



# *i960<sup>®</sup> Hx Microprocessor*

**Developer's Manual**

---

*September 1998*

Order Number: **272484-002**



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 i960 Hx microprocessor 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	The i960® Processor Family .....	1-2
1.2	i960® Hx Processor Key Features .....	1-2
1.2.1	Execution Architecture .....	1-2
1.2.2	Pipelined, Burst Bus .....	1-3
1.2.3	On-Chip Caches and Data RAM .....	1-4
1.2.4	Priority Interrupt Controller .....	1-4
1.2.5	Guarded Memory Unit .....	1-4
1.2.6	Dual-Programmable Timers .....	1-4
1.3	About this Manual.....	1-5
1.4	Notation and Terminology .....	1-7
1.4.1	Reserved and Preserved.....	1-7
1.4.2	Specifying Bit and Signal Values.....	1-8
1.4.3	Representing Numbers .....	1-8
1.4.4	Register Names.....	1-8
1.5	Related Documents.....	1-9
<b>2</b>	<b>Data Types and Memory Addressing Modes .....</b>	<b>2-1</b>
2.1	Data Types .....	2-1
2.1.1	Integers .....	2-2
2.1.2	Ordinals .....	2-2
2.1.3	Bits and Bit Fields.....	2-3
2.1.4	Triple- and Quad-Words .....	2-3
2.1.5	Register Data Alignment .....	2-3
2.1.6	Literals .....	2-4
2.2	Bit and Byte Ordering in Memory .....	2-4
2.2.1	Bit Ordering .....	2-4
2.2.2	Byte Ordering .....	2-4
2.3	Memory Addressing Modes.....	2-6
2.3.1	Absolute .....	2-7
2.3.2	Register Indirect .....	2-7
2.3.3	Index with Displacement .....	2-8
2.3.4	IP with Displacement.....	2-8
2.3.5	Addressing Mode Examples.....	2-8
<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 .....	3-2
3.2.2	Local Registers.....	3-3
3.2.3	Special Function Registers (SFRs) .....	3-4
3.2.4	Register Scoreboarding.....	3-4
3.2.5	Literals .....	3-5
3.2.6	Register and Literal Addressing and Alignment .....	3-5
3.3	Memory-Mapped Control Registers .....	3-6
3.3.1	Memory-Mapped Registers (MMR) .....	3-6



	3.3.1.1	Restrictions on Instructions that Access Memory-Mapped Registers .....	3-7
	3.3.1.2	Access Faults .....	3-7
3.4		Architecturally Defined Data Structures .....	3-14
3.5		Memory Address Space .....	3-15
	3.5.1	Memory Requirements .....	3-16
	3.5.2	Data and Instruction Alignment in the Address Space .....	3-16
	3.5.3	Byte, Word and Bit Addressing .....	3-17
	3.5.4	Internal Data RAM .....	3-18
	3.5.5	Instruction Cache .....	3-19
	3.5.6	Data Cache .....	3-20
3.6		Processor-State Registers .....	3-21
	3.6.1	Instruction Pointer (IP) Register .....	3-21
	3.6.2	Arithmetic Controls (AC) Register .....	3-21
		3.6.2.1 Initializing and Modifying the AC Register .....	3-22
		3.6.2.2 Condition Code (AC.cc) .....	3-22
	3.6.3	Process Controls (PC) Register .....	3-24
		3.6.3.1 Initializing and Modifying the PC Register .....	3-25
	3.6.4	Trace Controls (TC) Register .....	3-26
3.7		User-Supervisor Protection Model .....	3-26
	3.7.1	Supervisor Mode Resources .....	3-26
	3.7.2	Using the User-Supervisor Protection Model .....	3-27
4		<b>Cache and On-Chip Data RAM</b> .....	4-1
4.1		Internal Data Ram .....	4-1
4.2		Local Register Cache .....	4-3
4.3		Big Endian Accesses to Internal Ram and Data Cache .....	4-4
4.4		Instruction Cache .....	4-4
	4.4.1	Enabling and Disabling the Instruction Cache .....	4-5
	4.4.2	Operation While the Instruction Cache is Disabled .....	4-5
	4.4.3	Loading and Locking Instructions in the Instruction Cache .....	4-5
	4.4.4	Instruction Cache Visibility .....	4-5
	4.4.5	Instruction Cache Coherency .....	4-6
	4.4.6	Instruction Cache Interaction with Guarded Memory .....	4-6
4.5		Data Cache .....	4-6
	4.5.1	Enabling and Disabling the Data Cache .....	4-7
	4.5.2	Multi-Word Data Accesses that Partially Hit the Data Cache .....	4-8
	4.5.3	Data Cache Fill Policy .....	4-8
	4.5.4	Data Cache Write Policy .....	4-9
	4.5.5	Data Cache Coherency and Non-Cacheable Accesses .....	4-10
	4.5.6	External I/O and Bus Masters and Cache Coherency .....	4-10
	4.5.7	Quickly Invalidating Portions of Data Cache .....	4-11
	4.5.8	Data Cache Visibility .....	4-11
5		<b>Instruction Set Overview</b> .....	5-1
5.1		Instruction Formats .....	5-1
	5.1.1	Assembly Language Format .....	5-1
	5.1.2	Instruction Encoding Formats .....	5-2
	5.1.3	Instruction Operands .....	5-3
5.2		Instruction Groups .....	5-4
	5.2.1	Data Movement .....	5-5



5.2.1.1	Load and Store Instructions .....	5-5
5.2.1.2	Move.....	5-6
5.2.1.3	Load Address .....	5-6
5.2.2	Select Conditional .....	5-6
5.2.3	Arithmetic .....	5-7
5.2.3.1	Add, Subtract, Multiply, Divide, Conditional Add and Conditional Subtract .....	5-8
5.2.3.2	Remainder and Modulo .....	5-8
5.2.3.3	Shift, Rotate and Extended Shift .....	5-9
5.2.3.4	Extended Arithmetic .....	5-10
5.2.4	Logical.....	5-10
5.2.5	Bit, Bit Field and Byte Operations .....	5-11
5.2.5.1	Bit Operations.....	5-11
5.2.5.2	Bit Field Operations.....	5-11
5.2.5.3	Byte Operations.....	5-12
5.2.6	Comparison .....	5-12
5.2.6.1	Compare and Conditional Compare .....	5-12
5.2.6.2	Compare and Increment or Decrement.....	5-13
5.2.6.3	Test Condition Codes.....	5-13
5.2.7	Branch .....	5-14
5.2.7.1	Branch Prediction .....	5-14
5.2.7.2	Unconditional Branch .....	5-15
5.2.7.3	Conditional Branch.....	5-15
5.2.7.4	Compare and Branch .....	5-16
5.2.8	Call/Return .....	5-17
5.2.9	Faults.....	5-18
5.2.10	Debug .....	5-18
5.2.11	Atomic Instructions .....	5-19
5.2.12	Processor Management .....	5-19
5.2.13	Cache Control .....	5-20
<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 .....	6-3
6.1.4	Description .....	6-3
6.1.5	Action .....	6-3
6.1.6	Faults.....	6-5
6.1.7	Example .....	6-5
6.1.8	Opcode and Instruction Format.....	6-5
6.1.9	See Also .....	6-6
6.1.10	Side Effects .....	6-6
6.1.11	Notes .....	6-6
6.2	Instructions .....	6-6
6.2.1	ADD<cc> .....	6-7
6.2.2	addc.....	6-10
6.2.3	addi, addo.....	6-11
6.2.4	alterbit.....	6-12
6.2.5	and, andnot .....	6-13
6.2.6	atadd .....	6-14



6.2.7	atmod .....	6-15
6.2.8	b, bx .....	6-16
6.2.9	bal, balx .....	6-17
6.2.10	bbc, bbs .....	6-19
6.2.11	BRANCH<cc> .....	6-21
6.2.12	bswap .....	6-23
6.2.13	call .....	6-24
6.2.14	calls .....	6-25
6.2.15	callx .....	6-27
6.2.16	chkbit .....	6-28
6.2.17	clrbit .....	6-29
6.2.18	cmpdeci, cmpdeco .....	6-30
6.2.19	cmpinci, cmpinco .....	6-31
6.2.20	COMPARE .....	6-32
6.2.21	COMPARE AND BRANCH<cc> .....	6-34
6.2.22	concmipi, concmopo .....	6-36
6.2.23	dcctl .....	6-38
6.2.24	dcinv (80960Hx-Specific Instruction) .....	6-45
6.2.25	divi, divo .....	6-46
6.2.26	ediv .....	6-47
6.2.27	emul .....	6-48
6.2.28	eshro .....	6-49
6.2.29	extract .....	6-50
6.2.30	FAULT<cc> .....	6-51
6.2.31	flushreg .....	6-53
6.2.32	fmark .....	6-54
6.2.33	icctl .....	6-55
6.2.34	intctl .....	6-62
6.2.35	intdis .....	6-64
6.2.36	inten .....	6-65
6.2.37	LOAD .....	6-66
6.2.38	lda .....	6-69
6.2.39	mark .....	6-70
6.2.40	modac .....	6-71
6.2.41	modi .....	6-72
6.2.42	modify .....	6-73
6.2.43	modpc .....	6-74
6.2.44	modtc .....	6-75
6.2.45	MOVE .....	6-76
6.2.46	muli, mulo .....	6-78
6.2.47	nand .....	6-79
6.2.48	nor .....	6-80
6.2.49	not, notand .....	6-81
6.2.50	notbit .....	6-82
6.2.51	notor .....	6-83
6.2.52	or, ornot .....	6-84
6.2.53	remi, remo .....	6-85
6.2.54	ret .....	6-86
6.2.55	rotate .....	6-88
6.2.56	scanbit .....	6-89



6.2.57	scanbyte .....	6-90
6.2.58	SEL<cc>.....	6-91
6.2.59	setbit.....	6-93
6.2.60	SHIFT .....	6-94
6.2.61	spanbit.....	6-97
6.2.62	STORE .....	6-98
6.2.63	subc.....	6-103
6.2.64	SUB<cc> .....	6-104
6.2.65	subi, subo .....	6-106
6.2.66	syncf .....	6-107
6.2.67	sysctl .....	6-108
6.2.68	TEST<cc> .....	6-112
6.2.69	xnor, xor .....	6-114
<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 .....	7-2
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 .....	7-5
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 .....	7-6
7.1.3.1	Call Operation .....	7-6
7.1.3.2	Return Operation.....	7-7
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.....	7-13
7.4	Local Calls.....	7-14
7.5	System Calls .....	7-15
7.5.1	System Procedure Table .....	7-15
7.5.1.1	Procedure Entries.....	7-17
7.5.1.2	Supervisor Stack Pointer .....	7-17
7.5.1.3	Trace Control Bit .....	7-17
7.5.2	System Call to a Local Procedure .....	7-18
7.5.3	System Call to a Supervisor Procedure .....	7-18
7.6	User and Supervisor Stacks.....	7-19
7.7	Interrupt and Fault Calls .....	7-19
7.8	Returns.....	7-19
7.9	Branch-and-Link .....	7-21
<b>8</b>	<b>Faults.....</b>	<b>8-1</b>
8.1	Fault Handling Overview .....	8-1
8.2	Fault Types.....	8-3
8.3	Fault Table .....	8-5
8.4	Stack Used in Fault Handling .....	8-6



8.5	Fault Record.....	8-6
8.5.1	Fault Record Description.....	8-6
8.5.2	Fault Record Location .....	8-8
8.6	Multiple and Parallel Faults .....	8-9
8.6.1	Multiple Non-Trace Faults on the Same Instruction .....	8-9
8.6.2	Multiple Trace Fault Conditions on the Same Instruction.....	8-9
8.6.3	Multiple Trace and Non-Trace Fault Conditions on the Same Instruction .....	8-9
8.6.4	Parallel Faults .....	8-9
8.6.4.1	Faults on Multiple Instructions Executed in Parallel .....	8-10
8.6.4.2	Fault Record for Parallel Faults.....	8-10
8.6.5	Override Faults.....	8-11
8.6.6	System Error .....	8-12
8.7	Fault Handling Procedures.....	8-13
8.7.1	Possible Fault Handling Procedure Actions .....	8-13
8.7.2	Program Resumption Following a Fault.....	8-14
8.7.2.1	Faults Happening Before Instruction Execution .....	8-14
8.7.2.2	Faults Happening During Instruction Execution .....	8-14
8.7.2.3	Faults Happening After Instruction Execution .....	8-15
8.7.3	Return Instruction Pointer (RIP) .....	8-15
8.7.4	Returning to the Point in the Program Where the Fault Occurred.....	8-15
8.7.5	Returning to a Point in the Program Other Than Where the Fault Occurred .....	8-16
8.7.6	Fault Controls .....	8-16
8.8	Fault Handling Action .....	8-17
8.8.1	Local Fault Call .....	8-17
8.8.2	System-Local Fault Call .....	8-17
8.8.3	System-Supervisor Fault Call.....	8-18
8.8.4	Faults and Interrupts .....	8-18
8.9	Precise and Imprecise Faults.....	8-19
8.9.1	Precise Faults .....	8-19
8.9.2	Imprecise Faults.....	8-19
8.9.3	Asynchronous Faults.....	8-19
8.9.4	No Imprecise Faults (AC.nif) Bit.....	8-20
8.9.5	Controlling Fault Precision .....	8-20
8.10	Fault Reference.....	8-21
8.10.1	ARITHMETIC Faults.....	8-22
8.10.2	CONSTRAINT Faults .....	8-23
8.10.3	MACHINE Faults....	8-24
8.10.4	OPERATION Faults .....	8-26
8.10.5	OVERRIDE Faults.....	8-28
8.10.6	PARALLEL Faults .....	8-29
8.10.7	PROTECTION Faults .....	8-30
8.10.8	TRACE Faults .....	8-35
8.10.9	TYPE Faults .....	8-38
9	<b>Tracing and Debugging .....</b>	9-1
9.1	Trace Controls.....	9-1
9.1.1	Trace Controls (TC) Register .....	9-2
9.1.2	PC Trace Enable Bit and Trace-Fault-Pending Flag .....	9-3
9.2	Trace Modes .....	9-3



9.2.1	Instruction Trace.....	9-3
9.2.2	Branch Trace.....	9-4
9.2.3	Call Trace .....	9-4
9.2.4	Return Trace .....	9-4
9.2.5	Prereturn Trace .....	9-4
9.2.6	Supervisor Trace .....	9-5
9.2.7	Mark Trace .....	9-5
9.2.7.1	Software Breakpoints .....	9-5
9.2.7.2	Hardware Breakpoints .....	9-5
9.2.7.3	Requesting Modification Rights to Hardware Breakpoint Resources .....	9-6
9.2.7.4	Breakpoint Control Register .....	9-7
9.2.7.5	Data Address Breakpoint (DAB) Registers .....	9-8
9.2.7.6	Instruction Breakpoint (IPB) Registers .....	9-9
9.3	Generating a Trace Fault .....	9-10
9.4	Handling Multiple Trace Events.....	9-11
9.5	Trace Fault Handling Procedure .....	9-11
9.5.1	Tracing and Interrupt Procedures.....	9-11
9.5.2	Tracing on Calls and Returns .....	9-11
9.5.2.1	Tracing on Explicit Call.....	9-12
9.5.2.2	Tracing on Implicit Call .....	9-12
9.5.2.3	Tracing on Return from Explicit Call.....	9-13
9.5.2.4	Tracing on Return from Implicit Call: Fault Case.....	9-13
9.5.2.5	Tracing on Return from Implicit Call: Interrupt Case .....	9-13
<b>10</b>	<b>Timers .....</b>	<b>10-1</b>
10.1	Timer Registers .....	10-2
10.1.1	Timer Mode Registers (TMR0, TMR1) .....	10-2
10.1.1.1	Bit 0 - Terminal Count Status Bit (TMRx.tc) .....	10-3
10.1.1.2	Bit 1 - Timer Enable (TMRx.enable).....	10-4
10.1.1.3	Bit 2 - Timer Auto Reload Enable (TMRx.reload) .....	10-4
10.1.1.4	Bit 3 - Timer Register Supervisor Read/Write Control (TMRx.sup).....	10-5
10.1.1.5	Bits 4, 5 - Timer Input Clock Select (TMRx.csel1:0).....	10-5
10.1.2	Timer Count Register (TCR0, TCR1) .....	10-5
10.1.3	Timer Reload Register (TRR0, TRR1) .....	10-6
10.2	Timer Operation .....	10-7
10.2.1	Basic Timer Operation.....	10-7
10.2.2	Load/Store Access Latency for Timer Registers .....	10-8
10.3	Timer Interrupts .....	10-10
10.4	Powerup/Reset Initialization .....	10-10
10.5	Uncommon TCRX and TRRX Conditions .....	10-10
10.6	Timer State Diagram .....	10-11
<b>11</b>	<b>Interrupts .....</b>	<b>11-1</b>
11.1	Overview .....	11-1
11.1.1	The i960® Hx Processor Interrupt Controller.....	11-2
11.2	Software Requirements for Interrupt Handling .....	11-3
11.3	Interrupt Priority .....	11-3
11.4	Interrupt Table .....	11-4
11.4.1	Vector Entries.....	11-5
11.4.2	Pending Interrupts .....	11-5



	11.4.3 Caching Portions of the Interrupt Table .....	11-6
11.5	Interrupt Stack and Interrupt Record.....	11-6
11.6	Managing Interrupt Requests.....	11-8
	11.6.1 External Interrupts .....	11-8
	11.6.2 Non-Maskable Interrupt (NMI#).....	11-8
	11.6.3 Timer Interrupts.....	11-9
	11.6.4 Software Interrupts.....	11-9
	11.6.5 Posting Interrupts .....	11-9
	11.6.5.1 Posting Software Interrupts via sysctl .....	11-9
	11.6.5.2 Posting Software Interrupts Directly in the Interrupt Table .....	11-10
	11.6.5.3 Posting External Interrupts.....	11-10
	11.6.5.4 Posting Hardware Interrupts .....	11-11
	11.6.6 Resolving Interrupt Priority .....	11-11
	11.6.7 Sampling Pending Interrupts in the Interrupt Table.....	11-11
	11.6.8 Interrupt Controller Modes .....	11-13
	11.6.8.1 Dedicated Mode .....	11-13
	11.6.8.2 Expanded Mode .....	11-14
	11.6.8.3 Mixed Mode.....	11-16
	11.6.9 Saving the Interrupt Mask .....	11-16
11.7	External Interface Description .....	11-17
	11.7.1 Pin Descriptions .....	11-17
	11.7.2 Interrupt Detection Options .....	11-18
	11.7.3 Memory-Mapped Control Registers .....	11-19
	11.7.4 Interrupt Control Register (ICON) — SF3 .....	11-20
	11.7.5 Interrupt Mapping Registers (IMAP0-IMAP2).....	11-21
	11.7.5.1 Interrupt Mask (IMSK; SF1) and Interrupt Pending (IPND; SF0) Registers.....	11-23
	11.7.5.2 Interrupt Controller Register Access Requirements .....	11-25
	11.7.5.3 Default and Reset Register Values .....	11-25
11.8	Interrupt Operation Sequence .....	11-26
	11.8.1 Setting Up the Interrupt Controller .....	11-28
	11.8.2 Interrupt Service Routines.....	11-28
	11.8.3 Interrupt Context Switch.....	11-29
	11.8.3.1 Servicing an Interrupt from Executing State.....	11-29
	11.8.3.2 Servicing an Interrupt from Interrupted State .....	11-30
11.9	Optimizing Interrupt Performance .....	11-31
	11.9.1 Interrupt Service Latency .....	11-32
	11.9.2 Features to Improve Interrupt Performance .....	11-32
	11.9.2.1 Vector Caching Option .....	11-32
	11.9.2.2 Caching Interrupt Routines and Reserving Register Frames.....	11-33
	11.9.2.3 Caching the Interrupt Stack.....	11-34
	11.9.3 Base Interrupt Latency .....	11-34
	11.9.4 Maximum Interrupt Latency.....	11-35
	11.9.4.1 Avoiding Certain Destinations for MDU Operations .....	11-38
	11.9.4.2 Masking Integer Overflow Faults for syncf .....	11-38
12	Guarded Memory Unit (GMU) .....	12-1
	12.1 Illegal Access Protection .....	12-3
	12.2 Illegal Access Detection .....	12-3



12.3	GMU Register Description.....	12-4
12.3.1	GMU Control Register .....	12-5
12.3.2	GMU Memory Protect Address and Mask Registers.....	12-6
12.3.2.1	Programming the MPAR and MPMR registers.....	12-8
12.3.3	GMU Memory Detect Upper- and Lower-Bounds Registers .....	12-10
12.3.4	GMU Faults .....	12-13
12.4	GMU Powerup Modes .....	12-13
12.5	GMU Programming Considerations .....	12-14
<b>13</b>	<b>Initialization and System Requirements .....</b>	<b>13-1</b>
13.1	Overview .....	13-1
13.2	Initialization.....	13-2
13.2.1	Reset State Operation .....	13-4
13.2.2	Self Test Function (STEST, FAIL#).....	13-8
13.2.2.1	The STEST Pin .....	13-8
13.2.2.2	External Bus Confidence Test.....	13-8
13.2.2.3	The Fail Pin (FAIL#) .....	13-8
13.2.2.4	IMI Alignment Check and System Error .....	13-9
13.2.2.5	Self Test Failure# Codes .....	13-9
13.3	Architecturally Reserved Memory Space .....	13-10
13.3.1	Initial Memory Image (IMI).....	13-10
13.3.1.1	Initialization Boot Record (IBR) .....	13-13
13.3.1.2	Process Control Block (PRCB).....	13-17
13.3.2	Process PRCB Flow .....	13-19
13.3.2.1	AC Initial Image .....	13-20
13.3.2.2	Fault Configuration Word .....	13-20
13.3.2.3	Instruction Cache Configuration Word .....	13-20
13.3.2.4	Register Cache Configuration Word.....	13-21
13.3.3	Control Table .....	13-22
13.4	Device Identification on Reset.....	13-23
13.4.1	Reinitializing and Relocating Data Structures .....	13-24
13.5	Startup Code Example .....	13-25
13.6	System Requirements .....	13-35
13.6.1	Input Clock (CLKIN) .....	13-35
13.6.2	Power and Ground Requirements (VCC, VSS).....	13-35
13.6.3	VCC5 Pin Requirements .....	13-36
13.6.4	Power and Ground Planes .....	13-37
13.6.5	Decoupling Capacitors .....	13-37
13.6.6	I/O Pin Characteristics.....	13-38
13.6.6.1	Output Pins.....	13-38
13.6.6.2	Input Pins .....	13-38
13.6.7	High Frequency Design Considerations.....	13-39
13.6.7.1	Line Termination.....	13-39
13.6.7.2	Latchup.....	13-40
13.6.7.3	Interference .....	13-41
<b>14</b>	<b>Memory Configuration .....</b>	<b>14-1</b>
14.1	Memory Attributes .....	14-1
14.1.1	Physical Memory Attributes.....	14-1
14.1.1.1	Data Bus Width .....	14-2
14.1.1.2	Burst Accesses.....	14-2



14.1.1.3	Pipelined Read Accesses .....	14-2
14.1.1.4	Wait States .....	14-2
14.1.1.5	READY# and BTERM# Pin Operation .....	14-3
14.1.1.6	Data Bus Parity .....	14-4
14.1.2	Logical Memory Attributes.....	14-4
14.1.2.1	Byte Ordering .....	14-5
14.1.2.2	Logical Memory Region Cacheability .....	14-5
14.2	Programming the Physical Memory Configuration (PMCON) Registers .....	14-7
14.2.1	Bus Control (BCON) Register .....	14-9
14.3	Boundary Conditions for Physical Memory Regions .....	14-10
14.3.1	Internal Memory Locations .....	14-10
14.3.2	Bus Transactions across Region Boundaries .....	14-10
14.3.3	Modifying the PMCON Registers .....	14-11
14.4	Programming the Logical Memory Attributes .....	14-11
14.4.1	Defining the Effective Range of a Logical Memory Template .....	14-13
14.4.2	Selecting the Byte Order .....	14-14
14.4.3	Logical Region Invalidiation Control.....	14-14
14.4.4	Data Caching Enable .....	14-15
14.4.5	Enabling the Logical Memory Template .....	14-15
14.4.6	Initialization .....	14-15
14.4.7	Boundary Conditions for Logical Memory Templates.....	14-16
14.4.7.1	Internal Memory Locations .....	14-16
14.4.7.2	Overlapping Logical Data Template Ranges.....	14-16
14.4.7.3	Accesses across LMT Boundaries .....	14-16
14.4.8	Modifying the LMT Registers.....	14-16
15	<b>External Bus Description .....</b>	15-1
15.1	Overview .....	15-1
15.1.1	Terminology: Requests and Accesses .....	15-2
15.1.1.1	Request.....	15-2
15.1.1.2	Access.....	15-2
15.2	Bus Signals .....	15-3
15.2.1	Bus Clock .....	15-4
15.2.2	Address, Data and Parity Signals .....	15-4
15.2.3	Control Signals .....	15-5
15.2.3.1	Access Start and Finish .....	15-5
15.2.3.2	Wait State Control .....	15-5
15.2.3.3	Data Flow Control .....	15-5
15.2.3.4	Transceiver Control .....	15-6
15.2.3.5	Status Signals .....	15-6
15.3	Basic Bus Transaction .....	15-6
15.3.1	Wait States .....	15-8
15.3.1.1	Internally Generated Wait States .....	15-8
15.3.1.2	Externally Generated Wait States .....	15-10
15.3.2	Burst Accesses .....	15-13
15.3.3	Pipelined Read Accesses .....	15-18
15.3.4	Bus Width.....	15-22
15.3.5	Parity Generation and Checking .....	15-24
15.4	Little or Big Endian Memory Configuration.....	15-29
15.5	Atomic Memory Operations (The LOCK# Signal) .....	15-31
15.6	External Bus Arbitration .....	15-32



	15.6.1	The HOLD and HOLDA Signals .....	15-32
	15.6.2	The BREQ and BSTALL Signals.....	15-33
	15.6.3	Bus Backoff Function (BOFF# Pin) .....	15-34
<b>16</b>	<b>Test Features</b> .....		<b>16-1</b>
16.1	On-Circuit Emulation (ONCE) .....	16-1	
16.1.1	Entering/Exiting ONCE Mode .....	16-1	
16.1.2	ONCE Mode and Boundary-Scan (JTAG) are Incompatible .....	16-2	
16.2	Boundary-Scan (JTAG) .....	16-2	
16.2.1	Boundary-Scan Architecture .....	16-3	
16.2.2	TAP Pins .....	16-4	
16.2.3	Instruction Register .....	16-4	
16.2.3.1	Boundary-Scan Instruction Set.....	16-5	
16.2.4	TAP Test Data Registers.....	16-7	
16.2.4.1	Device Identification Register.....	16-7	
16.2.4.2	Bypass Register .....	16-8	
16.2.4.3	RUNBIST Register .....	16-8	
16.2.4.4	Boundary-Scan Register .....	16-8	
16.2.5	TAP Controller.....	16-11	
16.2.5.1	Test Logic Reset State .....	16-13	
16.2.5.2	Run-Test/Idle State .....	16-13	
16.2.5.3	Select-DR-Scan State .....	16-13	
16.2.5.4	Capture-DR State.....	16-13	
16.2.5.5	Shift-DR State .....	16-14	
16.2.5.6	Exit1-DR State.....	16-14	
16.2.5.7	Pause-DR State .....	16-14	
16.2.5.8	Exit2-DR State.....	16-14	
16.2.5.9	Update-DR State .....	16-15	
16.2.5.10	Select-IR Scan State .....	16-15	
16.2.5.11	Capture-IR State .....	16-15	
16.2.5.12	Shift-IR State .....	16-15	
16.2.5.13	Exit1-IR State .....	16-16	
16.2.5.14	Pause-IR State .....	16-16	
16.2.5.15	Exit2-IR State .....	16-16	
16.2.5.16	Update-IR State.....	16-16	
16.2.6	Boundary-Scan Example.....	16-17	
16.2.7	Boundary-Scan Description Language Example.....	16-21	
<b>A</b>	<b>Considerations for Writing Portable Code</b> .....	<b>A-1</b>	
A.1	Core Architecture .....	A-1	
A.2	Address Space Restrictions .....	A-2	
A.2.1	Reserved Memory .....	A-2	
A.2.2	Initialization Boot Record.....	A-2	
A.2.3	Internal Data RAM.....	A-2	
A.2.4	Instruction Cache .....	A-2	
A.2.5	Data and Data Structure Alignment.....	A-3	
A.3	Reserved Locations in Registers and Data Structures.....	A-4	
A.4	Instruction Set .....	A-4	
A.4.1	Instruction Timing .....	A-4	
A.4.2	Implementation-Specific Instructions.....	A-4	
A.5	Extended Register Set .....	A-5	
A.6	Initialization.....	A-5	



A.7	Memory Configuration .....	A-5
A.8	Interrupts .....	A-5
A.9	Other i960® Hx Processor Implementation-Specific Features .....	A-6
A.9.1	Data Control Peripheral Units.....	A-6
A.9.2	Timers .....	A-6
A.9.3	Guarded Memory Unit (GMU) .....	A-6
A.9.4	Fault Implementation.....	A-7
A.10	Breakpoints .....	A-7
<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>Machine-Level Instruction Formats.....</b>	<b>C-1</b>
C.1	General Instruction Format .....	C-1
C.2	REG Format .....	C-2
C.3	COBR Format .....	C-4
C.4	CTRL Format .....	C-4
C.5	MEM Format .....	C-5
C.5.1	MEMA Format Addressing .....	C-6
C.5.2	MEMB Format Addressing .....	C-6
<b>D</b>	<b>Register and Data Structures.....</b>	<b>D-1</b>
D.1	Registers .....	D-3
D.2	Data Structures .....	D-22
<b>E</b>	<b>Instruction Execution and Performance Optimization .....</b>	<b>E-1</b>
E.1	Internal Processor Structure .....	E-2
E.1.1	Instruction Scheduler (IS).....	E-3
E.1.2	Instruction Flow .....	E-4
E.1.3	Register File (RF) .....	E-5
E.1.4	Execution Unit (EU).....	E-6
E.1.5	Multiply/Divide Unit (MDU) .....	E-6
E.1.6	Address Generation Unit (AGU).....	E-6
E.1.7	Data RAM and Local Register Cache .....	E-7
E.1.8	Data Cache .....	E-7
E.1.8.1	Data Cache Organization .....	E-7
E.1.8.2	Bus Configuration .....	E-8
E.1.8.3	Global Control of the Cache .....	E-8
E.1.8.4	Data Fetch Policy .....	E-8
E.1.8.5	Write Policy .....	E-8
E.1.8.6	Data Cache Coherency .....	E-9
E.1.8.7	BCU Pipeline and Data Cache Interaction .....	E-9
E.1.8.8	BCU Queues and Cache Coherency .....	E-11
E.1.8.9	External I/O and Bus Masters and Cache Coherency.....	E-11
E.2	Parallel Instruction Processing .....	E-11
E.2.1	Parallel Issue.....	E-12
E.2.2	Parallel Execution .....	E-12
E.2.3	Scoreboarding .....	E-14
E.2.3.1	Register Scoreboarding.....	E-15
E.2.3.2	Resource Scoreboarding.....	E-16
E.2.3.3	Prevention of Pipeline Stalls.....	E-16
E.2.4	Processing Units .....	E-16



	E.2.4.1. Execution Unit (EU) .....	E-17
	E.2.4.2. Multiply/Divide Unit (MDU) .....	E-18
	E.2.4.3. Data RAM (DR) .....	E-20
	E.2.4.4. Address Generation Unit (AGU) .....	E-21
	E.2.4.5. Effective Address ( <i>efa</i> ) Calculations .....	E-22
	E.2.4.6. Bus Control Unit (BCU) .....	E-22
	E.2.4.7. Control Pipeline .....	E-24
	E.2.4.8. Unconditional Branches .....	E-24
	E.2.4.9. Conditional Branches .....	E-27
E.2.5	Instruction Cache and Fetch Execution.....	E-27
	E.2.5.1. Instruction Cache Organization.....	E-27
	E.2.5.2. Fetch Strategy .....	E-27
	E.2.5.3. Fetch Latency .....	E-28
	E.2.5.4. Cache Replacement.....	E-29
E.2.6	Micro-flow Execution .....	E-29
	E.2.6.1. Invocation and Execution .....	E-29
	E.2.6.2. Data Movement .....	E-31
	E.2.6.3. Bit and Bit Field .....	E-31
	E.2.6.4. Comparison .....	E-32
	E.2.6.5. Branch .....	E-32
	E.2.6.6. Call and Return .....	E-32
	E.2.6.7. Conditional Faults.....	E-33
	E.2.6.8. Debug .....	E-33
	E.2.6.9. Atomic .....	E-33
	E.2.6.10. Processor Management.....	E-34
E.2.7	Coding Optimizations .....	E-34
	E.2.7.1. Loads and Stores .....	E-35
	E.2.7.2. Multiplication and Division .....	E-35
	E.2.7.3. Advancing Comparisons .....	E-36
	E.2.7.4. Unrolling Loops .....	E-37
	E.2.7.5. Enabling Constant Parallel Issue .....	E-38
	E.2.7.6. Alternating from Side to Side.....	E-38
	E.2.7.7. Branch Prediction .....	E-42
	E.2.7.8. Branch Target Alignment.....	E-42
	E.2.7.9. Replacing Straight-Line Code and Calls .....	E-43
E.2.8	Utilizing On-chip Storage.....	E-43
	E.2.8.1. Instruction Cache .....	E-44
	E.2.8.2. Data Cache .....	E-44
	E.2.8.3. Register Cache.....	E-45
	E.2.8.4. Data RAM .....	E-45
E.2.9	Summary .....	E-46
F	<b>Bus Interface Examples.....</b>	F-1
F.1	Non-Pipelined Burst SRAM Interface .....	F-1
F.1.1	Background .....	F-1
F.1.2	Implementation.....	F-2
F.1.3	Block Diagram .....	F-2
	F.1.3.1. Chip Select Logic .....	F-3
	F.1.3.2. State Machine PLD .....	F-4
	F.1.3.3. Write Enable Generation Logic .....	F-4
F.1.4	Waveforms .....	F-5
	F.1.4.1. Chip Select Generation .....	F-6
	F.1.4.2. Wait State Selection .....	F-7



	F.1.4.3. Output Enable and Write Enable Logic .....	F-7
	F.1.4.4. State Machine Descriptions.....	F-8
F.2	F.1.5 Trade-offs and Alternatives .....	F-11
	F.2 Pipelined SRAM Read Interface .....	F-11
	F.2.1 Block Diagram.....	F-12
	F.2.1.1. Address Latch .....	F-12
	F.2.1.2. State Machine PLD .....	F-13
	F.2.1.3. Write Enable Logic .....	F-13
	F.2.2 Waveforms .....	F-14
	F.2.2.1. State Machines.....	F-14
F.3	F.2.3 Trade-offs and Alternatives .....	F-16
	F.3 Interfacing to Dynamic RAM .....	F-16
	F.3.1 Fast Page Mode DRAM .....	F-17
	F.3.2 DRAM Refresh Modes .....	F-17
	F.3.3 Address Multiplexer Input Connections.....	F-19
	F.3.4 Series Damping Resistors.....	F-19
	F.3.5 System Loading .....	F-20
	F.3.6 DRAM Address Generation.....	F-20
	F.3.7 Memory Ready .....	F-23
	F.3.8 Region Programming .....	F-23
	F.3.9 Design Example: Burst DRAM with Distributed CAS#-Before-RAS# Refresh Using READY# Control.....	F-25
F.4	F.3.10 DRAM Controller State Machine .....	F-26
	F.4 Interleaved Memory Systems.....	F-29
F.5	F.5 Interfacing to Slow Peripherals Using the Internal Wait State Generator .....	F-32
	F.5.1 Implementation.....	F-32
	F.5.2 Schematic .....	F-32
	F.5.3 Waveforms .....	F-34
F.6	F.6 Synchronous Flash Interface .....	F-38

## Glossary

## Index



## Figures

1-1	i960® Hx Processor Functional Block Diagram.....	1-1
2-1	Data Types and Ranges.....	2-1
2-2	Data Placement in Registers.....	2-6
3-1	i960® Hx Processor Programming Environment.....	3-2
3-2	Memory Address Space .....	3-15
3-3	Arithmetic Controls (AC) Register .....	3-21
3-4	Process Controls (PC) Register .....	3-24
4-1	Internal Data RAM and Register Cache .....	4-2
4-2	Cache Control Register (CCON) .....	4-7
5-1	Machine-Level Instruction Formats .....	5-2
6-1	<b>dcctl src1 and src/dst Formats</b> .....	6-39
6-2	Store Data Cache to Memory Output Format.....	6-40
6-3	D-Cache Tag and Valid Bit Formats.....	6-41
6-4	<b>icctl src1 and src/dst Formats</b> .....	6-56
6-5	Store Instruction Cache to Memory Output Format.....	6-58
6-6	I-Cache Set Data, Tag and Valid Bit Formats .....	6-59
6-7	Src1 Operand Interpretation.....	6-108
6-8	<i>src/dst</i> Interpretation for Breakpoint Resource Request .....	6-109
7-1	Procedure Stack Structure and Local Registers .....	7-3
7-2	Frame Spill .....	7-9
7-3	Frame Fill .....	7-10
7-4	System Procedure Table.....	7-16
7-5	Previous Frame Pointer Register (PFP) (r0) .....	7-20
8-1	Fault-Handling Data Structures .....	8-1
8-2	Fault Table and Fault Table Entries .....	8-5
8-3	Fault Record.....	8-7
8-4	Storage of the Fault Record on the Stack .....	8-8
8-5	Parity Fault Record.....	8-25
8-6	MACHINE.PARITY_ERROR Fault Access Type Word Definition (NFP-28).....	8-25
8-7	PROTECTION.BAD_ACCESS Fault Record .....	8-33
8-8	Indicator Word (NFP-32) .....	8-33
8-9	Access Type Fault (NFP-28) .....	8-34
9-1	Trace Controls (TC) Register .....	9-2
9-2	Breakpoint Control Register (BPCON) .....	9-7
9-3	Extended Breakpoint Control Register (XBPCON) .....	9-7
9-4	Data Address Breakpoint (DAB) Register Format.....	9-9
9-5	Instruction Breakpoint (IPB) Register Format.....	9-9
10-1	Timer Functional Diagram .....	10-1
10-2	Timer Mode Register (TMR0, TMR1).....	10-3
10-3	Timer Count Register (TCR0, TCR1).....	10-6
10-4	Timer Reload Register (TRR0, TRR1) .....	10-7
10-5	Timer Unit State Diagram.....	10-12
11-1	Interrupt Handling Data Structures.....	11-2
11-2	Interrupt Table .....	11-4
11-3	Storage of an Interrupt Record on the Interrupt Stack .....	11-7
11-4	Dedicated Mode .....	11-13
11-5	Expanded Mode .....	11-14



11-6	Implementation of Expanded Mode Sources .....	11-15
11-7	Interrupt Sampling.....	11-19
11-8	Interrupt Control (ICON) Register .....	11-20
11-9	Interrupt Mapping (IMAP0-IMAP2) Registers.....	11-22
11-10	Interrupt Pending (IPND) Register .....	11-23
11-11	Interrupt Mask (IMSK) Registers.....	11-24
11-12	Interrupt Controller .....	11-27
11-13	Interrupt Service Flowchart .....	11-31
12-1	Sample Application with Partitions .....	12-2
12-2	GMU Control Register (GCON).....	12-5
12-3	GMU Memory Protect Address Register (MPARx, MPMRx) .....	12-7
12-4	GMU MPAR and MPMR Programming Example .....	12-9
12-5	GMU MPAR and MPMR Programming Example .....	12-10
12-6	GMU Memory Violation Detection Upper and Lower-Bounds Registers.....	12-11
13-1	Processor Initialization Flow.....	13-3
13-2	Cold Reset Waveform .....	13-5
13-3	FAIL# Functional Timing .....	13-9
13-4	Initial Memory Image (IMI) and Process Control Block (PRCB).....	13-12
13-5	PMCON15 Register Bit Description in IBR .....	13-16
13-6	Process Control Block Configuration Words .....	13-18
13-7	Control Table.....	13-22
13-8	IEEE 1149.1 Device Identification Register .....	13-23
13-9	VCCPLL Lowpass Filter .....	13-36
13-10	VCC5 Current-Limiting Resistor.....	13-36
13-11	Reducing Characteristic Impedance .....	13-37
13-12	Series Termination .....	13-40
13-13	AC Termination .....	13-40
13-14	Avoiding Closed-Loop Signal Paths.....	13-41
14-1	PMCON and LMCON Example .....	14-6
14-2	PMCON Register Bit Descriptions .....	14-8
14-3	Bus Control Register (BCON) .....	14-10
14-4	Logical Memory Address Registers (LMAR14:0) .....	14-11
14-5	Logical Memory Mask Registers (LMMR14:0) .....	14-12
14-6	Default Logical Memory Configuration Register (DLMCON).....	14-13
15-1	Basic Read and Write Bus Accesses.....	15-7
15-2	Internal Programmable Wait States .....	15-9
15-3	Bus Request with READY# and BTERM# Control.....	15-11
15-4	Non-Burst, Non-Pipelined Read Request with Wait States.....	15-12
15-5	32-Bit-Wide Data Bus Bursts .....	15-14
15-6	16-Bit Wide Data Bus Bursts.....	15-15
15-7	8-Bit Wide Data Bus Bursts.....	15-15
15-8	32-Bit Bus, Burst, Non-Pipelined, Read Request with Wait States .....	15-16
15-9	32-Bit Bus, Burst, Non-Pipelined, Write Request without Wait States .....	15-17
15-10	Pipelined Read Memory System .....	15-18
15-11	Non-Burst, Pipelined Read Request without Wait States, 32-Bit Bus .....	15-19
15-12	Burst, Pipelined Read Request without Wait States, 32-Bit Bus.....	15-20
15-13	Pipelined to Non-Pipelined Transitions .....	15-21
15-14	Data Width and Byte Enable Encodings .....	15-22
15-15	Parity Error on Non-Burst Access .....	15-25
15-16	Parity Error during Burst Access, No Wait States .....	15-26



15-17	Cycle Type Pin Definition — Non-Burst Access .....	15-27
15-18	Cycle Type Pin Definitions — Burst Access .....	15-27
15-19	The LOCK# Signal .....	15-31
15-20	HOLD/HOLDA Bus Arbitration .....	15-33
15-21	Example Application of the Bus Backoff Function .....	15-35
15-22	Operation of the Bus Backoff Function .....	15-36
16-1	Test Access Port Block Diagram .....	16-3
16-2	TAP Controller State Diagram .....	16-12
16-3	Example Showing Typical JTAG Operations .....	16-18
16-4	Timing Diagram Illustrating the Loading of Instruction Register .....	16-19
16-5	Timing Diagram Illustrating the Loading of Data Register .....	16-20
C-1	Instruction Formats .....	C-1
D-1	AC (Arithmetic Controls) Register .....	D-3
D-2	BCON (Bus Control) Register .....	D-3
D-3	BPCON (Breakpoint Control) Register .....	D-4
D-4	CCON (Cache Control Register) .....	D-4
D-5	DAB (Data Address Breakpoint) Register Format .....	D-4
D-6	DLMCON (Default Logical Memory Configuration) Register .....	D-5
D-7	GCON (GMU Control) Register .....	D-6
D-8	IEEE 1149.1 Device Identification Register .....	D-6
D-9	ICON (Interrupt Control) Register .....	D-7
D-10	IMAP0–IMAP2 (Interrupt Mapping) Registers .....	D-8
D-11	IMSK (Interrupt Mask) Registers .....	D-9
D-12	IPB (Instruction Breakpoint) Register Format .....	D-10
D-13	IPND (Interrupt Pending) Register .....	D-11
D-14	LMAR0–14 (Logical Memory Address) Registers .....	D-12
D-15	LMMR0–14 (Logical Memory Mask Registers) .....	D-13
D-16	MDUB0–5, MDLB0–5 (GMU Memory Violation Detection Upper and Lower-Bounds) Registers .....	D-14
D-17	MPAR0–1, MPMR0–1 (GMU Memory Protect Address Register and Memory Protect Mask Register) .....	D-15
D-18	PC (Process Controls) Register .....	D-16
D-19	PFP (Previous Frame Pointer) Register (r0) .....	D-16
D-20	PMCON0–15 (Physical Memory Configuration) Register .....	D-17
D-21	PMCON15 (Physical Memory Configuration) Register Bit Description in IBR .....	D-18
D-22	TC (Trace Controls) Register .....	D-19
D-23	TCR0–1 (Timer Count Register) .....	D-19
D-24	TMR0–1 (Timer Mode Register) .....	D-20
D-25	TRR0–1 (Timer Reload Register) .....	D-20
D-26	XBPCon (Extended Breakpoint Control) Register .....	D-21
D-27	Control Table .....	D-22
D-28	Fault Table and Fault Table Entries .....	D-23
D-29	Fault Record .....	D-24
D-30	Initial Memory Image (IMI) and Process Control Block (PRCB) .....	D-25
D-31	Interrupt Table .....	D-26
D-32	Procedure Stack Structure and Local Registers .....	D-27
D-33	Process Control Block Configuration Words .....	D-28
D-34	Storage of an Interrupt Record on the Interrupt Stack .....	D-29
D-35	System Procedure Table .....	D-30



E-1	H-Series Core and Peripherals .....	E-1
E-2	i960 Hx Microprocessor Block Diagram .....	E-3
E-3	Instruction Pipeline .....	E-4
E-4	Six-Port Register File .....	E-5
E-5	Data Cache Organization .....	E-7
E-6	BCU and Data Cache Interaction .....	E-9
E-7	Issue Paths .....	E-14
E-8	EU Execution Pipeline .....	E-17
E-9	MDU Execution Pipeline .....	E-18
E-10	MDU Pipelined Back-To-Back Operations .....	E-19
E-11	Data RAM Execution Pipeline .....	E-20
E-12	The Ida Pipeline .....	E-21
E-13	Back-to-Back BCU Accesses .....	E-23
E-14	CTRL Pipeline for Branches to Branches .....	E-24
E-15	Branch in First Executable Group .....	E-25
E-16	Branch in Second Executable Group .....	E-26
E-17	Branch in Third Executable Group .....	E-26
E-18	Fetch Execution .....	E-29
E-19	Micro-Flow Invocation .....	E-30
F-1	32-Bit, Asynchronous, Non-Pipelined Burst SRAM Interface.....	F-3
F-2	Non-Pipelined SRAM Read Waveform .....	F-5
F-3	Non-Pipelined Burst SRAM Write Waveform .....	F-6
F-4	Chip Enable State Machine .....	F-8
F-5	A[3:2] Address Generation State Machine .....	F-9
F-6	Pipelined Read Address and Data Transactions .....	F-11
F-7	Pipelined SRAM Interface Block Diagram.....	F-12
F-8	Pipelined Burst Read Waveform .....	F-14
F-9	Pipelined Read Chip Enable State Machine .....	F-14
F-10	Pipelined Read PA[3:2] State Machine Diagram .....	F-15
F-11	Fast Page Mode DRAM Read .....	F-17
F-12	RAS#-only DRAM Refresh .....	F-18
F-13	CAS#-before-RAS# DRAM Refresh .....	F-18
F-14	Address Multiplexer Inputs .....	F-19
F-15	DRAM Address Generation State Machine .....	F-21
F-16	Fast Page DRAM System Read Waveform .....	F-23
F-17	Fast Page Mode DRAM System Write Waveform .....	F-24
F-18	Memory System Block Diagram .....	F-25
F-19	DRAM State Machine .....	F-26
F-20	Two-Way Interleaved Read Access Overlap .....	F-29
F-21	Two-Way Interleaved Memory System .....	F-30
F-22	Two-Way Interleaved Read Waveforms .....	F-31
F-23	8-bit Interface Schematic .....	F-33
F-24	Read Waveforms .....	F-34
F-25	Write Waveforms .....	F-35
F-26	State Machine Diagram .....	F-36
F-27	Flash Memory System Block Diagram .....	F-38
F-28	Four Double-Word Burst Followed by a Pipelined Two Double-Word Burst Read .....	F-39



## Tables

1-1	i960® Hx Processor Product Description .....	1-2
1-2	Register Terminology Conventions .....	1-9
2-1	Memory Contents for Little and Big Endian Example.....	2-5
2-2	Byte Ordering for Little and Big Endian Accesses.....	2-5
2-3	Memory Addressing Modes.....	2-6
3-1	Registers and Literals Used as Instruction Operands .....	3-3
3-2	Allowable Register Operands .....	3-6
3-3	Access Types .....	3-8
3-4	Supervisor Space Family Registers and Tables .....	3-9
3-5	User Space Family Registers and Tables .....	3-13
3-6	Data Structure Descriptions .....	3-14
3-7	Alignment of Data Structures in the Address Space .....	3-17
3-8	Condition Codes for True or False Conditions .....	3-22
3-9	Condition Codes for Equality and Inequality Conditions .....	3-22
3-10	Condition Codes for Carry Out and Overflow.....	3-23
5-1	80960Hx Instruction Set.....	5-4
5-2	Arithmetic Operations.....	5-7
6-1	Pseudo-Code Symbol Definitions .....	6-4
6-2	Faults Applicable to All Instructions.....	6-4
6-3	Common Faulting Conditions .....	6-5
6-4	Condition Code Mask Descriptions .....	6-8
6-5	concmopo example: register ordering and CC.....	6-36
6-6	<b>dcctl</b> Operand Fields .....	6-38
6-7	<b>dcctl</b> Status Values and D-Cache Parameters.....	6-39
6-8	<b>icctl</b> Operand Fields .....	6-55
6-9	<b>icctl</b> Status Values and Instruction Cache Parameters .....	6-57
6-10	<b>sysctl</b> Field Definitions.....	6-108
6-11	Cache Mode Configuration.....	6-109
7-1	Encodings of Entry Type Field in System Procedure Table .....	7-17
7-2	Encoding of Return Status Field.....	7-21
8-1	i960® Hx Processor Fault Types and Subtypes .....	8-3
8-2	Override Fault Record Format.....	8-12
8-3	Fault Control Bits and Masks .....	8-16
8-4	Access Size/Type Definitions .....	8-25
8-5	Access Size and Type Definitions .....	8-34
9-1	<i>src/dst</i> Encoding.....	9-6
9-2	Configuring the Data Address Breakpoint (DAB) Registers .....	9-8
9-3	Programming the Data Address Breakpoint (DAB) Modes .....	9-8
9-4	Instruction Breakpoint Modes .....	9-10
9-5	Tracing on Explicit Call.....	9-12
9-6	Tracing on Implicit Call .....	9-12
9-7	Tracing on Return from Explicit Call.....	9-13
10-1	Timer Performance Ranges .....	10-2
10-2	Timer Registers .....	10-2
10-3	Timer Input Clock (TCLOCK) Frequency Selection .....	10-5
10-4	Timer Mode Register Control Bit Summary.....	10-8
10-5	Timer Responses to Register Bit Settings .....	10-9
10-6	Timer Powerup Mode Settings .....	10-10



10-7	Uncommon TMRx Control Bit Settings .....	10-11
11-1	Interrupt Control Registers Memory-Mapped Addresses .....	11-19
11-2	Location of Cached Vectors in Internal RAM .....	11-33
11-3	Base Interrupt Latency .....	11-34
11-4	Worst-Case Interrupt Latency Controlled by <b>divo</b> to Destination r15 .....	11-35
11-5	Worst-Case Interrupt Latency Controlled by <b>divo</b> to Destination r3 .....	11-36
11-6	Worst-Case Interrupt Latency Controlled by <b>calls</b> .....	11-36
11-7	Worst-Case Interrupt Latency When Delivering a Software Interrupt .....	11-37
11-8	Worst-Case Interrupt Latency Controlled by <b>flushreg</b> of One Stack Frame.....	11-37
12-1	GMU Memory-Mapped Registers .....	12-4
12-2	MPAR Register Bit Descriptions .....	12-6
12-3	GMU Protected Memory Mask Register Block Sizes.....	12-8
12-4	MDUB Register Bit Descriptions .....	12-12
12-5	GMU Powerup and Reset Values .....	12-13
13-1	Pin Reset State .....	13-6
13-2	Register Values after Reset .....	13-6
13-3	Fail Codes for BIST (bit 7 = 1).....	13-9
13-4	Remaining Fail Codes (bit 7 = 0).....	13-10
13-5	Initialization Boot Record .....	13-13
13-6	PRCB Configuration .....	13-17
13-7	Register Cache Size vs. Available General Purpose RAM .....	13-21
13-8	Fields of IEEE 1149.1 Device ID .....	13-23
13-9	Input Pins .....	13-38
14-1	Region Table Mapping .....	14-7
14-2	PMCON15 Register Values after Reset.....	14-9
14-3	DLMCON Values at Reset .....	14-15
15-1	Bus Controller Signals.....	15-3
15-2	Data, Parity and Byte Enable Associations .....	15-4
15-3	Burst Transfers and Bus Widths .....	15-13
15-4	Byte Enable Encoding .....	15-23
15-5	CT[3:0] Encoding .....	15-28
15-6	Byte Ordering on Bus Transfers.....	15-30
16-1	TAP Controller Pin Definitions .....	16-4
16-2	Boundary-Scan Instruction Set .....	16-5
16-3	IEEE Instructions.....	16-6
16-4	i960 Hx Processor Boundary-Scan Register Bit Order .....	16-9
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 <b>sysctl</b> Operations .....	B-8
B-6	Cycle Counts for <b>icctl</b> Operations.....	B-8
B-7	Cycle Counts for <b>dcctl</b> Operations.....	B-8
B-8	Cycle Counts for <b>intctl</b> Operations .....	B-8
B-9	MEM Format Instruction Encodings .....	B-9
C-1	Instruction Field Descriptions .....	C-2
C-2	Encoding of <i>src1</i> and <i>src2</i> in REG Format.....	C-3
C-3	Encoding of <i>src/dst</i> in REG Format.....	C-3
C-4	Encoding of <i>src1</i> in COBR Format.....	C-4

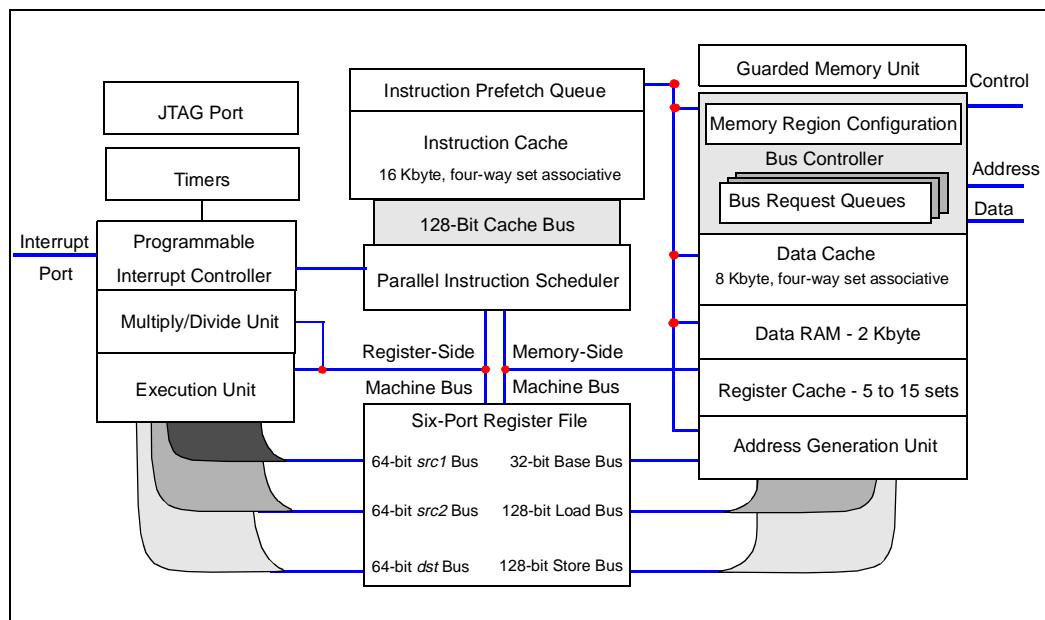


C-5	Encoding of <i>src2</i> in COBR Format .....	C-4
C-6	Addressing Modes for MEM Format Instructions .....	C-5
C-7	Encoding of Scale Field.....	C-7
E-1	BCU Instructions .....	E-10
E-2	Machine Type Sequences that Can Be Issued in Parallel .....	E-13
E-3	Scoreboarded Register Conditions .....	E-15
E-4	Scoreboarded Resource Conditions .....	E-16
E-5	EU Instructions .....	E-17
E-6	MDU Instructions.....	E-19
E-7	Data RAM Instructions .....	E-20
E-8	AGU Instructions .....	E-21
E-9	BCU Instructions .....	E-23
E-10	CTRL Instructions.....	E-24
E-11	Fetch Strategy .....	E-28
E-12	Load Micro-Flow Instruction Issue Clocks.....	E-31
E-13	Store Micro-flow Instruction Issue Clocks .....	E-31
E-14	Bit and Bit Field Micro-flow Instructions .....	E-31
E-15	<b>bx</b> and <b>balx</b> Performance.....	E-32
E-16	<b>callx</b> Performance.....	E-33
E-17	<b>sysctl</b> Performance.....	E-34
E-18	Creative Uses for the <b>ld<sub>a</sub></b> Instruction .....	E-39
E-19	Code Optimization Summary .....	E-46
F-1	Sample Memory Interface Systems .....	F-1



Intel's i960<sup>®</sup> Hx processor provides higher performance levels while maintaining backward compatibility (pin<sup>1</sup> and software) with the i960 CA/CF processors. This member of the family of i960 32-bit, RISC-style, embedded processors allows customers to create scalable designs to meet multiple price and performance points. This easy upgrade path is accomplished by providing processors that can run at the bus speed or faster using Intel's clock multiplying technology ([Table 1-1](#)). The i960 Hx processor is capable of executing 150 million instructions per second, using a sophisticated instruction scheduler that allows the processor to sustain execution of two instructions every core clock with a peak performance of three instructions per clock.

**Figure 1-1. i960<sup>®</sup> Hx Processor Functional Block Diagram**



1. Though not drop-in replaceable. Customers can design systems that accept either i960 Hx or Cx processors.

The three i960 H-series processors differ in the ratio of core clock speed versus the external bus speed:

**Table 1-1. i960® Hx Processor Product Description**

Product	Core	Voltage
80960HA	1x	3.3 V <sup>†</sup>
80960HD	2x	3.3 V <sup>†</sup>
80960HT	3x	3.3 V <sup>†</sup>

<sup>†</sup> The processor inputs are 5 Volt tolerant.

In addition to expanded clock frequency options, the i960 Hx processor provides essential enhancements for an emerging class of high-performance embedded applications. Features include increased instruction cache, data cache and data RAM. It also boasts a 32-bit demultiplexed and pipelined burst bus, fast interrupt mechanism, guarded memory unit, wait state generator, dual programmable timers, ONCE and IEEE 1149.1-compliant boundary-scan test and debug support, and new instructions.

## 1.1 The i960® Processor Family

The i960 processor family is a 32-bit RISC architecture created by Intel especially to serve the needs of embedded applications. The embedded market includes applications as diverse as industrial automation, avionics, medical instrumentation, image processing, graphics and communications. The i960 Hx processor meets the needs of these segments by providing fast event handling, efficient data processing and high-bandwidth packet movement.

Because all members of the i960 processor family share a common core architecture, i960 applications are code compatible. Each new processor in the family adds its own special set of functions to the core to satisfy the needs of a specific application or range of applications in the embedded market.

## 1.2 i960® Hx Processor Key Features

### 1.2.1 Execution Architecture

Resource scoreboarding allows simultaneous multiple instruction execution per clock without conflict. To sustain execution of multiple instructions in each clock cycle, the processor decodes multiple instructions in parallel and simultaneously issues these instructions to parallel processing units. The various processing units are then able to independently access instruction operands in parallel from a common register set.

A local register cache integrated on-chip provides automatic register management on call/return instructions. Upon a call instruction, the processor allocates a set of 16 local registers for the called procedure and stores the previous procedure's registers in the on-chip register cache. As additional procedures are called, the cache stores the associated registers such that the most recently called procedure is the first available by the next return (**ret**) instruction. The processor can store up to fifteen register sets, after which the oldest sets are stored (spilled) into external memory.

The i960 Hx processor supports the architecturally-defined branch prediction mechanism. This feature allows many branches to execute with no pipeline break. The i960 Hx processor's efficient pipeline results in a branch taking as few as zero clocks to execute. The maximum incorrect prediction penalty is two core clocks.

### 1.2.2 Pipelined, Burst Bus

A 32-bit high performance bus controller interfaces the 80960Hx core to the external memory and peripherals. The Bus Control Unit features a maximum transfer rate of 160 Mbytes per second (at an external bus clock frequency of 40 MHz). One of the key advantages of this design is its versatility. The user can program system memory's physical and logical attributes independently. Physical attributes include wait state profile, bus width and parity. Logical attributes include cacheability and big or little endian byte order. Internally programmable wait states and 16 separately configurable physical memory regions allow the processor to interface with a variety of memory subsystems with minimum system complexity. To reduce the effect of wait states, the bus design is decoupled from the core with a buffer queue. This lets the processor execute instructions while the bus performs memory accesses independently.

The bus controller's key features include:

- Demultiplexed, burst bus to support most efficient DRAM access modes
- Address pipelining to reduce memory cost while maintaining performance
- 32-, 16- and 8-bit modes for I/O interfacing ease
- Full internal wait state generation to reduce system cost
- Little and big endian support
- Unaligned access support implemented in hardware
- Three-deep request queue to decouple the bus from the core
- Independent physical and logical address space characteristics

### 1.2.3 On-Chip Caches and Data RAM

As shown in [Figure 1-1](#), the i960 Hx processor provides generous on-chip cache and storage features to decouple CPU execution from the external bus. The processor includes a 16 Kbyte instruction cache, an 8 Kbyte data cache and 2 Kbytes of data RAM. The caches are organized as 4-way set associative. Stores that hit the data cache are written through to memory. The data cache performs write allocation on cache misses. A fifteen-set stack frame cache allows the processor to rapidly allocate and deallocate local registers. The on-chip caches and data RAM sustain a 4-word (128-bit) access every clock cycle.

### 1.2.4 Priority Interrupt Controller

The interrupt controller provides the mechanism for the low latency and high throughput interrupt service essential for embedded applications. A priority interrupt controller provides full programmability of 240 interrupt sources with a typical interrupt task switch (latency) time of 17 bus clocks. The controller supports 31 priority levels. Interrupts are prioritized and signaled within 10 bus clocks of the request. If the interrupt is a higher priority than the processor priority, the context switch to the interrupt routine typically completes in another 7 bus clocks.

External agents post interrupts via the 8-bit external interrupt port. The interrupt controller also handles the two internal sources from the Timers. Interrupts can be level- or edge-triggered.

### 1.2.5 Guarded Memory Unit

The Guarded Memory Unit (GMU) provides memory protection without the address translation found in memory management units. The GMU contains two memory protection schemes:

- preventing illegal memory accesses
- detecting memory access violations

Both signal a fault to the processor. The programmable protection modes are:

- user read, write or execute
- supervisor read, write or execute

### 1.2.6 Dual-Programmable Timers

The processor provides two independent 32-bit timers that can be programmed to count at a rate equal to the bus clock frequency, or the bus clock divided by 2, 4 or 8. The user configures the timers via the Timer Unit registers. These registers are memory-mapped within the i960 Hx processor, addressable on 32-bit boundaries. The timers have a single-shot mode and auto-reload capabilities for continuous operation. Each timer has an independent interrupt request to the processor's interrupt controller.

## 1.3 About this Manual

This *i960® Hx Microprocessor User's Manual* provides detailed programming and hardware design information for the i960 Hx processor. It is written for programmers and hardware designers who understand the basic operating principles of microprocessors and their systems.

This manual does not provide electrical specifications such as DC and AC parametrics, operating conditions and packaging specifications. Such information is found in the *80960HA/HD/HT Embedded 32-bit Microprocessor* datasheet (order number 272495).

For information on other i960 processor family products or the architecture in general, refer to Intel's *Solutions960®* catalog (order number 270791). It lists all current i960 microprocessor family-related documents, support components, boards, software development tools, debug tools and more.

This manual is organized into these chapters and appendices:

- [Chapter 1, “Introduction”](#): Provides a features list of the i960 processor architecture and the i960 Hx processor. It also provides an overview of this manual, notation conventions and a list of additional references for the i960 processor.
- [Chapter 2, “Data Types and Memory Addressing Modes”](#): Describes the processor’s supported data types and addressing modes.
- [Chapter 3, “Programming Environment”](#): Describes the i960 Hx processor’s programming environment including global and local registers, special function registers, control registers, literals, processor-state registers and address space.
- [Chapter 4, “Cache and On-Chip Data RAM”](#): Describes the structure and user configuration of all forms of on-chip storage, including caches (data, local register and instruction) and data RAM.
- [Chapter 5, “Instruction Set Overview”](#): Provides an overview of the i960 microprocessor family’s instruction set and i960 Hx 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”](#): Provides detailed information about each instruction available to the i960 processors. Instructions are listed alphabetically by assembly language mnemonic.
- [Chapter 7, “Procedure Calls”](#): Describes mechanisms for making calls and returns, which include branch-and-link instructions, call instructions (**call**, **callx**, **calls**), return instruction (**ret**) and call actions caused by interrupts and faults.
- [Chapter 8, “Faults”](#): Describes the i960 processor’s fault handling facilities. Subjects covered include the fault handling data structures and fault handling mechanisms.

- [Chapter 9, “Tracing and Debugging”](#): Describes the facilities for runtime activity monitoring.
- [Chapter 10, “Timers”](#): Describes the dual, independent 32-bit timers. Topics include timer registers (TMR<sub>x</sub>, TCR<sub>x</sub> and TRR<sub>x</sub>), timer operation, timer interrupts, and timer register values at initialization.
- [Chapter 11, “Interrupts”](#): Describes the i960 processor core architecture interrupt mechanism and the i960 Hx processor interrupt controller. Key topics include the processor’s facilities for requesting and posting interrupts, the programmer’s interface to the on-chip interrupt controller, implementation, latency and how to optimize interrupt performance.
- [Chapter 12, “Guarded Memory Unit \(GMU\)”](#): Provides information about the Guarded Memory Unit including memory protection and detection schemes and programming the unit’s memory-mapped registers.
- [Chapter 13, “Initialization and System Requirements”](#): Describes the steps performed during initialization. Discussed are the RESET# pin, 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.
- [Chapter 14, “Memory Configuration”](#): Describes how to program the Bus Control Unit (BCU) to control many common types of memory and I/O subsystems.
- [Chapter 15, “External Bus Description”](#): This chapter serves as a guide for the hardware designer when interfacing memory and peripherals to the i960 processors.
- [Chapter 16, “Test Features”](#): Describes the test features, including ONCE (On-Circuit Emulation) and boundary-scan (JTAG).
- [Appendix A, “Considerations for Writing Portable Code”](#): Describes the aspects of the microprocessor that are implementation dependent, and is intended as a guide for writing application code that is directly portable to other i960 processor architecture implementations.
- [Appendix B, “Opcodes and Execution Times”](#): Lists the instruction encoding for each i960 processor instruction.
- [Appendix C, “Machine-Level Instruction Formats”](#): Describes the encoding format for instructions used by the i960 processors. Included is a description of the four instruction formats and how the addressing modes relate to these formats.
- [Appendix D, “Register and Data Structures”](#): Provides a compilation of all register and data structure figures described throughout the manual. Following each figure is a reference that indicates the section that discusses the figure.
- [Appendix E, “Instruction Execution and Performance Optimization”](#): Describes the i960 Hx processors’ core architecture and core features that enhance the processors’ performance and parallelism. This appendix also describes assembly language techniques for achieving the highest instruction-stream performance.
- [Appendix F, “Bus Interface Examples”](#): Describes how to interface the processor to external memory systems. Also discussed are non-pipelined and pipelined burst SRAM, non-pipelined burst DRAM, slow 8-bit memory systems and high performance pipelined burst EPROM.

## 1.4 Notation and Terminology

This section defines terminology and textual conventions that are used throughout the manual.

### 1.4.1 Reserved and Preserved

Certain fields in registers and data structures are described as being either *reserved* or *preserved*:

- A reserved field is one that may be used by other i960 processor architecture implementations. Correct treatment of reserved fields ensures software compatibility with other i960 processors. The processor uses these fields for temporary storage; as a result, the fields sometimes contain unusual values.
- A preserved field is one that the processor does not use. Software may use preserved fields for any function.

Reserved fields in certain data structures must be set to 0 (zero) when the data structure is created. Set reserved fields to 0 when creating the Interrupt Table, Fault Table and System Procedure Table. Software must not modify or rely on these reserved field values after a data structure is created. When the processor creates the Interrupt or Fault Record data structure on the stack, software should not depend on the value of the reserved fields within these data structures.

Some bits or fields in data structures and registers are shown as requiring specific encoding. These fields should be treated as if they were reserved fields. They must be set to the specified value when the data structure is created or when the register is initialized and software must not modify or rely on the value after that.

Reserved bits in the Arithmetic Controls (AC) register can be set to 0 after initialization to ensure compatibility with other i960 processor implementations. Reserved bits in the Process Controls (PC) register and Trace Controls (TC) register must not be initialized. When the AC, PC and TC registers are modified using **modac**, **modpc** or **modtc** instructions, the reserved locations in these registers must be masked.

Certain areas of memory may be referred to as *reserved memory* in this reference manual. Reserved — when referring to memory locations — implies that an implementation of the i960 architecture may use this memory for some special purpose. For example, memory-mapped peripherals might be located in reserved memory areas on future implementations.

## 1.4.2 Specifying Bit and Signal Values

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

The terms *assert* and *deassert* refer to the logically active or inactive value of a signal or bit, respectively. A signal is specified as an active 0 signal by an overbar. For example, the input is active low and is asserted by driving the signal to a logic 0 value.

## 1.4.3 Representing Numbers

All numbers in this manual can be assumed to be base 10 unless designated otherwise. In text, binary numbers are sometimes designated with a subscript 2 (for example,  $001_2$ ). If it is obvious from the context that a number is a binary number, the “2” subscript may be omitted.

Hexadecimal numbers are designated in text with the suffix H (for example, FFFF FF5AH). In pseudo-code action statements in the instruction reference section and occasionally in text, hexadecimal numbers are represented by adding the C-language convention “0x” as a prefix. For example “FF7AH” appears as “0xFF7A” in the pseudo-code.

## 1.4.4 Register Names

Special function registers and several of the global and local registers are referred to by their generic register names, as well as descriptive names which describe their function. The global register numbers are g0 through g15; local register numbers are r0 through r15; special function registers are sf0 through sf4. However, when programming the registers with instruction code, make sure to use the *instruction operand*. i960 microprocessor compilers recognize only the instruction operands listed in [Table 1-2](#). Throughout this manual, the registers’ descriptive names, numbers, operands and acronyms are used interchangeably, as dictated by context.

**Table 1-2. Register Terminology Conventions**

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	
sf0	special function 0	interrupt pending	IPND
sf1	special function 1	interrupt mask	IMSK
sf2	special function 2	cache control	CCON
sf3	special function 3	interrupt control	ICON
sf4	special function 4	GMU control	GCON
0-31		literals	

Groups of bits and single bits in registers and control words are called either *bits*, *flags* or *fields*. These terms have a distinct meaning in this manual:

- bit              Controls a processor function; programmed by the user.
- flag             Indicates status. Generally set by the processor; certain flags are user programmable.
- field            A grouping of bits (bit field) or flags (flag field).

Specific bits, flags and fields in registers and control words are usually referred to by a register abbreviation (in upper case) followed by a bit, flag or field name (in lower case). These items are separated with a period. A position number designates individual bits in a field. For example, the return type (rt) field in the previous frame pointer (PFP) register is designated as "PFP.rt". The least significant bit of the return type field is then designated as "PFP.rt0".

## 1.5 Related Documents

The following is a list of additional documentation that is useful when designing with and programming the i960 Hx processor. Contact your local Intel representative for more information on obtaining Intel documents.

- *80960HA/HD/HT Embedded 32-bit Microprocessor* datasheet  
Intel order number 272495
- *AP-506: Designing for 80960Cx and 80960Hx Compatibility*  
Intel Order No. 272559



# Data Types and Memory Addressing Modes

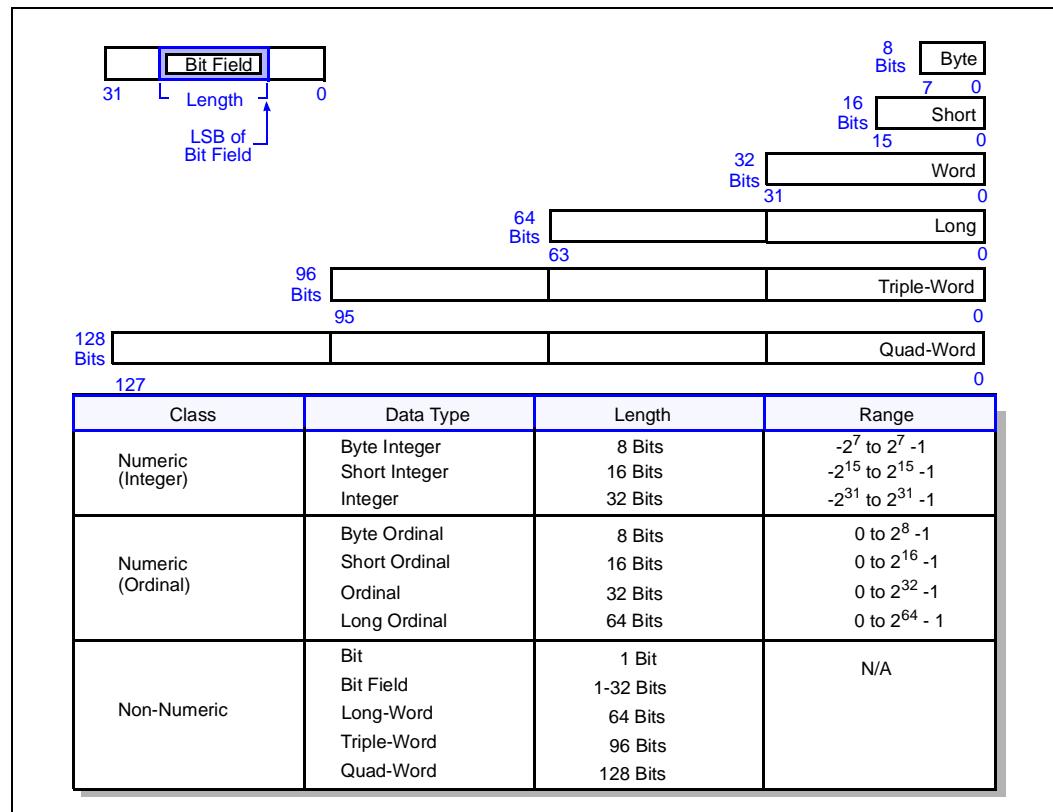
## 2.1 Data Types

The instruction set references or produces several data lengths and formats. The i960<sup>®</sup> Hx 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 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. If 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 8, “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.

## 2.1.2 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; the values may, however, 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 will truncate an ordinal value in a register to fit the destination memory. No overflow condition is signalled in this case.

### 2.1.3 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.

### 2.1.4 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. The 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.

### 2.1.5 Register Data Alignment

Several of the processor's 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). If a register reference for a source value is not properly aligned, the registers that the processor writes to are undefined.

The i960 Hx 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.

## 2.1.6 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 big endian or little endian. The following sections further describe byte ordering.

## 2.2.1 Bit Ordering

Bits within bytes are numbered such that if the byte is viewed as a value, bit 0 is the least significant bit and bit 7 is the most significant bit. For numeric values spanning several bytes, bit numbers higher than 7 indicate successively higher bit numbers in bytes with higher addresses. Unless otherwise noted, bits in illustrations in this manual are ordered such that the higher-numbered bits are to the left.

## 2.2.2 Byte Ordering

The i960 Hx processor can be programmed to use little or big endian byte ordering for memory accesses. Byte ordering refers to how data items larger than one byte are assembled:

- For little endian byte order, the byte with the lowest address in a multi-byte data item has the *least* significance.
- For big endian byte order, the byte with the lowest address in a multi-byte data item has the *most* significance.

For example, [Table 2-1](#) shows eight bytes of data in memory. [Table 2-2](#) shows the differences between little and big endian accesses for byte, short, word and long-word data. [Figure 2-2](#) shows the resultant data placement in registers.

Once data is read into registers, byte order is no longer relevant. The lowest significant bit is always bit 0. The most significant bit is always bit 31 for words, bit 15 for short words, and bit 7 for bytes.

Byte ordering affects the way the i960 Hx processor handles bus accesses. See [Section 14.1.2.1, “Byte Ordering” on page 14-5](#) for more information.

**Table 2-1. Memory Contents for Little and Big Endian Example**

ADDRESS	DATA
1000H	12H
1001H	34H
1002H	56H
1003H	78H
1004H	9AH
1005H	BCH
1006H	DEH
1007H	F0H

**Table 2-2. Byte Ordering for Little and BigEndian Accesses**

Access	Example	Register Contents (Little Endian)	Register Contents (Big Endian)
Byte at 1000H	ldob 0x1000, r3	12H	12H
Short at 1002H	ldos 0x1002, r3	7856H	5678H
Word at 1000H	ld 0x1000, r3	78563412H	12345678H
Long-Word at 1000H	ldl 0x1000, r4	78563412H (r4) F0DEBC9AH (r5)	12345678H (r4) 9ABCDEF0H (r5)

**Figure 2-2. Data Placement in Registers**

Byte	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>31</td><td>24 23</td><td>16 15</td><td>8 7</td><td>0</td></tr> <tr><td>XX</td><td>XX</td><td>XX</td><td>DD<sub>0</sub></td><td></td></tr> </table>	31	24 23	16 15	8 7	0	XX	XX	XX	DD <sub>0</sub>	
31	24 23	16 15	8 7	0							
XX	XX	XX	DD <sub>0</sub>								
Short	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>31</td><td>24 23</td><td>16 15</td><td>8 7</td><td>0</td></tr> <tr><td>XX</td><td>XX</td><td>DD<sub>1</sub></td><td>DD<sub>0</sub></td><td></td></tr> </table>	31	24 23	16 15	8 7	0	XX	XX	DD <sub>1</sub>	DD <sub>0</sub>	
31	24 23	16 15	8 7	0							
XX	XX	DD <sub>1</sub>	DD <sub>0</sub>								
Word	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>31</td><td>24 23</td><td>16 15</td><td>8 7</td><td>0</td></tr> <tr><td>DD<sub>3</sub></td><td>DD<sub>2</sub></td><td>DD<sub>1</sub></td><td>DD<sub>0</sub></td><td></td></tr> </table>	31	24 23	16 15	8 7	0	DD <sub>3</sub>	DD <sub>2</sub>	DD <sub>1</sub>	DD <sub>0</sub>	
31	24 23	16 15	8 7	0							
DD <sub>3</sub>	DD <sub>2</sub>	DD <sub>1</sub>	DD <sub>0</sub>								

**NOTES:**  
 D's are data transferred to/from memory  
 X's are zeros for ordinal data  
 X's are sign bit extensions for integer data

## 2.3 Memory Addressing Modes

The processor provides nine modes for addressing operands in memory. Each addressing mode is used to reference a byte location in the processor's address space. **Table 2-3** shows the memory addressing modes and a brief description of each mode's address elements and assembly code syntax.

**Table 2-3. Memory Addressing Modes**

Mode	Description	Assembler Syntax	Inst. Type
Absolute <i>offset</i>	offset (smaller than 4096)	exp	MEMA
	<i>displacement</i> (larger than 4095)	exp	MEMB
Register Indirect <i>with offset</i>	abase	(reg)	MEMA
	abase + offset	exp (reg)	MEMA
	abase + displacement	exp (reg)	MEMB
	abase + (index*scale)	(reg) [reg*scale]	MEMB
	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](#) in Appendix B 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 2-3](#)). 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 MEMB 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](#) 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 C, “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).

ldisxyz[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:
    cmpble 0,r3,.I34       # Store next array elements if
                           # index is not 0
    ret

```



This chapter describes the i960® Hx processor's programming environment including global and local registers, special function registers, control registers, literals, processor-state registers and address space.

## 3.1 Overview

The i960 processor 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, special function registers, control registers and a set of processor state registers.

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

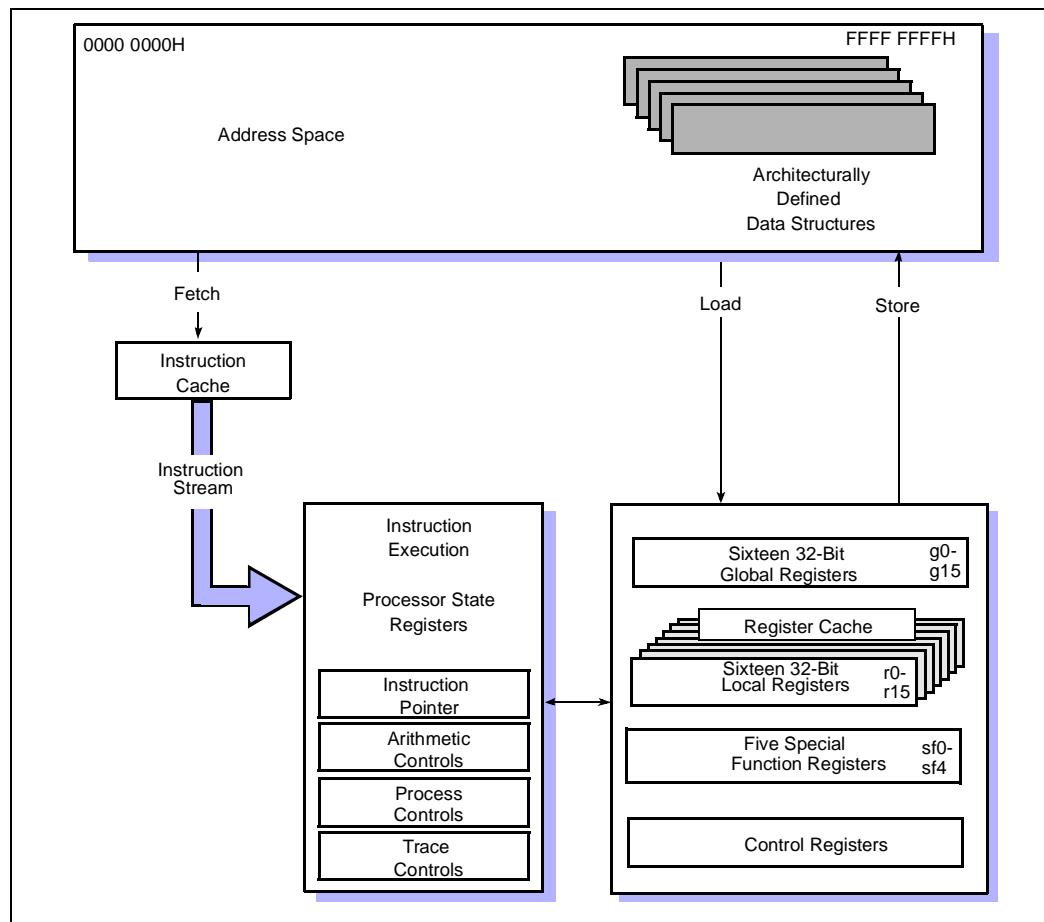
- interrupt stack
- control table
- system procedure table
- local stack
- fault table
- process control block
- supervisor stack
- interrupt table
- initialization boot record

## 3.2 Registers and Literals as Instruction Operands

With the exception of a few special instructions, the i960 Hx 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, five special function 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; special function registers are sf0, sf1 and sf2sf0 through sf4. 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® Hx 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	
sf0	special function 0	interrupt pending	IPND
sf1	special function 1	interrupt mask	IMSK
sf2	special function 2	data cache control	CCON
sf3	special function 3	interrupt control	ICON
sf4	special function 4	GMU control	GCON
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 [Chapter 7, “Call and Return Mechanism”](#) for a description of the FP and procedure stack.

After the processor is reset, register g0 contains device identification and stepping information. The Device Identification sections in the *80960HA/HD/HT Embedded 32-bit Microprocessor* datasheet describe information contained in g0. g0 retains this information until it is written over by the user program. The device identification and stepping information is also stored in the memory-mapped DEVICEID register located at FF008710H.

### 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.

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 Special Function Registers (SFRs)

The i960 architecture provides a mechanism to expand its architecturally-defined register set with up to 32 additional 32-bit registers. On the i960 Hx processor, five special function registers (SFRs) provide an extension to the architectural register model. These registers are designated sf0 – sf4 (see [Table 3-1](#)). Registers sf5 – sf31 are not implemented on the i960 Hx processor. Do not attempt to read or modify unimplemented registers. SFRs provide a means to configure and monitor the interrupt controller, D-cache and GMU.

The processor provides a mechanism that allows modification of SFRs in supervisor mode only. These registers can be accessed only while the processor is in supervisor execution mode. See [Section 3.7, “User-Supervisor Protection Model” on page 3-26](#). A TYPE.MISMATCH fault occurs if an instruction with an SFR operand is executed in user mode.

SFRs are not used as operands for instructions whose machine-level instruction format is of type MEM or CTRL. Such instructions include loads, stores and those that cause program redirection (call, return and branches). [Appendix C, “Machine-Level Instruction Formats”](#) describes machine-level encoding for operands. [Table 3-2](#) summarizes the use of SFRs as instruction operands.

### 3.2.4 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.

Register scoreboarding is implemented for global and local registers but not for SFRs. When an SFR is the destination of a multi-cycle instruction, the programmer must prevent access to the SFR until the multi-clock instruction returns a result to the SFR.

#### Example 3-1. Register Scoreboarding

```

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

```

### 3.2.5 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.

### 3.2.6 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 3 or 4 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, special function registers and literals are used directly as instruction operands. [Table 3-2](#) lists instruction operands for each machine-level instruction format and the positions that can be filled by each register or literal.

**Table 3-2. Allowable Register Operands**

		Operand (1)			
Instruction Encoding	Operand Field	Local Register	Global Register	Literal	SFR (2)
REG	<i>src1</i>	X	X	X	X
	<i>src2</i>	X	X	X	X
	<i>src/dst</i> (as <i>src</i> )	X	X	X	X
	<i>src/dst</i> (as <i>dst</i> )	X	X		X
	<i>src/dst</i> (as both)	X	X		
MEM	<i>src/dst</i>	X	X		
	<i>abase</i>	X	X		
	<i>index</i>	X	X		
COBR	<i>src1</i>	X	X	X	
	<i>src2</i>	X	X	X	
	<i>dst</i>	X (3)	X (3)	X (3)	

**NOTES:**

1. X" denotes the register can be used as an operand in a particular instruction field.
2. Special Function Registers (SFRs) cannot be used in the *src/dst* field of REG format instructions that use this field as both source and destination (e.g., **extract** and **modify**).
3. The COBR destination operands apply only to TEST instructions.

## 3.3 Memory-Mapped Control Registers

The i960 Hx processor gives software the interface to easily read and modify internal control registers. Each of these registers is accessed as a memory-mapped, 32-bit register with a unique memory address. The processor ensures that accesses to MMRs do not generate external bus cycles.

### 3.3.1 Memory-Mapped Registers (MMR)

Portions of the i960 Hx processor address space (addresses FF00 0000H through FFFF FFFFH) are reserved for memory-mapped registers (see [Section 13.3, “Architecturally Reserved Memory Space” on page 13-10](#)). These memory-mapped registers (MMRs) are accessed through word-operand memory instructions (**atmod**, **atadd**, **sysctl**, **Id** 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 3-4](#) and [Table 3-5](#) show all the memory-mapped registers and the application modes of access.

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 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 access they allow. To ensure correct operation, the access type restrictions for each register should be followed. The access type columns of [Table 3-4](#) and [Table 3-5](#) 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.

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 Hx processor; the breakpoints cannot be read using a **ld** instruction.

At initialization, the control table automatically loads 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 13, "Initialization and System Requirements"](#).

### 3.3.1.2 Access Faults

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 will cause 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.

**Table 3-3. 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 an <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. See <a href="#">Section 9.2.7.2, “Hardware Breakpoints” on page 9-5</a> 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. An <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 3-4. Supervisor Space Family Registers and Tables (Sheet 1 of 4)**

Register Name	Memory-Mapped Address	Access Type
<b>Guarded Memory Unit (GMU)</b>		
GCON GMU Control Register (sf4)	FF00 8000H	R/W
Reserved	FF00 8004H to FF00 800FH	—
(MPAR0) Memory Protection Address Register 0	FF00 8010H	R/W
(MPMR0) Memory Protection Mask Register 0	FF00 8014H	R/W
(MPAR1) Memory Protection Address Register 1	FF00 8018H	R/W
(MPMR1) Memory Protection Mask Register 1	FF00 801CH	R/W
Reserved	FF00 8020H TO FF00 807FH	—
(MDUB0) Memory Detect Upper Bounds Register 0	FF00 8080H	R/W
(MDLB0) Memory Detect Lower Bounds Register 0	FF00 8084H	R/W
(MDUB1) Memory Detect Upper Bounds Register 1	FF00 8088H	R/W
(MDLB1) Memory Detect Lower Bounds Register 1	FF00 808CH	R/W
(MDUB2) Memory Detect Upper Bounds Register 2	FF00 8090H	R/W
(MDLB2) Memory Detect Lower Bounds Register 2	FF00 8094H	R/W
(MDUB3) Memory Detect Upper Bounds Register 3	FF00 8098H	R/W
(MDLB3) Memory Detect Lower Bounds Register 3	FF00 809CH	R/W
(MDUB4) Memory Detect Upper Bounds Register 4	FF00 80A0H	R/W
(MDLB4) Memory Detect Lower Bounds Register 4	FF00 80A4H	R/W
(MDUB5) Memory Detect Upper Bounds Register 5	FF00 80A8H	R/W
(MDLB5) Memory Detect Lower Bounds Register 5	FF00 80ACH	R/W
Reserved	FF00 80B0H to FF00 80FFH	—
<b>Logical Memory Configuration LMCON Registers</b>		
(DLMCON) Default Logical Memory Configuration Register	FF00 8100H	R/W
Reserved	FF00 8104H	—
(LMADR0) Logical Memory Address Register 0	FF00 8108H	R/W
(LMMR0) Logical Memory Mask Register 0	FF00 810CH	R/W
(LMADR1) Logical Memory Address Register 1	FF00 8110H	R/W

NOTE: Shaded rows indicate reserved areas.

**Table 3-4. Supervisor Space Family Registers and Tables (Sheet 2 of 4)**

Register Name	Memory-Mapped Address	Access Type
(LMMR1) Logical Memory Mask Register 1	FF00 8114H	R/W
(LMADR2) Logical Memory Address Register 2	FF00 8118H	R/W
(LMMR2) Logical Memory Mask Register 2	FF00 811CH	R/W
(LMADR3) Logical Memory Address Register 3	FF00 8120H	R/W
(LMMR3) Logical Memory Mask Register 3	FF00 8124H	R/W
(LMADR4) Logical Memory Address Register 4	FF00 8128H	R/W
(LMMR4) Logical Memory Mask Register 4	FF00 812CH	R/W
(LMADR5) Logical Memory Address Register 5	FF00 8130H	R/W
(LMMR5) Logical Memory Mask Register 5	FF00 8134H	R/W
(LMADR6) Logical Memory Address Register 6	FF00 8138H	R/W
(LMMR6) Logical Memory Mask Register 6	FF00 813CH	R/W
(LMADR7) Logical Memory Address Register 7	FF00 8140H	R/W
(LMMR7) Logical Memory Mask Register 7	FF00 8144H	R/W
(LMADR8) Logical Memory Address Register 8	FF00 8148H	R/W
(LMMR8) Logical Memory Mask Register 8	FF00 814CH	R/W
(LMADR9) Logical Memory Address Register 9	FF00 8150H	R/W
(LMMR9) Logical Memory Mask Register 9	FF00 8154H	R/W
(LMADR10) Logical Memory Address Register 10	FF00 8158H	R/W
(LMMR10) Logical Memory Mask Register 10	FF00 815CH	R/W
(LMADR11) Logical Memory Address Register 11	FF00 8160H	R/W
(LMMR11) Logical Memory Mask Register 11	FF00 8164H	R/W
(LMADR12) Logical Memory Address Register 12	FF00 8168H	R/W
(LMMR12) Logical Memory Mask Register 12	FF00 816CH	R/W
(LMADR13) Logical Memory Address Register 13	FF00 8170H	R/W
(LMMR13) Logical Memory Mask Register 13	FF00 8174H	R/W
(LMADR14) Logical Memory Address Register 14	FF00 8178H	R/W
(LMMR14) Logical Memory Mask Register 14	FF00 817CH	R/W
Reserved	FF00 8180H to FF00 83FFH	—

**NOTE:** Shaded rows indicate reserved areas.

**Table 3-4. Supervisor Space Family Registers and Tables (Sheet 3 of 4)**

Register Name	Memory-Mapped Address	Access Type
<b>Breakpoint</b>		
(IPB0) Instruction Address Breakpoint Register 0	FF00 8400H	Sysctl- RwG/WwG
(IPB1) Instruction Address Breakpoint Register 1	FF00 8404H	Sysctl- RwG/WwG
(IPB2) Instruction Address Breakpoint Register 2	FF00 8408H	Sysctl- RwG/WwG
(IPB3) Instruction Address Breakpoint Register 3	FF00 840CH	Sysctl- RwG/WwG
(IPB4) Instruction Address Breakpoint Register 4	FF00 8410H	Sysctl- RwG/WwG
(IPB5) Instruction Address Breakpoint Register 5	FF00 8414H	Sysctl- RwG/WwG
Reserved	FF00 8418H to FF00 841FH	—
(DAB0) Data Address Breakpoint Register 0	FF00 8420H	R/W, WwG
(DAB1) Data Address Breakpoint Register 1	FF00 8424H	R/W, WwG
(DAB2) Data Address Breakpoint Register 2	FF00 8428H	R/W, WwG
(DAB3) Data Address Breakpoint Register 3	FF00 842CH	R/W, WwG
(DAB4) Data Address Breakpoint Register 4	FF00 8430H	R/W, WwG
(DAB5) Data Address Breakpoint Register 5	FF00 8434H	R/W, WwG
Reserved	FF00 8438H to FF00 843FH	—
(BPCON) Breakpoint Control Register	FF00 8440H	WwG
(XBPCON) Extended Breakpoint Control Register	FF00 8444H	WwG
Reserved	FF00 8448H to FF00 84FFH	—
<b>Interrupts</b>		
(IPND) Interrupt Pending Register	FF00 8500H	R/W
(IMSK) Interrupt Mask Register	FF00 8504H	R/W
Reserved	FF00 8508H to FF00 850FH	—
(ICON) Interrupt Control Word	FF00 8510H	R/W
Reserved	FF00 8514H to FF00 851FH	—
(IMAP0) Interrupt Map Register 0	FF00 8520H	R/W
(IMAP1) Interrupt Map Register 1	FF00 8524H	R/W
(IMAP2) Interrupt Map Register 2	FF00 8528H	R/W
Reserved	FF00 852CH to FF00 85FFH	—

**NOTE:** Shaded rows indicate reserved areas.

**Table 3-4. Supervisor Space Family Registers and Tables (Sheet 4 of 4)**

Register Name	Memory-Mapped Address	Access Type
<b>Physical Memory Configuration PMCON Registers</b>		
(PMCON0) Physical Memory Control Register 0	FF00 8600H	R/W
(PMCON1) Physical Memory Control Register 1	FF00 8604H	R/W
(PMCON2) Physical Memory Control Register 2	FF00 8608H	R/W
(PMCON3) Physical Memory Control Register 3	FF00 860CH	R/W
(PMCON4) Physical Memory Control Register 4	FF00 8610H	R/W
(PMCON5) Physical Memory Control Register 5	FF00 8614H	R/W
(PMCON6) Physical Memory Control Register 6	FF00 8618H	R/W
(PMCON7) Physical Memory Control Register 7	FF00 861CH	R/W
(PMCON8) Physical Memory Control Register 8	FF00 8620H	R/W
(PMCON9) Physical Memory Control Register 9	FF00 8624H	R/W
(PMCON10) Physical Memory Control Register 10	FF00 8628H	R/W
(PMCON11) Physical Memory Control Register 11	FF00 862CH	R/W
(PMCON12) Physical Memory Control Register 12	FF00 8630H	R/W
(PMCON13) Physical Memory Control Register 13	FF00 8634H	R/W
(PMCON14) Physical Memory Control Register 14	FF00 8638H	R/W
(PMCON15) Physical Memory Control Register 15	FF00 863CH	R/W
Reserved	FF00 8640H to FF00 86FBH	—
<b>Bus Configuration BCON Registers</b>		
(BCON) Bus Configuration Control Register	FF00 86FCH	R/W
<b>Process Control Block Pointer</b>		
(PRCB) Processor Control Block Pointer	FF00 8700H	RO
(ISP) Interrupt Stack Pointer	FF00 8704H	R/W
(SSP) Supervisor Stack Pointer	FF00 8708H	R/W
Reserved	FF00 870CH	—
<b>Device Identification Register</b>		
(DEVICEID) i960 Hx processor Device ID	FF00 8710H	RO
Reserved	FF00 8714H to FFFF FFFFH	—

**NOTE:** Shaded rows indicate reserved areas.

**Table 3-5. User Space Family Registers and Tables**

Register Name	Memory-Mapped Address	Access Type
<b>Timers</b>		
Reserved	FF00 0000H to FF00 02FFH	—
(TRR0) Timer Reload Register 0	FF00 0300H	R/W
(TCR0) Timer Count Register 0	FF00 0304H	R/W
(TMR0) Timer Mode Register 0	FF00 0308H	R/W
Reserved	FF00 030CH	—
(TRR1) Timer Reload Register 1	FF00 0310H	R/W
(TCR1) Timer Count Register 1	FF00 0314H	R/W
(TMR1) Timer Mode Register 1	FF00 0318H	R/W
Reserved	FF00 031CH to FF00 7FFFFH	—

## 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 3-6](#) defines the data structures and references other sections of this manual where detailed information can be found.

The i960 Hx 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.

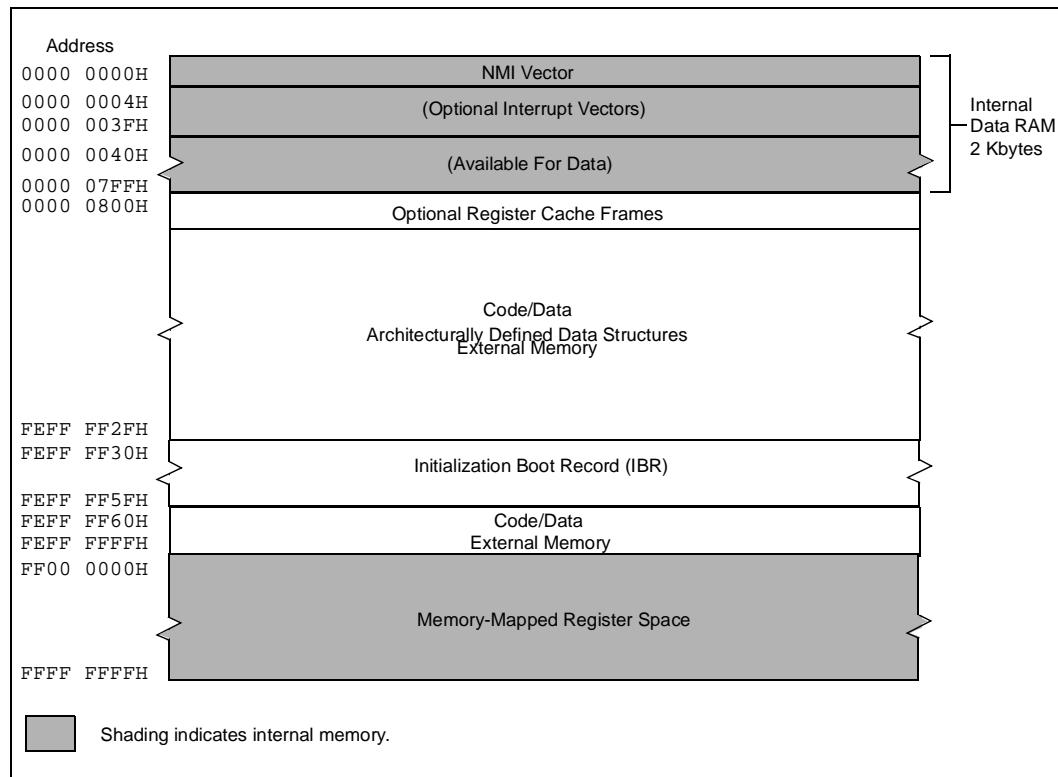
**Table 3-6. Data Structure Descriptions**

Structure (see also)	Description
<b>User and Supervisor Stacks</b> <a href="#">Section 7.6, "User and Supervisor Stacks" on page 7-19</a>	The processor uses these stacks when executing application code.
<b>Interrupt Stack</b> <a href="#">Section 11.5, "Interrupt Stack and Interrupt Record" on page 11-6</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 3-26</a> <a href="#">Section 7.5, "System Calls" on page 7-15</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 11.4, "Interrupt Table" on page 11-4</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 8.3, "Fault Table" on page 8-5</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 13.3.3, "Control Table" on page 13-22</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 Hx processor's 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. Memory Address Space**



The processor treats the entire address space as continuous and linear. There are no subdivisions of the address space into arbitrary segments or dedicated I/O space. The Guarded Memory Unit (GMU) can optionally restrict access to certain areas of memory, such as to protect a kernel's code, data and stack. See [Chapter 12, “Guarded Memory Unit \(GMU\)”](#) for more information. Applications can use an external Memory Management Unit (MMU) to subdivide memory into pages if desired.

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) are reserved for implementation-specific functions. i960 Hx processor programs cannot use this address space except for accesses to memory-mapped registers. As shown in [Figure 3-2](#), the initialization boot record is located just below the i960 Hx processor's reserved memory.

The i960 Hx processor requires some special consideration when using the lower 2 Kbytes of address space (addresses 0000H through 07FFH). 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 3-7](#).
- Align instruction operands for the atomic instructions (**atadd**, **atmod**) to word boundaries in memory.

The i960 Hx processor can perform unaligned load or store accesses. The processor handles a non-aligned load or store request by:

- Automatically servicing a non-aligned memory access with dedicated on-chip circuitry as described in [Section 14.3.2, “Bus Transactions across Region Boundaries” on page 14-10](#).
- After the access is completed, 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 13.3.1.2, “Process Control Block \(PRCB\)” on page 13-17](#).

**Table 3-7. 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**.

Normally 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.

The i960 Hx processor also provides an option for ordering bytes in the opposite manner in memory. The block's most significant byte is stored at the base address and the less significant bytes are stored at successively higher addresses. This byte-ordering scheme, referred to as "big endian", applies to data blocks which are short-words or words. For more about byte ordering, see [Section 14.4.2, "Selecting the Byte Order" on page 14-14](#).

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

Internal data RAM is mapped to the lower 2 Kbytes (0000H to 07FFH) 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. The lower 2 Kbytes of memory is data memory only. Instructions cannot be fetched from the internal data RAM. Instruction fetches directed to the data RAM cause a TYPE.MISMATCH fault to occur.

Some internal data RAM locations are reserved for functions other than general data storage ([Table 3-7](#)). 64 bytes of data RAM may be used to cache specific interrupt vectors. The word at location 0000H is always reserved for the cached NMI vector. With the exception of the cached NMI vector, software can use other reserved portions of the data RAM for data storage when the alternate function is not used.

As described in [Section 13.3.1.2, "Process Control Block \(PRCB\)" on page 13-17](#), local register cache size is specified by the value of the Process Control Block's Register Cache Configuration Word. The first five local register sets are cached internally; if more than five sets are to be cached, the local register cache can be extended into the internal data RAM. Up to ten more sets, occupying up to 640 bytes of data RAM, can be used. When the local register cache is extended, each new register set consumes 16 words of internal data RAM beginning at the lowest data RAM address. The user program is responsible for preventing corruption of the internal RAM areas set aside for the register cache. See [Chapter 7, "Procedure Calls"](#).

Internal RAM's first 256 bytes (0000H to 00FFH) are user mode write protected. This data RAM can be read while executing in user or supervisor mode; however, RAM can be modified only in supervisor mode. Writes to these locations while in user mode generate a TYPE.MISMATCH fault. This feature provides supervisor protection for Interrupt functions that use internal RAM. See [Section 3.7, “User-Supervisor Protection Model” on page 3-26](#). User mode write protection is optionally selected for the rest of the data RAM (0100H to 07FFH) by setting the Bus Configuration Register (BCON) RAM protection bit.

### 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 frees more bus bandwidth for data operations in external memory. Since the internal clocks on the i960 HD and HT processors are multiples of the external memory bus clock, applications must use the instruction cache to take full advantage of those processors' power.

The i960 Hx processor's instruction cache is a 16-Kbyte, four-way set-associative cache, organized as four sets of 128-word lines. The organization permits the cache to deliver up to four 32-bit instruction words to the superscalar processor core with each cycle.

To optimize cache updates when branches or interrupts occur, each eight-byte line has a separate valid bit. Cache misses cause the processor to issue either double- or quad-word fetches to update the cache. For a thorough discussion of instruction cache operation, see [Chapter 4, “Cache and On-Chip Data RAM.”](#)

i960 Hx processor's do not implement bus snooping. 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 moving code from a backplane bus or a disk.

To achieve cache coherence, instruction cache contents can be invalidated after code modification is complete. The **sysctl** instruction is used to invalidate the instruction cache for the i960 Hx component. **sysctl** is issued with an invalidate-instruction-cache message type. See [Section 6.2.67, “sysctl” on page 6-108](#).

The user program is responsible for synchronizing a program with code modification and cache invalidation. In general, a program must ensure that modified code space is not accessed until cache has been invalidated.

The instruction cache can be disabled, causing all instruction fetches to be directed to external memory. Disabling the instruction cache is useful for debugging or monitoring a system at the instruction prefetch level. To disable the instruction cache, execute **sysctl** with the configure-instruction-cache message.

Disabling the instruction cache also disables all instruction fetch queues within the instruction fetch unit. With the instruction cache disabled, every instruction generates an external bus access. Small code loops cannot hide in the buffers. This behavior contrasts with the so-called “baby cache” behavior of the i960 Cx processor.

The processor can be directed to load a block of instructions into the cache and then disable all normal updates to this load cache portion. This cache load-and-lock mechanism optimizes interrupt latency and throughput. The first instructions of time-critical interrupt routines are loaded into the locked cache. The interrupt, when serviced, is directed to the locked cache portion. No external accesses are required for these instructions when the interrupt is serviced.

When bit 1 of an interrupt vector is set to 1, the interrupts fetch instructions from the instruction cache’s locked portion. Execution continues from the locked cache until a miss occurs, such as a branch, call or return to code outside of the locked space. If an interrupt directed to the locked cache results in a miss, the processor fetches the targeted instruction from the normal memory hierarchy.

Any number of 4-Kbyte ways can be configured to load and lock. With only a portion of the cache loaded and locked, the remaining portion acts as a normal three-, two- or one-way cache. Normally, an application locks only one or two ways of the cache. Locking the entire cache forces all instruction fetches (except interrupts directed to the locked cache) to come from external memory. See [Section 4.4.3, “Loading and Locking Instructions in the Instruction Cache” on page 4-5](#) for more details.

The **icctl** instruction is the preferred method to control the i960 Hx processor instruction cache. In order to maintain backwards compatibility with the i960 Cx processors, the **sysctl** instruction does allow limited control of the cache. Using **sysctl** for i960 Hx processor designs is discouraged.

To load and lock all or part of the instruction cache, issue an **icctl** instruction with a load-and-lock message type. When the lock option is selected, an address is specified that points to a memory block to be loaded into the locked cache.

### 3.5.6 Data Cache

The data cache on the i960 Hx processor is a write-through 8-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.

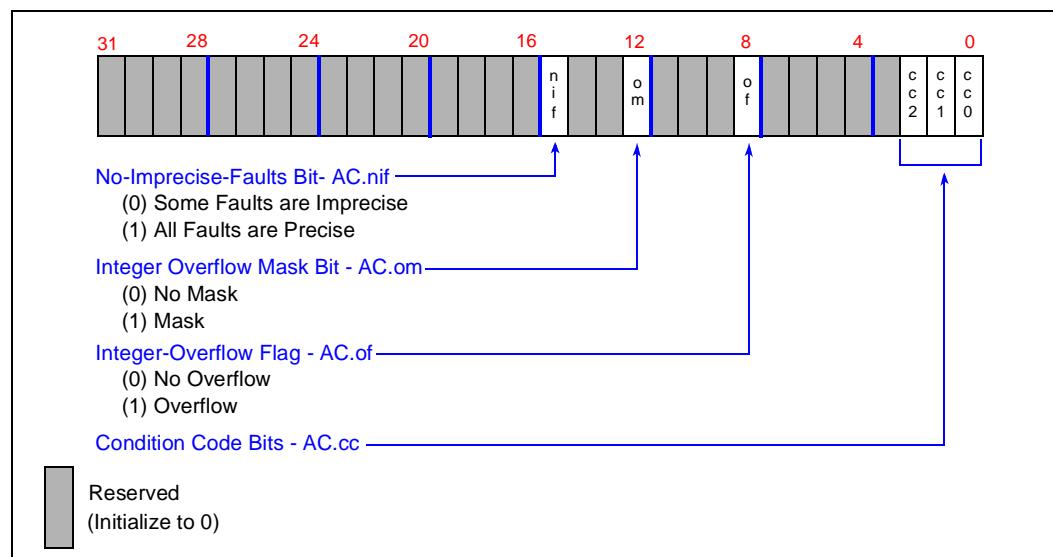
The IP register cannot be read directly. However, the IP-with-displacement addressing mode lets software use the IP as an offset into the address space. This addressing mode can also be used with the **Ida** (load address) instruction to read the current IP value.

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 (AC) Register

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

**Figure 3-3. Arithmetic Controls (AC) Register**



### 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 13, “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 3-8](#). To show equality and inequalities, the processor sets the condition code flags as shown in [Table 3-9](#).

**Table 3-8. Condition Codes for True or False Conditions**

Condition Code	Condition
$010_2$	true
$000_2$	false

**Table 3-9. Condition Codes for Equality and Inequality Conditions**

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

The term *unordered* is used when comparing floating point numbers. The i960 Hx 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 3-10](#).

**Table 3-10. Condition Codes for Carry Out and Overflow**

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

Certain instructions, such as the branch-if instructions, use a 3-bit mask to evaluate the condition code flags. For example, the branch-if-greater-or-equal instruction (**bge**) uses a mask of  $011_2$  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 ( $011_2$ ), less-or-equal ( $110_2$ ) and not-equal ( $101_2$ ). The mask is part of the instruction opcode; the instruction performs a bitwise AND of 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.

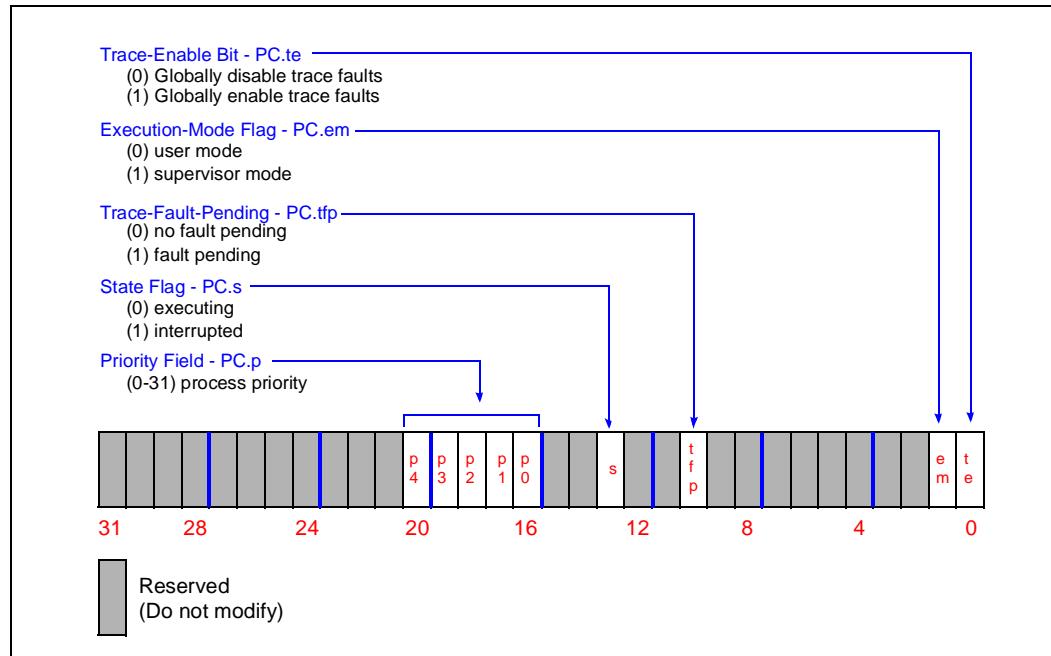
Once the processor sets this flag, the flag remains set until the application software clears it. Refer to the discussion of the ARITHMETIC.INTEGER\_OVERFLOW fault in [Chapter 8, “Faults”](#) for more information about the integer overflow mask bit and flag.

The *no imprecise faults (AC.nif) bit* (bit 15) determines whether or not faults are allowed to be imprecise. If set, all faults (except a synchronous fault such as MACHINE.PARITY) are required to be precise; if clear, certain faults can be imprecise. See [Section 8.9, “Precise and Imprecise Faults” on page 8-19](#) for more information. When set, the AC.nif bit disables the superscalar (parallel instruction execution) feature of the processor; therefore, no imprecise faults mode should be invoked only during debugging when maximum processor performance is not necessary.

### 3.6.3 Process Controls (PC) Register

The PC register (Figure 3-4) 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 3-26.

**Figure 3-4. Process Controls (PC) Register**



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 11, “Interrupts.”](#)

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 9, “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
- trace enable = disabled
- trace fault pending
- execution mode = supervisor
- state = interrupted

When the processor is re-initialized with a **sysctl** re-initialize message, the PC register returns to its reset value. See [Table 13-2 on page 13-6](#).

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 6-74](#) 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 9, “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.
- When an instruction executed in supervisor mode causes a bus access to occur, the processor asserts an external supervisor pin (SUP#) for loads, stores and instruction fetches. Hardware protection of system code or data can be implemented by using the supervisor pin to qualify write accesses to the protected memory.

- 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 SFRs, control registers and internal data RAM that is dedicated to interrupt controllers. A fault is generated if supervisor-only operations are attempted while the processor is in user mode.

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

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

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.

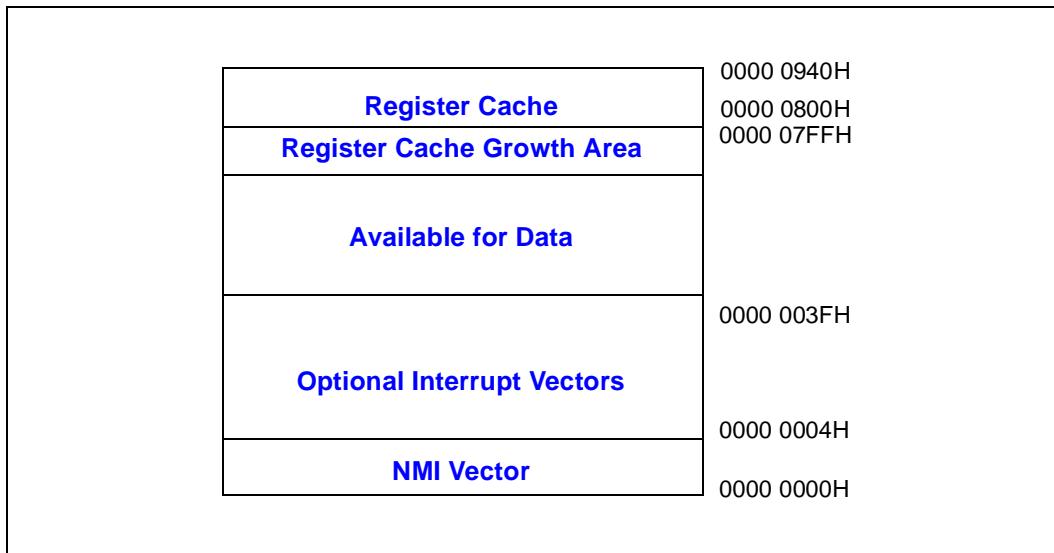
## 4.1 Internal Data Ram

Internal data RAM is mapped to the lower 2 Kbytes (0 to 07FFH) 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. However, the byte ordering of the internal data RAM is controlled by the byte-endian control bit in the DLMCON register.

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 14.2.1, “Bus Control \(BCON\) Register” on page 14-9](#). Protecting this portion of the data RAM from user and supervisor writes preserves the interrupt vectors that may be cached there. See [Section 11.9.2.1, “Vector Caching Option” on page 11-32](#).

**Figure 4-1. Internal Data RAM and Register Cache**

The on-chip register cache resides in the address space above the internal data RAM. When user software allocates more than 5 register frames in the register cache, the cache expands downward into the internal data RAM. [Table 13-8](#) shows the trade-offs between register cache size and available data RAM.

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.ip). Writes to internal data RAM locations while they are protected generate a TYPE.MISMATCH fault. See [Section 14.2.1, “Bus Control \(BCON\) Register” on page 14-9](#), for the format of the BCON register.

New versions of i960® processor compilers can 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 Hx 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 fifteen local register sets can be contained in the cache before sets must be saved in external memory. The default cache size is five register sets. 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 sixteenth 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 6.2.31, “flushreg” on page 6-53](#) 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.

See [Section 13.3.2.4, “Register Cache Configuration Word” on page 13-21](#) for more details on expanding the register cache size beyond five frames.

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 11 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 will result 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 = Total_Frames_Allocated - RCW[11:8];
if (interrupt_request)
    set_interrupt_handler_PC;
push_frame;
number_of_frames = number_of_frames + 1;
if (number_of_frames = Total_Frames_Allocated + 1) {
    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 15. 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 up to 14 frames to be shared by all 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 Big Endian Accesses to Internal Ram and Data Cache

The i960 Hx processor supports big-endian accesses to the internal data-RAM and data cache. The default byte order for data accesses is programmed in DLMCON.be to be either little or big-endian. The DLMCON.be controls the default byte-order for all internal (i.e., on-chip data ram and data cache) and external accesses. See [Section 14.4, “Programming the Logical Memory Attributes” on page 14-11](#) for more details.

## 4.4 Instruction Cache

The i960 Hx processor features a 16-Kbyte, 4-way set-associative instruction cache (I-cache) organized in lines of eight 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 aligned long-word (8 bytes) 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. Instruction fetches replace the least recently used line of the 4-way cache to reduce cache misses. The cache replacement algorithm stores new instructions in one of the two least recently used cache ways. The instruction cache assumes all areas of memory are cacheable.

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 i960<sup>®</sup> Cx processor software. Using **icctl** is the preferred and more versatile method for controlling the instruction cache on the i960 Hx processor.

Unlike the i960 Cx processor, the 80960Hx does not aggressively fetch instructions that may be unnecessary. Since the bus speed can be half or one third of the core speed, wasted external fetches can severely reduce performance. Instead, the i960 Hx processor fetches only as many words as are necessary (in two-word increments) to fill the cache void. The processor decides whether to fetch based on the instruction stream. For straight-line code, the processor fetches enough words to fill the current cache line. For branch instructions, no fetching occurs.

#### 4.4.1 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 [Figure 13-6](#). If 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 (inherited method from i960 Cx processor, not the preferred method for i960 Hx processor).
- The processor is reinitialized with a new value in the instruction cache configuration word.

#### 4.4.2 Operation While the Instruction Cache is Disabled

Disabling the instruction cache also disables all instruction fetch queues within the instruction fetch unit. With the instruction cache disabled, every instruction generates an external bus access. Small code loops cannot hide in the buffers. This behavior contrasts with the so-called “baby cache” behavior of the i960 Cx processor.

#### 4.4.3 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 Hx processor is any multiple of 4-Kbytes up to the full 16-Kbyte capacity of the cache. Any code can be locked into the cache, not just interrupt routines.

An **icctl** instruction invokes the load-and-lock mechanism for one, two, three, or all four 4-Kbyte ways of the instruction cache. Legacy software from the i960 Cx processor can still use the **sysctl** instruction to lock the cache, but with reduced flexibility. **sysctl** can load and lock only one 4-Kbyte way of the instruction cache due to backwards compatibility with the i960 Cx processor definition of the **sysctl** instruction. New software for the i960 Hx processor should use **icctl** for all instruction cache manipulations. With either instruction, when the lock option is selected, the processor loads the cache starting at an address specified as an operand to the instruction.

#### 4.4.4 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.

#### 4.4.5 Instruction Cache Coherency

The i960 Hx 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. The **icctl** instruction invalidates the instruction cache for the i960 Hx processor. Alternately, i960 Cx processor legacy software can use the **sysctl** instruction.

#### 4.4.6 Instruction Cache Interaction with Guarded Memory

The Guarded Memory Unit (GMU) protects only external memory accesses. Instructions executed from the instruction cache will not activate the GMU mechanisms since these instructions do not cause an external instruction fetch.

For example, consider the case of a program executing a section of code, then subsequently modifying the GCON register to fetch-protect that region. That block of instructions is already loaded into the instruction cache and can be accessed and executed again without the GMU signaling a fault.

To truly protect memory regions, the user must invalidate the instruction cache immediately after setting the GMU protection.

### 4.5 Data Cache

The i960 Hx processor features an 8-Kbyte, 4-way set-associative 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 (as is the i960 CF processor data cache). 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.

Data loads and stores replace least recently used lines of the 4-way cache to reduce cache misses. The cache replacement algorithm stores new data in one of the two least recently used cache ways.

User settings in the memory region configuration registers LMCON0-15 and DLMCON determine which data accesses are cacheable or non-cacheable based on memory region.

#### 4.5.1 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.

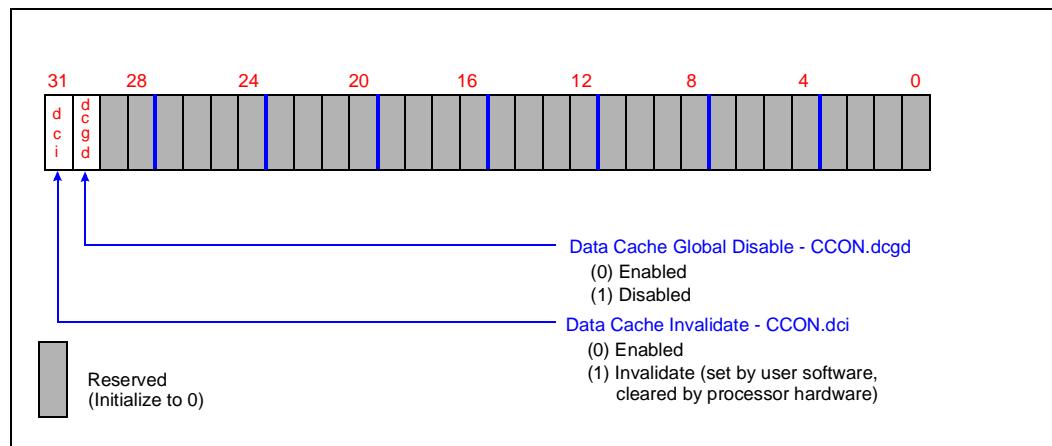
Though the **dcctl** instruction is the preferred method of controlling the data cache, i960 CF processor legacy software can still manipulate the data cache bits in the Cache Control (CCON) register directly, also known as sf2. Clear the data cache global disable (CCON.dcgd) bit (bit 30) to globally enable data caching. Set the bit to disable data caching. Invalidate the entire data cache by setting the data cache invalidate (CCON.dci) bit (bit 31) and polling (or “spinning”) on that bit until the processor clears it again, which indicates that the operation is finished. All other bits in CCON are reserved and must not be modified by user software.

[Figure 4-2](#) shows the CCON register bits.

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 14.4, “Programming the Logical Memory Attributes” on page 14-11](#) 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.

**Figure 4-2. Cache Control Register (CCON)**



## 4.5.2 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

If any words accessed by a **ldl**, **ldt**, or **ldq** instruction miss the data cache, every word accessed by that load instruction is updated in the cache.

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 15, “External Bus Description”](#) 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. (See [Chapter 15, “External Bus Description”](#) for more details.) The cache is also updated accordingly as described earlier in this chapter.

## 4.5.3 Data Cache Fill Policy

The i960 Hx 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.

#### 4.5.4 Data Cache Write Policy

The write policy determines what happens on cacheable writes (stores). The i960 Hx 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 Hx 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 D-cache is not updated; the tags and valid bits are not changed.

A byte or short-word will always be invalid in the D-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.

## 4.5.5 Data Cache Coherency and Non-Cacheable Accesses

The i960 Hx 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 if any of the following is true:

1. The access falls into an address range mapped by an enabled LMCN or DLMCON and the data-caching enabled bit in the matching LMCN 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.

## 4.5.6 External I/O and Bus Masters and Cache Coherency

The i960 Hx 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 LMCN 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 14.4, “Programming the Logical Memory Attributes” on page 14-11](#) for more information on this subject.

#### 4.5.7 Quickly Invalidating Portions of Data Cache

The entire data cache can be invalidated in one instruction using **dcctl**. This capability supports efficient program-controlled coherency enforcement by user software.

Sometimes, invalidating the entire cache is too extreme a step. In that case, user software can invalidate only a pre-defined region of memory selectively, leaving the rest of the cache intact and valid. The **dcctl** logical-region-invalidate command rapidly invalidates only the cache lines that have been earmarked for quick invalidation by the LMCON registers. Selective invalidation takes one clock cycle to complete.

The LMCON configuration at the time the cache line was allocated determines whether that line can be invalidated quickly. The LMCON value at the time of the **dcctl** logical-region-invalidate command is inconsequential.

The **dcinva** instruction invalidates individual quad words within the cache. See [Section 6.2.24, “dcinva \(80960Hx-Specific Instruction\)” on page 6-45](#) for more information.

A process must have data cache write permission to invalidate one or more addresses in the data cache.

#### 4.5.8 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 6-38](#) for more information. It should also be noted that the **dcinva** instruction invalidates cache data by clearing the valid bits; however, the invalidated data words are still available in the dumped cache contents. See [Section 6.2.24, “dcinva \(80960Hx-Specific Instruction\)” on page 6-45](#) for more information.



This chapter provides an overview of the i960® microprocessor family’s instruction set and i960 Hx 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 Hx 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 **add0**. 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 **add0**, ordinal operands in global registers g5 and g9 are added together, and the result is stored in g7:

```
add0 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	sf	special function register

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 2-8](#).

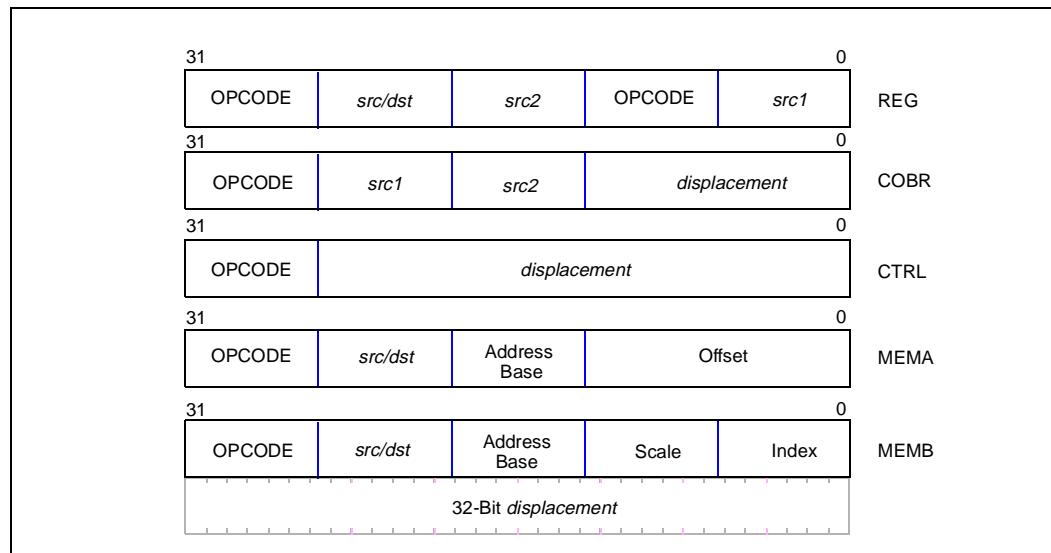
```
subi r3, r5, r6          #r6 = r5 - r3
setbit 13, g4, g5         #g5 = g4 with bit 13 set
lda 0xabcd, r12           #r12 = 0xabcd
ld (r4), g3                #g3 = memory location that r4 points to
st g10, (r6)[r7*2]        #g10 = memory location that r6+2*r7 points to
```

## 5.1.2 Instruction Encoding Formats

All instructions are encoded in one 32-bit machine language instruction — also known as 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 C, “Machine-Level Instruction Formats.”](#)

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 that combines compare and branch operations into one opword. Other compare and branch operations are also provided as REG and CTRL format instructions.
control	CTRL	Used 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, special function registers or literals. <i>src/dst</i> is either a global, local or special function 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, local or special function 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” on page 2-6</a> . 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 5-1](#). 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](#).

**Table 5-1. 80960Hx Instruction Set**

Data Movement	Arithmetic	Logical	Bit / Bit Field / Byte
Load Store Move Load Address Conditional Select <sup>(1)</sup>	Add Subtract Multiply Divide Conditional Add <sup>(1)</sup> Conditional Subtract <sup>(1)</sup> Remainder Modulo Shift Rotate Extended Shift Extended Multiply Extended Divide Add with Carry Subtract with Carry	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 <sup>(1)</sup>
Comparison	Branch	Call/Return	Fault
Compare Conditional Compare Compare byte <sup>(1)</sup> Compare short <sup>(1)</sup> Check Bit Compare and Increment Compare and Decrement Test Condition Code	Unconditional Branch Conditional Branch Compare and Branch Branch and Link	Call Call Extended Call System Return	Conditional Fault Synchronize Faults
Debug	Processor Mgmt	Atomic	Cache Control
Modify Trace Controls Mark Force Mark	Flush Local Registers Modify Arithmetic Controls Modify Process Controls Interrupt Enable/Disable <sup>(1,2)</sup> System Control <sup>(2)</sup> HALT <sup>(1,2)</sup>	Atomic Add Atomic Modify	Instruction Cache Control <sup>(1,2)</sup> Data Cache Control <sup>(1,2)</sup> Data Cache Invalidate by Address <sup>(1,2)</sup>

**NOTES:**

1. 80960Hx extensions to the 80960 core instruction set.
2. 80960Hx extensions to the 80960Cx instruction set.

## 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, global and special function 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 3-15](#) 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 if 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 if truncation resulted in the loss of significant bits.

### 5.2.1.2 Move

Move instructions copy data from a local, global, special function 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 (**Ida**) computes an effective address in the address space from an operand presented in one of the addressing modes. **Ida** 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 Hx processor, **Ida** is useful for performing simple arithmetic operations. The processor's parallelism allows **Ida** 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 5-2](#) lists arithmetic operations and data types for which the i960 Hx 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 5-19](#) for instructions that handle specific requirements for in-place memory operations.

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

**Table 5-2. 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	

### 5.2.3.1 Add, Subtract, Multiply, Divide, Conditional Add and 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>shrdi</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. If the shift operation results in an overflow, an integer-overflow fault is generated (if enabled). The destination register is written with the source shifted as much as possible without overflow and an integer-overflow fault is signaled.

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

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

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

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

### 5.2.3.4 Extended Arithmetic

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

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

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

## 5.2.5 Bit, Bit Field and Byte Operations

These instructions 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, global or special function registers.

### 5.2.5.1 Bit Operations

These instructions operate on a specified bit:

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

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

**chkbit**, described in [Section 5.2.6, “Comparison” on page 5-12](#), 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 if 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, global or special function registers as arguments.

**bswap** alters the order of bytes in a word, reversing its “endianess.” For more information on this subject, see [Section 14.1.2.1, “Byte Ordering” on page 14-5](#).

## 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>cmwo</b>	Compare Ordinal
<b>concmci</b>	Conditional Compare Integer
<b>concmwo</b>	Conditional Compare Ordinal
<b>chkbit</b>	Check Bit

These all use the REG format and can specify literals or local, global or special function 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 \(AC\) Register” on page 3-21](#) for a description of the condition codes for conditional operations.

**cmpi** and **cmwo** simply compare the two operands and set the condition code bits accordingly. **concmci** and **concmwo** first check the status of condition code bit 2:

- If not set, the operands are compared as with **cmpi** and **cmwo**.
- If 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 if A is between B and C (i.e.,  $B \leq A \leq C$ ). Here, a compare instruction (**cmpi** or **cmwo**) checks one side of the range (e.g.,  $A \geq B$ ) and a conditional compare instruction (**concmci** or **concmwo**) checks the other side (e.g.,  $A \leq C$ ) 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  $010_2$  if 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, global or special function 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{.t .f}</b>	test for equal
<b>testne{.t .f}</b>	test for not equal
<b>testl{.t .f}</b>	test for less
<b>testle{.t .f}</b>	test for less or equal
<b>testg{.t .f}</b>	test for greater
<b>testge{.t .f}</b>	test for greater or equal
<b>testo{.t .f}</b>	test for ordered
<b>testno{.t .f}</b>	test for unordered

If 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.

Using branch prediction suffixes on TEST<cc> instructions is allowed, but has no effect.

Since test instruction actions depend on a comparison, the architecture allows a programmer to predict the likely result of the operation for higher performance. The programmer's prediction is encoded in one bit of the opword. Intel 80960 assemblers encode the prediction with a mnemonic suffix of .t for true and .f for false.

## 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).

Since branch instruction actions depend on the result of a previous comparison, the architecture allows a programmer to predict the likely result of the branch operation for higher performance. The programmer's prediction is encoded in one bit of the opword. The Intel C Tools and GNU Tools encode the prediction with a mnemonic suffix of .t for true and .f for false.

### 5.2.7.1 Branch Prediction

Branch prediction is an implementation-specific feature of the i960 Hx processors. Not every implementation of the i960 architecture uses the branch prediction bit.

Since branch instruction actions depend on the result of a previous comparison, the architecture allows a programmer to predict the likely result of the branch operation for increased performance. The programmer's prediction is encoded in one bit of the machine language instruction. 80960 assemblers encode the prediction with a mnemonic suffix: .t = true, .f = false. Use the .t suffix to speed up execution when an instruction usually takes a branch; use the .f suffix when an instruction usually does not take a branch. See [Appendix C, “Machine-Level Instruction Formats”](#) for more on this subject.

### 5.2.7.2 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 7-21](#), 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.

### 5.2.7.3 Conditional Branch

With conditional branch (**BRANCH IF**) instructions, the processor checks the AC register condition code flags. If 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{.t f}</b>	branch if equal/true
<b>bne{.t f}</b>	branch if not equal
<b>bl{.t f}</b>	branch if less
<b>ble{.t f}</b>	branch if less or equal
<b>bg{.t f}</b>	branch if greater
<b>bge{.t f}</b>	branch if greater or equal
<b>bo{.t f}</b>	branch if ordered
<b>bno{.t f}</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 3-22](#) for a discussion of the condition code for conditional operations.

### 5.2.7.4 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{.t   .f}</b>	compare integer and branch if equal
<b>cmpibne{.t   .f}</b>	compare integer and branch if not equal
<b>cmpibl{.t   .f}</b>	compare integer and branch if less
<b>cmpible{.t   .f}</b>	compare integer and branch if less or equal
<b>cmpibg{.t   .f}</b>	compare integer and branch if greater
<b>cmpibge{.t   .f}</b>	compare integer and branch if greater or equal
<b>cmpibo{.t   .f}</b>	compare integer and branch if ordered
<b>cmpibno{.t   .f}</b>	compare integer and branch if unordered
<b>cmpobe{.t   .f}</b>	compare ordinal and branch if equal
<b>cmpobne{.t   .f}</b>	compare ordinal and branch if not equal
<b>cmpobl{.t   .f}</b>	compare ordinal and branch if less
<b>cmpoble{.t   .f}</b>	compare ordinal and branch if less or equal
<b>cmpobg{.t   .f}</b>	compare ordinal and branch if greater
<b>cmpobge{.t   .f}</b>	compare ordinal and branch if greater or equal
<b>bbs{.t   .f}</b>	check bit and branch if set
<b>bbc{.t   .f}</b>	check bit and branch if clear

All use the COBR machine instruction format and can specify literals, local, global and special function 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 5-12](#). 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<sub>2</sub> (true) if the bit is set and 000<sub>2</sub> (false) if 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 Hx 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, global or special function 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; that is, 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.

<b>faulte{.t .f}</b>	fault if equal
<b>faultne{.t .f}</b>	fault if not equal
<b>faultl{.t .f}</b>	fault if less
<b>faultle{.t .f}</b>	fault if less or equal
<b>faultg{.t .f}</b>	fault if greater
<b>faultge{.t .f}</b>	fault if greater or equal
<b>faulto{.t .f}</b>	fault if ordered
<b>faultno{.t .f}</b>	fault if unordered

All use the CTRL format. Since the actions of these instructions are dependent upon the result of a previous comparison, the architecture allows a programmer to predict the likely result of the conditional fault instructions for higher performance. The programmer's prediction is encoded in one bit of the opword. The Intel C Tools and GNU Tools encode the prediction with a mnemonic suffix of .t for true and .f for false.

The **syncf** instruction 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 9, “Tracing and Debugging”](#).

**modtc** permits trace controls to be modified. **mark** causes a breakpoint trace event to be generated if 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 Hx 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, global or special function 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
<b>sysctl</b>	Perform system control function
<b>inten</b>	Global interrupt enable
<b>intdis</b>	Global interrupt enable and disable

All use the REG format and can specify literals or local, global or special function 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.

## 5.2.13 Cache Control

The following instructions provide instruction and data cache control functions.

**icctl** Instruction cache control

**dcctl** Data cache control

**dcinva** Data cache invalidate by address

**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.

**dcinva** invalidates a user-specified quad word in the data cache.



# Instruction Set Reference

6

This chapter provides detailed information about each instruction available to the i960<sup>®</sup> Hx 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 Hx 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 B, “Opcodes and Execution Times”](#) - A quick-reference listing of instruction encodings assists debugging with a logic analyzer.
- [Appendix C, “Machine-Level Instruction Formats”](#) - Describes instruction set opword encodings.
- *i960 Hx Processor Instruction Set Quick Reference* (order number 272677) - A pocket-sized quick reference to all instructions.

## 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. If 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 Hx 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.

CTRL and COBR format instructions also allow the programmer to specify optional .t or .f mnemonic suffixes for branch prediction:

- .t indicates to the processor that the condition the instruction is testing for is likely to be true.
- .f indicates that the condition is likely to be false.

The processor uses the programmer's prediction to prefetch and decode instructions along the most likely execution path when the actual path is not yet known. If the prediction was wrong, all actions along the incorrect path are undone and the correct path is taken. For further discussion, see [Section E.2.7.7., “Branch Prediction” on page E-42](#).

When the programmer provides no suffix with an instruction which supports a suffix, the assembler makes its own prediction.

When an instruction supports prediction, the mnemonic listing includes the notation { .t|.f} to indicate the option, for example:

**be{.t|.f}** Branch If Equal

### 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/sfr	reg/lit/sfr	reg/sfr

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
sfr	Special Function Register (sf0 ... sf4)
disp	Signed displacement of range (-2 <sup>22</sup> ... 2 <sup>22</sup> - 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 if PC.mk is set.
	INSTRUCTION	An Instruction Trace Event is signaled after instruction completion. A Trace fault is generated if 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 if 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, by an operand reference to an unaligned long-, triple- or quad-register group, or by a non-defined sfr.
	UNIMPLEMENTED	This fault can occur due to an attempt to perform a non-word or unaligned access to a memory-mapped region or if trying 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. If 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 C, "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>

Mnemonic	<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		
Format	<b>add*</b>	<i>src1,</i> <i>reg/lit/sfr</i>	<i>src2,</i> <i>reg/lit/sfr</i>	<i>dst</i> <i>reg/sfr</i>
Description	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
addono	$000_2$	Unordered
addino		
addog	$001_2$	Greater
addig		
addoe	$010_2$	Equal
addie		
addoge	$011_2$	Greater or equal
addige		
addol	$100_2$	Less
addil		
addone	$101_2$	Not equal
addine		
addole	$110_2$	Less or equal
addile		
addoo	$111_2$	Ordered
addio		

Action

```

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

```

```

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

```

Faults

STANDARD	Refer to Section 6.1.6, “Faults” on page 6-5.
ARITHMETIC.OVERFLOW	Occurs only with <b>addi&lt;cc&gt;</b> .

Example	# Assume (AC.cc AND 001 <sub>2</sub> ) ≠ 0. addig r4, r8, r10 # r10 = r8 + r4
	# Assume (AC.cc AND 101 <sub>2</sub> ) = 0. addone r4, r8, r10 # r10 is not changed.
Opcode	<b>addono</b> 780H REG <b>addog</b> 790H REG <b>addoe</b> 7A0H REG <b>addoge</b> 7B0H REG <b>addol</b> 7C0H REG <b>addone</b> 7D0H REG <b>addole</b> 7E0H REG <b>addoo</b> 7F0H REG <b>addino</b> 781H REG <b>addig</b> 791H REG <b>addie</b> 7A1H REG <b>addige</b> 7B1H REG <b>addil</b> 7C1H REG <b>addine</b> 7D1H REG <b>addile</b> 7E1H REG <b>addio</b> 7F1H REG
See Also	<b>addc, SUB&lt;cc&gt;, addi, addo</b>

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

## 6.2.2 addc

Mnemonic	<b>addc</b> Add Ordinal With Carry					
Format	<b>addc</b>	<i>src1,</i> reg/lit/sfr	<i>src2,</i> reg/lit/sfr	<i>dst</i> reg/sfr		
Description	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><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.</p>						
An integer overflow fault is never signaled with this instruction.						
Action	<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>					
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .				
Example	<pre># Example of double-precision arithmetic. # Assume 64-bit source operands # in g0,g1 and g2,g3 cmppo 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>					
Opcode	<b>addc</b>	5B0H	REG			
See Also	<a href="#">ADD&lt;cc&gt;, SUB&lt;cc&gt;</a>					
Notes	Sets the condition code in the arithmetic controls.					

## 6.2.3 addi, addo

Mnemonic	<b>addo</b>	Add Ordinal
	<b>addi</b>	Add Integer
Format	<b>add*</b>	<i>src1</i> , <i>src2</i> , <i>dst</i> reg/lit/sfr            reg/lit/sfr            reg/sfr
Description	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.	
Action	<b>addo:</b> $dst = (src2 + src1)[31:0];$  <b>addi:</b> $true\_result = (src1 + src2);$ $dst = true\_result[31:0];$ $if((true\_result > (2^{**31} - 1)    (true\_result < -2^{**31})) \# Check for overflow$ { $if(AC.om == 1)$ $AC.of = 1;$ $else$ $generate\_fault(ARITHMETIC.OVERFLOW);$ }	
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5.</a>
	ARITHMETIC.OVERFLOW	Occurs only with <b>addi</b> .
Example	<b>addi r4, g5, r9 # r9 = g5 + r4</b>	
Opcode	<b>addo</b>	590H
	<b>addi</b>	591H
See Also	<b>addc, subi, subo, subc, ADD&lt;cc&gt;</b>	

## 6.2.4 alterbit

Mnemonic	<b>alterbit</b>	Alter Bit
Format	<b>alterbit</b>	<i>bitpos</i> , reg/lit/sfr <i>src</i> , reg/lit/sfr <i>dst</i> reg/sfr
Description	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> ).	
Action	<pre>if((AC.cc &amp; 010<sub>2</sub>)==0)     dst = src &amp; ~(2**((bitpos%32))); else     dst = src   2**((bitpos%32));</pre>	
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .
Example	# Assume AC.cc = 010 <sub>2</sub> . alterbit 24, g4,g9      # g9 = g4, with bit 24 set.	
Opcode	<b>alterbit</b>	58FH
See Also	<b>chkbit, clrbit, notbit, setbit</b>	

## 6.2.5 and, andnot

Mnemonic	<b>and</b>	And	
	<b>andnot</b>	And Not	
Format	<b>and</b> <i>src1</i> , reg/lit/sfr	<i>src2</i> , reg/lit/sfr	<i>dst</i> reg/sfr
	<b>andnot</b> <i>src1</i> , reg/lit/sfr	<i>src2</i> , reg/lit/sfr	<i>dst</i> reg/sfr
Description	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:		
	{ <i>src2</i> and not ( <i>src1</i> )} rather than { <i>src1</i> and not ( <i>src2</i> )}.		
Action	<b>and:</b> dst = src2 & src1;		
	<b>andnot:</b> dst = src2 & ~src1;		
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .	
Example	<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>		
Opcode	<b>and</b> 581H      REG		
	<b>andnot</b> 582H      REG		
See Also	<a href="#">nand</a> , <a href="#">nor</a> , <a href="#">not</a> , <a href="#">notand</a> , <a href="#">notor</a> , <a href="#">or</a> , <a href="#">ornot</a> , <a href="#">xnor</a> , <a href="#">xor</a>		

## 6.2.6 atadd

Mnemonic	<b>atadd</b> Atomic Add			
Format	<b>atadd</b>	<i>addr,</i> reg/sfr	<i>src,</i> reg/lit/sfr	<i>dst</i> reg/sfr
Description	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> .			
	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” on page 3-16</a> or more information on atomic accesses.			
	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.)			
Action	<pre>implicit_syncf(); tempa = addr &amp; 0xFFFFFFFFC; temp = atomic_read(tempa); atomic_write(tempa, temp+src); dst = temp;</pre>			
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .		
Example	<pre>atadd r8, r3, r11 # r8 contains the address of # memory location. # r11 = (r8) # (r8) = r11 + r3.</pre>			
Opcode	<b>atadd</b>	612H	REG	
See Also	<b>atmod</b>			

## 6.2.7 atmod

Mnemonic	<b>atmod</b>	Atomic Modify	
Format	<b>atmod</b>	<i>addr</i> , reg/sfr	<i>mask</i> , reg/lit/sfr
Description	<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 3-16</a> or more 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>		
Action	<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>		
Faults	STANDARD		
Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .			
Example	<pre>atmod g5, g7, g10      # tempa = (g5) # temp = (tempa andnot g7) or # (g10 and g7) # (g5) = temp # g10 = tempa</pre>		
Opcode	<b>atmod</b>	610H	REG
See Also	<b>atadd</b>		

## 6.2.8 b, bx

Mnemonic	<b>b</b>	Branch	
	<b>bx</b>	Branch Extended	
Format	<b>b</b>	<i>targ</i> <i>disp</i>	
	<b>bx</b>	<i>targ</i> mem	
Description	Branches to the specified target.		
	<p>With the <b>b</b> instruction, IP specified with <i>targ</i> operand can be no farther than -<math>2^{23}</math> to (<math>2^{23}</math>- 4) bytes from current IP. When using the Intel i960 processor assembler, <i>targ</i> operand must be a label which specifies target instruction's IP.</p> <p><b>bx</b> performs the same operation as <b>b</b> except the target instruction can be farther than -<math>2^{23}</math> to (<math>2^{23}</math>- 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.</p>		
	Refer to <a href="#">Section 2.3, “Memory Addressing Modes” on page 2-6</a> for information on this subject.		
Action	<b>b, bx:</b> IP[31:2] = effective_address( <i>targ</i> [31:2]); IP[1:0] = 0;		
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .	
Example	<pre>b xyz           # IP = xyz; bx 1332 (ip)   # IP = IP + 8 + 1332; # this example uses IP-relative addressing</pre>		
Opcode	<b>b</b>	08H	CTRL
	<b>bx</b>	84H	MEM
See Also	bal, balx, BRANCH<cc>, COMPARE AND BRANCH<cc>, bbc, bbs		

## 6.2.9 bal, balx

Opcode	<b>bal</b> <b>balx</b>	0BH 85H	CTRL MEM
See Also	<b>b, bx, BRANCH&lt;cc&gt;, COMPARE AND BRANCH&lt;cc&gt;, bbc, bbs</b>		

## 6.2.10 bbc, bbs

Mnemonic	<b>bbc{.t f}</b>	Check Bit and Branch If Clear		
	<b>bbs{.t f}</b>	Check Bit and Branch If Set		
Format	<b>bb*{.t f}</b>	<i>bitpos</i> ,	<i>src</i> ,	<i>targ</i>
		reg/lit	reg/sfr	disp
Description	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.			
	For <b>bbc</b> , if selected bit in <i>src</i> is clear, the processor sets condition code to $000_2$ and branches to instruction specified by <i>targ</i> ; otherwise, it sets condition code to $010_2$ and goes to next instruction.			
	For <b>bbs</b> , if selected bit is set, the processor sets condition code to $010_2$ and branches to <i>targ</i> ; otherwise, it sets condition code to $000_2$ and goes to next instruction.			
	<i>targ</i> can be no farther than $-2^{12}$ to $(2^{12} - 4)$ bytes from current IP. When using the Intel i960 processor assembler, <i>targ</i> must be a label which specifies target instruction's IP.			
Action	<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>			
Faults	STANDARD			
	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .			
Example	<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>			
Opcode	<b>bbc</b>	30H	COBR	
	<b>bbs</b>	37H	COBR	

See Also

**chkbit, COMPARE AND BRANCH<cc>, BRANCH<cc>**

Side Effects

Sets the condition code in the arithmetic controls.

## 6.2.11 BRANCH<cc>

Mnemonic	<b>be{.t .f}</b>	Branch If Equal
	<b>bne{.t .f}</b>	Branch If Not Equal
	<b>bl{.t .f}</b>	Branch If Less
	<b>ble{.t .f}</b>	Branch If Less Or Equal
	<b>bg{.t .f}</b>	Branch If Greater
	<b>bge{.t .f}</b>	Branch If Greater Or Equal
	<b>bo{.t .f}</b>	Branch If Ordered
	<b>bno{.t .f}</b>	Branch If Unordered
Format	<b>b*{.t .f}</b>	<i>targ</i> <i>disp</i>
Description	Branches to instruction specified with <i>targ</i> operand according to AC register condition code state.	
	Optional .t or .f suffix may be appended to mnemonic. Use .t to speed up execution when these instructions usually take the branch. Use .f to speed up execution when these instructions usually do not take the branch. If a suffix is not provided, the assembler is free to provide one.	
	For all branch<cc> instructions except <b>bno</b> , the processor branches to instruction specified with <i>targ</i> , if the logical AND of condition code and mask part of opcode is not zero. Otherwise, it goes to next instruction.	
	For <b>bno</b> , the processor branches to instruction specified with <i>targ</i> if the condition code is zero. Otherwise, it goes to next instruction.	
	For instance, <b>bno</b> (unordered) can be used as a branch if false instruction when coupled with <b>chkbit</b> . For <b>bno</b> , branch is taken if condition code equals $000_2$ . <b>be</b> can be used as branch-if true instruction.	
	The <i>targ</i> 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	<pre> if((mask &amp; 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; } </pre>																										
Faults	STANDARD Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5.</a>																										
Example	<pre> # Assume (AC.cc AND 100<sub>2</sub>) ≠ 0 bl xyz # IP = xyz; </pre>																										
Opcode	<table border="0"> <tr> <td><b>be</b></td> <td>12H</td> <td>CTRL</td> </tr> <tr> <td><b>bne</b></td> <td>15H</td> <td>CTRL</td> </tr> <tr> <td><b>bl</b></td> <td>14H</td> <td>CTRL</td> </tr> <tr> <td><b>ble</b></td> <td>16H</td> <td>CTRL</td> </tr> <tr> <td><b>bg</b></td> <td>11H</td> <td>CTRL</td> </tr> <tr> <td><b>bge</b></td> <td>13H</td> <td>CTRL</td> </tr> <tr> <td><b>bo</b></td> <td>17H</td> <td>CTRL</td> </tr> <tr> <td><b>bno</b></td> <td>10H</td> <td>CTRL</td> </tr> </table>			<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
<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>b, bx, bbc, bbs, COMPARE AND BRANCH&lt;cc&gt;, bal, balx, BRANCH&lt;cc&gt;</b>																										

## 6.2.12 bswap

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

## 6.2.13 call

Mnemonic	<b>call</b>	Call
Format	<b>call</b>	<i>targ</i> <i>disp</i>
Description	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.	
	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.	
	<i>targ</i> can be no farther than $-2^{23}$ to $(2^{23} - 4)$ bytes from current IP.	
Action	<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 i960 Hx processors. 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>	
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .
Example	<b>call xyz</b>	# IP = xyz
Opcode	<b>call</b>	09H
See Also	<b>bal, calls, callx</b>	

## 6.2.14 calls

Mnemonic	<b>calls</b>	Call System
Format	<b>calls</b>	<i>targ</i> reg/lit
Description	Calls a system procedure. The <i>targ</i> operand gives the number of the procedure being called. For <b>calls</b> , the processor performs system call operation described in <a href="#">Section 7.5, “System Calls” on page 7-15</a> . <i>targ</i> 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	<pre> # Wait for any uncompleted instructions to finish. implicit_syncf(); If (targ &gt; 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)) &amp; ~(SALIGN*16 - 1)     # Round stack pointer to next boundary.     # SALIGN=1 on i960 Hx processors.     temp.RRR = 0002; } else # Supervisor call from user mode. {     tempa = SSP; # Get Supervisor Stack pointer.     temp.RRR = 0102   PC.te;     PC.em = supervisor;     PC.te = temp.te; } PFP = FP; PFP.rrr = temp.RRR; FP = tempa; SP = tempa + 64; </pre>	
Faults	<p>STANDARD Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5.</a></p> <p>PROTECTION.LENGTH Specifies a procedure number greater than 259.</p>	
Example	<pre> calls r12      # IP = value obtained from                 # procedure table for procedure                 # number given in r12. calls 3        # Call procedure 3. </pre>	
Opcode	<b>calls</b>	660H REG
See Also	<b>bal, call, callx, ret</b>	

## 6.2.15 callx

Mnemonic	<b>callx</b>	Call Extended
Format	<b>callx</b>	<i>targ</i> mem
Description	Calls new procedure. <i>targ</i> specifies IP of called procedure's first instruction.	
		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.
		<b>callx</b> performs the same operation as <b>call</b> except the target instruction can be farther than $-2^{23}$ to $(2^{23} - 4)$ bytes from current IP.
		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.
		Refer to <a href="#">Chapter 2, “Data Types and Memory Addressing Modes”</a> for more information.
Action	# Wait for any uncompleted instructions to finish; implicit_syncf(); temp = (SP + (SALIGN*16 - 1)) & ~ (SALIGN*16 - 1) # Round stack pointer to next boundary. # SALIGN=1 on i960 Hx processors. RIP = IP; if (register_set_available) allocate_new_frame( ); else { save_register_set( ); # Save register set in memory at its FP; allocate_new_frame( ); } # Local register references now refer to new frame. IP[31:2] = effective_address(targ[31:2]); IP[1:0] = 0; PFP = FP; FP = temp; SP = temp + 64;	
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .
Example	callx (g5) # IP = (g5), where the address in g5 # is the address of the new procedure.	
Opcode	<b>callx</b>	86H
See Also	<b>bal, call, calls, ret</b>	

## 6.2.16 chkbit

Mnemonic	<b>chkbit</b>	Check Bit
Format	<b>chkbit</b>	<i>bitpos</i> , <i>src2</i> reg/lit/sfr      reg/lit/sfr
Description		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 $010_2$ ; if bit is clear, condition code is set to $000_2$ .
Action		<pre>if (((src2 &amp; 2**(bitpos % 32)) == 0)      AC.cc = 000<sub>2</sub>; else      AC.cc = 010<sub>2</sub>;</pre>
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .
Example	chkbit 13, g8	# Checks bit 13 in g8 and sets # AC.cc according to the result.
Opcode	<b>chkbit</b>	5AEH
See Also		<b>alterbit, clrbit, notbit, setbit, cmpi, cmpl</b>
Notes		Sets the condition code in the arithmetic controls.

## 6.2.17 clrbit

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

## 6.2.18 cmpdeci, cmpdeco

Mnemonic	<b>cmpdeci</b>	Compare and Decrement Integer
	<b>cmpdeco</b>	Compare and Decrement Ordinal
Format	<b>cmpdec*</b>	<i>src1</i> , <i>src2</i> , <i>dst</i> reg/lit/sfr      reg/lit/sfr      reg/sfr
Description		Compares <i>src2</i> and <i>src1</i> values and sets the condition code according to comparison results. <i>src2</i> is then decremented by one and result is stored in <i>dst</i> . The following table shows condition code setting for the three possible results of the comparison.

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

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

Action	<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>; dst = src2 -1; # Overflow suppressed for <b>cmpdeci</b>.</pre>	
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .
Example	cmpdeci 12, g7, g1	# Compares g7 with 12 and sets # AC.cc to indicate the result # g1 = g7 - 1.
Opcode	<b>cmpdeci</b>	5A7H           REG
	<b>cmpdeco</b>	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	<b>cmpinci</b>	Compare and Increment Integer								
	<b>cmpinco</b>	Compare and Increment Ordinal								
Format	<b>cmpinc*</b>	<i>src1</i> , <i>src2</i> , <i>dst</i> reg/lit/sfr      reg/lit/sfr      reg/sfr								
Description	Compares <i>src2</i> and <i>src1</i> values and sets the condition code according to comparison results. <i>src2</i> is then incremented by one and result is stored in <i>dst</i> . The following table shows condition code settings for the three possible comparison results.									
<table border="1"> <thead> <tr> <th>Condition Code</th><th>Comparison</th></tr> </thead> <tbody> <tr> <td><math>100_2</math></td><td><math>src1 &lt; src2</math></td></tr> <tr> <td><math>010_2</math></td><td><math>src1 = src2</math></td></tr> <tr> <td><math>001_2</math></td><td><math>src1 &gt; src2</math></td></tr> </tbody> </table>			Condition Code	Comparison	$100_2$	$src1 < src2$	$010_2$	$src1 = src2$	$001_2$	$src1 > src2$
Condition Code	Comparison									
$100_2$	$src1 < src2$									
$010_2$	$src1 = src2$									
$001_2$	$src1 > src2$									

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

Action	if ( <i>src1</i> < <i>src2</i> ) AC.cc = $100_2$ ; else if ( <i>src1</i> == <i>src2</i> ) AC.cc = $010_2$ ; else AC.cc = $001_2$ ;	
		dst = <i>src2</i> + 1;                          # Overflow suppressed for cmpinci.
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .
Example	cmpinco r8, g2, g9	# Compares the values in g2 # and r8 and sets AC.cc to # indicate the result: # g9 = g2 + 1
Opcode	<b>cmpinci</b> 5A5H           REG <b>cmpinco</b> 5A4H           REG	
See Also	<a href="#">cmpdeco</a> , <a href="#">cmopo</a> , <a href="#">cmpi</a> , <a href="#">cmpdeci</a> , <a href="#">COMPARE AND BRANCH&lt;cc&gt;</a>	
Side Effects	Sets the condition code in the arithmetic controls.	

## 6.2.20 COMPARE

Mnemonic	<b>cmpl</b>	Compare Integer								
	<b>cmpib</b>	Compare Integer Byte								
	<b>cmpis</b>	Compare Integer Short								
	<b>cmpl</b>	Compare Ordinal								
	<b>cmpob</b>	Compare Ordinal Byte								
	<b>cmpos</b>	Compare Ordinal Short								
Format	<b>cmp</b> * <i>src1</i> , reg/lit/sfr	<i>src2</i> reg/lit/sfr								
Description	Compares <i>src2</i> and <i>src1</i> values and sets condition code according to comparison results. The following table shows condition code settings for the three possible comparison results.									
	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: center; padding: 2px;">Condition Code</th><th style="text-align: center; padding: 2px;">Comparison</th></tr> </thead> <tbody> <tr> <td style="text-align: center; padding: 2px;"><math>100_2</math></td><td style="text-align: center; padding: 2px;"><i>src1 &lt; src2</i></td></tr> <tr> <td style="text-align: center; padding: 2px;"><math>010_2</math></td><td style="text-align: center; padding: 2px;"><i>src1 = src2</i></td></tr> <tr> <td style="text-align: center; padding: 2px;"><math>001_2</math></td><td style="text-align: center; padding: 2px;"><i>src1 &gt; src2</i></td></tr> </tbody> </table>		Condition Code	Comparison	$100_2$	<i>src1 &lt; src2</i>	$010_2$	<i>src1 = src2</i>	$001_2$	<i>src1 &gt; src2</i>
Condition Code	Comparison									
$100_2$	<i>src1 &lt; src2</i>									
$010_2$	<i>src1 = src2</i>									
$001_2$	<i>src1 &gt; src2</i>									
Action	<p># For cmpl, cmpi, N = 31.  # For cmpos, cmpis, N = 15.  # For cmpob, cmpib, N = 7.</p> <pre style="margin-left: 40px;">if (src1[N:0] &lt; src2[N:0])     AC.cc = 100<sub>2</sub>; else if (src1[N:0] == src2[N:0])     AC.cc = 010<sub>2</sub>; else if (src1[N:0] &gt; src2[N:0])     AC.cc = 001<sub>2</sub>;</pre>									
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .								
Example	<pre>cmpl 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.</pre>									

Opcode	<b>cmpl</b>	5A1H	REG
	<b>cmpib</b>	595H	REG
	<b>cmpis</b>	597H	REG
	<b>cmpl</b>	5A0H	REG
	<b>cmpob</b>	594H	REG
	<b>cmpos</b>	596H	REG
See Also	<b>COMPARE AND BRANCH&lt;cc&gt;, cmpdeci, cmpdeco, cmpinci, cmpinco, concmpi, concmpo</b>		
Side Effects	Sets the condition code in the arithmetic controls.		
Notes	The core instructions <b>cmpib</b> , <b>cmpis</b> , <b>cmpob</b> and <b>cmpos</b> are not implemented on Cx, Kx and Sx 80960 processors.		

## 6.2.21 COMPARE AND BRANCH<cc>

Mnemonic	<b>cmpibe{.t .f}</b> Compare Integer and Branch If Equal <b>cmpibne{.t .f}</b> Compare Integer and Branch If Not Equal <b>cmpibl{.t .f}</b> Compare Integer and Branch If Less <b>cmpible{.t .f}</b> Compare Integer and Branch If Less Or Equal <b>cmpibg{.t .f}</b> Compare Integer and Branch If Greater <b>cmpibge{.t .f}</b> Compare Integer and Branch If Greater Or Equal <b>cmpibo{.t .f}</b> Compare Integer and Branch If Ordered <b>cmpibno{.t .f}</b> Compare Integer and Branch If Not Ordered		
Format	<b>cmpib*{.t .f}</b> <i>src1</i> , <i>src2</i> , <i>targ</i> reg/lit      reg/sfr      disp <b>cmpob*{.t .f}</b> <i>src1</i> , <i>src2</i> , <i>targ</i> reg/lit      reg/sfr      disp		
Description	<p>Compares <i>src2</i> and <i>src1</i> 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 <i>targ</i>; otherwise, the processor goes to next instruction.</p> <p>Optional .t or .f suffix may be appended to mnemonic. Use .t to speed up execution when these instructions usually take the branch. Use .f to speed up execution when these instructions usually do not take the branch. If a suffix is not provided, the assembler is free to provide one.</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> <p>Functions these instructions perform can be duplicated with a <b>cmpi</b> or <b>cmwo</b> followed by a branch-if instruction, as described in <a href="#">Section 6.2.20, “COMPARE” on page 6-32</a>.</p> <p>The following table shows the condition-code mask for each instruction. The mask is in bits 0-2 of the opcode.</p>		

Instruction	Mask	Branch Condition
<b>cmpibno</b>	$000_2$	No Condition
<b>cmpibg</b>	$001_2$	$src1 > src2$
<b>cmpibe</b>	$010_2$	$src1 = src2$
<b>cmpibge</b>	$011_2$	$src1 \geq src2$
<b>cmpibl</b>	$100_2$	$src1 < src2$

<b>Instruction</b>	<b>Mask</b>	<b>Branch Condition</b>
<b>cmpibne</b>	$101_2$	$src1 \neq src2$
<b>cmpible</b>	$110_2$	$src1 \leq src2$
<b>cmpibo</b>	$111_2$	Any Condition
<b>cmpobg</b>	$001_2$	$src1 > src2$
<b>cmpobe</b>	$010_2$	$src1 = src2$
<b>cmpobge</b>	$011_2$	$src1 \geq src2$
<b>cmpobl</b>	$100_2$	$src1 < src2$
<b>cmpobne</b>	$101_2$	$src1 \neq src2$
<b>cmpoble</b>	$110_2$	$src1 \leq src2$

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

Action	<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>		
Faults	STANDARD		Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .
Example	<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>		
Opcode	<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
See Also	BRANCH<cc>, cmpi, cmpo, bal, balx		
Notes	Sets the condition code in the arithmetic controls.		

## 6.2.22 concmpi, concmbo

Mnemonic	<b>concmphi</b>	Conditional Compare Integer
	<b>concmbo</b>	Conditional Compare Ordinal
Format	<b>concm*</b>	<i>src1</i> , <i>src2</i> reg/lit/sfr      reg/lit/sfr
Description		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.
		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.
Action		The example below illustrates this application by testing whether g3 value is between g5 and g6 values, where g5 is assumed to be less than g6. First a comparison ( <b>cmpo</b> ) of g3 and g6 is performed. If g3 is less than or equal to g6 (i.e., condition code is either 010 <sub>2</sub> or 001 <sub>2</sub> ), a conditional comparison ( <b>concmbo</b> ) of g3 and g5 is then performed. If g3 is greater than or equal to g5 (indicating that g3 is within the bounds of g5 and g6), condition code is set to 010 <sub>2</sub> ; otherwise, it is set to 001 <sub>2</sub> .
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .
Example	<code>cmpo g6, g3</code> <code>concmbo g5, g3</code>	# Compares g6 and g3 # and sets AC.cc. # If AC.cc < 100 <sub>2</sub> (g6 ≥ g3) # g5 is compared with g3.

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

**Table 6-5. concmbo example: register ordering and CC**

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

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

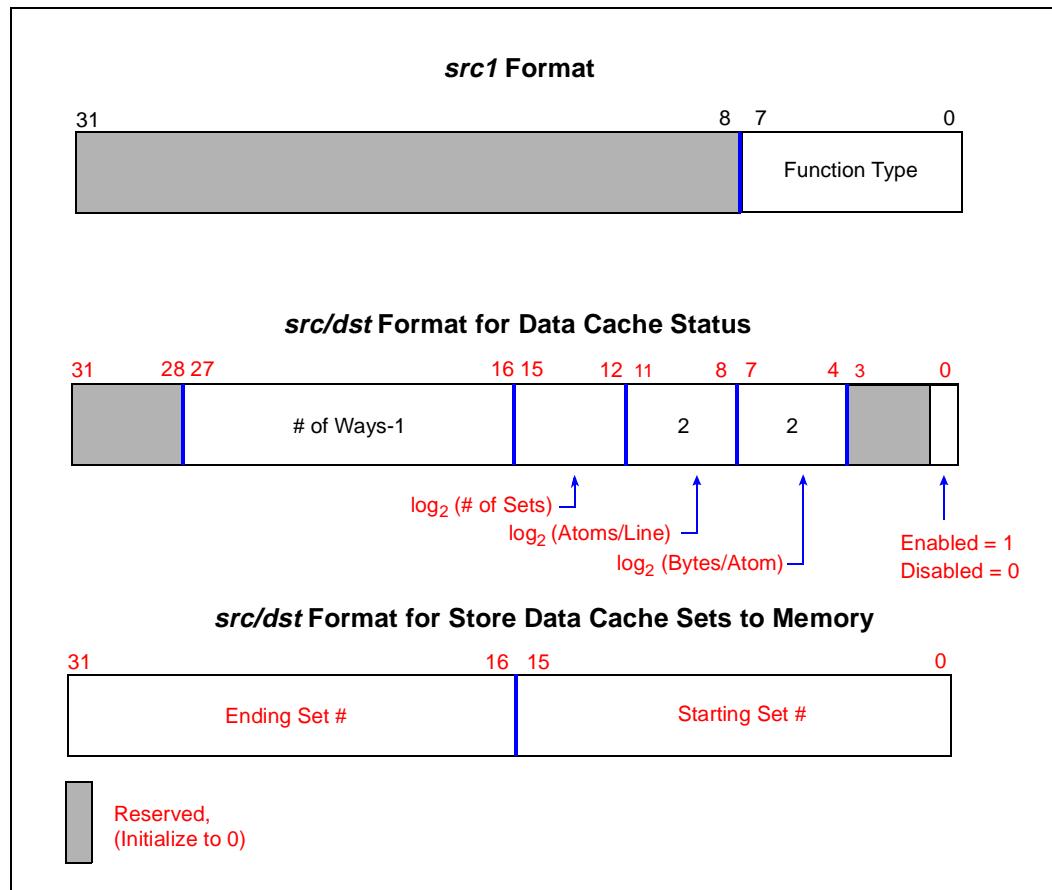
## 6.2.23 dcctl

Mnemonic	<b>dcctl</b> Data-cache Control		
Format	<i>src1</i> , reg/lit/sfr	<i>src2</i> , reg/lit/sfr	<i>src/dst</i> reg/sfr
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 <i>src1</i> . <i>src2</i> and <i>src/dst</i> 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	<b>src1</b>	<b>src2</b>	<b>src/dst</b>
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 (see <a href="#">Figure 6-1</a> ).
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 (see <a href="#">Figure 6-1</a> ).
Reserved	7	NA	NA
Quick invalidate	8	1	NA
Reserved	9	NA	NA

1. Invalidates data cache on 80960Hx.

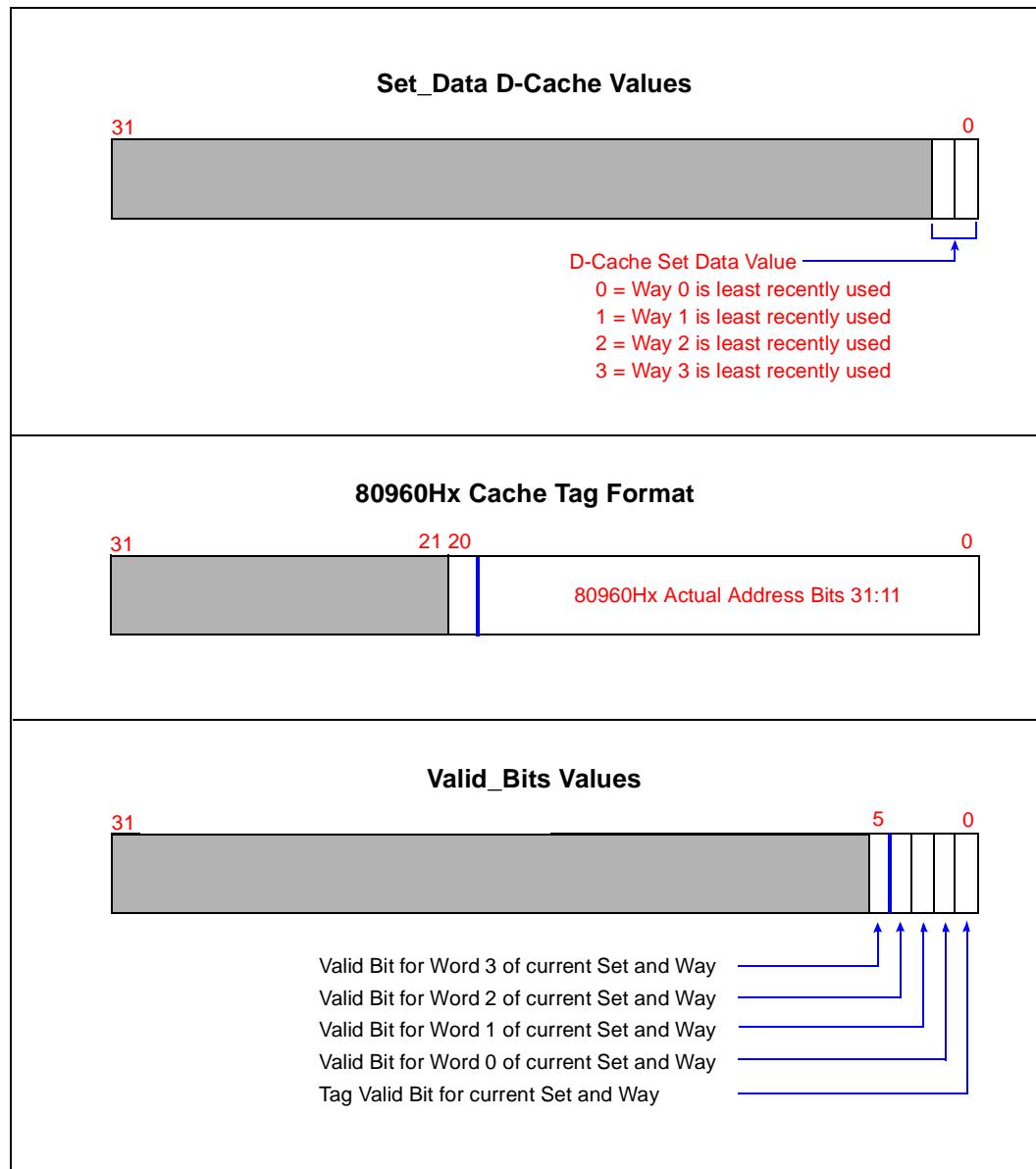
**Figure 6-1. dcctl *src1* and *src/dst* Formats**

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

bytes per atom	4
atoms per line	4
number of sets	128
number of ways	4
cache size	8-Kbytes

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

	Set_Data (Starting Set)	Destination Address (DA)
	Tag (Starting set)	DA + 4H
	Valid Bits (Starting set)	DA + 8H
Way 0	Word 0	DA + CH
	Word 1	DA + 10H
	Word 2	DA + 14H
	Word 3	DA + 18H
	Tag (Starting set)	DA + 1CH
	Valid Bits (Starting set)	DA + 20H
Way 1	Word 0	DA + 24H
	Word 1	DA + 28H
	Word 2	DA + 2cH
	Word 3	DA + 30H
	Tag (Starting set)	DA + 34H
	Valid Bits (Starting set)	DA + 38H
Way 2	Word 0	DA + 3CH
	Word 1	DA + 40H
	Word 2	DA + 44H
	Word 3	DA + 48H
	Tag (Starting set)	DA + 4CH
	Valid Bits (Starting set)	DA + 50H
Way 3	...	...

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



```

Action      f (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;
}

```

Action (continued)	<pre> 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 &gt;= Dcache_max_sets) end = Dcache_max_sets - 1;             if (start &gt; end) generate_fault                 (OPERATION.INVALID_OPERAND);             memadr = src2;# Must be word-aligned.             if (0x3 &amp; memadr! = 0)                 generate_fault(OPERATION.INVALID_OPERAND)             for (set = start; set &lt;= end; set++){                 # Set_Data is described at end of this code flow.                 memory[memadr] = Set_Data[set];                 memadr += 4;                 for (way = 0; way &lt; numb_ways; way++)                     {memory[memadr] = tags[set][way];                      memadr += 4;                      memory[memadr] = valid_bits[set][way];                      memadr += 4;                      for (word = 0; word &lt; words_in_line; word++)                          {memory[memadr] =                            Dcache_line[set][way][word];                            memadr += 4;                          }                      }                 break;             }         case 8:      # invalidate the lines that came from LMTs that had DCIIR set                     # at the time the line was allocated.                     # NOTE : for compatibility with future products that have                     # several independent regions, the value of src2 should be one.                     invalidate_DCIIR_lines_in_DCache;                     break;         default:     # Reserved.                     generate_fault(OPERATION.INVALID_OPERAND);                     break;             }         order_wrt(subsequent_operations)     </pre>
Faults	<p>STANDARD Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5.</a></p> <p>TYPE.MISMATCH Attempt to execute instruction while not in supervisor mode.</p> <p>OPERATION.INVALID_OPERAND</p>
Example	<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>
Opcode	<b>dcctl</b> 65CH REG
See Also	sysctl, dcinva

## 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 will be 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 80960RP, 80960Hx and 80960Jx processor families only, and may or may not be implemented on future i960 processors.

## 6.2.24 dcinva (80960Hx-Specific Instruction)

Mnemonic	<b>dcinva</b>	Data Cache Invalidate by Address.
Format	<b>dcinva</b> src1 mem efa	
Description		An effective linear address contained in <i>src1</i> is sent to the data cache. The quad word of data in the data cache in which the address falls is then invalidated.
Action		# beginning of quad word including effective_address line_start = effective_address & !0xF; invalidate_Dcache_quadword(line_start);
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .
Example		<pre># Placing dcinva in a critical section ensures data # coherency in case main memory is updated by an # external agent directly.  .set mem1, 0x34504      # address to invalidate .set mem1_sema, 1        # semaphore no. for address mem1 # label: wait_on_semaphore routine to wait on a # semaphore # label: signal_semaphore routine to release a # semaphore  # Wait on mem1_sema semaphore lda    mem1_sema, g1 call   wait_on_semaphore  # now in critical section dcinva mem1  # use the resource here, taking advantage of its caching  # Signal mem1_sema call   signal_semaphore</pre>
Opcode	<b>dcinva</b>	ADH            MEM
See Also		dcctl
Notes		This instruction is implemented on the i960 Hx processor family only and may or may not be implemented on future i960 processors.

## 6.2.25 divi, divo

Mnemonic	<b>divi</b>	Divide Integer
	<b>divo</b>	Divide Ordinal
Format	<b>div*</b>	<i>src1</i> , <i>src2</i> , <i>dst</i> reg/lit/sfr reg/lit/sfr reg/sfr
Description		Divides <i>src2</i> value by <i>src1</i> value and stores result in <i>dst</i> . Remainder is discarded.
		For <b>divi</b> , an integer-overflow fault can be signaled.
Action	<b>divo:</b> if (src1 == 0) {     dst = undefined_value; generate_fault (ARITHMETIC.ZERO_DIVIDE); else dst = src2/src1;	
	<b>divi:</b> if (src1 == 0) {     dst = undefined_value; generate_fault (ARITHMETIC.ZERO_DIVIDE);} else if ((src2 == -2**31) && (src1 == -1)) {     dst = -2**31	
		if (AC.om == 1) AC.of = 1; else generate_fault (ARITHMETIC.OVERFLOW);
		}
		else dst = src2 / src1;
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .
	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> .
Example	<code>divo r3, r8, r13 # r13 = r8/r3</code>	
Opcode	<b>divi</b>	74BH
	<b>divo</b>	70BH
See Also	ediv, mulo, muli, emul	

## 6.2.26 ediv

Mnemonic	<b>ediv</b>	Extended Divide
Format	<b>ediv</b>	<i>src1</i> , <i>src2</i> , <i>dst</i> reg/lit/sfr        reg/lit/sfr        reg/sfr
Description	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... or sf0, sf2,...). <i>src1</i> value is a normal ordinal (i.e., 32 bits).	
The result consists of a one-word remainder and a one-word quotient. Remainder is stored in the register designated by <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, ... or sf0, sf2, ...).		
This instruction performs ordinal arithmetic.		
If this operation overflows (quotient or remainder do not fit in 32 bits), no fault is raised and the result is undefined.		
Action	<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>	
Faults	<b>STANDARD</b> Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5.</a> <b>ARITHMETIC.ZERO_DIVIDE</b> The <i>src1</i> operand is 0.	
Example	<pre>ediv g3, g4, g10    # g10 = remainder of g4,g5/g3                       # g11 = quotient of g4,g5/g3</pre>	
Opcode	<b>ediv</b>	671H           REG
See Also	emul, divi, divo	

## 6.2.27 emul

Mnemonic	<b>emul</b>	Extended Multiply
Format	<b>emul</b>	$src1,$ reg/lit/sfr $src2,$ reg/lit/sfr $dst$ reg/sfr
Description	Multiplies $src2$ by $src1$ and stores the result in $dst$ . Result is a long ordinal (64 bits) stored in two adjacent registers. $dst$ specifies lower numbered register, which receives the result's least significant bits. $dst$ must be an even numbered register (i.e., g0, g2, ... r4, r6, r8, ... or sf0, sf2, ...).	
This instruction performs ordinal arithmetic.		
Action	<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>	
Faults	STANDARD Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .	
Example	<code>emul r4, r5, g2</code> $\# g2, g3 = r4 * r5.$	
Opcode	<b>emul</b> 670H REG	
See Also	<b>ediv, muli, mulo</b>	

## 6.2.28 eshro

Mnemonic	<b>eshro</b>	Extended Shift Right Ordinal
Format	<b>eshro</b> <i>src1</i> , <i>src2</i> , <i>dst</i> reg/lit/sfr      reg/lit/sfr      reg/sfr	
Description	Shifts <i>src2</i> right by ( <i>src1</i> mod 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> .	
Action	<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>	
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .
Example	<pre>eshro g3, g4, g11 # g11 = g4,5 shifted right by # (g3 MOD 32).</pre>	
Opcode	<b>eshro</b>	5D8H
See Also	SHIFT, extract	
Notes	This core instruction is not implemented on the Kx and Sx 80960 processors.	

## 6.2.29 extract

Mnemonic	<b>extract</b>	Extract
Format	<b>extract</b>	<i>bitpos</i> reg/lit/sfr <i>len</i> reg/lit/sfr <i>src/dst</i> reg
Description	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.	
Action	$\text{src\_dst} = (\text{src\_dst} >> \min(\text{bitpos}, 32))$ $\quad \& \sim (0xFFFFFFFF << \text{len});$	
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .
Example	<pre>extract 5, 12, g4    # g4 = g4 with bits 5 through # 16 shifted right.</pre>	
Opcode	<b>extract</b>	651H
See Also	modify	

## 6.2.30 FAULT<cc>

Mnemonic	<b>faulte{.t .f}</b>	Fault If Equal																											
	<b>faultne{.t .f}</b>	Fault If Not Equal																											
	<b>faultl{.t .f}</b>	Fault If Less																											
	<b>faultle{.t .f}</b>	Fault If Less Or Equal																											
	<b>faultg{.t .f}</b>	Fault If Greater																											
	<b>faultge{.t .f}</b>	Fault If Greater Or Equal																											
	<b>faulto{.t .f}</b>	Fault If Ordered																											
	<b>faultno{.t .f}</b>	Fault If Not Ordered																											
Format	<b>fault*{.t .f}</b>																												
Description	Raises a constraint-range fault if the logical AND of the condition code and opcode's mask part is not zero. For <b>faultno</b> (unordered), fault is raised if condition code is equal to $000_2$ .																												
	Optional .t or .f suffix may be appended to the mnemonic. Use .t to speed up execution when these instructions usually fault. Use .f to speed up execution when these instructions usually do not fault. If a suffix is not provided, the assembler is free to provide one.																												
	<b>faulto</b> and <b>faultno</b> 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.																												
	<table border="1"> <thead> <tr> <th>Instruction</th><th>Mask</th><th>Condition</th></tr> </thead> <tbody> <tr> <td><b>faultno</b></td><td><math>000_2</math></td><td>Unordered</td></tr> <tr> <td><b>faultg</b></td><td><math>001_2</math></td><td>Greater</td></tr> <tr> <td><b>faulte</b></td><td><math>010_2</math></td><td>Equal</td></tr> <tr> <td><b>faultge</b></td><td><math>011_2</math></td><td>Greater or equal</td></tr> <tr> <td><b>faultl</b></td><td><math>100_2</math></td><td>Less</td></tr> <tr> <td><b>faultne</b></td><td><math>101_2</math></td><td>Not equal</td></tr> <tr> <td><b>faultle</b></td><td><math>110_2</math></td><td>Less or equal</td></tr> <tr> <td><b>faulto</b></td><td><math>111_2</math></td><td>Ordered</td></tr> </tbody> </table>	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	
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	<b>For all except faultno:</b> if(mask && AC.cc != $000_2$ ) generate_fault(CONSTRAINT.RANGE);  <b>faultno:</b> if(AC.cc == $000_2$ ) generate_fault(CONSTRAINT.RANGE);																												
Faults	STANDARD CONSTRAINT.RANGE	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> . If condition being tested is true.																											

Example	# Assume (AC.cc AND 110 <sub>2</sub> ) ≠ 000 <sub>2</sub> faultle # Generate CONSTRAINT_RANGE fault		
Opcode	<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>		

## 6.2.31 flushreg

Mnemonic	<b>flushreg</b> Flush Local Registers		
Format	<b>flushreg</b>		
Description	Copies the contents of every cached register set, except the current set, to its associated stack frame in memory. The entire register cache is then marked as purged (or invalid). On a return to a stack frame for which the local registers are not cached, the processor reloads the locals from memory.		
<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.			
Action	Each local cached register set except the current one is flushed to its associated stack frame in memory and marked as purged, meaning that they are reloaded from memory if and when they become the current local register set.		
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a>	
Example	<code>flushreg</code>		
Opcode	<b>flushreg</b>	66DH	REG

## 6.2.32 fmark

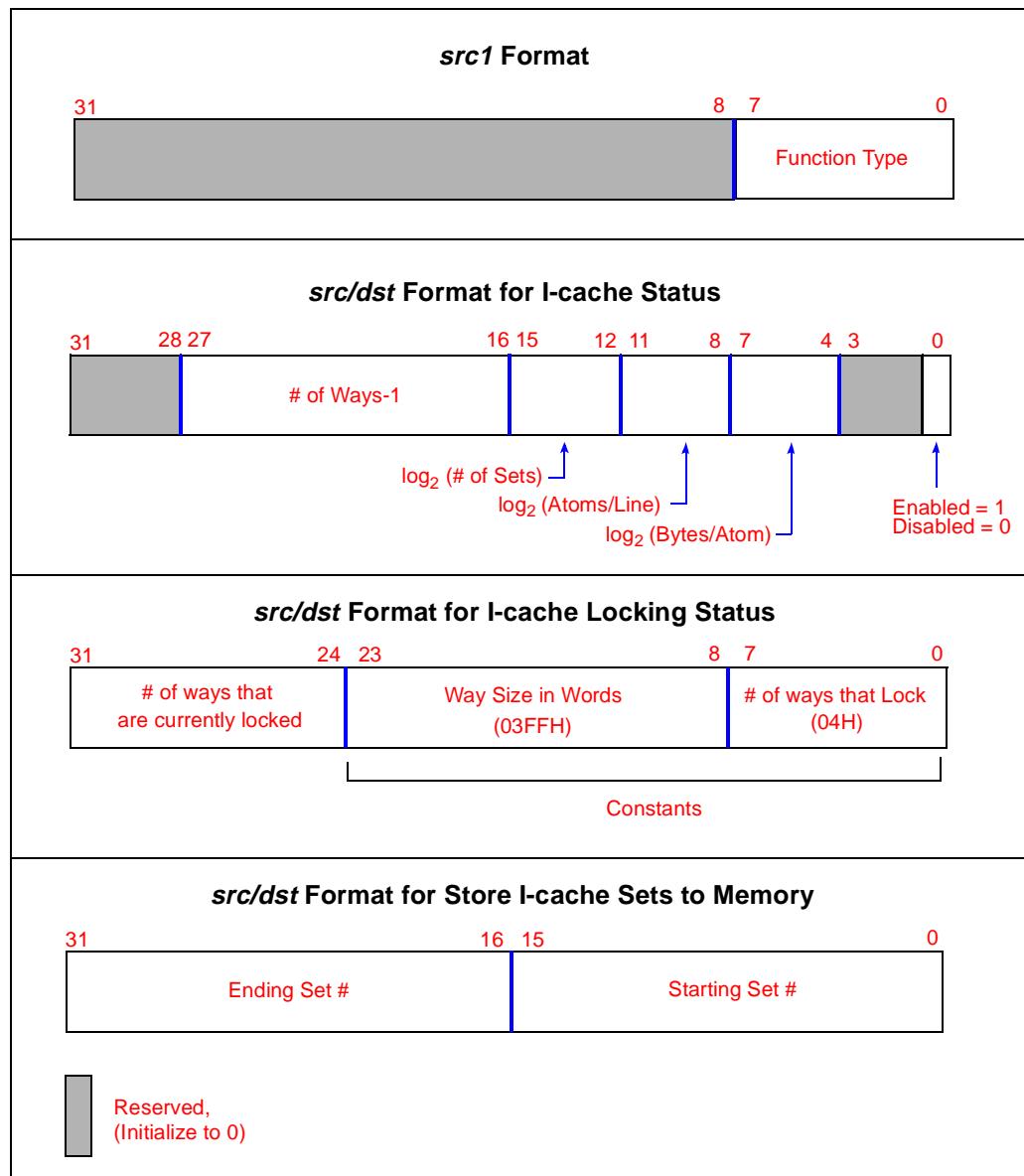
Mnemonic	<b>fmark</b> Force Mark		
Format	<b>fmark</b>		
Description	Generates a mark trace event. Causes a mark trace event to be generated, regardless of mark trace mode flag setting, providing the trace enable bit, bit 0 in the Process Controls, is set.		
	For more information on trace fault generation, refer to <a href="#">Chapter 9, “Tracing and Debugging”</a> .		
Action	A mark trace event is generated, independent of the setting of the mark-trace-mode flag.		
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> . TRACE.MARK A TRACE.MARK fault is generated if PC.te=1.	
Example	<pre># Assume PC.te = 1 fmark # Mark trace event is generated at this point in the # instruction stream.</pre>		
Opcode	<b>fmark</b>	66CH	REG
See Also	<a href="#">mark</a>		

### 6.2.33 icctl

Mnemonic	<b>icctl</b>	Instruction-cache Control		
Format	<b>icctl</b>	<i>src1</i> , reg/lit/sfr	<i>src2</i> , reg/lit/sfr	<i>src/dst</i> reg/sfr
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 <i>src1</i> . Some operations also use <i>src2</i> and <i>src/dst</i> . 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	<b>src1</b>	<b>src2</b>	<b>src/dst</b>
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 ways to lock.
Get I-cache status	4	NA	<i>dst</i> : Receives status (see <a href="#">Figure 6-4</a> ).
Get I-cache locking status	5	NA	<i>dst</i> : Receives status (see <a href="#">Figure 6-4</a> )
Store I-cache sets to memory	6	Destination address for cache sets	<i>src</i> : I-cache set #'s to be stored (see <a href="#">Figure 6-4</a> ).

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

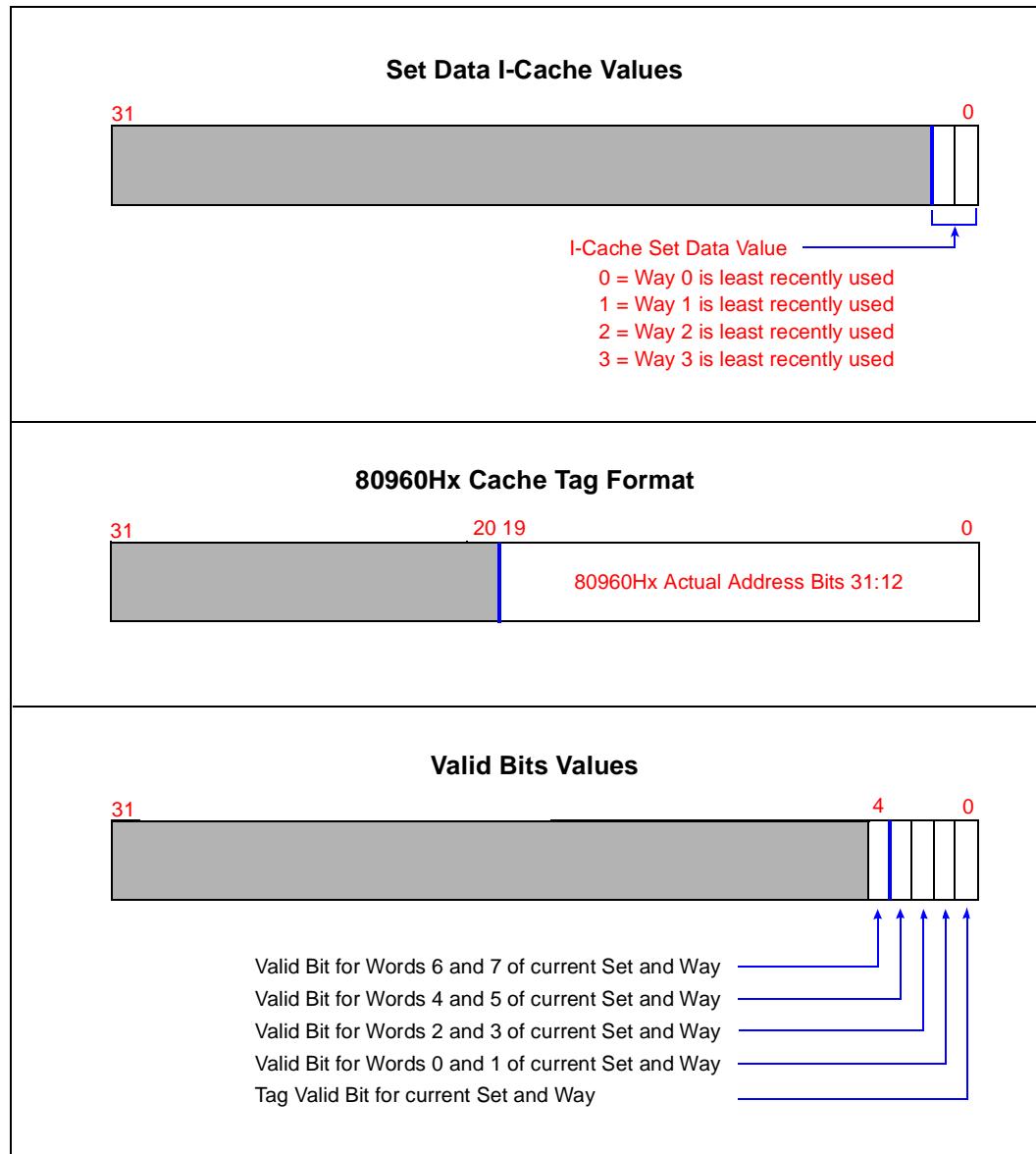
**Table 6-9. icctl Status Values and Instruction Cache Parameters**

Value	Value on 80960Hx
bytes per atom	4
atoms per line	8
number of sets	128
number of ways	4
cache size	16-Kbytes
Status[0] (enable/disable)	0 or 1
Status[1:3] (reserved)	0
Status[7:4] ( $\log_2(\text{bytes per atom})$ )	2
Status[11:8] ( $\log_2(\text{atoms per line})$ )	3
Status[15:12] ( $\log_2(\text{number of sets})$ )	8
Status[27:16] (number of ways - 1)	3
Lock Status[7:0] (number of blocks that lock)	
Lock Status[23:8] (block size in words)	1024
Lock Status[31:24] (number of blocks that are locked)	0-4

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

Way 0	Set_Data [Starting Set]	Destination Address (DA)
	Tag (Starting set)	DA + 4H
	Valid Bits (Starting set)	DA + 8H
	Word 0	DA + CH
	...	...
	Word 6	DA + 24H
	Word 7	DA + 28H
	Tag (Starting set)	DA + 2CH
Way 1	Valid Bits (Starting set)	DA + 30H
	Word 0	DA + 34CH
	...	...
	Word 6	DA + 4CH
	Word 7	DA + 50H
	Tag (Starting set)	DA + 54H
	Valid Bits (Starting set)	DA + 58H
	Word 0	DA + 5CH
Way 2	...	...
	Word 6	DA + 74H
	Word 7	DA + 78H
	Tag (Starting set)	DA + 7CH
	Valid Bits (Starting set)	DA + 80H
	Word 0	DA + 84H
	...	...
	Word 6	DA + 9CH
Way 3	Word 7	DA + A0H
	Set_Data [Starting Set + 1]	...
	...	...

**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 Hx, src2 is aligned to a quad word boundary
              aligned_addr = src2 & 0xFFFFFFFF0;
              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;
}

```

Action (cont'd)	<pre> 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 &gt;= Icache_max_sets)     end = Icache_max_sets - 1; if (start &gt; end)     generate_fault(OPERATION.INVALID_OPERAND); memadr = src2;           # Must be word-aligned. if(0x3 &amp; memadr != 0)     generate_fault(OPERATION.INVALID_OPERAND); for (set = start; set &lt;= end; set++){     # Set_Data is described at end of this code flow.     memory[memadr] = Set_Data[set];     memadr += 4;     for (way = 0; way &lt; numb_ways; way++)         {memory[memadr] = tags[set][way];          memadr += 4;          memory[memadr] = valid_bits[set][way];          memadr += 4;          for (word = 0; word &lt; words_in_line;               word++)              {memory[memadr] =               Icache_line[set][way][word];                memadr += 4;              }          clrbit 30, ccon # disable instruction cache         } break; default: # Reserved. generate_fault(OPERATION.INVALID_OPERAND); break;}</pre>		
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .	
	TYPE.MISMATCH	Attempt to execute instruction while not in supervisor mode.	
Example	icctl g0,g1,g2	<pre># g0 = 3, g1=0x10000000, g2=1 # Load and lock 1 block of cache # (one way) with # location of code at starting # 0x10000000.</pre>	
Opcode	<b>icctl</b>	65BH	REG
See Also	<b>sysctl</b>		
Notes	This instruction is implemented on the 80960RP, 80960Hx and 80960Jx processor families only, and may or may not be implemented on future i960 processors.		

## 6.2.34 intctl

Mnemonic	<b>intctl</b> Global Enable and Disable of Interrupts	
Format	<b>intctl</b>	<i>src1</i> <i>dst</i> reg/lit/sfr       reg/sfr
Description	Globally enables, disables or returns the current status of interrupts depending on the value of <i>src1</i> . Returns the previous interrupt enable state (1 for enabled or 0 for disabled) in <i>dst</i> . 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.)	
Action	<pre> 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; </pre>	
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5.</a>
	TYPE.MISMATCH	Attempt to execute instruction while not in supervisor mode.

Example		# ICON.gie = 0, interrupts enabled intctl 0, g4 # Disable interrupts (ICON.gie = 1) # g4 = 1
Opcode	<b>intctl</b>	658H REG
See Also		<b>intdis, inten</b>
Notes		This instruction is implemented on the 80960RP, 80960Hx and 80960Jx processor families only, and may or may not be implemented on future i960 processors.

## 6.2.35 intdis

Mnemonic	<b>intdis</b> Global Interrupt Disable		
Format	<b>intdis</b>		
Description	Globally disables interrupts and ensures that the change takes effect before the instruction completes. This operation is implemented by setting ICON.gie to one.		
Action	<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>		
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .	
	TYPE.MISMATCH	Attempt to execute instruction while not in supervisor mode.	
Example	<pre># ICON.gie = 0, interrupts enabled intdis      # Disable interrupts.             # ICON.gie = 1</pre>		
Opcode	<b>intdis</b>	5B4H	REG
See Also	<a href="#"><b>intctl</b></a> , <a href="#"><b>inten</b></a>		
Notes	This instruction is implemented on the 80960RP, 80960Hx and 80960Jx processor families only, and may or may not be implemented on future i960 processors.		

## 6.2.36 inten

Mnemonic	<b>inten</b> global interrupt enable		
Format	<b>inten</b>		
Description	Globally enables interrupts and ensures that the change takes effect before the instruction completes. This operation is implemented by clearing ICON.gie to zero.		
Action	<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>		
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .	
	TYPE.MISMATCH	Attempt to execute instruction while not in supervisor mode.	
Example		# ICON.gie = 1, interrupts disabled. inten # Enable interrupts. # ICON.gie = 0	
Opcode	<b>inten</b>	5B5H	REG
See Also	<b>intctl</b> , <b>intdis</b>		
Notes	This instruction is implemented on the 80960RP, 80960Hx and 80960Jx processor families only, and may or may not be implemented on future i960 processors.		

## 6.2.37 LOAD

Mnemonic	<b>Id</b>	Load
	<b>Idob</b>	Load Ordinal Byte
	<b>Idos</b>	Load Ordinal Short
	<b>Idib</b>	Load Integer Byte
	<b>Idis</b>	Load Integer Short
	<b>Idl</b>	Load Long
	<b>Idt</b>	Load Triple
	<b>Idq</b>	Load Quad
Format	<b>Id*</b>	<i>src</i> , mem <i>dst</i> reg
Description	Copies byte or byte string from memory into a register or group of successive registers.	
	The <i>src</i> operand specifies the address of first byte to be loaded. The full range of addressing modes may be used in specifying <i>src</i> . Refer to <a href="#">Chapter 2, “Data Types and Memory Addressing Modes”</a> for more information.	
	<i>dst</i> specifies a register or the first (lowest numbered) register of successive registers.	
	<b>Idob</b> and <b>Idib</b> load a byte and <b>Idos</b> and <b>Idis</b> 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.	
	<b>Id</b> , <b>Idl</b> , <b>Idt</b> and <b>Idq</b> instructions copy 4, 8, 12 and 16 bytes, respectively, from memory into successive registers.	
	For <b>Idl</b> , <i>dst</i> must specify an even numbered register (i.e., g0, g2, ... or r4, r6, ...). For <b>Idt</b> and <b>Idq</b> , <i>dst</i> 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 <b>Idl</b> , <b>Idt</b> or <b>Idq</b> .	

Action	<b>Id:</b> <pre>dst = read_memory(effective_address)[31:0]; if((effective_address[1:0] != 0<sub>2</sub>) &amp;&amp; unaligned_fault_enabled)     generate_fault(OPERATION.UNALIGNED);</pre> <b>Idob:</b> <pre>dst[7:0] = read_memory(effective_address)[7:0]; dst[31:8] = 0x000000;</pre> <b>Idib:</b> <pre>dst[7:0] = read_memory(effective_address)[7:0]; if(dst[7] == 0)     dst[31:8] = 0x000000; else     dst[31:8] = 0xFFFFFFF;</pre> <b>Idos:</b> <pre>dst = read_memory(effective_address)[15:0]; # Order depends on endianism. See # <a href="#">Section 2.2.2, “Byte Ordering” on page 2-4</a> dst[31:16] = 0x0000; if((effective_address[0] != 0<sub>2</sub>) &amp;&amp; unaligned_fault_enabled)     generate_fault(OPERATION.UNALIGNED);</pre> <b>Idis:</b> <pre>dst[15:0] = read_memory(effective_address)[15:0]; # Order depends on endianism. See # <a href="#">Section 2.2.2, “Byte Ordering” on page 2-4</a> if(dst[15] == 0<sub>2</sub>)     dst[31:16] = 0x0000; else     dst[31:16] = 0xFFFF; if((effective_address[0] != 0<sub>2</sub>) &amp;&amp; unaligned_fault_enabled)     generate_fault(OPERATION.UNALIGNED);</pre> <b>Idl:</b> <pre>if((reg_number(dst) % 2) != 0)     generate_fault(OPERATION.INVALID_OPERAND); # dst modified. else {     dst = read_memory(effective_address)[31:0];     dst_+_1 = read_memory(effective_address_+_4)[31:0];     if((effective_address[2:0] != 000<sub>2</sub>) &amp;&amp; unaligned_fault_enabled)         generate_fault(OPERATION.UNALIGNED); }</pre>
--------	---

Action (continued)	<pre><b>ldt:</b> if((reg_number(dst) % 4) != 0)     generate_fault(OPERATION.INVALID_OPERAND);     # dst modified.  <b>else</b> {     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] != 0000<sub>2</sub>) &amp;&amp; unaligned_fault_enabled)         generate_fault(OPERATION.UNALIGNED); }</pre> <pre><b>ldq:</b> if((reg_number(dst) % 4) != 0)     generate_fault(OPERATION.INVALID_OPERAND);     # dst modified.  <b>else</b> {     dst = read_memory(effective_address)[31:0];     # Order depends on endianism.     # See <a href="#">Section 2.2.2, “Byte Ordering” on page 2-4</a>     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] != 0000<sub>2</sub>) &amp;&amp; unaligned_fault_enabled)         generate_fault(OPERATION.UNALIGNED);</pre>																								
Faults	STANDARD Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> . OPERATION.UNALIGNED OPERATION.INVALID_OPERAND																								
Example	<b>ldl</b> 2450 (r3), r10      # r10, r11 = r3 + 2450 in # memory																								
Opcode	<table> <tbody> <tr> <td><b>ld</b></td><td>90H</td><td>MEM</td></tr> <tr> <td><b>ldob</b></td><td>80H</td><td>MEM</td></tr> <tr> <td><b>ldos</b></td><td>88H</td><td>MEM</td></tr> <tr> <td><b>ldib</b></td><td>C0H</td><td>MEM</td></tr> <tr> <td><b>ldis</b></td><td>C8H</td><td>MEM</td></tr> <tr> <td><b>ldl</b></td><td>98H</td><td>MEM</td></tr> <tr> <td><b>ldt</b></td><td>A0H</td><td>MEM</td></tr> <tr> <td><b>ldq</b></td><td>B0H</td><td>MEM</td></tr> </tbody> </table>	<b>ld</b>	90H	MEM	<b>ldob</b>	80H	MEM	<b>ldos</b>	88H	MEM	<b>ldib</b>	C0H	MEM	<b>ldis</b>	C8H	MEM	<b>ldl</b>	98H	MEM	<b>ldt</b>	A0H	MEM	<b>ldq</b>	B0H	MEM
<b>ld</b>	90H	MEM																							
<b>ldob</b>	80H	MEM																							
<b>ldos</b>	88H	MEM																							
<b>ldib</b>	C0H	MEM																							
<b>ldis</b>	C8H	MEM																							
<b>ldl</b>	98H	MEM																							
<b>ldt</b>	A0H	MEM																							
<b>ldq</b>	B0H	MEM																							
See Also	<b>MOVE, STORE</b>																								

## 6.2.38 lda

Mnemonic	<b>lda</b>	Load Address
Format	<b>lda</b>	<i>src</i> , mem efa <i>dst</i> reg
Description	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> .	
		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.)
Action	dst = effective_address;	
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5.</a>
Example	<pre>lda 58 (g9), g1    # g1 = g9+58 lda 0x749, r8      # r8 = 0x749</pre>	
Opcode	<b>lda</b>	8CH              MEM

## 6.2.39 mark

Mnemonic	<b>mark</b>	Mark
Format	<b>mark</b>	
Description		Generates mark trace fault if mark trace mode is enabled. Mark trace mode is enabled if the PC register trace enable bit (bit 0) and the TC register mark trace mode bit (bit 7) are set.
		If mark trace mode is not enabled, <b>mark</b> behaves like a no-op.
		For more information on trace fault generation, refer to <a href="#">Chapter 9, “Tracing and Debugging”</a> .
Action		if(PC.te && TC.mk) generate_fault(TRACE.MARK)
Faults	STANDARD TRACE.MARK	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> . Trace fault is generated if PC.te=1 and TC.mk=1.
Example	# Assume that the mark trace mode is enabled. ld xyz, r4 addi r4, r5, r6 mark # Mark trace event is generated at this point in the # instruction stream.	
Opcode	<b>mark</b>	66BH
See Also		<b>fmark, modpc, modtc</b>

## 6.2.40 modac

Mnemonic	<b>modac</b>	Modify AC
Format	<b>modac</b>	<i>mask</i> , <i>src</i> , <i>dst</i> reg/lit/sfr        reg/lit/sfr        reg/sfr
Description	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> .	
Action	$\text{temp} = \text{AC};$ $\text{AC} = (\text{src} \& \text{mask})   (\text{AC} \& \sim\text{mask});$ $\text{dst} = \text{temp};$	
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5.</a>
Example	<pre>modac g1, g9, g12 # AC = g9, masked by g1. # g12 = initial value of AC.</pre>	
Opcode	<b>modac</b>	645H           REG
See Also	<b>modpc, modtc</b>	
Notes	Sets the condition code in the arithmetic controls.	

## 6.2.41 modi

Mnemonic	<b>modi</b>	Modulo Integer
Format	<b>modi</b>	<i>src1</i> , <i>src2</i> , <i>dst</i> reg/lit/sfr reg/lit/sfr reg/sfr
Description	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> .	
Action	<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>	
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5.</a>
	ARITHMETIC.ZERO_DIVIDE	The <i>src1</i> operand is zero.
Example	modi r9, r2, r5	# r5 = modulo (r2/r9)
Opcode	<b>modi</b>	749H REG
See Also	<b>divi, divo, remi, remo</b>	
Notes	<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

Mnemonic	<b>modify</b>	Modify
Format	<b>modify</b>	<i>mask</i> , reg/lit/sfr <i>src</i> , reg/lit/sfr <i>src/dst</i> reg
Description	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> .	
Action	$\text{src\_dst} = (\text{src} \& \text{mask})   (\text{src\_dst} \& \sim\text{mask});$	
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .
Example	<code>modify g8, g10, r4# r4 = g10 masked by g8.</code>	
Opcode	<b>modify</b>	650H
See Also	<a href="#">alterbit</a> , <a href="#">extract</a>	

## 6.2.43 modpc

Mnemonic	<b>modpc</b> Modify Process Controls			
Format	<b>modpc</b>	<i>src</i> , reg/lit/sfr	<i>mask</i> , reg/lit/sfr	<i>src/dst</i> reg
Description	<p>Reads and modifies the PC register as specified with mask 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 mask value. If mask=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 (PC) Register” on page 3-24</a>.</p>			
Action	<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>			
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .		
	TYPE.MISMATCH			
Example	modpc g9, g9, g8	# process controls = g8 # masked by g9.		
Opcode	<b>modpc</b>	655H	REG	
See Also	<b>modac, modtc</b>			
Notes	<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

Mnemonic	<b>modtc</b> Modify Trace Controls					
Format	<b>modtc</b>	<i>mask</i> , reg/lit/sfr	<i>src2</i> , reg/lit/sfr	<i>dst</i> reg/sfr		
Description	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> .					
The changed trace controls may take effect immediately or may be delayed. If delayed, the changed trace controls may not take effect until after the first non-branching instruction is fetched from memory or after four non-branching instructions are executed.						
For more information on the trace controls, refer to <a href="#">Chapter 8, “Faults”</a> and <a href="#">Chapter 9, “Tracing and Debugging”</a> .						
Action	<pre>mode_bits = 0x000000FE; event_flags = 0x0F00FF00 temp = TC; tempa = (event_flags &amp; TC &amp; mask)   (mode_bits &amp; mask); TC = (tempa &amp; src2)   (TC &amp; ~tempa); dst = temp;</pre>					
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults”</a> on page 6-5.				
Example	<pre>modtc g12, g10, g2# trace controls = g10 masked # by g12; previous trace # controls stored in g2.</pre>					
Opcode	<b>modtc</b>	654H	REG			
See Also	<a href="#"><b>modac</b></a> , <a href="#"><b>modpc</b></a>					

## 6.2.45 MOVE

Mnemonic	<b>mov</b>	Move
	<b>movl</b>	Move Long
	<b>movt</b>	Move Triple
	<b>movq</b>	Move Quad
Format	<b>mov*</b>	<i>src1</i> , reg/lit/sfr <i>dst</i> reg/sfr
Description	Copies the contents of one or more source registers (specified with <i>src</i> ) to one or more destination registers (specified with <i>dst</i> ).  For <b>movl</b> , <b>movt</b> and <b>movq</b> , <i>src1</i> and <i>dst</i> specify the first (lowest numbered) register of several successive registers. <i>src1</i> and <i>dst</i> registers must be even numbered (e.g., g0, g2, ... or r4, r6, ... or sf0, sf2, ...) for <b>movl</b> and an integral multiple of four (e.g., g0, g4, ... or r4, r8, ... or sf0, sf4, ...) for <b>movt</b> and <b>movq</b> .  The moved register values are unpredictable when: 1) the <i>src</i> and <i>dst</i> operands overlap; 2) registers are not properly aligned.	
Action	<pre> <b>mov:</b> if(is_reg(src1))     dst = src1; else {     dst[4:0] = src1;  #src1 is a 5-bit literal.     dst[31:5] = 0; } <b>movl:</b> 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; } </pre>	

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

## 6.2.46 muli, mulo

Mnemonic	<b>muli</b>	Multiply Integer
	<b>mulo</b>	Multiply Ordinal
Format	<b>mul*</b>	<i>src1,</i> reg/lit/sfr <i>src2,</i> reg/lit/sfr <i>dst</i> reg/sfr
Description	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 muli can signal an integer overflow.	
Action	<b>mulo:</b> $dst = (src2 * src1)[31:0];$  <b>muli:</b> $true\_result = (src1 * src2);$ $dst = true\_result[31:0];$ $if((true\_result > (2^{**31}) - 1) \parallel (true\_result < -2^{**31})) \# Check for overflow$ { $if(AC.om == 1)$ $AC.of = 1;$ $else$ $generate\_fault(ARITHMETIC.OVERFLOW);$ }	
Faults	STANDARD ARITHMETIC.OVERFLOW	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</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.
Example	<code>muli r3, r4, r9 # r9 = r4 * r3</code>	
Opcode	<b>muli</b>	741H
	<b>mulo</b>	701H
See Also	<b>emul, ediv, divi, divo</b>	

## 6.2.47 nand

Mnemonic	<b>nand</b> Nand			
Format	<b>nand</b>	<i>src1</i> , reg/lit/sfr	<i>src2</i> , reg/lit/sfr	<i>dst</i> reg/sfr
Description	Performs a bitwise NAND operation on <i>src2</i> and <i>src1</i> values and stores the result in <i>dst</i> .			
Action	$dst = \sim src2   \sim src1;$			
Faults	STANDARD Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .			
Example	<code>nand g5, r3, r7 # r7 = r3 NAND g5</code>			
Opcode	<b>nand</b>	58EH	REG	
See Also	<a href="#">and</a> , <a href="#">andnot</a> , <a href="#">nor</a> , <a href="#">not</a> , <a href="#">notand</a> , <a href="#">notor</a> , <a href="#">or</a> , <a href="#">ornot</a> , <a href="#">xnor</a> , <a href="#">xor</a>			

## 6.2.48 nor

Mnemonic	<b>nor</b>	Nor
Format	<b>nor</b>	<i>src1</i> , reg/lit/sfr <i>src2</i> , reg/lit/sfr <i>dst</i> reg/sfr
Description	Performs a bitwise NOR operation on the <i>src2</i> and <i>src1</i> values and stores the result in <i>dst</i> .	
Action	$dst = \sim src2 \& \sim src1;$	
Faults	STANDARD Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .	
Example	nor g8, 28, r5    # r5 = 28 NOR g8	
Opcode	<b>nor</b> 588H                REG	
See Also	<a href="#">and</a> , <a href="#">andnot</a> , <a href="#">nand</a> , <a href="#">not</a> , <a href="#">notand</a> , <a href="#">notor</a> , <a href="#">or</a> , <a href="#">ornot</a> , <a href="#">xnor</a> , <a href="#">xor</a>	

## 6.2.49 not, notand

Mnemonic	<b>not</b>	Not	
	<b>notand</b>	Not And	
Format	<b>not</b>	<i>src1,</i> reg/lit/sfr	<i>dst</i> reg/sfr
	<b>notand</b>	<i>src1,</i> reg/lit/sfr	<i>src2,</i> reg/lit/sfr <i>dst</i> reg/sfr
Description	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> .		
Action	<b>not:</b> dst = ~src1;  <b>notand:</b> dst = ~src2 & src1;		
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .	
Example	not g2, g4 notand r5, r6, r7	# g4 = NOT g2 # r7 = NOT r6 AND r5	
Opcode	<b>not</b> 58AH <b>notand</b> 584H	REG REG	
See Also	<a href="#">and</a> , <a href="#">andnot</a> , <a href="#">nand</a> , <a href="#">nor</a> , <a href="#">notor</a> , <a href="#">or</a> , <a href="#">ornot</a> , <a href="#">xnor</a> , <a href="#">xor</a>		

## 6.2.50 notbit

Mnemonic	<b>notbit</b> Not Bit		
Format	<b>notbit</b>	<i>bitpos,</i> reg/lit/sfr	<i>src2,</i> reg/lit/sfr
Description	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.		
Action	$dst = src2 \wedge 2^{**}(src1\%32);$		
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .	
Example	notbit r3, r12, r7# r7 = r12 with the bit # specified in r3 toggled.		
Opcode	<b>notbit</b>	580H	REG
See Also	<a href="#">alterbit</a> , <a href="#">chkbit</a> , <a href="#">clrbit</a> , <a href="#">setbit</a>		

## 6.2.51 notor

Mnemonic	<b>notor</b>	Not Or
Format	<b>notor</b> <i>src1</i> , reg/lit/sfr	<i>src2</i> , reg/lit/sfr <i>dst</i> reg/sfr
Description	Performs a bitwise NOTOR operation on <i>src2</i> and <i>src1</i> values and stores result in <i>dst</i> .	
Action	dst = ~src2   src1;	
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .
Example	notor g12, g3, g6 # g6 = NOT g3 OR g12	
Opcode	<b>notor</b>	58DH      REG
See Also	<a href="#">and</a> , <a href="#">andnot</a> , <a href="#">nand</a> , <a href="#">nor</a> , <a href="#">not</a> , <a href="#">notand</a> , <a href="#">or</a> , <a href="#">ornot</a> , <a href="#">xnor</a> , <a href="#">xor</a>	

## 6.2.52 or, ornot

Mnemonic	<b>or</b> <b>ornot</b>	Or Or Not	
Format	<b>or</b>	<i>src1</i> , reg/lit/sfr	<i>src2</i> , reg/lit/sfr
	<b>ornot</b>	<i>src1</i> , reg/lit/sfr	<i>src2</i> , reg/lit/sfr
Description	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> .		
Action	<b>or:</b> $dst = src2 \mid src1;$  <b>ornot:</b> $dst = src2 \mid \sim src1;$		
Faults	STANDARD		Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .
Example	<pre>or 14, g9, g3      # g3 = g9 OR 14 ornot r3, r8, r11 # r11 = r8 OR NOT r3</pre>		
Opcode	<b>or</b>	587H	REG
	<b>ornot</b>	58BH	REG
See Also	<a href="#">and</a> , <a href="#">andnot</a> , <a href="#">nand</a> , <a href="#">nor</a> , <a href="#">not</a> , <a href="#">notand</a> , <a href="#">notor</a> , <a href="#">xnor</a> , <a href="#">xor</a>		

## 6.2.53 remi, remo

Mnemonic	<b>remi</b>	Remainder Integer
	<b>remo</b>	Remainder Ordinal
Format	<b>rem*</b>	<i>src1</i> , reg/lit/sfr <i>src2</i> , reg/lit/sfr <i>dst</i> reg/sfr
Description		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> .
Action		<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> ;
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5.</a> ARITHMETIC.ZERO_DIVIDE     The <i>src1</i> operand is 0.
Example	<b>remo r4, r5, r6</b>	# <i>r6</i> = <i>r5</i> rem <i>r4</i>
Opcode	<b>remi</b> 748H <b>remo</b> 708H	REG REG
See Also		<b>modi</b>
Notes		<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	<b>ret</b>	Return
Format	<b>ret</b>	
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	<pre>implicit_syncf(); if(pfp.p &amp;&amp; PC.te &amp;&amp; TC.p) {     pfp.p = 0;     generate_fault(TRACE.PRERETURN); } switch(return_status_field) {     case 000<sub>2</sub>:      #local return         get_FP_and_IP();         break;     case 001<sub>2</sub>:      #fault return         tempa = memory(FP-16);         tempb = memory(FP-12);         get_FP_and_IP();         AC = tempb;         if(execution_mode == supervisor)             PC = tempa;         break;     case 010<sub>2</sub>:      #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; }</pre>
--------	---

Action (continued)	<pre> case 011<sub>2</sub>:      # 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 100<sub>2</sub>:      #reserved - unpredictable behavior     break; case 101<sub>2</sub>:      #reserved - unpredictable behavior     break; case 110<sub>2</sub>:      #reserved - unpredictable behavior     break; case 111<sub>2</sub>:      #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; } </pre>		
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5.</a>	
Example	<b>ret</b>	# Program control returns to # context of calling procedure.	
Opcode	<b>ret</b>	0AH	CTRL
See Also	<b>call, calls, callx</b>		

## 6.2.55 rotate

Mnemonic	<b>rotate</b>	Rotate		
Format	<b>rotate</b>	<i>len,</i> reg/lit/sfr	<i>src2,</i> reg/lit/sfr	<i>dst</i> reg/sfr
Description	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.			
Action	<i>src2</i> is rotated by <i>len</i> mod 32. This value is stored in <i>dst</i> .			
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .		
Example	rotate 13, r8, r12# r12 = r8 with bits rotated # 13 bits to left.			
Opcode	<b>rotate</b>	59DH	REG	
See Also	<a href="#">SHIFT, eshro</a>			

## 6.2.56 scanbit

Mnemonic	<b>scanbit</b> Scan For Bit		
Format	<b>scanbit</b>	<i>src1,</i>	<i>dst</i>
		reg/lit/sfr	reg/sfr
Description	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> .		
Action	<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>		
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .	
Example	<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>		
Opcode	<b>scanbit</b>	641H	REG
See Also	<b>spanbit, setbit</b>		
Notes	Sets the condition code in the arithmetic controls.		

## 6.2.57 scanbyte

Mnemonic	<b>scanbyte</b>	Scan Byte Equal
Format	<b>scanbyte</b> <i>src1</i> , <i>src2</i> reg/lit/sfr      reg/lit/sfr	
Description		Performs byte-by-byte comparison of <i>src1</i> and <i>src2</i> and sets condition code to $010_2$ if any two corresponding bytes are equal. If no corresponding bytes are equal, condition code is set to $000_2$ .
Action		<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>
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .
Example		# Assume r9 = 0x11AB1100 scanbyte 0x00AB0011, r9# AC.cc = 010 <sub>2</sub>
Opcode	<b>scanbyte</b>	5ACH      REG
See Also		<b>bswap</b>
Side Effects		Sets the condition code in the arithmetic controls.

## 6.2.58 SEL<cc>

Mnemonic	<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																											
Format	<b>sel*</b>	<i>src1</i> , reg/lit/sfr <i>src2</i> , reg/lit/sfr <i>dst</i> reg/sfr																											
Description	Selects either <i>src1</i> or <i>src2</i> to be stored in <i>dst</i> 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 <i>src2</i> is stored in the destination. Else, the value of <i>src1</i> is stored in the destination.																												
<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: center;">Instruction</th> <th style="text-align: center;">Mask</th> <th style="text-align: center;">Condition</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;"><b>selno</b></td><td style="text-align: center;"><math>000_2</math></td><td style="text-align: center;">Unordered</td></tr> <tr> <td style="text-align: center;"><b>selg5</b></td><td style="text-align: center;"><math>001_2</math></td><td style="text-align: center;">Greater</td></tr> <tr> <td style="text-align: center;"><b>sele</b></td><td style="text-align: center;"><math>010_2</math></td><td style="text-align: center;">Equal</td></tr> <tr> <td style="text-align: center;"><b>selge</b></td><td style="text-align: center;"><math>011_2</math></td><td style="text-align: center;">Greater or equal</td></tr> <tr> <td style="text-align: center;"><b>sell</b></td><td style="text-align: center;"><math>100_2</math></td><td style="text-align: center;">Less</td></tr> <tr> <td style="text-align: center;"><b>selne</b></td><td style="text-align: center;"><math>101_2</math></td><td style="text-align: center;">Not equal</td></tr> <tr> <td style="text-align: center;"><b>selle</b></td><td style="text-align: center;"><math>110_2</math></td><td style="text-align: center;">Less or equal</td></tr> <tr> <td style="text-align: center;"><b>selo</b></td><td style="text-align: center;"><math>111_2</math></td><td style="text-align: center;">Ordered</td></tr> </tbody> </table>			Instruction	Mask	Condition	<b>selno</b>	$000_2$	Unordered	<b>selg5</b>	$001_2$	Greater	<b>sele</b>	$010_2$	Equal	<b>selge</b>	$011_2$	Greater or equal	<b>sell</b>	$100_2$	Less	<b>selne</b>	$101_2$	Not equal	<b>selle</b>	$110_2$	Less or equal	<b>selo</b>	$111_2$	Ordered
Instruction	Mask	Condition																											
<b>selno</b>	$000_2$	Unordered																											
<b>selg5</b>	$001_2$	Greater																											
<b>sele</b>	$010_2$	Equal																											
<b>selge</b>	$011_2$	Greater or equal																											
<b>sell</b>	$100_2$	Less																											
<b>selne</b>	$101_2$	Not equal																											
<b>selle</b>	$110_2$	Less or equal																											
<b>selo</b>	$111_2$	Ordered																											
Action	<pre>if ((mask &amp; AC.cc)    (mask == AC.cc))     dst = src2; else     dst = src1;</pre>																												
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a>																											
Example	<pre>          # AC.cc = 010<sub>2</sub> sel e g0,g1,g2      # g2 = g1            # AC.cc = 001<sub>2</sub> sell g0,g1,g2      # g2 = g0</pre>																												

Opcode	<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	<b>MOVE, TEST&lt;cc&gt;, cmpi, cmpo, SUB&lt;cc&gt;</b>		
Notes	These core instructions are not implemented on 80960Cx, Kx and Sx processors.		

## 6.2.59 setbit

Mnemonic	<b>setbit</b>	Set Bit
Format	<b>setbit</b>	<i>bitpos</i> , <i>src</i> , <i>dst</i> reg/lit/sfr        reg/lit/sfr        reg/sfr
Description	Copies <i>src</i> value to <i>dst</i> with one bit set. <i>bitpos</i> specifies bit to be set.	
Action	<i>dst</i> = <i>src</i>   (2**( <i>bitpos</i> %32));	
Faults	STANDARD                          Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .	
Example	setbit 15, r9, r1 # r1 = r9 with bit 15 set.	
Opcode	<b>setbit</b>	583H                          REG
See Also	<a href="#">alterbit</a> , <a href="#">chkbit</a> , <a href="#">clrbit</a> , <a href="#">notbit</a>	

## 6.2.60 SHIFT

Mnemonic	<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>shrdi</b>	Shift Right Dividing Integer		
Format	<b>sh*</b>	<i>len,</i> reg/lit/sfr	<i>src,</i> reg/lit/sfr	<i>dst</i> reg/sfr
Description	<p>Shifts <i>src</i> left or right by the number of bits indicated with the <i>len</i> operand and stores the result in <i>dst</i>. Bits shifted beyond register boundary are discarded. For values of <i>len</i> &gt; 32, the processor interprets the value as 32.</p> <p><b>shlo</b> shifts zeros in from the least significant bit; <b>shro</b> shifts zeros in from the most significant bit. These instructions are equivalent to <b>mulo</b> and <b>divo</b> by the power of 2, respectively.</p> <p><b>shli</b> 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, <i>dst</i> will equal <i>src</i> shifted left as much as possible without overflowing.</p> <p><b>shri</b> 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).</p> <p><b>shrdi</b> 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 <i>src</i> operand was negative, which produces the correct result for negative operands.</p> <p><b>shli</b> and <b>shrdi</b> are equivalent to <b>muli</b> and <b>divi</b> by the power of 2.</p>			

Action	<b>shlo:</b> if(src1 < 32) dst = src * (2**len); else dst = 0; <b>shro:</b> if(src1 < 32) dst = src / (2**len); else dst = 0; <b>shli:</b> 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); }  <b>shri:</b> 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;  <b>shrdi:</b> dst = src / (2**len);
Faults	STANDARD Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5.</a> ARITHMETIC.OVERFLOW For <b>shli</b> .
Example	shli 13, g4, r6 # g6 = g4 shifted left 13 bits.
Opcode	<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

Mnemonic	<b>spanbit</b> Span Over Bit		
Format	<b>spanbit</b>	<i>src</i> , reg/lit/sfr	<i>dst</i> reg/sfr
Description	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> .		
Action	<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>		
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .	
Example	<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>		
Opcode	<b>spanbit</b>	640H	REG
See Also	<b>scanbit</b>		
Side Effects	Sets the condition code in the arithmetic controls.		

## 6.2.62 STORE

Mnemonic	<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
Format	<b>st*</b>	<i>src1,</i> reg <i>dst</i> mem
Description	<p>Copies a byte or group of bytes from a register or group of registers to memory. <i>src</i> specifies a register or the first (lowest numbered) register of successive registers.</p> <p><i>dst</i> 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 <i>dst</i>. Refer to <a href="#">Section 2.3, “Memory Addressing Modes” on page 2-6</a> for a complete discussion.</p> <p><b>stob</b> and <b>stib</b> store a byte and <b>stos</b> and <b>stis</b> store a half word from the <i>src</i> 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.</p> <p><b>st, stl, stt</b> and <b>stq</b> copy 4, 8, 12 and 16 bytes, respectively, from successive registers to memory.</p> <p>For <b>stl</b>, <i>src</i> must specify an even numbered register (e.g., g0, g2, ... or r0, r2, ...). For <b>stt</b> and <b>stq</b>, <i>src</i> must specify a register number that is a multiple of four (e.g., g0, g4, g8, ... or r0, r4, r8, ...).</p>	
Action	<pre> <b>st:</b> if (illegal_write_to_on_chip_RAM)     generate_fault(TYPE.MISMATCH); else if ((effective_address[1:0] != 00<sub>2</sub>) &amp;&amp; unaligned_fault_enabled)     { store_to_memory(effective_address)[31:0] = src1;       generate_fault(OPERATION.UNALIGNED); } else     store_to_memory(effective_address)[31:0] = src1; </pre>	

Action  
(continued)

```
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] != 0xFFFF))
{
    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];
```

Action (continued)	<b>stis:</b> if (illegal_write_to_on_chip_RAM_or_MMR) generate_fault(TYPE.MISMATCH); else if ((effective_address[0] != 0 <sub>2</sub> ) && 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];  <b>stl:</b> 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] != 000 <sub>2</sub> ) && 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; }
-----------------------	--

Action (continued)	<b>stt:</b> if (illegal_write_to_on_chip_RAM_or_MMR) generate_fault(TYPE.MISMATCH); else if (reg_number(src1) % 4 != 0) generate_fault(OPERATION.INVALID_OPERAND); else if ((effective_address[3:0] != 0000 <sub>2</sub> ) && 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; }  <b>stq:</b> if (illegal_write_to_on_chip_RAM_or_MMR) generate_fault(TYPE.MISMATCH); else if (reg_number(src1) % 4 != 0) generate_fault(OPERATION.INVALID_OPERAND); else if ((effective_address[3:0] != 0000 <sub>2</sub> ) && 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 Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5.</a> ARITHMETIC.OVERFLOW For <b>stib</b> , <b>stis</b> .																								
Example	st g2, 1254 (g6) # Word beginning at offset # 1254 + (g6) = g2.																								
Opcode	<table> <tr> <td><b>st</b></td><td>92H</td><td>MEM</td></tr> <tr> <td><b>stob</b></td><td>82H</td><td>MEM</td></tr> <tr> <td><b>stos</b></td><td>8AH</td><td>MEM</td></tr> <tr> <td><b>stib</b></td><td>C2H</td><td>MEM</td></tr> <tr> <td><b>stis</b></td><td>CAH</td><td>MEM</td></tr> <tr> <td><b>stl</b></td><td>9AH</td><td>MEM</td></tr> <tr> <td><b>stt</b></td><td>A2H</td><td>MEM</td></tr> <tr> <td><b>stq</b></td><td>B2H</td><td>MEM</td></tr> </table>	<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
<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	<b>LOAD, MOVE</b>																								

Notes      illegal\_write\_to\_on\_chip\_RAM is an implementation-dependent mechanism.  
The mapping of register bits to memory(*efa*) depends on the endianism of the  
memory region and is implementation-dependent.

## 6.2.63 subc

Mnemonic	<b>subc</b>	Subtract Ordinal With Carry
Format	<b>subc</b> <i>src1</i> , <i>src2</i> , <i>dst</i> reg/lit/sfr      reg/lit/sfr      reg/sfr	
Description	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.	
This instruction can also be used for integer subtraction. Here, if integer subtraction results in an overflow, condition code bit 0 is set.		
Action	$\begin{aligned} \text{dst} &= (\text{src2} - \text{src1} - 1 + \text{AC.cc}[1])[31:0]; \\ \text{AC.cc}[2:0] &= 000_2; \\ \text{if}(\text{src2}[31] == \text{src1}[31]) \&\& (\text{src2}[31] != \text{dst}[31]) \\ &\quad \text{AC.cc}[0] = 1; \quad \# \text{Overflow bit.} \\ &\quad \text{AC.cc}[1] = (\text{src2} - \text{src1} - 1 + \text{AC.cc}[1])[32]; \quad \# \text{Carry out.} \end{aligned}$	
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .
Example	<pre>subc g5, g6, g7 # g7 = g6 - g5 - not(condition code bit 1)</pre>	
Opcode	<b>subc</b>	5B2H      REG
See Also	<b>addc, addi, addo, subi, subo</b>	
Side Effects	Sets the condition code in the arithmetic controls.	

## 6.2.64 SUB<cc>

Mnemonic	<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
Format	<b>sub*</b>	<i>src1</i> , <i>src2</i> , <i>dst</i> reg/lit/sfr      reg/lit/sfr      reg/sfr
Description	Subtracts <i>src1</i> from <i>src2</i> 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 <i>src1</i> is subtracted from <i>src2</i> and the result stored in the destination	

Instruction	Mask	Condition
subono, subino	000 <sub>2</sub>	Unordered
subog, subig	001 <sub>2</sub>	Greater
suboe, subie	010 <sub>2</sub>	Equal
suboge, subige	011 <sub>2</sub>	Greater or equal
subol, subil	100 <sub>2</sub>	Less
subone, subine	101 <sub>2</sub>	Not equal
subole, subile	110 <sub>2</sub>	Less or equal
suboo, subio	111 <sub>2</sub>	Ordered

Action	<b>SUBO&lt;cc&gt;:</b> if ((mask & AC.cc)    (mask == AC.cc)) dst = (src2 - src1)[31:0];																																																
	<b>SUBI&lt;cc&gt;:</b> if ((mask & AC.cc)    (mask == AC.cc)) {																																																
	{ true_result = (src2 - src1); dst = true_result[31:0]; }																																																
	if((true_result > (2**31) - 1)    (true_result < -2**31)) # Check for overflow { if (AC.om == 1) AC.of = 1; else generate_fault (ARITHMETIC.OVERFLOW); } }																																																
Faults	STANDARD Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5.</a> ARITHMETIC.OVERFLOW For the <b>SUBI&lt;cc&gt;</b> class.																																																
Example	<pre> suboge g0,g1,g2      # AC.cc = 010<sub>2</sub>                       # g2 = g1 - g0  subile g0,g1,g2      # AC.cc = 001<sub>2</sub>                       # g2 not modified </pre>																																																
Opcode	<table> <tbody> <tr><td><b>subono</b></td><td>782H</td><td>REG</td></tr> <tr><td><b>subog</b></td><td>792H</td><td>REG</td></tr> <tr><td><b>suboe</b></td><td>7A2H</td><td>REG</td></tr> <tr><td><b>suboge</b></td><td>7B2H</td><td>REG</td></tr> <tr><td><b>subol</b></td><td>7C2H</td><td>REG</td></tr> <tr><td><b>subone</b></td><td>7D2H</td><td>REG</td></tr> <tr><td><b>subole</b></td><td>7E2H</td><td>REG</td></tr> <tr><td><b>suboo</b></td><td>7F2H</td><td>REG</td></tr> <tr><td><b>subino</b></td><td>783H</td><td>REG</td></tr> <tr><td><b>subig</b></td><td>793H</td><td>REG</td></tr> <tr><td><b>subie</b></td><td>7A3H</td><td>REG</td></tr> <tr><td><b>subige</b></td><td>7B3H</td><td>REG</td></tr> <tr><td><b>subil</b></td><td>7C3H</td><td>REG</td></tr> <tr><td><b>subine</b></td><td>7D3H</td><td>REG</td></tr> <tr><td><b>subile</b></td><td>7E3H</td><td>REG</td></tr> <tr><td><b>subio</b></td><td>7F3H</td><td>REG</td></tr> </tbody> </table>	<b>subono</b>	782H	REG	<b>subog</b>	792H	REG	<b>suboe</b>	7A2H	REG	<b>suboge</b>	7B2H	REG	<b>subol</b>	7C2H	REG	<b>subone</b>	7D2H	REG	<b>subole</b>	7E2H	REG	<b>suboo</b>	7F2H	REG	<b>subino</b>	783H	REG	<b>subig</b>	793H	REG	<b>subie</b>	7A3H	REG	<b>subige</b>	7B3H	REG	<b>subil</b>	7C3H	REG	<b>subine</b>	7D3H	REG	<b>subile</b>	7E3H	REG	<b>subio</b>	7F3H	REG
<b>subono</b>	782H	REG																																															
<b>subog</b>	792H	REG																																															
<b>suboe</b>	7A2H	REG																																															
<b>suboge</b>	7B2H	REG																																															
<b>subol</b>	7C2H	REG																																															
<b>subone</b>	7D2H	REG																																															
<b>subole</b>	7E2H	REG																																															
<b>suboo</b>	7F2H	REG																																															
<b>subino</b>	783H	REG																																															
<b>subig</b>	793H	REG																																															
<b>subie</b>	7A3H	REG																																															
<b>subige</b>	7B3H	REG																																															
<b>subil</b>	7C3H	REG																																															
<b>subine</b>	7D3H	REG																																															
<b>subile</b>	7E3H	REG																																															
<b>subio</b>	7F3H	REG																																															
See Also	<b>subc, subi, subo, SEL&lt;cc&gt;, TEST&lt;cc&gt;</b>																																																
Notes	These core instructions are not implemented on 80960Cx, Kx and Sx processors.																																																

## 6.2.65 subi, subo

Mnemonic	<b>subi</b>	Subtract Integer
	<b>subo</b>	Subtract Ordinal
Format	<b>sub*</b>	<i>src1,</i> reg/lit/sfr <i>src2,</i> reg/lit/sfr <i>dst</i> reg/sfr
Description	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.	
Action	<b>subo:</b> $dst = (src2 - src1)[31:0];$  <b>subi:</b> $true\_result = (src2 - src1);$ $dst = true\_result[31:0];$ $if((true\_result > (2^{**}31) - 1) \parallel (true\_result < -2^{**}31)) \# Check for overflow$ { $if(AC.om == 1)$ $AC.of = 1;$ $else$ $generate\_fault(ARITHMETIC.OVERFLOW);$ }	
Faults	STANDARD For <b>subi</b> .	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a>
Example	<code>subi g6, g9, g12 # g12 = g9 - g6</code>	
Opcode	<b>subi</b> 593H	REG
	<b>subo</b> 592H	REG
See Also	<b>addi, addo, subc, addc</b>	

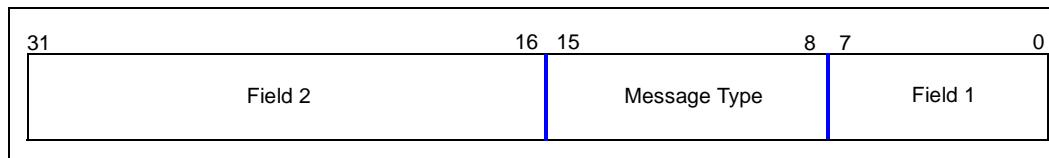
## 6.2.66 syncf

Mnemonic	<b>syncf</b> Synchronize Faults		
Format	<b>syncf</b>		
Description	Waits for all faults to be generated that are associated with any prior uncompleted instructions.		
Action	<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>		
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5</a> .	
Example	<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>		
Opcode	<b>syncf</b>	66FH	REG
See Also	<a href="#">mark</a> , <a href="#">fmark</a>		

## 6.2.67 sysctl

Mnemonic	<b>sysctl</b> System Control			
Format	<b>sysctl</b>	<i>src1</i> , reg/lit/sfr	<i>src2</i> , reg/lit/sfr	<i>src/dst</i> reg/sfr
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 <i>src1</i> is executed. The type field of <i>src1</i> is interpreted depending upon the command. Remaining <i>src1</i> bits are reserved. The <i>src2</i> and <i>src3</i> operands are also interpreted depending upon the command.			

**Figure 6-7. Src1 Operand Interpretation**



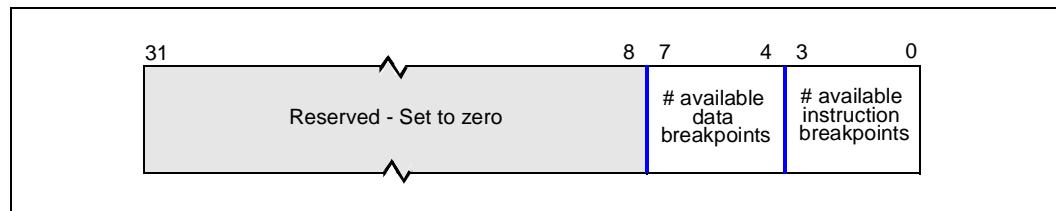
**Table 6-10. sysctl Field Definitions**

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 (See Table 6-11)	N/U	Cache load address	N/U
Reinitialize	0x3	N/U	N/U	Starting IP	PRCB Pointer
Load Control Register	0x4	Register Group Number	N/U	N/U	N/U
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 (See 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	80960HA/HD/HT
$000_2$	Normal cache enabled	16 Kbyte
$XX1_2$	Full cache disabled	16 Kbyte
$100_2$ or $110_2$	Load and lock cache	Lock 1 way (4K) 12K, 3 ways available

**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 i960 Hx processor.
        # src2 has starting address of code to lock.
        # src2 is aligned to a quad word
        # boundary.
        aligned_addr = src2 & 0xffffffff0;
        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;
}

```

	<pre> 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 4: # Load One Group of Control Registers From Control Table     grpoff = (src1 &amp; 0xff) * 16;     for (i = 0; i &lt; 4; i=i+4)         memory[control_reg_addr(i.grpoff)] = memory[i+grpoff];     }     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 &lt;&lt; 16)   (src1 &gt;&gt; 16);     temp = memory[addr];     memory[addr] = (src2 &amp; dst)   (temp &amp; ~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); </pre>	
Faults	STANDARD	Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5.</a>
Example		<pre> 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. </pre>
Opcode	<b>sysctl</b>	659H REG
See Also	<b>dcctl, icctl</b>	
Notes	This instruction is implemented on 80960RP, Hx, Jx and Cx processors, and may or may not be implemented on future i960 processors.	

## 6.2.68 TEST<cc>

Mnemonic	<b>teste{.t .f}</b> <b>testne{.t .f}</b> <b>testl{.t .f}</b> <b>testle{.t .f}</b> <b>testg{.t .f}</b> <b>testge{.t .f}</b> <b>testo{.t .f}</b> <b>testno{.t .f}</b>	Test For Equal Test For Not Equal Test For Less Test For Less Or Equal Test For Greater Test For Greater Or Equal Test For Ordered Test For Not Ordered
Format	<b>test*{.t .f}</b>	<i>dst:src1</i> reg
Description		Stores a true (01H) in <i>dst</i> if the logical AND of the condition code and opcode mask part is not zero. Otherwise, the instruction stores a false (00H) in <i>dst</i> . For <b>testno</b> (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

The optional .t or .f suffix may be appended to the mnemonic. Use .t to speed up execution when these instructions usually store a true (1) condition in *dst*. Use .f to speed up execution when these instructions usually store a false (0) condition in *dst*. If a suffix is not provided, the assembler is free to provide one.

Action	For all <b>TEST&lt;cc&gt;</b> except <b>testno</b> : if((mask & AC.cc) != 000 <sub>2</sub> ) src1 = 1;                          #true value else src1 = 0;                          #false value		
	<b>testno:</b> if(AC.cc == 000 <sub>2</sub> ) src1 = 1;                          #true value else src1 = 0;                          #false value		
Faults	STANDARD Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5.</a>		
Example	# Assume AC.cc = 100 <sub>2</sub> testl g9# g9 = 0x00000001		
Opcode	<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	<b>cmpi, cmpdeci, cmpinci</b>		

## 6.2.69    **xnor, xor**

Mnemonic	<b>xnor</b>	Exclusive Nor	
	<b>xor</b>	Exclusive Or	
Format	<b>xnor</b>	<i>src1,</i> reg/lit/sfr	<i>src2,</i> reg/lit/sfr
	<b>xor</b>	<i>src1,</i> reg/lit/sfr	<i>dst</i> reg/sfr
Description		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> .	
Action	<b>xnor:</b>	$dst = \sim(src2   src1)   (src2 \& src1);$	
	<b>xor:</b>	$dst = (src2   src1) \& \sim(src2 \& src1);$	
Faults	STANDARD		Refer to <a href="#">Section 6.1.6, “Faults” on page 6-5.</a>
Example	<b>xnor r3, r9, r12</b>	# r12 = r9 XNOR r3	
	<b>xor g1, g7, g4</b>	# g4 = g7 XOR g1	
Opcode	<b>xnor</b>	589H	REG
	<b>xor</b>	586H	REG
See Also	<b>and, andnot, nand, nor, not, notand, notor, or, ornot</b>		

## Procedure Calls

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® 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 <i>local stack</i> . 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 <i>supervisor stack</i> .                              |

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

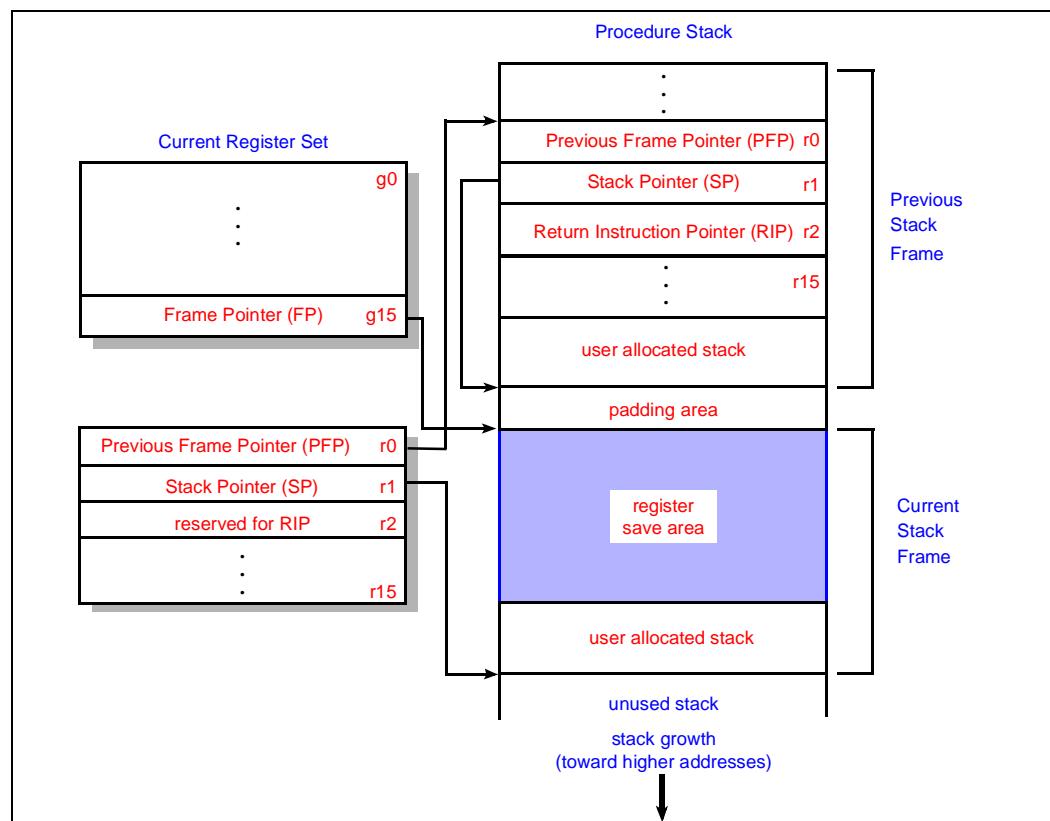
The processor automatically allocates a set of 16 local registers for each procedure. Since local registers are on-chip, they provide fast access storage for local variables. Of the 16 local registers, 13 are available for general use; r0, r1 and r2 are reserved for linkage information to tie procedures 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 (see [Section A.2.5, “Data and Data Structure Alignment” on page A-3](#)). In the i960 Hx processor, stacks are aligned on 16-byte boundaries (see [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
1d     20(r4),rn
...
1d     (r4),rn
mov    r4,sp
```

#### 7.1.2.5 Previous Frame Pointer

The previous frame pointer is the previous stack frame's first byte address. This address's upper 28 bits are stored in local register r0, the previous frame pointer (PFP) register. The four least-significant bits of the PFP are used to store the return type field. See [Figure 7-5](#) and [Table 7-2](#) for more information on the PFP and the return-type field.

#### 7.1.2.6 Return Type Field

PFP register bits 0 through 3 contain return type information for the calling procedure. When a procedure call is made — either explicit or implicit — the processor records the call type in the return type field. The processor then uses this information to select the proper return mechanism when returning to the calling procedure. The use of this information is described in [Section 7.8, “Returns” on page 7-19](#).

#### 7.1.2.7 Return Instruction Pointer

The actual RIP register (r2) is reserved by the processor to support the call and return mechanism and must not be used by software; the actual value of RIP is unpredictable at all times. For example, an implicit procedure call (fault or interrupt) can occur at any time and modify the RIP. An OPERATION.INVALID\_OPERAND fault is generated when attempting to write to 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. 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.
3. The frame pointer (g15) for the calling procedure is stored in the current stack's PFP register (r0). The return type field in the PFP register is set according to the call type which is performed. See [Section 7.8, “Returns” on page 7-19](#).
4. For a local or system-local call, a new stack frame is allocated by using the old stack pointer value saved in step 2. This value is first rounded to the next 16-byte boundary to create a new frame pointer, then stored in the FP register. Next, 64 bytes are added to create the new frame's register save area. This value is stored in the SP register.

For an interrupt call from user mode, the current interrupt stack pointer value is used instead of the value saved in step 2.

For a system-supervisor call from user mode, the current Supervisor Stack Pointer (SSP) value is used instead of the value saved in step 2.

5. The instruction pointer is loaded with the address of the first instruction in the called procedure. The processor gets the new instruction pointer from the **call**, the system procedure table, the interrupt table or the fault table, depending on the type of call executed.

Upon completion of these steps, the processor begins executing the called procedure. Sometime before a return or nested call, the local register set is bound to the allocated stack frame.

### 7.1.3.2 Return Operation

A return from any call type — explicit or implicit — is always initiated with a return (**ret**) instruction. On a return, the processor performs these operations:

1. The current stack frame and local registers are deallocated by loading the FP register with the value of the PFP register.
2. The local registers for the return target procedure are retrieved. The registers are usually read from the local register cache; however, in some cases, these registers have been flushed from register cache to memory and must be read directly from the save area in the stack frame.
3. The processor sets the instruction pointer to the value of the RIP register.

Upon completion of these steps, the processor executes the instruction to which it returns. The frames created before the **ret** instruction was executed will be 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. 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 6.2.31, “flushreg” on page 6-53](#)) 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 will be 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 11 through 8 of the register cache configuration word in the PRCB (see [Figure 13-6 “Process Control Block Configuration Words” on page 13-18](#)). 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 15. Setting the value to 0 reserves no frames for high-priority interrupts. Setting the value to 15 causes the register cache to become disabled for non-critical code. See [Section 13.3.1.2, “Process Control Block \(PRCB\)” on page 13-17](#).

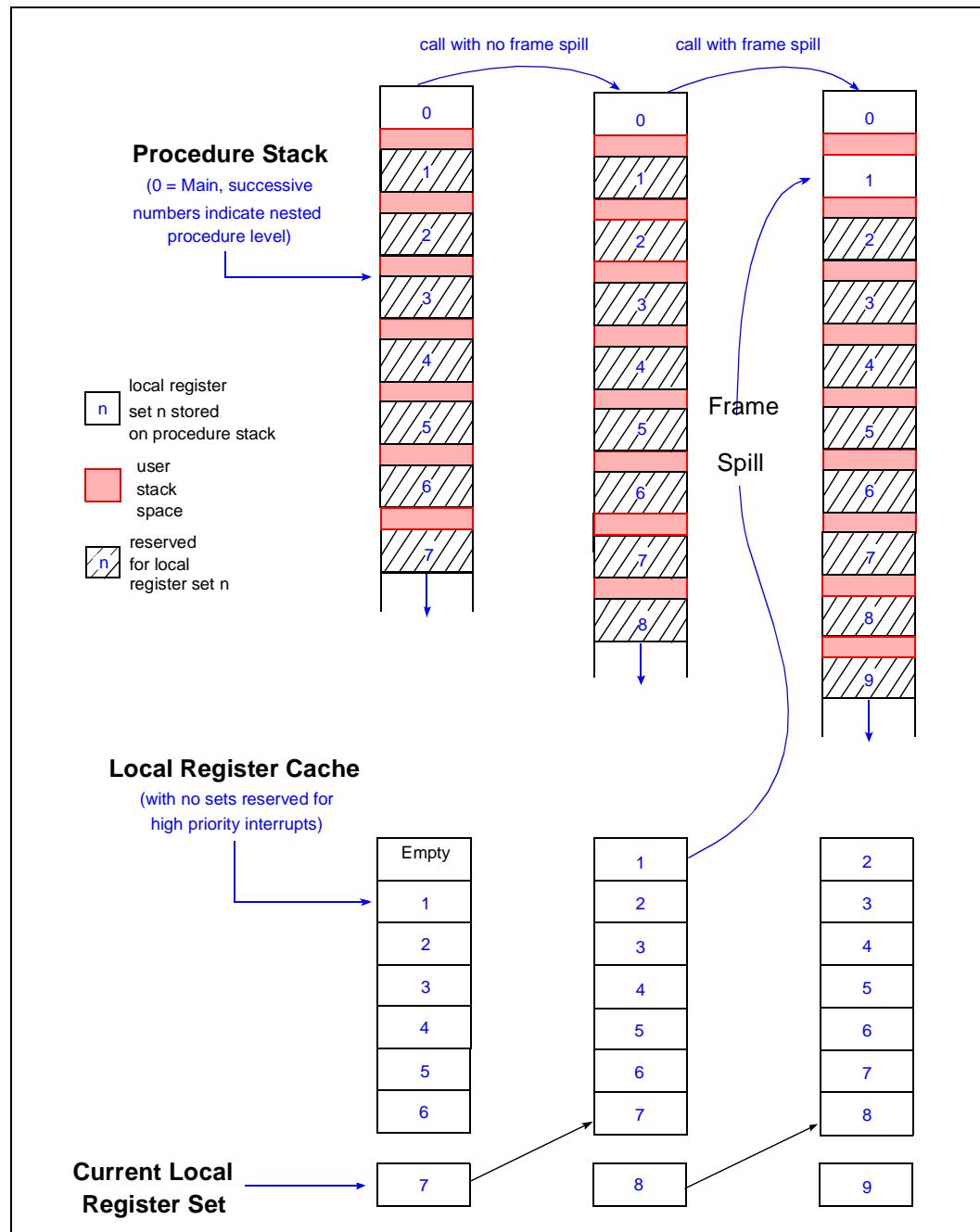
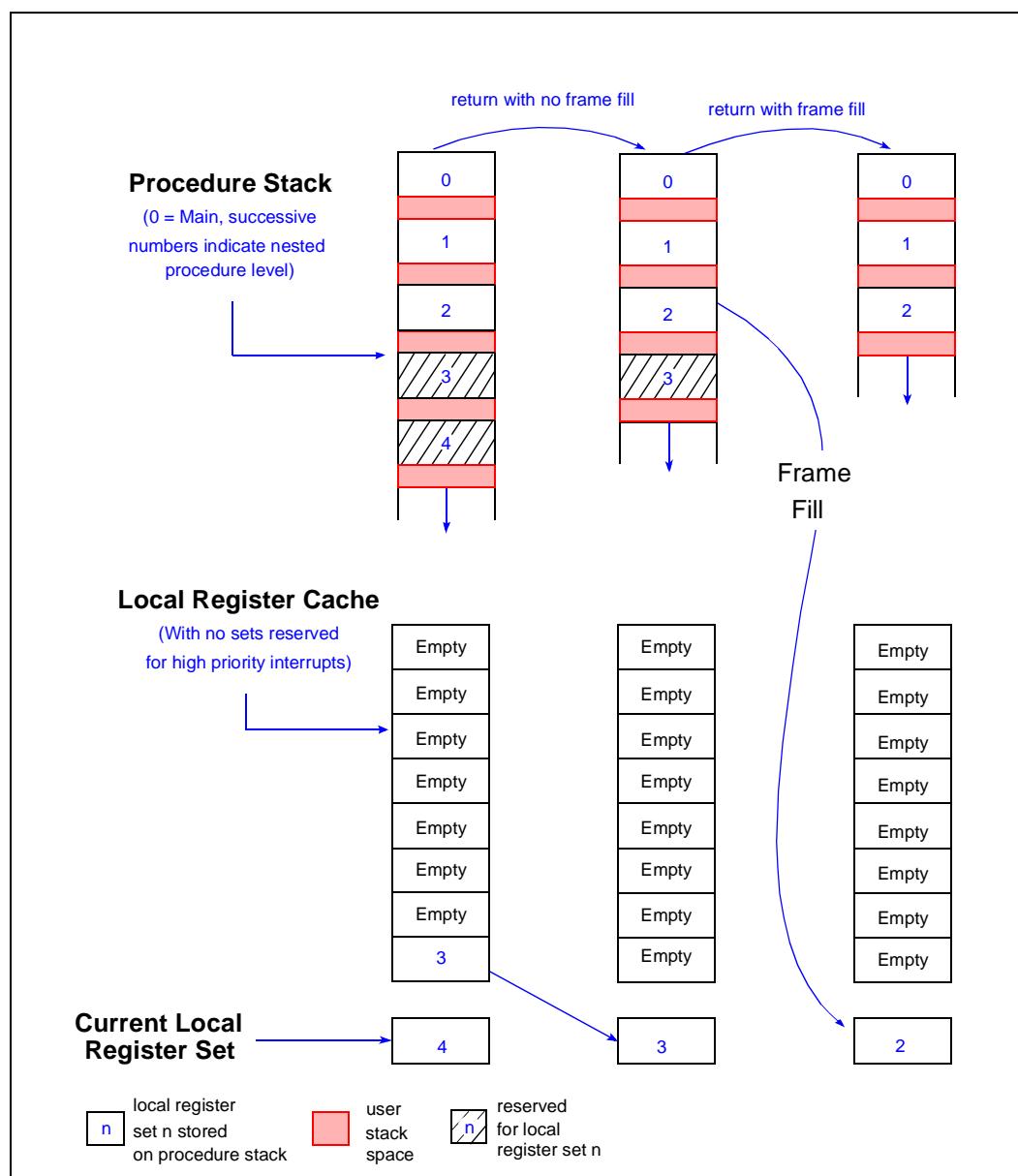
**Figure 7-2. Frame Spill**


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 will be 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 will return 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 (see [Example](#) ). 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 will execute very quickly because the local register cache has already been flushed by the **flushreg** before; only synchronization of the PFP will be performed. i960 processor implementations may provide other mechanisms to ensure PFP synchronization in addition to **flushreg**, but a **flushreg** after a PFP modification is ensured to work on all i960 processors.

## 7.3 Parameter Passing

Parameters are passed between procedures in two ways:

value	Parameters are passed directly to the calling procedure as part of the call and return mechanism. This is the fastest method of passing parameters.
reference	Parameters are stored in an argument list in memory and a pointer to the argument list is passed in a global register.

When passing parameters by value, the calling procedure stores the parameters to be passed in global registers. Since the calling procedure and the called procedure share the global registers, the called procedure has direct access to the parameters after the call.

When a procedure needs to pass more parameters than will fit 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 will be 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 7-15](#).

**call** specifies the address of the called procedures as the IP plus a signed, 24-bit displacement (i.e., - $2^{23}$  to  $2^{23} - 4$ ). **callx** allows any of the addressing modes to be used to specify the procedure address. The IP-with-displacement addressing mode allows full 32-bit IP-relative addressing.

When a local call is made with a **call** or **callx**, the processor performs the same operation as described in [Section 7.1.3.1, “Call Operation” on page 7-6](#). The target IP for the call is derived from the instruction’s operands and the new stack frame is allocated on the current stack.

## 7.5 System Calls

A system call is a call made via the system procedure table. It can be used to make a system-local call — similar to a local call made with **call** and **callx** in the sense that there is no stack nor mode switch — or a system supervisor call. A system call is initiated with **calls**, which requires a procedure number operand. The procedure number provides an index into the system procedure table, where the processor finds IPs for specific procedures.

Using an i960 processor language assembler, a system procedure is directly declared using the .sysproc directive. At link time, the optimized call directive, callj, is replaced with a **calls** when a system procedure target is specified. (Refer to current i960 processor assembler documentation for a description of the .sysproc and callj directives.)

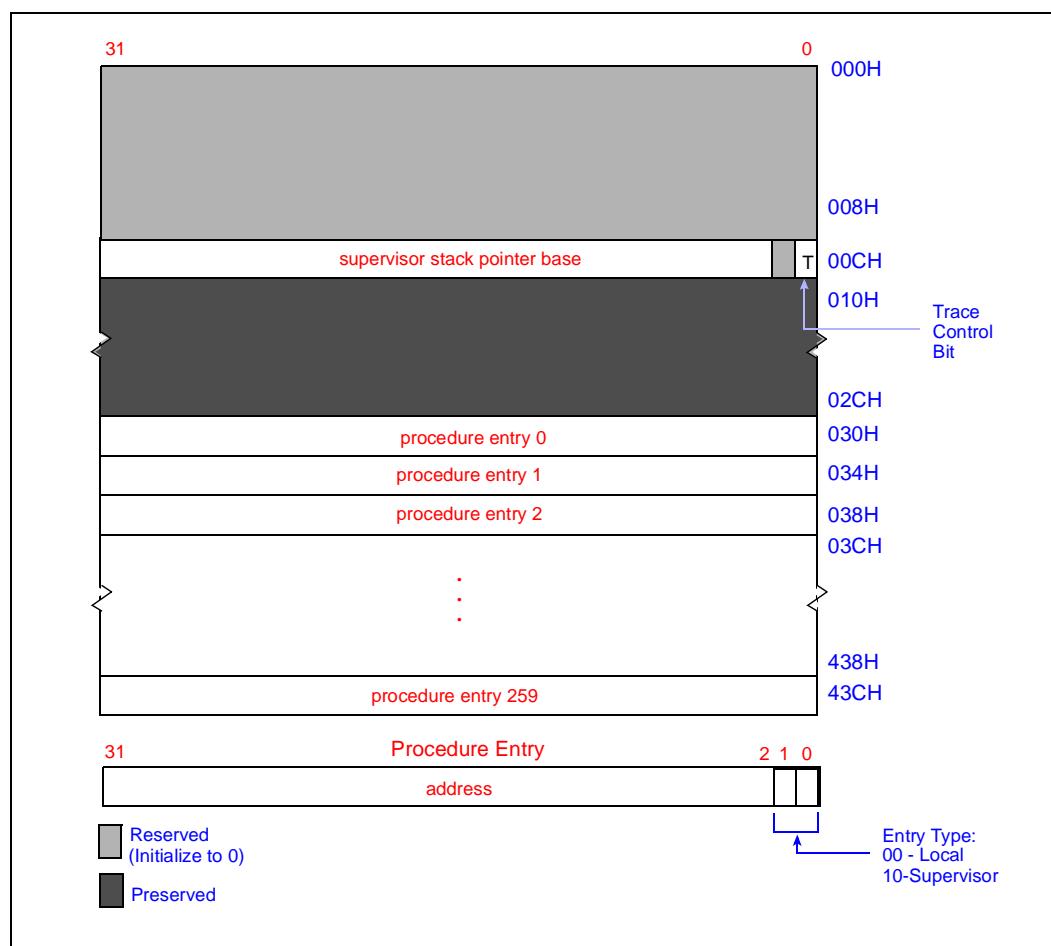
The system call mechanism offers two benefits. First, it supports application software portability. System calls are commonly used to call kernel services. By calling these services with a procedure number rather than a specific IP, applications software does not need to be changed each time the implementation of the kernel services is modified. Only the entries in the system procedure table must be changed. Second, the ability to switch to a different execution mode and stack with a system supervisor call allows kernel procedures and data to be insulated from applications code. This benefit is further described in [Section 3.7, “User-Supervisor Protection Model” on page 3-26](#).

### 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 8.1, “Fault Handling Overview” on page 8-1](#).

[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 9.1.2, “PC Trace Enable Bit and Trace-Fault-Pending Flag” on page 9-3.

## 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 6-25](#).

## 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 3-26](#).

## 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 13.2, “Initialization” on page 13-2](#). 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.

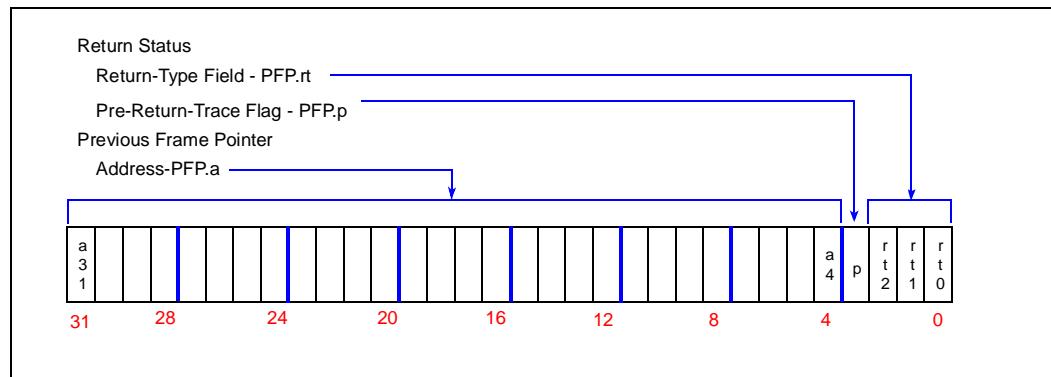
A call to a fault procedure is similar to a system call. Fault procedure calls can be local calls or supervisor calls. The processor obtains pointers to fault procedures through the fault table and (optionally) through the system procedure table.

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, “Faults”](#) and [Chapter 11, “Interrupts”](#) 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) (r0)**



*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 9.5.2.1, “Tracing on Explicit Call” on page 9-12](#).

*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 9.2, “Trace Modes” on page 9-3](#) 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.



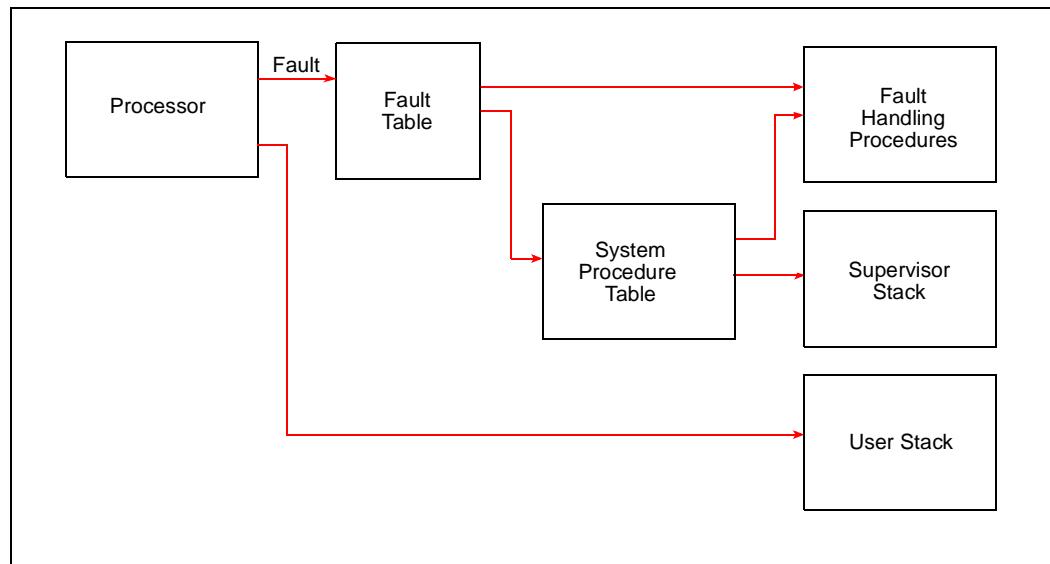
This chapter describes the i960® Hx processor's fault handling facilities. Subjects covered include the fault handling data structures and fault handling mechanisms. See [Section 8.10, “Fault Reference” on page 8-21](#) for detailed information on each fault type.

## 8.1 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 8-1](#), 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 8-1. 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. If the processor is in the interrupted state, the processor uses the interrupt stack.

Once these data structures and the code for the fault procedures are established in memory, the processor handles faults automatically and independently from application software.

The processor can detect a fault at any time while executing instructions, whether from a program, interrupt handling procedure or fault handling procedure. When a fault occurs, the processor determines the fault type and selects a corresponding fault handling procedure from the fault table. It then invokes the fault handling procedure by means of an implicit call. As described later in this chapter, the fault handler call can be:

- A local call (call-extended operation)
- A system-local call (local call through the system procedure table)
- A system-supervisor call (supervisor call through the system procedure table)

A normal fault condition is handled by the processor in the following manner:

- The current local registers are saved and cached on-chip.
- PFP = FP and the value 001 is written to the Return Type Field (Fault Call). Refer to [Section 7.8, “Returns” on page 7-19](#) for more information.
- If 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. If 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 (see [Section 8.10, “Fault Reference” on page 8-21](#), and the processor will resume execution at the RIP upon return from the fault handler. If 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

## 8.2 Fault Types

The i960 processor 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 Hx processor recognizes i960 processor architecture-defined faults and a new fault subtype for detecting unaligned memory accesses. [Table 8-1](#) lists all faults that the i960 Hx processor detects, arranged by type and subtype. Text that follows the table gives column definitions.

**Table 8-1. i960® Hx 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 8.6.4, "Parallel Faults" on page 8-9</a>
1H	TRACE	Bit 1	INSTRUCTION	0001 0002H
		Bit 2	BRANCH	0001 0004H
		Bit 3	CALL	0001 0008H
		Bit 4	RETURN	0001 0010H
		Bit 5	PRERETURN	0001 0020H
		Bit 6	SUPERVISOR	0001 0040H
		Bit 7	MARK	0001 0080H
2H	OPERATION	1H	INVALID_OPCODE	0002 0001H
		2H	UNIMPLEMENTED	0002 0002H
		3H	UNALIGNED	0002 0003H
		4H	INVALID_OPERAND	0002 0004H
3H	ARITHMETIC	1H	INTEGER_OVERFLOW	0003 0001H
		2H	ZERO-DIVIDE	0003 0002H
4H	Reserved			
5H	CONSTRAINT	1H	RANGE	0005 0001H
6H	Reserved			
7H	PROTECTION	Bit 1	LENGTH	0007 0002H
		Bit 5	BAD_ACCESS	0007 0020H
9H	Reserved			
8H	MACHINE	2H	PARITY_ERROR	0008 0002H
AH	TYPE	1H	MISMATCH	000A 0001H
BH - FH	Reserved			
10H	OVERRIDE	NA	NA	NA

In [Table 8-1](#):

- 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.

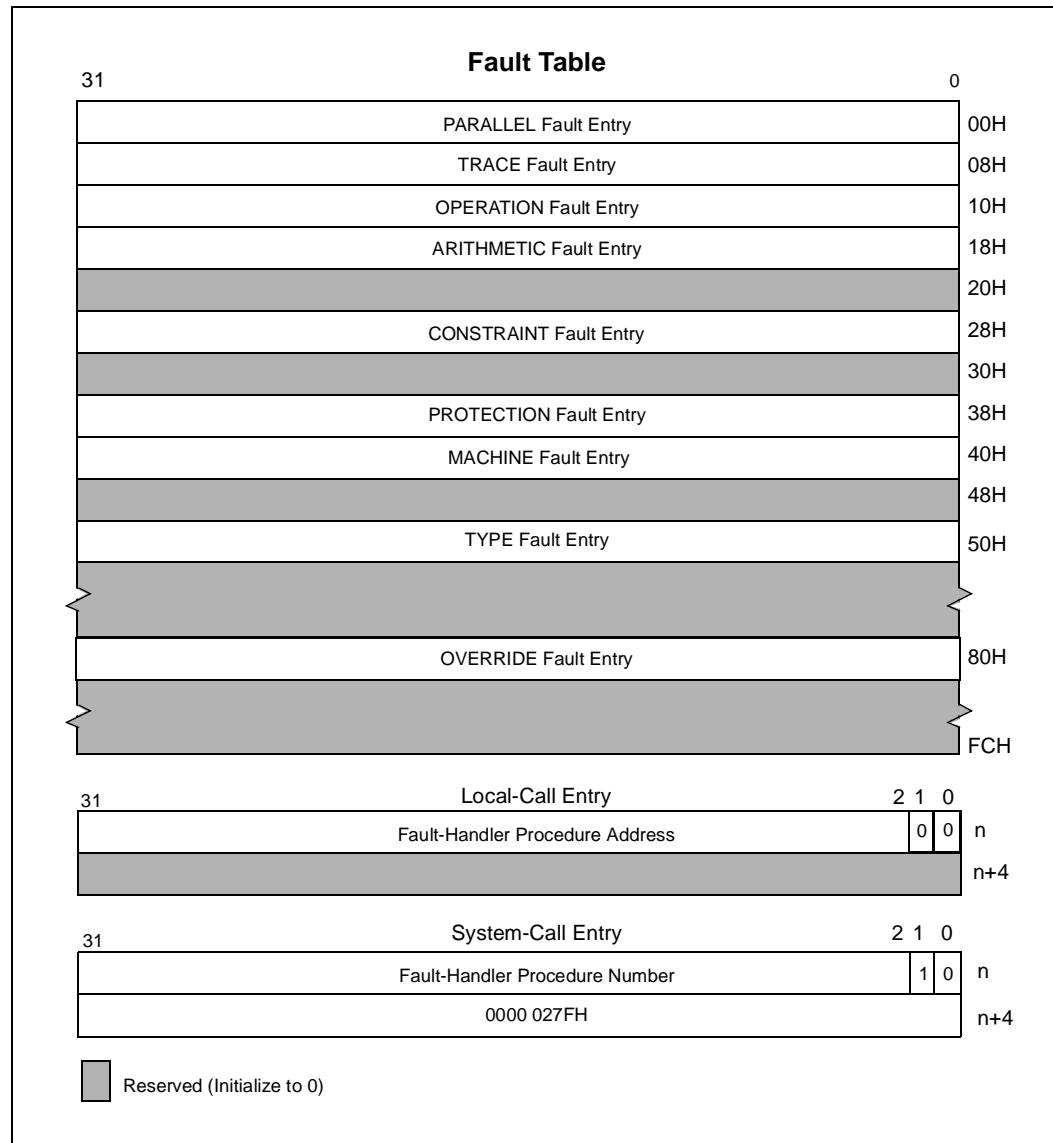
Other i960 processor family members may provide extensions that recognize additional fault conditions. Fault type and subtype encoding allows all faults to be included in the fault table: those that are common to all i960 processors and those that are specific to one or more family members. The fault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

## 8.3 Fault Table

The fault table (Figure 8-2) is the processor's pathway to the fault handling procedures. It can be located anywhere in the address space. From the control table, 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 8-2. Fault Table and Fault Table Entries**



As indicated in [Figure 8-2](#), 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 $00_2$ )	Provides an instruction pointer for the fault handling procedure. The processor uses this entry to invoke the specified procedure by means of an implicit local-call operation. The second word of a local procedure entry is reserved. It must be set to zero when the fault table is created and not accessed after that.
<i>system-call entry</i> (type $10_2$ )	Provides a procedure number in the system procedure table. This entry must have an entry type of $10_2$ and a value in the second word of 0000 027FH. The processor computes the system procedure number by shifting right the first word of the fault entry by two bit positions. Using this system procedure number, 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.

Other entry types ( $01_2$  and  $11_2$ ) are reserved and have unpredictable behavior.

To summarize, a fault handling procedure can be invoked through the fault table in any of three ways: a local call, a system-local call or a system-supervisor call.

## 8.4 Stack Used in Fault Handling

The i960 processor 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.

## 8.5 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 will use to handle the fault.

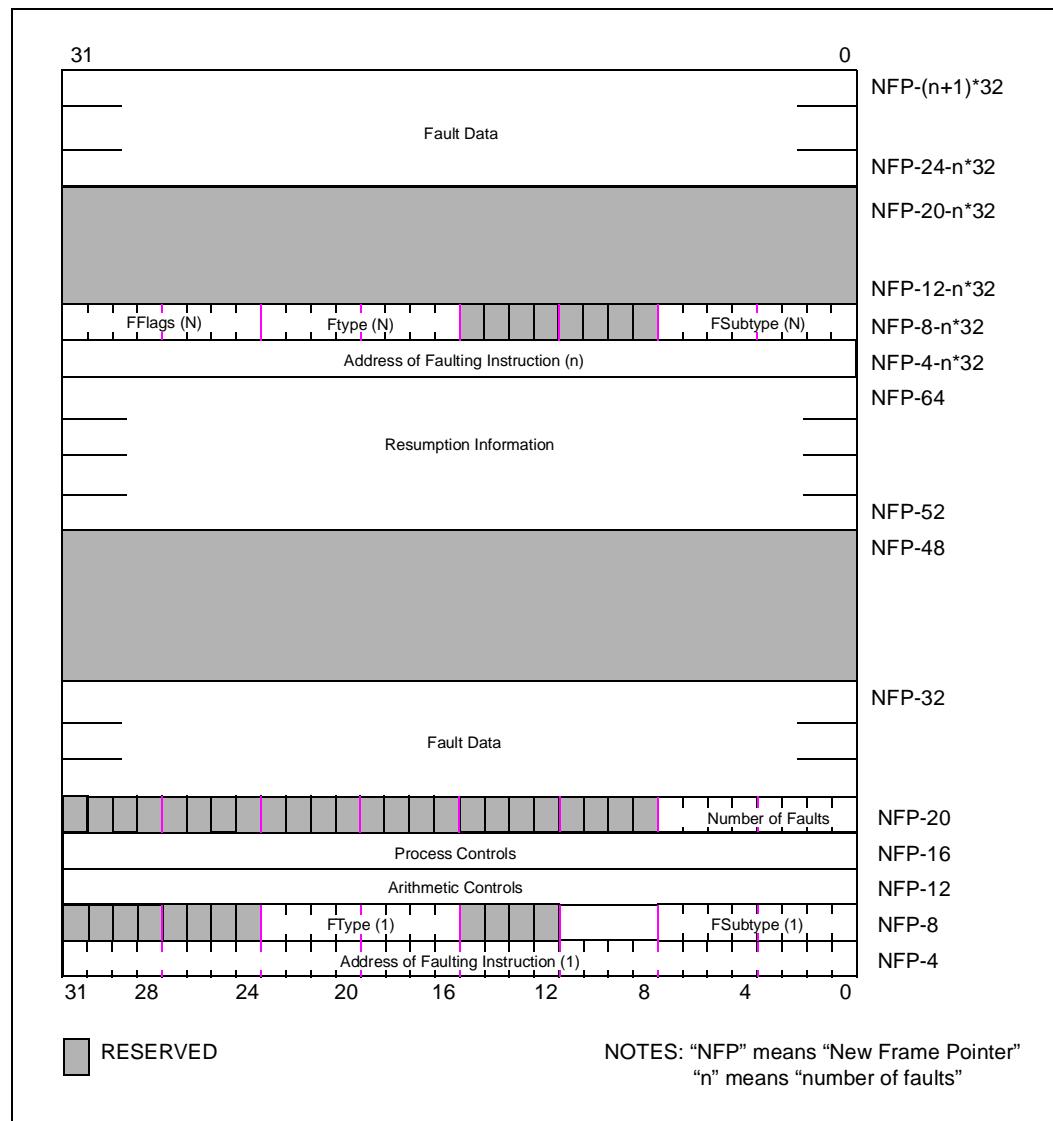
### 8.5.1 Fault Record Description

[Figure 8-3](#) 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.

In the i960 Hx processor, the Address of Faulting Instruction Field is undefined for GMU PROTECTION.BAD\_ACCESS faults and MACHINE.PARITY faults. Also the FAULT DATA field in the fault record contains information about the type of access and the address of the faulting access. It is only used for PROTECTION.BAD\_ACCESS, MACHINE.PARITY and OPERATION.UNALIGNED faults.

**Figure 8-3. Fault Record**



The processor can generate PARALLEL faults when instructions are executed in parallel. The Number of Faults field is used to describe PARALLEL faults. For single faults, the 80960Hx places the number of faults (one) in the OSubtype field, as it does for PARALLEL faults (greater than one).

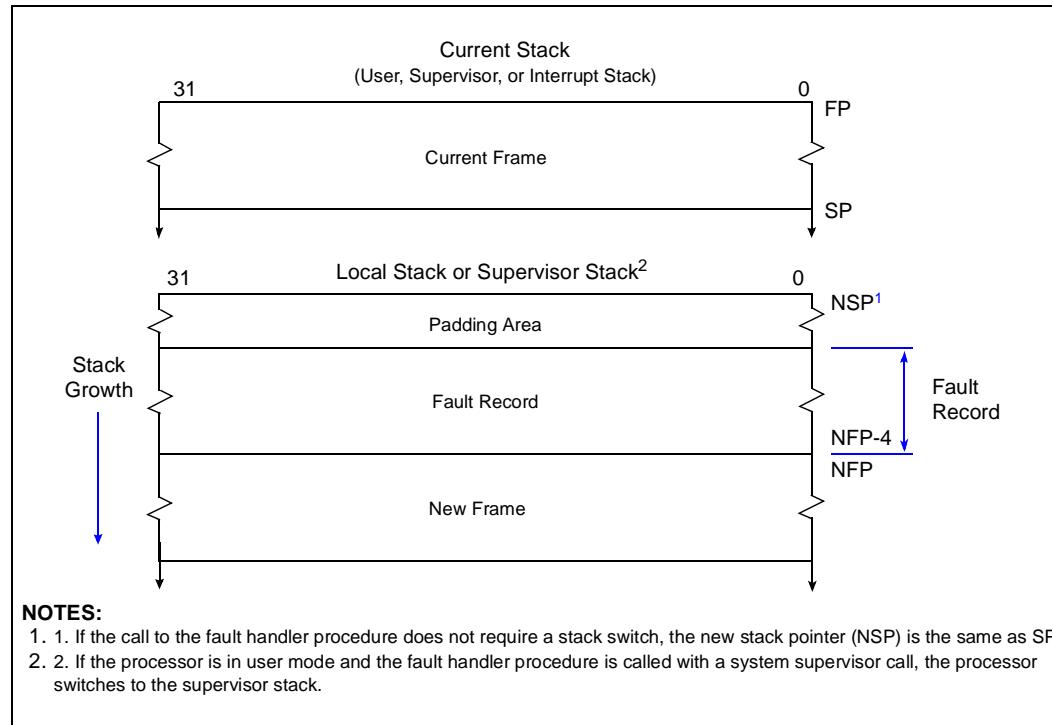
All unused bytes in the fault record are reserved. See [Section 8.6, “Multiple and Parallel Faults” on page 8-9](#) for more information.

The Resumption Field is used to store information about a pending trace fault. If 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.

## 8.5.2 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 8-4](#), 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.

**Figure 8-4. Storage of the Fault Record on the Stack**



## 8.6 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.

### 8.6.1 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 will report only one detected non-trace fault on a single instruction.

In a multiple fault situation, the reported fault condition is left to the implementation.

### 8.6.2 Multiple Trace Fault Conditions on the Same Instruction

Trace faults on different instructions cannot happen concurrently, because trace faults are precise (see [Section 8.9, “Precise and Imprecise Faults” on page 8-19](#)). 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 (see [Table 8-1](#)).

### 8.6.3 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 8.10, “Fault Reference” on page 8-21](#).
- 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.

### 8.6.4 Parallel Faults

The i960 Hx processor exploits the architecture’s tolerance of out-of-order instruction execution by issuing instructions to independent execution units on the chip. The following subsections describe how the processor handles faults in this environment.

#### 8.6.4.1

### Faults on Multiple Instructions Executed in Parallel

If 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 (see [Section 8.9.4, “No Imprecise Faults \(AC.nif\) Bit” on page 8-20](#)). 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.

If 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.

If the RIP is defined for all faults found in the fault record, then it will point to the next instruction not yet executed. The parallel fault handler can simply return to the next instruction not yet executed with a **ret** instruction.

Consider the following code example, where the **muli** and the **addi** instructions both have overflow conditions. AC.om=0, AC.nif = 0, and both instructions are in the instruction cache at the time of their execution. The **addi** and **muli** are allowed to execute in parallel because AC.nif = 0 and the faults that these instructions can generate (ARITHMETIC) are imprecise.

```
mulig2, g4, g6;
addig8, g9, g10;           # results in integer overflow
```

The fault on the **addi** is detected before the fault on the **muli** because the **muli** takes longer to execute. The fault call synchronizes faults on the way to the overflow fault handler for the **addi** instruction (see [Section 8.9.5, “Controlling Fault Precision” on page 8-20](#)), which is when the **muli** fault is detected. The processor builds a parallel fault record with information relative to both faults and calls the parallel fault handler. In the fault handler, ARITHMETIC faults may be recovered by storing the desired result of the instruction in the proper destination register and setting the AC.of flag (optional) to indicate that an overflow occurred. A **ret** at the end of the parallel fault handler routine will then return to the next instruction not yet executed in the program flow.

On the i960 Hx processor, the **muli** overflow, REG side, MEM side (see [Appendix E, “Instruction Execution and Performance Optimization”](#)) and parity faults are the only faults that can happen in parallel with any other defined 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.

#### 8.6.4.2

### 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 8.5.1, “Fault Record Description” on page 8-6](#). The remaining parallel faults are written to the fault record’s optional section, and the fault handling procedure for parallel faults is invoked. [Figure 8-3](#) shows the structure of the fault record for parallel faults.

The Number of Faults 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, if two parallel faults occur, the fault record for the second fault is located from NFP-96 to NFP-65.)

To calculate byte offsets, “n” indicates the fault number. Thus, for the second fault recorded (n=2), the relationship (NFP-8-(n \* 32)) reduces to NFP-72. On the i960 Hx processor, a parity fault can happen in parallel with any other fault.

## 8.6.5 Override Faults

The i960 Hx processor can detect a fault condition while the processor is preparing to service a previously detected fault. When this occurs, it is called an *override condition*. This section describes this condition and how the processor handles it.

A normal fault condition is handled by the processor in the following manner:

- The current local registers are saved and cached on-chip.
- PFP = FP and the value 001 is written to the Return Type Field (Fault Call). Refer to [Section 7.8, “Returns” on page 7-19](#) for more information.
- If 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 as a fault record in the new local register set as shown in [Table 8-2](#):

**Table 8-2. Override Fault Record Format**

Register	Contents
r3	IP within 8 bytes of the original faulting instruction.
r4	Override fault indicator for PROTECTION.BAD_ACCESS fault conditions. This field is unused for other fault conditions.
r5	Override Access type word for PROTECTION.BAD_ACCESS or MACHINE.PARITY fault conditions. This field is unused for other fault conditions.
r6	Override Fault address for PROTECTION.BAD_ACCESS or MACHINE.PARITY fault conditions. This field is unused for other fault conditions.
r7	Override Fault Type and Subtype.

- 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 16 decimal or 10h. If 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, MACHINE and OPERATION.UNIMPLEMENTED faults.

## 8.6.6 System Error

If a fault is detected while the processor is in the process of servicing an override 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 13.2.2.5, “Self Test Failure# Codes” on page 13-9](#).

## 8.7 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.

### 8.7.1 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.

If 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.

## 8.7.2 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)

### 8.7.2.1 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
- PROTECTION.BAD\_ACCESS due to protect violation

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. Because the state of the machine is undefined for a PROTECTION.BAD\_ACCESS fault condition that is due to a protect violation, do not resume an application that has generated this fault. 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.

### 8.7.2.2 Faults Happening During Instruction Execution

The following fault types occur during instruction execution:

- CONSTRAINT.RANGE
- OPERATION.UNALIGNED
- MACHINE.PARITY
- 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). Because the state of the machine is undefined for a MACHINE.PARITY fault condition, do not resume an application that has generated this fault.

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. If 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.

### 8.7.2.3 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
- PROTECTION.BAD\_ACCESS (due to detect violation)

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
- PROTECTION.BAD\_ACCESS (due to detect violation)

The effect of specific fault types on a program is defined in [Section 8.10, “Fault Reference” on page 8-21](#) under the heading *Program State Changes*.

### 8.7.3 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 8.10, “Fault Reference” on page 8-21](#).

### 8.7.4 Returning to the Point in the Program Where the Fault Occurred

As described in [Section 8.7.2, “Program Resumption Following a Fault” on page 8-14](#), 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 6-86](#).

## 8.7.5 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.

## 8.7.6 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 8-3](#) 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 8.10, “Fault Reference” on page 8-21](#).

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 8.9, “Precise and Imprecise Faults” on page 8-19](#).

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 8.10, “Fault Reference” on page 8-21](#). Further discussion of these flags is provided in [Chapter 9, “Tracing and Debugging”](#).

**Table 8-3. 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 (see [Figure 13-6](#)). It controls whether unaligned memory accesses generate a fault. See [Section 14.3.2, “Bus Transactions across Region Boundaries” on page 14-10](#).

## 8.8 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.

### 8.8.1 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. (See [Figure 8-4](#).)
- 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.

If the fault handling procedure is not able to perform a recovery action, it performs one of the actions described in [Section 8.7.2, “Program Resumption Following a Fault” on page 8-14](#).

If 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 7-7](#), except that the arithmetic controls field from the fault record is copied into the AC register. If 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.

### 8.8.2 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.

### 8.8.3 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:

- If 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.
- If the fault occurs while in supervisor mode, the processor creates a new frame on the current stack. If the processor is executing a supervisor procedure when the fault occurred, the current stack is the supervisor stack; if 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. (See [Figure 8-4](#).)
- The processor gets the IP for the first instruction of the fault handling procedure from the system procedure table (using the index provided in the fault table entry).
- The processor stores the fault return code ( $001_2$ ) in the PFP register return type field. If 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. If 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 7-7](#) with the addition of the following:

- The fault record arithmetic controls field is copied into the AC register.
- If 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. (If the processor was in user mode when the fault occurred, this operation causes a switch from the supervisor stack to the user stack.)
- If 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.

### 8.8.4 Faults and Interrupts

If an interrupt occurs during an instruction that will fault, an instruction that has already faulted, or fault handling procedure selection, the processor handles the interrupt in the following way:

1. It completes the selection of the fault handling procedure.
2. It creates the fault record.
3. It services the interrupt just prior to executing the first instruction of the fault handling procedure.
4. It handles the fault upon return from the interrupt.

Handling the interrupt before the fault reduces interrupt latency.

## 8.9 Precise and Imprecise Faults

As described in [Section 8.10.6, “PARALLEL Faults” on page 8-29](#), 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 8.9.5, “Controlling Fault Precision” on page 8-20](#) for more information. Faults are categorized as precise, imprecise and asynchronous. The following subsections describe each.

### 8.9.1 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. However, they can be found in parallel fault records with asynchronous faults.

### 8.9.2 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.

### 8.9.3 Asynchronous Faults

Asynchronous faults are those whose occurrence has no direct relationship to the instruction pointer. This group includes MACHINE faults, which are implemented on the 80960Hx.

## 8.9.4 No Imprecise Faults (AC.nif) Bit

The Arithmetic Controls no imprecise faults (AC.nif) bit controls imprecise fault generation. If AC.nif is set, out of order instruction execution is disabled and all faults generated are precise. Therefore, setting this bit will reduce processor performance. If 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.

## 8.9.5 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**.

## 8.10 Fault Reference

This section describes each fault type and subtype and gives detailed information about what is stored in the various fields of the fault record. The section is organized alphabetically by fault type. The following paragraphs describe the information that is provided for each fault type.

Fault Type:	Gives the number that appears in the fault record fault-type field when the fault is generated.
Fault Subtype:	Lists the fault subtypes and the number associated with each fault subtype.
Function:	Describes the purpose and handling of the fault type and each subtype.
RIP:	Describes the value saved in the image of the RIP register in the stack frame that the processor was using when the fault occurred. In the RIP definitions, “next instruction” refers to the instruction directly after the faulting instruction or to an instruction to which the processor can logically return when resuming program execution.  Note that the discussions of many fault types specify that the RIP contains the address of the instruction that would have executed next had the fault not occurred.
Fault IP:	Describes the contents of the fault record’s fault instruction pointer field, typically the faulting instruction’s IP.
Fault Data:	Describes any values stored in the fault record’s fault data field.
Class:	Indicates if a fault is precise or imprecise.
Program State Changes:	Describes the process state changes that would prevent re-executing the faulting instruction if applicable.
Trace Reporting:	Relates whether a trace fault (other than PRERET) can be detected on the faulting instruction, also if and when the fault is serviced.
Notes:	Additional information specific to particular implementations of the i960 processor architecture.

## 8.10.1 ARITHMETIC Faults

Fault Type:	3H																
Fault Subtype:	<b>Number</b>	<b>Name</b>															
	0H	Reserved															
	1H	INTEGER_OVERFLOW															
	2H	ZERO_DIVIDE															
	3H-FH	Reserved															
Function:	<p>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 <i>n</i> least significant bits are stored in the destination, where <i>n</i> is destination size. Instructions that generate this fault are:</p> <table border="0"> <tr> <td><b>addi</b></td> <td><b>subi</b></td> <td><b>stis</b></td> </tr> <tr> <td><b>stib</b></td> <td><b>shli</b></td> <td><b>ADDI&lt;cc&gt;</b></td> </tr> <tr> <td><b>muli</b></td> <td><b>divi</b></td> <td><b>SUBI&lt;cc&gt;</b></td> </tr> </table> <p>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:</p> <table border="0"> <tr> <td><b>divo</b></td> <td><b>divi</b></td> </tr> <tr> <td><b>ediv</b></td> <td><b>remi</b></td> </tr> <tr> <td><b>remo</b></td> <td><b>modi</b></td> </tr> </table>		<b>addi</b>	<b>subi</b>	<b>stis</b>	<b>stib</b>	<b>shli</b>	<b>ADDI&lt;cc&gt;</b>	<b>muli</b>	<b>divi</b>	<b>SUBI&lt;cc&gt;</b>	<b>divo</b>	<b>divi</b>	<b>ediv</b>	<b>remi</b>	<b>remo</b>	<b>modi</b>
<b>addi</b>	<b>subi</b>	<b>stis</b>															
<b>stib</b>	<b>shli</b>	<b>ADDI&lt;cc&gt;</b>															
<b>muli</b>	<b>divi</b>	<b>SUBI&lt;cc&gt;</b>															
<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.																

## 8.10.2 CONSTRAINT Faults

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

### 8.10.3 MACHINE Faults

Fault Type:

8H

Fault Subtype:

0H-1H	Reserved
2H	Parity
3H-FH	Reserved

Function:

Indicates errors relative to external memory or buses.

When the parity checking mechanism is activated in the PMCON of a memory region (see [Figure 14-2](#)), the bus controller generates a MACHINE.PARITY\_ERROR fault when detecting a parity violation on a load operation.

When several parity fault conditions are detected during the execution of an instruction, the information relative to the first parity violation is recorded in the fault data field of the parity fault record. Subsequent parity violations are discarded, until the parity recording mechanism is reactivated by the processor upon entry into the MACHINE fault handler.

If another parity fault is detected after reactivation of the parity detection mechanism, an override fault is generated instead of the parity fault. [Figure 8-5](#) shows the Parity Fault Record.

RIP:

No defined value

Fault IP:

Undefined

Fault Data:

The optional fault data field for MACHINE.PARITY\_ERROR faults consists of:

- Address of the faulting access at FP-24, the memory address of the faulting load or fetch.
- Access type word at FP-28, the size and type (load or fetch) of the access.

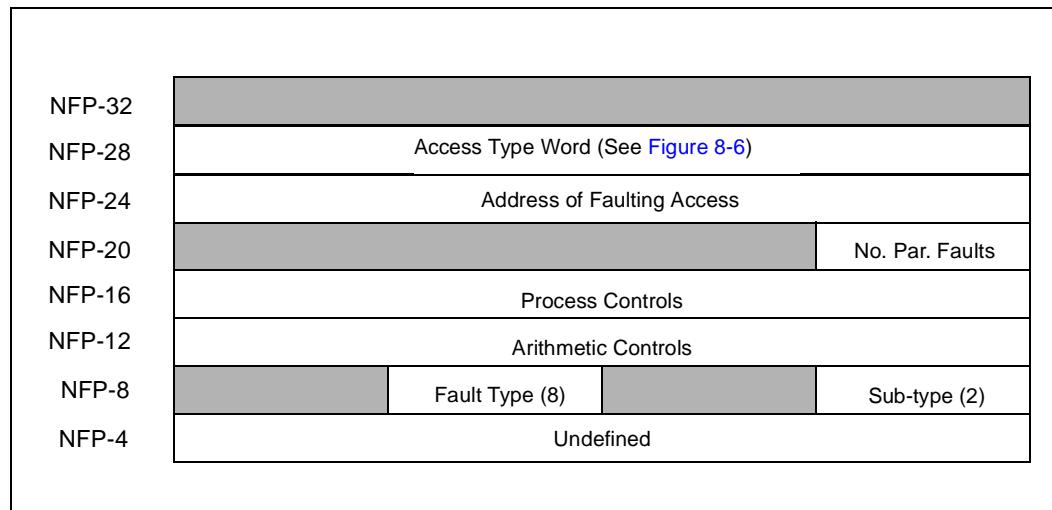
Class:

Parity faults are asynchronous: they have no direct relationship with the current IP. Parity faults are never precise, even in No Imprecise Faults mode (AC.nif=1). As a result, they lead to the creation of parallel fault records if their detection coincides with the detection of another fault, even in No Imprecise Faults mode. Use **syncf** to synchronize parity faults explicitly.

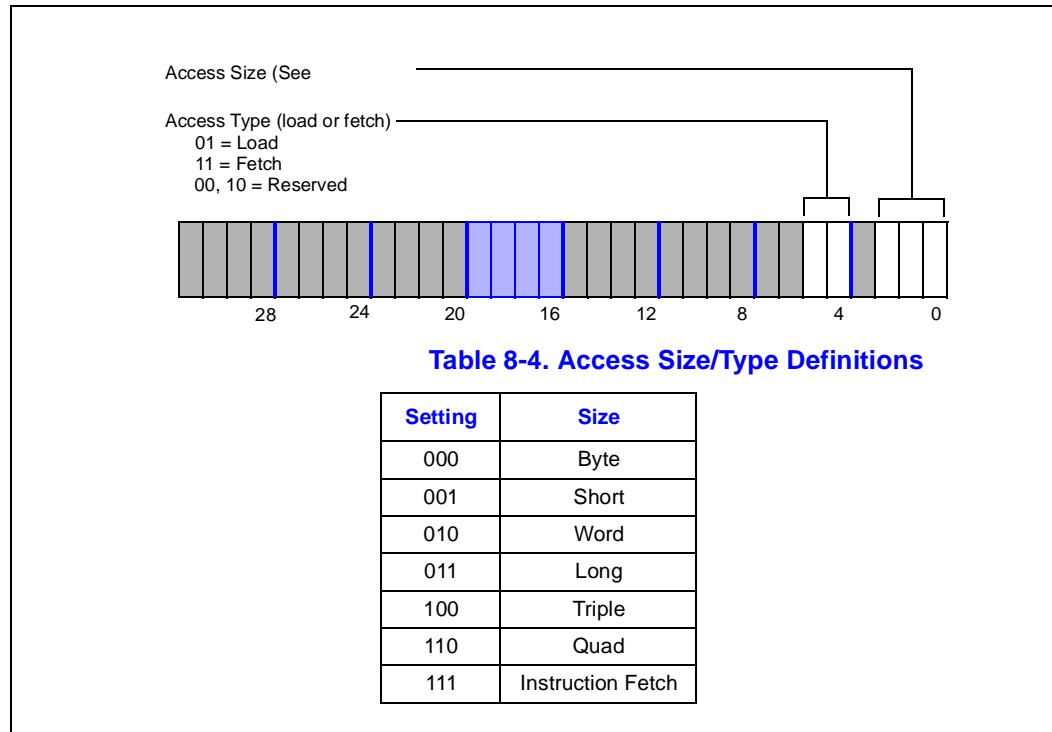
Program State Changes:

The contents of the destination register are lost.

**Figure 8-5. Parity Fault Record**



**Figure 8-6. MACHINE.PARITY\_ERROR Fault Access Type Word Definition (NFP-28)**



## 8.10.4 OPERATION Faults

Fault Type:	2H	
Fault Subtype:	<b>Number</b>	<b>Name</b>
	0H	Reserved
	1H	INVALID_OPCODE
	2H	UNIMPLEMENTED
	3H	UNALIGNED
	4H	INVALID_OPERAND
	5H - FH	Reserved
Function:	<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-existent SFR or 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>	
RIP:	No defined value.	
Fault IP:	Address of the faulting instruction.	
Fault Data:	<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>	
Class:	Imprecise.	

Program State Changes:	For the INVALID_OPCODE and UNIMPLEMENTED faults (case: store to MMR), the destination of the faulting instruction is not modified. (For the UNALIGNED fault, the memory operation completes correctly before the fault is reported.) In all other cases, the destination is undefined.
Trace Reporting:	OPERATION.UNALIGNED fault: the trace is reported upon return from the OPERATION fault handler. All other subtypes: the trace event is lost.
Note:	OPERATION.UNALIGNED fault is not implemented on i960 Kx and Sx CPUs.

## 8.10.5 OVERRIDE Faults

Fault Type:	Fault table entry = 10H
	See <a href="#">Section 8.6.5, “Override Faults” on page 8-11</a> for more information.
Function:	The override fault handler must be accessed through a system-supervisor call. Local and system-local override fault handlers are not supported and have an unpredictable behavior. Tracing is disabled upon entry into the override fault handler (PC.te is cleared). It is restored upon return from the handler. To prevent infinite internal loops, the override fault handler should not set PC.te.
Trace Reporting:	Same behavior as if the override condition had not existed. Refer to the description of the original program fault.
Note:	Fault handlers must not be placed in a GMU-protected area.

## 8.10.6 PARALLEL Faults

Fault Type:	Fault table entry = 0H Fault type in fault record = fault type of one of the parallel faults.
Fault Subtype:	Fault subtype of one of the parallel faults.
Number of Faults:	Number of parallel faults.
Function:	See <a href="#">Section 8.6.4, “Parallel Faults” on page 8-9</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 size of the fault record is variable and depends on the number of parallel faults. The number of parallel faults is indicated in the Parallel Faults Field (NFP-20). See <a href="#">Figure 8-3</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.
RIP:	If all parallel fault types allow a RIP to be defined, the RIP is the next instruction in the flow of execution, otherwise it is undefined.
Fault IP:	IP of one of the faulting instructions.
Class:	Imprecise.
Program State Changes:	State changes associated with all the parallel faults.
Trace Reporting:	If all parallel fault types allow for a resumption trace, then a trace is reported upon return from the parallel fault handler, or else it is lost.

## 8.10.7 PROTECTION Faults

Fault Type: 7H

Function:	Number	Name
	Bit 0	Reserved
	Bit 1	LENGTH
	Bits 2-4	Reserved
	Bit 5	BAD_ACCESS
	Bits 6-7	Reserved

Indicates that a program or procedure is attempting to perform an illegal operation that the architecture protects against.

A PROTECTION.LENGTH fault is generated when the index operand used in a **calls** instruction points to an entry beyond the extent of the system procedure table.

A PROTECTION.BAD\_ACCESS fault is generated when a program or procedure attempts a memory access that violates the permissions defined in the GMU memory protection or the GMU memory detection registers.

A memory access that violates the permissions defined in the GMU memory protection or detection registers causes a PROTECTION.BAD\_ACCESS fault. BAD\_ACCESS faults due to protection violations are handled immediately by the processor (breaking instruction execution), whereas BAD\_ACCESS faults due to detection violations are pended until the instruction completes. If during the execution of a complex instruction, the GMU signals a detect violation, the GMU records the information relative to this first violation in the fault data field. Subsequent detect violations for this instruction will not modify the data field, but a subsequent protect violation will. The protect violation interrupts the instruction execution and the final data field of the fault reported describes the memory protection violation access.

The words of the optional fault data field for PROTECTION.BAD\_ACCESS faults are:

- The address of the faulting access at FP-24. It holds the address of the first protect violation encountered or of the first detect violation if no protect violation occurred.
- The access type word at FP-28. It shows the size and type of the first protect violation encountered or of the first detect violation if no protect violation occurred.

- The indicator word at FP-32. It shows all bits corresponding to access violations during a memory access: if protect and detect regions overlap and a memory access triggers multiple region violations, all offending regions are listed in the indicator word of the fault record. However, the indicator word shows only the information relative to one machine memory access, which may be only a subset of all memory accesses done by the processor during the execution of a complex operation, like a call or return. In case of multiple memory accesses, the indicator word behaves like the other data fields: it contains the information relative to the first memory access containing a protect violation, or the information relative to the first memory access that triggered a detect violation, if no protect violation occurred.

In a region protected by the GMU, only code fetched from external memory will generate a PROTECTION.BAD\_ACCESS fault. Once the instructions are cached, the BAD\_ACCESS fault will not be generated when the instructions are executed. However, accessing data in a region protected by the GMU will cause a BAD\_ACCESS fault, whether the region is cached or not.

Due to instruction prefetching, a spurious BAD\_ACCESS fault may be generated when the target of a branch is in a region fetch-protected by the GMU, if the branch is predicted to be taken, but is not. For application debugging with the GMU, conditional branches to regions protected by the GMU should always be predicted not taken.

When a processor is configured to have more than 5 register cache frames, the number of frames in excess of 5 are located in the upper portion of SRAM. If this region is protected by the GMU, a BAD\_ACCESS fault will not occur when a register set is pushed or popped from its frame during a call or return operation. A BAD\_ACCESS fault only occurs if the application program directly loads or stores data from the SRAM region protected by the GMU.

Note that the faultIP field of the fault record is undefined for BAD\_ACCESS faults.

RIP:

IP of the faulting instruction.

GMU Protection: Undefined

GMU Detection: IP of the instruction that would have executed next if the fault had not occurred.

Fault IP:

LENGTH: IP of the faulting instruction.

BAD\_ACCESS (Protect): Undefined

BAD\_ACCESS (Detect): Undefined

Fault Data:

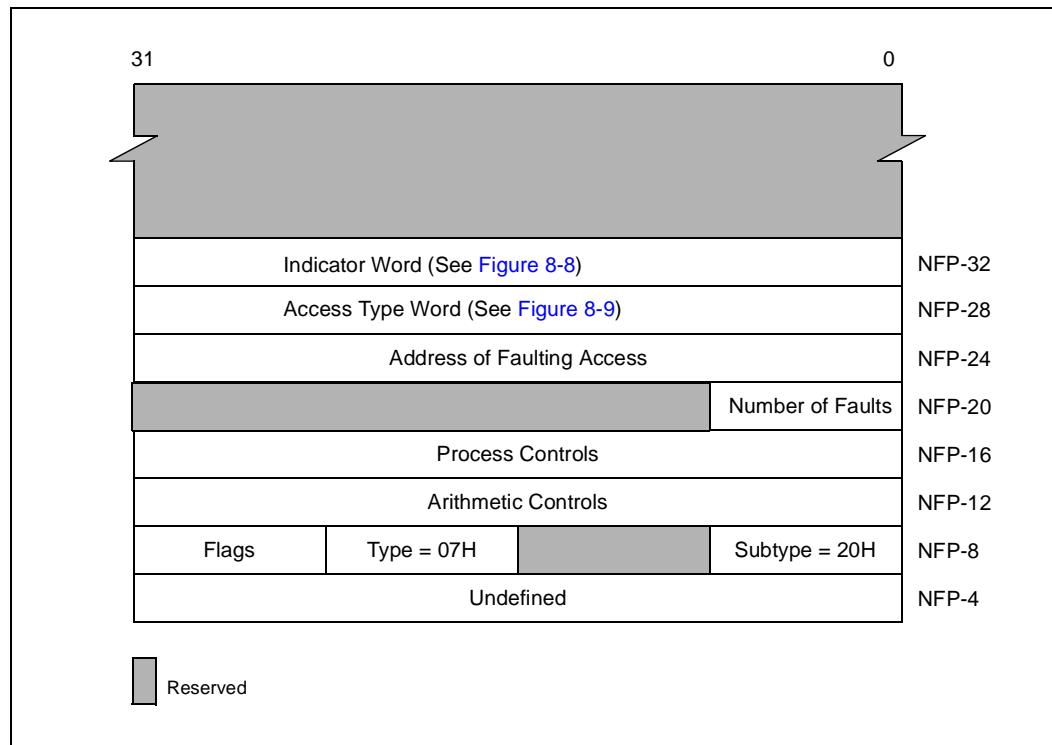
When a PROTECTION.BAD\_ACCESS fault occurs, the processor writes a fault record to memory. This fault record consists of the indicator word, the access type word, and the address of the faulting access. [Figure 8-7](#) outlines their relative position with respect to the FP.

Class:	Imprecise.
Program State Changes:	<p>LENGTH: The instruction does not execute.</p> <p>BAD_ACCESS (Protect): Memory is not updated. Contents of the destination register are lost.</p> <p>BAD_ACCESS (Detect): Memory and register updated with correct value (i.e., the instruction completes prior to faulting).</p>
Trace Reporting:	<p>PROTECTION.LENGTH: The trace event is lost.</p> <p>GMU Protection: The trace event is lost.</p> <p>GMU Detection: The trace is reported upon return from the fault handler.</p>
Notes:	<p>The fault handler should not attempt to return from a PROTECTION.BAD_ACCESS fault caused by a GMU Protection condition because the stack frame cache may be in an undefined state.</p> <p>The fault handler can determine if the fault was caused by a GMU Protection condition or a GMU Detection condition by analyzing the indicator word in the fault record (<a href="#">Figure 8-7</a>). Bits 1 to 0 define the GMU Memory Protect Register pair affected, while bits 21 to 16 define the affected GMU Memory Detect Register pair. See <a href="#">Section 12.3, “GMU Register Description” on page 12-4</a> for a detailed description of these registers.</p>

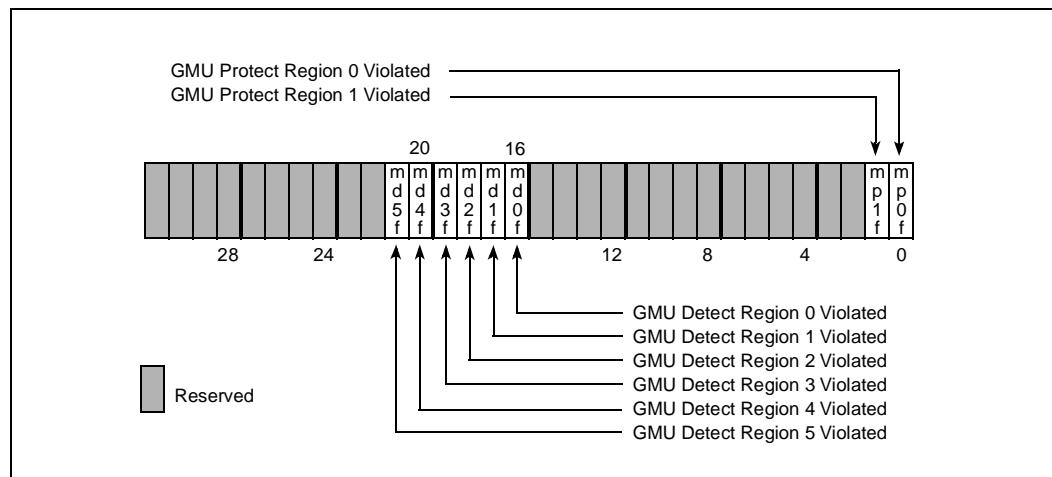
On the i960 Hx processor, it is possible to get a PROTECTION.BAD\_ACCESS fault by placing instructions in the 16 words directly below an area of memory (i.e., with smaller values for the address) that is protected by the GMU against instruction execution. This is because of the instruction prefetch feature of the i960 Hx processor. The processor may try to fetch words that are never actually executed but that are protected by the GMU against instruction execution.

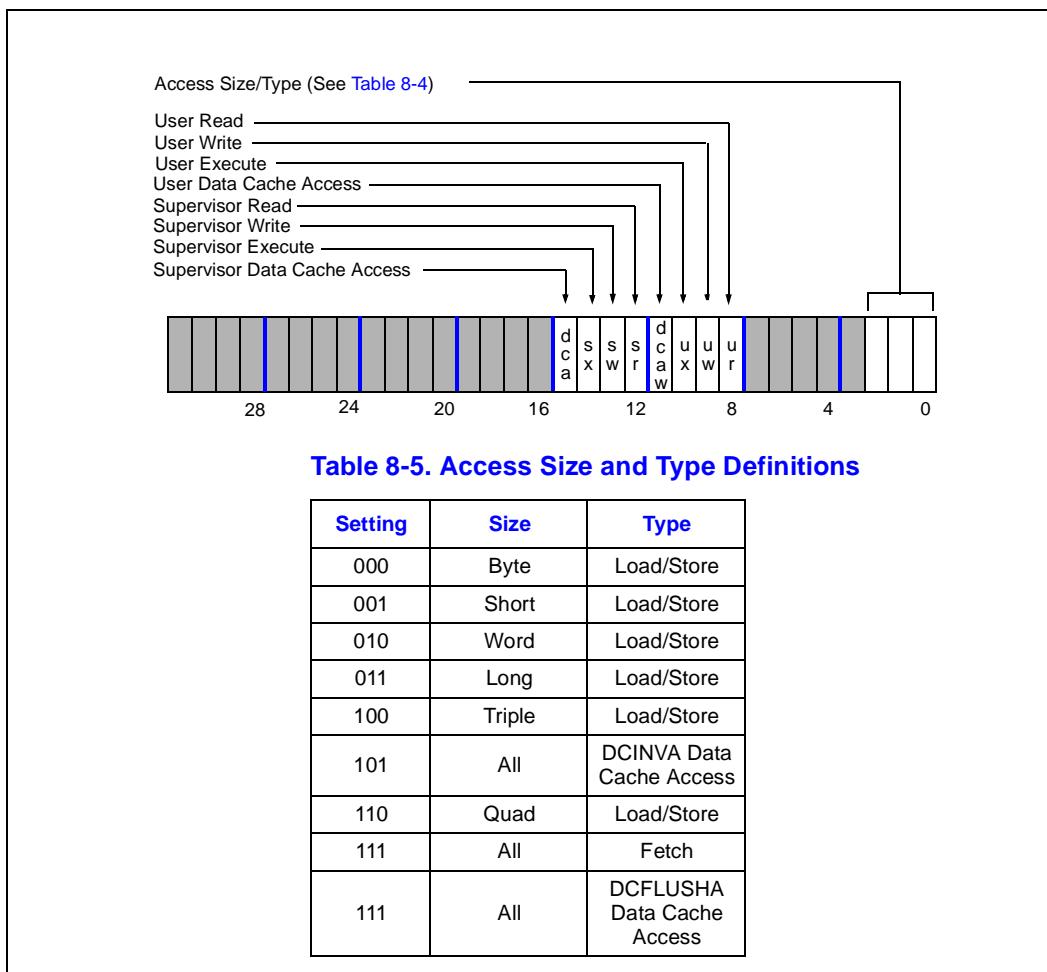
If any of the PROTECTION, PARALLEL, OVERRIDE or TRACE fault handlers is protected by the GMU, infinite recursion within the fault handlers may occur. If the PROTECTION handler is fetch-protected by the GMU, the processor loops infinitely with no way of being interrupted except through reset.

**Figure 8-7. PROTECTION.BAD\_ACCESS Fault Record**



## Figure 8-8. Indicator Word (NFP-32)



**Figure 8-9. Access Type Fault (NFP-28)**

## 8.10.8 TRACE Faults

Fault Type:	1H											
Fault Subtype:	<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 <a href="#">Chapter 9, “Tracing and Debugging”</a> .											
	<p>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.</p>											
	<p>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:</p>											
	<table border="0"> <tr> <td style="vertical-align: top;">INSTRUCTION</td><td>Generates a trace event following every instruction.</td></tr> <tr> <td style="vertical-align: top;">BRANCH</td><td>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).</td></tr> <tr> <td style="vertical-align: top;">CALL</td><td>Generates a trace event following any call or branch-and-link instruction or an implicit fault call.</td></tr> <tr> <td style="vertical-align: top;">RETURN</td><td>Generates a trace event following a <b>ret</b>.</td></tr> <tr> <td style="vertical-align: top;">PRERETURN</td><td>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.</td></tr> </table>		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.
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 **calls** instruction that references a supervisor procedure entry in the system procedure table and on a return from a supervisor procedure where the return status type in the PFP register is  $010_2$  or  $011_2$ .

**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.

## 8.10.9 TYPE Faults

Fault Type:	AH									
Fault Subtype:	<b>Number</b>	<b>Name</b>								
	0H	Reserved								
	1H	MISMATCH								
	2H-FH	Reserved								
Function:	Indicates a program or procedure attempted to perform an illegal operation on an architecture-defined data type or a typed data structure.									
	A TYPE.MISMATCH fault is generated when attempts are made to:									
	<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 Hx processor are:           <table border="0"> <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. See <a href="#">Figure 14-3</a>.</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.srp is set. See <a href="#">Figure 14-3</a>.</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> <li>• Access an SFR while the processor is in user mode.</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>										
RIP:	No defined value.									
Fault IP:	IP of the faulting instruction.									
Class:	Imprecise.									
Program State Changes:	The fault happens before execution of the instruction. The machine state is not changed.									
Trace Reporting:	The trace event is lost.									

This chapter describes the i960® Hx 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 Hx processor also provides twelve hardware breakpoint registers that generate trace events and trace faults. Six registers are dedicated to trapping on instruction execution addresses, while the remaining six registers can trap on the addresses of data accesses.

## 9.1 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.

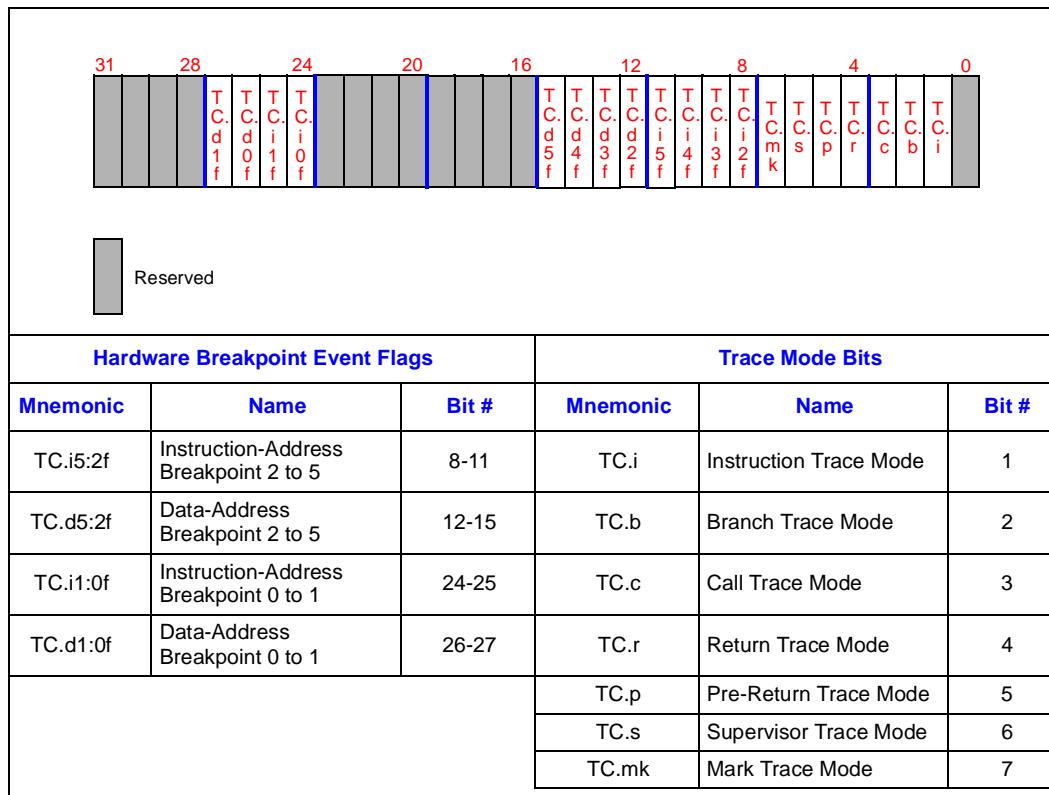
- TC register mode bits
- DAB0-DAB5 registers' address field and enable bit (in the control table)
- System procedure table supervisor-stack-pointer field trace control bit
- IPB0-IPB5 registers' address field (in the control table)
- PC register trace enable bit
- PFP register return status field prereturn trace flag (bit 3)
- BPCON and XBPCON register breakpoint mode bits and enable bits (in the control table)

These controls are described in the following subsections.

### 9.1.1 Trace Controls (TC) Register

The TC register (Figure 9-1) allows software to define conditions that generate trace events.

**Figure 9-1. Trace Controls (TC) Register**



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 9.2 on page 9-3](#). 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 Hx processor, TC register bits 0, 16 through 23 and 28 through 31 are reserved. Software must initialize these bits to zero and cannot modify them afterwards.

## 9.1.2 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 (see [Section 3.6.3, “Process Controls \(PC\) Register” on page 3-24](#)). 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 9.5.2, “Tracing on Calls and Returns” on page 9-11](#).

The update of PC.te through **modpc** may take up to four non-branching instructions to take effect. The update of PC.te through call and return operations is immediate.

The trace-fault-pending flag, in the PC field of the fault record, allows the processor to remember to service a trace fault when a trace event is detected at the same time as another event (e.g., non-trace fault, interrupt). The non-trace fault event is serviced before the trace fault, and depending on the event type and execution mode, the trace-fault-pending flag in the PC field of the fault record may be used to generate a fault upon return from the non-trace fault event (see [Section 9.5.2.4, “Tracing on Return from Implicit Call: Fault Case” on page 9-13](#)).

## 9.2 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 9.4, “Handling Multiple Trace Events” on page 9-11](#) for a description of processor function when multiple trace events occur.

### 9.2.1 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.

## 9.2.2 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.

## 9.2.3 Call Trace

When the call-trace mode is enabled in TC (TC.c = 1) and PC.te is set after the call, 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 9.5.2.1, “Tracing on Explicit Call” on page 9-12](#) 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 9.5.2.2, “Tracing on Implicit Call” on page 9-12](#) for a complete description of this case.

When the processor services a trace fault, it sets the prereturn-trace flag (PFP register bit 3) in the new frame created by the call operation or in the current frame if a branch-and-link operation was performed. The processor uses this flag to determine whether or not to signal a prereturn-trace event on a **ret** instruction.

## 9.2.4 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 9.5.2.3, “Tracing on Return from Explicit Call” on page 9-13](#).

A return from fault may be traced and a return from interrupt cannot. See [Section 9.5.2.4, “Tracing on Return from Implicit Call: Fault Case” on page 9-13](#) and [Section 9.5.2.5, “Tracing on Return from Implicit Call: Interrupt Case” on page 9-13](#) for details.

## 9.2.5 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.

## 9.2.6 Supervisor Trace

When supervisor-trace mode is enabled in TC and PC.te is set, the processor generates a supervisor-trace fault after either of the following:

- A call-system instruction (**calls**) executes from user mode and the procedure table entry is for a system-supervisor call.
- A **ret** instruction executes from supervisor mode and the return-type field is set to  $010_2$  or  $011_2$  (i.e., return from **calls**).

This trace mode allows a debugging program to determine kernel-procedure call boundaries within the instruction stream.

## 9.2.7 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.

### 9.2.7.1 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.

### 9.2.7.2 Hardware Breakpoints

The hardware breakpoint registers are provided to enable generation of trace faults on instruction execution and data access.

The i960 Hx processor implements six instruction and six data address breakpoint registers, denoted IPB0 through IPB5 and DAB0 through DAB5. 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 and XBPCON) registers. BPCON and XBPCON enable the data address breakpoint registers, and set the specific modes of these registers. Hardware breakpoints are globally enabled by the process controls trace enable bit (PC.te).

The IPBx, DABx, BPCON and XBPCON registers may be accessed using normal load and store instructions. The application must be in supervisor mode for a legal access to occur. See [Section 3.3, “Memory-Mapped Control Registers” on page 3-6](#) for more information on the address for each register.

Applications must request modification rights to the hardware breakpoint resources, before attempting to modify these resources. Rights are requested by executing the **sysctl** instruction, as described in the following section.

### 9.2.7.3 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 will be 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 or XBPCON) registers without first obtaining rights, an OPERATION.UNIMPLEMENTED fault will be generated. In this case, the breakpoint resource will not be 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 will be 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 9-1](#).

**Table 9-1. *src/dst* Encoding**

<i>src/dst 7:4</i>	<i>src/dst 3:0</i>
Number of Available Data Address Breakpoints	Number of Available Instruction Breakpoints

**NOTE:** SRC3 31:8 are reserved and will 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.
```

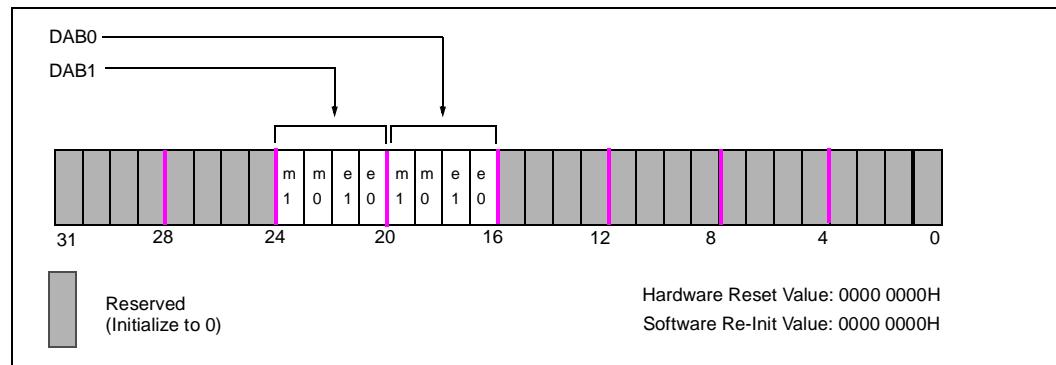
Assume in this example that after execution of the **sysctl** instruction, the value of r4 is 0000 0066H. This indicates that the application has gained modification rights to all instruction and all data address breakpoint registers. If the value returned is zero, the application has not gained the rights to the breakpoint resources.

Because the i960 Hx 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.

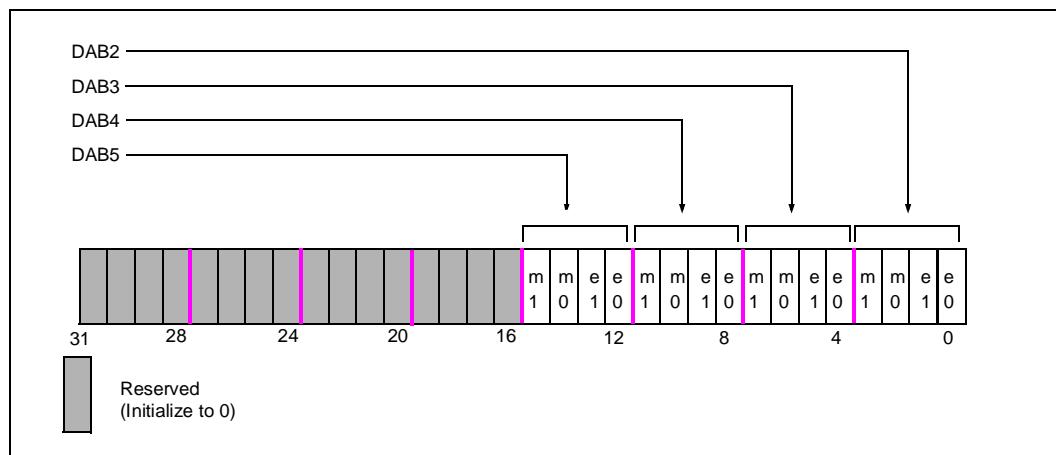
### 9.2.7.4 Breakpoint Control Register

The format of the BPCON and XBPCON registers are shown in [Figure 9-2](#) and [Figure 9-3](#). Each breakpoint has four control bits associated with it: two mode and two enable bits. The enable bits (DABx.e0, DABx.e1) in BPCON and XBPON act to enable or disable the data address breakpoints, while the mode bits (DABx.m0, DABx.m1) dictate which type of access will generate a break event.

**Figure 9-2. Breakpoint Control Register (BPCON)**



**Figure 9-3. Extended Breakpoint Control Register (XBPON)**



Programming the BPCON and XBPCON registers is summarized in [Table 9-2](#) and [Table 9-3](#).

**Table 9-2. Configuring the Data Address Breakpoint (DAB) Registers**

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 and XBPCON control what type of access generates a fault, trace message, or break event, as summarized in [Table 9-3](#)

**Table 9-3. Programming the Data Address Breakpoint (DAB) Modes**

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	Reserved.

### 9.2.7.5 Data Address Breakpoint (DAB) Registers

The format for the Data Address Breakpoint (DAB) registers is shown in [Figure 9-4](#). 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 or XBPCON and the appropriate DAB register. The mode bits for each DAB register, which are contained in BPCON or XBPCON (see section 9.2.7.4), qualify the access types that DAB will match. 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 will cause 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:

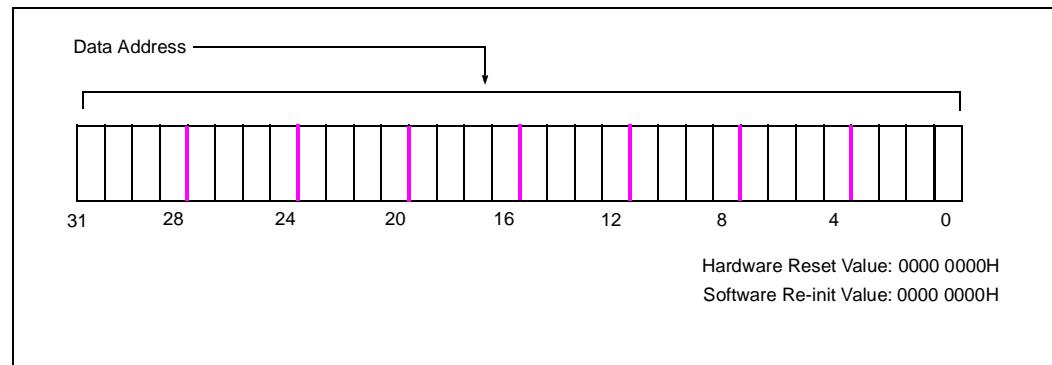
```
ldt 0x1000,r8
```

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 will always be reported in this case.

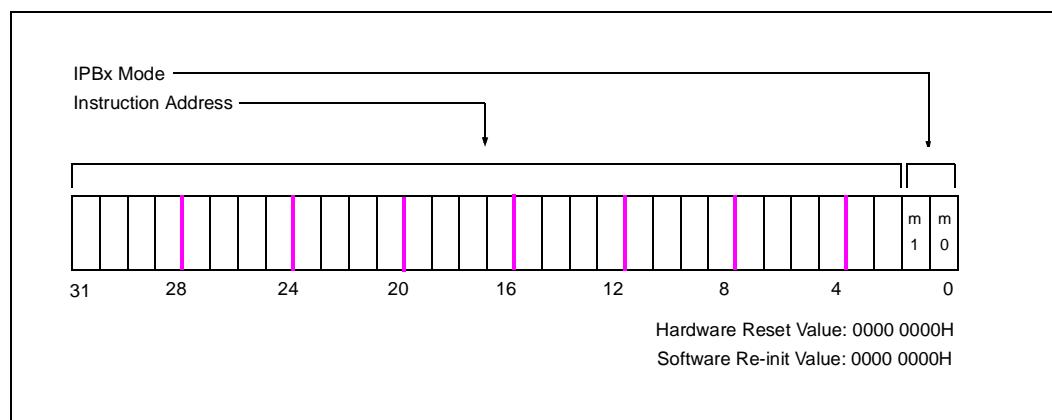
**Figure 9-4. Data Address Breakpoint (DAB) Register Format**



### 9.2.7.6 Instruction Breakpoint (IPB) Registers

The format for the instruction breakpoint registers is given in [Figure 9-5. Instruction Breakpoint \(IPB\) Register Format](#). The upper thirty 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.

**Figure 9-5. Instruction Breakpoint (IPB) Register Format**



Programming the instruction breakpoint register modes is shown in [Table 9-4](#)

On the i960 Hx processor, the instruction breakpoint memory-mapped registers can be read by using the **sysctl** instruction or by a word-length read instruction. 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 will always be 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 9-4. 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.

## 9.3

## 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.

## 9.4 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 will always report a supervisor event and it will always signal 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.

## 9.5 Trace Fault Handling Procedure

The processor calls the trace fault handling procedure when it detects a trace event. See [Section 8.7, “Fault Handling Procedures” on page 8-13](#) 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.

### 9.5.1 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 9.5.2.2, “Tracing on Implicit Call” on page 9-12](#) and [Section 9.5.2.5, “Tracing on Return from Implicit Call: Interrupt Case” on page 9-13](#) for detailed descriptions of tracing on calls and returns from interrupts.

### 9.5.2 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.

### 9.5.2.1 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 (see [Section 9.5.2.3, “Tracing on Return from Explicit Call” on page 9-13](#)). The **calls** instruction and all instructions of the procedure called are traced according to the new PC.te.

[Table 9-5](#) summarizes all cases; “a” and “x” are bit variables.

**Table 9-5. Tracing on Explicit Call**

Call Type	Calling Procedure Trace Enable	Calling Procedure Mode	Saved PFP.rt2:0	Called Procedure Trace Enable Bit
call, callx	PC.te	user or supervisor	000 <sub>2</sub>	PC.te
calls	PC.te	supervisor	000 <sub>2</sub>	PC.te
calls	PC.te	user	01t <sub>2</sub> Stores PC.te into bit 0 of PFP.rt2:0	SSP.te

**NOTE:** Refer to [Table 7-2 “Encoding of Return Status Field” on page 7-21](#).

### 9.5.2.2 Tracing on Implicit Call

Tracing on an implicit call happens before execution of the first instruction of the non-trace fault handler called. [Table 9-6](#) 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 9-6. 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>†</sup>	N.A.	001	a	a	a
10-Fault <sup>†</sup>	00	001	a	a	a
10-Fault <sup>†</sup>	10	001	a	SSP.te	SSP.te
00-Override Fault 00-Trace Fault	x	Type of trace fault not supported			
10-Override Fault 10-Trace Fault	00	Type of trace fault not supported			
10-Override Fault 10-Trace Fault	10	001	a	0	0
Interrupt	N.A.	111	a	0	0

<sup>†</sup> On i960<sup>®</sup> Hx processors, all faults except override and trace faults.

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.call 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 Hx processor, the override fault handler must be accessed through a system-supervisor call. Tracing is disabled on the way to the 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.

### 9.5.2.3 Tracing on Return from Explicit Call

Table 9-7 shows all cases.

**Table 9-7. Tracing on Return from Explicit Call**

PFP.rt2:0	Execution Mode PC.em	Trace Enable Used for Trace on Return
000 <sub>2</sub>	user or supervisor	PC.te
01t <sub>2</sub>	user	PC.te
01t <sub>2</sub>	super	t <sub>2</sub> (from PFP.r2:0)

Refer to Table 7-2 "Encoding of Return Status Field" on page 7-21.

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.

### 9.5.2.4 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.

### 9.5.2.5 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 Hx processor, the interrupt handler returns directly to the trace fault handler.

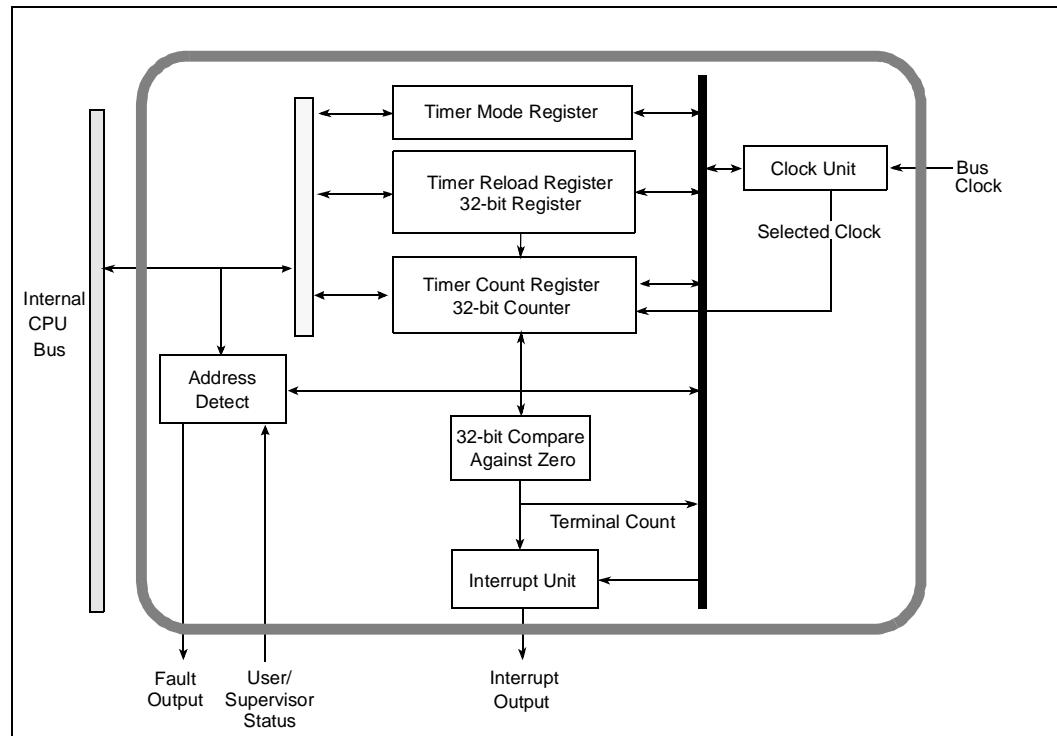
If the interrupt return is executed from user mode, the PC register is not restored and tracing of the return occurs according to the PC.te and TC.modes bit fields.



This chapter describes the i960® Hx processor's dual, independent 32-bit timers. Topics include timer registers (TMR<sub>x</sub>, TCR<sub>x</sub> and TRR<sub>x</sub>), 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 10-1](#) shows a diagram of the timer functions. See also [Figure 10-5](#) for the Timer Unit state diagram.

**Figure 10-1. Timer Functional Diagram**



**Table 10-1. Timer Performance Ranges**

Bus Frequency (MHz)	Max Resolution (ns)	Max Range (mins)
40	25	14.3
33	30.3	17.4
25	40	22.9
20	50	28.6
16	62.5	35.8

## 10.1 Timer Registers

As shown in [Table 10-2](#), 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 10.1.1, “Timer Mode Registers \(TMR0, TMR1\)” on page 10-2](#).
- Timer Count Register - contains the timer’s current count. See [Section 10.1.2, “Timer Count Register \(TCR0, TCR1\)” on page 10-5](#).
- Timer Reload Register - contains the timer’s reload count. See [Section 10.1.3, “Timer Reload Register \(TRR0, TRR1\)” on page 10-6](#).

**Table 10-2. Timer Registers**

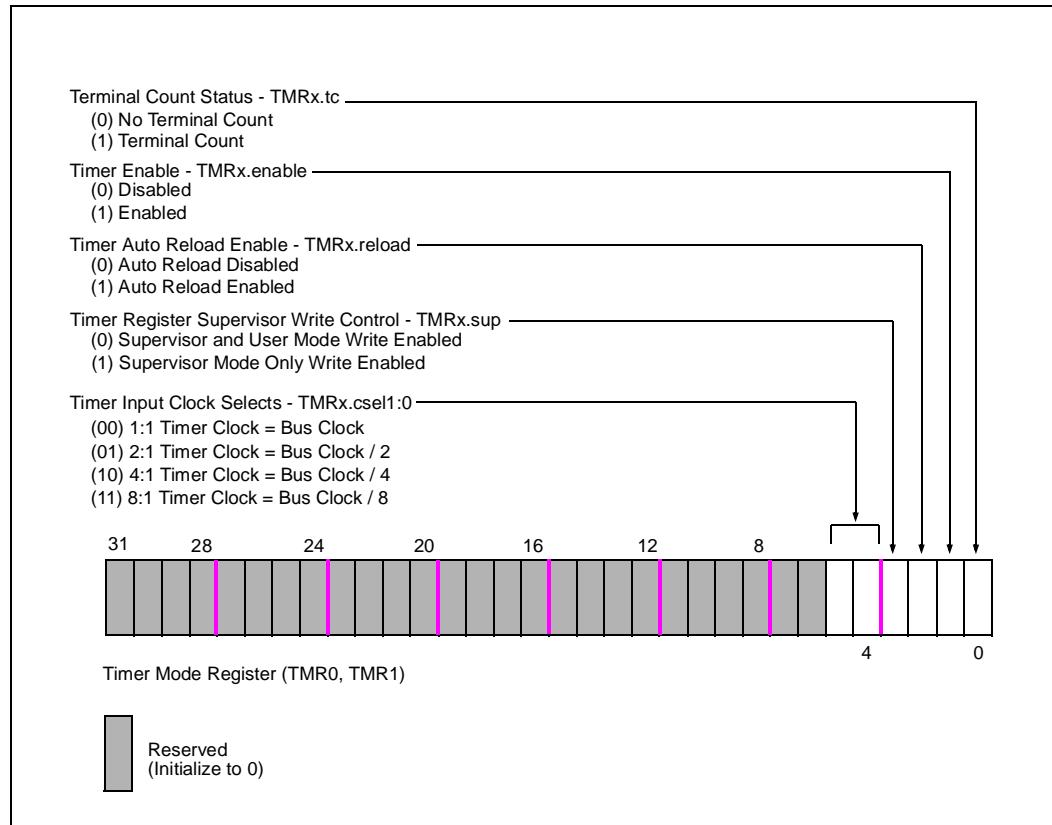
Timer Unit	Register Acronym	Register Name
Timer 0	TMR0	Timer Mode Register 0
	TCR0	Timer Count Register 0
	TRR0	Timer Reload Register 0
Timer 1	TMR1	Timer Mode Register 1
	TCR1	Timer Count Register 1
	TRR1	Timer Reload Register 1

For register memory locations, see [Table 3-5 “User Space Family Registers and Tables” on page 3-13](#).

### 10.1.1 Timer Mode Registers (TMR0, TMR1)

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 [Figure 10-2](#) and are summarized in [Table 10-4](#).

**Figure 10-2. Timer Mode Register (TMR0, TMR1)**



#### 10.1.1.1 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 13.2, “Initialization” on page 13-2](#).

### 10.1.1.2 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 <a href="#">Section 10.1.1.5, “Bits 4, 5 - Timer Input Clock Select (TMRx.csel1:0)” on page 10-5</a> . If TMRx.reload=0, the timer automatically clears TMRx.enable when the count reaches zero. If TMRx.reload=1, the bit remains set. See <a href="#">Section 10.1.1.3, “Bit 2 - Timer Auto Reload Enable (TMRx.reload)” on page 10-4</a> .
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 13.2, “Initialization” on page 13-2](#).

### 10.1.1.3 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: <ol style="list-style-type: none"> <li>1. Automatically loads TCRx with the value in the Timer Reload Register (TRRx), when TCRx value decrements to 0.</li> <li>2. Decrement TCRx until it equals 0 again.</li> </ol> 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. Refer to [Section 13.2, “Initialization” on page 13-2](#).

The processor clears this bit upon hardware or software reset. Refer to [Section 13.2, “Initialization” on page 13-2](#).

#### 10.1.1.4 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 13, "Initialization and System Requirements" on page 13-1](#).

#### 10.1.1.5 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 10-3](#). As shown in [Figure 10-1](#), 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 10-3. Timer Input Clock (TCLOCK) Frequency Selection**

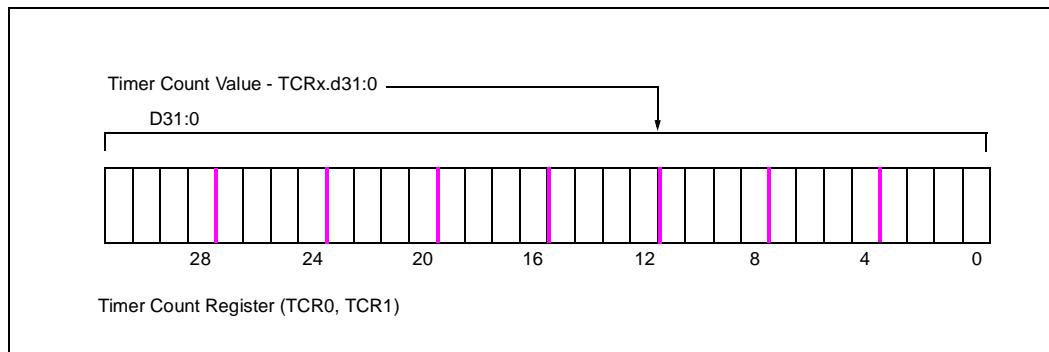
Bit 5 TMRx.csel1	Bit 4 TMRx.csel0	Timer Clock (TCLOCK)
0	0	Timer Clock = Bus Clock
0	1	Timer Clock = Bus Clock / 2
1	0	Timer Clock = Bus Clock / 4
1	1	Timer Clock = Bus Clock / 8

The processor clears these bits upon hardware or software reset (TCLOCK = Bus Clock).

#### 10.1.2 Timer Count Register (TCR0, TCR1)

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. If 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. [Figure 10-3](#) shows the timer count register.

**Figure 10-3. Timer Count Register (TCR0, TCR1)**



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 10.5, “Uncommon TCRX and TRRX Conditions” on page 10-10](#).)

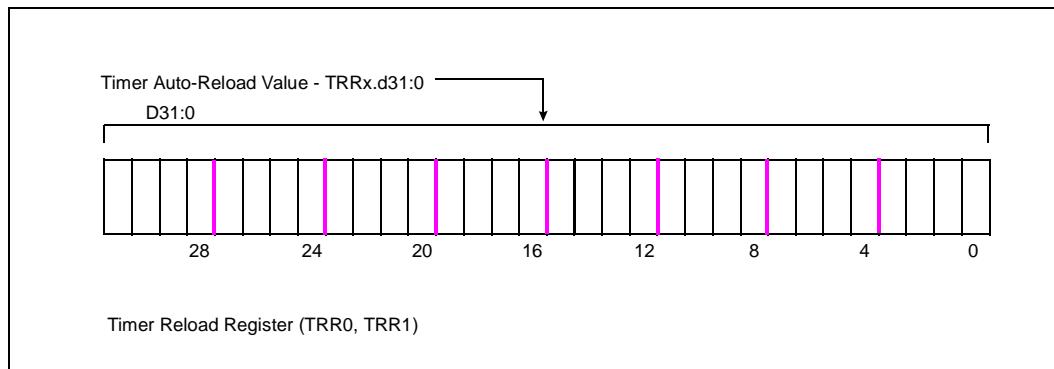
User software can read or write TCRx whether the timer is running or stopped. Bit 3 of TMRx determines user read/write control (see [Section 10.1.1.4](#)). The TCRx value is undefined after hardware or software reset.

### 10.1.3 Timer Reload Register (TRR0, TRR1)

The Timer Reload Register (TRRx; [Figure 10-4](#)) 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 10.5, “Uncommon TCRX and TRRX Conditions” on page 10-10](#) for more information.)

User software can access TRRx whether the timer is running or stopped. Bit 3 of TMRx determines read/write control (see [Section 10.1.1.4](#)). TRRx value is undefined after hardware or software reset.

**Figure 10-4. Timer Reload Register (TRR0, TRR1)**

## 10.2 Timer Operation

This section summarizes timer operation and describes load/store access latency for the timer registers.

### 10.2.1 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 10.3, “Timer Interrupts” on page 10-10](#) for more information. The timer checks the value of the timer reload bit (TMRx.reload) setting. If TMRx.reload = 1, the processor:

- Automatically reloads TCRx with the value in the Timer Reload Register (TRRx).
- Decrement TCRx until it equals 0 again.

This process repeats until software clears TMRx.reload or TMR.enable.

If 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 10-4. 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.

**NOTES:**

X = don't care

N = a number between 1H and FFFF FFFFH

## 10.2.2 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 10-5](#) 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 10-5. Timer Responses to Register Bit Settings**

Name	Status	Action
(TMRx.tc) Terminal Count Bit 0	<b>READ</b>	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 if TMRx.reload=0.
	<b>WRITE</b>	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	<b>READ</b>	Bit is available 1 bus clock after executing a read instruction from TMRx.
	<b>WRITE</b>	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	<b>READ</b>	Bit is available 1 bus clock after executing a read instruction from TMRx.
	<b>WRITE</b>	Writing a '1' enables the reload capability within 1 bus clock after the store instruction to TMRx has executed. The timer loads TRR <sub>x</sub> data into TCR <sub>x</sub> and decrements this value during the next bus clock cycle.
(TMRx.sup) Timer Register Supervisor Write Control Bit 3	<b>READ</b>	Bit is available 1 bus clock after executing a read instruction from TMRx.
	<b>WRITE</b>	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	<b>READ</b>	Bits are available 1 bus clock after executing a read instruction from TMRx.csel1:0 bit(s).
	<b>WRITE</b>	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	<b>READ</b>	The current TCRx count value is available within 1 bus clock cycle after executing a read instruction from TCRx. If the timer is running, the pre-decremented value is returned as the current value.
	<b>WRITE</b>	The value written to TCRx becomes the active value within 1 bus clock cycle. If the timer is running, the value written is decremented in the current clock cycle.
(TRRx.d31:0) Timer Reload Register	<b>READ</b>	The current TRR <sub>x</sub> count value is available within 1 bus clock after executing a read instruction from TRR <sub>x</sub> . If the timer is transferring the TRR <sub>x</sub> count into TCR <sub>x</sub> in the current count cycle, the timer returns the new TCR <sub>x</sub> count value to the executing read instruction.
	<b>WRITE</b>	The value written to TRR <sub>x</sub> becomes the active value stored in TRR <sub>x</sub> within 1 bus clock cycle. If the timer is transferring the TRR <sub>x</sub> value into the TCR <sub>x</sub> , data written to TRR <sub>x</sub> is also transferred into TCR <sub>x</sub> .

## 10.3 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 11, “Interrupts”](#) 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.

## 10.4 Powerup/Reset Initialization

Upon power up, external hardware reset or software reset (**sysctl**), the timer registers are initialized to the values shown in [Table 10-6](#).

**Table 10-6. 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

## 10.5 Uncommon TCRX and TRRX Conditions

[Table 10-4](#) 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 10-7](#) details the conditions and results when these conditions are set.

**Table 10-7. Uncommon TMRx Control Bit Settings**

<b>TRRx</b>	<b>TCRx</b>	<b>Bit 2 (TMRx.reload)</b>	<b>Bit 1 (TMRx.enable)</b>	<b>Action</b>
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.

**NOTES:**

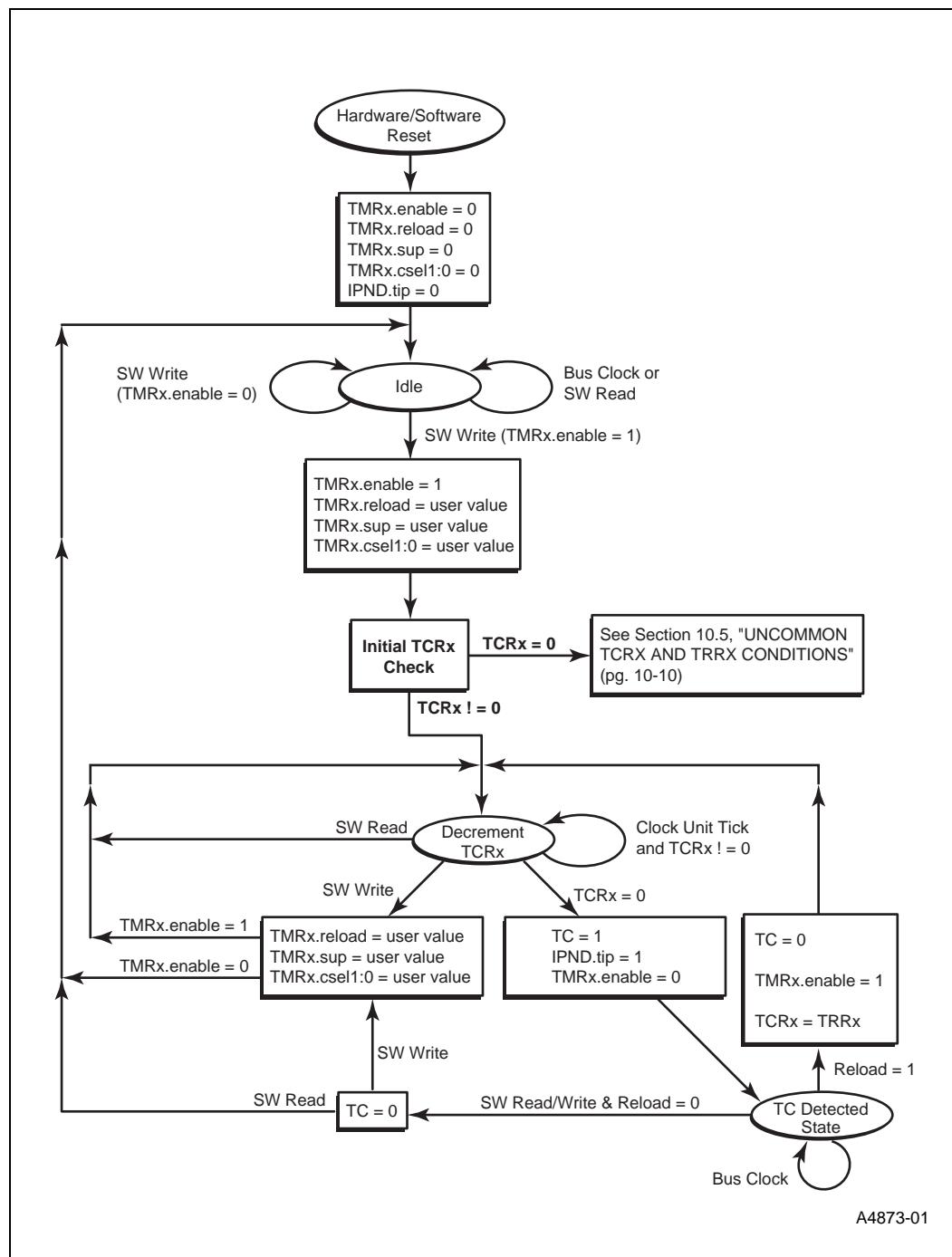
X = don't care

N = a number between 1H and FFFF FFFFH

## 10.6 Timer State Diagram

Figure 10-5 shows the common states of the Timer Unit. For uncommon conditions see Section 10.5, “Uncommon TCRX and TRRX Conditions” on page 10-10.

Figure 10-5. Timer Unit State Diagram



This chapter describes the i960® processor core architecture interrupt mechanism and the i960 Hx processor interrupt controller. Key topics include the i960 Hx processor's facilities for requesting and posting interrupts, the programmer's interface to the on-chip interrupt controller, latency and how to optimize interrupt performance.

## 11.1 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 Hx processor provides an on-chip programmable interrupt controller.

Requests for interrupt service come from many sources. These requests are prioritized so that instruction execution is redirected only if an interrupt request is of higher priority than that of the executing task. On the i960 Hx processor, interrupt requests may originate from external hardware sources, internal timer unit sources or from software. External interrupts are detected with the chip's 8-bit interrupt port and with a dedicated Non-Maskable Interrupt (NMI) input. 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. Integrated interrupt controller configuration and operation is described in [Section 11.7, “External Interface Description” on page 11-17](#).

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.

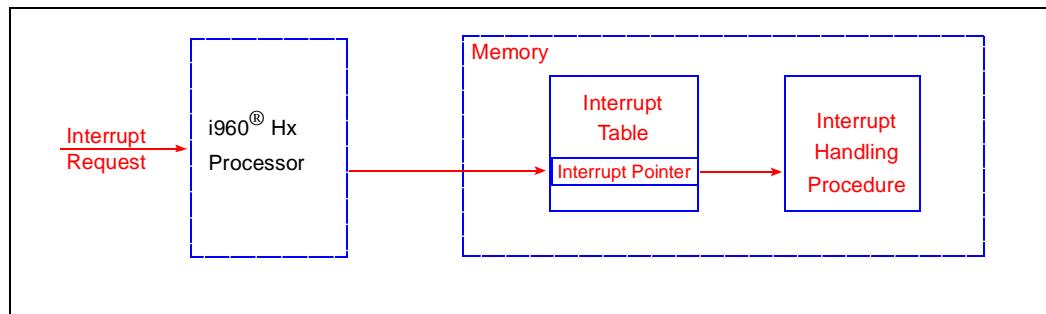
When the interrupt call is made, the processor uses a dedicated interrupt stack. The processor creates a new frame for the interrupt on this stack and a new set of local registers is allocated to the interrupt procedure. The interrupted program's current state is also saved.

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 being handled immediately. The mechanism for saving the interrupt is referred to as interrupt posting. Interrupt posting is described in [Section 11.6.5, “Posting Interrupts” on page 11-9](#).

The i960 core architecture defines two data structures to support interrupt processing: the interrupt table (see [Figure 11-1](#)) 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 allows the interrupt stack to be located in a different area of memory than the user and supervisor stack (e.g., fast SRAM).

**Figure 11-1. Interrupt Handling Data Structures**



### 11.1.1 The i960® Hx Processor Interrupt Controller

The i960 Hx 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.
- Support of up to 240 external sources.
- Eight external interrupt pins, one non-maskable interrupt pin, two internal timers sources for detection of hardware-requested interrupts.
- Edge or level detection on external interrupt pins.
- Debounce option on external interrupt pins.

The user program interfaces to the interrupt controller with six memory-mapped control registers. The interrupt control register (ICON) and interrupt map control registers (IMAP0-IMAP2) provide configuration information. The interrupt pending (IPND) register posts hardware-requested interrupts. The interrupt mask (IMSK) register selectively masks hardware-requested interrupts.

## 11.2 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.

## 11.3 Interrupt Priority

Each interrupt procedure pointer is eight bits in length, allowing up to 256 unique procedure pointers to be defined in principle. 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, 240 external interrupt sources and the non-maskable interrupt (NMI#) 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 if its priority is higher than the priority of the program or interrupt the processor is executing currently. If the interrupt priority is less than or equal to the processor's current priority, the processor does not service the request but rather posts it as a pending interrupt. See [Section 11.4.2, “Pending Interrupts” on page 11-5](#). When multiple interrupt requests are pending at the same priority level, the request with the highest vector number is serviced first.

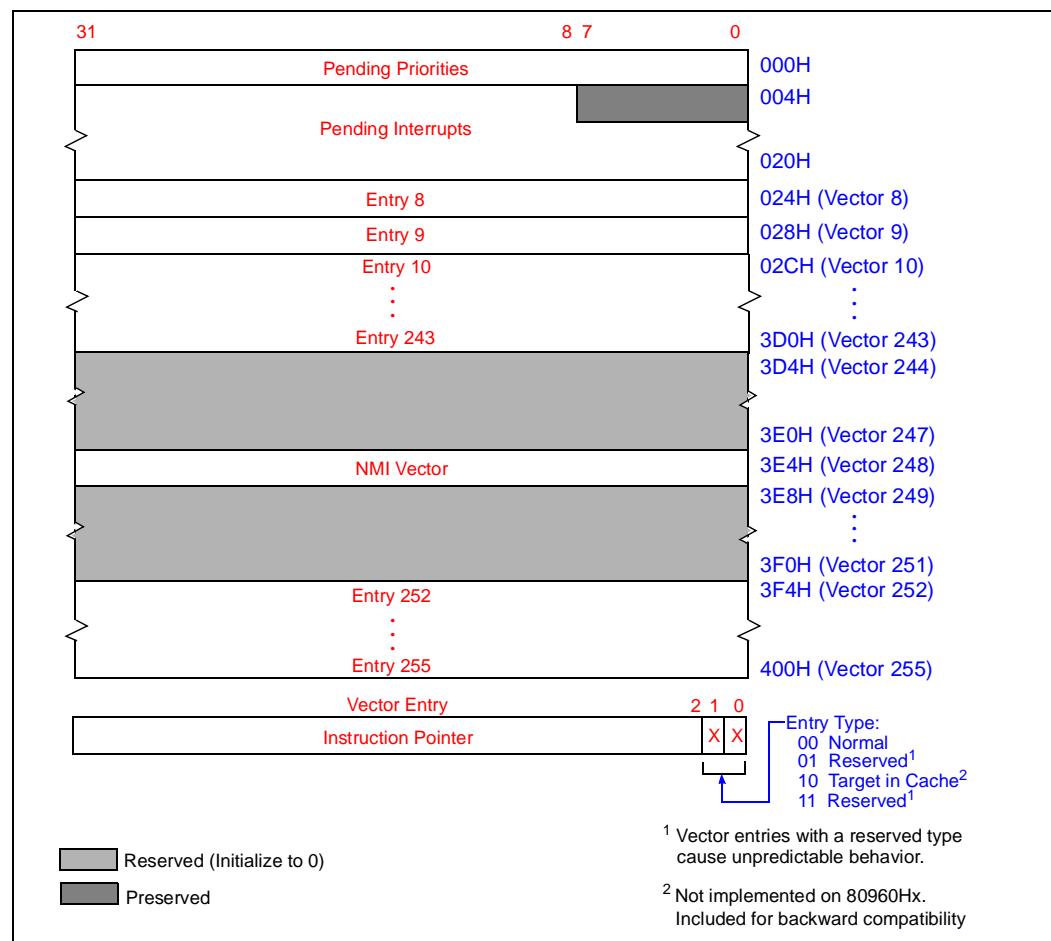
Priority-31 interrupts are handled as a special case. Even when the processor is executing at priority level 31, a priority-31 interrupt will interrupt the processor. On the i960 Hx processor, the non-maskable interrupt (NMI#) interrupts priority-31 execution; no interrupt can interrupt an NMI handler.

## 11.4 Interrupt Table

The interrupt table (see [Figure 11-2](#)) 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 since the processor must be able to read and write the table's pending interrupt section. Also, make sure the table's memory location is not protected by the GMU as this can also prevent the processor from reading and writing the table's pending interrupt section.

The interrupt table is divided into two sections: vector entries and pending interrupts. Each are described in the subsections that follow.

**Figure 11-2. Interrupt Table**



## 11.4.1 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 11-2](#). 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 000 indicates that the interrupt procedure should be fetched normally; type 010 indicates that the interrupt procedure should be fetched from the locked partition of the instruction cache. Refer to [Section 11.9.2.2, “Caching Interrupt Routines and Reserving Register Frames” on page 11-33](#). The other possible entry types are reserved and must not be used.

## 11.4.2 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. For example, if 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 represents 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 if there are any pending interrupts with a priority greater than the current program and then determine the vector number of the interrupt with the highest priority.

### 11.4.3 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 Hx 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.

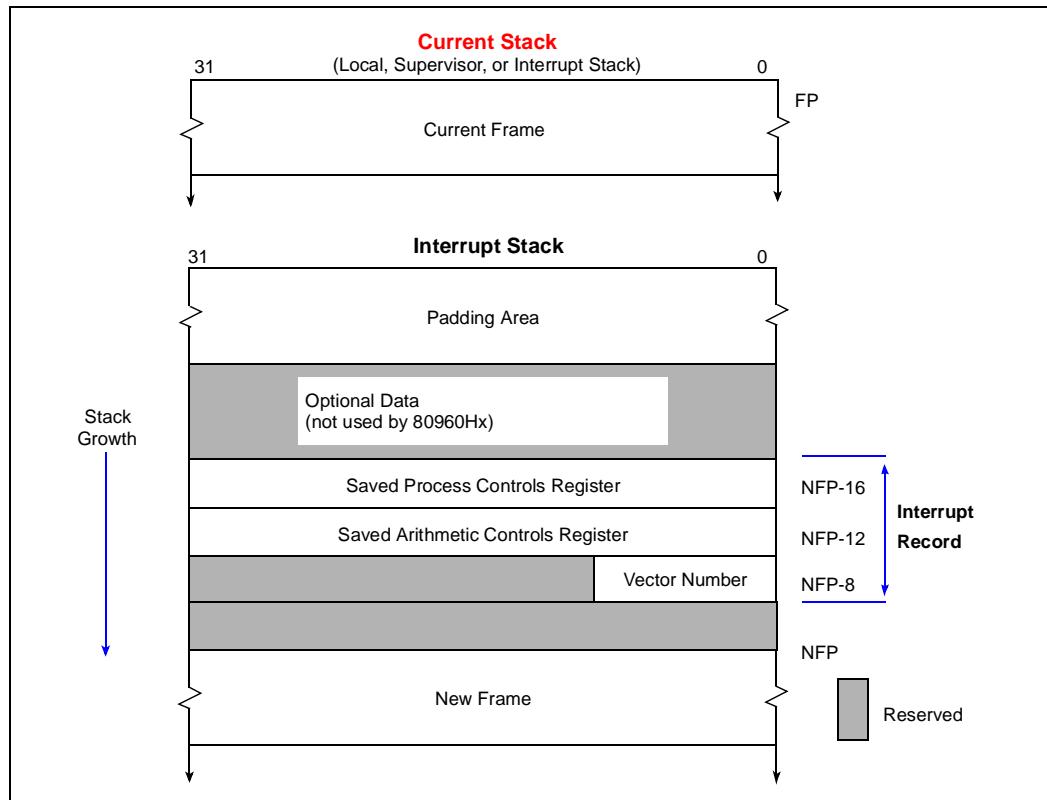
This caching mechanism is non-transparent; the processor may modify fields in a cached interrupt table without modifying the same fields in the interrupt table itself. Vector caching is described in Section 11.9.2.1, “Vector Caching Option” on page 11-32.

## 11.5 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 7-2. 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 11-3 shows the structure of this interrupt record.

**Figure 11-3. 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 Hx 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.

## 11.6 Managing Interrupt Requests

The i960 processor architecture provides a consistent interrupt model, as required for interrupt handler compatibility between various implementations of the i960 processor family. The architecture, however, leaves the interrupt request management strategy to the specific i960 processor family implementations. In the i960 Hx processor, a programmable on-chip interrupt controller manages all interrupt requests (Figure 11-12). These requests originate from:

- Eight-bit external interrupt pins XINT[7:0]#
- Two internal timer unit interrupts (TINT[1:0])
- Non-maskable interrupt pin (NMI#)
- **sysctl** instruction execution (software-initiated interrupts)

### 11.6.1 External Interrupts

External interrupt pins can be programmed to operate in three modes:

1. Dedicated mode: the pins may be individually mapped to interrupt vectors.
2. Expanded mode: the pins may be interpreted as a bit field which can request any of the 240 possible external interrupts that the i960 processor family supports.
3. Mixed mode: five pins operate in expanded mode and can request 32 different interrupts, and three pins operate in dedicated mode.

Dedicated-mode requests are posted in the Interrupt Pending Register (IPND). The processor's ICU does not post expanded-mode requests.

### 11.6.2 Non-Maskable Interrupt (NMI#)

The NMI# pin generates an interrupt for implementation of critical interrupt routines. 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. Note that a return-from-interrupt in user mode does not unblock NMI events and should be avoided by software. An interrupt request on the NMI# pin is always falling-edge detected.

### 11.6.3 Timer Interrupts

Each of the two timer units has an associated interrupt to allow the application to accept or post the interrupt request. Timer unit interrupt requests are always handled as dedicated-mode interrupt requests.

### 11.6.4 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 can request vector 248 (NMI#) as a software interrupt; however, the interrupt vector will be read from the interrupt table, not from the internal vector cache.

### 11.6.5 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 Hx processor.
- External Interrupts: interrupts posted through the interrupt table, by an external agent to the i960 Hx processor.
- Hardware interrupts: interrupts posted directly to the i960 Hx processor through an implementation-dependent mechanism that may avoid using the interrupt table.

#### 11.6.5.1 Posting Software Interrupts via **sysctl**

In the i960 Hx processor, **sysctl** is typically used to request an interrupt in a program (see [Example](#)). 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 6-108](#) for a complete discussion of **sysctl**.

##### Example 11-1. 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.

If 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 updates this register's value only when **sysctl** requests an interrupt or when a software-generated interrupt is serviced.

### 11.6.5.2 Posting Software Interrupts Directly in the Interrupt Table

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 and is not reliable.

### 11.6.5.3 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 11-2. 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 operation span multiple instructions.

#### 11.6.5.4 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.

### 11.6.6 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, if 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, which causes control to be transferred to the corresponding interrupt handler.

#### Example 11-3. 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; }
    }
}
```

#### 11.6.7 Sampling Pending Interrupts in the Interrupt Table

At specific points, the processor checks the interrupt table for pending interrupts. If one is found, it is handled as if the interrupt occurred at that time. In the i960 Hx 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 algorithm shown in [Example 11-4](#) 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 11-4. Sampling 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 if 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.

## 11.6.8 Interrupt Controller Modes

The eight external interrupt pins can be configured for one of three modes: dedicated, expanded or mixed. Each mode is described in the subsections that follow.

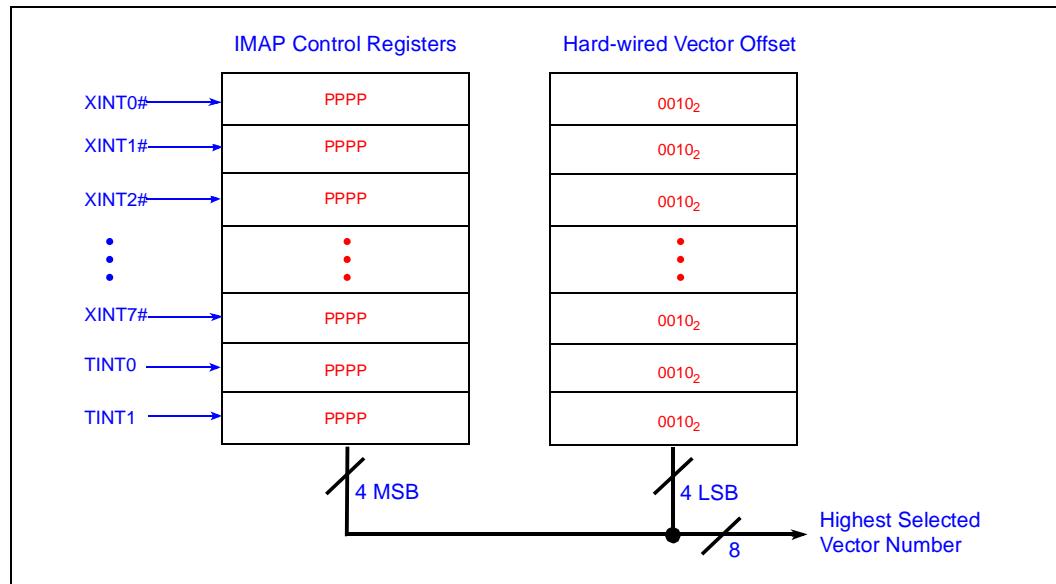
### 11.6.8.1 Dedicated Mode

In dedicated mode, 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 11-4), 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.)

Dedicated-mode 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 dedicated-mode interrupts. Optionally, the IMSK register can be saved and cleared when a dedicated interrupt is serviced. This allows other hardware-generated interrupts to be locked out until the mask is restored. See [Section 11.7.3, “Memory-Mapped Control Registers” on page 11-19](#) 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 dedicated-mode vectors. The timer interrupts are always dedicated-mode interrupts.

**Figure 11-4. Dedicated Mode**



### 11.6.8.2 Expanded Mode

In expanded mode, up to 240 interrupts can be requested from external sources. Multiple external sources are externally encoded into the 8-bit interrupt vector number. This vector number is then applied to the external interrupt pins (Figure 11-5), with the XINT0# pin representing the least-significant bit and XINT7# the most significant bit of the number. Note that external interrupt pins are active low; therefore, the inverse of the vector number is actually applied to the pins.

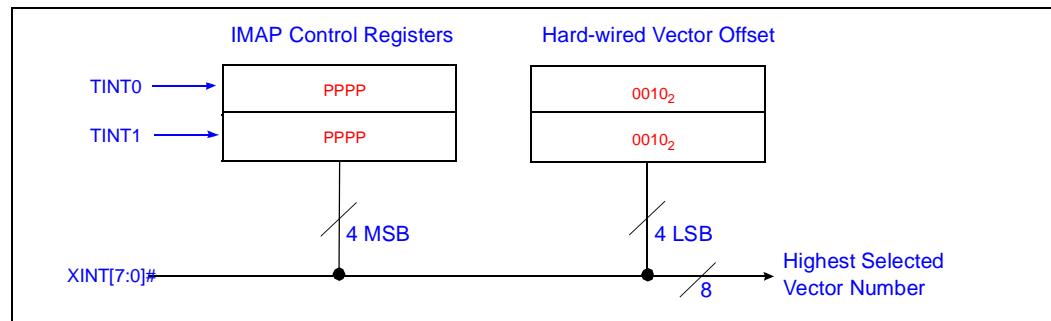
In expanded mode, external logic is responsible for posting and prioritizing external sources. Typically, this scheme is implemented with a simple configuration of external priority encoders. The interrupt source must remain asserted until the processor services the interrupt and explicitly clears the source. As shown in Figure 11-6, simple, combinational logic can handle prioritization of the external sources when more than one expanded mode interrupt is pending.

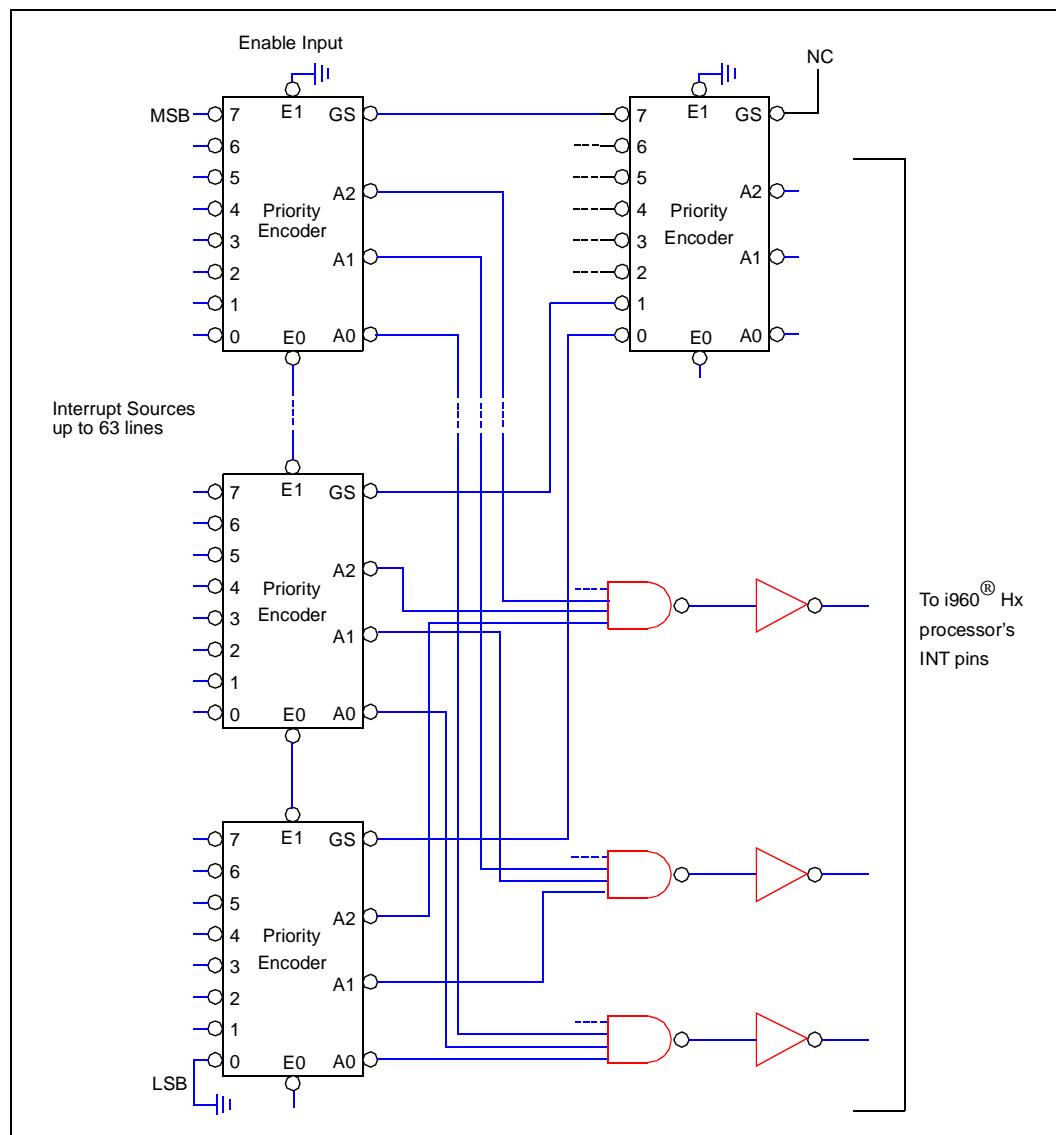
An expanded mode interrupt source must remain asserted until the processor services the interrupt and explicitly clears the source. External-interrupt pins in expanded mode are always active low and level-detect. The interrupt controller ignores vector numbers 0 through 7. The output of the external priority encoders in Figure 11-6 can use the 0 vector to indicate that no external interrupts are pending.

The low-order four bits of IMAP0 buffer the expanded-mode interrupt internally. XINT[7:4]# are placed in IMAP0[3:0]; XINT[3:0]# are latched in a special register for use in further arbitrating the interrupt and in selecting the interrupt handler.

IMSK register bit 0 provides a global mask for all expanded interrupts. The remaining bits (1-7) must be set to 0 in expanded mode. Optionally, the mask bit can be saved and cleared when an expanded mode interrupt is serviced. This allows other hardware-requested interrupts to be locked out until the mask is restored. IPND register bits 0-7 have no function in expanded mode, since external logic is responsible for posting interrupts.

**Figure 11-5. Expanded Mode**



**Figure 11-6. Implementation of Expanded Mode Sources**


### 11.6.8.3 Mixed Mode

In mixed mode, pins XINT0# through XINT4# are configured for expanded mode. These pins are encoded for the five most-significant bits of an expanded-mode vector number; the three least-significant bits of the vector number are set internally to  $010_2$ . Pins XINT5# through XINT7# are configured for dedicated mode.

The low-order four bits of IMAP0 are used to buffer the expanded-mode interrupt internally. XINT[4:1]# are placed in IMAP0[3:0]; XINT0# is latched in a special register for use in further arbitrating the interrupt and in selecting the interrupt handler.

IMSK register bit 0 is a global mask for the expanded-mode interrupts; bits 5 through 7 mask the dedicated interrupts from pins XINT5# through XINT7#, respectively. IMSK register bits 1-4 must be set to 0 in mixed mode. The IPND register posts interrupts from the dedicated-mode pins XINT[7:5]#. IPND register bits that correspond to expanded-mode inputs are not used.

### 11.6.9 Saving the Interrupt Mask

Whenever an interrupt requested by XINT[7:0]# 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 allows all interrupts except NMI#s to be masked while an interrupt is being 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 11.7.4, “Interrupt Control Register \(ICON\) — SF3” on page 11-20](#). Several options are provided for interrupt mask handling:

- Mask unchanged
- Cleared for dedicated-mode sources only
- Cleared for expanded-mode sources only
- Cleared for all hardware-requested interrupts (dedicated and expanded mode)

The second and third options are used in mixed mode, where both dedicated-mode and expanded-mode inputs are allowed. Timer unit interrupts are always dedicated-mode interrupts.

Note that when the same interrupt is requested simultaneously by a dedicated- and an expanded-mode source, the interrupt is considered an expanded-mode interrupt and the IMSK register is handled accordingly.

The IMSK register must be saved and cleared when expanded mode inputs request a priority-31 interrupt. Priority-31 interrupts are interrupted by other priority-31 interrupts. In expanded mode, the interrupt pins are level-activated. For level-activated interrupt inputs, instructions within the interrupt handler are typically responsible for causing the source to deactivate. When 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.

## 11.7 External Interface Description

This section describes the physical characteristics of the interrupt inputs. The i960 Hx processor provides eight external interrupt pins and one non-maskable interrupt pin for detecting external interrupt requests. The eight external pins can be configured as dedicated inputs, where each pin is capable of requesting a single interrupt. The external pins can also be configured in an expanded mode, where the value asserted on the external pins represents an interrupt vector number. In this mode, up to 240 values can be directly requested with the interrupt pins. The external interrupt pins can be configured in mixed mode. In this mode, some pins are dedicated inputs and the remaining pins are used in expanded mode.

### 11.7.1 Pin Descriptions

The interrupt controller provides nine interrupt pins:

- |            |  |
|------------|--|
| XINT[7:0]# | External Interrupt (input) - These eight pins cause interrupts to be requested. Pins are software configurable for three modes: dedicated, expanded, mixed. Each pin can be programmed as an edge- or level-detect input. Also, a debounce sampling mode for these pins can be selected under program control. |
| NMI#       | Non-Maskable Interrupt (input) - This edge-activated pin causes a non-maskable interrupt event to occur. NMI# is the highest priority interrupt recognized. A debounce sampling mode for NMI# can be selected under program control. This pin is internally synchronized.                                      |

External interrupt pin functions XINT[7:0]# depend on the operation mode (expanded, dedicated or mixed) and on several other options selected by setting ICON register bits.

## 11.7.2 Interrupt Detection Options

The XINT[7:0]# pins can be programmed for level-low or falling-edge detection when used as dedicated inputs. All dedicated inputs plus the NMI# pin are programmed (globally) for fast sampling or debounce sampling. Expanded-mode inputs are always sampled in debounce mode. Pin detection and sampling options are selected by programming the ICON register.

When falling-edge detection is enabled and a high-to-low transition is detected, the processor sets the corresponding pending bit in the IPND register. The processor clears the IPND bit upon entry into the interrupt handler.

When a pin is programmed 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 11-5** demonstrates how a level detect interrupt is typically handled. The example assumes that the **Id** from address “timer\_0,” deactivates the interrupt input.

### Example 11-5. Return from a Level-detect Interrupt

```

# Clear level-detect interrupts before return from handler
ld INTR_SRC, g0    # Dismiss the extern. interrupt
lda IPND_MMR, g1   # g1 = IPND MMR address
lda 0x80, g2        # g2 = mask to clear XINT7 IPND bit
# Loop until IPND bit 7 clears
wait:
    mov    0,g3
    # Try to clear the XINT7 IPND bit
    atmod g1, g2, g3
    bbs   0x7, g3, wait  # Branch until IPND bit 7 clears
# Optionally restore IMSK
    mov    r3, IMSK
    ret                 # Return from handler

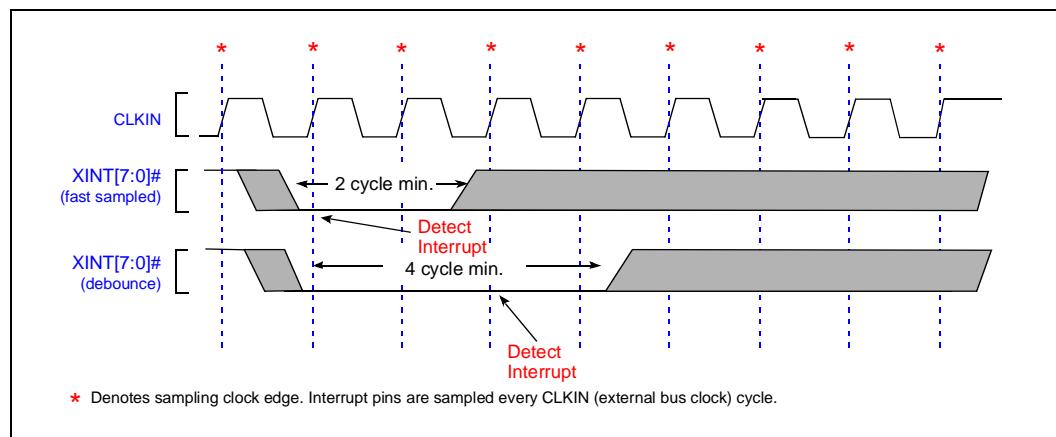
```

The debounce sampling mode provides a built-in filter for noisy or slow-falling inputs. The debounce sampling mode requires that a low level is stable for three consecutive cycles before the expanded mode vector is resolved internally. Expanded mode interrupts are always sampled using the debounce sampling mode. This allows for skew time between changing outputs of external priority encoders.

Figure 11-7 shows how a signal is sampled in each mode. The debounce-sampling option adds several clock cycles to an interrupt's latency due to the multiple clocks of sampling. Inputs are sampled once every CLKIN cycle (external bus clock).

Interrupt pins are asynchronous inputs. Setup or hold times relative to CLKIN are not needed to ensure proper pin detection. Note in Figure 11-7 that interrupt inputs are sampled every CLKIN cycle. For practical purposes, this means that asynchronous interrupting devices must generate an interrupt signal that is asserted for at least two CLKIN cycles for the fast sampling mode or four CLKIN cycles for the debounce sampling mode. See the *80960HA/HD/HT Embedded 32-bit Microprocessor* datasheet for setup and hold specifications that guarantee detection of the interrupt on particular edges of CLKIN. These specification are useful in designs that use synchronous logic to generate interrupt signals to the processor. These specification must also be used to calculate the minimum signal width, as shown in Figure 11-7.

**Figure 11-7. Interrupt Sampling**



### 11.7.3 Memory-Mapped Control Registers

The programmer's interface to the interrupt controller is through six memory-mapped control registers: ICON control register, IMAP0-IMAP2 control registers, IMSK register and IPND control register. Table 11-1 describes the ICU registers. ICON, IMSK and IPND are also accessible as special function registers.

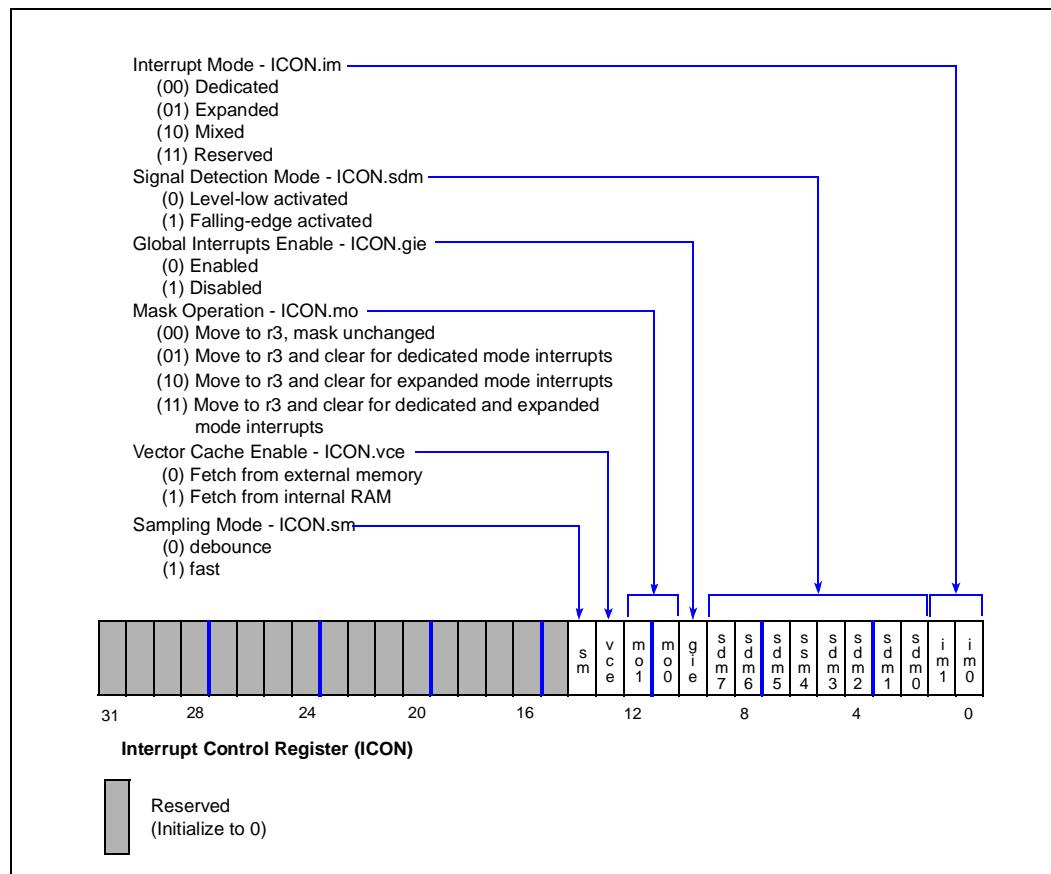
**Table 11-1. Interrupt Control Registers Memory-Mapped Addresses**

Register Name	Description	Address
IPND (sf0)	Interrupt Pending Register	FF00 8500H
IMSK (sf1)	Interrupt Mask Register	FF00 8504H
ICON (sf3)	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

## 11.7.4 Interrupt Control Register (ICON) — SF3

The ICON register (see Figure 11-8) 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.

**Figure 11-8. Interrupt Control (ICON) Register**



The *interrupt mode* field (bits 0 and 1) determines the operation mode for the external interrupt pins (XINT[7:0]#), dedicated, expanded or mixed.

The *signal detection mode bits* (bits 2 - 9) determine whether the signals on the individual external interrupt pins (XINT[7:0]#) are level-low activated or falling-edge activated. Expanded-mode inputs are always level-detected; the NMI# input is always edge-detected, regardless of the bit's value.

The *global interrupts enable bit* (bit 10) 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 mask register. The global interrupts enable bit is also changed indirectly by the use of the following instructions: **inten**, **intdis**, **intctl**.

The *mask-operation field* (bits 11, 12) determines the operation the core performs on the mask register when a hardware-generated interrupt is serviced. On an interrupt, the IMSK register is either unchanged; cleared for dedicated-mode interrupts; cleared for expanded-mode interrupts; or cleared for both dedicated- and expanded-mode interrupts. IMSK is never cleared for NMI or software interrupts.

The *vector cache enable bit* (bit 13) determines whether interrupt table vector entries are fetched from the interrupt table or from internal data RAM. Only vectors with the four least-significant bits equal to  $0010_2$  may be cached in internal data RAM.

The *sampling-mode bit* (bit 14) determines whether dedicated inputs and NMI# pin are sampled using debounce sampling or fast sampling. Expanded-mode inputs are always detected using debounce mode.

Bits 15 through 31 are reserved and must be set to 0 at initialization.

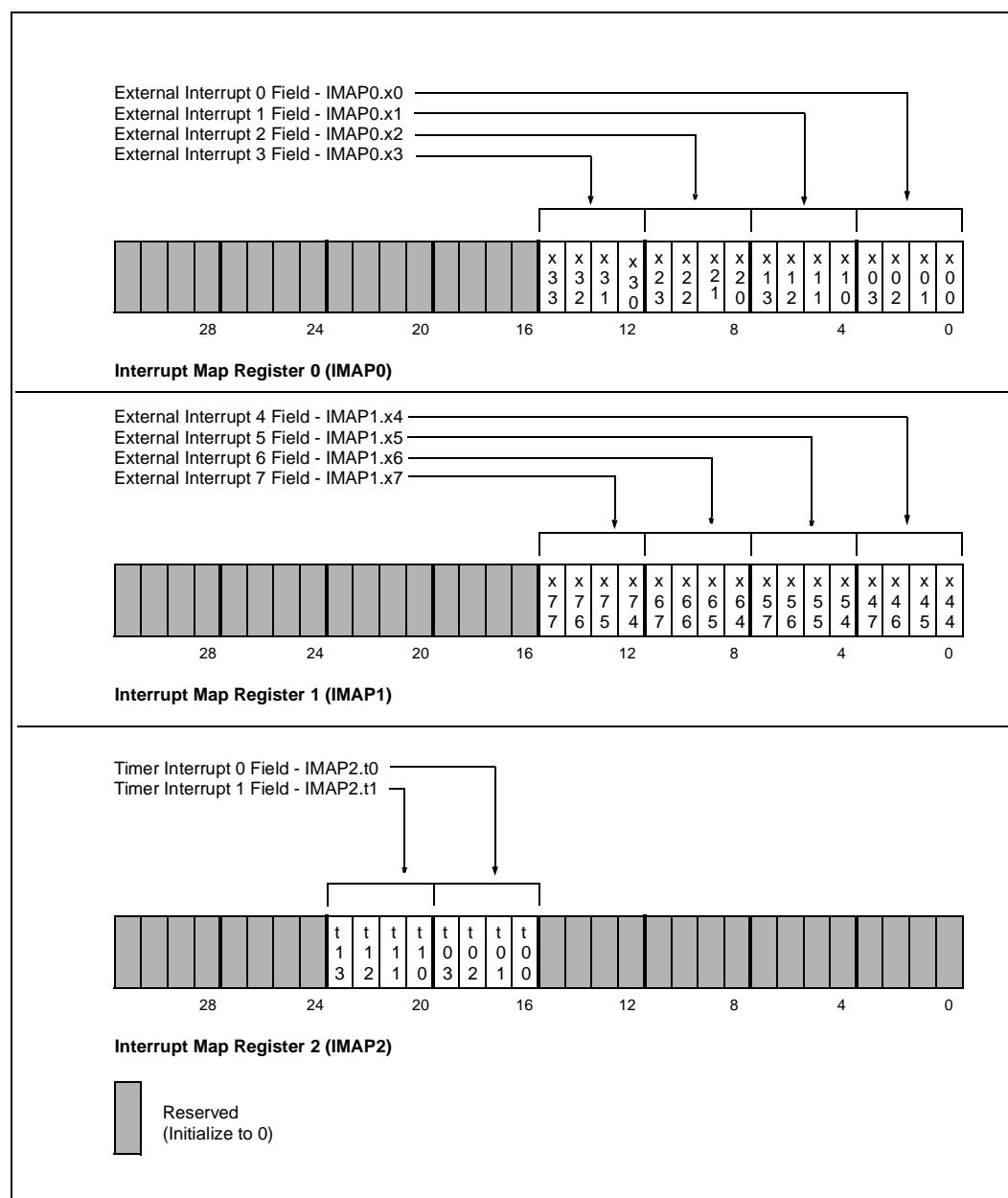
### 11.7.5 Interrupt Mapping Registers (IMAP0-IMAP2)

The IMAP registers (Figure 11-9) are three 32-bit registers (IMAP0 through IMAP2). These registers are used to program the vector number associated with the interrupt source when the source is connected to a dedicated-mode input. IMAP0 and IMAP1 contain mapping information for the external interrupt pins (four bits per pin). IMAP2 contains mapping information for the timer-interrupt inputs (four bits per interrupt).

Each set of four bits contains a vector number's four most-significant bits; the four least-significant bits are always  $0010_2$ . In other words, each source can be programmed for a vector number of PPPP  $0010_2$ , where “P” indicates a programmable bit. For example, IMAP0 bits 4 through 7 contain mapping information for the XINT1# pin. If these bits are set to  $0110_2$ , the pin is mapped to vector number  $0110\ 0010_2$  (or vector number 98).

Software can access the mapping registers using load/store type instructions. The mapping registers are also automatically loaded at initialization from the control table in external memory. Note that bits 16 through 31 of IMAP0 and IMAP1 are reserved and should be cleared to 0 at initialization. Bits 0-15 and 24-31 of IMAP2 are also reserved and should be cleared to 0.

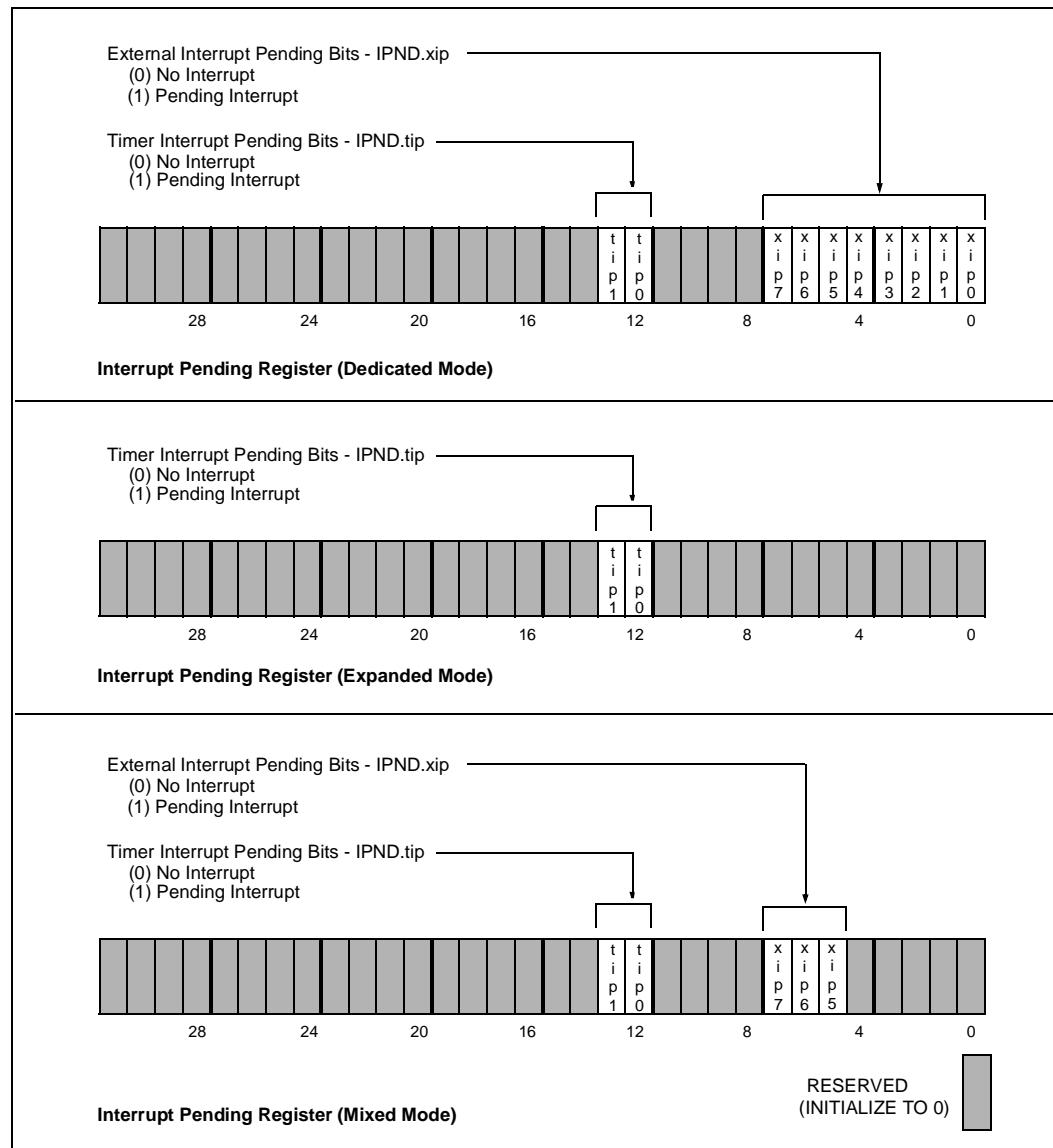
**Figure 11-9. Interrupt Mapping (IMAP0-IMAP2) Registers**



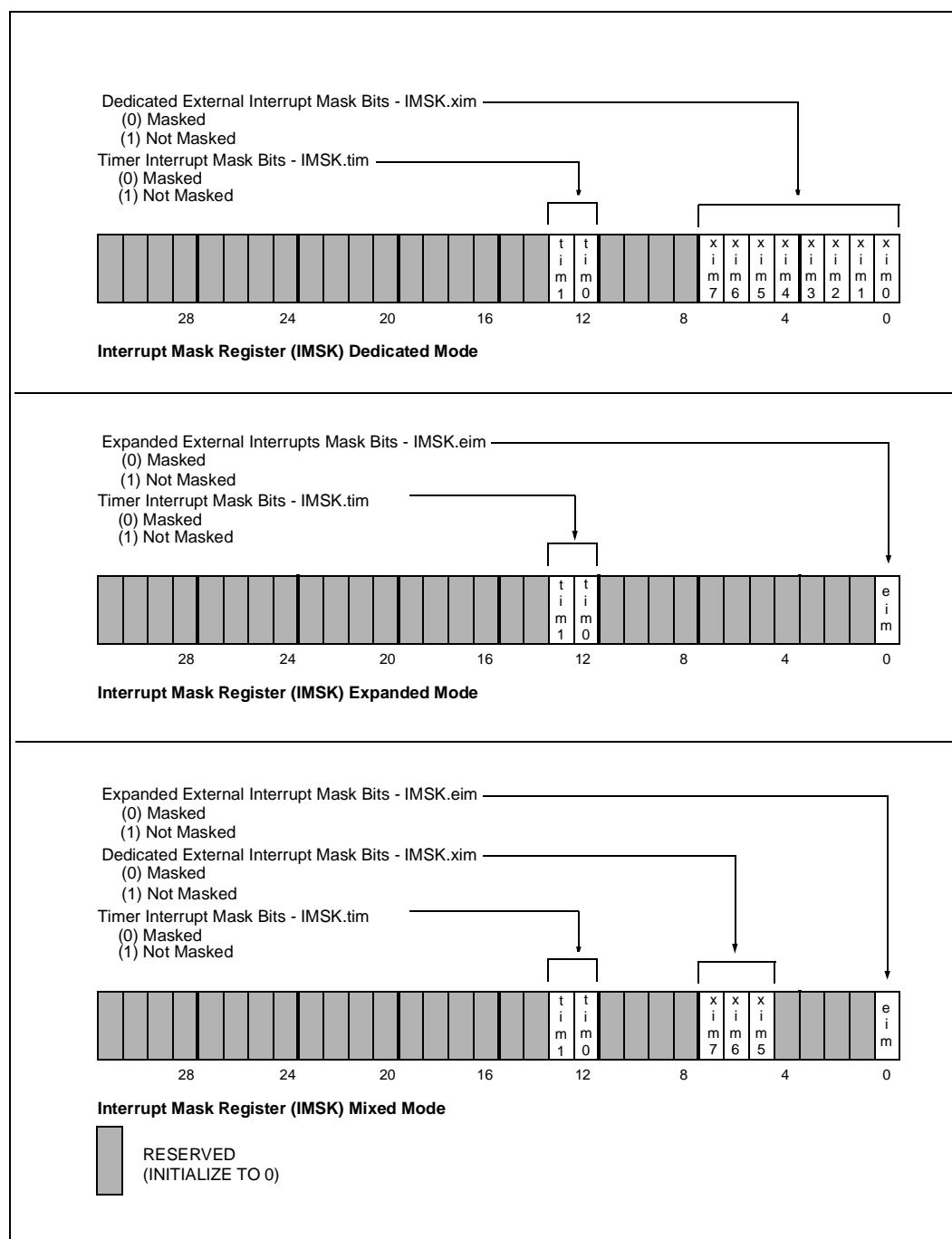
### 11.7.5.1 Interrupt Mask (IMSK; SF1) and Interrupt Pending (IPND; SF0) Registers

The IMSK and IPND registers (see Figure and Figure 11-11) are both memory-mapped registers. Bits 0 through 7 of these registers are associated with the external interrupt pins (XINT0# through XINT7#) and bits 12 and 13 are associated with the timer-interrupt inputs (TMR0 and TMR1). All other bits are reserved and should be set to 0 at initialization.

**Figure 11-10. Interrupt Pending (IPND) Register**



**Figure 11-11. Interrupt Mask (IMSK) Registers**



The IPND register posts dedicated-mode interrupts originating from the eight external dedicated sources (when configured in dedicated mode) and the two timer sources. Asserting one of these inputs causes a 1 to be latched into its associated bit in the IPND register. In expanded mode, bits 0 through 7 of this register are not used and should not be modified; in mixed mode, bits 0 through 4 are not used and should not be modified.

The mask register provides a mechanism for masking individual bits in the IPND register. An interrupt source is disabled if its associated mask bit is set to 0.

Mask register bit 0 has two functions: it masks interrupt pin XINT0# in dedicated mode and it masks all expanded-mode interrupts globally in expanded and mixed modes. In expanded mode, bits 1 through 7 are not used and should contain zeros only; in mixed mode, bits 1 through 4 are not used and should contain zeros only.

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.

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. The bit set in the IPND register must be associated with an interrupt source that is programmed for dedicated-mode operation.

### 11.7.5.2 Interrupt Controller Register Access Requirements

Like all other load accesses from internal memory-mapped registers, once issued, a load instruction that accesses an interrupt 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 new **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.

### 11.7.5.3 Default and Reset Register Values

The ICON and IMAP[2:0] control registers are loaded from the control table in external memory when the processor is initialized or reinitialized. The control table is described in [Section 13.3.3, “Control Table” on page 13-22](#). The IMSK register is set to 0 when the processor is initialized (RESET# is deasserted). The IPND register value is undefined after a power-up initialization (cold reset). The application is responsible for clearing this register before any mask register bits are set; otherwise, unwanted interrupts may be triggered. The pending register value is retained for a reset while power is on (warm reset).

## 11.8 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 Hx 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 8-bit external interrupt port (XINT[7:0]#), 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:

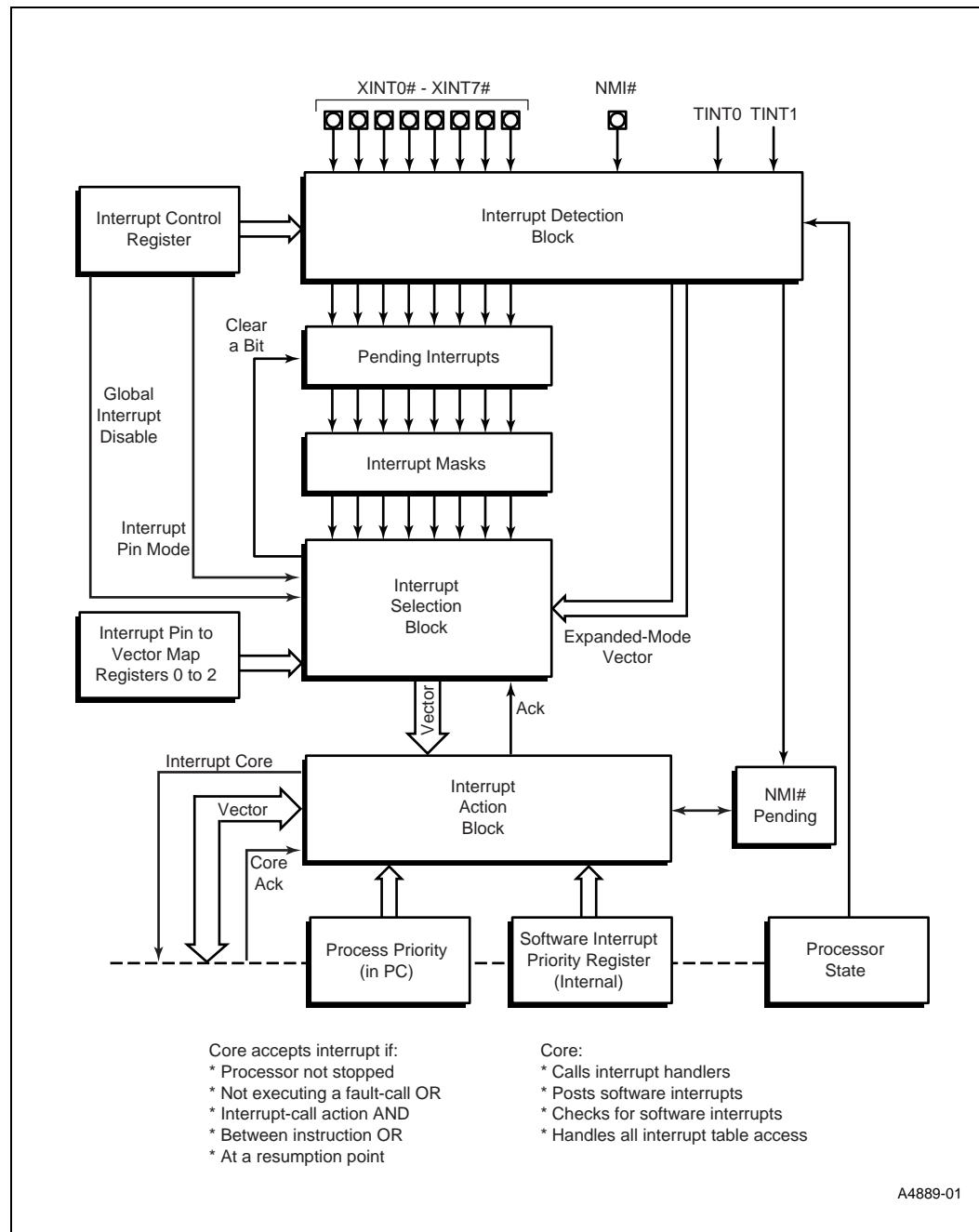
- Hardware interrupts are posted by setting the interrupt's assigned bit in the interrupt pending (IPND) memory mapped register
- Software interrupts are posted by setting the interrupt's assigned bit in the interrupt table's pending priorities and pending interrupts fields

**Checking Pending Interrupts** — The interrupt controller compares each pending interrupt's priority with the current process priority. If 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** — If 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 11-12** illustrates interrupt controller function. For best performance, the interrupt flow for hardware interrupt sources is implemented entirely in hardware.

The comparator signals the core only 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.

**Figure 11-12. Interrupt Controller**


## 11.8.1 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 11-6** sets up the interrupt controller for expanded-mode operation. Initially the IMSK register is masked to allow for setup. A value that selects expanded-mode operation is loaded into the ICON register and the IMSK is unmasked.

### Example 11-6. Programming the Interrupt Controller for Expanded Mode

```
# Example expanded mode setup . .
mov    0, g0
mov    1, g1
st     g0,IMSK           # mask, IMSK MMR at 0xFF008504
st     g1,ICON
st     g1,IMSK           # unmask expanded interrupts
```

## 11.8.2 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 being 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, special function 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 11.9.2.2, “Caching Interrupt Routines and Reserving Register Frames” on page 11-33](#) for a complete description.

### 11.8.3 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 that the processor uses to service an interrupt depends on the processor state when the interrupt is received. If the processor is executing a background task when an interrupt request is posted, the interrupt context switch must change stacks to the interrupt stack. This is called an executing-state interrupt. If the processor is already executing an interrupt handler, no stack switch is required since the interrupt stack is already in use. This is called an interrupted-state interrupt.

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.

#### 11.8.3.1 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 (as shown in [Figure 11-3](#)). 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. Clearing this bit allows the interrupt to be handled without trace faults being raised.
6. Sets the frame return status field pfp2:0 to  $111_2$ .
7. Performs a call operation as described in [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 then 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 [Chapter 7, “Procedure Calls”](#).
4. Resumes work on the program, if there are no pending interrupts to be serviced or trace faults to be handled.

### 11.8.3.2 Servicing an Interrupt from Interrupted State

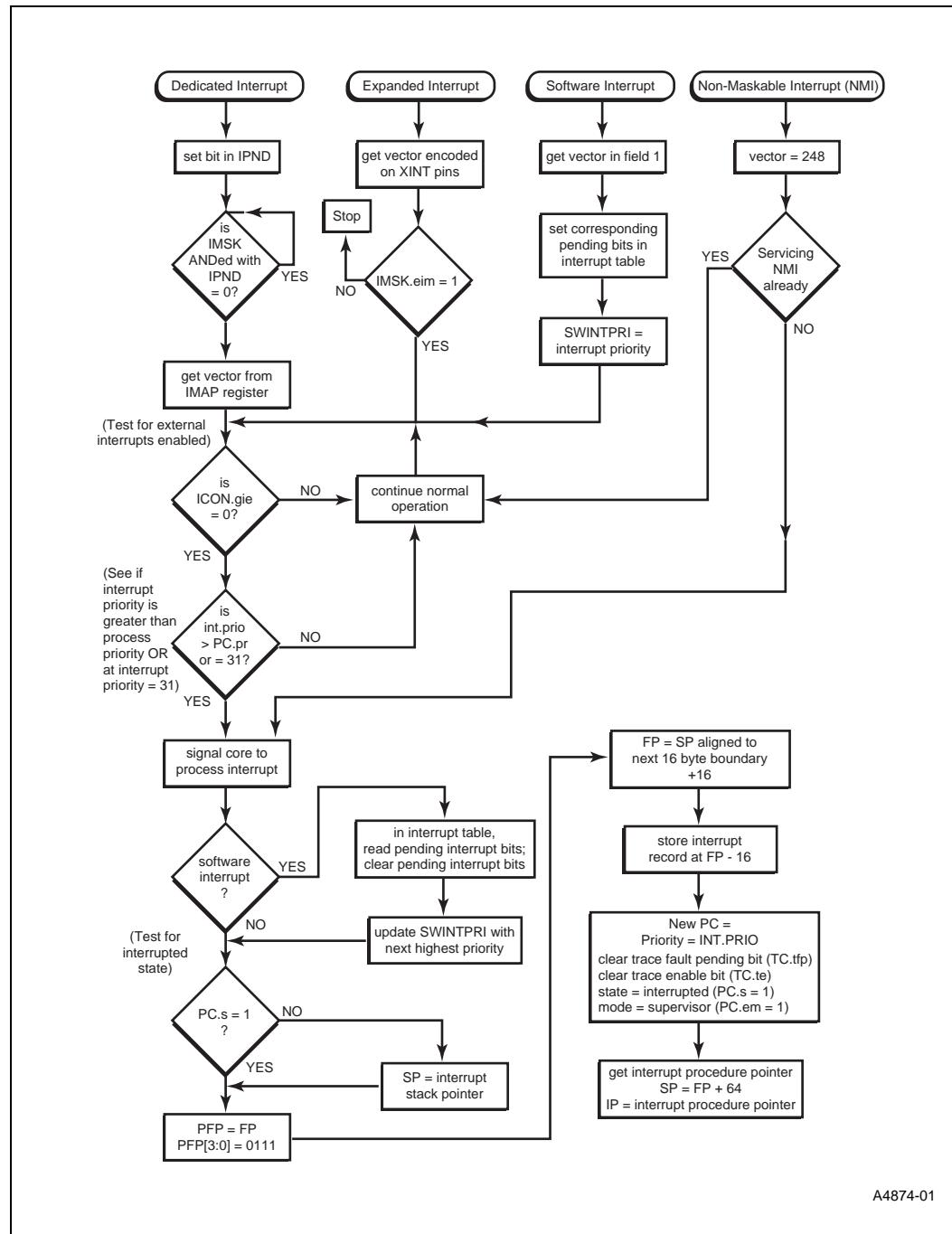
If the processor receives an interrupt while it is servicing another interrupt, and the new interrupt has a higher priority than the interrupt currently being serviced, the current interrupt-handler routine is interrupted. Here, the processor performs the same interrupt-servicing action as is described in [Section 11.8.3.1, “Servicing an Interrupt from Executing State” on page 11-29](#) 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 11-3](#).

On the return from the current interrupt handler to the previous interrupt handler, the processor deallocates the current stack frame and interrupt record, and stays on the interrupt stack.

## 11.9 Optimizing Interrupt Performance

Figure 11-13 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 11-13. Interrupt Service Flowchart**



A4874-01

### 11.9.1 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 activation of an interrupt source 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, and reflects differences between the 80960HA and the 80960HD/HT due to the 80960HD/HT processor's clock-multiplied core.

### 11.9.2 Features to Improve Interrupt Performance

The i960 Hx processor implementation 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

#### 11.9.2.1 Vector Caching Option

To reduce interrupt latency, the i960 Hx processors allow some interrupt table vector entries to be cached 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. The vectors that can be cached coincide with the vector numbers that are 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 11-2](#) shows the required vector mapping to specific locations in internal RAM. For example, the vector entry for vector number 18 must be stored at RAM location 04H, and so on.

The NMI# vector is also shown in [Table 11-2](#). 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 11-2. 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

### 11.9.2.2 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 user can reduce this fetch time by caching interrupt procedures or portions of procedures in the i960 Hx processor's instruction cache. The **icctl** instruction can load and lock these procedures into the instruction cache. See [Section 4.4, “Instruction Cache” on page 4-4](#) for information on the 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 4-3](#), for more details.

### 11.9.2.3 Caching the Interrupt Stack

By locating the interrupt stack in cacheable memory, the performance of interrupt returns can be improved. This is because potentially accesses to the interrupt record by the interrupt return can be satisfied by the data cache. See [Section 14.4, “Programming the Logical Memory Attributes” on page 14-11](#) for details on how to enable data caching for portions of memory.

### 11.9.3 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 11-3](#) shows the base latencies for all interrupt types, with varying pin sampling and vector caching options.

**Table 11-3. Base Interrupt Latency**

Interrupt Type	Detection Option	Vector Caching Enabled	Typical 80960Hx Latency (Bus Clocks)		
			HA	HD	HT
NMI#	Fast	Yes	21	10.5	7.3
	Debounced	Yes	23	12.5	9.3
Dedicated Mode XINT[7:0]#, TINT[1:0]	Fast	Yes	27	13	9.3
		No	29+a	15+d	10.3+g
	Debounced	Yes	28	15	11
		No	31+a	17.5+d	12+g
Expanded Mode XINT[7:0]#, TINT[1:0]	Debounced	Yes	30	16	11.3
		No	33+a	17.5+d	12.3+g
Software	NA	Yes	54+2b	27+2e	18+2h
		No	54+2b+c	27+2e+f	18+2h+g

**NOTES:**

a = MAX (0, N-6)

d= MAX (0, N-3)

g= MAX (0, N-2)

b = MAX (0, N-1)

e= MAX (0, N-0.5)

h= MAX (0, N-0.3)

c = MAX (0, N-7)

f= MAX (0, N-3.5)

where “N” is the number of bus cycles needed to perform a word load.

## 11.9.4 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 11-3](#), except for instruction execution. It is also assumed that the instructions are already in the cache and that tracing is disabled.

**Table 11-4. Worst-Case Interrupt Latency Controlled by divo to Destination r15**

Interrupt Type	Detection Option	Vector Caching Enabled	Worst 80960Hx Latency (Bus Clocks)		
			HA	HD	HT
NMI#	Fast	Yes	40	20	13.7
	Debounced	Yes	42	22	15.3
Dedicated Mode XINT[7:0]#, TINT[1:0]	Fast	Yes	42	21.5	14
		No	42+b	21.5+d	14+e
	Debounced	Yes	45	23.5	16
		No	45+b	23.5+d	16+e
Expanded Mode XINT[7:0]#, TINT[1:0]	Debounced	Yes	47	24.5	17.3
		No	47+b	24.5+d	17.3+e

**NOTES:**

a = MAX (0, N - 12)

c = MAX (0, N - 4.5)

e = MAX (0, N - 5.3)

b = MAX (0, N - 15)

d = MAX (0, N - 7.5)

f = MAX (0, N - 3.7)

g = MAX (0, N - 2)

where "N" is the number of bus cycles needed to perform a word load.

**Table 11-5. Worst-Case Interrupt Latency Controlled by divo to Destination r3**

Interrupt Type	Detection Option	Vector Caching Enabled	Worst 80960Hx Latency (Bus Clocks)		
			HA	HD	HT
NMI#	Fast	Yes	54	27	18.3
	Debounced	Yes	56	29	20
Dedicated Mode XINT[7:0]#, TINT[1:0]	Fast	Yes	59	30	20.3
		No	62+a	31.5+d	21.3+g
	Debounced	Yes	61	32	22
		No	64+a	33.5+d	23+g
Expanded Mode XINT[7:0]#, TINT[1:0]	Debounced	Yes	64	33	23
		No	67+a	34.5+d	24+g
Software	NA	Yes	70+2b	35+2e	23.3+2h
		No	70+2b+c	35+2e+f	23.3+2h+i

**NOTES:**

$a = \text{MAX}(0, N - 6)$        $d = \text{MAX}(0, N - 3)$        $g = \text{MAX}(0, N - 2)$   
 $b = \text{MAX}(0, N - 1)$        $e = \text{MAX}(0, N - 0.5)$        $h = \text{MAX}(0, N - 0.7)$   
 $c = \text{MAX}(0, N - 7)$        $f = \text{MAX}(0, N - 3.5)$        $i = \text{MAX}(0, N - 2.3)$

where "N" is the number of bus cycles needed to perform a word load.

**Table 11-6. Worst-Case Interrupt Latency Controlled by calls**

Interrupt Type	Detection Option	Vector Caching Enabled	Worst 80960Hx Latency (Bus Clocks)		
			HA	HD	HT
NMI#	Fast	Yes	55+b	27.5+d	18.3+f
	Debounced	Yes	58+b	29.5+d	20.7+f
Dedicated Mode XINT[7:0]#, TINT[1:0]	Fast	Yes	60+b	30.5+d	20.3+f
		No	63+a+b	32+c+d	21.3+e+f
	Debounced	Yes	62+b	32.5+d	22.3+f
		No	65+a+b	34+c+d	23.3+e+f
Expanded Mode XINT[7:0]#, TINT[1:0]	Debounced	Yes	65+b	33.5+d	23.3+f
		No	68+a+b	35+c+d	24.3+e+f

**NOTES:**

$a = \text{MAX}(0, N - 6)$        $e = \text{MAX}(0, N - 0.7)$   
 $b = \text{MAX}(0, N - 2)$        $f = \text{MAX}(0, (N + f1) - 0.7)$   
 $c = \text{MAX}(0, N - 3)$        $f1 = \text{MAX}(0, Fet - 2.3)$   
 $d = \text{MAX}(0, N - 1)$        $g = \text{MAX}(0, N - 2)$

where "N" is the number of bus cycles needed to perform a word load.

"b", "d" and "f" represent the number of additional cycles required for a non-cached system procedure entry.

**Table 11-7. Worst-Case Interrupt Latency When Delivering a Software Interrupt**

Interrupt Type	Detection Option	Vector Caching Enabled	Worst 80960Hx Latency (Bus Clocks)		
			HA	HD	HT
NMI#	Fast	Yes	77+2b+c	39+2e+f	25.7+2h+i
	Debounced	Yes	77+2b+c	39+2e+f	25.7+2h+i
Dedicated Mode XINT[7:0]#, TINT[1:0]	Fast	Yes	80+2b+c	40+2e+f	26.7+2h+i
		No	80+a+2b+c	40+d+2e+f	26.7+g+2h+i
	Debounced	Yes	80+2b+c	40+2e+f	26.7+2h+i
		No	80+a+2b+c	40+d+2e+f	26.7+g+2h+i
Expanded Mode XINT[7:0]#, TINT[1:0]	Debounced	Yes	80+2b+c	40+2e+f	26.7+2h+i
		No	80+a+2b+c	40+d+2e+f	26.7+g+2h+i

**NOTES:**

a = MAX (0, N - 6)  
 b = MAX (0, N - 1)  
 c = MAX (0, N - 7)

d = MAX (0, N - 3)  
 e = MAX (0, N - 0.5)  
 f = MAX (0, N - 3.5)

g = MAX (0, N - 2)  
 h = MAX (0, N - 0.3)  
 i = MAX (0, N - 2.3)

where "N" is the number of bus cycles needed to perform a word load.

**Table 11-8. Worst-Case Interrupt Latency Controlled by flushreg of One Stack Frame**

Interrupt Type	Detection Option	Vector Caching Enabled	Worst 80960Hx Latency (Bus Clocks)		
			HA	HD	HT
NMI#	Fast	Yes	41+a	21+f	14+i
	Debounced	Yes	43+a	23+f	16.3+i
Dedicated Mode XINT[7:0]#, TINT[1:0]	Fast	Yes	47+b	24+f	16+i
		No	50+c	25.5+e+f	17+h+i
	Debounced	Yes	49+b	26+f	18+i
		No	52+c	27.5+e+f	18+h+i
Expanded Mode XINT[7:0]#, TINT[1:0]	Debounced	Yes	52+b	27+f	19+i
		No	55+c	28.5+b+c	20+h+i

**NOTES:**

a = MAX(0, M - 24)  
 b = MAX(0, M - 28)  
 c = MAX(0, N + c1-6)  
 c1= MAX(0, 4\*Q - 24)

d = MAX(0, N + d1-3)  
 d1 = MAX(0, 4\*M - 12)  
 e = MAX(0, N + e1 - 3)  
 e1 = MAX(0, 3\*M - 4)  
 f = MAX(0, M - 7.5)

g = MAX(0, N + g1-2)  
 g1 = MAX(0, 4\*M - 8)  
 h = MAX(0, N+H1 - 2)  
 h1 = MAX(0, 3\*M - 3.3)  
 i = MAX(0, M - 5)

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.

#### 11.9.4.1 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 for general operations as these are used for procedure linking.)

#### 11.9.4.2 Masking Integer Overflow Faults for syncf

The i960 core architecture requires an implicit **syncf** before delivering an interrupt so that a fault handler can be dispatched first, if necessary. The **syncf** can require a number of cycles to complete if a multi-cycle multiply or divide instruction was issued previously and integer-overflow faults are unmasked (allowed to occur). Interrupt latency can be improved by masking integer-overflow faults, which allows the implicit **syncf** to complete in much shorter time.

This chapter provides information about the i960® Hx processor's Guarded Memory Unit (GMU). The GMU provides memory protection without the performance penalty found in Memory Management Units. The GMU contains two memory protection schemes: one prevents illegal memory accesses, the other only detects memory access violations. Both signal a fault to the microprocessor (assuming faults are enabled).

The *memory protection scheme* is designed to *prevent* access to a specified block of memory. The i960 Hx processor provides the ability to protect, as a minimum, two blocks of addresses with this mechanism. Attempts to illegally access memory protected by this mechanism do *not* generate internal cache accesses or external bus cycles.

The *memory detection scheme* allows the application to define ranges of memory for which illegal accesses generate a fault. However, this mechanism — although more flexible — does not prevent access; it only faults after the illegal access has occurred.

The programmable protection modes are:

- User Read
- User Write
- User Execute
- User Data Cache Write
- Supervisor Read
- Supervisor Write
- Supervisor Execute
- Supervisor Data Cache Write

[Figure 12-1](#) shows how an application might use the GMU. The logical partitions and the access types for each partition are shown.

The programmer interface to the GMU consists of seventeen 32-bit registers, accessible only on word boundaries. These registers are shown in [Table 12-1](#). The GMU registers are memory-mapped and may only be accessed in supervisor mode. The registers are described in greater detail in the following sections.

Figure 12-1. Sample Application with Partitions

		Access Type							
		Supervisor Read/Write		User Read/Write		Execute Sup/User		Data Cache Access Sup/User	
Application RAM		SMR	SMW	UMR	UMW	SME	UME	SCW	UCW
Application/User RAM				✓	✓			✓	✓
User Mode Stack				✓	✓			✓	✓
Interrupt, Supervisor, Fault Tables		✓		✓				✓	✓
Interrupt Stack		✓	✓					✓	✓
Supervisor Stack		✓	✓					✓	✓
Kernel RAM		✓	✓					✓	✓
Kernel Code and Application Code						✓	✓	✓	✓

✓ Indicates access allowed

Application RAM	Notes
Application/User Data	User mode execute regions require user mode read enabled. The tables located in the code space need to be accessed.
User Mode Stack	User mode stacks require both read and write enabled.
Interrupt, Supervisor, Fault Tables	Interrupt Tables need to be located in supervisor address space with both read and write enabled. Fault Table, System Procedure Table, Control Tables need supervisor read enabled.
Interrupt Stack	Since ISRs are always initiated in Supervisor mode, the GMU must allow Supervisor mode reads and writes.
Supervisor Stack	Supervisor mode stacks require both read and write enabled.
Kernel RAM	Supervisor mode execute regions also require supervisor mode read enabled. The tables located in the code space need to be accessed.
Kernel Code and Application Code	User mode RAM requires both read and write enabled.

## 12.1 Illegal Access Protection

The *memory protection scheme* is designed to prevent any fetch or data access to a specified block of memory. The programmable protection modes are:

- User Read
- User Write
- User Execute
- User Data Cache Write
- Supervisor Read
- Supervisor Write
- Supervisor Execute
- Supervisor Data Cache Write

The GMU can protect a minimum of two blocks of memory. A block is defined by a pair of registers. The first register determines the starting address of the block, and the second register determines the size of the block. This is done using a “compare under mask” operation that actually allows more than one block to be specified by the pair, as is explained in [Section 12.3.2](#).

For example, Illegal Access Protection would be useful during the software debugging phase of a project, when an application with software executing from RAM can be subject to errors. These errors may corrupt the executing software by overwriting the code located in RAM. [Figure 12-1](#) shows a block of the RAM containing both the kernel code and the application code. When the GMU detects an illegal access to the protected region, the unit cancels the illegal access and generates a fault.

The blocks defined by the GMU protection registers are independent of the physical and logical regions defined by the Bus Controller, and may span those regions.

## 12.2 Illegal Access Detection

The *memory detection scheme* is similar to the memory protection scheme, however, instead of preventing an access; memory detection only faults after the illegal access has occurred. The programmable detection modes are:

- User Read
- User Write
- User Execute
- User Data Cache Write
- Supervisor Read
- Supervisor Write
- Supervisor Execute
- Supervisor Data Cache Write

Memory detection also gives the application greater flexibility in defining ranges of memory. The application can program six ranges of memory with any combination of access qualifications shown above. A range is defined by a Lower-Bounds address register and a upper-bounds address register. The bounds registers contain the upper 24 bits of address. Each range can span across physical and logical memory region boundaries. The minimum range size is 256 bytes.

The GMU is especially useful during the software debugging phase of the application to detect accesses to specific data structures and arrays.

When an access to memory occurs without the programmed privileges, an imprecise fault is generated. The GMU *will not* prevent the access from completing.

## 12.3 GMU Register Description

The GMU contains 16 programmable registers, organized as eight pairs, for defining the memory protection blocks, and a mode control register for enabling and disabling each pair. Two different protection mechanisms are provided. The GMU registers are memory mapped, as shown in [Table 12-1](#).

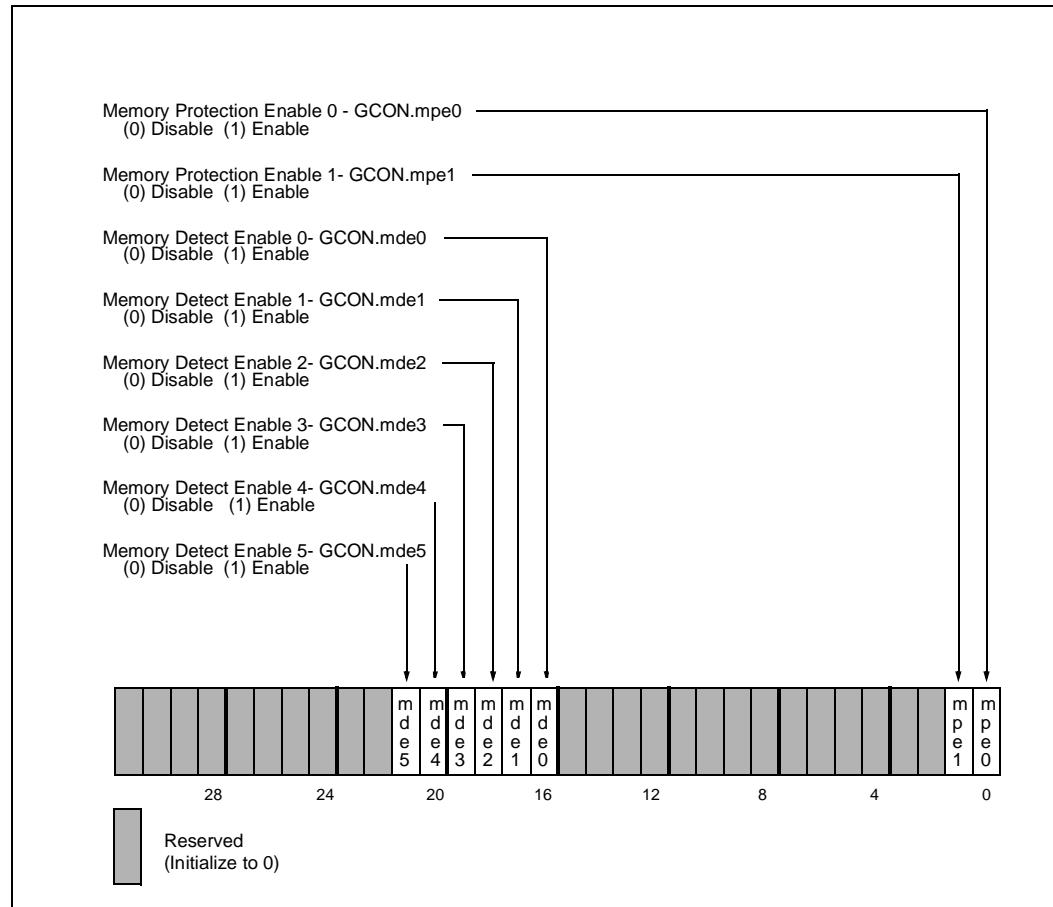
**Table 12-1. GMU Memory-Mapped Registers**

Register Name	GMU Register Description	Address
GCON (sf4)	Control Register	FF00 8000
MPAR0	Memory Protection Address Register 0	FF00 8010
MPMR0	Memory Protection Mask Register 0	FF00 8014
MPAR1	Memory Protection Address Register 1	FF00 8018
MPMR1	Memory Protection Mask Register 1	FF00 801C
MDUB0	Memory Detect Upper-Bounds Address Register 0	FF00 8080
MDLB0	Memory Detect Lower-Bounds Address Register 0	FF00 8084
MDUB1	Memory Detect Upper-Bounds Address Register 1	FF00 8088
MDLB1	Memory Detect Lower-Bounds Address Register 1	FF00 808C
MDUB2	Memory Detect Upper-Bounds Address Register 2	FF00 8090
MDLB2	Memory Detect Lower-Bounds Address Register 2	FF00 8094
MDUB3	Memory Detect Upper-Bounds Address Register 3	FF00 8098
MDLB3	Memory Detect Lower-Bounds Address Register 3	FF00 809C
MDUB4	Memory Detect Upper-Bounds Address Register 4	FF00 80A0
MDLB4	Memory Detect Lower-Bounds Address Register 4	FF00 80A4
MDUB5	Memory Detect Upper-Bounds Address Register 5	FF00 80A8
MDLB5	Memory Detect Lower-Bounds Address Register 5	FF00 80AC

### 12.3.1 GMU Control Register

The GMU Control register (GCON) enables and disables the memory protection registers. It is accessible as Special Function Register 4 (sf4) in the register set, and at address FF00 8000H as a memory-mapped register accessible from supervisor mode. Note that simultaneous access to the register via both mechanisms is not supported. [Figure 12-2](#) describes the bit representations of the GCON. The GCON is the only enable/disable mechanism for the GMU.

**Figure 12-2. GMU Control Register (GCON)**



### 12.3.2 GMU Memory Protect Address and Mask Registers

The GMU *memory protection scheme* uses two registers to define a protected block of addresses: the address register (MPAR) and the mask register (MPMR). The GMU provides registers for two memory protected blocks accessible only from supervisor mode. [Figure 12-3](#) describes the bit definitions of the MPAR and MPMR registers.

The MPAR contains the most significant 24 bits of the target address for the protected memory block(s). The minimum protected block size is 256 bytes. The low-order eight (8) bits in the MPAR register define the access privileges. The meanings of the bits are given below:

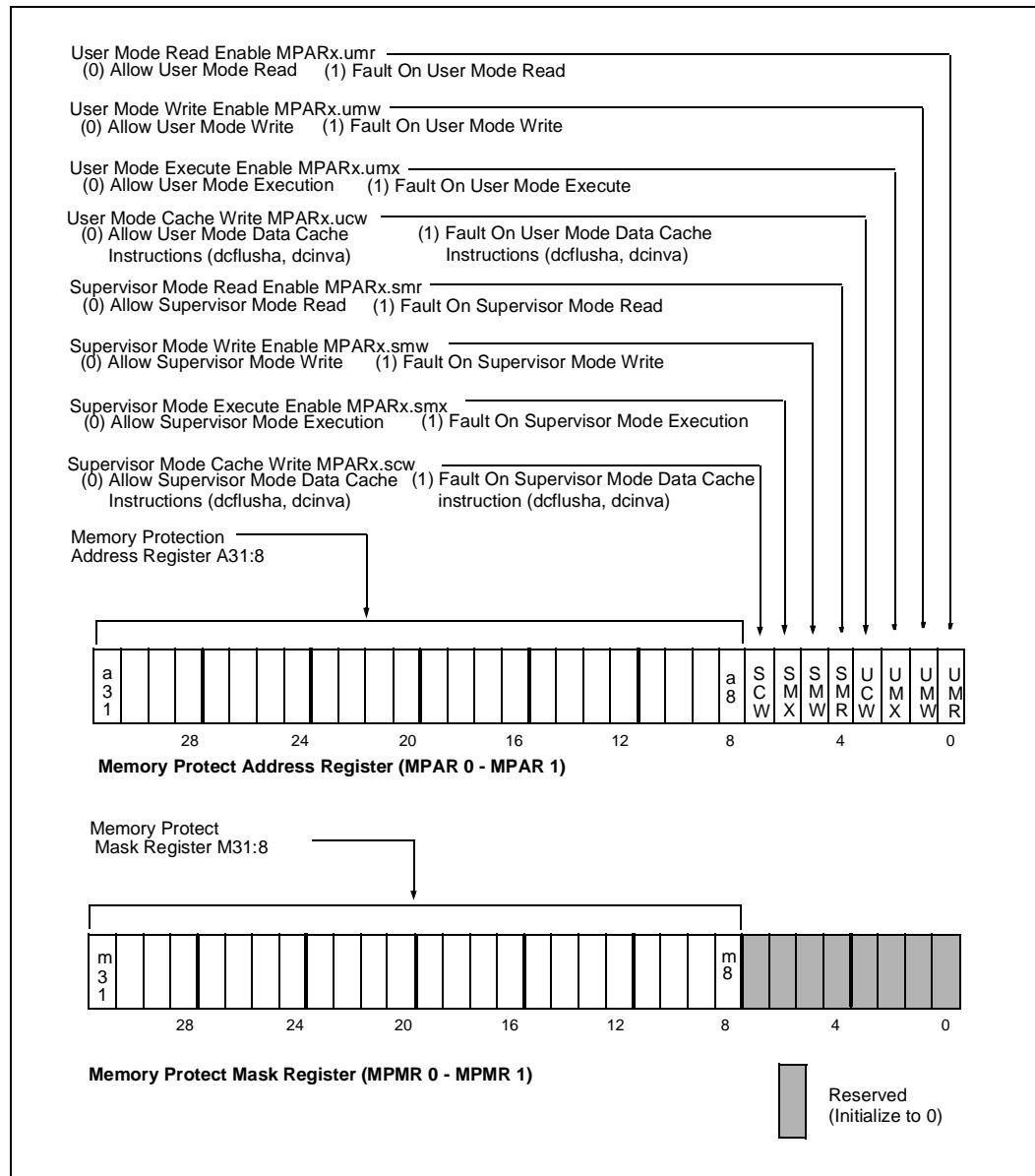
**Table 12-2. MPAR Register Bit Descriptions**

Bit Name	Bit #	Description
User Read	0	When cleared (0) user mode reads are allowed. When set (1) a fault is generated on a user mode read.
User Write	1	When cleared (0) user mode writes are allowed. When set (1) a fault is generated on a user mode write.
User Execute	2	When cleared (0) user mode execution is allowed. When set (1) a fault is generated on a user mode execution.
User Cache Write	3	When cleared (0) user mode <b>dcflusha</b> and <b>dcinva</b> instructions are allowed. When set (1) a fault is generated on user mode <b>dcflusha</b> and <b>dcinva</b> instructions.
Supervisor Read	4	When cleared (0) supervisor mode reads are allowed. When set (1) a fault is generated on a supervisor mode read.
Supervisor Write	5	When cleared (0) supervisor mode writes are allowed. When set (1) a fault is generated on a supervisor mode write.
Supervisor Execute	6	When cleared (0) supervisor mode execution is allowed. When set (1) a fault is generated on a supervisor mode execution.
Supervisor Cache Write	7	When cleared (0) supervisor mode <b>dcflusha</b> and <b>dcinva</b> instructions are allowed. When set (1) a fault is generated on supervisor mode <b>dcflusha</b> and <b>dcinva</b> instructions.
Target Address	31:8	Most significant 24 bits of the target address for protected memory.

The MPMR register is used in conjunction with the MPAR to define protected block boundaries. The mask register contains 24 mask bits. Each mask bit determines if the corresponding address bit found in the MPAR is compared by the GMU hardware: this mechanism is called “compare under mask”. [Figure 12-3](#) shows the layout of MPAR and MPMR.

If a mask bit is set (MPMR.mx = 1), the respective bit in the address register will be compared for a valid address. If the bit is cleared (MPMR.mx = 0), the respective address bit is not compared.

The block sizes are defined by the number of bits cleared in the GMU Mask registers. [Table 12-3](#) shows some examples of mask bit combinations and the block size they represent.

**Figure 12-3. GMU Memory Protect Address Register (MPARx, MPMRx)**


**Table 12-3. GMU Protected Memory Mask Register Block Sizes**

<b>Mask Value</b>	<b>Block Size</b>	<b>Mask Value</b>	<b>Block Size</b>
FFFF FF00	256 Bytes	FFF0 0000	1 Mbyte
FFFF FE00	512 Bytes	FFE0 0000	2 Mbytes
FFFF FC00	1 Kbytes	FFC0 0000	4 Mbytes
FFFF F800	2 Kbytes	FF80 000	8 Mbytes
FFFF F000	4 Kbytes	FF00 0000	16 Mbytes
FFFF E000	8 Kbytes	FE00 0000	32 Mbytes
FFFF C000	16 Kbytes	FC00 0000	64 Mbytes
FFFF 8000	32 Kbytes	F800 0000	128 Mbytes
FFFF 0000	64 Kbytes	F000 0000	256 Mbytes
FFFE 0000	128 Kbytes	E000 0000	512 Mbytes
FFFC 0000	256 Kbytes	C000 0000	1 Gbytes
FFF8 0000	512 Kbytes	8000 0000	2 Gbytes

### 12.3.2.1 Programming the MPAR and MPMR registers

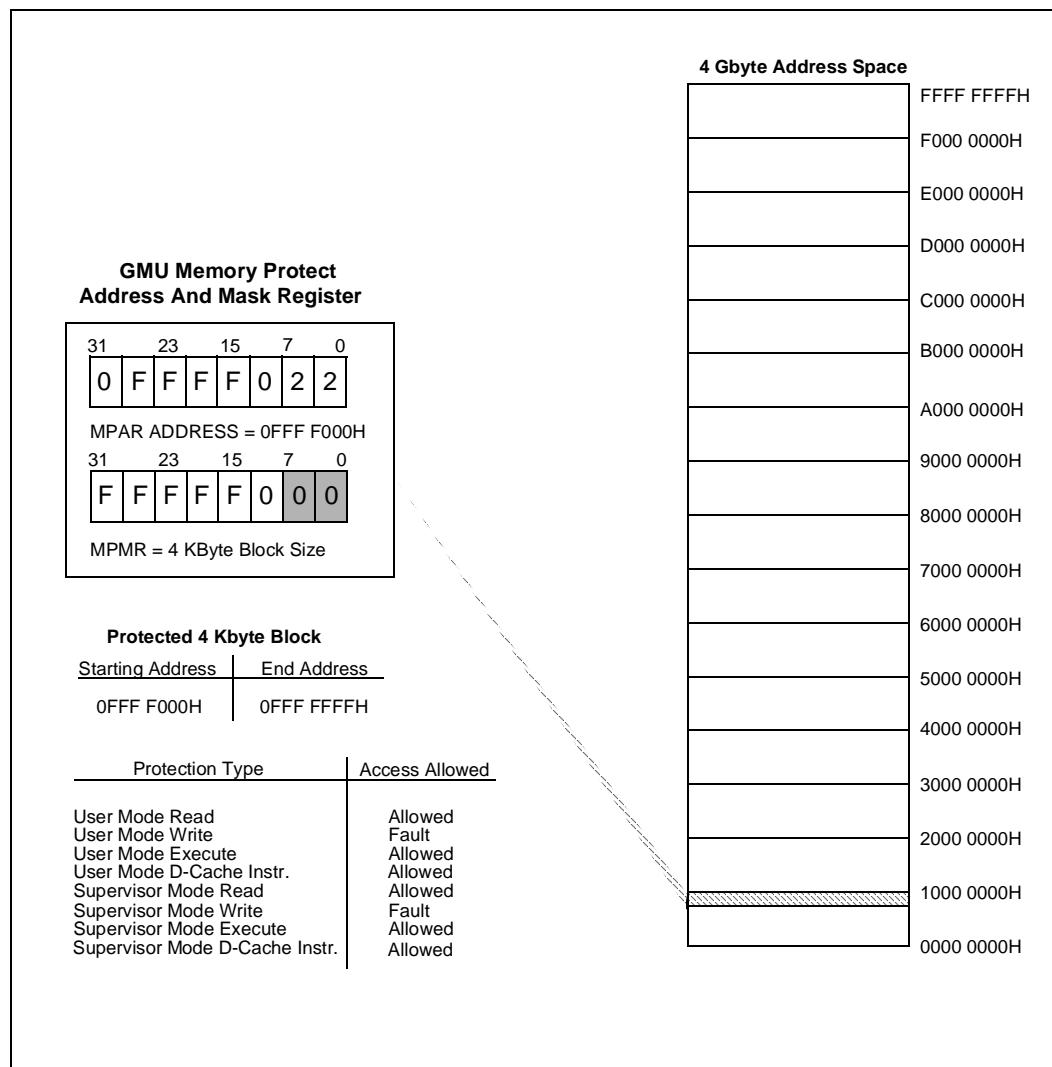
There are special considerations when programming the MPAR and MPMR registers. The mechanism allows only  $2^n$  byte blocks on  $2^n$  boundaries to be defined. For example, if the MPMR register contains the value FFFF F000H, the block size is 4 Kbytes. The starting and ending points of the block defined by the MPAR register will therefore be on 4-Kbyte boundaries. [Figure 12-4](#) describes this example in pictorial form.

Note that software must not attempt to protect the memory-mapped register region (FF00 0000 and above). Also, protected regions must not overlap.

It should be noted that if the MPAR register is programmed to a value not on the natural MPMR block boundary, the less significant bits of MPAR are ignored.

The next example describes the behavior of the MPAR and MPMR registers when some of the upper MPMR bits are cleared. The MPAR register contains the value 0FFC 0000H. The MPMR register contains the value 0FFC 0000H; the block size is 256 Kbytes. Note, bits 28-31 in the MPMR are cleared. [Figure 12-4](#) shows the multiple blocks of protected memory. This occurs because the upper four mask bits (MPMR bits 28-31) are 0. When the mask bits are cleared, the address bits are not compared. This creates the effect of multiple protected addresses with a single pair of registers.

**Figure 12-4. GMU MPAR and MPMR Programming Example**



### 12.3.3 GMU Memory Detect Upper- and Lower-Bounds Registers

The GMU memory detect violation mode uses pairs of registers to define a protected address range: an upper-bounds register (MDUB) and a lower-bounds register (MDLB). These registers define a *memory range*. Each memory range has independent access permission characteristics.

Note that software must not attempt to protect the memory-mapped register region (FF00 0000 and above). Also, protected regions must not overlap.

**Figure 12-5. GMU MPAR and MPMR Programming Example**

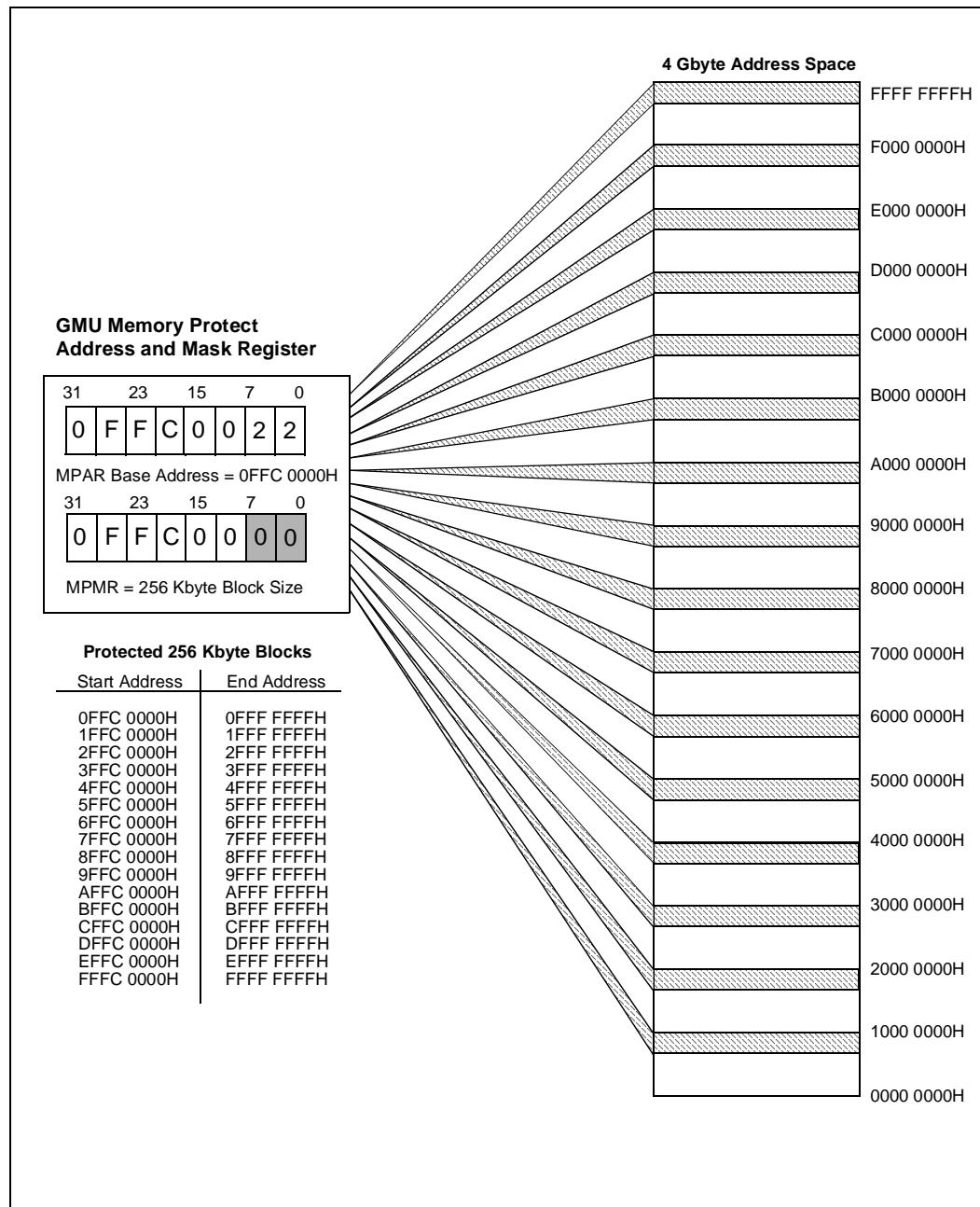
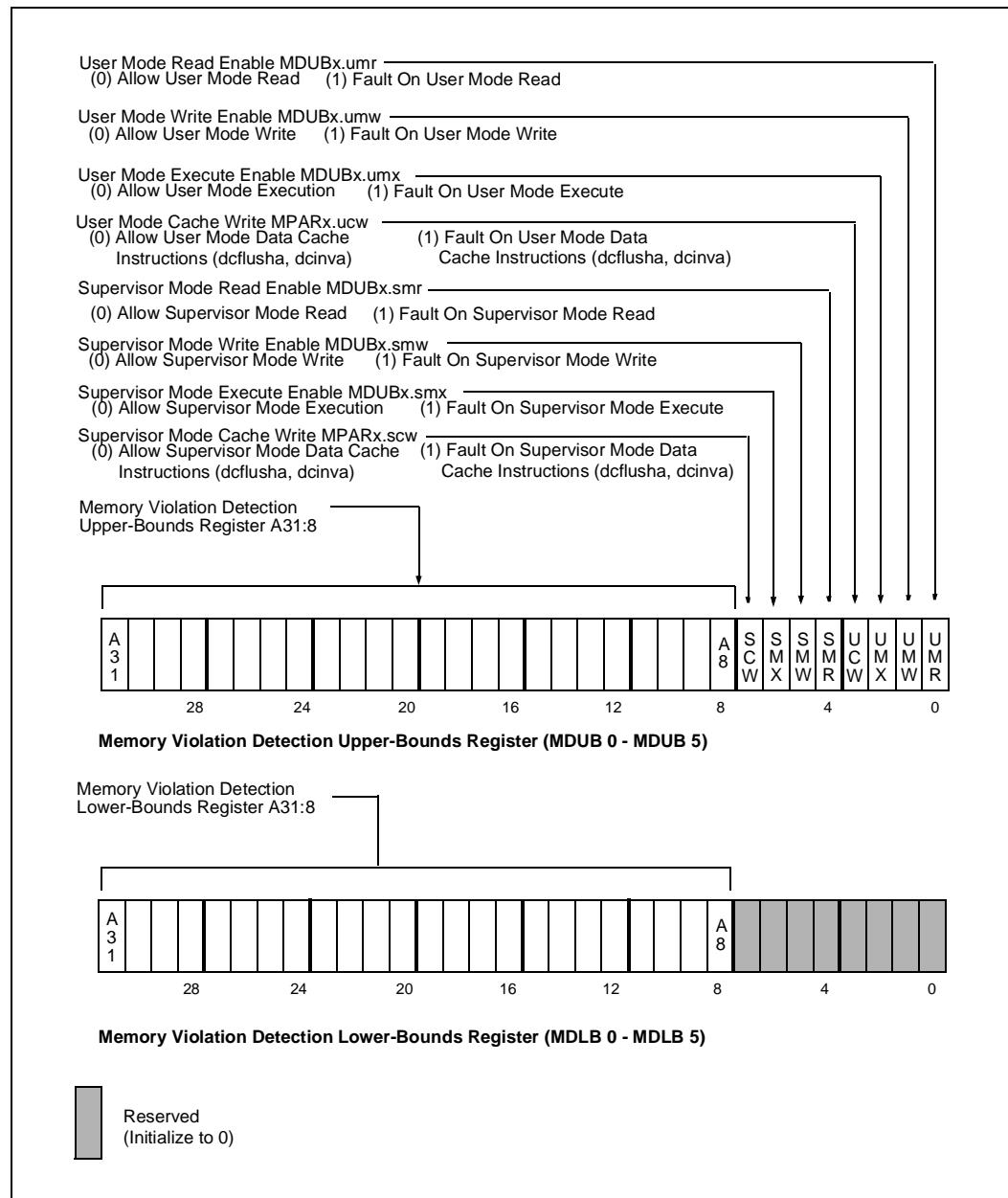


Figure 12-6 describes the MDLB and MDUB registers.

**Figure 12-6. GMU Memory Violation Detection Upper and Lower-Bounds Registers**



The MDLB register contains 24 bits used to define the lower bounds of the memory range. The MDLB register is *inclusive* of the address programmed. For example, if the MDLB contained the value 0C00 0000H, the beginning address in the range would be 0C00 0000H.

The MDUB register contains 24 bits used to define the upper bounds of the memory range. It also contains the access types bits. The MDUB register is *exclusive* of the address programmed. For example, if the MDUB contained the value 1E00 0024H, the last address contained in the range would be 1DFF FFFFH. The low-order eight (8) bits in the MDUB register define the access privileges. The bit representations are shown below:

**Table 12-4. MDUB Register Bit Descriptions**

Bit Name	Bit #	Description
User Read	0	When cleared (0) user mode reads are allowed. When set (1) a fault is generated on a user mode read.
User Write	1	When cleared (0) user mode writes are allowed. When set (1) a fault is generated on a user mode write.
User Execute	2	When cleared (bit 2 = 0) user mode execution is allowed. When set (1) a fault is generated on a user mode execution.
User Cache Write	3	When cleared (0) user mode <b>dcflusha</b> and <b>dcinva</b> instructions are allowed. When set (1) a fault is generated on user mode <b>dcflusha</b> and <b>dcinva</b> instructions.
Supervisor Read	4	When cleared (0) supervisor mode reads are allowed. When set (1) a fault is generated on a supervisor mode read.
Supervisor Write	5	When cleared (0) supervisor mode writes are allowed. When set (1) a fault is generated on a supervisor mode write.
Supervisor Execute	6	When cleared (0) supervisor mode execution is allowed. When set (1) a fault is generated on a supervisor mode execution.
Supervisor Cache Write	7	When cleared (0) supervisor mode <b>dcflusha</b> and <b>dcinva</b> instructions are allowed. When set (1) a fault is generated on supervisor mode <b>dcflusha</b> and <b>dcinva</b> instructions.
Memory Violation Detect Upper-Bounds Address	8-31	The 24 bits used to define the upper bounds of the memory range

The GMU provides registers for six memory ranges. These registers are accessible only when the processor is in supervisor mode. The MDUB and MDLB are not bound by any physical or logical memory regions, and can span multiple memory regions.

The programmed mode bits define which access types are allowed. Combinations of these are very important for normal code execution, such as those shown [Figure 12-1](#).

### 12.3.4 GMU Faults

The GMU generates faults based on the programmed protection and/or the mode of access to the GMU registers. See [Section 8.10.7, “PROTECTION Faults” on page 8-30](#) for detailed description of GMU faults and the GMU fault record.

## 12.4 GMU Powerup Modes

Upon powerup, an external hardware reset, or software reset (`sysctl`), the GMU registers are initialized to the values shown in [Table 12-5](#).

**Table 12-5. GMU Powerup and Reset Values**

Register	Value at Power-Up	Value after Reset	Notes
(GCON) GMU Control Register	0000 0000	0000 0000	All GMU Protection registers disabled.
(MPARx, MPMRx) All Memory Protection Registers	indeterminate	Value before reset	User must program these registers prior to enabling the GMU memory protection.
(MDUBx, MDLBx) All Memory Detection Address Registers	indeterminate	Value before reset	User must program these registers prior to enabling the GMU memory detection.

## 12.5 GMU Programming Considerations

The programmer should always disable GMU faults in the GCON before modifying any of the GMU address or mask registers, to prevent accidental generation of GMU faults.

If any of the PROTECTION, PARALLEL, OVERRIDE or TRACE fault handlers are protected by the GMU, infinite recursion within the fault handlers may occur. If the PROTECTION handler is fetch-protected by the GMU, the processor faults infinitely with no way of being interrupted except through reset.

The fault handler should not attempt to return from a PROTECTION.BAD\_ACCESS fault caused by a GMU protection condition, because the stack frame cache may be in an undefined state in that case.

The GMU cannot detect previously cached instruction fetches. To insure that all instruction fetches are monitored, invalidate the instruction cache (using an **icctl** instruction) after enabling GMU protection/detection.

Due to instruction prefetching, a spurious PROTECTION.BAD\_ACCESS fault may be generated when the target of a branch is in a region fetch-protected by the GMU and the branch is predicted to be taken, but not actually taken. For application debugging with the GMU, conditional branches to regions protected by the GMU should always be predicted as not taken.

Software should not program the GMU to protect the memory-mapped registers in the range of addresses FFFFFFF00H through FFFFFFFFH as this can lead to the unexpected generation of PROTECTION.BAD\_ACCESS faults.

In general, the Interrupt Table should not be protected against Supervisor mode accesses. Protecting the Interrupt Table from Supervisor mode writes is acceptable if it can be guaranteed that no software posting of interrupts will occur. Protecting the Interrupt Table against Supervisor mode reads will cause trouble if any hardware interrupts, including the NMI, occur. Violation of these cautions will result in improper system behavior.



# Initialization and System Requirements

13

This chapter describes the steps that the i960<sup>®</sup> Hx processor performs during initialization. Discussed are the RESET# pin and the reset state 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.

## 13.1 Overview

During the time that the RESET# pin is held asserted, the processor is in a quiescent reset state. All external pins are inactive and the internal processor state is forced to a known condition. The processor begins initialization when the RESET# pin is deasserted.

When 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.

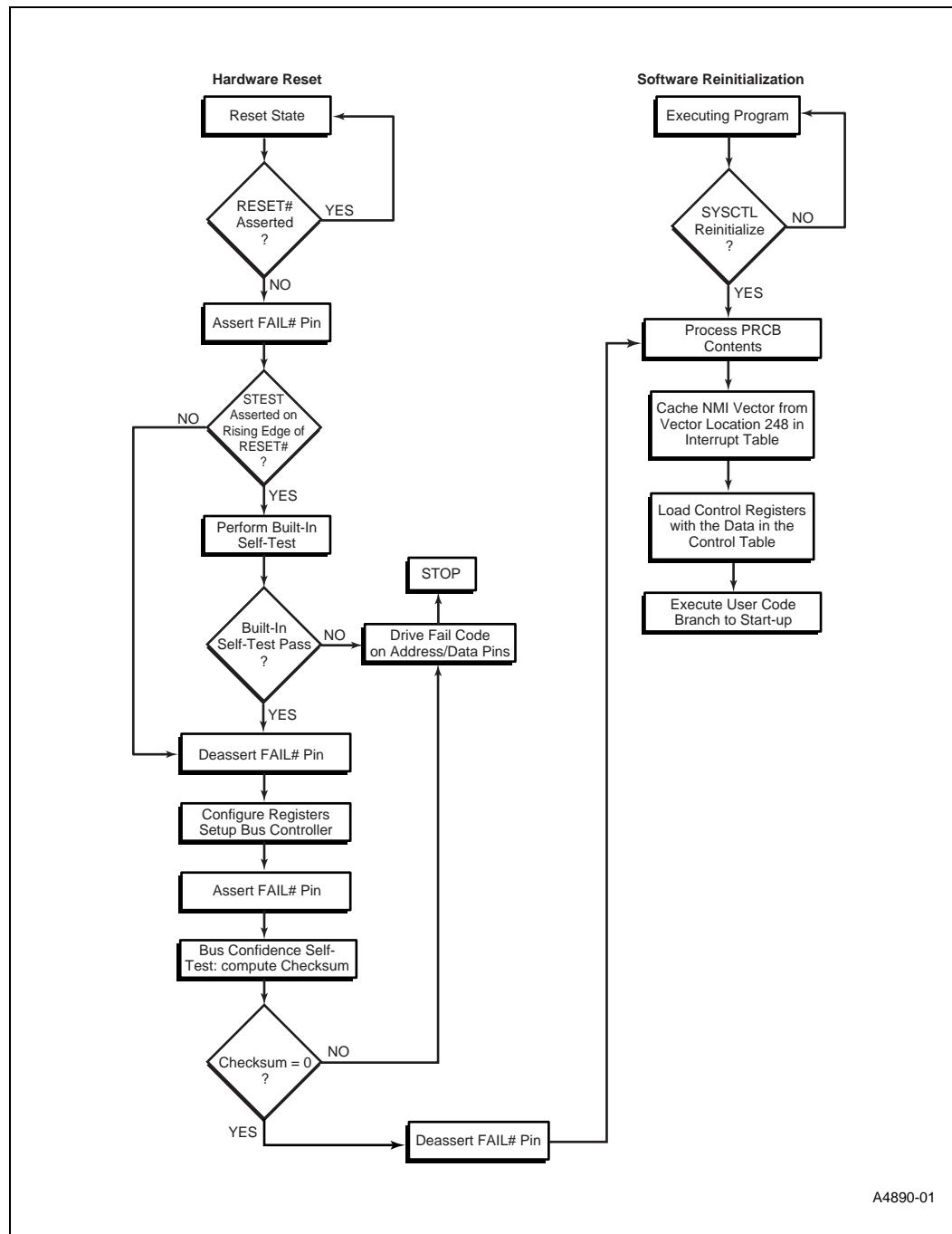
Software can reinitialize the processor using the **sysctl** instruction ([Section 6.2.67, “sysctl” on page 6-108](#)). When a reinitialization takes place, a new PRCB and reinitialization instruction pointer are specified. Reinitialization is useful for relocating data structures from ROM to RAM after initialization.

The i960 Hx 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 Hx processor also supports JTAG boundary-scan (see [Chapter 16, “Test Features”](#)). During initialization, the processor performs an internal functional self test and external 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 (CLKIN) and clean power and ground connections (V<sub>SS</sub> and V<sub>CC</sub>). 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.

## 13.2 Initialization

This section describes the mechanism that the processor uses to establish its initial state and begin instruction execution. Initialization begins when the RESET# pin is deasserted. At this time, the processor automatically configures itself with information specified in the IMI and performs its built-in self test based on the condition of the STEST pin. The processor then branches to the first instruction of user code. See [Figure 13-1](#) for a flow chart of i960 Hx processor initialization.

**Figure 13-1. Processor Initialization Flow**


A4890-01

The objective of the initialization sequence is to provide a complete working initial state when the first user instruction executes. The user's startup code needs only to perform several basic functions to place the processor in a configuration for executing application code.

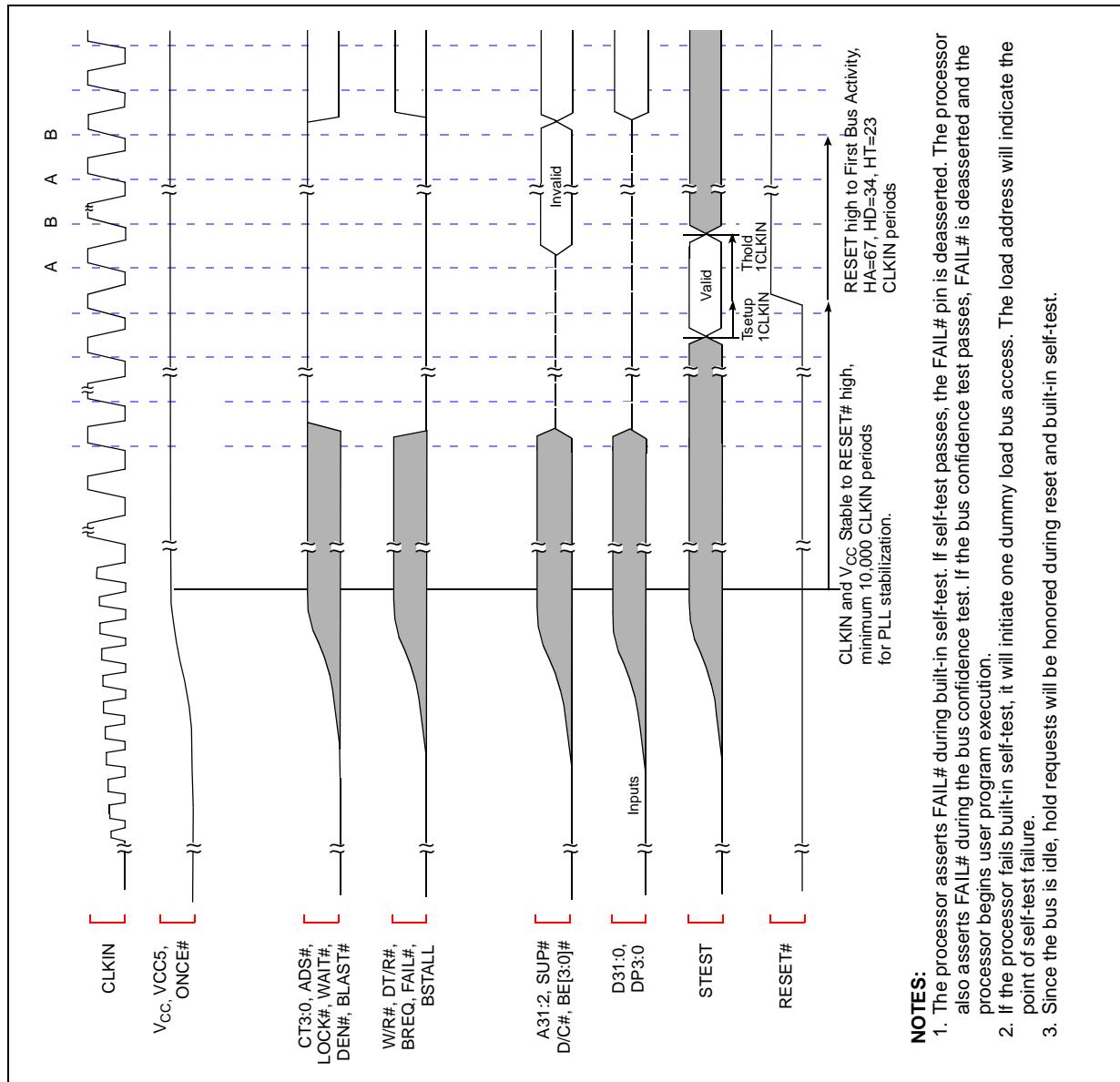
### 13.2.1 Reset State Operation

The RESET# pin, when asserted (active low), causes the processor to enter the reset state. The processor sets all external signals to a defined state ([Table 13-1](#)), initializes internal logic, and sets certain registers to defined values ([Table 13-2](#)). When the RESET# pin is deasserted, the processor initializes as shown in [Section 13.5, “Startup Code Example” on page 13-25](#). RESET# is a level-sensitive, asynchronous input. If HOLD is asserted while the processor is in reset, the processor will acknowledge the request. All external pins will assume their usual states while the bus is in the hold state. See [Section 15.6.1, “The HOLD and HOLDA Signals” on page 15-32](#).

The RESET# pin 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 (CLKIN) and V<sub>CC</sub> must be present and stable for a specified time before RESET# can be deasserted.

The processor may also be cycled through the reset state after execution has started. This sequence is referred to as *warm reset*. For a warm reset, the RESET# pin must be asserted for a minimum number of clock cycles. If a warm reset is asserted during a bus hold, the processor continues to drive HOLDA until HOLD is deasserted, at which point the processor begins the internal initialization process. Specifications for a cold and warm reset can be found in the *80960HA/HD/HT Embedded 32-bit Microprocessor* datasheet.

While the processor's RESET# pin is asserted, output pins are driven to the states as indicated in [Table 13-1](#). The reset state cannot be entered under direct control from user code. No reset instruction or other condition that forces a reset exists on the i960 Hx processors. The RESET# pin must be asserted to enter the reset state. The processor does, however, provide a means to re-enter the initialization process. See [Section 13.4.1, “Reinitializing and Relocating Data Structures” on page 13-24](#).

**Figure 13-2. Cold Reset Waveform**

**NOTES:**

1. The processor asserts FAIL# during built-in self-test. If self-test passes, the FAIL# pin is deasserted. The processor also asserts FAIL# during the bus confidence test. If the bus confidence test passes, FAIL# is deasserted and the processor begins user program execution.
2. If the processor fails built-in self-test, it will initiate one dummy load bus access. The load address will indicate the point of self-test failure.
3. Since the bus is idle, hold requests will be honored during reset and built-in self-test.

**Table 13-1. Pin Reset State**

Pins	Reset State	Pins	Reset State
A31:2	Floating	BREQ	Low (inactive)
D31:0	Floating	D/C#	Low (code)
BE[3:0]#	High (inactive)	SUP#	Floating
W/R#	Low (read)	FAIL#	Low (active)
ADS#	High (inactive)	TDO	Valid output
WAIT#	High (inactive)	CT[3:0]	Floating
BLAST#	High (inactive)	BSTALL	Low (inactive)
DT/R#	Low (receive)	HOLDA	Valid Output
DEN#	High (inactive)	PCHR#	High (inactive)
LOCK#	High (inactive)	DP3:0	Floating

**NOTE:** Pin states shown assume ONCE# pin is not asserted. If the ONCE# pin is asserted, the processor pins are all floated.

**Table 13-2. Register Values after Reset (Sheet 1 of 2)**

Register	Value after Power-On Reset, in Hex	Value after Software Re-Init, in Hex
g0	Device ID	Device ID
IPND (sf0)	undefined	value before software re-init
IMSK (sf1)	00	00
CCON(sf2)	bit 31 = 1, others = 0	bit 31 = 1, others = 0
ICON(sf3)	value of Control Table word 7	value of Control Table word 7
GCON(sf4)	bits 0-7 = 0; bits 8 -31 = undefined	bits 0-7 = 0; bits 8 -31 = undefined
MPAR0-1	undefined	value before software re-init
MPMR0-1	undefined	value before software re-init
MDUB0-5	undefined	value before software re-init
MDLB0-5	undefined	value before software re-init
LMAR0-14	undefined	value before software re-init
LMMR0-14	bit 0 = 0; bits 1 -31 = undefined	bit 0 = 0; bits 1 -31 = undefined
DLMCON	bit 0 see note 1; bits 1 and 4 =0; others undefined	bit 0 see note 1; bits 1 and 4 =0; others undefined
TRR0-1	undefined	value before software re-init
TCR0-1	undefined	value before software re-init
TMR0-1	bits 0 - 5 = 0; bits 6 - 31 = undefined	bits 0 - 5 = 0; bits 6 - 31 = undefined
IPB0	0000 000H	0000 000H
IPB1	0000 000H	0000 000H
DAB0	0000 000H	0000 000H
DAB1	0000 000H	0000 000H

1. Loaded from BBIGE bit in the Initialization Boot Record.

**Table 13-2. Register Values after Reset (Sheet 2 of 2)**

Register	Value after Power-On Reset, in Hex	Value after Software Re-Init, in Hex
IMAP0	initial image in Control Table, offset 10	initial image in Control Table, offset 10
IMAP1	initial image in Control Table, offset 14	initial image in Control Table, offset 14
IMAP2	initial image in Control Table, offset 18	initial image in Control Table, offset 18
ICON	initial image in Control Table, offset 1C	initial image in Control Table, offset 1C
PMCON0	initial image in Control Table, offset 20	initial image in Control Table, offset 20
PMCON1	initial image in Control Table, offset 24	initial image in Control Table, offset 24
PMCON2	initial image in Control Table, offset 28	initial image in Control Table, offset 28
PMCON3	initial image in Control Table, offset 2C	initial image in Control Table, offset 2C
PMCON4	initial image in Control Table, offset 30	initial image in Control Table, offset 30
PMCON5	initial image in Control Table, offset 34	initial image in Control Table, offset 34
PMCON6	initial image in Control Table, offset 38	initial image in Control Table, offset 38
PMCON7	initial image in Control Table, offset 3C	initial image in Control Table, offset 3C
PMCON8	initial image in Control Table, offset 40	initial image in Control Table, offset 40
PMCON9	initial image in Control Table, offset 44	initial image in Control Table, offset 44
PMCON10	initial image in Control Table, offset 48	initial image in Control Table, offset 48
PMCON11	initial image in Control Table, offset 4C	initial image in Control Table, offset 4C
PMCON12	initial image in Control Table, offset 50	initial image in Control Table, offset 50
PMCON13	initial image in Control Table, offset 54	initial image in Control Table, offset 54
PMCON14	initial image in Control Table, offset 58	initial image in Control Table, offset 58
PMCON15	initial image in Control Table, offset 5C	initial image in Control Table, offset 5C
BPCON	0000 000H	0000 000H
TC	initial image in Control Table, offset 68	initial image in Control Table, offset 68
BCON	initial image in Control Table, offset 6C	initial image in Control Table, offset 6C
DEVICEID	initialized by microcode	initialized by microcode
IPB2-IPB5	0000 0000H	0000 0000H
DAB2-DAB5	0000 0000H	0000 0000H
XBPCON	0000 0000H	0000 0000H
AC	AC initial image in PRCB	AC initial image in PRCB
PC	C01F2002H	C01F2002H
TC	initial image in Control Table, offset 68H	initial image in Control Table, offset 68H
FP (g15)	interrupt stack base	interrupt stack base
PFP (r0)	undefined	undefined
SP (r1)	interrupt stack base+64	interrupt stack base+64
RIP (r2)	undefined	undefined

1. Loaded from BBIGE bit in the Initialization Boot Record.

## 13.2.2 Self Test Function (STEST, FAIL#)

As part of initialization, the i960 Hx processor executes a 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) pin enables or disables built-in self test. The FAIL# pin indicates whether the self tests passed or failed. During normal operations the FAIL# pin can be asserted if a System Error is detected. The following subsections further describe these pin functions.

The Built-In Self Test (BIST) 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.

### 13.2.2.1 The STEST Pin

The STEST pin enables BIST. The user can disable BIST if the initialization time needs to be minimized or if diagnostics are not necessary. The processor samples STEST pin on the rising edge of the RESET# input:

- If STEST is asserted (high), the processor executes BIST.
- If STEST is deasserted, the processor bypasses BIST.

### 13.2.2.2 External Bus Confidence Test

The processor always performs the external bus confidence test regardless of STEST pin value.

The external bus confidence test checks external bus functionality; it reads eight words from the Initialization Boot Record (IBR) and performs a checksum on the words and the constant FFFF FFFFH. The test passes only when the processor calculates a sum of zero (0). The external bus confidence test can detect catastrophic bus failures such as external address, data or control lines that are stuck, shorted or open.

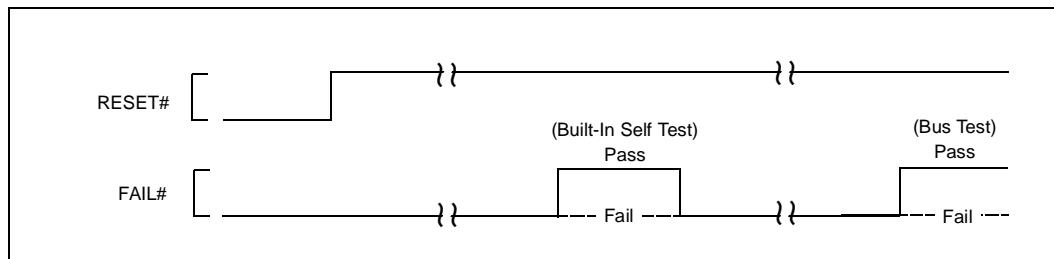
### 13.2.2.3 The Fail Pin (FAIL#)

The FAIL# pin signals errors in either the Built-In Self Test or the external bus confidence self test. FAIL# is asserted (low) during each self test ([Figure 13-3](#)):

- When any test fails, the FAIL# pin remains asserted, a fail code message is driven onto the address bus, and the processor stops execution at the point of failure.
- When an external system error occurs, FAIL# remains asserted. See section 13.2.2.4 for details.
- When the test passes, FAIL# is deasserted.

If FAIL# stays asserted, the only way to resume normal operation is to perform a reset operation. When the STEST pin is used to disable BIST, the test does not execute; however, FAIL# still asserts at the point where BIST would occur. FAIL# is deasserted after the bus confidence test passes. In [Figure 13-3](#), all transitions on the FAIL# pin are relative to CLKIN as described in the *80960HA/HD/HT Embedded 32-bit Microprocessor* datasheet.

**Figure 13-3. FAIL# Functional Timing**



#### 13.2.2.4 IMI Alignment Check and System 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 proper alignment of these key data structures. The alignment check is one case where a System Error could occur.

When the processor detects a System Error, it asserts the FAIL# pin, drives a fail code message onto the address bus, and stops execution at the point of failure. The only way to resume normal operation of the processor is to perform a reset operation. Because System Error generation can occur after the bus confidence test and even after initialization during normal processor operation, the FAIL# pin is a logical “1” before the detection of a System Error.

#### 13.2.2.5 Self Test Failure# Codes

When the processor fails the self test, the FAIL# pin asserts and the processor signals the cause of the failure. 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 1, indicates the mask contains failures from BIST; when 0, the mask indicates other failures. The fail codes are shown in [Table 13-3](#) and [Table 13-4](#).

**Table 13-3. Fail Codes for BIST (bit 7 = 1)**

Bit	When set:
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 13-4. Remaining Fail Codes (bit 7 = 0)**

Bit	When set:
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 System Error during normal operation has occurred.
2	The Bus Confidence test has failed.
1	Always zero.
0	Always zero.

## 13.3 Architecturally Reserved Memory Space

The i960 Hx processor contains  $2^{32}$  bytes of address space. This address space contains portions that are architecturally reserved and must not be used by customers. Section 3.5 Memory Address Space (pg. 3-15) shows the reserved address space. The i960 Hx processor suppresses all external bus cycles from 0 to 7FFH and from FF00 0000H to FFFF FFFFH.

Addresses FF00 0000H through FFFF FFFFH are reserved for implementation-specific functions. This address range is termed “reserved” since i960 architecture implementations may use these addresses for functions such as memory-mapped registers or data structures. To ensure complete object level compatibility, portable code must not access or depend on values in this region.

The i960 Hx processor uses the reserved address range 0000 0000H through 0000 07FFH for internal data RAM. This internal data RAM is used for storage of interrupt vectors plus general purpose storage available for application software variable allocation or data structures. Loads and stores directed to these addresses access internal memory; instruction fetches from these addresses are not allowed. See [Chapter 4, “Cache and On-Chip Data RAM”](#), for more details.

### 13.3.1 Initial Memory Image (IMI)

The IMI comprises the minimum set of data structures that the processor needs to initialize its system. As shown in [Figure 13-4](#), 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 and integrated peripherals
- 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 start-up

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 system's boot ROM, located in memory region 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 FF00 0000H. In addition, each of these structures must start at a word-aligned address; a System Error occurs if any of these structures are not word-aligned (see section 13.2.2.3).

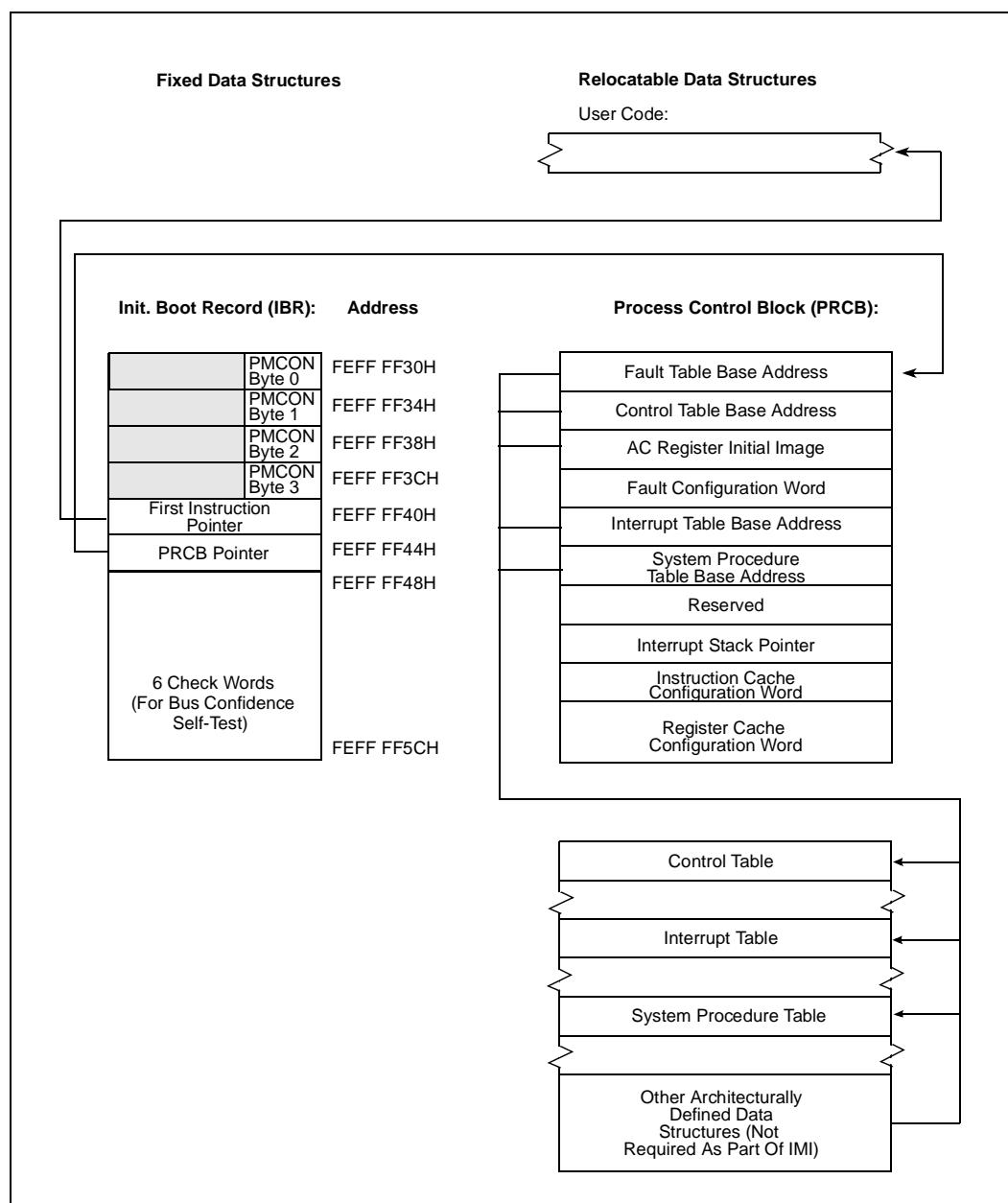
At initialization, the processor loads the Supervisor Stack Pointer (SSP) from the system procedure table, aligns it to a 16-byte boundary, and caches the pointer in the SSP memory-mapped control register (see [Section 3.3, “Memory-Mapped Control Registers” on page 3-6](#)). Recall that the Supervisor Stack Pointer is located in the preamble of the system procedure table at byte offset 12 from the base address. The system procedure table base address is programmed in the PRCB. Consult [Section 7.5.1, “System Procedure Table” on page 7-15](#) for the format of this table.

At initialization, the NMI vector loads from the interrupt table into 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.

Typically, applications locate the fault table in boot ROM. 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. Applications must locate these stacks in a system’s RAM.

**Figure 13-4. Initial Memory Image (IMI) and Process Control Block (PRCB)**



### 13.3.1.1 Initialization Boot Record (IBR)

The initialization boot record (IBR) is the primary data structure required to initialize the i960 Hx processor. The IBR is a 12-word structure that must be located at address FEFF FF30H (see [Table 13-5](#)). The IBR has four components: the initial bus configuration data, the first instruction pointer, the PRCB pointer and the bus confidence test checksum data.

**Table 13-5. Initialization Boot Record**

Byte Physical Address	Description
FEFF FF30H	PMCON15, byte 0
FEFF FF31H to FEFF FF33H	Reserved
FEFF FF34H	PMCON15, byte 1
FEFF FF35H to FEFF FF37H	Reserved
FEFF FF38H	PMCON15, byte 2
FEFF FF39H to FEFF FF3BH	Reserved
FEFF FF3CH	PMCON15, byte 3
FEFF FF3DH to FEFF FF3FH	Reserved
FEFF FF40H to FEFF FF43H	First Instruction Pointer
FEFF FF44H to FEFF FF47H	PRCB Pointer
FEFF FF48H to FEFF FF4BH	Bus Confidence Self-Test Check Word 0
FEFF FF4CH to FEFF FF4FH	Bus Confidence Self-Test Check Word 1
FEFF FF50H to FEFF FF53H	Bus Confidence Self-Test Check Word 2
FEFF FF54H to FEFF FF57H	Bus Confidence Self-Test Check Word 3
FEFF FF58H to FEFF FF5BH	Bus Confidence Self-Test Check Word 4
FEFF FF5CH to FEFF FF5FH	Bus Confidence Self-Test Check Word 5

When the processor reads the IMI during initialization, it must know the bus characteristics of external memory where the IMI is located. Specifically, it must know the bus width and endianism for the remainder of the IMI. At initialization, the processor sets the PMCON register to an 8-bit bus width. The processor then needs to form the initial DLMCON and PMCON15 registers so it can access the memory containing the IBR. The lowest-order byte of each of the IBR's first 4 words are used to form the register values.

### Example 13-1. Processor Initialization 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 PMCON15 to control all accesses */
    PMCON15 = 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 = 0xfefffff30;     /* ibr_ptr used to fetch IBR words      */

    /** Read PMCON15 image in IBR **/
    FAIL_pin = true;           IMSK          = 0;
    DLMCON.dcen = 0;           LMMR0.lmte = 0;   LMMR1.lmte = 0;
    DLMCON.be = (memory[ibr_ptr + 0xc] >> 7);
    PMCON15[byte2] = 0xc0 & memory[ibr_ptr + 8];

    /** Compute CheckSum on Boot Record **/
    carry = 0; CheckSum = 0xffffffff;
    for (i=0; i<8; i++) /* carry is carry out from previous add*/
        CheckSum = memory[ibr_ptr + 16 + i*4] + CheckSum + carry;
    if (CheckSum != 0)
        { fail_msg = 0xfffffff64; /* 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];
    ctrl_table = memory[prcb_ptr+4];
    Process_PRCB(prcb_ptr); /* See Process PRCB Section for Details */
    IP = memory[ibr_ptr+0x10];
    g0 = DEVICE_ID;
    return; /* Execute First Instruction */
}

```

The four bytes residing at FEFF FF30H, FEFF FF34H, FEFF FF38H and FEFF FF3CH form a composite word that describes the characteristics of external memory that are used to fetch the rest of the IMI, as shown in [Figure 13-5](#). (Note that the meanings of the bits in these bytes differ from those of the i960 Cx processor.) Bits 30 through 0 of this word are loaded into PMCON15, and PMCON15[31] is set to zero; this sets the physical characteristics such as timing and bus width. Bit 31 of the composite word is loaded into bit 0 of the Default LMCN, which sets the “endianess”.

Note that room for a re-mapped copy of the i960 CF processor IBR exists at address FEFF FF00H. This simplifies building a system where a single ROM can be used for either an i960 Cx or Hx processor.

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. (See [Chapter 15, “External Bus Description”](#) for a complete discussion of memory regions and configuring the bus controller.)

After the bus configuration data loads 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 processor caches the PRCB pointer and the first instruction pointer internally. The processor uses six checksum words along with the PRCB pointer and the first instruction pointer in a checksum calculation that implements an external bus confidence test. The pseudo-code flow in [Example 13-1](#) shows the checksum calculation. If the checksum calculation equals zero, then the confidence test of the external bus passes.

[Figure 13-5](#) further describes the IBR organization.

Figure 13-5. PMCON15 Register Bit Description in IBR

Mnemonic	Name	Bit #	Function
NRAD4:0	Number of Read Address to Data Wait States	0-4	Specifies the number of wait states (0 to 31) inserted between the assertion of ADS# and the first data cycle for read accesses.
NRDD1:0	Number of Read Data to Data Wait States	6-7	Specifies the number of wait states (0 to 3) inserted between successive data cycles for a burst read access.
NWAD4:0	Number of Write Address to Data Wait States	8-12	Specifies the number of wait states (0 to 31) inserted between the assertion of ADS# and the first data cycle for write accesses.
NWDD1:0	Number of Write Data to Data Wait States	14-15	Specifies the number of wait states (0 to 3) inserted between successive data cycles for a burst write access.
NXDA3:0	Number of Bus Turnaround Wait States	16-19	Specifies the number of wait states (0-15) inserted after the last data cycle for accesses in this region.
PEN	Parity Enable	20	Enables parity generation/checking for a region. 0 = not enabled, 1 = enabled
PODD	Parity Odd	21	Selects parity sense for a region. 0 = even, 1 = odd
BW1:0	Bus Width	22-23	Selects the bus width for a region: 00 = 8-bit, 01 = 16-bit, 10 = 32-bit bus 11 = reserved (do not use)
RPIPE	Read Pipelining Enable	24	Enables address pipelining for read accesses in this region. 0 = disabled, 1 = enabled
BEN	Burst Enable	28	Enables burst accesses for the region. 0 = disabled, 1 = enabled
RBEN	READY#/BTERM# Enable	29	Enables the READY# and BTERM# pins for a region. 0 = READY#/BTERM# ignored in this region 1 = READY#/BTERM# enabled
BBIGE	Boot Big-Endian Byte Order	31	0 = little-endian, 1 = big-endian This bit is copied to the DLMCON.be bit. See <a href="#">Section 14.4 "Programming the Logical Memory Attributes" on page 14-11</a>

### 13.3.1.2 Process Control Block (PRCB)

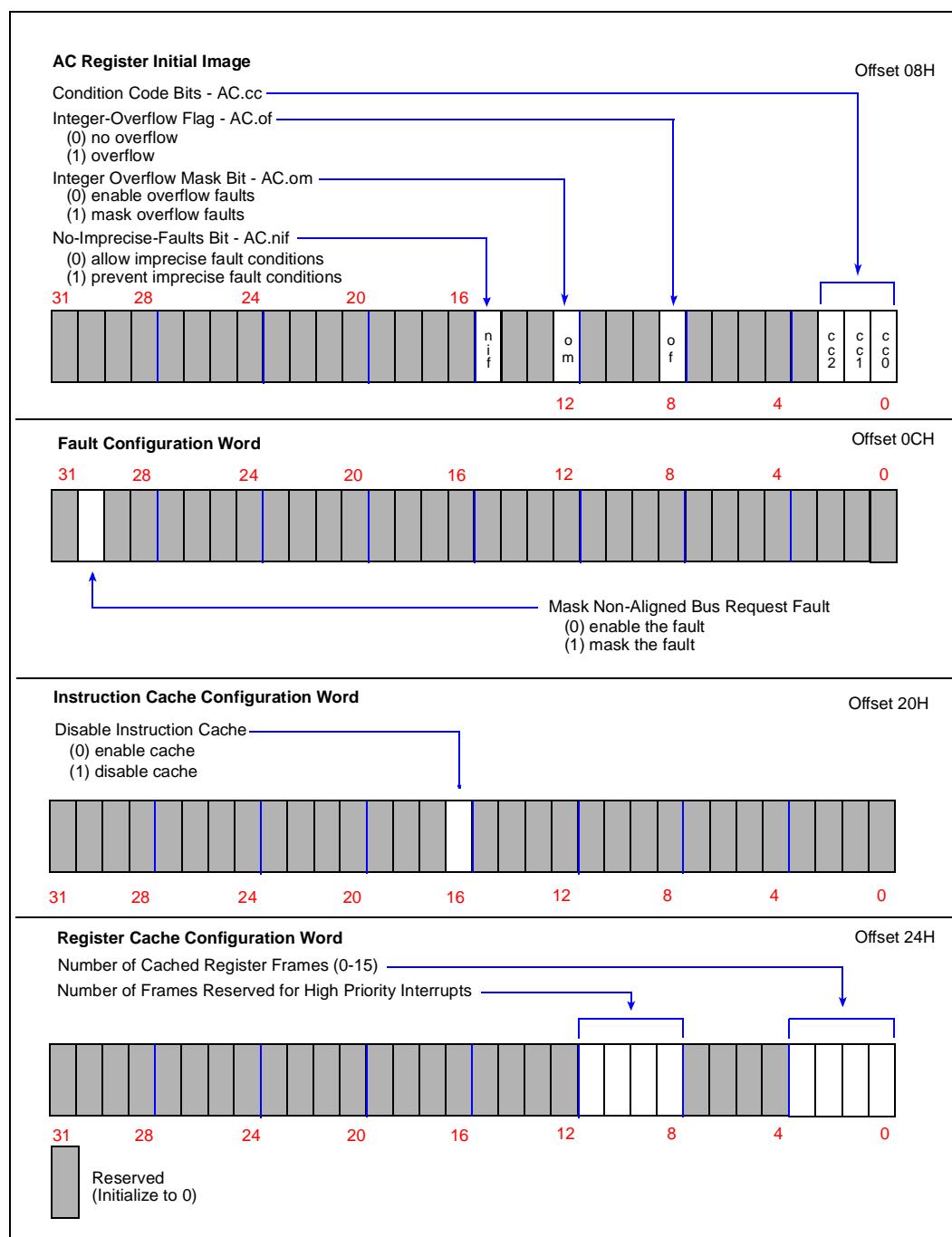
The PRCB contains base addresses for system data structures and initial configuration information for the core and integrated peripherals. Software can access the base addresses from these internal registers through the memory-mapped interface. Upon reset or reinitialization, the processor initializes these registers. The PRCB format is shown in [Table 13-6](#).

**Table 13-6. 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, load with zero
PRCB POINTER + 1CH	Interrupt Stack Pointer
PRCB POINTER + 20H	Instruction Cache Configuration Word
PRCB POINTER + 24H	Register Cache Configuration Word

The processor programs the initial configuration information in the arithmetic controls (AC) initial image, the fault configuration word, the instruction cache configuration word, and the register cache configuration word. [Figure 13-6](#) shows these configuration words.

Figure 13-6. Process Control Block Configuration Words



### 13.3.2 Process PRCB Flow

The following pseudo-code flow illustrates the PRCB processing. Note that this flow is used for both initialization and reinitialization (through **sysctl**).

#### Example 13-2. Process PRCB 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 and Cache NMI Vector Entry in Data RAM**/
    Reset_block_NMI;
    interrupt_table = memory[PRCB_mmr+0x10];
    memory[0] = memory[interrupt_table + (248*4) + 4];
    /** Process System Procedure Table **/
    sysproc = memory[PRCB_mmr+0x14];
    temp = memory[sysproc+0xc];
    SSP_mmr = (0x3) & temp;
    SSP.te = 1 & temp;
    /** 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);
    /** Configure Local Register Cache **/
    programmed_limit = (15 & memory[PRCB_mmr+0x24]) );
    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_control_table(ctrl_table+0x68 , ctrl_table+0x6c);
    /** 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;
}

```

### 13.3.2.1 AC Initial Image

The AC initial image loads into the on-chip AC register during initialization. The AC initial image allows selection of the initial value of the overflow mask, the no imprecise faults (AC.nif) bit and the condition code bits at initialization.

Applications can use the AC initial image condition code bits to specify the source of an initialization or reinitialization when a single instruction entry point to the user startup 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 startup code can detect the condition code values and thus the source of the reinitialization by using the compare or compare-and-branch instructions.

### 13.3.2.2 Fault Configuration Word

The fault configuration word lets the user mask the OPERATION.UNALIGNED fault when an unaligned memory request is issued. (See [Section 14.3.2, “Bus Transactions across Region Boundaries” on page 14-10](#) for a description of unaligned memory requests.) Whenever the processor encounters an unaligned access, it *always* performs the access, then determines whether or not it should generate a fault. If bit 30 in the fault configuration word is set, no fault is generated. If bit 30 is clear, the processor generates a fault after performing an unaligned memory request.

### 13.3.2.3 Instruction Cache Configuration Word

The instruction cache configuration word lets the user enable or disable the instruction cache at initialization. Setting bit 16 in the instruction cache configuration word disables the instruction cache and directs all instruction fetches 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:

- The processor reinitializes with a new value in the instruction cache configuration word
- Software issues an **icctl** instruction with the enable instruction cache operation
- Software issues a **sysctl** instruction with the configure-instruction-cache-message type and a cache configuration mode other than disable cache

### 13.3.2.4 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). See [Figure 13-6](#).

The register cache configuration word also specifies the number of additional register frames that can be cached on-chip, up to a total of 15 register frames. The default register cache can contain five register frames. Bits 3-0 of the register cache word expand the cache size by another 0 to 10 frames.

The additional register cache space comes at the expense of on-chip general purpose RAM space. The register cache grows downward into the RAM space from the highest RAM address. For example, the first additional register cache expansion consumes addresses 0000 07C0H through 0000 07FFH of RAM. [Table 13-7](#) illustrates the trade-offs between register cache size and available general purpose RAM space.

**Table 13-7. Register Cache Size vs. Available General Purpose RAM**

Register Cache Word Bits 3-0 (Hex)	On-Chip Register Cache Size (Frames)	Available General RAM (Hex)
0	0	0000 0000H to 0000 07FFH
1	1	
2	2	
3	3	
4	4	
5	5	
6	6	0000 0000H to 0000 07BFH
7	7	0000 0000H to 0000 077FH
8	8	0000 0000H to 0000 073FH
9	9	0000 0000H to 0000 06FFH
A	10	0000 0000H to 0000 06BFH
B	11	0000 0000H to 0000 067FH
C	12	0000 0000H to 0000 063FH
D	13	0000 0000H to 0000 05FFH
E	14	0000 0000H to 0000 05BFH
F	15	0000 0000H to 0000 057FH

Once allocated, register cache space fills starting at the lowest address. For example, if three additional register cache frames are reserved in RAM, the first frame will be cached beginning at address 0000 0740H. Therefore, user software cannot expect to use RAM addresses allocated to the register cache.

User software is responsible for preventing register cache corruption due to writes to RAM.

The register cache and the configuration word are explained further in [Section 4.2, “Local Register Cache”](#) on page [4-3](#).

### 13.3.3 Control Table

The control table is the data structure that contains the on-chip control register values. It is automatically loaded during initialization and must be completely constructed in the IMI. [Figure 13-7](#) shows the Control Table format.

For register bit definitions of the on-chip control table registers, see the following:

- IMAP — Figure 11-9., “Interrupt Mapping (IMAP0-IMAP2) Registers” (pg. 11-22)
- ICON — Figure 11-8., “Interrupt Control (ICON) Register” (pg. 11-20)
- PMCON — Figure 14-2., “PMCON Register Bit Descriptions” (pg. 14-8)
- TC — Figure 9-1., “Trace Controls (TC) Register” (pg. 9-2)
- BCON — Figure 14-3., “Bus Control Register (BCON)” (pg. 14-10)

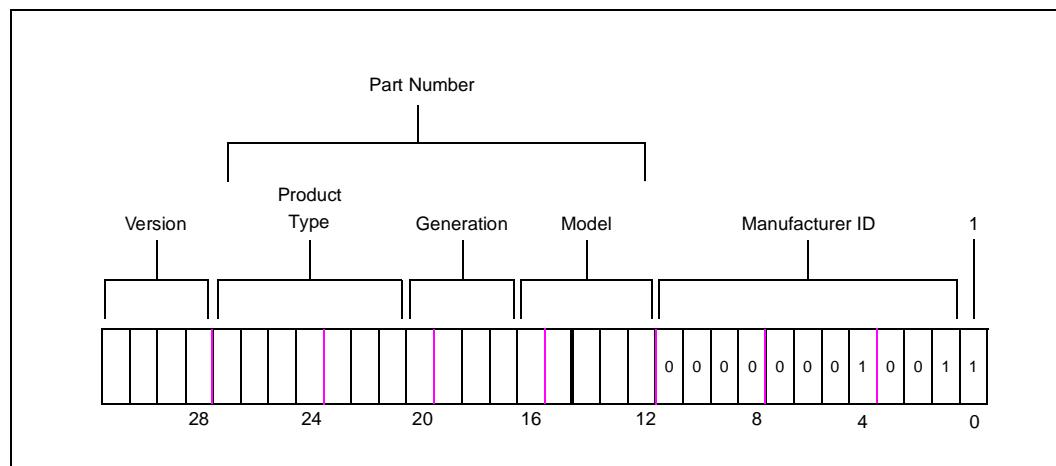
[Figure 13-7. Control Table](#)

31	0
Reserved (Initialize to 0)	00H
Interrupt Map 0 (IMAP0)	04H
Interrupt Map 1 (IMAP1)	08H
Interrupt Map 2 (IMAP2)	0CH
Interrupt Control (ICON)	10H
Physical Memory Region 0 Configuration (PMCON0)	14H
Physical Memory Region 1 Configuration (PMCON0)	18H
Physical Memory Region 2 Configuration (PMCON0)	0CH
Physical Memory Region 3 Configuration (PMCON0)	20H
Physical Memory Region 4 Configuration (PMCON0)	24H
Physical Memory Region 5 Configuration (PMCON0)	28H
Physical Memory Region 6 Configuration (PMCON0)	2CH
Physical Memory Region 7 Configuration (PMCON0)	30H
Physical Memory Region 8 Configuration (PMCON0)	34H
Physical Memory Region 9 Configuration (PMCON0)	38H
Physical Memory Region 10 Configuration (PMCON0)	3CH
Physical Memory Region 11 Configuration (PMCON0)	40H
Physical Memory Region 12 Configuration (PMCON0)	44H
Physical Memory Region 13 Configuration (PMCON0)	48H
Physical Memory Region 14 Configuration (PMCON0)	4CH
Physical Memory Region 15 Configuration (PMCON0)	50H
Reserved (Initialize to 0)	54H
Trace Controls (TC)	58H
Bus Configuration Control (BCON)	5CH
	60H
	64H
	68H
	6CH

## 13.4 Device Identification on Reset

The DEVICEID memory-mapped register contains a number characterizing the microprocessor type and stepping. During initialization, the processor places the DEVICEID register value into g0.

**Figure 13-8. IEEE 1149.1 Device Identification Register**



The value for device identification is compliant with the IEEE 1149.1 specification and Intel standards. **Table 13-8** describes the fields of the device ID. The Version field corresponds to silicon stepping; for example, 0000 refers to the A-0 stepping.

**Table 13-8. Fields of IEEE 1149.1 Device ID**

Field	Value	Definition
Version	0000 = A0 step	Used to indicate major steppings.
V <sub>CC</sub>	1 = 3.3 volt device	Indicates that the device uses a V <sub>CC</sub> of 3.3 volt
Product Type	00 0100	Designates type of product (Indicates i960 CPU)
Generation Type	0000 = reserved 0010 = H-series	Indicates the generation (or series) that the product belongs to.
Model	00000 = 1x core clock (HA) 00001 = 2x core clock (HD) 00010 = 3x core clock (HT)	Indicates member within series and specific model information.
Manufacturer ID	000 0000 1001	Manufacturer ID assigned by IEEE (Indicates Intel)

### 13.4.1 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 6-108](#) 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 13.3.1.2, “Process Control Block \(PRCB\)” on page 13-17](#).

Reinitialization is useful for relocating data structures to RAM after initialization. The interrupt table must be located in RAM to allow the processor to post software-generated interrupts by writing to the interrupts table’s pending priorities and pending interrupts fields. It may also be necessary to relocate the control table to RAM. It must be in RAM if 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 then reinitializes with a new PRCB containing the base addresses of the new data structures in RAM.

Reinitialization is required to relocate any of the data structures listed below, since the processor caches the pointers to the structures. The processor caches these pointers during its initialization.

- Interrupt Table Address
- Fault Table Address
- System Procedure Table Address
- Control Table Address

To modify these data structures, software re-initialization is needed.

During software re-initialization, BCON.sirp may be set up to protect the first 64 bytes of internal data RAM from supervisor mode writes. This bit is cleared during software re-initialization to allow the NMI vector to be cached. The remaining BCON bits are not modified at this point. Later in the software re-initialization process, the entire BCON MMR is reloaded with the value from the Control Table. In addition to clearing the BCON.sirp bit, all GMU protection is disabled to prevent GMU faults.

## 13.5 Startup Code Example

After initialization, user startup code typically copies initialized data structures from ROM to RAM, reinitializes the processor, sets up the first stack frame, changes the execution state to non-interrupted and calls the `_main` routine. This section presents an example startup routine and associated header file. This simplified startup file can be used as a basis for more complete initialization routines.

The examples in this section are useful for creating and evaluating startup code. The following lists the example's number, name and page.

- [Example 13-3 “Initialization Header File \(init.h\)” on page 13-26](#)
- [Example 13-4 “Start-up Routine \(init.s\) \(Sheet 1 of 4\)” on page 13-27](#)
- [Example 13-5 “High-Level Start-up Code \(initmain.c\)” on page 13-30](#)
- [Example 13-6 “Control Table \(ctltbl.c\)” on page 13-31](#)
- [Example 13-7 “Initialization Boot Record File \(rom\\_ibr.c\)” on page 13-32](#)
- [Example 13-8 “Linker Directive File \(init.ld\)” on page 13-33](#)
- [Example 13-9 “Makefile” on page 13-34](#)

### Example 13-3. Initialization Header File (init.h)

```

/*-----*/
/*  init.h                                         */
/*-----*/

#define BYTE_N(n,data)  (((unsigned)(data) >> (n*8)) & 0xFF)

typedef struct
{
    unsigned char bus_byte_0;
    unsigned char reserved_0[3];
    unsigned char bus_byte_1;
    unsigned char reserved_1[3];
    unsigned char bus_byte_2;
    unsigned char reserved_2[3];
    unsigned char bus_byte_3;
    unsigned char reserved_3[3];
    void        (*first_inst)();
    unsigned   *prcb_ptr;
    int         check_sum[6];
}IBR;

/* PMCON Bus Width can be 8,16 or 32, default to 8
 * PMCON15 BOOT_BIG_ENDIAN 0=little endian, 1=big endian
 */
#define BUS_WIDTH(bw)    ((bw==16)?(1<<22):(0) | 
((bw==32)?(2<<22):(0)))
#define BOOT_BIG_ENDIAN (on)    ((on)?(1<<31:0))

/* Bus configuration */
#define DEFAULT  (BUS_WIDTH(8) | BOOT_BIG_ENDIAN(0))
#define I_O      (BUS_WIDTH(8) | BOOT_BIG_ENDIAN(0))
#define DRAM     (BUS_WIDTH(32)| BOOT_BIG_ENDIAN(0))
#define ROM      (BUS_WIDTH(8) | BOOT_BIG_ENDIAN(0))

```

**Example 13-4. Start-up Routine (init.s) (Sheet 1 of 4)**

```

/*-----*/
/*  init.s                                         */
/*-----*/

/* initial PRCB */

.globl _rom_prcb
.align 4
_rom_prcb:
.word boot_flt_table           # 0 - Fault Table
.word _boot_control_table      # 4 - Control Table
.word 0x00001000               # 8 - AC reg mask
                                overflow fault
.word 0x40000000               # 12 - Flt CFG
.word boot_intr_table          # 16 - Interrupt Table
.word rom_sys_proc_table       # 20 - System Procedure Table
.word 0                         # 24 - Reserved
.word _intr_stack              # 28 - Interrupt Stack
                                # Pointer
.word 0x00000000               # 32 - Inst. Cache
                                # - enable cache
.word 0x00000205               # 36 - Register Cache
                                # Configuration

/* ROM system procedure table */

.equ supervisor_proc, 2
.text
.align 6 /* or .align 2 or .align 4 */
rom_sys_proc_table:
.space 12                      # Reserved
.word _supervisor_stack         # Supervisor stack pointer
.space 32                      # Preserved
.word _default_sysproc          # sysproc 0
.word _default_sysproc          # sysproc 1
.word _default_sysproc          # sysproc 2
.word _default_sysproc          # sysproc 3
.word _default_sysproc          # sysproc 4
.word _default_sysproc          # sysproc 5
.word _default_sysproc          # sysproc 6
.word _fault_handler + supervisor_proc # sysproc 7
.word _default_sysproc          # sysproc 8
.space 251*4                    # sysproc 9-259

/* Fault Table */
.equ syscall, 2
.equ fault_proc, 7
.text
.align 4
boot_flt_table:
.word (fault_proc<<2) + syscall    # 0-Parallel Fault
.word 0x27f
.word (fault_proc<<2) + syscall    # 1-Trace Fault

```

**Example 13-4. Start-up Routine (init.s) (Sheet 2 of 4)**

```

.word 0x27f
.word (fault_proc<<2) + syscall    # 2-Operation Fault
.word 0x27f
.word (fault_proc<<2) + syscall    # 3-Arithmetic Fault
.word 0x27f
.word 0                                # 4-Reserved
.word 0
.word (fault_proc<<2) + syscall    # 5-Constraint Fault
.word 0x27f
.word 0                                # 6-Reserved
.word 0
.word (fault_proc<<2) + syscall    # 7-Protection Fault
.word 0x27f
.word 0                                # 8-Reserved
.word 0
.word 0                                # 9-Reserved
.word 0
.word (fault_proc<<2) + syscall    # 0xa-Type Fault
.word 0x27f
.space 21*8                            # reserved

/* Boot Interrupt Table */

.text
boot_intr_table:
.word 0                      # Pending Priorities
.word 0, 0, 0, 0, 0, 0, 0  # Pending Interrupts          Vectors
.word _intx, _intx, _intx, _intx, _intx, _intx, _intx # 8
.word _intx, _intx, _intx, _intx, _intx, _intx, _intx # 10
.word _intx, _intx, _intx, _intx, _intx, _intx, _intx # 18
.word _intx, _intx, _intx, _intx, _intx, _intx, _intx # 20
.word _intx, _intx, _intx, _intx, _intx, _intx, _intx # 28
.word _intx, _intx, _intx, _intx, _intx, _intx, _intx # 30
.word _intx, _intx, _intx, _intx, _intx, _intx, _intx # 38
.word _intx, _intx, _intx, _intx, _intx, _intx, _intx # 40
.word _intx, _intx, _intx, _intx, _intx, _intx, _intx # 48
.word _intx, _intx, _intx, _intx, _intx, _intx, _intx # 50
.word _intx, _intx, _intx, _intx, _intx, _intx, _intx # 58
.word _intx, _intx, _intx, _intx, _intx, _intx, _intx # 60
.word _intx, _intx, _intx, _intx, _intx, _intx, _intx # 68
.word _intx, _intx, _intx, _intx, _intx, _intx, _intx # 70
.word _intx, _intx, _intx, _intx, _intx, _intx, _intx # 78
.word _intx, _intx, _intx, _intx, _intx, _intx, _intx # 80
.word _intx, _intx, _intx, _intx, _intx, _intx, _intx # 88
.word _intx, _intx, _intx, _intx, _intx, _intx, _intx # 90
.word _intx, _intx, _intx, _intx, _intx, _intx, _intx # 98
.word _intx, _intx, _intx, _intx, _intx, _intx, _intx # a0
.word _intx, _intx, _intx, _intx, _intx, _intx, _intx # a8
.word _intx, _intx, _intx, _intx, _intx, _intx, _intx # b0
.word _intx, _intx, _intx, _intx, _intx, _intx, _intx # b8
.word _intx, _intx, _intx, _intx, _intx, _intx, _intx # c0
.word _intx, _intx, _intx, _intx, _intx, _intx, _intx # c8
.word _intx, _intx, _intx, _intx, _intx, _intx, _intx # d0

```

**Example 13-4. Start-up Routine (init.s) (Sheet 3 of 4)**

```

.word    _intx, _intx, _intx, _intx, _intx, _intx, _intx, _intx # d8
.word    _intx, _intx, _intx, _intx, _intx, _intx, _intx, _intx # e0
.word    _intx, _intx, _intx, _intx, _intx, _intx, _intx, _intx # e8
.word    _intx, _intx, _intx, _intx, 0, 0, 0, 0 # f0
.word    _nmi, 0, 0, 0, _intx, _intx, _intx, _intx # f8

/* START */
/* Processor starts execution here after reset. */
.text
.globl _start_ip
.globl _reinit
_start_ip:
    mov 0, g14           /* g14 must be 0 for ic960 C compiler */
/* MON960 requires copying the .data area into RAM. If a user application
* does not require this it is not necessary.
* Copy the .data into RAM. The .data has been packed in the ROM after the
* code area. If the copy is not needed (RAM-based monitor), the symbol
* rom_data can be defined as 0 in the linker directives file.
*/
    lda rom_data, g1      # load source of copy
    cmpobe 0, g1, lf
    lda __Bdata, g2        # load destination
    lda __Edata, g3
init_data:
    ldq (g1), r4
    addo 16, g1, g1
    stq r4, (g2)
    addo 16, g2, g2
    cmpobl g2, g3, init_data
1:
/* Initialize the BSS area of RAM. */
    lda __Bbss, g2          # start of bss
    lda __Ebss, g3          # end of bss
    movq 0,r4
bss_fill:
    stq r4, (g2)
    addo 16, g2, g2
    cmpobl g2, g3, bss_fill

_reinit:
    ldconst 0x300, r4        # reinitialize sys control
    lda lf, r5
    lda _ram_prcb, r6
    sysctl r4, r5, r6
1:
    lda _user_stack, pfp
    lda 64(pfp), sp
    mov pfp, fp             /* new fp */
    flushreg

    ldconst 0x001f2403, r3      /* PC mask */
    ldconst 0x000f0003, r4      /* PC value */

```

**Example 13-4. Start-up Routine (init.s) (Sheet 4 of 4)**

```

        modpc r3, r3, r4      /* Lower interrupt priority */
*/
/* Clear the IPND register */
    lda    0xff008500, g0
    mov    0, gl
    st     gl,(g0)
    callx _main           #to main routine

    .globl _intr_stack
    .globl _user_stack
    .globl _supervisor_stack
    .bss   _user_stack, 0x0200, 6      # default application stack
    .bss   _intr_stack, 0x0200, 6      # interrupt stack
    .bss   _supervisor_stack, 0x0600, 6 # fault (supervisor) stack

    .text
_fault_handler:
    ldconst 'F', g0
    call   _co
    ret

_default_sysproc:
    ret

_intx:
    ldconst 'I', g0
    call   _co
    ret

```

**Example 13-5. High-Level Start-up Code (initmain.c)**

```

unsigned componentid = 0;

main()
{
    /* system- or board-specific code goes here */
} /* this code is called by init.s */
co()
{
    /* system or board-specific output routine goes here */
}

```

**Example 13-6. Control Table (ctltbl.c)**

```

/*-----*/
/*  ctltbl.c                                     */
/*-----*/
#include "init.h"

typedef struct
{
    unsigned control_reg[28];
}CONTROL_TABLE;

const CONTROL_TABLE boot_control_table = {
    /* Reserved */
    0, 0, 0, 0,
    /* Interrupt Map Registers */
    0, 0, 0, /* Interrupt Map Regs (set by code as needed) */
    0x43bc, /* ICON
               *          - dedicated mode,
               *          - enabled
               * system_init 0 - falling edge activated,
               * system_init 1 - falling edge activated,
               * system_init 2 - falling edge activated,
               * system_init 3 - falling edge activated,
               * system_init 4 - level-low activated,
               * system_init 5 - falling edge activated,
               * system_init 6 - falling edge activated,
               * system_init 7 - falling edge activated,
               *          - mask unchanged,
               *          - not cached,
               *          - fast,
               */
    /* Physical Memory Configuration Registers */

    DEFAULT, 0, /* Region 0 */
    DEFAULT, 0, /* Region 1 */
    DEFAULT, 0, /* Region 2 */
    DEFAULT, 0, /* Region 3 */
    DEFAULT, 0, /* Region 4 */
    DEFAULT, 0, /* Region 5 */
    I_O, 0, /* Region 6 */
    DEFAULT, 0, /* Region 7 */
    DEFAULT, 0, /* Region 8 */
    DEFAULT, 0, /* Region 9 */
    DEFAULT, 0, /* Region 10 */
    DEFAULT, 0, /* Region 11 */
    DRAM, 0, /* Region 12 */
    DEFAULT, 0, /* Region 13 */
    DEFAULT, 0, /* Region 14 */
    ROM, 0, /* Region 15 */

    0, /* Reserved */
    0, /* Breakpoint Control */
    0, /* Trace Controls */
    1 /* Bus Configuration Control (Region config. valid) */
};

```

**Example 13-7. Initialization Boot Record File (rom\_ibr.c)**

```
#include "init.h"
/*
 * NOTE: The ibr must be located at 0xFFFFFFF30. Use the linker to
 * locate this structure.
 * The boot configuration is always region 15, since the IBR
 * must be located there
 */

extern void start_ip();
extern unsigned rom_prcb;
extern unsigned checksum;

#define CS_6 (int) &checksum /* value calculated in linker */
#define BOOT_CONFIG ROM

const IBR init_boot_record =
{
    BYTE_N(0,BOOT_CONFIG),      /* PMCON15 byte 1 */
    0,0,0,                      /* reserved set to 0 */
    BYTE_N(1,BOOT_CONFIG),      /* PMCON15 byte 2 */
    0,0,0,                      /* reserved set to 0 */
    BYTE_N(2,BOOT_CONFIG),      /* PMCON15 byte 3 */
    0,0,0,                      /* reserved set to 0 */
    BYTE_N(3,BOOT_CONFIG),      /* PMCON15 byte 4 */
    0,0,0,                      /* reserved set to 0 */
    start_ip,
    &rom_prcb,
    -2,
    0,
    0,
    0,
    CS_6
};
```

**Example 13-8. Linker Directive File (init.ld)**

```

/*-----*/
/*  init.ld
/*-----*/

MEMORY
{
    /*
        Enough space must be reserved in ROM after the text
        section to hold the initial values of the data section.
    */
    rom:      o=0xfefefe0000,l=0x1fc00
    rom_dat: o=0xfeffffc00,l=0x0300 /* placeholder for .data image */

    ibr:      o=0xfffffff30,l=0x0030
    data:     o=0xa0000000,l=0x0300
    bss:      o=0xa0000300,l=0x7d00
}
SECTIONS
{
    .ibr :
    {
        rom_ibr.o
    } > ibr

    .text :
    {
    } > rom

    .data :
    {
    } > data

    .bss :
    {
    } > data
}

rom_data = __Etext; /* used in init.s as source of .data
                     section initial values. ROM960
                     "move" command places the .data
                     section right after the .text section */

_checksum = -(_rom_prcb + _start_ip);

HLL()

/*Rommer script embedded here: the following creates a ROM image
**move $0 .text 0
**move $0
**move $0 .ibr 0x1ff30
**mkimage $0 $0.ima
**ihex $0.ima $0.hex mode16
**map $0
**quit
*/

```

**Example 13-9. Makefile**

```
/*-----*/
/*  makefile
/*-----*/
LDFILE = init
FINALOBJ = init
OBJS = init.o ctltbl.o initmain.o
IBR = rom_ibr.o
LDFLAGS = -AJF -Fcoff -T$(LDFILE) -m
ASFLAGS = -AJF -V
CCFLAGS = -AJF -Fcoff -V -c
init.ima: $(FINALOBJ)
        rom960 $(LDFILE) $(FINALOBJ)

init: $(OBJS) $(IBR)
        gld960 $(LDFLAGS) -o $< $(OBJS)

.s.o:
        gas960c $(ASFLAGS) $<

.c.o:
        gcc960 $(CCFLAGS) $<
```

## 13.6 System Requirements

The following sections discuss generic hardware requirements for a system built around the i960 Hx processor. This section describes electrical characteristics of the processor's interface to the external circuit. The CLKIN, RESET#, STEST, FAIL#, ONCE#, V<sub>SS</sub> and V<sub>CC</sub> pins are described in detail. Specific signal functions for the external bus signals and interrupt inputs are discussed in their respective sections in this manual.

### 13.6.1 Input Clock (CLKIN)

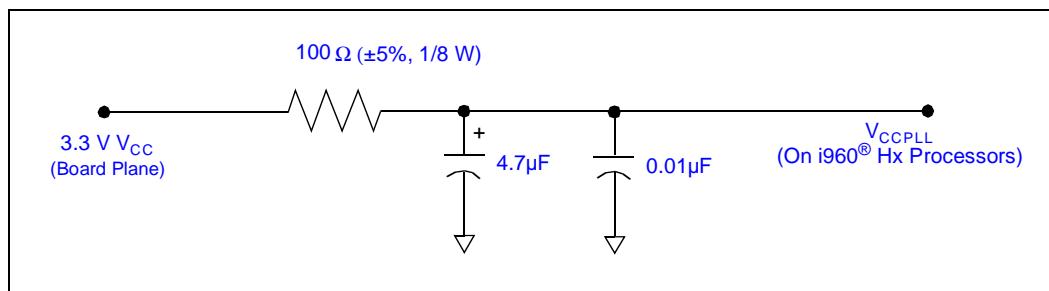
The clock input (CLKIN) determines processor execution rate and timing. It is designed to be driven by most common TTL crystal clock oscillators. The clock input must be free of noise and conform with the specifications listed in the data sheet. CLKIN input capacitance is minimal; for this reason, it may be necessary to terminate the CLKIN circuit board trace to reduce overshoot and undershoot.

### 13.6.2 Power and Ground Requirements (V<sub>CC</sub>, V<sub>SS</sub>)

The i960 Hx processor is designed for 3.3 V V<sub>CC</sub> and is compatible with typical 5 V TTL signals.

The large number of V<sub>SS</sub> and V<sub>CC</sub> pins reduces the impedance of power and ground connections to the chip effectively and reduces transient noise induced by current surges. The i960 Hx processor is implemented in CMOS technology. Unlike NMOS processes, power dissipation in the CMOS process is due to capacitive charging and discharging on-chip and in the processor's output buffers; there is almost no DC power component. The nature of this power consumption results in current surges when capacitors charge and discharge. The processor's power consumption depends mostly on frequency. It also depends on voltage and capacitive bus load (see the *80960HA/HD/HT Embedded 32-bit Microprocessor* datasheet).

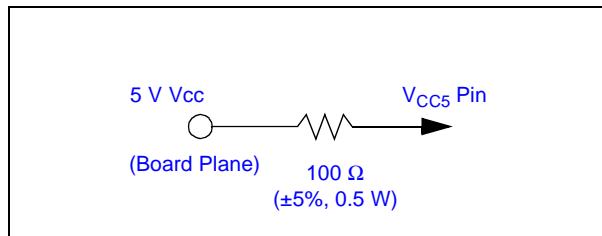
To reduce clock skew, i960 Hx processor isolates the V<sub>CCPLL</sub> pin for the Phase Lock Loop (PLL) circuit on the pinout. The lowpass filter, as shown in [Figure 13-9](#), reduces noise-induced clock jitter and its effects on timing relationships in system designs. The 4.7  $\mu$ F capacitor must be a low ESR solid tantalum, the 0.01  $\mu$ F capacitor must be of the type X7R and the node connecting V<sub>CCPLL</sub> must be as short as possible.

**Figure 13-9. V<sub>CCPLL</sub> Lowpass Filter**

### 13.6.3 $V_{CC5}$ Pin Requirements

In mixed voltage systems that drive the i960 Hx processor inputs in excess of  $3.3\ V$ , the  $V_{CC5}$  pin must be connected to the system's  $5\ V$  supply. To limit current flow into the  $V_{CC5}$  pin, there is a limit to the voltage differential between the  $V_{CC5}$  pin and the other  $V_{CC}$  pins. The voltage differential ( $V_{DIFF}$ ) between the 80960Hx  $V_{CC5}$  pin and its  $3.3\ V\ V_{CC}$  pins should never exceed  $2.25\ V$ . This limit applies to power up, power down, and steady-state operation. See the *80960HA/HD/HT Embedded 32-bit Microprocessor* datasheet for more details.

If the voltage difference requirements cannot be met due to system design limitations, an alternate solution may be employed. As shown in [Figure 13-10](#), a minimum of  $100\ \Omega$  series resistor may be used to limit the current into the  $V_{CC5}$  pin. This resistor ensures that current drawn by the  $V_{CC5}$  pin does not exceed the maximum rating for this pin.

**Figure 13-10.  $V_{CC5}$  Current-Limiting Resistor**

This resistor is not necessary in systems that can guarantee the  $V_{DIFF}$  specification. In  $3.3\ V$ -only systems and systems that drive the i960 Hx processor pins from  $3.3\ V$  logic, connect the  $V_{CC5}$  pin directly to the  $3.3\ V\ V_{CC}$  plane.

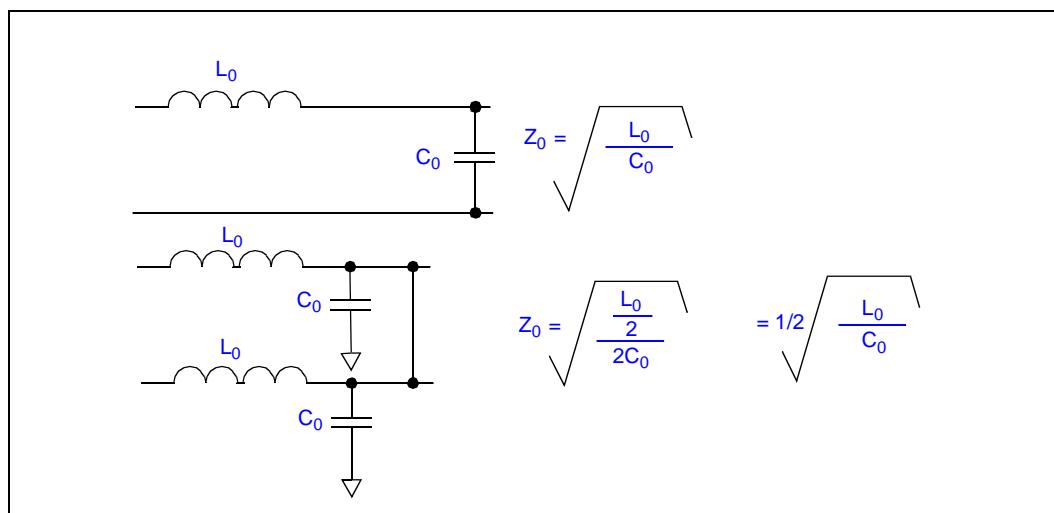
### 13.6.4 Power and Ground Planes

Power and ground planes must be used in i960 Hx processor systems to minimize noise. Justification for these power and ground planes is the same as for multiple V<sub>SS</sub> and V<sub>CC</sub> pins. Power and ground lines have inherent inductance and capacitance, therefore, an impedance  $Z = (L/C)^{1/2}$ .

Total characteristic impedance for the power supply can be reduced by adding more lines. This effect is illustrated in [Figure 13-11](#), which shows that two lines in parallel have half the impedance of one. Ideally, a plane — an infinite number of parallel lines — results in the lowest impedance. Fabricate power and ground planes with a 1 oz. copper for outer layers and 0.5 oz. copper for inner layers.

All power and ground pins must be connected to the planes. Ideally, the i960 Hx processor should be located at the center of the board to take full advantage of these planes, simplify layout and reduce noise.

**Figure 13-11. Reducing Characteristic Impedance**



### 13.6.5 Decoupling Capacitors

Decoupling capacitors placed across the processor between V<sub>CC</sub> and V<sub>SS</sub> reduce voltage spikes by supplying the extra current needed during switching. Place these capacitors close to the device because connection line inductance negates their effect. Also, for this reason, the capacitors should be low inductance. Chip capacitors (surface mount) exhibit lower inductance.

## 13.6.6 I/O Pin Characteristics

The i960 Hx processor interfaces to its system through its pins. This section describes the general characteristics of the input and output pins.

### 13.6.6.1 Output Pins

All output pins on the i960 Hx processor are three-state outputs. Each output can drive a logic 1 (low impedance to  $V_{CC}$ ); a logic 0 (low impedance to  $V_{SS}$ ); or float (present a high impedance to  $V_{CC}$  and  $V_{SS}$ ). Each pin can drive an appreciable external load. The *80960HA/HD/HT Embedded 32-bit Microprocessor* datasheet describes each pin's drive capability and provides timing and derating information to calculate output delays based on pin loading.

### 13.6.6.2 Input Pins

All i960 Hx processor inputs are designed to detect TTL thresholds, providing compatibility with the vast amount of available random logic and peripheral devices that use TTL outputs.

Most i960 Hx processor inputs are synchronous inputs (Table 13-9). A synchronous input pin must have a valid level (TTL logic 0 or 1) when the value is used by internal logic. If the value is not valid, it is possible for a metastable condition to be produced internally resulting in indeterminate behavior. The *80960HA/HD/HT Embedded 32-bit Microprocessor* datasheet specifies input valid setup and hold times relative to CLKIN for the synchronized inputs.

**Table 13-9. Input Pins**

Synchronous Inputs (sampled by CLKIN)	Asynchronous Inputs (sampled by CLKIN)	Asynchronous Inputs (sampled by RESET#)
D[31:0] RDYRCV# HOLD TDI TMS	RESET# XINT[7:0]# NMI#	STEST LOCK# ONCE#

i960 Hx processor inputs that are considered asynchronous are internally synchronized to the rising edge of CLKIN. Since they are internally synchronized, the pins need to be held only long enough for proper internal detection. In some cases, it is useful to know if an asynchronous input will be recognized on a particular CLKIN cycle or held off until a following cycle. The *80960HA/HD/HT Embedded 32-bit Microprocessor* datasheet provides setup and hold requirements relative to CLKIN which ensure recognition of an asynchronous input. The data sheet also supplies hold times required for detection of asynchronous inputs.

The ONCE# and STTEST inputs are asynchronous inputs. These signals are sampled and latched on the rising edge of the RESET# input instead of CLKIN.

## 13.6.7 High Frequency Design Considerations

At high signal frequencies and/or with fast edge rates, the transmission line properties of signal paths in a circuit must be considered. Transmission line effects and crosstalk become significant in comparison to the signals. These errors can be transient and therefore difficult to debug. The following sections of this chapter discuss some high-frequency design issues. For more information, consult a reference on high-frequency design.

### 13.6.7.1 Line Termination

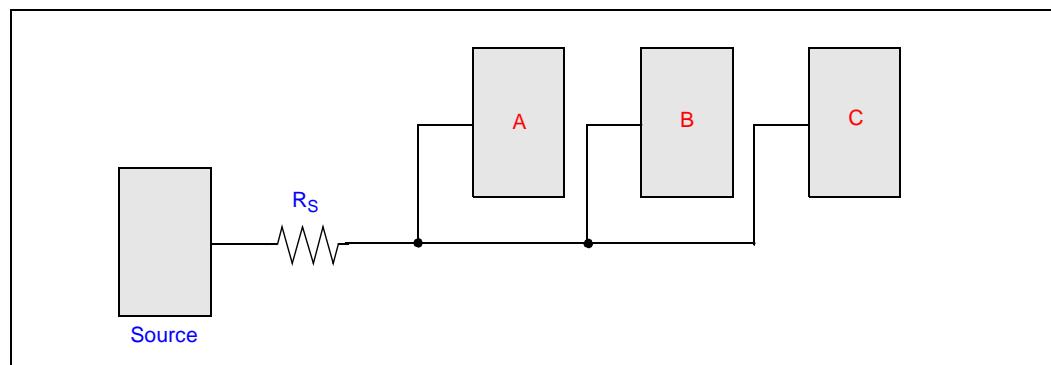
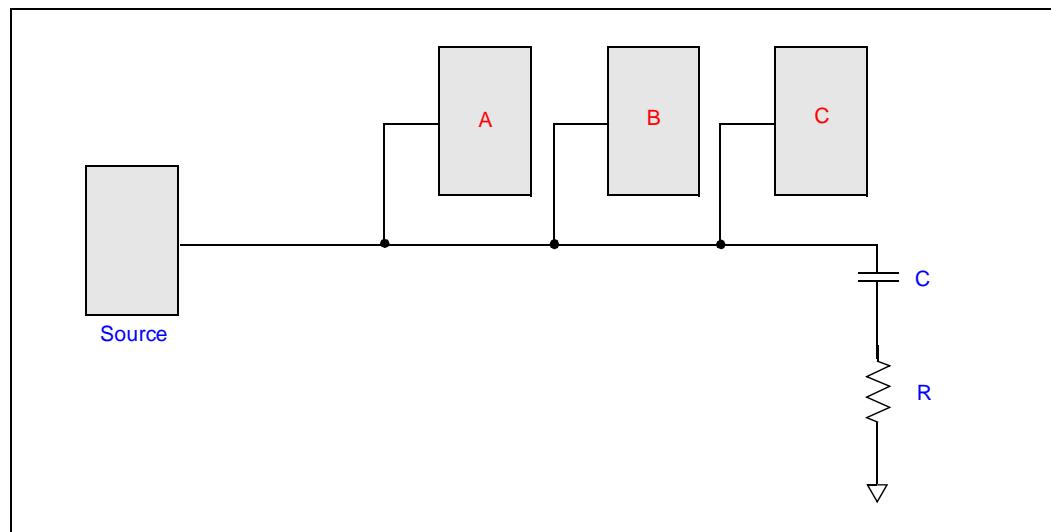
Input voltage level violations are usually due to voltage spikes that raise input voltage levels above the maximum limit (overshoot) and below the minimum limit (undershoot). These voltage levels can cause excess current on input gates, resulting in permanent damage to the device. Even if no damage occurs, many devices are not guaranteed to function as specified if input voltage levels are exceeded.

Signal lines are terminated to minimize signal reflections and prevent overshoot and undershoot. Terminate the line if the round-trip signal path delay is greater than signal rise or fall time. If the line is not terminated, the signal reaches its high or low level before reflections have time to dissipate and overshoot or undershoot occurs.

For the i960 Hx processor, two termination methods are attractive: AC and series. An AC termination matches the impedance of the trace, thereby eliminating reflections due to the impedance mismatch.

Series termination decreases current flow in the signal path by adding a series resistor as shown in [Figure 13-12](#). The resistor increases signal rise and fall times so that the change in current occurs over a longer period of time. Because the amount of voltage overshoot and undershoot depends on the change in current over time ( $V = L \frac{di}{dt}$ ), the increased time reduces overshoot and undershoot. Place the series resistor as close as possible to the signal source. AC termination is effective in reducing signal reflection (ringing). This termination is accomplished by adding an RC combination at the signal's farthest destination ([Figure 13-13](#)). While the termination provides no DC load, the RC combination damps signal transients.

Selection of termination methods and values is dependent upon many variables, such as output buffer impedance, board trace impedance and input impedance.

**Figure 13-12. Series Termination****Figure 13-13. AC Termination**

### 13.6.7.2 Latchup

Latchup is a condition in a CMOS circuit in which  $V_{CC}$  becomes shorted to  $V_{SS}$ . Intel's CMOS processes are immune to latchup under normal operation conditions. Latchup can be triggered when the voltage limits on I/O pins are exceeded, causing internal PN junctions to become forward biased. The following guidelines help prevent latchup:

- Observe the maximum rating for input voltage on I/O pins.
- Never apply power to an i960 Hx processor pin or a device connected to an i960 Hx processor pin before applying power to the processor itself.
- Prevent overshoot and undershoot on I/O pins by adding line termination and by designing to reduce noise and reflection on signal lines.

### 13.6.7.3 Interference

Interference is the result of electrical activity in one conductor that causes transient voltages to appear in another conductor. Interference increases with the following factors:

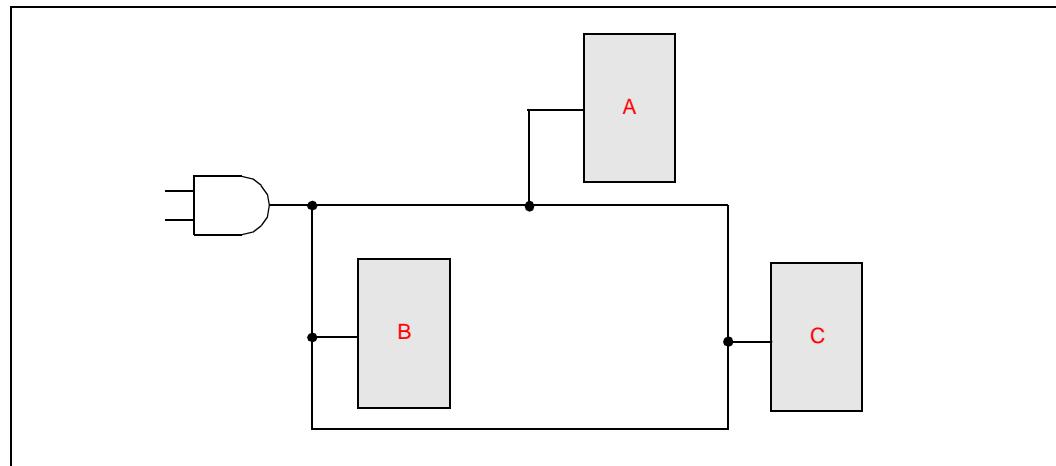
- Frequency interference is the result of changing currents and voltages. The more frequent the changes, the greater the interference.
- Closeness-of-conductors interference is due to electromagnetic and electrostatic fields whose effects are weaker further from the source.

Two types of interference must be considered in high-frequency circuits: electromagnetic interference (EMI) and electrostatic interference (ESI).

EMI is caused by the magnetic field that exists around any current-carrying conductor. The magnetic flux from one conductor can induce current in another conductor, resulting in transient voltage. Several precautions can minimize EMI:

- Run ground lines between two adjacent lines wherever they traverse a long section of the circuit board. The ground line should be grounded at both ends.
- Run ground lines between the lines of an address bus or a data bus if either of the following conditions exists:
  - The bus is on an external layer of the board.
  - The bus is on an internal layer but not sandwiched between power and ground planes that are at most 10 mils away.

**Figure 13-14. Avoiding Closed-Loop Signal Paths**



ESI is caused by the capacitive coupling of two adjacent conductors. The conductors act as the plates of a capacitor; a charge built up on one induces the opposite charge on the other.

The following steps reduce ESI:

- Separate signal lines so that capacitive coupling becomes negligible.
- Run a ground line between two lines to cancel the electrostatic fields.



The Bus Control Unit (BCU) includes logic to control many common types of memory and I/O subsystems. Every bus access is formatted according to the contents of the BCU control registers. The programming model for the BCU is consistent across the i960<sup>®</sup> Hx processor and i960 Jx processor families; however, not all capabilities are present in each device. The i960 Hx processor's BCU programming model differs from schemes used in other i960 processors.

## 14.1 Memory Attributes

Every location in memory has associated physical and logical attributes. For example, a specific location may have the following attributes:

- **Physical:** Memory is an 8-bit wide ROM
- **Logical:** Memory is ordered big-endian and data is non-cacheable

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 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.

### 14.1.1 Physical Memory Attributes

Programmable physical memory attributes for the i960 Hx processor include the following:

- Wait-state profile
- Parity enable/disable
- Parity sense (even/odd)
- Burst capability (can burst/cannot burst)
- Address read pipelining capability (can pipeline/cannot pipeline)
- READY# and BTERM# pin operation
- Bus width (8-, 16- or 32-bits wide)

For the purposes of assigning memory attributes, the physical address space is partitioned into 16 independently programmable, fixed 256-Mbyte regions determined by the upper four address bits. Region 0 maps to addresses 0000 0000H to 0FFF FFFFH and region 15 maps to addresses

F000 0000H to FFFF FFFFH. The physical memory attributes for each region are programmable through the Physical Memory Configuration (PMCON) registers. The PMCON registers are loaded from the Control Table. The i960 Hx microprocessor provides one PMCON register for each region. The descriptions of the PMCON registers and instructions on programming them are found in [Section 14.2](#).

#### 14.1.1.1 Data Bus Width

The i960 Hx processors allow an 8-, 16- or 32-bit wide data bus for each region. Byte enable signals encoded in each region provide the proper address for 8-, 16- or 32-bit memory systems. The i960 Hx processors use the lower order data lines when reading and writing to 8- or 16-bit memory. The unused upper order bits are undefined.

#### 14.1.1.2 Burst Accesses

When burst access is enabled, the bus controller generates an address followed by one to four data transfers. The lower two address bits are incremented for each consecutive data transfer. Burst accesses facilitate the interface to memories that support burst-mode data transfers. Wait states following the address cycle and wait states between data cycles can be controlled independently. Data cycle time is typically a fraction of address cycle time. For example, bursting provides an optimal wait state profile for fast page mode DRAM.

#### 14.1.1.3 Pipelined Read Accesses

When address pipelining is enabled, the next read address is asserted in the last data cycle of the current read access. Pipelining makes the address cycle transparent for back-to-back read accesses. There is no length limit to address pipelining.

#### 14.1.1.4 Wait States

A wait state generator within the bus controller generates wait states for memory accesses. For many memory interfaces, the internal wait state generator eliminates the necessity to externally generate a memory ready signal.

Typically, extra clock cycles — wait states — are associated with each data cycle. Wait states provide the required access times for external memory or peripherals. Five parameters, programmed for each region define wait state generator operation. These parameters are:

$N_{RAD}$	Number of wait cycles for Read Address-to-Data. The number of wait states between address cycle and first read data cycle. Programmable for 0-31 wait states.
$N_{RDD}$	Number of wait cycles for Read Data-to-Data. The number of wait states between consecutive data cycles of a burst read. Programmable for 0-3 wait states.
$N_{WAD}$	Number of wait cycles for Write Address-to-Data. The number of wait states that data is held after the address cycle and before the first write data cycle. Programmable for 0-31 wait states.
$N_{WDD}$	Number of wait cycles for Write Data-to-Data. The number of wait states that data is held between consecutive data cycles of a burst write. Programmable for 0-3 wait states.

$N_{XDA}$	Number of wait cycles for read or write Data-to-Address. The minimum number of wait states between the last data cycle of a bus access to the address cycle of the next bus access. $N_{XDA}$ applies to write and non-pipelined read requests. Programmable for 0-15 bus clocks.
-----------	---

$N_{RAD}$  and  $N_{WAD}$  describe address-to-data wait states.  $N_{RDD}$  and  $N_{WDD}$  specify the number of wait states between consecutive data when burst mode is enabled.  $N_{RDD}$  and  $N_{WDD}$  are used only in memory regions where bursting is enabled.

$N_{XDA}$  describes the number of wait states between consecutive bus requests.  $N_{XDA}$  is the bus turnaround time. An external device's ability to relinquish the bus on a read access (read deasserted to data float) determines the number of  $N_{XDA}$  cycles.

Note that for pipelined read accesses, the bus controller uses a value of zero for  $N_{XDA}$ , regardless of the parameter's programmed value. A non-zero  $N_{XDA}$  value defeats the purpose of pipelining. The programmed value of  $N_{XDA}$  is used for write requests to pipelined memory regions, as the i960 Hx processor does not support pipelined write accesses.

#### 14.1.1.5 READY# and BTERM# Pin Operation

The ready (READY#) and burst terminate (BTERM#) inputs dynamically control bus accesses. These inputs are enabled or disabled for each memory region. READY# extends accesses by forcing wait states. BTERM# allows a burst access to be broken into multiple accesses. The PMCON registers are programmed to enable or disable these inputs for each region.

READY# and BTERM# work with the programmed internal wait state counter. If READY# and BTERM# are enabled in a region, these pins are sampled only after the programmed number of wait states expire. If the inputs are disabled in a region, the inputs are ignored and the internal wait state counter alone determines access wait states. Refer to [Section 15.3.1, “Wait States” on page 15-8](#) for details on the operation of the READY# and BTERM# inputs.

Note that when pipelining is enabled, the bus controller ignores the READY# and BTERM# inputs for read requests. Since the i960 Hx processor does not support write pipelining, READY# and BTERM# are sampled for writes.

#### 14.1.1.6 Data Bus Parity

Data bus parity is enabled for a region by setting the Parity Enable bit (PMCON.pen) in the PMCON register. The parity sense, even or odd, is controlled by the Parity Sense (PMCON.podd) bit. Parity is always generated on write accesses. For read accesses, the processor checks parity only when the PMCON.pen bit is set.

When even parity is selected, the parity bit for each byte is set to a value that makes the total number of “1” bits in that byte *even*. When odd parity is selected, the parity bit for each byte is set to a value that makes the total number of “1” bits in that byte *odd*.

#### 14.1.2 Logical Memory Attributes

i960 Hx processor logical memory attributes include:

- Byte ordering (little endian/big endian).
- Data access cacheability (cacheable/non-cacheable).
- Rapid invalidation (whether data from that logical region can be quickly invalidated from the Data Cache without affecting the rest of the Data Cache).

The i960 Hx processor supports configuring several different logical memory regions within a single physical memory subsystem. For example, data within one area of DRAM may be non-cacheable while data in another area is cacheable. [Figure 14-1](#) 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. The Hx microprocessor has 15 pairs of LMCON registers. Each LMCON register pair defines a data template for memory areas that have common logical attributes. The extent of each data template is described by an address (aligned on 4-Kbyte boundaries) and an address mask. The address is programmed in the Logical Memory Address register (LMAR). The mask is programmed in the Logical Memory Mask register (LMMR). These two registers constitute the LMCON register pair.

The *Default Logical Memory Configuration* register is used to provide configuration data for areas of memory that do not fall within one of the 15 logical data templates.

The LMCON registers and their programming are described in [Section 14.4](#).

### 14.1.2.1 Byte Ordering

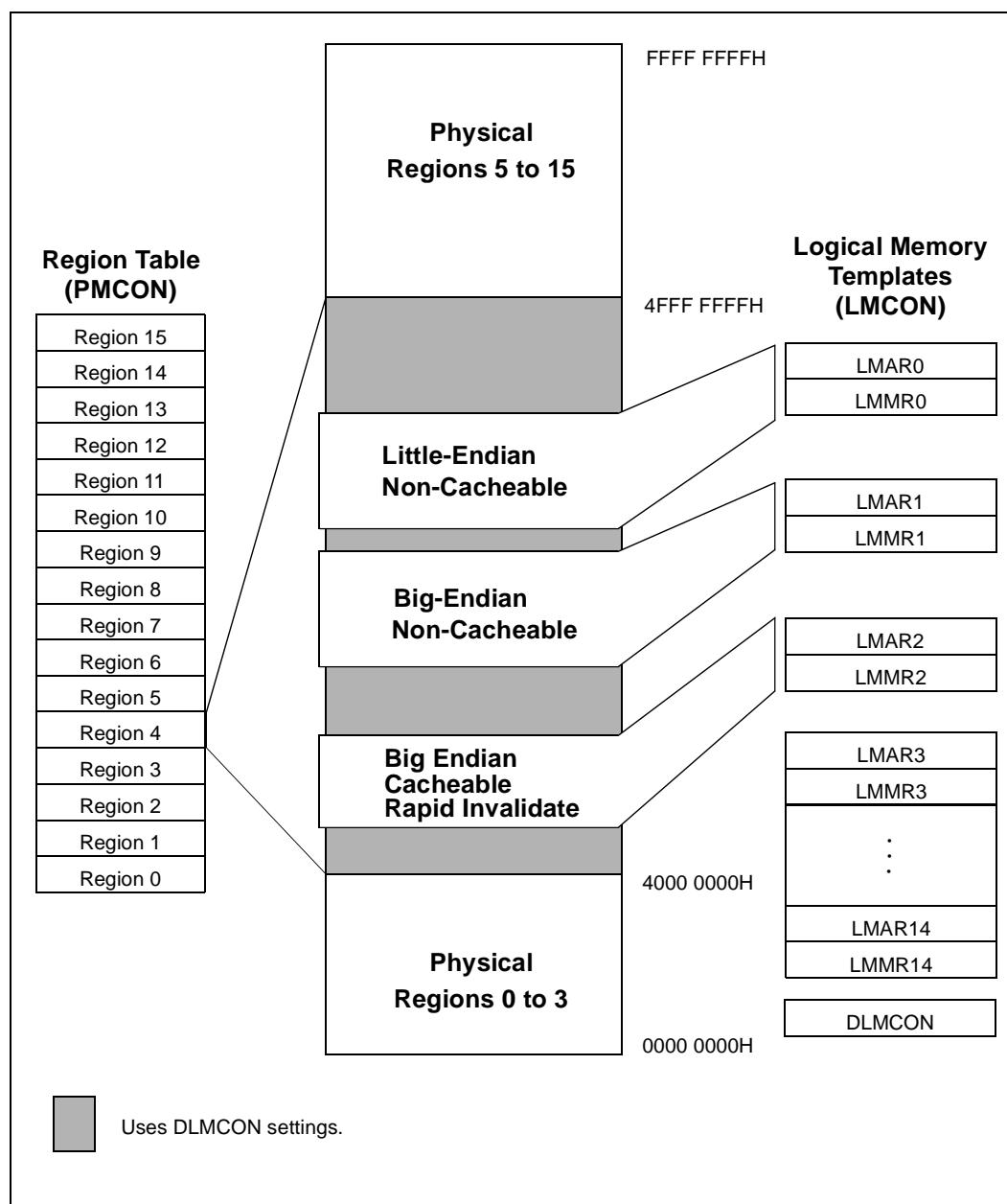
Byte ordering determines how data is read from or written to the bus and ultimately how data is stored in memory. Byte ordering can be individually selected for each logical memory region by setting a bit in the corresponding LMCN register. The bus controller supports big endian and little endian byte ordering for memory operations:

- |                      |  |
|----------------------|--|
| <i>little endian</i> | The controller reads or writes a data word's least-significant byte to the bus's eight least-significant data lines (D7:0). Little endian systems store a word's least-significant byte at the lowest byte address in memory. For example, if a little endian ordered word is stored at address 600, the least-significant byte is stored at address 600 and the most-significant byte at address 603. |
| <i>big endian</i>    | The controller reads or writes a data word's least-significant byte to the bus's eight most-significant data lines (D31:24). Big endian systems store the least-significant byte at the highest byte address in memory. So, if a big endian ordered word is stored at address 600, the least-significant byte is stored at address 603 and the most-significant byte at address 600.                   |

### 14.1.2.2 Logical Memory Region Cacheability

For loads and stores, the data cache uses the LMCN settings for a memory region to determine access cacheability. The LMCN registers also specify whether cached data from the logical region can be quickly invalidated. See [Chapter 4, “Cache and On-Chip Data RAM”](#) for a detailed description of the data cache quick invalidation feature.

Figure 14-1. PMCON and LMCON Example



## 14.2 Programming the Physical Memory Configuration (PMCON) Registers

The layout of the Physical Memory Configuration registers, PMCON0 to PMCON15, is shown in [Figure 14-2](#), which gives the descriptions of the individual bits. The PMCON registers reside within memory-mapped control register space. Each PMCON register controls one 256 Mbyte region of memory according to the mapping shown in [Table 14-1](#). See [Table 3-4](#) for information on programming the MMRs.

**Table 14-1. Region Table Mapping**

Register (Region Table Entry)	Region Controlled
PMCON0	0000 0000H to 0FFF FFFFH
PMCON1	1000 0000H to 1FFF FFFFH
PMCON2	2000 0000H to 2FFF FFFFH
PMCON3	3000 0000H to 3FFF FFFFH
PMCON4	4000 0000H to 4FFF FFFFH
PMCON5	5000 0000H to 5FFF FFFFH
PMCON6	6000 0000H to 6FFF FFFFH
PMCON7	7000 0000H to 7FFF FFFFH
PMCON8	8000 0000H to 8FFF FFFFH
PMCON9	9000 0000H to 9FFF FFFFH
PMCON10	A000 0000H to AFFF FFFFH
PMCON11	B000 0000H to BFFF FFFFH
PMCON12	C000 0000H to CFFF FFFFH
PMCON13	D000 0000H to DFFF FFFFH
PMCON14	E000 0000H to EFFF FFFFH
PMCON15	F000 0000H to FFFF FFFFH

**Figure 14-2. PMCON Register Bit Descriptions**

Mnemonic	Name	Bit #	Function
NRAD4:0	Number of Read Address to Data Wait States	0-4	Specifies the number of wait states (0 to 31) inserted between the assertion of ADS# and the first data cycle for read accesses.
NRDD1:0	Number of Read Data to Data Wait States	6-7	Specifies the number of wait states (0 to 3) inserted between successive data cycles for a burst read access.
NWAD4:0	Number of Write Address to Data Wait States	8-12	Specifies the number of wait states (0 to 31) inserted between the assertion of ADS# and the first data cycle for write accesses.
NWDD1:0	Number of Write Data to Data Wait States	14-15	Specifies the number of wait states (0 to 3) inserted between successive data cycles for a burst write access.
NXDA3:0	Number of Bus Turnaround Wait States	16-19	Specifies the number of wait states (0-15) inserted after the last data cycle for accesses in this region.
PEN	Parity Enable	20	Enables parity generation/checking for a region. 0 = not enabled, 1 = enabled
PODD	Parity Odd	21	Selects parity sense for a region. 0 = even, 1 = odd
BW1:0	Bus Width	22-23	Selects the bus width for a region: 00 = 8-bit, 01 = 16-bit, 10 = 32-bit bus 11 = reserved (do not use)
RPIPE	Read Pipelining Enable	24	Enables address pipelining for read accesses in this region. 0 = disabled, 1 = enabled
BEN	Burst Enable	28	Enables burst accesses for the region. 0 = disabled, 1 = enabled
RBEN	READY#/BTERM#Enable	29	Enables the READY# and BTERM# pins for a region. 0 = READY#/BTERM# ignored in this region 1 = READY#/BTERM# enabled

All 16 PMCON registers are loaded automatically during system initialization. The initial values are stored in the Control Table in the Initialization Boot Record (see [Section 13.3.1, “Initial Memory Image \(IMI\)” on page 13-10](#)). On a hardware reset, PMCON15 is automatically set to the values shown in [Table 14-2](#). This operation configures all regions to an 8-bit bus width with maximum wait states with bursting, pipelining, parity and ready-control disabled.

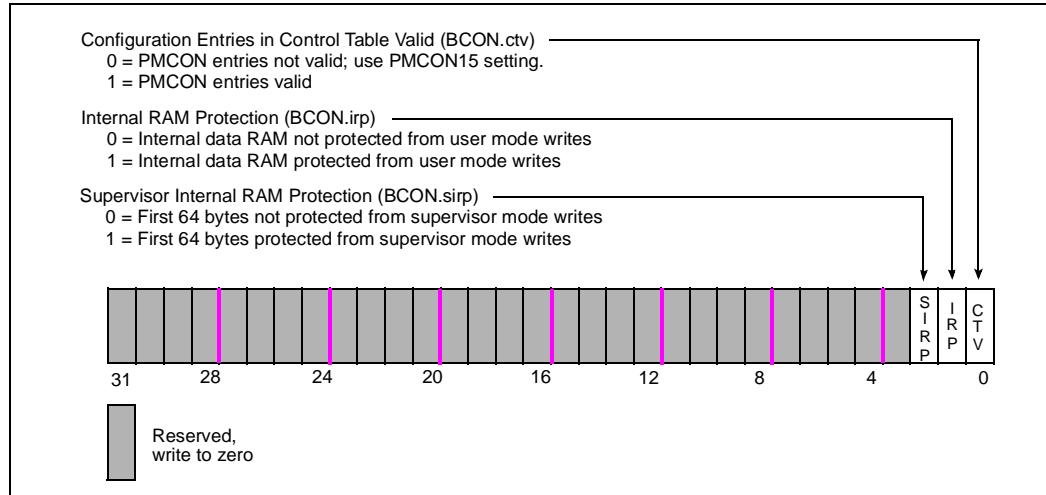
**Table 14-2. PMCON15 Register Values after Reset**

PMCON15 Bit or Bit Field	Value During Reset Microcode
NRAD4:0 [bits 0-4]	31
NRDD1:0 [bits 6-7]	X (don't care)
NWAD4:0 [bits 8-12]	31
NWDD1:0 [bits 14-15]	X (don't care)
NXDA3:0 [bits 16-19]	15
PEN [bit 20]	0 (disabled)
PODD [bit 21]	X (don't care)
BW1:0 [bits 22-23]	0 (8-bit)
RPIPE [bits 24]	0 (disabled)
BEN [bit 28]	0 (bursts disabled)
RBEN [bit 29]	0 (disabled)

### 14.2.1 Bus Control (BCON) Register

Immediately after a hardware reset, the Bus Control (BCON) register is set to zero, which marks the PMCON register contents as invalid. [Figure 14-3](#) shows the BCON register and Control Table Valid (BCON.ctv) bit. Whenever the PMCON entries are marked invalid by BCON, the BCU uses the parameters in PMCON15 for *all* regions. After the processor loads all PMCON registers, BCON is loaded from the Control Table. If BCON.ctv is clear, then PMCON15 remains in use for all bus accesses. If BCON.ctv is set, the region table is valid and the BCU uses the programmed PMCON values for each region.

**Figure 14-3. Bus Control Register (BCON)**



## 14.3 Boundary Conditions for Physical Memory Regions

The following sections describe the operation of the PMCON registers during conditions other than “normal” accesses.

### 14.3.1 Internal Memory Locations

The PMCON registers are ignored during accesses to internal memory or memory-mapped registers.

### 14.3.2 Bus Transactions across Region Boundaries

The BCU can perform unaligned word and short word requests. Unaligned requests greater than one word are automatically broken into word requests by microcode. Bus requests broken up by microcode are treated as separate requests by the BCU. Any word or short word request that spans region boundaries uses the PMCON settings of the first region.

For example, an unaligned word load/store beginning at address 1FFF FFFEH would cross boundaries from region 1 to 2. This request is broken into two accesses, whose addresses are 1FFF FFFEH and 2000 0000H. The word request at 1FFF FFFEH spans the region boundary and uses the physical parameters for region 1.

### 14.3.3 Modifying the PMCON Registers

An application can modify the value of a PMCON register by using the **st** or **sysctl** instruction. If a **st** instruction is used to modify a PMCON register, precede the **st** instruction with a **syncf** instruction. This ensures that there are no outstanding loads or stores in the BCU.

## 14.4 Programming the Logical Memory Attributes

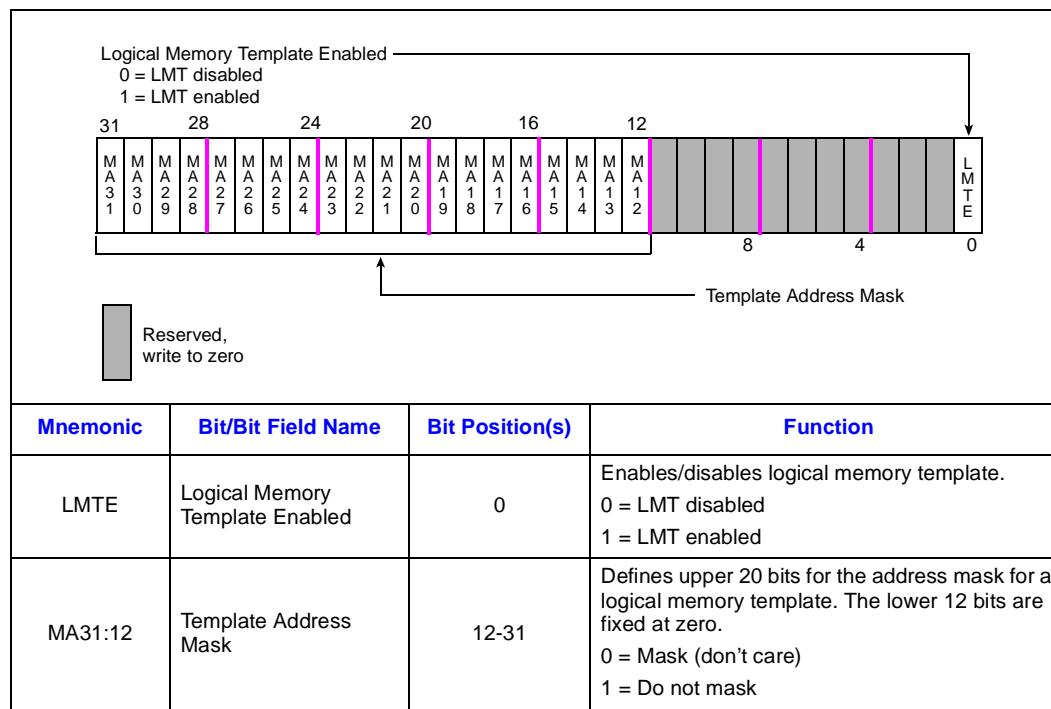
The layouts for the LMAR14:0 and LMMR14:0 registers are shown in [Figure 14-4](#) and [Figure 14-5](#). LMCON registers reside within the memory-mapped control register space. (See Section 3.3, “Memory-Mapped Control Registers” on page 3-6.)

**Figure 14-4. Logical Memory Address Registers (LMAR14:0)**

Reserved,  
write to zero

Mnemonic	Bit/Bit Field Name	Bit Position(s)	Function
BE	Big Endian Byte Order	0	Controls byte order for the template. 0 = Little-endian 1 = Big-endian
DCEN	Data Cache Enable	1	Controls data caching template. 0 = Data caching disabled 1 = Write-through caching enabled
DCIIR	Data Cache Independently Invalidatable Region	4	Controls Logical Region Invalidations. 0 = region is not independently invalidatable 1 = region is independently invalidatable using the <b>dcctl</b> “Invalidate Logical Region” command.
A31:12	Template Starting Address	12-31	Defines upper 20 bits for the starting address for a logical data template. The lower 12 bits are fixed at zero. The starting address is modulo 4 Kbytes.

Figure 14-5. Logical Memory Mask Registers (LMMR14:0)



The Default Logical Memory Configuration (DLMCON) register is shown in Figure 14-6. The BCU uses the parameters in the DLMCON register when the current access does not fall within one of the 15 logical memory templates (LMTs).

**Figure 14-6. Default Logical Memory Configuration Register (DLMCON)**

Bit Position(s)	Function
0	Controls byte order for areas not within other logical memory templates. 0 = Little-endian 1 = Big-endian
1	Controls data caching policy for areas not within other logical memory templates. 0 = Data caching disabled 1 = Write-through data caching enabled
4	Controls Logical Region Invalidation for areas not within other Logical Memory Templates. 0 = Region is not independently invalidatable 1 = Region is independently invalidatable using the <b>dctl</b> “Invalidate Logical Region” command.

Legend: Reserved, write to zero

#### 14.4.1 Defining the Effective Range of a Logical Memory Template

For each logical memory template, an LMAR register sets the base address using the A31:12 field. The LMMR register sets the address mask using the MA31:12 field. The effective address range for a logical data template is defined using the A31:12 field in an LMAR register and the MA31:12 field in an LMMR register. For each access, the upper 20 address bits (A31:12) are compared against A31:12 in the LMAR register. Only address bits with corresponding MA bits set are compared. Address bits with corresponding MA bits cleared (0) are automatically considered a “match”. The processor uses only the logical data template when all compared address bits match.

The following 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 LMAR and LMMR registers:

Candidate Address is of form:	0010 XXXX
LMAR <31:12> should be:	0010 0...
LMMR <31:12> should be:	FFFF 0...

- Multiple data templates can be created from a single LMAR/LMMR 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
LMAR <31:12> should be:	0000 0...
LMMR <31:12> should be:	FF0F 0...

#### 14.4.2 Selecting the Byte Order

The BCU can automatically convert big endian data in memory into little endian data for the processor core. The conversion is done transparently in hardware, with no performance penalty. The BE bit in the LMAR register controls the byte order for a logical data template. The BE bit in the DLMCON register controls the default byte ordering for the system.

Byte ordering is not applicable to memory-mapped registers since they are always accessed as words.

#### 14.4.3 Logical Region Invalidations Control

The i960 Hx processor allows independent Data Cache coherency management for a set of selected logical memory regions. This allows memory regions that are shared with other bus masters to be invalidated without the performance penalty associated with invalidating the entire cache.

Whether a particular LMT is part of this set of regions is indicated by the Data Cache Independently Invalidatable Region (LMAR.dciir) bit of the LMAR. If this bit is set, then data fetched using the LMT will be affected by the **dcctl** “Quick Invalidate” command, as well as by the commands and methods used to control the rest of the Data Cache.

The **dcctl** instruction's "Invalidate Logical Region" command affects all lines in the Data Cache that were fetched from any LMT that had the LMAR.dciir bit set at the time the line was allocated; it does not affect any other lines in the cache. The affected lines are all invalidated by the command. This command is extremely fast, completing in only a few cycles.

Note that there is no mechanism for invalidating only those lines that were fetched from a particular LMT. All lines from LMTs with the LMAR.dciir bit set are managed as a group.

Software can make changes to the LMCON at any time without regard to bus queues or bus activity. The processor synchronizes the LMCON settings to the bus requests automatically.

#### 14.4.4 Data Caching Enable

Enabling and disabling data caching for an LMT is controlled via the LMAR.dcen bit in the LMAR register. Likewise, the LMAR.dcen bit in DLMCON enables and disables data-caching for regions of memory that are not covered by the LMCON registers. The LMAR.dcen bit has no effect on the instruction cache.

#### 14.4.5 Enabling the Logical Memory Template

The LMTE bit activates the logical data template in the LMMR register for the programmed range.

#### 14.4.6 Initialization

Immediately following a hardware reset, all LMTs are disabled. The LMAR.lmte bit in each of the LMMR registers is cleared (0) and all other bits are undefined. Immediately after a hardware reset the Default Logical Memory Control register (DLMCON) has the values shown in [Table 14-3](#).

**Table 14-3. DLMCON Values at Reset**

DLMCON Bit	Value After Reset
DCIIR	0 (area not independently invalidatable)
DCEN	0 (data caching disabled)
BE (Big-Endian)	Loaded from BBIGE bit in the Initialization Boot Record

Application software may initialize and enable the logical memory template after hardware reset. The registers are not modified by software re-initialization.

The contents of LMAR registers are cleared after a hardware or software reset.

## 14.4.7 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.

### 14.4.7.1 Internal Memory Locations

The user can map internal data RAM into an LMT; however, only the byte-ordering information is used. Since internal data RAM locations are never cached, the BCU ignores LMT bits controlling caching for data RAM accesses.

The LMT registers are not used during accesses to memory-mapped registers.

### 14.4.7.2 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 logical memory attributes of the access are undefined.

To establish different logical memory attributes for the same address range, program non-overlapping logical ranges, then use partial physical address decoding.

### 14.4.7.3 Accesses across LMT Boundaries

Avoid accesses that cross LMT boundaries. The processor does not cache any data in a non-cacheable region, even if the request spans from a cacheable region.

## 14.4.8 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.

This chapter describes the bus pins, bus transactions and bus arbitration. It shows waveforms to illustrate some common bus configurations. This chapter serves as a guide for the hardware designer when interfacing memory and peripherals to the i960<sup>®</sup> Hx processor. For further details on external bus operation, refer to [Appendix F, “Bus Interface Examples”](#). For information on bus controller configuration, refer to [Chapter 14, “Memory Configuration”](#). For pin descriptions, refer to the *80960HA/HD/HT Embedded 32-bit Microprocessor* datasheet.

## 15.1 Overview

A 32-bit high performance Bus Control Unit (BCU) interfaces the i960 Hx processor core to external memory and peripherals. One of the key advantages of the BCU design is its versatility. The user can program system memory’s physical and logical attributes independently. Physical attributes include wait state profile, bus width and parity. Logical attributes include cacheability and big or little endian byte order. Internally programmable wait states and 16 separately configurable physical memory regions allow the processor to interface with a variety of memory subsystems with minimum system complexity. To reduce the effect of wait states, the bus design is decoupled from the core. Decoupling lets the processor execute instructions while the bus performs memory accesses independently.

The Bus Controller’s key features include:

- Demultiplexed, burst bus to support most efficient DRAM access modes
- Address pipelining to reduce memory cost while maintaining performance
- 32-, 16- and 8-bit modes for I/O interfacing ease
- Full internal wait state generation to reduce system cost
- Little and big endian support
- Unaligned access support implemented in hardware
- Request queue to decouple the bus from the core (four-deep for loads/stores, two-deep for fetches)
- Independent physical and logical address space characteristics

## 15.1.1 Terminology: Requests and Accesses

The terms *request* and *access* are used frequently when referring to bus controller operation. The description of the bus modes and burst bus operation is simplified by defining these terms.

### 15.1.1.1 Request

The terms *request*, *bus request* or *memory request* describe interaction between the core and bus controller. The bus controller is designed to decouple, as much as possible, bus activity from instruction execution in the core. When a load or store instruction or instruction fetch is issued, the core delivers a bus request to the bus control unit.

The bus control unit independently processes the request and retrieves data from memory for load instructions and instruction fetches. The bus controller delivers data to memory for store instructions. The i960 architecture defines byte, short word, word, double word, triple word and quad word data lengths for load and store instructions.

When a load or store instruction is encountered, the core issues a bus request of the appropriate data length to the bus controller. For example, **ldq** requests that four words of data be retrieved from memory, and **stob** requests that a single byte be delivered to memory.

The processor fetches instructions using double or quad word bus requests.

### 15.1.1.2 Access

The terms *access*, *bus access* or *memory access* describe the mechanism for moving data or instructions between the bus controller and memory. An access is bounded by the assertion of ADS# (address strobe) and BLAST# (burst last) signals, which are outputs from the processor. ADS# indicates that a valid memory address is present and an access has started. BLAST# indicates that the next data transferred is the end of access. The bus controller can be configured to initiate burst, non-burst or pipelined accesses. A burst access begins with ADS# followed by two to four data transfers. The last data transfer is indicated by assertion of BLAST#. Non-burst accesses begin with assertion of ADS# followed by a single data transfer. Pipelined accesses begin on the same clock cycle in which the previous cycle completes. This is accomplished by asserting ADS# and a valid address during the last data transfer of the previous cycle. Pipelined accesses may also be burst or non-burst.

## 15.2 Bus Signals

As described in [Table 15-1](#), the i960 Hx processor bus consists of 30 address signals, four byte enables, 32 data lines, four data parity signals, and various control and status signals.

**Table 15-1. Bus Controller Signals**

Signal Name	Description	Input/Output
<b>Clock</b>		
CLKIN	Clock Input	I
<b>Address/Data Signals</b>		
A[31:2]	Address Bus	O
D[31:0]	Data Bus	I/O
DP[3:0]	Data Parity	I/O
<b>Control Signals</b>		
ADS#	Address Strobe	O
BE[3:0]#	Byte Enables	O
BLAST#	Burst Last	O
BTERM#	Burst Terminate	I
DEN#	Data Enable	O
DT/R#	Data Transmit/Receive	O
PCHK#	Parity Check	O
READY#	Memory Ready	I
WAIT#	Wait States	O
W/R#	Write/Read	O
<b>Status Signals</b>		
CT[3:0]	Cycle Type	O
D/C#	Data/Code Request	O
SUP#	Supervisor Mode Request	O
<b>Bus Arbitration</b>		
BOFF#	Bus Backoff	I
BREQ	Bus Request Pending	O
BSTALL	Bus Stall	O
HOLD	Hold Request	I
HOLDA	Hold Acknowledge	O
LOCK#	Locked Request	O

## 15.2.1 Bus Clock

The i960 Hx processor generates a core clock based upon the CLKIN signal. The i960 HA processor core operates at the external bus speed. The i960 HD and HT processors multiply the input clock signal on CLKIN by two and three respectively and feed this multiplied clock to the CPU core. The external system bus uses CLKIN directly as its timing reference. For example, with a 33 MHz CLKIN signal the i960 HD processor core runs at 66 MHz and the external system bus runs at 33 MHz.

## 15.2.2 Address, Data and Parity Signals

The address and byte enable signals determine which of the  $2^{32}$  memory locations to access. Addressed data is then transferred on the data pins.

Data signals are bi-directional and are used for transferring data between the processor and external memory systems during reads and writes. The bus is programmable for 8-, 16- and 32-bit transfer widths. A request to an 8-bit region uses data bits 7:0. Bits 15:0 are used during 16-bit accesses, and 32-bit accesses use all of the data bits (31:0). Each data byte is associated with a programmable parity bit and a byte enable bit:

**Table 15-2. Data, Parity and Byte Enable Associations**

Data Bits	Parity Bit	Byte Enable
D[31:24]	DP3	BE3#
D[23:16]	DP2	BE2#
D[15:8]	DP1	BE1#
D[7:0]	DP0	BE0#

Parity is always generated on write requests, but checked only when the PMCON.pen bit is enabled for a memory region. On write accesses, the memory system should only store parity bits that have their associated Byte Enable (BE[3:0]#) signals active. The bus controller presents the results of parity checking onto the PCHK# signal one bus cycle after the data is read.

### 15.2.3 Control Signals

The bus controller uses the upper four bits of the address to determine which of the 16 Physical Memory Configuration (PMCON) registers apply to the bus request. The bus controller then uses the associated PMCON register settings to sequence several control signals. See [Section 14.2, “Programming the Physical Memory Configuration \(PMCON\) Registers” on page 14-7](#) for more information on PMCON registers. These signals specify parameters such as:

- Access start and finish (ADS#, BLAST#)
- Wait state control (WAIT#, READY#, BTERM#)
- Transceiver control (DEN#, DT/R#)
- Data flow control (W/R#, BE[3:0]#)

#### 15.2.3.1 Access Start and Finish

A bus access starts with an address cycle, which is defined by the assertion of address strobe (ADS#). Address and byte enables (A[31:2] and BE[3:0]#) are also presented with the ADS# strobe.

A bus access may be either non-burst or burst. A non-burst access ends after one data cycle to a single memory location. A burst access involves two to four data cycles to consecutive memory locations. The Burst Last signal (BLAST#) is asserted to indicate the last data cycle of an access. [Section 15.3.2, “Burst Accesses” \(pg. 15-13\)](#) explains how to configure the bus controller for burst or non-burst accesses.

Read accesses may be pipelined. In a pipelined access, the data cycle and address cycle of two accesses can overlap. This is possible because address and data lines are not multiplexed. A valid address can be presented on the address bus while a previous access ends with a data transfer on the data bus. [Section 15.3.2, “Burst Accesses” \(pg. 15-13\)](#) explains how to configure the bus for pipelined accesses. Write accesses are not pipelined in the i960 Hx processor. For more information, see

#### 15.2.3.2 Wait State Control

Wait states can be inserted by the internal wait state generator, by external logic, or by a combination of both to accommodate the access time for external memory or peripherals. The WAIT# pin reflects the status of the internal wait state generator. External logic can insert wait states by using the READY# and BTERM# pins.

#### 15.2.3.3 Data Flow Control

W/R# discerns between a write request (store) or a read request (load or fetch). The Byte Enables (BE[3:0]#) determine which of the four bytes addressed by A[31:2] are active during a bus access.

### 15.2.3.4 Transceiver Control

The DT/R# and DEN# pins are used to control data transceivers. Data transceivers may be used in a system to isolate a memory subsystem or control loading on data lines. DT/R# is used to control transceiver direction; the signal is low for read requests and high for write requests. indicates data transfer cycles during a bus access. DEN# is asserted at the start of the first data cycle in a bus access and de-asserted at the end of the last data cycle. DEN# remains asserted for an entire bus request, even when that request spans several bus accesses. For example, a **ldq** instruction starting at an unaligned quad word boundary is one bus request spanning at least two bus accesses. DEN# remains asserted throughout all the accesses (including ADS# states) and de-asserts when the **Iqd** instruction request is satisfied. DEN# is used with DT/R# to provide control for data transceivers connected to the data bus. DEN# remains asserted for sequential reads from pipelined memory regions.

### 15.2.3.5 Status Signals

D/C# and SUP# provide information about the source of a bus request. D/C# indicates whether the current request is a data access or a code fetch. SUP# indicates whether the current request originates from user or supervisor mode. When used with a logic analyzer, these signals aid in software debugging.

D/C# may also be used to implement separate external data and instruction memories. SUP# can be used to protect hardware from accesses while the processor is in user mode.

The Cycle Type pins (CT[3:0]) provide information about the source of bus accesses. Bus accesses can be a result of executing instructions, such as **ld** or **st**, or a result of events, such as interrupts. When connected to logic analyzers or other diagnostic hardware, the CT[3:0] signals are useful for system debugging.

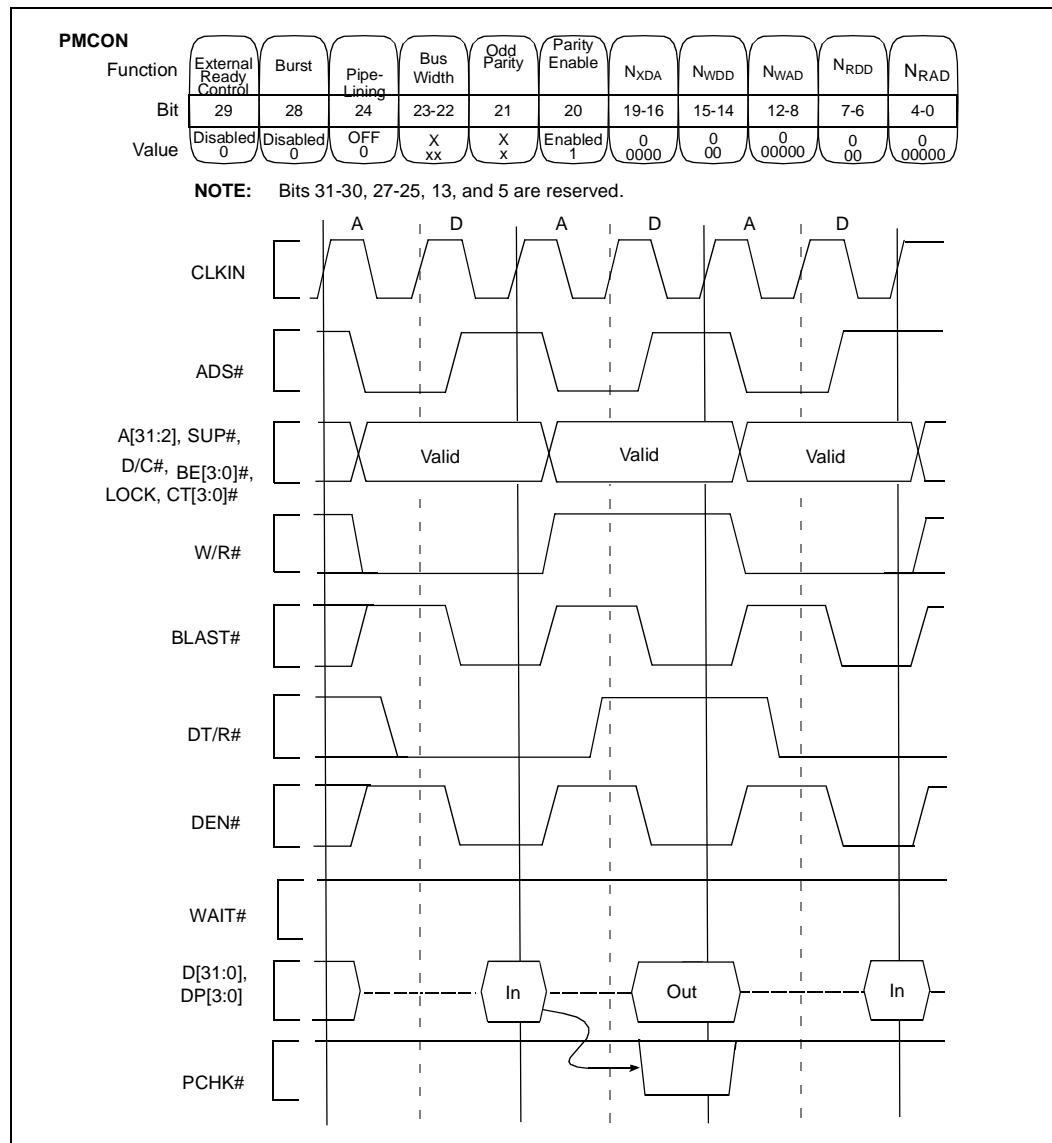
## 15.3 Basic Bus Transaction

A basic transaction (non-burst, non-pipelined; see [Figure 15-1](#)) is an address cycle followed by a single data cycle.

Assertion of ADS# indicates the address cycle, which is the beginning of an access. During the address cycle (marked as “A” on the waveform diagrams), the processor asserts signals such as A[31:2], BE[3:0]# and W/R# on the rising clock edge. The address cycle lasts for one bus cycle.

DT/R# is driven before the next rising edge of CLKIN. DT/R# specifies the direction of external data transceivers. The bus transitions from the address cycle into the data cycle when DEN# is asserted and ADS# is deasserted on the clock’s next rising edge. DT/R# is asserted early to ensure that it does not change while DEN# is asserted. DEN# is used to enable external data transceivers.

For write accesses, data is driven on the data lines (D[31:0]) at the start of a data cycle. For reads, the processor accepts the data at the end of the data cycle. The processor asserts BLAST# to indicate the end of the basic bus transaction. DEN# and BLAST# are deasserted at the next rising edge of CLKIN.

**Figure 15-1. Basic Read and Write Bus Accesses**


## 15.3.1 Wait States

The i960 Hx processor can generate wait states internally, or it can insert wait states based on the status of the BTERM# and READY# pins. Each physical memory region has its own wait state profile, set up using the PMCON registers. See [Section 14.2, “Programming the Physical Memory Configuration \(PMCON\) Registers” on page 14-7](#). The wait state profile can use:

- The internal wait state generator alone
- READY# and BTERM# together
- A combination of the above

For write accesses, the data lines are driven during wait states. For read accesses, data lines float. The following subsections describe generating wait states using these mechanisms.

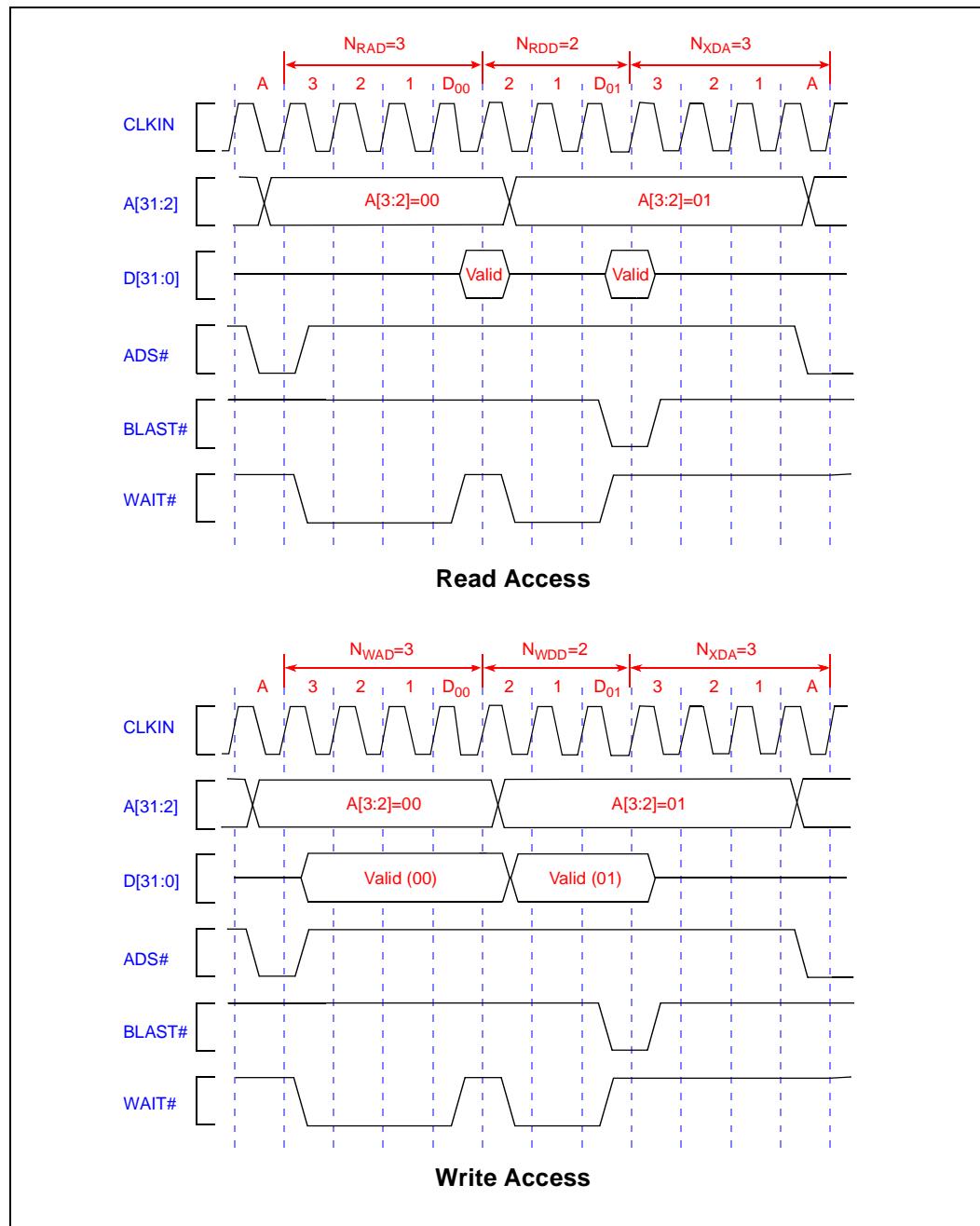
### 15.3.1.1 Internally Generated Wait States

The bus controller provides an internal counter for automatically inserting wait states. Five different wait state parameters are supported. [Figure 15-2](#) and the following text describe each parameter.

$N_{RAD}$	The number of clock cycles between the address cycle and first read data cycle.
$N_{RDD}$	The number of clock cycles between consecutive data cycles of a burst read.
$N_{WAD}$	The number of clock cycles that data is held after the address cycle and before the first write data cycle.
$N_{WDD}$	The number of clock cycles that data is held between consecutive data cycles of a burst write.
$N_{XDA}$	The minimum number of clock cycles between the last data cycle of a bus request and the address cycle of the next bus request (i.e., the bus turnaround time). $N_{XDA}$ applies to write and non-pipelined read requests. Note that $N_{XDA}$ is not generated after an access that is terminated with BTERM#, unless BLAST# is asserted in the same cycle.

The time required for an external device to relinquish the bus on a read request (read deasserted to data-float) determines the number of  $N_{XDA}$  cycles. Note that for pipelined read accesses, the bus controller uses a zero value for  $N_{XDA}$ , regardless of the parameter’s programmed value. The programmed value of  $N_{XDA}$  is used for write requests to pipelined memory regions, as the i960 Hx processor does not support pipelined write accesses.

The processor asserts the WAIT# signal when  $N_{RAD}$ ,  $N_{WAD}$ ,  $N_{RDD}$  or  $N_{WDD}$  is inserted. WAIT# is useful as a write strobe for simple external memory systems.  $N_{RDD}$  and  $N_{WDD}$  are not used in non-burst memory regions.

**Figure 15-2. Internal Programmable Wait States**


### 15.3.1.2 Externally Generated Wait States

Wait states can also be controlled with READY# and BTERM#. These inputs are enabled or disabled in a region by programming the PMCON registers. See [Section 14.1.1.4, “Wait States” on page 14-2](#).

When enabled, READY# indicates that read data on the bus is valid or that a write data transfer has completed. When used in conjunction with the internal wait state generator, the READY# pin value is ignored until the  $N_{RAD}$ ,  $N_{RDD}$ ,  $N_{WAD}$  or  $N_{WDD}$  wait states expire. At this time, if READY# is deasserted (high), wait states continue to be inserted until READY# is asserted. To bypass the internal wait state generator, program the  $N_{RAD}$ ,  $N_{RDD}$ ,  $N_{WAD}$  or  $N_{WDD}$  values to zero.

$N_{XDA}$  wait states are not affected by READY#. The READY# input is ignored during idle, address and  $N_{XDA}$  cycles. For read accesses, READY# is also ignored in memory regions where pipelining is enabled, regardless of memory region programming.

The burst terminate signal (BTERM#) breaks up a burst access. Asserting BTERM# for one clock cycle completes the current data transfer and invokes another address cycle. This allows a burst access to be dynamically broken into smaller accesses. The resulting accesses may also be burst accesses. For example, if BTERM# is asserted after the first word of a quad word burst, the bus controller initiates another access by asserting ADS#. The accompanying address is the address of the second word of the burst access ( $A[3:2] = 01_2$ ). The bus controller then bursts the remaining three words. The BLAST# (burst last) signal indicates the last data transfer of the access.

Read data is accepted on the clock edge where BTERM# is asserted. For writes, BTERM# indicates the memory system has accepted the data. In this way, BTERM# performs the same function as the memory READY# signal, ensuring that no data is lost when the current access is terminated. As with READY#, BTERM# is ignored for reads when pipelining is enabled in a region.

**Note:** When enabled, READY# and BTERM# are always used for write accesses, including regions where pipelining is enabled.

**Figure 15-3. Bus Request with READY# and BTERM# Control**

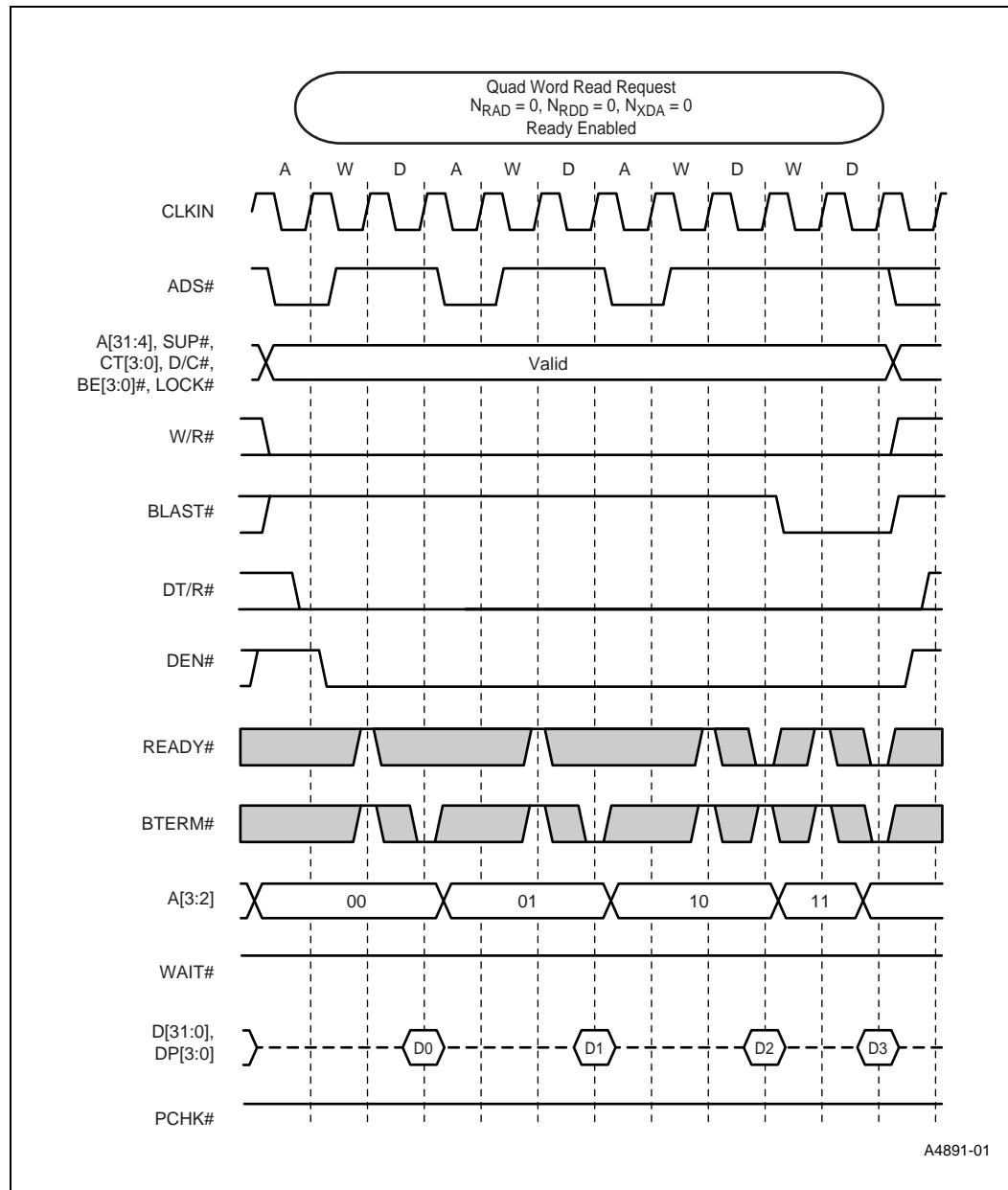
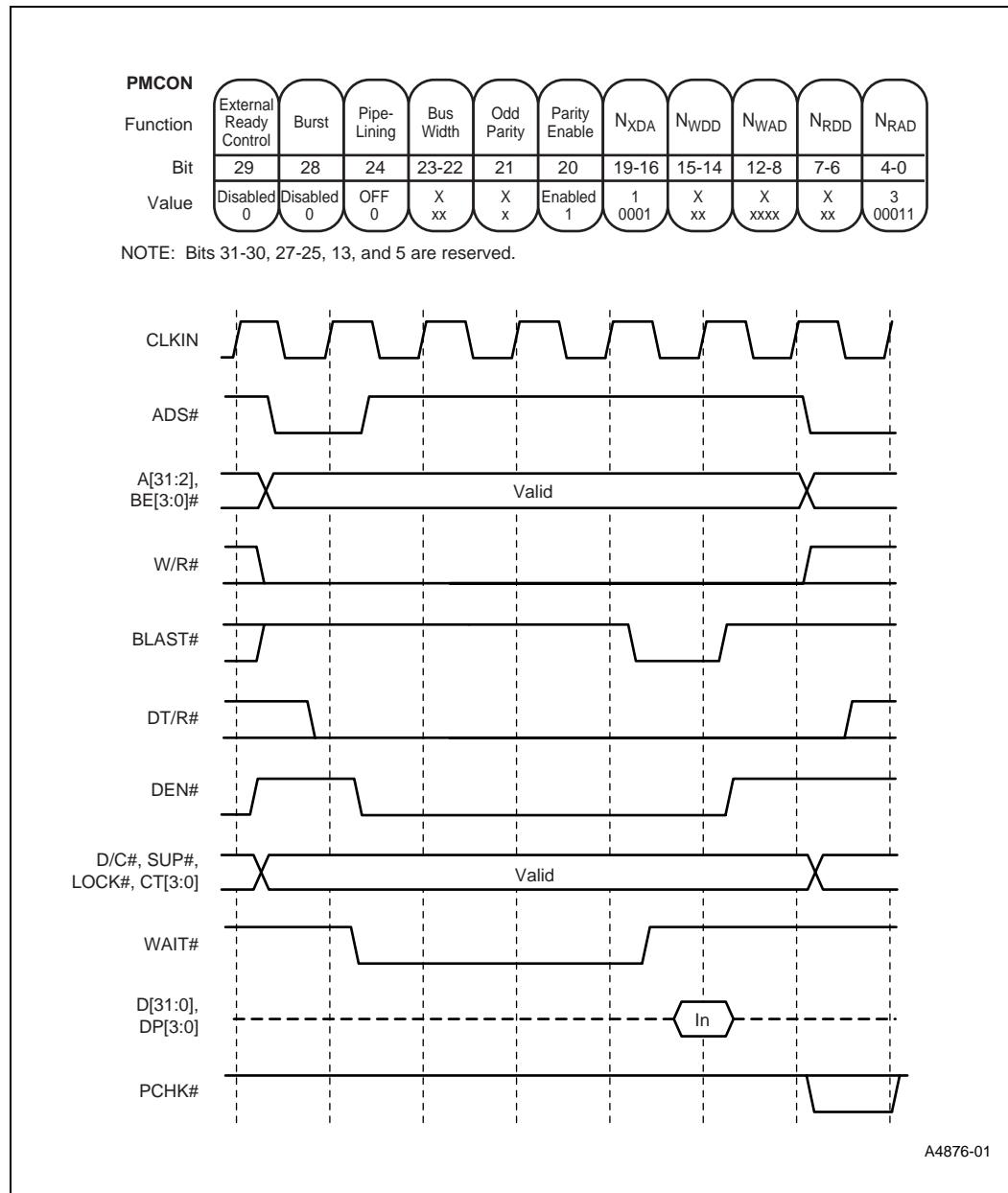


Figure 15-4. Non-Burst, Non-Pipelined Read Request with Wait States



BLAST# assertion indicates end of data transfer cycles for this access. DEN# is deasserted.  $N_{XDA}$  wait states (turnaround wait states) follow BLAST#; a new address cycle may start after  $N_{XDA}$  cycles expire.  $N_{XDA}$  states allow time for slow devices to relinquish the bus. For Figure 15-4, this access is the last access of a bus request because  $N_{XDA}$  wait states are inserted and DEN# is deasserted.

### 15.3.2 Burst Accesses

The i960 Hx processor's burst access allows up to four consecutive data cycles to follow a single address cycle. Compared to non-burst memory systems, burst mode memory systems achieve greater performance out of slower memory. SRAM, interleaved SRAM, Static Column Mode DRAM and Fast-Page Mode DRAM may be easily designed into burst-mode memory systems.

A burst read or write access consists of a single address cycle, 0-31 address-to-data wait states ( $N_{RAD}$  or  $N_{WAD}$ ) and 1-4 data cycles separated by 0-3 data-to-data wait states ( $N_{RDD}$  or  $N_{WDD}$ ). If READY#/BTERM# control is enabled in the region,  $N_{RAD}$ ,  $N_{WAD}$ ,  $N_{RDD}$  and  $N_{WDD}$  wait states may all be extended by not asserting READY#. BTERM# may be used to break a burst access into smaller accesses. See [Section 15.3.1, “Wait States” on page 15-8](#) for more information on this subject.

The maximum burst size is four data cycles. This maximum is independent of bus width. A byte-wide bus has a maximum burst size of four bytes; a word-wide bus has a maximum of four words. If a quad word load request (e.g., **ldq**) is made to an 8-bit data region, it results in four 4-byte burst accesses. (See [Table 15-3](#).)

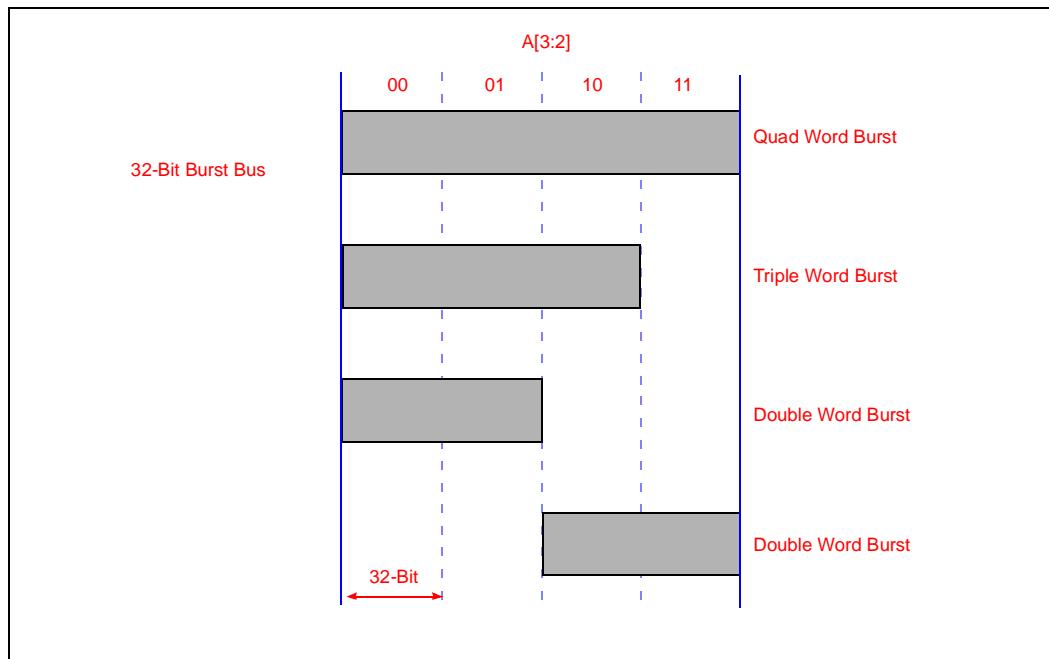
**Table 15-3. Burst Transfers and Bus Widths**

Request	Bus Width	Number of Burst Accesses	Number of Transfers per Burst	Number of Transfers
Quad Word	8 bit	4	4-4-4-4	16
	16 bit	2	4-4	8
	32 bit	1	4	4
Triple Word	8 bit	3	4-4-4	12
	16 bit	2	4-2	6
	32 bit	1	3	3
Double Word	8 bit	2	4-4	8
	16 bit	1	4	4
	32 bit	1	2	2
Word	8 bit	1	4	4
	16 bit	1	2	2
	32 bit	1	1	1
Short	8 bit	1	2	2
	16 bit	1	1	1
	32 bit	1	1	1
Byte	8 bit	1	1	1
	16 bit	1	1	1
	32 bit	1	1	1

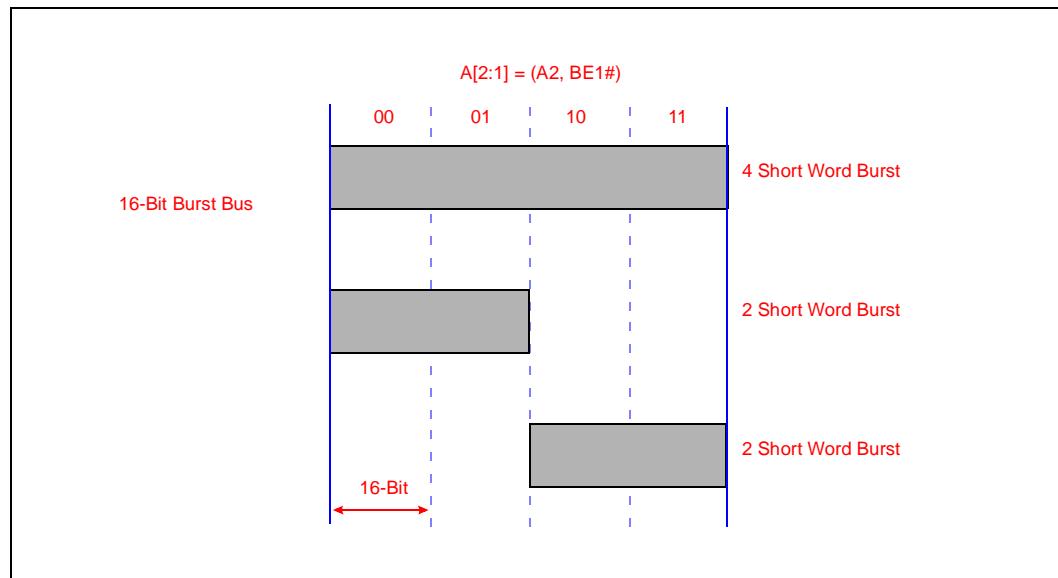
The address's two least-significant bits automatically increment after each burst data cycle. When a memory region is configured for a 32-bit data bus width, address pins A[3:2] increment. For a 16-bit memory region, BE1# is encoded as A1 and address bits A[2:1] increment. When a memory region is configured for an 8-bit data bus width, BE0# and BE1#, acting as the lower two bits of the address (A[1:0]), increment. See [Section 15.3.4, “Bus Width” on page 15-22](#) for more information on programming the bus width.

Burst accesses on a 32-bit bus are always aligned to even word boundaries. Quad word and triple word accesses always begin on quad word boundaries (A[3:2]=00); double word transfers always begin on double word boundaries (A2=0); single word transfers occur on single word boundaries. (See [Figure 15-5](#).)

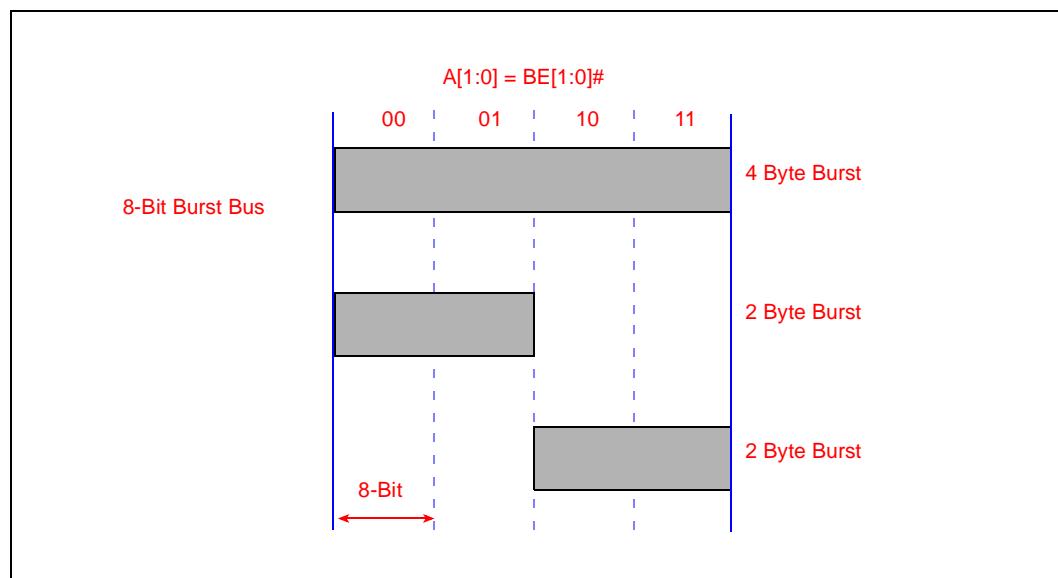
**Figure 15-5. 32-Bit-Wide Data Bus Bursts**



Burst accesses for a 16-bit bus are always aligned to even short word boundaries. A four short word burst access always begins on a four short word boundary (A2=0, A1=0). Two short word burst accesses always begin on a four word boundary (A2=0, A1=0). Single short word transfers occur on single short word boundaries (see [Figure 15-6](#)). For a 16-bit bus, valid data is transferred on data pins D[15:0]. Upper data lines D[31:16] are also driven on writes. For aligned accesses, the values are a duplicate of those driven on D[15:0].

**Figure 15-6. 16-Bit Wide Data Bus Bursts**


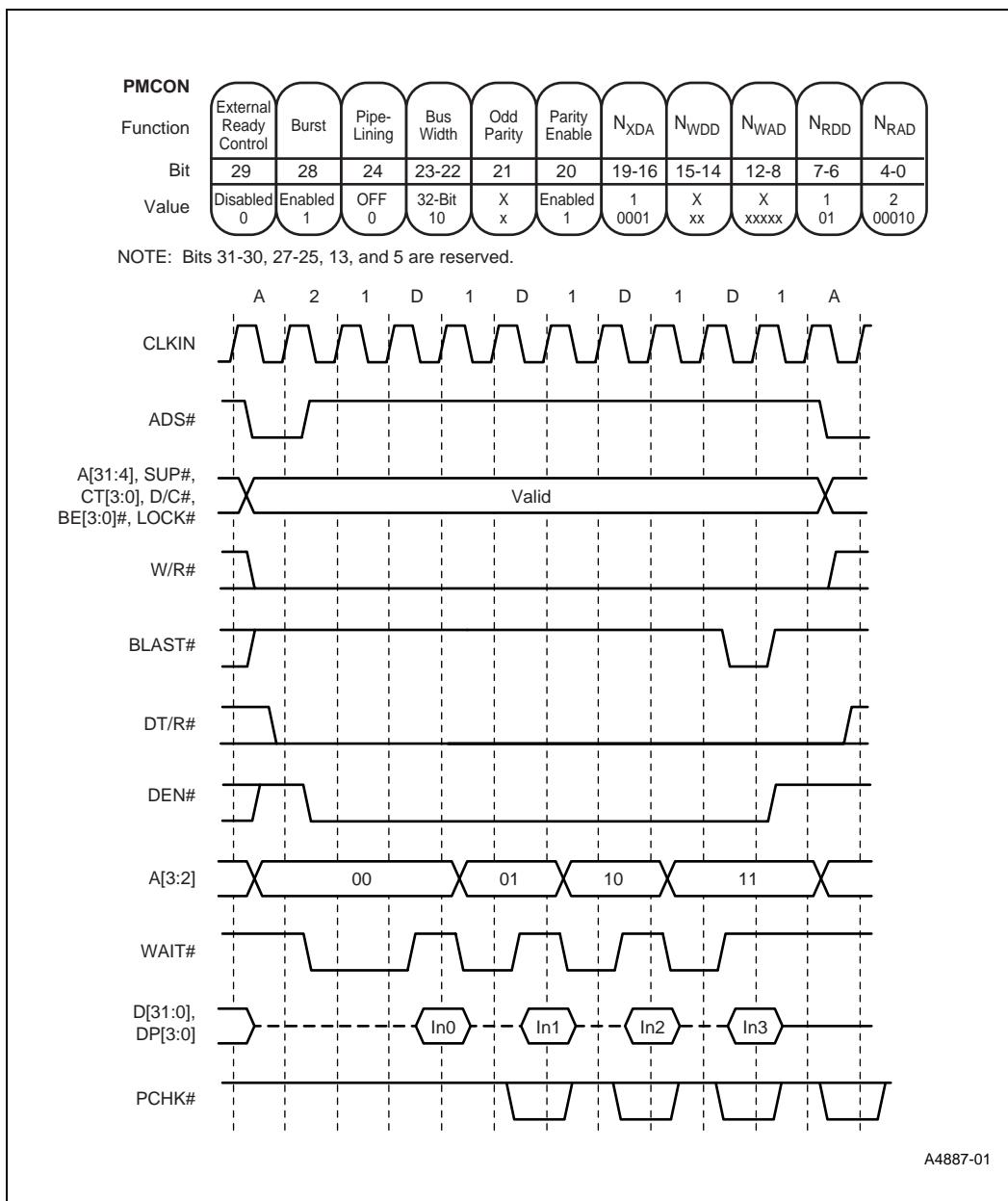
Burst accesses for an 8-bit bus are always aligned to even byte boundaries. Four-byte burst accesses always begin on a 4-byte boundary ( $A1=0$ ,  $A0=0$ ). Two-byte burst accesses always begin on an even byte boundary ( $A0=0$ ) (see [Figure 15-7](#)). For an 8-bit bus, data is transferred on data pins D[7:0]. Unlike the i960 Cx processor, data is not driven on the upper bytes of the data bus D[15:8], D[23:16] and D[31:24] on writes.

**Figure 15-7. 8-Bit Wide Data Bus Bursts**


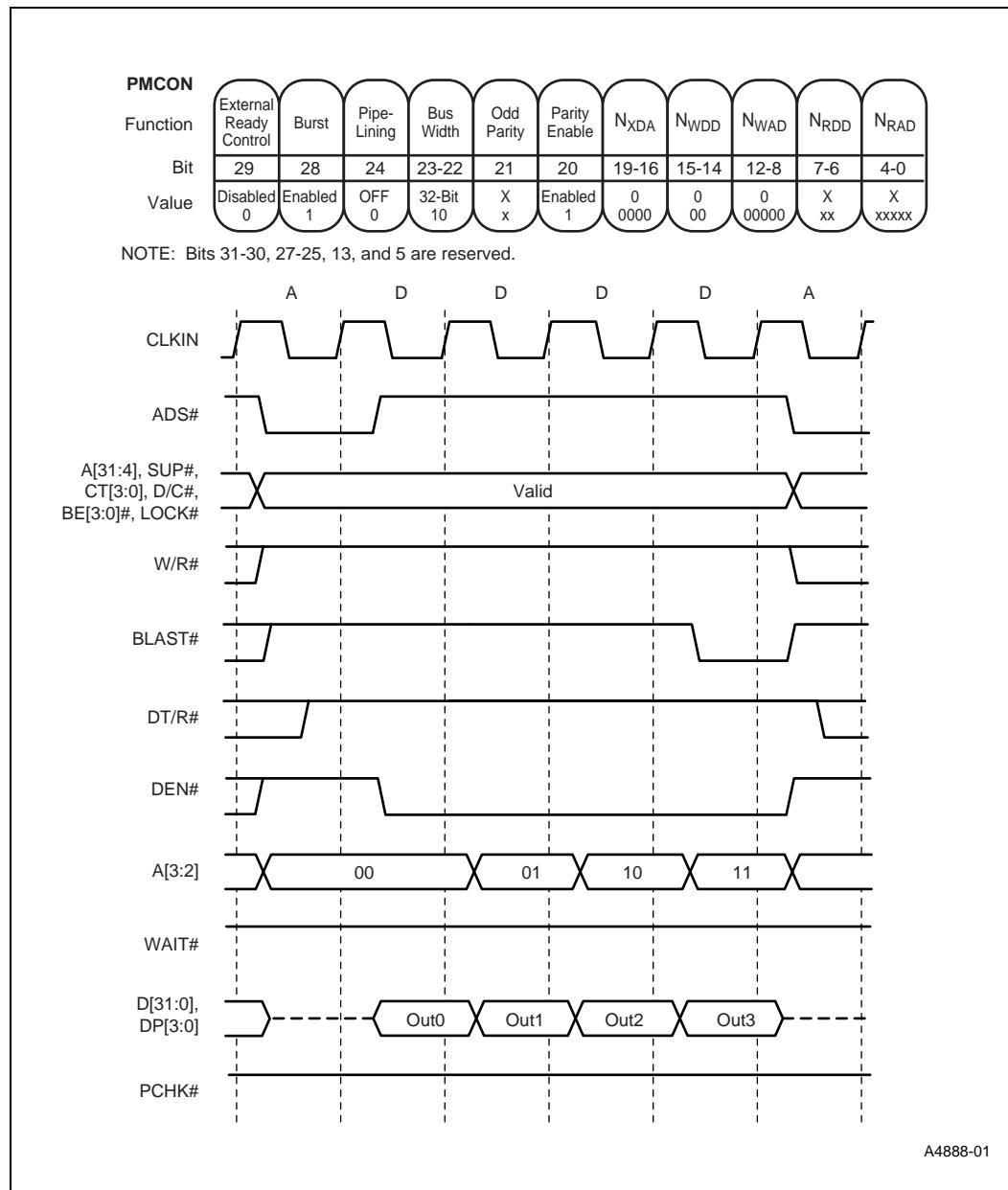
**Figure 15-8** shows a quad word read on a 32-bit bus; **Figure 15-9** shows a write. Burst access begins by asserting the proper address and status signals (ADS#, A[31:2], BE[3:0]#, SUP#, D/C#, W/R#). This is done on the rising edge that begins the address cycle ("A" on the figures). The processor asserts all byte enable signals (BE[3:0]#) during word read accesses.

DT/R# is driven before the clock's next rising edge to ensure that DT/R# does not change while DEN# is asserted. DEN# is asserted on the clock's next rising edge, the rising edge that ends the address cycle. ADS# is deasserted on this clock edge. DEN# is used to control external data transceivers. DEN# and DT/R# remain asserted throughout the burst access.

**Figure 15-8. 32-Bit Bus, Burst, Non-Pipelined, Read Request with Wait States**



**Figure 15-9. 32-Bit Bus, Burst, Non-Pipelined, Write Request without Wait States**



### 15.3.3 Pipelined Read Accesses

Pipelined read accesses provide the maximum data bandwidth. For pipelined reads, the next address is output during the current data cycle. This effectively removes the address cycle from consecutive pipelined accesses.

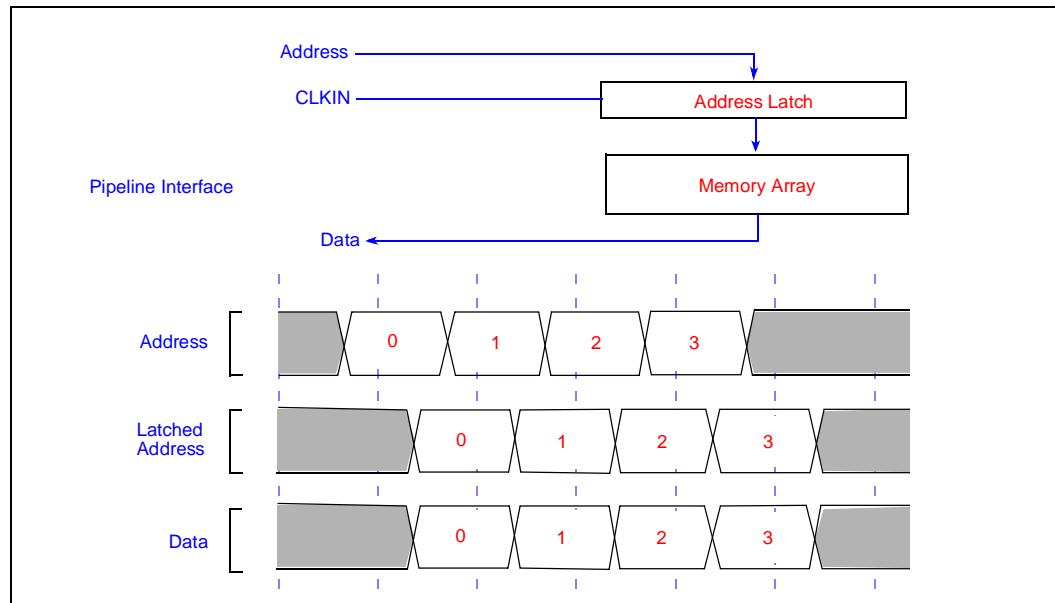
A pipelined read memory system is implemented by adding an address latch to the design (see Figure 15-10). The address latch holds the address for the current read access while the processor outputs the address for the next access. This allows the next address to be available during the data cycle of the current access.

Since the i960 Hx processor does not support pipelined write accesses, writes to a pipelined region behave the same as writes to a non-pipelined region. Also, the address for a read access following a write is not pipelined.

**Note:** When pipelining is enabled in a region, READY# and BTERM# are ignored for read cycles.

For pipelined reads, the bus controller uses a value of zero for the  $N_{XDA}$  parameter, regardless of the parameter's programmed value. A non-zero  $N_{XDA}$  value defeats the purpose of pipelining. The programmed value of  $N_{XDA}$  is used for write accesses to pipelined memory regions.

**Figure 15-10. Pipelined Read Memory System**



**Figure 15-11. Non-Burst, Pipelined Read Request without Wait States, 32-Bit Bus**

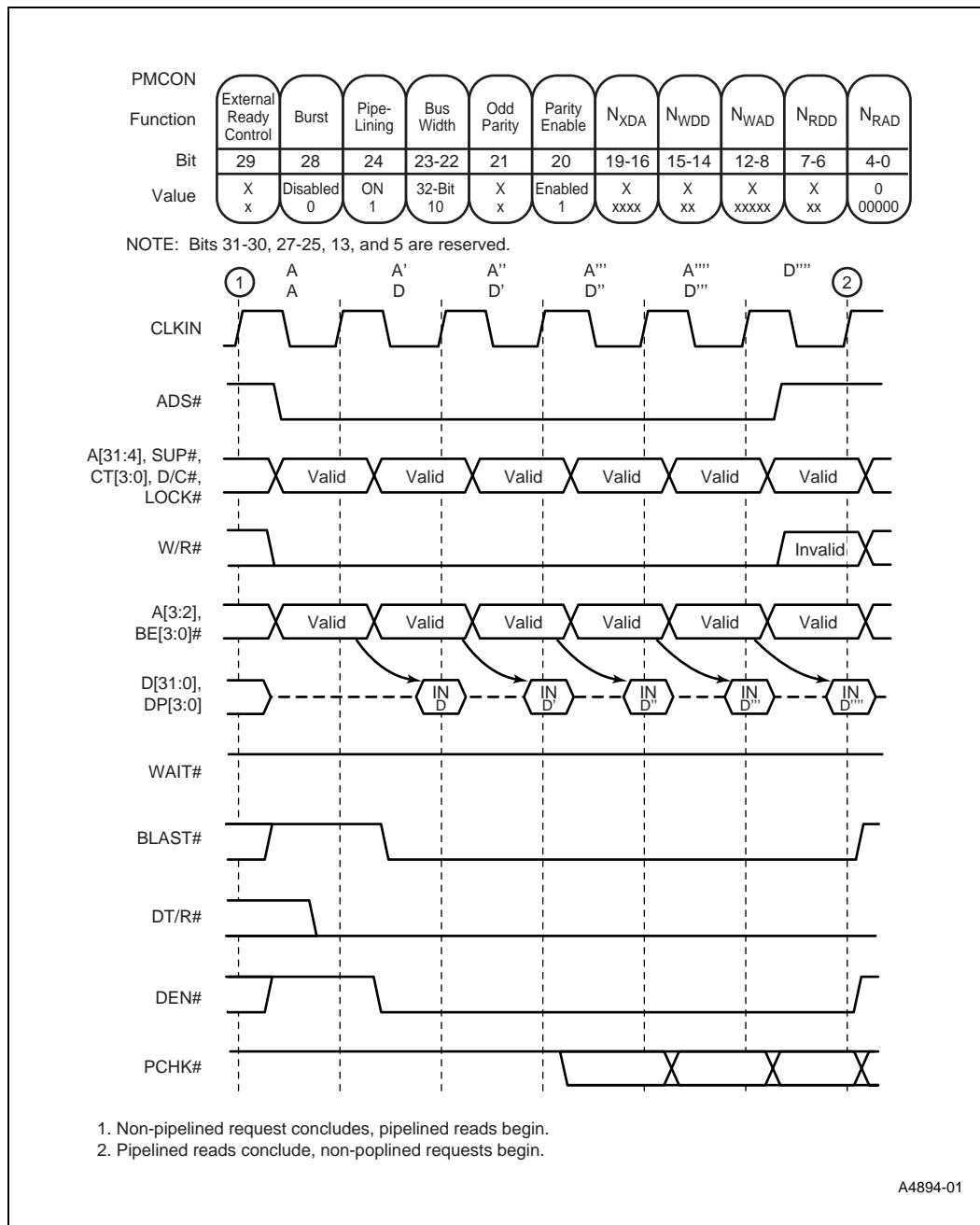
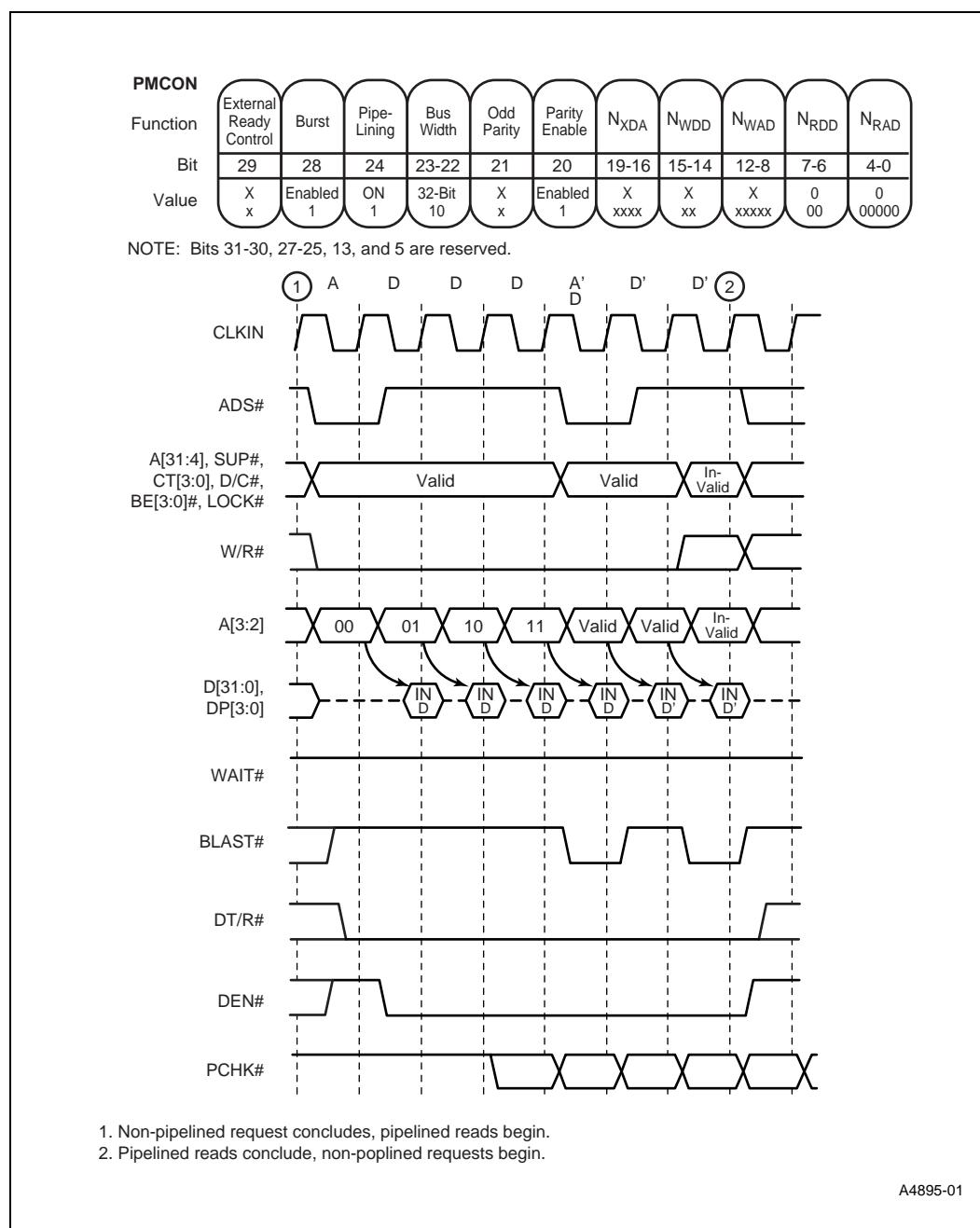
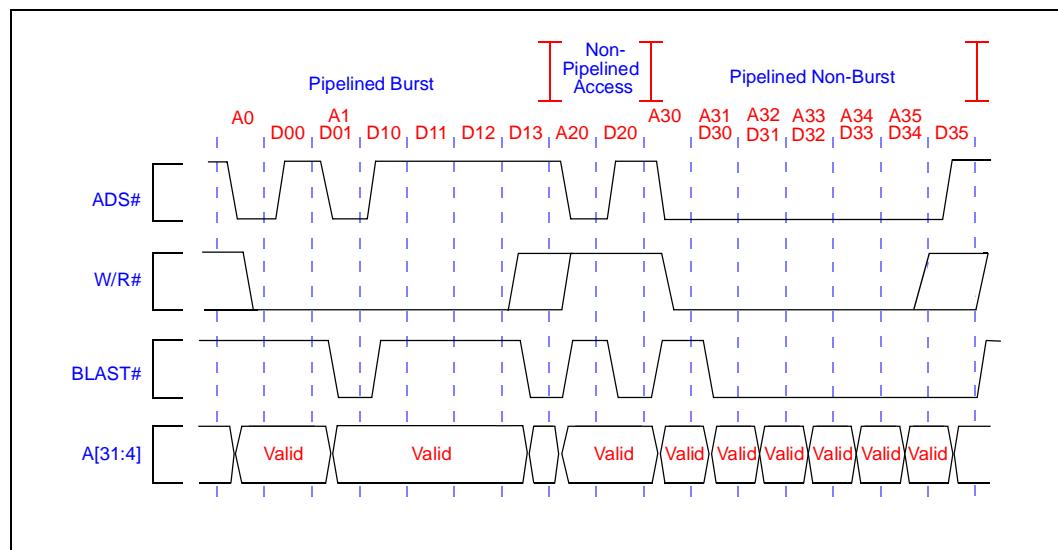


Figure 15-12. Burst, Pipelined Read Request without Wait States, 32-Bit Bus



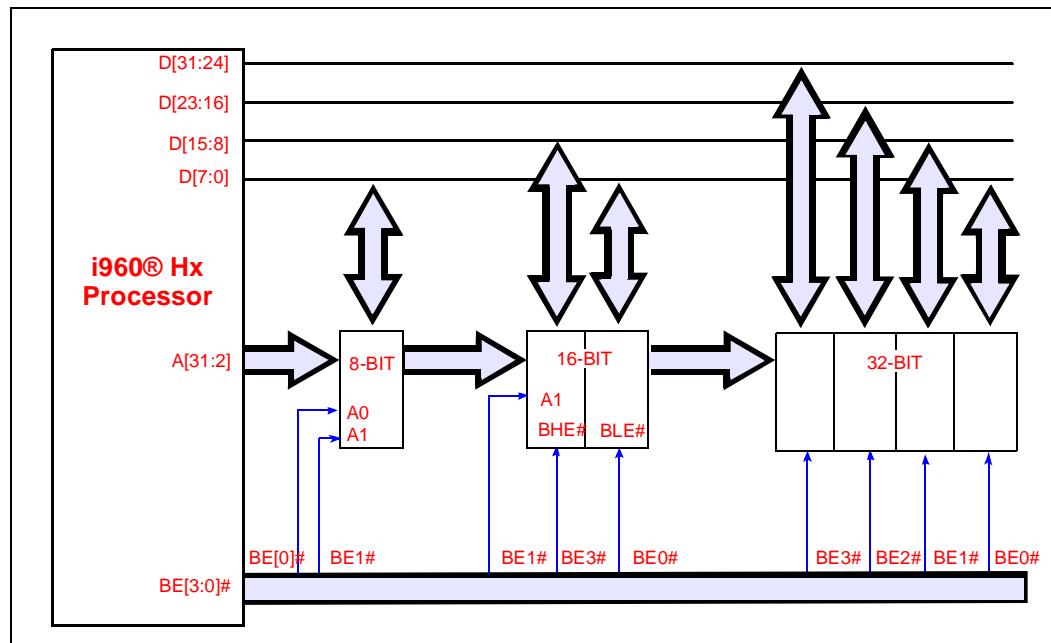
**Figure 15-13. Pipelined to Non-Pipelined Transitions**



### 15.3.4 Bus Width

Each region's data bus width is programmed using the PMCON registers (see [Section 14.1.1.1, "Data Bus Width" on page 14-2](#)). The i960 Hx processor allows an 8-, 16- or 32-bit-wide data bus for each region. The processor places 8- and 16-bit data on low order data pins. This simplifies interface to external devices. As shown in [Figure 15-14](#), 8-bit data is placed on lines D[7:0]; 16-bit data is placed on lines D[15:0]; 32-bit data is placed on lines D[31:0].

**Figure 15-14. Data Width and Byte Enable Encodings**



The four byte enable signals are encoded for each region to generate proper address signals for 8-, 16- or 32-bit memory systems:

- 8-bit region: BE0# is address line A0; BE1# is address line A1.
- 16-bit region: BE1# is address line A1; BE3# is the byte high enable signal (BHE#); BE0# is the byte low enable signal (BLE#).
- 32-bit region: byte enables are not encoded. Byte enables BE[3:0]# select byte 3 to byte 0, respectively. Address lines A[31:2] provide the most significant portion of the address. (See [Table 15-4](#).)

For regions configured for 8- and 16-bit bus widths, data is repeated on the upper data lines for aligned store operations. When storing a value to an 8-bit bus region, the processor drives the same byte-wide data onto lines D[7:0], D[15:8], D[23:16] and D[31:24] simultaneously. When storing a value to memory in a 16-bit bus region, the processor drives the same short word data onto lines D[15:0] and D[31:16] simultaneously.

**Table 15-4. Byte Enable Encoding**
**8-Bit Bus Width:**

<b>BYTE</b>	<b>BE3# (X)</b>	<b>BE2# (X)</b>	<b>BE1# (A1)</b>	<b>BE0# (A0)</b>
0	X	X	0	0
1	X	X	0	1
2	X	X	1	0
3	X	X	1	1

**16-Bit Bus Width:**

<b>BYTE</b>	<b>BE3# (BHE#)</b>	<b>BE2# (X)</b>	<b>BE1# (A1)</b>	<b>BE0# (BLE#)</b>
0,1	0	X	0	0
2,3	0	X	1	0
0	1	X	0	0
1	0	X	0	1
2	1	X	1	0
3	0	X	1	1

**32-Bit Bus Width:**

<b>BYTE</b>	<b>BE3#</b>	<b>BE2#</b>	<b>BE1#</b>	<b>BE0#</b>
0,1,2,3	0	0	0	0
1,2,3	0	0	0	1
0,1,2	1	0	0	0
2,3	0	0	1	1
1, 2	1	0	0	1
0,1	1	1	0	0
0	1	1	1	0
1	1	1	0	1
2	1	0	1	1
3	0	1	1	1

### 15.3.5 Parity Generation and Checking

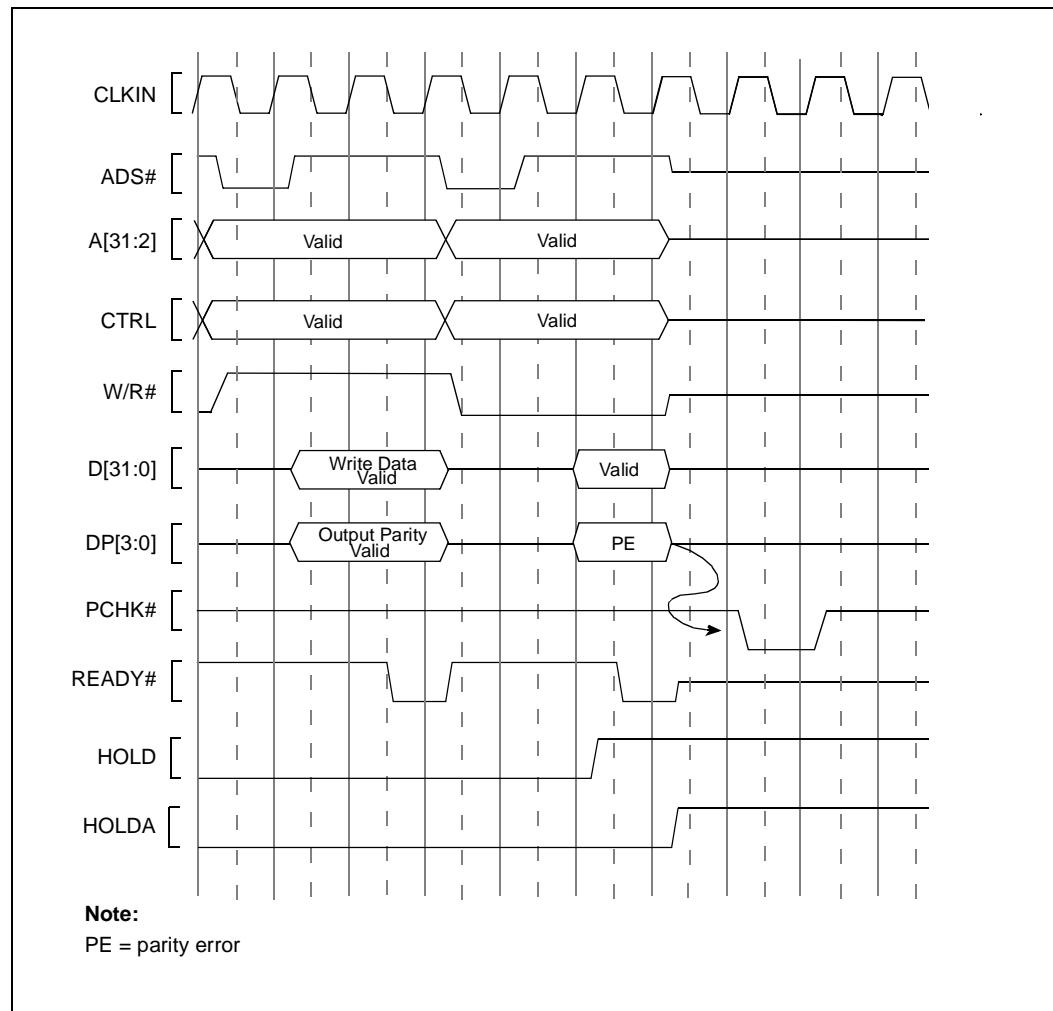
The i960 Hx processor's bus adds four data parity signals, DP[3:0], to indicate the parity of each byte of a data word. DP[3:0] timings are identical to those of D[31:0]. The processor generates parity for each store operation; however, parity is checked only for loads and fetches within regions that have parity checking enabled. See [Section 14.1.1.6, “Data Bus Parity” on page 14-4](#).

The processor signals a parity error by asserting the PCHK# pin after the rising edge of CLKIN in the clock cycle immediately following the data cycle that caused a parity error. See [Figure 15-15](#) and [Figure 15-16](#). PCHK# remains asserted for one clock cycle. If back-to-back data transfers (at zero wait states) cause parity errors, then PCHK# remains asserted. [Figure 15-15](#) shows DP[3:0] and PCHK# timings. The PCHK# pin goes low for parity failures during data accesses only. Accesses from the data cache do not test parity.

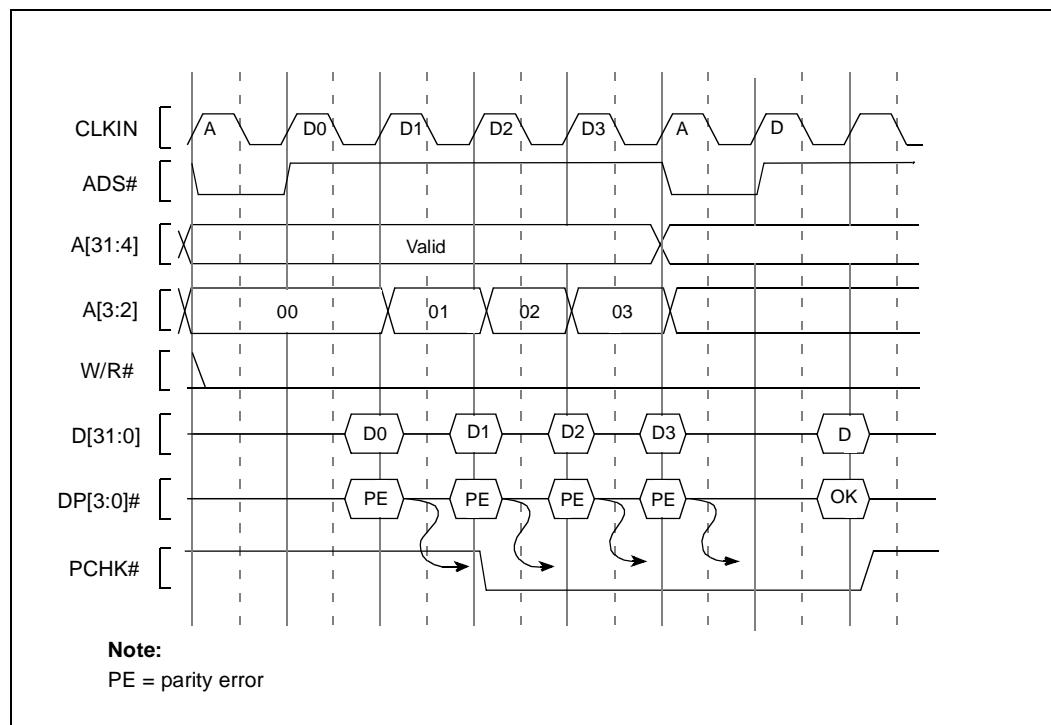
Since the PCHK# signal value becomes valid after an access completes, the processor does not float PCHK# when HOLDA is asserted.

When a parity error occurs, the processor can generate a MACHINE.PARITY fault. [Figure 8-5](#) shows the parity fault record. See [Section 8.10.3, “MACHINE Faults” on page 8-24](#) for details on the MACHINE.PARITY fault.

**Figure 15-15. Parity Error on Non-Burst Access**



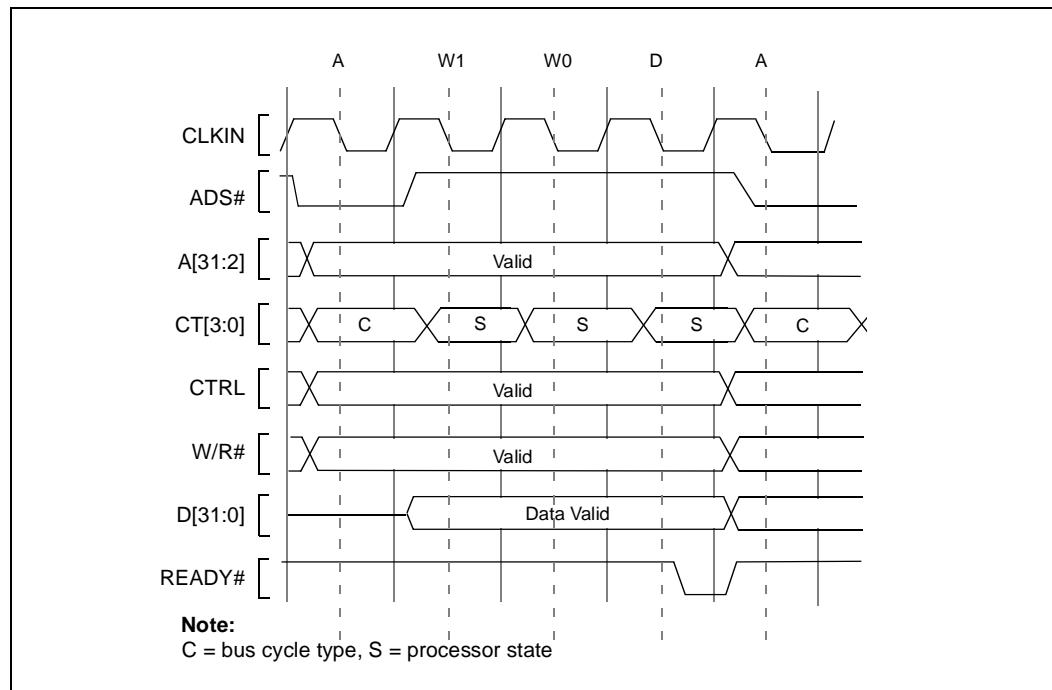
**Figure 15-16. Parity Error during Burst Access, No Wait States**



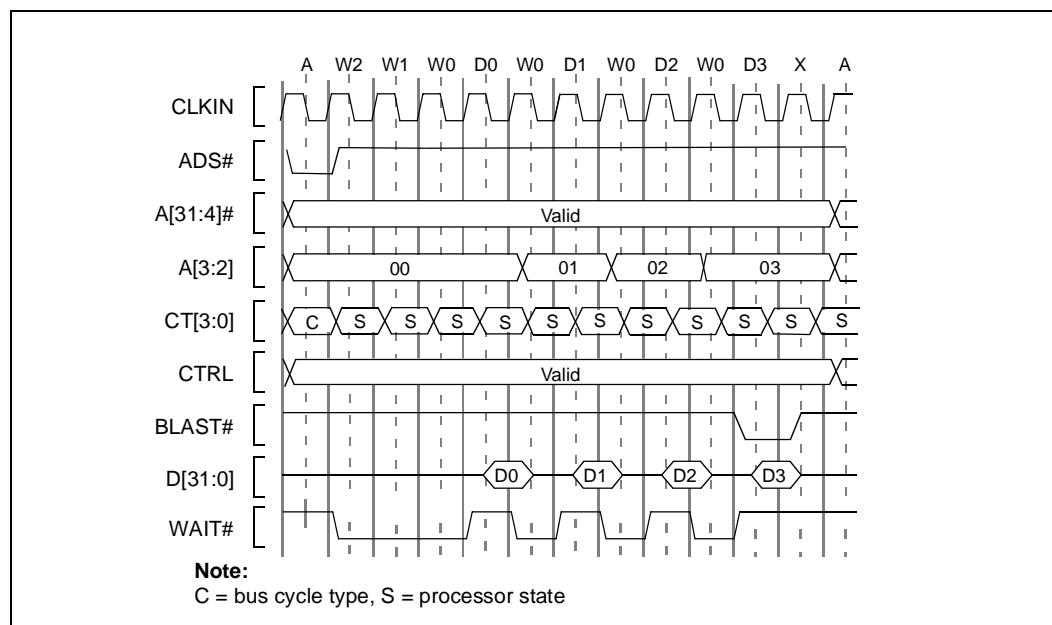
### 15.3.6 Cycle Type Pins

The i960 Hx processor includes cycle type pins, CT[3:0], that indicate the type of bus cycle in progress, or the processor state. The cycle type pins have the same timing as A[31:4]. See Figure 15-17.

**Figure 15-17. Cycle Type Pin Definition — Non-Burst Access**



**Figure 15-18. Cycle Type Pin Definitions — Burst Access**



The cycle type encodings are shown in **Table 15-5**. When ADS# is asserted, the pins contain information about the bus operation being initiated.

**Table 15-5. CT[3:0] Encoding**

Cycle Type	ADS	CT3	CT2	CT1	CT0
Program-initiated access using 8-bit bus	0	0	0	0	0
Program-initiated access using 16-bit bus			0	0	1
Program-initiated access using 32-bit bus			0	1	0
Event-initiated access using 8-bit bus			1	0	0
Event-initiated access using 16-bit bus			1	0	1
Event-initiated access using 32-bit bus			1	1	0
Reserved			X	1	1
Reserved for future products		1	X	X	X
Reserved	1	X	X	X	X

Some i960 Hx processor instructions are implemented in microcode, such as **atmod**, **atadd** and **sysctl**. There are also several non-instruction-related microcode routines, to perform functions such as initialization. Microcode routines are also called microflows.

*Program-initiated accesses* (cycle types 0000 through 0010) are initiated directly via instruction execution. Loads, stores and atomic accesses are considered program-initiated data access (qualified with the D/C# pin). Similarly, bus accesses requested by a program-initiated microflow (i.e., **calls** or **sysctl**) are considered program-initiated data accesses. Instruction fetches due to instruction cache misses are considered program-initiated code accesses.

*Event-initiated accesses* (cycle types 0100 through 0110) are initiated via microflow routines that are not called directly by the executing program. Examples include fault, interrupt and initialization microflows.

The width indicated by the cycle type pins is the width of the memory region being accessed as programmed in the PMCON registers of the processor. It is not the width of the actual transfer. For example, a program-initiated byte store to a 32-bit wide memory region would be cycle type 0010.

All unused encodings are reserved. Note that the i960 Hx processor does not drive signal CT3 high.

## 15.4 Little or Big Endian Memory Configuration

The bus controller supports big endian and little endian byte ordering for memory operations. Byte ordering determines how data is read from or written to the bus and ultimately how data is stored in memory. Little endian systems store a word's least significant byte at the lowest byte address in memory. For example, if a little endian ordered word is stored at address 600, the least significant byte is stored at address 600 and the most significant byte at address 603. Big endian systems store the least significant byte at the highest byte address in memory. So, if a big endian ordered word is stored at address 600, the least significant byte is stored at address 603 and the most significant byte at address 600.

Data in memory can be stored in either little or big endian order. The byte order for a logical memory region is programmed with the logical memory templates. See [Section 14.4.2, “Selecting the Byte Order” on page 14-14](#). Data and instructions can be located in either big or little endian regions.

Both byte ordering methods are supported for short word and word data types. [Table 15-6](#) shows how word, half word and byte data types are transferred on the bus according to the type of byte ordering used for the selected memory region and bus width (32, 16 or 8 bits). All transfers shown in the table are aligned memory accesses. The second column shows the lower address bits of the effective address (EFA). The EFA is the address of the beginning of the datum.

For word stores, assume that a hexadecimal value of aabbccddH is stored in an internal register, where aa is the word's most significant byte and dd is the least significant byte. [Table 15-6](#) shows how this word is transferred on the bus to either a little endian or big endian region of memory.

For half word stores, assume that a hexadecimal value of cdddH is stored in one of the internal registers. Note that the half word goes out on different data lines on a 32-bit bus depending on whether bit 1 of the memory address is set or cleared.

**Table 15-6** also shows that the i960 Hx processor handles byte data types the same regardless of byte ordering type. Multiple word bus requests (bursts) to a big endian region are handled as individual words. Bytes in each word are stored in big endian order. Big endian data types that exceed 32 bits are not supported and must be handled by software.

**Table 15-6. Byte Ordering on Bus Transfers**

Word Data Type			Bus Pins (Data Lines [31:0])							
Bus Width	EFA A[1:0]	Xfer	Little Endian				Big Endian			
			31:24	23:16	15:8	7:0	31:24	23:16	15:8	7:0
32 bit	00	1st	aa	bb	cc	dd	dd	cc	bb	aa
16 bit	00	1st	--	--	cc	dd	--	--	bb	aa
		2nd	--	--	aa	bb	--	--	dd	cc
8 bit	00	1st	--	--	--	dd	--	--	--	aa
		2nd	--	--	--	cc	--	--	--	bb
		3rd	--	--	--	bb	--	--	--	cc
		4th	--	--	--	aa	--	--	--	dd

Half Word Data Type			Bus Pins (Data Lines 31:0)							
Bus Width	EFA A[1:0]	Xfer	Little Endian				Big Endian			
			31:24	23:16	15:8	7:0	31:24	23:16	15:8	7:0
32 bit	00	1st	--	--	cc	dd	--	--	dd	cc
	10	1st	cc	dd	--	--	dd	cc	--	--
16 bit	X0	1st	--	--	cc	dd	--	--	dd	cc
8 bit	X0	1st	--	--	--	dd	--	--	--	cc
		2nd	--	--	--	cc	--	--	--	dd

Byte Data Type			Bus Pins (Data Lines 31:0)							
Bus Width	EFA A[1:0]	Xfer	Little and Big Endian							
			31:24		23:16		15:8		7:0	
32 bit	00	1st	--	--	--	--	--	--	--	dd
	01	1st	--	--	--	--	--	--	--	--
	10	1st	--	--	dd	--	--	--	--	--
	11	1st	dd	--	--	--	--	--	--	--
16 bit	X0	1st	--	--	--	--	--	--	--	dd
	X1	1st	--	--	--	--	--	--	--	--
8 bit	XX	1st	--	--	--	--	--	--	--	dd

## 15.5 Atomic Memory Operations (The LOCK# Signal)

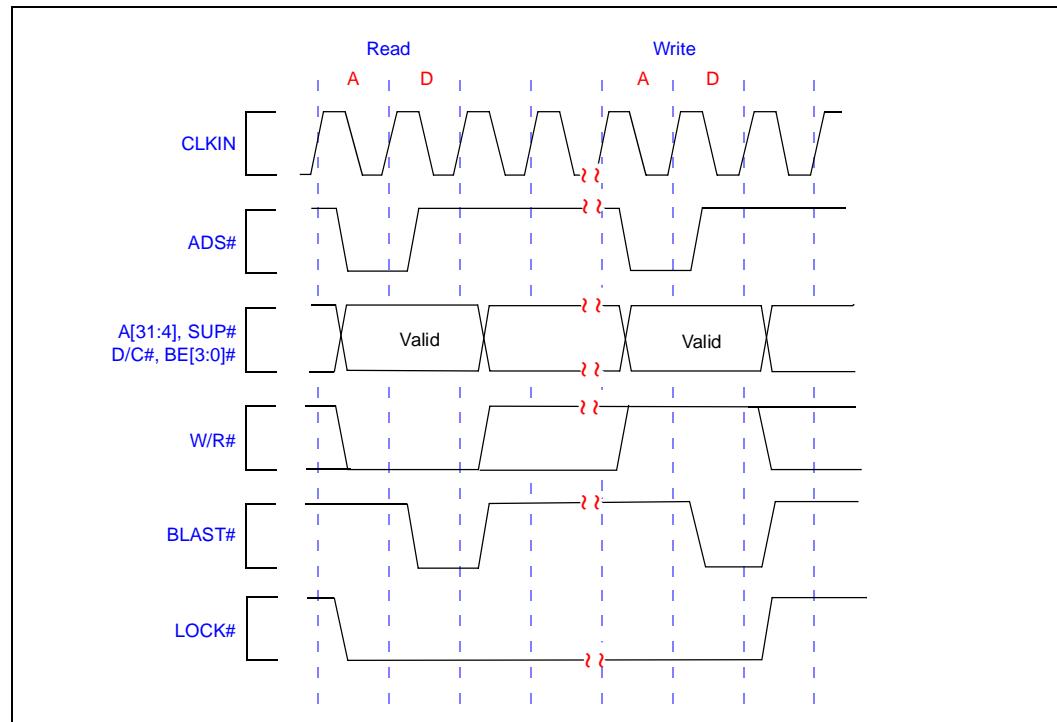
LOCK# output assertion indicates that the processor is executing an atomic read-modify-write operation. Atomic instructions (**atadd**, **atmod**) provide indivisible memory accesses. Atomic instructions consist of a load and store request to the same memory location. LOCK# is asserted in the first address cycle of the load request and deasserted in the cycle after the last data transfer of the store request. The LOCK# pin is not active during the N<sub>XDA</sub> states for the store request. During this operation, another bus agent must not access the target of the atomic instruction between read and write cycles.

When implementing a locked memory subsystem, consider the interaction that the following mechanisms may have with the system. A system must account for these conditions during locked accesses:

- HOLD requests are not acknowledged while LOCK# is asserted.
- An atomic load or store may be suspended using the BOFF# input.

LOCK# indicates that other agents must not write data to any address falling within the quad word boundary of the address on the bus when LOCK# was asserted. LOCK# is deasserted after the write portion of an atomic access. (See [Figure 15-19](#).)

**Figure 15-19. The LOCK# Signal**



## 15.6 External Bus Arbitration

The i960 Hx processor provides a shared bus protocol to allow another bus master to access the processor's bus. The two methods for an external agent to acquire the bus are using the HOLD/HOLDA pins or the BOFF# signal. When the processor relinquishes control of the bus, the data, address and control lines are floated to allow the external bus master to control the bus and memory interface. HOLD/HOLDA allow the processor to finish its current access(es) before releasing the bus. BOFF# stops the current access and gives the external agent immediate control. Additionally, the Bus Request (BREQ) and Bus Stall (BSTALL) signals provide status information for arbitration logic.

### 15.6.1 The HOLD and HOLDA Signals

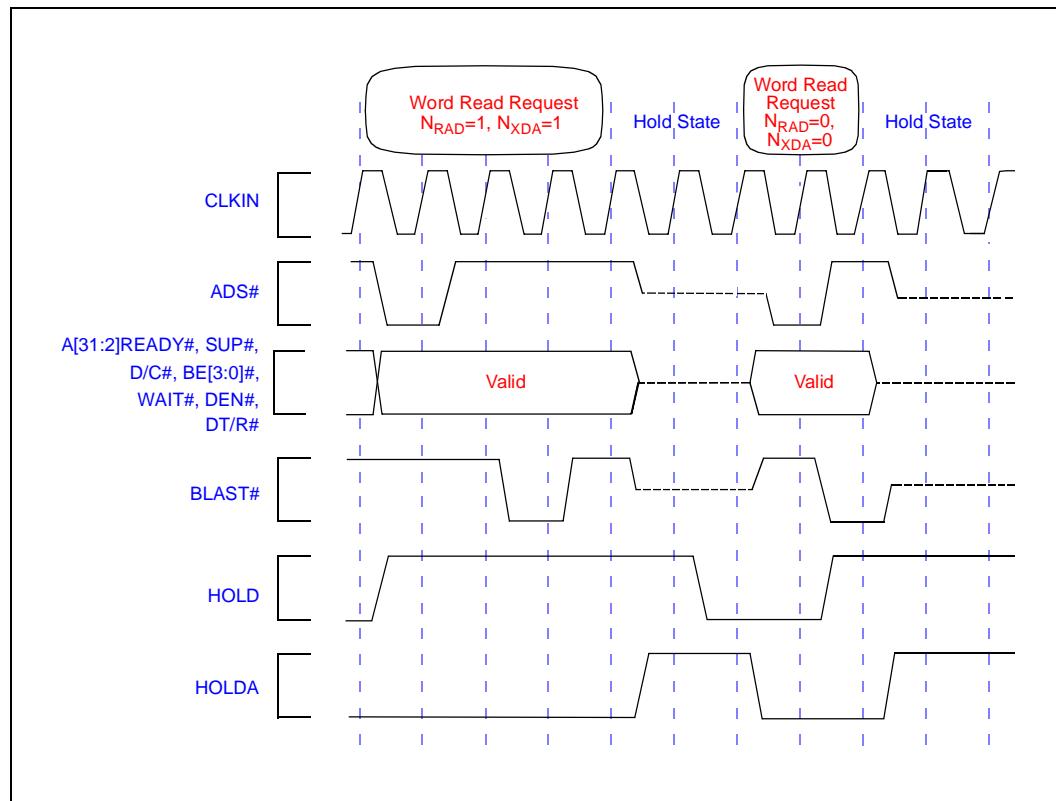
The HOLD input signal is asserted when another processor or peripheral is requesting control of the bus. The HOLDA (Hold Acknowledge) output signal acknowledges that the i960 Hx processor has relinquished the bus. Bus pins float on the same clock cycle in which the hold request is granted (HOLDA asserted). The i960 Hx processor uses the bus request signal (BREQ) to tell the other processor or peripheral when it needs to access the bus. If the processor is stalled while waiting for the bus, then BSTALL is asserted as well. When the HOLD signal is asserted, the i960 Hx processor grants the hold request (asserts HOLDA) and relinquishes control as follows:

- If the bus is in the idle state, the hold request is granted immediately.
- If a bus request is being serviced and LOCK# is not asserted, the hold request is granted at the end of the current bus request.
- If a bus request is being serviced and LOCK# is asserted, the hold request is granted after the request that releases LOCK#.
- If the processor is in the backoff state (BOFF# pin asserted), the hold request is granted after BOFF# is deasserted and the resumed request has completed.

When the HOLD signal is removed, HOLDA is deasserted on the following CLKIN cycle and the bus and control signals are driven. The HOLD signal is a synchronous input. Setup and hold times for this input are given in the *80960HA/HD/HT Embedded 32-bit Microprocessor* datasheet.

HOLD and HOLDA arbitration can also function during the reset state. The bus controller acknowledges HOLD while RESET# is asserted. If RESET# is asserted while HOLDA is asserted (the processor has acknowledged the HOLD), the processor remains in the HOLDA state. The processor does not go into the reset state until HOLD is removed and the processor removes HOLDA.

**Figure 15-20. HOLD/HOLDA Bus Arbitration**



## 15.6.2 The BREQ and BSTALL Signals

BREQ indicates that the bus controller queue contains one or more pending bus requests. The bus controller can queue up to three bus requests. When the bus queue is empty, the BREQ pin is deasserted.

Applications can use BREQ to qualify hold requests to optimize the processor's use of the bus when shared by external masters. Because the hold request is granted between bus requests, the bus controller queue may contain one or more entries when the request is granted. BREQ can be used to delay a hold request until all pending bus requests are complete. It should be noted that the processor can continue executing from on-chip cache while in the hold state. It is possible, then, that bus requests may be posted in the queue after the hold request is granted. In this case, the processor asserts BREQ when it needs the bus.

The i960 Hx processor includes the signal BSTALL that indicates when the CPU has stalled due to inability to access the bus during a HOLD or backoff condition. A stall may be caused by:

- Full bus queues resulting in stalled execution of a memory reference instruction
- Data dependency due to an outstanding load request
- Instruction cache miss

BSTALL may be used in combination with BREQ to provide a bus arbiter with additional information. The timing for BSTALL is the same as BREQ (see *80960HA/HD/HT Embedded 32-bit Microprocessor* datasheet).

### 15.6.3 Bus Backoff Function (BOFF# Pin)

The bus backoff input (BOFF#) suspends a bus request already in progress and allows another bus master to take control of the bus temporarily. Assertion of the BOFF# pin suspends the current bus request, and the processor's address, data and status pins are floated on the following clock cycle. At this time, an alternate bus master may take control of the local system bus. When the alternate bus master has completed its accesses, BOFF# is deasserted and the suspended request is resumed upon assertion of ADS# on the following clock cycle. See [Figure 15-22](#).

Backoff can be used only for requests to regions that have the READY#/BTERM# inputs enabled, with the  $N_{RAD}$ ,  $N_{RDD}$ ,  $N_{WAD}$  and  $N_{WDD}$  parameters programmed to 0. If BOFF# is asserted during an access to a memory region where the READY#/BTERM# inputs are disabled, the processor ignores the backoff request.

Note that if the wait state parameters are not programmed to zero in a region where BOFF# is asserted, improper bus controller behavior can occur.

BOFF# may be asserted only during a bus access. External logic must ensure that BOFF# is not asserted during idle bus cycles or during bus turnaround ( $N_{XDA}$ ) cycles, otherwise, unpredictable behavior may occur.

It is possible for HOLD and BOFF# to be asserted in the same clock cycle. When this occurs, BOFF# takes precedence. The bus is relinquished to a hold request only after BOFF# is deasserted and the current request is complete.

Bus backoff is intended for multiprocessor designs or bus architectures that do not implement “collision-free” bus arbitration schemes (such as VME\* and MULTIBUS I\*). A collision occurs when multiple processors begin a bus access simultaneously and a conflict for control of a processor's local memory occurs.

[Figure 15-21](#) illustrates a bus collision. In this system, several processors share a common bus. Each processor has local memory that is connected directly to that processor's address, data and control lines. Each processor can access another processor's local memory over the bus.

Processor A has a higher priority than Processor B for use of the bus. Processor A and B simultaneously request an access over the bus. Processor A attempts to access Processor B's local memory and Processor B attempts to access another memory subsystem on the bus. Use of the bus is granted to Processor A because it is the highest priority. For Processor A to complete its access, the local bus for Processor B must be relinquished (floated). This is accomplished by asserting the BOFF# pin for Processor B.

When BOFF# is asserted, external memory is responsible for cancelling the current access gracefully. This means that the memory control state machine should cancel write cycles and return to an idle state after BOFF# is asserted. The processor ignores read data after BOFF# is asserted.

**Figure 15-21. Example Application of the Bus Backoff Function**

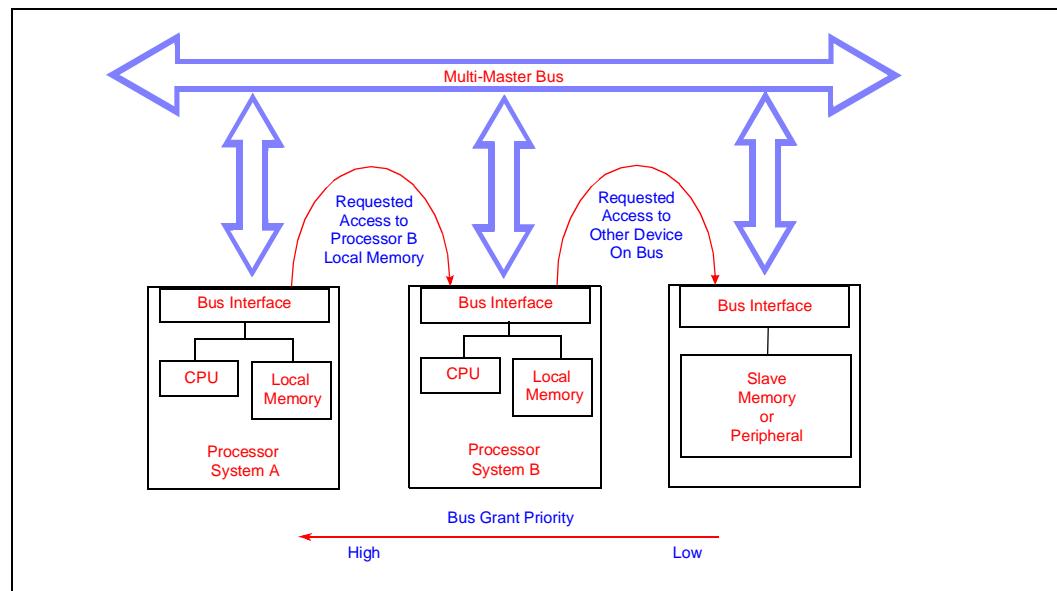
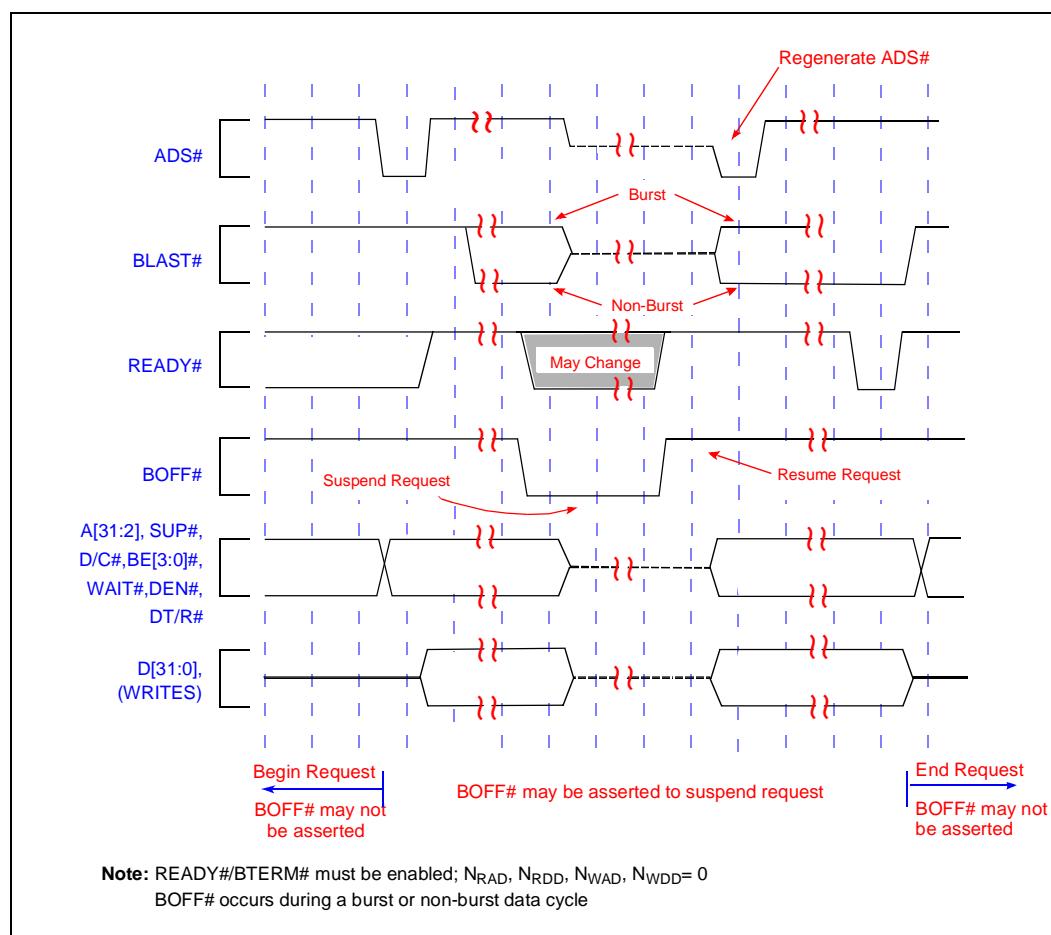


Figure 15-22. Operation of the Bus Backoff Function



This chapter describes the i960® Hx processor's 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.

## 16.1 On-Circuit Emulation (ONCE)

On-circuit emulation aids board-level testing. This feature allows a mounted i960 Hx 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. Once the processor is electrically removed, a functional tester such as an In-Circuit Emulator (ICE™) system can emulate the mounted processor and execute a test of the i960 Hx processor system.

**Note:** Do not use ONCE mode with boundary-scan (JTAG). See [Section 16.1.2, “ONCE Mode and Boundary-Scan \(JTAG\) are Incompatible” on page 16-2](#).

### 16.1.1 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. Anytime 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 *80960HA/HD/HT Embedded 32-bit Microprocessor* datasheet (Intel Literature order #272495) for specific timing of the ONCE# pin and the characteristics of the on-circuit emulation mode.

### 16.1.2 ONCE Mode and Boundary-Scan (JTAG) are Incompatible

Permanent damage can occur if 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 Hx processor signals while JTAG is active.

Since the i960 Hx 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.

## 16.2 Boundary-Scan (JTAG)

The i960 Hx 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 RP 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).

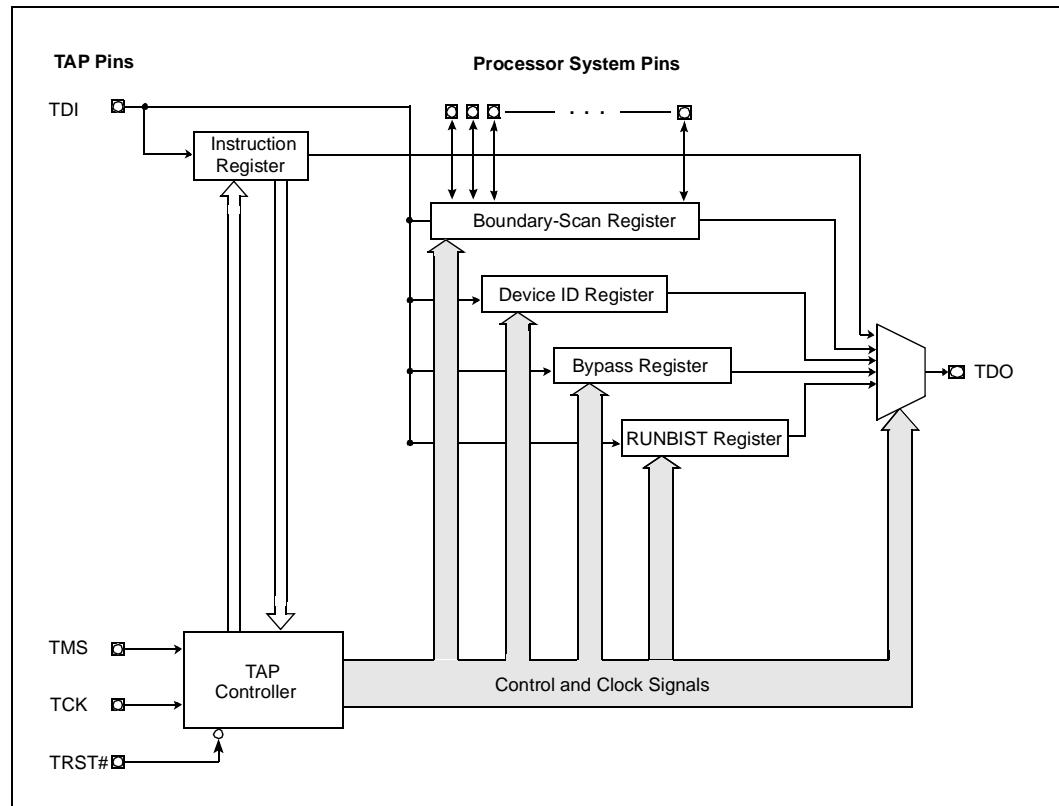
**Note:** Do not use ONCE mode with boundary-scan (JTAG). See [Section 16.1.2, “ONCE Mode and Boundary-Scan \(JTAG\) are Incompatible” on page 16-2](#).

## 16.2.1 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 16-1](#) illustrates how these pieces fit together to form the JTAG unit.

**Figure 16-1. Test Access Port Block Diagram**



## 16.2.2 TAP Pins

The i960 Hx 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 16-1](#). The TAP pins provide access to the instruction register and the test data registers.

**Table 16-1. TAP Controller Pin Definitions**

Pin Name	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.

## 16.2.3 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 will terminate.

The instruction determines the test to be performed, the test data register to be accessed, or both (see [Table 16-2](#)). The IR is four bits wide. When the IR is selected in the Shift\_IR state, the most significant bit is connected to TDI, and the least significant bit is connected to TDO. The value presented on the TDI pin is shifted into the IR on each rising edge of TCK, as long as the TAP controller remains in the Shift\_IR state. When the TAP controller changes to the Capture\_IR state, fixed parallel data ( $0001_2$ ) is captured. During Shift\_IR, when a new instruction is shifted in through TDI, the value  $0001_2$  is always shifted out through TDO, least significant bit first. This helps identify instructions in a long chain of serial data from several devices.

Upon activation of the TRST# reset pin, the latched instruction asynchronously changes to the **idcode** instruction. If the TAP controller moved into the Test\_Logic\_Reset state other than by reset activation, the opcode changes as TDI is shifted, and becomes active on the falling edge of TCK. See [Figure 16-4](#) for an example of loading the instruction register.

### 16.2.3.1 Boundary-Scan Instruction Set

The i960 Hx processor supports three mandatory boundary-scan instructions (**bypass**, **sample/preload** and **extest**) plus four additional public instructions (**idcode**, **clamp**, **highz** and **runbist**). [Table 16-2](#) lists the i960 Hx processor's boundary-scan instruction codes. Those codes listed as "not used" or "private" should not be used.

**Table 16-2. Boundary-Scan Instruction Set**

Instruction Code	Instruction Name	Instruction Code	Instruction Name
$0000_2$	<b>extest</b>	$1000_2$	<b>highz</b>
$0001_2$	<b>sample/preload</b>	$1001_2$	not used
$0010_2$	<b>idcode</b>	$1010_2$	not used
$0011_2$	not used	$1011_2$	private
$0100_2$	<b>clamp</b>	$1100_2$	private
$0101_2$	not used	$1101_2$	not used
$0110_2$	not used	$1110_2$	not used
$0111_2$	<b>runbist</b>	$1111_2$	<b>bypass</b>

**Table 16-3. IEEE Instructions (Sheet 1 of 2)**

<b>Instruction / Requisite</b>	<b>Opcode</b>	<b>Description</b>
<b>extest</b> IEEE 1149.1 Required	$0000_2$	<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	$0001_2$	<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 then applied to the system outputs by means of the <b>extest</b> instruction.</li> </ul>
<b>idcode</b> IEEE 1149.1 Optional	$0010_2$	<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 Hx processor Optional	$0111_2$	<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 $V_{CC}$ 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.

**Table 16-3. IEEE Instructions (Sheet 2 of 2)**

<b>Instruction / Requisite</b>	<b>Opcode</b>	<b>Description</b>
<b>bypass</b> IEEE 1149.1 Required	$1111_2$	<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. $0_2$ is captured in the CAPTURE_DR state. This is the only instruction that accesses the bypass register. While this instruction is in effect, all other test data registers have no effect on system operation. Test data registers with both test and system functionality perform their system functions when this instruction is selected.
<b>highz</b>	$1000_2$	The execution of <b>highz</b> generates a signal that is read on the rising-edge of RESET#. If 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>	$0100_2$	<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 will not change while the <b>clamp</b> instruction is selected.

## 16.2.4 TAP Test Data Registers

The i960 Hx 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 16-5](#) for an example of loading the data register.

### 16.2.4.1 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 [Figure 13-8](#). The format of the register is discussed in [Section 13.4, Device Identification on Reset \(pg. 13-23\)](#). [Table 13-8](#) lists the codes corresponding to the i960 Hx processor. 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.

#### 16.2.4.2 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 i960 Hx processor itself.

#### 16.2.4.3 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 13.2.2, “Self Test Function \(STEST, FAIL#\)” on page 13-8](#) 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.

#### 16.2.4.4 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 16-4](#) shows the bit order of the i960 Hx processor boundary-scan register. All table cells that contain “Control” select the direction of bidirectional pins or HIGHZ output pins. If 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 Hx 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 16-4. i960 Hx Processor Boundary-Scan Register Bit Order (Sheet 1 of 3)**

<b>Bit</b>	<b>Boundary-Scan Cell</b>	<b>Cell Type</b>	<b>Bit</b>	<b>Boundary-Scan Cell</b>	<b>Cell Type</b>
1	DP3	I/O	26	D15	I/O
2	DP2	I/O	27	D16	I/O
3	DP0	I/O	28	D17	I/O
4	DP1	I/O	29	D18	I/O
5	STEST	I	30	D19	I/O
6	FAIL#	O	31	D20	I/O
7	FAIL#, BREQ, BSTALL enable cell	Control	32	D21	I/O
8	ONCE#	I	33	D22	I/O
9	BOFF#	I	34	D23	I/O
10	D0	I/O	35	D24	I/O
11	D1	I/O	36	D25	I/O
12	D2	I/O	37	D26	I/O
13	D3	I/O	38	D27	I/O
14	D4	I/O	39	D28	I/O
15	D5	I/O	40	D29	I/O
16	D6	I/O	41	D30	I/O
17	D7	I/O	42	D31	I/O
18	D[31:0], DP[3:0] enable cell	Control	43	BTERM#	I
19	D8	I/O	44	RDY#	I
20	D9	I/O	45	HOLD	I
21	D10	I/O	46	HOLDA	O
22	D11	I/O	47	HOLDA enable cell	Control
23	D12	I/O	48	ADS#	O
24	D13	I/O	49	BE3#	O
25	D14	I/O	50	BE2#	O

**NOTES:**

1. Cell #1 connects to TDO and cell # 112 connects to TDI.
2. All outputs are three-states.
3. For output-only signals, a logic 1 on the Enable signal enables the output. A logic 0 three-states the output.
4. For bi-directional signals, a logic 1 on the Enable signal enables the output. A logic 0 selects the input direction.

**Table 16-4. i960 Hx Processor Boundary-Scan Register Bit Order (Sheet 2 of 3)**

Bit	Boundary-Scan Cell	Cell Type	Bit	Boundary-Scan Cell	Cell Type
51	BE1#	O	75	A21	O
52	BE0#	O	76	A20	O
53	BLAST#	O	77	A19	O
54	DEN#	O	78	A18	O
55	W/R#	O	79	A17	O
56	DTR#	O	80	A16	O
57	Enable for DTR#	Control	81	A[31:2], CT[3:0] enable cell	Control
58	WAIT#	O	82	A15	O
59	BSTALL	O	83	A14	O
60	D/C#	O	84	A13	O
61	SUP#	O	85	A12	O
62	SUP#, LOCK#, D/C#, WAIT#, W/R#, DEN#, BLAST#, BE[3:0]#, ADS# enable cell	Control	86	A11	O
63	LOCK#	O	87	A10	O
64	BREQ	O	88	A9	O
65	A31	O	89	A8	O
66	A30	O	90	A7	O
67	A29	O	91	A6	O
68	A28	O	92	A5	O
69	A27	O	93	A4	O
70	A26	O	94	A3	O
71	A25	O	95	A2	O
72	A24	O	96	NMI#	I
73	A23	O	97	XINT7#	I
74	A22	O	98	XINT6#	I

## NOTES:

1. Cell #1 connects to TDO and cell # 112 connects to TDI.
2. All outputs are three-states.
3. For output-only signals, a logic 1 on the Enable signal enables the output. A logic 0 three-states the output.
4. For bi-directional signals, a logic 1 on the Enable signal enables the output. A logic 0 selects the input direction.

**Table 16-4. i960 Hx Processor Boundary-Scan Register Bit Order (Sheet 3 of 3)**

Bit	Boundary-Scan Cell	Cell Type	Bit	Boundary-Scan Cell	Cell Type
99	XINT5#	I	106	CLKIN	I
100	XINT4#	I	107	CT3	O
101	XINT3#	I	108	CT2	O
102	XINT2#	I	109	CT1	O
103	XINT1#	I	110	CT0	O
104	XINT0#	I	111	PCHK#	O
105	RESET#	I	112	PCHK# enable	Control

**NOTES:**

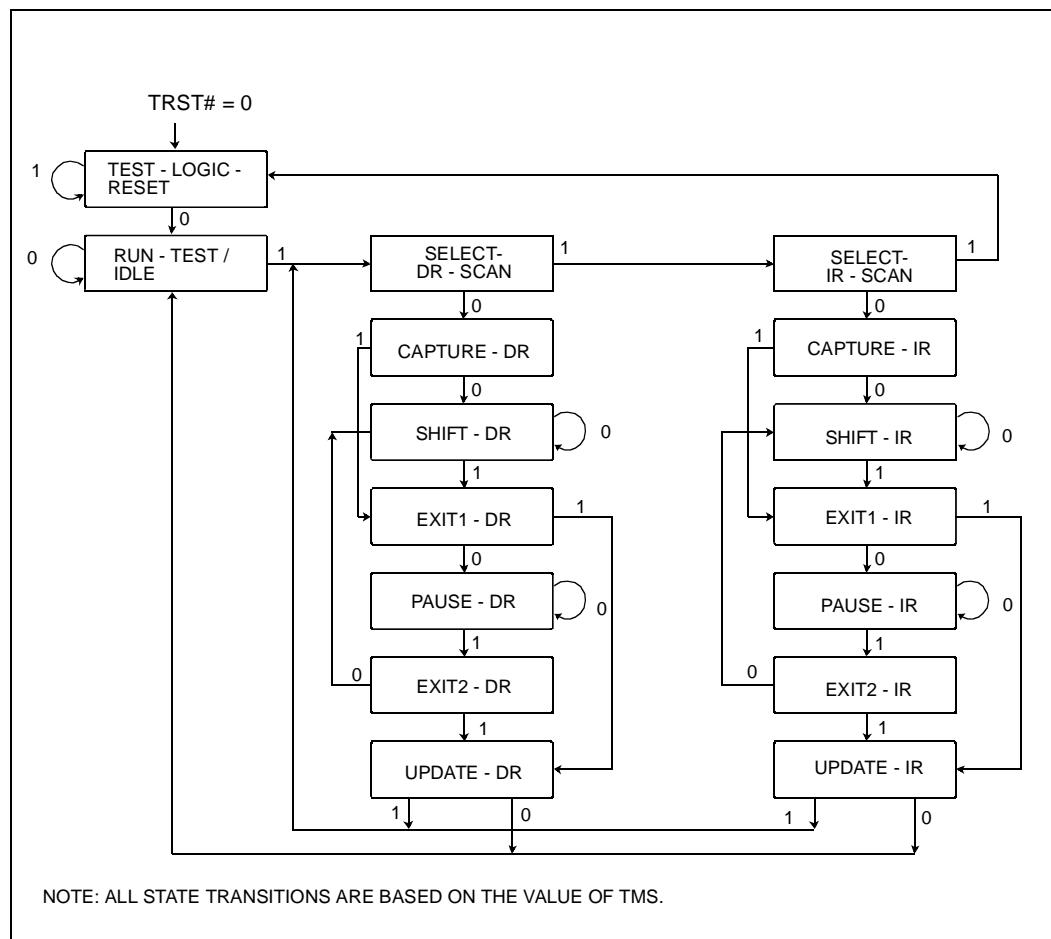
1. Cell #1 connects to TDO and cell # 112 connects to TDI.
2. All outputs are three-states.
3. For output-only signals, a logic 1 on the Enable signal enables the output. A logic 0 three-states the output.
4. For bi-directional signals, a logic 1 on the Enable signal enables the output. A logic 0 selects the input direction.

## 16.2.5 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 16-2](#) for the state diagram of the TAP controller. An uninitialized TAP controller can result in erratic processor behavior even if 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 16-2. TAP Controller State Diagram



### 16.2.5.1 Test Logic Reset State

In this state, test logic is disabled to allow normal operation of the i960 Hx processor. Upon entering the Test\_Logic\_Reset state, the device identification register is loaded. No matter what the present state of the controller, it enters Test-Logic-Reset state when the TMS input is held high ( $1_2$ ) for at least five rising edges of TCK. The controller remains in this state while TMS is high. The TAP controller is also forced to enter this state asynchronously by asserting TRST#.

If 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.

### 16.2.5.2 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. If 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.

### 16.2.5.3 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. If TMS is held low on the rising edge of TCK, the controller moves into the Capture-DR state. If TMS is held high on the rising edge of TCK, the controller moves into the Select-IR-Scan state. See [Section 16.2.5.10, Select-IR Scan State \(pg. 16-15\)](#). The instruction register does not change while the TAP controller is in this state.

### 16.2.5.4 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.

If TMS is high on the rising edge of TCK, the controller enters the Exit1-DR state. If TMS is low on the rising edge of TCK, the controller enters the Shift-DR state.

### 16.2.5.5 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.

If TMS is high on the rising edge of TCK, the controller enters the Exit1-DR state. If TMS is low on the rising edge of TCK, the controller remains in the Shift-DR state.

### 16.2.5.6 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. If 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.

### 16.2.5.7 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.

### 16.2.5.8 Exit2-DR State

Exit2-DR is a temporary state. If TMS is held high on the rising edge of TCK, the controller enters the Update-DR state, which terminates the scanning process. If 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.

### 16.2.5.9 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. If TMS is held low on the rising edge of TCK, the controller re-enters the Run-Test/Idle state.

### 16.2.5.10 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, if 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. If 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.

### 16.2.5.11 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. If TMS is held low on the rising edge of TCK, the controller enters the Shift-IR state.

### 16.2.5.12 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.

If TMS is held high on the rising edge of TCK, the controller enters the Exit1-IR state. If TMS is held low on the rising edge of TCK, the controller remains in the Shift-IR state.

#### 16.2.5.13 Exit1-IR State

This is a temporary state. If TMS is held high on the rising edge of TCK, the controller enters the Update-IR state, which terminates the scanning process. If 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.

#### 16.2.5.14 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.

#### 16.2.5.15 Exit2-IR State

This is a temporary state. If TMS is held high on the rising edge of TCK, the controller enters the Update-IR state, which terminates the scanning process. If 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.

#### 16.2.5.16 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.

If TMS is held high on the rising edge of TCK, the controller re-enters the Select-DR-Scan state. If TMS is held low on the rising edge of TCK, the controller re-enters the Run-Test/Idle state.

## 16.2.6 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 16-3](#) for an illustration of the waveforms involved in this example. The steps are:

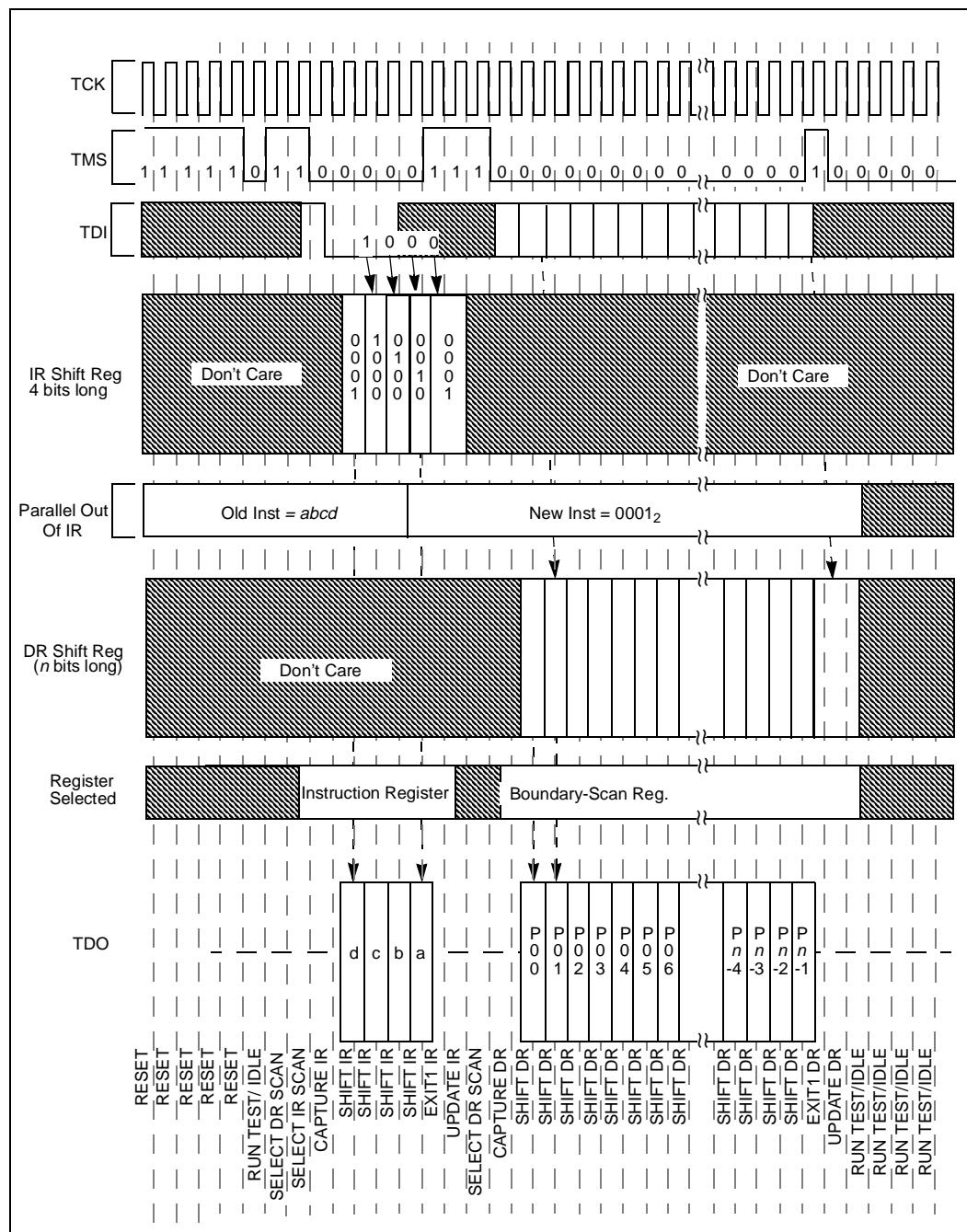
1. Load the **sample/preload** instruction into the instruction register:
  - a. Use TMS to select the Shift-IR state. While in the Shift-IR state, shift in the new instruction, least significant byte first.
  - b. Use the Shift-IR state four times to read the least- through most-significant instruction bits into the instruction register (one does not care what old instruction is being shifted out of the TDO pin).
  - c. Enter the Update-IR state to make the instruction take effect.
2. Capture pin data and shift the data out through the TDO pin:
  - a. Use TMS to select the Select-DR-Scan state.
  - b. Transition the TAP controller to the Capture-DR state to latch pin data in the boundary-scan register cells.
  - c. Enter and stay in the Shift-DR state for 110 TCK cycles. These TDO values are compared against expected data to determine if component operation and connection are correct. Record the TDO values after each cycle. New serial data enters the boundary-scan register through the TDI pin, while old data is scanned out.
  - d. Pass through the Exit1-DR state to the Update-DR state. Here boundary-scan data to be driven out of the system output pins is latched and driven.
  - e. Transition back to the Select-DR state to begin another iteration.

This example does not use Pause states. These states allow software to pause the JTAG state machine to accommodate slow board-level data paths. The Pause states allow indefinite interruptions in the shifting while the external tester performs other tasks.

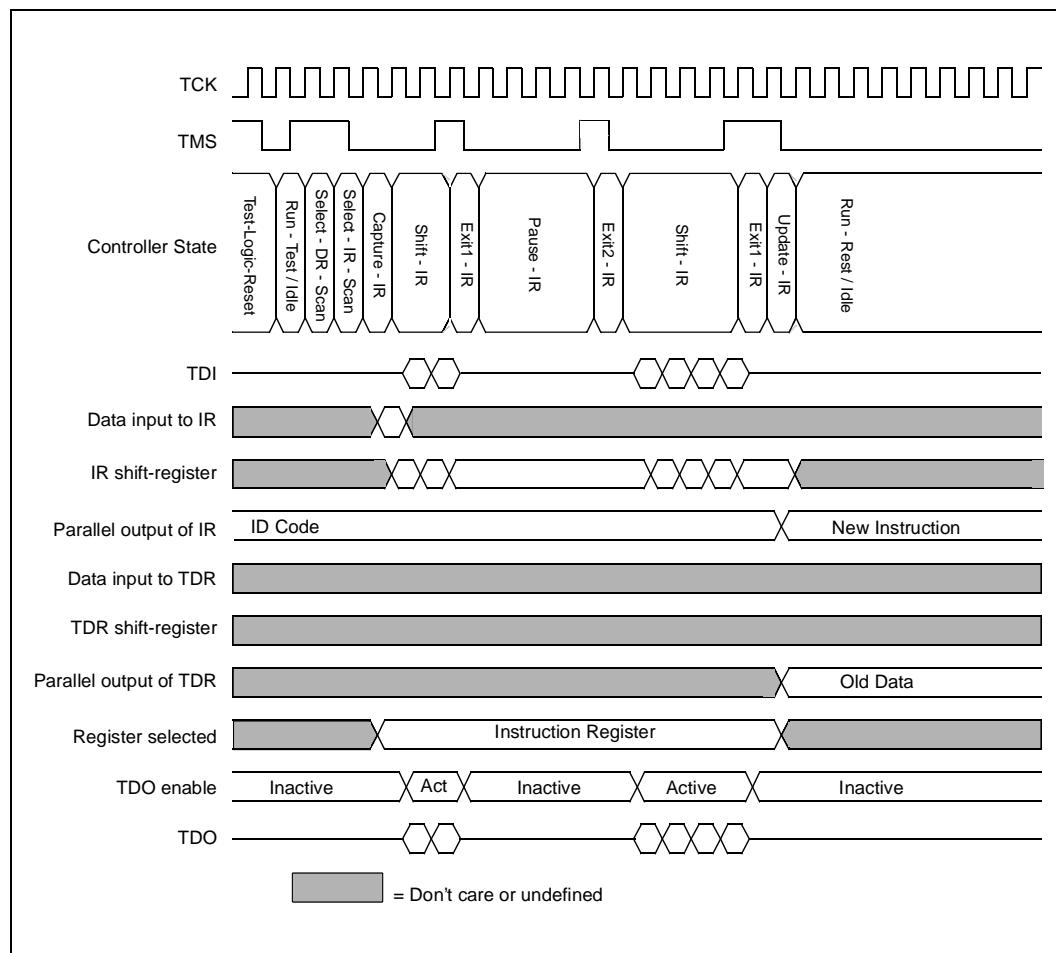
The old instruction was *abcd* in the example. The original instruction register value will be the ID code since the example starts from the reset state. Other times it will represent the previous opcode. The new instruction opcode is  $0001_2$  (**sample/preload**). All pins are captured into the serial 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 then toggles the clock high. Typically, software then drives TCK low quickly. The program then monitors the TDO pin values as they are shifted out.

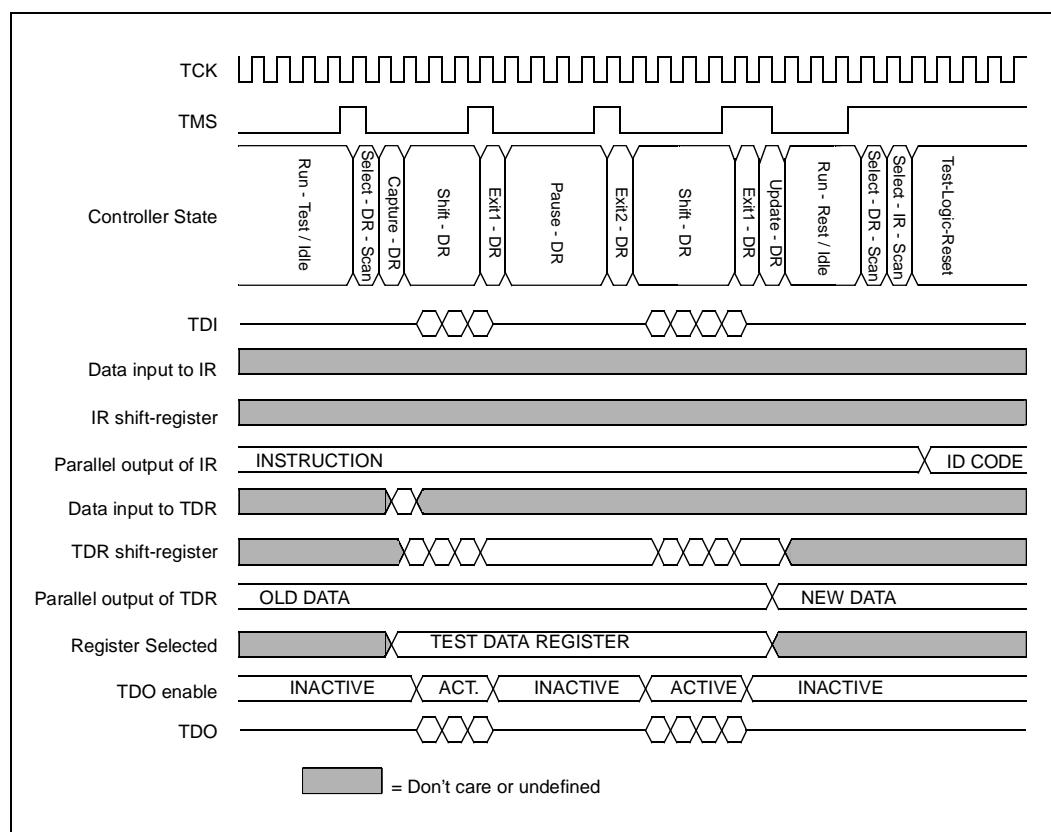
### Figure 16-3. Example Showing Typical JTAG Operations



**Figure 16-4. Timing Diagram Illustrating the Loading of Instruction Register**



**Figure 16-5. Timing Diagram Illustrating the Loading of Data Register**



## 16.2.7 Boundary-Scan Description Language Example

Boundary-Scan Description Language (BSDL) [Example 16-1](#) meets the de facto standard means of describing essential features of ANSI/IEEE 1149.1-1993 compliant devices.

### Example 16-1. i960 Hx Processor Boundary-Scan Description Language (BSDL) Example (Sheet 1 of 5)

```
-- i960 (R) Hx Processor BSDL Model

-- Rev 0.6      08 Dec 1994
-- The following list describes all of the pins that are contained in the
-- i960 Hx microprocessor.

entity Ha_Processor is
    generic(PHYSICAL_PIN_MAP : string:= "PGA");

    port (A          : out     bit_vector(2 to 31);
          ADSBAR    : out     bit;
          BEBAR     : out     bit_vector(0 to 3);
          BLASTBAR  : out     bit;
          BOFFBAR   : in      bit;
          BREQ      : out     bit;
          BSTALL    : out     bit;
          BTERM BAR : in      bit;
          CT         : out     bit_vector(0 to 3);
          CLKIN     : in      bit;
          D          : inout   bit_vector(0 to 31);
          DENBAR    : out     bit;
          DP         : inout   bit_vector(0 to 3);
          DTRBAR    : out     bit;
          DCBAR     : out     bit;
          FAILBAR   : out     bit;
          HOLD      : in      bit;
          HOLDA     : out     bit;
          LOCKBAR   : out     bit;
          NMIBAR    : in      bit;
          ONCEBAR   : in      bit;
          PCHKBAR   : out     bit;
          READYBAR  : in      bit;
          RESETBAR  : in      bit;
          STEST     : in      bit;
          SUPBAR    : out     bit;
          TCK        : in      bit;
          TDI        : in      bit;
          TDO        : out     bit;
          TMS        : in      bit;
          TRST       : in      bit;
          WAITBAR   : out     bit;
          WRBAR     : out     bit;
          XINTBAR   : in      bit_vector(0 to 7);
          FIVEVREF  : linkage bit;
          VCCPLL    : linkage bit;
          VOLTDDET  : linkage bit;
          VCC1       : linkage bit_vector(0 to 23);
          VCC2       : linkage bit_vector(0 to 20);
          VSS1       : linkage bit_vector(0 to 25);
          VSS2       : linkage bit_vector(0 to 22);
          NC         : linkage bit_vector(0 to 4)
```

### Example 16-1. i960 Hx Processor Boundary-Scan Description Language (BSDL) Example (Sheet 2 of 5)

```

);
use STD_1149_1_1990.all;
use i960ha_a.all;

attribute PIN_MAP of Ha_Processor : entity is PHYSICAL_PIN_MAP;

constant PGA:PIN_MAP_STRING :=
  "A      : (D16, D17, E16, E17, F17, G16, G17, H17, J17, K17,"&
  "        L17, L16, M17, N17, N16, P17, Q17, P16, P15, Q16,"&
  "        R17, R16, Q15, S17, R15, S16, Q14, R14, Q13, S15),"&
  "ADSBAR   : R06,"&
  "BEBAR    : (R09, S07, S06, S05),"&
  "BLASTBAR : S08,"&
  "BOFFBAR  : B01,"&
  "BREQ     : R13,"&
  "BSTALL   : R12,"&
  "BTERMBAR : R04,"&
  "CT       : (A11, A12, A13, A14),"&
  "CLKIN   : C13,"&
  "D      : (E03, C02, D02, C01, E02, D01, F02, E01, F01, G01,"&
  "        H02, H01, J01, K01, L02, L01, M01, N01, N02, P01,"&
  "        P02, Q01, P03, Q02, R01, S01, Q03, R02, Q04, S02,"&
  "        Q05, R03),"&
  "DENBAR   : S09,"&
  "DP       : (A03, B03, A04, B04),"&
  "DTRBAR   : S11,"&
  "DCBAR    : S13,"&
  "FAILBAR  : A02,"&
  "HOLD     : R05,"&
  "HOLDA   : S04,"&
  "LOCKBAR  : S14,"&
  "NMIBAR   : D15,"&
  "ONCEBAR  : C03,"&
  "PCHKBAR : B08,"&
  "READYBAR : S03,"&
  "RESETBAR : A16,"&
  "STEST    : B02,"&
  "SUPBAR   : Q12,"&
  "TCK     : B05,"&
  "TDI     : A07,"&
  "TDO     : A08,"&
  "TMS     : B06,"&
  "TRST    : A06,"&
  "WAITBAR  : S12,"&
  "WRBAR   : S10,"&
  "XINTBAR  : (B15, A15, A17, B16, C15, B17, C16, C17),"&
  "FIVEVREF : C05,"&
  "VOLTDET  : A05,"&
  "VCCPLL   : B10,"&
  "VCC1     : (M02, K02, J02, G02, N03, F03, C06, B07, B09, B11,"&
  "        B12, C14, E15, F16, H16, J16, K16, M16, N15, Q06,"&
  "        R07, R08, R10, R11),"&

```

**Example 16-1. i960 Hx Processor Boundary-Scan Description Language (BSDL) Example  
(Sheet 3 of 5)**

```

"VSS1      : (G03, H03, J03, K03, L03, M03, C07, C08, C09, C10,"&
"           C11, C12, Q07, Q08, Q09, Q10, Q11, F15, G15, H15,"&
"           J15, K15, L15, M15, A01, C04),"&
"NC       : (A09, A10, B13, B14, D03)";

attribute Tap_Scan_In    of TDI    : signal is true;
attribute Tap_Scan_Mode   of TMS    : signal is true;
attribute Tap_Scan_Out    of TDO    : signal is true;
attribute Tap_Scan_Reset   of TRST   : signal is true;
attribute Tap_Scan_Clock   of TCK    : signal is (66.0e6, BOTH);

attribute Instruction_Length of Ha_Processor: entity is 4;

attribute Instruction_Opcode of Ha_Processor: entity is

"BYPASS      (1111)," &
"EXTEST      (0000)," &
"SAMPLE      (0001)," &
"IDCODE      (0010)," &
"RUBIST      (0111)," &
"CLAMP       (1011)," &
"HIGHZ       (1100)";

attribute Instruction_Capture of Ha_Processor: entity is "0001";

attribute Instruction_Private of Ha_Processor: entity is "Reserved" ;

attribute Idcode_Register of Ha_Processor: entity is
"0000"                      & --version,
"1000100001000000"          & --part number
"00000001001"                & --manufacturers identity
"1";                         --required by the standard

attribute Register_Access of Ha_Processor: entity is
"Runbist[32]      (RUBIST)," &
"Bypass          (CLAMP, HIGHZ)";

--{*****}
--{ The first cell, cell 0, is closest to TDO }
--{ BC_1:Control, Output3 CBSC_1:Bidir BC_4: Input, Clock }
--{*****}

attribute Boundary_Cells of Ha_Processor: entity is "BC_4, BC_1, CBSC_1";
attribute Boundary_Length of Ha_Processor: entity is 112;
attribute Boundary_Register of Ha_Processor: entity is

"0 (CBSC_1, DP(3),      bidir, X, 17, 1, Z)," &
"1 (CBSC_1, DP(2),      bidir, X, 17, 1, Z)," &
"2 (CBSC_1, DP(0),      bidir, X, 17, 1, Z)," &
"3 (CBSC_1, DP(1),      bidir, X, 17, 1, Z)," &
"4 (BC_4,     STEST,      input, X)," &
"5 (BC_1,     FAILBAR,    output3, X, 6, 1, Z)," &
"6 (BC_1,     *,          control, 1)," &
"7 (BC_4,     ONCEBAR,    input, X)," &
"8 (BC_4,     BOFFBAR,    input, X)," &
"9 (CBSC_1,   D(0),      bidir, X, 17, 1, Z)," &

```

### Example 16-1. i960 Hx Processor Boundary-Scan Description Language (BSDL) Example (Sheet 4 of 5)

"10 (CBSC_1, D(1),	bidir, X, 17, 1, Z)," &
"11 (CBSC_1, D(2),	bidir, X, 17, 1, Z)," &
"12 (CBSC_1, D(3),	bidir, X, 17, 1, Z)," &
"13 (CBSC_1, D(4),	bidir, X, 17, 1, Z)," &
"14 (CBSC_1, D(5),	bidir, X, 17, 1, Z)," &
"15 (CBSC_1, D(6),	bidir, X, 17, 1, Z)," &
"16 (CBSC_1, D(7),	bidir, X, 17, 1, Z)," &
"17 (BC_1, *,	control, 1)," &
"18 (CBSC_1, D(8),	bidir, X, 17, 1, Z)," &
"19 (CBSC_1, D(9),	bidir, X, 17, 1, Z)," &
"20 (CBSC_1, D(10),	bidir, X, 17, 1, Z)," &
"21 (CBSC_1, D(11),	bidir, X, 17, 1, Z)," &
"22 (CBSC_1, D(12),	bidir, X, 17, 1, Z)," &
"23 (CBSC_1, D(13),	bidir, X, 17, 1, Z)," &
"24 (CBSC_1, D(14),	bidir, X, 17, 1, Z)," &
"25 (CBSC_1, D(15),	bidir, X, 17, 1, Z)," &
"26 (CBSC_1, D(16),	bidir, X, 17, 1, Z)," &
"27 (CBSC_1, D(17),	bidir, X, 17, 1, Z)," &
"28 (CBSC_1, D(18),	bidir, X, 17, 1, Z)," &
"29 (CBSC_1, D(19),	bidir, X, 17, 1, Z)," &
"30 (CBSC_1, D(20),	bidir, X, 17, 1, Z)," &
"31 (CBSC_1, D(21),	bidir, X, 17, 1, Z)," &
"32 (CBSC_1, D(22),	bidir, X, 17, 1, Z)," &
"33 (CBSC_1, D(23),	bidir, X, 17, 1, Z)," &
"34 (CBSC_1, D(24),	bidir, X, 17, 1, Z)," &
"35 (CBSC_1, D(25),	bidir, X, 17, 1, Z)," &
"36 (CBSC_1, D(26),	bidir, X, 17, 1, Z)," &
"37 (CBSC_1, D(27),	bidir, X, 17, 1, Z)," &
"38 (CBSC_1, D(28),	bidir, X, 17, 1, Z)," &
"39 (CBSC_1, D(29),	bidir, X, 17, 1, Z)," &
"40 (CBSC_1, D(30),	bidir, X, 17, 1, Z)," &
"41 (CBSC_1, D(31),	bidir, X, 17, 1, Z)," &
"42 (BC_4, BTERMBAR,	input, X)," &
"43 (BC_4, READYBAR,	input, X)," &
"44 (BC_4, HOLD,	input, X)," &
"45 (BC_1, HOLDA,	output3, X, 46, 1, Z)," &
"46 (BC_1, *,	control, 1)," &
"47 (BC_1, ADSBAR,	output3, X, 61, 1, Z)," &
"48 (BC_1, BEBAR(3),	output3, X, 61, 1, Z)," &
"49 (BC_1, BEBAR(2),	output3, X, 61, 1, Z)," &
"50 (BC_1, BEBAR(1),	output3, X, 61, 1, Z)," &
"51 (BC_1, BEBAR(0),	output3, X, 61, 1, Z)," &
"52 (BC_1, BLASTBAR,	output3, X, 61, 1, Z)," &
"53 (BC_1, DENBAR,	output3, X, 61, 1, Z)," &
"54 (BC_1, WRBAR,	output3, X, 61, 1, Z)," &
"55 (BC_1, DTRBAR,	output3, X, 56, 1, Z)," &
"56 (BC_1, *,	control, 1)," &
"57 (BC_1, WAITBAR,	output3, X, 61, 1, Z)," &
"58 (BC_1, BSTALL,	output3, X, 6, 1, Z)," &
"59 (BC_1, DCBAR,	output3, X, 61, 1, Z)," &
"60 (BC_1, SUPBAR,	output3, X, 61, 1, Z)," &
"61 (BC_1, *,	control, 1)," &
"62 (BC_1, LOCKBAR,	output3, X, 61, 1, Z)," &
"63 (BC_1, BREQ,	output3, X, 6, 1, Z)," &
"64 (BC_1, A(31),	output3, X, 80, 1, Z)," &
"65 (BC_1, A(30),	output3, X, 80, 1, Z)," &
"66 (BC_1, A(29),	output3, X, 80, 1, Z)," &

**Example 16-1. i960 Hx Processor Boundary-Scan Description Language (BSDL) Example  
(Sheet 5 of 5)**

"67 (BC_1, A(28), output3, X, 80, 1, Z)," &
"68 (BC_1, A(27), output3, X, 80, 1, Z)," &
"69 (BC_1, A(26), output3, X, 80, 1, Z)," &
"70 (BC_1, A(25), output3, X, 80, 1, Z)," &
"71 (BC_1, A(24), output3, X, 80, 1, Z)," &
"72 (BC_1, A(23), output3, X, 80, 1, Z)," &
"73 (BC_1, A(22), output3, X, 80, 1, Z)," &
"74 (BC_1, A(21), output3, X, 80, 1, Z)," &
"75 (BC_1, A(20), output3, X, 80, 1, Z)," &
"76 (BC_1, A(19), output3, X, 80, 1, Z)," &
"77 (BC_1, A(18), output3, X, 80, 1, Z)," &
"78 (BC_1, A(17), output3, X, 80, 1, Z)," &
"79 (BC_1, A(16), output3, X, 80, 1, Z)," &
"80 (BC_1, *, control, 1)," &
"81 (BC_1, A(15), output3, X, 80, 1, Z)," &
"82 (BC_1, A(14), output3, X, 80, 1, Z)," &
"83 (BC_1, A(13), output3, X, 80, 1, Z)," &
"84 (BC_1, A(12), output3, X, 80, 1, Z)," &
"85 (BC_1, A(11), output3, X, 80, 1, Z)," &
"86 (BC_1, A(10), output3, X, 80, 1, Z)," &
"87 (BC_1, A(9), output3, X, 80, 1, Z)," &
"88 (BC_1, A(8), output3, X, 80, 1, Z)," &
"89 (BC_1, A(7), output3, X, 80, 1, Z)," &
"90 (BC_1, A(6), output3, X, 80, 1, Z)," &
"91 (BC_1, A(5), output3, X, 80, 1, Z)," &
"92 (BC_1, A(4), output3, X, 80, 1, Z)," &
"93 (BC_1, A(3), output3, X, 80, 1, Z)," &
"94 (BC_1, A(2), output3, X, 80, 1, Z)," &
"95 (BC_4, NMIBAR, input, X)," &
"96 (BC_4, XINTBAR(7), input, X)," &
"97 (BC_4, XINTBAR(6), input, X)," &
"98 (BC_4, XINTBAR(5), input, X)," &
"99 (BC_4, XINTBAR(4), input, X)," &
"100(BC_4, XINTBAR(3), input, X)," &
"101(BC_4, XINTBAR(2), input, X)," &
"102(BC_4, XINTBAR(1), input, X)," &
"103(BC_4, XINTBAR(0), input, X)," &
"104(BC_4, RESETBAR, input, X)," &
"105(BC_4, CLKIN, input, X)," &
"106(BC_1, CT(3), output3, X, 80, 1, Z)," &
"107(BC_1, CT(2), output3, X, 80, 1, Z)," &
"108(BC_1, CT(1), output3, X, 80, 1, Z)," &
"109(BC_1, CT(0), output3, X, 80, 1, Z)," &
"110(BC_1, PCHKBAR, output3, X, 111, 1, Z)," &
"111(BC_1, *, control, 1);"
end Ha_Processor;





# ***Considerations for Writing Portable Code***

**A**

---

This appendix describes the aspects of the microprocessor that are implementation-dependent. The following information is intended as a guide for writing application code that is directly portable to other i960® processor implementations.

## **A.1 Core Architecture**

All i960 microprocessor family products are based on the core architecture definition. An i960 processor can be thought of as consisting of two parts: the core architecture implementation and implementation-specific features. The core architecture defines the following mechanisms and structure:

- Programming environment: global and local registers, literals, processor state registers, data types, memory addressing modes, etc.
- Implementation-independent instruction set.
- Procedure call mechanism.
- Mechanism for servicing interrupts and the interrupt and process priority structure.
- Mechanism for handling faults and the implementation-independent fault types and subtypes.

Implementation-specific features are one or all of:

- Additions to the instruction set beyond the instructions defined by the core architecture.
- Extensions to the register set beyond the global, local and processor-state registers that are defined by the core architecture.
- On-chip program or data memory.
- Integrated peripherals that implement features not defined explicitly by the core architecture.

Code is directly portable (object-code compatible) when it does not depend on implementation-specific instructions, mechanisms or registers. The aspects of this microprocessor that are implementation dependent are described below. Those aspects not described below are part of the core architecture.

## A.2 Address Space Restrictions

Address space properties that are implementation-specific to this microprocessor are described in the subsections that follow.

### A.2.1 Reserved Memory

Addresses in the range FF00 0000H to FFFF FFFFH are reserved by the i960 architecture. The i960 Hx processor cannot access this memory, so any use of reserved memory by other i960 processor code is not portable to the i960 Hx processor.

### A.2.2 Initialization Boot Record

The i960 Hx processor uses a section just below the reserved address space for the initialization boot record; see [Section 13.3.1.1, “Initialization Boot Record \(IBR\)” on page 13-13](#). This differs from the i960 Cx processor, which requires that user to place the Initialization Boot Record (IBR) in a section of reserved memory.

The initialization boot record may not exist or may be structured differently for other implementations of the i960 architecture.

### A.2.3 Internal Data RAM

Internal data RAM — an i960 Hx processor implementation-specific feature — is mapped to the first 2 Kbytes of the processor’s address space (0000H – 07FFH). The on-chip data RAM may be used to cache interrupt vectors and may be protected against user and supervisor mode writes. Code that relies on these special features is not directly portable to all i960 processor implementations.

### A.2.4 Instruction Cache

The i960 architecture allows instructions to be cached on-chip in a non-transparent fashion. This means that the cache may not detect modification of the program memory by loads, stores or alteration by external agents. Each implementation of the i960 architecture that uses an integrated instruction cache provides a mechanism to purge the cache or some other method that forces consistency between external memory and internal cache.

This feature is implementation dependent. Application code that supports modification of the code space must use this implementation-specific feature and, therefore, is not object-code portable to all i960 processor implementations.

The i960 HA/HD/HT processor has a 16-Kbyte instruction cache. The instruction cache is purged using the system control (**sysctl**) or instruction cache control (**icctl**) instruction. These instructions are not available on all i960 processors.

An **icctl** instruction invokes the load-and-lock mechanism for one, two, three, or all four 4-Kbyte ways of the instruction cache. Legacy software from the i960 Cx processor can still use the **sysctl** instruction to lock the cache, but with reduced flexibility. **sysctl** can load and lock only one 4 Kbyte way of the instruction cache due to backwards compatibility with the i960 Cx processor definition of the **sysctl** instruction. New software for the i960 Hx processor should use **icctl** for all instruction cache manipulations. With either instruction, when the lock option is selected, the processor loads the cache starting at an address specified as an operand to the instruction.

The i960 Hx processor data cache is 8 Kbytes. Caching may be enabled on a per region basis by programming the LMCN registers. A logical region can be programmed as cacheable with the quick-invalidate option. Not all i960 processors support the quick-invalidate feature.

## A.2.5 Data and Data Structure Alignment

The i960 processor architecture does not define how to handle loads and stores to non-aligned addresses. Therefore, code that generates non-aligned addresses may not be compatible with all i960 processor implementations.

The i960 Hx processor automatically handles non-aligned load and store requests with a combination of microcode and hardware. See [Section 14.3.2, “Bus Transactions across Region Boundaries” on page 14-10](#) for details.

The address boundaries on which an operand begins can affect processor performance. Operands that span more word boundaries than necessary suffer a cost in speed due to extra bus cycles.

Alignment of architecturally defined data structures in memory is implementation dependent. See [Chapter 3, “Programming Environment”](#). Code that relies on specific alignment of data structures in memory is not portable to every i960 processor type.

Stack frames in the i960 processor architecture are aligned on (SALIGN\*16)-byte boundaries, where SALIGN is an implementation-specific parameter. For the i960 Hx processors, SALIGN = 1, so stack frames are aligned on 16-byte boundaries. The low-order N bits of the Frame Pointer are ignored and are always interpreted to be zero. The N parameter is defined by the following expression: SALIGN\*16 =  $2^N$ . Thus for the i960 Hx processors, N is 4.

## A.3 Reserved Locations in Registers and Data Structures

Some register and data structure fields are defined as reserved locations. A reserved field may be used by future implementations of the i960 architecture. For portability and compatibility, code should initialize reserved locations to zero. When an implementation uses a reserved location, the implementation-specific feature is activated by a value of 1 in the reserved field. Setting the reserved locations to 0 guarantees that the features are disabled.

## A.4 Instruction Set

The i960 architecture defines a comprehensive instruction set. Code that uses only the architecturally-defined instruction set is object-level portable to other implementations of the i960 architecture. Some implementations may favor a particular code ordering to optimize performance. This special ordering, however, is never required by an implementation. The following subsections describe implementation-dependent instruction set properties.

### A.4.1 Instruction Timing

An objective of the i960 architecture is to allow micro-architectural advances to translate directly into increased performance. The architecture does not restrict parallel or out-of-order instruction execution, nor does it define the time required to execute any instruction or function. Code that depends on instruction execution times, therefore, is not portable to all i960 processor architecture implementations.

### A.4.2 Implementation-Specific Instructions

Most of the processor's instruction set is defined by the core architecture. Several instructions are specific to the i960 Hx processor. These instructions are either functional extensions to the instruction set or instructions that control implementation-specific functions. [Chapter 6, "Instruction Set Reference"](#) denotes each implementation-specific instruction.

- |                 |                           |                 |                          |
|-----------------|---------------------------|-----------------|--------------------------|
| • <b>dcctl</b>  | Data cache control        | • <b>inten</b>  | Global interrupt enable  |
| • <b>icctl</b>  | Instruction cache control | • <b>intdis</b> | Global interrupt disable |
| • <b>intctl</b> | Interrupt control         | • <b>sysctl</b> | System control           |

In addition, the i960 Hx processor features the **dclnva** instruction, which allows software to invalidate a quad word of data from the D-cache.

Application code using implementation-specific instructions is not directly portable to the entire i960 processor family. Attempted execution of an unimplemented instruction results in an OPERATION.INVALID\_OPCODE fault.

The i960 Jx and Hx processors introduce several new core instructions. These instructions may or may not be supported on other i960 processors. The new core instructions include:

- |                        |                         |                        |                              |
|------------------------|-------------------------|------------------------|------------------------------|
| • <b>ADD&lt;cc&gt;</b> | Conditional add         | • <b>eshro</b>         | Extended shift right ordinal |
| • <b>bswap</b>         | Byte swap               | • <b>SEL&lt;cc&gt;</b> | Conditional select           |
| • <b>COMPARE</b>       | Byte and short compares | • <b>SUB&lt;cc&gt;</b> | Conditional subtract         |

## A.5 Extended Register Set

The i960 processor architecture defines a way to address an extended set of 32 registers in addition to the 16 global and 16 local registers. Some or all of these registers may be implemented on a specific i960 processor.

The i960 Hx processor uses 5 special function registers. Two of these are portable with the i960 Cx processor, sf0 (IPND) and sf1 (IMSK). The i960 Hx processor uses sf2 as the Cache Control special function register which is used as the DMA control register on the i960 Cx processor. For complete descriptions of the special function registers, see [Section 3.2.3, “Special Function Registers \(SFRs\)” on page 3-4](#).

## A.6 Initialization

The i960 architecture does not define an initialization mechanism. The way that an i960-based product is initialized is implementation dependent. Code that accesses locations in initialization data structures is not portable to other i960 processor implementations.

The i960 Hx processors use an initialization boot record (IBR) and a process control block (PRCB) to hold initial configuration and a first instruction pointer.

## A.7 Memory Configuration

The i960 Hx processors employ Physical Memory Control (PMCON) and Logical Memory Control (LMCON) registers to control bus width, byte order and the data cache. This capability is analogous to the MCON register scheme employed by the i960 Cx processor. Memory configurations, like the bus control unit, are implementation specific.

## A.8 Interrupts

The i960 architecture defines the interrupt servicing mechanism. This includes priority definition, interrupt table structure and interrupt context switching that occurs when an interrupt is serviced. The core architecture does not define the means for requesting interrupts (external pins, software, etc.) or for posting interrupts (i.e., saving pending interrupts).

The method for requesting interrupts depends on the implementation. The i960 Hx processors have an interrupt controller that manages nine external interrupt pins. The organization of these pins and the registers of the interrupt controller are implementation specific. Code that configures the interrupt controller is not directly portable to other i960 implementations.

On the i960 Hx processors, interrupts may also be requested in software with the **sysctl** instruction. This instruction and the software request mechanism are implementation specific.

Posting interrupts is also implementation specific. Different implementations may optimize interrupt posting according to interrupt type and interrupt controller configuration. A pending priorities and pending interrupts field is provided in the interrupt table for interrupt posting. However, the i960 Hx processors post hardware-requested interrupts internally in the IPND register instead. Code that requests interrupts by setting bits in the pending priorities and pending interrupts field of the interrupt table is not portable. Also, application code that expects interrupts to be posted in the interrupt table is not object-code portable to all i960-based products.

The i960 Hx processors do not store a resumption record for suspended instructions in the interrupt or fault record. Portable programs must tolerate interrupt stack frames with and without these resumption records.

## A.9 Other i960® Hx Processor Implementation-Specific Features

Subsections that follow describe additional implementation-specific features of the i960 Hx processors. These features do not relate directly to application code portability.

### A.9.1 Data Control Peripheral Units

The bus controller and interrupt controller are implementation-specific extensions to the core architecture. Operation, setup and control of these units is not a part of the core architecture. Other implementations of the i960 architecture are free to augment or modify such system integration features.

### A.9.2 Timers

The i960 Hx processor contains two 32-bit timers that are implementation-specific extensions to the i960 architecture. Code involving operation, setup and control of the timers may or may not be directly portable to other i960 processors.

### A.9.3 Guarded Memory Unit (GMU)

The GMU contains two memory protection schemes to either prevent or merely detect illegal memory accesses. Both schemes signal a fault to the processor (assuming faults are enabled). To date, this feature is found only on the i960 Hx processor.

## A.9.4 Fault Implementation

The architecture defines a subset of fault types and subtypes that apply to all implementations of the architecture. Other fault types and subtypes may be defined by implementations to detect errant conditions that relate to implementation-specific features. For example, the i960 Hx microprocessor provides an OPERATION.UNALIGNED fault for detecting non-aligned memory accesses, a MACHINE.PARITY fault for reporting parity errors and a PROTECTION.BAD\_ACCESS fault for reporting illegal accesses to memory regions protected by the GMU. Future i960 processor implementations that generate these faults are expected to assign the same fault type and subtype numbers to these faults.

## A.10 Breakpoints

Breakpoint registers are not defined in the i960 architecture. The i960 Hx processor implements six instruction and six data breakpoint registers.



## B.1 Instruction Reference by Opcode

This section lists the instruction encoding for each i960® Hx 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	—	<i>src1</i> is a special function register
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	—	<i>src2</i> is a special function register
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 or a special function register when used as a destination. 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	<i>src1</i> is a global or local register, <i>src2</i> and <i>dst</i> are special function registers
—	—	1	1	—	0	<i>src1</i> is a literal, <i>src2</i> and <i>dst</i> are special function registers
<b>COBR Format and CTRL Format</b>						
—	—	x	—	x	0	Outcome of conditional test is predicted to be true.
—	—	x	—	x	1	Outcome of conditional test is predicted to be false.

**Table B-2. REG Format Instruction Encodings** (Sheet 1 of 4)

<b>Opcode</b>	<b>Mnemonic</b>	<b>Cycles to Execute</b>	<b>Opcode (11 - 4)</b>	<b>src/dst</b>	<b>src2</b>	<b>Mode</b>			<b>Opcode (3-0)</b>	<b>Special Flags</b>	<b>src1</b>	
58:0	<b>notbit</b>		31 ..... 24	23 .... 19	18 ... 14	13	12	11	10 .... 7	6	5	4 ..... 0
58:1	<b>and</b>	1	0101 1000	<i>dst</i>	<i>src</i>	M3	M2	M1	0000	S2	S1	<i>bitpos</i>
58:2	<b>andnot</b>	1	0101 1000	<i>dst</i>	<i>src2</i>	M3	M2	M1	0001	S2	S1	<i>src1</i>
58:3	<b>setbit</b>	1	0101 1000	<i>dst</i>	<i>src</i>	M3	M2	M1	0010	S2	S1	<i>src1</i>
58:4	<b>notand</b>	1	0101 1000	<i>dst</i>	<i>src2</i>	M3	M2	M1	0011	S2	S1	<i>bitpos</i>
58:6	<b>xor</b>	1	0101 1000	<i>dst</i>	<i>src2</i>	M3	M2	M1	0110	S2	S1	<i>src1</i>
58:7	<b>or</b>	1	0101 1000	<i>dst</i>	<i>src2</i>	M3	M2	M1	0111	S2	S1	<i>src1</i>
58:8	<b>nor</b>	1	0101 1000	<i>dst</i>	<i>src2</i>	M3	M2	M1	1000	S2	S1	<i>src1</i>
58:9	<b>xnor</b>	1	0101 1000	<i>dst</i>	<i>src2</i>	M3	M2	M1	1001	S2	S1	<i>src1</i>
58:A	<b>not</b>	1	0101 1000	<i>dst</i>		M3	M2	M1	1010	S2	S1	<i>src</i>
58:B	<b>ornot</b>	1	0101 1000	<i>dst</i>	<i>src2</i>	M3	M2	M1	1011	S2	S1	<i>src1</i>
58:C	<b>clrbit</b>	1	0101 1000	<i>dst</i>	<i>src</i>	M3	M2	M1	1100	S2	S1	<i>bitpos</i>
58:D	<b>notor</b>	1	0101 1000	<i>dst</i>	<i>src2</i>	M3	M2	M1	1101	S2	S1	<i>src1</i>
58:E	<b>nand</b>	1	0101 1000	<i>dst</i>	<i>src2</i>	M3	M2	M1	1110	S2	S1	<i>src1</i>
58:F	<b>alterbit</b>	1	0101 1000	<i>dst</i>	<i>src</i>	M3	M2	M1	1111	S2	S1	<i>bitpos</i>
59:0	<b>addo</b>	1	0101 1001	<i>dst</i>	<i>src2</i>	M3	M2	M1	0000	S2	S1	<i>src1</i>
59:1	<b>addi</b>	1	0101 1001	<i>dst</i>	<i>src2</i>	M3	M2	M1	0001	S2	S1	<i>src1</i>
59:2	<b>subo</b>	1	0101 1001	<i>dst</i>	<i>src2</i>	M3	M2	M1	0010	S2	S1	<i>src1</i>
59:3	<b>subi</b>	1	0101 1001	<i>dst</i>	<i>src2</i>	M3	M2	M1	0011	S2	S1	<i>src1</i>
59:4	<b>cmpob</b>	1	0101 1001		<i>src2</i>	M3	M2	M1	0100	S2	S1	<i>src1</i>
59:5	<b>cmpib</b>	1	0101 1001		<i>src2</i>	M3	M2	M1	0101	S2	S1	<i>src1</i>
59:6	<b>cmpos</b>	1	0101 1001		<i>src2</i>	M3	M2	M1	0110	S2	S1	<i>src1</i>
59:7	<b>cmpis</b>	1	0101 1001		<i>src2</i>	M3	M2	M1	0111	S2	S1	<i>src1</i>
59:8	<b>shro</b>	1	0101 1001	<i>dst</i>	<i>src</i>	M3	M2	M1	1000	S2	S1	<i>len</i>
59:A	<b>shrdi</b>	3	0101 1001	<i>dst</i>	<i>src</i>	M3	M2	M1	1010	S2	S1	<i>len</i>
59:B	<b>shri</b>	1	0101 1001	<i>dst</i>	<i>src</i>	M3	M2	M1	1011	S2	S1	<i>len</i>
59:C	<b>shlo</b>	1	0101 1001	<i>dst</i>	<i>src</i>	M3	M2	M1	1100	S2	S1	<i>len</i>
59:D	<b>rotate</b>	1	0101 1001	<i>dst</i>	<i>src</i>	M3	M2	M1	1101	S2	S1	<i>len</i>
59:E	<b>shli</b>	1	0101 1001	<i>dst</i>	<i>src</i>	M3	M2	M1	1110	S2	S1	<i>len</i>
5A:0	<b>cmpo</b>	1	0101 1010		<i>src2</i>	M3	M2	M1	0000	S2	S1	<i>src1</i>
5A:1	<b>cmpi</b>	1	0101 1010		<i>src2</i>	M3	M2	M1	0001	S2	S1	<i>src1</i>

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	Mode			Opcode (3-0)	Special Flags	src1
			Opcode (11 - 4)	src/dst	src2			
5A:2	<b>concmpo</b>	31 ..... 24	23 ... 19	18 ... 14	13	12	11	10.....7 6 5 4.....0
5A:3	<b>concmpl</b>	0101 1010		src2	M3	M2	M1	0010 S2 S1 src1
5A:4	<b>cmpinco</b>	0101 1010		src2	M3	M2	M1	0011 S2 S1 src1
5A:5	<b>cmpincl</b>	0101 1010	dst	src2	M3	M2	M1	0100 S2 S1 src1
5A:6	<b>cmpdeco</b>	0101 1010	dst	src2	M3	M2	M1	0101 S2 S1 src1
5A:7	<b>cmpdeci</b>	0101 1010	dst	src2	M3	M2	M1	0110 S2 S1 src1
5A:C	<b>scanbyte</b>	0101 1010		src2	M3	M2	M1	0111 S2 S1 src1
5A:D	<b>bswap</b>	0101 1010	dst		M3	M2	M1	1100 S2 S1 src1
5A:E	<b>chkbit</b>	0101 1010		src	M3	M2	M1	1101 S2 S1 src1
5B:0	<b>addc</b>	0101 1011	dst	src2	M3	M2	M1	0000 S2 S1 src1
5B:2	<b>subc</b>	0101 1011	dst	src2	M3	M2	M1	0010 S2 S1 src1
5B:4	<b>intdis</b>	0101 1011			M3	M2	M1	0100 S2 S1
5B:5	<b>inten</b>	0101 1011			M3	M2	M1	0101 S2 S1
5C:C	<b>mov</b>	0101 1100	dst		M3	M2	M1	1100 S2 S1 src
5D:8	<b>eshro</b>	0101 1101	dst	src2	M3	M2	M1	1000 S2 S1 src1
5D:C	<b>movl</b>	0101 1101	dst		M3	M2	M1	1100 S2 S1 src
5E:C	<b>movt</b>	0101 1110	dst		M3	M2	M1	1100 S2 S1 src
5F:C	<b>movq</b>	0101 1111	dst		M3	M2	M1	1100 S2 S1 src
61:0	<b>atmod</b>	0110 0010	dst	src2	M3	M2	M1	0000 S2 S1 src1
61:2	<b>atadd</b>	0110 0010	dst	src2	M3	M2	M1	0010 S2 S1 src1
64:0	<b>spanbit</b>	0110 0100	dst		M3	M2	M1	0000 S2 S1 src
64:1	<b>scanbit</b>	0110 0100	dst		M3	M2	M1	0001 S2 S1 src
64:5	<b>modac</b>	0110 0100	mask	src	M3	M2	M1	0101 S2 S1 dst
65:0	<b>modify</b>	0110 0101	src/dst	src	M3	M2	M1	0000 S2 S1 mask
65:1	<b>extract</b>	0110 0101	src/dst	len	M3	M2	M1	0001 S2 S1 bitpos
65:4	<b>modtc</b>	0110 0101	mask	src	M3	M2	M1	0100 S2 S1 dst
65:5	<b>modpc</b>	0110 0101	src/dst	mask	M3	M2	M1	0101 S2 S1 src
65:8	<b>intctl</b>	0110 0101	dst		M3	M2	M1	1000 S2 S1 src1
65:9	<b>sysctl</b>	0110 0101	src/dst	src2	M3	M2	M1	1001 S2 S1 src1
65:B	<b>icctl</b>	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)**

<b>Opcode</b>	<b>Mnemonic</b>	<b>Cycles to Execute</b>	<b>Opcode (11 - 4)</b>	<b>src/dst</b>	<b>src2</b>	<b>Mode</b>			<b>Opcode (3-0)</b>	<b>Special Flags</b>		<b>src1</b>
		31 ..... 24	23 .... 19	18 ... 14	13	12	11	10 .... 7	6	5	4 .... 0	
65:C	<b>dcctl</b>	See Table B-7 <sup>1</sup>	0110 0101	src/dst	src2	M3	M2	M1	1100	S2	S1	src1
66:0	<b>calls</b>	22+spill	0110 0110			M3	M2	M1	0000	S2	S1	src
66:B	<b>mark</b>	8	0110 0110			M3	M2	M1	1011	S2	S1	
66:C	<b>fmark</b>	5	0110 0110			M3	M2	M1	1100	S2	S1	
66:D	<b>flushreg</b>	21 • # frames	0110 0110			M3	M2	M1	1101	S2	S1	
66:F	<b>syncf</b>	8	0110 0110			M3	M2	M1	1111	S2	S1	
67:0	<b>emul</b>	1	0110 0111	dst	src2	M3	M2	M1	0000	S2	S1	src1
67:1	<b>ediv</b>	3-36	0110 0111	dst	src2	M3	M2	M1	0001	S2	S1	src1
70:1	<b>mulo</b>	1	0111 0000	dst	src2	M3	M2	M1	0001	S2	S1	src1
70:8	<b>remo</b>	1	0111 0000	dst	src2	M3	M2	M1	1000	S2	S1	src1
70:B	<b>divo</b>	1	0111 0000	dst	src2	M3	M2	M1	1011	S2	S1	src1
74:1	<b>muli</b>	1	0111 0100	dst	src2	M3	M2	M1	0001	S2	S1	src1
74:8	<b>remi</b>	1	0111 0100	dst	src2	M3	M2	M1	1000	S2	S1	src1
74:9	<b>modi</b>	1	0111 0100	dst	src2	M3	M2	M1	1001	S2	S1	src1
74:B	<b>divi</b>	7-30	0111 0100	dst	src2	M3	M2	M1	1011	S2	S1	src1
78:0	<b>addono</b>	1	0111 1000	dst	src2	M3	M2	M1	0000	S2	S1	src1
78:1	<b>addino</b>	1	0111 1000	dst	src2	M3	M2	M1	0001	S2	S1	src1
78:2	<b>subono</b>	1	0111 1000	dst	src2	M3	M2	M1	0010	S2	S1	src1
78:3	<b>subino</b>	1	0111 1000	dst	src2	M3	M2	M1	0011	S2	S1	src1
78:4	<b>selno</b>	1	0111 1000	dst	src2	M3	M2	M1	0100	S2	S1	src1
79:0	<b>addog</b>	1	0111 1001	dst	src2	M3	M2	M1	0000	S2	S1	src1
79:1	<b>addig</b>	1	0111 1001	dst	src2	M3	M2	M1	0001	S2	S1	src1
79:2	<b>subog</b>	1	0111 1001	dst	src2	M3	M2	M1	0010	S2	S1	src1
79:3	<b>subig</b>	1	0111 1001	dst	src2	M3	M2	M1	0011	S2	S1	src1
79:4	<b>selg</b>	1	0111 1001	dst	src2	M3	M2	M1	0100	S2	S1	src1
7A:0	<b>addoe</b>	1	0111 1010	dst	src2	M3	M2	M1	0000	S2	S1	src1
7A:1	<b>addie</b>	1	0111 1010	dst	src2	M3	M2	M1	0001	S2	S1	src1
7A:2	<b>suboe</b>	1	0111 1010	dst	src2	M3	M2	M1	0010	S2	S1	src1
7A:3	<b>subie</b>	1	0111 1010	dst	src2	M3	M2	M1	0011	S2	S1	src1
7A:4	<b>sele</b>	1	0111 1010	dst	src2	M3	M2	M1	0100	S2	S1	src1
7B:0	<b>addoge</b>	1	0111 1011	dst	src2	M3	M2	M1	0000	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	Mode			Opcode (3-0)	Special Flags	src1
			Opcode (11 - 4)	src/dst	src2			
		31 ..... 24	23 ... 19	18 ... 14	13	12	11	10.....7 6 5 4.....0
7B:1	<b>addige</b>	1	0111 1011	dst	src2	M3	M2	M1 0001 S2 S1 src1
7B:2	<b>suboge</b>	1	0111 1011	dst	src2	M3	M2	M1 0010 S2 S1 src1
7B:3	<b>subige</b>	1	0111 1011	dst	src2	M3	M2	M1 0011 S2 S1 src1
7B:4	<b>selge</b>	1	0111 1011	dst	src2	M3	M2	M1 0100 S2 S1 src1
7C:0	<b>addol</b>	1	0111 1100	dst	src2	M3	M2	M1 0000 S2 S1 src1
7C:1	<b>addil</b>	1	0111 1100	dst	src2	M3	M2	M1 0001 S2 S1 src1
7C:2	<b>subol</b>	1	0111 1100	dst	src2	M3	M2	M1 0010 S2 S1 src1
7C:3	<b>subil</b>	1	0111 1100	dst	src2	M3	M2	M1 0011 S2 S1 src1
7C:4	<b>sell</b>	1	0111 1100	dst	src2	M3	M2	M1 0100 S2 S1 src1
7D:0	<b>addone</b>	1	0111 1101	dst	src2	M3	M2	M1 0000 S2 S1 src1
7D:1	<b>addine</b>	1	0111 1101	dst	src2	M3	M2	M1 0001 S2 S1 src1
7D:2	<b>subone</b>	1	0111 1101	dst	src2	M3	M2	M1 0010 S2 S1 src1
7D:3	<b>subine</b>	1	0111 1101	dst	src2	M3	M2	M1 0011 S2 S1 src1
7D:4	<b>selne</b>	1	0111 1101	dst	src2	M3	M2	M1 0100 S2 S1 src1
7E:0	<b>addole</b>	1	0111 1110	dst	src2	M3	M2	M1 0000 S2 S1 src1
7E:1	<b>addile</b>	1	0111 1110	dst	src2	M3	M2	M1 0001 S2 S1 src1
7E:2	<b>subole</b>	1	0111 1110	dst	src2	M3	M2	M1 0010 S2 S1 src1
7E:3	<b>subile</b>	1	0111 1110	dst	src2	M3	M2	M1 0011 S2 S1 src1
7E:4	<b>selle</b>	1	0111 1110	dst	src2	M3	M2	M1 0100 S2 S1 src1
7F:0	<b>addoo</b>	1	0111 1111	dst	src2	M3	M2	M1 0000 S2 S1 src1
7F:1	<b>addio</b>	1	0111 1111	dst	src2	M3	M2	M1 0001 S2 S1 src1
7F:2	<b>suboo</b>	1	0111 1111	dst	src2	M3	M2	M1 0010 S2 S1 src1
7F:3	<b>subio</b>	1	0111 1111	dst	src2	M3	M2	M1 0011 S2 S1 src1
7F:4	<b>sellio</b>	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
			31 ..... 24	23... 19	18 ... 14	13	12..... 2	1	0
20	testno	10-12	0010 0000	dst		M1		T	S2
21	testg	10-12	0010 0001	dst		M1		T	S2
22	teste	10-12	0010 0010	dst		M1		T	S2
23	testge	10-12	0010 0011	dst		M1		T	S2
24	testl	10-12	0010 0100	dst		M1		T	S2
25	testne	10-12	0010 0101	dst		M1		T	S2
26	testle	10-12	0010 0110	dst		M1		T	S2
27	testo	10-12	0010 0111	dst		M1		T	S2
30	bbc	1-3 <sup>1</sup>	0011 0000	bitpos	src	M1	targ	T	S2
31	cmpobg	1-3	0011 0001	src1	src2	M1	targ	T	S2
32	cmpobe	1-3	0011 0010	src1	src2	M1	targ	T	S2
33	cmpobge	1-3	0011 0011	src1	src2	M1	targ	T	S2
34	cmpobl	1-3	0011 0100	src1	src2	M1	targ	T	S2
35	cmpobne	1-3	0011 0101	src1	src2	M1	targ	T	S2
36	cmpoble	1-3	0011 0110	src1	src2	M1	targ	T	S2
37	bbs	1-3	0011 0111	bitpos	src	M1	targ	T	S2
38	cmpibno	1-3	0011 1000	src1	src2	M1	targ	T	S2
39	cmpibg	1-3	0011 1001	src1	src2	M1	targ	T	S2
3A	cmpibe	1-3	0011 1010	src1	src2	M1	targ	T	S2
3B	cmpibge	1-3	0011 1011	src1	src2	M1	targ	T	S2
3C	cmpibl	1-3	0011 1100	src1	src2	M1	targ	T	S2
3D	cmpibne	1-3	0011 1101	src1	src2	M1	targ	T	S2
3E	cmpible	1-3	0011 1110	src1	src2	M1	targ	T	S2
3F	cmpibo	1-3	0011 1111	src1	src2	M1	targ	T	S2

1. Indicates that it takes two cycles to execute the instruction plus an additional cycle to fetch the target instruction if the branch is taken.

**Table B-4. CTRL Format Instruction Encodings**

<b>Opcode</b>	<b>Mnemonic</b>	<b>Cycles to Execute</b>	<b>Opcode</b>	<b>Displacement</b>	<b>T</b>	<b>o</b>
		31.....24	23.....2	1	0	
08	<b>b</b>	0-2 <sup>1</sup>	0000 1000	<i>targ</i>	T	0
09	<b>call</b>	4+spill	0000 1001	<i>targ</i>	T	0
0A	<b>ret</b>	4+fill	0000 1010		T	0
0B	<b>bal</b>	1-2	0000 1011	<i>targ</i>	T	0
10	<b>bno</b>	0-2	0001 0000	<i>targ</i>	T	0
11	<b>bg</b>	0-2	0001 0001	<i>targ</i>	T	0
12	<b>be</b>	0-2	0001 0010	<i>targ</i>	T	0
13	<b>bge</b>	0-2	0001 0011	<i>targ</i>	T	0
14	<b>bl</b>	0-2	0001 0100	<i>targ</i>	T	0
15	<b>bne</b>	0-2	0001 0101	<i>targ</i>	T	0
16	<b>ble</b>	0-2	0001 0110	<i>targ</i>	T	0
17	<b>bo</b>	0-2	0001 0111	<i>targ</i>	T	0
18	<b>faultno</b>	7-12	0001 1000		T	0
19	<b>faultg</b>	7-12	0001 1001		T	0
1A	<b>faulte</b>	7-12	0001 1010		T	0
1B	<b>faultge</b>	7-12	0001 1011		T	0
1C	<b>faultl</b>	7-12	0001 1100		T	0
1D	<b>faultne</b>	7-12	0001 1101		T	0
1E	<b>faultle</b>	7-12	0001 1110		T	0
1F	<b>faulto</b>	7-12	0001 1111		T	0

1. Indicates that it takes 1 cycle to execute the instruction plus an additional cycle to fetch the target instruction if 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**

31.....24	23...19	18 .....14	13 ..... 12	11 .....	0
<b>Opcode</b>	<b>src/ dst</b>	<b>ABASE</b>	<b>Mode</b>	<b>Offset</b>	

31.....24	23...19	18 .....14	13 .....12 ..11..... 10	9..... 7	6...5	4 ..... 0
<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>						

**Effective Address**

<i>efa =</i>	offset	<b>Opcode</b>	<b>dst</b>		0	0				<i>offset</i>
--------------	--------	---------------	------------	--	---	---	--	--	--	---------------

offset( <i>reg</i> )	<b>Opcode</b>	<b>dst</b>		<b>reg</b>	1	0				<i>offset</i>
----------------------	---------------	------------	--	------------	---	---	--	--	--	---------------

( <i>reg</i> )	<b>Opcode</b>	<b>dst</b>		<b>reg</b>	0	1	0	0		00	
----------------	---------------	------------	--	------------	---	---	---	---	--	----	--

<i>disp + 8 (IP)</i>	<b>Opcode</b>	<b>dst</b>		0	1	0	1		00	
<i>displacement</i>										

( <i>reg1</i> )[ <i>reg2</i> * <i>scale</i> ]	<b>Opcode</b>	<b>dst</b>		<b>reg1</b>	0	1	1	1	<b>scale</b>	00	<b>reg2</b>
---	---------------	------------	--	-------------	---	---	---	---	--------------	----	-------------

<i>disp</i>	<b>Opcode</b>	<b>dst</b>		1	1	0	0		00	
<i>displacement</i>										

<i>disp(reg)</i>	<b>Opcode</b>	<b>dst</b>		<b>reg</b>	1	1	0	1		00	
<i>displacement</i>											

<i>disp[reg * scale]</i>	<b>Opcode</b>	<b>dst</b>		1	1	1	0	<b>scale</b>	00	<b>reg</b>
<i>displacement</i>										

<i>disp(reg1)[reg2*scale]</i>	<b>Opcode</b>	<b>dst</b>		<b>reg1</b>	1	1	1	1	<b>scale</b>	00	<b>reg2</b>
<i>displacement</i>											

<b>Opcode</b>	<b>Mnemonic</b>	<b>Cycles to Execute</b>	<b>Opcode</b>	<b>Mnemonic</b>	<b>Cycles to Execute</b>
80	<b>ldob</b>	1-4+bus (See Note 1)	9A	<b>stl</b>	1-4 (See Note 1)
82	<b>stob</b>	1-4 (See Note 1)	A0	<b>ldt</b>	1-4+bus (See Note 1)
84	<b>bx</b>	3-5	A2	<b>stt</b>	1-4 (See Note 1)
85	<b>balx</b>	3-5	AD	<b>dcinva</b>	1-4
86	<b>callx</b>	6-8+spill	B0	<b>ldq</b>	1-4+bus (See Note 1)
88	<b>ldos</b>	1-4+bus (See Note 1)	B2	<b>stq</b>	1-4 (See Note 1)
8A	<b>stos</b>	1-4 (See Note 1)	C0	<b>ldib</b>	1-4+bus (See Note 1)
8C	<b>lda</b>	1-3 (See Note 1)	C2	<b>stib</b>	1-4 (See Note 1)
90	<b>ld</b>	1-4+bus (See Note 1)	C8	<b>ldis</b>	1-4+bus (See Note 1)
92	<b>st</b>	1-4 (See Note 1)	CA	<b>stis</b>	1-4 (See Note 1)
98	<b>ldl</b>	1-4+bus (See Note 1)			

1. The number of cycles required to execute these instructions is based on the addressing mode used.

# Machine-Level Instruction Formats C

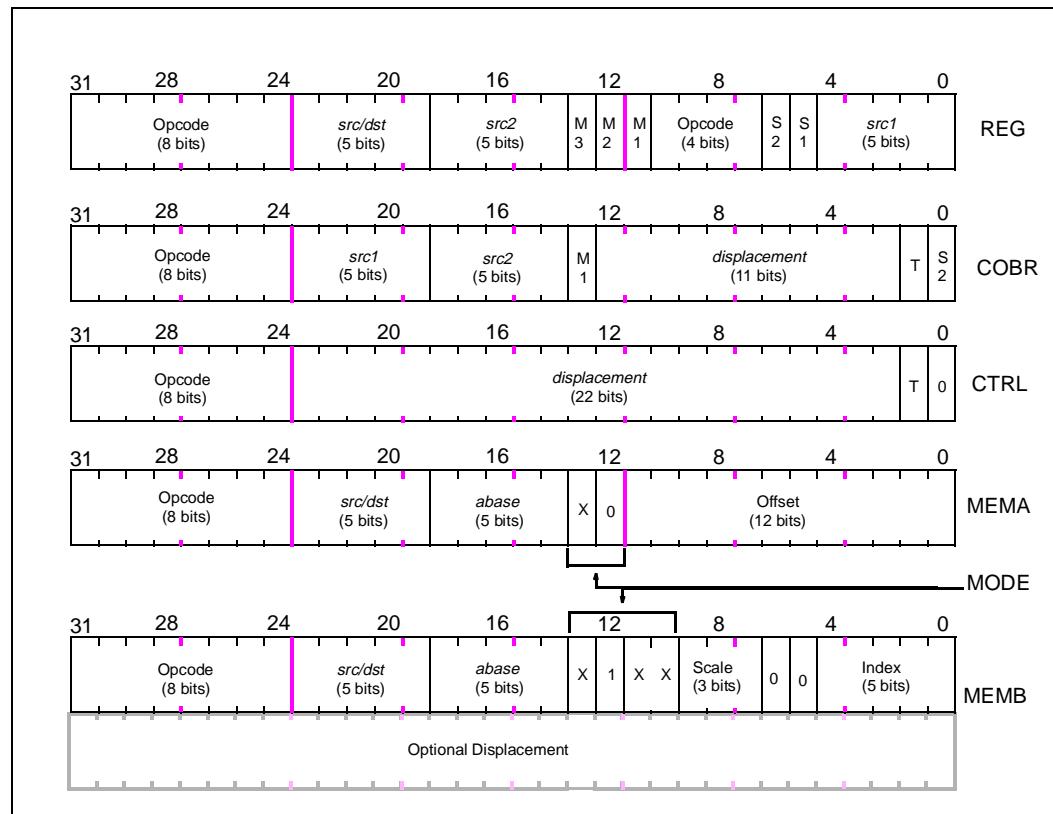
---

This appendix describes the encoding format for instructions used by the i960® processors. Included is a description of the four instruction formats and how the addressing modes relate to these formats. Refer also to [Appendix B, “Opcodes and Execution Times”](#).

## C.1 General Instruction Format

The i960 processor architecture defines four basic instruction encoding formats: REG, COBR, CTRL and MEM (see [Figure C-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 C-1. Instruction Formats**



**Table C-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 6-5</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 register's value is used in computing a memory address.
INDEX	A register whose register's 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).
T	The branch prediction bit. If this bit is 0 in a conditional instruction, the condition is likely to be true. If this bit is 1 in a conditional instruction, the condition is likely to be false. Conditional instructions affected are conditional branch instructions, compare and branch instructions, conditional fault instructions, and conditional test instructions.
S1, S2	These fields further define the meaning of the <i>src1</i> and <i>src2</i> fields respectively as shown in <a href="#">Table C-2</a> .
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 C-2</a> and <a href="#">Table C-3</a> .

When a particular instruction is defined as not using a particular field, the field is ignored.

## C.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, local or special function registers or literals. Mode bits (M1 for *src1* and M2 for *src2*), special-purpose bits (S1 for *src1* and S2 for *src2*) and the instruction type determine what an operand specifies. Table C-2 shows this relationship.

**Table C-2. Encoding of *src1* and *src2* in REG Format**

M1 or M2	S1 or S2	<i>src1</i> or <i>src2</i> Operand Value	Meaning
0	0	00000 ... 01111	r0 ... r15
		10000 ... 11111	g0 ... g15
1	0	00000 ... 11111	literal 0 ... 31
0	1	00000 ... 00100	sf0 ... sf4
		00101 ... 11111	reserved
1	1	00000 ... 11111	reserved

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 C-3. If M3 is set, the *src/dst* operand can be used as a source-only operand that is: (1) a literal, or (2) a destination-only operand that is a special function register.

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 C-3. Encoding of *src/dst* in REG Format**

M3	<i>src/dst</i>	<i>src Only</i>	<i>dst Only</i>
0	g0 ... g15 r0 ... r15	g0 ... g15 r0 ... r15	g0 ... g15 r0 ... r15
1	Reserved	Reserved	sf0 .. sf4

## C.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 specify a global, local or special function register as determined by special purpose bit S2. [Table C-4](#) shows the M1, *src1* relationship and [Table C-4](#) shows the S2, *src2* relationship.

**Table C-4. Encoding of *src1* in COBR Format**

M1	src1
0	g0 ... g15 r0 ... r15
1	SF0-SF4

**Table C-5. Encoding of *src2* in COBR Format**

S2	src2
0	g0 ... g15 r0 ... r15
1	sf0 ... sf4

The T bit supports the 80960Hx processor's branch prediction for conditional instructions. If T is cleared, the condition being tested is likely to be true; if set to 1, the condition is likely to be false.

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.

## C.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.

The T bit performs the same branch prediction function for CTRL instructions as it does for COBR instructions. If this bit is 0 in a conditional instruction, the condition is likely to be true.

## C.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 **Ida** 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 C-6](#) summarizes the addressing modes for the two MEM-format encodings. Fields used in these addressing modes are described in the following sections.

**Table C-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.

## C.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 **Ida** 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.

## C.5.2 MEMB Format Addressing

The MEMB format provides the following seven addressing modes:

- |   |   |
|---|---|
| <ul style="list-style-type: none"> <li>• absolute displacement</li> <li>• register indirect with displacement</li> <li>• register indirect with index and displacement</li> <li>• IP with displacement</li> </ul> | <ul style="list-style-type: none"> <li>• register indirect</li> <li>• register indirect with displacement</li> <li>• index with displacement</li> </ul> |
|---|---|

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 C-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 C-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.



# Register and Data Structures

D

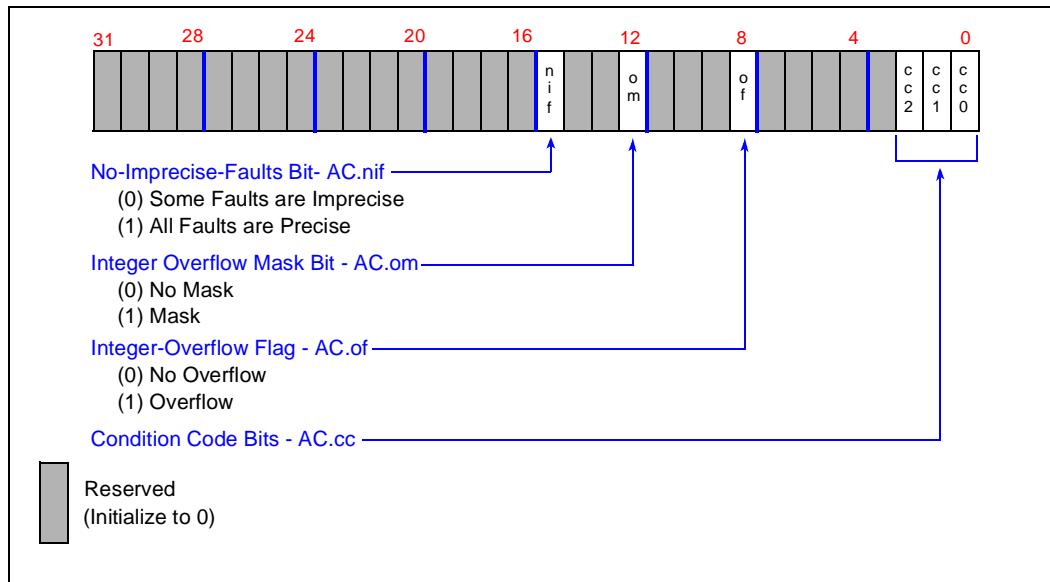
This appendix is a compilation of all register and data structure figures described throughout the manual. Following each figure is a reference that indicates the section that discusses the figure.

Fig.	Register / Data Structure	Where Defined in the manual	Page
<b>Registers</b>			
D-1	AC (Arithmetic Controls) Register	Section 3.2.6, "Register and Literal Addressing and Alignment" on page 3-5	D-3
D-2	BCON (Bus Control) Register	Section 14.2.1, "Bus Control (BCON) Register" on page 14-9	D-3
D-3	BPCON (Breakpoint Control) Register	Section 9.2.7.4, "Breakpoint Control Register" on page 9-7	D-4
D-4	CCON (Cache Control Register)	Section 4.5.1, "Enabling and Disabling the Data Cache" on page 4-7	D-4
D-5	DAB (Data Address Breakpoint) Register Format	Section 9.2.7.5, "Data Address Breakpoint (DAB) Registers" on page 9-8	D-4
D-6	DLMCON (Default Logical Memory Configuration) Register	Section 14.4, "Programming the Logical Memory Attributes" on page 14-11	D-5
D-7	GCON (GMU Control) Register	Section 12.3.1, "GMU Control Register" on page 12-5	D-6
D-8	IEEE 1149.1 Device Identification Register	Section 13.4, "Device Identification on Reset" on page 13-23	D-6
D-9	ICON (Interrupt Control) Register	Section 11.7.4, "Interrupt Control Register (ICON) — SF3" on page 11-20	D-7
D-10	IMAP0–IMAP2 (Interrupt Mapping) Registers	Section 11.7.5, "Interrupt Mapping Registers (IMAP0–IMAP2)" on page 11-21	D-8
D-11	IMSK (Interrupt Mask) Registers	Section 11.7.5.1, "Interrupt Mask (IMSK; SF1) and Interrupt Pending (IPND; SF0) Registers" on page 11-23	D-9
D-12	IPB (Instruction Breakpoint) Register Format	Section 9.2.7.6, "Instruction Breakpoint (IPB) Registers" on page 9-9	D-10
D-13	IPND (Interrupt Pending) Register	Section 11.7.5.1, "Interrupt Mask (IMSK; SF1) and Interrupt Pending (IPND; SF0) Registers" on page 11-23	D-11
D-14	LMAR0–14 (Logical Memory Address) Registers	Section 14.4, "Programming the Logical Memory Attributes" on page 14-11	D-12
D-15	LMMR0–14 (Logical Memory Mask Registers)	Section 14.4, "Programming the Logical Memory Attributes" on page 14-11	D-13
D-16	MDUB0–5, MDLB0–5 (GMU Memory Violation Detection Upper and Lower-Bounds) Registers	Section 12.3.3, "GMU Memory Detect Upper- and Lower-Bounds Registers" on page 12-10	D-14
D-17	MPAR0–1, MPMR0–1 (GMU Memory Protect Address Register and Memory Protect Mask Register)	Section 12.3.2, "GMU Memory Protect Address and Mask Registers" on page 12-6	D-15
D-18	PC (Process Controls) Register	Section 3.6.3, "Process Controls (PC) Register" on page 3-24	D-16
D-19	PFP (Previous Frame Pointer) Register (r0)	Section 7.1.2.5, "Previous Frame Pointer" on page 7-5	D-16
D-20	PMCON0–15 (Physical Memory Configuration) Register	Section 14.2, "Programming the Physical Memory Configuration (PMCON) Registers" on page 14-7	D-17

<b>Fig.</b>	<b>Register / Data Structure</b>	<b>Where Defined in the manual</b>	<b>Page</b>
D-21	PMCON15 (Physical Memory Configuration) Register Bit Description in IBR	Section 13.3.1.1, "Initialization Boot Record (IBR)" on page 13-13	D-18
D-22	TC (Trace Controls) Register	Section 9.1.1, "Trace Controls (TC) Register" on page 9-2	D-19
D-23	TCR0-1 (Timer Count Register)	Section 10.1.2, "Timer Count Register (TCR0, TCR1)" on page 10-5	D-19
D-24	TMR0-1 (Timer Mode Register)	Section 10.1.1, "Timer Mode Registers (TMR0, TMR1)" on page 10-2	D-20
D-25	TRR0-1 (Timer Reload Register)	Section 10.1.3, "Timer Reload Register (TRR0, TRR1)" on page 10-6	D-20
D-26	XBPCON (Extended Breakpoint Control) Register	Section 9.2.7.4, "Breakpoint Control Register" on page 9-7	D-21
<b>Data Structures</b>			
D-27	Control Table	Section 13.3.3, "Control Table" on page 13-22	D-22
D-28	Fault Table and Fault Table Entries	Section 8.3, "Fault Table" on page 8-5	D-23
D-29	Fault Record	Section 8.5, "Fault Record" on page 8-6	D-24
D-30	Initial Memory Image (IMI) and Process Control Block (PRCB)	Section 13.3.1, "Initial Memory Image (IMI)" on page 13-10	D-25
D-31	Interrupt Table	Section 11.4, "Interrupt Table" on page 11-4	D-26
D-32	Procedure Stack Structure and Local Registers	Section 7.1.1, "Local Registers and the Procedure Stack" on page 7-2	D-27
D-33	Process Control Block Configuration Words	Section 13.3.1.2, "Process Control Block (PRCB)" on page 13-17	D-28
D-34	Storage of an Interrupt Record on the Interrupt Stack	Section 11.5, "Interrupt Stack and Interrupt Record" on page 11-6	D-29
D-35	System Procedure Table	Section 7.5.1, "System Procedure Table" on page 7-15	D-30

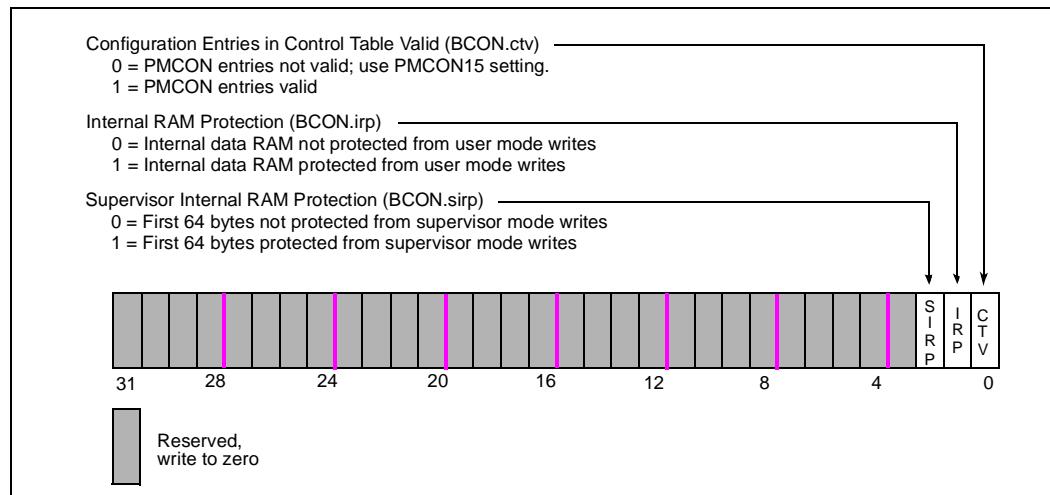
## D.1 Registers

**Figure D-1. AC (Arithmetic Controls) Register**

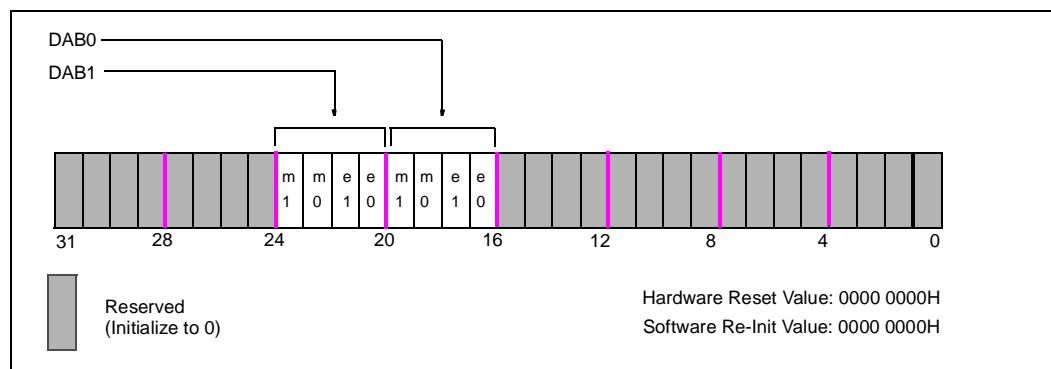


Section 3.6.2, "Arithmetic Controls (AC) Register" on page 3-21

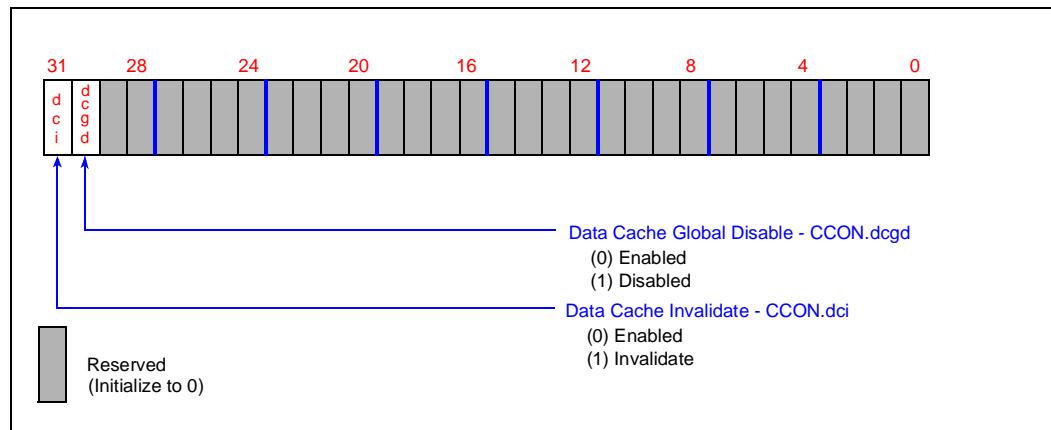
**Figure D-2. BCON (Bus Control) Register**



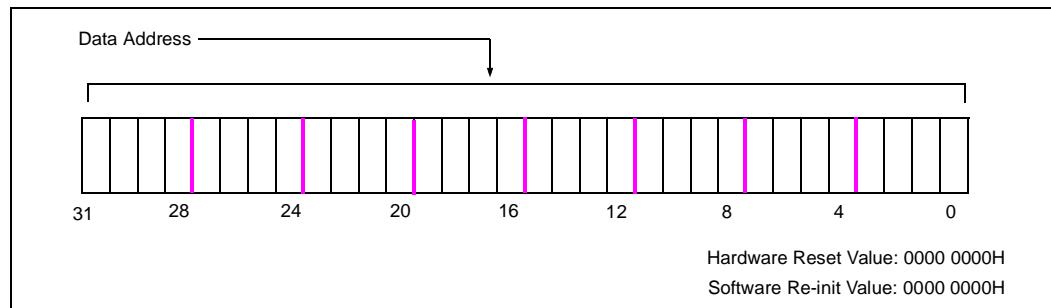
Section 14.2.1, "Bus Control (BCON) Register" on page 14-9

**Figure D-3. BPCON (Breakpoint Control) Register**

[Section 9.2.7.4, “Breakpoint Control Register” on page 9-7](#)

**Figure D-4. CCON (Cache Control Register)**

[Section 4.5.1, “Enabling and Disabling the Data Cache” on page 4-7](#)

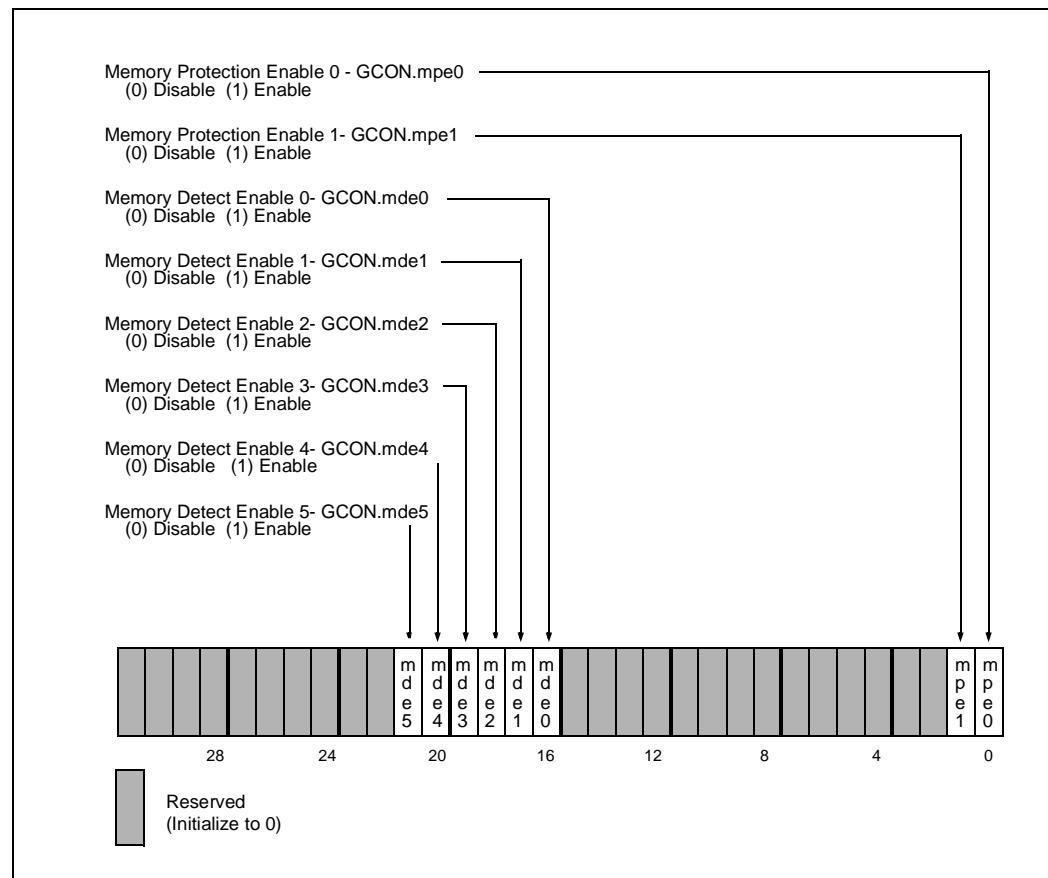
**Figure D-5. DAB (Data Address Breakpoint) Register Format**

[Section 9.2.7.5, “Data Address Breakpoint \(DAB\) Registers” on page 9-8](#)

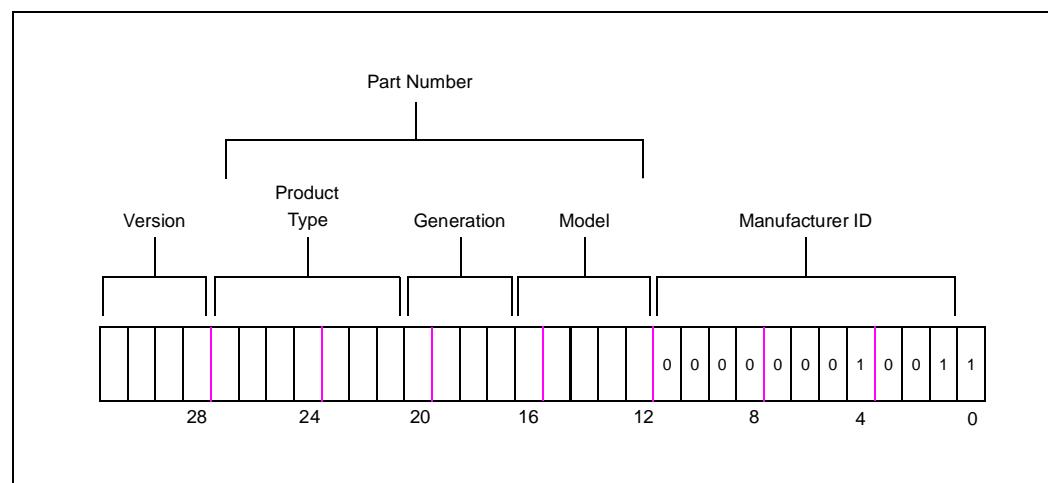
**Figure D-6. DLMCON (Default Logical Memory Configuration) Register**

Mnemonic	Bit/Bit Field Name	Bit Position(s)	Function
BE	Big Endian Byte Order	0	Controls byte order for areas not within other logical memory templates. 0 = Little-endian 1 = Big-endian
DCEN	Data Cache Enable	1	Controls data caching policy for areas not within other logical memory templates. 0 = Data caching disabled 1 = Write-through data caching enabled
DCIIR	Data Cache Independently Invalidatable Region	4	Controls Logical Region Invalidation for areas not within other Logical Memory Templates. 0 = Region is not independently invalidatable 1 = Region is independently invalidatable using the <b>dctl</b> "Invalidate Logical Region" command.

Section 14.4, "Programming the Logical Memory Attributes" on page 14-11

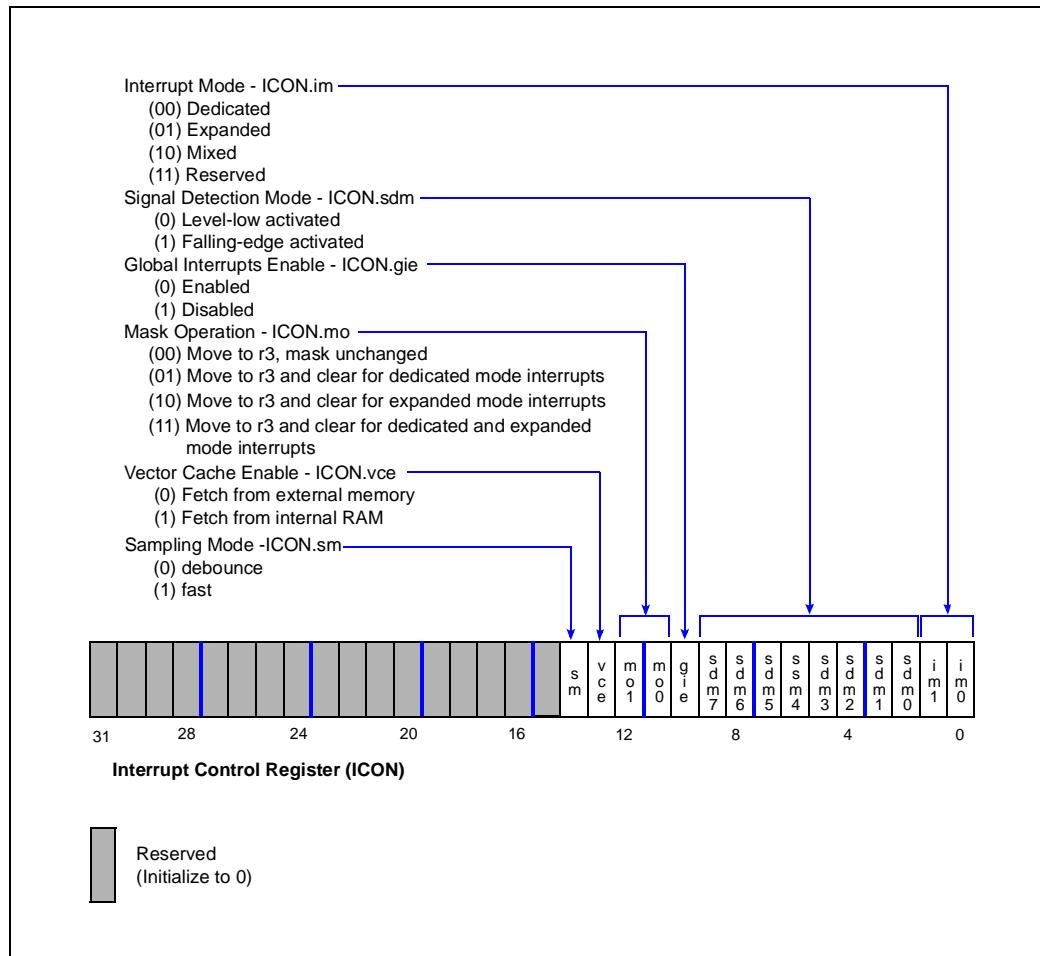
**Figure D-7. GCON (GMU Control) Register**

[Section 12.3.1, "GMU Control Register" on page 12-5](#)

**Figure D-8. IEEE 1149.1 Device Identification Register**

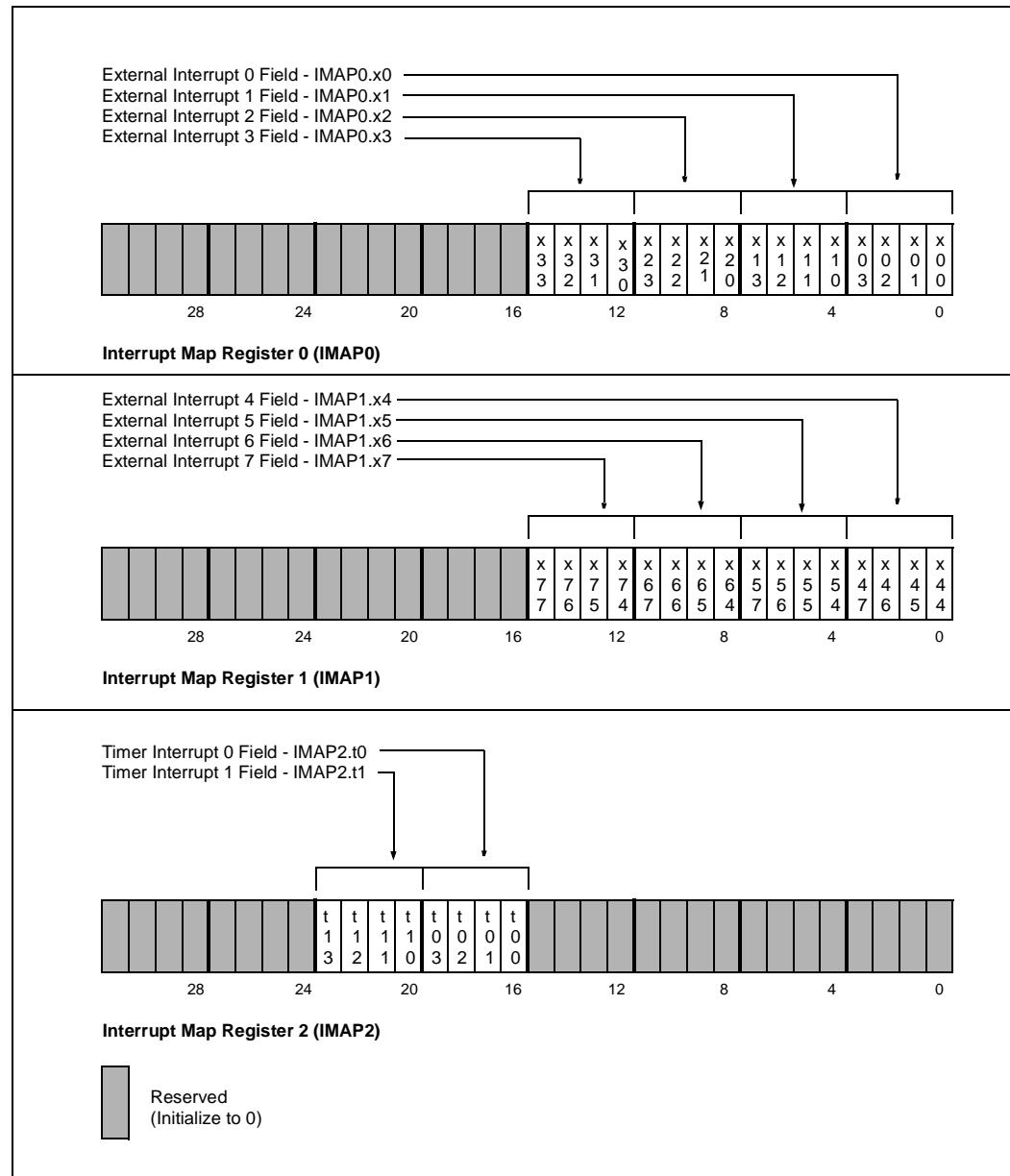
[Section 13.4, "Device Identification on Reset" on page 13-23](#)

**Figure D-9. ICON (Interrupt Control) Register**

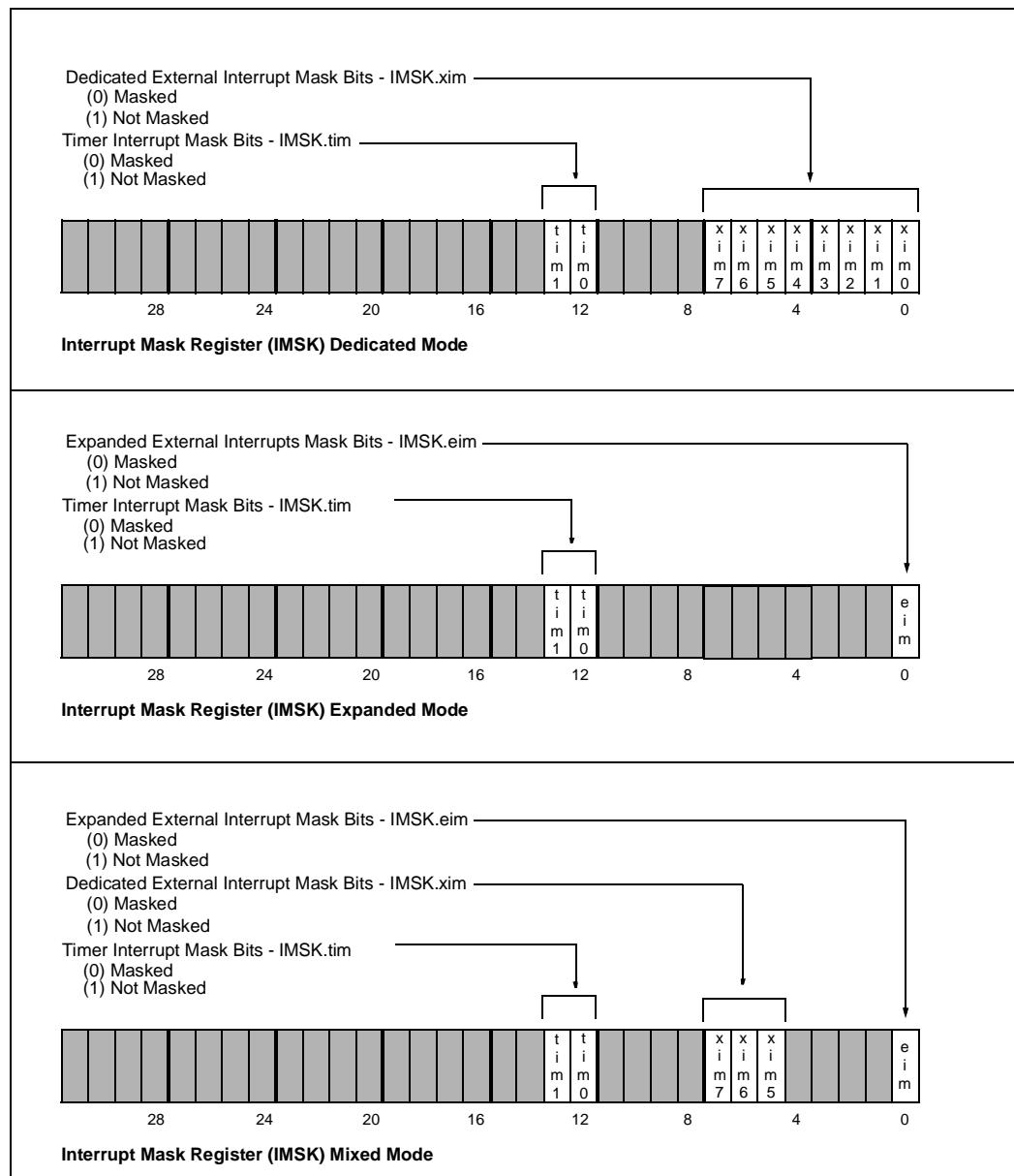


Section 11.7.4, “Interrupt Control Register (ICON) — SF3” on page 11-20

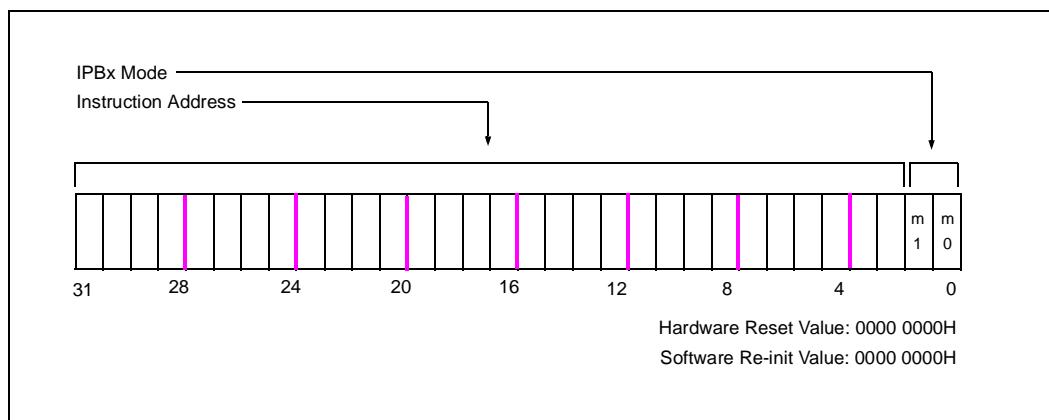
**Figure D-10. IMAP0–IMAP2 (Interrupt Mapping) Registers**



[Section 11.7.5, “Interrupt Mapping Registers \(IMAP0-IMAP2\)” on page 11-21](#)

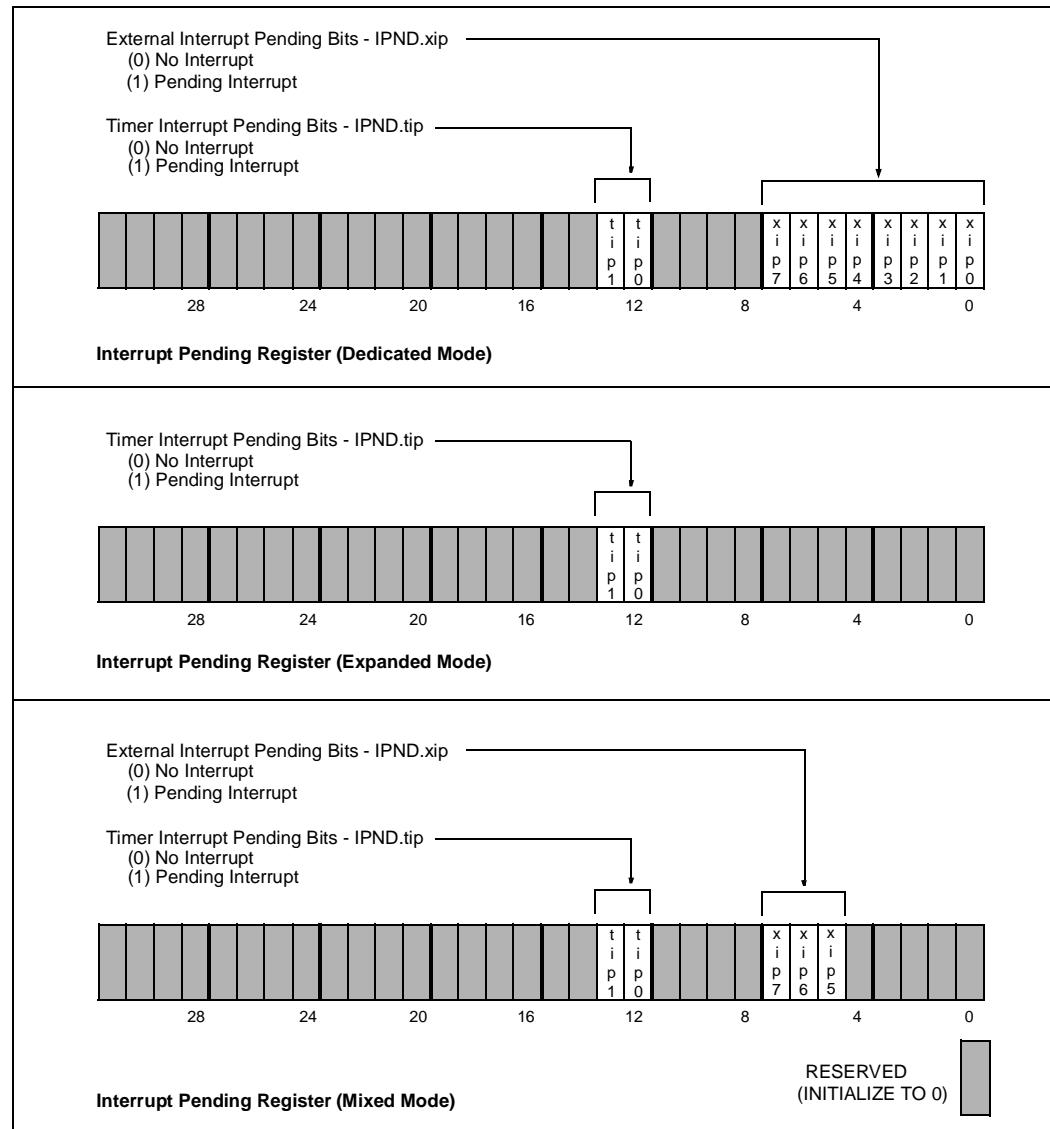
**Figure D-11. IMSK (Interrupt Mask) Registers**


Section 11.7.5.1, “Interrupt Mask (IMSK; SF1) and Interrupt Pending (IPND; SF0) Registers” on page 11-23

**Figure D-12. IPB (Instruction Breakpoint) Register Format**

Section 9.2.7.6, "Instruction Breakpoint (IPB) Registers" on page 9-9

**Figure D-13. IPND (Interrupt Pending) Register**

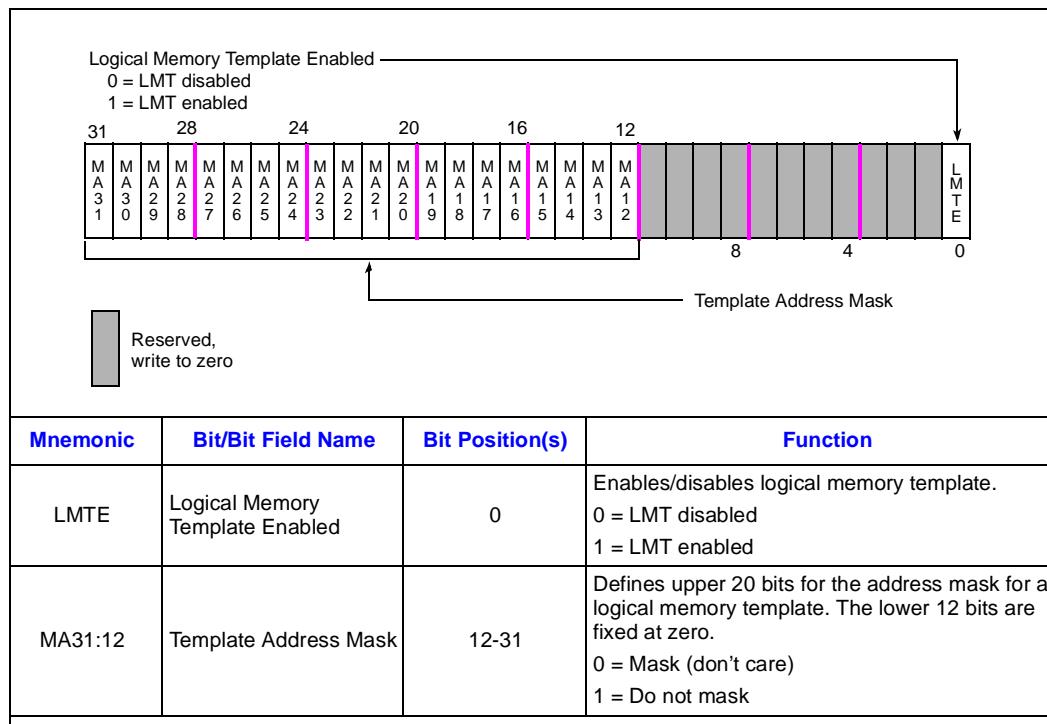


Section 11.7.5.1, "Interrupt Mask (IMSK; SF1) and Interrupt Pending (IPND; SF0) Registers" on page 11-23

**Figure D-14. LMAR0–14 (Logical Memory Address) Registers**

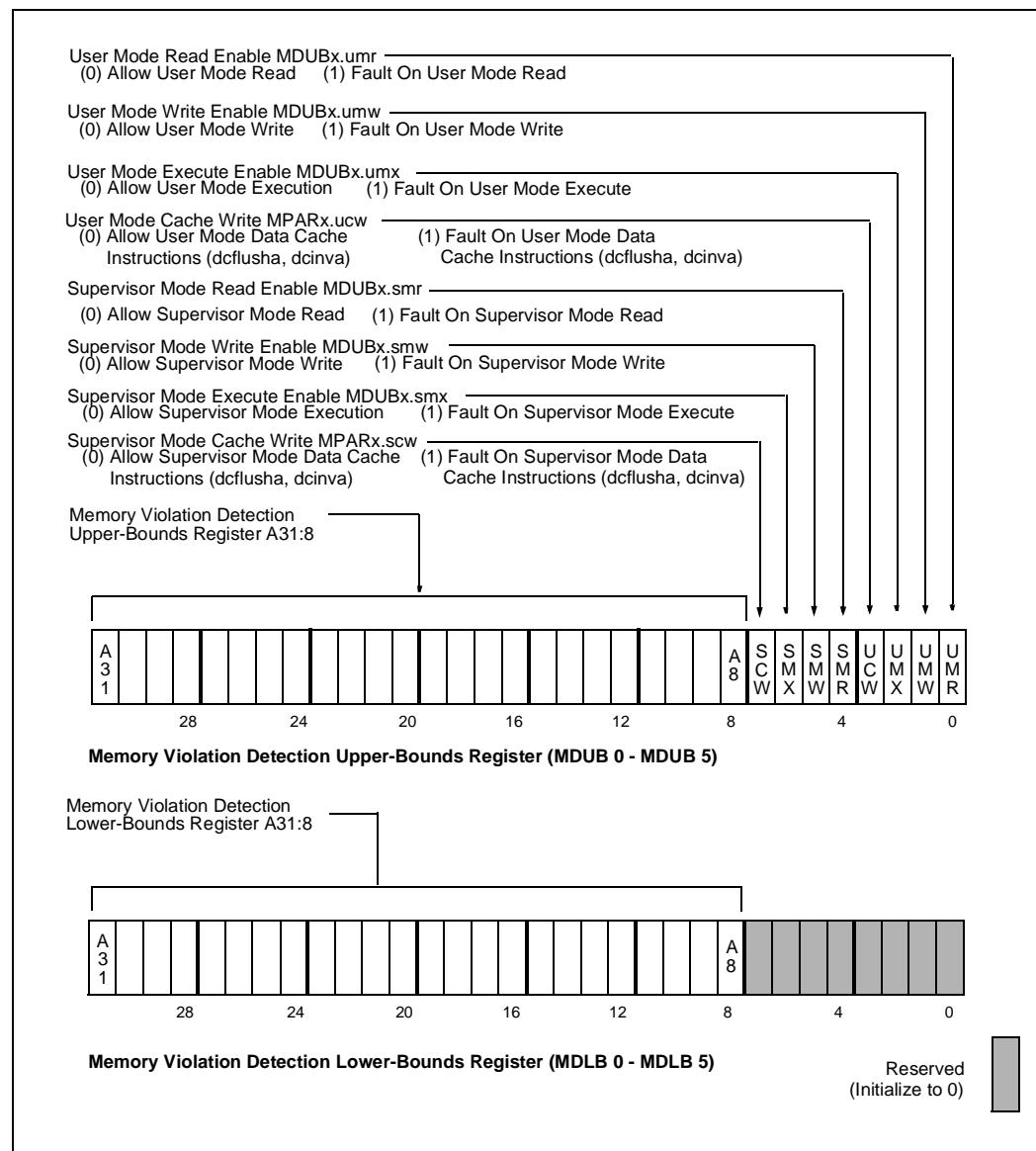
Mnemonic	Bit/Bit Field Name	Bit Position(s)	Function
BE	Big Endian Byte Order	0	Controls byte order for the template. 0 = Little-endian 1 = Big-endian
DCEN	Data Cache Enable	1	Controls data caching template. 0 = Data caching disabled 1 = Write-through caching enabled
DCIIR	Data Cache Independently Invalidatable Region	4	Controls Logical Region Invalidiation. 0 = region is not independently invalidatable 1 = region is independently invalidatable using the <b>dcctl</b> "Invalidate Logical Region" command.
A31:12	Template Starting Address	12-31	Defines upper 20 bits for the starting address for a logical data template. The lower 12 bits are fixed at zero. The starting address is modulo 4 Kbytes.

Section 14.4, “Programming the Logical Memory Attributes” on page 14-11

**Figure D-15. LMMR0–14 (Logical Memory Mask Registers)**


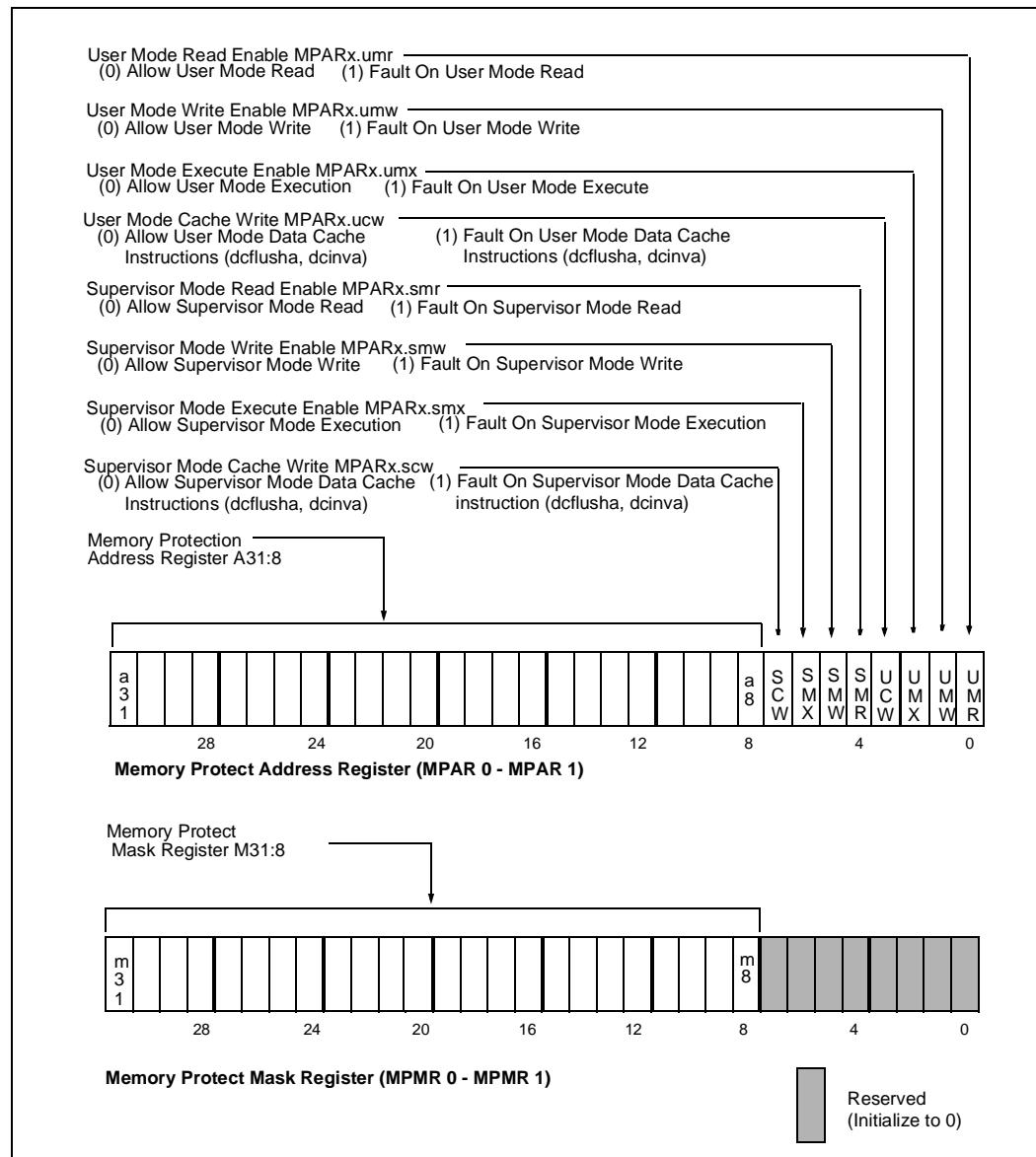
Section 14.4, “Programming the Logical Memory Attributes” on page 14-11

**Figure D-16. MDUB0–5, MDLB0–5 (GMU Memory Violation Detection Upper and Lower-Bounds) Registers**

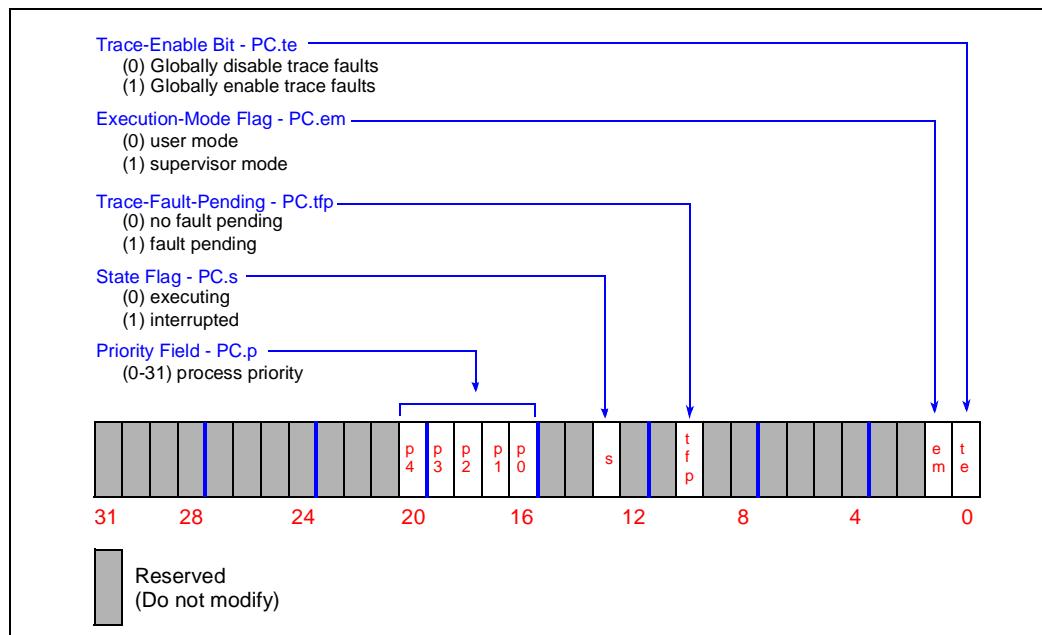


Section 12.3.3, “GMU Memory Detect Upper- and Lower-Bounds Registers” on page 12-10

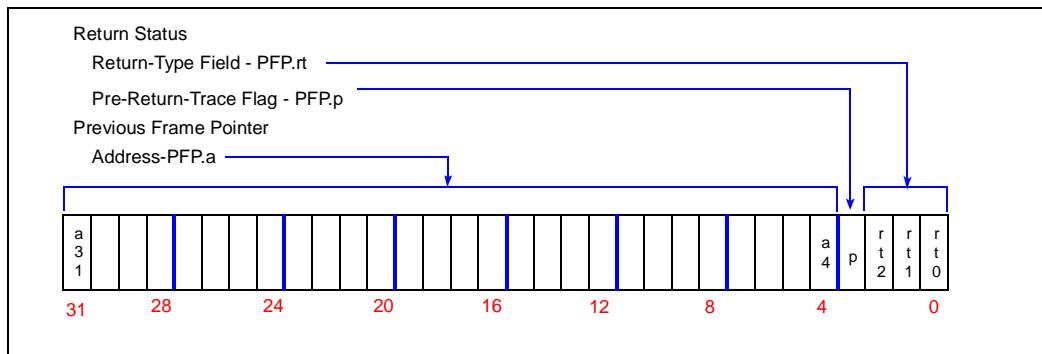
**Figure D-17. MPAR0–1, MPMR0–1 (GMU Memory Protect Address Register and Memory Protect Mask Register)**



Section 12.3.2, "GMU Memory Protect Address and Mask Registers" on page 12-6

**Figure D-18. PC (Process Controls) Register**

Section 3.6.3, "Process Controls (PC) Register" on page 3-24

**Figure D-19. PFP (Previous Frame Pointer) Register (r0)**

Section 7.1.2.5, "Previous Frame Pointer" on page 7-5

**Figure D-20. PMCON0–15 (Physical Memory Configuration) Register**

Mnemonic	Name	Bit #	Function
NRAD4:0	Number of Read Address to Data Wait States	0-4	Specifies the number of wait states (0 to 31) inserted between the assertion of ADS# and the first data cycle for read accesses.
NRDD1:0	Number of Read Data to Data Wait States	6-7	Specifies the number of wait states (0 to 3) inserted between successive data cycles for a burst read access.
NWAD4:0	Number of Write Address to Data Wait States	8-12	Specifies the number of wait states (0 to 31) inserted between the assertion of ADS# and the first data cycle for write accesses.
NWDD1:0	Number of Write Data to Data Wait States	14-15	Specifies the number of wait states (0 to 3) inserted between successive data cycles for a burst write access.
NXDA3:0	Number of Bus Turnaround Wait States	16-19	Specifies the number of wait states (0-15) inserted after the last data cycle for accesses in this region.
PEN	Parity Enable	20	Enables parity generation/checking for a region. 0 = not enabled, 1 = enabled
PODD	Parity Odd	21	Selects parity sense for a region. 0 = even, 1 = odd
BW1:0	Bus Width	22-23	Selects the bus width for a region: 00 = 8-bit, 01 = 16-bit, 10 = 32-bit bus 11 = reserved (do not use)
RPIPE	Read Pipelining Enable	24	Enables address pipelining for read accesses in this region 0 = disabled, 1 = enabled
BEN	Burst Enable	28	Enables burst accesses for the region. 0 = disabled, 1 = enabled
RBEN	READY#/BTERM# Enable	29	Enables the READY# and BTERM# pins for a region. 0 = READY#/BTERM# ignored in this region 1 = READY#/BTERM# enabled

Section 14.1.1, "Physical Memory Attributes" on page 14-1

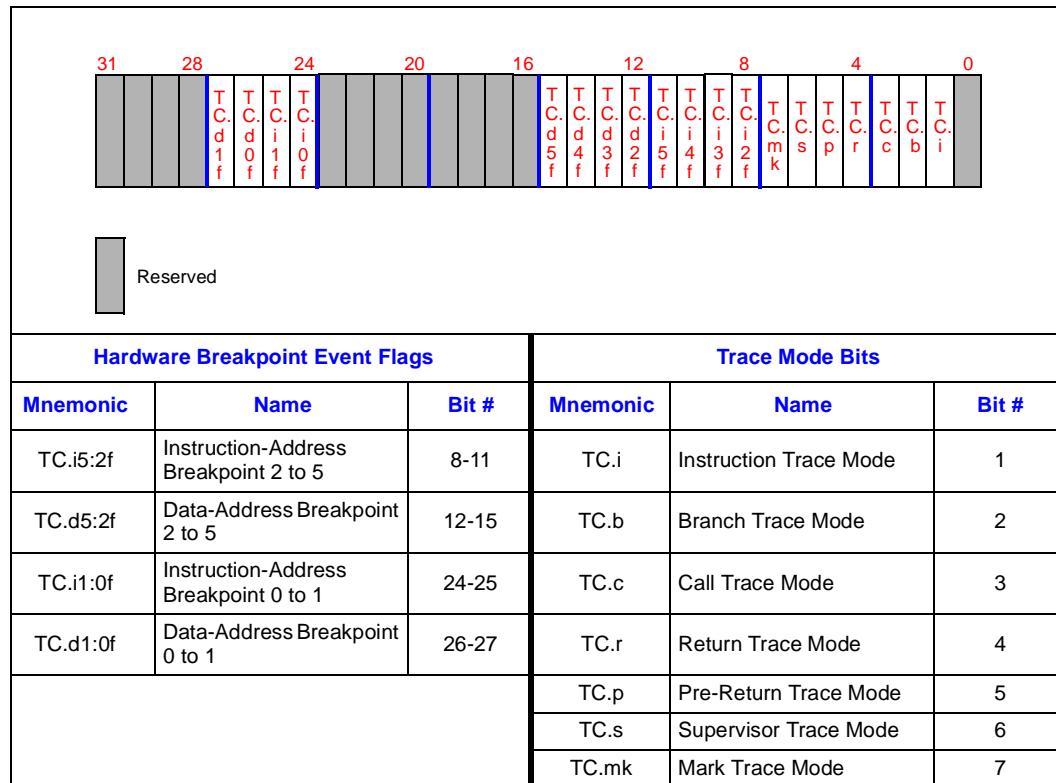
**Figure D-21. PMCON15 (Physical Memory Configuration) Register Bit Description in IBR**

31	28	24	20	16	12	8	4	0				
B B I G E	R B E N	R P I W P E	B W 0	P O D N 3 2	X X D A 3 2	X X D A 1 0	N W D A 0 4	N W D A 1 0	N R D A 0 4	N R A D 2 3	N R A D 1 0	N R A D 0 0

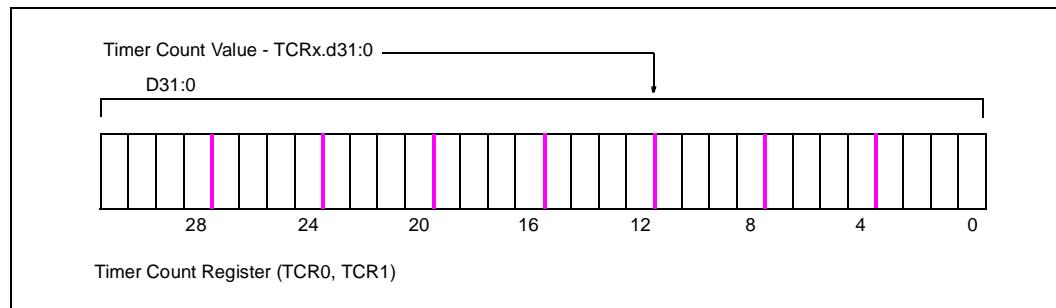
  

Mnemonic	Name	Bit #	Function
NRAD4:0	Number of Read Address to Data Wait States	0-4	Specifies the number of wait states (0 to 31) inserted between the assertion of ADS# and the first data cycle for read accesses.
NRDD1:0	Number of Read Data to Data Wait States	6-7	Specifies the number of wait states (0 to 3) inserted between successive data cycles for a burst read access.
NWAD4:0	Number of Write Address to Data Wait States	8-12	Specifies the number of wait states (0 to 31) inserted between the assertion of ADS# and the first data cycle for write accesses.
NWDD1:0	Number of Write Data to Data Wait States	14-15	Specifies the number of wait states (0 to 3) inserted between successive data cycles for a burst write access.
NXDA3:0	Number of Bus Turnaround Wait States	16-19	Specifies the number of wait states (0-15) inserted after the last data cycle for accesses in this region.
PEN	Parity Enable	20	Enables parity generation/checking for a region. 0 = not enabled, 1 = enabled
PODD	Parity Odd	21	Selects parity sense for a region. 0 = even, 1 = odd
BW1:0	Bus Width	22-23	Selects the bus width for a region: 00 = 8-bit, 01 = 16-bit, 10 = 32-bit bus 11 = reserved (do not use)
RPIPE	Read Pipelining Enable	24	Enables address pipelining for read accesses in this region. 0 = disabled, 1 = enabled
BEN	Burst Enable	28	Enables burst accesses for the region. 0 = disabled, 1 = enabled
RBEN	READY#/BTERM# Enable	29	Enables the READY# and BTERM# pins for a region. 0 = READY#/BTERM# ignored in this region 1 = READY#/BTERM# enabled
BBIGE	Boot Big-Endian Byte Order	31	0 = little-endian, 1 = big-endian This bit is copied to the DLMCON.be bit. See <a href="#">Section 14.4 “Programming the Logical Memory Attributes” on page 14-11</a>

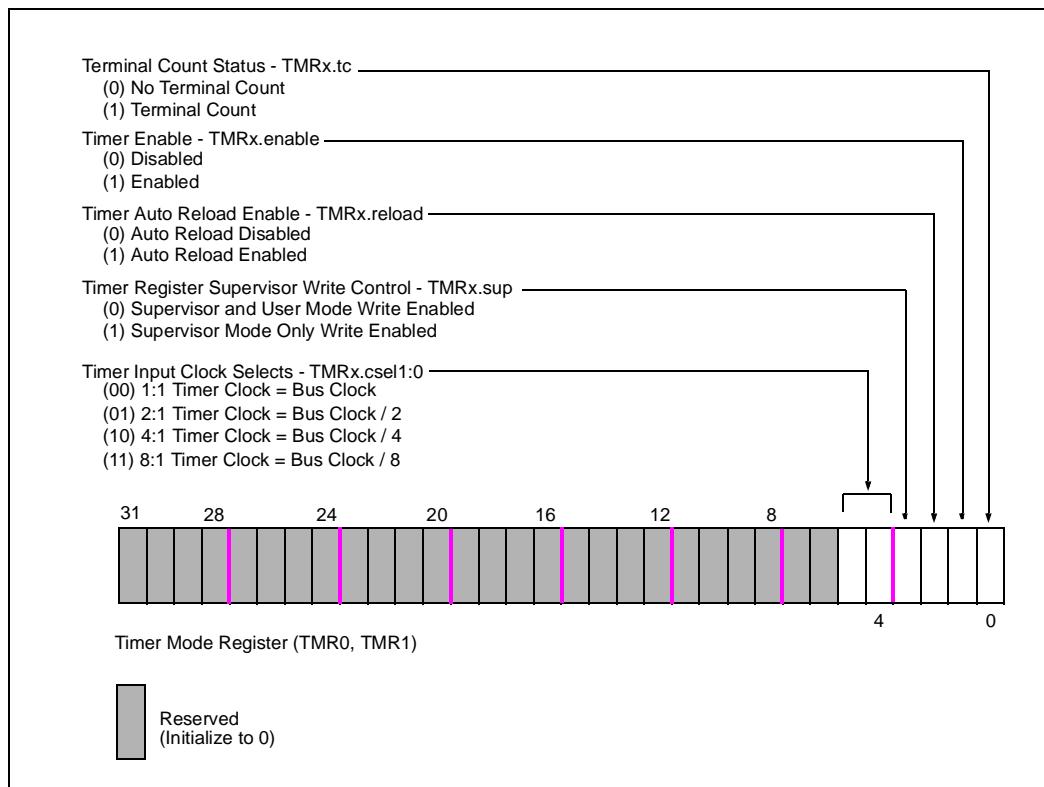
[Section 13.3.1.1, “Initialization Boot Record \(IBR\)” on page 13-13](#)

**Figure D-22. TC (Trace Controls) Register**


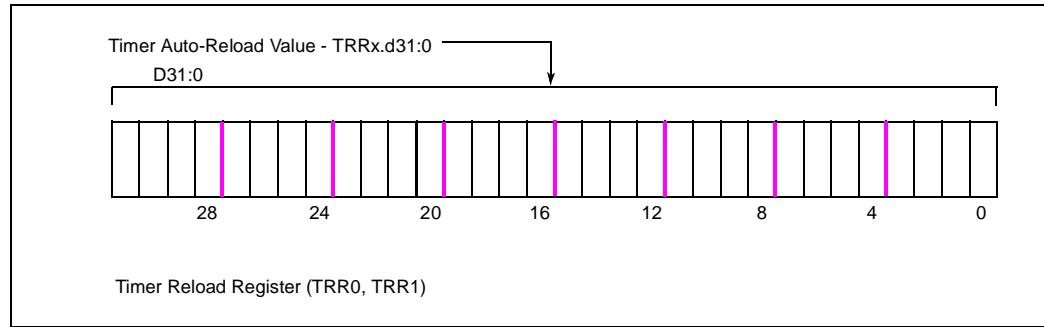
Section 9.1.1, "Trace Controls (TC) Register" on page 9-2

**Figure D-23. TCR0-1 (Timer Count Register)**


Section 10.1.2, "Timer Count Register (TCR0, TCR1)" on page 10-5

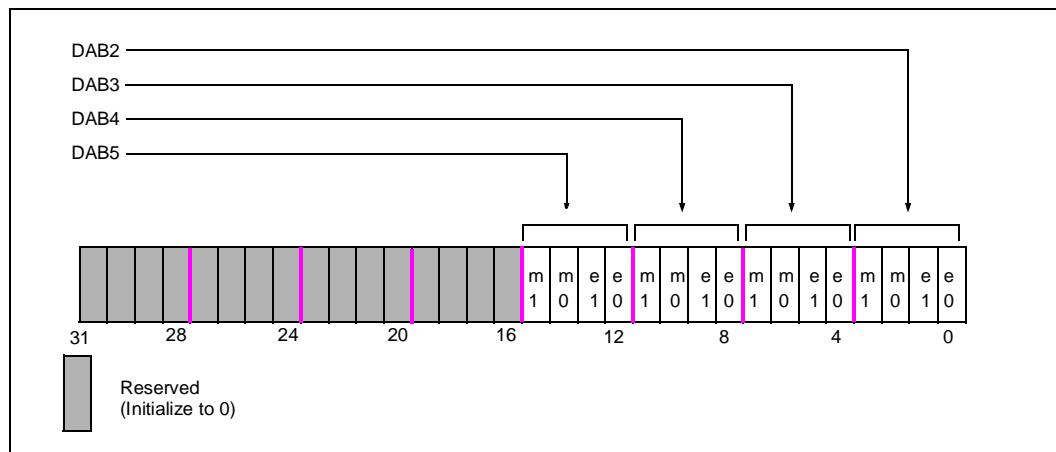
**Figure D-24. TMR0–1 (Timer Mode Register)**

Section 10.1.1, "Timer Mode Registers (TMR0, TMR1)" on page 10-2

**Figure D-25. TRR0-1 (Timer Reload Register)**

Section 10.1.3, "Timer Reload Register (TRR0, TRR1)" on page 10-6

**Figure D-26. XBPCon (Extended Breakpoint Control) Register**



Section 9.2.7.4, “Breakpoint Control Register” on page 9-7

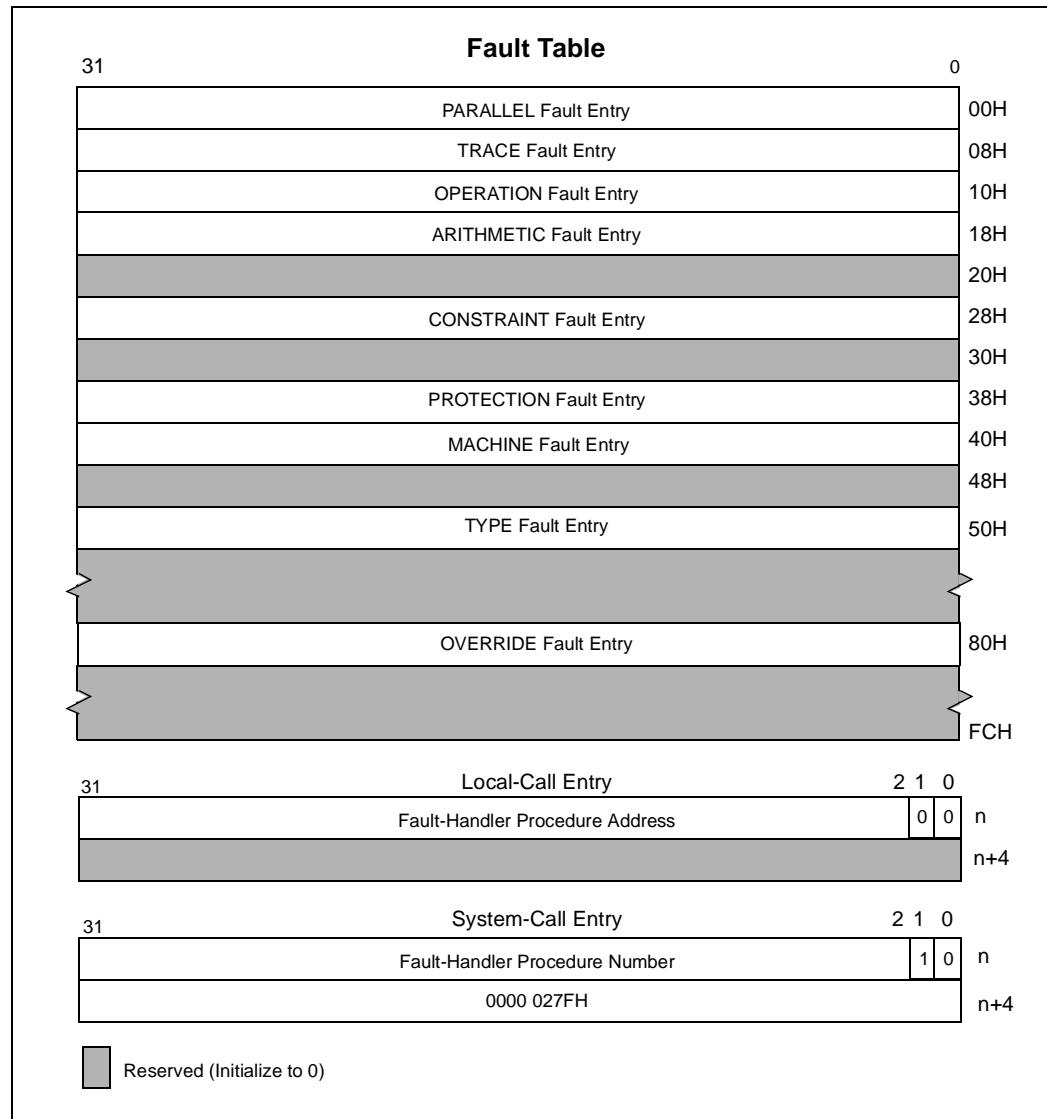
## D.2 Data Structures

## Figure D-27. Control Table

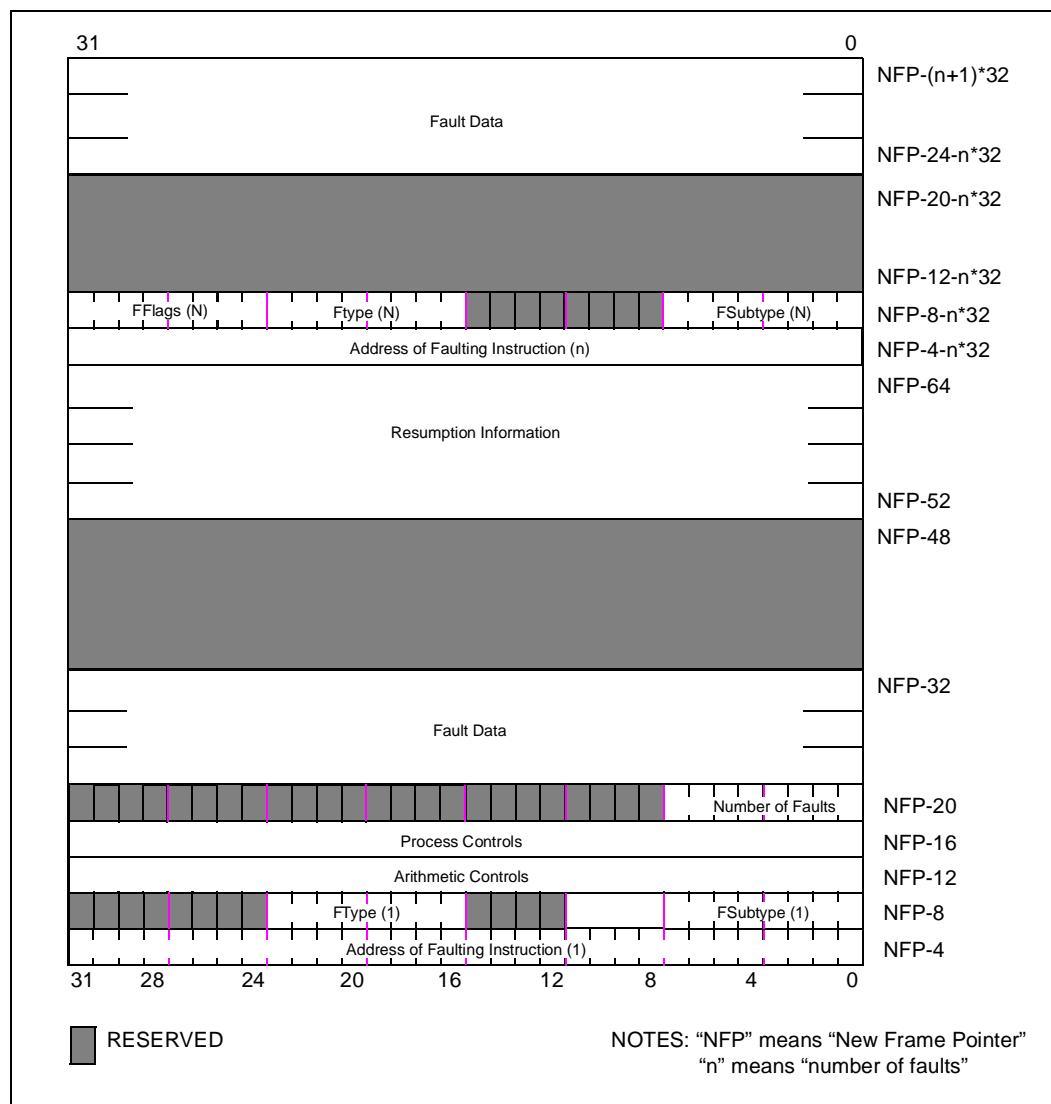
31		0
	Reserved (Initialize to 0)	00H
	Interrupt Map 0 (IMAP0)	04H
	Interrupt Map 1 (IMAP1)	08H
	Interrupt Map 2 (IMAP2)	0CH
	Interrupt Control (ICON)	10H
	Physical Memory Region 0 Configuration (PMCON0)	14H
	Physical Memory Region 1 Configuration (PMCON0)	18H
	Physical Memory Region 2 Configuration (PMCON0)	0CH
	Physical Memory Region 3 Configuration (PMCON0)	20H
	Physical Memory Region 4 Configuration (PMCON0)	24H
	Physical Memory Region 5 Configuration (PMCON0)	28H
	Physical Memory Region 6 Configuration (PMCON0)	2CH
	Physical Memory Region 7 Configuration (PMCON0)	30H
	Physical Memory Region 8 Configuration (PMCON0)	34H
	Physical Memory Region 9 Configuration (PMCON0)	38H
	Physical Memory Region 10 Configuration (PMCON0)	3CH
	Physical Memory Region 11 Configuration (PMCON0)	40H
	Physical Memory Region 12 Configuration (PMCON0)	44H
	Physical Memory Region 13 Configuration (PMCON0)	48H
	Physical Memory Region 14 Configuration (PMCON0)	4CH
	Physical Memory Region 15 Configuration (PMCON0)	50H
	Reserved (Initialize to 0)	54H
	Trace Controls (TC)	58H
	Bus Configuration Control (BCON)	5CH
		60H
		64H
		68H
		6CH

### Section 13.3.3, “Control Table” on page 13-22

**Figure D-28. Fault Table and Fault Table Entries**

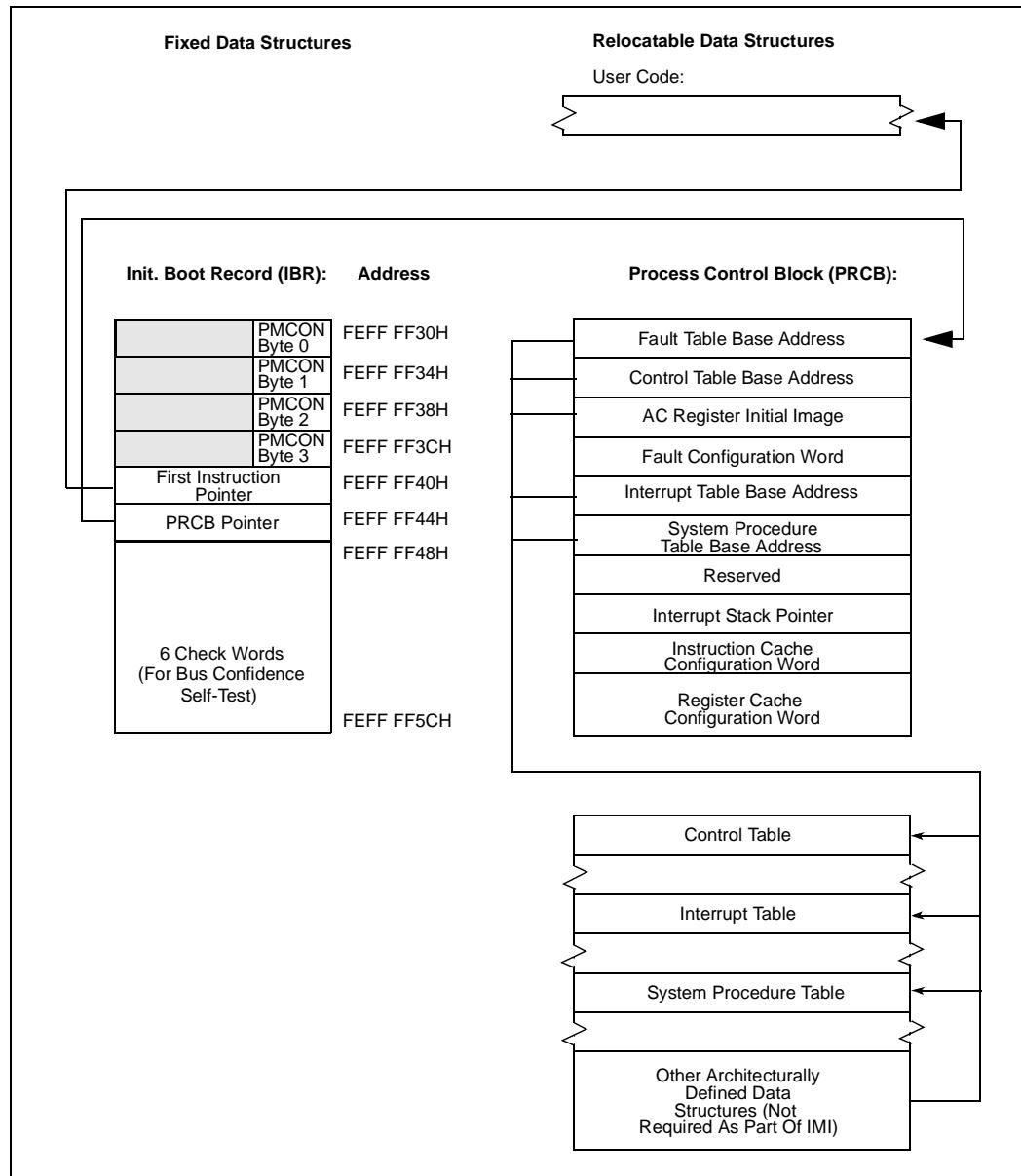


Section 8.3, “Fault Table” on page 8-5

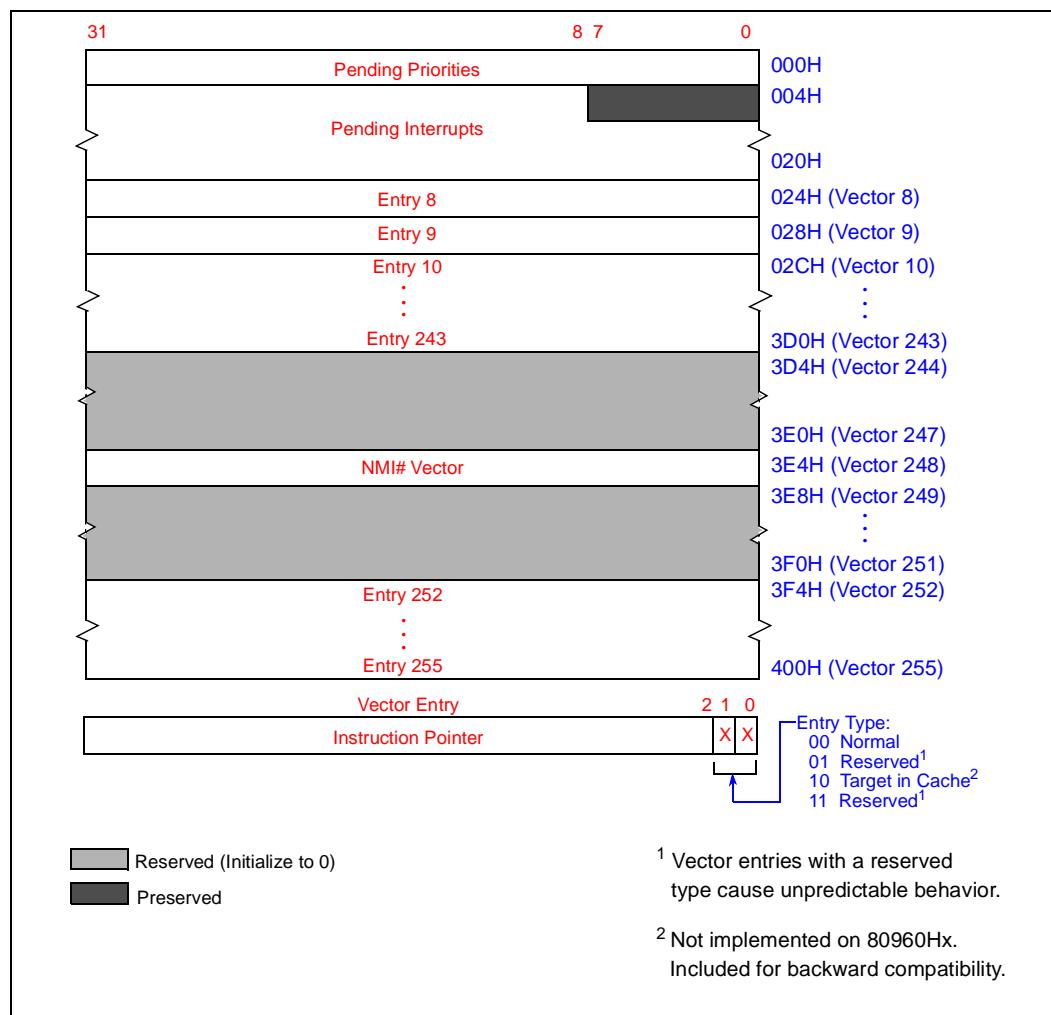
**Figure D-29. Fault Record**

Section 8.5, "Fault Record" on page 8-6

**Figure D-30. Initial Memory Image (IMI) and Process Control Block (PRCB)**

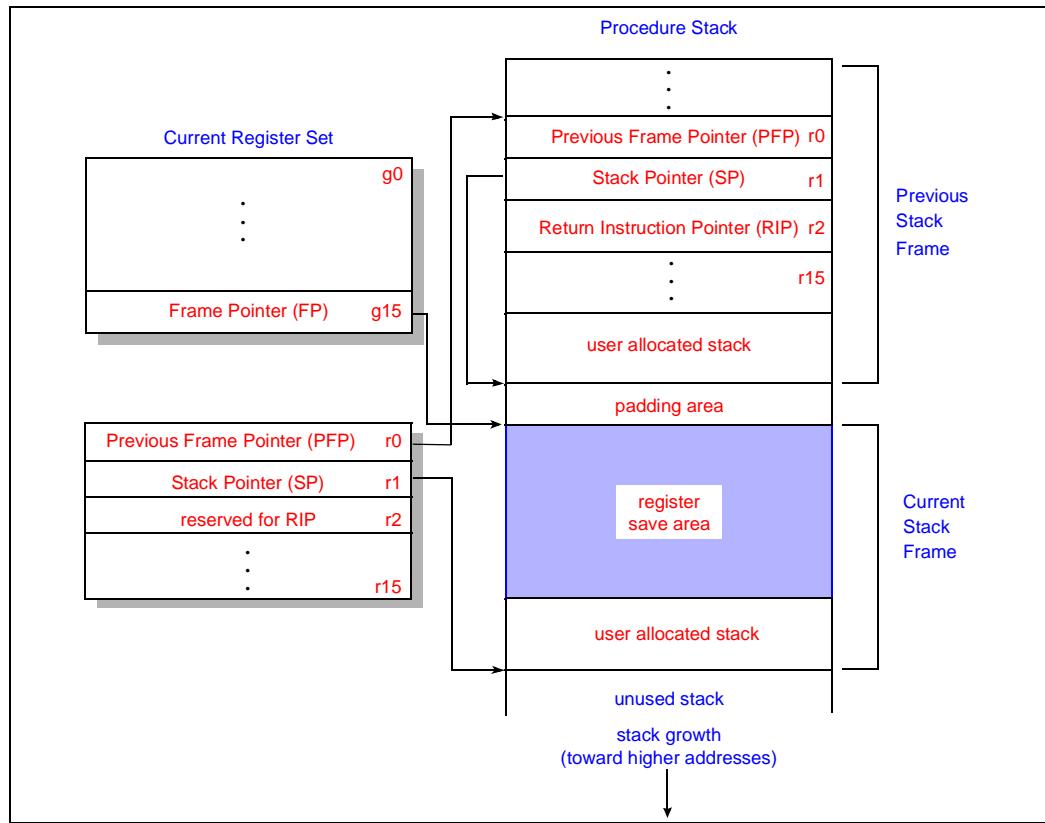


Section 13.3.1, "Initial Memory Image (IMI)" on page 13-10

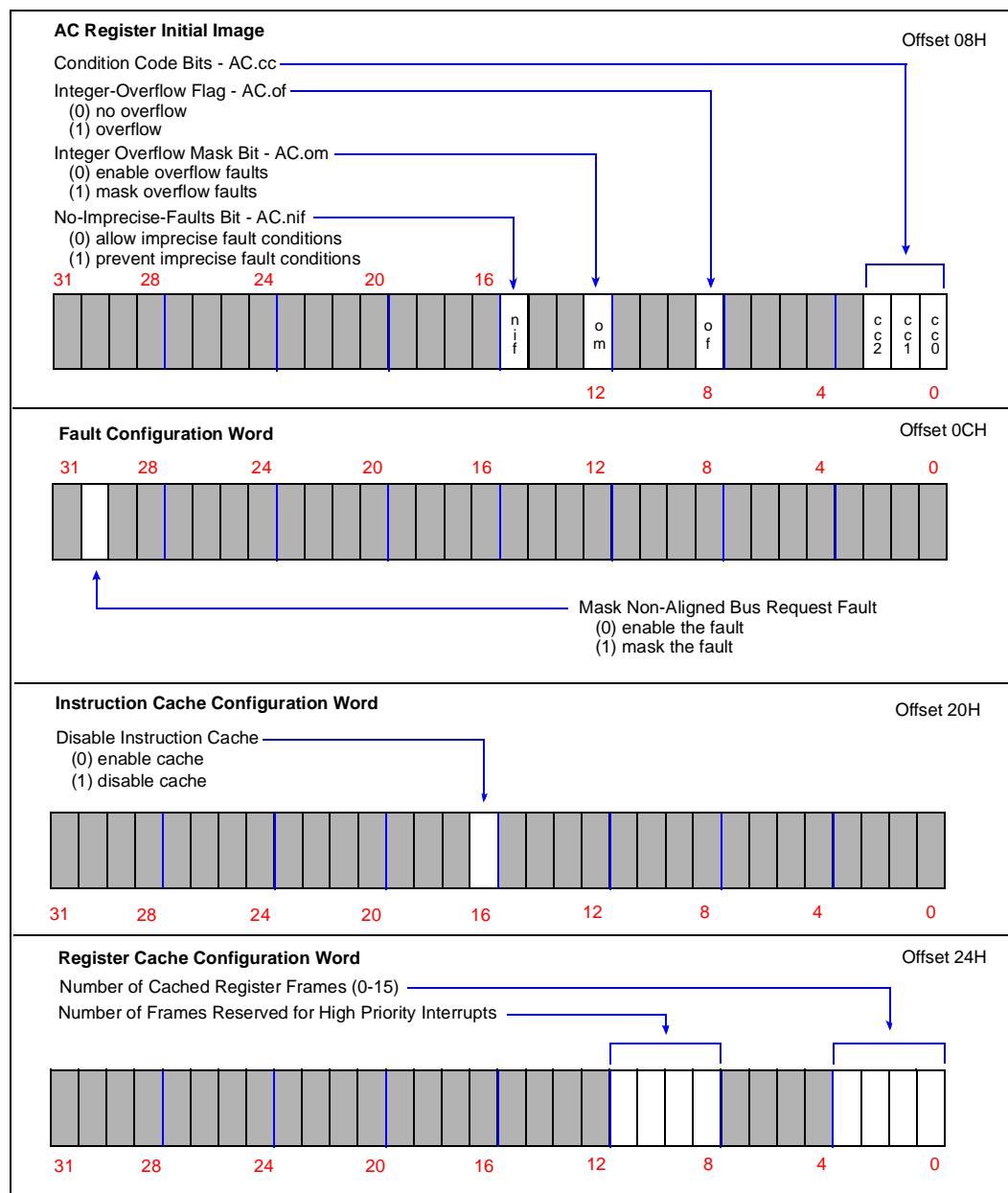
**Figure D-31. Interrupt Table**

Section 11.4, "Interrupt Table" on page 11-4

**Figure D-32. Procedure Stack Structure and Local Registers**

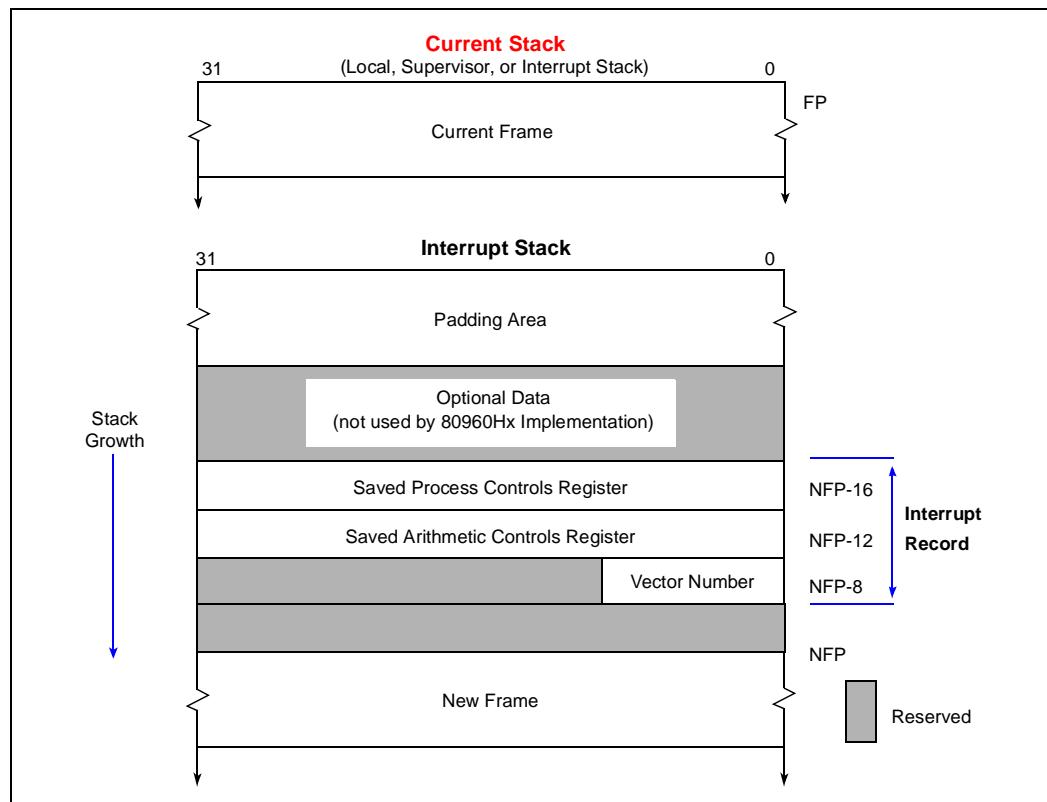


Section 7.1.1, "Local Registers and the Procedure Stack" on page 7-2

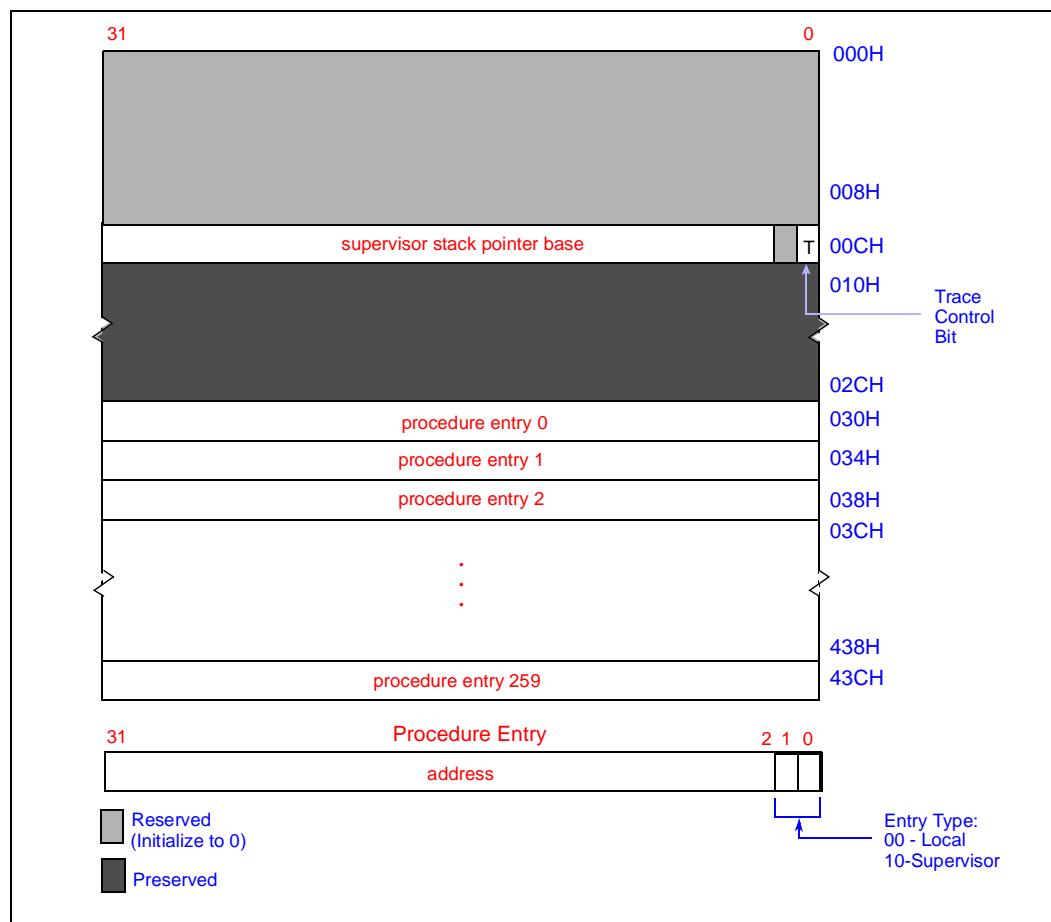
**Figure D-33. Process Control Block Configuration Words**

Section 13.3.1.2, “Process Control Block (PRCB)” on page 13-17

**Figure D-34. Storage of an Interrupt Record on the Interrupt Stack**



Section 11.5, "Interrupt Stack and Interrupt Record" on page 11-6

**Figure D-35. System Procedure Table**

Section 7.5.1, "System Procedure Table" on page 7-15

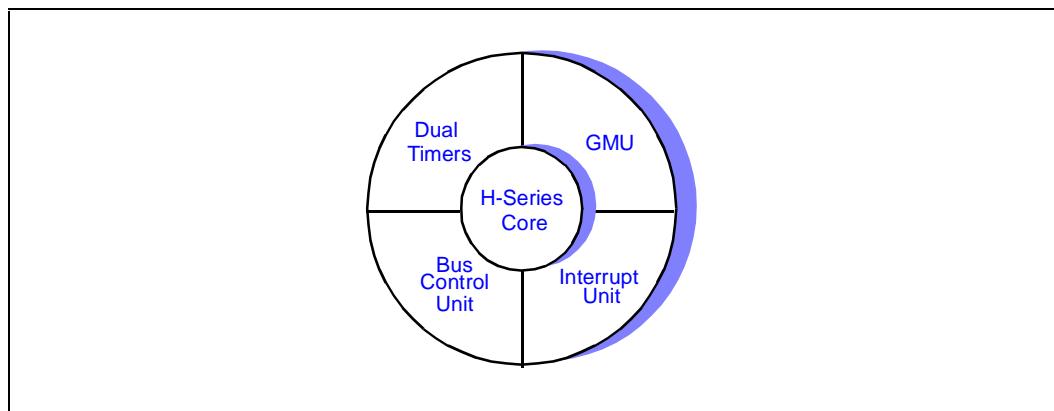
# Instruction Execution and Performance Optimization

E

This appendix describes the i960<sup>®</sup> Hx processor's core architecture and core features that enhance the processor's performance. This appendix also describes assembly language techniques for achieving the highest instruction-stream performance.

The i960 processor core architecture defines the programming environment, basic interrupt mechanism and fault mechanism for all members of the i960 microprocessor family. The H-series core is a high-performance, highly parallel implementation. The i960 Hx processor integrates a bus controller, dual timers, a guarded memory unit and an interrupt controller around the core architecture (Figure E-1).

**Figure E-1. H-Series Core and Peripherals**



State-of-the-art silicon technology and innovative micro-architectural constructs achieve high performance due to these features:

- Parallel instruction decoding allows sustained, simultaneous execution of two instructions in every clock cycle.
- Most instructions execute in a single clock cycle.
- Multiple, independent execution units enable multi-clock instructions to execute in parallel.
- Resource and register scoreboard provide efficient and transparent management for parallel execution.
- Branch look-ahead and branch prediction features enable branches to execute in parallel with other instructions.
- A local register cache permits fast calls, returns, interrupts and faults to be implemented.
- A 16-Kbyte four-way set-associative instruction cache is integrated on-chip.
- An 8-Kbyte four-way set-associative data cache is integrated on-chip.
- 2-Kbytes of static data RAM are integrated on-chip.

## E.1 Internal Processor Structure

The i960 Hx processor core contains the following main functional units:

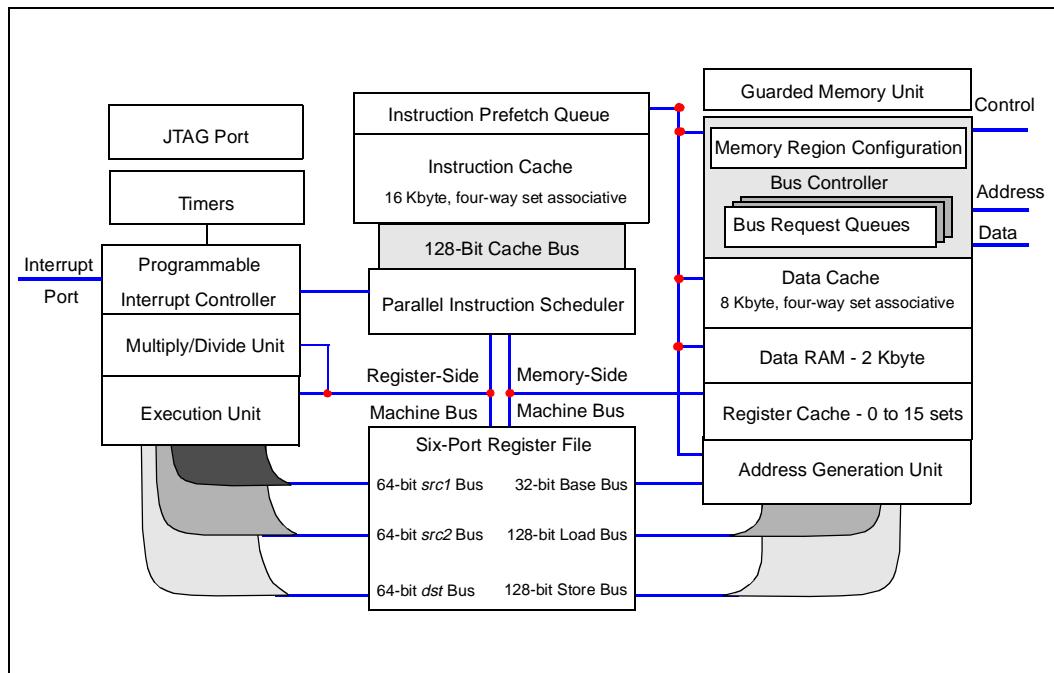
- |   |  |
|---|--|
| <ul style="list-style-type: none"> <li>• Instruction Scheduler (IS)</li> <li>• Register File (RF)</li> <li>• Execution Unit (EU)</li> </ul> | <ul style="list-style-type: none"> <li>• Multiply/Divide Unit (MDU)</li> <li>• Address Generation Unit (AGU)</li> <li>• Data RAM/Local Register Cache</li> </ul> |
|---|--|

[Figure E-2](#) shows the i960 Hx processor block diagram. The IS and RF are the “heart” of the processor. Other core functional units — referred to as *coprocessors* — interface to the IS and RF, connecting to either the register (REG) side or the memory (MEM) side of the processors.

The IS issues directives via the REG and MEM interfaces which target a specific coprocessor. That coprocessor then executes an express function virtually decoupled from the IS and the other coprocessors. The REG and MEM data buses transfer data between the common RF and the coprocessors.

The i960 Hx processors are designed to allow application-specific coprocessors to interface to the IS in the same way as core-defined coprocessors. The integrated peripherals — bus controller, interrupt controller — interface to the i960 Hx processor’s REG and MEM sides.

**Figure E-2. i960 Hx Microprocessor Block Diagram**



### E.1.1

### Instruction Scheduler (IS)

The IS decodes the instruction stream and drives the decoded instructions onto the machine bus, which is the major control bus. The IS can decode up to three instructions at a time, one from each of three different classes of instructions: one REG format, one MEM format and one CTRL format instruction. The IS directly executes the CTRL format instruction (branches), manages the instruction pipeline and keeps track of which instructions are in the pipeline so faults can be detected.

The IS is assisted by three associated functional blocks: instruction fetch unit, instruction cache and microcode ROM.

The instruction fetch unit provides the IS with up to four words of instructions each cycle. It extracts instructions from the instruction cache, microcode ROM and its instruction fetch queue for presentation to the scheduler. The instruction fetch unit requests external fetch operations from the bus controller whenever a cache miss occurs.

The instruction cache is 16-Kbyte and four-way set associative. This cache delivers to the IS up to four instructions per clock. The cache allows many inner loops of code to execute with no external instruction fetches; this maximizes the core's performance.

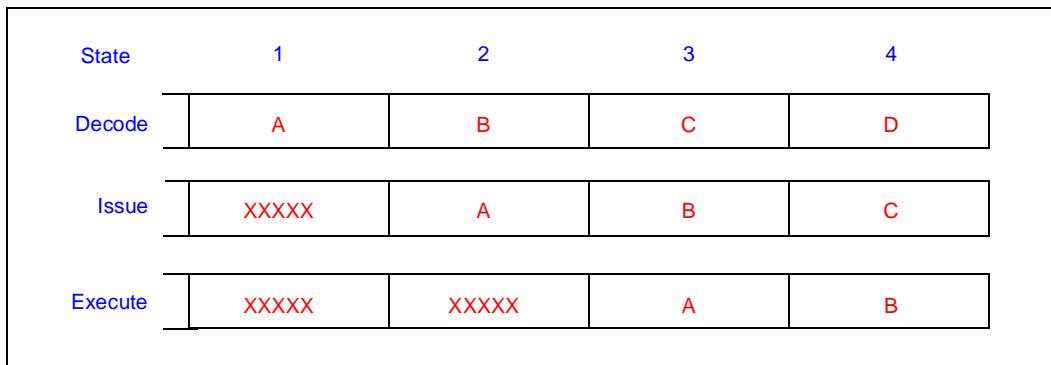
The i960 Hx processors use a microcode engine to implement complex instructions and functions. This includes implicit and explicit calls, returns and initialization sequences. Unlike conventional microcode, i960 Hx processor microcode uses a RISC subset of the instruction set in addition to specific micro-instructions. Microcode, therefore, can be thought of as a RISC program containing operational routines for complex instructions. When the instruction pointer references a microcoded instruction, the instruction fetch unit automatically branches to the appropriate microcode routine. The i960 Hx processors perform this microcode branch in 0 clocks.

## E.1.2 Instruction Flow

Most instructions flow through a three-stage pipeline (Figure E-3):

- The decode stage calculates the address used to fetch the next instruction from the instruction cache. Additionally, this stage starts decoding the instruction.
- The issue stage completes instruction decode and sends it to the appropriate execution unit.
- During the execute stage, the operation is performed and the result is returned to the RF.

**Figure E-3. Instruction Pipeline**



In the decode stage, the IS decodes the instruction and calculates the next instruction address. This could be a macro- or micro-instruction address. It is either the next sequential address or the target of a branch. For conditional branches, the IS uses condition codes or internal hardware flags to determine which way to branch. If branch conditions are not valid when the IS sees a branch, the processor guesses the branch direction, using the branch prediction specified in the instruction. If the guess was wrong, the IS cancels the instructions on the wrong path and begins fetching along the correct path.

In the issue stage, instructions are emitted or issued to the rest of the machine via the machine bus. The machine bus consists of three parts: REG format instruction portion, MEM format instruction portion and CTRL format portion. Each part of the machine bus goes to the coprocessor that executes the appropriate instruction. The RF supplies operands and stores results for REG and MEM format instructions. For this reason, the RF is connected to both the REG and MEM portions of the machine bus. The CTRL portion stays within the instruction sequencer since it directly executes the branch operations. Several events occur when an instruction is issued:

1. The information is driven onto the machine bus.
2. The IS reads the source operands and checks that all resources needed to execute the instruction are available.
3. The instruction is cancelled if any resource that the instruction requires is busy. The resource is busy if a previous, incomplete instruction reserved it or the resource is already working on an instruction.
4. The IS then attempts to re-issue the instruction on the next clock; the same sequence of events is repeated.

This processor resource management mechanism is called *resource scoreboarding*. A specific form of resource scoreboarding is register scoreboarding. When an instruction's computation stage takes more than one clock, the result registers are scoreboxed. A subsequent operation needing that particular register is delayed until the multi-clock operation completes. Instructions which do not use the scoreboxed registers can execute in parallel.

The execute stage performs the instruction. This stage is handled by the coprocessors which connect to the REG- and MEM-side buses. In this stage, the coprocessor has received operands from the RF and recognized opcode which tells the coprocessor which instruction to execute. Execution begins and a result is returned in this stage for single-clock instructions.

The execute stage is a single- or multi-clock pipeline stage, depending on the operation performed and the coprocessor targeted. For single-clock coprocessors—such as the integer execution unit—the result of an operation is always returned immediately. Because of the three-stage pipeline construction and the register bypassing mechanism, no conflicts between source access and result return can occur. For multi-clock coprocessors—such as the multiply/divide unit—the coprocessor must arbitrate access to the return path.

### E.1.3 Register File (RF)

The RF contains the 16 local and 16 global registers and has six ports (Figure E-4). This allows several of the core's coprocessors to access the register set in parallel. This parallel access results in an ability to execute one simple logic or arithmetic instruction, one memory operation (**LOAD/STORE**) and one address calculation per clock.

**Figure E-4. Six-Port Register File**



MEM coprocessors interface to the RF with a 128-bit wide load bus and a 128-bit wide store bus. An additional 32-bit port allows the Address Generation Unit to simultaneously fetch an address or address reduction operand. These wide load and store data paths:

- enable up to four words of source data and four words of destination data to simultaneously pass between the RF and a MEM coprocessor in a single clock.
- provide a high-bandwidth path between data RAM, data cache and local register cache to implement high-speed data operations.
- provide a highly efficient means for moving load, store and fetch data between the bus controller and the RF.

REG coprocessors interface to the RF with two 64-bit source buses and a single 64-bit destination bus. The source and result from different REG coprocessors can access the RF simultaneously using this bus structure. The 64-bit source and destination buses allow the **eshro**, **mov** and **movl** instructions to execute in a single cycle.

To manage register dependencies during parallel register accesses, *register bypassing* (result forwarding) is implemented. The register bypassing mechanism is activated whenever an instruction's source register is the same as the previous instruction's destination register. The instruction pipeline allows no time for the contents of a destination register to be written before it is read again by another instruction. Because of this, the RF forwards the result data from the return bus directly to the source bus without reading the source register.

## E.1.4 Execution Unit (EU)

The EU is the i960 Hx processor core's 32-bit arithmetic and logic unit. The EU can be viewed as a self-contained REG coprocessor with its own instruction set. As such, the EU is responsible for executing or supporting the execution of all integer and ordinal arithmetic instructions, logic and shift instructions, move instructions, bit and bit-field instructions and compare operations. The EU performs any arithmetic or logical instructions in a single clock.

## E.1.5 Multiply/Divide Unit (MDU)

The MDU is a REG coprocessor which performs integer and ordinal multiply, divide, remainder and modulo operations. The MDU detects integer-overflow and divide-by-zero errors. The MDU is optimized for multiplication, performing extended multiplies (32 by 32) in four to five clocks. The MDU performs multiplies and divides in parallel with the main execution unit.

## E.1.6 Address Generation Unit (AGU)

The AGU is a MEM coprocessor which computes the effective addresses for memory operations. It directly executes the load address instruction (**lda**) and calculates addresses for loads and stores based on the addressing mode specified in these instructions. Address calculations are performed in parallel with the main execution unit (EU).

### E.1.7 Data RAM and Local Register Cache

The data RAM and local register cache are part of a 2.5 Kbyte block of on-chip Static RAM (SRAM). Two Kbytes of this SRAM are mapped into the i960 Hx processor's address space from location 0000 0000H to 0000 07FFH. A portion of the remaining 512 bytes is dedicated to the local register cache. This part of internal SRAM is not directly visible to the user. Loads and stores, including quad word accesses to the internal data RAM, are typically performed in only one clock. The complete local register set, therefore, can be moved to the local register cache in only four clocks.

## E.1.8 Data Cache

The i960 Hx processor has an 8-Kbyte four-way set-associative data cache which enhances performance by reducing the number of load and store accesses to external memory. The data cache can return up to a quad word (128 bits) to the register file in a single clock cycle on a cache hit.

External memory is configured as cacheable or non-cacheable on a region-by-region basis, using special bits in the memory region configuration LMCON registers. This makes it easy to partition a system into cacheable regions and non-cacheable regions.

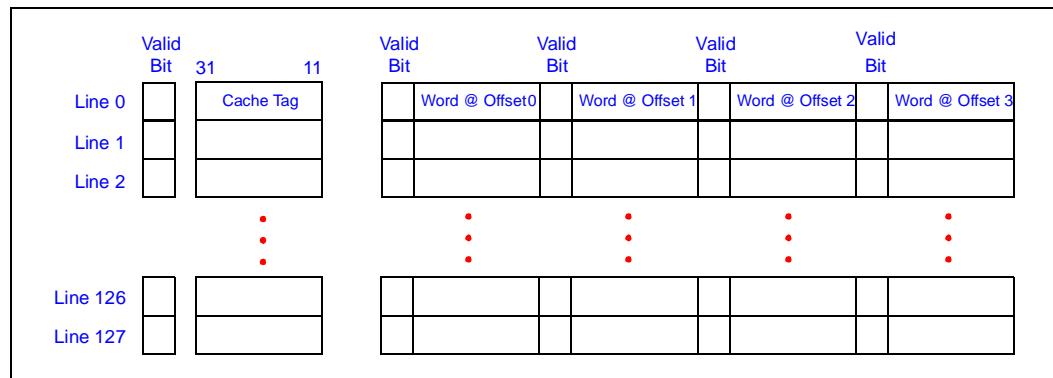
The i960 Hx processor implements a simple coherency mechanism. The **dcctl** instruction is used to invalidate all cacheable regions marked as quick-invalidate. The data cache can also be enabled, disabled or invalidated on a global basis using the **dcctl** instruction.

#### E.1.8.1. Data Cache Organization

The data cache has a four word line size (see Figure E-5). Each of the 128 cache lines has an associated cache tag containing the 21 most significant bits of the address and a valid bit. Each line is further subdivided into single word blocks, each with its own valid bit. This *subblock placement* technique reduces latency on cache misses.

Data accesses result in cache hits and misses. Accesses that match valid address tags and word(s) marked as valid are cache hits; other data accesses are misses.

**Figure E-5. Data Cache Organization**



### E.1.8.2. Bus Configuration

Certain data accesses are implicitly non-cacheable. All atomic (**atmod**, **atadd**) accesses are non-cacheable. User settings in the LMCN registers determine which data accesses are cacheable or non-cacheable. Refer to [Section 14.1.2.2, “Logical Memory Region Cacheability” on page 14-5](#).

Upon reset or initialization, the processor clears all valid bits to zero to ensure that accesses are not made to a cache line that may contain invalid data.

### E.1.8.3. Global Control of the Cache

The following example code shows how to disable/enable the cache using the **dcctl** command. Disabling the cache does not invalidate any of its entries.

```
dcctl 0,0,r4    # Disable
dcctl 1,0,r4    # Enable
```

### E.1.8.4. Data Fetch Policy

Data fetch policy determines what happens to a load that misses the cache. The processor employs a *natural* fetch policy. Word, double word, triple word and quad word loads are issued to the bus control logic in their original widths. Byte and short word loads are promoted to word bus requests. Because most applications have 32-bit data buses, there is seldom a bandwidth penalty for promoting a byte or short word load to a full word bus operation.

### E.1.8.5. Write Policy

Write policy determines what happens on cacheable store operations. The write policy for the i960 Hx processor is *write-through* and *write-allocate*. For cacheable stores, data is written into both the cache and external memory simultaneously, regardless of whether the write is a hit or miss. This maintains coherency between the data cache and external memory.

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 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, on a cache miss, the other three valid bits are always cleared.

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 all cases the processor simultaneously writes the data into the cache and the external memory.

### E.1.8.6. Data Cache Coherency

Whenever the cacheability of a region is changed, cache coherency becomes an immediate issue. The coherency mechanism solves this issue directly. The processor compares a non-cacheable store to the relevant tag in the data cache. If the store address matches the tag, the processor invalidates all words in the cache line. In a single processor system, this guarantees that the data cache never contains stale data. When the data cache is globally disabled, all stores are non-cacheable and the processor invalidates relevant tags whenever addresses match.

Atomic accesses from the **atmod** and **atadd** instructions are implicitly non-cacheable. Otherwise, entire memory regions would have to be programmed as non-cacheable to support semaphore operations.

### E.1.8.7. BCU Pipeline and Data Cache Interaction

The BCU's interaction with the data cache affects overall bus throughput. [Figure E-6](#) shows how the BCU and data cache process a series of hits and misses for cacheable loads and stores.

ld	(g0),g1	:data cache hit
ld	(g2),g4	:data cache miss
ldq	(g3),g8	:data cache hit
st	g1,(g0)	:store is scoreboarded

[Figure E-6. BCU and Data Cache Interaction](#)

Instruction Scheduler	ld	ld	— —	ldq	— —	— —	st		
BCU Pipeline	Address Out Bus			g2			g0		
	St Bus						g1		
	External Address Bus			g2			g0		
	External Data Bus				(g2)			g1	
	LD Bus					g4←(g2)			
Data Cache Pipeline	Address Out Bus	g0 Hit	g2 Miss		g3 Hit		g0		
	St Bus						Cache←g1		
	LD Bus	g1←(g0)		quad g8←(g3)	Cache ←(g2)				

During the first issue clock, the data cache receives the first load address and recognizes a cache hit. The following clock is an execute clock; the cache returns data to the register file over the LD bus. In the next issue clock the cache receives the second load address and recognizes a miss. It is passed on to the BCU in the following clock. The BCU then processes the load as if there were no data cache. Note that the following load quad instruction is scoreboarded for a single clock while the previous cache miss is issued to the BCU. The load quad instruction is determined to be a hit in the third issue clock and all 128 bits of data are returned to the register file in the following execute clock.

The i960 Hx processor scoreboards the store instruction until the pending load returns data to the cache. The processor writes the data to the register file and the cache in the same clock, updating the cache tag and valid bits. In the next clock, the store instruction is issued. For the store, the processor writes unconditionally into the cache during the issue clock.

When using the i960 Hx processor, refer to [Table E-1](#) for a listing of the single clock load and store instructions. The table is valid when offset, displacement or indirect addressing modes are used over an external bus with the following characteristics:

$N_{XAD} = N_{XDD} = N_{XDA} = 0$ , Burst On, Pipelining On, Ready Disabled

For other addressing modes, information in [Section E.2.6, “Micro-flow Execution”](#) on page E-29 applies.

For each instruction that requires multiple reads on the external bus, such as **ldq**, the BCU buffers the return data until all data is returned from the bus. This optimization reduces the internal load bus overhead to a minimum and allows the processor to access the MEM-side while external loads are in progress. If instructions are issued back-to-back with no register dependencies and hit the cache, execution can proceed at the rate of one instruction per clock. For cache misses, the processor issues instructions until the cache is full. Subsequent back-to-back execution proceeds at bus bandwidth.

**Table E-1. BCU Instructions**

<b>Mnemonic</b>	<b>Issue Clocks</b>	<b>Result Latency Clocks</b>	<b>Back-to-Back Throughput</b>	<b>Result Latency Clocks</b>	<b>Back-to-Back Throughput</b>
		<b>Hits</b>	<b>Hits</b>	<b>Misses</b>	<b>Misses</b>
<b>ld</b> <b>ldob</b> <b>ldib</b> <b>ldos</b> <b>ldis</b>	1	1	1	4	2
<b>ldl</b>	1	1	1	6	2
<b>ldt</b>	1	1	1	7	3
<b>ldq</b>	1	1	1	8	4
<b>st</b> <b>stob</b> <b>stib</b> <b>stos</b> <b>stis</b>	1	N/A	2	N/A	2
<b>stl</b>	1	N/A	3	N/A	3
<b>stt</b>		N/A	4	N/A	4
<b>stq</b>		N/A	5	N/A	5

### E.1.8.8. BCU Queues and Cache Coherency

The bus control unit is implemented as a coprocessor. Many clock cycles can pass after a cacheable load instruction is issued before data is returned to the data cache and registers. Because of this delay, the BCU was modified to support data cache operation. The processor scoreboards all stores when cacheable loads are present in the BCU queue. Consider the following case:

```
ld xyz, R0# load from address xyz misses the data cache  
st r4, xyz# store is issued to the same address
```

The load instruction misses the data cache and is then issued to the bus control unit. It can take several clocks before data is actually written to r0 and the data cache. If the store were issued before the load returns data, an inconsistency would result. External memory would receive correct data from the store, but the data cache would contain incorrect data from the load. The processor prevents this inconsistency by stalling the store until the load returns data.

The policy of scoreboardng stores on outstanding cacheable loads typically decreases overall processor performance by less than one percent.

### E.1.8.9. External I/O and Bus Masters and Cache Coherency

The i960 Hx 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.

Users typically program the LMCN registers such that I/O regions are non-cacheable. Partitioning the system this way eliminates I/O as a source of coherency problems.

## E.2 Parallel Instruction Processing

At the center of the i960 Hx processor core is a set of parallel processing units capable of executing multiple single-clock instructions in every clock. To support this rate, the IS can issue up to three new instructions in every clock. Each processing unit has access to the multiple ports of the chip's six-ported register file; therefore, each processing unit can execute instructions independently and in parallel.

In general, the register file, instruction scheduler, cache and fetch unit keep the parallel processing units busy, given the typical diversity of instructions found in a rolling quad word group of instructions. To achieve highly optimized performance for critical code sequences, the user must understand how instructions execute on the processor.

The following section describes instruction execution on the i960 Hx processor with the goal of instruction stream optimization in mind. See [Section , “” on page E-34](#) for specific optimization techniques applicable to the i960 Hx processors.

## E.2.1 Parallel Issue

The IS looks at a rolling quad word group of unexecuted instructions every clock and issues all instructions that can be executed in that clock. The scheduler can issue up to three instructions every clock to the processing units and can sustain an issue rate of two instructions per clock. To achieve parallelism, the IS detects to which machine “side” — REG, MEM or CTRL — each instruction in the current quad word group belongs.

When the IS issues a group of instructions, the appropriate parallel processing units acknowledge receipt and begin execution. However, register and resource dependencies can delay instruction execution. The processor transparently manages these interactions through register scoreboarding and register bypassing.

To maximize the IS’s ability to issue instructions in parallel, the instruction cache is organized to provide four instructions per clock to the scheduler. To minimize the cost of a cache miss, the instruction fetch unit constantly checks whether a cache miss will occur on the next clock. If a miss is imminent, an instruction fetch is issued.

The following discussions assume that instructions are always available from the instruction cache. For a discussion of cache organization and the impact of cache misses, see [Section E.2.5, “Instruction Cache and Fetch Execution” on page E-27](#).

## E.2.2 Parallel Execution

Six parallel processing units are attached to the six-ported register file:

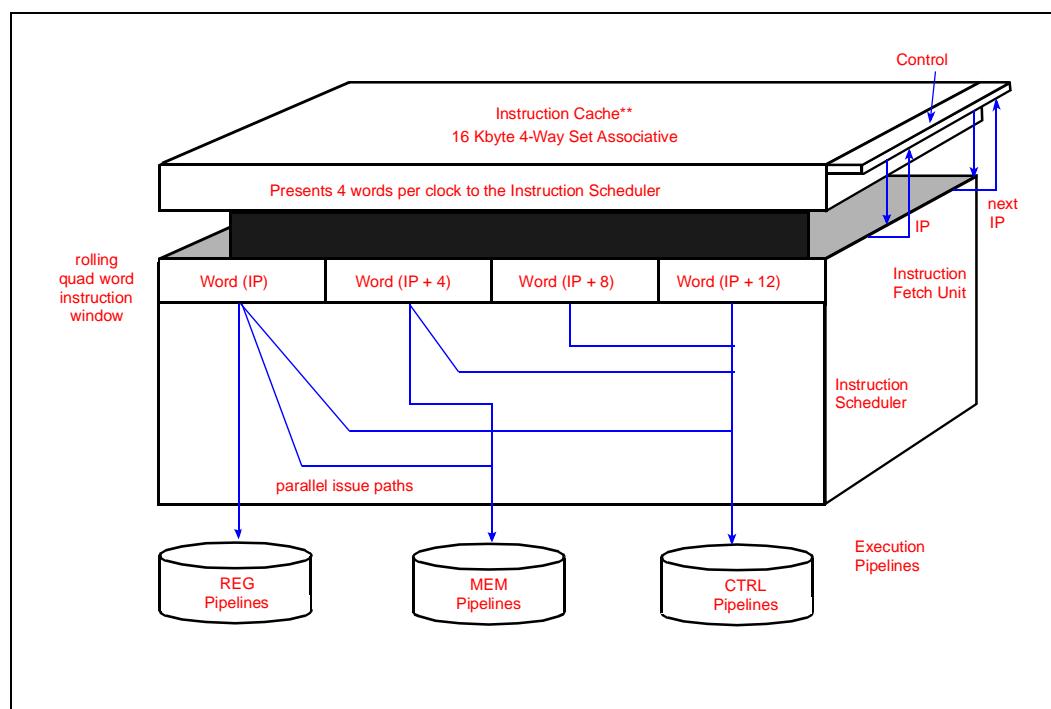
MEM-side:	Three units are attached to the machine’s memory side. MEM-side instructions are dispatched over the MEM machine-bus.
BCU	Bus Control Unit executes memory reads and writes for instructions which reference an operand in external memory.
DR	Data RAM handles memory reads and writes for instructions that reference on-chip data RAM and MMR space.
AGU	Address Generation Unit executes the <b>lda</b> , <b>callx</b> , <b>bx</b> and <b>balx</b> instructions and assists address calculation for all loads and stores.
REG-side:	Two units are attached to the register side. REG-side instructions are dispatched over the REG machine bus.
MDU	Multiply/Divide Unit executes the multiply, divide, remainder, modulo and extended multiply and divide instructions.
EU	Execution Unit executes all other arithmetic, logical, shift, comparison, bit, bit field, move instructions and the <b>scanbyte</b> instruction.
CTRL-side:	One unit is on the control side.
IS	Instruction Scheduler directly executes control instructions by modifying the next instruction pointer given to the instruction cache.

The processor uses on-chip ROM to execute instructions not directly executed by one of the parallel processing units. This ROM contains a sequence of RISC instructions for each complex instruction not directly executable in one of the parallel processing units. When the scheduler encounters a complex instruction, the appropriate sequence of RISC instructions is issued for execution. This sequence of instructions is called a *micro-flow*.

The IS can issue multiple instructions in every clock when the instructions decoded in that clock can be executed by different machine sides. For example, an add can begin in the same clock as a load since the addition is performed by the EU on the REG side, while the load is executed by the BCU on the MEM side. Furthermore, a branch can be issued in the same clock as the add and load since the IS executes it directly (three instructions per clock). The IS does not exploit every possible combination of three instruction types in four consecutive words. [Table E-2](#) summarizes the sequences of instruction machine types that can be issued in parallel. A group of one or more instructions which can be issued in the same clock is referred to in this appendix as an *executable group* of instructions. [Figure E-7](#) shows the paths that the IS has available for dispatching each word of the rolling quad word to the three machine sides.

**Table E-2. Machine Type Sequences that Can Be Issued in Parallel**

Sequence	Description
R M x x	REG-side followed immediately by a MEM-side instruction
R M C x	REG-side followed immediately by a MEM-side followed immediately by a CTRL instruction
R M x C	REG-side followed immediately by a MEM-side followed by a CTRL instruction in the same rolling quad word
R C x x	REG-side followed immediately by a CTRL instruction
R x C x R x x C	REG-side followed by a CTRL instruction in the same rolling quad word
M C x x	MEM-side followed immediately by a CTRL instruction
M x C x M x x C	MEM-side followed by a CTRL instruction in the same rolling quad word

**Figure E-7. Issue Paths**

### E.2.3 Scoreboarding

When the scheduler issues a group of instructions, the targeted parallel processing units immediately acknowledge receipt of instructions and the scheduler begins considering the next four unexecuted words of the instruction stream. The scheduler checks for register dependencies between instructions *before* issuing them. The scheduler does not issue a group of instructions if:

1. a register is specified as a destination more than once, or
2. a register is specified as a destination in one instruction and a source in a subsequent instruction

A single register may, however, be specified as a source in multiple instructions or as a source in one instruction and a destination in a subsequent instruction. The six-port register set supports these cases. For example, the following instructions *cannot* be issued in parallel due to register dependencies:

```

addo    g0, g1, g2      # g2 is a destination
st     g2, (g3)        # g2 is a source;4.45
                  # store must wait for addo to complete

```

or:

```

addog0,   g1, g2      # g2 is a destination
ld      (g3), g2      # g2 is also a destination;
                  # load must wait for addo to complete

```

However, the following instructions *can* be issued in parallel:

```

addog0,   g1, g2      # g0 is a source for both instructions
st     g0, (g3)

```

or:

```

addog0,   g1, g2      # g0 is a source for addo and
ld      (g3), g0      # a destination for load

```

In all cases of parallel instruction issue, the IS ensures that the program operates as if the instructions were actually issued sequentially.

Two conditions can delay the execution of one or more of the instructions that the scheduler attempted to issue: a *scoreboarded register* or a *scoreboarded resource*.

### E.2.3.1. Register Scoreboarding

If an instruction's source (or destination) register is the destination of a prior incomplete multi-clock instruction (such as a load), the instruction is delayed. The scheduler attempts to reissue the instruction every clock until the scoreboardd register is updated and the delayed instruction can be executed. [Table E-3](#) summarizes conditions which cause a delay due to a scoreboardd register.

**Table E-3. Scoreboarded Register Conditions**

Condition	Description
<i>src busy</i>	One or both of the registers specified as a source for the instruction was referenced as a destination of a prior instruction which has not completed.
<i>dst busy</i>	The destination referenced by the instruction was referenced as a destination of a prior instruction which has not completed.
<i>cc busy</i>	AC register condition codes are not valid. Correct branch prediction eliminates dead clocks due to condition code dependencies.

### E.2.3.2. Resource Scoreboarding

A scoreboarded resource also defeats the scheduler's attempt to issue an instruction. A resource is scoreboarded when it is needed to execute the instruction but is not available. The parallel processing units are the resources. [Table E-4](#) lists cases which cause an instruction to be delayed due to a(scoreboarded resource. The description that follows the table describes what happens to an instruction once it is issued to a processing unit.

### E.2.3.3. Prevention of Pipeline Stalls

To maintain the logical intent of the sequential instruction stream, the i960 Hx processors implement register scoreboarding and register bypassing. Examples of each are demonstrated in the descriptions and examples in this appendix. These mechanisms eliminate possible pipeline stalls due to parallel register access dependencies. It is not necessary to perform any code optimizations to take advantage of this parallel support hardware.

**Table E-4. Scoreboarded Resource Conditions**

Condition	Description
BCU Queue Full	Bus Controller queues are full and the scheduler is attempting to issue a memory request.
MDU Busy	The Multiply/Divide Unit is busy executing a previously issued instruction and the scheduler is attempting to issue another instruction for which the MDU is responsible.
DR Busy	On-chip data RAM can support one 128-bit load or store every clock. However, the data RAM has no queues for storing requests. The unit stalls execution if a new request is issued to it when it has not been allowed to return data from a prior instruction.  For example, if the DR and BCU attempt to return results over the load bus in the same clock, the BCU wins the arbitration. This delays the DR result by one clock. If, simultaneously, the IS is attempting to issue another instruction to the data RAM, the DR stalls the processor for one clock.

Register scoreboarding maintains register coherency by preventing parallel execution units from accessing registers for which there is an outstanding operation. When the IS issues an instruction that requires multiple clocks to return a result, the instruction's destination register is locked to further accesses until it is updated. To manage this destination register locking, the processors use a 33rd bit in each register to indicate whether the register is available or locked. This bit is called the scoreboard bit. There is a scoreboard bit for each of the 32 registers.

Register bypassing eliminates a pipeline stall that would otherwise occur when one parallel processing unit is returning a result to a register over one port while, in the same clock, another unit is accessing the same register over a different port. Register bypassing logic constantly monitors all register addresses being written and read. If a register is being read and written in the same clock, bypass logic routes incoming data from the write port directly to the read port.

## E.2.4 Processing Units

Once the IS issues a group of instructions, the appropriate processing units begin instruction execution in parallel with all other processor operations. The following sections describe each unit's pipelines and execution times of the instructions they process.

### E.2.4.1. Execution Unit (EU)

The EU performs arithmetic, logical, move, comparison, bit and bit-field operations. The EU receives its instructions over the REG-machine bus and receives source operands over the *src1* and *src2* buses and returns its result over the *dst* bus.

The EU pipeline is shown in [Figure](#). In the clock in which an EU instruction is issued, the EU latches the source operands and begins performing the operation. In the following clock, the instruction completes and the result is written to the destination register. When an instruction immediately follows an EU operation that references the EU's destination register, the new instruction is issued in the clock following the EU operation.

The EU directly executes the instructions listed in [Table E-5](#). The EU is pipelined such that back-to-back EU operations execute at a one-clock sustained rate. The EU returns its result to the destination register in the clock following the clock in which the instruction was issued. If a fix-up is needed during **shrdi** execution, the processor executes a four-clock micro-flow. See [Section E.2.6, “Micro-flow Execution” on page E-29](#).

```

addo    g0, g1, g2
shlo   g3, g4, g5
subo   g5, g6, g7
shro   g8, g9, g10
    
```

**Figure E-8. EU Execution Pipeline**

Instruction Scheduler	Issue	addo	shlo	subo	shro	
EU Pipeline	Read src 1, <i>src2</i>	g0, g1	g3, g4	g5, g6	g8, g9	
	Execute and Write <i>dst</i>		g2←g0+g1	g5←g4<<g3	g7←g6-g5	g10←g9>>g8

**Table E-5. EU Instructions**

<b>addo</b> <b>addi</b> <b>addc</b> <b>ADD&lt;cc&gt;</b> <b>subo</b> <b>subi</b> <b>subc</b> <b>SUB&lt;cc&gt;</b>  <b>setbit</b> <b>clrbit</b> <b>notbit</b>	<b>shlo</b> <b>shro</b> <b>shri</b> <b>shli</b> <b>shrdi</b> <b>eshro</b>  <b>alterbit</b> <b>chkbit</b>	<b>mov</b> <b>movl</b> <b>cmpo</b> <b>cmpi</b> <b>cmpdeco</b> <b>cmpdeci</b> <b>compare byte</b> <b>compare short</b> <b>SEL&lt;cc&gt;</b>  <b>scanbyte</b> <b>bswap</b>	<b>and</b> <b>andnot</b> <b>notand</b> <b>nand</b> <b>or</b> <b>nor</b> <b>ornot</b> <b>notor</b> <b>xnor</b> <b>xor</b> <b>not</b> <b>rotate</b>
---	--	---	--

### E.2.4.2. Multiply/Divide Unit (MDU)

The MDU performs multiplication, division, remainder and modulo operations. The MDU receives its instructions over the REG-machine bus and source operands over the *src1* and *src2* buses and returns its result over the *dst* bus. Once the IS issues an MDU instruction, the MDU performs its operations in parallel with all other execution.

The MDU pipeline for the 32x32 **mulo** instruction is shown in [Figure](#). In the clock in which the multiply is issued, the MDU latches the source operands and begins the operation. The multiply completes and the result is written to the destination register in the fifth clock following the clock in which the instruction was issued. When an instruction immediately follows a multiply which references the multiply's destination, the instruction is not issued until the clock in which the multiply result is returned. For example, an **addo** which follows a multiply and references the destination of the multiply is delayed until the fourth clock after the processor issues the multiply. This five-clock multiply latency is easily hidden; four to eight instructions could be placed between the multiply and add without increasing the total number of processor clocks used.

```

addo    g0, g1, g2
mulo    g3, g4, g5
addo    g5, g6, g7

```

[Figure E-9. MDU Execution Pipeline](#)

Instruction Scheduler	Issue	addo	mulo	---	---	---	---	addo	
EU Pipeline	Read <i>src1</i> , <i>src2</i>	g0, g1						g5, g6	
	Execute and Write <i>dst</i>		$g2 \leftarrow g0+g1$						$g7 \leftarrow g5+g6$
MDU Pipeline	Read <i>src1</i> , <i>src2</i>		g3, g4						
	Execute								
	Write <i>dst</i>							$g5 \leftarrow g3*g4$	

The MDU incorporates a one-clock pipeline unless integer overflow faults are enabled. The IS can issue a new MDU instruction one clock before the previous result is written. For example, back-to-back 32x32 multiply throughput is four clocks per multiply versus a five-clock multiply latency.

[Figure E-10](#) shows the execution pipeline for back-to-back multiplies in which adjacent instructions do not have a register dependency between them.

```

addog0, g1, g2
mulog2, g3, g4
mulog5, g6, g7
addog8, g9, g10

```

**Figure E-10. MDU Pipelined Back-To-Back Operations**

Instruction Scheduler	Issue	addo	mulo	---	---	---	mulo	addo	
EU Pipeline	Read <i>src1</i> , <i>src2</i>	g0, g1						g8, g9	
	Execute and Write <i>dst</i>		g2←g0+g1						g10←g8+g9
MDU Pipeline	Read <i>src1</i> , <i>src2</i>		g2, g3				g5, g6		
	Execute								
	Write <i>dst</i>							g4←g2*g3	

The MDU directly executes instructions listed in **Table E-6**. The scheduler issues an MDU instruction in one clock. The table also shows the length of the execution stage (latency) for each instruction. Subsequent instructions not dependent upon MDU results are issued and executed in parallel with the MDU. If instructions in the table are issued back-to-back and they have no register dependency between them, the MDU pipeline improves throughput by one clock per instruction.

**Table E-6. MDU Instructions**

Mnemonic	Issue Clocks	Result Latency	Back-to-Back Throughput (AC.com = 1)	Back-to-Back Throughput (AC.com = 0)
<b>muli</b> 32x32 16x32	1	5	4	5
	1	3	2	3
<b>mulo</b> 32x32 16x32	1	5	4	4
	1	3	2	3
<b>emul</b> 32x32 16x32	1	6	5	6
	1	3	2	3
divi	13	37	36	36
divo	3	36	35	35
ediv remi remo modi	3	36	35	35

### E.2.4.3. Data RAM (DR)

On-chip data RAM (DR), described in [Section 4.1, “Internal Data Ram” on page 4-1](#), is single-ported and 128-bits wide to support accesses of up to one quad-load or quad-store per clock. The DR receives instructions over the MEM-machine bus, stores addresses over the 32-bit Address Out bus and stores data over the 128-bit store bus. The DR returns data over the 128-bit load bus.

The one-clock DR pipeline for reads is shown in [Figure E-11](#). When the IS issues a load from the DR, load data is written to the destination register in the following clock.

An instruction which immediately follows a load from the DR and references the load destination cannot execute in the same clock as the load. As shown in the figure, the instruction is issued in the clock in which the load data is returning.

[Table E-7](#) lists the instructions executed directly in most addressing modes (without micro-flow execution) using the DR. As seen in [Figure E-11](#), if these instructions are issued back-to-back, they execute at a one-clock sustained rate, with or without register dependencies.

```
addo16, g0, g0
ldq (g0), g4
addog4, g5, g6
ldt (g7), g8
ldq (g8), g0
```

**Figure E-11. Data RAM Execution Pipeline**

Instruction Scheduler		Issue	addo	ldq	addo ldt	ldq
EU Pipeline	Read src1, src2	16, g0			g4, g5	
	Execute and Write dst				g0←g0+16	g6←g4+g5

**Table E-7. Data RAM Instructions**

Load Latency = 1 clock	Store Latency = 1 clock
<b>ld</b> <b>ldob</b> <b>ldib</b> <b>ldos</b> <b>ldis</b> <b>ldl</b> <b>ldt</b> <b>ldq</b>	<b>st</b> <b>stob</b> <b>stib</b> <b>stos</b> <b>stis</b> <b>stl</b> <b>stt</b> <b>stq</b>

#### E.2.4.4. Address Generation Unit (AGU)

The AGU contains a 32-bit parallel shifter-adder to speed memory address calculations. It also directly executes the **lda** instruction. The AGU receives instructions over the MEM-machine bus and offset and displacement values over the address out bus from the IS. The AGU reads the global and local registers over the 32-bit base bus register port and writes the registers over the 128-bit load bus.

The AGU calculates an effective address (*efa*) which is either written to a destination register in (the case of an **lida** instruction) or used as a memory address (in the case of loads, stores, extended branches or extended calls). When an **lida** instruction is issued, the AGU returns the *efa* to the destination register in the following clock for most addressing modes. An instruction which immediately follows the **lida** and references the **lida** destination is not issued in the same clock as the **lida**. As shown in [Figure E-12](#), it is issued in the clock in which **lida** is writing the destination register.

[Table E-8](#) lists the **lida** addressing mode combinations that the AGU executes directly. As seen in the figure, if **lida** instructions are issued back-to-back using one of the addressing modes in the table, the instructions execute at a one-clock sustained rate with or without register dependencies.

```

addo16, g0, g0
lida 16 (g0), g4
addog4, g5, g6
lida 16 [g7 * 4], g8
lida 16 (g8), g0
    
```

[Figure E-12. The lida Pipeline](#)

Instruction Scheduler	Issue	addo	lida	addo lida	lida	
EU Pipeline	Read <i>src1, src2</i>	16, g0		g4, g5		
	Execute and Write <i>dst</i>		$g0 \leftarrow g0 + 16$		$g6 \leftarrow g4 + g5$	
AGU Pipeline	Read over Base Bus		g0	g7	g8	
	Execute and Write over Ldbus			$g4 \leftarrow g0 + 16$	$g8 \leftarrow (g7 * 4) + 16$	$g0 \leftarrow g8 + 16$

[Table E-8. AGU Instructions](#)

Mnemonic	Issue Clocks	Addressing Mode	Result Latency Clocks
lida	1	offset disp (reg) offset(reg) disp(reg) disp[reg * scale]	1

### E.2.4.5. Effective Address (efa) Calculations

The AGU calculates the *efa* for instructions which require one. When the addressing mode specified by an instruction is the *offset*, *disp* or (*reg*) mode, the AGU generates the *efa* in parallel with the instruction's issuance. As shown in the previous pipeline figure for the DR (Figure E-11), load and store instructions begin immediately for these addressing modes with no delay for address generation. See [Section E.2.6, “Micro-flow Execution” on page E-29](#) for a description of how other addressing modes are handled.

### E.2.4.6. Bus Control Unit (BCU)

The BCU executes memory operations for load and store instructions, instruction fetches and micro-flows. It executes memory load requests in two clocks (zero wait states) and returns a result on the third clock. Using address pipelining and on-chip request queuing, the BCU can accept a load or store from the IS every clock and return load data every clock. The BCU receives instructions over the MEM-machine bus, stores addresses over the 32-bit address out bus and stores data over the 128-bit store bus. The BCU returns data over the 128-bit load bus.

The BCU receives a load address during the “issue” clock. The address is placed on the system bus during the next clock (the first BCU execute stage). The system returns data at the end of the following clock (the second BCU execute stage). On the next clock the BCU writes the data to the destination register. This write is bypassed to the REG-side and MEM-side source buses and the scoreboarded instruction is issued in the same clock.

The zero wait state load causes a two clock execution delay of the next instruction because the load data is referenced immediately after the load is issued. If the memory system has wait states, the load data delay will be longer. If the load is advanced in the code such that it is separated from the instruction which uses the data, the load delay can be completely overlapped with the execution of other instructions.

Store instruction execution would proceed as does the load, except that there would be no return clock and no instructions could be stalled due to a scoreboarded register.

[Table E-9](#) lists instructions that the i960 Hx processor’s BCU executes directly. For each instruction that requires multiple reads (such as **ldq**) the BCU buffers the return data until all data is returned. This optimization reduces the internal load bus overhead to the minimum, giving more clocks to the processor to access the DR and perform **lda** operations while external loads are in progress. The table is valid when offset, displacement or indirect memory addressing modes are used over an external bus with the following characteristics:

$$N_{XAD} = N_{XDD} = N_{XDA} = 0, \text{Burst On, Pipelining On, Ready Disabled}$$

For other addressing modes, see [Section E.2.6, “Micro-flow Execution” on page E-29](#).

If instructions listed in the table are issued back-to-back with no register dependencies, they will execute at a rate of one instruction per clock until the BCU queues are full. Once the queues are full, further back-to-back BCU instructions execute at the bus bandwidth. [Figure E-13](#) shows back-to-back loads being executed.

**Table E-9. BCU Instructions**

Mnemonic	Issue Clocks	Result Latency Clocks	Back-to-Back Throughput
<b>ld</b> <b>ldob</b> <b>ldib</b> <b>ldos</b> <b>ldis</b>	1	3	1
<b>ldr</b>	1	4	2
<b>ldt</b>	1	5	3
<b>ldq</b>	1	6	4
<b>st</b> <b>stob</b> <b>stib</b> <b>stos</b> <b>stis</b>	1	N/A	2
<b>stl</b>	1	N/A	3
<b>stt</b>	1	N/A	4
<b>stq</b>	1	N/A	5

To allow programs to issue load requests before the data is needed — and thus decouple memory speeds from instruction execution — the BCU contains four queue entries. Each entry stores all the information needed for a memory request:

- For loads, the BCU contains the source address, destination register number and load type
- For stores, BCU contains the destination address, store type and the store data

If a **stq** is executed, all four registers are written to the BCU queue in one clock. The BCU performs the actual bus request without taking any further clocks from instruction execution. BCU queues maintain memory requests in order. The requests are executed on the bus in the order that they are issued from the instruction stream.

```
ld (g0), g1
ld (g2), g3
ld (g4), g5
addog1, g6, g7
```

**Figure E-13. Back-to-Back BCU Accesses**

Instruction Scheduler	Issue	Id	Id	Id	addo			
BCU Pipeline	Address Out bus St bus	g0	g2	g4				
	External Address Bus		g0	g2	g4			
	External Data Bus			(g0)	(g2)	(g4)		
	LD Bus				g1←(g0)	g3←(g2)	g5←(g4)	
EU Pipeline	Read <i>src1, src2</i>				g1, g6			
	Execute and Write <i>dst</i>					g7←g1+g6		

### E.2.4.7. Control Pipeline

The IS directly executes program flow control instructions. Branches take two clocks to execute in the CTRL pipeline; however, the IS is able to see branches as many as four instructions ahead of the current instruction pointer. This allows the scheduler to issue the branch early and, in most cases, execute the branch without inserting a dead clock in the REG and MEM instruction streams.

[Table E-10](#) lists the instructions that the IS executes directly, without the aid of micro-flows. For information on other control flow instructions, see [Section E.2.6, “Micro-flow Execution”](#) on page [E-29](#).

### E.2.4.8. Unconditional Branches

[Figure E-15](#) shows the IS issue stage and the CTRL pipeline for the case where the branch target is another branch, disabling the IS’s ability to look ahead. The IS issues the branch in one clock; the branch is executed in the next clock. The branch target is another branch, which the scheduler issues immediately. Hence, branch instructions have a two-clock sustained rate when issued back-to-back.

**Table E-10. CTRL Instructions**

Mnemonic	Issue Clocks	Latency Clocks	Back-to-Back Throughput Clocks
<b>be bne bl ble bg bge bo bno</b>	1	2	2

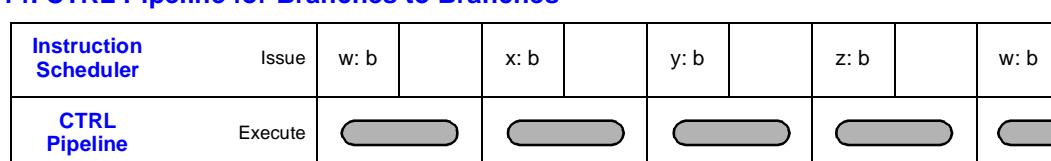
w: b x

...  
x: b y

...  
y: b z

...  
z: b w

**Figure E-14. CTRL Pipeline for Branches to Branches**



An *executable group* of instructions is a group of sequential instructions in the currently visible quad word which can be issued in the same clock. See [Section E.2, “Parallel Instruction Processing”](#) on page [E-11](#).

[Figure E-15](#) shows the cases where a branch, when first seen by the IS, is in the first executable group of instructions. The IS issues the branch immediately, along with the first one (or two) instruction(s) ahead of it. Since the branch takes two clocks in the CTRL pipeline to execute, a one-clock break in the IS's ability to issue instructions occurs. On the next clock, the IS issues a new group of instructions from the branch target.

In the figure, two other instructions were issued simultaneously with the branch. Hence, the branch could be said to have taken one clock to execute. When the branch is the first instruction in the group (the branch is a branch target) no other instructions are issued in parallel with the branch and it takes a full two clocks to execute (as seen in [Figure E-15](#)).

```

        b      x
        ...
x: addo   g0, g1, g2
    lda    2(g3), g4
    b      y
    ...
y: addo   g5, g6, g7
    lda    2(g8), g9
    
```

[Figure E-15. Branch in First Executable Group](#)

Instruction Scheduler	Issue		addo lda b	—	addo lda			
CTRL Pipeline	Execute							
EU Pipeline	Read src1, src2		g0, g1		g5, g6			
	Execute and Write dst			g2←g0+g1		g7←g5+g6		
AGU Pipeline	Read over Base Bus		g3		g8			
	Execute and Write over Ldbus			g4←2+g3		g9←g8+2		

[Figure E-16](#) shows the case where a branch, when first seen by the IS, is in the second executable group (B) of instructions in the rolling quad word, not the first executable group (A) which is about to be issued. The IS issues the branch immediately, along with the first group of instructions ahead of it (A). Since the branch takes two clocks in the CTRL pipeline to execute, there is no break in the IS's ability to issue instructions. On the next clock, the IS issues a new group of instructions from the branch target.

In the figure, two other instructions were issued simultaneously with the branch and one instruction was issued during the clock in which the branch was executing. Hence, it can be said that this branch takes zero clocks to execute.

[Figure E-17](#) shows the case where a branch, when first seen by the IS, is in the third executable group (C) of instructions of the rolling quad word, not the first executable group (A) which is about to be issued. The IS issues group A, then issues the branch and group B simultaneously. Since the branch takes two clocks in the CTRL pipeline to execute, there is no break in the IS's ability to issue instructions. On the clock following the issuance of group B, the IS issues a new group of instructions from the branch target.

```

      b      x
      ...
x: addo  g0, g1, g2  A
    lda   2(g3), g4  A
    lda   2(g5), g6  B
    b     Y          A
y: ...
    addo  g7, g8, g9
    lda   2(g10), g11
  
```

**Figure E-16. Branch in Second Executable Group**

		Group:	A	B				
Instruction Scheduler	Issue		addo lda b	lda	addo lda			
CTRL Pipeline	Execute	[ ]		[ ]				
EU Pipeline	Read src1, src2		g0, g1		g7, g8			
	Execute and Write dst			g2←g0+g1		g9←g7+g8		
AGU Pipeline	Read over Base Bus		g3	g5	g10			
	Execute and Write over Ldbus			g4←g3+2	g6←g5+2	g11←g10+2		

```

      b      x
      ...
x: lda   2(g3), g4  A
    addo g0, g1, g2  B
    addo g5, g6, g7  C
    b     Y          A
      ...
y: addo  g8, g9, g10
    lda   2(g11), g12
  
```

**Figure E-17. Branch in Third Executable Group**

		Group:	A	B	C			
Instruction Scheduler	Issue		lda	addo b	addo	addo lda		
CTRL Pipeline	Execute	[ ]		[ ]				
EU Pipeline	Read src1, src2			g0, g1	g5, g6	g8, g9		
	Execute and Write dst				g2←g0+g1	g7←g5+g6	g10←g8+g9	
AGU Pipeline	Read over Base Bus		g3			g11		
	Execute and Write over Ldbus			g4←g3+2			g12←g11+2	

### E.2.4.9. Conditional Branches

When the IS sees a conditional branch instruction, the condition codes are sometimes not yet determined. For example, a conditional branch which immediately follows a compare instruction cannot be allowed to complete execution until the result of the comparison is known. However, the processor begins to execute the branch based upon the branch prediction bit set by the programmer for that branch.

When one or more executable instruction groups separate the conditional instruction from the instruction that changed the condition code, the condition code will have already settled in the pipeline by the time the prefetch mechanism sees the conditional instruction. This situation allows the branch to execute in zero clock cycles, as described in [Figure E-17](#).

If the conditional instruction and the instruction that sets the condition codes are in the same executable group or in consecutive groups, the condition code is not valid when the IS sees the branch; a guess is required. If the prediction turns out to be correct, the branch executes in its normal amount of time, as described in the previous section. If the prediction is wrong, the pipeline is flushed. Any erroneously started single- or multiple-cycle instructions are killed and the branch executes as if there had been no lookahead or prediction. In other words:

- the branch takes two clocks out of the IS's issue stage if it is in the same executable group as the instruction which modified the condition codes; or
- the branch takes one clock if it is in the executable group adjacent to the group that modifies the condition codes.

## E.2.5 Instruction Cache and Fetch Execution

The instruction cache provides four consecutive opcode words to the IS on every clock. This capability allows the processor to dispatch instructions from the processor's sequential instruction stream to multiple independent parallel processing units. When a cache miss occurs or is about to occur, the Instruction Fetch Unit issues instruction fetch requests to the BCU.

### E.2.5.1. Instruction Cache Organization

On every clock, the cache accesses one or two lines and multiplexes the correct four words to the IS.

The i960 Hx processor's instruction cache supports pre-loading and locking 0 to 4 ways of the instruction cache. Each way is 4 Kbytes and any contiguous section of code can be locked in a way.

The instruction scheduler checks all ways of the cache for every instruction fetched. If an instruction is not found, it is fetched from external memory and loaded into the unlocked portion of the instruction cache.

### E.2.5.2. Fetch Strategy

When any of the four words presented to the scheduler are invalid, a cache miss is signaled and an instruction fetch is issued. The Instruction Fetch Unit makes the fetch and prefetch decisions.

Since the cache supports two word and quad word replacement within a line, instruction fetches can be issued in either size. The conditions of the cache miss determine which fetch is issued. [Table E-11](#) describes the fetch decision.

**Table E-11. Fetch Strategy**

Words Provided To Scheduler				Fetch Initiated	
IP	IP+4	IP+8	IP+12	A3:2 of requested IP = 0X <sub>2</sub>	A3:2 of requested IP = 1X <sub>2</sub>
Hit	Hit	Hit	Hit	no fetch	no fetch
Hit Miss Miss	Miss Hit Miss	Hit Hit Hit	Hit Hit Hit	fetch two words at IP	fetch two words at IP
Hit Hit Hit	Hit Hit Hit	Hit Miss Miss	Miss Hit Miss	fetch two words at IP+8	fetch two words at IP+8
All other cases				fetch four words at IP	fetch two words at IP and four words at IP+8

### E.2.5.3. Fetch Latency

The Instruction Fetch Unit initiates an instruction fetch by requesting quad word or long word loads from the BCU. These fetches differ from actual instruction stream loads in two ways: load destination and load data buffering.

First, the load destination of an instruction fetch is the instruction fetch buffer, not the register file. Since fetch data goes directly from the BCU to the instruction fetch buffer and IS, the scheduler can issue fetched instructions during the clock after they are read from external memory.

Second, to reduce fetch latency, the BCU buffers fetch data differently than a regular load instruction. Instead of buffering four words of instructions before sending data to the fetch unit, the BCU sends each word as it is received over the bus. If the fetches are from 8- or 16-bit memory, the BCU collects 32 bits before sending the word to each fetch unit.

If the fetch request is the result of a prefetch decision, the IS is not stalled unless it needs an instruction from the prefetch request.

If the processor is executing straight-line code which always misses the cache, the IS is only able to issue instructions at a one-instruction-per-clock rate. It is never able to see multiple instructions in one clock. The bus bandwidth of the memory subsystem containing the code limits the application's performance.

```

      b          y
      ...
y:  addo    g0, g1, g2← Cache Miss
    subo    g3, g4, g5
  
```

**Figure E-18. Fetch Execution**

Instruction Scheduler	Issue		y: ---	---	---	---	addo	subo		
<b>CTRL Pipeline</b>	Execute		Cache Miss							
<b>BCU Pipeline</b>	Address Out bus St bus			Fetch Miss						
	External Address Bus				A					
	External Data Bus					D addo	D subo			
	Ld Bus						D addo	D subo		
<b>EU Pipeline</b>	Read <i>src1</i> , <i>src2</i>						g0, g1	g3, g4		
	Execute and Write <i>dst</i>							g2←g0+g1	g5←g4-g3	

#### E.2.5.4. Cache Replacement

Data fetched as a result of a cache miss is always written to the cache.

### E.2.6 Micro-flow Execution

The i960 Hx processor's parallel processing units directly execute about half of the processor's instructions. The processor services the remaining complex instructions by executing a sequence of simple instructions from an on-chip ROM. Complex instructions are detected in the clock in which they are fetched. This information becomes part of the instruction encoding stored in the instruction fetch unit queue and/or instruction cache.

Micro-flow instruction sequences are written to enable the parallel processing units to perform the required function as fast as possible. Micro-flows use instructions described in prior sections of this appendix (machine types REG, MEM and CTRL) and some special parallel circuitry to carry out the complex instructions. An instruction which cannot be directly issued to a parallel processing unit is said to have the machine type  $\mu$ .

#### E.2.6.1. Invocation and Execution

Invoking a micro-flow can be considered analogous to the processor's execution of an unconditional branch into the on-chip ROM. However, pre-decoding and optimized lookahead logic make the micro-flow invocation more efficient than a branch instruction.

While the IS is issuing one group of instructions, parallel decode circuitry checks to see if the *next* executable instruction is a  $\mu$  instruction (Figure E-19). If so, the opcode words presented to the IS in the *next* clock come from the on-chip ROM location that contains the micro-flow for the detected complex instruction. The IS actually never attempts to issue a complex encoding. The processor detects the encoding when the instruction is fetched, then traps during the clock in which the instruction is presented to the IS.

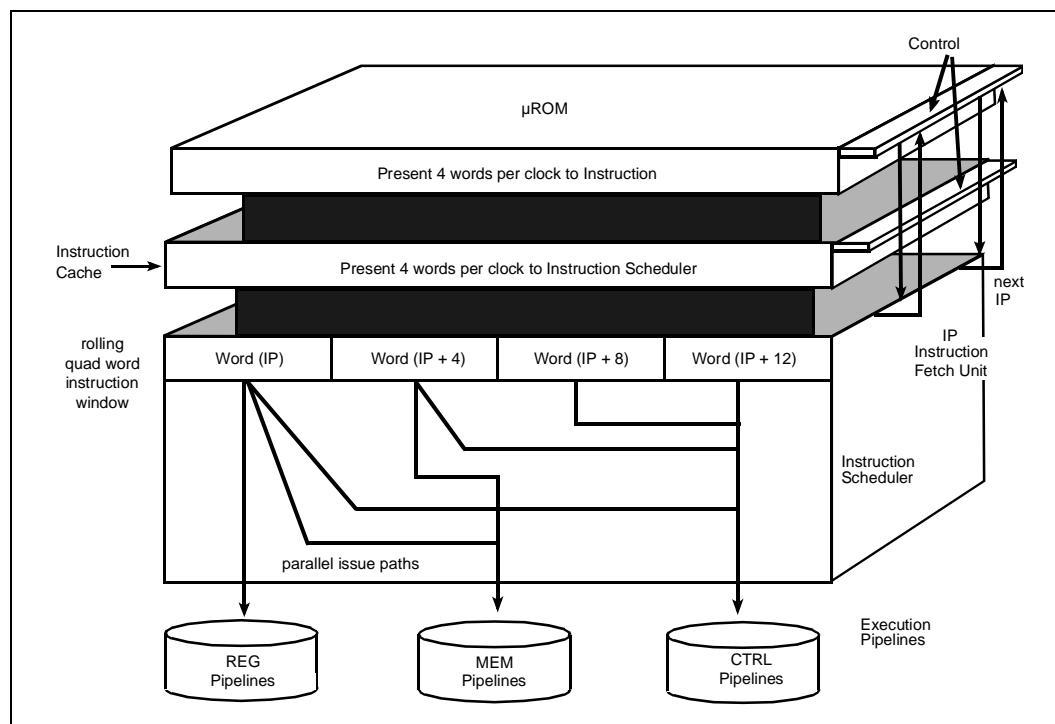
Generally, no clocks are lost when switching to a micro-flow. However, two conditions can defeat the lookahead logic:

- branches to REG-, CTRL- or COBR-format instructions which are implemented as micro-flows ( $\mu$ ); or
- cache misses from straight-line code execution.

Under these conditions, the switch to on-chip ROM causes a one-clock break in the IS's ability to issue instructions.

Complex instructions encoded with the MEM-format do not require lookahead detection to trap to the ROM without overhead. Therefore, MEM-format instructions of machine type  $\mu$  do not see a one-clock performance loss even when lookahead logic is defeated. Furthermore, micro-flows return to general execution with no overhead; back-to-back micro-flows do not incur the one-clock defeated lookahead penalty.

**Figure E-19. Micro-Flow Invocation**



When micro-flows execute, they consume the instruction scheduler's activity. From the first clock through the last clock of a micro-flow, the IS is typically issuing two instructions per clock. MEM-side micro-flows — such as loads and stores — can be issued in parallel with REG-side instructions. Performance of micro-flowed instructions is measured by the number of clocks taken to issue instructions.

## E.2.6.2. Data Movement

Data movement instructions supported as micro-flows include the triple and quad word register move instructions and the **lda**, load and store instructions which use complex addressing modes.

**movt** and **movq** each take two clocks to execute.

Load and store instructions are summarized in [Table E-14](#) and [Table E-15](#). The number of clocks shown is the *additional* number of issue clocks consumed for address calculation prior to the load or store being issued to the BCU or DR. These instructions can be issued in parallel with a machine type REG instruction. To find the result latency of the BCU or DR, see the appropriate section earlier in this appendix.

**Table E-12. Load Micro-Flow Instruction Issue Clocks**

The following load instructions consume <b>n</b> additional issue clocks for address calculation before initiating a load request to the BCU or DR, where <b>n</b> for each addressing mode is as follows:			
<b>Mnemonic</b>	disp(reg) offset(reg) disp[reg * scale]	(reg)[reg * scale] disp(reg)[reg * scale]	disp(IP)
<b>ld, ldob, ldib, ldos, ldis, ldl, ldt, ldq</b>	1	2	4

NOTE: *offset, disp* and *(reg)* memory addressing modes incur no address calculation overhead.

## E.2.6.3. Bit and Bit Field

**scanbit**, **spanbit**, **extract** and **modify** are executed as micro-flows. [Table E-14](#) lists their execution times. For these instructions, the IS issues **n** clocks of instructions in place of the single word i960 Hx processor instruction encoding, where **n** is shown in the table.

**Table E-13. Store Micro-flow Instruction Issue Clocks**

The following store instructions consume <b>n</b> additional issue clocks for address calculation prior to initiating a store request to the BCU or DR, where <b>n</b> for each addressing mode is as follows:			
<b>Mnemonic</b>	disp(reg) offset(reg) disp[reg * scale]	(reg)[reg * scale] disp(reg)[reg * scale]	disp(IP)
<b>st, stob, stib, stos, stis, stl, stt, stq</b>	1	2	4

NOTE: *offset, disp* and *(reg)* memory addressing modes incur no address calculation overhead.

**Table E-14. Bit and Bit Field Micro-flow Instructions**

<b>Mnemonic</b>	<b>Execution Clocks (n)</b>
<b>scanbit</b>	1
<b>spanbit</b>	3
<b>extract</b>	4
<b>modify</b>	3

#### E.2.6.4. Comparison

**test\*** instructions are executed as micro-flows. Execution time depends upon condition code validity and prediction bit settings. When condition codes are valid or the prediction bit is set correctly, a **test\*** instruction takes 10 issue clocks if its correct result is a 1, and 12 issue clocks if its correct result is a 0.

#### E.2.6.5. Branch

Compare and branch, extended branch, branch and link and extended branch and link instructions are implemented with micro-flows.

**cmpib\*** and **cmpob\*** instructions take one issue clock if the prediction was correct and two issue clocks if the prediction was incorrect, assuming a cached branch target.

**bal** takes two issue clocks to execute, assuming a cache hit.

**bx** and **balx** are summarized in [Table E-15](#). The number of clocks shown is the total number of issue clocks consumed by the instruction prior to the code at the branch target being issued. Times shown assume instruction cache hits and a DR-based link target. These instructions may be issued in parallel with a machine-type R instruction.

[Table E-15. bx and balx Performance](#)

The following instructions consume <b>n</b> issue clocks before target code is issued, where <b>n</b> for each addressing mode is as follows:			
Mnemonic	disp offset (reg) disp(reg) offset(reg) disp[reg * scale]	(reg)[reg * scale] disp(reg)[reg * scale]	disp(IP)
<b>bx, balx</b>	4	4	6

#### E.2.6.6. Call and Return

Procedure call, return and system procedure call instructions are implemented as micro-flows. **call** consumes four issue clocks when the target is cached and a register cache location is available. When a frame spill is required, an additional 22 issue clocks are consumed in a zero-wait-state system before the target code begins execution. The worst-case memory activity for a call with a frame spill and a cache miss is one quad word instruction fetch followed by four quad word stores. Wait states in the instruction fetch directly impact call speed, while wait states in the frame stores are decoupled from internal execution by the BCU queues.

**ret** consumes four issue clocks when the target and the previous register set are both cached. When a frame fill is required, an additional 38 issue clocks are consumed in a zero-wait-state system before the target code begins execution. The worst-case memory activity for a return with a frame fill and a cache miss is four quad word reads followed by one quad word fetch. Wait states in the instruction fetch or the frame fill directly impact return speed.

**calls** consumes up to 56 issue clocks if the call is to a supervisor procedure. If the call is to a non-supervisor procedure, **calls** takes 38 issue clocks. These times assume an available register cache location and a cached target. During **calls** execution, the processor accesses the system procedure table with a single word read and a long word read. The presence of several wait states in these reads directly affects the instruction's performance. The impact of non-cached target code or a frame spill on the **calls** instruction is identical to the impact on the **call** instruction.

**callx** timing is similar to **call** instruction timing with the exception of issue clocks. [Table E-16](#) shows total issue clocks for **callx**.

**Table E-16. callx Performance**

The following instruction consumes <b>n</b> issue clocks before target code is issued, where <b>n</b> for each addressing mode is as follows:			
Mnemonic	disp offset (reg) disp(reg) offset(reg) disp[reg * scale]	(reg)[reg * scale] disp(reg)[reg * scale]	disp(IP)
<b>callx</b>	7	9	9

Times shown assume instruction cache hits.

### E.2.6.7. Conditional Faults

**fault\*** instructions are implemented with micro-flows and require one issue clock if the prediction bit is correct and no fault occurs. If the prediction bit is incorrect and no fault occurs, the instructions require two issue clocks. The time it takes to enter a fault handler varies greatly depending upon the state of the processor's parallel processing units.

### E.2.6.8. Debug

**mark** and **fmark** are implemented with micro-flows. **mark** takes one issue clock if no trace fault is signaled. If a trace fault is signaled or **fmark** is executed, the processor performs an implicit call to the trace fault handler. As with conditional faults, the time required to enter a fault handler varies greatly.

### E.2.6.9. Atomic

Atomic instructions are implemented with micro-flows. **atadd** takes seven issue clocks and **atmod** takes eight issue clocks to execute with an idle bus in a zero-wait state system. Memory wait states directly affect execution speed.

### E.2.6.10. Processor Management

Processor management instructions implemented as micro-flows include: **modpc**, **modac**, **modtc**, **syncf**, **flushreg** and **sysctl**.

<b>modpc</b>	requires 17 clocks to execute if process priority is changed and 12 clocks if process priority is not changed.
<b>modac</b>	requires 9 clocks.
<b>modtc</b>	requires 15 clocks.
<b>syncf</b>	takes 4 issue clocks if there are no possible outstanding faults. Otherwise, the instruction locks the IS until it is certain that no prior instruction will fault.
<b>flushreg</b>	requires 24 clocks for each frame that is flushed. This translates to 120 cycles to flush five frames. Wait states in the memory being written affect this instruction's performance.
<b>sysctl</b>	Timings shown in <a href="#">Table E-17</a> assume a zero wait-state memory system.

## E.2.7 Coding Optimizations

[Table E-17. sysctl Performance](#)

Message	Message Type	Issue Clocks
Request Interrupt	00H	20 + bus wait states
Invalidate Instruction Cache	01H	19
Configure Instruction Cache	02H	20 to enable I-cache 22 to disable I-cache TBD+ bus wait states to load and lock 1 way (4 Kbytes)
Reinitialize	03H	329 + bus wait states
Load Control Register Group	04H	39 to 51+ bus wait states
Modify Memory-Mapped Control Register (MMR)	05H	26 + bus wait states
Breakpoint Resource Request	06H	21 - 22

Embedded applications often benefit from hand-optimized interrupt handlers and critical primitives. This section reviews coding optimizations which arise due to the micro-architecture of the i960 Hx instruction set processor. The examples in this section are constructed to illustrate particular optimization techniques. In general, every example could be further optimized by applying several techniques instead of one.

### E.2.7.1. Loads and Stores

Separate load instructions from instructions that use load data. Remember that store instructions can also be reordered. Although it returns no results to a register, a poorly placed store in front of a critical load slows down the load. Reorder to issue the load first. [Example E-1](#) shows a simple change that saved one clock from a five-clock loop.

#### [Example E-1. Overlapping Loads \(Checksum\)](#)

loop:	opt_loop:
ldob (g0), g1	ldob (g0), g1
addo g1, g2, g2	cmpinco g0, g3, g3
cmpinco g0, g3, g3	addo g1, g2, g2
bl.t loop	bl.t opt_loop

Execution:

Clock	REGop	MEMop	CTRLop
1		ldob	
2		:	
3		:	
4	addo		bl.t
5	cmpinco		:
6		ldob	

Execution:

Clock	REGop	MEMop	CTRLop
1		ldob	
2	cmpinco	:	
3		:	bl.t
4	addo		:
5		ldob	

### E.2.7.2. Multiplication and Division

Begin multiply and divide instructions several cycles before instructions that use their results. MDU instructions consume less than one clock if they are sufficiently separated from the instructions that use their results. Also, use shift instructions to replace multiplication and division by powers of two. [Example E-2](#) shows overlapping pointer math and a comparison with the 32x32 multiply time in a simple multiply-accumulate loop.

### Example E-2. Overlapping MDU Operations (Multiply Accumulate)

<b>loop:</b> ld (g0), g2 ld (g1), g3 muli g2, g3, g4 addi g4, g5, g5 addo 4, g0, g0 addo 4, g1, g1 cmpobl.t g0, g6, loop	<b>opt_loop:</b> ld (g0), g2 ld (g1), g3 muli g2, g3, g4 addo 4, g0, g0 cmpo g0, g6 addo 4, g1, g1 addi g4, g5, g5 bl.t opt_loop
---	--

Execution (from DR):

Clock	REGop	MEMop	CTRLop
1		ld	
2		ld	
3	muli		
4	:		
5	:		
6	:		
7	:		
8	addi		
9	addo		
10	addo		bl.t
11	cmpo		:
12		ld	

Execution (from DR):

Clock	REGop	MEMop	CTRLop
1			ld
2			ld
3	muli		
4	:	addo	
5	:	cmpo	
6	:	addo	
7	:		bl.t
8		addi	:
9			ld

#### E.2.7.3. Advancing Comparisons

Where possible, instructions which change condition codes should be separated from instructions that use condition codes. Although correct branch prediction gives the same performance as separating the compare from the branch, prediction is statistical while separation is deterministic. In the previous example, optimized code advanced the comparison enough that branch prediction is not being relied upon to keep the branch-true path executing at nine clocks. Furthermore, the branch-false path does not take extra clocks since the condition codes are known when the branch is encountered.

In a situation where the comparison and a branch cannot be separated to achieve a performance advantage, use the combined compare and branch instructions. This is likely to lead to faster execution since the two instructions are encoded in a single word. This code economy frees another location in the cache and the IS may be able to see the branch earlier because the branch is encoded in the same opcode word.

#### E.2.7.4. Unrolling Loops

Expand small loops into larger loops which fill the cache, use more registers and pipeline their memory operations. The strategy is to begin accessing the memory system as soon as the routine is entered and to make the best use of the bus. Less bus bandwidth is used for the same operations if the algorithm is implemented with quad loads and/or stores.

The large register set allows an unrolled loop to have multiple sets of working temporary registers for operations in various stages. For example, the previous checksum example is repeated in [Example E-3](#). The loop is unrolled to perform checksums nearly twice as fast as the simple loop.

##### [Example E-3. Unrolling Loops \(Checksum\)](#)

```
-- initialize --
loop:
    ldob          (g0), g1
    addo          g1, g2, g2
    cmpinco      g0, g3, g3
    bl.t          loop
    ret

-- initialize --
opt_loop:
    ldob          (g0), g1
    cmpinco      g0, g3, g3
    addo          g4, g2, g2
    bge.f         exit1
    ldob          (g0), g4
    cmpinco      g0, g3, g3
    addo          g1, g2, g2
    bl.t          opt_loop
    exit2:
    addo          g4, g2, g2
    ret
    exit1:
    addo          g1,g2,g2
    ret
```

Execution:

Clock	REGop	MEMop	CTRLop
1		ldob	
2		:	
3		:	
4	addo		bl.t
5	cmpinco		:
6		ldob	

Execution:

Clock	REGop	MEMop	CTRLop
1		ldob g1	
2	cmpinco	:	bge.f
3	addo g4	:	:
4		ldob g4	
5	cmpinco	:	bl.t
6	addo g1	:	:
7		ldob g1	

### E.2.7.5. Enabling Constant Parallel Issue

As described in [Section E.2.1, “Parallel Issue”](#) on page E-12, certain sequences of machine-type instructions can be executed in parallel, such as REG-MEM, REG-MEM-CTRL, MEM-CTRL. In [Example E-4](#) the checksum loop is repeated. Another clock is eliminated by reordering code for parallel issue.

#### Example E-4. Order for Parallelism (Checksum)

```
-- initialize --
loop:
    ldob          (g0), g1
    addo          g1, g2, g2
    cmpinco      g0, g3, g3
    bl.t          loop
    ret

-- initialize --
opt_loop:
    addo          g4, g2, g2
    ldob          (g0), g1
    cmpinco      g0, g3, g3
    bge.f         exit1
    ldob          (g0), g4
    cmpinco      g0, g3, g3
    addo          g1, g2, g2
    bl.t          opt_loop
exit2:
    addo          g4, g2, g2
    ret
exit1:
    addo          g1,g2,g2
    ret
```

Execution:

Clock	REGop	MEMop	CTRLop
1		ldob	
2		:	
3		:	
4	addo		bl.t
5	cmpinco		:
6		ldob	

Execution:

Clock	REGop	MEMop	CTRLop
1	addo g4	ldob g1	bge.f
2	cmpinco	:	:
3		ldob g4	
4	cmpinco	:	bl.t
5	addo g1	:	:
6	addo g4	ldob g1	

### E.2.7.6. Alternating from Side to Side

The i960 Hx processor can sustain execution of two instructions per clock. To maximize this capability, try to start instructions in two of the three pipelines each clock. To increase parallelism, move an instruction from a unit which has become a critical path to a unit with available clocks. The AGU performs shifts, additions and moves that can replace EU operations. Literal addressing mode, in combination with EU or AGU operations, provides some freedom in deciding which side loads constants into registers. Remember to use addressing modes that the AGU executes directly (machine type M, not  $\mu$ ).

[Table E-18](#) lists several conversions that can move an instruction to the AGU from either the EU or MDU. [Example E-5](#) exploits the **lda** instruction to increase a 3x3 lowpass filter’s performance by approximately 30 percent.

**Table E-18. Creative Uses for the lda Instruction**

Operation	Equivalent lda instruction
addo 5, g0, g1# constant addition	lda 5(g0), g1
shlo 2, g1, g2# shifts by a constant	lda [ g1 * 4], g2
mov 31, g0# constant load	lda 31, g0
shlo 2, g1, g2 # shift/add combination addo 5, g2, g2	lda 5[ g1 * 4], g2
mov g0, g1# register move	lda (g0), g1

**Example E-5. Change the Type of Instruction Used (3x3 Lowpass Mask)**

$$Y[ ] = X[ ] * M[ ]$$

$$M[ ] = \begin{bmatrix} \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \\ \frac{2}{16} & \frac{4}{16} & \frac{2}{16} \\ \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \end{bmatrix}$$

```

# initial values                                # initial values
# g0 points to X(0,0)                          # g0 points to X(0,0)
# g1 points to Y(1,1)                          # g1 points to Y(1,0)
# g2 contains imax                           # g2 contains imax
# r4 load temp                               # r4 load temp
# r5 accumulator                            # r5 accumulator
# r6 = imax (i count temp)                  # r6 = imax (i count temp)
# r7 = jmax (j count temp)                  # r7 = jmax (j count temp)
# r8 = imax-1                                 # r8 = imax-1
# (new mask row offset)                      # (new mask row offset)
# r9 = 2*imax - 2                           # r9 = 2*imax - 2
# (new i offset)                             # (new i offset)
# r10 is 2*imax + 1                         # r10 is 2*imax + 1
# (new j offset)                             # (new j offset)
b next_j:
next_i:
    subor9, g0, g0
next_j:
# first mask row
    ldob(g0), r5
    addo1, g1, g1
    ldob(g0), r5
    addo1, g0, g0
    ldob(g0), r4
    addo1, g0, g0
    shlo1, r4, r4
    addor4, r5, r5
    lda [r4 * 2], r4
    addor4, r5, r5
    new_next_i:
    new_next_j:

```

```

ldob(g0), r4
addor4, r5, r5
addor8, g0, g0
# second mask row
ldob(g0), r4
addol, g0, g0
shlo1, r4, r4
addor4, r5, r5
ldob(g0), r4
addol, g0, g0
shlo2, r4, r4
addor4, r5, r5
ldob(g0), r4
shlo1, r4, r4
addor4, r5, r5
addor8, g0, g0
# third mask row
ldob(g0), r4
addol, g0, g0
addor4, r5, r5
ldob(g0), r4
addol, g0, g0
shlo1, r4, r4
addor4, r5, r5
ldob(g0), r4
addor4, r5, r5
shro4, r5, r5
stobr5, (g1)
addol, g1, g1

# update pointers
cmpdeco2, r6, r6
bg next_i
mov g2, r6
cmpdeco2, r7, r7
subor10, g0, g0
addo2, g1, g1
bg next_j
ret

Execution from DR (new loop):

```

```

ldob(g0), r4
addor4, r5, r5
addor8, g0, g0
# second mask row
ldob(g0), r4
addol, g0, g0
addor4, r5, r5
lda [r4 * 2], r4
ldob(g0), r4
addol, g0, g0
lda [r4 * 4], r4
addor4, r5, r5
ldob(g0), r4
addor8, g0, g0
lda [r4 * 2], r4
addor4, r5, r5
# third mask row
ldob(g0), r4
addol, g0, g0
addor4, r5, r5
ldob(g0), r4
addol, g0, g0
lda [r4 * 2], r4
addor4, r5, r5
ldob(g0), r4
addor4, r5, r5
shro4, r5, r5
cmpdeco2, r6, r6
stobr5, (g1)
subor9, g0, g0

# update pointers
bg.tnew_next_i
addor9, g0, g0
lda (g2), r6
cmpdeco2, r7, r7
lda 2(g1), g1
subor10, g0, g0
bg.tnew_next_j
ret

Execution from DR (loop):

```

Clock	REGop	MEMop	CTRLop
1	subo		
2		ldob	
3	addo		
4		ldob	
5	addo		
6	shlo		

Clock	REGop	MEMop	CTRLop
1	addo	ldob	
2	addo		
3		ldob	
4	addo	lda	
5	addo	ldob	
6	addo		

<b>Clock</b>	<b>REGop</b>	<b>MEMop</b>	<b>CTRLop</b>	<b>Clock</b>	<b>REGop</b>	<b>MEMop</b>	<b>CTRLop</b>
7	addo			7	addo	ldob	
8		ldob		8	addo	lda	
9	addo			9	addo	ldob	
10	addo			10	addo	lda	
11		ldob		11	addo	ldob	
12	addo			12	addo	lda	
13	shlo			13	addo	ldob	
14	addo			14	addo		
15		ldob		15	addo	ldob	
16	addo			16	addo	lda	
17	shlo			17	addo	ldob	
18	addo			18	addo		
19		ldob		19	shro		
20	shlo			20	cmpdeco	stob	bg.t
21	addo			21	subo		:
22	addo			22	addo	ldob	
23		ldob					
24	addo						
25	addo						
26		ldob					
27	addo						
28	shlo						
29	addo						
30		ldob					
31	addo						
32	shro						
33		stob					
34	addo		bg.t				
35	cmpdeco		:				
36	subo						

### E.2.7.7. Branch Prediction

Conditional branches execute faster if the branch direction is correctly predicted using the branch prediction bits on conditional instructions. This is particularly true when a comparison cannot be separated from the test in a conditional instruction. When the prediction is correct, branches generally execute in parallel with other execution. If prediction is not correct, the worst case branch time for cached execution is still two clocks.

Although prediction bits are most likely set to gain maximum throughput, different strategies can be used for setting the prediction bits. A code sequence dominated by comparisons and conditional branches might see large differences between execution time of the fastest path and slowest path. Prediction bits can be set to provide the best average throughput to ensure the fastest worst case execution or to minimize deviation between slowest and fastest times.

### E.2.7.8. Branch Target Alignment

Branch target code executes with more parallelism in the first clock if the branch target is long word or quad word aligned. Quad word alignment is preferable for prefetch efficiency.

The IS sees four words in a clock when the requested IP is long word aligned and three words when the requested IP is not on a long word boundary. Aligned branch targets give the scheduler another word to examine on the first clock following a branch. However, there are only a few cases where this optimization pays off.

The IS takes advantage of seeing four words on the first clock after a branch when the fourth word is a branch or micro-flow and all three previous opcodes are executable in one clock. [Example E-6](#) shows a three word executable group (**add** followed by **ld<sub>a</sub>** with 32-bit constant) followed by a micro-flow. The sequence executes one clock faster when the branch target is long word aligned. The reason for the extra clock is described in [Section E.2.6, “Micro-flow Execution” on page E-29](#). Since optimization can save one clock under such circumstances, it could be worthwhile in small, frequently executed loops.

### Example E-6. Align Branch Targets

```

-- initialize --          -- initialize --
.align 2                .align 2
mov g0, g0      #nop    target:
target:                 add      g0, g1
add      g0, g1           lda      0xffffffff, g2
lda      0xffffffff, g2   scanbit  g3, g4
scanbit  g3, g4           addo     g5, g6
addo     g5, g6           - more -
- more -
Execution:               Execution:


| Clock | REGop   | MEMop | CTRLop   | Clock | REGop   | MEMop | CTRLop   |
|-------|---------|-------|----------|-------|---------|-------|----------|
| 1     |         |       | b target | 1     |         |       | b target |
| 2     |         |       | :        | 2     |         |       | :        |
| 3     | addo    | lda   |          | 3     | addo    | lda   |          |
| μ 4   | scanbit |       |          | μ 4   | scanbit |       |          |
| μ 5   | :       |       |          | 5     | addo    |       |          |
| 6     | addo    |       |          | 6     | more    |       |          |
| 7     | more    |       |          |       |         |       |          |


```

### E.2.7.9. Replacing Straight-Line Code and Calls

**bal** takes three or four clocks to execute and does not cause a frame spill to memory. Replacing **calls** with branch and link instructions is an obvious optimization. However, a not-so-obvious but equally beneficial optimization is to use branches and **bal** to reduce a critical procedure's code size.

When porting optimized algorithms originally written for other processors, the software engineer often expands the code in a straight-line fashion due to branch speed penalties of the original target and the lack of on-chip caching. On the i960 Hx processors, branches are virtually free in cached programs and cached program execution is dramatically faster than non-cached execution.

Therefore, branches and the branch-and-link instruction should be used to compress algorithms into the cache. For example, the previous low-pass filter routine could be modified to use coefficients from registers instead of literals. A short code piece could then sequence different filter coefficients through the registers and branch (using **bal**) to the filter loop. The entire routine would fit in the instruction cache and could perform a chain of linear filters without a procedure call.

### E.2.8 Utilizing On-chip Storage

The processor has the ability to consume instructions and execute quad word memory operations in parallel with arithmetic operations every clock. The instruction cache, data cache, register cache and on-chip data RAM are valuable resources for sustaining such optimized execution.

Compiler experimentation is an important aid to maximize utilization of on-chip storage resources. Compiler optimization is not limited to instruction caching. In particular, execution profiling will automate assignment of frequently used data to the data RAM. Availability of data RAM provides more options for partitioning data among context-based storage (register cache), general storage (data cache where available) and static caching (data RAM).

### E.2.8.1. Instruction Cache

If an algorithm can be compressed to fit into the instruction cache, it generally executes faster than if it did not fit. This is true even if the compressed code contains more comparisons and branches than the original code contains.

If a loop fits in the cache but is not capable of executing two instructions per clock due to memory or resource dependencies, keep unrolling the loop and pipelining operations until the cache is full. To increase performance of loops with multiple iterations and memory operations, unroll the loops until all registers are used or the cache is full.

If the system is interrupt-intensive, consider locking interrupt service routines into the cache. On the i960 Hx processor, cache locking is extended to any frequently executed code segments. Some experimentation may be necessary to determine if cache locking affects performance of remaining non-locked code.

Finally, as mentioned in a previous section on branches, aligning branch targets can improve performance. While long word aligned branch targets improve the scheduler's lookahead ability in the first clock of the branch, quad word aligned branch targets reduce the number of long word instruction fetches issued. Although the long word fetch is implemented to reduce cache miss latency for many cases, the quad word instruction fetch is more efficient for system throughput.

### E.2.8.2. Data Cache

The i960 Hx processor has an 8-Kbyte, four-way, set-associative data cache. The effect of data caching on performance is usually not as great as the effect of instruction caching because the processor often accesses data in a random, occasional pattern compared to the repetitive, looping pattern commonly seen with instruction execution.

The data cache behaves like SRAM for cache hits, delivering data in a single clock. Data cache misses require BCU interaction, as do all stores to external memory addresses. Data caching can be enabled for particular memory regions. In most cases, programmers will use this function only to distinguish non-cacheable memory-mapped I/O space from ordinary data memory. Once the data cache is enabled, its operation is transparent as there are no further programming options.

### E.2.8.3. Register Cache

Register cache can be thought of as a data cache which selectively caches only that data related to procedure context. [Section 7.1, “Call and Return Mechanism” on page 7-2](#) describes the i960 Hx processor’s register cache.

The register cache/data RAM partition is programmable. Therefore, the user can determine the trade-off between procedural context caching and static caching of procedure variables in the on-chip data RAM. Experiments can be run to measure the sensitivity of system performance to register cache depth of a fixed program. Minimizing register cache depth maximizes on-chip data RAM for variable caching.

Some situations exist where **flushreg** can optimize register cache usage. When an application crosses the boundary between non-real-time processing and real-time processing, it might be desirable to flush the register set. Flushing the register set at the beginning of a routine saves time that would otherwise be spent on frame spills later in the routine. However, this approach may actually result in a greater number of spills occurring than would otherwise have occurred without the premature flush.

This technique may be used to control interrupt latency within sections of background code. For example, it may be advantageous to execute a flush at the beginning of a routine which executes many loads from very slow memory. This reduces interrupt latency within that code section since there is no possibility of the interrupt’s frame spill being impeded by slow memory operations.

### E.2.8.4. Data RAM

On every clock, 128 bits of data can be loaded from or stored to the data RAM. This rate is sustained simultaneously with single-clock arithmetic operations executing from the independent REG-side register ports.

Allocated correctly, this resource dramatically increases performance of critical application algorithms. If data RAM space is scarce, locations can be dynamically allocated. If data RAM space is plentiful, locations can be globally allocated to achieve minimum latency to critical variables.

Variables which are used heavily over short periods of time or are used heavily by one procedure should be dynamically allocated. Such variables could be coefficients for filters which process large images on command. Dynamically allocated data RAM space would be loaded from main memory at the onset of intense processing and restored to main memory as the activity subsides.

Global allocation of DR space should be saved for storing variables that are heavily used by a variety of procedures over a long period of time or for storing variables needed by latency-critical activities. For example, the programmer may wish to allocate space for coefficients of a continuously operating filter.

## E.2.9 Summary

**Table E-19** summarizes code optimization tactics presented in the previous sections. Optimizing compilers for the i960 processor family are designed to exploit most of these techniques. Advanced compilers also incorporate profiling features to automate much of the experimentation process.

**Table E-19. Code Optimization Summary**

Tactic	Description
Advance “long” operations	Separate comparisons, loads, stores and MDU operations from the instructions that use their results.
Unroll loops	Unroll time-consuming loops until: 1) processor executes loop with two instructions per clock; 2) bus is saturated with quad operations; 3) no registers are left; 4) loop does not fit in the cache.
Order for parallelism	Alternate REG-side instructions with MEM-side instructions so they may be issued in parallel.
Migrate the operation	To enable parallelism, move EU and MDU operations to the AGU or vice versa.
Use branch prediction	Set prediction bits correctly in conditional instructions.
Align branch targets	Align branch targets of critical loops on an even word or quad word boundary.
Compress code to fit	If loop does not fit in cache, use branches, branch-and-links or calls to compress code size so it fits. Use code size optimization instructions (e.g., <b>cmppobe</b> ) where possible.
Use data RAM	Use high-bandwidth data RAM space for performance-critical and/or latency-critical variables

This appendix describes how to interface the processor to external memory systems. [Table F-1](#) shows the sample applications included in this chapter and the page number where each section begins.

**Table F-1. Sample Memory Interface Systems**

Memory Type	Refer to
Non-Pipelined Burst SRAM Interface	<a href="#">page F-1</a>
Pipelined SRAM Read Interface	<a href="#">page F-11</a>
Interfacing to Dynamic RAM	<a href="#">page F-16</a>
Interfacing to Slow Peripherals Using the Internal Wait State Generator	<a href="#">page F-32</a>
Synchronous Flash Interface	<a href="#">page F-38</a>

All issues discussed in each example are independent of operating frequency.

## F.1 Non-Pipelined Burst SRAM Interface

This example uses a simple SRAM design to demonstrate how the bus and control signals are used. The design also demonstrates the internal wait state generator. The SRAM interface provides the basic information needed to design most I/O and memory interfaces. The design supports burst and non-burst bus accesses. The SRAM interface is important for shared memory systems; variations can be used to communicate with external memory-mapped peripherals.

### F.1.1 Background

SRAM devices are available in a wide variety of packages and densities. SRAM address pins are always dedicated as inputs. Data pins may be configured in two ways:

- Each pin can be dedicated as an input or an output
- A set of data pins may be used for both data in and data out

Control signals usually found on asynchronous SRAM include: Chip Enable (CE#), Output Enable (OE#) and Write Enable (WE#). The following example deals with SRAM that has CE#, OE# and WE# control signals, address inputs and data input/output pins.

Memory is read when CE# and OE# are asserted and WE# is not asserted. Memory is written when CE# and WE# are asserted. The OE# input becomes don't care when WE# is asserted. However, it is recommended that OE# not be asserted at the beginning or end of a write cycle; this can lead to bus contention.

## F.1.2 Implementation

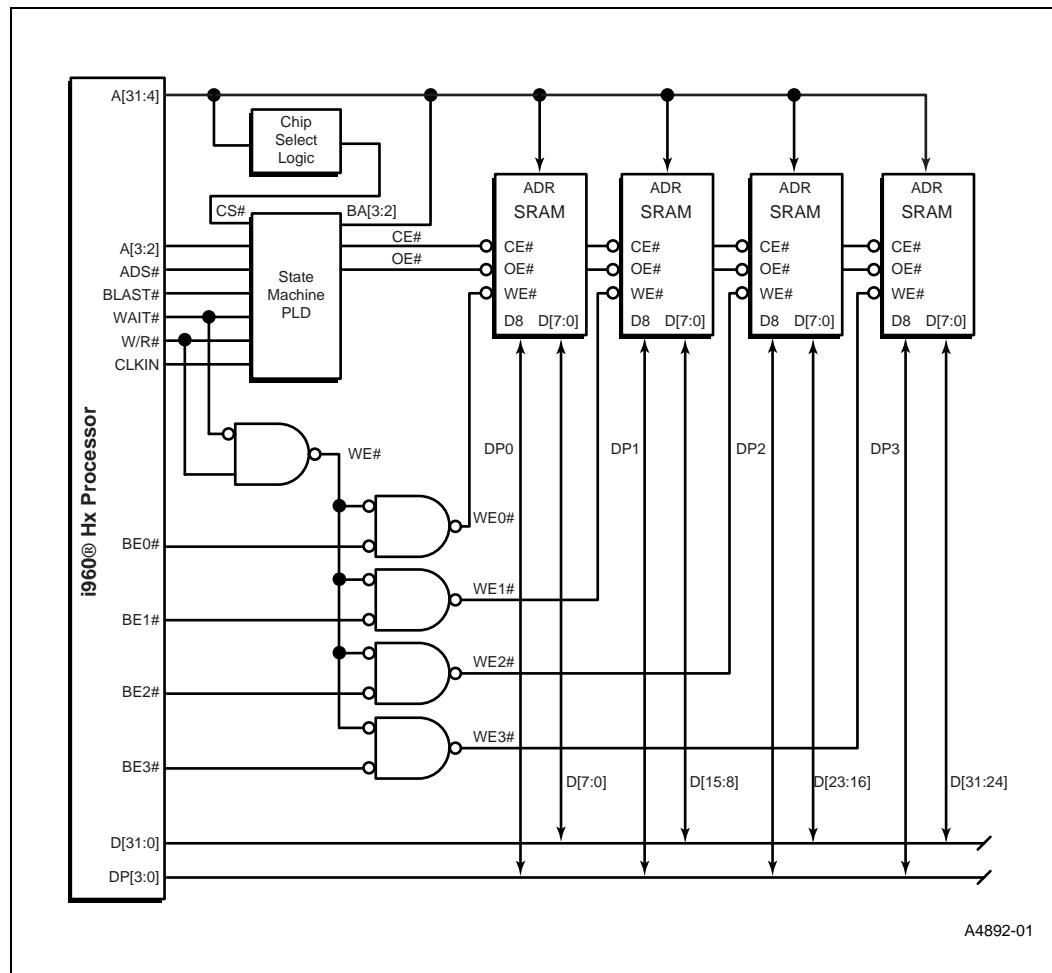
Figure F-1 illustrates a 32-bit burst access SRAM interface for a non-pipelined, RDY# and BTERM# disabled region. The design may be simplified if burst access modes are not required; it is easily modified for 8- or 16-bit buses. Note that this example requires wait states for writes.

WAIT#, generated by the internal wait state generator, is used to generate write strobes at the proper place in the write cycle. WAIT# is used in the address generation circuit to generate mid-burst addresses. External address generation improves performance in burst accesses.

## F.1.3 Block Diagram

The 32-bit burst SRAM interface consists of chip select logic, a state machine Programmable Logic Device (PLD) and write enable logic.

**Figure F-1. 32-Bit, Asynchronous, Non-Pipelined Burst SRAM Interface**



### F.1.3.1. Chip Select Logic

Chip select logic is a simple asynchronous decoder; it can be implemented with an external state machine or PLD. Chip Select (CS#) is based only on the address and is not qualified with any other signals. The state machine PLD qualifies CS# with ADS# to create the SRAM CE# signals. See [Section F.2.2, “Waveforms” on page F-14](#) for a more in-depth discussion of chip select generation.

### F.1.3.2. State Machine PLD

The SRAM state machine PLD generates the CE# and OE# signals to the SRAM. This PLD also contains the logic for creating burst address signals (BA[3:2]); this logic improves burst access performance. The improvement occurs because the i960<sup>®</sup> Hx processor's worst-case address valid delay is typically longer than the PLD's worst-case delay.

### F.1.3.3. Write Enable Generation Logic

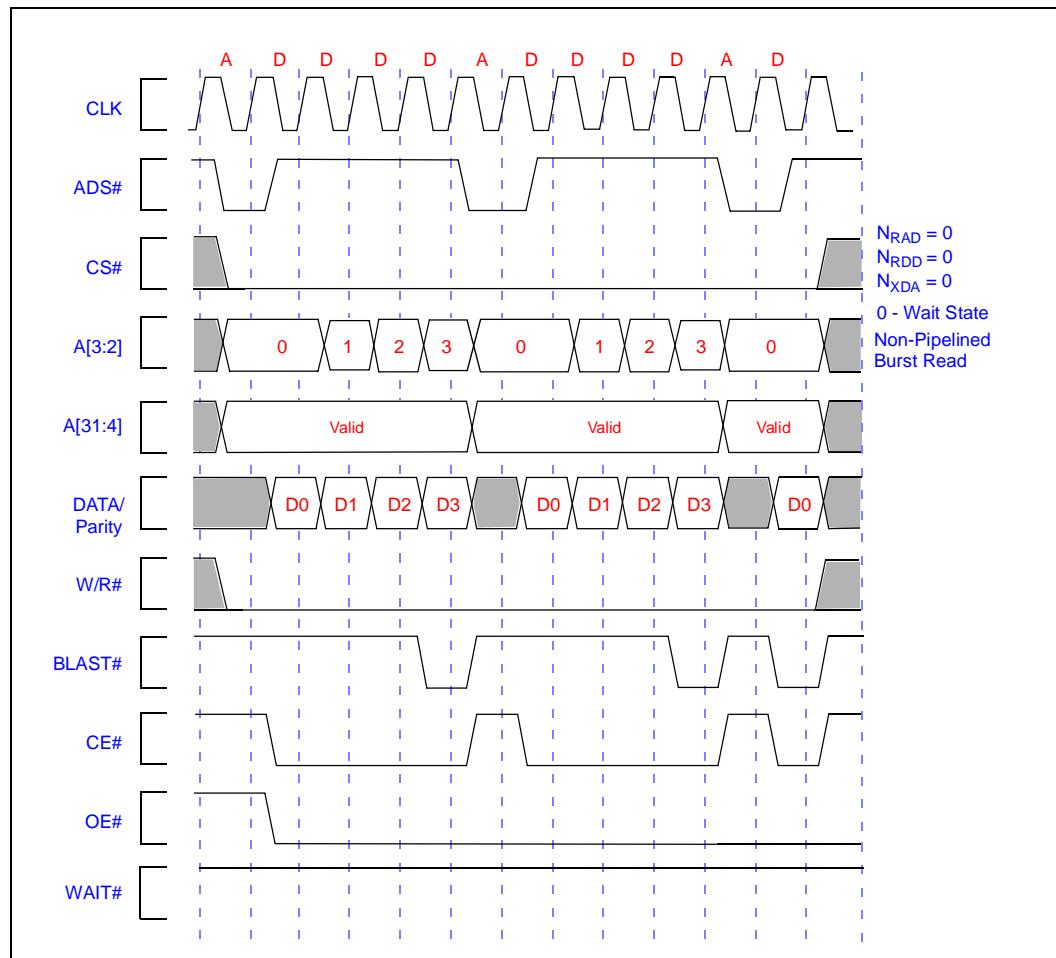
The write enable generation logic generates the WE# signal to the SRAM. WE# signals are conditioned on the i960 Hx processor byte enables (BE[3:0]#), the write/read signal (W/R#) and the wait signal (WAIT#).

There is a write enable signal (WE[3:0]#) for each byte position corresponding to the byte enable signals (BE[3:0]#); this allows byte, short word and word-wide writes. Read accesses to this memory system always result in word reads. In the case of byte- or short word reads, the i960 Hx processor read the data from the correct place on the data bus.

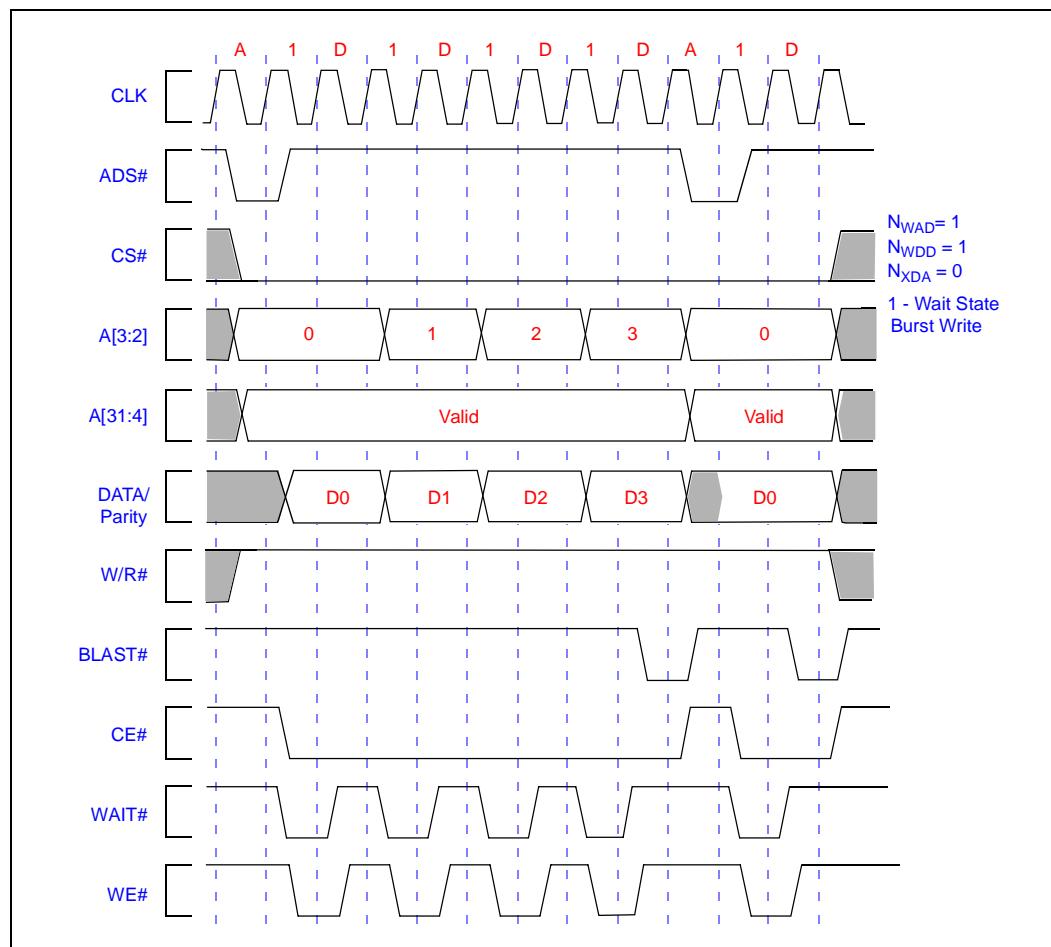
## F.1.4 Waveforms

Figure F-2 shows a non-pipelined SRAM read waveform; Figure F-3 shows a non-pipelined burst SRAM write waveform.

**Figure F-2. Non-Pipelined SRAM Read Waveform**



**Figure F-3. Non-Pipelined Burst SRAM Write Waveform**



#### F.1.4.1. Chip Select Generation

ADS# assertion during the CLKIN rising edge indicates the address is valid. Address setup time to this clock edge is CLKIN period ( $T_{PP}$ ), minus address output delay ( $T_{OV}$ ). CS# signal generation time (CS#\_gen) must satisfy the input setup time of the State Machine PLD( $T_{PLD\_setup}$ ). Therefore:

$$T_{CS\#\_gen} = T_{PP} - T_{OV} - T_{PLD\_setup}$$

#### F.1.4.2. Wait State Selection

The i960 Hx processor incorporates an internal wait state generator. Wait state selection is dictated by the memory system. The number of  $N_{RAD}$  wait states required is a function of output enable access time, chip enable access time or address access time.  $N_{RAD}$  must be selected so the wait states and data cycle accommodate the longest of these times. It is important to consider PLD output delay.

The number of  $N_{RDD}$  wait states required is a function of address access time.  $N_{RDD}$  must be selected so that the wait states and data cycle accommodate the memory system's address to data time. If the memory system is using the burst addresses provided by the i960 Hx processor, it is important to consider address output delay from the i960 Hx processor. If external address generation is used, PLD delay is important.

The number of  $N_{WAD}$  and  $N_{WDD}$  wait states required is a function of memory write cycle time. There must be at least 1  $N_{WAD}$  and 1  $N_{WDD}$  wait state for this example design to work properly. The number of  $N_{XDA}$  wait states required is a function of the memory system's output-to-float time.  $N_{XDA}$  determines how soon read data from the memory must be off the data bus before any other device asserts data on the data bus. This could be a read from another memory system or a write from the i960 Hx processor.

#### F.1.4.3. Output Enable and Write Enable Logic

The output enable signal is simply (see [Figure F-2](#)):

$$OE\# = W/R\#$$

The PLD is used to buffer the W/R# signal; this may be necessary to reduce the load on the processor's W/R# signal.

The write enable signals are:

$$WE\# = !(WAIT \& W/R\#);$$

or

$$WE0\# = !(WE \& BE0);$$

$$WE1\# = !(WE \& BE1);$$

$$WE2\# = !(WE \& BE2);$$

$$WE3\# = !(WE \& BE3);$$

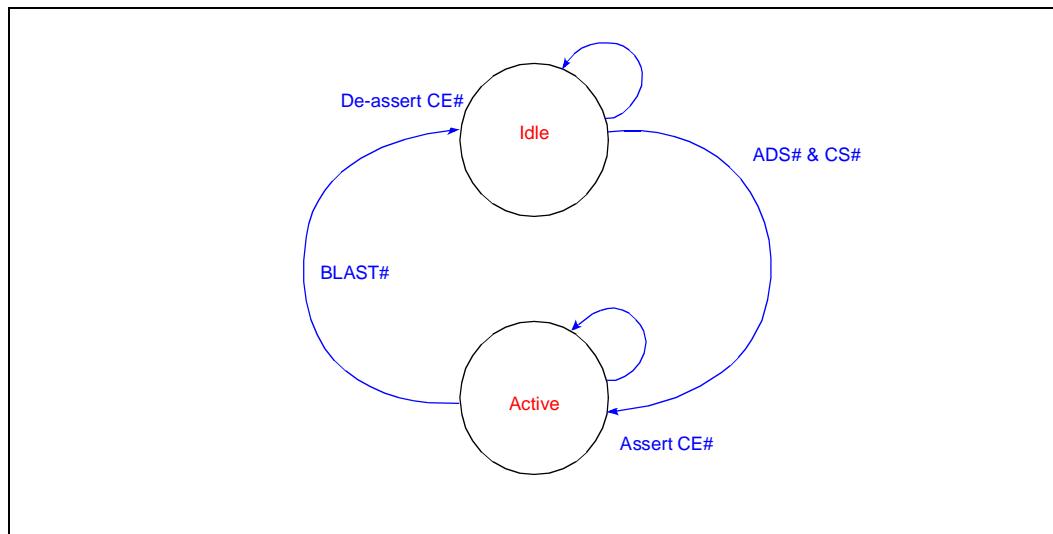
The WAIT# signal is used to create the write strobe. When W/R# indicates a write and BE# and WAIT# are asserted, the logic asserts WE#. The *80960HA/HD/HT Embedded 32-bit Microprocessor* datasheet guarantees a relationship from WAIT# high to write data invalid.

#### F.1.4.4. State Machine Descriptions

The state machine PLD incorporates two state machines: one controls SRAM chip enable (CE#); the other generates the A[3:2] address signals for multiple word burst accesses.

The chip enable state machine ([Figure F-4](#)) controls the CE# signal. CE# is normally not asserted, but when both ADS# and SRAM\_CS# are asserted, CE# is asserted and remains asserted until BLAST# is asserted. ADS indicates the beginning of an access, BLAST# indicates that the access is complete. CE# is the output of the state register; therefore, the CE# output delay is the clock-to-output time of the PLD. Minimizing CE# delay provides more memory access time.

[Figure F-4. Chip Enable State Machine](#)

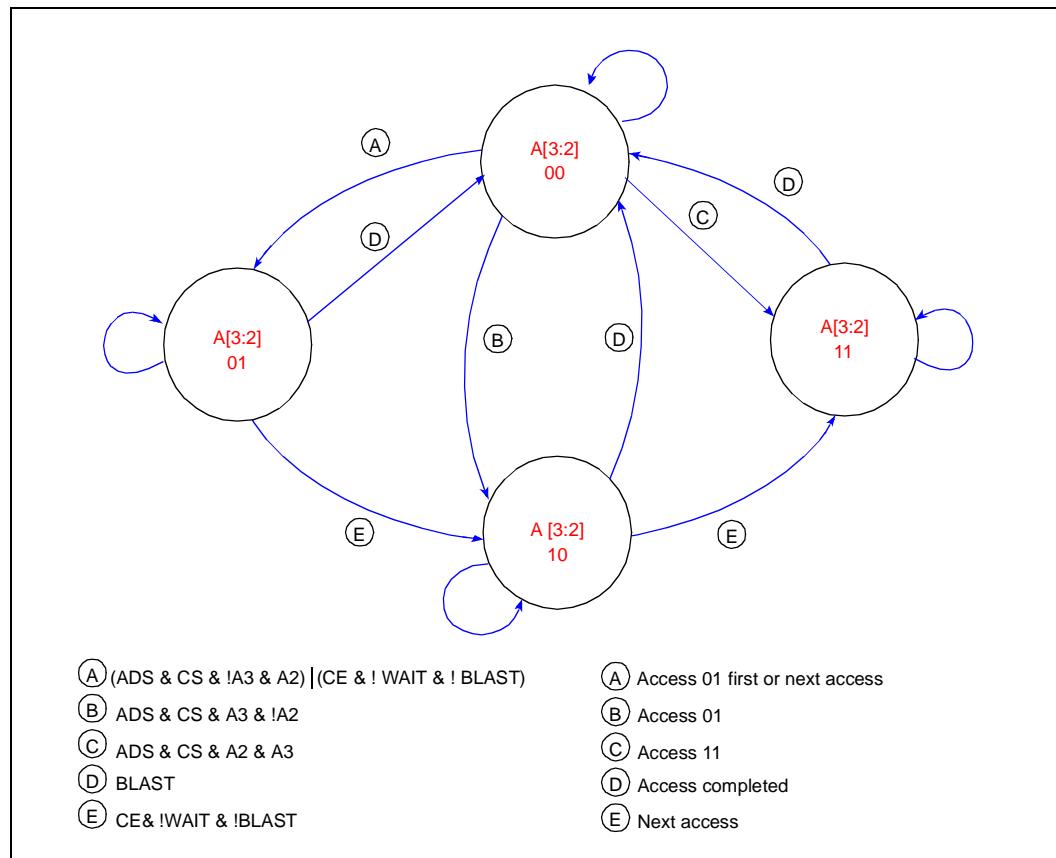


The A[3:2] address generation state machine ([Figure F-5](#)) generates consecutive addresses for multiple word burst accesses. The address generation state machine is not necessary if the memory region is defined in the region configuration table as non-burst, or if the processor's  $T_{OV}$  for A[3:2] meets system timing requirements.

The burst address outputs (BA[3:2]) correspond to registers within the PLD. Address generation time then corresponds to the clock-to-output time of the PLD. The BA[3:2] signals are forced to 0 when BLAST# is asserted.

The pseudo-code descriptions that follow the figures are provided only to describe the state machine diagrams. They are not intended to be PLD equations. A trailing # indicates a signal is asserted low.

In the pseudo-code description, the assertion of ADS# and SRAM\_CS# indicates the beginning of an access. The state machine jumps to the proper state based on A[3:2]. The assertion of CE# indicates that an access has begun. The assertion of CE#, !WAIT and !BLAST indicates that the current transfer is complete and it is time to generate the next address. The assertion of BLAST# indicates the access is complete.

**Figure F-5. A[3:2] Address Generation State Machine**


Pseudo-code Key			
#	signal is asserted low	==	equality test
!	logical NOT	:=	clocked assignment
&	logical AND	=	value assignment
	logical OR	X	Don't Care

```

STATE_0:          /* BA3:2 = 00 */
    IF          /* access 01 OR Next access */
        (ADS && SRAM_CS && (A3:2 == 01))||(CE & !WAIT & !BLAST);
    THEN
        next state is STATE_1;
    ELSE IF         /* access 10 */
        ADS && SRAM_CS && (A3:2 == 10);
    THEN
        next state is STATE_2;
    ELSE IF         /* access 11 */
        ADS && SRAM_CS && (A3:2 == 11);
    THEN
        next state is STATE_3;
    ELSE           /* Idle or access 00 */
        next state is STATE_0;
STATE_1:          /* BA3:2 = 01 */
    IF          /* Next access */
        CE & !WAIT & !BLAST;
    THEN
        next state is STATE_2;
    ELSE IF         /* Done */
        BLAST;
    THEN
        next state is STATE_0;
    ELSE           /* Just Wait */
        next state is STATE_1;
STATE_2:          /* BA3:2 = 10 */
    IF          /* Next access */
        CE & !WAIT & !BLAST;
    THEN
        next state is STATE_3;
    ELSE IF         /* Done */
        BLAST;
    THEN
        next state is STATE_0;
    ELSE           /* Just Wait */
        next state is STATE_2;
STATE_3:          /* BA3:2 = 11 */
    IF          /* Done */
        BLAST;
    THEN
        next state is STATE_0;
    ELSE
        next state is STATE_3;

```

## F.1.5 Trade-offs and Alternatives

The SRAM example just described demonstrates a burst SRAM interface. If a non-burst interface is desired, simply remove the address generation section of the state machine PLD. The design is also easily expanded to accommodate multiple banks of SRAM by providing multiple chip enables.

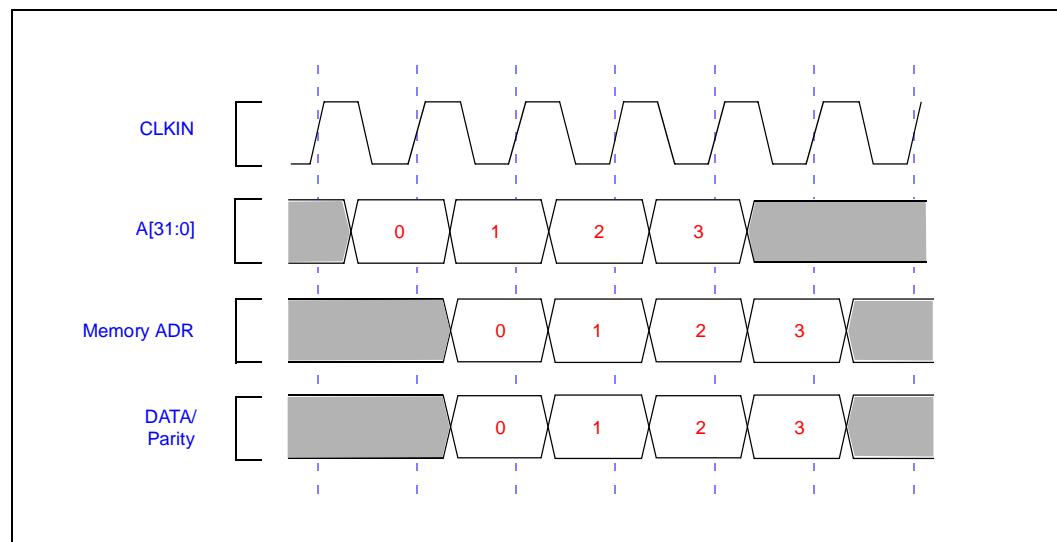
Using the i960 Hx processor's internal wait state generator frees the external memory control PLD from accounting for different memory speeds. Memory access parameters are entered into the Physical Memory Configuration (PMCON) registers. See [Section 14.2, “Programming the Physical Memory Configuration \(PMCON\) Registers” on page 14-7](#).

## F.2 Pipelined SRAM Read Interface

The following example illustrates the implementation of a pipelined read SRAM system. A zero wait state pipelined read memory system can have up to a 20 percent improvement in read data bandwidth over a non-pipelined memory system using the same memory devices. The pipelined read memory system is similar in design to the burst memory system.

A pipelined read memory system is the highest performance memory system that can be interfaced to the i960 Hx processor. The address cycle of consecutive read accesses is overlapped with the data cycle of the previous access. This results in the maximum bandwidth utilization of the bus. (See [Figure F-6](#).)

**Figure F-6. Pipelined Read Address and Data Transactions**



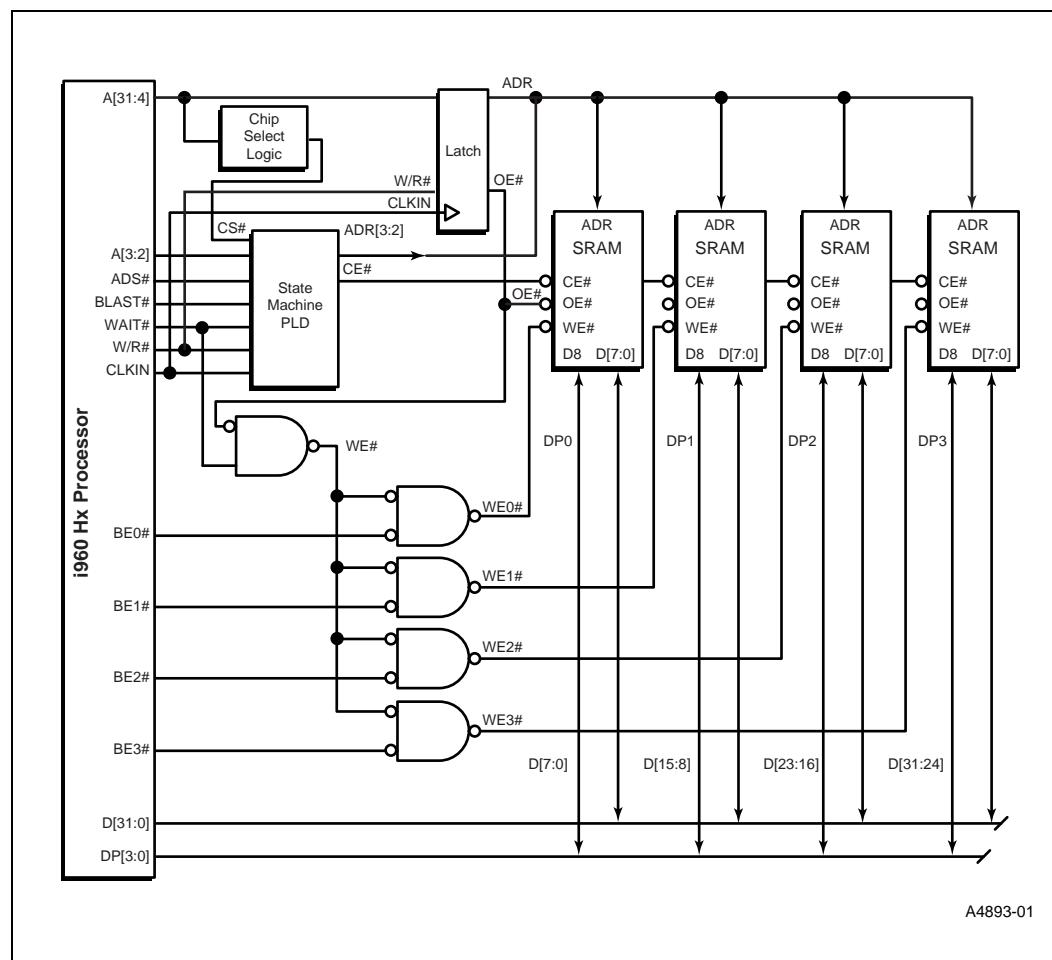
## F.2.1 Block Diagram

The same SRAM used in a non-pipelined read memory system can be used in a pipelined read memory system. [Figure F-7](#) shows a 32-bit-wide burst read pipelined memory system. Burst mode is used to speed write accesses.

The design of a pipelined read SRAM interface is very similar to the design of a non-pipelined SRAM interface. The difference is that an address latch and a W/R# latch have been added.

Chip select logic is a simple asynchronous data selector. Chip select (CS#) is based only on the address and is not qualified with any other signals. See [Section F.1, “Non-Pipelined Burst SRAM Interface” on page F-1](#) for more information on chip select generation.

**Figure F-7. Pipelined SRAM Interface Block Diagram**



### F.2.1.1. Address Latch

During pipelined reads, the i960 Hx processor outputs the next address during the last data cycle of the current access. This requires either an address latch or memory devices that are designed to work with the pipelined bus.

### F.2.1.2. State Machine PLD

The state machine PLD contains logic to control CE# and address signals A[3:2]. CE# is controlled by a simple state machine; A[3:2] automatically increment during burst accesses. For read accesses, the A[3:2] signals are pipelined and must be latched. Write accesses are not pipelined; therefore it is necessary to pass A[3:2] through without latching when W/R# is high. The A[3:2] generation is implemented as a state machine to achieve minimum address delay out of the PLD. ADR[3:2] (pipelined address 3:2) outputs are also the state bit of the PLD. This configuration ensures that the address delay is only the clock-to-output time for the PLD.

### F.2.1.3. Write Enable Logic

Write enable logic uses the byte enable signals (BE[3:0]#), the WAIT# signal and a latched version of the W/R# signal (OE#). Therefore:

$$\begin{aligned} \text{WE\#} &= !(\text{OE\#} \& \text{WAIT\#\& BE}); \\ \text{or:} \\ \text{WE0\#} &= !\text{OE\#} \mid \text{WAIT\#} \mid \text{BE0\#}; \\ \text{WE1\#} &= !\text{OE\#} \mid \text{WAIT\#} \mid \text{BE1\#}; \\ \text{WE2\#} &= !\text{OE\#} \mid \text{WAIT\#} \mid \text{BE2\#}; \\ \text{WE3\#} &= !\text{OE\#} \mid \text{WAIT\#} \mid \text{BE3\#}; \end{aligned}$$

DEN# remains asserted as long as consecutive pipelined read accesses continue. DEN# and DT/R# are related to the data, not the address; therefore, DEN# and DT/R# are not pipelined and retain the same timing for pipelined and non-pipelined reads.

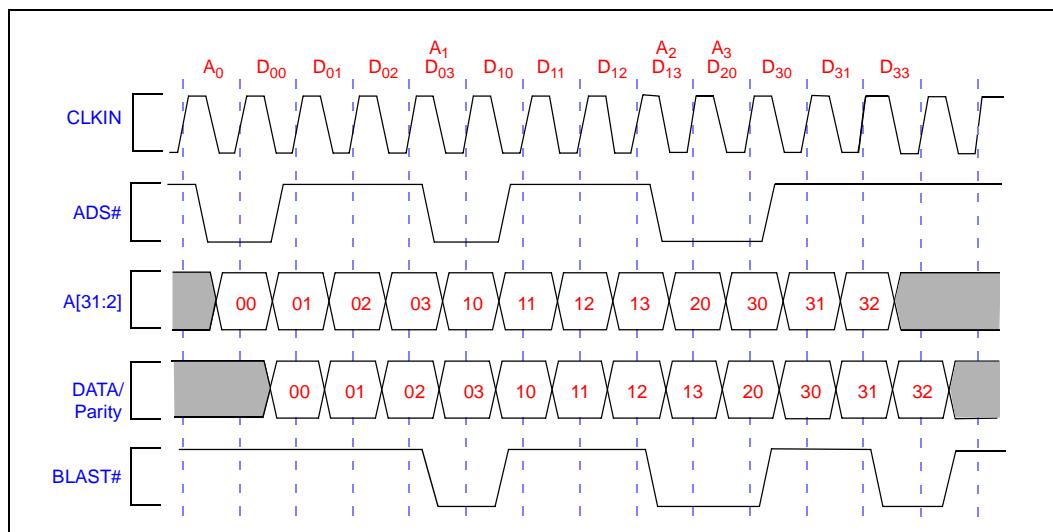
In pipelined read mode, a series of non-burst, zero-wait state accesses results in ADS# remaining asserted for consecutive clock cycles. Similarly, BLAST# remains asserted for several clock cycles.

W/R# behaves slightly differently for pipelined reads than for non-pipelined reads. W/R# is not valid for the last cycle of a pipelined read. This requires that W/R# be latched for pipelined reads similar to A[31:2]. The following signals are pipelined during pipelined read accesses: A[31:2], BE[3:0]#, SUP#, DMA# and D/C#. All of these pipelined signals are invalid during the last cycle of a pipelined read.

Address delay time for the pipelined read is the output valid time of the address latch (or the PA[3:2] generation PLD). Minimizing address delay maximizes access time.

## F.2.2 Waveforms

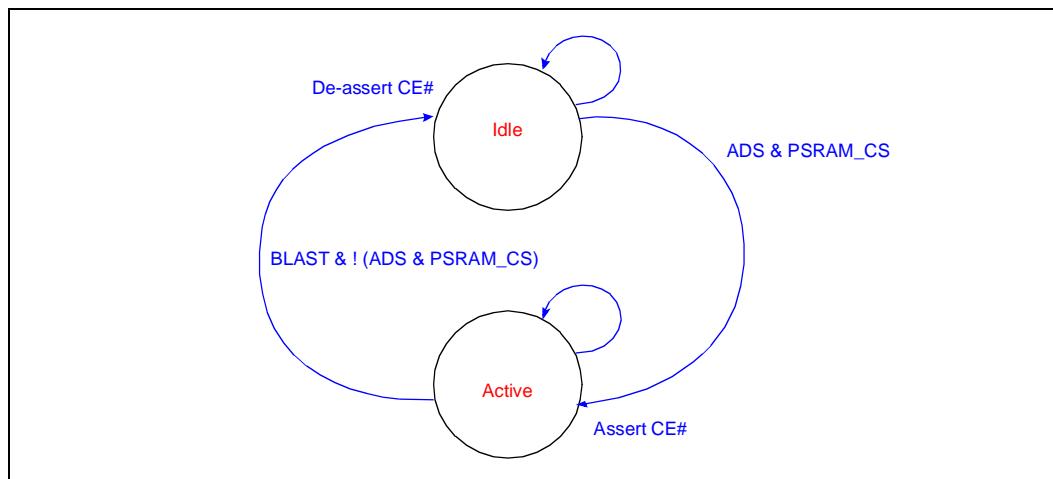
**Figure F-8. Pipelined Burst Read Waveform**



### F.2.2.1. State Machines

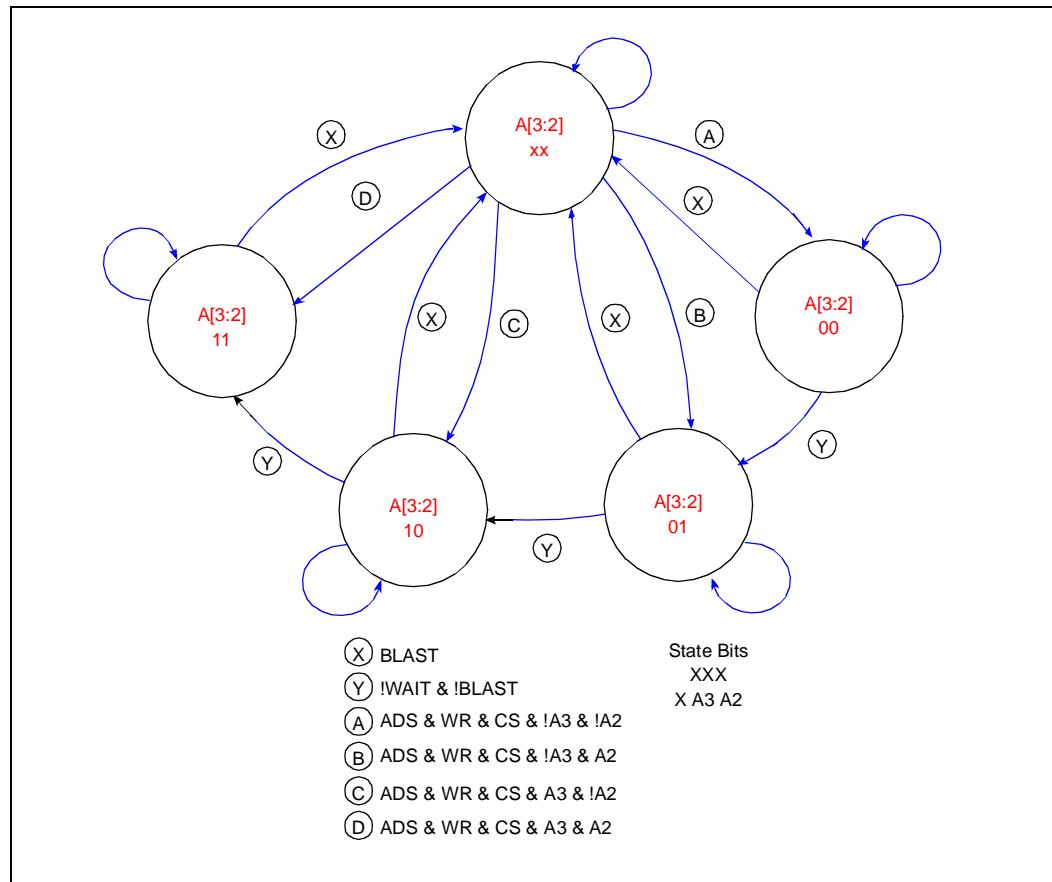
Chip enable (CE#) is controlled by a simple state machine. The state machine is normally in the idle state and CE# is not asserted. When ADS# and PSRAM\_CS# are asserted, the CE# state machine goes to the active state. CE# remains active until BLAST# is asserted unless another SRAM access follows immediately.

**Figure F-9. Pipelined Read Chip Enable State Machine**



The PA[3:2] state machine latches the A[3:2] address bits on read and generates the low address bit for writes. During read, PA[3:2] is a latched version of A[3:2]. If a write access occurs, the state machine generates the proper PA[3:2] addresses.

**Figure F-10. Pipelined Read PA[3:2] State Machine Diagram**



In the READ\_STATE, the state machine simply latches A[3:2] and outputs them as PA[3:2]. On a write, the state machine jumps to the appropriate state based on the value of A[3:2]. When in a write state, the state machine will advance to the next write state if WAIT# and BLAST# are not asserted. The state machine can advance from any write state to the READ\_STATE.

### F.2.3 Trade-offs and Alternatives

The example described above demonstrates a burst pipelined read SRAM interface. Burst mode is used to improve write performance. If write performance is not critical (i.e., if the region is used only for code), the next address generation PLD can be removed.

## F.3 Interfacing to Dynamic RAM

This section provides an overview of DRAM and DRAM access modes and describes an i960 Hx processor-specific DRAM interface. A specific design example using the CAS#-before-RAS# refresh method is also included. This design illustrates the advantage of the i960 Hx processor's burst bus and the fast column address access times available on many modern DRAMs.

The burst bus and memory region configuration tables simplify DRAM interface to the i960 Hx processor. DRAM systems can be designed in many ways, there are memory access options, memory system configuration options and refresh mode options.

DRAM offers high data density and low cost per bit compared to SRAM. DRAM is available in a wide variety of packages, making it easy to pack a lot of memory into a small space. DRAM features described here are provided as general information. (See specific data sheets for detailed information.)

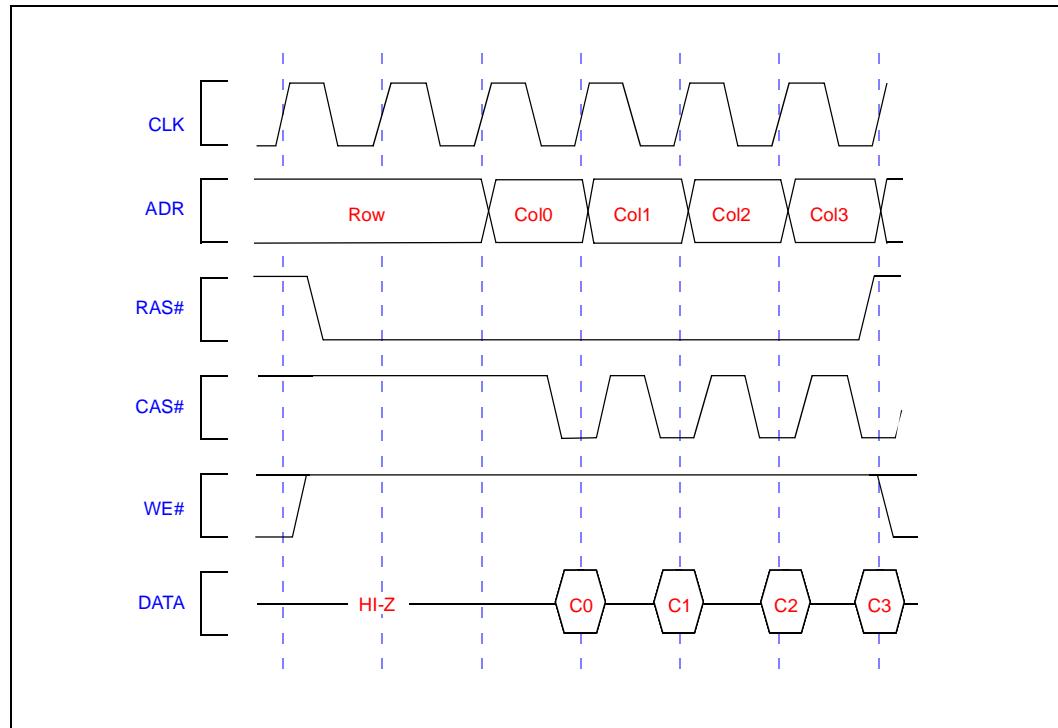
The i960 Hx processor's burst mode bus is well suited to the high-speed multiple column access modes found in DRAM. Fast page mode DRAM can easily be exploited to improve memory system performance.

All DRAMs have a multiplexed address bus, a write enable input (WE#) and two address strobes: row address strobe (RAS#) and column address strobe (CAS#). Some DRAMs also have an output enable input (OE#). DRAMs are accessed by placing a valid row address on the address input pins and asserting RAS#; then the column address is driven onto the DRAM address pins and CAS# is asserted. Write enable (WE#) input on the DRAM determines whether the access is a read or write. Output enable input (OE#) — found on some DRAMs — controls the DRAM output buffers and can be useful for multibanked and interleaved designs.

### F.3.1 Fast Page Mode DRAM

Fast page mode DRAM (see Figure F-11) allows any column within a selected row to be read or written at a high data rate. A read or write cycle starts by asserting RAS#. Strobing CAS# accesses the selected column data. During reads, the CAS# falling edge latches the address (internal to the DRAM) and enables the output. The processor's four word burst bus can easily take advantage of the faster column access times provided by fast page mode DRAM.

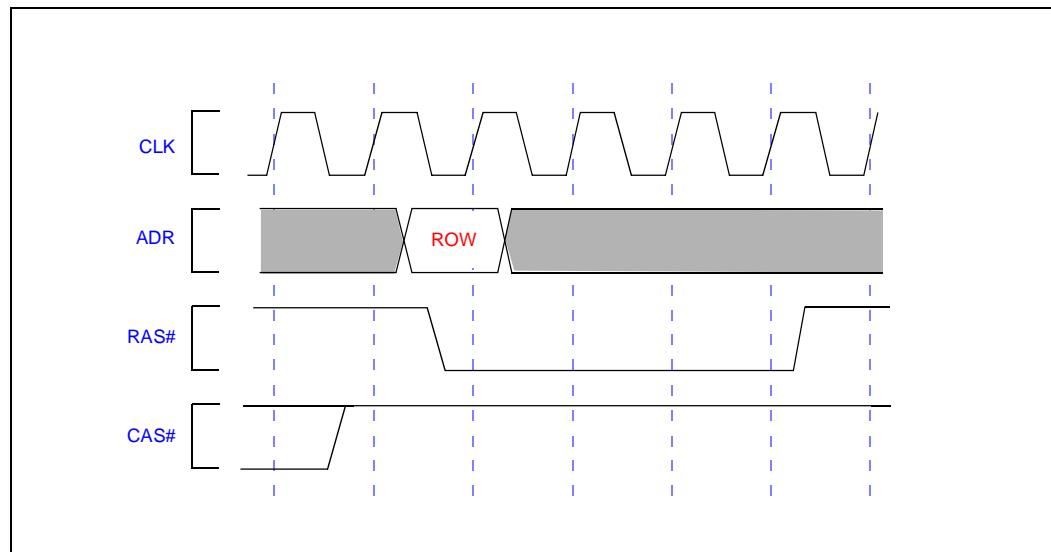
**Figure F-11. Fast Page Mode DRAM Read**



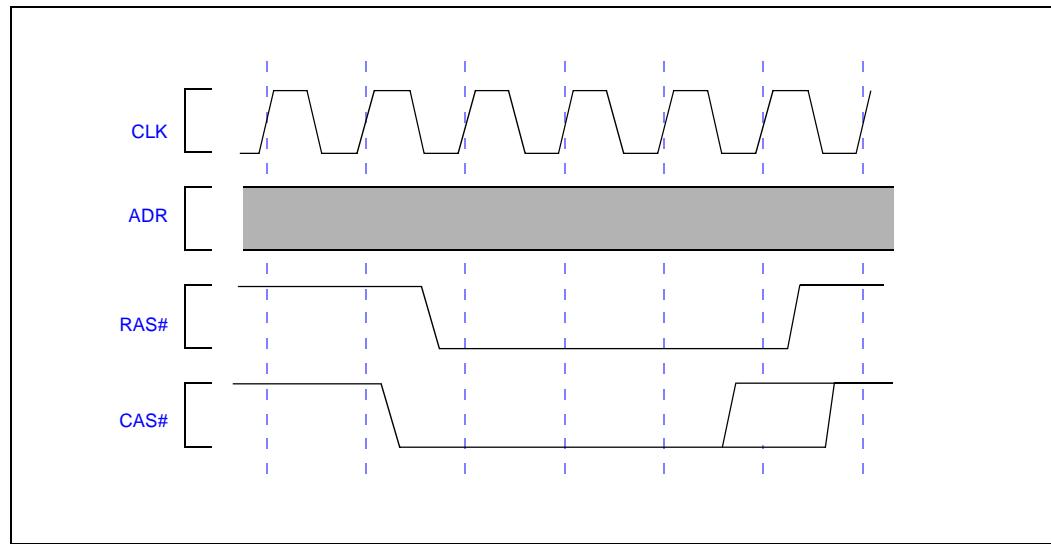
### F.3.2 DRAM Refresh Modes

All DRAMs require periodic refreshes to retain data. DRAMs may be refreshed in one of two ways: RAS#-only refresh or CAS#-before-RAS# refresh. RAS#-only refresh (Figure F-12) is done by asserting a row address on the address pins and asserting RAS#. CAS# is not asserted. A single, RAS#-only refresh cycle refreshes all columns within the selected row. RAS#-only refresh was the method of refreshing older DRAMs; however, most modern DRAMs provide the much easier CAS#-before-RAS# refresh method. CAS#-before-RAS# refreshes (Figure F-13) do not require an address to be generated; DRAM generates the row address with an internal counter.

**Figure F-12. RAS#-only DRAM Refresh**



**Figure F-13. CAS#-before-RAS# DRAM Refresh**



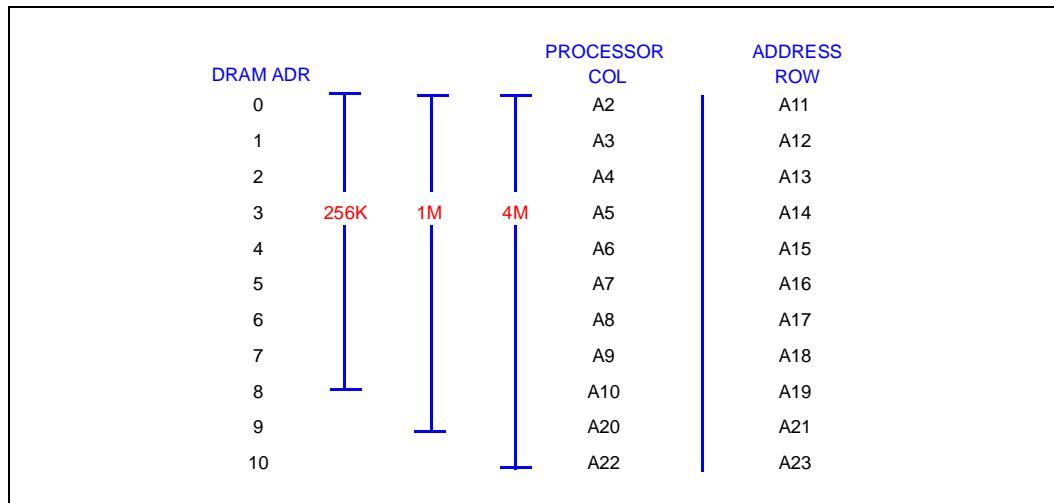
DRAM may be refreshed in either a distributed or a burst manner. Burst refresh does not refer to the burst access bus. The term simply means that all memory rows are sequentially accessed when the refresh interval time expires. Distributed refresh implies that refresh cycles are distributed within the refresh interval required by the memory.

Distributed refresh cycles are spread out over the refresh interval, reducing possible access latency. Burst refreshing may lock the processor out of the DRAM for a longer period and so may be inappropriate for some applications. Burst refreshing, however, guarantees that no refresh activity occurs between refresh intervals. Some applications may take advantage of this to burst refresh the DRAM during a time it will not be accessed, making refresh invisible to the application.

### F.3.3 Address Multiplexer Input Connections

Address multiplexer inputs can be ordered such that 256-Kbyte through 4-Mbyte DRAM can be supported. Interleaving the upper address signals provides compatibility with all these memory densities. [Figure F-14](#) illustrates this arrangement. Availability of DRAM modules with standard pinouts provides an easy path for future memory expansion.

**Figure F-14. Address Multiplexer Inputs**



### F.3.4 Series Damping Resistors

Series-damping resistors are recommended on all DRAM control and address inputs. These resistors prevent overshoot and undershoot on input lines. Damping is required because of the large capacitive load present when many DRAMs are connected together, combined with circuit board trace inductance. Damping resistor values are typically between 15 and 100 Ohms, depending on the load, the lower the load, the higher the required damping resistor value. If the damping resistor value is too high, the signal is overdamped, extending memory cycle time. If the damping resistor value is too low, overshoot or undershoot is not sufficiently damped. Place damping resistors near the driving source.

### F.3.5 System Loading

The i960 Hx processor can drive a large capacitive load. However, systems with many DRAM banks may require data buffers and, for interleaved designs, multiplexers to isolate the DRAM load from the i960 Hx processor or other system components with less drive capability (e.g., high speed SRAM).

RAS# and CAS# inputs to the DRAM should also be designed with consideration for capacitive load. When many DRAMs are connected to common RAS# and CAS# signals, the capacitive load can be considerable.

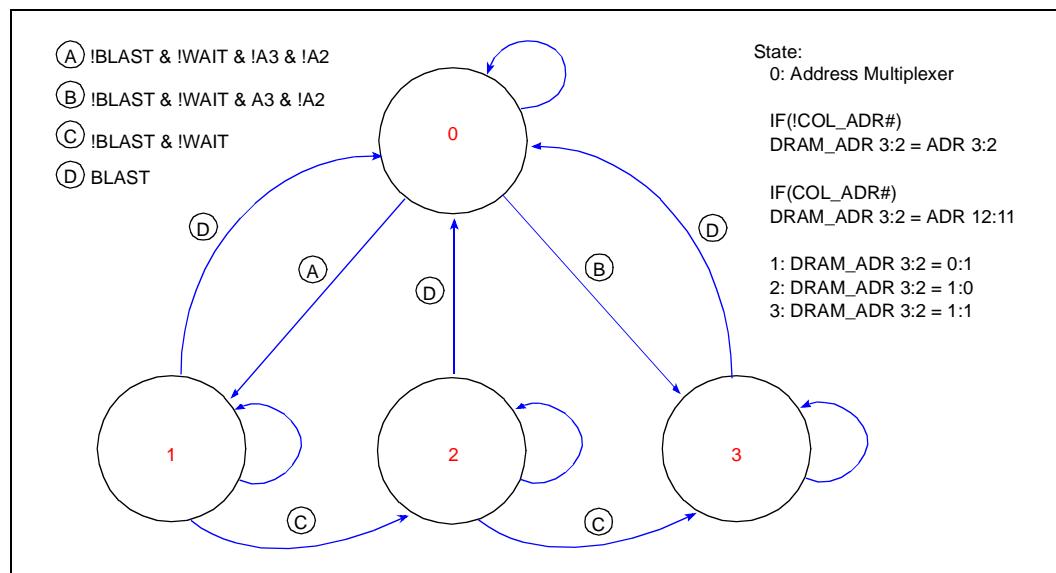
### F.3.6 DRAM Address Generation

DRAM address generation logic speeds burst accesses for fast page mode DRAM. This is accomplished by reducing the time required to present the consecutive column addresses during a burst access. If the address generator is not present, the address valid delay time consists of the worst-case address valid delay time plus the worst-case propagation delay through the input address multiplexer.

DRAM address generation logic must control the DRAM address's two least significant bits. During the initial DRAM access, address generation logic acts like a multiplexer. During column accesses within a burst, address generation logic generates consecutive addresses. Therefore, DRAM address generation logic is designed to function as a multiplexer and an address generator.

If an address generator is used, address valid delay time is equal to address generation time. Address generation delay time consists of the clock-to-feedback and feedback-to-output delays for the selected device.

**Figure F-15** shows the state diagram for address generation logic (see also [Example F-1](#)). Signals into the DRAM burst address logic are: ADR2, ADR3, ADR11, ADR12, WAIT# and BLAST# from the processor and COL\_ADR# from the DRAM control logic. COL\_ADR# indicates whether the DRAM controller is requesting the row address (COL\_ADR# not asserted) or column address (COL\_ADR# asserted). Signals output from DRAM burst address logic are the DRAM address two least significant bits, DRAM\_ADR[3:2]. The pseudo-code following the figure is provided only to describe the state machine diagram. It is not intended for direct use as PLD equations.

**Figure F-15. DRAM Address Generation State Machine**


**Example F-1. Address Generation Logic State Machine Pseudocode**

```

STATE_0: /* Multiplexer Emulation */
    DRAM_ADR3 = (COL_ADR && A3) || (!COL_ADR && A12);
    DRAM_ADR2 = (COL_ADR && A2) || (!COL_ADR && A11);
    IF          /* address generation */
        WAIT && !BLAST && COL_ADR
        && (ADR3 == 0) && (ADR2 == 0);
    THEN
        next state is STATE_1;
    ELSE IF
        WAIT && BLAST && COL_ADR
        && (ADR3 == 1) && (ADR2 == 0);
    THEN
        next state is STATE_3;
    ELSE
        next state is STATE_0;
STATE_1: /* Generate address 01 */
    DRAM_ADR3 = 0;
    DRAM_ADR2 = 1;
    IF
        BLAST;
    THEN
        next state is STATE_0;
    ELSE IF
        !BLAST && !WAIT;
    THEN
        next state is STATE_2;
    ELSE
        next state is STATE_1
STATE_2: /* Generate address 10 */
    DRAM_ADR3 = 1;
    DRAM_ADR2 = 0;
    IF
        BLAST;
    THEN
        next state is STATE_0;
    ELSE IF
        !BLAST && !WAIT;
    THEN
        next state is STATE_3;
    ELSE
        next state is STATE_2
STATE_3: /* Generate address 11 */
    DRAM_ADR3 = 1;
    DRAM_ADR2 = 1;
    IF
        BLAST;
    THEN
        the next state is STATE_0;
    ELSE
        next state is STATE_3

```

### F.3.7 Memory Ready

The memory ready input to the i960 Hx processor (READY#) indicates the completion of a DRAM read or write cycle. READY# must be generated by the DRAM controller and must satisfy setup and hold times specified in the data sheet. If multiple memory systems are using READY#, ready signals from these memory systems must be logically ORed together.

### F.3.8 Region Programming

Region programming is critical to DRAM operation.  $N_{RAD}$  and  $N_{WAD}$  wait states must satisfy RAS#, CAS# and address valid times for the DRAM.  $N_{RDD}$  and  $N_{WDD}$  times must satisfy the column address to data access times. The  $N_{XDA}$  time must satisfy RAS# precharge time.

Figure F-16 and Figure F-17 show typical system waveforms for this design. Note that RAS# is not asserted until the cycle after the address cycle; this delay is intended to accommodate the RAS# precharge time. In some DRAM designs, it may be possible to remove RAS# before an access is complete. If RAS# can be removed early in the access, RAS# precharge can occur during the access.

**Figure F-16. Fast Page DRAM System Read Waveform**

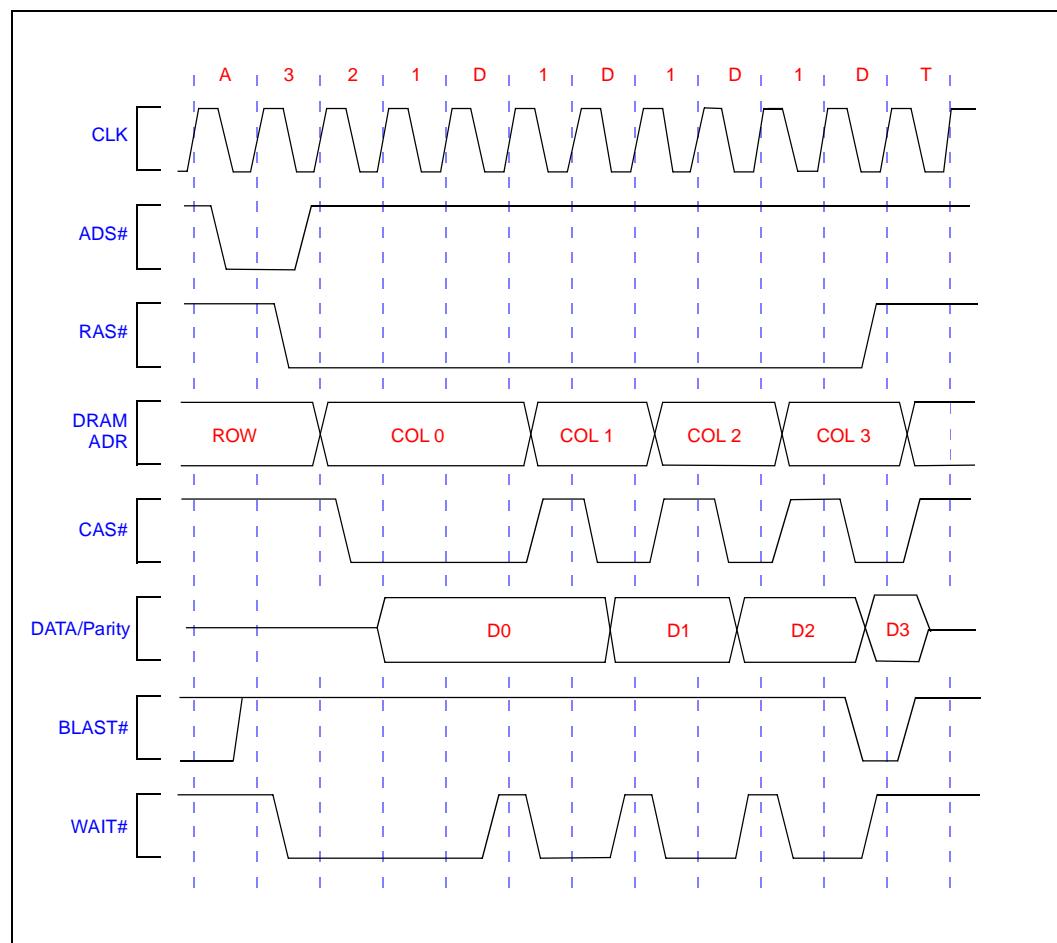
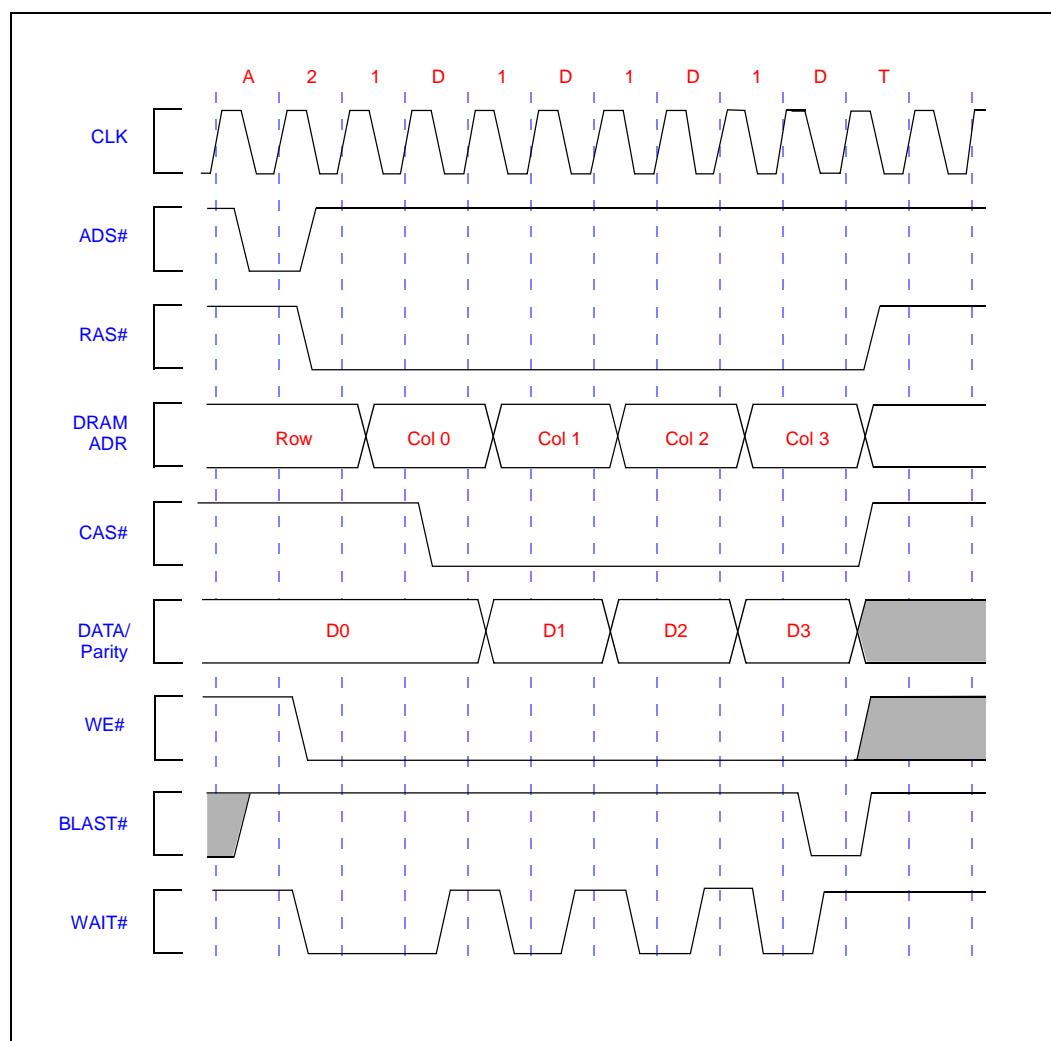


Figure F-17. Fast Page Mode DRAM System Write Waveform

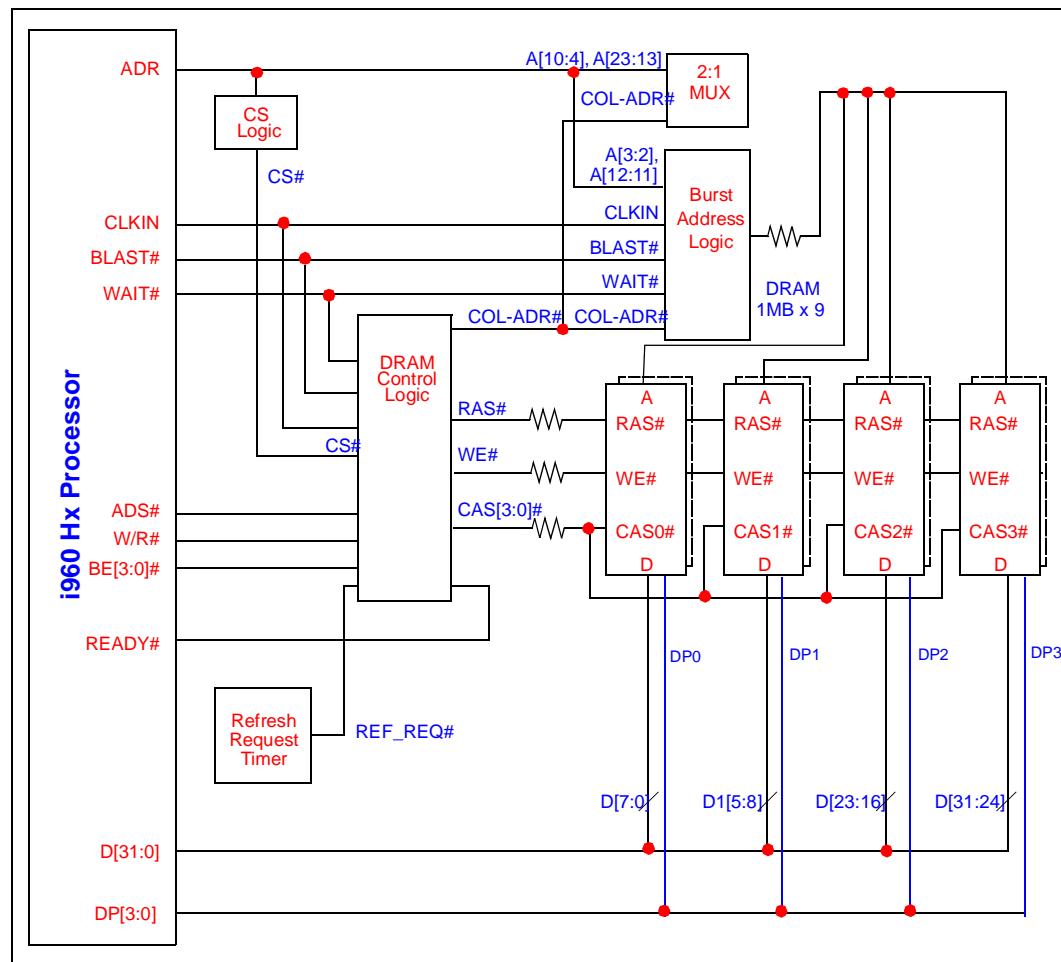


### F.3.9 Design Example: Burst DRAM with Distributed CAS#-Before-RAS# Refresh Using READY# Control

This example illustrates a 4 Mbyte DRAM system design that uses CAS#-before-RAS# refresh and READY# control. CAS#-before-RAS# refresh uses the internal refresh address generation capabilities of modern DRAMs. READY# must be generated by the DRAM controller to indicate that a data transfer is complete. The controller must also arbitrate between access and refresh requests, and control the address multiplexer and RAS# precharge time. The internal wait state generator is not used. The DRAM controller must be designed with information about processor and DRAM speed.

The memory system block diagram is shown in Figure F-18.

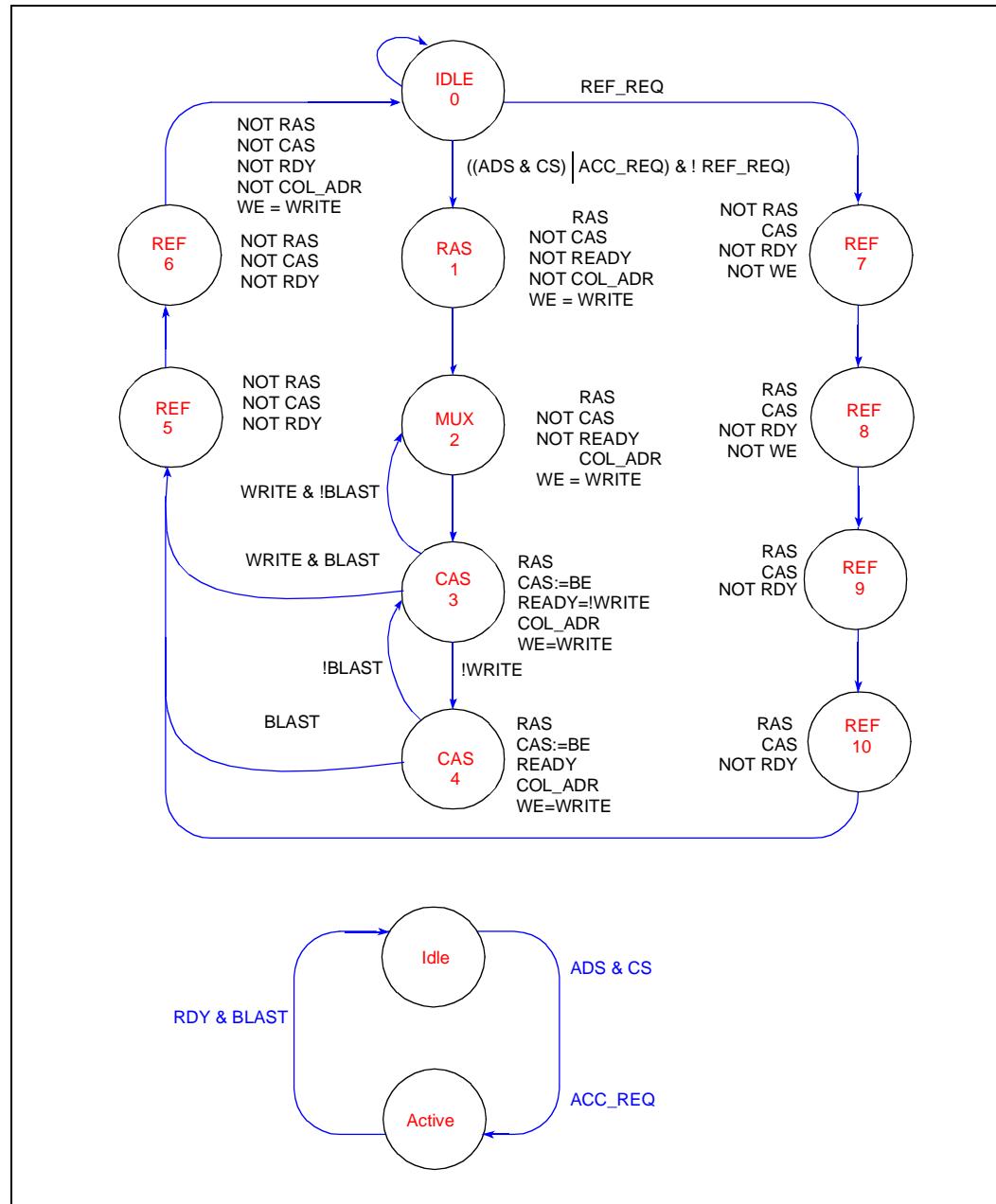
**Figure F-18. Memory System Block Diagram**



### F.3.10 DRAM Controller State Machine

The state machine shown in Figure F-19 controls both normal accesses and DRAM refreshes. CAS#-before-RAS# refresh mode does not require the bus or any processor intervention; therefore, DRAM refresh occurs autonomously. The DRAM controller state machine described here assumes 80 ns fast page mode DRAM with a 33 MHz clock (CLKIN). This DRAM controller does not require the internal wait state generator; as a result, all wait state parameters can be programmed to zero (0).

**Figure F-19. DRAM State Machine**



The refresh request timer generates the refresh request signal (REF\_REQ#), indicating that it is time to refresh the DRAM. The controller gives preference to refresh requests over access requests. This ensures that the entire memory remains refreshed. The access request signal (ACC\_REQ) shown on the state diagram is a latched signal. ACC\_REQ is asserted when ADS# and DRAM\_CS# are both asserted. ACC\_REQ is deasserted when BLAST is asserted. It is necessary to latch the access request because the controller could be in a refresh or RAS# precharge state when the processor accesses the DRAM.

The pseudo-code description below is provided only to describe the state machine diagram. It is not intended to be used directly as PLD equations.

```

STATE_0:           /* Idle */
    RAS      is not asserted;
    CAS3:0   is not asserted;
    COL_ADR  is not asserted;
    READY    is not asserted;
    WE = W/R;
    IF
        REF_REQ;
    THEN
        the next state is STATE_7; /* Refresh */
    ELSE IF
        (ADS && DRAM_CS) || ACC_REQ;
    THEN
        the next state is STATE_1; /* Access */
    ELSE
        the next state is STATE_0; /* Idle */
STATE_1:           /* Assert RAS# */
    RAS      is asserted;
    CAS3:0   is not asserted;
    COL_ADR  is not asserted;
    READY    is not asserted;
    WE = W/R;
    the next state is STATE_2;
STATE_2:           /* MUX the address */
    RAS      is asserted;
    CAS3:0   is not asserted;
    COL_ADR  is asserted;
    READY    is not asserted;
    WE = W/R;
STATE_3:           /* Assert CAS#, write is
                     ready, read is not */
    RAS      is asserted;
    CAS3:0 = BE3:0;
    COL_ADR is asserted;
    READY = !W/R;
STATE_0:           /* Idle */
    WE = W/R;
    IF
        W/R && BLAST; /* Write access not done */
    THEN
        the next state is STATE_2; /* remove CAS# */
    ELSE IF
        W/R# && BLAST; /* Write Finished */
    THEN
        the next state is STATE_5; /* RAS# Precharge */
    ELSE
        /* !W/R#, Read */
        the next state is STATE_4; /* Read */

```

```

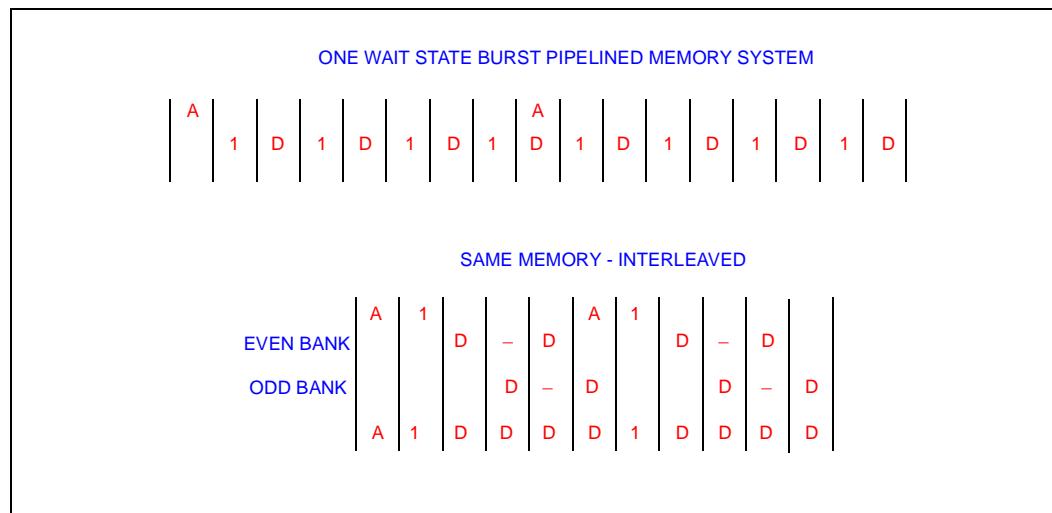
STATE_4:                                /* Read data ready */
    RAS      is asserted;
    CAS3:0 = BE3:0;
    COL_ADR is asserted;
    READY    is asserted;
    WE = W/R;
    IF
        BLAST           /* read not Done */
    THEN
        the next state is STATE_3; /* Remove READY */
    ELSE
        /* BLAST, Read Done */
        the next state is STATE_5; /*RAS# Precharge*/
        the next state is STATE_3;
STATE_5:                                /* RAS# Precharge */
    RAS      is not asserted;
    CAS3:0  is not asserted;
    COL_ADR = X;
    READY    is not asserted;
    WE = X;
    the next state is STATE_6;
STATE_6:                                /* More RAS# Precharge */
    RAS      is not asserted;
    CAS3:0  is not asserted;
    COL_ADR = X;
    READY    is not asserted;
    WE = X;
    the next state is STATE_0; /*Return to idle*/
STATE_7:                                /* Refresh, assert CAS# */
    RAS      is not asserted;
    CAS3:0  is asserted;
    COL_ADR = X;
    READY    is not asserted;
    WE      is not asserted;
    the next state is STATE_8;
STATE_8:                                /* Refresh, assert RAS# */
    RAS      is asserted;
    CAS3:0  is asserted;
    COL_ADR = X;
    READY    is not asserted;
    WE      is not asserted;
    the next state is STATE_8;STATE_9: /* Refresh Hold
                                         RAS# */
    RAS      is asserted;
    CAS3:0  is asserted;
    COL_ADR = X;
    READY    is not asserted;
    WE      is X;
    the next state is STATE_10;
STATE_10:                               /* Refresh Hold RAS# */
    RAS      is asserted;
    CAS3:0  is asserted;
    COL_ADR = X;
    READY    is not asserted;
    WE      is not asserted;
    the next state is STATE_5;      /*RAS# Precharge*/

```

## F.4 Interleaved Memory Systems

Interleaving memory can provide a significant improvement in memory system performance. Interleaved memory systems overlap accesses to consecutive addresses; this results in higher performance with slower memory. Two-way memory interleaving is accomplished by dividing the memory into banks: one bank for even word addresses, one for odd word addresses. The least significant address bit (A2) is used to select a bank. The two banks are read in parallel and the data is put onto the data bus by a multiplexer. This can allow the wait states of the second access to be overlapped with the data transfer of the first access. [Figure F-20](#) shows the access overlap for a burst access.

**Figure F-20. Two-Way Interleaved Read Access Overlap**



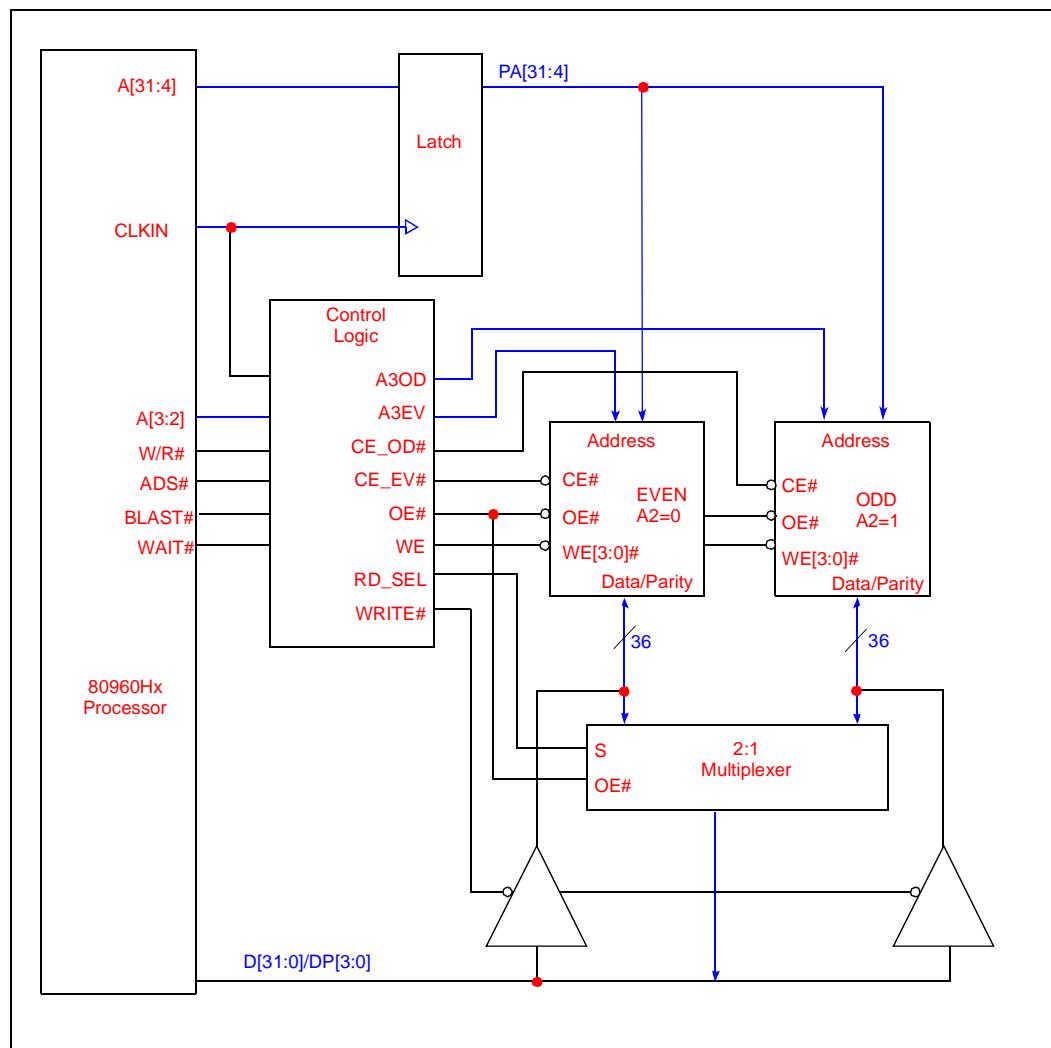
[Figure F-21](#) is a simple schematic of a two-way, interleaved, pipelined memory system. The design is similar to the design of a non-interleaved pipelined memory design with the following exceptions:

- An output data multiplexer is used to prevent data contention
- The write data buffers isolate the memory data buses for writes
- The low address bit to the memory devices is A3

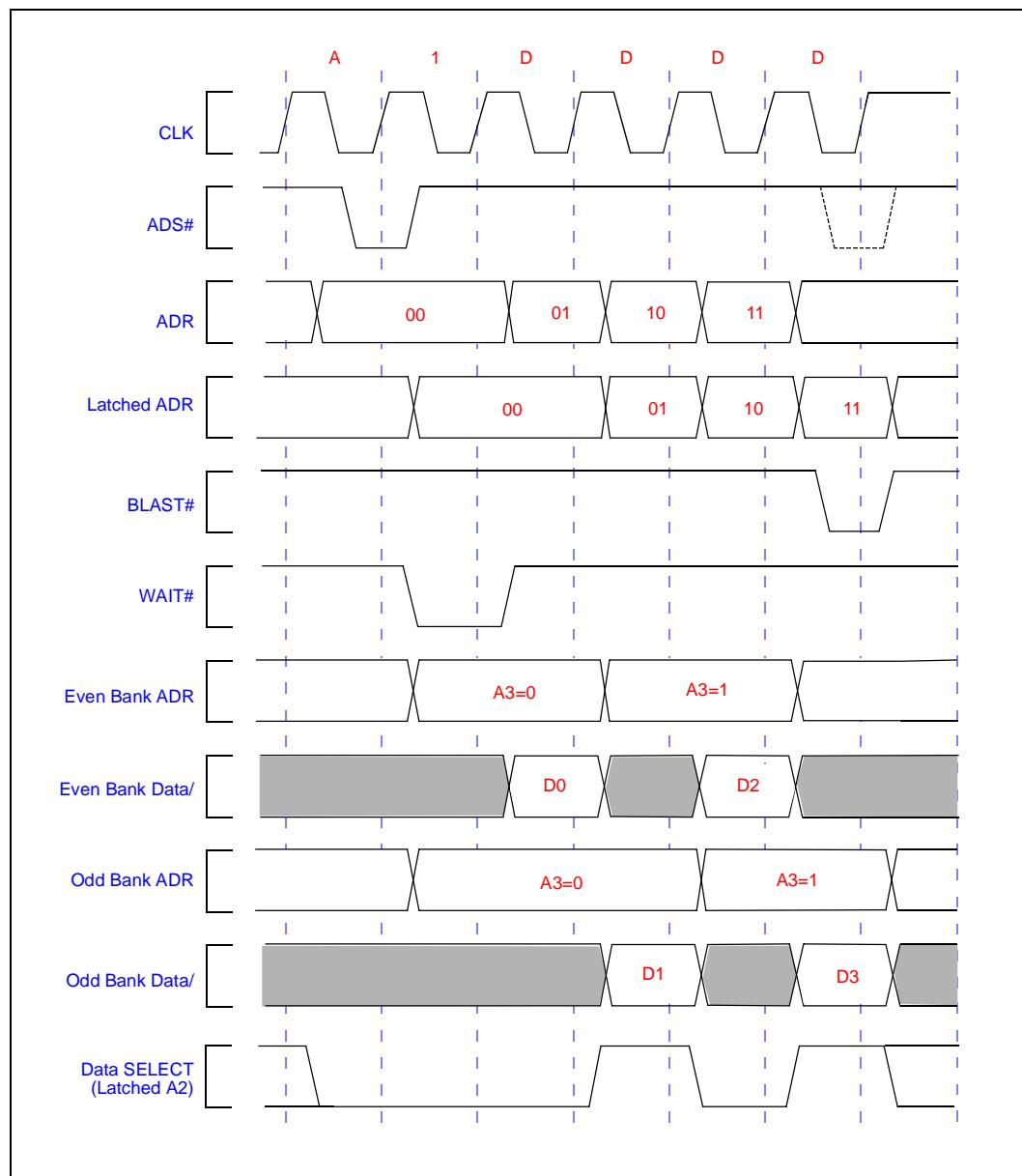
The A2 address determines which bank (even or odd word) is selected. [Figure F-22](#) shows the read waveform.

[Figure F-22](#) illustrates a memory system that interleaves read accesses. Write interleaving requires latching the written data and controlling memory access with the READY# signal. Write interleaving provides less performance improvement than read interleaving. Write data must come from the processor; this means a write interleaved system must queue data. The i960 Hx processor bus controller queues all access; therefore, write interleaving does not significantly benefit most applications.

Figure F-21. Two-Way Interleaved Memory System



Memory interleaving can be applied to SRAM, DRAM and even EPROM systems. Interleaved SRAM and EPROM systems overlap access times for consecutive accesses to improve memory system performance. The i960 Hx processor's pipelined read mode can be used on SRAM and EPROM systems to further increase memory system performance. However, pipelined read mode is not appropriate for DRAM systems that require  $N_{XDA}$  states or READY# control. Interleaved DRAM systems can overlap the memory access time and RAS# precharge time of consecutive accesses.

**Figure F-22. Two-Way Interleaved Read Waveforms**


**F.5**

## Interfacing to Slow Peripherals Using the Internal Wait State Generator

This section illustrates how easy it is to interface slow peripherals to the i960 Hx processor. The example shows the interface to a timer/counter and a UART. The integrated internal wait state generator, programmable data bus width and data transceiver control signals simplify the logic required to implement the interface.

A system may require several slower-speed peripherals; other peripherals may use the interface described here.

**F.5.1**

### Implementation

Both the timer/counter and UART have address, read, write and chip enable inputs and an 8-bit bidirectional data bus. The slow peripherals example considers only the memory-mapped interface to chip control registers. The timer/counter and UART are memory-mapped into a region programmed for non-burst, non-pipelined reads and an 8-bit data bus.

The RD# high to data float time dictates the number of  $N_{XDA}$  wait states required. Recovery time between reads or writes requires special treatment. The following example assumes a 33 MHz bus. The issues are the same at other operating frequencies.

**F.5.2**

### Schematic

The interface consists of chip select logic, a registered PLD with at least two combinatorial outputs and a data transceiver.

Chip select logic is the same as in previous examples. A simple demultiplexer is based only on the address. The PLD that controls access qualifies this signal with the address strobe (ADS#).

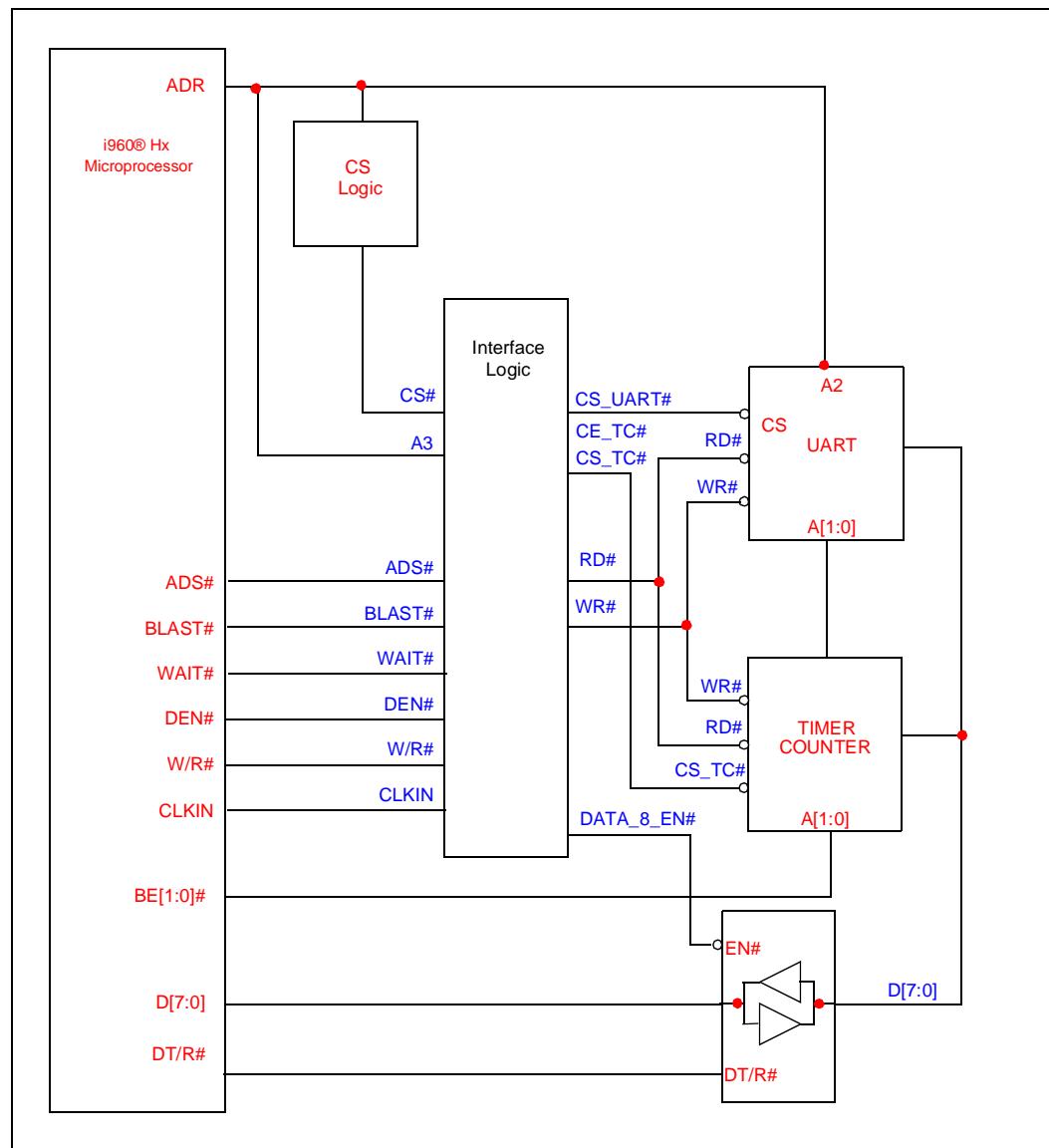
The state machine PLD generates chip enable, read and write signals for the UART and timer/counter. It also generates the data enable control for the data transceiver. The A3 address signal determines which peripheral is enabled.

The data transceiver is enabled by the PLD. The transceiver is activated when both the CS# and DEN# signals are asserted. The equation is:

$$\text{DATA\_8\_EN\#} = \text{CS\#} \mid \text{DEN\#}$$

Transceiver direction control is connected directly to the DT/R# signal of the i960 Hx processor. Data transceiver usage is optional; it is used here to reduce capacitive loading on the data bus. The i960 Hx processor can drive substantial capacitive loads; however, high-speed SRAM may have limited drive capabilities. If high-speed SRAM is on the data bus, it may be necessary to buffer the slower peripherals.

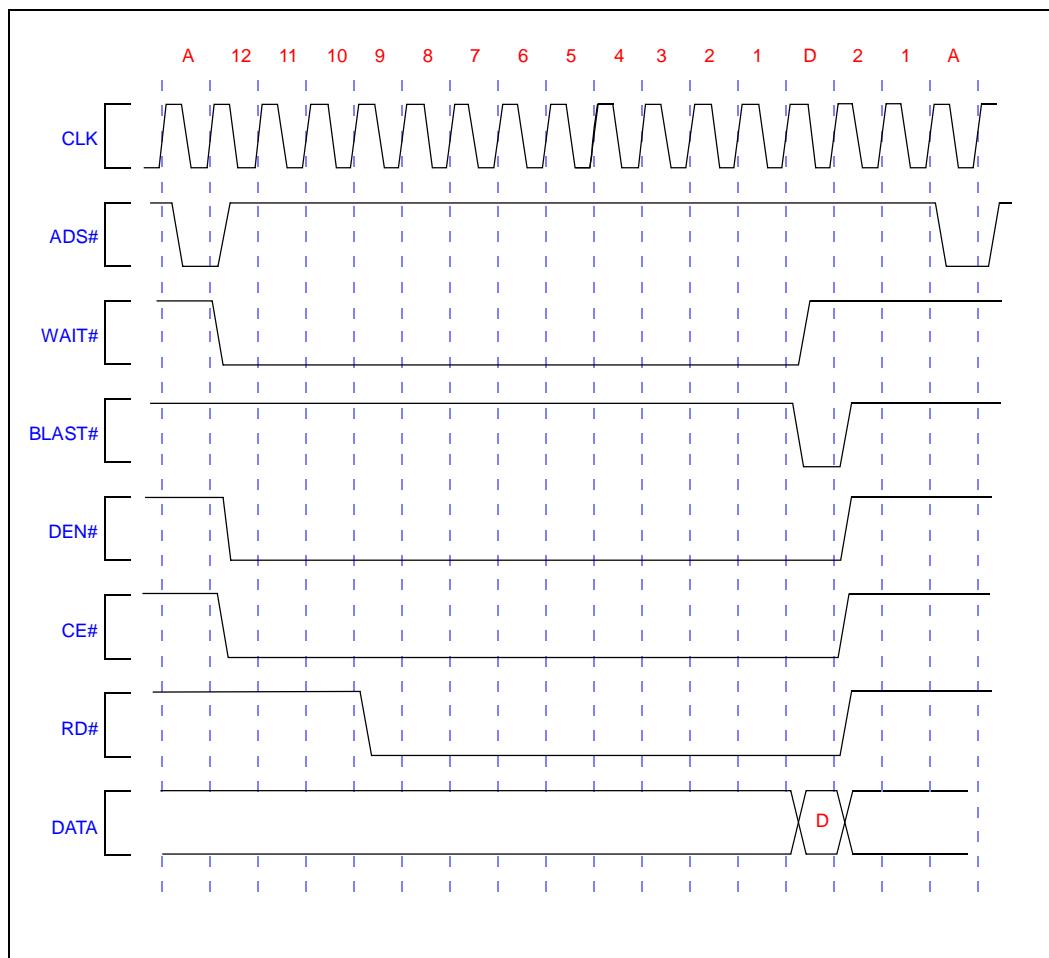
**Figure F-23. 8-bit Interface Schematic**



### F.5.3 Waveforms

The timer/counter and UART have long address setup times to read or write. They also have long read and write recovery times. This design uses a PLD to implement a state machine that delays the read or write signal. Delaying the read or write signal satisfies command recovery times. Using the internal wait state generator to determine the length of the overall read or write cycle adds flexibility and simplifies the state machine.

**Figure F-24. Read Waveforms**



Data lines are not driven during  $N_{XDA}$  wait states. This requires gating the W/R# signal with the WAIT# signal, so that W/R# goes high while the data is still asserted. There is a relative timing for output data hold after WAIT# goes high. The data hold requirement of the peripheral and the delay time to gate the write signal with WAIT# determines if this is an appropriate solution.

The state machine simply delays the read or write signal so that back-to-back commands to the peripheral satisfy the peripheral's command recovery time. When the write state is entered, the W/R# output of the PLD is a gated version of the WAIT# signal. This guarantees that the peripheral's write data hold time is satisfied.

**Figure F-25. Write Waveforms**

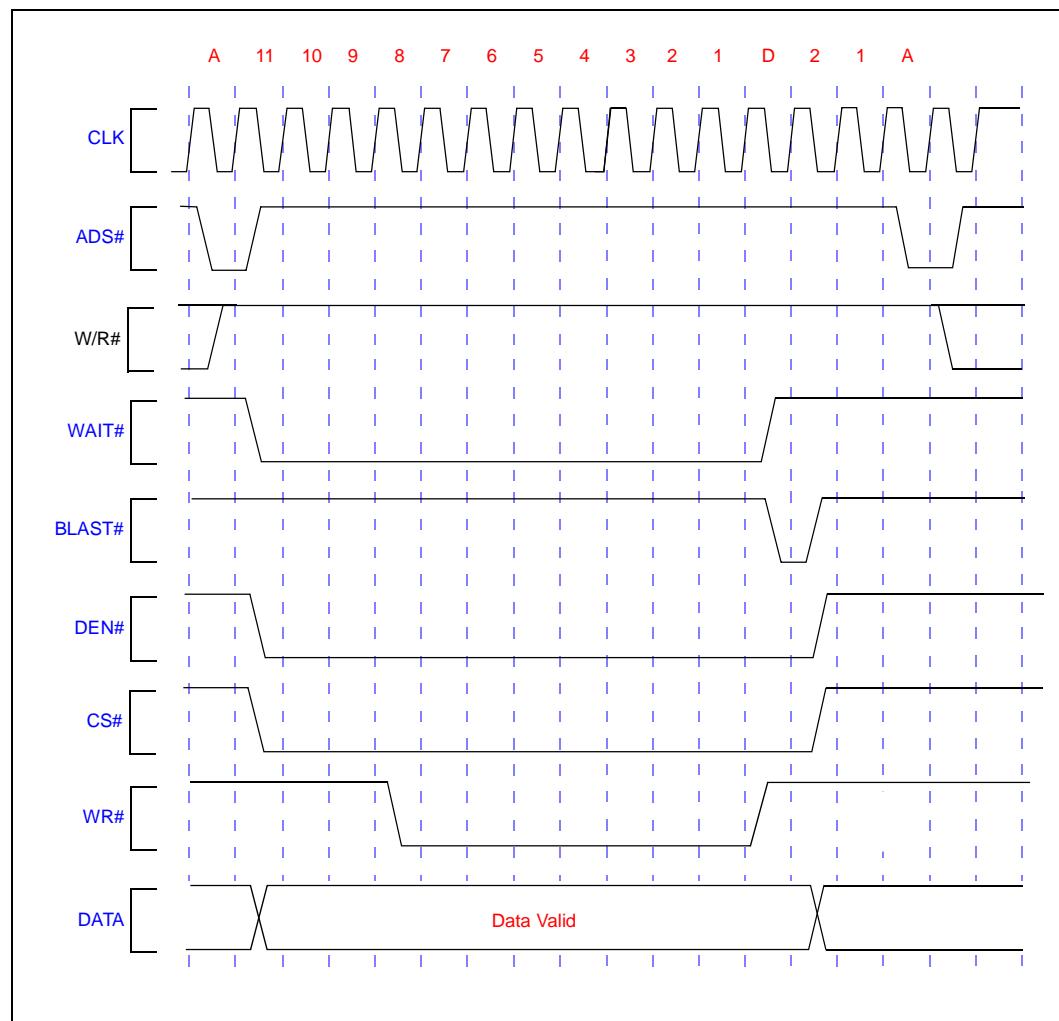
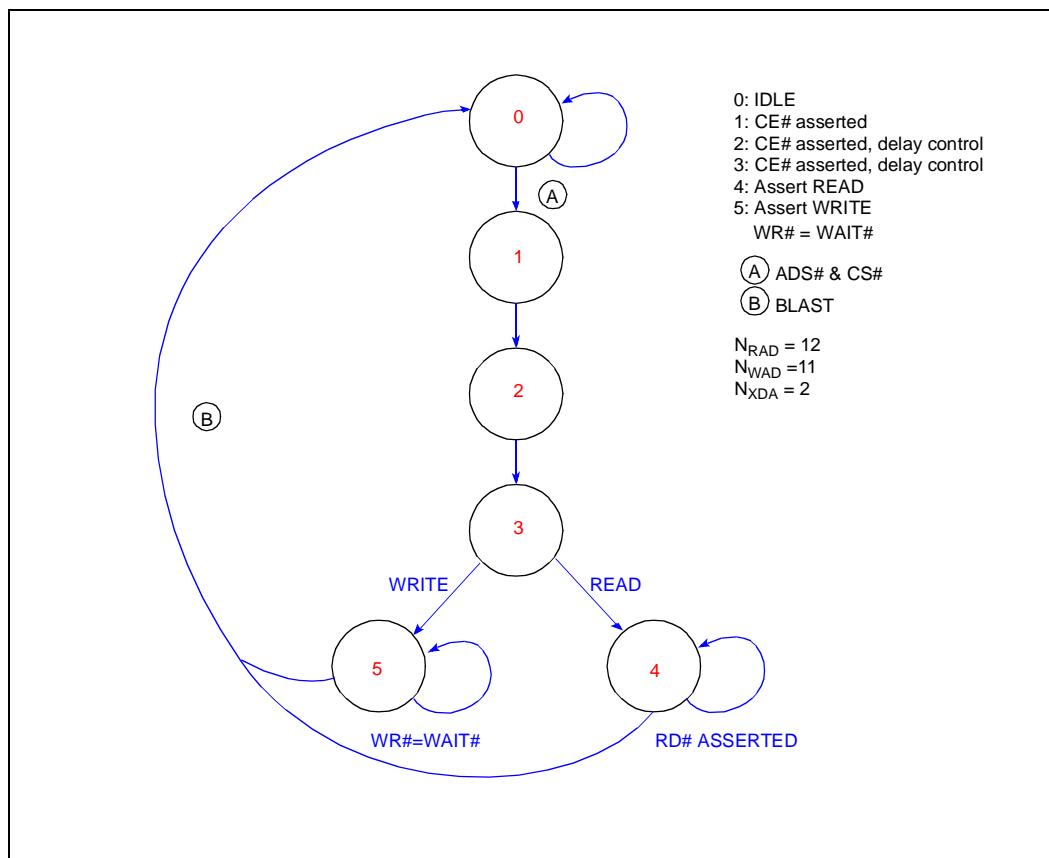


Figure F-26. State Machine Diagram



This pseudo-code example is provided only to describe the state machine diagram shown in Figure F-26. It is not intended for direct use as PLD equations.

```

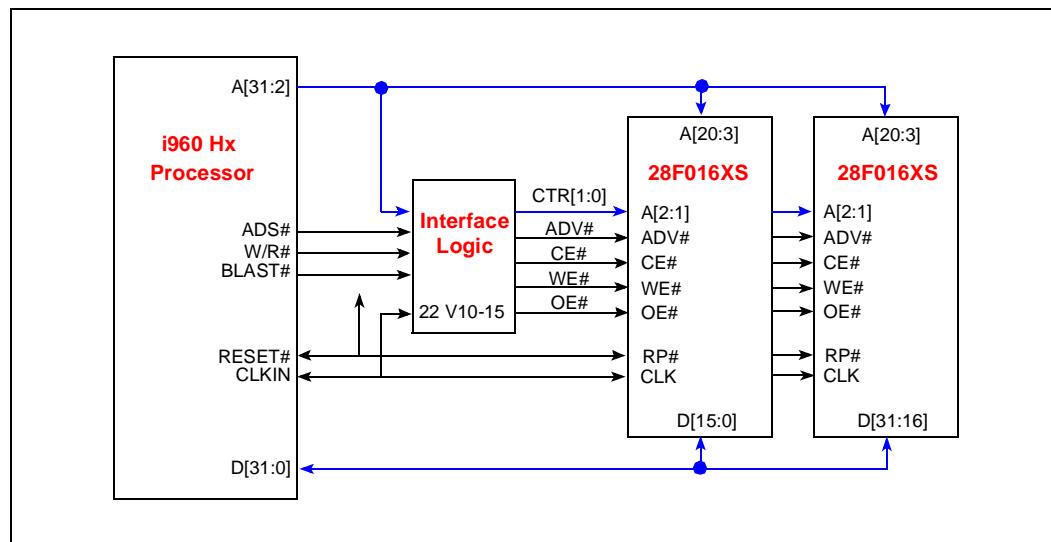
STATE_0:/*idle */
CE_UART      is not asserted;
CE_TC       is not asserted;
RD          is not asserted;
W/R         is not asserted;
IF          /* selected */
    ADS & CS;
THEN
    next state is STATE_1;
ELSE
    next state is STATE_0;
STATE_1:           /* Enable Selected Chip, Hold Off
                    Write or Read */
    CE_UART = A3;
    CE_TC = !A3;
    RD      is not asserted;
    W/R      is not asserted;
    the next state is state_2
STATE_2:           /* Enable Selected Chip, Hold Off
                    Write or Read */
    CE_UART = A3;
    CE_TC = !A3;
    RD      is not asserted;
    W/R      is not asserted;
    the next state is state_3
STATE_3:           /* Enable Selected Chip, Hold Off
                    Write or Read */
    CE_UART = A3;
    CE_TC = !A3;
    RD      is not asserted;
    W/R      is not asserted;
IF
    !READ           /* read */
THEN
    next state is STATE_4;
ELSE
    /* write */
    next state is STATE_5;
STATE_4:           /* Read asserted to
                    selected peripheral */
    CE_UART = A3;
    CE_TC = !A3;
    RD      is asserted;
    W/R      is not asserted;
IF
    BLAST           /* Done */
THEN
    next state is STATE_0;
ELSE
    /* write */
    next state is STATE_4;
STATE_5:           /* Write asserted to selected peripheral */
    CE_UART = A3;
    CE_TC = !A3;
    RD      is not asserted;
    W/R = WAIT#
IF
    BLAST           /* Done */
THEN
    next state is STATE_0;
ELSE
    /* write */
    next state is STATE_5;

```

## F.6 Synchronous Flash Interface

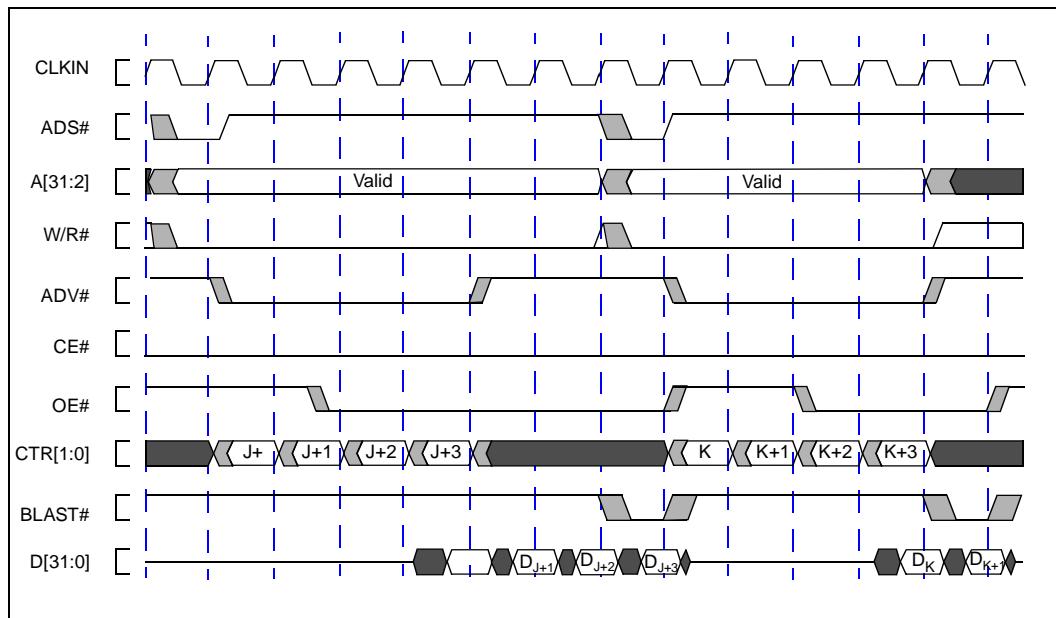
By using 28F016XS synchronous flash memory, one of Intel's embedded flash RAM components, the user can easily construct a high performance memory subsystem for the i960 Hx processor. The design shown in [Figure F-27](#) provides a pipelined 2-0-0-0 wait-state memory subsystem ideal for code storage and execution. In doing so, the 28F016XS replaces redundant volatile execution RAM and nonvolatile storage memories. The interface logic, managing the CPU-to-memory communication, supports burst transfers and address pipelining and fits easily within an inexpensive 22V10-15.

[Figure F-27. Flash Memory System Block Diagram](#)



Upon power-up or reset, the i960 Hx processor, interface logic and 28F016XSs default to an initial configuration. The processor's memory region configuration register (PMCON0) setting disables bursting and address pipelining and sets the wait-states to 5. The processor remains in this state until system software optimizes 28F016XSs configuration register (SFI Configuration = 2) for 33 MHz operation and sets the interface logic's internal configuration bit. Then, the i960 Hx processor's PMCON0 setting can change to enable bursting, address pipelining and burst wait-state control equal to 3-0-0-0 (enabling 2-0-0-0 pipelined reads).

**Figure F-28. Four Double-Word Burst Followed by a Pipelined Two Double-Word Burst Read**



In the initial configuration, the interface logic monitors ADS# and W/R# to identify the start and type of a bus transaction. Upon detecting ADS# active, the interface drives ADV# (Address Valid) active and loads a two-bit counter within the interface logic with the processor's lower address lines (A[3:2]). The counter then drives the flash memory's lower address lines, A[2:1].

Interface logic holds ADV# active for one CLK cycle, thereby supplying the flash with only one read cycle. After driving ADV#, the interfacing state machine waits for the processor to assert BLAST#, then transitions to an inactive state where it waits for the start of a new bus transaction.

The optimized interface configuration drives ADV# active for four consecutive CLK cycles or until BLAST# is sampled active (see [Figure F-28](#)). While ADV# is active, the counter increments through i960 Hx processor's burst order in anticipation of a four double-word burst transaction, keeping the flash's internal pipe full. As the access transitions into its data phase, the state machine examines ADS# to identify the start of a pipelined read access. If a pipelined access is detected, the interface logic immediately drives ADV# active and loads the two-bit counter, kicking off a new transaction. If ADS# is not sampled active during the last data phase, the interface logic transitions to an inactive state that is similar to the initial configuration.

During pipelined reads, the i960 Hx processor outputs the next burst cycle address in the last data phase of the present cycle. Therefore, the flash memory's CE# signal must be latched to preserve its value as the address bus changes. To accomplish this task, a simple one-bit state machine within the interface logic latches the status of CE#. When the bus is idle, the state machine holds CE# active in anticipation of a flash memory access. CE# remains active until a bus transaction targeting some memory or I/O location other than synchronous flash memory is detected.





## Glossary

---

Address Space	An array of bytes used to store program code, data, stacks and system data structures required to execute a program. Address space is <i>linear</i> – also called <i>flat</i> – and byte addressable, with addresses running contiguously from 0 to $2^{32} - 1$ . It can be mapped to read-write memory, read-only memory and memory-mapped I/O. i960 architecture does not define a dedicated, addressable I/O space.
Address	A 32-bit value in the range 0 to FFFF FFFFH used to reference in memory a single byte, half-word (2 bytes), word (4 bytes), double-word (8 bytes), triple-word (12 bytes) or quad-word (16 bytes). Choice depends on the instruction used.
Arithmetic Controls (AC) Register	A 32-bit register that contains flags and masks used in controlling the various arithmetic and comparison operations that the processor performs. Flags and masks contained in this register include the condition code flags, integer-overflow flag and mask bit and the no-imprecise-faults (NIF) bit. All unused bits in this register are reserved and must be set to 0.
Asynchronous Faults	Faults that occur with no direct relationship to a particular instruction in the instruction stream. When an asynchronous fault occurs, the address of the faulting instruction in the fault record and the saved IP are undefined. i960 core architecture does not define any fault types that are asynchronous.
Big Endian	The bus controller reads or writes a data word's least-significant byte to the bus' eight most-significant data lines (D31:24). Big endian systems store the least-significant byte at the highest byte address in memory. So, if a big endian ordered word is stored at address 600, the least-significant byte is stored at address 603 and the most-significant byte at address 600. Compare with little endian.
Condition Code Flags	AC register bits 0, 1 and 2. The condition code flags indicate the results of certain instructions – usually compare instructions. Other instructions, such as conditional branch instructions, examine these flags and perform functions according to their state. Once the processor sets the condition code flags, they remain unchanged until the processor executes another instruction that uses these flags to store results.
Execution Mode Flag	PC register bit 1. This flag determines whether the processor is operating in user mode (0) or supervisor mode (1).
Fault Call	An implicit call to a fault handling procedure. The processor performs fault calls automatically without any intervention from software. It gets pointers to fault handling procedures from the fault table.
Fault Table	An architecture-defined data structure that contains pointers to fault handling procedures. Each fault table entry is associated with a particular fault type. When the processor generates a fault, it uses the fault table to select the proper fault handling procedure for the type of fault condition detected.

Fault	An event that the processor generates to indicate that, while executing the program, a condition arose that could cause the processor to go down a wrong and possibly disastrous path. One example of a fault condition is a divisor operand of zero in a divide operation; another example is an instruction with an invalid opcode.
Frame Pointer (FP)	The address of the first byte in the current (topmost) stack frame of the procedure stack. The FP is contained in global register g15.
Frame	See Stack Frame.
Global Registers	A set of 16 general-purpose registers (g0 through g15) whose contents are preserved across procedure boundaries. Global registers are used for general storage of data and addresses and for passing parameters between procedures.
Guarded Memory Unit (GMU)	A section of the processor that monitors all of the processor's memory transactions and can prevent accesses to predefined address regions or warn the user program if accesses occur.
Hardware Reset	The assertion of the RESET# pin; equivalent to powerup.
IBR	See Initialization Boot Record.
IMI	See Initial Memory Image.
Imprecise Faults	Faults that are allowed to be generated out-of-order from where they occur in the instruction stream. When an imprecise fault is generated, the processor indicates the address of the faulting instruction, but it does not guarantee that software can recover from the fault and resume execution of the program with no break in the program's control flow. The NIF bit in the arithmetic controls register determines whether all faults must be precise (1) or some faults are allowed to be imprecise (0).
Initialization Boot Record (IBR)	One of three IMI components, IBR is the primary data structure required to initialize the processor. IBR is 12-word structure which must be located at address FFFF FF00H.
Initial Memory Image (IMI)	Comprises the minimum set of data structures the processor needs to initialize its system. Performs three functions for the processor: 1) provides initial configuration information for the core and integrated peripherals; 2) provides pointers to system data structures and the first instruction to be executed after processor initialization; 3) provides checksum words that the processor uses in self-test at startup. See also IBR, PRCB and System Data Structures.
Instruction Cache	A memory array used for temporary storage of instructions fetched from main memory. Its purpose is to streamline instruction execution by reducing the number of instruction fetches required to execute a program.
Instruction Pointer (IP)	A 32-bit register that contains the address (in the address space) of the instruction currently being executed. Since instructions are required to be aligned on word boundaries in memory, the IP's two least-significant bits are always zero.
Integer Overflow Flag	AC register bit 8. When integer overflow faults are masked, the processor sets the integer overflow flag whenever integer overflow occurs to indicate that the fault condition has occurred even though the fault has been masked. If the fault is not masked, the fault is allowed to occur and the flag is not set.

Integer Overflow Mask Bit	AC register bit 12. This bit masks the integer overflow fault.
Interrupt Call	An implicit call to a interrupt handling procedure. The processor performs interrupt calls automatically without any intervention from software. It gets vectors (pointers) to interrupt handling procedures from the interrupt table.
Interrupt Stack	Stack the processor uses when it executes interrupt handling procedures.
Interrupt Table	A data structure that contains vectors to interrupt handling procedures and fields for storing pending interrupts. When the processor receives an interrupt, it uses the vector number that accompanies the interrupt to locate an interrupt vector in the interrupt table. The interrupt table's pending interrupt fields contain bits that indicate priorities and vector numbers of interrupts waiting to be serviced.
Interrupt Vector	A pointer to an interrupt handling procedure. In the i960 architecture, interrupts vectors are stored in the interrupt table.
Interrupt	An event that causes program execution to be suspended temporarily to allow the processor to handle a more urgent chore.
Leaf Procedure	Leaf procedures call no other procedures. They are called “leaf procedures” because they reside at the “leaves” of the call tree.
Literals	A set of 32 ordinal values ranging from 0 to 31 (5 bits) that can be used as operands in certain instructions.
Little Endian	The bus controller reads or writes a data word’s least-significant byte to the bus’ eight least-significant data lines (D7:0). Little endian systems store a word’s least-significant byte at the lowest byte address in memory. For example, if a little endian ordered word is stored at address 600, the least-significant byte is stored at address 600 and the most-significant byte at address 603. Compare with big endian.
Local Call	A procedure call that does not require a switch in the current execution mode or a switch to another stack. Local calls can be made explicitly through the <b>call</b> , <b>callx</b> and <b>calls</b> instructions and implicitly through the fault call mechanism.
Local Registers	A set of 16 general-purpose data registers (r0 through r15) whose contents are associated with the procedure currently being executed. Local registers hold the local variables for a procedure. Each time a procedure is called, the processor automatically allocates a new set of local registers for that procedure and saves the local registers for the calling procedure.
Memory	Array to which address space is mapped. Memory can be read-write, read-only or a combination of the two. A memory address is generally synonymous with an address in the address space.
Memory-Mapped Register (MMR)	A 32-bit register located in memory used to control specific sections of the processor. All MMRs reside inside the processor. These registers can be manipulated like any other register, but their contents affect the processor’s behavior directly.

“Natural” Fill Policy	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.
No Imprecise Faults (NIF) Bit	AC register bit 15. This flag determines whether or not imprecise faults are allowed to occur. If set, all faults are required to be precise; if clear, certain faults can be imprecise.
Non Maskable Interrupt (NMI)	Provides an interrupt that cannot be masked and has a higher priority than priority-31 interrupts and priority-31 process priority. The core services NMI requests immediately.
Parallel Faults	A condition which occurs when multiple execution units, executing instructions in parallel, report multiple faults simultaneously. Setting the NIF bit prohibits execution conditions which could cause parallel faults.
Pending Interrupt	An interrupt that the processor saves to be serviced at a later time. When the processor receives an interrupt, it compares the interrupt's priority with the priority of the current processing task. If the priority of the interrupt is equal to or less than that of the current task, the processor saves the interrupt's priority and vector number in the pending interrupt fields of the interrupt table, then continues work on the current processing task.
PFP	See Previous Frame Pointer.
Pointer	An address in the address space (or memory). The term pointer generally refers to the first byte of a procedure or data structure or a specific byte location in a stack.
PRCB	See Process Control Block.
Precise Faults	Faults generated in the order in which they occur in the instruction stream and with sufficient fault information to allow software to recover from the faults without altering program's control flow. The AC register NIF bit and the <b>syncf</b> instruction allow software to force all faults to be precise.
Previous Frame Pointer (PFP)	The address of the previous stack frame's first byte. It is contained in bits 4 through 31 of local register r0.
Priority Field	PC register bits 16 through 20. This field determines processor priority (from 0 to 31). When the processor is in the executing state, it sets its priority according to this value. It also uses this field to determine whether to service an interrupt immediately or to save the interrupt for later service.
Priority	A value from 0 to 31 that indicates the priority of a program or interrupt; highest priority is 31. The processor stores the priority of the task (program or interrupt) that it is currently working on in the priority field of the PC register. See also NMI.
Process Control Block (PRCB)	One of three (IMI) components, PRCB contains base addresses for system data structures and initial configuration information for the core and integrated peripherals.
Process Controls (PC) Register	A 32-bit register that contains miscellaneous pieces of information used to control processor activity and show current processor state. Flags and fields in this register include the trace enable bit, execution mode flag, trace fault pending flag, state flag, priority field and internal state field. All unused bits in this register are reserved and must be set to 0.

Register Scoreboarding	Internal flags that indicate a particular register or group of registers is being used in an operation. This feature enables the processor to execute some instructions in parallel and out-of-order. When the processor begins executing an instruction, it sets the scoreboard flag for the destination register in use by that instruction. If the instructions that follow do not use scoreboarded registers, the processor can execute one or more of those instructions concurrently with the first instruction.
Return Instruction Pointer (RIP)	The address of the instruction following a call or branch-and-link instruction that the processor is to execute after returning from the called procedure. The RIP is contained in local register r2. When the processor executes a procedure call, it sets the RIP to the address of the instruction immediately following the procedure call instruction.
Return Type Field	Bits 0, 1 and 2 of local register r0. When a procedure call is made using the integrated call and return mechanism, this field indicates the call type: local, supervisor, interrupt or fault. The processor uses this information to select the proper return mechanism when returning from the called procedure.
RIP	See Return Instruction Pointer.
Software Reset	Re-running of the Reset microcode without physically asserting the RESET# pin or removing power from the CPU.
SP	See Stack Pointer.
Special Function Registers (SFRs)	A 32-bit register (sf0-sf4) used to control specific sections of the processor. These registers can be manipulated like any other register, but their contents affect the processor's behavior directly.
Stack Frame	A block of bytes on a stack used to store local variables for a specific procedure. Another term for a stack frame is an <i>activation record</i> . Each procedure that the processor calls has its own stack frame associated with it. A stack frame is always aligned on a 64-byte boundary. The first 64 bytes in a stack frame are reserved for storage of the local registers associated with the procedure. The frame pointer (FP) and stack pointer (SP) for a particular frame indicate location and boundaries of a stack frame within a stack.
Stack Pointer (SP)	The address of the last byte in the current (topmost) frame of the procedure stack. The SP is contained in local register r1.
Stack	A contiguous array of bytes in the address space that grows from low addresses to high addresses. It consists of contiguous frames, one frame for each active procedure. i960 architecture defines three stacks: local, supervisor and interrupt.
State Flag	PC register bit 10. This flag indicates to software that the processor is currently executing a program (0) or servicing an interrupt (1).
State	The type of task that the processor is currently working on: a program or an interrupt handling procedure. The processor sets the PC register state flag to indicate its current state.
Status and Control Registers	A set of four 32-bit registers that contain status and control information used in controlling program flow. These registers include the instruction pointer (IP), AC register, PC register and TC register.

Supervisor Call	A system call (made with the <b>calls</b> instruction) where the entry type of the called procedure is 102. If the processor is in user mode when a supervisor call is made, it switches to the supervisor stack and to supervisor mode.
Supervisor Mode	One of two execution modes – user and supervisor – that the processor can use. The processor uses the supervisor stack when in supervisor mode. Also, while in supervisor mode, software is allowed to execute supervisor mode instructions such as <b>sysctl</b> and <b>modpc</b> .
Supervisor Stack Pointer	The address of the first byte of the supervisor stack. The supervisor stack pointer is contained in bytes 12 through 15 of the system procedure table and the trace table.
Supervisor Stack	The procedure stack that the processor uses when in supervisor mode.
System Call	An explicit procedure call made with the <b>calls</b> instruction. The two types of system calls are a system-local call and system-supervisor call. On a system call, the processor gets a pointer to the system procedure through the system procedure table.
System Data Structures	One of three IMI components. The following system data structures contain values the processor requires for initialization: PRCB, IBR, system procedure table, control table, interrupt table.
System Procedure Table	An architecturally-defined data structure that contains pointers to system procedures and (optionally) to fault handling procedures. It also contains the supervisor stack pointer and the trace control flag.
Trace Table	An architecturally-defined data structure that contains pointers to trace-fault-handling procedures. The trace table has the same structure as the system procedure table.
Trace Control Bit	Bit 0 of byte 12 of the system procedure table. This bit specifies the new value of the trace enable bit when a supervisor call causes a switch from user mode to supervisor mode. Setting this bit to 1 enables tracing; setting it to 0 disables tracing.
Trace Controls (TC) Register	A 32-bit register that controls processor tracing facilities. This register contains one event bit and one mode bit for each trace fault subtype (i.e., instruction, branch, call, return, prereturn, supervisor and breakpoint). The mode bits enable the various tracing modes; the event flags indicate that a particular type of trace event has been detected. All the unused bits in this register are reserved and must be set to 0.
Trace Enable Bit	PC register bit 0. This bit determines whether trace faults are to be generated (1) or not generated (0).
Trace Fault Pending Flag	PC register bit 10. This flag indicates that a trace event has been detected (1) but not yet generated. Whenever the processor detects a trace fault at the same time that it detects a non-trace fault, it sets the trace fault pending flag then calls the fault handling procedure for the non-trace fault. On return from the fault procedure for the non-trace fault, the processor checks the trace fault pending flag. If set, it generates the trace fault and handles it.
Tracing	The ability of the processor to detect execution of certain instruction types, such as branch, call and return. When tracing is enabled, the processor generates a fault whenever it detects a trace event. A trace fault handler can then be designed to call a debug monitor to provide information on the trace event and its location in the instruction stream.

User Mode	One of two execution modes – user and supervisor – that the processor can be in. When the processor is in user mode, it uses the local stack and is not allowed to use the <b>modpc</b> instruction or any other implementation-defined instruction that is designed to be used only in supervisor mode.
Vector Number	The number of an entry in the interrupt table where an interrupt vector is stored. The vector number also indicates the priority of the interrupt.
Vector	See Interrupt Vector.





## A

absolute  
    displacement addressing mode 2-7  
    memory addressing mode 2-7  
    offset addressing mode 2-7  
**AC** 3-21  
AC register, see Arithmetic Controls (AC) register  
access faults 3-7  
access types  
    restrictions 3-7  
**ADD** 6-7  
add  
    conditional instructions 6-7  
    integer instruction 6-11  
    ordinal instruction 6-11  
    ordinal with carry instruction 6-10  
**addc** 6-9  
**addi** 6-11  
**addie** 6-7  
**addig** 6-7  
**addige** 6-7  
**addil** 6-7  
**addile** 6-7  
**addine** 6-7  
**addino** 6-7  
**addio** 6-7  
**addo** 6-11  
**addoe** 6-7  
**addog** 6-7  
**addoge** 6-7  
**addol** 6-7  
**adbole** 6-7  
**addone** 6-7  
**addono** 6-8  
**addoo** 6-7  
Address Generation Unit (AGU) E-6, E-21  
address space restrictions  
    data structure alignment A-3  
    instruction cache A-2  
    internal data RAM A-2  
    reserved memory A-2  
    stack frame alignment A-3  
addressing mode  
    examples 2-8  
    register indirect 2-7  
addressing registers and literals 3-5  
alignment, registers and literals 3-5  
**alterbit** 6-12  
**and** 6-13  
**andnot** 6-13  
argument list 7-13  
Arithmetic Controls (AC) Register 3-21  
Arithmetic Controls (AC) register 3-21  
    condition code flags 3-22  
    initial image 13-20  
    initialization 3-22  
    integer overflow flag 3-23  
    integer overflow mask bit 3-23  
    no imprecise faults bit 3-23

## arithmetic instructions 5-7

    add, subtract, multiply or divide 5-8  
    extended-precision instructions 5-10  
    remainder and modulo instructions 5-8  
    shift and rotate instructions 5-9  
arithmetic operations and data types 5-7  
**atadd** 3-16, 4-10, 6-14  
**atmod** 3-8, 3-16, 4-10, 6-15  
atomic access 3-16  
atomic add instruction 6-14  
atomic instructions 5-19  
    (LOCK# signal) 15-31  
atomic modify instruction 6-15

## B

**b** 6-16  
**bal** 6-17  
**balx** 6-17  
**bbc** 6-19  
**bbs** 6-19  
**BCON** 14-10  
BCON register, see Bus Control (BCON) register  
BCU, see Bus Control Unit (BCU)  
**be** 6-21  
**bg** 6-21  
**bge** 3-23, 6-21  
big endian byte order 2-4, 15-29  
bit definition 1-9  
bit field instructions 5-11  
bit instructions 5-11  
bit ordering 2-4  
bit values  
    naming conventions 1-8  
bits and bit fields 2-3  
**bl** 6-21  
**ble** 6-21  
**bne** 6-21  
**bno** 6-21  
**bo** 6-21  
BOFF#, see bus backoff (BOFF#) signal  
boundary conditions  
    internal memory-mapped locations 14-10, 14-16  
    LMT boundaries 14-16  
    logical data template ranges 14-16  
boundary-scan register 16-8  
boundary-scan (JTAG) 16-1  
    architecture 16-2  
    test logic 16-3  
**BPCON** 9-7  
branch  
    and link extended instruction 6-17  
    and link instruction 6-17  
    check bit and branch if clear set instruction 6-19  
    check bit and branch if set instruction 6-19  
    conditional instructions 6-21  
    extended instruction 6-16  
    instruction 6-16

branch instructions, overview 5-14  
     compare and branch instructions 5-16  
     conditional branch instructions 5-15  
     unconditional branch instructions 5-15  
 branch prediction 5-14  
 branch-and-link 7-1  
     returning from 7-21  
 branch-and-link instruction 7-1  
 branch-if-greater-or-equal instruction 3-23  
 breakpoint  
     registers A-7  
     resource request message 9-6  
 Breakpoint Control Register (BPCON) 9-7  
 Breakpoint Control (BPCON) register 9-7, D-4  
     programming 9-8  
 BREQ, see Bus Request (BREQ) signal  
 BSTALL, see Bus Stall (BSTALL) signal  
**bswap** 6-23  
 Built-In Self Test (BIST) 13-2  
 burst access 14-2  
 bus access 15-2  
 bus backoff (BOFF#) signal 15-34  
 bus confidence self test 13-8  
 Bus Configuration (BCON) register  
     RAM protection bit 3-19  
 Bus Control Register (BCON) 14-10  
 Bus Control Unit (BCU) 14-1, E-22  
     boundary conditions 14-10  
     loads E-22  
     memory attributes 14-1  
     physical memory attributes 14-7  
     PMCON initialization 14-9  
     queue entries E-23  
     stores E-22  
     wait state generator 14-2  
 Bus Control (BCON) register 14-9  
     BCON.ipr bit 4-2  
     BCON.sirp bit 4-1  
 Bus Request (BREQ) signal 15-34  
 bus requests 15-2  
 bus snooping 4-6, 4-10  
 Bus Stall (BSTALL) signal 15-34  
 bus width 15-22  
**bx** 6-16  
 byte instructions 5-12  
 byte order, little or big endian 2-4, 15-29  
 byte swap instruction 6-23

## C

cache  
     data 3-20  
         cache coherency and non-cacheable accesses 4-10  
         described 4-6  
         enabling and disabling 4-7  
         fill policy 4-8  
         invalidating 4-11  
         partial-hit multi-word data accesses 4-8  
         visibility 4-11  
         write policy 4-9

instruction 3-19  
     enabling and disabling 4-5  
     invalidation 3-19  
     loading and locking instruction 4-5  
     visibility 4-5  
     load-and-lock mechanism 4-5  
     local register 3-18, 4-3  
     stack frame 4-3  
 Cache Control Register (CCON) 4-7  
 Cache Control (CCON) register 4-7  
     CCON.dcgd bit 4-7  
     CCON.dci bit 4-7  
 cacheable writes (stores) 4-9  
 caching of interrupt-handling procedure 11-33  
 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 6-27  
     instruction 6-24  
     system instruction 6-25  
**call** 6-24, 7-2, 7-6  
 call and return instructions 5-17  
 call and return mechanism 7-1, 7-2  
     explicit calls 7-1  
     implicit calls 7-1  
     local register cache 7-3  
     local registers 7-2  
     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 7-6  
     call operation 7-6  
     return operation 7-7  
**calls** 3-27, 6-25, 7-2, 7-6  
 call-trace mode 9-3  
**callx** 6-27, 7-2, 7-6  
 carry conditions 3-22  
 CCON 4-7  
 check bit instruction 6-28  
**chkbit** 6-28  
 clear bit instruction 6-29  
 CLKIN 15-4  
 clock input (CLKIN) 13-35  
 clock rate  
     multiplying 15-4  
**clrbit** 6-29  
**cmpdeci** 6-30  
**cmpdeco** 6-30  
**cmpi** 5-12, 6-32  
**cmpib** 5-12  
**cmpibe** 6-34  
**cmpibg** 6-34  
**cmpibge** 6-34



**cmpibl** 6-34  
**cmpible** 6-34  
**cmpibne** 6-34  
**cmpibno** 6-34  
**cmpibo** 6-34  
**cmpincli** 6-31  
**cmpinco** 6-31  
**cmpis** 5-12  
**cmipo** 5-12, 6-32  
**cmpobe** 6-34  
**cmpobg** 6-34  
**cmpobge** 6-34  
**cmpobl** 6-34  
**cmpoble** 6-34  
**cmpobne** 6-34  
coding optimizations  
    branch prediction E-42  
    branch target alignment E-42  
    comparison and branching E-36  
    compressing algorithms using branching E-43  
    data cache E-44  
    data RAM E-45  
    instruction cache E-44  
    loads and stores E-35  
    loop expansion E-37  
    maximizing instruction execution E-38  
    multiplication and division E-35  
    on-chip storage E-43  
    register cache E-45  
    reordering code for parallel issue E-38  
cold reset 11-25, 13-4  
compare  
    and branch conditional instructions 6-34  
    and conditional compare instructions 5-12  
    and decrement integer instruction 6-30  
    and decrement ordinal instruction 6-30  
    and increment integer instruction 6-31  
    and increment ordinal instruction 6-31  
    integer conditional instruction 6-36  
    integer instruction 6-32  
    ordinal conditional instruction 6-36  
    ordinal instruction 6-32  
comparison instructions, overview  
    compare and increment or decrement instructions 5-13  
    test condition instructions 5-13  
**concmpli** 6-36  
**concmopo** 6-36  
conditional branch instructions 3-22  
conditional fault instructions 5-18  
control registers 3-1, 3-7  
    memory-mapped 3-6  
control table 3-1, 3-7, 3-14  
    alignment 3-17  
control table valid (BCON.ctv) bit 14-9  
core architecture  
    and performance optimization E-1  
    and software portability A-1  
CTRL pipeline  
conditional branches E-27  
unconditional branches E-24  
Cycle Type pins 15-26

## D

DAB 9-9  
Data Address Breakpoint (DAB) Register Format 9-9  
Data Address Breakpoint (DAB) registers 9-8  
    programming 9-8  
data alignment in external memory 3-16  
data bus parity 14-4  
data cache 3-20, E-7  
    BCU interaction E-9  
    bus configuration E-8  
    cache coherency and non-cacheable accesses 4-10  
    coherency 14-14, E-9  
    BCU queues E-11  
    I/O and bus masters 4-10, E-11  
    control instruction 6-38  
    data fetch policy E-8  
    described 4-6  
    enabling and disabling 4-7  
    fill policy 4-8  
    hits and misses E-7  
    invalidating 4-11  
    organization E-7  
    partial-hit multi-word data accesses 4-8  
    subblock placement E-7  
    visibility 4-11  
    write policy 4-9, E-8  
Data Cache Enable (DCEN) bit 14-15  
data cache enable (LMAR.dcen) bit 14-15  
data cache global disable (CCON.dcgd) bit 4-7  
data cache invalidate (CCON.dci) bit 4-7  
data control peripheral units A-6  
data fetch policy E-8  
data movement instructions 5-5  
    load address instruction 5-6  
    load instructions 5-5  
    move instructions 5-6  
data parity signals 15-24  
data RAM 3-18, E-7, E-20  
data register  
    timing diagram 16-20  
data structures  
    control table 3-1, 3-7, 3-14  
    fault table 3-1, 3-14  
    Initialization Boot Record (IBR) 3-1, 3-14  
    interrupt stack 3-1, 3-14  
    interrupt table 3-1, 3-14  
    literals 3-5  
    local stack 3-1  
    Process Control Block (PRCB) 3-1, 3-14  
    supervisor stack 3-1, 3-14  
    system procedure table 3-1, 3-14  
    user stack 3-14

- data types  
 bits and bit fields 2-3  
 integers 2-2  
 literals 2-4  
 ordinals 2-2  
 supported 2-1  
 triple and quad words 2-3
- dctl** 3-27, 4-6, 4-7, 4-11, 6-38
- DCEN bit, see Data Cache Enable (DCEN) bit
- dcen bit, see data cache enable (LMAR.dcen) bit
- dcinva** 4-11
- debug  
 overview 9-1  
 debug instructions 5-18  
 decoupling capacitors 13-37  
 Default Logical Memory Configuration Register (DLMCON) 14-13  
 Default Logical Memory Configuration (DLMCON) register 14-4  
 DLMCON.be bit 4-4  
 design considerations  
 high frequency 13-39  
 interference 13-41  
 latchup 13-40  
 line termination 13-39  
 performance E-1  
 detection scheme, GMU  
 described 12-3
- Device ID register 16-7
- device ID register D-6
- DEVICEID** 13-23
- DEVICEID register location 3-3
- divi** 6-46
- divide integer instruction 6-46
- divide ordinal instruction 6-46
- divo** 6-46
- DLMCON 14-13
- DLMCON registers
- DLMCON, see Default Logical Memory Configuration (DLMCON) register
- E**
- ediv** 6-47
- effective address (efa) E-21  
 calculations E-22
- electromagnetic interference (EMI) 13-41
- electrostatic interference (ESI) 13-41
- emul** 6-48
- endian 14-5  
 converting big- and little-endian data 14-14
- eshro** 6-49
- Event initiated accesses 15-28
- executable group E-13, E-24
- execution architecture 1-2
- Execution Unit (EU) E-6, E-17
- explicit calls 7-1
- extended addressing instructions 5-14
- Extended Breakpoint Control Register (XBPBON) 9-7
- extended divide instruction 6-47
- extended multiply instruction 6-48
- extended shift right ordinal instruction 6-49
- external interrupt (XINT#) signals 11-17
- external memory requirements 3-16
- extract** 6-50
- F**
- FAIL# signal 13-8
- fault  
 OPERATION.UNIMPLEMENTED 4-1  
 fault conditional instructions 6-51  
 fault conditions 8-1  
 fault handling  
 data structures 8-1  
 fault record 8-2, 8-6  
 fault table 8-2, 8-5  
 fault type and subtype numbers 8-3  
 fault types 8-4  
 local calls 8-2  
 multiple fault conditions 8-9  
 procedure invocation 8-6  
 return instruction pointer (RIP) 8-15  
 stack usage 8-6  
 supervisor stack 8-2  
 system procedure table 8-2  
 system-local calls 8-2  
 system-supervisor calls 8-2  
 user stack 8-2  
 fault record 8-6  
 address-of-faulting-instruction field 8-6  
 fault subtype field 8-6  
 location 8-6, 8-8  
 optional data fields 8-8  
 structure 8-6  
 fault table 3-1, 3-14, 8-5  
 alignment 3-17  
 local-call entry 8-6  
 location 8-5  
 system-call entry 8-6  
 fault type and subtype numbers 8-3  
 fault types 8-4  
**faulfe** 6-51
- faultg** 6-51
- faultge** 6-51
- faultl** 6-51
- faultle** 6-51
- faultne** 6-51
- faultno** 6-51
- faulto** 6-51
- faults A-7  
 access 3-7  
 AC.nif bit 8-20  
 ARITHMETIC.INTEGER\_OVERFLOW 6-85  
 ARITHMETIC.OVERFLOW 6-8, 6-11, 6-46, 6-78, 6-95, 6-101, 6-106  
 ARITHMETIC.ZERO\_DIVIDE 6-46, 6-47, 6-72, 6-85  
 CONSTRAINT.RANGE 6-51  
 controlling precision of (**syncf**) 8-20



OPERATION.INVALID\_OPERAND 6-43  
PROTECTION.LENGTH 6-26  
TRACE.MARK 6-54, 6-70  
TYPE.MISMATCH 6-43, 6-61, 6-62, 6-64, 6-65, 6-74  
fetch latency E-28  
fetch strategy E-27  
field definition 1-9  
flag definition 1-9  
floating point 3-23  
flush local registers instruction 6-53  
**flushreg** 6-53, 7-11  
**fmark** 6-54  
force mark instruction 6-54  
FP, see Frame Pointer  
frame fills 7-7  
Frame Pointer (FP) 7-4  
    location 3-3  
frame spills 7-7

## G

GCON 12-5  
global registers 3-1, 3-2  
    overview 1-8  
GMU 12-11  
GMU Control Register (GCON) 12-5  
GMU Control (GCON) register 4-6, 12-5  
GMU Memory Protect Address Register (MPARx, MPMRx)  
    12-7  
GMU Memory Violation Detection Upper and Lower-Bounds  
    Registers 12-11  
Guarded Memory Unit (GMU) 3-15, 4-6  
    described 12-1

## H

hardware breakpoint resources 9-5  
    requesting access privilege 9-6  
high priority interrupts 4-3

## I

IBR, see initialization boot record  
**ioctl** 3-27, 4-4, 4-5, 4-6  
ICON 11-20  
IEEE Standard Test Access Port 16-2  
IEEE Std. 1149.1 16-2  
IEEE 1149.1 Device Identification Register 13-23  
IMAP0-IMAP2 11-22  
**IMI** 13-10  
implementation-specific features A-1  
implicit calls 7-1, 8-2  
IMSK 11-24

index with displacement addressing mode 2-8  
indivisible access 3-16  
inequalities (greater than, equal or less than) conditions 3-22  
Initial Memory Image (IMI) 13-1, 13-10  
initialization 13-1, 13-2  
    CLKIN 13-35  
        code example 13-25  
        hardware requirements 13-35  
        MON960 13-25  
        power and ground 13-35  
        software 6-108  
Initialization Boot Record (IBR) 3-1, 3-14, 13-1, 13-13, 13-15  
    alignment 3-17  
initialization data structures 3-14  
initialization mechanism A-5  
initialization requirements  
    control table 13-22, D-22  
    data structures 13-10  
    Process Control Block 13-17  
    reserved memory space 13-10  
instruction breakpoint modes  
    programming 9-10  
Instruction Breakpoint (IBP) registers 9-9  
Instruction Breakpoint (IPB) Register Format 9-9  
instruction cache 3-19  
    bus snooping 3-19  
    coherency 4-6  
    configuration 3-19  
    disabling 3-19  
    effects of disabling 4-5  
    enabling and disabling 4-5, 13-20  
    fetch latency E-28  
    fetch strategy E-27  
    invalidation 3-19  
    load-and-lock mechanism 3-20  
    locking instructions 4-5  
    overview 4-4  
    visibility 4-5  
Instruction Fetch Unit (IFU) E-27  
instruction flow E-4  
    decode stage E-4  
    execute stage E-5  
    issue stage E-4  
instruction formats 5-3  
    assembly language format 5-1  
    branch prediction 5-14  
    instruction encoding format 5-2  
Instruction Pointer (IP) Register 3-21  
Instruction Pointer (IP) register 3-21  
Instruction Register (IR) 16-4  
    timing diagram 16-19  
Instruction Scheduler (IS) E-3  
    instruction cache E-3  
    instruction fetch unit E-3  
    microcode E-3

instruction set  
**ADD** 6-7  
**addc** 6-9  
**addi** 6-11  
**addie** 6-7  
**addig** 6-7  
**addige** 6-7  
**addil** 6-7  
**addile** 6-7  
**addine** 6-7  
**addino** 6-7  
**addo** 6-11  
**addoe** 6-7  
**addog** 6-7  
**addoge** 6-7  
**addol** 6-7  
**addole** 6-7  
**addone** 6-7  
**addono** 6-8  
**addoo** 6-7  
**alterbit** 6-12  
**and** 6-13  
**andnot** 6-13  
**atadd** 3-16, 4-10, 6-14  
**atmod** 3-8, 3-16, 4-10, 6-15  
**b** 6-16  
**bal** 6-17  
**balx** 6-17  
**bbc** 6-19  
**bbs** 6-19  
**be** 6-21  
**bg** 6-21  
**bge** 3-23, 6-21  
**bl** 6-21  
**ble** 6-21  
**bne** 6-21  
**bno** 6-21  
**bo** 6-21  
**bswap** 6-23  
**bx** 6-16  
**call** 6-24, 7-2, 7-6  
**calls** 3-27, 6-25, 7-2, 7-6  
**callx** 6-27, 7-2, 7-6  
**chkbit** 6-28  
**clrbit** 6-29  
**cmpdeci** 6-30  
**cmpdeco** 6-30  
**cmpi** 5-12, 6-32  
**cmpib** 5-12  
**cmpibe** 6-34  
**cmpibg** 6-34  
**cmpibge** 6-34  
**cmpibl** 6-34  
**cmpible** 6-34  
**cmpibne** 6-34  
**cmpibno** 6-34  
**cmpibo** 6-34  
**cmpinci** 6-31  
**cmpinco** 6-31  
**cmpis** 5-12

**cmipo** 5-12, 6-32  
**cmpobe** 6-34  
**cmpobg** 6-34  
**cmpobge** 6-34  
**cmpobl** 6-34  
**cmpoble** 6-34  
**cmpobne** 6-34  
**concmpl** 6-36  
**concmpo** 6-36  
**dcctl** 3-27, 4-6, 4-7, 4-11, 6-38  
**dcinva** 4-11  
**divi** 6-46  
**divo** 6-46  
**ediv** 6-47  
**emul** 6-48  
**eshro** 6-49  
**extract** 6-50  
**faulite** 6-51  
**faultg** 6-51  
**faultge** 6-51  
**faultl** 6-51  
**faultle** 6-51  
**faultne** 6-51  
**faultno** 6-51  
**faulto** 6-51  
**flushreg** 6-53  
**fmark** 6-54  
**icctl** 3-27, 4-4, 4-5, 4-6  
implementation-specific A-4  
**intctl** 3-27, 6-62  
**intdis** 3-27, 6-64  
**inten** 3-27, 6-65  
**ld** 2-2, 3-17, 6-66  
**lda** 6-69  
**ldib** 2-2, 6-66  
**ldis** 2-2, 6-66  
**ldl** 3-5, 4-8, 6-66  
**ldob** 2-2, 6-66  
**ldos** 2-2, 6-66  
**ldq** 3-17, 4-8, 6-66  
**ldt** 4-8, 6-66  
**mark** 6-70  
**modac** 3-22, 6-71  
**modi** 6-72  
**modify** 6-73  
**modpc** 3-25, 3-27, 6-74, 9-3  
**modtc** 6-75, 9-2  
**mov** 6-76  
**movl** 6-76  
**movq** 6-76  
**movt** 6-76  
**muli** 6-78  
**mulo** 6-78  
**nand** 6-79  
**nor** 6-80  
**not** 6-81  
**notand** 6-81  
**notbit** 6-82  
**notor** 6-83  
**or** 6-84



**ornot** 6-84  
**remi** 6-85  
**remo** 6-85  
**ret** 6-86  
**rotate** 6-88  
**scanbit** 6-89  
**scanbyte** 6-90  
**sele** 5-6, 6-91  
**selg** 5-6, 6-91  
**selge** 5-6, 6-91  
**sell** 5-6, 6-91  
**selle** 5-6, 6-91  
**selne** 5-6, 6-91  
**selno** 5-6, 6-91  
**selo** 5-6, 6-91  
**setbit** 6-93  
**shli** 6-94  
**shlo** 6-94  
**shrdi** 6-94  
**shri** 6-94  
**shro** 6-94  
**spanbit** 6-97  
**st** 2-2, 3-17, 6-98  
**stib** 2-2, 6-98  
**stis** 2-2, 6-98  
**stl** 3-17, 4-8, 6-98  
**stob** 2-2, 6-98  
**stos** 2-2  
**stq** 3-17, 4-8, 6-98  
**stt** 4-8, 6-98  
**subc** 6-103  
**subi** 6-106  
**subie** 6-104  
**subig** 6-104  
**subige** 6-104  
**subil** 6-104  
**subile** 6-104  
**subine** 6-104  
**subino** 6-104  
**subio** 6-104  
**subo** 6-106  
**suboe** 6-104  
**subog** 6-104  
**suboge** 6-104  
**subol** 6-104  
**subole** 6-104  
**subone** 6-104  
**subono** 6-104  
**suboo** 6-104  
**syncf** 6-107, 8-20  
**sysctl** 3-27, 4-4, 4-5, 4-6, 6-108, 9-6  
**teste** 6-112  
**testg** 6-112  
**testge** 6-112  
**testl** 6-112  
**testle** 6-112  
**testne** 6-112  
**testno** 6-112  
**testo** 6-112  
timing A-4

**xnor** 6-114  
**xor** 6-114  
instruction set functional groups 5-4  
Instruction Trace Event 6-4  
Instructions  
    TRISTATE 16-7  
instructions  
    conditional branch 3-22  
    parallel execution 1-2  
    parallel issue E-12  
    parallel processing E-11  
    scoreboarding E-14  
instruction-trace mode 9-3  
**intctl** 3-27, 6-62  
**intdis** 3-27, 6-64  
integers 2-2  
    data truncation 2-2  
    sign extension 2-2  
**inten** 3-27, 6-65  
internal data RAM 4-1  
    local register cache 3-18  
    location 3-18  
    modification 3-19, 4-1  
    size 4-1  
    write protection 3-19  
interrupt  
    timer 11-9  
Interrupt Control (ICON) Register 11-20  
Interrupt Control (ICON) register  
    memory-mapped addresses 11-19  
interrupt controller 11-1  
    configuration 11-28  
    interrupt pins 11-17  
    overview 11-2  
    program interface 11-3  
    programmer interface 11-19  
    setup 11-28  
interrupt handling procedures 11-28  
    AC and PC registers 11-28  
    address space 11-28  
    global registers 11-28  
    instruction cache 11-28  
    interrupt stack 11-28  
    local registers 11-28  
    location 11-28  
    special function registers 11-28  
    supervisor mode 11-28  
Interrupt Mapping (IMAP0-IMAP2) Registers 11-22  
Interrupt Mapping (IMAP0-IMAP2) registers 11-21  
interrupt mask  
    saving 11-16  
Interrupt Mask (IMSK) register 11-23, D-9  
Interrupt Mask (IMSK) Registers 11-24  
Interrupt Pending (IPND) Register 11-25  
Interrupt Pending (IPND) register 11-23  
interrupt performance  
    caching of interrupt-handling 11-33  
    interrupt stack 11-34  
    local register cache 11-33  
interrupt pins

- dedicated mode 11-8
- expanded mode 11-8
- mixed mode 11-8
- interrupt posting 11-2
- interrupt procedure pointer 11-5
- interrupt record 11-7
  - location 11-7
- interrupt request management 11-8
- interrupt requests
  - sysctl** 11-9
- interrupt sequencing of operations 11-26
- interrupt servicing mechanism A-5
- interrupt stack 3-1, 3-14, 11-6, 11-34
  - alignment 3-17
  - structure 11-6
- interrupt table 3-1, 3-14, 11-4
  - alignment 3-17, 11-4
  - caching mechanism 11-6
  - location 11-4
  - pending interrupts 11-5
  - vector entries 11-5
- interrupt vectors
  - caching 4-1
- interrupts
  - dedicated mode 11-13
  - dedicated mode posting 11-13
  - expanded mode 11-14
  - function 11-1
  - global disable instruction 6-64
  - global enable and disable instruction 6-62
  - global enable instruction 6-65
  - high priority 4-3
  - internal RAM 11-32
  - interrupt context switch 11-29
  - interrupt handling procedures 11-28
  - interrupt record 11-7
  - interrupt stack 11-6
  - interrupt table 11-4
  - masking hardware interrupts 11-17
  - mixed mode 11-16
  - Non-Maskable Interrupt (NMI#) 11-3, 11-8
    - overview 11-1
  - physical characteristics 11-17
  - posting 11-2
  - priority handling 11-11
  - priority-31 interrupts 11-3, 11-17
  - programmable options 11-18
  - restoring r3 11-17
  - servicing 11-3
  - sysctl** 11-9
  - vector caching 11-32
- IP 3-21
  - IP register, see Instruction Pointer (IP) register
  - IP with displacement addressing mode 2-8
  - IPB 9-9
  - IPND 11-25
  - IS, see Instruction Scheduler (IS)
- i960 processor
  - block diagram 1-1
  - Family description 1-2
  - 80960Hx family members 1-2
- J**
- JTAG (boundary-scan) 16-1
- L**
- ld** 2-2, 3-17, 6-66
- lda** 6-69
- ldib** 2-2, 6-66
- ldis** 2-2, 6-66
- ldl** 3-5, 4-8, 6-66
- ldob** 2-2, 6-66
- ldos** 2-2, 6-66
- ldq** 3-17, 4-8, 6-66
- ldt** 4-8, 6-66
- leaf procedures 7-1
- literal addressing and alignment 3-5
- literals 2-4, 3-1, 3-5
  - addressing 3-5
- little endian byte order 2-4, 3-18, 15-29
- LMADR register
- LMAR14:0 14-11
- LMCON registers
- LMMR14:0 14-12
- load address instruction 6-69
- load instructions 5-5, 6-66
- load-and-lock mechanism 3-20, 4-5
- local calls 7-2, 7-14, 8-2
  - call** 7-2
  - callx** 7-2
- local register cache 3-18, 7-3, E-7
  - overview 4-3
- local registers 3-1, 7-2
  - allocation 3-3, 7-2
  - management 3-3
  - overview 1-8
  - usage 7-2
- local stack 3-1
- Logical Configuration (LMCON) registers 4-11
- logical data templates
  - effective range 14-13
- logical instructions 5-10
- Logical Memory Address Registers (LMAR14:0) 14-11
- Logical Memory Address (LMADR) register 14-4
  - programming 14-11
- logical memory attributes 14-4
- Logical Memory Configuration (LMCON) registers 14-4
- Logical Memory Mask Registers (LMMR14:0) 14-12
- Logical Memory Mask (LMMR) registers
  - programming 14-11
- Logical Memory Templates (LMTs)
  - accesses across boundaries 14-16
  - boundary conditions 14-16
  - enabling 14-15



enabling and disabling data caching 14-15  
modifying 14-16  
overlapping ranges 14-16  
values after reset 14-15

## M

**mark** 6-70  
Mark Trace Event 6-4  
MDLB, see Memory Detect Lower Bounds (MDLB) register  
MDUB, see Memory Detect Upper Bounds (MDUB) register  
memory  
    internal data RAM 3-18  
    memory access 15-2  
    memory address space 3-1  
        external memory requirements 3-16  
            atomic access 3-16  
            big endian byte order 3-18  
            data alignment 3-16  
            data block sizes 3-17  
            data block storage 3-18  
            indivisible access 3-16  
            instruction alignment in external memory 3-16  
            little endian byte order 3-18  
            reserved memory 3-16  
    memory addressing modes  
        absolute 2-7  
        examples 2-8  
        index with displacement 2-8  
        IP with displacement 2-8  
        overview 2-6  
        register indirect 2-7  
    Memory Detect Lower Bounds (MDLB) register 12-11  
    Memory Detect Upper Bounds (MDUB) register 12-12  
    memory detection scheme, GMU  
        described 12-3  
    Memory Management Unit (MMU) 3-15  
    Memory Protect Mask Register (MPMR) 12-6  
    Memory Protection Address Register (MPAR) 12-6  
    memory protection scheme, GMU  
        described 12-3  
    memory request 15-2  
    memory-mapped control registers 3-6  
    Memory-Mapped Registers (MMR) 3-6, 3-16  
    micro-flows  
        atomic instructions E-33  
        bit and bit field instructions E-31  
        branch instructions E-32  
        call and return instructions E-32  
        comparison instructions E-32  
        data movement instructions E-31  
        debug instructions E-33  
        definition E-13  
        execution E-30  
        fault instructions E-33  
        invocation E-29  
        processor management instructions E-34  
    MMR, see Memory-Mapped Registers (MMR)  
**modac** 3-22, 6-71  
**modi** 6-72

## **modify** 6-73

modify arithmetic controls instruction 6-71  
modify process controls instruction 6-74  
modify trace controls instruction 6-75, 9-2  
**modpc** 3-25, 3-27, 6-74, 9-3  
**modtc** 6-75, 9-2  
modulo integer instruction 6-72  
**mov** 6-76  
move instructions 6-76  
**movl** 6-76  
**movq** 6-76  
**movt** 6-76  
MPARx, MPMRx 12-7  
MPAR, see Memory Protect Address Register (MPAR)  
MPMR, see Memory Protect Mask Register (MPMR)  
**muli** 6-78  
**mulo** 6-78  
multiple fault conditions 8-9  
multiply integer instruction 6-78  
multiply ordinal instruction 6-78  
Multiply/Divide Unit (MDU) E-6, E-18

## N

**nand** 6-79  
NMI, see Non-Maskable Interrupt (NMI#)  
No Imprecise Faults (AC.nif) bit 8-16, 8-20  
Non-Maskable Interrupt (NMI#) 11-3, 11-8  
Non-Maskable Interrupt (NMI)  
    signal 11-17  
**nor** 6-80  
**not** 6-81  
**notand** 6-81  
notation and terminology 1-7  
**notbit** 6-82  
**notor** 6-83  
NRAD 14-2, 15-8  
NRDD 14-2, 15-8  
number representations 1-8  
NWAD 14-2, 15-8  
NWDD 14-2, 15-8  
NXDA 14-3, 15-8

## O

On-Circuit Emulation (ONCE) mode 13-1, 16-1  
OPERATION.UNIMPLEMENTED 4-1  
**or** 6-84  
ordinals 2-2  
    sign and sign extension 2-3  
**ornot** 6-84  
output pins 13-38  
overflow conditions 3-22

## P

parallel instruction execution  
    overview 1-2  
parallel issue E-12

- parallel processing E-11, E-12
  - parallel execution E-12
- parameter passing 7-13
  - argument list 7-13
  - by reference 7-13
  - by value 7-13
- parity 15-24
  - parity checking error 15-24
- PC 3-24
  - PC register, see Process Controls (PC) register
- pending interrupts 11-5
  - encoding 11-5
  - interrupt procedure pointer 11-5
  - pending priorities field 11-5
- PFP r0 7-20
  - Physical Memory Configuration (PMCON) registers
    - application modification 14-11
    - initial values 14-9
  - pipeline stalls
    - register bypassing E-16
    - register scoreboarding E-16
  - pipelined read accesses 15-18
- PMCON 14-8
  - PMCON Register Bit Descriptions 14-8
  - PMCON15 in IBR 13-16
  - PMCON15 Register Bit Description in IBR 13-16
  - power and ground planes 13-37
  - powerup/reset initialization
    - timer powerup 10-10
- PRCB, see Processor Control Block (PRCB)
- prereturn-trace mode 9-4
- preserved field 1-7
- Previous Frame Pointer Register (PFP) (r0) 7-20
  - Previous Frame Pointer (PFP) 3-1, 7-4, 7-5
    - location 3-3
    - r0 7-19
  - priority-31 interrupts 11-3, 11-17
- 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 (PRCB) 3-1, 3-14, 4-5, 13-1, 13-17
  - alignment 3-17
  - configuration 13-17
  - register cache configuration word 13-21
- Process Controls (PC) Register 3-24
  - Process Controls (PC) register 3-24
    - execution mode flag 3-24
    - initialization 3-26
    - modification 3-25
    - modpc 3-25
    - priority field 3-25
    - processor state flag 3-24
    - trace enable bit 3-25
    - trace fault pending flag 3-25
- processing units E-16
  - Address Generation Unit (AGU) E-21
  - Bus Control Unit (BCU) E-22
  - data RAM E-20
  - Execution Unit E-17
  - Multiply/Divide Unit E-18
- processor initialization 13-1
- processor management instructions 5-19
- processor state registers 3-1, 3-21
  - Arithmetic Controls (AC) register 3-21
  - Instruction Pointer (IP) register 3-21
  - Process Controls (PC) register 3-24
  - Trace Controls (TC) register 3-26
- program-initiated accesses 15-28
- programming
  - logical memory attributes 14-15, 14-16
- protection modes, GMU 12-1
- protection scheme, GMU
  - described 12-3

## R

- RAM 3-14
  - internal data
    - described 4-1
  - RAM, internal data 3-18
- region boundaries
  - bus transactions across 14-10
- register
  - access 11-25
  - addressing 3-5
    - addressing and alignment 3-5
    - boundary-scan 16-8
  - Breakpoint Control (BPCON) 9-7
  - bypassing E-6
  - cache 4-3
  - control 3-7
    - memory-mapped 3-6
  - DEVICEID
    - memory location 3-3
  - global 3-2
  - indirect addressing mode
    - register-indirect-with-displacement 2-7
    - register-indirect-with-index 2-7
    - register-indirect-with-index-and-displacement 2-8
    - register-indirect-with-offset 2-7
  - Instruction 16-4
  - Interrupt Control (ICON) 11-19
  - Interrupt Mapping (IMAP0-IMAP2) 11-21
  - Interrupt Mask (IMSK) 11-23
  - Interrupt Pending (IPND) 11-23, 11-25
    - local
      - allocation 3-3
      - management 3-3
  - Logical Memory Templates (LMTs) 14-16
  - processor-state 3-21



scoreboarding E-5, E-15  
example 3-4  
implementation 3-4  
pipeline stalls E-16  
**TCRx** 10-5  
Register File (RF) E-5  
  CTRL units E-12  
  MEM E-6  
  MEM units E-12  
  REG E-6  
  REG units E-12  
Registers  
  Arithmetic Controls (AC) Register 3-21  
  Breakpoint Control Register (BPCON) 9-7  
  Cache Control Register (CCON) 4-7  
  Data Address Breakpoint (DAB) Register Format 9-9  
  Default Logical Memory Configuration Register (DLM-CON) 14-13  
  Extended Breakpoint Control Register (XBPCON) 9-7  
  GMU Control Register (GCON) 12-5  
  GMU Memory Protect Address Register (MPARx, MP-MRx) 12-7  
  GMU Memory Violation Detection Upper and Lower-Bounds Registers 12-11  
  IEEE 1149.1 Device Identification Register 13-23  
  Instruction Breakpoint (IPB) Register Format 9-9  
  Instruction Pointer (IP) Register 3-21  
  Interrupt Control (ICON) Register 11-20  
  Interrupt Mapping (IMAP0-IMAP2) Registers 11-22  
  Interrupt Mask (IMSK) register 11-24  
  Interrupt Pending (IPND) Register 11-25  
  Logical Memory Address Registers (LMAR14:0) 14-11  
  Logical Memory Mask Registers (LMMR14:0) 14-12  
  PMCON Register Bit Descriptions 14-8  
  PMCON15 Register Bit Description in IBR 13-16  
  Previous Frame Pointer Register (PFP) (r0) 7-20  
  Process Controls (PC) Register 3-24  
  Timer Count Register (TCR0, TCR1) 10-6  
  Timer Mode Register (TMR0, TMR1) 10-3  
  Timer Reload Register (TRR0, TRR1) 10-7  
  Trace Controls (TC) Register 3-26, 9-2  
registers  
  device ID D-6  
  Interrupt Pending (IPND) D-11  
re-initialization  
  software 6-108  
related documents 1-9  
remainder integer instruction 6-85  
remainder ordinal instruction 6-85  
**remi** 6-85  
**remo** 6-85  
reserved field 1-7  
reserved locations A-4  
reserved memory 1-7  
reserving frames in the local register cache 11-33  
reset state 13-4  
resource scoreboard E-5, E-16  
**ret** 6-86  
Return Instruction Pointer (RIP) 7-4  
location 3-3  
return operation 7-7  
return type field 7-5  
RF, see Register File (RF)  
RIP, see Return Instruction Pointer (RIP)  
ROM 3-14  
**rotate** 6-88  
Registers  
  Bus Control Register (BCON) 14-10  
Run Built-In Self-Test (RUNBIST) register 16-8  
r0 Previous Frame Pointer (PFP) 7-19

## S

**SALIGN** A-3  
saving the interrupt mask 11-16  
**scanbit** 6-89  
**scanbyte** 6-90  
scoreboarding  
  instruction E-14  
  register E-15  
  pipeline stalls E-16  
  resource E-16  
**sele** 5-6, 6-91  
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 13-8  
**selg** 5-6, 6-91  
**selge** 5-6, 6-91  
**sell** 5-6, 6-91  
**selle** 5-6, 6-91  
**selne** 5-6, 6-91  
**selno** 5-6, 6-91  
**selo** 5-6, 6-91  
**setbit** 6-93  
SFRs, see special function registers (SFRs) 3-1  
shift instructions 6-94  
**shli** 6-94  
**shlo** 6-94  
**shrdi** 6-94  
**shri** 6-94  
**shro** 6-94  
sign extension  
  integers 2-2  
  ordinals 2-3  
software re-initialization 6-108  
**spanbit** 6-97  
special function registers (SFRs) 3-1, 3-4  
  reading or modifying 3-4  
  usage 3-4  
SP, see Stack Pointer  
SRAM, see Static RAM (SRAM)  
*src/dst* parameter encodings 9-6  
**st** 2-2, 3-17, 6-98  
stack frame  
  allocation 7-2  
stack frame cache 4-3

Stack Pointer (SP) 7-4  
     location 3-3  
 stacks 3-14  
 Static RAM (SRAM) E-7  
     interface F-1  
**stib** 2-2, 6-98  
**stis** 2-2, 6-98  
**stl** 3-17, 4-8, 6-98  
**stob** 2-2, 6-98  
 store instructions 5-5, 6-98  
**stos** 2-2  
**stq** 3-17, 4-8, 6-98  
**stt** 4-8, 6-98  
**subc** 6-103  
**subi** 6-106  
**subie** 6-104  
**subig** 6-104  
**subige** 6-104  
**subil** 6-104  
**subile** 6-104  
**subine** 6-104  
**subino** 6-104  
**subio** 6-104  
**subo** 6-106  
**suboe** 6-104  
**subog** 6-104  
**suboge** 6-104  
**subol** 6-104  
**subole** 6-104  
**subone** 6-104  
**subono** 6-104  
**suboo** 6-104  
 subtract  
     conditional instructions 6-104  
     integer instruction 6-106  
     ordinal instruction 6-106  
     ordinal with carry instruction 6-103  
 suggested reading 1-9  
 supervisor calls 7-2  
 supervisor mode resources 3-26  
 supervisor space family registers and tables 3-9  
 supervisor stack 3-1, 3-14  
     alignment 3-17  
 Supervisor (SUP#) signal 3-26  
 supervisor-trace mode 9-3  
**syncf** 6-107, 8-20  
 synchronize faults instruction 6-107  
**sysctl** 3-8, 3-27, 4-4, 4-5, 4-6, 6-108, 9-6  
 system calls 7-2, 7-15  
     **calls** 7-2  
         system-local 7-2, 8-2  
         system-supervisor 7-2, 8-2  
 system control instruction 6-108  
 system procedure table 3-1, 3-14  
     alignment 3-17

**T**  
 TC 3-26, 9-2  
 TCR0, TCR1 10-6  
 Test Access Port (TAP) controller 16-11  
     block diagram 16-3  
     state diagram 16-12  
 test features 16-2  
 test instructions 6-112  
 Test Mode Select (TMS) line 16-11  
**teste** 6-112  
**testg** 6-112  
**testge** 6-112  
**testl** 6-112  
**testle** 6-112  
**testne** 6-112  
**testno** 6-112  
**testo** 6-112  
 three-state output pins 13-38  
 timer  
     interrupts 11-9  
     memory-mapped addresses 10-2  
 Timer Count Register (TCRx) 10-5  
     address and access type 3-13  
 Timer Count Register (TCR0, TCR1) 10-6  
 Timer Mode Register  
     timer mode control bit summary 10-8  
 Timer Mode Register (TMRx)  
     address and access type 3-13  
     terminal count 10-3  
     timer clock encodings 10-5  
 Timer Mode Register (TMR0, TMR1) 10-3  
 Timer Reload Register (TRR)  
     address and access type 3-13  
 Timer Reload Register (TRR0, TRR1) 10-7  
 TRR0, TRR1 10-3  
 Trace Controls (TC) Register 3-26, 9-2  
 Trace Controls (TC) register 3-26, 9-2  
 trace events 9-1  
     hardware breakpoint registers 9-1  
     **mark** and **fmark** 9-1  
     PC and TC registers 9-1  
     trace-fault-pending flag 9-3  
     TRISTATE 16-7  
     TRR), TRR1 10-7  
     true/false conditions 3-22  
     TTL input pins 13-38  
     typeface conventions 1-7

**U**  
 unordered numbers 3-23  
 user space family registers and tables 3-13  
 user stack 3-14  
     alignment 3-17  
 user supervisor protection model 3-26  
     supervisor mode resources 3-26  
     usage 3-27



## V

vector entries 11-5  
NMI 11-5  
structure 11-5

## W

warm reset 11-25, 13-4  
words  
triple and quad 2-3

write policy  
data cache E-8

## X

XBPCON 9-7  
XINT#, see external interrupt 11-17  
**xnor** 6-114  
**xor** 6-114

