Nack is then sent after the next byte is received to indicate the last byte.

In slave mode, the I²C unit automatically acknowledges its own slave address, independent of the Ack/Nack bit setting in the ICR. As a slave-receiver, an Ack response is automatically given to a data byte, independent of the Ack/Nack bit setting in the ICR. The I²C unit sends the Ack value after receiving the eighth data bit of the byte.

In slave-transmit mode, receiving a Nack from the master indicates the last byte is transferred. The master then sends either a STOP or repeated START. The ISR's unit busy bit (2) will remain set until a STOP or repeated START is received.

Figure 22-6. Acknowledge on the I²C Bus



22.3.4 Arbitration

Arbitration on the I²C bus is required due to the multi-master capabilities of the I²C bus. Arbitration is used when two or more masters simultaneously generate a START condition within the minimum I²C hold time of the START condition.

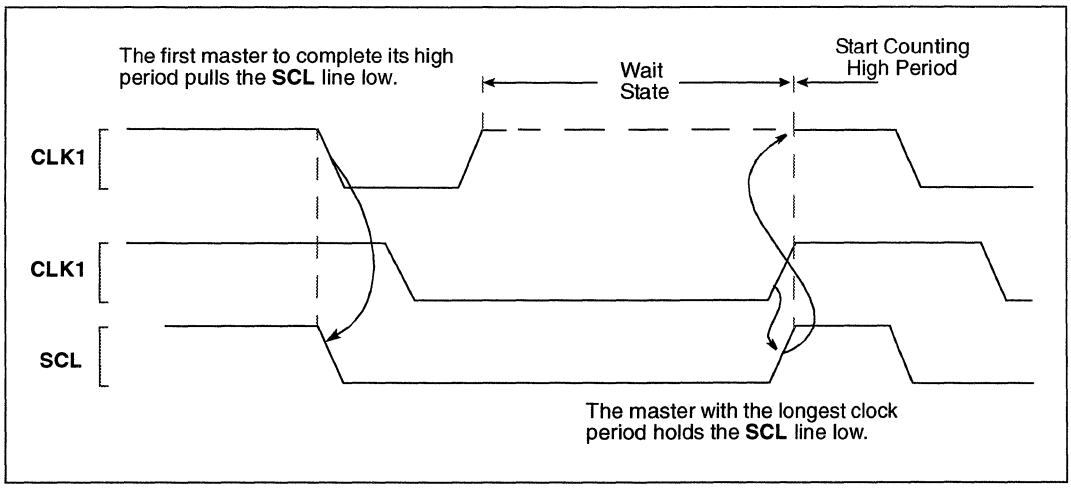
Arbitration can continue for a long period. If the address bit and the R/W# are the same, the arbitration moves to the data. Due to the wired-AND nature of the I²C bus, no data is lost if both (or all) masters are outputting the same bus states. If the address, the R/W# bit, or the data are different, the master which outputted the high state (master's data will be different from **SDA**) will lose arbitration and shut its data drivers off. When losing arbitration, the I²C Bus Interface Unit will shut off the **SDA** or **SCL** drivers for the remainder of the byte transfer, set the Arbitration Loss Detected bit, then return to idle (Slave-Receive) mode.

22.3.4.1 SCL Arbitration

Each master on the I²C bus generates its own clock on the **SCL** line for data transfers. With masters generating their own clocks, clocks with different frequencies may be connected to the **SCL** line. Since data is valid when the clock is in the high period, a defined clock synchronization procedure is needed during bit-by-bit arbitration.

Clock synchronization is accomplished by using the wired-AND connection of the I²C interfaces to the **SCL** line. When a master's clock transitions from high to low, this causes the master to hold down the **SCL** line for its associated period (see Figure 22-7). The low to high transition of the clock may not change when another master has not completed its period. Therefore, the master with the longest low period holds down the **SCL** line. Masters with shorter periods are held in a high wait-state during this time. Once the master with the longest period completes, the **SCL** line transitions to the high state, masters with the shorter periods can continue the data cycle.

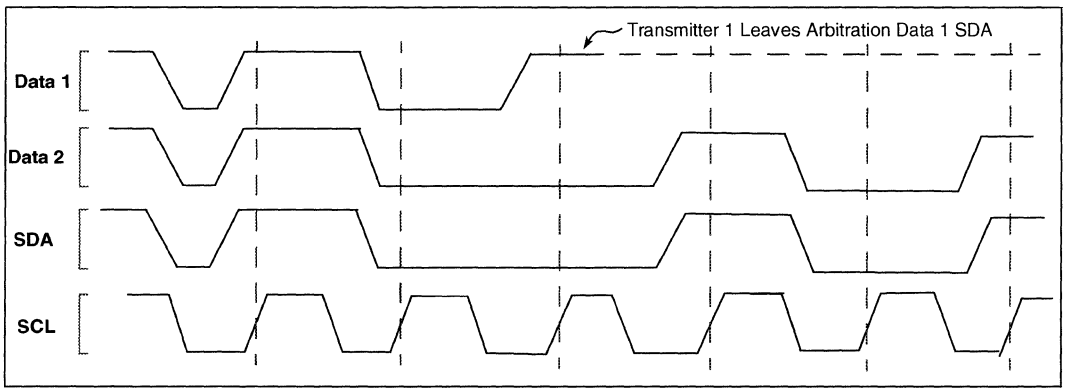
Figure 22-7. Clock Synchronization During the Arbitration Procedure



22.3.4.2 SDA Arbitration

Arbitration on the **SDA** line can continue for a long period, starting with address and R/W# bits and continuing with data bits. Figure 22-8 shows the arbitration procedure for two masters (more than two may be involved depending on how many masters are connected to the bus). When the address and R/W# are the same, arbitration moves to the data. Due to the wired-AND nature of the I^2C bus, no data is lost when both (or all) masters are outputting the same bus states. When address, R/W#, or data is different, the master that output the first low data bit loses arbitration and shuts its data drivers off. When the I^2C unit loses arbitration, it shuts off the **SDA** or **SCL** drivers for remainder of byte transfer, sets arbitration loss detected ISR bit, then returns to idle (Slave-Receive) mode.

Figure 22-8. Arbitration Procedure of Two Masters



When the I^2C unit loses arbitration during transmission of the seven address bits and the 80303 I/O processor is not being addressed as a slave device, the I^2C unit re-sends the address when the I^2C bus becomes free. This is possible because the IDBR and ICR registers are not overwritten when arbitration is lost.

When the arbitration loss is to due to another bus master addressing the 80303 I/O processor as a slave device, the I^2C unit switches to slave-receive mode and the original data in the I^2C data buffer register is overwritten. Software is responsible for clearing the start and re-initiating the master transaction at a later time.

Note: Software must not allow the I^2C unit to write to its own slave address. This can cause the I^2C bus to enter an indeterminate state.

Boundary conditions exist for arbitration when an arbitration process is in progress and a repeated START or STOP condition is transmitted on the I^2C bus. To prevent errors, the I^2C unit, acting as a master, provides for the following sequences:

- No arbitration takes place between a repeated START condition and a data bit
- No arbitration takes place between a data bit and a STOP condition
- No arbitration takes place between a repeated START condition and a STOP condition

These situations arise only when different masters write the same data to the same target slave simultaneously and arbitration is not resolved after the first data byte transfer.

Note: Typically, software is responsible for ensuring arbitration is lost soon after the transaction begins. For example, the protocol might insist that all masters transmit their I^2C address as the first data byte of any transaction ensuring arbitration is ended. A restart is then sent to begin a valid data transfer (the slave can then discard the master's address).

22.3.5 Master Operations

When software initiates a read or write on the I²C bus, the I²C unit transitions from the default slave-receive mode to master-transmit mode. The start pulse is sent followed by the 7-bit slave address and the R/W# bit. After the master receives an acknowledge, the I²C unit has the option of two master modes:

- Master-Transmit — The 80303 I/O processor writes data
- Master-Receive — The 80303 I/O processor reads data

The 80303 I/O processor initiates a master transaction by writing to the ICR register. Data is read and written from the I²C unit through the memory-mapped registers.

Table 22-5 describes the I²C Bus Interface Unit responsibilities as a master device.

Table 22-5. Master Transactions (Sheet 1 of 3)

I²C Master Action	Mode of Operation	Definition
Generate clock output	Master-transmit Master-receive	• The master always drives the **SCL** line. • The ICCR register is written. • The **SCL** Enable bit must be set. • The Unit Enable bit must be set.
Write target slave address to IDBR	Master-transmit Master-receive	• The Intel® i960® core processor writes to IDBR bits 7-1 before a START condition is enabled. • First 7 bits sent on bus after START. • See Section 22.2.3.
Write R/W# Bit to IDBR	Master-transmit Master-receive	• The i960 core processor writes to the least significant IDBR bit with the target slave address. • If low, the master remains a master-transmitter. If high, the master transitions to a master-receiver. • See Section 22.3.2.
Signal START Condition	Master-transmit Master-receive	• See "Generate clock output" above. • Performed after the target slave address and the R/W# bit are in the IDBR. • i960 core processor sets the START bit. • i960 core processor sets the Transfer Byte bit which initiates the start condition. • See Section 22.2.3.
Initiate first data byte transfer	Master-transmit Master-receive	• i960 core processorwrites byte to IDBR • I²C Bus Interface Unit transmits the byte when the Transfer Byte bit is set. • I²C Bus Interface Unit clears the Transfer Byte bit and sets the IDBR Transmit Empty bit when the transfer is complete.

Table 22-5. Master Transactions (Sheet 2 of 3)

I²C Master Action	Mode of Operation	Definition
Arbitrate for I²C Bus	Master-transmit Master-receive	• If two or more masters signal a start within the same clock period, arbitration must occur. • The I²C Bus Interface Unit will arbitrate for as long as necessary. Arbitration takes place during slave address, R/W# bit, and data transmission and continues until all but one master loses the bus. No data is lost during arbitration. • If the I²C Bus Interface Unit loses arbitration, it will set the Arbitration Loss Detect ISR bit after byte transfer is complete and transition to slave-receive (default) mode. • If I²C Bus Interface Unit loses arbitration while attempting to send the target address byte, the I²C Bus Interface Unit will attempt to resend it when the bus becomes free. • The system designer must ensure the boundary conditions described in Section 22.3 do not occur.
Write one data byte to the IDBR	Master-transmit only	• Data transmit mode of I²C master operation. • Occurs when the IDBR Transmit Empty ISR bit is set and the Transfer Byte bit is clear. If enabled, the IDBR Transmit Empty Interrupt is signalled to the i960 core processor. • i960 core processor will write 1 data byte to the IDBR, set the appropriate START/STOP bit combination, and then set the Transfer Byte bit to send the data. Eight bits are written on the serial bus followed by a STOP if requested.
Wait for Acknowledge from slave-receiver	Master-transmit only	• As a master-transmitter, the I²C Bus Interface Unit will generate the clock for the acknowledge pulse. The I²C Bus Interface Unit is responsible for releasing the **SDA** line to allow slave-receiver Ack transmission. • See Section 22.3.3.
Read one byte of I²C Data from the IDBR	Master-receive only	• Data receive mode of I²C master operation. • Eight bits are read from the serial bus, collected in the shift register then transferred to the IDBR after the Ack/Nack bit is read. • The i960 core processor reads the IDBR when the IDBR Receive Full bit is set and the Transfer Byte bit is clear. If enabled, a IDBR Receive Full Interrupt is signalled to the i960 core processor. • When the IDBR is read, if the Ack/Nack Status is clear (indicating Ack), the i960 core processor will write the Ack/Nack Control bit and set the Transfer Byte bit to initiate the next byte read. • If the Ack/Nack Status bit is set (indicating Nack), Transfer Byte bit is clear, STOP bit in the ICR is set, and Unit Busy bit in the ISR is set, then the last data byte has been read into the IDBR and the I²C Bus Interface Unit is sending the STOP. • If the Ack/Nack Status bit is set (indicating Nack), Transfer Byte bit is clear, but the STOP bit is clear, then the i960 core processor has two options: 1. set the START bit, write a new target address to the IDBR, and set the Transfer Byte bit which will send a repeated start condition, 2. set the Master Abort bit and leave the Transfer Byte clear which will send a STOP only.
Transmit Acknowledge to slave-transmitter	Master-receive only	• As a master-receiver, the I²C Bus Interface Unit will generate the clock for the acknowledge pulse. The I²C Bus Interface Unit is also responsible for driving the **SDA** line during the Ack cycle. • If the next data byte is to be the last transaction, the i960 core processor will set the Ack/Nack Control bit for Nack generation. • See Section 22.3.3.

Table 22-5. Master Transactions (Sheet 3 of 3)

I²C Master Action	Mode of Operation	Definition
Generate a Repeated START to chain I²C transactions	Master-transmit Master-receive	• If data chaining is desired, a repeated START condition is used instead of a STOP condition. • This occurs after the last data byte of a transaction has been written to the bus. • The i960 core processor will write the next target slave address and the R/W# bit to the IDBR, set the START bit, and set the Transfer Byte bit. • See Section 22.2.3.
Generate a STOP	Master-transmit Master-receive	• Generated after the i960 core processor writes the last data byte on the bus. • i960 core processor generates a STOP condition by setting the STOP bit in the ICR. • See Section 22.2.3.

When the 80303 I/O processor needs to read data, the I^2C unit transitions from slave-receive mode to master-transmit mode to transmit the start address and immediately following the ACK pulse transitions to master-receive mode to wait for the reception of the read data from the slave device (see Figure 22-9). It is also possible to have multiple transactions during an I^2C operation such as transitioning from master-receive to master-transmit through a repeated start or Data Chaining (see Figure 22-10). Figure 22-11 shows the wave forms of **SDA** and **SCL** for a complete data transfer.

Figure 22-9. Master-Receiver Read from Slave-Transmitter

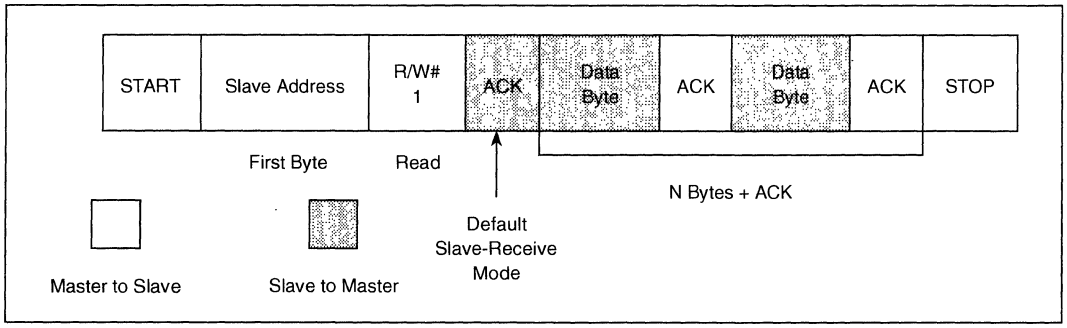


Figure 22-10. Master-Receiver Read from Slave-Transmitter / Repeated Start / Master-Transmitter Write to Slave-Receiver

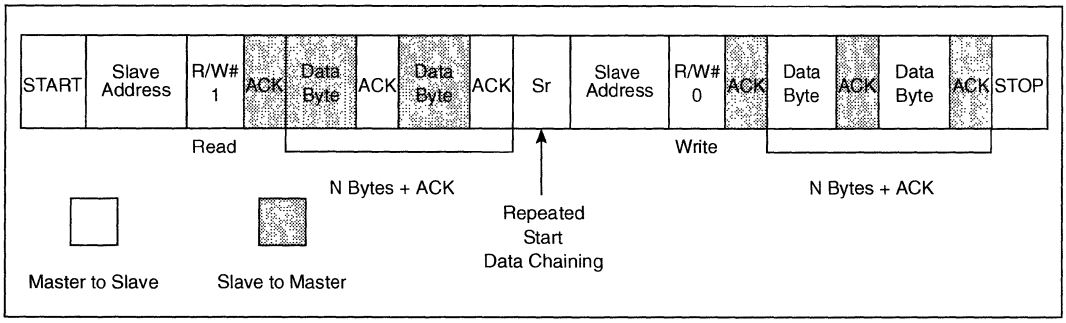
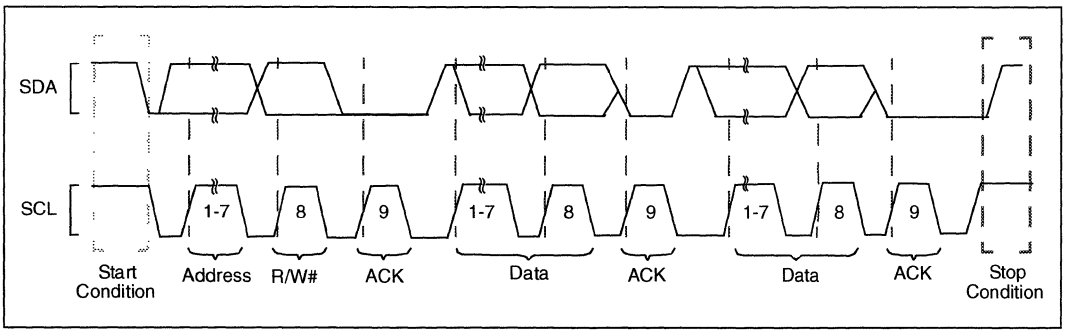


Figure 22-11. A Complete Data Transfer



22.3.6 Slave Operations

Table 22-6 describes the I²C Bus Interface Unit's responsibilities as a slave device.

Table 22-6. Slave Transactions

I²C Slave Action	Mode of Operation	Definition
Slave-receive (default mode)	Slave-receive only	• I²C Bus Interface Unit monitors all slave address transactions. • The I²C Bus Interface Unit Enable bit must be set. • I²C Bus Interface Unit monitors bus for START conditions. When a START is detected, the interface reads the first 8 bits and compares the most significant 7 bits with the 7 bit I²C Slave Address Register and the General Call address (00H). If there is a match, the I²C Bus Interface Unit sends an Ack. • If the first 8 bits are all zero's, this is a general call address. If the General Call Disable bit is clear, both the General Call Address Detected bit and the Slave Mode Operation bit in the ISR will be set. See Section 22.3.7. • If the 8th bit of the first byte (R/W# bit) is low, the I²C Bus Interface Unit stays in slave-receive mode and the Slave Mode Operation bit is cleared. If the R/W# bit is high, the I²C Bus Interface Unit transitions to slave-transmit mode and the Slave Mode Operation bit is set.
Setting the Slave Address Detected bit	Slave-receive Slave-transmit	• Indicates the interface has detected an I²C operation that addresses the Intel® 80303 I/O processor (this includes general call address). The Intel® i960® core processor can distinguish an ISAR match from a General Call by reading the General Call Address Detected bit. • An interrupt is signalled (if enabled) after the matching slave address is received and acknowledged.
Read one byte of I²C Data from the IDBR	Slave-receive only	• Data receive mode of I²C slave operation. • Eight bits are read from the serial bus into the shift register. When a full byte has been received and the Ack/Nack bit has completed, the byte is transferred from the shift register to the IDBR. • Occurs when the IDBR Receive Full bit in the ISR is set and the Transfer Byte bit is clear. If enabled, the IDBR Receive Full Interrupt is signalled to the i960 core processor. • i960 core processor will read 1 data byte from the IDBR. When the IDBR is read, the i960 core processor will write the desired Ack/Nack Control bit and set the Transfer Byte bit. This causes the I²C Bus Interface Unit to stop inserting wait states and let the master transmitter write the next piece of information.
Transmit Acknowledge to master-transmitter	Slave-receive only	• As a slave-receiver, the I²C Bus Interface Unit is responsible for pulling the **SDA** line low to generate the Ack pulse during the high **SCL** period. • The Ack/Nack Control bit controls the Ack data the I²C Bus Interface Unit drives. See Section 22.3.3.
Write one byte of I²C data to the IDBR	Slave-transmit only	• Data transmit mode of I²C slave operation. • Occurs when the IDBR Transmit Empty bit is set and the Transfer Byte bit is clear. If enabled, the IDBR Transmit Empty Interrupt is signalled to the i960 core processor. • i960 core processor will write a data byte to the IDBR and set the Transfer Byte bit to initiate the transfer.
Wait for Acknowledge from master-receiver	Slave-transmit only	• As a slave-transmitter, the I²C Bus Interface Unit is responsible for releasing the **SDA** line to allow the master-receiver to pull the line low for the Ack. • See Section 22.3.3.

Figure 22-12 through Figure 22-14 are examples of I^2C transactions. These show the relationships between master and slave devices.

Figure 22-12. Master-Transmitter Write to Slave-Receiver

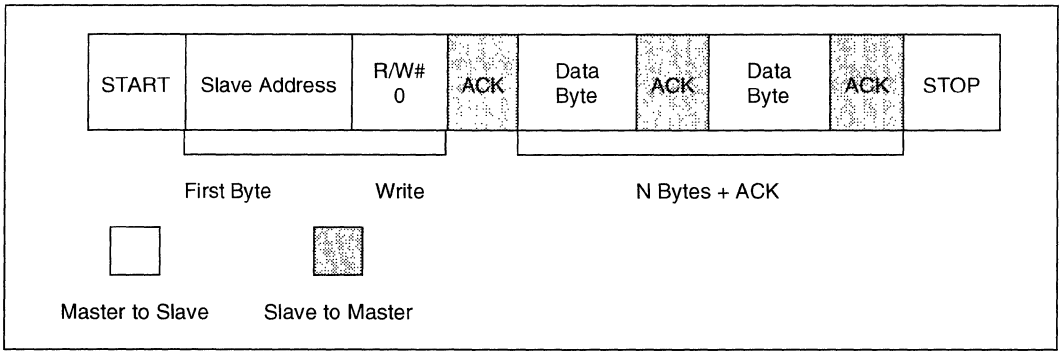


Figure 22-13. Master-Receiver Read to Slave-Transmitter

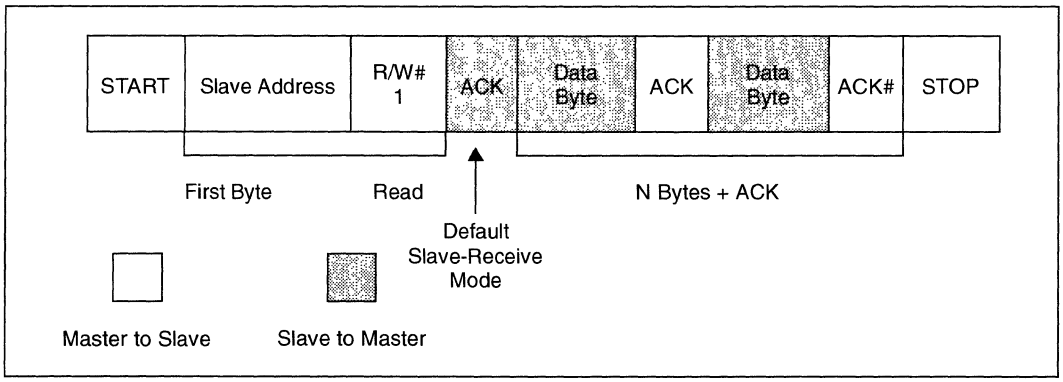
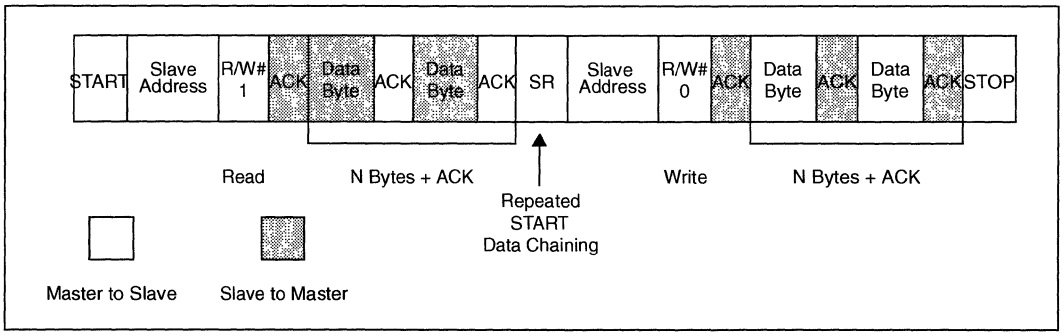


Figure 22-14. Master-Receiver Read to Slave-Transmitter, Repeated START, Master-Transmitter Write to Slave-Receiver



22.3.7 General Call Address

The I²C unit supports both sending and receiving general call address transfers on the I²C bus. When sending a general call message from the I²C unit, software must set the General Call Disable bit in the ICR to keep the I²C unit from responding as a slave. Failure to set this bit causes the I²C Bus to enter an indeterminate state.

A general call address is defined as a transaction with a slave address of 00H. When a device requires the data from a general call address, it acknowledges the transaction and stays in slave-receiver mode. Otherwise, the device can ignore the general call address. The second and following bytes of a general call transaction are acknowledged by every device using it on the bus. Any device not using these bytes must not Ack. The meaning of a general call address is defined in the second byte sent by the master-transmitter. Figure 22-15 shows a general call address transaction. The least significant bit (B) of the second byte defines the transaction. Table 22-7, "General Call Address Second Byte Definitions" on page 22-20 shows the valid values and definitions when B=0.

When the 80303 I/O processor is acting as a slave, and the I²C unit receives a general call address and the ICR General Call Disable bit is clear the I²C unit:

- Sets the ISR general call address detected bit
- Sets the ISR slave address detected bit
- Interrupts (when enabled) the 80303 I/O processor

When the I²C unit receives a general call address and the ICR General Call Disable bit is set, the I²C unit will ignore the general call address.

Figure 22-15. General Call Address

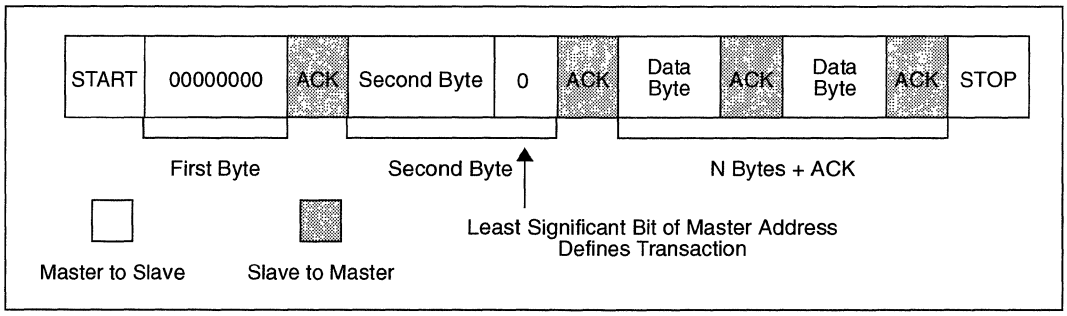


Table 22-7. General Call Address Second Byte Definitions

Least Significant Bit of Second Byte (B)	Second Byte Value	Definition
0	06H	2-byte transaction where the second byte tells the slave to reset and then store this value in the programmable part of their slave address.
0	04H	2-byte transaction where the second byte tells the slave to store this value in the programmable part of their slave address. No reset.
0	00H	Not allowed as a second byte

When directed to reset, the I²C Bus Interface Unit will return to its default reset condition with the exception of the ISAR. The 80303 I/O processor is responsible for ensuring this occurs, not the I²C Bus Interface Unit hardware.

When B=1, the sequence is used as a hardware general call by hardware masters only they cannot transmit a slave address, only their own address. The I²C Bus Interface Unit does not support this mode of operation.

I²C 10-bit addressing and CBUS compatibility are not supported.

22.4 Slave Mode Programming Examples

22.4.1 Initialize Unit

1. Write ICCR: Set clock count
2. Write ISAR: Set slave address
3. Write ICR: Enable all interrupts, set Unit Enable

22.4.2 Write 1 Byte as a Slave

1. Wait for Slave Address Detected interrupt.
   Read ISR: Slave Address Detected (set), Unit Busy (set), R/W# bit (0), Ack/Nack (Clear - Ack)
2. Write IDBR: Load data byte to transfer
3. Write ICR: Set Transfer Byte bit
4. Wait for IDBR Transmit Empty interrupt.
   Read ISR: IDBR Transmit Empty (set), Ack/Nack (set - indicates last byte write), R/W# bit (0)
5. Clear interrupt by clearing the IDBR Transmit Empty Interrupt bit.
6. Wait for interrupt.
   Read ISR: Unit Busy (clear), Slave STOP Detected (set)
7. Clear interrupt by clearing Slave STOP Detected Interrupt bit.

22.4.3 Read 2 Bytes as a Slave

1. Wait for Slave Address Detected interrupt.
   Read ISR: Slave Address Detected (set), Unit busy (set), R/W# bit (0)
2. Read byte 1 on I^2C bus
   Write ICR: Set Transfer Byte bit to initiate the transfer
3. Wait for interrupt.
   Read ISR: IDBR Receive Full (set), Ack/Nack (clear), R/W# bit (0)
   Clear interrupt by clearing IDBR Receive Full bit.
   Read IDBR: To get the data.
4. Read byte 2 on I2C bus
   Write ICR: Set Transfer Byte bit to initiate the transfer
5. Wait for interrupt.
   Read ISR: IDBR Receive Full (set), Ack/Nack (clear), R/W# bit (0)
   Clear interrupt by clearing IDBR Receive Full bit.
   Read IDBR: To get the data.
   Write ICR: Set Transfer Byte bit (to release I^2C bus allowing next transfer)
6. Wait for interrupt.
   Read ISR: Unit busy (clear), Slave STOP Detected (set)
   Clear interrupt by clearing Slave STOP Detected bit.

22.5 Master Programming Examples

22.5.1 Initialize Unit

1. Write ICCR: Set clock count
2. Write ISAR: Set slave address
3. Write ICR: Enable all interrupts (except Arb Loss), set **SCL** Enable, set Unit Enable

22.5.2 Write 1 Byte as a Master

1. Write IDBR: Target slave address and R/W# bit (0 for write)
2. Write ICR: Set START bit, Clear STOP bit, Set Transfer Byte bit to initiate the access
3. Wait for IDBR Transmit Empty interrupt. When interrupt arrives:
   Read status register: IDBR Transmit Empty (set), Unit Busy (set), R/W# bit (clear)
   Clear IDBR Transmit Empty Interrupt bit to clear the interrupt.

Note: The Arbitration Loss Detected bit may be set. Because the Arb Loss interrupt was disabled, if arbitration was lost, an address retry would occur when the bus became free. Clear the Arbitration Loss Detected bit if set.

4. Send byte with STOP
   Write IDBR: With data byte to send
   Write ICR: Clear START bit, Set STOP bit, Enable Arb Loss interrupt, Set Transfer Byte bit to initiate the access
5. Wait for Buffer empty interrupt. When interrupt arrives (Note: Unit will be sending STOP):
   Read status register: IDBR Transmit Empty (set), Unit busy (set - maybe), R/W# bit (clear)
   Clear IDBR Transmit Empty Interrupt bit to clear the interrupt.
   Clear ICR STOP bit (optional)
   Wait until Unit busy is clear before clearing the ICR SCL Enable bit. (optional)

22.5.3 Read 1 Byte as a Master

1. Write IDBR: Target slave address and R/W# bit (1 for read)
2. Write ICR: Set START bit, Clear STOP bit, Disable Arb loss interrupt, Set Transfer Byte bit to initiate the access
3. Wait for IDBR Transmit Empty interrupt. When interrupt arrives:
   Read status register: IDBR Transmit Empty (set), Unit busy (set), R/W# bit (set)
   Clear IDBR Transmit Empty bit to clear the interrupt.
4. Read byte with STOP
   Write ICR: Clear START bit, Set STOP bit, Enable arb loss interrupt, Set Ack/Nack bit (Nack), Set Transfer Byte bit to initiate the access
5. Wait for Buffer full interrupt. When interrupt arrives (Note: Unit will be sending STOP):
   Read status register: IDBR Receive Full (set), Unit Busy (set - maybe), R/W# bit (Set), Ack/Nack bit (Set)
   Clear IDBR Receive Full bit to clear the interrupt.
   Read IDBR data.

Clear ICR STOP bit (optional), Clear ICR Ack/Nack Control bit (optional)
Wait until Unit busy is clear before clearing the ICR SCL Enable bit. (optional)

22.5.4 Write 2 Bytes and Repeated Start Read 1 Byte as a Master

1. Write IDBR: Target slave address and R/W# bit (0 for write)
2. Write ICR: Set START bit, Clear STOP bit, Set Transfer Byte bit to initiate the access
3. Wait for IDBR Transmit Empty interrupt. When interrupt arrives:
   Read status register: IDBR Transmit Empty (set), Unit busy (set), R/W# bit (clear)
   Clear IDBR Transmit Empty bit to clear the interrupt.
4. Send byte 1
   Write IDBR: With data byte to send
   Write ICR: Clear START bit, Clear STOP bit, Enable Arb Loss interrupt, Set Transfer Byte bit to initiate the access
5. Wait for Buffer empty interrupt.
   Read status register: IDBR Transmit Empty (set), Unit busy (set), R/W# bit (clear)
   Clear IDBR Transmit Empty bit to clear the interrupt.
6. Send byte 2
   Write IDBR: With data byte to send
   Write ICR: Clear START bit, Clear STOP bit, Set Transfer Byte bit to initiate the access
7. Wait for Buffer empty interrupt.
   Read status register: IDBR Transmit Empty (set), Unit busy (set), R/W# bit (clear)
   Clear IDBR Transmit Empty bit to clear the interrupt.
8. Send repeated start as a master
   Write IDBR: Target slave address and R/W# bit (1 for read)
   Write ICR: Set START bit, Clear STOP bit, Disable Arb Loss interrupt, Set Transfer Byte bit the initiate the access
9. Wait for IDBR Transmit Empty interrupt. When interrupt comes.
   Read status register: IDBR Transmit Empty (set), Unit busy (set), R/W# bit (set)
   Clear IDBR Transmit Empty bit to clear the interrupt.
10. Read byte with STOP
    Write ICR: Clear START bit, Set STOP bit, Enable arb loss interrupt, Set Ack/Nack bit (Nack), Set Transfer Byte bit to initiate the access
11. Wait for Buffer full interrupt. When interrupt comes (Note: Unit will be sending STOP).
    Read status register: IDBR Receive Full (set), Unit busy (set - maybe), R/W# bit (Set), Ack/Nack bit (Set)
    Clear IDBR Receive Full bit to clear the interrupt.
    Read IDBR data.
    Clear ICR STOP bit (optional), Clear ICR Ack/Nack Control bit (optional)
    Wait until Unit busy is clear before clearing the ICR SCL Enable bit. (optional)

22.5.5 Read 2 Bytes as a Master - Send STOP Using the Abort

1. Write IDBR: Target slave address and R/W# bit (1 for read)
2. Write ICR: Set START bit, Clear STOP bit, Disable Arb loss interrupt, Set Transfer Byte bit to initiate the access

3. Wait for IDBR Transmit Empty interrupt. When interrupt comes.
   Read status register: IDBR Transmit Empty (set), Unit Busy (set), R/W# bit (set)
   Clear IDBR Transmit Empty bit to clear the interrupt.
4. Read byte 1
   Write ICR: Clear START bit, Clear STOP bit, Enable Arb Loss interrupt, Clear Ack/Nack bit (Ack), Set Transfer Byte bit to initiate the access
5. Wait for Buffer full interrupt.
   Read status register: IDBR Receive Full (set), Unit busy (set), R/W# bit (Set), Ack/Nack bit (Clear)
   Clear IDBR Receive Full bit to clear the interrupt.
   Read IDBR data.
6. Read byte 2 with Nack (STOP is not set because STOP or Repeated START will be decided on the byte read)
   Write ICR: Clear START bit, Clear STOP bit, Enable Arb Loss interrupt, Set Ack/Nack bit (Nack), Set Transfer Byte bit to initiate the access
7. Wait for Buffer full interrupt.
   Read status register: IDBR Receive Full (set), Unit Busy (set), R/W# bit (Set), Ack/Nack bit (Set)
   Clear IDBR Receive Full bit to clear the interrupt.
   Read IDBR data.

There are now two options based on the byte read:

- Send a repeated START
- Send a STOP only

Here, a STOP abort is sent.

Note: Had a NACK not been sent, the next transaction *must* involve another data byte read.

8. Send STOP abort condition. (STOP with no data transfer.)
   Write ICR: Set Master abort.

22.6 Glitch Suppression Logic

The I²C Bus Interface Unit has built-in glitch suppression logic. Glitches will be suppressed according to: 4 * internal bus clock period. For example, with a 66 MHz (15 ns period) 80303 I/O processor clock, glitches of 60ns or less will be suppressed. At 40 MHz (25 ns period) clock, glitches of 100 ns or less will be suppressed. This is within the 50 ns glitch suppression specification.

22.7 Reset Conditions

The I²C unit is reset with **I_RST#**. Software is responsible for ensuring the I²C unit is not busy (ISR[3]) before asserting reset. Software is also responsible for ensuring the I²C bus is idle when the unit is enabled after reset. When directed to reset, the I²C unit returns to its default reset condition with the exception of the ISAR. ISAR is not affected by a reset.

When the Unit Reset bit in the ICR is set, only the 80303 I/O processor I²C unit resets, the associated I²C MMRs remain intact. When resetting the I²C unit with the ICR's unit reset, use the following guidelines:

1. In the ICR register, set the reset bit and clear the remainder of the register.
2. Clear the ISR register.
3. Clear reset in the ICR.

22.8 Register Definitions

The following registers are associated with the I²C Bus Interface Unit. They are all located within the peripheral memory- mapped address space of the 80303 I/O processor. See Section 22.8, "Register Definitions" on page 22-28 for the register addresses

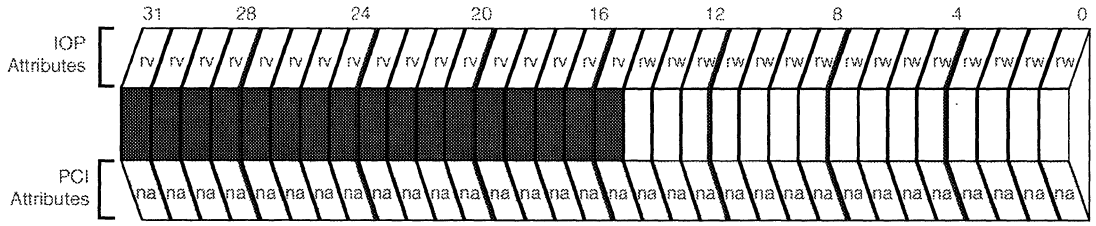
Table 22-8. I²C Register Summary Table

Section, Register Name, Page
Section 22.8.1, "I2C Control Register- ICR" on page 22-29
Section 22.8.2, "I2C Status Register- ISR" on page 22-32
Section 22.8.3, "I2C Slave Address Register- ISAR" on page 22-34
Section 22.8.4, "I2C Data Buffer Register- IDBR" on page 22-35
Section 22.8.5, "I2C Clock Count Register- ICCR" on page 22-36
Section 22.8.6, "I2C Bus Monitor Register- IBMR" on page 22-37

22.8.1 I²C Control Register- ICR

The 80303 I/O processor uses the bits in the I²C Control Register (ICR) to control the I²C unit.

Table 22-9. I²C Control Register - ICR (Sheet 1 of 3)

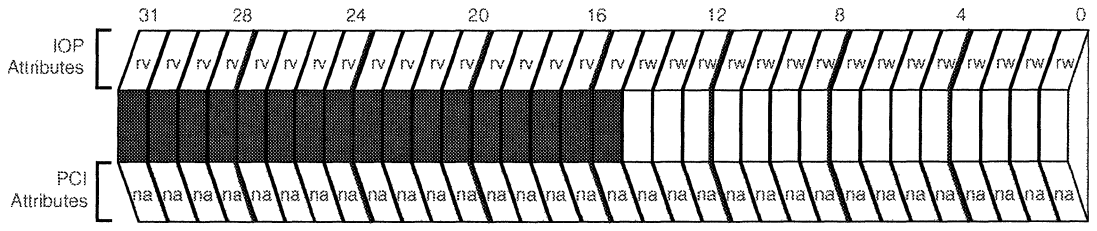


Intel® 80960 Core Local Bus Address
1680H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:14	0000H	Reserved
14	0_2	**Unit Reset:** 0 = No reset. 1 = Reset the I²C unit only.
13	0_2	**Slave Address Detected Interrupt Enable:** 0 = Disable interrupt. 1 = Enables the I²C unit to interrupt the Intel® 80303 I/O processor upon detecting a slave address match or a general call address.
12	0_2	**Arbitration Loss Detected Interrupt Enable:** 0 = Disable interrupt. 1 = Enables the I²C unit to interrupt the 80303 I/O processor upon losing arbitration while in master mode.
11	0_2	**Slave STOP Detected Interrupt Enable:** 0 = Disable interrupt. 1 = Enables the I²C unit to interrupt the 80303 I/O processor when it detects a STOP condition while in slave mode.
10	0_2	**Bus Error Interrupt Enable:** 0 = Disable interrupt. 1 = Enables the I²C unit to interrupt the 80303 I/O processor for the following I²C bus errors: • As a master transmitter, no Ack was detected after a byte was sent. • As a slave receiver, the I²C unit generated a Nack pulse. **NOTE:** Software is responsible for guaranteeing that misplaced START and STOP conditions do not occur. See Section 22.6, "Glitch Suppression Logic" on page 22-26.
09	0_2	**IDBR Receive Full Interrupt Enable:** 0 = Disable interrupt. 1 = Enables the I²C unit to interrupt the 80303 I/O processor when the IDBR has received a data byte from the I²C bus.
08	0_2	**IDBR Transmit Empty Interrupt Enable:** 0 = Disable interrupt. 1 = Enables the I²C unit to interrupt the 80303 I/O processor after transmitting a byte onto the I²C bus.

Table 22-9. I²C Control Register - ICR (Sheet 2 of 3)

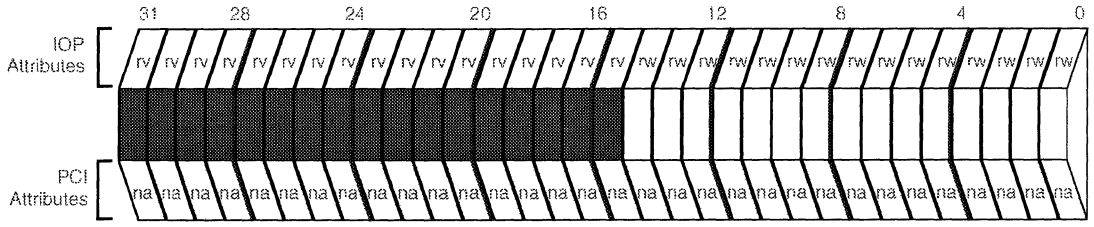


Intel® 80960 Core Local Bus Address
1680H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
07	0_2	**General Call Disable**: 0 = Enables the I²C unit to respond to general call messages. 1 = Disables I²C unit response to general call messages as a slave. This bit must be set when sending a master mode general call message from the I²C unit.
06	0_2	**I²C Unit Enable**: 0 = Disables the unit and does not master any transactions or respond to any slave transactions. 1 = Enables the I²C unit (defaults to slave-receive mode). Software must guarantee the I²C bus is idle before setting this bit.
05	0_2	**SCL Enable**: 0 = Disables the I²C unit from driving the **SCL** line. 1 = Enables the I²C clock output for master mode operation. The ICCR (see Section 22.8.5, "I2C Clock Count Register- ICCR" on page 22-36) must be programmed with a valid value before setting this bit.
04	0_2	**Master Abort**: used by the I²C unit when in master mode to generate a STOP without transmitting another data byte. 0 = The I²C unit transmits STOP using the STOP ICR bit only. 1 = The I²C unit sends STOP without data transmission. When in Master transmit mode, after transmitting a data byte, the ICR's Transfer Byte bit is clear and IDBR Transmit Empty bit is set. When no more data bytes need to be sent, setting master abort bit sends the STOP. The Transfer Byte bit (03) must remain clear. In master-receive mode, when a Nack is sent without a STOP (STOP ICR bit was not set) and the 80303 I/O processor does not send a repeated START, setting this bit sends the STOP. Once again, the Transfer Byte bit (03) must remain clear.
03	0_2	**Transfer Byte**: used to send/receive a byte on the I²C bus. 0 = Cleared by I²C unit when the byte is sent/received. 1 = Send/receive a byte. The 80303 I/O processor can monitor this bit to determine when the byte transfer has completed. In master or slave mode, after each byte transfer including Ack/Nack bit, the I²C unit holds the **SCL** line low (inserting wait states) until the Transfer Byte bit is set.
02	0_2	**Ack/Nack Control**: defines the type of Ack pulse sent by the I²C unit when in master receive mode. 0 = The I²C unit sends an Ack pulse after receiving a data byte. 1 = The I²C unit sends a negative Ack (Nack) after receiving a data byte. The I²C unit automatically sends an Ack pulse when responding to its slave address or when responding in slave-receive mode, independent of the Ack/Nack control bit setting.

Table 22-9. I²C Control Register - ICR (Sheet 3 of 3)



Intel® 80960 Core Local Bus Address
1680H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
01	0_2	**STOP**: used to initiate a STOP condition after transferring the next data byte on the I²C bus when in master mode. In master-receive mode, the Ack/Nack control bit must be set in conjunction with this bit. See Section 22.2.3.3, "STOP Condition" on page 22-7 for more details on the STOP state. 0 = Do not send a STOP. 1 = Send a STOP.
00	0_2	**START**: used to initiate a START condition to the I²C unit when in master mode. See Section 22.2.3.1, "START Condition" on page 22-6 for more details on the START state. 0 = Do not send a START. 1 = Send a START.

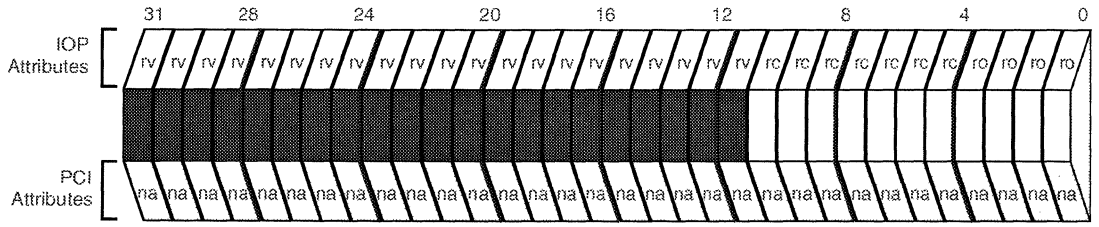
22.8.2 I²C Status Register- ISR

I²C interrupts are signalled through **XINT7#** and the XINT7 Interrupt Status Register (X7ISR), which shows the pending XINT7 interrupts (see Chapter 8, "PCI and Peripheral Interrupt Controller Unit"). **XINT7#** is set by the I²C Interrupt Status Register (ISR). Software uses the ISR bits to check the status of the I²C unit and bus. ISR bits (bits 9-5) are updated after the Ack/Nack bit has completed on the I²C bus.

The ISR is also used to clear interrupts signalled from the I²C Bus Interface Unit. These are:

- IDBR Receive Full
- IDBR Transmit Empty
- Slave Address Detected
- Bus Error Detected
- STOP Condition Detect
- Arbitration Lost

Table 22-10. I²C Status Register - ISR (Sheet 1 of 2)

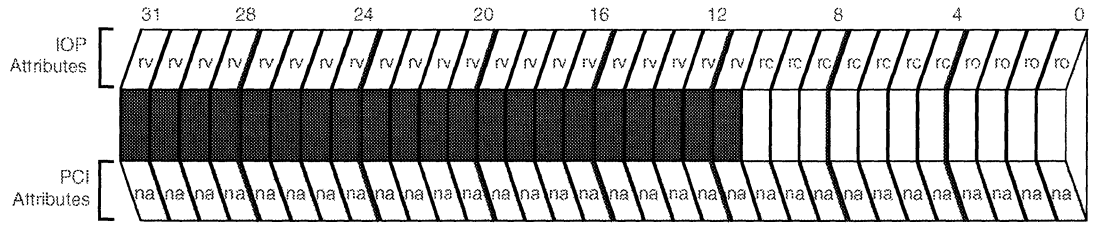


Intel® 80960 Core Local Bus Address
1684H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:11	000000H	Reserved
10	0_2	**Bus Error Detected:** 0 = No error detected. 1 = The I²C unit sets this bit when it detects one of the following error conditions: • As a master transmitter, no Ack was detected on the interface after a byte was sent. • As a slave receiver, the I²C unit generates a Nack pulse. **NOTE:** When an error occurs, I²C bus transactions continue. Software must guarantee that misplaced START and STOP conditions do not occur. See Section 22.3.4, "Arbitration" on page 22-12.
09	0_2	**Slave Address Detected:** 0 = No slave address detected. 1 = I²C unit detected a 7-bit address that matches the general call address or ISAR. An interrupt is signalled when enabled in the ICR.
08	0_2	**General Call Address Detected:** 0 = No general call address received. 1 = I²C unit received a general call address.

Table 22-10. I²C Status Register - ISR (Sheet 2 of 2)



Intel® 80960 Core Local Bus Address
1684H

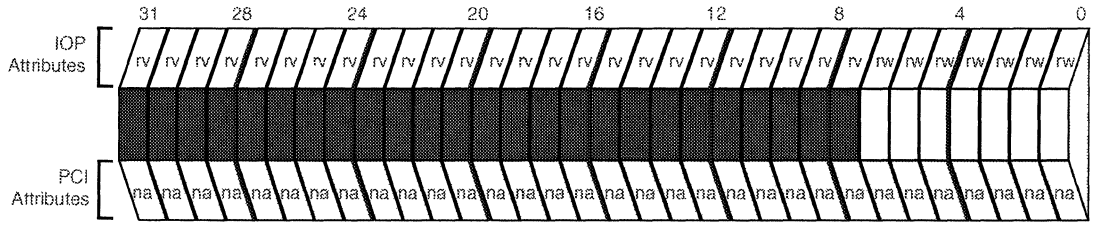
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
07	0_2	**IDBR Receive Full**: 0 = The IDBR has not received a new data byte or the I²C unit is idle. 1 = The IDBR register received a new data byte from the I²C bus. An interrupt is signalled when enabled in the ICR.
06	0_2	**IDBR Transmit Empty**: 0 = The data byte is still being transmitted. 1 = The I²C unit has finished transmitting a data byte on the I²C bus. An interrupt is signalled when enabled in the ICR.
05	0_2	**Arbitration Loss Detected**: used during multi-master operation. 0 = Cleared when arbitration is won or never took place. 1 = Set when the I²C unit loses arbitration.
04	0_2	**Slave STOP Detected**: 0 = No STOP detected. 1 = Set when the I²C unit detects a STOP while in slave-receive or slave-transmit mode.
03	0_2	**I²C Bus Busy**: 0 = I²C bus is idle or the I²C unit is using the bus (i.e., unit busy). 1 = Set when the I²C bus is busy but the Intel® 80303 I/O processor's I²C unit is not involved in the transaction.
02	0_2	**Unit Busy**: 0 = I²C unit not busy. 1 = Set when the 80303 I/O processor's I²C unit is busy. This is defined as the time between the first START and STOP.
01	0_2	**Ack/Nack Status**: 0 = The I²C unit received or sent an Ack on the bus. 1 = The I²C unit received or sent a Nack. This bit is used in slave transmit mode to determine when the byte transferred is the last one. This bit is updated after each byte and Ack/Nack information is received.
00	0_2	**Read/Write Mode**: 0 = The I²C unit is in master-transmit or slave-receive mode. 1 = The I²C unit is in master-receive or slave-transmit mode. This is the R/W# bit of the slave address. It is automatically cleared by hardware after a stop state.

22.8.3 I²C Slave Address Register- ISAR

The I²C Slave Address Register (see Table 22-11) defines the I²C unit's 7-bit slave address to which the 80303 I/O processor responds when in slave-receive mode. This register is written by the 80303 I/O processor before enabling I²C operations. The register is fully programmable (no address is assigned to the I²C unit) so it can be set to a value other than those of hard-wired I²C slave peripherals that might exist in the system. The ISAR is not affected by the 80303 I/O processor being reset. The ISAR register default value is 0000000_2.

Table 22-11. I²C Slave Address Register - ISAR



Intel® 80960 Core Local Bus Address
1688H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:07	000000H	Reserved
06:00	00H	**I²C Slave Address**: The 7-bit address to which the I²C unit responds when in slave-receive mode.

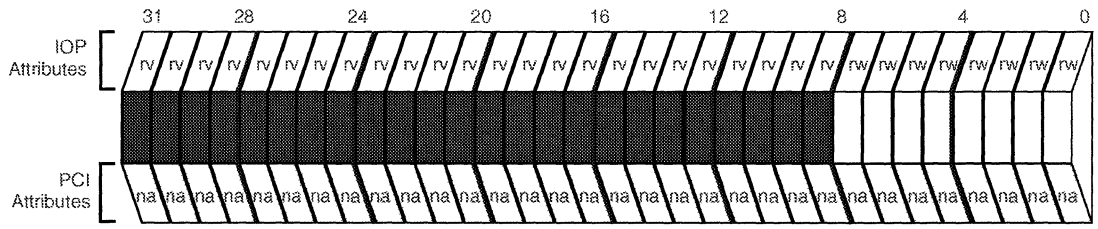
22.8.4 I²C Data Buffer Register- IDBR

The I²C Data Buffer Register is used by the 80303 I/O processor to transmit and receive data from the I²C bus. The accesses the IDBR by the 80303 I/O processor on one side and by the I²C shift register on the other. Data coming into the I²C Bus Interface Unit is received into the IDBR after a full byte has been received and acknowledged. Data going out of the I²C Bus Interface Unit is written to the IDBR by the 80303 I/O processor core and sent to the serial bus.

When the I²C Bus Interface Unit is in transmit mode (master or slave), the 80303 I/O processor writes data to the IDBR over the internal bus. This occurs when a master transaction is initiated or when the IDBR Transmit Empty Interrupt is signalled. Data is moved from the IDBR to the shift register when the Transfer Byte bit is set. The IDBR Transmit Empty Interrupt will be signalled (if enabled) when a byte has been transferred on the I²C bus and the acknowledge cycle is complete. If the IDBR is not written by the 80303 I/O processor (and a STOP condition was not in place) before the I²C bus is ready to transfer the next byte packet, the I²C Bus Interface Unit will insert wait states until the i960 core processor writes the IDBR and sets the Transfer Byte bit.

When the I²C Bus Interface Unit is in receive mode (master or slave), the processor will read IDBR data over the internal bus. This occurs when the IDBR Receive Full Interrupt is signalled. The data is moved from the shift register to the IDBR when the Ack cycle is complete. The I²C Bus Interface Unit will insert wait states until the IDBR has been read. Refer to Section 22.3.3, "I2C Acknowledge" on page 22-11 for acknowledge pulse information in receiver mode. After the 80303 I/O processor reads the IDBR, the Ack/Nack Control bit is written and the Transfer Byte bit is written, allowing the next byte transfer to proceed on the I²C Bus. The IDBR register is 00H after reset.

Table 22-12. I²C Data Buffer Register - IDBR



Intel® 80960 Core Local Bus Address
168CH

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

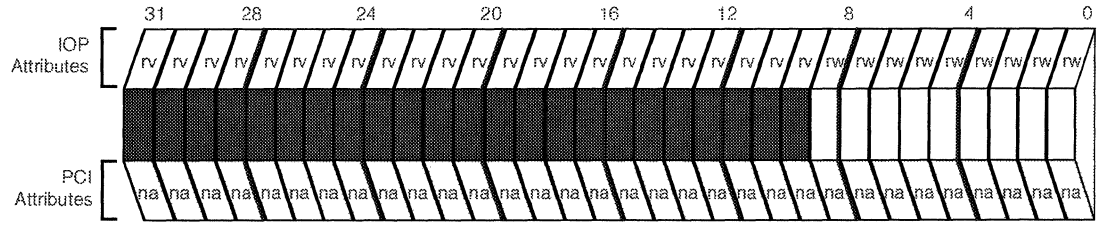
Bit	Default	Description
31:08	000000H	Reserved
07:00	00H	**I²C Data Buffer**: Buffer for I²C bus send/receive data.

22.8.5 I²C Clock Count Register- ICCR

The I²C Clock Count Register (ICCR) defines the multiplier used to generate the I²C **SCL** clock. This register is used with an internal 9-bit counter. When the **SCL** enable bit in the ICR is set, this counter decrements from the programmed ICCR value to zero, then resets to the programmed ICCR value and decrements again. This continues until the **SCL** enable bit in the ICR is cleared. Each time the counter reaches zero, the **SCL** line transitions from low to high or vice versa, depending on the current state. This creates the I²C clock output during I²C master operations.

Changing this register while the **SCL** enable bit is set results in undefined behavior.

Table 22-13. I²C Clock Count Register - ICCR



Intel® 80960 Core Local Bus Address
1690H

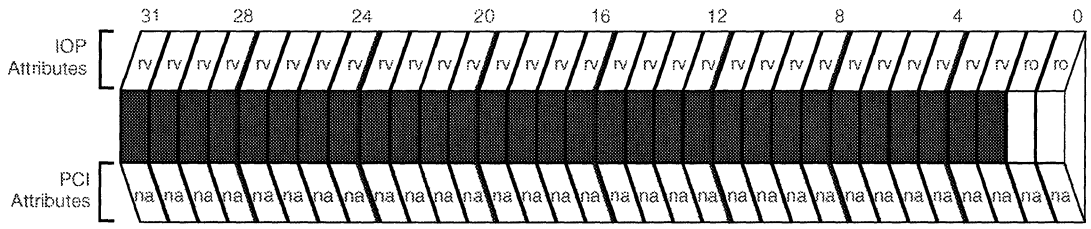
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:10	000000H	Reserved
09:00	0	**I²C Clock Count**: 9 bit count value used to generate an I²C clock from the Intel® 80303 I/O processor internal bus clock.

22.8.6 I²C Bus Monitor Register- IBMR

The I²C Bus Monitor Register (IBMR) tracks the status of the **SCL** and **SDA** pins. The values of these pins are recorded in this read-only register so that software may determine if the I²C bus is hung and the I²C unit must be reset.

Table 22-14. I²C Bus Monitor Register - IBMR



Intel® 80960 Core Local Bus Address
1694H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
31:02	0	Reserved
01	1	**SCL** Status: This bit continuously reflects the value of the **SCL** pin.
00	1	**SDA** Status: This bit continuously reflects the value of the **SDA** pin.

General Purpose Input Output (GPIO) 23

This chapter describes the Intel® 80303 I/O processor General Purpose Input Output (GPIO) Unit.

23.1 General Purpose Input Output Support

Eight pins are provided as General Purpose Input Output (GPIO) pins. The eight pins are **GPIO[7:0]**. These pins can be used by the Intel® i960® JF processor to control or monitor external devices in the I/O subsystem.

23.1.1 General Purpose Inputs

The current state of the eight GPIO pins can be read in Section 23.2.2, "GPIO Input Data Register - GPID" on page 23-4).

23.1.2 General Purpose Outputs

The output function of the GPIO pins is controlled by two registers, as stated in Section 23.2.3, "GPIO Output Data Register - GPOD" on page 23-5) and Section 23.2.1, "GPIO Output Enable Register - GPOE" on page 23-2).

The output enables are mapped on a per bit basis to each of the data bits in the GPIO Output Data Register. When a bit of the GPIO Output Enable Register is cleared, the corresponding data bit value in the GPIO Output Data Register will be actively driven on the appropriate GPIO pin.

23.1.3 Reset Initialization of General Purpose Input Output Function

Both the GPIO Output Data Register and the GPIO Input Data Register will be initialized to 00H upon assertion of P_RST#.

The GPIO Output Enable Register is initialized to the value of the eight associated GPIO pins upon assertion of P_RST#. The I/O pin design will provide internal weak pull-up devices that are driven on the GPIO pin during P_RST# assertion.

In order to enable a particular GPIO pin to operate as an output following the deassertion of P_RST#, the user will need to provide a weak pull-down on the GPIO pin, to overdrive the internal weak pull-up device.

23.2 Register Definitions

All ten registers are visible as 80303 I/O processor memory mapped registers and can be accessed through the internal memory bus. Each is a 8-bit register and is memory-mapped in the 80303 processor memory space. The memory-mapped addresses of the GPIO control registers are found in Appendix C, "Peripheral Memory-Mapped Registers."

There are four control and status registers for the PCI And Peripheral Interrupt Controller:

- GPIO Output Enable Register
- XINT6 Interrupt Status Register
- XINT7 Interrupt Status Register
- NMI Interrupt Status Register

The PCI Interrupt Routing Select Register is accessible from the internal memory bus and also during PCI configuration cycles through the PCI configuration register space (function #0). See Chapter 15, "PCI Address Translation Unit" for additional information regarding the PCI configuration cycles that access the PCI Interrupt Routing Select Register. The programmer's interface to the interrupt controller is through ten memory-mapped control registers. Table C-19, "Peripheral Memory-Mapped Register Locations" on page C-6 describes these registers.

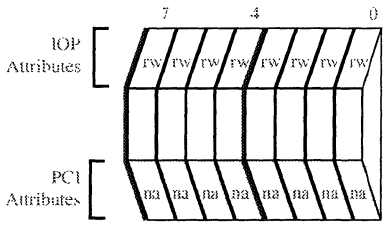
23.2.1 GPIO Output Enable Register - GPOE

The GPIO Output Enable Register enables on a per pin basis the output value contained in the GPIO Output Data Register onto the appropriate pin.

The GPIO Output Enable Register is initialized to the value of the eight associated GPIO pins upon assertion of P_RST#. The I/O pin design will provide internal weak pull-up devices that are driven on the GPIO pin during P_RST# assertion.

In order to enable a particular GPIO pin to operate as an output following the deassertion of P_RST#, the user will need to provide a weak pull-down on the GPIO pin, to overdrive the internal weak pull-up device.

Table 23-1. GPIO Output Enable Register - GPOE



Intel® 80960 Core Local Bus Address
0000 171CH

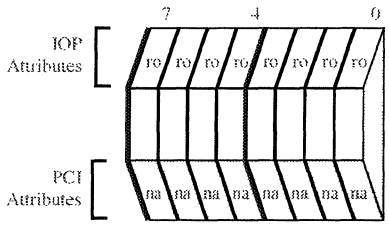
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
07	**GPIO[7]** during P_RST# assertion	GPIO7 Output Enable -- When clear, bit 7 of the GPIO Output Data Register will be enabled onto the **GPIO[7]** pin.
06	**GPIO[6]** during P_RST# assertion	GPIO6 Output Enable -- When clear, bit 6 of the GPIO Output Data Register will be enabled onto the **GPIO[6]** pin.
05	**GPIO[5]** during P_RST# assertion	GPIO5 Output Enable -- When clear, bit 5 of the GPIO Output Data Register will be enabled onto the **GPIO[5]** pin.
04	**GPIO[4]** during P_RST# assertion	GPIO4 Output Enable -- When clear, bit 4 of the GPIO Output Data Register will be enabled onto the **GPIO[4]** pin.
03	**GPIO[3]** during P_RST# assertion	GPIO3 Output Enable -- When clear, bit 3 of the GPIO Output Data Register will be enabled onto the **GPIO[3]** pin.
02	**GPIO[2]** during P_RST# assertion	GPIO2 Output Enable -- When clear, bit 2 of the GPIO Output Data Register will be enabled onto the **GPIO[2]** pin.
01	**GPIO[1]** during P_RST# assertion	GPIO1 Output Enable -- When clear, bit 1 of the GPIO Output Data Register will be enabled onto the **GPIO[1]** pin.
00	**GPIO[0]** during P_RST# assertion	GPIO0 Output Enable -- When clear, bit 0 of the GPIO Output Data Register will be enabled onto the **GPIO[0]** pin.

23.2.2 GPIO Input Data Register - GPID

The GPIO Input Data Register will reflect the state of the appropriate **IRQ** bus pin following the deassertion of P_RST#.

Table 23-2. GPIO Input Data Register - GPID



Intel® 80960 Core Local Bus Address
0000 1720H

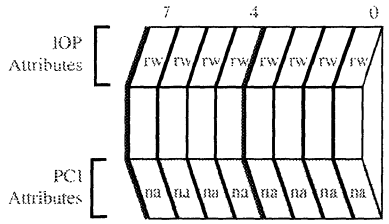
Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
07	0_2	GPIO7 Input Data -- This bit will reflect the state of the **GPIO[7]** pin following the deassertion of P_RST#.
06	0_2	GPIO6 Input Data -- This bit will reflect the state of the **GPIO[6]** pin following the deassertion of P_RST#.
05	0_2	GPIO5 Input Data -- This bit will reflect the state of the **GPIO[5]** pin following the deassertion of P_RST#.
04	0_2	GPIO4 Input Data -- This bit will reflect the state of the **GPIO[4]** pin following the deassertion of P_RST#.
03	0_2	GPIO3 Input Data -- This bit will reflect the state of the **GPIO[3]** pin following the deassertion of P_RST#.
02	0_2	GPIO2 Input Data -- This bit will reflect the state of the **GPIO[2]** pin following the deassertion of P_RST#.
01	0_2	GPIO1 Input Data -- This bit will reflect the state of the **GPIO[1]** pin following the deassertion of P_RST#.
00	0_2	GPIO0 Input Data -- This bit will reflect the state of the **GPIO[0]** pin following the deassertion of P_RST#.

23.2.3 GPIO Output Data Register - GPOD

The GPIO Output Data Register will be driven on a per bit basis on the appropriate **IRQ** bus pin following the deassertion of P_RST# if the corresponding bit in the GPOE register is cleared.

Table 23-3. Output Data Register - GPOD



Intel® 80960 Core Local Bus Address
0000 1724H

Attribute Legend:
RV = Reserved
PR = Preserved
RS = Read/Set
RW = Read/Write
RC = Read Clear
RO = Read Only
NA = Not Accessible

Bit	Default	Description
07	0_2	GPIO7 Output Data -- This bit value will be driven on the **GPIO[7]** pin if bit 7 of the GPOE register is cleared.
06	0_2	GPIO6 Output Data -- This bit value will be driven on the **GPIO[6]** pin if bit 6 of the GPOE register is cleared.
05	0_2	GPIO5 Output Data -- This bit value will be driven on the **GPIO[5]** pin if bit 5 of the GPOE register is cleared.
04	0_2	GPIO4 Output Data -- This bit value will be driven on the **GPIO[4]** pin if bit 4 of the GPOE register is cleared.
03	0_2	GPIO3 Output Data -- This bit value will be driven on the **GPIO[3]** pin if bit 3 of the GPOE register is cleared.
02	0_2	GPIO2 Output Data -- This bit value will be driven on the **GPIO[2]** pin if bit 2 of the GPOE register is cleared.
01	0_2	GPIO1 Output Data -- This bit value will be driven on the **GPIO[1]** pin if bit 1 of the GPOE register is cleared.
00	0_2	GPIO0 Output Data -- This bit value will be driven on the **GPIO[0]** pin if bit 0 of the GPOE register is cleared.

Test Features 24

This chapter describes the Intel® 80303 I/O processor test features, including ONCE (On-Circuit Emulation) and boundary-scan (JTAG). Together these two features create a powerful environment for design debug and fault diagnosis.

24.1 On-Circuit Emulation (ONCE)

On-circuit emulation aids board-level testing. This feature allows a mounted 80303 I/O processor to electrically "remove" itself from a circuit board. This allows for system-level testing where a remote tester exercises the processor system. In ONCE mode, the processor presents a high impedance on every pin, except for the JTAG test data Output (TDO). All pull-up transistors present on input pins are also disabled and internal clocks stop. In this state the processor's power demands on the circuit board are nearly eliminated.

Note: Do not use ONCE mode with boundary-scan (JTAG). See Section 24.1.2, "ONCE Mode and Boundary-Scan (JTAG) are Incompatible" on page 24-2.

24.1.1 Entering/Exiting ONCE Mode

The **ONCE#** pin, in concert with the **RESET#** pin, invokes ONCE mode.

To invoke ONCE mode, assert the **ONCE#** pin (low) while the processor is in the reset state. (The processor recognizes the **ONCE#** pin signal only while **RESET#** is asserted.) The processor enters ONCE mode immediately. The rising edge of **RESET#** latches the **ONCE#** pin state until **RESET#** goes true again.

Enter ONCE mode by asserting the following sequence with an external tester:

1. Drive the **ONCE#** pin low (overcoming the internal pull-up resistor).
2. Initiate a normal reset cycle.
3. After the **RESET#** pin goes high again, the **ONCE#** pin can be deasserted.

Exit ONCE mode, by performing a normal reset with the **RESET#** pin while holding the **ONCE#** pin high. A power off-on cycle is not necessary to exit ONCE mode.

See the *Intel® 80960RM I/O Processor Data Sheet* for specific timing of the **ONCE#** pin and the characteristics of the on-circuit emulation mode.

24.1.2 ONCE Mode and Boundary-Scan (JTAG) are Incompatible

Permanent damage can occur when an in-circuit emulator is used concurrently with boundary-scan (JTAG). Do not use any system that relies on ONCE mode when using boundary-scan. Signal contentions and resultant damage may occur if an external system, such as an emulator development system, invokes ONCE mode and manipulates the 80303 I/O processor signals while JTAG is active.

Since the 80303 I/O processor complies fully with IEEE Std. 1149.1, JTAG boundary-scan instructions always override ONCE mode. While ONCE mode intends to disable all processor outputs so an external emulator can drive them, JTAG boundary-scan can enable those outputs, causing contention with the external emulator.

To avoid damage, and as a general design rule, force **TRST#** low to disable boundary-scan whenever ONCE mode is active.

24.1.2.1 DEN# Alternatives

To use an **ICE** with your 80303 design, alternatives to **DEN#** are:

- Ground the **OE#** pin of the transceiver
- Re-create a **DEN#** signal with the circuit shown below

24.2 Boundary-Scan (JTAG)

The 80303 I/O processor provides test features compliant to IEEE standard test access port and boundary-scan architecture (IEEE Std. 1149.1). JTAG ensures that components function correctly, connections between components are correct, and components interact correctly on the printed circuit board.

To date, the i960 Hx, Jx and Rx processors implement IEEE 1149.1 standard test access port and boundary-scan architecture, and i960 Kx, Sx and Cx processors do not. For information about using JTAG in a design, refer to IEEE Std. 1149.1 (available from the Institute of Electrical and Electronics Engineers Inc., 345 E. 47th St., New York, NY 10017).

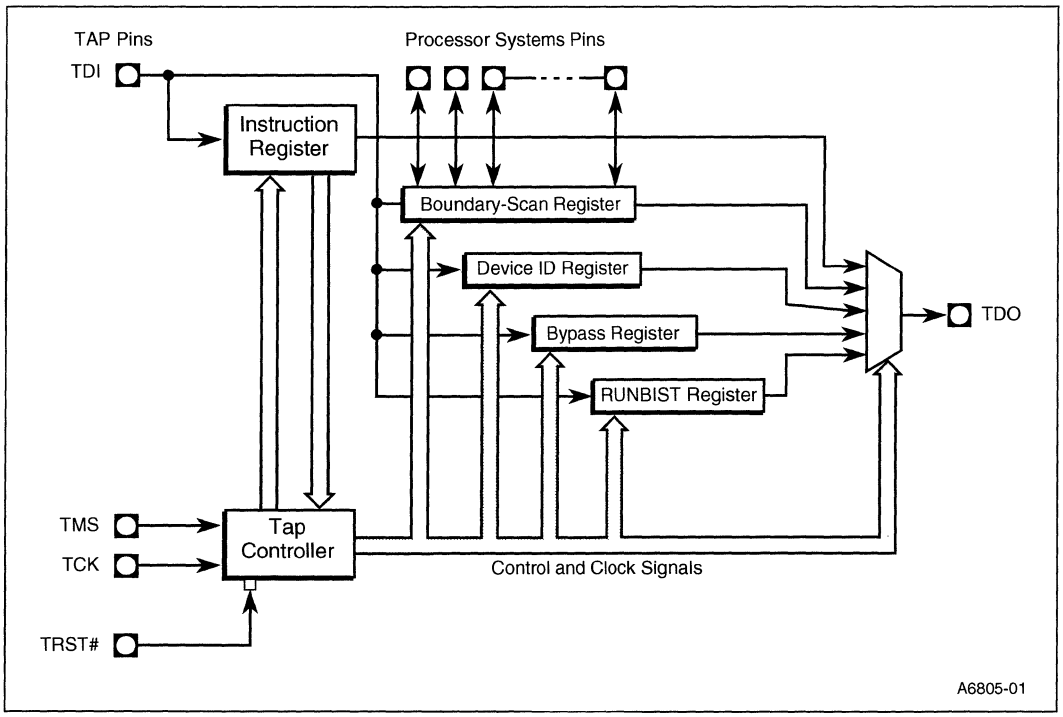
Note: Do not use ONCE mode with boundary-scan (JTAG). See Section 24.1.2, "ONCE Mode and Boundary-Scan (JTAG) are Incompatible" on page 24-2.

24.2.1 Boundary-Scan Architecture

Boundary-scan test logic consists of a boundary-scan register and support logic. These are accessed through a Test Access Port (TAP). The TAP provides a simple serial interface that allows all processor signal pins to be driven and/or sampled, thereby providing direct control and monitoring of processor pins at the system level.

This mode of operation is valuable for design debugging and fault diagnosis since it permits examination of connections not normally accessible to the test system. The following subsections describe the boundary-scan test logic elements: TAP pins, instruction register, test data registers and TAP controller. Figure 24-1 illustrates how these pieces fit together to form the JTAG unit.

Figure 24-1. Test Access Port Block Diagram



24.2.2 TAP Pins

The 80303 I/O processor's TAP pins form a serial port composed of four input connections (**TMS**, **TCK**, **TRST#** and **TDI**) and one output connection (TDO). These pins are described in Table 24-1. The TAP pins provide access to the instruction register and the test data registers.

Table 24-1. TAP Controller Pin Definitions

Pin	Type	Definition
TCK	Input	**Test Clock** provides the clock for the JTAG logic. The JTAG test logic retains its state indefinitely when TCK is stopped at "0" or "1".
TMS	Input	**Test Mode** is decoded by the TAP controller state machine to control test operations. TMS is sampled by the test logic on the rising edge of TCK. TMS is pulled high internally when not driven.
TDI	Input	**Test Data Input** is the serial port where test instructions and data is received by the test logic. Signals presented at TDI are sampled into the test logic on the rising edge of TCK. TDI is pulled high internally when not driven. Data shifted into TDI is not inverted on its way to the TDO input.
TDO	Output	**Test Data Output** is the serial output for test instructions and data from the JTAG test logic. Changes in the state of TDO occur only on the falling edge of TCK. The TDO output is active only during data shifting (SHDR or SHIR); it is inactive (high-Z) at all other times.
TRST#	Input	**Test Reset** provides for an asynchronous initialization of the TAP controller. Asserting a logic "0" on this pin puts the TAP controller state machine and all other test logic on the processor in the *Test-Logic-Reset* (initial) state. **TRST#** is pulled high internally when not driven. **Note:** The system must ensure that **TRST#** is asserted after power-up in order to put the TAP controller in a known state. Failure to do so may cause improper processor operation.

24.2.3 Instruction Register

The Instruction Register (IR) holds instruction codes. These codes are shifted in through the Test Data Input (TDI) pin. The instruction codes are used to select the specific test operation to be performed and the test data register to be accessed.

The instruction register is a parallel-loadable, master/slave-configured 4-bit wide, serial-shift register with latched outputs. Data is shifted into and out of the IR serially through the TDI pin clocked by the rising edge of TCK when the TAP controller is in the Shift_IR state. The shifted-in instruction becomes active upon latching from the master stage to the slave stage in the Update_IR state. At that time the IR outputs along with the TAP finite state machine outputs are decoded to select and control the test data register selected by that instruction. Upon latching, all actions caused by any previous instructions terminates.

The instruction determines the test to be performed, the test data register to be accessed, or both (Table 24-2). The IR is four bits wide. When the IR is selected in the Shift_IR state, the most significant bit is connected to TDI, and the least significant bit is connected to TDO. The value presented on the TDI pin is shifted into the IR on each rising edge of TCK, as long as the TAP controller remains in the Shift_IR state. When the TAP controller changes to the Capture_IR state, fixed parallel data (0001_2) is captured. During Shift_IR, when a new instruction is shifted in through TDI, the value 0001_2 is always shifted out through TDO, least significant bit first. This helps identify instructions in a long chain of serial data from several devices.

Upon activation of the **TRST#** reset pin, the latched instruction asynchronously changes to the **idcode** instruction. When the TAP controller moves into the Test_Logic_Reset state other than by reset activation, the opcode changes as TDI is shifts, and becomes active on the falling edge of TCK. See Figure 24-4 for an example of loading the instruction register.

24.2.3.1 Boundary-Scan Instruction Set

The 80303 I/O processor supports three mandatory boundary-scan instructions (**bypass, sample/preload** and **extest**) plus four additional public instructions (**idcode**, **clamp**, **highz** and **runbist**). Table 24-2 lists the 80303 I/O processor's boundary-scan instruction codes. Those codes listed as "not used" or "private" should not be used.

Table 24-2. Boundary-Scan Instruction Set

Instruction Code	Instruction Name	Instruction Code	Instruction Name
0000_2	**extest**	1000_2	**highz**
0001_2	**sample/preload**	1001_2	not used
0010_2	**idcode**	1010_2	not used
0011_2	**not used**	1011_2	**private**
0100_2	**clamp**	1100_2	**private**
0101_2	not used	1101_2	not used
0110_2	not used	1110_2	not used
0111_2	**runbist**	1111_2	**bypass**

Table 24-3. IEEE Instructions

Instruction / Requisite	Opcode	Description
extest IEEE 1149.1 Required	0000_2	**extest** initiates testing of external circuitry, typically board-level interconnects and off chip circuitry. **extest** connects the boundary-scan register between TDI and TDO in the Shift_DR state only. When **extest** is selected, all output signal pin values are driven by values shifted into the boundary-scan register and may change only on the falling edge of TCK in the Update_DR state. Also, when **extest** is selected, all system input pin states must be loaded into the boundary-scan register on the rising-edge of TCK in the Capture_DR state. Values shifted into input latches in the boundary-scan register are never used by the processor's internal logic.
sample/ preload IEEE 1149.1 Required	0001_2	**sample/preload** performs two functions: • When the TAP controller is in the Capture-DR state, the **sample** instruction occurs on the rising edge of TCK and provides a snapshot of the component's normal operation without interfering with that normal operation. The instruction causes boundary-scan register cells associated with outputs to sample the value being driven by or to the processor. • When the TAP controller is in the Update-DR state, the **preload** instruction occurs on the falling edge of TCK. This instruction causes the transfer of data held in the boundary-scan cells to the slave register cells. Typically the slave latched data is applied to the system outputs via the **extest** instruction.
idcode IEEE 1149.1 Optional	0010_2	**idcode** is used in conjunction with the device identification register. It connects the device identification register between TDI and TDO in the Shift_DR state. When selected, **idcode** parallel-loads the hard-wired identification code (32 bits) into the device identification register on the rising edge of TCK in the Capture_DR state. **NOTE:** Device identification register is not altered by data being shifted in on TDI.
runbist Intel® 80303 I/O processor Optional	0111_2	**runbist** selects the one-bit RUNBIST register, loads a value of 1 into it and connects it to TDO. It also initiates the processor's built-in self test (BIST) feature which is able to detect approximately 82% of all the possible stuck-at faults on the device. The processor AC/DC specifications for V_{CC} and CLKIN must be met and **RESET#** must be de-asserted prior to executing **runbist**. After loading **runbist** instruction code into the instruction register, the TAP controller must be placed in the Run-Test/Idle state. BIST begins on the first rising edge of TCK after the Run-Test/Idle state is entered. The TAP controller must remain in the Run-Test/Idle state until BIST is completed. **runbist** requires approximately 414,000 core cycles to complete BIST and report the result to the RUNBIST register. The results are stored in bit 0 of the RUNBIST register. After the report completes, the value in the RUNBIST register is shifted out on TDO during the Shift-DR state. A value of 0 being shifted out on TDO indicates BIST completed successfully. A value of 1 indicates a failure occurred. After BIST completes, the processor must be cycled through the reset state to resume normal operation.
bypass IEEE 1149.1 Required	1111_2	**bypass** instruction selects the one-bit bypass register between TDI and TDO pins while in SHIFT_DR state, effectively bypassing the processor's test logic. 0_2 is captured in the CAPTURE_DR state. This is the only instruction that accesses the bypass register. While this instruction is in effect, all other test data registers have no effect on system operation. Test data registers with both test and system functionality perform their system functions when this instruction is selected.
highz	1000_2	Executing **highz** generates a signal that is read on the rising-edge of **RESET#**. When this signal is found asserted, the device is put into the ONCE mode (all output pins are floated). Also, when this instruction is active, the Bypass register is connected between TDI and TDO. This register can be accessed via the JTAG Test-Access Port throughout the device operation. Access to the Bypass register can also be obtained with the **bypass** instruction. **highz** provides an alternate method of entering ONCE mode.
clamp	0100_2	**clamp** instruction allows the state of the signals driven from the Intel® i960® Jx processor pins to be determined from the boundary-scan register while the BYPASS register is selected as the serial path between TDI and TDO. Signals driven from the component pins does not change while the **clamp** instruction is selected.

24.2.4 TAP Test Data Registers

The 80303 I/O processor contains four test data registers (device identification, bypass, RUNBIST and boundary-scan). Each test data register selected by the TAP controller is connected serially between TDI and TDO. TDI is connected to the test data register's most significant bit. TDO is connected to the least significant bit. Data is shifted one bit position within the register towards TDO on each rising edge of TCK. While any register is selected, data is transferred from TDI to TDO without inversion. The following sections describe each of the test data registers. See Figure 24-5 for an example of loading the data register.

24.2.4.1 Device Identification Register

The device identification register is a 32-bit register containing the manufacturer's identification code, part number code, version code and other information in the format shown in the *Intel®80960RM I/O Processor Data Sheet*. The identification register is selected only by the **idcode** instruction. When the TAP controller's Test_Logic_Reset state is entered, **idcode** is asynchronously loaded into the instruction register. The device identification register loads the fixed parallel input value in the Capture_DR state.

24.2.4.2 Bypass Register

The required bypass register, a one-bit shift register, provides the shortest path between TDI and TDO when a **bypass** instruction is in effect. This allows rapid movement of test data to and from other components on the board. This path can be selected when no test operation is being performed on the processor.

24.2.4.3 RUNBIST Register

The RUNBIST register, a one-bit register, contains the result of the execution of the processor's BIST routine. After the built-in self-test completes, the processor must be cycled through the reset state to resume normal operation. See Chapter 11, "Initialization and System Requirements" for details of the built-in self test algorithm. The processor runs the BIST routine when the TAP controller enters the Test_Logic_Reset state while the **runbist** instruction is selected.

24.2.4.4 Boundary-Scan Register

The boundary-scan register contains a cell for each pin as well as control cells for I/O and the HIGHZ pin.

Table 24-4 shows the bit order of the 80303 I/O processor boundary-scan register. All table cells that contain "Control" select the direction of bidirectional pins or HIGHZ output pins. When a "0" is loaded into the control cell, the associated pin(s) are HIGHZ or selected as input.

The boundary-scan register is a required set of serial-shiftable register cells, configured in master/slave stages and connected between each of the 80303 I/O processor's pins and on-chip system logic. The V_{CC}, V_{SS} and JTAG pins are NOT in the boundary-scan chain.

The boundary-scan register cells are dedicated logic and do not have any system function. Data may be loaded into the boundary-scan register master cells from the device input pins and output pin-drivers in parallel by the mandatory **sample/preload** and **extest** instructions. Parallel loading takes place on the rising edge of TCK in the Capture_DR state.

Data may be scanned into the boundary-scan register serially via the TDI serial input pin, clocked by the rising edge of TCK in the Shift_DR state. When the required data has been loaded into the master-cell stages, it can be driven into the system logic at input pins or onto the output pins on the falling edge of TCK in the Update_DR state. Data may also be shifted out of the boundary-scan register by means of the TDO serial output pin at the falling edge of TCK.

Table 24-4. Intel® 80303 I/O Processor Boundary Scan Register Bit Order (Sheet 1 of 9)

"0	(CBSC_1,	dq(44),	bidir,	X,	85,	1,	Z),	"&
"1	(CBSC_1,	dq(45),	bidir,	X,	89,	1,	Z),	"&
"2	(CBSC_1,	dq(43),	bidir,	X,	85,	1,	Z),	"&
"3	(CBSC_1,	dq(13),	bidir,	X,	78,	1,	Z),	"&
"4	(CBSC_1,	dq(46),	bidir,	X,	89,	1,	Z),	"&
"5	(CBSC_1,	dq(15),	bidir,	X,	78,	1,	Z),	"&
"6	(CBSC_1,	dq(14),	bidir,	X,	78,	1,	Z),	"&
"7	(CBSC_1,	dq(12),	bidir,	X,	78,	1,	Z),	"&
"8	(CBSC_1,	dq(47),	bidir,	X,	89,	1,	Z	"&
"9	(CBSC_1,	scb(5),	bidir,	X,	83,	1,	Z),	"&
"10	(CBSC_1,	scb(1),	bidir,	X,	80,	1,	Z),	"&
"11	(CBSC_1,	scb(0),	bidir,	X,	79,	1,	Z),	"&
"12	(CBSC_1,	scb(4),	bidir,	X,	83,	1,	Z),	"&
"13	(BC_1,	scasz,	output3,	X,	90,	1,	Z),	"&
"14	(CBSC_1,	sdqm(0),	bidir,	X,	84,	1,	Z),	"&
"15	(CBSC_1,	sdqm(4),	bidir,	X,	84,	1,	Z),	"&
"16	(BC_1,	scez(1),	output3,	X,	90,	1,	Z),	"&
"17	(BC_1,	swez,	output3,	X,	90,	1,	Z),	"&
"18	(CBSC_1,	sdqm(5),	bidir,	X,	84,	1,	Z),	"&
"19	(BC_1,	scez(0),	output3,	X,	90,	1,	Z),	"&
"20	(BC_1,	sa(1),	output3,	X,	90,	1,	Z),	"&
"21	(CBSC_1,	sdqm(1),	bidir,	X,	84,	1,	Z),	"&
"22	(BC_1,	sa(2),	output3,	X,	90,	1,	Z),	"&
"23	(BC_1,	srasz,	output3,	X,	90,	1,	Z),	"&
"24	(BC_1,	sa(0),	output3,	X,	90,	1,	Z),	"&
"25	(BC_1,	sa(4),	output3,	X,	90,	1,	Z),	"&
"26	(BC_1,	sa(6),	output3,	X,	90,	1,	Z),	"&

Table 24-4. Intel® 80303 I/O Processor Boundary Scan Register Bit Order (Sheet 2 of 9)

"27	(BC_1,	sa(8),	output3,	X,	90,	1,	Z),	"&
"28	(BC_1,	sa(5),	output3,	X,	90,	1,	Z),	"&
"29	(BC_1,	scke(0),	output3,	X,	90,	1,	Z),	"&
"30	(BC_1,	sa(3),	output3,	X,	90,	1,	Z),	"&
"31	(BC_1,	sa(10),	output3,	X,	90,	1,	Z),	"&
"32	(BC_1,	sba(1),	output3,	X,	90,	1,	Z),	"&
"33	(BC_1,	sa(9),	output3,	X,	90,	1,	Z),	"&
"34	(BC_1,	scke(1),	output3,	X,	90,	1,	Z),	"&
"35	(BC_1,	sa(7),	output3,	X,	90,	1,	Z),	"&
"36	(CBSC_1,	sdqm(6),	bidir,	X,	84,	1,	Z),	"&
"37	(BC_1,	sba(0),	output3,	X,	90,	1,	Z),	"&
"38	(CBSC_1,	scb(2),	bidir,	X,	81,	1,	Z),	"&
"39	(BC_1,	sa(11),	output3,	X,	90,	1,	Z),	"&
"40	(CBSC_1,	sdqm(2),	bidir,	X,	84,	1,	Z),	&
"41	(CBSC_1,	sdqm(3),	bidir,	X,	84,	1,	Z),	"&
"42	(CBSC_1,	scb(3),	bidir,	X,	82,	1,	Z),	"&
"43	(CBSC_1,	sdqm(7),	bidir,	X,	84,	1,	Z),	"&
"44	(CBSC_1,	scb(7),	bidir,	X,	87,	1,	Z),	"&
"45	(CBSC_1,	scb(6),	bidir,	X,	86,	1,	Z),	"&
"46	(CBSC_1,	dq(48),	bidir,	X,	89,	1,	Z),	"&
"47	(CBSC_1,	dq(17),	bidir,	X,	78,	1,	Z),	"&
"48	(CBSC_1,	dq(16),	bidir,	X,	78,	1,	Z),	"&
"49	(CBSC_1,	dq(18),	bidir,	X,	78,	1,	Z),	"&
"50	(CBSC_1,	dq(49),	bidir,	X,	89,	1,	Z),	"&
"51	(CBSC_1,	dq(50),	bidir,	X,	89,	1,	Z),	"&
"52	(CBSC_1,	dq(19),	bidir,	X,	78,	1,	Z),	"&
"53	(CBSC_1,	dq(52),	bidir,	X,	89,	1,	Z),	"&
"54	(CBSC_1,	dq(51),	bidir,	X,	89,	1,	Z),	"&
"55	(CBSC_1,	dq(20),	bidir,	X,	78,	1,	Z),	"&
"56	(CBSC_1,	dq(53),	bidir,	X,	88,	1,	Z),	"&
"57	(CBSC_1,	dq(21),	bidir,	X,	78,	1,	Z),	"&
"58	(CBSC_1,	dq(23),	bidir,	X,	78,	1,	Z),	"&
"59	(CBSC_1,	dq(22),	bidir,	X,	78,	1,	Z),	"&
"60	(CBSC_1,	dq(24),	bidir,	X,	78,	1,	Z),	"&
"61	(CBSC_1,	dq(54),	bidir,	X,	88,	1,	Z),	"&
"62	(CBSC_1,	dq(56),	bidir,	X,	88,	1,	Z),	"&
"63	(CBSC_1,	dq(57),	bidir,	X,	85,	1,	Z),	"&
"64	(CBSC_1,	dq(55),	bidir,	X,	88,	1,	Z),	"&
"65	(CBSC_1,	dq(58),	bidir,	X,	85,	1,	Z),	"&
"66	(CBSC_1,	dq(25),	bidir,	X,	78,	1,	Z),	"&
"67	(CBSC_1,	dq(27),	bidir,	X,	78,	1,	Z),	"&
"68	(CBSC_1,	dq(26),	bidir,	X,	78,	1,	Z),	"&
"6	(CBSC_1,	dq(60),	bidir,	X,	85,	1,	Z),	"&
"70	(CBSC_1,	dq(59),	bidir,	X,	85,	1,	Z),	"&
"71	(CBSC_1,	dq(28),	bidir,	X,	78,	1,	Z),	"&
"72	(CBSC_1,	dq(29),	bidir,	X,	78,	1,	Z),	"&
"73	(CBSC_1,	dq(31),	bidir,	X,	78,	1,	Z),	"&
"74	(CBSC_1,	dq(30),	bidir,	X,	78,	1,	Z),	"&
"75	(CBSC_1,	dq(61),	bidir,	X,	85,	1,	Z),	"&

Table 24-4. Intel® 80303 I/O Processor Boundary Scan Register Bit Order (Sheet 3 of 9)

"76	(CBSC_1,	dq(62),	bidir,	X,	85,	1,	Z),	"&
"77	(CBSC_1,	dq(63),	bidir,	X,	85,	1,	Z),	"&
"78	(BC_1,	*,	control,	1),	" &			
"79	(BC_1,	*,	control,	1),	" &			
"80	(BC_1,	*,	control,	1),	" &			
"81	(BC_1,	*,	control,	1),	" &			
"82	(BC_1,	*,	control,	1),	" &			
"83	(BC_1,	*,	control,	1),	" &			
"84	(BC_1,	*,	control,	1),	" &			
"85	(BC_1,	*,	control,	1),	" &			
"86	(BC_1,	*,	control,	1),	" &			
"87	(BC_1,	*,	control,	1),	" &			
"88	(BC_1,	*,	control,	1),	" &			
"89	(BC_1,	*,	control,	1),	" &			
"90	(BC_1,	*,	control,	1),	" &			
"91	(BC_4,	s_reqz(3),	input,	X),	"&			
"92	(BC_4,	s_reqz(5),	input,	X),	"&			
"93	(CBSC_1,	s_gntz(3),	bidir,	X,	184,	1,	Z),	"&
"94	(CBSC_1,	s_gntz(5),	bidir,	X,	184,	1,	Z),	"&
"95	(BC_4,	s_reqz(1),	input,	X),	"&			
"96	(BC_4,	s_reqz(4),	input,	X),	"&			
"97	(CBSC_1,	s_gntz(4),	bidir,	X,	184,	1,	Z),	"&
"98	(BC_4,	s_reqz(2),	input,	X),	"&			
"99	(CBSC_1,	s_gntz(1),	bidir,	X,	184,	1,	Z),	"&
"100	(CBSC_1,	s_ad(31),	bidir,	X,	189,	1,	Z),	"&
"101	(CBSC_1,	s_gntz(2),	bidir,	X,	184,	1,	Z),	"&
"102	(CBSC_1,	s_gntz(0),	bidir,	X,	184,	1,	Z),	"&
"103	(BC_1,	s_rstz,	output3,	X,	205,	1,	Z),	"&
"104	(CBSC_1,	s_ad(27),	bidir,	X,	189,	1,	Z),	"&
"105	(CBSC_1,	s_ad(28),	bidir,	X,	189,	1,	Z),	"&
"106	(BC_4,	s_reqz(0),	input,	X),	"&			
"107	(CBSC_1,	s_ad(30),	bidir,	X,	189,	1,	Z),	"&
"108	(CBSC_1,	s_ad(29),	bidir,	X,	189,	1,	Z),	"&
"109	(CBSC_1,	s_cbez(3),	bidir,	X,	193,	1,	Z),	"&
"110	(CBSC_1,	s_ad(24),	bidir,	X,	189,	1,	Z),	"&
"111	(CBSC_1,	s_ad(25),	bidir,	X,	189,	1,	Z),	"&
"112	(CBSC_1,	s_ad(26),	bidir,	X,	189,	1,	Z),	"&
"113	(CBSC_1,	s_ad(20),	bidir,	X,	190,	1,	Z),	"&
"114	(CBSC_1,	s_ad(23),	bidir,	X,	190,	1,	Z),	"&
"115	(CBSC_1,	s_ad(21),	bidir,	X,	190,	1,	Z),	"&
"116	(CBSC_1,	s_ad(17),	bidir,	X,	190,	1,	Z),	"&
"117	(CBSC_1,	s_ad(22),	bidir,	X,	190,	1,	Z),	"&
"118	(CBSC_1,	s_ad(18),	bidir,	X,	190,	1,	Z),	"&
"119	(CBSC_1,	s_ad(16),	bidir,	X,	190,	1,	Z),	"&
"120	(CBSC_1,	s_framez,	bidir,	X,	204,	1,	Z),	"&
"121	(CBSC_1,	s_ad(19),	bidir,	X,	190,	1,	Z),	"&
"122	(CBSC_1,	s_trdyz,	bidir,	X,	202,	1,	Z),	"&
"123	(CBSC_1,	s_perrz,	bidir,	X,	200,	1,	Z),	"&
"124	(CBSC_1,	s_irdyz,	bidir,	X,	203,	1,	Z),	"&

Table 24-4. Intel® 80303 I/O Processor Boundary Scan Register Bit Order (Sheet 4 of 9)

"125	(CBSC_1,	s_lockz,	bidir,	X,	201,	1,	Z),	"&
"126	(CBSC_1,	s_cbez(2),	bidir,	X,	193,	1,	Z),	"&
"127	(CBSC_1,	s_par,	bidir,	X,	198,	1,	Z),	"&
"128	(CBSC_1,	s_stopz,	bidir,	X,	202,	1,	Z),	"&
"129	(CBSC_1,	s_ad(15),	bidir,	X,	191,	1,	Z),	"&
"130	(CBSC_1,	s_serrz,	bidir,	X,	199,	1,	Z),	"&
"131	(CBSC_1,	s_devselz,	bidir,	X,	202,	1,	Z),	"&
"132	(CBSC_1,	s_ad(11),	bidir,	X,	191,	1,	Z),	"&
"133	(CBSC_1,	s_cbez(1),	bidir,	X,	193,	1,	Z),	"&
"134	(CBSC_1,	s_cbez(0),	bidir,	X,	193,	1,	Z),	"&
"135	(CBSC_1,	s_ad(14),	bidir,	X,	191,	1,	Z),	"&
"136	(CBSC_1,	s_ad(12),	bidir,	X,	191,	1,	Z),	"&
"137	(CBSC_1,	s_ad(9),	bidir,	X,	191,	1,	Z),	"&
"138	(CBSC_1,	s_ad(13),	bidir,	X,	191,	1,	Z),	"&
"139	(CBSC_1,	s_ad(4),	bidir,	X,	192,	1,	Z),	"&
"140	(CBSC_1,	s_ad(10),	bidir,	X,	191,	1,	Z),	"&
"141	(CBSC_1,	s_ad(8),	bidir,	X,	191,	1,	Z),	"&
"142	(CBSC_1,	s_ad(7),	bidir,	X,	192,	1,	Z),	"&
"143	(CBSC_1,	s_ad(5),	bidir,	X,	192,	1,	Z),	"&
"144	(CBSC_1,	s_ad(6),	bidir,	X,	192,	1,	Z),	"&
"145	(CBSC_1,	s_ad(3),	bidir,	X,	192,	1,	Z),	"&
"146	(CBSC_1,	s_cbez(6),	bidir,	X,	194,	1,	Z),	"&
"147	(CBSC_1,	s_ad(1),	bidir,	X,	192,	1,	Z),	"&
"148	(CBSC_1,	s_ad(2),	bidir,	X,	192,	1,	Z),	"&
"149	(CBSC_1,	s_ad(59),	bidir,	X,	185,	1,	Z),	"&
"150	(CBSC_1,	s_ad(0),	bidir,	X,	192,	1,	Z),	"&
"151	(CBSC_1,	s_ack64z,	bidir,	X,	195,	1,	Z),	"&
"152	(CBSC_1,	s_req64z,	bidir,	X,	196,	1,	Z),	"&
"153	(CBSC_1,	s_cbez(7),	bidir,	X,	194,	1,	Z),	"&
"154	(CBSC_1,	s_cbez(4),	bidir,	X,	194,	1,	Z),	"&
"155	(CBSC_1,	s_cbez(5),	bidir,	X,	194,	1,	Z),	"&
"156	(CBSC_1,	s_par64,	bidir,	X,	197,	1,	Z),	"&
"157	(CBSC_1,	s_ad(63),	bidir,	X,	185,	1,	Z),	"&
"158	(CBSC_1,	s_ad(51),	bidir,	X,	186,	1,	Z),	"&
"159	(CBSC_1,	s_ad(62),	bidir,	X,	185,	1,	Z),	"&
"160	(CBSC_1,	s_ad(61),	bidir,	X,	185,	1,	Z),	"&
"161	(CBSC_1,	s_ad(57),	bidir,	X,	185,	1,	Z),	"&
"162	(CBSC_1,	s_ad(60),	bidir,	X,	185,	1,	Z),	"&
"163	(CBSC_1,	s_ad(55),	bidir,	X,	186,	1,	Z),	"&
"164	(CBSC_1,	s_ad(58),	bidir,	X,	185,	1,	Z),	"&
"165	(CBSC_1,	s_ad(53),	bidir,	X,	186,	1,	Z),	"&
"166	(CBSC_1,	s_ad(56),	bidir,	X,	185,	1,	Z),	"&
"167	(CBSC_1,	s_ad(50),	bidir,	X,	186,	1,	Z),	"&
"168	(CBSC_1,	s_ad(54),	bidir,	X,	186,	1,	Z),	"&
"169	(CBSC_1,	s_ad(47),	bidir,	X,	187,	1,	Z),	"&
"170	(CBSC_1,	s_ad(52),	bidir,	X,	186,	1,	Z),	"&
"171	(CBSC_1,	s_ad(42),	bidir,	X,	187,	1,	Z),	"&
"172	(CBSC_1,	s_ad(49),	bidir,	X,	186,	1,	Z),	"&
"173	(CBSC_1,	s_ad(45),	bidir,	X,	187,	1,	Z),	"&

Table 24-4. Intel® 80303 I/O Processor Boundary Scan Register Bit Order (Sheet 5 of 9)

"174	(CBSC_1,	s_ad(48),	bidir,	X,	186,	1,	Z),	"&
"175	(CBSC_1,	s_ad(43),	bidir,	X,	187,	1,	Z),	"&
"176	(CBSC_1,	s_ad(41),	bidir,	X,	187,	1,	Z),	"&
"177	(CBSC_1,	s_ad(46),	bidir,	X,	187,	1,	Z),	"&
"178	(CBSC_1,	s_ad(39),	bidir,	X,	188,	1,	Z),	"&
"179	(CBSC_1,	s_ad(44),	bidir,	X,	187,	1,	Z),	"&
"180	(CBSC_1,	s_ad(37),	bidir,	X,	188,	1,	Z),	"&
"181	(CBSC_1,	s_ad(40),	bidir,	X,	187,	1,	Z),	"&
"182	(CBSC_1,	s_ad(36),	bidir,	X,	188,	1,	Z),	"&
"183	(CBSC_1,	s_ad(38),	bidir,	X,	188,	1,	Z),	"&
"184	(BC_1,	*,	control,	1),	" &			
"185	(BC_1,	*,	control,	1),	" &			
"186	(BC_1,	*,	control,	1),	" &			
"187	(BC_1,	*,	control,	1),	" &			
"188	(BC_1,	*,	control,	1),	" &			
"189	(BC_1,	*,	control,	1),	" &			
"190	(BC_1,	*,	control,	1),	" &			
"191	(BC_1,	*,	control,	1),	" &			
"192	(BC_1,	*,	control,	1),	" &			
"193	(BC_1,	*,	control,	1),	" &			
"194	(BC_1,	*,	control,	1),	" &			
"195	(BC_1,	*,	control,	1),	" &			
"196	(BC_1,	*,	control,	1),	" &			
"197	(BC_1,	*,	control,	1),	" &			
"198	(BC_1,	*,	control,	1),	" &			
"199	(BC_1,	*,	control,	1),	" &			
"200	(BC_1,	*,	control,	1),	" &			
"201	(BC_1,	*,	control,	1),	" &			
"202	(BC_1,	*,	control,	1),	" &			
"203	(BC_1,	*,	control,	1),	" &			
"204	(BC_1,	*,	control,	1),	" &			
"205	(BC_1,	*,	control,	1),	" &			
"206	(CBSC_1,	s_ad(33),	bidir,	X,	292,	1,	Z),	"&
"207	(CBSC_1,	s_ad(35),	bidir,	X,	292,	1,	Z),	"&
"208	(CBSC_1,	s_ad(32),	bidir,	X,	292,	1,	Z),	"&
"209	(CBSC_1,	s_ad(34),	bidir,	X,	292,	1,	Z),	"&
"210	(BC_4,	nc1,	input,	X),	"&			
"211	(CBSC_1,	p_ad(33),	bidir,	X,	293,	1,	Z),	"&
"212	(CBSC_1,	p_ad(34),	bidir,	X,	293,	1,	Z),	"&
"213	(CBSC_1,	p_ad(32),	bidir,	X,	293,	1,	Z),	"&
"214	(CBSC_1,	p_ad(40),	bidir,	X,	294,	1,	Z),	"&
"215	(CBSC_1,	p_ad(36),	bidir,	X,	293,	1,	Z),	"&
"216	(CBSC_1,	p_ad(37),	bidir,	X,	293,	1,	Z),	"&
"217	(CBSC_1,	p_ad(39),	bidir,	X,	293,	1,	Z),	"&
"218	(CBSC_1,	p_ad(35),	bidir,	X,	293,	1,	Z),	"&
"219	(CBSC_1,	p_ad(43),	bidir,	X,	294,	1,	Z),	"&
"220	(CBSC_1,	p_ad(41),	bidir,	X,	294,	1,	Z),	"&
"221	(CBSC_1,	p_ad(49),	bidir,	X,	295,	1,	Z),	"&
"222	(CBSC_1,	p_ad(38),	bidir,	X,	293,	1,	Z),	"&

Table 24-4. Intel® 80303 I/O Processor Boundary Scan Register Bit Order (Sheet 6 of 9)

"223	(CBSC_1,	p_ad(45),	bidir,	X,	294,	1,	Z),	"&
"224	(CBSC_1,	p_ad(48),	bidir,	X,	295,	1,	Z),	"&
"225	(CBSC_1,	p_ad(44),	bidir,	X,	294,	1,	Z),	"&
"226	(CBSC_1,	p_ad(46),	bidir,	X,	294,	1,	Z),	"&
"227	(CBSC_1,	p_ad(42),	bidir,	X,	294,	1,	Z),	"&
"228	(CBSC_1,	p_ad(51),	bidir,	X,	295,	1,	Z),	"&
"229	(CBSC_1,	p_ad(47),	bidir,	X,	294,	1,	Z),	"&
"230	(CBSC_1,	p_ad(53),	bidir,	X,	295,	1,	Z),	"&
"231	(CBSC_1,	p_ad(50),	bidir,	X,	295,	1,	Z),	"&
"232	(CBSC_1,	p_ad(52),	bidir,	X,	295,	1,	Z),	"&
"233	(CBSC_1,	p_ad(57),	bidir,	X,	296,	1,	Z),	"&
"234	(CBSC_1,	p_ad(56),	bidir,	X,	296,	1,	Z),	"&
"235	(CBSC_1,	p_ad(55),	bidir,	X,	295,	1,	Z),	"&
"236	(CBSC_1,	p_ad(54),	bidir,	X,	295,	1,	Z),	"&
"237	(CBSC_1,	p_ad(58),	bidir,	X,	296,	1,	Z),	"&
"238	(CBSC_1,	p_ad(60),	bidir,	X,	296,	1,	Z),	"&
"239	(CBSC_1,	p_ad(61),	bidir,	X,	296,	1,	Z),	"&
"240	(CBSC_1,	p_par64,	bidir,	X,	306,	1,	Z),	"&
"241	(CBSC_1,	p_ad(59),	bidir,	X,	296,	1,	Z),	"&
"242	(CBSC_1,	p_ad(63),	bidir,	X,	296,	1,	Z),	"&
"243	(CBSC_1,	p_cbez(4),	bidir,	X,	310,	1,	Z),	"&
"244	(CBSC_1,	p_cbez(5),	bidir,	X,	310,	1,	Z),	"&
"245	(CBSC_1,	p_ad(62),	bidir,	X,	296,	1,	Z),	"&
"246	(CBSC_1,	p_ack64z,	bidir,	X,	311,	1,	Z),	"&
"247	(CBSC_1,	p_req64z,	bidir,	X,	303,	1,	Z),	"&
"248	(CBSC_1,	p_cbez(7),	bidir,	X,	310,	1,	Z),	"&
"249	(CBSC_1,	p_ad(2),	bidir,	X,	297,	1,	Z),	"&
"250	(CBSC_1,	p_cbez(6),	bidir,	X,	310,	1,	Z),	"&
"251	(CBSC_1,	p_ad(3),	bidir,	X,	297,	1,	Z),	"&
"252	(CBSC_1,	p_ad(1),	bidir,	X,	297,	1,	Z),	"&
"253	(CBSC_1,	p_ad(7),	bidir,	X,	297,	1,	Z),	"&
"254	CBSC_1,	p_ad(0),	bidir,	X,	297,	1,	Z),	"&
"255	(CBSC_1,	p_cbez(0),	bidir,	X,	309,	1,	Z),	"&
"256	(CBSC_1,	p_ad(4),	bidir,	X,	297,	1,	Z),	"&
"257	(CBSC_1,	p_ad(6),	bidir,	X,	297,	1,	Z),	"&
"258	(CBSC_1,	p_ad(9),	bidir,	X,	298,	1,	Z),	"&
"259	(CBSC_1,	p_ad(5),	bidir,	X,	297,	1,	Z),	"&
"260	(CBSC_1,	p_ad(10),	bidir,	X,	298,	1,	Z),	"&
"261	(CBSC_1,	p_ad(8),	bidir,	X,	298,	1,	Z),	"&
"262	(CBSC_1,	p_ad(13),	bidir,	X,	298,	1,	Z),	"&
"263	(CBSC_1,	p_ad(11),	bidir,	X,	298,	1,	Z),	"&
"264	(CBSC_1,	p_ad(12),	bidir,	X,	298,	1,	Z),	"&
"265	(CBSC_1,	p_ad(14),	bidir,	X,	298,	1,	Z),	"&
"266	(CBSC_1,	p_ad(15),	bidir,	X,	298,	1,	Z),	"&
"267	(CBSC_1,	p_cbez(1),	bidir,	X,	309,	1,	Z),	"&
"268	(CBSC_1,	p_par,	bidir,	X,	305,	1,	Z),	"&
"269	(CBSC_1,	p_perrz,	bidir,	X,	304,	1,	Z),	"&
"270	(CBSC_1,	p_serrz,	bidir,	X,	302,	1,	Z),	"&
"271	(CBSC_1,	p_stopz,	bidir,	X,	301,	1,	Z),	"&

Table 24-4. Intel® 80303 I/O Processor Boundary Scan Register Bit Order (Sheet 7 of 9)

"272	(CBSC_1,	p_devselz,	bidir,	X,	301,	1,	Z),	"&
"273	(BC_4,	p_lockz,	input,	X),	"&			
"274	(CBSC_1,	p_trdyz,	bidir,	X,	301,	1,	Z),	"&
"275	(CBSC_1,	p_irdyz,	bidir,	X,	307,	1,	Z),	"&
"276	(CBSC_1,	p_cbez(2),	bidir,	X,	309,	1,	Z),	"&
"277	(CBSC_1,	p_framez,	bidir,	X,	308,	1,	Z),	"&
"278	(CBSC_1,	p_ad(18),	bidir,	X,	299,	1,	Z),	"&
"279	(CBSC_1,	p_ad(17),	bidir,	X,	299,	1,	Z),	"&
"280	(CBSC_1,	p_ad(16),	bidir,	X,	299,	1,	Z),	"&
"281	(CBSC_1,	p_ad(20),	bidir,	X,	299,	1,	Z),	"&
"282	(CBSC_1,	p_ad(19),	bidir,	X,	299,	1,	Z),	"&
"283	(CBSC_1,	p_ad(22),	bidir,	X,	299,	1,	Z),	"&
"284	(CBSC_1,	p_ad(21),	bidir,	X,	299,	1,	Z),	"&
"285	(CBSC_1,	p_ad(23),	bidir,	X,	299,	1,	Z),	"&
"286	(CBSC_1,	p_cbez(3),	bidir,	X,	309,	1,	Z),	"&
"287	(CBSC_1,	p_ad(24),	bidir,	X,	300,	1,	Z),	"&
"288	(BC_4,	p_idsel,	input,	X),	"&			
"289	(CBSC_1,	p_ad(26),	bidir,	X,	300,	1,	Z),	"&
"290	(CBSC_1,	p_ad(25),	bidir,	X,	300,	1,	Z),	"&
"291	(CBSC_1,	p_ad(27),	bidir,	X,	300,	1,	Z),	"&
"292	(BC_1,	*,	control,	1),	" &			
"293	(BC_1,	*,	control,	1),	" &			
"294	(BC_1,	*,	control,	1),	" &			
"295	(BC_1,	*,	control,	1),	" &			
"296	(BC_1,	*,	control,	1),	" &			
"297	(BC_1,	*,	control,	1),	" &			
"298	(BC_1,	*,	control,	1),	" &			
"299	(BC_1,	*,	control,	1),	" &			
"300	(BC_1,	*,	control,	1),	" &			
"301	(BC_1,	*,	control,	1),	" &			
"302	(BC_1,	*,	control,	1),	" &			
"303	(BC_1,	*,	control,	1),	" &			
"304	(BC_1,	*,	control,	1),	" &			
"305	(BC_1,	*,	control,	1),	" &			
"306	(BC_1,	*,	control,	1),	" &			
"307	(BC_1,	*,	control,	1),	" &			
"308	(BC_1,	*,	control,	1),	" &			
"309	(BC_1,	*,	control,	1),	" &			
"310	(BC_1,	*,	control,	1),	" &			
"311	(BC_1,	*,	control,	1),	" &			
"312	(BC_1,	*,	control,	1),	" &			
"313	(CBSC_1,	p_ad(28),	bidir,	X,	401,	1,	Z),	"&
"314	(CBSC_1,	p_ad(30),	bidir,	X,	401,	1,	Z),	"&
"315	(CBSC_1,	p_ad(31),	bidir,	X,	401,	1,	Z),	"&
"316	(CBSC_1,	p_reqz,	bidir,	X,	402,	1,	Z),	"&
"317	(BC_1,	p_intz(2),	output3,	X,	398,	1,	Z),	"&
"318	(BC_1,	p_intz(3),	output3,	X,	397,	1,	Z),	"&
"319	(CBSC_1,	p_ad(29),	bidir,	X,	401,	1,	Z),	"&
"320	(BC_4,	p_rstz,	input,	X),	"&			

Table 24-4. Intel® 80303 I/O Processor Boundary Scan Register Bit Order (Sheet 8 of 9)

"321	(BC_4,	p_gntz,	input,	X),	"&			
"322	(BC_1,	p_intz(1),	output3,	X,	399,	1,	Z),	"&
"323	(BC_1,	p_intz(0),	output3,	X,	400,	1,	Z),	"&
"324	(CBSC_1,	sda,	bidir,	X,	388,	1,	Z),	"&
"325	(CBSC_1,	scl,	bidir,	X,	389,	1,	Z),	"&
"326	(BC_4,	s_intz_xintz(2),	input,	X),	"&			
"327	(BC_4,	s_intz_xintz(1),	input,	X),	"&			
"328	(BC_4,	scnmodez,	input,	X),	"&			
"329	(BC_4,	s_intz_xintz(0),	input,	X),	"&			
"330	(BC_4,	nmiz,	input,	X),	"&			
"331	(BC_4,	xint5z,	input,	X),	"&			
"332	(BC_4,	xint4z,	input,	X),	"&			
"333	(BC_4,	s_intz_xintz(3),	input,	X),	"&			
"334	(BC_1,	i_rstz,	output3,	X,	403,	1,	Z),	"&
"335	(BC_4,	scbodz,	input,	X),	"&			
"336	(BC_1,	failz,	output3,	X,	403,	1,	Z),	"&
"337	(CBSC_1,	rad(3),	bidir,	X,	390,	1,	Z),	"&
"338	(CBSC_1,	rad(0),	bidir,	X,	390,	1,	Z),	"&
"339	(CBSC_1,	rad(2),	bidir,	X,	390,	1,	Z),	&
"340	(CBSC_1,	rad(7),	bidir,	X,	390,	1,	Z),	"&
"341	(CBSC_1,	rad(5),	bidir,	X,	390,	1,	Z),	"&
"342	(CBSC_1,	rad(1),	bidir,	X,	390,	1,	Z),	"&
"343	(CBSC_1,	rad(4),	bidir,	X,	390,	1,	Z),	"&
"344	(CBSC_1,	rad(6),	bidir,	X,	390,	1,	Z),	"&
"345	(CBSC_1,	rad(15),	bidir,	X,	391,	1,	Z),	"&
"346	(CBSC_1,	rad(10),	bidir,	X,	391,	1,	Z),	"&
"347	(CBSC_1,	rad(9),	bidir,	X,	391,	1,	Z),	"&
"348	(CBSC_1,	rad(11),	bidir,	X,	391,	1,	Z),	"&
"349	(CBSC_1,	rad(12),	bidir,	X,	391,	1,	Z),	"&
"350	(CBSC_1,	rad(8),	bidir,	X,	390,	1,	Z),	"&
"351	(CBSC_1,	rad(13),	bidir,	X,	391,	1,	Z),	"&
"352	(CBSC_1,	rad(14),	bidir,	X,	391,	1,	Z),	"&
"353	(CBSC_1,	rad(16),	bidir,	X,	391,	1,	Z),	"&
"354	(BC_1,	rale,	output3,	X,	403,	1,	Z),	"&
"355	(CBSC_1,	rcez(1),	bidir,	X,	395,	1,	Z),	"&
"356	(CBSC_1,	rcez(0),	bidir,	X,	394,	1,	Z),	"&
"357	(BC_4,	p_clk,	input,	X),	"&			
"358	(BC_1,	rwez,	output3,	X,	403,	1,	Z),	"&
"359	(BC_4,	lcdinitz,	input,	X),	"&			
"360	(BC_1,	roez,	output3,	X,	403,	1,	Z),	"&
"361	(BC_4,	p_cclk,	input,	X),	"&			
"362	(BC_4,	oncez,	input,	X),	"&			
"363	(CBSC_1,	dq(32),	bidir,	X,	396,	1,	Z),	"&
"364	(BC_1,	dclkout,	output3,	X,	403,	1,	Z),	"&
"365	(BC_4,	dclkin,	input,	X),	"&			
"366	(CBSC_1,	dq(36),	bidir,	X,	396,	1,	Z),	"&
"367	(CBSC_1,	dq(0),	bidir,	X,	396,	1,	Z),	"&
"368	(CBSC_1,	dq(33),	bidir,	X,	396,	1,	Z),	"&
"369	(CBSC_1,	dq(1),	bidir,	X,	396,	1,	Z),	"&

Table 24-4. Intel® 80303 I/O Processor Boundary Scan Register Bit Order (Sheet 9 of 9)

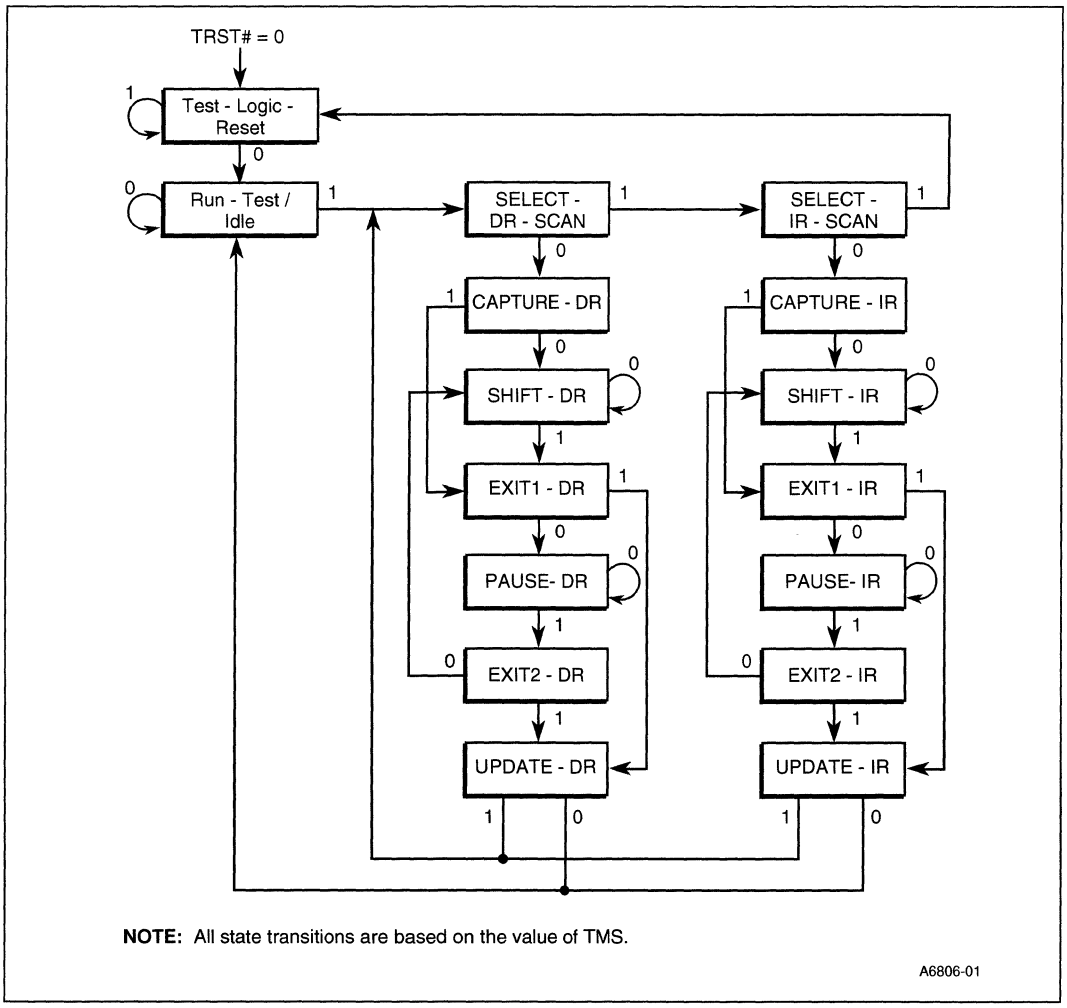
"370	(CBSC_1,	dq(34),	bidir,	X,	396,	1,	Z),	"&
"371	(CBSC_1,	dq(2),	bidir,	X,	396,	1,	Z),	"&
"372	(CBSC_1,	dq(35),	bidir,	X,	396,	1,	Z),	"&
"373	(CBSC_1,	dq(3),	bidir,	X,	396,	1,	Z),	"&
"374	(CBSC_1,	dq(6),	bidir,	X,	396,	1,	Z),	"&
"375	(CBSC_1,	dq(4),	bidir,	X,	396,	1,	Z),	"&
"376	(CBSC_1,	dq(38),	bidir,	X,	392,	1,	Z),	"&
"377	(CBSC_1,	dq(5),	bidir,	X,	396,	1,	Z),	"&
"378	(CBSC_1,	dq(8),	bidir,	X,	396,	1,	Z),	"&
"379	(CBSC_1,	dq(40),	bidir,	X,	392,	1,	Z),	"&
"380	(CBSC_1,	dq(37),	bidir,	X,	396,	1,	Z),	"&
"381	(CBSC_1,	dq(39),	bidir,	X,	392,	1,	Z),	"&
"382	(CBSC_1,	dq(7),	bidir,	X,	396,	1,	Z),	"&
"383	(CBSC_1,	dq(9),	bidir,	X,	396,	1,	Z),	"&
"384	(CBSC_1,	dq(10),	bidir,	X,	396,	1,	Z),	"&
"385	(CBSC_1,	dq(41),	bidir,	X,	392,	1,	Z),	"&
"386	(CBSC_1,	dq(11),	bidir,	X,	396,	1,	Z),	"&
"387	(CBSC_1,	dq(42),	bidir,	X,	393,	1,	Z),	"&
"388	(BC_1,	*,	control,	1),	" &			
"389	(BC_1,	*,	control,	1),	" &			
"390	(BC_1,	*,	control,	1),	" &			
"391	(BC_1,	*,	control,	1),	" &			
"392	(BC_1,	*,	control,	1),	" &			
"393	(BC_1,	*,	control,	1),	" &			
"394	(BC_1,	*,	control,	1),	" &			
"395	(BC_1,	*,	control,	1),	" &			
"396	(BC_1,	*,	control,	1),	" &			
"397	(BC_1,	*,	control,	1),	" &			
"398	(BC_1,	*,	control,	1),	" &			
"399	(BC_1,	*,	control,	1),	" &			
"400	(BC_1,	*,	control,	1),	" &			
"401	(BC_1,	*,	control,	1),	" &			
"402	(BC_1,	*,	control,	1),	" &			
"403	(BC_1,	*,	control,	1)";				

24.2.5 TAP Controller

The TAP (Test Access Port) controller is a 16-state synchronous finite state machine that controls the sequence of test logic operations. The TAP can be controlled via a bus master. The bus master can be either automatic test equipment or a component (i.e., PLD) that interfaces to the TAP. The TAP controller changes state only in response to a rising edge of TCK. The value of the test mode state (TMS) input signal at a rising edge of TCK controls the sequence of state changes. The TAP controller is initialized after power-up by applying a low to the **TRST#** pin. In addition, the TAP controller can be initialized by applying a high signal level on the TMS input for a minimum of five TCK periods. See Figure 24-2 for the state diagram of the TAP controller. An uninitialized TAP controller can result in erratic processor behavior even when there is no intention to use the JTAG portion of the processor.

The behavior of the TAP controller and other test logic in each controller state is described in the following subsections. For greater detail on the state machine and the public instructions, refer to the *IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture* document (available from the IEEE).

Figure 24-2. TAP Controller State Diagram



24.2.5.1 Test Logic Reset State

In this state, test logic is disabled to allow normal operation of the 80303 I/O processor. Upon entering the Test_Logic_Reset state, the device identification register is loaded. No matter what the present state of the controller, it enters Test-Logic-Reset state when the TMS input is held high (1_2) for at least five rising edges of TCK. The controller remains in this state while TMS is high. The TAP controller is also forced to enter this state asynchronously by asserting **TRST#**.

When the controller exits the Test-Logic-Reset controller state as a result of an erroneous low signal on the TMS line at the time of a rising edge on TCK (for example, a glitch due to external interference), it returns to the Test-Logic-Reset state following three rising edges of TCK with the TMS line at the intended high logic level.

24.2.5.2 Run-Test/Idle State

The TAP controller enters the Run-Test/Idle state between scan operations. The controller remains in this state as long as TMS is held low. When the **runbist** instruction is selected, it executes during the Run-Test/Idle state and the result is reported in the RUNBIST register. Instructions that do not call functions generate no activity in the test logic while the controller is in this state. The instruction register and all test data registers retain their current state. When TMS is high on the rising edge of TCK, the controller moves to the Select-DR-Scan state. The instruction register does not change while the TAP controller is in this state.

24.2.5.3 Select-DR-Scan State

The Select-DR-Scan state is a transitional controller state. While in the Select-DR-Scan state, the test data registers selected by the current instruction retain their previous states. When TMS is held low on the rising edge of TCK, the controller moves into the Capture-DR state. When TMS is held high on the rising edge of TCK, the controller moves into the Select-IR-Scan state. See Section 24.2.5.10, "Select-IR Scan State" on page 24-19. The instruction register does not change while the TAP controller is in this state.

24.2.5.4 Capture-DR State

In this state, the selected test data register is loaded with its parallel value on the rising edge of TCK. When the controller is in the Capture-DR state and the current instruction is **sample/preload**, the boundary-scan register captures input pin data on the rising edge of TCK. Test data registers that do not have a parallel input are not changed. The boundary-scan registers cannot be updated from the parallel inputs any other way. The instruction register does not change while the TAP controller is in this state.

When TMS is high on the rising edge of TCK, the controller enters the Exit1-DR state. When TMS is low on the rising edge of TCK, the controller enters the Shift-DR state.

24.2.5.5 Shift-DR State

In the Shift-DR state, the test data register selected by the current instruction shifts data one bit position nearer to the TDO serial output on each rising edge of TCK. All other test data registers retain their previous values during this state.

The instruction register does not change while the TAP controller is in this state.

When TMS is high on the rising edge of TCK, the controller enters the Exit1-DR state. When TMS is low on the rising edge of TCK, the controller remains in the Shift-DR state.

24.2.5.6 Exit1-DR State

Exit1-DR is a temporary controller state. When the TAP controller is in the Exit1-DR state and TMS is held high on the rising edge of TCK, the controller enters the Update-DR state, which terminates the scanning process. When TMS is held low on the rising edge of TCK, the controller enters the Pause-DR state.

The instruction register does not change while the TAP controller is in this state. All test data registers selected by the current instruction retain their previous value during this state.

24.2.5.7 Pause-DR State

The Pause-DR state allows the test controller to temporarily halt the shifting of data through the test data register in the serial path between TDI and TDO. The test data register selected by the current instruction retains its previous value during this state. The instruction register does not change in this state.

The controller remains in this state as long as TMS is low. When TMS is high on the rising edge of TCK, the controller moves to the Exit2-DR state.

24.2.5.8 Exit2-DR State

Exit2-DR is a temporary state. When TMS is held high on the rising edge of TCK, the controller enters the Update-DR state, which terminates the scanning process. When TMS is held low on the rising edge of TCK, the controller re-enters the Shift-DR state.

The instruction register does not change while the TAP controller is in this state. All test data registers selected by the current instruction retain their previous value during this state.

24.2.5.9 Update-DR State

The boundary-scan register is provided with a latched parallel output. This output prevents changes at the parallel output while data is shifted in response to the extest, sample/preload instructions. When the boundary-scan register is selected while the TAP controller is in the Update-DR state, data is latched onto the boundary-scan register's parallel output from the shift-register path on the falling edge of TCK. The data held at the latched parallel output does not change unless the controller is in this state.

While the TAP controller is in this state, all of the test data register's shift-register bit positions selected by the current instruction retain their previous values. The instruction register does not change while the TAP controller is in this state.

When the TAP controller is in this state and TMS is held high on the rising edge of TCK, the controller re-enters the Select-DR-Scan state. When TMS is held low on the rising edge of TCK, the controller re-enters the Run-Test/Idle state.

24.2.5.10 Select-IR Scan State

Select-IR is a temporary controller state. The test data registers selected by the current instruction retain their previous states. In this state, when TMS is held low on the rising edge of TCK, the controller enters the Capture-IR state and a scan sequence for the instruction register is initiated. When TMS is held high on the rising edge of TCK, the controller re-enters the Test-Logic-Reset state. The instruction register does not change in this state.

24.2.5.11 Capture-IR State

When the controller is in the Capture-IR state, the shift register contained in the instruction register appends the instruction with the fixed value 01_2 on the rising edge of TCK.

The test data register selected by the current instruction retains its previous value during this state. The instruction does not change in this state. While in this state, holding TMS high on the rising edge of TCK causes the controller to enter the Exit1-IR state. When TMS is held low on the rising edge of TCK, the controller enters the Shift-IR state.

24.2.5.12 Shift-IR State

When the controller is in this state, the shift register contained in the instruction register is connected between TDI and TDO and shifts data one bit position nearer to its serial output on each rising edge of TCK. The test data register selected by the current instruction retains its previous value during this state. The instruction register does not change.

When TMS is held high on the rising edge of TCK, the controller enters the Exit1-IR state. When TMS is held low on the rising edge of TCK, the controller remains in the Shift-IR state.

24.2.5.13 Exit1-IR State

This is a temporary state. When TMS is held high on the rising edge of TCK, the controller enters the Update-IR state, which terminates the scanning process. When TMS is held low on the rising edge of TCK, the controller enters the Pause-IR state.

The test data register selected by the current instruction retains its previous value during this state.

The instruction does not change and the instruction register retains its state.

24.2.5.14 Pause-IR State

The Pause-IR state allows the test controller to temporarily halt the shifting of data through the instruction register. The test data registers selected by the current instruction retain their previous values during this state. The instruction does not change and the instruction register retains its state.

The controller remains in this state as long as TMS is held low. When TMS is high on the rising edges of TCK, the controller enters the Exit2-IR state.

24.2.5.15 Exit2-IR State

This is a temporary state. When TMS is held high on the rising edge of TCK, the controller enters the Update-IR state, which terminates the scanning process. When TMS is held low on the rising edge of TCK, the controller re-enters the Shift-IR state.

This test data register selected by the current instruction retains its previous value during this state. The instruction does not change and the instruction register retains its state.

24.2.5.16 Update-IR State

The instruction shifted into the instruction register is latched onto the parallel output from the shift-register path on the falling edge of TCK. Once latched, the new instruction becomes the current instruction. Test data registers selected by the current instruction retain their previous values.

When TMS is held high on the rising edge of TCK, the controller re-enters the Select-DR-Scan state. When TMS is held low on the rising edge of TCK, the controller re-enters the Run-Test/Idle state.

24.2.6 Boundary-Scan Example

The following example describes two command actions. The example assumes the TAP controller starts in the Test-Logic-Reset state. The TAP controller then loads and executes a new instruction. See Figure 24-3 for an illustration of the waveforms involved in this example. The steps are:

1. Load the **sample/preload** instruction into the instruction register:
   a. Use TMS to select the Shift-IR state. While in the Shift-IR state, shift in the new instruction, least significant byte first.
   b. Use the Shift-IR state four times to read the least- through most-significant instruction bits into the instruction register (one does not care what old instruction is being shifted out of the TDO pin).
   c. Enter the Update-IR state to make the instruction take effect.
2. Capture pin data and shift the data out through the TDO pin:
   a. Use TMS to select the Select-DR-Scan state.
   b. Transition the TAP controller to the Capture-DR state to latch pin data in the boundary-scan register cells.
   c. Enter and stay in the Shift-DR state for 110 TCK cycles. These TDO values are compared against expected data to determine if component operation and connection are correct. Record the TDO values after each cycle. New serial data enters the boundary-scan register through the TDI pin, while old data is scanned out.
   d. Pass through the Exit1-DR state to the Update-DR state. Here boundary-scan data to be driven out of the system output pins is latched and driven.
   e. Transition back to the Select-DR state to begin another iteration.

This example does not use Pause states. These states allow software to pause the JTAG state machine to accommodate slow board-level data paths. The Pause states allow indefinite interruptions in the shifting while the external tester performs other tasks.

The old instruction was *abcd* in the example. The original instruction register value becomes the ID code since the example starts from the reset state. Other times it represents the previous opcode. The new instruction opcode is 0001_2 (**sample/preload**). All pins are captured into the serial boundary-scan register and the values are output to the TDO pin.

The TCK signal at the top of the diagram shows a continuous pulse train. In many designs, however, TCK is more irregular. In such cases, software controls TCK by writing to a port bit. Software writes the TMS and TDI signals and toggles the clock high. Typically, software drives TCK low quickly. The program monitors the TDO pin values as they are shifted out.

Figure 24-3. Example Showing Typical JTAG Operations

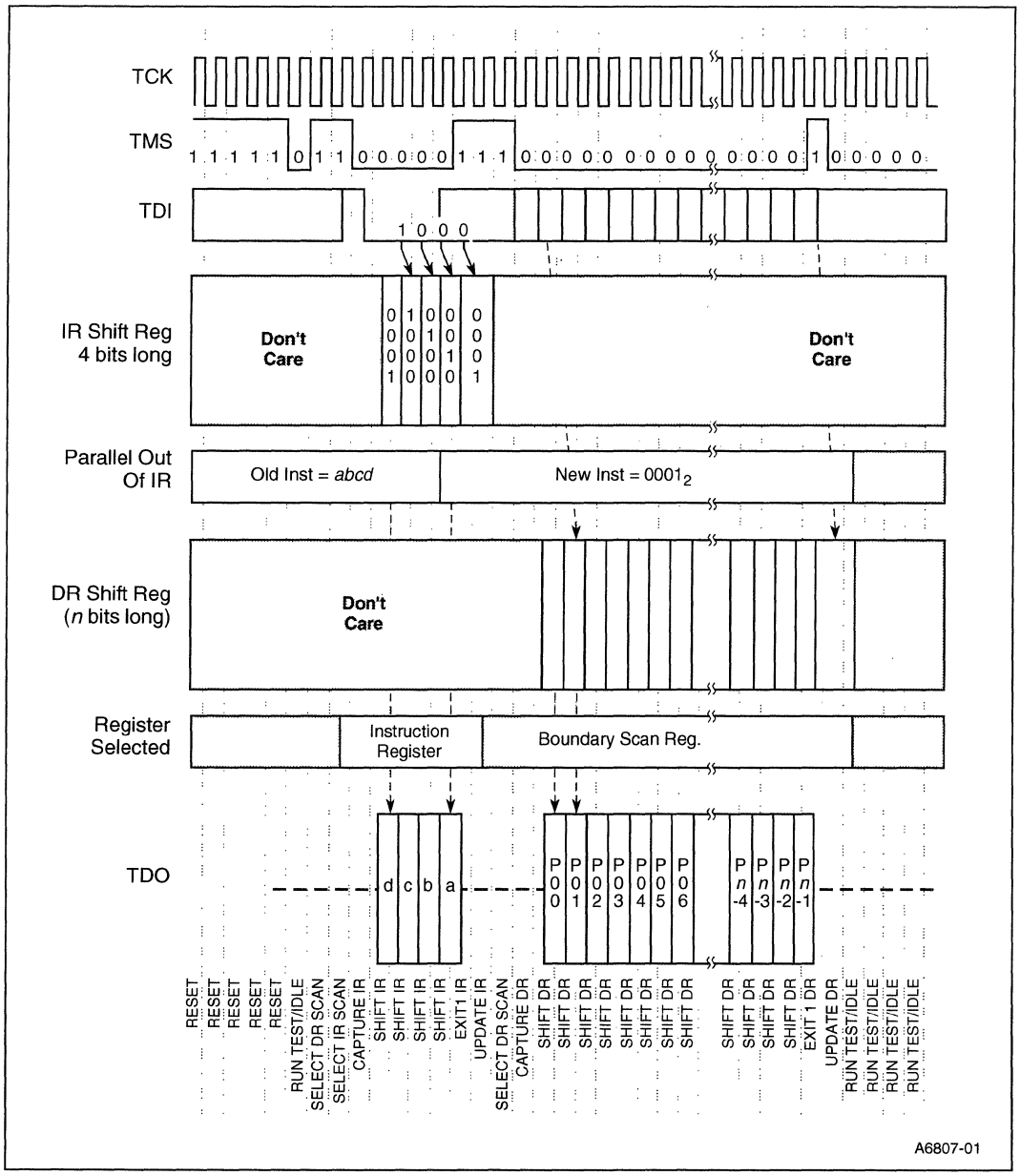


Figure 24-4. Timing Diagram Illustrating the Loading of Instruction Register

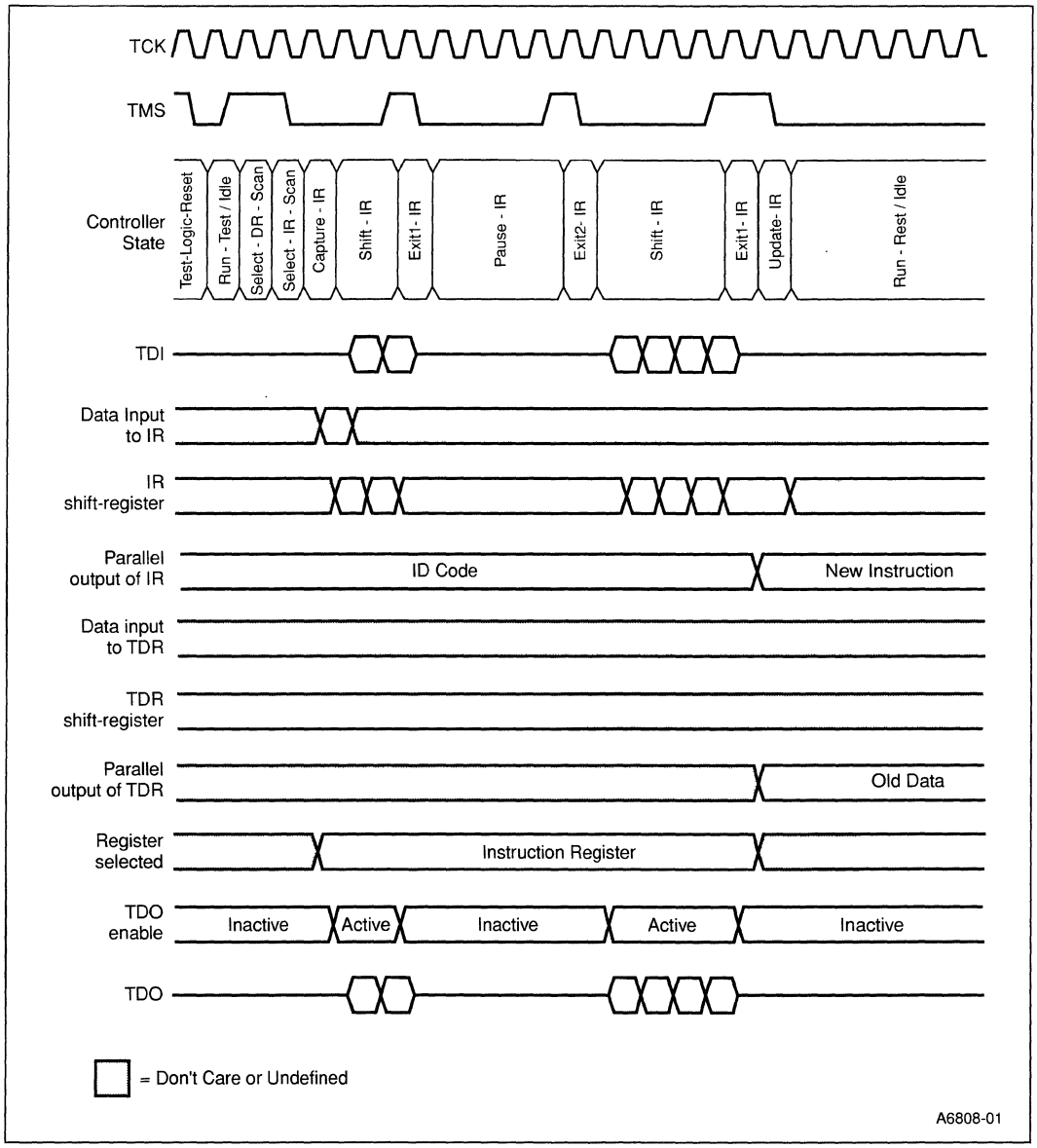
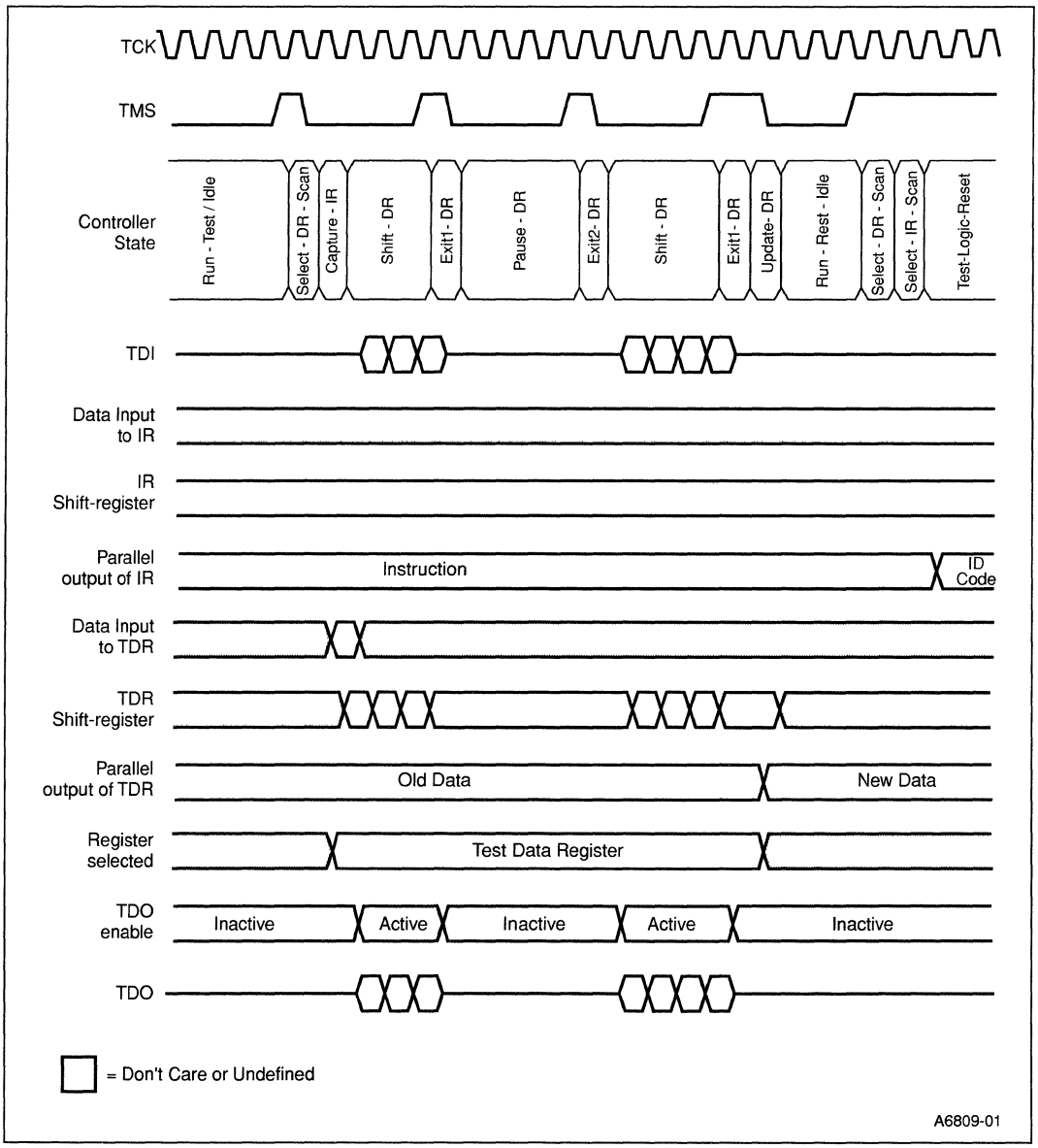


Figure 24-5. Timing Diagram Illustrating the Loading of Data Register



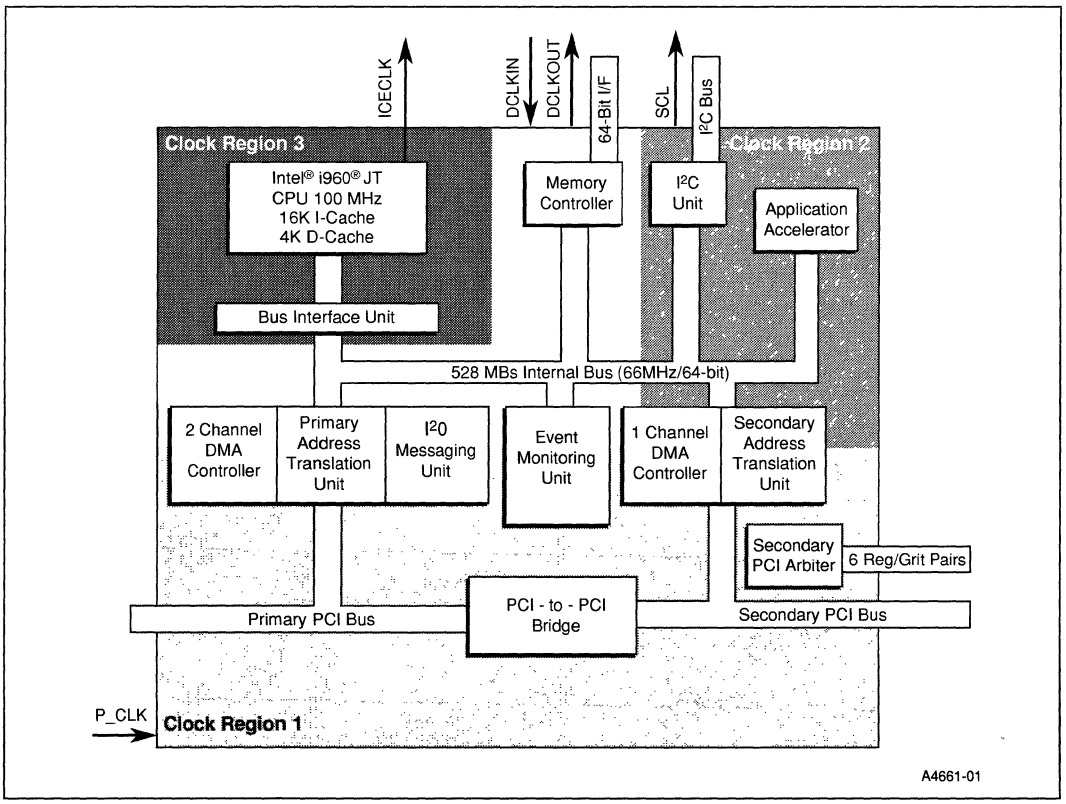
Clocking and Reset 25

This chapter describes the clocking and reset function. The intent of this chapter is to elaborate and clarify descriptions of the clocking and reset mechanisms.

25.1 Clocking Overview

The Intel® 80303 I/O processor contains various clocking boundaries internally. The clocks for all of the units within the 80303 I/O processor are generated from a single input clock. This input feeds the Phase Lock Loop (PLL) circuitry which generates all of the internal clocks. The block diagram of the 80303 I/O processor, shown in Figure 25-1, highlights the four clocking regions.

Figure 25-1. Clocking Regions Diagram



Within each of the clocking regions identified in Figure 25-1, exists various clock requirements for the 80303 I/O processor units and for the output clocks pins provided for the external subsystem.

25.1.1 Clocking Theory of Operation

Each region within the 80303 I/O processor contains different clocking requirements. These requirements are summarized in the following sections.

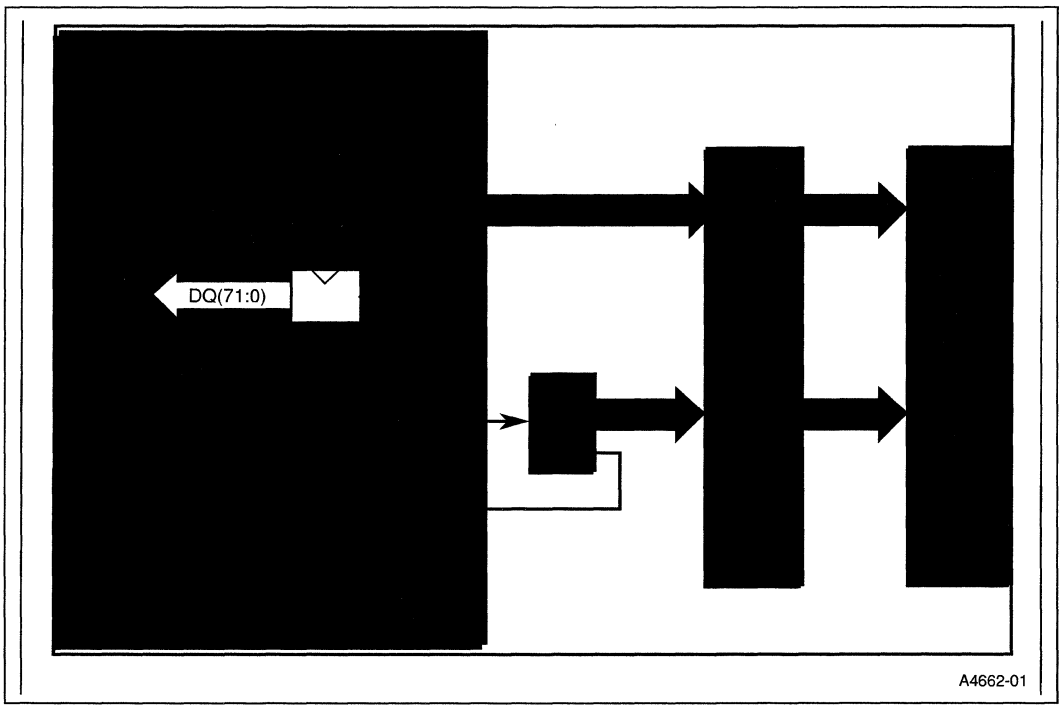
25.1.2 Clocking Region 1

Region 1 contains the main input clock for providing the 80303 I/O processor with all of its clock sources. This input clock is provided by the system designer. This input clock, called the primary PCI bus clock, is connected to the input pin **P_CLK**. The 80303 I/O processor supports an input frequency of 33MHz for normal PCI bus operation on the primary PCI interface. The secondary interface of Region 1 obtains its input clock from the clocking unit specified in clocking Region 1 by **P_CLK**.

25.1.3 Clocking Region 2

Region 2 obtains its input clock from the clocking unit specified in clocking region 1. This region is the internal bus of the 80303 I/O processor. It supports clock frequencies up to a maximum of 66 MHz operation. The clocking unit provides one SDRAM output clock, based on a dedicated PLL. The clocking unit contains one output clock, called **DCLKOUT** and one SDRAM input clock called **DCLKIN**. The **DCLKOUT** output is used by external circuitry (clock buffering) to generate the clocks for the SDRAM memory subsystem. The **DCLKIN** signal is used to skew **DCLKOUT** appropriately to accommodate flight time and clock buffer delays. Refer to Figure 25-2 for a diagram that describes the SDRAM clocking requirements.

Figure 25-2. SDRAM Clocking Diagram



Region 2 also contains an output clock used for the I^2C bus interface (Chapter 22, "I2C Bus Interface Unit"). The output clock frequency for I^2C operation is 100KHz or 400KHz. This clock is generated from internal bus clock. In order to use the I^2C interface, a clock divider value must be written into the I^2C Clock Count Register.

25.1.4 Clocking Region 3

Region 3 obtains its input clock from the clocking unit specified in clocking region 1. This region is the i960 Core Processor and the Bus Interface Unit. It supports clock frequencies up to a maximum of 100 MHz operation. The region 4 clock is a multiple of the **P_CLK**.

25.1.5 Clocking Region Summary

Table 25-1 summarizes all of the input clock pins, output clock pins, and clock strapping option pins used in the 80303 I/O processor.

Table 25-1. Clock Pin Summary

Pin	Input/Output	Description
P_CLK	Input	Primary PCI Input Clock
DCLKIN	Input	SDRAM Input Clock
DCLKOUT	Output	SDRAM Output Clock
SCL	Output	I^2C Output Clock

Table 25-2 summarizes all of the clocks generated to the three regions within the 80303 I/O processor.

Table 25-2. Clock Region Summary

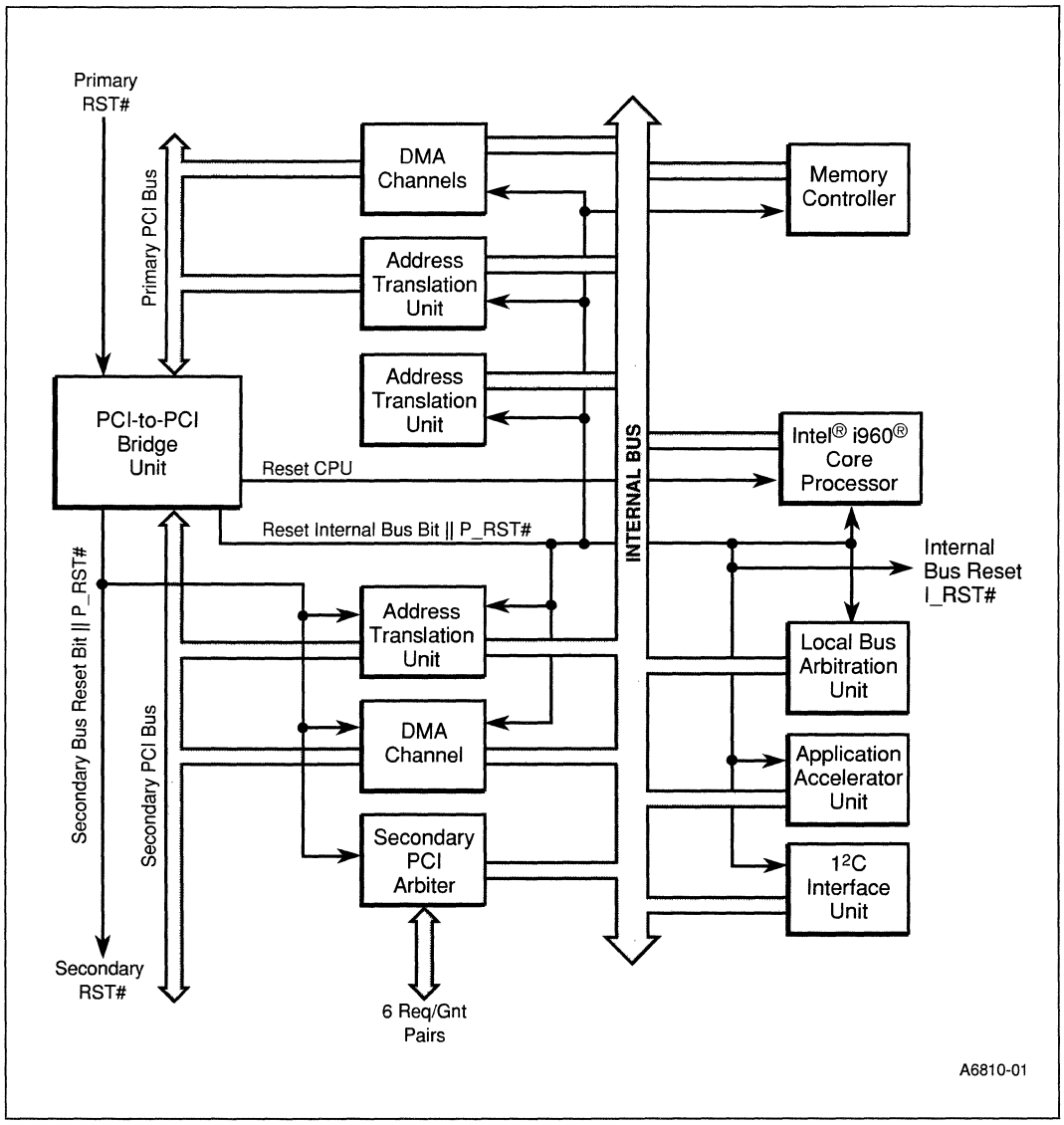
Input Clock	Region/Clock
P_CLK= 33 MHz	Region 1: 1x **P_CLK**
	Region 2: 2x **P_CLK**
	Region 3: 3x **P_CLK**

25.2 Reset Overview

There are three ways to reset the 80303 I/O processor. The main reset is controlled through the primary PCI bus reset signal (P_RST#). When the primary PCI bus asserts this signal, the entire 80303 I/O processor is placed in a reset state. In addition to the primary PCI reset pin, the 80303 I/O processor provides software control of units within the 80303 I/O processor and the secondary PCI interface.

Figure 25-3 shows the logical block diagram of the reset conditions.

Figure 25-3. Reset Block Diagram



When the primary PCI signal (P_RST#) is asserted, the reset signal causes all configuration registers, internal control and enable signals, state machines, and output buffers to their initialized state. The specification is well defined for signal attached to the PCI bus.

25.2.1 Primary PCI Reset

When the primary PCI bus reset signal P_RST# is asserted, the 80303 I/O processor:

- asserts the secondary PCI bus reset signal S_RST#
- resets the Intel® i960® core processor and the internal bus
- resets all internal units
- resets all Memory Mapped Registers
- latches all configuration straps on the rising edge of P_RST#, refer to Section 25.3
- latches **P_REQ64#** to determine the primary PCI bus interface width
- asserts the I_RST# output signal

The assertion and deassertion of the PCI reset signal (P_RST#) is asynchronous with respect to **P_CLK**. The rising edge of the P_RST# signal must be monotonic through the input switching range and must meet the minimum slew rate. The PCI local bus specification defines the assertion of P_RST# for a period of 1 ms after power is stable.

Upon the assertion of P_RST#, all units within the 80303 I/O processor are reset. This reset will reset all internal memory mapped registers (MMRs) to their default configuration state. The reset value for each register is defined within each register description.

Upon the deassertion of P_RST#, the 80303 I/O processor samples a series of strapping pins to set configuration modes (refer to Section 25.3, "Reset Strapping Options" on page 25-7). One strap which alters the behavior of the 80303 I/O processor on the deassertion of P_RST# is the **RST_MODE#** strap. If the **RST_MODE#** pin is asserted on the rising edge of P_RST#, the 80303 I/O processor will continue to assert the individual reset to the i960 core processor. This mode, will hold the i960 core processor in reset until the Core Processor Reset Bit in the Extended Bridge Configuration Register (PCI Bridge) is cleared, thus allowing the i960 core processor to enter its initialization procedure.

The primary PCI interface of the 80303 I/O processor samples the **P_REQ64#** signal to determine if the 80303 I/O processor is connected to a 64-bit data path. The central resource is required to drive the **P_REQ64#** signal low during the time that P_RST# is asserted. The state of **P_REQ64#** on the rising edge of the P_RST# signal notifies the primary ATU, DMA channel 0, DMA channel 1, and the primary interface of the PCI bridge that the 80303 I/O processor is connected to a 64-bit or 32-bit PCI bus.

25.2.2 Secondary PCI Reset

When the secondary PCI bus reset signal S_RST# is asserted, the 80303 I/O processor:

- asserts the secondary PCI bus reset signal S_RST#
- resets the SATU
- resets DMA channel 2
- resets all Memory Mapped Registers in the SATU and DMA2
- latches **S_REQ64#** to determine the secondary PCI bus interface width

Upon the assertion of P_RST#, the 80303 I/O processor asserts the secondary PCI reset output (S_RST#). S_RST# remains asserted for the same period as P_RST#. The secondary PCI arbiter is connected to the S_RST#. As with the primary PCI interface, the secondary PCI interface is required to sample **S_REQ64#** on the rising edge of S_RST# to determine whether the 80303 I/O processor is connected to a 64-bit or a 32-bit wide PCI bus. Since the secondary PCI arbiter is integrated into the 80303 I/O processor, the secondary arbiter is required to drive **S_REQ64#** on the rising edge of S_RST# based on the strapping option pin **32BITPCI_EN#**. Refer to Chapter 17, "Intel® 80303 I/O Processor Arbitration" for additional information.

Secondary PCI reset is also available through the Bridge Control register (BCR) in the PCI to PCI Bridge Unit. The secondary PCI reset unit contains sideband signals from and to the SATU and DMA2. These sideband signals are used to ensure a graceful completion of these units on the internal bus during the secondary PCI reset.

25.2.3 Internal Bus Reset

The Reset Internal Bus bit in the Extended Bridge Control Register resets the i960 core processor and all units on the internal bus. Before resetting, the DMA channels and the ATUs shall gracefully halt all PCI bus transactions. It is the responsibility of the software to ensure that the I^2C bus is idle before the reset occurs. The i960 core processor may or may not be held in reset when the Reset Local Bus bit is cleared by software. This depends on the default value of the Core Processor Reset bit in the EBCR. The Local Bus Reset does not reset the PCI to PCI Bridge Unit or its configuration registers.

When the reset local bus bit in the Extended Bridge Control Register is set, there are sideband signals notifying the BIU, PATU, SATU, and the DMAs that a reset is coming.

25.3 Reset Strapping Options

There are many initialization modes that can be selected when the processor is reset. Table 25-3 shows the configuration modes. All of the configuration modes defined are determined on the rising edge of P_RST#.

Table 25-3. Configuration Modes

NAME	DESCRIPTION
RAD[4]/STEST	**SELF TEST** enables or disables the processor's internal self-test feature at initialization. **STEST** is examined at the end of P_RST#.
RAD[3]/RETRY	**RETRY** is sampled at the end of P_RST# to determine if the Primary PCI interface will be disabled.
RAD[6]/RST_MODE#	**RESET MODE** is sampled at the end of P_RST# to determine if the 80303 I/O processor is to be held in reset.
RAD[1]/32BITPCI_EN#	**32-BIT Secondary PCI Enable** is sampled at the end of P_RST# to notify the secondary PCI arbiter if the 64-bit protocol is enabled on the secondary PCI bus.
RAD[2]/32BITMEM_EN#	**32-BIT MemoryEnable** is sampled at the end of P_RST# to notify the memory controller if 32-bit wide SDRAM memories are connected to the memory controller.
ONCE#	**ONCE MODE**: is sampled during reset to stop all clocks and float all output pins of the i960® core processor except the **TDO** pin.

Machine-Level Instruction Formats A

This appendix describes the encoding format for instructions used by the Intel® i960® processors. Included is a description of the four instruction formats and how the addressing modes relate to these formats. Refer also to Appendix B, "Opcodes and Execution Times".

A.1 General Instruction Format

The Intel® i960® architecture defines four basic instruction encoding formats: REG, COBR, CTRL and MEM (Figure A-1). Each instruction uses one of these formats, which is defined by the instruction's opcode field. All instructions are one word long and begin on word boundaries. MEM format instructions are encoded in one of two sub-formats: MEMA or MEMB. MEMB supports an optional second word to hold a displacement value. The following sections describe each format's instruction word fields.

Figure A-1. Instruction Formats

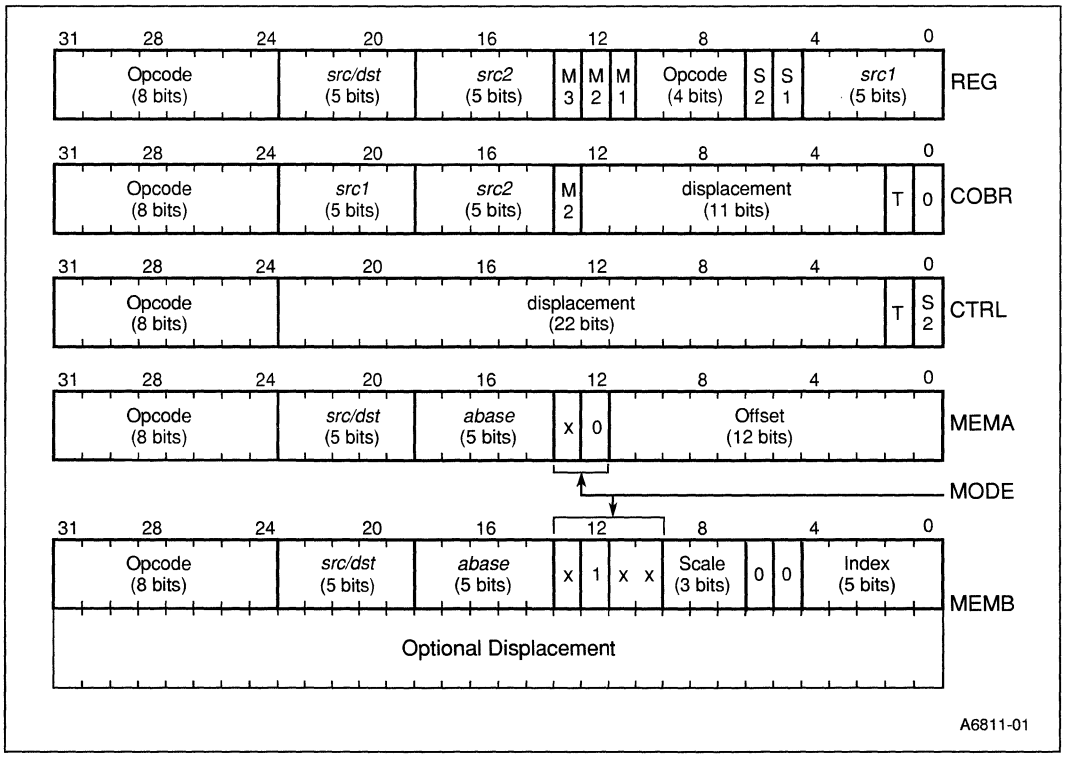


Table A-1. Instruction Field Descriptions

Instruction Field	Description
Opcode	The opcode of the instruction. Opcode encodings are defined in Section 6.1.8, "Opcode and Instruction Format" on page 6-5.
src1	An input to the instruction. This field specifies a value or address. In one case of the COBR format, this field is used to specify a register in which a result is stored.
src2	An input to the instruction. This field specifies a value or address.
src/dst	Depending on the instruction, this field can be (1) an input value or address, (2) the register where the result is stored, or (3) both of the above.
abase	A register whose value is used in computing a memory address.
INDEX	A register whose value is used in computing a memory address.
DISPLACEMENT	A signed two's complement number.
Offset	An unsigned positive number.
Optional Displacement	A signed two's complement number used in the two-word MEMB format.
MODE	A specification of how a memory address for an operand is computed and, for MEMB, specifies whether the instruction contains a second word to be used as a displacement.
SCALE	A specification of how a register's contents are multiplied for certain addressing modes (i.e., for indexing).
M1, M2, M3	These fields further define the meaning of the *src1*, *src2*, and *src/dst* fields respectively as shown in Table A-3.

When a particular instruction is defined as not using a particular field, the field is ignored.

A.2 REG Format

REG format is used for operations performed on data contained in registers. Most of the i960 processor family's instructions use this format.

The opcode for the REG instructions is 12 bits long (three hexadecimal digits) and is split between bits 7 through 10 and bits 24 through 31. For example, the **addi** opcode is 591H. Here, bits 24 through 31 contain 59H and bits 7 through 10 contain 1H.

src1 and *src2* fields specify the instruction's source operands. Operands can be global or local registers or literals. Mode bits (M1 for *src1* and M2 for *src2*) and the instruction type determine what an operand specifies. Table A-3 shows this relationship.

Table A-2. Encoding of *src1* and *src2* in REG Format

M1 or M2	Src1 or Src2 Operand Value	Register Number	Literal Value
0	00000 ... 01111	r0 ... r15	NA
	10000 ... 11111	g0 ... g15	NA
1	00000 ... 11111	NA	0 ... 31

The *src/dst* field can specify a source operand, a destination operand or both, depending on the instruction. Here again, mode bit M3 determines how this field is used. If M3 is clear, the *src/dst* operand is a global or local register that is encoded as shown in Table A-3. If M3 is set, the *src/dst* operand can be used as a source-only operand that is a literal.

When a literal is specified, it is always an unsigned 5-bit value that is zero-extended to a 32-bit value and used as the operand. When the instruction defines an operand to be larger than 32 bits, values specified by literals are zero-extended to the operand size.

Table A-3. Encoding of *src/dst* in REG Format

M3	src/dst	src Only	dst Only
0	g0 ... g15 r0 ... r15	g0 ... g15 r0 ... r15	g0 ... g15 r0 ... r15
1	Reserved	Reserved	reserved

A.3 COBR Format

The COBR format is used primarily for compare-and-branch instructions. The test-if instructions also use the COBR format. The COBR opcode field is eight bits (two hexadecimal digits).

The *src1* and *src2* fields specify source operands for the instruction. The *src1* field can specify either a global or local register or a literal as determined by mode bit M1. The src2 field can only specify a global or local register. Table A-4 shows the M1, *src1* relationship and Table A-5 shows the S2, *src2* relationship.

Table A-4. Encoding of *src1* in COBR Format

M1	src1
0	g0 ... g15 r0 ... r15
1	Literal

Table A-5. Encoding of *src2* in COBR Format

S2	src2
0	g0 ... g15 r0 ... r15
1	reserved

The *displacement* field contains a signed two's complement number that specifies a word displacement. The processor uses this value to compute the address of a target instruction to which the processor branches as a result of the comparison. The displacement field's value can range from -2^{10} to 2^{10} -1. To determine the target instruction's IP, the processor converts the displacement value to a byte displacement (i.e., multiplies the value by 4). It then adds the resulting byte displacement to the IP of the current instruction.

A.4 CTRL Format

The CTRL format is used for instructions that branch to a new IP, including the **BRANCH<cc>**, **bal** and **call** instructions. Note that **balx**, **bx** and **callx** do not use this format. **ret** also uses the CTRL format. The CTRL opcode field is eight bits (two hexadecimal digits).

A branch target address is specified with the displacement field in the same manner as COBR format instructions. The displacement field specifies a word displacement as a signed, two's complement number in the range -2^{21} to 2^{21}-1. The processor ignores the **ret** instruction's displacement field.

A.5 MEM Format

The MEM format is used for instructions that require a memory address to be computed. These instructions include the **LOAD**, **STORE** and **lda** instructions. Also, the extended versions of the branch, branch-and-link and call instructions (**bx**, **balx** and **callx**) use this format.

The two MEM-format encodings are MEMA and MEMB. MEMB can optionally add a 32-bit displacement (contained in a second word) to the instruction. Bit 12 of the instruction's first word determines whether MEMA (clear) or MEMB (set) is used.

The opcode field is eight bits long for either encoding. The *src/dst* field specifies a global or local register. For load instructions, *src/dst* specifies the destination register for a word loaded into the processor from memory or, for operands larger than one word, the first of successive destination registers. For store instructions, this field specifies the register or group of registers that contain the source operand to be stored in memory.

The mode field determines the address mode used for the instruction. Table A-6 summarizes the addressing modes for the two MEM-format encodings. Fields used in these addressing modes are described in the following sections.

Table A-6. Addressing Modes for MEM Format Instructions

Format	MODE	Addressing Mode	Address Computation	# of Instr Words
MEMA	00	Absolute Offset	offset	1
	10	Register Indirect with Offset	(abase) + offset	1
MEMB	0100	Register Indirect	(abase)	1
	0101	IP with Displacement	(IP) + displacement + 8	2
	0110	Reserved	reserved	NA
	0111	Register Indirect with Index	(abase) + (index) * 2^{scale}	1
	1100	Absolute Displacement	displacement	2
	1101	Register Indirect with Displacement	(abase) + displacement	2
	1110	Index with Displacement	(index) * 2^{scale} + displacement	2
	1111	Register Indirect with Index and Displacement	(abase) + (index) * 2^{scale} + displacement	2

NOTES:
1. In these address computations, a field in parentheses indicates that the value in the specified register is used in the computation.
2. Usage of a reserved encoding may cause generation of an OPERATION.INVALID_OPCODE fault.

A.5.1 MEMA Format Addressing

The MEMA format provides two addressing modes:

- Absolute offset
- Register indirect with offset

The *offset* field specifies an unsigned byte offset from 0 to 4096. The abase field specifies a global or local register that contains an address in memory.

For the absolute-offset addressing mode (MODE = 00), the processor interprets the *offset* field as an offset from byte 0 of the current process address space; the abase field is ignored. Using this addressing mode along with the **lda** instruction allows a constant in the range 0 to 4096 to be loaded into a register.

For the register-indirect-with-offset addressing mode (MODE = 10), *offset* field value is added to the address in the abase register. Clearing the offset value creates a register indirect addressing mode; however, this operation can generally be carried out faster by using the MEMB version of this addressing mode.

A.5.2 MEMB Format Addressing

The MEMB format provides the following seven addressing modes:

- absolute displacement
- register indirect
- register indirect with displacement
- register indirect with displacement
- register indirect with index and displacement
- index with displacement
- IP with displacement

The abase and index fields specify local or global registers, the contents of which are used in address computation. When the index field is used in an addressing mode, the processor automatically scales the index register value by the amount specified in the SCALE field. Table A-7 gives the encoding of the scale field. The optional displacement field is contained in the word following the instruction word. The displacement is a 32-bit signed two's complement value.

Table A-7. Encoding of Scale Field

Scale	Scale Factor (Multiplier)
000	1
001	2
010	4
011	8
100	16
101 to 111	Reserved

NOTE: Usage of a reserved encoding causes an unpredictable result.

For the IP with displacement mode, the value of the displacement field plus eight is added to the address of the current instruction.

intel®

Opcodes and Execution Times B

B.1 Instruction Reference by Opcode

This section lists the instruction encoding for each Intel® 80303 I/O processor instruction. Instructions are grouped by instruction format and listed by opcode within each format.

Table B-8. Miscellaneous Instruction Encoding Bits

M3	M2	M1	S2	S1	T	Description
REG Format						
x	x	0	x	0	—	*src1* is a global or local register
x	x	1	x	0	—	*src1* is a literal
x	x	0	x	1	—	reserved
x	x	1	x	1	—	reserved
x	0	x	0	x	—	*src2* is a global or local register
x	1	x	0	x	—	*src2* is a literal
x	0	x	1	x	—	reserved
x	1	x	1	x	—	reserved
0	x	x	x	x	—	*src/dst* is a global or local register
1	x	x	x	x	—	*src/dst* is a literal when used as a source. M3 may not be 1 when *src/dst* is used as a destination only or is used both as a source and destination in an instruction (**atmod**, **modify**, **extract**, **modpc**).
COBR Format						
—	—	0	0	—	x	*src1*, *src2* and *dst* are global or local registers
—	—	1	0	—	x	*src1* is a literal, *src2* and *dst* are global or local registers
—	—	0	1	—	x	reserved
—	—	1	1	—	x0	reserved

Table B-9. REG Format Instruction Encodings (Sheet 1 of 4)

Opcode	Mnemonic	Cycles to Execute	Opcode (11-4)	src/dst	src2	Mode			Opcode (3-0)	Special Flags		src1
			[31,24]	[23,19]	[18,14]	13	12	11	[10,7]	6	5	[4,0]
58:0	**notbit**	1	0101 1000	dst	src	M3	M2	M1	0000	S2	S1	bitpos
58:1	**and**	1	0101 1000	dst	src2	M3	M2	M1	0001	S2	S1	src1
58:2	**andnot**	1	0101 1000	dst	src2	M3	M2	M1	0010	S2	S1	src1
58:3	**setbit**	1	0101 1000	dst	src	M3	M2	M1	0011	S2	S1	bitpos
58:4	**notand**	1	0101 1000	dst	src2	M3	M2	M1	0100	S2	S1	src1
58:6	**xor**	1	0101 1000	dst	src2	M3	M2	M1	0110	S2	S1	src1
58:7	**or**	1	0101 1000	dst	src2	M3	M2	M1	0111	S2	S1	src1
58:8	**nor**	1	0101 1000	dst	src2	M3	M2	M1	1000	S2	S1	src1
58:9	**xnor**	1	0101 1000	dst	src2	M3	M2	M1	1001	S2	S1	src1
58:A	**not**	1	0101 1000	dst		M3	M2	M1	1010	S2	S1	src
58:B	**ornot**	1	0101 1000	dst	src2	M3	M2	M1	1011	S2	S1	src1
58:C	**clrbit**	1	0101 1000	dst	src	M3	M2	M1	1100	S2	S1	bitpos
58:D	**notor**	1	0101 1000	dst	src2	M3	M2	M1	1101	S2	S1	src1
58:E	**nand**	1	0101 1000	dst	src2	M3	M2	M1	1110	S2	S1	src1
58:F	**alterbit**	1	0101 1000	dst	src	M3	M2	M1	1111	S2	S1	bitpos
59:0	**addo**	1	0101 1001	dst	src2	M3	M2	M1	0000	S2	S1	src1
59:1	**addi**	1	0101 1001	dst	src2	M3	M2	M1	0001	S2	S1	src1
59:2	**subo**	1	0101 1001	dst	src2	M3	M2	M1	0010	S2	S1	src1
59:3	**subi**	1	0101 1001	dst	src2	M3	M2	M1	0011	S2	S1	src1
59:4	**cmpob**	1	0101 1001		src2	M3	M2	M1	0100	S2	S1	src1
59:5	**cmpib**	1	0101 1001		src2	M3	M2	M1	0101	S2	S1	src1
59:6	**cmpos**	1	0101 1001		src2	M3	M2	M1	0110	S2	S1	src1
59:7	**cmpis**	1	0101 1001		src2	M3	M2	M1	0111	S2	S1	src1
59:8	**shro**	1	0101 1001	dst	src	M3	M2	M1	1000	S2	S1	len
59:A	**shrdi**	6	0101 1001	dst	src	M3	M2	M1	1010	S2	S1	len
59:B	**shri**	1	0101 1001	dst	src	M3	M2	M1	1011	S2	S1	len
59:C	**shlo**	1	0101 1001	dst	src	M3	M2	M1	1100	S2	S1	len
59:D	**rotate**	1	0101 1001	dst	src	M3	M2	M1	1101	S2	S1	len
59:E	**shli**	1	0101 1001	dst	src	M3	M2	M1	1110	S2	S1	len
5A:0	**cmpo**	1	0101 1010		src2	M3	M2	M1	0000	S2	S1	src1
5A:1	**cmpi**	1	0101 1010		src2	M3	M2	M1	0001	S2	S1	src1
5A:2	**concmpo**	1	0101 1010		src2	M3	M2	M1	0010	S2	S1	src1
5A:3	**concmpi**	1	0101 1010		src2	M3	M2	M1	0011	S2	S1	src1
5A:4	**cmpinco**	1	0101 1010	dst	src2	M3	M2	M1	0100	S2	S1	src1

1. Execution time based on function performed by instruction.

Table B-9. REG Format Instruction Encodings (Sheet 2 of 4)

Opcode	Mnemonic	Cycles to Execute	Opcode (11-4)	src/dst	src2	Mode			Opcode (3-0)	Special Flags		src1
5A:5	**cmpinci**	1	0101 1010	dst	src2	M3	M2	M1	0101	S2	S1	src1
5A:6	**cmpdeco**	1	0101 1010	dst	src2	M3	M2	M1	0110	S2	S1	src1
5A:7	**cmpdeci**	1	0101 1010	dst	src2	M3	M2	M1	0111	S2	S1	src1
5A:C	**scanbyte**	1	0101 1010		src2	M3	M2	M1	1100	S2	S1	src1
5A:D	**bswap**	10	0101 1010	dst		M3	M2	M1	1101	S2	S1	src1
5A:E	**chkbit**	1	0101 1010		src	M3	M2	M1	1110	S2	S1	bitpos
5B:0	**addc**	1	0101 1011	dst	src2	M3	M2	M1	0000	S2	S1	src1
5B:2	**subc**	1	0101 1011	dst	src2	M3	M2	M1	0010	S2	S1	src1
5B:4	**intdis**	4	0101 1011			M3	M2	M1	0100	S2	S1	
5B:5	**inten**	4	0101 1011			M3	M2	M1	0101	S2	S1	
5C:C	**mov**	1	0101 1100	dst		M3	M2	M1	1100	S2	S1	src
5D:8	**eshro**	11	0101 1101	dst	src2	M3	M2	M1	1000	S2	S1	src1
5D:C	**movl**	4	0101 1101	dst		M3	M2	M1	1100	S2	S1	src
5E:C	**movt**	5	0101 1110	dst		M3	M2	M1	1100	S2	S1	src
5F:C	**movq**	6	0101 1111	dst		M3	M2	M1	1100	S2	S1	src
61:0	**atmod**	24	0110 0010	dst	src2	M3	M2	M1	0000	S2	S1	src1
61:2	**atadd**	24	0110 0010	dst	src2	M3	M2	M1	0010	S2	S1	src1
64:0	**spanbit**	6	0110 0100	dst		M3	M2	M1	0000	S2	S1	src
64:1	**scanbit**	5	0110 0100	dst		M3	M2	M1	0001	S2	S1	src
64:5	**modac**	10	0110 0100	mask	src	M3	M2	M1	0101	S2	S1	dst
65:0	**modify**	6	0110 0101	src/dst	src	M3	M2	M1	0000	S2	S1	mask
65:1	**extract**	7	0110 0101	src/dst	len	M3	M2	M1	0001	S2	S1	bitpos
65:4	**modtc**	10	0110 0101	mask	src	M3	M2	M1	0100	S2	S1	dst
65:5	**modpc**	17	0110 0101	src/dst	mask	M3	M2	M1	0101	S2	S1	src
65:8	**intctl**	12-16	0110 0101	dst		M3	M2	M1	1000	S2	S1	src1
65:9	**sysctl**	10-100[1]	0110 0101	src/dst	src2	M3	M2	M1	1001	S2	S1	src1
65:B	**icctl**	10-100[1]	0110 0101	src/dst	src2	M3	M2	M1	1011	S2	S1	src1
65:C	**dcctl**	10-100[1]	0110 0101	src/dst	src2	M3	M2	M1	1100	S2	S1	src1
65:D	**halt**	•	0110 0101			M3	M2	M1	1101	S2	S1	src1
66:0	**calls**	30	0110 0110			M3	M2	M1	0000	S2	S1	src
66:B	**mark**	8	0110 0110			M3	M2	M1	1011	S2	S1	
66:C	**fmark**	8	0110 0110			M3	M2	M1	1100	S2	S1	
66:D	**flushreg**	15	0110 0110			M3	M2	M1	1101	S2	S1	
66:F	**syncf**	4	0110 0110			M3	M2	M1	1111	S2	S1	
67:0	**emul**	7	0110 0111	dst	src2	M3	M2	M1	0000	S2	S1	src1

1. Execution time based on function performed by instruction.

Table B-9. REG Format Instruction Encodings (Sheet 3 of 4)

Opcode	Mnemonic	Cycles to Execute	Opcode (11-4)	src/dst	src2	Mode			Opcode (3-0)	Special Flags		src1
67:1	**ediv**	40	0110 0111	dst	src2	M3	M2	M1	0001	S2	S1	src1
70:1	**mulo**	2-4	0111 0000	dst	src2	M3	M2	M1	0001	S2	S1	src1
70:8	**remo**	40	0111 0000	dst	src2	M3	M2	M1	1000	S2	S1	src1
70:B	**divo**	40	0111 0000	dst	src2	M3	M2	M1	1011	S2	S1	src1
74:1	**muli**	2-4	0111 0100	dst	src2	M3	M2	M1	0001	S2	S1	src1
74:8	**remi**	40	0111 0100	dst	src2	M3	M2	M1	1000	S2	S1	src1
74:9	**modi**	40	0111 0100	dst	src2	M3	M2	M1	1001	S2	S1	src1
74:B	**divi**	40	0111 0100	dst	src2	M3	M2	M1	1011	S2	S1	src1
78:0	**addono**	1	0111 1000	dst	src2	M3	M2	M1	0000	S2	S1	src1
78:1	**addino**	1	0111 1000	dst	src2	M3	M2	M1	0001	S2	S1	src1
78:2	**subono**	1	0111 1000	dst	src2	M3	M2	M1	0010	S2	S1	src1
78:3	**subino**	1	0111 1000	dst	src2	M3	M2	M1	0011	S2	S1	src1
78:4	**selno**	1	0111 1000	dst	src2	M3	M2	M1	0100	S2	S1	src1
79:0	**addog**	1	0111 1001	dst	src2	M3	M2	M1	0000	S2	S1	src1
79:1	**addig**	1	0111 1001	dst	src2	M3	M2	M1	0001	S2	S1	src1
79:2	**subog**	1	0111 1001	dst	src2	M3	M2	M1	0010	S2	S1	src1
79:3	**subig**	1	0111 1001	dst	src2	M3	M2	M1	0011	S2	S1	src1
79:4	**selg**	1	0111 1001	dst	src2	M3	M2	M1	0100	S2	S1	src1
7A:0	**addoe**	1	0111 1010	dst	src2	M3	M2	M1	0000	S2	S1	src1
7A:1	**addie**	1	0111 1010	dst	src2	M3	M2	M1	0001	S2	S1	src1
7A:2	**suboe**	1	0111 1010	dst	src2	M3	M2	M1	0010	S2	S1	src1
7A:3	**subie**	1	0111 1010	dst	src2	M3	M2	M1	0011	S2	S1	src1
7A:4	**sele**	1	0111 1010	dst	src2	M3	M2	M1	0100	S2	S1	src1
7B:0	**addoge**	1	0111 1011	dst	src2	M3	M2	M1	0000	S2	S1	src1
7B:1	**addige**	1	0111 1011	dst	src2	M3	M2	M1	0001	S2	S1	src1
7B:2	**suboge**	1	0111 1011	dst	src2	M3	M2	M1	0010	S2	S1	src1
7B:3	**subige**	1	0111 1011	dst	src2	M3	M2	M1	0011	S2	S1	src1
7B:4	**selge**	1	0111 1011	dst	src2	M3	M2	M1	0100	S2	S1	src1
7C:0	**addol**	1	0111 1100	dst	src2	M3	M2	M1	0000	S2	S1	src1
7C:1	**addil**	1	0111 1100	dst	src2	M3	M2	M1	0001	S2	S1	src1
7C:2	**subol**	1	0111 1100	dst	src2	M3	M2	M1	0010	S2	S1	src1
7C:3	**subil**	1	0111 1100	dst	src2	M3	M2	M1	0011	S2	S1	src1
7C:4	**sell**	1	0111 1100	dst	src2	M3	M2	M1	0100	S2	S1	src1
7D:0	**addone**	1	0111 1101	dst	src2	M3	M2	M1	0000	S2	S1	src1
7D:1	**addine**	1	0111 1101	dst	src2	M3	M2	M1	0001	S2	S1	src1

1. Execution time based on function performed by instruction.

Table B-9. REG Format Instruction Encodings (Sheet 4 of 4)

Opcode	Mnemonic	Cycles to Execute	Opcode (11-4)	src/dst	src2	Mode			Opcode (3-0)	Special Flags		src1
7D:2	**subone**	1	0111 1101	dst	src2	M3	M2	M1	0010	S2	S1	src1
7D:3	**subine**	1	0111 1101	dst	src2	M3	M2	M1	0011	S2	S1	src1
7D:4	**selne**	1	0111 1101	dst	src2	M3	M2	M1	0100	S2	S1	src1
7E:0	**addole**	1	0111 1110	dst	src2	M3	M2	M1	0000	S2	S1	src1
7E:1	**addile**	1	0111 1110	dst	src2	M3	M2	M1	0001	S2	S1	src1
7E:2	**subole**	1	0111 1110	dst	src2	M3	M2	M1	0010	S2	S1	src1
7E:3	**subile**	1	0111 1110	dst	src2	M3	M2	M1	0011	S2	S1	src1
7E:4	**selle**	1	0111 1110	dst	src2	M3	M2	M1	0100	S2	S1	src1
7F:0	**addoo**	1	0111 1111	dst	src2	M3	M2	M1	0000	S2	S1	src1
7F:1	**addio**	1	0111 1111	dst	src2	M3	M2	M1	0001	S2	S1	src1
7F:2	**suboo**	1	0111 1111	dst	src2	M3	M2	M1	0010	S2	S1	src1
7F:3	**subio**	1	0111 1111	dst	src2	M3	M2	M1	0011	S2	S1	src1
7F:4	**sello**	1	0111 1111	dst	src2	M3	M2	M1	0100	S2	S1	src1

1. Execution time based on function performed by instruction.

Table B-10. COBR Format Instruction Encodings

Opcode	Mnemonic	Cycles to Execute	Opcode	src1	src2	M	Displacement	T	S2
			[31,24]	[23,19]	[18,14]	13	[12,2]	1	0
20	**testno**	4	0010 0000	dst		M1		T	S2
21	**testg**	4	0010 0001	dst		M1		T	S2
22	**teste**	4	0010 0010	dst		M1		T	S2
23	**testge**	4	0010 0011	dst		M1		T	S2
24	**testl**	4	0010 0100	dst		M1		T	S2
25	**testne**	4	0010 0101	dst		M1		T	S2
26	**testle**	4	0010 0110	dst		M1		T	S2
27	**testo**	4	0010 0111	dst		M1		T	S2
30	**bbc**	2 + 1[1]	0011 0000	bitpos	src	M1	targ	T	S2
31	**cmpobg**	2 + 1	0011 0001	src1	src2	M1	targ	T	S2
32	**cmpobe**	2 + 1	0011 0010	src1	src2	M1	targ	T	S2
33	**cmpobge**	2 + 1	0011 0011	src1	src2	M1	targ	T	S2
34	**cmpobl**	2 + 1	0011 0100	src1	src2	M1	targ	T	S2
35	**cmpobne**	2 + 1	0011 0101	src1	src2	M1	targ	T	S2
36	**cmpoble**	2 + 1	0011 0110	src1	src2	M1	targ	T	S2
37	**bbs**	2 + 1	0011 0111	bitpos	src	M1	targ	T	S2
38	**cmpibno**	2 + 1	0011 1000	src1	src2	M1	targ	T	S2
39	**cmpibg**	2 + 1	0011 1001	src1	src2	M1	targ	T	S2
3A	**cmpibe**	2 + 1	0011 1010	src1	src2	M1	targ	T	S2
3B	**cmpibge**	2 + 1	0011 1011	src1	src2	M1	targ	T	S2
3C	**cmpibl**	2 + 1	0011 1100	src1	src2	M1	targ	T	S2
3D	**cmpibne**	2 + 1	0011 1101	src1	src2	M1	targ	T	S2
3E	**cmpible**	2 + 1	0011 1110	src1	src2	M1	targ	T	S2
3F	**cmpibo**	2 + 1	0011 1111	src1	src2	M1	targ	T	S2

1. Indicates that 2 cycles are required to execute the instruction plus an additional cycle to fetch the T_A get instruction when the branch is taken.

Table B-11. CTRL Format Instruction Encodings

Opcode	Mnemonic	Cycles to Execute	Opcode	Displacement	T	0
			[31,24]	[23,2]	1	0
08	**b**	1 + 1[1]	0000 1000	targ	T	0
09	**call**	7	0000 1001	targ	T	0
0A	**ret**	6	0000 1010		T	0
0B	**bal**	1 + 1	0000 1011	targ	T	0
10	**bno**	1 + 1	0001 0000	targ	T	0
11	**bg**	1 + 1	0001 0001	targ	T	0
12	**be**	1 + 1	0001 0010	targ	T	0
13	**bge**	1 + 1	0001 0011	targ	T	0
14	**bl**	1 + 1	0001 0100	targ	T	0
15	**bne**	1 + 1	0001 0101	targ	T	0
16	**ble**	1 + 1	0001 0110	targ	T	0
17	**bo**	1 + 1	0001 0111	targ	T	0
18	**faultno**	13	0001 1000		T	0
19	**faultg**	13	0001 1001		T	0
1A	**faulte**	13	0001 1010		T	0
1B	**faultge**	13	0001 1011		T	0
1C	**faultl**	13	0001 1100		T	0
1D	**faultne**	13	0001 1101		T	0
1E	**faultle**	13	0001 1110		T	0
1F	**faulto**	13	0001 1111		T	0

1. Indicates that 2 cycles are required to execute the instruction plus an additional cycle to fetch the target instruction when the branch is taken.

Table B-12. Cycle Counts for sysctl Operations

Operation	Cycles to Execute
Post Interrupt	20
Purge I-cache	19
Enable I-cache	20
Disable I-cache	22
Software Reset	329+bus
Load Control Register Group	26
Request Breakpoint Resource	21-22

Table B-13. Cycle Counts for icctl Operations

Operation	Cycles to Execute
Disable I-cache	18
Enable I-cache	16
Invalidate I-cache	18
Load and Lock I-cache	5193
I-cache Status Request	21
I-cache Locking Status	20

Table B-14. Cycle Counts for dcctl Operations

Operations	Cycles to Execute
Disable D-cache	18
Enable D-cache	18
Invalidate D-cache	19
Load and Lock D-cache	19
D-cache Status Request	16
Quick Invalidate D-cache	14

Table B-15. Cycle Counts for intctl Operations

Operation	Cycles to Execute
Disable Interrupts	13
Enable Interrupts	13
Interrupt Status Request	8

Table B-16. MEM Format Instruction Encodings

[31:24]	[23:19]	[18:14]	[13:12]	[11:0]
Opcode	**src/dst**	**ABASE**	**Mode**	**Offset**

[31:24]	[23:19]	[18:14]	[13:10]	[9:7]	[6:5]	[4:0]
Opcode	**src/dst**	**ABASE**	**Mode**	**Scale**	**00**	**Index**
Displacement						

Effective Address

efa = offset	Opcode	dst		0	0	offset				
offset(*reg*)	Opcode	dst	reg	1	0	offset				
(*reg*)	Opcode	dst	reg	0	1	0	0		00	
disp + 8 (IP)	Opcode	dst		0	1	0	1		00	
	Displacement									
(*reg1*)[*reg2* * *scale*]	Opcode	dst	reg1	0	1	1	1	scale	00	reg2
disp	Opcode	dst		1	1	0	0		00	
	Displacement									
disp(*reg*)	Opcode	dst	reg	1	1	0	1		00	
	Displacement									
disp[*reg* * *scale*]	Opcode	dst		1	1	1	0	scale	00	reg
	Displacement									
disp(*reg1*)[*reg2***scale*]	Opcode	dst	reg1	1	1	1	1	scale	00	reg2
	Displacement									

Opcode	Mnemonic	Cycles to Execute
80	**ldob**	
82	**stob**	
84	**bx**	4-7
85	**balx**	5-8
86	**callx**	9-12
88	**ldos**	
8A	**stos**	
8C	**lda**	
90	**ld**	
92	**st**	
98	**ldl**	

Opcode	Mnemonic	Cycles to Execute
9A	**stl**	
A0	**ldt**	
A2	**stt**	
B0	**ldq**	
B2	**stq**	
C0	**ldib**	
C2	**stib**	
C8	**ldis**	
CA	**stis**	

1. The number of cycles required to execute these instructions is based on the addressing mode used (see Table B-10).

Table B-17. Addressing Mode Performance

Mode	Assembler Syntax	Memory Format	Number of Instruction Words	Cycles to Execute
Absolute Offset	exp	MEMA	1	1
Absolute Displacement	exp	MEMB	2	2
Register Indirect	(reg)	MEMB	1	1
Register Indirect with Offset	exp(reg)	MEMA	1	1
Register Indirect with Displacement	exp(reg)	MEMB	2	2
Index with Displacement	exp[reg*scale]	MEMB	2	2
Register Indirect with Index	(reg)[reg*scale]	MEMB	1	6
Register Indirect with Index + Displacement	exp(reg)[reg*scale]	MEMB	2	6
Instruction Pointer with Displacement	exp(IP)	MEMB	2	6

Peripheral Memory-Mapped Registers C

This chapter describes the memory-mapped registers for the integrated peripherals.

C.1 Overview

The Peripheral Memory-Mapped Register (PMMR) interface gives software the ability to read and modify internal control registers. Each of these registers is accessed as a memory-mapped 32-bit register with a unique memory address. Access is accomplished through regular memory-format instructions from the Intel® i960® core processor.

These memory-mapped registers are specific to the Intel® 80303 I/O processor only. They support the:

- DMA Controller Unit
- Memory Controller
- I2C Bus Interface Unit
- PCI and Peripheral Interrupt Controller Unit
- Messaging Unit
- Intel® 80303 I/O Processor Arbitration
- PCI-to-PCI Bridge Unit
- PCI Address Translation Unit
- Performance Monitoring Unit
- Application Accelerator Unit
- General Purpose Input Output (GPIO)
-

Each of these peripherals fully describe the independent functionality of the registers, control and usage.

Portions of the 80303 I/O Processor address space are already reserved by the i960 core processor. Addresses 0000 0000H through 0000 03FFH are reserved for the processor internal data RAM. This memory is dedicated to the i960 core processor only and is inaccessible from internal bus masters. Addresses FF00 0000H through FFFF FFFFH are reserved for the processor specific memory-mapped registers. Accesses to this address space do not generate external bus cycles.

The PMMR interface provides full accessibility from the Primary ATU, Secondary ATU, and the i960 core processor. Addresses 0000 1000H through 0000 17FFH are allocated to the PMMR interface.

C.2 Accessing the Peripheral Memory-Mapped Registers

The PMMR interface is a slave device connected to the 80303 I/O Processor internal bus. This interface accepts data transactions which appear on the internal bus from the Primary ATU, Secondary ATU, and the i960 core processor.

The PMMR interface allows these devices to perform read, write, or read-modify-write transactions. The specific actions taken when modifying any value in the PMMR space is independently defined within each chapter which describes the functionality of the register.

Note: The PMMR interface does not support multi-word burst accesses from any internal bus master.

All PMMR transactions shall be allowed from the i960 core processor operating in either user mode or supervisor mode. In addition, the PMMR shall not provide any access fault to the i960 core processor.

The following PMMR registers have read/write access from the internal bus (for both the PCI Bridge and ATU):

- Vendor ID Register
- Device ID Register
- Revision ID Register
- Class Code Register
- Header Type Register
- Subsystem ID Register (ATU Only)
- Subsystem Vendor ID Register (ATU Only)

For accesses through PCI configuration cycles, access is specified in the register definition located in the appropriate chapter.

For PCI Configuration Read transactions, the PMMR shall return a value of zero for registers declared as "reserved". For PCI Configuration Write transactions, the PMMR shall discard the data. For all other types of access, reading or writing a register declared as "reserved" is undefined.

C.3 Architecturally Reserved Memory Space

The 80303 I/O Processor provides 4 Gbytes of address space. Portions of this address space is architecturally reserved and refrained from use by the customers. Figure C-2 shows the reserved address space.

Addresses FF00 0000H through FFFF FFFFH are reserved for implementation-specific functions. This address range is termed "reserved" for future Intel® 80960 architecture implementations. Future 80960 architecture implementations may use these addresses for special functions such as mapped registers or data structures. Therefore, to ensure complete object level compatibility, portable code must not access or depend on values in this region.

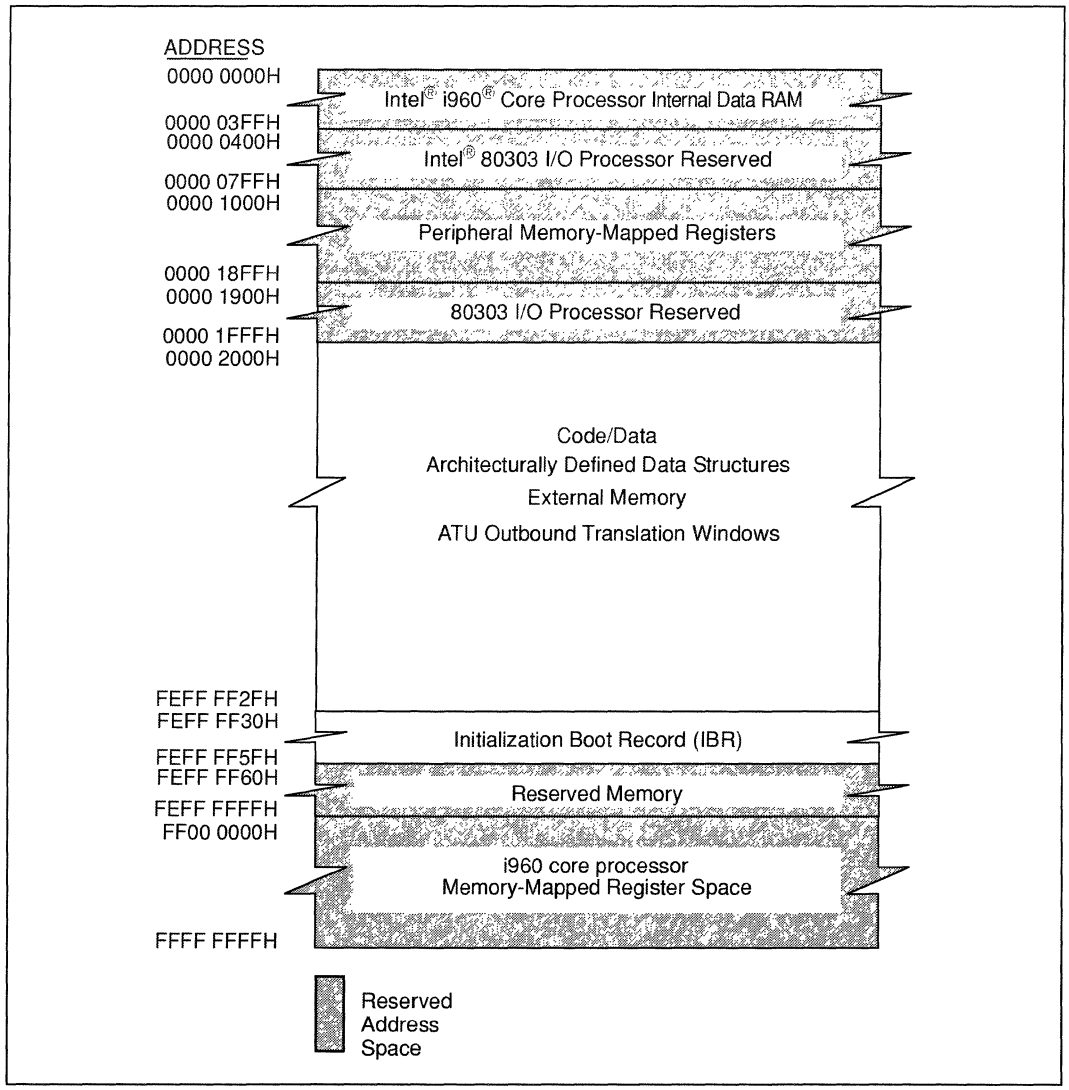
Addresses 0000 0000H through 0000 03FFH are reserved for the internal data RAM of the i960 core processor. This internal data RAM contains interrupt vectors plus RAM available to the application software for variable allocation or data structures. Loads and stores directed to these addresses access internal memory; instruction fetches from these addresses are not allowed for the 80303 I/O Processor.

Addresses 0000 0400H through 0000 1FFFH are reserved for 80303 I/O Processor use and should not be used by the system designer.

Addresses 0000 1000H through 0000 18FFH are allocated to the PMMR interface. These registers are reserved for 80303 I/O Processor use and should not be written by the system designer.

Figure C-2 shows the 80303 address space and addresses available to the applications.

Figure C-2. Intel® 80303 I/O Processor Address Space



C.4 Peripheral Memory-Mapped Register Address Space

The PMMR address space is divided to support the integrated peripherals on the 80303 I/O Processor. Table C-18 shows all of the 80303 I/O Processor integrated peripheral memory-mapped registers and their internal bus addresses.

Table C-18. Intel® 80960 Local Addresses Assigned to Integrated Peripherals

Integrated Peripheral	Internal Address Block
PCI to PCI Bridge Unit	0000 1000H through 0000 10FFH
Performance Monitoring Unit	0000 1100H through 0000 11FFH
Address Translation Unit	0000 1200H through 0000 12FFH
Messaging Unit	0000 1300H through 0000 13FFH
DMA Controller	0000 1400H through 0000 14FFH
Memory Controller	0000 1500H through 0000 15FFH
Internal Arbitration Unit	0000 1600H through 0000 163FH
Bus Interface Unit	0000 1640H through 0000 167FH
I^2C Bus Interface Unit	0000 1680H through 0000 16FFH
PCI And Peripheral Interrupt Controller	0000 1700H through 0000 17FFH
Application Accelerator Unit	0000 1800H through 0000 18FFH

The registers accessible via PCI configuration transactions are:

- PCI-to-PCI Bridge Unit
- Address Translation Units

The registers which must have the address translation logic configured to translate PCI addresses into the 80960 address space, to access the memory-mapped registers are:

- DMA Controllers
- Bus Interface Unit
- Memory Controller
- I^2C Bus Interface Unit
- Messaging Unit
- Application Accelerator Unit
- Internal Arbitration Unit
- Performance Monitoring
- PCI and Peripheral Interrupt Controller

Table C-19. Peripheral Memory-Mapped Register Locations (Sheet 1 of 8)

Intel® 80303 I/O Processor Peripheral	Register Description (Name)	Register Size in Bits	Internal Bus Address	PCI Configuration Space Register Number
PCI to PCI Bridge Unit	Vendor ID Register	16	0000 1000H	00H
	Device ID Register	16	0000 1002H	00H
	Primary Command Register	16	0000 1004H	01H
	Primary Status Register	16	0000 1006H	01H
	Revision ID Register	8	0000 1008H	02H
	Class Code Register	24	0000 1009H	02H
	Cacheline Size Register	8	0000 100CH	03H
	Primary Latency Timer Register	8	0000 100DH	03H
	Header Type Register	8	0000 100EH	03H
	Reserved	x	0000 100FH through 0000 1017H	04H through 05H
	Primary Bus Number Register	8	0000 1018H	06H
	Secondary Bus Number Register	8	0000 1019H	06H
	Subordinate Bus Number Register	8	0000 101AH	06H
	Secondary Latency Timer Register	8	0000 101BH	06H
	I/O Base Register	8	0000 101CH	07H
	I/O Limit Register	8	0000 101DH	07H
	Secondary Status Register	16	0000 101EH	07H
	Memory Base Register	16	0000 1020H	08H
	Memory Limit Register	16	0000 1022H	08H
	Prefetchable Memory Base Register	16	0000 1024H	09H
	Prefetchable Memory Limit Register	16	0000 1026H	09H
	Reserved	x	0000 1028H through 0000 1033H	0AH through 0CH
	Capabilities Pointer Register	8	0000 1034H	0DH
	Reserved	x	0000 1035H through 0000 103DH	0DH through 0FH
	Bridge Control Register	16	0000 103EH	0FH
	Extended Bridge Control Register	16	0000 1040H	10H
	Secondary IDSEL Control Register	16	0000 1042H	10H
	Primary Bridge Interrupt Status Register	32	0000 1044H	11H
	Secondary Bridge Interrupt Status Register	32	0000 1048H	12H
	Secondary Arbitration Control Register	32	0000 104CH	13H
	PCI Interrupt Routing Select Register	32	0000 1050H	14H
	Secondary I/O Base Register	8	0000 1054H	15H
	Secondary I/O Limit Register	8	0000 1055H	15H
	Reserved	x	0000 1056H	15H
	Secondary Memory Base Register	16	0000 1058H	16H
	Secondary Memory Limit Register	16	0000 105AH	16H
	Secondary Decode Enable Register	16	0000 105CH	17H

Table C-19. Peripheral Memory-Mapped Register Locations (Sheet 2 of 8)

Intel® 80303 I/O Processor Peripheral	Register Description (Name)	Register Size in Bits	Internal Bus Address	PCI Configuration Space Register Number
	Queue Control	16	0000 105EH	17H
	Reserved	32	0000 1060H	18H
	Reserved	32	0000 1064H	19H
	Capability Identifier Register	8	0000 1068H	1AH
	Next Item Pointer Register	8	0000 1069H	1AH
	Power Management Capabilities Register	16	0000 106AH	1AH
	Power Management Control/Status Register	16	0000 106CH	1BH
	PMCSR PCI to PCI Bridge Support	8	0000 106EH	1BH
	Reserved	x	0000 106FH through 0000 10FFH	
Performance Monitoring Unit	Global Timer Mode Register	32	0000 1100H	Must Translate PCI address to the Intel® 80960 Memory-Mapped Address
	Event Select Register	32	0000 1104H	
	Event Monitoring Interrupt Status Register	32	0000 1108H	
	Reserved	x	0000 110CH	
	Global Time Stamp Register	32	0000 1110H	
	Programmable Event Counter Register 1	32	0000 1114H	
	Programmable Event Counter Register 2	32	0000 1118H	
	Programmable Event Counter Register 3	32	0000 111CH	
	Programmable Event Counter Register 4	32	0000 1120H	
	Programmable Event Counter Register 5	32	0000 1124H	
	Programmable Event Counter Register 6	32	0000 1128H	
	Programmable Event Counter Register 7	32	0000 112CH	
	Programmable Event Counter Register 8	32	0000 1130H	
	Programmable Event Counter Register 9	32	0000 1134H	
	Programmable Event Counter Register 10	32	0000 1138H	
	Programmable Event Counter Register 11	32	0000 113CH	
	Programmable Event Counter Register 12	32	0000 1140H	
	Programmable Event Counter Register 13	32	0000 1144H	
	Programmable Event Counter Register 14	32	0000 1148H	
	Reserved	x	0000 114CH through 0000 11FFH	

Table C-19. Peripheral Memory-Mapped Register Locations (Sheet 3 of 8)

Intel® 80303 I/O Processor Peripheral	Register Description (Name)	Register Size in Bits	Internal Bus Address	PCI Configuration Space Register Number
Address Translation Unit	ATU Vendor ID Register	16	0000 1200H	00H
	ATU Device ID Register	16	0000 1202H	00H
	Primary ATU Command Register	16	0000 1204H	01H
	Primary ATU Status Register	16	0000 1206H	01H
	ATU Revision ID Register	8	0000 1208H	02H
	ATU Class Code Register	24	0000 1209H	02H
	ATU Cacheline Size Register	8	0000 120CH	03H
	ATU Latency Timer Register	8	0000 120DH	03H
	ATU Header Type Register	8	0000 120EH	03H
	BIST Register	8	0000 120FH	03H
	Primary Inbound ATU Base Address Register	32	0000 1210H	04H
	Reserved	32	0000 1214H	05H
	Reserved	32	0000 1218H	06H
	Reserved	32	0000 121CH	07H
	Reserved	32	0000 1220H	08H
	Reserved	32	0000 1224H	09H
	Reserved	32	0000 1228H	0AH
	ATU Subsystem Vendor ID Register	16	0000 122CH	0BH
	ATU Subsystem ID Register	16	0000 122EH	0BH
	Expansion ROM Base Address Register	32	0000 1230H	0CH
	ATU Capabilities Pointer Register	8	0000 1234H	0DH
	Reserved	24	0000 1235H	0DH
	Reserved	32	0000 1238H	0EH
	ATU Interrupt Line Register	8	0000 123CH	0FH
	ATU Interrupt Pin Register	8	0000 123DH	0FH
	ATU Minimum Grant Register	8	0000 123EH	0FH
	ATU Maximum Latency Register	8	0000 123FH	0FH

Table C-19. Peripheral Memory-Mapped Register Locations (Sheet 4 of 8)

Intel® 80303 I/O Processor Peripheral	Register Description (Name)	Register Size in Bits	Internal Bus Address	PCI Configuration Space Register Number
Address Translation Unit Extended Configuration Registers	Primary Inbound ATU Limit Register	32	0000 1240H	10H
	Primary Inbound ATU Translate Value Register	32	0000 1244H	11H
	Secondary Inbound ATU Base Address Register	32	0000 1248H	12H
	Secondary Inbound ATU Limit Register	32	0000 124CH	13H
	Secondary Inbound ATU Translate Value Register	32	0000 1250H	14H
	Primary Outbound Memory Window Value Register	32	0000 1254H	15H
	Reserved	32	0000 1258H	16H
	Primary Outbound I/O Window Value Register	32	0000 125C	17H
	Primary Outbound DAC Window Value Register	32	0000 1260H	18H
	Primary Outbound Upper 64-bit DAC Register	32	0000 1264H	19H
	Secondary Outbound Memory Window Value Register	32	0000 1268H	1AH
	Secondary Outbound I/O Window Value Register	32	0000 126CH	1BH
	Reserved	32	0000 1270H	1CH
	Expansion ROM Limit Register	32	0000 1274H	1DH
	Expansion ROM Translate Value Register	32	0000 1278H	1EH
	Reserved	32	0000 127CH	1FH
	ATU Capability Identifier Register	8	0000 1280H	20H
	ATU Next Item Pointer Register	8	0000 1281H	20H
	ATU Power Management Capabilities Register	16	0000 1282H	20H
	ATU Power Management Control/Status Register	16	0000 1284H	21H
	Reserved	16	0000 1286H	21H
	ATU Configuration Register	32	0000 1288H	22H
	Reserved	32	0000 128CH	23H
	Primary ATU Interrupt Status Register	32	0000 1290H	24H
	Secondary ATU Interrupt Status Register	32	0000 1294H	25H
	Secondary ATU Command Register	16	0000 1298H	26H
	Secondary ATU Status Register	16	0000 129AH	26H
	Secondary Outbound DAC Window Value Register	32	0000 129CH	27H
	Secondary Outbound Upper 64-bit DAC Register	32	0000 12A0H	28H
	Primary Outbound Configuration Cycle Address Register	32	0000 12A4H	29H
	Secondary Outbound Configuration Cycle Address Register	32	0000 12A8H	2AH
	Primary Outbound Configuration Cycle Data Register	32	0000 12ACH	Reserved
	Secondary Outbound Configuration Cycle Data Register	32	0000 12B0H	Reserved
	Primary ATU Queue Control Register	32	0000 12B4H	2DH
	Secondary ATU Queue Control Register	32	0000 12B8H	2EH
	Primary ATU Interrupt Mask Register	32	0000 12BCH	2FH
	Secondary ATU Interrupt Mask Register	32	0000 12C0H	30H
	Reserved	x	0000 12C4H through 0000 12FFH	

Table C-19. Peripheral Memory-Mapped Register Locations (Sheet 5 of 8)

Intel® 80303 I/O Processor Peripheral	Register Description (Name)	Register Size in Bits	Internal Bus Address	PCI Configuration Space Register Number
Messaging Unit	Reserved	x	0000 1300H through 0000 130CH	Available through ATU Primary Inbound Translation Window or must translate PCI address to the 80960 Memory-Mapped Address
	Inbound Message Register 0	32	0000 1310H	
	Inbound Message Register 1	32	0000 1314H	
	Outbound Message Register 0	32	0000 1318H	
	Outbound Message Register 1	32	0000 131CH	
	Inbound Doorbell Register	32	0000 1320H	
	Inbound Interrupt Status Register	32	0000 1324H	
	Inbound Interrupt Mask Register	32	0000 1328H	
	Outbound Doorbell Register	32	0000 132CH	
	Outbound Interrupt Status Register	32	0000 1330H	
	Outbound Interrupt Mask Register	32	0000 1334H	
	Reserved	x	0000 1338H through 0000 134FH	Must Translate PCI address to the 80960 Memory-Mapped Address
	MU Configuration Register	32	0000 1350H	
	Queue Base Address Register	32	0000 1354H	
	Reserved	32	0000 1358H	
	Reserved	32	0000 135CH	
	Inbound Free Head Pointer Register	32	0000 1360H	
	Inbound Free Tail Pointer Register	32	0000 1364H	
	Inbound Post Head Pointer Register	32	0000 1368H	
	Inbound Post Tail Pointer Register	32	0000 136CH	
	Outbound Free Head Pointer Register	32	0000 1370H	
	Outbound Free Tail Pointer Register	32	0000 1374H	
	Outbound Post Head Pointer Register	32	0000 1378H	
	Outbound Post Tail Pointer Register	32	0000 137CH	
	Index Address Register	32	0000 1380H	
	Reserved	x	0000 1384H through 0000 13FFH	

Table C-19. Peripheral Memory-Mapped Register Locations (Sheet 6 of 8)

Intel® 80303 I/O Processor Peripheral	Register Description (Name)	Register Size in Bits	Internal Bus Address	PCI Configuration Space Register Number
DMA Controller	Channel 0 Channel Control Register	32	0000 1400H	Must Translate PCI address to the 80960 Memory-Mapped Address
	Channel 0 Channel Status Register	32	0000 1404H	
	Reserved	32	0000 1408H	
	Channel 0 Descriptor Address Register	32	0000 140CH	
	Channel 0 Next Descriptor Address Register	32	0000 1410H	
	Channel 0 PCI Address Register	32	0000 1414H	
	Channel 0 PCI Upper Address Register	32	0000 1418H	
	Channel 0 Internal Bus Address Register	32	0000 141CH	
	Channel 0 Byte Count Register	32	0000 1420H	
	Channel 0 Descriptor Control Register	32	0000 1424H	
	Reserved	x	0000 1428H through 0000 143FH	
	Channel 1 Channel Control Register	32	0000 1440H	
	Channel 1 Channel Status Register	32	0000 1444H	
	Reserved	32	0000 1448H	
	Channel 1 Descriptor Address Register	32	0000 144CH	
	Channel 1 Next Descriptor Address Register	32	0000 1450H	
	Channel 1 PCI Address Register	32	0000 1454H	
	Channel 1 PCI Upper Address Register	32	0000 1458H	
	Channel 1 Internal Bus Address Register	32	0000 145CH	
	Channel 1 Byte Count Register	32	0000 1460H	
	Channel 1 Descriptor Control Register	32	0000 1464H	
	Reserved	x	0000 1468H through 0000 147FH	
	Channel 2 Channel Control Register	32	0000 1480H	
	Channel 2 Channel Status Register	32	0000 1484H	
	Reserved	32	0000 1488H	
	Channel 2 Descriptor Address Register	32	0000 148CH	
	Channel 2 Next Descriptor Address Register	32	0000 1490H	
	Channel 2 PCI Address Register	32	0000 1494H	
	Channel 2 PCI Upper Address Register	32	0000 1498H	
	Channel 2 Internal Bus Address Register	32	0000 149CH	
	Channel 2 Byte Count Register	32	0000 14A0H	
	Channel 2 Descriptor Control Register	32	0000 14A4H	
	Reserved	x	0000 14A8H through 0000 14FFH	

Table C-19. Peripheral Memory-Mapped Register Locations (Sheet 7 of 8)

Intel® 80303 I/O Processor Peripheral	Register Description (Name)	Register Size in Bits	Internal Bus Address	PCI Configuration Space Register Number
Memory Controller	SDRAM Initialization Register	32	0000 1500H	Must Translate PCI address to the 80960 Memory-mapped Address
	SDRAM Control Register	32	0000 1504H	
	SDRAM Base Register	32	0000 1508H	
	SDRAM Bank 0 Size Register	32	0000 150CH	
	SDRAM Bank 1 Size Register	32	0000 1510H	
	Reserved	32	0000 1514H	
	Reserved	32	0000 1518H	
	Reserved	32	0000 151CH	
	Reserved	32	0000 1520H	
	Reserved	32	0000 1524H	
	Reserved	32	0000 1528H	
	Reserved	32	0000 152CH	
	Reserved	32	0000 1530H	
	ECC Control Register	32	0000 1534H	
	ECC Log 0 Register	32	0000 1538H	
	ECC Log 1 Register	32	0000 153CH	
	ECC Address 0 Register	32	0000 1540H	
	ECC Address 1 Register	32	0000 1544H	
	ECC Test Register	32	0000 1548H	
	Flash Base 0 Register	32	0000 154CH	
	Flash Base 1 Register	32	0000 1550H	
	Flash Bank 0 Size Register	32	0000 1554H	
	Flash Bank 1 Size Register	32	0000 1558H	
	Flash Wait State 0 Register	32	0000 155CH	
	Flash Wait State 1 Register	32	0000 1560H	
	Memory Controller Interrupt Status Register	32	0000 1564H	
	Refresh Frequency Register	32	0000 1568H	
	Reserved	x	0000 156CH through 0000 15FFH	
Internal Arbitration Unit	Internal Arbitration Control Register	32	0000 1600H	Must Translate PCI address to the 80960 Memory-mapped Address
	Master Latency Timer Register	32	0000 1604H	
	Multi-Transaction Timer Register	32	0000 1608H	
	Reserved	x	0000 160CH through 0000 163FH	
Bus Interface Unit	BIU Control Register	32	0000 1640H	Must Translate PCI address to the 80960 Memory-mapped Address
	BIU Interrupt Status Register	32	0000 1644H	
	Reserved	x	0000 1648H through 0000 167FH	

Table C-19. Peripheral Memory-Mapped Register Locations (Sheet 8 of 8)

Intel® 80303 I/O Processor Peripheral	Register Description (Name)	Register Size in Bits	Internal Bus Address	PCI Configuration Space Register Number
I^2C Bus Interface Unit	I^2C Control Register	32	0000 1680H	Must Translate PCI address to the 80960 Memory-mapped Address
	I^2C Status Register	32	0000 1684H	
	I^2C Slave Address Register	32	0000 1688H	
	I^2C Data Buffer Register	32	0000 168CH	
	I^2C Clock Control Register	32	0000 1690H	
	I^2C Bus Monitor Register	32	0000 1694H	
	Reserved	x	0000 1698H through 0000 16FFH	
PCI And Peripheral Interrupt Controller	NMI Interrupt Status Register	32	0000 1700H	Must Translate PCI address to the 80960 Memory-mapped Address
	XINT7 Interrupt Status Register	32	0000 1704H	
	XINT6 Interrupt Status Register	32	0000 1708H	
	PCI Interrupt Routing Select Register	32	See PCI to PCI Bridge Configuration Space (0000 1050H)	14H
	Processor Device ID Register	32	0000 1710H	Must Translate PCI address to the 80960 Memory-mapped Address
	Reserved	32	0000 1714H	
	Reserved	32	0000 1718H	
	GPIO Output Enable Register	32	0000 171CH	
	GPIO Input Data Register	32	0000 1720H	
	GPIO Output Data Register	32	0000 1724H	
	Reserved	x	0000 1728H through 0000 17FFH	
Application Accelerator Unit	Accelerator Control Register	32	0000 1800H	
	Accelerator Status Register	32	0000 1804H	
	Accelerator Descriptor Address Register	32	0000 1808H	
	Accelerator Next Descriptor Address Register	32	0000 180CH	
	80960 Source Address 1 Register	32	0000 1810H	
	80960 Source Address 2 Register	32	0000 1814H	
	80960 Source Address 3 Register	32	0000 1818H	
	80960 Source Address 4 Register	32	0000 181CH	
	80960 Destination Address Register	32	0000 1820H	
	Accelerator Byte Count Register	32	0000 1824H	
	Accelerator Descriptor Control Register	32	0000 1828H	
	80960 Source Address 5 Register	32	0000 182CH	
	80960 Source Address 6 Register	32	0000 1830H	
	80960 Source Address 7 Register	32	0000 1834H	
	80960 Source Address 8 Register	32	0000 1838H	
	Reserved	x	0000 183CH through 0000 18FFH	

Intel around the world

UNITED STATES AND CANADA
Intel Corporation
Robert Noyce Building
2200 Mission College Boulevard
P.O. Box 58119
Santa Clara, CA 95052-8119
USA
Phone: (800) 628-8686

EUROPE
Intel Corporation (UK) Ltd.
Pipers Way
Swindon
Wiltshire SN3 1RJ
UK

Phone:
England (44) 1793 403 000
Germany (49) 89 99143 0
France (33) 1 4571 7171
Italy (39) 2 575 441
Israel (972) 2 589 7111
Netherlands (31) 10 286 6111
Sweden (46) 8 705 5600

ASIA-PACIFIC
Intel Semiconductor Ltd.
32/F Two Pacific Place
88 Queensway, Central
Hong Kong, SAR
Phone: (852) 2844 4555

JAPAN
Intel Kabushiki Kaisha
P.O. Box 115 Tsukuba-gakuen
5-6 Tokodai, Tsukuba-shi
Ibaraki-ken 305
Japan
Phone: (81) 298 47 8522

SOUTH AMERICA
Intel Semicondutores do Brazil
Rue Florida, 1703-2 and CJ22
CEP 04565-001 Sao Paulo-SP
Brazil
Phone: (55) 11 5505 2296

FOR MORE INFORMATION
To learn more about Intel Corporation, visit our site on the World Wide Web at www.intel.com

Printed in USA

intel®