



# i960<sup>®</sup> RM/RN I/O Processor

Developer's Manual

---

*July 1998*

Order Number: 273158-001



Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The 80960RM/RN may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature may be obtained by calling 1-800-548-4725 or by visiting Intel's website at <http://www.intel.com>.

Copyright © Intel Corporation, 1998

\*Third-party brands and names are the property of their respective owners.



# Contents

---

<b>1</b>	<b>Introduction</b> .....	<b>1-1</b>
1.1	Intel's i960 <sup>®</sup> RM/RN I/O Processor .....	1-1
1.2	i960 <sup>®</sup> RM/RN I/O Processor Features.....	1-2
1.2.1	Intelligent I/O (I <sub>2</sub> O).....	1-2
1.2.2	PCI-to-PCI Bridge Unit.....	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	I <sup>2</sup> C Bus Interface Unit.....	1-4
1.2.9	Secondary PCI Arbitration Unit.....	1-4
1.2.10	Performance Monitoring Unit .....	1-4
1.2.11	Application Accelerator .....	1-4
1.2.12	Bus Interface Unit .....	1-5
1.2.13	Wind River Systems IxWorks* RTOS .....	1-5
1.3	i960 <sup>®</sup> Core Processor Features (80960JT) .....	1-6
1.3.1	80960 Local Bus .....	1-7
1.3.2	Timer Unit .....	1-7
1.3.3	Priority Interrupt Controller.....	1-7
1.3.4	Faults and Debugging.....	1-8
1.3.5	On-Chip Cache and Data RAM .....	1-8
1.3.6	Local Register Cache .....	1-8
1.3.7	Test Features.....	1-8
1.3.8	Memory-Mapped Control Registers.....	1-9
1.3.9	Instructions, Data Types and Memory Addressing Modes .....	1-9
1.4	About This Document .....	1-9
1.4.1	Terminology .....	1-10
1.4.2	Representing Numbers.....	1-10
1.4.3	Fields .....	1-10
1.4.4	Specifying Bit and Signal Values.....	1-11
1.4.5	Signal Name Conventions .....	1-11
1.4.6	<i>Solutions960</i> <sup>®</sup> Program .....	1-11
1.4.7	Related Documents .....	1-12
1.4.8	Electronic Information.....	1-12
<b>2</b>	<b>Data Types and Memory Addressing Modes</b> .....	<b>2-1</b>
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
<b>3</b>	<b>Programming Environment.....</b>	<b>3-1</b>
3.1	Overview.....	3-1
3.2	Registers and Literals as Instruction Operands.....	3-1
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).....	3-5
3.3.1	i960® Core Processor Function Memory-Mapped Registers.....	3-6
3.3.1.1	Restrictions on Instructions that Access the i960® Core Processor Memory-Mapped Registers.....	3-6
3.3.1.2	Access Faults for i960® Core Processor MMRs.....	3-7
3.3.2	i960® RM/RN 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).....	3-14
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.....	3-18
<b>4</b>	<b>Cache and On-Chip Data RAM.....</b>	<b>4-1</b>
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 and Bus Masters and Cache Coherency .....	4-8
4.4.7	Data Cache Visibility.....	4-8
<b>5</b>	<b>Instruction Set Overview.....</b>	<b>5-1</b>
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.....	5-6
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 .....	5-10
5.2.6.1	Compare and Conditional Compare .....	5-11
5.2.6.2	Compare and Increment or Decrement .....	5-11
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 .....	5-20
5.3.2.1	Masking of Integer Overflow .....	5-20
5.3.2.2	Avoid Using PFP, SP, R3 As Destinations for MDU Instructions .....	5-20
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

<b>6</b>	<b>Instruction Set Reference .....</b>	<b>6-1</b>
6.1	Notation .....	6-1
6.1.1	Alphabetic Reference .....	6-1
6.1.2	Mnemonic .....	6-2
6.1.3	Format .....	6-2
6.1.4	Description.....	6-2
6.1.5	Action.....	6-3
6.1.6	Faults .....	6-4
6.1.7	Example.....	6-4
6.1.8	Opcode and Instruction Format .....	6-4
6.1.9	See Also .....	6-4
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.....	6-8
6.2.3	addi, addo.....	6-9
6.2.4	alterbit.....	6-10
6.2.5	and, andnot.....	6-11
6.2.6	atadd.....	6-12
6.2.7	atmod.....	6-13
6.2.8	b, bx.....	6-14
6.2.9	bal, balx .....	6-15
6.2.10	bbc, bbs .....	6-16
6.2.11	BRANCH<cc>.....	6-17
6.2.12	bswap .....	6-19
6.2.13	call .....	6-20
6.2.14	calls .....	6-21
6.2.15	callx .....	6-23
6.2.16	chkbit .....	6-24
6.2.17	clrbit .....	6-25
6.2.18	cmpdeci, cmpdeco.....	6-26
6.2.19	cmpinci, cmpinco .....	6-27
6.2.20	COMPARE .....	6-28
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.....	6-39
6.2.25	ediv .....	6-40



6.2.26	emul .....	6-41
6.2.27	eshro .....	6-42
6.2.28	extract .....	6-43
6.2.29	FAULT<cc> .....	6-44
6.2.30	flushreg .....	6-46
6.2.31	fmark .....	6-47
6.2.32	halt .....	6-48
6.2.33	icctl .....	6-49
6.2.34	intctl .....	6-55
6.2.35	intdis .....	6-56
6.2.36	inten .....	6-57
6.2.37	LOAD .....	6-58
6.2.38	lda .....	6-61
6.2.39	mark .....	6-62
6.2.40	modac .....	6-63
6.2.41	modi .....	6-64
6.2.42	modify .....	6-65
6.2.43	modpc .....	6-66
6.2.44	modtc .....	6-67
6.2.45	MOVE .....	6-68
6.2.46	muli, mulo .....	6-70
6.2.47	nand .....	6-71
6.2.48	nor .....	6-72
6.2.49	not, notand .....	6-73
6.2.50	notbit .....	6-74
6.2.51	notor .....	6-75
6.2.52	or, ornot .....	6-76
6.2.53	remi, remo .....	6-77
6.2.54	ret .....	6-78
6.2.55	rotate .....	6-80
6.2.56	scanbit .....	6-81
6.2.57	scanbyte .....	6-82
6.2.58	SEL<cc> .....	6-83
6.2.59	setbit .....	6-84
6.2.60	SHIFT .....	6-85
6.2.61	spanbit .....	6-87
6.2.62	STORE .....	6-88
6.2.63	subc .....	6-91
6.2.64	SUB<cc> .....	6-92
6.2.65	subi, subo .....	6-94
6.2.66	syncf .....	6-95
6.2.67	sysctl .....	6-96
6.2.68	TEST<cc> .....	6-100
6.2.69	xnor, xor .....	6-102
<b>7</b>	<b>Procedure Calls .....</b>	<b>7-1</b>
7.1	Call and Return Mechanism .....	7-2
7.1.1	Local Registers and the Procedure Stack .....	7-3
7.1.2	Local Register and Stack Management .....	7-4
7.1.2.1	Frame Pointer .....	7-4

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 .....	7-5
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.....	7-11
7.3	Parameter Passing.....	7-12
7.4	Local Calls.....	7-13
7.5	System Calls .....	7-14
7.5.1	System Procedure Table .....	7-14
7.5.1.1	Procedure Entries.....	7-15
7.5.1.2	Supervisor Stack Pointer .....	7-16
7.5.1.3	Trace Control Bit.....	7-16
7.5.2	System Call to a Local Procedure .....	7-16
7.5.3	System Call to a Supervisor Procedure.....	7-16
7.6	User and Supervisor Stacks.....	7-17
7.7	Interrupt and Fault Calls .....	7-17
7.8	Returns.....	7-17
7.9	Branch-and-Link .....	7-19
<b>8</b>	<b>PCI and Peripheral Interrupt Controller Unit .....</b>	<b>8-1</b>
8.1	Overview .....	8-1
8.1.1	The i960 <sup>®</sup> RM/RN I/O Processor Core Interrupt Architecture .....	8-2
8.1.2	Software Requirements For Interrupt Handling .....	8-3
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 .....	8-8
8.1.8	Sampling Pending Interrupts in the Interrupt Table .....	8-9
8.1.9	Saving the Interrupt Mask.....	8-11
8.2	The i960 <sup>®</sup> Core Processor Interrupt Controller .....	8-11
8.2.1	Interrupt Controller Dedicated Mode.....	8-13
8.2.2	Interrupt Detection .....	8-13
8.2.3	Non-Maskable Interrupt (NMI#) .....	8-14
8.2.4	Timer Interrupts .....	8-15
8.2.5	Software Interrupts .....	8-15





8.2.6	Interrupt Operation Sequence .....	8-15
8.2.7	Setting Up the Interrupt Controller .....	8-16
8.2.8	Interrupt Service Routines .....	8-16
8.2.9	Interrupt Context Switch .....	8-17
8.2.9.1	Servicing An Interrupt From Executing State.....	8-17
8.2.9.2	Servicing An Interrupt From Interrupted State .....	8-18
8.3	PCI and Peripheral Interrupts .....	8-18
8.3.1	Pin Descriptions.....	8-20
8.3.2	PCI Interrupt Routing .....	8-21
8.3.3	Internal Peripheral Interrupt Routing.....	8-21
8.3.3.1	XINT6 Interrupt Sources .....	8-21
8.3.3.2	XINT7 Interrupt Sources .....	8-22
8.3.3.3	NMI# Interrupt Sources.....	8-23
8.3.4	PCI Outbound Doorbell Interrupts .....	8-25
8.4	Default Status .....	8-25
8.4.1	Interrupt Controller Register Access Requirements .....	8-26
8.4.2	Optimizing Interrupt Performance.....	8-26
8.4.3	Interrupt Service Latency.....	8-27
8.4.4	Features to Improve Interrupt Performance.....	8-28
8.4.4.1	Vector Caching Option.....	8-28
8.4.4.2	Caching Interrupt Routines and Reserving Register Frames .....	8-29
8.4.4.3	Caching the Interrupt Stack .....	8-29
8.4.5	Base Interrupt Latency.....	8-29
8.4.6	Maximum Interrupt Latency .....	8-30
8.4.7	Avoiding Certain Destinations for MDU Operations.....	8-32
8.4.8	Secondary PCI to Primary PCI Interrupt Routing Latency.....	8-32
8.5	Register Definitions .....	8-32
8.5.1	Interrupt Control Register (ICON) .....	8-33
8.5.2	Interrupt Mapping Registers (IMAP0-IMAP2) .....	8-34
8.5.3	Interrupt Pending (IPND) and Interrupt Mask (IMSK) Registers .....	8-36
8.5.4	PCI Interrupt Routing Select Register - PIRSR .....	8-39
8.5.5	XINT6 Interrupt Status Register - X6ISR .....	8-40
8.5.6	XINT7 Interrupt Status Register- X7ISR .....	8-41
8.5.7	NMI Interrupt Status Register - NISR .....	8-42
<b>9</b>	<b>Faults.....</b>	<b>9-1</b>
9.1	Fault Handling Overview .....	9-1
9.2	Fault Types.....	9-3
9.3	Fault Table.....	9-4
9.4	Stack Used in Fault Handling .....	9-6
9.5	Fault Record .....	9-6
9.5.1	Fault Record Description .....	9-7
9.5.2	Fault Record Location.....	9-8
9.6	Multiple and Parallel Faults .....	9-8
9.6.1	Multiple Non-Trace Faults on the Same Instruction.....	9-8
9.6.2	Multiple Trace Fault Conditions on the Same Instruction .....	9-9
9.6.3	Multiple Trace and Non-Trace Fault Conditions on the Same Instruction.....	9-9
9.6.4	Parallel Faults .....	9-9

	9.6.4.1	Faults on Multiple Instructions Executed in Parallel .....	9-9
	9.6.4.2	Fault Record for Parallel Faults .....	9-10
	9.6.5	Override Faults .....	9-11
	9.6.6	System Error.....	9-12
9.7		Fault Handling Procedures.....	9-12
	9.7.1	Possible Fault Handling Procedure Actions.....	9-12
	9.7.2	Program Resumption Following a Fault.....	9-12
	9.7.2.1	Faults Happening Before Instruction Execution.....	9-13
	9.7.2.2	Faults Happening During Instruction Execution.....	9-13
	9.7.2.3	Faults Happening After Instruction Execution.....	9-13
	9.7.3	Return Instruction Pointer (RIP).....	9-14
	9.7.4	Returning to Point in Program Where Fault Occurred .....	9-14
	9.7.5	Returning to a Point in the Program Other Than Where the Fault Occurred .....	9-14
	9.7.6	Fault Controls .....	9-15
9.8		Fault Handling Action .....	9-15
	9.8.1	Local Fault Call .....	9-16
	9.8.2	System-Local Fault Call.....	9-16
	9.8.3	System-Supervisor Fault Call .....	9-17
	9.8.4	Faults and Interrupts.....	9-17
9.9		Precise and Imprecise Faults .....	9-18
	9.9.1	Precise Faults .....	9-18
	9.9.2	Imprecise Faults .....	9-18
	9.9.3	Asynchronous Faults .....	9-18
	9.9.4	No Imprecise Faults (AC.nif) Bit .....	9-19
	9.9.5	Controlling Fault Precision.....	9-19
9.10		Fault Reference.....	9-20
	9.10.1	ARITHMETIC Faults.....	9-21
	9.10.2	CONSTRAINT Faults.....	9-22
	9.10.3	OPERATION Faults.....	9-23
	9.10.4	OVERRIDE Faults .....	9-24
	9.10.5	PARALLEL Faults.....	9-25
	9.10.6	PROTECTION Faults .....	9-26
	9.10.7	TRACE Faults.....	9-27
	9.10.8	TYPE Faults .....	9-29
<b>10</b>		<b>Tracing and Debugging .....</b>	<b>10-1</b>
	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 .....	10-3
	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 .....	10-4
	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 .....	10-9
10.2.7.6	Instruction Breakpoint Registers – IPBx .....	10-10
10.3	Generating a Trace Fault.....	10-11
10.4	Handling Multiple Trace Events.....	10-11
10.5	Trace Fault Handling Procedure.....	10-11
10.5.1	Tracing and Interrupt Procedures .....	10-12
10.5.2	Tracing on Calls and Returns .....	10-12
10.5.2.1	Tracing on Explicit Call .....	10-12
10.5.2.2	Tracing on Implicit Call .....	10-13
10.5.2.3	Tracing on Return from Explicit Call .....	10-14
10.5.2.4	Tracing on Return from Implicit Call: Fault Case .....	10-14
10.5.2.5	Tracing on Return from Implicit Call: Interrupt Case.....	10-14
<b>11</b>	<b>Initialization and System Requirements .....</b>	<b>11-1</b>
11.1	Overview.....	11-1
11.1.1	Core Initialization .....	11-1
11.1.2	General Initialization .....	11-2
11.2	i960 <sup>®</sup> RM/RN 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	i960 <sup>®</sup> 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	80960 Local Bus Confidence Test.....	11-7
11.3.1.3	The Fail Signal (FAIL#).....	11-7
11.3.1.4	IMI Alignment Check and Core Processor Error.....	11-8
11.3.1.5	FAIL# Code.....	11-8
11.4	Initial Memory Image (IMI).....	11-9
11.4.1	Initialization Boot Record (IBR).....	11-11
11.4.2	Process Control Block – PRCB .....	11-14
11.4.3	Process PRCB Flow .....	11-16
11.4.3.1	AC Initial Image .....	11-17
11.4.3.2	Fault Configuration Word.....	11-17
11.4.3.3	Instruction Cache Configuration Word.....	11-17
11.4.3.4	Register Cache Configuration Word .....	11-17
11.4.4	Control Table .....	11-18
11.5	Device Identification on Reset.....	11-19
11.6	Reinitializing and Relocating Data Structures .....	11-20
11.6.1	Output Clocks .....	11-20
<b>12</b>	<b>Core Processor and Internal Operation.....</b>	<b>12-1</b>
12.1	Core Processor Memory Attributes .....	12-1
12.2	Physical Memory Attributes .....	12-1

12.2.1	PMCON Registers .....	12-1
12.2.2	Bus Control Register – BCON .....	12-3
12.3	Programming the Logical Memory Attributes .....	12-4
12.3.1	Logical Memory Attributes .....	12-4
12.3.2	Logical Memory Address Registers - LMADR0:1 .....	12-5
12.3.3	Defining the Effective Range of a Logical Data Template .....	12-7
12.3.4	Data Caching Enable .....	12-7
12.3.5	Enabling the Logical Memory Template .....	12-7
12.3.6	Initialization .....	12-8
12.3.7	Boundary Conditions for Logical Memory Templates .....	12-8
12.3.7.1	Internal Memory Locations and Peripheral MMRs.....	12-8
12.3.7.2	Overlapping Logical Data Template Ranges .....	12-8
12.3.7.3	Accesses Across LMT Boundaries .....	12-8
12.3.8	Modifying the LMT Registers .....	12-8
12.4	Bus Interface Unit.....	12-9
12.4.1	Overview.....	12-9
12.4.2	Addressing.....	12-11
12.4.2.1	<b>Bus Width</b> .....	<b>12-11</b>
12.4.3	Multi-Transaction Timer .....	12-11
12.4.4	Features .....	12-11
12.4.4.1	Write Buffering .....	12-11
12.4.4.2	Instruction Fetch Bypass .....	12-12
12.4.4.3	Instruction Prefetch .....	12-12
12.4.4.4	Write Merging .....	12-12
12.4.4.5	Atomic Accesses .....	12-13
12.4.5	Interrupts and Error Conditions .....	12-13
12.4.5.1	Master-Abort .....	12-13
12.4.5.2	PCI Target-Abort.....	12-14
12.4.5.3	Internal Bus Target-Abort .....	12-14
12.4.6	Register Definitions .....	12-16
12.4.6.1	BIU Control Register - BIUCR .....	12-16
12.4.6.2	BIU Interrupt Status Register - BIUISR.....	12-17
<b>13</b>	<b>Memory Controller</b> .....	<b>13-1</b>
13.1	Overview .....	13-1
13.1.1	Memory Controller Terminology .....	13-2
13.2	Flash Memory Support .....	13-3
13.2.1	Flash Memory Addressing.....	13-4
13.2.2	Flash Read Cycle .....	13-5
13.2.3	Flash Write Cycle .....	13-8
13.3	SDRAM Memory Support.....	13-9
13.3.1	SDRAM Sizes and Configurations.....	13-11
	Address Register Programming Examples 13-12	
13.3.2	SDRAM Addressing.....	13-13
13.3.3	32-bit Mode.....	13-14
13.3.4	Page Hit/Miss Determination .....	13-14
13.3.5	SDRAM Commands .....	13-17
13.3.6	SDRAM Initialization .....	13-18
13.3.6.1	SDRAM Mode Programming .....	13-19
13.3.6.2	SDRAM Read Cycle .....	13-20
13.3.6.3	SDRAM Write Cycle .....	13-24



13.3.6.4	SDRAM Refresh Cycle .....	13-27
13.3.7	Error Correction and Detection .....	13-29
13.3.7.1	ECC Generation .....	13-29
13.3.7.2	ECC Generation for Partial Writes .....	13-30
13.3.7.3	ECC Checking .....	13-31
13.3.7.4	Scrubbing.....	13-32
13.3.7.5	ECC Disabled .....	13-32
13.3.7.6	ECC Testing .....	13-32
13.3.8	SDRAM Clocking .....	13-33
13.4	Power Failure Mode .....	13-34
13.4.1	Power Failure Sequence .....	13-35
13.4.1.1	Power Failure Impact on the System .....	13-35
13.4.1.2	System Assumptions .....	13-35
13.4.2	Memory Controller Response to Reset.....	13-36
13.4.2.1	External Logic Required for Power Failure .....	13-38
13.5	Interrupts/Error Conditions .....	13-39
13.5.1	Single-Bit Error Detection .....	13-40
13.5.2	Double-Bit/Nibble Error Detection.....	13-41
13.5.3	Overlapping Memory Regions .....	13-41
13.6	Register Definitions .....	13-42
13.6.1	SDRAM Initialization Register - SDIR .....	13-43
13.6.2	SDRAM Control Register - SDCR .....	13-44
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
<b>14</b>	<b>PCI-to-PCI Bridge .....</b>	<b>14-1</b>
14.1	Overview.....	14-1
14.2	Theory of Operation.....	14-2
14.3	Architectural Description.....	14-3
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.....	14-6
14.4.1	Type 0 Commands .....	14-8
14.4.2	Type 1 Commands and Type 1 to Type 0 Conversions .....	14-9
14.4.3	Type 1 to Type 1 Forwarding.....	14-10
14.4.4	Type 1 to Special Cycle Conversion.....	14-11
14.4.5	Private Type 0 Commands on the Secondary Interface .....	14-11

14.4.6	Special Cycles .....	14-13
14.5	Address Decoding .....	14-14
14.5.1	I/O Address Space .....	14-14
14.5.1.1	Disabling the I/O Address Range .....	14-15
14.5.1.2	ISA Mode .....	14-15
14.5.2	Memory Address Space .....	14-16
14.5.2.1	Burst Order .....	14-17
14.5.2.2	Disabling the Memory Address Range .....	14-17
14.5.3	64-Bit Address Decoding - Dual Address Cycles .....	14-18
14.5.4	Private Address Space .....	14-20
14.5.5	Secondary PCI to Messaging Unit Access .....	14-20
14.5.6	Address Decode Summary .....	14-21
14.6	Bridge Operation .....	14-23
14.6.1	PCI Interfaces .....	14-23
14.6.1.1	Primary Interface .....	14-23
14.6.1.2	Secondary Interface .....	14-24
14.6.2	Claiming a PCI Transaction .....	14-24
14.6.2.1	Latency Timers .....	14-24
14.6.2.2	Delayed Transactions .....	14-25
14.6.2.3	Posted Transactions .....	14-26
14.6.3	64-Bit Operation .....	14-26
14.6.3.1	64-Bit Protocol .....	14-27
14.6.3.2	64-Bit Operation with 32-Bit Targets .....	14-29
14.6.4	PCI Read Transactions .....	14-31
14.6.4.1	Read Streaming .....	14-36
14.6.4.2	Read Boundary .....	14-36
14.6.5	PCI Write Transactions .....	14-37
14.6.5.1	Delayed Write Transactions .....	14-37
14.6.5.2	Posted Write Transactions .....	14-38
14.6.5.3	Memory Write Command .....	14-39
14.6.5.4	Memory Write and Invalidate Command .....	14-39
14.6.5.5	I/O Write Command .....	14-40
14.6.5.6	Write Boundary .....	14-40
14.6.5.7	Qword Unaligned Memory Write Transactions .....	14-41
14.6.5.8	Fast Back to Back Transactions .....	14-41
14.7	Queue Architecture .....	14-41
14.7.1	Queue Operation .....	14-42
14.7.1.1	Upstream/Downstream Posted Memory Write Queue Structures .....	14-43
14.7.1.2	Upstream/Downstream Delayed Read Completion Queues .....	14-44
14.7.1.3	Upstream/Downstream Delayed Write Completion Queue .....	14-45
14.7.1.4	Upstream/Downstream Transaction Queues .....	14-46
14.7.2	Transaction Ordering .....	14-46
14.8	Bridge Data Flow .....	14-49
14.8.1	Delayed Read Transaction .....	14-49
14.8.2	Delayed Write Transaction .....	14-50
14.8.3	Posted Write Transaction .....	14-52
14.9	Exclusive Access .....	14-53
14.9.1	Secondary Interface Error Handling .....	14-54
14.10	PCI Transaction Termination .....	14-55



14.10.1	Termination as a Master (Initiator)	14-55
14.10.1.1	Completion	14-55
14.10.1.2	Time-out	14-55
14.10.1.3	Time-out during Memory Write and Invalidate	14-55
14.10.1.4	Master-Abort	14-55
14.10.2	Termination as a Slave (Target)	14-56
14.10.2.1	Retry	14-56
14.10.2.2	Disconnect	14-56
14.10.2.3	Target-Abort	14-57
14.11	Error Conditions	14-57
14.11.1	Address Parity Errors	14-57
14.11.1.1	Address Parity Errors on Primary Interface	14-58
14.11.1.2	Address Parity Errors on Secondary Interface	14-58
14.11.2	Data Parity Errors	14-59
14.11.2.1	Read Data Parity	14-59
14.11.2.2	Delayed Write Data Parity	14-60
14.11.2.3	Posted Write Data Parity	14-62
14.11.3	SERR# Assertion	14-64
14.11.4	Discard Timers	14-64
14.11.5	PCI-to-PCI Bridge Error Summary	14-65
14.12	Primary and Secondary Clocking	14-68
14.13	Initialization and Reset Requirements	14-68
14.13.1	Bridge Reset	14-69
14.13.2	Configuring the Bridge	14-69
14.13.3	64-Bit Bus Configuration	14-70
14.14	Powerup/Default States	14-70
14.15	Register Definitions	14-70
14.15.1	Vendor Identification Register - VIDR	14-73
14.15.2	Device ID Register - DIDR	14-74
14.15.3	Primary Command Register - PCR	14-75
14.15.4	Primary Status Register - PSR	14-76
14.15.5	Revision ID Register - RID	14-77
14.15.6	Class Code Register - CCR	14-77
14.15.7	Cacheline Size Register - CLSR	14-78
14.15.8	Primary Latency Timer Register - PLTR	14-79
14.15.9	Header Type Register - HTR	14-80
14.15.10	Primary Bus Number Register - PBNR	14-81
14.15.11	Secondary Bus Number Register - SBNR	14-82
14.15.12	Subordinate Bus Number Register - SubBNR	14-83
14.15.13	Secondary Latency Timer Register - SLTR	14-84
14.15.14	I/O Base Register - IOBR	14-85
14.15.15	I/O Limit Register - IOLR	14-86
14.15.16	Secondary Status Register - SSR	14-87
14.15.17	Memory Base Register - MBR	14-88
14.15.18	Memory Limit Register - MLR	14-89
14.15.19	Prefetchable Memory Base Register - PMBR	14-90
14.15.20	Prefetchable Memory Limit Register - PMLR	14-91
14.15.21	Bridge Subsystem Vendor ID Register - BSVIR	14-92
14.15.22	Bridge Subsystem ID Register - BSIR	14-92
14.15.23	Bridge Control Register - BCR	14-93
14.15.24	Extended Bridge Control Register - EBCR	14-96

14.15.25	Secondary IDSEL Select Register - SISR .....	14-99
14.15.26	Primary Bridge Interrupt Status Register - PBISR .....	14-101
14.15.27	Secondary Bridge Interrupt Status Register - SBISR .....	14-102
14.15.28	Secondary Arbitration Control Register - SACR .....	14-103
14.15.29	PCI Interrupt Routing Select Register - PIRSR .....	14-103
14.15.30	Secondary I/O Base Register - SIOBR .....	14-103
14.15.31	Secondary I/O Limit Register - SIOLR .....	14-104
14.15.32	Secondary Memory Base Register - SMBR .....	14-105
14.15.33	Secondary Memory Limit Register - SMLR .....	14-106
14.15.34	Secondary Decode Enable Register - SDER .....	14-107
14.15.35	Queue Control Register - QCR .....	14-109

<b>15</b>	<b>Address Translation Unit .....</b>	<b>15-1</b>
15.1	Overview .....	15-1
15.2	ATU Address Translation .....	15-3
15.2.1	Inbound Transactions .....	15-5
15.2.1.1	Inbound Address Translation .....	15-5
15.2.1.2	Inbound Write Transaction .....	15-8
15.2.1.3	Inbound Read Transaction .....	15-10
15.2.1.4	Inbound Configuration Cycle Translation .....	15-12
15.2.1.5	Discard Timers .....	15-13
15.2.2	Outbound Transactions .....	15-13
15.2.2.1	Outbound Address Translation .....	15-13
15.2.2.2	Outbound Address Translation Windows .....	15-14
15.2.2.3	Direct Addressing Window .....	15-17
15.2.2.4	Outbound Write Transaction .....	15-18
15.2.2.5	Outbound Read Transaction .....	15-19
15.2.3	Private PCI Address Space / Outbound Configuration Cycle Translation .....	15-20
15.2.4	PCI Multi-Function Device Swapping/Disabling .....	15-22
15.2.5	64-Bit PCI Operation .....	15-22
15.2.5.1	64-Bit Protocol .....	15-23
15.2.5.2	64-Bit Operation with 32-Bit Targets .....	15-25
15.3	Messaging Unit .....	15-27
15.4	Expansion ROM Translation Unit .....	15-27
15.5	ATU Queue Architecture .....	15-28
15.5.1	Inbound Queues .....	15-28
15.5.1.1	Inbound Write Queue Structure .....	15-28
15.5.1.2	Inbound Read Queues and Inbound Transaction Queues .....	15-29
15.5.1.3	Inbound Delayed Write Queue .....	15-30
15.5.2	Outbound Queues .....	15-31
15.5.3	Transaction Ordering .....	15-32
15.6	ATU Error Conditions .....	15-35
15.6.1	Address Parity Errors on the PCI Interface .....	15-35
15.6.2	Data Parity Errors on the PCI Interface .....	15-36
15.6.2.1	Outbound Read Data Parity Errors - Master .....	15-36
15.6.2.2	Outbound Write Data Parity Errors - Master .....	15-37
15.6.2.3	Inbound Read Data Parity Errors - Slave .....	15-37
15.6.2.4	Inbound Write Data Parity Errors - Slave .....	15-37
15.6.2.5	Inbound Configuration Write Data Parity Errors - Slave ...	15-38
15.6.3	Master Aborts on the PCI Interface .....	15-39





15.6.4	Target Aborts on the PCI Interface .....	15-39
15.6.5	SERR# Assertion and Detection.....	15-40
15.6.6	Internal Bus Error Conditions.....	15-41
15.6.6.1	Master Abort on the Internal Bus .....	15-42
15.6.6.2	Target Abort on the Internal Bus.....	15-43
15.6.7	ATU Error Summary .....	15-44
15.7	Register Definitions .....	15-47
15.7.1	ATU Vendor ID Register - ATUVID.....	15-51
15.7.2	ATU Device ID Register - ATUDID .....	15-52
15.7.3	Primary ATU Command Register - PATUCMD .....	15-53
15.7.4	Primary ATU Status Register - PATUSR .....	15-54
15.7.5	ATU Revision ID Register - ATURID .....	15-55
15.7.6	ATU Class Code Register - ATUCCR .....	15-56
15.7.7	ATU Cacheline Size Register - ATUCLSR .....	15-56
15.7.8	ATU Latency Timer Register - ATULT .....	15-57
15.7.9	ATU Header Type Register - ATUHTR.....	15-57
15.7.10	ATU BIST Register - ATUBISTR .....	15-58
15.7.11	Primary Inbound ATU Base Address Register - PIABAR .....	15-59
15.7.12	ATU Subsystem Vendor ID Register - ASVIR .....	15-60
15.7.13	ATU Subsystem ID Register - ASIR .....	15-60
15.7.14	Expansion ROM Base Address Register - ERBAR .....	15-61
15.7.15	Determining Block Sizes for Base Address Registers .....	15-62
15.7.16	ATU Interrupt Line Register - ATUILR .....	15-63
15.7.17	ATU Interrupt Pin Register - ATUIPR .....	15-64
15.7.18	ATU Minimum Grant Register - ATUMGNT.....	15-65
15.7.19	ATU Maximum Latency Register - ATUMLAT .....	15-66
15.7.20	Primary Inbound ATU Limit Register - PIALR.....	15-67
15.7.21	Primary Inbound ATU Translate Value Register - PIATVR .....	15-68
15.7.22	Secondary Inbound ATU Base Address Register - SIABAR .....	15-69
15.7.23	Secondary Inbound ATU Limit Register - SIALR.....	15-70
15.7.24	Secondary Inbound ATU Translate Value Register - SIATVR.....	15-71
15.7.25	Primary Outbound Memory Window Value Register - POMWVR.....	15-72
15.7.26	Primary Outbound I/O Window Value Register - POIOWVR.....	15-73
15.7.27	Primary Outbound DAC Window Value Register - PODWVR .....	15-74
15.7.28	Primary Outbound Upper 64-bit DAC Register - POUDR.....	15-75
15.7.29	Secondary Outbound Memory Window Value Register - SOMWVR.....	15-76
15.7.30	Secondary Outbound I/O Window Value Register - SOIOWVR ....	15-77
15.7.31	Expansion ROM Limit Register - ERLR.....	15-78
15.7.32	Expansion ROM Translate Value Register - ERTVR.....	15-79
15.7.33	ATU Configuration Register - ATUCR .....	15-80
15.7.34	Primary ATU Interrupt Status Register - PATUISR .....	15-82
15.7.35	Secondary ATU Interrupt Status Register - SATUISR.....	15-84
15.7.36	Secondary ATU Command Register - SATUCMD .....	15-86
15.7.37	Secondary ATU Status Register - SATUSR .....	15-87
15.7.38	Secondary Outbound DAC Window Value Register - SODWVR ..	15-88
15.7.39	Secondary Outbound Upper 64-bit DAC Register - SOUDR.....	15-89
15.7.40	Primary Outbound Configuration Cycle Address Register - POCCAR.....	15-90

15.7.41	Secondary Outbound Configuration Cycle Address Register - SOCCAR.....	15-91
15.7.42	Primary Outbound Configuration Cycle Data Register - POCCDR.....	15-92
15.7.43	Secondary Outbound Configuration Cycle Data Register - SOCCDR.....	15-93
15.7.44	Primary ATU Interrupt Mask Register - PATUIMR .....	15-94
15.7.45	Secondary ATU Interrupt Mask Register - SATUIMR.....	15-95

**16 Messaging Unit ..... 16-1**

16.1	Overview .....	16-1
16.2	Theory of Operation .....	16-1
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.....	16-11
16.5.3	Outbound Post Queue.....	16-12
16.5.4	Outbound Free Queue.....	16-13
16.6	Index Registers .....	16-14
16.7	Messaging Unit Error Conditions.....	16-14
16.8	Register Definitions .....	16-15
16.8.1	Inbound Message Register - IMRx.....	16-17
16.8.2	Outbound Message Register - OMRx.....	16-17
16.8.3	Inbound Doorbell Register - IDR.....	16-18
16.8.4	Inbound Interrupt Status Register - IISR .....	16-19
16.8.5	Inbound Interrupt Mask Register - IIMR.....	16-20
16.8.6	Outbound Doorbell Register - ODR .....	16-21
16.8.7	Outbound Interrupt Status Register - OISR.....	16-22
16.8.8	Outbound Interrupt Mask Register - OIMR .....	16-23
16.8.9	MU Configuration Register - MUCR .....	16-24
16.8.10	Queue Base Address Register - QBAR.....	16-25
16.8.11	Inbound Free Head Pointer Register - IFHPR .....	16-26
16.8.12	Inbound Free Tail Pointer Register - IFTPR .....	16-27
16.8.13	Inbound Post Head Pointer Register - IPHPR .....	16-28
16.8.14	Inbound Post Tail Pointer Register - IPTPR .....	16-29
16.8.15	Outbound Free Head Pointer Register - OFHPR .....	16-30
16.8.16	Outbound Free Tail Pointer Register - OFTPR.....	16-31
16.8.17	Outbound Post Head Pointer Register - OPHPR .....	16-32
16.8.18	Outbound Post Tail Pointer Register - OPTPR.....	16-33
16.8.19	Index Address Register - IAR .....	16-34
16.9	Power/Default Status.....	16-34

**17 i960<sup>®</sup> RM/RN 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	Arbitration Signalling Protocol.....	17-5
17.2.1.3	Secondary PCI Bus Arbitration Parking.....	17-7
17.2.2	Atomic Accesses .....	17-7
17.2.3	Internal and Secondary PCI Arbiter Differences.....	17-8
17.2.3.1	Multi-Transaction Timer .....	17-8
17.3	PCI Selector Operation.....	17-9
17.3.1	Primary PCI Bus Arbitration Parking.....	17-9
17.4	Master Latency Timer Operation .....	17-9
17.4.1	Primary and Secondary PCI Master Latency Timers.....	17-9
17.4.2	Internal Master Latency Timer .....	17-9
17.5	Reset Conditions .....	17-10
17.5.1	S_REQ64# Control .....	17-10
17.6	Register Definitions .....	17-11
17.6.1	Secondary Arbitration Control Register - SACR .....	17-12
17.6.2	Internal Arbitration Control Register - IACR.....	17-13
17.6.3	Master Latency Timer Register - MLTR.....	17-14
17.6.4	Multi-Transaction Timer Register - MTTR .....	17-14
<b>18</b>	<b>Timers .....</b>	<b>18-1</b>
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
<b>19</b>	<b>DMA Controller Unit .....</b>	<b>19-1</b>
19.1	Overview.....	19-1
19.2	Theory of Operation.....	19-2
19.3	DMA Transfer .....	19-3
19.3.1	Chain Descriptors .....	19-4
19.3.2	Initiating DMA Transfers .....	19-6
19.3.3	Scatter Gather DMA Transfers .....	19-7
19.3.4	Synchronizing a Program to Chained Transfers.....	19-8
19.3.5	Appending to The End of a Chain.....	19-9
19.4	64-bit Transfers on a 64-bit PCI Bus .....	19-10
19.4.1	64-bit Operation with 64-bit Targets .....	19-11
19.4.2	64-bit Operation with 32-bit Targets .....	19-11

19.4.3	64-bit Addressing.....	19-11
19.5	Data Transfers.....	19-12
19.5.1	PCI to Local Memory Transfers.....	19-12
19.5.2	Local Memory to PCI Transfers: Memory Write Command.....	19-12
19.5.3	Local Memory to PCI Transfers: Memory Write and Invalidate Command.....	19-12
19.5.4	Exclusive Access.....	19-13
19.6	Data Queues.....	19-13
19.7	Packing and Unpacking.....	19-13
19.7.1	64-bit Unaligned Data Transfers.....	19-14
19.7.2	64/32-bit Unaligned Data Transfers.....	19-15
19.8	Channel Priority.....	19-16
19.9	Programming Model State Diagram.....	19-16
19.10	DMA Channel Programming Examples.....	19-17
19.10.1	Software DMA Controller Initialization.....	19-17
19.10.2	Software Start DMA Transfer.....	19-17
19.10.3	Software Suspend Channel.....	19-18
19.11	Interrupts.....	19-18
19.12	Error Conditions.....	19-19
19.12.1	PCI Errors.....	19-19
19.12.2	Internal Bus Errors.....	19-20
19.13	Powerup/Default Status.....	19-21
19.14	Register Definitions.....	19-21
19.14.1	Channel Control Register - CCR.....	19-22
19.14.2	Channel Status Register - CSR.....	19-23
19.14.3	Next Descriptor Address Register - NDAR.....	19-25
19.14.4	Descriptor Address Register - DAR.....	19-26
19.14.5	Byte Count Register - BCR.....	19-27
19.14.6	PCI Address Register - PADR.....	19-28
19.14.7	PCI Upper Address Register - PUADR.....	19-29
19.14.8	Local Address Register - LADR.....	19-30
19.14.9	Descriptor Control Register - DCR.....	19-31
<b>20</b>	<b>Application Accelerator Unit.....</b>	<b>20-1</b>
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 (4 Source Addresses).....	20-3
20.3.3	Chain Descriptor Format (Eight Source Addresses).....	20-6
20.3.4	The Bitwise-XOR Algorithm.....	20-8
20.3.5	Initiating the XOR Operation.....	20-11
20.3.6	Scatter Gather Transfers.....	20-12
20.3.7	Synchronizing a program to Chained operation.....	20-12
20.3.8	Appending to The End of a Chain.....	20-14
20.4	Store Queue.....	20-14
20.5	Packing and Unpacking.....	20-15
20.5.1	64-bit Unaligned Data Transfers.....	20-15
20.6	Application Accelerator Unit Priority.....	20-16
20.7	Programming Model State Diagram.....	20-16



20.8	Programming the Application Accelerator Unit.....	20-17
20.8.1	Application Accelerator Unit Initialization.....	20-17
20.8.2	Start XOR Transfer.....	20-17
20.8.3	Suspend Application Accelerator Unit.....	20-18
20.9	Interrupts.....	20-18
20.9.1	Interrupts - Special Case (ADCR.dwe = 0).....	20-19
20.10	Error Conditions.....	20-19
20.11	Powerup/Default Status.....	20-20
20.12	Register Definitions.....	20-20
20.12.1	Accelerator Control Register - ACR.....	20-21
20.12.2	Accelerator Status Register - ASR.....	20-22
20.12.3	Accelerator Descriptor Address Register - ADAR.....	20-23
20.12.4	Accelerator Next Descriptor Address Register - ANDAR.....	20-24
20.12.5	80960 Source Address Register - SAR.....	20-25
20.12.6	80960 Destination Address Register - DAR.....	20-26
20.12.7	Accelerator Byte Count Register - ABCR.....	20-27
20.12.8	Accelerator Descriptor Control Register - ADCR.....	20-28

<b>21</b>	<b>Performance Monitoring Unit.....</b>	<b>21-1</b>
21.1	Overview.....	21-1
21.2	Theory of Operation.....	21-1
21.2.1	Global Time Stamp.....	21-1
21.2.2	Programmable Event Counters.....	21-2
21.2.2.1	Occurrence Events.....	21-2
21.2.2.2	Duration Events.....	21-3
21.2.3	Performance Monitoring.....	21-3
21.3	Event Description.....	21-4
21.3.1	Mode0: Performance Monitoring Disabled.....	21-4
21.3.2	Mode1: Primary PCI bus and Internal Agents.....	21-5
21.3.2.1	M1_PPCIBus_idle.....	21-5
21.3.2.2	M1_PPCIBus_data.....	21-5
21.3.2.3	M1_PPCIBus_bridge_acq.....	21-5
21.3.2.4	M1_PPCIBus_bridge_own.....	21-5
21.3.2.5	M1_PPCIBus_DMA0_acq.....	21-5
21.3.2.6	M1_PPCIBus_DMA0_own.....	21-5
21.3.2.7	M1_PPCIBus_DMA1_acq.....	21-6
21.3.2.8	M1_PPCIBus_DMA1_own.....	21-6
21.3.2.9	M1_PPCIBus_PATU_acq.....	21-6
21.3.2.10	M1_PPCIBus_PATU_own.....	21-6
21.3.2.11	M1_PPCIBus_DMA0_gnt.....	21-6
21.3.2.12	M1_PPCIBus_DMA1_gnt.....	21-6
21.3.2.13	M1_PPCIBus_PATU_gnt.....	21-6
21.3.2.14	M1_PPCIBus_bridge_gnt.....	21-7
21.3.3	Mode 2: Secondary PCI Bus and Internal Agents.....	21-7
21.3.3.1	M2_SPCIBus_idle.....	21-7
21.3.3.2	M2_SPCIBus_data.....	21-7
21.3.3.3	M2_SPCIBus_SATU_acq.....	21-7
21.3.3.4	M2_SPCIBus_SATU_own.....	21-7
21.3.3.5	M2_SPCIBus_bridge_acq.....	21-7
21.3.3.6	M2_SPCIBus_bridge_own.....	21-8
21.3.3.7	M2_SPCIBus_DMA2_acq.....	21-8
21.3.3.8	M2_SPCIBus_DMA2_own.....	21-8

21.3.3.9	M2_SPCIBus_bridge_gnt .....	21-8
21.3.3.10	M2_SPCIBus_SATU_gnt.....	21-8
21.3.3.11	M2_SPCIBus_DMA2_gnt .....	21-8
21.3.3.12	M2_PPCCIBus_idle.....	21-8
21.3.3.13	M2_PPCCIBus_data .....	21-9
21.3.3.14	M2_IBus_data.....	21-9
21.3.4	Mode 3: Secondary PCI Bus and External Agents .....	21-9
21.3.4.1	M3_SPCIBus_idle .....	21-9
21.3.4.2	M3_SPCIBus_data.....	21-9
21.3.4.3	M3_SPCIBus_IOP_acq.....	21-9
21.3.4.4	M3_SPCIBus_IOP_own.....	21-10
21.3.4.5	M3_SPCIBus_D0_acq .....	21-10
21.3.4.6	M3_SPCIBus_D0_own .....	21-10
21.3.4.7	M3_SPCIBus_D1_acq .....	21-10
21.3.4.8	M3_SPCIBus_D1_own .....	21-10
21.3.4.9	M3_SPCIBus_D2_acq .....	21-10
21.3.4.10	M3_SPCIBus_D2_own .....	21-10
21.3.4.11	M3_SPCIBus_IOP_gnt .....	21-11
21.3.4.12	M3_SPCIBus_D0_gnt.....	21-11
21.3.4.13	M3_SPCIBus_D1_gnt.....	21-11
21.3.4.14	M3_SPCIBus_D2_gnt.....	21-11
21.3.5	Mode 4: Secondary PCI Bus and External Agents .....	21-11
21.3.5.1	M4_SPCIBus_idle .....	21-11
21.3.5.2	M4_SPCIBus_data.....	21-12
21.3.5.3	M4_SPCIBus_D3_acq .....	21-12
21.3.5.4	M4_SPCIBus_D3_own .....	21-12
21.3.5.5	M4_SPCIBus_D4_acq .....	21-12
21.3.5.6	M4_SPCIBus_D4_own .....	21-12
21.3.5.7	M4_SPCIBus_D5_acq .....	21-12
21.3.5.8	M4_SPCIBus_D5_own .....	21-12
21.3.5.9	M4_SPCIBus_D3_gnt.....	21-13
21.3.5.10	M4_SPCIBus_D4_gnt.....	21-13
21.3.5.11	M4_SPCIBus_D5_gnt.....	21-13
21.3.5.12	M4_SPCIBus_IOP_gnt .....	21-13
21.3.5.13	M4_SPCIBus_IOP_acq.....	21-13
21.3.5.14	M4_SPCIBus_IOP_own.....	21-13
21.3.6	Mode 5: i960 <sup>®</sup> RM/RN I/O Processor Internal Bus and Agents Events .....	21-14
21.3.6.1	M5_IBus_idle .....	21-14
21.3.6.2	M5_IBus_data.....	21-14
21.3.6.3	M5_IBus_AAU_acq .....	21-14
21.3.6.4	M5_IBus_AAU_own.....	21-14
21.3.6.5	M5_IBus_DMA0_acq.....	21-14
21.3.6.6	M5_IBus_DMA0_own.....	21-14
21.3.6.7	M5_IBus_DMA1_acq.....	21-15
21.3.6.8	M5_IBus_DMA1_own.....	21-15
21.3.6.9	M5_IBus_DMA2_acq.....	21-15
21.3.6.10	M5_IBus_DMA2_own.....	21-15
21.3.6.11	M5_IBus_AAU_gnt .....	21-15
21.3.6.12	M5_IBus_DMA0_gnt.....	21-15
21.3.6.13	M5_IBus_DMA1_gnt.....	21-15
21.3.6.14	M5_IBus_DMA2_gnt.....	21-15
21.3.7	Mode 6: i960 <sup>®</sup> RM/RN I/O Processor Internal Bus and Agents Events .....	21-16



21.3.7.1	M6_IBus_core_acq.....	21-16
21.3.7.2	M6_IBus_core_own.....	21-16
21.3.7.3	M6_IBus_PATU_acq.....	21-16
21.3.7.4	M6_IBus_PATU_own.....	21-16
21.3.7.5	M6_IBus_SATU_acq.....	21-16
21.3.7.6	M6_IBus_SATU_own.....	21-16
21.3.7.7	M6_IBus_PBOFF_time.....	21-17
21.3.7.8	M6_IBus_PBOFF_cnt.....	21-17
21.3.7.9	M6_IBus_SBOFF_time.....	21-17
21.3.7.10	M6_IBus_SBOFF_cnt.....	21-17
21.3.7.11	M6_IBus_PATU_gnt.....	21-17
21.3.7.12	M6_IBus_SATU_gnt.....	21-17
21.3.7.13	M6_IBus_core_gnt.....	21-17
21.3.7.14	M6_IBus_ATU_retries.....	21-17
21.3.8	Mode 7: i960 <sup>®</sup> RM/RN Processor Internal Bus, Secondary PCI Bus and Primary PCI Bus Events.....	21-18
21.3.8.1	M7_IBus_idle.....	21-18
21.3.8.2	M7_IBus_data.....	21-18
21.3.8.3	M7_SPClbus_idle.....	21-18
21.3.8.4	M7_SPClbus_data.....	21-18
21.3.8.5	M7_SPClbus_IOP_own.....	21-18
21.3.8.6	M7_SPClbus_D0_own.....	21-19
21.3.8.7	M7_SPClbus_D1_own.....	21-19
21.3.8.8	M7_SPClbus_D2_own.....	21-19
21.3.8.9	M7_SPClbus_D3_own.....	21-19
21.3.8.10	M7_SPClbus_D4_own.....	21-19
21.3.8.11	M7_SPClbus_D5_own.....	21-19
21.3.8.12	M7_PPCLbus_IOP_own.....	21-19
21.3.8.13	M7_PPCLbus_idle.....	21-19
21.3.8.14	M7_PPCLbus_data.....	21-19
21.4	Interrupts.....	21-20
21.5	Reset Conditions.....	21-20
21.6	Register Definitions.....	21-20
21.6.1	Global Timer Mode Register (GTMR).....	21-21
21.6.2	Event Select Register (ESR).....	21-22
21.6.3	Event Monitoring Interrupt Status Register (EMISR).....	21-23
21.6.4	Global Time Stamp Register (GTSR).....	21-24
21.6.5	Programmable Event Counter Register (PECRx).....	21-25
<b>22</b>	<b>I<sup>2</sup>C Bus Interface Unit.....</b>	<b>22-1</b>
22.1	Overview.....	22-1
22.2	Theory of Operation.....	22-1
22.2.1	Operational Blocks.....	22-2
22.2.2	I <sup>2</sup> C 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.....	22-6
22.3	I <sup>2</sup> C Bus Operation.....	22-7
22.3.1	Serial Clock Line (SCL) Generation.....	22-7
22.3.2	Data and Addressing Management.....	22-8
22.3.2.1	Addressing a Slave Device.....	22-9
22.3.3	I <sup>2</sup> C Acknowledge.....	22-10

22.3.4	Arbitration .....	22-11
22.3.4.1	SCL Arbitration .....	22-11
22.3.4.2	SDA Arbitration .....	22-12
22.3.5	Master Operations .....	22-13
22.3.6	Slave Operations .....	22-16
22.3.7	General Call Address .....	22-18
22.4	Slave Mode Programming Examples .....	22-19
22.4.1	Initialize Unit .....	22-19
22.4.2	Write 1 bytes as a slave .....	22-19
22.4.3	Read 2 bytes as a Slave .....	22-19
22.5	Master Programming Examples .....	22-20
22.5.1	Initialize Unit .....	22-20
22.5.2	Write 1 byte as a master .....	22-20
22.5.3	Read 1 byte as a master .....	22-20
22.5.4	Write 2 bytes and repeated start read 1 byte as a master .....	22-21
22.5.5	Read 2 bytes as a Master - Send STOP using the Abort .....	22-22
22.6	Glitch Suppression Logic .....	22-23
22.7	Reset Conditions .....	22-23
22.8	Register Definitions .....	22-23
22.8.1	I <sup>2</sup> C Control Register- ICR .....	22-24
22.8.2	I <sup>2</sup> C Status Register- ISR .....	22-27
22.8.3	I <sup>2</sup> C Slave Address Register- ISAR .....	22-29
22.8.4	I <sup>2</sup> C Data Buffer Register- IDBR .....	22-30
22.8.5	I <sup>2</sup> C Clock Count Register- ICCR .....	22-31
22.8.6	I <sup>2</sup> C Bus Monitor Register- IBMR .....	22-32

<b>23</b>	<b>Test Features .....</b>	<b>23-1</b>
23.1	On-Circuit Emulation (ONCE) .....	23-1
23.1.1	Entering/Exiting ONCE Mode .....	23-1
23.1.2	ONCE Mode and Boundary-Scan (JTAG) are Incompatible .....	23-2
23.1.2.1	DEN# Alternatives .....	23-2
23.2	Boundary-Scan (JTAG) .....	23-2
23.2.1	Boundary-Scan Architecture .....	23-2
23.2.2	TAP Pins .....	23-3
23.2.3	Instruction Register .....	23-4
23.2.3.1	Boundary-Scan Instruction Set .....	23-4
23.2.4	TAP Test Data Registers .....	23-6
23.2.4.1	Device Identification Register .....	23-6
23.2.4.2	Bypass Register .....	23-6
23.2.4.3	RUNBIST Register .....	23-6
23.2.4.4	Boundary-Scan Register .....	23-6
23.2.5	TAP Controller .....	23-16
23.2.5.1	Test Logic Reset State .....	23-17
23.2.5.2	Run-Test/Idle State .....	23-17
23.2.5.3	Select-DR-Scan State .....	23-17
23.2.5.4	Capture-DR State .....	23-17
23.2.5.5	Shift-DR State .....	23-17
23.2.5.6	Exit1-DR State .....	23-18
23.2.5.7	Pause-DR State .....	23-18
23.2.5.8	Exit2-DR State .....	23-18
23.2.5.9	Update-DR State .....	23-18





	23.2.5.10	Select-IR Scan State .....	23-18
	23.2.5.11	Capture-IR State .....	23-19
	23.2.5.12	Shift-IR State .....	23-19
	23.2.5.13	Exit1-IR State.....	23-19
	23.2.5.14	Pause-IR State .....	23-19
	23.2.5.15	Exit2-IR State.....	23-19
	23.2.5.16	Update-IR State .....	23-19
	23.2.6	Boundary-Scan Example .....	23-20
<b>24</b>	<b>Clocking and Reset</b> .....		<b>24-1</b>
24.1	Clocking Overview .....		24-1
24.1.1	Clocking Theory of Operation .....		24-1
24.1.2	Clocking Region 1.....		24-2
24.1.3	Clocking Region 2.....		24-2
24.1.4	Clocking Region 3.....		24-3
24.1.5	Clocking Region Summary .....		24-3
24.2	Reset Overview .....		24-3
24.2.1	Primary PCI Reset .....		24-5
24.2.2	Secondary PCI Reset .....		24-6
24.2.3	Internal Bus Reset .....		24-6
24.3	Reset Strapping Options .....		24-7
<b>A</b>	<b>Machine-Level Instruction Formats</b> .....		<b>A-1</b>
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>B</b>	<b>Opcodes and Execution Times</b> .....		<b>B-1</b>
B.1	Instruction Reference by Opcode .....		B-1
<b>C</b>	<b>Memory-Mapped Registers</b> .....		<b>C-1</b>
C.1	Overview.....		C-1
C.2	Supervisor Space Family Registers and Tables.....		C-1
C.3	Peripheral Memory-Mapped Register Address Space .....		C-4
C.4	Accessing The Peripheral Memory-Mapped Registers .....		C-5
C.5	Architecturally Reserved Memory Space .....		C-6
C.6	Peripheral Memory-Mapped Register Address Space .....		C-7
	<b>Index</b> .....		<b>index-1</b>

## Figures

1-1	i960 <sup>®</sup> RM/RN I/O Processor Functional Block Diagram.....	1-1
1-2	80960JT Core Processor Block Diagram.....	1-6
2-1	Data Types and Ranges .....	2-1
3-1	i960 <sup>®</sup> RM/RN I/O Processor Programming Environment.....	3-2
3-2	Local Memory Address Space .....	3-10
4-1	Internal Data RAM and Register Cache.....	4-1
5-1	Machine-Level Instruction Formats .....	5-2
6-1	dcctl <i>src1</i> and <i>src/dst</i> 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 <i>src1</i> and <i>src/dst</i> Formats .....	6-50
6-5	Store Instruction Cache to Memory Output Format .....	6-51
6-6	I-Cache Set Data, Tag and Valid Bit Formats .....	6-52
6-7	<i>Src1</i> Operand Interpretation.....	6-96
6-8	<i>src/dst</i> Interpretation for Breakpoint Resource Request .....	6-97
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.....	7-15
7-5	Previous Frame Pointer Register – PFP .....	7-18
8-1	Interrupt Handling Data Structures.....	8-2
8-2	Interrupt Table.....	8-4
8-3	Storage of an Interrupt Record on the Interrupt Stack .....	8-6
8-4	Interrupt Controller .....	8-12
8-5	Interrupt Pin Vector Assignment .....	8-13
8-6	Interrupt Fast Sampling.....	8-14
8-7	Interrupt Controller Connections .....	8-19
8-8	Interrupt Service Flowchart .....	8-27
9-1	Fault-Handling Data Structures.....	9-1
9-2	Fault Table and Fault Table Entries .....	9-5
9-3	Fault Record.....	9-7
9-4	Storage of the Fault Record on the Stack .....	9-8
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).....	11-10
11-5	Control Table.....	11-18
12-1	LMCON Example .....	12-4
12-2	Core Processor/BIU Interface Block Diagram.....	12-9
12-3	Internal Block Diagram.....	12-10
13-1	4 Mbyte Flash Memory System.....	13-4
13-2	90 ns Flash Read Cycle.....	13-6
13-3	60 ns Flash Burst Read Cycle.....	13-7
13-4	90 ns Flash Write Cycle .....	13-8
13-5	Dual-Bank SDRAM Memory Subsystem.....	13-10
13-6	Logical Memory Image of a 16 Mbit SDRAM Memory Subsystem .....	13-15
13-7	Logical Memory Image of a 64 Mbit SDRAM Memory Subsystem .....	13-16
13-8	Supported SDRAM Mode Register Settings .....	13-18



13-9	SDRAM Initialization Sequence (controlled with software)	13-19
13-10	SDRAM Read, 40 bytes, ECC Enabled, Page Hit	13-21
13-11	SDRAM Read, 40 bytes, ECC Enabled, Page Miss	13-23
13-12	SDRAM Write, 40 bytes, ECC Enabled, Page Hit	13-25
13-13	SDRAM Write, 40 bytes, ECC Enabled, Page Miss	13-26
13-14	Refresh Following a Read Cycle	13-28
13-15	Sub 64-bit SDRAM Write (D <sub>1</sub> )	13-30
13-16	SDRAM Clocking	13-33
13-17	Power Failure Sequence	13-35
13-18	Power Failure State Machine	13-36
13-19	Power Failure Sequence	13-37
13-20	External Power Failure State Machine	13-38
13-21	External Power Failure Logic in the System	13-38
14-1	PCI-to-PCI Bridge Unit Block Diagram	14-2
14-2	Bridge Operation	14-3
14-3	PCI Configuration Access Formats	14-7
14-4	Secondary IDSEL Example	14-13
14-5	ISA Mode Address Decode	14-15
14-6	Overlapping Memory Address Ranges	14-16
14-7	64-bit Dual Address Read Cycle	14-19
14-8	PCI 64-Bit Transfer to a 64-Bit Target	14-28
14-9	64-Bit Write Request with 32-Bit Transfer	14-30
14-10	Downstream Data Path Queue Completion	14-47
14-11	Bridge Configuration Header Format	14-71
14-12	Primary Status Register - PSR	14-76
15-1	ATU Block Diagram	15-2
15-2	ATU Queue Architecture Block Diagram	15-3
15-3	Inbound Address Detection	15-6
15-4	Inbound Translation Example	15-7
15-5	80960 Memory Map - Outbound Translation Window	15-15
15-6	Outbound Address Translation Windows	15-16
15-7	Direct Addressing Window	15-17
15-8	PCI 64-Bit Transfer from a 64-Bit Target	15-24
15-9	64-Bit Write Request with 32-Bit Transfer	15-26
15-10	Inbound Queue Completion	15-34
15-11	ATU Interface Configuration Header Format	15-47
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	i960 <sup>®</sup> RM/RN I/O Processor Arbitration Block Diagram	17-1
17-2	Secondary PCI Arbitration Example	17-3
17-3	Arbitration Between Two Masters	17-5
17-4	BIU Back-to-Back Transactions with MTT enabled	17-8
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	19-2
19-3	DMA Chain Descriptor	19-4
19-4	DMA Chaining Operation	19-5
19-5	Example of Gather Chaining	19-7

19-6	Synchronizing to Chained Transfers .....	19-9
19-7	Optimization of an Unaligned DMA .....	19-14
19-8	Optimization of an Unaligned DMA .....	19-15
19-9	DMA Programming Model State Diagram .....	19-16
20-1	Application Accelerator Unit .....	20-1
20-2	Application Accelerator Unit Block Diagram.....	20-2
20-3	Chain Descriptor Format .....	20-4
20-4	XOR Chaining Operation .....	20-5
20-5	Chain Descriptor Format for 8 Source Addresses (XOR Function) .....	20-7
20-6	XOR Chaining Operation .....	20-8
20-7	The Bit-wise XOR Algorithm .....	20-9
20-8	Hardware Assist XOR Unit.....	20-10
20-9	Example of Gather Chaining for Four Source Blocks .....	20-12
20-10	Synchronizing to Chained XOR Operation.....	20-13
20-11	Optimization of an Unaligned Data Transfer .....	20-15
20-12	Application Accelerator Unit Programming Model State Diagram.....	20-16
22-1	I <sup>2</sup> C Bus Configuration Example .....	22-2
22-2	I <sup>2</sup> C 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 .....	22-9
22-6	Acknowledge on the I <sup>2</sup> C Bus .....	22-10
22-7	Clock Synchronization During the Arbitration Procedure .....	22-11
22-8	Arbitration Procedure of Two Masters.....	22-12
22-9	Master-Receiver Read from Slave-Transmitter.....	22-15
22-10	Master-Receiver Read from Slave-Transmitter / Repeated Start / Master-Transmitter Write to Slave-Receiver .....	22-15
22-11	A Complete Data Transfer .....	22-15
22-12	Master-Transmitter Write to Slave-Receiver .....	22-17
22-13	Master-Receiver Read to Slave-Transmitter .....	22-17
22-14	Master-Receiver Read to Slave-Transmitter, Repeated START, Master-Transmitter Write to Slave-Receiver .....	22-17
22-15	General Call Address .....	22-18
23-1	Test Access Port Block Diagram.....	23-3
23-2	TAP Controller State Diagram.....	23-16
23-3	Example Showing Typical JTAG Operations .....	23-21
23-4	Timing Diagram Illustrating the Loading of Instruction Register.....	23-22
23-5	Timing Diagram Illustrating the Loading of Data Register.....	23-23
24-1	Clocking Regions Diagram.....	24-1
24-2	SDRAM Clocking Diagram .....	24-2
24-3	Reset Block Diagram .....	24-4
A-1	Instruction Formats .....	A-1
C-2	i960 <sup>®</sup> RM/RN I/O Processor Address Space .....	C-6



## Tables

1-1	Additional Information Sources .....	1-12
1-2	Electronic Information.....	1-12
2-1	80960 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	Arithmetic Controls Register – AC.....	3-14
3-6	Condition Codes for True or False Conditions .....	3-15
3-7	Condition Codes for Equality and Inequality Conditions .....	3-15
3-8	Condition Codes for Carry Out and Overflow.....	3-15
3-9	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	i960 <sup>®</sup> RM/RN I/O Processor Instruction Set.....	5-4
5-3	Arithmetic Operations.....	5-7
6-1	Pseudo-Code Symbol Definitions.....	6-3
6-2	Faults Applicable to All Instructions.....	6-3
6-3	Common Faulting Conditions .....	6-4
6-4	Condition Code Mask Descriptions .....	6-6
6-5	concmpo Example: Register Ordering and CC .....	6-32
6-6	dcctl Operand Fields .....	6-33
6-7	dcctl Status Values and D-Cache Parameters .....	6-34
6-8	icctl Operand Fields.....	6-49
6-9	icctl Status Values and I-Cache Parameters.....	6-51
6-10	sysctl Field Definitions.....	6-96
6-11	Cache Mode Configuration.....	6-96
7-1	Encodings of Entry Type Field in System Procedure Table .....	7-15
7-2	Encoding of Return Status Field.....	7-18
8-1	Interrupt Input Pin Descriptions .....	8-20
8-2	PCI Interrupt Routing Summary .....	8-21
8-3	XINT6# Interrupt Sources.....	8-22
8-4	XINT7 Interrupt Sources.....	8-22
8-5	NMI Interrupt Sources .....	8-24
8-6	Default Interrupt Routing and Status Values .....	8-25
8-7	Location of Cached Vectors in Internal RAM .....	8-28
8-8	Base Interrupt Latency .....	8-29
8-9	Worst-Case Interrupt Latency Controlled by divo to Destination r15 .....	8-30
8-10	Worst-Case Interrupt Latency Controlled by divo to Destination r3 .....	8-30
8-11	Worst-Case Interrupt Latency Controlled by calls .....	8-30
8-12	Worst-Case Interrupt Latency When Delivering a Software Interrupt .....	8-31
8-13	Worst-Case Interrupt Latency Controlled by flushreg of One Stack Frame .....	8-31
8-14	Interrupt Control Registers Addresses .....	8-32
8-15	Interrupt Control (ICON) Register.....	8-33
8-16	Interrupt Map Register 0 (IMAP0) .....	8-34
8-17	Interrupt Map Register 1 (IMAP1) .....	8-35
8-18	Interrupt Map Register 2 (IMAP2) .....	8-35

8-19	Interrupt Pending (IPND) Register .....	8-37
8-20	Interrupt Mask (IMSK) Register .....	8-38
8-21	PCI Interrupt Routing Select Register (PIRSR).....	8-39
8-22	XINT6 Interrupt Status Register- X6ISR .....	8-40
8-23	XINT7 Interrupt Status Register- X7ISR .....	8-41
8-24	NMI Interrupt Status Register- NISR .....	8-42
9-1	i960 <sup>®</sup> RM/RN I/O Processor Fault Types and Subtypes.....	9-3
9-2	Fault Control Bits and Masks .....	9-15
10-1	80960RM/RN Trace Controls Register – TC.....	10-2
10-2	src/dst Encoding.....	10-6
10-3	Breakpoint Control Register – BPCON .....	10-7
10-4	Configuring the Data Address Breakpoint Registers – DABx .....	10-8
10-5	Programming the Data Address Breakpoint Modes – DABx.....	10-8
10-6	Data Address Breakpoint Register – DABx.....	10-9
10-7	Instruction Breakpoint Register – IPBx .....	10-10
10-8	Instruction Breakpoint Modes.....	10-10
10-9	Tracing on Explicit Call.....	10-12
10-10	Tracing on Implicit Call.....	10-13
10-11	Tracing on Return from Explicit Call.....	10-14
11-1	Initialization Modes.....	11-2
11-2	Reset Values .....	11-5
11-3	BIST Failure Codes.....	11-8
11-4	Non-BIST Failure Codes .....	11-9
11-5	Initialization Boot Record .....	11-11
11-6	PMCON14_15 Register Bit Description in IBR .....	11-13
11-7	PRCB Configuration .....	11-14
11-8	Process Control Block Configuration Words .....	11-15
11-9	Processor Device ID Register - PDIDR.....	11-19
11-10	i960 <sup>®</sup> Core Processor Device ID Register - DEVICEID .....	11-19
12-1	PMCON Address Mapping.....	12-1
12-2	Physical Memory Control Registers – PMCON0:15.....	12-2
12-3	Bus Control Register – BCON.....	12-3
12-4	Logical Memory Address Registers – LMADR0:1 .....	12-5
12-5	Logical Memory Mask Registers – LMMR0:1 .....	12-6
12-6	Default Logical Memory Configuration Register – DLMCON .....	12-6
12-7	Bus Interface Unit Register Table .....	12-16
12-8	BIU Control Register - BIUCR .....	12-16
12-9	BIU Interrupt Status Register - BIUISR .....	12-17
13-1	Flash Interface Signals.....	13-3
13-2	Address Decoding for Flash Memory Space.....	13-5
13-3	Flash Wait State Profile Programming .....	13-7
13-4	SDRAM Memory Configuration Options .....	13-9
13-5	SDRAM Interface Signals .....	13-9
13-6	Supported SDRAM Configurations .....	13-11
13-7	SDRAM Address Register Definitions.....	13-11
13-8	Address Decoding for SDRAM Memory Space .....	13-12
13-9	Programming Values for the SDRAM Boundary Registers (SBRx[5:0]) .....	13-12
13-10	SDRAM Address Translation for 16 Mbit Devices .....	13-13
13-11	SDRAM Address Translation for 64 Mbit Devices .....	13-13
13-12	SDRAM Commands.....	13-17



13-13	Syndrome Decoding.....	13-31
13-14	MCU Error Response.....	13-39
13-15	Overlapping Address Priorities.....	13-41
13-16	Memory Controller Register Reference.....	13-42
13-17	SDRAM Initialization Register - SDIR.....	13-43
13-18	SDRAM Control Register - SDCR.....	13-44
13-19	Drive Strength Programmability Options (16-Mbit SDRAM Technology).....	13-45
13-20	Drive Strength Programmability Options (64-Mbit SDRAM Technology).....	13-46
13-21	SDRAM Base Register - SDBR.....	13-47
13-22	SDRAM Boundary Register 0 - SBR0.....	13-48
13-23	SDRAM Boundary Registers - SBR1.....	13-49
13-24	ECC Control Register - ECCR.....	13-50
13-25	ECC Log Registers - ELOG0, ELOG1.....	13-51
13-26	ECC Address Registers - ECAR0, ECAR1.....	13-52
13-27	ECC Test Register - ECTST.....	13-53
13-28	Flash Base Register 0 - FEBR0.....	13-54
13-29	Flash Base Register 1 - FEBR1.....	13-55
13-30	Flash Bank Size Register 0 - FBSR0.....	13-56
13-31	Flash Bank Size Register 1 - FBSR1.....	13-57
13-32	Flash Wait State Registers - FWSR0, FWSR1.....	13-58
13-33	Memory Controller Interrupt Status Register - MCISR.....	13-59
13-34	Refresh Frequency Register - RFR.....	13-60
14-1	PCI Configuration Command Access Formats.....	14-6
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.....	14-12
14-5	Primary to Secondary Memory Address Decoding Summary.....	14-21
14-6	Primary to Secondary I/O Address Decoding Summary.....	14-21
14-7	Secondary to Primary Memory Address Decoding Summary.....	14-22
14-8	Secondary to Primary I/O Address Decoding Summary.....	14-22
14-9	PCI Commands.....	14-23
14-10	Delayed Transactions vs. Posted Transactions.....	14-26
14-11	Prefetchable and Non-Prefetchable Memory Summary.....	14-33
14-12	Downstream Memory Read Prefetch Size.....	14-33
14-13	Upstream Memory Read Prefetch Size.....	14-34
14-14	Bridge Unit Queue.....	14-42
14-15	D_DRC Assignments.....	14-44
14-16	U_DRC Assignments.....	14-44
14-17	Bridge Transaction Ordering Rules.....	14-46
14-18	Bridge Transaction Ordering and Priority Mechanism.....	14-48
14-19	LOCK# Operation State Definitions.....	14-53
14-20	PSR Error Reporting Summary.....	14-65
14-21	SSR Error Reporting Summary.....	14-67
14-22	64-Bit Configuration Options at Reset.....	14-70
14-23	PCI-to-PCI Bridge Register Table.....	14-72
14-24	Vendor Identification Register - VIDR.....	14-73
14-25	Device Identification Register - DIDR (80960RN).....	14-74
14-26	Device Identification Register - DIDR (80960RM).....	14-74
14-27	Primary Command Register - PCR.....	14-75
14-28	Revision Identification Register - RID.....	14-77

14-29	Class Code Register - CCR .....	14-77
14-30	Cacheline Size Register - CLSR .....	14-78
14-31	Primary Latency Timer Register- PLTR .....	14-79
14-32	Header Type Register- HTR .....	14-80
14-33	Primary Bus Number Register- PBNR .....	14-81
14-34	Secondary Bus Number Register - SBNR .....	14-82
14-35	Subordinate Bus Number Register - SubBNR .....	14-83
14-36	Secondary Latency Timer Register - SLTR .....	14-84
14-37	I/O Base Register - IOBR .....	14-85
14-38	I/O Limit Register - IOLR .....	14-86
14-39	Secondary Status Register - SSR .....	14-87
14-40	Memory Base Register - MBR .....	14-88
14-41	Memory Limit Register - MLR .....	14-89
14-42	Prefetchable Memory Base Register - PMBR .....	14-90
14-43	Prefetchable Memory Limit Register - PMLR .....	14-91
14-44	Bridge Subsystem Vendor ID Register - BSVIR .....	14-92
14-45	Bridge Subsystem ID Register - BSIR .....	14-92
14-46	Bridge Control Register - BCR .....	14-93
14-47	Extended Bridge Control Register - EBCR .....	14-96
14-48	Secondary IDSEL Select Register - SISR .....	14-99
14-49	Primary Bridge Interrupt Status Register - PBISR .....	14-101
14-50	Secondary Bridge Interrupt Status Register - SBISR .....	14-102
14-51	Secondary I/O Base Register - SIOBR .....	14-103
14-52	Secondary I/O Limit Register - SIOLR .....	14-104
14-53	Secondary Memory Base Register - SMBR .....	14-105
14-54	Secondary Memory Limit Register - SMLR .....	14-106
14-55	Secondary Decode Enable Register - SDER .....	14-107
14-56	Queue Control Register- QCR .....	14-109
15-1	ATU Command Support .....	15-4
15-2	Outbound Read Prefetch Sizes .....	15-19
15-3	PCI Multi-Function Device Swapping/Disabling Summary .....	15-22
15-4	Inbound Queues .....	15-28
15-5	Inbound Read Prefetch Data Sizes .....	15-30
15-6	Outbound Queues .....	15-31
15-7	ATU Inbound Data Flow Ordering Rules .....	15-32
15-8	ATU Outbound Data Flow Ordering Rules .....	15-32
15-9	Address Parity Errors on PCI Interface .....	15-35
15-10	Outbound Read Data Parity Errors - Master .....	15-36
15-11	Outbound Write Data Parity Errors - Master .....	15-37
15-12	Inbound Write Data Parity Errors - Slave .....	15-37
15-13	Master Aborts on the PCI Interface .....	15-39
15-14	Target Abort Signaled on the PCI Interface .....	15-39
15-15	Target Abort Detected on the PCI Interface .....	15-40
15-16	SERR# Asserted by PCI Interface .....	15-41
15-17	SERR# Detected by PCI Interface .....	15-41
15-18	Master Abort Detected by Internal Master Interface During Inbound Write .....	15-42
15-19	Master Abort Detected by Internal Master Interface During Inbound Read .....	15-42





15-20	Target Abort Detected by Internal Master Interface During Inbound Write .....	15-43
15-21	Target Abort Detected by Internal Master Interface During Inbound Read.....	15-44
15-22	Primary ATU Error Reporting Summary - PCI Interface.....	15-44
15-23	Secondary ATU Error Reporting Summary - PCI Interface.....	15-45
15-24	Primary ATU Error Reporting Summary - Internal Bus Interface .....	15-46
15-25	Secondary ATU Error Reporting Summary - Internal Bus Interface.....	15-46
15-26	Address Translation Unit Registers.....	15-48
15-27	ATU PCI Configuration Register Space .....	15-50
15-28	ATU Vendor ID Register - ATUVID .....	15-51
15-29	Device ID Register - DID (80960RN) .....	15-52
15-30	Device ID Register - DID (80960RM) .....	15-52
15-31	Primary ATU Command Register - PATUCMD .....	15-53
15-32	Primary ATU Status Register - PATUSR .....	15-54
15-33	ATU Revision ID Register - ATURID.....	15-55
15-34	ATU Class Code Register - ATUCCR .....	15-56
15-35	ATU Cacheline Size Register - ATUCLSR.....	15-56
15-36	ATU Latency Timer Register - ATULT .....	15-57
15-37	ATU Header Type Register - ATUHTR .....	15-57
15-38	ATU BIST Register - ATUBISTR.....	15-58
15-39	Primary Inbound ATU Base Address - PIABAR .....	15-59
15-40	ATU Subsystem Vendor ID Register - ASVIR.....	15-60
15-41	ATU Subsystem ID Register - ASIR.....	15-60
15-42	Expansion ROM Base Address Register -ERBAR.....	15-61
15-43	Memory Block Size Read Response Table.....	15-62
15-44	ATU Base Registers and Associated Limit Registers .....	15-62
15-45	ATU Interrupt Line Register - ATUILR.....	15-63
15-46	ATU Interrupt Pin Register - ATUIPR.....	15-64
15-47	ATU Minimum Grant Register - ATUMGNT .....	15-65
15-48	ATU Maximum Latency Register - ATUMLAT.....	15-66
15-49	Primary Inbound ATU Limit Register - PIALR .....	15-67
15-50	Primary Inbound ATU Translate Value Register - PIATVR.....	15-68
15-51	Secondary Inbound ATU Base Address Register - SIABAR.....	15-69
15-52	Secondary Inbound ATU Limit Register - SIALR .....	15-70
15-53	Secondary Inbound Translate ATU Value Register - SIATVR .....	15-71
15-54	Primary Outbound Memory Window Value Register - POMWVR .....	15-72
15-55	Primary Outbound I/O Window Value Register - POIOWVR .....	15-73
15-56	Primary Outbound DAC Window Value Register - PODWVR .....	15-74
15-57	Primary Outbound Upper 64-bit DAC Register - POU DR .....	15-75
15-58	Secondary Outbound Memory Window Value Register - SOMWVR .....	15-76
15-59	Secondary Outbound I/O Window Value Register - SOIOWVR.....	15-77
15-60	Expansion ROM Limit Register - ERLR .....	15-78
15-61	Expansion ROM Translate Value Register - ERTVR .....	15-79
15-62	ATU Configuration Register - ATUCR.....	15-80
15-63	Primary ATU Interrupt Status Register - PATUISR .....	15-82
15-64	Secondary ATU Interrupt Status Register - SATUISR .....	15-84
15-65	Secondary ATU Command Register - SATUCMD .....	15-86
15-66	Secondary ATU Status Register - SATUSR.....	15-87
15-67	Secondary Outbound DAC Window Value Register - SODWVR .....	15-88

15-68	Secondary Outbound Upper 64-bit DAC Register - SOUDR .....	15-89
15-69	Primary Outbound Configuration Cycle Address Register - POCCAR .....	15-90
15-70	Secondary Outbound Configuration Cycle Address Register - SOCCAR .....	15-91
15-71	Primary Outbound Configuration Cycle Data Register - POCCDR .....	15-92
15-72	Secondary Outbound Configuration Cycle Data Register - SOCCDR .....	15-93
15-73	Primary ATU Interrupt Mask Register - PATUIMR .....	15-94
15-74	Secondary ATU Interrupt Mask Register - SATUIMR .....	15-95
16-1	MU Summary .....	16-4
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 .....	16-13
16-6	Message Unit Register Table .....	16-16
16-7	Inbound Message Register - IMRx .....	16-17
16-8	Outbound Message Register - OMRx .....	16-17
16-9	Inbound Doorbell Register - IDR .....	16-18
16-10	Inbound Interrupt Status Register - IISR .....	16-19
16-11	Inbound Interrupt Mask Register - IIMR .....	16-20
16-12	Outbound Doorbell Register - ODR .....	16-21
16-13	Outbound Interrupt Status Register - OISR .....	16-22
16-14	Outbound Interrupt Mask Register - OIMR .....	16-23
16-15	MU Configuration Register - MUCR .....	16-24
16-16	Queue Base Address Register - QBAR .....	16-25
16-17	Inbound Free Head Pointer Register - IFHPR .....	16-26
16-18	Inbound Free Tail Pointer Register - IFTPR .....	16-27
16-19	Inbound Post Head Pointer Register - IPHPR .....	16-28
16-20	Inbound Post Tail Pointer Register - IPTPR .....	16-29
16-21	Outbound Free Head Pointer Register - OFHPR .....	16-30
16-22	Outbound Free Tail Pointer Register - OFTPR .....	16-31
16-23	Outbound Post Head Pointer Register - OPHPR .....	16-32
16-24	Outbound Post Tail Pointer Register - OPTPR .....	16-33
16-25	Index Address Register - IAR .....	16-34
17-1	Bus Master / Programmed Priorities .....	17-3
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 .....	17-10
17-6	Secondary Arbiter Register Table .....	17-11
17-7	Secondary Arbitration Control Register - SACR .....	17-12
17-8	2-Bit Priorities .....	17-12
17-9	Internal Arbitration Control Register - IACR .....	17-13
17-10	Master Latency Timer Register - MLTR .....	17-14
17-11	Multi-Transaction Timer Register - MTRR .....	17-14
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.....	19-3
19-2	DMA Interrupt Summary.....	19-18
19-3	DMA Controller Unit Registers.....	19-21
19-4	Channel Control Register - CCR.....	19-22
19-5	Channel Status Register - CSR.....	19-23
19-6	Next Descriptor Address Register - NDAR.....	19-25
19-7	Descriptor Address Register - DAR.....	19-26
19-8	Byte Count Register - BCR.....	19-27
19-9	PCI Address Register - PADR.....	19-28
19-10	PCI Upper Address Register - PUADR.....	19-29
19-11	Local Address Register - LADR.....	19-30
19-12	Descriptor Control Register - DCR.....	19-31
19-13	PCI Commands.....	19-32
20-1	Register Description.....	20-3
20-2	Application Accelerator Unit Interrupt Summary.....	20-18
20-3	AAU Interrupts - Special Case.....	20-19
20-4	Application Accelerator Unit Registers.....	20-20
20-5	Accelerator Control Register - ACR.....	20-21
20-6	Accelerator Status Register - ASR.....	20-22
20-7	Accelerator Descriptor Address Register - ADAR.....	20-23
20-8	Accelerator Next Descriptor Address Register - ANDAR.....	20-24
20-9	80960 Source Address Register - SARx.....	20-25
20-10	80960 Destination Address Register - DAR.....	20-26
20-11	Accelerator Byte Count Register - ABCR.....	20-27
20-12	Accelerator Descriptor Control Register - ADCR.....	20-28
21-1	Occurrence Events.....	21-2
21-2	Duration Events.....	21-3
21-3	Relationship between the Monitored mode and Monitored Interface.....	21-3
21-4	Event Monitor Register Table.....	21-20
21-5	Global Timer Mode Register (GTMR).....	21-21
21-6	Event Select Register (ESR).....	21-22
21-7	Event Monitoring Interrupt Status Register - EMISR.....	21-23
21-8	Global Time Stamp Register - GTSR.....	21-24
21-9	Programmable Event Counter Register - PECRx.....	21-25
22-1	I <sup>2</sup> C Bus Definitions.....	22-1
22-2	Modes of Operation.....	22-4
22-3	START and STOP Bit Definitions.....	22-5
22-4	ICCR Programming Values.....	22-7
22-5	Master Transactions.....	22-13
22-6	Slave Transactions.....	22-16
22-7	General Call Address Second Byte Definitions.....	22-18
22-8	I <sup>2</sup> C Register Summary Table.....	22-23
22-9	I <sup>2</sup> C Control Register - ICR.....	22-24
22-10	I <sup>2</sup> C Status Register - ISR.....	22-27
22-11	I <sup>2</sup> C Slave Address Register - ISAR.....	22-29
22-12	I <sup>2</sup> C Data Buffer Register - IDBR.....	22-30
22-13	I <sup>2</sup> C Clock Count Register - ICCR.....	22-31

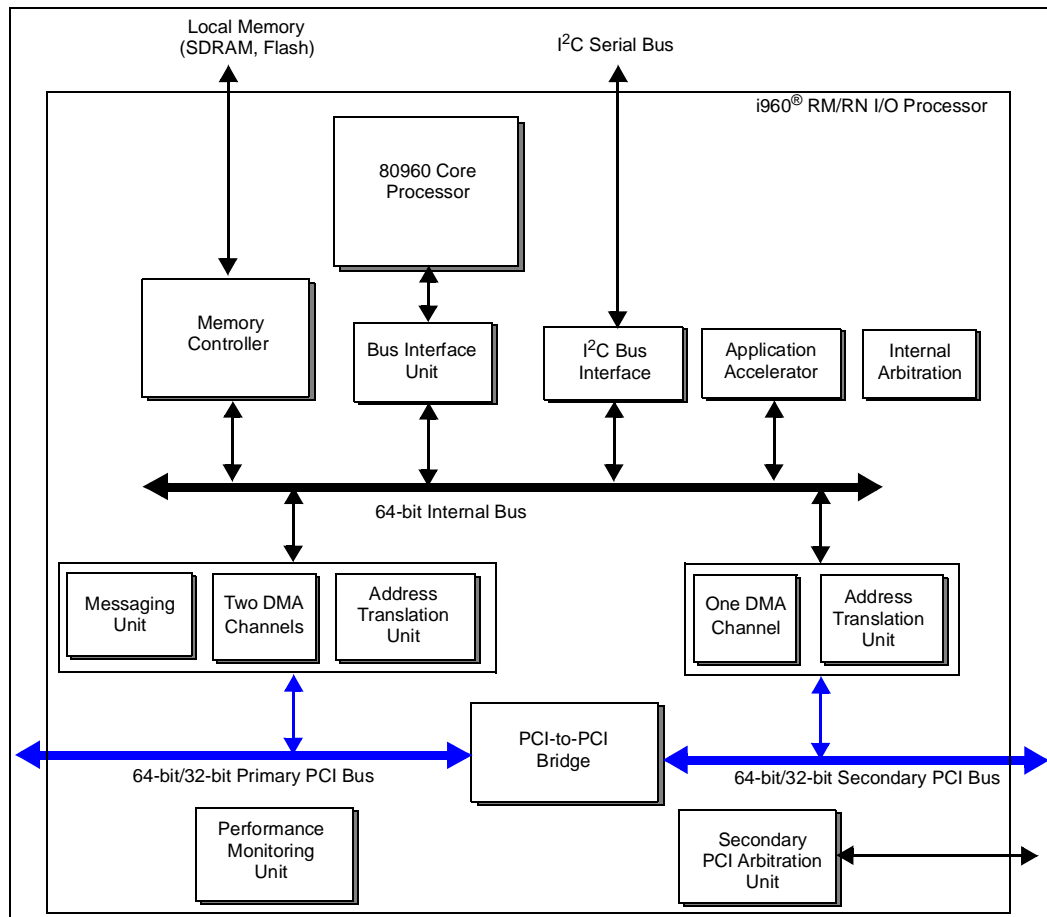
22-14	I <sup>2</sup> C Bus Monitor Register - IBMR .....	22-32
23-1	TAP Controller Pin Definitions.....	23-3
23-2	Boundary-Scan Instruction Set .....	23-4
23-3	IEEE Instructions.....	23-5
23-4	i960 <sup>®</sup> RM/RN I/O Processor Boundary Scan Register Bit Order .....	23-7
24-1	Clock Pin Summary.....	24-3
24-2	Clock Region Summary .....	24-3
24-3	Configuration Modes .....	24-7
A-1	Instruction Field Descriptions .....	A-2
A-2	Encoding of <i>src1</i> and <i>src2</i> in REG Format.....	A-3
A-3	Encoding of <i>src/dst</i> in REG Format.....	A-3
A-4	Encoding of <i>src1</i> in COBR Format.....	A-4
A-5	Encoding of <i>src2</i> in COBR Format.....	A-4
A-6	Addressing Modes for MEM Format Instructions .....	A-5
A-7	Encoding of Scale Field .....	A-6
B-1	Miscellaneous Instruction Encoding Bits.....	B-1
B-2	REG Format Instruction Encodings.....	B-2
B-3	COBR Format Instruction Encodings .....	B-6
B-4	CTRL Format Instruction Encodings .....	B-7
B-5	Cycle Counts for <i>sysctl</i> Operations .....	B-8
B-6	Cycle Counts for <i>icctl</i> Operations .....	B-8
B-7	Cycle Counts for <i>dcctl</i> Operations.....	B-8
B-8	Cycle Counts for <i>intctl</i> Operations.....	B-8
B-9	MEM Format Instruction Encodings .....	B-9
B-10	Addressing Mode Performance.....	B-10
C-1	Access Types .....	C-1
C-2	Supervisor Space Register Addresses .....	C-2
C-3	Timer Registers.....	C-4
C-5	80960 Internal Addresses Assigned to Integrated Peripherals .....	C-7
C-6	Peripheral Memory-Mapped Register Locations .....	C-8

## 1.1 Intel's i960<sup>®</sup> RM/RN I/O Processor

The i960 RM/RN I/O processor integrates a high-performance 80960 “core” 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<sup>2</sup>C Bus Interface Unit, Application Accelerator Unit, Performance Monitoring Unit and Bus Interface Unit.

The PCI Bus is an industry standard, high performance, low latency system bus. The PCI-to-PCI bridge provides a connection path between two independent PCI buses and provides the ability to overcome PCI electrical loading limits. The addition of the i960 core processor brings intelligence to the PCI bus bridge.

Figure 1-1. i960<sup>®</sup> RM/RN I/O Processor Functional Block Diagram



## 1.2 i960<sup>®</sup> RM/RN I/O Processor Features

The i960 RM/RN I/O processor (“80960RM/RN”) combines the i960<sup>®</sup> JT processor with powerful new features to create an intelligent I/O processor. This multi-function PCI device is fully compliant with the *PCI Local Bus Specification*, Revision 2.1. 80960RM/RN-specific features include:

- [Intelligent I/O \(I2O\)](#)
- [PCI-to-PCI Bridge](#)
- [Private PCI Device Support](#)
- [DMA Controller Unit](#)
- [Address Translation Unit](#)
- [Messaging Unit](#)
- [Memory Controller](#)
- [I2C Bus Interface Unit](#)
- [Secondary PCI Arbitration Unit](#)
- [Performance Monitoring Unit](#)
- [Application Accelerator Unit](#)
- [Bus Interface Unit](#)
- [Wind River Systems IxWorks\\* RTOS Compatibility](#)

Because the 80960RM/RN’s core processor is based upon the 80960JT, the two i960 family members are object code compatible and can maintain a sustained execution rate of one instruction per clock. The 80960 local bus, a 32-bit multiplexed burst bus, is connected to the high-speed internal bus through a Bus Interface Unit. Physical and logical memory attributes are programmed via memory-mapped control registers (MMRs). See [Section 1.3, “i960<sup>®</sup> Core Processor Features \(80960JT\)”](#) on page 1-6 for more information.

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

### 1.2.1 Intelligent I/O (I<sub>2</sub>O)

Addressing the software side of I/O, the i960 RM/RN I/O processor supports the industry-standard Intelligent I/O (I<sub>2</sub>O) interface for PCI applications. This specification was formed by Intel and industry leaders in hardware and software to create a standard interface that increases I/O performance and decreases developer time-to-market. This specification provides a common I/O device driver that is independent to both the specific controlled device and the host operating system. The I<sub>2</sub>O architecture facilitates intelligent I/O subsystems by supporting message passing between multiple independent processors. I<sub>2</sub>O provides a standard interface to which all peripheral and network adapter card software can be developed, and remain compliant with popular network operating systems. The I<sub>2</sub>O architecture improves performance by relieving the host of interrupt-intensive I/O tasks. By providing a standard interface, new technologies can be implemented quickly and uniformly.

### 1.2.2 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 (e.g., independent clocks). Dedicated data queues support high-performance bandwidth on the PCI buses. The 80960RM/RN supports PCI 64-bit Dual Address Cycle (DAC) addressing. The bridge has dedicated PCI configuration space that is accessible through the primary PCI bus. See [Chapter 14, “PCI-to-PCI Bridge”](#).

### 1.2.3 Private PCI Device Support

A key 80960RM/RN 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 80960 local memory. Three separate DMA channels accommodate data transfers: two for the 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. See [Chapter 19, “DMA Controller Unit”](#).

### 1.2.5 Address Translation Unit

The Address Translation Unit (ATU) allows PCI transactions direct access to the 80960RM/RN local memory. The ATU supports transactions between PCI address space and 80960RM/RN 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. See [Chapter 15, “Address Translation Unit”](#).

### 1.2.6 Messaging Unit

The Messaging Unit (MU) provides data transfer between the PCI system and the 80960RM/RN. It uses interrupts to notify each system when new data arrives. The MU has four messaging mechanisms:

- Message Registers
- Doorbell Registers
- Circular Queues
- Index Registers

Each allows a host processor or external PCI device and the 80960RM/RN to communicate through message passing and interrupt generation. See [Chapter 16, “Messaging Unit”](#).

### 1.2.7 Memory Controller

The Memory Controller allows direct control of external memory systems, including SDRAM, 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, ECC single-bit error correction and double-bit error detection. External memory can be configured as PCI addressable memory or private 80960RM/RN memory. See [Chapter 13, “Memory Controller”](#).

## 1.2.8 I<sup>2</sup>C Bus Interface Unit

The I<sup>2</sup>C (Inter-Integrated Circuit) Bus Interface Unit allows the i960 core processor to serve as a master and slave device residing on the I<sup>2</sup>C bus. The I<sup>2</sup>C unit uses a serial bus developed by Philips Semiconductor consisting of a two-pin interface. The bus allows the 80960RM/RN to interface to other I<sup>2</sup>C 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. See [Chapter 22, “I<sup>2</sup>C Bus Interface Unit”](#). Also refer to the document *I<sup>2</sup>C Peripherals for Microcontrollers* (Philips Semiconductor).

## 1.2.9 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. See [Chapter 17, “i960® RM/RN I/O Processor Arbitration”](#).

## 1.2.10 Performance Monitoring Unit

The Performance Monitoring Unit (PMON) is used to gather performance measurements that can be used to refine code for improved system level performance. The PMON consists of:

- One dedicated Global Time Stamp Counter
- Fourteen programmable Event Counters

The PMON contains eight different modes which can be used to measure occurrence and duration events on the primary PCI bus, the secondary PCI bus, and the internal bus.

## 1.2.11 Application Accelerator

The Application Accelerator Unit (AAU) provides low-latency, high-throughput data transfer between the AAU and the 80960 local memory. It executes data transfers to and from 80960 local memory and has a programmable interface.

The AAU can perform the following functions:

- Transfers data (reads) from 80960 local memory through the MCU
- Performs an optional XOR operation
- Transfers data (writes) to 80960 local memory through the MCU

The AAU can perform an XOR of up to eight 128-byte pieces of data and write the result to a single destination. It can also be used without the XOR as a local memory to local memory DMA.



## 1.2.12 Bus Interface Unit

The Bus Interface Unit (BIU) interfaces the 32-bit, 100 MHz 80960 local bus to the 64-bit, 66 MHz internal bus which contains external memory and peripherals. The BIU forwards core processor bus accesses to the internal bus, and is responsible for their completion. No address translation is performed.

The BIU has several address/data buffers:

- Write Buffer - stores 1 address & 16 bytes of data
- Read Buffer - stores 16 bytes of data
- Prefetch Buffer - stores 16 bytes of instructions

The BIU has two optional features that can increase overall performance. The BIU can extend core processor instruction fetches by 16 bytes and store the additional bytes in a prefetch buffer. Under special conditions, the BIU can merge two sequential write accesses into one 64-bit internal bus access.

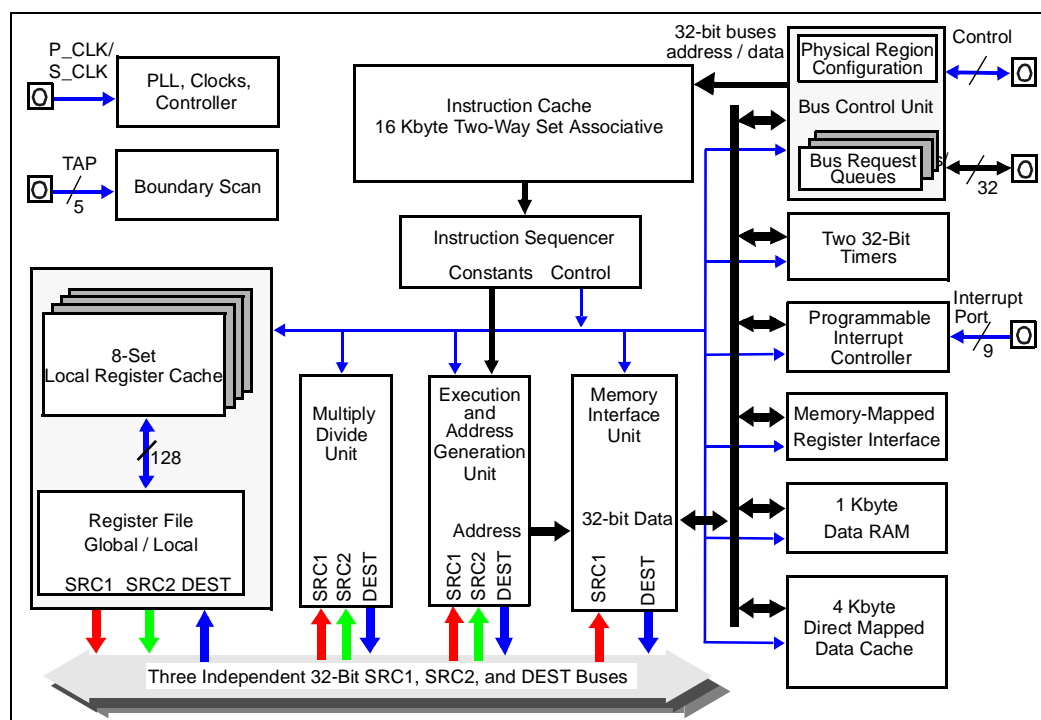
## 1.2.13 Wind River Systems IxWorks\* RTOS

A key feature of the i960 RM/RN I/O processor is Wind River System's IxWorks\* Real-Time Operating System (RTOS). With clearly defined Application Program Interfaces (APIs), IxWorks creates a user-friendly environment to write basic device drivers. IxWorks supports NOS-to-driver independence, and allows multiple I/O software to co-exist reliably. In addition, developers get a 30-day evaluation copy of the Tornado\* development environment. For more information, contact your local Intel representative.

## 1.3 i960<sup>®</sup> Core Processor Features (80960JT)

The processing power of the 80960RM/RN comes from the 80960JT processor core. The 80960JT is a new, scalar implementation of the i960 core architecture. Figure 1-2 shows a block diagram of the 80960JT core processor.

Figure 1-2. 80960JT Core Processor Block Diagram



Factors that contribute to the 80960JT's performance include:

- Single-clock execution of most instructions
- Independent Multiply/Divide Unit
- Efficient instruction pipeline minimizes pipeline break latency
- Register and resource scoreboarding allow overlapped instruction execution
- 128-bit register bus speeds local register caching
- 16 Kbyte two-way set-associative, integrated instruction cache
- 4 Kbyte direct-mapped, integrated data cache
- 1 Kbyte integrated data RAM delivers zero wait state program data

The i960 core processor operates out of its own memory space located on the internal bus, which is independent of the PCI address space. The 80960 local memory can be:

- Made visible to the PCI address space
- Kept private to the i960 core processor
- Allocated as a combination of the two

### 1.3.1 80960 Local Bus

The 100 MHz 80960 local bus exists between the 80960 core processor and the Bus Interface Unit (BIU). The local bus features a 32-bit high performance bus controller which fetches instructions and transfers data at a rate of up to four 32-bit words per six clock cycles. The 80960 local bus controller's features include:

- Unaligned bus accesses performed transparently
- Three-deep load/store queue decouples the bus from the i960 core processor
- Data caching programmable by region

### 1.3.2 Timer Unit

As described in [Chapter 18, “Timers”](#), The Timer Unit (TU) contains two independent 32-bit timers that are capable of counting at software-defined clock rates and generating interrupts. Each is programmed by use of the Timer Unit memory-mapped registers. The timers have a single-shot mode and auto-reload capabilities for continuous operation. Each timer has an independent interrupt request to the 80960RM/RN's interrupt controller.

### 1.3.3 Priority Interrupt Controller

[Chapter 8, “PCI and Peripheral Interrupt Controller Unit”](#) explains how low interrupt latency is critical to many embedded applications. As part of its highly flexible interrupt mechanism, the 80960RM/RN exploits several techniques to minimize latency:

- Interrupt vectors and interrupt handler routines can be reserved on-chip
- Register frames for high-priority interrupt handlers can be cached on-chip
- The interrupt stack can be placed in cacheable memory space

### 1.3.4 Faults and Debugging

The 80960RM/RN employs a comprehensive fault model. The processor responds to faults by making implicit calls to fault handling routines. Specific information collected for each fault allows the fault handler to diagnose exceptions and recover appropriately.

The processor also has built-in debug capabilities. Via software, the 80960RM/RN may be configured to detect as many as seven different trace event types. Alternatively, **mark** and **fmark** instructions can generate trace events explicitly in the instruction stream. Hardware breakpoint registers are also available to trap on execution and data addresses. See [Chapter 9, “Faults”](#).

### 1.3.5 On-Chip Cache and Data RAM

As discussed in [Chapter 4, “Cache and On-Chip Data RAM”](#), memory subsystems often impose substantial wait state penalties. The 80960RM/RN integrates considerable storage resources on-chip to decouple CPU execution from the external bus. The 80960RM/RN includes a 16 Kbyte instruction cache, a 4 Kbyte data cache and 1 Kbyte data RAM.

### 1.3.6 Local Register Cache

The 80960RM/RN rapidly allocates and deallocates local register sets during context switches. The processor needs to flush a register set to the stack only when it saves more than seven sets to its local register cache.

### 1.3.7 Test Features

The 80960RM/RN incorporates features that enhance the user’s ability to test both the processor and the system to which it is attached. These features include ONCE (On-Circuit Emulation) mode and IEEE Std. 1149.1 Boundary Scan (JTAG). See [Chapter 23, “Test Features”](#).

One of the boundary scan instructions, **HIGHZ**, forces the processor to float all its output pins (ONCE mode). ONCE mode can also be initiated at reset without using the boundary scan mechanism.

ONCE mode is useful for board-level testing. This feature allows a mounted 80960RM/RN to electrically “remove” itself from a circuit board. This mode allows system-level testing where a remote tester can exercise the processor system. The test logic does not interfere with component or system behavior and ensures that components function correctly, and also the connections between various components are correct.

The JTAG Boundary Scan feature is an alternative to conventional “bed-of-nails” testing. It can examine connections that might otherwise be inaccessible to a test system.

For reliability, the 80960RM/RN conducts an internal self test upon reset. Before executing its first instruction, it performs a local bus confidence test by performing a checksum on the first words of the Initialization Boot Record.

### 1.3.8 Memory-Mapped Control Registers

The 80960RM/RN is compliant with 80960 family architecture and has the added advantage of memory-mapped, internal control registers not found on the 80960Kx, Sx or Cx processors. This feature provides software an interface to easily read and modify internal control registers.

Each memory-mapped, 32-bit register is accessed via regular memory-format instructions. The processor ensures that these accesses do not generate external bus cycles. See [Chapter 13](#), “Memory Controller”.

### 1.3.9 Instructions, Data Types and Memory Addressing Modes

As with all 80960 family processors, the 80960RM/RN instruction set supports several different data types and formats:

- Bit
- Bit fields
- Integer (8-, 16-, 32-, 64-bit)
- Ordinal (8-, 16-, 32-, 64-bit unsigned integers)
- Triple word (96 bits)
- Quad word (128 bits)

Several chapters describe the i960 RM/RN I/O processor instruction set, including:

- [Chapter 3](#), “Programming Environment”
- [Chapter 5](#), “Instruction Set Overview”
- [Chapter 6](#), “Instruction Set Reference”

## 1.4 About This Document

The i960 RM/RN I/O processor incorporates Peripheral Component Interconnect (PCI) functionality with the i960 JT processor. As such, it is assumed that the reader has a working understanding of the Peripheral Component Interconnect (PCI), *PCI Local Bus Specification*, Revision 2.1, and the i960 core processor.

## 1.4.1 Terminology

In this document, the following terms are used:

- *80960RM/RN* refers generically to the i960 RM/RN I/O processor family. As of this printing, the family includes the 32-bit PCI 80960RM and the 64-bit 80960RN.
- *80960 internal bus* refers to the i960 RM/RN I/O processor's internal local bus, not the PCI local bus.
- *80960 core local bus* refers to the 80960JT bus, between the core processor and the BIU.
- *Primary and Secondary PCI buses* are the i960 RM/RN I/O processor's internal PCI buses that conform to PCI Special Interest Group specifications.
- *i960 core processor* refers to the i960 JT processor that is integrated into the 80960RM/RN.
- *DWORD* is a 32-bit data word.
- *80960 Local memory* is a memory subsystem on the 80960RM/RN internal bus.

The following terms are used primarily in [Chapter 14, "PCI-to-PCI Bridge"](#):

- *Downstream* — at or toward a PCI bus with a higher number (after configuration).
- *Host processor* — Processor located upstream from the i960 RM/RN I/O processor.
- *Local processor* — i960 core processor within the i960 RM/RN I/O processor.
- *Upstream* — At or toward a PCI bus with a lower number (after configuration).

## 1.4.2 Representing Numbers

Assume that all numbers are base 10 unless designated otherwise. In text, numbers in base 16 are represented as "nnnH", where the "H" signifies hexadecimal. In pseudocode descriptions, hexadecimal numbers are represented in the form 0x1234ABCD. Binary numbers are not explicitly identified and are assumed when bit operations or bit ranges are used.

## 1.4.3 Fields

A *preserved* field in a data structure is one that the processor does not use. Preserved fields can be used by software; the processor does not modify such fields.

A *reserved* field is a field that may be used by an implementation. When 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 only* field can be read to return the current value. Writes to *read only* fields are treated as no-op operations and do not change the current value or 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 causes no change to the field. A write to a *read/clear* field with a data value of 1 causes the field to be cleared (reset to the value of 0). For example, when a *read/clear* field has a value of F0H, and a data value of 55H is written, the resultant field is 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 causes no change to the field. A write to a *read/set* field with a data value of 1 causes the field to be set (set to the value of 1). For example, when a *read/set* field has a value of F0H, and a data value of 55H is written, the resultant field is F5H.

#### 1.4.4 Specifying Bit and Signal Values

The terms *set* and *clear* in this specification refer to bit values in register and data structures. When a bit is set, its value is 1; when 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.

#### 1.4.5 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 80960 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 level.

#### 1.4.6 Solutions960® Program

Intel’s Solutions960® program features a wide variety of development tools that support the i960 processor family. Many of these tools are developed by partner companies; some are developed by Intel, such as profile-driven optimizing compilers. For more information on these products, contact your local Intel representative.

## 1.4.7 Related Documents

Intel documentation is available from your Intel Sales Representative or Intel Literature Sales.

Intel Corporation  
Literature Sales  
P.O. Box 5937  
Denver, CO 80217-9808  
1-800-548-4725

**Table 1-1. Additional Information Sources**

Document Title	Order / Contact
<i>i960<sup>®</sup> RM/RN I/O Processor Specification Update</i>	Intel Order # 273164
<i>i960<sup>®</sup> RM I/O Processor Data Sheet</i>	Intel Order # 273156
<i>i960<sup>®</sup> RN I/O Processor Data Sheet</i>	Intel Order # 273157
<i>i960<sup>®</sup> Jx Microprocessor Developer's Manual</i>	Intel Order # 272483
<i>IQ80960RM/RN Evaluation Board Manual</i>	Intel Order # 273160
<i>i960<sup>®</sup> RM/RN I/O Processor Design Guide</i>	Intel Order # 273139
<i>PCI Local Bus Specification, Revision 2.1</i>	PCI Special Interest Group 1-800-433-5177
<i>PCI-to-PCI Bridge Architecture Specification, Revision 1.0</i>	PCI Special Interest Group 1-800-433-5177
<i>PCI System Design Guide, Revision 1.0</i>	PCI Special Interest Group 1-800-433-5177
<i>I<sup>2</sup>C Peripherals for Microcontrollers</i>	Philips Semiconductor
<i>I<sup>2</sup>C Bus and How to Use It (Including Specifications)</i>	Philips Semiconductor
<i>I<sup>2</sup>C Peripherals for Microcontrollers (Including Fast Mode)</i>	Signetics
<i>New DRAM Technologies by Steven Przybylski</i>	Book Store
<i>IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture</i>	Institute of Electrical and Electronics Engineers Inc. 345 E. 47th St. New York, NY 10017

## 1.4.8 Electronic Information

Intel's documentation and other information is available from Intel's website ([Table 1-2](#)).

**Table 1-2. Electronic Information**

Information Source	URL
Intel's World-Wide Web Home Page	<a href="http://www.intel.com/">http://www.intel.com/</a>
Wind River System's IxWorks	<a href="http://www.wrs.com/">http://www.wrs.com/</a>
I <sub>2</sub> O Special Interest Group Web Site	<a href="http://www.i2osig.org/">http://www.i2osig.org/</a>



# Data Types and Memory Addressing Modes

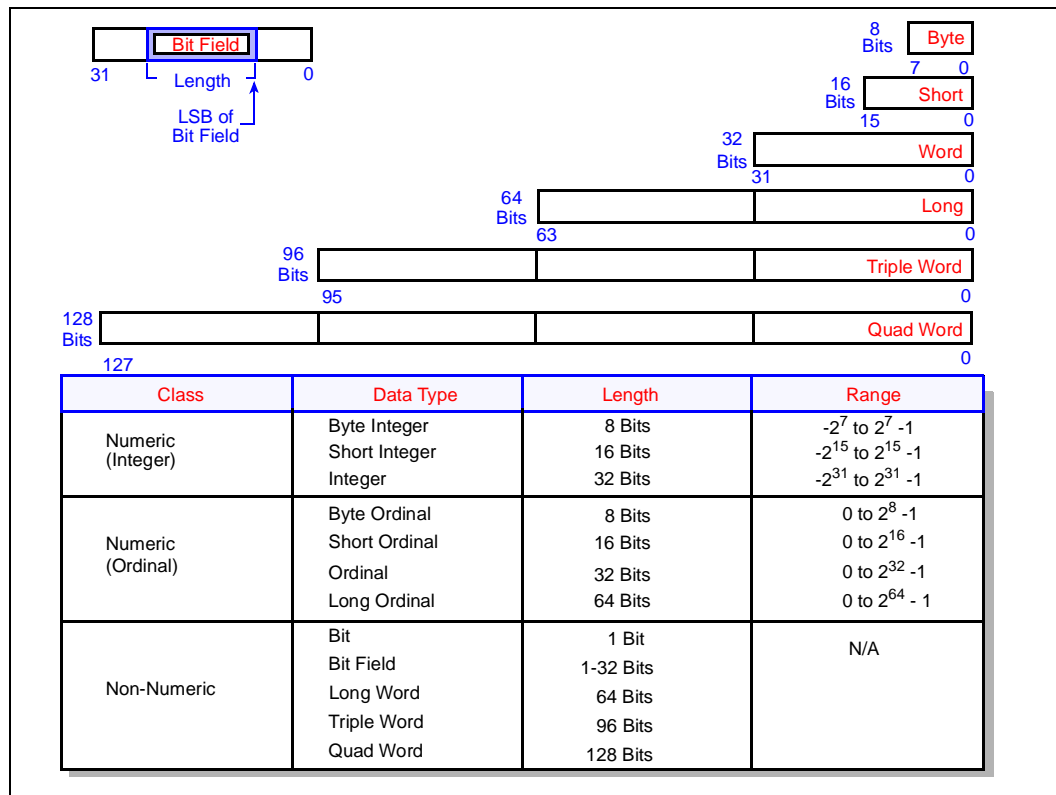
## 2.1 Data Types

The instruction set references or produces several data lengths and formats. The i960<sup>®</sup> RM/RN I/O processor supports the following data types:

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

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

Figure 2-1. Data Types and Ranges



## 2.1.1 Word/Dword Notation

Data lengths, as described in the *PCI Local Bus Specification* Revision 2.1, differ from the conventions used for the 80960 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 i960 RM/RN I/O processor, the term *word* refers to a 32-bit block of data.

**Table 2-1. 80960 and PCI Architecture Data Word Notation Differences**

No. of Bits	PCI Architecture	80960 Architecture
16	<b>word</b>	short word or half word
32	doubleword or dword	<b>word</b>

## 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 i960 RM/RN 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	<i>offset</i>	offset (smaller than 4096)	exp	MEMA
	<i>displacement</i>	displacement (larger than 4095)	exp	MEMB
Register Indirect	abase	(reg)		MEMB
	<i>with offset</i>	abase + offset	exp (reg)	MEMA
	<i>with displacement</i>	abase + displacement	exp (reg)	MEMB
	<i>with index</i>	abase + (index*scale)	(reg) [reg*scale]	MEMB
	<i>with index and displacement</i>	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-9 “MEM Format Instruction Encodings”](#) on [page B-9](#) 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

```

This chapter describes the i960® RM/RN I/O processor's programming environment including global and local registers, control registers, literals, processor-state registers and address space.

## 3.1 Overview

The 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:

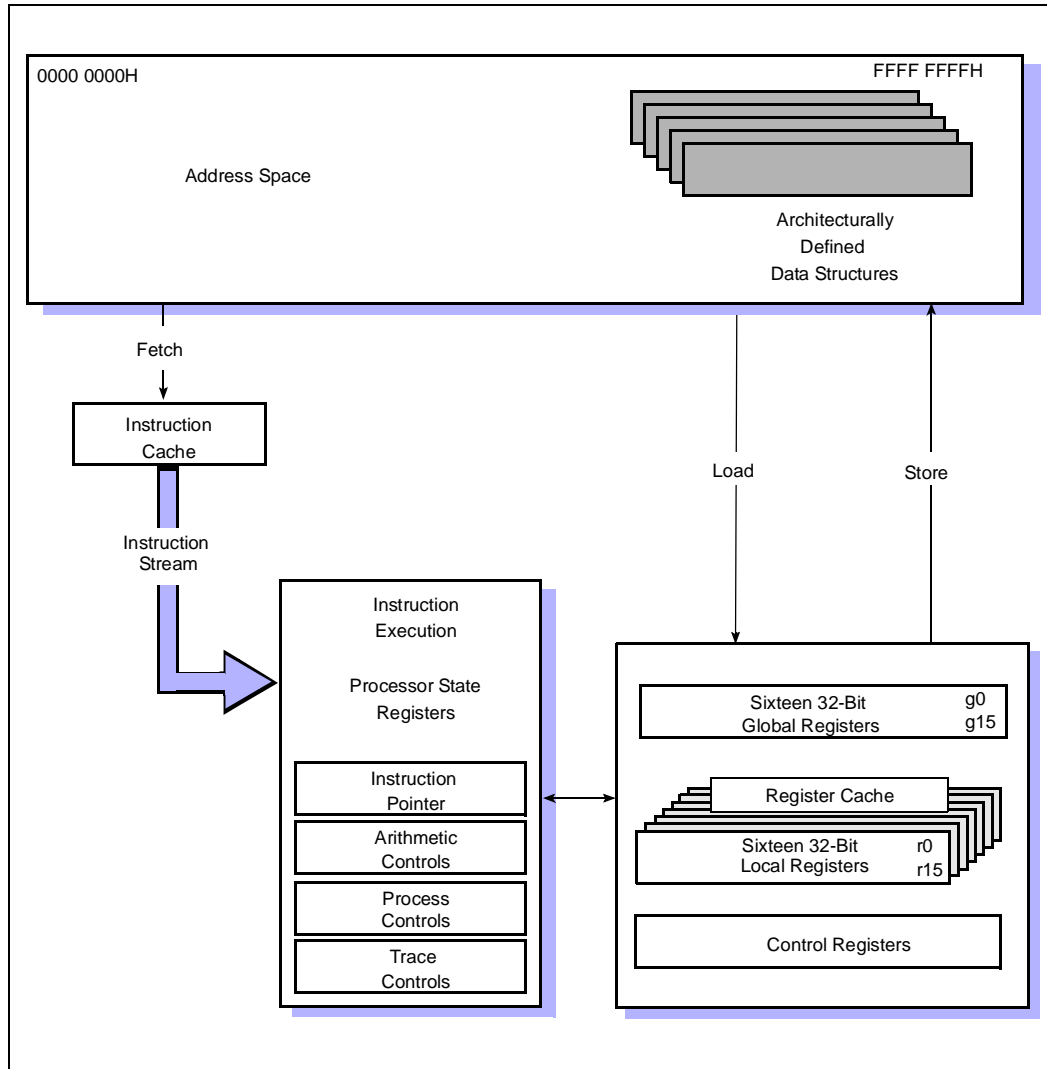
- interrupt stack
- local stack
- supervisor stack
- control table
- fault table
- interrupt table
- system procedure table
- process control block
- initialization boot record

## 3.2 Registers and Literals as Instruction Operands

With the exception of a few special instructions, the i960 RM/RN 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. i960<sup>®</sup> RM/RN 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 i960 core processor device identification and stepping information. g0 retains this information until it is written over by the user program. The i960 core processor device identification and stepping information is also stored in the memory-mapped DEVICEID register located at FF00 8710H. In addition, the i960 RM/RN 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

```

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

```

### 3.2.4 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.

### 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**

Instruction Encoding	Operand Field	Operand <sup>1</sup>		
		Local Register	Global Register	Literal
<b>REG</b>	<i>src1</i>	X	X	X
	<i>src2</i>	X	X	X
	<i>src/dst</i> (as <i>src</i> )	X	X	X
	<i>src/dst</i> (as <i>dst</i> )	X	X	
	<i>src/dst</i> (as both)	X	X	
<b>MEM</b>	<i>src/dst</i>	X	X	
	<i>abase</i>	X	X	
	<i>index</i>	X	X	
<b>COBR</b>	<i>src1</i>	X	X	
	<i>src2</i>	X	X	X
	<i>dst</i>	X <sup>2</sup>	X <sup>2</sup>	X <sup>2</sup>

**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 i960 RM/RN 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 i960 RM/RN I/O processor. The first set exists in the FF00 0000H through FFFF FFFFH address range and is used to control the i960 core processor functions. The second set exists in the 0000 1000H through 0000 18FFH address range and is used to control the i960 RM/RN I/O processor integrated peripherals. The processor ensures that accesses to MMRs do not generate external bus cycles.

### 3.3.1 i960<sup>®</sup> Core Processor Function Memory-Mapped Registers

Portions of the i960 RM/RN 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-2](#) and [Table C-3](#) 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 i960<sup>®</sup> 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-2](#) and [Table C-3](#) 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 i960 RM/RN 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 i960 RM/RN 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 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 i960® RM/RN 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 i960 core processor. See [Appendix C, “Memory-Mapped Registers”](#).

The memory-mapped registers discussed in this chapter are specific to the i960 RM/RN 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<sup>2</sup>C 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 i960 core processor.

### 3.3.2.1 Accessing The Peripheral Memory-Mapped Registers

The PMMR interface is a slave device connected to the 80960 internal bus. This interface accepts data transactions that appear on the 80960 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 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 i960 core 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 80960 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 80960 internal bus atomic operations from the i960 core processor. The i960 core processor provides **atmod** (atomic modify) and **atadd** (atomic add) instructions for atomic accesses to memory. When the 80960 processor executes an **atmod** or **atadd** instruction, the LOCK# signal is asserted. The 80960 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 i960 core processor operating in either user mode or supervisor mode. In addition, the PMMR does not provide any access fault to the i960 core processor.

The following PMMR registers have read/write access from the 80960 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-2](#) and [Table C-3](#) 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 i960 RM/RN 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.

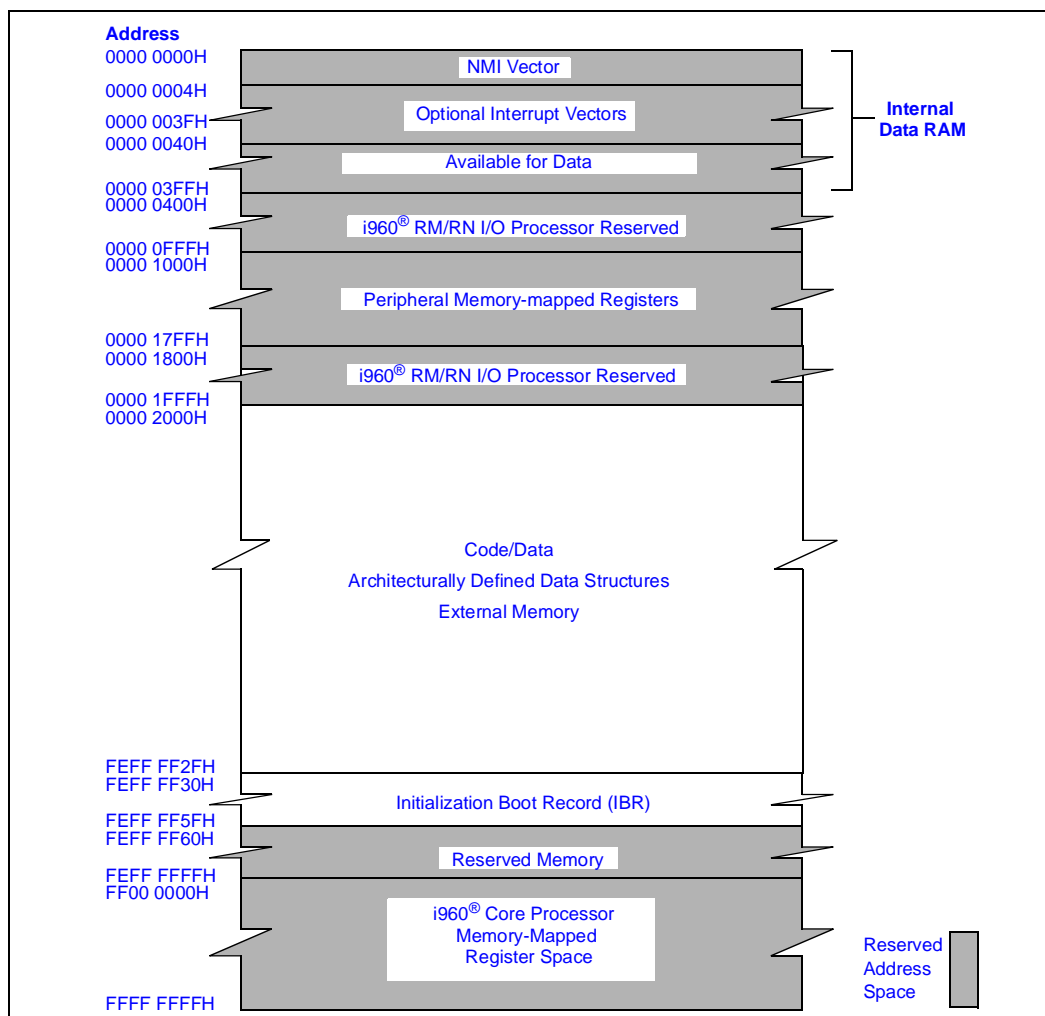
**Table 3-3. Data Structure Descriptions**

Structure	Description
<b>User and Supervisor Stacks</b> <a href="#">Section 7.6, "User and Supervisor Stacks" on page 7-17</a>	The processor uses these stacks when executing application code.
<b>Interrupt Stack</b> <a href="#">Section 8.1.5, "Interrupt Stack And Interrupt Record" on page 8-6</a>	A separate interrupt stack is provided to ensure that interrupt handling does not interfere with application programs.
<b>System Procedure Table</b> <a href="#">Section 3.7, "User-Supervisor Protection Model" on page 3-18</a> <a href="#">Section 7.5, "System Calls" on page 7-14</a>	Contains pointers to system procedures. Application code uses the system call instruction ( <b>calls</b> ) 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.
<b>Interrupt Table</b> <a href="#">Section 8.1.4, "Interrupt Table" on page 8-4</a>	The interrupt table contains vectors (pointers) to interrupt handling procedures. When an interrupt is serviced, a particular interrupt table entry is specified.
<b>Fault Table</b> <a href="#">Section 9.3, "Fault Table" on page 9-4</a>	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.
<b>Control Table</b> <a href="#">Section 11.4.4, "Control Table" on page 11-18</a>	Contains on-chip control register values. Control table values are moved to on-chip registers at initialization or with <b>sysctl</b> .

## 3.5 Memory Address Space

The i960 RM/RN 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 LOCK# signal to qualify hold requests from external bus agents. The processor asserts LOCK# 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. i960 RM/RN 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 i960 RM/RN I/O processor's reserved memory.

The i960 RM/RN 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 i960 RM/RN 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 11-14.

**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 FFFF 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 i960 RM/RN 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 i960 RM/RN 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 i960 RM/RN 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
- Process Controls (PC) register
- Arithmetic Controls (AC) 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.

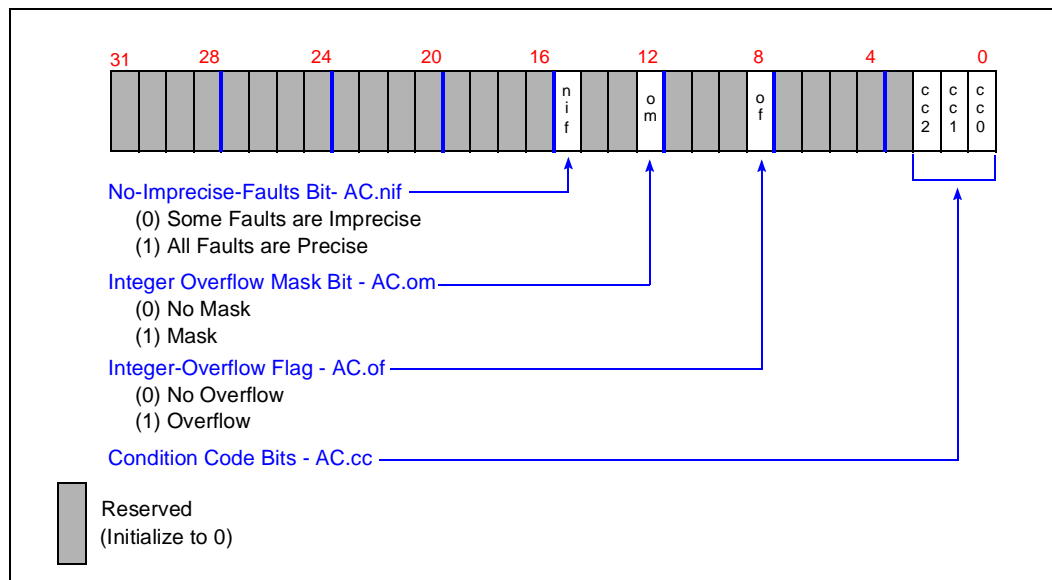
The IP register cannot be read directly. However, the IP-with-displacement addressing mode lets software use the IP as an offset into the address space. This addressing mode can also be used with the **lda** (load address) instruction to read the current IP value.

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 3-5) contains condition code flags, integer overflow flag, mask bit and a bit that controls faulting on imprecise faults. Unused AC register bits are reserved.

**Table 3-5. 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’s *condition code flags* (bits 0-2) to indicate the 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 the extended arithmetic instructions. To show true or false conditions, the processor sets the flags as shown in [Table 3-6](#). To show equality and inequalities, the processor sets the condition code flags as shown in [Table 3-7](#).

**Table 3-6. Condition Codes for True or False Conditions**

Condition Code	Condition
010 <sub>2</sub>	true
000 <sub>2</sub>	false

**Table 3-7. Condition Codes for Equality and Inequality Conditions**

Condition Code	Condition
000 <sub>2</sub>	unordered
001 <sub>2</sub>	greater than
010 <sub>2</sub>	equal
100 <sub>2</sub>	less than

The term *unordered* is used when comparing floating point numbers. The i960 RM/RN 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 3-8](#).

**Table 3-8. Condition Codes for Carry Out and Overflow**

Condition Code	Condition
01X <sub>2</sub>	carry out
0X1 <sub>2</sub>	overflow

Certain instructions, such as the branch-if instructions, use a 3-bit mask to evaluate the condition code flags. For example, the branch-if-greater-or-equal instruction (**bge**) uses a mask of 011<sub>2</sub> to determine if the condition code is set to either greater-than or equal. Conditional instructions use similar masks for the remaining conditions such as: greater-or-equal (011<sub>2</sub>), less-or-equal (110<sub>2</sub>) and not-equal (101<sub>2</sub>). The mask is part of the instruction opcode; the instruction performs a bitwise AND of the 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.

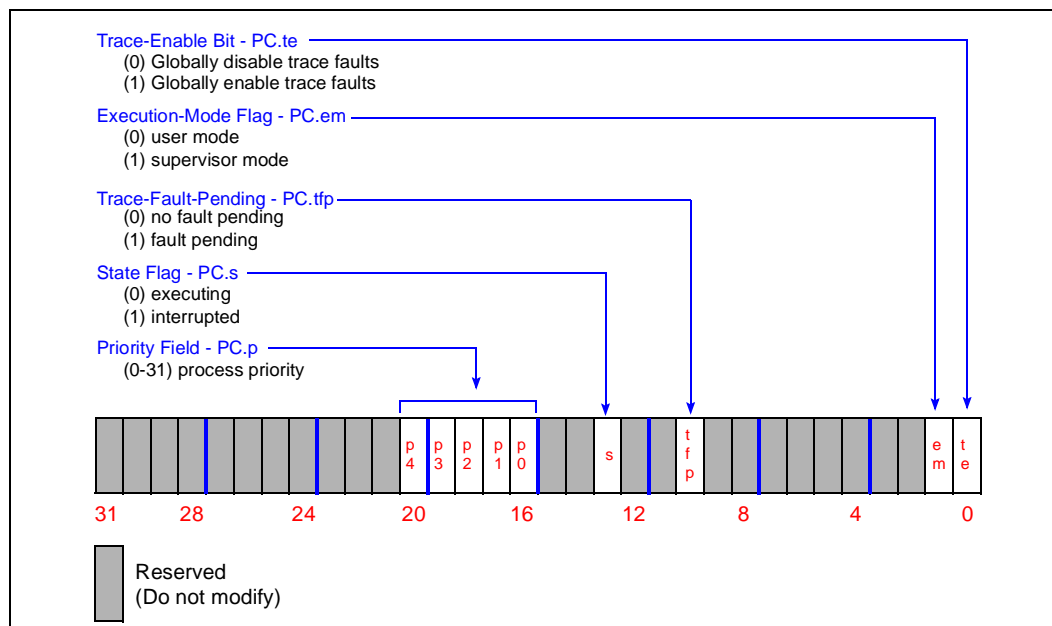
Once the processor sets this flag, the flag remains set until the application software clears it. Refer to the discussion of the ARITHMETIC.INTEGER\_OVERFLOW fault in [Chapter 9, “Faults”](#) for more information about the integer overflow mask bit and flag.

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 9-18](#) for more information.

### 3.6.3 Process Controls Register – PC

The PC register (Table 3-9) 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 3-9. 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 6-66](#) 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.

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



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

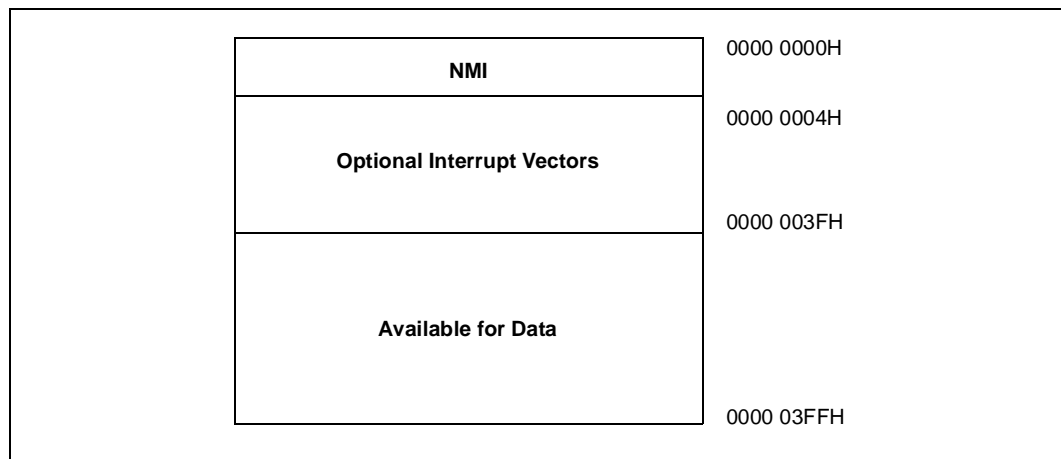
Internal data RAM is mapped to the lower 1 Kbyte (0 to 03FFH) of the address space. Loads and stores with target addresses in internal data RAM operate directly on the internal data RAM; no external bus activity is generated. Data RAM allows time-critical data storage and retrieval without dependence on external bus performance. Only data accesses are allowed to the internal data RAM; instructions cannot be fetched from the internal data RAM. Instruction fetches directed to the 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 latency for these interrupts. The word at location 0000H is always reserved for the cached NMI vector. With the exception of the cached NMI vector, other reserved portions of the data RAM can be used for data storage when the alternate function is not used. All locations of the 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 12-3. Protecting this portion of the data RAM from user and supervisor rights preserves the interrupt vectors that may be cached there. See [Section 8.4.4.1, “Vector Caching Option”](#) on page 8-28.

**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.ird). 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 12-3 for the format of the BCON register.

New versions of 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 i960<sup>®</sup> Rx 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 6-46 explains the instruction itself and [Section 7.2, “Modifying the PFP Register”](#) on page 7-11 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 i960 RM/RN 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 i960 RM/RN 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 11-15](#). 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 i960 RM/RN 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 i960 RM/RN 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 i960 RM/RN 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 i960 RM/RN I/O processor. Alternatively, i960 processor legacy software can use **sysctl**.

## 4.4 Data Cache

The i960 RM/RN 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 12.2.1, “PMCON Registers”](#) on page 12-1 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 i960 RM/RN 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 i960 RM/RN 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 i960 RM/RN 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 i960 RM/RN 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 and Bus Masters and Cache Coherency

The i960 RM/RN 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 12-1](#) 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.

This chapter provides an overview of the i960® microprocessor family’s instruction set and i960 RM/RN 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

i960 RM/RN 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 80960 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.

```
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
```

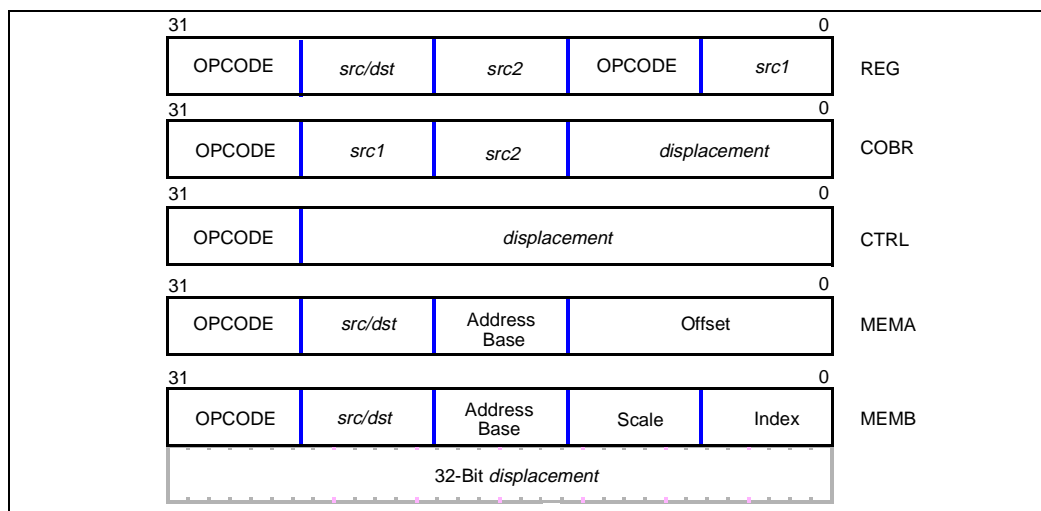
## 5.1.2 Instruction Encoding Formats

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, CTRL, 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.

Format	Operand(s)	Description
REG	<i>src1, src2, src/dst</i>	<i>src1</i> and <i>src2</i> can be global registers, local registers or literals. <i>src/dst</i> is either a global or a local register.
CTRL	<i>displacement</i>	CTRL format is used for branch and call instructions. <i>displacement</i> value indicates the target instruction of the branch or call.
COBR	<i>src1, src2, displacement</i>	<i>src1, src2</i> indicate values to be compared; <i>displacement</i> indicates branch target. <i>src1</i> can specify a global register, local register or a literal. <i>src2</i> can specify a global or local register.
MEM	<i>src/dst, efa</i>	Specifies source or destination register and an effective address ( <i>efa</i> ) formed by using the processor's addressing modes as described in <a href="#">Section 2.3, "Memory Addressing Modes"</a> 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 5-2. 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 5-2. i960<sup>®</sup> RM/RN 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.

<b>ld</b>	load word	<b>st</b>	store word
<b>ldob</b>	load ordinal byte	<b>stob</b>	store ordinal byte
<b>ldos</b>	load ordinal short	<b>stos</b>	store ordinal short
<b>ldib</b>	load integer byte	<b>stib</b>	store integer byte
<b>ldis</b>	load integer short	<b>stis</b>	store integer short
<b>ldl</b>	load long	<b>stl</b>	store long
<b>ldt</b>	load triple	<b>stt</b>	store triple
<b>ldq</b>	load quad	<b>stq</b>	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.

<b>mov</b>	move word
<b>movl</b>	move long word
<b>movt</b>	move triple word
<b>movq</b>	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 i960 RM/RN 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.

<b>selno</b>	Select Based on Unordered
<b>selg</b>	Select Based on Greater
<b>sele</b>	Select Based on Equal
<b>selge</b>	Select Based on Greater or Equal
<b>sell</b>	Select Based on Less
<b>selne</b>	Select Based on Not Equal
<b>selle</b>	Select Based on Less or Equal
<b>selo</b>	Select Based on Ordered

## 5.2.3 Arithmetic

Table 5-3 lists arithmetic operations and data types for which the i960 RM/RN 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 5-3. 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:

<b>addi</b>	Add Integer
<b>addo</b>	Add Ordinal
<b>subi</b>	Subtract Integer
<b>subo</b>	Subtract Ordinal
<b>SUB&lt;cc&gt;</b>	Conditional Subtract
<b>muli</b>	Multiply Integer
<b>mulo</b>	Multiply Ordinal
<b>divi</b>	Divide Integer
<b>divo</b>	Divide Ordinal

**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:

<b>remi</b>	remainder integer
<b>remo</b>	remainder ordinal
<b>modi</b>	modulo integer

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:

<b>shlo</b>	shift left ordinal
<b>shro</b>	shift right ordinal
<b>shli</b>	shift left integer
<b>shri</b>	shift right integer
<b>shr di</b>	shift right dividing integer
<b>rotate</b>	rotate left
<b>eshro</b>	extended shift right ordinal

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.)

**shr di** 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 **shr di** 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):

<b>addc</b>	add ordinal with carry
<b>subc</b>	subtract ordinal with carry
<b>emul</b>	extended multiply
<b>ediv</b>	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:

<b>and</b>	<i>src2</i> AND <i>src1</i>
<b>notand</b>	(NOT <i>src2</i> ) AND <i>src1</i>
<b>andnot</b>	<i>src2</i> AND (NOT <i>src1</i> )
<b>xor</b>	<i>src2</i> XOR <i>src1</i>
<b>or</b>	<i>src2</i> OR <i>src1</i>
<b>nor</b>	NOT ( <i>src2</i> OR <i>src1</i> )
<b>xnor</b>	<i>src2</i> XNOR <i>src1</i>
<b>not</b>	NOT <i>src1</i>
<b>notor</b>	(NOT <i>src2</i> ) or <i>src1</i>
<b>ornot</b>	<i>src2</i> or (NOT <i>src1</i> )
<b>nand</b>	NOT ( <i>src2</i> AND <i>src1</i> )

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:

<b>setbit</b>	set bit
<b>clrbt</b>	clear bit
<b>notbit</b>	invert bit
<b>alterbit</b>	alter bit
<b>scanbit</b>	scan for bit
<b>spanbit</b>	span over bit

**setbit**, **clrbt** 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 5-10](#), 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:

<b>cmpi</b>	Compare Integer
<b>cmpib</b>	Compare Integer Byte
<b>cmpis</b>	Compare Integer Short
<b>cmpo</b>	Compare Ordinal
<b>concmpi</b>	Conditional Compare Integer
<b>concmpo</b>	Conditional Compare Ordinal
<b>chkbit</b>	Check Bit

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:

<b>cmpinci</b>	compare and increment integer
<b>cmpinco</b>	compare and increment ordinal
<b>cmpdeci</b>	compare and decrement integer
<b>cmpdeco</b>	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:

<b>teste</b>	test for equal
<b>testne</b>	test for not equal
<b>testl</b>	test for less
<b>testle</b>	test for less or equal
<b>testg</b>	test for greater
<b>testge</b>	test for greater or equal
<b>testo</b>	test for ordered
<b>testno</b>	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:

<b>b</b>	Branch
<b>bx</b>	Branch Extended
<b>bal</b>	Branch and Link
<b>balx</b>	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 7-19](#), 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:

<b>be</b>	branch if equal/true
<b>bne</b>	branch if not equal
<b>bl</b>	branch if less
<b>ble</b>	branch if less or equal
<b>bg</b>	branch if greater
<b>bge</b>	branch if greater or equal
<b>bo</b>	branch if ordered
<b>bno</b>	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 3-14](#) 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:

<b>cmpibe</b>	compare integer and branch if equal
<b>cmpibne</b>	compare integer and branch if not equal
<b>cmpibl</b>	compare integer and branch if less
<b>cmpible</b>	compare integer and branch if less or equal
<b>cmpibg</b>	compare integer and branch if greater
<b>cmpibge</b>	compare integer and branch if greater or equal
<b>cmpibo</b>	compare integer and branch if ordered
<b>cmpibno</b>	compare integer and branch if unordered
<b>cmpobe</b>	compare ordinal and branch if equal
<b>cmpobne</b>	compare ordinal and branch if not equal
<b>cmpobl</b>	compare ordinal and branch if less
<b>cmpoble</b>	compare ordinal and branch if less or equal
<b>cmpobg</b>	compare ordinal and branch if greater
<b>cmpobge</b>	compare ordinal and branch if greater or equal
<b>bbs</b>	check bit and branch if set
<b>bbc</b>	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 5-10](#). 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 i960 RM/RN 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:

<b>call</b>	call
<b>callx</b>	call extended
<b>calls</b>	call system
<b>ret</b>	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.

<b>faulte</b>	fault if equal
<b>faultne</b>	fault if not equal
<b>faultl</b>	fault if less
<b>faultle</b>	fault if less or equal
<b>faultg</b>	fault if greater
<b>faultge</b>	fault if greater or equal
<b>faulto</b>	fault if ordered
<b>faultno</b>	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:

<b>modpc</b>	modify process controls
<b>modtc</b>	modify trace controls
<b>mark</b>	mark
<b>fmark</b>	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 i960 RM/RN 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:

<b>modpc</b>	Modify the Process Controls register
<b>flushreg</b>	Flush cached local register sets to memory
<b>modac</b>	Modify the Arithmetic Controls register

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 i960 RM/RN 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 i960 RM/RN 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:

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

On the i960 RM/RN 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 i960 RM/RN 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.

**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 i960 RM/RN I/O processor. They require two cycles to execute, and one more cycle to fetch the target instruction if the branch is actually taken. The instructions are:

- **cmpobno**      • **cmpobo**      • **cmpobl**      • **cmpoble**      • **cmpobe**      • **cmpobne**
- **cmpobg**      • **cmpobge**      • **cmpibno**      • **cmpibo**      • **cmpibl**      • **cmpible**
- **cmpibe**      • **cmpibg**      • **cmpibne**      • **cmpibge**      • **bbc**      • **bbs**

### 5.3.1.7 Memory Instructions

The i960 RM/RN 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:

**ldob, ldib, ldos, ldis, ld, lda stob, stib, stos, stis, st**

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:

**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 (**multi**) 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.

<b>icctl</b>	Instruction cache control
<b>dcctl</b>	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.





This chapter provides detailed information about each instruction available to the i960® RM/RN 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 6-1](#).

Information in this chapter is oriented toward programmers who write assembly language code for the i960 RM/RN I/O processor. Information provided for each instruction includes:

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

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 i960 RM/RN I/O processor-specific extensions to the 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 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:

**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:

<b>sub*</b>	<i>src1</i>	<i>src2</i>	<i>dst</i>
	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:

reg	Global (g0 ... g15) or local (r0 ... r15) register
lit	Literal of the range 0 ... 31
disp	Signed displacement of range $(-2^{22} \dots 2^{22} - 1)$
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:

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:

```

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**

=	Assignment
==, !=	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:

```
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>

<b>Mnemonic:</b>	<b>addono</b>	Add Ordinal if Unordered
	<b>addog</b>	Add Ordinal if Greater
	<b>addoe</b>	Add Ordinal if Equal
	<b>addoge</b>	Add Ordinal if Greater or Equal
	<b>addol</b>	Add Ordinal if Less
	<b>addone</b>	Add Ordinal if Not Equal
	<b>addole</b>	Add Ordinal if Less or Equal
	<b>addoo</b>	Add Ordinal if Ordered
	<b>addino</b>	Add Integer if Unordered
	<b>addig</b>	Add Integer if Greater
	<b>addie</b>	Add Integer if Equal
	<b>addige</b>	Add Integer if Greater or Equal
	<b>addil</b>	Add Integer if Less
	<b>addine</b>	Add Integer if Not Equal
	<b>addile</b>	Add Integer if Less or Equal
	<b>addio</b>	Add Integer if Ordered
<b>Format:</b>	<b>add*</b>	<i>src1</i> , <i>src2</i> , <i>dst</i> reg/lit reg/lit reg
<b>Description:</b>	Conditionally adds <i>src2</i> and <i>src1</i> values and stores the result in <i>dst</i> 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 the mask part of the opcode is not 0, then the values are added and placed in the destination. Otherwise the destination is left unchanged. <a href="#">Table 6-4</a> shows the condition code mask for each instruction. The mask is in opcode bits 4-6.	

**Table 6-4. Condition Code Mask Descriptions**

Instruction	Mask	Condition
<b>addono</b>	000 <sub>2</sub>	Unordered
<b>addino</b>		
<b>addog</b>	001 <sub>2</sub>	Greater
<b>addig</b>		
<b>addoe</b>	010 <sub>2</sub>	Equal
<b>addie</b>		
<b>addoge</b>	011 <sub>2</sub>	Greater or equal
<b>addige</b>		
<b>addol</b>	100 <sub>2</sub>	Less
<b>addil</b>		
<b>addone</b>	101 <sub>2</sub>	Not equal
<b>addine</b>		
<b>addole</b>	110 <sub>2</sub>	Less or equal
<b>addile</b>		
<b>addoo</b>	111 <sub>2</sub>	Ordered
<b>addio</b>		



## 6.2.2 **addc**

<b>Mnemonic:</b>	<b>addc</b>	Add Ordinal With Carry
<b>Format:</b>	<b>addc</b>	<i>src1</i> , <i>src2</i> , <i>dst</i> reg/lit reg/lit reg
<b>Description:</b>	<p>Adds <i>src2</i> and <i>src1</i> values and condition code bit 1 (used here as a carry-in) and stores the result in <i>dst</i>. 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.</p> <p><b>addc</b> can be used for ordinal or integer arithmetic. <b>addc</b> 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.</p>	
<b>Action:</b>	<pre>dst = (src1 + src2 + AC.cc[1])[31:0]; AC.cc[2:0] = 000<sub>2</sub>; if((src2[31] == src1[31]) &amp;&amp; (src2[31] != dst[31]))     AC.cc[0] = 1;           # Set overflow bit. AC.cc[1] = (src2 + src1 + AC.cc[1])[32];   # Carry out.</pre>	
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-4</a> .
<b>Example:</b>	<pre># 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.</pre>	
<b>Opcode:</b>	<b>addc</b>	5B0H REG
<b>See Also:</b>	<b>ADD&lt;cc&gt;</b> , <b>SUB&lt;cc&gt;</b>	
<b>Side Effects:</b>	Sets the condition code in the arithmetic controls.	



### 6.2.3 **addi, addo**

<b>Mnemonic:</b>	<b>addo</b>	Add Ordinal		
	<b>addi</b>	Add Integer		
<b>Format:</b>	<b>add*</b>	<i>src1</i> , reg/lit	<i>src2</i> , reg/lit	<i>dst</i> reg
<b>Description:</b>	Adds <i>src2</i> and <i>src1</i> values and stores the result in <i>dst</i> . The binary results from these two instructions are identical. The only difference is that <b>addi</b> can signal an integer overflow.			
<b>Action:</b>	<p><b>addo:</b></p> <pre>dst = (src2 +src1)[31:0];</pre> <p><b>addi:</b></p> <pre>true_result = (src1 + src2); dst = true_result[31:0]; if((true_result &gt; (2**31) - 1)    (true_result &lt; -2**31))# Check for overflow {     if(AC.om == 1)         AC.of = 1;     else         generate_fault(ARITHMETIC.OVERFLOW); }</pre>			
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.		
	ARITHMETIC.OVERFLOW	Occurs only with <b>addi</b> .		
<b>Example:</b>	<code>addi r4, g5, r9     # r9 = g5 + r4</code>			
<b>Opcode:</b>	<b>addo</b>	590H	REG	
	<b>addi</b>	591H	REG	
<b>See Also:</b>	<b>addc, subi, subo, subc, ADD&lt;cc&gt;</b>			

## 6.2.4 alterbit

<b>Mnemonic:</b>	<b>alterbit</b>	Alter Bit		
<b>Format:</b>	<b>alterbit</b>	<i>bitpos</i> ,	<i>src</i> ,	<i>dst</i>
		reg/lit	reg/lit	reg
<b>Description:</b>	Copies <i>src</i> value to <i>dst</i> with one bit altered. <i>bitpos</i> operand specifies bit to be changed; condition code determines the value to which the bit is set. If condition code is X1X <sub>2</sub> , bit 1 = 1, the selected bit is set; otherwise, it is cleared. Typically this instruction is used to set the <i>bitpos</i> bit in the <i>targ</i> register if the result of a compare instruction is the equal condition code (010 <sub>2</sub> ).			
<b>Action:</b>	<pre>if((AC.cc &amp; 010<sub>2</sub>)==0)     dst = src &amp; ~(2**(bitpos%32)); else     dst = src   2**(bitpos%32);</pre>			
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.		
<b>Example:</b>	<pre># Assume AC.cc = 010<sub>2</sub>. alterbit 24, g4,g9 # g9 = g4, with bit 24 set.</pre>			
<b>Opcode:</b>	<b>alterbit</b>	58FH	REG	
<b>See Also:</b>	<b>chkbit, clrbit, notbit, setbit</b>			

## 6.2.5 and, andnot

<b>Mnemonic:</b>	<b>and</b>	And		
	<b>andnot</b>	And Not		
<b>Format:</b>	<b>and</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>
		reg/lit	reg/lit	reg
	<b>andnot</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>
		reg/lit	reg/lit	reg
<b>Description:</b>	Performs a bitwise AND ( <b>and</b> ) or AND NOT ( <b>andnot</b> ) operation on <i>src2</i> and <i>src1</i> values and stores result in <i>dst</i> . Note in the action expressions below, <i>src2</i> operand comes first, so that with <b>andnot</b> the expression is evaluated as:			
	$\{src2 \text{ and not } (src1)\}$ rather than $\{src1 \text{ and not } (src2)\}.$			
<b>Action:</b>	<b>and:</b>	dst = src2 & src1;		
	<b>andnot:</b>	dst = src2 & ~src1;		
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.		
<b>Example:</b>	<pre>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.</pre>			
<b>Opcode:</b>	<b>and</b>	581H	REG	
	<b>andnot</b>	582H	REG	
<b>See Also:</b>	<b>nand, nor, not, notand, notor, or, ornot, xnor, xor</b>			

## 6.2.6 atadd

<b>Mnemonic:</b>	<b>atadd</b>	Atomic Add	
<b>Format:</b>	<b>atadd</b>	<i>addr</i> , reg	<i>src</i> , reg/lit
<b>Description:</b>	<p>Adds <i>src</i> value (full word) to value in the memory location specified with <i>addr</i> 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 <i>dst</i>.</p> <p>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 <i>src/dst</i> operand until operation completes). See <a href="#">Section 3.5.1, “Memory Requirements”</a> on page 3-11 or more information on atomic accesses.</p> <p>Memory location in <i>addr</i> is the word’s first byte (LSB) address. Address is automatically aligned to a word boundary. (Note that <i>addr</i> operand maps to <i>src1</i> operand of the REG format.)</p>		
<b>Action:</b>	<pre>implicit_syncf(); tempa = addr &amp; 0xFFFFFFFF; temp = atomic_read(tempa); atomic_write(tempa, temp+src); dst = temp;</pre>		
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.	
<b>Example:</b>	<pre>atadd r8, r3, r11 # r8 contains the address of                   # memory location.                   # r11 = (r8)                   # (r8) = r11 + r3.</pre>		
<b>Opcode:</b>	<b>atadd</b>	612H	REG
<b>See Also:</b>	<b>atmod</b>		

## 6.2.7 atmod

<b>Mnemonic:</b>	<b>atmod</b>	Atomic Modify		
<b>Format:</b>	<b>atmod</b>	<i>addr</i> , reg	<i>mask</i> , reg/lit	<i>src/dst</i> reg
<b>Description:</b>	<p>Copies the selected bits of <i>src/dst</i> value into memory location specified in <i>addr</i>. The read-modify-write operation is performed on the actual data in memory and never on a cached value on chip. Bits set in <i>mask</i> operand select bits to be modified in memory. Initial value from memory is stored in <i>src/dst</i>. See <a href="#">Section 3.5.1, “Memory Requirements” on page 3-11</a> for information on atomic accesses.</p> <p>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 <i>src/dst</i> operand until operation completes).</p> <p>Memory location in <i>addr</i> is the modified word’s first byte (LSB) address. Address is automatically aligned to a word boundary.</p>			
<b>Action:</b>	<pre>implicit_syncf(); tempa = addr &amp; 0xFFFFFFFFFC; tempb = atomic_read(tempa); temp = (tempb &amp; ~ mask)   (src_dst &amp; mask); atomic_write(tempa, temp); src_dst = tempb;</pre>			
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-4</a> .		
<b>Example:</b>	<pre>atmod g5, g7, g10 # tempa = (g5)                   # temp = (tempa andnot g7) or                   # (g10 and g7)                   # (g5) = temp                   # g10 = tempa</pre>			
<b>Opcode:</b>	<b>atmod</b>	610H	REG	
<b>See Also:</b>	<b>atadd</b>			

## 6.2.8 **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 6-4.

**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

<b>Mnemonic:</b>	<b>bal</b>	Branch and Link	
	<b>balx</b>	Branch and Link Extended	
<b>Format:</b>	<b>bal</b>	<i>targ</i> disp	
	<b>balx</b>	<i>targ</i> , mem	<i>dst</i> reg
<b>Description:</b>	<p>Stores address of instruction following <b>bal</b> or <b>balx</b> in a register then branches to the instruction specified with the <i>targ</i> operand.</p> <p>The <b>bal</b> and <b>balx</b> 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>b</b> or <b>bx</b> 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.</p> <p>With <b>bal</b>, address of next instruction is stored in register g14. <i>targ</i> operand value can be no farther than <math>-2^{23}</math> to <math>(2^{23} - 4)</math> bytes from current IP. When using the Intel i960 processor assembler, <i>targ</i> must be a label which specifies the target instruction's IP.</p> <p><b>balx</b> performs same operation as <b>bal</b> except next instruction address is stored in <i>dst</i> (allowing the return IP to be stored in any available register). With <b>balx</b>, 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.</p> <p>See <a href="#">Section 2.3, "Memory Addressing Modes" on page 2-4</a> for a complete discussion of addressing modes available with memory-type operands.</p>		
<b>Action:</b>	<b>bal:</b>	<pre>g14 = IP + 4; IP[31:2] = effective_address(targ[31:2]); IP[1:0] = 0;</pre>	
	<b>balx:</b>	<pre>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;</pre>	
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, "Faults" on page 6-4</a> .	
<b>Example:</b>	<b>bal xyz</b>	# g14 = IP + 4	# IP = xyz
	<b>balx (g2), g4</b>	# g4 = IP + 4	# IP = (g2)
<b>Opcode:</b>	<b>bal</b>	0BH	CTRL
	<b>balx</b>	85H	MEM
<b>See Also:</b>	<b>b, bx, BRANCH&lt;cc&gt;, COMPARE AND BRANCH&lt;cc&gt;, bbc, bbs</b>		

## 6.2.10 **bbc, bbs**

<b>Mnemonic:</b>	<b>bbc</b>	Check Bit and Branch If Clear	
	<b>bbs</b>	Check Bit and Branch If Set	
<b>Format:</b>	<b>bb*</b>	<i>bitpos</i> ,	<i>src</i> , <i>targ</i> reg/lit            reg            disp
<b>Description:</b>	<p>Checks bit (designated by <i>bitpos</i>) in <i>src</i> and sets AC register condition code according to <i>src</i> value. The processor then performs conditional branch to instruction specified with <i>targ</i>, based on condition code state.</p> <p>For <b>bbc</b>, if selected bit in <i>src</i> is clear, the processor sets condition code to 000<sub>2</sub> and branches to instruction specified by <i>targ</i>; otherwise, it sets condition code to 010<sub>2</sub> and goes to next instruction.</p> <p>For <b>bbs</b>, if selected bit is set, the processor sets condition code to 010<sub>2</sub> and branches to <i>targ</i>; otherwise, it sets condition code to 000<sub>2</sub> and goes to next instruction.</p> <p><i>targ</i> can be no farther than <math>-2^{12}</math> to <math>(2^{12} - 4)</math> bytes from current IP. When using the Intel i960 processor assembler, <i>targ</i> must be a label which specifies target instruction's IP.</p>		
<b>Action:</b>	<b>bbs:</b>	<pre>if((src &amp; 2**(bitpos%32)) == 1) {   AC.cc = 010<sub>2</sub>;   temp[31:2] = sign_extension(targ[12:2]);   IP[31:2] = IP[31:2] + temp[31:2];   IP[1:0] = 0; } else   AC.cc = 000<sub>2</sub>;</pre>	
	<b>bbc:</b>	<pre>if((src &amp; 2**(bitpos%32)) == 0) {   AC.cc = 000<sub>2</sub>;   temp[31:2] = sign_extension(targ[12:2]);   IP[31:2] = IP[31:2] + temp[31:2];   IP[1:0] = 0; } else   AC.cc = 010<sub>2</sub>;</pre>	
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, "Faults"</a> on page 6-4.	
<b>Example:</b>	<pre># 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;</pre>		
<b>Opcode:</b>	<b>bbc</b>	30H	COBR
	<b>bbs</b>	37H	COBR
<b>See Also:</b>	<b>chkbit, COMPARE AND BRANCH&lt;cc&gt;, BRANCH&lt;cc&gt;</b>		
<b>Side Effects:</b>	Sets the condition code in the arithmetic controls.		



## 6.2.11 BRANCH<cc>

<b>Mnemonic:</b>	<b>be</b>	Branch If Equal
	<b>bne</b>	Branch If Not Equal
	<b>bl</b>	Branch If Less
	<b>ble</b>	Branch If Less Or Equal
	<b>bg</b>	Branch If Greater
	<b>bge</b>	Branch If Greater Or Equal
	<b>bo</b>	Branch If Ordered
	<b>bno</b>	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
<b>bno</b>	$000_2$	Unordered
<b>bg</b>	$001_2$	Greater
<b>be</b>	$010_2$	Equal
<b>bge</b>	$011_2$	Greater or equal
<b>bl</b>	$100_2$	Less
<b>bne</b>	$101_2$	Not equal
<b>ble</b>	$110_2$	Less or equal
<b>bo</b>	$111_2$	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 6-4.



**Example:**           # Assume (AC.cc AND 100<sub>2</sub>) ≠ 0  
 bl xyz                           # IP = xyz;

**Opcode:**

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

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

## 6.2.12 bswap

<b>Mnemonic:</b>	<b>bswap</b>	Byte Swap
<b>Format:</b>	<b>bswap</b>	<i>src1:src,</i> <i>src2:dst</i> reg/lit                    reg
<b>Description:</b>	Alters the order of bytes in a word, reversing its “endianess.”	
	Copies bytes 3:0 of <i>src1</i> to <i>src2</i> reversing order of the bytes. Byte 0 of <i>src1</i> becomes byte 3 of <i>src2</i> , byte 1 of <i>src1</i> becomes byte 2 of <i>src2</i> , etc.	
<b>Action:</b>	dst = (rotate_left(src 8) & 0x00FF00FF) +(rotate_left(src 24) & 0xFF00FF00);	
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.
<b>Example:</b>	<pre> # g8 = 0x89ABCDEF bswap g8, g10      # Reverse byte order.                    # g10 now 0xEFCDAB89 </pre>	
<b>Opcode:</b>	<b>bswap</b>	5ADH            REG
<b>See Also:</b>	<b>scanbyte, rotate</b>	
<b>Notes:</b>	This core instruction is not implemented on 80960Cx, Kx and Sx processors.	

## 6.2.13 call

<b>Mnemonic:</b>	<b>call</b>	Call
<b>Format:</b>	<b>call</b>	<i>targ</i> disp
<b>Description:</b>	<p>Calls a new procedure. <i>targ</i> operand specifies the IP of called procedure's first instruction. When using the Intel i960 processor assembler, <i>targ</i> must be a label.</p> <p>In executing this instruction, the processor performs a local call operation as described in <a href="#">Section 7.1.3.1, "Call Operation" on page 7-6</a>. 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 <i>targ</i> and begins execution.</p> <p><i>targ</i> can be no farther than <math>-2^{23}</math> to <math>(2^{23} - 4)</math> bytes from current IP.</p>	
<b>Action:</b>	<pre># Wait for any uncompleted instructions to finish. implicit_syncf(); temp = (SP + (SALIGN*16 - 1)) &amp; ~(SALIGN*16 - 1) # Round stack pointer to next boundary. # SALIGN=1 on 80960RM/RN. 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;</pre>	
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, "Faults" on page 6-4</a> .
<b>Example:</b>	call xyz	# IP = xyz
<b>Opcode:</b>	<b>call</b>	09H CTRL
<b>See Also:</b>	<b>bal, calls, callx</b>	

## 6.2.14 calls

**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 7-14](#). *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 80960RM/RN.  
                          temp.RRR = 000<sub>2</sub>;  
                      }  
                      else        # Supervisor call from user mode.  
                      {    tempa = SSP;                    # Get Supervisor Stack pointer.  
                          temp.RRR = 010<sub>2</sub> | PC.te;  
                          PC.em = supervisor;  
                          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 6-4.  
 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**

<b>Mnemonic:</b>	<b>callx</b>	Call Extended
<b>Format:</b>	<b>callx</b>	<i>targ</i> mem
<b>Description:</b>	<p>Calls new procedure. <i>targ</i> specifies IP of called procedure's first instruction.</p> <p>In executing <b>callx</b>, the processor performs a local call as described in <a href="#">Section 7.1.3.1, "Call Operation" on page 7-6</a>. 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 <i>targ</i> and begins execution of new procedure.</p> <p><b>callx</b> performs the same operation as call except the target instruction can be farther than <math>-2^{23}</math> to <math>(2^{23} - 4)</math> bytes from current IP.</p> <p>The <i>targ</i> 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.</p> <p>Refer to <a href="#">Chapter 2, "Data Types and Memory Addressing Modes"</a> for more information.</p>	
<b>Action:</b>	<pre># Wait for any uncompleted instructions to finish; implicit_syncf(); temp = (SP + (SALIGN*16 - 1)) &amp; ~(SALIGN*16 - 1)       # Round stack pointer to next boundary.       # SALIGN=1 on 80960RM/RN. 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;</pre>	
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, "Faults" on page 6-4</a> .
<b>Example:</b>	<pre>callx (g5)  # IP = (g5), where the address in g5             # is the address of the new procedure.</pre>	
<b>Opcode:</b>	<b>callx</b>	86H MEM
<b>See Also:</b>	<b>bal, call, calls, ret</b>	

## 6.2.16 **chkbit**

<b>Mnemonic:</b>	<b>chkbit</b>	Check Bit
<b>Format:</b>	<b>chkbit</b>	<i>bitpos</i> , <i>src2</i> reg/lit reg/lit
<b>Description:</b>	Checks bit in <i>src2</i> designated by <i>bitpos</i> and sets condition code according to value found. If bit is set, condition code is set to 010 <sub>2</sub> ; if bit is clear, condition code is set to 000 <sub>2</sub> .	
<b>Action:</b>	<pre>if (((src2 &amp; 2**(<i>bitpos</i> % 32)) == 0)     AC.cc = 000<sub>2</sub>; else     AC.cc = 010<sub>2</sub>;</pre>	
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.
<b>Example:</b>	<code>chkbit 13, g8</code>	# Checks bit 13 in g8 and sets # AC.cc according to the result.
<b>Opcode:</b>	<b>chkbit</b>	5AEH REG
<b>See Also:</b>	<b>alterbit, clrbit, notbit, setbit, cmpi, cmpo</b>	
<b>Side Effects:</b>	Sets the condition code in the arithmetic controls.	



## 6.2.17 **clrbt**

<b>Mnemonic:</b>	<b>clrbt</b>	Clear Bit		
<b>Format:</b>	<b>clrbt</b>	<i>bitpos</i> , reg/lit	<i>src</i> , reg/lit	<i>dst</i> reg
<b>Description:</b>	Copies <i>src</i> value to <i>dst</i> with one bit cleared. <i>bitpos</i> operand specifies bit to be cleared.			
<b>Action:</b>	$dst = src \& \sim(2^{*(bitpos\%32)})$ ;			
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.		
<b>Example:</b>	clrbt 23, g3, g6 # g6 = g3 with bit 23 cleared.			
<b>Opcode:</b>	<b>clrbt</b>	58CH	REG	
<b>See Also:</b>	<b>alterbit, chkbit, notbit, setbit</b>			

## 6.2.18 **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 <sub>2</sub>	<i>src1</i> < <i>src2</i>
010 <sub>2</sub>	<i>src1</i> = <i>src2</i>
001 <sub>2</sub>	<i>src1</i> > <i>src2</i>

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 6-4](#).

**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**

**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 <sub>2</sub>	<i>src1</i> < <i>src2</i>
010 <sub>2</sub>	<i>src1</i> = <i>src2</i>
001 <sub>2</sub>	<i>src1</i> > <i>src2</i>

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 6-4](#).

**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

**Mnemonic:**

<b>cmpi</b>	Compare Integer
<b>cmpib</b>	Compare Integer Byte
<b>cmpis</b>	Compare Integer Short
<b>cmpo</b>	Compare Ordinal
<b>cmpob</b>	Compare Ordinal Byte
<b>cmpos</b>	Compare Ordinal Short

**Format:**

<b>cmp*</b>	<i>src1</i> ,	<i>src2</i>
	reg/lit	reg/lit

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

Condition Code	Comparison
100 <sub>2</sub>	<i>src1</i> < <i>src2</i>
010 <sub>2</sub>	<i>src1</i> = <i>src2</i>
001 <sub>2</sub>	<i>src1</i> > <i>src2</i>

**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 6-4](#).

**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:**

<b>cmpi</b>	5A1H	REG
<b>cmpib</b>	595H	REG
<b>cmpis</b>	597H	REG
<b>cmpo</b>	5A0H	REG
<b>cmpob</b>	594H	REG
<b>cmpos</b>	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>

<b>Mnemonic:</b>	<b>cmpibe</b>	Compare Integer and Branch If Equal
	<b>cmpibne</b>	Compare Integer and Branch If Not Equal
	<b>cmpibl</b>	Compare Integer and Branch If Less
	<b>cmpible</b>	Compare Integer and Branch If Less Or Equal
	<b>cmpibg</b>	Compare Integer and Branch If Greater
	<b>cmpibge</b>	Compare Integer and Branch If Greater Or Equal
	<b>cmpibo</b>	Compare Integer and Branch If Ordered
	<b>cmpibno</b>	Compare Integer and Branch If Not Ordered
	<b>cmpobe</b>	Compare Ordinal and Branch If Equal
	<b>cmpobne</b>	Compare Ordinal and Branch If Not Equal
	<b>cmpobl</b>	Compare Ordinal and Branch If Less
	<b>cmpoble</b>	Compare Ordinal and Branch If Less Or Equal
	<b>cmpobg</b>	Compare Ordinal and Branch If Greater
	<b>cmpobge</b>	Compare Ordinal and Branch If Greater Or Equal

<b>Format:</b>	<b>cmpib*</b>	<i>src1</i> , reg/lit	<i>src2</i> , reg	<i>targ</i> disp
	<b>cmpob*</b>	<i>src1</i> , reg/lit	<i>src2</i> , reg	<i>targ</i> 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 6-28.

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
<b>cmpibno</b>	000 <sub>2</sub>	No Condition
<b>cmpibg</b>	001 <sub>2</sub>	$src1 > src2$
<b>cmpibe</b>	010 <sub>2</sub>	$src1 = src2$
<b>cmpibge</b>	011 <sub>2</sub>	$src1 \geq src2$
<b>cmpibl</b>	100 <sub>2</sub>	$src1 < src2$
<b>cmpibne</b>	101 <sub>2</sub>	$src1 \neq src2$
<b>cmpible</b>	110 <sub>2</sub>	$src1 \leq src2$
<b>cmpibo</b>	111 <sub>2</sub>	Any Condition
<b>cmpobg</b>	001 <sub>2</sub>	$src1 > src2$
<b>cmpobe</b>	010 <sub>2</sub>	$src1 = src2$
<b>cmpobge</b>	011 <sub>2</sub>	$src1 \geq src2$
<b>cmpobl</b>	100 <sub>2</sub>	$src1 < src2$
<b>cmpobne</b>	101 <sub>2</sub>	$src1 \neq src2$
<b>cmpoble</b>	110 <sub>2</sub>	$src1 \leq src2$

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

<b>Action:</b>	<pre> if(src1 &lt; src2)     AC.cc = 100<sub>2</sub>; else if(src1 == src2)     AC.cc = 010<sub>2</sub>; else     AC.cc = 001<sub>2</sub>; if((mask &amp;&amp; AC.cc) != 000<sub>2</sub>)     IP[31:2] = efa[31:2];    # Resume execution at the new IP.     IP[1:0] = 0;                 </pre>																																										
<b>Faults:</b>	STANDARD <span style="float: right;">Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.</span>																																										
<b>Example:</b>	<pre> # Assume g3 &lt; g9 cmpibl g3, g9, xyz # g9 is compared with g3;                   # IP = xyz.  # assume 19 ≥ r7 cmpobge 19, r7, xyz # 19 is compared with r7;                   # IP = xyz.                 </pre>																																										
<b>Opcode:</b>	<table border="0"> <tr><td><b>cmpibe</b></td><td>3AH</td><td>COBR</td></tr> <tr><td><b>cmpibne</b></td><td>3DH</td><td>COBR</td></tr> <tr><td><b>cmpibl</b></td><td>3CH</td><td>COBR</td></tr> <tr><td><b>cmpible</b></td><td>3EH</td><td>COBR</td></tr> <tr><td><b>cmpibg</b></td><td>39H</td><td>COBR</td></tr> <tr><td><b>cmpibge</b></td><td>3BH</td><td>COBR</td></tr> <tr><td><b>cmpibo</b></td><td>3FH</td><td>COBR</td></tr> <tr><td><b>cmpibno</b></td><td>38H</td><td>COBR</td></tr> <tr><td><b>cmpobe</b></td><td>32H</td><td>COBR</td></tr> <tr><td><b>cmpobne</b></td><td>35H</td><td>COBR</td></tr> <tr><td><b>cmpobl</b></td><td>34H</td><td>COBR</td></tr> <tr><td><b>cmpoble</b></td><td>36H</td><td>COBR</td></tr> <tr><td><b>cmpobg</b></td><td>31H</td><td>COBR</td></tr> <tr><td><b>cmpobge</b></td><td>33H</td><td>COBR</td></tr> </table>	<b>cmpibe</b>	3AH	COBR	<b>cmpibne</b>	3DH	COBR	<b>cmpibl</b>	3CH	COBR	<b>cmpible</b>	3EH	COBR	<b>cmpibg</b>	39H	COBR	<b>cmpibge</b>	3BH	COBR	<b>cmpibo</b>	3FH	COBR	<b>cmpibno</b>	38H	COBR	<b>cmpobe</b>	32H	COBR	<b>cmpobne</b>	35H	COBR	<b>cmpobl</b>	34H	COBR	<b>cmpoble</b>	36H	COBR	<b>cmpobg</b>	31H	COBR	<b>cmpobge</b>	33H	COBR
<b>cmpibe</b>	3AH	COBR																																									
<b>cmpibne</b>	3DH	COBR																																									
<b>cmpibl</b>	3CH	COBR																																									
<b>cmpible</b>	3EH	COBR																																									
<b>cmpibg</b>	39H	COBR																																									
<b>cmpibge</b>	3BH	COBR																																									
<b>cmpibo</b>	3FH	COBR																																									
<b>cmpibno</b>	38H	COBR																																									
<b>cmpobe</b>	32H	COBR																																									
<b>cmpobne</b>	35H	COBR																																									
<b>cmpobl</b>	34H	COBR																																									
<b>cmpoble</b>	36H	COBR																																									
<b>cmpobg</b>	31H	COBR																																									
<b>cmpobge</b>	33H	COBR																																									
<b>See Also:</b>	<b>BRANCH&lt;cc&gt;, cmpi, cmpo, bal, balx</b>																																										
<b>Side Effects:</b>	Sets the condition code in the arithmetic controls.																																										

## 6.2.22 **concmpi, concmpo**

<b>Mnemonic:</b>	<b>concmpi</b>	Conditional Compare Integer
	<b>concmpo</b>	Conditional Compare Ordinal
<b>Format:</b>	<b>concmp*</b>	<i>src1</i> , <i>src2</i> reg/lit reg/lit
<b>Description:</b>	<p>Compares <i>src2</i> and <i>src1</i> 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.</p> <p>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.</p> <p>The example below illustrates this application by testing whether <i>g3</i> value is between <i>g5</i> and <i>g6</i> values, where <i>g5</i> is assumed to be less than <i>g6</i>. First a comparison (<b>cmpo</b>) of <i>g3</i> and <i>g6</i> is performed. If <i>g3</i> is less than or equal to <i>g6</i> (i.e., condition code is either 010<sub>2</sub> or 001<sub>2</sub>), a conditional comparison (<b>concmpo</b>) of <i>g3</i> and <i>g5</i> is then performed. If <i>g3</i> is greater than or equal to <i>g5</i> (indicating that <i>g3</i> is within the bounds of <i>g5</i> and <i>g6</i>), condition code is set to 010<sub>2</sub>; otherwise, it is set to 001<sub>2</sub>.</p>	
<b>Action:</b>	<pre>if (AC.cc != 1XX<sub>2</sub>) {   if(src1 &lt;= src2)     AC.cc = 010<sub>2</sub>;     else     AC.cc = 001<sub>2</sub>; }</pre>	
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-4.</a>
<b>Example:</b>	<pre>cmpo g6, g3           # Compares g6 and g3                        # and sets AC.cc. concmpo g5, g3       # If AC.cc &lt; 100<sub>2</sub> (g6 &gt;= g3)                        # g5 is compared with g3.</pre>	

At this point, depending on the register ordering, the condition code is one of those listed on [Table 6-5](#).

**Table 6-5. *concmpo* Example: Register Ordering and CC**

Order	CC
<i>g5</i> < <i>g6</i> < <i>g3</i>	100 <sub>2</sub>
<i>g5</i> < <i>g6</i> = <i>g3</i>	010 <sub>2</sub>
<i>g5</i> < <i>g3</i> < <i>g6</i>	010 <sub>2</sub>
<i>g5</i> = <i>g3</i> < <i>g6</i>	010 <sub>2</sub>
<i>g3</i> < <i>g5</i> < <i>g6</i>	001 <sub>2</sub>

<b>Opcode:</b>	<b>concmpi</b>	5A3H	REG
	<b>concmpo</b>	5A2H	REG
<b>See Also:</b>	<b>cmpo, cmpi, cmpdeci, cmpdeco, cmpinci, cmpinco, COMPARE AND BRANCH&lt;cc&gt;</b>		
<b>Side Effects:</b>	Sets the condition code in the arithmetic controls.		



## 6.2.23 dcctl

**Mnemonic:** **dcctl** Data-cache Control

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

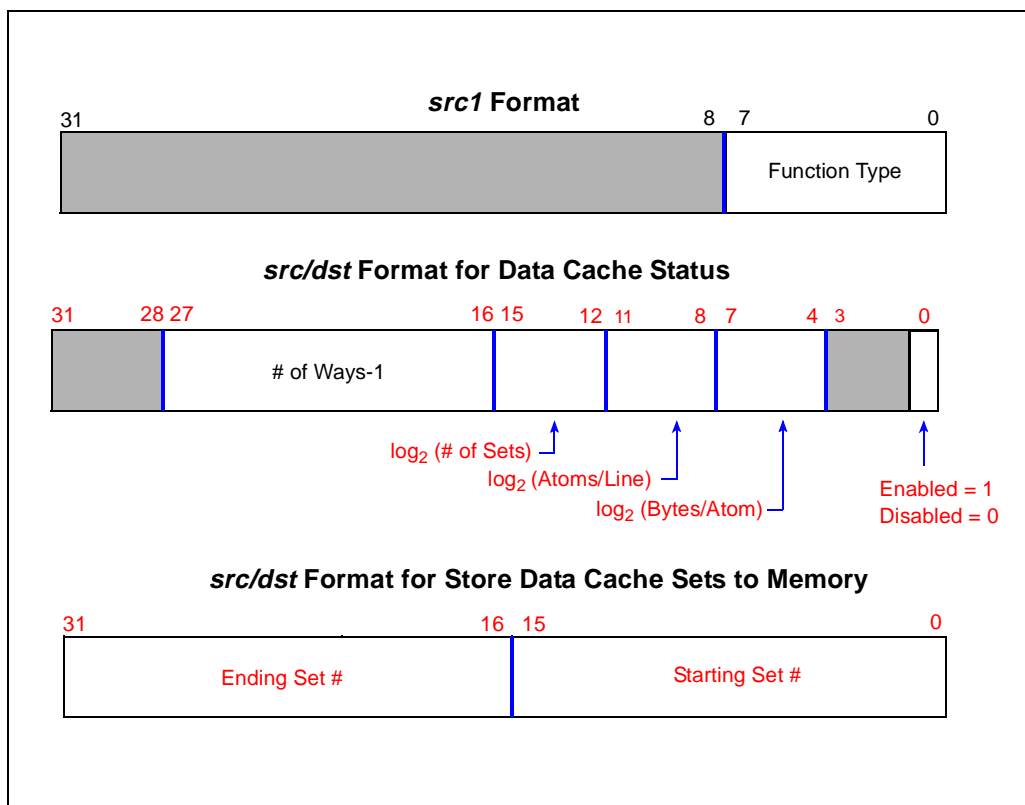
**Description:** Performs management and control of the data cache including disabling, enabling, invalidating, ensuring coherency, getting status, and storing cache contents to memory. Operations are indicated by the value of *src1*. *src2* and *src/dst* are also used by some operations. When needed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

**Table 6-6. 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 coherency <sup>1</sup>	3	NA	NA
Get D-cache status	4	NA	<i>src</i> : NA <i>dst</i> : Receives D-cache status (Figure 6-1).
Reserved	5	NA	NA
Store D-cache to memory	6	Destination address for cache sets	<i>src</i> : 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 80960RM/RN.

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



**Table 6-7. dcctl Status Values and D-Cache Parameters**

Value	Value on 80960RM/RN
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] (number of ways - 1)	0

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

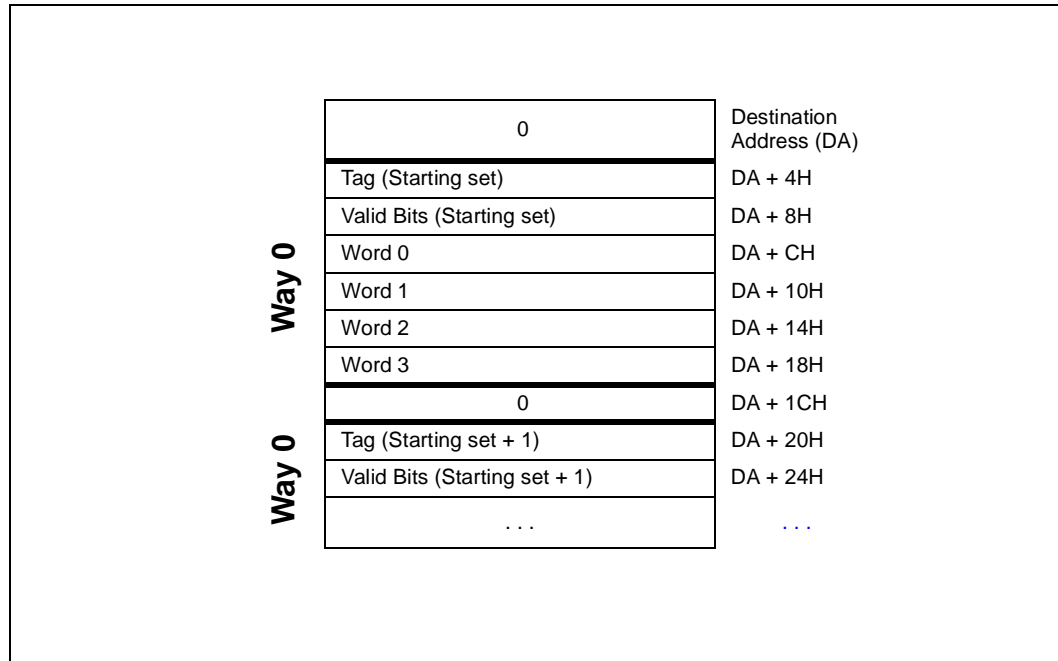
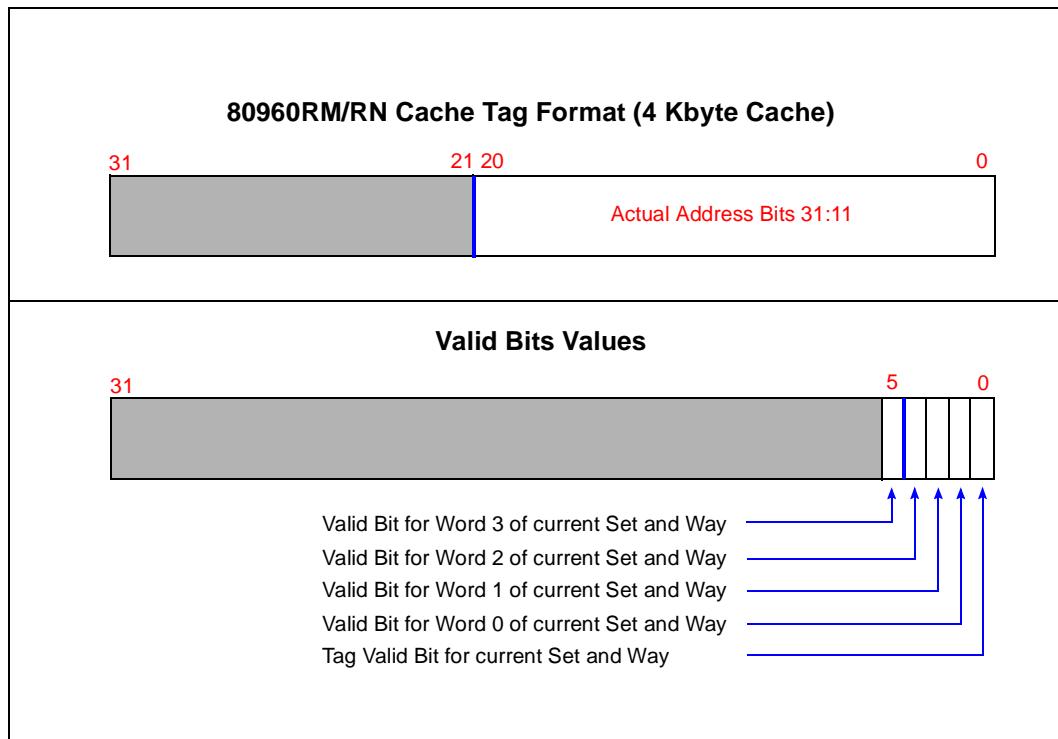


Figure 6-3. D-Cache Tag and Valid Bit Formats



```

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);
    
```

<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.
	TYPE.MISMATCH	Attempt to execute instruction while not in supervisor mode.
	OPERATION.INVALID_OPERAND	
<b>Example:</b>		<pre> # g0 = 6, g1 = 0x10000000, # g2 = 0x001F0001 dcctl g0,g1,g2 # Store the status of D-cache # sets 1-0x1F to memory starting # at 0x10000000.                 </pre>
<b>Opcode:</b>	<b>dcctl</b>	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 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

<b>Mnemonic:</b>	<b>divi</b>	Divide Integer		
	<b>divo</b>	Divide Ordinal		
<b>Format:</b>	<b>div*</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>
		reg/lit	reg/lit	reg
<b>Description:</b>	Divides <i>src2</i> value by <i>src1</i> value and stores the result in <i>dst</i> . Remainder is discarded.			
	For <b>divi</b> , an integer-overflow fault can be signaled.			
<b>Action:</b>	<p><b>divo:</b></p> <pre> if (src1 == 0) {     dst = undefined_value;     generate_fault (ARITHMETIC.ZERO_DIVIDE); } else     dst = src2/src1;                     </pre> <p><b>divi:</b></p> <pre> if (src1 == 0) {     dst = undefined_value;     generate_fault (ARITHMETIC.ZERO_DIVIDE);} else if ((src2 == -2**31) &amp;&amp; (src1 == -1)) {     dst = -2**31      if (AC.om == 1)         AC.of = 1;     else         generate_fault (ARITHMETIC.OVERFLOW); } else     dst = src2 / src1;                     </pre>			
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.		
	ARITHMETIC.ZERO_DIVIDE	The <i>src1</i> operand is 0.		
	ARITHMETIC.OVERFLOW	Result too large for destination register ( <b>divi</b> 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 <i>dst</i> .		
<b>Example:</b>	divo r3, r8, r13 # r13 = r8/r3			
<b>Opcode:</b>	<b>divi</b>	74BH	REG	
	<b>divo</b>	70BH	REG	
<b>See Also:</b>	<b>ediv, mulo, muli, emul</b>			

## 6.2.25 **ediv**

<b>Mnemonic:</b>	<b>ediv</b>	Extended Divide
<b>Format:</b>	<b>ediv</b>	<i>src1</i> , <i>src2</i> , <i>dst</i> reg/lit reg/lit reg
<b>Description:</b>	<p>Divides <i>src2</i> by <i>src1</i> and stores result in <i>dst</i>. The <i>src2</i> value is a long ordinal (64 bits) contained in two adjacent registers. <i>src2</i> specifies the lower numbered register which contains operand's least significant bits. <i>src2</i> must be an even numbered register (i.e., g0, g2, ... or r4, r6, r8... ). <i>src1</i> value is a normal ordinal (i.e., 32 bits).</p> <p>The result consists of a one-word remainder and a one-word quotient. Remainder is stored in the register designated by <i>dst</i>; quotient is stored in the next highest numbered register. <i>dst</i> must be an even numbered register (i.e., g0, g2, ... r4, r6, r8, ...).</p> <p>This instruction performs ordinal arithmetic.</p> <p>If this operation overflows (quotient or remainder do not fit in 32 bits), no fault is raised and the result is undefined.</p>	
<b>Action:</b>	<pre>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); } </pre>	
<b>Faults:</b>	STANDARD ARITHMETIC.ZERO_DIVIDE	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-4</a> . The <i>src1</i> operand is 0.
<b>Example:</b>	<pre>ediv g3, g4, g10 # g10 = remainder of g4,g5/g3                 # g11 = quotient of g4,g5/g3 </pre>	
<b>Opcode:</b>	<b>ediv</b>	671H REG
<b>See Also:</b>	<b>emul, divi, divo</b>	



## 6.2.26 **emul**

<b>Mnemonic:</b>	<b>emul</b>	Extended Multiply
<b>Format:</b>	<b>emul</b>	<i>src1</i> , <i>src2</i> , <i>dst</i> reg/lit reg/lit reg
<b>Description:</b>	Multiplies <i>src2</i> by <i>src1</i> and stores the result in <i>dst</i> . Result is a long ordinal (64 bits) stored in two adjacent registers. <i>dst</i> specifies lower numbered register, which receives the result's least significant bits. <i>dst</i> must be an even numbered register (i.e., g0, g2, ... r4, r6, r8, ...).	
	This instruction performs ordinal arithmetic.	
<b>Action:</b>	<pre>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]; } </pre>	
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.
<b>Example:</b>	emul r4, r5, g2 # g2,g3 = r4 * r5.	
<b>Opcode:</b>	<b>emul</b>	670H REG
<b>See Also:</b>	<b>ediv, muli, mulo</b>	

## 6.2.27 **eshro**

<b>Mnemonic:</b>	<b>eshro</b>	Extended Shift Right Ordinal		
<b>Format:</b>	<b>eshro</b>	<i>src1</i> , reg/lit	<i>src2</i> , reg/lit	<i>dst</i> reg
<b>Description:</b>	Shifts <i>src2</i> right by ( <i>src1</i> <b>mod</b> 32) places and stores the result in <i>dst</i> . Bits shifted beyond the least-significant bit are discarded.			
	<i>src2</i> value is a long ordinal (i.e., 64 bits) contained in two adjacent registers. <i>src2</i> operand specifies the lower numbered register, which contains operand's least significant bits. <i>src2</i> operand must be an even numbered register (i.e., r4, r6, r8, ... or g0, g2).			
	<i>src1</i> operand is a single 32-bit register or literal where the lower 5 bits specify the number of places that the <i>src2</i> operand is to be shifted.			
	The least significant 32 bits of the shift operation result are stored in <i>dst</i> .			
<b>Action:</b>	<pre> 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]; </pre>			
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-4</a> .		
<b>Example:</b>	eshro g3, g4, g11 # g11 = g4,5 shifted right by # (g3 MOD 32).			
<b>Opcode:</b>	<b>eshro</b>	5D8H	REG	
<b>See Also:</b>	<b>SHIFT, extract</b>			
<b>Notes:</b>	This core instruction is not implemented on the 80960Kx and Sx processors.			

## 6.2.28 **extract**

<b>Mnemonic:</b>	<b>extract</b>	Extract		
<b>Format:</b>	<b>extract</b>	<i>bitpos</i> reg/lit	<i>len</i> reg/lit	<i>src/dst</i> reg
<b>Description:</b>	Shifts a specified bit field in <i>src/dst</i> right and zero fills bits to left of shifted bit field. <i>bitpos</i> value specifies the least significant bit of the bit field to be shifted; <i>len</i> value specifies bit field length.			
<b>Action:</b>	<pre>src_dst = (src_dst &gt;&gt; min(bitpos, 32))           &amp; ~ (0xFFFFFFFF &lt;&lt; len);</pre>			
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.		
<b>Example:</b>	<pre>extract 5, 12, g4 # g4 = g4 with bits 5 through                   # 16 shifted right.</pre>			
<b>Opcode:</b>	<b>extract</b>	651H	REG	
<b>See Also:</b>	<b>modify</b>			

## 6.2.29 FAULT<cc>

<b>Mnemonic:</b>	<b>faulte</b>	Fault If Equal
	<b>faultne</b>	Fault If Not Equal
	<b>faultl</b>	Fault If Less
	<b>faultle</b>	Fault If Less Or Equal
	<b>faultg</b>	Fault If Greater
	<b>faultge</b>	Fault If Greater Or Equal
	<b>faulto</b>	Fault If Ordered
	<b>faultno</b>	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 following table shows the condition-code mask for each instruction. The mask is opcode bits 0-2.

Instruction	Mask	Condition
<b>faultno</b>	$000_2$	Unordered
<b>faultg</b>	$001_2$	Greater
<b>faulte</b>	$010_2$	Equal
<b>faultge</b>	$011_2$	Greater or equal
<b>faultl</b>	$100_2$	Less
<b>faultne</b>	$101_2$	Not equal
<b>faultle</b>	$110_2$	Less or equal
<b>faulto</b>	$111_2$	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 6-4.](#)  
 CONSTRAINT.RANGE If condition being tested is true.

**Example:**  
 # Assume (AC.cc AND  $110_2$ ) ≠  $000_2$   
 faultle # Generate CONSTRAINT\_RANGE fault

<b>Opcode:</b>	<b>faulte</b>	1AH	CTRL
	<b>faultne</b>	1DH	CTRL
	<b>faultl</b>	1CH	CTRL
	<b>faultle</b>	1EH	CTRL
	<b>faultg</b>	19H	CTRL
	<b>faultge</b>	1BH	CTRL
	<b>faulto</b>	1FH	CTRL
	<b>faultno</b>	18H	CTRL

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

## 6.2.30 flushreg

<b>Mnemonic:</b>	<b>flushreg</b>	Flush Local Registers
<b>Format:</b>	<b>flushreg</b>	
<b>Description:</b>	<p>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.</p> <p><b>flushreg</b> 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 <b>flushreg</b> must be executed prior to modifying the PFP to return to a frame other than the one directly below the current frame.</p> <p>To reduce interrupt latency, <b>flushreg</b> is abortable. If an interrupt of higher priority than the current process is detected while <b>flushreg</b> is executing, <b>flushreg</b> flushes at least one frame and aborts. After executing the interrupt handler, the processor returns to the <b>flushreg</b> instruction and re-executes it. <b>flushreg</b> does not reflush any frames that were flushed before the interrupt occurred. <b>flushreg</b> 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.</p>	
<b>Action:</b>	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.	
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, "Faults"</a> on page 6-4.
<b>Example:</b>	flushreg	
<b>Opcode:</b>	<b>flushreg</b>	66DH                      REG

## 6.2.31 fmark

<b>Mnemonic:</b>	<b>fmark</b>	Force Mark
<b>Format:</b>	<b>fmark</b>	
<b>Description:</b>	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 <a href="#">Chapter 10, “Tracing and Debugging”</a> .	
<b>Action:</b>	A mark trace event is generated, independent of the setting of the mark-trace-mode flag.	
<b>Faults:</b>	STANDARD TRACE.MARK	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4. A TRACE.MARK fault is generated if PC.te=1.
<b>Example:</b>	<pre># Assume PC.te = 1 fmark # Mark trace event is generated at this point in the # instruction stream.</pre>	
<b>Opcode:</b>	<b>fmark</b>	66CH REG
<b>See Also:</b>	<b>mark</b>	

## 6.2.32 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_quiescent;
enter_HALT_mode;

```

**Faults:** STANDARD Refer to [Section 6.1.6, “Faults” on page 6-4](#).  
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 80960RM/RN 80960RP/RD, and 80960Jx processor families only, and may or may not be implemented on future i960 processors.



### 6.2.33 icctl

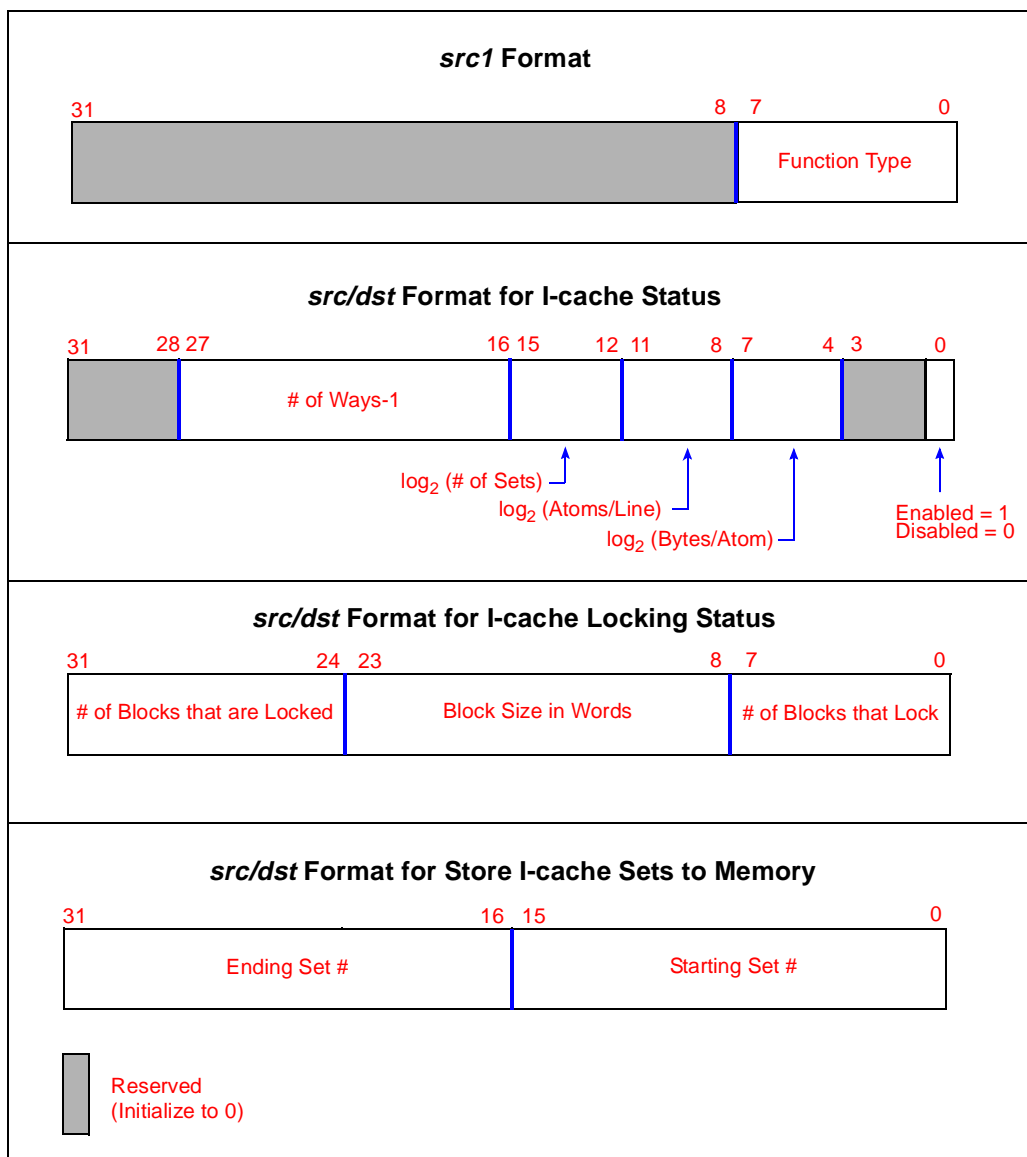
<b>Mnemonic:</b>	<b>icctl</b>	Instruction-cache Control		
<b>Format:</b>	<b>icctl</b>	<i>src1</i> , reg/lit	<i>src2</i> , reg/lit	<i>src/dst</i> reg

**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 6-8. icctl Operand Fields**

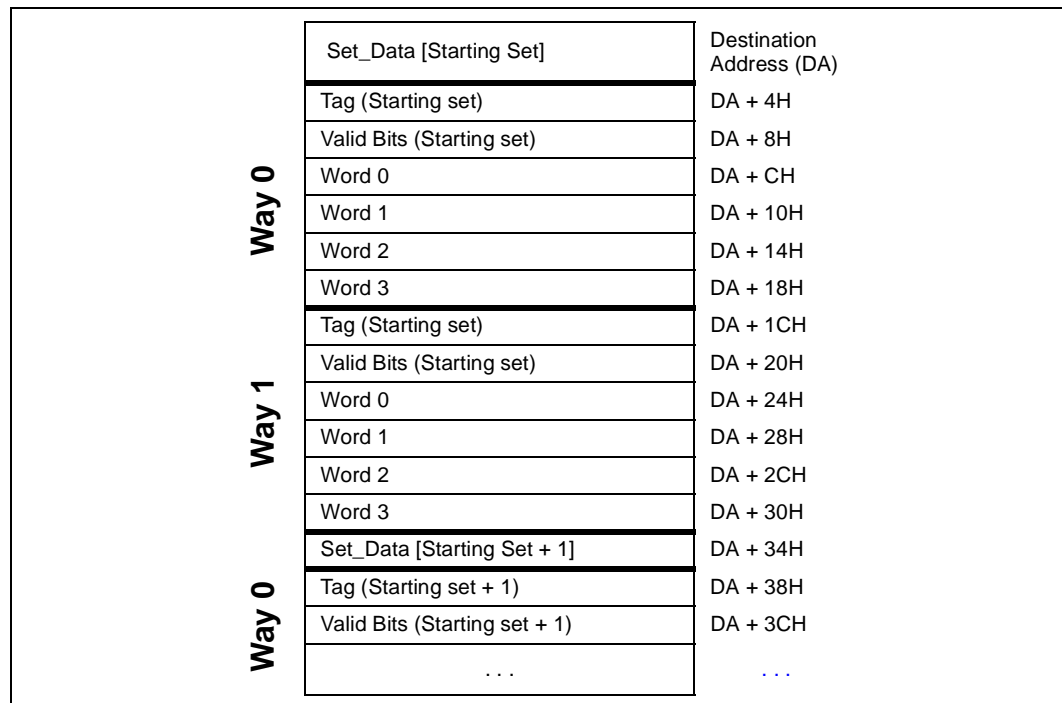
Function	src1	src2	src/dst
Disable I-cache	0	NA	NA
Enable I-cache	1	NA	NA
Invalidate I-cache	2	NA	NA
Load and lock I-cache	3	<i>src</i> : Starting address of code to lock.	Number of blocks to lock.
Get I-cache status	4	NA	<i>dst</i> : Receives status (Figure 6-4).
Get I-cache locking status	5	NA	<i>dst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

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

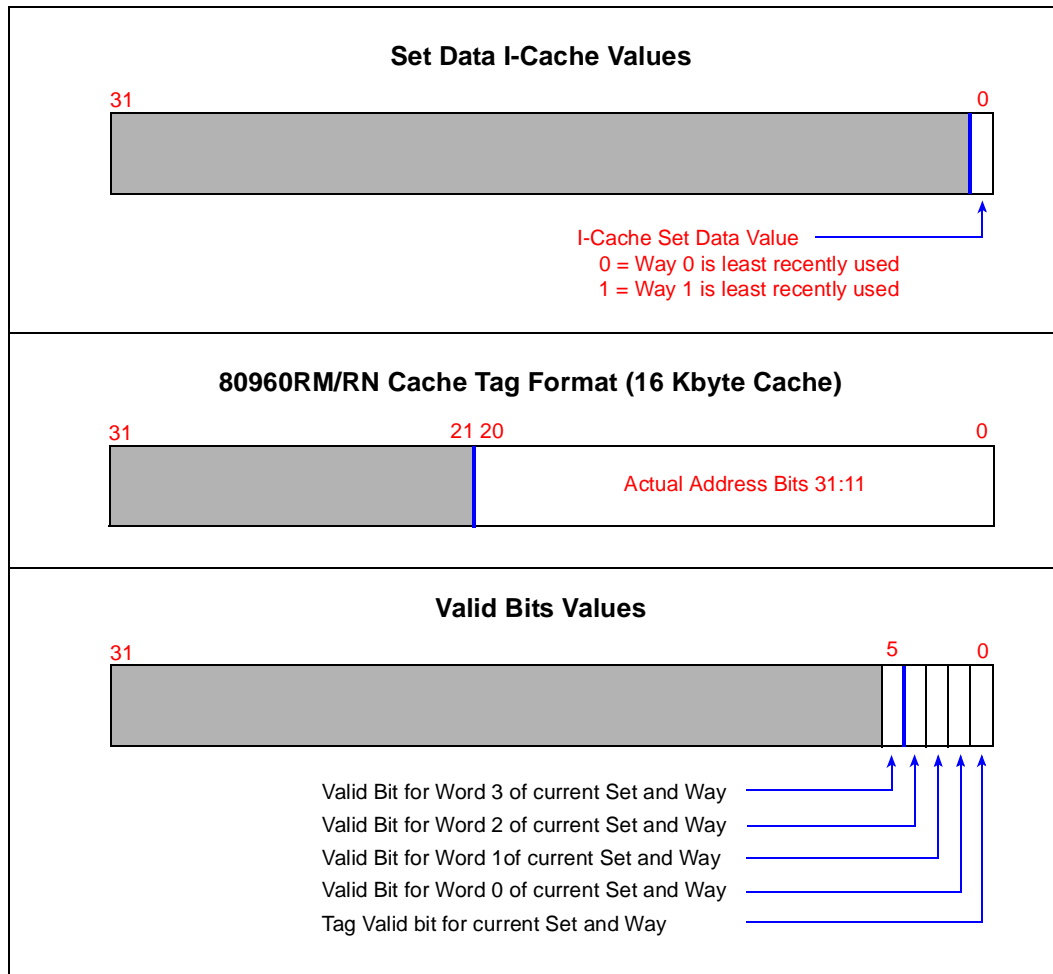


**Table 6-9. icctl Status Values and I-Cache Parameters**

Value	Value on i960RP 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] (log2(bytes per atom))	2
Status[11:8] (log2(atoms per line))	2
Status[15:12] (log2(number of sets))	9
Status[27:16] (number of ways - 1)	1
Lock Status[7:0] (number of blocks that lock)	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 >= Icache_max_sets)
    end = Icache_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] =
              Icache_line[set][way][word];
              memadr += 4;
              }
         } } break;

default:    # Reserved.
            generate_fault(OPERATION.INVALID_OPERAND);
            break;}

```

**Faults:** STANDARD TYPE.MISMATCH Refer to [Section 6.1.6, “Faults” on page 6-4](#). Attempt to execute instruction while not in supervisor mode.

**Example:**

```

icctl g0,g1,g2      # g0 = 3, g1=0x10000000, g2=1
                    # 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 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

**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.)

<i>src1</i> 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)
    dst = 1;
else
    dst = 0;
    
```

**Faults:** STANDARD Refer to [Section 6.1.6, “Faults”](#) on page 6-4.  
TYPE.MISMATCH Attempt to execute instruction while not in supervisor mode.

**Example:**

```

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

**Opcode:** **intctl** 658H REG

**See Also:** **intdis, inten**

**Notes:** This instruction is implemented on the 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**

<b>Mnemonic:</b>	<b>intdis</b>	Global Interrupt Disable
<b>Format:</b>	<b>intdis</b>	
<b>Description:</b>	Globally disables interrupts and ensures that the change takes effect before the instruction completes. This operation is implemented by setting ICON.gie to one.	
<b>Action:</b>	<pre>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);</pre>	
<b>Faults:</b>	STANDARD TYPE.MISMATCH	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4. Attempt to execute instruction while not in supervisor mode.
<b>Example:</b>	<pre>intdis</pre>	<pre># ICON.gie = 0, interrupts enabled # Disable interrupts. # ICON.gie = 1</pre>
<b>Opcode:</b>	<b>intdis</b> 5B4H	REG
<b>See Also:</b>	<b>intctl, inten</b>	
<b>Notes:</b>	This instruction is implemented on the 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**

<b>Mnemonic:</b>	<b>inten</b>	global interrupt enable
<b>Format:</b>	<b>inten</b>	
<b>Description:</b>	Globally enables interrupts and ensures that the change takes effect before the instruction completes. This operation is implemented by clearing ICON.gie to zero.	
<b>Action:</b>	<pre> 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);                     </pre>	
<b>Faults:</b>	STANDARD TYPE.MISMATCH	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-4</a> . Attempt to execute instruction while not in supervisor mode.
<b>Example:</b>	<pre> inten                # ICON.gie = 1, interrupts disabled.                     # Enable interrupts.                     # ICON.gie = 0                     </pre>	
<b>Opcode:</b>	<b>inten</b>	5B5H                      REG
<b>See Also:</b>	<b>intctl, intdis</b>	
<b>Notes:</b>	This instruction is implemented on the 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

<b>Mnemonic:</b>	<b>ld</b>	Load
	<b>ldob</b>	Load Ordinal Byte
	<b>ldos</b>	Load Ordinal Short
	<b>ldib</b>	Load Integer Byte
	<b>ldis</b>	Load Integer Short
	<b>ldl</b>	Load Long
	<b>ldt</b>	Load Triple
	<b>ldq</b>	Load Quad

<b>Format:</b>	<b>ld*</b>	<i>src</i> ,	<i>dst</i>
		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] != 00<sub>2</sub>) && 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] = 0xFFFFFFFF;

**ldos:**

```

dst = read_memory(effective_address)[15:0];
                                     # Order depends on endianness.
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 endianness.
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_address)[31:0];
    dst+_1 = read_memory(effective_address+_4)[31:0];
    dst+_2 = read_memory(effective_address+_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_address)[31:0];
                                     # Order depends on endianness.
    dst+_1 = read_memory(effective_address+_4)[31:0];
    dst+_2 = read_memory(effective_address+_8)[31:0];
    dst+_3 = read_memory(effective_address+_12)[31:0];
    if((effective_address[3:0] != 00002) && unaligned_fault_enabled)

```



## 6.2.38 **Ida**

<b>Mnemonic:</b>	<b>Ida</b>	Load Address	
<b>Format:</b>	<b>Ida</b>	<i>src</i> , mem efa	<i>dst</i> reg
<b>Description:</b>	<p>Computes the effective address specified with <i>src</i> and stores it in <i>dst</i>. The <i>src</i> address is not checked for validity. Any addressing mode may be used to calculate <i>efa</i>.</p> <p>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, <b>mov</b> can be used with a literal as the <i>src</i> operand.)</p>		
<b>Action:</b>	dst = effective_address;		
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.	
<b>Example:</b>	lda 58 (g9), g1	# g1 = g9+58	
	lda 0x749, r8	# r8 = 0x749	
<b>Opcode:</b>	<b>Ida</b>	8CH	MEM

## 6.2.39 **mark**

<b>Mnemonic:</b>	<b>mark</b>	Mark
<b>Format:</b>	<b>mark</b>	
<b>Description:</b>	<p>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.</p> <p>If mark trace mode is not enabled, <b>mark</b> behaves like a no-op.</p> <p>For more information on trace fault generation, refer to <a href="#">Chapter 10, “Tracing and Debugging”</a>.</p>	
<b>Action:</b>	<pre>if(PC.te &amp;&amp; TC.mk)     generate_fault(TRACE.MARK)</pre>	
<b>Faults:</b>	STANDARD TRACE.MARK	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4. Trace fault is generated if PC.te=1 and TC.mk=1.
<b>Example:</b>	<pre># 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.</pre>	
<b>Opcode:</b>	<b>mark</b>	66BH REG
<b>See Also:</b>	<b>fmark, modpc, modtc</b>	

## 6.2.40 modac

<b>Mnemonic:</b>	<b>modac</b>	Modify AC		
<b>Format:</b>	<b>modac</b>	<i>mask</i> ,	<i>src</i> ,	<i>dst</i>
		reg/lit	reg/lit	reg
<b>Description:</b>	Reads and modifies the AC register. <i>src</i> contains the value to be placed in the AC register; <i>mask</i> specifies bits that may be changed. Only bits set in <i>mask</i> are modified. Once the AC register is changed, its initial state is copied into <i>dst</i> .			
<b>Action:</b>	temp = AC; AC = (src & mask)   (AC & ~mask); dst = temp;			
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.		
<b>Example:</b>	modac g1, g9, g12 # AC = g9, masked by g1. # g12 = initial value of AC.			
<b>Opcode:</b>	<b>modac</b>	645H	REG	
<b>See Also:</b>	<b>modpc, modtc</b>			
<b>Side Effects:</b>	Sets the condition code in the arithmetic controls.			

## 6.2.41 **modi**

<b>Mnemonic:</b>	<b>modi</b>	Modulo Integer
<b>Format:</b>	<b>modi</b>	<i>src1</i> , <i>src2</i> , <i>dst</i> reg/lit reg/lit reg
<b>Description:</b>	Divides <i>src2</i> by <i>src1</i> , where both are integers and stores the modulo remainder of the result in <i>dst</i> . If the result is nonzero, <i>dst</i> has the same sign as <i>src1</i> .	
<b>Action:</b>	<pre> if(src1 == 0) {     dst = undefined_value;     generate_fault(ARITHMETIC.ZERO_DIVIDE); } dst = src2 - (src2/src1) * src1; if((src2 *src1 &lt; 0) &amp;&amp; (dst != 0))     dst = dst + src1; </pre>	
<b>Faults:</b>	STANDARD ARITHMETIC.ZERO_DIVIDE	See <a href="#">Section 6.1.6, “Faults” on page 6-4</a> . The <i>src1</i> operand is zero.
<b>Example:</b>	modi r9, r2, r5 # r5 = modulo (r2/r9)	
<b>Opcode:</b>	<b>modi</b>	749H REG
<b>See Also:</b>	<b>divi, divo, remi, remo</b>	
<b>Notes:</b>	<b>modi</b> generates the correct result (0) when computing $-2^{31} \bmod -1$ , although the corresponding 32-bit division does overflow, it does not generate a fault.	



## 6.2.42 modify

<b>Mnemonic:</b>	<b>modify</b>	Modify		
<b>Format:</b>	<b>modify</b>	<i>mask</i> ,	<i>src</i> ,	<i>src/dst</i>
		reg/lit	reg/lit	reg
<b>Description:</b>	Modifies selected bits in <i>src/dst</i> with bits from <i>src</i> . The <i>mask</i> operand selects the bits to be modified: only bits set in the <i>mask</i> operand are modified in <i>src/dst</i> .			
<b>Action:</b>	$src\_dst = (src \& mask)   (src\_dst \& \sim mask);$			
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.		
<b>Example:</b>	modify g8, g10, r4 # r4 = g10 masked by g8.			
<b>Opcode:</b>	<b>modify</b>	650H	REG	
<b>See Also:</b>	<b>alterbit, extract</b>			

## 6.2.43 **modpc**

<b>Mnemonic:</b>	<b>modpc</b>	Modify Process Controls
<b>Format:</b>	<b>modpc</b>	<i>src</i> , <i>mask</i> , <i>src/dst</i> reg/lit reg/lit reg
<b>Description:</b>	<p>Reads and modifies the PC register as specified with <i>mask</i> and <i>src/dst</i>. <i>src/dst</i> operand contains the value to be placed in the PC register; <i>mask</i> operand specifies bits that may be changed. Only bits set in the <i>mask</i> are modified. Once the PC register is changed, its initial value is copied into <i>src/dst</i>. The <i>src</i> operand is a dummy operand that should specify a literal or the same register as the <i>mask</i> operand.</p> <p>The processor must be in supervisor mode to use this instruction with a non-zero <i>mask</i> value. If <i>mask</i>=0, this instruction can be used to read the process controls, without the processor being in supervisor mode.</p> <p>If the action of this instruction lowers the processor priority, the processor checks the interrupt table for pending interrupts.</p> <p>When process controls are changed, the processor recognizes the changes immediately except in one situation: if <b>modpc</b> 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 <a href="#">Section 3.6.3, “Process Controls Register – PC”</a> on page 3-16.</p>	
<b>Action:</b>	<pre>if(mask != 0) {   if(PC.em != supervisor)     generate_fault(TYPE.MISMATCH);   temp = PC;   PC = (mask &amp; src_dst)   (PC &amp; ~mask);   src_dst = temp;   if(temp.priority &gt; PC.priority)     check_pending_interrupts; } else   src_dst = PC;</pre>	
<b>Faults:</b>	STANDARD TYPE.MISMATCH	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.
<b>Example:</b>	<pre>modpc g9, g9, g8    # process controls = g8                    # masked by g9.</pre>	
<b>Opcode:</b>	<b>modpc</b>	655H REG
<b>See Also:</b>	<b>modac, modtc</b>	
<b>Notes:</b>	<p>Since <b>modpc</b> 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.</p>	

## 6.2.44 modtc

<b>Mnemonic:</b>	<b>modtc</b>	Modify Trace Controls		
<b>Format:</b>	<b>modtc</b>	<i>mask</i> ,	<i>src2</i> ,	<i>dst</i>
		reg/lit	reg/lit	reg
<b>Description:</b>	<p>Reads and modifies TC register as specified with <i>mask</i> and <i>src2</i>. The <i>src2</i> operand contains the value to be placed in the TC register; <i>mask</i> operand specifies bits that may be changed. Only bits set in <i>mask</i> are modified. <i>mask</i> must not enable modification of reserved bits. Once the TC register is changed, its initial state is copied into <i>dst</i>.</p> <p>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.</p> <p>For more information on the trace controls, refer to <a href="#">Chapter 9, “Faults”</a> and <a href="#">Chapter 10, “Tracing and Debugging”</a>.</p>			
<b>Action:</b>	<pre>mode_bits = 0x000000FE; event_flags = 0X0F000000 temp = TC; tempa = (event_flags &amp; TC &amp; mask)   (mode_bits &amp; mask); TC = (tempa &amp; src2)   (TC &amp; ~tempa); dst = temp;</pre>			
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.		
<b>Example:</b>	<pre>modtc g12, g10, g2 # trace controls = g10 masked                   # by g12; previous trace                   # controls stored in g2.</pre>			
<b>Opcode:</b>	<b>modtc</b>	654H	REG	
<b>See Also:</b>	<b>modac, modpc</b>			

## 6.2.45 MOVE

<b>Mnemonic:</b>	<b>mov</b>	Move
	<b>movl</b>	Move Long
	<b>movt</b>	Move Triple
	<b>movq</b>	Move Quad

<b>Format:</b>	<b>mov*</b>	<i>src1</i> ,	<i>dst</i>
		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;
    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;
}
}

```

<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.
<b>Example:</b>	<code>movt g8, r4</code>	# r4, r5, r6 = g8, g9, g10
<b>Opcode:</b>	<b>mov</b> 5CCH	REG
	<b>movl</b> 5DCH	REG
	<b>movt</b> 5ECH	REG
	<b>movq</b> 5FCH	REG
<b>See Also:</b>	<b>LOAD, STORE, lda</b>	

## 6.2.46 muli, mulo

<b>Mnemonic:</b>	<b>muli</b>	Multiply Integer	
	<b>mulo</b>	Multiply Ordinal	
<b>Format:</b>	<b>mul*</b>	<i>src1</i> , reg/lit	<i>src2</i> , reg/lit <i>dst</i> reg
<b>Description:</b>	Multiplies the <i>src2</i> value by the <i>src1</i> value and stores the result in <i>dst</i> . The binary results from these two instructions are identical. The only difference is that <i>muli</i> can signal an integer overflow.		
<b>Action:</b>	<p><b>mulo:</b></p> <pre>dst = (src2 * src1)[31:0];</pre> <p><b>muli:</b></p> <pre>true_result = (src1 * src2); dst = true_result[31:0]; if((true_result &gt; (2**31) - 1)    (true_result &lt; -2**31))# Check for overflow {     if(AC.om == 1)         AC.of = 1;     else         generate_fault(ARITHMETIC.OVERFLOW); }</pre>		
<b>Faults:</b>	STANDARD ARITHMETIC.OVERFLOW	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-4</a> . Result is too large for destination register ( <b>muli</b> only). If a condition of overflow occurs, the least significant 32 bits of the result are stored in the destination register.	
<b>Example:</b>	<code>muli r3, r4, r9      # r9 = r4 * r3</code>		
<b>Opcode:</b>	<b>muli</b>	741H	REG
	<b>mulo</b>	701H	REG
<b>See Also:</b>	<b>emul, ediv, divi, divo</b>		

## 6.2.47 nand

<b>Mnemonic:</b>	<b>nand</b>	Nand		
<b>Format:</b>	<b>nand</b>	<i>src1</i> , reg/lit	<i>src2</i> , reg/lit	<i>dst</i> reg
<b>Description:</b>	Performs a bitwise NAND operation on <i>src2</i> and <i>src1</i> values and stores the result in <i>dst</i> .			
<b>Action:</b>	$dst = \sim src2   \sim src1;$			
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.		
<b>Example:</b>	nand g5, r3, r7 # r7 = r3 NAND g5			
<b>Opcode:</b>	<b>nand</b>	58EH	REG	
<b>See Also:</b>	<b>and, andnot, nor, not, notand, notor, or, ornot, xnor, xor</b>			

**6.2.48**    **nor**

<b>Mnemonic:</b>	<b>nor</b>	Nor		
<b>Format:</b>	<b>nor</b>	<i>src1</i> , reg/lit	<i>src2</i> , reg/lit	<i>dst</i> reg
<b>Description:</b>	Performs a bitwise NOR operation on the <i>src2</i> and <i>src1</i> values and stores the result in <i>dst</i> .			
<b>Action:</b>	dst = ~src2 & ~src1;			
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.		
<b>Example:</b>	nor g8, 28, r5            # r5 = 28 NOR g8			
<b>Opcode:</b>	<b>nor</b>	588H	REG	
<b>See Also:</b>	<b>and, andnot, nand, not, notand, notor, or, ornot, xnor, xor</b>			



## 6.2.49 not, notand

<b>Mnemonic:</b>	<b>not</b>	Not		
	<b>notand</b>	Not And		
<b>Format:</b>	<b>not</b>	<i>src1</i> , reg/lit	<i>dst</i> reg	
	<b>notand</b>	<i>src1</i> , reg/lit	<i>src2</i> , reg/lit	<i>dst</i> reg
<b>Description:</b>	Performs a bitwise NOT ( <b>not</b> instruction) or NOT AND ( <b>notand</b> instruction) operation on the <i>src2</i> and <i>src1</i> values and stores the result in <i>dst</i> .			
<b>Action:</b>	<b>not:</b> dst = ~src1;			
	<b>notand:</b> dst = ~src2 & src1;			
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.		
<b>Example:</b>	not g2, g4 # g4 = NOT g2 notand r5, r6, r7 # r7 = NOT r6 AND r5			
<b>Opcode:</b>	<b>not</b>	58AH	REG	
	<b>notand</b>	584H	REG	
<b>See Also:</b>	<b>and, andnot, nand, nor, notor, or, ornot, xnor, xor</b>			

**6.2.50 notbit**

<b>Mnemonic:</b>	<b>notbit</b>	Not Bit		
<b>Format:</b>	<b>notbit</b>	<i>bitpos</i> , reg/lit	<i>src2</i> , reg/lit	<i>dst</i> reg
<b>Description:</b>	Copies the <i>src2</i> value to <i>dst</i> with one bit toggled. The <i>bitpos</i> operand specifies the bit to be toggled.			
<b>Action:</b>	$dst = src2 \wedge 2^{*(src1 \% 32)}$ ;			
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.		
<b>Example:</b>	notbit r3, r12, r7 # r7 = r12 with the bit # specified in r3 toggled.			
<b>Opcode:</b>	<b>notbit</b>	580H	REG	
<b>See Also:</b>	<b>alterbit, chkbit, clrbit, setbit</b>			

## 6.2.51 notor

<b>Mnemonic:</b>	<b>notor</b>	Not Or		
<b>Format:</b>	<b>notor</b>	<i>src1</i> , reg/lit	<i>src2</i> , reg/lit	<i>dst</i> reg
<b>Description:</b>	Performs a bitwise NOTOR operation on <i>src2</i> and <i>src1</i> values and stores result in <i>dst</i> .			
<b>Action:</b>	$dst = \sim src2   src1;$			
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.		
<b>Example:</b>	<code>notor g12, g3, g6 # g6 = NOT g3 OR g12</code>			
<b>Opcode:</b>	<b>notor</b>	58DH	REG	
<b>See Also:</b>	<b>and, andnot, nand, nor, not, notand, or, ornot, xnor, xor</b>			

## 6.2.52 or, ornot

<b>Mnemonic:</b>	<b>or</b>	Or		
	<b>ornot</b>	Or Not		
<b>Format:</b>	<b>or</b>	<i>src1</i> , reg/lit	<i>src2</i> , reg/lit	<i>dst</i> reg
	<b>ornot</b>	<i>src1</i> , reg/lit	<i>src2</i> , reg/lit	<i>dst</i> reg
<b>Description:</b>	Performs a bitwise OR ( <b>or</b> instruction) or ORNOT ( <b>ornot</b> instruction) operation on the <i>src2</i> and <i>src1</i> values and stores the result in <i>dst</i> .			
<b>Action:</b>	<p><b>or:</b> dst = src2   src1;</p> <p><b>ornot:</b> dst = src2   ~src1;</p>			
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.		
<b>Example:</b>	<pre>or 14, g9, g3      # g3 = g9 OR 14 ornot r3, r8, r11 # r11 = r8 OR NOT r3</pre>			
<b>Opcode:</b>	<b>or</b>	587H	REG	
	<b>ornot</b>	58BH	REG	
<b>See Also:</b>	<b>and, andnot, nand, nor, not, notand, notor, xnor, xor</b>			

## 6.2.53 remi, remo

<b>Mnemonic:</b>	<b>remi</b>	Remainder Integer		
	<b>remo</b>	Remainder Ordinal		
<b>Format:</b>	<b>rem*</b>	<i>src1</i> , reg/lit	<i>src2</i> , reg/lit	<i>dst</i> reg
<b>Description:</b>	Divides <i>src2</i> by <i>src1</i> and stores the remainder in <i>dst</i> . The sign of the result (if nonzero) is the same as the sign of <i>src2</i> .			
<b>Action:</b>	<b>remi, remo:</b> if( <i>src1</i> == 0) generate_fault(ARITHMETIC.ZERO_DIVIDE); <i>dst</i> = <i>src2</i> - ( <i>src2</i> / <i>src1</i> )* <i>src1</i> ;			
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-4</a> .		
	ARITHMETIC.ZERO_DIVIDE	The <i>src1</i> operand is 0.		
<b>Example:</b>	remo r4, r5, r6      # r6 = r5 rem r4			
<b>Opcode:</b>	<b>remi</b>	748H	REG	
	<b>remo</b>	708H	REG	
<b>See Also:</b>	<b>modi</b>			
<b>Notes:</b>	<b>remi</b> produces the correct result (0) even when computing $-2^{31}$ <b>remi</b> -1, which would cause the corresponding division to overflow, although no fault is generated.			

## 6.2.54 **ret**

**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_synchf();
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();
        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;
}

```

<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.
<b>Example:</b>	ret	# Program control returns to # context of calling procedure.
<b>Opcode:</b>	ret          0AH	CTRL
<b>See Also:</b>	call, calls, callx	

## 6.2.55 rotate

<b>Mnemonic:</b>	<b>rotate</b>	Rotate		
<b>Format:</b>	<b>rotate</b>	<i>len</i> ,	<i>src2</i> ,	<i>dst</i>
		reg/lit	reg/lit	reg
<b>Description:</b>	Copies <i>src2</i> to <i>dst</i> and rotates the bits in the resulting <i>dst</i> operand to the left (toward higher significance). Bits shifted off left end of word are inserted at right end of word. The <i>len</i> operand specifies number of bits that the <i>dst</i> 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 <i>len</i> operand.			
<b>Action:</b>	<i>src2</i> is rotated by <i>len</i> mod 32. This value is stored in <i>dst</i> .			
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.		
<b>Example:</b>	<pre>rotate 13, r8, r12 # r12 = r8 with bits rotated                   # 13 bits to left.</pre>			
<b>Opcode:</b>	<b>rotate</b>	59DH	REG	
<b>See Also:</b>	<b>SHIFT, eshro</b>			



## 6.2.56 scanbit

<b>Mnemonic:</b>	<b>scanbit</b>	Scan For Bit
<b>Format:</b>	<b>scanbit</b>	<i>src1</i> , <i>dst</i> reg/lit reg
<b>Description:</b>	Searches <i>src1</i> 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 <i>dst</i> and the condition code is set to 010 <sub>2</sub> . If <i>src</i> value is zero, all 1's are stored in <i>dst</i> and condition code is set to 000 <sub>2</sub> .	
<b>Action:</b>	<pre>dst = 0xFFFFFFFF; AC.cc = 000<sub>2</sub>; for(i = 31; i &gt;= 0; i--) {   if((src1 &amp; 2**i) != 0)     {   dst = i;         AC.cc = 010<sub>2</sub>;         break;     } }</pre>	
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, "Faults" on page 6-4</a> .
<b>Example:</b>	<pre># assume g8 is nonzero scanbit g8, g10    # g10 = bit number of most-                   # significant set bit in g8;                   # AC.cc = 010<sub>2</sub>.</pre>	
<b>Opcode:</b>	<b>scanbit</b>	641H REG
<b>See Also:</b>	<b>spanbit, setbit</b>	
<b>Side Effects:</b>	Sets the condition code in the arithmetic controls.	

## 6.2.57 scanbyte

<b>Mnemonic:</b>	<b>scanbyte</b>	Scan Byte Equal
<b>Format:</b>	<b>scanbyte</b>	<i>src1</i> , <i>src2</i> reg/lit reg/lit
<b>Description:</b>	Performs byte-by-byte comparison of <i>src1</i> and <i>src2</i> and sets condition code to 010 <sub>2</sub> if any two corresponding bytes are equal. If no corresponding bytes are equal, condition code is set to 000 <sub>2</sub> .	
<b>Action:</b>	<pre> if((src1 &amp; 0x000000FF) == (src2 &amp; 0x000000FF)        (src1 &amp; 0x0000FF00) == (src2 &amp; 0x0000FF00)        (src1 &amp; 0x00FF0000) == (src2 &amp; 0x00FF0000)        (src1 &amp; 0xFF000000) == (src2 &amp; 0xFF000000))     AC.cc = 010<sub>2</sub>; else     AC.cc = 000<sub>2</sub>; </pre>	
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.
<b>Example:</b>	<pre> # Assume r9 = 0x11AB1100 scanbyte 0x00AB0011, r9# AC.cc = 010<sub>2</sub> </pre>	
<b>Opcode:</b>	<b>scanbyte</b>	5ACH REG
<b>See Also:</b>	<b>bswap</b>	
<b>Side Effects:</b>	Sets the condition code in the arithmetic controls.	

## 6.2.58 SEL<cc>

<b>Mnemonic:</b>	<b>selno</b>	Select Based on Unordered
	<b>selg</b>	Select Based on Greater
	<b>sele</b>	Select Based on Equal
	<b>selge</b>	Select Based on Greater or Equal
	<b>sell</b>	Select Based on Less
	<b>selne</b>	Select Based on Not Equal
	<b>selle</b>	Select Based on Less or Equal
	<b>selo</b>	Select Based on Ordered

<b>Format:</b>	<b>sel*</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>
		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
<b>selno</b>	000 <sub>2</sub>	Unordered
<b>selg</b>	001 <sub>2</sub>	Greater
<b>sele</b>	010 <sub>2</sub>	Equal
<b>selge</b>	011 <sub>2</sub>	Greater or equal
<b>sell</b>	100 <sub>2</sub>	Less
<b>selne</b>	101 <sub>2</sub>	Not equal
<b>selle</b>	110 <sub>2</sub>	Less or equal
<b>selo</b>	111 <sub>2</sub>	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 6-4](#).

**Example:**

```
sele g0,g1,g2    # AC.cc = 0102
                 # g2 = g1

sell g0,g1,g2   # AC.cc = 0012
                 # g2 = g0
```

<b>Opcode:</b>	<b>selno</b>	784H	REG
	<b>selg</b>	794H	REG
	<b>sele</b>	7A4H	REG
	<b>selge</b>	7B4H	REG
	<b>sell</b>	7C4H	REG
	<b>selne</b>	7D4H	REG
	<b>selle</b>	7E4H	REG
	<b>selo</b>	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**

<b>Mnemonic:</b>	<b>setbit</b>	Set Bit		
<b>Format:</b>	<b>setbit</b>	<i>bitpos</i> , reg/lit	<i>src</i> , reg/lit	<i>dst</i> reg
<b>Description:</b>	Copies <i>src</i> value to <i>dst</i> with one bit set. <i>bitpos</i> specifies bit to be set.			
<b>Action:</b>	$dst = src   (2^{*(bitpos \% 32)})$ ;			
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-4.</a>		
<b>Example:</b>	setbit 15, r9, r1 # r1 = r9 with bit 15 set.			
<b>Opcode:</b>	<b>setbit</b>	583H	REG	
<b>See Also:</b>	<b>alterbit, chkbit, clrbit, notbit</b>			

## 6.2.60 SHIFT

<b>Mnemonic:</b>	<b>shl</b>	Shift Left Ordinal
	<b>shro</b>	Shift Right Ordinal
	<b>shli</b>	Shift Left Integer
	<b>shri</b>	Shift Right Integer
	<b>shrdi</b>	Shift Right Dividing Integer

<b>Format:</b>	<b>sh*</b>	<i>len</i> ,	<i>src</i> ,	<i>dst</i>
		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];
    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 6-4.](#)  
 ARITHMETIC.OVERFLOW For **shli**.

**Example:** shli 13, g4, r6 # g6 = g4 shifted left 13 bits.

<b>Opcode:</b>	<b>shlo</b>	59CH	REG
	<b>shro</b>	598H	REG
	<b>shli</b>	59EH	REG
	<b>shri</b>	59BH	REG
	<b>shrdi</b>	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

<b>Mnemonic:</b>	<b>spanbit</b>	Span Over Bit
<b>Format:</b>	<b>spanbit</b>	<i>src</i> , <i>dst</i> reg/lit reg
<b>Description:</b>	Searches <i>src</i> value for the most significant clear bit (0 bit). If a most significant 0 bit is found, its bit number is stored in <i>dst</i> and condition code is set to 010 <sub>2</sub> . If <i>src</i> value is all 1's, all 1's are stored in <i>dst</i> and condition code is set to 000 <sub>2</sub> .	
<b>Action:</b>	<pre>dst = 0xFFFFFFFF; AC.cc = 000<sub>2</sub>; for(i = 31; i &gt;= 0; i--) {   if((src1 &amp; 2**i) == 0))     {   dst = i;         AC.cc = 010<sub>2</sub>;         break;     } }</pre>	
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, "Faults"</a> on page 6-4.
<b>Example:</b>	<pre># Assume r2 is not 0xffffffff spanbit r2, r9      # r9 = bit number of most-                    # significant clear bit in r2;                    # AC.cc = 010<sub>2</sub></pre>	
<b>Opcode:</b>	<b>spanbit</b>	640H REG
<b>See Also:</b>	<b>scanbit</b>	
<b>Side Effects:</b>	Sets the condition code in the arithmetic controls.	

## 6.2.62 STORE

<b>Mnemonic:</b>	<b>st</b>	Store
	<b>stob</b>	Store Ordinal Byte
	<b>stos</b>	Store Ordinal Short
	<b>stib</b>	Store Integer Byte
	<b>stis</b>	Store Integer Short
	<b>stl</b>	Store Long
	<b>stt</b>	Store Triple
	<b>stq</b>	Store Quad

<b>Format:</b>	<b>st*</b>	<i>src1</i> ,	<i>dst</i>
		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;

```

**Action:**

```

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];

```

```

stib:
if (illegal_write_to_on_chip_RAM_or_MMR)
    generate_fault(TYPE.MISMATCH);
else if ((src1[31:8] != 0) && (src1[31:8] != 0xFFFFFFFF))
    { 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];

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] != 00002) && 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] != 00002) && 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);
    }
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 ARITHMETIC.OVERFLOW Refer to [Section 6.1.6, “Faults” on page 6-4.](#)  
For **stib**, **stis**.

**Example:** `st g2, 1254 (g6)` # Word beginning at offset  
# 1254 + (g6) = g2.

<b>Opcode:</b>	<b>st</b>	92H	MEM
	<b>stob</b>	82H	MEM
	<b>stos</b>	8AH	MEM
	<b>stib</b>	C2H	MEM
	<b>stis</b>	CAH	MEM
	<b>stl</b>	9AH	MEM
	<b>stt</b>	A2H	MEM
	<b>stq</b>	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 endianness of the memory region and is implementation-dependent.

## 6.2.63 **subc**

<b>Mnemonic:</b>	<b>subc</b>	Subtract Ordinal With Carry		
<b>Format:</b>	<b>subc</b>	<i>src1</i> , reg/lit	<i>src2</i> , reg/lit	<i>dst</i> reg
<b>Description:</b>	<p>Subtracts <i>src1</i> from <i>src2</i>, then subtracts the opposite of condition code bit 1 (used here as the carry bit) and stores the result in <i>dst</i>. If the ordinal subtraction results in a carry, condition code bit 1 is set to 1, otherwise it is set to 0.</p> <p>This instruction can also be used for integer subtraction. Here, if integer subtraction results in an overflow, condition code bit 0 is set.</p> <p><b>subc</b> does not distinguish between ordinals and integers: it sets condition code bits 0 and 1 regardless of data type.</p>			
<b>Action:</b>	<pre>dst = (src2 - src1 -1 + AC.cc[1])[31:0]; AC.cc[2:0] = 000<sub>2</sub>; if((src2[31] == src1[31]) &amp;&amp; (src2[31] != dst[31]))     AC.cc[0] = 1;          # Overflow bit. AC.cc[1] = (src2 - src1 -1 + AC.cc[1])[32];    # Carry out.</pre>			
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.		
<b>Example:</b>	<pre>subc g5, g6, g7 # g7 = g6 - g5 - not(condition code bit 1)</pre>			
<b>Opcode:</b>	<b>subc</b>	5B2H	REG	
<b>See Also:</b>	<b>addc, addi, addo, subi, subo</b>			
<b>Side Effects:</b>	Sets the condition code in the arithmetic controls.			

## 6.2.64 SUB<cc>

<b>Mnemonic:</b>	<b>subono</b>	Subtract Ordinal if Unordered
	<b>subog</b>	Subtract Ordinal if Greater
	<b>suboe</b>	Subtract Ordinal if Equal
	<b>suboge</b>	Subtract Ordinal if Greater or Equal
	<b>subol</b>	Subtract Ordinal if Less
	<b>subone</b>	Subtract Ordinal if Not Equal
	<b>subole</b>	Subtract Ordinal if Less or Equal
	<b>suboo</b>	Subtract Ordinal if Ordered
	<b>subino</b>	Subtract Integer if Unordered
	<b>subig</b>	Subtract Integer if Greater
	<b>subie</b>	Subtract Integer if Equal
	<b>subige</b>	Subtract Integer if Greater or Equal
	<b>subil</b>	Subtract Integer if Less
	<b>subine</b>	Subtract Integer if Not Equal
	<b>subile</b>	Subtract Integer if Less or Equal
	<b>subio</b>	Subtract Integer if Ordered

<b>Format:</b>	<b>sub*</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>
		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
<b>subono, subino</b>	000 <sub>2</sub>	Unordered
<b>subog, subig</b>	001 <sub>2</sub>	Greater
<b>suboe, subie</b>	010 <sub>2</sub>	Equal
<b>suboge, subige</b>	011 <sub>2</sub>	Greater or equal
<b>subol, subil</b>	100 <sub>2</sub>	Less
<b>subone, subine</b>	101 <sub>2</sub>	Not equal
<b>subole, subile</b>	110 <sub>2</sub>	Less or equal
<b>suboo, subio</b>	111 <sub>2</sub>	Ordered



## 6.2.65 **subi, subo**

<b>Mnemonic:</b>	<b>subi</b>	Subtract Integer	
	<b>subo</b>	Subtract Ordinal	
<b>Format:</b>	<b>sub*</b>	<i>src1</i> , reg/lit	<i>src2</i> , reg/lit <i>dst</i> reg
<b>Description:</b>	Subtracts <i>src1</i> from <i>src2</i> and stores the result in <i>dst</i> . The binary results from these two instructions are identical. The only difference is that <b>subi</b> can signal an integer overflow.		
<b>Action:</b>	<b>subo:</b>	dst = (src2 - src1)[31:0];	
	<b>subi:</b>	<pre> true_result = (src2 - src1); dst = true_result[31:0]; if((true_result &gt; (2**31) - 1)    (true_result &lt; -2**31))# Check for overflow {     if(AC.om == 1)         AC.of = 1;     else         generate_fault(ARITHMETIC.OVERFLOW); } </pre>	
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-4.</a>	
	ARITHMETIC.OVERFLOW	For <b>subi</b> .	
<b>Example:</b>	subi g6, g9, g12    # g12 = g9 - g6		
<b>Opcode:</b>	<b>subi</b>	593H	REG
	<b>subo</b>	592H	REG
<b>See Also:</b>	<b>addi, addo, subc, addc</b>		

## 6.2.66 syncf

<b>Mnemonic:</b>	<b>syncf</b>	Synchronize Faults
<b>Format:</b>	<b>syncf</b>	
<b>Description:</b>	Waits for all faults to be generated that are associated with any prior uncompleted instructions.	
<b>Action:</b>	<pre>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.</pre>	
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.
<b>Example:</b>	<pre>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.</pre>	
<b>Opcode:</b>	<b>syncf</b>	66FH                    REG
<b>See Also:</b>	<b>mark, fmark</b>	

## 6.2.67 sysctl

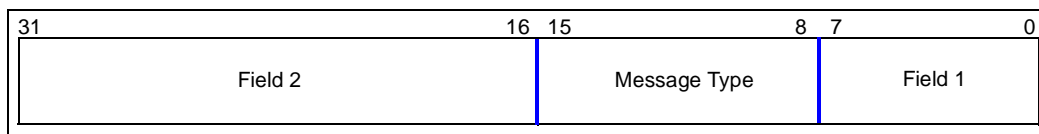
**Mnemonic:** **sysctl** System Control

**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 6-10. sysctl Field Definitions**

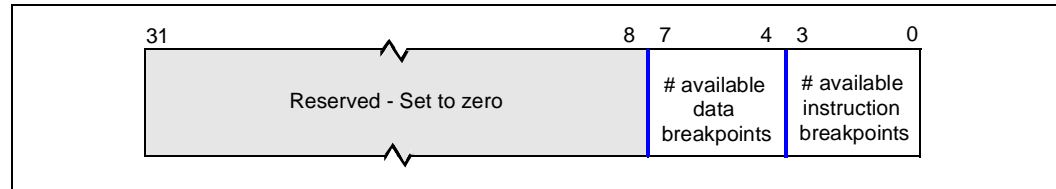
Message	Src1			Src2	Src/Dst
	Type	Field 1	Field 2	Field 3	Field 4
Request Interrupt	0x0	Vector Number	N/U	N/U	N/U
Invalidate Cache	0x1	N/U	N/U	N/U	N/U
Configure Instruction Cache	0x2	Cache Mode Configuration (Table 6-11)	N/U	Cache load address	N/U
Reinitialize	0x3	N/U	N/U	Starting IP	PRCB Pointer
Modify 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 6-11. Cache Mode Configuration**

Mode Field	Mode Description	80960RM/RN
000 <sub>2</sub>	Normal cache enabled	16 Kbyte
XX1 <sub>2</sub>	Full cache disabled	16 Kbyte
100 <sub>2</sub> or 110 <sub>2</sub>	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 80960RM/RN.
# src2 has starting address of code to lock.
# src2 is aligned to a quad word
# boundary.
aligned_addr = src2 & 0xfffff0;
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 6-4](#).

**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:** **dcctl, icctl**

**Notes:** This instruction is implemented on 80960RM/RN, 80960RP/RD, 80960Hx, 80960Jx and 80960Cx processors, and may or may not be implemented on future i960 processors.

## 6.2.68 TEST<cc>

<b>Mnemonic:</b>	<b>teste</b>	Test For Equal
	<b>testne</b>	Test For Not Equal
	<b>testl</b>	Test For Less
	<b>testle</b>	Test For Less Or Equal
	<b>testg</b>	Test For Greater
	<b>testge</b>	Test For Greater Or Equal
	<b>testo</b>	Test For Ordered
	<b>testno</b>	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<sub>2</sub>, 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
<b>testno</b>	000 <sub>2</sub>	Unordered
<b>testg</b>	001 <sub>2</sub>	Greater
<b>teste</b>	010 <sub>2</sub>	Equal
<b>testge</b>	011 <sub>2</sub>	Greater or equal
<b>testl</b>	100 <sub>2</sub>	Less
<b>testne</b>	101 <sub>2</sub>	Not equal
<b>testle</b>	110 <sub>2</sub>	Less or equal
<b>testo</b>	111 <sub>2</sub>	Ordered

**Action:** For all **TEST<cc>** except **testno**:  
if((mask & AC.cc) != 000<sub>2</sub>)  
    src1 = 1;    #true value  
else  
    src1 = 0;    #false value

**testno:**  
if(AC.cc == 000<sub>2</sub>)  
    src1 = 1;    #true value  
else  
    src1 = 0;    #false value

**Faults:** STANDARD Refer to [Section 6.1.6, “Faults” on page 6-4](#).

**Example:**  
# Assume AC.cc = 100<sub>2</sub>  
testl g9                   # g9 = 0x00000001

<b>Opcode:</b>	<b>teste</b>	22H	COBR
	<b>testne</b>	25H	COBR
	<b>testl</b>	24H	COBR
	<b>testle</b>	26H	COBR
	<b>testg</b>	21H	COBR
	<b>testge</b>	23H	COBR
	<b>testo</b>	27H	COBR
	<b>testno</b>	20H	COBR

**See Also:** **cmpi, cmpdeci, cmpinci**

**6.2.69**    **xnor, xor**

<b>Mnemonic:</b>	<b>xnor</b>	Exclusive Nor		
	<b>xor</b>	Exclusive Or		
<b>Format:</b>	<b>xnor</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>
		reg/lit	reg/lit	reg
	<b>xor</b>	<i>src1</i> ,	<i>src2</i> ,	<i>dst</i>
		reg/lit	reg/lit	reg
<b>Description:</b>	Performs a bitwise XNOR ( <b>xnor</b> instruction) or XOR ( <b>xor</b> instruction) operation on the <i>src2</i> and <i>src1</i> values and stores the result in <i>dst</i> .			
<b>Action:</b>	<b>xnor:</b> $dst = \sim(src2   src1)   (src2 \& src1);$			
	<b>xor:</b> $dst = (src2   src1) \& \sim(src2 \& src1);$			
<b>Faults:</b>	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-4.		
<b>Example:</b>	<pre>xnor r3, r9, r12    # r12 = r9 XNOR r3 xor g1, g7, g4     # g4 = g7 XOR g1</pre>			
<b>Opcode:</b>	<b>xnor</b>	589H	REG	
	<b>xor</b>	586H	REG	
<b>See Also:</b>	<b>and, andnot, nand, nor, not, notand, notor, or, ornot</b>			

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 i960<sup>®</sup> processor architecture supports two methods for making procedure calls:

- A RISC-style branch-and-link: a fast call best suited for calling procedures that do not call 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 the 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 the 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 then handles register and stack management for the call. Since the i960 architecture provides a fully integrated call and return mechanism, coding calls with branch-and-link 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 the called procedure is placed on the *local stack*. The local stack refers to the stack of the 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's 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 that the processor obtains the called procedure's IP from the *system procedure table*. A system call, when executed, is directed to perform either the 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 a *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

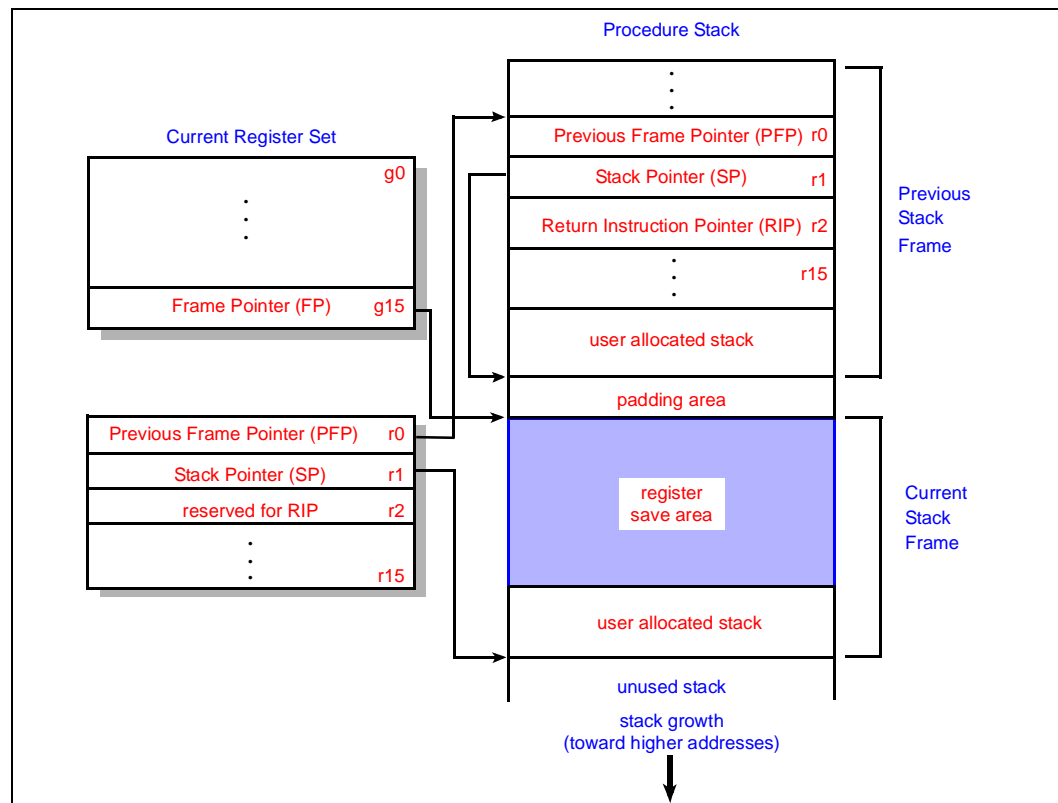
The processor automatically allocates a set of 16 local registers for each procedure. Since local registers are on-chip, they provide fast access storage for local variables. Of the 16 local registers, 13 are available for general use; r0, r1 and r2 are reserved for linkage information to tie procedures 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 i960<sup>®</sup> RM/RN 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 7-17.

### 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 i960 RM/RN 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 7-17](#).
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 6-46](#), 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.

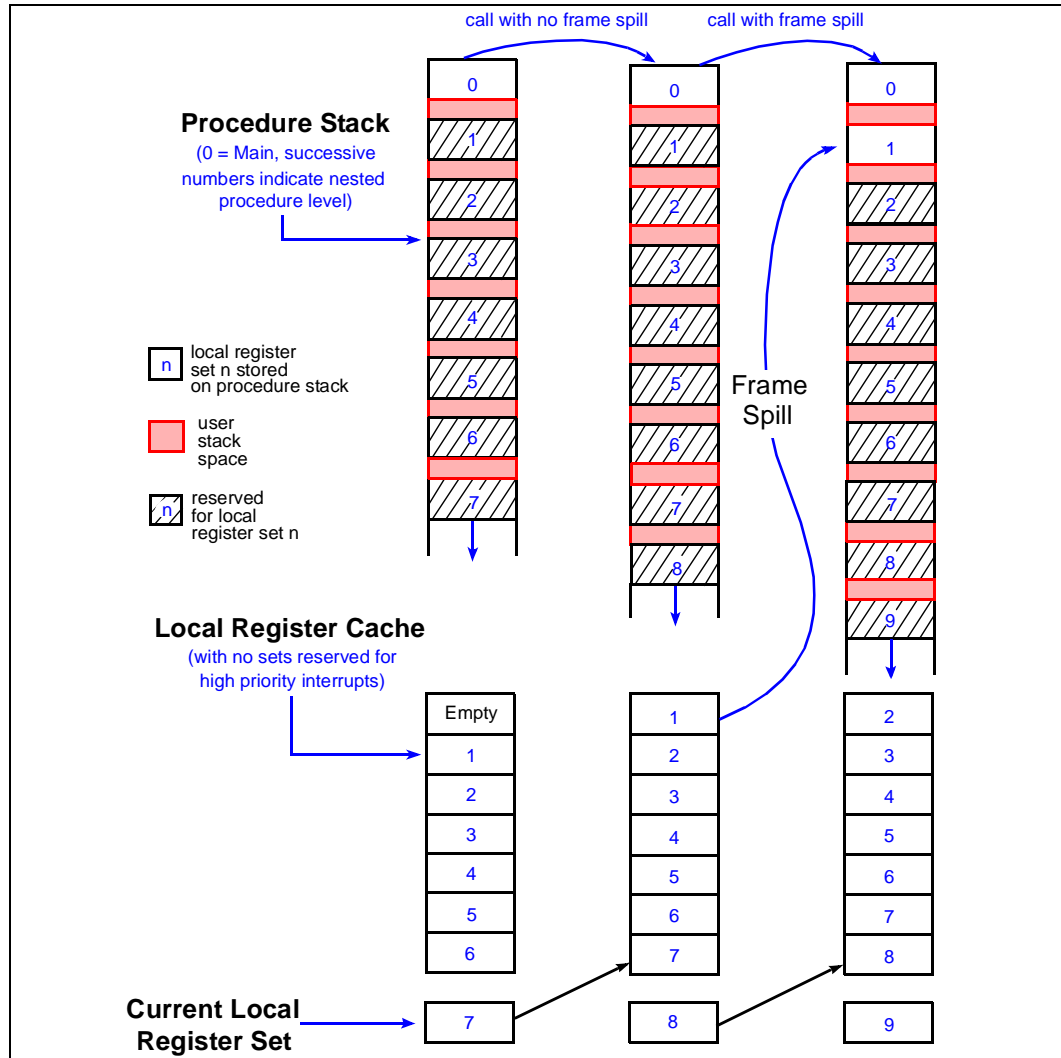
**flushreg** is commonly used in debuggers or fault handlers to gain access to all saved local registers. In this way, call history may be traced back through nested procedures.

### 7.1.4.1 Reserving Local Register Sets for High Priority Interrupts

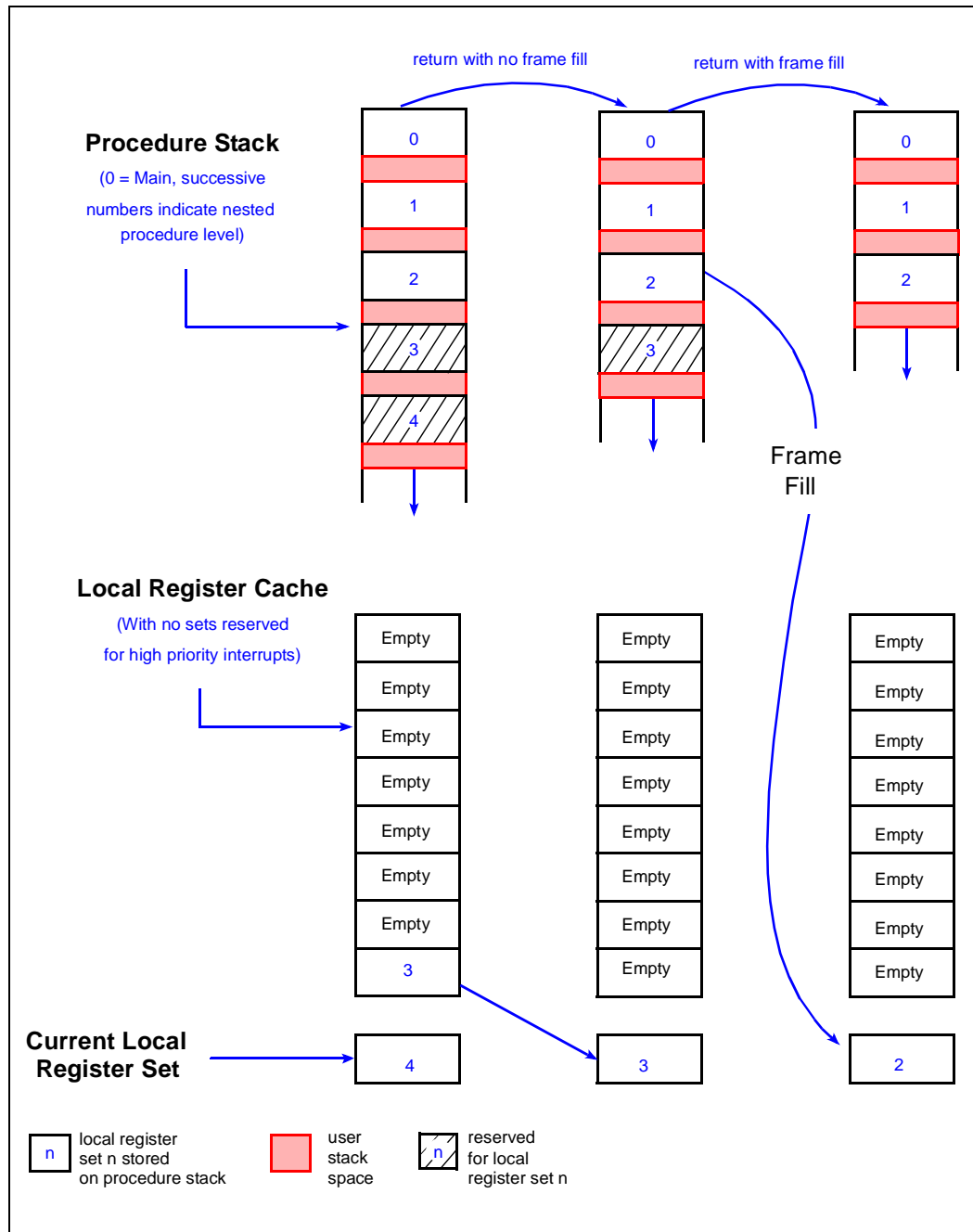
To decrease interrupt latency for high priority interrupts, software can limit the number of frames available to all remaining code. This includes code that is either in the executing state (non-interrupted) or code that is in the interrupted state but has a process priority less than 28. For the purposes 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 11-15](#)). 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 11-14](#).

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:

<i>value</i>	Parameters are passed directly to the calling procedure as part of the call and return mechanism. This is the fastest method of passing parameters.
<i>reference</i>	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 = procl(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       _procl
    mov         g0,r3          # save return value in "a"
    .
    .
_procl:
    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 7-14](#).

**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.)

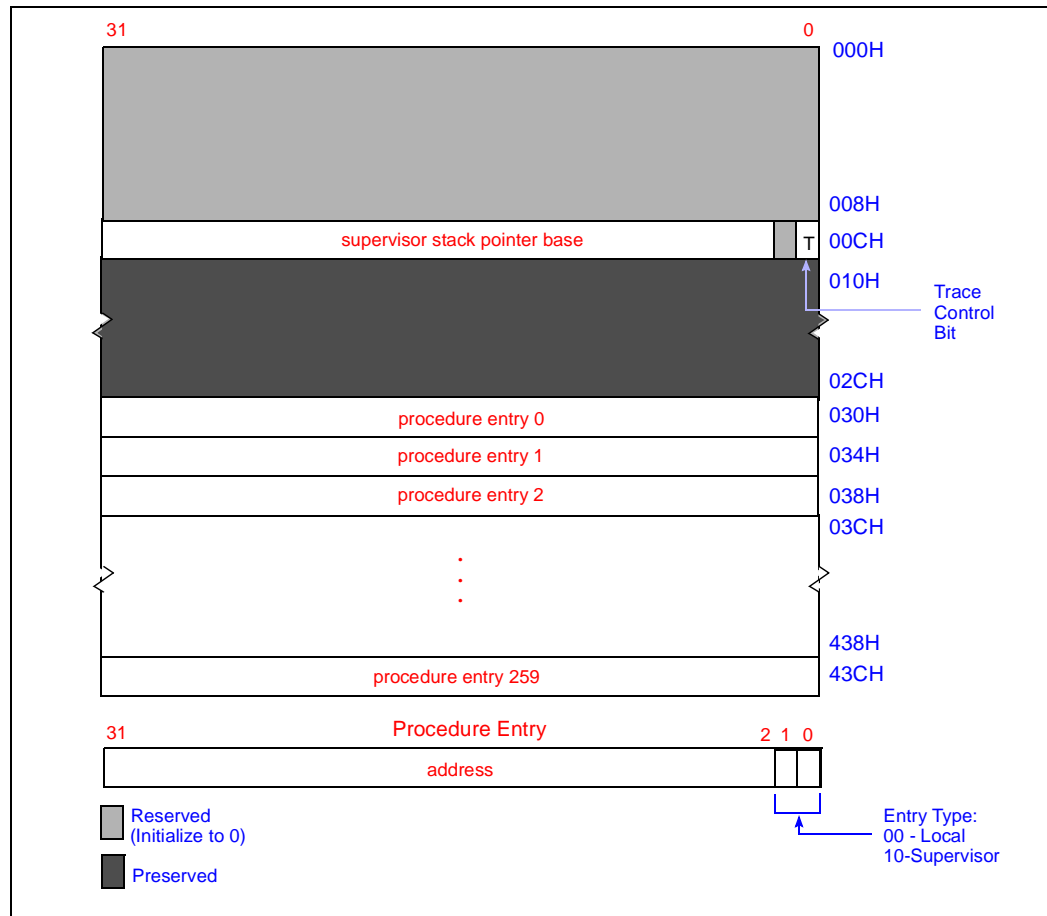
The system call mechanism offers two benefits. First, it supports application software portability. System calls are commonly used to call kernel services. By calling these services with a procedure number rather than a specific IP, applications software does not need to be changed each time the implementation 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 <sup>1</sup>
10	System-Supervisor Call
11	Reserved <sup>1</sup>

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 6-21.

## 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<sub>2</sub>, 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<sub>2</sub> (**calls** instruction) or 001<sub>2</sub> (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, “i960® RM/RN 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.

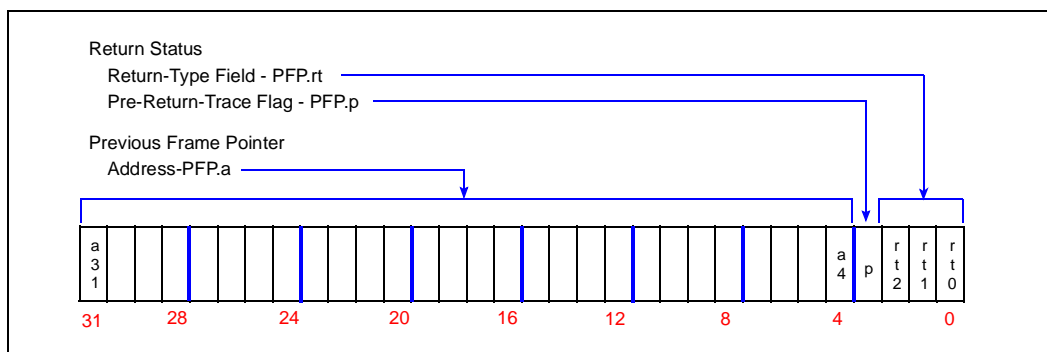
A call to a fault procedure is similar to a system call. Fault procedure calls can be local calls or supervisor calls. The processor obtains pointers to fault procedures through the fault table and (optionally) through the system procedure table.

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 10-12.

*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 <sup>1</sup> bit stored in the PFP register on the call)
100	reserved <sup>2</sup>	
101	reserved <sup>2</sup>	
110	reserved <sup>2</sup>	
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

This chapter describes the i960® RM/RN I/O processor Interrupt Controller Unit, including:

- operation modes
- setup
- external memory interface
- implementation of the interrupts

## 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 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 i960 RM/RN 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.

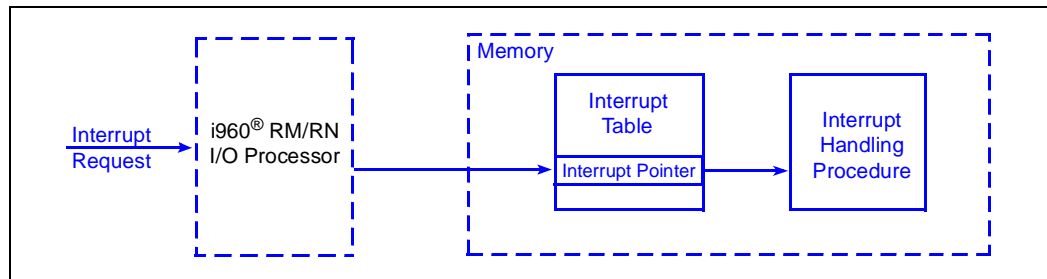
When the interrupt call is made, the processor uses a dedicated interrupt stack. The processor creates a new frame for the interrupt on this stack and a new set of local registers is allocated to the interrupt procedure. The interrupted program's current state is also saved.

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 core architecture 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 SDRAM).

**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 i960 RM/RN I/O processor, interrupt requests may originate from external hardware sources, internal peripherals or software. The i960 RM/RN 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 & 1
- Primary ATU
- Secondary ATU
- I<sup>2</sup>C 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 i960 core processor's 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 i960<sup>®</sup> RM/RN I/O Processor Core Interrupt Architecture

The i960 RM/RN 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.

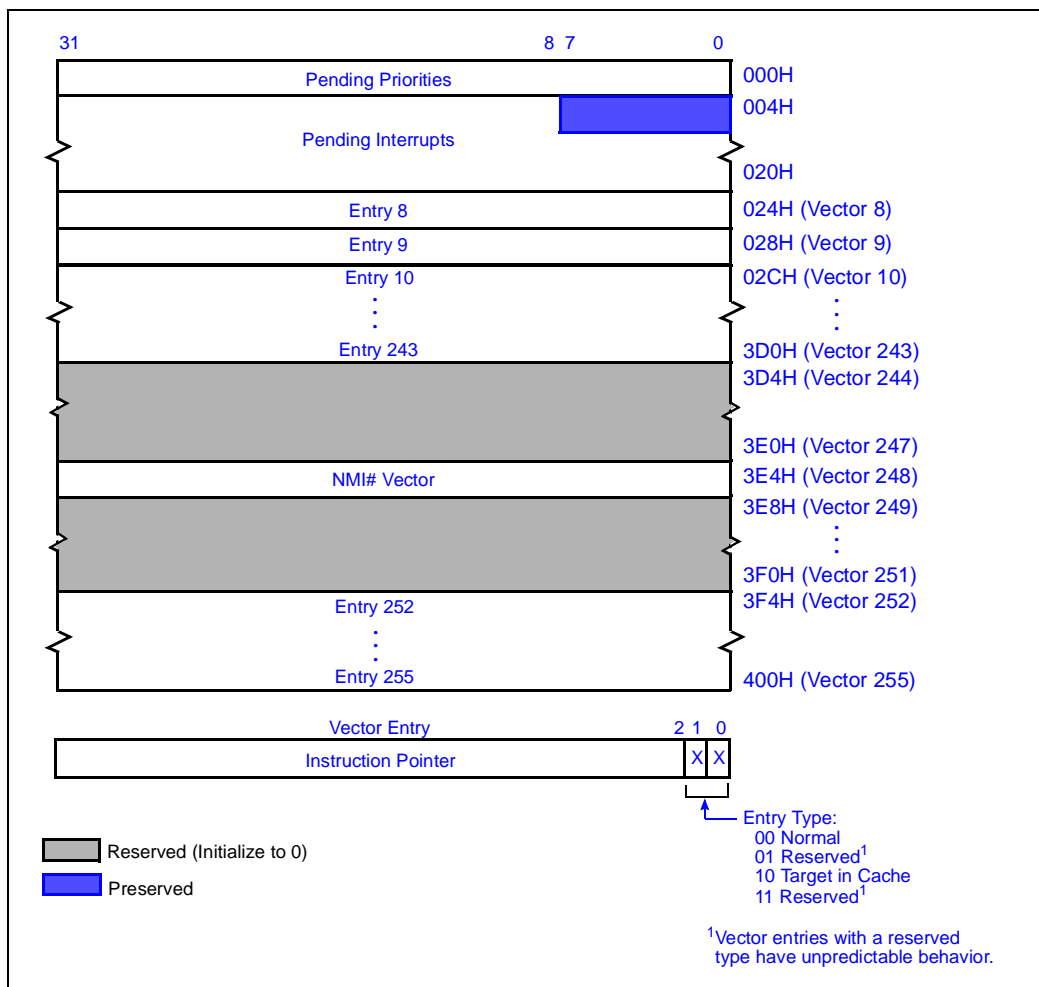
Priority-31 interrupts are handled as a special case. Even when the processor is executing at priority level 31, a priority-31 interrupt interrupts the processor. On the i960 RM/RN I/O processor, the non-maskable interrupt (NMI#) interrupts priority-31 execution; no interrupt can interrupt an NMI# handler.

### 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 that points to a vector entry in the interrupt table. 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 used 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 NMI# vector located in the interrupt table is automatically read and stored in location 0H of internal data RAM. The NMI# vector is subsequently fetched from internal data RAM to improve this interrupt's 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 that the vector's two least significant bits are 0. Bits 0 and 1 of an entry indicate entry type: type 00 indicates that the interrupt procedure should be fetched normally; type 10 indicates that the interrupt procedure should be fetched from the locked partition of the instruction cache. Refer to [Section 8.4.4.2, "Caching Interrupt Routines and Reserving Register Frames" on page 8-29](#). The other possible entry types are reserved and must not be used.

### 8.1.4.2 Pending Interrupts

The pending interrupts section comprises the interrupt table's first 36 bytes, 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's 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 bit in the pending interrupt field 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 i960 RM/RN 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.

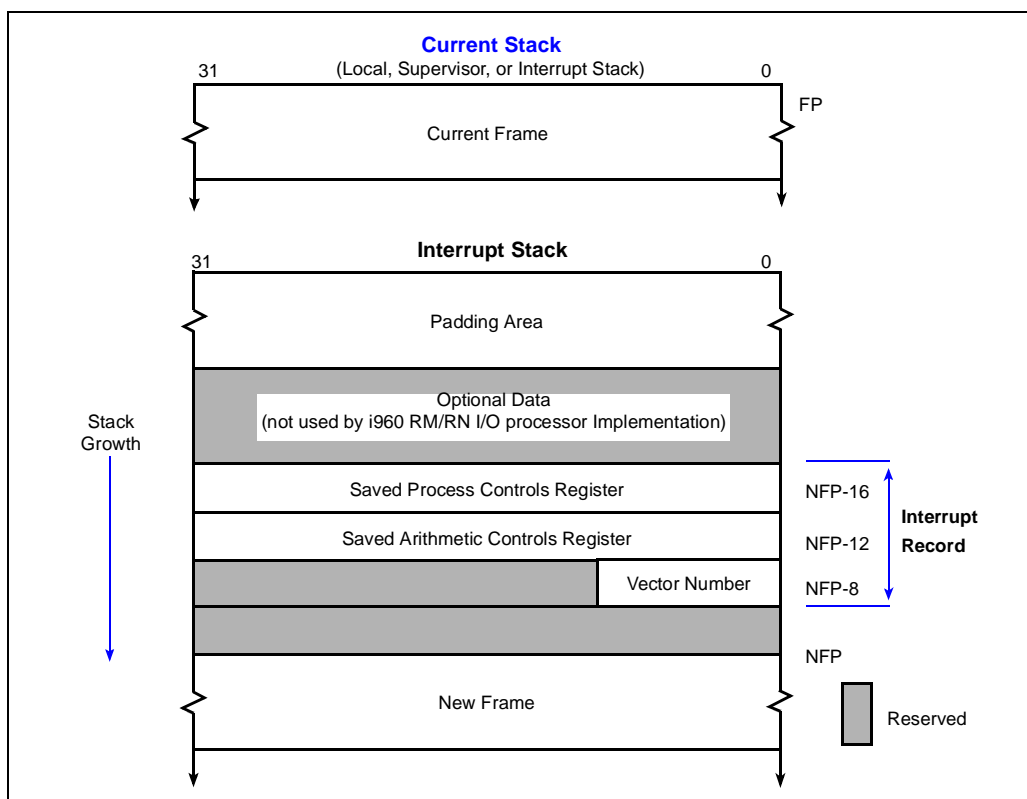
This caching mechanism is non-transparent; the processor may modify fields in a cached interrupt table without modifying the same fields in the interrupt table itself. Vector caching is described in [Section 8.4.4.1, "Vector Caching Option" on page 8-28](#).

## 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. The interrupt stack has the same structure as the local procedure stack described in Section 7.1.1, “Local Registers and the Procedure Stack” on page 7-3. 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 i960 RM/RN 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 i960 RM/RN I/O processor.
- External Interrupts: interrupts posted through the interrupt table, by an external agent to the i960 RM/RN I/O processor.
- Hardware interrupts: interrupts posted directly to the i960 RM/RN 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 i960 RM/RN I/O processor, `sysctl` is typically used to request an interrupt in a program (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 Section 6.2.67, “`sysctl`” on page 6-96 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 value of the highest pending priority 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:

#### Example 8-2. External Agent Posting

```

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 8-3. 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 i960 RM/RN 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 8-4. Check for 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 8.5.1, “Interrupt Control Register \(ICON\)”](#) on page 8-33.

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.

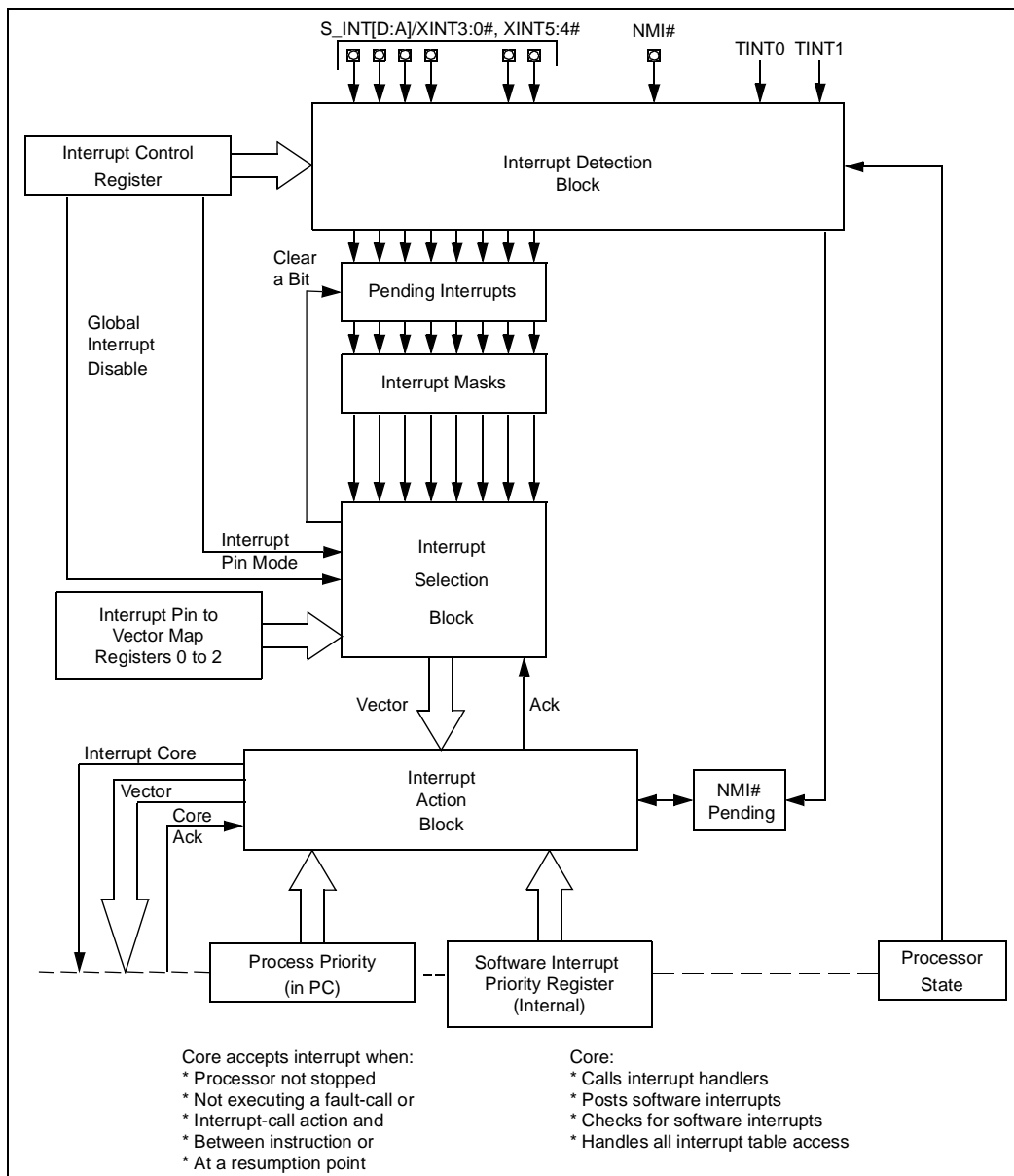
## 8.2 The i960® Core Processor Interrupt Controller

The i960 RM/RN 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
- Two internal interrupts lines

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.

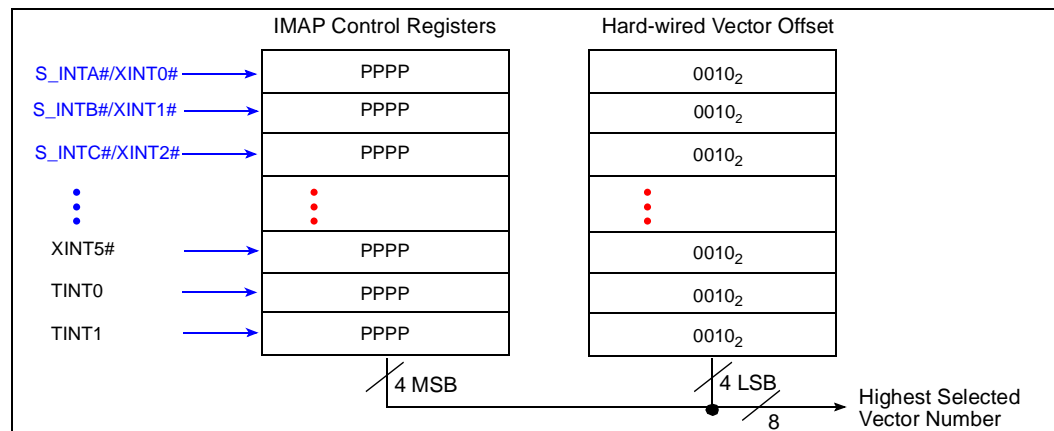
## 8.2.1 Interrupt Controller Dedicated Mode

The i960 RM/RN 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<sub>2</sub> (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<sub>2</sub> 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 8.5, “Register Definitions” on page 8-32 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 8-5. Return from a Level-detect Interrupt

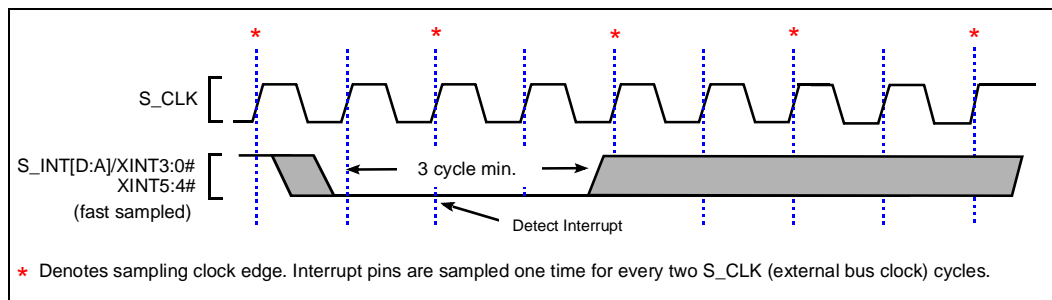
```

# 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 *80960RM I/O Processor Data Sheet* and the *80960RN I/O Processor 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 specifications must also be used to calculate the minimum signal width, as shown in Figure 8-6.

Figure 8-6. Interrupt Fast Sampling



## 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 80960JT 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 i960 RM/RN 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 [Chapter 18, “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 i960 RM/RN 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 IMASK register is masked to allow for setup. A value that selects vector caching is loaded into the ICON register and the IMASK is unmasked.

### Example 8-6. Programming the Interrupt Controller for Vector Caching

```
# Example vector caching setup . . .
mov     0x0, g0
mov     0x00006000, g1
st      g0,IMSK           # mask, IMASK 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 8.4.4.2, "Caching Interrupt Routines and Reserving Register Frames"](#) on page 8-29 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  $pfpr[2:0]$  to  $111_2$ .
7. Performs a call operation as described in the 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 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, “Servicing An Interrupt From Executing State”](#) on page 8-17 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 PCI and Peripheral Interrupts

The PCI And Peripheral Interrupt Controller (PPIC) provides the ability to generate interrupts to both the i960 core processor and the PCI bus. The i960 RM/RN 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<sup>2</sup>C 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:

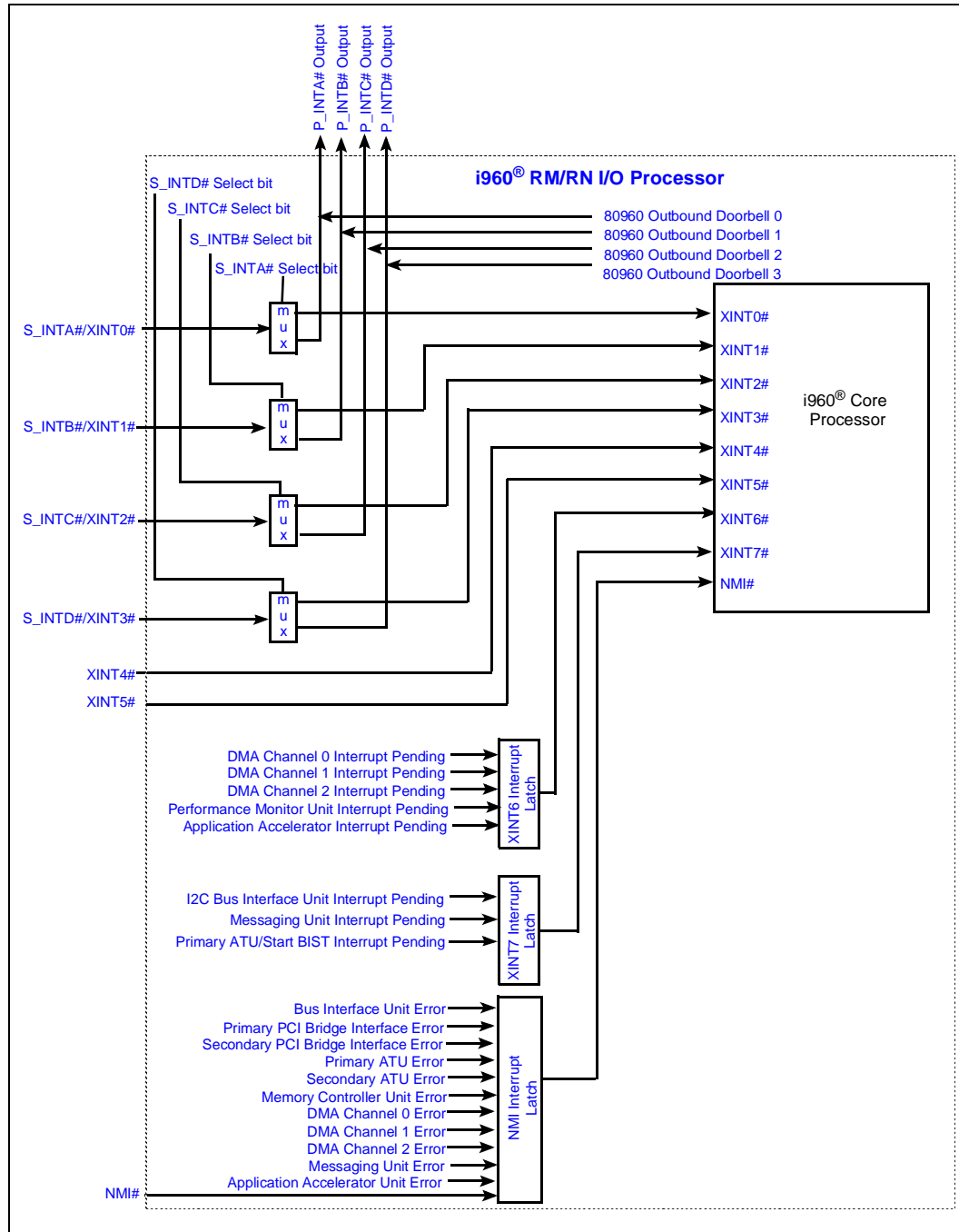
- Internal Peripheral Interrupt Control
- PCI Interrupt Routing

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 i960 RM/RN I/O processor 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 i960 RM/RN I/O processor.

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**



### 8.3.1 Pin Descriptions

The i960 RM/RN I/O processor provides six external interrupt pins and one non-maskable interrupt pin for detecting external interrupt requests. The six external pins are configured as dedicated inputs, where each pin is capable of requesting a single interrupt, in some cases from several different sources. The external interrupt input interface for the i960 RM/RN I/O processor consists of the following pins:

**Table 8-1. Interrupt Input Pin Descriptions**

Signal	Description
S_INTA#/XINT0#	Can be directed to the P_INTA# output or the i960 core interrupt input XINT0#. When routed to the P_INTA# output, this pin is shared with two internal interrupts. They are the interrupts from the Messaging Unit. When routed to the i960 core internal input XINT0#, this input is not shared.
S_INTB#/XINT1#	Can be directed to the P_INTB# output or the i960 core interrupt input XINT1#. When routed to the P_INTB1# output, this pin is shared with two internal interrupts. They are the interrupts from the Messaging Unit. When routed to the i960 core internal input XINT1#, this input is not shared.
S_INTC#/XINT2#	Can be directed to the P_INTC# output or the i960 core interrupt input XINT2#. When routed to the P_INTC2# output, this pin is shared with two internal interrupts. They are the interrupts from the Messaging Unit. When routed to the i960 core internal input XINT2#, this input is not shared.
S_INTD#/XINT3#	Can be directed to the P_INTD# output or the i960 core interrupt input XINT3#. When routed to the P_INTD# output, this pin is shared with two internal interrupts. They are the interrupts from the Messaging Unit. When routed to the i960 core internal input XINT3#, this input is not shared.
XINT4#	Always connected to the i960 core interrupt input XINT4#.
XINT5#	Always connected to the i960 core interrupt input XINT5#.
NMI#	Shared with ten internal interrupts. They include error interrupts from the core processor, primary PCI bridge interface, secondary PCI bridge interface, primary ATU, secondary ATU, three DMA channels, application accelerator, and the messaging unit. All of the interrupts are directed to the i960 core NMI# input. Software must read the NMI Interrupt Status Register to determine the exact source of the interrupt. NMI# is the highest priority interrupt recognized.

All pins in [Table 8-1](#) are level-low activated. See [Section 8.2.2, “Interrupt Detection”](#) on page 8-13.

## 8.3.2 PCI Interrupt Routing

The four secondary PCI interrupt inputs can be routed to either i960 core processor interrupt inputs or to primary PCI interrupt output pins. Routing of interrupt inputs is controlled by bits 3:0 in the PCI Interrupt Routing Select Register (PIRSR). [Table 8-2](#) summarizes the usage of these bits.

**Table 8-2. 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:** Please check the *i960<sup>®</sup> RM/RN I/O Processor Specification Update* for possible issues with the PIRSR.

## 8.3.3 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.

### 8.3.3.1 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 Unit. Each DMA channel interrupt is either for a DMA End of Transfer interrupt or a 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 application accelerator 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	Channel Status Register 0
	End of Transfer	Channel Status Register 0
DMA Channel 1	End of Chain	Channel Status Register 1
	End of Transfer	Channel Status Register 1
DMA Channel 2	End of Chain	Channel Status Register 2
	End of Transfer	Channel Status Register 2
Application Accelerator	End of Chain	Accelerator Status Register
	End of Transfer	Accelerator Status Register
Performance Monitor	Counter Overflow	Event Monitoring Interrupt Status Register

### 8.3.3.2 XINT7 Interrupt Sources

The XINT7# interrupt on the i960 core processor receives interrupts from the I<sup>2</sup>C 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	Interrupt Condition	Register
I <sup>2</sup> C Bus Interface Unit	Receive Buffer Full	I <sup>2</sup> C Status Register
	Transmit Buffer Empty	
	Slave Address Detect	
	STOP Detected	
	Bus Error Detected	
	Arbitration Lost Detected	
Messaging Unit	Index Register Interrupt	Inbound Interrupt Status Register
	Inbound Post Queue Interrupt	
	Inbound Doorbell Interrupt	
	Inbound Message 1 Interrupt	
	Inbound Message 0 Interrupt	
Primary ATU	ATU BIST Start	Primary ATU Interrupt Status Register



### 8.3.3.3 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 NMI# input of the i960 core processor is edge-triggered. The external NMI# input of the i960 RM/RN 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.

The unit interrupt sources which drive the inputs to the NMI interrupt latch are detailed in [Table 8-5](#).

**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 (PIRSR), XINT6 Interrupt Status Register (X6ISR), XINT7 Interrupt Status Register (X7ISR), and NMI Interrupt Status Register (NISR) are described in [Section 8.5, “Register Definitions”](#) on page 8-32.

### 8.3.4 PCI Outbound Doorbell Interrupts

The i960 RM/RN 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 generates 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) determines which PCI interrupt (P\_INTA# through P\_INTD#) is generated. Refer to *PCI Local Bus Specification Revision 2.1* for complete details on the bit-field definition of the ATUIPR.

## 8.4 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	0000H	S_INTA#/XINT0# routed to the i960 core processor S_INTB#/XINT1# routed to the i960 core processor S_INTC#/XINT2# routed to the i960 core processor S_INTD#/XINT3# routed to the i960 core processor
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

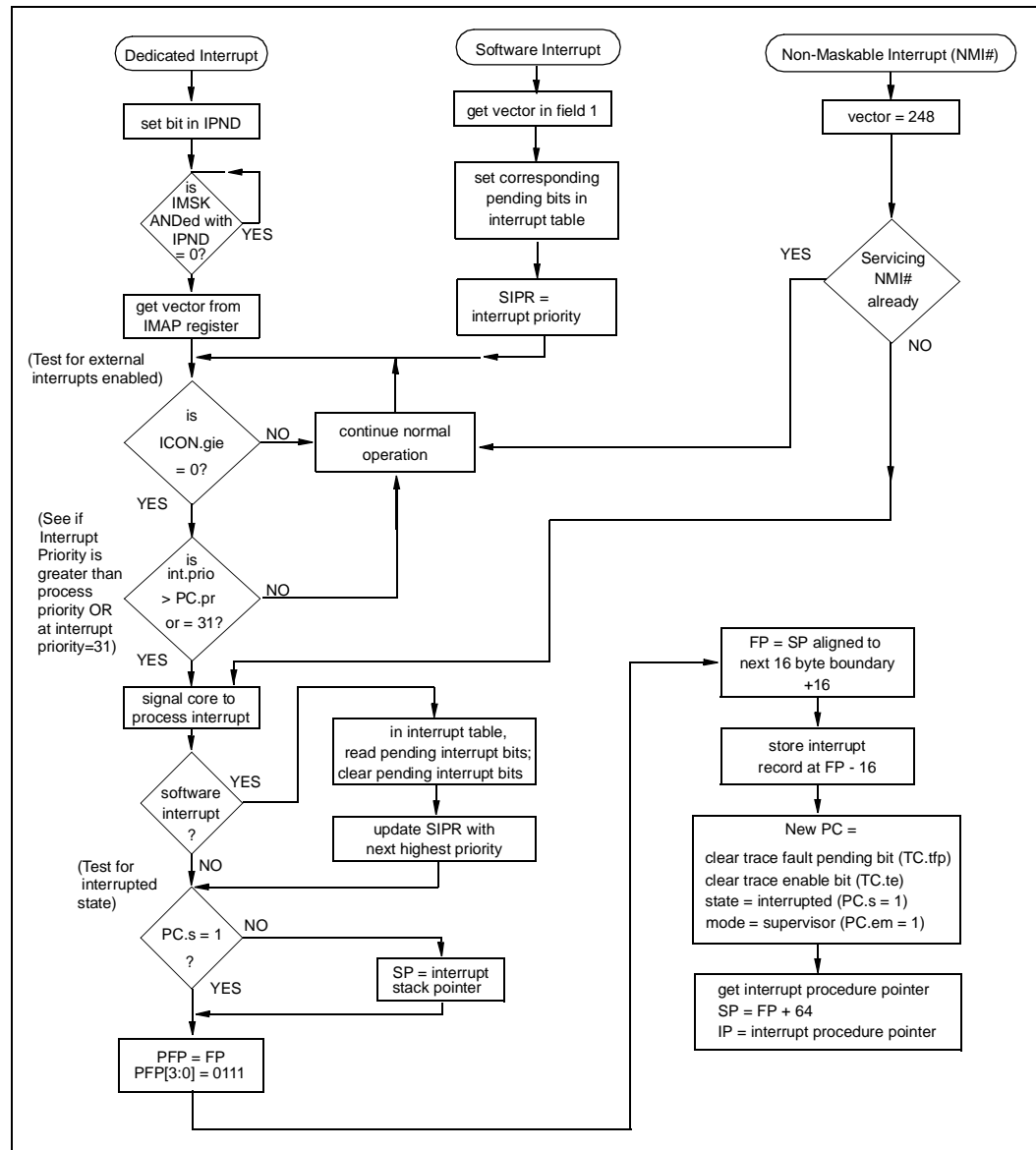
## 8.4.1 Interrupt Controller Register Access Requirements

A load instruction that accesses the IPND, IMSK, IMA2:0 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 [Chapter 6, “Instruction Set Reference”](#) for more information on these instructions.

## 8.4.2 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**


### 8.4.3 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.

## 8.4.4 Features to Improve Interrupt Performance

The i960 RM/RN 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

### 8.4.4.1 Vector Caching Option

To reduce interrupt latency, the i960 RM/RN 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 0004H, 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 <sub>2</sub>	18	0004H
0010 0010 <sub>2</sub>	34	0008H
0011 0010 <sub>2</sub>	50	000CH
0100 0010 <sub>2</sub>	66	0010H
0101 0010 <sub>2</sub>	82	0014H
0110 0010 <sub>2</sub>	98	0018H
0111 0010 <sub>2</sub>	114	001CH
1000 0010 <sub>2</sub>	130	0020H
1001 0010 <sub>2</sub>	146	0024H
1010 0010 <sub>2</sub>	162	0028H
1011 0010 <sub>2</sub>	178	002CH
1100 0010 <sub>2</sub>	194	0030H
1101 0010 <sub>2</sub>	210	0034H
1110 0010 <sub>2</sub>	226	0038H
1111 0010 <sub>2</sub>	242	003CH

### 8.4.4.2 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 i960 RM/RN 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 [Section 4.2, "Local Register Cache" on page 4-2](#), for more details.

### 8.4.4.3 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 [Section 12.2.1, "PMCON Registers" on page 12-1](#) for details on how to enable data caching for portions of memory.

## 8.4.5 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 8-8](#) 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) <sup>1,2</sup>
NMI#	Yes	30
XINT[5:4]#, TINT1:0	Yes	34
	No	40+a
XINT[7:6]#, XINT[3:0]#	Yes	35
	No	41+a
Software	Yes	68
	No	69+a

**NOTES:**

1. a = MAX (0, N - 7), where "N" is the number of bus cycles needed to perform a word load.
2. Bus Clocks are 80960 core processor bus clocks. The core processor bus is typically 100 MHz.



### 8.4.6 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 8-8, 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) <sup>1,2</sup>
NMI#	Yes	43
XINT[5:4]#, TINT1:0	Yes	45
	No	45+a
XINT[7:6]# XINT[3:0]#	Yes	46
	No	46+a

**NOTES:**

- 1. a = MAX (0,N - 11), where “N” is the number of bus cycles needed to perform a word load.
- 2. Bus Clocks are 80960 core processor bus clocks. The core processor bus is typically 100 MHz.

**Table 8-10. Worst-Case Interrupt Latency Controlled by divo to Destination r3**

Interrupt Type	Vector Caching Enabled	Worst Latency (Bus Clocks) <sup>1,2</sup>
NMI#	Yes	60
XINT[5:4]#, TINT1:0	Yes	65
	No	72+a
XINT[7:6]# XINT[3:0]#	Yes	66
	No	73+a

**NOTES:**

- 3. a = MAX (0,N - 7), where “N” is the number of bus cycles needed to perform a word load.
- 4. Bus Clocks are 80960 core processor bus clocks. The core processor bus is typically 100 MHz.

**Table 8-11. Worst-Case Interrupt Latency Controlled by calls**

Interrupt Type	Vector Caching Enabled	Worst Latency (Bus Clocks) <sup>1,2</sup>
NMI#	Yes	54+a
XINT[5:4]#, TINT1:0	Yes	58+a
	No	66+a+b
XINT[7:6]# XINT[3:0]#	Yes	59+a
	No	67+a+b

**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.

- 2. Bus Clocks are 80960 core processor bus clocks. The core processor bus is typically 100 MHz.



**Table 8-12. Worst-Case Interrupt Latency When Delivering a Software Interrupt**

Interrupt Type	Vector Caching Enabled	Worst Latency (Bus Clocks) <sup>1,2</sup>
NMI#	Yes	97
XINT[5:4]#, TINT1:0	Yes	99
	No	107+a
XINT[7:6]# XINT[3:0]#	Yes	100
	No	108+a

**NOTES:**

- a = MAX (0, N - 7), where "N" is the number of bus cycles needed to perform a word load.
- Bus Clocks are 80960 core processor bus clocks. The core processor bus is typically 100 MHz.

**Table 8-13. Worst-Case Interrupt Latency Controlled by flushreg of One Stack Frame**

Interrupt Type	Vector Caching Enabled	Worst Latency (Bus Clocks) <sup>1,2</sup>
NMI#	Yes	78+a+b
XINT[5:4]#, TINT1:0	Yes	82+a+b
	No	89+a+b+c
XINT[7:6]# XINT[3:0]#	Yes	83+a+b
	No	90+a+b+c

**NOTES:**

- 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.

- Bus Clocks are 80960 core processor bus clocks. The core processor bus is typically 100 MHz.

## 8.4.7 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.)

## 8.4.8 Secondary PCI to Primary PCI 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 80960JT core interrupt controller or the PCI output interrupt pins.

## 8.5 Register Definitions

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
XINT6	XINT6 Interrupt Status Register	0000 1708H
XINT7	XINT7 Interrupt Status Register	0000 1704H
NISR	NMI Interrupt Status Register	0000 1700H

All ten registers are visible as i960 RM/RN 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 i960 RM/RN I/O processor processor memory space. The memory-mapped addresses of the interrupt control registers are found in [Appendix C, "Memory-Mapped Registers"](#).

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 3 for additional information regarding the PCI configuration cycles that access the PCI Interrupt Routing Select Register.

## 8.5.1 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**

Bit	Default	Description
31:15	Default Value loaded from image in Control Table	Reserved. These bits must be cleared (0).
14		Reserved. 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 <sub>2</sub> 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 <b>inten</b> , <b>intdis</b> , <b>intctl</b> .
9:0		Reserved. These bits must be cleared (0).

Internal bus address FF00 8510H	Attribute Legend: RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set	RW = Read/Write RC = Read Clear RO = Read Only NA = Not Accessible
------------------------------------	--	---



## 8.5.2 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<sub>2</sub>. In other words; each source can be programmed for a vector number of PPPP 0010<sub>2</sub>, 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<sub>2</sub>, the pin is mapped to vector number 0110 0010<sub>2</sub> (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)**

IOP Attributes	31	28	24	20	16	12	8	4	0
	rv	rv	rv	rv	rv	rv	rv	rv	rv
PCI Attributes	x	x	x	x	x	x	x	x	x
	3	3	3	3	2	2	2	1	1
	na	na	na	na	na	na	na	na	na
Internal bus address FF00 8520H					Attribute Legend:				
					RW = Read/Write				
					RV = Reserved				
					RC = Read Clear				
					PR = Preserved				
					RO = Read Only				
					RS = Read/Set				
					NA = Not Accessible				
Bit	Default	Description							
31:16	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							
9:4		External Interrupt 1 Field - IMAP0.x1							
3:0		External Interrupt 0 Field - IMAP0.x0							

**Table 8-17. Interrupt Map Register 1 (IMAP1)**

		31	28	24	20	16	12	8	4	0	
IOP Attributes		rv	rv	rv	rv	rv	rv	rv	rv	rv	
		rv	rv	rv	rv	rv	rv	rv	rv	rv	
PCI Attributes		na	na	na	na	na	na	na	na	na	
		na	na	na	na	na	na	na	na	na	
		Internal bus address FF00 8524H					Attribute Legend: RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set 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		Internal Interrupt 7 Field - IMAP1.x7									
11:8		Internal 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)**

		31	28	24	20	16	12	8	4	0	
IOP Attributes		rv	rv	rv	rv	rv	rv	rv	rv	rv	
		rv	rv	rv	rv	rv	rv	rv	rv	rv	
PCI Attributes		na	na	na	na	na	na	na	na	na	
		na	na	na	na	na	na	na	na	na	
		Internal bus address FF00 8528H					Attribute Legend: RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set 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)									

### 8.5.3 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# - XINT5#), internal interrupts (XINT6#-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 six external dedicated sources, two internal 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**

Bit	Default	Description
31:14	0	Reserved (initialize to 0)
13:12	x	Timer Interrupt Pending Bits - IPND, tip (0) No Interrupt (1) Pending Interrupt
11:8	0000 <sub>2</sub>	Reserved (initialize to 0)
7:6	x	Internal Interrupt Pending Bits - IPND, xip (0) No Interrupt (1) Pending Interrupt
5:0	x	External Interrupt Pending Bits - IPND, xip (0) No Interrupt (1) Pending Interrupt

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
------------------------------------	---	--

31	28	24	20	16	12	8	4	0
IOP Attributes	rv rv rv rv	rv rv rv rv	rv rv rv rv	rv rv rv rv	rv rv rv rv	rv rv rv rv	rv rv rv rv	rv rv rv rv
PCI Attributes	na na na na	na na na na	na na na na	na na na na	na na na na	na na na na	na na na na	na na na na



**Table 8-20. Interrupt Mask (IMSK) Register**

31	28	24	20	16	12	8	4	0
IOP Attributes								
rv	rv	rv	rv	rv	rv	rv	rv	rv
PCI Attributes								
na	na	na	na	na	na	na	na	na
Internal bus address FF00 8504H					Attribute Legend:			
					RV = Reserved		RW = Read/Write	
					PR = Preserved		RC = Read Clear	
					RS = Read/Set		RO = Read Only	
					NA= Not Accessible			
<b>Bit</b>	<b>Default</b>	<b>Description</b>						
31:14	0	Reserved (initialize to 0)						
13:12	00 <sub>2</sub>	Timer Interrupt Pending Bits - IMSK, tim (0) Masked (1) Not Masked						
11:8	0000 <sub>2</sub>	Reserved (initialize to 0)						
7:6	00H	Internal Interrupt Mask Bits - IMSK, xim (0) Masked (1) Not Masked						
5:0	00H	External Interrupt Mask Bits - IMSK, xim (0) Masked (1) Not Masked						



## 8.5.4 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 RM/RN I/O 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.

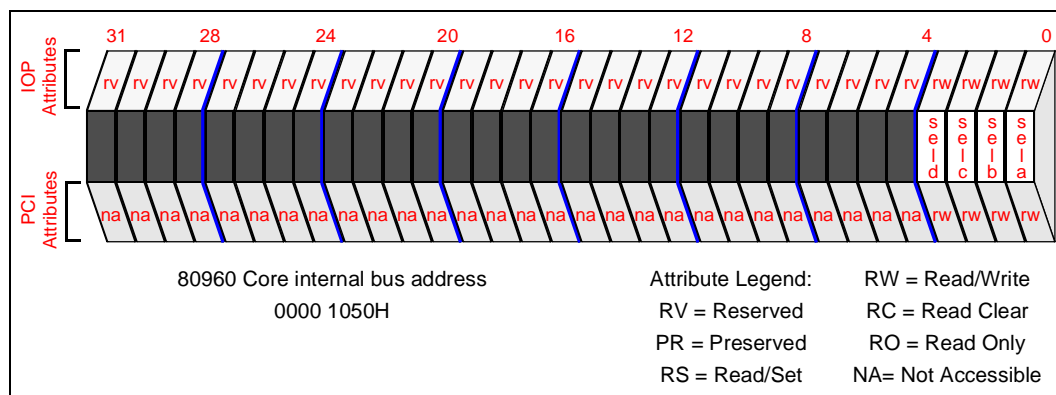
When any or all of the secondary PCI interrupt inputs are routed to the primary PCI interrupt pins, the corresponding i960 core processor inputs (XINT3:0#) must be set inactive (level ‘1’).

Table 8-21 shows the bit definitions for programming the PCI Interrupt Routing Select Register.

**Table 8-21. PCI Interrupt Routing Select Register (PIRSR)**

Bit	Default	Description
31:4	0	Reserved (initialize to 0)
3	0	<b>S_INTD#</b> Select Bit - PIRSR, xsel (1) Interrupt routed to 80960 core interrupt controller input (XINT3#) (0) Interrupt routed to P_INTD# pin
2	0	<b>S_INTC#</b> Select Bit - PIRSR, xsel (1) Interrupt routed to 80960 core interrupt controller input (XINT2#) (0) Interrupt routed to P_INTC# pin
1	0	<b>S_INTB#</b> Select Bit - PIRSR, xsel (1) Interrupt routed to 80960 core interrupt controller input (XINT1#) (0) Interrupt routed to P_INTB# pin
0	0	<b>S_INTA#</b> Select Bit - PIRSR, xsel (1) Interrupt routed to 80960 core interrupt controller input (XINT0#) (0) Interrupt routed to P_INTA# pin

**NOTE:** Please check the *i960<sup>®</sup> RM/RN I/O Processor Specification Update* for possible issues with the PIRSR.





## 8.5.5 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 8.3.3, on page 8-21.

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) is cleared. The X6ISR reflects the current state of the input to the XINT6# Interrupt Latch.

Due to the asynchronous nature of the i960 RM/RN 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 is 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**

		<p>Internal bus address 0000 1708H</p> <p>Attribute Legend:                  RW = Read/Write                  RV = Reserved                  PR = Preserved                  RS = Read/Set                  RC = Read Clear                  RO = Read Only                  NA= Not Accessible</p>																						
			<table border="1"> <thead> <tr> <th>Bit</th> <th>Default</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>31:06</td> <td>0</td> <td>Reserved</td> </tr> <tr> <td>05</td> <td>0</td> <td>Application Accelerator Interrupt Pending - when set, an end of chain condition has been signaled by the Application Accelerator. When clear, no interrupt condition exists.</td> </tr> <tr> <td>04</td> <td>0<sub>2</sub></td> <td>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.</td> </tr> <tr> <td>03</td> <td>0<sub>2</sub></td> <td>Reserved.</td> </tr> <tr> <td>02</td> <td>0<sub>2</sub></td> <td>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.</td> </tr> <tr> <td>01</td> <td>0<sub>2</sub></td> <td>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.</td> </tr> <tr> <td>00</td> <td>0<sub>2</sub></td> <td>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.</td> </tr> </tbody> </table>	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 <sub>2</sub>	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 <sub>2</sub>	Reserved.	02	0 <sub>2</sub>	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 <sub>2</sub>	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.
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 <sub>2</sub>	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 <sub>2</sub>	Reserved.																						
02	0 <sub>2</sub>	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 <sub>2</sub>	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 <sub>2</sub>	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.																						

## 8.5.6 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 8.3.3, on page 8-21.

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) is cleared. The X7ISR reflects the current state of the input to the XINT7 Interrupt Latch.

Due to the asynchronous nature of the i960 RM/RN 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 is 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**

Bit	Default	Description
31:05	0	Reserved
04	0 <sub>2</sub>	Reserved
03	0 <sub>2</sub>	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 <sub>2</sub>	Messaging Unit Interrupt Pending - when set, an interrupt from the Messaging Unit is pending. When clear, no interrupt is pending.
01	0 <sub>2</sub>	I <sup>2</sup> C Interrupt Pending - when set, an interrupt is from the I <sup>2</sup> C Bus Interface Unit is pending. When clear, no interrupt is pending.
00	0 <sub>2</sub>	Reserved

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
------------------------------------	---

IOP Attributes 31 28 24 20 16 12 8 4 0 rv b i s t i n d b i s q c rv ro ro rv	PCI Attributes na
--	---



## 8.5.7 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 8.3.3, on page 8-21](#).

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](#)) is cleared. The NISR reflects the current state of the input to the NMI Interrupt Latch.

Due to the asynchronous nature of the i960 RM/RN 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 is 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 i960 RM/RN I/O processor requires a level input and must be latched external to the i960 RM/RN I/O processor.

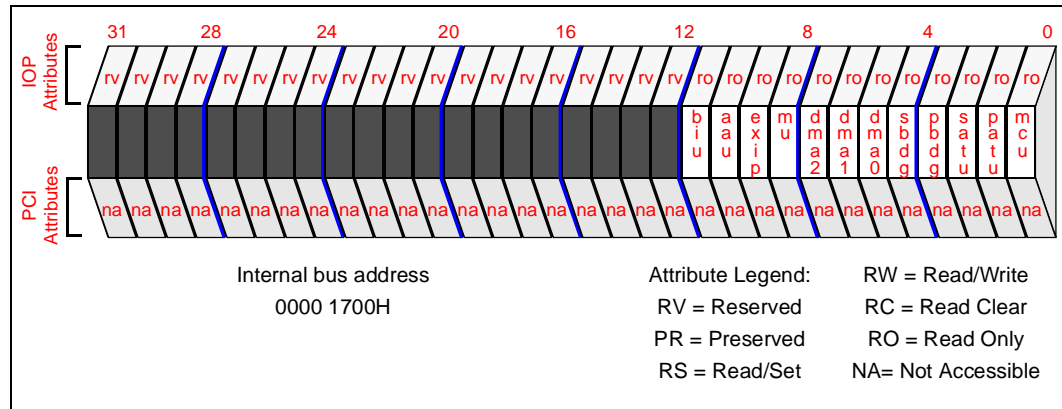
[Table 8-24](#) details the bit definitions for the NMI interrupt status register.

**Table 8-24. NMI Interrupt Status Register- NISR (Sheet 1 of 2)**

		31	28	24	20	16	12	8	4	0		
IOP Attributes		rv	rv	rv	rv	rv	rv	rv	rv	rv		
		rv	rv	rv	rv	rv	rv	rv	rv	rv		
PCI Attributes		na	na	na	na	na	na	na	na	na		
		na	na	na	na	na	na	na	na	na		
		Internal bus address 0000 1700H						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	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 <sub>2</sub>	Application Accelerator Unit Error - when set, an internal bus error condition exists within AA unit. When clear, no error exists.										
09	0 <sub>2</sub>	External NMI# Interrupt - when set, an interrupt is pending on the external NMI# input. When clear, no interrupt exists.										
08	0 <sub>2</sub>	Messaging Unit Interrupt - when set, an NMI interrupt or error exists in the Messaging Unit. When clear, no error exists.										

**Table 8-24. NMI Interrupt Status Register- NISR (Sheet 2 of 2)**

Bit	Default	Description
07	0 <sub>2</sub>	DMA Channel 2 Error - when set, a PCI or internal bus error condition exists within DMA channel. When clear, no error exists.
06	0 <sub>2</sub>	DMA Channel 1 Error - when set, a PCI or internal bus error condition exists within DMA channel. When clear, no error exists.
05	0 <sub>2</sub>	DMA Channel 0 Error - when set, a PCI or internal bus error condition exists within DMA channel. When clear, no error exists.
04	0 <sub>2</sub>	Secondary Bridge Error - when set, a PCI error condition exists within the secondary interface of the bridge. When clear, no error exists.
03	0 <sub>2</sub>	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 <sub>2</sub>	Secondary ATU Error - when set, a PCI or internal bus error condition exists within the secondary ATU. When clear, no error exists.
01	0 <sub>2</sub>	Primary ATU Error - when set, a PCI or internal bus error condition exists within the primary ATU. When clear, no error exists.
00	0 <sub>2</sub>	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.





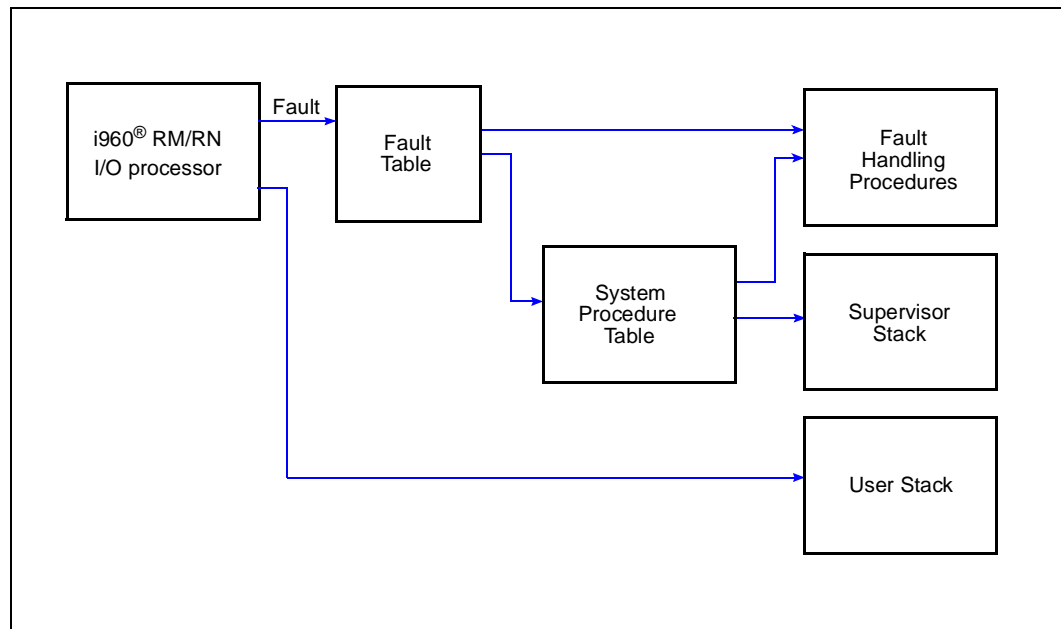
This chapter describes the i960® RM/RN I/O processor's fault handling facilities. Subjects covered include the fault handling data structures and fault handling mechanisms. See [Section 9.10, "Fault Reference"](#) on page 9-20 for detailed information on each fault type.

## 9.1 Fault Handling Overview

The i960 processor 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 7-17](#) 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 9-20](#)), 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 i960 RM/RN 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 i960 RM/RN I/O processor detects, arranged by type and subtype. Text that follows the table gives column definitions.

**Table 9-1. i960<sup>®</sup> RM/RN I/O Processor Fault Types and Subtypes**

Fault Type		Fault Subtype		Fault Record
Number	Name	Number or Bit Position	Name	
0H	PARALLEL	NA	NA	see <a href="#">Section 9.6.4, "Parallel Faults"</a> on page 9-9
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.

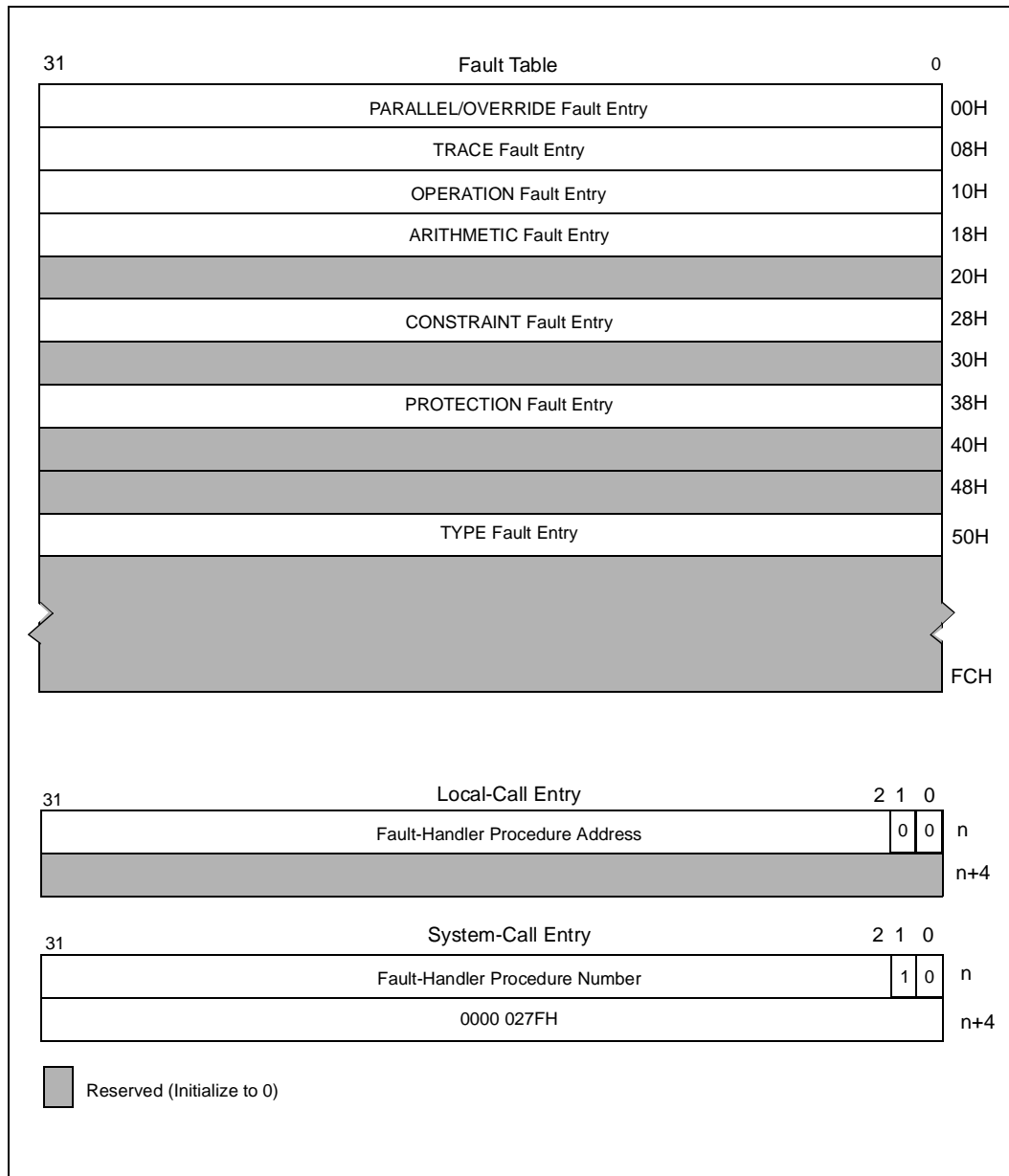
Other i960 processor family members may provide extensions that recognize additional fault conditions. Fault type and subtype encoding allows all faults to be included in the fault table: those that are common to all i960 processors and those that are specific to one or more family members. The fault 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 9-2](#)) 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 9-2](#), 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.

<i>local-call entry</i> (type 00 <sub>2</sub> )	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.
<i>system-call entry</i> (type 10 <sub>2</sub> )	Provides a procedure number in the system procedure table. This entry must have an entry type of 10 <sub>2</sub> 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 <b>calls</b> 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<sub>2</sub> and 11<sub>2</sub>) 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.

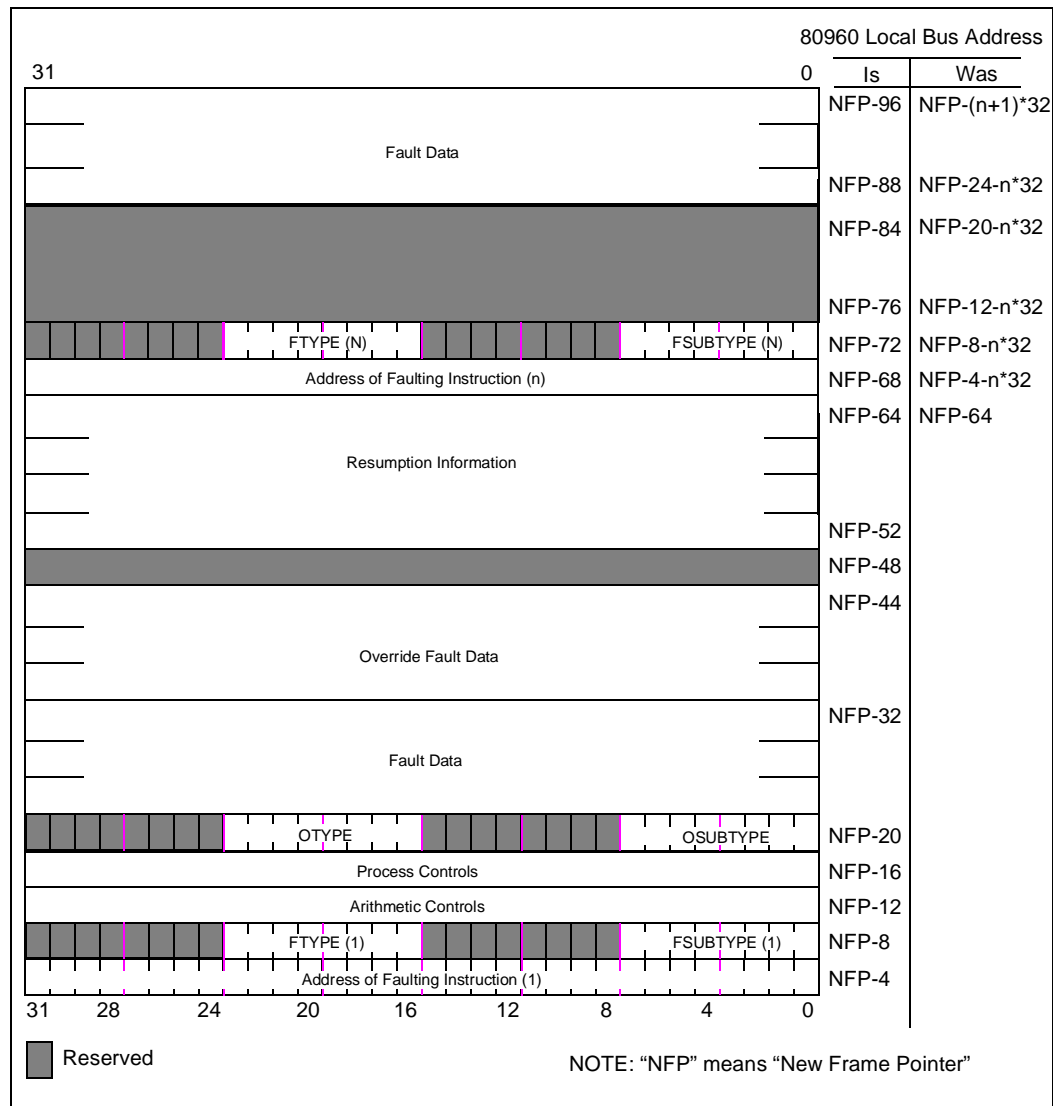
## 9.5.1 Fault Record Description

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.

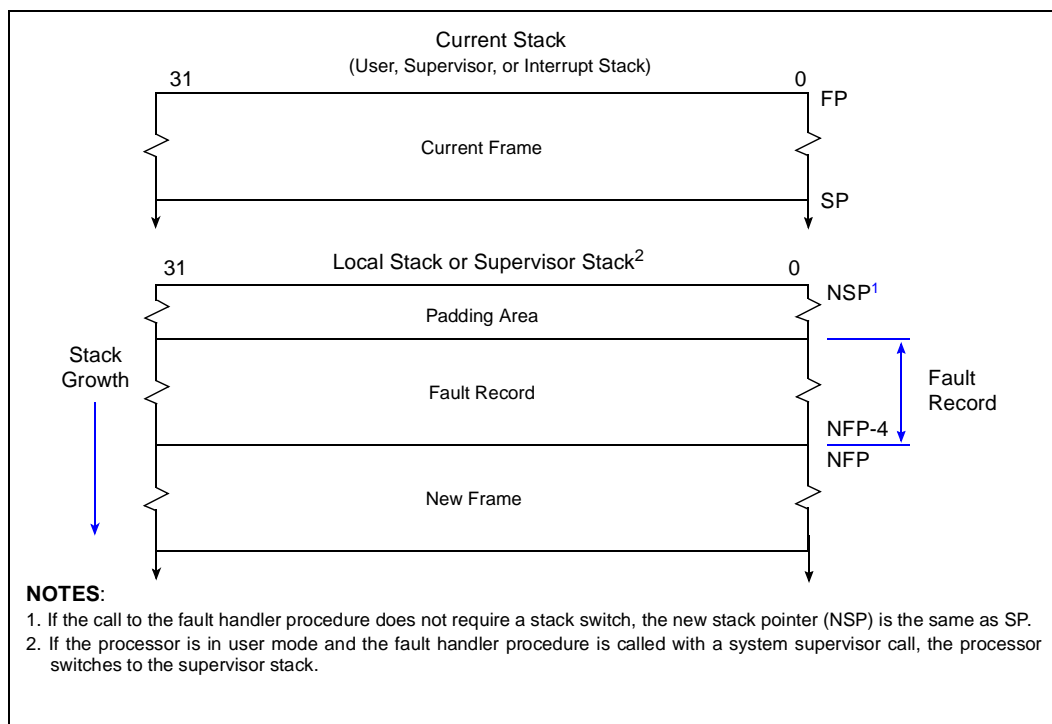
Figure 9-3. Fault Record



## 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



## 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 9-18). 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 9-20.
- 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 i960 RM/RN I/O processor exploits the architecture’s 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 9-19). 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 **mul** and the **add** 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 **add** and **mul** are allowed to execute in parallel because AC.nif = 0 and the faults that these instructions can generate (ARITHMETIC) are imprecise.

### Example 9-1. Imprecise Fault Generations

```

mul    g2, g4, g6;
add    g8, g9, g10;           # results in integer overflow

```

The fault on the **add** is detected before the fault on the **mul** because the **mul** takes longer to execute. The fault call synchronizes faults on the way to the overflow fault handler for the **add** instruction (Section 9.9.5, “Controlling Fault Precision” on page 9-19), which is when the **mul** 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 i960 RM/RN I/O processor, the **mul** 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 **mul** 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 9-7. 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 i960 RM/RN 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 **mul** instruction.



## 9.6.5 Override Faults

The i960 RM/RN 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 7-17](#) 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 11-8.

## 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 9-20 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 9-20](#).

### 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 9-12](#), 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 6-78](#).

### 9.7.5 Returning to a Point in the Program Other Than Where the Fault Occurred

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 9-20.

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 9-18.

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 9-20. 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 <b>fmark</b>
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 9-12](#).

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<sub>2</sub>) 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 9-25](#), 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 9-19](#) 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 i960 RM/RN 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.

<b>Fault Type:</b>	Gives the number that appears in the fault record fault-type field when the fault is generated.
<b>Fault Subtype:</b>	Lists the fault subtypes and the number associated with each fault subtype.
<b>Function:</b>	Describes the purpose and handling of the fault type and each subtype.
<b>RIP:</b>	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 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.
<b>Fault IP:</b>	Describes the contents of the fault record’s fault instruction pointer field, typically the faulting instruction’s IP.
<b>Fault Data:</b>	Describes any values stored in the fault record’s fault data field.
<b>Class:</b>	Indicates if a fault is precise or imprecise.
<b>Program State Changes:</b>	Describes the process state changes that would prevent re-executing the faulting instruction if applicable.
<b>Trace Reporting:</b>	Relates whether a trace fault (other than PRERET) can be detected on the faulting instruction, also if and when the fault is serviced.
<b>Notes:</b>	Additional information specific to particular implementations of the i960 architecture.

## 9.10.1 ARITHMETIC Faults

**Fault Type:** 3H

<b>Fault Subtype:</b>	<b>Number</b>	<b>Name</b>
	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:

<b>addi</b>	<b>subi</b>	<b>stis</b>
<b>stib</b>	<b>shli</b>	<b>ADDI&lt;cc&gt;</b>
<b>mul</b>	<b>divi</b>	<b>SUBI&lt;cc&gt;</b>

An ARITHMETIC.ZERO\_DIVIDE fault is generated when the divisor operand of an ordinal- or integer-divide instruction is zero. Instructions that generate this fault are:

<b>divo</b>	<b>divi</b>
<b>ediv</b>	<b>remi</b>
<b>remo</b>	<b>modi</b>

**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

<b>Fault Type:</b>	5H	
<b>Fault Subtype:</b>	<b>Number</b>	<b>Name</b>
	0H	Reserved
	1H	RANGE
	2H-FH	Reserved
<b>Function:</b>	Indicates the program or procedure violated an architectural constraint.	
	A CONSTRAINT.RANGE fault is generated when a <b>FAULT&lt;cc&gt;</b> instruction is executed and the AC register condition code field matches the condition required by the instruction.	
<b>RIP:</b>	No defined value.	
<b>Fault IP:</b>	Faulting instruction.	
<b>Class:</b>	Imprecise.	
<b>Program State Changes:</b>	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 <b>FAULT&lt;cc&gt;</b> instruction executes. The program state is not affected.	
<b>Trace Reporting:</b>	Serviced upon return from the Constraint fault handler.	

### 9.10.3 OPERATION Faults

<b>Fault Type:</b>	2H	
<b>Fault Subtype:</b>	<b>Number</b>	<b>Name</b>
	0H	Reserved
	1H	INVALID_OPCODE
	2H	UNIMPLEMENTED
	3H	UNALIGNED
	4H	INVALID_OPERAND
	5H - FH	Reserved
<b>Function:</b>	<p>Indicates the processor cannot execute the current instruction because of invalid instruction syntax or operand semantics.</p> <p>An INVALID_OPCODE fault is generated when the processor attempts to execute an instruction containing an undefined opcode or addressing mode.</p> <p>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.</p> <p>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.</p> <p>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-defined <b>sysctl</b>, <b>icctl</b>, <b>dcctl</b> or <b>intctl</b> 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).</p>	
<b>RIP:</b>	No defined value.	
<b>Fault IP:</b>	Address of the faulting instruction.	
<b>Fault Data:</b>	<p>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.</p>	
<b>Class:</b>	Imprecise.	
<b>Program State Changes:</b>	<p>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.</p>	
<b>Trace Reporting:</b>	<p>OPERATION.UNALIGNED fault: the trace is reported upon return from the OPERATION fault handler.</p> <p>All other subtypes: the trace event is lost.</p>	
<b>Notes:</b>	<p>OPERATION.UNALIGNED fault is not implemented on i960 Kx and Sx CPUs.</p>	

## 9.10.4 OVERRIDE Faults

<b>Fault Type:</b>	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.
<b>Fault Subtype:</b>	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.
<b>Fault OType:</b>	The fault type of the additional fault detected while attempting to deliver the program fault.
<b>Fault OSubtype:</b>	The fault subtype of the additional fault detected while attempting to deliver the program fault.
<b>Function:</b>	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.
<b>Trace Reporting:</b>	Same behavior as if the override condition had not existed. Refer to the description of the original program fault.

## 9.10.5 PARALLEL Faults

<b>Fault Type:</b>	Fault table entry = 0H Fault type in fault record = fault type of one of the parallel faults.
<b>Fault Subtype:</b>	Fault subtype of one of the parallel faults.
<b>Fault OType:</b>	0H
<b>Fault OSubtype:</b>	Number of parallel faults.
<b>Function:</b>	See <a href="#">Section 9.6.4, “Parallel Faults”</a> on <a href="#">page 9-9</a> 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 <a href="#">Figure 9-3</a> . 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.
<b>RIP:</b>	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.
<b>Fault IP:</b>	IP of one of the faulting instructions.
<b>Class:</b>	Imprecise.
<b>Program State Changes:</b>	State changes associated with all the parallel faults.
<b>Trace Reporting:</b>	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

<b>Fault Type:</b>	7H	
<b>Fault Subtype:</b>	<b>Number</b>	<b>Name</b>
	Bit 0	Reserved
	Bit 1	LENGTH
	Bit 2-7	Reserved
<b>Function:</b>	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 <b>calls</b> instruction, points to an entry beyond the extent of the system procedure table.	
<b>RIP:</b>	IP of the faulting instruction.	
	IP of the faulting instruction.	
<b>Fault IP:</b>	LENGTH: IP of the faulting instruction.	
<b>Class:</b>	Imprecise. (PROTECTION.LENGTH is precise even though the PROTECTION fault class is imprecise.)	
<b>Program State Changes:</b>	LENGTH: The instruction does not execute.	
<b>Trace Reporting:</b>	PROTECTION.LENGTH: The trace event is lost.	



## 9.10.7 TRACE Faults

**Fault Type:** 1H

<b>Fault Subtype:</b>	<b>Number</b>	<b>Name</b>
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 the 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 <b>ret</b> .
PRERETURN	Generates a trace event prior to any <b>ret</b> 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 <b>calls</b> 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 010 <sub>2</sub> or 011 <sub>2</sub> .

**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.

A TRACE fault subtype bit is associated with each mode. Multiple fault subtypes can occur simultaneously; all trace fault conditions detected on one instruction (except prereturn) are reported in one single trace fault, with the fault subtype bit set for each subtype that occurs. The prereturn trace is always reported alone.

When a fault type other than a TRACE fault is generated during execution of an instruction that causes a trace event, the non-trace fault is handled before the trace fault. An exception is the prereturn-trace fault, which occurs before the processor detects a non-trace fault and is handled first.

Similarly, if an interrupt occurs during an instruction that causes a trace event, the interrupt is serviced before the TRACE fault is handled. Again, the TRACE.PRERETURN fault is different. Since it is generated before the instruction, it is handled before any interrupt that occurs during instruction execution.

A trace fault handler must be accessed through a system-supervisor call (it must be a supervisor procedure in the system procedure table). Local and system-local trace fault handlers are not supported by the architecture and may have unpredictable behavior. Tracing is automatically disabled when entering the trace fault handler and is restored upon return from the trace fault handler. The trace fault handler should not modify PC.te.

**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.

**Program State Changes:** All trace faults except PRERETURN are serviced after the execution of the faulting instruction. The processor returns to the instruction immediately following the instruction traced, in instruction issue order. For PRERETURN, the return is traced before it executes. The processor re-executes the return instruction after completion of the PRERETURN trace fault handler.

### 9.10.8 TYPE Faults

<b>Fault Type:</b>	AH									
<b>Fault Subtype:</b>	<b>Number</b>	<b>Name</b>								
	0H	Reserved								
	1H	MISMATCH								
	2H-FH	Reserved								
<b>Function:</b>	<p>Indicates a program or procedure attempted to perform an illegal operation on an architecture-defined data type or a typed data structure.</p> <p>A TYPE.MISMATCH fault is generated when attempts are made to:</p> <ul style="list-style-type: none"> <li>Execute a privileged (supervisor-mode only) instruction while the processor is in user mode. Privileged instructions on the i960 RM/RN I/O processor are: <table border="0" style="margin-left: 40px;"> <tr> <td><b>modpc</b></td> <td><b>intctl</b></td> </tr> <tr> <td><b>sysctl</b></td> <td><b>inten</b></td> </tr> <tr> <td><b>icctl</b></td> <td><b>intdis</b></td> </tr> <tr> <td><b>dcctl</b></td> <td></td> </tr> </table> </li> <li>Write to on-chip data RAM while the processor is in supervisor-only write mode and BCON.irp is set.</li> <li>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.</li> <li>Write to memory-mapped registers in supervisor space from user mode.</li> <li>Write to timer registers while in user mode, when timer registers are protected against user-mode writes.</li> </ul>		<b>modpc</b>	<b>intctl</b>	<b>sysctl</b>	<b>inten</b>	<b>icctl</b>	<b>intdis</b>	<b>dcctl</b>	
<b>modpc</b>	<b>intctl</b>									
<b>sysctl</b>	<b>inten</b>									
<b>icctl</b>	<b>intdis</b>									
<b>dcctl</b>										
<b>RIP:</b>	No defined value.									
<b>Fault IP:</b>	IP of the faulting instruction.									
<b>Class:</b>	Imprecise.									
<b>Program State Changes:</b>	The fault happens before execution of the instruction. Machine state is not changed.									
<b>Trace Reporting:</b>	The trace event is lost.									



This chapter describes the i960® RM/RN I/O processor's facilities for runtime activity monitoring. The 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 i960 RM/RN 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.

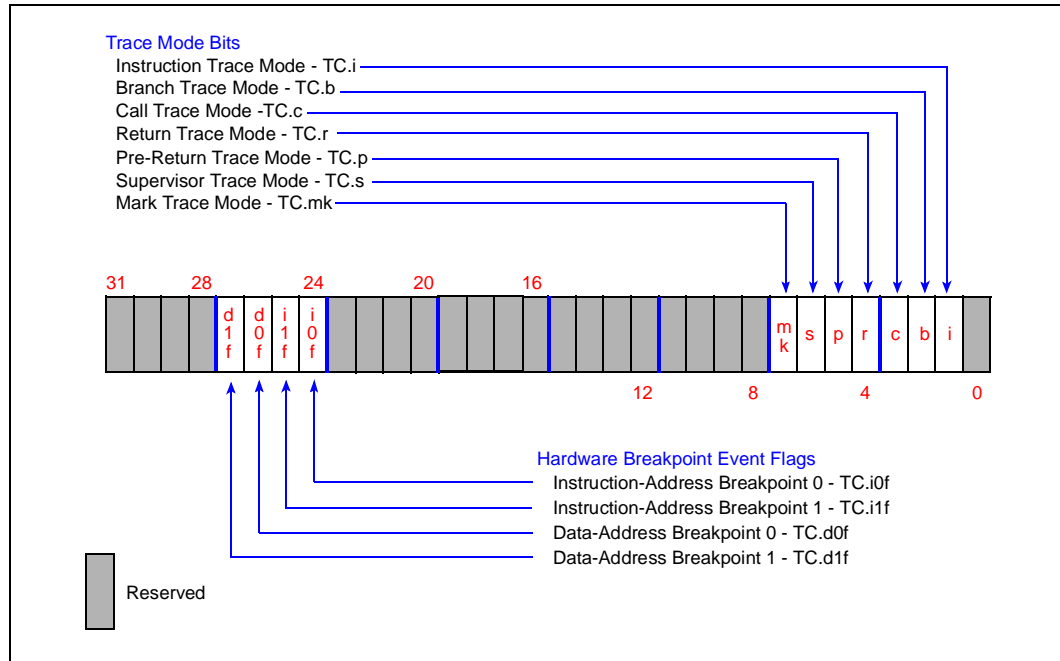
- TC register mode bits
- DAB0-DAB1 registers' address field and enable bit (in the control table)
- System procedure table supervisor-stack-pointer field trace control bit
- IPB0-IPB1 registers' address field (in the control table)
- PC register trace enable bit
- PFP register return status field prereturn 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. 80960RM/RN 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 i960 RM/RN 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 10-12](#).

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 10-14](#)).

## 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 10-11](#) 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 10-12](#) 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 10-13](#) 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 10-14](#).

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 10-14](#) and [Section 10.5.2.5, “Tracing on Return from Implicit Call: Interrupt Case” on page 10-14](#) 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<sub>2</sub> or 011<sub>2</sub> (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 i960 RM/RN 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* the memory access has been issued to the bus controller.

Hardware breakpoint registers may be armed or disarmed. When the registers are armed, hardware breakpoints can generate an architectural trace fault. When the registers are disarmed, no action occurs, and execution continues normally. Since instructions are always word aligned, the two low-order bits of the IPBx registers act as control bits. Control bits for the DABx registers reside in the Breakpoint Control (BPCON) register. BPCON enables the data address breakpoint registers, and sets the 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 3-5 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

<i>src/dst</i> 7:4	<i>src/dst</i> 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.

```
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 i960 RM/RN 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

Bit	Default	Description
31:24	00H	Reserved. Initialize to 0.
23	0 <sub>2</sub>	DAB1 Breakpoint Mode Control Bit: DAB1.m1
22	0 <sub>2</sub>	DAB1 Breakpoint Mode Control Bit: DAB1.m0
21	0 <sub>2</sub>	DAB1 Breakpoint Enable Control Bit: DAB1.e1
20	0 <sub>2</sub>	DAB1 Breakpoint Enable Control Bit: DAB1.e0
19	0 <sub>2</sub>	DAB0 Breakpoint Mode Control Bit: DAB0.m1
18	0 <sub>2</sub>	DAB0 Breakpoint Mode Control Bit: DAB0.m0
17	0 <sub>2</sub>	DAB0 Breakpoint Enable Control Bit: DAB0.e1
16	0 <sub>2</sub>	DAB0 Breakpoint Enable Control Bit: DAB0.e0
15:00	0000H	Reserved. Initialize to 0.

Programming the BPCON register is summarized in [Table 10-4](#) and [Table 10-5](#).

**Table 10-4. Configuring the 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 the 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:

```
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**

Bit	Default	Description
31:00	0000 0000H	Data Address.

<b>LBA:</b> Ch 0-8420H Ch 1-8424H <b>PCI:</b> NA	<b>Legend:</b> RV = Reserved RS = Read/Set LBA = 80960 local bus address	NA = Not Accessible PR = Preserved RC = Read Clear PCI = PCI Configuration Address Offset	RO = Read Only RW = Read/Write
--	---	--	-----------------------------------

### 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	31	28	24	20	16	12	8	4	0
	rw rw								
PCI	na na								
<b>LBA:</b>	Ch 0-8400H		<b>Legend:</b>		NA = Not Accessible		RO = Read Only		
	Ch 1-8404H		RV = Reserved		PR = Preserved		RW = Read/Write		
<b>PCI:</b>	NA		RS = Read/Set		RC = Read Clear		LBA = 80960 local bus address    PCI = PCI Configuration Address Offset		
<b>Bit</b>	<b>Default</b>	<b>Description</b>							
31:02	0000 0000H	Instruction Address.							
01	0 <sub>2</sub>	IPBX Mode: IPB1							
00	0 <sub>2</sub>	IPBX Mode: IPB0							

Programming the instruction breakpoint register modes is shown in Table 10-8.

On the i960 RM/RN 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 9-12](#) 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 9-12](#) 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 10-13](#) and [Section 10.5.2.5, “Tracing on Return from Implicit Call: Interrupt Case” on page 10-14](#) 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 10-14](#)). 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
<b>call, callx</b>	PC.te	user or supervisor	000 <sub>2</sub>	PC.te
<b>calls</b>	PC.te	supervisor	000 <sub>2</sub>	PC.te
<b>calls</b>	PC.te	user	01t <sub>2</sub> Stores PC.te into bit 0 of PFP.rt2:0	SSP.te

Refer to [Table 7-2 “Encoding of Return Status Field” on page 7-18](#)).



### 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 <sup>1</sup>	N.A.	001	a <sup>2</sup>	a	a
10-Fault <sup>1</sup>	00	001	a	a	a
10-Fault <sup>1</sup>	10	001	a	SSP.te	SSP.te
00-Parallel/Override Fault 00-Trace Fault	x <sup>2</sup>	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 i960<sup>®</sup> RM/RN 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 i960 RM/RN 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**

PFPRt2:0	Execution Mode PC.em	Trace Enable Used for Trace on Return
000 <sub>2</sub>	user or supervisor	PC.te
01a <sub>2</sub>	user	PC.te
01a <sub>2</sub>	supervisor	t <sub>2</sub> (from PFPRt2:0)

Refer to Table 7-2 "Encoding of Return Status Field" on page 7-18.

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 i960 RM/RN 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

This chapter describes the steps that the i960® RM/RN 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 i960 RM/RN I/O processor initialization can basically be separated into two steps: initialization of the 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 i960 RM/RN I/O processor's 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 i960 RM/RN 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 i960 RM/RN 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 i960 RM/RN I/O processor also supports JTAG boundary scan ([Chapter 23, “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 i960<sup>®</sup> RM/RN I/O Processor Initialization

Several functional units within the i960 RM/RN 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 i960 RM/RN 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	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”](#). 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 i960 RM/RN 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 i960 RM/RN 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 i960 RM/RN 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 80960 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

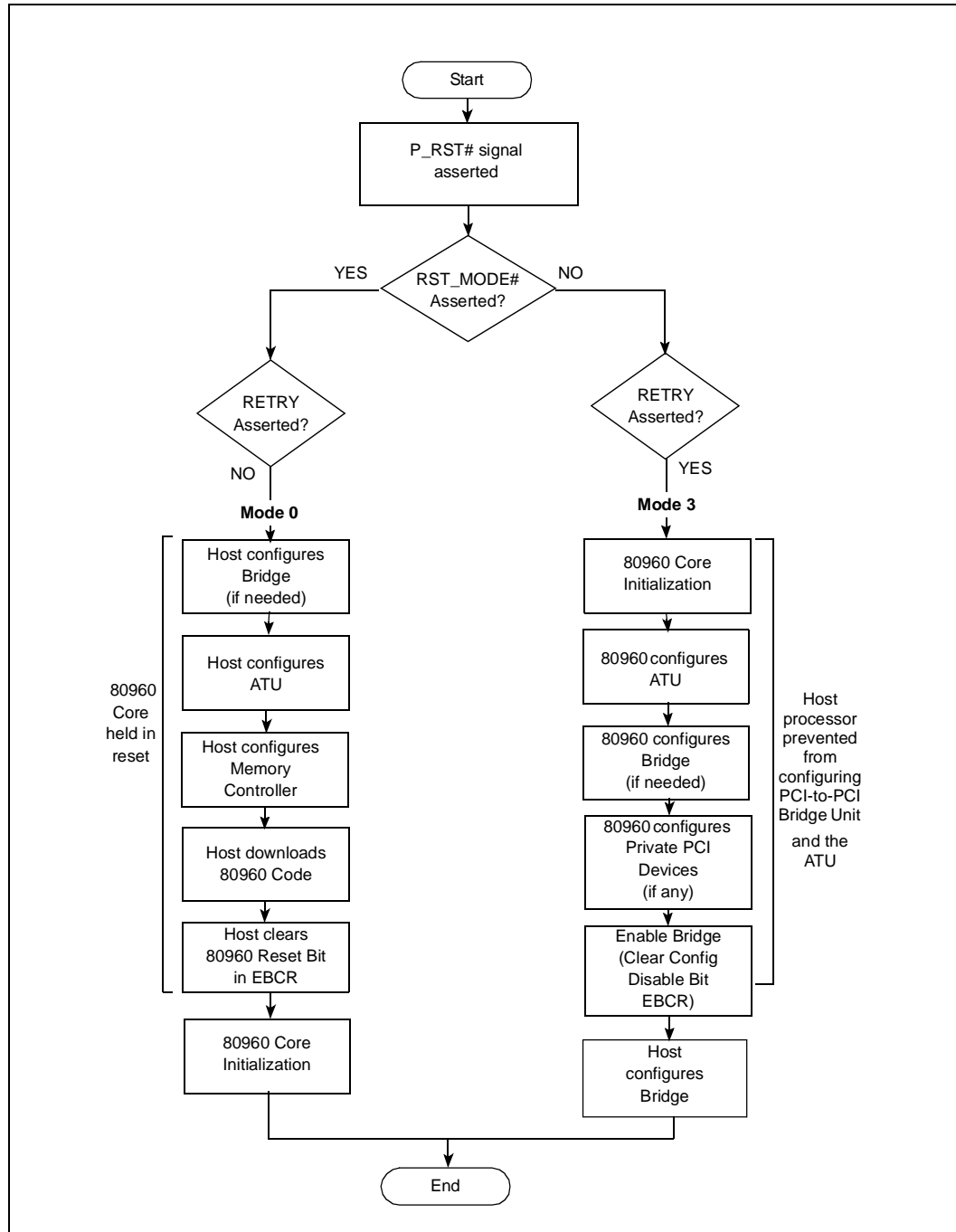
Intel does not recommend the use of Mode 2 initialization.

## 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 i960 RM/RN 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 i960 RM/RN 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 80960RM/RN, 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 i960 RM/RN 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<sub>CC</sub> 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*.

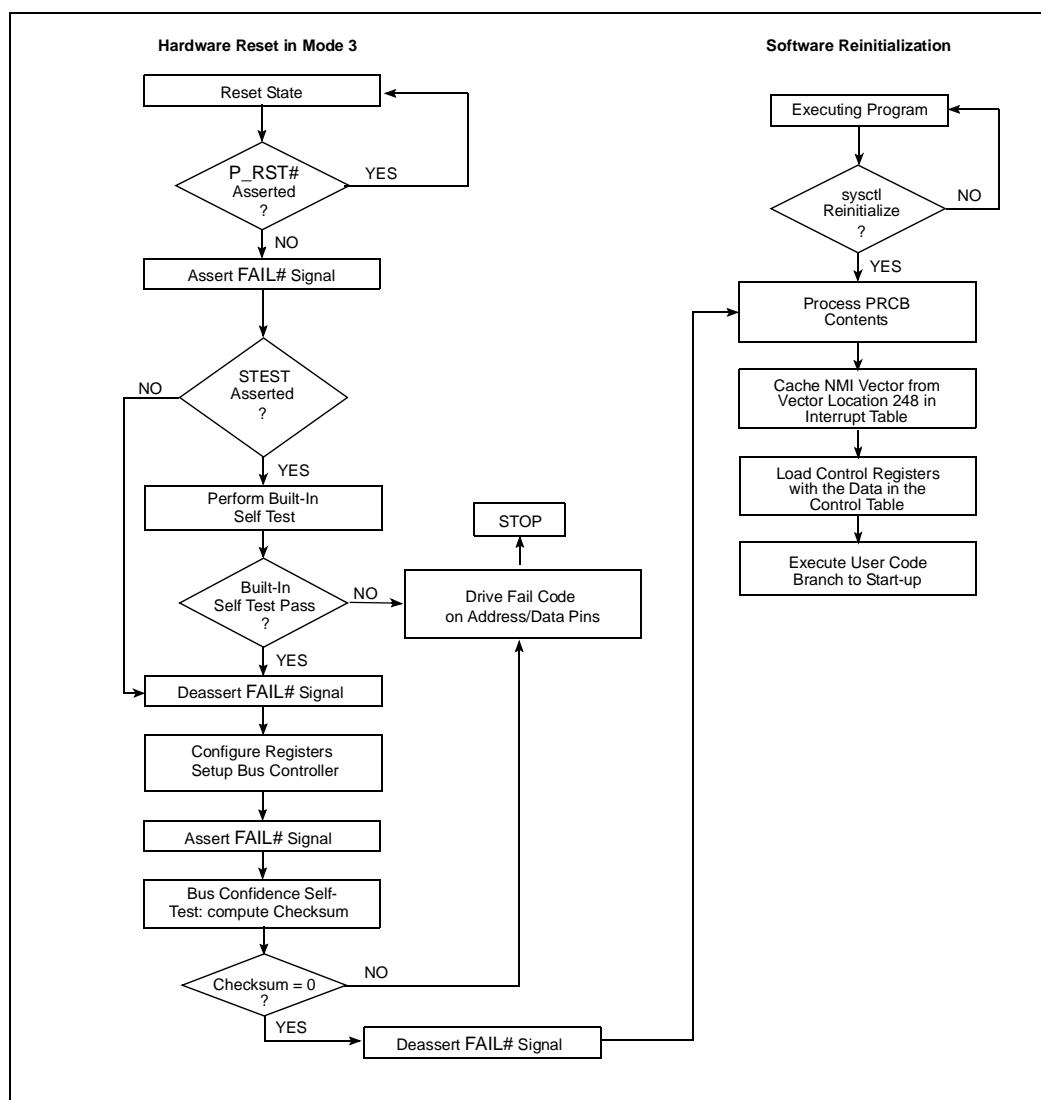
User software cannot reset the entire i960 RM/RN I/O processor; however, the **sysctl** instruction can reset the i960 core processor. The P\_RST# signal must be asserted to enter the reset state. See Section 11.6, “Reinitializing and Relocating Data Structures” on page 11-20.

## 11.3 i960<sup>®</sup> 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 i960 RM/RN 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 80960RM/RN 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 an 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 80960 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.

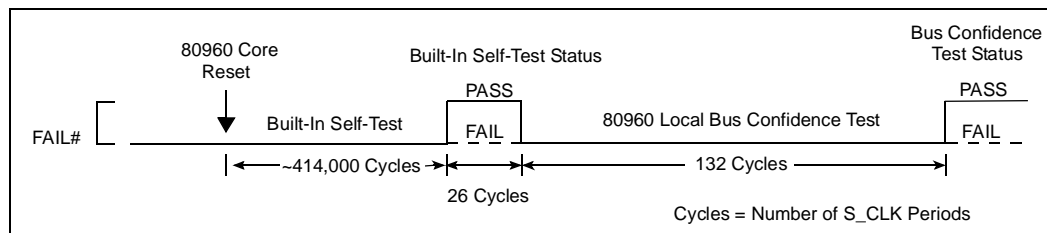
#### 11.3.1.3 The Fail Signal (FAIL#)

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 80960RM I/O Processor Data Sheet and the 80960RN I/O Processor Data Sheet.

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 FFFF 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 11-7](#).

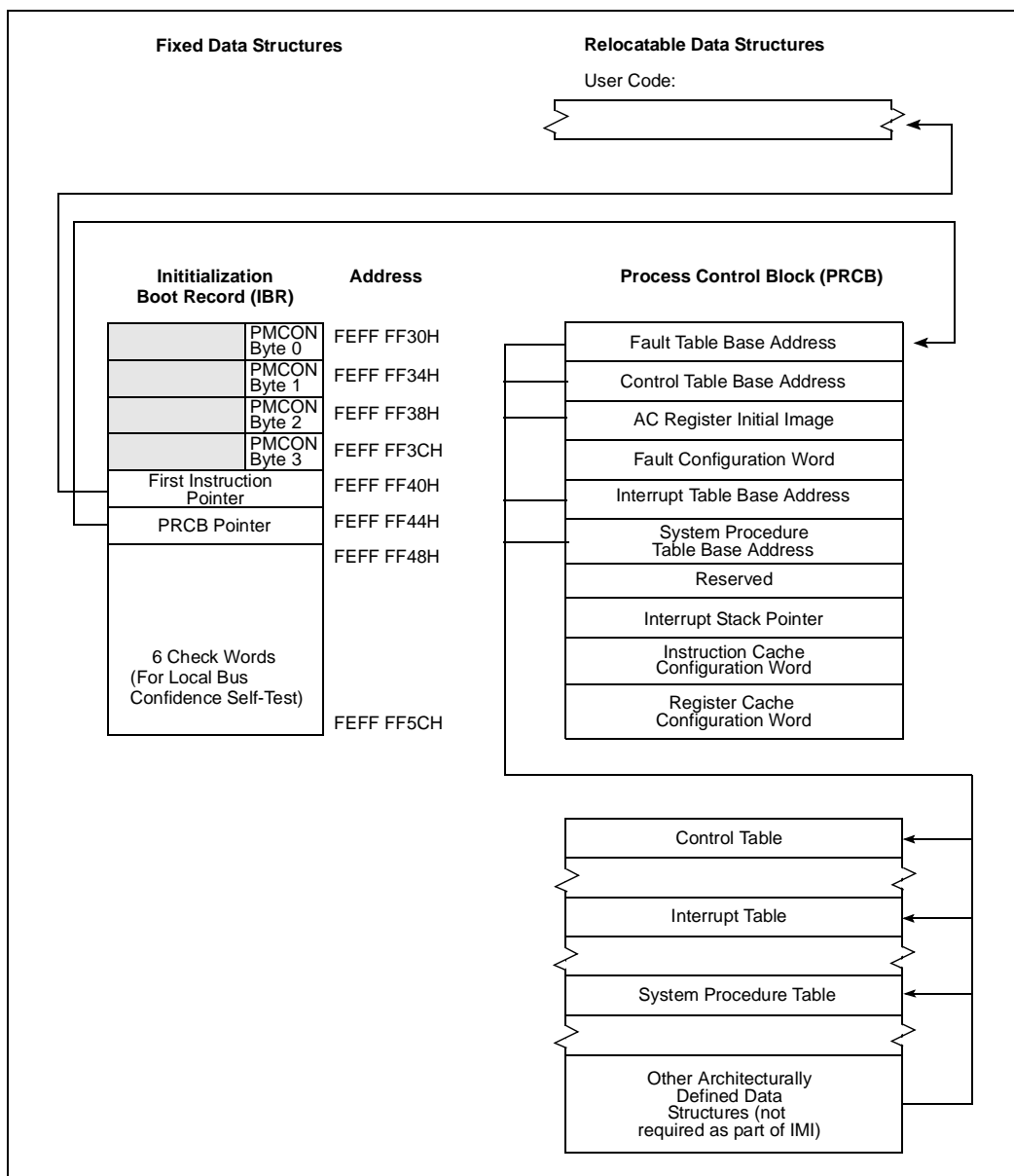
At initialization, the processor loads the Supervisor Stack Pointer (SSP) from the system procedure table, aligns it to a 16-byte boundary, and caches the pointer in the SSP memory-mapped control register. Recall that the supervisor stack pointer is located in the preamble of the system procedure table 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 7-14](#) 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 i960 RM/RN 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 i960 RM/RN I/O processor. The IBR is a 12-word structure which must be located at address FFFF 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
FFFF FF30H	PMCON14_15, byte 0 (Program to 0000 0000H) for 80960RM/RN processor
FFFF FF31H to FFFF FF33H	Reserved
FFFF FF34H	PMCON14_15, byte 1 (Program to 0000 0000H) for 80960RM/RN processor
FFFF FF35H to FFFF FF37H	Reserved
FFFF FF38H	PMCON14_15, byte 2 (Program to 0000 0080H) for 80960RM/RN processor
FFFF FF39H to FFFF FF3BH	Reserved
FFFF FF3CH	PMCON14_15, byte 3 (Program to 0000 0000H) for 80960RM/RN processor
FFFF FF3DH to FFFF FF3FH	Reserved
FFFF FF40H to FFFF FF43H	First Instruction Pointer
FFFF FF44H to FFFF FF47H	PRCB Pointer
FFFF FF48H to FFFF FF4BH	Bus Confidence Self-Test Check Word 0
FFFF FF4CH to FFFF FF4FH	Bus Confidence Self-Test Check Word 1
FFFF FF50H to FFFF FF53H	Bus Confidence Self-Test Check Word 2
FFFF FF54H to FFFF FF57H	Bus Confidence Self-Test Check Word 3
FFFF FF58H to FFFF FF5BH	Bus Confidence Self-Test Check Word 4
FFFF FF5CH to FFFF 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 endianness 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 IBR's first 4 words are used to form the register values. On the i960 RM/RN I/O processor, the bytes at FFFF FF30H and FFFF FF34H are not needed, so the processor starts fetching at address FFFF FF38. The loading of these registers is shown in the pseudo-code flow in Example 11-1.

**Note:** The 80960RM/RN 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;          IMASK = 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 = 80960core_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	31	28	24	20	16	12	8	4	0	
	rv	rv	rv	rv	rv	rv	rv	rv	rv	
PCI	na	na	na	na	na	na	na	na	na	
<b>ADD:</b>	8638H									
<b>PCI:</b>	NA									
<b>Legend:</b>		NA = Not Accessible			RO = Read Only					
		RV = Reserved			PR = Preserved			RW = Read/Write		
		RS = Read/Set			RC = Read Clear					
		ADD = 80960 internal bus address			PCI = PCI Configuration Address Offset					
<b>Bit</b>	<b>Default</b>	<b>Description</b>								
31:24	00H	Reserved. Initialize to 0.								
23:22	00 <sub>2</sub>	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](#).

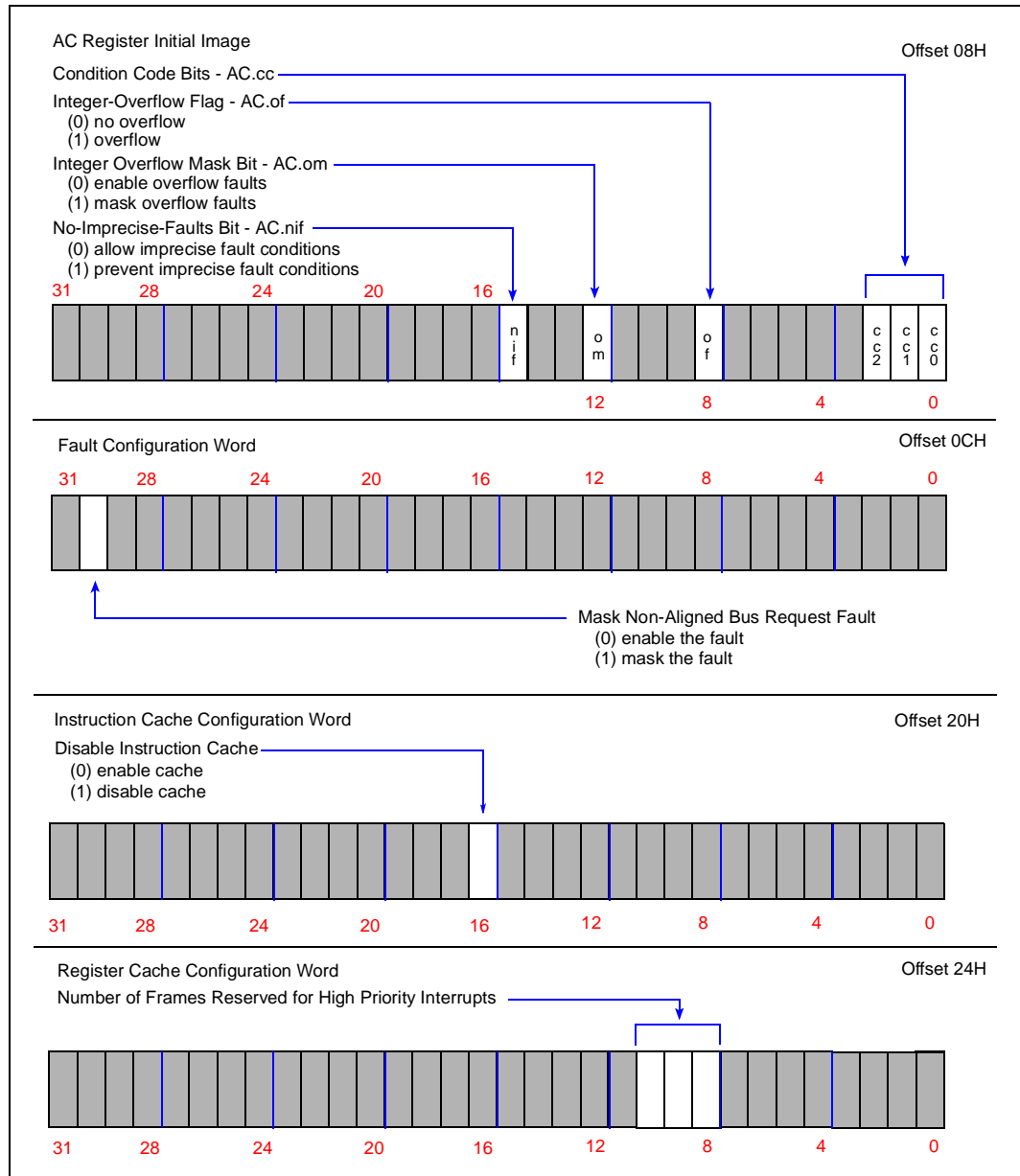
**Table 11-7. PRCB Configuration**

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 8-35
- ICON — Table 8-15 “Interrupt Control (ICON) Register” on page 8-33
- PMCON — Table 12-2 “Physical Memory Control Registers – PMCON0:15” on page 12-2
- TC — Table 10-1 “80960RM/RN Trace Controls Register – TC” on page 10-2
- BCON — Table 12-3 “Bus Control Register – BCON” on page 12-3

Figure 11-5. Control Table

31	0	
		Reserved (Initialize to 0) 00H
		Reserved (Initialize to 0) 04H
		Reserved (Initialize to 0) 08H
		Reserved (Initialize to 0) 0CH
		Interrupt Map 0 (IMAP0) 10H
		Interrupt Map 1 (IMAP1) 14H
		Interrupt Map 2 (IMAP2) 18H
		Interrupt Configuration (ICON) 1CH
		Physical Memory Region 0:1 Configuration (PMCON0_1) (0080 0000H) 20H
		Reserved (Initialize to 0) 24H
		Physical Memory Region 2:3 Configuration (PMCON2_3) (0080 0000H) 28H
		Reserved (Initialize to 0) 2CH
		Physical Memory Region 4:5 Configuration (PMCON4_5) (0080 0000H) 30H
		Reserved (Initialize to 0) 34H
		Physical Memory Region 6:7 Configuration (PMCON6_7) (0080 0000H) 38H
		Reserved (Initialize to 0) 3CH
		Physical Memory Region 8:9 Configuration (PMCON8_9) (0080 0000H) 40H
		Reserved (Initialize to 0) 44H
		Physical Memory Region 10:11 Configuration (PMCON10_11) (0080 0000H) 48H
		Reserved (Initialize to 0) 4CH
		Physical Memory Region 12:13 Configuration (PMCON12_13) (0080 0000H) 50H
		Reserved (Initialize to 0) 54H
		Physical Memory Region 14:15 Configuration (PMCON14_15) (0080 0000H) 58H
		Reserved (Initialize to 0) 5CH
		Reserved (Initialize to 0) 60H
		Reserved (Initialize to 0) 64H
		Trace Controls (TC) 68H
		Bus Configuration Control (BCON) 6CH

## 11.5 Device Identification on Reset

During the manufacturing process, values characterizing the i960 RM/RN I/O processor type and stepping are programmed into the memory-mapped registers. The i960 RM/RN 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	31	28	24	20	16	12	8	4	0
	ro	ro	ro	ro	ro	ro	ro	ro	ro
PCI	na	na	na	na	na	na	na	na	na
<b>ADD:</b>	1710H		<b>Legend:</b>		NA = Not Accessible	RO = Read Only			
<b>PCI:</b>	NA		RV = Reserved	PR = Preserved	RS = Read/Set	RC = Read Clear	RW = Read/Write		
			ADD = 80960 internal bus address    PCI = PCI Configuration Address Offset						
<b>Bit</b>	<b>Default</b>		<b>Description</b>						
31:0	X		The values programmed into this register vary with stepping. Refer to the <i>i960<sup>®</sup> RM/RN I/O Processor Specification Update (273164)</i> for the correct value.						

**Table 11-10. i960<sup>®</sup> Core Processor Device ID Register - DEVICEID**

ADD	31	28	24	20	16	12	8	4	0
	ro	ro	ro	ro	ro	ro	ro	ro	ro
PCI	na	na	na	na	na	na	na	na	na
<b>ADD:</b>	FF00 8710H		<b>Legend:</b>		NA = Not Accessible	RO = Read Only			
<b>PCI:</b>	NA		RV = Reserved	PR = Preserved	RS = Read/Set	RC = Read Clear	RW = Read/Write		
			ADD = 80960 internal bus address    PCI = PCI Configuration Address Offset						
<b>Bit</b>	<b>Default</b>		<b>Description</b>						
31:0	X		The values programmed into this register vary with stepping. Refer to the <i>i960<sup>®</sup> RM/RN I/O Processor Specification Update (273164)</i> 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 6-96](#) 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 11-14](#).

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 i960 RM/RN I/O processor supports an I<sup>2</sup>C bus interface. The output clock frequency for I<sup>2</sup>C operation is 100 KHz or 400 KHz. This clock is generated from the i960 core processor clock. To use the I<sup>2</sup>C interface, a clock divider value must be written into the I<sup>2</sup>C Clock Count Register. See [Section 22.8.5, “I<sup>2</sup>C Clock Count Register- ICCR” on page 22-31](#).

# Core Processor and Internal Operation

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 80960 core local bus width, the Bus Interface Unit (BIU), and the 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:** 80960RM/RN processors require 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 80960RM/RN processor are controlled through the PMCON registers and the BCON.

### 12.2.1 PMCON Registers

The 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 the mapping shown in [Table 12-1](#).

**Table 12-1. PMCON Address Mapping (Sheet 1 of 2)**

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



**Table 12-1. PMCON Address Mapping (Sheet 2 of 2)**

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 80960 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 11-9].

**Table 12-2. Physical Memory Control Registers – PMCON0:15**

<b>LBA:</b>	Table 12-1	<b>Legend:</b> NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset
<b>PCI:</b>	NA	
<b>Bit</b>	<b>Default</b>	<b>Description</b>
31:24	00H	Reserved. Initialize to 0.
23:22	00 <sub>2</sub>	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 2 of BCON is clear, PMCON14\_15 remains in use for all local bus accesses. When bit 2 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**

Bit	Default	Description
31:03	0000 0000H	Reserved.
02	0 <sub>2</sub>	Configuration Entries in Control Table Valid (0) = PMCON entries not valid, default to PMCON14_15 setting (1) = PMCON entries valid
01	0 <sub>2</sub>	Internal RAM Protection (0) = Internal data RAM not protected from user mode writes (1) = Internal data RAM protected from user mode write
00	0 <sub>2</sub>	Supervisor Internal RAM Protection (0) = First 64 bytes not protected from supervisor mode write (1) = First 64 bytes protected from supervisor mode writes

<p><b>LBA:</b> 86FCH</p> <p><b>PCI:</b> NA</p>	<p><b>Legend:</b> NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset</p>
--	--

## 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](#), “Memory-Mapped Registers”.)

### 12.3.1 Logical Memory Attributes

The i960 RM/RN 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.

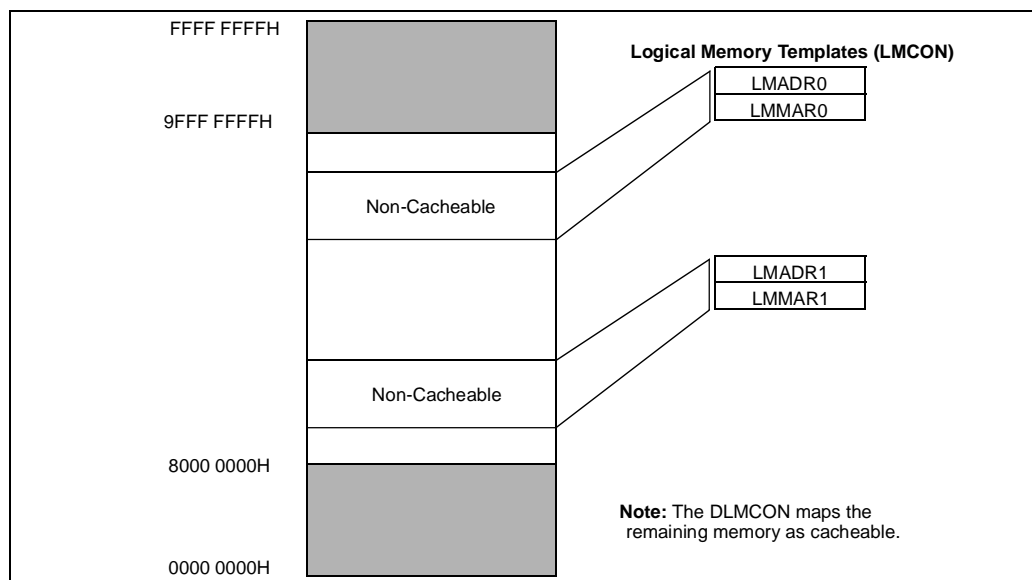
There are typically several different LMTs defined within a single memory subsystem. For example, data within one area of DRAM may be non-cacheable while data in another area is cacheable. [Figure 12-1](#) shows the use of the Control Table (PMCON registers) with logical memory templates for a single DRAM region in a typical application.

Each logical memory template is defined by programming *Logical Memory Configuration (LMCON) registers*. An LMCON register pair defines a data template for areas of memory that have common logical attributes. The i960 RM/RN 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 (LMMASK). 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 12-4.

**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**

<b>LBA:</b> CH0-8108H CH1-8110H <b>PCI:</b> NA	<b>Legend:</b> NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 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 <sub>2</sub>	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 <sub>2</sub>	Reserved.

**Table 12-5. Logical Memory Mask Registers – LMMR0:1**

<b>LBA:</b> CH0-810CH CH1-8114H <b>PCI:</b> NA	<b>Legend:</b> NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 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 <sub>2</sub>	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**

<b>LBA:</b> 8100H <b>PCI:</b> NA	<b>Legend:</b> NA = Not Accessible RO = Read Only RV = Reserved PR = Preserved RW = Read/Write RS = Read/Set RC = Read Clear LBA = 80960 Local Bus Address PCI = PCI Configuration Address Offset	
Bit	Default	Description
31:02	0000 0000H	Reserved.
01	0 <sub>2</sub>	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 <sub>2</sub>	Reserved.

### 12.3.3 Defining the Effective Range of a Logical Data Template

For each logical data template, an LMADR<sub>x</sub> 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 LMADR<sub>x</sub> register and bits 31:12 in the LMMR<sub>x</sub> register.

For each access, only those address bits in the range 31:12 marked as unmasked (defined by bits MA<sub>31:12</sub> in the LMMR<sub>x</sub> register), are compared against bits 31:12 in the LMMR<sub>x</sub> register. When all of the unmasked bits of the address match bits 31:12 of the LMMR<sub>x</sub> 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:

$$(EFA_{31:12} \text{ xnor } LMADR_{x31:12}) \text{ or } (\text{not } LMMR_{x31:12})$$

Where EFA<sub>31:12</sub> 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 LMADR<sub>x</sub>LMMR<sub>x</sub> 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

LMMR<sub>x</sub> 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 (LMADR<sub>x</sub> 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 i960 RM/RN 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.

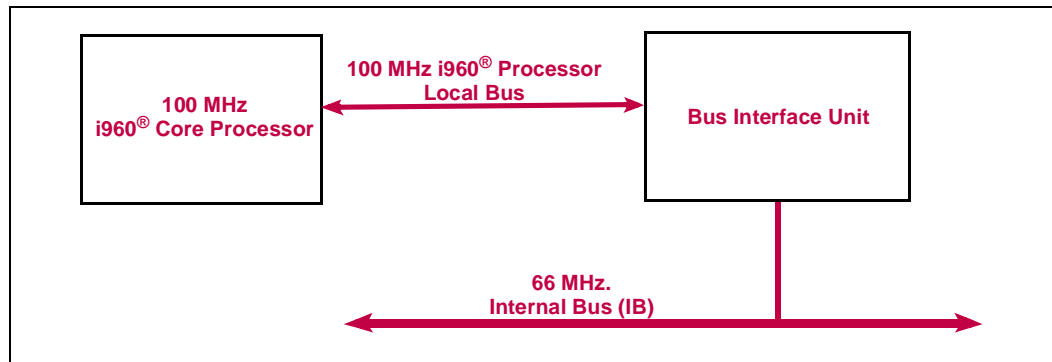
## 12.4 Bus Interface Unit

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 12-12.

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.

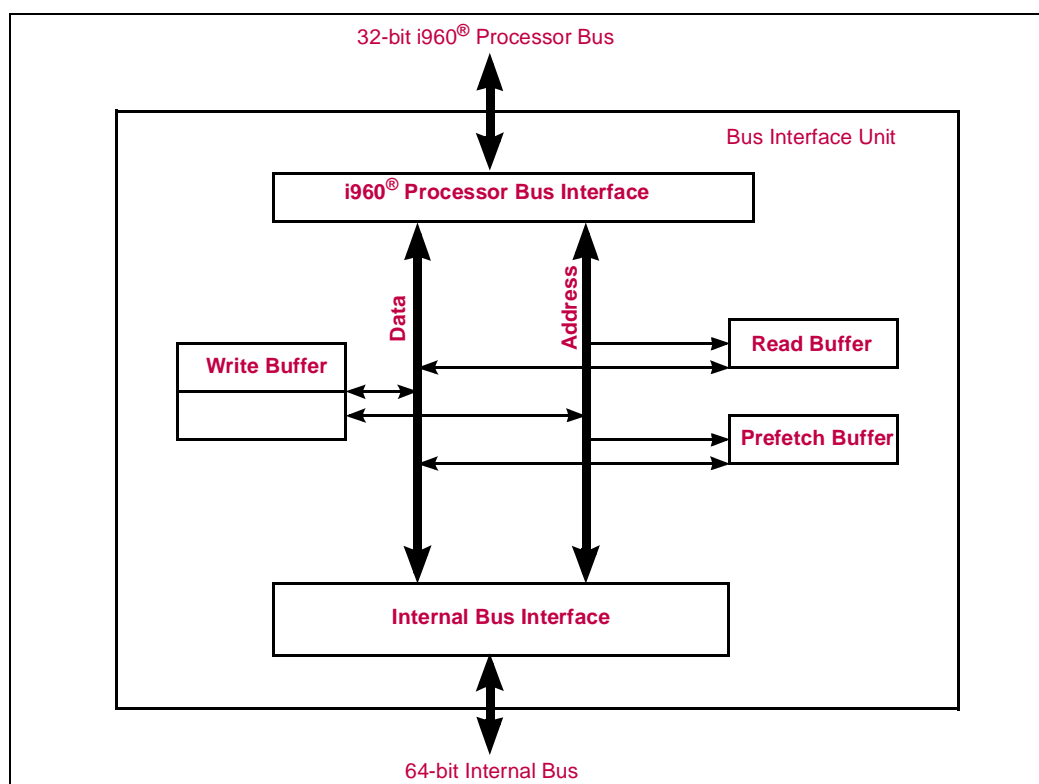
The Read Buffer temporarily stores read data for read accesses returning from the IB to the i960 core processor. The Read Buffer is one entry deep and each entry can store up to 16 bytes of data.

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<sub>2</sub> (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 support 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, “i960® RM/RN 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<sub>2</sub> for the first access and 100<sub>2</sub> 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 12-14](#) 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 sections.

### Target-Abort During BIU Write

When an Internal Bus access initiated by the BIU 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.

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 the target abort occurs on the last data cycle of a burst, the transaction is completed by the BIU before the MCU had determined there is a multi-bit ECC error. Therefore, to the BIU, the transaction appears as if it has 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 an Internal Bus access initiated by the BIU 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 an Internal Bus access initiated by the BIU 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 an Internal Bus access initiated by the BIU 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 the 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 12-16
Section 12.4.6.2, "BIU Interrupt Status Register - BIUISR" on page 12-17

### 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**

Bit	Default	Description
31:02	0000 0000H	Reserved
01	1 <sub>2</sub>	Write Merging Enable - When set, the BIU merges sequential DWORD Write accesses. When clear, the BIU does not merge any accesses.
00	1 <sub>2</sub>	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.

80960 Core Internal Address  
1640H

Attribute Legend: RW = Read/Write  
RV = Reserved RC = Read Clear  
PR = Preserved RO = Read Only  
RS = Read/Set NA = Not Accessible

### 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**

Bit	Default	Description
31:03	00000000H	Reserved
02	0 <sub>2</sub>	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 <sub>2</sub>	Reserved





This chapter describes the 80960RM/RN integrated Memory Controller Unit (MCU). The operating modes, initialization, external interfaces, and implementation are detailed in this chapter.

## 13.1 Overview

The i960<sup>®</sup> RM/RN I/O processor integrates a Memory Controller to provide a direct interface between the i960 RM/RN I/O processor and its local memory subsystem. The Memory Controller supports:

- Up to 16 Mbytes of 8-bit Flash
- Between 8 and 128 Mbytes of 64-bit Synchronous DRAM (SDRAM) or  
Between 4 and 64 Mbytes of 32-bit Synchronous DRAM for low cost solutions
- Single-bit error correction, double-bit and nibble detection support (ECC)

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 i960 RM/RN 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 66 MHz, 64-bit wide data path to support 528 Mbytes/sec throughput. 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 two-bank dual inline memory module (DIMM) or two single-bank DIMMs, or  
The MCU supports a 32-bit SDRAM data interface. This mode enables lower-cost solutions at the cost of system performance.
- 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.

## 13.1.1 Memory Controller Terminology

This section lists commonly used terms throughout this chapter:

- *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 and 1 Kbyte for 32-bit wide memory). Therefore, if an SDRAM physical configuration supports a larger page size, the MCU breaks it up into smaller 2 Kbyte or 1 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). Correcting the memory location is referred to as “scrubbing the array.” The MCU relies on software to scrub any errors.
- *Syndromes* — 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 on page 13-31](#) defines the syndrome decoding.

## 13.2 Flash Memory Support

The i960 RM/RN 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 and/or LEDs. The 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 Mbits. Refer to the timing diagrams in [Figure 13-2](#) and [Figure 13-3](#) for details about how the pins are multiplexed.

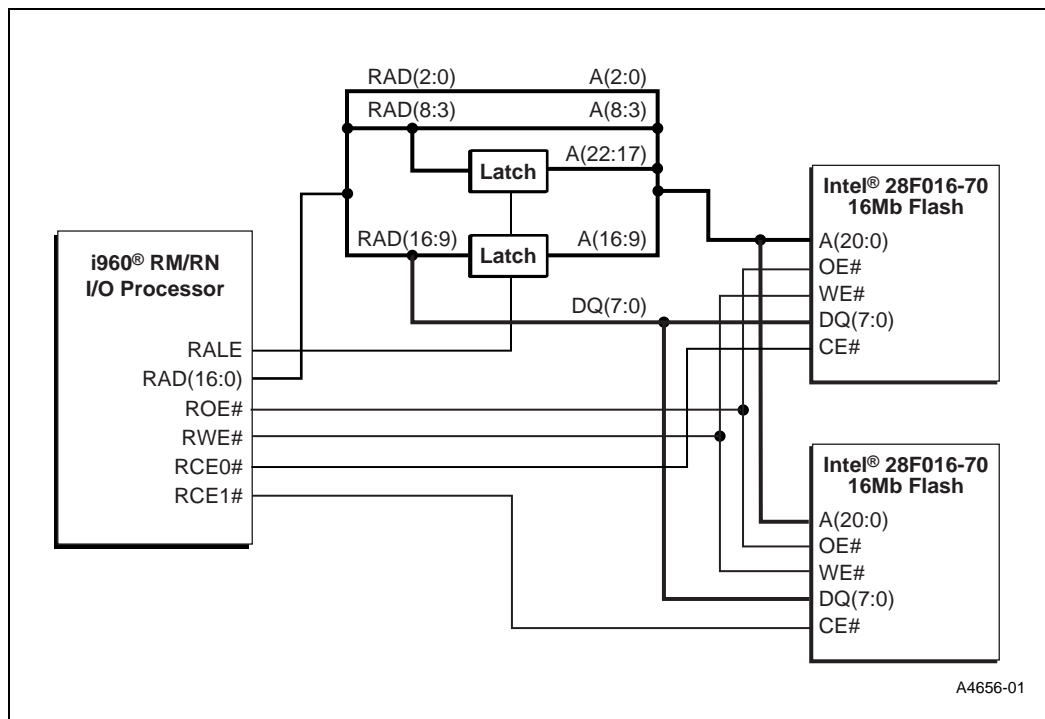
Flash memory space is separate from the SDRAM space. The Flash chip enables activate the appropriate Flash bank when the address falls within one of the Flash address ranges. [Table 13-1](#) shows the Flash interface signals.

**Table 13-1. 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 13-1 illustrates how two Flash devices would interface with the i960 RM/RN I/O processor through the MCU.

**Figure 13-1. 4 Mbyte Flash Memory System**



### 13.2.1 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 a 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 13-2](#) indicates which bits of the FEBRx are compared with the corresponding bits of L\_AD[31:0].

**Table 13-2. 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 13.5.3, “Overlapping Memory Regions”](#) on page 13-41 for prioritization details if the two Flash memory regions overlap.

## 13.2.2 Flash Read Cycle

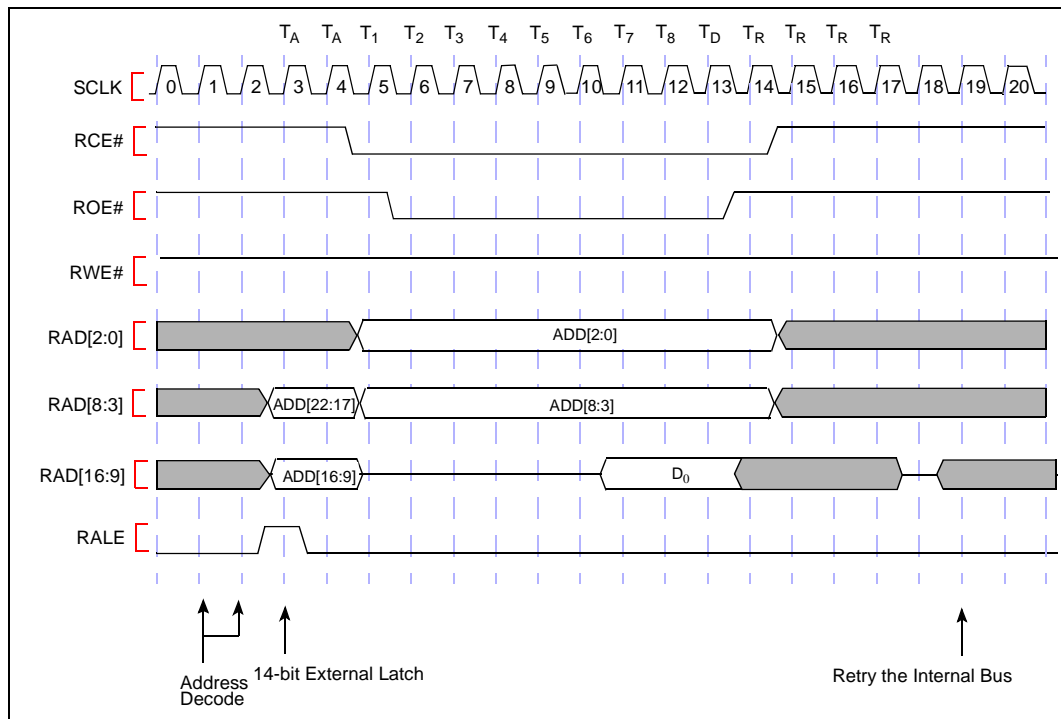
Reading a Flash device involves driving the address, output enable, and chip enable. Depending on the speed of the Flash device, the data returns several cycles later.

The definition of address-to-data wait states are the number of cycles between the assertion RCE[1:0]# 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 13-3](#) for the programmable address-to data and recovery wait states.

Figure 13-2 illustrates a read cycle from a 90 ns Flash device.

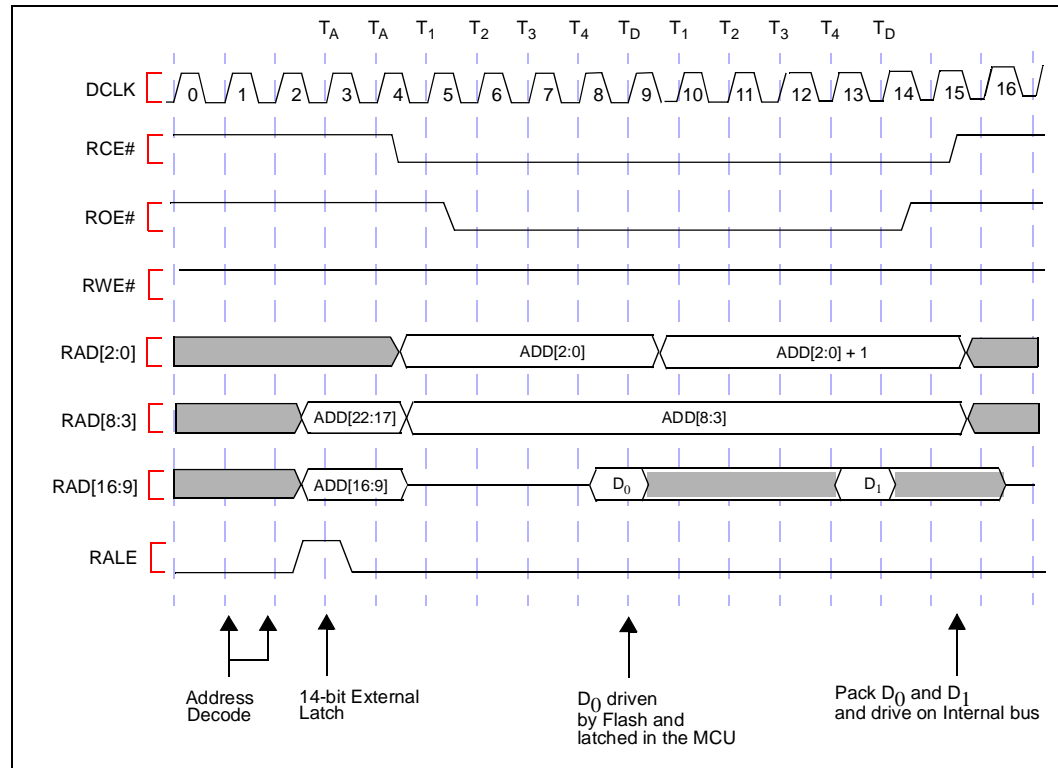
**Figure 13-2. 90 ns Flash Read Cycle**



When an internal bus master requests data from the Flash memory region, the MCU decodes the internal bus byte enables 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 internal bus master reading data always gets disconnected after the first data phase.

Figure 13-3 illustrates a bursted read cycle from a 60 ns Flash device.

**Figure 13-3. 60 ns Flash Burst Read Cycle**



Refer to Table 13-3 for the programmable address-to data wait states.

**Table 13-3. Flash Wait State Profile Programming**

Flash Speed	Address-to-Data Wait States	Recovery Wait States
$\leq 55$ ns	4	0
$\leq 115$ ns	8	4
$\leq 175$ ns	12	4

### 13.2.3 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 13-3](#) for the programmable address-to data wait states.

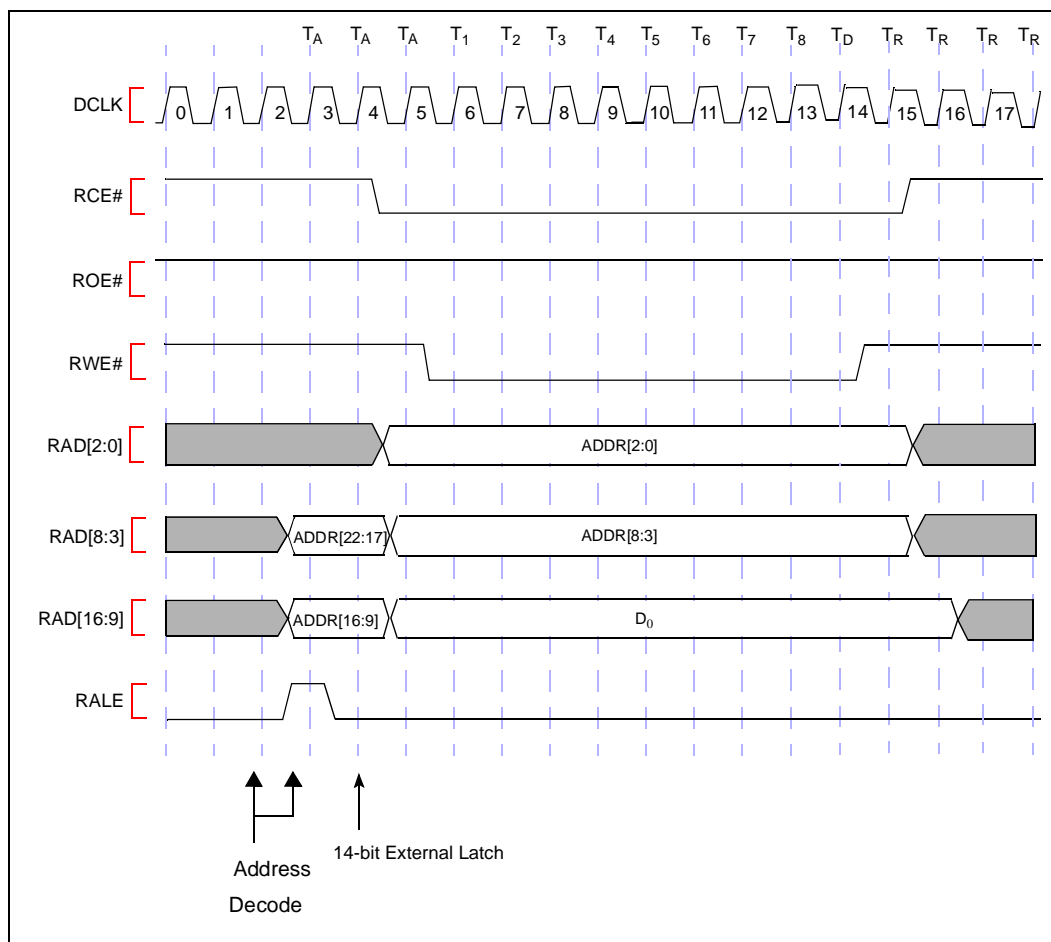
**Note:** The MCU always adds one extra address-to-data wait state for write operations. This extra wait state accommodates for the fact that read time is initiated with RCE# while write time is initiated with RWE#. RWE# is asserted one cycle after RCE#.

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 time between writes is typically 6 ms. The core is the only device permitted to write to the Flash device and the software must ensure that only a single byte is written.

Figure 13-4 illustrates a write cycle to a 90 ns Flash device.

Figure 13-4. 90 ns Flash Write Cycle





## 13.3 SDRAM Memory Support

The i960 RM/RN I/O processor memory controller supports one or two banks of SDRAM. SDRAM allows zero data-to-data wait-state operation at 66 MHz. It also offers an extremely wide range of configuration options emerging from the SDRAM's 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. The page size depends on the data bus width indicated in the SDCR register. Page boundaries are naturally aligned 2 Kbyte blocks for a 64-bit data bus and 1 Kbyte blocks for a 32-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 as shown in [Table 13-5](#). [Table 13-5](#) shows the SDRAM interface signals.

**Table 13-4. SDRAM Memory Configuration Options**

Data Bus Width	ECC Enabled	Maximum Throughput
64 bit	Yes	528 Mbyte/s
64 bit	No	528 Mbyte/s
32 bit	Yes	256 Mbyte/s
32 bit	No	256 Mbyte/s

**Table 13-5. SDRAM Interface Signals**

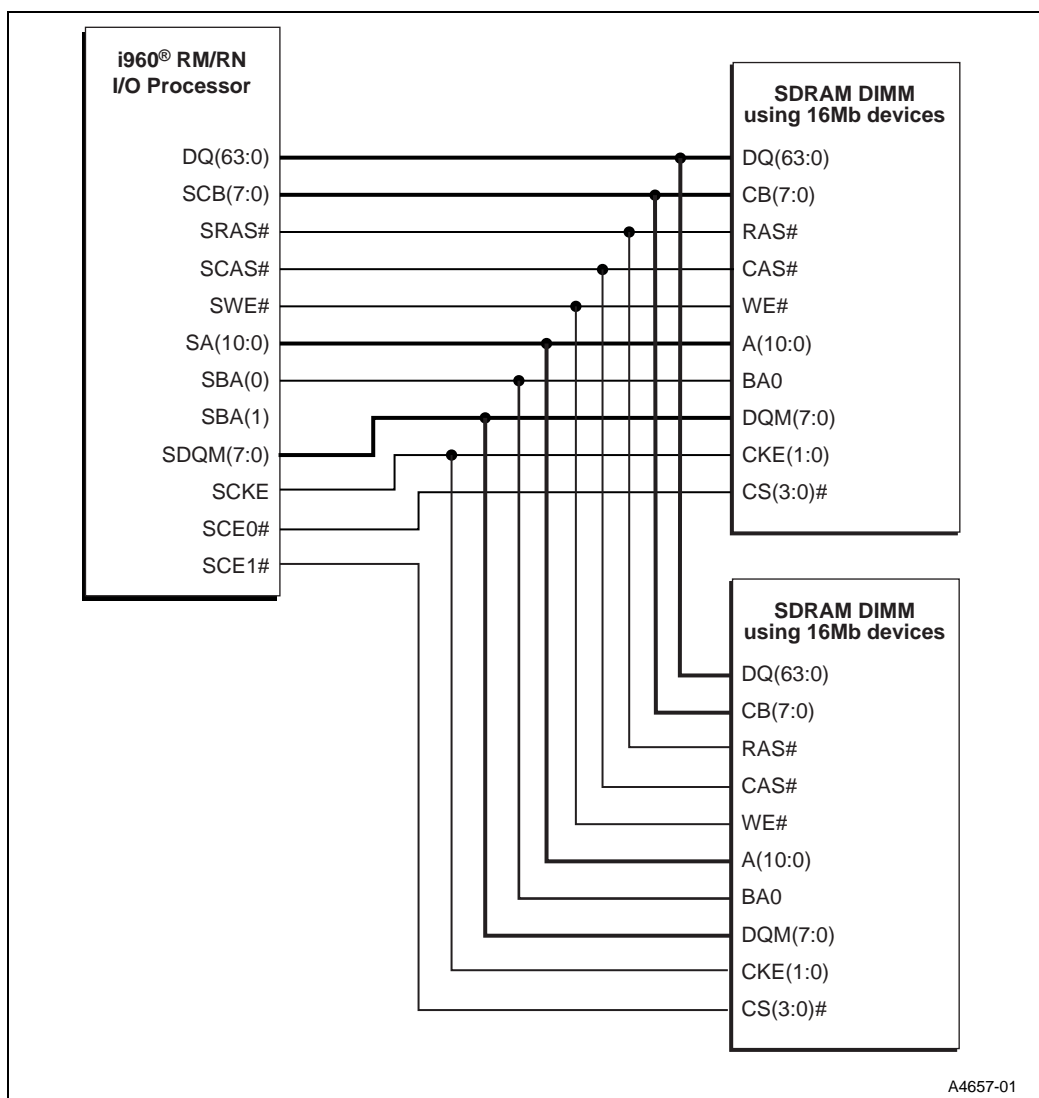
Pin Name	Description
DCLKOUT	<i>SDRAM Clock Out</i> - This is the clock to the off-chip SDRAM clock buffer driven by the i960 RM/RN I/O processor. <a href="#">Section 13.3.8, "SDRAM Clocking" on page 13-33</a> describes the SDRAM clocking strategy.
DCLKIN	<i>SDRAM Clock In</i> - This is the clock returning from the off-chip SDRAM clock buffer. <a href="#">Section 13.3.8, "SDRAM Clocking" on page 13-33</a> describes the SDRAM clocking strategy.
SCKE[1:0]	<i>Clock enables</i> - 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]	<i>Data Mask</i> - On a write, these signals disable the data on a byte-by-byte basis thus preventing certain bytes from being written. On a read, two clocks after asserting SDQM[7:0] the output data bytes are disabled.
SCE[1:0]#	<i>Chip Select</i> - Must be asserted for all transactions to the SDRAM device. One per bank.
SWE#	<i>Write Enable</i> - 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]	<i>SDRAM Bank Selects</i> - 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]	<i>Address bit 10</i> - If high during a read or write command, auto-precharge occurs after the command. During a <b>row-activate</b> command, this bit is part of the address ( <a href="#">Table 13-6</a> ).
SA[11:0]	<i>Address bits 11 through 0</i> - Indicates the row or column to access depending on the state of SRAS# and SCAS# ( <a href="#">Table 13-6</a> ).
SRAS#	<i>Row Address Strobe</i> - Indicates that the current address on SA[11:0] is the row.
SCAS#	<i>Column Address Strobe</i> - Indicates that the current address on SA[11:0] is the column.
DQ[63:0]	<i>Data Bus</i> - 64-bit wide data bus.
SCB[7:0]	<i>ECC Bus</i> - 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 Mbit devices. The number of available pages depends on the memory subsystem population. A single 64 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 i960 RM/RN I/O processor's system environment. The MCU's paging algorithm is detailed in [Section 13.3.4, "Page Hit/Miss Determination" on page 13-14](#). The waveforms illustrating the performance issues are in [Section 13.3.6.2, "SDRAM Read Cycle" on page 13-20](#) and [Section 13.3.6.3, "SDRAM Write Cycle" on page 13-24](#).

Figure 13-5 illustrates how two banks of SDRAM would interface with the i960 RM/RN I/O processor through the MCU.

**Figure 13-5. Dual-Bank SDRAM Memory Subsystem**



### 13.3.1 SDRAM Sizes and Configurations

The MCU supports a memory subsystem ranging from 16 to 128 Mbytes. If the memory subsystem supports ECC, there is a limitation on the SDRAM devices used. Since an ECC-supported system needs to be 72 bits wide, x16 devices are not optimal. A non-ECC system may be implemented using x8, or x16 devices. This allows flexibility and offers between 8 and 128 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]	
16 Mbit	2M x 8	1	11	9	-	I_AD[23]	16M
		2					32M
	1M x 16	1	11	8	-	I_AD[22]	8M
		2					16M
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

16 Mbit devices comprise two internal leaves. The MCU controls the leaf select within 16 Mbit SDRAM by toggling SBA[0]. 64 Mbit SDRAM devices comprise four internal leaves. The MCU controls the leaf selects within 64 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 4 Mbyte 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 4 Mbyte boundary.
SDRAM Boundary Register 0 (SBR0)	The upper address for bank 0 of SDRAM memory space. Also, the lower address for bank 1 of SDRAM memory space.
SDRAM Boundary Register 1 (SBR1)	The upper address 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 4 Mbyte boundary and must *never* cross a 128 Mbyte boundary.

The base register defines the upper ten address bits of the SDRAM memory space. The boundary registers define the address limits for each SDRAM bank in 4 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
$I\_AD[31:27]$ is not equal to the SDBR	None
$I\_AD[31:22]$ is greater than or equal to the SDBR $I\_AD[26:22]$ is less than the value in SBR0	Bank 0
$I\_AD[31:22]$ is greater than or equal to the SDBR $I\_AD[26:22]$ is greater or equal than the value in SBR0 $I\_AD[26:22]$ is less than the value in SBR1	Bank 1

**Address Register Programming Examples**

**Example 13-1.** The user wants to program the SDRAM memory space to begin at B000 0000H. Bank 0 is 4 Mbytes and Bank 1 is 8 Mbytes yielding in a total memory of 12 Mbytes. The registers would be programmed as follows:

SDBR = B000 0000H  
 SBR0 =  $000001_2 = 0000\ 0001H$   
 SBR1 =  $000011_2 = 0000\ 0003H$

**Example 13-2.** The user wants to program the SDRAM memory space to begin at B000 0000H. Bank 0 is 16 Mbytes and Bank 1 is unpopulated. The registers would be programmed as follows:

SDBR = B000 0000H  
 SBR0 =  $000100_2 = 0000\ 0004H$   
 SBR1 =  $000100_2 = 0000\ 0004H$

**Example 13-3.** The user wants to program the SDRAM memory space to begin at B000 0000H. Bank 0 is 8 Mbytes and Bank 1 is 8 Mbytes yielding in a total memory of 16 Mbytes. The registers would be programmed as follows:

SDBR = B000 0000H  
 SBR0 =  $000010_2 = 0000\ 0002H$   
 SBR1 =  $000100_2 = 0000\ 0004H$

Table 13-9 shows the programming for SDRAM memory space.

**Table 13-9. Programming Values for the SDRAM Boundary Registers (SBRx[5:0])**

Bank Size	Bank 0 (SBR0)	Bank 1 (SBR1)
Empty	$SBR0 = 0x00 + SDBR[26:22]$	$SBR1 = 0x00 + SBR0[5:0]$
4 Mbytes	$SBR0 = 0x01 + SDBR[26:22]$	$SBR1 = 0x01 + SBR0[5:0]$
8 Mbytes	$SBR0 = 0x02 + SDBR[26:22]$	$SBR1 = 0x02 + SBR0[5:0]$
16 Mbytes	$SBR0 = 0x04 + SDBR[26:22]$	$SBR1 = 0x04 + SBR0[5:0]$
32 Mbytes	$SBR0 = 0x08 + SDBR[26:22]$	$SBR1 = 0x08 + SBR0[5:0]$
64 Mbytes	$SBR0 = 0x10 + SDBR[26:22]$	$SBR1 = 0x10 + SBR0[5:0]$
128 Mbytes	$SBR0 = 0x20 + SDBR[26:22]$	$SBR1 = 0x20 + SBR0[5:0]$

### 13.3.2 SDRAM Addressing

Table 13-10 illustrates internal address mapping to the SA[11:0] lines for 16 Mbit SDRAM devices.

**Table 13-10. SDRAM Address Translation for 16 Mbit Devices**

SA[11:0]	Row	Col
11	–	–
10	L_AD[21]	V <sup>1</sup>
9	L_AD[20]	–
8	L_AD[19]	L_AD[22]
7	L_AD[18]	L_AD[10]
6	L_AD[17]	L_AD[9]
5	L_AD[16]	L_AD[8]
4	L_AD[15]	L_AD[7]
3	L_AD[14]	L_AD[6]
2	L_AD[13]	L_AD[5]
1	L_AD[12]	L_AD[4]
0	L_AD[11]	L_AD[3]

**NOTES:**

1. A10 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.

Table 13-11 illustrates internal address mapping to the SA[11:0] lines for 64 Mbit SDRAM devices.

**Table 13-11. SDRAM Address Translation for 64 Mbit Devices**

SA[11:0]	Row	Col
11	L_AD[22]	–
10	L_AD[21]	V <sup>1</sup>
9	L_AD[20]	–
8	L_AD[19]	L_AD[23]
7	L_AD[18]	L_AD[10]
6	L_AD[17]	L_AD[9]
5	L_AD[16]	L_AD[8]
4	L_AD[15]	L_AD[7]
3	L_AD[14]	L_AD[6]
2	L_AD[13]	L_AD[5]
1	L_AD[12]	L_AD[4]
0	L_AD[11]	L_AD[3]

**NOTES:**

1. A10 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.

Differences between 16 Mbit and 64 Mbit address translations occur in SA[9:8] during the column access. The leaf select decoding (SBA[1:0]) is shown in 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.

### 13.3.3 32-bit Mode

Using 16 Mbit SDRAM, a 64-bit data bus yields a minimum memory size of 8 Mbytes. To address cost-sensitive applications requiring less than 8 Mbytes of local memory, the MCU supports a 32-bit data bus. While 32-bit mode decreases the memory size, the bus throughput reduces to 264 Mbytes/sec.

The MCU does not support switching between 32-bit mode and 64-bit mode. The data bus width is sampled from the 32BITMEM\_EN# pin on reset. The data bus width can be polled with software by reading bit 2 of the SDCR.

Reducing the data bus width by half also reduces the page size by half. Therefore, when the MCU is in 32-bit mode, the page size is 1 Kbytes versus 2 Kbytes for 64-bit mode. The MCU disconnects from the internal bus if the page is crossed during a burst read or write.

### 13.3.4 Page Hit/Miss Determination

For 64-bit mode, the MCU address translation assumes a 2 Kbyte page even if the physical addressing allows a greater page size. For 32-bit mode, the MCU address translation assumes a 1 Kbyte page. For 16 Mbit devices, the MCU keeps two pages per bank (4 maximum) open simultaneously allowing greater performance for sequential accesses distributed across multiple internal bus transactions. For 64 Mbit devices, the MCU keeps four pages per bank (8 maximum) open simultaneously.

For 16 Mbit devices, the MCU keeps only one page each of Bank0/Leaf0, Bank0/Leaf1, Bank1/Leaf0, and Bank1/Leaf1 open simultaneously. This rule implies that one 2 Kbyte page per quarter (1 Kbyte for 32-bit mode) of the memory can be open. See [Figure 13-6](#) for an example organization using 16 Mbit devices.

For 64 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 (1 Kbyte for 32-bit mode) 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 Mbit SDRAM devices, 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 13-6. Logical Memory Image of a 16 Mbit SDRAM Memory Subsystem**

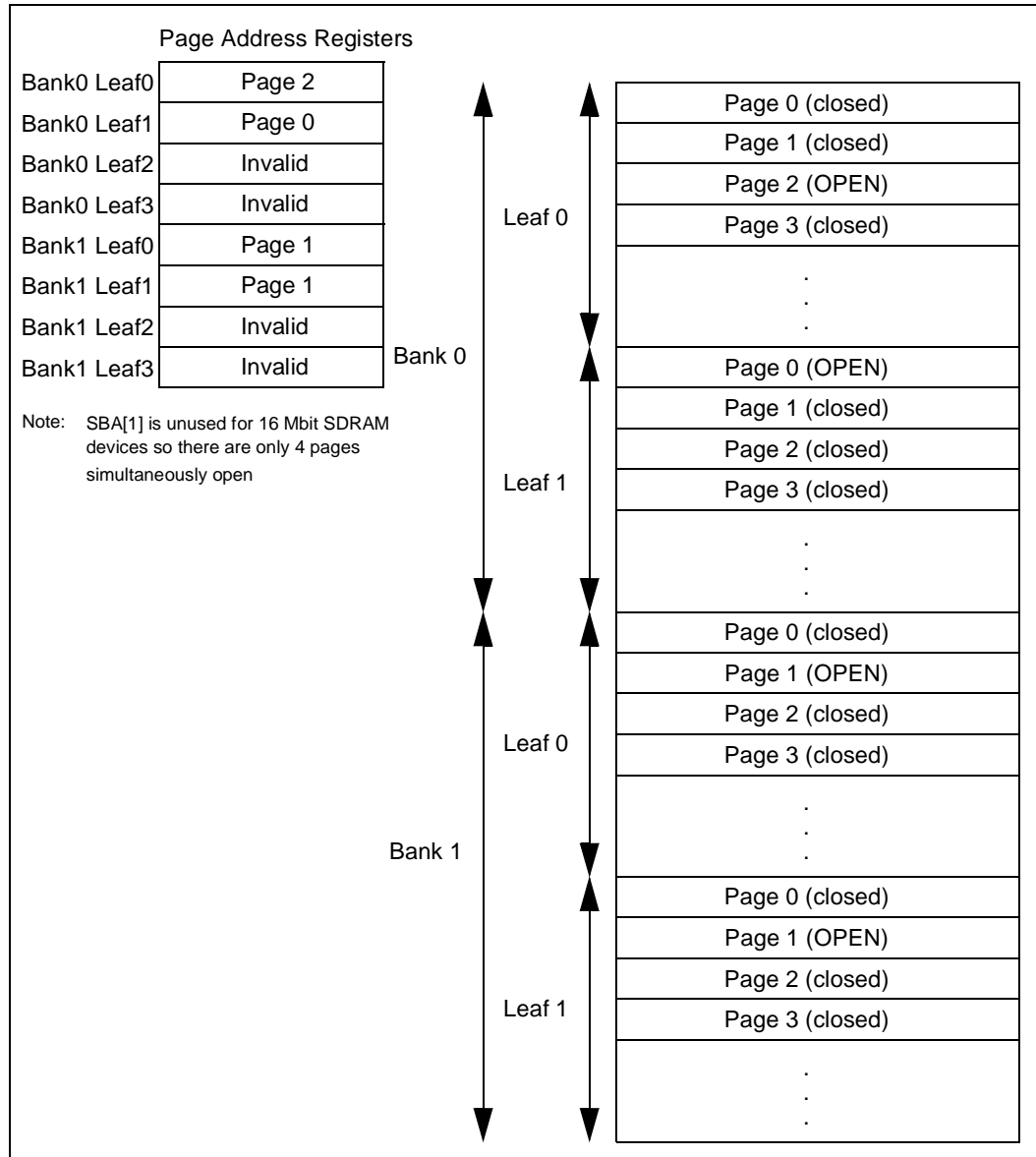


Figure 13-6 illustrates how the logical memory image is partitioned with respect to open and closed pages. If the above image represents an 8 Mbyte SDRAM memory size, each bank is 4 Mbytes and each leaf is 2 Mbytes.

**Figure 13-7. Logical Memory Image of a 64 Mbit SDRAM Memory Subsystem**

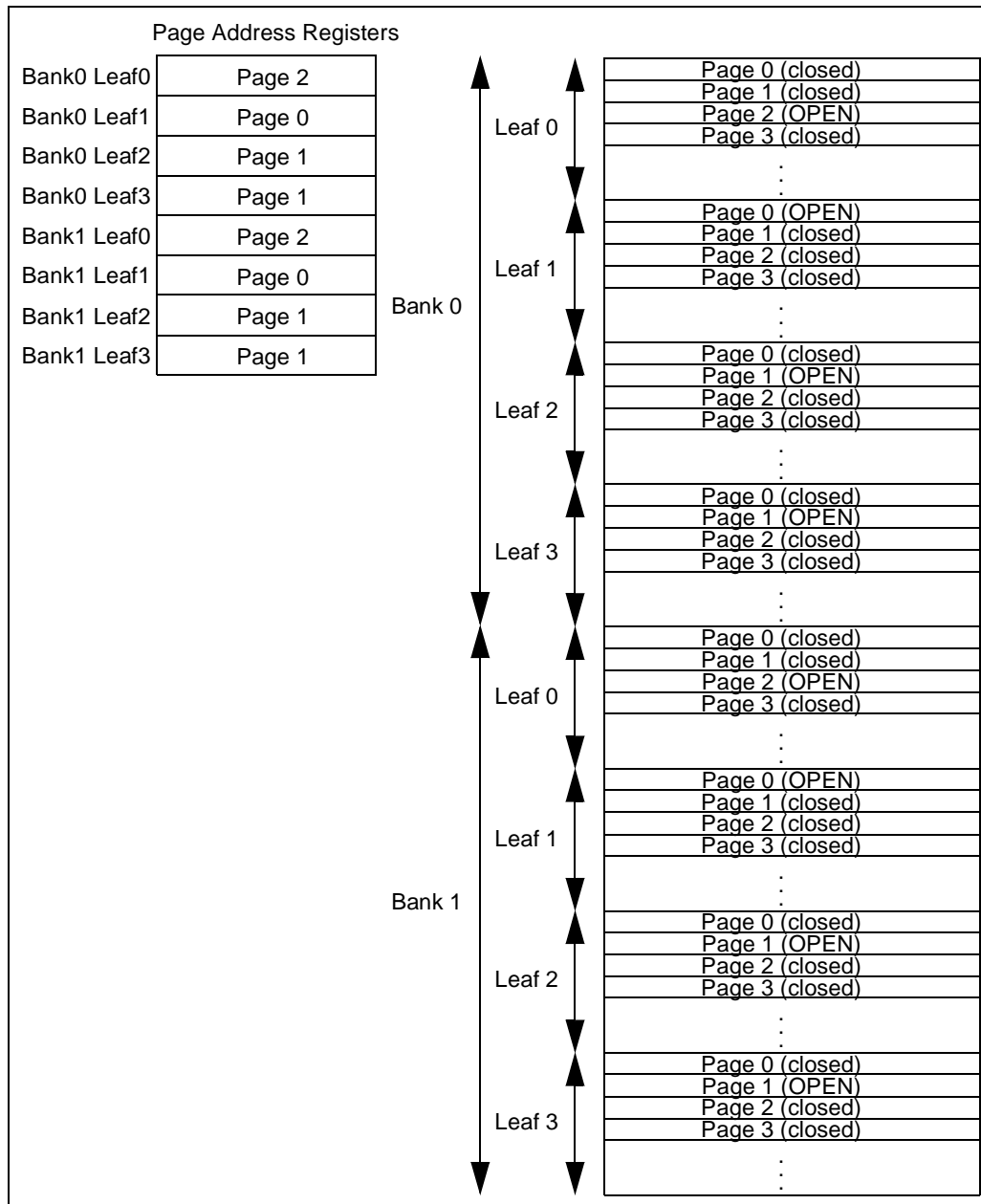


Figure 13-7 illustrates how the logical memory image is partitioned with respect to open and closed pages. If the above image represents a 32 Mbyte SDRAM memory size, each bank is 16 Mbytes and each leaf is 4 Mbytes.

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 2 Kbytes for 64-bit mode and 1 Kbyte for 32-bit mode. The programmer can optimize SDRAM transactions by partitioning code and data across the leaf boundaries to maximize the number of page hits.



### 13.3.5 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[11:0]
Row Activate	0	0	1	1	SBA[0] = Leaf	Activate a row specified on SA[11:0]
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.

**NOTE:** Shaded boxes indicate commands not supported by i960 RM/RN I/O processor. They are included for completeness.

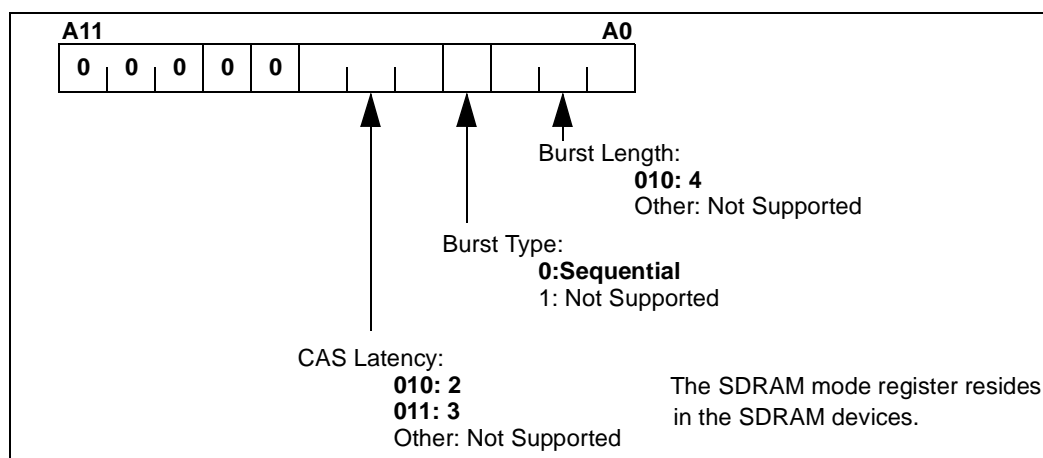
SDRAM commands are synchronous to the clock so the MCU sets up the above conditions prior to the DCLKOUT rising edge.

### 13.3.6 SDRAM Initialization

Since SDRAM devices contain a controller within the device, the MCU must initialize them specifically. On reset, software initializes the SDRAM devices with the sequence illustrated with Figure 13-9:

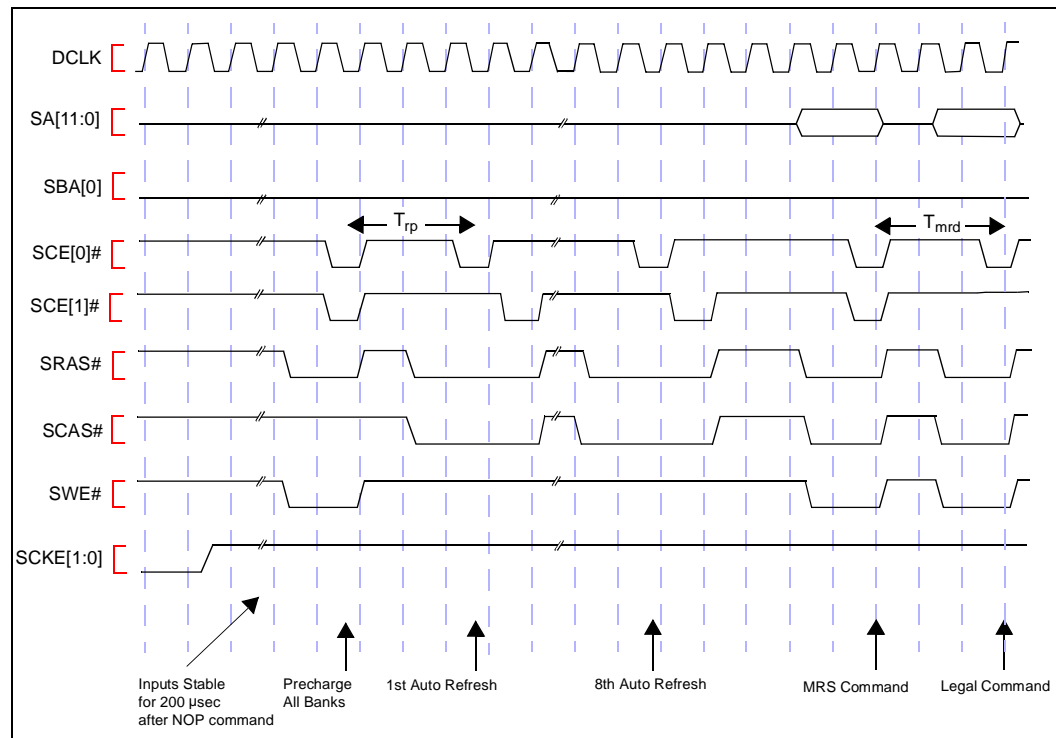
1. The MCU applies the clock (DCLKOUT) at power up along with system power (clock frequency unknown).
2. The MCU must stabilize DCLKOUT within 100  $\mu$ s after power stabilizes.
3. The MCU holds all the control inputs inactive (SRAS#, SCAS#, SWE#, SCE[1:0]# = 1) and deasserts SCKE[1:0] for a minimum 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<sub>2</sub>. The MCU asserts SCKE[1:0] with the **NOP**.
6. Software pauses 200  $\mu$ sec 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<sub>2</sub>.
9. Software provides eight **auto-refresh** cycles. An **auto-refresh** cycle is accomplished by setting the SDIR to 100<sub>2</sub>. 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<sub>2</sub> programs the MCU for CAS Latency of two while setting the SDIR to 001<sub>2</sub> 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
11. The MCU may issue a **row-activate** command three clocks after the **mode-register-set** command ( $T_{mrd}$ ).

Figure 13-8. Supported SDRAM Mode Register Settings



The waveform in [Figure 13-9](#) illustrates the SDRAM initialization sequence.

**Figure 13-9. SDRAM Initialization Sequence (controlled with software)**



**Note:** If the power failure logic is implemented ([Section 13.4, “Power Failure Mode”](#) on page 13-34), the initialization sequence differs between a cold start and a warm start. If the power fails, the above initialization sequence is not required when the system is powered up. In fact, disabling the refresh counter might cause the data to be lost while waiting the 200  $\mu$ s. After recovering from a power failure, a single MRS command is required to both assert SCKE[1:0] and reprogram the CAS latency within the MCU.

If the SDRAM subsystem implements ECC ([Section 13.3.7, “Error Correction and Detection”](#) on page 13-29), then initialization software should initialize the entire memory array with the i960 RM/RN 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 reports an ECC error even though software did not specifically request the uninitialized data.

### 13.3.6.1 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 ([Section 13.3.6, “SDRAM Initialization”](#) on page 13-18) 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.

### 13.3.6.2 SDRAM Read Cycle

Read performance is optimized for page hits and the MCU's behavior is different for the hit and miss scenario. Both read descriptions below assume that ECC is enabled.

**Note:** To accommodate a heavily loaded memory subsystem ( $\geq 18$  SDRAM devices), the MCU drives SA[11:0], 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.

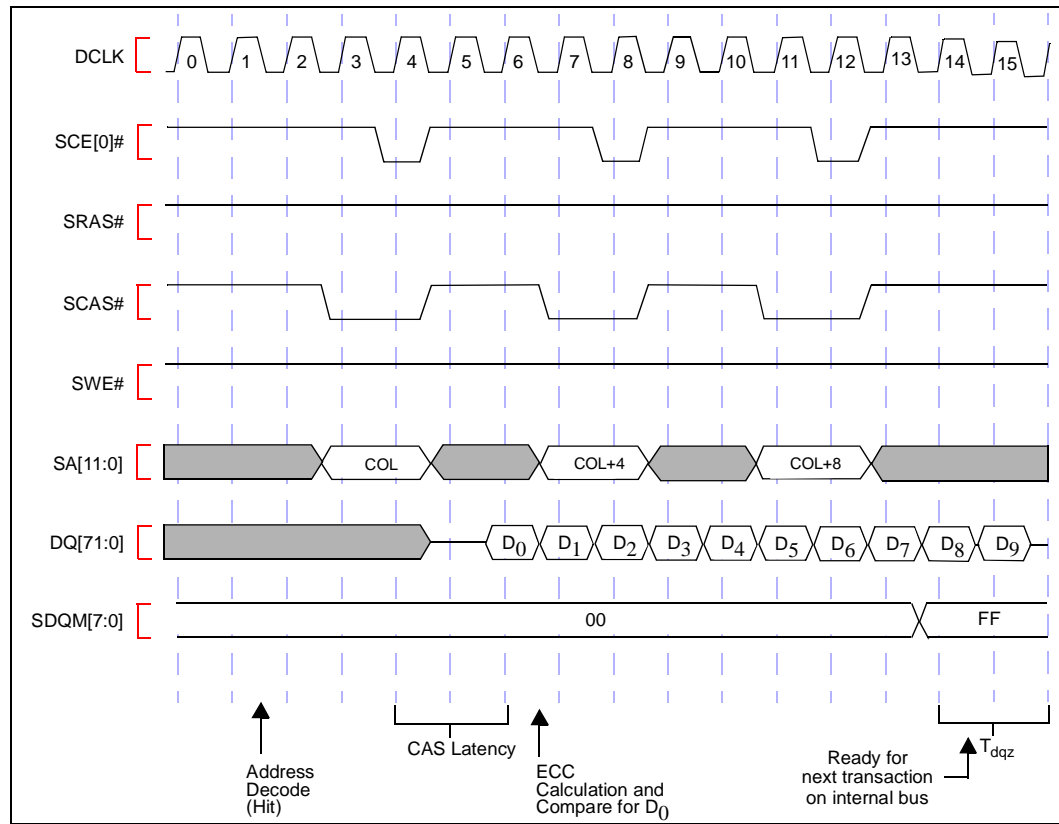
#### Read Page Hit

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](#).

- 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[11:0]. 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 13.3.7, “Error Correction and Detection” on page 13-29](#) 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.

Figure 13-10. SDRAM Read, 40 bytes, ECC Enabled, Page Hit

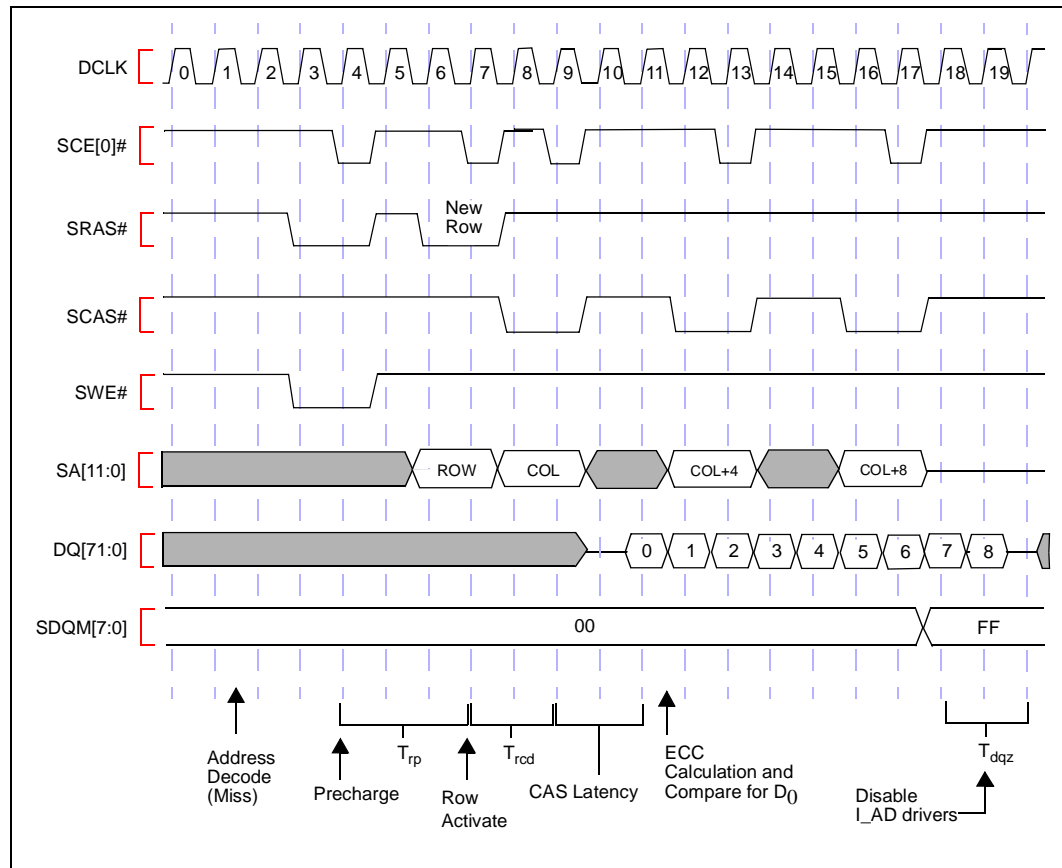


### Read Page Miss

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 13.3.4, “Page Hit/Miss Determination” on page 13-14](#) 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.

- 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  $SA[11:0]$ .
- After  $T_{rcd}$  (2) cycles, the MCU issues the **read** command by asserting  $SCAS\#$  while driving the column address on  $SA[11:0]$ .

The remainder of the read transaction is identical to a [“Read Page Hit” on page 13-20](#) beginning at clock cycle 8.

**Figure 13-11. SDRAM Read, 40 bytes, ECC Enabled, Page Miss**


### 13.3.6.3 SDRAM Write Cycle

The performance is best for page hits and therefore the MCU's behavior is different for the hit and miss scenario. Both descriptions below assume that ECC is enabled. [Section 13.3.7, "Error Correction and Detection"](#) on page 13-29 explains the ECC algorithm in more detail.

**Note:** To accommodate a heavily loaded memory subsystem ( $\geq 18$  SDRAM devices), the MCU drives SA[11:0], 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.

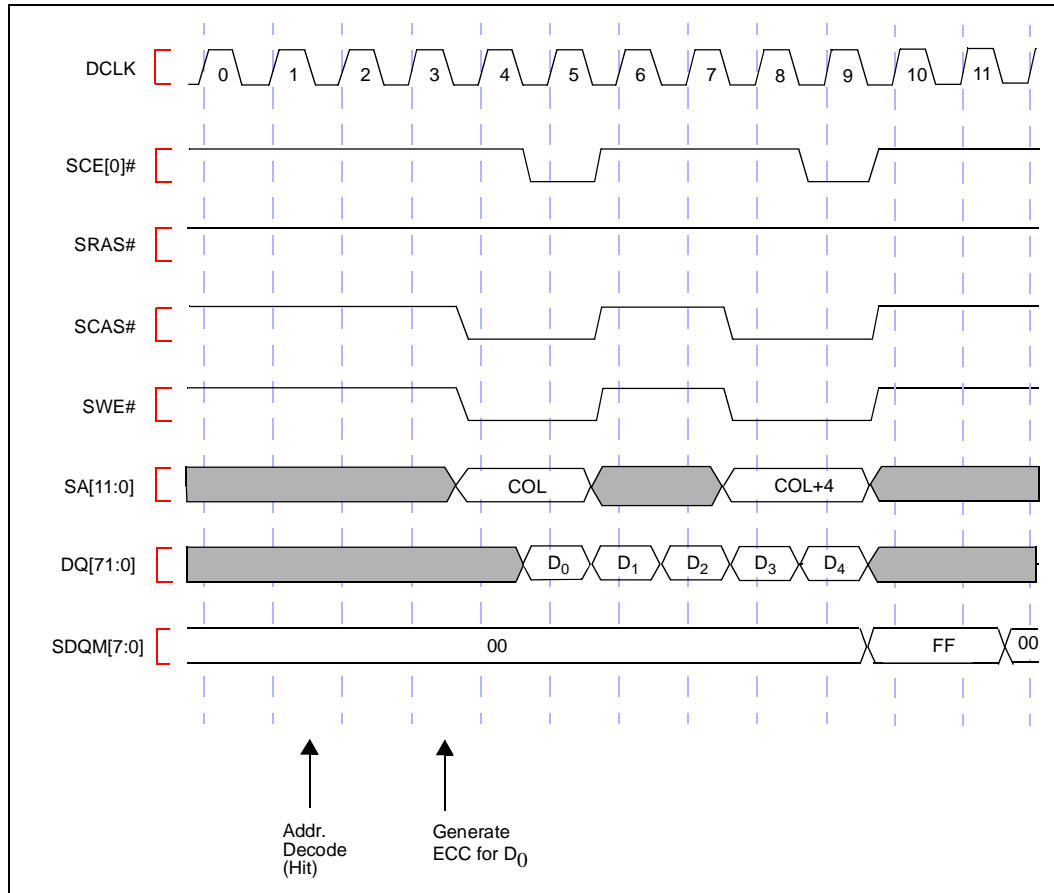
#### Write Page Hit

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](#).

- 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[11:0]. 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.
- If the data to write is not aligned on a 32 byte boundary, the unneeded bytes are masked by asserting SDQM[7:0] during the extra data cycles.



Figure 13-12. SDRAM Write, 40 bytes, ECC Enabled, Page Hit



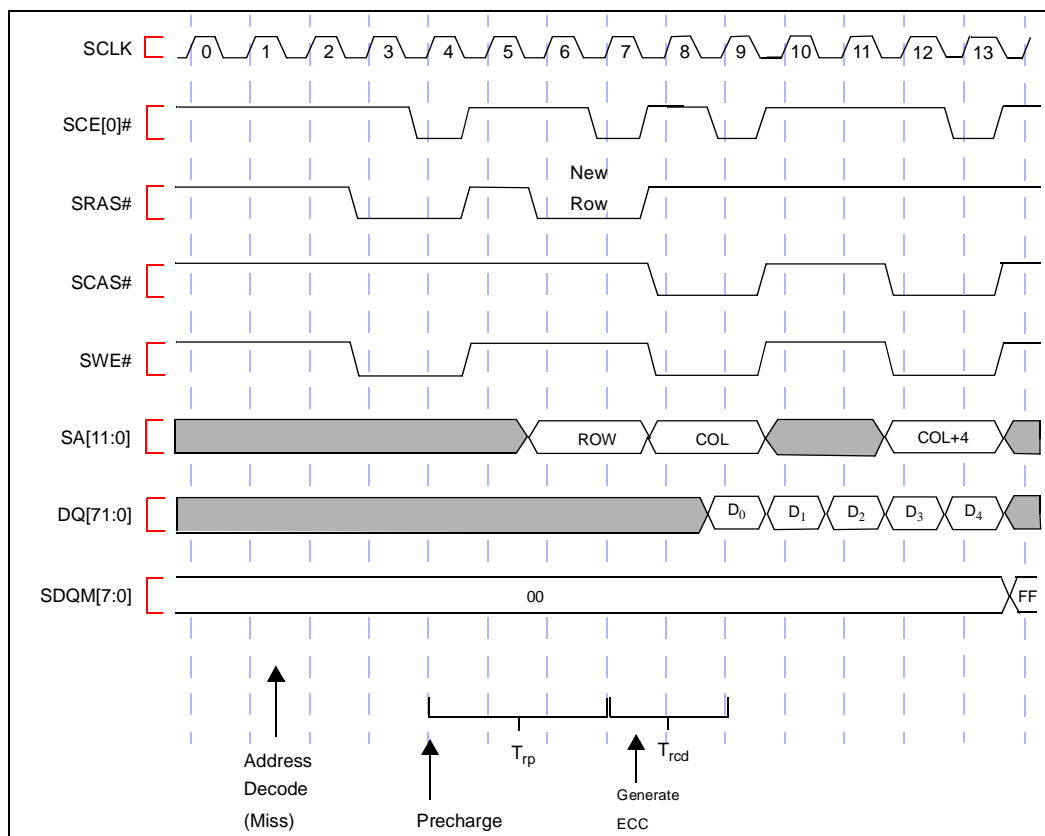
### Write Page Miss

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.

- 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.
- 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[11:0].
- After  $T_{rcd}$  (2) cycles, the MCU issues the **write** command by asserting **SCAS#** and driving the column address on SA[11:0].

The remainder of the write transaction is identical to “Write Page Miss” on page 13-26.

**Figure 13-13. SDRAM Write, 40 bytes, ECC Enabled, Page Miss**



### 13.3.6.4 SDRAM Refresh Cycle

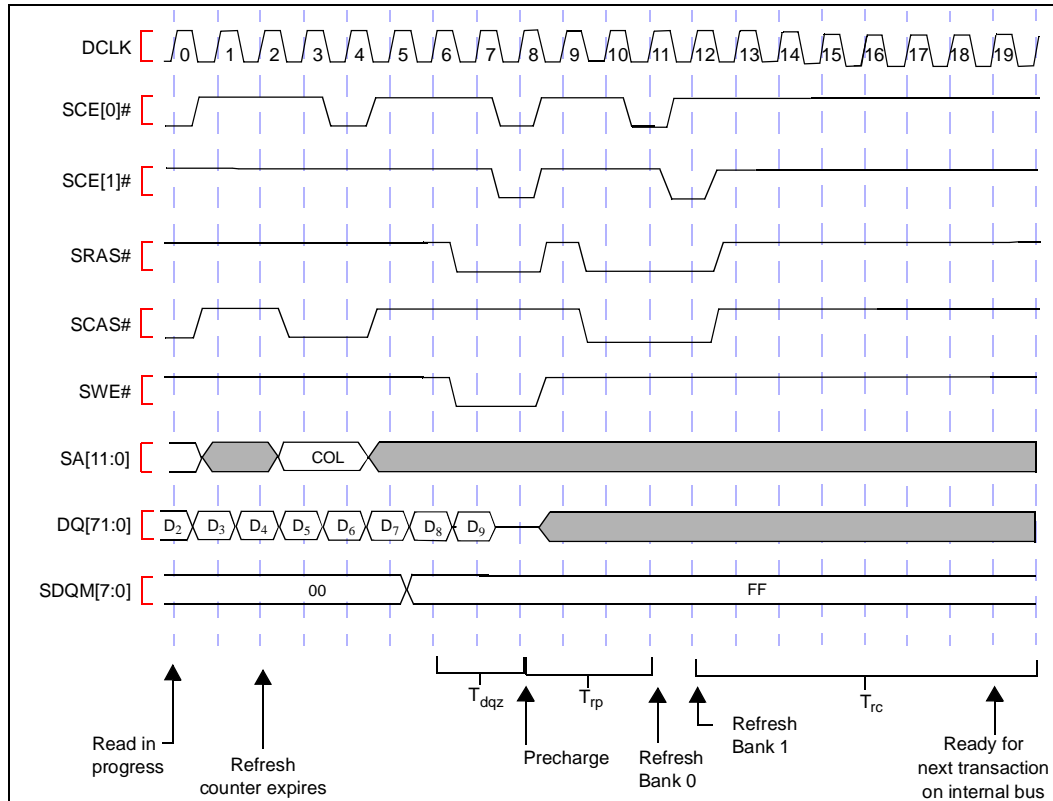
Since the SDRAM is a dynamic memory, the MCU issues a refresh cycle periodically. The interval of these refresh cycles is programmable in the RFR register. The SDRAM device generates the refresh address internally. The MCU initiates two sequential refresh cycles (one per bank) after the MCU's refresh 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.

- 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 both 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. If the primary PCI bus is 33 MHz, the internal bus is 66 MHz and RFR should be programmed to 400H. If the primary PCI bus is less than 66 MHz, the RFR should be programmed to 300H.

A timer value of 300H is sufficient for internal bus frequencies down to 50 MHz. The longest possible internal bus transaction is writing a 2 Kbyte page where each data cycle results in a read-modify-write due to partial writes ([Section 13.3.7.2, “ECC Generation for Partial Writes” on page 13-30](#)). Such a transaction could potentially require queuing two refresh cycles.

Figure 13-14. Refresh Following a Read Cycle



## 13.3.7 Error Correction and Detection

The MCU is capable of correcting any single bit errors and detecting any double bit errors in the i960 RM/RN 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 MCU does not correct a single-bit error in the array. For the i960 RM/RN 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.

The MCU supports ECC for 32-bit data mode in the same fashion as 64-bit mode. The top 32 data bits are assumed to be all zeroes when generating or comparing the ECC in 32-bit mode. For 32-bit mode details, see [Section 13.3.3, "32-bit Mode" on page 13-14](#).

### 13.3.7.1 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 with the G-matrix
    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
```

### 13.3.7.2 ECC Generation for Partial Writes

If the internal bus master writes less than the data bus width programmed in the SDCR and ECC is enabled in the ECCR, 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:

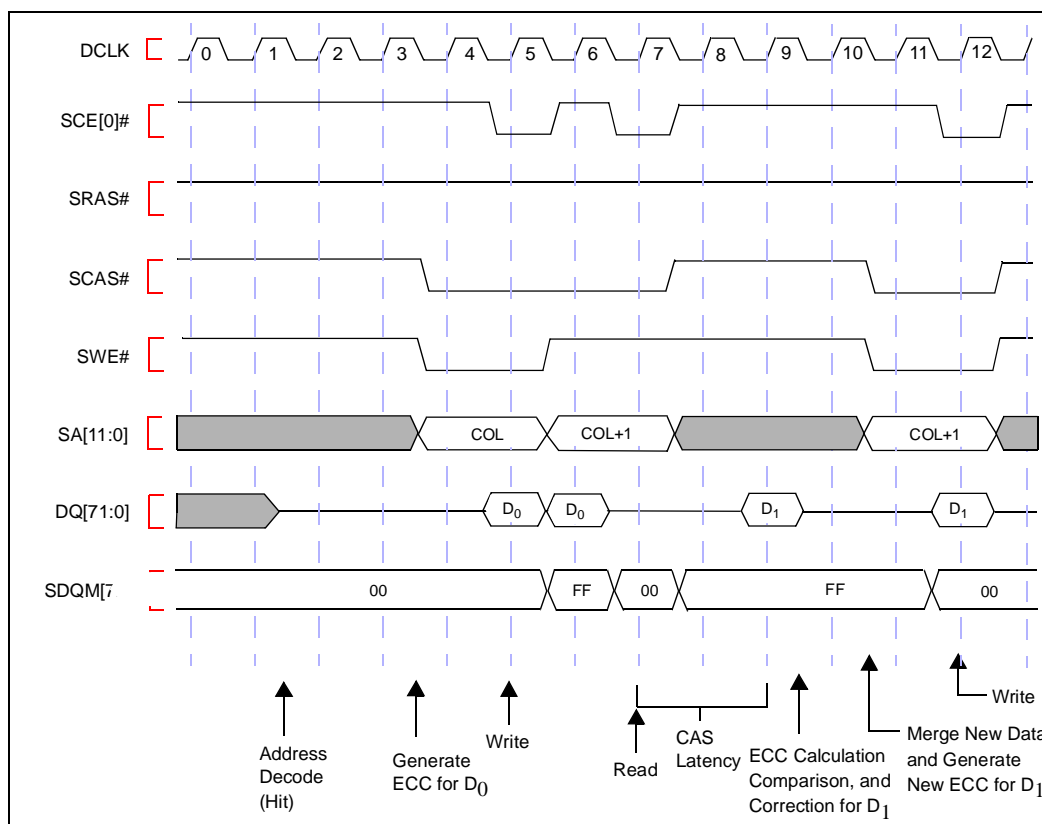
1. Issues a 64-bit read for a 64-bit data bus or a 32-bit read for a 32-bit data bus.
2. Modifies the value with the new portion to be written.
3. Calculates the ECC on the modified value.
4. Writes the 64 or 32-bit value and ECC.

If ECC is not enabled, the above read-modify-write flow is not required for a partial write. The MCU writes the partial data without penalty.

Figure 13-15 shows an example where the second data of a burst write to bank 0 is less than 64-bits wide. The data bus is programmed for 64 bits wide in the SDCR. The waveform illustrates how the MCU issues a read-modify-write cycle for the second data (D<sub>1</sub>).

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<sub>1</sub>)



### 13.3.7.3 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 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". The other nibble contains all "0". 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 ([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 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 13.5, "Interrupts/Error Conditions" on page 13-39](#).

### 13.3.7.4 Scrubbing

Fixing the data error in memory is called scrubbing. The i960 RM/RN I/O processor relies on software scrubbing. Once the MCU detects an error during a read, the MCU registers the address where the error occurred and interrupts the core. The core decides how to fix the error through an interrupt handler. To prevent coherency problems, software needs to use the 80960JT core processor's **atmod** instruction to fix the error. Software could decide to perform the scrubbing on:

- the data location that failed
- the entire row of the data that failed
- the entire memory

For single-bit errors, the error is fixed by reading the location that failed and writing back the data after the ECC hardware fixed it. If the SDRAM array implements a 64-bit wide array, then the scrubbing routine should read the 64-bit word using a **ldl** instruction and write the data with a **stl** instruction. A 32-bit wide memory can be handled with a **ld/st** combination.

**Note:** If 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.

Double-bit or nibble errors cannot be fixed.

### 13.3.7.5 ECC Disabled

If software disables ECC, the MCU does not generate the ECC byte for writes or check the ECC byte for reads. In addition, any writes less than 64 bits does not result in a read-modify-write operation as in [Figure 13-15 “Sub 64-bit SDRAM Write \(D1\)” on page 13-30](#).

### 13.3.7.6 ECC Testing

[Section 13.3.7.4, “Scrubbing” on page 13-32](#) explains how the software is responsible for correcting an error in the memory array once it has been detected by the ECC logic. 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 ([Section 13.5, “Interrupts/Error Conditions” on page 13-39](#)).

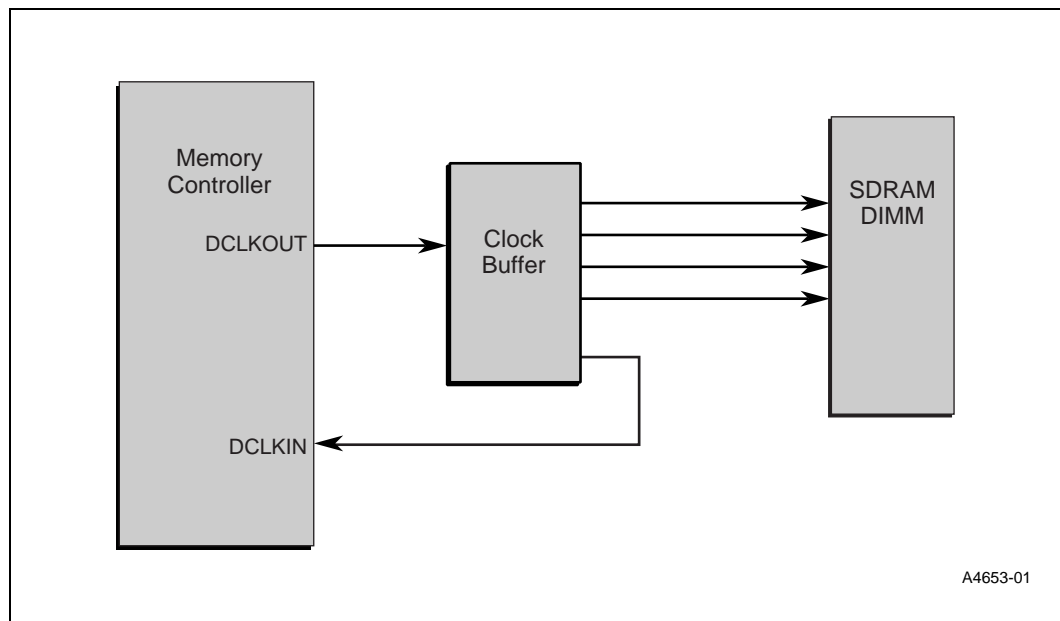


### 13.3.8 SDRAM Clocking

The MCU provides 1 clock (DCLKOUT) to the SDRAM memory subsystem at 66 MHz. The 72-bit 2-bank SDRAM DIMM specification requires 4 clocks to distribute the loading across eighteen x8 SDRAM components. DCLKOUT is buffered external to the i960 RM/RN I/O processor. To satisfy the loading requirements, the buffer outputs are driven to the SDRAM subsystem.

One of the buffered clocks is driven back into the i960 RM/RN I/O processor so that DCLKOUT may be skewed back to accommodate for the clocks' flight time. The amount of skew is determined by the board trace length. Refer to [Figure 13-16](#) for the layout diagram. SDRAM layout details as well as the clocking strategy are recommended in the *i960® RM/RN I/O Processor Design Guide*.

**Figure 13-16. SDRAM Clocking**



## 13.4 Power Failure Mode

This section defines the mechanism that the i960 RM/RN I/O processor's memory controller uses to ensure that the data within local memory is not lost during a power failure.

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 refreshes itself autonomously with internal logic and timers. The **self-refresh** command is defined in [Table 13-12](#).

The SDRAM device remains 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.

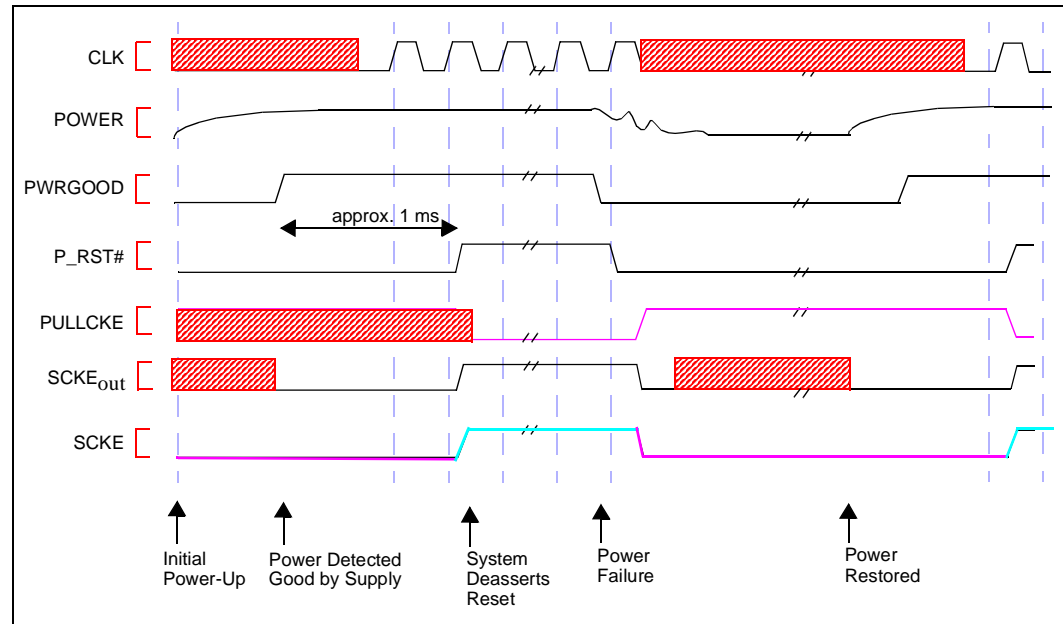
Power to the SDRAM subsystem is ensured with an adequate battery backup and a reliable method for switching between system power and battery power. The memory controller is responsible for deasserting SCKE[1:0] when issuing the **self-refresh** command but while power gradually drops, SCKE[1:0] **MUST** remain deasserted regardless of the state of  $V_{cc}$  powering the i960 RM/RN I/O processor.

**Note:** The operation of any memory (SDRAM/flash) transactions are not guaranteed when P\_RST# is asserted.

## 13.4.1 Power Failure Sequence

Figure 13-17 illustrates the sequence of events during a power failure as defined by *PCI Local Bus Specification Revision 2.1*.

Figure 13-17. Power Failure Sequence



### 13.4.1.1 Power Failure Impact on the System

Upon initial power-up a power supply provides the appropriate voltage to the system. The voltage level increases 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 2.1* 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

### 13.4.1.2 System Assumptions

Specific assumptions are made 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 i960 RM/RN I/O processor when there is at least of reliable power remaining. This is required so that the memory controller can execute it's power-failure state machine in response to the assertion of P\_RST#.

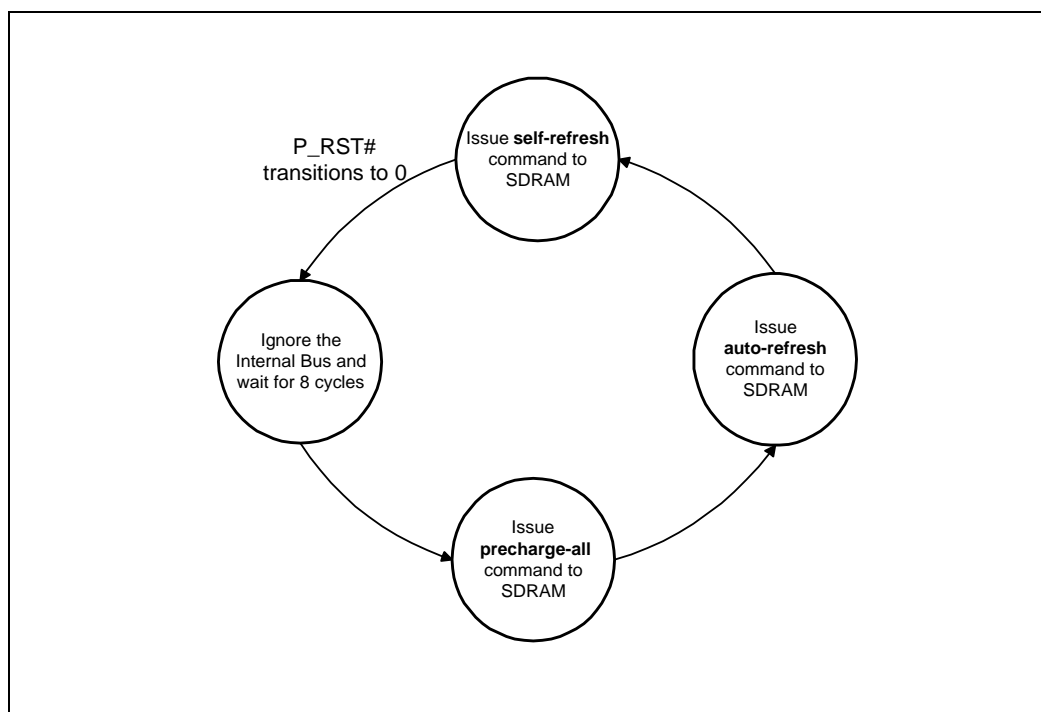
## 13.4.2 Memory Controller Response to Reset

The memory controller assumes a power failure condition whenever P\_RST# is asserted. 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 is 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 66 MHz operation, the memory controller must drive two copies to the SDRAM DIMM. The board layout distributes the two SCKE[1:0] signals between the two SDRAM banks equally.

Refer to [Figure 13-18](#) for a high-level state machine representation illustrating the memory controller's behavior during a power failure condition.

**Figure 13-18. Power Failure State Machine**

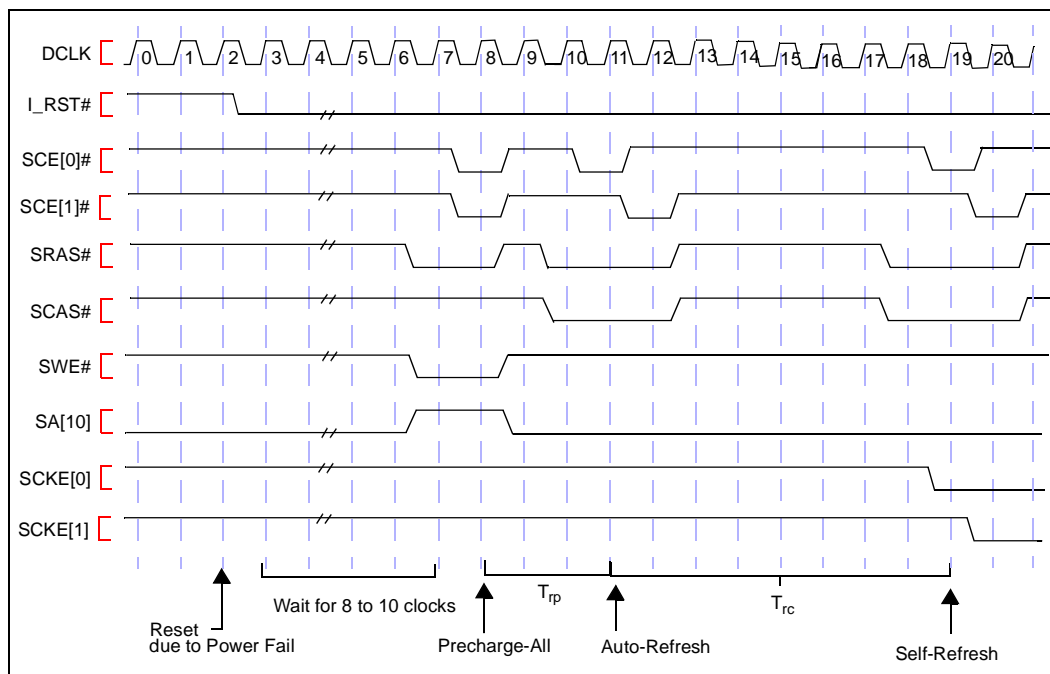


Once the memory controller detects the assertion of I\_RST#, the memory controller:

- Ignores the internal bus.
- Waits 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.
- Deactivates all SDRAM leaves with the **precharge-all** command.
- Issues an **auto-refresh** command and wait  $T_{RC}$  (8) clocks.
- Issues a **self-refresh** command to the SDRAM devices and continue to deassert SCKE[1:0].

Figure 13-19 illustrates the SDRAM waveforms upon the assertion of I\_RST#.

**Figure 13-19. 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 i960 RM/RN 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 13-20 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.

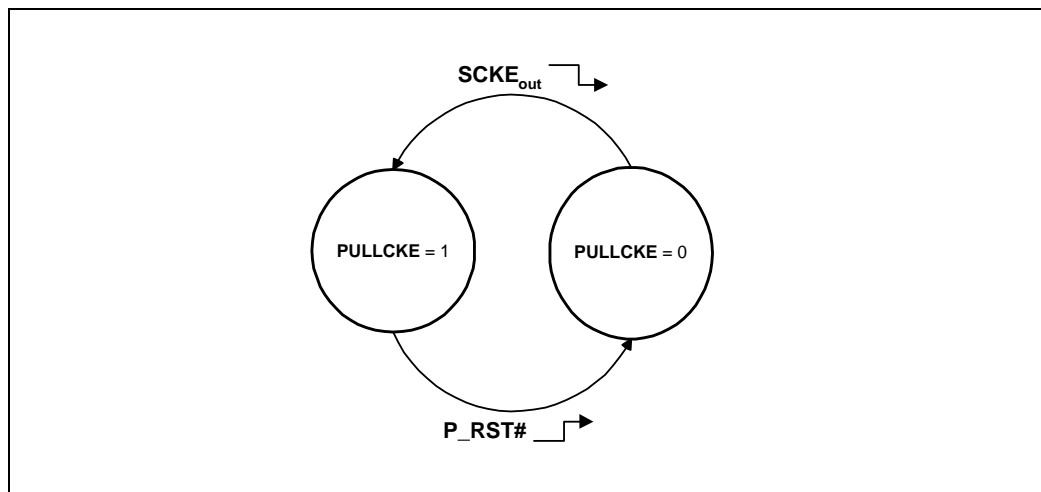
When power is restored, the system asserts P\_RST# to the i960 RM/RN I/O processor. While the i960 RM/RN I/O processor is reset, SCKE[1:0] is held low by the memory controller. After P\_RST# is deasserted, the i960 RM/RN I/O processor must be re-initialized to reset the CAS Latency parameter. The MRS 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 13.3.6, “SDRAM Initialization” on page 13-18.

**Note:** The power failure mechanism in the memory controller is not responsible for maintaining the i960 RM/RN 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 i960 RM/RN I/O processor’s components (ATUs, BIU, etc.) is lost.

### 13.4.2.1 External Logic Required for Power Failure

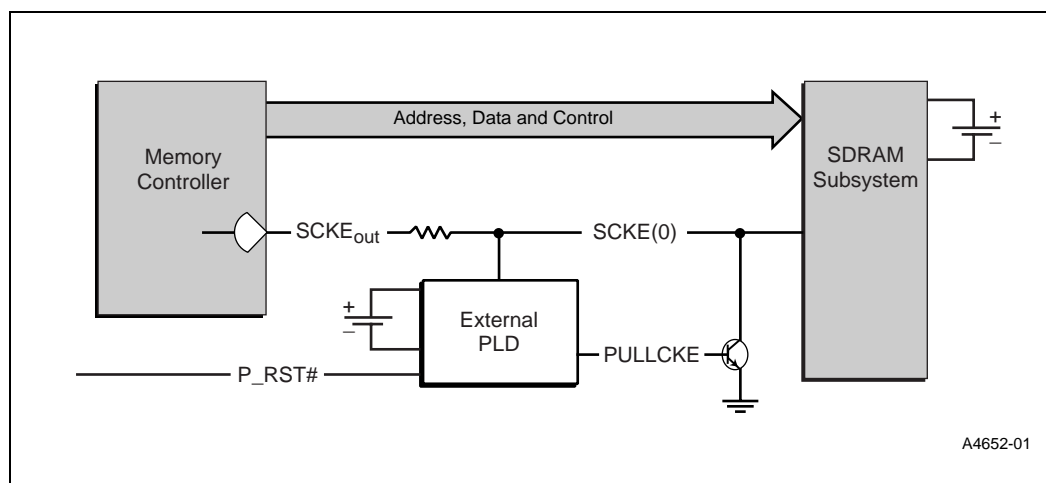
Refer to Figure 13-20 for a state machine of the external logic required for power failure mode. Actual implementations may vary. This state machine can be implemented in a programmable logic device illustrated in Figure 13-21.

Figure 13-20. External Power Failure State Machine



The implementation illustrated in Figure 13-21 requires that all external logic is powered by  $V_{batt}$ . The edge detect state machine turns 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 reliably controls SCKE[1:0] at this point driving it low.

Figure 13-21. External Power Failure Logic in the System



**Note:** Figure 13-21 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 needs to be replicated for each SCKE[1:0].

## 13.5 Interrupts/Error Conditions

The MCU has two conditions which require intervention from the i960 RM/RN 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<sup>1</sup>, 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 13-14 shows how the MCU responds to error conditions when ECC is enabled with ECCR[3].

**Table 13-14. 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.

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 13.3.7.1, “ECC Generation” on page 13-29 for details.

## 13.5.1 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 I\_AD[63:0] but the interrupt handler is responsible for scrubbing the error in the array (Section 13.3.7.4, “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.
- 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.

ESTAT, ECARx and ELOGx remain registered until software explicitly clears them. I\_RST# does not affect these registers. If an uncorrectable error resets the i960 RM/RN I/O processor, the faulty address remains registered so the system may correct the problem during initialization.

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 13-14.



## 13.5.2 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 13-14](#).

It is the interrupt handler's responsibility to decide how to handle this error condition and clear the MCISR.

## 13.5.3 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 13-15](#) for the priority rules.

**Table 13-15. 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

## 13.6 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 Reference**

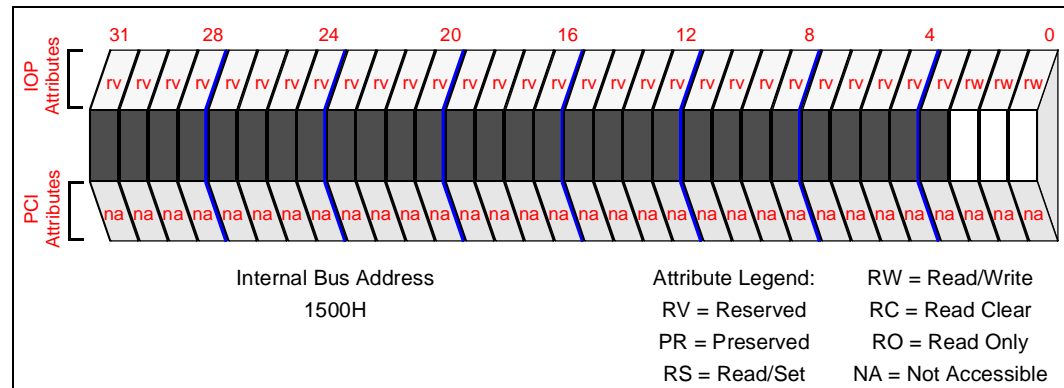
Section, Register Name - Acronym (Page)
Section 13.6.1, "SDRAM Initialization Register - SDIR" on page 13-43
Section 13.6.2, "SDRAM Control Register - SDCR" on page 13-44
Section 13.6.3, "SDRAM Base Register - SDBR" on page 13-47
Section 13.6.4, "SDRAM Boundary Register 0 - SBR0" on page 13-48
Section 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 (Section 13.3.6, “SDRAM Initialization” on page 13-18).

**Table 13-17. SDRAM Initialization Register - SDIR**

Bit	Default	Description
31:03	0	Reserved
02:00	111 <sub>2</sub>	<p><b>Special SDRAM Command:</b> These bits are used for SDRAM initialization. See Section 13.3.6, “SDRAM Initialization” on page 13-18 for details. While not in the initialization sequence, these bits should be set to 11x<sub>2</sub>. For details on the exact SDRAM commands, refer to Table 13-12 “SDRAM Commands” on page 13-17.</p> <ul style="list-style-type: none"> <li>• 000<sub>2</sub> - <b>Mode-Register-Set Command</b> where CAS# Latency = 2.</li> <li>• 001<sub>2</sub> - <b>Mode-Register-Set Command</b> where CAS# Latency = 3.</li> <li>• 010<sub>2</sub> - <b>Precharge-All Command:</b> The MCU issues one <b>precharge-all</b> command to the SDRAM devices.</li> <li>• 011<sub>2</sub> - <b>NOP Command:</b> The MCU issues one <b>NOP</b> command to the SDRAM devices.</li> <li>• 100<sub>2</sub> - <b>Auto-Refresh Command:</b> The MCU issues one <b>auto-refresh</b> command to the SDRAM devices.</li> <li>• 11x<sub>2</sub> - <b>Normal SDRAM Operation</b></li> </ul>



## 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
31:08	0	Reserved
07	0 <sub>2</sub>	<b>Address and Control Drive Strength:</b> Controls the strength of the SA[11:0], SBA[1:0], SRAS#, SCAS#, SWE# <b>SDRAM</b> output buffers. <ul style="list-style-type: none"> <li>0 - low drive strength</li> <li>1 - high drive strength</li> </ul>
06	0 <sub>2</sub>	<b>Data Mask Drive Strength:</b> Controls the strength of the SDQM[7:0] <b>SDRAM</b> output buffers. <ul style="list-style-type: none"> <li>0 - low drive strength</li> <li>1 - high drive strength</li> </ul>
05	0 <sub>2</sub>	<b>Chip Enable 1 Drive Strength:</b> Controls the strength of the SCE[1]# and SCKE[1] <b>SDRAM</b> output buffers. <ul style="list-style-type: none"> <li>0 - low drive strength</li> <li>1 - high drive strength</li> </ul>
04	0 <sub>2</sub>	<b>Chip Enable 0 Drive Strength:</b> Controls the strength of the SCE[0]# and SCKE[0] <b>SDRAM</b> output buffers. <ul style="list-style-type: none"> <li>0 - low drive strength</li> <li>1 - high drive strength</li> </ul>
03	0 <sub>2</sub>	<b>Data Bus Drive Strength:</b> Controls the strength of the DQ[63:0] and SCB[7:0] <b>SDRAM</b> output buffers. <ul style="list-style-type: none"> <li>0 - low drive strength</li> <li>1 - high drive strength</li> </ul>
02	Varies with the external state of the 32BITMEM_EN# pin at internal bus reset	<b>Data Bus Width:</b> Indicates the width of the data bus. See <a href="#">Section 13.3.3, “32-bit Mode”</a> on page 13-14. <ul style="list-style-type: none"> <li>0 - 32 bits</li> <li>1 - 64 bits</li> </ul> <p>The state of this bit is based on the external state of the 32BITMEM_EN# pin at the rising edge of P_RST#. If the external state of this pin is low, the default value for this bit is 0. If the external state of this pin is high, the default value for this bit is 1.</p>
01:00	00 <sub>2</sub>	Reserved

**Table 13-19. Drive Strength Programmability Options (16-Mbit SDRAM Technology)**

Bus Width	Memory Size (Mbytes)	Form Factor	Bank 0	Bank 1	SDCR[3] (DQ)	SDCR[4] (CKE0)	SDCR[5] (CKE1)	SDCR[6] (DQM)	SDCR[7] (SA[11:0])
32	4	On-board	2x1M16	None	0	0	0	0	0
	8		4x2M8		0	0	0	0	0
	8		4x1M16		0	0	0	0	0
	16		8x2M8		0	1	0	0	1
64	8	1 Single-sided DIMM	4x1M16	None	0	0	0	0	0
	16		8x2M8		0	1	0	0	1
	16	1 Double-sided DIMM	4x1M16	4x1M16	1	0	0	1	1
	32		8x2M8	8x2M8	1	1	1	1	1
	16	2 Single-sided DIMMs	4x1M16	4x1M16	1	0	0	1	1
	24		4x1M16	8x2M8	1	0	1	1	1
	24		8x2M8	4x1M16	1	1	0	1	1
	32		8x2M8	8x2M8	1	1	1	1	1
72	16	On-board	9x2M8	None	0	1	0	0	1
	16	1 Single-sided DIMM	9x2M8		0	1	0	0	1
	32	1 Double-sided DIMM	9x2M8	9x2M8	1	1	1	1	1
	32	2 Single-sided DIMMs	9x2M8	9x2M8	1	1	1	1	1

Table 13-20. Drive Strength Programmability Options (64-Mbit SDRAM Technology)

Bus Width	Memory Size (Mbytes)	Form Factor	Bank 0	Bank 1	SDCR[3] (DQ)	SDCR[4] (CKE0)	SDCR[5] (CKE1)	SDCR[6] (DQM)	SDCR[7] (SA[11:0])
32	16	On-board	2x4M16	None	0	0	0	0	0
	32		4x8M8		0	0	0	0	0
	32		4x4M16		0	0	0	0	0
	64		8x8M8		0	1	0	0	1
64	32	1 Single-sided DIMM	4x4M16	None	0	0	0	0	0
	64		8x8M8		0	1	0	0	1
	64	1 Double-sided DIMM	4x4M16	4x4M16	1	0	0	1	1
	128		8x8M8	8x8M8	1	1	1	1	1
	64	2 Single-sided DIMMs	4x4M16	4x4M16	1	0	0	1	1
	96		4x4M16	8x8M8	1	0	1	1	1
	96		8x8M8	4x4M16	1	1	0	1	1
	128		8x8M8	8x8M8	1	1	1	1	1
72	64	On-board	9x8M8	None	0	1	0	0	1
	64	1 Single-sided DIMM	9x8M8		0	1	0	0	1
	128	1 Double-sided DIMM	9x8M8	9x8M8	1	1	1	1	1
	128	2 Single-sided DIMMs	9x8M8	9x8M8	1	1	1	1	1

### 13.6.3 SDRAM Base Register - SDBR

This register indicates the beginning of SDRAM space. See [Section 13.3.1, “SDRAM Sizes and Configurations”](#) on page 13-11 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 128 Mbyte boundary.

**Table 13-21. SDRAM Base Register - SDBR**

<b>Bit</b>	<b>Default</b>	<b>Description</b>
31:22	0	<b>SDRAM Base Address:</b> These bits define the upper ten bits of the SDRAM base address.
21:00	0	Reserved

### 13.6.4 SDRAM Boundary Register 0 - SBR0

This register indicates the upper boundary of SDRAM bank 0 and its memory technology. If bank 0 is unpopulated, SBR0[4:0] is programmed with all zeros. See [Section 13.3.1, “SDRAM Sizes and Configurations”](#) on page 13-11 for more details and programming examples.

**Table 13-22. SDRAM Boundary Register 0 - SBR0**

		Internal Bus Address 150CH	Attribute Legend: RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set RC = Read Clear RO = Read Only NA = Not Accessible
Bit	Default	Description	
31	0 <sub>2</sub>	<b>SDRAM Technology:</b> Defines the memory subsystem technology. <ul style="list-style-type: none"> <li>• 0 - 16 Mbit</li> <li>• 1 - 64 Mbit</li> </ul>	
30:06	0	Reserved	
05:00	000000 <sub>2</sub>	<b>SDRAM Boundary:</b> Defines the upper limit of SDRAM bank 0.	



### 13.6.5 SDRAM Boundary Registers 1 - SBR1

This register indicates the upper boundary of SDRAM bank 1 and its memory technology. If bank 1 is unpopulated, SBR1[5:0] is programmed with all zeroes. If bank 1 is populated, SBR1[5:0] must be programmed greater than or equal to SBR0[4:0]. See [Section 13.3.1, “SDRAM Sizes and Configurations”](#) on page 13-11 for more details and programming examples.

**Table 13-23. SDRAM Boundary Registers - SBR1**

Bit	Default	Description	
31	0 <sub>2</sub>	<b>SDRAM Technology:</b> Defines the memory subsystem technology. <ul style="list-style-type: none"> <li>• 0 - 16 Mbit</li> <li>• 1 - 64 Mbit</li> </ul>	
30:06	0	Reserved	
05:00	000000 <sub>2</sub>	<b>SDRAM Boundary:</b> Defines the upper limit of SDRAM bank 1.	

## 13.6.6 ECC Control Register - ECCR

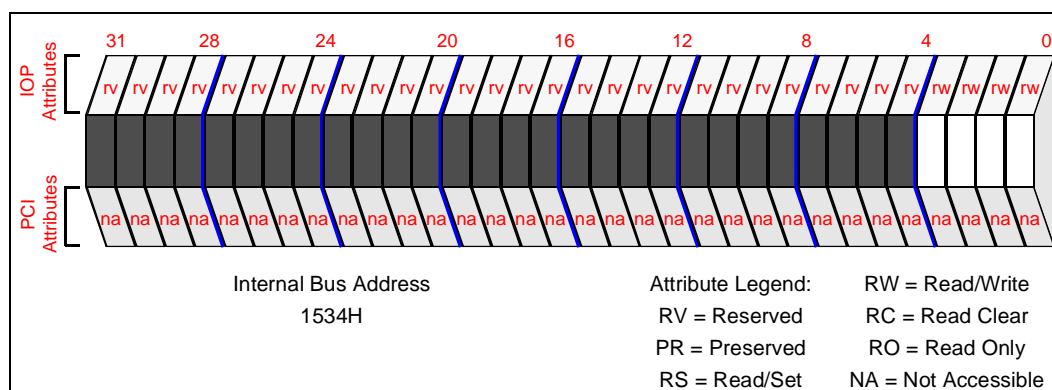
This register programs the MCU error correction and detection capabilities. The configuration depends on the application’s needs but a typical configuration is:

- ECC mode enabled
- Enable double-bit error reporting
- Disable single-bit error reporting
- Enable single-bit error correcting

For more details, see [Section 13.3.7, “Error Correction and Detection”](#) on page 13-29 and [Section 13.5, “Interrupts/Error Conditions”](#) on page 13-39.

**Table 13-24. ECC Control Register - ECCR**

Bit	Default	Description
31:04	000 0000H	Reserved
03	0 <sub>2</sub>	<b>ECC Enabled:</b> Enables ECC calculation and generation. <ul style="list-style-type: none"> <li>• 0 - ECC Disabled</li> <li>• 1 - ECC Enabled</li> </ul>
02	0 <sub>2</sub>	<b>Single Bit Error Correction Enable:</b> Enables or disables the correction of a single bit error. <ul style="list-style-type: none"> <li>• 0 - Disable single bit error correction</li> <li>• 1 - Enable single bit error correction</li> </ul>
01	0 <sub>2</sub>	<b>Multi-Bit Error Reporting Enable:</b> Enables or disables the reporting of a multi-bit error condition. <ul style="list-style-type: none"> <li>• 0 - Disable multi-bit error reporting</li> <li>• 1 - Enable multi-bit error reporting</li> </ul>
00	0 <sub>2</sub>	<b>Single Bit Error Reporting Enable:</b> Enables or disables the reporting of a single bit error condition. <ul style="list-style-type: none"> <li>• 0 - Disable single bit error reporting</li> <li>• 1 - Enable single bit error reporting</li> </ul>







### 13.6.9 ECC Test Register - ECTST

This register allows testing between the ECC logic and the memory subsystem (Section 13.3.7.6, “ECC Testing” on page 13-32). 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 13-27. ECC Test Register - ECTST**

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 13.3.7.6, “ECC Testing” on page 13-32.

Internal Bus Address 1548H	Attribute Legend: RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set RC = Read Clear RO = Read Only NA = Not Accessible
-------------------------------	--

### 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 13.2.1, “Flash Memory Addressing”](#) on page 13-4.

**Table 13-28. Flash Base Register 0 - FEBR0**

		<p>Internal Bus Address 154CH</p> <p>Attribute Legend:                  RW = Read/Write                  RV = Reserved                  RC = Read Clear                  PR = Preserved                  RO = Read Only                  RS = Read/Set                  NA = Not Accessible</p>
Bit	Default	Description
31:16	FE80H	<b>Flash Base Address:</b> 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 13.2.1, “Flash Memory Addressing”](#) on page 13-4.

**Table 13-29. Flash Base Register 1 - FEBR1**

		Attribute Legend:				RW = Read/Write		RV = Reserved		RC = Read Clear		PR = Preserved		RO = Read Only		RS = Read/Set		NA = Not Accessible			
Bit	Default	Description																			
31:16	0000H	<b>Flash Base Address:</b> 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 13.2.1, “Flash Memory Addressing”](#) on page 13-4 for more details.

**Table 13-30. Flash Bank Size Register 0 - FBSR0**

Bit	Default	Description
31:04	0	Reserved
03:00	1000 <sub>2</sub>	<p><b>Flash Bank Size:</b> Defines the size for the Flash bank.</p> <ul style="list-style-type: none"> <li>• 0000 - Bank disabled</li> <li>• 0001 - 64 Kbytes</li> <li>• 0010 - 128 Kbytes</li> <li>• 0011 - 256 Kbytes</li> <li>• 0100 - 512 Kbytes</li> <li>• 0101 - 1 Mbytes</li> <li>• 0110 - 2 Mbytes</li> <li>• 0111 - 4 Mbytes</li> <li>• 1XXX - 8 Mbytes</li> </ul>



### 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 13.2.1, “Flash Memory Addressing”](#) on page 13-4 for more details.

**Table 13-31. Flash Bank Size Register 1 - FBSR1**

Bit	Default	Description
31:04	0	Reserved
03:00	0000 <sub>2</sub>	<p><b>Flash Bank Size:</b> Defines the size for the Flash bank.</p> <ul style="list-style-type: none"> <li>• 0000 - Bank disabled</li> <li>• 0001 - 64 Kbytes</li> <li>• 0010 - 128 Kbytes</li> <li>• 0011 - 256 Kbytes</li> <li>• 0100 - 512 Kbytes</li> <li>• 0101 - 1 Mbytes</li> <li>• 0110 - 2 Mbytes</li> <li>• 0111 - 4 Mbytes</li> <li>• 1XXX - 8 Mbytes</li> </ul>

Internal Bus Address 1558H	<p>Attribute Legend:</p> <p>RW = Read/Write                      RV = Reserved                      PR = Preserved                      RS = Read/Set</p>	<p>RW = Read/Write                      RC = Read Clear                      RO = Read Only                      NA = Not Accessible</p>
-------------------------------	---	--

### 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, “Flash Read Cycle”](#) on page 13-5 and [Section 13.2.3, “Flash Write Cycle”](#) on page 13-8.

**Table 13-32. Flash Wait State Registers - FWSR0, FWSR1**

Bank #	Internal Bus Address	Attribute Legend:	RW = Read/Write
0	155CH	RV = Reserved	RC = Read Clear
1	1560H	PR = Preserved	RO = Read Only
		RS = Read/Set	NA = Not Accessible
Bit	Default	Description	
31:07	0	Reserved	
06:04	111 <sub>2</sub>	<b>Recovery Cycle Wait States:</b> Defines the number of recovery cycle wait states for the Flash bank. <ul style="list-style-type: none"> <li>• 000 - 1 Recovery wait state</li> <li>• 001 - 4 Recovery wait states</li> <li>• 010 - 8 Recovery wait states</li> <li>• 011 - 12 Recovery wait states</li> <li>• 100 - 16 Recovery wait states</li> <li>• 1x1 - 20 Recovery wait states</li> </ul>	
03	0 <sub>2</sub>	Reserved	
02:00	111 <sub>2</sub>	<b>Address-to-Data Wait States:</b> Defines the number of address-to-data wait states for the Flash bank during a read or write transaction. <ul style="list-style-type: none"> <li>• 000 - 4 Address-to-Data wait states</li> <li>• 001 - 8 Address-to-Data wait states</li> <li>• 010 - 12 Address-to-Data wait states</li> <li>• 011 - 16 Address-to-Data wait states</li> <li>• 1xx - 20 Address-to-Data wait states</li> </ul>	

### 13.6.15 Memory Controller Interrupt Status Register - MCISR

Setting the MCISR asserts an NMI to the core. Upon an interrupt, the 80960JT core processor 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 13-33. Memory Controller Interrupt Status Register - MCISR**

Internal Bus Address 1564H		
Attribute Legend: RW = Read/Write RV = Reserved RC = Read Clear PR = Preserved RO = Read Only RS = Read/Set NA = Not Accessible		
Bit	Default	Description
31:03	0	Reserved
02	0 <sub>2</sub>	<b>ECC Error N:</b> Indicates that the MCU detected an ECC error while MCISR[1] and MCISR[0] are both set. <ul style="list-style-type: none"> <li>0 - No error detected</li> <li>1 - Error detected</li> </ul>
01	0 <sub>2</sub>	<b>ECC Error 1:</b> Indicates that the MCU detected an ECC error and recorded the error in ELOG1. <ul style="list-style-type: none"> <li>0 - No error detected</li> <li>1 - Error detected and recorded in ELOG1</li> </ul>
00	0 <sub>2</sub>	<b>ECC Error 0:</b> Indicates that the MCU detected an ECC error and recorded the error in ELOG0. <ul style="list-style-type: none"> <li>0 - No error detected</li> <li>1 - Error detected and recorded in ELOG0</li> </ul>

### 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 66 MHz operation, the RFR should be programmed with a value of 400H. For frequencies below 66 MHz, the RFR should be programmed with 300H.

**Table 13-34. Refresh Frequency Register - RFR**

Internal Bus Address 1568H		Attribute Legend: RW = Read/Write RV = Reserved RC = Read Clear PR = Preserved RO = Read Only RS = Read/Set NA = Not Accessible
Bit	Default	Description
31:11	0	Reserved
10:00	300H	<b>Refresh Interval:</b> Programs the number of clocks that triggers a refresh cycle to the SDRAM interface. If all zeroes, refresh cycles are disabled.

This chapter describes the PCI-to-PCI Bridge including functionality, modes of operation, configuration, and integration into the i960<sup>®</sup> RM/RN I/O processor system architecture.

## 14.1 Overview

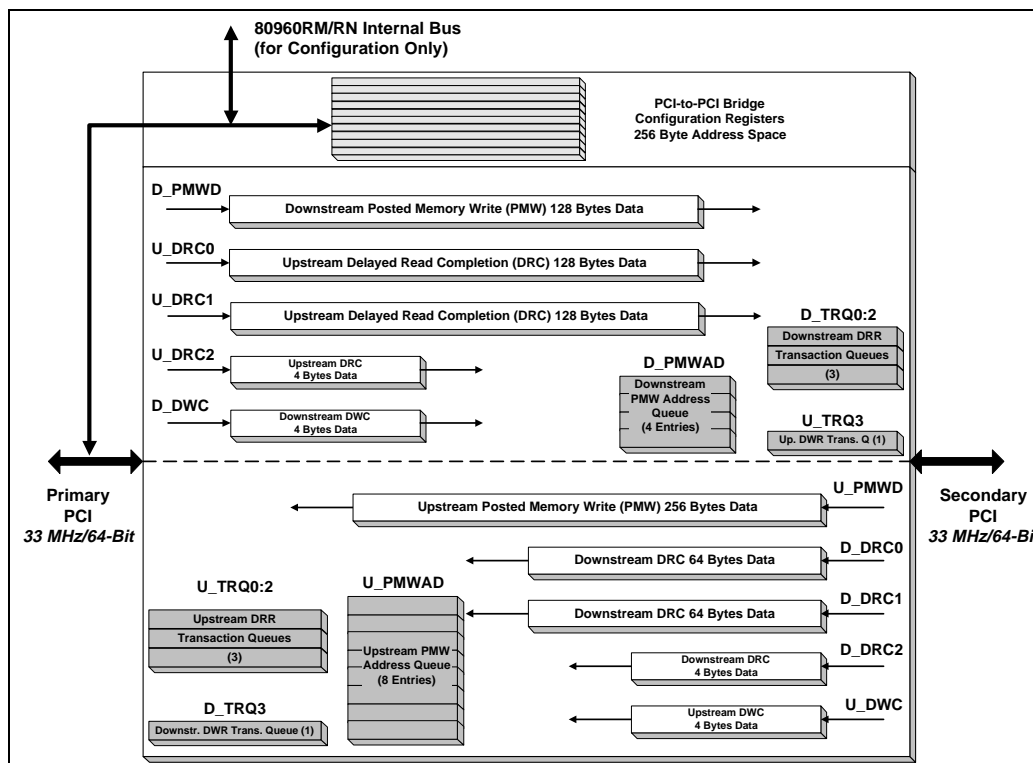
The PCI-to-PCI bridge unit extends a PCI Bus beyond its physical constraint of ten electrical PCI loads at 33 MHz. The bridge unit uses the concept of hierarchical buses; each bus in the hierarchy is electrically a separate entity, but all buses within the hierarchy are logically one bus. The PCI-to-PCI bridge unit does not increase the bandwidth of a PCI bus, 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 2.1
- Full compliance to the *PCI-to-PCI Bridge Architecture Specification* Revision 1.0
- 264 MBytes/sec PCI bandwidth on both the primary and secondary buses through 64-bit operation
- Synchronous operation between primary and secondary PCI busses
- 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 for 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 4 PMW transactions with a total of 128 Bytes of write data on downstream transactions
  - Up to 8 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
  - 260 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
- 64-bit addressing mode (Dual Address Command) for upstream cycles initiated from the secondary PCI interface
- Private device configuration and address space for private PCI devices on the secondary PCI bus

Figure 14-1 shows a block diagram of the i960 RM/RN I/O processor PCI-to-PCI Bridge unit.

Figure 14-1. PCI-to-PCI Bridge Unit Block Diagram



## 14.2 Theory of Operation

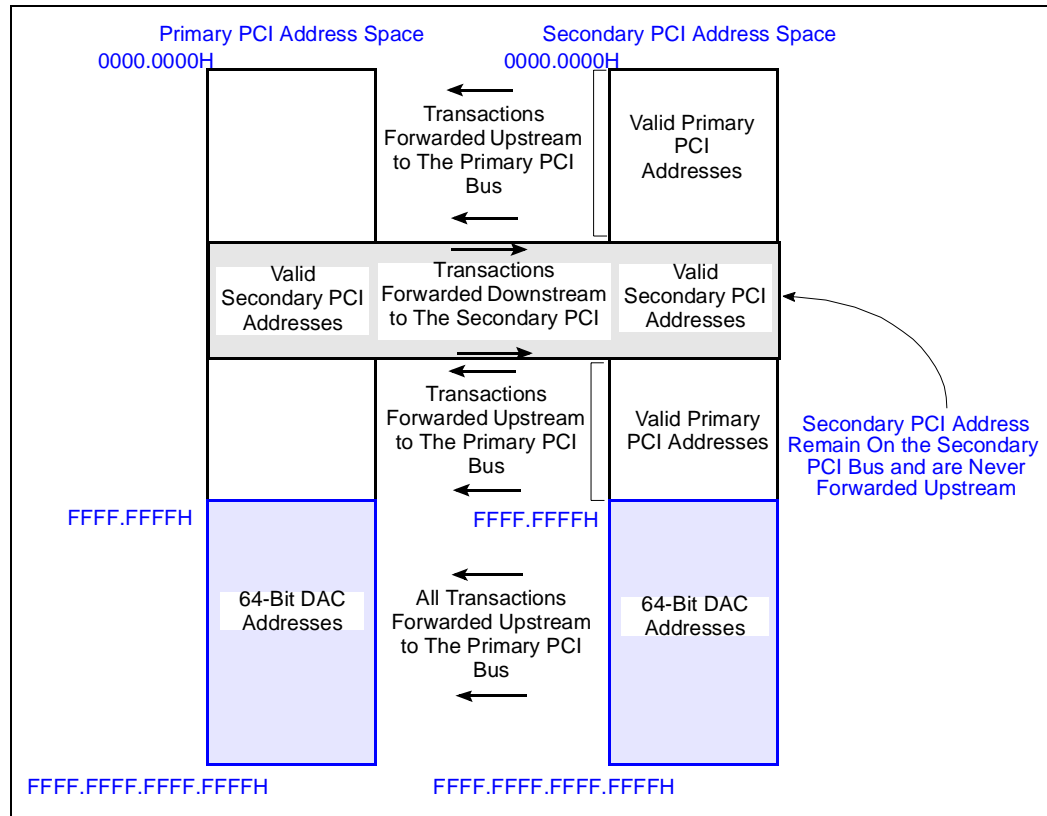
The bridge unit operates as an address filter unit between the primary and the secondary PCI buses. PCI supports three separate address spaces:

- 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 Local Bus Specification* Revision 2.1 compliant master and target devices. A PCI transaction initiated on one side of the bridge addresses the initiating bus bridge interface as a target and the transaction is 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 i960 RM/RN I/O processor adheres, at a minimum, to the required features found in the *PCI-to-PCI Bridge Architecture Specification* Revision 1.0 and the *PCI Local Bus Specification* Revision 2.1. This chapter describes bridge functionality and refers to the *PCI-to-PCI Bridge Architecture Specification* Revision 1.0 and the *PCI Local Bus Specification* Revision 2.1 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 is 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 1.0, four optional interrupt pins, and the 39 pins required by the PCI 64-bit extension. Refer to the *PCI Local Bus Specification* Revision 2.1 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's primary bus.

The primary interface of the i960 RM/RN I/O processor supports enhanced PCI bandwidth of 264 MBytes/sec through the use of the 64-bit PCI extension at a frequency of up to 33 MHz.

The additional bandwidth that the i960 RM/RN 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 1.0 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's 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, on page 14-11](#) and [Section 14.5.4, on page 14-20](#).

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 claims *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 i960 RM/RN I/O processor implements an extensive queuing 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 2.1 compliant device, the bridge unit supports both posted and delayed transactions.

In a Delayed transaction, 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, on page 14-23](#).

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 8 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
- 260 bytes of delayed read completion (DRC) data queue with three separate Transaction Address Queues
- Two 128 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 2.1 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 14.6.4, on page 14-31](#) and [Section 14.7.2, on page 14-46](#) 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 2.1 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 i960 RM/RN I/O processor contains the predefined 64 byte header registers plus additional configuration registers for device dependent operation (Section 14.15, on page 14-70).

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 01H for a PCI-to-PCI bridge.

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.

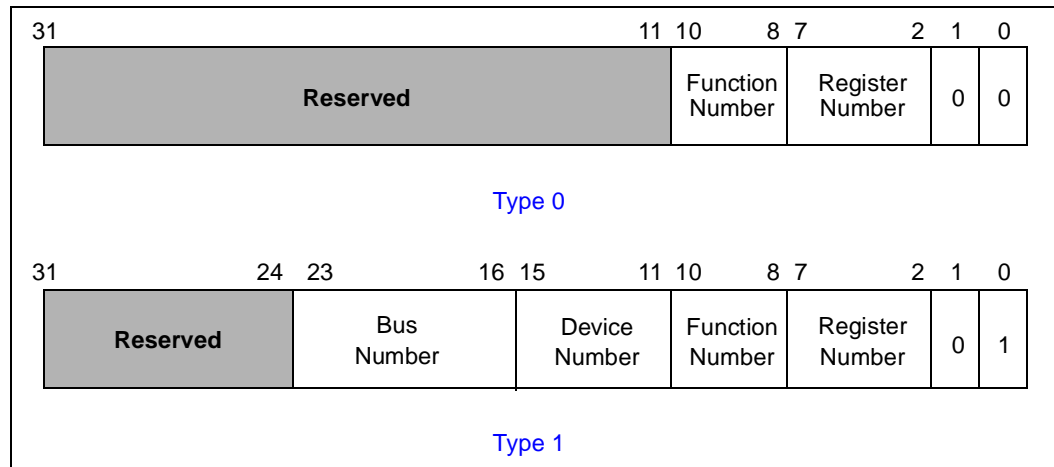
There are two classes of targets for PCI configuration commands:

- 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 ignores all configuration reads and Type 0 configuration writes on the secondary interface.

Configuration commands are only be accepted on the primary interface if the Configuration Cycle Retry bit within the Extended Bridge Command Register (EBCR, [Section 14.15, on page 14-70](#)) is cleared. If the Configuration Cycle Retry bit is set, the primary PCI interface signals a Retry on all Type 1 and Type 0 configuration commands.

All configuration commands are 32-bit only and therefore do not use the 64-bit extensions of both the primary and secondary PCI bus interfaces. See [Section 14.6.3, on page 14-26](#) for complete details of 64-bit operation. In addition, the i960 RM/RN I/O processor does not support bursting during Type 0 or Type 1 configuration cycles. Type 0 and Type 1 configuration writes are disconnected after the first 32-bit data phase. Type 1 configuration reads (handled as delayed transactions) can read a maximum of one 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
Type 0 - Bridge Ignores	Type 0 - Bridge Ignores Config Reads - 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: 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: Bus number = SBNR and Address not equal to 0XXXXXFF01H	Type 1 converted to a Special Cycle on Primary Side if: Bus number = PBNR and Address = 0XXXXXFF01H
Type 1 converted to a Special Cycle on Secondary Side if: Bus number is equal to SBNR and Address equals 0XXXXXFF01H	

### 14.4.1 Type 0 Commands

If address bits P\_AD[1:0] are 00<sub>2</sub>, 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 claims 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<sub>2</sub> all during the address phase. The primary interface ignores any configuration command (P\_IDSEL active) where P\_AD[1:0] are not 00<sub>2</sub> (Section 14.4.2, on page 14-9 for the case of 01<sub>2</sub>). 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 only responds to 000<sub>2</sub> in this bit field and ignore all other bit combinations. (Refer to Section 15.2.4, “PCI Multi-Function Device Swapping/Disabling” on page 15-22 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<sub>2</sub>, 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's 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, on page 14-82](#)), 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<sub>2</sub> on the primary interface, bits 31:11 are not asserted on the secondary interface.

In addition, the Secondary IDSEL Select Register (SISR, [Section 14.15, on page 14-70](#)) 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, on page 14-11](#) 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 <sub>2</sub>
00001	0000 0000 0000 0010 0000 0 <sub>2</sub>
00010	0000 0000 0000 0100 0000 0 <sub>2</sub>
00011	0000 0000 0000 1000 0000 0 <sub>2</sub>
00100	0000 0000 0001 0000 0000 0 <sub>2</sub>
00101	0000 0000 0010 0000 0000 0 <sub>2</sub>
00110	0000 0000 0100 0000 0000 0 <sub>2</sub>
00111	0000 0000 1000 0000 0000 0 <sub>2</sub>
01000 <sub>2</sub>	0000 0001 0000 0000 0000 0 <sub>2</sub>
01001 <sub>2</sub>	0000 0010 0000 0000 0000 0 <sub>2</sub>
01010 <sub>2</sub>	0000 0100 0000 0000 0000 0 <sub>2</sub>
01011 <sub>2</sub>	0000 1000 0000 0000 0000 0 <sub>2</sub>
01100 <sub>2</sub>	0001 0000 0000 0000 0000 0 <sub>2</sub>
01101 <sub>2</sub>	0010 0000 0000 0000 0000 0 <sub>2</sub>
01110 <sub>2</sub>	0100 0000 0000 0000 0000 0 <sub>2</sub>
01111 <sub>2</sub>	1000 0000 0000 0000 0000 0 <sub>2</sub>
10000 <sub>2</sub> - 11111 <sub>2</sub>	0000 0000 0000 0000 0000 0 <sub>2</sub>

### 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, 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 generates 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 is 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<sub>2</sub>
- Function Number is all ones - S\_AD[10:8] = 111<sub>2</sub>
- Register Number is all zeros - S\_AD[7:2] = 00000<sub>2</sub>
- 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 generates a Type 1 on the primary side with exactly the same information as on the secondary side. This Type 1 command is 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 is 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 no Master-Abort occurs (bit 13 of the Secondary Status Register is *not* set). The data for the Special Cycle on the secondary interface is 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 is 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 no Master-Abort occurs (bit 13 of the Primary Status Register is *not* set). The data for the Special Cycle on the primary interface is 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 i960 RM/RN 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 *is not* part of the standard secondary PCI address space and therefore is not configured by the system host processor. These devices are hidden from PCI configuration software but are accessible from the i960 RM/RN I/O processor Secondary Address Translation Unit. See [Chapter 15, “Address Translation Unit”](#) for a complete description of the private PCI address space implementation.

In Type 0 commands on the secondary interface,  $S\_AD[31:11]$  are used to select the IDSEL input of the target device. 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. 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 5 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 9 through 0 (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 0000<sub>2</sub> - 0100<sub>2</sub> 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 0 is set, S\_AD[16] is never 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 the primary interface receives a Type 1 command that intends to use one of the S\_AD address lines reserved for private PCI devices, the bridge performs the Type 1 to Type 0 conversion but does not assert the reserved S\_AD address line. The Type 0 command is then ignored on the secondary PCI bus.

By using the SISR register and the 5 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**

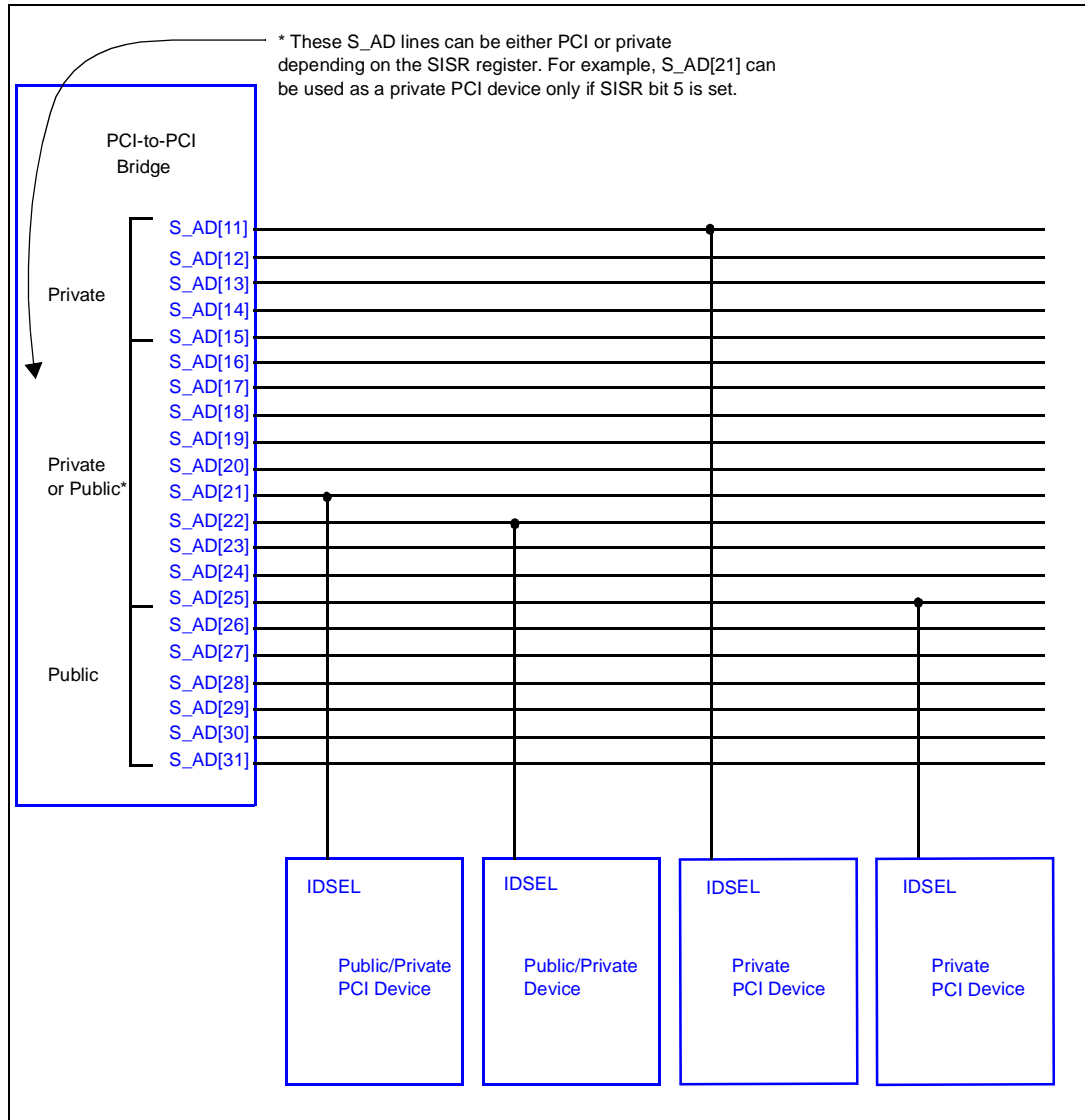
Primary Address P_AD[15:11]	Secondary Addresses S_AD[31:11] with All SISR Bits = 0	Secondary IDSEL Select Register Bits 9 - 0	Secondary Addresses S_AD[31:11] with SISR Bits Programmed
0000	0000 0000 0000 0001 0000 0 <sub>2</sub>	XXXXXXXXX <sub>2</sub>	0000 0000 0000 0000 0000 0 <sub>2</sub>
0001	0000 0000 0000 0010 0000 0 <sub>2</sub>	XXXXXXXX1X <sub>2</sub>	0000 0000 0000 0000 0000 0 <sub>2</sub>
00010	0000 0000 0000 0100 0000 0 <sub>2</sub>	XXXXXXXX1XX <sub>2</sub>	0000 0000 0000 0000 0000 0 <sub>2</sub>
00011	0000 0000 0000 1000 0000 0 <sub>2</sub>	XXXXXX1XXX <sub>2</sub>	0000 0000 0000 0000 0000 0 <sub>2</sub>
00100	0000 0000 0001 0000 0000 0 <sub>2</sub>	XXXXX1XXXX <sub>2</sub>	0000 0000 0000 0000 0000 0 <sub>2</sub>
00101	0000 0000 0010 0000 0000 0 <sub>2</sub>	XXXX1XXXXX <sub>2</sub>	0000 0000 0000 0000 0000 0 <sub>2</sub>
00110	0000 0000 0100 0000 0000 0 <sub>2</sub>	XXX1XXXXXX <sub>2</sub>	0000 0000 0000 0000 0000 0 <sub>2</sub>
00111	0000 0000 1000 0000 0000 0 <sub>2</sub>	XX1XXXXXXXX <sub>2</sub>	0000 0000 0000 0000 0000 0 <sub>2</sub>
0100 <sub>2</sub>	0000 0001 0000 0000 0000 0 <sub>2</sub>	X1XXXXXXXXX <sub>2</sub>	0000 0000 0000 0000 0000 0 <sub>2</sub>
01001 <sub>2</sub>	0000 0010 0000 0000 0000 0 <sub>2</sub>	1XXXXXXXXXX <sub>2</sub>	0000 0000 0000 0000 0000 0 <sub>2</sub>

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 neither initiates nor accepts 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.

## 14.5 Address Decoding

The i960 RM/RN 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 i960 RM/RN 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.

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 does 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. ([Chapter 15, “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 forwards an I/O transaction from the primary to secondary interface, that has an address within the address range defined (inclusively) by the IOBR and 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 i960 RM/RN 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 is not 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 2.1.

I/O Read and I/O Write transactions with invalid byte enables (those that are inconsistent with the byte address) are transparently passed by the bridge. In this case, it is expected that the target — target-aborts, and the bridge passes 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’s 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)

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 ISA Enable bit is discussed in the following section.

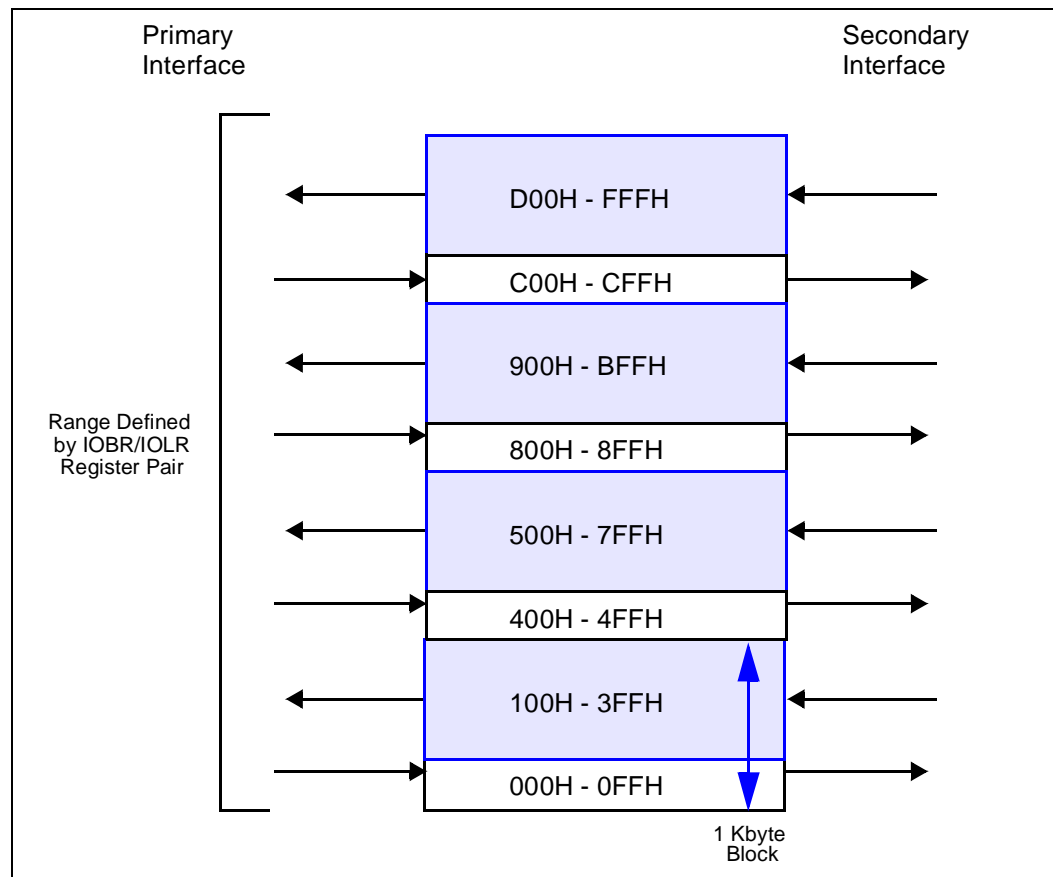
### 14.5.1.1 Disabling the I/O Address Range

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 i960 RM/RN I/O processor does 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 are forwarded upstream through the bridge.

### 14.5.1.2 ISA Mode

The PCI-to-PCI bridge unit of the i960 RM/RN 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 filters 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 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.

Figure 14-5. ISA Mode Address Decode

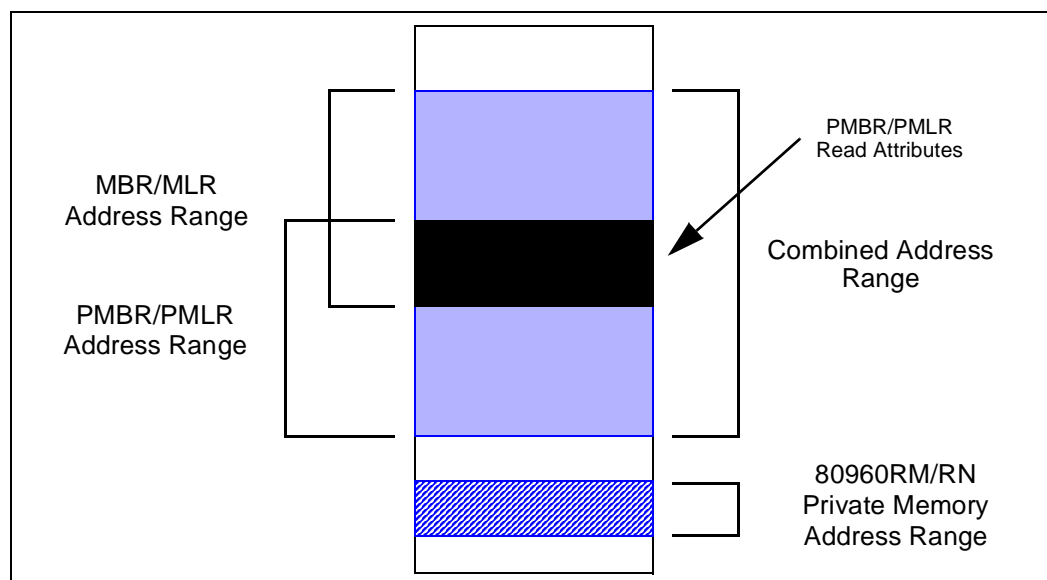


ISA I/O cards only decode the lower 10 bits of the address (1 Kbyte). The upper 768 bytes of the 1 Kbyte is assigned for general I/O. Because these cards do not decode the upper 6 bits of the 16-bit I/O address, the ISA address is aliased 64 times in the 64 Kbyte I/O address space. The combination of 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.

## 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 forwards *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) are 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 (Section 14.6.5.4) and within the non-prefetchable address space, *Memory Read Multiple* and *Memory Read Line* commands can be converted to *Memory Read* commands (Section 14.6.4).

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) are 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 forwards 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 (Section 14.5.3). Under certain conditions *Memory Write and Invalidate* commands can be converted to *Memory Write* commands (Section 14.6.5.4) and within the non-prefetchable address space, *Memory Read Multiple* and *Memory Read Line* commands can be converted to *Memory Read* commands (Section 14.6.4).

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 14.5.3 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)

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.

### 14.5.2.1 Burst Order

The bridge only supports linear incrementing addresses for burst order ( $AD[1:0] = 002$ ). For any other burst order, the Bridge disconnects the transaction after the first 32-bit data phase. See Section 14.8.1, “Delayed Read Transaction” on page 14-49 for information on non-linear MRL’s and MRM’s.

### 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 does 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.

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 i960 RM/RN I/O processor does not forward any transactions from the primary to the secondary. In this case, all Memory transactions from the secondary to the primary are forwarded upstream through the bridge.

### 14.5.3 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 when 64-bit PCI operation is enabled to maintain backwards compatibility to 32-bit PCI buses.

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 4GB address space. All DAC transactions are treated as prefetchable and adhere to the prefetch data amounts defined in [Table 14-13 on page 14-34](#).

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 does 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 2.1, is shown in [Figure 14-7](#). 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<sub>2</sub>) 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 14-7](#) 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.

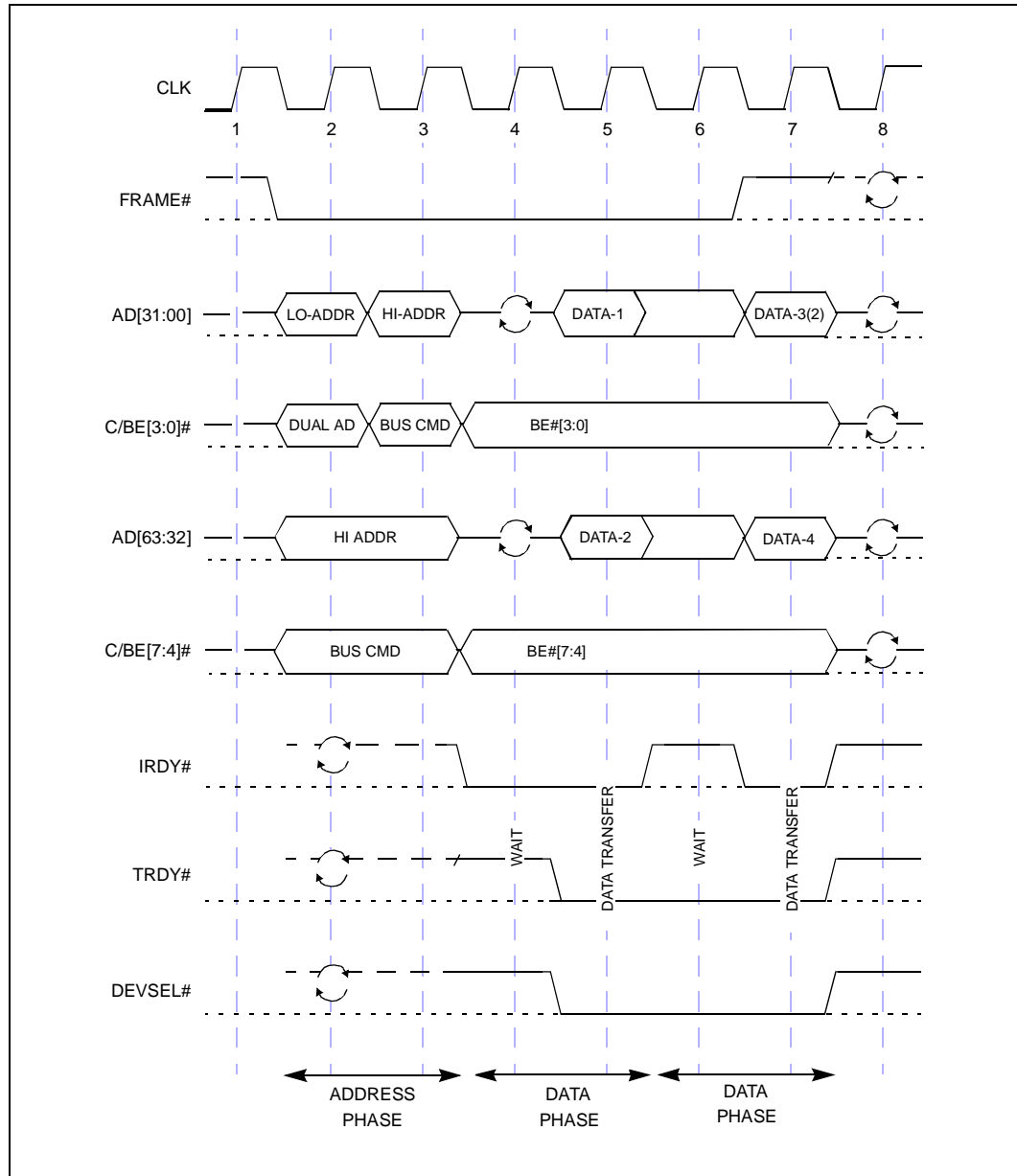
The response to DAC commands on the secondary interface may be modified by the following register bit from the bridge configuration space:

- 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 does not accept *any DAC write transactions at all*.

Figure 14-7. 64-bit Dual Address Read Cycle



## 14.5.4 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 appears 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 i960 RM/RN I/O processor to private devices, or transactions from private device to private device. The bridge does 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 does 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, the bridge does not claim these transactions. This is true for all outbound transactions from the any ATU or DMA channel since the i960 RM/RN I/O processor is never a master and slave on the secondary bus during the same cycle.

For 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 does 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.

## 14.5.5 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 4KB 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 does not claim the first 4KB of it's inbound address space, allowing the bridge the opportunity. In addition, setting this bit enables the bridge to act as a master on the primary interface and the PATU/MU to act as a slave on the primary interface at the same time. The i960 RM/RN I/O processor is not a master and a slave at the same time on any other interface. This bit is contained in the ATUCR in the ATU configuration space ([Chapter 15, "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 occurs or the transaction is claimed by the incorrect target.
- Normal Upstream read prefetch behavior applies. The Messaging Unit disconnects (as a 32-bit device) after delivering one 32-bit Dword.



## 14.5.6 Address Decode Summary

Four tables in this section contain a summary of the address decode options:

- One pair of tables summarize how addresses are decoded for Primary to Secondary transactions
- One pair of tables 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	Primary to Secondary		
	In MBR/MLR range	In PMBR/PMLR range	Outside all valid ranges
0	Ignore	Ignore	Ignore
1	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	Primary to Secondary			
		In IOBR/IOLR range	In ISA range (Lower 256 bytes)	In ISA range (Upper 768 bytes)	Outside all valid ranges
0	X	Ignore			
1	0	Forward	Not Valid	Not Valid	Ignore
1	1	Forward	Forward	Ignore	Ignore

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	Secondary to Primary				
			In MBR/MLR range	In PMBR/PMLR range	In SMBR/SMLR range	In ATU Inbound Address range	Outside all valid ranges
0	X	X	Ignore				
1	0	0	Forward	Forward	Not Valid	Ignore	Forward
1	0	1	Ignore	Ignore	Not Valid	Ignore	Forward
1	1	0	Forward	Forward	Ignore	Ignore	Forward
1	1	1	Ignore	Ignore	Ignore	Ignore	Forward

Table 14-6 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	Secondary to Primary				
				In IOBR/IOLR range	In SIOBR/SIOLR range	In ISA range (Lower 256 bytes)	In ISA range (Upper 768 bytes)	Outside all valid ranges
0	X	X	X	Ignore				
1	0	0	0	Forward	Not Valid	Not Valid	Not Valid	Forward
1	0	1	0	Forward	Not Valid	Not Valid	Not Valid	Forward
1	1	0	0	Ignore	Not Valid	Not Valid	Not Valid	Forward
1	1	1	0	Forward	Not Valid	Ignore	Forward	Forward
1	0	0	1	Forward	Ignore	Not Valid	Not Valid	Forward
1	0	1	1	Forward	Ignore	Not Valid	Not Valid	Forward
1	1	0	1	Ignore	Ignore	Not Valid	Not Valid	Forward
1	1	1	1	Forward	Ignore	Ignore	Forward	Forward

## 14.6 Bridge Operation

The bridge unit of the i960 RM/RN 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 <sub>2</sub>	Interrupt Acknowledge	Ignore	Ignore
0001 <sub>2</sub>	Special Cycle	Ignore	Ignore
0010 <sub>2</sub>	I/O Read	Forward	Forward
0011 <sub>2</sub>	I/O Write	Forward	Forward
0100 <sub>2</sub>	Reserved	Ignore	Ignore
0101 <sub>2</sub>	Reserved	Ignore	Ignore
0110 <sub>2</sub>	Memory Read	Forward	Forward
0111 <sub>2</sub>	Memory Write	Forward	Forward
1000 <sub>2</sub>	Reserved	Ignore	Ignore
1001 <sub>2</sub>	Reserved	Ignore	Ignore
1010 <sub>2</sub>	Configuration Read	Forward	Ignore
1011 <sub>2</sub>	Configuration Write	Forward	Forward (Type 1 Only)
1100 <sub>2</sub>	Memory Read Multiple	Forward	Forward
1101 <sub>2</sub>	Dual Address Cycle	Ignore	Forward
1110 <sub>2</sub>	Memory Read Line	Forward	Forward
1111 <sub>2</sub>	Memory Write and Invalidate	Forward	Forward

### 14.6.1 PCI Interfaces

The i960 RM/RN 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 is the interface connected to the lower numbered PCI bus between the two PCI buses that the i960 RM/RN 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 2.1 and the *PCI-to-PCI Bridge Architecture Specification* Revision 1.0.

### 14.6.1.2 Secondary Interface

The secondary PCI interface of the bridge unit is the interface connected to the higher numbered PCI bus between the two PCI buses that the i960 RM/RN 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 Local Bus Specification and the *PCI Local Bus Specification* Revision 2.1.

## 14.6.2 Claiming a PCI Transaction

The PCI-to-PCI bridge unit, as a target on the initiating bus, uses medium timing to assert DEVSEL# to claim a bus transaction. This meets the PCI specification of claiming a transaction within five clocks of the assertion of FRAME# by the initiating PCI device. The bridge target interface claims 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 signals 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 2.1 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 starts 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 ([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<sub>2</sub> which forces a minimum granularity for the timer of 8 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<sub>2</sub> and 00000000<sub>2</sub> 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 2.1 requirement for no more than 16 clocks of latency between the PCI address and the first data word. All PCI read and configuration transactions (except for Special Cycle) complete as a Delayed transaction.

The bridge processes 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 are processed as Posted transactions (default state). If Posting Disable bit is set, Memory Write commands are processed as Delayed transactions and Memory Write and Invalidate commands are 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 initiates 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 terminates with the same termination as the target bus transaction. For example, if the target bus transaction terminated with Disconnect, the Delayed Completion transaction terminates with Disconnect.

The bridge has a discard timer associated with each data buffer used for Delayed transactions (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 i960 RM/RN I/O processor are capable of PCI 64-bit operation to support data transfer rates of up to 264 MBytes/sec. 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) independently sample their respective REQ64# signals. If this signal is low, the bus is 64-bit capable and the respective master state machines attempt to complete all memory transactions as 64 bit cycles. See [Section 14.13.3](#) 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 is typically 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 is attempted as a 64-bit transaction. The previous statements also apply to read transactions.

### 14.6.3.1 64-Bit Protocol

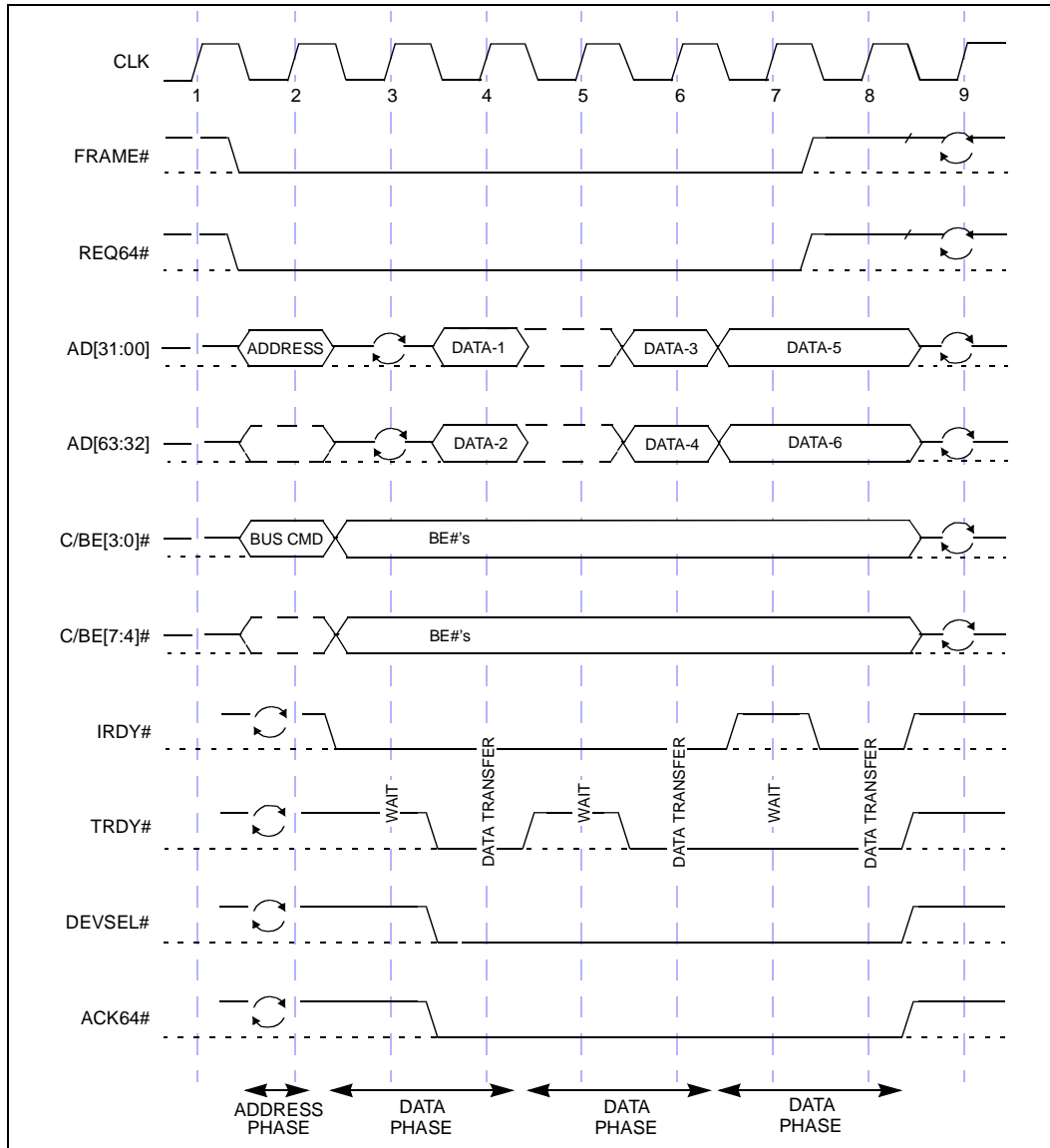
The 64-bit PCI extensions have been developed to coincide with the existing 32-bit protocol. The additional 32 bits of address/data require an additional four byte enables and a parity signal to cover them. The bus timing, protocol, and turn-around cycles behave exactly the same for the 64-bit signals 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 i960 RM/RN 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 asserts 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, uses 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 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 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 14-8](#) 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 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 14-8. 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 asserts and deasserts 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# remains 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#:

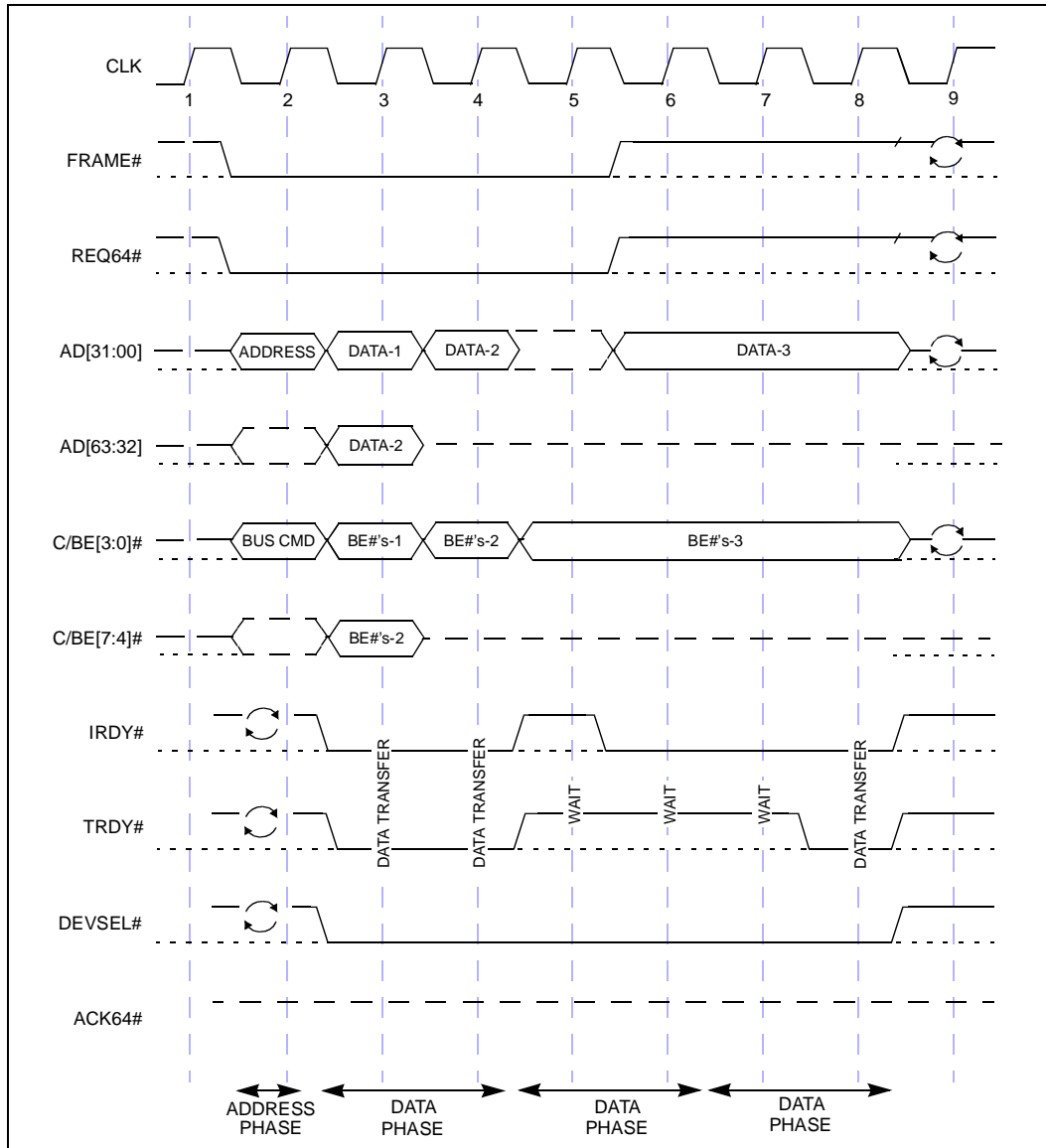
1. ACK64# deasserted but a burst can be sustained
2. ACK64# deasserted but a burst can not be sustained

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 14-9](#) for a diagram of this transaction. As a note, a disconnect after the first data phase of the burst transfer write results 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 does 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 bridge unit 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). 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 14.6.4](#) 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 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 14-9. 64-Bit Write Request with 32-Bit Transfer



As a PCI-to-PCI bridge, the i960 RM/RN I/O processor may be in a system environment with 64-bit devices on one side of the bridge and 32-bit devices on the other side of the bridge. 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 2.1 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 never returns ACK64# and always performs 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 asserts REQ64#. If the initiator starts a transaction with A2=1, the target interface does 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 returns 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 PCI Read Transactions

The i960 RM/RN 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 2.1 for full details on memory and I/O read transactions. Prefetchable Memory read commands are attempted as 64-bit transactions ([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, “Queue Operation” on page 14-42](#) 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 are 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 signals 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 then initiates 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 signals a Retry without latching any information.

Two requests 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 does not compare byte enables for all prefetchable transactions that have linear addresses. Byte enables are compared for prefetchable transactions that are non-linear.

If the request is accepted as a delayed transaction, the bridge retries 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 stores 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 accepts one 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 signals a Disconnect on the target bus.

The amount of data that 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 aliases the command to *Memory Read* on the target interface. For the purposes of matching MRL/MRM in the non-prefetchable address space, the bridge matches 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**

PCI Command	Prefetchable		Non-Prefetchable	
	Downstream	Upstream	Downstream	Upstream
Memory Read	In PMBR/PMLR Address Range	Upstream Prefetch Enable bit = 1 in EBCR	In MBR/MLR Address Range	Upstream Prefetch Enable bit = 0 in EBCR
Memory Read Line			In MBR/MLR Address Range <sup>1</sup>	Upstream Prefetch Enable bit = 0 in EBCR <sup>2</sup>
Memory Read Multiple				
DAC Read	N/A		N/A	Upstream Prefetch Enable bit = 0 in EBCR <sup>3</sup>

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 treats 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 depends 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 <sup>1</sup> = 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 <sup>2</sup>	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 ([Section 14.15.35](#))

Table 14-13. Upstream Memory Read Prefetch Size

Read Command	Prefetchable Memory Address Space		Non-Prefetchable Memory Address Space	
	CLS <sup>1</sup> = 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 <sup>2</sup>	Up to 64 Bytes <sup>3</sup>	Up to 4 Bytes	Up to 4 Bytes
Memory Read Multiple	Up to 128 Bytes	Up to 128 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 ([Section 14.15.35](#))
3. Up to 128 Bytes if the Upstream MRL Prefetch Size Bit is set in the Queue Control Register ([Section 14.15.35](#))

If the value in the CLS is anything other than 8 or 16, the read prefetch behavior is that of a CLS value of 8 (32 bytes).

The MRL control bits within the Queue Control Register are capable of promoting the prefetch size of the *Memory Read Line* Command to 2x the amount in the 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 4 bytes for these transaction types. The bridge signals 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 terminates the Delayed Completion transaction with:

- Completion termination if the transaction on the target bus terminated normally.
- Master-Abort termination or 1's (the number of 1's passed back, either 32-bit or 64-bit, is based on the PCI bus size of the initiating master, and in the 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 is discarded upon transaction completion from the DRC queue. The bridge does *not* follow the termination rules above when it reads more data than is requested. The bridge terminates with Completion termination if the initiator requests less data words than the bridge read from the target. For example, if the bridge reads 8 Dwords from the target and is terminated with a Disconnect while the initiator only reads four DWORDs, the bridge terminates 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 performs 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 only returns 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 ([Section 14.11.4](#)).

Under *PCI Local Bus Specification* Revision 2.1, the initiator must repeat the read transaction exactly with the same address, byte enables, REQ64#, and command or the bridge treats the transaction as a new request which results in a deadlock condition. To support PCI Specification, rev. 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 Read Command Alias Bit in the QCR register is set, the bridge does not distinguish the read commands on transactions which were 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 Read Command 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 Read Command 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 ( $AD[1:0] = 002$ ). For a non-linear ( $AD[1:0]$  do not equal 002) Memory Read transaction, the bridge fetches and transfers 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 converts the transaction to an MR on the target bus, fetch and transfer one Dword of data, and then signal a Disconnect to the initiator.

### 14.6.4.1 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. A target termination by the PCI slave on the target bus or the completion of the prefetch data size overrides the programmed value (if necessary) and allows the retrying master access to the data in the DRC 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 only occurs 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 (Table 14-11).
- The bridge unit reads beyond the prefetch read sizes to accommodate read streaming.
- Read streaming stops if the target performs a disconnect, the master terminates, or the 4 KB read boundary is reached.
- The bridge unit never inserts target or master D-D waitstates 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's target interface inserts up to 16 target waitstates (from the master's 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.
- Waitstates are inserted until read low watermark in the DRC queue. The read low watermark is defined by QCR bits [9:8] and defaults to two Qwords.
- Once the read low watermark is reached the initiating interface asserts 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 asserts STOP# signaling a Retry to the initiator.
- This mechanism is used for all prefetchable read transactions crossing the bridge.

### 14.6.4.2 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 signals a Disconnect on the target bus.



## 14.6.5 PCI Write Transactions

The i960 RM/RN 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 2.1* for full details on memory and I/O write transactions. Memory write commands are attempted as 64-bit transactions (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.

### 14.6.5.1 Delayed Write Transactions

A Delayed write transaction is very similar to a Delayed read transaction. The bridge claims 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 then signals a Retry to the initiator.

Delayed write transactions are limited to one data cycle of 4 bytes for I/O and configuration and 4 bytes 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 the write data from the initiating bus to the target bus. The bridge keeps 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 accepts 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 uses 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 (Section 14.10.1.4, on page 14-55).
- 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 treats the transaction as a new request. If the initiator does not repeat the write transaction, the data and associated information may be discarded (Section 14.11.4).

### 14.6.5.2 Posted Write Transactions

In a posted write transaction, the bridge accepts the write data and asserts 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 transfers 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 holds 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
  - 20 bytes of data in data queue
  - 4 bytes of address (1 entry) in address queue
2. PMW 2
  - 74 bytes of data in data queue
  - 4 bytes for address (1 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 4 transactions:

1. PMW 1
  - 4 bytes for address (1 entry) in address queue
  - 4 bytes of data in data queue
2. PMW 2
  - 4 bytes for address(1 entry) in address queue
  - 8 bytes of data in data queue
3. PMW 3
  - 4 bytes for address in address queue
  - 64 bytes of data in data queue
4. PMW 4
  - 4 bytes for address (1 entry) in address queue
  - 128 bytes of data in data queue

For a total of 204 bytes of data in the data queue and 4 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 signals 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 fills 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 inserts target waitstates (deassertion of TRDY#) or master waitstates (deassertion of IRDY#) to support the sustaining of a streaming transaction through the bridge. The target interface of the bridge may insert master waitstates to guarantee the transfer of an entire cacheline during Memory Write and Invalidate transfers. See [Section 14.6.5.4, “Memory Write and Invalidate Command”](#) on page 14-39.

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 ([Section 14.7.2, on page 14-46](#)), 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 signals 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.

### 14.6.5.3 Memory Write Command

A PCI initiator uses 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 only transfer one 32-bit PCI data phase. This means FRAME# is only asserted for one clock on the target interface and that the initiating interface signals a target Disconnect after the first data transfer. Refer to the *PCI Local Bus Specification* Revision 2.1 for more details.

### 14.6.5.4 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 only allows 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 at least a enough free queue space in the PMW data queue to accept an MWI transaction. If this is not true, the MWI is retried on the target bus. 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 performs 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 completes 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 completes 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 deasserts IRDY# (master waitstate) 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 master waitstates from the initiator on the initiating bus. (When master waitstates are 3 or greater, this 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 0 or if the Cacheline Size Register is programmed to a value other than 8 or 16. Refer to the *PCI Local Bus Specification* Revision 2.1 for the full details of a Memory Write and Invalidate command.

If posting is disabled, the bridge does not allow the MWI command to appear on the target bus. The bridge converts 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 accepts 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 aliases 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*.

#### 14.6.5.5 I/O Write Command

All I/O Write transactions are processed as Delayed transactions. The i960 RM/RN 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 claims 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.

#### 14.6.5.6 Write Boundary

The bridge of the i960 RM/RN 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 signals 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.

### 14.6.5.7 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 attempts 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 attempts 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

### 14.6.5.8 Fast Back to Back Transactions

The i960 RM/RN 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 i960 RM/RN 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 5 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 i960 RM/RN I/O processor bridge architecture is defined in [Section 14.7.2](#). Refer to the *PCI Local Bus Specification* Revision 2.1, 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 i960 RM/RN I/O processor bridge unit.

**Table 14-14. Bridge Unit Queue**

Queue Mnemonic	Queue Name	Queue Size	Transactions Possible in Queue <sup>1</sup>
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	128 Bytes	MR, MRL, MRM
U_DRC1	Upstream Delayed Read Completion 1	128 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 8 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:

- 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 ([Section 14.15.24](#)). 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 (Table 14-14 “Bridge Unit Queue” on page 14-42).

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	128 Byte Queue
Memory Read Line	128 Byte Queue	128 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., 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 remains 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 only disconnects the read in response to a disconnect on the target bus during the DRC completion cycle on 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 uses D\_TRQ3 and U\_DWC uses U\_TRQ3. The DWC queue holds all *I/O Write* and *Configuration Write* data. In addition, if write posting is disabled all *Memory Write* and *Memory Write and Invalidate* commands are delayed and uses the initiating bus DWC Queue.

During the write request cycle on the initiating bus, the slave interface of the bridge claims 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 is 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 i960 RM/RN I/O processor are listed in [Table 14-17](#).

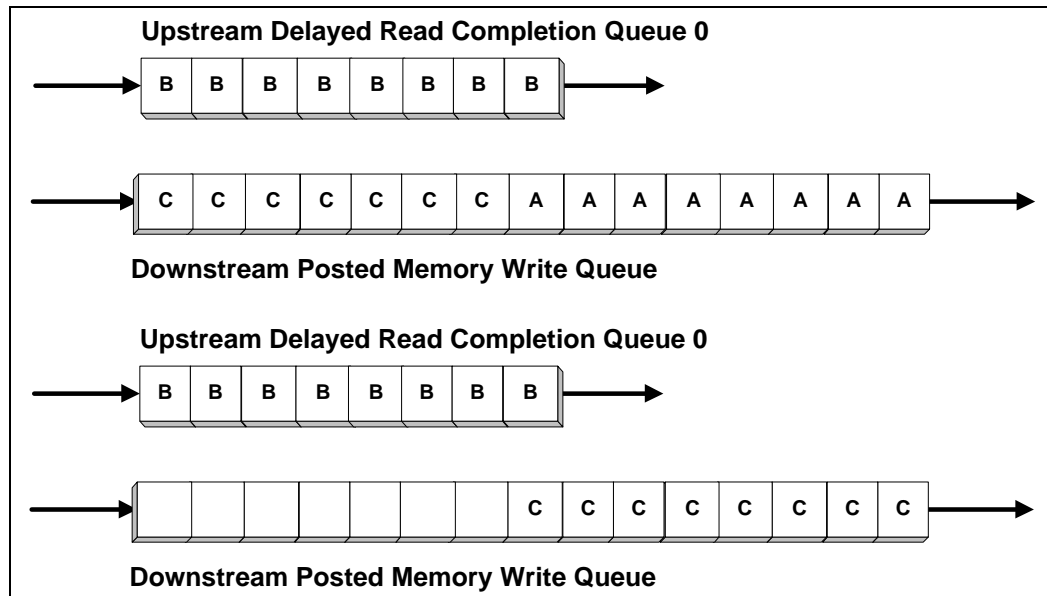
**Table 14-17. Bridge Transaction Ordering Rules**

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	No
Delayed Read Completion (DRC)	No	Yes	Yes	Yes	Yes
Delayed Write Completion (DWC)	Yes	Yes	No	Yes	No

These transaction ordering rules define the base line operation for the way in which data moves in both directions through the PCI-to-PCI Bridge. In [Table 14-17](#) a **NO** response in a box means that 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 that 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 2.1*.

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.

Figure 14-10. Downstream Data Path Queue Completion



In Figure 14-10, 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 completes 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 is completed. If the secondary interface of the bridge acquires the secondary bus first, Transaction C is completed. 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.

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 head of the respective queue, the question in column 2 is asked.
2. Based on the answer in column 3, 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 3, the Action in column 4 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 <sup>1</sup>
	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 <sup>1</sup>
	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 <sup>1</sup>
	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

1. Allow previous Transaction to Complete

## 14.8 Bridge Data Flow

The bridge allows transactions to cross both PCI buses through the i960 RM/RN 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 the returning read data in a Delayed Read Completion Queue. The original PCI master on the initiating bus continuously retries the read transaction until the slave interface on the initiating bus claims the transaction and returns the read data present in the DRC Queue. The data flow for a delayed read transaction is summarized in the following statements:

- 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 the transaction to master-abort and assert SERR#. This assumes parity checking is enabled and SERR# assertion is enabled.

**Note:** SERR# is not asserted on the secondary bus interface, only on the primary bus interface, see [Section 14.11.1](#)).

- If the transaction is inside an address window and a Transaction Queue is not available or it is inside an address window, A cycle match occurs but the read data is not ready (!DRC\_Ready), then retry the transaction.
- If the transaction is inside an address window, a cycle match occurs, the 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 continues 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 initiates 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 requests 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 initiates 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 asserts 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 re-attempts 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 reads 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 continues 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's master interface encounters a 4 Kbyte boundary.
- If parity checking is enabled and a data parity error is detected, the master interface asserts 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 compares 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 claims 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 requests the PCI bus when after address and data from the request cycle have been received and ordering has been satisfied ([Section 14.7.2](#)). Once the bus is granted, the target PCI interface writes 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 returns 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 continues 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 signals a disconnect to the master and return to idle.
- If a data parity error is detected, the slave interface asserts 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 requests the PCI bus when posted write transaction address is at the head of the PMW Queue and transaction ordering has been satisfied ([Section 14.7.2](#)). Once the bus is granted, the target PCI interface writes 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 transfers 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, leaves 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 <b>Secondary Locking</b> when locked DRR received on primary interface
Secondary Locking	Restricted	All new downstream transactions retried.	Unrestricted	Accept all transactions on interface	Move to <b>Primary Locking</b> when bridge has finished mastering the locked request on the secondary interface. If an abort occurs while mastering this transaction the bridge transitions directly to the <b>Unlocking</b> 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 <b>Locked</b> state. If the DRC's discard timer expires before the transaction is completed, the bridge transitions directly to the <b>Unlocking</b> 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 releases the S_LOCK# signal and transition to the <b>Unlocking</b> state.
Unlocking	Restricted	All new downstream transactions retried	Restricted	All upstream transactions retried	Moves to the <b>Unlocked</b> 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 (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 transitions 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 i960 RM/RN I/O processor's 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's 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 2.1 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 uses 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 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 releases 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, terminates a PCI transaction under the different situations described in the following sections.

#### 14.10.1.1 Completion

The target interface completes the transaction in response to a completion on the initiating interface. Completion termination occurs when the initiator bus has FRAME# and IRDY# deasserted. FRAME# is always deasserted during the second to last data transfer of a transaction. Refer to the *PCI Local Bus Specification* Revision 2.1.

#### 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 (Section 14.6.2.1). A normal termination occurs 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 retains ownership until an entire cacheline has been transferred from the PMW Queues. Refer to the *PCI Local Bus Specification* Revision 2.1.

#### 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 i960 RM/RN 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 1's (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 (assuming the write is still occurring) within three clocks of the Master-Abort on the target interface. 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 holds 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 does not wait for the final data transfer. The following sections summarize the bridge's actions as a PCI target for termination situations. See the *PCI Local Bus Specification* Revision 2.1 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 signals 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 sets the Target Abort (master) bit in the target bus's status register (PSR or SSR) and the Target Abort (target) bit in the initiating bus's 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 is not reflected back to the master and the Target Abort (target) bit in the initiating bus's status register is not 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 signals a Target-Abort to the initiator on the initiating bus. The bridge sets the Target Abort (master) bit in the target bus's status register (PSR or SSR) and the Target Abort (target) bit in the initiating bus's status register.

If the posted write transaction is complete on the initiating interface, the bridge asserts P\_SERR# (if enabled) on the primary interface indicating a system error. The bridge also sets the Target Abort (master) bit in the target bus's 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 2.1 without special considerations for a bridge.

The following sections detail the bridge's 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 does 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 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 i960 RM/RN 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 does *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 i960 RM/RN 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 does *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 these cases, the bridge forwards 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 are passed to the opposite interface whenever possible. This enables the parity error recovery mechanisms outlined in the *PCI Local Bus Specification* Revision 2.1 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 asserts PERR# on the target interface. The bridge passes the bad data and the bad parity to the initiating interface and bus where the initiator also detects the bad parity and data and assert PERR# on the initiating bus. The bridge sets 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 asserts PERR#. No other action is taken by the bridge unit.

Specifically for downstream reads (initiated by a master on the primary bus interface), the i960 RM/RN 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 the 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 ([Section 14.6.4](#)), 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 i960 RM/RN 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 the 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 ([Section 14.6.4](#)), 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 the bridge unit delivering the 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 delays 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 i960 RM/RN I/O processor's 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 is 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 is 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 an 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 retries the transaction on the initiating bus and the bridge returns 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 retries the transaction with no other response. A new transaction is not enqueued due to queue architecture constraints ([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 does not detect a match for the completion attempt and retries the transaction. In this case the transaction is most likely never completed, and the enqueued data is eventually 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 asserts 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 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 retries the transaction with no other response. A new transaction is not enqueued due to queue architecture constraints (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 does not detect a match for the completion attempt and retries the transaction. In this case the transaction is most likely never completed, and the enqueued data is eventually 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 asserts 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's 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 always performs 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 timer is 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. The timer starts 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 timer expires, the completion transaction is discarded and P\_SERR# is optionally asserted on the primary bus.

There are 8 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 08 in the BCR) or  $2^{15}$  (enabled by clearing bit 08 in the BCR). Delayed cycles initiated from the secondary bus interface have a programmable discard value of  $2^{10}$  (enabled by setting bit 09 in the BCR) or  $2^{15}$  (enabled by clearing bit 08 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 do not count and delayed completion transactions remain in their respective queues until retrieved by a PCI master.

## 14.11.5 PCI-to-PCI Bridge Error Summary

The following tables summarize 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 06
	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 06
Data Parity Error Detected - bit 08	Upstream Delayed Read Data Parity Error on Primary Bus	Primary Parity Error Response Enable - bit 06
	Upstream Delayed Write Data Parity Error During Completion Cycle on Primary Bus	Primary Parity Error Response Enable - bit 06
	Upstream Posted Memory Write Data Parity Error on the Primary Bus	Primary Parity Error Response Enable - bit 06

**Table 14-20. PSR Error Reporting Summary (Sheet 2 of 3)**

<b>Error Bit in Primary Status Register (PSR)</b>	<b>Error Condition</b>	<b>Qualifying Bit in Primary Command Register (PCR Unless Otherwise Noted)</b>
Signaled System Error - bit 14	Address Parity Error on Primary Interface	Primary Parity Error Response Enable - bit 06 SERR# Enable - bit 08
	Address Parity Error on Secondary Interface	Secondary Parity Error Response Enable - bit 00 (BCR) SERR# Enable - bit 08 SERR# Forwarding - bit 01 (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 06 Secondary Parity Error Response Enable - bit 00 (BCR) SERR# Enable - bit 08
	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 06 Secondary Parity Error Response Enable - bit 00 (BCR) SERR# Enable - bit 08
	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 08
	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 08
	Downstream Posted Memory Write that Ends in a Master Abort on the Secondary Bus	Master Abort Mode - bit 05 (BCR) SERR# Enable - bit 08
	Upstream Posted Memory Write that Ends in a Master Abort on the Primary Bus	Master Abort Mode - bit 05 (BCR) SERR# Enable - bit 08
	1 of 8 Discard Timers Expire	Discard Timer SERR# Enable - bit 11 (BCR) SERR# Enable - bit 08
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 00
	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 00
Data Parity Error Detected - bit 08	Downstream Delayed Read Data Parity Error on Secondary Bus	Secondary Parity Error Response Enable - bit 00
	Downstream Delayed Write Data Parity Error During Completion Cycle on Secondary Bus	Secondary Parity Error Response Enable - bit 00
	Downstream Posted Memory Write Data Parity Error on the Secondary Bus	Secondary Parity Error Response Enable - bit 00
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 00 (BCR)

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)
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
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 Primary and Secondary Clocking

The P\_CLK clock input provides the only clock source for the i960 RM/RN I/O processor PCI-to-PCI bridge. It is the system designer's responsibility to provide clock sources to all secondary devices which are synchronous to the primary input clock.

Refer to the *PCI Local Bus Specification* Revision 2.1 for details on PCI bus clock specifications.

## 14.13 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 2.1 and the *PCI-to-PCI Bridge Architecture Specification* Revision 1.0.



### 14.13.1 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 secondary interface S\_RST# output is asynchronous with respect to the secondary S\_CLK signal and therefore there are no synchronization issues with the internal reset signal crossing clock domains from the primary side to the secondary side. To support this, the path from both the primary P\_RST# input and the Secondary Bus Reset bit to the secondary PCI bus S\_RST# output must be combinatorial. The bridge does not take any action if the 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 i960 RM/RN 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](#)” on page 14-107

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, “Secondary PCI Bus Arbitration Parking”](#) on page 17-7 for details on bridge parking.

### 14.13.2 Configuring the Bridge

For the bridge unit to operate in a system environment, several things must be properly initialized. The procedure outlined 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 the 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 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 is 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 is passed to the primary bus. Likewise, if the Memory Enable bit in the PCR is not set, all memory accesses on the secondary bus is forwarded to the primary bus.
- 5) The CLSR, PLTR, and SLTR must be set to appropriate values before the bridge is fully functional. Most systems want the SERR# Enable bits in the PCR and the BCR registers set as well.
- 6) The Configuration Cycle Retry Bit in [Section 14.15.24, “Extended Bridge Control Register - EBCR”](#) on page 14-96 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 i960 RM/RN 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.

### 14.13.3 64-Bit Bus Configuration

At i960 RM/RN 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 attempts 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

## 14.14 Powerup/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 14.13.1 for details on resetting the PCI-to-PCI bridge unit.

## 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 80960 core processor operations. Figure 14-11 describes the entire bridge PCI configuration space.

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 80960 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 Rev. 1.0. Registers with offsets greater than 3EH are implementation specific to the i960 RM/RN I/O processor.

See Chapter 1, “Introduction” for definitions of *reserved*, *read only*, and *read/clear*. All registers adhere to the definitions found in the *PCI Local Bus Specification* Revision 2.1 and the *PCI-to-PCI Bridge Architecture Specification* Revision 1.0 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, “Memory-Mapped Registers”.

The i960 core processor reads 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 80960 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 returns 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**

Bridge Configuration Header				PCI Address Offset
Device ID		Vendor ID		00H
Primary Status		Primary Command		04H
Class Code			Revision ID	08H
<b>Reserved</b>	Header Type	Primary Latency Timer	Cacheline Size	0CH
<b>Reserved</b>				10H
<b>Reserved</b>				14H
Secondary Latency Timer	Subordinate Bus Number	Secondary Bus Number	Primary Bus Number	18H
Secondary Status		I/O Limit	I/O Base	1CH
Memory Limit		Memory Base		20H
Prefetchable Memory Limit		Prefetchable Memory Base		24H
<b>Reserved</b>				28H
<b>Reserved</b>				2CH
<b>Reserved</b>				30H
Subsystem ID		Subsystem Vendor ID		34H
<b>Reserved</b>				38H
Bridge Control		<b>Reserved</b>		3CH
Secondary IDSEL Control		Extended Bridge Control		40H
Primary Bridge Interrupt Status				44H
Secondary Bridge Interrupt Status				48H
Secondary Arbitration Control				4CH
PCI Interrupt Routing Control				50H
<b>Reserved</b>		Secondary I/O Limit	Secondary I/O Base	54H
Secondary Memory Limit		Secondary Memory Base		58H
Queue Control		Secondary Decode Enable		5CH
Centralized Discard Timer				60H

↑

PCI-to-PCI Bridge

↓

↑

80960RM/RN Processor Specific

↓

**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 14-73
1002H	Section 14.15.2, "Device ID Register - DIDR" on page 14-74
1004H	Section 14.15.3, "Primary Command Register - PCR" on page 14-75
1006H	Section 14.15.4, "Primary Status Register - PSR" on page 14-76
1008H	Section 14.15.5, "Revision ID Register - RID" on page 14-77
1009H	Section 14.15.6, "Class Code Register - CCR" on page 14-77
100CH	Section 14.15.7, "Cacheline Size Register - CLSR" on page 14-78
100DH	Section 14.15.8, "Primary Latency Timer Register - PLTR" on page 14-79
100EH	Section 14.15.9, "Header Type Register - HTR" on page 14-80
1018H	Section 14.15.10, "Primary Bus Number Register - PBNR" on page 14-81
1019H	Section 14.15.11, "Secondary Bus Number Register - SBNR" on page 14-82
101AH	Section 14.15.12, "Subordinate Bus Number Register - SubBNR" on page 14-83
101BH	Section 14.15.13, "Secondary Latency Timer Register - SLTR" on page 14-84
101CH	Section 14.15.14, "I/O Base Register - IOBR" on page 14-85
101DH	Section 14.15.15, "I/O Limit Register - IOLR" on page 14-86
101EH	Section 14.15.16, "Secondary Status Register - SSR" on page 14-87
1020H	Section 14.15.17, "Memory Base Register - MBR" on page 14-88
1022H	Section 14.15.18, "Memory Limit Register - MLR" on page 14-89
1024H	Section 14.15.19, "Prefetchable Memory Base Register - PMBR" on page 14-90
1026H	Section 14.15.20, "Prefetchable Memory Limit Register - PMLR" on page 14-91
1034H	Section 14.15.21, "Bridge Subsystem Vendor ID Register - BSVIR" on page 14-92
1036H	Section 14.15.22, "Bridge Subsystem ID Register - BSIR" on page 14-92
103EH	Section 14.15.23, "Bridge Control Register - BCR" on page 14-93
1040H	Section 14.15.24, "Extended Bridge Control Register - EBCR" on page 14-96
1042H	Section 14.15.25, "Secondary IDSEL Select Register - SISR" on page 14-99
1044H	Section 14.15.26, "Primary Bridge Interrupt Status Register - PBISR" on page 14-101
1048H	Section 14.15.27, "Secondary Bridge Interrupt Status Register - SBISR" on page 14-102
104CH	Section 14.15.28, "Secondary Arbitration Control Register - SACR" on page 14-103
1050H	Section 14.15.29, "PCI Interrupt Routing Select Register - PIRSR" on page 14-103
1054H	Section 14.15.30, "Secondary I/O Base Register - SIOBR" on page 14-103
1055H	Section 14.15.31, "Secondary I/O Limit Register - SIOLR" on page 14-104
1058H	Section 14.15.32, "Secondary Memory Base Register - SMBR" on page 14-105
105AH	Section 14.15.33, "Secondary Memory Limit Register - SMLR" on page 14-106
105CH	Section 14.15.34, "Secondary Decode Enable Register - SDER" on page 14-107
105EH	Section 14.15.35, "Queue Control Register - QCR" on page 14-109

### 14.15.1 Vendor Identification Register - VIDR

Vendor ID Register bits adhere to the definitions in the *PCI Local Bus Specification* Revision 2.1.

**Table 14-24. Vendor Identification Register - VIDR**

PCI Configuration Offset 00 - 01H	Internal Bus Address 0000 1000H	Attribute Legend: RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set 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 2.1.

**Table 14-25. Device Identification Register - DIDR (80960RN)**

PCI Configuration Offset 02 - 03H	Internal Bus Address 0000 1002H	Attribute Legend: RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set	RC = Read Clear RO = Read Only NA = Not Accessible
Bit	Default	Description	
15:00	0964H	Device ID - This is a 16-bit value assigned to the i960 RM/RN I/O processor. This register, combined with the VID, uniquely identify any PCI device.	

**Table 14-26. Device Identification Register - DIDR (80960RM)**

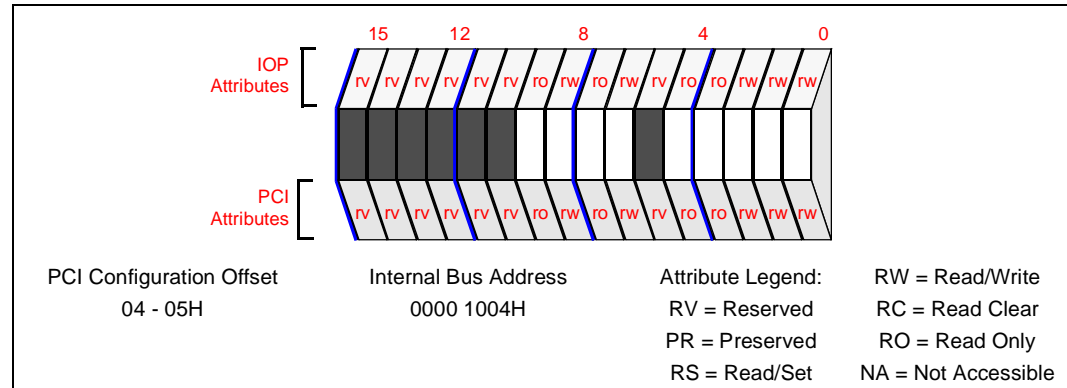
PCI Configuration Offset 02 - 03H	Internal Bus Address 0000 1002H	Attribute Legend: RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set	RC = Read Clear RO = Read Only NA = Not Accessible
Bit	Default	Description	
15:00	0962H	Device ID - This is a 16-bit value assigned to the i960 RM/RN I/O processor. 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 2.1 and in most cases affect the behavior of the primary interface of the PCI-to-PCI bridge.

**Table 14-27. Primary Command Register - PCR**

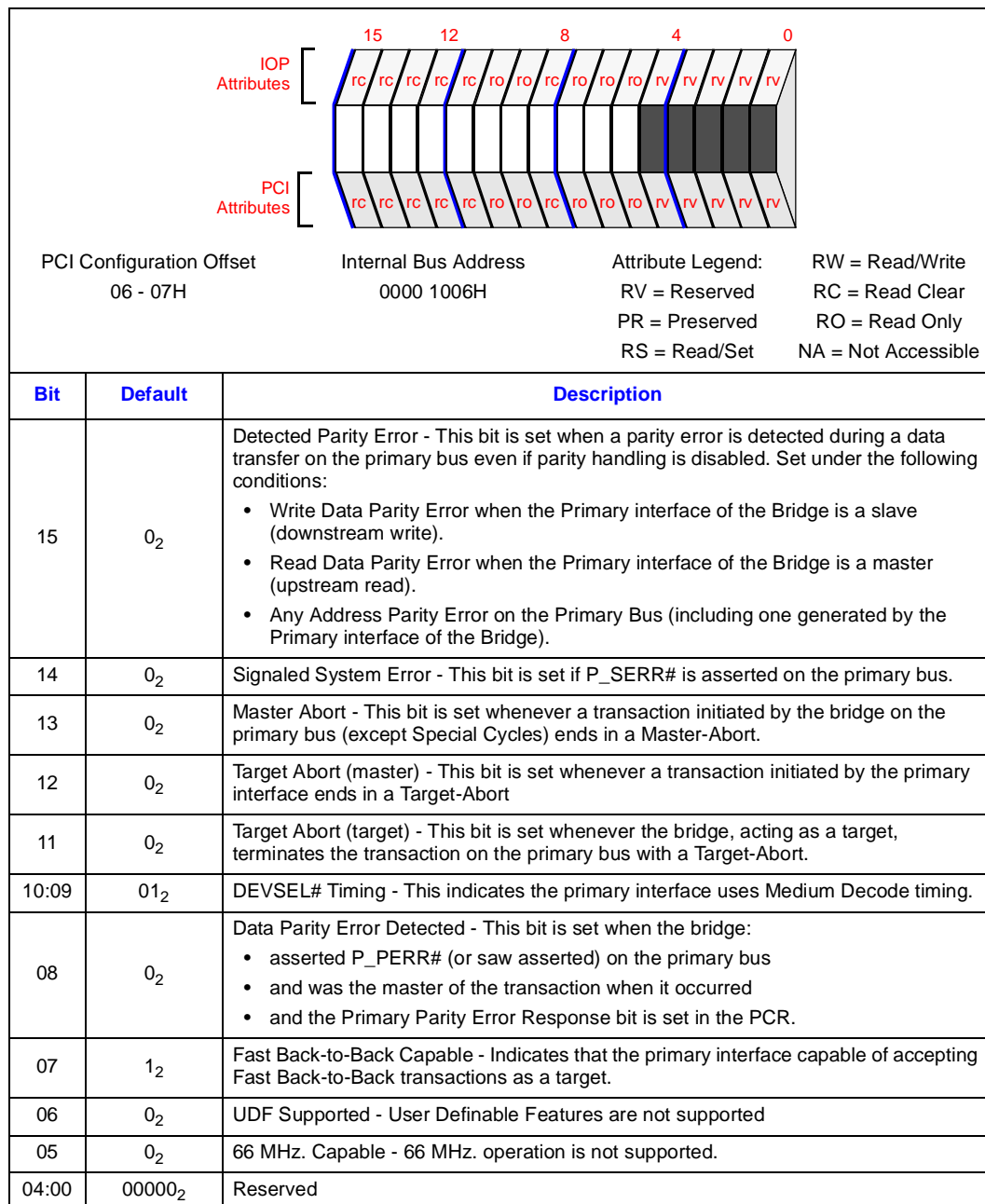
Bit	Default	Description
15:10	00000 <sub>2</sub>	Reserved
09	0 <sub>2</sub>	Fast Back to Back Enable - This primary interface does not perform fast back to back transactions.
08	0 <sub>2</sub>	SERR# Enable - If this bit is cleared, the i960 RM/RN I/O processor is not allowed to assert P_SERR# on its primary interface.
07	0 <sub>2</sub>	Wait Cycle Control - controls address/data stepping. Not implemented and a reserved bit field
06	0 <sub>2</sub>	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 <sub>2</sub>	VGA Palette Snoop Enable - VGA Palette Snooping is not supported.
04	0 <sub>2</sub>	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 <sub>2</sub>	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 <sub>2</sub>	Bus Master Enable - Controls the bridge's ability to operate as a master on the primary interface for memory and I/O transactions. This bit does not affect the bridge's 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 does not claim all memory or I/O transactions on the secondary PCI interface.
01	0 <sub>2</sub>	Memory Enable - Controls the bridge's response to both memory and prefetchable memory accesses. If this bit is cleared, the bridge does not respond to any memory access on the primary PCI interface.
00	0 <sub>2</sub>	I/O Space Enable - Controls the bridges response to I/O transactions on the primary PCI interface. If this bit is cleared, the bridge does 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 2.1 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 or by writing a 1<sub>2</sub> to the register.

Figure 14-12. Primary Status Register - PSR





### 14.15.5 Revision ID Register - RID

Revision ID Register bits adhere to definitions in the *PCI Local Bus Specification* Revision 2.1.

**Table 14-28. Revision Identification Register - RID**

PCI Configuration Offset	Internal Bus Address	Attribute Legend:	RW = Read/Write
08H	0000 1008H	RV = Reserved	RC = Read Clear
		PR = Preserved	RO = Read Only
		RS = Read/Set	NA = Not Accessible
Bit	Default	Description	
07:00	00H	Revision ID - This value identifies the revision number of the i960 RM/RN 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 2.1. 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	Internal Bus Address	Attribute Legend:	RW = Read/Write
09 - 0BH	0000 1009H	RV = Reserved	RC = Read Clear
		PR = Preserved	RO = Read Only
		RS = Read/Set	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 <i>PCI-to-PCI Bridge Architecture Specification</i> 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 2.1 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 behaves as if a value of 0 was written.

**Table 14-30. Cacheline Size Register - CLSR**

PCI Configuration Offset	Internal Bus Address	Attribute Legend:
0CH	0000 100CH	RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set
		RC = Read Clear RO = Read Only 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 2.1 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	Internal Bus Address 0000 100DH	Attribute Legend: RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set RC = Read Clear RO = Read Only NA = Not Accessible
Bit	Default	Description
07:03	00000 <sub>2</sub>	Programmable Latency Timer - This portion of the register varies the latency timer for the primary interface from a minimum of 0 clocks to a maximum of 248 clocks.
02:00	000 <sub>2</sub>	Latency Timer Granularity - These bits are read only giving a programmable granularity of 8 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 2.1. 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 1.0. (Refer to [Section 15.2.4, “PCI Multi-Function Device Swapping/Disabling”](#) on page 15-22 for exceptions to this statement.)

**Table 14-32. Header Type Register- HTR**

PCI Configuration Offset	Internal Bus Address	Attribute Legend:	RW = Read/Write
0EH	0000 100EH	RV = Reserved	RC = Read Clear
		PR = Preserved	RO = Read Only
		RS = Read/Set	NA = Not Accessible
Bit	Default	Description	
7	1 <sub>2</sub>	Single Function/Multi-Function Device - This bit identifies whether or not the i960 RM/RN I/O processor is a single or multi-function PCI device. The i960 RM/RN I/O processor is considered a multi-function device.	
06:00	000001 <sub>2</sub>	PCI Header Type - This bit field tells the system initialization code what type of PCI header is implemented. The i960 RM/RN I/O processor has a PCI-to-PCI bridge header as defined in Rev. 1 of the bridge architecture specification.	

### 14.15.10 Primary Bus Number Register - PBNR

Primary Bus Number Register bits adhere to the definitions in the *PCI Local Bus Specification* Revision 2.1. 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	Internal Bus Address	Attribute Legend:
18H	0000 1018H	RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set
		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's 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	Internal Bus Address	Attribute Legend:	RW = Read/Write
19H	0000 1019H	RV = Reserved	RC = Read Clear
		PR = Preserved	RO = Read Only
		RS = Read/Set	NA = Not Accessible
Bit	Default	Description	
07:00	00H	Secondary Bus Number - This field is programmed with the PCI bus number of the bridge's 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 2.1. 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	Internal Bus Address	Attribute Legend:
1AH	0000 101AH	RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set
		RC = Read Clear 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 2.1 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	Internal Bus Address	Attribute Legend:	RW = Read/Write
1BH	0000 101BH	RV = Reserved	RC = Read Clear
		PR = Preserved	RO = Read Only
		RS = Read/Set	NA = Not Accessible
Bit	Default	Description	
07:03	00000 <sub>2</sub>	Programmable Latency Timer - This portion of the register varies the latency timer for the secondary interface from a minimum of 0 clocks to a maximum of 248 clocks.	
02:00	000 <sub>2</sub>	Latency Timer Granularity - These bits are read only giving a programmable granularity of 8 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 2.1. 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 are not accepted by the primary side of the bridge, even if the address falls within the defined I/O address range.

**Table 14-37. I/O Base Register - IOBR**

PCI Configuration Offset 1CH	Internal Bus Address 0000 101CH	Attribute Legend: RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set 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 2.1. 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 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 are not accepted by the primary side of the bridge, even if the address falls within the defined I/O address range.

**Table 14-38. I/O Limit Register - IOLR**

PCI Configuration Offset	Internal Bus Address	Attribute Legend:
1DH	0000 101DH	RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set
		RC = Read Clear RO = Read Only NA = Not Accessible
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 2.1 (with modifications made to bit 14 by the *PCI-to-PCI Bridge Architecture Specification* Revision 1.0) 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<sub>2</sub> to the bit location.

**Table 14-39. Secondary Status Register - SSR**

Bit	Default	Description
15	0 <sub>2</sub>	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: <ul style="list-style-type: none"> <li>Write Data Parity Error when the Secondary interface of the Bridge is a slave (upstream write).</li> <li>Read Data Parity Error when the Secondary interface of the Bridge is a master (downstream read).</li> <li>Any Address Parity Error on the Secondary Bus (including one generated by the Secondary interface of the Bridge).</li> </ul>
14	0 <sub>2</sub>	Received System Error - When set indicates that S_SERR# was detected by the bridge on the secondary interface.
13	0 <sub>2</sub>	Master Abort - This bit is set whenever a transaction initiated by the secondary interface (except Special Cycles) ends in Master-Abort
12	0 <sub>2</sub>	Target Abort (master) - This bit is set whenever a transaction initiated by the secondary interface ends in a Target-Abort.
11	0 <sub>2</sub>	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 <sub>2</sub>	DEVSEL# Timing - Medium Decode Timing for the secondary interface
08	0 <sub>2</sub>	Data Parity Error Detected - This bit is set when the bridge: <ul style="list-style-type: none"> <li>asserted S_PERR# (or saw asserted) on the secondary bus</li> <li>and was the master of the transaction when it occurred</li> <li>and the Secondary Parity Error Response bit is set in the BCR.</li> </ul>
07	1 <sub>2</sub>	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 <sub>2</sub>	UDF Supported - This indicates that User Definable Features is not supported
05	0 <sub>2</sub>	66 MHz Capable - This indicates that 66 MHz operation is not supported
04:00	00000 <sub>2</sub>	Reserved

### 14.15.17 Memory Base Register - MBR

Memory Base Register bits adhere to the definitions in the *PCI Local Bus Specification* Revision 2.1. 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 is aligned on a 1 Mbyte boundary.

**Table 14-40. Memory Base Register - MBR**

		PCI Configuration Offset 20 - 21H	Internal Bus Address 0000 1020H	Attribute Legend: RW = Read/Write RV = Reserved RC = Read Clear PR = Preserved RO = Read Only RS = Read/Set NA = Not Accessible
Bit	Default	Description		
15:04	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 2.1. 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 FFFFH. This forces a 1 Mbyte granularity on the memory address range.

**Table 14-41. Memory Limit Register - MLR**

PCI Configuration Offset	Internal Bus Address	Attribute Legend:	RW = Read/Write
22 - 23H	0000 1022H	RV = Reserved	RC = Read Clear
		PR = Preserved	RO = Read Only
		RS = Read/Set	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 is aligned on a 1 Mbyte boundary.

**Table 14-42. Prefetchable Memory Base Register - PMBR**

PCI Configuration Offset	Internal Bus Address	Attribute Legend:	RW = Read/Write
24 - 25H	0000 1024H	RV = Reserved	RC = Read Clear
		PR = Preserved	RO = Read Only
		RS = Read/Set	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 are 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 FFFFFFFH. This forces a 1 Mbyte granularity on the memory address range.

**Table 14-43. Prefetchable Memory Limit Register - PMLR**

PCI Configuration Offset	Internal Bus Address	Attribute Legend:	RW = Read/Write
26 - 27H	0000 1026H	RV = Reserved	RC = Read Clear
		PR = Preserved	RO = Read Only
		RS = Read/Set	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 Bridge Subsystem Vendor ID Register - BSVIR

Bridge Subsystem Vendor ID Register bits adhere to the definitions in the *PCI Local Bus Specification* Revision 2.1.

**Table 14-44. Bridge Subsystem Vendor ID Register - BSVIR**

		PCI Configuration Offset 34 - 35H	Internal 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			
15:0	0000H	Subsystem Vendor ID - This register uniquely identifies the vendor of the add-in board or subsystem			

### 14.15.22 Bridge Subsystem ID Register - BSIR

Bridge Subsystem ID Register bits adhere to the definitions in the *PCI Local Bus Specification* Revision 2.1.

**Table 14-45. Bridge Subsystem ID Register - BSIR**

		PCI Configuration Offset 36 - 37H	Internal Bus Address 0000 1036H	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	000H	Subsystem ID - This register uniquely identifies the add-in board or subsystem			

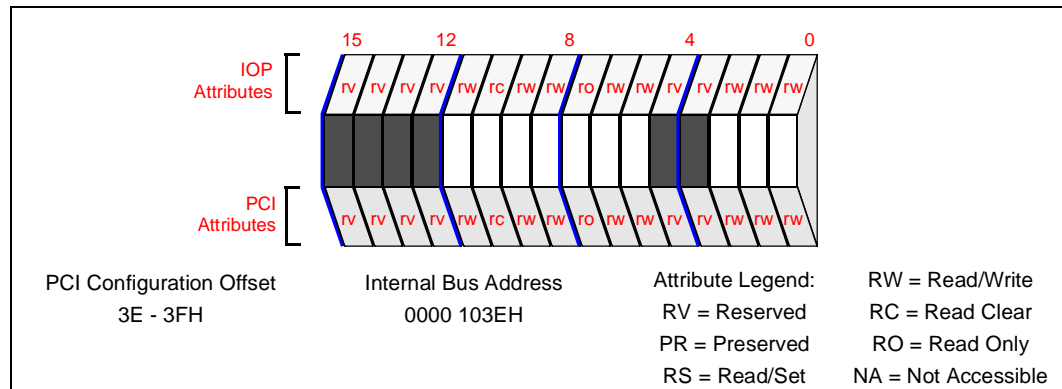


### 14.15.23 Bridge Control Register - BCR

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 14-46. Bridge Control Register - BCR (Sheet 1 of 3)**

Bit	Default	Description
15:12	00H	Reserved
11	0 <sub>2</sub>	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 <sub>2</sub>	Discard Timer Status - This bit indicates the status of the discard timers. A value of 0 indicates that no discard timers have expired. A value of 1 indicates that at least one of the 8 discard timers has expired.
09	0 <sub>2</sub>	Secondary Discard Timer Value - This bit controls the time-out value for the 4 discard timers attached to the queues holding data for transactions initiated on the secondary bus. A value of 0 indicates the time-out value is 2 <sup>15</sup> clocks. A value of 1 indicates the time-out value is 2 <sup>10</sup> clocks.
08	0 <sub>2</sub>	Primary Discard Timer Value - This bit controls the time-out value for the 4 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 <sup>15</sup> clocks. A value of 1 indicates the time-out value is 2 <sup>10</sup> clocks.
07	0 <sub>2</sub>	Fast Back to Back Enable - This secondary interface does not perform fast back to back transactions.



**Table 14-46. Bridge Control Register - BCR (Sheet 2 of 3)**

Bit	Default	Description
06	0 <sub>2</sub>	<p>Secondary Bus Reset - This bit controls the secondary bus S_RST# signal. When set:</p> <ul style="list-style-type: none"> <li>The PCI-to-PCI Bridge Unit resets 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 retries all transactions, except Type 0 configuration transactions, until this bit is cleared.</li> <li>DMA Channel 2 immediately halts any PCI transactions and gracefully complete any local bus transactions. It then returns to an idle state. DMA Channel 2 does not begin any new transfers until the Secondary Bus Reset bit is cleared.</li> <li>Secondary ATU immediately halts any PCI transactions and gracefully complete any local bus transactions. The i960 core processor is released from back-off, if necessary. The Secondary ATU does not accept any new i960 core processor requests until the Secondary Bus Reset bit is cleared. The Secondary ATU configuration registers are reset.</li> <li>An interrupt may be sent to the core processor based upon the setting of bit 3 in the SDER.</li> </ul> <p>When this bit is cleared, the S_RST# signal is deasserted. The software must clear this bit.</p>
05	0 <sub>2</sub>	<p>Master Abort Mode - This bit controls the bridge functionality whenever a Master-Absort termination occurs on either interface for transactions in which the bridge is the slave.</p> <p>When clear, reads 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 is accepted by the bridge and discarded.</p> <p>When set, the bridge signals a Master-Absort to the requesting master when the corresponding transaction on the other side of the bridge terminates with a Master-Absort 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).</p>
04	0 <sub>2</sub>	Reserved
03	0 <sub>2</sub>	VGA Enable - VGA Addressing is not supported.
02	0 <sub>2</sub>	<p>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)</p> <p>When set, the bridge does 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 are forwarded up the bridge if the address the last 768 bytes in each 1 Kbyte block.</p>

**Table 14-46. Bridge Control Register - BCR (Sheet 3 of 3)**

PCI Configuration Offset 3E - 3FH	Internal Bus Address 0000 103EH	Attribute Legend: RW = Read/Write RV = Reserved RC = Read Clear PR = Preserved RO = Read Only RS = Read/Set NA = Not Accessible
Bit	Default	Description
01	0 <sub>2</sub>	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 then asserts P_SERR# on the primary interface. When clear, S_SERR# assertions are not forwarded to the primary interface.
00	0 <sub>2</sub>	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 is 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.

### 14.15.24 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 1.0.

**Table 14-47. Extended Bridge Control Register - EBCR (Sheet 1 of 3)**

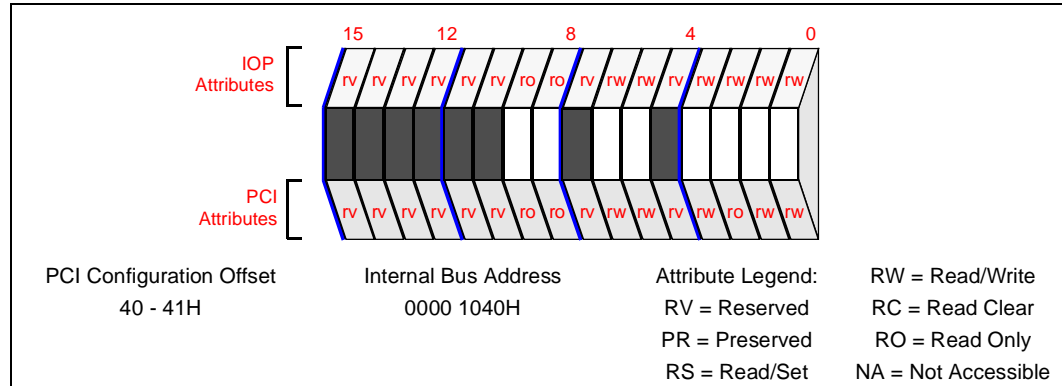
Bit	Default	Description
15:10	00000 <sub>2</sub>	Reserved
09	Varies with external state of S_REQ64# at secondary PCI bus reset	Secondary PCI Bus 64-Bit Capable - When clear, the secondary PCI bus interface has been configured as 64-bit capable by the assertion of S_REQ64# on the rising edge of S_RST#. 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 <sub>2</sub>	Reserved
06	0 <sub>2</sub>	Secondary DAC Medium Decode Enable - When set, DAC cycles on the secondary PCI interface of the bridge are 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 are claimed with subtractive decode timing and forwarded to the primary PCI interface.

PCI Configuration Offset 40 - 41H	Internal Bus Address 0000 1040H	Attribute Legend: RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set RC = Read Clear RO = Read Only NA = Not Accessible
--------------------------------------	------------------------------------	--

**Table 14-47. Extended Bridge Control Register - EBCR (Sheet 2 of 3)**

Bit	Default	Description
05	0 <sub>2</sub>	Reset Internal Bus - This bit controls the reset of the i960 core processor and all units on the internal bus. When set: <ul style="list-style-type: none"> <li>The PCI-to-PCI Bridge Unit is not reset. Upstream and downstream bridge I/O and memory transactions are unaffected during the IB reset.</li> <li>All current PCI transactions being mastered by the ATU and DMA are completed, and the ATU and DMA master interfaces proceed to an idle state. No additional transactions are mastered by these units until the IB reset is complete.</li> <li>All current transactions being slaved by the ATU on either the PCI bus or the internal bus completes, and the ATU slave interfaces proceeds to an idle state. All future slave transactions master abort, with the exception of the completion cycle for the transaction that set the Reset Internal Bus bit in the EBCR.</li> <li>If the value of the Core Processor Reset bit in the EBCR (upon normal reset) is set, the i960 core processor is held in reset when the IB reset is complete.</li> <li>The Bridge and the ATU ignores configuration cycles, and they appear as master aborts for: 32 PCI clocks + the number of PCI clocks needed to finish all ATU and DMA transactions that completes before the IB reset (as described in the above text).</li> <li>The i960 RM/RN I/O processor hardware clears this bit after the reset operation completes.</li> </ul>
04	0 <sub>2</sub>	Reserved
03	1 <sub>2</sub>	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 i960 RM/RN I/O processor responds to all configuration cycles with a Retry condition. When clear, the i960 RM/RN I/O processor responds 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 is required to clear this bit to deassert 80960 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.



**Table 14-47. Extended Bridge Control Register - EBCR (Sheet 3 of 3)**

PCI Configuration Offset 40 - 41H	Internal 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
00	0 <sub>2</sub>	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.)

## 14.15.25 Secondary IDSEL Select Register - SISR

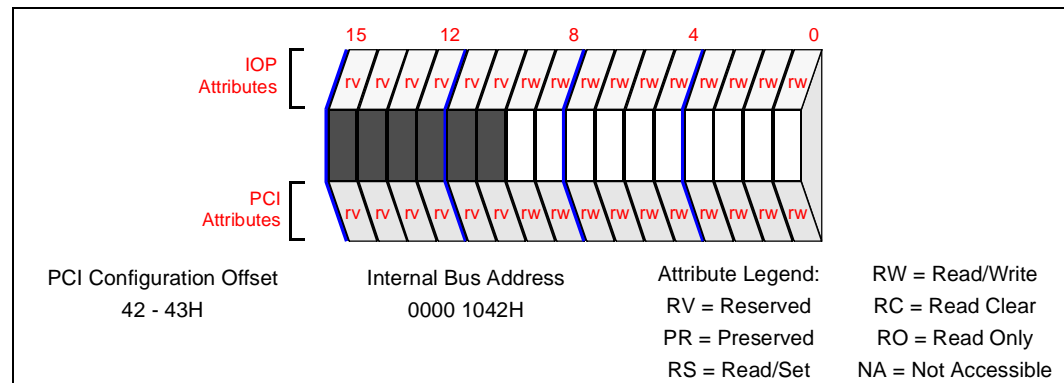
The Secondary IDSEL Select Register controls the usage of S\_AD[25:16] in Type 1 to Type 0 conversions from the primary to secondary interface. In default operation, a unique encoding on primary addresses P\_AD[15:11] results in the assertion of one bit on the secondary address bus S\_AD[31:16] during a Type 1 to Type 0 conversion (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.

**Note:** Please check the i960<sup>®</sup> RM/RN I/O Processor Specification Update for possible issues with the SISR.

**Table 14-48. Secondary IDSEL Select Register - SISR (Sheet 1 of 2)**

Bit	Default	Description
15:10	000000 <sub>2</sub>	Reserved
09	0 <sub>2</sub>	AD25- IDSEL Disable - When this bit is set, AD25 is deasserted for any possible Type 1 to Type 0 conversion. When clear, AD25 is asserted when primary addresses AD[15:11] = 01001 <sub>2</sub> during a Type 1 to Type 0 conversion.
08	0 <sub>2</sub>	AD24- IDSEL Disable - When this bit is set, AD24 is deasserted for any possible Type 1 to Type 0 conversion. When clear, AD24 is asserted when primary addresses AD[15:11] = 01000 <sub>2</sub> during a Type 1 to Type 0 conversion.
07	0 <sub>2</sub>	AD23 - IDSEL Disable - When this bit is set, AD23 is deasserted for any possible Type 1 to Type 0 conversion. When clear, AD23 is asserted when primary addresses AD[15:11] = 00111 <sub>2</sub> during a Type 1 to Type 0 conversion.
06	0 <sub>2</sub>	AD22 - IDSEL Disable - When this bit is set, AD22 is deasserted for any possible Type 1 to Type 0 conversion. When clear, AD22 is asserted when primary addresses AD[15:11] = 00110 <sub>2</sub> during a Type 1 to Type 0 conversion.
05	0 <sub>2</sub>	AD21 - IDSEL Disable - When this bit is set, AD21 is deasserted for any possible Type 1 to Type 0 conversion. When clear, AD21 is asserted when primary addresses AD[15:11] = 00101 <sub>2</sub> during a Type 1 to Type 0 conversion.
04	0 <sub>2</sub>	AD20 - IDSEL Disable - When this bit is set, AD20 is deasserted for any possible Type 1 to Type 0 conversion. When clear, AD20 is asserted when primary addresses AD[15:11] = 00100 <sub>2</sub> during a Type 1 to Type 0 conversion.



**Table 14-48. Secondary IDSEL Select Register - SISR (Sheet 2 of 2)**

PCI Configuration Offset 42 - 43H	Internal 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
03	0 <sub>2</sub>	AD19 - IDSEL Disable - When this bit is set, AD19 is deasserted for any possible Type 1 to Type 0 conversion. When clear, AD19 is asserted when primary addresses AD[15:11] = 00011 <sub>2</sub> during a Type 1 to Type 0 conversion.
02	0 <sub>2</sub>	AD18 - IDSEL Disable - When this bit is set, AD18 is deasserted for any possible Type 1 to Type 0 conversion. When clear, AD18 is asserted when primary addresses AD[15:11] = 00010 <sub>2</sub> during a Type 1 to Type 0 conversion.
01	0 <sub>2</sub>	AD17 - IDSEL Disable - When this bit is set, AD17 is deasserted for any possible Type 1 to Type 0 conversion. When clear, AD17 is asserted when primary addresses AD[15:11] = 00001 <sub>2</sub> during a Type 1 to Type 0 conversion.
00	0 <sub>2</sub>	AD16 - IDSEL Disable - When this bit is set, AD16 is deasserted for any possible Type 1 to Type 0 conversion. When clear, AD16 is asserted when primary addresses AD[15:11] = 00000 <sub>2</sub> during a Type 1 to Type 0 conversion.



## 14.15.26 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 i960 RM/RN I/O processor (Section 8.3, “PCI and Peripheral Interrupts” on page 8-18). 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 and bit 4 of the SDER. Refer to Section 14.15.34 for details.

**Table 14-49. Primary Bridge Interrupt Status Register - PBISR**

Bit	Default	Description
31:06	0000000H	Reserved
05	0 <sub>2</sub>	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: <ul style="list-style-type: none"> <li>Write Data Parity Error when the Primary interface of the Bridge is a slave (downstream write).</li> <li>Read Data Parity Error when the Primary interface of the Bridge is a master (upstream read).</li> <li>Any Address Parity Error on the Primary Bus (including one generated by the Primary interface of the Bridge).</li> </ul>
04	0 <sub>2</sub>	P_SERR# Asserted - This bit is set if P_SERR# is asserted on the primary PCI bus.
03	0 <sub>2</sub>	PCI Master Abort - This bit is set whenever a transaction initiated by the primary master interface ends in a Master-Abort.
02	0 <sub>2</sub>	PCI Target Abort (Master) - This bit is set whenever a transaction initiated by the primary master interface ends in a Target-Abort.
01	0 <sub>2</sub>	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 <sub>2</sub>	PCI Master Parity Error - The primary interface sets this bit when three conditions are met: <ol style="list-style-type: none"> <li>the bus agent asserted P_PERR# itself or observed P_PERR# asserted.</li> <li>the agent setting the bit acted as the bus master for the operation in which the error occurred.</li> <li>the Parity Checking Enable bit (PCR Register) is set.</li> </ol>

## 14.15.27 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 i960 RM/RN I/O processor (Section 8.3, “PCI and Peripheral Interrupts” on page 8-18). 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 14-50. Secondary Bridge Interrupt Status Register - SBISR**

Bit	Default	Description
31:07	0000000H	Reserved
06	0 <sub>2</sub>	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 <sub>2</sub>	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: <ul style="list-style-type: none"> <li>Write Data Parity Error when the Secondary interface of the Bridge is a slave (upstream write).</li> <li>Read Data Parity Error when the Secondary interface of the Bridge is a master (downstream read).</li> <li>Any Address Parity Error on the Secondary Bus (including one generated by the Secondary interface of the Bridge).</li> </ul>
04	0 <sub>2</sub>	Received System Error - This bit is set if S_SERR# is detected on the secondary PCI bus.
03	0 <sub>2</sub>	PCI Master Abort - This bit is set whenever a transaction initiated by the secondary master interface ends in a Master-Abort.
02	0 <sub>2</sub>	PCI Target Abort (Master) - This bit is set whenever a transaction initiated by the secondary master interface ends in a Target-Abort.
01	0 <sub>2</sub>	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 <sub>2</sub>	PCI Master Parity Error - Secondary interface sets this bit when three conditions are met: <ol style="list-style-type: none"> <li>bus agent asserted S_PERR# itself or observed S_PERR# asserted</li> <li>agent setting the bit acted as bus master for the operation in which the error occurred</li> <li>the Secondary Parity Error Response bit (BCR Register) is set</li> </ol>

## 14.15.28 Secondary Arbitration Control Register - SACR

Refer to Section 17.6.1, “Secondary Arbitration Control Register - SACR” on page 17-12 for a description of the Secondary Arbitration Control Register.

## 14.15.29 PCI Interrupt Routing Select Register - PIRSR

Refer to Section 8.5.4, “PCI Interrupt Routing Select Register - PIRSR” on page 8-39 for a description of the PCI Interrupt Routing Select Register.

## 14.15.30 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 14-51. Secondary I/O Base Register - SIOBR**

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.

PCI Configuration Offset 54H	Internal Bus Address 0000 1054H	Attribute Legend: RW = Read/Write RV = Reserved RC = Read Clear PR = Preserved RO = Read Only RS = Read/Set NA = Not Accessible
---------------------------------	------------------------------------	--

### 14.15.31 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 14-52. Secondary I/O Limit Register - SIOLR**

PCI Configuration Offset	Internal Bus Address	Attribute Legend:
55H	0000 1055H	RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set
		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.

### 14.15.32 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 is aligned on a 1 Mbyte boundary.

**Table 14-53. Secondary Memory Base Register - SMBR**

PCI Configuration Offset 58 - 59H	Internal Bus Address 0000 1058H	Attribute Legend: RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set	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 does 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	Internal Bus Address	Attribute Legend:	RW = Read/Write
5A - 5BH	0000 105AH	RV = Reserved	RC = Read Clear
		PR = Preserved	RO = Read Only
		RS = Read/Set	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 80960JT core.

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 (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)**

Bit	Default	Description
15	1 <sub>2</sub>	S_SERR# Detected Interrupt Mask - When set, detecting S_SERR# on the secondary interface resulting in bit 14 of the SSR being set does <i>not</i> result in bit 4 of the SBISR being set. When clear, an error that sets bit 14 of the SSR causes bit 4 of the SBISR to be set
14	1 <sub>2</sub>	Secondary PCI Master Abort Interrupt Mask - When set, a master abort error resulting in bit 13 of the SSR being set does <i>not</i> result in bit 3 of the SBISR being set. When clear, an error that sets bit 13 of the SSR causes bit 3 of the SBISR to be set.
13	1 <sub>2</sub>	Secondary PCI Target Abort (Master) Interrupt Mask- When set, a target abort error resulting in bit 12 of the SSR being set does <i>not</i> result in bit 2 of the SBISR being set. When clear, an error that sets bit 12 of the SSR causes bit 2 of the SBISR to be set.
12	1 <sub>2</sub>	Secondary PCI Target Abort (Target) Interrupt Mask - When set, a target abort error resulting in bit 11 of the SSR being set does <i>not</i> result in bit 1 of the SBISR being set. When clear, an error that sets bit 11 of the SSR causes bit 1 of the SBISR to be set.
11	1 <sub>2</sub>	Secondary PCI Master Parity Error Interrupt Mask - When set a parity error resulting in bit 8 of the SSR being set does <i>not</i> result in bit 0 of the SBISR being set. When clear, an error that sets bit 8 of the SSR causes bit 0 of the SBISR to be set.

PCI Configuration Offset 5C - 5DH	Internal Bus Address 0000 105CH	Attribute Legend: RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set	RC = Read Clear RO = Read Only NA = Not Accessible
--------------------------------------	------------------------------------	--	--

**Table 14-55. Secondary Decode Enable Register - SDER (Sheet 2 of 2)**

Bit	Default	Description
10	1 <sub>2</sub>	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 does <i>not</i> result in bit 4 of the PBISR being set. When clear, an error that sets bit 14 of the PSR causes bit 4 of the PBISR to be set
09	1 <sub>2</sub>	Primary PCI Master Abort Interrupt Mask - When set, a master abort error resulting in bit 13 of the PSR being set does <i>not</i> result in bit 3 of the PBISR being set. When clear, an error that sets bit 13 of the PSR causes bit 3 of the PBISR to be set.
08	1 <sub>2</sub>	Primary PCI Target Abort (Master) Interrupt Mask- When set, a target abort error resulting in bit 12 of the PSR being set does <i>not</i> result in bit 2 of the PBISR being set. When clear, an error that sets bit 12 of the PSR causes bit 2 of the PBISR to be set.
07	1 <sub>2</sub>	Primary PCI Target Abort (Target) Interrupt Mask - When set, a target abort error resulting in bit 11 of the PSR being set does <i>not</i> result in bit 1 of the PBISR being set. When clear, an error that sets bit 11 of the PSR causes bit 1 of the PBISR to be set.
06	1 <sub>2</sub>	Primary PCI Master Parity Error Interrupt Mask - When set a parity error resulting in bit 8 of the PSR being set does <i>not</i> result in bit 0 of the PBISR being set. When clear, an error that sets bit 8 of the PSR causes bit 0 of the PBISR to be set.
05	1 <sub>2</sub>	Secondary Detected Parity Error Bit Interrupt Mask - When set a parity error resulting in bit 15 of the SSR being set does not result in bit 5 of the SBISR being set.
04	1 <sub>2</sub>	Primary Detected Parity Error Bit Interrupt Mask - When set a parity error resulting in bit 15 of the PSR being set does not result in bit 5 of the PBISR being set.
03	1 <sub>2</sub>	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 is not set.
02	0 <sub>2</sub>	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.
01:00	00 <sub>2</sub>	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**

Bit	Default	Description
15:04	000H	Reserved
03	0 <sub>2</sub>	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 <sub>2</sub>	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.
01:00	00 <sub>2</sub>	Reserved.

PCI Configuration Offset 5E - 5FH	Internal Bus Address 0000 105EH	Attribute Legend:	RW = Read/Write
		RV = Reserved	RC = Read Clear
		PR = Preserved	RO = Read Only
		RS = Read/Set	NA = Not Accessible



This chapter describes the operation modes, setup, and implementation of the mechanism which interfaces between the primary and secondary PCI buses and the i960<sup>®</sup> RM/RN 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 i960 RM/RN 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 i960 RM/RN I/O Processor Internal Bus.

Transactions initiated on a PCI bus and targeted at the i960 RM/RN I/O Processor Internal Bus are referred to as *inbound transactions* (PCI to internal bus); transactions initiated on the i960 RM/RN 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 i960 RM/RN 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 i960 RM/RN 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 i960 RM/RN 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 mode is enabled, secondary PCI masters can access the MU by addressing anywhere in the first 4 K of the SATU directly.

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 i960 RM/RN I/O processor memory controller. Refer to the *PCI Local Bus Specification* Revision 2.1 for details of Expansion ROM usage.

The Primary and Secondary Address Translation Units 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 i960 RM/RN I/O processor. (Refer to [Section 15.2.4, “PCI Multi-Function Device Swapping/Disabling”](#), 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 extensions providing up to 264 Mbytes/sec of PCI bandwidth. On the internal interface, the Primary and Secondary ATU implement the i960 RM/RN I/O processor internal bus protocol which provides for a maximum of 528 Mbytes/sec using 64-bit/66 MHz signaling.

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**

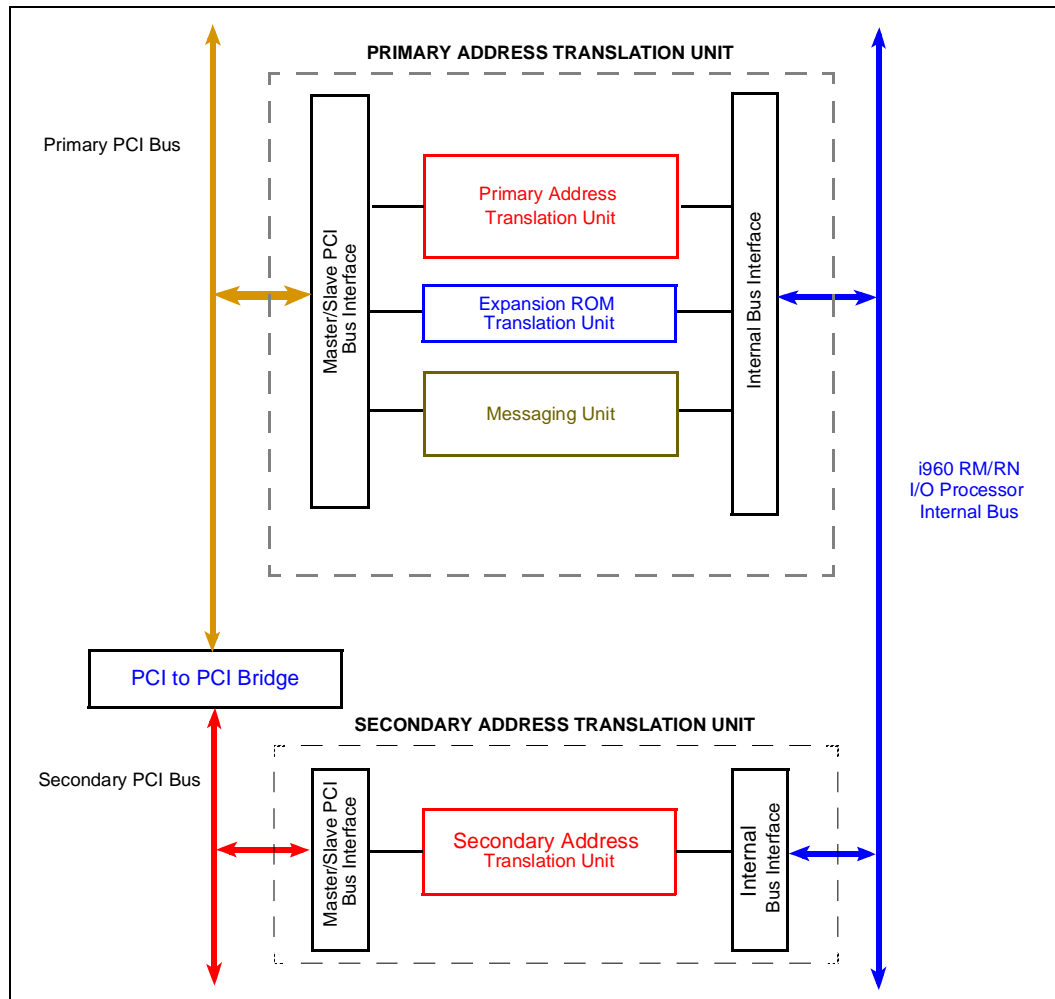
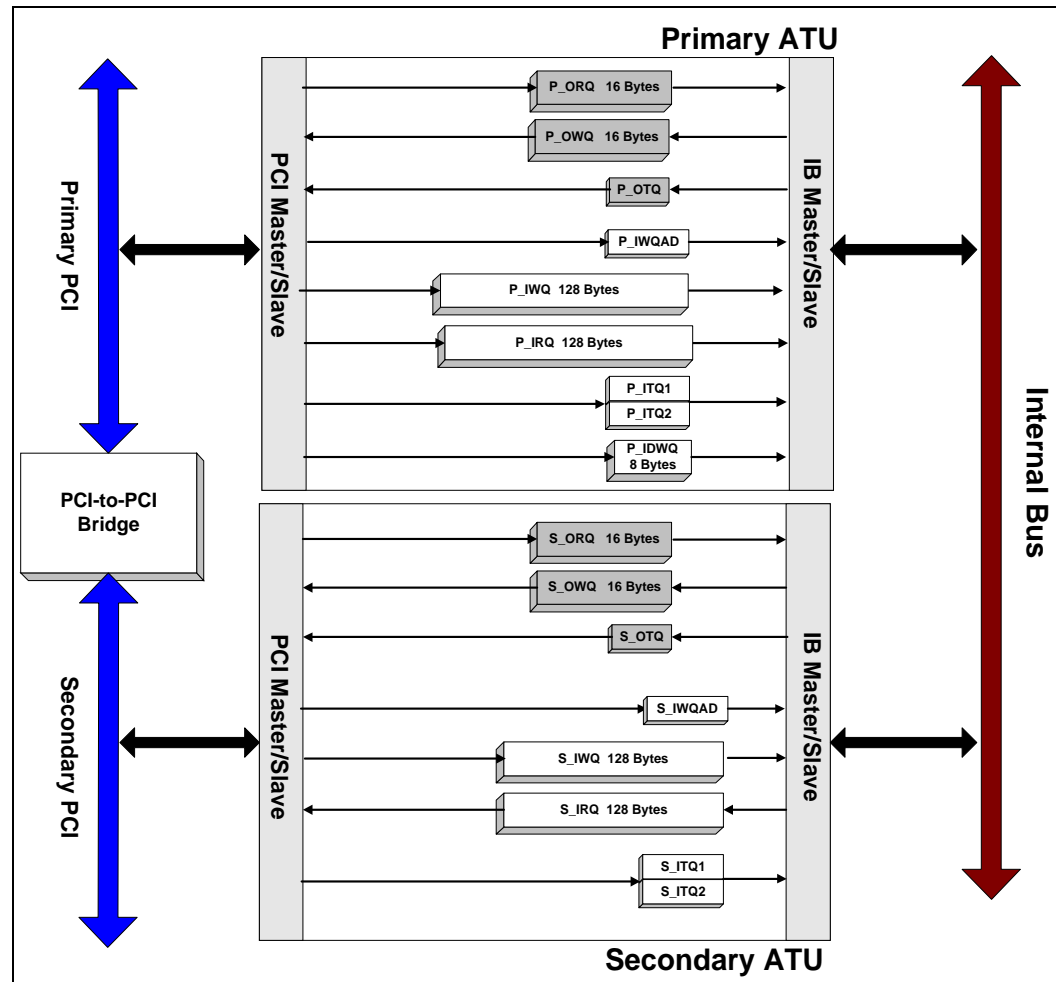


Figure 15-2. ATU Queue Architecture Block Diagram



## 15.2 ATU Address Translation

The primary ATU and the secondary ATU support transactions from both directions through the i960 RM/RN I/O processor. The primary ATU allows PCI masters on the primary PCI bus to initiate transactions to the i960 RM/RN I/O Processor Internal Bus and allows the 80960JT 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”](#).
- The address windowing mechanism for outbound translation is described in [Section 15.2.2.1, “Outbound Address Translation”](#).

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 i960 RM/RN 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 80960 local address and the fixed outbound windowing scheme. See [Section 15.2.2.1, “Outbound Address Translation”](#) for the full details on outbound PCI cycle selection.

Both ATUs support the 64-bit addressing specified by the *PCI Local Bus Specification* Revision 2.1. 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 i960 RM/RN 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 80960 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 i960 RM/RN 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 i960 RM/RN 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 2.1.

### 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 i960 RM/RN I/O processor memory-mapped registers or i960 RM/RN 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 i960 RM/RN 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 15-47](#) 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 PATU, in the event that an address can be claimed by both the ATU and the bridge, the PATU PCI interface has priority. 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, is claimed by the SATU.

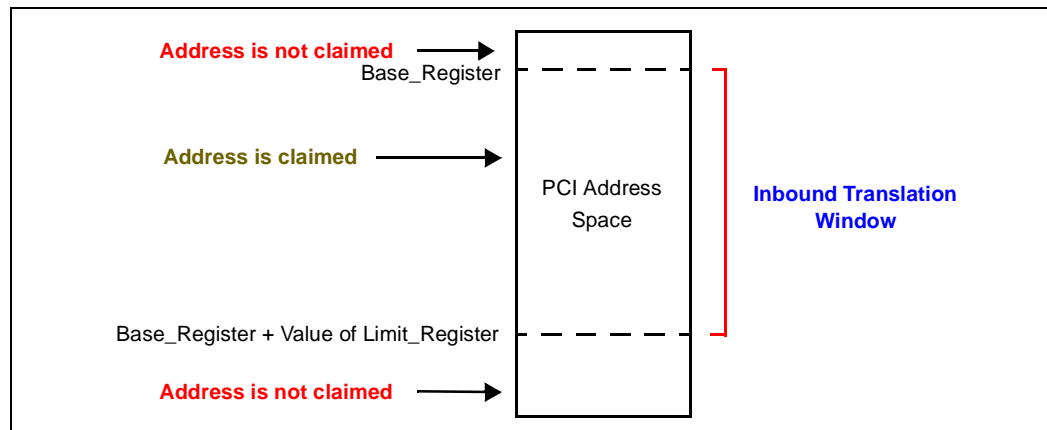
The first 4 Kbytes of the SATU inbound address space is dependent on the value of bit 12 (Secondary Bus-Messaging Unit Access Enable bit) of the ATUCR. If set, these addresses are claimed by the secondary interface of the Bridge Unit (if a valid bridge address). If clear, these addresses are claimed 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:

When  $(\text{PCI\_Address} \ \& \ \text{Limit\_Register} == \text{Base\_Register})$  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 ATU's 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 15-27.

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:

$$\text{i960}^{\text{®}} \text{ RM/RN I/O Processor Internal Bus Address} = (\text{PCI\_Address} \ \& \ \sim\text{Limit\_Register})$$
  

$$\text{ATU\_Translate\_Value\_Register}$$

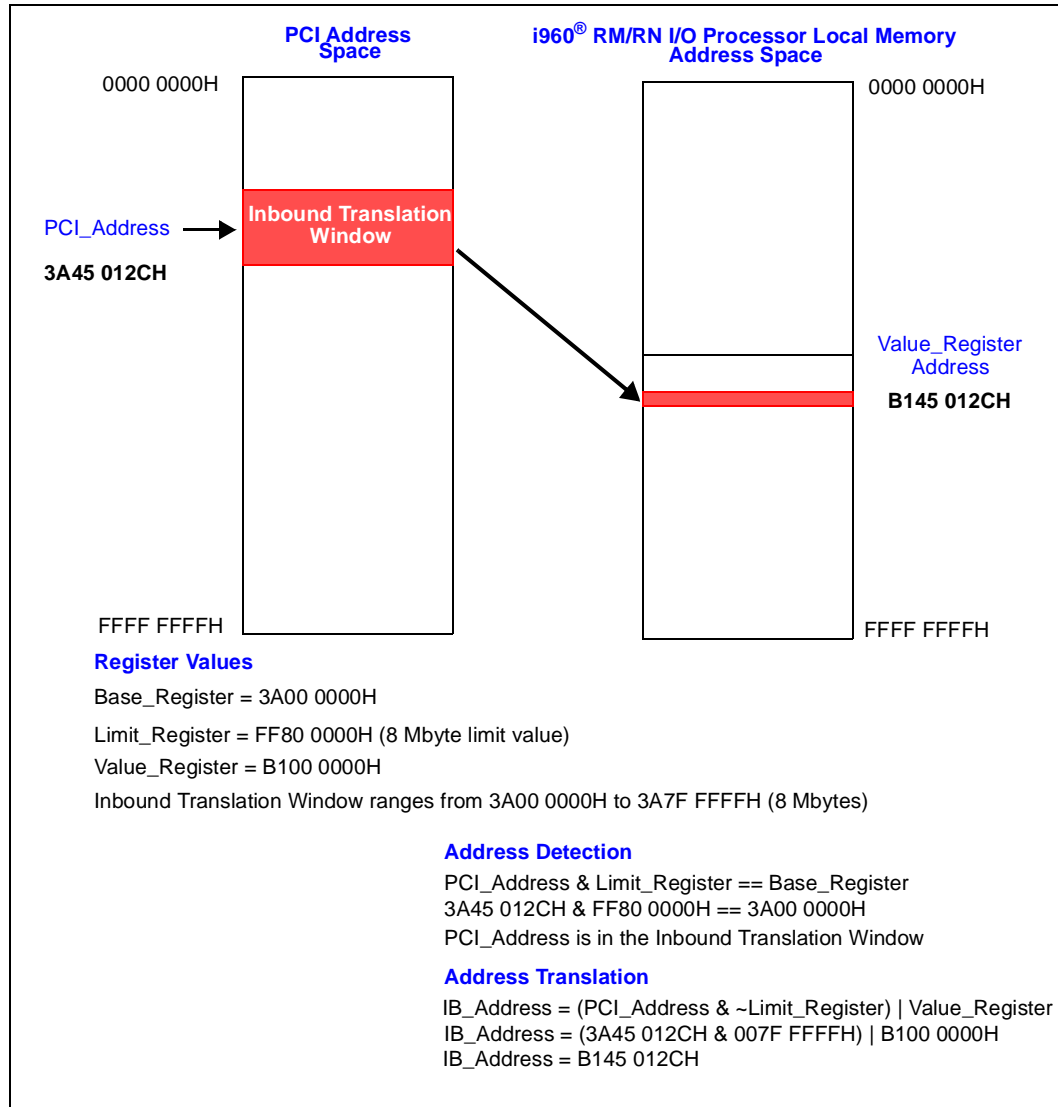
The incoming 32-bit PCI address is first bitwise ANDed with the bitwise inverse of the limit register. This result is bitwise ORed with the ATU Translate Value and the result is the internal bus address. This translation mechanism is used for all inbound memory read and write commands excluding inbound configuration read and writes. Inbound configuration cycle translation is described in [Section 15.2.1.4](#), on page 15-12. Address aliasing of multiple PCI addresses to the same physical 80960 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 i960 RM/RN I/O processor local memory or a i960 RM/RN 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, 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 performs a disconnect, and SERR# is asserted based upon the setting of the PATUIMR or SATUIMR, see [Section 15.7.44](#), “Primary ATU Interrupt Mask Register - PATUIMR” and [Section 15.7.45](#), “Secondary ATU Interrupt Mask Register - SATUIMR”.

Once the PCI interface places a PCI address in the IWQAD and at least 1 64-byte boundary is crossed or if the master disconnects on the PCI bus, the ATU’s 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 attempts 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 does 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 a PCI address and at least 2 64-byte boundaries are crossed by the current transaction on the PCI bus appears in the IWQAD/IWQ 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 15-3](#). 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 15.7.33, “ATU Configuration Register - ATUCR”](#).
- 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 asserts 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. Refer to the Internal Bus Chapter for details of internal bus operation.
- 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 performs 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 15.7.44, “Primary ATU Interrupt Mask Register - PATUIMR”](#) and [Section 15.7.45, “Secondary ATU Interrupt Mask Register - SATUIMR”](#). 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 15.7.44, “Primary ATU Interrupt Mask Register - PATUIMR”](#) and [Section 15.7.45, “Secondary ATU Interrupt Mask Register - SATUIMR”](#). 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 attempts 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 i960 RM/RN I/O processor local memory or a i960 RM/RN 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 ATU's 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 15.7.33, "ATU Configuration Register - ATUCR" on page 15-80](#)). 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, the 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 the PCI transaction. If there is data left unread in the IRQ, the data 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 waits 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. Refer to [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 requests 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 drives the translated address onto the bus and wait for `I_DEVSEL#`. If a Retry is signaled, the request is 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 starts 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 continues 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 reacquires the internal bus.

To support *PCI Local Bus Specification* Revision 2.1 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 does 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 i960 RM/RN I/O processor). (Refer to [Section 15.2.4, “PCI Multi-Function Device Swapping/Disabling”](#) 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 claims 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<sub>2</sub> all during the address phase. The ATU primary interface ignores any configuration command (P\_IDSEL active) where P\_AD[1:0] are not 00<sub>2</sub> (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 only responds to 001<sub>2</sub> in this bit field and ignore all other bit combinations. (Refer to [Section 15.2.4, “PCI Multi-Function Device Swapping/Disabling”](#) for exceptions to this statement.)

The ATU configuration address space starts at internal address 0000.1200H. Therefore P\_AD[7:2] equal to 000000<sub>2</sub> equates to address 0000.1200H and P\_AD[7:2] equal to 000001<sub>2</sub> 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 is 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 requests 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. 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 i960 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 15.2.5](#) 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 2.1.

### 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 i960 RM/RN 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 i960 RM/RN 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

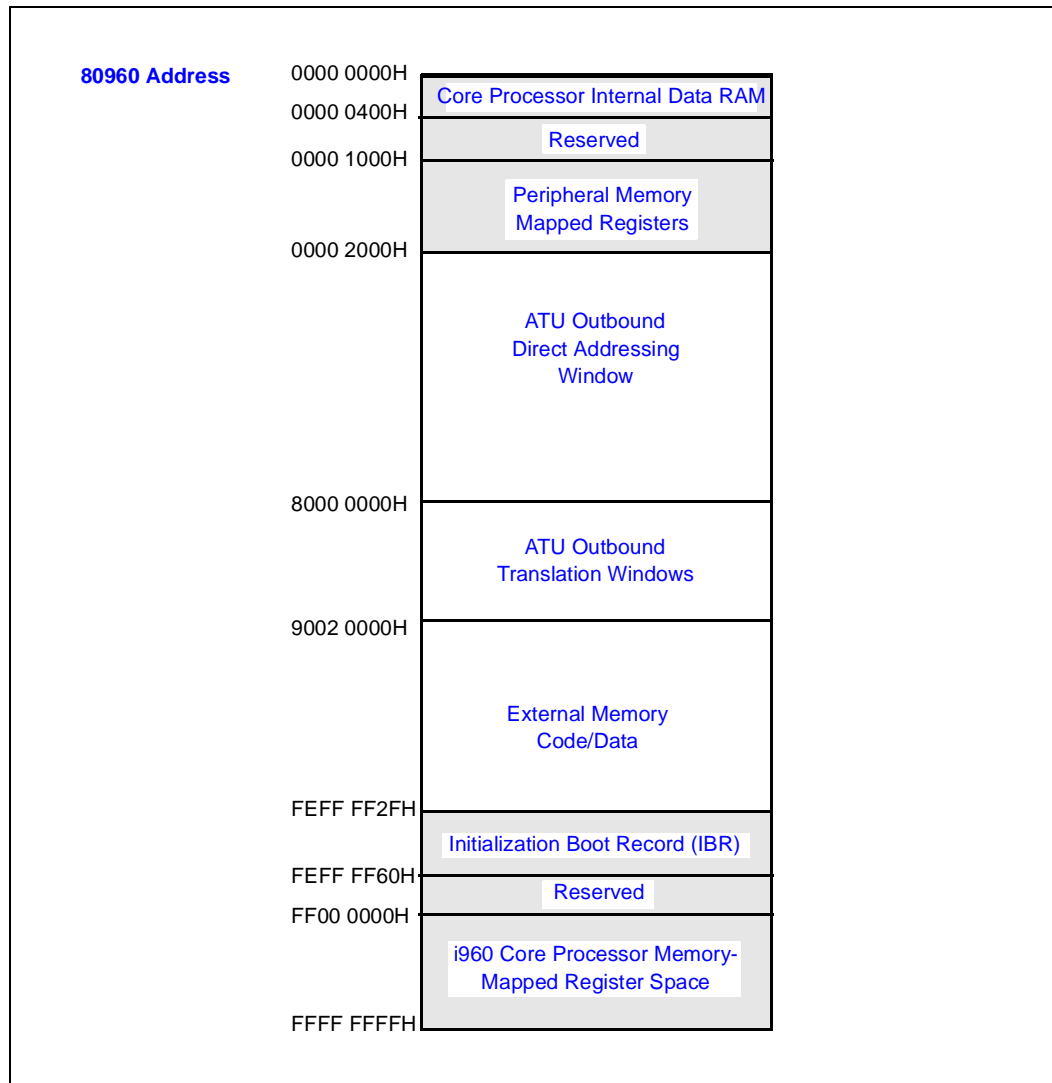
Refer to [Figure 15-6](#) for the sub-window addresses involved in primary and secondary outbound translation.

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. 80960 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 the SOUDR contain the high order 32-bits of a dual cycle 64-bit address. Each ATU uses the following registers 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, “Register Definitions”](#) on page 15-47 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:

$$\text{PCI Address} = (\text{Internal\_Bus\_Address} \& \text{03FF.FFFFH}) | \text{Window\_Value\_Register}$$

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.

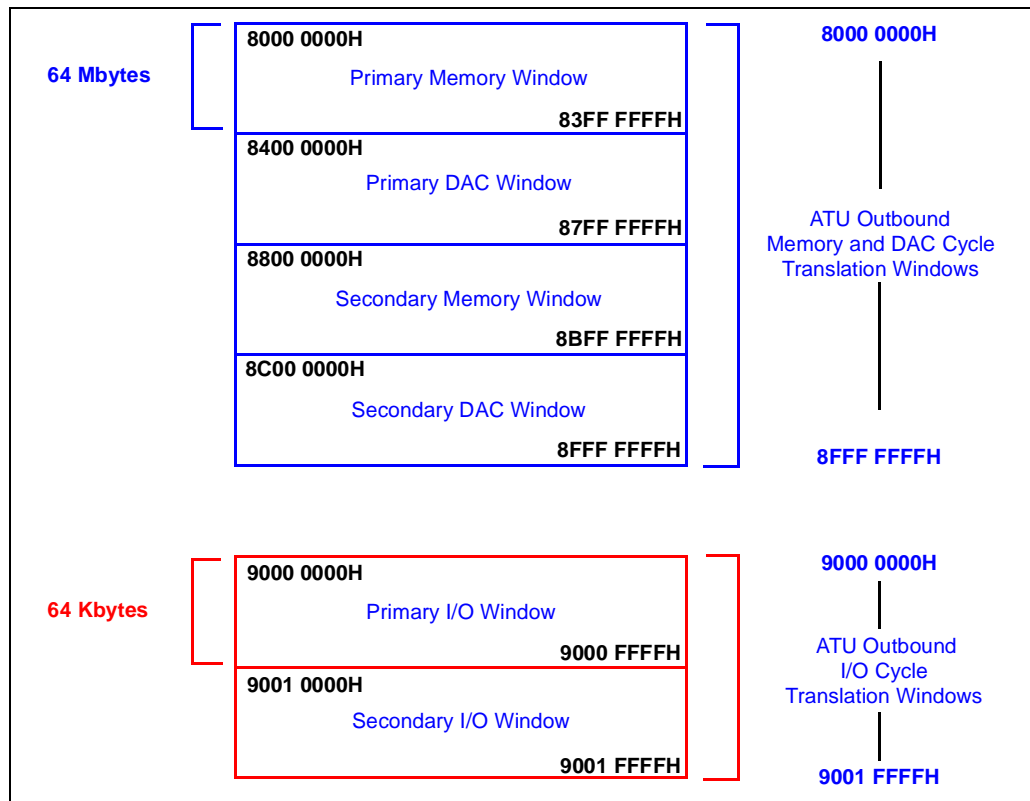
For I/O transactions, the algorithm is:

$$\text{PCI Address} = (\text{Internal\_Bus\_Address} \& \text{0000.FFFFH}) | \text{Window\_Value\_Register}$$

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's 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**

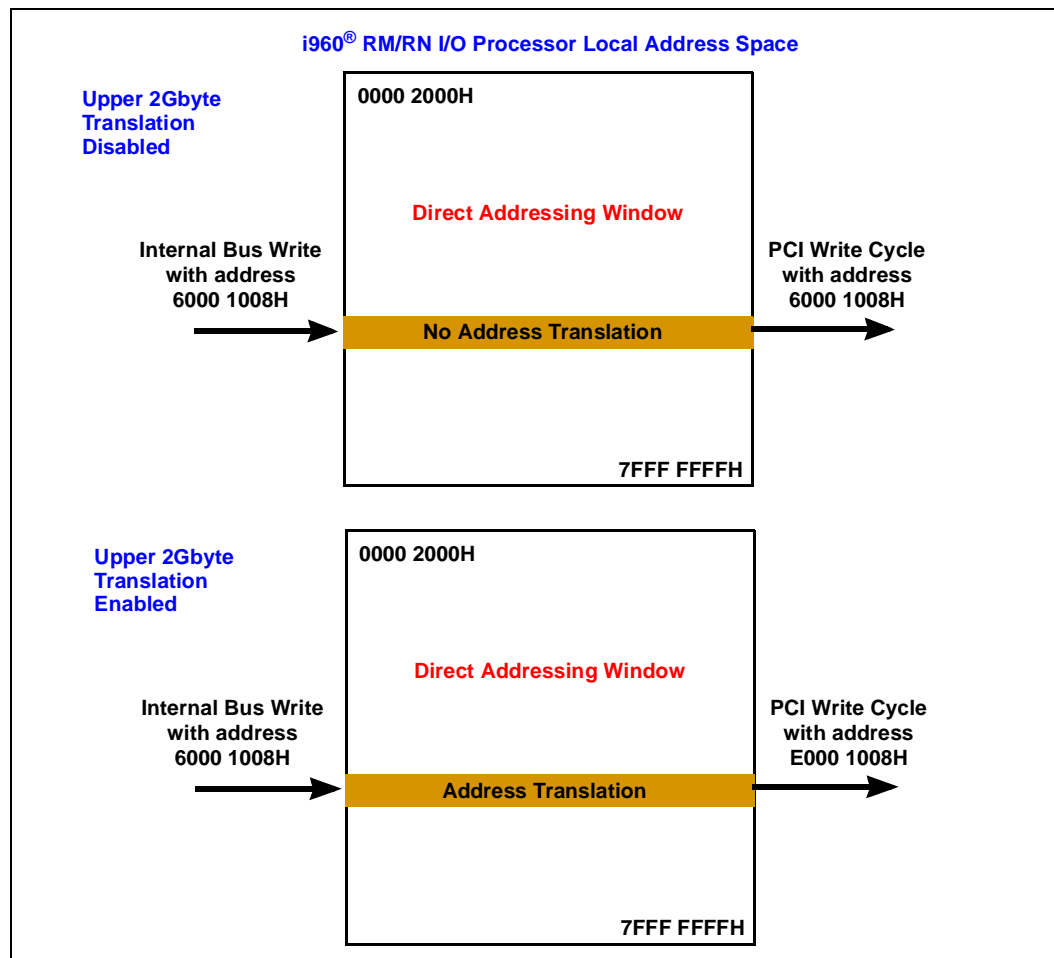


### 15.2.2.3 Direct Addressing Window

The second method used by outbound cycles from the i960 RM/RN I/O processor to the PCI bus is the direct addressing window. This is a window of addresses in i960 RM/RN 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 i960 RM/RN 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 i960 RM/RN I/O processor local address space (except for the first 8 Kbytes which is reserved for the i960 core processor's internal data RAM and i960 RM/RN 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 forwards 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 occurs.

#### 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 i960 RM/RN I/O Processor Internal Bus to a PCI bus through the OTQ and OWQ, respectively.

The ATU's slave internal bus 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 latches the address from the internal bus into the OTQ when that address is inside one of the outbound translate windows ([Section 15.5](#)) and the OTQ is empty.
- Once the outbound address is latched, the internal bus slave interface stores the write data into the OWQ until the internal bus transaction completes. The initiator of the transaction performs 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 signals a Retry on the internal bus to the outbound cycle initiator.
- 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 requests the PCI bus when an address is written to the OTQ (a write request). Once the bus is granted, the PCI master interface writes 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 transfers 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 attempts 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 attempts 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 continues 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 ATU's 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 prefetch 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 prefetch data sizes used during outbound ATU read transactions:

**Table 15-2. Outbound Read Prefetch Sizes**

Internal Bus Command	Outbound Prefetch Size
Memory Read	4 Bytes (1 Dword)
Memory Read Line	8 Bytes (2 Dwords)
Memory Read Multiple	16 (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 signals 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 (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 requests the PCI bus when an address is written to the OTQ (a read request). Once the bus is granted, the PCI interface transfers 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 prefetch data amount is satisfied. The master interface stops reading under the following circumstances:
  - A disconnect is signaled from the PCI target. The master interface attempts to reacquire the bus and continue reading until the prefetch 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 continue reading until the prefetch 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 prefetch 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 i960 RM/RN 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. (Chapter 14, “PCI-to-PCI Bridge” 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.

Outbound transactions from the secondary ATU and DMAs are not claimed by the bridge unit and therefore can access private devices on the secondary 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 2.1 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 issued a disconnect with data for 32-bits only. This cycle does 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 15-47](#) 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 i960 RM/RN 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 i960 RM/RN 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 i960 RM/RN 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 [Table 15-62 “ATU Configuration Register - ATUCR”](#) on [page 15-80](#) and setting the value in the [Table 15-37 “ATU Header Type Register - ATUHTR”](#) on [page 15-57](#) and [Table 14-32 “Header Type Register- HTR”](#) on [page 14-80](#) from the i960 core processor. The i960 RM/RN 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	i960® RM/RN 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 i960 RM/RN 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 264 MBytes/sec. The 64-bit PCI extensions add 39 additional signals to each ATU 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 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) independently samples 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”](#) for details.



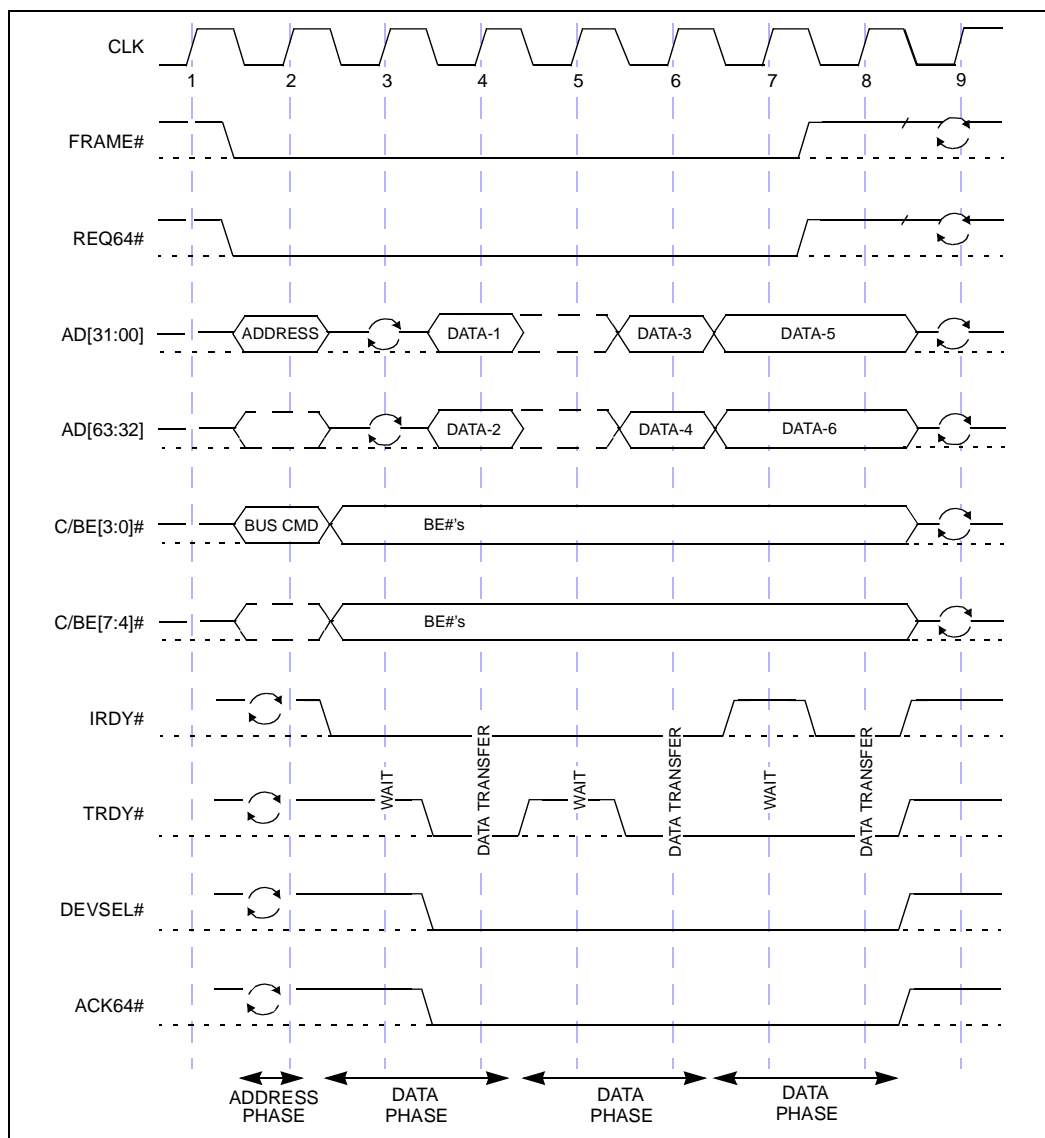
### 15.2.5.1 64-Bit Protocol

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 i960 RM/RN 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 asserts 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 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 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 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 asserts and deasserts 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 becomes 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# remains 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 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#:

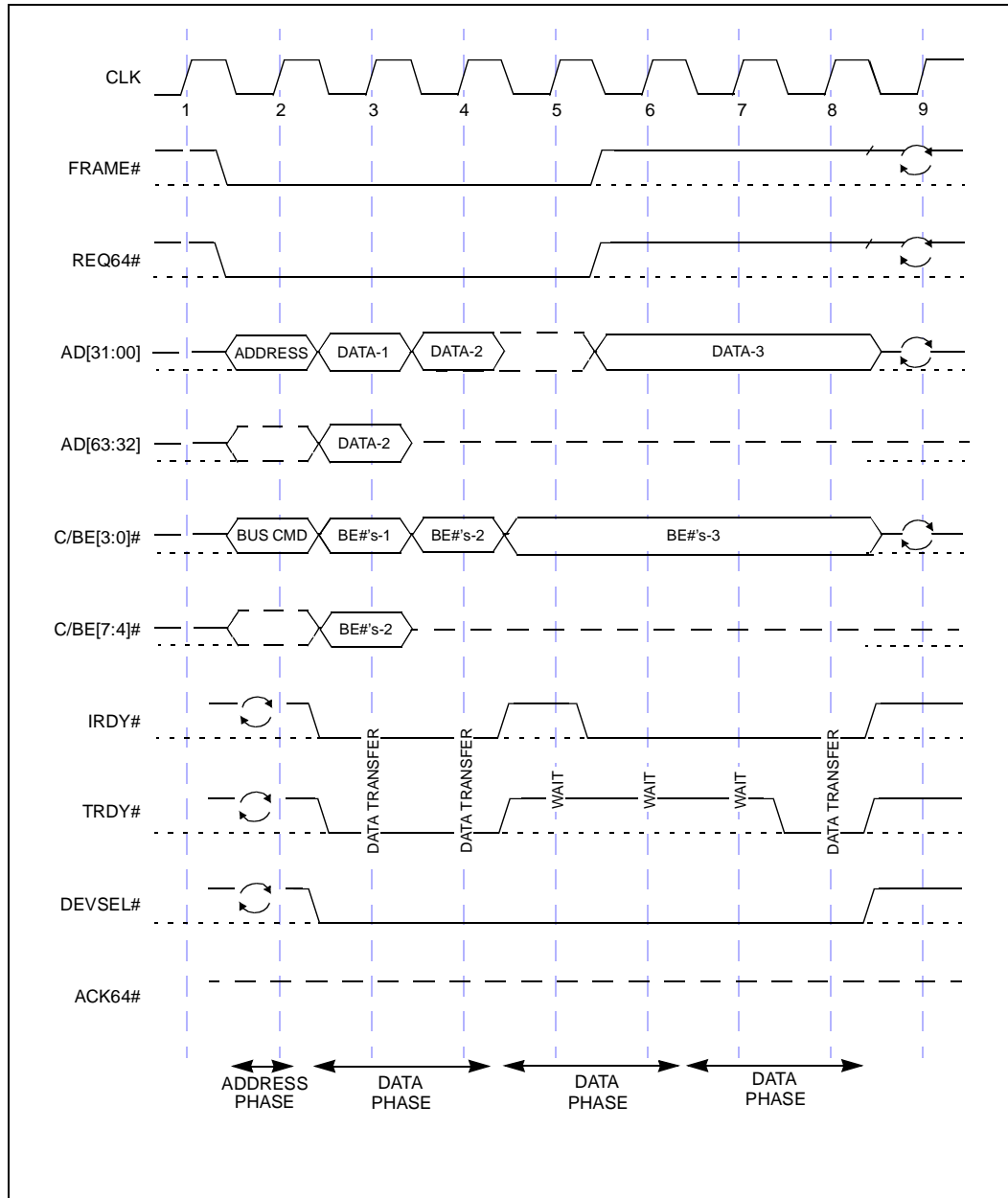
1. ACK64# deasserted but a burst can be sustained
2. ACK64# deasserted but a burst can not be sustained

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 results 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 does 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 master 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



## 15.3 Messaging Unit

The Messaging Unit (MU) transfers data between the PCI system and the i960 RM/RN 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 i960 RM/RN 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 15.7.33, “ATU Configuration Register - ATUCR” on page 15-80](#) 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”](#).

## 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 2.1 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 in [Section 15.7.14](#), [Section 15.7.31](#), and [Section 15.7.32](#).

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”](#). The only width Expansion ROM supported by the i960 RM/RN 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 fills 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 queuing 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 i960 RM/RN I/O processor local memory or a i960 RM/RN 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	128
	P_IWQAD	Primary Inbound Write Address Queue	4 Transaction Addresses
	P_IRQ	Primary Inbound Read Data Queue	128
	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	128
	S_IWQAD	Secondary Inbound Write Address Queue	4 Transaction Addresses
	S_IRQ	Secondary Inbound Read Data Queue	128
	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 128 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 128 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. 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 ([Section 15.5.3, “Transaction Ordering”](#)). 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 buses 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 waitstates 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 DRR 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 starts 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 only begins 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 does 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	64
	Memory Read Multiple	128
SATU	Memory Read	32
	Memory Read Line	64
	Memory Read Multiple	128

### 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 the SATU and result in a Master Abort.

The IDWQ contains both the 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 used is the specific configuration translation defined in [Section 15.2.1.4](#). The transaction is forwarded to the IB bus once transaction ordering has been satisfied and the translated write cycle is performed on the internal bus with the 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 queues of the ATU are used to hold read and write transactions from the core processor directed at the PCI buses. Each ATU outbound queue structure has a separate read queue, write queue, 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, memory 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 ([Section 15.7.28](#) and [Section 15.7.39](#)).

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 read command presented on the internal bus 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 the 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 i960 RM/RN 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 <sup>1</sup> Inbound Writes	No	No	No/Yes <sup>2</sup>	No	Yes
	MU <sup>3</sup> Inbound Writes	No	No	No	No	Yes
Inbound Read Request		No	No	No	No	Yes
Inbound Configuration Write Request <sup>4</sup>		No	No	No	No	Yes
Outbound Read Completion		No <sup>5</sup>	No <sup>5</sup>	Yes	No <sup>5</sup>	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.
5. The only situation where an outbound read completion can pass an inbound write is if the outbound read master-aborts or target-aborts on the PCI bus.

**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 <sup>1</sup>	Yes	Yes	Yes	No

1. Not valid for Secondary ATU.

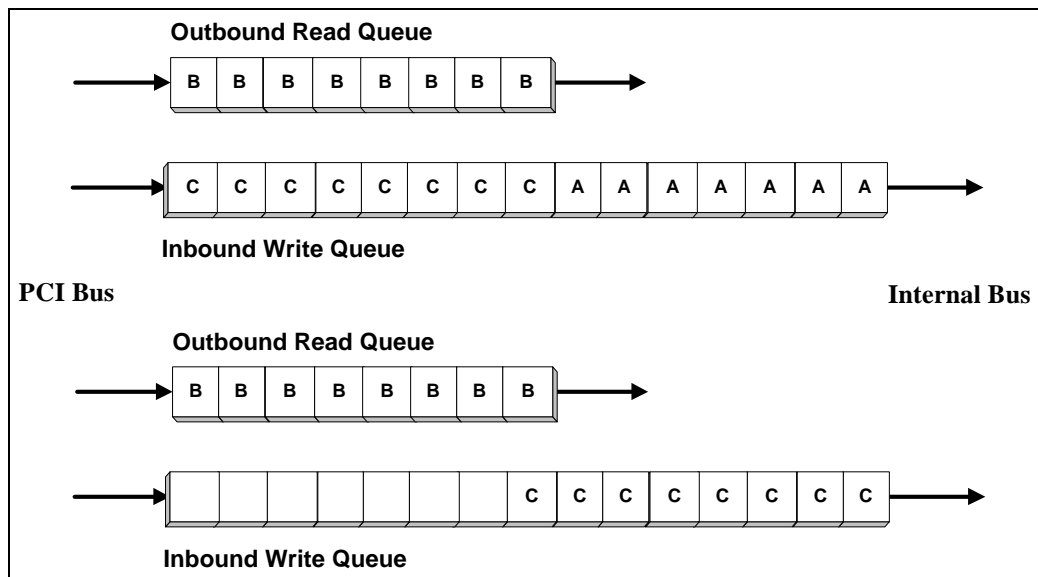
Definitions of the terms used in [Table 15-7](#) and [Table 15-8](#) are indicated as follows. PCI terms are noted in parenthesis:

- 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 a read transactions retried and delayed on the PCI bus. Is mastered on the internal bus to retrieve data for the Inbound Read Completion.
- Inbound Configuration Write Request - (DWR) - The address and data associated with a configuration write transaction from primary PCI and targeted at the ATU PCI configuration address space. Once completed on the internal bus, creates an Inbound Configuration Write Completion. Only available in the PATU.
- Outbound Read Completion (DRC) - The data read on PCI in the process of being returned to the BIU on the internal bus. This data is the completion cycle that results from an Outbound Read Request.
- Outbound Write (PMW) - The address and data from a write initiated on the internal bus and eventually completing on the PCI bus.
- Outbound Read Request (DRR) - The 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) - The 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) - The status of an inbound write configuration cycle traveling from the internal bus back towards the primary PCI bus. This is only present in the PATU.

These transaction ordering rules define the way in which 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 (the row) can not pass the previous transaction (the column) under any circumstance. A **Yes** response in the box means that 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.

**Figure 15-10. Inbound Queue Completion**



In [Figure 15-10](#), 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 completes (defined in the next section). In this case, if the BIU acquires the internal bus first, Transaction B completes. If the ATU acquires the internal bus first, Transaction C completes. 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 i960 RM/RN 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 2.1. 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 ([Section 13.5, “Interrupts/Error Conditions” on page 13-39](#) 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 15-47](#).

The following sections detail all ATU error conditions on the PCI bus and the i960 RM/RN 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 i960 RM/RN I/O processor performs the following actions based on the constraints specified:

**Table 15-9. Address Parity Errors on PCI Interface (Sheet 1 of 2)**

Primary ATU	Secondary ATU
If the Parity Error Response bit in the PATUCMD is set, the PATU does <i>not</i> claim the transaction by <i>not</i> 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 does <i>not</i> claim the transaction by <i>not</i> 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.

**Table 15-9. Address Parity Errors on PCI Interface (Sheet 2 of 2)**

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 detects 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 detects 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, results in the bad data being delivered back to the initiator on the internal bus of the i960 RM/RN 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 continues 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, does 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 continues 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 delays 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 delivers the status of the operation back to the original master.

The i960 RM/RN 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 is 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 retries the transaction on the PCI bus and the PATU returns 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 retries the transaction with no other response. A new transaction is not enqueued due to queue architecture constraints (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 (Chapter 12, “Core Processor and Internal Operation”).

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 purposely master aborts 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 results 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 Abort Signaled 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 Abort Detected on the PCI Interface**

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 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 15.7.33, “ATU Configuration Register - ATUCR”](#) on [page 15-80](#) 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# Asserted by PCI Interface**

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# Detected by PCI Interface**

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 that 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 i960 RM/RN 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 results 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 returns 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 Detected by Internal Master Interface During Inbound Write**

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 Detected by Internal Master Interface During Inbound Read**

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 always results 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 and 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 do 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 Detected by Internal Master Interface During Inbound Write**

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 NMI# interrupt to the core processor. The inbound write queue (IWQ) is not cleared and the ATU internal bus master interface re-arbitrates 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 Detected by Internal Master Interface During Inbound Read**

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 is considered to have target aborted.

## 15.6.7 ATU Error Summary

The following four tables summarize the ATU error reporting for PCI bus errors and internal bus errors. 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 80960JT 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 (Sheet 1 of 2)**

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 07
	P_SERR# Asserted - bit 14	P_SERR# Asserted - bit 10	PATUIMR bit 06
	N/A	P_SERR# Detected - bit 4	ATUCR bit 09
Inbound Write Data Parity Error	Detected Parity Error - bit 15	Detected Parity Error - bit 9	PATUIMR bit 07
	P_SERR# Asserted - bit 14	P_SERR# Asserted - bit 10	PATUIMR bit 06
Inbound Write Master or Target Abort	N/A	P_SERR# Detected - bit 4	ATUCR bit 09
	Detected Parity Error - bit 15	Detected Parity Error - bit 9	PATUIMR bit 07
Inbound Read Address Parity Error	P_SERR# Asserted - bit 14	P_SERR# Asserted - bit 10	PATUIMR bit 06
	N/A	P_SERR# Detected - bit 4	ATUCR bit 09
	Detected Parity Error - bit 15	Detected Parity Error - bit 9	PATUIMR bit 07
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 03
Outbound Write Master Abort	Master Abort - bit 13	PCI Master Abort - bit 3	PATUIMR bit 05

**Table 15-22. Primary ATU Error Reporting Summary - PCI Interface (Sheet 2 of 2)**

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
Outbound Write Data Parity Error	Master Parity Error - bit 8	PCI Master Parity Error - bit 0	PATUIMR bit 02
Outbound Write Target Abort	Target Abort (master) - bit 12	PCI Target Abort (master) - bit 2	PATUIMR bit 04
Outbound Read Master Abort	Master Abort - bit 13	PCI Master Abort - bit 3	PATUIMR bit 05
Outbound Read Data Parity Error	Detected Parity Error - bit 15	Detected Parity Error - bit 9	PATUIMR bit 07
	Master Parity Error - bit 8	PCI Master Parity Error - bit 0	PATUIMR bit 02
Outbound Read Target Abort	Target Abort (master) - bit 12	PCI Target Abort (master) - bit 2	PATUIMR bit 04
P_SERR# Detected	N/A	P_SERR# Detected - bit 4	ATUCR bit 09

**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 07
	S_SERR# Asserted - bit 14	S_SERR# Asserted - bit 10	SATUIMR bit 06
	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 07
Inbound Write Master or Target Abort	S_SERR# Asserted - bit 14	S_SERR# Asserted - bit 10	SATUIMR bit 06
	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 07
	S_SERR# Asserted - bit 14	S_SERR# Asserted - bit 10	SATUIMR bit 06
	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 03
Outbound Write Master Abort	Master Abort - bit 13	PCI Master Abort - bit 3	SATUIMR bit 05
Outbound Write Data Parity Error	Master Parity Error - bit 8	PCI Master Parity Error - bit 0	SATUIMR bit 02
Outbound Write Target Abort	Target Abort (master) - bit 12	PCI Target Abort (master) - bit 2	SATUIMR bit 04
Outbound Read Master Abort	Master Abort - bit 13	PCI Master Abort - bit 3	SATUIMR bit 05
Outbound Read Data Parity Error	Detected Parity Error - bit 15	Detected Parity Error - bit 9	SATUIMR bit 07
	Master Parity Error - bit 8	PCI Master Parity Error - bit 0	SATUIMR bit 02
Outbound Read Target Abort	Target Abort (master) - bit 12	PCI Target Abort (master) - bit 2	SATUIMR bit 04
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 06
	N/A	P_SERR# Detected - bit 4	ATUCR bit 09
Inbound Write Target Abort	P_SERR# Asserted - bit 14	P_SERR# Asserted - bit 10	PATUIMR bit 06
	N/A	P_SERR# Detected - bit 4	ATUCR bit 09
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 <sup>1</sup>	N/A	N/A	N/A
Outbound Write Target Abort <sup>2</sup>	N/A	N/A	N/A
Outbound Read Master Abort <sup>3</sup>	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 06
	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 06
	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 <sup>1</sup>	N/A	N/A	N/A
Outbound Write Target Abort <sup>2</sup>	N/A	N/A	N/A
Outbound Read Master Abort <sup>3</sup>	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 2.1 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 is accepted on the internal bus as 32-bit transactions. Refer to Appendix C, “Memory-Mapped Registers”.

**Figure 15-11. ATU Interface Configuration Header Format**

ATU Device ID		Vendor ID		00H
Primary Status		Primary Command		04H
ATU Class Code			Revision ID	08H
BIST	Header Type	Latency Timer	Cacheline Size	0CH
Primary Inbound ATU Base Address				10H
Reserved				14H
				18H
				1CH
				20H
				24H
				28H
ATU Subsystem ID		ATU Subsystem Vendor ID		2CH
Expansion ROM Base Address				30H
				34H
				38H
				3CH
Maximum Latency	Minimum Grant	Interrupt Pin	Interrupt Line	

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 15-12. ATU configuration space is function number one of the i960 RM/RN I/O processor multi-function PCI device. (Refer to Section 15.2.4, “PCI Multi-Function Device Swapping/Disabling” on page 15-22 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 2.1 for details on accessing and programming configuration register space.

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 1.4, “About This Document”](#) on page 1-9 for a description of *reserved*, *read only*, and *read/clear*. All registers adhere to the definitions found in the *PCI Local Bus Specification* Revision 2.1 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 i960 core processor. See also [Appendix C, “Memory-Mapped Registers”](#).

**Table 15-26. Address Translation Unit Registers (Sheet 1 of 2)**

Register Name	Register Size in Bits	PCI Configuration Cycle Register Number	Internal Bus Address
ATU Vendor ID Register - ATUVID	16	0	0000.1200H
Device ID Register - DID (80960RN) Device ID Register - DID (80960RM)	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 - 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
Reserved	32	13	0000.1234H
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

**Table 15-26. Address Translation Unit Registers (Sheet 2 of 2)**

Register Name	Register Size in Bits	PCI Configuration Cycle Register Number	Internal Bus Address
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
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
Reserved	32	32	0000.1280H
Reserved	32	33	0000.1284H
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
Secondary Outbound Configuration Cycle Data Register - SOCCDR	32	Not Available in PCI Configuration Space	0000.12ACH
Primary Outbound Configuration Cycle Data Register - POCCDR	32	Not Available in PCI Configuration Space	0000.12B0H
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

Internal Bus Address Offset	ATU PCI Configuration Register Section, Name, Page
00H	Section 15.7.1, "ATU Vendor ID Register - ATUVID" on page 15-51
02H	Section 15.7.2, "ATU Device ID Register - ATUDID" on page 15-52
04H	Section 15.7.3, "Primary ATU Command Register - PATUCMD" on page 15-53
06H	Section 15.7.4, "Primary ATU Status Register - PATUSR" on page 15-54
08H	Section 15.7.5, "ATU Revision ID Register - ATURID" on page 15-55
09H	Section 15.7.6, "ATU Class Code Register - ATUCCR" on page 15-56
0CH	Section 15.7.7, "ATU Cacheline Size Register - ATUCLSR" on page 15-56
0DH	Section 15.7.8, "ATU Latency Timer Register - ATULT" on page 15-57
0EH	Section 15.7.9, "ATU Header Type Register - ATUHTR" on page 15-57
0FH	Section 15.7.10, "ATU BIST Register - ATUBISTR" on page 15-58
10H	Section 15.7.11, "Primary Inbound ATU Base Address Register - PIABAR" on page 15-59
2CH	Section 15.7.12, "ATU Subsystem Vendor ID Register - ASVIR" on page 15-60
2EH	Section 15.7.13, "ATU Subsystem ID Register - ASIR" on page 15-60
30H	Section 15.7.14, "Expansion ROM Base Address Register - ERBAR" on page 15-61
3CH	Section 15.7.15, "Determining Block Sizes for Base Address Registers" on page 15-62
3DH	Section 15.7.16, "ATU Interrupt Line Register - ATUILR" on page 15-63
3EH	Section 15.7.18, "ATU Minimum Grant Register - ATUMGNT" on page 15-65
3FH	Section 15.7.19, "ATU Maximum Latency Register - ATUMLAT" on page 15-66
40H	Section 15.7.20, "Primary Inbound ATU Limit Register - PIALR" on page 15-67
44H	Section 15.7.21, "Primary Inbound ATU Translate Value Register - PIATVR" on page 15-68
48H	Section 15.7.22, "Secondary Inbound ATU Base Address Register - SIABAR" on page 15-69
4CH	Section 15.7.23, "Secondary Inbound ATU Limit Register - SIALR" on page 15-70
50H	Section 15.7.24, "Secondary Inbound ATU Translate Value Register - SIATVR" on page 15-71
54H	Section 15.7.25, "Primary Outbound Memory Window Value Register - POMWVR" on page 15-72
5CH	Section 15.7.26, "Primary Outbound I/O Window Value Register - POIOWVR" on page 15-73
60H	Section 15.7.27, "Primary Outbound DAC Window Value Register - PODWVR" on page 15-74
64H	Section 15.7.28, "Primary Outbound Upper 64-bit DAC Register - POUADR" on page 15-75
68H	Section 15.7.29, "Secondary Outbound Memory Window Value Register - SOMWVR" on page 15-76
6CH	Section 15.7.30, "Secondary Outbound I/O Window Value Register - SOIOWVR" on page 15-77
74H	Section 15.7.31, "Expansion ROM Limit Register - ERLR" on page 15-78
78H	Section 15.7.32, "Expansion ROM Translate Value Register - ERTVR" on page 15-79
88H	Section 15.7.33, "ATU Configuration Register - ATUCR" on page 15-80
90H	Section 15.7.34, "Primary ATU Interrupt Status Register - PATUISR" on page 15-82
94H	Section 15.7.35, "Secondary ATU Interrupt Status Register - SATUISR" on page 15-84
98H	Section 15.7.36, "Secondary ATU Command Register - SATUCMD" on page 15-86
9AH	Section 15.7.37, "Secondary ATU Status Register - SATUSR" on page 15-87
9CH	Section 15.7.38, "Secondary Outbound DAC Window Value Register - SODWVR" on page 15-88
A0H	Section 15.7.39, "Secondary Outbound Upper 64-bit DAC Register - SOADR" on page 15-89
A4H	Section 15.7.40, "Primary Outbound Configuration Cycle Address Register - POCCAR" on page 15-90
A8H	Section 15.7.41, "Secondary Outbound Configuration Cycle Address Register - SOCCAR" on page 15-91
ACH	Section 15.7.42, "Primary Outbound Configuration Cycle Data Register - POCCDR" on page 15-92
B0H	Section 15.7.43, "Secondary Outbound Configuration Cycle Data Register - SOCCDR" on page 15-93
BCH	Section 15.7.44, "Primary ATU Interrupt Mask Register - PATUIMR" on page 15-94
C0H	Section 15.7.45, "Secondary ATU Interrupt Mask Register - SATUIMR" on page 15-95

### 15.7.1 ATU Vendor ID Register - ATUVID

ATU Vendor ID Register bits adhere to the definitions in the *PCI Local Bus Specification* Revision 2.1.

**Table 15-28. ATU Vendor ID Register - ATUVID**

		<p>Internal Bus Address 1200H</p> <p>PCI Configuration Address Offset 00H - 01H</p> <p>Attribute Legend:          RW = Read/Write          RV = Reserved          PR = Preserved          RS = Read/Set          RC = Read Clear          RO = Read Only          NA = Not Accessible</p>
Bit	Default	Description
15:00	8086H	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 i960 RM/RN 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 2.1.

**Table 15-29. Device ID Register - DID (80960RN)**

Internal 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	1964H	ATU Device ID - This is a 16-bit value assigned to the ATU and MU. This ID, combined with the VID, uniquely identify any PCI device.	

**Table 15-30. Device ID Register - DID (80960RM)**

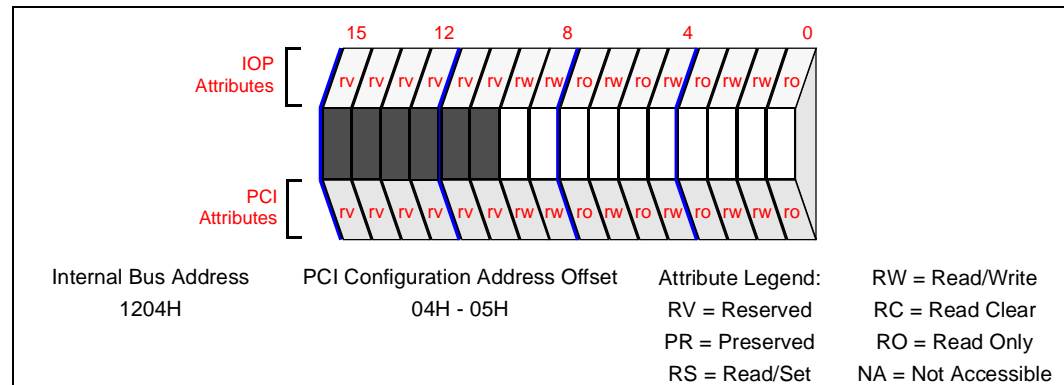
Internal Bus Address 1202H	PCI Configuration Address Offset 02H - 03H	Attribute Legend: RV = Reserved PR = Preserved RS = Read/Set	RW = Read/Write RC = Read Clear RO = Read Only NA = Not Accessible
Bit	Default	Description	
15:00	1962H	ATU Device ID - This is a 16-bit value assigned to the ATU and MU. 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 2.1 and in most cases, affect the behavior of the primary PCI ATU and devices on the primary PCI bus.

**Table 15-31. Primary ATU Command Register - PATUCMD**

Bit	Default	Description
15:10	000000 <sub>2</sub>	Reserved
09	0 <sub>2</sub>	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 <sub>2</sub>	P_SERR# Enable - When cleared, the ATU primary interface is not allowed to assert P_SERR# on the primary PCI interface.
07	0 <sub>2</sub>	Wait Cycle Control - controls address/data stepping. Not implemented and a reserved bit field.
06	0 <sub>2</sub>	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 <sub>2</sub>	VGA Palette Snoop Enable - The primary ATU interface does not support I/O writes and therefore, does not perform VGA palette snooping.
04	0 <sub>2</sub>	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 <sub>2</sub>	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 <sub>2</sub>	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 <sub>2</sub>	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 <sub>2</sub>	I/O Space Enable - Controls the ATU interface response to I/O transactions on the primary side. Not implemented and a reserved bit field.

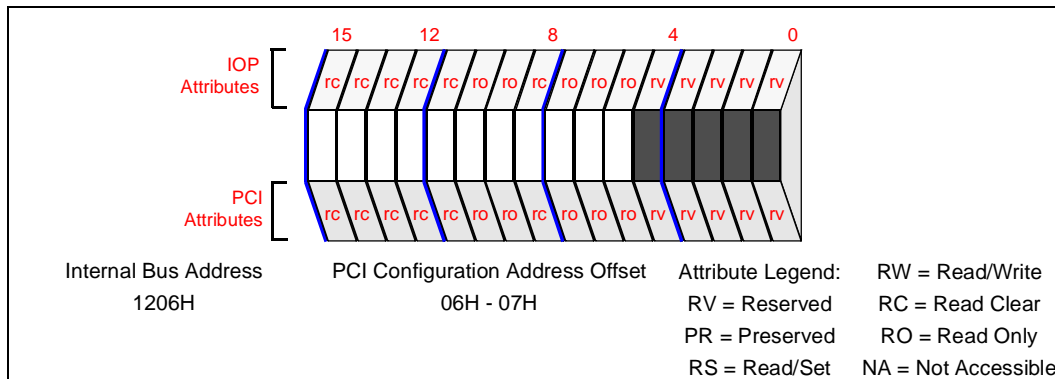


## 15.7.4 Primary ATU Status Register - PATUSR

The Primary ATU Status Register bits adhere to the *PCI Local Bus Specification* Revision 2.1 definitions. The *read/clear* bits can only be set by internal hardware and cleared by either a reset condition or by writing a 1<sub>2</sub> to the register.

**Table 15-32. Primary ATU Status Register - PATUSR (Sheet 1 of 2)**

Bit	Default	Description
15	0 <sub>2</sub>	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: <ul style="list-style-type: none"> <li>Write Data Parity Error when the PATU is a slave (inbound write).</li> <li>Read Data Parity Error when the PATU, DMA Channel 0, or DMA Channel1 is a master (outbound read).</li> <li>Any Address Parity Error on the Primary Bus (including one generated by the PATU or DMA Channels 0 &amp;1).</li> </ul>
14	0 <sub>2</sub>	P_SERR# Asserted - set when P_SERR# is asserted on the PCI bus by the primary ATU.
13	0 <sub>2</sub>	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 does not cause an ATU interrupt to the core.
12	0 <sub>2</sub>	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 does not cause an ATU interrupt to the core.
11	0 <sub>2</sub>	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	01 <sub>2</sub>	DEVSEL# Timing - These bits are read-only and define the slowest DEVSEL# timing for a target device (except configuration accesses). <ul style="list-style-type: none"> <li>00<sub>2</sub> = Fast</li> <li>01<sub>2</sub> = Medium</li> <li>10<sub>2</sub> = Slow</li> <li>11<sub>2</sub> = Reserved</li> </ul> This primary and secondary ATU interfaces uses Medium timing (01 <sub>2</sub> )





**Table 15-32. Primary ATU Status Register - PATUSR (Sheet 2 of 2)**

Bit	Default	Description
08	0 <sub>2</sub>	Master Parity Error - The primary ATU interface sets this bit under the following conditions: <ul style="list-style-type: none"> <li>The PATU, DMA Channel 0, or DMA Channel 1 asserted S_PERR# itself or the PATU observed S_PERR# asserted.</li> <li>And the PATU, DMA Channel 0, or DMA Channel 1 acted as the bus master for the operation in which the error occurred.</li> <li>And the PATUCMD register's Parity Error Response bit is set</li> </ul> Setting of this bit due to an error condition from either DMA Channel does not cause an ATU interrupt to the core.
07	1 <sub>2</sub>	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 <sub>2</sub>	UDF Supported - User Definable Features are not supported
05	0 <sub>2</sub>	66 MHz. Capable - 66 MHz operation is not supported.
04:00	00000 <sub>2</sub>	Reserved

## 15.7.5 ATU Revision ID Register - ATURID

Revision ID Register bit definitions adhere to *PCI Local Bus Specification* Revision 2.1.

**Table 15-33. ATU Revision ID Register - ATURID**

Bit	Default	Description
07:00	00H	ATU Revision - identifies the i960 RM/RN 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 2.1. Auto configuration software reads this register to determine the PCI device function.

**Table 15-34. ATU Class Code Register - ATUCCR**

Internal Bus Address 1209H	PCI Configuration Address Offset 09H - 0BH	Attribute Legend: RW = Read/Write RV = Reserved RC = Read Clear PR = Preserved RO = Read Only RS = Read/Set 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 2.1. 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 15-35. ATU Cacheline Size Register - ATUCLSR**

Internal Bus Address 120CH	PCI Configuration Address Offset 0CH	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	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 15-36. ATU Latency Timer Register - ATULT**

Bit	Default	Description
07:03	00000 <sub>2</sub>	Programmable Latency Timer - This field varies the latency timer for the primary interface from 0 to 248 clocks.
02:00	000 <sub>2</sub>	Latency Timer Granularity - These Bits are read only giving a programmable granularity of 8 clocks for the latency timer.

Bit	Default	Description
07:03	00000 <sub>2</sub>	Programmable Latency Timer - This field varies the latency timer for the primary interface from 0 to 248 clocks.
02:00	000 <sub>2</sub>	Latency Timer Granularity - These Bits are read only giving a programmable granularity of 8 clocks for the latency timer.

Internal Bus Address  
120DH

PCI Configuration Address Offset  
0DH

Attribute Legend:  
 RW = Read/Write  
 RV = Reserved  
 PR = Preserved  
 RS = Read/Set  
 RC = Read Clear  
 RO = Read Only  
 NA = Not Accessible

## 15.7.9 ATU Header Type Register - ATUHTR

Header Type Register bit definitions adhere to *PCI Local Bus Specification* Revision 2.1. 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 15-22 for using this register in other than its default state.)

**Table 15-37. ATU Header Type Register - ATUHTR**

Bit	Default	Description
07	1 <sub>2</sub>	Single Function/Multi-Function Device - Identifies the ATU as a multi-function PCI device.
06:00	000000 <sub>2</sub>	PCI Header Type - This bit field indicates the type of PCI header implemented. The ATU interface header conforms to <i>PCI Local Bus Specification</i> Revision 2.1.

Bit	Default	Description
07	1 <sub>2</sub>	Single Function/Multi-Function Device - Identifies the ATU as a multi-function PCI device.
06:00	000000 <sub>2</sub>	PCI Header Type - This bit field indicates the type of PCI header implemented. The ATU interface header conforms to <i>PCI Local Bus Specification</i> Revision 2.1.

Internal Bus Address  
120EH

PCI Configuration Address Offset  
0EH

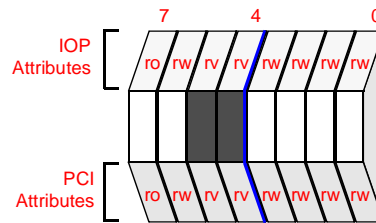
Attribute Legend:  
 RW = Read/Write  
 RV = Reserved  
 PR = Preserved  
 RS = Read/Set  
 RC = Read Clear  
 RO = Read Only  
 NA = Not Accessible

## 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 i960 RM/RN I/O processor replying with the results from the software implementation of the BIST functionality.

**Table 15-38. ATU BIST Register - ATUBISTR**

Bit	Default	Description
07	0 <sub>2</sub>	BIST Capable - This bit value is always equal to the ATUCR ATU BIST Interrupt Enable bit. See <a href="#">Section 15.7.33, "ATU Configuration Register - ATUCR"</a> on page 15-80.
06	0 <sub>2</sub>	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 <sub>2</sub>	Reserved
03:00	0000 <sub>2</sub>	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.



Internal Bus Address  
120FH

PCI Configuration Address Offset  
0FH

Attribute Legend: RW = Read/Write  
RV = Reserved RC = Read Clear  
PR = Preserved RO = Read Only  
RS = Read/Set NA = Not Accessible

## 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 i960 RM/RN I/O Processor Internal Bus with a translated address to map into i960 RM/RN I/O processor local memory. The PIABAR defines the base address and describes the required memory block size; see [Section 15.7.15, on page 15-62](#). Bits 31 through 12 of the PIABAR is 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 2.1.

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 2.1 for additional information on programming base address registers.

**Table 15-39. Primary Inbound ATU Base Address - PIABAR**

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 <sub>2</sub>	Prefetchable Indicator - Defines the memory spaces as prefetchable.
02:01	00 <sub>2</sub>	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 <sub>2</sub>	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.

Internal Bus Address 1210H	PCI Configuration Address Offset 10H - 13H	Attribute Legend: RW = Read/Write RV = Reserved RC = Read Clear PR = Preserved RO = Read Only RS = Read/Set NA = Not Accessible
-------------------------------	---	--

### 15.7.12 ATU Subsystem Vendor ID Register - ASVIR

ATU Subsystem Vendor ID Register bit definitions adhere to *PCI Local Bus Specification* Revision 2.1.

**Table 15-40. ATU Subsystem Vendor ID Register - ASVIR**

Internal Bus Address 122CH	PCI Configuration Address Offset 2CH - 2DH	Attribute Legend: RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set	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 2.1.

**Table 15-41. ATU Subsystem ID Register - ASIR**

Internal Bus Address 122EH	PCI Configuration Address Offset 2EH - 2FH	Attribute Legend: RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set	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 15.7.15](#). Expansion ROM address space (limit size) can be a maximum of 16 MBytes. Bits 31 through 12 of the ERBAR is 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 2.1.

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 2.1 for additional information on programming Expansion ROM base address registers.

**Table 15-42. Expansion ROM Base Address Register -ERBAR**

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 <sub>2</sub>	Address Decode Enable - This bit field shows the ROM address decoder is enabled or disabled. When cleared, indicates the address decoder is disabled.

Internal Bus Address 1230H	PCI Configuration Address Offset 30H - 33H	Attribute Legend: RW = Read/Write RV = Reserved RC = Read Clear PR = Preserved RO = Read Only RS = Read/Set NA = Not Accessible
-------------------------------	---	--

## 15.7.15 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. Table 15-30 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
FFFF8000H	2K	F8000000H	128 M
FFFF0000H	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<sub>2</sub>, 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.



### 15.7.16 ATU Interrupt Line Register - ATUILR

ATU Interrupt Line Register bit definitions adhere to *PCI Local Bus Specification* Revision 2.1. 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**

Internal Bus Address	PCI Configuration Address Offset	Attribute Legend:	RW = Read/Write
123CH	3CH		RV = Reserved
			RC = Read Clear
			PR = Preserved
			RO = Read Only
			RS = Read/Set
			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".	

### 15.7.17 ATU Interrupt Pin Register - ATUIPR

ATU Interrupt Pin Register bit definitions adhere to *PCI Local Bus Specification* Revision 2.1. This register identifies the interrupt pin the ATU and Messaging Unit interface uses. The i960 RM/RN 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**

Internal Bus Address	PCI Configuration Address Offset	Attribute Legend:
123DH	3DH	RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set
		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.

## 15.7.18 ATU Minimum Grant Register - ATUMGNT

ATU Minimum Grant Register bit definitions adhere to *PCI Local Bus Specification Revision 2.1*. 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**

Internal Bus Address 123EH	PCI Configuration Address Offset 3EH	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	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.

## 15.7.19 ATU Maximum Latency Register - ATUMLAT

ATU Maximum Latency Register bit definitions adhere to *PCI Local Bus Specification* Revision 2.1. 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**

Internal Bus Address 123FH	PCI Configuration Address Offset 3FH	Attribute Legend: RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set 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.

## 15.7.20 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 i960 RM/RN 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, on page 15-59](#). When determining block size requirements — as described in [Section 15.7.15, on page 15-62](#) — 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, on page 15-5](#).

The i960 RM/RN 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 i960 RM/RN I/O processor value register are invalid. Refer to the *PCI Local Bus Specification* Revision 2.1 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 has 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**

Bit	Default	Description
31:12	FFFFEH	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 8KB.
11:00	000H	Reserved

Internal Bus Address 1240H	PCI Configuration Address Offset 40H - 43H	Attribute Legend: RW = Read/Write RV = Reserved RC = Read Clear PR = Preserved RO = Read Only RS = Read/Set NA = Not Accessible
-------------------------------	---	--

## 15.7.21 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**

Internal Bus Address		PCI Configuration Address Offset		Attribute Legend:	
1244H		44H - 47H		RW = Read/Write	
				RV = Reserved	
				RC = Read Clear	
				PR = Preserved	
				RO = Read Only	
				RS = Read/Set	
				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 i960 RM/RN I/O processor memory-mapped registers.			
11:00	000H	Reserved			

## 15.7.22 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 i960 RM/RN I/O Processor Internal Bus with a translated address to map into the i960 RM/RN I/O processor internal memory. The SIABAR defines the base address and describes the required memory block size; see [Section 15.7.15, “Determining Block Sizes for Base Address Registers” on page 15-62](#). Bits 31 through 12 of the SIABAR is 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 2.1.

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 2.1 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 15.7.33, “ATU Configuration Register - ATUCR” on page 15-80](#), 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**

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 <sub>2</sub>	Prefetchable Indicator - This bit defines the memory spaces as prefetchable.
02:01	00 <sub>2</sub>	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 <sub>2</sub>	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.

Internal 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
-------------------------------	---	--



### 15.7.23 Secondary Inbound ATU Limit Register - SIALR

Secondary inbound address translation occurs for data transfers occurring from the secondary PCI bus to the i960 RM/RN 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 15.7.22, “Secondary Inbound ATU Base Address Register - SIABAR” on page 15-69. When determining the block size requirements as described in section Section 15.7.15, “Determining Block Sizes for Base Address Registers” on page 15-62, 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 15-5.

The programmed value within the i960 RM/RN 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 i960 RM/RN I/O processor value register is invalid. The default value for the limit register is FFFFE00H, which is a 8 KByte limit. Refer to the *PCI Local Bus Specification* Revision 2.1 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 has 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**

		Internal Bus Address 124CH	PCI Configuration Address Offset 4CH - 4FH
<b>Bit</b>	<b>Default</b>	<b>Description</b>	
31:12	FFFEH	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	



## 15.7.24 Secondary Inbound ATU Translate Value Register - SIATVR

The Secondary Inbound ATU Translate Value Register (SIATVR) contains the i960 RM/RN 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 i960 RM/RN 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**

Internal Bus Address		PCI Configuration Address Offset		Attribute Legend: RW = Read/Write	
1250H		50H - 53H		RV = Reserved RC = Read Clear	
				PR = Preserved RO = Read Only	
				RS = Read/Set 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 i960 RM/RN I/O Processor Internal Bus. (The default value of the entire register is 0000 1000H.)			
11:00	000H	Reserved			



## 15.7.25 Primary Outbound Memory Window Value Register - POMWVR

The Primary Outbound Memory Window Value Register (POMWVR) contains the primary PCI address used to convert i960 RM/RN 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 15-13 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**

Internal Bus Address	PCI Configuration Address Offset	Attribute Legend:	
1254H	54H - 57H	RW = Read/Write	RV = Reserved
		PR = Preserved	RC = Read Clear
		RS = Read/Set	RO = Read Only
			NA = Not Accessible
Bit	Default	Description	
31:04	0000 000H	Primary Outbound MW Value - Used to convert i960 RM/RN I/O Processor Internal Bus addresses to PCI addresses.	
03:02	00 <sub>2</sub>	Reserved	
01:00	00 <sub>2</sub>	Burst Order - This bit field shows the address sequence during a memory burst. Only linear incrementing mode is supported.	

## 15.7.26 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 15-13 for details on outbound address translation.

The primary I/O window is from i960 RM/RN 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**

IOP Attributes	31	28	24	20	16	12	8	4	0				
	rw	rw	rw	rw	rw	rw	rw	rw	rw	rv	rv	rv	rv
PCI Attributes	rw	rw	rw	rw	rw	rw	rw	rw	rw	rv	rv	rv	rv
	Internal Bus Address 125CH				PCI Configuration Address Offset 5CH - 5FH				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 I/O Window Value - Used to convert internal bus addresses to PCI addresses.											
03:00	0H	Reserved											

## 15.7.27 Primary Outbound DAC Window Value Register - PODWVR

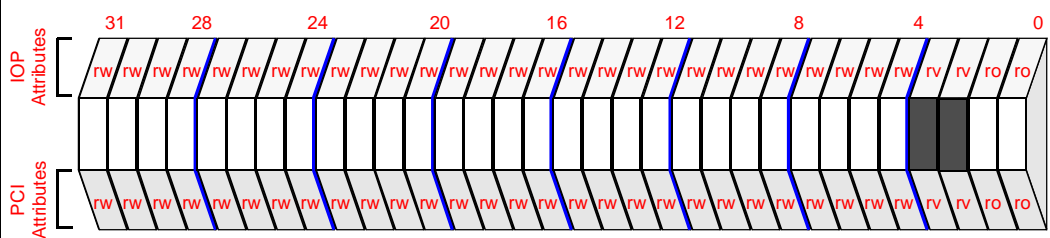
The Primary Outbound DAC Window Value Register (PODWVR) contains the primary PCI DAC address used to convert a i960 RM/RN I/O Processor Internal Bus address. This address is driven on the primary PCI bus as a result of the primary outbound ATU address translation. See [Section 15.2.2.1, “Outbound Address Translation” on page 15-13](#) 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 i960 RM/RN 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**

Bit	Default	Description
31:04	0000 000H	Primary Outbound DAC Window Value - This value the primary ATU uses to convert i960 RM/RN I/O Processor Internal Bus addresses to PCI addresses.
03:02	00 <sub>2</sub>	Reserved
01:00	00 <sub>2</sub>	Burst Order - This bit field shows the address sequence during a memory burst. Only linear incrementing mode is supported.

Internal Bus Address 1260H	PCI Configuration Address Offset 60H - 63H	<b>Attribute Legend:</b> RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set RC = Read Clear RO = Read Only NA = Not Accessible
-------------------------------	---	---

## 15.7.28 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**

		Internal Bus Address	PCI Configuration Address Offset	Attribute Legend:																															
		1264H	64H - 67H	RW = Read/Write	RV = Reserved																														
				PR = Preserved	RC = Read Clear																														
				RS = Read/Set	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).																																	

## 15.7.29 Secondary Outbound Memory Window Value Register - SOMWVR

The Secondary Outbound Memory Window Value Register (SOMWVR) contains the secondary PCI address used to convert i960 RM/RN 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 15-13 for details on outbound address translation.

The secondary memory window is from i960 RM/RN 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**

Bit	Default	Description
31:04	0000 000H	Secondary Outbound Memory Window Value - Used to convert i960 RM/RN I/O Processor Internal Bus addresses to PCI addresses.
03:02	00 <sub>2</sub>	Reserved
01:00	00 <sub>2</sub>	Burst Order - This bit field shows the address sequence during a memory burst. Only linear incrementing mode is supported.

### 15.7.30 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 i960 RM/RN 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 15-13 for details on outbound address translation.

The secondary I/O window is from i960 RM/RN 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**

Internal Bus Address 126CH	PCI Configuration Address Offset 6CH - 6FH	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	Secondary Outbound I/O Window Value - Used to convert internal bus addresses to PCI addresses.
03:00	0H	Reserved



### 15.7.31 Expansion ROM Limit Register - ERLR

The Expansion ROM Limit Register (ERLR) defines the block size of addresses the primary ATU defines 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**

		Internal Bus Address	PCI Configuration Address Offset	Attribute Legend:																													
		1274H	74H - 77H	RW = Read/Write	RV = Reserved																												
				RC = Read Clear	PR = Preserved																												
				RO = Read Only	RS = Read/Set																												
				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																															



### 15.7.32 Expansion ROM Translate Value Register - ERTVR

The Expansion ROM Translate Value Register contains the i960 RM/RN I/O Processor Internal Bus address which the primary ATU converts 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**

Bit	Default	Description
31:12	00000H	Expansion ROM Translation Value - Used to convert PCI addresses to i960 RM/RN 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

Internal Bus Address 1278H	PCI Configuration Address Offset 78H - 7BH	Attribute Legend: RW = Read/Write RV = Reserved RC = Read Clear PR = Preserved RO = Read Only RS = Read/Set NA = Not Accessible
-------------------------------	---	--



### 15.7.33 ATU Configuration Register - ATUCR

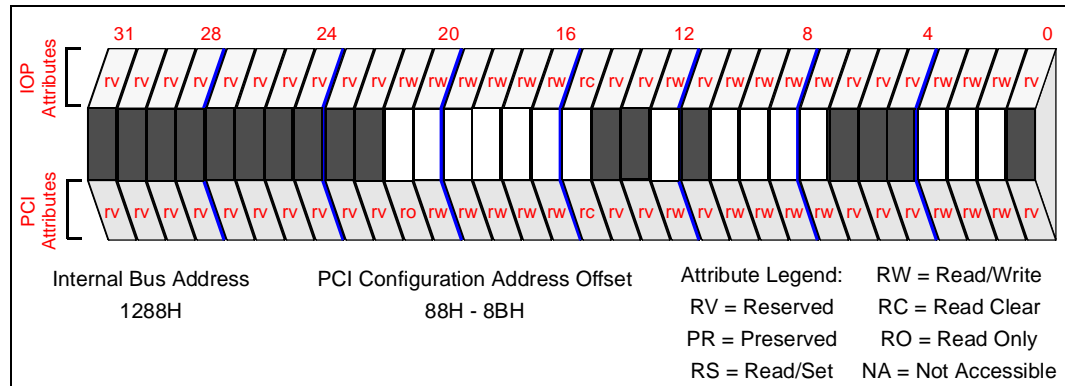
The ATU Configuration Register controls the outbound address translation for both the primary and secondary outbound translation units. It also contains bits for DRC aliasing, discard timer status, P\_SERR# and S\_SERR# manual assertion, access to the messaging unit from the secondary PCI bus, P\_SERR# and S\_SERR# detection interrupt masking, and ATU Bist interrupt enabling.

Table 15-62. ATU Configuration Register - ATUCR (Sheet 1 of 2)

Bit	Default	Description
31:22	00H	Reserved
21	0 <sub>2</sub>	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 i960 RM/RN I/O processor as a single-function device with either the ATU or the Bridge as the single function. (Refer to <a href="#">Section 15.2.4, "PCI Multi-Function Device Swapping/Disabling"</a> on page 15-22 for programming information.)
20	0 <sub>2</sub>	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 <sub>2</sub>	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 <sub>2</sub>	Direct Addressing Upper 2Gbytes Translation Enable - When set, with Direct Addressing enabled (Bit 07 of the ATUCR set), the ATU forwards 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 occurs.
17	0 <sub>2</sub>	S_SERR# Manual Assertion - when set, the SATU asserts S_SERR# for one clock on the secondary PCI interface. Until cleared, S_SERR# may not be manually asserted again. Once cleared, operation proceeds as specified.
16	0 <sub>2</sub>	P_SERR# Manual Assertion - when set, the PATU asserts P_SERR# for one clock on the primary PCI interface. Until cleared, P_SERR# may not be manually asserted again. Once cleared, operation proceeds as specified.
15	0 <sub>2</sub>	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 <sub>2</sub>	Reserved

Table 15-62. ATU Configuration Register - ATUCR (Sheet 2 of 2)

Bit	Default	Description
12	0 <sub>2</sub>	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 defaults to the bridge for forwarding to the MU on the primary interface. If clear, the SATU has priority and claims 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 simultaneously allows the Messaging Unit to claim target cycles which are mastered by the primary interface of the Bridge Unit.
11	0 <sub>2</sub>	Reserved
10	0 <sub>2</sub>	S_SERR# Interrupt Enable - When set, the i960 core processor is signalled an NMI# interrupt if the SATU detects that S_SERR# was asserted on the secondary interface. When clear, the i960 core processor is not interrupted when S_SERR# is detected as asserted on the secondary interface.
09	0 <sub>2</sub>	P_SERR# Interrupt Enable - When set, the i960 core processor is signalled an NMI# interrupt if the PATU detects that P_SERR# was asserted on the primary interface. When clear, the i960 core processor is not interrupted when P_SERR# is detected as asserted on the primary interface.
08	0 <sub>2</sub>	Direct Addressing Enable - Setting this bit enables direct outbound addressing through the ATUs. Internal bus cycles with an address between 0000.2000H and 7FFF.FFFFH is automatically 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 <sub>2</sub>	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 <sub>2</sub>	Reserved
03	0 <sub>2</sub>	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 <sub>2</sub>	Secondary Outbound ATU Enable - When set, enables the secondary outbound address translation unit. When cleared, disables the secondary outbound ATU.
01	0 <sub>2</sub>	Primary Outbound ATU Enable - When set, enables the primary outbound address translation unit. When cleared, disables the primary outbound ATU.
00	0 <sub>2</sub>	Reserved





### 15.7.34 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 i960 RM/RN 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 i960 RM/RN 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, bit 9 and bit 7 can result in an NMI# interrupt being driven to the i960 core processor.

**Table 15-63. Primary ATU Interrupt Status Register - PATUISR (Sheet 1 of 2)**

Internal Bus Address	PCI Configuration Address Offset	Attribute Legend:	RW = Read/Write
1290H	90H - 93H	RV = Reserved	RC = Read Clear
		PR = Preserved	RO = Read Only
		RS = Read/Set	NA = Not Accessible
Bit	Default	Description	
31:11	000000H	Reserved	
10	0 <sub>2</sub>	P_SERR# Asserted - set when P_SERR# is asserted on the PCI bus by the primary ATU.	
09	0 <sub>2</sub>	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: <ul style="list-style-type: none"> <li>Write Data Parity Error when the PATU is a slave (inbound write).</li> <li>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.</li> <li>Any Address Parity Error on the Primary Bus (including one generated by the PATU or DMA Channels 0 &amp; 1 when loopback is enabled).</li> </ul>	
08	0 <sub>2</sub>	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 <sub>2</sub>	Internal Bus Master Abort - set when a transaction initiated by the ATU internal bus master interface ends in a Master-abort.	
06:05	00 <sub>2</sub>	Reserved	
04	0 <sub>2</sub>	P_SERR# Detected - set when P_SERR# is detected on the PCI bus by the primary ATU.	
03	0 <sub>2</sub>	PCI Master Abort - set when a transaction initiated by the ATU PCI master interface ends in a Master-abort.	

**Table 15-63. Primary ATU Interrupt Status Register - PATUISR (Sheet 2 of 2)**

<p>IOP Attributes</p> <p>PCI Attributes</p>																																	
		<p>Internal Bus Address 1290H</p>	<p>PCI Configuration Address Offset 90H - 93H</p>	<p>Attribute Legend:                      RW = Read/Write                      RV = Reserved                      PR = Preserved                      RS = Read/Set</p>	<p>RW = Read/Write                      RC = Read Clear                      RO = Read Only                      NA = Not Accessible</p>																												
Bit	Default	Description																															
02	0 <sub>2</sub>	PCI Target Abort (master) - set when a transaction initiated by the ATU PCI master interface ends in a Target-abort.																															
01	0 <sub>2</sub>	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 <sub>2</sub>	PCI Master Parity Error - The ATU interface sets this bit when three conditions are met: <ul style="list-style-type: none"> <li>the PATU asserted S_PERR# or observed S_PERR# asserted</li> <li>the PATU acted as the bus master for the operation in which the error occurred</li> <li>Parity Error Response bit is set (in the Primary ATU Command Register)</li> </ul>																															



### 15.7.35 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 i960 RM/RN 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 i960 RM/RN 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 15-64. Secondary ATU Interrupt Status Register - SATUISR (Sheet 1 of 2)**

Internal Bus Address	PCI Configuration Address Offset	Attribute Legend:	RW = Read/Write
1294H	94H - 97H	RV = Reserved	RC = Read Clear
		PR = Preserved	RO = Read Only
		RS = Read/Set	NA = Not Accessible
Bit	Default	Description	
31:11	000000H	Reserved	
10	0 <sub>2</sub>	S_SERR# Asserted - set when S_SERR# is asserted on the PCI bus by the secondary ATU.	
09	0 <sub>2</sub>	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: <ul style="list-style-type: none"> <li>• Write Data Parity Error when the SATU is a slave (inbound write).</li> <li>• 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.</li> <li>• Any Address Parity Error on the Secondary Bus (including one generated by the SATU or DMA Channel 2 when loopback is enabled).</li> </ul>	
08	0 <sub>2</sub>	Reserved	
07	0 <sub>2</sub>	Internal Bus Master Abort - set when a transaction initiated by the ATU internal bus master interface ends in a Master-abort.	
06:05	00 <sub>2</sub>	Reserved	
04	0 <sub>2</sub>	S_SERR# Detected - set when S_SERR# is detected on the PCI bus by the secondary ATU.	
03	0 <sub>2</sub>	PCI Master Abort - set when a transaction initiated by the ATU PCI master interface ends in a Master-abort.	
02	0 <sub>2</sub>	PCI Target Abort (master) - set when a transaction initiated by the ATU PCI master interface ends in a Target-abort.	

**Table 15-64. Secondary ATU Interrupt Status Register - SATUISR (Sheet 2 of 2)**

Internal Bus Address		PCI Configuration Address Offset																Attribute Legend:				RW = Read/Write											
1294H		94H - 97H																RV = Reserved				RC = Read Clear											
						PR = Preserved				RO = Read Only				RS = Read/Set		NA = Not Accessible																	
Bit	Default	Description																															
01	0 <sub>2</sub>	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 <sub>2</sub>	PCI Master Parity Error - The secondary ATU interface sets this bit when three conditions are met: <ul style="list-style-type: none"> <li>the SATU asserted S_PERR# or observed S_PERR# asserted</li> <li>the SATU acted as the bus master for the operation in which the error occurred</li> <li>Parity Error Response bit is set (in the Secondary ATU Command Register)</li> </ul>																															

## 15.7.36 Secondary ATU Command Register - SATUCMD

Secondary ATU Command Register bits adhere to the definitions in the *PCI Local Bus Specification* Revision 2.1 and in most cases affect the behavior of the device on the secondary PCI bus.

**Table 15-65. Secondary ATU Command Register - SATUCMD**

Bit	Default	Description
15:10	00H	Reserved
09	0 <sub>2</sub>	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 <sub>2</sub>	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 <sub>2</sub>	Wait Cycle Control - controls address/data stepping. Not implemented and a reserved bit field
06	0 <sub>2</sub>	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 <sub>2</sub>	VGA Palette Snoop Enable - The secondary ATU interface does not support I/O writes and therefore, does not perform VGA palette snooping.
04	0 <sub>2</sub>	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 <sub>2</sub>	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 <sub>2</sub>	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 a DMA transfer on the PCI bus.
01	0 <sub>2</sub>	Memory Enable - Controls the secondary ATU interface's response to PCI memory addresses. When this bit is cleared, the ATU interface does not respond to any memory access on the PCI bus.
00	0 <sub>2</sub>	I/O Space Enable - Controls the ATU interface response to I/O transactions on the primary side. Not implemented and a reserved bit field.

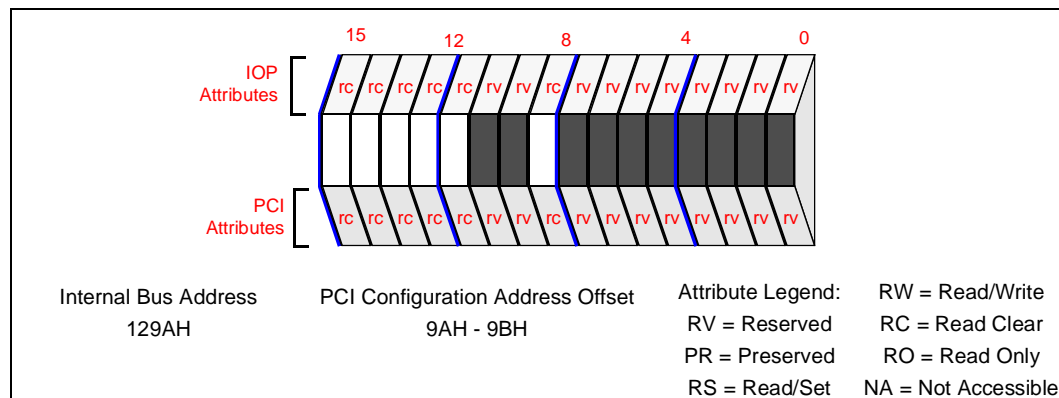


## 15.7.37 Secondary ATU Status Register - SATUSR

Secondary ATU Status Register bits adhere to the definitions in the *PCI Local Bus Specification* Revision 2.1. 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<sub>2</sub> to the register.

**Table 15-66. Secondary ATU Status Register - SATUSR**

Bit	Default	Description
15	0 <sub>2</sub>	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: <ul style="list-style-type: none"> <li>Write Data Parity Error when the SATU is a slave (inbound write).</li> <li>Read Data Parity Error when the SATU or DMA Channel 2 is a master (outbound read).</li> <li>Any Address Parity Error on the Secondary Bus (including one generated by the SATU or DMA Channel 2).</li> </ul>
14	0 <sub>2</sub>	S_SERR# Asserted - set when S_SERR# is asserted by the secondary ATU.
13	0 <sub>2</sub>	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 does not cause an ATU interrupt to the core.
12	0 <sub>2</sub>	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 does not cause an ATU interrupt to the core.
11	0 <sub>2</sub>	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 <sub>2</sub>	Reserved
08	0 <sub>2</sub>	Master Parity Error - The secondary ATU interface sets this bit under the following conditions: <ul style="list-style-type: none"> <li>The SATU or DMA Channel 2 asserted S_PERR# itself or the SATU observed S_PERR# asserted.</li> <li>And the SATU or DMA Channel 2 acted as the bus master for the operation in which the error occurred.</li> <li>And the SATUCMD register's Parity Error Response bit is set</li> </ul> Setting of this bit due to an error condition from a DMA Channel does not cause an ATU interrupt to the core.
07:00	00H	Reserved



### 15.7.38 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 i960 RM/RN 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 15-13 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 i960 RM/RN I/O Processor Internal Bus address 8C00 000H to 8FFF FFFFH with the fixed length of 64 Mbytes.

**Table 15-67. Secondary Outbound DAC Window Value Register - SODWVR**

Internal Bus Address	PCI Configuration Address Offset	Attribute Legend: RW = Read/Write RV = Reserved RC = Read Clear PR = Preserved RO = Read Only RS = Read/Set NA = Not Accessible
129CH	9CH - 9FH	
Bit	Default	Description
31:04	0000 000H	Secondary Outbound DAC Window Value - The secondary ATU uses this value to convert i960 RM/RN I/O Processor Internal Bus addresses to PCI addresses.
03:02	00 <sub>2</sub>	Reserved
01:00	00 <sub>2</sub>	Burst Order - This bit field shows the address sequence during a memory burst. Only linear incrementing mode is supported.

### 15.7.39 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 15-68. Secondary Outbound Upper 64-bit DAC Register - SOUDR**

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).

Internal 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
-------------------------------	---	--

## 15.7.40 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 15-69. Primary Outbound Configuration Cycle Address Register - POCCAR**

IOP Attributes	31	28	24	20	16	12	8	4	0
	rw	rw	rw	rw	rw	rw	rw	rw	rw
PCI Attributes	ro	ro	ro	ro	ro	ro	ro	ro	ro
	ro	ro	ro	ro	ro	ro	ro	ro	ro
Internal Bus Address		PCI Configuration Address Offset		Attribute Legend:		RW = Read/Write			
12A4H		A4H - A7H		RV = Reserved		RC = Read Clear			
				PR = Preserved		RO = Read Only			
				RS = Read/Set		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.							

## 15.7.41 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 15-70. Secondary Outbound Configuration Cycle Address Register - SOCCAR**

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.

Internal Bus Address: 12A8H  
 PCI Configuration Address Offset: A8H - ABH

Attribute Legend:  
 RW = Read/Write  
 RV = Reserved  
 PR = Preserved  
 RS = Read/Set  
 RC = Read Clear  
 RO = Read Only  
 NA = Not Accessible



## 15.7.42 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 reads or writes 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\_ORQ. 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 i960 RM/RN I/O Processor Internal Bus address space and appears as a reserved value within the ATU configuration space.

**Table 15-71. Primary Outbound Configuration Cycle Data Register - POCCDR**

		Internal Bus Address	Attribute Legend:
		12ACH	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	Primary Configuration Cycle Data - These bits define the data used during an outbound configuration read or write cycle.	

### 15.7.43 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 reads or writes 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 i960 RM/RN I/O Processor Internal Bus address space and appears as a reserved value within the ATU configuration space.

**Table 15-72. Secondary Outbound Configuration Cycle Data Register - SOCCDR**

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.

Internal Bus Address 12B0H	Attribute Legend: RW = Read/Write RV = Reserved RC = Read Clear PR = Preserved RO = Read Only RS = Read/Set NA = Not Accessible
-------------------------------	--

### 15.7.44 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 15-73. Primary ATU Interrupt Mask Register - PATUIMR**

Bit	Default	Description
31:08	0000 00H	Reserved
07	0 <sub>2</sub>	PATU Detected Parity Error Interrupt Mask - When set, a parity error detected on the primary PCI bus that sets bit 15 of the PATUSR does <i>not</i> result in bit 9 of the PATUISR being set. When clear, an error that sets bit 15 of the PATUSR results in bit 9 of the PATUISR being set.
06	0 <sub>2</sub>	PATU P_SERR# Asserted Interrupt Mask - When set, asserting P_SERR# on the primary interface resulting in bit 14 of the PATUSR being set does not result in bit 10 of the PATUISR being set. When clear, an error that sets bit 14 of the PATUSR causes 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 <sub>2</sub>	PATU PCI Master Abort Interrupt Mask - When set, a master abort error resulting in bit 13 of the PATUSR being set does <i>not</i> result in bit 3 of the PATUISR being set. When clear, an error that sets bit 13 of the PATUSR causes bit 3 of the PATUISR to be set.
04	0 <sub>2</sub>	PATU PCI Target Abort (Master) Interrupt Mask- When set, a target abort error resulting in bit 12 of the PATUSR being set does <i>not</i> result in bit 2 of the PATUISR being set. When clear, an error that sets bit 12 of the PATUSR causes bit 2 of the PATUISR to be set.
03	0 <sub>2</sub>	PATU PCI Target Abort (Target) Interrupt Mask- When set, a target abort error resulting in bit 11 of the PATUSR being set does <i>not</i> result in bit 1 of the PATUISR being set. When clear, an error that sets bit 11 of the PATUSR causes bit 1 of the PATUISR to be set.
02	0 <sub>2</sub>	PATU PCI Master Parity Error Interrupt Mask - When set, a parity error resulting in bit 8 of the PATUSR being set does <i>not</i> result in bit 0 of the PATUISR being set. When clear, an error that sets bit 8 of the PATUSR causes bit 0 of the PATUISR to be set.
01	0 <sub>2</sub>	Primary ATU Inbound Error P_SERR# Enable - When set, the PATU asserts (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# is not asserted under the previous conditions.
00	0 <sub>2</sub>	Primary ATU ECC Target Abort Enable - When set, the PATU performs 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.



## 15.7.45 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 15-74. Secondary ATU Interrupt Mask Register - SATUIMR**

Bit	Default	Description
31:08	0000 00H	Reserved
07	0 <sub>2</sub>	SATU Detected Parity Error Interrupt Mask - When set, a parity error detected on the secondary PCI bus that sets bit 15 of the SATUSR does <i>not</i> result in bit 09 of the SATUISR being set. When clear, an error that sets bit 15 of the SATUSR results in bit 09 of the SATUISR being set.
06	0 <sub>2</sub>	SATU S_SERR# Asserted Interrupt Mask - When set, asserting S_SERR# on the secondary interface resulting in bit 14 of the SATUSR being set does not result in bit 10 of the SATUISR being set. When clear, an error that sets bit 14 of the SATUSR causes 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 <sub>2</sub>	SATU PCI Master Abort Interrupt Mask - When set, a master abort error resulting in bit 13 of the SATUSR being set does <i>not</i> result in bit 3 of the SATUISR being set. When clear, an error that sets bit 13 of the SATUSR causes bit 3 of the SATUISR to be set.
04	0 <sub>2</sub>	SATU PCI Target Abort (Master) Interrupt Mask - When set, a target abort error resulting in bit 12 of the SATUSR being set does <i>not</i> result in bit 2 of the SATUISR being set. When clear, an error that sets bit 12 of the SATUSR causes bit 2 of the SATUISR to be set.
03	0 <sub>2</sub>	SATU PCI Target Abort (Target) Interrupt Mask - When set, a target abort error resulting in bit 11 of the SATUSR being set does <i>not</i> result in bit 1 of the SATUISR being set. When clear, an error that sets bit 11 of the SATUSR causes bit 1 of the SATUISR to be set.
02	0 <sub>2</sub>	SATU PCI Master Parity Error Interrupt Mask - When set, a parity error resulting in bit 8 of the SATUSR being set does <i>not</i> result in bit 0 of the PATUISR being set. When clear, an error that sets bit 8 of the SATUSR causes bit 0 of the SATUISR to be set.
01	0 <sub>2</sub>	Secondary ATU Inbound Error S_SERR# Enable - When set, the SATU asserts (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# is not asserted under the previous conditions.
00	0 <sub>2</sub>	Secondary ATU ECC Target Abort Enable - When set, the SATU performs 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.



This chapter describes the Messaging Unit (MU) of the i960<sup>®</sup> RM/RN I/O Processor. The MU is closely related to the Primary Address Translation Unit (PATU) described in [Chapter 15, “Address Translation Unit”](#).

## 16.1 Overview

The Messaging Unit provides a mechanism for data to be transferred between the PCI system and the 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 RM/RN I/O processor to communicate through message passing and interrupt generation. The four mechanisms are:

- **Message Registers** — allow the i960 RM/RN I/O 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 RM/RN I/O 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 i960 RM/RN 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 i960 RM/RN 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 i960 RM/RN 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 i960 RM/RN 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, “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 i960 RM/RN 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”](#) 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 16-14](#)). 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

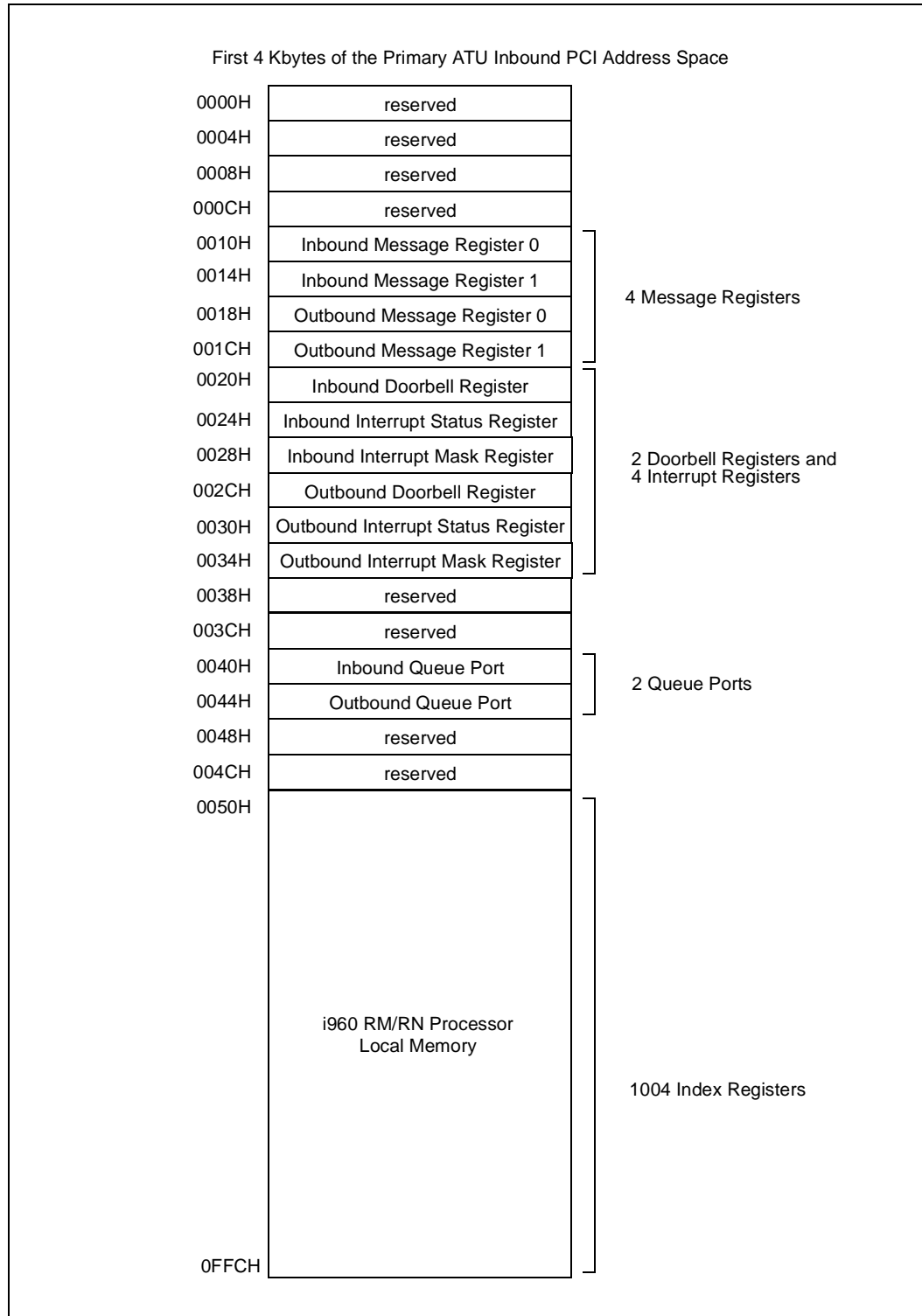


Table 16-1 provides a summary of the four messaging mechanisms used in the Messaging Unit.

**Table 16-1. MU Summary**

Mechanism	Quantity	Assert PCI Interrupt Signals?	Generate i960® Core Processor Interrupt?
Message Registers	2 Inbound 2 Outbound	Optional	Optional
Doorbell Registers	1 Inbound 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 RM/RN I/O 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 i960 RM/RN I/O processor. Outbound messages are sent by the i960 RM/RN I/O 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, “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, “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 RM/RN I/O processor and free outbound messages flow toward the i960 RM/RN I/O 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 i960 RM/RN I/O processor	Written
Inbound Free Queue	Queue for empty inbound messages from the i960 RM/RN I/O processor available for use by other processors	Read
Outbound Post Queue	Queue for outbound messages from the i960 RM/RN I/O 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 RM/RN I/O processor	Written

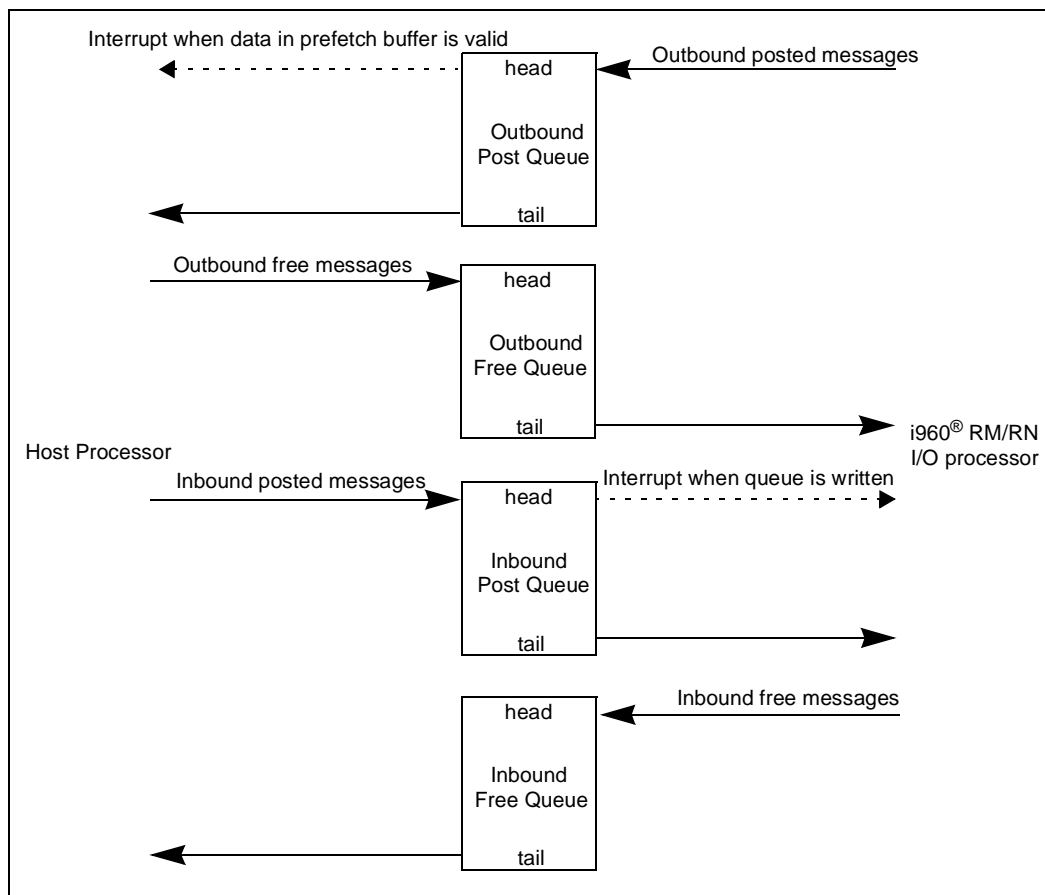
The two outbound queues allow the i960 core processor to post outbound messages in one queue and to receive free messages returning from the host processor. The i960 core processor posts outbound messages, the host processor receives the posted message and when it is finished with the message, places 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 RM/RN I/O processor in one queue and to receive free messages returning from the i960 RM/RN I/O 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 i960 RM/RN 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 16-25). 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.

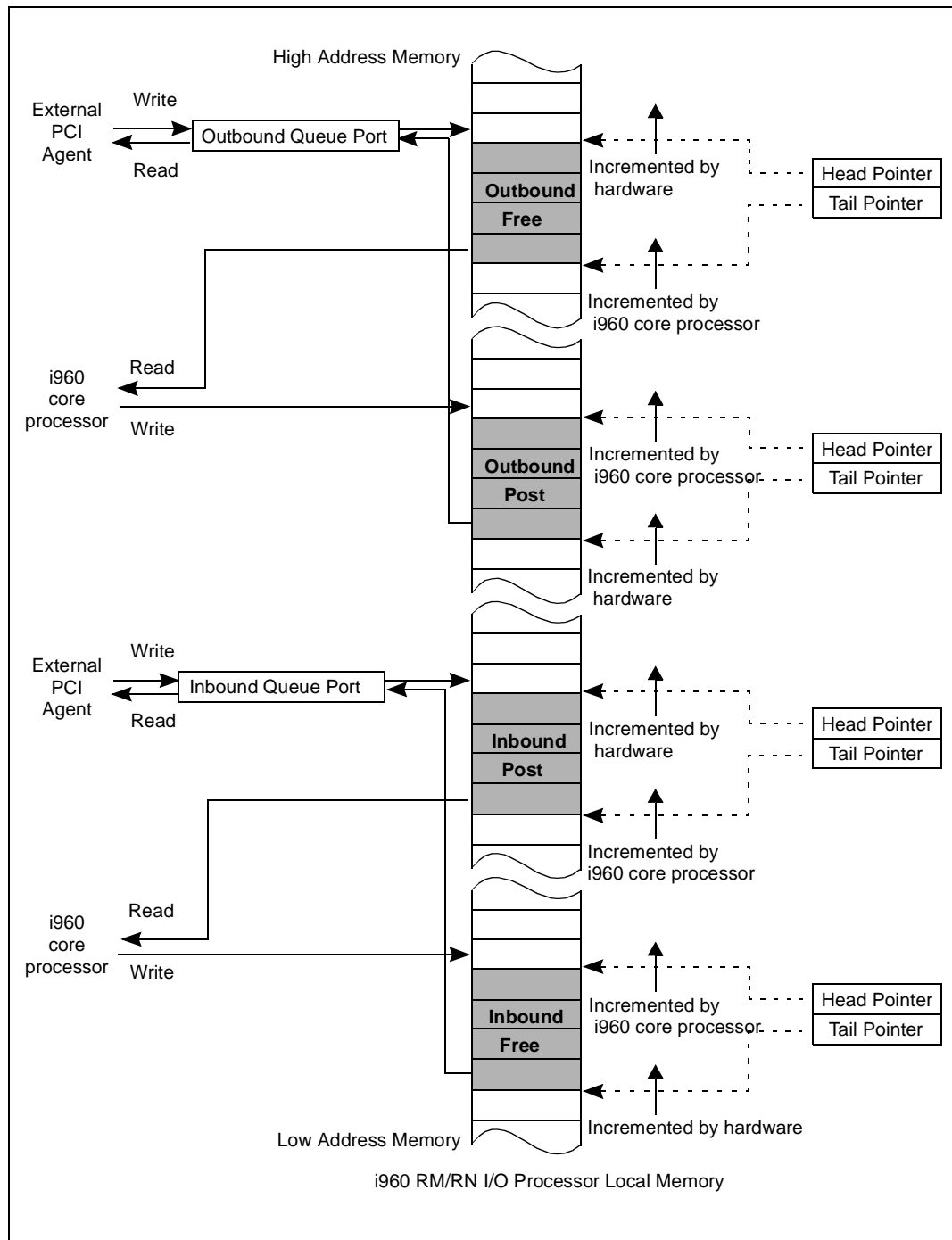
There is one base address for all four queues. It is stored in the Queue Base Address Register (QBAR). The starting addresses of each queue is based on the Queue Base Address and the Queue Size field. [Table 16-4](#) shows an example of how the circular queues should be set up based on the *Intelligent I/O (I<sub>2</sub>O) 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 <sup>®</sup> 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 i960 RM/RN 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, “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, “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, “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 16-17
1314H	Section 16.8.1, "Inbound Message Register - IMRx" on page 16-17
1318H	Section 16.8.2, "Outbound Message Register - OMRx" on page 16-17
131CH	Section 16.8.2, "Outbound Message Register - OMRx" on page 16-17
1320H	Section 16.8.3, "Inbound Doorbell Register - IDR" on page 16-18
1324H	Section 16.8.4, "Inbound Interrupt Status Register - IISR" on page 16-19
1328H	Section 16.8.5, "Inbound Interrupt Mask Register - IIMR" on page 16-20
132CH	Section 16.8.6, "Outbound Doorbell Register - ODR" on page 16-21
1330H	Section 16.8.7, "Outbound Interrupt Status Register - OISR" on page 16-22
1334H	Section 16.8.8, "Outbound Interrupt Mask Register - OIMR" on page 16-23
1350H	Section 16.8.9, "MU Configuration Register - MUCR" on page 16-24
1354H	Section 16.8.10, "Queue Base Address Register - QBAR" on page 16-25
1360H	Section 16.8.11, "Inbound Free Head Pointer Register - IFHPR" on page 16-26
1364H	Section 16.8.12, "Inbound Free Tail Pointer Register - IFTPR" on page 16-27
1368H	Section 16.8.13, "Inbound Post Head Pointer Register - IPHPR" on page 16-28
136CH	Section 16.8.14, "Inbound Post Tail Pointer Register - IPTPR" on page 16-29
1370H	Section 16.8.15, "Outbound Free Head Pointer Register - OFHPR" on page 16-30
1374H	Section 16.8.16, "Outbound Free Tail Pointer Register - OFTPR" on page 16-31
1378H	Section 16.8.17, "Outbound Post Head Pointer Register - OPHPR" on page 16-32
137CH	Section 16.8.18, "Outbound Post Tail Pointer Register - OPTPR" on page 16-33
1380H	Section 16.8.19, "Index Address Register - IAR" on page 16-34

## 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.

80960RM/RN internal bus address	Attribute Legend:
IMR0 1310H	RW = Read/Write RV = Reserved RC = Read Clear PR = Preserved RO = Read Only RS = Read/Set NA = Not Accessible
IMR1 1314H	

## 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**

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.

80960RM/RN internal bus address	Attribute Legend:
OMR0 1318H	RW = Read/Write RV = Reserved RC = Read Clear PR = Preserved RO = Read Only RS = Read/Set NA = Not Accessible
OMR1 131CH	

### 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**

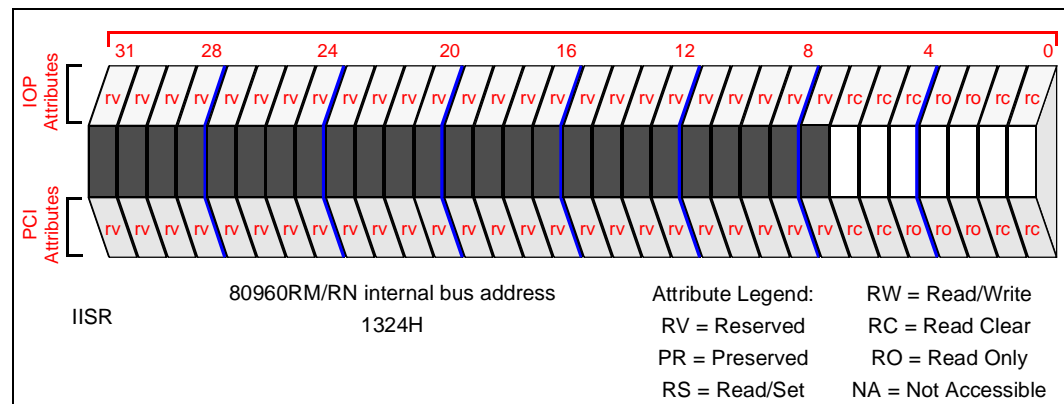
		<p>Attribute Legend:</p> <ul style="list-style-type: none"> <li>RW = Read/Write</li> <li>RV = Reserved</li> <li>RC = Read Clear</li> <li>PR = Preserved</li> <li>RO = Read Only</li> <li>RS = Read/Set</li> <li>NA = Not Accessible</li> </ul>
Bit	Default	Description
31	0 <sub>2</sub>	NMI Interrupt - Generate 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 three 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**

Bit	Default	Description
31:07	0000000H 0 <sub>2</sub>	Reserved
06	0 <sub>2</sub>	Index Register Interrupt - This bit is set by the MU hardware when an Index Register is written after a PCI transaction.
05	0 <sub>2</sub>	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 <sub>2</sub>	Inbound Post Queue Interrupt - This bit is set by the MU hardware when the Inbound Post Queue has been written.
03	0 <sub>2</sub>	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 <sub>2</sub>	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 <sub>2</sub>	Inbound Message 1 Interrupt - This bit is set by the MU hardware when the Inbound Message 1 Register has been written.
00	0 <sub>2</sub>	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 <sub>2</sub>	Reserved
06	0 <sub>2</sub>	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 <sub>2</sub>	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 <sub>2</sub>	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 <sub>2</sub>	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 <sub>2</sub>	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 <sub>2</sub>	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 <sub>2</sub>	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**

Bit	Default	Description
31	0 <sub>2</sub>	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 <sub>2</sub>	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 <sub>2</sub>	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 <sub>2</sub>	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.

ODR	80960RM/RN internal bus address 132CH	Attribute Legend: RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set	RW = Read/Write RC = Read Clear RO = Read Only NA = Not Accessible
-----	--	--	---

## 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**

Bit	Default	Description
31:08	000000H	Reserved
07	0 <sub>2</sub>	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 <sub>2</sub>	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 <sub>2</sub>	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 <sub>2</sub>	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 <sub>2</sub>	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 <sub>2</sub>	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 <sub>2</sub>	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 <sub>2</sub>	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.

OISR	80960RM/RN internal bus address 1330H	<b>Attribute Legend:</b> RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set RO = Read Only NA = Not Accessible
------	--	--



## 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 <sub>2</sub>	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 <sub>2</sub>	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 <sub>2</sub>	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 <sub>2</sub>	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 <sub>2</sub>	Outbound Post Queue Interrupt Mask - When set, this bit masks the PCI interrupt generated when data in the prefetch buffer is valid.
02	0 <sub>2</sub>	Outbound Doorbell Interrupt Mask - When set, this bit masks the Software Interrupt generated by the Outbound Doorbell Register.
01	0 <sub>2</sub>	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 <sub>2</sub>	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**

MUCR      80960RM/RN internal bus address      1350H		Attribute Legend: RW = Read/Write RV = Reserved      RC = Read Clear PR = Preserved      RO = Read Only RS = Read/Set      NA = Not Accessible
Bit	Default	Description
31:06	0000000H 0 <sub>2</sub>	Reserved
05:01	00001 <sub>2</sub>	Circular Queue Size - This field determines the size of each Circular Queue. All four queues are the same size. <ul style="list-style-type: none"> <li>• 00001<sub>2</sub> - 4K Entries (16 Kbytes)</li> <li>• 00010<sub>2</sub> - 8K Entries (32 Kbytes)</li> <li>• 00100<sub>2</sub> - 16K Entries (64 Kbytes)</li> <li>• 01000<sub>2</sub> - 32K Entries (128 Kbytes)</li> <li>• 10000<sub>2</sub> - 64K Entries (256 Kbytes)</li> </ul>
00	0 <sub>2</sub>	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.



### 16.8.11 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**

IFHPR	80960RM/RN internal bus address 1360H	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 <sub>2</sub>	Inbound Free Head Pointer - Local memory offset of the head pointer for the Inbound Free Queue.
01:00	00 <sub>2</sub>	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 <sub>2</sub>	Inbound Free Tail Pointer - Local memory offset of the tail pointer for the Inbound Free Queue.
01:00	00 <sub>2</sub>	Reserved

IFTPR	80960RM/RN internal bus address 1364H	Attribute Legend: RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set RC = Read Clear RO = Read Only NA = Not Accessible
-------	--	--

### 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**

IPHPR	80960RM/RN internal bus address 1368H	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 <sub>2</sub>	Inbound Post Head Pointer - Local memory offset of the head pointer for the Inbound Post Queue.
01:00	00 <sub>2</sub>	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 <sub>2</sub>	Inbound Post Tail Pointer - Local memory offset of the tail pointer for the Inbound Post Queue.
01:00	00 <sub>2</sub>	Reserved

IPTPR 80960RM/RN internal bus address 136CH	Attribute Legend: RW = Read/Write RV = Reserved RC = Read Clear PR = Preserved RO = Read Only RS = Read/Set NA = Not Accessible
---	--

## 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**

OFHPR	80960RM/RN internal bus address 1370H	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 <sub>2</sub>	Outbound Free Head Pointer - Local memory offset of the head pointer for the Outbound Free Queue.
01:00	00 <sub>2</sub>	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 <sub>2</sub>	Outbound Free Tail Pointer - Local memory offset of the tail pointer for the Outbound Free Queue.
01:00	00 <sub>2</sub>	Reserved

OFTPR	80960RM/RN internal bus address 1374H	Attribute Legend: RW = Read/Write RV = Reserved RC = Read Clear PR = Preserved RO = Read Only RS = Read/Set NA = Not Accessible
-------	--	--





## 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**

		IOP Attributes																PCI Attributes															
		80960RM/RN internal bus address																Attribute Legend:															
		IAR																RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set RC = Read Clear RO = Read Only NA = Not Accessible															
		1380H																															
Bit	Default	Description																															
31:12	000000H	Reserved																															
11:02	00H 00 <sub>2</sub>	Index Address - is the local memory offset of the Index Register written (050H to FFCH)																															
01:00	00 <sub>2</sub>	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.

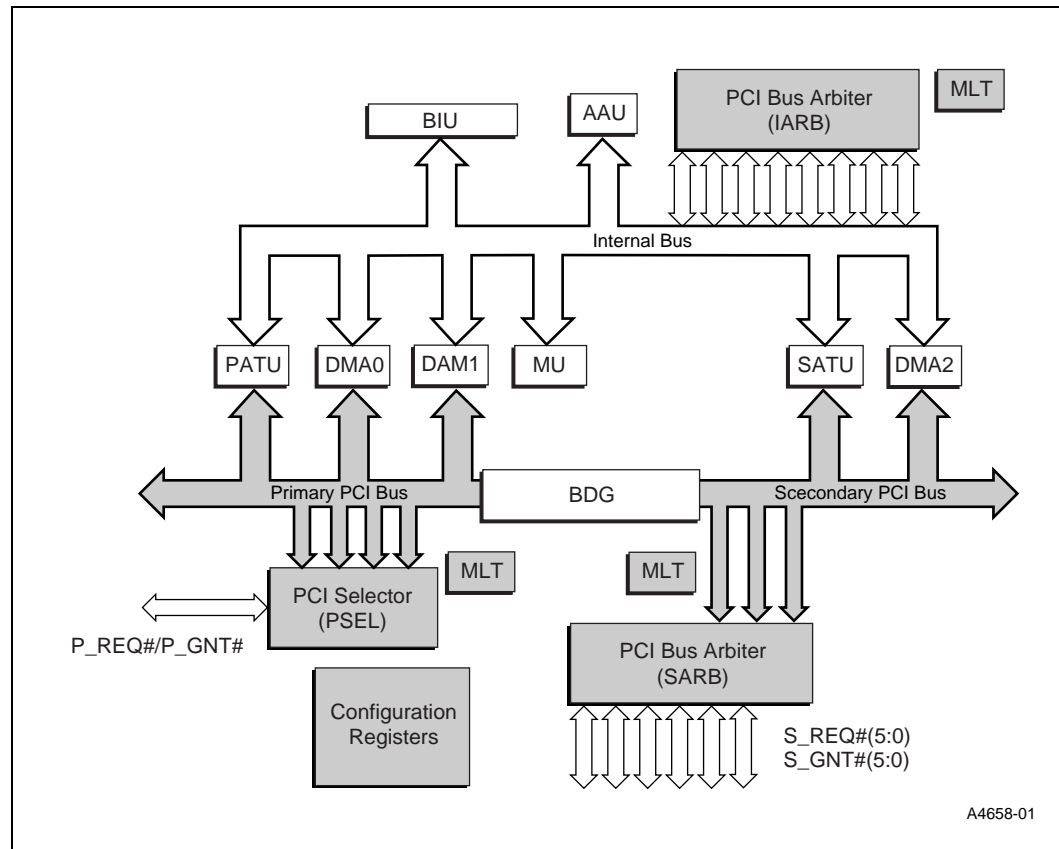
# i960<sup>®</sup> RM/RN I/O Processor Arbitration

This chapter describes the components which comprise i960<sup>®</sup> RM/RN 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 i960 RM/RN 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 i960 RM/RN 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. i960<sup>®</sup> RM/RN I/O Processor Arbitration Block Diagram



The four components which comprise i960 RM/RN 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 i960 RM/RN 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 i960 RM/RN 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 17-9) - The PCI selector arbitrates between the i960 RM/RN 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 17-9) - 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 i960 RM/RN 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 17-11) - 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 2.1 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 2.1 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

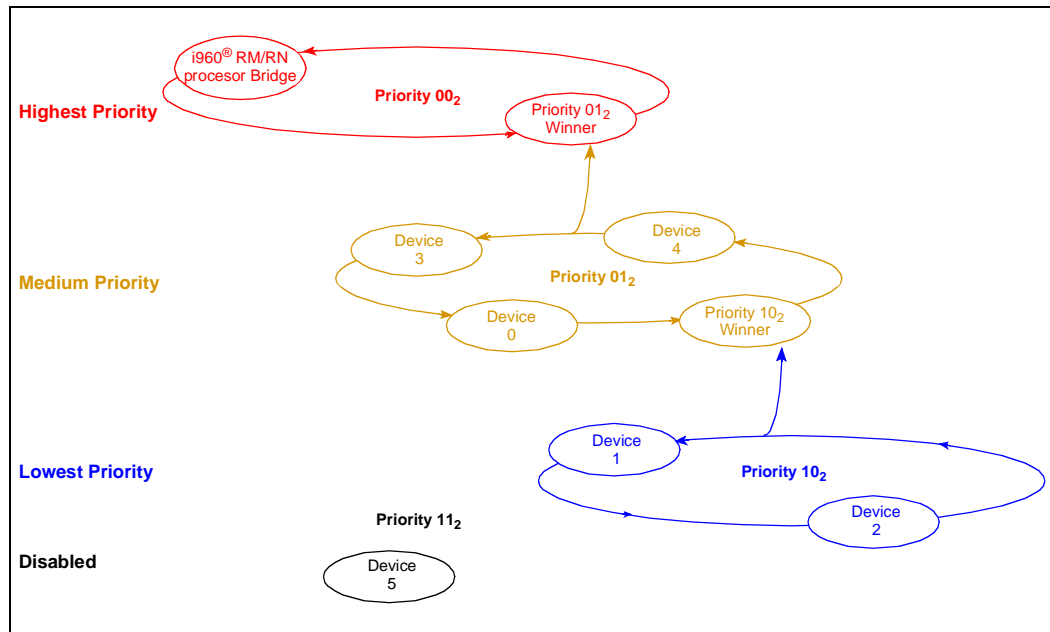
The purpose of the PCI Arbiter is to provide a fair arbitration scheme for all masters on the PCI bus. The PCI Arbiter adheres to all the requirements of the *PCI Local Bus Specification*, Revision 2.1.

### 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
i960 RM/RN I/O Processor Bridge	High - 00 <sub>2</sub>
Device 0	Medium - 01 <sub>2</sub>
Device 3	Medium - 01 <sub>2</sub>
Device 4	Medium - 01 <sub>2</sub>
Device 1	Low - 10 <sub>2</sub>
Device 2	Low - 10 <sub>2</sub>
Device 5	Disabled - 11 <sub>2</sub>

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.

### Example 17-1. 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 would follow as:

**ABACABACABACABAC**



### Example 17-2. 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	DE	DE	DE	CE	CE	CE	CDF	CDF	CDF
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.

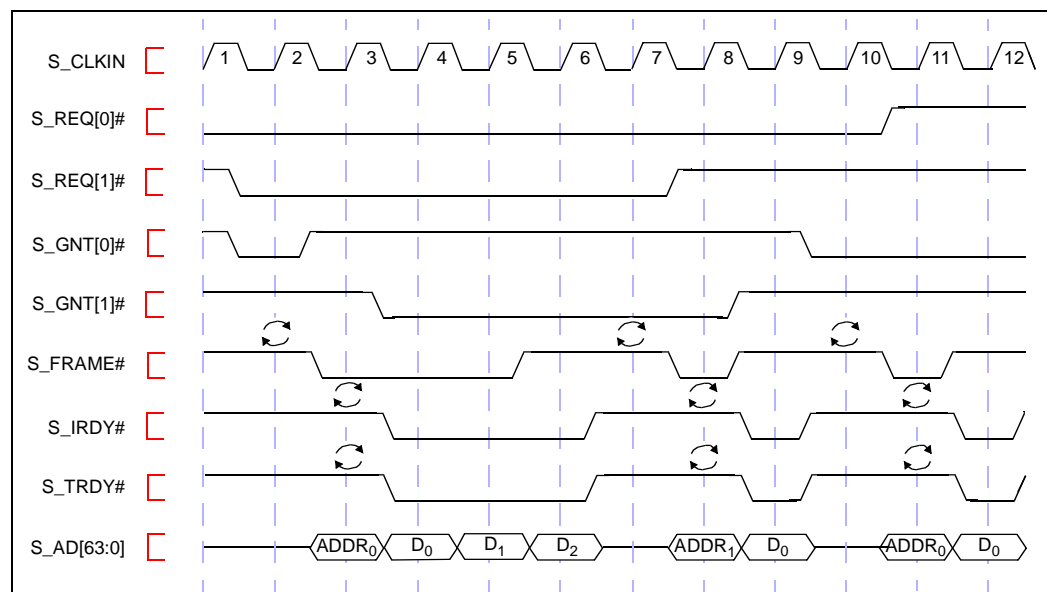
Continuing with Table 17-3, the winning bus master pattern would follow as:

**ABCABDABFABCABDABEABCABDABF**

#### 17.2.1.2 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 2.1 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](#).

**Note:** The PCI Arbiter arbitrates the PCI bus by checking REQ[8:0]# on every cycle independent of any transactions on the bus<sup>1</sup>.

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 2.1 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.

---

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.

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 ( <a href="#">Section 17.2.1.1, “Priority Mechanism” on page 17-3</a> ).

### 17.2.1.3 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 i960 RM/RN I/O processor. Per the *PCI Local Bus Specification*, Revision 2.1, 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 i960 RM/RN 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 i960 RM/RN 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 17-8](#) 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.

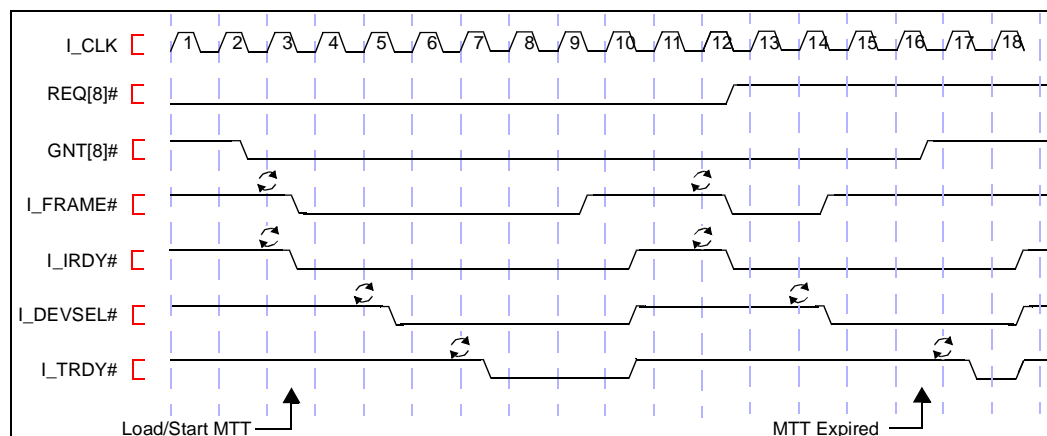
The MTT overcomes this potential bottleneck by guaranteeing a programmed timeslice during which the BIU is granted the internal bus. Once the IARB grants the internal bus to the BIU and the BIU initially asserts I\_FRAME#, the MTT is loaded with the value programmed in the Multi-Transaction Timer Register (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 17-9](#). 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 i960 RM/RN I/O processor. i960 RM/RN 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 i960 RM/RN I/O processor: DMA0, DMA1, PATU, and BDG. If one of the i960 RM/RN 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 i960 RM/RN 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 i960 RM/RN I/O processor. Per the *PCI Local Bus Specification*, Revision 2.1, 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 i960 RM/RN 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 2.1.

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 17-12 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 17-12
Section 17.6.2, "Internal Arbitration Control Register - IACR" on page 17-13
Section 17.6.3, "Master Latency Timer Register - MLTR" on page 17-14
Section 17.6.4, "Multi-Transaction Timer Register - MTTR" on page 17-14



## 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 i960 RM/RN I/O processor core.

**Table 17-7. Secondary Arbitration Control Register - SACR**

Bit	Default	Description
31:14	0	Reserved
13:12	10 <sub>2</sub>	Device 5 Priority
11:10	10 <sub>2</sub>	Device 4 Priority
9:8	10 <sub>2</sub>	Device 3 Priority
7:6	10 <sub>2</sub>	Device 2 Priority
5:4	10 <sub>2</sub>	Device 1 Priority
3:2	10 <sub>2</sub>	Device 0 Priority
1:0	00 <sub>2</sub>	Secondary PCI Interface Priority (Bridge, DMA Channel 2, or Secondary ATU)

Internal Bus Address: 104CH

Attribute Legend:  
 RW = Read/Write  
 RV = Reserved  
 PR = Preserved  
 RS = Read/Set  
 RC = Read Clear  
 RO = Read Only  
 NA = Not Accessible

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 i960 RM/RN I/O processor.

**Table 17-8. 2-Bit Priorities**

2-Bit Programmed Value	Priority Level
00 <sub>2</sub>	High Priority
01 <sub>2</sub>	Medium Priority
10 <sub>2</sub>	Low Priority
11 <sub>2</sub>	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 i960 RM/RN I/O processor core.

**Table 17-9. Internal Arbitration Control Register - IACR**

Bit	Default	Description
31:14	0	Reserved
13:12	00 <sub>2</sub>	Application Accelerator Priority
11:10	00 <sub>2</sub>	BIU Priority
9:8	00 <sub>2</sub>	DMA Channel 2 Priority
7:6	00 <sub>2</sub>	DMA Channel 1 Priority
5:4	00 <sub>2</sub>	DMA Channel 0 Priority
3:2	00 <sub>2</sub>	Secondary ATU Priority
1:0	00 <sub>2</sub>	Primary ATU and Messaging Unit Priority

Internal Bus Address 1600H		Attribute Legend:	RC = Read Clear
		RV = Reserved	RO = Read Only
		PR = Preserved	NA = Not Accessible
		RS = Read/Set	

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.



This chapter describes the i960® RM/RN I/O processor’s 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

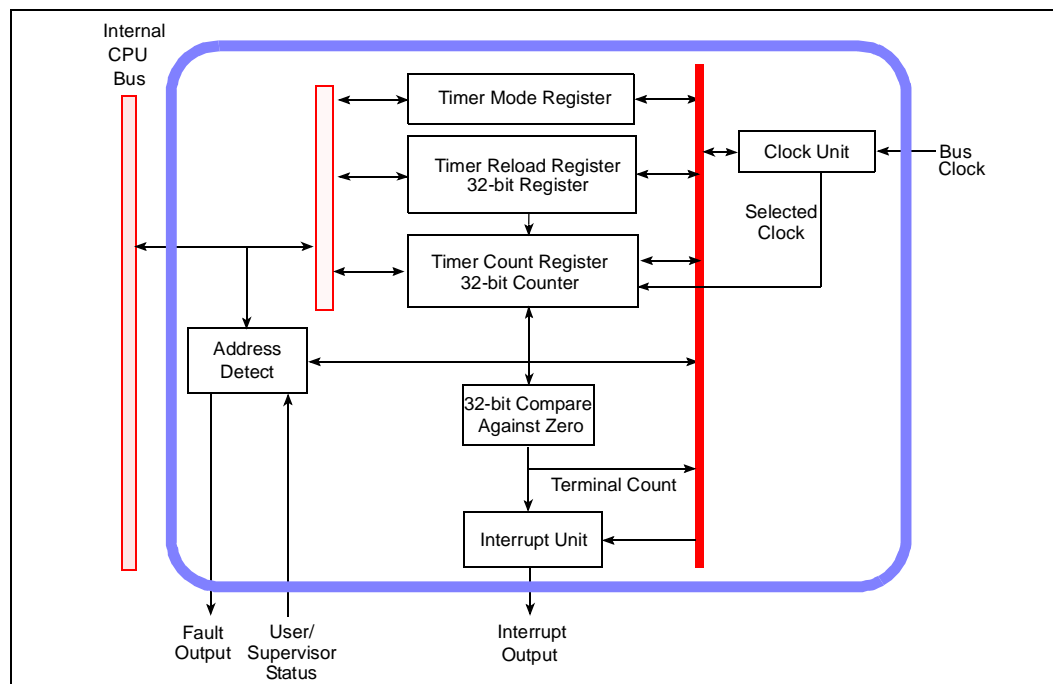


Table 18-1. Timer Performance Ranges

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 C-3 “Timer Registers”](#) on page C-4.

## 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	31	28	24	20	16	12	8	4	0
	rv	rv	rv	rv	rv	rv	rv	rv	rv
PCI	na	na	na	na	na	na	na	na	na
<b>LBA:</b>	CH 0-0308H		<b>Legend:</b>		NA = Not Accessible		RO = Read Only		
	CH 1-0318H		RV = Reserved		PR = Preserved		RW = Read/Write		
<b>PCI:</b>	NA		RS = Read/Set		RC = Read Clear		LBA = 80960 local bus address		
			LBA = 80960 local bus address		PCI = PCI Configuration Address Offset				
<b>Bit</b>	<b>Default</b>	<b>Description</b>							
31:06	0000 000H	Reserved. Initialize to 0.							
05:04	00 <sub>2</sub>	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 <sub>2</sub>	Timer Register Supervisor Write Control - TMRx.sup (0) Supervisor and User Mode Write Enabled (1) Supervisor Mode Only Write Enabled							
02	0 <sub>2</sub>	Timer Auto Reload Enable - TMRx.reload (0) Auto Reload Disabled (1) Auto Reload Enabled							
01	0 <sub>2</sub>	Timer Enable - TMRx.enable (0) Disabled (1) Enabled							
00	0 <sub>2</sub>	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, “i960® RM/RN 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, “i960® RM/RN 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, “i960® RM/RN 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**

<b>LBA:</b>	CH 0-0304H CH 1-0314H	<b>Legend:</b>	NA = Not Accessible      RO = Read Only RV = Reserved          PR = Preserved          RW = Read/Write RS = Read/Set          RC = Read Clear LBA = 80960 local bus address      PCI = PCI Configuration Address Offset
<b>PCI:</b>	na		
<b>Bit</b>	<b>Default</b>	<b>Description</b>	
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**

<b>LBA:</b>	CH 0-0300H CH 1-0310H	<b>Legend:</b>	NA = Not Accessible RV = Reserved RS = Read/Set LBA = 80960 local bus address	RO = Read Only PR = Preserved RC = Read Clear PCI = PCI Configuration Address Offset
<b>PCI:</b>	NA			
<b>Bit</b>	<b>Default</b>	<b>Description</b>		
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.
X	X	N	0	1	Timer enabled, TMRx.enable is cleared when TCRx decrements to zero.
X	N	N	1	1	Timer and auto reload enabled, TMRx.enable remains set when TCRx=0. When TCRx=0, TCRx equals the TRRx value.
0	X	X	X	X	No faults for user mode writes are generated.
1	X	X	X	X	TYPE.MISMATCH fault generated on user mode write.

**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**

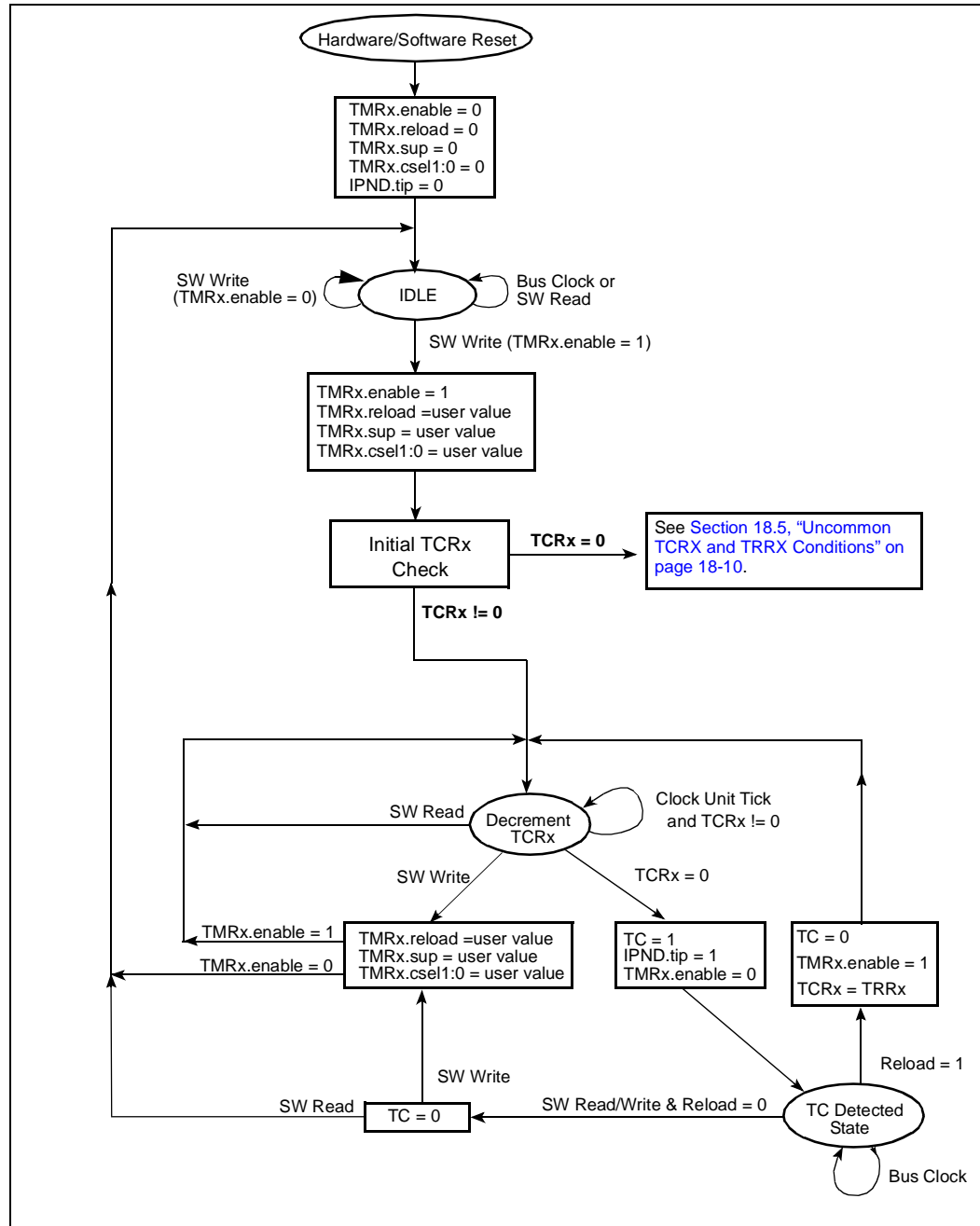
TRRx	TCRx	Bit 2 (TMRx.reload)	Bit 1 (TMRx.enable)	Action
X	0	0	1	TMRx.tc and TINTx set, TMR.enable cleared
0	0	1	1	Timer and auto reload enabled, TINTx not generated and timer enable remains set.
0	N	1	1	Timer and auto reload enabled. TINT.x set when TCRx=0. The timer remains enabled but further TINTx's are not generated.
N	0	1	1	Timer and auto reload enabled, TINTx not set initially, TCRx = TRRx, TINTx set when TCRx has completely decremented the value it loaded from TRRx. TMRx.enable remains set.

**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





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

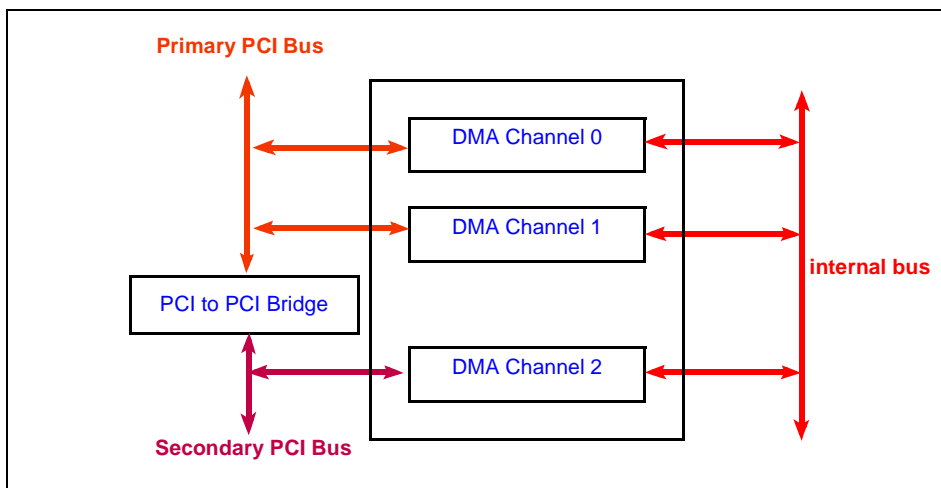
The DMA Controller provides low-latency, high-throughput data transfer capability. The DMA Controller optimizes block transfers of data between the PCI bus and the local processor memory. The DMA is an initiator on the PCI bus with burst capabilities providing a maximum throughput of 132 Mbytes/sec at 33 MHz when the PCI bus is operating in 32-bit mode. When the PCI bus is operating in 64-bit mode, the maximum throughput is 264 Mbytes/sec at 33 MHz.

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 i960® RM/RN I/O Processor Memory Controller Interface
- $2^{32}$  addressing range on the i960 RM/RN I/O processor 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 264 Mbytes/sec burst support for both the PCI bus and the i960 RM/RN I/O processor processor internal bus
- Direct addressing to and from the PCI bus
- Fully programmable from the i960 core processor
- Support for automatic data chaining for gathering and scattering of data blocks
- 64-bit PCI and i960 RM/RN I/O processor local memory interface

Figure 19-1 shows the connections of the DMA channels to the PCI buses.

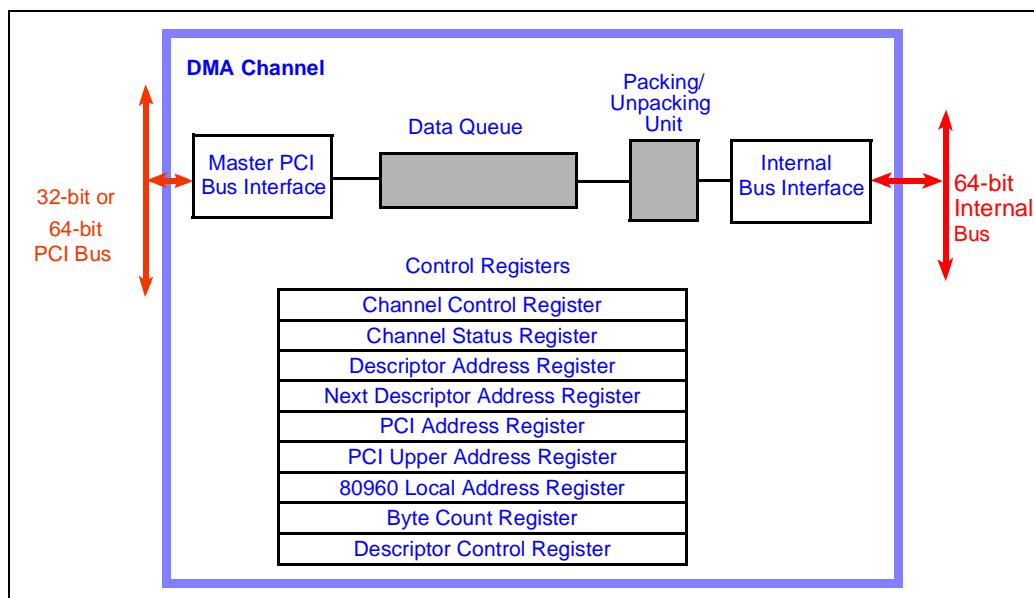
**Figure 19-1. DMA Controller**



## 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 i960 RM/RN I/O processor local memory. Channel 2 transfers blocks of data between the secondary PCI bus and i960 RM/RN 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 unlimited burst lengths.

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 i960 RM/RN 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 i960 RM/RN 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 packing and unpacking unit. This unit enables data transfers from or to unaligned addresses in either the PCI address space or the i960 RM/RN I/O processor local address space. All combinations of unaligned data are supported with the packing and unpacking 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 i960 RM/RN I/O processor internal bus, the DMA controller attempts 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
i960 RM/RN I/O Processor Local Address Register	LADR	i960 RM/RN I/O Processor 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 i960 RM/RN I/O processor local memory. [Figure 19-3](#) shows the format of an individual chain descriptor. Every descriptor requires six contiguous words in i960 RM/RN I/O processor 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 i960 RM/RN I/O processor 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 ignores bits 04:00 of this address.
- The second word is the lower 32-bit PCI source/destination address. This address is 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 is used during Dual Address Cycles for driving 64-bit PCI addresses. The address is ignored if DAC is disabled. This value is loaded into the PCI Upper Address Register.
- The fourth word is the i960 RM/RN I/O processor source/destination address. This address is driven on the internal bus. This value is loaded into the i960 RM/RN I/O processor Local Address Register.
- The fifth word is the Byte Count value. This value determines the number of bytes to transfer. This value is 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 is loaded into the Descriptor Control Register.

There are no data alignment requirements for either the PCI address or the i960 RM/RN I/O processor address. However, maximum performance is obtained from aligned transfers, especially small transfers ([Section 19.7, on page 19-13](#)).

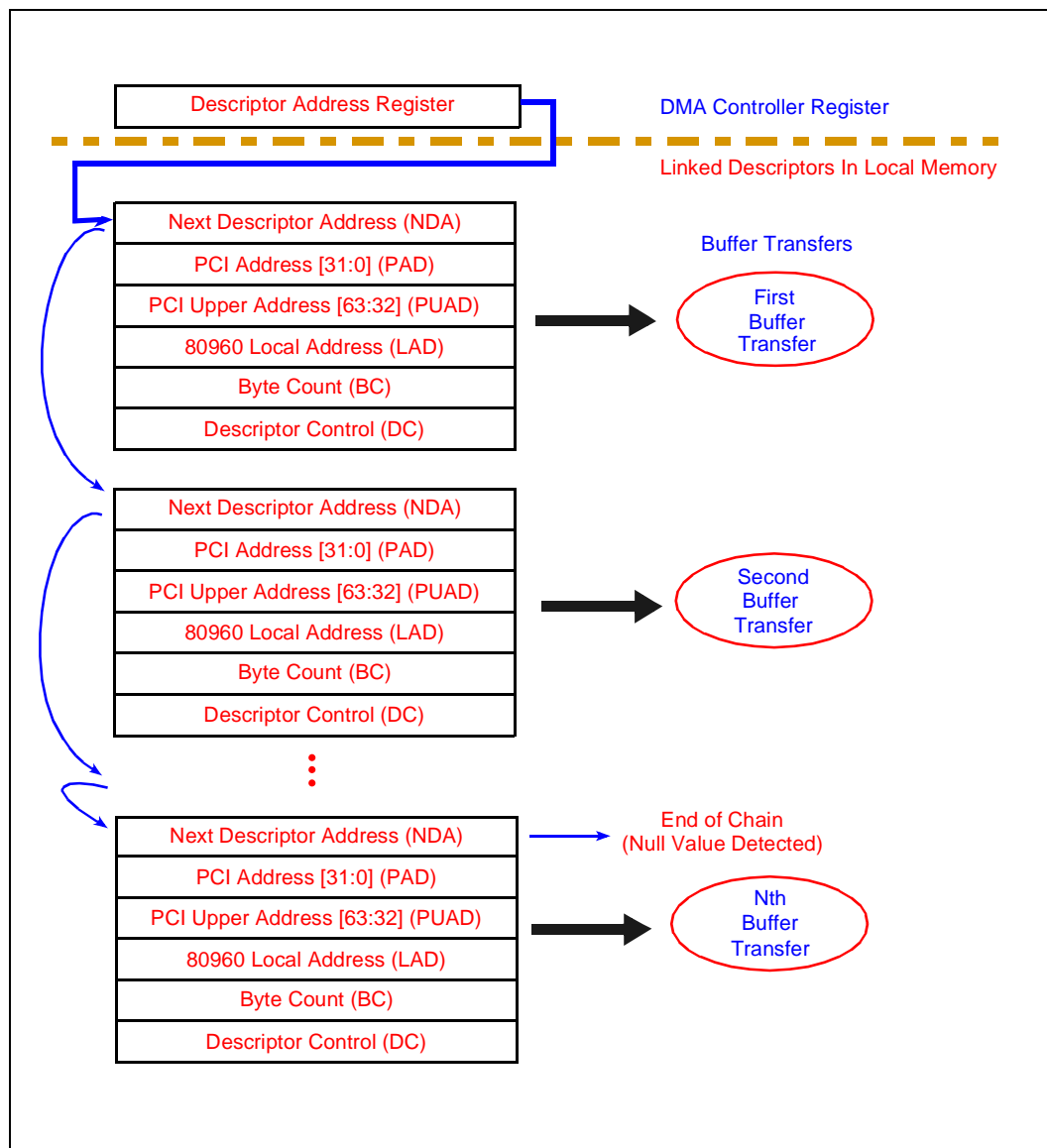
Refer to [Section 19.14](#) for additional descriptions about the DMA Controller registers.

**Figure 19-3. DMA Chain Descriptor**

Chain Descriptor in 80960 Memory	Description
Next Descriptor Address (NDA)	Address of Next Chain Descriptor
PCI Address [31:0] (PAD)	Lower 32-bit PCI Source/Destination Address
PCI Upper Address [63:32] (PUAD)	Upper 32-bit PCI Source/Destination Address
80960 Local Address (LAD)	80960 Local Source/Destination Address
Byte Count (BC)	Number of Bytes to Transfer
Descriptor Control (DC)	Descriptor Control

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 i960 RM/RN I/O processor 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 i960 RM/RN I/O 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 i960 RM/RN I/O processor 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 i960 RM/RN I/O processor 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 now contains the address of the chain descriptor just read and the Next Descriptor Address Register now contains the Next Descriptor Address from the chain descriptor just read.

The last descriptor in the DMA chain list has 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.

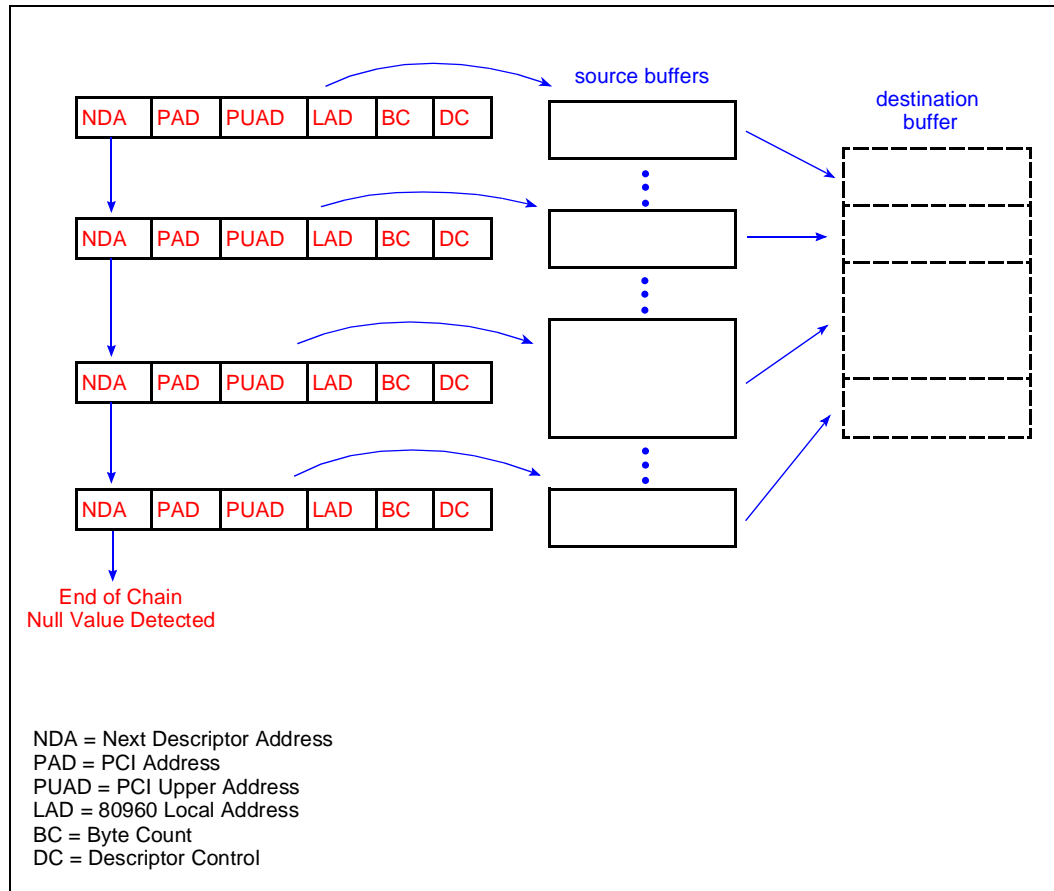
When descriptors are read from external memory, bus latency and memory speed affect chaining latency. Chaining latency is defined as the time required for the channel to access the next chain descriptor plus the time required to set up for the next DMA transfer.

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 is 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

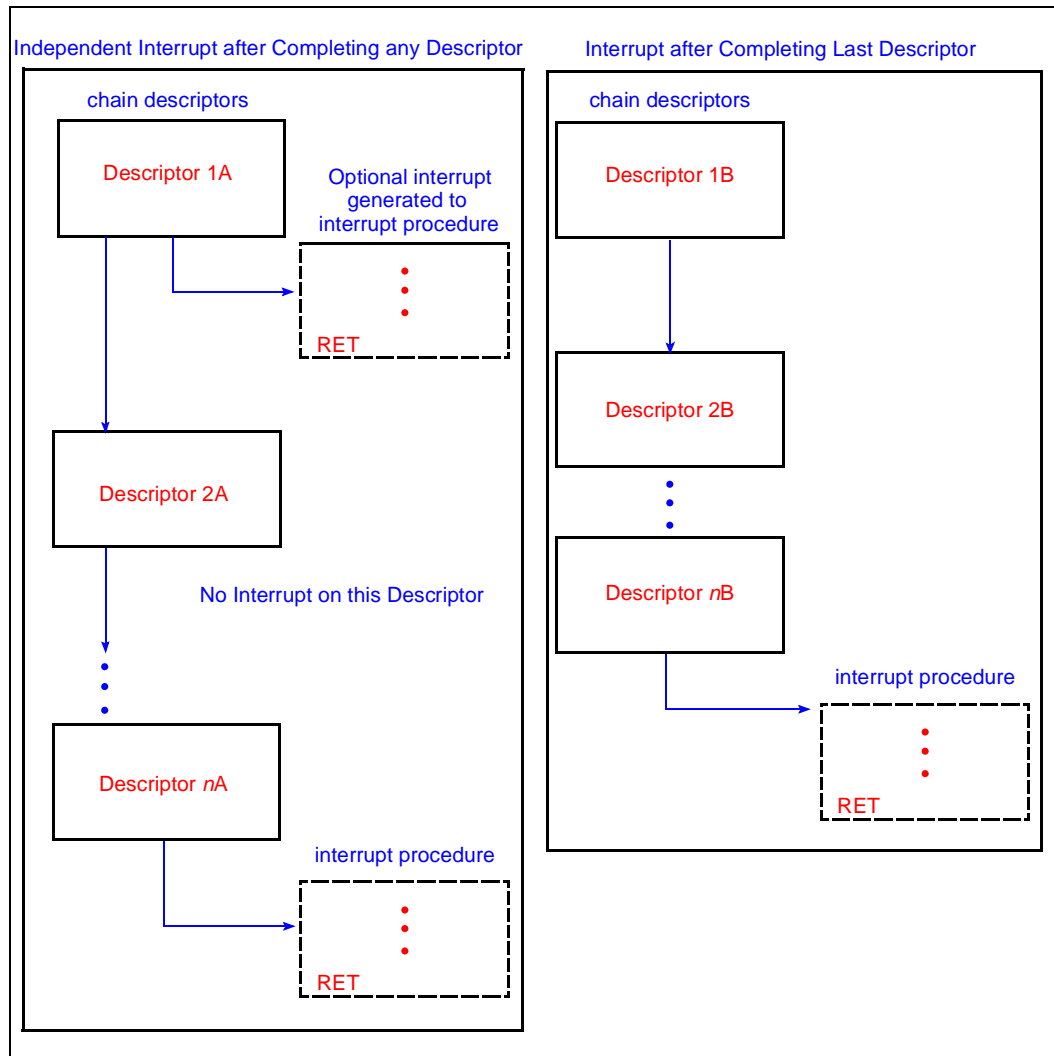
Chained DMA transfers can be synchronized to a program executing on the i960 core processor through the use of processor interrupts. The channel generates an interrupt to the i960 core processor under certain conditions. They are:

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 is generated to the i960 core processor. This interrupt is for synchronization purposes only. The channel sets 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 starts 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 is generated to the i960 core processor. The channel sets 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 halts 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 *nA*, 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 always contains the address of the next chain descriptor to be read and the Descriptor Address Register 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 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 re-reads 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 clears the *Chain Resume* bit and then examine the Next Descriptor Address Register. If the Next Descriptor Address Register is not zero, the channel reads the chain descriptor using this new address and begin a new DMA transfer. If the Next Descriptor Address Register is zero, the channel remains or return to idle.

The three cases to consider are:

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 is 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 specification 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 2.1 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 i960 RM/RN I/O processor 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# is 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 decrements the byte count by 8 for every successful data transfer cycle.

Refer to [Section 14.6.3.1, “64-Bit Protocol”](#) on page 14-27 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# remains 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 decrements the byte count by 4 for every successful data transfer cycle.

Should a slave disconnect on an even word boundary, then all future transfers is 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 14-29 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 14.5.3, “64-Bit Address Decoding - Dual Address Cycles”](#) on page 14-18 for complete details on the 64-bit addressing protocol.

## 19.5 Data Transfers

The i960 RM/RN 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 requests 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 specification 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 requests 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 attempts to use the Memory Write and Invalidate command on the PCI bus whenever programmed by the application software. The DMA channel requests 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 converts 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 performs 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 Packing and Unpacking

Each channel contains a hardware data packing and unpacking unit to support unaligned data transfers between the source and destination buses. The packing unit optimizes data transfers to and from 32 and 64-bit memory. The channel reformats data words for the correct bus data width. When the channel needs to pack or unpack data, the data is held internally to the channel and does not need to be re-read.

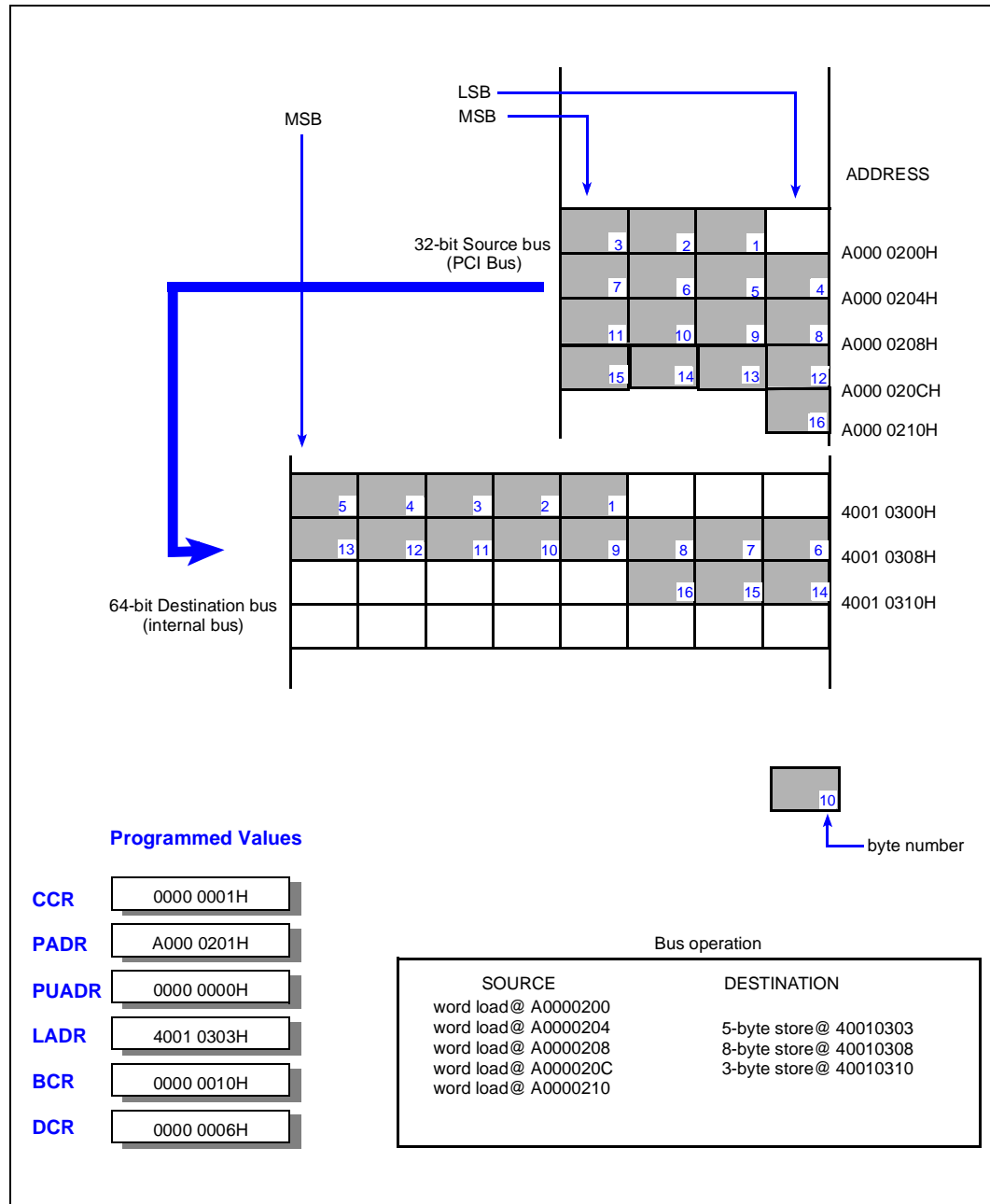
Aligned data transfers involve data accesses that fall on natural boundaries. For example; double words are aligned on 8-byte boundaries and words are aligned on 4-byte boundaries. DMA transfers can occur with both the source and destination addresses unaligned.



## 19.7.2 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



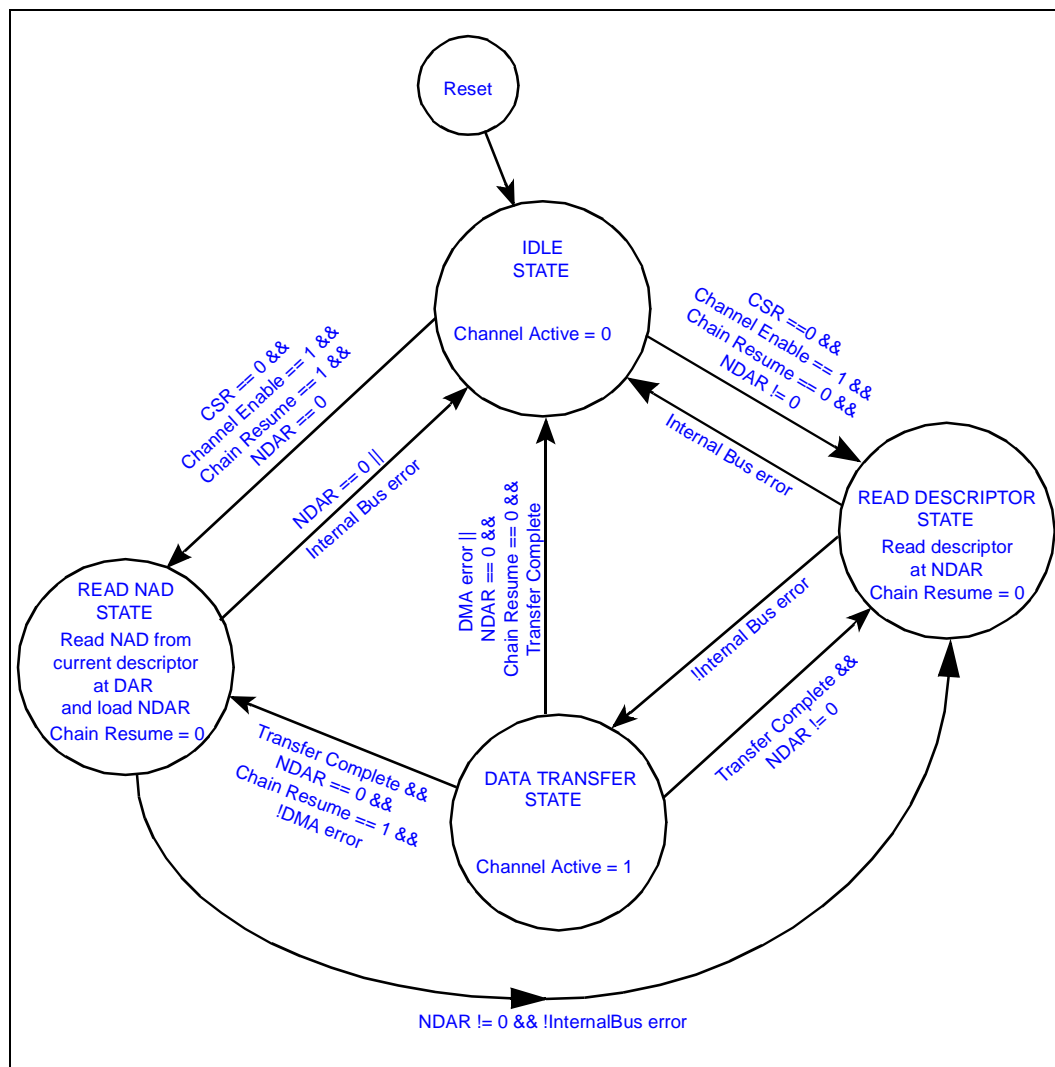
## 19.8 Channel Priority

The i960 RM/RN I/O processor 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, “i960® RM/RN 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 [Example 19-1](#).

#### Example 19-1. 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. [Example 19-2](#) describes the pseudo code for starting a DMA transfer on channel 0.

#### Example 19-2. 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 resumes DMA operation without requiring the software to save the channel configuration. [Example 19-3](#) describes the pseudo code for suspending channel 0.

#### Example 19-3. 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”](#). [Table 19-2](#) 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 is 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 sets bit 3 in the CSR. The channel also reflects the error to the Address Translation Units (PATU or SATU depending on which channel was the master while the error occurred). The ATU in turn, records this error condition by setting the appropriate bit in its status register (PATUSR or SATUSR). Refer to [Chapter 15, “Address Translation Unit”](#) for complete details.
- If a PCI Target-Abort (Master) occurs during a DMA transfer, the channel sets bit 2 in the CSR. The channel also reflects the error to the Address Translation Units (PATU or SATU depending on which channel was the master while the error occurred). The ATU in turn, records this error condition by setting the appropriate bit in its status register (PATUSR or SATUSR). Refer to [Chapter 15, “Address Translation Unit”](#) for complete details.
- If a PCI data parity error occurs during a DMA transfer, the channel sets bit 0 in the CSR. The channel also reflects the error to the Address Translation Units (PATU or SATU depending on which channel was the master while the error occurred). The ATU in turn, records 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, “Address Translation Unit”](#) for complete details.

## 19.12.2 Internal Bus Errors

Internal Bus error conditions and the actions taken, are detailed below:

- If an error occurs during a read of the Chain Descriptor or Next Descriptor Address, the channel sets the appropriate error flag in the CSR, 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 prematurely ends the transaction and stop transferring data as soon as possible.
- If the channel has asserted it's PCI request signal, but not yet started the transaction, the channel deasserts it's request.
- If the channel has not yet asserted a request for the PCI bus, the channel never asserts 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 invalidates any data held in the queue and not read any new chain descriptors.
- The channel sets the appropriate error flag in the Channel Status Register. For example; if a PCI Master-Abort occurred during a DMA transfer, the channel sets bit 3 in the CSR. During an MWI transaction, the channel completes the transfer of the cache line before stopping.
- The channel also signals an interrupt to the i960 core processor.
- The channel does 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 Powerup/Default Status

Upon powerup, an external hardware reset, the DMA Registers is 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**

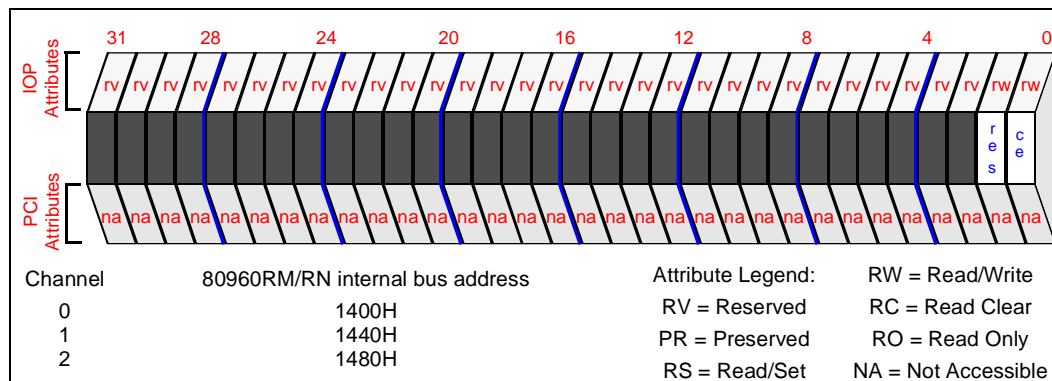
Section, Register Name, Acronym (page)
Section 19.14.1, "Channel Control Register - CCR" on page 19-22
Section 19.14.2, "Channel Status Register - CSR" on page 19-23
Section 19.14.3, "Next Descriptor Address Register - NDAR" on page 19-25
Section 19.14.4, "Descriptor Address Register - DAR" on page 19-26
Section 19.14.5, "Byte Count Register - BCR" on page 19-27
Section 19.14.6, "PCI Address Register - PADR" on page 19-28
Section 19.14.7, "PCI Upper Address Register - PUADR" on page 19-29
Section 19.14.8, "Local Address Register - LADR" on page 19-30
Section 19.14.9, "Descriptor Control Register - DCR" on page 19-31

## 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. Table 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**

Bit	Default	Description
31:02	0000 0000H	Reserved
01	0 <sub>2</sub>	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: <ul style="list-style-type: none"> <li>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.</li> <li>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</li> </ul>
00	0 <sub>2</sub>	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.



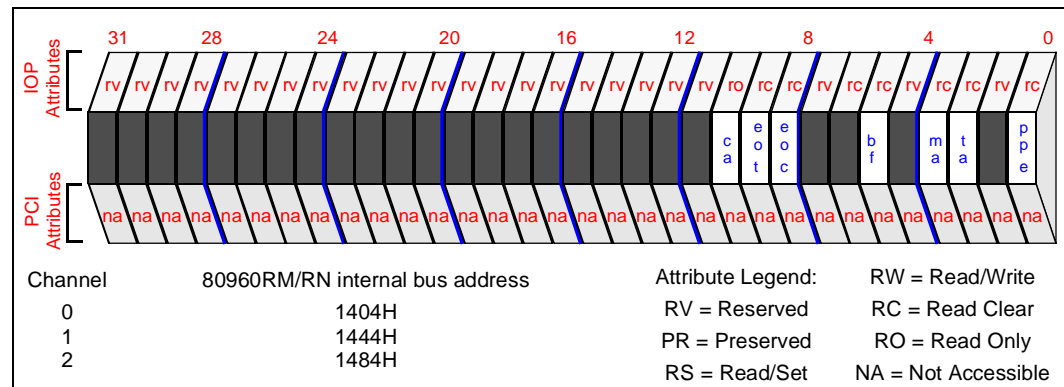
## 19.14.2 Channel Status Register - CSR

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)**

Bit	Default	Description
31:11	000000H	Reserved
10	0 <sub>2</sub>	<p>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:</p> <ul style="list-style-type: none"> <li>byte count reached zero and last chain descriptor is encountered (NULL value detected for Next Descriptor Address in chain descriptor)</li> <li>PCI Master-abort occurred on the PCI interface</li> <li>PCI Target-abort occurred on the PCI interface</li> <li>PCI parity error occurred on the PCI interface</li> <li>Internal Bus Errors</li> </ul> <p>The Channel Active flag is set when a Chain Descriptor is read from memory.</p>
09	0 <sub>2</sub>	End of Transfer Interrupt Flag - set when the channel has signalled an interrupt to the i960 core processor after successfully completing an error-free DMA transfer but it is not the last descriptor in a chain.
08	0 <sub>2</sub>	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 <sub>2</sub>	Reserved
05	0 <sub>2</sub>	All Master-aborts when the channel is the master on the internal bus is reflected by setting this bit.
04	0 <sub>2</sub>	Reserved
03	0 <sub>2</sub>	PCI Master Abort Flag - set when the channel has initiated a transaction on the PCI bus and has detected a Master-abort.





### 19.14.3 Next Descriptor Address Register - NDAR

The Next Descriptor Address Register (NDAR<sub>x</sub>) contains the address of the next chain descriptor in i960 RM/RN I/O processor 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**

Bit	Default	Description
31:05	X	Next Descriptor Address - local memory address of the next chain descriptor to be read by the channel.
04:00	00000 <sub>2</sub>	Reserved

Channel	80960RM/RN internal bus address	Attribute Legend:	RW = Read/Write
0	1410H	RV = Reserved	RC = Read Clear
1	1450H	PR = Preserved	RO = Read Only
2	1490H	RS = Read/Set	NA = Not Accessible

### 19.14.4 Descriptor Address Register - DAR

The Descriptor Address Register (DARx) contains the address of the current chain descriptor in i960 RM/RN I/O processor 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**

IOP Attributes		31	28	24	20	16	12	8	4	0	
		ro	ro	ro	ro	ro	ro	ro	ro	ro	ro
PCI Attributes		na	na	na	na	na	na	na	na	na	
		na	na	na	na	na	na	na	na	na	na
Channel		80960RM/RN internal bus address								Attribute Legend:	RW = Read/Write
0		140CH								RV = Reserved	RC = Read Clear
1		144CH								PR = Preserved	RO = Read Only
2		148CH								RS = Read/Set	NA = Not Accessible
Bit	Default	Description									
31:05	X	Current Descriptor Address - local memory address of the current chain descriptor that was read by the channel.									
04:00	00000 <sub>2</sub>	Reserved									





### 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 drives the P\_AD[1:0] or S\_AD[1:0] to a value of 00<sub>2</sub> 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 does not notify the application of this condition.

**Table 19-9. PCI Address Register - PADR**

Channel		80960RM/RN internal bus address																Attribute Legend:		RW = Read/Write													
0		1414H																RV = Reserved		RC = Read Clear													
1		1454H																PR = Preserved		RO = Read Only													
2		1494H																RS = Read/Set		NA = Not Accessible													
<b>Bit</b>	<b>Default</b>	<b>Description</b>																															
31:00	X	PCI Address - is the PCI source/destination address.																															

### 19.14.7 PCI Upper Address Register - PUADR

The PCI Upper Address Register (PUADR<sub>x</sub>) 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**

IOP Attributes	31	28	24	20	16	12	8	4	0
	ro	ro	ro	ro	ro	ro	ro	ro	ro
PCI Attributes	na	na	na	na	na	na	na	na	na
	na	na	na	na	na	na	na	na	na
Channel	80960RM/RN internal bus address				Attribute Legend:		RW = Read/Write		
0	1418H				RV = Reserved		RC = Read Clear		
1	1458H				PR = Preserved		RO = Read Only		
2	1498H				RS = Read/Set		NA = Not Accessible		
Bit	Default	Description							
31:00	X	PCI Upper Address - is the PCI source/destination upper 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**

Bit	Default	Description
31:06	000000H	Reserved
05	0 <sub>2</sub>	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 (PUADR <sub>x</sub> ) 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 <a href="#">Section 19.4</a> for details on 64-bit addressing.
04	0 <sub>2</sub>	Interrupt Enable - when set, the channel generates an interrupt to the i960 RM/RN I/O processor upon completion of a DMA transfer. When clear, no interrupt is generated.
03:00	0000 <sub>2</sub>	PCI Command - determines PCI bus command type on the PCI bus for this DMA transfer. This value is used directly for the PCI bus command; e.g., when PCI Command is 0000 <sub>2</sub> , the PCI Command is 0000 <sub>2</sub> , a reserved command type. See <a href="#">Table 19-13</a> . Hardware does not check for reserved or unsupported command types.

The following PCI command types are not supported:

- I/O Read
- I/O Write
- Configuration Read
- Configuration Write

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 19-12 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 <sub>2</sub>	reserved	Not Supported
0001 <sub>2</sub>	reserved	Not Supported
0010 <sub>2</sub>	I/O Read	Not Supported
0011 <sub>2</sub>	I/O Write	Not Supported
0100 <sub>2</sub>	reserved	Not Supported
0101 <sub>2</sub>	reserved	Not Supported
0110 <sub>2</sub>	Memory Read	Memory Read of less than one cacheline
0111 <sub>2</sub>	Memory Write	Memory Write
1000 <sub>2</sub>	reserved	Not Supported
1001 <sub>2</sub>	reserved	Not Supported
1010 <sub>2</sub>	Configuration Read	Not Supported
1011 <sub>2</sub>	Configuration Write	Not Supported
1100 <sub>2</sub>	Memory Read Multiple	Memory Read of more than one cacheline
1101 <sub>2</sub>	reserved	Not Supported
1110 <sub>2</sub>	Memory Read Line	Memory Read of one cacheline
1111 <sub>2</sub>	Memory Write and Invalidate	Memory Write which guarantees the transfer of a complete cache line during the current transaction

This chapter describes the integrated Application Accelerator Unit (AAU). The operation modes, setup, external interface, and implementation of the AAU are detailed in this chapter.

## 20.1 Overview

The AAU provides low-latency, high-throughput data transfer capability between the AAU and 80960 local memory. It executes data transfers to and from 80960 local memory and also provides the necessary programming interface. The AAU performs the following functions:

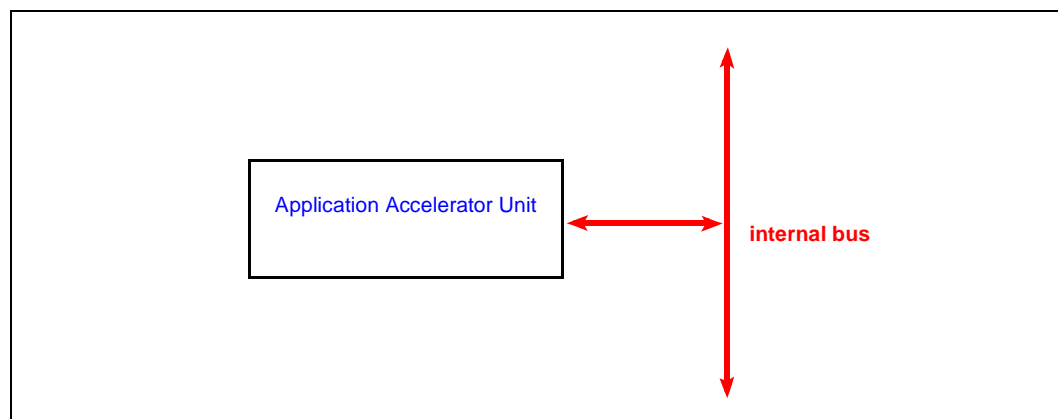
- Transfers data (read) from memory controller
- Performs an optional boolean operation (XOR) on read data
- Transfers data (write) to memory controller

The AAU features:

- 128-byte, arranged as 8-byte x 16-deep store queue
- Utilization of the i960® RM/RN I/O processor memory controller Interface
- $2^{32}$  addressing range on the 80960 local memory interface
- Hardware support for unaligned data transfers for the internal bus
- Fully programmable from the i960 core processor
- Support for automatic data chaining for gathering and scattering of data blocks

Figure 20-1 shows a simplified connection of the AAU to the i960 RM/RN I/O Processor Internal Bus.

**Figure 20-1. Application Accelerator Unit**



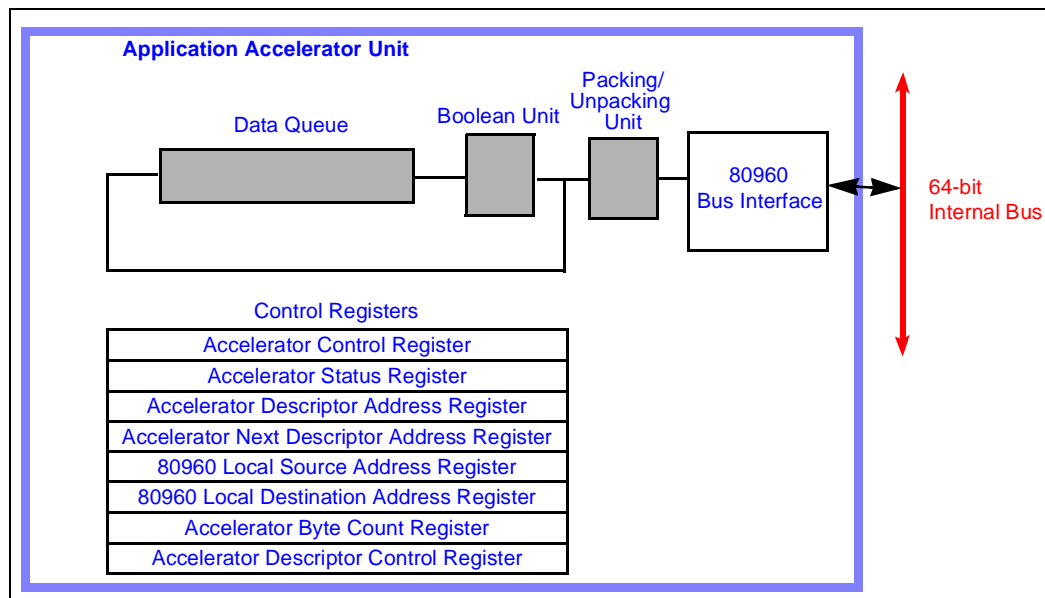
## 20.2 Theory of Operation

The AAU 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 AAU 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 i960 RM/RN I/O processor memory controller unit.

**Figure 20-2. Application Accelerator Unit Block Diagram**



The AAU 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 \(4 Source Addresses\)”](#) on page 20-3.

The AAU 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 AAU until ready to be stored back to local memory. This is done using a 128-byte holding queue (arranged as 8-byte x 16-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 AAU 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 AAU 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 hereafter is referred to as *XOR-transfer*.
- The process of reading or writing data hereafter is 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 (4 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.

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 ignores 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 is 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 is 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 is 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 is driven on the internal bus. This value is loaded into the Source Address Register 4.
- The sixth word is the destination address where data is stored in 80960 local memory. This address is 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 AAU 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 20.5, on page 20-15](#).

Refer to [Section 20.12](#) for additional description on the control registers.

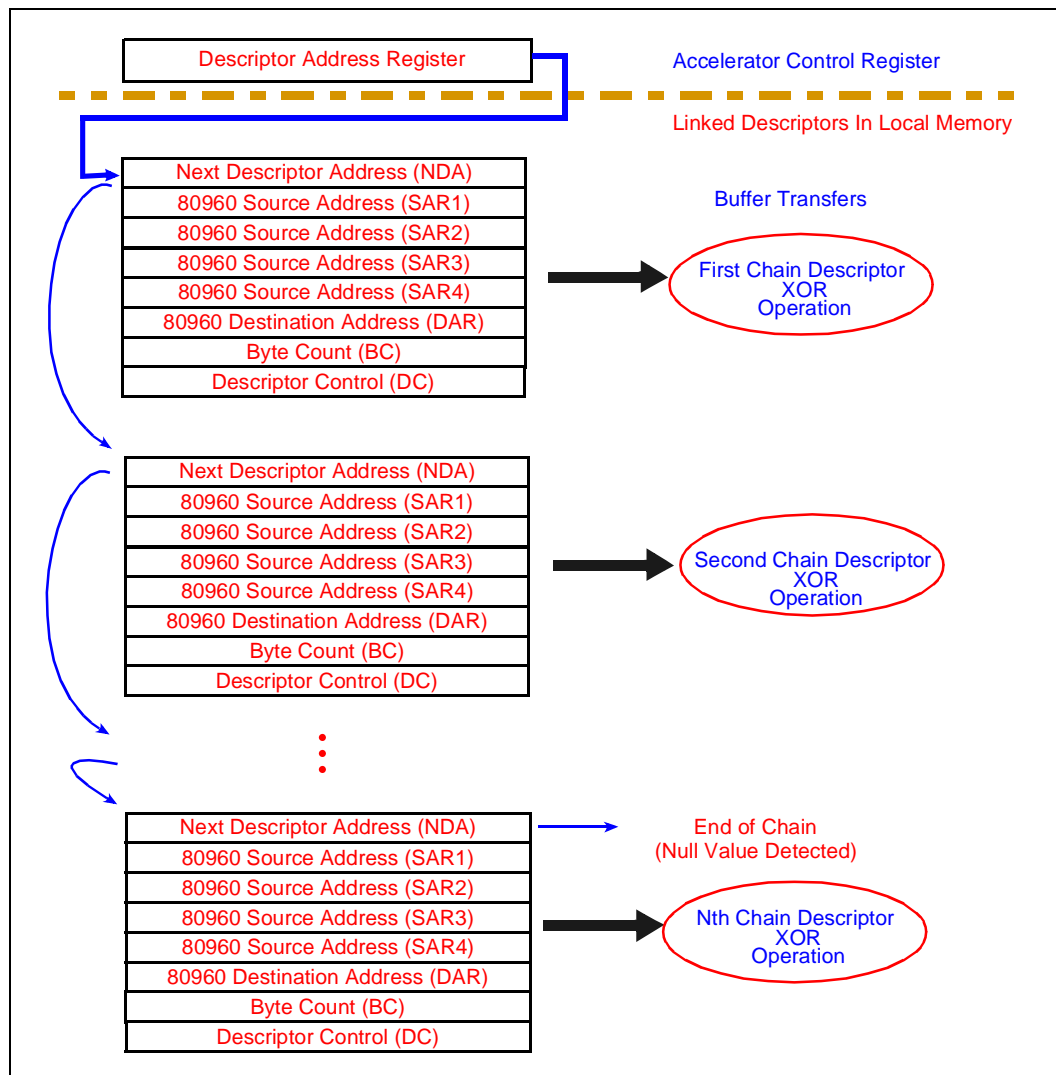
**Figure 20-3. Chain Descriptor Format**

Chain Descriptor in 80960 Memory	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 Destination Address (DAR)	Destination Address
Byte Count (BC)	Number of bytes
Descriptor Control (DC)	Descriptor Control

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 four source blocks of data (up to eight), a special chain descriptor needs to be configured:

- The first part (*principal-descriptor*) contains the address of the first four source data blocks along with other information.
- The second part (*mini-descriptor*) contains four, 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.

- 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 ignores 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 is 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 is 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 is 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 is driven on the internal bus. This value is loaded into SAR4.
- The sixth word is the destination address where the XOR-ed data is stored in 80960 local memory. This address is 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 AAU 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 is 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 is 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 is driven on the internal bus. This value is loaded into 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 is driven on the internal bus. This value is loaded into SAR8.

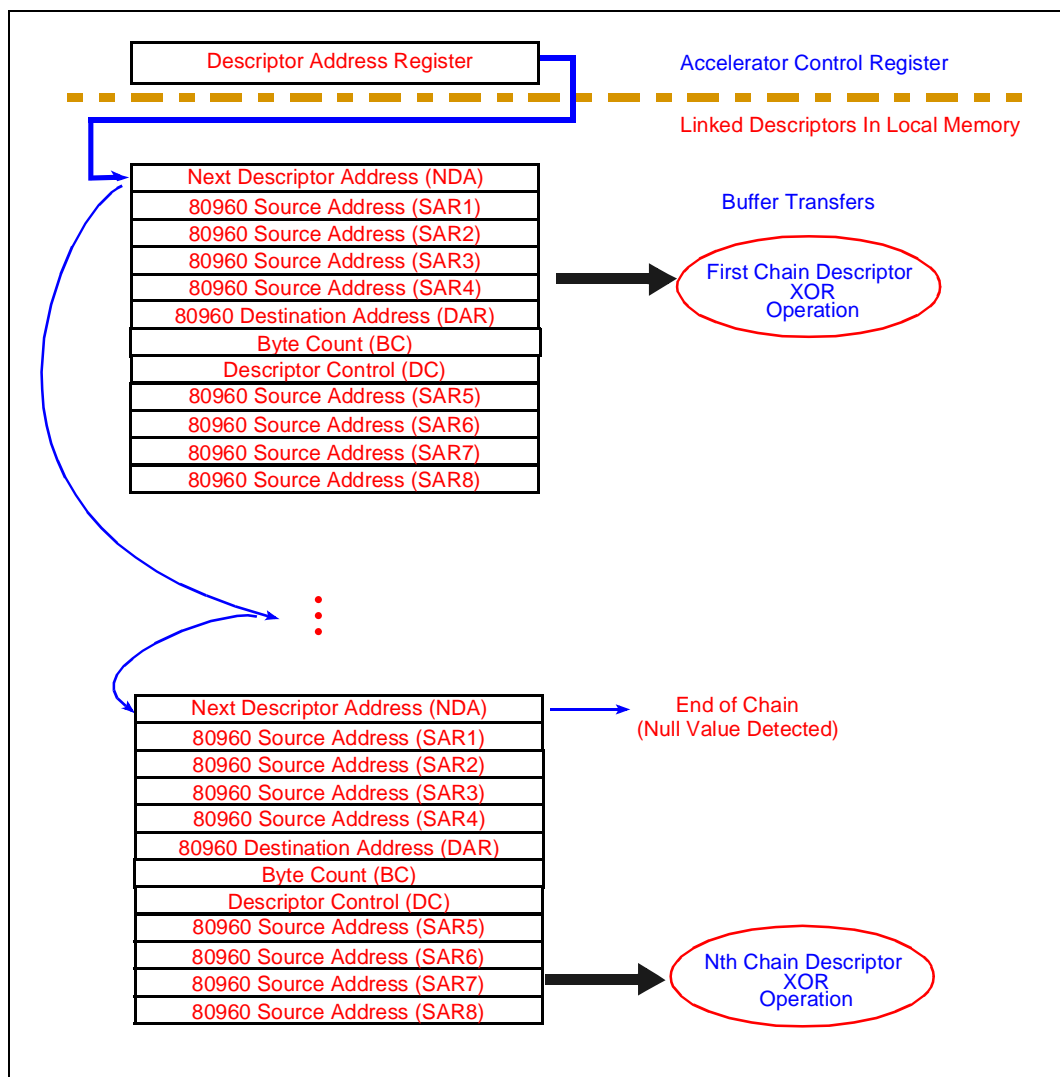
**Figure 20-5. Chain Descriptor Format for 8 Source Addresses (XOR Function)**

Chain Descriptor in 80960 Memory	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 Destination 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

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 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

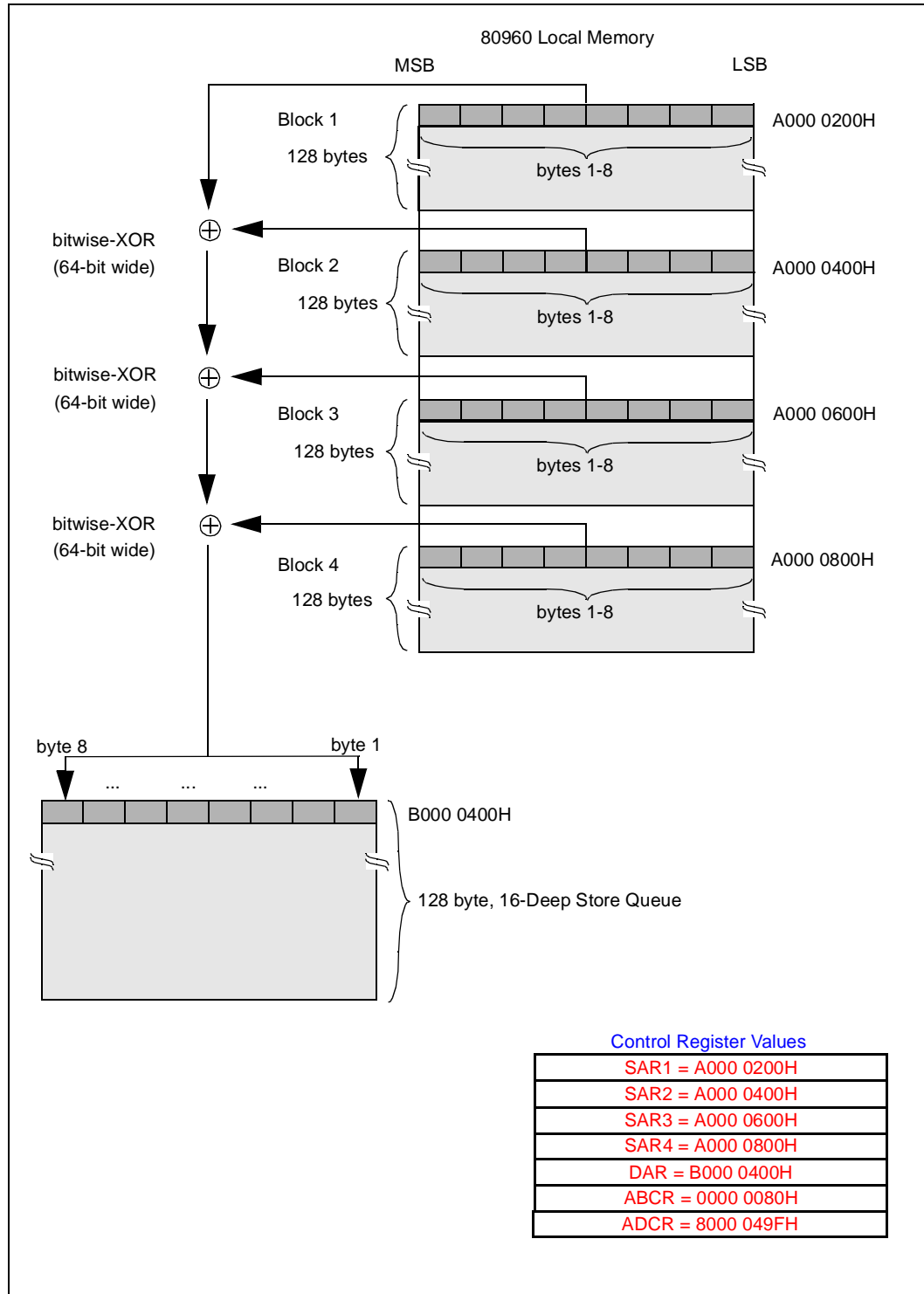


### 20.3.4 The Bitwise-XOR Algorithm

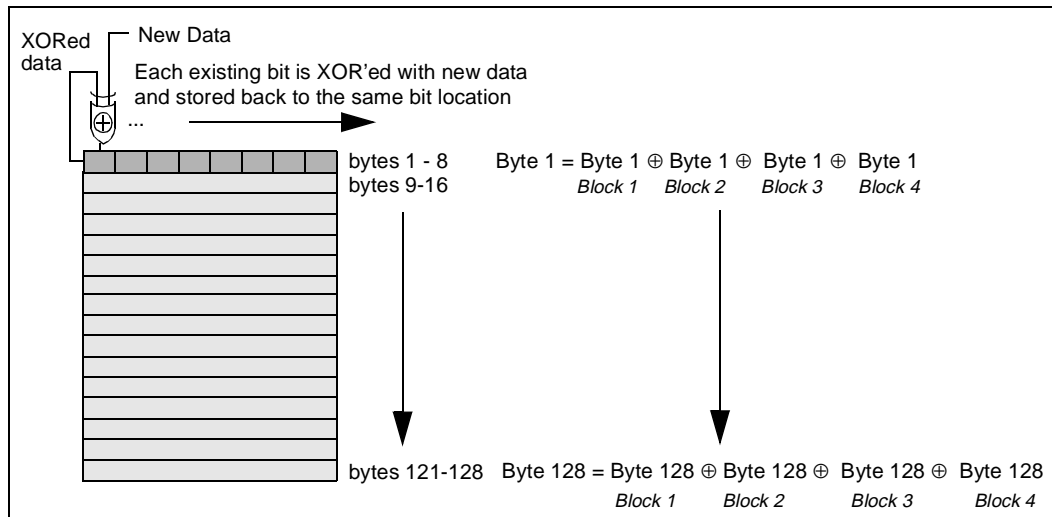
Figure 20-7 describes the XOR algorithm implementation. In this illustrative example, there are four blocks of source data to be XOR-ed. The intermediate result is stored in the 128-byte store queue in the AAU before being written back to 80960 local memory. The source data is located at addresses A000 0200H, A000 0400H, A000 0600H and A000 0800H respectively.

All data transfers needed for this operation are controlled by chain descriptors located in local memory. The AAU 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 AAU 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 128. The total number of bytes to *XOR-transfer* is specified by the Byte Count (BC) field in the chain descriptor.

**Note:** The AAU operates on 128 bytes of data at a time. If the Byte Count Register contains a value greater than 128, the AAU completes the *XOR-transfer* operation on the first 128 bytes of data obtained from each Source Register (SAR1 - SAR4), then proceeds with the next 128 bytes of data. This process is repeated until the BCR contains a zero value.

2. The AAU 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 AAU 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 128 bytes 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 128 bytes 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 128 and the ADCR.dwe bit is cleared, the AAU only reads the first 128 bytes and perform the specified function. It does not read the remaining bytes specified in the ABCR. Further, the AAU proceeds 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.

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 now contains the Next Descriptor Address from the chain descriptor just read.

The last descriptor in the XOR chain list has 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.

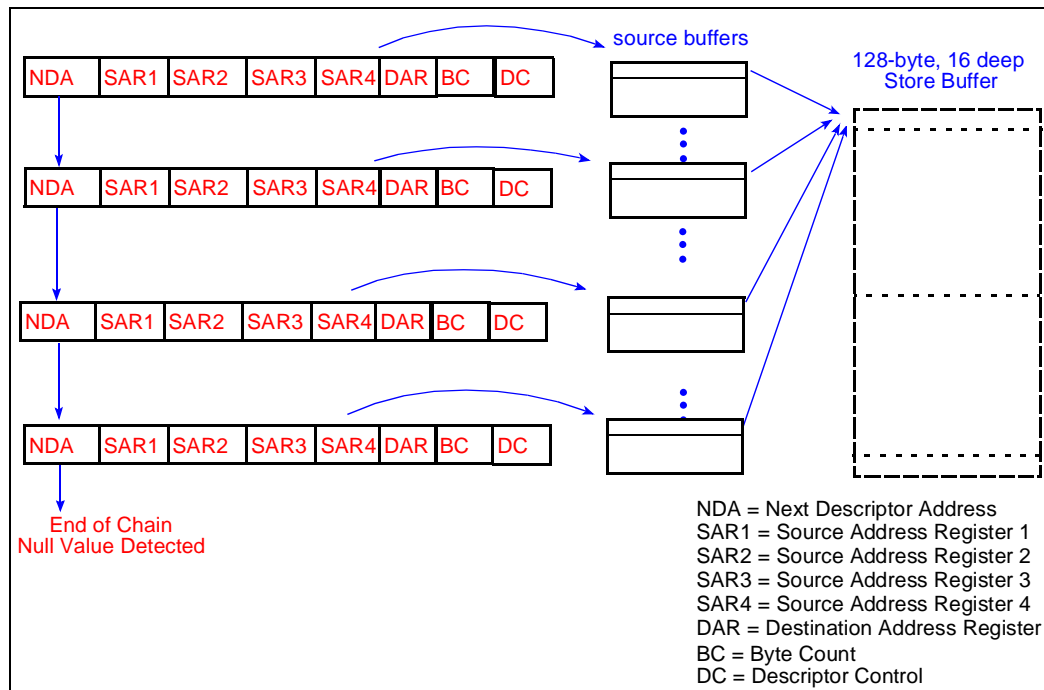
Once an XOR operation is active, it can be temporarily suspended by clearing the *Accelerator Enable* bit in the ACR. Note that this does not abort the XOR operation. The unit resumes the process when the *Accelerator Enable* bit is set.

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 AAU 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 specifies the destination location such that once all data has been processed, the data is contiguous in memory. Figure 20-9 shows how the destination pointers can gather data.

Figure 20-9. Example of Gather Chaining for Four Source Blocks



## 20.3.7 Synchronizing a program to Chained operation

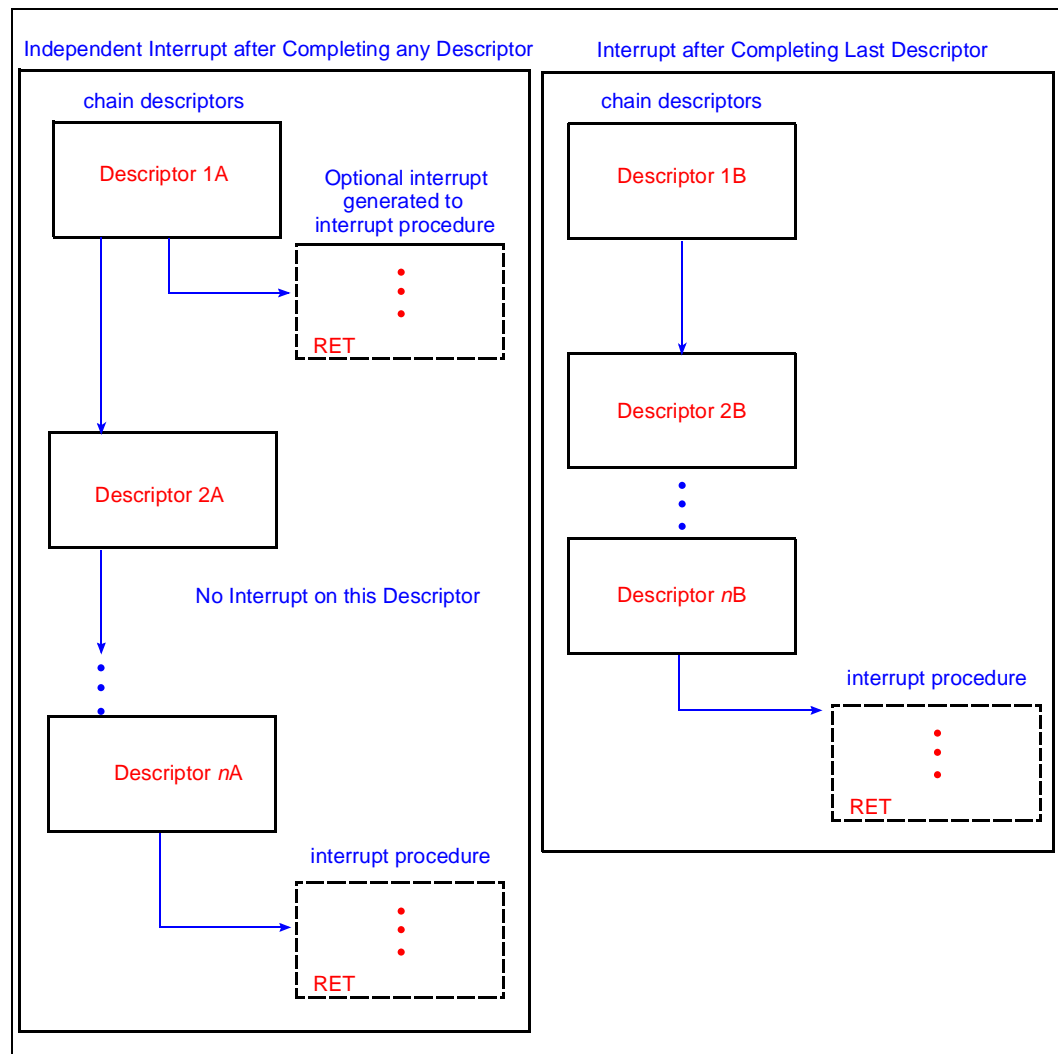
Any operation involving the AAU can be synchronized to a program executing on the i960 core processor through the use of processor interrupts. The AAU generates an interrupt to the i960 core processor under certain conditions. They are:

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 is generated to the i960 core processor. This interrupt is for synchronization purposes. The AAU sets 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 starts 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 is generated to the i960 core processor. The AAU sets the *End Of Chain Interrupt* flag in the ASR.
3. [Error] An error condition occurs (refer to [Section 20.10, “Error Conditions”](#) on page 20-19 for AAU error conditions) during a transfer. The AAU halts 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 *nA*, 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 re-reads 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 reads the chain descriptor using this new address and begin a new operation. If ANDAR is zero, the AAU remains 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 is 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 Store Queue

The AAU contains one 128-byte store queue. This is arranged as an 8-byte x 16-deep queue. The queue is used to hold data. Depending on the configuration of the Accelerator Descriptor Control Register (ADCR), data in the queue is either written to 80960 local memory or is held in the queue.

When a transaction defined by a Chain Descriptor completes in its entirety, the contents of the queue are undefined. The storage queue should therefore not be used as a history buffer when setting up new transactions.



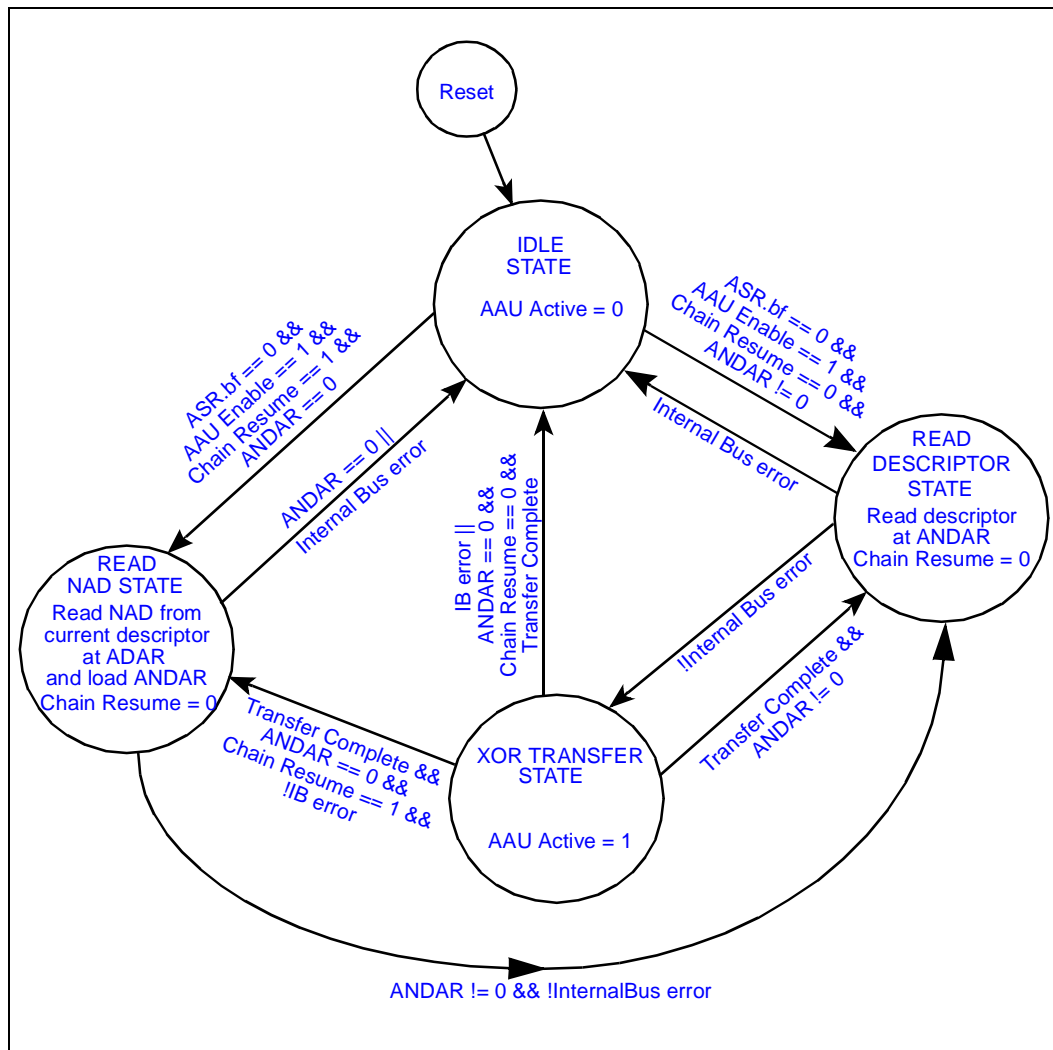
## 20.6 Application Accelerator Unit Priority

The internal bus arbitration logic determines which internal bus master has access to the i960 RM/RN I/O Processor Internal Bus. The AAU has an independent Bus Request/Grant signal pair to the internal bus arbitration logic. Chapter 17, “i960® RM/RN I/O Processor Arbitration” describes in detail the priority scheme between all of the bus masters on the internal bus.

## 20.7 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 Unit Programming Model State Diagram



## 20.8 Programming the Application Accelerator Unit

The software for initiating an *XOR-transfer* using the AAU 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.

### 20.8.1 Application Accelerator Unit 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 [Example 20-1](#).

#### Example 20-1. Pseudo Code: AAU Initialization

```
ACR = 0x0000 0000 ; Disable the application accelerator
Call setup_accelerator
```

### 20.8.2 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. [Example 20-2](#) describes the pseudo code for initiating an XOR operation with the AAU.

#### Example 20-2. Pseudo Code: XOR Transfer Operation for Four Source Addresses and One Destination Address

```
; Set up descriptor in 80960 local memory at address d
d.nda = 0          /* No chaining */
d.SAR1 = 0xA000 0200/* Source address of Data Block 1      */
d.SAR2 = 0xA000 0400/* Source address of Data Block 2      */
d.SAR3 = 0xA000 0600/* Source address of Data Block 3      */
d.SAR4 = 0xA000 0800/* Source address of Data Block 4      */
d.DAR = 0xB000 0100/* Destination address of XOR-ed data */
d.ABCR = 128      /* Byte Count of 128 */
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 AA & no pending interrupts
if (ASR != 0) exit /* If AA is not ready, exit */

; Start Operation
ANDAR = &d          ; Set up descriptor address
ACR = 0x00000001   ; Set AA Enable bit
```

## 20.8.3 Suspend Application Accelerator Unit

The AAU provides the ability to suspend the current state without losing status information. The AAU resumes without requiring application software to save the current configuration. [Example 20-3](#) describes pseudo-code for suspending the ongoing operation and then restarting.

### Example 20-3. Pseudo Code: Suspend Application Accelerator Unit

```
ACR = 0x0000 0000 ; Suspend ongoing AA transfer

;Suspend Application Accelerator
ACR = 0x0000 0001 ; Resume AA transfer
```

## 20.9 Interrupts

The AAU 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 20.10](#) 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”](#). [Table 20-2](#) summarizes the status flags and conditions when interrupts are generated in the Accelerator Status Register (ASR).

**Table 20-2. Application Accelerator Unit Interrupt Summary**

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
Byte count == 0 && ANDAR != NULL (End of Transfer)	1	1	0	0	Y	N
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	0	0

**Note:** End-of-Transfer and End-of-Chain flags is 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 transfer completes 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.



## 20.9.1 Interrupts - Special Case (ADCR.dwe = 0)

Once the AAU is enabled, the AAU loads the chain descriptor fields into the respective registers. If ADCR.dwe = 0, then an interrupt is generated (if enabled) after the descriptor is fetched and processed as defined by the block control fields in the ADCR. Table 20-3 summarizes the status flags and conditions when interrupts are generated in the Accelerator Status Register (ASR) for this special case.

**Table 20-3. AAU Interrupts - Special Case**

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	0	0

**Note:** End-of-Transfer and End-of-Chain flags is 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.

## 20.10 Error Conditions

Master Aborts that occur during a transfer are recorded by the AAU.

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 does not read any new chain descriptors.
- The AAU sets the error flag in the Accelerator Status Register. For example; if an IB master-abort occurred during a transfer, the channel sets bit 5 in the ASR.
- The AAU signals an interrupt to the i960 core processor.
- The AAU does 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 error conditions are handled as detailed below:

- **Multi-bit ECC error on MCU Data Read:** Refer [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.

## 20.11 Powerup/Default Status

Upon powerup, an external hardware reset, the AAU Registers is initialized to their default values.

## 20.12 Register Definitions

The AAU 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 20-4. Application Accelerator Unit Registers**

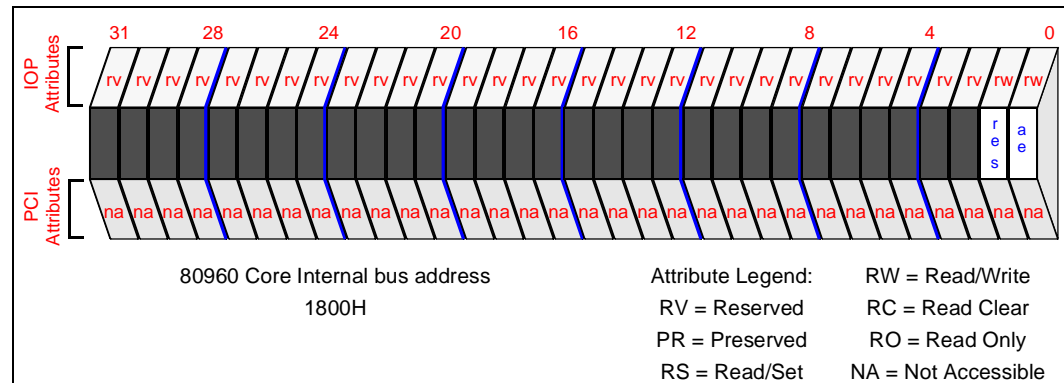
Section, Register Name - Acronym (page)
Section 20.12.1, “Accelerator Control Register - ACR” on page 20-21
Section 20.12.2, “Accelerator Status Register - ASR” on page 20-22
Section 20.12.3, “Accelerator Descriptor Address Register - ADAR” on page 20-23
Section 20.12.4, “Accelerator Next Descriptor Address Register - ANDAR” on page 20-24
Section 20.12.5, “80960 Source Address Register - SAR” on page 20-25
Section 20.12.6, “80960 Destination Address Register - DAR” on page 20-26
Section 20.12.7, “Accelerator Byte Count Register - ABCR” on page 20-27
Section 20.12.8, “Accelerator Descriptor Control Register - ADCR” on page 20-28

## 20.12.1 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 20-5 shows the register format. This register can be read or written while the AAU is active.

**Table 20-5. Accelerator Control Register - ACR**

Bit	Default	Description
31:02	0	Reserved
01	0 <sub>2</sub>	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: <ul style="list-style-type: none"> <li>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.</li> <li>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</li> </ul>
00	0 <sub>2</sub>	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.



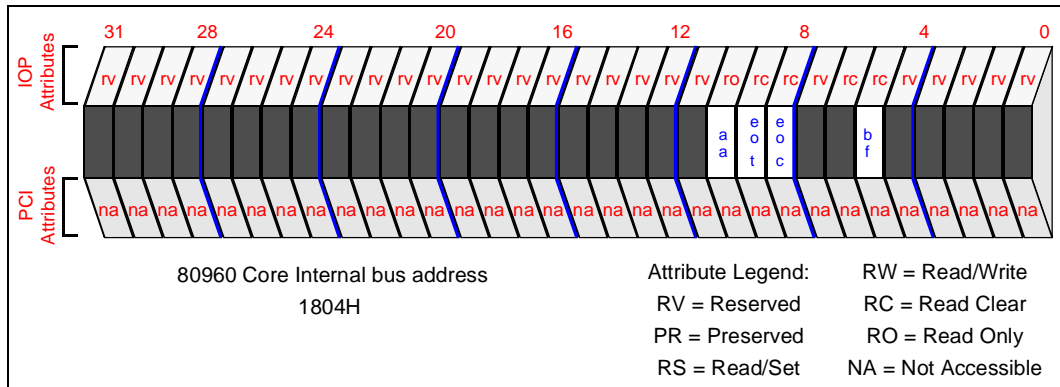
## 20.12.2 Accelerator Status Register - ASR

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 20.10](#) for a description of the error conditions that are reported in the ASR. See [Section 20.9](#) for a description of interrupts caused by the AAU.

If an AAU error occurs, application software should check the status of Accelerator Active flag before processing the interrupt.

**Table 20-6. Accelerator Status Register - ASR**

Bit	Default	Description
31:11	000000H	Reserved
10	0 <sub>2</sub>	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: <ul style="list-style-type: none"> <li>byte count reached zero and last chain descriptor is encountered (NULL value detected for Next Descriptor Address in chain descriptor)</li> <li>Internal Bus Errors</li> <li>Last chain descriptor is processed (NULL value detected for Next Descriptor Address in chain descriptor) and ADCR.dwe = 0.</li> </ul> The Accelerator Active flag is set once a Chain Descriptor is read from memory.
09	0 <sub>2</sub>	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 <sub>2</sub>	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 <sub>2</sub>	Reserved
05	0 <sub>2</sub>	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 <sub>2</sub>	Reserved



### 20.12.3 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 20-7 depicts the Accelerator Descriptor Address Register. All chain descriptors are aligned on an eight, 32-bit word boundary.

**Table 20-7. Accelerator Descriptor Address Register - ADAR**

		31	28	24	20	16	12	8	4	0	
IOP Attributes	[	ro	ro	ro	ro	ro	ro	ro	ro	ro	
		ro	ro	ro	ro	ro	ro	ro	ro	ro	ro
PCI Attributes	[	na	na	na	na	na	na	na	na	na	
		na	na	na	na	na	na	na	na	na	na
		80960 Core Internal bus address 1808H						Attribute Legend:		RW = Read/Write	
								RV = Reserved		RC = Read Clear	
								PR = Preserved		RO = Read Only	
								RS = Read/Set		NA = Not Accessible	
Bit	Default	Description									
31:05	X	Current Descriptor Address - local memory address of the current chain descriptor read by the AAU.									
04:00	00000 <sub>2</sub>	Reserved									

## 20.12.4 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 20-8 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 20-8. Accelerator Next Descriptor Address Register - ANDAR**

Bit	Default	Description	
31:05	X	Next Descriptor Address - local memory address of the next chain descriptor to be read by the AAU.	
04:00	00000 <sub>2</sub>	Reserved	

## 20.12.5 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 AAU. The ADCR register (Table 20-12 on page 20-28) 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 returns 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 128-byte data blocks, the value in register SAR1 is the equal to the programmed descriptor value + 384 (SAR1 + 384).

Table 20-9 shows the 80960 Source Address Register. These read-only registers are loaded when a chain descriptor is read from memory.

**Table 20-9. 80960 Source Address Register - SARx**

Bit	Default	Description
31:00	X	80960 Local Address - The 80960 local source address.

80960 Core Internal bus address	Attribute Legend:
SAR1 1810H	RW = Read/Write
SAR2 1814H	RV = Reserved
SAR3 1818H	RC = Read Clear
SAR4 181CH	PR = Preserved
SAR5 182CH	RO = Read Only
SAR6 1830H	RS = Read/Set
SAR7 1834H	NA = Not Accessible
SAR8 1838H	





## 20.12.7 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 i960 RM/RN I/O Processor Internal Bus.

Anytime this register is read by the i960 core processor, it contains the number of bytes left to *XOR-transfer* on the i960 RM/RN I/O Processor Internal Bus. Note that valid data may be present in the AAU store queue. This register is decremented by 1 through 8 for every successful transfer from the store queue to the destination location. [Table 20-11](#) shows the Accelerator Byte Count Register.

**Table 20-11. Accelerator Byte Count Register - ABCR**

IOP Attributes	31	28	24	20	16	12	8	4	0
	rv	rv	rv	rv	rv	rv	rv	rv	rv
PCI Attributes	na	na	na	na	na	na	na	na	na
	na	na	na	na	na	na	na	na	na
80960 Core Internal bus address 1824H									
Attribute Legend: RW = Read/Write RV = Reserved RC = Read Clear PR = Preserved RO = Read Only RS = Read/Set NA = Not Accessible									
Bit	Default	Description							
31:24	00H	Reserved							
23:00	X	Byte Count - is the number of bytes to transfer for an <i>XOR-transfer</i> operation.							

**Note:** 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).

## 20.12.8 Accelerator Descriptor Control Register - ADCR

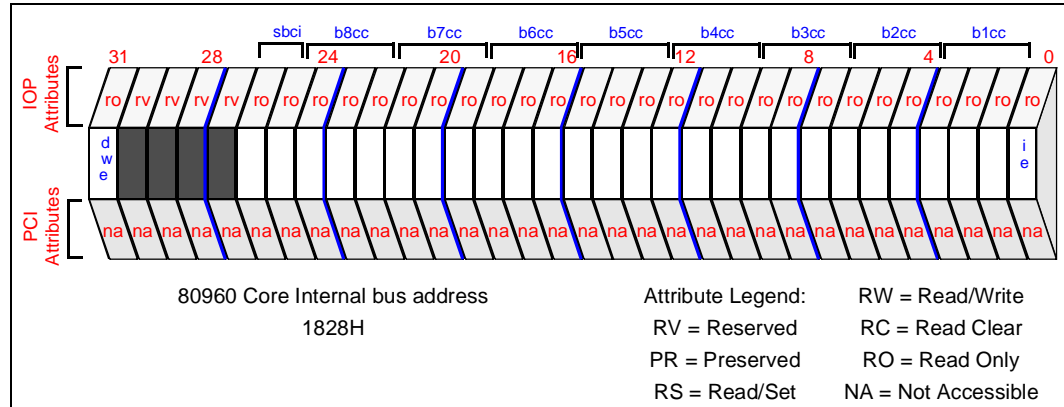
The Accelerator Descriptor Control Register contains control values for data transfer on a per-chain descriptor basis. This read-only register is loaded when a chain descriptor is read from memory. These values may vary from chain descriptor to chain descriptor. The AAU determines whether a mini-descriptor is appended to the end of the current chain descriptor by examining bits 26:25. Table 20-12 shows the definition of the Accelerator Descriptor Control Register.

Table 20-12. Accelerator Descriptor Control Register - ADCR (Sheet 1 of 5)

Bit	Default	Description
31	0 <sub>2</sub>	<p>Destination Write Enable - Determines whether data present in the 128-byte store queue is written out to 80960 local memory. When set, data in the queue is 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 is held in the queue.</p> <p><b>NOTE:</b> If the ABCR register contains a value greater than 128 and this bit is cleared, the AAU only reads the first 128 bytes and perform the specified function. It does not read the remaining bytes specified in the ABCR. Further, the AAU proceeds to process the next chain descriptor if it is specified.</p>
30:27	0H	Reserved
26:25	00	<p>Supplemental Block Control Interpreter - This bit field specifies the number of data blocks on which the <i>XOR-transfer</i> operation is executed.</p> <p>00 0 Blocks - This specifies that no additional data blocks exist. The AAU does not read the mini-descriptor to initialize registers SAR5 - SAR8.</p> <p>01 4 Blocks - This specifies that there are up to 4 additional data blocks. The AAU therefore reads the mini-descriptor to initialize registers SAR5 - SAR8.</p> <p>10 Reserved</p> <p>11 Reserved</p>

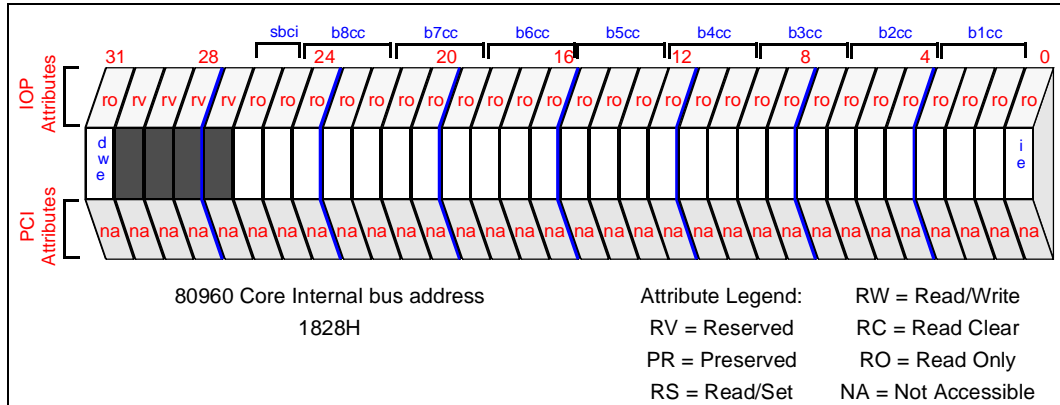
**Table 20-12. Accelerator Descriptor Control Register - ADCR (Sheet 2 of 5)**

Bit	Default	Description
24:22	0	<p><b>Block 8 Command Control</b> - This bit field specifies the type of operation to be carried out on the data pointed at by SAR8 register.</p> <p>000 Null command - This implies that Block 8 Data can be disregarded for the current chain descriptor. The AAU does not transfer data from this block while processing the current chain descriptor.</p> <p>001 XOR command - This implies that Block 8 Data is transferred to the AAU to execute the XOR function.</p> <p>010 Reserved</p> <p>011 Reserved</p> <p>100 Reserved</p> <p>101 Reserved</p> <p>110 Reserved</p> <p>111 Reserved</p>
21:19	0	<p><b>Block 7 Command Control</b> - This bit field specifies the type of operation to be carried out on the data pointed at by SAR7 register.</p> <p>000 Null command - This implies that Block 7 Data can be disregarded for the current chain descriptor. The AAU does not transfer data from this block while processing the current chain descriptor.</p> <p>001 XOR command - This implies that Block 7 Data is transferred to the AAU to execute the XOR function.</p> <p>010 Reserved</p> <p>011 Reserved</p> <p>100 Reserved</p> <p>101 Reserved</p> <p>110 Reserved</p> <p>111 Reserved</p>



**Table 20-12. Accelerator Descriptor Control Register - ADCR (Sheet 3 of 5)**

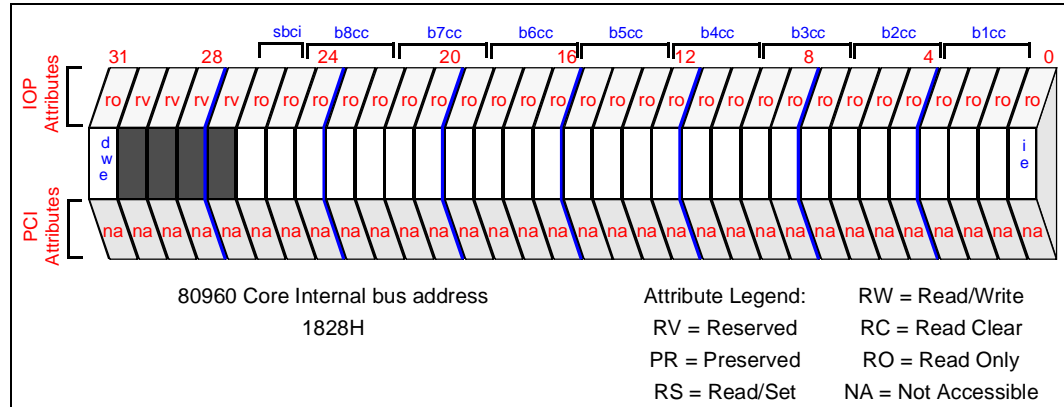
Bit	Default	Description
18:16	0	<p><b>Block 6 Command Control</b> - This bit field specifies the type of operation to be carried out on the data pointed at by SAR6 register.</p> <p>000 Null command - This implies that Block 6 Data can be disregarded for the current chain descriptor. The AAU does not transfer data from this block while processing the current chain descriptor.</p> <p>001 XOR command - This implies that Block 6 Data is transferred to the AAU to execute the XOR function.</p> <p>010 Reserved</p> <p>011 Reserved</p> <p>100 Reserved</p> <p>101 Reserved</p> <p>110 Reserved</p> <p>111 Reserved</p>
15:13	0	<p><b>Block 5 Command Control</b> - This bit field specifies the type of operation to be carried out on the data pointed at by SAR5 register.</p> <p>000 Null command - This implies that Block 5 Data can be disregarded for the current chain descriptor. The AAU does not transfer data from this block while processing the current chain descriptor.</p> <p>001 XOR command - This implies that Block 5 Data is transferred to the AAU to execute the XOR function.</p> <p>010 Reserved</p> <p>011 Reserved</p> <p>100 Reserved</p> <p>101 Reserved</p> <p>110 Reserved</p> <p>111 Reserved</p>



Attribute Legend: RW = Read/Write  
 RV = Reserved RC = Read Clear  
 PR = Preserved RO = Read Only  
 RS = Read/Set NA = Not Accessible

**Table 20-12. Accelerator Descriptor Control Register - ADCR (Sheet 4 of 5)**

Bit	Default	Description
12:10	0	<p><b>Block 4 Command Control</b> - This bit field specifies the type of operation to be carried out on the data pointed at by SAR4 register.</p> <p>000 Null command - This implies that Block 4 Data can be disregarded for the current chain descriptor. The AAU does not transfer data from this block while processing the current chain descriptor.</p> <p>001 XOR command - This implies that Block 4 Data is transferred to the AAU to execute the XOR function.</p> <p>010 Reserved</p> <p>011 Reserved</p> <p>100 Reserved</p> <p>101 Reserved</p> <p>110 Reserved</p> <p>111 Reserved</p>
09:07	0	<p><b>Block 3 Command Control</b> - This bit field specifies the type of operation to be carried out on the data pointed at by SAR3 register.</p> <p>000 Null command - This implies that Block 3 Data can be disregarded for the current chain descriptor. The AAU does not transfer data from this block while processing the current chain descriptor.</p> <p>001 XOR command - This implies that Block 3 Data is transferred to the AAU to execute the XOR function.</p> <p>010 Reserved</p> <p>011 Reserved</p> <p>100 Reserved</p> <p>101 Reserved</p> <p>110 Reserved</p> <p>111 Reserved</p>

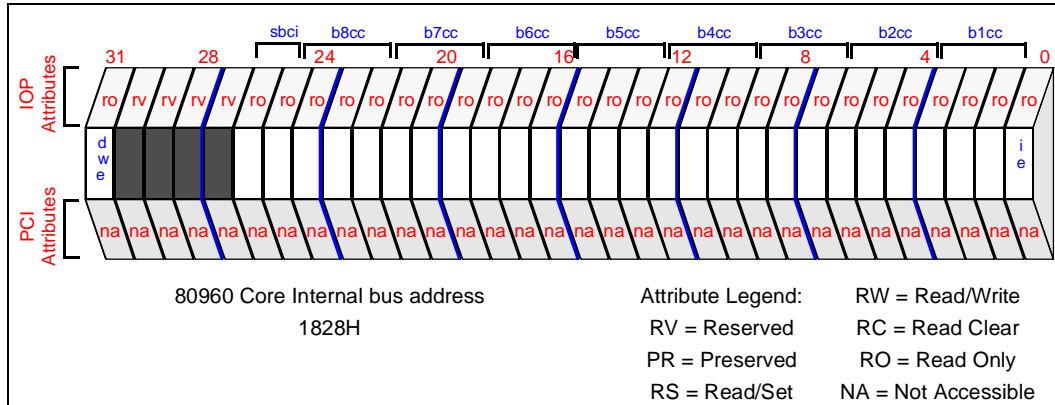


80960 Core Internal bus address  
1828H

Attribute Legend:  
RW = Read/Write  
RV = Reserved  
RC = Read Clear  
PR = Preserved  
RO = Read Only  
RS = Read/Set  
NA = Not Accessible

**Table 20-12. Accelerator Descriptor Control Register - ADCR (Sheet 5 of 5)**

Bit	Default	Description
06:04	0	<p><b>Block 2 Command Control</b> - This bit field specifies the type of operation to be carried out on the data pointed at by SAR2 register.</p> <p>000 Null command - This implies that Block 2 Data can be disregarded for the current chain descriptor. The AAU does not transfer data from this block while processing the current chain descriptor.</p> <p>001 XOR command - This implies that Block 2 Data is transferred to the AAU to execute the XOR function.</p> <p>010 Reserved</p> <p>011 Reserved</p> <p>100 Reserved</p> <p>101 Reserved</p> <p>110 Reserved</p> <p>111 Reserved</p>
03:01	0	<p><b>Block 1 Command Control</b> - This bit field specifies the type of operation to be carried out on the data pointed at by SAR1 register.</p> <p>000 Null command - This implies that Block 1 Data can be disregarded for the current chain descriptor. The AAU does not transfer data from this block while processing the current chain descriptor.</p> <p>001 XOR command - This implies that Block 1 Data is transferred to the AAU to execute the XOR function.</p> <p>010 Reserved</p> <p>011 Reserved</p> <p>100 Reserved</p> <p>101 Reserved</p> <p>110 Reserved</p> <p>111 Direct Fill - This implies that Block 1 Data is transferred directly from 80960 local memory to the 128-byte store queue. In this instance, the data bypasses the Boolean Execution Unit within the AAU.</p>
00	0	<p><b>Interrupt Enable</b> - When set, the AAU generates an interrupt to the i960 RM/RN I/O processor upon completion of a transfer. When clear, no interrupt is generated.</p>



This chapter describes the Performance Monitoring features integrated on the i960® RM/RN 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 i960 RM/RN 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 i960 RM/RN 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 i960 RM/RN 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 sets 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 i960 RM/RN 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 (SATU)
	Number of grants to DMA Ch-2
	Number of grants to the i960 RM/RN 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 the i960 RM/RN I/O processor
	Number of grants to DMA Ch-0 and Ch-1
i960 RM/RN processor internal bus	Number of grants to 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 i960 RM/RN 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 i960 RM/RN I/O processor (summation of all internal masters on secondary interface) on the secondary PCI bus
i960 RM/RN processor internal bus	Acquisition latency and ownership metrics for the 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 on the i960 RM/RN I/O processor 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 i960 RM/RN I/O processor)
M4	Secondary PCI bus and PCI agents (external masters 3..5 and i960 RM/RN I/O processor processor)
M5	Internal bus: DMA Channels and Application Accelerator
M6	Internal bus: PATU, SATU and i960 processor
M7	Internal bus: Primary PCI bus, Secondary PCI bus and Secondary PCI agents (external masters 0..5 & i960 RM/RN 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 i960 RM/RN 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 i960 RM/RN 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.
- **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 i960 RM/RN I/O processor can either be duration events or occurrence events. There are 98 events monitored on the i960 RM/RN I/O processor. A maximum of fourteen (14) events can be monitored concurrently. There are eight monitoring modes implemented on the i960 RM/RN I/O processor described below:

- **Mode 0:** Performance Monitoring disabled on the i960 RM/RN 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 i960 RM/RN processor internal bus.
- **Mode 6:** Monitors events on the i960 RM/RN processor internal bus.
- **Mode 7:** Monitors events on the Primary PCI bus, Secondary PCI bus and i960 RM/RN processor internal bus.

### 21.3.1 Mode0: Performance Monitoring Disabled

Programming Mode0 (M0) in the ESR disables performance monitoring on the i960 RM/RN 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 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.

### 21.3.2.2 M1\_PPCIBus\_data

This duration event increments the counter every primary PCI data cycle. Data cycles comprise of two instances:

- The i960 RM/RN 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 i960 RM/RN I/O processor or to other masters on the bus.

### 21.3.2.3 M1\_PPCIBus\_bridge\_acq

This duration event counts the number of clocks spent by the bridge acquiring the primary PCI bus. 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 also increments 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\_PPCIBus\_bridge\_own

This duration event counts the duration for which the bridge is the master on the primary PCI bus. The counter increments on every clock cycle during which the bridge is the bus master.

### 21.3.2.5 M1\_PPCIBus\_DMA0\_acq

This duration event counts the number of clocks spent by the DMA Ch-0 acquiring the primary PCI bus. 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 also increments 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\_PPCIBus\_DMA0\_own

This duration event counts the duration for which DMA Ch-0 is the master on the primary PCI bus. The counter increments on every clock cycle during which the channel is the bus master.

### 21.3.2.7 M1\_PPCIBus\_DMA1\_acq

This duration event counts the number of clocks spent by the DMA Ch-1 acquiring the primary PCI bus. 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 also increments 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\_PPCIBus\_DMA1\_own

This duration event counts the duration for which DMA Ch-1 is the master on the primary PCI bus. The counter increments on every clock cycle during which the channel is the bus master.

### 21.3.2.9 M1\_PPCIBus\_PATU\_acq

This duration event counts the number of clocks spent by the PATU acquiring the primary PCI bus. 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 also increments 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\_PPCIBus\_PATU\_own

This duration event counts the duration for which PATU is the master on the primary PCI bus. The counter increments on every clock cycle during which the unit is the bus master.

### 21.3.2.11 M1\_PPCIBus\_DMA0\_gnt

This occurrence event monitors the number of times DMA Ch-0 is granted the primary 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\_PPCIBus\_DMA1\_gnt

This occurrence event monitors the number of times DMA Ch-1 is granted the primary 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\_PPCIBus\_PATU\_gnt

This occurrence event monitors the number of times the PATU is granted the primary 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\_PPCCIBus\_bridge\_gnt

This occurrence event monitors the number of times the bridge is granted the primary 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\_data

This duration event increments the counter every secondary PCI data cycle. Data cycles comprise of two instances:

- The i960 RM/RN 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 i960 RM/RN I/O processor or to other masters on the bus.

#### 21.3.3.3 M2\_SPCIBus\_SATU\_acq

This duration event counts the number of clocks spent by the SATU acquiring the secondary PCI bus. 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 also increments 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\_SPCIBus\_SATU\_own

This duration event counts the duration for which SATU is the master on the secondary PCI bus. The counter increments on every clock cycle during which the unit is the bus master.

#### 21.3.3.5 M2\_SPCIBus\_bridge\_acq

This duration event counts the number of clocks spent by the bridge acquiring the secondary PCI bus. 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 also increments 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\_SPCIBus\_bridge\_own**

This duration event counts the duration for which the bridge is the master on the secondary PCI bus. The counter increments on every clock cycle during which the bridge is the bus master.

### 21.3.3.7 **M2\_SPCIBus\_DMA2\_acq**

This duration event counts the number of clocks spent by the DMA Ch-2 acquiring the secondary PCI bus. 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 also increments 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-2) to calculate the average acquisition latency for the channel.

### 21.3.3.8 **M2\_SPCIBus\_DMA2\_own**

This duration event counts the duration for which DMA Ch-2 is the master on the secondary PCI bus. The counter increments on every clock cycle during which the channel is the bus master.

### 21.3.3.9 **M2\_SPCIBus\_bridge\_gnt**

This occurrence event monitors the 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\_SPCIBus\_SATU\_gnt**

This occurrence event monitors the number of times the SATU is granted the secondary 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\_SPCIBus\_DMA2\_gnt**

This occurrence event monitors the number of times DMA Ch-2 is granted the secondary 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\_PPCCIBus\_data

This duration event increments the counter every primary PCI data cycle. Data cycles comprise of two instances:

- The i960 RM/RN 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 i960 RM/RN I/O processor or to other masters on the bus.

### 21.3.3.14 M2\_IBus\_data

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 (i960 RM/RN I/O processor, Master0, Master1, Master2) on the PCI bus. Master0 indicates the external PCI device connected to the REQ0 and GNT0 signals of the internal arbiter in the i960 RM/RN 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 i960 RM/RN I/O processor in this mode.

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 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.

### 21.3.4.2 M3\_SPCibus\_data

This duration event increments the counter every secondary PCI data cycle. Data cycles comprise of two instances:

- The i960 RM/RN 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 i960 RM/RN I/O processor or to other masters on the bus.

### 21.3.4.3 M3\_SPCibus\_IOP\_acq

This duration event counts the number of clocks spent by the i960 RM/RN I/O processor (includes the bridge, dma Ch-2, and satu) acquiring the PCI bus. The counter increments on every clock cycle after the processor has requested use of the PCI bus but has not actively driven the PCI bus as a master. The counter also increments 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 i960 RM/RN I/O processor) to calculate the average acquisition latency for the processor.

#### 21.3.4.4 M3\_SPClbus\_IOP\_own

This duration event counts the duration for which the i960 RM/RN 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. This measure combines the individual ownership times of the bridge, DMA ch-2 and SATU.

#### 21.3.4.5 M3\_SPClbus\_D0\_acq

This duration event counts the number of clocks spent by PCI Master 0 acquiring the secondary PCI bus. 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 also increments 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\_SPClbus\_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.

#### 21.3.4.7 M3\_SPClbus\_D1\_acq

This duration event counts the number of clocks spent by PCI Master 1 acquiring the secondary PCI bus. 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 also increments 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\_SPClbus\_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.

#### 21.3.4.9 M3\_SPClbus\_D2\_acq

This duration event counts the number of clocks spent by PCI Master 2 acquiring the secondary PCI bus. 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 also increments 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\_SPClbus\_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.



#### 21.3.4.11 M3\_SPClbus\_IOP\_gnt

This occurrence event monitors the number of times the i960 RM/RN I/O processor is granted the secondary PCI bus. It increments the counter when the processor 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. The count value is a summation of the individual grants received by the bridge, satu and dma Ch-2.

#### 21.3.4.12 M3\_SPClbus\_D0\_gnt

This occurrence event monitors the number of times PCI Master 0 is granted the secondary 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\_SPClbus\_D1\_gnt

This occurrence event monitors the number of times PCI Master 1 is granted the secondary 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\_SPClbus\_D2\_gnt

This occurrence event monitors the number of times PCI Master 2 is granted the secondary 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 (i960 RM/RN I/O processor, Master3, Master4 and Master5) on the PCI bus. Master3 indicates the external PCI device connected to REQ3 and GNT3 signals of the internal arbiter in the i960 RM/RN 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\_SPClbus\_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.

### 21.3.5.2 M4\_SPClbus\_data

This duration event increments the counter every secondary PCI data cycle. Data cycles comprise of two instances:

- The i960 RM/RN 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 i960 RM/RN I/O processor or to other masters on the bus.

### 21.3.5.3 M4\_SPClbus\_D3\_acq

This duration event counts the number of clocks spent by PCI Master 3 acquiring the secondary PCI bus. 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 also increments 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\_SPClbus\_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.

### 21.3.5.5 M4\_SPClbus\_D4\_acq

This duration event counts the number of clocks spent by PCI Master 4 acquiring the secondary PCI bus. 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 also increments 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\_SPClbus\_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.

### 21.3.5.7 M4\_SPClbus\_D5\_acq

This duration event counts the number of clocks spent by PCI Master 5 acquiring the secondary PCI bus. 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 also increments 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\_SPClbus\_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.

### 21.3.5.9 M4\_SPClbus\_D3\_gnt

This occurrence event monitors the number of times PCI Master 3 is granted the secondary 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\_SPClbus\_D4\_gnt

This occurrence event monitors the number of times PCI Master 4 is granted the secondary 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\_SPClbus\_D5\_gnt

This occurrence event monitors the number of times PCI Master 5 is granted the secondary 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\_SPClbus\_IOP\_gnt

This occurrence event monitors the number of times the i960 RM/RN I/O processor is granted the secondary PCI bus. It increments the counter when the processor 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. The count value is a summation of the individual grants received by the bridge, satu and dma Ch-2.

### 21.3.5.13 M4\_SPClbus\_IOP\_acq

This duration event counts the number of clocks spent by the i960 RM/RN I/O processor (includes the bridge, dma Ch-2, and satu) acquiring the secondary PCI bus. The counter increments on every clock cycle after the processor has requested use of the PCI bus but has not actively driven the PCI bus as a master. The counter also increments 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 i960 RM/RN I/O processor) to calculate the average acquisition latency for the processor.

### 21.3.5.14 M4\_SPClbus\_IOP\_own

This duration event counts the duration for which the i960 RM/RN 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.

## 21.3.6 Mode 5: i960<sup>®</sup> RM/RN I/O Processor Internal Bus and Agents Events

Programming Mode5 (M5) in the ESR enables performance monitoring on the i960 RM/RN I/O processor internal bus. In addition, performance monitoring is done for selected agents. In this mode, the monitored agents are: DMA channels (Ch-0, Ch-1 and Ch-2) and the Application Accelerator. All counters are clocked at the internal bus frequency. The following sections describe the 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\_data

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\_IBus\_AAU\_acq

This duration event counts the number of clocks spent by the Application Accelerator (AA) acquiring the internal bus. 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 also increments 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\_IBus\_AAU\_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\_IBus\_DMA0\_acq

This duration event counts the number of clocks spent by DMA Ch-0 acquiring the internal bus. 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 also increments 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\_IBus\_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\_IBus\_DMA1\_acq

This duration event counts the number of clocks spent by DMA Ch-1 acquiring the internal bus. 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 also increments 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\_IBus\_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\_IBus\_DMA2\_acq

This duration event counts the number of clocks spent by DMA Ch-2 acquiring the internal bus. 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 also increments 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\_IBus\_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\_IBus\_AAU\_gnt

This occurrence event monitors the number of times the AA is granted the internal 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\_IBus\_DMA0\_gnt

This occurrence event monitors the number of times DMA Ch-0 is granted the internal 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\_IBus\_DMA1\_gnt

This occurrence event monitors the number of times DMA Ch-1 is granted the internal 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\_IBus\_DMA2\_gnt

This occurrence event monitors the number of times DMA Ch-2 is granted the internal 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: i960<sup>®</sup> RM/RN I/O Processor Internal Bus and Agents Events

Programming Mode6 (M6) in the ESR enables performance monitoring on the i960 RM/RN 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\_IBus\_core\_acq

This duration event counts the number of clocks spent by i960 processor acquiring the internal bus. The counter increments on 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 this agent's *Request Signal* is asserted but bus ownership currently belongs to another master.

### 21.3.7.2 M6\_IBus\_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 during which the i960 processor is the bus master.

### 21.3.7.3 M6\_IBus\_PATU\_acq

This duration event counts the number of clocks spent by PATU acquiring the internal bus. 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 also increments 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\_IBus\_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\_IBus\_SATU\_acq

This duration event counts the number of clocks spent by SATU acquiring the internal bus. 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 also increments 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\_IBus\_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\_IBus\_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\_IBus\_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\_IBus\_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\_IBus\_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\_IBus\_PATU\_gnt

This occurrence event monitors the number of times the PATU is granted the internal 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\_IBus\_SATU\_gnt

This occurrence event monitors the number of times the SATU is granted the internal 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\_IBus\_core\_gnt

This occurrence event monitors the number of times the core is granted the internal 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.

### 21.3.7.14 M6\_IBus\_ATU\_retries

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 are not included in this metric.

## 21.3.8 Mode 7: i960<sup>®</sup> RM/RN Processor Internal Bus, Secondary PCI Bus and Primary PCI Bus Events

Programming Mode7 (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 (i960 RM/RN I/O processor and external masters 0..5) on the secondary bus and for i960 RM/RN 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 i960 RM/RN 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\_data

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\_data

This duration event increments the counter every secondary PCI data cycle. Data cycles comprise of two instances:

- The i960 RM/RN 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 i960 RM/RN I/O processor or to other masters on the bus.

### 21.3.8.5 M7\_SPCibus\_IOP\_own

This duration event counts the duration for which the i960 RM/RN 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. The count value is a summation of ownership times of the bridge, SATU and DMA Ch-2.



**21.3.8.6 M7\_SPClbus\_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\_SPClbus\_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\_SPClbus\_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\_SPClbus\_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\_SPClbus\_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\_SPClbus\_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\_PPClbus\_IOP\_own**

This duration event counts the duration for which the i960 RM/RN 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. The count value is a summation of ownership times of the bridge, PATU and DMA Ch-0 and 1.

**21.3.8.13 M7\_PPClbus\_idle**

This duration event increments the counter every 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\_PPClbus\_data**

This duration event increments the counter every primary PCI data cycle. Data cycles comprise of two instances:

- The i960 RM/RN 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 i960 RM/RN 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 i960 RM/RN 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 i960 RM/RN 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 the table.

**Table 21-4. Event Monitor Register Table**

Section, Register Name - Acronym (Page)
Section 21.6.1, "Global Timer Mode Register (GTMR)" on page 21-21
Section 21.6.2, "Event Select Register (ESR)" on page 21-22
Section 21.6.3, "Event Monitoring Interrupt Status Register (EMISR)" on page 21-23
Section 21.6.4, "Global Time Stamp Register (GTSR)" on page 21-24
Section 21.6.5, "Programmable Event Counter Register (PECRx)" on page 21-25

## 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)**

Bit	Default	Description
31:03	0	Reserved
2	0 <sub>2</sub>	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 <sub>2</sub>	Reserved
0	0 <sub>2</sub>	Bit value determines whether the Global Time Stamp Counter interrupts the processor on an overflow condition. 0 Interrupt disabled 1 Interrupt enabled

Internal Bus Address 0000 1100H	Attribute Legend: RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set RC = Read Clear RO = Read Only NA = Not Accessible
------------------------------------	--

## 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)**

Internal Bus Address 0000 1104H		Attribute Legend: RW = Read/Write RV = Reserved RC = Read Clear PR = Preserved RO = Read Only RS = Read/Set NA = Not Accessible
Bit	Default	Description
31:17	0	Reserved
16	0 <sub>2</sub>	PECCR <sub>x</sub> Master Interrupt Enable: When set (1), any/all the programmable event counters interrupt the i960 processor on an overflow. When clear (0), none of the programmable event counters interrupt the processor on an overflow. In this mode, any counter that has an overflow condition rolls over to zero and start incrementing.
15:3	0	Reserved
2:0	0	Value in this bit field determines the monitored interface on the i960 RM/RN I/O processor. 000 Mode 0 Performance Monitoring Disabled 001 Mode 1 Primary PCI Bus & Internal Agents 010 Mode 2 Secondary PCI Bus & Internal Agents 011 Mode 3 Secondary PCI Bus & PCI Agents 100 Mode 4 Secondary PCI Bus & PCI Agents (external masters 3..5) 101 Mode 5 i960 RM/RN processor internal bus, DMA Channels & AA 110 Mode 6 i960 RM/RN processor internal bus, PATU, SATU & i960 processor 111 Mode 7 i960 RM/RN 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 i960 RM/RN 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**

Bit	Default	Description
31:15	0	Reserved
14:1	0 <sub>2</sub>	Bit value indicates status of the Programmable Event Counter x(PECx) during event monitoring. When clear (0), no PECx overflow interrupt is pending. When set (1), a PECx overflow interrupt is pending.
0	0 <sub>2</sub>	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.

Internal Bus Address 0000 1108H	Attribute Legend: RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set RC = Read Clear RO = Read Only NA = Not Accessible
------------------------------------	--

IOP Attributes 31 28 24 20 16 12 8 4 0 rv rv rv rv rv rv rv rv rv rv rv rv rv rv rv rc rc rc rc rc rc rc rc rc rc rc rc rc rc	PCI Attributes na
---	---

## 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 returns the current count value.

**Table 21-8. Global Time Stamp Register - GTSR**

		<p>Internal Bus Address 0000 1110H</p> <p>Attribute Legend:                  RC = Read Clear                  RV = Reserved                  PR = Preserved                  RS = Read/Set                  RO = Read Only                  NA = Not Accessible</p>
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 21-9](#). 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.

**Table 21-9. Programmable Event Counter Register - PECRx**

Bit	Default	Description
31:00	X	This is a 32-bit, read-only register. When accessed, it returns the current count value in the respective event counter.

Internal Bus Address	Attribute Legend:
PECR1 0000 1114H	RW = Read/Write
PECR2 0000 1118H	RV = Reserved
PECR3 0000 111CH	PR = Preserved
PECR4 0000 1120H	RC = Read Clear
PECR5 0000 1124H	RO = Read Only
PECR6 0000 1128H	RS = Read/Set
PECR7 0000 112CH	NA = Not Accessible
PECR8 0000 1130H	
PECR9 0000 1134H	
PECR10 0000 1138H	
PECR11 0000 113CH	
PECR12 0000 1140H	
PECR13 0000 1144H	
PECR14 0000 1148H	





This chapter describes the I<sup>2</sup>C (Inter-Integrated Circuit) bus interface unit of the i960<sup>®</sup> RM/RN I/O Processor, including the operation modes and setup. Throughout this manual, this peripheral is referred to as the I<sup>2</sup>C unit.

## 22.1 Overview

The I<sup>2</sup>C Bus Interface Unit allows the i960 RM/RN I/O processor to serve as a master and slave device residing on the I<sup>2</sup>C bus. The I<sup>2</sup>C 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 I<sup>2</sup>C bus.

The I<sup>2</sup>C bus allows the i960 RM/RN I/O processor to interface to other I<sup>2</sup>C 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 i960 RM/RN I/O processor subsystem to an external device.

The I<sup>2</sup>C Bus Interface Unit is a peripheral device that resides on the i960 RM/RN I/O processor internal bus. Data is transmitted to and received from the I<sup>2</sup>C bus via a buffered interface. Control and status information is relayed through a set of memory-mapped registers. Refer to the I<sup>2</sup>C Bus Specification for complete details on I<sup>2</sup>C bus operation.

## 22.2 Theory of Operation

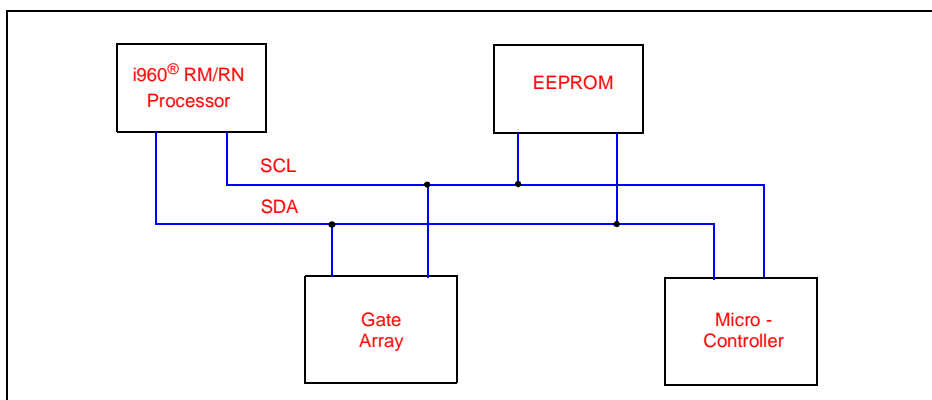
The I<sup>2</sup>C bus defines a serial protocol for passing information between agents on the I<sup>2</sup>C bus using only a two pin interface. The interface consists of a Serial Data/Address (SDA) line and a Serial Clock Line (SCL). Each device on the I<sup>2</sup>C bus is recognized by a unique 7-bit address and can operate as a transmitter or as a receiver. In addition to transmitter and receiver, the I<sup>2</sup>C bus uses the concept of master and slave. [Table 22-1](#) lists the I<sup>2</sup>C device types.

**Table 22-1. I<sup>2</sup>C Bus Definitions**

I <sup>2</sup> C Device	Definition
Transmitter	Sends data to the I <sup>2</sup> C bus.
Receiver	Receives data from the I <sup>2</sup> C 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 I<sup>2</sup>C bus operation, consider the case of the i960 RM/RN I/O processor acting as a master on the bus (Figure 22-1). The i960 RM/RN I/O processor, as a master, addresses an EEPROM as a slave to receive data. The i960 RM/RN I/O processor is a master-transmitter and the EEPROM is a slave-receiver. When the i960 RM/RN I/O processor reads data, the i960 RM/RN 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. I<sup>2</sup>C Bus Configuration Example



The I<sup>2</sup>C 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 I<sup>2</sup>C bus arbitration relies on the wired-AND connection of all I<sup>2</sup>C interfaces to the I<sup>2</sup>C 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 low 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.

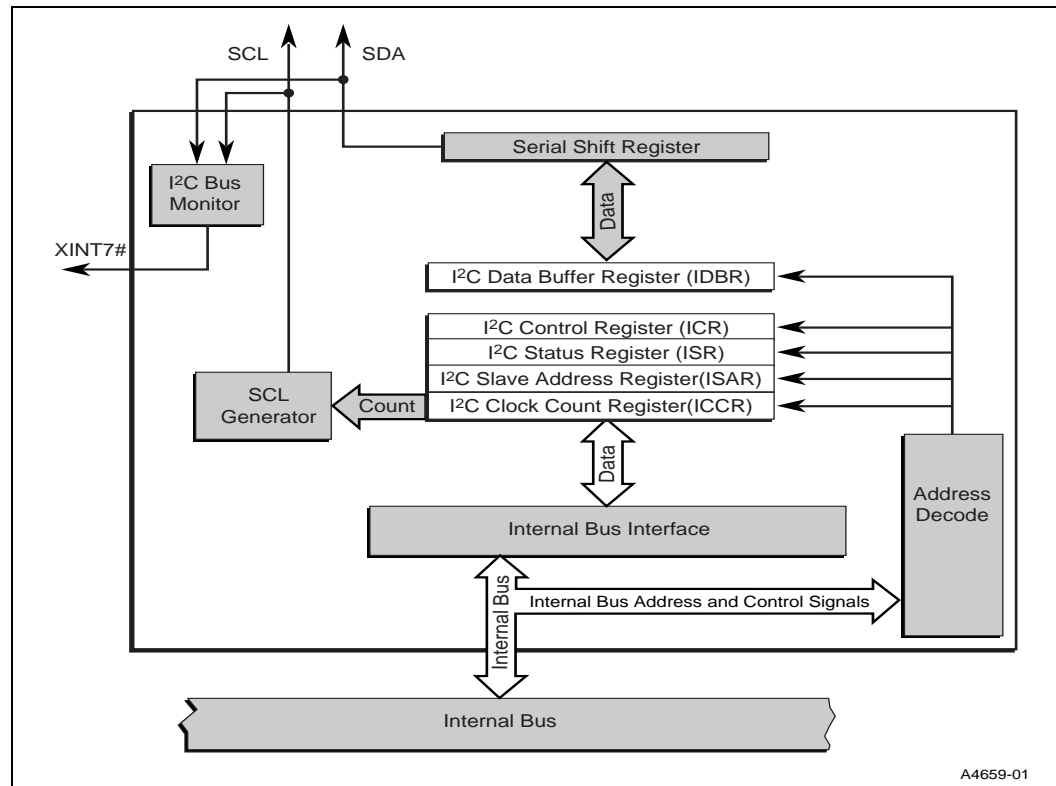
The I<sup>2</sup>C bus serial operation uses an open-drain wired-AND bus structure, which allows multiple devices to drive the bus lines and to communicate status about events such as arbitration, wait states, error conditions and so on. For example, when a master drives the clock (SCL) line during a 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.

I<sup>2</sup>C transactions are either initiated by the i960 RM/RN 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 I<sup>2</sup>C bus.

## 22.2.1 Operational Blocks

The I<sup>2</sup>C Bus Interface Unit is a slave peripheral device that is connected to the internal bus. The i960 RM/RN I/O processor interrupt mechanism can be used for notifying the i960 RM/RN I/O processor that there is activity on the I<sup>2</sup>C 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 I<sup>2</sup>C Bus Interface Unit and its interface to the internal bus.

The I<sup>2</sup>C Bus Interface Unit consists of the two wire interface to the I<sup>2</sup>C bus, an 8-bit buffer for passing data to and from the i960 RM/RN I/O processor, a set of control and status registers, and a shift register for parallel/serial conversions.

**Figure 22-2. I<sup>2</sup>C Bus Interface Unit Block Diagram**


The I<sup>2</sup>C interrupts are signalled through i960 RM/RN I/O processor interrupt XINT7# and the XINT7 Interrupt Status Register (X7ISR) in the PCI and Peripheral Interrupt Controller (Chapter 8, “PCI and Peripheral Interrupt Controller Unit”). The I<sup>2</sup>C 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, “I<sup>2</sup>C Status Register- ISR” on page 22-27 for details.

The I<sup>2</sup>C Data Buffer Register (IDBR) is an 8-bit data buffer that receives a byte of data from the shift register interface of the I<sup>2</sup>C bus on one side and parallel data from the i960 RM/RN 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 I<sup>2</sup>C memory-mapped address space (1680H to 1690H). The registers and their function are defined in Section 22.8.

The I<sup>2</sup>C 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 I<sup>2</sup>C is designed to meet the slower, standard mode requirements for these two specifications. Refer to any of the following literature for information on I<sup>2</sup>C bus operation:

<i>I<sup>2</sup>C Peripherals for Microcontrollers</i>	Philips Semiconductor
<i>I<sup>2</sup>C Bus and How to Use It (Including Specifications)</i>	Philips Semiconductor
<i>I<sup>2</sup>C Peripherals for Microcontrollers (Including Fast Mode)</i>	Signetics

## 22.2.2 I<sup>2</sup>C Bus Interface Modes

The I<sup>2</sup>C 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	<ul style="list-style-type: none"> <li>I<sup>2</sup>C Bus Interface Unit acts as a master.</li> <li>Used for a write operation.</li> <li>I<sup>2</sup>C Bus Interface Unit sends the data.</li> <li>I<sup>2</sup>C Bus Interface Unit is responsible for clocking.</li> <li>Slave device is in slave-receive mode</li> </ul>
Master - Receive	<ul style="list-style-type: none"> <li>I<sup>2</sup>C Bus Interface Unit acts as a master.</li> <li>Used for a read operation.</li> <li>I<sup>2</sup>C Bus Interface Unit receives the data.</li> <li>I<sup>2</sup>C Bus Interface Unit is responsible for clocking.</li> <li>Slave device is in slave-transmit mode</li> </ul>
Slave - Transmit	<ul style="list-style-type: none"> <li>I<sup>2</sup>C Bus Interface Unit acts as a slave.</li> <li>Used for a read (master) operation.</li> <li>I<sup>2</sup>C Bus Interface Unit sends the data.</li> <li>Master device is in master-receive mode.</li> </ul>
Slave - Receive (default)	<ul style="list-style-type: none"> <li>I<sup>2</sup>C Bus Interface Unit acts as a slave.</li> <li>Used for a write (master) operation.</li> <li>I<sup>2</sup>C Bus Interface Unit receives the data.</li> <li>Master device is in master-transmit mode.</li> </ul>

While the I<sup>2</sup>C 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 i960 RM/RN I/O processor.

When the I<sup>2</sup>C Bus Interface Unit receives an address that matches the 7-bit address found in the I<sup>2</sup>C Slave Address Register (ISAR) or the General Call Address (00H), the interface either remains 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 I<sup>2</sup>C Bus Interface Unit remains 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 22-16](#).

When the i960 RM/RN I/O processor wants to initiate a read or write on the I<sup>2</sup>C bus, the I<sup>2</sup>C Bus Interface Unit transitions from the default Slave-Receive mode to Master-Transmit mode. If the i960 RM/RN I/O processor wants to write data, the interface remains in Master-Transmit mode after the address transfer has completed. ([Section 22.2.3.1, “START Condition” on page 22-6](#) for START information). If the i960 RM/RN I/O processor wants to read data, the I<sup>2</sup>C Bus Interface Unit transmits the start address, then transition to Master-Receive mode. Master operation is further defined in [Section 22.3.5, “Master Operations” on page 22-13](#).

### 22.2.3 Start and Stop Bus States

The I<sup>2</sup>C 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 i960 RM/RN I/O processor uses the START and STOP bits in the I<sup>2</sup>C Control Register (ICR) to:

- initiate an additional byte transfer
- initiate a START condition on the I<sup>2</sup>C bus
- enable Data Chaining (repeated START)
- initiate a STOP condition on the I<sup>2</sup>C bus

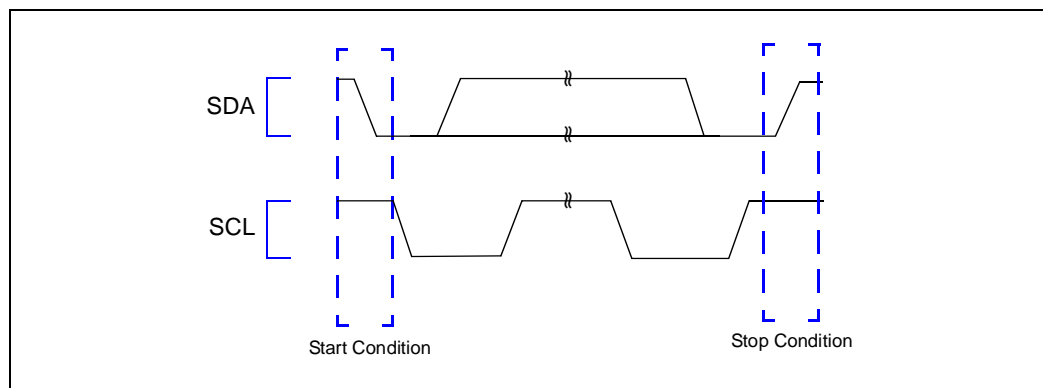
Table 22-3 summarizes the definition of the START and STOP bits in the ICR.

**Table 22-3. START and STOP Bit Definitions**

STOP bit	START bit	Condition	Notes
0	0	No START or STOP	<ul style="list-style-type: none"> <li>• No START or STOP condition is sent by the I<sup>2</sup>C Bus Interface Unit. This is used when multiple data bytes need to be transferred.</li> </ul>
0	1	START Condition & Repeated START	<ul style="list-style-type: none"> <li>• The I<sup>2</sup>C Bus Interface Unit sends 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.</li> <li>• For a repeated start, the IDBR contents contains the target slave address and the R/W# bit. This enables multiple transfers to different slaves without giving up the bus.</li> <li>• The interface stays in Master-Transmit mode if a write is used or transition to master-receive mode if a read is requested.</li> </ul>
1	X	STOP Condition	<ul style="list-style-type: none"> <li>• In Master-Transmit mode, the I<sup>2</sup>C Bus Interface Unit transmits the 8-bit IDBR and then send a STOP on the I<sup>2</sup>C bus.</li> <li>• In Master-Receive mode, the Ack/Nack Control bit in the ICR must be changed to a negative Ack (<a href="#">Section 22.3.3</a>). The I<sup>2</sup>C Bus Interface Unit writes the Nack bit (Ack/Nack Control bit must be 1), receive the data byte in the IDBR, then send a STOP on the I<sup>2</sup>C bus.</li> </ul>

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<sub>2</sub>) 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, “I<sup>2</sup>C Data Buffer Register- IDBR” on page 22-30](#)) before setting the START ICR bit. The START and the IDBR contents are transmitted on the I<sup>2</sup>C bus when the ICR Transfer Byte bit is set. The I<sup>2</sup>C 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<sup>2</sup>C unit. When arbitration is lost while initiating a START, the I<sup>2</sup>C unit may re-attempt the START when the bus becomes free. See [Section 22.3.4, “Arbitration” on page 22-11](#) for details on how the I<sup>2</sup>C unit functions under those circumstances.

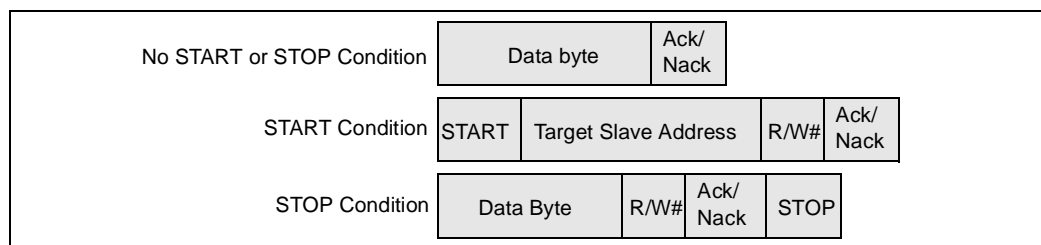
### 22.2.3.2 No START or STOP Condition

No START or STOP condition (bits 1:0 of the ICR set to 00<sub>2</sub>) is used in master-transmit mode while the i960 RM/RN I/O processor is transmitting multiple data bytes ([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<sup>2</sup>C unit after the transmission of a START, STOP or repeated START.

After each byte transfer (including the Ack/Nack bit) the I<sup>2</sup>C unit holds the SCL line low (inserting wait states) until the Transfer Byte bit in the ICR is set. This action notifies the I<sup>2</sup>C 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<sub>2</sub>) 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 ([Figure 22-3](#)). In master-receive mode, to initiate the last transfer the i960 RM/RN 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 I<sup>2</sup>C Bus Operation

The I<sup>2</sup>C Bus Interface Unit transfers in 1 byte increments. A data transfer on the I<sup>2</sup>C 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 i960 RM/RN I/O processor's I<sup>2</sup>C unit is required to generate the I<sup>2</sup>C 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 = ICCR Decimal Value \* i960 RM/RN 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 I<sup>2</sup>C unit performs the necessary clock synchronization. The ICCR provides a simple method for determining I<sup>2</sup>C clock frequencies. Table 22-4 details sample programming values for the ICCR.

**Table 22-4. ICCR Programming Values**

PCI Bus Frequency	i960 <sup>®</sup> RM/RN I/O Processor Internal Bus Frequency	I <sup>2</sup> C Clock Frequency = [1/(SCL Transition Per. * 2)]	SCL Transition Period	ICCR Value		
33 MHz	66 MHz	397.59 KHz	1.26 μs	01010011 <sub>2</sub>	053H	83
		99.10 KHz	5.05 μs	10100110 <sub>2</sub>	14DH	333
25 MHz	50 MHz	397.59 KHz	1.26 μs	00111111 <sub>2</sub>	03FH	63
		99.20 KHz	5.04 μs	11111100 <sub>2</sub>	0FCH	252

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 I<sup>2</sup>C Data Buffer Register (IDBR) and the I<sup>2</sup>C Slave Address Register (ISAR). The IDBR ([Section 22.8.4, “I<sup>2</sup>C Data Buffer Register- IDBR” on page 22-30](#)) contains data or a slave address and R/W# bit. The ISAR contains the i960 RM/RN I/O processor's programmable slave address. Data coming into the I<sup>2</sup>C 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 I<sup>2</sup>C unit passes this onto the serial bus when the Transfer Byte bit in the ICR is set. See [Section 22.8.1, “I<sup>2</sup>C Control Register- ICR” on page 22-24](#).

When the I<sup>2</sup>C 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 I<sup>2</sup>C 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 I<sup>2</sup>C bus and the acknowledge cycle is complete.
4. When the I<sup>2</sup>C bus is ready to transfer the next byte before the processor has written the IDBR (and a STOP condition is not in place), the I<sup>2</sup>C unit inserts wait states until the processor writes a new value into the IDBR and sets the ICR Transfer Byte bit.

When the I<sup>2</sup>C unit is in receive mode (master or slave):

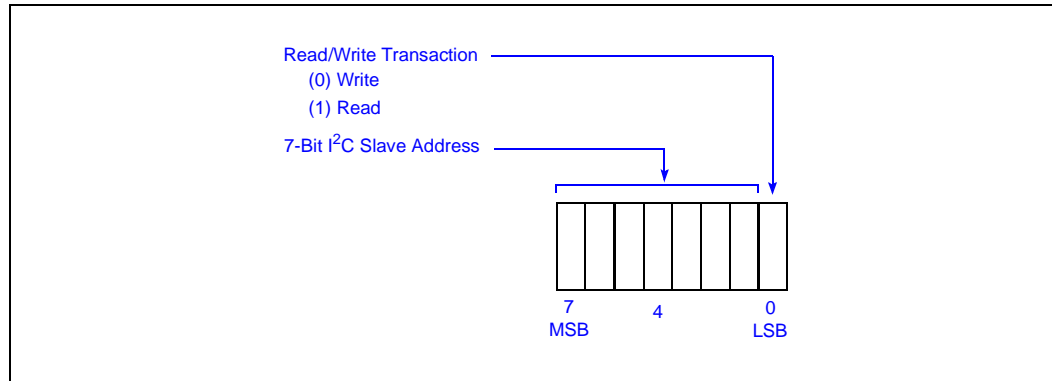
1. The processor reads the IDBR data over the internal bus after the IDBR Receive Full interrupt is signalled.
2. The I<sup>2</sup>C unit transfers data from the shift register to the IDBR after the Ack cycle completes.
3. The I<sup>2</sup>C unit inserts wait states until the IDBR is read. Refer to [Section 22.3.3, “I<sup>2</sup>C Acknowledge” on page 22-10](#) for acknowledge pulse information in receiver mode.
4. After the processor reads the IDBR, the I<sup>2</sup>C 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 I<sup>2</sup>C 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 (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 I<sup>2</sup>C unit remains in master-transmit mode and the addressed slave device stays in slave-receive mode. When the transaction is a read, the I<sup>2</sup>C 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 I<sup>2</sup>C unit aborts the transaction by automatically sending a STOP and setting the ISR bus error bit.

When the I<sup>2</sup>C unit is enabled and idle (no bus activity), it stays in slave-receive mode and monitors the I<sup>2</sup>C bus for a START signal. Upon detecting a START pulse, the I<sup>2</sup>C unit reads the first seven bits and compares them to those in the I<sup>2</sup>C Slave Address Register (ISAR) and the general call address (00H). When the bits match those of the ISAR register, the I<sup>2</sup>C unit reads the eighth bit (R/W# bit) and transmits an Ack pulse. The I<sup>2</sup>C 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 22-18 for actions when a general call address is detected.

### 22.3.3 I<sup>2</sup>C Acknowledge

Every I<sup>2</sup>C 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 (Figure 22-6).

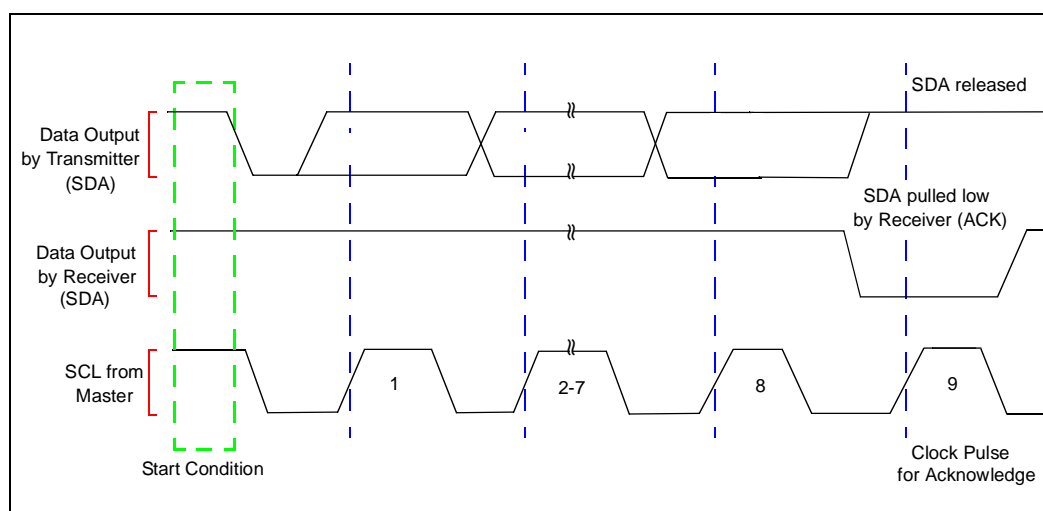
In master-transmit mode, when the target slave receiver device cannot generate the acknowledge pulse, the SDA line remains high. This lack of acknowledge (Nack) causes the I<sup>2</sup>C unit to set the bus error detected bit in the ISR and generate the associated interrupt (when enabled). The I<sup>2</sup>C unit aborts the transaction by generating a STOP automatically.

In master-receive mode, the I<sup>2</sup>C unit signals the slave-transmitter to stop sending data by using the negative acknowledge (Nack). The Ack/Nack bit value driven by the I<sup>2</sup>C 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 I<sup>2</sup>C bus protocol). The I<sup>2</sup>C 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 I<sup>2</sup>C 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 I<sup>2</sup>C 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) remains set until a STOP or repeated START is received.

Figure 22-6. Acknowledge on the I<sup>2</sup>C Bus



## 22.3.4 Arbitration

Arbitration on the I<sup>2</sup>C bus is required due to the multi-master capabilities of the I<sup>2</sup>C bus. Arbitration is used when two or more masters simultaneously generate a START condition within the minimum I<sup>2</sup>C hold time of the START condition.

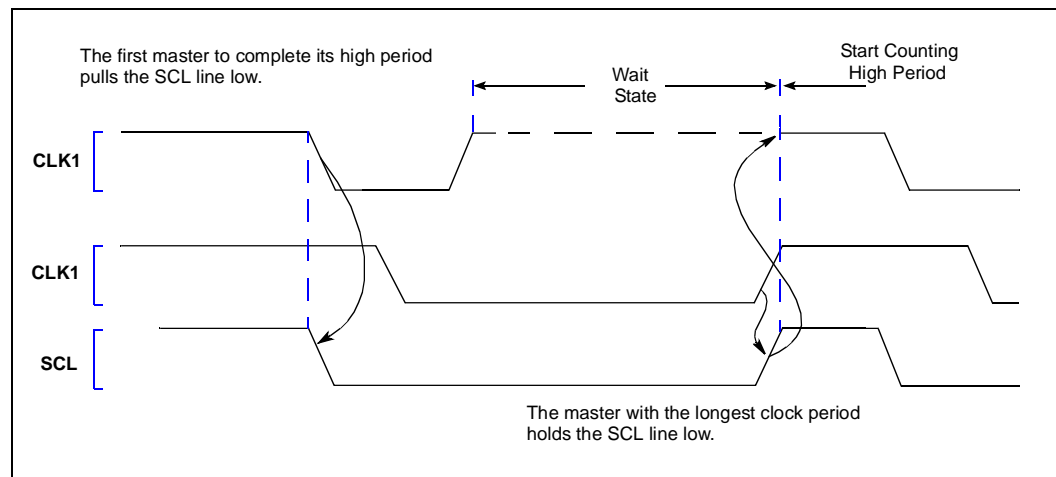
Arbitration can continue for a long period. If the address bit and the R/W# are the same, the arbitration moves to the data. Due to the wired-AND nature of the I<sup>2</sup>C bus, no data is lost if both (or all) masters are outputting the same bus states. If the address, the R/W# bit, or the data are different, the master which outputted the high state (master's data is different from SDA) loses arbitration and shut its data drivers off. When losing arbitration, the I<sup>2</sup>C Bus Interface Unit shuts 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 I<sup>2</sup>C 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 I<sup>2</sup>C 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 (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.

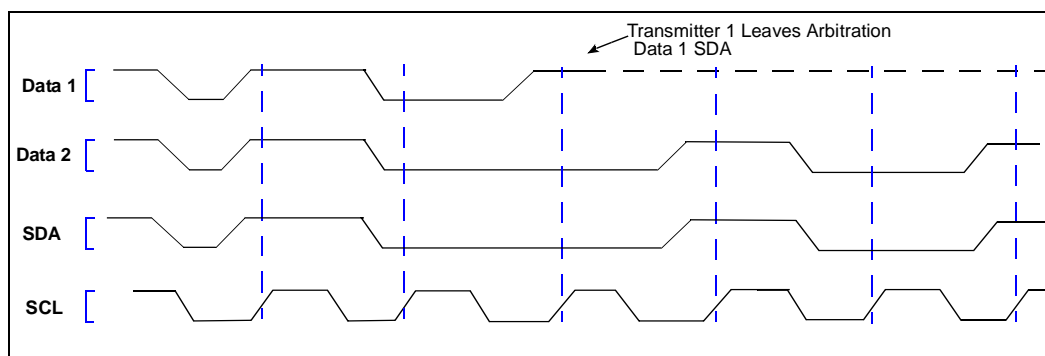
Figure 22-7. Clock Synchronization During the Arbitration Procedure



### 22.3.4.2 SDA Arbitration

Arbitration on the SDA line can continue for a long period starting with the address and R/W# bits and continuing with the 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 bit and the R/W# are the same, the arbitration moves to the data. Due to the wired-AND nature of the I<sup>2</sup>C bus, no data is lost when both (or all) masters are outputting the same bus states. When the address, R/W# bit, or data is different, the master that output the first low data bit loses arbitration and shuts its data drivers off. When the I<sup>2</sup>C unit loses arbitration, it shuts off the SDA or SCL drivers for the remainder of the byte transfer, sets the arbitration loss detected ISR bit, then returns to idle (Slave-Receive) mode.

Figure 22-8. Arbitration Procedure of Two Masters



When the I<sup>2</sup>C unit loses arbitration during transmission of the seven address bits and the i960 RM/RN I/O processor is not being addressed as a slave device, the I<sup>2</sup>C unit re-sends the address when the I<sup>2</sup>C bus becomes free. This is possible because the IDBR and ICR registers are not overwritten when arbitration is lost.

When the arbitration loss is due to another bus master addressing the i960 RM/RN I/O processor as a slave device, the I<sup>2</sup>C unit switches to slave-receive mode and the original data in the I<sup>2</sup>C 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<sup>2</sup>C unit to write to its own slave address. This can cause the I<sup>2</sup>C 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<sup>2</sup>C bus. To prevent errors, the I<sup>2</sup>C 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<sup>2</sup>C 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 I<sup>2</sup>C bus, the I<sup>2</sup>C 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 I<sup>2</sup>C unit has the option of two master modes:

- Master-Transmit — The i960 RM/RN I/O processor writes data
- Master-Receive — The i960 RM/RN I/O processor reads data

The i960 RM/RN I/O processor initiates a master transaction by writing to the ICR register. Data is read and written from the I<sup>2</sup>C unit through the memory-mapped registers.

Table 22-5 describes the I<sup>2</sup>C Bus Interface Unit responsibilities as a master device.

**Table 22-5. Master Transactions (Sheet 1 of 2)**

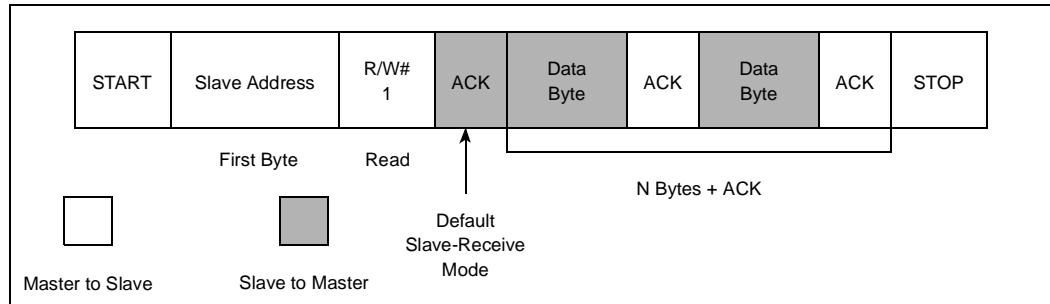
I <sup>2</sup> C Master Action	Mode of Operation	Definition
Generate clock output	Master-transmit Master-receive	<ul style="list-style-type: none"> <li>• The master always drives the SCL line.</li> <li>• The ICCR register is written.</li> <li>• The SCL Enable bit must be set.</li> <li>• The Unit Enable bit must be set.</li> </ul>
Write target slave address to IDBR	Master-transmit Master-receive	<ul style="list-style-type: none"> <li>• The i960 core processor writes to IDBR bits 7-1 before a START condition is enabled.</li> <li>• First 7 bits sent on bus after START.</li> <li>• See <a href="#">Section 22.2.3</a>.</li> </ul>
Write R/W# Bit to IDBR	Master-transmit Master-receive	<ul style="list-style-type: none"> <li>• The i960 core processor writes to the least significant IDBR bit with the target slave address.</li> <li>• If low, the master remains a master-transmitter. If high, the master transitions to a master-receiver.</li> <li>• See <a href="#">Section 22.3.2</a>.</li> </ul>
Signal START Condition	Master-transmit Master-receive	<ul style="list-style-type: none"> <li>• See “Generate clock output” above.</li> <li>• Performed after the target slave address and the R/W# bit are in the IDBR.</li> <li>• i960 core processor sets the START bit.</li> <li>• i960 core processor sets the Transfer Byte bit which initiates the start condition.</li> <li>• See <a href="#">Section 22.2.3</a>.</li> </ul>
Initiate first data byte transfer	Master-transmit Master-receive	<ul style="list-style-type: none"> <li>• i960 core processor writes byte to IDBR</li> <li>• I<sup>2</sup>C Bus Interface Unit transmits the byte when the Transfer Byte bit is set.</li> <li>• I<sup>2</sup>C Bus Interface Unit clears the Transfer Byte bit and sets the IDBR Transmit Empty bit when the transfer is complete.</li> </ul>
Arbitrate for I <sup>2</sup> C Bus	Master-transmit Master-receive	<ul style="list-style-type: none"> <li>• If two or more masters signal a start within the same clock period, arbitration must occur.</li> <li>• The I<sup>2</sup>C Bus Interface Unit arbitrates 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.</li> <li>• If the I<sup>2</sup>C Bus Interface Unit loses arbitration, it sets the Arbitration Loss Detect ISR bit after byte transfer is complete and transition to slave-receive (default) mode.</li> <li>• If I<sup>2</sup>C Bus Interface Unit loses arbitration while attempting to send the target address byte, the I<sup>2</sup>C Bus Interface Unit attempts to resend it when the bus becomes free.</li> <li>• The system designer must ensure the boundary conditions described in <a href="#">Section 22.3</a> do not occur.</li> </ul>

Table 22-5. Master Transactions (Sheet 2 of 2)

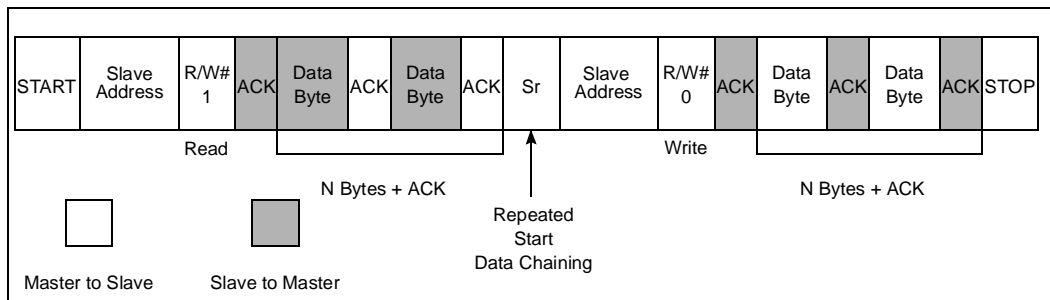
I <sup>2</sup> C Master Action	Mode of Operation	Definition
Write one data byte to the IDBR	Master-transmit only	<ul style="list-style-type: none"> <li>Data transmit mode of I<sup>2</sup>C master operation.</li> <li>Occurs when the IDBR Transmit Empty ISR bit is set and the Transfer Byte bit is clear. If enabled, the IDBR Transmit Empty Interrupt is signalled to the i960 core processor.</li> <li>i960 core processor writes 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.</li> </ul>
Wait for Acknowledge from slave-receiver	Master-transmit only	<ul style="list-style-type: none"> <li>As a master-transmitter, the I<sup>2</sup>C Bus Interface Unit generates the clock for the acknowledge pulse. The I<sup>2</sup>C Bus Interface Unit is responsible for releasing the SDA line to allow slave-receiver Ack transmission.</li> <li>See <a href="#">Section 22.3.3</a>.</li> </ul>
Read one byte of I <sup>2</sup> C Data from the IDBR	Master-receive only	<ul style="list-style-type: none"> <li>Data receive mode of I<sup>2</sup>C master operation.</li> <li>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.</li> <li>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.</li> <li>When the IDBR is read, if the Ack/Nack Status is clear (indicating Ack), the i960 core processor writes the Ack/Nack Control bit and set the Transfer Byte bit to initiate the next byte read.</li> <li>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 I<sup>2</sup>C Bus Interface Unit is sending the STOP.</li> <li>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 sends a repeated start condition, 2. set the Master Abort bit and leave the Transfer Byte clear which sends a STOP only.</li> </ul>
Transmit Acknowledge to slave-transmitter	Master-receive only	<ul style="list-style-type: none"> <li>As a master-receiver, the I<sup>2</sup>C Bus Interface Unit generates the clock for the acknowledge pulse. The I<sup>2</sup>C Bus Interface Unit is also responsible for driving the SDA line during the Ack cycle.</li> <li>If the next data byte is to be the last transaction, the i960 core processor sets the Ack/Nack Control bit for Nack generation.</li> <li>See <a href="#">Section 22.3.3</a>.</li> </ul>
Generate a Repeated START to chain I <sup>2</sup> C transactions	Master-transmit Master-receive	<ul style="list-style-type: none"> <li>If data chaining is desired, a repeated START condition is used instead of a STOP condition.</li> <li>This occurs after the last data byte of a transaction has been written to the bus.</li> <li>The i960 core processor writes the next target slave address and the R/W# bit to the IDBR, set the START bit, and set the Transfer Byte bit.</li> <li>See <a href="#">Section 22.2.3</a>.</li> </ul>
Generate a STOP	Master-transmit Master-receive	<ul style="list-style-type: none"> <li>Generated after the i960 core processor writes the last data byte on the bus.</li> <li>i960 core processor generates a STOP condition by setting the STOP bit in the ICR.</li> <li>See <a href="#">Section 22.2.3</a>.</li> </ul>

When the i960 RM/RN I/O processor needs to read data, the I<sup>2</sup>C 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 ([Figure 22-9](#)). It is also possible to have multiple transactions during an I<sup>2</sup>C operation such as transitioning from master-receive to master-transmit through a repeated start or Data Chaining ([Figure 22-10](#)). [Figure 22-11](#) shows the wave forms of SDA and SCL for a complete data transfer.

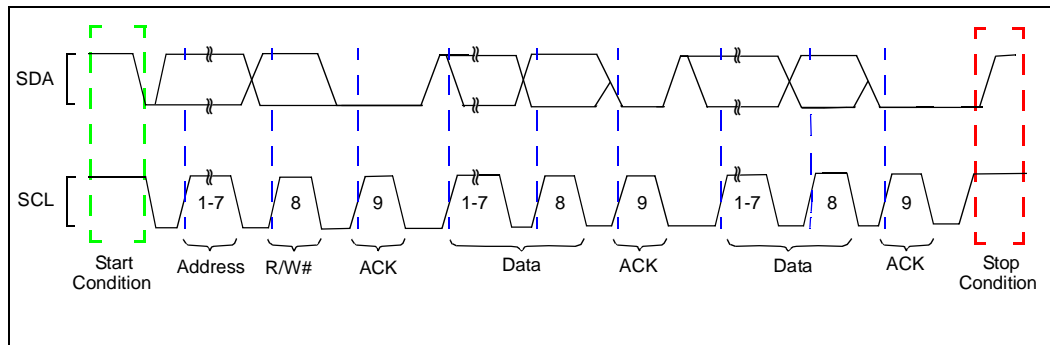
**Figure 22-9. Master-Receiver Read from Slave-Transmitter**



**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 I<sup>2</sup>C Bus Interface Unit's responsibilities as a slave device.

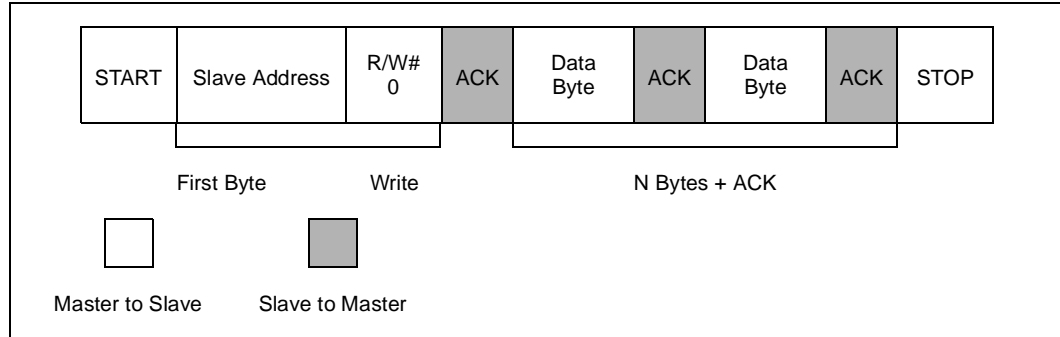
**Table 22-6. Slave Transactions**

I <sup>2</sup> C Slave Action	Mode of Operation	Definition
Slave-receive (default mode)	Slave-receive only	<ul style="list-style-type: none"> <li>I<sup>2</sup>C Bus Interface Unit monitors all slave address transactions.</li> <li>The I<sup>2</sup>C Bus Interface Unit Enable bit must be set.</li> <li>I<sup>2</sup>C 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 I<sup>2</sup>C Slave Address Register and the General Call address (00H). If there is a match, the I<sup>2</sup>C Bus Interface Unit sends an Ack.</li> <li>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 is set. See <a href="#">Section 22.3.7</a>.</li> <li>If the 8th bit of the first byte (R/W# bit) is low, the I<sup>2</sup>C Bus Interface Unit stays in slave-receive mode and the Slave Mode Operation bit is cleared. If the R/W# bit is high, the I<sup>2</sup>C Bus Interface Unit transitions to slave-transmit mode and the Slave Mode Operation bit is set.</li> </ul>
Setting the Slave Address Detected bit	Slave-receive Slave-transmit	<ul style="list-style-type: none"> <li>Indicates the interface has detected an I<sup>2</sup>C operation that addresses the i960 RM/RN I/O processor (this includes general call address). The i960 core processor can distinguish an ISAR match from a General Call by reading the General Call Address Detected bit.</li> <li>An interrupt is signalled (if enabled) after the matching slave address is received and acknowledged.</li> </ul>
Read one byte of I <sup>2</sup> C Data from the IDBR	Slave-receive only	<ul style="list-style-type: none"> <li>Data receive mode of I<sup>2</sup>C slave operation.</li> <li>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.</li> <li>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.</li> <li>i960 core processor reads 1 data byte from the IDBR. When the IDBR is read, the i960 core processor writes the desired Ack/Nack Control bit and set the Transfer Byte bit. This causes the I<sup>2</sup>C Bus Interface Unit to stop inserting wait states and let the master transmitter write the next piece of information.</li> </ul>
Transmit Acknowledge to master-transmitter	Slave-receive only	<ul style="list-style-type: none"> <li>As a slave-receiver, the I<sup>2</sup>C Bus Interface Unit is responsible for pulling the SDA line low to generate the Ack pulse during the high SCL period.</li> <li>The Ack/Nack Control bit controls the Ack data the I<sup>2</sup>C Bus Interface Unit drives. See <a href="#">Section 22.3.3</a>.</li> </ul>
Write one byte of I <sup>2</sup> C data to the IDBR	Slave-transmit only	<ul style="list-style-type: none"> <li>Data transmit mode of I<sup>2</sup>C slave operation.</li> <li>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.</li> <li>i960 core processor writes a data byte to the IDBR and set the Transfer Byte bit to initiate the transfer.</li> </ul>
Wait for Acknowledge from master-receiver	Slave-transmit only	<ul style="list-style-type: none"> <li>As a slave-transmitter, the I<sup>2</sup>C Bus Interface Unit is responsible for releasing the SDA line to allow the master-receiver to pull the line low for the Ack.</li> <li>See <a href="#">Section 22.3.3</a>.</li> </ul>

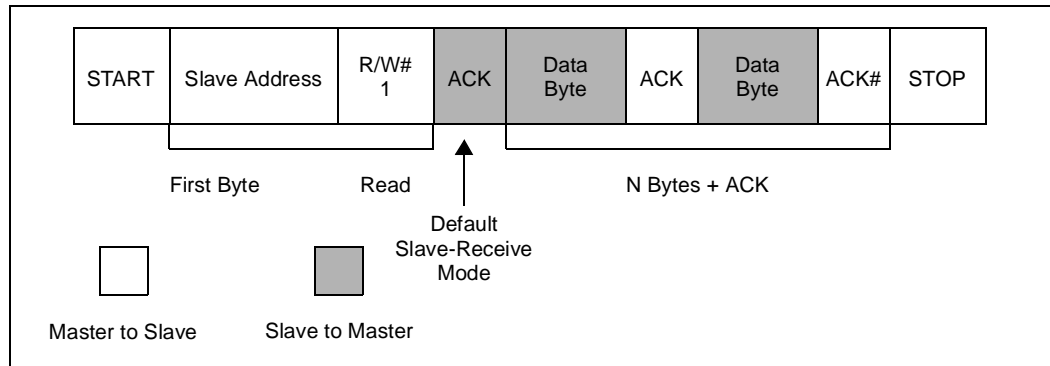


Figure 22-12 through Figure 22-14 are examples of I<sup>2</sup>C transactions. These show the relationships between master and slave devices.

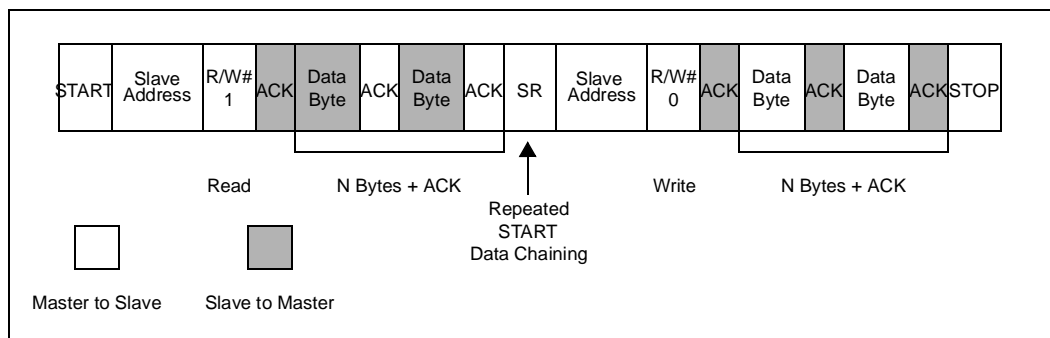
**Figure 22-12. Master-Transmitter Write to Slave-Receiver**



**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 I<sup>2</sup>C unit supports both sending and receiving general call address transfers on the I<sup>2</sup>C bus. When sending a general call message from the I<sup>2</sup>C unit, software must set the General Call Disable bit in the ICR to keep the I<sup>2</sup>C unit from responding as a slave. Failure to set this bit causes the I<sup>2</sup>C 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 shows the valid values and definitions when B=0.

When the i960 RM/RN I/O processor is acting as a slave, and the I<sup>2</sup>C unit receives a general call address and the ICR General Call Disable bit is clear the I<sup>2</sup>C unit:

- Sets the ISR general call address detected bit
- Sets the ISR slave address detected bit
- Interrupts (when enabled) the i960 RM/RN I/O processor

When the I<sup>2</sup>C unit receives a general call address and the ICR General Call Disable bit is set, the I<sup>2</sup>C unit ignores the general call address.

Figure 22-15. General Call Address

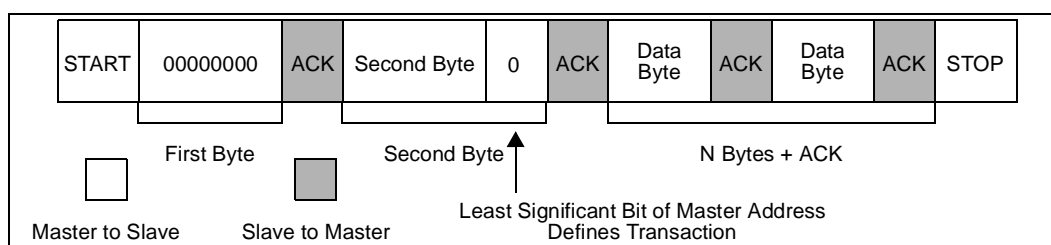


Table 22-7. General Call Address Second Byte Definitions

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 I<sup>2</sup>C Bus Interface Unit returns to its default reset condition with the exception of the ISAR. The i960 RM/RN I/O processor is responsible for ensuring this occurs, not the I<sup>2</sup>C 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 I<sup>2</sup>C Bus Interface Unit does not support this mode of operation.

I<sup>2</sup>C 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 bytes 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<sup>2</sup>C 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 I<sup>2</sup>C 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<sup>2</sup>C 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 sends 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)

### 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 sends 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)

## 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 to 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 sends 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)

## 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 is 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 I<sup>2</sup>C Bus Interface Unit has built-in glitch suppression logic. Glitches is suppressed according to: 4 \* internal bus clock period. For example, with a 66 MHz (15ns period) i960 RM/RN I/O processor clock, glitches of 60ns or less is suppressed. At 40 MHz (25ns period) clock, glitches of 100ns or less is suppressed. This is within the 50ns glitch suppression specification.

## 22.7 Reset Conditions

The I<sup>2</sup>C unit is reset with I\_RST#. Software is responsible for ensuring the I<sup>2</sup>C unit is not busy (ISR[3]) before asserting reset. Software is also responsible for ensuring the I<sup>2</sup>C bus is idle when the unit is enabled after reset. When directed to reset, the I<sup>2</sup>C 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 i960 RM/RN I/O processor I<sup>2</sup>C unit resets, the associated I<sup>2</sup>C MMRs remain intact. When resetting the I<sup>2</sup>C 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 I<sup>2</sup>C Bus Interface Unit. They are all located within the peripheral memory- mapped address space of the i960 RM/RN I/O processor. See [Section 12.3, "Programming the Logical Memory Attributes" on page 12-4](#) [Section 11.10, "Register Definitions" on page 11-11](#) for the register addresses

**Table 22-8. I<sup>2</sup>C Register Summary Table**

Section, Register Name, Page
Section 22.8.1, "I2C Control Register- ICR" on page 22-24
Section 22.8.2, "I2C Status Register- ISR" on page 22-27
Section 22.8.3, "I2C Slave Address Register- ISAR" on page 22-29
Section 22.8.4, "I2C Data Buffer Register- IDBR" on page 22-30
Section 22.8.5, "I2C Clock Count Register- ICCR" on page 22-31
Section 22.8.6, "I2C Bus Monitor Register- IBMR" on page 22-32

## 22.8.1 I<sup>2</sup>C Control Register- ICR

The i960 RM/RN I/O processor uses the bits in the I<sup>2</sup>C Control Register (ICR) to control the I<sup>2</sup>C unit.

**Table 22-9. I<sup>2</sup>C Control Register - ICR (Sheet 1 of 3)**

Bit	Default	Description
31:14	0000H	Reserved
14	0 <sub>2</sub>	<b>Unit Reset:</b> 1 = Reset the I <sup>2</sup> C unit only. 0 = No reset.
13	0 <sub>2</sub>	<b>Slave Address Detected Interrupt Enable:</b> 1 = Enables the I <sup>2</sup> C unit to interrupt the i960 RM/RN I/O processor upon detecting a slave address match or a general call address. 0 = Disable interrupt.
12	0 <sub>2</sub>	<b>Arbitration Loss Detected Interrupt Enable:</b> 1 = Enables the I <sup>2</sup> C unit to interrupt the i960 RM/RN I/O processor upon losing arbitration while in master mode. 0 = Disable interrupt.
11	0 <sub>2</sub>	<b>Slave STOP Detected Interrupt Enable:</b> 1 = Enables the I <sup>2</sup> C unit to interrupt the i960 RM/RN I/O processor when it detects a STOP condition while in slave mode. 0 = Disable interrupt.
10	0 <sub>2</sub>	<b>Bus Error Interrupt Enable:</b> 1 = Enables the I <sup>2</sup> C unit to interrupt the i960 RM/RN I/O processor for the following I <sup>2</sup> C bus errors: As a master transmitter, no Ack was detected after a byte was sent. As a slave receiver, the I <sup>2</sup> C unit generated a Nack pulse. <b>Note:</b> Software is responsible for guaranteeing that misplaced START and STOP conditions do not occur. See <a href="#">Section 22.6</a> . 0 = Disable interrupt.
09	0 <sub>2</sub>	<b>IDBR Receive Full Interrupt Enable:</b> 1 = Enables the I <sup>2</sup> C unit to interrupt the i960 RM/RN I/O processor when the IDBR has received a data byte from the I <sup>2</sup> C bus. 0 = Disable interrupt.
08	0 <sub>2</sub>	<b>IDBR Transmit Empty Interrupt Enable:</b> 1 = Enables the I <sup>2</sup> C unit to interrupt the i960 RM/RN I/O processor after transmitting a byte onto the I <sup>2</sup> C bus. 0 = Disable interrupt.



**Table 22-9. I<sup>2</sup>C Control Register - ICR (Sheet 2 of 3)**

Bit	Default	Description
07	0 <sub>2</sub>	<p><b>General Call Disable:</b>            1 = Disables I<sup>2</sup>C unit response to general call messages as a slave.            0 = Enables the I<sup>2</sup>C unit to respond to general call messages.            This bit must be set when sending a master mode general call message from the I<sup>2</sup>C unit.</p>
06	0 <sub>2</sub>	<p><b>I<sup>2</sup>C Unit Enable:</b>            1 = Enables the I<sup>2</sup>C unit (defaults to slave-receive mode).            0 = Disables the unit and does not master any transactions or respond to any slave transactions.            Software must guarantee the I<sup>2</sup>C bus is idle before setting this bit.</p>
05	0 <sub>2</sub>	<p><b>SCL Enable:</b>            1 = Enables the I<sup>2</sup>C clock output for master mode operation. The ICCR (Section 22.8.5) must be programmed with a valid value before setting this bit.            0 = Disables the I<sup>2</sup>C unit from driving the SCL line.</p>
04	0 <sub>2</sub>	<p><b>Master Abort:</b> used by the I<sup>2</sup>C unit when in master mode to generate a STOP without transmitting another data byte.            1 = The I<sup>2</sup>C unit sends STOP without data transmission.            0 = The I<sup>2</sup>C unit transmits STOP using the STOP ICR bit only.            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 i960 RM/RN 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.</p>
03	0 <sub>2</sub>	<p><b>Transfer Byte:</b> used to send/receive a byte on the I<sup>2</sup>C bus.            1 = send/receive a byte.            0 = cleared by I<sup>2</sup>C unit when the byte is sent/received.            The i960 RM/RN 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 I<sup>2</sup>C unit holds the SCL line low (inserting wait states) until the Transfer Byte bit is set.</p>
02	0 <sub>2</sub>	<p><b>Ack/Nack Control:</b> defines the type of Ack pulse sent by the I<sup>2</sup>C unit when in master receive mode.            1 = The I<sup>2</sup>C unit sends a negative Ack (Nack) after receiving a data byte.            0 = The I<sup>2</sup>C unit sends an Ack pulse after receiving a data byte.            The I<sup>2</sup>C 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.</p>

**Table 22-9. I<sup>2</sup>C Control Register - ICR (Sheet 3 of 3)**

<p>IOP Attributes</p> <p>PCI Attributes</p>																																	
		<p>80960 Core Local Bus Address 1680H</p>																<p>Attribute Legend:                  RW = Read/Write                  RV = Reserved                  PR = Preserved                  RS = Read/Set                  RC = Read Clear                  RO = Read Only                  NA = Not Accessible</p>															
Bit	Default	Description																															
01	0 <sub>2</sub>	<p><b>STOP:</b> used to initiate a STOP condition after transferring the next data byte on the I<sup>2</sup>C bus when in master mode. In master-receive mode, the Ack/Nack control bit must be set in conjunction with this bit. See <a href="#">Section 22.2.3.3, "STOP Condition" on page 22-6</a> for more details on the STOP state.</p> <p>1 = Send a STOP 0 = Do not send a STOP</p>																															
00	0 <sub>2</sub>	<p><b>START:</b> used to initiate a START condition to the I<sup>2</sup>C unit when in master mode. See <a href="#">Section 22.2.3.1, "START Condition" on page 22-6</a> for more details on the START state.</p> <p>1 = Send a START 0 = Do not send a START</p>																															

## 22.8.2 I<sup>2</sup>C Status Register- ISR

I<sup>2</sup>C interrupts are signalled through XINT7# and the XINT7 Interrupt Status Register (X7ISR), which shows the pending XINT7 interrupts (Chapter 22, “I<sup>2</sup>C Bus Interface Unit”). XINT7# is set by the I<sup>2</sup>C Interrupt Status Register (ISR). Software uses the ISR bits to check the status of the I<sup>2</sup>C unit and bus. ISR bits (bits 9-5) are updated after the Ack/Nack bit has completed on the I<sup>2</sup>C bus.

The ISR is also used to clear interrupts signalled from the I<sup>2</sup>C 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. I<sup>2</sup>C Status Register - ISR (Sheet 1 of 2)

Bit	Default	Description
31:11	000000H	Reserved
10	0 <sub>2</sub>	<p><b>Bus Error Detected:</b>                      1 = The I<sup>2</sup>C 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 I<sup>2</sup>C unit generates a Nack pulse.  <b>Note:</b> When an error occurs, I<sup>2</sup>C bus transactions continue. Software must guarantee that misplaced START and STOP conditions do not occur. See <a href="#">Section 22.3.4, “Arbitration”</a> on page 22-11.                      0 = No error detected.</p>
09	0 <sub>2</sub>	<p><b>Slave Address Detected:</b>                      1 = I<sup>2</sup>C unit detected a 7-bit address that matches the general call address or ISAR. An interrupt is signalled when enabled in the ICR.                      0 = No slave address detected.</p>
08	0 <sub>2</sub>	<p><b>General Call Address Detected:</b>                      1 = I<sup>2</sup>C unit received a general call address.                      0 = No general call address received.</p>

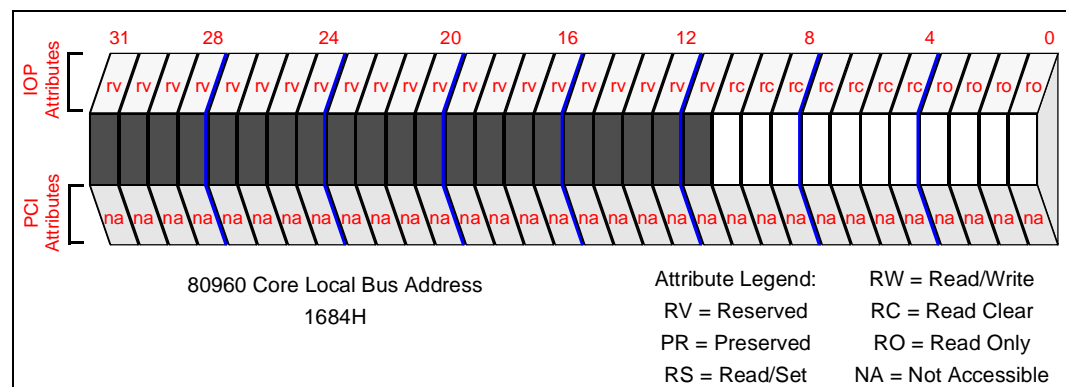


Table 22-10. I<sup>2</sup>C Status Register - ISR (Sheet 2 of 2)

Bit	Default	Description
07	0 <sub>2</sub>	<p><b>IDBR Receive Full:</b>                      1 = The IDBR register received a new data byte from the I<sup>2</sup>C bus. An interrupt is signalled when enabled in the ICR.                      0 = The IDBR has not received a new data byte or the I<sup>2</sup>C unit is idle.</p>
06	0 <sub>2</sub>	<p><b>IDBR Transmit Empty:</b>                      1 = The I<sup>2</sup>C unit has finished transmitting a data byte on the I<sup>2</sup>C bus. An interrupt is signalled when enabled in the ICR.                      0 = The data byte is still being transmitted.</p>
05	0 <sub>2</sub>	<p><b>Arbitration Loss Detected:</b> used during multi-master operation.                      1 = Set when the I<sup>2</sup>C unit loses arbitration.                      0 = Cleared when arbitration is won or never took place.</p>
04	0 <sub>2</sub>	<p><b>Slave STOP Detected:</b>                      1 = Set when the I<sup>2</sup>C unit detects a STOP while in slave-receive or slave-transmit mode.                      0 = No STOP detected.</p>
03	0 <sub>2</sub>	<p><b>I<sup>2</sup>C Bus Busy:</b>                      1 = Set when the I<sup>2</sup>C bus is busy but the i960 RM/RN I/O processor's I<sup>2</sup>C unit is not involved in the transaction.                      0 = I<sup>2</sup>C bus is idle or the I<sup>2</sup>C unit is using the bus (i.e., unit busy).</p>
02	0 <sub>2</sub>	<p><b>Unit Busy:</b>                      1 = Set when the i960 RM/RN I/O processor's I<sup>2</sup>C unit is busy. This is defined as the time between the first START and STOP.                      0 = I<sup>2</sup>C unit not busy.</p>
01	0 <sub>2</sub>	<p><b>Ack/Nack Status:</b>                      1 = The I<sup>2</sup>C unit received or sent a Nack.                      0 = The I<sup>2</sup>C unit received or sent an Ack on the bus.                      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.</p>
00	0 <sub>2</sub>	<p><b>Read/Write Mode:</b>                      1 = The I<sup>2</sup>C unit is in master-receive or slave-transmit mode.                      0 = The I<sup>2</sup>C unit is in master-transmit or slave-receive mode.                      This is the R/W# bit of the slave address. It is automatically cleared by hardware after a stop state.</p>

### 22.8.3 I<sup>2</sup>C Slave Address Register- ISAR

The I<sup>2</sup>C Slave Address Register (Table 22-11) defines the I<sup>2</sup>C unit's 7-bit slave address to which the i960 RM/RN I/O processor responds when in slave-receive mode. This register is written by the i960 RM/RN I/O processor before enabling I<sup>2</sup>C operations. The register is fully programmable (no address is assigned to the I<sup>2</sup>C unit) so it can be set to a value other than those of hard-wired I<sup>2</sup>C slave peripherals that might exist in the system. The ISAR is not affected by the i960 RM/RN I/O processor being reset. The ISAR register default value is 0000000<sub>2</sub>.

Table 22-11. I<sup>2</sup>C Slave Address Register - ISAR

Bit	Default	Description
31:07	000000H	Reserved
06:00	00H	<b>I<sup>2</sup>C Slave Address:</b> The 7-bit address to which the I <sup>2</sup> C unit responds when in slave-receive mode.

## 22.8.4 I<sup>2</sup>C Data Buffer Register- IDBR

The I<sup>2</sup>C Data Buffer Register is used by the i960 RM/RN I/O processor to transmit and receive data from the I<sup>2</sup>C bus. The accesses the IDBR by the i960 RM/RN I/O processor on one side and by the I<sup>2</sup>C shift register on the other. Data coming into the I<sup>2</sup>C Bus Interface Unit is received into the IDBR after a full byte has been received and acknowledged. Data going out of the I<sup>2</sup>C Bus Interface Unit is written to the IDBR by the i960 RM/RN I/O processor core and sent to the serial bus.

When the I<sup>2</sup>C Bus Interface Unit is in transmit mode (master or slave), the i960 RM/RN 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 is signalled (if enabled) when a byte has been transferred on the I<sup>2</sup>C bus and the acknowledge cycle is complete. If the IDBR is not written by the i960 RM/RN I/O processor (and a STOP condition was not in place) before the I<sup>2</sup>C bus is ready to transfer the next byte packet, the I<sup>2</sup>C Bus Interface Unit inserts wait states until the i960 core processor writes the IDBR and sets the Transfer Byte bit.

When the I<sup>2</sup>C Bus Interface Unit is in receive mode (master or slave), the processor reads 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 I<sup>2</sup>C Bus Interface Unit inserts wait states until the IDBR has been read. Refer to [Section 22.3.3, “I<sup>2</sup>C Acknowledge”](#) on page 22-10 for acknowledge pulse information in receiver mode. After the i960 RM/RN 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 I<sup>2</sup>C Bus. The IDBR register is 00H after reset.

**Table 22-12. I<sup>2</sup>C Data Buffer Register - IDBR**

		80960 Core Local Bus Address 168CH		Attribute Legend: RV = Reserved      RC = Read Clear PR = Preserved      RO = Read Only RS = Read/Set      NA = Not Accessible	
		<b>Bit</b>	<b>Default</b>	<b>Description</b>	
31:08	000000H	Reserved			
07:00	00H	I <sup>2</sup> C Data Buffer: Buffer for I <sup>2</sup> C bus send/receive data.			

## 22.8.5 I<sup>2</sup>C Clock Count Register- ICCR

The I<sup>2</sup>C Clock Count Register (ICCR) defines the multiplier used to generate the I<sup>2</sup>C 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 I<sup>2</sup>C clock output during I<sup>2</sup>C master operations.

Changing this register while the SCL enable bit is set results in undefined behavior.

**Table 22-13. I<sup>2</sup>C Clock Count Register - ICCR**

Bit	Default	Description
31:10	000000H	Reserved
09:00	0	<b>I<sup>2</sup>C Clock Count:</b> 9 bit count value used to generate an I <sup>2</sup> C clock from the i960 RM/RN I/O processor internal bus clock.

80960 Core Local Bus Address 1690H	<b>Attribute Legend:</b> RW = Read/Write RV = Reserved PR = Preserved RS = Read/Set RC = Read Clear RO = Read Only NA = Not Accessible
---------------------------------------	---

## 22.8.6 I<sup>2</sup>C Bus Monitor Register- IBMR

The I<sup>2</sup>C 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 I<sup>2</sup>C bus is hung and the I<sup>2</sup>C unit must be reset.

**Table 22-14. I<sup>2</sup>C Bus Monitor Register - IBMR**

IOP Attributes		31	28	24	20	16	12	8	4	0	
		rv	rv	rv	rv	rv	rv	rv	rv	rv	rv
PCI Attributes		na	na	na	na	na	na	na	na	na	
		na	na	na	na	na	na	na	na	na	na
		80960 Core Local Bus Address					Attribute Legend:				
		1694H					RW = Read/Write				
							RV = Reserved				
							RC = Read Clear				
							PR = Preserved				
							RO = Read Only				
							RS = Read/Set				
							NA = Not Accessible				
Bit	Default	Description									
31:02	0	Reserved									
01	1	SCL <b>Status</b> : This bit continuously reflects the value of the SCL pin.									
00	1	SDA <b>Status</b> : This bit continuously reflects the value of the SDA pin.									



This chapter describes the i960<sup>®</sup> RM/RN 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.

## 23.1 On-Circuit Emulation (ONCE)

On-circuit emulation aids board-level testing. This feature allows a mounted i960 RM/RN 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 pullup 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 23.1.2, “ONCE Mode and Boundary-Scan \(JTAG\) are Incompatible”](#) on page 23-2.

### 23.1.1 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 *80960RM I/O Processor Data Sheet* and the *80960RN I/O Processor Data Sheet* for specific timing of the ONCE# pin and the characteristics of the on-circuit emulation mode.

## 23.1.2 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 i960 RM/RN I/O processor signals while JTAG is active.

Since the i960 RM/RN 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.

### 23.1.2.1 DEN# Alternatives

To use an ICE with your 80960RM/RN design, alternatives to DEN# are:

- Ground the OE# pin of the transceiver
- Re-create a DEN# signal with the circuit shown below

## 23.2 Boundary-Scan (JTAG)

The i960 RM/RN 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).

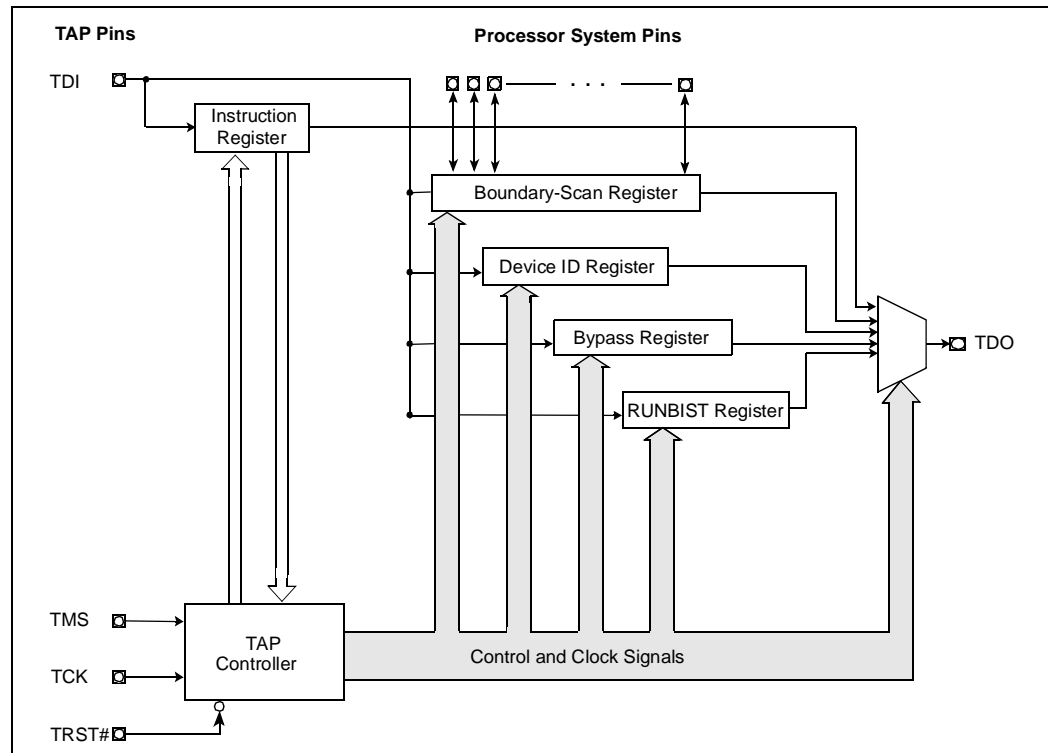
**Note:** Do not use ONCE mode with boundary-scan (JTAG). See [Section 23.1.2, “ONCE Mode and Boundary-Scan \(JTAG\) are Incompatible”](#) on page 23-2.

### 23.2.1 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 23-1](#) illustrates how these pieces fit together to form the JTAG unit.

Figure 23-1. Test Access Port Block Diagram



## 23.2.2 TAP Pins

The i960 RM/RN 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 23-1](#). The TAP pins provide access to the instruction register and the test data registers.

Table 23-1. TAP Controller Pin Definitions

Pin	Type	Definition
TCK	Input	<b>Test Clock</b> 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	<b>Test Mode</b> 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	<b>Test Data Input</b> 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	<b>Test Data Output</b> 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	<b>Test Reset</b> 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 <i>Test-Logic-Reset</i> (initial) state. TRST# is pulled high internally when not driven. <b>Note:</b> 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.

### 23.2.3 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 23-2). 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<sub>2</sub>) is captured. During Shift\_IR, when a new instruction is shifted in through TDI, the value 0001<sub>2</sub> 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 23-4 for an example of loading the instruction register.

#### 23.2.3.1 Boundary-Scan Instruction Set

The i960 RM/RN 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 23-2 lists the i960 RM/RN I/O processor's boundary-scan instruction codes. Those codes listed as “not used” or “private” should not be used.

**Table 23-2. Boundary-Scan Instruction Set**

Instruction Code	Instruction Name	Instruction Code	Instruction Name
0000 <sub>2</sub>	<b>extest</b>	1000 <sub>2</sub>	<b>highz</b>
0001 <sub>2</sub>	<b>sample/preload</b>	1001 <sub>2</sub>	not used
0010 <sub>2</sub>	<b>idcode</b>	1010 <sub>2</sub>	not used
0011 <sub>2</sub>	<b>not used</b>	1011 <sub>2</sub>	private
0100 <sub>2</sub>	<b>clamp</b>	1100 <sub>2</sub>	private
0101 <sub>2</sub>	not used	1101 <sub>2</sub>	not used
0110 <sub>2</sub>	not used	1110 <sub>2</sub>	not used
0111 <sub>2</sub>	<b>runbist</b>	1111 <sub>2</sub>	<b>bypass</b>

Table 23-3. IEEE Instructions

Instruction / Requisite	Opcode	Description
<b>extest</b> IEEE 1149.1 Required	0000 <sub>2</sub>	<b>extest</b> initiates testing of external circuitry, typically board-level interconnects and off chip circuitry. <b>extest</b> connects the boundary-scan register between TDI and TDO in the Shift_DR state only. When <b>extest</b> 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 <b>extest</b> 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.
<b>sample/preload</b> IEEE 1149.1 Required	0001 <sub>2</sub>	<b>sample/preload</b> performs two functions: <ul style="list-style-type: none"> <li>When the TAP controller is in the Capture-DR state, the <b>sample</b> 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.</li> <li>When the TAP controller is in the Update-DR state, the <b>preload</b> 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 <b>extest</b> instruction.</li> </ul>
<b>idcode</b> IEEE 1149.1 Optional	0010 <sub>2</sub>	<b>idcode</b> 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, <b>idcode</b> 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. <b>NOTE:</b> The device identification register is not altered by data being shifted in on TDI.
<b>runbist</b> i960 RM/RN I/O processor Optional	0111 <sub>2</sub>	<b>runbist</b> 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 <sub>CC</sub> and CLKIN must be met and RESET# must be de-asserted prior to executing <b>runbist</b> . After loading <b>runbist</b> 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. <b>runbist</b> 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.
<b>bypass</b> IEEE 1149.1 Required	1111 <sub>2</sub>	<b>bypass</b> 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 <sub>2</sub> 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.
<b>highz</b>	1000 <sub>2</sub>	Executing <b>highz</b> 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 <b>bypass</b> instruction. <b>highz</b> provides an alternate method of entering ONCE mode.
<b>clamp</b>	0100 <sub>2</sub>	<b>clamp</b> instruction allows the state of the signals driven from the 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 <b>clamp</b> instruction is selected.

## 23.2.4 TAP Test Data Registers

The i960 RM/RN 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 23-5](#) for an example of loading the data register.

### 23.2.4.1 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 *80960RM I/O Processor Data Sheet* and the *80960RN 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.

### 23.2.4.2 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.

### 23.2.4.3 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 [Section 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.

### 23.2.4.4 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 23-4](#) shows the bit order of the i960 RM/RN 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 i960 RM/RN I/O processor's pins and on-chip system logic. The V<sub>CC</sub>, V<sub>SS</sub> 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 23-4. i960<sup>®</sup> RM/RN 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),	"&
"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),	"&

Table 23-4. i960® RM/RN I/O Processor Boundary Scan Register Bit Order (Sheet 2 of 9)

"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),	"&
"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),	"	&		



**Table 23-4. i960<sup>®</sup> RM/RN I/O Processor Boundary Scan Register Bit Order (Sheet 3 of 9)**

"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),	"&
"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),	"&

**Table 23-4. i960<sup>®</sup> RM/RN I/O Processor Boundary Scan Register Bit Order (Sheet 4 of 9)**

"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),	"&
"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),	" &			

**Table 23-4. i960<sup>®</sup> RM/RN I/O Processor Boundary Scan Register Bit Order (Sheet 5 of 9)**

"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),	"&
"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),	"&

**Table 23-4. i960<sup>®</sup> RM/RN I/O Processor Boundary Scan Register Bit Order (Sheet 6 of 9)**

"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),	"&
"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),	"&

**Table 23-4. i960<sup>®</sup> RM/RN I/O Processor Boundary Scan Register Bit Order (Sheet 7 of 9)**

"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),	"&			
"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),	"&

**Table 23-4. i960<sup>®</sup> RM/RN I/O Processor Boundary Scan Register Bit Order (Sheet 8 of 9)**

"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,	rak,	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,	ldinitz,	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),	"&
"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),	"&

**Table 23-4. i960<sup>®</sup> RM/RN I/O Processor Boundary Scan Register Bit Order (Sheet 9 of 9)**

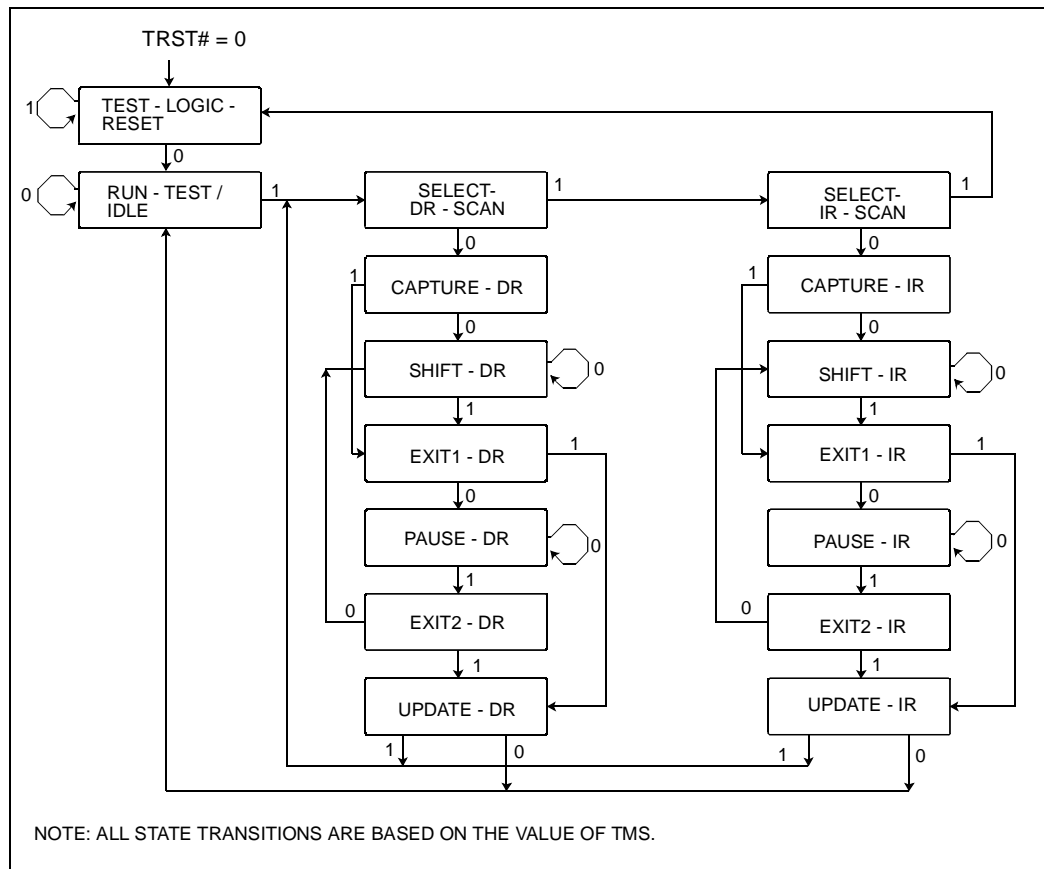
"386	(CBSC_1,	dq(11),	bidir,	X,	396,	1,	Z),	"&
"387	(CBSC_1,	dq(42),	bidir,	X,	393,	1,	Z),	"&
"388	(BC_1,	*	control,	1),	" &			
"389	(BC_1,	*	control,	1),	" &			
"390	(BC_1,	*	control,	1),	" &			
"391	(BC_1,	*	control,	1),	" &			
"392	(BC_1,	*	control,	1),	" &			
"393	(BC_1,	*	control,	1),	" &			
"394	(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)";				

## 23.2.5 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 23-2 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 23-2. TAP Controller State Diagram





### 23.2.5.1 Test Logic Reset State

In this state, test logic is disabled to allow normal operation of the i960 RM/RN 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<sub>2</sub>) 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.

### 23.2.5.2 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.

### 23.2.5.3 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 23.2.5.10, “Select-IR Scan State” on page 23-18](#). The instruction register does not change while the TAP controller is in this state.

### 23.2.5.4 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.

### 23.2.5.5 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.

### 23.2.5.6 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.

### 23.2.5.7 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.

### 23.2.5.8 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.

### 23.2.5.9 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.

### 23.2.5.10 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.

### 23.2.5.11 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.

### 23.2.5.12 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.

### 23.2.5.13 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.

### 23.2.5.14 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.

### 23.2.5.15 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.

### 23.2.5.16 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.

## 23.2.6 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 23-3](#) for an illustration of the waveforms involved in this example. The steps are:

1. Load the **sample/preload** instruction into the instruction register:
  - 1.1. Use TMS to select the Shift-IR state. While in the Shift-IR state, shift in the new instruction, least significant byte first.
  - 1.2. 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).
  - 1.3. Enter the Update-IR state to make the instruction take effect.
2. Capture pin data and shift the data out through the TDO pin:
  - 2.1. Use TMS to select the Select-DR-Scan state.
  - 2.2. Transition the TAP controller to the Capture-DR state to latch pin data in the boundary-scan register cells.
  - 2.3. 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.
  - 2.4. 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.
  - 2.5. 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 23-3. Example Showing Typical JTAG Operations

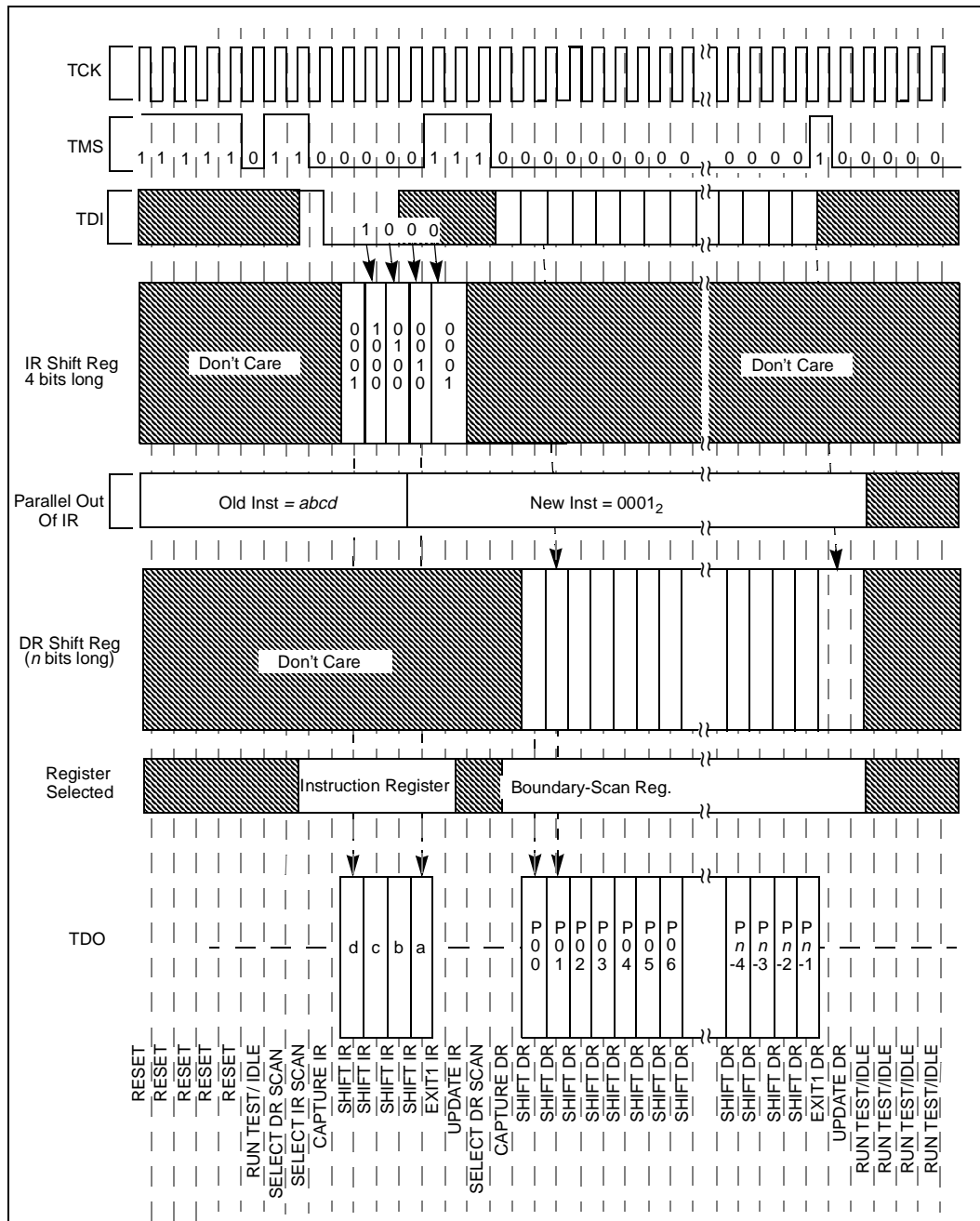


Figure 23-4. Timing Diagram Illustrating the Loading of Instruction Register

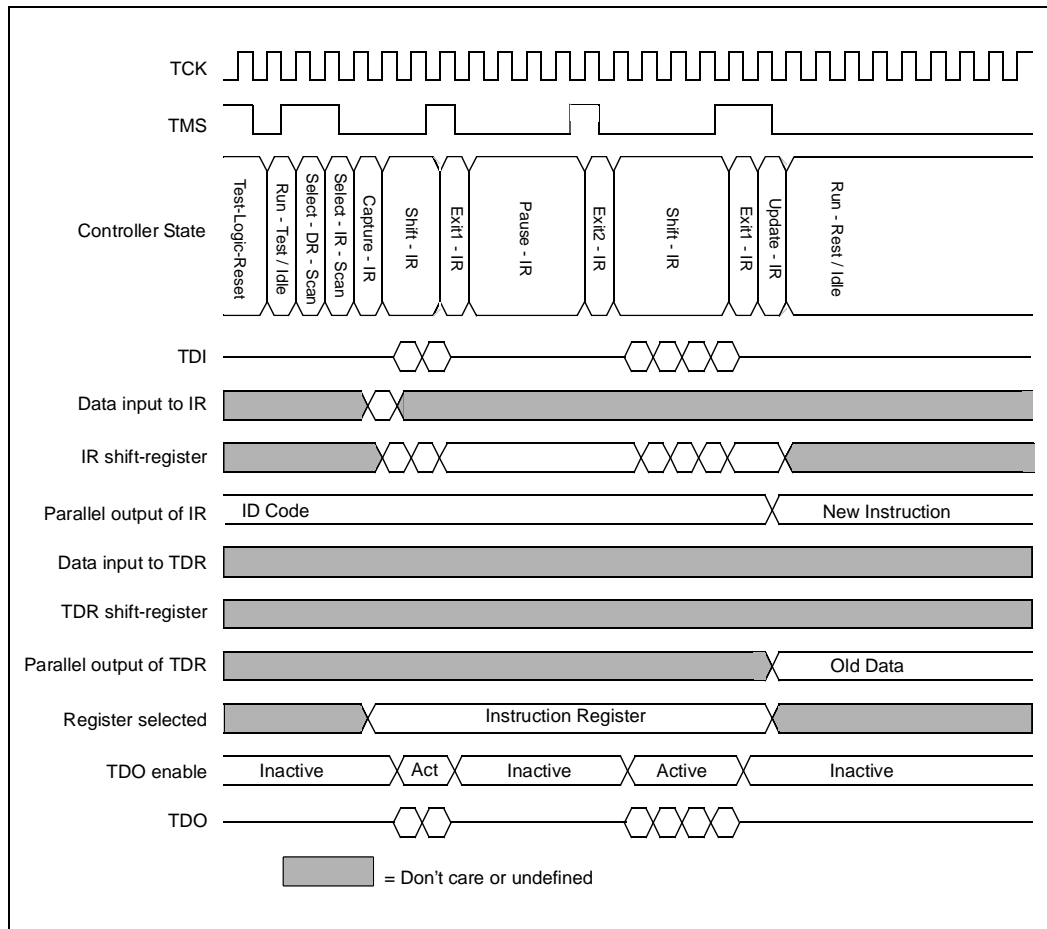
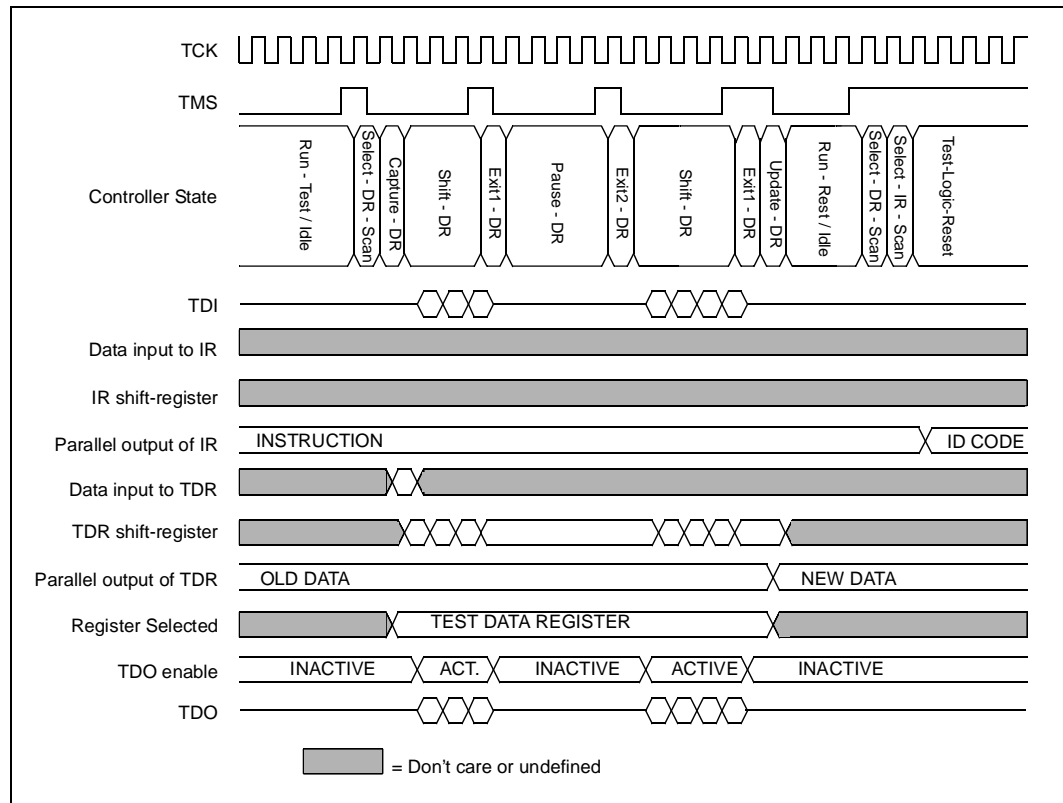


Figure 23-5. Timing Diagram Illustrating the Loading of Data Register





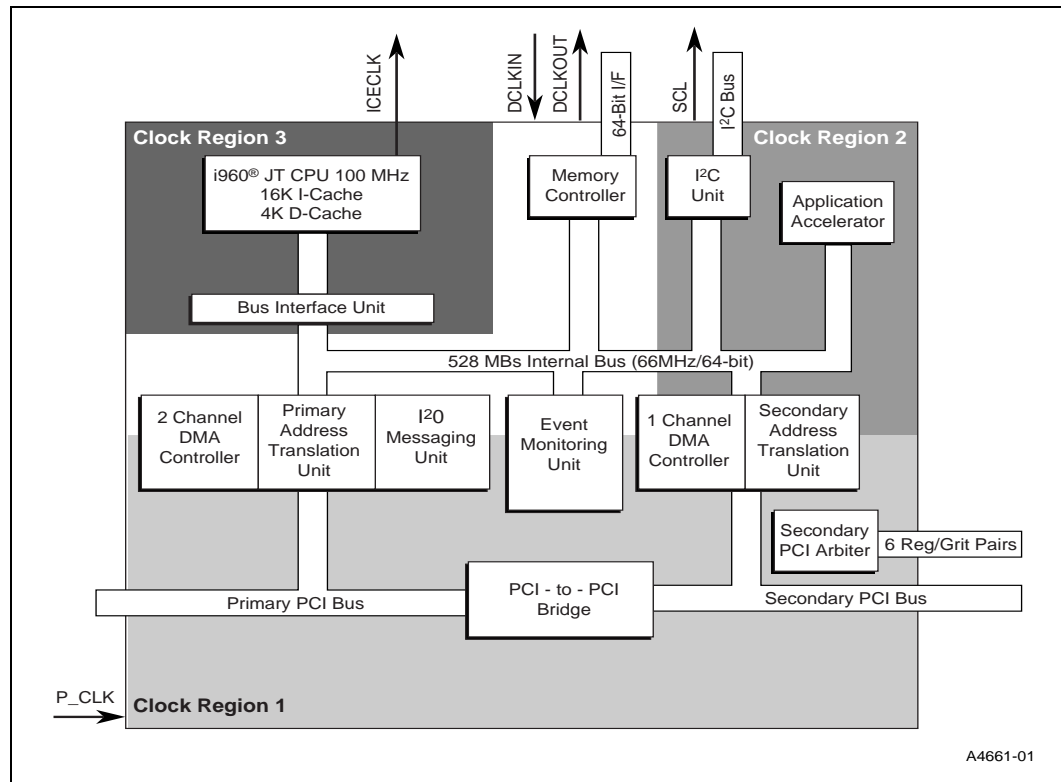


This chapter describes the clocking and reset function. The intent of this chapter is to elaborate and clarify descriptions of the clocking and reset mechanisms.

## 24.1 Clocking Overview

The i960<sup>®</sup> RM/RN I/O Processor contains various clocking boundaries internally. The clocks for all of the units within the i960 RM/RN 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 i960 RM/RN I/O processor, shown in [Figure 24-1](#), highlights the four clocking regions.

**Figure 24-1. Clocking Regions Diagram**



Within each of the clocking regions identified in [Figure 24-1](#), exists various clock requirements for the i960 RM/RN I/O processor units and for the output clocks pins provided for the external subsystem.

### 24.1.1 Clocking Theory of Operation

Each region within the i960 RM/RN I/O processor contains different clocking requirements. These requirements are summarized in the following sections.

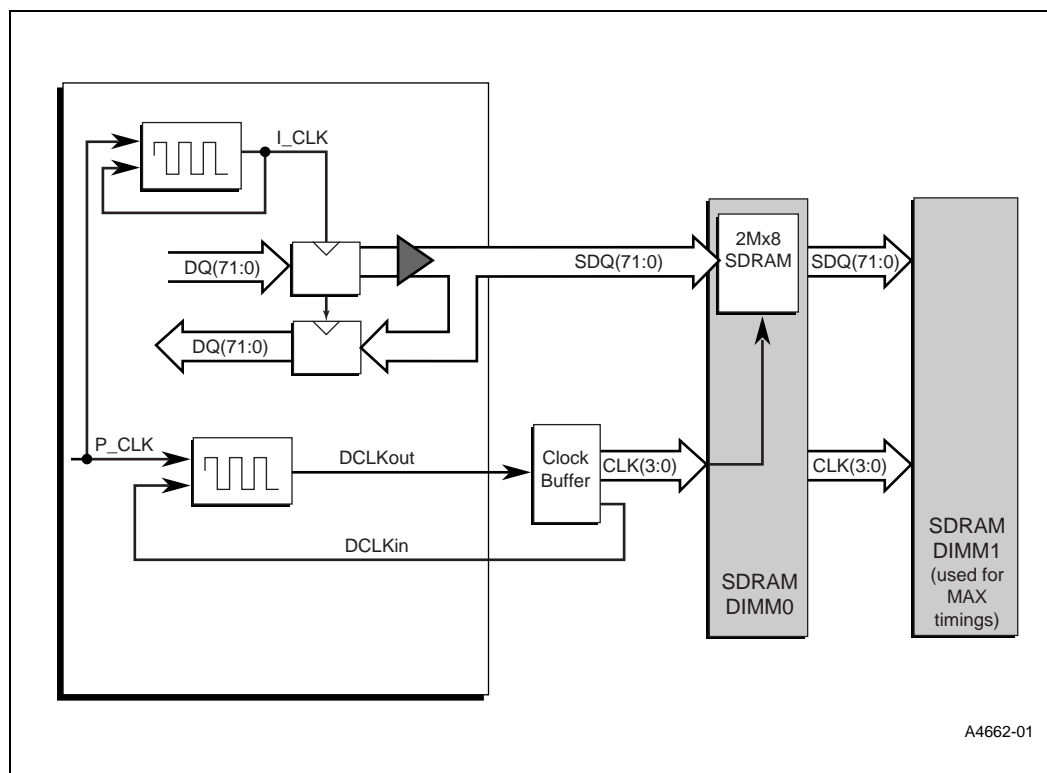
## 24.1.2 Clocking Region 1

Region 1 contains the main input clock for providing the i960 RM/RN 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 i960 RM/RN 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.

## 24.1.3 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 i960 RM/RN 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 24-2](#) for a diagram that describes the SDRAM clocking requirements.

**Figure 24-2. SDRAM Clocking Diagram**



Region 2 also contains an output clock used for the I<sup>2</sup>C bus interface ([Chapter 22, “I<sup>2</sup>C Bus Interface Unit”](#)). The output clock frequency for I<sup>2</sup>C operation is 100KHz or 400KHz. This clock is generated from internal bus clock. In order to use the I<sup>2</sup>C interface, a clock divider value must be written into the I<sup>2</sup>C Clock Count Register.

### 24.1.4 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.

### 24.1.5 Clocking Region Summary

Table 24-1 summarizes all of the input clock pins, output clock pins, and clock strapping option pins used in the i960 RM/RN I/O processor.

**Table 24-1. 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 <sup>2</sup> C Output Clock

Table 24-2 summarizes all of the clocks generated to the three regions within the i960 RM/RN I/O processor.

**Table 24-2. Clock Region Summary**

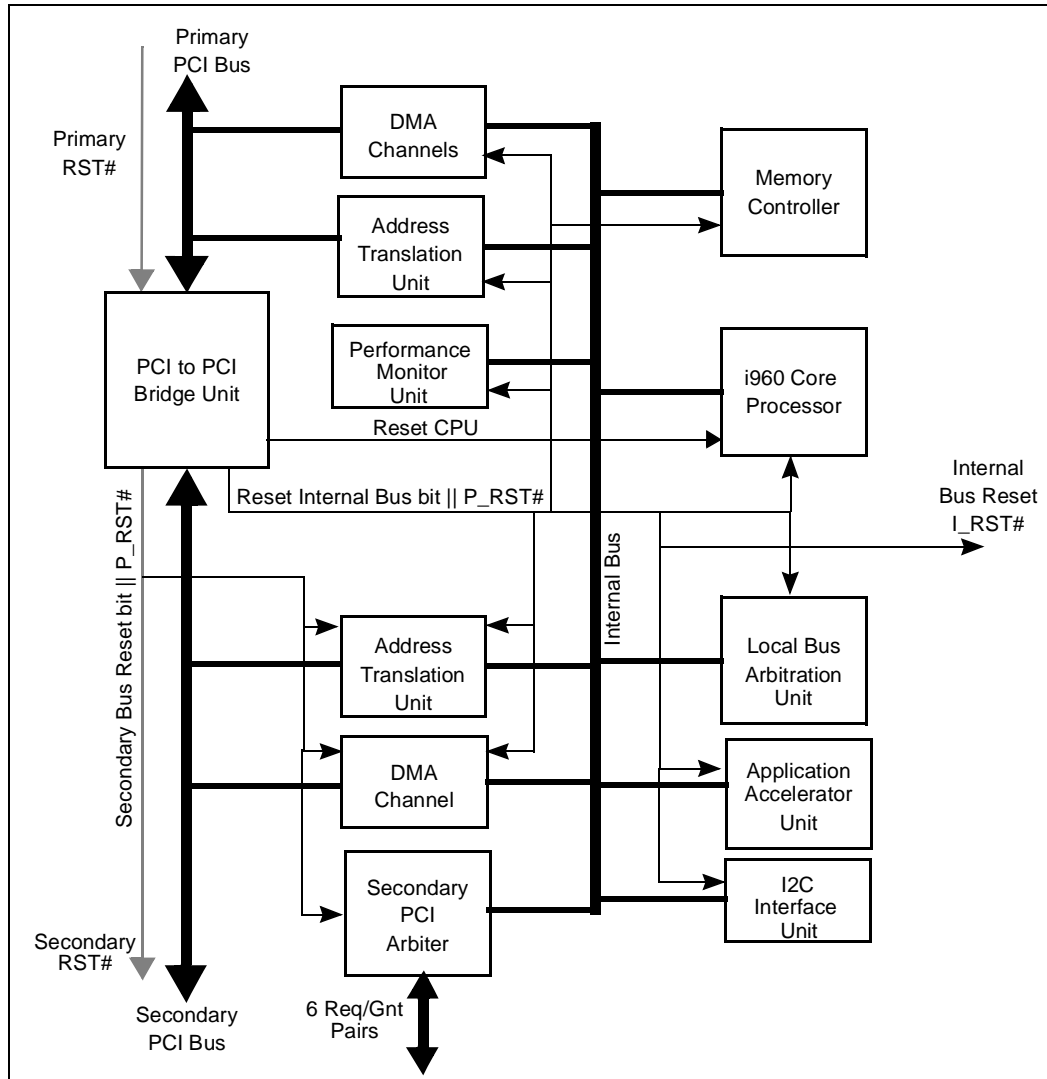
Input Clock	Region/Clock
P_CLK= 33 MHz	Region 1: 1x P_CLK
	Region 2: 2x P_CLK
	Region 3: 3x P_CLK

## 24.2 Reset Overview

There are three ways to reset the i960 RM/RN 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 i960 RM/RN I/O processor is placed in a reset state. In addition to the primary PCI reset pin, the i960 RM/RN I/O processor provides software control of units within the i960 RM/RN I/O processor and the secondary PCI interface.

Figure 24-3 shows the logical block diagram of the reset conditions.

Figure 24-3. 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.

## 24.2.1 Primary PCI Reset

When the primary PCI bus reset signal P\_RST# is asserted, the i960 RM/RN I/O processor:

- asserts the secondary PCI bus reset signal S\_RST#
- resets the 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 24.3](#)
- latches P\_REQ64# to determine the primary PCI bus interface width
- asserts the L\_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 i960 RM/RN 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 i960 RM/RN I/O processor samples a series of strapping pins to set configuration modes (refer to [Section 24.3, “Reset Strapping Options” on page 24-7](#)). One strap which alters the behavior of the i960 RM/RN 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 i960 RM/RN 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 i960 RM/RN I/O processor samples the P\_REQ64# signal to determine if the i960 RM/RN 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 i960 RM/RN I/O processor is connected to a 64-bit or 32-bit PCI bus.

## 24.2.2 Secondary PCI Reset

When the secondary PCI bus reset signal S\_RST# is asserted, the i960RM/RN 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 i960 RM/RN 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 i960 RM/RN 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 i960 RM/RN 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, “i960® RM/RN 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.

## 24.2.3 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<sup>2</sup>C 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.

## 24.3 Reset Strapping Options

There are many initialization modes that can be selected when the processor is reset. [Table 24-3](#) shows the configuration modes. All of the configuration modes defined are determined on the rising edge of P\_RST#.

**Table 24-3. Configuration Modes**

NAME	DESCRIPTION
RAD[4]/STEST	<b>SELF TEST</b> enables or disables the processor's internal self-test feature at initialization. STEST is examined at the end of P_RST#.
RAD[3]/RETRY	<b>RETRY</b> is sampled at the end of P_RST# to determine if the Primary PCI interface will be disabled.
RAD[6]/RST_MODE#	<b>RESET MODE</b> is sampled at the end of P_RST# to determine if the i960 RM/RN I/O processor is to be held in reset.
RAD[1]/32BITPCI_EN#	<b>32-BIT Secondary PCI Enable</b> 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#	<b>32-BIT MemoryEnable</b> 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#	<b>ONCE MODE:</b> is sampled during reset to stop all clocks and float all output pins of the 80960 core processor except the TDO pin.





# Machine-Level Instruction Formats A

This appendix describes the encoding format for instructions used by the i960<sup>®</sup> 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 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**

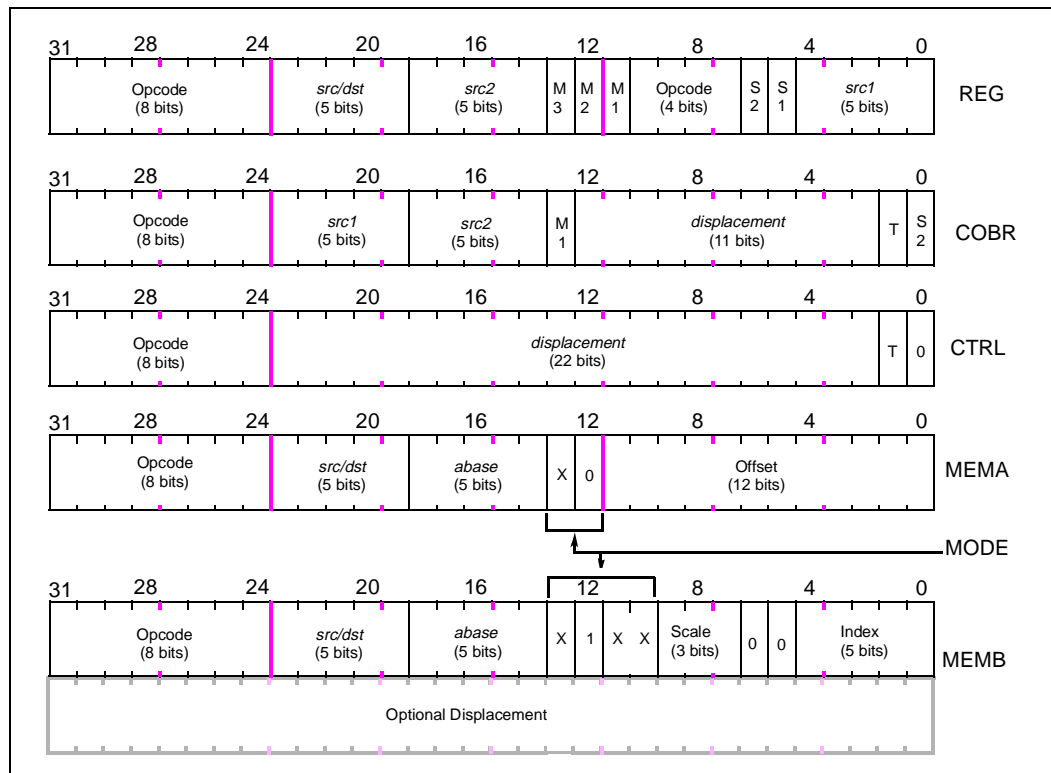


Table A-1. Instruction Field Descriptions

Instruction Field	Description
Opcode	The opcode of the instruction. Opcode encodings are defined in <a href="#">Section 6.1.8, "Opcode and Instruction Format" on page 6-4</a> .
<i>src1</i>	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.
<i>src2</i>	An input to the instruction. This field specifies a value or address.
<i>src/dst</i>	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 <i>src1</i> , <i>src2</i> , and <i>src/dst</i> fields respectively as shown in <a href="#">Table A-3</a> .

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	<i>src/dst</i>	<i>src</i> Only	<i>dst</i> 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 <sup>scale</sup>	1
	1100	Absolute Displacement	displacement	2
	1101	Register Indirect with Displacement	(abase) + displacement	2
	1110	Index with Displacement	(index) * 2 <sup>scale</sup> + displacement	2
1111	Register Indirect with Index and Displacement	(abase) + (index) * 2 <sup>scale</sup> + 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 with displacement
- register indirect with index and displacement
- IP with displacement
- register indirect
- register indirect with displacement
- index 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.

## B.1 Instruction Reference by Opcode

This section lists the instruction encoding for each i960<sup>®</sup> RM/RN I/O Processor instruction. Instructions are grouped by instruction format and listed by opcode within each format.

**Table B-1. Miscellaneous Instruction Encoding Bits**

M3	M2	M1	S2	S1	T	Description
<b>REG Format</b>						
x	x	0	x	0	—	<i>src1</i> is a global or local register
x	x	1	x	0	—	<i>src1</i> is a literal
x	x	0	x	1	—	reserved
x	x	1	x	1	—	reserved
x	0	x	0	x	—	<i>src2</i> is a global or local register
x	1	x	0	x	—	<i>src2</i> is a literal
x	0	x	1	x	—	reserved
x	1	x	1	x	—	reserved
0	x	x	x	x	—	<i>src/dst</i> is a global or local register
1	x	x	x	x	—	<i>src/dst</i> is a literal when used as a source. M3 may not be 1 when <i>src/dst</i> is used as a destination only or is used both as a source and destination in an instruction ( <b>atmod</b> , <b>modify</b> , <b>extract</b> , <b>modpc</b> ).
<b>COBR Format</b>						
—	—	0	0	—	x	<i>src1</i> , <i>src2</i> and <i>dst</i> are global or local registers
—	—	1	0	—	x	<i>src1</i> is a literal, <i>src2</i> and <i>dst</i> are global or local registers
—	—	0	1	—	x	reserved
—	—	1	1	—	x0	reserved

**Table B-2. 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			13	12	11		6	5	
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	clrbt	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	shrdd	6	0101	1001	dst	src	M3	M2	M1	1010	S2	S1	len
59:B	shrdi	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

1. Execution time based on function performed by instruction.



**Table B-2. 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
						13	12	11		10	7	
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
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	12-1 6	0110 0101	dst		M3	M2	M1	1000	S2	S1	src1
65:9	sysctl	10-1 00 <sup>1</sup>	0110 0101	src/dst	src2	M3	M2	M1	1001	S2	S1	src1
65:B	icctl	10-1 00 <sup>1</sup>	0110 0101	src/dst	src2	M3	M2	M1	1011	S2	S1	src1

1. Execution time based on function performed by instruction.

**Table B-2. 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
						13	12	11		6	5	
65:C	dcctl	10-1 00 <sup>1</sup>	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
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

1. Execution time based on function performed by instruction.

**Table B-2. 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
						13	12	11		6	5	
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
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-3. COBR Format Instruction Encodings**

Opcode	Mnemonic	Cycles to Execute	Opcode	src1	src2	M	Displacement	T	S2
			3124	23 19	1814	13	122	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 <sup>1</sup>	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<sub>A</sub> get instruction when the branch is taken.

**Table B-4. CTRL Format Instruction Encodings**

Opcode	Mnemonic	Cycles to Execute	Opcode	Displacement	T	0
			31.....24	23.....2	1	0
08	b	1 + 1 <sup>1</sup>	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-5. 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-6. 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-7. Cycle Counts for dcctl Operations**

Operation	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-8. Cycle Counts for intctl Operations**

Operation	Cycles to Execute
Disable Interrupts	13
Enable Interrupts	13
Interrupt Status Request	8

**Table B-9. MEM Format Instruction Encodings**

	3124	23.19	1814	1312	110				
	<b>Opcode</b>	<b>src/dst</b>	<b>ABASE</b>	<b>Mode</b>	<b>Offset</b>				
	3124	23.19	1814	13121110			97	65	40
	<b>Opcode</b>	<b>src/dst</b>	<b>ABASE</b>	<b>Mode</b>			<b>Scale</b>	<b>00</b>	<b>Index</b>
	<b>Displacement</b>								
<b>Effective Address</b>									
<i>efa</i> =	offset	Opcode	dst		0	0	offset		
	<i>offset(reg)</i>	Opcode	dst	reg	1	0	offset		
	<i>(reg)</i>	Opcode	dst	reg	0	1	0	0	00
	<i>disp + 8 (IP)</i>	Opcode	dst		0	1	0	1	00
	<b>Displacement</b>								
	<i>(reg1)[reg2 * scale]</i>	Opcode	dst	reg1	0	1	1	1	scale 00 reg2
	<i>disp</i>	Opcode	dst		1	1	0	0	00
	<b>Displacement</b>								
	<i>disp(reg)</i>	Opcode	dst	reg	1	1	0	1	00
	<b>Displacement</b>								
	<i>disp[reg * scale]</i>	Opcode	dst		1	1	1	0	scale 00 reg
	<b>Displacement</b>								
	<i>disp(reg1)[reg2 * scale]</i>	Opcode	dst	reg1	1	1	1	1	scale 00 reg2
	<b>Displacement</b>								

Opcode	Mnemonic	Cycles to Execute	Opcode	Mnemonic	Cycles to Execute
80	<b>ldob</b>		9A	<b>stl</b>	
82	<b>stob</b>		A0	<b>ldt</b>	
84	<b>bx</b>	4-7	A2	<b>stt</b>	
85	<b>balx</b>	5-8			
86	<b>callx</b>	9-12	B0	<b>ldq</b>	
88	<b>ldos</b>		B2	<b>stq</b>	
8A	<b>stos</b>		C0	<b>ldib</b>	
8C	<b>lda</b>		C2	<b>stib</b>	
90	<b>ld</b>		C8	<b>ldis</b>	
92	<b>st</b>		CA	<b>stis</b>	
98	<b>ldi</b>				

1. The number of cycles required to execute these instructions is based on the addressing mode used (see Table B-10).

**Table B-10. 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



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 register 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. These memory-mapped registers are specific to the i960<sup>®</sup> RM/RN I/O processor only.

## C.2 Supervisor Space Family Registers and Tables

**Table C-1. Access Types**

Access Type	Description	
R	Read	Read ( <b>ld</b> instruction) accesses are allowed.
RO	Read Only	Only Read ( <b>ld</b> instruction) accesses are allowed. Write ( <b>st</b> instruction) accesses are ignored.
W	Write	Write ( <b>st</b> instruction) accesses allowed.
R/W	Read/Write	<b>ld</b> , <b>st</b> , and <b>sysctl</b> instructions are allowed access.
WwG	Write when Granted	Writing or Modifying (through a <b>st</b> or <b>sysctl</b> instruction) the register is only allowed when modification-rights to the register have been granted. An OPERATION.UNIMPLEMENTED fault occurs if an attempt is made to write the register before rights are granted. <a href="#">Section 10.2.7.2, "Hardware Breakpoints"</a> on page 10-5 for details about getting modification rights to breakpoint registers.
Sysctl-RwG	<b>sysctl</b> Read when Granted	The value of the register can only be read by executing a <b>sysctl</b> instruction issued with the modify memory-mapped register message type. Modification rights to the register must be granted first or an OPERATION.UNIMPLEMENTED fault occurs when the <b>sysctl</b> is executed. A <b>ld</b> instruction to the register returns unpredictable results.
AtMod	<b>atmod</b> update	Register can be updated quickly through the <b>atmod</b> instruction. The <b>atmod</b> ensures correct operation by performing the update of the register in an atomic manner which provides synchronization with previous and subsequent operations. This is a faster update mechanism than <b>sysctl</b> and is optimized for a few special registers.

**Table C-2. Supervisor Space Register Addresses (Sheet 1 of 2)**

Section/Table, Register Name - Acronym, Page	80960 Internal Bus Address
Reserved	FF00 8000H to FF00 80FFH
Section 12.3.2, "Logical Memory Address Registers - LMADR0:1" on page 12-5	FF00 8100H
Reserved	FF00 8104H
Table 12-4 "Logical Memory Address Registers – LMADR0:1" on page 12-5 - 0	FF00 8108H
Table 12-5 "Logical Memory Mask Registers – LMMR0:1" on page 12-6 - 0	FF00 810CH
Table 12-4 "Logical Memory Address Registers – LMADR0:1" on page 12-5 - 1	FF00 8110H
Table 12-5 "Logical Memory Mask Registers – LMMR0:1" on page 12-6 - 1	FF00 8114H
Reserved	FF00 8118H to FF00 83FFH
Section 10.2.7.6, "Instruction Breakpoint Registers – IPBx" on page 10-10	FF00 8400H to FF00 8404H
Reserved	FF00 8408H to FF00 841FH
Section 10.2.7.5, "Data Address Breakpoint Registers – DABx" on page 10-9	FF00 8420H to FF00 8424H
Reserved	FF00 8428H to FF00 843FH
Section 10.2.7.4, "Breakpoint Control Register – BPCON" on page 10-7	FF00 8440H
Reserved	FF00 8444H to FF00 84FFH
Section 8.5.3, "Interrupt Pending (IPND) and Interrupt Mask (IMSK) Registers" on page 8-36 - IPND	FF00 8500H
Section 8.5.3, "Interrupt Pending (IPND) and Interrupt Mask (IMSK) Registers" on page 8-36 – IMSK	FF00 8504H
Reserved	FF00 8508H to FF00 850FH
Section 8.5.1, "Interrupt Control Register (ICON)" on page 8-33	FF00 8510H
Reserved	FF00 8514H to FF00 851FH
Section 8.5.2, "Interrupt Mapping Registers (IMAP0-IMAP2)" on page 8-34 – IMAP0	FF00 8520H
Section 8.5.2, "Interrupt Mapping Registers (IMAP0-IMAP2)" on page 8-34 – IMAP1	FF00 8524H
Section 8.5.2, "Interrupt Mapping Registers (IMAP0-IMAP2)" on page 8-34 – IMAP2	FF00 8528H
Reserved	FF00 852CH through FF00 85FFH

**Table C-2. Supervisor Space Register Addresses (Sheet 2 of 2)**

Section/Table, Register Name - Acronym, Page	80960 Internal Bus Address
Section 12.2.1, "PMCON Registers" on page 12-1 - Register 0 - PMCON0_1 (must be set to 32-bit bus width)	FF00 8600H
Reserved	FF00 8604H
Section 12.2.1, "PMCON Registers" on page 12-1 - Register 1 - PMCON2_3 (must be set to 32-bit bus width)	FF00 8608H
Reserved	FF00 860CH
Section 12.2.1, "PMCON Registers" on page 12-1 - Register 2 - PMCON4_5 (must be set to 32-bit bus width)	FF00 8610H
Reserved	FF00 8614H
Section 12.2.1, "PMCON Registers" on page 12-1 - Register 3 - PMCON6_7 (must be set to 32-bit bus width)	FF00 8618H
Reserved	FF00 861CH
Section 12.2.1, "PMCON Registers" on page 12-1 - Register 4 - PMCON8_9 (must be set to 32-bit bus width)	FF00 8620H
Reserved	FF00 8624H
Section 12.2.1, "PMCON Registers" on page 12-1 - Register 5 - PMCON10_11 (must be set to 32-bit bus width)	FF00 8628H
Reserved	FF00 862CH
Section 12.2.1, "PMCON Registers" on page 12-1 - Register 6 - PMCON12_13 (must be set to 32-bit bus width)	FF00 8630H
Reserved	FF00 8634H
Table 11-6 "PMCON14_15 Register Bit Description in IBR" on page 11-13 Section 12.2.1, "PMCON Registers" on page 12-1 - Register 7 - PMCON14_15 (must be set to 32-bit bus width)	FF00 8638H
Reserved	FF00 863CH through FF00 86F8H
Section 12.2.2, "Bus Control Register – BCON" on page 12-3	FF00 86FCH
Section 11.4.2, "Process Control Block – PRCB" on page 11-14	FF00 8700H
Section 8.1.5, "Interrupt Stack And Interrupt Record" on page 8-6	FF00 8704H
Section 7.6, "User and Supervisor Stacks" on page 7-17	FF00 8708H
Reserved	FF00 870CH
Table 11-9 "Processor Device ID Register - PDIDR" on page 11-19	FF00 8710H
Table 11-10 "i960® Core Processor Device ID Register - DEVICEID" on page 11-19	FF00 8710H
Reserved	FF00 8714H through FFFF FFFFH

**Table C-3. Timer Registers**

Register Name	80960 Local Bus Address
<i>Reserved</i>	FF00 0000H to FF00 02FFH
Table 18.1.3 “Timer Reload Register – TRR0:1” on page 18-7 - 0	FF00 0300H
Table 18.1.2 “Timer Count Register – TCR0:1” on page 18-6 - 0	FF00 0304H
Table 18.1.1 “Timer Mode Registers – TMR0:1” on page 18-3 - 0	FF00 0308H
<i>Reserved</i>	FF00 030CH
Table 18.1.3 “Timer Reload Register – TRR0:1” on page 18-7 - 1	FF00 0310H
Table 18.1.2 “Timer Count Register – TCR0:1” on page 18-6 - 1	FF00 0314H
Table 18.1.1 “Timer Mode Registers – TMR0:1” on page 18-3 Table 18.1.1 “Timer Mode Registers – TMR0:1” on page 18-3 - 1	FF00 0318H
<i>Reserved</i>	FF00 031CH to FF00 7FFFH

## C.3 Peripheral Memory-Mapped Register Address Space

The PMMR address space is divided to support the integrated peripherals on the i960 RM/RN I/O processor. [Table A-4](#) provides a summary of all of the non-core specific PMMR registers.

They support the DMA Controllers, Memory Controller, Bus Interface Unit, PCI And Peripheral Interrupt Controller, Messaging Unit, Internal Arbitration Unit, PCI-to-PCI Bridge Unit, PCI Address Translation Unit, I<sup>2</sup>C Bus Interface Unit, Performance Monitoring Unit, and the Application Accelerator Unit. Each of these peripherals fully describe the independent functionality of the registers, control and usage.

Portions of the i960 core processor address space are already reserved by the i960 core processor. Addresses 0000 0000H through 0000 03FFH are reserved for the core processor internal data RAM. This memory is dedicated to the i960 core processor only and inaccessible from local bus masters. Addresses FF00 0000H through FFFF FFFFH are reserved for the core 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 18FFH are allocated to the PMMR interface.

## C.4 Accessing The Peripheral Memory-Mapped Registers

The PMMR interface is a slave device connected to the 80960RM/RN 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
- 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 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.5 Architecturally Reserved Memory Space

The i960 RM/RN 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.

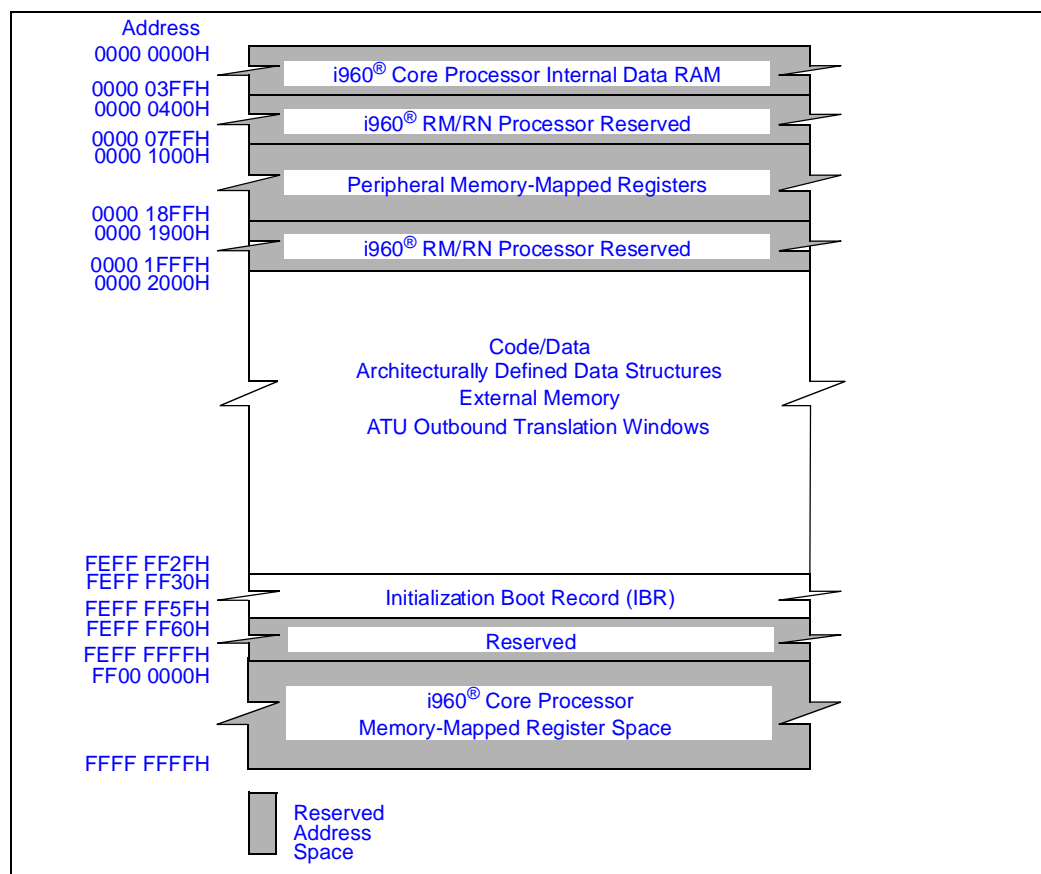
Addresses FF00 0000H through FFFF FFFFH are reserved for implementation-specific functions. This address range is termed “reserved” because future i960 architecture implementations may use these addresses for special functions such as mapped registers or data structures. Therefore, to ensure complete 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 80960RM/RN.

Addresses 0000 0400H through 0000 1FFFH are reserved for i960 RM/RN 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 i960 RM/RN I/O processor use and should not be written by the system designer.

**Figure C-2. i960 RM/RN I/O Processor Address Space**



## C.6 Peripheral Memory-Mapped Register Address Space

The PMMR address space is divided to support the integrated peripherals on the i960 RM/RN I/O processor. [Table C-6](#) shows all of the i960 RM/RN I/O processor integrated peripheral memory-mapped registers and their internal bus addresses.

**Table C-5. 80960 Internal 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 <sup>2</sup> C 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 for the PCI-to-PCI Bridge Unit and Address Translation Units are accessible via PCI configuration transactions. The DMA Controllers, Bus Interface Unit, Memory Controller, I<sup>2</sup>C Bus Interface Unit, Messaging Unit, Application Accelerator Unit, Internal Arbitration Unit, Performance Monitoring, and the PCI and Peripheral Interrupt Controller must have the address translation logic configured to translate PCI addresses into the i960 RM/RN I/O Processor address space to access the memory-mapped registers.

[Table C-6](#) shows all i960<sup>®</sup> RM/RN I/O processor integrated peripheral memory-mapped registers and their 80960 internal bus addresses.

Table C-6. Peripheral Memory-Mapped Register Locations (Sheet 1 of 9)

80960RM/RN 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 <sup>1</sup>	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	10H
	Bridge Subsystem Vendor ID Register	16	0000 1034H	11H
	Bridge Subsystem ID Register	16	0000 1036H	11H
	Reserved	x	0000 1038H through 0000 103DH	0AH through 0EH
	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



**Table C-6. Peripheral Memory-Mapped Register Locations (Sheet 2 of 9)**

80960RM/RN Peripheral	Register Description (Name)	Register Size in Bits	Internal Bus Address	PCI Configuration Space Register Number
PCI to PCI Bridge Unit	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
	Queue Control	16	0000 105EH	17H
	Reserved	x	0000 1060H through 0000 10FFH	
Performance Monitoring Unit	Global Timer Mode Register	32	0000 1100H	Must Translate PCI address to the i960 RM/RN I/O Processor 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-6. Peripheral Memory-Mapped Register Locations (Sheet 3 of 9)

80960RM/RN 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
	Reserved	32	0000 1234H	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
	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	

**Table C-6. Peripheral Memory-Mapped Register Locations (Sheet 4 of 9)**

80960RM/RN Peripheral	Register Description (Name)	Register Size in Bits	Internal Bus Address	PCI Configuration Space Register Number
Address Translation Unit	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
	Reserved	32	0000 1280H	20H
	Reserved	32	0000 1284H	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-6. Peripheral Memory-Mapped Register Locations (Sheet 5 of 9)

80960RM/RN 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 i960 RM/RN I/O Processor 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 i960 RM/RN I/O Processor 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-6. Peripheral Memory-Mapped Register Locations (Sheet 6 of 9)**

80960RM/RN 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 i960 RM/RN I/O Processor 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-6. Peripheral Memory-Mapped Register Locations (Sheet 7 of 9)

80960RM/RN 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 i960 RM/RN I/O Processor 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 i960 RM/RN I/O Processor Memory-mapped Address
	Master Latency Timer Register	32	0000 1604H	
	Multi-Transaction Timer Register	32	0000 1608H	
	Reserved	x	0000 160CH through 0000 163FH	

**Table C-6. Peripheral Memory-Mapped Register Locations (Sheet 8 of 9)**

80960RM/RN Peripheral	Register Description (Name)	Register Size in Bits	Internal Bus Address	PCI Configuration Space Register Number
Bus Interface Unit	BIU Control Register	32	0000 1640H	Must Translate PCI address to the i960 RM/RN I/O Processor Memory-mapped Address
	BIU Interrupt Status Register	32	0000 1644H	
	Reserved	x	0000 1648H through 0000 167FH	
I <sup>2</sup> C Bus Interface Unit	I <sup>2</sup> C Control Register	32	0000 1680H	Must Translate PCI address to the i960 RM/RN I/O Processor Memory-mapped Address
	I <sup>2</sup> C Status Register	32	0000 1684H	
	I <sup>2</sup> C Slave Address Register	32	0000 1688H	
	I <sup>2</sup> C Data Buffer Register	32	0000 168CH	
	I <sup>2</sup> C Clock Control Register	32	0000 1690H	
	I <sup>2</sup> C Bus Monitor Register	32	0000 1694H	
	Reserved	x	0000 1698H through 0000 16FFH	

Table C-6. Peripheral Memory-Mapped Register Locations (Sheet 9 of 9)

80960RM/RN Peripheral	Register Description (Name)	Register Size in Bits	Internal Bus Address	PCI Configuration Space Register Number
PCI And Peripheral Interrupt Controller	NMI Interrupt Status Register	32	0000 1700H	Must Translate PCI address to the i960 RM/RN I/O Processor 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 i960 RM/RN I/O Processor Memory-mapped Address
	Reserved	x	0000 1714H through 0000 17FFH	
Application Accelerator Unit	Accelerator Control Register	32	0000 1800H	Must Translate PCI address to the i960 RM/RN I/O Processor Memory-mapped Address
	Accelerator Status Register	32	0000 1804H	
	Accelerator Descriptor Address Register	32	0000 1808H	
	Accelerator Next Descriptor Address Register	32	0000 180CH	
	i960 RM/RN I/O Processor Source Address 1 Register	32	0000 1810H	
	i960 RM/RN I/O Processor Source Address 2 Register	32	0000 1814H	
	i960 RM/RN I/O Processor Source Address 3 Register	32	0000 1818H	
	i960 RM/RN I/O Processor Source Address 4 Register	32	0000 181CH	
	i960 RM/RN I/O Processor Destination Address Register	32	0000 1820H	
	Accelerator Byte Count Register	32	0000 1824H	
	Accelerator Descriptor Control Register	32	0000 1828H	
	i960 RM/RN I/O Processor Source Address 5 Register	32	0000 182CH	
	i960 RM/RN I/O Processor Source Address 6 Register	32	0000 1830H	
	i960 RM/RN I/O Processor Source Address 7 Register	32	0000 1834H	
	i960 RM/RN I/O Processor Source Address 8 Register	32	0000 1838H	
	Reserved	x	0000 183CH through 0000 18FFH	



## A

- absolute
  - displacement addressing mode 2-5
  - memory addressing mode 2-5
  - offset addressing mode 2-5
- AC 3-14
- AC register, see Arithmetic Controls (AC) register
- access faults 3-7
- access types
  - restrictions 3-6
- ADD 6-6**
- add
  - conditional instructions 6-6
  - integer instruction 6-9
  - ordinal instruction 6-9
  - ordinal with carry instruction 6-8
- addc** 6-8
- addi** 6-9
- addie** 6-6
- addig** 6-6
- addige** 6-6
- addil** 6-6
- addile** 6-6
- addine** 6-6
- addino** 6-6
- addio** 6-6
- addo** 6-9
- addoe** 6-6
- addog** 6-6
- addoge** 6-6
- addol** 6-6
- addole** 6-6
- addone** 6-6
- addono** 6-6
- addoo** 6-6
- Address Translation Unit 1-3
  - initialization 11-2
- addressing mode
  - examples 2-6
  - register indirect 2-5
- addressing registers and literals 3-4
- alignment, registers and literals 3-4
- alterbit** 6-10
- and** 6-11
- andnot** 6-11
- Application 20-1
- argument list 7-12
- Arithmetic Controls Register - AC 3-14
- Arithmetic Controls (AC) register 3-14
  - condition code flags 3-14
  - initialization 3-14
  - integer overflow flag 3-15
  - integer overflow mask bit 3-15

- no imprecise faults bit 3-15
- arithmetic instructions 5-6
  - add, subtract, multiply or divide 5-7
  - extended-precision instructions 5-9
  - remainder and modulo instructions 5-8
  - shift and rotate instructions 5-8
- arithmetic operations and data types 5-7
- assert (defined) 1-11
- atadd** 3-12, 4-8, 6-12
- atmod** 3-12, 4-8, 6-13, C-1
- atomic access 3-11
- atomic add instruction 6-12
- atomic instructions 5-17
- atomic modify instruction 6-13
- atomic-read-modify-write sequence 3-6
- ATU
  - Error conditions 15-35
  - Expansion ROM Translation Unit 15-27
  - inbound address translation 15-5
  - initialization 11-2
  - overview 15-1
  - transaction queues 15-28
  - translating in/outbound address 15-3
- ATU Interrupt Pin Register - ATUIPR 15-64
- ATU Subsystem ID Register - ASIR 15-60
- ATU Subsystem Vendor ID Register - ASVIR 15-60

## B

- b** 6-14
- bal** 6-15
- balx** 6-15
- bbc** 6-16
- bbs** 6-16
- BCON 12-3
- BCON register 12-3
- be** 6-17
- bg** 6-17
- bge** 3-15, 6-17
- Big endian 12-8
- bit field instructions 5-10
- bit instructions 5-10
- bits
  - clear 1-11
  - set 1-11
- bits and bit fields 2-3
- bl** 6-17
- ble** 6-17
- bne** 6-17
- bno** 6-17
- bo** 6-17
- boundary conditions
  - internal memory locations 12-8

- LMT boundaries 12-8
- logical data template ranges 12-8
- Boundary-Scan Register 23-6
- boundary-scan register 23-6
- boundary-scan (JTAG) 23-1
  - architecture 23-2
  - test logic 23-2
- BPCON 10-7
- branch
  - and link extended instruction 6-15
  - and link instruction 6-15
  - check bit and branch if clear set instruction 6-16
  - check bit and branch if set instruction 6-16
  - conditional instructions 6-17
  - extended instruction 6-14
  - instruction 6-14
- branch instructions, overview 5-12
  - compare and branch instructions 5-14
  - conditional branch instructions 5-13
  - unconditional branch instructions 5-13
- branch-and-link 7-1
  - returning from 7-19
- branch-and-link instruction 7-1
- branch-if-greater-or-equal instruction 3-15
- breakpoint
  - resource request message 10-6
- Breakpoint Control Register - BPCON 10-7
- Breakpoint Control (BPCON) register 10-7
  - programming 10-8
- bswap** 6-19
- built-in self test 11-6
- bus confidence self test 11-7
- Bus Control Register Bit Definitions - BCON 12-3
- Bus Control (BCON) register 12-3
  - BCON.irp bit 4-2
  - BCON.sirp bit 4-1
- Bus Controller
  - logical memory attributes 12-4
  - memory attributes 12-1
- Bus Controller Unit (BCU)
  - PMCON initialization 12-2
- bus snooping 4-5, 4-8
- bx** 6-14
- bypass register 23-6
- byte instructions 5-10
- byte swap instruction 6-19

## C

- cache
  - data
    - cache coherency and non-cacheable accesses 4-8
    - described 4-5
    - enabling and disabling 4-5
    - fill policy 4-6
    - partial-hit multi-word data accesses 4-6
    - visibility 4-8
    - write policy 4-7
  - instruction
    - enabling and disabling 4-4

- loading and locking instruction 4-4
  - visibility 4-5
- load-and-lock mechanism 4-4
- local register 4-2
  - stack frame 4-2
- cacheable writes (stores) 4-7
- caching of interrupt-handling procedure 8-29
- caching of local register sets
  - frame fills 7-7
  - frame spills 7-7
  - mapping to the procedure stack 7-11
  - updating the register cache 7-11
- call
  - extended instruction 6-23
  - instruction 6-20
  - system instruction 6-21
- call** 6-20, 7-2, 7-6
- call and return instructions 5-15
- call and return mechanism 7-1, 7-2
  - explicit calls 7-1
  - implicit calls 7-1
  - local register cache 7-3
  - local registers 7-3
  - procedure stack 7-3
  - register and stack management 7-4
    - frame pointer 7-4
    - previous frame pointer 7-5
    - return type field 7-5
    - stack pointer 7-4
  - stack frame 7-2
- call and return operations 7-5
  - call operation 7-6
  - return operation 7-6
- calls** 3-18, 6-21, 7-2, 7-6
- call-trace mode 10-3
- callx** 6-23, 7-2, 7-6
- carry conditions 3-14
- check bit instruction 6-24
- chkbit** 6-24
- clear bit instruction 6-25
- clear bits 1-11
- clrbt** 6-25
- cmpdeci** 6-26
- cmpdeco** 6-26
- cmpi** 5-11, 6-28
- cmpib** 5-11
- cmpibe** 6-30
- cmpibg** 6-30
- cmpibge** 6-30
- cmpibl** 6-30
- cmpible** 6-30
- cmpibne** 6-30
- cmpibno** 6-30
- cmpibo** 6-30
- cmpinci** 6-27
- cmpinco** 6-27
- cmpis** 5-11
- cmpo** 5-11, 6-28
- cmpobe** 6-30
- cmpobg** 6-30



- cmpobge** 6-30
- cmpobi** 6-30
- cmpoble** 6-30
- cmpobne** 6-30
- cold reset 11-5
- compare
  - and branch conditional instructions 6-30
  - and conditional compare instructions 5-11
  - and decrement integer instruction 6-26
  - and decrement ordinal instruction 6-26
  - and increment integer instruction 6-27
  - and increment ordinal instruction 6-27
  - integer conditional instruction 6-32
  - integer instruction 6-28
  - ordinal conditional instruction 6-32
  - ordinal instruction 6-28
- comparison instructions, overview
  - compare and increment or decrement instructions 5-11
  - test condition instructions 5-12
- concmpi** 6-32
- concmpo** 6-32
- conditional branch instructions 3-14
- conditional fault instructions 5-16
- control registers 3-1, 3-6
  - memory-mapped 3-5
- control table 3-1, 3-6, 3-9
  - alignment 3-12

## D

- DABx 10-9
- DAC 1-2
- Data Address Breakpoint Register - DABx 10-9
- Data Address Breakpoint (DAB) registers 10-9
  - programming 10-8
- data alignment in external memory 3-12
- data cache
  - cache coherency and non-cacheable accesses 4-8
  - coherency
    - I/O and bus masters 4-8
  - control instruction 6-33
  - described 4-5
  - enabling and disabling 4-5
  - fill policy 4-6
  - partial-hit multi-word data accesses 4-6
  - visibility 4-8
  - write policy 4-7
- data movement instructions 5-5
  - load address instruction 5-6
  - load instructions 5-5
  - move instructions 5-6
- data register
  - timing diagram 23-23
- data structures
  - control table 3-1, 3-6, 3-9
  - fault table 3-1, 3-9
  - Initialization Boot Record (IBR) 3-1, 3-9
  - interrupt stack 3-1, 3-9
  - interrupt table 3-1, 3-9
  - literals 3-4

- local stack 3-1
- Process Control Block (PRCB) 3-1, 3-9
- supervisor stack 3-1, 3-9
- system procedure table 3-1, 3-9
- user stack 3-9
- data types
  - bits and bit fields 2-3
  - integers 2-2
  - literals 2-4
  - ordinals 2-3
  - supported 2-1
  - triple and quad words 2-3
- dcctl** 3-18, 4-5, 4-8, 6-33
- debug
  - overview 10-1
- debug instructions 5-16
- Default Logical Memory Configuration Register - DLMCON 12-6
- Default Logical Memory Configuration (DLMCON) register 12-4
- Delayed 14-25
- Device ID register 23-6
- device identification register 23-6
- DEVICEID 11-19
- DEVICEID register location 3-3
- divi** 6-39
- divide integer instruction 6-39
- divide ordinal instruction 6-39
- divo** 6-39
- DLMCON 12-6
- DLMCON registers
- DMA 19-1
- downstream (defined) 1-10
- Dual Address Cycle addressing 1-2
- DWORD (defined) 1-10

## E

- ediv** 6-40
- emul** 6-41
- eshro** 6-42
- Expansion ROM Base Address Register - ERBAR 15-61
- Expansion ROM Translation Unit 15-27
- explicit calls 7-1
- extended addressing instructions 5-12
- extended divide instruction 6-40
- extended multiply instruction 6-41
- extended shift right ordinal instruction 6-42
- external memory requirements 3-11
- extract** 6-43

## F

- FAIL# pin 11-7
- fault
  - OPERATION.UNIMPLEMENTED 4-1
- fault conditional instructions 6-44
- fault conditions 9-1
- fault handling

- data structures 9-1
  - fault record 9-2, 9-6
  - fault table 9-2, 9-4
  - fault type and subtype numbers 9-3
  - fault types 9-4
  - local calls 9-2
  - multiple fault conditions 9-8
  - procedure invocation 9-6
  - return instruction pointer (RIP) 9-14
  - stack usage 9-6
  - supervisor stack 9-2
  - system procedure table 9-2
  - system-local calls 9-2
  - system-supervisor calls 9-2
  - user stack 9-2
  - fault record 9-6
    - address-of-faulting-instruction field 9-7
    - fault subtype field 9-7
    - location 9-6, 9-8
    - structure 9-7
  - fault table 3-1, 3-9, 9-4
    - alignment 3-12
    - local-call entry 9-6
    - location 9-4
    - system-call entry 9-6
  - fault type and subtype numbers 9-3
  - fault types 9-4
  - faulte** 6-44
  - faultg** 6-44
  - faultge** 6-44
  - faulti** 6-44
  - faultle** 6-44
  - faultne** 6-44
  - faultno** 6-44
  - faulto** 6-44
  - faults
    - access 3-7
    - AC.nif bit 9-19
    - ARITHMETIC.INTEGER\_OVERFLOW 6-77
    - ARITHMETIC.OVERFLOW 6-7, 6-9, 6-39, 6-70, 6-86, 6-90, 6-94
    - ARITHMETIC.ZERO\_DIVIDE 6-39, 6-40, 6-64, 6-77
    - CONSTRAINT.RANGE 6-44
    - controlling precision of (**syncf**) 9-19
    - imprecise 5-21
    - OPERATION.INVALID\_OPERAND 6-37
    - PROTECTION.LENGTH 6-22
    - TRACE.MARK 6-47, 6-62
    - TYPE.MISMATCH 6-37, 6-48, 6-54, 6-55, 6-56, 6-57, 6-66
  - fields
    - preserved 1-10
    - read only 1-10
    - read/clear 1-10
    - read/set 1-11
    - reserved 1-10
  - floating point 3-15
  - flush local registers instruction 6-46
  - flushreg** 6-46, 7-11
  - fmark** 6-47
  - force mark instruction 6-47
  - FP, see Frame Pointer
  - frame fills 7-7
  - Frame Pointer (FP) 7-4
    - location 3-3
  - frame spills 7-7
- ## G
- global registers 3-1, 3-3
- ## H
- halt** 6-48
  - halt CPU instruction 6-48
  - hardware breakpoint resources 10-5
    - requesting access privilege 10-6
  - hexadecimal numbering (defined) 1-10
  - high priority interrupts 4-2
  - Host processor (defined) 1-10
- ## I
- IBR 11-13
  - IBR, see initialization boot record
  - icctl** 3-18, 4-3, 4-4, 4-5
  - IEEE Standard Test Access Port 23-2
  - IEEE Std. 1149.1 23-2
  - IMI 11-1, 11-9
  - implicit calls 7-1, 9-2
  - imprecise faults 5-21
  - inbound address translation 15-5
  - index with displacement addressing mode 2-5
  - indivisible access 3-11
  - inequalities (greater than, equal or less than) conditions 3-14
  - Initial Memory Image (IMI) 11-1, 11-9
  - initialization 11-6
    - software 6-96
  - Initialization Boot Record (IBR) 3-1, 3-9, 11-1, 11-11, 11-13
    - alignment 3-12
  - initialization data structures 3-9
  - initialization requirements
    - control table 11-18
    - data structures 11-9
    - Process Control Block 11-14
  - instruction breakpoint modes
    - programming 10-10
  - Instruction Breakpoint Register - IPBx 10-10
  - instruction cache 3-13
    - coherency 4-5
    - configuration 3-13
    - enabling and disabling 4-4, 11-17
    - locking instructions 4-4
    - overview 4-3
    - visibility 4-5
  - instruction formats 5-3
    - assembly language format 5-1
    - instruction encoding format 5-2
  - instruction optimizations 5-18



Instruction Pointer (IP) register 3-13

Instruction Register (IR) 23-4

timing diagram 23-22

Instruction set

**atmod** C-1

**sysctl** C-1

instruction set

**6-6**

**ADD 6-6**

**addc** 6-8

**addi** 6-9

**addie** 6-6

**addig** 6-6

**addige** 6-6

**addil** 6-6

**addile** 6-6

**addine** 6-6

**addino** 6-6

**addo** 6-9

**addoe** 6-6

**addog** 6-6

**addoge** 6-6

**addol** 6-6

**addole** 6-6

**addone** 6-6

**addono** 6-6

**addoo** 6-6

**alterbit** 6-10

**and** 6-11

**andnot** 6-11

**atadd** 3-12, 4-8, 6-12

**atmod** 3-12, 4-8, 6-13

**b** 6-14

**bal** 6-15

**balx** 6-15

**bbc** 6-16

**bbs** 6-16

**be** 6-17

**bg** 6-17

**bge** 3-15, 6-17

**bl** 6-17

**ble** 6-17

**bne** 6-17

**bno** 6-17

**bo** 6-17

**bswap** 6-19

**bx** 6-14

**call** 6-20, 7-2, 7-6

**calls** 3-18, 6-21, 7-2, 7-6

**callx** 6-23, 7-2, 7-6

**chkbit** 6-24

**clrbit** 6-25

**cmpdeci** 6-26

**cmpdeco** 6-26

**cmpi** 5-11, 6-28

**cmpib** 5-11

**cmpibe** 6-30

**cmpibg** 6-30

**cmpibge** 6-30

**cmpibl** 6-30

**cmpible** 6-30

**cmpibne** 6-30

**cmpibno** 6-30

**cmpibo** 6-30

**cmpinci** 6-27

**cmpinco** 6-27

**cmpis** 5-11

**cmpo** 5-11, 6-28

**cmpobe** 6-30

**cmpobg** 6-30

**cmpobge** 6-30

**cmpobl** 6-30

**cmpoble** 6-30

**cmpobne** 6-30

**concmpi** 6-32

**concmpo** 6-32

**dcctl** 3-18, 4-5, 4-8, 6-33

**divi** 6-39

**divo** 6-39

**ediv** 6-40

**emul** 6-41

**eshro** 6-42

**extract** 6-43

**faulte** 6-44

**faultg** 6-44

**faultge** 6-44

**faultl** 6-44

**faultle** 6-44

**faultne** 6-44

**faultno** 6-44

**faulto** 6-44

**flushreg** 6-46

**fmark** 6-47

**halt** 6-48

**icctl** 3-18, 4-3, 4-4, 4-5

**intctl** 3-18, 6-55

**intdis** 3-18, 6-56

**inten** 3-18, 6-57

**ld** 2-2, 3-12, 6-58

**lda** 6-61

**ldib** 2-2, 6-58

**ldis** 2-2, 6-58

**ldl** 3-4, 4-6, 6-58

**ldob** 2-3, 6-58

**ldos** 2-3, 6-58

**ldq** 3-12, 4-6, 6-58

**ldt** 4-6, 6-58

**mark** 6-62

**modac** 3-14, 6-63

**modi** 6-64

**modify** 6-65

**modpc** 3-16, 3-17, 3-18, 6-66, 10-3

**modtc** 6-67, 10-2

**mov** 6-68

**movl** 6-68

**movq** 6-68

**movt** 6-68

**muli** 6-70

**mulo** 6-70

**nand** 6-71

**nor** 6-72  
**not** 6-73  
**notand** 6-73  
**notbit** 6-74  
**notor** 6-75  
**or** 6-76  
**ornot** 6-76  
**remi** 6-77  
**remo** 6-77  
**ret** 6-78  
**rotate** 6-80  
**scanbit** 6-81  
**scanbyte** 6-82  
**sele** 5-6, 6-83  
**selg** 5-6, 6-83  
**selge** 5-6, 6-83  
**sell** 5-6, 6-83  
**selle** 5-6, 6-83  
**selne** 5-6, 6-83  
**selno** 5-6, 6-83  
**selo** 5-6, 6-83  
**setbit** 6-84  
**shli** 6-85  
**shlo** 6-85  
**shrdi** 6-85  
**shri** 6-85  
**shro** 6-85  
**spanbit** 6-87  
**st** 2-2, 3-12, 6-88  
**stib** 2-2, 6-88  
**stis** 2-2, 6-88  
**stl** 3-12, 4-6, 6-88  
**stob** 2-3, 6-88  
**stos** 2-3  
**stq** 3-12, 4-6, 6-88  
**stt** 4-6, 6-88  
**subc** 6-91  
**subi** 6-94  
**subie** 6-92  
**subig** 6-92  
**subige** 6-92  
**subil** 6-92  
**subile** 6-92  
**subine** 6-92  
**subino** 6-92  
**subio** 6-92  
**subo** 6-94  
**suboe** 6-92  
**subog** 6-92  
**suboge** 6-92  
**subol** 6-92  
**subole** 6-92  
**subone** 6-92  
**subono** 6-92  
**suboo** 6-92  
**syncf** 6-95, 9-19  
**sysctl** 3-18, 4-3, 4-4, 4-5, 6-96, 10-6  
**teste** 6-100  
**testg** 6-100  
**testge** 6-100  
**testl** 6-100  
**testle** 6-100  
**testne** 6-100  
**testno** 6-100  
**testo** 6-100  
**xnor** 6-102  
**xor** 6-102  
instruction set functional groups 5-4  
Instruction Trace Event 6-3  
Instructions  
    TRISTATE 23-5  
instructions  
    conditional branch 3-14  
instruction-trace mode 10-3  
INTA#/XINT0# 8-20  
INTB#/XINT1# 8-20  
**intctl** 3-18, 6-55  
INTC#/XINT2# 8-20  
**intdis** 3-18, 6-56  
INTD#/XINT3# 8-20  
integer flow masking 5-20  
integers 2-2  
    data truncation 2-2  
    sign extension 2-2  
Integrated Memory Controller 1-3  
**inten** 3-18, 6-57  
Inter-Integrated Circuit Bus Interface Unit 1-4  
internal data RAM 4-1  
    modification 4-1  
    size 4-1  
internal self test program 11-7  
interrupt  
    timer 8-15  
Interrupt Control (ICON) register  
    memory-mapped addresses 8-32  
interrupt controller  
    configuration 8-16  
    overview 8-11  
    program interface 8-12  
    programmer interface 8-32  
    setup 8-16  
interrupt handling procedures 8-16  
    AC and PC registers 8-16  
    address space 8-16  
    global registers 8-16  
    instruction cache 8-16  
    interrupt stack 8-16  
    local registers 8-16  
    location 8-16  
    supervisor mode 8-16  
Interrupt Mack (IMSK) register  
    atomic-read-modify-write sequence 3-6  
Interrupt Mapping (IMAP0-IMAP2) registers 8-34  
interrupt mask  
    saving 8-11  
Interrupt Mask (IMSK) register 8-36  
Interrupt Pending (IPND) register 8-36  
    atomic-read-modify-write sequence 3-6  
interrupt performance  
    caching of interrupt-handling 8-29



- interrupt stack 8-29
- local register cache 8-29
- interrupt posting 8-1
- interrupt procedure pointer 8-5
- interrupt record 8-6
  - location 8-6
- interrupt requests
  - sysctl 8-7
- interrupt sequencing of operations 8-15
- interrupt service latency 8-27
- interrupt stack 3-1, 3-9, 8-6, 8-29
  - alignment 3-12
  - structure 8-6
- interrupt table 3-1, 3-9, 8-4
  - alignment 3-12, 8-4
  - caching mechanism 8-5
  - location 8-4
  - pending interrupts 8-5
  - vector entries 8-5
- interrupt vectors
  - caching 4-1
- interrupts
  - dedicated mode posting 8-13
  - executing-state 8-17
  - function 8-1
  - global disable instruction 6-56
  - global enable and disable instruction 6-55
  - global enable instruction 6-57
  - high priority 4-2
  - internal RAM 8-28
  - interrupt context switch 8-17
  - interrupt handling procedures 8-16
  - interrupt record 8-6
  - interrupt stack 8-6
  - interrupt table 8-4
  - interrupted-state 8-17
  - masking hardware interrupts 8-11
  - Non-Maskable Interrupt (NMI) 8-3, 8-14
  - overview 8-1
  - physical characteristics 8-20
  - posting 8-1
  - priority handling 8-8
  - priority-31 interrupts 8-3, 8-11
  - programmable options 8-13
  - restoring r3 8-11
  - servicing 8-3
  - sysctl** 8-15
  - vector caching 8-28
- IP register, see Instruction Pointer (IP) register
- IP with displacement addressing mode 2-6
- IPBx 10-10
- IxWorks Real-Time Operating System (RTOS) 1-5
- IxWorks (Wind River Systems RTOS) 1-5
- I2C interface unit 22-1
- i960 Core Processor Device ID Register - DEVICEID 11-19
- i960 core processor (defined) 1-10

## J

- JTAG (boundary-scan) 23-1

## L

- ld** 2-2, 3-12, 6-58
- lda** 6-61
- ldib** 2-2, 6-58
- ldis** 2-2, 6-58
- ldl** 3-4, 4-6, 6-58
- ldob** 2-3, 6-58
- ldos** 2-3, 6-58
- ldq** 3-12, 4-6, 6-58
- ldt** 4-6, 6-58
- leaf procedures 7-1
- literal addressing and alignment 3-4
- literals 2-4, 3-1, 3-4
  - addressing 3-4
- Little endian 12-8
- little endian byte order 3-12
- LMADR register
  - LMADR0
    - 1 12-5
- LMCON registers
- LMMR0
  - 1 12-6
- load address instruction 6-61
- load instructions 5-5, 6-58
- load-and-lock mechanism 4-4
- local bus (defined) 1-10
- local calls 7-2, 7-13, 9-2
  - call** 7-2
  - callx** 7-2
- Local memory (defined) 1-10
- Local processor (defined) 1-10
- local register cache 7-3
  - overview 4-2
- local registers 3-1, 7-3
  - allocation 3-3, 7-3
  - management 3-3
  - usage 7-3
- local stack 3-1
- logical data templates
  - effective range 12-7
- logical instructions 5-9
- Logical Memory Address Registers - LMADR0
  - 1 12-5
- Logical Memory Address (LMADR) register 12-4
- Logical Memory Address (LMADR) registers
  - programming 12-4
- Logical Memory Configuration (LMCON) registers 12-4
- Logical Memory Mask Registers - LMMR0
  - 1 12-6
- Logical Memory Mask (LMMR) registers
  - programming 12-4
- Logical Memory Template registers (LMTs)
  - modifying 12-8
- Logical Memory Templates (LMTs)
  - accesses across boundaries 12-8
  - boundary conditions 12-8
  - enabling 12-7
  - enabling and disabling data caching 12-7
  - overlapping ranges 12-8

values after reset 12-8

## M

**mark** 6-62

Mark Trace Event 6-3

memory address space 3-1

external memory requirements 3-11

atomic access 3-11

data alignment 3-12

data block sizes 3-12

data block storage 3-12

indivisible access 3-11

instruction alignment in external memory 3-12

little endian byte order 3-12

reserved memory 3-11

location 3-10

management 3-10

memory addressing modes

absolute 2-5

examples 2-6

index with displacement 2-5

IP with displacement 2-6

register indirect 2-5

memory-mapped control registers 3-5

Memory-Mapped Registers (MMR) 3-6, 3-11

MMR, see Memory-Mapped Registers (MMR)

**modac** 3-14, 6-63

**modi** 6-64

**modify** 6-65

modify arithmetic controls instruction 6-63

modify process controls instruction 6-66

modify trace controls instruction 6-67, 10-2

**modpc** 3-16, 3-17, 3-18, 6-66, 10-3

**modtc** 6-67, 10-2

modulo integer instruction 6-64

**mov** 6-68

move instructions 6-68

**movl** 6-68

**movq** 6-68

**movt** 6-68

**muli** 6-70

**mulo** 6-70

multiple fault conditions 9-8

multiply integer instruction 6-70

multiply ordinal instruction 6-70

## N

**nand** 6-71

NMI# 8-20

No Imprecise Faults (AC.nif) bit 9-15, 9-19

Non-Maskable Interrupt (NMI) 8-3

**nor** 6-72

**not** 6-73

**notand** 6-73

**notbit** 6-74

**notor** 6-75

## O

On-Circuit Emulation (ONCE) mode 11-1, 11-2, 23-1

OPERATION.UNIMPLEMENTED 4-1

**or** 6-76

ordinals 2-3

sign and sign extension 2-3

**ornot** 6-76

overflow conditions 3-14

## P

parameter passing 7-12

argument list 7-12

by reference 7-12

by value 7-12

PC 3-16

PC register, see Process Controls (PC) register

PCI 14-1

PCI-to-PCI Bridge 1-2

initialization 11-2

PDIDR 11-19

pending interrupts 8-5

encoding 8-5

interrupt procedure pointer 8-5

pending priorities field 8-5

Performance 21-1

performance optimization 5-18

Philips Corporation 1-4

Physical Memory Configuration (PMCON) registers

initial values 12-2

Physical Memory Control Registers - PMCON0

15 12-2

PMCON0

15 12-2

PMCON14\_15 Register Bit Description in IBR 11-13

powerup/reset initialization

timer powerup 18-10

PRCB 11-15

PRCB, see Processor Control Block (PRCB)

prereturn-trace mode 10-4

preserved fields 1-10

Previous Frame Pointer (PFP) 3-1, 7-4, 7-5

location 3-3

Primary and Secondary PCI buses (defined) 1-10

Primary PCI Bus Reset signal 11-2

priority-31 interrupts 8-3, 8-11

procedure calls

branch-and-link 7-1

call and return mechanism 7-1

leaf procedures 7-1

procedure stack 7-3

growth 7-3

Process Control Block AC Register Initial Image 11-15

Process Control Block (PRCB) 3-1, 3-9, 4-4, 11-1, 11-14

alignment 3-12

configuration 11-14

register cache configuration word 11-17

Process Control Register - PC 3-16

Process Controls (PC) register





- execution mode flag 3-16
- initialization 3-17
- modification 3-17
- modpc 3-17
- priority field 3-16
- processor state flag 3-16
- trace enable bit 3-17
- trace fault pending flag 3-17
- Processor Device ID Register - PDIDR 11-19
- processor management instructions 5-17
- processor state registers 3-1, 3-13
  - Arithmetic Controls (AC) register 3-14
  - Instruction Pointer (IP) register 3-13
  - Process Controls (PC) register 3-16
  - Trace Controls (TC) register 3-17
- programming
  - logical memory attributes 12-8
- P\_RST# 11-2, 11-3

## R

- RAM 3-9
  - internal data
    - described 4-1
- read only fields 1-10
- read/clear fields 1-10
- read/set fields 1-11
- register
  - addressing 3-4
  - addressing and alignment 3-4
  - boundary-scan 23-6
  - Breakpoint Control (BPCON) 10-7
  - cache 4-2
  - control 3-6
    - memory-mapped 3-5
  - DEVICEID
    - memory location 3-3
  - global 3-3
  - indirect addressing mode
    - register-indirect-with-displacement 2-5
    - register-indirect-with-index 2-5
    - register-indirect-with-index-and-displacement 2-5
    - register-indirect-with-offset 2-5
  - Interrupt Control (ICON) 8-32
  - Interrupt Mapping (IMAP0-IMAP2) 8-34
  - Interrupt Mask (IMSK) 8-36
  - Interrupt Pending (IPND) 8-36
  - local
    - allocation 3-3
    - management 3-3
  - processor-state 3-13
  - scoreboarding
    - example 3-4
  - TCRx 18-6
- Registers
  - Arithmetic Controls Register - AC 3-14
  - Boundary-Scan 23-6
  - Breakpoint Control Register - BPCON 10-7
  - Bus Control Register Bit Definitions - BCON 12-3
  - bypass 23-6

- Data Address Breakpoint Register - DABx 10-9
- Default Logical Memory Configuration Register - DLMCON 12-6
- Instruction Breakpoint Register - IPBx 10-10
- i960 Core Processor Device ID Register - DEVICEID 11-19
- Logical Memory Address Registers - LMADR0 1 12-5
- Logical Memory Mask Registers - LMMR0 1 12-6
- Physical Memory Control Registers - PMCON0 15 12-2
- PMCON14\_15 Register Bit Description in IBR 11-13
- Process Control Block AC Register Initial Image 11-15
- Process Control Register - PC 3-16
- Processor Device ID Register - PDIDR 11-19
- RUNBIST 23-6
- Timer Count Register - TCRx 18-6
- Timer Mode Register - TMRx 18-3
- Timer Reload Register - TRRx 18-7
- 80960RxJx Trace Controls Register - TC 10-2

- registers
  - Logical Memory Templates (LMTs) 12-8
- re-initialization
  - software 6-96
- remainder integer instruction 6-77
- remainder ordinal instruction 6-77
- remi** 6-77
- remo** 6-77
- reserved fields 1-10
- reserving frames in the local register cache 8-29
- reset state 11-5
- ret** 6-78
- Return Instruction Pointer (RIP) 7-4
  - location 3-3
- return operation 7-6
- return type field 7-5
- RIP, see Return Instruction Pointer (RIP)
- ROM 3-9
- rotate** 6-80
- RST\_MODE 11-2
- RTOS 1-5
- Run Built-In Self-Test (RUNBIST) register 23-6
- RUNBIST register 23-6

## S

- scanbit** 6-81
- scanbyte** 6-82
- Secondary PCI Bus Arbiter
  - initialization 11-2
- sele** 5-6, 6-83
  - select based on equal instruction 5-6
  - select based on less or equal instruction 5-6
  - select based on not equal instruction 5-6
  - select based on ordered instruction 5-6
  - Select Based on Unordered 5-6
- self test (STEST) pin 11-7
- selg** 5-6, 6-83
- selge** 5-6, 6-83

- sell** 5-6, 6-83
  - selle** 5-6, 6-83
  - selne** 5-6, 6-83
  - selno** 5-6, 6-83
  - selo** 5-6, 6-83
  - set bits 1-11
  - setbit** 6-84
  - shift instructions 6-85
  - shli** 6-85
  - shlo** 6-85
  - shrdi** 6-85
  - shri** 6-85
  - shro** 6-85
  - sign extension
    - integers 2-2
    - ordinals 2-3
  - Signal 1-11
  - software re-initialization 6-96
  - spanbit** 6-87
  - SP, see Stack Pointer
  - src/dst parameter encodings 10-6
  - st** 2-2, 3-12, 6-88
  - stack frame
    - allocation 7-2
  - stack frame cache 4-2
  - Stack Pointer (SP) 7-4
    - location 3-3
  - stacks 3-9
  - STEST 11-7
  - stib** 2-2, 6-88
  - stis** 2-2, 6-88
  - stl** 3-12, 4-6, 6-88
  - stob** 2-3, 6-88
  - store instructions 5-5, 6-88
  - stos** 2-3
  - stq** 3-12, 4-6, 6-88
  - stt** 4-6, 6-88
  - subc** 6-91
  - subi** 6-94
  - subie** 6-92
  - subig** 6-92
  - subige** 6-92
  - subil** 6-92
  - subile** 6-92
  - subine** 6-92
  - subino** 6-92
  - subio** 6-92
  - subo** 6-94
  - suboe** 6-92
  - subog** 6-92
  - suboge** 6-92
  - subol** 6-92
  - subole** 6-92
  - subone** 6-92
  - subono** 6-92
  - suboo** 6-92
  - subtract
    - conditional instructions 6-92
    - integer instruction 6-94
    - ordinal instruction 6-94
    - ordinal with carry instruction 6-91
  - supervisor calls 7-2
  - supervisor mode resources 3-18
  - supervisor space family registers and tables C-1, C-2
  - supervisor stack 3-1, 3-9
    - alignment 3-12
  - supervisor-trace mode 10-3
  - syncf** 6-95, 9-19
  - synchronize faults instruction 6-95
  - sysctl** 3-18, 4-3, 4-4, 4-5, 6-96, 10-6, C-1
  - system calls 7-2, 7-14
    - calls** 7-2
      - system-local 7-2, 9-2
      - system-supervisor 7-2, 9-2
  - system control instruction 6-96
  - system procedure table 3-1, 3-9
    - alignment 3-12
- ## T
- TAP Test Data Registers 23-6
  - TC 10-2
  - TCRx 18-6
  - Test Access Port (TAP) controller 23-16
    - block diagram 23-3
    - state diagram 23-16
  - Test Data Input (TDI) pin 23-4
  - test features 23-2
  - test instructions 6-100
  - Test Mode Select (TMS) line 23-16
  - teste** 6-100
  - testg** 6-100
  - testge** 6-100
  - testi** 6-100
  - testle** 6-100
  - testne** 6-100
  - testno** 6-100
  - testo** 6-100
  - timer
    - interrupts 8-15
    - memory-mapped addresses 18-2
  - Timer Count Register - TCRx 18-6
  - Timer Count Register (TCRx) 18-6
  - Timer Mode Register
    - timer mode control bit summary 18-8
  - Timer Mode Register - TMRx 18-3
  - Timer Mode Register (TMRx)
    - terminal count 18-3
    - timer clock encodings 18-5
  - Timer Reload Register - TRRx 18-7
  - TMRx 18-3
  - Trace Controls (TC) register 3-17, 10-2
  - trace events 10-1
    - hardware breakpoint registers 10-1
    - mark** and **fmark** 10-1
    - PC and TC registers 10-1
  - trace-fault-pending flag 10-3
  - TRISTATE 23-5
  - TRRx 18-7
  - true/false conditions 3-14



## U

- unordered numbers 3-15
- Upstream (defined) 1-10
- user space family registers and tables C-4
- user stack 3-9
  - alignment 3-12
- user supervisor protection model 3-18
  - supervisor mode resources 3-18
  - usage 3-18

## V

- vector entries 8-5
  - structure 8-5

## W

- warm reset 11-5

- words

- triple and quad 2-3
- Word/Data Word notation conventions 2-2

## X

- XINT4# 8-20
- XINT5# 8-20
- xnor** 6-102
- xor** 6-102

## Z

- 05\_Figure 8-19
- 80960 core
  - initialization 11-2
- 80960RxJx Trace Controls Register - TC 10-2

