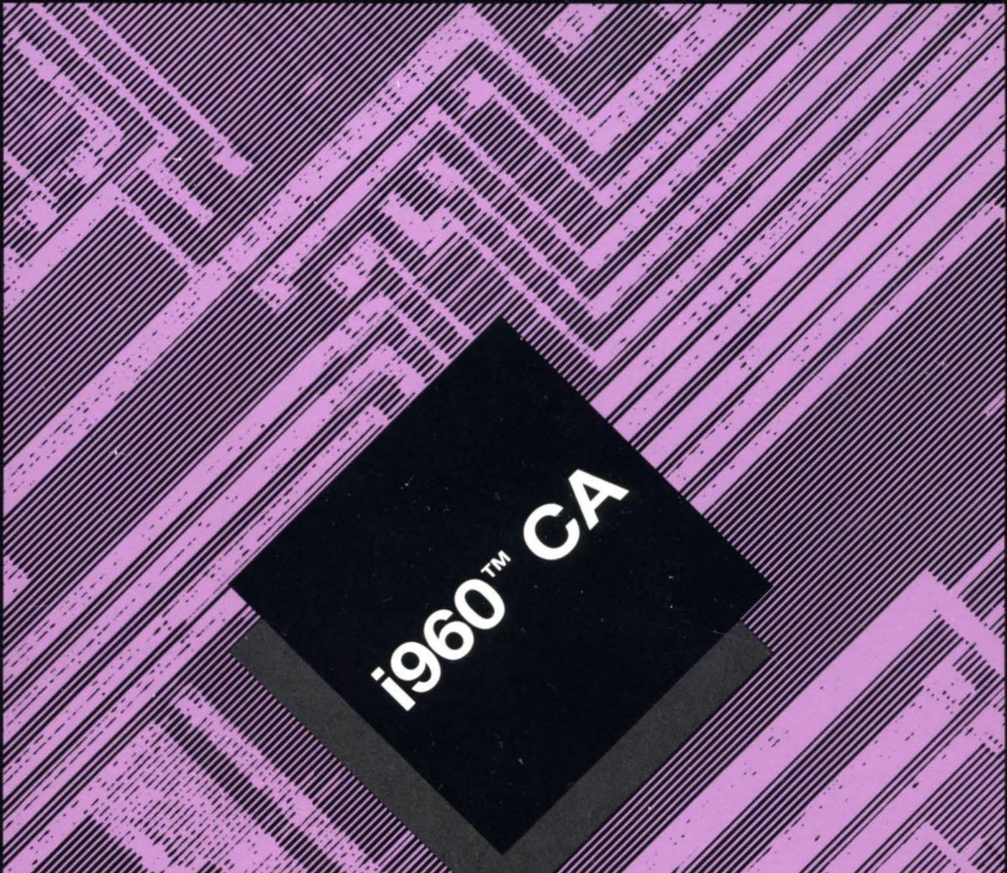
intel®

i960™ CA Microprocessor Users Manual



i960™ CA Microprocessor Users Manual

intel®

LITERATURE

To order Intel literature or obtain literature pricing information in the U.S. and Canada call or write Intel Literature Sales. In Europe and other international locations, please contact your ***local*** sales office or distributor.

INTEL LITERATURE SALES
P.O. Box 7641
Mt. Prospect, IL 60056-7641

In the U.S. and Canada
call toll free
(800) 548-4725
This 800 number is for external customers only.

CURRENT HANDBOOKS

Product line handbooks contain data sheets, application notes, article reprints and other design information. All handbooks can be ordered individually, and most are available in a pre-packaged set in the U.S. and Canada.

TITLE	INTEL ORDER NUMBER	ISBN
SET OF TEN HANDBOOKS (Available in U.S. and Canada)	**231003**	**N/A**
CONTENTS LISTED BELOW FOR INDIVIDUAL ORDERING:		
EMBEDDED CONTROLLERS & PROCESSORS (2 volume set)	270645	1-55512-140-3
MEMORY PRODUCTS	210830	1-55512-144-6
MICROCOMMUNICATIONS	231658	1-55512-148-9
MICROCOMPUTER PRODUCTS	280407	1-55512-143-8
MICROPROCESSORS	230843	1-55512-150-0
MULTIMEDIA & SUPERCOMPUTING PROCESSORS	272084	1-55512-149-7
PACKAGING	240800	1-55512-145-4
PERIPHERAL COMPONENTS	296467	1-55512-146-2
PRODUCT OVERVIEW (A guide to Intel Architectures and Applications)	210846	1-55512-142-x
PROGRAMMABLE LOGIC	296083	1-55512-147-0
ADDITIONAL LITERATURE: (Not included in handbook set)		
AUTOMOTIVE HANDBOOK	231792	1-55512-125-x
COMPONENTS QUALITY/RELIABILITY	210997	1-55512-132-2
CUSTOMER LITERATURE GUIDE	210620	N/A
EMBEDDED APPLICATIONS	270648	1-55512-123-3
INTERNATIONAL LITERATURE GUIDE (Available in Europe only)	E00029	N/A
MILITARY HANDBOOK (2 volume set)	210461	1-55512-126-8
SYSTEMS QUALITY/RELIABILITY	231762	1-55512-046-6
HANDBOOK DIRECTORY (Index of all data sheets contained in the handbooks)	241197	N/A

intel®

U.S. and CANADA LITERATURE ORDER FORM

NAME: ______________________________

COMPANY: ______________________________

ADDRESS: ______________________________

CITY: ____________________ STATE: ________ ZIP: ________

COUNTRY: ______________________________

PHONE NO.: (   ) ______________________________

ORDER NO.	TITLE	QTY.		PRICE		TOTAL
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	

Subtotal ________

Must Add Your Local Sales Tax ________

Include postage:
Must add 15% of Subtotal to cover U.S. and Canada postage. (20% all other.)

→ Postage ________

Total ________

Pay by check, money order, or include company purchase order with this form ($200 minimum). We also accept VISA, MasterCard or American Express. Make payment to Intel Literature Sales. Allow 2-4 weeks for delivery.

☐ VISA ☐ MasterCard ☐ American Express Expiration Date ______________________________

Account No. ______________________________

Signature ______________________________

Mail To: Intel Literature Sales
P.O. Box 7641
Mt. Prospect, IL 60056-7641

International Customers outside the U.S. and Canada should use the International order form on the next page or contact their local Sales Office or Distributor.

For phone orders in the U.S. and Canada
Call Toll Free: (800) 548-4725

Prices good until 12/31/92.
Source HB

CG/LOF1/091091

intel®

INTERNATIONAL LITERATURE ORDER FORM

NAME: ____________________

COMPANY: ____________________

ADDRESS: ____________________

CITY: ____________________ STATE: __________ ZIP: __________

COUNTRY: ____________________

PHONE NO.: (   ) ____________________

ORDER NO.	TITLE	QTY.		PRICE		TOTAL
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	

Subtotal __________

Must Add Your Local Sales Tax __________

Total __________

PAYMENT

Cheques should be made payable to your ***local*** Intel Sales Office (see inside back cover).

Other forms of payment may be available in your country. Please contact the Literature Coordinator at your ***local*** Intel Sales Office for details.

The completed form should be marked to the attention of the LITERATURE COORDINATOR and returned to your ***local*** Intel Sales Office.

CG/091091

intel®

i960™ CA MICROPROCESSOR USER'S MANUAL

1992

Additional copies of this manual or other Intel literature may be obtained from:

Intel Corporation
Literature Sales
P.O. Box 7641
Mt. Prospect, IL 60056-7641

CG-031292

CONTENTS

FIGURES

EXAMPLES

Introduction to the i960™ CA Microprocessor 1

CHAPTER 1
INTRODUCTION TO THE i960™ CA MICROPROCESSOR

Intel's i960 CA microprocessor, a member of the i960 family of 32-bit embedded processors, is the first commercially available superscalar processor. Superscalar technology enables this processor to execute up to three instructions in a single clock cycle. It is an ideal communications controller; as such, it is the natural choice to use as a connection processor in the emerging field of Computer Supported Collaboration (CSC), where high speed networks are used to link multimedia PCs.

The i960 CA product represents Intel's commitment to provide a spectrum of reliable, cost-effective, high-performance processors to satisfy the requirements of today's innovative products. It is designed for applications which require greater performance on a single chip than is usually found in an entire embedded system. The sheer speed of the i960 CA processor enriches traditional embedded applications and makes many new functions possible at a reduced cost. This embedded processor is versatile; it is found in diverse product lines such as laser printers, X-terminals, bridges, routers and PC add-in cards.

As shown in Figure 1.1, the i960 CA component integrates many features onto a single CHMOS device, including the multiple-instruction per clock C-series core, a 1 Kbyte two-way set associative instruction cache, a programmable register cache, a 1 Kbyte on-chip data RAM, a multi-mode programmable bus controller for its demultiplexed bus, a four-channel 59 Mbyte per second DMA controller and a high-speed interrupt controller.

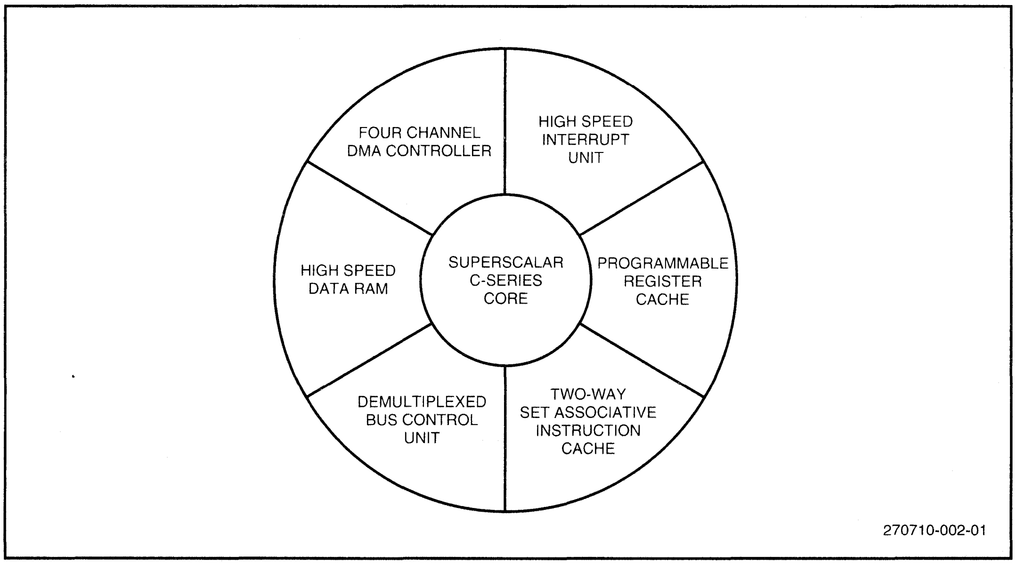

FOUR CHANNEL DMA CONTROLLER
HIGH SPEED INTERRUPT UNIT
HIGH SPEED DATA RAM
SUPERSCALAR C-SERIES CORE
PROGRAMMABLE REGISTER CACHE
DEMULTIPLEXED BUS CONTROL UNIT
TWO-WAY SET ASSOCIATIVE INSTRUCTION CACHE
270710-002-01

Figure 1.1. The Single-Chip i960™ CA Superscalar Processor

THE I960™ CA MICROPROCESSOR ARCHITECTURE

The i960 architecture provides a high-performance computing model. The architecture profits from reduced instruction set computer (RISC) concepts and includes refinements for execution of more than one instruction per clock through superscalar implementations. Furthermore, the architecture provides a high-speed procedure call/return model, a powerful instruction set suited to parallelism and integrated interrupt- and fault-handling models appropriate in a parallel execution environment.

Parallel Instruction Execution

To sustain execution of multiple instructions in each clock cycle, a processor must decode multiple instructions in parallel and simultaneously issue these instructions to parallel processing units. The various processing units must then be able to independently access instruction operands in parallel from a common register set.

On-chip instruction cache enables parallel decode by constantly providing the next four unexecuted instructions to the processor's instruction scheduler. In a single clock cycle, the scheduler inspects all four instructions and issues one, two or three of these instructions in the same clock cycle.

Parallel decode also speeds conditional operations such as branches. These instructions are decoded and executed ahead of the current instruction pointer while maintaining the logical control flow of the sequential program.

Once the scheduler issues an instruction or group of instructions, one of six parallel processing units begins to execute each instruction. Each parallel unit handles a different subset of the instruction set, enabling multiple instructions to be issued and executed every clock cycle. Each unit executes its instructions in parallel with other processor operations.

The i960 CA processor's 32 general purpose 32-bit registers are each six-ported to allow unimpeded parallel access to independent processing units. To maintain the logical integrity of sequential instructions which are being executed in parallel, the processor implements register scoreboarding and resource scoreboarding interlocks.

The 960 CA processor's superscalar can decode multiple instructions at once and issue them to independent processing units where they are executed in parallel. As a result, the processor delivers sustained execution of multiple instructions per clock from a sequential instruction stream.

Full Procedure Call Model

This processor supports two types of procedure calls: an integrated call-and-return mechanism and a RISC-style branch-and-link instruction. The integrated call-and-return mechanism automatically saves local registers when a call instruction is executed and restores them when a return is executed. The RISC-style branch-and-link is a fast call that does not save any of the registers. These mechanisms result in high performance and reduced code size, while maintaining assembly-level compatibility.

To attain the highest performance for procedure calls and returns, the i960 CA microprocessor integrates a programmable depth register cache. The register cache internally saves the local registers for procedure calls, rather than actually writing the data to the external procedure stack. This caching greatly reduces the external bus traffic associated with procedure context saving and restoring.

Versatile Instruction Set and Addressing

The i960 CA microprocessor offers a full set of load, store, move, arithmetic, shift, comparison and branch instructions and supports operations on both integer and ordinal data types. It also provides a complete set of Boolean and bit-field instructions to simplify manipulation of bits and bit strings.

Most of the processor's instructions are typical RISC operations. However, several commonly used complex instructions are also part of the instruction set. Performance is optimized by implementing these commonly used functions with parallel hardware. For instance, the 32x32 multiply operation — a single instruction — takes less than five clocks to execute: 150 ns or less at 33 MHz. Furthermore, the multiplier is a parallel unit; this allows instructions that follow a multiply to execute before the multiplication is complete. In fact, if several unrelated instructions follow a multiply, the multiplication consumes only one clock of execution.

Integrated Priority Interrupt Model

The i960 CA microprocessor provides a priority-based mechanism for servicing interrupts. The mechanism transparently manages up to 248 distinct sources with 31 levels of priority. Interrupt requests may be generated from external hardware, internal hardware or software.

The interrupt mechanism is managed by hardware which operates in parallel with a program's execution. This reduces interrupt latency and overhead and provides flexible interrupt handling control.

Complete Fault Handling and Debug Capabilities

To aid in program development, the i960 CA microprocessor detects faults (exceptions). When a fault is detected, the processor makes an implicit call to a fault handling routine. Information collected for each fault allows program developers to quickly correct faulting code. It also allows automatic recovery from most faults.

To support system debugging, the i960 architecture provides a mechanism for monitoring processor activities through a software tracing facility. The i960 CA device can be configured to detect as many as seven different trace events, including breakpoints, branches, calls, supervisor calls, returns, prereturns and the execution of each instruction (for single-stepping through a program). The i960 CA component also provides four breakpoint registers that allow break decisions to be made based upon instruction or data addresses.

SYSTEM INTEGRATION

The i960 CA microprocessor is based on the C-series core, which is object code compatible with the 32-bit i960 core architecture. Additionally, the i960 CA device integrates three data control peripherals around the core: bus control unit, DMA controller and interrupt controller.

Pipelined Burst Bus Control Unit

The i960 CA microprocessor integrates a 32-bit high-performance bus controller to interface to external memory and peripherals. The bus control unit incorporates full wait state logic and bus width control to provide high system performance with minimal system design complexity. The bus control unit features a maximum transfer rate of 132 Mbytes per second (at 33 MHz). Internally programmable wait states and 16 separately configurable memory regions allow the processor to interface with a variety of memory subsystems with minimum complexity and maximum performance.

Flexible DMA Controller

A four-channel DMA controller provides high-speed DMA data transfers. Source and destination can be any combination of internal RAM, external memory or peripherals. DMA channels perform single-cycle or multi-cycle transfers and can perform data packing and unpacking between peripherals and memory with varying bus widths. Also provided are block transfers, in addition to source- or destination-synchronized transfers.

The DMA supports various transfer types such as high speed fly-by, quad-word transfers and data chaining with the use of linked descriptor lists. The high performance fly-by mode is capable of transfer speeds of up to 59 Mbytes per second at 33MHz.

Priority Interrupt Controller

The interrupt controller provides full programmability of 248 interrupt sources into 31 priority levels. The interrupt controller handles prioritization of software interrupts, hardware interrupts and process priority. In addition, it also manages four internal sources from the DMA controller and a single non-maskable interrupt input.

i960™ MICROPROCESSOR FAMILY

A standard core architecture allows software designers to develop building block software, such as real-time kernels or libraries of functions optimized for the i960 core architecture. These building blocks are portable to any implementation of the i960 architecture.

As indicated in Figure 1.2, all i960 family products are compatible. Each is a specialized applications device, consisting of a core architecture implementation plus a set of specific building blocks or peripherals. The architecture is expandable to include different peripherals on a processor to meet the needs of specific processing and control applications. Future versions of the i960 microprocessor will feature different attributes to meet the price performance demands of all forms of embedded processor applications.

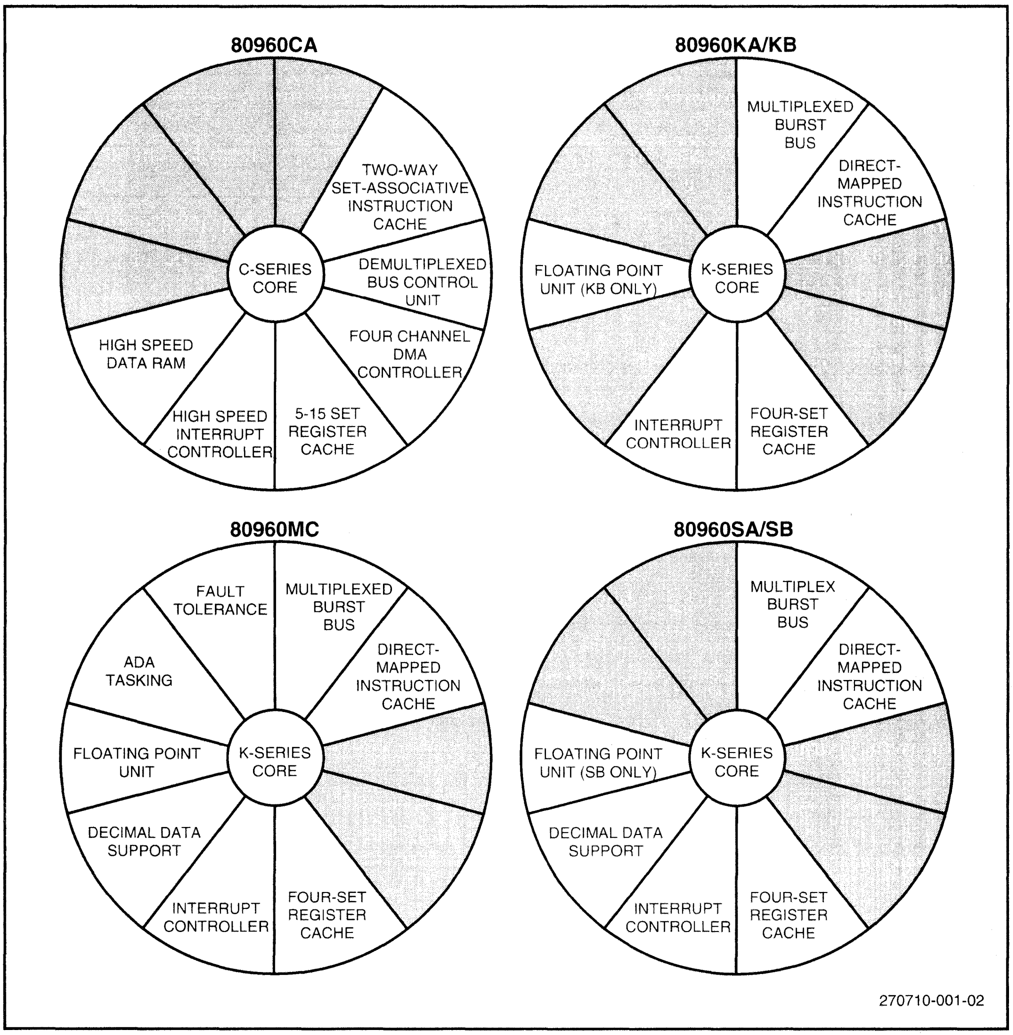


Figure 1.2. i960™ Microprocessor Family

ABOUT THIS MANUAL

This *i960 CA Microprocessor Reference Manual* provides detailed programming and hardware design information for the i960 CA microprocessor. 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 *i960 CA Microprocessor Data Sheet.*

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

This manual is organized in three parts; each part comprises multiple chapters and/or appendices. The following briefly describes each part:

- *Part I-Programming the i960 CA Microprocessor* details the programming environment for the i960 CA component. Described here are the processor's registers, instruction set, data types, addressing modes, interrupt mechanism, external interrupt interface and fault mechanism.
- *Part II-System Implementation* identifies requirements for designing a system around the i960 CA component, such as external bus interface, interrupt controller and integrated DMA controller. Also described are programming requirements for the DMA controller, bus controller and processor initialization.
- *Part III-Appendices* include quick references for hardware design and programming. Appendices are also provided which describe the internal architecture, how to write assembly-level code to exploit the parallelism of the processor and considerations for writing software which is portable between all members of the i960 family.

NOTATION AND TERMINOLOGY

The following paragraphs describe notation and terminology used in this manual that have special meaning.

Reserved and Preserved

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

- A reserved field is one that may be used by other implementations of the i960 architecture. Correct treatment of reserved fields ensures software compatibility with other i960 products. 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 should be set to 0 when the data structure is created. Set reserved fields to 0 when creating the Control Table, Interrupt Table, Fault Table, System Procedure Table, Initialization Boot Record and Processor Control Block. Software should 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 are shown as requiring specific encoding. These fields should be treated as if they were reserved fields. They should be set to the specified value when the data structure is created and software should not modify or rely on the value in the field after that.

Reserved bits in the Special Function Registers must be set to 0 after initialization to ensure compatibility with future implementations. Reserved bits in the Process Controls (PC) register and Trace Controls (TC) register should not be initialized.

When the Arithmetic Controls (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 would likely be located in a reserved memory area on future implementations. Programs may use reserved memory just like any other memory unless it is specifically documented otherwise.

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 $\overline{\text{BTERM}}$ input is active low and is asserted by driving the signal to a logic 0 value.

Representing Numbers

All numbers in this manual can be assumed to be base 10 unless designated otherwise. In text, binary numbers are 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 is sometimes 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, hexadecimal numbers are represented by adding the C-language convention "0x" as a prefix. For example "FF7AH" appears as "0xFF7A" in the pseudo-code.

Register Names

The i960 CA processor's 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, sf1 and sf2. However, when programming the registers in user-generated code, make sure the *instruction operand* is used. The i960 compilers recognize only the instruction operands listed in the following table. Throughout this manual, the register's descriptive names, numbers, operands and acronyms are used interchangeably, as dictated by context.

Register Descriptive Name	**Register Number**	**Instruction Operand**	**Acronym**
Global Registers	g0 - g15	g0 - g14	
Frame Pointer	g15	fp	FP
Local Registers	r0 - r15	r3 - r15	
Previous Frame Pointer	r0	pfp	PFP
Stack Pointer	r1	sp	SP
Return Instruction Pointer	r2	rip	RIP
Interrupt Pending Register	sf0	sf0	IPND
Interrupt Mask Register	sf1	sf1	IMSK
DMA Command Register	sf2	sf2	DMAC

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; however, the user may also program certain flags.
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”.

Programming Environment 2

CHAPTER 2 PROGRAMMING ENVIRONMENT

This chapter describes the i960 CA microprocessor's programming environment which includes global and local registers, special function registers, control registers, literals, processor-state registers and address space.

PROGRAMMING ENVIRONMENT OVERVIEW

The i960 architecture defines a programming environment in which programs are executed and data is stored and manipulated. Figure 2.1 shows the programming environment elements which include a 4 Gbyte (232 byte) flat address space, a 1 Kbyte instruction cache, 16 global and 16 local general purpose registers, a set of literals, special function registers, control registers and a set of processor state registers. A register cache, also shown in Figure 2.1, saves the 16 procedure-specific local registers.

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

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

REGISTERS AND LITERALS AS INSTRUCTION OPERANDS

The i960 CA processor uses only simple load and store instructions to access memory. Therefore, operations take place at the register level. It uses 16 global, 16 local and three special functions registers as instruction operands, as well as 32 literals (constants 0-31).

The global register numbers are g0 through g15; local register numbers are r0 through r15; special function registers are sf0, sf1 and sf2. However, when programming the registers in user-generated code, make sure the *instruction operand* is used. The i960 compilers recognize only the instruction operands listed in Table 2.1. Throughout this manual, the register's descriptive names, numbers, operands and acronyms are used interchangeably, as dictated by context.

Global Registers

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

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. The FP and procedure stack are described in *Chapter 5, Procedure Calls.*

After the processor is reset, register g0 contains die stepping information. Software must read the value of g0 before any action is taken to modify this register. The *i960 CA Microprocessor Data Sheet* Stepping Register Information section describes die stepping information contained in register g0.

Local Registers

Local registers (r0 through r15) provide a separate set of 32-bit data registers — in addition to the global registers — for each active procedure. They 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 for that procedure and saves the calling procedure's local registers. The processor performs local register management; a program need not explicitly save and restore these registers.

Local 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). PFP, SP and RIP are discussed in *Chapter 5, Procedure Calls.*

NOTE

The processor does not always clear or initialize a set of local registers assigned to a new procedure. 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.

Table 2.1. 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	DMA command	DMAC
0-31		literals	

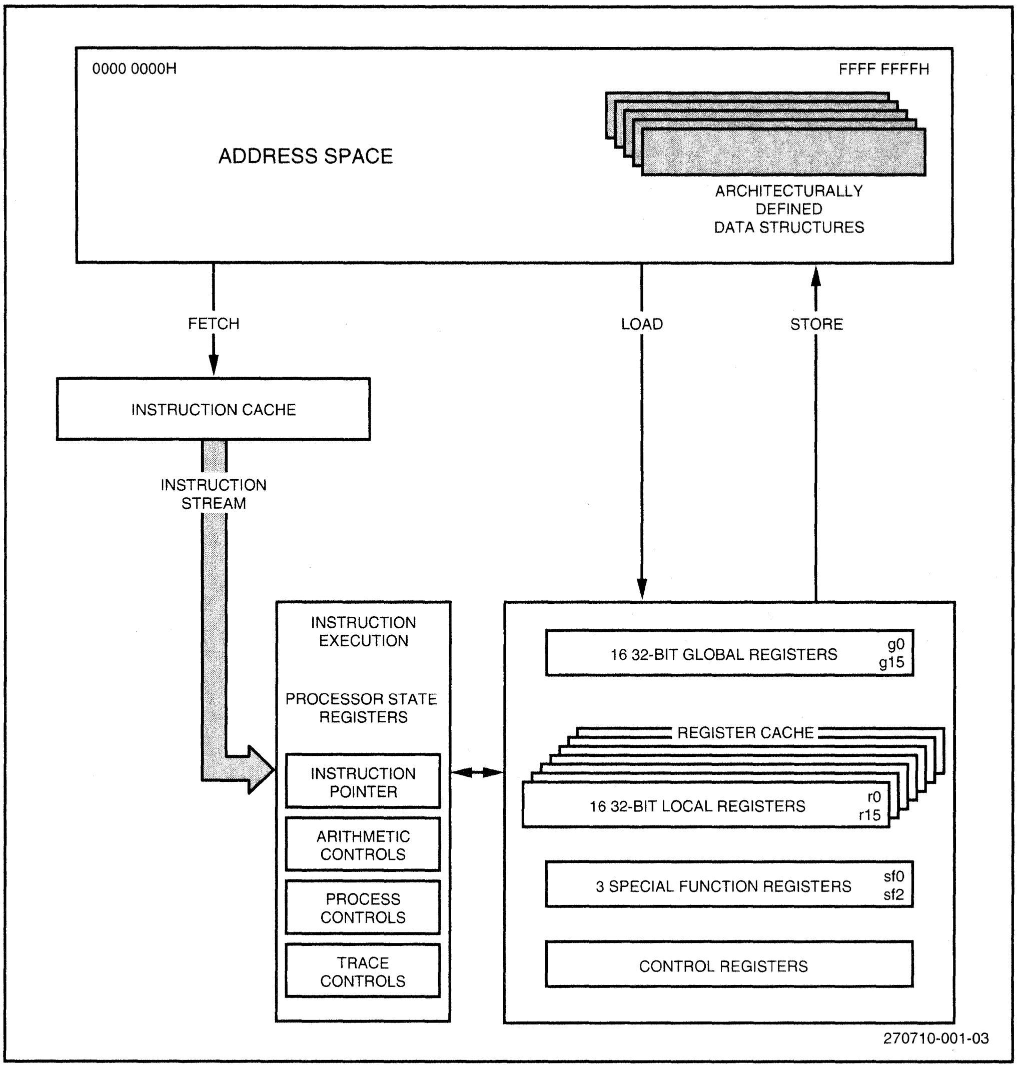


Figure 2.1. i960™ Microprocessor Programming Environment

Special Function Registers (SFRs)

The i960 architecture provides a mechanism to expand its architecture-defined register set with up to 32 additional 32-bit registers. On the i960 CA microprocessor, three special function registers (SFRs) are provided as an extension to the architectural register model. These registers are designated sf0, sf1, sf2 (see Table 2.1). Registers sf3 - sf31 are not implemented on the i960 CA component. Reading or modifying unimplemented registers causes the operation-invalid-opcode fault to occur.

SFRs provide a means to configure and monitor interrupt controller and DMA controller status. SFR function in the i960 CA device is described in *Chapter 12, Interrupt Controller* and *Chapter 13, DMA Controller.*

The processor provides a mechanism which allows only privileged access to SFRs. These registers can only be accessed while the processor is in supervisor execution mode (see *User-Supervisor Protection Model* later in this chapter). A type-mismatch fault occurs if an instruction with a 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. Instructions with these formats include loads, stores and instructions which cause program redirection (call, return and branches; see *Appendix D, Instruction Set Reference* for a description of the machine-level encoding for operands). Table 2.2 summarizes the use of SFRs as instruction operands.

Register Scoreboarding

Register scoreboarding allows concurrent execution of sequential instructions. When an instruction executes, the processor sets a register-scoreboard bit to indicate that a particular register or group of registers is being used in an operation. If the instructions that follow do not use registers in that group, the processor can execute those instructions before the prior instruction execution completes.

A common application of this feature is to execute one or more single-cycle instructions concurrently with a multi-cycle instruction (e.g., multiply or divide). The following example shows a case where register scoreboarding prevents a subsequent instruction from executing. It also illustrates overlapping instructions which do not have register dependencies.

Register scoreboarding is implemented for global and local registers but not for SFRs. When a 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 2.1. Register Scoreboarding

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

Literals

The architecture defines a set of 32 literals which 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.

Register and Literal Addressing and Alignment

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

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

Example 2.2. 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 2.2 lists instruction operands for each machine level instruction format and positions which can be filled by each register or literal.

Table 2.2. Allowable Register Operands

		Operand (1)			
Instruction Encoding	**Operand Field**	**Local Register**	**Global Register**	**Extended Register (SFR)**	**Literal**
REG	*src1*	X	X	X	X
	src2	X	X	X	X
	src/DST (as *src*)	X	X		X
	src/DST (as *DST*)	X	X	X	
	src/DST (as both)	X	X	(2)	
MEM	*src/DST*	X	X		
	abase	X	X		
	index	X	X		
COBR	*src1*	X	X		X
	src2	X	X	X	
	DST	X (3)	X(3)	X(3)	

NOTES:

1. X denotes register can be used as an operand in a particular instruction field.
2. Extended registers cannot be addressed in the *src/DST* field of REG format instructions in which this field is used as both source and destination (e.g., **extract** and **modify**).
3. The COBR destination operands apply only to TEST instructions.

CONTROL REGISTERS

Control registers are internal registers which are used to configure the on-chip peripherals: DMA controller, interrupt controller and bus controller. A program cannot access control registers directly as instruction operands; instead, control registers are loaded from a data structure called the control table (see Figure 2.2).

The system control (**sysctl**) instruction is used to move control table values to on-chip control registers. The control table is divided into seven quad-word groups; each group is assigned a group number from zero to six. When **sysctl** executes, the load control register message type and the group number is specified. **sysctl** moves the quad-word group of register values from the control table in memory and writes the values in the on-chip registers. (See *System Control Functions* later in this chapter.)

At initialization, the control table is automatically loaded into the on-chip control registers. This action simplifies the user's startup code by providing a transparent setup of the i960 CA device's peripherals at initialization. (See *Chapter 14, Initialization and System Requirements.*)

31 0	
IP BREAKPOINT 0 (IPB0)	0H
IP BREAKPOINT 1 (IPB1)	4H
DATA ADDRESS BREAKPOINT 0 (DAB0)	8H
DATA ADDRESS BREAKPOINT 1 (DAB1)	CH
INTERRUPT MAP 0 (IMAP0)	10H
INTERRUPT MAP 1 (IMAP1)	14H
INTERRUPT MAP 2 (IMAP2)	18H
INTERRUPT CONTROL (ICON)	1CH
MEMORY REGION 0 CONFIGURATION (MCON0)	20H
MEMORY REGION 1 CONFIGURATION (MCON1)	24H
MEMORY REGION 2 CONFIGURATION (MCON2)	28H
MEMORY REGION 3 CONFIGURATION (MCON3)	2CH
MEMORY REGION 4 CONFIGURATION (MCON4)	30H
MEMORY REGION 5 CONFIGURATION (MCON5)	34H
MEMORY REGION 6 CONFIGURATION (MCON6)	38H
MEMORY REGION 7 CONFIGURATION (MCON7)	3CH
MEMORY REGION 8 CONFIGURATION (MCON8)	40H
MEMORY REGION 9 CONFIGURATION (MCON9)	44H
MEMORY REGION 10 CONFIGURATION (MCON10)	48H
MEMORY REGION 11 CONFIGURATION (MCON11)	4CH
MEMORY REGION 12 CONFIGURATION (MCON12)	50H
MEMORY REGION 13 CONFIGURATION (MCON13)	54H
MEMORY REGION 14 CONFIGURATION (MCON14)	58H
MEMORY REGION 15 CONFIGURATION (MCON15)	5CH
RESERVED (INITIALIZE TO 0)	60H
BREAKPOINT CONTROL (BPCON)	64H
TRACE CONTROLS (TC)	68H
BUS CONFIGURATION CONTROL (BCON)	6CH

270710-002-02

Figure 2.2. Control Table

ARCHITECTURE-DEFINED DATA STRUCTURES

The architecture defines a set of data structures which includes stacks, interfaces to system procedures, interrupt handling procedures and fault handling procedures. Data structure function is described in the following paragraphs.

user stack	Stack the processor uses when executing applications code. This stack is described in *Chapter 5, Procedure Calls.*

system procedure table	Contains pointers to system procedures. Application code uses the system call instruction (**calls**) to access system procedures through this table. A specific type of system call, known as a system supervisor call, causes a switch in execution mode from user mode to supervisor mode. When the processor switches to supervisor mode, it also switches to a new stack: the *supervisor stack.* System procedure table structure and system call mechanism are described in *Chapter 5, Procedure Calls.* The user-supervisor protection model is described in the section *User-Supervisor Model* in this chapter.
interrupt table	Contains vectors (pointers) to interrupt handling procedures. When an interrupt is serviced, a particular interrupt table entry is specified. A separate *interrupt stack* is provided to ensure that interrupt handling does not interfere with application programs. The interrupt handling mechanism is described in *Chapter 6, Interrupts.*
fault table	Contains pointers to fault handling procedures. When the processor detects a fault, the processor 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 processor execution mode when the fault occurred and type of call made to the fault handling procedure. Fault handling is described in *Chapter 7, Faults.*
control table	Contains on-chip control register values. Control table values are moved to on-chip registers at initialization or with **sysctl**.

The i960 CA microprocessor defines two initialization data structures: *initialization boot record (IBR)* and *processor 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 data structures, 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 in RAM because the processor sometimes writes to it.

MEMORY ADDRESS SPACE

The i960 microprocessor's address space is byte-addressable with addresses running contiguously from 0 to 2^{32} - 1. Some of this address space is reserved or is assigned special functions as shown in Figure 2.3.

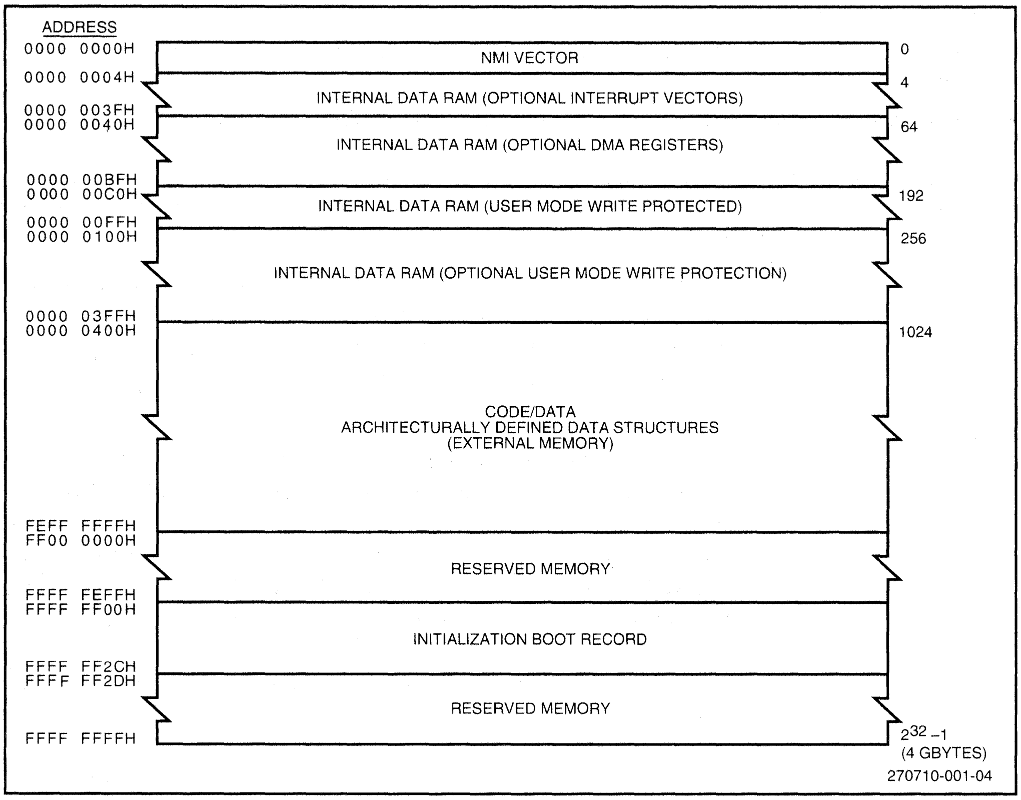


Figure 2.3. Address Space

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

An address in memory is a 32-bit value in the range 0H to FFFFFFFFH. Depending on the instruction, it can be 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). Refer to load and store instruction descriptions in *Chapter 9, Instruction Set Reference* for multiple-byte addressing information.

Memory Requirements

The architecture requires that external memory have the following properties:

- Memory must be byte-addressable.
- No memory is mapped at reserved addresses which are specifically used by an 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. Definitions follow:

indivisible access	Guarantees that a processor, reading or writing a set of memory locations, completes 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 is allowed access to the same location. An atomic memory system can be implemented by using the $\overline{\text{LOCK}}$ signal to qualify hold requests from external bus agents. The $\overline{\text{LOCK}}$ signal is asserted for the duration of an atomic memory operation. (See *Chapter 10, The Bus Controller.*)

The address space upper 16 Mbytes — addresses FF000000H through FFFFFFFFH — are reserved for implementation-specific functions. In general, programs can access this address space section unless an implementation specifically uses the memory or forbids access.

This address range is termed "reserved" so future i960 architecture implementations may use these addresses for special functions such as mapped registers or data structures. Therefore, to ensure complete object level compatibility, portable code must not access or depend on values in this region. The initialization boot record is located in reserved memory of the i960 CA microprocessor. (See Figure 2.3.)

The i960 CA component requires some special consideration when using the lower 1 Kbyte of address space (addresses 0000H-03FFH). Loads and stores directed to these addresses access internal memory; instruction fetches from these addresses are not allowed for the i960 CA microprocessor. (See *Internal Data RAM* in this chapter.)

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 architecture defined data structures on the boundaries specified in Table 2.3.
- Align instruction operands for the atomic instructions (**atadd**, **atmod**) to word boundaries in memory.

The i960 CA microprocessor does not require that load and store data be aligned in memory. It can handle a non-aligned load or store request by either of two methods:

- It can automatically service a non-aligned memory access with microcode assistance (see *Chapter 10, Bus Controller*).
- It can generate an operation unaligned fault when a non-aligned access is detected.

The method for handling non-aligned accesses is selected at initialization based on the value of Fault Configuration Word in the Process Control Block (see *Chapter 14, Initialization and System Requirements*).

Table 2.3. Alignment of Data Structures in the Address Space

Data Structure	Alignment
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 FF00H

Byte, Word and Bit Addressing

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

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

When a data block is stored in memory, normally 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 CA microprocessor also provides the option for ordering bytes in an 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 *Chapter 10, Bus Controller.*

When loading a byte, half 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. Bits can only be addressed in data that resides in a register; bit 0 in a register is the least significant bit, bit 31 is the most significant bit.

Internal Data RAM

Internal data RAM is mapped to the address space lower 1 Kbyte (0000H to 03FFH). 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 1 Kbyte 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 alternate functions other than general data storage (Figure 2.3). When the DMA controller is active, 32 bytes of data RAM are reserved for each channel in use. Additionally, 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, other reserved portions of the data RAM can be used for data storage when the alternate function is not used.

Local register cache size is specified by the value of the Register Cache Configuration Word in the Process Control Block (PRCB; see *Chapter 14, Initialization and System Requirements* for PRCB description.) The first five local register sets are cached internally; if more than five sets are to be cached, then 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 extended, each new register set consumes 16 words of internal data RAM beginning at the highest data RAM address. The user program is responsible for preventing any corruption to the areas of internal RAM set aside for the register cache. (See *Chapter 5, 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 only be modified in supervisor mode. Writes to these locations while in user mode cause a type mismatch fault to be generated. This feature provides supervisor protection for DMA and Interrupt functions which use internal RAM (see *User-Supervisor Protection Model* in this chapter). User mode write protection is optionally selected for the rest of the data RAM (0100H to 03FFH) by setting the Bus Configuration Register (BCON) RAM protection bit.

Instruction Cache

The i960 CA component's instruction cache enhances performance by reducing the number of instruction fetches from external memory. The cache provides fast execution of cached code and loops of code in the cache and also provides more bus bandwidth for data operations in external memory.

The instruction cache is a 1 Kbyte, two-way set associative cache, organized in lines of eight 32 bit words. To optimize cache updates when branches or interrupts are executed, each word in the line has a separate valid bit. Cache misses cause the processor to issue either double- or quad-word fetches to update the cache. Refer to *Appendix A, Optimizing Code for the i960 CA Microprocessor* for a thorough discussion of the instruction cache operation.

Bus snooping is not implemented with the i960 CA cache. 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 uploading 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 CA component. **sysctl** is issued with an invalidate-instruction-cache message type. (See *System Control Functions* later in this chapter.)

The user program is responsible for synchronizing a program with the code modification and cache invalidation. In general, a program must ensure that modified code space is not accessed until modification and cache-invalidate is completed.

Instruction cache can be turned off, 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, **sysctl** is executed with the configure-instruction-cache message (see *System Control Functions* later in this chapter.)

When the cache is disabled, the processor depends on a 16 word instruction buffer to provide decoding instructions. The instruction buffer is organized as two sets of two way set associative cache, with a four word line size. When the main cache is disabled, small loops of code may still execute entirely within the instruction buffer.

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 is provided to optimize 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.

Only interrupts can be directed to fetch instructions from the instruction cache's locked portion. Other causes of program redirection always fetch from the normal memory hierarchy, even if the target address of the redirection is represented in the locked cache. When bit 1 of an interrupt vector is set to 1, the interrupt is fetched 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 targeted instruction is fetched from the normal memory hierarchy. See *Chapter 6, Interrupts* for more details on the cache load-and-lock feature.

The full 1 Kbyte cache or 512 bytes of the cache can be configured to load and lock. When only one half of the cache is loaded and locked, the other half acts as a normal two way set associative cache. Normally, an application locks only 512 bytes. Locking the full 1 Kbyte cache means that all instruction fetches come from external memory except for interrupts directed to the locked cache.

sysctl is issued with a configure-instruction-cache message type to select the load and lock mechanism. When the lock option is selected, an address is specified which points to a memory block which is loaded into the locked cache. See *System Control Function* later in this chapter.

PROCESSOR-STATE REGISTERS

The architecture defines four 32 bit registers that contain status and control information. These registers, defined in this section, are:

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

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 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 allows the IP to be used as an offset into the address space. This addressing mode can also be used with the **lda** (load address) instruction to read the current IP value.

When a break occurs in the instruction stream — due to an interrupt, procedure call or fault — the IP of the next instruction to be executed is stored in local register r2 which is usually referred to as the return IP or RIP register. Refer to *Chapter 5, Procedure Calls* for further discussion of this operation.

Arithmetic Controls (AC) Register

The AC register (Figure 2.4) contains condition code flags, integer overflow flag, mask bit and a bit that controls faulting on imprecise faults. Unused AC register bits are reserved.

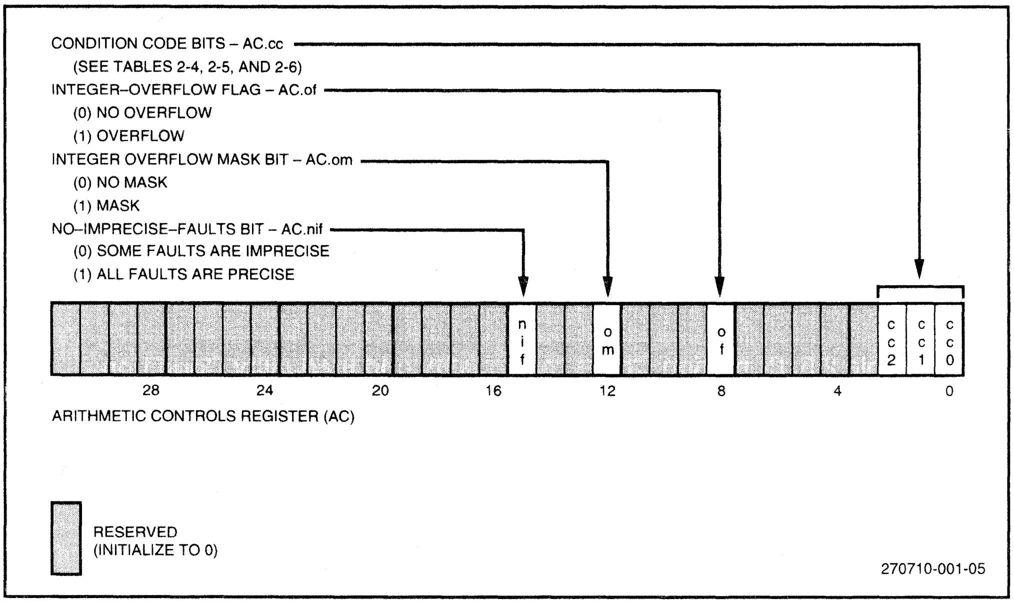

CONDITION CODE BITS – AC.cc
(SEE TABLES 2-4, 2-5, AND 2-6)
INTEGER–OVERFLOW FLAG – AC.of
(0) NO OVERFLOW
(1) OVERFLOW
INTEGER OVERFLOW MASK BIT – AC.om
(0) NO MASK
(1) MASK
NO–IMPRECISE–FAULTS BIT – AC.nif
(0) SOME FAULTS ARE IMPRECISE
(1) ALL FAULTS ARE PRECISE
nif
om
of
cc2
cc1
cc0
28
24
20
16
12
8
4
0
ARITHMETIC CONTROLS REGISTER (AC)
RESERVED
(INITIALIZE TO 0)
270710-001-05

Figure 2.4. Arithmetic Controls (AC) Register

Initializing and Modifying the AC Register

At initialization, the AC register is loaded from the Initial AC image field in the Process Control Block (see *Chapter 14, Initialization and System Requirements*). Reserved bits are set to 0 in the AC Register Initial Image. After initialization, software must not modify or depend on the AC register's reserved location. After initialization, the modify arithmetic controls (**modac**) instruction allows any of the register bits to be examined and modified. This instruction provides a mask operand that can be used to 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.

Condition Code

The processor sets the AC register's *condition code flags* (bits 0-2) to indicate the results of certain instructions — usually compare instructions. Other instructions, such as conditional branch instructions, examine these flags and perform functions according to the state of the condition code. 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 or 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 2.4. To show equality and inequalities, the processor sets the condition code flags as shown in Table 2.5.

Table 2.4. Condition Codes for True or False Conditions

Condition Code	Condition
010_2	true
000_2	false

Table 2.5. Condition Codes for Equality and Inequality Conditions

Condition Code	Condition
000_2	unordered (false)
001_2	greater than (true)
010_2	equal
100_2	less than

NOTE

Some implementations of the i960 architecture provide integrated floating point processing. The terms ordered and unordered are used when comparing floating point numbers. If, when comparing two floating point values, one of the values is a NaN (not a number), the relationship is said to be "unordered." The i960 CA microprocessor does not implement the floating point processor on-chip.

To show carry out and overflow, the processor sets the condition code flags as shown in Table 2.6.

Table 2.6. 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. These masks cover the additional conditions of greater-or-equal (011_2), less-or-equal (110_2) and not-equal (101_2). The mask is part of the instruction opcode and the instruction performs a bitwise AND of the mask and condition code.

Integer Overflow

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, the processor — instead of generating a fault — sets the integer overflow flag when integer overflow is encountered. If the fault is not masked, the fault is allowed to occur and the flag is not set.

Once the processor sets this flag, it never implicitly clears it; the flag remains set until the program clears it. Refer to the discussion of the arithmetic-integer-overflow fault in *Chapter 7, Faults* for more information about the integer overflow mask bit and flag.

No Imprecise Faults

The *no imprecise faults bit* (bit 15) determines whether or not faults are allowed to be imprecise. If set, all faults are required to be precise; if clear, certain faults can be imprecise. (See *Chapter 7, Faults* for more information about precise and imprecise faults.)

Process Controls (PC) Register

The process controls (PC) register (Figure 2.5) contains information to control processor activity and show the processor's current state. This register's various functions are described in this section.

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
- Alter the saved process controls prior to a return from a fault handler.

modpc directly reads and modifies the PC register. The processor must be in supervisor mode to execute this instruction; a type-mismatch fault is generated if **modpc** is executed in user mode. 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 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.

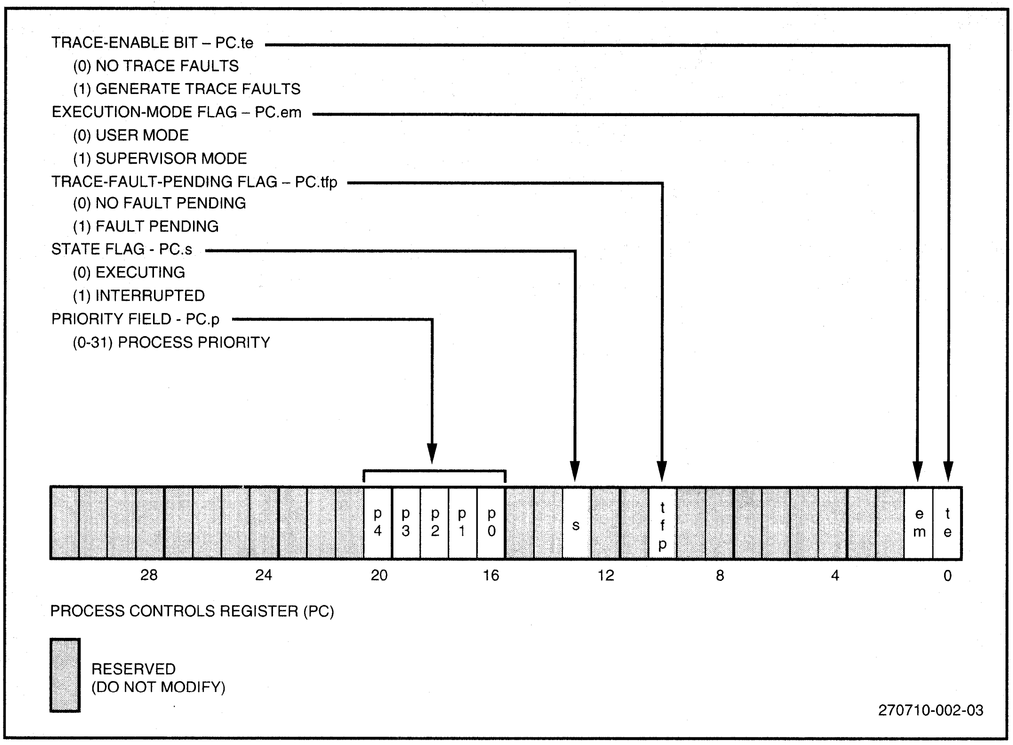


Figure 2.5. Process Controls (PC) Register

NOTE

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 instructions are executed.

After initialization (hardware reset), the process controls reflect the following conditions: priority = 31, execution mode = supervisor, trace enable = off, state = interrupted. When the processor is reinitialized via the system control instruction and reinitialize message, the PC register reflects the same conditions, except that the processor retains the same priority as before reinitialization.

Bits 2-7, 9-12, 14, 15 and 21-31 are reserved. These bits should never be set to zero and user software should not depend on the value of the reserved bits. Do not use **modpc** to directly modify execution mode, trace fault pending and state flags.

Execution Mode

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 *User and Supervisor Protection Model.*)

Program State

PC register *state flag* (bit 13) indicates 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 and then switches back to executing state on the return from the initial interrupt procedure.

Priority

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 using 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 the priority of the interrupt. (See *Chapter 6, Interrupts*)

Trace Status and Control

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 to be generated (1) or not generated (0). The trace fault pending flag indicates that a trace event has been detected (1) or not detected (0). The trace controls are discussed in *Chapter 8, Tracing and Debugging.*

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 which are used to enable specific tracing modes and record trace events, respectively. Trace controls are described in *Chapter 8, Tracing and Debugging.*

USER SUPERVISOR MODEL

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.

Supervisor Mode Resources

The processor can be in either of two execution modes: user or supervisor. Supervisor mode is a privileged mode which 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 system debugging software or a system monitor to be accessed, even if an applications program destroys its own stack.
- When an instruction executed in supervisor mode causes a bus access to occur, an external supervisor pin $\overline{\text{SUP}}$ is asserted 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 (see *Chapter 10, Bus Controller*).
- 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 which can modify DMA or interrupt controller behavior or reconfigure bus controller characteristics can only be performed in supervisor mode. These functions include modification of SFRs, control registers or internal data RAM which is dedicated to the DMA and interrupt controllers. A fault is generated if supervisor-only operations are attempted while the processor is in user mode (see *Chapter 7, Faults*). Table 2.7 lists supervisor-only operations and the fault which is generated if the operation is attempted 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.

Table 2.7. Supervisor-Only Operations and Faults Generated in User Mode

Supervisor-Only Operation	User-Mode Fault
modpc (modify process controls)	type-mismatch
sysctl (system control)	constraint-privileged
sdma (setup DMA)	constraint-privileged
SFR as instruction operand	type-mismatch
Protected internal data RAM write	type-mismatch

Using the User-Supervisor Protection Model

A program switches between user mode and 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 the **calls** instruction, 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. The **calls** instruction and the system procedure table thus provide a tightly controlled interface to procedures which 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 some faults also 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.

Figure 2.6 shows a system which implements the user-supervisor protection model to protect kernel code and data. The code and data structures in the shaded areas can only be accessed in supervisor mode.

In this example, kernel procedures are accessed through the system procedure table with system-supervisor calls. These procedures execute in supervisor mode. Some application procedures are also called through the system procedure table using a system-local call. Fault procedures are executed in supervisor mode by directing the faults through the system procedure table. Interrupt procedures, which are likely to modify SFRs, process controls or use other supervisor operations, are executed in supervisor mode. The interrupt stack and supervisor stack are insulated from the user stack in this system.

If an application does not require 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.

SYSTEM CONTROL FUNCTIONS

System control functions are a group of operations specific to the i960 CA component. All of these operations are performed by issuing the system control (**sysctl**) instruction. The **sysctl** instruction is a general purpose instruction and performs a variety of functions. A *message type field* is an operand of the instruction that determines which function is performed. The system control functions include posting interrupts, configuring the instruction cache, invalidating the instruction cache, software reinitialization and loading control registers.

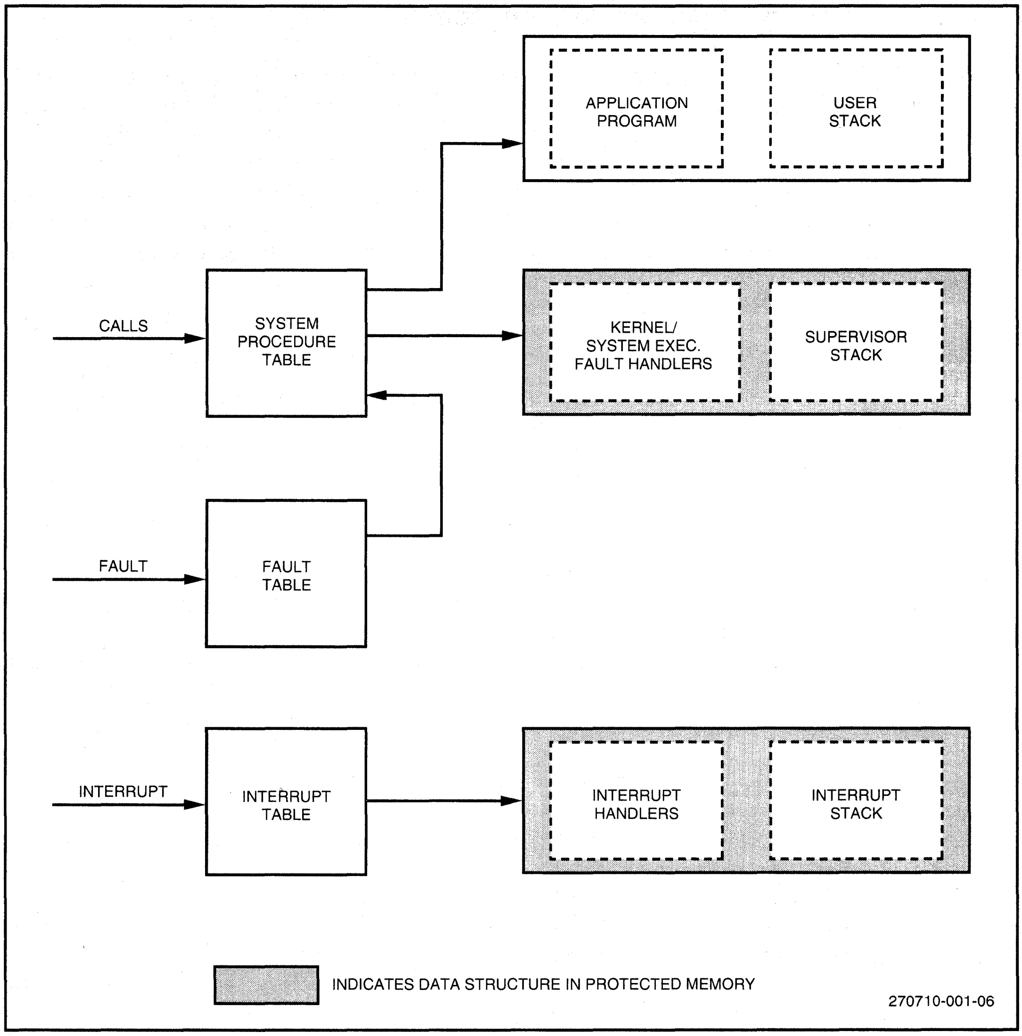


Figure 2.6. Example Application of the User-Supervisor Protection Model

sysctl Instruction Syntax

sysctl instruction syntax is generalized because the function of the operands differ, depending on message type selection. The instruction takes three source operands (Figure 2.7). The message type field is always the second byte of the source 1 operand. The instruction's generalized operand fields, designated as fields 1-4, are interpreted differently or may not be used depending on the function selected in the message type field (Table 2.8).

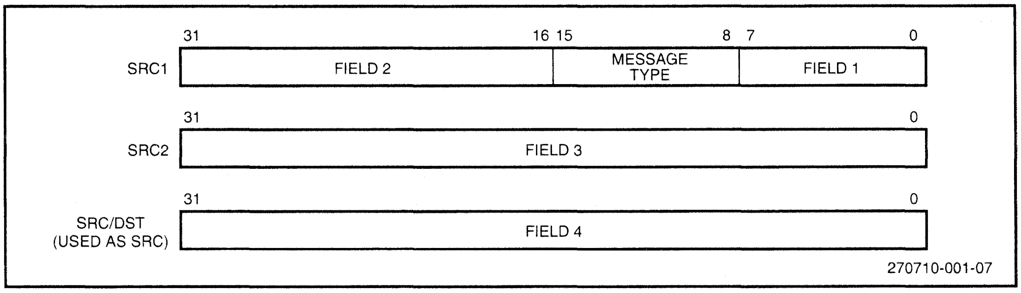


Figure 2.7. Source Operands for sysctl

sysctl is a supervisor only instruction. Executing this instruction while in user mode generates the type-mismatch fault.

Table 2.8. System Control Message Types and Operand Fields

	Source 1			Source 2	Source 3
Message	**Type**	**Field 1**	**Field 2**	**Field 3**	**Field 4**
Request Interrupt	00H	Vector No.	unused	unused	unused
Invalidate Cache	01H	unused	unused	unused	unused
Configure Cache	02H	Mode (Table 2.9)	unused	Cache load address	unused
Reinitialize	03H	unused	unused	1st Inst. address	PRCB address
Load Control Register	04H	Register Group No.	unused	unused	unused

NOTE

The processor ignores unused sources and fields.

System Control Messages

Five system control messages are defined in the sections that follow. The request interrupt message causes an interrupt to be serviced or posted. The configure cache message disables or locks instructions in a portion of the instruction cache. The invalidate cache message causes the contents of the instruction to be purged. The reinitialize message restarts the processor. The load control register message loads the on-chip control registers.

Request Interrupt

Executing **sysctl** with a message type of 00H causes an interrupt to be requested. Field 1 of the instruction specifies the vector number of the interrupt requested. The remaining fields are not defined. Requesting an interrupt with **sysctl** causes the following actions to occur:

- The core 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. This action posts the software requested interrupt.
- The core updates the software priority register with the value of the highest pending priority from the interrupt table. This may be the priority of the interrupt which was just posted. This action causes the interrupt to be serviced if its priority is greater than the current process priority or equal to 31.

Requesting an interrupt with a priority equal to 0 causes a check for posted interrupts in the interrupt table. See *Chapter 6, Interrupts* for more information concerning interrupts requested by software.

Invalidate Cache

Executing **sysctl** with a message type of 01H invalidates all cache entries. This mode clears all valid cache bits. After the operation, the cache is updated normally as misses occur. The mode is provided to allow a program to load or modify program space; it ensures that instructions are fetched from the modified space and not the cache.

Configure Instruction Cache

Executing **sysctl** with a message type of 02H selects cache mode. One of four cache modes are selected with the configure instruction cache message:

1. 1 Kbyte normal cache
2. cache disabled
3. load and lock 1 Kbyte of the cache
4. load and lock 512 bytes of the cache and 512 bytes of normal cache

The particular configure cache operation performed is determined by **sysctl** field 1 value (Table 2.9). Field 3 is a word-aligned 32-bit address when a load and lock mode is selected; otherwise, this field is ignored. Text following the table further defines the modes.

Table 2.9. Cache Configuration Modes

Mode Field	Mode Description
000_2	1 Kbyte normal cache enabled
$XX1_2$	1 Kbyte cache disabled (execute off-chip)
100_2	Load and lock 1 Kbyte cache (execute off-chip)
110_2	Load and lock 512 bytes, 512 bytes normal cache enabled
010_2	Reserved

Mode 000_2 configures the cache as a 1 Kbyte two way set associative cache. Mode $XX1_2$ completely disables the cache. Either of these cache configurations can be specified when the processor initializes by programming the Cache Configuration Word in the PRCB (see *Chapter 14, Initialization and System Requirements*). The modes allow the cache to be turned off temporarily to aid in debugging.

When the cache is disabled, the processor depends on a 16 word instruction buffer to provide decoding instructions. The instruction buffer operates as a small cache, organized as two sets of two way set associative cache, with a four word line size. When the main cache is disabled, small code loops may still execute entirely within the instruction buffer.

Modes 100_2 and 110_2 select cache load-and-lock options. When one of these modes is selected, either 512 bytes or the full 1 Kbyte cache is loaded with instructions and locked against further updates. Field 3 of the **sysctl** instruction must contain an address of a quad-word aligned block of memory, in the external address space, which is represented in the cache. The instructions loaded into the cache can only be accessed by selected interrupts which vector to the addresses of these instructions. The load-and-lock mechanism selectively optimizes latency and throughput for interrupts. (See *Chapter 6, Interrupts.*)

Reinitialize Processor

Executing **sysctl** with message type 03H reinitializes the processor. **sysctl** fields 3 and 4 must contain, respectively, the First Instruction Pointer and the PRCB Pointer. Reinitialization bypasses the i960 CA processor's built-in self-test. The PRCB is processed and the processor branches to the first instruction (see *Chapter 14, Initialization and System Requirements* for a complete description of the processor reinitialization steps).

The reinitialize message is useful for changing the Initial Memory Image. For example, at initialization, the interrupt table is moved to RAM so the interrupts may be posted in the table's pending interrupts and priorities fields. In this case, the reinitialize message specifies a new PRCB which contains a pointer to the new interrupt table in RAM (see *Chapter 14, Initialization and System Requirements* for a description of reinitialization and relocating data structures).

Load Control Registers

Executing **sysctl** with message type 04H causes the on-chip control registers to be loaded with data from external memory. Each **sysctl** invocation causes four words from the Control Register Table in external memory to be read and then placed in their respective internal control registers. Field 1 must contain the number of the register group to be loaded. Table 2.10 shows register group number and the registers represented in the Control Register Table.

At initialization, or when the processor is reinitialized, all groups in the control table are automatically loaded into the on-chip control registers.

Table 2.10. Control Register Table and Register Group Numbers

Group	Byte Offset in Table	Control Register Loaded
00H	00H	IP Breakpoint Register 0 (IPB0)
	04H	IP Breakpoint Register 1 (IPB1)
	08H	Data Address Breakpoint 0 (DAB0)
	0CH	Data Address Breakpoint 1 (DAB1)
01H	10H	Interrupt Map Register 0 (IMAP0)
	14H	Interrupt Map Register 1 (IMAP1)
	18H	Interrupt Map Register 2 (IMAP2)
	1CH	Interrupt Control Register (ICON)
02H	20H	Memory Region 0 Configuration (MCON0)
	24H	Memory Region 1 Configuration (MCON1)
	28H	Memory Region 2 Configuration (MCON2)
	2CH	Memory Region 3 Configuration (MCON3)
03H	30H	Memory Region 4 Configuration (MCON4)
	34H	Memory Region 5 Configuration (MCON5)
	38H	Memory Region 6 Configuration (MCON6)
	3CH	Memory Region 7 Configuration (MCON7)
04H	40H	Memory Region 8 Configuration (MCON8)
	44H	Memory Region 9 Configuration (MCON9)
	48H	Memory Region 10 Configuration (MCON10)
	4CH	Memory Region 11 Configuration (MCON11)
05H	50H	Memory Region 12 Configuration (MCON12)
	54H	Memory Region 13 Configuration (MCON13)
	58H	Memory Region 14 Configuration (MCON14)
	5CH	Memory Region 15 Configuration (MCON15)
06H	60H	Reserved
	64H	Breakpoint Control Register (BPCON)
	68H	Trace Controls Register (TC)
	6CH	Bus Configuration Control (BCON)

Data Types and Memory Addressing Modes

3

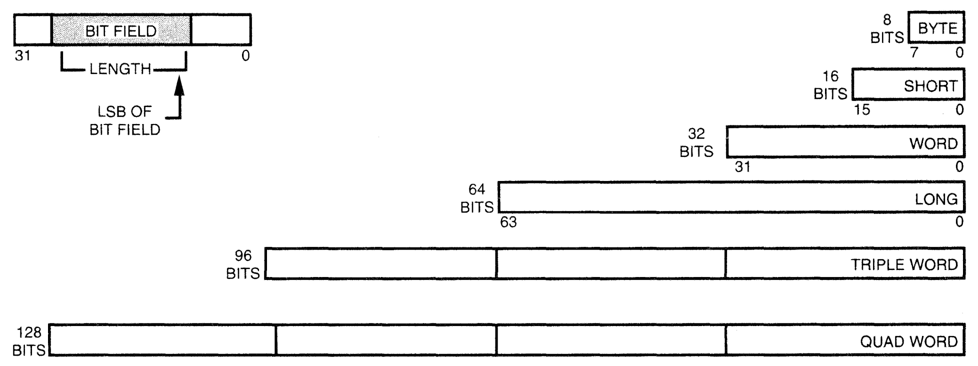
CHAPTER 3
DATA TYPES AND MEMORY ADDRESSING MODES

DATA TYPES

The instruction set references or produces several data lengths and formats. The i960 architecture defines the following data types:

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

Figure 3.1 shows i960 architecture data types and the length and numeric range of each.



CLASS	DATA TYPE	LENGTH	RANGE
NUMERIC (INTEGER)	BYTE INTEGER SHORT INTEGER INTEGER LONG INTEGER	8 BITS 16 BITS 32 BITS 64 BITS	-2^{7} TO $2^{7}-1$ -2^{15} TO $2^{15}-1$ -2^{31} TO $2^{31}-1$ -2^{63} TO $2^{63}-1$
NUMERIC (ORDINAL)	BIT ORDINAL SHORT ORDINAL ORDINAL LONG ORDINAL	8 BITS 16 BITS 32 BITS 64 BITS	0 TO $2^{8}-1$ 0 TO $2^{16}-1$ 0 TO $2^{32}-1$ 0 TO $2^{64}-1$
NON-NUMERIC	BIT BIT FIELD TRIPLE WORD QUAD WORD	1 BIT 1-32 BITS 96 BITS 128 BITS	N/A

270710-001-08

Figure 3.1. Data Types and Ranges

Integers

Integers are signed whole numbers which 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 only referenced by the byte and short classes of the load and store instructions. None of the i960 CA's instructions reference or produce the long-integer data type. The architecture defines four integer sizes:

Integer size	Descriptive name
8 bit	byte integers
16 bit	short integer
32 bit	integers
64 bit	long integers

NOTE

HLL compilers may define long integer types differently than defined by the i960 architecture.

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 which is the destination for the load.

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, an AC register flag is set or the integer overflow fault is generated. *Chapter 7, 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.

Ordinals

Ordinals, an unsigned integer data type, are stored and operated on as positive binary values. The processor recognizes four ordinal sizes:

Ordinal size	Descriptive name
8 bit	byte ordinals
16 bit	short ordinals
32 bit	ordinals
64 bit	long ordinals

The large number of instructions which perform logical, bit manipulation and unsigned arithmetic operations reference 32-bit ordinal operands. When ordinals are used to represent Boolean values, a 1_2 represents a TRUE and a 0_2 represents a FALSE. Several extended arithmetic instructions reference the long ordinal data type. Only load and store instructions reference the byte and short ordinal data types.

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

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. The least significant bit of a 32-bit register is bit 0; the most significant bit is bit 31.

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

NOTE

Loads and stores on bit and bit field data are normally performed with the ordinal load and store instructions. The integer load and store instructions operate on two's complement numbers. Depending on the value, a byte or short integer load can result in sign extension of data in a register; a byte or short store can signal an integer overflow condition.

Triple and Quad Words

Triple and quad words refer to consecutive words in memory or in registers. Triple- and quad-word loads, stores and move instructions use this data type. These instructions facilitate data block movement. 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 — or as encompassing — the other data types described. The data in each word subset of a quad-word is likely the operand or result of an ordinal, integer, bit or bit field instruction.

Data Alignment

Data in registers and memory must adhere to specific alignment requirements:

- Align long-word operands in registers to double-register boundaries.
- Align triple- and quad-word operands in registers to quad-register boundaries.

For the i960 CA component, data alignment in memory is not required. Unaligned memory accesses, by programmable option, can cause a fault or be handled automatically. Refer to

Chapter 2, Programming Environment for a complete description of alignment requirements for data and instructions.

MEMORY ADDRESSING MODES

The processor provides nine modes for addressing operands in memory. Each addressing mode is used to reference a byte in the processor's address space. Table 3.1 shows the memory addressing modes, a brief description of the elements of the address in each mode and the assembly code syntax for each mode. These modes are grouped as follows:

- Absolute
- Index with Displacement
- Register Indirect
- IP with Displacement

Table 3.1. Memory Addressing Modes

Mode	Description	Assembler Syntax
Absolute offset	offset	exp
Absolute displacement	displacement	exp
Register Indirect	abase	(reg)
Register Indirect with offset	abase + offset	exp (reg)
Register Indirect with displacement	abase + displacement	exp (reg)
Register Indirect with index	abase + (index*scale)	(reg) [reg*scale]
Register Indirect with index and displacement	abase + (index*scale) + displacement	exp (reg) [reg*scale]
Index with displacement	(index*scale) + displacement	exp [reg*scale]
IP with displacement	IP + displacement + 8	exp (IP)

NOTE

reg is register and *exp* is an expression or symbolic label.

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 is an integer, called a displacement, ranging from -2^{31} to 2^{31}-1. The absolute displacement addressing mode is encoded in the MEMB format.

Encoding level addressing modes and instruction formats are described in *Appendix D, Instruction Set Reference.*

At the assembly language level the two absolute addressing modes are combined into one; both 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 a displacement and selects the appropriate instruction encoding format and addressing mode.

Register Indirect

Register indirect addressing modes use a 32-bit value in a register as a base for the address calculation. The register value is referred to as the address base (designated *abase* in Table 3.1). 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 gives the first array element address; an offset (or displacement) selects a particular array element.

In register-indirect-with-index addressing mode, the index is specified by means of a value placed in a register. This index value is then multiplied by a scale factor. Allowable scale factors are 1, 2, 4, 8 and 16.

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

At the assembly language level, the assembler allows 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; it is encoded in the MEMB instruction format.

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.

IP with Displacement

This addressing mode is used with load and store instructions to make them 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.

Addressing Mode Examples

The following examples show how i960 addressing modes are encoded in assembly language. Example 3.1 shows addressing mode mnemonics; Example 3.2 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 block is passed to the procedure in g0, the block size in g1 and the fill data in g2.

Example 3.1. 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 loaded to mem
                              # location g4 + xyz + (g5 scaled by 2).
ldis    xyz[r12*1],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 3.2. Use of Index Plus Scaled Index Mode

```
array_op:
     mov      g0,r4                # pointer to array is moved 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 array at index
     st       g2,0x30(r4)[r3*4]    # fill array at index + constant offset
     subi     1,r3,r3              # decrement index
.I33:
     cmpible  0,r3,.I34            # store next array elements if
     ret                           # index is not 0
```

Instruction Set Summary 4

CHAPTER 4
INSTRUCTION SET SUMMARY

This chapter overviews the i960 family's instruction set and i960 CA processor-specific instruction set extensions. This chapter describes assembly-language and instruction-encoding formats, overviews various instruction groups and each group's instructions.

Refer to *Chapter 9, Instruction Set Reference* for descriptions of each instruction, including assembly language syntax, the action taken when the instruction is executed and examples of how the instruction might be used. Instructions in *Chapter 9* are listed in alphabetic order.

INSTRUCTION FORMATS

Instructions described in this reference manual are in two formats: assembly language and instruction encoding. The following sections provide brief descriptions of these formats.

Assembly Language Format

Throughout this manual, instructions are referred to by their assembly language mnemonics. For example, the add ordinal instruction is referred to as **addo**. Examples use Intel 80960 assembler assembly language syntax, consisting of the instruction mnemonic followed by zero to three operands, separated by commas. Following is an assembly language statement example for **addo**. In this example, ordinal operands in global registers g5 and g9 are added together; the result is stored in g7:

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

In the assembly languages listing in this chapter, registers are denoted as:

g	global register	**r**	local register
sf	special function register	**#**	pound sign precedes a comment

All numbers used as literals or in address expressions are assumed to be decimal. Hexadecimal numbers are denoted with a 0x prefix (e.g., 0xffff0012). Several assembly language instruction statement examples follow. Additional assembly language examples are given in *Chapter 3, Data Types and Addressing Modes.* Further information about assembly language syntax can be found in the *Intel 80960 Assembler Manual.*

```
subi      3, r5, r6          # r6 ← r5 - 3
setbit    13, g4, g5         # g5 ← g4 with bit 13 set
lda       0xfab3, r12        # r12 ← 0xfab3
ld        (r4), g3           # g3 ← memory location
                             # pointed to by r4
st        g10, (r6)[r7*2]    # g10 ← memory location
                             # pointed to by r6 + 2*r7
```

Branch Prediction

NOTE

Branch prediction is an implementation-specific feature of the i960 CA component. 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.

Because test and conditional-fault instructions also use condition codes, prediction suffixes are also implemented on these instructions. See *Appendix A, Optimizing Code for the i960 CA Microprocessor* for a complete discussion of prediction.

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 4.1):

register	REG	Most instructions are encoded in this format. Used primarily for instructions which perform register-to-register operations.
compare and branch	COBR	An encoding optimization which combines comparison and branch operations into one opword. Separate comparison 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. Instruction encoding formats are described in *Appendix D, Instruction Set Reference*.

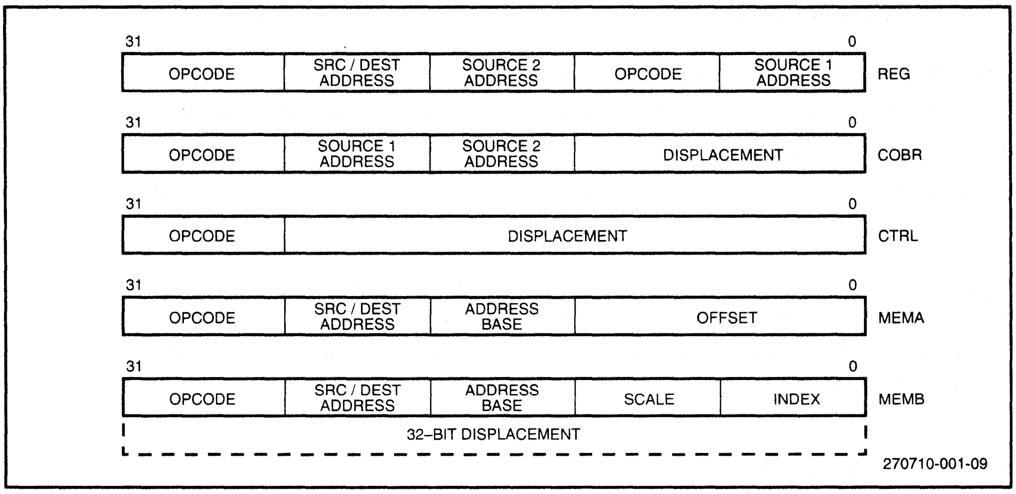


Figure 4.1. Machine-Level Instruction Formats

Instruction Operands

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

Format	Operand(s)	Description
REG	*src1*, *src2*, *src/dst*	*src1* and *src2* can be global registers, local registers, special function registers or literals. *src/dst* is either a global, local or special function register.
CTRL	*displacement*	CTRL format is used for branch and call instructions. *displacement* value indicates the target instruction of the branch or call.
COBR	*src1*, *src2*, *displacement*	*src1*, *src2* indicate values to be compared; *displacement* indicates branch target. *src1* can specify a global register, local register or a literal. *src2* can specify a global, local or special function register. See *Chapter 2, Programming Environment* for discussion of special function registers.
MEM	*src/dst*, *efa*	Specifies source or destination register and an effective address (*efa*) formed by using the processor's addressing modes described in *Chapter 3, Data Types and Memory Addressing Modes.* Registers specified in a MEM format instruction must be either a global or local register.

INSTRUCTION GROUPS

The i960 instruction set can be arranged into the following functional groups:

- Data Movement
- Arithmetic (Ordinal and Integer)
- Logical
- Bit, Bit Field and Byte
- Comparison
- Branch
- Call/Return
- Fault
- Debug
- Atomic
- Processor Management

Table 4.1 shows the instructions in these groups. 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 briefly overview each group's instructions.

DATA MOVEMENT

Data movement instructions are used to move data from memory to global and local registers; from global and local registers to memory; and data among local, global and special function registers.

NOTE

Rules for register alignment must be followed when using load, store and move instructions that move 8, 12 or 16 bytes at a time. Refer to the section *Memory Address Space* in *Chapter 2, Programming Environment* for alignment requirements for code portability across implementations.

Table 4.1. i960™ CA Microprocessor Instruction Set Summary

Data Movement	Arithmetic	Logical	Bit, Bit Field, and Byte
Load	Add	AND	Set Bit
Store	Subtract	NOT AND	Clear Bit
Move	Multiply	AND NOT	Not Bit
Load Address	Divide	OR	Alter Bit
	Add with carry	Exclusive OR	Scan For Bit
	Subtract with carry	NOT OR	Span Over Bit
	Extended Multiply	OR NOT	Extract
	Extended Divide	NOT	Modify
	Remainder	Exclusive NOR	Scan Byte For Equal
	Modulo	NOT	
	Shift	NAND	
	*Extended Shift		
	Rotate		
Comparison	**Branch**	**Call/Return**	**Fault**
Compare	Unconditional Branch	Call	Conditional Fault
Conditional Compare	Conditional Branch	Call Extended	Synchronize Faults
Check Bit	Compare and Branch	Call System	
Compare and Increment		Return	
Compare and Decrement		Branch and Link	
Test Condition Code			
Debug	**Atomic**	**Processor**	
Modify Trace Controls	Atomic Add	Flush Local Registers	
Mark	Atomic Modify	Modify Arithmetic Controls	
Force Mark		Modify Process Controls	
		*System Control	
		*DMA Control	

NOTE

Asterisk (*) denotes instructions that are i960 CA component-specific extensions to the i960 family's instruction set.

Load and Store Instructions

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

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

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

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

For **ld**, **ldob**, **ldos**, **ldib** and **ldis**, the instruction specifies a memory address and register and 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 operand and does not create a fault if truncation resulted in the loss of significant bits.

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.

mov	move word
movl	move long word
movt	move triple word
movq	move quad word

Load Address

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

On the i960 CA processor, **lda** is useful for performing simple arithmetic operations. The microprocessor's parallelism allows **lda** to execute in the same clock as another arithmetic or logical operation.

ARITHMETIC

Table 4.2 lists arithmetic operations and data types for which the i960 CA processor provides instructions. "X" in this table indicates that the microprocessor provides an instruction for the specified operation and data type. Extended shift right operation is an i960 CA component-specific extension to the i960 family's instruction set. All arithmetic operations are carried out on operands in registers. Refer to the section titled *Atomic Instructions* later in this chapter for instructions which 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 sections describe arithmetic instructions for ordinal and integer data types.

Table 4.2. Arithmetic Operations

	Data Types	
Arithmetic Operations	Integer	Ordinal
Add	X	X
Add with Carry	X	X
Subtract	X	X
Subtract with Carry	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	

*i960 CA component-specific extension to the 80960 instruction set.

Add, Subtract, Multiply and Divide

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

addi	add integer
addo	add ordinal
subi	subtract integer
subo	subtract ordinal
muli	multiply integer
mulo	multiply ordinal
divi	divide integer
divo	divide ordinal

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

Extended Arithmetic

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

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

addc adds two word operands (literals or contained in registers) plus condition code bit 1 (used here as a carry bit) in the AC Register. If the result has a carry, bit 1 of the condition code is set; otherwise, it is cleared. This instruction's description in *Chapter 9* 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).

Remainder and Modulo

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

remi	remainder integer
remo	remainder ordinal
modi	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.

Shift and Rotate

The processor provides the following shift instructions, which shift an operand a specified number of bits left or right:

shlo	shift left ordinal
shro	shift right ordinal
shli	shift left integer
shri	shift right integer
shrdi	shift right dividing integer
rotate	rotate left
eshro	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 a shift of the specified places would result 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 overflowing, and an integer-overflow fault is signaled.

shri performs a conventional arithmetic shift right operation by shifting the sign bit in from the most significant 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.

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

eshro is an i960 CA component-specific extension to the i960 family's instruction set. This 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.

LOGICAL

The following instructions perform bitwise Boolean operations on the specified operands:

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

These instructions all use the REG format and can specify literals or local, global or special function registers.

The processor provides logical operations in addition to **and**, **or** and **xor** as a performance optimization. This optimization reduces the number of instructions required to perform a logical operation and reduces the number of registers and instructions associated with bitwise mask storage and creation.

BIT AND BIT FIELD

These instructions perform operations on a specified bit or bit field in an ordinal operand. All use the REG format and can specify literals or local, global or special function registers.

Bit Operations

The following instructions operate on a specified bit:

setbit	set bit
clrbit	clear bit
notbit	not bit
alterbit	alter bit
scanbit	scan for bit
spanbit	span over bit

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

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

chkbit (described later in this chapter in the section titled *Comparison*) 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.

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, under control of a mask, into another register. Only unmasked bits in the destination register are modified. **modify** is equivalent to a bit field move.

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 according to the results of the comparison. This instruction uses the REG format and can specify literals or local, global or special function registers.

COMPARISON

The processor provides several types of instructions that are used to compare two operands, as described in the following sections.

Compare and Conditional Compare

The instructions listed below compare two operands then set the condition code bits in the AC register according to the results of the comparison.

cmpi	compare integer
cmpo	compare ordinal
concmpi	conditional compare integer
concmpo	conditional compare ordinal

These instructions 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 *Chapter 2, Programming Environment* for a discussion of meanings of the condition code for conditional operations.

cmpi and **cmpo** simply compare the two operands and set the condition code bits accordingly. **concmpi** and **concmpo** first check the status of bit 2 of the condition code. If it is not set, the operands are compared as with **cmpi** and **cmpo**. If bit 2 is 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 ≤ A ≤ C). Here, a compare instruction (**cmpi** or **cmpo**) checks one side of the range (e.g., A ≥ B) and a conditional compare instruction (**concmpi** or **concmpo**) checks the other side (e.g., A ≤ 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.

Compare and Increment or Decrement

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

cmpinci	compare and increment integer
cmpinco	compare and increment ordinal
cmpdeci	compare and decrement integer
cmpdeco	compare and decrement ordinal

These instructions 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., comparison and addition) to be executed in a single cycle. These instructions are intended for use at the end of iterative loops.

Test Condition Codes

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

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

These cause a TRUE (01H) to be stored in a destination register if the condition code matches the instruction-specified condition. Otherwise, a FALSE (00H) is stored in the register. All use the COBR format and can operate on local, global and special function registers.

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. See *Appendix A, Optimizing Code for the i960 CA Microprocessor* for a complete discussion of branch prediction.

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 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 80960 Assembler encodes the prediction with a mnemonic suffix of ".t" for true and ".f" for false. See the section of *Appendix A, Optimizing Code for the i960 CA Microprocessor* for a complete discussion of prediction.

Unconditional Branch

The following four instructions are used for unconditional branching:

b	Branch
bx	Branch Extended
bal	Branch and Link
balx	Branch and Link Extended

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

bal and **balx** store the next instruction's address in a specified register, then jump to the specified target IP. (For **bal**, the RIP is automatically stored in register g14; for **balx**, the RIP location is specified with an instruction operand.) As described in *Chapter 5, Procedure Calls* the 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).

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

Conditional Branch

With the 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:

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

All use the CTRL format. **bo** and **bno** are used with real numbers. Refer to *Chapter 2, Programming Environment* for a discussion of the condition code for conditional operations.

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:

cmpibe	compare integer and branch if equal
cmpibne	compare integer and branch if not equal
cmpibl	compare integer and branch if less
cmpible	compare integer and branch if less or equal
cmpibg	compare integer and branch if greater
cmpibge	compare integer and branch if greater or equal
cmpibo	compare integer and branch if ordered
cmpibno	compare integer and branch if unordered
cmpobe	compare ordinal and branch if equal
cmpobne	compare ordinal and branch if not equal
cmpobl	compare ordinal and branch if less
cmpoble	compare ordinal and branch if less or equal
cmpobg	compare ordinal and branch if greater
cmpobge	compare ordinal and branch if greater or equal
bbs	check bit and branch if set
bbc	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 and compare integer and branch instructions, two operands are compared and the condition code bits are set as described for compare instructions earlier in this chapter. A conditional branch is then executed as with the conditional branch (branch if) instructions.

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

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

CALL AND RETURN

The processor offers an on-chip call/return mechanism for making procedure calls. This integrated call/return mechanism is described in *Chapter 2, Programming Environment.* The following four instructions are provided to support this mechanism.

call	call
callx	call extended
calls	call system
ret	return

call and **ret** use the CTRL machine-instruction format. **callx** uses the MEM format and can specify local or global registers. **calls** uses the REG format and can specify local, 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 — procedures that provide a kernel or system-executive services. 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 also switches the processor to supervisor mode and 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 in *Chapter 5, 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.

CONDITIONAL 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. Faults are discussed in *Chapter 7, Faults.* The following conditional fault instructions permit a program to explicitly generate a fault according to the state of the condition code flags.

faulte	fault if equal
faultne	fault if not equal
faultl	fault if less
faultle	fault if less or equal
faultg	fault if greater
faultge	fault if greater or equal
faulto	fault if ordered
faultno	fault if 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 80960 Assembler encodes the prediction with a mnemonic suffix of ".t" for true and ".f" for false. See *Appendix A, Optimizing Code for the i960 CA Microprocessor* for a complete discussion of prediction.

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:

modpc	modify process controls
modtc	modify trace controls
mark	mark
fmark	force mark

These instructions use the REG format. Trace functions are controlled with bits in the processor's trace control register. These bits allow various types of tracing to be enabled or disabled. Other flags in the trace controls register indicate when an enabled trace event has been detected. Trace controls are described in *Chapter 8, Tracing and Debugging.*

modpc has the ability to enable/disable trace fault generation; **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.

The i960 CA component-specific **sysctl** instruction, described in the *Chapter 2, Programming Environment*, also provides control over breakpoint trace event generation. This instruction is used, in part, to load and control the i960 CA microprocessor's breakpoint registers.

ATOMIC INSTRUCTIONS

Atomic instructions perform read-modify-write operations on operands in memory. They allow a system to ensure that, when an atomic operation is performed on a specified memory location, the operation completes before another agent is allowed to perform an operation on the same memory. 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.

PROCESSOR MANAGEMENT

The following instructions control processor-related functions:

modpc	modify the process controls register
flushreg	flush cached local register sets to memory
modac	modify the AC register
sysctl	perform system control function
sdma	set up a DMA controller channel
udma	copy current DMA pointers to internal data RAM

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**) is provided to allow the AC register 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 an i960 CA component-specific extension to the i960 family's instruction set which is used to configure the on-chip bus controller, interrupt controller, breakpoint registers and instruction cache. The instruction also permits software to signal an interrupt or cause a processor reset and reinitialization. **sysctl** may only be executed by programs operating is supervisor mode. See *Chapter 2, Programming Environment* and *Chapter 9, Instruction Set Reference* for a complete description.

sdma and **udma** are i960 CA component-specific extensions to the i960 family's instruction set which configure and monitor the on-chip DMA controller. These instructions may only be executed by programs operating in supervisor mode. Refer to *Chapter 9, Instruction Set Reference* and *Chapter 13, DMA Controller* for a description of these instructions.

Procedure Calls 5

CHAPTER 5
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.

OVERVIEW

The i960 architecture supports two methods for making procedure calls:

- A RISC-style branch-and-link. This is a fast call best suited for calling procedures that do not call other procedures.
- An integrated call and return mechanism. This is 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**) is executed.

In many RISC architectures, a branch-and-link instruction is used as the base instruction for coding a procedure call. Register and stack management for the call is then handled by the user program. Since the i960 architecture provides a fully integrated call and return mechanism, coding calls with branch-and-link is not necessary. Additionally, the integrated call is much faster than typical RISC-coded calls.

The branch-and-link instruction in the i960 family, therefore, is used primarily for calling leaf procedures. Leaf procedures call no other procedures. They are called "leaf procedures" because they reside at the "leaves" of the call tree.

The integrated call and return mechanism is used in two ways in the i960 architecture: explicit calls to procedures in a user's program and implicit calls to interrupt and fault handlers. The remainder of this chapter explains the generalized call mechanism used for explicit and implicit calls and call and return instructions.

The processor performs two call actions:

local	When a local call is made, execution mode remains unchanged and the stack frame for the called procedure is placed on the *local stack.* The local stack refers to the stack of the calling procedure.
supervisor	When a supervisor call is made, execution mode is switched to supervisor and the stack frame for the called procedure is placed on the *supervisor stack.*

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.*

CALL AND RETURN MECHANISM

At any point in a program, the i960 device 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 is executed, a new stack frame is allocated for the called procedure. Additionally, the processor 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 is executed, the current local register set and current stack frame are deallocated. The previous local register set and previous stack frame are restored.

Local Registers and the Procedure Stack

For each procedure, the processor automatically allocates a set of 16 local registers. 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 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 5.1). 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 does not have to be written out to the save area in the stack frame in memory. Refer to the sections later in this chapter titled *Caching of Local Register Sets* and *Mapping the Local Registers to the Procedure Stack* for further discussion about local registers and procedure stack interrelations.

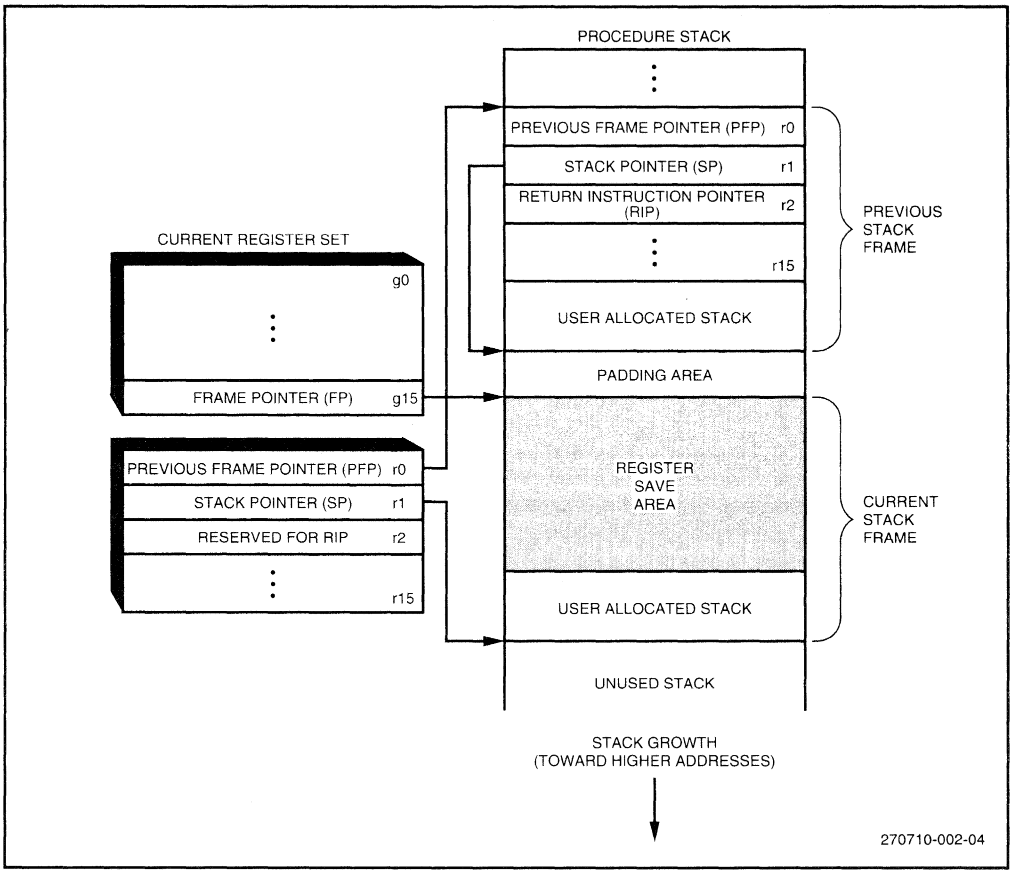


Figure 5.1. Procedure Stack Structure and Local Registers

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 5.1). The following paragraphs describe this linkage information.

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. In the i960 CA processor, frames are aligned on 16-byte boundaries (Figure 5.1). When the processor creates a new frame on a procedure call, it will, if necessary, add a padding area to the stack so that the new frame starts on a 16-byte alignment boundary.

Stack frame alignment is defined for each implementation of the i960 family. This alignment boundary is calculated from the relationship SALIGN*16. For example, if SALIGN is set to 4, stack frames are aligned on 64-byte boundaries. In the i960 CA microprocessor, SALIGN=1.

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 user 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.

Previous Frame Pointer

The previous frame pointer is the previous stack frame's first byte address. This address' 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.

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 later in this chapter in the section titled *Returns*.

Return Instruction Pointer

When a call is made, the processor saves the address of the instruction after the call, providing a reentry point when the return instruction is executed. This address is automatically stored in local register r2 of the calling frame. Register r2 is referred to as the return instruction pointer (RIP) register. The RIP register is a special register; do not use r2 to hold operand values. Since interrupts and faults trigger an implicit call action, the RIP register may be written at any time with the return pointer associated with the interrupt or fault event.

Call and Return Action

To clarify how procedures are linked and how the local register 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 described above.

The events for call and return operations are given in a logical order of operation. The i960 CA microprocessor is able to execute independent operations in parallel, therefore, many of these events execute simultaneously. For example, to improve performance, the processor often begins prefetch of the target instruction for the call or return before the operation is complete.

Call Operation

When a call instruction is executed or an implicit call is triggered, the processor performs the following operations:

1. The processor stores the instruction pointer for the instruction following the call in the current stack's RIP register (r2).
2. 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 the section titled *Returns* later in this chapter.)
3. The current local registers — including the PFP, SP and RIP registers — are saved, freeing these registers for use by the called procedure. Because saved local registers are cached on the i960 CA component, the registers are always saved in the on-chip local register cache at this time.
4. A new stack frame is allocated by using the stack pointer value saved in step 3. 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.
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 instruction, 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.

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 procedure to which it returns.

Caching of Local Register Sets

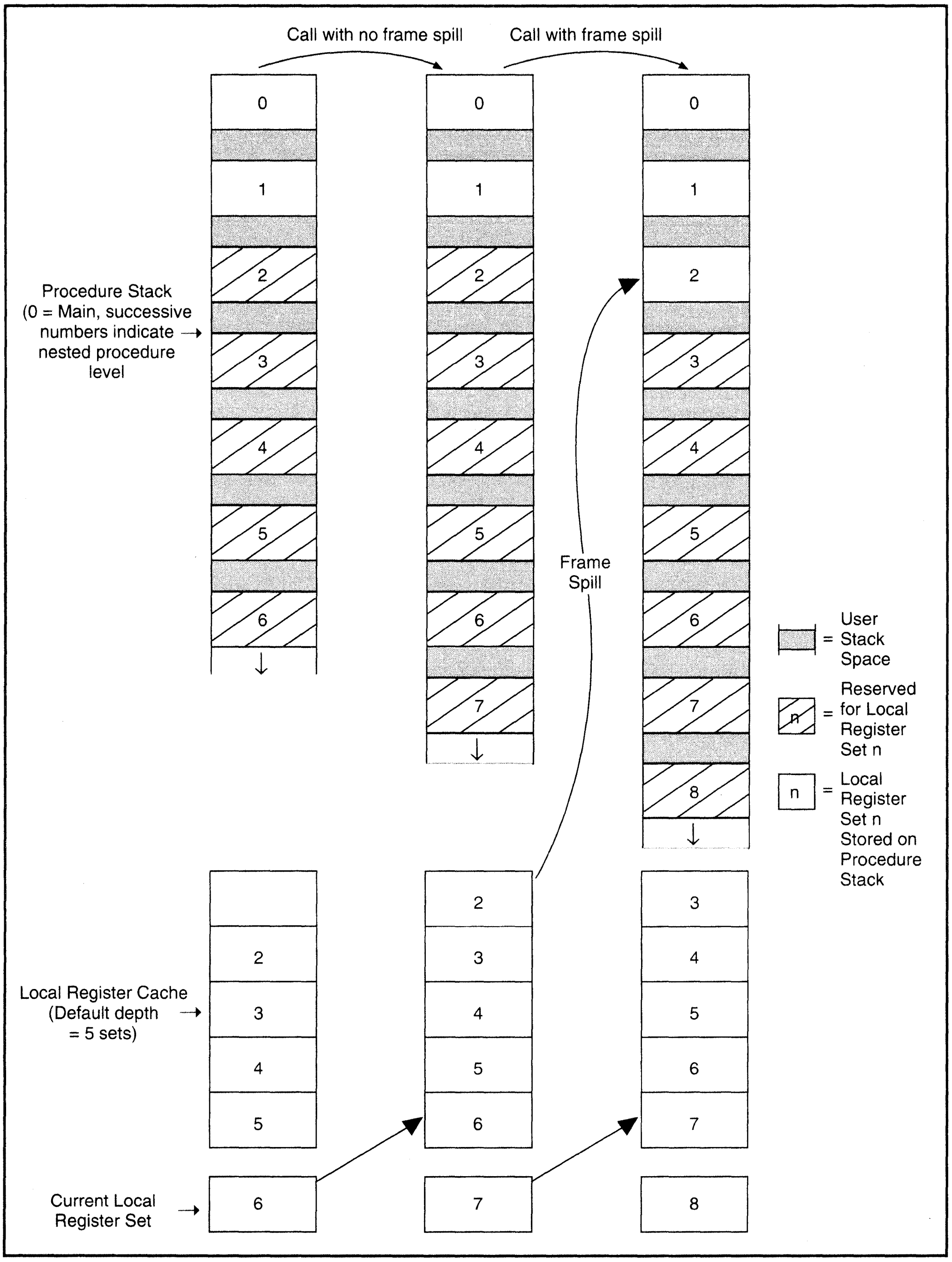
The i960 CA component provides a *local register cache* to improve call and return performance. Local registers are typically saved and restored from the local register cache when calls and returns are executed. For the i960 CA microprocessor, movement of a local register set between local registers and cache takes only four clock cycles. 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 or restored to their associated save areas in the procedure stack. Because these operations require access to external memory, this local cache miss impacts call and return performance.

When a call is made and the register cache is full, 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 5.2 illustrates a call operation with and without a frame spill.

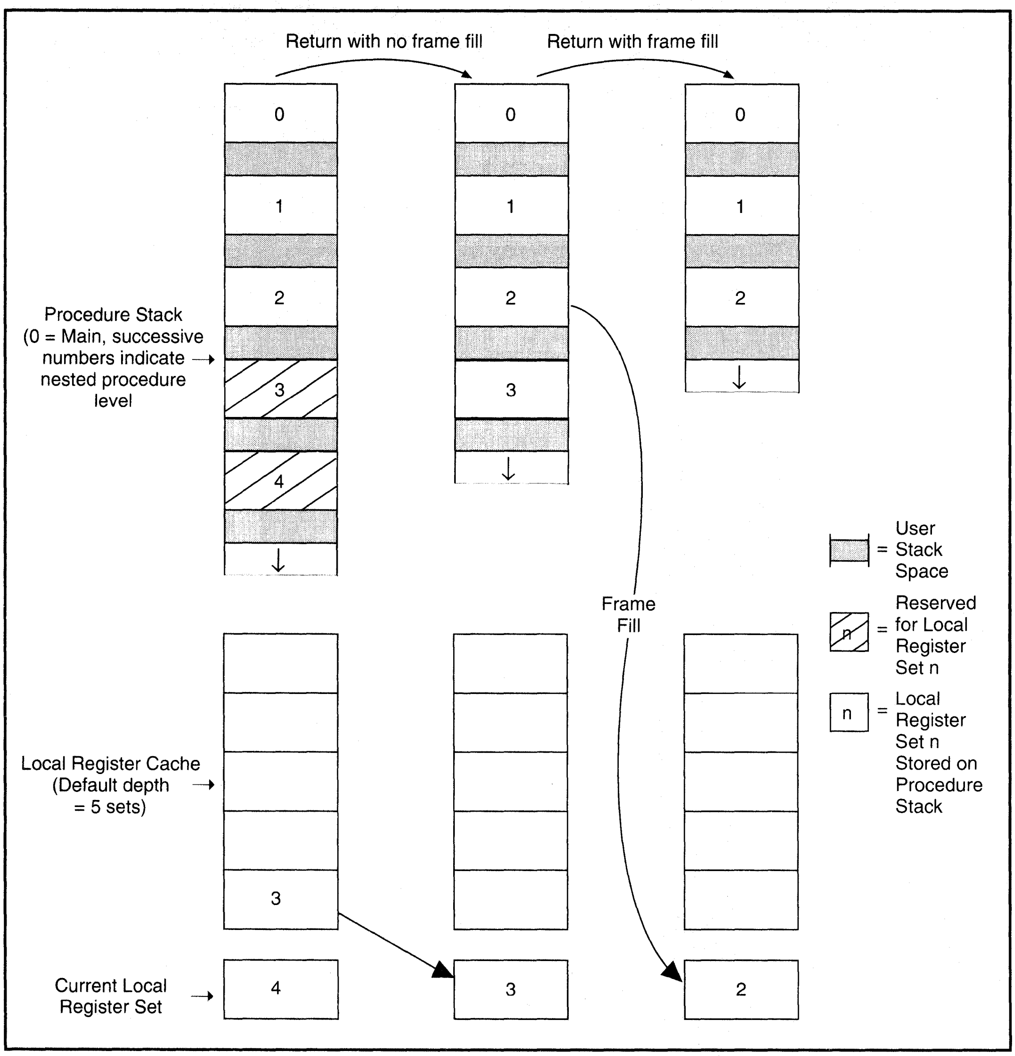
Similarly, when a return is made and the local register set for the target procedure is not available in the cache, these local registers must be retrieved from the procedure stack in memory. This operation is referred to as a *frame fill.* Figure 5.3 illustrates a return operation with and without a frame fill.

Register cache size is specified at initialization by the register cache configuration word value in the PRCB. Register cache size is adjustable to hold from 1 to 14 sets of local registers. See *Chapter 14, Initialization and System Requirements* for more information about initialization and the PRCB.



Call with no frame spill
Call with frame spill
Procedure Stack (0 = Main, successive numbers indicate nested procedure level
Frame Spill
Local Register Cache (Default depth = 5 sets)
Current Local Register Set
= User Stack Space
= Reserved for Local Register Set n
= Local Register Set n Stored on Procedure Stack
0
1
2
3
4
5
6
7
8

Figure 5.2. Frame Spill



Return with no frame fill
Return with frame fill
Procedure Stack (0 = Main, successive numbers indicate nested procedure level)
Local Register Cache (Default depth = 5 sets)
Current Local Register Set
Frame Fill
0
1
2
3
4
User Stack Space
Reserved for Local Register Set n
Local Register Set n Stored on Procedure Stack

Figure 5.3. Frame Fill

Up to five local register sets are cached by default with no impact to the processor's available resources. When the cache is configured for 6 to 14 sets, part of the internal data RAM is used to expand the cache. Data RAM usage begins at the highest address of internal RAM (03FFH) and grows downward.

The amount of internal data RAM used (in bytes) is determined by the formula:

n*16

where :	CCW = the programmed value of the cache configuration word in the PRCB			
and:	n = 0	for	CCW=0	Number of cached sets = 1
	n = 0	for	$1 \le CCW \le 5$	Number of cached sets = CCW
	n = CCW-5	for	$6 \le CCW \le 15$	Number of cached sets = CCW-1

Register cache cannot be disabled. Register cache size equals 1 when the cache configuration word is programmed to a value of 0. Also, a value of 5 or 6 produces the same cache number of cache sets; however, when programmed to 6, 16 bytes of internal data RAM is used, when programmed to 5, no internal data RAM is used.

The user program is responsible for preventing any corruption to the areas of internal RAM which are used for the register cache. In a typical program, most procedure calls and returns cause procedure depth to oscillate a few levels around a median call depth. The cache tends to be partially filled at the median call depth. Cache flushes occur when oscillations around the median depth are larger than the cache size can accommodate. Configuring local register cache to hold five sets of local registers avoids numerous cache fills and spills for most applications and does not use any of the data RAM which is available for general data storage. The user should configure the cache for a minimum of five register sets.

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 5.1). Saved local register sets are frequently cached on-chip rather than saved to memory. This caching is performed non-transparently. 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 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. The set must be written (or flushed) to memory because of a frame spill prior to the modification for the modification to be valid.

flushreg 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 register 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 operations. After **flushreg** is executed, the procedure stack may also be changed by modifying the previous frame in memory and executing a return operation.

NOTE

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.

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. Table 5.1 summarizes the global register model used by the i960 compilers. Refer to the *iC960 User's Guide* for details about the register allocation model.

This example illustrates a typical implementation of parameter passing between procedures and the use of the global and local registers in this scheme.

Parameter registers pass values into a function. Up to 12 parameters are passed by the value in the global registers. If the number of parameters exceeds 12, additional parameters are passed on the calling procedure's stack and a pointer to the argument block 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 5.1 illustrates parameter passing by value and reference.

Table 5.1. Global Register Function with i960™ Compilers

Instruction Operand	Value on Call	Value on Return
g0	Parameter 0	Return Argument 0
g1	Parameter 1	Return Argument 1
g2	Parameter 2	Return Argument 2
g3	Parameter 3	Return Argument 3
g4	Parameter 4	Not defined
g5	Parameter 5	Not defined
g6	Parameter 6	Not defined
g7	Parameter 7	Not defined
g8	Parameter 8/temp 5	Not defined/temp 5
g9	Parameter 9/temp 4	Not defined/temp 4
g10	Parameter 10/temp 3	Not defined/temp 3
g11	Parameter 11/temp 2	Not defined/temp 2
g12	temp 1	temp 1
g13	Return argument block pointer	Not defined
g14	Call parameter block pointer	Not defined
fp	Frame pointer (reserved)	

NOTE

If not used as parameters, g8 - g11 must be preserved by the called procedure. g0 - g11 may also be used for data storage. g14 must be set to 0 when not used as a parameter block pointer.

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 are 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 register, 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. This is necessary because the interrupt or fault occurs at any point in the user's program and return from 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 5.1. Using Global Register for Parameter Passing

```
# Example of parameter passing . . .
# C-source:   int a,b[10];
#             a = proc1(a,1,'x',&b[0]);
#             assembles to ...
    mov       r3,g0         # value of a
    ldconst   1,g1          # value of 1
    ldconst   120,g2        # value of 'x'
    lda       0x40(fp),g3   # reference to b[10]
    call      _proc1
    mov       g0,r3         #save return value in "a"
              .
              .
_proc1:
    movq      g0,r4         # save parameters
              .
              .             # other instructions in procedure
              .             # and nested calls
    mov       r3,g0         # load return parameter
    ret
```

LOCAL CALLS

A local call does not cause a stack switch. A local call can be made two ways: 1) with the **call** and **callx** instructions or 2) with a system-local call (system-local call is described in the following section titled *System Calls*). **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. See *Chapter 9, Instruction Set Reference* for a further description of **call** and **callx**.

When a local call is made with a **call** or **callx**, the processor performs the same operation as described earlier in this chapter in the section titled *Call Operation*. The target IP for the call is derived from the instruction's operands and the new stack frame is allocated on the current stack. **call** and **callx** algorithms are further described in *Chapter 9, Instruction Set Reference.*

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** — 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. See *Chapter 9, Instruction Set Reference* for a further description of **calls**.

Using an i960 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 assembler documents 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 *Chapter 2, Programming Environment.*

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 a system call; or fault handling procedures, which the processor can access through its fault handling mechanism. Using the system procedure table to store IPs for fault handling is described in *Chapter 7, Faults.*

System procedure table structure is shown in Figure 5.4. It is 1088 bytes in length and can have up to 260 procedure entries. The processor gets a pointer to the system procedure table at initialization. The following sections describe this table's fields.

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 (or 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 5.2). On a system call, the processor performs different actions depending on the type of call selected.

Table 5.2. Encodings of Entry Type Field in System Procedure Table

Encoding	Call Type
00_2	System-Local Call
01_2	Reserved
10_2	System-Supervisor Call
11_2	Reserved

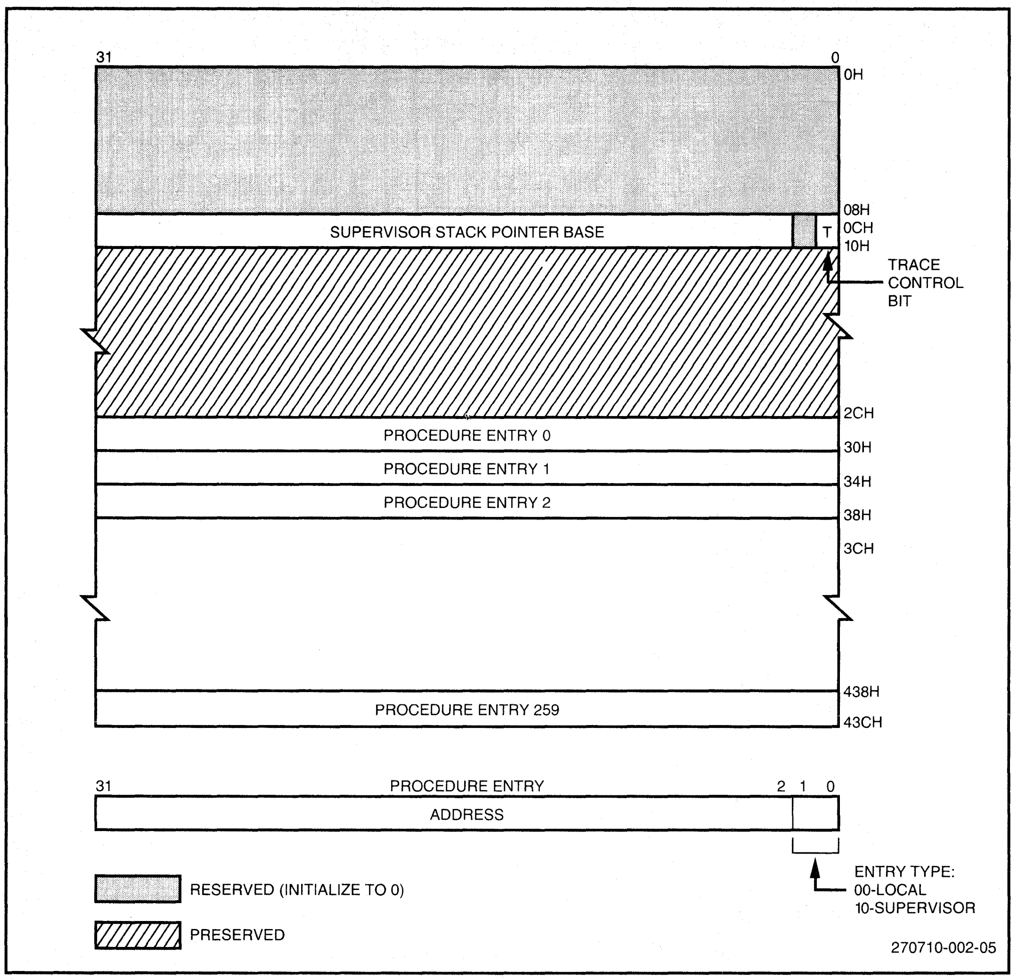


Figure 5.4. System Procedure Table

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 5.4) 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.

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 *Chapter 8, Tracing and Debugging.*

System-Local Call

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 earlier in this chapter's section titled *Call Operation*. The call's target IP is taken from the system procedure table and the new stack frame is allocated on the current stack. The **calls** algorithm is described in *Chapter 9, Instruction Set Reference.*

System-Supervisor Call

When a **calls** instruction references an entry in the system procedure table with an entry type of 10_2, the processor executes a system-supervisor call to the selected procedure. The call's target IP is taken from the system procedure table. The processor performs the same action as described earlier in this chapter's section titled *Call Operation*, with the following exceptions:

- If the processor is in user mode, it switches to supervisor mode.
- The new frame for the called procedure is placed on the supervisor 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.

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 *Chapter 2, Programming Environment.*

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 5.1). The base stack pointer for the supervisor stack is automatically read from the system procedure table and cached internally at initialization or when the processor is reinitialized with **sysctl**. 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 *Chapter 14, Initialization and System Requirements*). The base stack pointers must be aligned to a 16-byte boundary; otherwise, the first frame pointer in the stack is rounded up to the next 16-byte boundary.

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 7, Faults* and *Chapter 6, Interrupts* for more information on the structure of the fault and interrupt records.

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** is executed, the processor uses the information from the return-type field in the PFP register (Figure 5.5) to determine the type of return action to take.

return-type field indicates the type of call which was made. Table 5.3 shows the return-type field encoding for the various calls: local call, supervisor call, interrupt call and fault call.

trace-on-return flag (PFP.rt0 or bit 0 of the return-type field) stores the trace enable bit value when a 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 *Chapter 8, Tracing and Debugging.*

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 *Chapter 8, Tracing and Debugging* for a discussion of interaction between call-trace and prereturn-trace modes with the prereturn-trace flag.

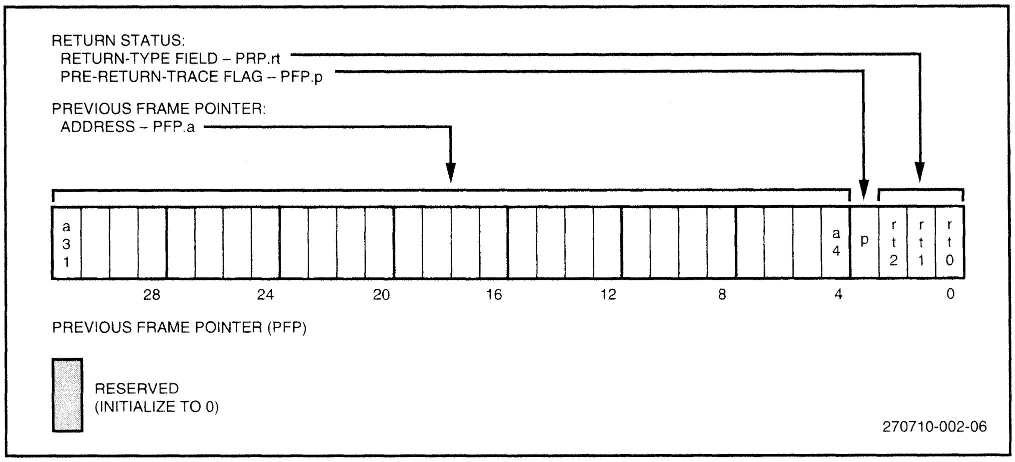

RETURN STATUS:
RETURN-TYPE FIELD – PRP.rt
PRE-RETURN-TRACE FLAG – PFP.p
PREVIOUS FRAME POINTER:
ADDRESS – PFP.a
a31
a4
p
rt2
rt1
rt0
28
24
20
16
12
8
4
0
PREVIOUS FRAME POINTER (PFP)
RESERVED (INITIALIZE TO 0)
270710-002-06

Figure 5.5. Previous Frame Pointer Register (PFP) (r0)

Table 5.3. Encoding of Return Status Field

Return Status Field	Call Type	Return Action
p000	Local call (system-local call or system-supervisor call made from supervisor mode)	Local return (return to local stack; no mode switch)
p001	Fault call	Fault return (See *Chapter 7, Faults*)
p01t	System-supervisor from user mode	Supervisor return (return to user stack, mode switch to user mode, trace enable bit is replaced with the t bit stored in the PFP register on the call.
p100	reserved	
p101	reserved	
p110	reserved	
p111	Interrupt call	Interrupt return (See *Chapter 6, Interrupts.*)

NOTE:

"p" is PFP.p (prereturn trace flag). "t" denotes the trace-on-return flag. This flag is used only for system supervisor calls which cause a user-to-supervisor mode switch.

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 is executed, 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.

Interrupts 6

CHAPTER 6
INTERRUPTS

This chapter describes how a programmer:

- uses the processor's interrupt mechanism
- defines data structures used for interrupt handling
- describes actions that the processor takes when handling an interrupt

Chapter 12, Interrupt Controller describes the mechanism for signaling and posting interrupts; it is best suited for a system implementor.

OVERVIEW

An interrupt is an event that causes a temporary break in program execution so the processor can handle another chore. Interrupts commonly request I/O services or synchronize the processor with some external hardware activity. For interrupt handler portability across implementations of the i960 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 CA processor provides an on-chip programmable interrupt controller.

Requests for interrupt service come from many sources. These requests are transparently prioritized so that instruction execution is redirected only if an interrupt request is of higher priority than that of the executing task.

When the processor is redirected to service an interrupt, it uses a vector number that accompanies the interrupt request to locate the interrupt table — an entry in a data structure. From that entry, it gets a vector 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. A new frame is created 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*. The mechanism the i960 CA device uses for posting interrupts is described in *Chapter 12, Interrupt Controller.*

On the i960 CA processor, interrupt requests may originate from external hardware sources, internal DMA sources or from software. External interrupts are detected with the chip's 8-bit interrupt port and with a dedicated $\overline{\text{NMI}}$ input. Interrupt requests originate from software by the **sysctl** instruction which signals interrupts. To manage and prioritize all possible interrupts, the microprocessor integrates an on-chip programmable interrupt controller. Integrated interrupt controller configuration and operation is described in *Chapter 12, Interrupt Controller.*

The i960 architecture defines two data structures to support interrupt processing (see Figure 6.1): the interrupt table and interrupt stack. The interrupt table contains 248 vectors for interrupt handling procedures and an area for posting software requested interrupts. The interrupt stack prevents interrupt handling procedures from overwriting 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).

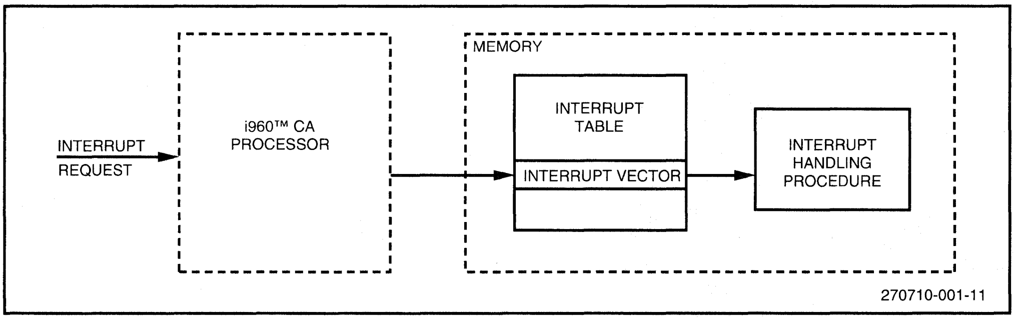


Figure 6.1. Interrupt Handling Data Structures

INTERRUPT PRIORITY

To provide transparent prioritization of the 248 possible interrupts, interrupt vectors are grouped into 31 distinct levels of priority, with eight vectors per priority.

Every interrupt request is associated with an interrupt vector in the interrupt table. The table contains 248 vectors: from vector number 8, assigned the lowest priority, to vector number 255, the highest priority. Since there are 31 priority levels, each vector's priority is determined by the vector number's upper five bits. Thus, at each priority level, there are eight possible vector numbers. When multiple interrupt requests are pending at the same priority level, the highest vector number is serviced first.

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 the interrupt request priority is higher than the processor's current priority (the priority of the program or interrupt the processor is executing). If the interrupt priority is less than or equal to the processor's current priority, the processor does not service the request.

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. The processor may post requests for later servicing. Interrupts waiting to be serviced, called *pending interrupts*, are discussed later in this chapter.

NOTE

On the i960 CA processor implementation, the non-maskable interrupt (NMI) interrupts priority-31 execution; no interrupt can interrupt an NMI handler.

The lowest program priority allowed is 0. If the current program has a 0 priority, a priority-0 interrupt is never accepted. This is why vectors 0 through 7 cannot be used. In fact, no entries are provided for these vectors in the interrupt table.

INTERRUPT TABLE

The interrupt table (Figure 6.2), 1028 bytes in length, can be located anywhere in the non-reserved address space; it must be aligned on a word boundary. The processor reads a pointer to 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.

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

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 which points to a vector entry in the interrupt table. The vector entry section contains 248 one-word entries. Vector numbers 0 through 7 are not defined and do not have associated entries in the interrupt table. Vector numbers 8 through 243 and 252 through 255 and their associated vector entries are used for conventional interrupts. Vector number 244 through 247 and 249 through 251 are reserved; do not use these. Vector number 248 and its associated vector entry is used for the non-maskable interrupt (NMI).

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.

Vector entry structure is given at the bottom of Figure 6.2. Each interrupt procedure must begin on a word boundary, so the processor assumes that the vector's two least significant bits are 0. Bits 0 and 1 of an entry indicate entry type: type 00_2 indicates that the interrupt procedure should be fetched normally; type 10_2 indicates that the interrupt procedure should be fetched from the locked partition of the instruction cache (see Chapter 12 section titled *Caching of Interrupt Handling Procedures*). The other possible entry types are reserved and must not be used.

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 represents 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 represent an interrupt vector. 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.

Posting Interrupts

For the i960 CA component, only software-requested interrupts are posted in the interrupt table; hardware-requested interrupts are posted in the interrupt pending (IPND) register. This register and the mechanism for requesting and posting hardware interrupts is described *Chapter 12, Interrupt Controller.* Software posting of interrupts in the interrupt table can assist an application in prioritizing processing demands as follows:

- By posting interrupt requests in the interrupt table, the application can delay the servicing of low priority tasks which were signaled by a higher priority interrupt.
- In systems with more than one processor, both processors can post and service interrupts from a shared interrupt table. This interrupt table sharing allows processors to share the interrupt handling load or provide a communication mechanism between the processors.

To post a pending interrupt in the memory-resident interrupt table, the processor performs the following atomic read/write operation that locks the interrupt table until the posting operation has completed.

```
# x and z are temporary registers
x ← atomic_read(pending_priorities);          # assert LOCK pin
z ← read(pending_interrupts(vector_number/8));
x(vector_number/8) ← 1;
z(vector_number mod 8) ← 1;
write(pending_interrupts(vector_number/8)) ← z;
atomic_write(pending_priorities) ← x;         # deassert LOCK pin
```

The $\overline{\text{LOCK}}$ pin can be used to prevent other agents on the bus from accessing the interrupt table during the posting operation. On the i960 CA microprocessor, posting software interrupts is performed by **sysctl**.

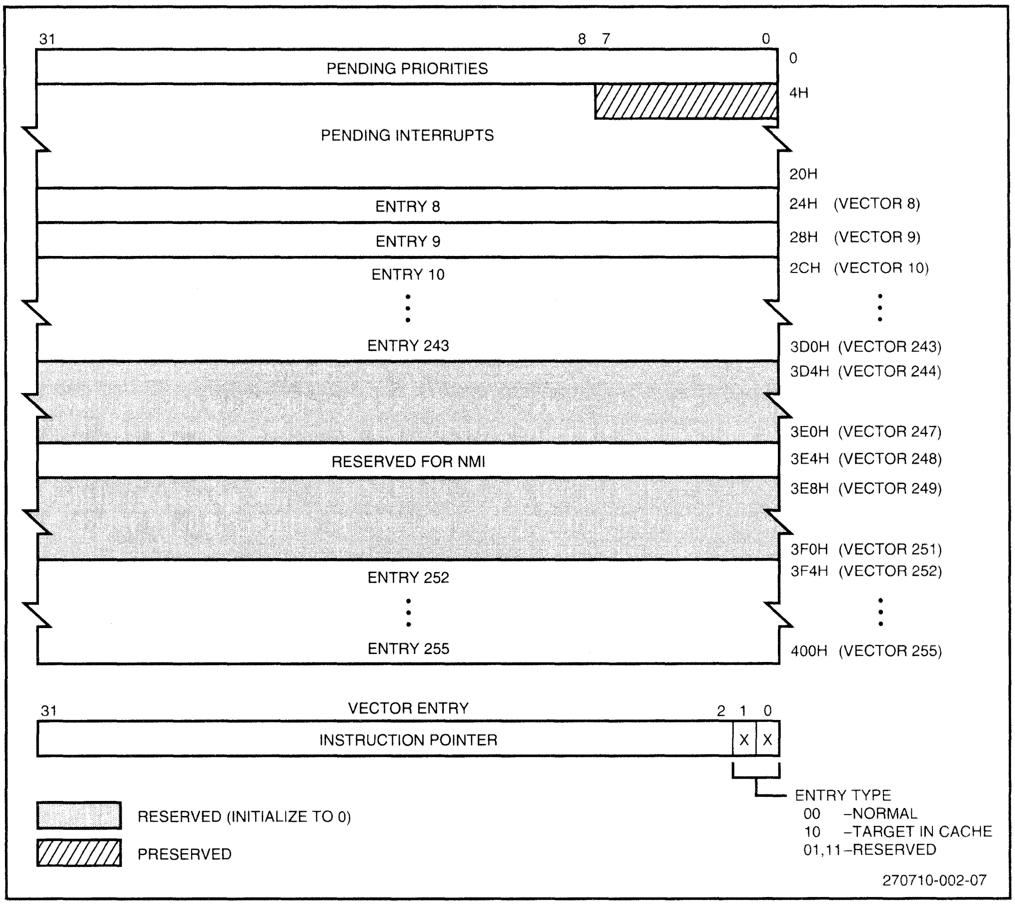


Figure 6.2. Interrupt Table

Posting Interrupts Directly to the Interrupt Table

The i960 CA processor — or external agent that is sharing memory with the microprocessor (such as an I/O processor or another i960 CA device) — can post pending interrupts directly in the interrupt table by setting the appropriate bits in the pending priorities and pending interrupts fields. This action, however, does not ensure that the core will handle the interrupt immediately, nor does it cause the core to update the value in the software priority register. To do this, the **sysctl** instruction should be used as described above.

sysctl can be used at any time to explicitly force the core to check the interrupt table for pending interrupts. This is done by specifying a vector number with a priority of zero (that is, vector numbers 0 to 7). For example, when an external agent is posting interrupts to a shared interrupt table, **sysctl** could be executed periodically to guarantee recognition of pending interrupts which were posted in the table by the external agent.

An external I/O agent or a coprocessor posts interrupts to a processor's interrupt table in memory in the same manner described above, providing it has the capability to perform atomic operations on memory. When interrupts are posted in this manner, pending interrupts and pending priorities must be modified in specific order and not allow access by the processor or other external agents during the atomic modify operations:

```
#set pending interrupt bit
atomic_modify(pending_interrupts(vector_number/8));
#set pending priority bit
atomic_modify(pending_priorities);
```

The processor automatically checks the memory-based interrupt table when the processor posts an interrupt using **sysctl** with a post interrupt message type.

When the processor finds a pending interrupt, it handles it as if it had just received the interrupt. If the processor finds two pending interrupts at the same priority, it services the interrupt with the highest vector number first.

NOTES

1. When a modify-process-controls (**modpc**) instruction causes a program's priority to be lowered, other i960 family members check for pending interrupts in the memory-based interrupt table; the i960 CA device internally stores the priority of the highest pending interrupt found in the interrupt table's pending interrupts field. To improve performance, the stored priority is checked — rather than the memory-based interrupt table — when **modpc** changes a process priority. The internal priority value is updated each time an interrupt is posted using **sysctl**.
2. i960 architecture does not define a portable method for posting interrupts. Different implementations may implement optimized interrupt posting mechanisms. The i960 CA device records pending interrupts differently depending upon interrupt type and interrupt controller configuration. See this chapter's sections titled *Interrupt Modes* and *Software Generated Interrupts*.

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 access to certain interrupt vectors and to the pending interrupt information without having to make memory accesses. The microprocessor caches the following:

- The value of the highest priority posted in the pending priorities field.
- A predefined subset of interrupt vectors (that is, interrupt vector entries from the interrupt table).

- Pending interrupts received from external interrupt pins and on-chip DMA controller (hardware requested interrupts).

This caching mechanism is non-transparent; in other words, the processor may modify fields in a cached interrupt table without modifying the same fields in the interrupt table itself (*non-transparent caching*). Vector caching is described in *Chapter 12, Interrupt Controller.*

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 *Chapter 5, Procedure Calls*. 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 6.3 shows the structure of this interrupt record.

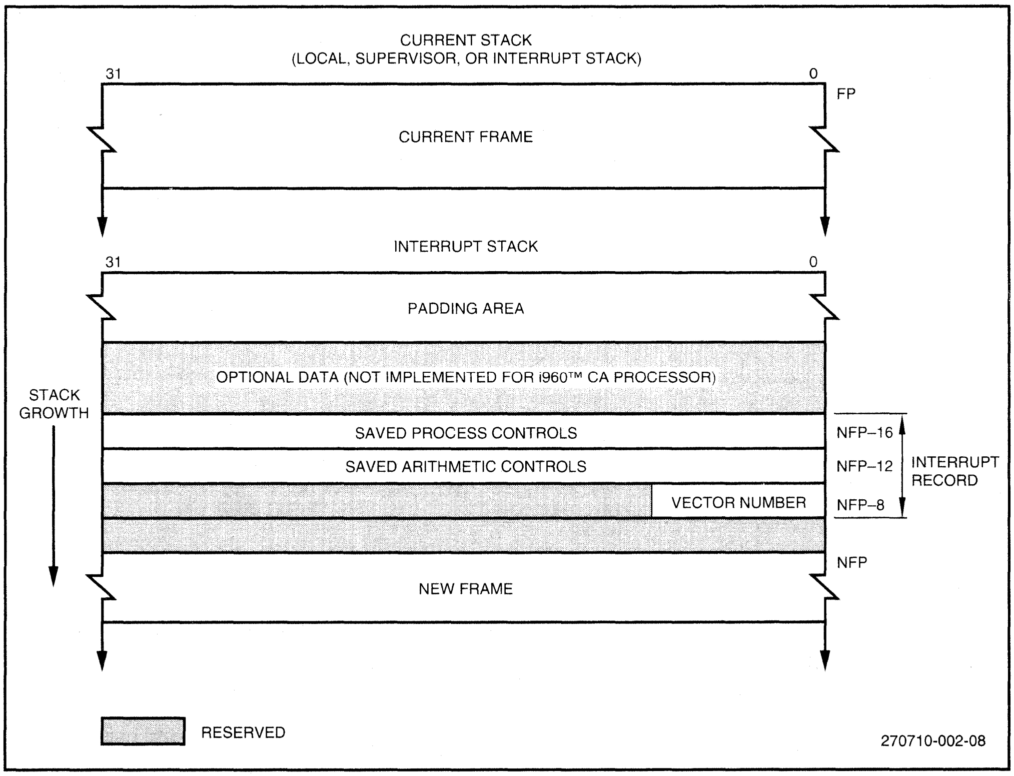


Figure 6.3. Storage of an Interrupt Record on the Interrupt Stack

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 received and the interrupt vector number used. Referenced to the new frame pointer address (designated NFP), the saved AC register is located at address NFP-12; the saved PC register is located at address NFP-16.

The interrupt record may also contain a resumption record which stores the context of instructions which began — but not completed — when the interrupt was serviced. Although the i960 CA processor never creates a resumption record, portable programs must tolerate interrupt stack frames with and without resumption records.

INTERRUPT HANDLER PROCEDURES

An interrupt handling procedure performs a specific action that is associated with a particular interrupt vector. For example, one interrupt handler task might be to initiate a DMA transfer. The interrupt handler procedures can be located anywhere in the non-reserved address space. Since instructions in the i960 family 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.

CAUTION!

Interrupt procedures must preserve and restore the state of any resources shared with a non-cooperating program. For example, an interrupt procedure which uses a global register which is not permanently allocated to it should save the register's contents before it uses 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 Chapter 12 section titled *Caching of Interrupt Handling Procedures* for a complete description.

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 to be serviced, 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 will already be in use. This is called an *interrupted-state* interrupt.

The following two sections 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.

Executing-State Interrupt

When the processor receives an interrupt while in the executing state (i.e., executing a program), it performs the following actions, regardless of whether the processor is in user or supervisor mode when the interrupt occurs:

- The new frame pointer (FP) for the interrupt handler is set to point to the interrupt stack and is incremented to create space for an interrupt record (see Figure 6.3). The interrupt record is described earlier in this chapter's section titled *Interrupt Record.* The current state of the AC register, PC register and interrupt vector number are saved in the interrupt record.
- The processor stores the interrupt return status (111_2) in the current PFP's return status field then changes the following fields and flags in the PC register:
  - Sets state flag (bit 13) to interrupted.
  - Sets execution mode flag (bit 1) to supervisor; processor switches to supervisor mode.
  - Sets priority field (bits 16-20) to the priority of the interrupt. Setting the processor's priority to that of the interrupt ensures that lower priority interrupts cannot interrupt current interrupt servicing.
  - Sets to 0 the trace-fault-pending flag (bit 10) and trace-enable bit (bit 0). Clearing these bits allows the interrupt to be handled without trace faults being raised.
- The processor performs a call operation as described in *Chapter 5, Procedure Calls.* The target IP for the call is the selected entry in the interrupt table.

When the processor executes a return operation and the return-type field is 111_2, it performs the following:

- The interrupt record's arithmetic controls and process controls fields are copied into the AC and PC registers, respectively. Restoring the PC register causes the processor's state to be returned to executing and its execution mode and priority to be returned to what they were prior to the interrupt. It also returns the trace-fault-pending flags and trace-enable bit to their value before the interrupt occurred.

NOTE

If the interrupt handling procedure sets execution mode to user prior to the return, the PC register is not restored upon return.

- Pending interrupts that need to be handled — such as pending interrupts with higher priority than that of the program being returned to — are handled at this time, prior to returning to the previously interrupted program. If the trace-fault-pending flag and trace-enable bit are set, the trace fault is handled at this time.

- The processor then performs a return operation as described in *Chapter 5, Procedure Calls.* This causes the processor to switch back to the local stack or supervisor stack; whichever it was using when interrupted.

Assuming that there are no pending interrupts to be serviced or trace faults to be handled, the processor resumes work on the interrupted program upon completion of the return operation.

Interrupted-State Interrupt

If the processor is servicing an interrupt and receives an interrupt with a higher priority, the current interrupt handler routine is interrupted. Here, the processor performs the same action to save the interrupted interrupt handler routine's state, as described in the previous section for an executing-state interrupt. The interrupt record is saved on the top of the interrupt stack, prior to the new frame that is created for servicing the new interrupt.

On 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.

REQUESTING INTERRUPTS

On the i960 CA microprocessor, interrupt requests may originate from external hardware sources, internal DMA sources or from software. External interrupts are detected with the chip's 8-bit interrupt port and with a dedicated $\overline{\text{NMI}}$ input. Interrupt requests originate from software by the **sysctl** instruction which signals interrupts. To manage and prioritize all possible interrupts, the microprocessor integrates an on-chip programmable interrupt controller. The configuration and operation of the integrated interrupt controller is described in *Chapter 12, Interrupt Controller.*

Interrupts may be requested directly by a user's program. This mechanism is often useful for requesting and prioritizing low-level tasks in a real time application.

Software can request interrupts in the following two ways:

1. With the **sysctl** instruction.
2. By the i960 CA microprocessor, or another processor, posting an interrupt in the interrupt table's pending-interrupts and pending-priorities fields.

SYSTEM CONTROL INSTRUCTION (sysctl)

sysctl is typically used to request an interrupt in a program (Example 6.1). The request interrupt message type (00H) is selected and the interrupt vector number is specified in the least significant byte of the instruction operand. (See *Chapter 2, Programming Environment* for a complete discussion of **sysctl.**)

Example 6.1. Requesting an Interrupt with the sysctl Instruction

```
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 core when it executes the **sysctl** instruction is as follows:

1. The core performs an atomic write to the interrupt table and sets bits in the pending-interrupts and pending-priorities fields that correspond to the requested interrupt.
2. The core 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 are taken:

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

Faults 7

CHAPTER 7
FAULTS

This chapter describes the i960 CA processor's fault handling facilities. Subjects covered include the fault handling data structures and fault handling mechanism. A reference section at the end of the chapter contains detailed information on each fault type.

FAULT HANDLING FACILITIES OVERVIEW

The architecture defines various conditions in code or the processor's internal state that could cause the processor to deliver incorrect or inappropriate results or that could cause it to head down 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, for inappropriate operand values and for invalid opcodes and addressing modes.

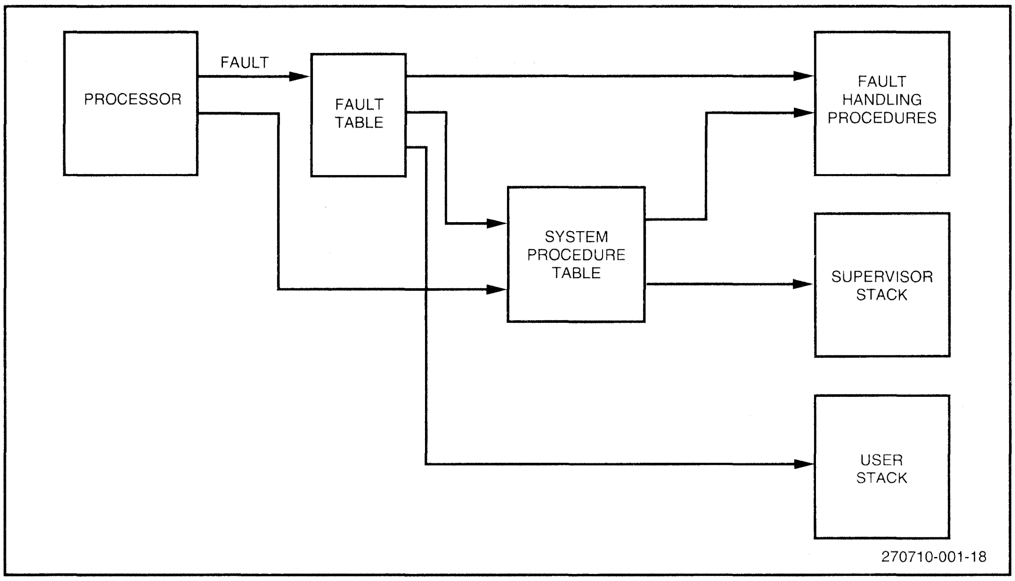


Figure 7.1. Fault-Handling Data Structures

As shown in Figure 7.1, the architecture defines a fault table, a system procedure table, a set of fault handling procedures and a stack (user stack, supervisor stack or both) to handle processor-generated faults.

The fault table contains pointers to fault handling procedures. The system procedure table is optionally used to provide an interface to any fault handling procedures and to allow 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.

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

The processor can detect a fault at any time while executing instructions, whether from a program, interrupt handling procedure or fault handling procedure. If a fault occurs when executing a program, 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 (also through the system procedure table)

As part of the implicit call to the fault handling procedure, the processor creates a fault record on the stack — the stack in use by the fault handling procedure. This record includes information on the fault and the processor's state when the fault was generated.

Following fault record creation, the processor begins executing the selected fault handling procedure. If the fault handling procedure recovers from the fault, the processor then restores itself to its state prior to the fault and resumes work on the program with no break in program control flow. If the fault handling procedure is not able to recover from the fault, the fault handler can call a debug monitor or perform an action such as resetting the processor.

The procedure call mechanism described above is used to handle faults that occur while the processor is servicing an interrupt or that occur while the processor is working on another fault handling procedure.

FAULT TYPES

The i960 architecture defines a basic set of faults which are categorized by type and subtype. Each fault has a unique type number and a subtype number. When the processor detects a fault, it records the fault type and subtype numbers in a fault record. It then uses the type number to select a fault handling procedure.

The fault handling procedure has the option of using the subtype number to select a specific fault handling action. The i960 CA processor recognizes i960 architecture-defined faults and a new fault subtype for detecting unaligned memory accesses. Table 7.1 lists all faults that the i960 CA processor detects, arranged by type and subtype. Text that follows the table gives column definitions.

Table 7.1. i960™ CA Processor Fault Types and Subtypes

Fault Type		Fault Subtype		Fault Record
Number	*Name*	*Number/Bit Position*	*Name*	
1H	Trace	Bit 1	Instruction Trace	XX01 XX02H
		Bit 2	Branch Trace	XX01 XX04H
		Bit 3	Call Trace	XX01 XX08H
		Bit 4	Return Trace	XX01 XX10H
		Bit 5	Prereturn Trace	XX01 XX20H
		Bit 6	Supervisor Trace	XX01 XX40H
		Bit 7	Breakpoint Trace	XX01 XX80H
2H	Operation	1H	Invalid Opcode	XX02 XX01H
		2H	Unimplemented	XX02 XX02H
		3H	Unaligned (see note)	XX02 XX03H
		4H	Invalid Operand	XX02 XX04H
3H	Arithmetic	1H	Integer Overflow	XX03 XX01H
		2H	Arithmetic Zero-Divide	XX03 XX02H
4H	Reserved (Floating Point)			
5H	Constraint	1H	Constraint Range	XX05 XX01H
		2H	Privileged	XX05 XX02H
6H	Reserved			
7H	Protection	2H	Length	XX07 XX01H
8H - 9H	Reserved			
AH	Type	1H	Type Mismatch	XX0A XX01H
BH - FH	Reserved			

NOTE

The operation-unaligned fault is an i960 CA processor-specific extension.

The first column of Table 7.1 gives fault type numbers in hexadecimal; the second column gives the fault type name.

The third column gives the fault subtype number: as a hexadecimal number or as a bit position in the 8-bit fault subtype field in the fault record. The bit position method of indicating a fault subtype is used for faults such as trace faults, where it is possible for two or more fault subtypes to be generated simultaneously.

The fourth column gives the fault subtype name. For convenience, individual faults are referred to in this manual by their fault-subtype name. Thus an *operation-invalid-operand fault* is referred to as simply an *invalid-operand fault* or an *arithmetic-integer-overflow fault* is referred to as an *integer-overflow fault*.

The fifth column of Table 7.1 shows the encoding of the word in the fault record that contains the fault type and fault subtype numbers.

Other i960 family members may provide different extensions that recognize additional fault conditions. Fault type and subtype encoding allows any of these additional faults to be included in the fault table along with the basic faults. Space in the fault table is reserved in such a way that specific implementation-defined faults are encoded the same for each processor that uses them. For example, Fault Type 4 is reserved for floating point faults. Any of the i960 family processors that provide floating point operations use Entry 4 to store the pointer to the floating point fault handling procedure.

FAULT TABLE

The fault table (Figure 7.2) provides the processor with a pathway to fault handling procedures. It can be located anywhere in the address space. The processor obtains a pointer to the fault table during initialization.

There is one entry in the fault table 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 a fault handling procedure is called, it has the option of reading the fault subtype or subtypes from the fault record, to determine the appropriate fault recovery action.

As shown in Figure 7.2, two fault table entry types are allowed: local-call entry and system-call entry. Each entry type is two words long. The entry type field (bits 0 and 1 of the first word of the entry) and the value in the second word of the entry determine the entry type.

A local-call entry (type 00) provides an instruction pointer (address in the address space) for the fault handling procedure. Using this entry, the processor invokes the specified procedure by means of an implicit local-call operation. The second word of a local procedure entry is reserved. It should be set to zero when the fault table is created and not accessed after that.

A system-call entry provides a procedure number in the system procedure table. This entry must have an entry type of 10 and a value in the second word of 0000 027FH. Using this entry, the processor invokes the specified fault handling procedure by means of an implicit call-system operation similar to that performed for the **calls** instruction. A fault handling procedure in the system procedure table can be called with a system-local call or a system-supervisor call, depending on the entry type in the system-procedure table.

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.

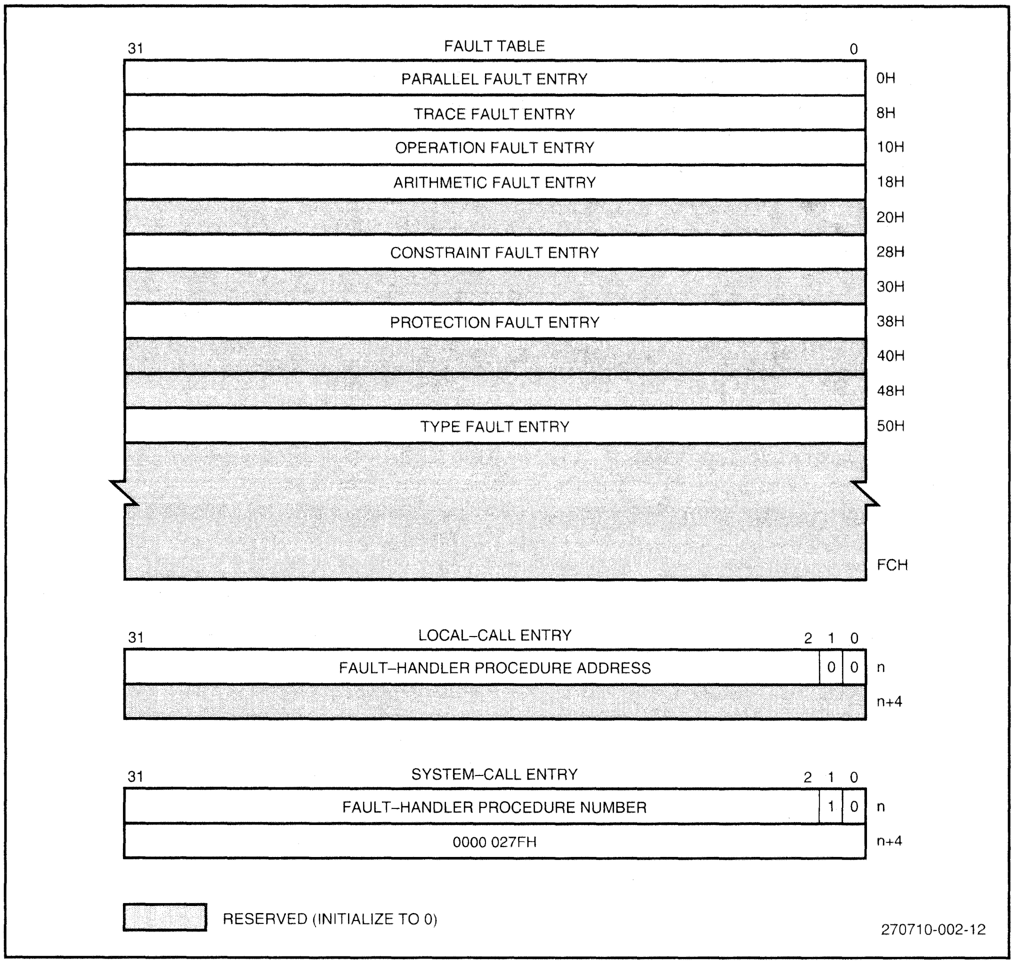


Figure 7.2. Fault Table and Fault Table Entries

STACK USED IN FAULT HANDLING

The architecture does not define a dedicated fault handling stack. Instead, the processor uses the stack that is active when the fault is generated (user stack, interrupt stack or supervisor stack) to handle a fault, with one exception: if the user stack is active when a fault is generated and the fault handling procedure is called with an implicit supervisor call, the processor switches to the supervisor stack to handle the fault.

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 stack that the fault handling procedure will use to handle the fault.

Fault Record Data

Figure 7.3 shows the structure of the fault record. In this record, the type number of the fault is stored in the fault type field and the subtype number (or bit positions for multiple subtypes) of the fault subtype is stored in the fault subtype field. The address-of-faulting-instruction field contains the IP of the instruction upon which the processor faulted.

Values in the PC and AC registers when a fault is generated are stored in their respective fault record fields. This information is used to resume work on the program after the fault is handled. In the case of parallel instruction execution, these fields contain the states of the registers when the processor has completed all parallel and out-of-order instruction execution.

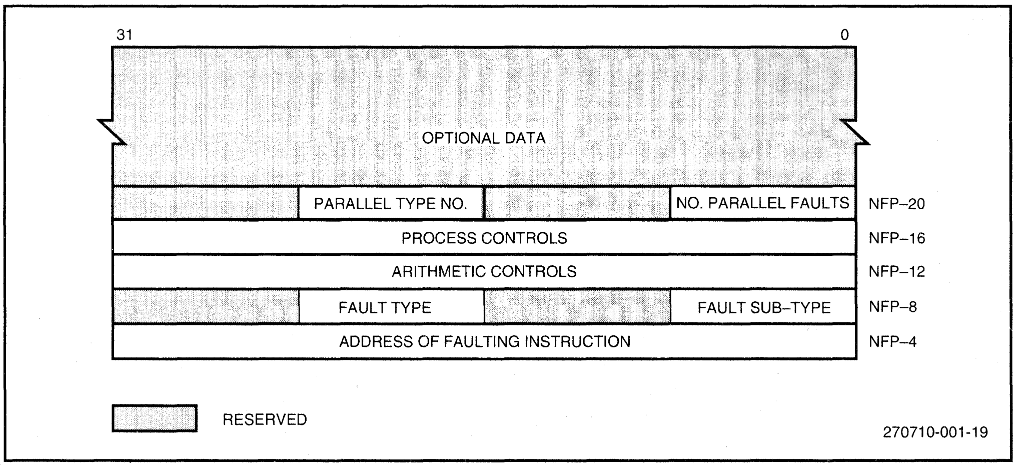


Figure 7.3. Fault Record

Optional data fields are defined for certain faults. These fields contain additional information about the faulting conditions, usually to assist resumption. Parallel fault and operation-unaligned fault types are the only faults in the i960 CA processor that use optional data fields. The processor can generate parallel faults when instructions are executed in parallel. Parallel faults and the contents of the optional data fields for this fault type are described later in the section titled *Multiple Fault Conditions*. The operation-unaligned fault and its optional data field are described later in the section titled *Operation Faults*. All unused bytes in the fault record are reserved.

Return Instruction Pointer

When a fault handling procedure is called — as with any call — a return instruction pointer is saved in the RIP register (r2). The RIP is intended to point 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. The *Fault Reference* section, later in this chapter, defines the RIP content for each fault.

When the RIP refers to a "next instruction", this does not always mean the instruction directly after the faulting instruction. Instead, it is an instruction to which the processor can logically return to resume program execution.

Fault Record Location

The fault record is stored in the stack that the processor uses to execute the fault handling procedure. As shown in Figure 7.4, 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 which is computed by adding the memory size allocated for padding and the fault record to the new stack pointer (NSP).

The processor automatically determines the number of bytes required for the fault record and increments the FP by that amount, rounding it off to the next highest 16-byte boundary. Fault record size is variable, based on the size of the optional fault-data portion of the fault record.

Stack frame alignment is defined for each implementation of the i960 architecture. This alignment boundary is calculated from the relationship SALIGN*16. For example, if SALIGN is selected to be 4, stack frames are aligned on 64-byte boundaries. In the i960 CA processor, SALIGN=1.

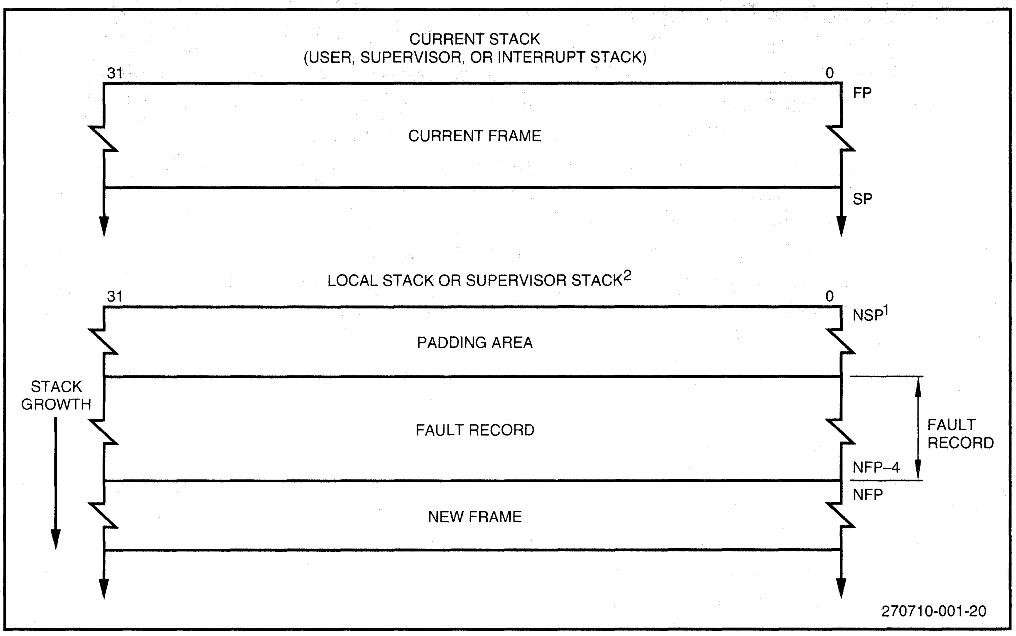


Figure 7.4. Storage of the Fault Record on the Stack

NOTES

1. If the call to the fault handler procedure does not require a stack switch, the new stack pointer (NSP) is the same as SP.
2. If the processor is in user mode and the fault handler procedure is called with a system-supervisor call, the processor switches to the supervisor stack.

MULTIPLE AND PARALLEL FAULTS

Multiple fault conditions can occur in two circumstances: (1) during a single instruction execution; (2) during multiple instruction execution when the instructions are executed by parallel execution units within the processor. The following sections describe how faults are handled under these conditions.

Multiple Faults

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 may not report all detected faults.

In a multiple fault situation, the reported fault condition is left to the implementation. The architecture, however, does define the criteria for determining which fault to report when trace fault conditions are one or more of the fault conditions.

Multiple Trace Fault Conditions Only

Multiple trace fault conditions that single instruction executions generate are reported in a single trace fault. To support this multiple fault reporting, the trace fault uses bit positions in the fault-subtype field to indicate occurrences of multiple faults of the same type (Table 7.1).

For example, when instruction tracing is enabled, an instruction trace fault condition is detected on each instruction that is executed, along with other trace fault conditions that are enabled (e.g., a call trace fault or a branch trace fault.) The processor generates a trace fault after each instruction and sets the appropriate bit or bits in the fault-subtype field to indicate the instruction trace fault and any other trace fault subtypes that occurred. See *Chapter 8, Tracing and Debugging* for a description of the trace fault.

Multiple Trace Fault Conditions with Other Fault Conditions

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 processor generates at least two faults: a non-trace fault and a trace fault.

The non-trace fault is handled first and the trace fault is triggered immediately after executing the return instruction **(ret)** at the end of the non-trace fault handler.

Parallel Faults

As described in *Appendix A, Optimizing Code for the i960 CA Microprocessor*, the i960 CA processor exploits the architecture's tolerance of parallel and out-of-order instruction execution by issuing instructions to multiple, independent execution units on the chip. The following sections describe how the processor handles faults in this environment.

Faults in One Parallel Instruction

When a fault occurs during the execution of a particular instruction, it is not possible to suspend other instructions that are already executing in other execution units. To handle the fault, the processor continues executing new instructions until each execution unit completes execution of its respective instruction and all out-of-order instructions are executed. For example, if an integer overflow occurs during the addition in the following code example, the fault is detected before the multiply has completed execution. Before invoking the integer-overflow fault handling procedure, the processor waits for the multiply to complete.

```
        muli  g2, g4, g6;
        addi  g8, g9, g10;        # results in integer overflow
```

Faults in Multiple Parallel Instructions

When executing instructions in parallel, it is possible for faults to occur in more than one currently executing instruction. In the code sequence above, for example, an integer overflow fault could occur for both the **muli** and **addi** instructions, with the fault from the **addi** instruction being recognized by the processor first. To report multiple parallel faults, the architecture provides the parallel fault type.

In these parallel fault situations, the processor saves the fault type and subtype in the optional data field for each fault detected after the first fault. 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.

The existence of multiple parallel faults is often catastrophic. Multiple parallel faults are generated as imprecise faults, which means that recovery from the faults is normally not possible. (Imprecise faults are described later in this chapter's section titled *Precise and Imprecise Faults.*) Unless imprecise faults are disallowed, a parallel-fault-handling procedure generally does not attempt to recover from the faults, but instead calls a debug monitor to analyze the faults. If recovery from every parallel fault is possible, the RIP allows the processor to resume executing the program when the fault handling has completed.

Even though multiple faults can be generated by multiple instructions executing in parallel, only one fault is ordinarily generated per instruction, as described in the previous section titled *Multiple Faults.*

Fault Record for Parallel Faults

Figure 7.5 shows the structure of the fault record for parallel faults.

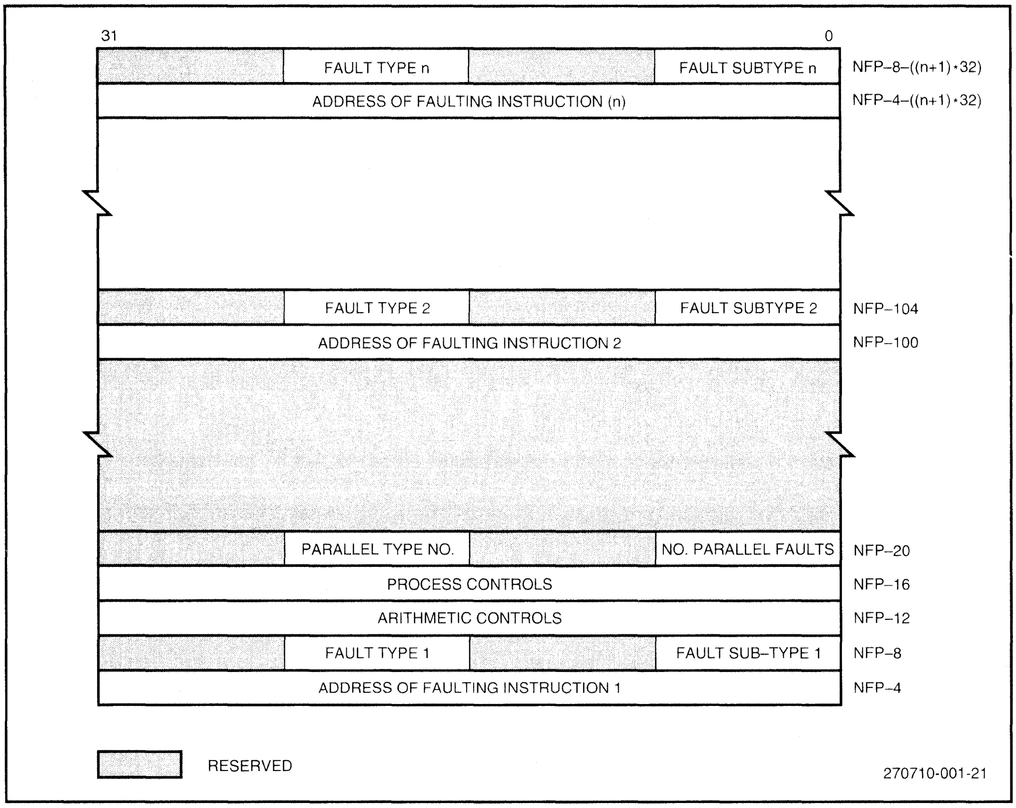


Figure 7.5. Fault Record for Parallel Faults

To calculate byte offsets, "n" indicates fault number. Thus, for the second fault recorded (n=2), the relationship (NFP-4 -(n+1)*32) reduces to NFP-100. For the i960 CA device, number of parallel faults allowed is 2 or 3.

When multiple 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 the section titled *Fault Record.* Information for the remaining parallel faults is then written to the fault record's optional data field and the fault handling procedure for parallel faults is invoked.

The first word in the fault record's optional data field (NFP-20) contains information about the parallel faults. The byte at offset NFP-18 contains 00H (encoding for the parallel fault type); the byte at NFP-20 contains the number of parallel faults. The optional data field also contains a 32-byte parallel fault record for each additional fault. These parallel fault records are stored incrementally in the fault record starting at byte offset NFP-97. The fault record for each additional fault contains only the fault type, fault subtype and address-of-faulting instruction field. (AC and PC register values are not given for these faults because they are already given in the fault record for the first fault.)

FAULT HANDLING PROCEDURES

The fault handling procedures can be located anywhere in the address space. Each procedure must begin on a word boundary.

The processor can execute the procedure in the user mode or the supervisor mode, depending on the type of fault table entry.

To resume work on a program at the point where a fault occurred (following the recovery action of the fault handling procedure), the fault handling procedure must be executed in supervisor mode. The reason for this requirement is described in a following section titled *Returning to the Point in the Program Where the Fault Occurred.*

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 interrupt pending 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.
- Explicitly write the processor state and fault record into memory and perform processor or system shutdown.
- Perform processor or system shutdown without explicitly saving the processor state or fault information.

When working with the processor at the development level, a common fault handling procedure action is to save the fault and processor state information and make a call to a debugging device such as a debugging monitor. This device can then be used to analyze the fault information.

Program Resumption Following a Fault

Because of the i960 CA processor's multi-stage execution pipeline, faults can occur:

- before execution of the faulting instruction (i.e., the instruction that causes the fault)
- during instruction execution
- immediately following execution

When the fault occurs before the faulting instruction is executed, the faulting instruction may be re-executed upon return from the fault handling procedure.

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.

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 Operation Subtypes
- All Constraint Subtypes
- Length
- Arithmetic Zero Divide
- All Trace Subtypes

Resumption of the program may or may not be possible with the following fault subtype:

- Integer Overflow

The effect that specific fault types have on a program is given in the fault reference section at the end of this chapter under the heading *Program State Changes.*

Returning to the Point in the Program Where the Fault Occurred

As described above, most faults can be handled such that program control flow is not affected. In this case, the processor allows work on 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.

To use this mechanism, the fault handling procedure must be invoked using a supervisor call. This method is required because — to resume work on the program at the point where the fault occurred — the saved process controls in the fault record must be copied back into the PC register upon return from the fault handling procedure. The processor only performs this action if the processor is in supervisor mode when the return is executed.

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 predictably perform a return from a fault handling procedure to an alternate point in the program, the fault handling procedure should perform the following four 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 process controls field of the fault record before the return,
4. Execute a return with the **ret** instruction.

This technique should be used carefully and only in situations where the fault handling procedure is closely coupled with the application program. Also, a return of this type can only be performed if the processor is in supervisor mode prior to the return.

FAULT CONDITIONS AND FAULT CONTROL

The processor generates faults implicitly when fault conditions occur and explicitly at the request of software. For several fault conditions, the programmer may control whether or not a fault is actually signaled when the condition is recognized. The following sections describe conditions which cause faults and facilities for controlling faults which are optionally generated.

Implicit Fault Generation

Most faults are generated implicitly; that is, they occur as a side effect of an instruction execution which has encountered difficulty. Following paragraphs summarize conditions which cause faults. The *Fault Reference* section at the end of this chapter provides a detailed description of each fault type and subtype.

Destination Overflow – An integer overflow fault is signaled when the result of an integer operation does not fit in the specified destination. The integer overflow fault handling procedure is invoked if the AC register integer overflow mask bit is set to enable these faults.

addi	**subi**
stib	**shli**
muli	**divi**

Division by Zero – The zero-divide fault is generated when the divisor of an integer or ordinal division is zero.

divo	**divi**
ediv	**remo**
remi	

Supervisor Protection Violations – The constraint-privileged fault is generated if the application attempts to execute a supervisor-only instruction while not in supervisor mode.

sdma	**sysctl**

The type-mismatch fault is generated if the application attempts to modify a supervisor-only resource while not in supervisor mode. On the i960 CA processor, supervisor-only resources are the PC register, on-chip data RAM and special function registers. The following actions generate a type-mismatch fault if attempted when the processor is not in supervisor mode:

- Using **modpc** to modify the PC register. (Using **modpc** to read the register is allowed in non-supervisor mode and does not cause a fault.)
- Writing to the protected on-chip data RAM.
- Reading or writing a SFR.

Out-of-bounds System-Procedure Call – The protection-length fault is generated if the processor attempts to execute a **calls** with a system procedure number specified which is greater than the size of the system procedure table.

Invalid Instruction Encodings – The operation-invalid-opcode fault is generated if the processor encounters an invalid opcode or an invalid encoding of a MEM-format instruction addressing mode.

Unaligned Register Reference – The invalid-operand fault is generated if the processor detects any unaligned register reference in any instruction which references long, triple or quad groups of registers.

Unaligned Memory Access – The operation-unaligned fault is signaled if the processor attempts to issue a memory request to an unaligned location. The unaligned-fault mask bit located in the fault-control word (PRCB) determines whether the fault handling procedure is invoked or whether access is handled transparently by the processor, without a fault. The fault-control word and PRCB are described in *Chapter 14, Initialization and System Requirements.*

Referencing a Non-existent SFR - The invalid-operand fault is generated if the processor executes an instruction which references a non-existent special function register. On the i960 CA processor, only sf0, sf1 and sf2 are implemented.

Issuing a Bad System Control Command - The operation-invalid-operand fault is generated if the processor executes an instruction which specifies a non-existent **sysctl** command.

Execution from Internal Data RAM - The operation-unimplemented fault is generated if an attempt is made to execute an instruction fetched from the i960 CA processor's on-chip data RAM.

Instruction Type is being Traced - A trace-fault is generated when the processor executes an instruction selected for tracing in the TC register and tracing is enabled by the PC register trace enable bit. See *Chapter 8, Tracing and Debugging* for a complete description.

Breakpoint Detected - A trace fault is generated when:

- The processor executes an instruction at an instruction pointer which matches one of the programmed instruction-address breakpoints and trace faults are enabled.
- The processor issues a memory request that matches one of the programmed data-address breakpoints and trace faults are enabled.

See *Chapter 8, Tracing and Debugging* for a complete discussion of the breakpoint registers.

Explicit Fault Generation

Two sets of instructions allow explicit fault generation anywhere in a program. The fault-if instructions (**faulte**, **faultne**, **faultl**, **faultle**, **faultg**, **faultge**, **faulto**, **faultno**) allow a fault to be generated conditionally. When one of these instructions is executed, the processor checks the AC register condition code bits then generates a constraint-range fault if the condition specified with the instruction is met.

mark and **fmark** (force mark) instructions allow a breakpoint-trace fault to be generated anywhere in the instruction stream.

Fault Controls

Certain fault types and subtypes have mask bits or flags associated with them that determine whether or not a fault is generated when a fault condition occurs. Table 7.2 summarizes these flags and masks, data structures in which they are located, fault subtypes they affect and where more information about them may be found.

The integer overflow mask bit inhibits an integer overflow faults from being generated. The use of this mask is discussed in the *Fault Reference* section at the end of this chapter.

The no-imprecise-faults (NIF) bit controls the synchronizing of faults for a category of faults called imprecise faults. The function of this bit is described later in this chapter's section titled *Precise and Imprecise Faults.*

TC register trace mode bits and PC register trace enable bit support trace faults. Trace mode bits enable trace modes; trace enable bit enables trace fault generation. The use of these bits is described in the *Fault Reference* section on trace faults at the end of this chapter. Further discussion of these flags is provided in *Chapter 8, Tracing and Debugging*.

Table 7.2. Fault Flags or 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 Flags	Trace Controls (TC) Register	All Trace Faults
Unaligned Fault Mask	Process Control Block (PRCB)	Unaligned fault

NOTE

The unaligned fault, unaligned fault mask and the processor control block are i960 CA processor extensions to the i960 architecture.

The unaligned fault mask bit is located in the process control block (PRCB), which is read during initialization. It controls whether unaligned memory accesses are handled by the processor or generate a fault. (See *Chapter 10, The Bus Controller.*)

FAULT HANDLING ACTION

Once a fault occurs, the processor saves the program state; calls the fault handling procedure; and restores the program state (if possible) once the fault recovery action is completed. No software other than the fault handling procedures is required to support this activity.

Three different types of implicit procedure calls can be used to invoke the fault handling procedure according to the information in the selected fault table entry: a local call, a system-local call and a system-supervisor call.

The following sections 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.

Local Fault Call

When the selected fault handler entry in the fault table is an entry type 00_2 (local procedure), the processor performs the same operation as is described in the section of *Chapter 5, Procedure Calls* titled *Call Operation*, 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 7.4.)
- 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 the section earlier in this chapter titled *Program Resumption Following a Fault.*

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 pending when the fault occurred. Upon return, the processor performs the action described in the section of *Chapter 5, Procedure Calls* titled *Return Operation*, except that the arithmetic controls field from the fault record is copied into the AC register. Since the call made is local, the process controls field from the fault record is not copied back to the PC register.

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.

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 the section of *Chapter 5, Procedure Calls* titled *Call Operation*, with the following exceptions:

- If in user mode when the fault occurs: 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 in supervisor mode when the fault occurs: 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 7.4.)
- 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 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 fault, the trace enable bit is cleared.

On a return from the fault handling procedure, the processor performs the action described in the section of *Chapter 5, Procedure Calls* titled *Return Operation*, with the following exceptions:

- 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. (Restoring the PC register restores the trace-fault-pending flag and trace enable bit values to their pre-fault values.) Also, if the processor was in user mode when the fault occurred, the mode is set back to user mode; otherwise, the processor remains in supervisor mode.
- 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 interrupts are pending that are higher than the priority of the program being returned to, they are handled as if the interrupt had occurred at this point. If the trace-fault-pending flag and trace enable bit are set, the trace fault is also handled at this time.

PC register restoration causes any changes to the process controls caused by the fault handling procedure to be lost. In particular, if the **ret** instruction from the fault handling procedure caused the PC register trace-fault-pending flag to be set, this setting would be lost upon return.

Faults and Interrupts

If an interrupt occurs during 1) an instruction that will fault or 2) an instruction that has already faulted or 3) during fault handling procedure selection, the processor handles the interrupt in the following way: It completes the selection of the fault handling procedure, then services the interrupt just prior to executing the first instruction of the fault handling procedure. The fault is handled upon return from the interrupt. Handling the interrupt before the fault reduces interrupt latency.

PRECISE AND IMPRECISE FAULTS

As described earlier in this chapter in the section titled *Parallel Faults*, the i960 architecture — to support parallel and out-of-order instruction execution — allows some faults to be generated together and not in sequence. When this situation occurs, it may be impossible to recover from some faults, because the state of the instructions surrounding the faulting instruction has changed or the RIP is unpredictable.

The processor provides two mechanisms for controlling the circumstances under which faults are generated: the AC register no-imprecise-faults bit (NIF bit) and the synchronize-faults instruction (**syncf**). The following paragraphs describe how these mechanisms can be used.

Faults are grouped into the following categories: precise, imprecise and asynchronous. *Precise faults* are those intended to be software recoverable. For any instruction that can generate a precise fault, the processor:

1. does not execute the instruction if an unfinished prior instruction will fault and
2. does not execute subsequent out-of-order instructions that will fault.

Also, the RIP points to an instruction where the processor can resume program execution without breaking program control flow. Two faults are always precise: trace faults and protection faults.

Imprecise faults are those where the architecture does not guarantee that sufficient information is saved in the fault record to allow recovery from the fault. For imprecise faults, the faulting instruction address is correct, but the state of execution of instructions surrounding the faulting instruction may be unpredictable. Also, the architecture allows imprecise faults to be generated out of order, which means that the RIP may not be of any value for recovery. Faults that the architecture allows to be imprecise include:

- operation
- constraint
- arithmetic
- type

Refer to the *Fault Reference* section of this chapter to determine whether specific faults are precise.

Asynchronous faults are those whose occurrence has no direct relationship to the instruction pointer. The i960 architecture does not define any faults in this category and the i960 CA processor generates no such faults.

The NIF bit controls imprecise fault generation. When this bit is set, all faults generated are precise. This means the following conditions hold true:

1. All faults are generated in order.
2. A precise fault record is provided for each fault: the faulting instruction address is correct and the RIP provides a valid reentry point into the program.

When the NIF bit is clear, imprecise faults are allowed to be generated: in parallel, out of order and with an imprecise RIP. Here, the following conditions hold true:

1. When an imprecise fault occurs, the faulting instruction address in the fault record is valid, but the saved IP is unpredictable.
2. If instructions are executed out of order and parallel faults occur, recovery from some faults may not be possible because the faulting instruction's source operands may be modified when subsequent instructions are executed out of order.

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:

1. force faults to be precise when the NIF bit is clear.
2. ensure that all instructions are complete and all faults are generated in one block of code before the execution of another block of code begins.

Compiled code should execute with the NIF bit clear, using **syncf** where necessary to ensure that faults occur in order. In this mode, imprecise faults are considered as catastrophic errors from which recovery is not needed.

The NIF bit should be set if recovery from one or more imprecise faults is required. For example, the 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.

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 and the notation used.

Fault Type and Subtype	Gives the number which appears in the fault record fault-type field when the fault is generated. The fault-subtype section lists fault subtypes and number associated with each fault subtype.
Function	Describes the purpose of fault type and fault subtype. It also describes how the processor handles each fault subtype.
RIP	Describes the value saved in the RIP register of the stack frame that the processor was using when the fault occurred.
Program State Changes	Describes fault subtype effects on a program's control flow.

Arithmetic Faults

Fault Type: 3H

Fault Subtype:

Number	Name
0H	Reserved
1H	Integer Overflow
2H	Arithmetic Zero Divide
3H-FH	Reserved

Function: Indicates problem with operand an arithmetic instruction result. Integer overflow fault is generated when a result of integer instruction overflows destination and AC register integer overflow mask is cleared. Here, the result's *n* least significant bits are stored in the destination, where *n* is destination size. Instructions that generate this fault are:

addi **subi**
stib **shli**
muli **divi**

Arithmetic zero divide fault is generated when divisor operand of ordinal or integer divide instruction is zero. Instructions that generate this fault are:

divo **divi**
ediv **remi**
remo

RIP: IP for next-executed instruction if fault had not occurred.

Program State Changes: Faults may be imprecise when executing with NIF bit cleared. Integer overflow fault may not be recoverable because result is stored in destination before fault is generated; e.g., faulting instruction cannot be re-executed if destination register was also a source register for the instruction. Arithmetic zero divide fault is generated before execution of the faulting instruction.

Constraint Faults

Fault Type: 5H

Fault Subtype:

Number	Name
0H	Reserved
1H	Constraint Range
2H	Privileged
3H-FH	Reserved

Function: Indicates program or procedure violated an architectural constraint.

Constraint-range fault is generated when a fault-if instruction is executed and AC register condition code field matches the condition required by the instruction.

Privileged fault is also generated when program or procedure attempts to use a privileged (supervisor-mode only) instruction while processor is in user mode. Privileged instructions for the i960 CA processor are:

sdma sysctl

RIP: No defined value.

Program State Changes: These faults may be imprecise when executing with NIF bit cleared. No changes in program's control flow accompany these faults. Constraint-range fault is generated after fault-if instruction executes; program state is not affected. Privileged fault is generated before faulting instruction executes.

Operation Faults

Fault Type: 2H

Fault Subtype:

Number	**Name**
0H	Reserved
1H	Invalid Opcode
2H	Unimplemented
3H	Unaligned
4H	Invalid Operand
5H - FH	Reserved

Function: Indicates processor cannot execute current instruction because of invalid instruction syntax or operand semantics. Invalid-opcode fault is generated when processor attempts to execute instruction containing undefined opcode or addressing mode.

Unimplemented fault is generated when processor attempts to execute an instruction fetched from on-chip data RAM.

Unaligned fault is generated when the following conditions are present: (1) processor attempts to access an unaligned word or group of words in memory and (2) fault is enabled by the unaligned-fault mask bit in the PRCB fault configuration word.

The i960 CA processor handles unaligned accesses to little endian regions of memory in microcode and carries out the access regardless of unaligned-fault mask bit setting. Processor does not support unaligned accesses to big endian regions; such attempts result in incoherent data in memory. Enabling the unaligned fault when using big endian byte ordering provides a means of detecting unsupported unaligned accesses.

When an unaligned fault is signaled, the effective address of the unaligned access is placed in the fault record optional data section, beginning at address NFP-24. This address is useful to debug a program that is making unintentional unaligned accesses.

Invalid-operand fault is generated when processor attempts to execute an instruction for which one or more operands have special requirements which are not satisfied. Fault is caused by specifying non-existent SFR or non-defined **sysctl** and/or references to an unaligned long-, triple- or quad-register group.

RIP: No defined value.

Program State Changes: Faults may be imprecise when executing with the NIF bit cleared. A change in the program's control flow does not accompany operation faults; faults occur before instruction execution.

Parallel Faults

Fault Type: See the section titled *Parallel Faults* in this chapter.

Fault Subtype:

Function: Indicates that one or more faults occurred when processor was executing instructions in parallel by different execution units. This fault type can occur only when AC register NIF bit is cleared.

If parallel faults occur, the *Number of parallel faults* field in the fault record is a non-zero value, indicating the number of parallel faults recorded. This field is located in the fault record at location NFP-20.

A fault record is saved for each parallel fault detected. Information contained in these records is the same as is described in this section for specific fault types.

RIP: IP of instruction that would execute next if faults were not generated.

Program State Changes: Precision of faults recorded in a parallel fault record depends on fault types detected. A change in program's control flow may or may not accompany parallel faults, depending on fault types detected.

Protection Faults

Fault Type:	7H

Fault Subtype:

Number	**Name**
0H-1H	Reserved
2H	Length
3H-FH	Reserved

Function: Indicates program or procedure attempting to perform illegal operation that the architecture protects against.

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

RIP: Same as the address-of-faulting-instruction field.

Program State Changes: This fault type is always precise, regardless of NIF bit value. Change in program's control flow does not accompany length fault; fault is generated before faulting instruction.

Trace Faults

Fault Type: 1H

Fault Subtype:

Number	**Name**
Bit 0	Reserved
Bit 1	Instruction Trace
Bit 2	Branch Trace
Bit 3	Call Trace
Bit 4	Return Trace
Bit 5	Prereturn Trace
Bit 6	Supervisor Trace
Bit 7	Breakpoint Trace

Function:

Indicates processor detected one or more trace events. Event tracing mechanism is described in *Chapter 8, Tracing and Debugging.*

A trace event is the occurrence of a particular instruction or instruction type in the instruction stream. Processor recognizes seven different trace events: instruction, branch, call, return, prereturn, supervisor, breakpoint. It detects these events only if TC register mode bit is set for the event. If PC register trace enable bit is also set, processor generates a fault when trace event is detected.

A trace fault is generated following the instruction that causes a trace event (or prior to the instruction for the prereturn trace event). The following trace modes are available:

Instruction	Generates trace event following every instruction.
Branch	Generates trace event following any branch instruction when branch is taken (branch trace event does not occur on branch-and-link or call instructions).
Call	Generates trace event following any call or branch-and-link instruction or any implicit procedure call (i.e., fault- or interrupt-call).
Return	Generates trace event following any **ret** instruction.
Prereturn	Generates trace event prior to any **ret** instruction, providing PFP register prereturn trace flag is set (processor sets flag automatically when prereturn tracing is enabled).

Supervisor	Generates 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.
Breakpoint	Generates a trace event following any processor action that causes a breakpoint condition (such as a **mark** or **fmark** instruction or a match of the instruction-address breakpoint register or the data-address breakpoint register).

Trace fault subtype and fault subtype field bits are associated with each mode. Multiple fault subtypes can occur simultaneously; fault subtype bit is set for each subtype that occurs.

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

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

Address of the faulting instruction field in the fault record contains the IP for the instruction that causes the trace event, except for the prereturn trace fault; this field has no defined value.

RIP: IP for the instruction that would have been executed next if the fault had not occurred.

Program State Changes: This fault type is always precise, regardless NIF bit value. A change in the program's control flow accompanies all trace faults (except prereturn trace fault), because events that can cause a trace fault occur after the faulting instruction is completed. As a result, the faulting instruction cannot be re-executed upon returning from the fault handling procedure.

Since the prereturn trace fault is generated before the **ret** instruction is executed, a change in the program's control flow does not accompany this fault and the faulting instruction can be executed upon returning from the fault handling procedure.

Type Faults

Fault Type:	AH	
Fault Subtype:	**Number**	**Name**
	0H	Reserved
	1H	Type 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. Type-mismatch fault is generated when attempts are made to:

- Modify the PC register with **modpc** while processor is in user mode.
- Write to on-chip data RAM while processor is in user mode.
- Access a special function register while processor is in user mode.

RIP: No defined value.

Program State Changes: These faults may be imprecise when executing with the NIF bit cleared. A change in program's control flow does not accompany the type-mismatch fault because the fault occurs before execution of the faulting instruction.

Tracing and Debugging 8

CHAPTER 8
TRACING AND DEBUGGING

This chapter describes the i960 CA 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 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 processor also provides four hardware breakpoint registers that generate trace events and trace faults. Two registers are dedicated to trapping on instruction execution addresses, while the remaining two registers can trap on the addresses of various types of data accesses.

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. These controls are described in the following sections.

- TC register mode bits
- PC register trace enable bit
- PC register trace fault pending flag
- PFP register return status field prereturn trace flag (bit 0)
- System procedure table supervisor-stack-pointer field trace control bit
- BPCON register breakpoint mode bits and enable bits (in the control table)
- IPB0-IPB1 registers address field (in the control table)
- DAB0-DAB1 registers address field and enable bit (in the control table)

Trace Controls (TC) Register

The TC register (Figure 8.1) allows software to define the conditions under which trace events are generated.

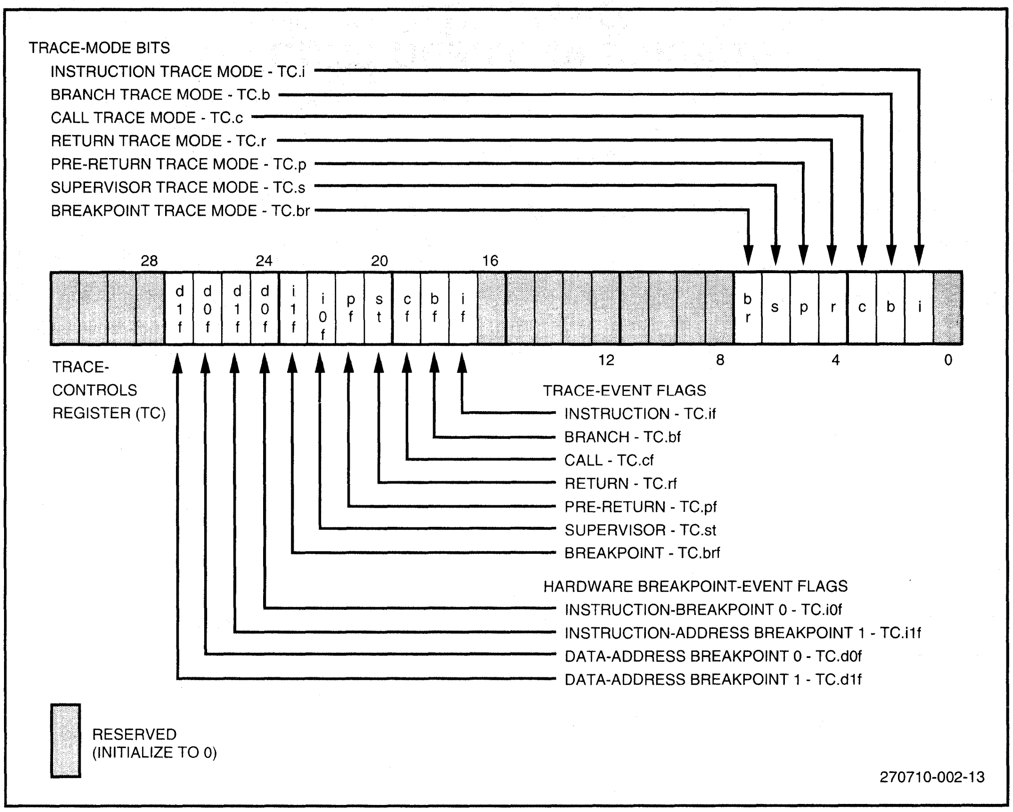

TRACE-MODE BITS
INSTRUCTION TRACE MODE - TC.i
BRANCH TRACE MODE - TC.b
CALL TRACE MODE - TC.c
RETURN TRACE MODE - TC.r
PRE-RETURN TRACE MODE - TC.p
SUPERVISOR TRACE MODE - TC.s
BREAKPOINT TRACE MODE - TC.br
28
24
20
16
d1f
d0f
d1f
d0f
i1f
i0f
pf
st
cf
bf
if
br
s
p
r
c
b
i
12
8
4
0
TRACE-CONTROLS REGISTER (TC)
TRACE-EVENT FLAGS
INSTRUCTION - TC.if
BRANCH - TC.bf
CALL - TC.cf
RETURN - TC.rf
PRE-RETURN - TC.pf
SUPERVISOR - TC.st
BREAKPOINT - TC.brf
HARDWARE BREAKPOINT-EVENT FLAGS
INSTRUCTION-BREAKPOINT 0 - TC.i0f
INSTRUCTION-ADDRESS BREAKPOINT 1 - TC.i1f
DATA-ADDRESS BREAKPOINT 0 - TC.d0f
DATA-ADDRESS BREAKPOINT 1 - TC.d1f
RESERVED (INITIALIZE TO 0)
270710-002-13

Figure 8.1. 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 whenever a call or branch-and-link operation executes. (Trace modes are described later in this chapter's section titled *Trace Modes.*) The processor uses event flags to keep track of which trace events have been generated.

A special instruction, the modify-trace-controls (**modtc**) instruction, allows software to modify the TC register. On initialization, all TC register bits and flags are cleared. **modtc** can then be used to set or clear trace mode bits as required. Software can also access event flags using **modtc**; however, this is generally not necessary. The processor automatically sets and clears these flags as part of its trace handling mechanism.

TC register bits 0, 8 through 16 and 28 through 31 are reserved. Software must initialize these bits to zero and not modify them afterwards.

Trace Enable Bit and Trace-Fault-Pending Flag

The PC register trace enable bit and the trace-fault-pending flag control tracing. 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 this chapter's section titled *Tracing and Interrupt Procedures.*

The trace-fault-pending flag allows the processor to track when a trace event is detected for an enabled trace condition. The processor uses this flag as follows:

1. When the processor detects a trace event and tracing is enabled, it sets the flag.
2. Before executing an instruction, the processor checks the flag.
3. If the flag is set and tracing is enabled, it signals a trace fault.

By providing a means to record trace event occurrences, the trace-fault-pending flag allows the processor to service an interrupt or handle a fault other than a trace fault before handling the trace fault. Software should not modify this flag.

Trace Control on Supervisor Calls

The trace control bit allows tracing to be enabled or disabled when a call-system instruction (**calls**) executes which results in a switch to supervisor mode. This action occurs independent of whether or not tracing is enabled prior to the call. A supervisor call is a **calls** instruction that references an entry in the system procedure table with an entry type 10_2. When a supervisor call executes, the processor:

1. Saves current PC register trace enable bit status in the PFP register return-type field bit 0.
2. Sets the PC register trace enable bit to the value of the trace control bit. The processor gets the trace control bit from bit 0 of the supervisor stack pointer, which is cached during the reset initialization sequence.

When the trace control bit is set, tracing is enabled on supervisor calls; when cleared, tracing is disabled on supervisor calls. Upon return from the supervisor procedure, the PC register trace enable bit is restored to the value saved in the PFP register return-type field.

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. The section later in this chapter titled *Handling Multiple Trace Events* describes processor function when multiple trace events occur.

- Instruction trace
- Branch trace
- Breakpoint trace
- Prereturn trace
- Call trace
- Return trace
- Supervisor trace

Instruction Trace

When the instruction-trace mode is enabled, the processor generates an instruction-trace event each time an instruction executes. A debugging monitor can use this mode to single-step the processor.

Branch Trace

When the branch-trace mode is enabled, the processor generates a branch-trace event any time a branch instruction executes and the branch is taken. A branch-trace event is not generated for conditional-branch instructions that do not branch or for branch-and-link, call or return instructions.

Call Trace

When the call-trace mode is enabled, the processor generates a call-trace event any time a call instruction (**call**, **callx** or **calls**) or a branch-and-link instruction (**bal** or **balx**) executes. An implicit call — such as the action used to invoke a fault handling or an interrupt handling procedure — also causes a call-trace event to be generated.

When the processor detects a call-trace event, it also 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 when to signal a prereturn-trace event on a **ret** instruction.

Return Trace

When the return-trace mode is enabled, the processor generates a return-trace event any time a **ret** instruction executes.

Prereturn Trace

The prereturn-trace mode causes the processor to generate a prereturn-trace event prior to **ret** execution, providing the PFP register prereturn-trace flag is set. (Prereturn tracing cannot be used without enabling call tracing.) The processor sets the prereturn-trace flag whenever it detects a call-trace event as described above for call-trace mode. This flag performs a prereturn-trace-pending function.

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.

Supervisor Trace

When supervisor-trace mode is enabled, the processor generates a supervisor-trace event when:

1. a call-system instruction (**calls**) executes, where the procedure table entry is for a system-supervisor call

   — or —

2. a **ret** instruction executes and the return-type field is set to 010_2 or 011_2 (i.e., return from supervisor mode).

When these procedures are called with supervisor calls, this trace mode allows a debugging program to determine kernel-procedure call boundaries within the instruction stream.

Breakpoint Trace

Breakpoint trace mode allows trace events to be generated at places other than those specified with the other trace modes. This mode is used in conjunction with **mark** and **fmark**.

Software Breakpoints

mark and **fmark** allow breakpoint trace events to be generated at specific points in the instruction stream. When breakpoint trace mode is enabled, the processor generates a breakpoint trace event any time it encounters a **mark**. **fmark** causes the processor to generate a breakpoint trace event regardless of whether or not breakpoint trace mode is enabled.

Hardware Breakpoints

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

Breakpoint trace events can be generated when the processor executes an instruction with an IP that matches one of the addresses programmed into the two instruction breakpoint registers (IPB0 - IPB1). Each instruction address breakpoint may be enabled or disabled individually by programming the two least significant bits in IPB0 or IPB1. Figure 8.2 describes the instruction address breakpoint registers.

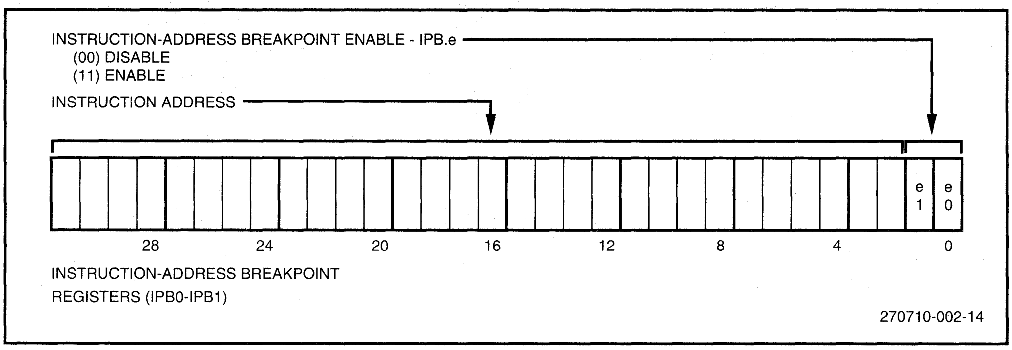


Figure 8.2. Instruction Address Breakpoint Registers (IPB0 - IPB1)

Breakpoint trace events may also be generated when a memory access is issued which matches conditions programmed in one of two data address breakpoint registers (DAB0 - DAB1, Figure 8.3). Each breakpoint register is programmed to fault when the address of an access matches the breakpoint register and the access is one of four types: 1) any store, 2) any load or store, 3) any data load or store or any instruction fetch or 4) any memory access.

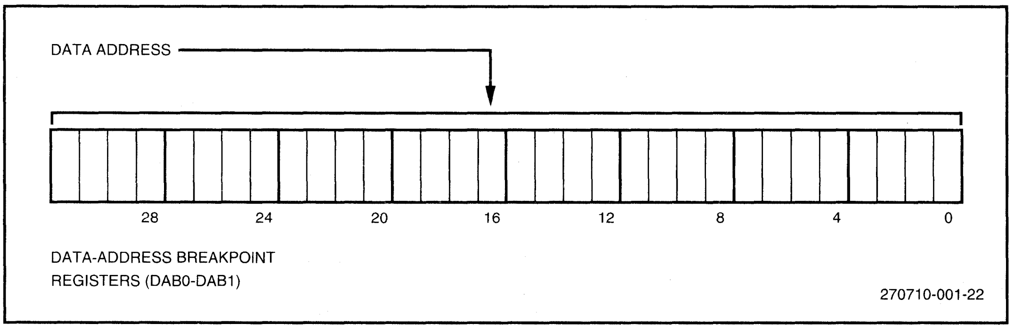


Figure 8.3. Data Address Breakpoint Registers (DAB0 - DAB1)

The programmer configures the BPCON register to set the data address breakpoint mode which corresponds to one of these access types (Figure 8.4). Each data address breakpoint may also be enabled or disabled individually by programming the BPCON enable bits.

The instruction-address breakpoint, data-address breakpoint and breakpoint control registers are on-chip control registers. These registers are loaded from the control table in memory at initialization or may be modified using **sysctl**. Control registers are described in *Chapter 2, Programming Environment.*

A breakpoint trace event is signalled when the processor attempts an access which is set for detection (instruction or data breakpoint). Breakpoint trace is enabled by setting the appropriate field in the IPB0, IPB1 and BPCON registers. If breakpoint trace is enabled, the appropriate

TC register hardware breakpoint trace event flags are set. If tracing is enabled, a trace fault is generated after the faulting instruction completes execution.

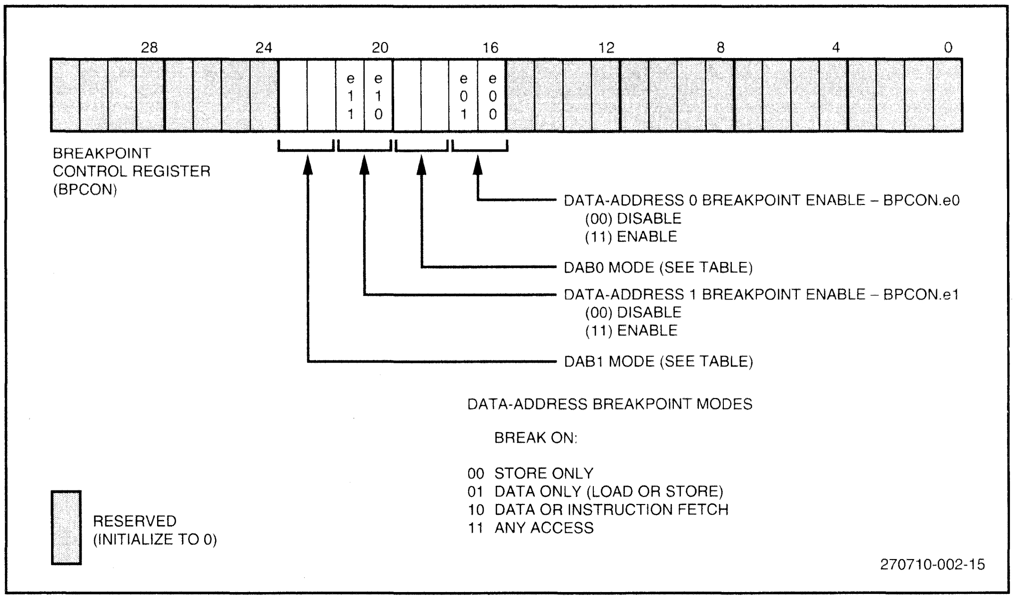

28
24
20
16
12
8
4
0
e11
e10
e01
e00
BREAKPOINT CONTROL REGISTER (BPCON)
DATA-ADDRESS 0 BREAKPOINT ENABLE – BPCON.e0
(00) DISABLE
(11) ENABLE
DAB0 MODE (SEE TABLE)
DATA-ADDRESS 1 BREAKPOINT ENABLE – BPCON.e1
(00) DISABLE
(11) ENABLE
DAB1 MODE (SEE TABLE)
DATA-ADDRESS BREAKPOINT MODES
BREAK ON:
00 STORE ONLY
01 DATA ONLY (LOAD OR STORE)
10 DATA OR INSTRUCTION FETCH
11 ANY ACCESS
RESERVED (INITIALIZE TO 0)
270710-002-15

Figure 8.4. Hardware Breakpoint Control Register (BPCON)

SIGNALING A TRACE EVENT

To summarize the information presented in the previous sections, the processor signals a trace event when it 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.
- An implicit call operation executed and the call-trace mode is enabled.
- A **mark** instruction executed and the breakpoint-trace mode is enabled.
- A **fmark** instruction executed.
- The processor is executing an instruction at an IP matching an enabled instruction address breakpoint register.
- The processor has issued a memory access matching the conditions of an enabled data address breakpoint register.

When the processor detects a trace event and the PC register trace enable bit is set, the processor performs the following action:

1. The processor sets the appropriate TC register trace event flag. If a trace event meets the conditions of more than one of the enabled trace modes, a trace event flag is set for each trace mode condition that is met.

2. The processor sets the PC register trace-fault-pending flag. The processor may set a trace event flag and trace-fault-pending flag before completing execution of the instruction that caused the event. However, the processor only handles trace events between instruction executions.

If — when the processor detects a trace event — the PC register trace enable bit is clear, the processor sets the appropriate event flags but does not set the PC register trace-fault-pending flag.

HANDLING MULTIPLE TRACE EVENTS

If the processor detects multiple trace events, it records one or more of them based on the following precedence, where 1 is the highest precedence:

1. Supervisor-trace event
2. Breakpoint- (from **mark** or **fmark** instruction or from a breakpoint register), branch-, call- or return-trace event
3. Instruction-trace event

When multiple trace events are detected, the processor may not signal each event; however, it at least signals the one with the highest precedence.

TRACE FAULT HANDLING PROCEDURE

The trace fault handling procedure (which the processor calls when it detects a trace event) is a type of fault handling procedure. General requirements for fault handling procedures are given in *Chapter 7, Faults.*

The trace fault handling procedure is involved in a specific way and is handled slightly different than other faults. A trace fault handler must be involved with an implicit system-supervisor call. When the call is made, the PC register trace enable bit in is cleared. This disables trace faults when the trace fault handler is executing. 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 exceptional 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.

TRACE HANDLING ACTION

Once a trace event is signaled, the processor determines how to handle the trace event, according to the PC register trace enable bit and trace fault pending flag settings and to other events that might occur simultaneously with the trace event, such as an interrupt or non-trace fault. Subsections that follow describe how the processor handles trace events for various situations.

Normal Handling of Trace Events

Prior to executing an instruction, the processor performs the following action regarding trace events:

1. The processor checks the state of the trace fault pending flag:
   a. If clear, the processor begins execution of the next instruction.
   b. If set, the processor performs the following actions.
2. The processor checks the PC register trace enable bit state:
   a. If clear, the processor clears any trace event flags that are set prior executing the next instruction.
   b. If set, the processor signals a trace fault and begins fault handling action as described in *Chapter 7, Faults*.

Prereturn Trace Handling

The processor handles a prereturn trace event the same as described above except when it occurs at the same time as a non-trace fault. In this case, the non-trace fault is handled first. On returning from the fault handler for the non-trace fault, the processor checks the PFP register prereturn trace flag. If set, the processor generates a prereturn trace event, then handles it as described above.

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, then clearing the PC register trace enable bit and trace fault pending flag.

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 and trace fault pending flag states. If these two flags were set prior to calling the interrupt procedure, a trace fault is signaled on return from the interrupt procedure.

NOTE

On a return from an interrupt handling procedure, the trace fault pending flag is restored. If this flag was set as a result of the interrupt procedure's **ret** instruction (i.e., indicating a return trace event), the detected trace event is lost. This is also true on a return from a fault handler, when the fault handler has been called with an implicit supervisor call.

Instruction Set Reference 9

CHAPTER 9
INSTRUCTION SET REFERENCE

This chapter provides detailed information about each instruction the processor uses. Instructions are listed alphabetically by assembly language mnemonic. Format and notation used in this chapter are defined in the following section titled *Notation*.

INTRODUCTION

Information in this chapter is oriented toward programmers who write assembly language code for the processor. The information provided for each instruction includes the following:

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

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

- *Chapter 4, Instruction Set Summary* - Summarizes the instruction set by group and describes the assembly language instruction format.
- *Appendix D, Instruction Set Reference* - Describes instruction set opword encodings. A quick-reference listing of instruction encodings is also provided to assist debug with a logic analyzer.
- *Instruction Set Quick Reference* - Contains a tabular quick reference of each instruction's operation and side-effects.

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.

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 top of 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** or **FAULT IF**).

The i960 CA component-specific extensions to the i960 microprocessor instruction set are indicated with a box around the instruction's alphabetic reference. The following i960 CA device's instructions are such extensions:

$\boxed{\textbf{eshro}}$ $\boxed{\textbf{sdma}}$

$\boxed{\textbf{sysctl}}$ $\boxed{\textbf{udma}}$

Instruction set extensions are generally not portable to other i960 family implementations.

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

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 for which the instruction is testing 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 *Appendix A, Optimizing Code for the i960 CA Microprocessor.*

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

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:

sub*	*src1,*	*src2,*	*dst*
	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) in the mnemonic indicates a variable: in the above example, **sub*** is either **subi** or **subo**.

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 ... sf2)
disp	Signed displacement of range -2^{22}] ... (2^{22}] - 1)
efa	Address defined with the full range of addressing modes
targ	A relative offset or displacement to the target of instruction. Usually specified as a label in assembly code.

NOTE

For future implementations, the i960 architecture will allow up to a total of 32 Special Function Registers (SFRs). However, sf0, sf1 and sf2 are the only SFRs implemented on the i960 CA processor.

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
disp	Displacement

Description

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

Action

The *Action* section gives an algorithm written in a 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. For example, **shli** requires seven lines of pseudo-code to completely describe its function. Although it might appear from the algorithm that the instruction should take multiple clocks to execute, the i960 CA processor executes the instruction in a single clock.

The following is an example of the action algorithm for the **alterbit** instruction:

```
if ((AC.cc1 = 0) = 0)
    dst ← src and not (2^(bitpos mod 32));
    else dst ← src or (2^(bitpos mod 32));
```

In these action statements, the term AC.cc refers to the AC register condition code field; AC.cc1 means bit 1 of this field. The symbol "^" indicates an exponent; for example: 2^(*bitpos* **mod** 32) is equivalent to $2^{(bitpos\ \mathbf{mod}\ 32)}$.

Table 9.1 defines each abbreviation used in the instruction reference pseudo-code. Table 9.2 explains the symbols used in the pseudo-code.

NOTE

Since special function registers (*sfr*) may change independent of instruction execution, the following distinctions are important when interpreting the algorithm of any instruction which references a *sfr*.

1. When a source operand is a *sfr* and referenced more than once in an algorithm, the operand's value at every reference is the same as the first reference. In other words, the instruction operates as if the *sfr* was actually read only once, at the beginning of the instruction.
2. When the same *sfr* is specified as the source for multiple operands of the same instruction, the instruction operates as if the source *sfr* was actually read only once, at the beginning of the instruction. When either source operand appears in the action algorithm, the single operand value is used.
3. When a *sfr* is specified as a destination and the algorithm indicates more than one modification of the destination, the instruction operates as if the *sfr* were written only once, at the end of the instruction.

Table 9.1. Abbreviations in Pseudo-code

AC.xxx	Arithmetic Controls Register fields	
	AC.cc	Condition Code flags (AC.cc2:0)
	AC.cc0	Condition Code Bit 0
	AC.cc1	Condition Code Bit 1
	AC.cc2	Condition Code Bit 2
	AC.nif	No Imprecise Faults flag
	AC.of	Integer Overflow flag
	AC.om	Integer Overflow Mask Bit
PC.xxx	Process Controls Register fields	
	PC.em	Execution Mode flag
	PC.s	State Flag
	PC.tfp	Trace Fault Pending flag
	PC.p	Priority Field (PC.p5:0)
	PC.te	Trace Enable Bit
TC.xxx	Trace Controls Register fields	
	TC.i	Instruction Trace Mode Bit
	TC.c	Call Trace Mode Bit
	TC.p	Pre-return Trace Mode Bit
	TC.br	Breakpoint Trace Mode Bit
	TC.b	Branch Trace Mode Bit
	TC.r	Return Trace Mode Bit
	TC.s	Supervisor Trace Mode Bit
	TC.if	Instruction Trace Event flag
	TC.cf	Call Trace Event flag
	TC.pf	Pre-return Trace Event flag
	TC.brf	Breakpoint Trace Event flag
	TC.bf	Branch Trace Event flag
	TC.rf	Return Trace Event flag
	TC.sf	Supervisor Trace Event flag
PFP.xxx	Previous Frame Pointer (r0)	
	PFP.add	Address (PFP.add31:4)
	PFP.rt	Return Type Field (PFP.rt2:0)
	PFP.p	Pre-return Trace flag
sp	Stack Pointer (r1)	
fp	Frame Pointer (g15)	
rip	Return Instruction Pointer (r2)	
SPT	System Procedure Table	
	SPT.base	Supervisor Stack Base Address
	SPT(targ)	Address of SPT Entry targ

Table 9.2. Pseudo-code Symbol Definitions

$\leftarrow$	Assignment
$=, \neq$ $<, >$ $\leq, \geq$	Comparison: equal, not equal less than, greater than less than or equal to, greater than or equal to
<<, >>	Logical Shift
^	Exponentiation
and,or, not, xor	Bitwise Logical Operations
mod	Modulo
+, -	Addition, Subtraction
*	Multiplication (Integer or Ordinal)
/	Division (Integer or Ordinal)
# . .	Comment delimiter
memory()	Memory access of specified width memory_{byte|short|word|long|triple|quad}() memory() Width implied by context

Faults

The *Faults* section lists faults that can be signaled as a direct result of instruction execution. Two possible faulting conditions are common to the entire instruction set and could directly result from any instruction. These fault types are abbreviated in the instruction reference.

Fault Type	**Subtype/Description**
Trace	*Instruction.* An Instruction Trace Event is signaled after instruction completion. A Trace fault is generated if both PC.te and TC.i=1. *Breakpoint.* A Breakpoint Trace Event is signaled after completion of an instruction for which there is a hardware breakpoint condition match and TC.br is set. A Trace fault is generated if PC.te and TC.br are both=1.
Operation	*Unimplemented.* An attempt to execute any instruction fetched from internal data RAM causes an operation unimplemented fault.

Three possible faulting conditions are common to large subsets of the instruction set:

Fault Type	**Subtype/Description**
Type	*Mismatch.* Any instruction that references a special function register while not in supervisor mode causes a type mismatch fault.
	Mismatch. Any instruction that attempts to write to internal data RAM while not in supervisor mode causes a type mismatch fault.
Operation	*Unimplemented.* Any instruction that causes an unaligned memory access causes an operation unimplemented fault if unaligned faults are not masked in the Processor Control Block (PRCB).

Other instructions can generate faults in addition to above faults. If an instruction can generate a fault, it is noted in the *Faults* section of the instruction reference.

Example

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

Opcode and Instruction Format

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

subi	593H	REG

The opcode is given in hexadecimal format. The instruction encoding format is one of four possible formats: REG, COBR, CTRL and MEM. Refer to *Appendix D, Instruction Set Reference* for more information on the formats.

See Also

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

INSTRUCTIONS

This section contains reference information on the processor's instructions. It is arranged alphabetically by instruction or instruction group.

addc

Mnemonic:	**addc**	Add Ordinal With Carry		
Format:	**addc**	*src1,* reg/lit/sfr	*src2,* reg/lit/sfr	*dst* reg/sfr

Description:

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

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

An integer overflow fault is never signaled with this instruction.

Action:

```
dst ← src2 + src1 + AC.cc1;
AC.cc0 ← 0CV2;
# V = 1 if integer addition would have generated an overflow.
# V = 0 otherwise

# C is carry out of the ordinal addition of src 2 and src 1
```

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr.*

Example:

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

Opcode:	**addc**	5B0H	REG

See Also: **addi, addo, subc, subi, subo**

addi, addo

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

Description: Adds *src2* and *src1* values and stores the result in *dst*.

Action: *dst* ← *src2* + *src1*;

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr*.
Arithmetic	*Integer Overflow.* Result too large for destination register (**addi** only). If overflow occurs and AC.om =1, fault is suppressed and AC.io is set to 1. Least significant 32-bits of the result are stored in *dst*.

Example:

```
addi r4, g5, r9        # r9 ← g5 + r4
```

Opcode:

addi	591H	REG
addo	590H	REG

See Also: **addc, subi, subo, subc**

alterbit

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

Description: Copies *src* value to *dst* with one bit altered. *bitpos* operand specifies bit to be changed; condition code determines value to which the bit is set. If condition code bit 1 = 1, selected bit is set; otherwise, it is cleared.

Action:

```
if (AC.cc1=0) dst ← src and not (2^(bitpos mod 32));
          else dst ← src or 2^(bitpos mod 32);
```

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr*.

Example:

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

Opcode: **alterbit** 58FH REG

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

and, andnot

Mnemonic:

and	And
andnot	And Not

Format:

and	*src1*, reg/lit/sfr	*src2*, reg/lit/sfr	*dst* reg/sfr
andnot	*src1*, reg/lit/sfr	*src2*, reg/lit/sfr	*dst* reg/sfr

Description:

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

{*src2* **andnot** (*src1*)}

rather than

{*src1* **andnot** (*src2*)}.

Action:

and: *dst* ← *src2* **and** *src1*;

andnot: *dst* ← *src2* **and not** (*src1*);

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr*.

Example:

```
and 0x17, g8, g2      # g2 ← g8 AND 0x17
andnot r3, r12, r9    # r9 ← r12 AND NOT r3
```

Opcode:

and	581H	REG
andnot	582H	REG

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

atadd

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

Description:

Adds *src* value (full word) to value in the memory location specified with *src/dst* operand. Initial value from memory is stored in *dst.*

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

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

Action:

```
tempa ← src/dst and not(0x3);    # force alignment to word boundary

temp ← memory_word (tempa);  # $\overline{\text{LOCK}}$ asserted at begin of read

memory_word (tempa) ← temp + src; # ordinal addition
                                  # $\overline{\text{LOCK}}$ deasserted after
                                  # memory write completes
dst ← temp;
```

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr.* And/or non-supervisor attempt to write to internal data RAM.

Example:

```
atadd r8, r2, r11     # r8 ← r2 + address r8, where r8
                      # specifies the address of a word
                      # in memory;
                      # r11 ← initial value, stored at
                      # address r8 in memory
```

Opcode:	**atadd**	612H	REG
See Also:	**atmod**		

atmod

Mnemonic: **atmod** Atomic Modify

Format:

atmod	*src*/dst	*mask*,	*src/dst*
	reg/sfr	reg/lit/sfr	reg
	addr		

Description: Moves selected bits of *src/dst* value into memory location specified in *src*. Bits set in *mask* operand select bits to be modified in memory. Initial value from memory is stored in *src/dst*.

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

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

Action:

```
tempa ← src and not (0x3);      # force alignment to word boundary
temp ← memory_word(tempa);      # LOCK asserted at
                                # beginning of memory read
memory_word(tempa) ← (src/dst and mask) or (temp and not(mask));
# LOCK deasserted after the memory write completes
src/dst ← temp;
```

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr* and/or non-supervisor attempt to write to internal data RAM.

Example:

```
atmod g5, g7, g10   # g5 ← g5 masked by g7, where g5
                    # specifies the address of a word in memory;
                    # g10 ← initial value, stored at
                    # address g5 in memory
```

Opcode: **atmod** 610H REG

See Also: **atadd**

b, bx

Mnemonic:	**b**	Branch
	bx	Branch Extended
Format:	**b**	*targ* disp
	bx	*efa* addr

efa:

(reg)	disp + 8(IP)	disp [reg * scale]
offset	disp	(reg1) [reg2 * scale]
offset (reg)	disp (reg)	disp (reg 1) [reg 2 * scale]

Description: Branches to the specified target.

With the **b** instruction, IP specified with *targ* operand can be no farther than -2^{23} to $(2^{23} - 4)$ bytes from current IP. When using the Intel i960 family assembler, *targ* operand must be a label which specifies target instruction's IP.

bx performs the same operation as **b** except the target instruction can be farther than -2^{23} to $(2^{23} - 4)$ bytes from current IP. Here, the target operand is an effective address, which allows the full range of addressing modes to be used to specify target instruction's IP. The "IP + displacement" addressing mode allows instruction to be IP-relative. Indirect branching can be performed by placing target address in a register then using a register-indirect addressing mode.

Refer to *Chapter 3, Data Types and Memory Addressing Modes* for a complete discussion of the addressing modes.

Action:

```
b:    IP ← IP + displacement;   # resume execution at new IP
bx:   IP ← efa;                 # resume execution at new IP
```

Faults:

Trace	*Instruction. Branch. Breakpoint.* Instruction and Branch Trace Events are signaled after instruction completion. Trace fault is generated if PC.te=1 and TC.i or TC.b=1.
Operation	*Unimplemented.* Execution from on-chip data RAM.
	Operand. Invalid operand value encountered. (**bx** only)

Opcode. Invalid operand encoding encountered (**bx** only).

Example:	b xyz	# IP ← xyz;
	bx 1330 (ip)	# IP ← IP + 8 + 1330;
		# this example uses ip-relative addressing

Opcode:	**b**	08H	CTRL
	bx	84H	MEM

See Also: **bal**, **balx**, BRANCH IF, COMPARE AND BRANCH, **bbc, bbs**

bal, balx

Mnemonic:	**bal**	Branch And Link
	balx	Branch And Link Extended
Format:	**bal**	*targ* disp
	balx	*efa*, *dst* addr reg

efa:

(reg)	disp + 8(IP)	disp [reg * scale]
offset	disp	(reg1) [reg2 * scale]
offset (reg)	disp (reg)	disp (reg 1) [reg 2 * scale]

Description:

Stores address of instruction following **bal** or **balx** then branches to specified target.

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

balx performs same operation as **bal** except next instruction address is stored in *dst*. With **balx**, target instruction can be farther than -2^{23} to $(2^{23} - 4)$ bytes from current IP. Here, the target operand is *efa*, which allows full range of addressing modes to be used to specify target IP. "IP + displacement" addressing mode allows instruction to be IP-relative. Indirect branching can be performed by placing target address in a register and then using a register-indirect addressing mode.

Refer to *Chapter 3, Data Types and Addressing Modes* for a complete discussion of addressing modes.

Action:

```
bal:   g14 ← IP + 4;                # next IP destination is always g14
       IP ← IP + displacement;      # resume execution at new IP

balx:  dst ← IP + inst length;      # instruction length is 4 or 8 bytes
       IP ← efa;                    # resume execution at the new IP
```

Faults: Trace *Instruction . Branch. Breakpoint.*

Instruction and Branch Trace Events are signaled after instruction completion. Trace fault is generated if PC.te=1 and TC.i or TC.br=1.

Operation	*Unimplemented.* Execution from on-chip data RAM.
	Operand. Invalid operand value encountered.
	Opcode. Invalid operand encoding encountered.

Example:

```
bal xyz             # IP ← xyz;

balx (g2), g4       # IP ← (g2);
                    # address of return instruction is stored in g4;
                    # example of indirect addressing.
```

Opcode:

bal	0BH	CTRL
balx	85H	MEM

See Also: **b**, **bx,** BRANCH IF, COMPARE AND BRANCH, **bbc, bbs**

bbc, bbs

Mnemonic:	**bbc**{.t|.f}	Check Bit and Branch If Clear
	bbs{.t|.f}	Check Bit and Branch If Set

Format:	**bb***{.t|.f}	*bitpos,*	*src,*	*targ*
		reg/lit	reg/sfr	disp

Description:

Checks bit in *src* (designated by *bitpos*) and sets AC register condition code according to *src* value. Processor then performs conditional branch to instruction specified with *targ*, based on 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 suffix is not provided, assembler is free to provide one.

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

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

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

Action:

bbc:

```
if ((src and 2^(bitpos mod 32)) = 0)
        {
        AC.cc ← 010₂;
        IP ← IP + 4 + (displacement * 4);
        # resume execution at new IP
        }
else AC.cc ← 000₂;
# resume execution at next IP
```

bbs:

```
if ((src and 2^(bitpos mod 32)) = 1)
        {
        AC.cc ← 010₂;
        IP ← IP + 4 + (displacement * 4);
        # resume execution at new IP
        }
else AC.cc ← 000₂;
# resume execution at next IP
```

Faults:

Trace	*Instruction. Branch* (if taken). *Breakpoint.* Instruction and Branch Trace Events are signaled after instruction completion. Trace fault is generated if PC.te=1 and TC.i or TC.b=1.
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr.*

Example:

```
# assume bit 10 of r6 is clear
bbc 10, r6, xyz   # bit 10 of r6 is checked
                  # and found clear;
                  # AC.cc ← 010
                  # IP ← xyz;
```

Opcode:

bbc	30H	COBR
bbs	37H	COBR

See Also:

chkbit, b, bx bal, balx, COMPARE AND BRANCH, **bbc, bbs,** BRANCH IF

BRANCH IF

Mnemonic:

be{**.t**|**.f**}	Branch If Equal/True
bne{**.t**|**.f**}	Branch If Not Equal
bl{**.t**|**.f**}	Branch If Less
ble{**.t**|**.f**}	Branch If Less Or Equal
bg{**.t**|**.f**}	Branch If Greater
bge{**.t**|**.f**}	Branch If Greater Or Equal
bo{**.t**|**.f**}	Branch If Ordered
bno{**.t**|**.f**}	Branch If Unordered/False

Format: **b***{**.t**|**.f**} *targ*
disp

Description:

Branches to instruction specified with *targ* 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, assembler is free to provide one.

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

For **bno**, the processor branches to instruction specified with *targ* if logical AND of condition code and mask-part of opcode is zero. Otherwise, it goes to next instruction.

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

NOTE

bo and **bno** are used by implementations that include floating point coprocessor for branch operations involving real numbers. **bno** can be used as branch-if-false instruction when used after **chkbit**. **be** can be used as branch-if-true instruction when following **chkbit.**

targ value or absolute addresses can be no farther than -2^{23} to $(2^{23} - 4)$ bytes from current IP. When using the Intel i960 family assembler, *targ* must be a label which specifies target instruction's IP.

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

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

Action:

For all instructions except **bno**:

```
if ((mask and AC.cc) ≠ 000₂) IP ← IP + displacement;
                    # resume execution at new IP
else;               # resume execution at next IP
```

bno:

```
if (AC.cc = 000₂) IP ← IP + displacement;
                    # resume execution at new IP
else                # resume execution at next IP
```

Faults:

Trace	*Instruction. Branch* (if taken). *Breakpoint.* Instruction and Branch Trace Events are signaled after instruction completion. Trace fault is generated if PC.te=1 and TC.i or TC.b=1.
Operation	*Unimplemented.* Execution from on-chip data RAM.

Example:

```
# assume (AC.cc AND 100₂) ≠ 0
bl xyz      # IP ← xyz;
```

Opcode:

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

See Also: **b**, **bx, bbc, bbs,** COMPARE AND BRANCH, **bal, balx,** BRANCH IF

call

Mnemonic: **call** Call

Format: **call** *targ*
disp

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

In executing this instruction, the processor performs a local call operation as described in *Local Calls* section of *Chapter 5, Procedure Calls*. As part of this operation, the processor saves the set of local registers associated with the calling procedure and allocates a new set of local registers and a new stack frame for the called procedure. Processor then goes to the instruction specified with *targ* and begins execution.

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

Action:

```
wait for any uncompleted instructions to finish;
temp ← (SP + 0x10) and not (0xf);  # round to next boundary,
memory(FP) ← r0:15;                # these accesses are cached in
RIP← next IP                       # local register cache
PFP ← FP;
PFP.rt ← 000₂;
FP ← temp;
SP ← temp + 64;
IP ← IP + displacement;
```

Faults:

Trace	*Instruction. Call. Breakpoint.* Instruction and Call Trace Events are signaled after instruction completion. Trace fault is generated if PC.te=1 and TC.i or TC.c is=1.
Operation	*Unimplemented.* Execution from on-chip data RAM.

Example: call xyz # IP ← xyz

Opcode: **call** 09H CTRL

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

calls

Mnemonic: **calls** Call System

Format: **calls** *src*
reg/lit/sfr

Description: Calls a system procedure. *targ* specifies called procedure's number. For **calls**, the processor performs system call operation described in *System Calls* section of *Chapter 5, Procedure Calls. targ* provides an index to a system procedure table entry from which the processor gets the called procedure's IP.

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

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

Action:

```
if (src> 259) Protection-length fault;
wait for any uncompleted instructions to finish;
temp_entry ← memory_word(SPT(src));
# SPT(src) is the address of the system procedure table entry targ.
RIP ← next IP;
if ((temp_entry.type = local) or (PC.em = supervisor))
            {                   # no stack switch required
                                # round to next boundary,
            temp_FP ← (SP + 0x10) and not(0xf);
   temp_rt ← 000₂;              # return type is local
            }
else
            {                   # stack switch to supervisor stack
                                # required; read supervisor
            temp_FP ← memory_word(cached(SPT);

                                # stack pointer
                                # set return type to supervisor

            if (PC.te = 0) temp_rt ← 010₂;     # with trace disabled
            else temp_rt ← 011₂;               # with trace enabled
            PC.em ← supervisor;
            # Trace enable bit of the supervisor
            PC.te ← temp_FP.T;
            # stack pointer is written to PC.te
            }
```

```
# These accesses are cached in the local register cache.
memory(FP) ← r0:15
PFP ← FP;
PFP.ft ← temp_rt;
FP ← temp_FP;
SP ← temp_FP + 64;
IP ← temp_entry and not (0x3);
```

Faults:

Trace	*Instruction. Call. Supervisor. Breakpoint.* Instruction, Call and Supervisor Trace Events are signaled after instruction completion. Trace fault is generated if PC.te=1 and TC.i, TC.c or TC.s=1.
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr.*
Protection	*Length.* Specified a system procedure number greater than 259.

Example:

```
calls r12          # IP ← value obtained from
                   # procedure table for procedure
                   # number given in r12
```

Opcode:

calls	660H	REG

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

callx

Mnemonic: **callx** Call Extended

Format: **callx** *efa*
addr

efa:

(reg)	disp + 8(IP)	disp [reg * scale]
offset	disp	(reg1) [reg2 * scale]
offset (reg)	disp (reg)	disp (reg 1) [reg 2 * scale]

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

In executing **callx**, the processor performs a local call as described in *Local Calls* section of *Chapter 5, Procedure Calls.* 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 *efa* and begins execution of new procedure.

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

efa 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 calls can be performed by placing the target address in a register and then using a register-indirect addressing mode.

Refer to *Chapter 3, Data Types and Memory Addressing Modes* for a complete discussion of addressing modes.

Action:

```
wait for any uncompleted instructions to finish;
temp ← (SP + 0x10) and not (0xf);   # round to next boundary
RIP ← next IP;
memory(FP) ← r0:15                  # these accesses are cached in
                                    # local register cache
PFP ← FP;
PFP.rt ← 000₂
FP ← temp;
SP ← temp + 64;
IP ← efa;
```

Faults:	Trace	*Instruction. Call. Breakpoint.* Instruction and Call Trace Events are signaled after instruction completion. Trace fault is generated if PC.te=1 and TC.i or TC.c=1.
	Operation	*Unimplemented.* Execution from on-chip data RAM.
		Operand. Invalid operand value encountered.
		Opcode. Invalid operand encoding encountered.

Example:

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

Opcode:	**callx**	86H	MEM

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

chkbit

Mnemonic:	**chkbit**	Check Bit	
Format:	**chkbit**	*bitpos*, reg/lit/sfr	*src* reg/lit/sfr

Description: Checks bit in *src* designated by *bitpos* and sets condition code according to value found. If bit is set, condition code is set to 010_2; if bit is clear, condition code is set to 000_2.

Action:

```
if ((src and 2^(bitpos mod 32)) = 0) AC.cc ← 000₂;
        else AC.cc ← 010₂;
```

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr.*

Example:

```
chkbit 13, g8        # checks bit 13 in g8 and
                     # sets AC.cc according to the result
```

Opcode: **chkbit** 5AEH REG

See Also: **alterbit, clrbit, notbit, setbit, cmpi, cmpo**

clrbit

Mnemonic:	**clrbit** Clear Bit
Format:	**clrbit** *bitpos*, reg/lit/sfr *src*, reg/lit/sfr *dst* reg/sfr
Description:	Copies *src* value to *dst* with one bit cleared. *bitpos* operand specifies bit to be cleared.
Action:	*dst* ← *src* **and** **not**(2^(*bitpos* **mod** 32));
Faults:	Trace — *Instruction. Breakpoint.*
	Operation — *Unimplemented.* Execution from on-chip data RAM.
	Type — *Mismatch.* Non-supervisor reference of a *sfr*.
Example:	clrbit 23, g3, g6 # g6 ← g3 with bit 23 cleared
Opcode:	**clrbit** 58CH REG
See Also:	**alterbit**, **chkbit**, **notbit**, **setbit**

cmpdeci, cmpdeco

Mnemonic:

cmpdeci	Compare and Decrement Integer
cmpdeco	Compare and Decrement Ordinal

Format:

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

Description:

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

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

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

Action:

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

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr.*

Example:

```
cmpdeci 12, g7, g1   # compares g7 with 12 and sets
                     # AC.cc to indicate the result;
                     # g1 ← g7 - 1
```

Opcode:

cmpdeci	5A7H	REG
cmpdeco	5A6H	REG

See Also:

cmpinco, cmpo, cmpi, cmpinci, COMPARE AND BRANCH

cmpi, cmpo

Mnemonic:	**cmpi**	Compare Integer	
	cmpo	Compare Ordinal	
Format:	**cmp***	*src1*, reg/lit/sfr	*src2* reg/lit/sfr

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

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

cmpi followed by a branch-if instruction is equivalent to a compare-integer-and-branch instruction. The latter method of comparing and branching produces more compact code; however, the former method can result in faster running code if used to take advantage of pipelining in the architecture. Same is true for **cmpo** and the compare-ordinal-and-branch instructions.

Action:

```
if (src1 < src2) AC.cc ← 100₂;
else if (src1 = src2) AC.cc ← 010₂;
        else AC.cc ← 001₂;
```

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr.*

Example:

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

Opcode:

cmpi	5A1H	REG
cmpo	5A0H	REG

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

cmpinci, cmpinco

Mnemonic:

cmpinci	Compare and Increment Integer
cmpinco	Compare and Increment Ordinal

Format:

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

Description:

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

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

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

Action:

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

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr*.

Example:

```
cmpinco r8, g2, g9  # compares the values in g2 and
                    # r8 and sets AC.cc to indicate the result;
                    # g9 ← g2 + 1
```

Opcode:

cmpinci	5A5H	REG
cmpinco	5A4H	REG

See Also:

cmpdeco, cmpo, cmpi, cmpdeci, COMPARE AND BRANCH

COMPARE AND BRANCH

Mnemonic:

cmpibe{**.t**|**.f**}	Compare Integer And Branch If Equal
cmpibne{**.t**|**.f**}	Compare Integer And Branch If Not Equal
cmpibl{**.t**|**.f**}	Compare Integer And Branch If Less
cmpible{**.t**|**.f**}	Compare Integer And Branch If Less Or Equal
cmpibg{**.t**|**.f**}	Compare Integer And Branch If Greater
cmpibge{**.t**|**.f**}	Compare Integer And Branch If Greater Or Equal
cmpibo{**.t**|**.f**}	Compare Integer And Branch If Ordered
cmpibno{**.t**|**.f**}	Compare Integer And Branch If Not Ordered
cmpobe{**.t**|**.f**}	Compare Ordinal And Branch If Equal
cmpobne{**.t**|**.f**}	Compare Ordinal And Branch If Not Equal
cmpobl{**.t**|**.f**}	Compare Ordinal And Branch If Less
cmpoble{**.t**|**.f**}	Compare Ordinal And Branch If Less Or Equal
cmpobg{**.t**|**.f**}	Compare Ordinal And Branch If Greater
cmpobge{**.t**|**.f**}	Compare Ordinal And Branch If Greater Or Equal

Format:

cmpib*{**.t**|**.f**}	*src1*,	*src2*,	*targ*
	reg/lit	reg/sfr	disp
cmpob*{.t|.f}	*src1*,	*src2*,	*targ*
	reg/lit	reg/sfr	disp

Description:

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

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 suffix is not provided, assembler is free to provide one.

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

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

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

NOTE

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

Functions that these instructions perform can be duplicated with a **cmpi** or **cmpo** followed by a branch-if instruction, as described in this chapter for the **cmpi** and **cmpo** instructions.

Action:

```
if (src1 < src2) AC.cc ← $100_2$;
else if (src1 = src2) AC.cc ← $010_2$;
          else AC.cc ← $001_2$;
if ((mask and AC.cc) ≠ $000_2$) IP ← IP + 4 + (displacement * 4);
                       # resume execution at the new IP
else IP ← IP + 4;      # resume execution at the next IP
```

Faults:

Trace	*Instruction. Branch* (if taken). *Breakpoint.* Instruction and Branch Trace Events are signaled after instruction completion. Trace fault is generated if PC.te=1 and TC.i or TC.br=1.
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr.*

Example:

```
# assume g3 < 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.
```

Opcode:

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

See Also: BRANCH IF, **cmpi**, **cmpo, bal, balx**

concmpi, concmpo

Mnemonic:

concmpi	Conditional Compare Integer
concmpo	Conditional Compare Ordinal

Format:

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

Description:

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

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

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

Action:

```
if (AC.cc2 = 0)
        {
        if (src1 ≥ src2) AC.cc ← 010₂;
        else AC.cc ← 001₂;
        };
```

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr.*

Example:

```
cmpo g6, g3        # compares g6 and g3 and
                   # sets AC.cc
concmpo g5, g3     # if AC.cc ≠ 1XX,
                   # g5 is compared with g3
```

Opcode:

concmpi	5A3H	REG
concmpo	5A2H	REG

See Also:

cmpo, cmpi, cmpdeci, cmpdeco, cmpinci, cmpinco, COMPARE AND BRANCH

divi, divo

Mnemonic:

divi	Divide Integer
divo	Divide Ordinal

Format:

div*	*src1*,	*src2*,	*dst*
	reg/lit/sfr	reg/lit/sfr	reg/sfr

Description:

Divides *src2* value by *src1* value and stores quotient of the result in *dst*. Remainder (if any) is discarded.

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

Action:

```
if (src2 = 0) Arithmetic Zero Divide fault;
dst ← quotient(src2 / src1);
# src2, src1 and dst are 32-bits
```

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr*.
Arithmetic	*Zero Divide.* The *src1* operand is 0. *Integer Overflow.* Result too large for destination register (**divi** only). If overflow occurs and AC.om=1, fault is suppressed and AC.io is set to 1. Result's least significant 32-bits are stored in *dst*.

Example:

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

Opcode:

divi	74BH	REG
divo	70BH	REG

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

ediv

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

Description: Divides *src2* by *src1* and stores result in *dst*. The *src2* value is a long ordinal (64 bits) contained in two adjacent registers. *src2* specifies the lower numbered register which contains operand's least significant bits. *src2* must be an even numbered register (i.e., r0, r2, r4, ... or g0, g2, ... or sf0, sf2, ...). *src1* value is a normal ordinal (i.e., 32 bits).

The result consists of a one-word remainder and a one-word quotient. Remainder is stored in the register designated by *dst*; quotient is stored in the next highest numbered register. *dst* must be an even numbered register (i.e., r0, r2, r4, ... or g0, g2, ... 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:

```
if (src2=0) Arithmetic Zero Divide fault;
dst ← (src2 - (src2 / src1) * src1);          # remainder
dst + 1 ← (src2 / src1);                      # quotient
# src2 is 64-bits; src1, dst and dst+1 are 32-bits
```

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr*.
Arithmetic	*Zero Divide.* The *src1* operand is 0.

Example:

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

Opcode: **ediv** 671H REG

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

emul

Mnemonic:	**emul**	Extended Multiply		
Format:	**emul**	*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., r0, r2, r4, ... or g0, g2, ... or sf0, sf2, ...).

This instruction performs ordinal arithmetic.

Action: *dst* ← *src2* * *src1*; # *src1* and *src2* are 32-bits; *dst* is 64-bits.

Faults:	Trace	*Instruction. Breakpoint.*
	Operation	*Unimplemented.* Execution from on-chip data RAM.
	Type	*Mismatch.* Non-supervisor reference of a *sfr.*
Example:	emul r4, r5, g2	# g2,g3 ← r4 * r5
Opcode:	**emul** 670H	REG
See Also:	**ediv, muli, mulo**	

eshro

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

Description:

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

src2 value is a long ordinal (i.e., 64 bits) contained in two adjacent registers. *src2* operand specifies the lower numbered register, which contains operand's least significant bits. *src2* operand must be an even numbered register (i.e., r0, r2, r4, ... or g0, g2, ... or sf0, sf2, ...).

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

The shift operation result's least significant 32 bits is stored in *dst*.

Action:

dst ← *src2* >> (*src1* **mod** 32);
# *src2* is 64 bits, *src1* and *dst* are 32 bits

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr*.

Example:

eshro g3, g4, g11 # g11 ← g4,5 shifted right by (g3 MOD 32)

Opcode:

eshro	5D8H	REG

See Also:

SHIFT, **extract**

extract

Mnemonic:	**extract**	Extract	
Format:	**extract** *bitpos,* reg/lit/sfr	*len,* reg/lit/sfr	*src/dst* reg

Description: Shifts a specified bit field in *src/dst* right and zero fills bits to left of shifted bit field. *bitpos* value specifies the least significant bit of the bit field to be shifted; *len* value specifies bit field length.

Action: *src/dst* ← (*src/dst* /2^ (*bitpos* **mod** 32)) **and** (2^(*len* - 1);

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr.*

Example: extract 5, 12, g4   # g4 ← g4 with bits 5 through 16 shifted right

Opcode: **extract**   651H   REG

See Also: modify

FAULT IF

Mnemonic:

faulte{**.t**|**.f**}	Fault If Equal
faultne{**.t**|**.f**}	Fault If Not Equal
faultl{**.t**|**.f**}	Fault If Less
faultle{**.t**|**.f**}	Fault If Less Or Equal
faultg{**.t**|**.f**}	Fault If Greater
faultge{**.t**|**.f**}	Fault If Greater Or Equal
faulto{**.t**|**.f**}	Fault If Ordered
faultno{**.t**|**.f**}	Fault If Not Ordered

Format: **fault***{**.t**|**.f**}

Description:

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

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.

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

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

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

Action:

For all instructions except **faultno**:

```
        if ((mask and AC.cc) ≠ 000₂) Constraint-range fault;
```

faultno:

```
        if (AC.cc=000₂) Constraint-range fault;
```

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Constraint	*Range.* If condition being tested is true.

Example:

```
# assume (AC.cc AND 110₂) - 000₂
faultle      # Constraint Range Fault is generated
```

Opcode:

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

See Also: BRANCH IF, TEST

flushreg

Mnemonic: **flushreg** Flush Local Registers

Format: **flushreg**

Description: Copies the contents of every cached register set, except the current set, to its associated stack frame in memory. The entire register cache is then marked as purged (or invalid). On a return to a stack frame for which the local registers are not cached, the processor reloads the locals from memory.

flushreg is provided to allow a compiler or applications program to circumvent the processor's normal call/return mechanism. For example, a compiler may need to go back several frames in the stack on the next return, rather than using the normal return mechanism that returns one frame at a time. Since the local registers of an unknown number of previous stack frames may be cached, a flushreg must be executed prior to modifying the PFP to return to a frame other than the one directly below the current frame.

Action: Write all cached local register sets – except the current set – to memory; Invalidate the local register cache.

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor attempt to write to internal data RAM.

Example: flushreg

Opcode: **flushreg** 66D REG

fmark

Mnemonic: **fmark** Force Mark

Format: **fmark**

Description: Generates a breakpoint trace event. Causes a breakpoint trace event to be generated, regardless of breakpoint trace mode flag setting, providing the PC register trace enable bit (bit 0) is set.

When a breakpoint trace event is detected, the PC register trace-fault-pending flag (bit 10) and the TC register breakpoint-trace-event flag (bit 23) are set. Then, a breakpoint-trace fault is generated before the next instruction executes.

For more information on trace fault generation, refer to *Chapter 7, Faults.*

Action:

```
if (PC.te=1)
        {
        PC.tfp ← 1;
        TC.bte ← 1;
        Trace Breakpoint trace fault
        }
```

Faults:

Trace	*Instruction. Breakpoint.* Instruction and Breakpoint Trace Events are signaled after instruction completion. Trace fault is generated if PC.te=1.
Operation	*Unimplemented.* Execution from on-chip data RAM.

Example:

```
ld xyz, r4
addi r4, r5, r6
fmark
# Breakpoint trace event is generated at
# this point in the instruction stream.
```

Opcode: **fmark** 66CH REG

See Also: **mark**

LOAD

Mnemonic:

ld	Load
ldob	Load Ordinal Byte
ldos	Load Ordinal Short
ldib	Load Integer Byte
ldis	Load Integer Short
ldl	Load Long
ldt	Load Triple
ldq	Load Quad

Format:

ld*	*efa*,	*dst*
	addr	reg

efa:

(reg)	disp + 8(IP)	disp [reg * scale]
offset	disp	(reg1) [reg2 * scale]
offset (reg)	disp (reg)	disp (reg 1) [reg 2 * scale]

Description:

Copies byte or byte string from memory into a register or group of successive registers.

efa specifies the address of first byte to be loaded. The full range of addressing modes may be used in specifying *efa*. (Refer to *Chapter 3* section titled *Addressing Modes* for description of addressing modes.)

dst specifies a register or the first (lowest numbered) register of successive registers.

ldob and **ldib** load a byte and **ldos** and **ldis** load a half word and convert it to a full 32-bit word. Data being loaded is sign-extended during integer loads and zero-extended during ordinal loads.

ld, **ldl**, **ldt** and **ldq** instructions copy 4, 8, 12 and 16 bytes, respectively, from memory into successive registers.

For **ldl**, *dst* must specify an even numbered register (e.g., g0, g2, ... or r0, r2, ...). For **ldt** and **ldq**, *dst* must specify a register number that is a multiple of four (e.g., g0, g4, g8, ... or r0, r4, r8, ...). Results are unpredictable if registers are not aligned on the required boundary or if data extends beyond register g15 or r15 for **ldl**, **ldt** or **ldq**.

Action:

ld:	*dst* ← memory_word (*efa*);
ldob:	*dst* ← memory_byte (*efa*) zero-extended to 32 bits;
ldos:	*dst* ← memory_short (*efa*) zero-extended to 32 bits;
ldib:	*dst* ← memory_byte (*efa*) sign-extended to 32 bits;
ldis:	*dst* ← memory_short (*efa*) sign-extended to 32 bits;
ldl:	*dst* ← memory_long (*efa*);
ldt:	*dst* ← memory_triple (*efa*);
ldq:	*dst* ← memory_quad (*efa*);

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
	Unimplemented. An unaligned *efa* was referenced; unaligned support was disabled.
	Operand. Invalid operand value encountered.
	Opcode. Invalid opcode encoding encountered.

Example: ldl 2450 (r3), r10 # r10, r11 ← r3 + 2450 in memory

Opcode:

ld	90H	MEM
ldob	80H	MEM
ldos	88H	MEM
ldib	C0H	MEM
ldis	C8H	MEM
ldl	98H	MEM
ldt	A0H	MEM
ldq	B0H	MEM

See Also: MOVE, STORE

lda

Mnemonic: **lda** Load Address

Format: **lda** *efa*, *dst*
reg

efa:

(reg)	disp + 8(IP)	disp [reg * scale]
offset	disp	(reg1) [reg2 * scale]
offset (reg)	disp (reg)	disp (reg 1) [reg 2 * scale]

Description: Computes the effective address (*efa*) and stores it in *dst*. Computed value is not checked for validity. Any addressing mode may be used to calculate *efa*.

An important application of this instruction is to load a constant longer than 5 bits into a register. (To load a register with a constant of 5 bits or less, **mov** can be used with a literal as the *src* operand.)

Action: *dst* ← efa;

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
	Operand. Invalid operand value encountered.
	Opcode. Invalid opcode encoding encountered.

Example:

```
lda 58 (g9), g1     # g1 ← effective address of g9 + 58
lda 0x749, r8       # r8 ← constant 0x749
```

Opcode: **lda** 8CH MEM

mark

Mnemonic: **mark** Mark

Format: **mark**

Description: Generates breakpoint trace event if breakpoint trace mode is enabled. Breakpoint trace mode is enabled if the PC register trace enable bit (bit 0) and the TC register breakpoint trace mode bit (bit 7) are set.

When a breakpoint trace event is detected, the PC register trace-fault-pending flag (bit 10) and the TC register breakpoint-trace-event flag (bit 23) are set. Then, before the next instruction is executed, a breakpoint trace fault is generated.

If breakpoint trace mode is not enabled, **mark** behaves like a no-op.

For more information on trace fault generation, refer to *Chapter 8, Tracing and Debugging*.

Action:

```
if ((PC.te=1) and (TC.br=1))
        {
        PC.tfp ← 1;
        TC.bte ← 1;
        Trace Breakpoint trace fault;
        }
```

Faults:

Trace	*Instruction. Breakpoint* (if enabled). Instruction and Breakpoint Trace Events are signaled after instruction completion. Trace fault is generated if PC.te=1 and TC.i or TC.br=1.
Operation	*Unimplemented.* Execution from on-chip data RAM.

Example:

```
# Assume that the breakpoint trace mode is enabled.
ld xyz, r4
addi r4, r5, r6
mark
# Breakpoint trace event is generated at this point
# in the instruction stream.
```

Opcode: **mark** 66BH REG

See Also: **fmark**, **modpc**, **modtc**

modac

Mnemonic:	**modac**	Modify AC		
Format:	**modac**	*mask*, reg/lit/sfr	*src*, reg/lit/sfr	*dst* reg/sfr

Description: Reads and modifies the AC register. *src* contains the value to be placed in the AC register; *mask* specifies bits that may be changed. Only bits set in *mask* are modified. Once the AC register is changed, its initial state is copied into *dst*.

Action:

```
temp ← AC
AC ← (src and mask) or (AC and not (mask));
dst ← temp;
```

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr.*

Example:

```
modac g1, g9, g12   # AC ← g9, masked by g1
                    # g12 ← initial value of AC
```

Opcode: **modac** 645H REG

See Also: **modpc**, **modtc**

modi

Mnemonic:	**modi** Modulo Integer

Format:

modi	*src1*,	*src2*,	*dst*
	reg/lit/sfr	reg/lit/sfr	reg/sfr

Description: Divides *src2* by *src1*, where both are integers and stores the modulo remainder of the result in *dst*. If the result is nonzero, *dst* has the same sign as *src1*.

Action:

```
if (src1= 0) Arithmetic Zero Divide fault;
dst ← src2 - ((src2/src1) * src1);
if ((src2 * src1 < 0) and (dst ≠ 0)) dst ← dst + src1;
# src1, src2 and dst are 32 bits
```

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr*.
Arithmetic	*Zero Divide*. The *src1* operand is 0.

Example:

```
modi r9, r2, r5        # r5 ← modulo (r2/r9)
```

Opcode:

modi	749H	REG

See Also: **divi, divo, remi**

modify

Mnemonic:	**modify**	Modify		
Format:	**modify**	*mask*, reg/lit/sfr	*src*, reg/lit/sfr	*src/dst* reg

Description: Modifies selected bits in *src/dst* with bits from *src*. The *mask* operand selects the bits to be modified: only bits set in the *mask* operand are modified in *src/dst*.

Action: *src/dst* ← (*src* **and** *mask*) **or** (*src/dst* **and not** (*mask*));

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr*.

Example:

```
modify g8, g10, r4   # r4 ← g10 masked by g8
```

Opcode: **modify** 650H REG

See Also: **alterbit**, **extract**

modpc

Mnemonic: **modpc** Modify Process Controls

Format:

modpc	*src,*	*mask,*	*src/dst*
	reg/lit/sfr	reg/lit/sfr	reg

Description: Reads and modifies the PC register as specified with *mask* and *src/dst*. *src/dst* operand contains the value to be placed in the PC register; *mask* operand specifies bits that may be changed. Only bits set in the *mask* are modified. Once the PC register is changed, its initial value is copied into *src/dst*. The *src* operand is a dummy operand that should specify a literal or the same register as the *mask* operand.

The processor must be in supervisor mode to use this instruction with a non-zero *mask* value. If *mask*=0, this instruction can be used to read the process controls, without the processor being in supervisor mode.

If the action of this instruction results in processor priority being lowered, the interrupt table is checked for pending interrupts.

Changing the PC register reserved fields can lead to unpredictable behavior as described in *Chapter 2, Programming Environment.*

Action:

```
if ((mask ≠ 0)
            {
            if (PC.em ≠ supervisor)) Type-mismatch fault;
            temp ← PC;
            PC ← (mask and src/dst) or (PC and not (mask));
            src/dst ← temp;
            if (temp.p > PC.p) check_pending_interrupts;
            }
else src/dst ← PC;
```

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr*.
	Mismatch. Attempted to execute instruction with non-zero *mask* value while not in supervisor mode.

Example: modpc g9, g9, g8 # process controls ← g8 masked by g9

Opcode: **modpc** 655H REG

See Also: **modac, modtc**

modtc

Mnemonic: **modtc** Modify Trace Controls

Format:

modtc	*mask,*	*src,*	*dst*
	reg/lit/sfr	reg/lit/sfr	reg/sfr

Description:

Reads and modifies TC register as specified with *mask* and *src*. The *src* operand contains the value to be placed in the TC register; *mask* operand specifies bits that may be changed. Only bits set in *mask* are modified. *mask* must not enable modification of reserved bits. Once the TC register is changed, its initial state is copied into *dst*.

The changed trace controls may take effect immediately or may be delayed. If delayed, the changed trace controls may not take effect until after the first non-branching instruction is fetched from memory or after four non-branching instructions are executed.

For more information on the trace controls, refer to *Chapter 7, Faults* and *Chapter 8, Tracing and Debugging*.

Action:

```
temp ← TC;
TC ← (mask and src) or (temp and not(mask));
dst ← temp;
```

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr.*

Example:

```
modtc g12, g10, g2  # trace controls ← g10 masked by
                    # g12; previous trace controls stored in g2
```

Opcode: **modtc** 654H REG

See Also: **modac**, **modpc**

9

MOVE

Mnemonic:

mov	Move
movl	Move Long
movt	Move Triple
movq	Move Quad

Format:

mov*	*src,*	*dst*
	reg/lit/sfr	reg/sfr

Description:

Copies the contents of one or more source registers (specified with *src*) to one or more destination registers (specified with *dst*).

For **movl**, **movt** and **movq**, *src* and *dst* specify the first (lowest numbered) register of several successive registers. *src* and *dst* registers must be even numbered (e.g., g0, g2, ... or r0, r2, ... or sf0, sf2, ...) for **movl** and an integral multiple of four (e.g., g0, g4, ... or r0, r4, ... or sf0, sf4, ...) for **movt** and **movq**.

The moved register values are unpredictable when: 1) the *src* and *dst* operands overlap; 2) registers are not properly aligned.

Action:

dst ← *src*;

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr*.

Example:

```
movt g8, r4        # r4, r5, r6 ← g8, g9, g10
```

Opcode:

mov	5CCH	REG
movl	5DCH	REG
movt	5ECH	REG
movq	5FCH	REG

See Also:

LOAD, STORE, **lda**

muli, mulo

Mnemonic:	**muli**	Multiply Integer
	mulo	Multiply Ordinal

Format:	**mul***	*src1,*	*src2,*	*dst*
		reg/lit/sfr	reg/lit/sfr	reg/sfr

Description: Multiplies the *src2* value by the *src1* value and stores the result in *dst.*

Action:

```
dst ← src2 * src1;
# src1, src2 and dst are 32 bits
```

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr.*
Arithmetic	*Integer Overflow.* Result is too large for destination register (**muli** only). If overflow occurs and AC.om=1, the fault is suppressed and AC.io is set to 1. Result's least significant 32 bits are stored in *dst.*

Example:

```
muli r3, r4, r9        # r9 ← r4 TIMES r3
```

Opcode:

muli	741H	REG
mulo	701H	REG

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

nand

Mnemonic:	**nand**	Nand		
Format:	**nand**	*src1*, reg/lit/sfr	*src2*, reg/lit/sfr	*dst* reg/sfr

Description: Performs a bitwise NAND operation on *src2* and *src1* values and stores the result in *dst*.

Action: *dst* ← (**not** (*src2*)) **or** (**not** (*src1*));

Faults:	Trace	*Instruction. Breakpoint.*
	Operation	*Unimplemented.* Execution from on-chip data RAM.
	Type	*Mismatch.* Non-supervisor reference of a *sfr*.

Example: nand g5, r3, r7 # r7 ← r3 NAND g5

Opcode: **nand** 58EH REG

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

nor

Mnemonic:	**nor**	Nor		
Format:	**nor**	*src1,* reg/lit/sfr	*src2,* reg/lit/sfr	*dst* reg/sfr

Description: Performs a bitwise NOR operation on the *src2* and *src1* values and stores the result in *dst.*

Action: *dst* ← (**not** (*src2*)) **and** (**not** (*src1*));

Faults:	Trace	*Instruction. Breakpoint.*
	Operation	*Unimplemented.* Execution from on-chip data RAM.
	Type	*Mismatch.* Non-supervisor reference of a *sfr.*

Example:

```
nor g8, 28, r5        # r5 ← 28 NOR g8
```

Opcode: **nor** 588H REG

See Also: **and**, **andnot**, **nand**, **not**, **notand**, **notor**, **or**, **ornot**, **xnor**, **xor**

not, notand

Mnemonic:	**not**	Not		
	notand	Not And		
Format:	**not**	*src*,	*dst*	
		reg/lit/sfr	reg/sfr	
	notand	*src1*,	*src2*,	*dst*
		reg/lit/sfr	reg/lit/sfr	reg/sfr

Description: Performs A bitwise NOT (**not** instruction) or NOT AND (**notand** instruction) operation on the *src2* and *src1* values and stores the result in *dst*.

Action:

not: *dst* ← **not** (*src*);

notand: *dst* ← (**not** (*src2*)) **and** *src1*;

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr*.

Example:

```
not g2, g4           # g4 ← NOT g2
notand r5, r6, r7    # r7 ← NOT r6 AND r5
```

Opcode:

not	58AH	REG
notand	584H	REG

See Also: **and**, **andnot**, **nand**, **nor**, **notor**, **or**, **ornot**, **xnor**, **xor**

notbit

Mnemonic:	**notbit**	Not Bit		
Format:	**notbit**	*bitpos*, reg/lit/sfr	*src*, reg/lit/sfr	*dst* reg/sfr

Description: Copies the *src* value to *dst* with one bit toggled. The *bitpos* operand specifies the bit to be toggled.

Action: *dst* ← *src* **xor** 2^(*bitpos* **mod** 32);

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr*.

Example:

```
notbit r3, r12, r7     # r7 ← r12 with the bit
                       # specified in r3 toggled
```

Opcode: **notbit** 580H REG

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

notor

Mnemonic:	**notor**	Not Or	
Format:	**notor** *src1*, reg/lit/sfr	*src2*, reg/lit/sfr	*dst* reg/sfr

Description: Performs a bitwise NOT OR operation on *src2* and *src1* values and stores result in *dst*.

Action: *dst* ← (**not** (*src2*)) **or** *src1*;

Faults:		
	Trace	*Instruction. Breakpoint.*
	Operation	*Unimplemented.* Execution from on-chip data RAM.
	Type	*Mismatch.* Non-supervisor reference of a *sfr*.

Example: notor g12, g3, g6 # g6 ← NOT g3 OR g12

Opcode: **notor** 58DH REG

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

or, ornot

Mnemonic:	**or**	Or		
	ornot	Or Not		
Format:	**or**	*src1*,	*src2*,	*dst*
		reg/lit/sfr	reg/lit/sfr	reg/sfr
	ornot	*src1*,	*src2*,	*dst*
		reg/lit/sfr	reg/lit/sfr	reg/sfr

Description: Performs a bitwise OR (**or** instruction) or ORNOT (**ornot** instruction) operation on the *src2* and *src1* values and stores the result in *dst*.

Action:

or: *dst* ← *src2* **or** *src1*;
ornot: *dst* ← *src2* **or** (**not** (*src1*));

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr*.

Example:

```
or 14, g9, g3          # g3 ← g9 OR 14
ornot r3, r8, r11      # r11 ← r8 OR NOT r3
```

Opcode:

or	587H	REG
ornot	58BH	REG

See Also: **and**, **andnot**, **nand**, **nor**, **not**, **notand**, **notor**, **xnor**, **xor**

remi, remo

Mnemonic:

remi	Remainder Integer
remo	Remainder Ordinal

Format:

rem*	*src1*,	*src2*,	*dst*
	reg/lit/sfr	reg/lit/sfr	reg/sfr

Description: Divides *src2* by *src1* and stores the remainder in dst. The sign of the result (if nonzero) is the same as the sign of *src2*.

Action:

```
if (src2=0) Arithmetic Zero Divide fault;
dst ← src2 - ((src2 / src1) * src1);
# src1, src2 and dst are 32 bits
```

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr.*
Arithmetic	*Zero Divide.* The *src1* operand is 0
	Integer Overflow. Result is too large for destination register (**remi** only). If overflow occurs and AC.om=1, the fault is suppressed and AC.io is set to 1. The least significant 32 bits of the result are stored in *dst.*

Example:

```
remo r4, r5, r6        # r6 ← r5 rem r4
```

Opcode:

remi	748H	REG
remo	708H	REG

See Also: **modi**

ret

Mnemonic: **ret** Return

Format: **ret**

Description: Returns program control to the calling procedure. The current stack frame (i.e., that of the called procedure) is deallocated and the FP is changed to point to the calling procedure's stack frame. Instruction execution is continued at the instruction pointed to by the RIP in the calling procedure's stack frame, which is the instruction immediately following the call instruction.

As shown in the action statement below, the return-status field and prereturn-trace flag determine the action that the processor takes on the return. These fields are contained in bits 0 through 3 of register r0 of the called procedure's local registers.

Refer to *Chapter 5, Procedure Calls* for further discussion of **ret**.

Action:

```
wait for any uncompleted instructions to finish;
case return_type is

if ((PFP.rt=001₂) or (PFP.rt=111₂))
            {       # return from fault or interrupt handler
            AC ← memory(FP - 12);
            if (PC.em=supervisor) PC ← memory(FP - 16);
            }
else if ((PFP.rt=010₂) or (PFP.rt=011₂))
            {       # return to non-supervisor procedure
            PC.te ← PFP.rt0;
            PC.em ← user;
            }
else if (PFP.rt=000₂)
            {       # return from local
            }
else Operation Unimplemented fault;
FP ← PFP;
# these accesses are cached in the local register cache
r0:15 ← memory(FP);
IP ← RIP;
```

Faults:	Trace	*Instruction. Return. Pre-Return. Breakpoint.* Instruction, Return and Pre-Return Trace Events are signaled after instruction completion. Trace fault is generated if PC.te=1 and TC.i or TC.r or TC.p=1.
	Operation	*Unimplemented.* Execution from on-chip data RAM.
		Unimplemented. Reserved return type encountered.

Example:

```
ret            # program control returns to context of
               # calling procedure
```

Opcode: **ret** 0AH CTRL

See Also: **call**, **calls**, **callx**

rotate

Mnemonic:	**rotate**	Rotate		
Format:	**rotate**	*len,* reg/lit/sfr	*src,* reg/lit/sfr	*dst* reg/sfr

Description:

Copies *src* to *dst* and rotates the bits in the resulting *dst* operand to the left (toward higher significance). (Bits shifted off left end of word are inserted at right end of word.) The *len* operand specifies number of bits that the *dst* operand is rotated. *len* can range from 0 to 31.

This instruction can also be used to rotate bits to the right. Here, the number of bits the word is to be rotated right is subtracted from 32 to get the *len* operand.

Action:

dst ← *src* **rotate_left** (*len* **mod** 32);

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr*.

Example:

```
rotate 13, r8, r12      # r12 ← r8 with bits rotated
                        # 13 bits to left
```

Opcode:

rotate 59DH REG

See Also:

SHIFT, **eshro**

scanbit

Mnemonic: **scanbit** Scan For Bit

Format: **scanbit** *src*, *dst*

scanbit	*src*,	*dst*
	reg/lit/sfr	reg/sfr

Description: Searches *src* value for most-significant set bit (1 bit). If a most significant 1 bit is found, its bit number is stored in *dst* and condition code is set to 010_2. If *src* value is zero, all 1's are stored in *dst* and condition code is set to 000_2.

Action:

```
tempsrc ← src;
if (tempsrc=0)
    {
    dst ← 0xFFFFFFFF;
    AC.cc ← 000₂;
    }
else
    {
    i ← 31;
    while ((tempsrc and 2^i)=0)
    {
    i ← i - 1;
    }
    dst ← i;
    AC.cc ← 010₂;
    }
```

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr.*

Example:

```
# assume g8 is nonzero
scanbit g8, g10     # g10 ← bit number of most-
                    # significant set bit in g8;
                    # AC.cc ← 010₂
```

Opcode: scanbit 641H REG

See Also: spanbit, setbit

scanbyte

Mnemonic:	**scanbyte**	Scan Byte Equal	
Format:	**scanbyte**	*src1*, reg/lit/sfr	*src2* reg/lit/sfr

Description: Performs byte-by-byte comparison of *src1* and *src2* and sets condition code to 010_2 if any two corresponding bytes are equal. If no corresponding bytes are equal, condition code is set to 000_2.

Action:

```
tmpsrc1 ← src1;
tmpsrc2 ← src2;
if      (((tmpsrc1 and 0x000000FF) = (tmpsrc2 and 0x000000FF))
or
         ((tmpsrc1 and 0x0000FF00) = (tmpsrc2 and 0x0000FF00))
or
         ((tmpsrc1 and 0x00FF0000) = (tmpsrc2 and 0x00FF0000))
or
         ((tmpsrc1 and 0xFF000000) = (tmpsrc2 and 0xFF000000)))
         AC.cc ← 010₂;
else AC.cc ← 000₂;
```

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr*.

Example:

```
# assume r9 = 0x11AB1100
scanbyte 0x00AB0011, r9     # AC.cc ← 010₂
```

Opcode: **scanbyte** 5ACH REG

sdma

Mnemonic: **sdma** Setup DMA Channel

Format:

sdma	*src1*,	*src2*,	*src3*
	reg/lit/sfr	reg/lit/sfr	reg/lit

Description:

The DMA channel specified by *src1* is set up using the control word in *src2*. Dedicated data RAM for the specified DMA channel is written with *src3* value. First two bits of *src1* specify channel; *src2* specifies DMA control word as a literal or single 32-bit register; *src3* specifies a single 32-bit register if channel is data-chaining. This register contains the address of the first chaining descriptor in memory. *src3* must specify a register with a register number divisible by four.

If channel is not data chaining, *src3* specifies a triple word contained in registers *src3, src3+1* and *src3+2*. *src3* contains byte count for DMA; *src3+1* contains source address; *src3+2* contains destination address.

Action:

```
dma_control_for_channel[src1 mod 4] ← src2;
if (not chaining mode)
            dma_ram[src1 mod 4] ← src3;     # triple-word store
else        dma_ram[src1 mod 4] ← src3;     # word store
start_dma_channel[src1 mod 4];
```

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Constraint	*Privileged.* Attempt to execute while not in supervisor mode.

Example:

```
ldconst   3,r6;                     # set channel
ldconst   Channel_3_Modes,r7;       # load controls
ldq       Channel_3_transfer, r8;   # load pointers
sdma r6, r7, r8                     # and byte count from memory
                                    # configure dma channel 3
```

Opcode: **sdma** 630H REG

See Also: **udma**

setbit

Mnemonic:	**setbit**	Set Bit		
Format:	**setbit**	*bitpos*, reg/lit/sfr	*src*, reg/lit/sfr	*dst* reg/sfr

Description: Copies *src* value to *dst* with one bit set. *bitpos* specifies bit to be set.

Action: *dst* ← *src* **or** 2^(*bitpos* **mod** 32);

Faults:	Trace	*Instruction. Breakpoint*
	Operation	*Unimplemented.* Execution from on-chip data RAM.
	Type	*Mismatch.* Non-supervisor reference of a *sfr*.
Example:	setbit 15, r9, r1	# r1 ← r9 with bit 15 set
Opcode:	**setbit** 583H	REG

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

SHIFT

Mnemonic:

shlo	Shift Left Ordinal
shro	Shift Right Ordinal
shli	Shift Left Integer
shri	Shift Right Integer
shrdi	Shift Right Dividing Integer

Format:

sh*	*len,*	*src,*	*dst*
	reg/lit/sfr	reg/lit/sfr	reg/sfr

Description:

Shifts *src* left or right by the number of bits indicated with the *len* operand and stores the result in *dst.* Bits shifted beyond register boundary are discarded. For values of *len* greater than 32, the processor interprets the value as 32.

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

shli shifts zeros in from the least significant bit. An overflow fault is generated if the bits shifted out are not the same as the most significant bit (bit 31). If overflow occurs, *dst* will equal *src* shifted left as much as possible without overflowing.

shri performs a conventional arithmetic shift-right operation by shifting in the most significant bit (bit 31). When this instruction is used to divide a negative integer operand by the power of 2, it produces an incorrect quotient (discarding the bits shifted out has the effect of rounding the result toward negative).

shrdi is provided for dividing integers by the power of 2. With this instruction, 1 is added to the result if the bits shifted out are non-zero and the *src* operand was negative, which produces the correct result for negative operands.

shli and **shrdi** are equivalent to **muli** and **divi** by the power of 2.

eshro is provided for extracting a 32-bit value from a long ordinal (i.e., 64 bits), which is contained in two adjacent registers. Refer to *Instruction Set Reference* titled ***eshro*** for details.

Action:

shlo:

```
if (len < 32) dst ← src << len;
else dst ← 0;
```

shro:

```
if (len < 32) dst ← src >> len;
else dst ← 0;
```

shli:

```
if (len > 32) i ← 32;
else i ← len;
temp ← src;
while ((temp.31 = temp.30) and (i ≠ 0))
  {
  temp ← temp << 1;
  i ← i - 1;
  }
dst ← temp;
```

shri:

```
if (len >32) i ← 32;
else i ← len;
temp ← src;
while (i ≠ 0)
  {
  temp ← temp >> 1;             # shift temp right one bit
  temp.bit31 ← temp.bit30;      # extend temp's sign bit
  i ← i - 1;
  }
dst ← temp;
```

shrdi:

```
i ← len;
if (i > 32) i ← 32;
temp ← src;
s_sign ← temp.bit31
lost_bit ← 0;
while (i ≠ 0)
  {
  lost_bit ← lost_bit or temp.bit0;
  temp ← temp >> 1;             # shift temp left one bit
  temp.bit31 ← temp.bit30;      # extend temp's sign bit
  i ← i - 1;
  }
if ((s_sign = 1) and (lost_bit = 1)) temp ← temp + 1;
dst ← temp;
```

Faults:	Trace	*Instruction. Breakpoint.*
	Operation	*Unimplemented.* Execution from on-chip data RAM.
	Type	*Mismatch.* Non-supervisor reference of a *sfr.*

Arithmetic *Integer Overflow.* Result is too large for the destination register (**shli** only). If overflow occurs and AC.om is a 1, the fault is suppressed and AC.io is set to a 1. After an overflow, *dst* will equal *src* shifted left as much as possible without overflowing.

Example: shli 13, g4, r6 # g6 ← g4 shifted left 13 bits

Opcode:

shlo	59CH	REG
shro	598H	REG
shli	59EH	REG
shri	59BH	REG
shrdi	59AH	REG

See Also: **divi, muli, rotate, eshro**

spanbit

Mnemonic: **spanbit** Span Over Bit

Format: **spanbit** *src*, *dst*

spanbit	*src*,	*dst*
	reg/lit/sfr	reg/sfr

Description: Searches *src* value for the most significant clear bit (0 bit). If a most significant 0 bit is found, its bit number is stored in *dst* and condition code is set to 010_2. If *src* value is all 1's, all 1's are stored in *dst* and condition code is set to 000_2.

Action:

```
if (src = 0xFFFFFFFF)
  {
  dst ← 0xFFFFFFFF;
  AC.cc ← 000₂;
  }
else
  {
  i ← 31;
while ((src and 2^i) ≠ 0)
  {
  i ← i - 1;
  }
  dst ← i;
  AC.cc ← 010₂;
  }
```

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr.*

Example:

```
# assume r2 is not 0xffffffff
spanbit r2, r9       # r9 ← bit number of most-significant
                     # clear bit in r2; AC.cc ← 010₂
```

Opcode: **spanbit** 640H REG

See Also: **scanbit**

STORE

Mnemonic:

st	Store
stob	Store Ordinal Byte
stos	Store Ordinal Short
stib	Store Integer Byte
stis	Store Integer Short
stl	Store Long
stt	Store Triple
stq	Store Quad

Format:

st*	*src*,	*efa*
	reg	addr

efa:

(reg)	disp + 8(IP)	disp [reg * scale]
offset	disp	(reg1) [reg2 * scale]
offset (reg)	disp (reg)	disp (reg 1) [reg 2 * scale]

Description:

Copies a byte or group of bytes from a register or group of registers to memory. *src* specifies a register or the first (lowest numbered) register of successive registers.

efa 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 *efa*. (Refer to the section of *Chapter 3* titled *Addressing Modes* for a complete discussion.)

stob and **stib** store a byte and **stos** and **stis** store a half word from the *src* register's low order bytes . Data for ordinal stores is truncated to fit the destination width. If the data for integer stores cannot be represented correctly in the destination width, an Arithmetic Integer Overflow fault is signaled.

st, **stl**, **stt** and **stq** copy 4, 8, 12 and 16 bytes, respectively, from successive registers to memory.

For **stl**, *src* must specify an even numbered register (e.g., g0, g2, ... or r0, r2, ...). For **stt** and **stq**, *src* must specify a register number that is a multiple of four (e.g., g0, g4, g8, ... or r0, r4, r8, ...).

Action:

st:	memory_word (*efa*) ← *src*;
stob:	memory_byte (*efa*) ← *src* truncated to 8 bits;
stib:	memory_byte (*efa*) ← *src* truncated to 8 bits;
stos:	memory_short (*efa*) ← *src* truncated to 16 bits;
stis:	memory_short (*efa*) ← *src* truncated to 16 bits;
stl:	memory_long (*efa*) ← *src*;
stt:	memory_triple (*efa*) ← *src*;
stq:	memory_quad (*efa*) ← *src*;

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
	Unimplemented. An unaligned *efa* was referenced and unaligned support was disabled.
	Operand. Invalid operand value encountered.
	*Opcode.*Invalid opcode encoding encountered.
Arithmetic	*Integer Overflow.* Result is too large for destination (**stib** and **stis** only). If overflow occurs and AC.om=1, the fault is suppressed and AC.io is set to 1. After an overflow, destination contains the least significant n-bits of the store, where n is the transfer width (8 or 16 bits).
Type	*Mismatch.* Non-supervisor attempt to write to internal data RAM.

Example:

```
st g2, 1254 (g6)    # word beginning at offset
                    #1254 + (g6) ← g2
```

Opcode:

st	92H	MEM
stob	82H	MEM
stos	8AH	MEM
stib	C2H	MEM
stis	CAH	MEM
stl	9AH	MEM
stt	A2H	MEM
stq	B2H	MEM

See Also: LOAD, MOVE

subc

Mnemonic:	**subc** Subtract Ordinal With Carry

Format:

subc	*src1*,	*src2*,	*dst*
	reg/lit/sfr	reg/lit/sfr	reg/sfr

Description:

Subtracts *src1* from *src2*, then subtracts **not**(AC.cc1) and stores the result in *dst*. If the ordinal subtraction results in a carry, AC.cc1 is set to 1, otherwise AC.cc1 is set to 0.

This instruction can also be used for integer subtraction. Here, if integer subtraction results in an overflow, condition code bit 0 is set.

subc does not distinguish between ordinals and integers: it sets condition code bits 0 and 1 regardless of data type.

Action:

```
dst ← src2 - src1 - not(AC.cc1);
AC.cc ←  0CV2;
# V is   1 if integer subtraction would have generated an overflow,
#        0 otherwise
# C is   Carry out of the ordinal addition of src2 to not (src1) and
#        carry in.
```

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr*.

Example:

```
subc g5, g6, g7      # g7 ← g6 - g5 - not( Carry Bit)
```

Opcode:

subc	5B2H	REG

See Also: **addc, addi, addo, subi, subo**

subi, subo

Mnemonic:	**subi**	Subtract Integer		
	subo	Subtract Ordinal		
Format:	**sub***	*src1*,	*src2*,	*dst*
		reg/lit/sfr	reg/lit/sfr	reg/sfr

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

Action: *dst* ← *src2* - *src1*;

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr*.
Arithmetic	*Integer Overflow.* Result too large for destination register (**subi** only). Result's least significant 32 bits are stored in *dst*. If overflow occurs and AC.om=1, the fault is suppressed and AC.io is set to a 1. The least significant 32 bits of the result are stored in *dst*.

Example:

```
subi g6, g9, g12      # g12 ← g9 - g6
```

Opcode:

subi	593H	REG
subo	592H	REG

See Also: **addi, addo, subc, addc**

syncf

Mnemonic: **syncf** Synchronize Faults

Format: **syncf**

Description: Waits for all faults to be generated that are associated with any prior uncompleted instructions.

Action:

```
if (AC.nif ≠ 1)
    {
    wait until no imprecise faults can occur associated with
    instructions which have begun, but are not completed.;
    }
```

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.

Example:

```
ld xyz, g6
addi r6, r8, r8
syncf
and g6, 0xFFFF, g8
# the syncf instruction ensures that any faults
# that may occur during the execution of the
# ld and addi instructions occur before the
# and instruction is executed
```

Opcode: **syncf** 66FH REG

See Also: **mark, fmark**

sysctl

Mnemonic: **sysctl** System Control

Format: **sysctl** *src1*, *src2*, *src3*;
reg/lit/sfr reg/lit/sfr reg/lit
message, type

Description: Processor control function specified by the *message* field of *src1* is executed. The *type* field of *src1* is interpreted depending upon the command. Remaining *src1* bits are reserved. The *src2* and *src3* operands are also interpreted depending upon the command.
The *src1* operand is interpreted as follows:

	31 — 16	15 — 8	7 — 0
src1	FIELD 2	MESSAGE TYPE	FIELD 1

The following table lists i960 CA processor commands.

Message	*Src1*			*Src 2*	*Src 3*
	Type	Field 1	Field 2	Field 3	Field 4
Request Interrupt	00H	Vector Number	N/U	N/U	N/U
Invalidate Cache	01H	N/U	N/U	N/U	N/U
Configure Cache	02H	Cache Mode Configuration (see table)	N/U N/U	Cache load address	N/U
Reinitialize	03H	N/U	N/U	1st Inst. address	PRCB address
Load Control Register	04H	Register Group Number	N/U	N/U	N/U

NOTE

Sources and fields which are not used (designated N/U) are ignored.

Cache Mode Configuration Table

Mode Field(1)	Mode Description
000_2	1 Kbyte normal cache enabled
001_2	1 Kbyte cache disabled (execute off-chip)
100_2	Load and lock 1 Kbyte cache (execute off-chip)
110_2	Load and lock 512 bytes, 512 bytes normal cache enabled

NOTE

1) Modes which are not defined are reserved.

Action:

```
temp ← src1;
tmpmessage ← (temp and 0xf0) >> 8;
switch (tmpmessage)
case 0:     # Signal an Interrupt
            post_interrupt(temp and 0xf);
            break;
case 1:     # Invalidate the Instruction Cache
            invalidate_instruction_cache;
            break;
case 2:     # Configure Instruction Cache
            tmptype ← (src1 and 0xff);
            if (tmptype.bit0 = 1) disable_instruction_cache;
            else if (tmptype = 0x0) enable_1k_instruction_cache;
            else if (tmptype = 0x4)
              {     # Load and freeze 1k cache
              instr_cache ← memory_1k(src2); # load 1k bytes
              freeze_1k_instruction_cache;
              }
            else if (tmptype = 0 x 6)
              {     # Load and freeze 512 bytes of cache
              instr_cache ← memory_512(src2) # load 512 bytes
              freeze_512_instruction_cache;
              }
            else    Reserved;
            break;
case 3:     # Software Reset
            temp ← src2;
            load PRCB pointed to by src3;
            IP ← temp;
            break;
```

```
case 4:      # Load One Group of Control Registers
             # from the Control Table
             temp [0-3] ←memory_quad (Control Table Base + group
             offset);
             for (i ←0; i²3, i ←i+1 control_reg[i] ←temp[i];
             break
default:     Operation invalid-operand fault;
```

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
	Unimplemented. Attempted to execute unimplemented command.

Example:

```
ldconst Clear_cache, g6     # set the clear cache message
sysctl r6,r7,r8             # execute cache invalidation
                            # note: r7, r8 are dummies here
be uploaded_code            # branch to code which was uploaded
```

Opcode:

sysctl 659H REG

9

TEST

Mnemonic:

teste{**.t**|**.f**}	Test For Equal
testne{**.t**|**.f**}	Test For Not Equal
testl{**.t**|**.f**}	Test For Less
testle{**.t**|**.f**}	Test For Less Or Equal
testg{**.t**|**.f**}	Test For Greater
testge{**.t**|**.f**}	Test For Greater Or Equal
testo{**.t**|**.f**}	Test For Ordered
testno{**.t**|**.f**}	Test For Not Ordered

Format:

test*{.t|.f} *dst*
reg/sfr

Description:

Stores a true (01H) in *dst* if the logical AND of the condition code and opcode mask-part is not zero. Otherwise, the instruction stores a false (00H) in *dst*. For **testno** (Unordered), a true is stored if the condition code is 000_2, otherwise a false is stored.

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

Instruction	Mask	Condition
testno	000_2	Unordered
testg	001_2	Greater
teste	010_2	Equal
testge	011_2	Greater or equal
testl	100_2	Less
testne	101_2	Not equal
testle	110_2	Less or equal
testo	111_2	Ordered

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 instructions except **testno**:

```
if ((mask and AC.cc) - $000_2$) dst ← 0x1; # dst set for true
else dst ← 0x0; # dst set for false
```

testno:

```
if (AC.cc = $000_2$) dst ← 0x1; #dst set for true
else dst ← 0x0; # dst set for false
```

Faults:

Trace	*Instruction. Breakpoint.*
Operation RAM.	*Unimplemented.* Execution from on-chip data
Type	*Mismatch.* Non-supervisor reference of a *sfr.*

Example:

```
# assume AC.cc = 100₂

testl g9            # g9 ← 0x00000001
```

Opcode:

teste	22H	COBR
testne	25H	COBR
testl	24H	COBR
testle	26H	COBR
testg	21H	COBR
testge	23H	COBR
testo	27H	COBR
testno	20H	COBR

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

udma

Mnemonic:	**udma** Update DMA-Channel RAM
Format:	**udma**
Description:	The current status of the DMA channels is written to the dedicated DMA RAM.
Action:	**for** (i = 0 to 3) dma_ram[i] ← dma_status_channel[i];
Faults:	Trace *Instruction. Breakpoint.*
	Operation *Unimplemented.* Execution from on-chip data RAM.

Example:

```
udma                    # update status to dma ram
ldq Channel_3_ram,r4    # read current pointers
                        # and byte count for dma channel 3
```

Opcode: **udma** 631H REG

See Also: **sdma**

xnor, xor

Mnemonic:	**xnor**	Exclusive Nor		
	xor	Exclusive Or		

Format:

xnor	*src1*,	*src2*,	*dst*
	reg/lit/sfr	reg/lit/sfr	reg/sfr
xor	*src1*,	*src2*,	*dst*
	reg/lit/sfr	reg/lit/sfr	reg/sfr

Description: Performs a bitwise XNOR (**xnor** instruction) or XOR (**xor** instruction) operation on the *src2* and *src1* values and stores the result in *dst*.

Action:

xnor: *dst* ← **not** (*src2* **or** *src1*) **or** (*src2* **and** *src1*);

xor: *dst* ← (*src2* **or** *src1*) **and not** (*src2* **and** *src1*);

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of a *sfr*.

Example:

```
xnor r3, r9, r12      # r12 ← r9 XNOR r3
xor g1, g7, g4        # g4 ← g7 XOR g1
```

Opcode:

xnor	589H	REG
xor	586H	REG

See Also: **and**, **andnot**, **nand**, **nor**, **not**, **notand**, **notor**, **or**, **ornot**

The Bus Controller 10

CHAPTER 10
THE BUS CONTROLLER

This chapter serves as a guide for a software developer when configuring the bus controller. It overviews bus controller capabilities and implementation and describes how to program the bus controller. System designers should reference *Chapter 11, External Bus Description* for a functional description of the bus controller.

OVERVIEW

The bus controller supports a synchronous, 32-bit-wide, demultiplexed external bus which consists of 30 address lines, four byte enables, 32 data lines, a clock output and control and status signals. The bus controller manages instruction fetches, data loads/stores and DMA transfer requests. Bus management is accomplished by queuing bus requests which effectively decouples instruction execution speed from external memory access time.

Load and store instructions — the program's interface to the bus controller — work on ordinal (unsigned) or integer (signed) data. A single load or store instruction can move from 1 to 16 bytes of data. The bus controller also handles instruction fetches, which read either 8 bytes (two words) or 16 bytes (four words).

The bus controller divides the flat 4 Gbyte memory space into 16 regions; each region has independent software programmable parameters that define data bus width, ready control, number of wait states, pipeline read mode, byte ordering and burst mode. These parameters are stored in the memory region configuration table. Each memory region is 2^{28} bytes (256 Mbytes).

The purpose of configurable memory regions is to provide system hardware interface support. Regions are transparent to the software. The address' upper four bits (A31:28) indicate which region is enabled.

A data bus width parameter in the region table configures the external data bus as an 8-, 16- or 32-bit bus for a region. This parameter determines byte enable signal encoding and the physical location of data on data bus pins.

When a burst bus mode is enabled, a single address cycle can be followed with up to four data cycles. This mode enables very high speed data bus transfers. When disabled, accesses appear as one data cycle per address cycle. The burst bus mode can be enabled or disabled on a region-by-region basis.

A programmable wait state generator inserts a programmed number of wait states into any memory access. These wait states, independently programmable by region, can be specified between:

- address and data cycles
- consecutive data cycles of burst accesses
- the last data cycle and the address cycle of the next request

An external, memory-ready input permits the user's hardware to insert wait states into any memory cycle. This pin works with the wait state generator and is enabled or disabled on a region-by-region basis.

Pipelined read mode provides the highest data bandwidth for reads and instruction fetches. When a region is programmed for pipelined reads, the next read's address cycle overlaps the current read's data cycle.

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.

MEMORY REGION CONFIGURATION

Programmable memory region configurations simplify external memory system designs and reduce system parts count. Particular bus access characteristics may be programmed. This programmed bus scheme allows accesses made to different areas (or regions) in memory to have different characteristics. For example, one area in memory can be configured for slow 8-bit accesses; this is optimal for peripherals. Another area in memory can be configured for 32-bit wide burst accesses; this is optimal for fast DRAM interfaces. Bus function in each region is determined by the memory region configuration. The following bus characteristics are selected for each region:

- Selectable 8-, 16- or 32-bit-wide data bus
- Five wait state parameters
- Programmable pipelined reads
- Programmable high performance burst access
- Memory-ready and burst cycle terminate for dynamic access control
- Big or little endian byte order

These bus characteristic can be programmed independently for accesses made to each of 16 different regions in memory. The value of the memory address upper four bits (A31:28) determine the selected region. Memory region configuration affects all accesses to the addressed memory region. Loads, stores, DMA transfers and instruction fetches all use the parameters defined for the region.

Programming region characteristics is accomplished by setting values in the memory region configuration registers. A separate register allows the user to program the characteristics for each of the 16 memory regions. Memory region configuration registers are described in this chapter's section titled *Programming the Bus Controller.* The following subsections describe the i960 CA processor's programmable bus characteristics.

Data Bus Width

Each region's data bus width is programmed in the memory region configuration table. The i960 CA processor allows 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 CA processor uses the lower order data lines when reading and writing to 8- or 16-bit memory.

Burst and Pipelined Read Accesses

To improve bus bandwidth, the i960 CA device provides a burst access and pipelined read access. These burst and pipelining modes are separately enabled or disabled for each memory region by programming the memory region configuration table.

When burst access is enabled, the bus controller generates an address — the burst address — followed by one to four data transfers. The lower address bits are incremented for each consecutive data transfer. Burst accesses facilitate the interface to fast page mode DRAM; 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. This provides an optimal wait state profile for fast page mode DRAM.

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 invisible for back-to-back read accesses.

Wait States

A wait state generator within the bus controller generates wait states for a memory access. For many memory interfaces, the internal wait state generator eliminates the necessity to externally generate a memory ready signal to indicate a valid data transfer.

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 **R**ead **A**ddress-to-**D**ata. 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 **R**ead **D**ata-to-**D**ata. 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 **W**rite **A**ddress-to-**D**ata. 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 **W**rite **D**ata-to-**D**ata. 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 **X** (read or write) **D**ata-to-**A**ddress. The minimum number of wait states between the last data cycle of a bus request to the address cycle of the next bus request. N_{XDA} applies to read and write requests. Programmable for 0-3 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 not used in non-burst memory regions.

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

For pipelined read accesses, the bus controller uses a value of 0 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.

The ready ($\overline{\text{READY}}$) and burst terminate ($\overline{\text{BTERM}}$) inputs dynamically control bus accesses. These inputs are enabled or disabled for each memory region. $\overline{\text{READY}}$ extends accesses by forcing wait states. $\overline{\text{BTERM}}$ allows a burst access to be broken into multiple accesses, with no lost data. The memory region registers are programmed to enable or disable these inputs for each region.

$\overline{\text{READY}}$ and $\overline{\text{BTERM}}$ work with the programmed internal wait state counter. If $\overline{\text{READY}}$ and $\overline{\text{BTERM}}$ 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 *Chapter 11, External Bus Description* for details on the operation of the $\overline{\text{READY}}$ and $\overline{\text{BTERM}}$ inputs.

NOTE

$\overline{\text{READY}}$ and $\overline{\text{BTERM}}$ must be disabled in regions where pipelined reads are enabled.

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 memory region by setting a bit in the region table entry for the region. The bus controller supports big endian and little endian byte ordering for memory operations:

little endian ordering	The controller reads or writes a data word's least-significant byte to the bus' eight least-significant data lines (D0-D7). 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 ordering	The controller reads or writes a data word's least-significant byte to the bus' eight most-significant data lines (D31-D24). 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.

PROGRAMMING THE BUS CONTROLLER

The bus controller is programmed using 17 control registers; 16 of which make up the region table, the remaining one is the Bus Configuration (BCON) Register. Control registers are automatically loaded at initialization from the control table in external memory. Control registers are modified by using the load control registers message of the system control (**sysctl**) instruction. See *Chapter 2, Programming Environment* for control register definition.

Region Table (MCON0-MCON15)

The region table contains 16 entries. Each entry is stored in a control register and specifies:

- number of wait states
- burst mode
- data bus width
- pipeline mode
- byte ordering
- external ready mode for the region that it controls

An address' four most-significant bits indicate which region is being accessed. A region table entry is 32 bits wide (see Figures 10.1 and 10.2); however, not all bits are currently used. Table 10.1 defines the region table's programmable bits.

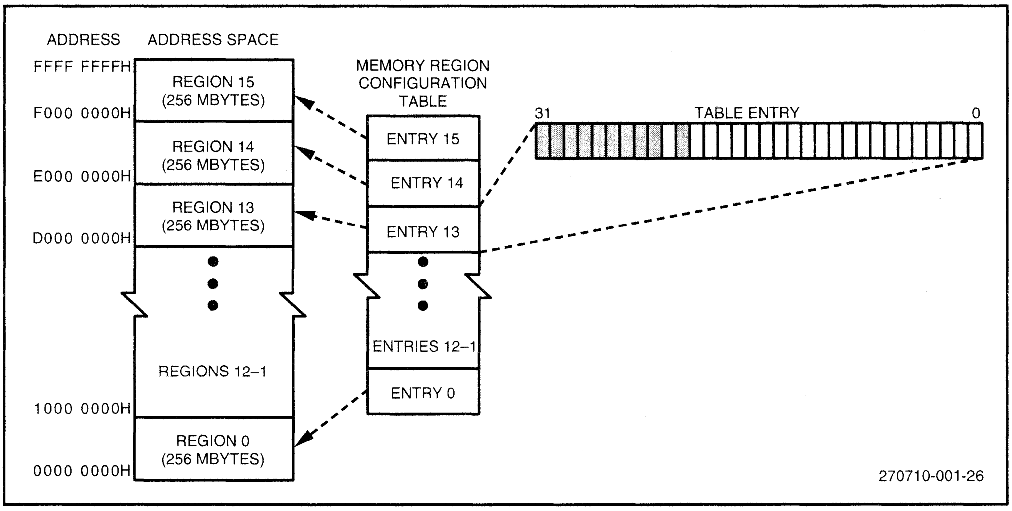


Figure 10.1. Region Table Configures External Memory

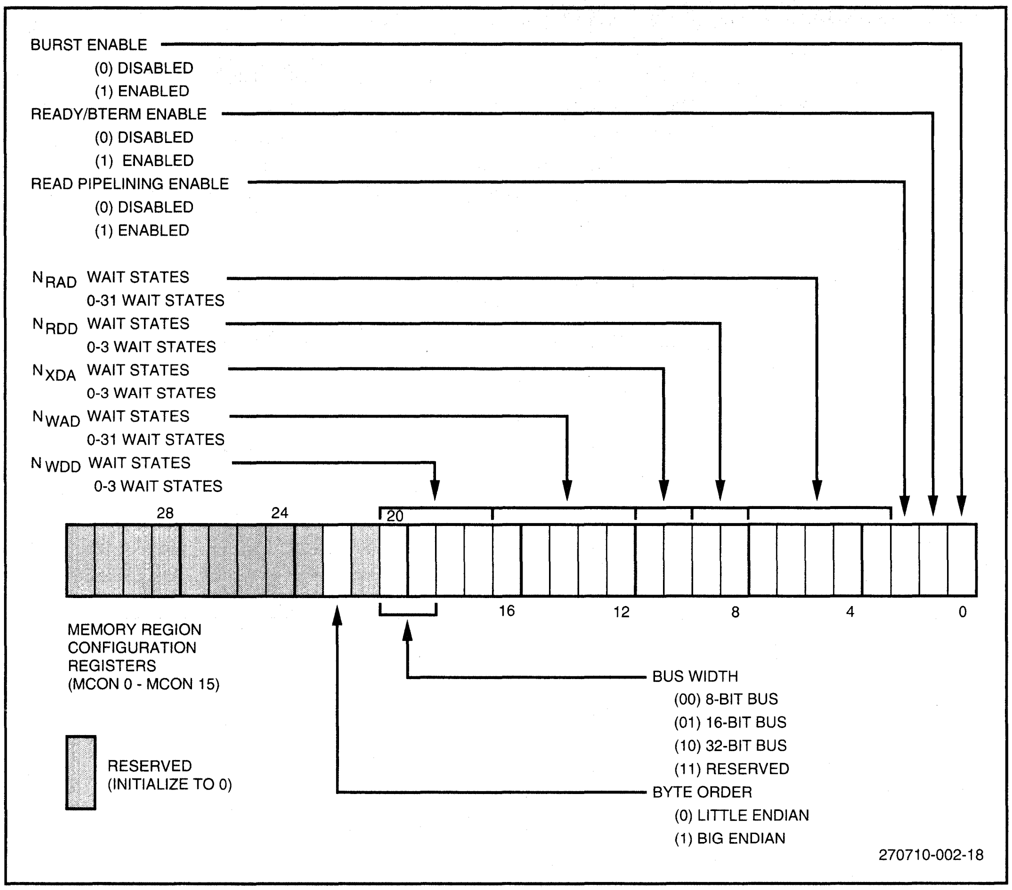


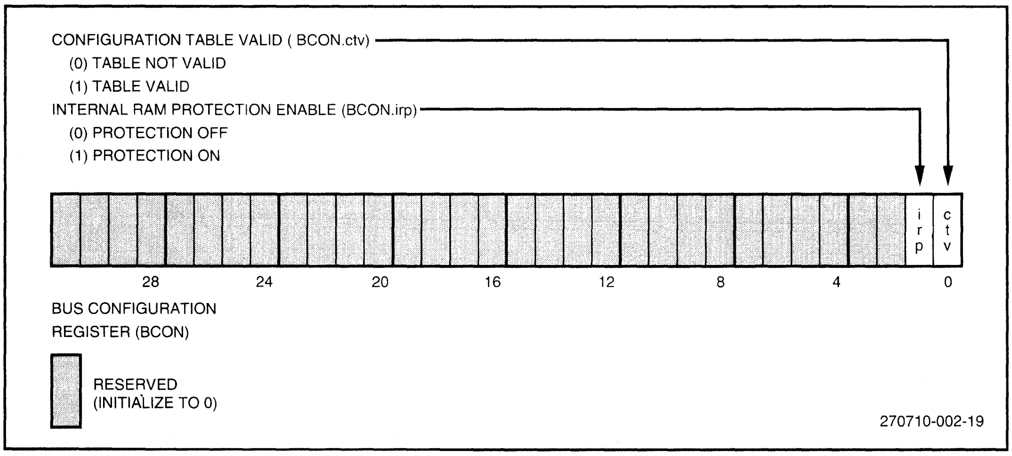
Figure 10.2. Memory Region Configuration Register (MCON0-MCON15)

Bus Configuration Register (BCON)

The Bus Configuration Register (BCON), shown in Figure 10.3, is a 32-bit register that controls the region configuration table and internal data RAM protection. Table 10.2 defines the BCON Register's programmable bits.

Table 10.1. Region Table Bit Definitions

Entry Name	Bit #	Definition
Burst Enable	0	Enables or disables burst accesses for the region.
READY/BTERM Enable	1	Enables or disables region's $\overline{READY}$ and $\overline{BTERM}$ inputs. If disabled, $\overline{READY}$ and $\overline{BTERM}$ are ignored.
Read Pipelining Enable	2	Enables or disables address pipelining of region's read accesses. $\overline{READY}$ and $\overline{BTERM}$ are ignored during pipelined reads.
N_{RAD} Wait States	3-7	Number of Read Address-to-Data wait states in the region. (Programmed for 0-31 Wait States)
N_{RDD} Wait States	8-9	Number of Read Data-to-Data wait states in the region. (Programmed for 0-3 Wait States)
N_{XDA} Wait States	10-11	Number of X (read or write) Data-to-Address wait states in the region. (Programmed for 0-3 Wait States). N_{XDA} wait states are only inserted at the end of a bus request.
N_{WAD} Wait States	12-16	Number of Write Address-to-Data wait states in the region. (Programmed for 0-31 Wait States)
N_{WDD} Wait States	17-18	Number of Write Data-to-Data wait states in the region. (Programmed for 0-3 Wait States)
Bus Width	19-20	Determines region's data bus width. Effects encoding of byte-enable signals ($\overline{BE3:0}$)
Byte Ordering	22	Selects region's byte ordering: little endian or big endian.



CONFIGURATION TABLE VALID (BCON.ctv)
(0) TABLE NOT VALID
(1) TABLE VALID
INTERNAL RAM PROTECTION ENABLE (BCON.irp)
(0) PROTECTION OFF
(1) PROTECTION ON
irp
ctv
28
24
20
16
12
8
4
0
BUS CONFIGURATION REGISTER (BCON)
RESERVED (INITIALIZE TO 0)
270710-002-19

Figure 10.3. Bus Configuration Register (BCON)

Table 10.2. BCON Register Bit Definitions

Entry Name	Bit #	Definition
Configuration Table Valid	0	When BCON.ctv bit is clear, all memory is accessed as defined by Region Table Entry 0. When BCON.ctv bit is set, the entire region table is used.
Internal RAM Protection	1	Enables supervisor write protection for internal data RAM at address 100H to 3FFH.

Configuring the Bus Controller

The bus controller is configured automatically when the processor initializes. All region table values are loaded from the control table and the BCON.ctv bit is set (table valid) before the first instruction of application code executes. The user only has to supply the correct value in the control table in external memory. See *Chapter 14, Initialization and System Requirements* for more details on the processor's actions at initialization.

The region table value may be altered after initialization by use of the **sysctl** instruction. It is important to avoid altering an enabled region table entry while a bus access to that region is in progress. It is acceptable, however, to write the same data to an enabled region table entry while a bus access to that region is in progress. This consideration is especially important for Region Table Entry 0, when it is the master entry (BCON.ctv = 0).

DATA ALIGNMENT

Aligned bus requests generate an address that occurs on a data type's natural boundary. Quad words and triple words are aligned on 16-byte boundaries; double words on 8-byte boundaries; words on 4-byte boundaries; short words (half words) on 2-byte boundaries; bytes on 1-byte boundaries.

Unaligned bus requests do not occur on these natural boundaries. Any unaligned bus request to a little endian memory region is executed; however, unaligned requests to big endian regions are supported only if software adheres to particular address alignment restrictions.

The processor handles all unaligned bus requests to little endian memory regions. The processor executes unaligned little endian requests as several aligned requests. This method of handling an unaligned bus request results in some performance loss compared to aligned requests: microcode uses CPU cycles to generate aligned requests and more bus cycles are used to transfer unaligned data.

The processor may generate an operation-unaligned fault when any unaligned request is encountered. This fault can be masked with the PRCB fault configuration word. Refer to *Chapter 14, Initialization and System Requirements* for Fault Configuration Word discussion.

When the processor encounters an unaligned request, microcode breaks the unaligned request into a series of aligned requests. For example, if a read request is issued to read a little endian word from address XXXXXXX1H (unaligned), a byte request followed by a short request followed by a byte request is executed. Figure 10.4 shows how aligned and unaligned bus transfers are carried out for memory regions that use little endian byte ordering.

If the unaligned fault is not masked, the bus controller executes the unaligned access — the same as it does when the fault is masked — and signals an operation-unaligned fault. The unaligned access fault can be used as a debug feature. Removing unaligned memory accesses from an application increases performance.

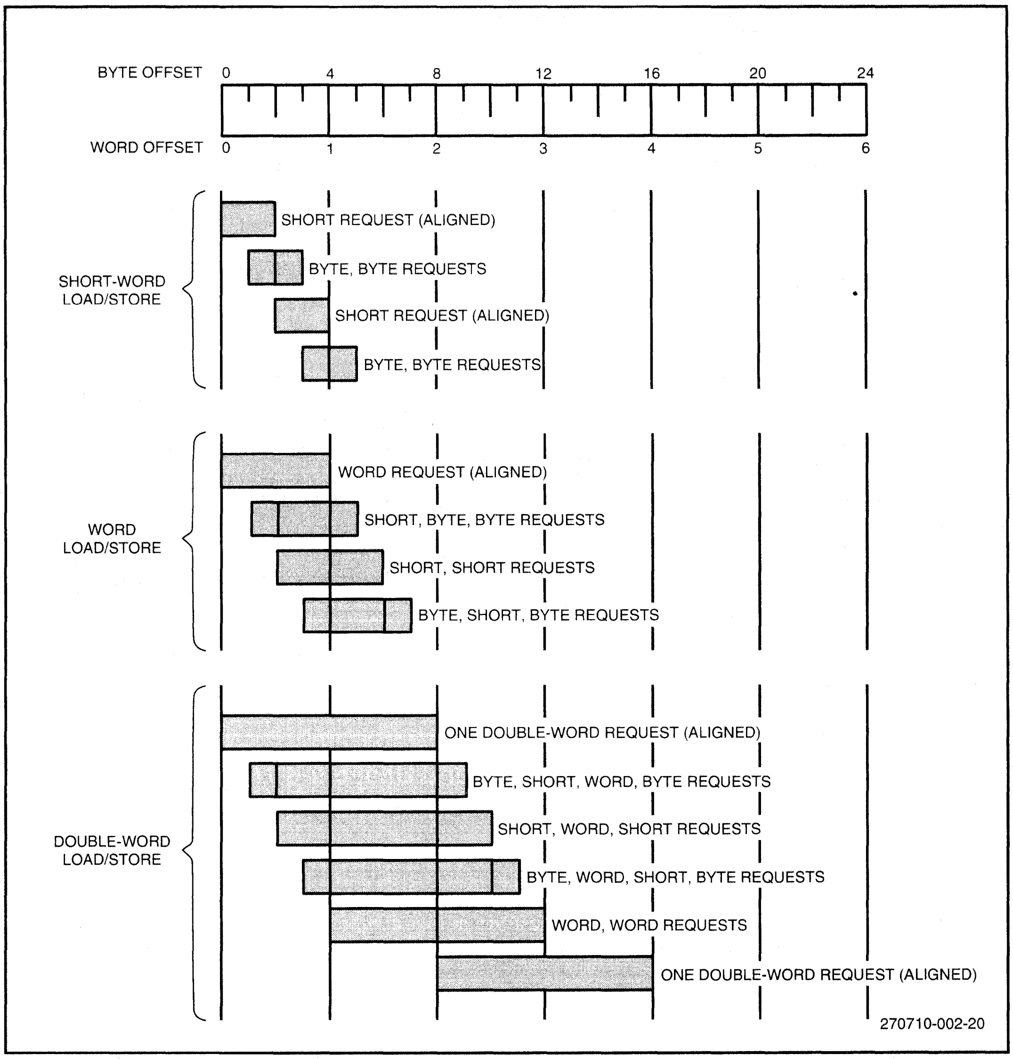

BYTE OFFSET 0 4 8 12 16 20 24
WORD OFFSET 0 1 2 3 4 5 6
SHORT-WORD LOAD/STORE
SHORT REQUEST (ALIGNED)
BYTE, BYTE REQUESTS
SHORT REQUEST (ALIGNED)
BYTE, BYTE REQUESTS
WORD LOAD/STORE
WORD REQUEST (ALIGNED)
SHORT, BYTE, BYTE REQUESTS
SHORT, SHORT REQUESTS
BYTE, SHORT, BYTE REQUESTS
DOUBLE-WORD LOAD/STORE
ONE DOUBLE-WORD REQUEST (ALIGNED)
BYTE, SHORT, WORD, BYTE REQUESTS
SHORT, WORD, SHORT REQUESTS
BYTE, WORD, SHORT, BYTE REQUESTS
WORD, WORD REQUESTS
ONE DOUBLE-WORD REQUEST (ALIGNED)
270710-002-20

Figure 10.4. Summary of Aligned and Unaligned Transfers for Little Endian Regions

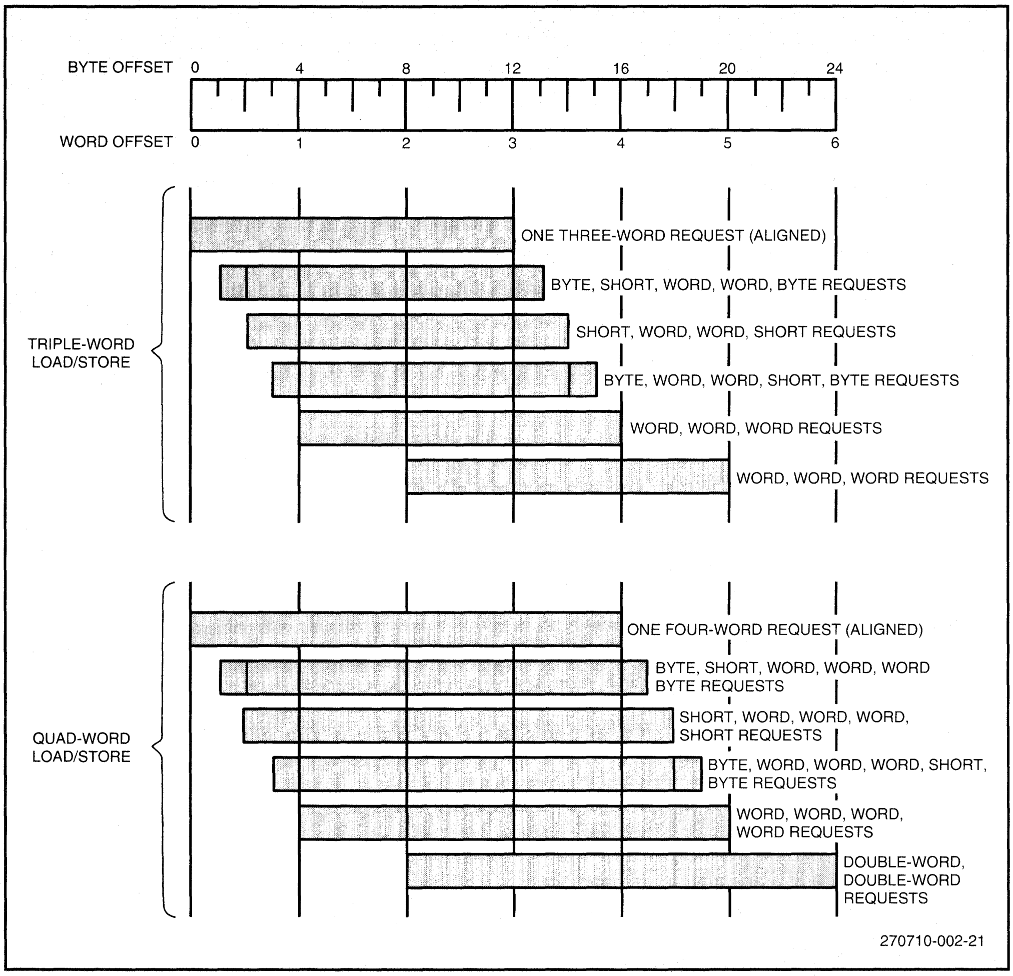

BYTE OFFSET 0 4 8 12 16 20 24
WORD OFFSET 0 1 2 3 4 5 6
TRIPLE-WORD LOAD/STORE
ONE THREE-WORD REQUEST (ALIGNED)
BYTE, SHORT, WORD, WORD, BYTE REQUESTS
SHORT, WORD, WORD, SHORT REQUESTS
BYTE, WORD, WORD, SHORT, BYTE REQUESTS
WORD, WORD, WORD REQUESTS
WORD, WORD, WORD REQUESTS
QUAD-WORD LOAD/STORE
ONE FOUR-WORD REQUEST (ALIGNED)
BYTE, SHORT, WORD, WORD, WORD BYTE REQUESTS
SHORT, WORD, WORD, WORD, SHORT REQUESTS
BYTE, WORD, WORD, WORD, SHORT, BYTE REQUESTS
WORD, WORD, WORD, WORD REQUESTS
DOUBLE-WORD, DOUBLE-WORD REQUESTS
270710-002-21

Figure 10.4. Summary of Aligned and Unaligned Transfers for Little Endian Regions (continued)

NOTE

When an unsupported unaligned bus request to a big endian region is attempted, the bus controller handles the transfer exactly the same as it does for little endian regions; that is, it treats the data as little endian data. Thus, the data is not stored coherently in memory.

INTERNAL DATA RAM

The i960 CA microprocessor contains one Kbyte of user-visible internal data RAM which is mapped into the first 1K of the address space (addresses 00H - 3FFH). Internal data RAM is accessed only by loads, stores or DMA transfers. Instruction fetches directed to these addresses cause an operation-unimplemented fault to occur.

A portion of this internal data RAM is optionally used to store DMA status, cached interrupt vectors and, in some applications, cached local registers. The remaining data RAM can be used by application software. Internal data RAM is described in *Chapter 2, Programming Environment.*

Internal data RAM interfaces directly to an internal 128-bit bus. This bus is the pathway between registers and data RAM. Because of the wide internal path, a quad word read or write is usually performed in a single clock.

BUS CONTROLLER IMPLEMENTATION

The bus controller consists of four units (see Figure 10.5):

- queue
- translation unit
- packing unit
- sequencer

The i960 CA processor's instruction fetch unit, execution unit and DMA unit all pass memory requests to the bus controller unit which arbitrates, queues and executes these requests.

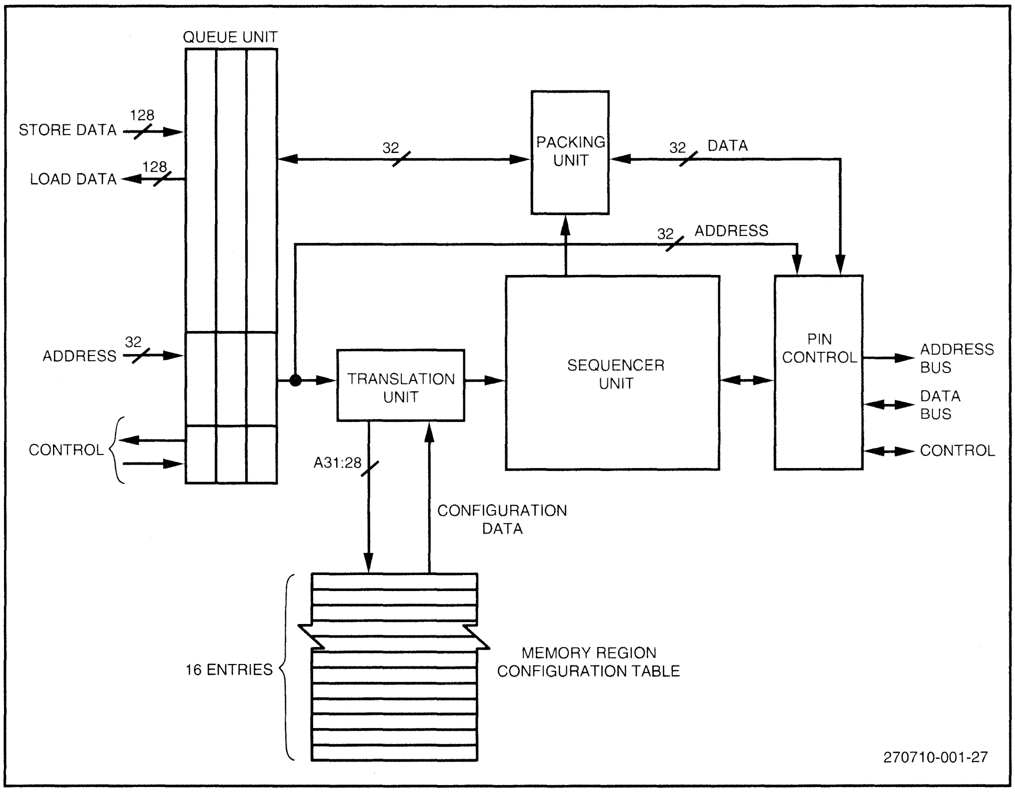


Figure 10.5. Bus Controller Block Diagram

Bus Queue

The bus controller has a queue which contains entries for up to three bus requests. Each queue entry consists of a 32-bit address, up to 128-bits of data (four words) and control information. The bus queue decouples high bandwidth (128-bit-wide data) internal data buses from the lower bandwidth (32-bit-wide data) external bus.

Two of these queue entries are reserved for bus requests generated from user code. The third queue entry is used by the DMA controller. If no DMA channels are set up, the third slot is also used by user code. User requests are serviced in a first-in, first-out (FIFO) manner. The DMA does not issue back-to-back requests; therefore, the CPU is guaranteed access to the external bus between DMA accesses, thus allowing the user and DMA processes to execute concurrently while sharing the external bus.

Queue depth affects bus request and interrupt latency. Queued requests must be serviced before the pending request can be serviced. If an interrupt occurs when all three bus queue entries are full, the three outstanding requests must be serviced before the first interrupt instruction may be fetched from memory.

Data Packing Unit

The data packing unit handles data movement between queues and external bus. It controls data alignment and data packing:

- Data is unpacked when data store request width exceeds physical bus width
- Data is packed when data load request width exceeds physical bus width

If a word load is issued to an 8-bit bus, the bus controller issues four 1-byte reads and the data packing unit assembles incoming data into a single word. If a quad word-store is issued to an 8-bit bus, the bus controller issues four one-word reads and the data packing unit unpacks the outgoing data.

Bus Translation Unit and Sequencer

The bus translation unit is responsible for looking up the memory configuration in the region table. The look-up is based on the bus request's address. The bus request and region table data are passed to the bus sequencer when the external bus is available. The sequencer then breaks the request into a set of bus accesses; this generates the signals on the external bus pins.

External Bus Description 11

CHAPTER 11
EXTERNAL BUS DESCRIPTION

This chapter discusses 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 CA processor. For further details on external bus operation, refer to *Appendix B, Bus Interface Examples*. For information on bus controller configuration, refer to *Chapter 10, Bus Controller*.

OVERVIEW

The i960 CA processor's integrated bus controller and external bus provide a flexible, easy-to-use interface to memory and peripherals. All bus transactions are synchronized with the processor clock outputs (PCLK2:1); therefore, most memory system control logic can easily be implemented as state machines. The internal, programmable wait state generator, external ready control signals, bus arbitration signals, data transceiver control signals and programmable bus width parameters all combine to reduce system component count and ease the design task.

Terminology: Requests and Accesses

The terms *request* and *access* are used frequently when referring to bus controller operation. The description of the i960 CA processor's bus modes and burst bus operation is simplified by defining these terms:

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 prefetch is issued, the core delivers a bus request to the bus controller unit.

The bus controller unit independently processes the request and retrieves data from memory for load instructions and instruction prefetches. 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 to the bus controller a bus request of the appropriate data length: for example, **ldq** requests that four words of data be retrieved from memory; **stob** requests that a single byte is delivered to memory. The processor fetches instructions using double or quad word bus requests. The processor's microcode issues load and store requests to perform DMA transfers.

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 $\overline{\text{ADS}}$ (address strobe) and $\overline{\text{BLAST}}$ (burst last) signals, which are outputs from the processor. $\overline{\text{ADS}}$ indicates that a valid memory address is present and an access has started. $\overline{\text{BLAST}}$ indicates that the next data which is 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 $\overline{\text{ADS}}$ followed by two to four data transfers. The last data transfer is indicated by assertion of $\overline{\text{BLAST}}$. Non-burst accesses begin with assertion of $\overline{\text{ADS}}$ 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 $\overline{\text{ADS}}$ and a valid address during the last data transfer of the previous cycle. Pipelined accesses may also be burst or non-burst.

Load, store and prefetch mechanisms which deliver "bus requests" to the bus controller are discussed in *Chapter 4, Instruction Set Summary* and *Appendix A, Optimizing Code for the i960 CA Microprocessor.* The bus controller can be configured for various modes to optimize interfaces to external memory. Access type — burst, non-burst or pipelined — is selected when the bus controller is configured.

The bus controller can be configured in various ways. Bus width and access type can be set based on external memory system requirements. For example, peripheral devices commonly have slow, non-burst, 8-bit buses. The bus controller can be configured to make memory accesses to these 8-bit non-burst devices. Each memory access to the peripheral begins with assertion of $\overline{\text{ADS}}$ and a valid address. $\overline{\text{BLAST}}$ is asserted and, after the desired number of wait states, eight bits of data are transferred.

A peripheral device is accessed as described above regardless of which bus request type is issued. For example, if a program includes a **ld** (word load instruction) from the peripheral, the load is executed as four 8-bit accesses to the peripheral.

BUS OPERATION

The i960 CA processor bus consists of 30 address signals, four byte enables, 32 data lines and various control and status signals. Some signals are referred to as *status* signals. A status signal is valid for the duration of a bus request. Other signals are referred to as *control* signals. Control signals are used to define and manage a bus request. This chapter defines the bus pins and pin function.

Table 11.1. Bus Controller Pins

Pin Name	Description	Input/Output
PCLK2:1	Processor Output Clocks	O
D31:0	Data Bus	I/O
A31:2	Address Bus	O
Control Signals:		
$\overline{\text{BE3:0}}$	Byte Enables	O
$\overline{\text{ADS}}$	Address Strobe	O
$\overline{\text{WAIT}}$	Wait States	O
$\overline{\text{BLAST}}$	Burst Last	O
$\overline{\text{READY}}$	Memory Ready	I
$\overline{\text{BTERM}}$	Burst Terminate	I
$\overline{\text{DEN}}$	Data Enable	O
Status Signals:		
W/$\overline{\text{R}}$	Write/Read	O
DT/$\overline{\text{R}}$	Data Transmit/Receive	O
D/$\overline{\text{C}}$	Data/Code Request	O
$\overline{\text{DMA}}$	DMA Request	O
$\overline{\text{SUP}}$	Supervisor Mode Request	O
Bus Arbitration:		
HOLD	Hold Request	I
HOLDA	Hold Acknowledge	O
$\overline{\text{LOCK}}$	Locked Request	O
BREQ	Bus Request Pending	O
$\overline{\text{BOFF}}$	Bus Backoff	I

A bus access starts with an address cycle; address cycle is defined by the assertion of address strobe ($\overline{\text{ADS}}$). Address and byte enables (A31:2 and $\overline{\text{BE3:0}}$) are also presented in the address cycle.

After the address cycle, extra clock cycles called *wait states* may be inserted to accommodate the access time for external memory or peripherals. For write accesses, the data lines are driven during wait states. For read accesses, data lines float. Wait states are discussed later in this chapter in the section titled *Wait States*.

A *data cycle* follows wait states. For write accesses, the data cycle is the last clock cycle in which valid data is driven onto the data bus. For read accesses, external memory must present valid data on the rising edge of PCLK2:1 during the data cycle. Setup and hold time for input data is specified in the *i960 CA Microprocessor Data Sheet*.

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. $\overline{\text{BLAST}}$ — the burst last signal — is asserted to indicate the last data cycle of an access. *Chapter 10, Bus Controller* 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 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. *Chapter 10, Bus Controller* explains how to configure the bus for pipelined accesses.

W/$\overline{\text{R}}$ is a status signal which discerns between a write request (store) or a read request (load or prefetch).

DT/$\overline{\text{R}}$ and $\overline{\text{DEN}}$ 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/$\overline{\text{R}}$ is used to control transceiver direction; the signal is low for read requests and high for write requests. DT/$\overline{\text{R}}$ is valid on the falling PCLK2:1 edge during the address cycle. $\overline{\text{DEN}}$ is used to enable the transceivers; it is asserted on the rising PCLK2:1 edge following the address cycle. DT/$\overline{\text{R}}$ and $\overline{\text{DEN}}$ timings ensure that DT/$\overline{\text{R}}$ does not change when $\overline{\text{DEN}}$ is asserted.

D/$\overline{\text{C}}$, $\overline{\text{DMA}}$ and $\overline{\text{SUP}}$ provide information about the source of bus request. D/$\overline{\text{C}}$ indicates that the current request is data or a code fetch. $\overline{\text{DMA}}$ indicates that the current request is a DMA access. $\overline{\text{SUP}}$ indicates that the current request was originated by a supervisor mode process. When used with a logic analyzer, these signals aid in software debugging.

D/$\overline{\text{C}}$ may also be used to implement separate external data and instruction memories. $\overline{\text{SUP}}$ can be used to protect hardware from accesses while the processor is not in user mode.

The bus is in the idle state between bus requests. Idle bus state begins after N_{XDA} cycles and ends when $\overline{\text{ADS}}$ is asserted.

The bus controller aligns all bus accesses; non-aligned accesses are translated into a series of smaller-aligned accesses. Alignment is described in *Chapter 10, Bus Controller.*

Wait States

In non-burst mode, it is possible to insert wait states between the address and data cycle. In a burst mode access, it is possible to insert wait states between the address cycle and data cycle and between subsequent data cycles for a burst access. It is also possible to insert wait states between bus requests which occur back-to-back.

The i960 CA bus controller provides an internal counter for automatically inserting wait states. The bus controller provides control of five different wait state parameters. Figure 11.1 and the following text describe each parameter.

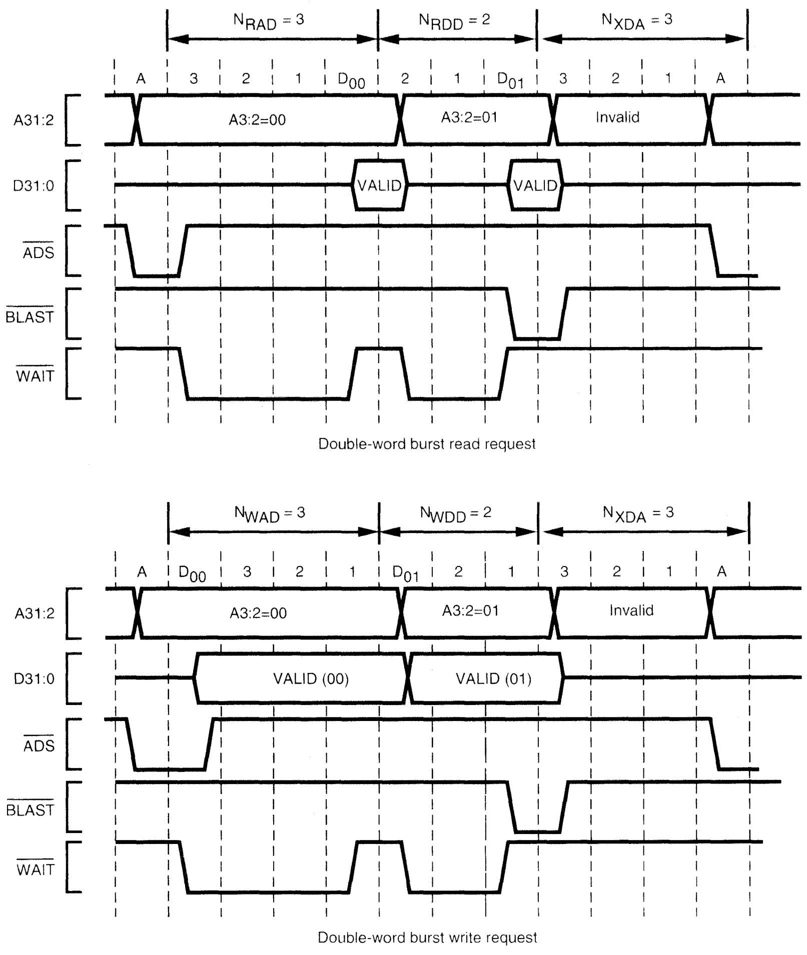


Figure 11.1. Internal Programmable Wait States

N_{RAD}	Number of wait cycles for **R**ead **A**ddress-to-**D**ata. The number of wait states between the address cycle and first read data cycle. N_{RAD} can be programmed for 0-31 wait states.
N_{RDD}	Number of wait cycles for **R**ead **D**ata-to-**D**ata. The number of wait states between consecutive data cycles of a burst read. N_{RDD} can be programmed for 0-3 wait states.
N_{WAD}	Number of wait cycles for **W**rite **A**ddress-to-**D**ata. The number of wait states that data is held after the address cycle and before the first write data cycle. N_{WAD} can be programmed for 0-31 wait states.

$\mathbf{N_{WDD}}$	Number of wait cycles for **W**rite **D**ata-to-**D**ata. The number of wait states that data is held between consecutive data cycles of a burst write. N_{WDD} can be programmed for 0-3 wait states.
$\mathbf{N_{XDA}}$	Number of wait cycles for **X** (read or write) **D**ata to **A**ddress. The minimum number of wait states between the last data cycle of a bus request to the address cycle of the next bus request. N_{XDA} applies to read and write requests. N_{XDA} can be programmed for 0-3 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 not used in non-burst memory regions.

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 request (read deasserted to data-float) determines the number of N_{XDA} cycles.

NOTE

N_{XDA} states are only inserted after the last data transfer of a bus request. Therefore, for requests composed of multiple accesses, N_{XDA} states do not appear between each access. For example, on an 8-bit burst bus, N_{XDA} states are inserted only after the fourth byte of a word request rather than after every byte. See Figure 11.2.

For pipelined read accesses, the bus controller uses a value of zero for the N_{XDA} parameter, regardless of the programmed value for the parameter. 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.

The processor asserts the $\overline{\text{WAIT}}$ signal when N_{RAD}, N_{WAD}, N_{RDD} or N_{WDD} are inserted. $\overline{\text{WAIT}}$ can be used as a read or write strobe for the external memory system.

Wait states can also be controlled with $\overline{\text{READY}}$ and $\overline{\text{BTERM}}$. These inputs are enabled or disabled in a region by programming the memory region configuration table. Refer to *Chapter 10, Bus Controller* for details on setting up bus controller for wait states.

When enabled, $\overline{\text{READY}}$ indicates to the processor that read data on the bus is valid or a write data transfer has completed. The $\overline{\text{READY}}$ pin value is ignored until the N_{RAD}, N_{RDD}, N_{WAD} or N_{WDD} wait states expire. At this time, if $\overline{\text{READY}}$ is deasserted (high), wait states continue to be inserted until $\overline{\text{READY}}$ is asserted (low).

N_{XDA} wait states cannot be extended by $\overline{\text{READY}}$. The $\overline{\text{READY}}$ input is ignored during the idle cycles, the address cycle and N_{XDA} cycles. $\overline{\text{READY}}$ is also ignored in memory regions where pipelining is enabled, regardless of memory region programming.

NOTE

For proper bus operation, the $\overline{\text{READY}}/\overline{\text{BTERM}}$ inputs should be disabled in regions that have pipelining enabled.

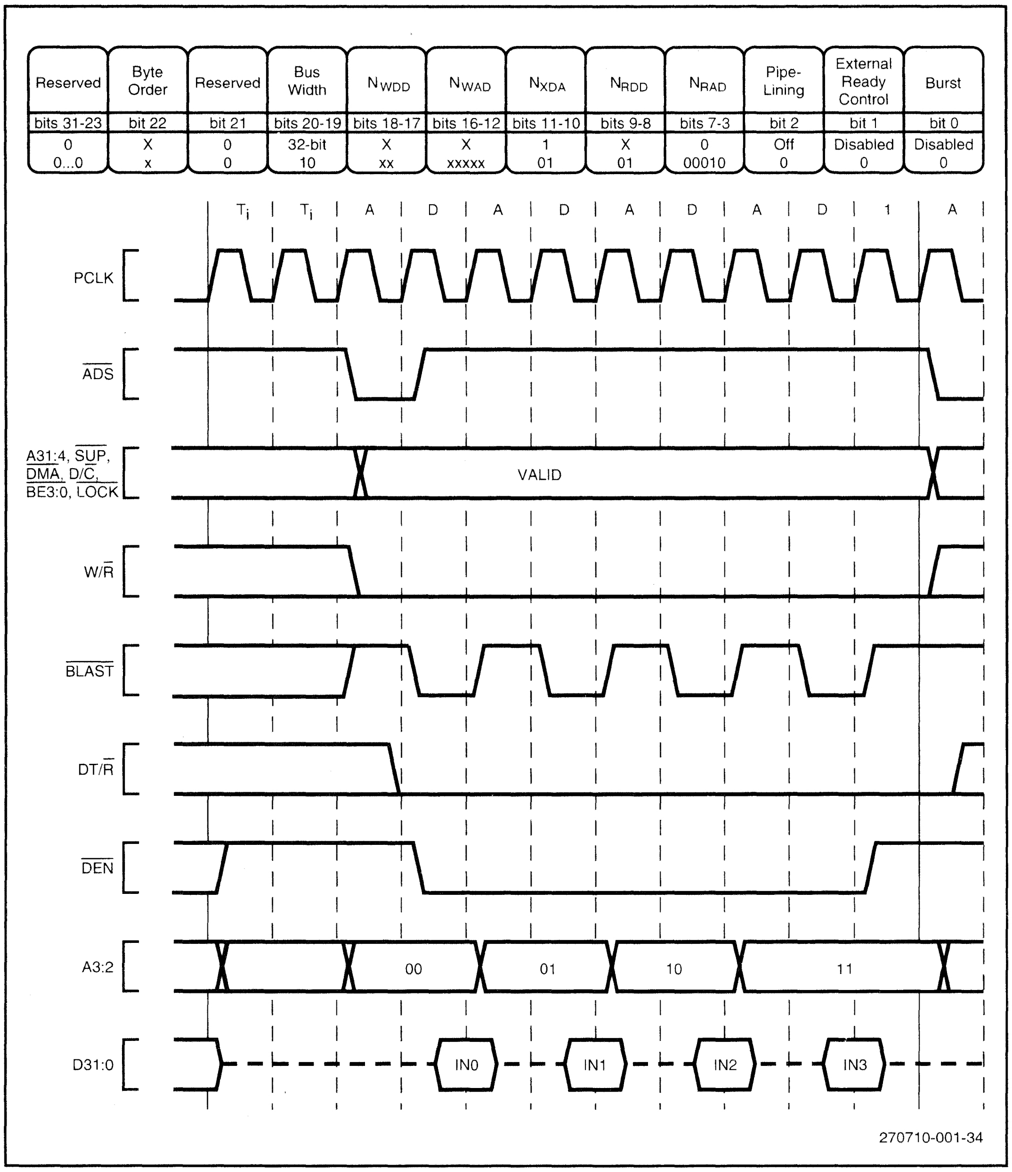
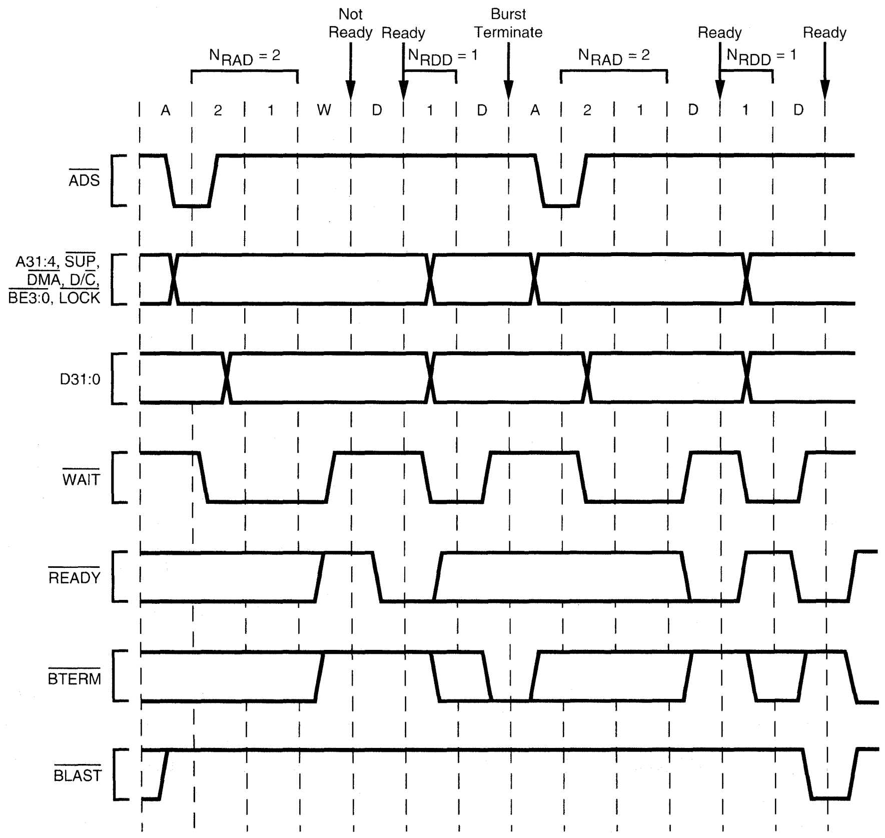


Figure 11.2. Quad-word Read from 32-bit Non-burst Memory

The burst terminate signal (BTERM) breaks up a burst access. Asserting BTERM (low) 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 (A3:2 = 01_2). The bus controller then bursts the remaining three words. The $\overline{\text{BLAST}}$ (burst last) signal indicates the last data transfer of the access.

Read data is accepted on the clock edge that asserts $\overline{\text{BTERM}}$; write data is assumed written. $\overline{\text{BTERM}}$ effectively overrides the memory ready ($\overline{\text{READY}}$) signal when it is asserted. In this way, no data is lost when the current access is terminated. When $\overline{\text{BTERM}}$ is asserted, $\overline{\text{READY}}$ is ignored until after the address cycle which resumes the burst. As with $\overline{\text{READY}}$, $\overline{\text{BTERM}}$ is ignored when pipelining is enabled in a region, regardless of how the region is programmed.

Reserved	Byte Order	Reserved	Bus Width	N_{WDD}	N_{WAD}	N_{XDA}	N_{RDD}	N_{RAD}	Pipe-Lining	External Ready Control	Burst
bits 31-23	bit 22	bit 21	bits 20-19	bits 18-17	bits 16-12	bits 11-10	bits 9-8	bits 7-3	bit 2	bit 1	bit 0
0	X	0	32-bit	X	X	1	1	2	OFF	1	Enabled
0...0	x	0	10	XX	XXXXX	01	01	00010	0	01	1



Not Ready
Ready
Burst Terminate
Ready
Ready
N_{RAD} = 2
N_{RDD} = 1
N_{RAD} = 2
N_{RDD} = 1
A
2
1
W
D
1
D
A
2
1
D
1
D
$\overline{\text{ADS}}$
A31:4, $\overline{\text{SUP}}$, $\overline{\text{DMA}}$, D/$\overline{\text{C}}$, $\overline{\text{BE3:0}}$, $\overline{\text{LOCK}}$
D31:0
$\overline{\text{WAIT}}$
$\overline{\text{READY}}$
$\overline{\text{BTERM}}$
$\overline{\text{BLAST}}$

270710-002-17

Figure 11.3. Bus Request with $\overline{\text{READY}}$ and $\overline{\text{BTERM}}$ Control

Bus Width

Each region's data bus width is programmed in the memory region configuration table. The i960 CA device allows an 8-, 16- or 32-bit-wide data bus for each region. The i960 CA processor places 8- and 16-bit data on low order data pins. This simplifies interface to external devices. As shown in Figure 11.4, 8-bit data is placed on lines D7:0; 16-bit data is placed on lines D15:0; 32-bit data is placed on lines D31:0.

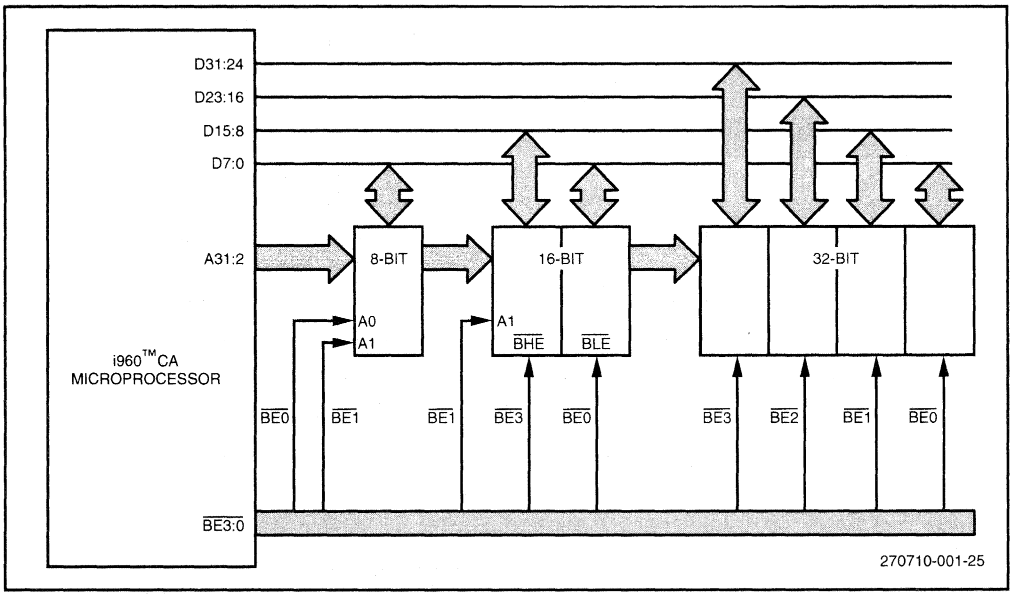


Figure 11.4. Data Width and Byte Enable Encodings

The four byte enable signals are encoded in each region to generate proper address signals for 8-, 16- or 32-bit memory systems:

- 8-bit region: $\overline{BE0}$ is address line A0; $\overline{BE1}$ is address line A1.
- 16-bit region: $\overline{BE1}$ is address line A1; $\overline{BE3}$ is the byte high enable signal ($\overline{BHE}$); $\overline{BE0}$ is the byte low enable signal ($\overline{BLE}$).
- 32-bit region: byte enables are not encoded. Byte enables $\overline{BE3:0}$ select byte 3 to byte 0, respectively. Address lines A31:2 provide the most significant portion of the address. (See Table 11.2.)

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 D7:0, D15:8, D23:16 and D31:24 simultaneously. When storing a value to memory in a 16-bit bus region, the processor drives the same short-word data onto lines D15:0 and D31:16 simultaneously.

Table 11.2. Byte Enable Encoding

8-Bit Bus Width:

BYTE	$\overline{\text{BE3}}$ (X)	$\overline{\text{BE2}}$ (X)	$\overline{\text{BE1}}$ (A1)	$\overline{\text{BE0}}$ (A0)
0	X	X	0	0
1	X	X	0	1
2	X	X	1	0
3	X	X	1	1

16-Bit Bus Width:

BYTE	$\overline{\text{BE3}}$ ($\overline{\text{BHE}}$)	$\overline{\text{BE2}}$ (X)	$\overline{\text{BE1}}$ (A1)	$\overline{\text{BE0}}$ ($\overline{\text{BLE}}$)
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

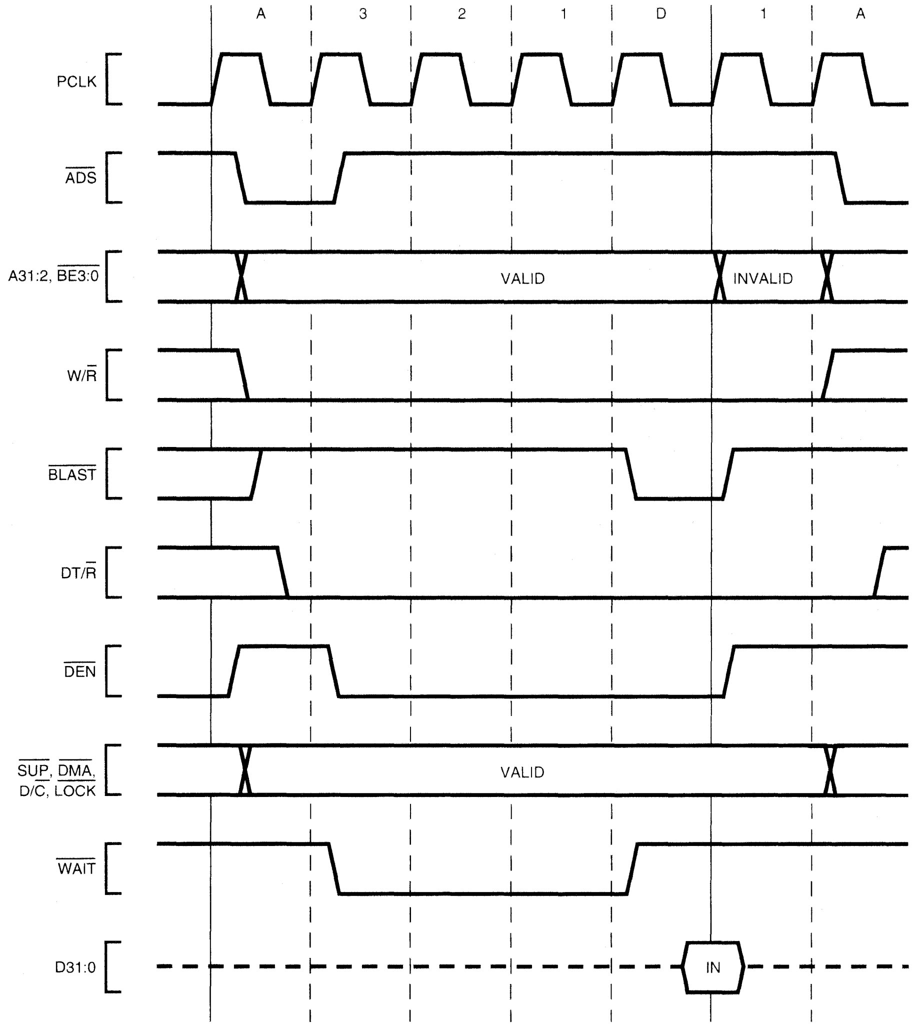
32-Bit Bus Width:

BYTE	$\overline{\text{BE3}}$	$\overline{\text{BE2}}$	$\overline{\text{BE1}}$	$\overline{\text{BE0}}$
0,1,2,3	0	0	0	0
2,3	0	0	1	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

Non-Burst Requests

A basic request (non-burst, non-pipelined; see Figure 11.5) is an address cycle followed by a single data cycle, including any optional wait states associated with the request. Wait states may be generated internally by the wait state generator or externally using the i960 CA processor's $\overline{\text{READY}}$ input.

Reserved	Byte Order	Reserved	Bus Width	N_{WDD}	N_{WAD}	N_{XDA}	N_{RDD}	N_{RAD}	Pipe-Lining	External Ready Control	Burst
bits 31-23	bit 22	bit 21	bits 20-19	bits 18-17	bits 16-12	bits 11-10	bits 9-8	bits 7-3	bit 2	bit 1	bit 0
0	X	0	X	X	X	1	X	3	Off	Disabled	Disabled
0...0	x	0	xx	xx	xxxxx	01	xx	00011	0	0	0



270710-001-28

Figure 11.5. Basic Read Request, Non-Pipelined, Non-Burst, Wait-States

Non-burst accesses and non-pipelined reads are the most basic form of memory access. Non-burst regions may be used to memory map peripherals and memory that cannot support burst accesses. Ready control may be enabled or disabled for the region.

N_{RAD}, N_{WAD} and N_{XDA} wait state fields of a region table entry control basic accesses:

- N_{RAD} specifies the number of wait states between address and data cycles for read accesses.
- N_{WAD} specifies the number of wait states between address and data cycle for write accesses.
- N_{XDA} specifies the number of wait states between data cycle and next address cycle.

Data-to-data wait states (N_{RDD}, N_{WDD}) are not used if burst accesses are not enabled.

A read access begins by asserting the proper address and status signals ($\overline{\text{ADS}}$, A31:2, $\overline{\text{BE0}}$, $\overline{\text{BE3}}$, $\overline{\text{SUP}}$, D/$\overline{\text{C}}$, $\overline{\text{DMA}}$, W/$\overline{\text{R}}$) on the rising clock edge that begins the address cycle (marked as "A" on the figures). Assertion of $\overline{\text{ADS}}$ indicates the beginning of an access.

DT/$\overline{\text{R}}$ is driven on the clock's next falling edge. This signal is asserted early to ensure that DT/$\overline{\text{R}}$ does not change while $\overline{\text{DEN}}$ is asserted. $\overline{\text{DEN}}$ is asserted on the clock's next rising edge (the rising edge in which $\overline{\text{ADS}}$ is deasserted and the address cycle ends). $\overline{\text{DEN}}$ can be used to control external data transceivers.

The cycles that follow are N_{RAD} wait states. $\overline{\text{WAIT}}$ is asserted while the internal wait state generator is counting. If $\overline{\text{READY/BTERM}}$ control is enabled in this region and $\overline{\text{READY}}$ is not asserted after the wait state generator has finished counting, wait states continue to be inserted until $\overline{\text{READY}}$ is asserted.

$\overline{\text{BLAST}}$ assertion indicates end of data transfer cycles for this access. $\overline{\text{DEN}}$ is deasserted. N_{XDA} wait states (turnaround wait states) follow $\overline{\text{BLAST}}$; a new address cycle may start after N_{XDA} cycles expire. N_{XDA} states allow time for slow devices to get off the bus. For this figure, this access is the last access of a bus request because N_{XDA} wait states are inserted and $\overline{\text{DEN}}$ is deasserted.

Burst Accesses

A burst access is an address cycle followed by two to four data cycles. The two least-significant address signals automatically increment during a burst access.

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 11.3.)

Reserved	Byte Order	Reserved	Bus Width	N_{WDD}	N_{WAD}	N_{XDA}	N_{RDD}	N_{RAD}	Pipe-Lining	External Ready Control	Burst
bits 31-23	bit 22	bit 21	bits 20-19	bits 18-17	bits 16-12	bits 11-10	bits 9-8	bits 7-3	bit 2	bit 1	bit 0
0	X	0	X	X	0	0	X	0	Off	Disabled	Disabled
0...0	x	0	xx	xx	00000	00	xx	00000	0	0	0

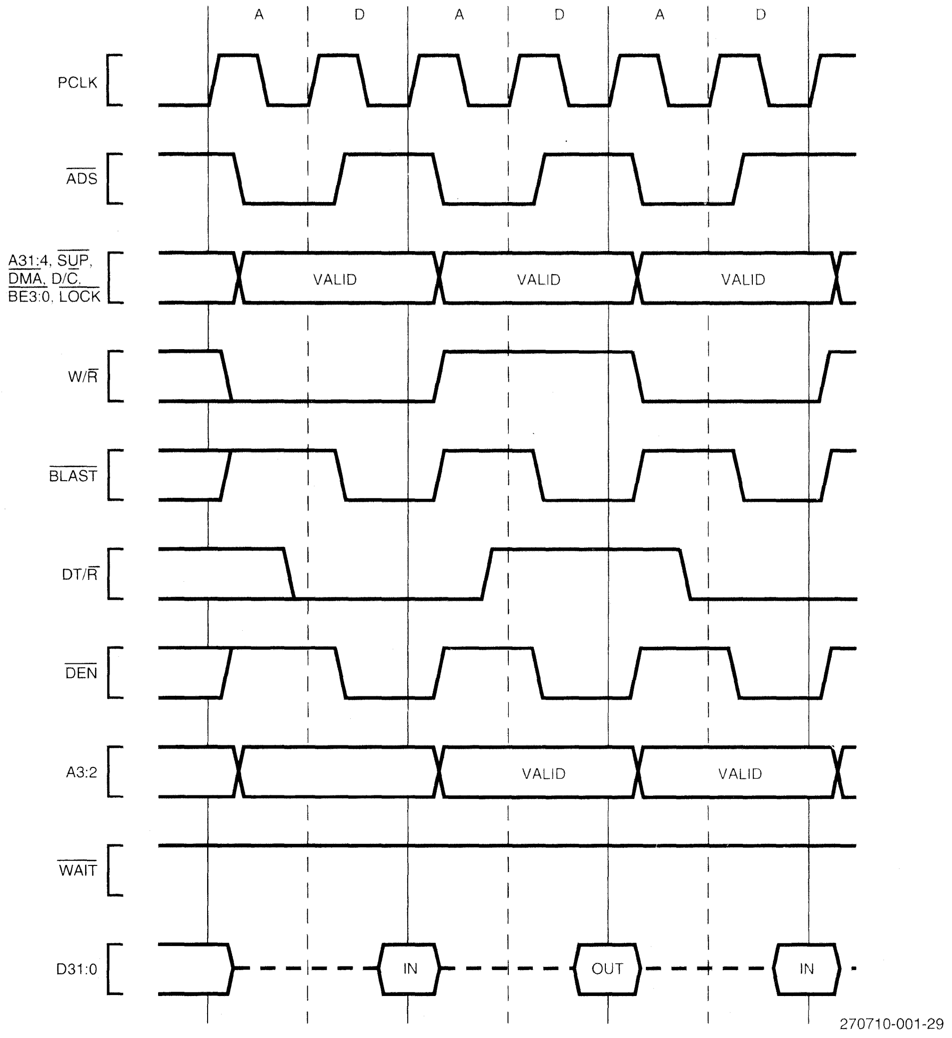


Figure 11.6. Basic Read and Write Requests, Non-Pipelined, Non-Burst, No Wait States

Table 11.3. Burst Transfers and Bus Widths

Request	Bus Width	Number of Burst Accesses	Number of Transfers/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	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

Burst accesses increase bus bandwidth over non-burst accesses. The i960 CA processor 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 to 31 address-to-data wait states (N_{RAD} or N_{WAD}) and one to four data cycles, separated by zero to three data-to-data wait states (N_{RDD} or N_{WDD}). If $\overline{\text{READY}}/\overline{\text{BTERM}}$ control is enabled in the region, N_{RAD}, N_{WAD}, N_{RDD} and N_{WDD} wait states may all be extended by not asserting $\overline{\text{READY}}$. $\overline{\text{BTERM}}$ may be used to break a burst access into smaller accesses.

The address' two least-significant bits automatically increment after each burst data cycle. This is true for 8-, 16- and 32-bit-wide data buses. When a memory region is configured for a 32-bit data bus width, address pins A2 and A3 increment. For a 16-bit memory region, $\overline{\text{BE1}}$ is encoded as A1 and address pins A2 and A1 increment. When a memory region is configured for an 8-bit data bus width, $\overline{\text{BE0}}$ and $\overline{\text{BE1}}$ — acting as the lower two bits of the address — increment.

Maximum burst size is four data transfers per access. For an 8- or 16-bit bus, this means that some bus requests may result in multiple burst accesses. For example, a quad-word (16 byte) request to an 8 bit memory results in four 4-byte burst accesses. Each burst access is limited to four bytewide data transfers.

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 (A3:2=00); double-word transfers always begin on double-word boundaries (A2=0); single-word transfers occur on single word boundaries. (See Figure 11.7.)

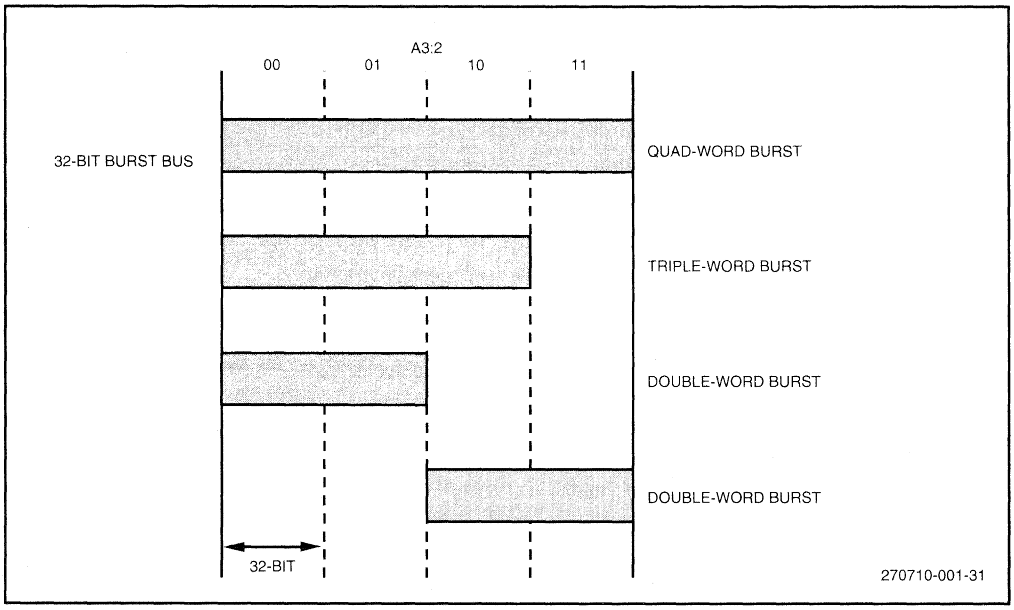


Figure 11.7. 32-Bit-Wide Data Bus Bursts

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 an even short-word boundary (A1=0). Single short-word transfers occur on single short-word boundaries (see Figure 11.8.) For a 16-bit bus, data is transferred on data pins D15:0. Data is also driven on upper data lines D31:16.

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 11.9). For an 8-bit bus, data is transferred on data pins D7:0. Data is also driven on the upper bytes of the data bus D15:8, D23:16 and D31:24.

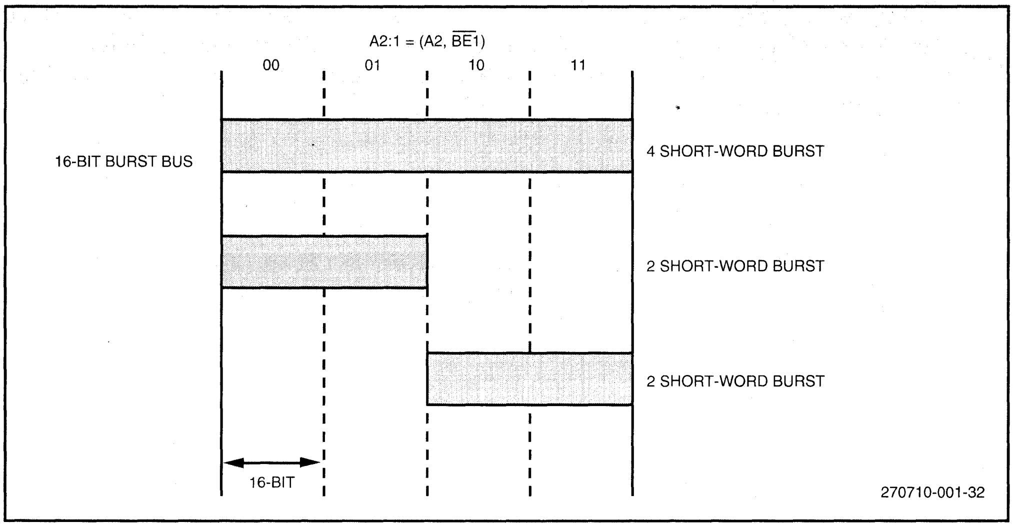


Figure 11.8. 16-Bit Wide Data Bus Bursts

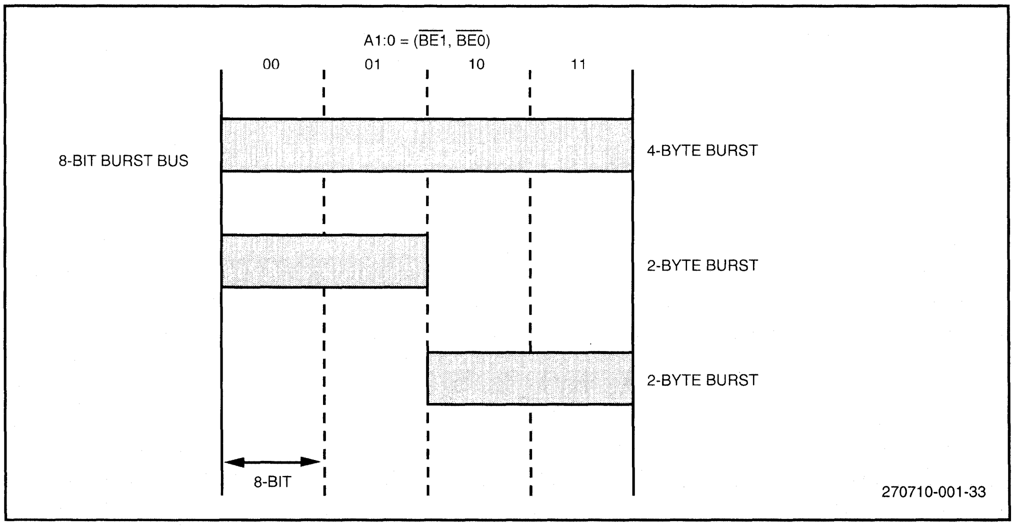


Figure 11.9. 8-Bit Wide Data Bus Bursts

Figure 11.10 shows a quad-word read on a 32-bit bus. Burst access begins by asserting the proper address and status signals ($\overline{ADS}$, A31:2, $\overline{BE3:0}$, $\overline{SUP}$, D/$\overline{C}$, $\overline{DMA}$, W/$\overline{R}$). This is done on the rising edge that begins the address cycle ("A" on the figures). Word read asserts all byte enable signals $\overline{BE3:0}$. $\overline{ADS}$ assertion indicates beginning of access.

DT/$\overline{R}$ is driven on the clock's next falling edge to ensure that DT/$\overline{R}$ does not change while $\overline{DEN}$ is asserted. $\overline{DEN}$ is asserted on the clock's next rising edge — the rising edge that ends address cycle. $\overline{ADS}$ is deasserted on this clock edge. $\overline{DEN}$ is used to control external data transceivers. $\overline{DEN}$ and DT/$\overline{R}$ remain asserted throughout the burst access.

Wait-state cycles that follow are address and N_{RAD} wait states. $\overline{WAIT}$ is asserted while the internal wait-state generator is counting. If $\overline{READY}/\overline{BTERM}$ control is enabled in this region and $\overline{READY}$ and $\overline{BTERM}$ are not asserted after the wait-state generator has finished counting, wait states continue to be inserted until $\overline{READY}$ is asserted. If $\overline{BTERM}$ is asserted, $\overline{READY}$ is ignored. Data is then read and a new address cycle is generated. (See section titled *Ready and Burst Terminate Control* later in this chapter.)

The data cycle is followed by N_{RDD} wait states. These wait states separate burst data cycles and can be used to extend data access time of reads and data setup and hold times for writes.

$\overline{BLAST}$ assertion indicates the end of data transfer cycles for this access. At this time, $\overline{DEN}$ is deasserted.

N_{XDA} wait states (turnaround wait states) are inserted after the last access of a bus request. N_{XDA} wait states follow $\overline{BLAST}$ only when $\overline{BLAST}$ is asserted for the last access of a bus request. A new address cycle may start after N_{XDA} cycles have expired. N_{XDA} states allow slow devices to get off the bus.

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.

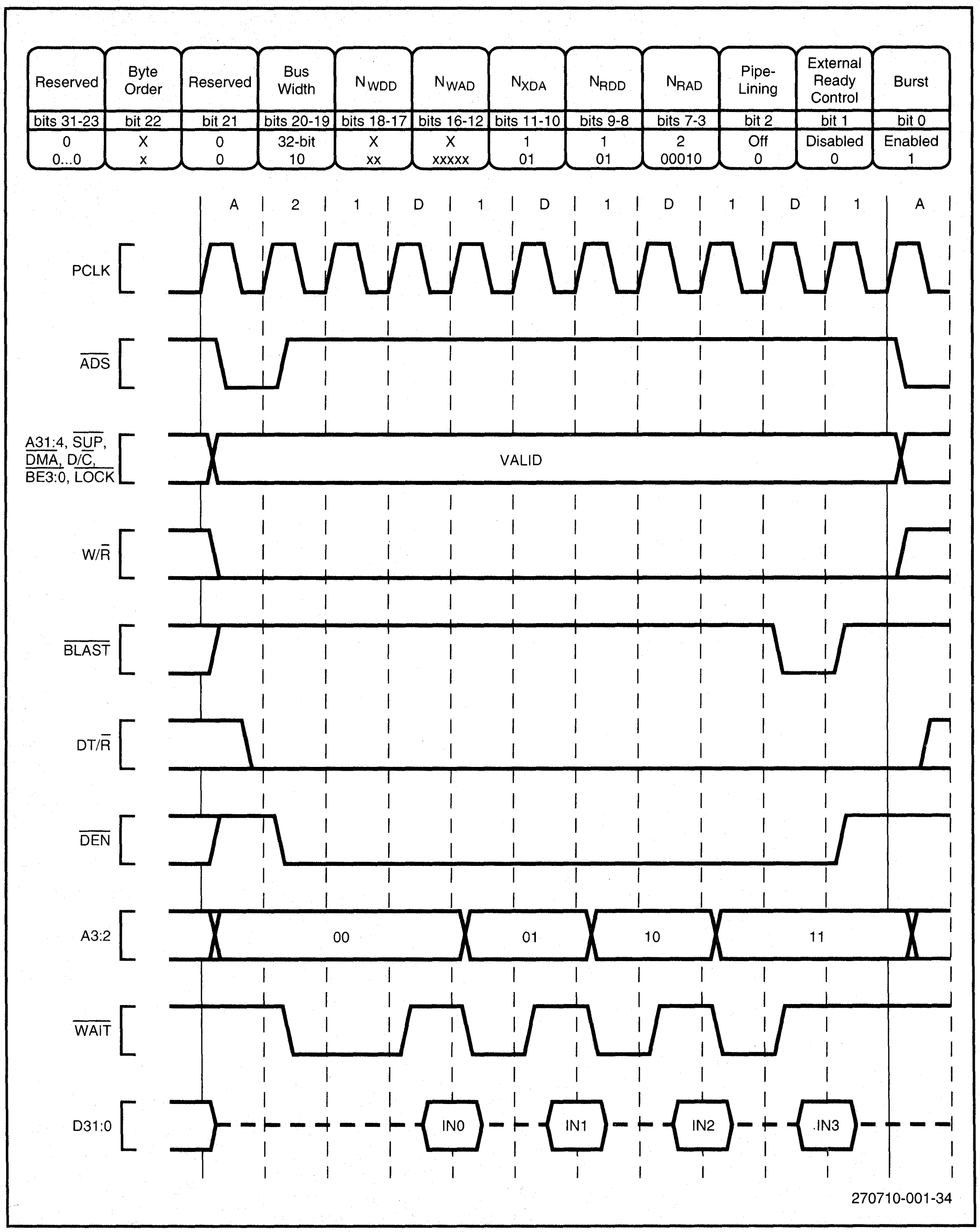
A pipelined read memory system is implemented by adding an address latch to the design (see Figure 11.12). The address latch holds the address for the current read access while the processor outputs the address for the next access. This allows the next address to be available during the data cycle of the current access. Overlapping address and data cycles improves data bandwidth.

Write accesses to a pipelined region act the same as writes to a non-pipelined region. This means that the address for a write access is not pipelined. Similarly, the address for a read access following a write is not pipelined.

NOTE

When pipelining is enabled in a region, the $\overline{READY}$ and $\overline{BTERM}$ inputs are ignored for read and write cycles. These inputs must be disabled in regions that have pipelining enabled.

For pipelined read accesses, the bus controller uses a value of zero for the N_{XDA} parameter, regardless of the programmed value for the parameter. 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.



Reserved	Byte Order	Reserved	Bus Width	N_{WDD}	N_{WAD}	N_{XDA}	N_{RDD}	N_{RAD}	Pipe-Lining	External Ready Control	Burst
bits 31-23	bit 22	bit 21	bits 20-19	bits 18-17	bits 16-12	bits 11-10	bits 9-8	bits 7-3	bit 2	bit 1	bit 0
0	X	0	32-bit	X	X	1	1	2	Off	Disabled	Enabled
0...0	x	0	10	xx	xxxxx	01	01	00010	0	0	1

Figure 11.10. 32-Bit Bus, Burst, Non-Pipelined, Read Request with Wait States

Reserved	Byte Order	Reserved	Bus Width	N_{WDD}	N_{WAD}	N_{XDA}	N_{RDD}	N_{RAD}	Pipe-Lining	External Ready Control	Burst
bits 31-23	bit 22	bit 21	bits 20-19	bits 18-17	bits 16-12	bits 11-10	bits 9-8	bits 7-3	bit 2	bit 1	bit 0
0	X	0	32-bit	1	3	1	X	X	Off	Disabled	Enabled
0...0	x	0	10	01	00011	01	xx	xxxxx	0	0	1

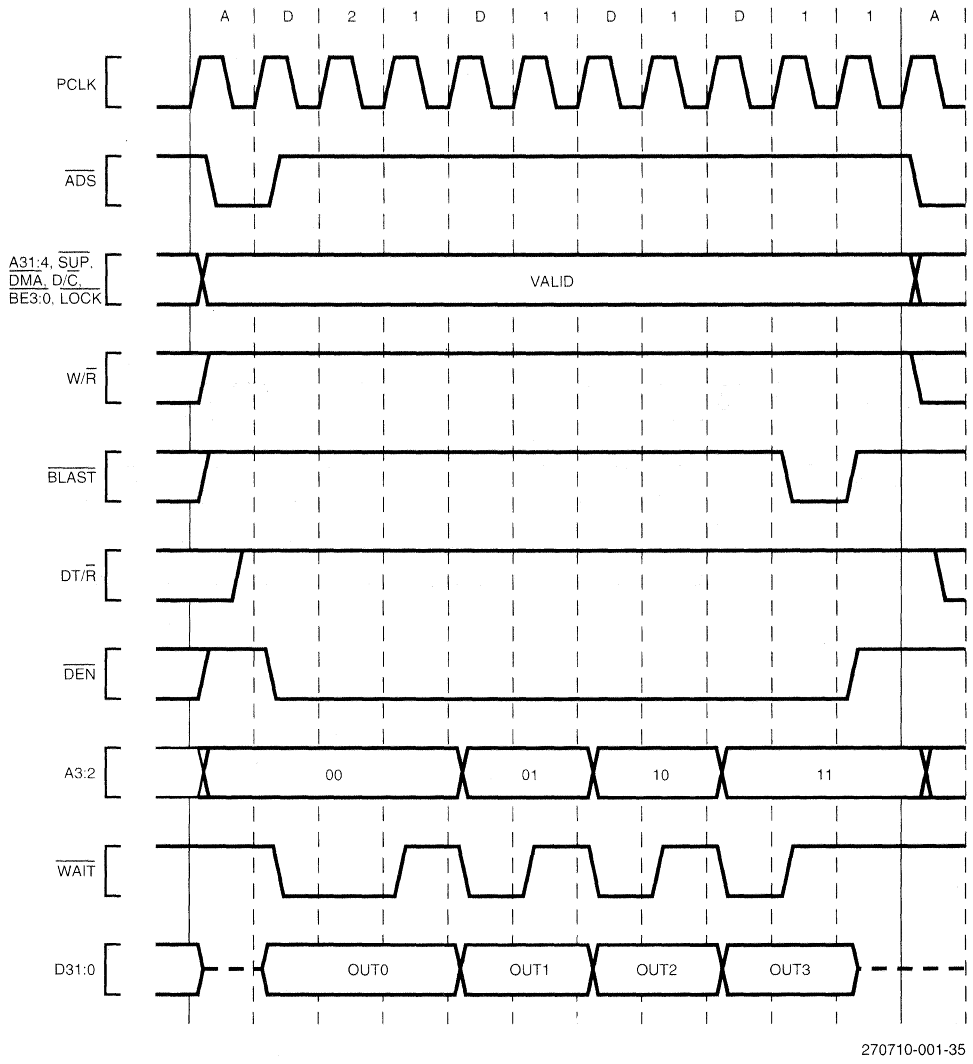


Figure 11.11. 32-Bit Bus, Burst, Non-Pipelined, Write Request with Wait States

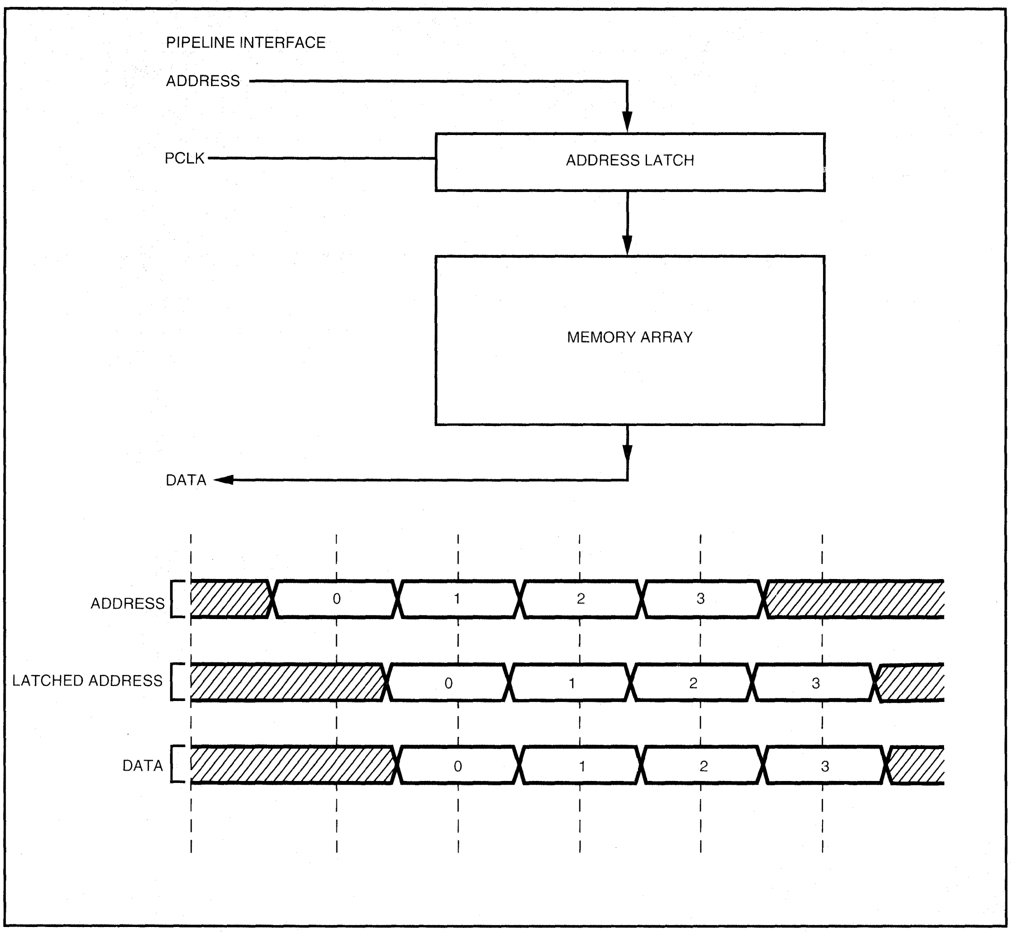
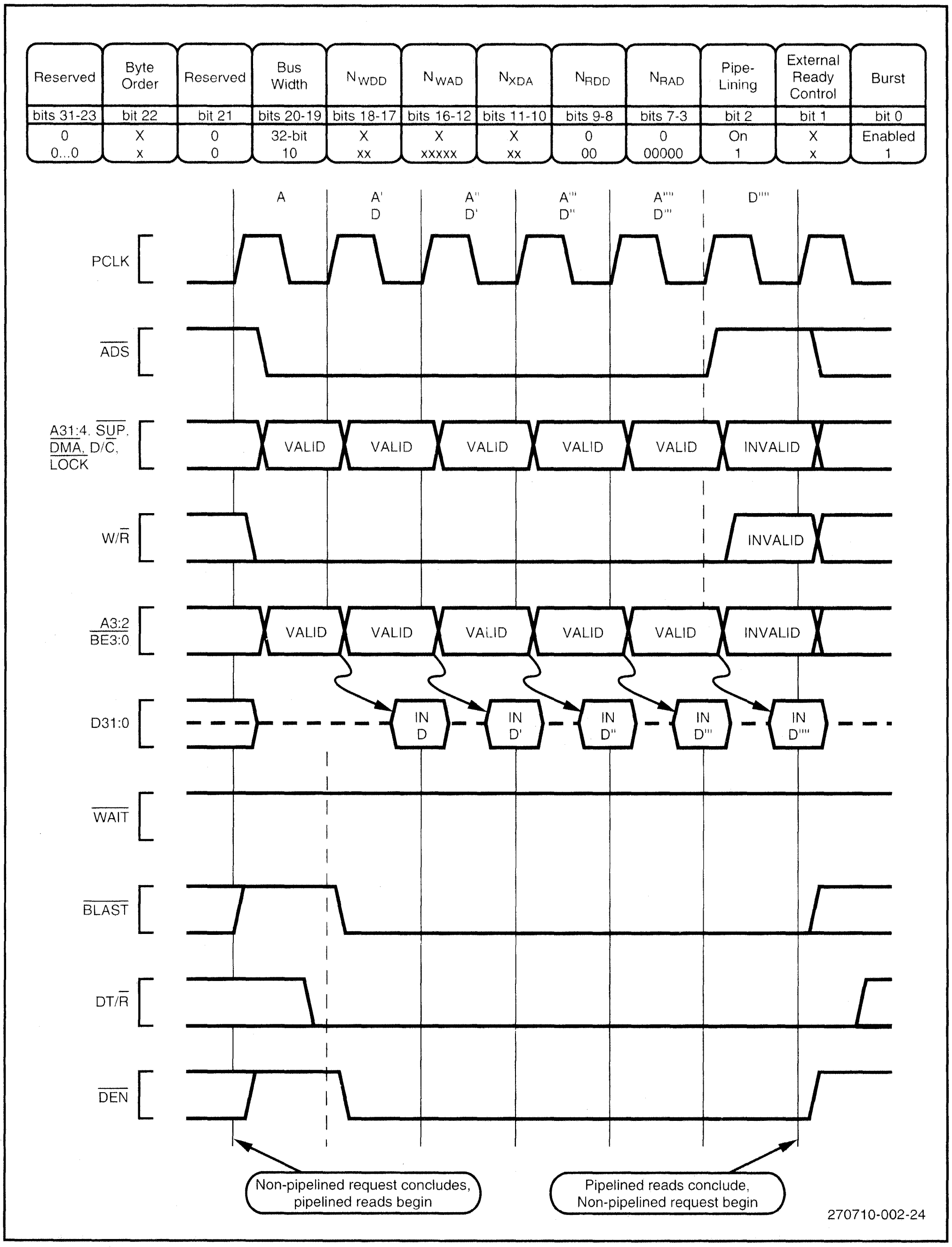


Figure 11.12. Pipelined Read Memory System



Reserved	Byte Order	Reserved	Bus Width	N_{WDD}	N_{WAD}	N_{XDA}	N_{RDD}	N_{RAD}	Pipe-Lining	External Ready Control	Burst
bits 31-23	bit 22	bit 21	bits 20-19	bits 18-17	bits 16-12	bits 11-10	bits 9-8	bits 7-3	bit 2	bit 1	bit 0
0	X	0	32-bit	X	X	X	0	0	On	X	Enabled
0...0	x	0	10	xx	xxxxx	xx	00	00000	1	x	1

A
A' D
A'' D'
A''' D''
A'''' D'''
D''''
PCLK
ADS
A31:4, SUP, DMA, D/C, LOCK
VALID
INVALID
W/R
A3:2 BE3:0
D31:0
IN D
IN D'
IN D''
IN D'''
IN D''''
WAIT
BLAST
DT/R
DEN
Non-pipelined request concludes, pipelined reads begin
Pipelined reads conclude, Non-pipelined request begin
270710-002-24

Figure 11.13. Non-Burst Pipelined Read Waveform

Reserved	Byte Order	Reserved	Bus Width	N_{WDD}	N_{WAD}	N_{XDA}	N_{RDD}	N_{RAD}	Pipe-Lining	External Ready Control	Burst
bits 31-23	bit 22	bit 21	bits 20-19	bits 18-17	bits 16-12	bits 11-10	bits 9-8	bits 7-3	bit 2	bit 1	bit 0
0 0...0	X x	0 0	32-bit 10	X xx	X xxxxx	X xx	0 00	0 00000	On 1	X x	Enabled 1

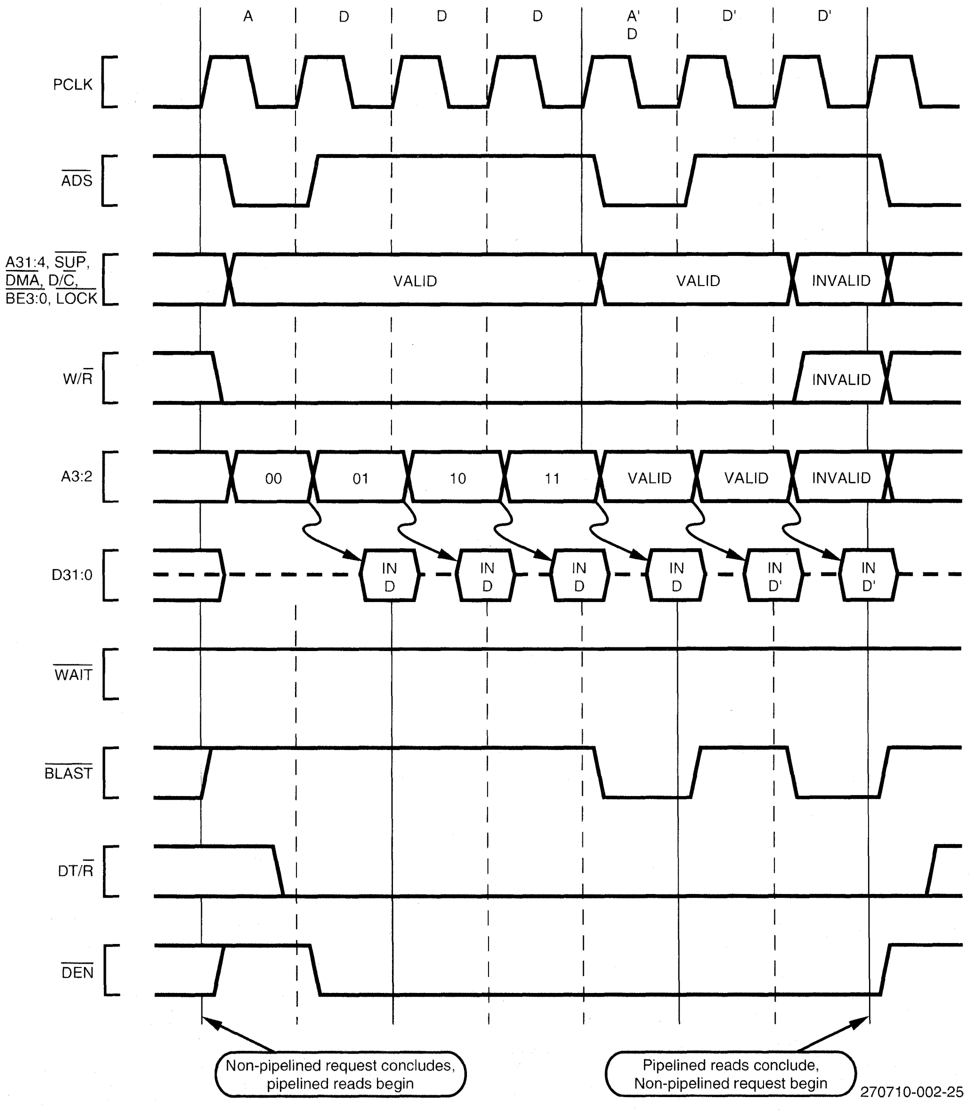

A
D
D
D
A'
D
D'
D'
PCLK
ADS
A31:4, SUP, DMA, D/C, BE3:0, LOCK
VALID
VALID
INVALID
W/R
INVALID
A3:2
00
01
10
11
VALID
VALID
INVALID
D31:0
IN D
IN D
IN D
IN D
IN D'
IN D'
WAIT
BLAST
DT/R
DEN
Non-pipelined request concludes, pipelined reads begin
Pipelined reads conclude, Non-pipelined request begin
270710-002-25

Figure 11.14. Burst Pipelined Read Waveform

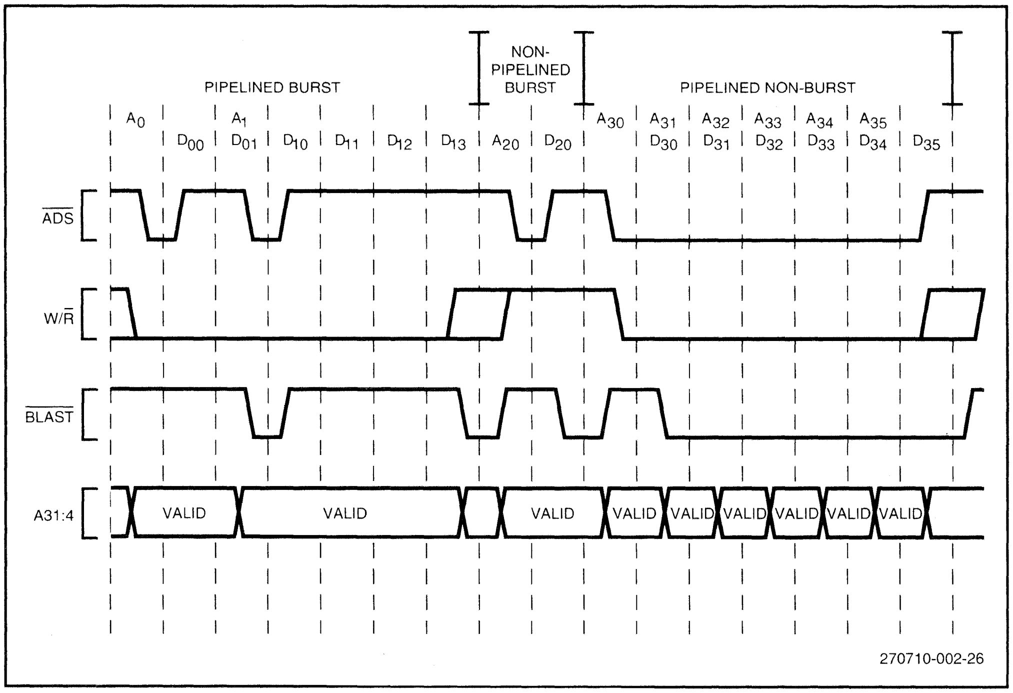


Figure 11.15. Pipelined to Non-Pipelined Transitions

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.

The i960 CA processor uses little endian byte ordering internally for data-in registers and data-in internal data RAM. Data-in memory (except for internal data RAM) can be stored in either little or big endian order. A bit in the region table entry for a memory region determines the type of.byte ordering used in that region. 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 11.4 shows how a 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.

For the word data type, assume that a hexadecimal value of **aabbccddH** is stored in an internal i960 CA processor register, where **aa** is the word's most significant byte and **dd** is the least significant byte. Table 11.4 then shows how this word is transferred on the bus to either a little endian or big endian region of memory.

For the half-word data type, assume that a hexadecimal value of **ccddH** is stored in one of the i960 CA processor's internal registers. Table 11.4 then shows how this half word is transferred on the bus to either a little endian or big endian memory region. Note that the half-word goes out on different data lines on a 32-bit bus depending on whether address line A1 is odd or even.

Table 11.4 also exhibits how the i960 CA 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; however, words are stored in little endian order. Big endian data types that exceed 32 bits are not supported and must be handled by software.

Table 11.4. Byte Ordering on Bus Transfers

Word Data Type			Bus Pins (data lines 31:0)							
			Little Endian				Big Endian			
Bus Width	Addr Bits A1, A0	Xfer	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
	00	2nd	--	--	aa	bb	--	--	dd	cc
8 bit	00	1st	--	--	--	dd	--	--	--	aa
	00	2nd	--	--	--	cc	--	--	--	bb
	00	3rd	--	--	--	bb	--	--	--	cc
	00	4th	--	--	--	aa	--	--	--	dd

Half-Word Data Type			Bus Pins (data lines 31:0)							
			Little Endian				Big Endian			
Bus Width	Addr Bits A1, A0	Xfer	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)			
			Little and Big Endian			
Bus Width	Addr Bits A1, A0	Xfer	31:24	23:16	15:8	7:0
32 bit	00	1st	--	--	--	dd
	01	1st	--	--	dd	--
	10	1st	--	dd	--	--
	11	1st	dd	--	--	--
16 bit	X0	1st	--	--	--	dd
	X1	1st	--	--	dd	--
8 bit	XX	1st	--	--	--	dd

ATOMIC MEMORY OPERATIONS (THE $\overline{\text{LOCK}}$ SIGNAL)

$\overline{\text{LOCK}}$ output assertion indicates that the processor is executing an atomic read-modify-write operation. Atomic instructions (**atadd**, **atmod**) require indivisible memory access. That is, another bus agent must not access the target of the atomic instruction between read and write cycles. $\overline{\text{LOCK}}$ can be used to implement indivisible accesses to memory.

Atomic instructions consist of a load and store request to the same memory location. $\overline{\text{LOCK}}$ 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 $\overline{\text{LOCK}}$ pin is not active during the N_{XDA} states for the store request.

When implementing a locked memory subsystem, consider the interaction that the following mechanisms may have on the system. A system must account for these conditions during locked accesses:

- HOLD requests are acknowledged while $\overline{\text{LOCK}}$ is asserted.
- An atomic load or store may be suspended using the $\overline{\text{BOFF}}$ input.
- A DMA request may occur between the atomic load and store requests.

$\overline{\text{LOCK}}$ indicates that other agents should not write data to any address falling within the quad word boundary of the address on the bus when $\overline{\text{LOCK}}$ was asserted. $\overline{\text{LOCK}}$ is deasserted after the write portion of an atomic access. It is the responsibility of external arbitration logic to monitor the $\overline{\text{LOCK}}$ pin and enforce its meaning for atomic memory operations. (See Figure 11.16.)

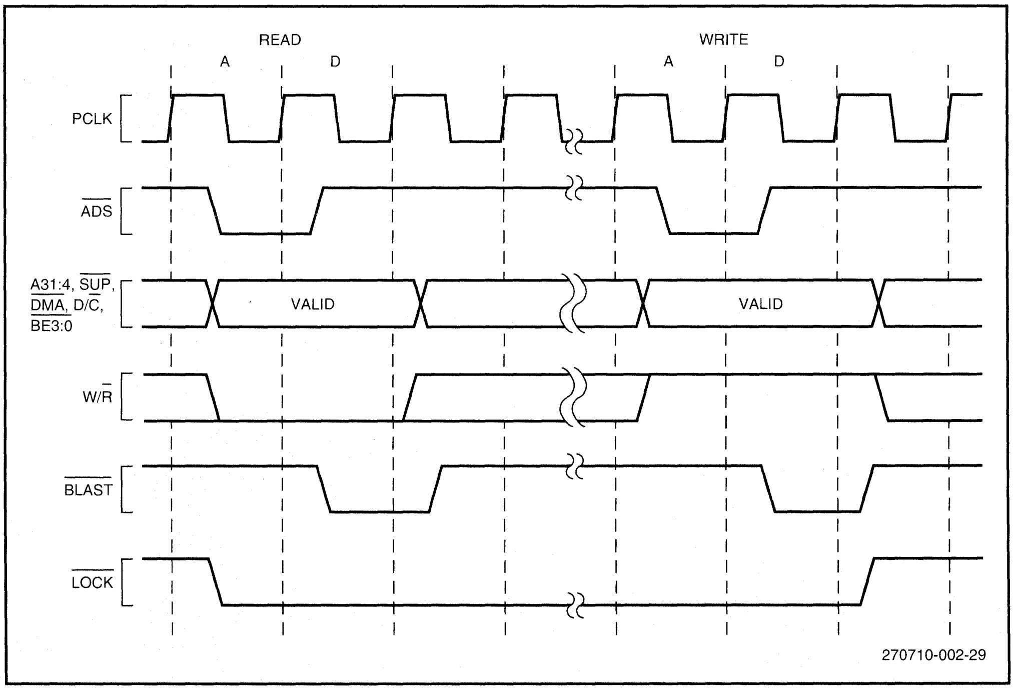


Figure 11.16. The $\overline{\text{LOCK}}$ Signal

EXTERNAL BUS ARBITRATION

The i960 CA processor provides a shared bus protocol to allow another bus master to access the processor's bus. The processor enters the *hold state* when an external bus master is granted bus control. In the hold state, the processor's data, address and control lines are floated (high Z) to allow the external bus master to control the bus and memory interface.

The HOLD input signal is asserted to indicate that another processor or peripheral is attempting to control the bus. The HOLDA (Hold Acknowledge) output signal acknowledges that the i960 CA processor has relinquished the bus. Bus pins float on the same clock cycle in which the hold request is granted (HOLDA asserted). When the i960 CA processor needs to access the bus, it uses the bus request signal (BREQ) to signal the other processor or peripheral.

When the HOLD signal is asserted, the i960 CA 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 request is in progress, the hold request is granted at the end of the current bus request.
- If the processor is in the backoff state ($\overline{\text{BOFF}}$ pin asserted), the hold request is granted after $\overline{\text{BOFF}}$ is deasserted and the resumed request has completed.

The hold request may be acknowledged between internal DMA load and store operations and atomic requests (read-modify-write accesses that assert $\overline{\text{LOCK}}$).

When the HOLD signal is removed, HOLDA is deasserted on the following PCLK2:1 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 *i960 CA Microprocessor Data Sheet.*

BREQ indicates that the bus controller queue contains one or more pending bus requests. The bus controller can queue up to three bus requests (refer to *Chapter 10, Bus Controller* for a complete description of the bus queue). When the bus queue is empty, the BREQ pin is deasserted. BREQ determines bus queue state during a hold state or before the hold state is requested. It may be useful to use BREQ to qualify hold requests and 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. The processor may continue executing from on-chip cache; therefore, it is possible that bus requests may be posted in the queue after the hold request is granted. In this case, BREQ can be used to relinquish the hold request when the processor needs the bus.

The HOLD and HOLDA arbitration can also function during the reset state. The bus controller acknowledges HOLD while $\overline{\text{RESET}}$ is asserted. If $\overline{\text{RESET}}$ is asserted while the processor has asserted HOLDA (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.

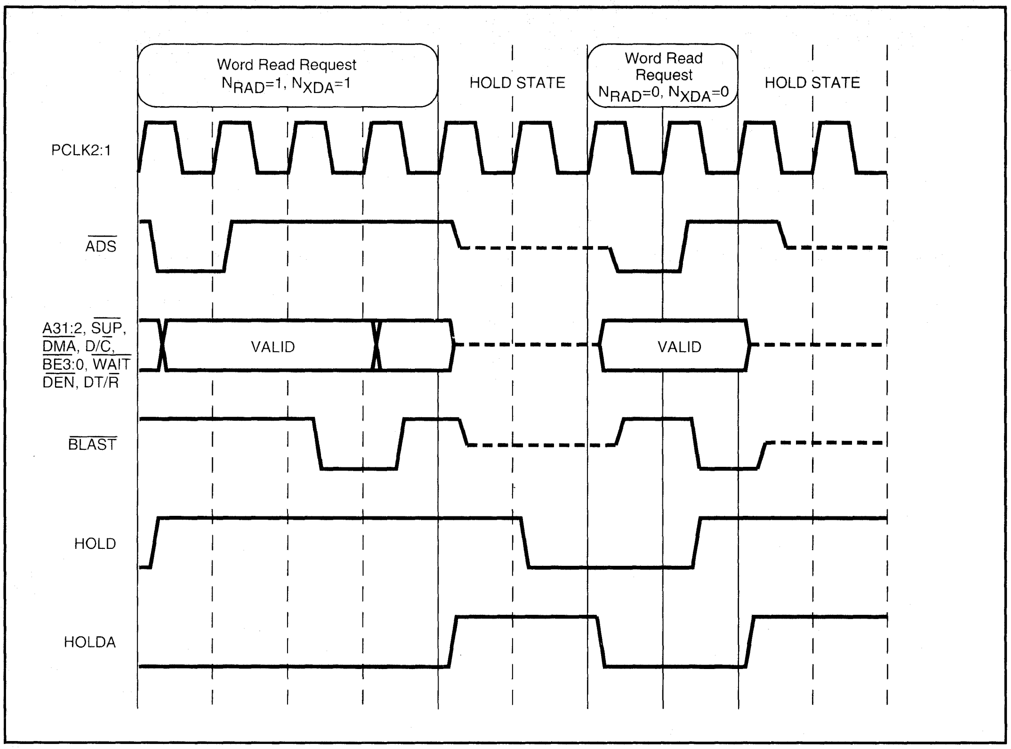


Figure 11.17. HOLD/HOLDA Bus Arbitration

Bus Backoff Function ($\overline{\text{BOFF}}$ pin)

The bus backoff input ($\overline{\text{BOFF}}$) suspends a bus request already in progress and allow another bus master to temporarily take control of the bus. The $\overline{\text{BOFF}}$ pin causes the current bus request to be suspended. When $\overline{\text{BOFF}}$ is asserted, 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, $\overline{\text{BOFF}}$ is deasserted and the suspended request is resumed upon assertion of $\overline{\text{ADS}}$ on the following clock cycle. (Figure 11.18).

The backoff function differs from the bus hold mechanism. The backoff function suspends a bus request which has already started. The request is later resumed when the pin is deasserted. The bus hold mechanism allows another bus master to control the bus only after all executing bus requests have completed.

Backoff can only be used for requests to regions which have the $\overline{\text{READY}}/\overline{\text{BTERM}}$ inputs enabled, with the N_{RAD}, N_{RDD}, N_{WAD} and N_{WDD} parameters programmed to 0.

$\overline{\text{BOFF}}$ may only be asserted during a bus access. Recall that a bus access includes and is bounded by clock cycles in which $\overline{\text{ADS}}$ is valid and the clock cycle in which $\overline{\text{BLAST}}$ is valid and $\overline{\text{READY}}$ input is asserted. External logic responsible for asserting $\overline{\text{BOFF}}$ must ensure that the signal is not asserted during idle bus cycles or during bus turnaround (N_{XDA}) cycles. Unpredictable behavior may occur if $\overline{\text{BOFF}}$ is subsequently deasserted during an idle bus or turnaround cycle.

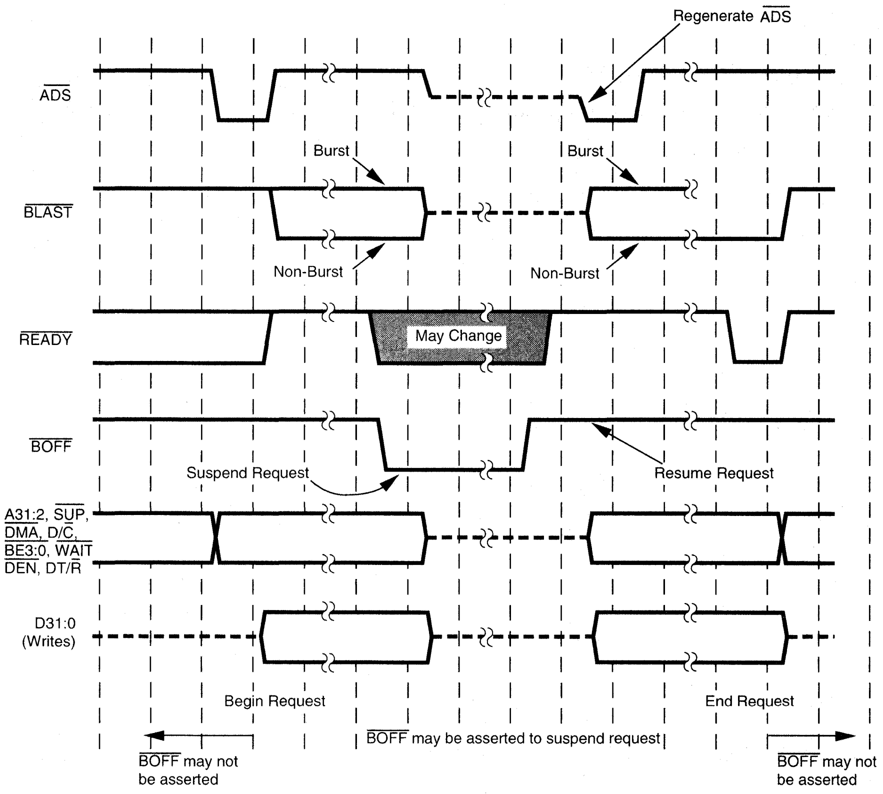
It is possible for HOLD and $\overline{\text{BOFF}}$ to be asserted in the same clock cycle. In this case, $\overline{\text{BOFF}}$ takes precedence. The bus is relinquished to a hold request only after the current request is complete.

Bus backoff is intended for use with special multiprocessor designs or bus architectures that do not implement "collision free" bus arbitration schemes (such as VME and MULTIUSERS I). A collision occurs when multiple processors begin a bus access simultaneously and a conflict for control of one of the processor's local memory occurs.

A bus collision is illustrated in the system diagram shown in Figure 11.19. In this system, several processors share a common bus. Each processor has local memory which 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 highest priority and Processor C has lowest priority 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 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 $\overline{\text{BOFF}}$ pin for Processor B.

When $\overline{\text{BOFF}}$ is asserted, external memory is responsible for gracefully cancelling the current access. This means that the memory control state machine should cancel write cycles and return to an idle state after $\overline{\text{BOFF}}$ is asserted. The processor ignores read data after $\overline{\text{BOFF}}$ is asserted.



NOTE: READY/ BTERM must be enabled; N_{RAD}, N_{RDD}, N_{WAD}, N_{WDD} = 0

Figure 11.18. Operation of the Bus Backoff Function

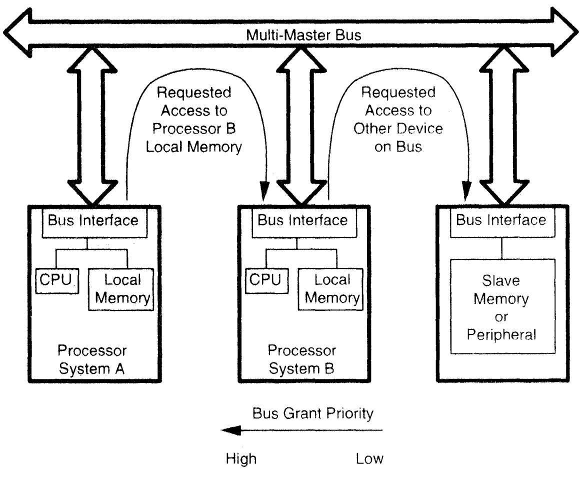


Figure 11.19. Example Application of the Bus Backoff Function

PIN AND BUS STATE DESCRIPTION

The following pin descriptions provide an easy reference to determine the state and behavior of the bus pins.

Pins are designated as an input (**I**), output (**O**) or input/output (**I/O**).

All output pins are synchronous to the PCLK2:1 output clock signals. Most outputs are synchronous to the rising edge of PCLK2:1. These outputs are designated with the symbol **S(R)**. Outputs which are synchronous to the falling edge of PCLK2:1 are designated with the symbol, **S(F)**.

Most input pins are synchronous inputs. A designer must adhere to specified setup and hold times for proper operation of these inputs. Synchronous inputs are designated in the same way as outputs, **S(R)** and **S(L)** depending on the PCLK2:1 edge on which the signal is sampled.

Some input pins are asynchronous inputs. These pins are designated by the symbols **A(L)** and **A(E)**. The **A(L)** symbol designates that the pin is a level sensitive input; **A(E)** indicates that the pin is falling edge latched.

The following list summarizes these pin designations:

Symbol	**Description**
I	Input only pin
O	Output only pin
I/O	Input/Output pin
S(R)	Synchronous input, output or input/output pin referenced to the rising edge of PCLK2:1
S(F)	Synchronous input, output or input/output pin referenced to the falling edge of PCLK2:1
A(L)	Asynchronous input, level activated
A(E)	Asynchronous input, falling edge activated

The pin descriptions also provide information concerning the state of the pin during different processor states. The symbols used to represent each state is given in the following list:

Symbol	**Bus State**
R( )	Reset State ($\overline{\text{RESET}}$ active)
I( )	Idle State (No executing bus requests
T( )	Turnaround State (N_{XDA} wait states)
H( )	Hold State (HOLDA signal is active)
B( )	Backoff State ($\overline{\text{BOFF}}$ signal is active)
O( )	On Circuit Emulation Mode ($\overline{\text{ONCE}}$ asserted on rising $\overline{\text{RESET}}$)

Bus state symbol argument specifies the state of output pins or the required value of input pins during each bus state. State symbol arguments are provided in the following list:

Input state symbol	**Description**
X	Input is a don't care (either high or low) during this state
0	Input must be driven low (0) during this state
1	Input must be driven high (1) during this state
Q	Input is recognized as a valid input. Function is given in pin description

Output state symbol	**Description**
Z	Output floats (high Z) during this state
X	Output is either high (1) or low (0) during this state
0	Output is low (0) during this state
1	Output is high (1) during this state
Q	Valid output during this state. Function is given in pin description.

NOTE

Arguments for I/O state symbols consist of in input and output argument separated by a "/" symbol. The argument designates that the pin is a known input or output in a particular state by using the "_" symbol on one side of the "/" symbol.

The following example demonstrates the use of these symbols when describing the pins:

D31:0	**I/O** **S(R)**	R(X/Z) I(X/Z) T(X/Z) H(X/Z) B(X/Z) O(X/Z)	32 Bit Data Bus. 32-, 16- and 8-bit values are transmitted and received on these lines. When a memory region is configured as an 8 bit bus, data is transferred on D7:0 only. When a memory region is configured as a 16 bit bus, data is transferred on D15:0 only. The data bus drives valid data for write operation and floats during reads and instruction fetches.

Data pin designation (D31:0) in the first column specifies that the pins are synchronous input/output pins.

I/O and **S(R)** in the second column indicate that the pins are referenced to the rising edge of PCLK2:1.

The bus state symbols in the third column indicate that — in the reset, idle, turnaround, hold, backoff and ONCE states — pins are floated and values driven on the pins are ignored.

Pin descriptions in this chapter do not discuss pins associated with the interrupt or DMA controllers. Refer to *Chapter 6, Interrupts* and *Chapter 13, DMA Controller* for a description of these pins. More details on the bus electrical characteristics are given in the *i960 CA Microprocessor Data Sheet.*

Bus Control Signals

PCLK2:1	**O**	R(Q) I(Q) T(Q) H(Q) B(Q) O(Z)	Processor Output Clocks. PCLK1 and PCLK2 are identical clock outputs for the processor's synchronous bus. All other bus operations are synchronized to these clocks. Two identical clock output pins are provided for additional drive capability. When configured for One-X clock mode, PCLK2:1 is synchronous to CLKIN input clock signal. When configured for Two-X clock mode, PCLK2:1 is one half the frequency of CLKIN. One-X or Two-X clock mode is selected with the CLKMODE pin.
D31:0	**I/O** **S(R)**	R(X/Z) I(X/Z) T(X/Z) H(X/Z) B(X/Z) O(X/Z)	32 Bit Data Bus. 32-, 16- and 8-bit values are transmitted and received on these lines. When a memory region is configured as an 8 bit bus, data is transferred on D7:0 only. When a memory region is configured as a 16 bit bus, data is transferred on D15:0 only. The data bus drives valid data for write operation and floats during reads and instruction fetches.

A31:2	**O**
	S(R)
		R(Z)
		I(X)
		T(X)
		H(Z)
		B(Z)
		O(Z)
			30 Bit Address. A31:2 carries physical address upper 30 bits. A31 is most significant address bit; A2 is least significant. The 30 bit address bus identifies all external addresses to word (4 Byte) boundaries. Byte enable lines ($\overline{BE3:0}$) indicate selected byte in each word. A3 and A2 increment during 32 bit burst accesses.

$\overline{BE3:0}$	**O**
	S(R)
		R(1)
		I(X)
		T(X)
		H(Z)
		B(Z)
		O(Z)
			Byte Enables. These select which of four addressed bytes are active in a 32-bit memory access. $\overline{BE0}$ applies to D7:0; $\overline{BE1}$ applies to D15:8; $\overline{BE2}$ applies to D23:16; $\overline{BE3}$ applies to D31:24. When a memory region is configured for an 8-bit data bus width, $\overline{BE1}$ and $\overline{BE0}$ act as the address lower two bits, A1 and A0, respectively. For a 16-bit memory region, $\overline{BE3}$, $\overline{BE1}$ and $\overline{BE0}$ are encoded as $\overline{BHE}$, A1 and $\overline{BLE}$, respectively.

$\overline{ADS}$	**O**
	S(R)
		R(1)
		I(1)
		T(1)
		H(Z)
		B(Z)
		O(Z)
			Address Strobe. This control signal indicates valid address and the start of a new bus access. $\overline{ADS}$ is asserted for the first clock of a bus access.

$\overline{WAIT}$	**O**
	S(R)
		R(1)
		I(1)
		T(1)
		H(Z)
		B(Z)
		O(Z)
			Wait. This signal indicates internal wait state generator status. $\overline{WAIT}$ is active when wait states are caused by the internal wait state generator inserting either N_{RAD}, N_{RDD}, N_{WAD} or N_{WDD} wait states. $\overline{WAIT}$ is not asserted during N_{XDA} cycles or when wait states are caused by $\overline{READY}$. $\overline{WAIT}$ can be used to derive a write data strobe. $\overline{WAIT}$ can also be considered as a memory ready output, which the processor provides when inserting wait states.

$\overline{BLAST}$	**O**
	S(R)
		R(0)
		I(1)
		T(1)
		H(Z)
		B(Z)
		O(Z)
			Burst Last. $\overline{BLAST}$ is a control signal that indicates the end of a bus access. This signal is asserted in the last data transfer of every bus access, whether burst, non-burst or pipelined.

$\overline{\text{READY}}$	**I**
	S(R)
		R(X)
		I(X)
		T(X)
		H(X)
		B(X)
		O(X)
			Memory Ready. $\overline{\text{READY}}$ input signal indicates that read data on the bus is valid or a write-data transfer has completed. $\overline{\text{READY}}$ works in conjunction with the internally programmed wait state generator. If $\overline{\text{READY}}$ and $\overline{\text{BTERM}}$ are enabled in a region, $\overline{\text{READY}}$ is sampled after the programmed number of wait states has expired. If $\overline{\text{BTERM}}$ is not asserted (high) and $\overline{\text{READY}}$ is deasserted (high), wait states continue to be inserted until $\overline{\text{READY}}$ is asserted (low). This is true for N_{RAD}, N_{RDD}, N_{WAD} and N_{WDD} wait states. N_{XDA} wait states cannot be extended by $\overline{\text{READY}}$. To satisfy $\overline{\text{READY}}$ setup and hold times, $\overline{\text{READY}}$ must be externally synchronized. Setup and hold specifications are given in the *i960 CA Microprocessor Data Sheet*.

$\overline{\text{BTERM}}$	**I**
	S(R)
		R(X)
		I(X)
		T(X)
		H(X)
		B(X)
		O(X)
			Burst Terminate. This signal breaks up a burst access and causes another address cycle to occur. $\overline{\text{BTERM}}$ works in conjunction with the internally programmed wait state generator. If $\overline{\text{READY}}$ and $\overline{\text{BTERM}}$ are enabled in a region, $\overline{\text{BTERM}}$ is sampled after the programmed number of wait states has expired. When $\overline{\text{BTERM}}$ is deasserted, a new $\overline{\text{ADS}}$ signal is generated and the access is completed. $\overline{\text{READY}}$ input is ignored when $\overline{\text{BTERM}}$ is asserted. $\overline{\text{BTERM}}$ must be externally synchronized to satisfy $\overline{\text{BTERM}}$ setup and hold times. Setup and hold specifications are given in the *i960 CA Microprocessor Data Sheet*.

$\overline{\text{DEN}}$	**O**
	S(R)
		R(1)
		I(1)
		T(1)
		H(Z)
		B(Z)
		O(Z)
			Data Enable. $\overline{\text{DEN}}$ is asserted (low) after the first address cycle of a bus request and is deasserted at the end of the last data cycle of the request (before the N_{XDA} cycles). $\overline{\text{DEN}}$ is used to control external data transceivers. $\overline{\text{DEN}}$ remains asserted for sequential accesses to pipelined regions.

Bus Status Signals

W/$\overline{\text{R}}$	**O**
	S(R)
		R(0)
		I(X)
		T(Q)
		H(Z)
		B(Z)
		O(Z)
			Write/Read. This request status signal is low for loads and instruction fetch requests and high for store requests. W/$\overline{\text{R}}$ changes in the same clock cycle that $\overline{\text{ADS}}$ is asserted and remains valid for the entire request in a non-pipelined regions. In pipelined regions, W/$\overline{\text{R}}$ is not valid in the last data cycle of a read request.

DT/$\overline{\text{R}}$	**O**
	S(F)
		R(0)
		I(X)
		T(Q)
		H(Z)
		B(Z)
		O(Z)
			Data Transmit/Receive. Used for direction control for data transceivers. DT/$\overline{\text{R}}$ is low when the i960 CA processor is reading data and high when writing data. DT/$\overline{\text{R}}$ does not change while $\overline{\text{DEN}}$ is asserted. DT/$\overline{\text{R}}$ remains valid for the entire bus request, including N_{XDA} cycles.

D/$\overline{\text{C}}$	**O**
	S(R)
		R(Z)
		I(X)
		T(Q)
		H(Z)
		B(Z)
		O(Z)
			Data/Code. D/$\overline{\text{C}}$ is a request status output that indicates that the current request is either a data transfer or instruction fetch.

$\overline{\text{DMA}}$	**O**
	S(R)
		R(Z)
		I(X)
		T(Q)
		H(Z)
		B(Z)
		O(Z)
			CPU/DMA. $\overline{\text{DMA}}$ is a request status output that indicates that a bus request is issued by the DMA controller (low) or by the user program (high).

$\overline{\text{SUP}}$	**O**
	S(R)
		R(Z)
		I(X)
		T(Q)
		H(Z)
		B(Z)
		O(Z)
			User/Supervisor. $\overline{\text{SUP}}$ is a request status output that indicates that a bus request is issued by the processor from the supervisor mode.

Bus Arbitration Signals

HOLD	**I**
	S(R)
		R(Q)
		I(Q)
		T(X)
		H(1)
		B(X)
		O(X)
			Hold. Used by an external bus master to request bus access. The processor asserts HOLDA and relinquishes the bus after the current bus request completes. HOLD may be generated by external bus arbitration logic that monitors the HOLDA, BREQ and $\overline{\text{LOCK}}$ signals. It must be externally synchronized to satisfy the timings found in the *i960 CA Microprocessor Data Sheet.*

$\overline{\text{LOCK}}$	**O**
	S(R)
		R(1)
		I(1)[1]
		T(Q)[2]
		H(Z)
		B(Z)
		O(Z)
			Lock indicates that an atomic memory operation (**atadd**, **atmod**) is in progress. Atomic memory operations are read-modify-write operations. $\overline{\text{LOCK}}$ indicates that other processors or peripherals should not write data to any address that falling within the quad word boundary of the address on the bus when $\overline{\text{LOCK}}$ was asserted. $\overline{\text{LOCK}}$ is deasserted after the write portion of an atomic access. A HOLD request is acknowledged by HOLDA during locked cycles. It is the responsibility of external arbitration logic to monitor the $\overline{\text{LOCK}}$ pin and enforce its meaning for atomic memory operations.
			
			[1] $\overline{\text{LOCK}}$ is active (low) during IDLE cycles between read and write requests of a locked access.
			
			[2] Valid during turnaround states for first request of atomic access only.

HOLDA	**O**
	S(R)
		R(Q)
		I(Q)
		T(0)
		H(1)
		B(0)
		O(Z)
			Hold Acknowledge. HOLDA indicates to a bus requester that the processor has relinquished bus control. It is asserted in the same clock that the bus goes into the high impedance state. HOLDA is put into a high impedance state during ONCE operation.

BREQ	**O**
	S(R)
		R(0)
		I(0)
		T(Q)
		H(Q)
		B(Q)
		O(Z)
			Bus Request. While HOLDA is asserted, BREQ indicates that the i960 CA processor bus controller wishes to perform an external memory operation. BREQ can be used with external bus arbitration logic to regain bus control. It is put into a high impedance state during ONCE operation.

$\overline{\text{BOFF}}$	**I**
	S(R)
		R(1)
		I(1)
		T(1)
		H(1)
		B(0)
		O(X)
			Bus Backoff (input). The backoff pin, when asserted (low), suspends the current access and causes bus pins to float. When the pin is deasserted (high), $\overline{\text{ADS}}$ is asserted on the next clock cycle and the access is resumed.

Processor Control Signals

$\overline{\text{ONCE}}$	**I**
	A(L)
		R(Q)
		I(X)
		T(X)
		H(X)
		B(X)
		O(X)
			On Circuit Emulation (input). This signal is pulled up internally; the user is advised to leave it unconnected for normal operation. If $\overline{\text{ONCE}}$ is asserted (low) while $\overline{\text{RESET}}$ is asserted (low), all output pins float and all internal pull-ups and pull-downs are turned off. This allows in-circuit testing by external testers and allows ICE systems to emulate in-circuit devices.

CLKMODE	**I**
	A(L)
		R(Q)
		I(Q)
		T(Q)
		H(Q)
		B(Q)
		O(X)
			Clock Mode (input). Clock mode input selects the division factor applied to external clock input (CLKIN). When CLKMODE is high, PCLK2:1 and internal clocks are the same frequency as CLKIN. When CLKMODE is low, CLKIN is divided by two to create PCLK2:1 and the processor's internal clocks. CLKMODE input should be tied either high or low in a system because pin value is not latched by the processor.

STEST	**I**
	A(L)
		R(Q)
		I(X)
		T(X)
		H(X)
		B(X)
		O(X)
			Self Test Select (input). The self test input causes the processor's internal self test feature to be enabled or disabled at initialization. STEST is latched on the rising edge of $\overline{\text{RESET}}$. When asserted (high), the processor's internal self test and bus confidence test run at initialization. If deasserted (low), only the bus confidence test runs at initialization; internal self test is bypassed.

$\overline{\text{FAIL}}$	**O**
	S(R)
		R(0)
		I(Q)[1]
		T(1)
		H(Z)
		B(Z)
		O(Z)
			Self test Fail. The fail output indicates failure of the processor's self test at initialization. When $\overline{\text{RESET}}$ is deasserted and the processor begins initialization, $\overline{\text{FAIL}}$ is asserted (low). An internal self test is performed as part of the initialization process. If this self test passes, $\overline{\text{FAIL}}$ is deasserted (high), otherwise it remains asserted. $\overline{\text{FAIL}}$ is reasserted while the processor performs an external bus self confidence test. If this self test passes, the processor deasserts $\overline{\text{FAIL}}$ and branches to the user's initialization routine; otherwise, $\overline{\text{FAIL}}$ remains asserted. Internal self test and the use of the $\overline{\text{FAIL}}$ pin can be disabled with the STEST pin.
			
			[1]The $\overline{\text{FAIL}}$ pin is only valid after initialization failure.

Interrupt Controller 12

CHAPTER 12
INTERRUPT CONTROLLER

This chapter contains interrupt controller information that is of particular importance to the system implementor. The method for handling interrupt requests from user code is described in *Chapter 6, Interrupts*. Specifically, this chapter describes the i960 CA microprocessor'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.

OVERVIEW

The interrupt controller's primary functions are to provide a flexible, low-latency means for requesting and posting interrupts and to minimize the core's interrupt handling burden. The interrupt controller handles the posting of interrupts requested by hardware and software sources. The interrupt controller, acting independently from the core, compares the priorities of posted interrupts with the current process priority, off-loading this task from the core.

The interrupt controller provides the following features for managing hardware-requested interrupts:

- Low latency, high throughput handling.
- Support of up to 248 external sources.
- Eight external interrupt pins, one non-maskable interrupt pin, four internal DMA 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 four control registers and two special function registers. The interrupt control register (ICON) and interrupt map control registers (IMAP0-IMAP3) provide configuration information. The interrupt pending (IPND) special function register posts hardware-requested interrupts. The interrupt mask (IMSK) special function register selectively masks hardware-requested interrupts.

MANAGING INTERRUPT REQUESTS

The i960 family architecture provides a consistent interrupt model, as required for interrupt handler compatibility between various implementations of the i960 family. The architecture, however, leaves the interrupt request management strategy to the specific i960 family implementations. In the i960 CA microprocessor, the programmable on-chip interrupt controller transparently manages all interrupt requests (Figure 12.1). These requests originate from:

- 8-bit external interrupt pins ($\overline{\text{XINT7:0}}$)
- non-maskable interrupt pin ($\overline{\text{NMI}}$)
- four DMA controller channels
- **sysctl** instruction execution

External interrupt pins can be programmed to operate in several modes: the pins may be individually mapped to interrupt vectors (dedicated mode) or they may be interpreted as a bit field which can request any of the 248 possible interrupts that the i960 family supports (expanded mode). Dedicated-mode requests are posted in the Interrupt Pending Register (IPND). The processor does not post expanded-mode requests.

Interrupt pins may also be configured in a mixed mode which places three pins into dedicated-mode operation and the remaining five pins in expanded-mode operation.

The $\overline{\text{NMI}}$ pin allows a highest-priority, non-maskable and non-interruptible interrupt to be requested. $\overline{\text{NMI}}$ is always a dedicated-mode input.

Each of the four DMA channels has an associated interrupt request to allow the application to synchronize with the DMA operations of each channel. DMA interrupt requests are always handled as dedicated-mode interrupt requests.

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 interrupt controller continuously compares the priorities of the highest-posted software interrupt and the highest-pending hardware interrupt to the processor's priority. The core is interrupted when a pending interrupt request is higher than the processor priority or a priority-31. In the event that both hardware- and software-requested interrupts are posted at the same level, the hardware interrupt is serviced before the software interrupt, when the priority is 1 to 30. At priority 31, the software interrupt is serviced first.

The following sections describe interrupt controller modes, interrupt request pins and inputs, user interface to the interrupt controller, the method for posting software-generated interrupt requests and methods for controlling interrupt latency.

Interrupt Controller Modes

The eight external interrupt pins can be configured for one of three modes: expanded, dedicated and mixed.

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_2 (Figure 12.2), where bits marked P are programmed with bits in the interrupt map (IMAP) registers. This encoding of programmable bits and preset bits can designate 15 unique vector numbers, each with a unique, even-numbered priority. (Vector 0000 0010_2 is undefined; it has a priority of 0.)

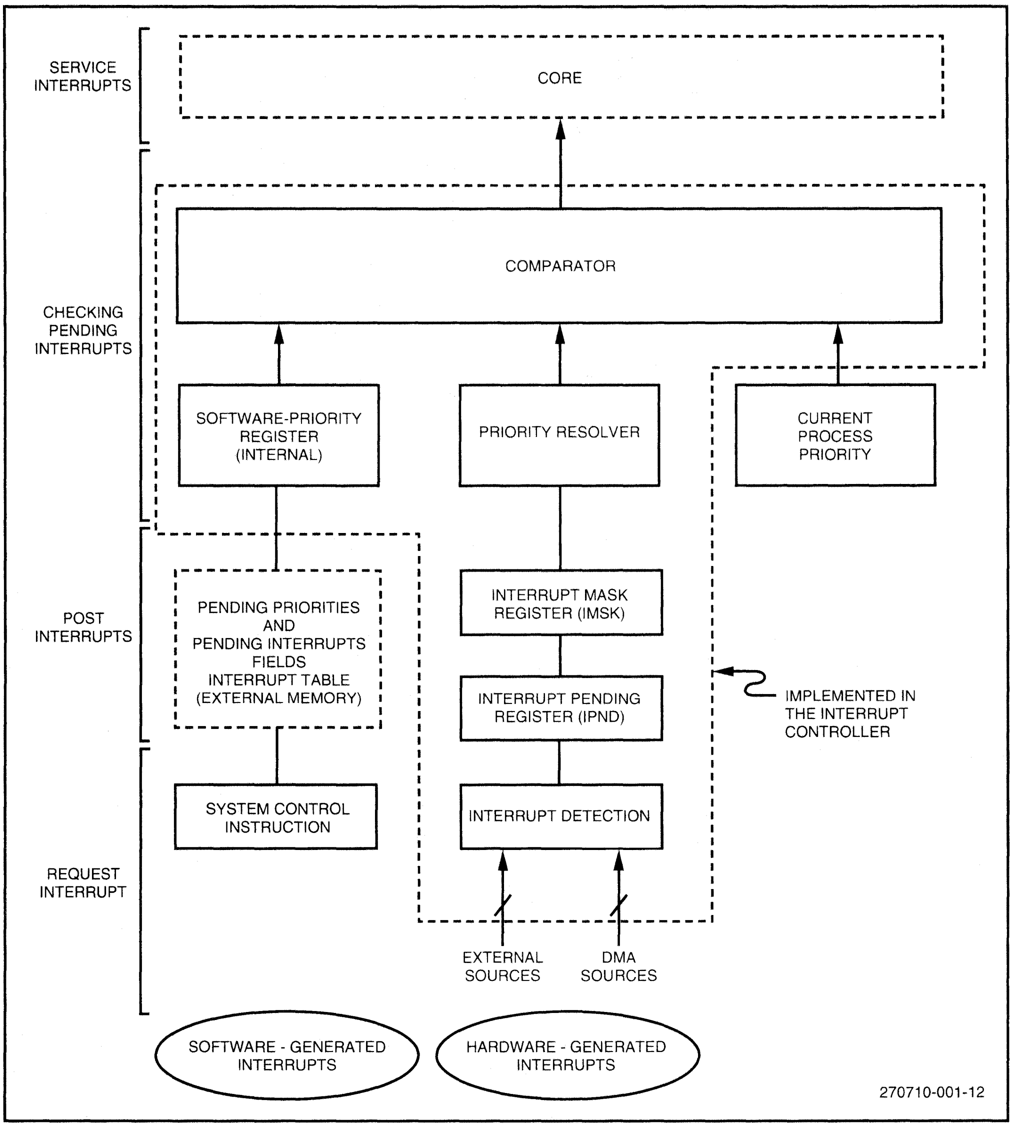

SERVICE INTERRUPTS
CORE
CHECKING PENDING INTERRUPTS
COMPARATOR
SOFTWARE-PRIORITY REGISTER (INTERNAL)
PRIORITY RESOLVER
CURRENT PROCESS PRIORITY
POST INTERRUPTS
PENDING PRIORITIES AND PENDING INTERRUPTS FIELDS INTERRUPT TABLE (EXTERNAL MEMORY)
INTERRUPT MASK REGISTER (IMSK)
INTERRUPT PENDING REGISTER (IPND)
IMPLEMENTED IN THE INTERRUPT CONTROLLER
SYSTEM CONTROL INSTRUCTION
INTERRUPT DETECTION
REQUEST INTERRUPT
EXTERNAL SOURCES
DMA SOURCES
SOFTWARE - GENERATED INTERRUPTS
HARDWARE - GENERATED INTERRUPTS
270710-001-12

Figure 12.1. i960™ CA Processor's Interrupt Controller

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, plus the four DMA inputs to the interrupt controller. The interrupt mask (IMSK) register selectively masks each of the dedicated-mode interrupts. The IMSK register can optionally 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 *Programmer's Interface* in this chapter for a further description of the IMSK, IPND and IMAP registers.

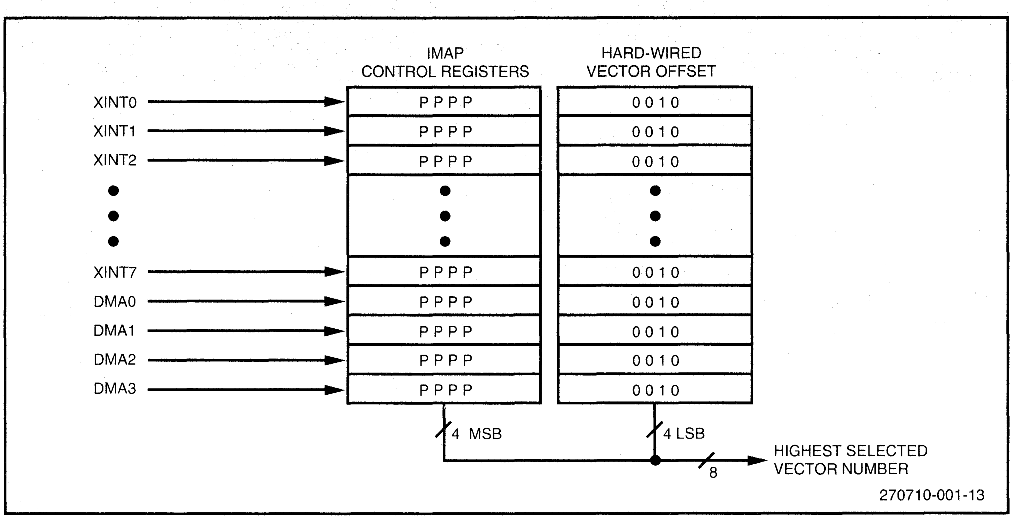


Figure 12.2. Dedicated Mode

Interrupt vectors are assigned to DMA inputs in the same way external pins are assigned dedicated-mode vectors. The DMA interrupts are always dedicated-mode interrupts.

Expanded Mode

In expanded mode, up to 248 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 12.3), with the $\overline{\text{XINT0}}$ pin representing the least-significant bit and $\overline{\text{XINT7}}$ 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. As shown in Figure 12.4, simple, combinational logic can handle prioritization of the external sources when more than one expanded interrupt is pending.

NOTE

The interrupt source, as shown in Figure 12.4, must remain asserted until the processor services the interrupt and explicitly clears the source.

External-interrupt pins in expanded mode are always active low. The interrupt controller ignores vector numbers 0 though 7. The output of the external priority encoders in Figure 12.4 can use the 0 vector to indicate that no external interrupts are pending.

IMSK register bit 0 provides a global mask for all expanded interrupts. The remaining bits (1-7) should be set to 0 in expanded mode. The mask bit can optionally 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. (See *Mask Options* later in this chapter.) IPND register bits 0-7, in expanded mode, have no function since external logic is responsible for posting interrupts.

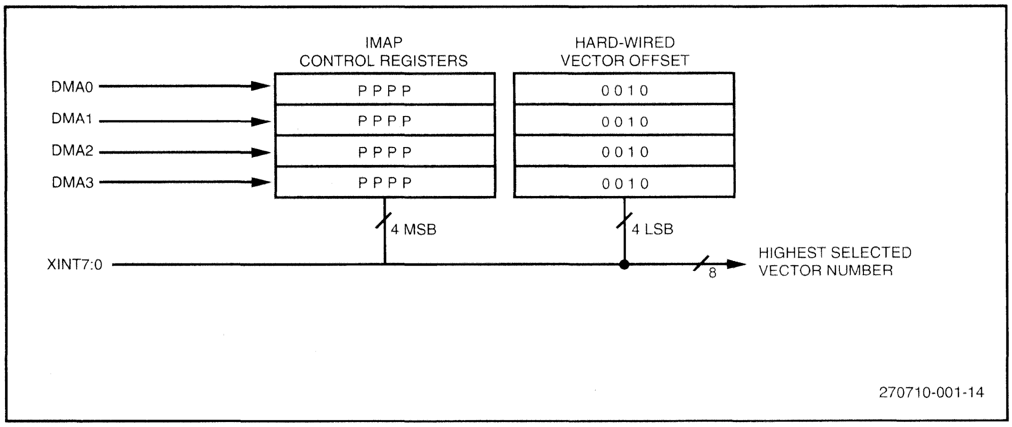


Figure 12.3. Expanded Mode

Mixed Mode

In mixed mode, pins $\overline{\text{XINT0}}$ through $\overline{\text{XINT4}}$ 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 be 010_2. Pins $\overline{\text{XINT5}}$ through $\overline{\text{XINT7}}$ are configured for dedicated mode.

IMSK register bit 0 is a global mask for the expanded-mode interrupts; bits 5 through 7 mask the dedicated interrupts from pins $\overline{\text{XINT5}}$ through $\overline{\text{XINT7}}$, respectively. IMSK register bits 1-4 must be set to 0 in mixed mode. The IPND register posts interrupts from the dedicated-mode pins ($\overline{\text{XINT5}}$-$\overline{\text{XINT7}}$). IPND register bits that correspond to expanded-mode inputs are not used.

CAUTION!

When setting IMSK register bits in mixed mode, make sure IMSK register bits 1-4 are set to 0.

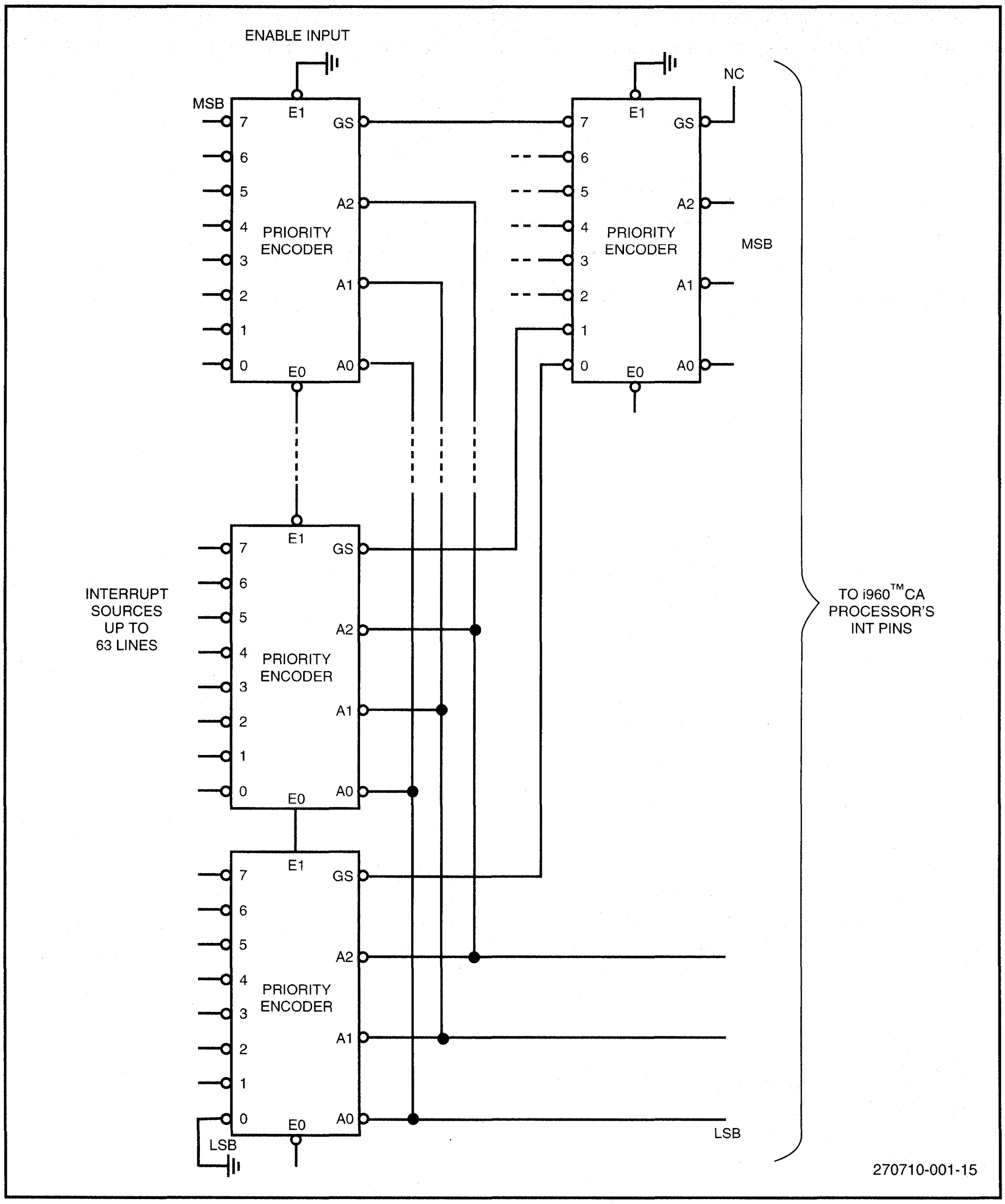


Figure 12.4. Implementation of Expanded Mode Sources

Non-Maskable Interrupt

The NMI pin generates an interrupt for implementation of highly-critical interrupt routines. The NMI provides an interrupt that cannot be masked and that has a higher priority than priority-31 interrupts and priority-31 process priority. The interrupt vector for the NMI resides in the interrupt table as vector number 248. During initialization, the core caches the vector for the NMI on-chip, to reduce NMI latency. The NMI vector is cached in location 0H of internal data RAM.

When the core receives an NMI request, it is serviced immediately. While servicing an NMI, the core does not respond to any other interrupt requests — even another NMI request — until it returns from the NMI-handling procedure. An interrupt request on the $\overline{\text{NMI}}$ pin is always falling-edge detected.

Saving the Interrupt Mask

The IMSK register is automatically saved in register r3 when a hardware-requested interrupt is serviced. After the mask is saved, the IMSK register is optionally cleared. This action allows all interrupts, except NMIs, 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 (described in this chapter). Several options are provided for interrupt mask handling:

1. Mask is unchanged.
2. Clear for dedicated-mode sources only.
3. Clear for expanded-mode sources only.
4. Clear for all hardware-requested interrupts (dedicated and expanded mode).

Options 2 and 3 are used in mixed mode, where both dedicated-mode and expanded-mode inputs are allowed. Recall that DMA interrupts are always dedicated-mode interrupts.

NOTE

If 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. If these priority-31 interrupts are not masked, another priority-31 interrupt will be signaled and serviced before the handler is able to 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.

External Interface Description

This section describes the physical characteristics of the interrupt inputs. The i960 CA microprocessor 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 248 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.

Pin Descriptions

The interrupt controller provides nine interrupt pins:

$\overline{\text{XINT7:0}}$	External Interrupt (Input) - These pins cause interrupts to be requested. Pins are software configurable for three modes: dedicated, expanded, mixed. Each pin can be programmed as an edge-detect input or as a level-detect input. Additionally, a debouncing mode for these pins can be selected under program control.
$\overline{\text{NMI}}$	Non-Maskable Interrupt (Input) Causes a non-maskable interrupt event to occur. NMI is the highest priority interrupt recognized. The $\overline{\text{NMI}}$ pin is an edge-activated input. A debouncing mode for $\overline{\text{NMI}}$ can be selected under program control. These pins are internally synchronized.

External interrupt pin functions $\overline{\text{XINT7:0}}$ depend on the operation mode (expanded, dedicated or mixed) and on several other options selected by setting ICON register bits.

Interrupt Detection Options

The $\overline{\text{XINT7:0}}$ pins can be programmed for level-low or falling-edge detection when used as dedicated inputs. All dedicated inputs plus the $\overline{\text{NMI}}$ 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 a pin is programmed for falling-edge detection, the corresponding pending bit in the IPND register is set when a high-to-low transition is detected. The processor clears the IPND bit on 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.

Typically, the external source for a level-detect interrupt is deactivated under software control. 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 12.1 demonstrates how a level detection interrupt is typically handled. The example assumes that the **ld** from address "timer_0," deactivates the interrupt input.

Example 12.1. Return from a Level-detect Interrupt

```
# Clear level-detect interrupts before return from handler
    ld        timer_0, g0    # Get timer value and clear XINT0
wait:
    clrbit    0,sf0,sf0      # Attempt to clear bit
    bbs       0,sf0,wait     # Retry if not clear
    ret                      # Return from handler
```

The debounce sampling mode requires that a low level is detected for three consecutive samples before input is detected. For expanded interrupts, all expanded mode pins must be stable for three consecutive samples before the expanded mode vector is resolved internally. The debounce sampling mode provides a built-in filter for noisy or slow-falling inputs.

NOTE

Expanded mode interrupts are always sampled using the debounce sampling mode. This mode provides time for interrupts to trickle through external priority encoders.

Figure 12.5 shows how a signal is sampled in each mode. The debounce-sampling option adds several clocks to an interrupt's latency due to the multiple clocks of sampling. Inputs are sampled internally once every two PCLK cycles.

Interrupt pins are asynchronous inputs. Setup or hold times relative to PCLK2:1 are not needed to ensure proper pin detection. Note in Figure 12.5 that interrupt inputs are sampled once for every two PCLK2:1 cycles. For practical purposes, this means that asynchronous interrupting devices must generate an interrupt signal which is asserted for at least three PLCK2:1 cycles for the fast sampling mode or five PCLK2:1 cycles for the debounce sampling mode.

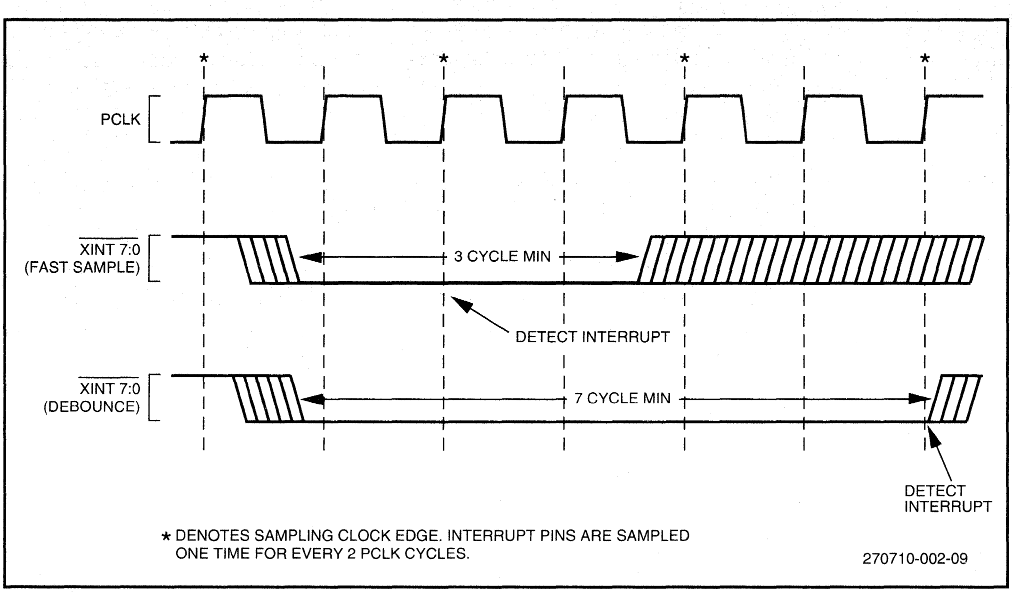


Figure 12.5. Interrupt Sampling

Programmer's Interface

The programmer's interface to the interrupt controller is through four control registers and two special function registers (all described in this section): ICON control register, IMAP0-IMAP2 control registers, IMSK special-function register and IPND special function register.

Interrupt Control Register (ICON)

The ICON register (Figure 12.6) is a 32-bit control register that sets up the interrupt controller. Software can load this register using the **sysctl** instruction. The ICON register is also automatically loaded at initialization from the control table in external memory.

The ICON register's *interrupt mode field* (bits 0 and 1) determine operation mode for the external interrupt pins (XINT0 through XINT7) — dedicated, expanded or mixed.

signal-detection-mode bits (bits 2 through 9) determine whether the signals on the individual external interrupt pins (XINT0 - XINT7) are level-low activated or falling-edge activated. Expanded-mode inputs are always level-detected and NMI input is always edge-detected — regardless of this bit's value.

global-interrupts enable bit (bit 10) globally enables or disables the external interrupt pins and DMA inputs. It does not affect the $\overline{\text{NMI}}$ pin. This bit performs the same function as clearing the mask register.

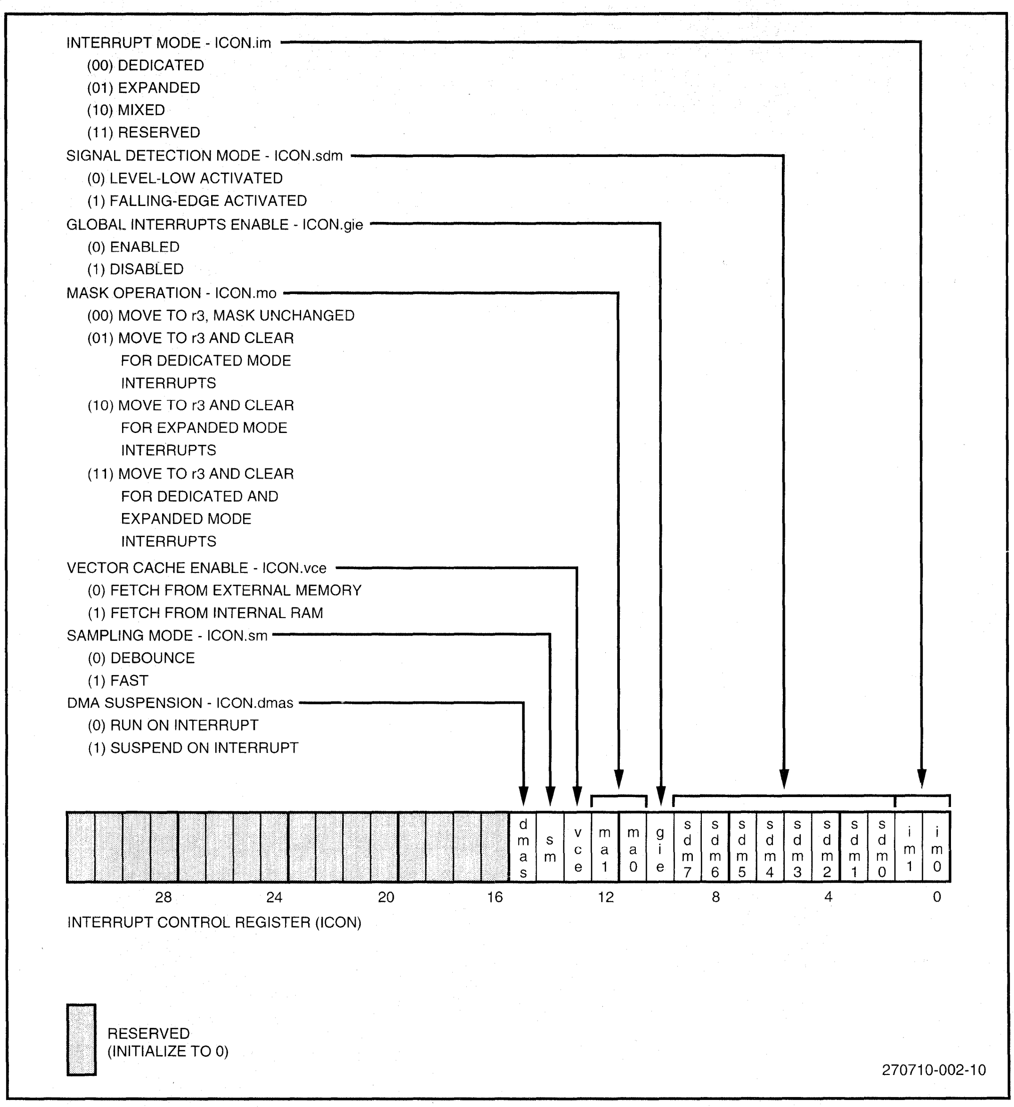
mask-operation field (bits 11 and 12) determines the operation the core performs on the mask register when a hardware-generated interrupt is serviced. On an interrupt, the IMSK register is either unchanged; cleared for dedicated-mode interrupts; cleared for expanded-mode interrupts; or cleared for both dedicated- and expanded-mode interrupts.

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 four least-significant bits equal to 0010_2 may be cached in internal data RAM.

sampling-mode bit (bit 14) determines whether dedicated inputs and $\overline{\text{NMI}}$ pin are sampled using debounce sampling or fast sampling. Expanded-mode inputs are always detected using debounce mode.

DMA-suspension bit (bit 15) determines whether DMA continues running or is suspended while an interrupt procedure is being called.

Bits 16 through 31 are reserved and must be set to 0 at initialization.



INTERRUPT MODE - ICON.im
(00) DEDICATED
(01) EXPANDED
(10) MIXED
(11) RESERVED
SIGNAL DETECTION MODE - ICON.sdm
(0) LEVEL-LOW ACTIVATED
(1) FALLING-EDGE ACTIVATED
GLOBAL INTERRUPTS ENABLE - ICON.gie
(0) ENABLED
(1) DISABLED
MASK OPERATION - ICON.mo
(00) MOVE TO r3, MASK UNCHANGED
(01) MOVE TO r3 AND CLEAR FOR DEDICATED MODE INTERRUPTS
(10) MOVE TO r3 AND CLEAR FOR EXPANDED MODE INTERRUPTS
(11) MOVE TO r3 AND CLEAR FOR DEDICATED AND EXPANDED MODE INTERRUPTS
VECTOR CACHE ENABLE - ICON.vce
(0) FETCH FROM EXTERNAL MEMORY
(1) FETCH FROM INTERNAL RAM
SAMPLING MODE - ICON.sm
(0) DEBOUNCE
(1) FAST
DMA SUSPENSION - ICON.dmas
(0) RUN ON INTERRUPT
(1) SUSPEND ON INTERRUPT
dmas
sm
vce
ma1
ma0
gie
sdm7
sdm6
sdm5
sdm4
sdm3
sdm2
sdm1
sdm0
im1
im0
28
24
20
16
12
8
4
0
INTERRUPT CONTROL REGISTER (ICON)
RESERVED (INITIALIZE TO 0)
270710-002-10

Figure 12.6. Interrupt Control (ICON) Register

Interrupt Mapping Registers (IMAP0-IMAP2)

The IMAP registers (Figure 12.7) are three 32-bit registers (IMAP0 through IMAP2). These register's bits 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 DMA-interrupt inputs (four bits per input).

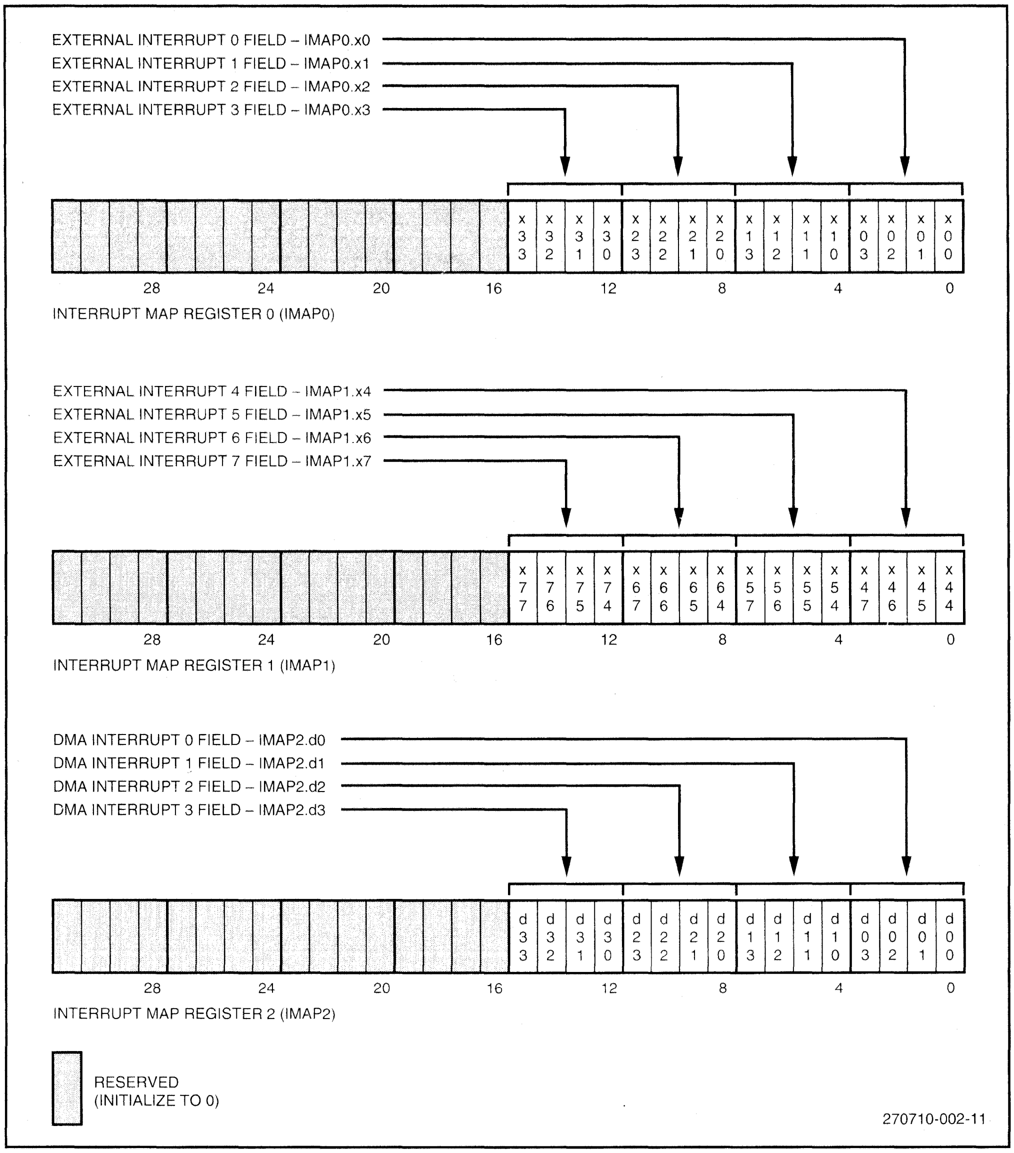

EXTERNAL INTERRUPT 0 FIELD – IMAP0.x0
EXTERNAL INTERRUPT 1 FIELD – IMAP0.x1
EXTERNAL INTERRUPT 2 FIELD – IMAP0.x2
EXTERNAL INTERRUPT 3 FIELD – IMAP0.x3
x33 x32 x31 x30 x23 x22 x21 x20 x13 x12 x11 x10 x03 x02 x01 x00
28 24 20 16 12 8 4 0
INTERRUPT MAP REGISTER 0 (IMAP0)
EXTERNAL INTERRUPT 4 FIELD – IMAP1.x4
EXTERNAL INTERRUPT 5 FIELD – IMAP1.x5
EXTERNAL INTERRUPT 6 FIELD – IMAP1.x6
EXTERNAL INTERRUPT 7 FIELD – IMAP1.x7
x77 x76 x75 x74 x67 x66 x65 x64 x57 x56 x55 x54 x47 x46 x45 x44
28 24 20 16 12 8 4 0
INTERRUPT MAP REGISTER 1 (IMAP1)
DMA INTERRUPT 0 FIELD – IMAP2.d0
DMA INTERRUPT 1 FIELD – IMAP2.d1
DMA INTERRUPT 2 FIELD – IMAP2.d2
DMA INTERRUPT 3 FIELD – IMAP2.d3
d33 d32 d31 d30 d23 d22 d21 d20 d13 d12 d11 d10 d03 d02 d01 d00
28 24 20 16 12 8 4 0
INTERRUPT MAP REGISTER 2 (IMAP2)
RESERVED
(INITIALIZE TO 0)
270710-002-11

Figure 12.7. Interrupt Mapping (IMAP2-IMAP0) Registers

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 load the mapping registers using the **sysctl** instruction. The mapping registers are also automatically loaded at initialization from the control table in external memory. Note that bits 16 through 31 of each register are reserved and should be set to 0 at initialization.

Interrupt Mask and Pending Registers (IMSK, IPND)

The IMSK and IPND registers (Figure 12.8) are special-function registers (sf1 and sf0, respectively). Bits 0 through 7 of these registers are associated with the external interrupt pins (XINT0 through XINT7) and bits 8 through 11 are associated with the DMA-interrupt inputs (DMA0 through DMA3). Bits 12 through 31 are reserved and should be set to 0 at initialization.

The IPND register posts dedicated-mode interrupts originating from the eight external dedicated sources (when configured in dedicated mode) and the four DMA 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 the dedicated mode and it globally masks all expanded-mode interrupts in the expanded and mixed modes. In expanded mode, bits 1 through 7 are not used and should only contain zeros; in mixed mode, bits 1 through 4 are not used and should only contain zeros.

Software can read and write the IPND and IMSK registers, using any instruction that can use special-function registers as operands.

When the 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 handler is executed.

An alternative method of posting interrupts in the IPND register (other than through the external interrupt pins and DMA-interrupt inputs) is to set bits in the register directly using an instruction — such as a move instruction. This operation has the same effect as requesting an interrupt through the external interrupt pins or DMA-interrupt inputs. The bit set in the IPND register must be associated with an interrupt source that is programmed for dedicated-mode operation.

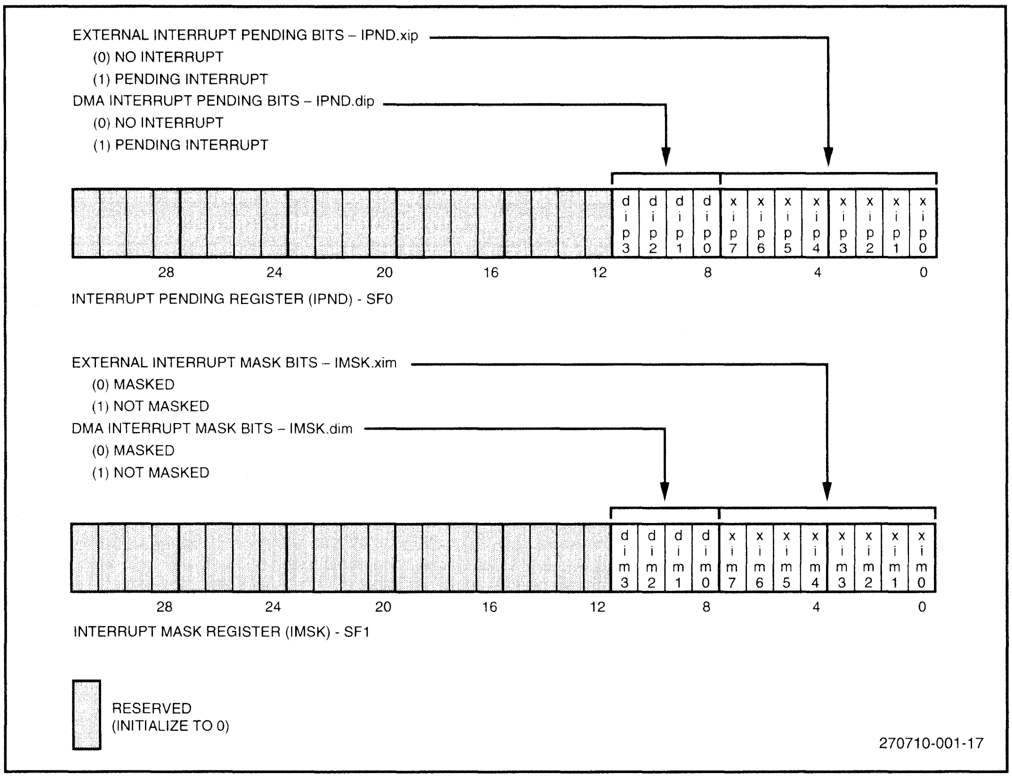

EXTERNAL INTERRUPT PENDING BITS – IPND.xip
(0) NO INTERRUPT
(1) PENDING INTERRUPT
DMA INTERRUPT PENDING BITS – IPND.dip
(0) NO INTERRUPT
(1) PENDING INTERRUPT
dip3 dip2 dip1 dip0 xip7 xip6 xip5 xip4 xip3 xip2 xip1 xip0
28 24 20 16 12 8 4 0
INTERRUPT PENDING REGISTER (IPND) - SF0
EXTERNAL INTERRUPT MASK BITS – IMSK.xim
(0) MASKED
(1) NOT MASKED
DMA INTERRUPT MASK BITS – IMSK.dim
(0) MASKED
(1) NOT MASKED
dim3 dim2 dim1 dim0 xim7 xim6 xim5 xim4 xim3 xim2 xim1 xim0
28 24 20 16 12 8 4 0
INTERRUPT MASK REGISTER (IMSK) - SF1
RESERVED
(INITIALIZE TO 0)
270710-001-17

Figure 12.8. Interrupt Mask (IMSK) and Interrupt Pending (IPND) Registers

Default and Reset Register Values

The ICON and IMAP2-0 control registers are loaded from the control table in external memory when the processor is initialized or reinitialized. (Control table is described in *Chapter 2, Programming Environment.*) The IMSK register is set to 0 when the processor is initialized (RESET is deasserted). IPND register value is undefined after a power-up initialization (cold reset). The user is responsible for clearing this register before any mask register bits are set; otherwise, unwanted interrupts may be triggered. For a reset while power is on (warm reset), the pending register value is retained.

Setting Up the Interrupt Controller

This section provides several examples of setting up the interrupt controller. Recall that the IMAP and ICON registers are control registers. When the entire control table is automatically read at initialization, the ICON and IMAP registers are loaded with the values pre-programmed in the table. In many applications, setting these register values in the initial control table is the only setup required. The following examples describe how the interrupt controller can be dynamically configured after initialization.

Example 12.2 sets up the interrupt controller for expanded-mode operation. Here, a value which selects expanded-mode operation is loaded into the ICON register. The **sysctl** instruction is issued with the load-control register message type (03H) and selecting group number 01H from the control table. Group 01H contains the ICON and IMAP registers. Note that the IMAP registers, as well as the ICON register, are reloaded with this operation.

Modifying the control table implies that the table — or part of the table — must reside in RAM. If the control registers are modified after initialization, the control register must be relocated to RAM by reinitializing the processor. (See *Chapter 14, Initialization and System Requirements* for a description of relocating data structures after initialization.)

Example 12.2. Programming the Interrupt Controller for Expanded Mode

```
# Example expanded mode setup . . .
mov         0,sf1
ldconst     0x01, g0                  # clear IMSK register
                                      # (mask all interrupts)
st          g0,ctrl_table_ICON        # store mode information to
                                      # control table
ldconst     0x401,r4                  # create operand for sysctl,
                                      # selects load control
                                      # register message type,
                                      # selects register group 1
sysctl      r4, r4, r4                # load control register
mov         1,sf1                     # unmask expanded interrupts
```

Implementation

The interrupt controller, microcode and core resources handle all stages of interrupt service. Interrupt service is handled in the following stages:

Request Interrupt — In the i960 CA processor, the programmable on-chip interrupt controller transparently manages all interrupt requests. Interrupts are generated by hardware (external events) or software (the user program). Hardware requests are signaled on the 8-bit external interrupt port ($\overline{\text{XINT7:0}}$), the non-maskable interrupt pin ($\overline{\text{NMI}}$) or the four DMA controller 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*. An interrupt, once posted, 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) special function register
- software interrupts are posted by setting the interrupt's assigned bit in the interrupt table's pending priorities and pending interrupts fields

Check Pending Interrupts – Interrupts posted for later service must be compared to 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 12.9 illustrates interrupt controller function. For best performance, the interrupt flow for hardware interrupt sources is implemented entirely in hardware.

The comparator only signals the core when a posted interrupt is a lower priority than the process priority. Because the comparator function is implemented in hardware, microcode cycles are never consumed unless an interrupt is serviced.

Interrupt Service Latency

The time required to perform an interrupt task switch is referred to 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. In the following discussion, interrupt service latency is derived in number of PCLK2:1 cycles. The established measure of interrupt service latency (in units of seconds) is derived with the following equation:

$$\text{Interrupt Service Latency (in seconds)} = \frac{N_{L\_int}}{f_c}$$

where: f_c = PCLK2:1 frequency (Hz)

N_{L_int} = number of PCLK2:1 cycles

For real-time applications, worst-case interrupt latency must be considered for critical handling of external events. For example, an interrupt from a FIFO buffer may need service to prevent the FIFO from overrun.

For many applications, typical interrupt latency must be considered in determining overall system performance. For example, a timer interrupt may frequently trigger a task switch in a multi-tasking kernel.

The flow chart in Figure 12.9 is used to determine worst-case interrupt latency, based on the specifics of a system. The values from Figure 12.9 are based on the assumption that the interrupt controller is configured in the following way:

- Hardware interrupt is requested ($\overline{\text{XINT7:0}}$ pins or $\overline{\text{NMI}}$)
- Fast sample mode - Fast sample mode is selected (ICON.sm=1)
- Cached interrupt vector - Interrupt vector is fetched from internal data RAM. This is automatic for the $\overline{\text{NMI}}$ vector or is selected in the ICON register (ICON.vce=1)
- Cached interrupt handler - Cache hit for interrupt call target
- DMA suspended on interrupt - DMA suspend on interrupt is enabled (ICON.dmas=1)
- Minimum Bus Latency - All memory in the system is configured as zero wait state and burst access mode

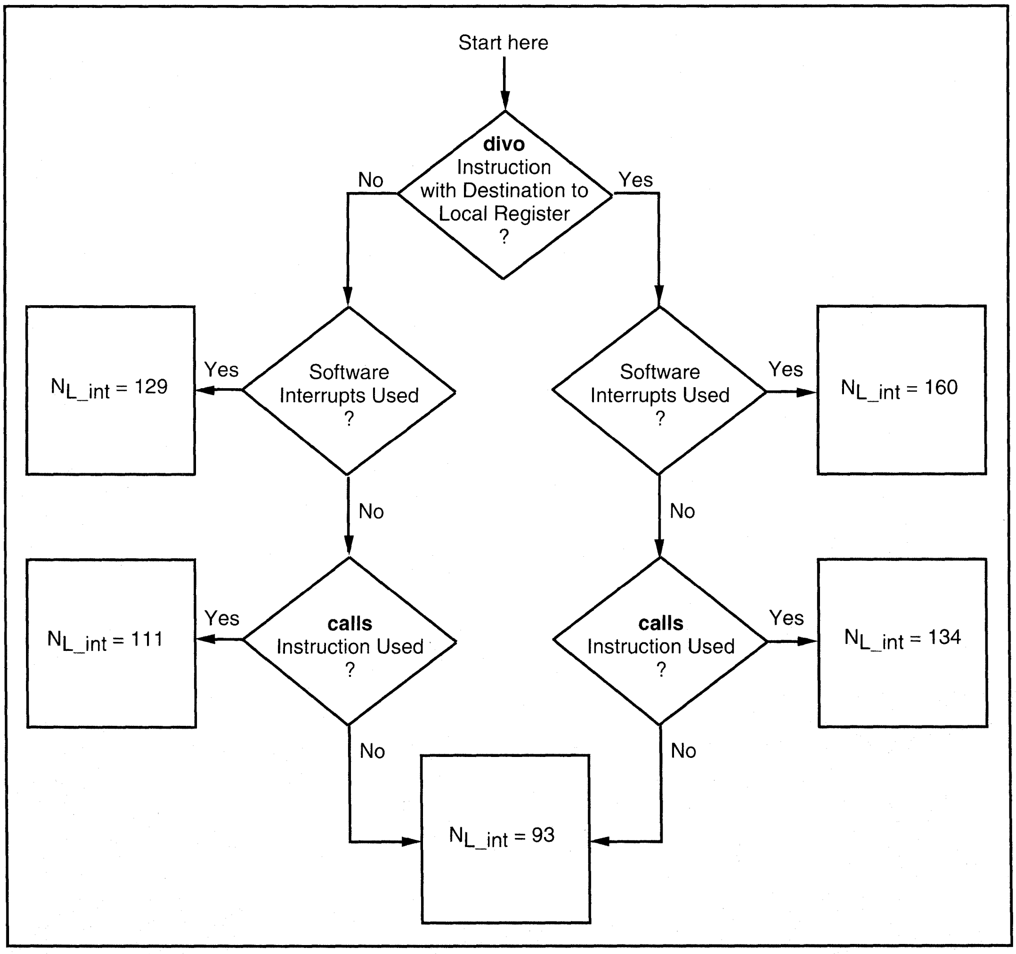


Figure 12.9. Calculation of worst case interrupt latency - N_{L_int}

NOTE

The worst-case interrupt latency value does not account for interaction of faults and interrupts. It is assumed that faults are not signaled in a stable system.

Because of the processor's instruction mix and the nature of on-chip register cache, typical interrupt latency is derived assuming that the interrupt occurs under the following constraints, in addition to those listed above:

- Interrupts a single cycle RISC instruction
- Frame flush does not occur
- Bus queue is empty

The value for typical interrupt latency (N_{L_int}) is: N_{L_int} (typical) = 30 PCLK2:1 cycles

Optimizing Interrupt Performance

The i960 CA processor provides several features aimed at reducing the time required to respond to and service interrupts. The following section describes three methods for reducing interrupt latency:

- caching interrupt vectors on-chip
- DMA suspension while servicing interrupts
- caching of interrupt handling procedure code

Figure 12.9 shows that controlling the use of long instructions may also be used to optimize interrupt performance.

Vector Caching Option

To reduce interrupt latency, the i960 CA processor allows some interrupt table vector entries to be cached in internal data RAM. When the caching option is selected, all interrupts with a vector number with the four least-significant bits equal to 0010_2 are cached. When the vector option is enabled and an interrupt request is received for one of these interrupts, the controller fetches the associated vector from internal RAM rather than from the interrupt table in memory. This option is selected when programming the ICON register.

NOTE

To use the caching feature described in this section, software must explicitly store the vector entries in internal RAM.

Since the internal RAM is mapped directly to the address space, this operation can be performed using the core's store instructions. Table 12.1 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 12.1 (reminder: 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.

Vectors that can be cached coincide with the vector numbers that can be selected with the mapping registers and assigned to dedicated-mode inputs.

Table 12.1. Location of Cached Vectors in Internal RAM

Interrupt/NMI Vector Number		Internal RAM Address
NMI	(248)	0000H
$0001\ 0010_2$	(18)	0004H
$0010\ 0010_2$	(34)	0008H
$0011\ 0010_2$	(50)	000CH
$0100\ 0010_2$	(66)	0010H
$0101\ 0010_2$	(82)	0014H
$0110\ 0010_2$	(98)	0018H
$0111\ 0010_2$	(114)	001CH
$1000\ 0010_2$	(130)	0020H
$1001\ 0010_2$	(146)	0024H
$1010\ 0010_2$	(162)	0028H
$1011\ 0010_2$	(178)	002CH
$1100\ 0010_2$	(194)	0030H
$1101\ 0010_2$	(210)	0034H
$1110\ 0010_2$	(226)	0038H
$1111\ 0010_2$	(242)	003CH

DMA Suspension on Interrupts

Core resources required to execute a DMA operation may affect interrupt latency. A DMA operation may be temporarily suspended to reduce the effects of the DMA when interrupt-response time is critical. The DMA suspension option is programmed in the ICON register. When the option is selected, the core suspends DMA processing while executing a call to an interrupt-handling procedure for a hardware-requested interrupt. Once the core begins executing the interrupt procedure, it restores DMA processing.

To improve interrupt throughput, DMA processing can be suspended until the execution of an interrupt-handling procedure is complete. To accomplish this, the interrupt procedure must explicitly suspend DMA operation by clearing the DMA command (DMAC) register's channel enable field. (See *Chapter 13, DMA Controller* for more information.)

Caching of Interrupt-Handling Procedures

The time required to fetch the first instructions of an interrupt-handling procedure affects interrupt response time and throughput. The controller allows this fetch time to be reduced by caching interrupt procedures — or portions of procedures — in the i960 CA processor's instruction cache. Paragraphs that follow describe this caching of interrupt procedures.

Instruction cache is divided into two 512-byte halves (Figure 12.10). One or both halves can be used for storing interrupt-procedure code. Typically, one half is used as normal instruction cache and the other half for caching interrupt procedures.

The interrupt-handling procedure sections to be cached must be placed in a contiguous memory block. The last instruction for each procedure in this block must be a return from the interrupt procedure or a branch to the remainder of the procedure, located in another area of address space. Maximum block size is 512 or 1024 bytes, depending on how the instruction cache is to be configured.

The **sysctl** instruction provides the mechanism for loading and locking this block of interrupt procedures into the instruction cache. **sysctl** is issued with the configure instruction cache message type. The address of the block of interrupt procedures in memory is specified as an operand of the instruction.

The interrupt vector's two least-significant bits must be set to 10_2 to fetch the interrupt procedure from locked cache rather than the normal memory hierarchy. The procedure executed if it is in the cache. If a miss at the locked cache occurs, the interrupt procedure is executed from the normal memory hierarchy (see *Chapter 2, Programming Environment* for **sysctl** information and how to configure instruction cache load-and-lock features).

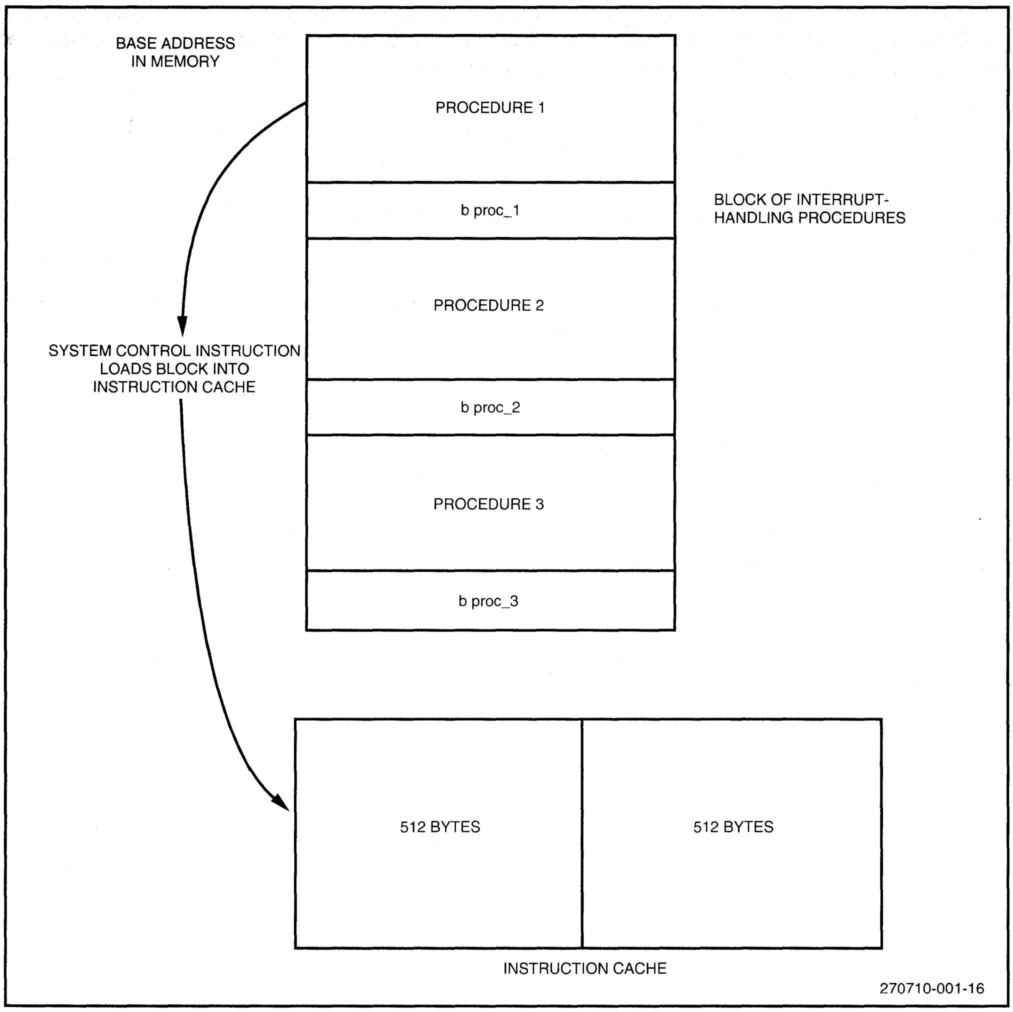


Figure 12.10. Caching Interrupt-Handling Code

DMA Controller 13

CHAPTER 13
DMA CONTROLLER

This chapter describes the i960 CA processor's integrated Direct Memory Access (DMA) Controller, including: operation modes, setup, external interface and DMA controller implementation.

OVERVIEW

The DMA controller concurrently manages up to four independent DMA channels. Each channel supports memory-to-memory transfers where the source and destination can be any combination of internal data RAM or external memory. The DMA mechanism provides two unique methods for performing DMA transfers:

- Demand-mode transfers (synchronized to external hardware). Typically used for transfers between an external device and memory. In demand mode, external hardware signals for each channel are provided to synchronize DMA transfers with external requesting devices.
- Block-mode transfers (non-synchronized). Typically used to move blocks of data within memory.

To perform a DMA operation, the DMA controller uses microcode, the core's multi-process resources, the bus controller and internal hardware dedicated to the DMA controller. Loads and stores are executed in DMA microcode to perform each DMA transfer. The bus controller, directed by DMA microcode, handles data transactions in external memory. DMA controller hardware synchronizes transfers with external devices or memory, provides the programmers interface to the DMA controller and manages the priority for servicing the four DMA channels.

The DMA controller uses multi-process resources, designed into the core, to enable DMA operations to execute in microcode concurrently with the user's program. This sharing of core resources is accomplished with hardware-implemented processes for each of the four DMA channels (the *DMA processes*) and a separate process for the user's program (the *user process*). Alternating between DMA processes and the user process enables a user's program and up to four DMAs (one per channel) to run at the same time.

To execute a DMA operation, a DMA process issues memory load or store requests. The bus controller executes these memory processes as it would a load, store or prefetch request from the user process. External bus access is shared equally between the user and DMA process. The bus controller executes bus requests by each process in alternating fashion.

The DMA controller is configurable to best exploit the core's processing capabilities and external bus performance. Source and destination request lengths are programmed for each DMA channel. Based on request length, the DMA controller optimizes transfer performance between source and destination with different external data bus widths. A DMA can be programmed for quad-word transfers, taking best advantage of external bus burst capabilities. The DMA controller can also efficiently execute transfers of unaligned data.

A single cycle "fly-by" transfer mode gives the highest performance transfers for a DMA. In this mode, a single bus request executes a transfer of data from source to destination.

A data-chaining mode simplifies several commonly-performed DMA operations such as scatter or gather. Data-chained DMAs are configured with a series of descriptors in memory. Each descriptor describes the transfer of a single buffer or portion of the entire DMA. These descriptors can be dynamically changed as the chained DMA progresses.

DMA setup and control is simple and efficient. The setup DMA (**sdma**) instruction sets up a DMA operation. **sdma** specifies addressing, transfer type and DMA modes. A special-function register — the DMA command register (DMAC) — is an interface for commonly-used command and status functions for each channel.

Flexibility and a high degree of programmability for a DMA operation create a number of options for balancing DMA and processor performance and DMA latency. This flexibility enables the programmer to select the best DMA configuration for a particular application.

DEMAND AND BLOCK MODE DMA

A channel can be configured as a demand mode or block mode DMA channel. Demand mode DMAs move data between memory and an external I/O device; block mode DMAs typically move blocks of data from memory to memory.

When a channel is configured for demand mode, an external device requests a DMA transfer with a DMA request input ($\overline{\text{DREQ3:0}}$). The DMA controller acknowledges the requesting device with a DMA acknowledge signal ($\overline{\text{DACK3:0}}$). The $\overline{\text{DACK3:0}}$ signal is asserted during the bus request which the DMA controller makes to the requesting device. The specific timing of the $\overline{\text{DREQ3:0}}$ and $\overline{\text{DACK3:0}}$ signals is described later in this chapter's section titled *DMA External Interface*.

After a DMA channel is configured the channel must be enabled by software through the DMA control register (DMAC). The DMA operation continues until it is (1) terminated (by an external source with $\overline{\text{EOP}}$), (2) suspended (by software), (3) ends because of a zero byte count. An interrupt may be generated to detect any of these three cases.

SOURCE AND DESTINATION ADDRESSING

When a DMA operation is set up, it is described with a source address, destination address and byte count. For each channel, an address is either held fixed or incremented after each transfer. A fixed address is used for addressing external I/O devices; an address which increments is used for the memory side of a DMA transfer. When a channel is set up, address increment or hold is selected separately for the source and destination address.

Source and destination address and byte count are 32-bit values. Source and destination are byte addressable over the entire address space. DMA operation length can be up to 4 Gbytes (2^{32} Bytes). Source and destination address and byte count are specified when **sdma** executes.

DMA TRANSFERS

The following sections explain DMA transfer characteristics, especially those transfer characteristics affected by channel setup. Intelligent selection of transfer characteristics works to balance DMA performance and functionality with the performance of the user's program.

Source/destination request length selects the bus request types which the DMA microcode issues when executing a DMA transfer. To perform a transfer, combinations of byte, short-word, word and quad-word load and store requests are issued. Refer to *Chapter 11, External Bus Description* for a detailed description of *bus request.*

transfer type is specified when a channel is set up using **sdma**. Transfer type specifies *source/destination request length* for a DMA operation and whether DMA transfer is performed as a *multiple - cycle transfer* or as a *fly-by (1 bus cycle) transfer.*

Multi-cycle transfer is performed with two or more bus requests; fly-by transfer with a single bus request. Fly-by and multi-cycle transfers are described in the following sections.

Table 13.1. Transfer Type Options

Source Request Length	Destination Request Length	Transfer Type
Byte (8 bits)	Byte (8 bits)	Multi-Cycle
Byte (8 bits)	Byte (8 bits)	Fly-by
Byte (8 bits)	Short (16 bits)	Multi-Cycle
Byte (8 bits)	Word (32 bits)	Multi-Cycle
Short (16 bits)	Byte (8 bits)	Multi-Cycle
Short (16 bits)	Short (16 bits)	Multi-Cycle
Short (16 bits)	Short (16 bits)	Fly-by
Short (16 bits)	Word (32 bits)	Multi-Cycle
Word (32 bits)	Byte (8 bits)	Multi-Cycle
Word (32 bits)	Short (16 bits)	Multi-Cycle
Word (32 bits)	Word (32 bits)	Multi-Cycle
Word (32 bits)	Word (32 bits)	Fly-by
Quad-Word (128 bits)	Quad-Word (128 bits)	Multi-Cycle
Quad-Word (128 bits)	Quad-Word (128 bits)	Fly-by

Multi-Cycle Transfers

Multi-cycle DMA transfer comprises two or more bus requests. For these multi-cycle transfers, loads from a source address are followed by stores to a destination address. To execute the transfer, DMA microcode issues the proper combination of bus requests; for example, a typical multi-cycle DMA transfer could appear as a single byte load request followed by a single byte store request.

For a multi-cycle transfer, source data is first loaded into on-chip DMA registers before it is stored to the destination. The processor effectively buffers the data for each transfer. When a DMA transfer is configured for destination synchronization, the DMA controller buffers source data, waiting for the request (active $\overline{\text{DREQ3:0}}$ signal) from the destination requestor. This operation reduces latency. The initial DMA request, however, still requires the source data to be loaded before the request is acknowledged. Source data buffering is shown in Figure 13.1. The DMA controller does not perform multi-cycle transfers atomically. A DMA transfer does not cause the processor's $\overline{\text{LOCK}}$ output to be asserted. A bus hold request may also be acknowledged between the bus requests which make up a multi-cycle transfer.

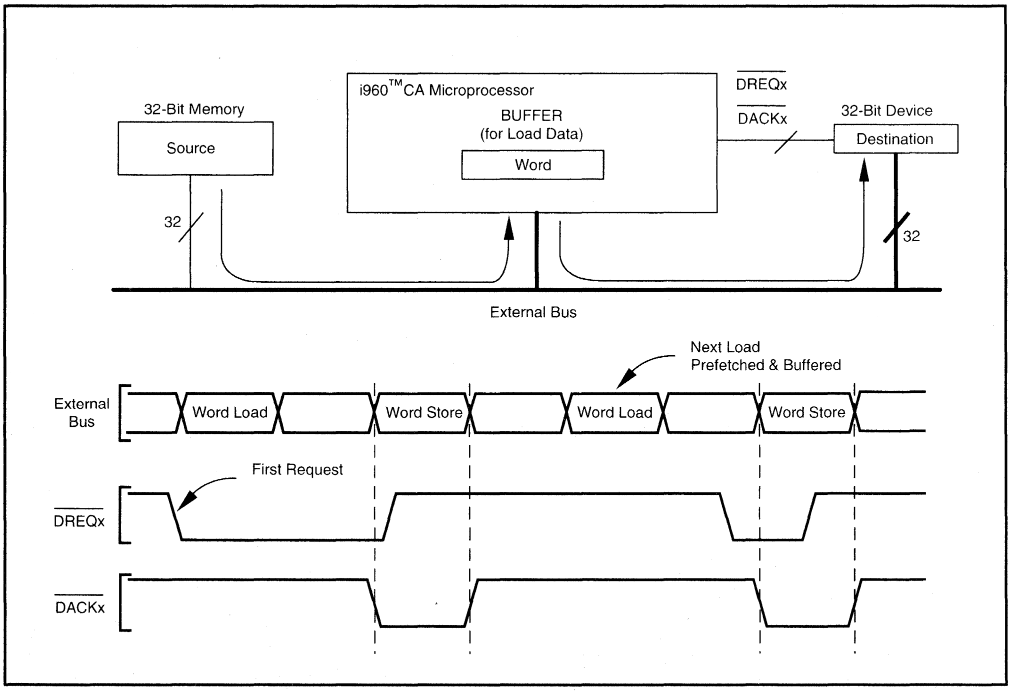


Figure 13.1. Source Data Buffering for Destination Synchronized DMAs

Fly-By Single-Cycle Transfers

Fly-by transfers are executed with only a single load or store request. Source data is not buffered internally; instead, the data passes directly between source and destination via the external data bus.

Fly-by transfers are commonly used for high-performance peripheral to memory transfers. The fly-by mechanism is best described by giving an example of a source-synchronized demand mode DMA (Figure 13.2). In the example, a peripheral at a fixed address is the source of a DMA and memory is the destination. Each transfer is synchronized with the source.

The source requests a transfer by asserting the request pin ($\overline{\text{DREQ3:0}}$). When the request is serviced, a store is issued to the destination memory while the requesting device is selected by the DMA acknowledge pin ($\overline{\text{DACK3:0}}$). The source device, when selected, must drive the data bus for the store instead of the processor. (The processor floats the data bus for a fly-by transfer.)

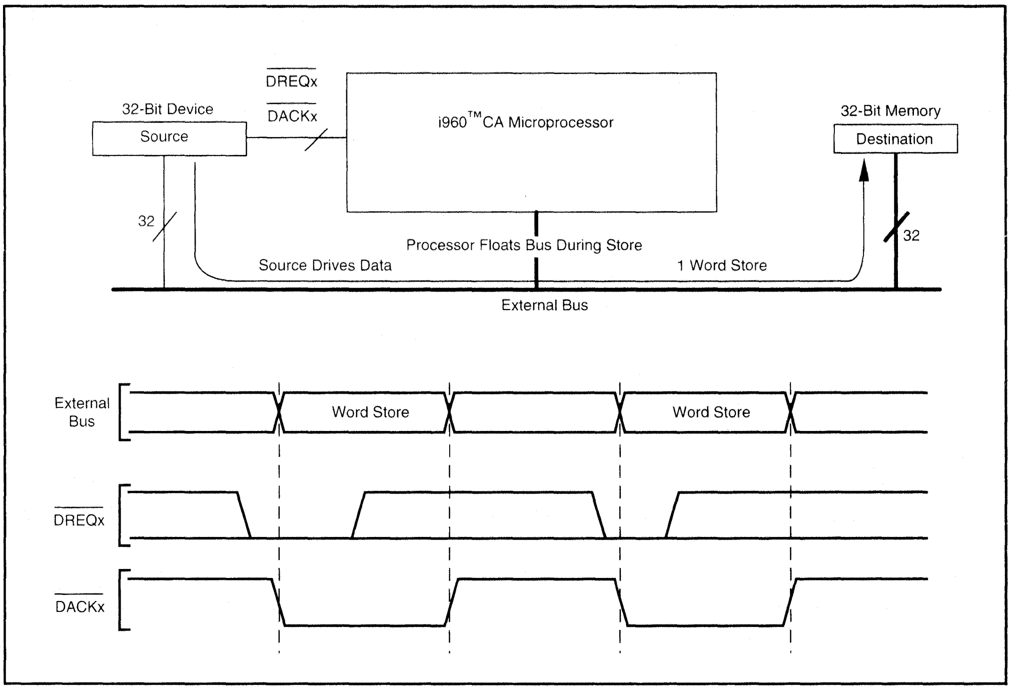


Figure 13.2. Example of Source Synchronized Fly-by DMA

If the destination of a fly-by is the requestor (destination synchronization), a load is issued to the source while the destination is selected with the acknowledge pin. The destination, when selected, reads the load data; the processor ignores the data from the load.

NOTE

Fly-by mode may not access internal data RAM.

A fly-by DMA in block mode is started by software like any block-mode operation. Request pins $\overline{\text{DREQ3:0}}$ are ignored in block mode. Fly-by block-mode DMAs can be used to implement high-performance memory-to-memory transfers where source and destination addresses are fixed at block boundaries. In this case, the acknowledge pin must be used in conjunction with external hardware to uniquely address the source and destination for the transfer.

Source/Destination Request Length

Source and destination request length is selected when a DMA channel is configured. Request length determines bus request types that the DMA microcode issues. Byte, short-word or quad-word bus requests are issued by the DMA controller microcode.

The request length selected for a DMA operation — byte, short-word, word or quad-word — should not be confused with external data-bus width or other characteristics programmed in the memory-region configuration table. Request length dictates the type of bus request issued by DMA controller microcode, while the region configuration of a DMA's source and destination memory control how that bus request is executed on the external bus.

As an example, consider a system in which a DMA source memory region is configured for 8-bit, non-burst accesses and a word source request length is selected. DMA microcode issues word loads (identical to the **ld** instruction) to DMA addresses in the source region. Since the source memory region is configured as 8 bits, the bus controller handles the word loads as four 8-bit accesses in that region. To contrast this example, if the DMA is configured for a byte source request length, DMA microcode issues byte loads (identical to the **ldob** instruction) to DMA addresses in the source region. The byte load to this region is executed as a single 8-bit access. *Chapter 11, External Bus Description* fully describes bus configuration and how the bus controller executes bus requests.

In demand mode transfers, $\overline{\text{DREQ3:0}}$ is asserted to request a DMA transfer. $\overline{\text{DACK3:0}}$ is asserted during the bus request issued in response to the DMA request. Continuing the example started above: if the DMA controller is set up for source synchronized demand mode, $\overline{\text{DREQ3:0}}$ causes a word (**ld**) request to be issued when source request length equals word and causes a byte (**ldob**) request to be issued when the source request length equals byte. $\overline{\text{DACK3:0}}$ is asserted for the duration of the bus request for each case.

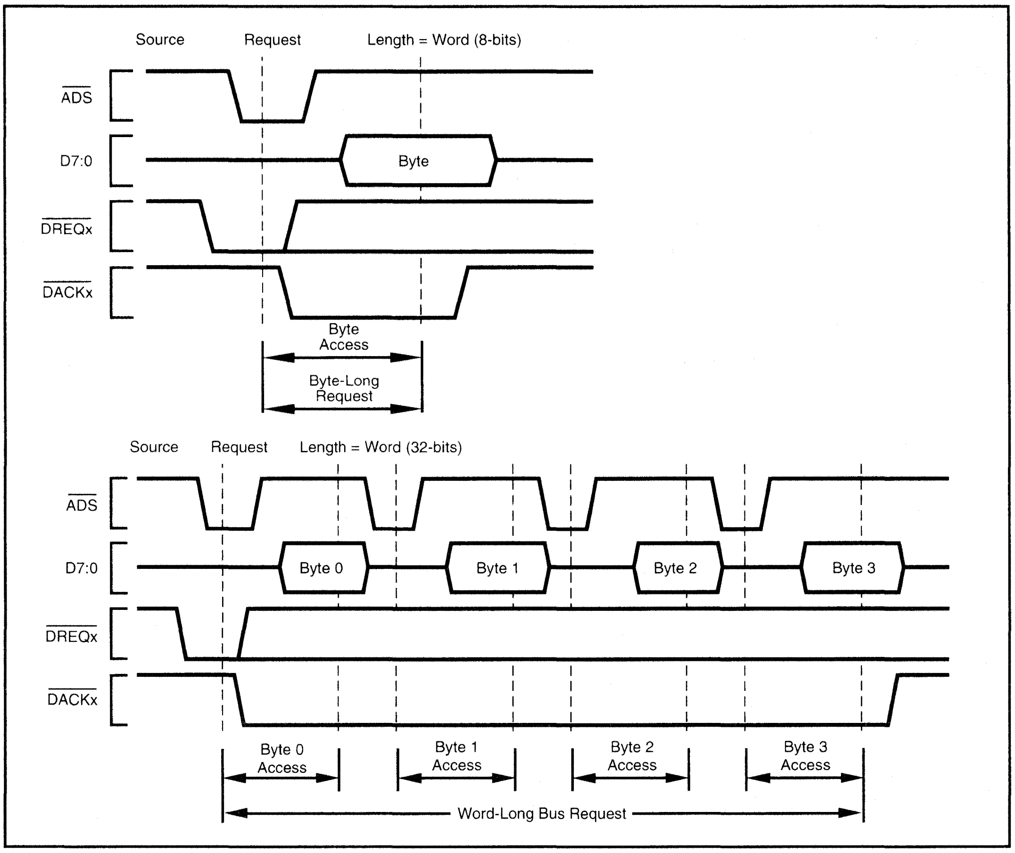


Figure 13.3. Source Synchronized DMA Loads from an 8-bit, Non-burst, Non-pipelined Memory Region

For demand mode transfers, the request length is typically selected to match the external bus width of the external DMA device. If request length is greater than bus width, the DMA device must be designed to support multiple data cycles for each DMA transfer requested. This may be accomplished by using a small FIFO and an external circuit to load and unload the FIFO. This method reduces bus loading by the DMA process.

For block mode transfers, source and destination request lengths are typically selected to match external data bus width. This configuration uses the external bus most efficiently and also reduces latency for bus requests issued by the user process.

In instances where source and destination bus widths are different, DMA performance may be increased by setting up the DMA with matching source and destination request lengths. This configuration reduces DMA microcode overhead required to pack or unpack data between unequal request lengths. Packing/unpacking is handled more efficiently by the bus controller unit. Matching the request lengths may increase latency for bus requests issued by the user process.

Quad-word source and destination request lengths are used for highest DMA performance. Quad transfers use the external bus most efficiently when the source or destination memory regions support burst accesses. Since the request length for quad word transfers is always greater than the bus width, DMA devices must support multiple data cycles for each requested DMA transfer. Using quad-word request lengths may increase bus latency for loads, stores and instruction fetches that the user's program generates.

In cases where source address, destination address or byte count are unaligned, requests shorter than the selected request length are issued to align the transfers. (Refer to the section in this chapter titled *Data Alignment.*)

Assembly and Disassembly

The DMA controller internally assembles or disassembles data between different source and destination request lengths. Assembly refers to the packing of narrow data into wider data. Disassembly refers to the unpacking of wide data into narrow data. Assembly and disassembly is performed automatically when a channel is set up with different source and destination request lengths. Assembly and disassembly is performed for all aligned transfers configured with combinations of byte, short-word and word request lengths. Quad-word DMA transfers require that source and destination request lengths equal quad word; therefore, data assembly and disassembly is not applicable to this DMA mode.

Figure 13.4 shows a typical demand mode configuration in which an 8-bit device is the source requestor for a DMA and 32-bit memory is the destination. If byte source and word destination request length is selected for this DMA, data from four source requests is buffered before load to the 32-bit memory is executed. This configuration represents an optimal use of bus resources for a DMA between an 8-bit device and 32-bit memory.

NOTE

Microcode algorithms which perform assembly and disassembly are less efficient than algorithms which perform transfers between source and destination with equal request lengths. DMA controller assembly and disassembly is provided for convenience and for most efficient external bus usage. For example, the system shown in Figure 13.4 functions the same when source and destination request lengths are both byte-long. In this case, each transfer is performed with a byte load followed by a byte store. DMA throughput is increased; however, the DMA makes more bus requests to transfer the same amount of data.

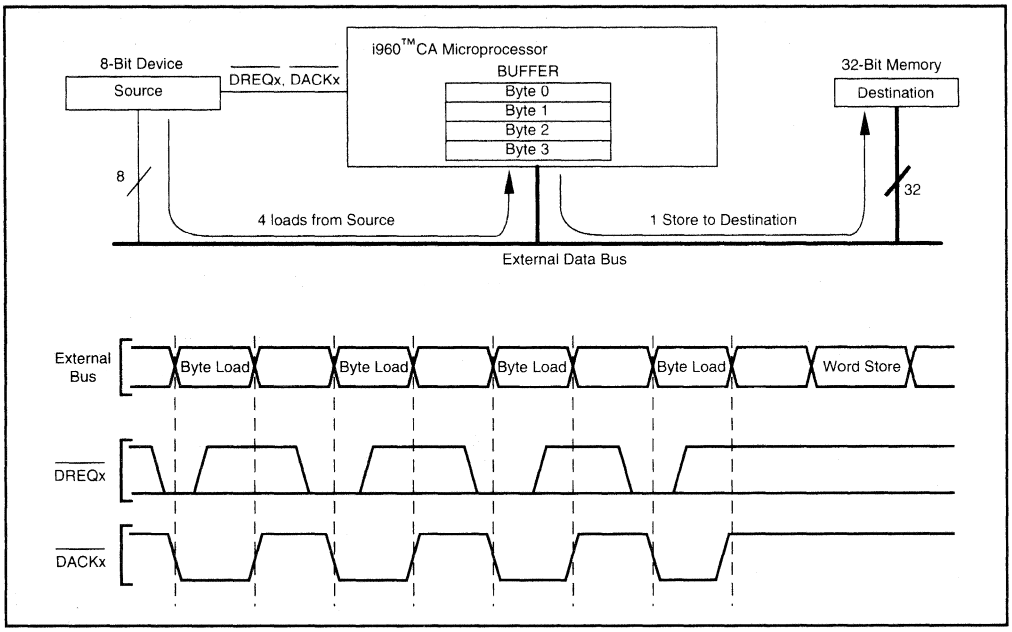


Figure 13.4. Byte to Word Assembly

Data Alignment

The DMA controller is able to transfer data to and from unaligned memory blocks. A DMA channel's source or destination address may be set up to increment for memory block transfers. When the address increments, there are no alignment requirements for byte, short or word-long request lengths. Addresses for quad-word request lengths must always be quad-word aligned. To interface to external DMA devices, the source or destination address may be set up as fixed. Fixed addresses must always be aligned to the request-length boundary. Table 13.2 summarizes the alignment requirements for all DMA transfers.

The minimum byte count depends on the configuration of the DMA controller:

Configuration	Minimum byte count
Multi-cycle block mode with byte, short-word or word long source or destination request length	1
Multi-cycle block mode with quad-word request length	16
Multi-cycle source sync. demand mode	source request length (bits) / 8
Multi-cycle destination sync. demand mode	destination request length (bits) / 8
All fly-by mode transfers	request length (bits) / 8

Multi-cycle DMAs to aligned memory blocks perform better than DMAs to unaligned memory blocks. Additional microcode cycles are required to access the unaligned memory.

Most unaligned DMA transfers, however, use the external bus almost as efficiently as aligned DMAs. Multi-cycle DMA configurations which use the bus efficiently when memory blocks are unaligned are:

Word-to-Word	Byte-to-Short
Byte-to-Word	Short-to-Byte
Word-to-Byte	

Table 13.2. DMA Transfer Alignment Requirements

Transfer Types (Source-to-Destination)	Boundary Alignment Requirements			
	Source Address or Fly-by Address		Destination Address	
	Fixed	Incr.	Fixed	Incr.
Byte-to-Byte (8/8 bit)				
Multi-cycle	Byte	Byte	Byte	Byte
Fly-by	Byte	Byte	N/A	N/A
Byte-to-Short (8/16 bit)				
Multi-cycle	Byte	Byte	Short	Byte
Byte-to-Word (8/32 bit)				
Multi-cycle	Byte	Byte	Word	Byte
Short-to-Byte (16/8 bit)				
Multi-cycle	Short	Byte	Byte	Byte
Short-to-Short (16/16 bit)				
Multi-cycle	Short	Byte	Short	Byte
Fly-by	Short	Short	N/A	N/A
Short-to-Word (16/32 bit)				
Multi-cycle	Short	Byte	Word	Byte
Word-to-Byte (32/8 bit)				
Multi-cycle	Word	Byte	Byte	Byte
Word-to-Short (32/16 bit)				
Multi-cycle	Word	Byte	Short	Byte
Word-to-Word (32/32 bit)				
Multi-cycle	Word	Byte	Word	Byte
Fly-by	Word	Word	N/A	N/A
Quad-to-Quad (128/128 bit)				
Multi-cycle	Quad	Quad	Quad	Quad
Fly-by	Quad	Quad	N/A	N/A

These optimized unaligned transfers are executed by performing byte requests until alignment is enforced. At this time, aligned source and destination requests are executed. At end of transfer, the DMA may revert to byte transfers to complete the DMA. This alignment mechanism is shown in Figure 13.5. Alignment overhead occurs at the beginning and end of the DMA operation and, depending on DMA byte count, may be negligible. For Short-Short, Short-to-Word and Word-to-Short multi-cycle transfers, the DMA performs byte request when a memory block is unaligned.

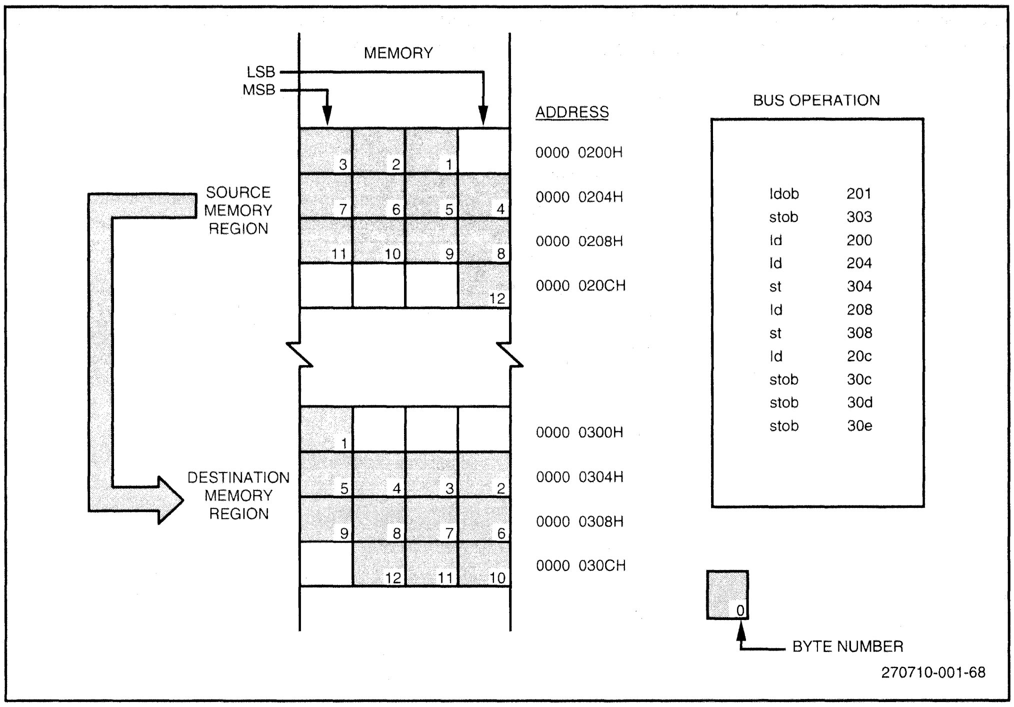


Figure 13.5. Optimization of an Unaligned DMA

DATA CHAINING

Data chaining can generate complex DMAs by linking together multiple transfer operations and is accomplished by using memory-based chaining descriptors to describe component parts of a more complex DMA operation.

The component parts of the chained DMA are referred to as *chaining buffers*. To describe a single DMA chaining buffer, a chaining descriptor (Figure 13.6) supplies source address (SA), destination address (DA) and byte count (BC). Chaining buffers are linked together with the value of the next pointer (NPTR) field in the chaining descriptor. NPTR contains the chaining descriptor address which describes the next part of the chained DMA operation. DMA operation ends when an NPTR of 0 (null pointer) is encountered.

A chained DMA operation is started by specifying a pointer to the first chaining descriptor when **sdma** is used to configure the DMA channel. Initial source address, destination address and byte count are taken from the first chaining descriptor. Chained DMAs are configured such that subsequent buffer transfers use either source, destination or both of these addresses to continue the chained DMA. These modes are referred to as source chaining, destination chaining or source/destination chaining. For example, if a channel is configured for source chaining (Figure 13.7), the source address for the DMA operation is updated to the value specified in each new descriptor. The destination address is continually incremented from the address specified in the DA field of the first descriptor or is held fixed at that address. (Recall that addresses may be incremented or held fixed for any DMA operation.)

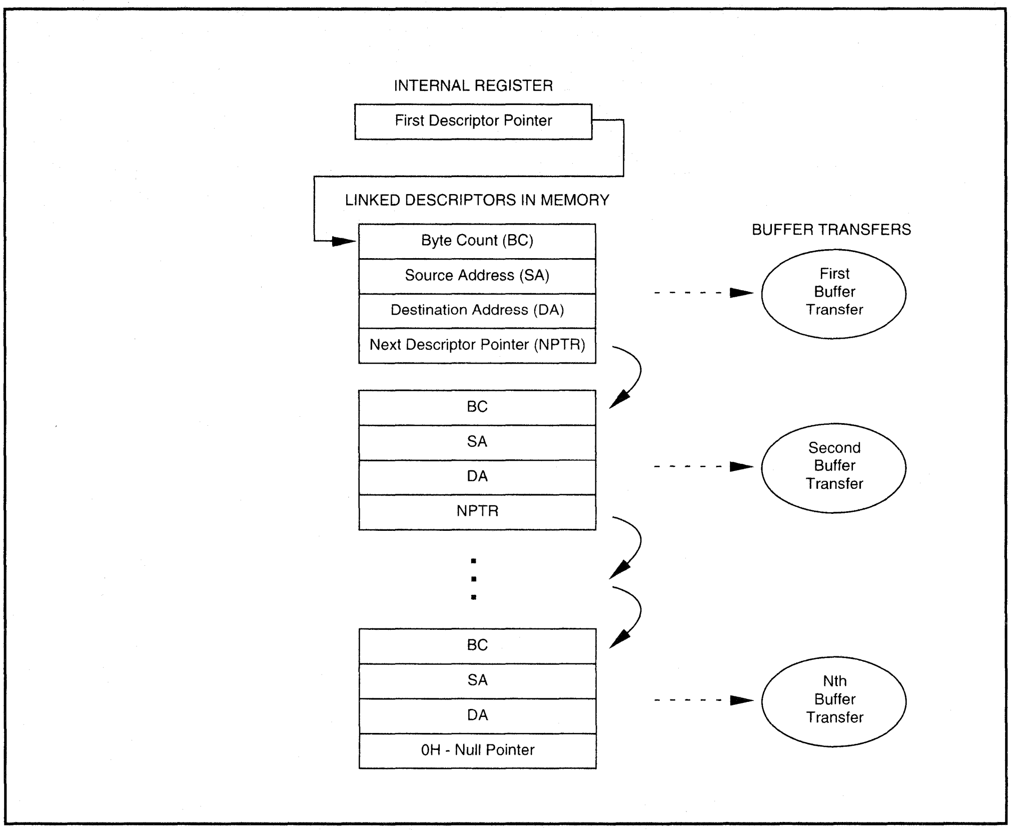


Figure 13.6. DMA Chaining Operation

Each buffer transfer is handled by the DMA controller as if it were a single non-chained DMA. Data alignment requirements for each buffer are identical to the requirements for any other DMA. (See *Data Alignment* in this chapter.) Since each buffer is considered a single DMA, data is never internally buffered when moving from one buffer to another for unaligned DMAs.

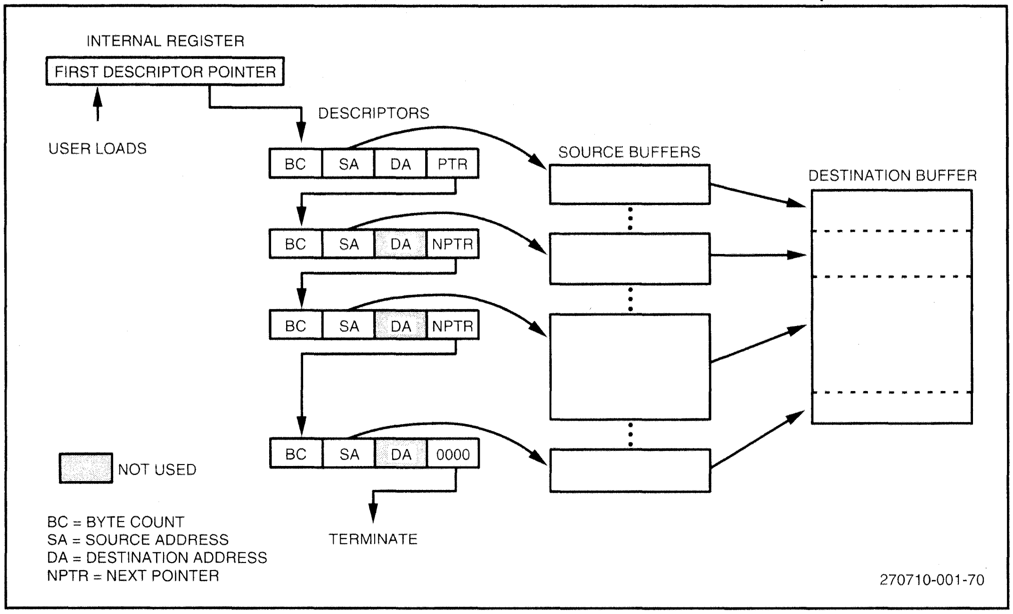


Figure 13.7. Source Chaining

Depending on DMA channel configuration and the chaining mode selected, certain fields in the chaining descriptor are ignored, but must be set to zero for future compatibility:

1. When a channel is source chained, the DA field of the first descriptor specifies the destination address; the DA field in subsequent descriptors is ignored.
2. When a channel is destination chained, the SA field of the first descriptor specifies the source address; the SA field in subsequent descriptors is ignored.
3. When a channel is configured for chained fly-by mode, the SA field always contains the fly-by address; the DA field is ignored.

When descriptors are read from external memory, bus latency and memory speed affect *chaining latency*. Chaining latency is defined as the time required for the DMA controller to access the next descriptor, plus the time required to set up for the next buffer transfer. Chaining latency is reduced by placing descriptors in internal data RAM or fast memory.

DMA-SOURCED INTERRUPTS

Each DMA channel is the source for one interrupt. When a DMA channel signals an interrupt, the DMA interrupt-pending bit corresponding to that channel is set in the interrupt-pending (IPND) register. Each channel's interrupt can be selectively masked in the interrupt mask (IMSK) register or handled as a dedicated hardware-requested interrupt. (Refer to *Chapter 6, Interrupts* for a complete description of hardware-requested interrupts.)

The interrupt-pending bit for a DMA channel is set for the following conditions:

1. A non-chained DMA terminates because byte count reaches 0 or a chained DMA terminates because the null chaining pointer is reached.
2. $\overline{\text{EOP3:0}}$ pin is programmed as an input and asserted to end a DMA or to terminate a source and destination-chained buffer transfer.
3. For a chained DMA, the interrupt-on-buffer-complete function is enabled and the end of a chaining buffer is reached.

SYNCHRONIZING A PROGRAM TO CHAINED BUFFER TRANSFERS

When any of the conditions listed above occur, the current DMA request is completed before the pending bit in the IPND register is set.

Two mechanisms, illustrated in Figure 13.8, enable a program to synchronize with a completed chained buffer transfer. With either mechanism, an interrupt is generated when the chained buffer is complete. The distinction between the mechanisms are:

1. DMA operation continues with no delay on the next chaining buffer. The interrupt service routine may process the data transferred for the completed buffer.
2. DMA waits until the user program processes the first chaining buffer and sets up the next buffer transfer by modifying the chaining descriptors. DMA continues with the next buffer transfer when a bit in the DMA control register (DMAC) is cleared.

These options are selected when the DMA channel is set up with the **sdma** instruction.

TERMINATING OR SUSPENDING A DMA

A DMA operation normally ends when one of the following events is encountered:

- DMA byte count reaches 0 for a non-chained DMA mode.
- $\overline{\text{EOP3:0}}$ pin programmed as an input becomes active for a channel that is non-chained, source-only chained or destination-only chained.
- $\overline{\text{EOP3:0}}$ pin programmed as an input becomes active during the last buffer transfer for a channel which is source/destination chained.
- The null chaining pointer is encountered in any chaining mode.

The DMA takes the following actions when any one of these events occur:

- DMAC register channel done flag is set.
- DMAC register channel terminal count flag is set, only if the byte count has reached 0 (non-chained) or the null chaining pointer is reached (chaining).
- DMAC register channel active bit is reset after all channel activity has completed.
- IPND register channel interrupt pending bit is set. If the corresponding bit in the IMSK is cleared, an interrupt is signaled.

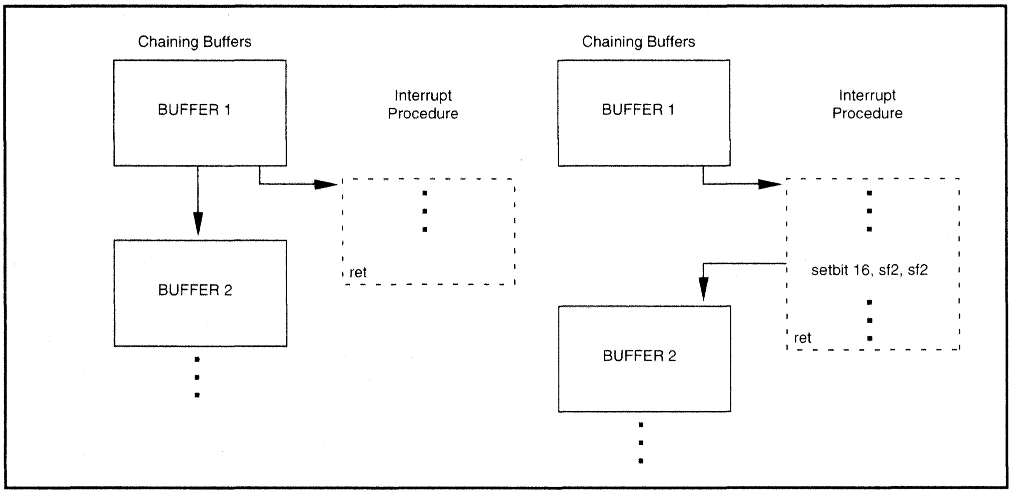


Figure 13.8. Synchronizing to Chained Buffer Transfers

When a chained DMA channel is set up for source/destination chaining, the $\overline{\text{EOP3:0}}$ inputs are designed to terminate only the current chaining buffer. The DMA controller continues normally with the next buffer transfer. The DMA ends as described above if the $\overline{\text{EOP3:0}}$ pin is asserted during the last buffer transfer.

When $\overline{\text{EOP3:0}}$ is asserted, the entire DMA bus request completes before the DMA terminates. For example, assume the DMA is programmed for quad-word transfers. If $\overline{\text{EOP3:0}}$ is asserted, the entire quad-word is transferred before the DMA terminates.

The DMA controller may be configured to generate an interrupt when a DMA terminates. A program may determine how a DMA has ended by reading the DMAC register channel terminal count and channel done flag values:

- If a channel's terminal count flag and done flag are set, the DMA has ended due to a byte count of 0 (non-chaining) or a null chaining pointer reaching 0 (chaining).
- If only the done flag is set for the channel, the DMA has ended because of an active $\overline{\text{EOP3:0}}$ input.

For source/destination chained DMAs, an interrupt is generated by asserting $\overline{\text{EOP3:0}}$ to terminate the current chaining buffer.

NOTE

For source/destination chained DMAs, an interrupt is generated when $\overline{\text{EOP3:0}}$ is asserted or when a buffer transfer is complete and the interrupt-on-buffer complete mode is enabled. There is no way in software to distinguish between these two interrupt sources. If this distinction is necessary, the $\overline{\text{EOP3:0}}$ pin may be connected to a dedicated external interrupt source.

A DMA operation can be suspended at any time by clearing the DMAC register channel-enable bit. It may be necessary to synchronize software to the completion of a channel's bus activity after the enable bit is cleared. This is accomplished by polling the DMA channel active bit as shown in the following assembly code segment.

```
      clrbit 0,sf2,sf2      # disable channel 0
self: bbs    4,sf2,self     # wait for channel
                            # activity to complete
```

DMA operation is restarted by setting the channel enable bit. A channel may be suspended to allow a section of time-critical user code to execute with the maximum core and bus resources available.

To reduce interrupt latency, all DMAs can be suspended when an interrupt is serviced. This option is set in the Interrupt Control (ICON) register. When the option is selected, all DMA operations are suspended during the time that the core processes the interrupt context switch. DMAs are restarted before the interrupt procedure's first instruction is encountered. This option reduces interrupt latency by providing full processor resources to the interrupt context switch.

DMA operations can be suspended by user code in an interrupt procedure to increase procedure throughput. This is accomplished by clearing the DMAC register channel enable field. (See *DMA Command Register* in this chapter.) The interrupt procedure should re-enable all suspended channels before returning.

Issuing **sdma** for an active channel causes the current DMA transfer to abort. Current DMA operation is terminated and the channel is set up with the newly-issued **sdma** instruction. Do not terminate a DMA operation with **sdma**; this instruction causes a "non-graceful" termination of a DMA transfer. In other words, the transfer may be aborted between a source and destination access, potentially losing part of the source data. Additionally, status information for the terminated DMA is lost when the new **sdma** instruction reconfigures the channel. The channel done bit is not set when a DMA is terminated with **sdma**.

CHANNEL PRIORITY

Each DMA channel is assigned a priority. When more than one DMA channel is enabled, channel priority determines the order in which transfers execute for each channel. Channel priority can be programmed in one of two modes: fixed priority or rotating priority mode. The mode is selected with the priority mode bit in DMAC register.

When fixed mode is selected, each channel has a set priority. Channel 0 has the highest priority, followed by Channel 1, 2 and 3; Channel 3 has the lowest priority. In this mode, low-priority DMAs assigned to Channels 1-3 can be locked out while a time-critical DMA assigned to channel 0 receives all of the DMA controller's attention.

When rotating priority is selected, a channel's priority depends on the last channel serviced (Table 13.3). After a channel is serviced, the priority of that channel is automatically changed to the lowest channel priority. The priority of the remaining enabled channels is increased with a new channel becoming the highest priority. Rotating mode ensures that no single channel is locked out for an extended period of time.

Table 13.3. Rotating Channel Priority

Last Channel Serviced	Priority *Lowest*			*Highest*
0	0	3	2	1
1	1	0	3	2
2	2	1	0	3
3	3	2	1	0

Rotating priority is useful for producing a uniform latency for every DMA channel. When rotating mode is selected, the maximum latency for a single channel is the total of all latencies associated with all enabled channels. When fixed mode is enabled, latency for any channel is dependent on the activity of all channels of higher priority.

CHANNEL SETUP, STATUS AND CONTROL

The DMA controller uses the DMA command register (DMAC) and setup DMA instruction (**sdma**) to configure and control the four DMA channels. The update DMA instruction (**udma**) monitors the status of an in-progress DMA operation.

The DMAC register is a special function register (sf2). This register enables or disables each channel and holds frequently-accessed status and control bits for the DMA controller, including idle or active status and termination status for a channel.

sdma configures each channel. **sdma** specifies source address, destination address, byte count, transfer type, chained or non-chained operation.

When a channel is set up using **sdma**, an eight-word (32-byte) block of internal data RAM is allocated for the channel. Channel state is stored in this section of data RAM when operation is preempted by another DMA channel. The user can access the current status for any active or idle DMA operation by examining data RAM assigned to a channel. This status includes the current source and destination addresses and the remaining byte count. **udma** copies the state of an active DMA channel to internal RAM.

The following action is usually taken to set up and start a DMA operation on the i960 CA processor:

1. A channel is set up using the **sdma** instruction.
2. DMAC register is modified to enable the DMA.
3. DMAC register is then read to monitor the activity of the DMA operation.
4. **udma** can be issued and DMA data RAM examined for the current DMA status.

DMA Command Register (DMAC)

The DMA command register (Figure 13.9) is a 32-bit Special Function Register (SFR) specified as sf2 in assembly language. Bits 21-0 are used for DMA status and configuration; the remaining bits (bits 31-22) are reserved. These reserved bits should be programmed to 0 at initialization and not modified thereafter. These reserved bits are not implemented on the i960 CA processor; clearing these bits at initialization is only required for portability to other i960 family products. DMAC function is described below.

channel enable bits (bits 3-0) enable (1) or suspend (0) a DMA after a channel is set up. Bits 0 through 3 enable or disable channels 0 through 3, respectively. If an enable bit for a channel is cleared when a channel is active, the DMA is suspended after pending DMA requests for the channel are completed and all bus activity for the pending request is complete. The channel active bits indicate the channel is suspended. DMA operation resumes at the point it was suspended when the channel enable bit is set. To ensure that a DMA channel does not start immediately after it is set up, the enable bit for the channel must be cleared by software before **sdma** is issued. This is necessary because the DMA controller does not explicitly clear the enable bit after a DMA has completed.

channel terminal count flags (bits 7-4) are set when a DMA has stopped because 1) byte count has reached zero for a non-chained DMA or 2) a null pointer in a chaining descriptor is encountered in data chaining mode. Flags 4 through 7 indicate terminal count for channels 0, through 3, respectively. A terminal count flag is set only after the last request for the channel is serviced and all bus activity for that request is complete. A channel's terminal count flag must be cleared by software before the DMA channel is enabled. This is necessary because the DMA controller does not explicitly clear the terminal count flags after a DMA has completed – this action must be performed by software. The terminal count flags indicate status only. Modifying these bits by software has no effect on a DMA operation.

channel active flags (bits 11-8) indicate that a channel is either idle (0) or active (1). Bits 8 through 11 indicate active channels 0 though 3, respectively. For demand mode, the active bit is set when the DMA request is recognized by internal hardware and remains set until all bus activity for that request is complete. In block mode, the channel active bit remains set for the duration of the block mode DMA. Channel active flags indicate status only. These flags cannot be modified by software; attempts to modify these bits by software has no effect on a DMA operation.

channel done flags (bits 15-12) indicate that a channel's DMA has finished. Bits 12 through 15 indicate a completed DMA on channels 0 through 3, respectively. The DMA controller sets a channel done flag when a DMA operation has finished in one of three ways:

- byte count reached zero in a non-chaining mode
- null pointer reached in a chaining mode
- $\overline{\text{EOP3:0}}$ signal is asserted which ends the DMA operation

DMA controller channel done flags are not cleared when a channel is set up or enabled. This action must be performed by software. Channel done flags indicate status only; modifying these flags does not affect DMA controller operation.

channel wait bits (bits 19-16) signal that a chaining descriptor was read and, optionally, enables a read of the next chaining descriptor in memory. Channel wait bits only enable the descriptor read when the channel is set up with the channel wait function enabled. (See the section titled *Set Up DMA Instruction* in this chapter.) This function provides synchronization for programs which dynamically change chaining descriptors when a DMA is in progress. The DMA controller sets a channel wait bit when a chaining descriptor is read from memory. If the channel wait function is enabled, the DMA controller waits for the channel wait bit to be cleared by software before the next descriptor is read. (See the section in this chapter titled *Data Chaining*.)

priority mode bit (bit 20) selects fixed (0) or rotating (1) priority mode. The priority mode determines the order in which DMA channels are serviced if more than one request is pending. (See *Channel Priority*.)

throttle bit (bit 21) selects the maximum ratio of DMA process time to user process time. If the throttle bit is set, the DMA process can take up to one clock for every one clock of the user process. If the bit is clear, the DMA process can take up to four clocks for every one user process clock. The effect of the throttle bit on DMA performance is fully described in the DMA Performance section of this chapter.

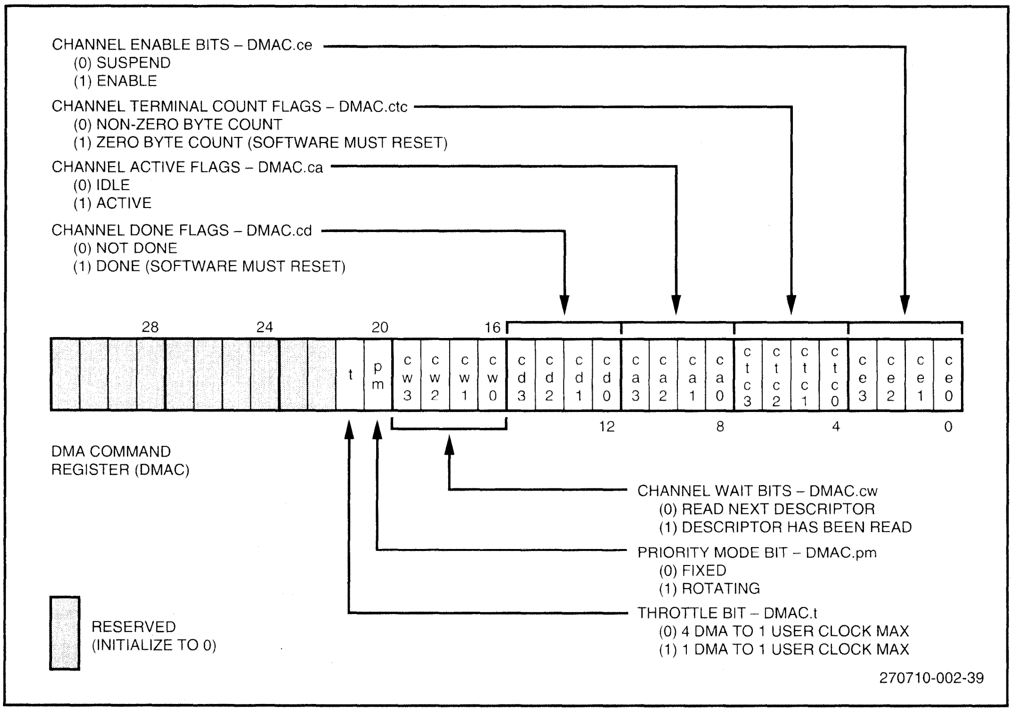

CHANNEL ENABLE BITS – DMAC.ce
(0) SUSPEND
(1) ENABLE
CHANNEL TERMINAL COUNT FLAGS – DMAC.ctc
(0) NON-ZERO BYTE COUNT
(1) ZERO BYTE COUNT (SOFTWARE MUST RESET)
CHANNEL ACTIVE FLAGS – DMAC.ca
(0) IDLE
(1) ACTIVE
CHANNEL DONE FLAGS – DMAC.cd
(0) NOT DONE
(1) DONE (SOFTWARE MUST RESET)
28 24 20 16 12 8 4 0
t pm cw3 cw2 cw1 cw0 cd3 cd2 cd1 cd0 ca3 ca2 ca1 ca0 ctc3 ctc2 ctc1 ctc0 ce3 ce2 ce1 ce0
DMA COMMAND REGISTER (DMAC)
CHANNEL WAIT BITS – DMAC.cw
(0) READ NEXT DESCRIPTOR
(1) DESCRIPTOR HAS BEEN READ
PRIORITY MODE BIT – DMAC.pm
(0) FIXED
(1) ROTATING
THROTTLE BIT – DMAC.t
(0) 4 DMA TO 1 USER CLOCK MAX
(1) 1 DMA TO 1 USER CLOCK MAX
RESERVED (INITIALIZE TO 0)
270710-002-39

Figure 13.9. DMA Command Register (DMAC)

Set Up DMA Instruction (sdma)

sdma configures a DMA channel. The **sdma** instruction has the following format:

```
sdma  op1,           op2,           op3
      reg/lit/sfr    reg/lit/sfr    reg
```

The three operands are described in Figure 13.10 and in the following text:

op1: This operand is the number of the channel (0-3) which is set up with **sdma**. Values other than the valid channel numbers are reserved and can cause unpredictable results if used.

op2: This operand is the DMA control word for the channel. The control word selects the modes and options for a DMA. (The value of this operand is described in the next section, *DMA Control Word.*)

op3: This operand is used differently depending on the DMA configuration:

- *Non-chaining multi-cycle DMAs*: op3 is the first of three consecutive 32-bit registers. The first register must be programmed with byte count; the second, the source address; the third, the destination address.
- *Non-chained fly-by DMAs*: op3 is the first of two consecutive 32-bit registers. The first register must be programmed with byte count; the second, the fly-by address.
- *All chained DMAs*: op3 is a single 32-bit register. op3 must be programmed with a pointer to the first chaining descriptor. See the section in this chapter titled *Data Chaining* for more information on chaining descriptors.

NOTE

The op3 operand must be a quad-aligned register (r4, r8, r12, g0, g4, g8 or g12).

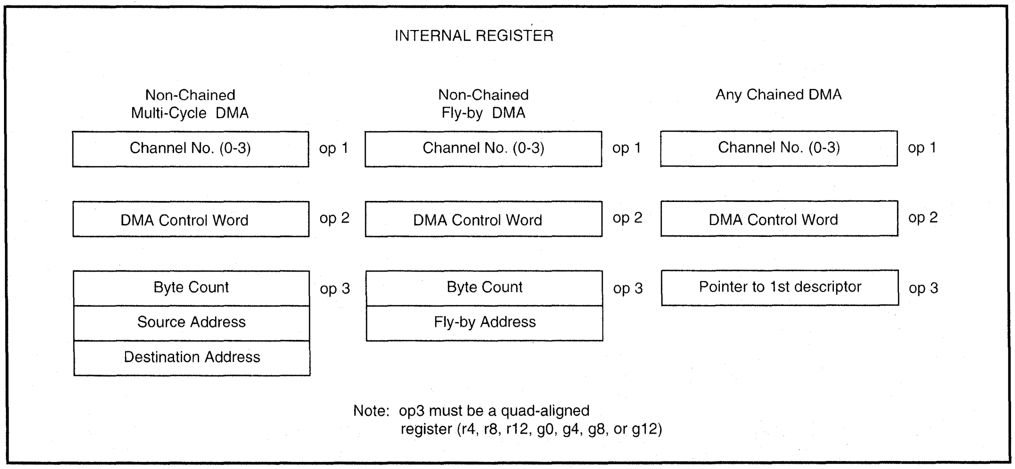


Figure 13.10. Setup DMA (sdma) Instruction Operands

The channel setup mechanism, started with the **sdma** instruction, is two-part. **sdma** is a multi-cycle instruction. When **sdma** is issued:

1. the instruction executes — reading the register operands for the DMA operation — then completes, freeing these registers for use by other instructions.
2. a DMA setup process is triggered to complete the channel setup. The setup process runs concurrently with the execution of the user's program.

After the setup process is started, it is possible to enable a channel through the DMAC register before the setup completes. In this case, the DMA controller waits for the setup to complete before the DMA operation begins. The result is the potential for additional latency on the first DMA request. To decrease this additional latency, issue the **sdma** instruction well in advance of enabling the DMA channel.

A second **sdma** instruction can be issued before a previously-issued DMA setup event completes. The second **sdma** must wait for the first event to complete, preventing other instructions from executing. If the segment of code which issues the **sdma** instructions is time-critical, it may be beneficial to overlap other operations — other than **sdma** — with the setup event and space the **sdma** instructions in the code instead of issuing them back-to-back. A waiting **sdma** instruction is interruptible; therefore, back-to-back **sdma** instructions do not adversely increase interrupt latency.

DMA Control Word

DMA control word (Figure 13.11) specifies DMA modes and options. The control word is an operand (op2) of the **sdma** instruction.

transfer type field (bits 3-0) specifies the request length of bus requests issued by the DMA controller and selects between multi-cycle and fly-by transfers.

source/destination addressing bits (bits 4 and 5) determine if the source or destination address for a channel is held fixed (1) or incremented (0) during a DMA. Bit 5 controls the source address and bit 4 controls the destination address. The source addressing bit (bit 5) controls address increment and hold for fly-by transfers.

synchronization mode bit (bit 6) specifies that a multi-cycle demand mode transfer is synchronized with the source (0) or the destination (1). In fly-by mode, the bit specifies whether fly-by stores (0) or fly-by loads (1) are performed. Fly-by stores are source synchronized; fly-by loads are destination synchronized. In block mode, this bit is ignored.

synchronization select bit (bit 7) determines whether a transfer is demand mode (1) or block mode (0).

EOP/TC select bit (bit 8) selects $\overline{\text{EOP/TC3:0}}$ pin function. If bit is set, the pins are configured as end-of-process inputs ($\overline{\text{EOP3:0}}$). If the $\overline{\text{EOP/TC3:0}}$ select bit is cleared, the pin is configured as a terminal count output ($\overline{\text{TC3:0}}$).

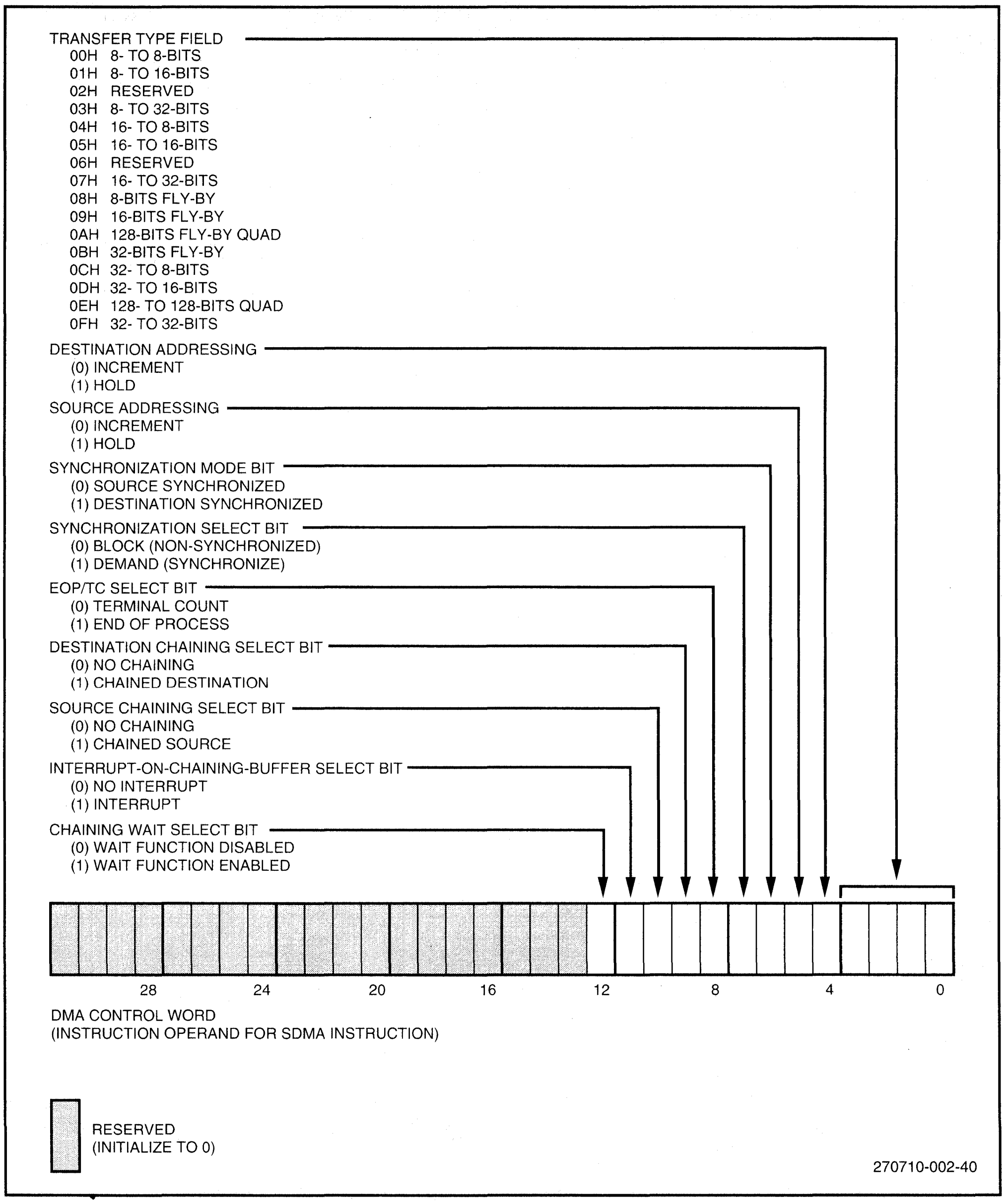


Figure 13.11. DMA Control Word

The following bits in the DMA control word control data chaining. If chaining mode is not used, the source/destination chaining select bits (bits 9 and 10) must be set to 0.

source/destination chaining select bits (bits 9 and 10) are set to enable data chaining mode. Setting bit 9 enables destination chaining; setting bit 10 enables source chaining. Setting bits 9 and 10 enables source/destination chaining. Non-chaining mode is selected if both bits are clear. (See *Data Chaining* in this chapter.)

interrupt-on-chaining-buffer select bit (bit 11) is set to cause an interrupt to be generated when byte count for a chained buffer reaches 0. Bit is ignored in a non-chaining mode.

chaining-wait select bit (bit 12) is set to enable the channel-wait function. When the wait enable function is selected, DMAC register channel-wait bits must be cleared before a chaining descriptor is read. This channel-wait function, together with the interrupt-on, buffer-complete function, allows chaining descriptors to be dynamically changed during the course of a chained DMA operation. This bit is ignored when a non-chaining mode is selected. (See *Data Chaining* in this chapter.)

DMA Data RAM

The DMA controller uses up to 32 words of internal data RAM to swap service between active channels. When a channel is set up, the DMA controller dedicates 8 words of data RAM to that channel (Figure 13.12). When channel service swaps from one to another, the state of the active channel is saved in data RAM. The state is retrieved when the channel is again serviced. DMA data RAM for a channel is only updated when service swaps to another channel or **udma** is executed.

NOTE

Channel swapping occurs when channel priority for a pending DMA request is higher than that of the currently active or last-serviced channel. (See *Channel Priority* in this chapter.)

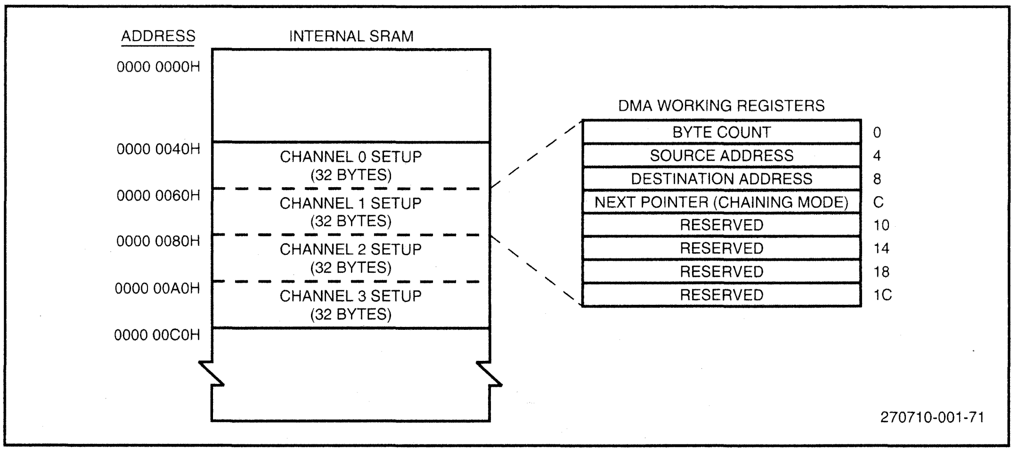


Figure 13.12. DMA Data RAM

udma flushes the state of a currently executing channel to data RAM. Additional DMA transfers can occur between the time that **udma** executes and a program reads the locations in data RAM. The channel may be suspended before **udma** executes to ensure coherence between the values read from data RAM and actual DMA progress.

DMA data RAM is 128 bytes of internal RAM located at 0000 0040H to 0000 00BFH (Figure 13.12). This memory is read/write in supervisor mode and read only in user mode. This supervisor protection prevents errant modification of the DMA RAM by a program.

DMA data RAM for any channel can be used for general purpose storage when the channel is not in use. A program, however, must not modify data RAM dedicated for a channel which is already set up and awaiting activity. In general, any modification of DMA Data RAM for an active or idle channel may cause unpredictable DMA controller operation. Conversely, executing **sdma** may cause previously stored data to be overwritten in the data RAM.

Channel Setup Examples

Example 13.1. Simple Block Mode Setup

```
## Block mode setup . . .
 mov        0xc,g4              # Byte count = 12
 ldconst    c0_src_addr,g5      # Source address for channel 0
 ldconst    c0_dest_addr,g6     # Destination addr for channel 0
 ldconst    0xf,g3              # DMA ctl word (32/32 std-source
                                # inc. - dest. inc. - block)
 sdma       0,g3,g4             # Setup channel 0
  .
  .                             # Other instructions (optional)
  .
 setbit     0,sf2,sf2           # enable channel 0
```

Example 13.2. Chaining Mode Setup

```
## Chaining mode setup . . .
ldconst  ptr1,g4      # Initial descriptor pointer
ldconst  0x1a6f,g3    # DMA ctl word (32/32 std-source)
                      # hold-dest inc. -demand source sync.-
                      # dest. chain,channel wait,interrupt on
                      # buffer complete)
sdma     1,g3,g4      # Setup channel 1
 .
 .                    # Other instructions (optional)
 .
setbit   1,sf2,sf2    # enable channel 1
## Descriptor list in memory for chaining . . .
ptr1:
         .word 0x100, b0_src_addr, b1_dest_addr, ptr3
ptr2:
         .word 0x200, 0x0, b0_dest_addr, 0x0
ptr3:
         .word 0x100, 0x0, b2_dest_addr, ptr2
```

DMA EXTERNAL INTERFACE

DMA signal characteristics (DACK3:0, DREQ3:0, EOP/TC3:0 and DMA) and DMA transfer timing requirements are described in the following sections. Refer to the *i960 CA Microprocessor Data Sheet* for AC specifications.

Pin Description

DREQ3:0	DMA Request (input) - DMA request pins are individual, asynchronous channel-request inputs used by peripheral circuits to obtain DMA service. In fixed priority mode, DREQ0 has the highest priority; DREQ3 has the lowest priority. A request is generated by asserting the DREQ3:0 pin for a channel.
DACK3:0	DMA Acknowledge (output) - notifies an external DMA device that a transfer is taking place. The pin is active during the bus request issued to the DMA device.
EOP/TC3:0	End of Process (input EOP3:0) or Terminal Count (output TC3:0) - Configured as an output, the pin is driven active (low) during the last transfer for a DMA and has the same timing as the DACK3:0 signals. TC3:0 pins are asserted when byte count reaches zero for a chained or non-chained DMA. Programmed as an input, an asynchronous active (low) signal on the pin for a minimum of two clock cycles causes DMA to terminate as described in the section titled *Terminating or Suspending a DMA*.
DMA	DMA Bus Request (output) - This pin indicates that a bus request is issued by the DMA controller. The pin is active during a bus request originating from the DMA controller and inactive during all other bus requests. DMA pin value is indeterminate during idle bus cycles. The DMA pin is not active when chaining descriptors are loaded from memory.

Demand Mode Request/Acknowledge Timing

Demand-mode transfers require that the DMA request (DREQ3:0) signal is asserted before the transfer is started. Demand mode transfers should satisfy two requirements:

1. After the transfer is requested, the DMA controller must be fast in responding to the requesting device. This characteristic is referred to as latency.
2. The requesting device must be given enough time to deassert the request signal to prevent an unwanted DMA transfer.

The timing for demand mode transfers is described in the following sections. Latency characteristics of a DMA transfer are described in this chapter's *DMA Performance* section.

An external device initiates a demand mode transfer by asserting (active low) one of the DMA request pins. The acknowledge pin is driven active by the DMA controller during the bus request issued to access the DMA requestor. Figure 13.14 shows DACK3:0 output timings.

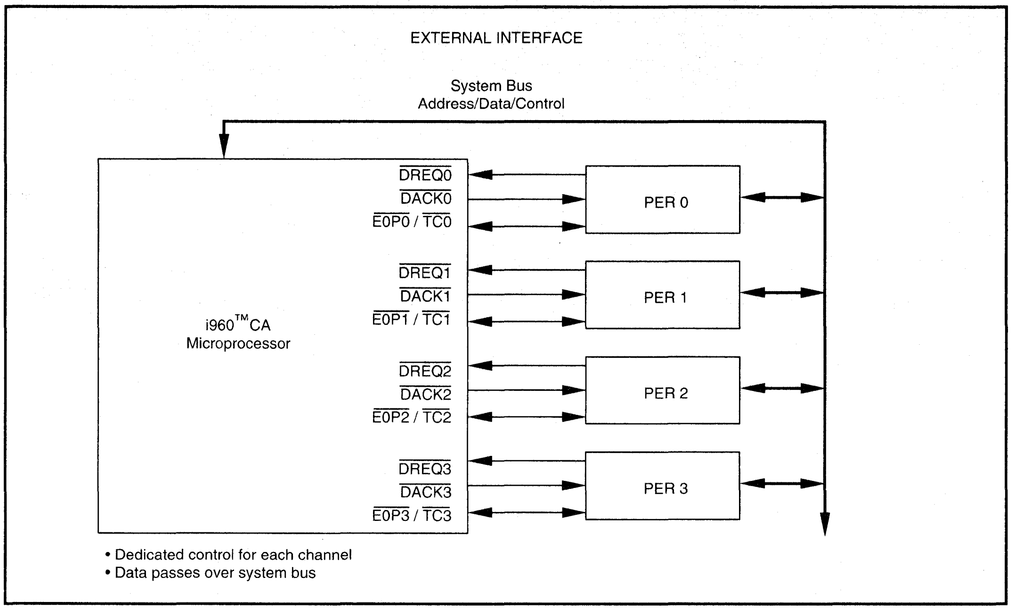


Figure 13.13. DMA External Interface

To start a demand mode DMA, $\overline{\text{DREQ3:0}}$ must be held asserted until the acknowledge bus request is started. $\overline{\text{EOP3:0}}$ pins do not require external synchronization; however, to guarantee detection on a particular PCLK2:1 cycle, setup and hold requirements must be satisfied.

At the end of the acknowledge bus request, $\overline{\text{DREQ3:0}}$ may be held active to initiate further DMA transfers or $\overline{\text{DREQ3:0}}$ may be driven inactive to prevent further transfers. Depending on DMA mode, arbitration for the next DMA transfer begins:

Case 1: On the PCLK2:1 cycle in which $\overline{\text{DACK3:0}}$ is deasserted - This timing applies to demand mode fly-by transfers — and multi-cycle packing or unpacking modes — with adjacent request loads or adjacent request stores.

Case 2: Two PCLK2:1 cycles after $\overline{\text{DACK3:0}}$ is deasserted - This timing applies to demand mode multi-cycle transfers with alternating request loads and stores.

NOTE

When a DMA operation is destination-synchronized, the next load access is performed even if the request input is deasserted. This "prefetch" is implemented to increase performance. If the following DMA cycle is prevented, prefetch data is saved internally and stored when the next transfer is requested. The entire DMA cycle is not repeated.

End Of Process/Terminal Count Timing

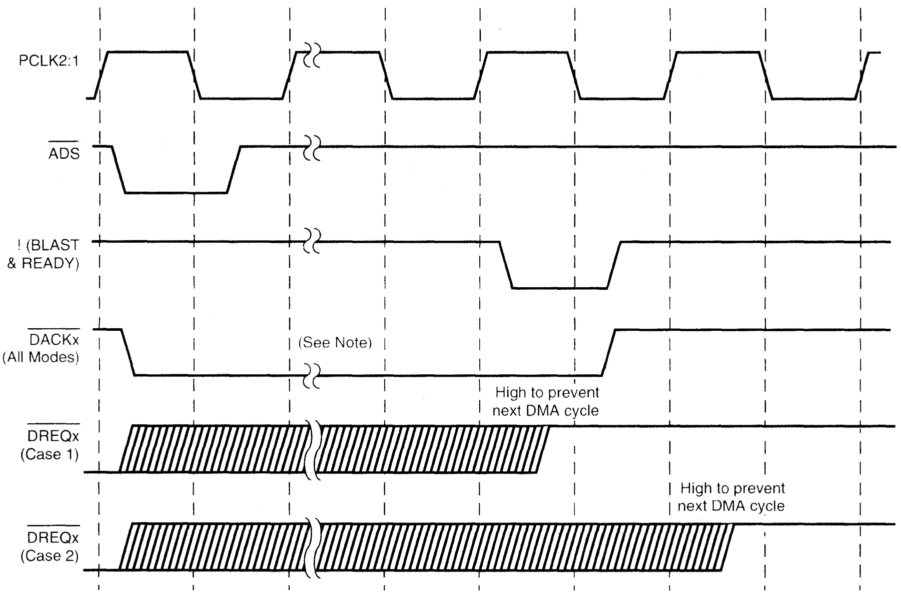
EOP/TC3:0 can be programmed as an input (EOP3:0) or output (TC3:0) for each channel. EOP/TC3:0 pins are configured when a channel is setup using **sdma**.

TC3:0 is asserted when byte count reaches 0 for a chained or non-chained DMA. A TC3:0 pin for a channel is driven active during the last acknowledge bus request. TC3:0 pins have the same timing as DACK3:0.

EOP3:0 pins are asserted to terminate a DMA. EOP3:0 pins are active-level detected. For proper internal detection, EOP3:0 pins must be asserted for a minimum of two and maximum of 17 PCLK2:1 cycles (Figure 13.15). EOP3:0 pins do not require external synchronization; however, to guarantee detection on a particular PCLK2:1 cycle, setup and hold requirements must be satisfied. Setup and hold times are specified in the *i960 CA Microprocessor Data Sheet.* EOP3:0 inputs adhere to the same timing requirements as DREQ3:0 for arbitration of the next DMA transfer.

NOTE

The maximum pulse width requirement for the EOP3:0 pin is to prevent more than one buffer transfer to terminate in the source/destination chaining mode.



NOTE: DACKx is asserted for the duration of a DMA bus request. The request may consist of multiple bus accesses (defined by ADS and BLAST).

Figure 13.14. DMA Request and Acknowledge Timing

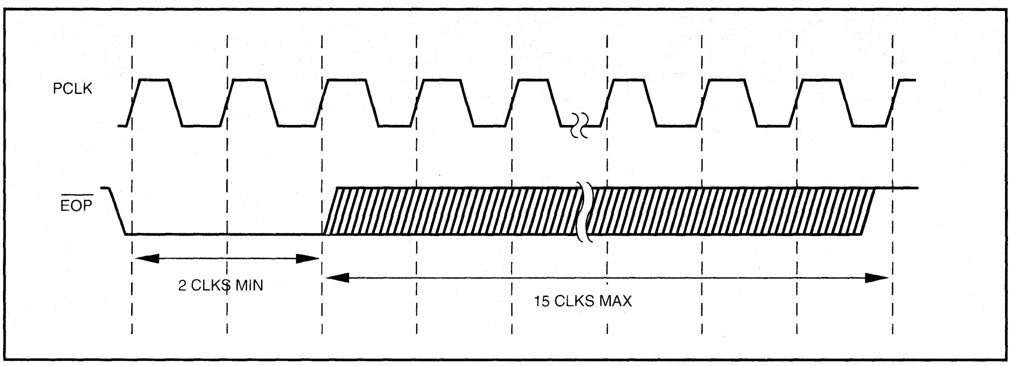


Figure 13.15. $\overline{EOP3:0}$ Timing

Block Mode Transfers

Block mode DMAs require no synchronization with a source or a destination device. $\overline{DREQ3:0}$ inputs are ignored during block mode DMAs. The acknowledge signal ($\overline{DACK3:0}$) is driven active when the source is accessed. $\overline{EOP/TC3:0}$ pins have the same function as described above in the section *End of Process/Terminal Count Timing*.

DMA Bus Request Pin

The DMA request pin ($\overline{DMA}$) indicates that the DMA controller initiated a bus access. The pin is asserted (low) for any DMA load or store bus request. $\overline{DMA}$ is deasserted (high) for other bus requests. The $\overline{DMA}$ pin has the same timing as the $W/\overline{R}$ pin. (See *Chapter 11, External Bus Description* for a complete timing description of the $\overline{DMA}$ pin.)

The $\overline{DMA}$ pin is not active when chaining descriptors are fetched from memory.

DMA Controller Implementation

The i960 CA processor's DMA functions are implemented primarily in microcode. Processor clock cycles are required to setup and execute a DMA operation. DMA features — including data chaining, data alignment, byte assembly and disassembly — are implemented in microcode. DMA hardware arbitrates channel requests, handles the DMA external hardware interface and interfaces to microcode for most efficient use of the core resources.

When considering whether to use the DMA controller, two questions generally arise:

1. When a DMA transfer is executing, how many internal processor clock cycles does the DMA operation consume?
2. When a DMA transfer is executing, how much of the total bus bandwidth is consumed by the DMA bus operations?

These questions are addressed in the following sections.

DMA and User Program Processes

The i960 CA processor allows DMA operations to be executed in microcode while providing core bandwidth for the user's program. This sharing of core resources is accomplished by implementing separate hardware processes for each DMA channel and for the user's program. Alternating between the DMA and the user process enables the user code and up to four DMA processes (one per channel) to run concurrently.

The environments for the DMA and user processes are implemented entirely in internal hardware, as well as the mechanism for switching between processes. This hardware implementation enables the i960 CA processor to switch processes on clock boundaries — no instruction overhead is necessary to switch the process. With this switching mechanism, DMA microcode and the user program can frequently alternate execution with absolutely no performance loss caused by the process switching.

A process switch from user process to DMA process occurs as a result of a *DMA event*. A DMA event is signaled when a DMA channel requires service or is in the process of setting up a channel. Signaling the DMA event is controlled by DMA logic.

After a DMA event is signaled, the DMA process takes a certain number of clock cycles and then the user process is restored. The maximum ratio of DMA-to-user cycles is 4:1. This means that, at most, the DMA process takes four clock cycles to every single-user process clock. The ratio of DMA to user cycles can also be selected as 1:1 to increase execution speed of the user process while a DMA is in progress. The user-to-DMA cycle ratio is controlled by the throttle bit in the DMA command register (DMAC.t).

A DMA rarely uses the maximum available cycles for the DMA process. Actual cycle allocation between user process and DMA process depends on the type of DMA operation performed, DMA channel activity and external bus loading and performance. Maximum allocation of internal processor clocks to DMA processes are specified in *DMA Performance*.

Bus Controller Unit

The bus controller unit (BCU) accesses memory and devices which are source and destination of a transfer. When the DMA process is active, DMA microcode issues load or store requests to the bus controller to perform DMA data transfers. The DMA and user processes equally share access to the bus on a request-by-request basis. If both processes attempt to flood the bus controller with memory requests, the bus is shared equally; this prevents lockout of either process. If either process does require the bus, the bus controller resource may be used entirely by either process.

The BCU contains a queue which accepts up to three pending requests for bus transactions (Figure 13.16). When a DMA channel is set up, the queue is divided such that one slot is dedicated for DMA process requests and two slots are dedicated for user process requests. DMA and core entries are arranged in such a way that when both a user and DMA slot are filled, bus request servicing alternates between requests issued by the user and DMA processes.

DMA Controller Logic

DMA controller logic manages the execution of DMA operations independently from the core. This logic performs the following functions:

- Synchronizes DMA transfers with external request/acknowledge signals.
- Provides the program interface to set up each of the four DMA channels.
- Provides the program interface to monitor the status of the four channels.
- Arbitrates requests between multiple DMA channels by managing channel priority.
- Produces the DMA event which causes DMA microcode to execute.

DMA Performance

DMA performance is characterized by two values: throughput and latency (Figure 13.17). Throughput measurement is needed as a measure of the DMA transfer bandwidth. Worst-case latency is required to determine if the DMA is fast enough in responding to transfer requests from DMA devices.

Throughput describes how fast data is moved by a DMA operation. In this discussion, throughput is derived as the number of PCLK2:1 cycles per DMA request. This value is denoted as N_{T_DMA}. The established measure of throughput, in units of bytes/second, is derived with the following equation:

$$\text{Throughput (bytes/second)} = \frac{(n_R * f_c)}{N_{T\_DMA}}$$

where:

N_{T_DMA}	= throughput per DMA request (PCLK2:1 cycles)
n_R	= bytes per DMA request
f_c	= PCLK2:1 frequency

Latency is defined as the maximum time delay measured between the assertion of $\overline{\text{DREQ3:0}}$ and the assertion of the corresponding $\overline{\text{DACK3:0}}$ pin. This section deals with worst-case latency. In this section, latency is derived in number of PCLK2:1 cycles. This value is denoted by the symbol N_{L_DMA}. The established measure of DMA latency, in units of seconds, is derived with the following equation:

$$\text{DMA Latency} = \frac{N_{L\_DMA}}{f_c}$$

where:

N_{L_DMA}	= Latency (PCLK2:1 cycles)
f_c	= PCLK2:1 frequency

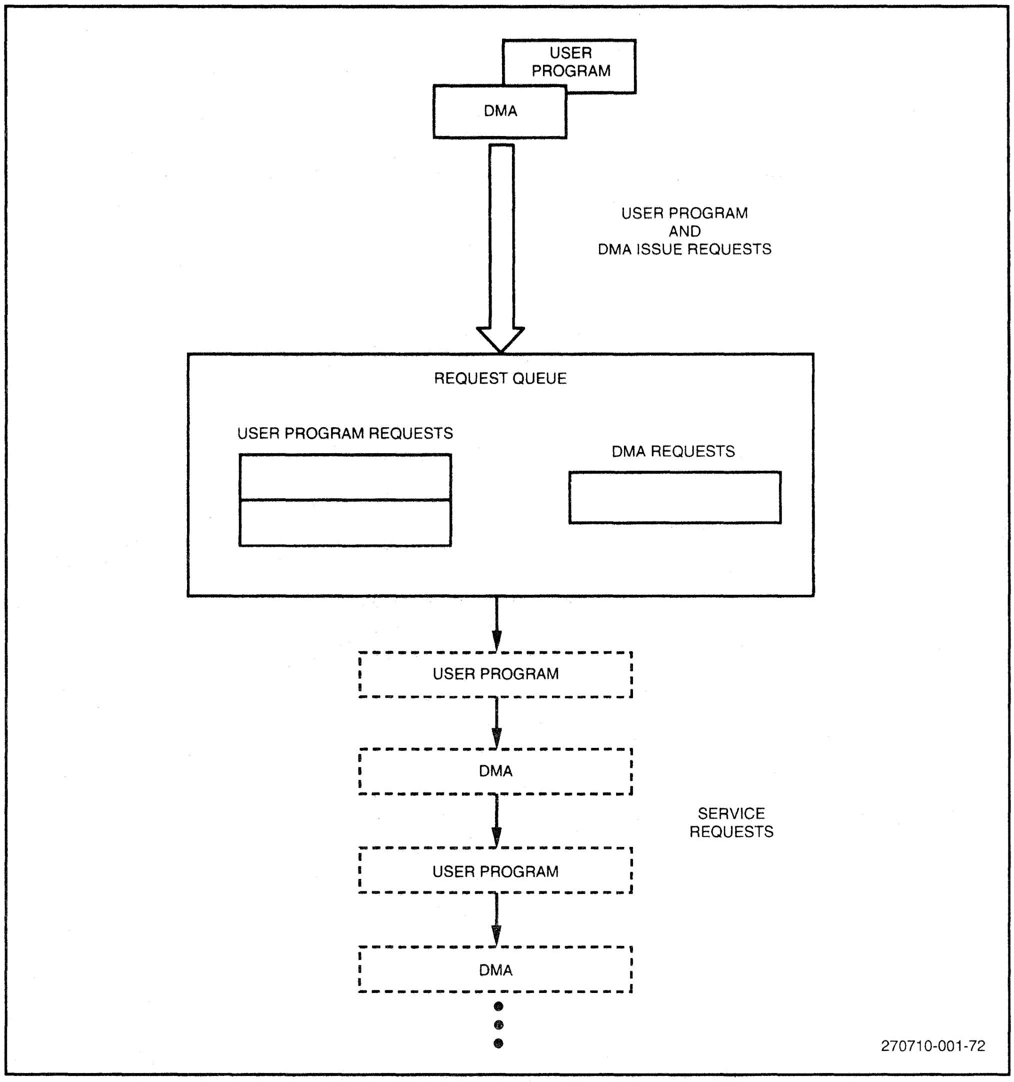
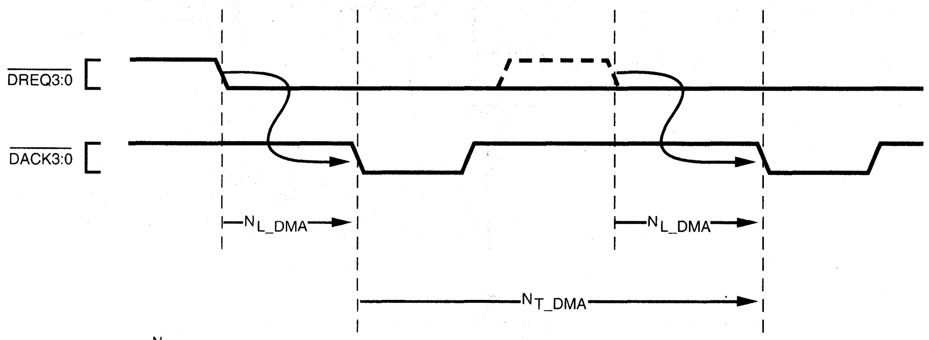


Figure 13.16. DMA and User Requests in the Bus Queue

DMA Throughput

DMA throughput (N_{T_DMA}) for a particular system is governed by the following factors:

- DMA transfer type
- memory system configuration
- Bus activity generated by the user process
- DMA throttle bit value



$$\text{LATENCY} = \frac{N_{L\_DMA}}{fc} \text{ (SEC)}$$

$$\text{THROUGHPUT} = \frac{n_R f_C}{N_{T\_DMA}} \text{ (BYTES/SEC)}$$

N_{L_DMA} = NUMBER OF LATENCY CLOCKS
N_{T_DMA} = NUMBER OF CLOCKS PER DMA REQUEST
n_R = NUMBER OF BYTES PER DMA REQUEST
fc = PCLK2:1 FREQUENCY

270710-001-73

Figure 13.17. DMA Throughput and Latency

N_{T_DMA} is derived from the transfer clocks provided in Table 13.4. Values in this table are derived assuming:

- No bus activity is generated by the user process.
- DMA transfer source and destination memory are zero wait states or internal data RAM.

Table 13.4 provides the number of PCLK2:1 cycles required for each unit DMA transfer. Transfer clock values, denoted by the symbol N_{DMA}, are provided in the two boldface columns. These columns show transfer clocks for the DMA throttle bit set to 1:1 and 4:1 configuration. Transfer clocks are given in pairs separated by a "/": the number on the left is the value for source synchronized demand mode transfers; the number on the right is the value for destination synchronized demand mode transfers.

The number of bytes per transfer is provided in Table 13.4. This is the number of bytes which are transferred in N_{DMA} clock cycles. Bytes per transfer is denoted by the symbol n_{DMA}.

DMA throughput (N_{T_DMA}) is calculated using the following equation:

$$N_{T\_DMA} = N_{DMA} * (\frac{n_R}{n_{DMA}})$$

where:

N_{DMA}	= number of PCLK2:1 cycles per transfer
n_R	= number of bytes transferred per DMA request
n_{DMA}	= number of bytes per DMA transfer

Table 13.4. DMA Transfer Clocks - N_{DMA}

Transfer Type (source-to-destination data length)	Bytes per Transfer (n_{DMA})	Transfer Clocks N_{DMA} in PCLK2:1 cycles (Source Sync./Destination Sync.)				
			Throttle = 4:1		**Throttle = 1:1**	
		DMA Process	User Process	N_{DMA}	User Process	N_{DMA}
8-to-8 Multi-Cycle	1	4/4	6/6	**10/10**	7/7	**11/11**
8-to-16 Multi-Cycle	2	11/11	10/11	**21/22**	18/19	**29/30**
8-to-32 Multi-Cycle	4	23/25	16/15	**39/40**	30/29	**53/54**
16-to-8 Multi-Cycle	2	10/10	8/8	**18/18**	14/13	**24/23**
16-to-16 Multi-Cycle	2	4/4	6/6	**10/10**	7/7	**11/11**
16-to-32 Multi-Cycle	4	9/12	11/8	**20/20**	17/14	**26/26**
32-to-8 Multi-Cycle	4	22/22	13/13	**35/35**	26/23	**48/45**
32-to-16 Multi-Cycle	2	10/11	8/8	**18/19**	14/13	**24/24**
32-to-32 Multi-Cycle (aligned)	4	4/4	6/6	**10/10**	7/7	**11/11**
32-to-32 Multi-Cycle (unaligned)	4	6/6	6/6	**12/12**	9/9	**15/15**
128-to-128 Multi-Cycle	16	6/7	9/9	**15/16**	10/10	**16/17**
8-bit Fly-by	1	3/3	3/3	**6/6**	4/4	**7/7**
16-bit Fly-by	2	3/3	3/3	**6/6**	4/4	**7/7**
32-bit Fly-by	4	3/3	3/3	**6/6**	4/4	**7/7**
128-bit Fly-by	16	3/3	6/6	**9/9**	6/6	**9/9**

The columns in Table 13.4 labeled *DMA Process* and *User Process* show the number of clock cycles allocated to either these processes during a single DMA transfer. The following formula provides the minimum fraction of processor bandwidth remaining for the user process during a DMA transfer:

$$\text{Minimum User Process Bandwidth} = \left(\frac{\text{User Process Clocks}}{N_{DMA}}\right) * 100\%$$

DMA Latency

DMA latency in a system depends on the following factors:

- DMA Transfer type and subsequently the worst-case throughput value calculated for that transfer
- Number of channels enabled and the priority of the requesting channel
- Status of the suspend DMA on interrupt bit in the DMA control register (DMA.dmas)

DMA latency is the sum of the worst-case throughput for the channel plus added components which are dependent on the configuration of the DMA controller. DMA latency is denoted as N_{L_DMA} in the following discussion and is measured in number of PCLK2:1 cycles.

Values for worst-case throughput are provided in Table 13.5. N_{T_DMA}, N_{T_first} and N_{T_chain} describe DMA throughput. N_{T_DMA}, derived in the previous section, describes the average DMA throughput, measured for a transfer which is in progress. N_{T_first} and N_{T_chain} represent boundary conditions of throughput for the following conditions:

First DMA transfer in non-chained modes - N_{T_first} is the throughput of the first transfer of a non-chained DMA operation. After the setup microcode completes, additional microcode is required to start the first DMA transfer.

First DMA transfer of a chained DMA buffer - N_{T_chain} is the throughput between chained buffers (chaining mode only). The time required to arbitrate another buffer transfer in chaining mode, read the next chaining descriptor from memory and acknowledge the first transfer of the new buffer. Two values are given in Table 13.5 for N_{T_chain} to account for differences in throughput for EOP chaining mode. EOP chaining occurs when the DMA controller is configured for both source and destination chaining, the $\overline{\text{EOP}}/\overline{\text{TC3:0}}$ pins are configured as inputs and $\overline{\text{EOP3:0}}$ is asserted by the external system to cause chaining to the next buffer transfer.

N_{T_first} and N_{T_chain} are calculated using the following equation:

$N_{T\_first} = [N_{T0\_first} + N_{T0\_first} *(0.6*\text{throttle})]$

$N_{T\_chain} = [N_{T0\_chain} + N_{T0\_first} *(0.6*\text{throttle})]$

where:

throttle = 0 for 4:1 throttle mode; 1 for 1:1 throttle mode

The factor of 0.6 is used to characterize the effect on the worst-case base throughput value of disabling the throttle mode. For determination of N_{T_DMA}, Table 13.4 provides separate measurements with the throttle bit both enabled and disabled.

Additional components of worst-case DMA latency depend on DMA controller configuration. These components are described below and their values are given in Table 13.6.

Set up the DMA channel (N_{setup}) - Describes the time required for microcode to complete channel setup after **sdma** is executed. This latency component may be ignored if the channel is enabled N_{setup} clock cycles after **sdma** is executed.

Swap the DMA channel (N_{swap}) - Time required for a higher priority channel to preempt a lower priority channel and the time required to copy the associated DMA working registers to internal data RAM. If only one channel is enabled in a system, then N_{swap} equals 0.

Lower Priority Channels (N_{lower}) - Latency of lower priority channels which are preempted when a DMA for the highest priority channel is requested. A transfer on the lower priority channel must complete before the higher priority channel is serviced.

Interrupt Latency (N_{int}) - Latency caused by servicing an interrupt with the suspend DMA mode enabled. N_{int} is the same as the worst case interrupt latency for the system.

Table 13.5. Base Values of Worst-case DMA Throughput used for DMA Latency Calculation

Transfer Type (source-to-dest. data length)	Base worst-case throughput per request (PCLK2:1 cycles) (Source Sync./Destination Sync.)		
	N_{T0_first}	N_{T0_chain} (no EOP)	N_{T0_chain} (with EOP)
8-to-8 Multi-Cycle	15/22	61/63	85/84
8-to-16 Multi-Cycle			
aligned	17/32	63/71	95/92
unaligned	20/32	62/69	98/92
8-to-32 Multi-Cycle			
aligned	18/53	63/90	96/113
unaligned	18/53	60/90	96/113
16-to-8 Multi-Cycle			
aligned	20/23	69/62	108/81
unaligned	20/23	62/60	108/81
16-to-16 Multi-Cycle			
aligned	20/24	90/89	114/112
unaligned	35/50	112/117	129/138
16-to-32 Multi-Cycle			
aligned	35/42	104/103	150/127
unaligned	55/73	123/136	170/158
32-to-8 Multi-Cycle			
aligned	21/25	92/64	87/83
unaligned	21/28	63/65	87/86
32-to-16 Multi-Cycle			
aligned	20/26	93/89	110/110
unaligned	52/66	120/129	142/150
32-to-32 Multi-Cycle			
aligned	24/33	92/74	94/95
unaligned	30/52	118/93	114/114
128-to-128 Multi-Cycle	19/29	63/68	67/75
8-bit Fly-by	27/27	59/59	88/80
16-bit Fly-by	27/27	59/59	88/80
32-bit Fly-by	27/27	59/59	88/80
128-bit Fly-by	27/27	59/59	88/80

Table 13.6. Values of DMA Latency Components

Latency Component	Condition	Value (PCLK2:1 Cycles)	Notes
N_{setup}	Non-chained DMA modes	36	
	Chained DMA modes	44	
	Channel enable delayed from **sdma** execution by > 36 clock cycles in non-chaining mode or > 44 clock cycles in a chained DMA mode.	0	
N_{swap}	Single DMA channel enabled - No channel preemption	0	
	Multiple DMA channels enabled - Preempt lower priority channels	5*(Number of channels preempted)	
N_{lower}	Single DMA channel enabled - No channel preemption	0	(1)
	Multiple DMA channels enabled - Preempt lower priority channel	N_L'	
N_{int}	DMA suspend on interrupt disabled	0	(2)
	DMA suspend on interrupt enabled	Worst-case Interrupt Latency	

NOTES

1. N_L' is the sum of maximum latencies of all channels which may be preempted by the requesting channel. For example, with four DMA channels enabled and rotating priority mode, a channel request may be required to preempt three other channels with pending requests. In this case, the N_L' component is the sum of all of these latencies.
2. This value is defined in the preceding section titled *DMA Latency*.

Worst-case DMA latency is finally calculated as the sum of the individual latency components plus the worst-case throughput condition:

Non-chaining modes:

$$N_{L\_DMA} \text{ (worst case)} = \max(N_T, N_{T\_first}) + N_{setup} + N_{swap} + N_{lower} + N_{int}$$

Chaining modes:

$$N_{L\_DMA} \text{ (worst case)} = N_{T\_chain} + N_{setup} + N_{swap} + N_{lower} + N_{int}$$

Initialization and System Requirements 14

CHAPTER 14
INITIALIZATION AND SYSTEM REQUIREMENTS

This chapter describes the steps that the i960 CA processor takes during its initialization. Discussed are the $\overline{\text{RESET}}$ pin, the reset state of the processor, built-in self test (BIST) features and on-circuit emulation function (ONCE). The 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.

OVERVIEW

During the time that the $\overline{\text{RESET}}$ pin is asserted, the i960 CA 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 $\overline{\text{RESET}}$ pin is deasserted.

When initialization begins, the processor uses an Initial Memory Image (IMI) to establish its state. The IMI contains:

- 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 - several data structure pointers are cached internally at initialization.

The i960 CA processor may be reinitialized by software. 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 processor supports several facilities to assist in system testing and startup diagnostics. The ONCE mode electrically removes the i960 CA processor from a system. This feature is useful for system-level testing where a remote tester exercises the processor system. During initialization, the processor performs an internal functional self test and external bus self test. These features are useful for system diagnostics to ensure base functionality of the i960 CA processor and system bus.

The processor is designed to minimize the requirements of its external system. The processor requires an input clock (CLKIN) and clean power and ground connections (VSS and VCC). 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.

INITIALIZATION

Initialization describes the mechanism that the processor uses to establish its initial state and begin instruction execution. Initialization begins when $\overline{\text{RESET}}$ is deasserted. At this time, the processor automatically configures itself with information specified in the IMI and performs its built-in self test. The processor then branches to the first instruction of user code.

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 has only to perform several base functions to place the processor in a configuration for executing application code.

Reset Operation ($\overline{\text{RESET}}$)

The $\overline{\text{RESET}}$ pin, when asserted (active low), causes the processor to enter the reset state. All external signals go to a defined state (Table 14.1); internal logic is initialized; and certain registers are set to defined values (Table 14.2). When the $\overline{\text{RESET}}$ pin is deasserted, the processor begins initialization as described later in this chapter. $\overline{\text{RESET}}$ is a level-sensitive, asynchronous input.

The $\overline{\text{RESET}}$ 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 VCC must be present and stable for a specified time before the $\overline{\text{RESET}}$ pin can be deasserted.

The processor may also be cycled through the reset state after execution has started. This is referred to as *warm reset*. For a warm reset, the $\overline{\text{RESET}}$ pin must be asserted for a minimum number of clock cycles. Specifications for a cold and warm reset can be found in the *i960 CA Microprocessor Data Sheet*.

The reset state cannot be entered under direct control from a program. No reset instruction – or other condition which forces a reset – exists on the i960 CA processor. The $\overline{\text{RESET}}$ pin must be asserted to enter the reset state. The processor does, however, provide a means to reenter the initialization process. (See *Reinitialization and Relocating Data Structures* later in this chapter.)

Self Test Function (STEST, $\overline{\text{FAIL}}$)

As part of initialization, the i960 CA processor executes a bus confidence self test and, optionally, an internal self test program. The self test (STEST) pin enables or disables internal self test. The failure ($\overline{\text{FAIL}}$) pin indicates that either of the self tests passed or failed.

Internal self test checks basic functionality of internal data paths, registers and memory arrays on-chip. Internal self test is not intended for a full validation of the processor's functionality. Internal self test detects catastrophic internal failures and complements a user's system diagnostics by ensuring a confidence level in the processor before any system diagnostics are executed.

Internal self test is disabled with the STEST pin. Internal self test can be disabled if the initialization time needs to be minimized or if diagnostics are simply not necessary. The STEST pin is sampled on the rising edge of the $\overline{\text{RESET}}$ input. If asserted (high), the processor executes the internal self test; if deasserted, the processor bypasses internal self test. The external bus confidence test is always performed regardless of STEST pin value.

Table 14.1. Pin Reset State

Pins(1)	Reset State	Pins(1)	Reset State
A31:2	Floating	$\overline{\text{DMA}}$	Floating
D31:0	Floating	$\overline{\text{SUP}}$	Floating
$\overline{\text{BE3:0}}$	High (inactive)	$\overline{\text{FAIL}}$	Low (active)
W/$\overline{\text{R}}$	High (inactive)	$\overline{\text{DACK3}}$	High (inactive)
$\overline{\text{ADS}}$	High (inactive)	$\overline{\text{DACK2}}$	High (inactive)
$\overline{\text{WAIT}}$	High (inactive)	$\overline{\text{DACK1}}$	High (inactive)
$\overline{\text{BLAST}}$	High (inactive)	$\overline{\text{DACK0}}$	High (inactive)
DT/$\overline{\text{R}}$	High (inactive)	$\overline{\text{EOP/TC3}}$	Floating (input)
$\overline{\text{DEN}}$	High (inactive)	$\overline{\text{EOP/TC2}}$	Floating (input)
$\overline{\text{LOCK}}$	High (inactive)	$\overline{\text{EOP/TC1}}$	Floating (input)
BREQ	Low (inactive)	$\overline{\text{EOP/TC0}}$	Floating (input)
D/$\overline{\text{C}}$	Floating		

NOTE

(1)Pin states shown assume HOLD and $\overline{\text{ONCE}}$ pins are not asserted. If HOLD is asserted during reset, the hold is acknowledged by asserting HOLDA and the processor pins are configured in the Hold Acknowledge state (See *Chapter 10, Bus Controller.*) If the $\overline{\text{ONCE}}$ pin is asserted, the processor pins are all floated.

Table 14.2. Register Values after Reset

Register(1)	Value after cold reset	Value after warm reset
AC	AC initial image in PRCB	AC initial image in PRCB
PC	C01F2002H	C01F2002H
TC	TC initial image in PRCB	TC initial image in PRCB
FP (g15)	interrupt stack base	interrupt stack base
PFP (r0)	undefined	value before warm reset
SP (r1)	interrupt stack base+64	interrupt stack base+64
RIP (r2)	undefined	undefined
IPND (sf0)	undefined	value before warm reset
IMSK (sf1)	00H	00H
DMAC (sf2)	00H	00H

NOTE

(1)All control registers (not listed) are configured with their respective values from the control table after reset.

External bus confidence self test checks external bus functionality. This test is performed by reading eight words from the Initialization Boot Record (IBR) and performing a checksum on the words and the constant FFFF FFFFH. If the processor calculates a sum of 0, the test passes. The external bus confidence test can detect catastrophic bus failures such as shorted address, data or control lines in the external system. (See *Initial Memory Image.*)

The $\overline{\text{FAIL}}$ pin signals errors in either the internal self test or bus confidence self test. $\overline{\text{FAIL}}$ is asserted (low) for each self test (Figure 14.1). If the test fails, the pin remains asserted and the processor attempts to stop at the point of failure. If the test passes, $\overline{\text{FAIL}}$ is deasserted. When the internal self test is disabled (with the STEST pin), $\overline{\text{FAIL}}$ still toggles at the point where the internal self test would occur even though the internal self test is not executed. $\overline{\text{FAIL}}$ is deasserted after the bus confidence test passes. In Figure 14.1, all transitions on the $\overline{\text{FAIL}}$ pin are relative to PCLK2:1 with output valid equal to t_{OV7}/t_{OH7} as shown in the *i960 CA Microprocessor Data Sheet.*

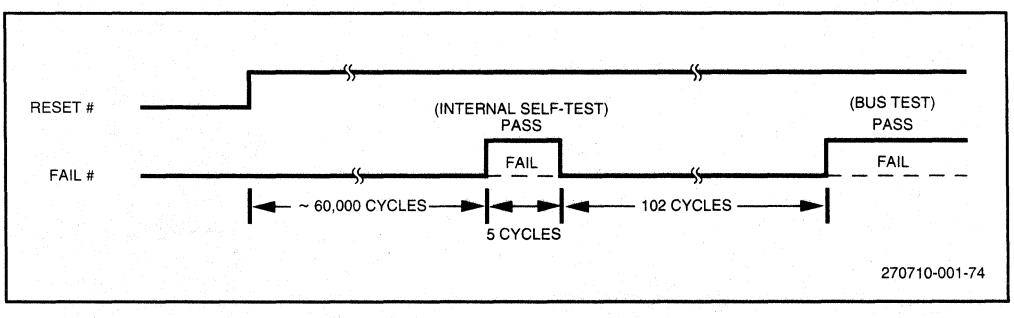


Figure 14.1. $\overline{\text{FAIL}}$ Timing

On-Circuit Emulation

On-circuit emulation aids board level testing. This feature allows a mounted i960 CA processor to electrically remove itself from a circuit board. In ONCE mode, the processor presents a high impedance on every pin, nearly eliminating the processor's power demands on the circuit board. Once the processor is electrically removed, a functional tester can take the place of (emulate) the mounted processor and execute a test of the i960 CA processor system.

The on-circuit emulation mode is entered by asserting (low) the $\overline{\text{ONCE}}$ pin while the i960 CA processor is in the reset state. $\overline{\text{ONCE}}$ pin value is latched on $\overline{\text{RESET}}$ signal's rising edge. The $\overline{\text{ONCE}}$ pin should be left unconnected in an i960 CA processor system. The pin is connected to VCC through an internal pull-up resistor, causing the unconnected pin to remain in the inactive state. To enter on-circuit emulation mode, an external tester simply drives the $\overline{\text{ONCE}}$ pin low (overcoming the pull-up resistor) and initiates a reset cycle. To exit on-circuit emulation mode, the reset cycle must be repeated with the $\overline{\text{ONCE}}$ pin deasserted prior to the rising edge of $\overline{\text{RESET}}$. (See the *i960 CA Microprocessor Data Sheet* for specific timing of the $\overline{\text{ONCE}}$ pin and the characteristics of the on-circuit emulation mode.)

Initial Memory Image (IMI)

The IMI comprises the minimum set of data structures that the processor needs to initialize its system. The IMI performs three functions for the processor:

1. it provides initial configuration information for the core and integrated peripherals
2. it provides pointers to the system data structures and the first instruction to be executed after the processor's initialization
3. it provides checksum words that the processor uses in its self test routine at startup

The IMI is made up of three components: the initialization boot record (IBR), process control block (PRCB) and system data structures. Figure 14.2 shows the IMI components. The IBR is fixed in memory; the other components are referenced directly or indirectly by pointers in the IBR and the PRCB.

Initialization Boot Record (IBR)

The IBR is the primary data structure required to initialize the i960 CA processor. The IBR is a 12-word structure which must be located at address FFFF FF00H (Figure 14.2). The IBR is made up of four components: the initial bus configuration data, the first instruction pointer, the PRCB pointer and the self test checksum data.

When the processor reads the IMI during initialization, it must know the bus characteristics of external memory where the IMI is located. This bus configuration is read from the IBR's first three words. At initialization, the processor performs loads from the lower order byte of the IBR's first three words. These three bytes are combined and loaded into the memory region 0 configuration register (MCON0) to program the initial bus characteristics for the system.

The byte in IBR word 0 is loaded into the lowest byte position of the MCON0 register; the next two bytes from word 1 and word 2 are loaded into successively higher byte positions. The byte in IBR word 4 is reserved and must be set to 00H. This byte is not loaded at initialization (See *Chapter 10, Bus Controller* for a discussion of memory region configuration.)

When initialization begins, the region configuration table valid bit (BCON.ctv) is cleared. This means that every bus request issued takes configuration information from the MCON0 register, regardless of the memory region associated with the request. The MCON0 register is initially set by microcode to a value which allows the bus configuration data in the IBR to be loaded regardless of actual memory configuration. This is done by configuring the external bus with its most relaxed options:

- Non-burst
- Non-pipelined
- Ready disabled
- Bus width = 8 bits
- Little endian byte order
- $N_{RAD} = 31$
- $N_{RDD} = 3$
- $N_{WAD} = 31$
- $N_{WDD} = 31$
- $N_{XDA} = 3$

With this region configuration, the first byte of bus configuration data is loaded from the IBR. This byte is immediately placed into the lower byte of the MCON0 register. This action provides the user-specified N_{RAD}, pipeline control, ready control and burst control values for bus configuration. The remaining configuration data bytes are then read with requests which use the new N_{RAD} value. Once all three bytes are read, MCON0 is rewritten and initialization continues. This reduces the number of clocks required to load the bus configuration data.

The bus configuration data is typically programmed for a system's region 15 bus characteristics. This is done because the remainder of the IBR and the data structures must be loaded using the new bus characteristics and the IBR is fixed in region 15.

As part of 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 can be set in the control table to validate the region table after it is loaded. In this way, the bus controller is completely configured during initialization. (See *Chapter 10, Bus Controller* for a complete discussion of memory regions and configuring the bus controller.)

After the bus configuration data is loaded and the new bus configuration is in place, the processor loads the remainder of the IBR which consists of the first instruction pointer, the PRCB pointer and six checksum words. The PRCB pointer and the first instruction pointer are internally cached. The six checksum words – along with the PRCB pointer and the first instruction pointer – are used in a checksum calculation which implements a confidence test of the external bus. The sum of these eight words plus FFFF FFFFH must equal 0.

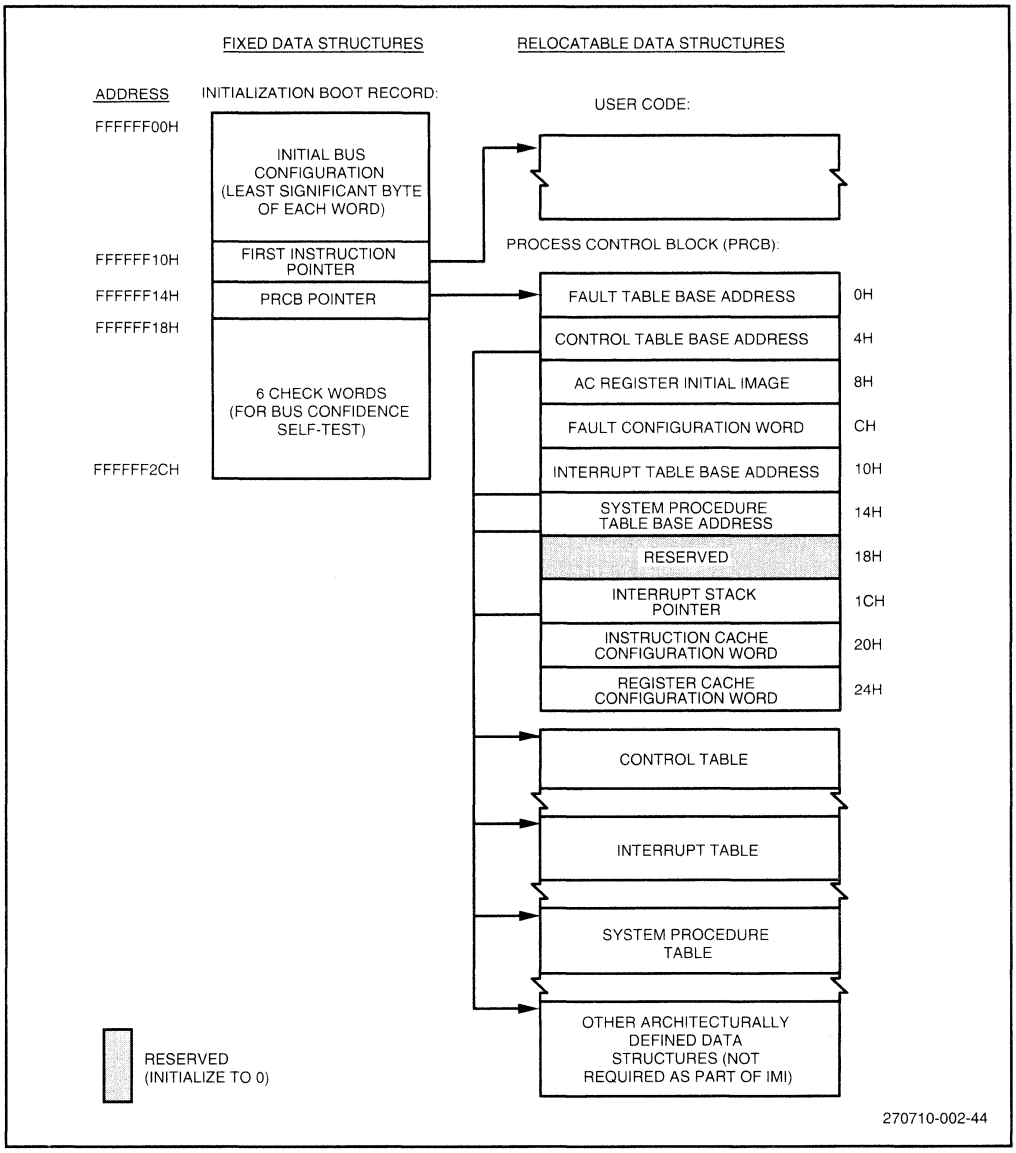

FIXED DATA STRUCTURES
RELOCATABLE DATA STRUCTURES
ADDRESS
INITIALIZATION BOOT RECORD:
FFFFFF00H
INITIAL BUS CONFIGURATION (LEAST SIGNIFICANT BYTE OF EACH WORD)
FFFFFF10H
FIRST INSTRUCTION POINTER
FFFFFF14H
PRCB POINTER
FFFFFF18H
6 CHECK WORDS (FOR BUS CONFIDENCE SELF-TEST)
FFFFFF2CH
USER CODE:
PROCESS CONTROL BLOCK (PRCB):
FAULT TABLE BASE ADDRESS
0H
CONTROL TABLE BASE ADDRESS
4H
AC REGISTER INITIAL IMAGE
8H
FAULT CONFIGURATION WORD
CH
INTERRUPT TABLE BASE ADDRESS
10H
SYSTEM PROCEDURE TABLE BASE ADDRESS
14H
RESERVED
18H
INTERRUPT STACK POINTER
1CH
INSTRUCTION CACHE CONFIGURATION WORD
20H
REGISTER CACHE CONFIGURATION WORD
24H
CONTROL TABLE
INTERRUPT TABLE
SYSTEM PROCEDURE TABLE
OTHER ARCHITECTURALLY DEFINED DATA STRUCTURES (NOT REQUIRED AS PART OF IMI)
RESERVED (INITIALIZE TO 0)
270710-002-44

Figure 14.2. Initial Memory Image (IMI)

Process Control Block (PRCB)

The PRCB contains base addresses for system data structures and initial configuration information for the core and integrated peripherals. The base address pointers are cached in internal registers at initialization. The base addresses are accessed from these internal registers until the processor is reset or reinitialized.

The initial configuration information is programmed in the arithmetic controls (AC) initial image, the register cache configuration word, the fault configuration word and the instruction cache configuration word. These configuration words are shown in Figure 14.3.

The *AC initial image* is loaded into the on-chip AC register during initialization. The AC initial image allows the initial value of the overflow mask, no imprecise faults bit and condition code bits to be selected at initialization.

The AC initial image condition code bits can be used to specify the source of an initialization or reinitialization when a single instruction entry point to the user 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.

The *fault configuration word* allows the operation-unaligned fault to be masked when a non-aligned memory request is issued (See *Chapter 10, Bus Controller* for a description of non-aligned memory requests.) If bit 30 in the fault configuration word is set, a fault is not generated when a non-aligned bus request is issued. The i960 CA processor, in this case, automatically performs the required sequence of aligned bus requests. An application may elect to generate a fault to detect unwanted non-aligned accesses by initializing bit 30 to 0, thus enabling the fault.

The *instruction cache configuration word* allows the instruction cache to be enabled or disabled at initialization. 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. Instruction cache remains disabled until one of two operations is performed:

1. Processor is reinitialized with a new value in the instruction cache configuration word
2. **sysctl** is issued with the configure instruction cache message type and a cache configuration mode other than disable cache.

The *register cache configuration word* specifies the number of register sets cached on-chip. The integrated procedure call mechanism saves the local register set when a call is executed. Local registers are saved to the local register cache. When this cache is full, the oldest set of local registers is flushed to the stack in external memory.

The register cache configuration word least four bits specify the number of local register sets internally cached. The number programmed in this word specifies from 0 to 15 register sets. When more than five register sets are selected, space is taken from internal data RAM for the register cache. (See *Chapter 7, Procedure Calls* for a complete description of the register caching mechanism.)

REQUIRED DATA STRUCTURES

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

At initialization, the processor loads the supervisor stack pointer from the system procedure table and caches the pointer in an internal register. Recall that the supervisor stack pointer is located in the preamble of the system procedure table at byte offset 12 from the base address. The system procedure table base address is programmed in the PRCB. (See *Chapter 5, Procedure Calls* for a description of the system procedure 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. (See *Chapter 2, Programming Environment* for a description of the control table.)

At initialization, the NMI vector is loaded from the interrupt table and saved at location 0000H of the internal data RAM. The interrupt table is typically programmed in the boot ROM and then relocated to RAM by reinitializing the processor. (See *Chapter 6, Interrupts* for a description of NMI and the interrupt table.)

The remaining data structures which an application may need are the fault table, user stack, supervisor stack and interrupt stack. The necessary stacks must be located in a system's RAM. The fault table is typically located in boot ROM. If it is necessary to locate the fault table in RAM, the processor must be reinitialized.

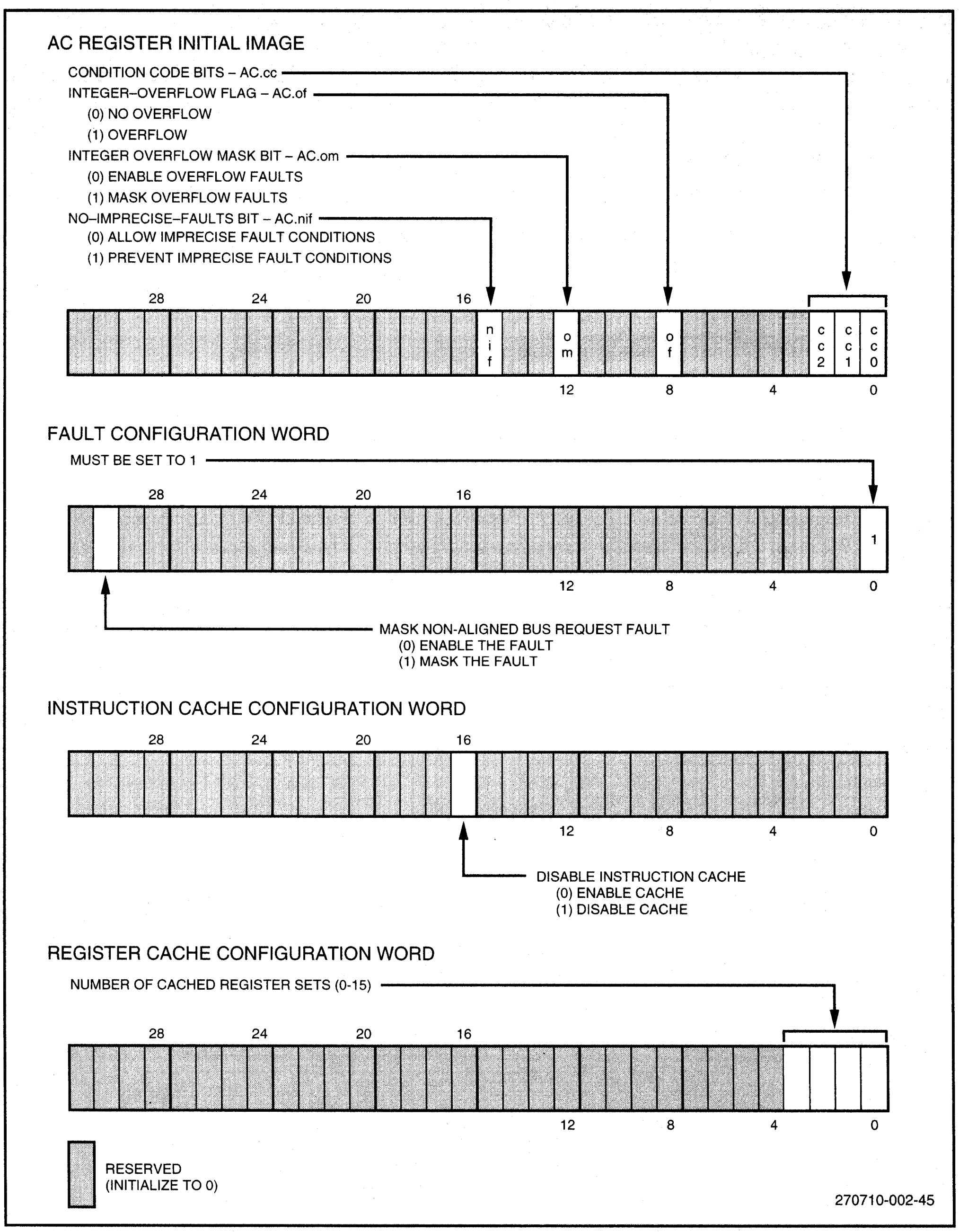

AC REGISTER INITIAL IMAGE
CONDITION CODE BITS – AC.cc
INTEGER–OVERFLOW FLAG – AC.of
(0) NO OVERFLOW
(1) OVERFLOW
INTEGER OVERFLOW MASK BIT – AC.om
(0) ENABLE OVERFLOW FAULTS
(1) MASK OVERFLOW FAULTS
NO–IMPRECISE–FAULTS BIT – AC.nif
(0) ALLOW IMPRECISE FAULT CONDITIONS
(1) PREVENT IMPRECISE FAULT CONDITIONS
28
24
20
16
nif
om
of
cc2
cc1
cc0
12
8
4
0
FAULT CONFIGURATION WORD
MUST BE SET TO 1
1
MASK NON-ALIGNED BUS REQUEST FAULT
(0) ENABLE THE FAULT
(1) MASK THE FAULT
INSTRUCTION CACHE CONFIGURATION WORD
DISABLE INSTRUCTION CACHE
(0) ENABLE CACHE
(1) DISABLE CACHE
REGISTER CACHE CONFIGURATION WORD
NUMBER OF CACHED REGISTER SETS (0-15)
RESERVED
(INITIALIZE TO 0)
270710-002-45

Figure 14.3. Configuration Words in the PRCB

Reinitialization 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 *Chapter 2, Programming Environment* 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 earlier in this chapter.

Reinitialization is useful for relocating data structures to RAM after initialization. The interrupt table must be located in RAM: to post software-generated interrupts, the processor writes to the pending priorities and pending interrupts fields in this table. It may also be necessary to relocate the control table to RAM: it must be in RAM if the control register values are to be changed by the user program. In some systems, it is necessary to relocate other data structures (fault table and system procedure table) to RAM because of poor load performance from ROM. However, these data structures are typically located in a high-performance ROM – such as a burst EPROM – and do not benefit from relocation.

After initialization, the user program is responsible for copying data structures from ROM into RAM. The processor is then reinitialized with a new PRCB which contains the base addresses of the new data structures in RAM.

Reinitialization is required to relocate any of several data structures since the processor caches the pointers to the structures. The processor caches the following pointers during its initialization:

- Interrupt Table Address
- Supervisor Stack Pointer
- Fault Table Address
- PRCB Address
- System Procedure Table Address
- Interrupt Stack Pointer
- Control Table Address

Initialization Flow

This section summarizes initialization by presenting a flow of the steps that the processor takes during initialization (Figure 14.4). The entry point for reinitialization is also shown.

Startup Code Example

After initialization is complete, 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. In this section, an example startup routine and associated declaration files are presented.

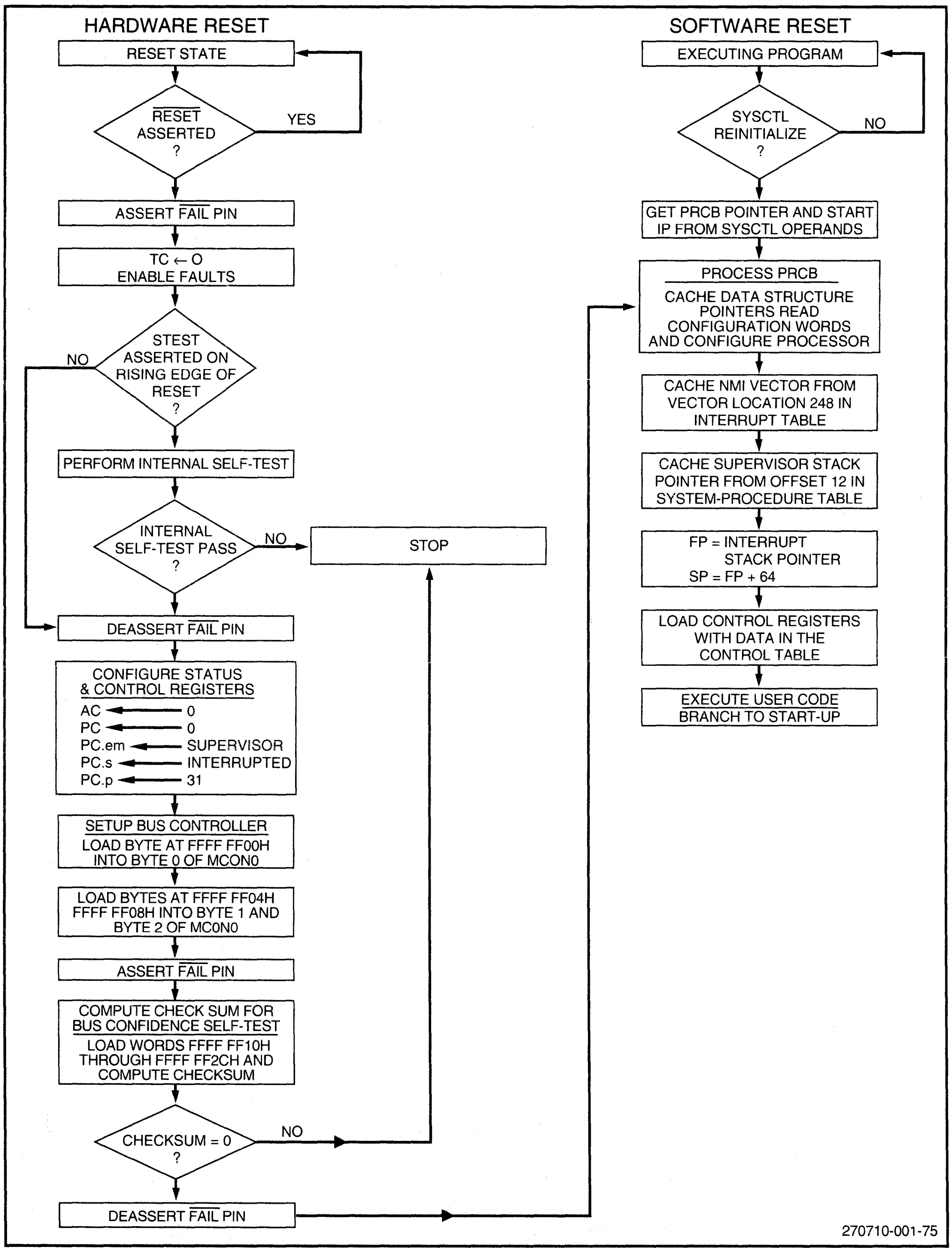

HARDWARE RESET
RESET STATE
RESET ASSERTED ?
YES
ASSERT FAIL PIN
TC ← 0
ENABLE FAULTS
STEST ASSERTED ON RISING EDGE OF RESET ?
NO
PERFORM INTERNAL SELF-TEST
INTERNAL SELF-TEST PASS ?
NO
STOP
DEASSERT FAIL PIN
CONFIGURE STATUS & CONTROL REGISTERS
AC ← 0
PC ← 0
PC.em ← SUPERVISOR
PC.s ← INTERRUPTED
PC.p ← 31
SETUP BUS CONTROLLER
LOAD BYTE AT FFFF FF00H INTO BYTE 0 OF MCON0
LOAD BYTES AT FFFF FF04H FFFF FF08H INTO BYTE 1 AND BYTE 2 OF MCON0
ASSERT FAIL PIN
COMPUTE CHECK SUM FOR BUS CONFIDENCE SELF-TEST
LOAD WORDS FFFF FF10H THROUGH FFFF FF2CH AND COMPUTE CHECKSUM
CHECKSUM = 0 ?
NO
DEASSERT FAIL PIN
SOFTWARE RESET
EXECUTING PROGRAM
SYSCTL REINITIALIZE ?
NO
GET PRCB POINTER AND START IP FROM SYSCTL OPERANDS
PROCESS PRCB
CACHE DATA STRUCTURE POINTERS READ CONFIGURATION WORDS AND CONFIGURE PROCESSOR
CACHE NMI VECTOR FROM VECTOR LOCATION 248 IN INTERRUPT TABLE
CACHE SUPERVISOR STACK POINTER FROM OFFSET 12 IN SYSTEM-PROCEDURE TABLE
FP = INTERRUPT STACK POINTER
SP = FP + 64
LOAD CONTROL REGISTERS WITH DATA IN THE CONTROL TABLE
EXECUTE USER CODE
BRANCH TO START-UP
270710-001-75

Figure 14.4. Processor Initialization Flow

The `startup.s` routine is presented in Example 14.1. Example 14.2 shows the ".ld" file used to locate the IBR, access the link-time variables needed during initialization and set the checksum words. Example 14.3 is a typical minimum declaration file of data structures – including the IBR, PRCB and control table – used in the processor's initialization. Example 14.4 and 14.5 provide useful header files for configuring the bus controller and interrupt controller, respectively. Files from both Example 14.4 and 14.5 are used in Example 14.3.

Example 14.1. Startup Routine

```
/*************************************************************
**************************************************************
****                                                     ****
****    startup.s      80960CA Example initialization    ****
****                                                     ****
****                                                     ****
**************************************************************
*************************************************************/
     .text
     .align  2
     .globl  _start
     .globl  _exit

_start:
      mov     0,g14            /*g14 must be 0 for ic960 C compiler */

/* copy .data from EPROM to RAM */

     lda     _ram_data, r4   /* start address of data in ram */
     lda     _edata, r5      /* end address of data in ram */
     lda     _rom_data, r6   /* start address of data in EPROM*/

_move_data_to_ram:
     mpibg   r4, r5, _move_done
     ld      (r6), r7        /* load data word from ROM */
     addo    r6, 4, r6       /* increment pointer */
     st      r7, (r4)        /* store data to memory */
     addo    r4, 4, r4       /* increment destination */
     b       _move_data_to_ram
_move_done:

     ldconst 0x300, r4        /*select reinit message type */
     ldconst _reinit_ip, r5   /*reinit instruction pointer */
     ldconst _rom_prcb, r6    /*select rom prcb again, could specify
                               a different PRCB with which to
                               reinitialize if desired */
     sysctl  r4, r5, r6        /*execute reinitialization */

     b       _exit

_reinit_ip:

     ldconst 0x0, r4     /* select PC.s = executing (not interrupted)*/
     ldconst 0x2002, r5  /* create mask to change PC.s only */
     modpc   r4, r5, r4  /* change to non-interrupted state */

     ldconst _user_stack, fp /* set first frame to user stack base */
     lda     0x40(fp),sp     /* initialize sp */

      callj  _main           /* call the main routine */

_exit:
      fmark                /* if main returns... */
     b       _exit
```

Example 14.2. Linker Directives File

```
/***************************************************************
****************************************************************
****                                                        ****
****   ca.ld     Example .ld file for an 80960CA system     ****
****                                                        ****
****************************************************************
***************************************************************/

MEMORY
     {
     sram       :  org = 0xB0000000, len = 0x10000     /* 64K */
     dram       :  org = 0xE0000000, len = 0x1000000   /* 1M  */
     eprom      :  org = 0xffff8000, len = 0x7fff      /* 32K */
     }

SECTIONS
     {
     ibr_sec 0xffffff00:
              {
              boot_ca.o     /* locates initial boot record */
              }

     GROUP:
              {
              .text  :
                     {
                     }
              romdata (NOLOAD)  : /* dummy section to set _rom_data
                                  to the end of the .text section */
                     {
                     _rom_data = _etext;
                     }
              } >eprom
     GROUP:
              {
              .data  :
                     {
                     _ram_data = .; /* start address of the data in ram */
                     }
              .bss  :
                     {
                     _user_stack = .;
                     .+= 0x200;
                     _interrupt_stack = .;
                     .+= 0x200;
                     _supervisor_stack = .;
                     .+= 0x200;
                     }
              } >sram
     }
cs1 = -2;     /* we know there will be two carry outs when _start and*/
cs2 = 0x0     /* _rom_prcb are added with the processor's checksum algorithm*/
cs3 = 0x0     /* since both addresses have most of the high order address*/
cs4 = 0x0     /* bits set. We put -2 here to reduce checksum subtotal by two */
cs5 = 0x0     /* to remove the addition of the two carries.*/
/* the following equation causes checksum to go to zero.*/
cs6 = -( _rom_prcb + _start);
```

Example 14.3. Boot-up Data Declarations

```
/************************************************************
*************************************************************
****                                                     ****
****     boot_ca.s    Example IBR and PRCB for 80960CA    ****
****                                                     ****
****     THE INITIALIZATION BOOT RECORD                   ****
****     MUST BE LOCATED AT ADDRESS                       ****
****     0xFFFFFF00 BY THE LINKER!                        ****
****                                                     ****
****     ----->  USE THE C PRE-PROCESSOR                  ****
****                                                     ****
****                                                     ****
*************************************************************
************************************************************/
#include <bus.h>
#include <int.h>
/*---------------------------------------------------------*/
/*  Declare Global Labels                                  */
/*---------------------------------------------------------*/
     .globl      _rom_control_table /* used on hard reset */
     .globl      _rom_prcb          /* used on hard reset */

     .globl  cs1       /* Set all Checksum words in the link file */
     .globl  cs2
     .globl  cs3
     .globl  cs4
     .globl  cs5
     .globl  cs6
/*---------------------------------------------------------*/
     /* Convenient defines to extract bytes */
#define BYTE_0(data)  (data & 0x000000FF)
#define BYTE_1(data)  ((data & 0x0000FF00) >> 8)
#define BYTE_2(data)  ((data & 0x00FF0000) >> 16)
#define BYTE_3(data)  ((data & 0xFF000000) >> 24)
/*---------------------------------------------------------*/
/* Bus Region Table definitions for an example hardware environment      */
/*---------------------------------------------------------*/

/* Standard Byte Wide EPROM */
#define EPROM      (BUS_WIDTH_8 | NRAD(12) | NRDD(0) | NXDA(1) | \
                   NWAD(20) | NWDD(0))

/* Fast Pipelined Static RAM */
#define PSRAM      (PIPELINE_ENABLE | BURST_ENABLE | \
                   BUS_WIDTH_32 | NRAD(0) | \
                   NRDD(0) | NXDA(0) | NWAD(1) | NWDD(1))

/* Burst Dynamic RAM */
#define BDRAM      (READY_ENABLE | BURST_ENABLE | BUS_WIDTH_32 )

/* Misc. Slow 8-bit I/O */
#define I_O        ( BUS_WIDTH_8 | NRAD(12) | NXDA(2) | NWAD(11))

/* iSBX Interface */
#define SBX_0      ( BUS_WIDTH_8 | NRAD(15) | NXDA(3) | NWAD(15))
```

Example 14.3. Boot-up Data Declarations (cont.)

```
/* Place-holder for Empty regions */
#define BUS_CONFIG            EPROM

#define  REGION_0_CONFIG     EPROM
#define  REGION_1_CONFIG     BUS_CONFIG
#define  REGION_2_CONFIG     BUS_CONFIG
#define  REGION_3_CONFIG     BUS_CONFIG
#define  REGION_4_CONFIG     BUS_CONFIG
#define  REGION_5_CONFIG     BUS_CONFIG
#define  REGION_6_CONFIG     BUS_CONFIG
#define  REGION_7_CONFIG     BUS_CONFIG
#define  REGION_8_CONFIG     BUS_CONFIG
#define  REGION_9_CONFIG     BUS_CONFIG
#define  REGION_A_CONFIG     BUS_CONFIG
#define  REGION_B_CONFIG     PSRAM
#define  REGION_C_CONFIG     SBX_0
#define  REGION_D_CONFIG     I_O
#define  REGION_E_CONFIG     BDRAM
#define  REGION_F_CONFIG     EPROM
/*------------------------------------------------------------*/
/* Interrupt Priority Map for an example hardware environment*/
/*------------------------------------------------------------*/

/* Example Interrupt System Configuration  */
#define   ICON_CONFIG \
            (SUSPEND_DMA | FAST_SAMPLE |  VECTOR_CACHE_ENABLE | \
            MASK_UNCHANGED_ALWAYS | I_DISABLE | \
            XINT0_LEVEL | XINT1_LEVEL | XINT2_EDGE | XINT3_EDGE | \
            XINT4_EDGE | XINT5_EDGE | XINT6_LEVEL | XINT7_LEVEL | \
            MIXED_MODE)

/* Example Interrupt Priority Settings */
/* (Specify the full 8-bit vector number, where the least significant nibble
    must be 2.  Such as 0x12, 0x22, 0x32, ..., 0xE2 or 0xF2) */
#define     IMAP0_CONFIG \
               (XINT0_P(0xE2) | XINT1_P(0xD2) | XINT2_P(0xC2) | XINT3_P(0x22))
#define     IMAP1_CONFIG \
               (XINT4_P(0x32) | XINT5_P(0x42) | XINT6_P(0x52) | XINT7_P(0x82))
#define     IMAP2_CONFIG \
               (DMA0_P(0xF2)  | DMA1_P(0xA2)  | DMA2_P(0xB2)  | DMA3_P(0x92))
/*-----------------------------------------------------------*/
/* Define the IBR (Initialization Boot Record)              */
/*-----------------------------------------------------------*/
      text
_init_boot_record:
     .word     BYTE_0(EPROM)     /* Strip the bytes for the IBR Bus Config. */
     .word     BYTE_1(EPROM)
     .word     BYTE_2(EPROM)
     .word     BYTE_3(EPROM)

     .word     _start
     .word     _rom_prcb
     .word     cs1               /* set all checksum words in ".ld" file */
     .word     cs2
     .word     cs3
     .word     cs4
     .word     cs5
     .word     cs6
```

Example 14.3. Boot-up Data Declarations (cont.)

```
/*-----------------------------------------------------------*/
/* Define the Rom-based PRCB for cold starts                 */
/*-----------------------------------------------------------*/
     .text
     .align 4
_rom_prcb:
     .word     _fault_table          /* adr of fault table (ram) */
     .word     _rom_control_table    /* adr of control_table in rom    */
     .word     0x00001000            /* init AC reg mask overflow fault */
     .word     0x40000001            /* Flt - Mask Unaligned fault */
     .word     _interrupt_table      /* Interrupt Table Address */
     .word     _system_proc_table    /* System Procedure Table */
     .word     0                     /* Reserved */
     .word     _interrupt_stack      /* Interrupt Stack Pointer */
     .word     0x00000000            /* Inst. Cache  - enable cache  */
     .word     5                     /* Reg. Cache  - 5 sets cached */
/*-----------------------------------------------------------*/
/* Define the Rom-based Control Table for initialization     */
/*-----------------------------------------------------------*/
     .text
     .align 4
_rom_control_table:
     /* -- Group 0 -- Breakpoint Registers */
     .word     0                     /* IPB0 IP Breakpoint Reg 0 */
     .word     0                     /* IPB1 IP Breakpoint Reg 1 */
     .word     0                     /* DAB0 Data Adr Breakpoint Reg 0 */
     .word     0                     /* DAB1 Data Adr Breakpoint Reg 1 */
     /* -- Group 1 -- Interrupt Map Registers */
     .word     IMAP0_CONFIG          /* IMAP0 Interrupt Map Reg 0 */
     .word     IMAP1_CONFIG          /* IMAP1 Interrupt Map Reg 1 */
     .word     IMAP2_CONFIG          /* IMAP2 Interrupt Map Reg 2 */
     .word     ICON_CONFIG           /* ICON  Interrupt Controller Modes*/
     /* -- Group 2-- Bus Configuration Registers */
     .word     REGION_0_CONFIG
     .word     REGION_1_CONFIG
     .word     REGION_2_CONFIG
     .word     REGION_3_CONFIG
      /* -- Group 3 -- */
     .word     REGION_4_CONFIG
     .word     REGION_5_CONFIG
     .word     REGION_6_CONFIG
     .word     REGION_7_CONFIG
      /* -- Group 4 -- */
     .word     REGION_8_CONFIG
     .word     REGION_9_CONFIG
     .word     REGION_A_CONFIG
     .word     REGION_B_CONFIG
     /* -- Group 5 -- */
     .word     REGION_C_CONFIG
     .word     REGION_D_CONFIG
     .word     REGION_E_CONFIG
     .word     REGION_F_CONFIG
     /* -- Group 6 -- Breakpoint, Trace and Bus Control Registers */
     .word     0                     /* Reserved, set to 0 */
     .word     0                     /* BPCON Breakpoint control reg */
     .word     0                     /* TC Trace Controls Initial Image */
     .word     0x00000001            /* BCON Bus Controller Mode  */
/*-----------------------------------------------------------*/
/* End boot_ca.s                                             */
/*-----------------------------------------------------------*/
```

Example 14.4. Bus Controller Header File

```
/**************************************************************
**************************************************************
****                                                      ****
****      bus.h   header file for 80960CA bus controller  ****
****                                                      ****
****                                                      ****
****                                                      ****
**************************************************************
*************************************************************/
/*----------------------------------------------------------*/
/*  Bus Configuration Defines                               */
/*----------------------------------------------------------*/
#define     BURST_ENABLE       0x1
#define     BURST_DISABLE      0x0

#define     READY_ENABLE       0x2
#define     READY_DISABLE      0x0

#define     PIPELINE_ENABLE    0x4
#define     PIPELINE_DISABLE   0x0

#define     BUS_WIDTH_8        0x0
#define      BUS_WIDTH_16      (0x1 << 19)
#define      BUS_WIDTH_32      (0x2 << 19)

#define      BIG_ENDIAN        (0x1 << 22)
#define     LITTLE_ENDIAN      0x0

#define     NRAD(WS)      (WS << 3)       /* WS can be 0-31     */
#define     NRDD(WS)      (WS << 8)       /* WS can be 0-3      */
#define     NXDA(WS)      (WS << 10)      /* WS can be 0-3      */
#define     NWAD(WS)      (WS << 12)      /* WS can be 0-31     */
#define     NWDD(WS)      (WS << 17)      /* WS can be 0-3      */

/*-----------------------------------------------------------*/
/*  EXAMPLE Region Configuration                             */
/*-----------------------------------------------------------*/
/* Perform a bit-wise OR of the desired parameters to specify a region.

#define     BUS_REGION_1_CONFIG \
     (BURST_ENABLE | BUS_WIDTH_32 | READY_ENABLE | \
     LITTLE_ENDIAN | PIPELINE_ENABLE | \
     NRAD(3) | \
     NRDD(1) | \
     NXDA(1) | \
     NWAD(2) | \
     NWDD(2))
*/
/*-----------------------------------------------------------*/
/*  End bus.h                                                */
/*-----------------------------------------------------------*/
```

Example 14.5. Interrupt Controller Header File

```
/**************************************************************
***************************************************************
****                                                      ****
****  int.h   header file for 80960CA interrupt controller ****
****                                                      ****
****                                                      ****
****                                                      ****
***************************************************************
**************************************************************/
/*-----------------------------------------------------------*/
/*  ICON Defines                                             */
/*-----------------------------------------------------------*/
#define     DEDICATED_MODE      0x0
#define     EXPANDED_MODE       0x1
#define     MIXED_MODE          0x2

#define     XINT0_LEVEL         0x0
#define     XINT0_EDGE        ( 0x1 << 2)
#define     XINT1_LEVEL         0x0
#define     XINT1_EDGE        ( 0x1 << 3)
#define     XINT2_LEVEL         0x0
#define     XINT2_EDGE        ( 0x1 << 4)
#define     XINT3_LEVEL         0x0
#define     XINT3_EDGE        ( 0x1 << 5)
#define     XINT4_LEVEL         0x0
#define     XINT4_EDGE        ( 0x1 << 6)
#define     XINT5_LEVEL         0x0
#define     XINT5_EDGE        ( 0x1 << 7)
#define     XINT6_LEVEL         0x0
#define     XINT6_EDGE        ( 0x1 << 8)
#define     XINT7_LEVEL         0x0
#define     XINT7_EDGE        ( 0x1 << 9)

#define     I_DISABLE         ( 0x1 << 10)
#define     I_ENABLE            0x0

#define     MASK_UNCHANGED_ALWAYS        0x0
#define     SAVE_MASK_DEDICATED        ( 0x1 << 11)
#define     SAVE_MASK_EXPANDED         ( 0x2 << 11)

#define     VECTOR_CACHE_ENABLE      ( 0x1 << 13 )
#define     VECTOR_CACHE_DISABLE       0x0

#define     FAST_SAMPLE           ( 0x1 << 14)
#define     DEBOUNCE                0x0

#define     SUSPEND_DMA           ( 0x1 << 15)
#define     NO_DMA_SUSPEND           0x0
/*-----------------------------------------------------------*/
/*  EXAMPLE Mode Configuration                               */
/*-----------------------------------------------------------*/
/*
Perform a bit-wise OR of the desired parameters to specify configuration.

#define  INT_CONFIG \ (SUSPEND_DMA | FAST_SAMPLE | \ VECTOR_CACHE_ENABLE | \
         SAVE_MASK_DEDICATED | SAVE_MASK_EXPANDED | \ I_DISABLE | \
         XINT0_LEVEL | XINT1_LEVEL | XINT2_EDGE | XINT3_EDGE | \
         XINT4_EDGE  | XINT5_EDGE  | XINT6_LEVEL | XINT7_LEVEL  |   \
         MIXED_MODE)
*/
```

14

Example 14.5. Interrupt Controller Header File (cont.)

```
/*-----------------------------------------------------------*/
/*  IMAP Defines                                             */
/*-----------------------------------------------------------*/
#define     XINT0_P(VNUM)       (VNUM >> 4)
#define     XINT1_P(VNUM)       ((VNUM >> 4) << 4)
#define     XINT2_P(VNUM)       ((VNUM >> 4) << 8)
#define     XINT3_P(VNUM)       ((VNUM >> 4) << 12)
#define     XINT4_P(VNUM)       (VNUM >> 4)
#define     XINT5_P(VNUM)       ((VNUM >> 4) << 4)
#define     XINT6_P(VNUM)       ((VNUM >> 4) << 8)
#define     XINT7_P(VNUM)       ((VNUM >> 4) << 12)

#define     DMA0_P(VNUM)        (VNUM >> 4)
#define     DMA1_P(VNUM)        ((VNUM >> 4) << 4)
#define     DMA2_P(VNUM)        ((VNUM >> 4) << 8)
#define     DMA3_P(VNUM)        ((VNUM >> 4) << 12)
/*------------------------------------------------------------*/
/*  EXAMPLE IMAP Configuration                                */
/*------------------------------------------------------------*/
/*
Perform a bit-wise OR of the desired parameters to specify configuration.
 (Specify the full 8-bit vector number, where the least significant nibble
  is 2.  Such as 0x12, 0x22, ..., 0xE2, 0xF2.) */

#define     IMAP0_CONFIG  \
            (XINT0_P(0xE2) | XINT1_P(0xD2)   |   \
            XINT2_P(0xC2) | XINT3_P(0x22))

*/
/*------------------------------------------------------------*/
/*  End int.h                                                               */
/*------------------------------------------------------------*/
```

SYSTEM REQUIREMENTS

The following sections discuss generic hardware requirements for a system built around the i960 CA processor. This section describes electrical characteristics of the i960 CA processor's interface to the external circuit. The CLKIN, $\overline{\text{RESET}}$, STEST, $\overline{\text{FAIL}}$, $\overline{\text{ONCE}}$, VSS and VCC pins are described in detail. Specific signal functions for the external bus signals, DMA signals and interrupt inputs are discussed in their respective sections in this manual.

Input Clock (CLKIN)

The clock input (CLKIN) determines processor execution rate and timing. The clock input is internally divided by two — or used directly — to produce the external processor clock outputs, PCLK1 and PCLK2. CLKMODE pin state determines whether the input clock is in two-X or one-X mode. When CLKMODE is tied to ground or left floating, the CLKIN input is internally divided by two to produce PCLK2:1 (two-X mode). When CLKMODE is pulled to a logic 1 (high), the CLKIN input is used to create PCLK2:1 at the same frequency, using an internal phase-locked loop circuit (one-X mode). Refer to the *i960 CA Microprocessor Data Sheet* for CLKIN specifications.

The clock input 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 at the processor to prevent overshoot and undershoot. Additionally, a series-damping resistor may be required to damp ringing on the input.

Power and Ground Requirements (VCC, VSS)

The large number of VSS and VCC pins effectively reduces the impedance of power and ground connections to the chip and reduces transient noise induced by current surges. The i960 CA processor is implemented in CHMOS IV technology. Unlike NMOS processes, power dissipation in the CHMOS process is due to capacitive charging and discharging on-chip and in the processor's output buffers; there is almost no DC component of power. The nature of this power consumption results in current surges when capacitors charge and discharge. The i960 CA processor employs 24 VCC and 24 VSS pins to ensure clean on-chip power distribution. The processor's power consumption depends mostly on frequency. It also depends on voltage and capacitive bus load. (See the *i960 CA Microprocessor Data Sheet*).

To reduce clock skew on later versions of the i960 CA processor, the VCC pin for the Phase Lock Loop (PLL) circuit is isolated on the pinout. The lowpass filter shown below reduces CLKIN to PCLK2:1 skew in system designs. This circuit is compatible with those i960 CA processor versions which do not implement isolated PLL power.

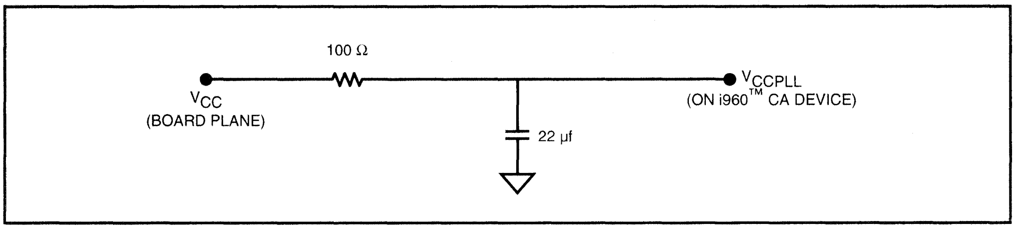
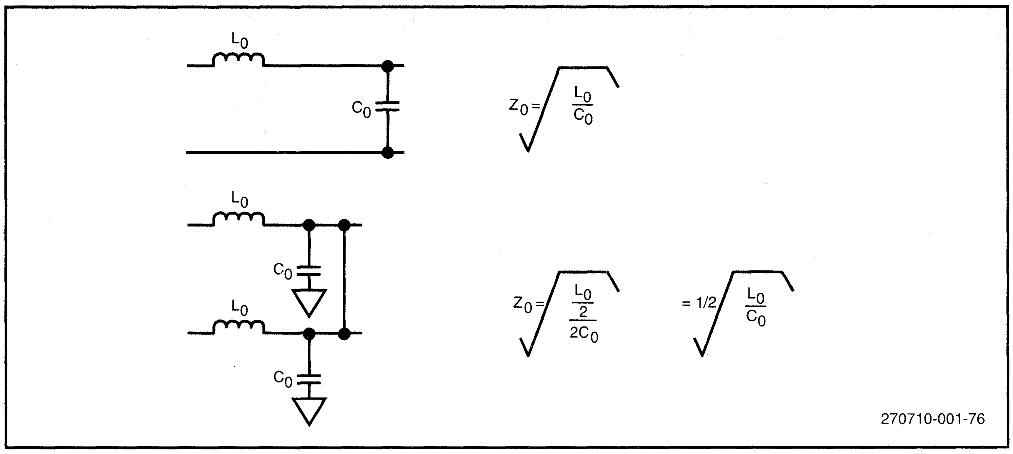


Figure 14.5. VCCPLL Lowpass Filter

Power and Ground Planes

Power and ground planes must be used in i960 CA processor systems to minimize noise. Justification for these power and ground planes is the same as for multiple VSS and VCC 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 14.6, which shows that two lines in parallel have half the impedance of one. To reduce impedance even further, add more lines. Ideally, a plane – an infinite number of parallel lines – results in the lowest impedance. Fabricate ground planes with a minimum of 2 oz. copper.

All power and ground pins must be connected to a plane. Ideally, the i960 CA processor should be located at the center of the board to take full advantage of these planes, simplify layout and reduce noise.



L_0
C_0
$Z_0 = \sqrt{\frac{L_0}{C_0}}$
L_0
C_0
L_0
C_0
$Z_0 = \sqrt{\frac{\frac{L_0}{2}}{2C_0}}$
$= 1/2 \sqrt{\frac{L_0}{C_0}}$
270710-001-76

Figure 14.6. Reducing Characteristic Impedance

Decoupling Capacitors

Decoupling capacitors placed across the device between VCC and VSS reduce voltage spikes by supplying the extra current needed during switching. Place these capacitors close to their devices because connection line inductance negates their effect. Also, for this reason, the capacitors should be low inductance. Chip capacitors (surface mount) exhibit lower inductance and require less board space than conventional leaded capacitors.

I/O Pin Characteristics

The i960 CA processor interfaces to its system through its pins. This section describes the general characteristics of the input and output pins.

Output Pins

All output pins on the i960 CA processor are three-state outputs. Each output can drive a logic 1 (low impedance to VCC); a logic 0 (low impedance to VSS); or float (present a high impedance to VCC and VSS). Each pin can drive an appreciable external load. The *i960 CA Microprocessor Data Sheet* describes each pin's drive capability and provides timing and derating information to calculate output delays based on pin loading.

Output drivers on the i960 CA processor are specially designed to provide a uniform drive current over the entire range of operating temperatures and voltages. This feature eliminates excess noise produced by output drivers under adverse operating conditions.

Input Pins

All i960 CA 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 CA processor inputs are synchronous inputs (Table 14.3). 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 bistable condition to be produced internally. The bistable condition is avoided by qualifying the synchronous inputs with the rising edge of PCLK2:1 or a derivative of PCLK2:1. The *i960 CA Microprocessor Data Sheet* specifies input valid setup and hold times relative to PCLK for the synchronized inputs.

Table 14.3. i960™ CA Processor Input Pins

Synchronous Inputs	Asynchronous Inputs (sampled by PCLK2:1)	Asynchronous Inputs (sampled by $\overline{\text{RESET}}$)
D31:0	$\overline{\text{RESET}}$	STEST
$\overline{\text{READY}}$	$\overline{\text{XINT7:0}}$	$\overline{\text{ONCE}}$
$\overline{\text{BTERM}}$	$\overline{\text{NMI}}$	CLKMODE
HOLD	$\overline{\text{DREQ3:0}}$	
	$\overline{\text{EOP3:0}}$	

i960 CA processor inputs which are considered asynchronous (Table 14.3) are internally synchronized to the rising edge of PCLK2:1. Since they are internally synchronized, the pins only need to be held long enough for proper internal detection. In some cases, it is useful to know if an asynchronous input will be recognized on a particular PCLK2:1 cycle or held off until a following cycle. The *i960 CA Microprocessor Data Sheet* provides setup and hold requirements relative to PCLK2:1 which ensure recognition of an asynchronous input on a particular clock. The data sheet also supplies hold times required for detection of asynchronous inputs.

The $\overline{\text{ONCE}}$, CLKMODE and STEST inputs are asynchronous inputs (Table 14.3). These signals are sampled and latched on the rising edge of the $\overline{\text{RESET}}$ input instead of PCLK2:1.

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. Reflections, interference and noise become significant in comparison to the high-frequency signals. These errors can be transient and therefore difficult to debug. In this section, some high-frequency design issues are discussed; for more information, consult a reference book on high-frequency design.

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 and undershoot occur.

For the i960 CA processor, two termination methods are attractive: AC and series. An AC termination damps the signal at the end of the series line; termination compensates for excess current before the signal travels down the line.

Series termination decreases current flow in the signal path by adding a series resistor as shown in Figure 14.7. 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 ($V = L\ di/dt$), the increased time reduces overshoot and undershoot. Place the series resistor as close as possible to the signal source. Series termination, however, reduces signal rise and fall times, so it should not be used when these times are critical.

AC termination is effective in reducing signal reflection (ringing). This termination is accomplished by adding an RC combination at the signal's destination (Figure 14.8). 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 length and timings that must be met.

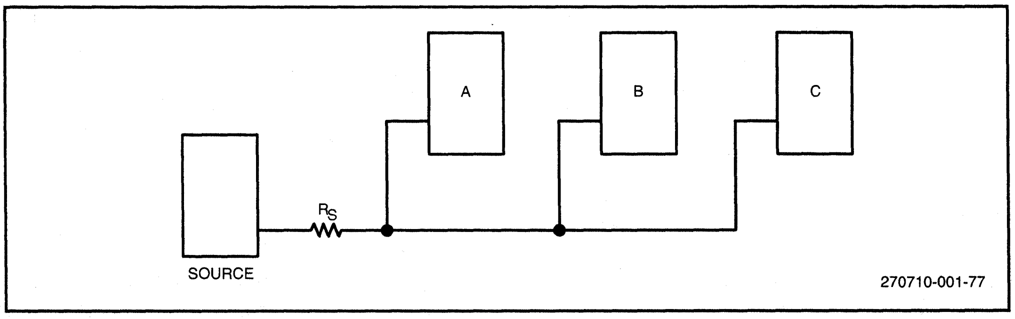


Figure 14.7. Series Termination

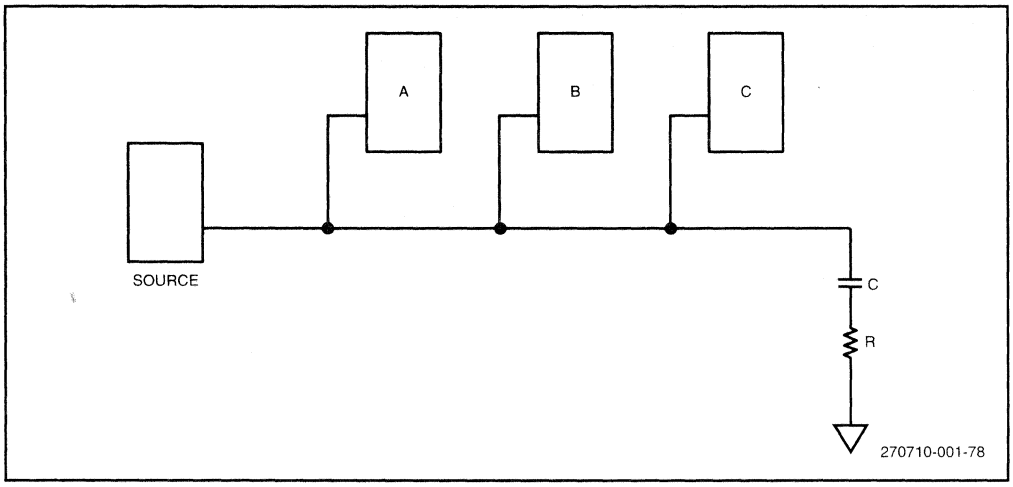


Figure 14.8. AC Termination

Latchup

Latchup is a condition in a CMOS circuit in which VCC becomes shorted to VSS. Intel's CHMOS IV process is 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 CA processor pin or a device connected to an i960 CA processor pin before applying power to the i960 CA 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.

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 two 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 (also called crosstalk) 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 exist:
  - — 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.
- Avoid closed loops in signal paths (Figure 14.9). Closed loops cause excessive current and create inductive noise, especially in the circuitry enclosed by a loop.

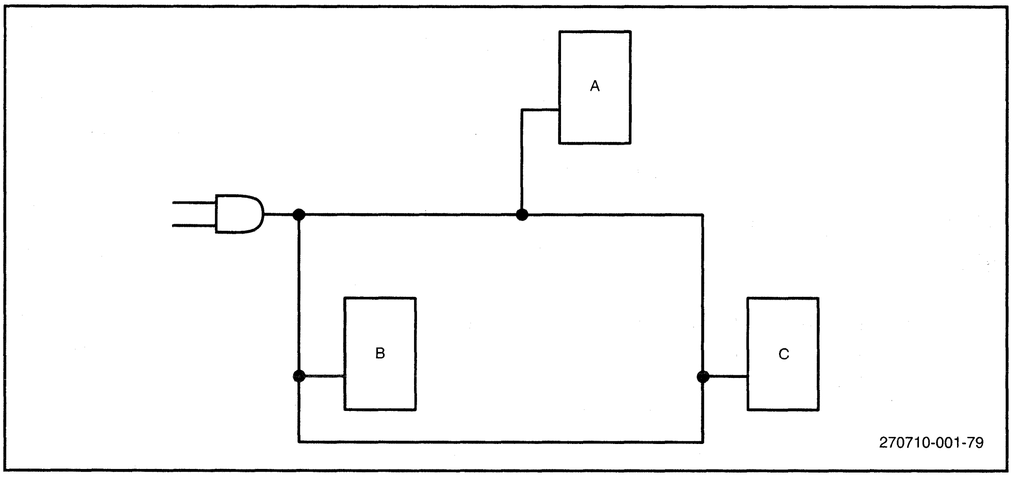


Figure 14.9. Avoid Closed-Loop Signal Paths

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.

Appendix A
Optimizing Code for the i960™ CA Microprocessor

APPENDIX A
OPTIMIZING CODE FOR THE i960™ CA MICROPROCESSOR

This appendix describes the i960 CA microprocessor core's internal construction, also referred to as the core microarchitecture, and core features which enhance this processor's performance and parallelism. This appendix also describes the processor's parallel instruction execution and assembly language techniques for achieving the highest instruction-stream performance.

i960 core microarchitecture defines programming environment, basic interrupt mechanism and fault mechanism for all members of the i960 microprocessor family. The i960 CA processor's core — the C-series core — is a high-performance, highly parallel implementation of the i960 core architecture. The i960 CA processor integrates a bus controller, DMA controller and interrupt controller around the core architecture (Figure A.1).

Processors based on the C-series core can operate at a sustained speed of 66 MIPS (33 MHz clock). State-of-the-art silicon technology and innovative microarchitectural constructs achieve this performance as follows:

- Advanced silicon technology allows operation with a 33 MHz internal clock.
- 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 scoreboarding 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.
- 1 Kbyte of two-way set associative instruction cache is integrated on-chip.
- 1 Kbyte of static data RAM is integrated on-chip.

BASIC CORE STRUCTURE

The i960 CA processor's core contains the following main functional units:

- Instruction Scheduler (IS)
- Register File (RF)
- Execution Unit (EU)
- Multiply/Divide Unit (MDU)
- Address Generation Unit (AGU)
- Data RAM/Local Register Cache

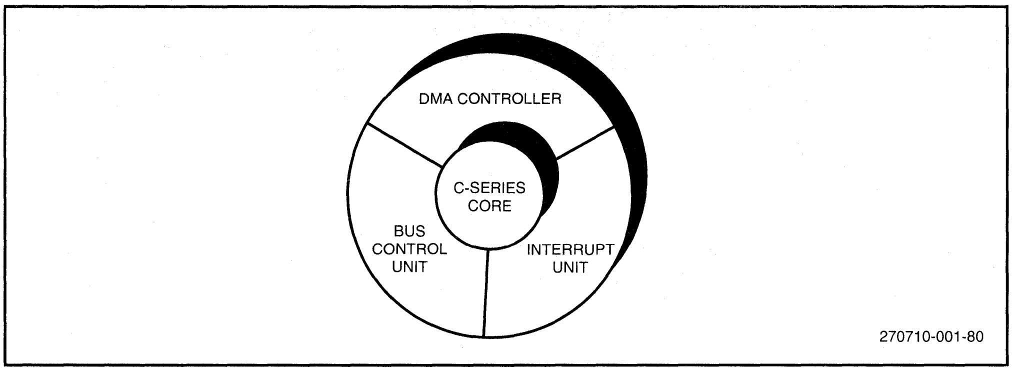


Figure A.1. i960™ CA Processor Core and Peripherals

Figure A.2 shows i960 CA processor's block diagram. The heart of the processor is the IS and RF. 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 processor.

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 CA processor is designed for expansibility by allowing application specific coprocessors to interface to the IS in the same way as the core-defined coprocessors. The integrated peripherals (bus controller, interrupt controller and DMA controller) interface to the REG and MEM side of the i960 CA processor.

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). The IS 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.

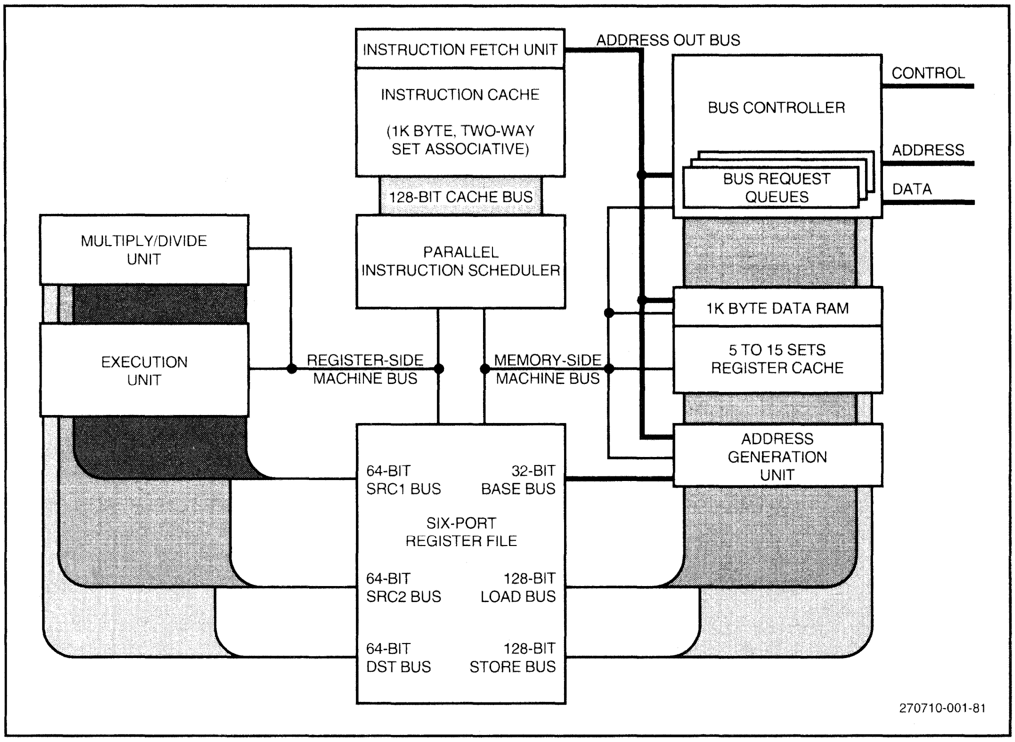


Figure A.2. i960™ CA Microprocessor Block Diagram

The *instruction cache* is a 1 Kbyte, two-way set associative non-transparent cache. This cache delivers up to four instructions per clock to the IS. The cache also allows inner loops of code to execute with no external instruction fetches; this maximizes the core's performance.

The i960 CA processor uses a *microcode ROM* to implement complex instructions and functions. This includes implicit and explicit calls, returns, DMA assists and initialization sequences. Microcode provides an inexpensive and simple method for implementing complex instructions in the processor's RISC environment. Unlike conventional microcode, i960 CA processor's microcode uses a RISC subset of the instruction set in addition to specific microinstructions. 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 CA processor performs this microcode branch in 0 clocks.

Instruction Flow

Most instructions flow through a simple three-stage pipeline (Figure A.3). These stages are referred to as the decode, issue and execute stages:

- Decode stage calculates the next address used to fetch the next instruction from the instruction cache. Additionally, this stage starts decoding the instruction.
- Issue stage completes instruction decode and sends it to the appropriate execution unit.
- During execute stage, the operation is performed and the result is returned to the RF.

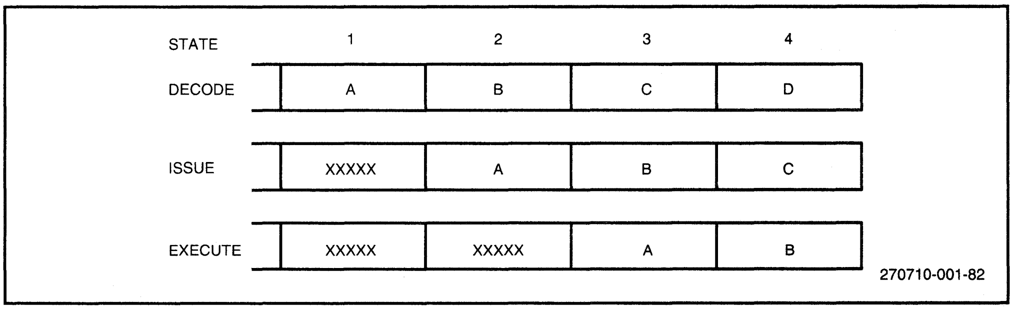


Figure A.3. Instruction Pipeline

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 portion 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 either reserved by a previous incomplete instruction or 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 scoreboarded. A subsequent operation needing that particular register is delayed until the multi-clock operation completes. Instructions which do not use the scoreboarded registers can execute in parallel.

The execute stage executes 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.

Register File (RF)

The RF contains the 16 local and 16 global registers and has six ports (Figure A.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 — per clock — one simple logic or arithmetic instruction, one memory operation (LOAD/STORE) and one address calculation.

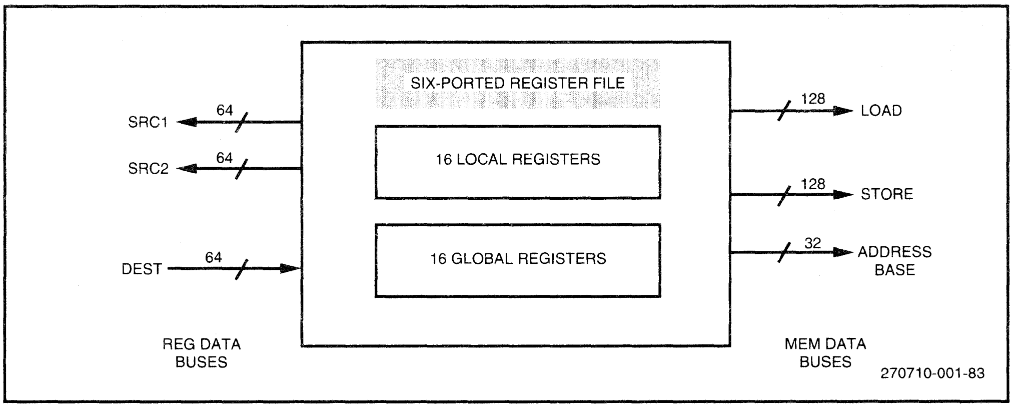


Figure A.4. Six-port Register File

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, in a single clock, up to four words of source data and four words of destination data to simultaneously pass between the RF and a MEM coprocessor.
- provide a high-bandwidth path between data RAM and local register cache to implement high-speed calls, returns and operations in data RAM.
- 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.

Execution Unit (EU)

The EU is the i960 CA 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.

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.

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. The address calculations are performed in parallel with the main execution unit (EU).

Data RAM and Local Register Cache

The Data RAM and Local Register Cache are part of a 1.5 Kbyte block of on-chip Static RAM (SRAM). One Kbyte of this SRAM is mapped into the i960 CA processor's address space from location 00000000H to 000003FFH. 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.

MICROARCHITECTURE REVIEW

At the center of the i960 CA processor core (Figure A.2) is a set of parallel processing units capable of executing multiple single-clock instructions in every clock. To support this rate, the IS can initiate (i.e., 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.

Parallel Issue

To keep the processing units busy, the IS investigates a rolling quad-word group of unexecuted instructions every clock and issues all instructions which can be executed in that clock. The scheduler can issue up to three instructions every clock to the processing units and sustain an issue rate of two instructions per clock.

To maximize the IS's ability to issue instructions in parallel, the instruction cache is organized such that it can provide three or 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.

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 which reference on-chip data-RAM.
AGU	Address Generation Unit executes the **lda** instruction and assists address calculation for all loads and stores; executes **callx**, **bx** and **balx**.

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 **scanbyte** 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 simple (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 ROMed sequence of RISC instructions is issued for execution. This sequence of instructions is called a micro-flow (μ); when taken as a whole, they perform a complex function – i.e., a macro.

Optimizations

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. However, achieving absolutely optimized performance for critical code sequences is made possible by understanding the inner workings of how instructions execute on the processor.

The following section describes instruction execution on the i960 CA processor with the goal of instruction stream optimization in mind. The *Instruction-Stream Optimizations* section describes specific code optimization techniques applicable to the i960 CA processor.

Parallel Instruction Issue

An instruction is executed after being *issued* by the instruction scheduler (IS). The IS keeps the parallel processing units busy by issuing as many new instructions as possible in every clock. To perform this task, the IS looks at the next three or four unexecuted words of the instruction stream every clock and determines which instructions it can issue in parallel. To achieve this parallelism, the IS detects to which machine "side" – REG, MEM or CTRL – each instruction in the current quad-word group belongs and ensures that there are no register dependencies between the instructions.

When the IS issues a group of instructions, the appropriate parallel processing units acknowledge receipt and begin execution. However, register dependencies and resource dependencies could delay instruction execution. The processor transparently manages these interactions through register scoreboarding and register bypassing.

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 the section of this appendix titled *Instruction Cache and Fetch Effects.*

Machine Type Parallelism

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 **ld** 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 **ld** since the IS executes it directly (three instructions per clock).

Figure A.5 shows the paths that the IS has available for dispatching each word of the rolling quad-word to the three machine sides. The IS is not implemented to fully exploit every possible combination of three instruction types in four consecutive words; this would have been prohibitive and many of the possible cases are meaningless.

Table A.1 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.

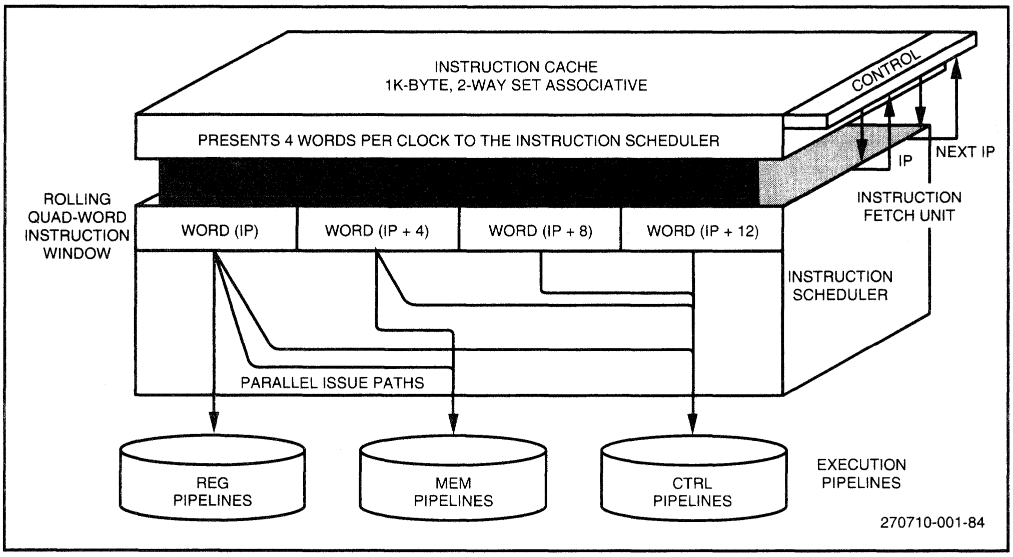


Figure A.5. Issue Paths

Instruction Independence

The scheduler also checks for register dependencies between instructions before issuing them in parallel. The scheduler does not issue a group of instructions if:

1. the same register is specified as a destination more than once
2. the same register is specified as a destination in one instruction and a source in a subsequent instruction

Table A.1. Machine Type Sequences Which 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

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 multi-port register set supports these cases. For example, the following instructions cannot be issued in parallel due to the register dependencies:

```
addo    g0, g1, g2        # g2 is a destination
st      g2, (g3)          # g2 is a source;
                          # store must wait for addo to complete
```

or:

```
addo    g0, 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:

```
addo    g0, g1, g2        # g0 is a source for both instructions
st      g0, (g3)
```

or:

```
addo    g0, 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.

When Instructions are Delayed

In general, 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. There are, however, two conditions in which the execution of one or more of the instructions that the scheduler attempted to issue would be delayed. These conditions are: a *scoreboarded register* or a *scoreboarded resource.*

Scoreboarded Register

If a source (or destination) register of an instruction that the scheduler is attempting to issue is the destination of a prior multi-clock instruction (such as a load) which is not completed, the instruction is delayed. The scheduler attempts to reissue the instruction every clock until the scoreboarded register is updated (e.g., by the BCU) and the delayed instruction can be executed. Table A.2 summarizes conditions which cause a delay due to a scoreboarded register.

Table A.2. Scoreboarded Register Conditions

Condition	Description
src busy	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.
dst busy	The destination referenced by the instruction was referenced as a destination of a prior instruction which has not completed.
cc busy	AC register condition codes are not valid. Correct branch prediction eliminates dead clocks due to condition code dependencies.

Scoreboarded Resource

A scoreboarded resource also thwarts 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 A.3 lists cases which cause an instruction to be delayed due to a scoreboarded resource. Text that follows the table describes what happens to an instruction once it is issued to a processing unit.

Register Scoreboarding and Bypassing

To maintain the logical intent of the sequential instruction stream, the i960 CA processor implements 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. These mechanisms are described to provide an understanding of how the processor operates; it is not necessary to perform any code optimizations to take advantage of this parallel support hardware.

Table A.3. 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	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 DR and BCU attempt to return results over the load bus in the same clock, BCU wins the arbitration. This delays 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. Register scoreboarding works as follows. When the IS issues an instruction which 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 processor uses 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 assessing the same register over a different port. Register bypassing logic constantly monitors all register addresses being written and read. If the same register is being read and written in the same clock, bypass logic – instead of delaying the read – routes incoming data from the write port directly to the read port.

Parallel Execution

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 which they process.

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 A.6. 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 which references the EU's destination register, the new instruction is not issued in the same clock as the EU instruction. As seen in the figure, the new instruction is issued in the clock following the EU operation.

The EU directly executes the instructions listed in Table A.4. The EU is pipelined such that back-to-back EU operations execute at a one-clock sustained rate.

```
addo  g0, g1, g2
shlo  g3, g4, g5
subo  g5, g6, g7
shro  g8, g9, g10
```

INSTRUCTION SCHEDULER	Issue	addo	shlo	subo	shro	
EU PIPELINE	Read src1, src2	g0, g1	g3, g4	g5, g6	g8, g9	
	Execute and Write dst		g2 ← g0+g1	g5 ← g4 << g3	g7 ← g6-g9	g10← g9>>g8

Figure A.6. EU Execution Pipeline

Table A.4. EU Instructions

addo	**shlo**	**mov**	**and**
addi	**shro**	**movl**	**andnot**
addc	**shri**	**cmpo**	**notand**
subo	**shli**	**cmpi**	**nand**
subi	**shrdi**	**cmpdeco**	**or**
subc	**eshro**	**cmpdeci**	**nor**
			ornot
setbit	**alterbit**	**scanbyte**	**notor**
clrbit	**chkbit**		**xnor**
notbit			**xor**
			not
			rotate

NOTE

For these instructions, the EU returns its result to the destination register in the clock following the clock in which the instruction was issued. If a fixup is needed during **shrdi** instruction execution, the processor executes a four-clock micro flow. See Micro flows in this appendix.

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 A.7. 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 immediately follows a multiply — and references the destination of the multiply — is delayed until the fourth clock after the multiply is issued. 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
```

INSTRUCTION SCHEDULER	Issue	addo	mulo	— —	— —	— —	— —	addo		
EU PIPELINE	Read src1, src2	g0, g1						g5, g6		
	Execute and Write dst		g2 ← g0+g1						g7 ← g5+g6	
MDU PIPELINE	Read src1, src2		g3, g4							
	Execute			▨	▨	▨	▨			
	Write dst							g5 ← g3*g4		

270710-001-86

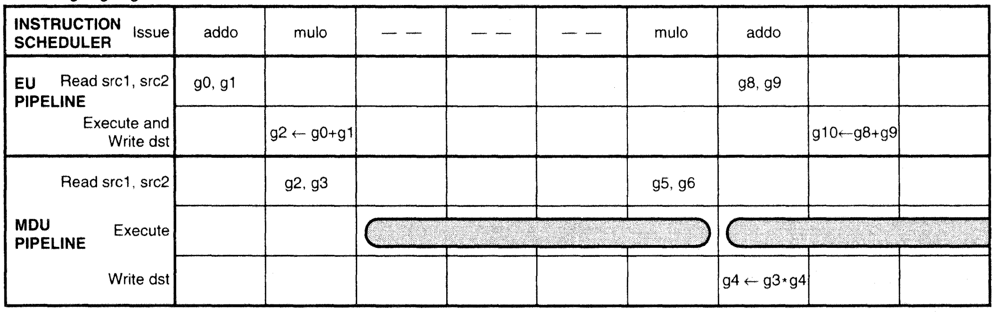
Figure A.7. MDU Execution Pipeline

The MDU incorporates a one-clock pipeline so that 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 A.8 shows the execution pipeline for back-to-back multiplies in which adjacent instructions do not have a register dependency between them.

NOTE

This one-clock pipelining of MDU operations does not occur if integer overflow faults are enabled by the integer overflow mask being set to zero.

```
addo  g0, g1, g2
mulo  g2, g3, g4
mulo  g5, g6, g7
addo  g8, g9, g10
```



270710-001-87

Figure A.8. MDU Pipelined Back-To-Back Operations

The MDU directly executes instructions listed in Table A.5. The scheduler issues an MDU instruction in one clock. The table also shows the latency — the length of the execution stage 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 A.5. MDU Instructions

Mnemonic	Issue Clocks	Result Latency	Back-to-Back Throughput (AC.om = 1)	Back-to-Back Throughput (AC.om = 0)
muli *32x32* *16x32*	1 1	5 3	4 2	5 3
mulo *32x32* *16x32*	1 1	5 3	4 2	4 3
muli *32x32* *16x32*	1 1	6 3	5 2	6 3
divi	13	37	36	36
divo	3	36	35	35
ediv **remi** **remo** **modi**	3	36	35	35

A

Data RAM (DR)

On-chip data RAM (DR) is described in *Chapter 2, Programming Environment*. DR is single-ported and 128-bits wide to support accesses per clock of up to one quad-load or quad-store.

DR receives instructions over the MEM-machine bus; store addresses over the 32-bit Address Out bus; store data over the 128-bit Store bus. DR returns data over the 128-bit Load bus.

The one-clock DR pipeline for reads is shown in Figure A.9. 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 A.6 lists the instructions executed directly using the DR. As seen in Figure A.9, if these instructions are issued back-to-back, they execute at a one-clock sustained rate, with or without register dependencies.

```
addo  g16, g0, g0
ldq   (g0), g4
addo  g4, g5, g6
ldt   (g7), g8
ldq   (g8), g0
```

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	

Figure A.9. Data RAM Execution Pipeline

Table A.6. Data RAM Instructions

Load Latency = 1 clock	Store Latency = 1 clock
ld	st
ldob	stob
ldib	stib
ldos	stos
ldis	stis
ldl	stl
ldt	stt
ldq	stq

NOTE

This table applies to the offset, displacement and indirect memory addressing modes. For other addressing modes, see the Micro flows section of this appendix.

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 unit calculates an effective address (*efa*) which is either: 1) written to a destination register in the case of an **lda** instruction or 2) used as a memory address in the case of loads, stores, extended branches or extended calls.

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 lda Instruction Pipeline

For six of the nine i960 CA processor addressing modes, when a **lda** instruction is issued, the AGU returns the *efa* to the destination register in the following clock. An instruction which immediately follows the **lda** and references the **lda** destination is not issued in the same clock as the **lda**; as shown in Figure A.10, the instruction is issued in the clock in which **lda** is writing the destination register.

Table A.7 lists the **lda** addressing mode combinations that the AGU executes directly. As seen in the figure, if **lda** 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.

```
addo   16, g0, g0
lda    16 (g0), g4
addo   g4, g5, g6
lda    16 [g7*4], g8
lda    16 (g8), g0
```

INSTRUCTION SCHEDULER	Issue	addo	lda	addo lda	lda		
EU PIPELINE	Read src1, src2	16, g0		g4, g5			
	Execute and Write dst		g0 ← g0+16		g6 ← g4+g5		
AGU PIPELINE	Read over Base bus		g0	g7	g8		
	Execute and Write over Ldbus			g4 ← g0+16	g8←(g7*4)+16	g0 ← g8+16	

270710-001-89

Figure A.10. The lda Pipeline

Table A.7. AGU Instructions

Mnemonic	Issue Clocks	Addressing Mode	Result Latency Clocks
lda	1	offset disp (reg) offset(reg) disp(reg) disp[reg * scale]	1

NOTE

For other memory addressing modes, see the Micro flows section of this appendix.

Effective Address (efa) Calculations for Other Operations

When an instruction is issued which requires an effective address (*efa*) calculation, the AGU calculates the *efa* for use by the instruction. 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 A.9), load and store instructions begin immediately for these addressing modes with no delay for address generation. See the section in this appendix titled Micro flows for a description of how other addressing modes are handled.

Bus Control Unit (BCU)

The BCU, as described in *Chapter 10, The Bus Controller*, executes memory requests in two clocks (zero wait state) and returns a result (for loads) on the third clock. Through address pipelining in the system and on-chip request queuing, the BCU is capable of accepting a load or store from the IS every clock and returning load data every clock.

The BCU receives instructions over the MEM-machine bus, store addresses over the 32-bit Address Out bus and store data over the 128-bit Store bus. The BCU returns data over the 128-bit Load bus.

BCU Pipeline

The BCU executes memory operations for load and store instructions, instruction fetches, micro flows and DMA operations; however, its execution pipeline can be easily understood by looking at simple load and store requests. Figure A.11 shows a load instruction execution assuming that: 1) no prior requests were stored in the BCU queues and 2) the worst case that the instruction following the load references the destination of the load.

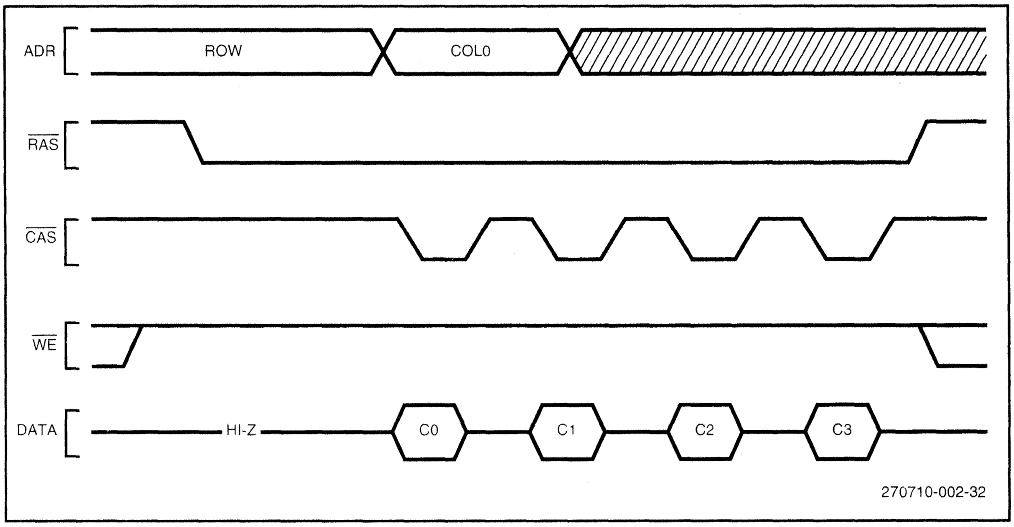


Figure A.11. BCU Pipeline for Loads

The BCU receives the 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 caused a two clock delay in execution of the next instruction because the load data was referenced immediately after the load was issued. If the memory system had wait states, the load data delay would have been longer. If the load were advanced in the code such that it was separated from the instruction which used the data, the load delay could be completely overlapped with the execution of other instructions, even when the system has wait states.

Store instruction execution would proceed as did the load, except that there would be no return clock and no instructions could be stalled due to a scoreboarded register.

Table A.8 lists instructions that the BCU executes directly. 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 external bus. 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.

If instructions listed in the table were issued back-to-back, with no register dependencies, the instructions would execute at a rate of one instruction per clock until the BCU queues were full. Once the queues are full, further back-to-back BCU instructions execute at the bus bandwidth. Figure A.12 shows back-to-back loads being executed.

Table A.8. BCU Instructions

Mnemonic	Issue Clocks	Result Latency Clocks	Back-to-Back Throughput
ld **ldob** **ldib** **ldos** **ldis**	1	3	1
ldl	1	4	2
ldt	1	5	3
ldq	1	6	4
st **stob** **stib** **stos** **stis**	1	N/A	2
stl	1	N/A	3
stt	1	N/A	4
stq	1	N/A	5

NOTE

The table data is valid when the offset, displacement and indirect memory addressing modes over an external bus with the following characteristics (For other addressing modes, see the Micro flows section of this appendix):

$N_{XAD} = N_{XDD} = N_{XDA} = 0$, Burst On, Pipelining On, Ready Disabled

BCU Queues

To allow programs to issue load requests before the data is needed — and thus decouple memory speeds from instruction execution — the BCU contains three queue entries. Each entry stores all the information needed for a memory request. For each request:

- For loads, the source address, destination register number and load type are queued.
- For stores, the destination address, store type and the store data are queued.

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 the 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
addo    g1, g6, g7
```

INSTRUCTION SCHEDULER Issue	ld	ld	ld	addo					
AddressOut bus St bus	g0	g2	g4						
External Address Bus		g0	g2	g4					
BCU PIPELINE External Data Bus			(g0)	(g2)	(g4)				
LD Bus				g1 ← (g0)	g3 ← (g2)	g5 ← (g4)			
EU PIPELINE Read src1, src2				g1, g6					
Execute and Write dst					g7 ← g1+g6				

270710-001-91

Figure A.12. Back-to-Back BCU Accesses

When the DMA controller is enabled, one of the three queue entries is dedicated for DMA operations. This reduces queuing of the instruction stream's loads and stores while improving DMA performance and latency. (See *Chapter 13, DMA Controller*)

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 issuance of instructions to the REG and MEM-machine buses.

Table A.9 lists the instructions that the IS executes directly, without the aid of micro flows. For information on other control flow instructions, see Micro flows later in this appendix.

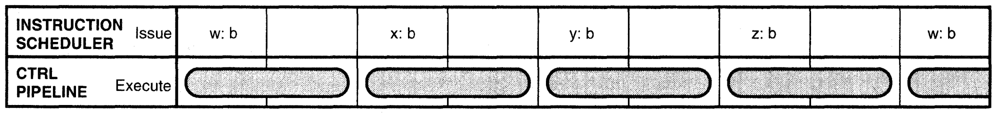
Unconditional Branches

Figure A.13 shows the IS issue stage and the CTRL pipeline for the case where branches branch to branches, essentially 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 A.9. CTRL Instructions

Mnemonic	Issue Clocks	Latency Clocks	Back-to-Back Throughput Clocks
b **be** **bne** **bl** **ble** **bg** **bge** **bo** **bno**	1	2	2

```
w: b   x
 ...
x: b   y
 ...
y: b   z
 ...
z: b   w
```



270710-001-92

Figure A.13. CTRL Pipeline for Branches to Branches

Figures A.14, 15 and 16 show the IS issue stage and the CTRL pipeline for each case of possible IS branch lookahead detection. Assuming that the IS can see four instructions every clock from the instruction cache, the branch can be in the first, second or third group of instructions seen.

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 the Parallel Instruction Issue section earlier in this appendix.

Figure A.14 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 — i.e., 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 A.13.).

```
    b     x
    ...
x:  addo  g0, g1, g2
    lda   2(g3), g4
    b     y
    ...
y:  addo  g5, g6, g7
    lda   2(g8), g9
```

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				

270710-001-93

Figure A.14. Branch in First Executable Group

Figure A.15 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 A.16 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   }
     lda    2(g5), g6   } ← B
     b      y
y:   ...
     addo   g7, g8, g9
     lda    2(g10), g11
```

Group:		A	B					
INSTRUCTION SCHEDULER Issue		addo lda b	lda	addo lda				
CTRL PIPELINE Execute	▨	▨	▨					
EU PIPELINE Read src1, src2		g0, g1		g7, g8				
EU PIPELINE Execute and Write dst			g2 ← g0+g1		g9 ← g7+g8			
AGU PIPELINE Read over Base bus		g3	g5	g10				
AGU PIPELINE Execute and Write over Ldbus			g4 ← g3+2	g6 ← g5+2	g11 ← g10+2			

270710-001-94

Figure A.15. Branch in Second Executable Group

```
     b      x
     ...
x:   lda    2(g3), g4  ) ← A
     addo   g0, g1, g2 ) ← B
     addo   g5, g6, g7 \ ← C
     b      y          )
     ...
y:   addo   g8, g9, g10
     lda    2(g11), g12
```

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			
EU PIPELINE Execute and Write dst				g2 ← g0+g1	g7 ← g5+g6	g10 ← g8+g9		
AGU PIPELINE Read over Base bus		g3			g11			
AGU PIPELINE Execute and Write over Ldbus			g4 ← g3+2			g12 ← g11+2		

270710-001-95

Figure A.16. Branch in Third Executable Group

Conditional Branches

Conditional branches differ from unconditional branches only because the condition codes are sometimes not valid as early as the IS sees the branch instruction. 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; from this it determines which direction the branch will go. No "guess" is required. This situation allows the branch to execute in zero clock cycles, as described in Figure A.16.

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.

Instruction Cache and Fetch Effects

The non-transparent instruction cache is organized to provide any three or four consecutive opwords to the IS on every clock. This capability is critical to the ability to dispatch multiple instructions from the i960 CA 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.

Cache Organization

The 1 Kbyte cache is two-way set associative and organized into two sets of 16 eight-word lines. Each line is composed of four two-word blocks which can be replaced independently.

On every clock, the cache accesses one or two lines and multiplexes the correct three or four words to the IS:

- Three words are valid if the requested address is for an odd word in memory (A2=1)
- Four words are valid if the requested address is for an even word of memory (A2=0).

Fetch Strategy

When any of the three or 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 A.10 describes the fetch decision.

Table A.10. Fetch Strategy

Words Provided To Scheduler				Fetch Initiated	
IP	IP+4	IP+8	IP+12	A3:2 of Requested IP = $0X_2$	A3:2 of Requested IP = $1X_2$
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 Four Words at IP +8
All other cases				Fetch Four Words at IP	Fetch Two Words at IP and Four Words at IP+8

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, 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.

Figure A.17 shows the execution of a two-word fetch that resulted from a cache miss. At the end of the clock in which instructions would be issued had there been a hit, the fetch unit detects the cache miss. The fetch unit issues the instruction fetch in the following clock.

Assuming that the BCU is not busy with another operation, the request begins on the external bus in the next clock. The first word of the fetch is returned to the fetch unit in the clock in which it is received from the memory system; the IS attempts to issue the instruction to an execution unit in that same clock. The remaining words of a fetch are returned as they are received from the system (i.e., one each clock).

If the fetch request is the result of a prefetch decision, the IS is not stalled unless it needed 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, since 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
```

INSTRUCTION SCHEDULER Issue		y: — —	— —	— —	— —	addo	subo		
CTRL PIPELINE Execute		Cache Miss							
AddressOut bus St bus			Fetch Address						
External Address Bus				A					
External Data Bus					D addo	D subo			
BCU PIPELINE Ld Bus						D addo	D subo		
EU PIPELINE Read src1, src2						g0, g1	g3, g4		
Execute and Write dst							g2 ← g0+g1	g5 ← g4-g3	

270710-001-96

Figure A.17. Fetch Execution

Cache Replacement

Data fetched as a result of cache miss is written to the cache when and if the fetched data is requested by the IS. This optimization keeps unexecuted prefetched data from taking up valuable cache space.

As the fetches come in from the BCU, the fetch unit stores incomplete fetch blocks in a queue. If the IS requests one or more instructions which are in the queue, the fetch unit satisfies the queue request. If the queue entry that the scheduler requests contains a full group (two words) of instructions, the valid groups in the queue are also written to the cache in the same clock that they are given to the scheduler. The least recently used set is updated.

Micro-flows

The i960 CA parallel processing units directly execute about half of the processor's instruction encodings. The processor services the remaining complex encodings by executing a sequence of simple instructions from an on-board ROM.

Instruction sequences stored in ROM are written in such a way that enables 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 R, M and C — 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 μ.

This section describes how the complex encodings are detected and execution times associated with each instruction.

Detection

To prevent micro-flowed instruction support from impacting the processor's speed or pipeline depth, 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.

Invocation

Invocation for a complex instruction's 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 makes 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 μ instruction (Figure A.18). If so, the opwords 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 encodings are detected when the opword is fetched, then trapped-out during the clock in which they are 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 (μ) 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 μ 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.

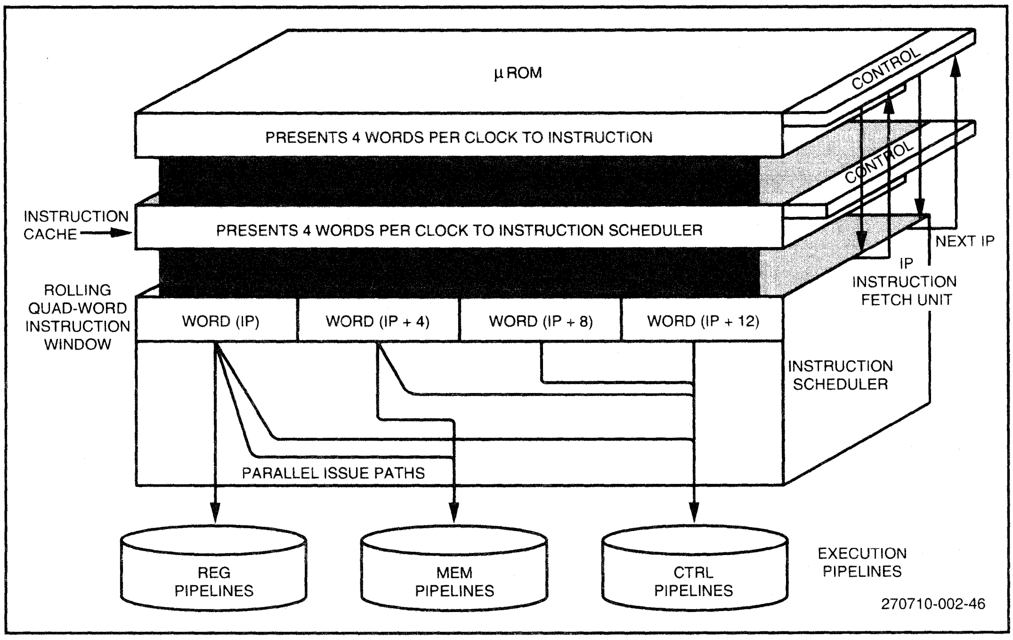


Figure A.18. Micro-flow Invocation

Execution

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 a REG-side instruction. Performance of micro-flowed instructions is described by the number of clocks taken to issue instructions. The following sections describe micro-flowed instruction performance by functional group.

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.

lda takes two clocks to execute for the *(reg)[reg * scale]* and *disp(reg)[reg * scale]* addressing modes and can be issued in parallel with an instruction of machine type R. **lda** using the *disp(IP)* addressing mode takes four clocks to execute and can be issued in parallel with a machine type R instruction. The AGU executes **lda** directly for all other addressing modes.

Load and store instructions are summarized in A.11 and 12. 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 R instruction. To find the result latency of the BCU or DR, see the appropriate section earlier in this appendix.

Table A.11. Load Micro-flow Instruction Issue Clocks

	The following load instructions consume **n** additional issue clocks for address calculation before initiating a load request to the BCU or DR, where **n** for each addressing mode is as follows:		

Mnemonic	*disp(reg)*
	 offset(reg)
	 *disp[reg * scale]*
		*(reg)[reg * scale]*
		 *disp(reg)[reg * scale]*
			disp(IP)

ld **ldob** **ldib** **ldos** **ldis** **ldl** **ldt** **ldq**	1	2	4

NOTE

offset, *disp* and *(reg)* memory addressing modes incur no address calculation overhead. See the Bus Controller and Data RAM sections of this appendix.

Arithmetic

Every arithmetic instruction encoding is directly executed by the EU or MDU parallel processing units.

Logical

Every logical instruction encoding is directly executed by the EU parallel processing unit.

Bit and Bit Field

scanbit, **spanbit**, **extract** and **modify** are executed as micro-flows. Table A.13 lists their execution times. For these instructions, the IS issues **n**-clocks of instructions in place of the single-word i960 CA processor instruction encoding, where **n** is shown in the table.

Table A.12. Store Micro-flow Instruction Issue Clocks

	The following store instructions consume **n** additional issue clocks for address calculation prior to initiating a store request to the BCU or DR, where **n** for each addressing mode is as follows:		
Mnemonic	*disp(reg)* *offset(reg)* *disp[reg * scale]*	*(reg)[reg * scale]* *disp(reg)[reg * scale]*	*disp(IP)*
st **stob** **stib** **stos** **stis** **stl** **stt** **stq**	1	2	4

NOTE

offset, *disp* and *(reg)* memory addressing modes incur no address calculation overhead. See the Bus Controller and Data RAM sections of this appendix.

Table A.13. Bit and Bit Field Micro-flow Instructions

Mnemonic	Execution Clocks (n)
scanbit	1
spanbit	2
extract	4
modify	3

Byte Operations

scanbyte is dirctly executed by the EU parallel processing unit.

Comparison

test* instructions are implemented with a micro-flow. Execution time depends upon condition code validity and prediction bit settings. When condition codes are valid or prediction bit is set correctly, **test*** instructions take one issue clock if the instruction's correct result is a 1 and two issue clocks if the correct result is a 0. Otherwise, the instructions take three issue clocks to execute.

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 bit is set correctly 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 A.14. 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. These instructions may be issued in parallel with a machine-type R instruction.

Table A.14. bx and balx Performance

	The following instructions consume **n** issue clocks before target code is issued, where **n** 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)*
bx **balx**	4	4	6

NOTES

Times shown assume instruction cache hits and a DR-based link target for **balx**.

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, a single-word read and a long-word read access to the system procedure table. The presence of several wait states in these reads directly affect 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.

Table A.15. callx Performance

	The following instruction consumes **n** issue clocks before target code is issued, where **n** 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)*
callx	7	9	9

NOTE

Times shown assume instruction cache hits.

Conditional Faults

fault* instructions are implemented with micro-flows and require one issue clock if the prediction bit is correct and no fault occurred. 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; however, this time should be no longer than 60 clocks for most conditions.

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 switches to the trace fault handler.

Atomic

Atomic instructions are implemented with micro-flows. **atadd** takes seven issue clocks and **atmod** takes eight to execute with an idle bus in a zero-wait state system. Wait states in the memory accessed by these instructions directly affects execution speed.

Processor Management

Processor management instructions implemented as micro-flows include: **modpc**, **modtc**, **modac**, **syncf**, **flushreg**, **sdma**, **udma** and **sysctl**.

modpc requires 17 clocks to execute if process priority is changed and 12 clocks if process priority is not changed. **modac** requires 9; **modtc** requires 15.

syncf takes four issue clocks if there are no possible outstanding faults. Otherwise, the instruction locks the IS until it is certain that no prior instruction that could fault, will fault.

flushreg 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.

sdma executes in 22 clocks; **udma** executes in 4. In the case of back-to-back **sdma** instructions, 40 clocks are required.

sysctl timings are listed in Table A.16. The table lists the times assuming a zero wait-state memory system.

Table A.16. sysctl Performance

Message	Message Type	Issue Clocks
Request Interrupt	00H	37 + bus wait states
Invalidate Cache	01H	38
Configure Cache	02H	52 with 1 Kbyte cache enabled; 48 with 1Kbyte cache disabled. 2078 + bus wait states with load and lock 1Kbyte; 1103 + bus wait states with load and lock 512 bytes.
Reinitialize	03H	243 + bus wait states
Load Control Register Group	04H	42 + bus wait states

Instruction-Stream Optimizations

Embedded applications often benefit from hand-optimized interrupt handlers and critical primitives. This section reviews coding optimizations which arise due to the microarchitecture of the i960 CA instruction set processor. Familiarity with the previous sections of this appendix is assumed and no attempt is made to present techniques which are not specific to the i960 CA processor.

Note that the examples in this section are constructed to illustrate particular optimization tricks. In general, every example could be further optimized by applying several techniques instead of one.

Advancing "Long" Operations

A few operations take multiple clocks to execute in their respective parallel processing units: loads and stores through the BCU and multiplies and divides in the MDU. These instructions consume the least effective execution time (less than one clock) if they are sufficiently separated from the instructions that use their results.

Loads and Stores

Separate load instructions from instructions that use load data. Remember that store instructions can also be reordered. Although they return no results to registers, a poorly placed store in front of a critical load slows down the load. Reorder to issue the load first. Example A.1 shows a simple change that saved one clock from a five clock loop.

Example A.1. Overlap Loads (Checksum)

```
loop:
  ldob        (g0), g1
  addo        g1, g2, g2
  cmpinco     g0, g3, g0
  bl.t        loop
Execution:
```

Clock	REGop	MEMop	CTRLop
1		ldob	
2		|	
3		|	
4	addo		bl.t
5	cmpinco		|
6		ld	

```
opt_loop:
  ldob        (g0), g1
  cmpinco     g0, g3, g0
  addo        g1, g2, g2
  bl.t        opt_loop
Execution:
```

Clock	REGop	MEMop	CTRLop
1		ldob	
2	cmpinco	|	
3		|	bl.t
4	addo		|
5		ld	

Multiplies and Divides

Begin multiply and divide instructions several cycles before instructions that use their results. Also remember to use shift instructions to replace multiplication and division by powers of two. The following example shows overlapping pointer math and a comparison with the 32x32 multiply time in a simple multiply-accumulate loop.

Example A.2. Overlap MDU Operations (Multiply-Accumulate)

```
loop:                                   opt_loop:
  ld          (g0), g2                    ld          (g0), g2
  ld          (g1), g3                    ld          (g1), g3
  muli        g2, g3, g4                  muli        g2, g3, g4
  addi        g4, g5, g5                  addo        4, g0, g0
  addo        4, g0, g0                   cmpo        g0, g6
  addo        4, g1, g1                   addo        4, g1, g1
  cmpobl.t    g0, g6, loop                addi        g4, g5, g5
                                          bl.t        opt_loop
Execution (from DR):                    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
µ11	cmpo		|
12		ld	

Clock	REGop	MEMop	CTRLop
1		ld	
2		ld	
3	muli		
4	|addo		
5	|cmpo		
6	|addo		
7	|		bl.t
8	addi		|
9		ld	

Advancing Comparisons

Where possible, instructions which change the condition codes should be separated from instructions that use the 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 such that branch prediction is not being relied upon to keep the branch-true path executing at nine clocks. Further, 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. Not only does this code economy provide another location in the cache, but the IS may be able to see the upcoming branch earlier since it's encoded in the same opword as the comparison.

Unroll Loops to Use All Registers

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 immediately when the routine is entered and 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 temporaries for operations in various stages. For example, the previous checksum example is repeated here. The loop is unrolled to perform checksums nearly twice as fast as the simple loop.

In general, if the registers are not completely used — or the bus is not saturated with quad operations — more unrolling can be done.

Example A.3. Unroll Loops (Checksum)

```
    -- initialize --
loop:
    ldob            (g0), g1
    addo            g1, g2, g2
    cmpinco         g0, g3, g0
    bl.t            loop
    ret
```

Execution:

Clock	REGop	MEMop	CTRLop
1		ldob	
2		|	
3		|	
4	addo		bl.t
5	cmpinco		|
6		ldob	

```
    -- initialize --
opt_loop:
    ldob            (g0), g1
    cmpinco         g0, g3, g0
    addo            g4, g2, g2
    bge.f           exit1
    ldob            (g0), g4
    cmpinco         g0, g3, g0
    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 g1	
2	cmpinco	|	bge.f
3	addo g4	|	|
4		ldob g4	
5	cmpinco	|	bl.t
6	addo g1	|	|
7		ldob g1	

Enabling Constant Parallel Issue

As described in the Parallel Instruction Issue section of this appendix, certain sequences of instruction machine-types can be executed in parallel, such as: RM, RMC, MC. For example, the checksum loop is repeated with another clock eliminated from code reordering for parallel issue.

Example A.4. Order for Parallelism (Checksum)

```
    -- initialize --
loop:
    ldob        (g0), g1
    addo        g1, g2, g2
    cmpinco     g0, g3, g0
    bl.t        loop
    ret
```

Execution:

Clock	REGop	MEMop	CTRLop
1		ldob	
2		|	
3		|	
4	addo		bl.t
5	cmpinco		|
6		ldob	

```
    -- initialize --
opt_loop:
    addo        g4, g2, g2
    ldob        (g0), g1
    cmpinco     g0, g3, g0
    bge.f       exit1

    ldob        (g0), g4

    cmpinco     g0, g3, g0
    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	addo g4	ldob g1	bge.f
2	cmpinco	|	|
3		ldob g4	
4	cmpinco	|	bl.t
5	addo g1	|	|
6	addo g4	ldob g1	

Migrating from Side to Side

The i960 CA 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. 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 μ).

Table A.17 lists several conversions that can move an instruction to the AGU from either the EU or MDU. Example A.5 exploits the **lda** instruction to increase a 3x3 low-pass filter's performance of an image by approximately 30 percent.

Table A.17. 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 A.5. Change the Type of Instruction Used (3x3 Lowpass Mask)

$$Y[\ ] = X[\ ] \circledast M[\ ]$$

$$M[\ ] = \begin{pmatrix} \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \\ \frac{2}{16} & \frac{4}{16} & \frac{2}{16} \\ \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \end{pmatrix}$$

```
# initial values
# g0 points to X(0,0)
# g1 points to Y(1,1)
# g2 contains imax
# r4 load temp
# r5 accumulator
# r6 = imax (i count temp)
# r7 = jmax (j count temp)
# r8 = imax-1
       # (new mask row offset)
# r9 = 2*imax - 2
       # (new i offset)
# r10 is 2*imax + 1
       # (new j offset)
       b          next_j
next_i:
       subo       r9, g0, g0
next_j:
# first mask row
       ldob       (g0), r5
        addo      1, g0, g0
       ldob       (g0), r4
        addo      1, g0, g0
        shlo      1, r4, r4
        addo      r4, r5, r5
       ldob       (g0), r4
        addo      r4, r5, r5
        addo      r8, g0, g0
# second mask row
       ldob       (g0), r4
        addo      1, g0, g0
        shlo      1, r4, r4
        addo      r4, r5, r5
       ldob       (g0), r4
        addo      1, g0, g0
        shlo      2, r4, r4
        addo      r4, r5, r5
       ldob       (g0), r4
        shlo      1, r4, r4
        addo      r4, r5, r5
        addo      r8, g0, g0
# third mask row
       ldob       (g0), r4
        addo      1, g0, g0
        addo      r4, r5, r5
       ldob       (g0), r4
        addo      1, g0, g0
        shlo      1, r4, r4
        addo      r4, r5, r5
       ldob       (g0), r4
        addo      r4, r5, r5
        shro      4, r5, r5
       stob       r5, (g1)
       addo       1, g1, g1
```

```
# initial values
# g0 points to X(0,0)
# g1 points to Y(1,0)
# g2 contains imax
# r4 load temp
# r5 accumulator
# r6 = imax (i count temp)
# r7 = jmax (j count temp)
# r8 = imax-1
     # (new mask row offset)
# r9 = 2*imax - 2
     # (new i offset)
     # r10 is 2*imax + 1
     # (new j offset)
new_next_i:
new_next_j:

# first mask row
       addo      1, g1, g1
      ldob       (g0), r5
       addo      1, g0, g0
      ldob       (g0), r4
       addo      1, g0, g0
       lda       [r4 * 2], r4
       addo      r4, r5, r5
      ldob       (g0), r4
       addo      r4, r5, r5
       addo      r8, g0, g0
# second mask row
      ldob       (g0), r4
       addo      1, g0, g0
       addo      r4, r5, r5
       lda       [r4 * 2], r4
      ldob       (g0), r4
       addo      1, g0, g0
       lda       [r4 * 4],
       addo      r4, r5, r5
      ldob       (g0), r4
       addo      r8, g0, g0
       lda       [r4 * 2], r4
       addo      r4, r5, r5
# third mask row
      ldob       (g0), r4
       addo      1, g0, g0
       addo      r4, r5, r5
      ldob       (g0), r4
       addo      1, g0, g0
       lda       [r4 * 2], r4
       addo      r4, r5, r5
      ldob       (g0), r4
       addo      r4, r5, r5
       shro      4, r5, r5
       cmpdeco   2, r6, r6
      stob       r5, (g1)
       subo      r9, g0, g0
```

```
# update pointers
        cmpdeco     2, r6, r6
        bg          next_i
        mov         g2, r6
        cmpdeco     2, r7, r7
        subo        r10, g0, g0
        addo        2, g1, g1
        bg          next_j
        ret
```

Execution from DR (loop):

Clock	REGop	MEMop	CTRLop
1	subo		
2		ldob	
3	addo		
4		ldob	
5	addo		
6	shlo		
7	addo		
8		ldob	
9	addo		
10	addo		
11		ldob	
12	addo		
13	shlo		
14	addo		
15		ldob	
16	addo		
17	shlo		
18	addo		
19		ldob	
20	shlo		
21	addo		
22	addo		
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		

```
# update pointers
          bg.t        new_next_i
          addo        r9, g0, g0
          lda         (g2), r6
          cmpdeco     2, r7, r7
          lda         2(g1), g1
          subo        r10, g0, g0
          bg.t        new_next_j
        ret
```

Execution from DR (new loop):

Clock	REGop	MEMop	CTRLop
1	addo	ldob	
2	addo		
3		ldob	
4	addo	lda	
5	addo	ldob	
6	addo		
7	addo	ldob	
8	addo	lda	
9	addo	ldob	
10	addo	lda	
11	addo	ldob	
12	addo	lda	
13	addo	ldob	
14	addo		
15	addo	ldob	
16	addo	lda	
17	addo	ldob	
18	addo		
19	shro		
20	cmpdeco	stob	bg.t
21	subo		|
22	addo	ldob	

A

Branching Optimizations

Conditional branches execute faster if the actual branch direction is correctly predicted using the i960 CA processor branch prediction bits on conditional instructions. Conditional and unconditional branch-target code execute with more parallelism in the first clock if the branch target is long-word or quad-word aligned. (Quad-word is preferable for prefetch efficiency). Branches – specifically the Branch-and-Link instruction – can be used in place of procedure calls to avoid possible frame spills and fills.

Correct Branch Prediction

Setting the prediction bit to indicate the direction that a conditional instruction most often takes improves throughput, especially when the comparison related to the conditional instruction cannot be separated from the test. 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. For example, a code sequence dominated by a jungle of comparisons and conditional branches might see large differences between execution time of the fastest path to slowest path. Prediction bits can be set to provide the best average throughput to: 1) ensure the fastest worst case execution or 2) minimize deviation between slowest and fastest times.

Branch Target Alignment

Since 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. This optimization is easy; however, there are only a few cases where the optimization pays off.

The IS takes advantage of seeing four words on the first clock after a branch (instead of three) when the fourth word is a branch or micro-flow and all three previous opwords are executable in one clock. Example A.6 shows a three-word executable group (**add** followed by **lda** 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 the Micro-flows section of this appendix. Since optimization can save one clock under such circumstances, it could be worthwhile in small loops that execute in only a few clocks, but execute often.

Example A.6. Align Branch Targets

```
        -- initialize --
.align 2
mov g0, g0              #nop
target:
        add             g0, g1
        lda             0xffff ffff, g2
        scanbit         g3, g4
        addo            g5, g6
- more -
Execution:
```

Clock	REGop	MEMop	CTRLop
1			b target
2			|
3	addo	lda	
μ 4	scanbit		
μ 5	|		
6	addo		
7	more		

```
    -- initialize --
    .align 2
target:
    add             g0, g1
    lda             0xffff ffff, g2
    scanbit         g3, g4
    addo            g5, g6
- more -
Execution:
```

Clock	REGop	MEMop	CTRLop
1			b target
2			|
3	addo	lda	
μ 4	scanbit		
5	addo		
6	more		

Compress Code with Branches and bal

bal takes three or four clocks to execute and does not cause a frame spill to memory. Replacing calls with branch_and_links 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 on other processor architectures, the code is often expanded in a straight-line fashion due the branch speed penalties of the original target and the lack of on-chip caching. On the i960 CA processor, branches are virtually free in cached programs and cached program execution is dramatically faster than non-cached code. 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, versus literals. A short code piece could then sequence different filter coefficients through the registers and branch_and_link to the filter loop. The entire routine, which would fit in the instruction cache, could perform a chain of linear filters without a procedure call or cache miss.

Caching

Given the processor's vast ability to consume instructions and execute quad-word memory operations in parallel with arithmetic operations every clock, the instruction cache, register cache and on-chip data RAM are valuable resources for sustaining optimized execution.

Utilizing the Instruction Cache

If an algorithm fits in the instruction cache, it generally executes faster than if it did not fit. This has not always been true with other processors, given the increased number of comparisons and branches that occur when code is compressed.

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 cache is full. Generally, to increase performance of loops which iterate many times and perform memory operations, unroll until all registers are used and/or the cache is full.

Finally, as mentioned in the 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 most efficient from a system throughput point of view. See the section of this appendix titled Instruction Cache and Fetch Effects.

Utilizing On-Chip Register Cache

Register cache can be thought of as a data cache which selectively caches only that data related to procedure context. The section of *Chapter 2, Programming Environment* titled Procedure Call/Return Model describes the i960 CA processor's register cache.

The register cache/data RAM partition is programmable, therefore, the user can determine the tradeoff between the level of procedural context caching versus 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 (frees up) the most on-chip data RAM for variable caching.

Some situations exist where **flushreg** can optimize register cache usage. When an application crosses that imaginary boundary between non-real-time processing to real-time processing, it might be desirable to flush the register set so that initial frame spills are out of the way. A routine which flushes the register cache on entry has the effect of advancing frame spills which might happen within the routine to the beginning of the routine. This approach simply moves the time at which frame spills occur – however, this may actually cause a greater total number of spills to occur than would have otherwise occurred without the premature flush.

flushreg can also control interrupt latency within specific sections of background code. For example, it may be wise to execute a flush at the beginning of a routine which executes a large number of loads from very, very slow memory. This reduces interrupt latency within that code piece since there is no possibility of the interrupt's frame spill being lodged behind slow memory operations.

Usage of this premature flush tactic is very application specific; however, it almost always makes sense to flush the register cache at the beginning of the application's main loop (i.e., after all initialization).

Utilizing On-Chip Data RAM

On every clock, 128 bits of data can be loaded from or stored to the DR. This is a 528 Mb/sec. memory transfer rate (33 MHz clock), which 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. Locations within the DR can be dynamically allocated to leverage scarce DR space and/or globally allocated to achieve minimum latency to critical variables.

Dynamically allocated variables should be those which are used heavily over short periods of time or are used heavily by one procedure. Such variables could be DMA descriptors for the currently active packets or coefficients for filters which process large images on command. Dynamically allocated DR 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 which 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 the following in data RAM: coefficients for a continuously operating filter (e.g., FIR) and/or standard DMA descriptor templates from which run-time descriptors are built.

Summary

Table A.18 summarizes code optimization tactics presented in the previous sections. Figure A.19 is a copy of the execution pipeline template used to create the pipeline examples in this appendix.

Table A.18. 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., **cmpobe**) where possible.
Use Data RAM	Use high-bandwidth data RAM space for performance-critical and/or latency-critical variables.

INSTRUCTION SCHEDULER	Issue									
CTRL PIPELINE	Execute									
EU PIPELINE	Read src1, src2									
	Execute and Write dst									
MDU PIPELINE	Read src1, src2									
	Execute									
	Write dst									
AGU PIPELINE	Read over Base bus									
	Execute and Write over Ldbus									
DR PIPELINE	AddressOut bus St bus									
	Ld bus									
BCU PIPELINE	AddressOut bus St bus									
	External Address bus									
	External Data bus									
	Ld bus									

270710-001-97

Figure A.19. Execution Pipeline Template

Appendix B
Bus Interface Examples

APPENDIX B
BUS INTERFACE EXAMPLES

This appendix 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. All examples assume a 33 MHz bus; issues discussed in each example are independent of operating frequency.

Design examples, state machines and pseudo-code are example only; refer to *EV80960CA Microprocessor Evaluation Board User's Manual* for actual programmable logic equations.

NON-PIPELINED BURST SRAM INTERFACE

This appendix uses a simple SRAM design to demonstrate how the i960 CA processor bus and control signals are used. The design also demonstrates the internal wait state generator. The basic SRAM interface provides the fundamental 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.

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 dedicated as input or output or one set of data pins may be used for both data in and data out. Control signals usually found on SRAM include: Chip Enable ($\overline{CE}$), Output Enable ($\overline{OE}$) and Write Enable ($\overline{WE}$). The following example deals with a SRAM that has $\overline{CE}$, $\overline{OE}$ and $\overline{WE}$ control signals, address inputs and data input/output pins.

The memory is read when $\overline{CE}$ and $\overline{OE}$ are asserted and $\overline{WE}$ is not asserted. The memory is written when $\overline{CE}$ and $\overline{WE}$ are asserted. The $\overline{OE}$ input becomes don't care when $\overline{WE}$ is asserted. However, it is recommended that $\overline{OE}$ is not asserted at the beginning or end of a write cycle; this can lead to bus contention.

Implementation

The following example illustrates a 32-bit burst access SRAM interface. The design may be simplified if burst access modes are not required; it is easily modified for 8- or 16-bit buses.

$\overline{WAIT}$, generated by the internal wait state generator, is used to generate write strobes at the proper place in the write cycle. $\overline{WAIT}$ is used in the address generation circuit to generate mid-burst addresses. External address generation improves performance in burst accesses.

Block Diagram

The 32-bit burst SRAM interface consists of chip select logic, a state machine PLD and write enable logic.

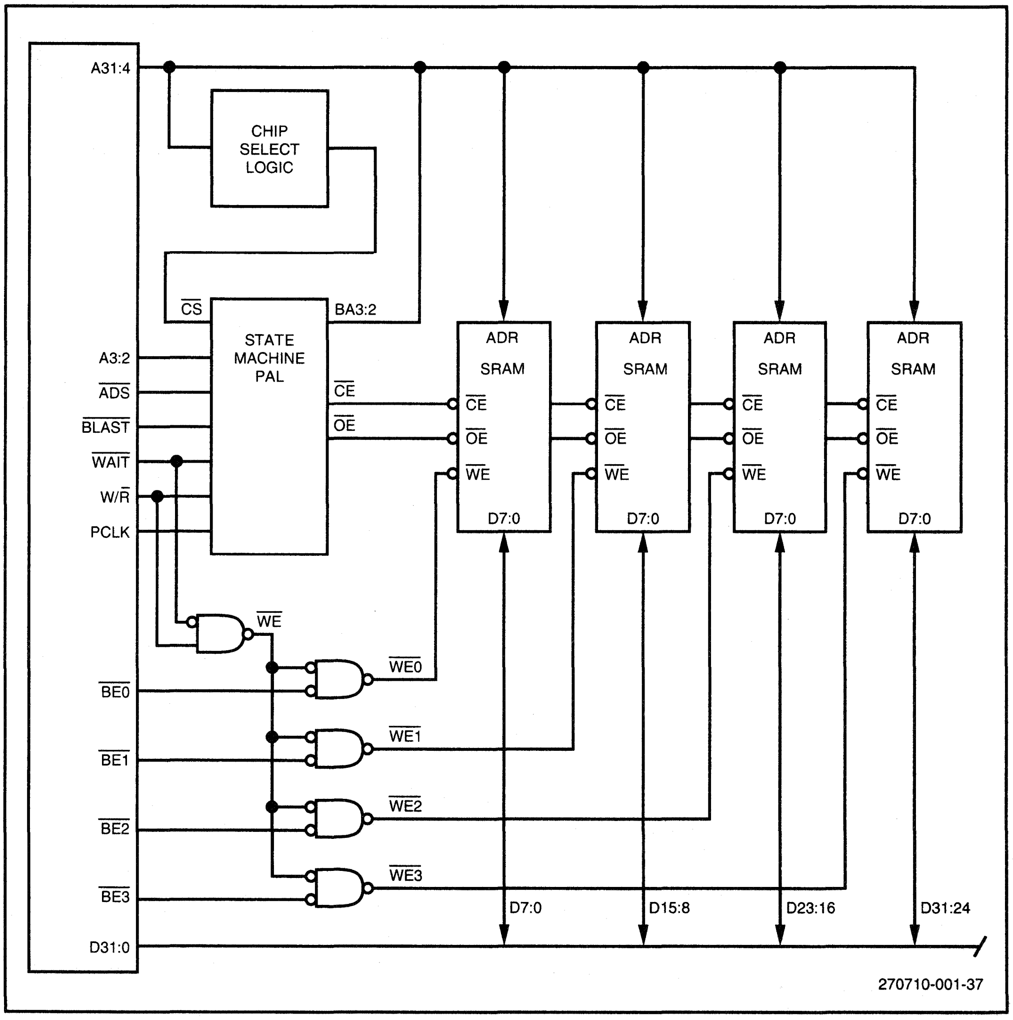


Figure B.1. Non-Pipelined Burst SRAM Interface

Chip Select Logic

Chip select logic is a simple asynchronous data selector; it can be implemented with a demultiplexer or PLD. Chip select ($\overline{CS}$) is based only on the address and is not qualified with any other signals. The state machine PLD qualifies $\overline{CS}$ with $\overline{ADS}$ (see the *Waveforms* section for more discussion on chip select generation.)

State Machine PLD

The SRAM state machine PLD generates the $\overline{\text{CE}}$ and $\overline{\text{OE}}$ signals to the SRAM. This PLD also contains the next-address generation logic; this logic improves burst access performance. The improvement occurs because the i960 CA processor's worst-case address valid delay is longer than the PLD's worst-case delay.

Write Enable Generation Logic

The write enable generation logic generates the $\overline{\text{WE}}$ signal to the SRAM. $\overline{\text{WE}}$ signals are conditioned on the i960 CA processor byte enables ($\overline{\text{BE3:0}}$), the write/read signal (W/$\overline{\text{R}}$) and the wait signal ($\overline{\text{WAIT}}$).

There is a write enable signal, $\overline{\text{WE3:0}}$, for each byte position corresponding to the byte enable signals, $\overline{\text{BE3:0}}$; this allows byte, short-word and word-wide writes. Read accesses to this memory system always result in word reads. The i960 CA device, in the case of byte- or short-word reads, reads the data from the correct place on the data bus.

Chip Select Generation

$\overline{\text{ADS}}$ assertion during the PCLK rising edge indicates the address is valid. Address setup time to this clock edge is PCLK period (T_{PP}), minus address output delay (T_{OV}). $\overline{\text{CS}}$ signal generation time $\overline{\text{CS}}_{\_gen}$) must satisfy the input setup time of the State Machine PLD(T_{PLD_setup}). Therefore:

$$\overline{\text{CS}}_{\_gen} = T_{PP} - T_{OV} - T_{PLD\_setup}$$

Waveforms

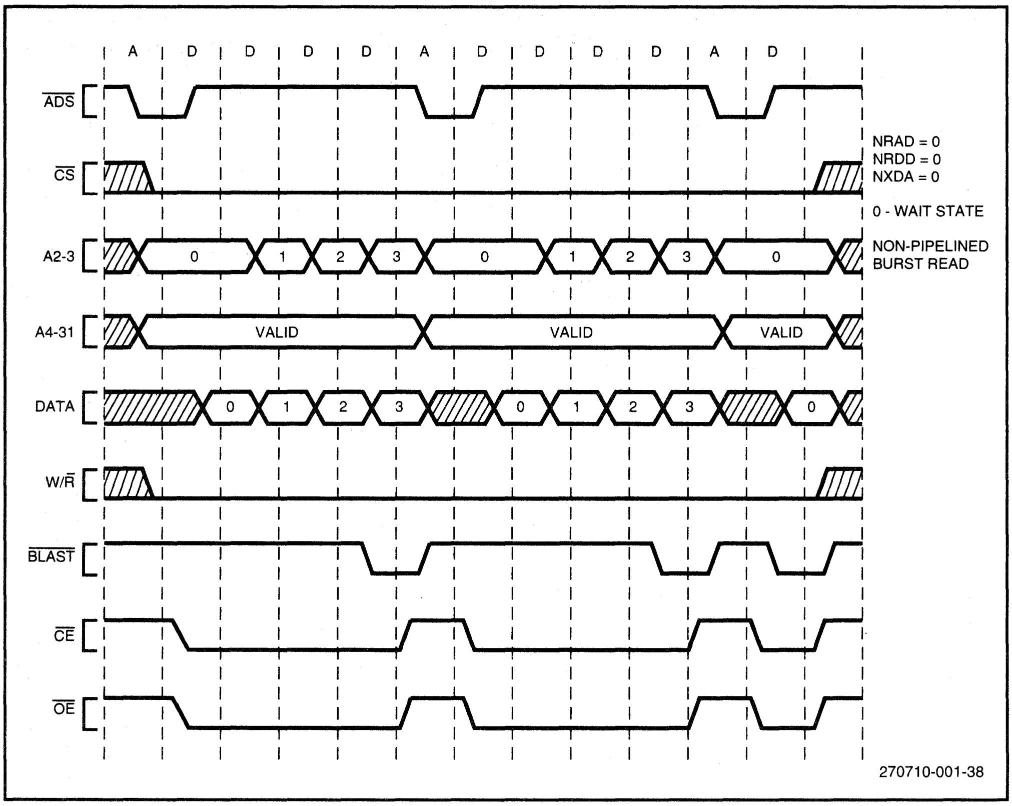


Figure B.2. Non-Pipelined SRAM Read Waveform

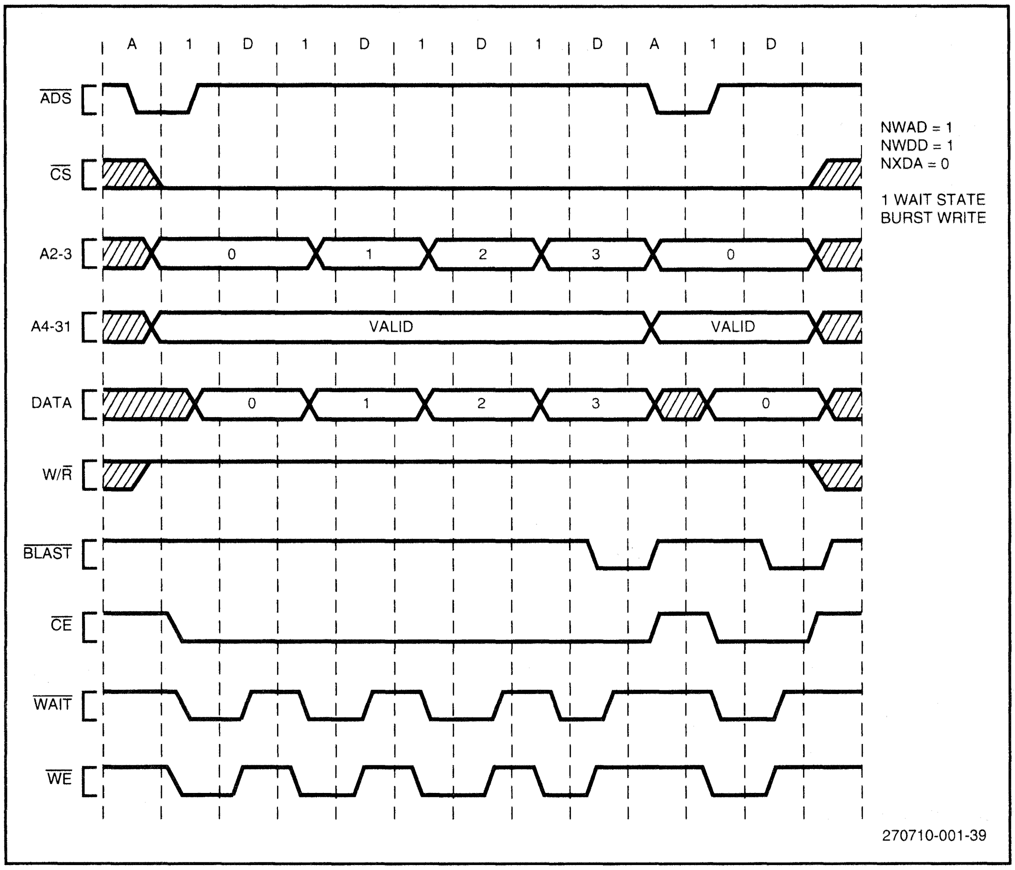


Figure B.3. Non-Pipelined SRAM Write Waveform

Wait State Selection

The i960 CA 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 CA processor, then it is important to consider address output delay from the i960 CA device. 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. 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 CA processor.

Output Enable and Write Enable Logic

The output enable signal is simply (see Figure B.1):

$$\overline{OE} = W/\overline{R}$$

The PLD is used to buffer the $W/\overline{R}$ signal; this may be necessary to reduce the load on the $W/\overline{R}$ signal.

The write enable signals are:

$$\overline{WE} = !(WAIT \;\&\; W/\overline{R});$$

or

$$\overline{WE0} = !(WE \;\&\; \overline{BE0});$$
$$\overline{WE1} = !(WE \;\&\; \overline{BE1});$$
$$\overline{WE2} = !(WE \;\&\; \overline{BE2});$$
$$\overline{WE3} = !(WE \;\&\; \overline{BE3});$$

The $\overline{WAIT}$ signal is used to create the write strobe. When $W/\overline{R}$ indicates a write and $\overline{BEx}$ and $\overline{WAIT}$ are asserted, the logic asserts $\overline{WE}$. The *i960 CA Microprocessor Data Sheet* guarantees a relationship from $\overline{WAIT}$ high to write data invalid.

State Machine Descriptions

The state machine PLD incorporates two state machines: one controls SRAM chip enable ($\overline{CE}$); the other generates the A3:2 address signals for multiple word burst accesses.

The chip enable state machine controls the $\overline{CE}$ signal. $\overline{CE}$ is normally not enabled, but when both $\overline{ADS}$ and $\overline{BSRAM\_CS}$ are asserted, $\overline{CE}$ is asserted and remains asserted until $\overline{BLAST}$ is asserted. $\overline{BLAST}$ indicates the access is complete. $\overline{CE}$ is the output of the state register; therefore, the $\overline{CE}$ output delay is the clock-to-output time of the PLD. Minimizing $\overline{CE}$ delay provides more memory access time.

The A3:2 address generation state machine 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.

The burst address outputs (BA3:2) correspond to registers within the PLD. Address generation time then corresponds to the clock-to-output time of the PLD. The BA3:2 signals are forced to 0 when $\overline{BLAST}$ is asserted.

The pseudo-code description that follows the figure is provided only to describe the state machine diagram. It is not intended to be PLD equations. A trailing # indicates a signal is asserted low.

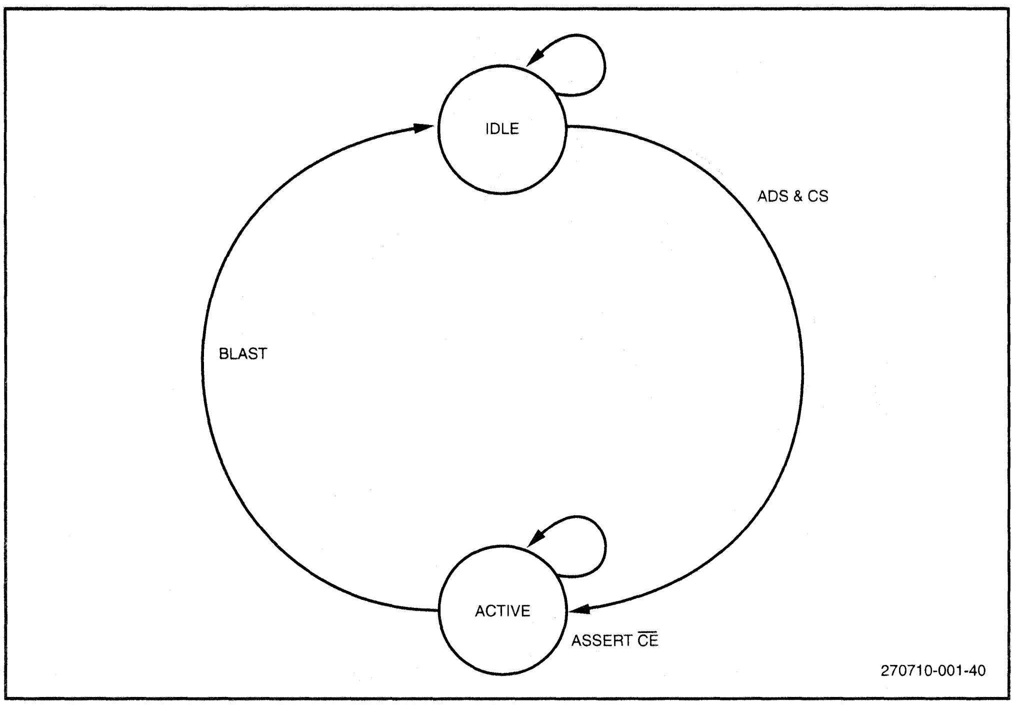


Figure B.5. A32 Address Generation State Machine

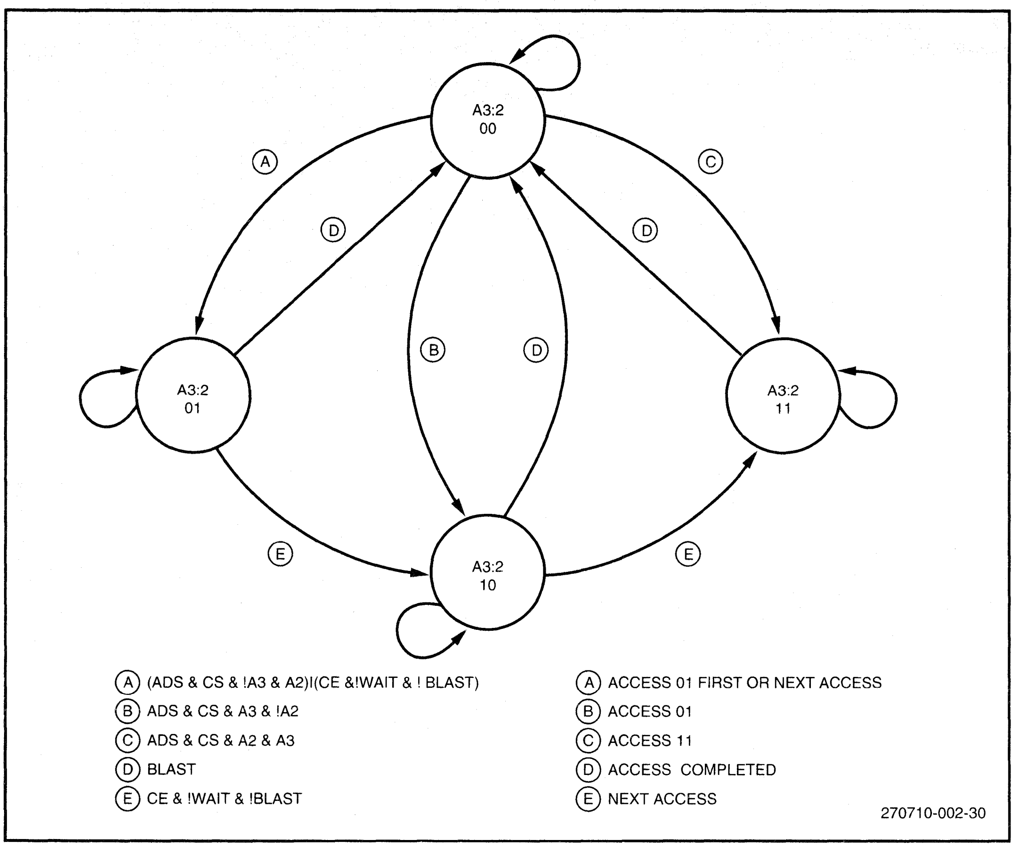


Figure B.4. Chip Enable State Machine

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;
```

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 A3:2. The assertion of CE indicates that an access is underway. 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.

Tradeoffs and Alternatives

The SRAM example just described demonstrates a burst SRAM memory interface. If a non-burst interface is desired, the address generation section of the state machine PLD may be removed. The design is also easily expanded to accommodate multiple banks of SRAM.

The i960 CA processor integrated bus controller simplifies external memory system design. The internal wait state generator decouples the memory speed from the memory controller. The memory control PLD does not use any of the memory access parameters. So, operation of the memory control PLD is independent of memory access times. Memory access parameters are entered into the Memory Region Configuration Table via software.

PIPELINED READ SRAM INTERFACE

The following example illustrates the implementation of a pipelined read SRAM system. A zero wait state pipelined read memory system will have 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; the only major addition is an address latch.

A pipelined read memory system is the highest performance memory system that can be interfaced to the i960 CA processor. The address cycle of consecutive accesses is overlapped with the data cycle of the previous access. This results in the maximum bandwidth utilization of the bus. (See Figure B.6.)

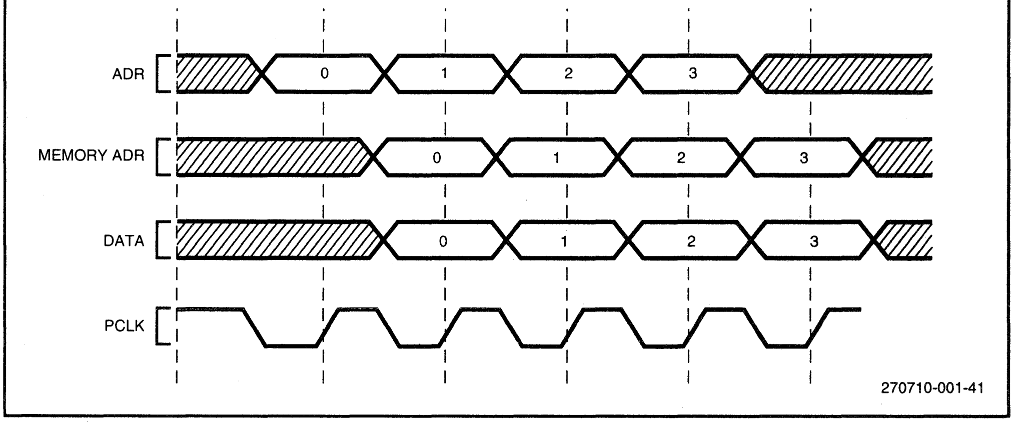


Figure B.6. Pipelined Read Address and Data

Block Diagram

The same SRAM used in a non-pipelined read memory system is used in a pipelined read memory system. Figure B.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 $W/\overline{R}$ latch have been added.

Chip select logic is a simple asynchronous data selector. Chip select ($\overline{CS}$) is based only on the address and is not qualified with any other signals. (See the section in this appendix titled *Non Pipelined 1 -Burst SRAM* example for more information on chip select generation.)

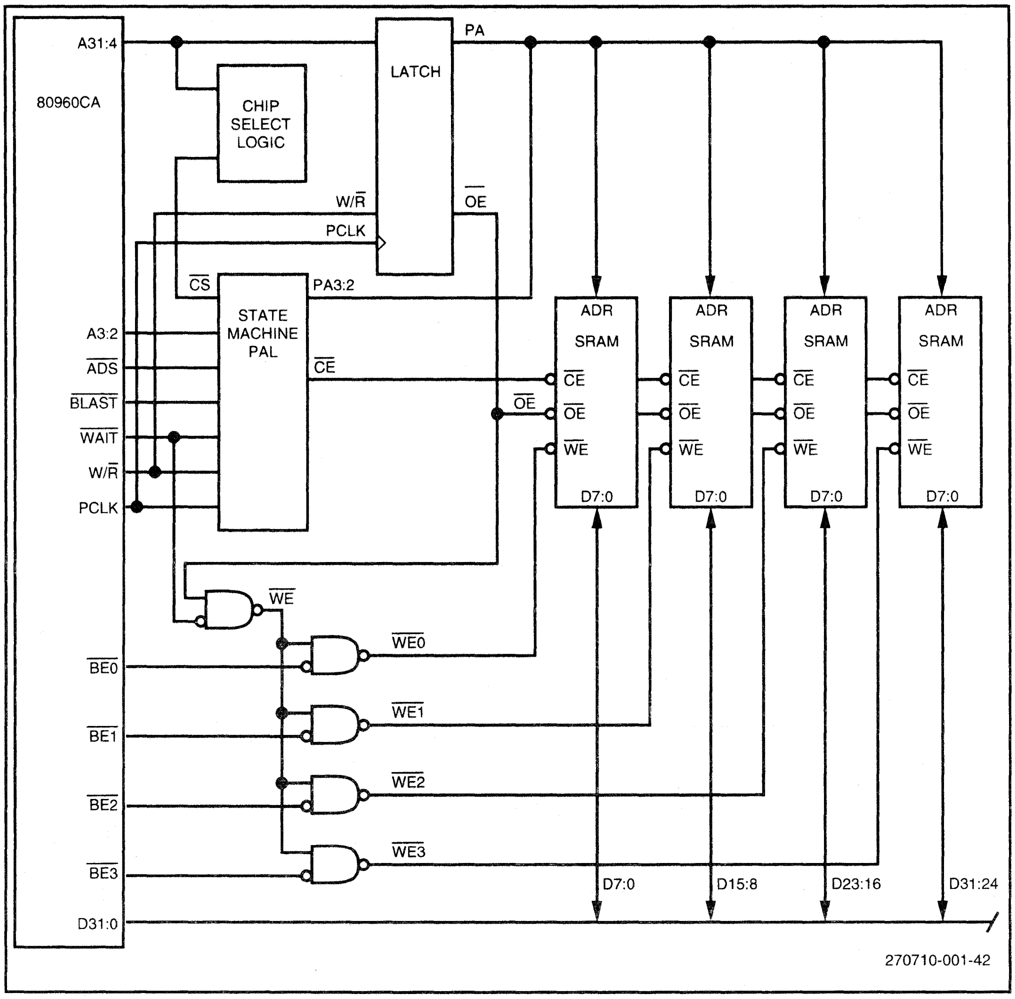


Figure B.7. Pipelined SRAM Interface Block Diagram

Address Latch

During pipelined reads, the i960 CA 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.

State Machine PLD

The state machine PLD contains logic to control $\overline{CE}$ and address signals A3:2. $\overline{CE}$ is controlled by a simple state machine; A3:2 automatically increment during burst accesses. The A3:2 signals are pipelined; they must be latched for read accesses. Write accesses are not pipelined; therefore it is necessary for burst writes to latch A3:2 on reads and pass A3:2 through. The A3:2 generation is implemented as a state machine to achieve minimum address delay out of the PLD. PA3:2 (pipelined address 3:2) outputs are also the state bit of the PLD. This ensures that the address delay is only the clock-to-output time for the PLD.

Write Enable Logic

Write enable logic uses the byte enable signals ($\overline{\text{BE3:0}}$), the $\overline{\text{WAIT}}$ signal and a latched version of the W/$\overline{\text{R}}$ signal ($\overline{\text{OE}}$). Therefore:

$$\overline{\text{WE}} = !(\overline{\text{OE}} \ \& \ \overline{\text{WAIT}} \ \& \ \overline{\text{BE}});$$

or:

$$\overline{\text{WE0}} = !\overline{\text{OE}} \ || \ \overline{\text{WAIT}} \ || \ \overline{\text{BE0}};$$
$$\overline{\text{WE1}} = !\overline{\text{OE}} \ || \ \overline{\text{WAIT}} \ || \ \overline{\text{BE1}};$$
$$\overline{\text{WE2}} = !\overline{\text{OE}} \ || \ \overline{\text{WAIT}} \ || \ \overline{\text{BE2}};$$
$$\overline{\text{WE3}} = !\overline{\text{OE}} \ || \ \overline{\text{WAIT}} \ || \ \overline{\text{BE3}};$$

Waveforms

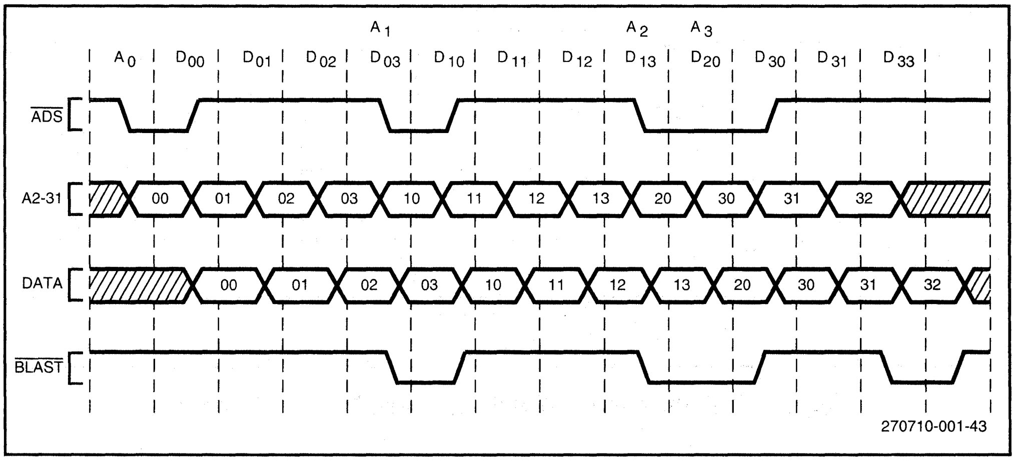


Figure B.8. Pipelined Read Waveform

$\overline{\text{DEN}}$ remains asserted as long as consecutive pipelined read accesses continue. $\overline{\text{DEN}}$ and DT/$\overline{\text{R}}$ are related to the data, not the address; therefore, $\overline{\text{DEN}}$ and DT/$\overline{\text{R}}$ are not pipelined and retain the same timing for pipelined and non-pipelined reads.

In the pipelined read mode, a series of non-burst accesses results in $\overline{\text{ADS}}$ remaining asserted for several clock cycles. Similarly, $\overline{\text{BLAST}}$ remains asserted for several clock cycles.

W/$\overline{\text{R}}$ behaves slightly differently for pipelined reads than for non-pipelined reads. W/$\overline{\text{R}}$ is not valid for the last cycle of a pipelined read. This requires that W/$\overline{\text{R}}$ be latched for pipelined reads similar to A31:2. The following signals are pipelined during pipelined read accesses: A31:2, $\overline{\text{BE3:0}}$, $\overline{\text{SUP}}$, $\overline{\text{DMA}}$ and D/$\overline{\text{C}}$. All of these pipelined signals are invalid during the last cycle of a pipelined read.

Address delay time for the pipelined read is a the clock-to-Q time of the address latch (or the PA3:2 generation PLD). Minimizing address delay maximizes access time.

State Machines

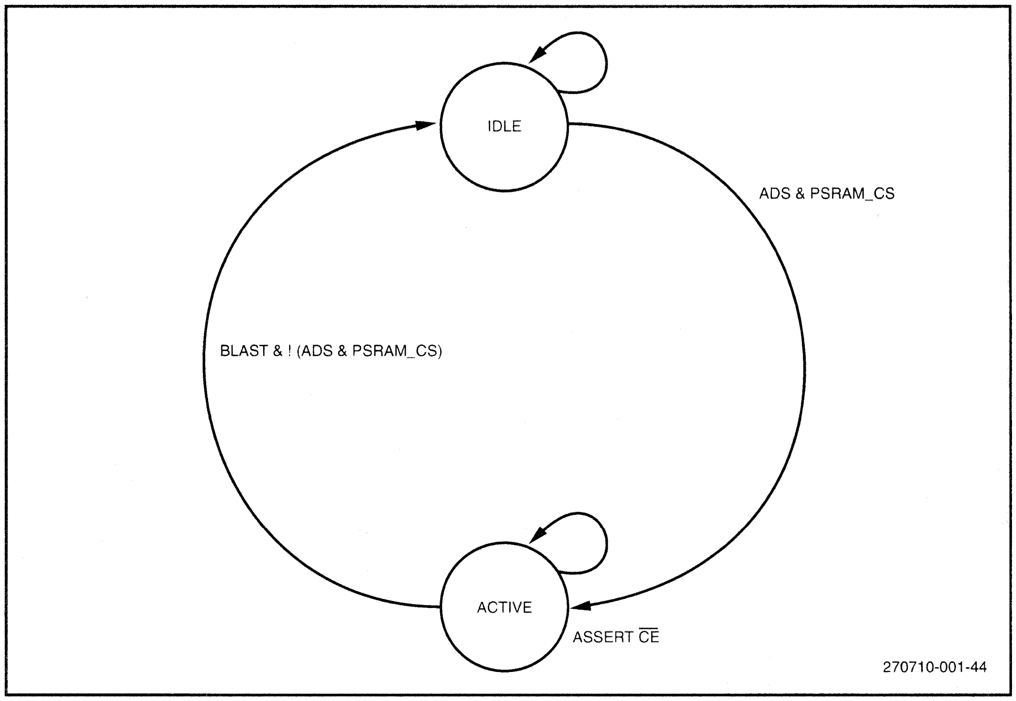


Figure B.9. Pipelined Read Chip Enable State Machine

Chip enable ($\overline{\text{CE}}$) is controlled by a simple state machine. The state machine is normally in the idle state and $\overline{\text{CE}}$ is not asserted. When $\overline{\text{ADS}}$ and $\overline{\text{PSRAM\_CS}}$ are asserted, the $\overline{\text{CE}}$ state machine goes to the active state. $\overline{\text{CE}}$ remains active until $\overline{\text{BLAST}}$ is asserted.

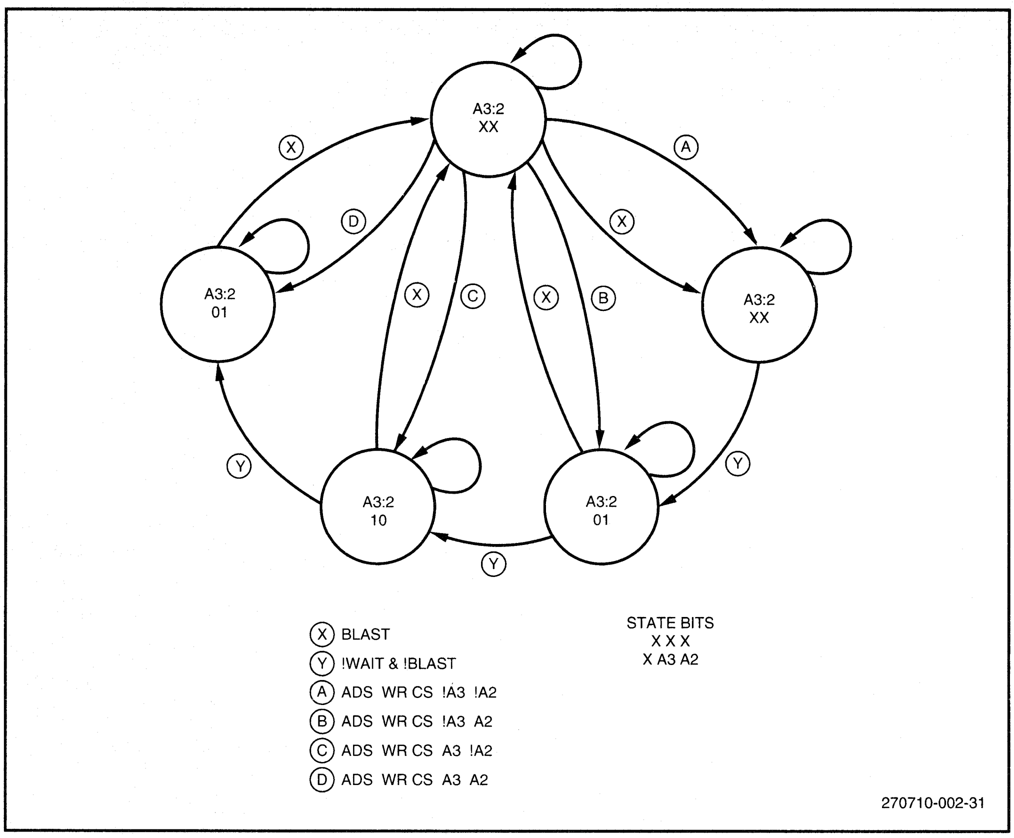


Figure B.10. Pipelined Read PA3:2 State Machine Diagram

The PA3:2 state machine latches the A3:2 address bits on read and generates the low address bit for writes. During read, PA3:2 is a latched version of A3:2. If a write access occurs, the state machine generates the proper PA3:2 addresses.

The pseudo-code description below is provided only to describe the state machine diagram. It is not intended for use directly as PLD equations.

Pseudo-code Key			
#	signal is asserted low	==	equality test
!	logical NOT	:=	clocked assignment
&&	logical AND	=	value assignment
| |	logical OR	X	Don't Care

```
READ_STATE:                    /* PA3:2 := A3:2 */
    IF
        ADS && WR && PSRAM_CS && (A3:2 == 0);
    THEN
        the next state is WRITE_0;
    ELSE IF
        ADS && WR && PSRAM_CS && (A3:2 == 1);
    THEN
        the next state is WRITE_1;
    ELSE IF
        ADS && WR && PSRAM_CS && (A3:2 == 2);
    THEN
        the next state is WRITE_2;
    ELSE IF
        ADS && WR && PSRAM_CS && (A3:2 == 3);
    THEN
        the next state is WRITE_3;
    ELSE
        PA3 := A3;
        PA2 := A2;
        the next state is the READ_STATE;

WRITE_0:                       /* A3:2 = 0 */
    IF
        BLAST;
    THEN
        the next state is the READ_STATE;
    ELSE IF
        !WAIT & !BLAST;
    THEN
        the next state is WRITE_1;
    ELSE
        the next state is WRITE_0;

WRITE_1:                       /* A3:2 = 1 */
    IF
        BLAST;
    THEN
        the next state is the READ_STATE;
    ELSE IF
        !WAIT & !BLAST;
    THEN
        the next state is WRITE_2;
    ELSE
        the next state is WRITE_1;

WRITE_2:                       /* A3:2 = 2 */
    IF
        BLAST;
    THEN
        the next state is the READ_STATE;
    ELSE IF
        !WAIT & !BLAST ;
    THEN
```

```
        the next state is WRITE_3;
    ELSE
        the next state is WRITE_2;

WRITE_3:                    /* A3:2 = 3 */
    IF
        BLAST;
    THEN
        the next state is the READ_STATE;
    ELSE
        the next state is WRITE_3;
```

In the READ_STATE, the state machine simply latches A3:2 and outputs them as PA3:2. On a write, the state machine jumps to the appropriate state based on the value of A3:2. When in a write state, the state machine will advance to the next write state if $\overline{\text{WAIT}}$ and $\overline{\text{BLAST}}$ are not asserted. The state machine can advance from any write state to the READ_STATE.

Tradeoffs and Alternatives

The example described above demonstrates a burst pipelined read SRAM memory 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. The design is easily expanded to accommodate multiple SRAM banks.

INTERFACING DYNAMIC RAM WITH THE i960™ CA PROCESSOR

This section provides an overview of DRAM and DRAM access modes and describes an i960 CA processor-specific DRAM interface. Two specific design examples are also included: one design uses the integrated DMA unit to refresh the DRAM, the other example uses the $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ method of refresh. Both designs illustrate the advantage of the i960 CA 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 CA processor. DRAM systems can be designed in many ways — there are memory access options, memory system configuration options and refresh mode options.

DRAM OVERVIEW

DRAMs offer high data density, fast access times and low cost per bit. DRAMs are 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 CA processor's burst mode bus is well suited to the high speed multiple column access modes found in DRAM. Nibble, fast page and static column modes of DRAM can easily be exploited to improve i960 CA processor memory system performance.

All DRAMs have a multiplexed address bus, a write enable input ($\overline{\text{WE}}$) and two address strobes: row address strobe ($\overline{\text{RAS}}$) and column address strobe ($\overline{\text{CAS}}$). Some DRAMs also have an output enable input ($\overline{\text{OE}}$). DRAMs are accessed by placing a valid row address on the address input pins and asserting $\overline{\text{RAS}}$; then the column address is driven onto the DRAM address pins and $\overline{\text{CAS}}$ is asserted. Write enable ($\overline{\text{WE}}$) input on the DRAM determines whether the access is a read or write. Output enable input ($\overline{\text{OE}}$), found on some DRAMs, controls the DRAM output buffers and can be useful for multibanked and interleaved designs.

DRAM Access Modes

Nibble mode DRAM allows up to four consecutive columns within a selected row to be read or written at a high data rate. A read or write cycle starts by asserting $\overline{\text{RAS}}$. Strobing $\overline{\text{CAS}}$ accesses the consecutive column data. Input address is ignored after the first column access.

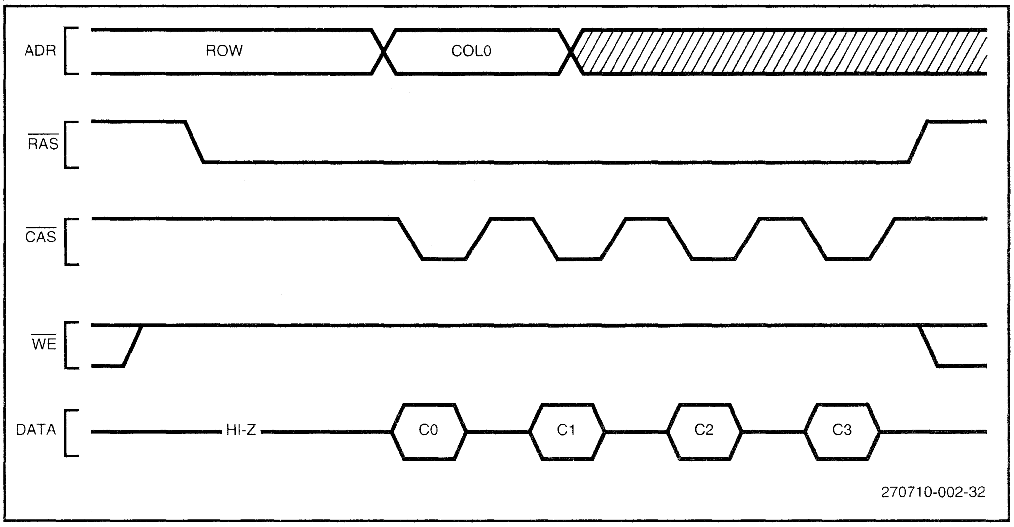


Figure B.11. Nibble Mode Read

Fast page mode DRAM is similar to nibble mode DRAM, except fast page mode 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 $\overline{\text{RAS}}$. Strobing $\overline{\text{CAS}}$ accesses the selected column data. During reads, the $\overline{\text{CAS}}$ 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.

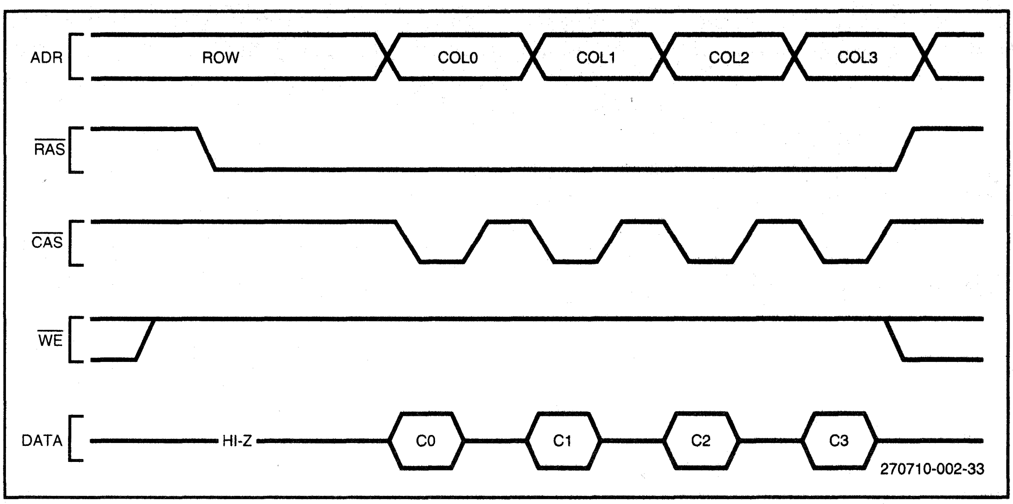


Figure B.12. Fast Page Mode DRAM Read

Static column mode DRAM write accesses are similar to fast page mode writes. Static column read cycles start by asserting $\overline{RAS}$. Accesses to any column within the selected row may be treated as static RAM, using $\overline{CAS}$ as an output enable. The fastest DRAM read accesses are achieved with static column DRAM. The i960 CA processor's four word burst bus can easily take advantage of the faster column access times provided by nibble mode, fast page mode or static column mode DRAM.

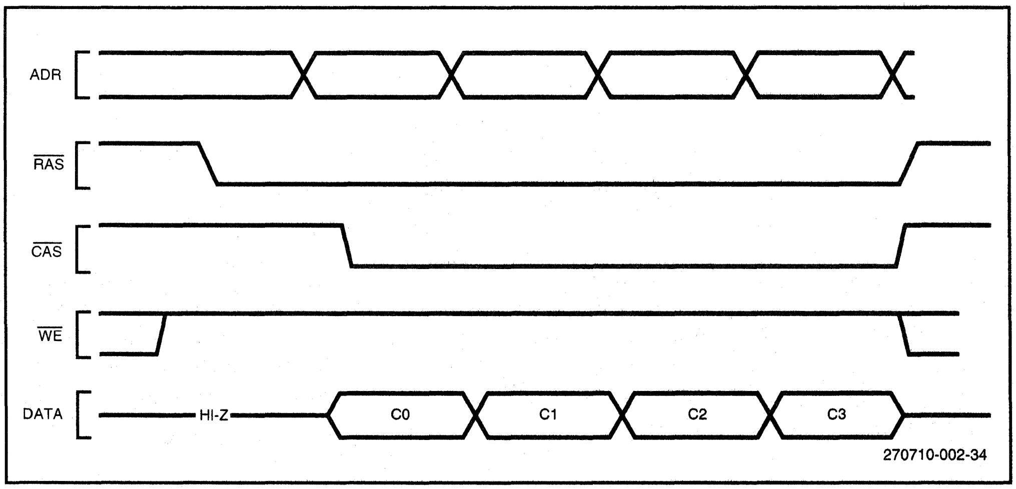


Figure B.13. Static Column Mode DRAM Read

DRAM Refresh Modes

All DRAMs require periodic refresh to retain data. DRAMs may be refreshed in one of two ways: $\overline{\text{RAS}}$-only refresh or $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ refresh. $\overline{\text{RAS}}$-only refresh is realized by asserting a row address on the address pins and asserting $\overline{\text{RAS}}$. $\overline{\text{CAS}}$ is not asserted. A single, $\overline{\text{RAS}}$-only refresh cycle refreshes all columns within the selected row. $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ refreshes do not require an address to be generated; the DRAM generates the row address with an internal counter.

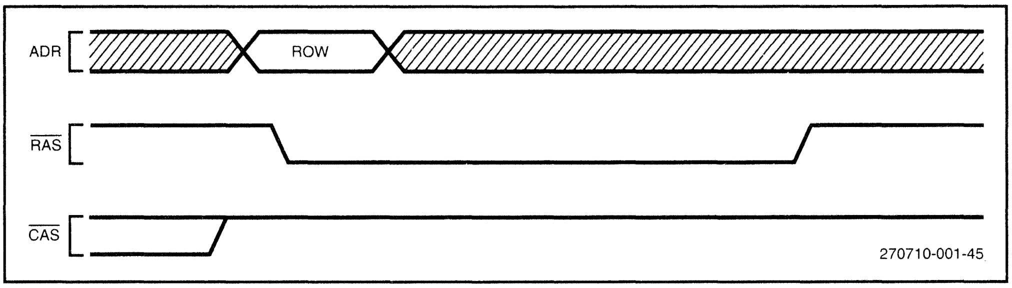


Figure B.14. $\overline{\text{RAS}}$ only DRAM Refresh

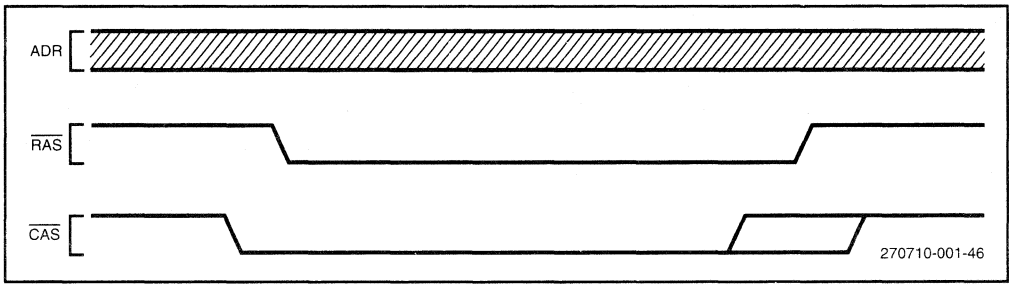


Figure B.15. $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ DRAM Refresh

DRAMs 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 the possible access latency. Burst refreshing may lock the processor out of the DRAM for a longer period of time; it 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.

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 B.16 illustrates this arrangement. Availability of DRAM modules with standard pinouts makes this an attractive way to ensure future memory expansion.

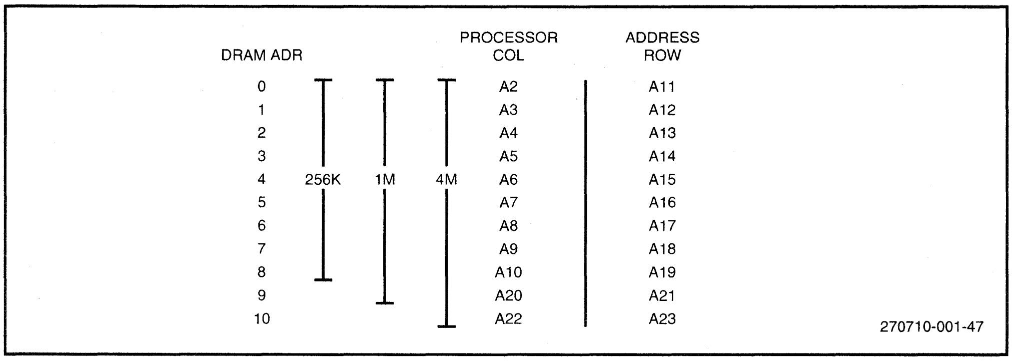


Figure B.16. Address Multiplexer Inputs

Series Damping Resistors

Series-damping resistors are recommended on all DRAM control and address inputs. Series-damping resistors prevent overshoot and undershoot on DRAM 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 will be overdamped, extending memory cycle time. If the damping resistor value is too low, overshoot or undershoot will not be sufficiently damped.

System Loading

The i960 CA 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 CA processor or other system components with less drive capability (e.g., high speed SRAM).

$\overline{\text{RAS}}$ and $\overline{\text{CAS}}$ inputs to the DRAM should also be designed with consideration for capacitive load. When many DRAMs are connected to common $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$ signals, the capacitive load can become considerable.

Design Example: Burst DRAM with Distributed $\overline{\text{RAS}}$ Only Refresh Using DMA

The goal of this design is to illustrate a DRAM interface controller that provides good memory performance while maintaining controller independence with respect to memory speed and processor clock frequency. One of the four on-chip integrated DMA channels is used for DRAM refresh. The region table, DMA and the i960 CA processor bus signals are used to develop a transparent DRAM controller that does not require any information about the memory subsystem.

Figure B.17 shows the DRAM system design. The DRAM is configured as a single, byte accessible, 32-bit-wide bank. $\overline{\text{RAS}}$ is common to the entire bank; $\overline{\text{CAS3:0}}$ serve as byte selects within the bank. $\overline{\text{WE}}$ is common to all the DRAM. The byte accessible bank can be built from four 8-bit-wide DRAM modules; eight 4-bit-wide DRAM modules; eight 4-bit-wide DRAM chips; or 32 1-bit-wide DRAM devices.

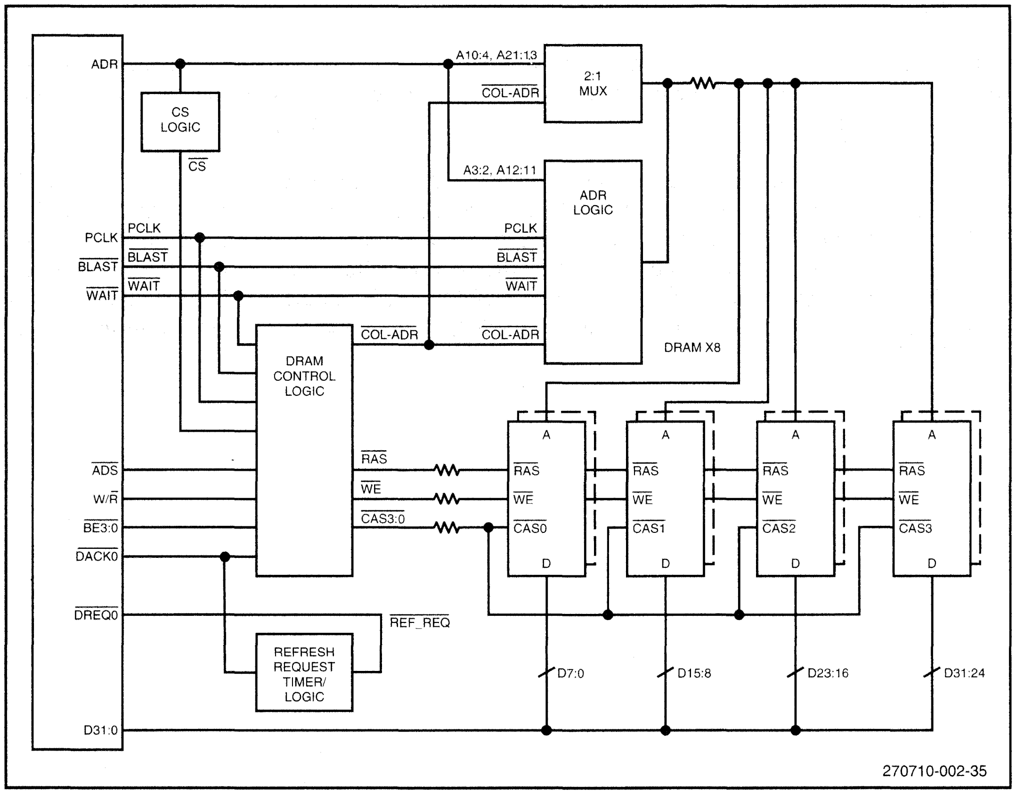


Figure B.17. DRAM System with DMA Refresh

Control logic is divided into three logical blocks: DRAM control logic, DRAM address generation logic and refresh request timer logic. DRAM control logic is the main controller. It controls the address multiplexer and all DRAM control lines during normal and refresh accesses. Address generation logic serves as a multiplexer and an address generator. The refresh request timer logic generates the periodic refresh request to the DMA unit.

DRAM Address Generation

DRAM address generation logic speeds burst accesses for static column mode and 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 i960 CA processor address valid delay time (T_{OV}), plus the worst-case propagation delay through the input address multiplexer.

DRAM address generation logic must control the DRAM address 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.

The following state machine description illustrates the requirements for address generation logic. Signals going into the DRAM address generation logic are: ADR2, ADR3, ADR12, ADR13, $\overline{\text{WAIT}}$ and $\overline{\text{BLAST}}$ from the i960 CA processor and $\overline{\text{COL\_ADR}}$ from the DRAM controller logic. $\overline{\text{COL\_ADR}}$ indicates if the DRAM controller is requesting the row address ($\overline{\text{COL\_ADR}}$ not asserted) or column address ($\overline{\text{COL\_ADR}}$ asserted). Signals output from DRAM address generation logic are the DRAM address two least significant bits, DRAM_ADR2:3.

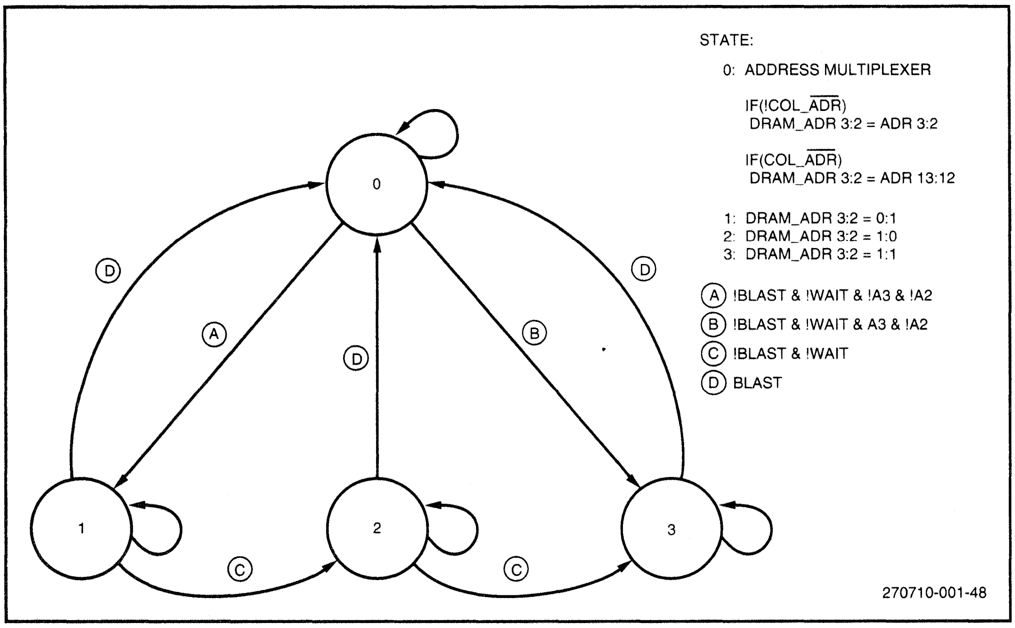


Figure B.18. DRAM Address Generation State Machine

The pseudo-code description below is provided only to describe the state machine diagram. It is not intended for direct use as PLD equations.

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:                          /* Multiplexer Emulation */
        DRAM_ADR2 = (!COL_ADR && A2) || (COL_ADR && A11);
        DRAM_ADR3 = (!COL_ADR && A3) || (COL_ADR && A12);
    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_ADR2 = 1;
        DRAM_ADR3 = 0;
    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_ADR2 = 0;
        DRAM_ADR3 = 1;
    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_ADR0 = 1;
        DRAM_ADR1 = 1;
    IF
        BLAST;
    THEN
        the next state is STATE_0;
    ELSE
        next state is STATE_3
```

DRAM Controller State Machine

Figure B.19 is a state machine that describes DRAM control logic. The state machine shown, or subsets thereof, may be implemented in a large variety of ways depending on the applications requirements. PLD implementations are the easiest and the design may fit into a variety of high speed PLDs.

Signals going into the DRAM control logic are: $\overline{\text{ADS}}$, PCLK, W/$\overline{\text{R}}$, $\overline{\text{BLAST}}$, $\overline{\text{WAIT}}$, $\overline{\text{BE3:0}}$ from the bus controller; $\overline{\text{DACK0}}$, the DMA acknowledge signal; and $\overline{\text{DRAM\_CS}}$, a system generated chip select that indicates a DRAM access. DRAM control logic generates $\overline{\text{RAS}}$, $\overline{\text{CAS3:0}}$, $\overline{\text{WE}}$ and $\overline{\text{COL\_ADR}}$. Control signal for the address multiplexer is $\overline{\text{COL\_ADR}}$.

Controller logic relies on the wait state region table and DMA controller. Programming these on-chip peripherals is described later. DMA acknowledge, $\overline{\text{DACK0}}$, indicates a DRAM refresh cycle. The DRAM $\overline{\text{WE}}$ signal is generated with combinatorial logic ($\overline{\text{WE}}$=!(W/$\overline{\text{R}}$)).

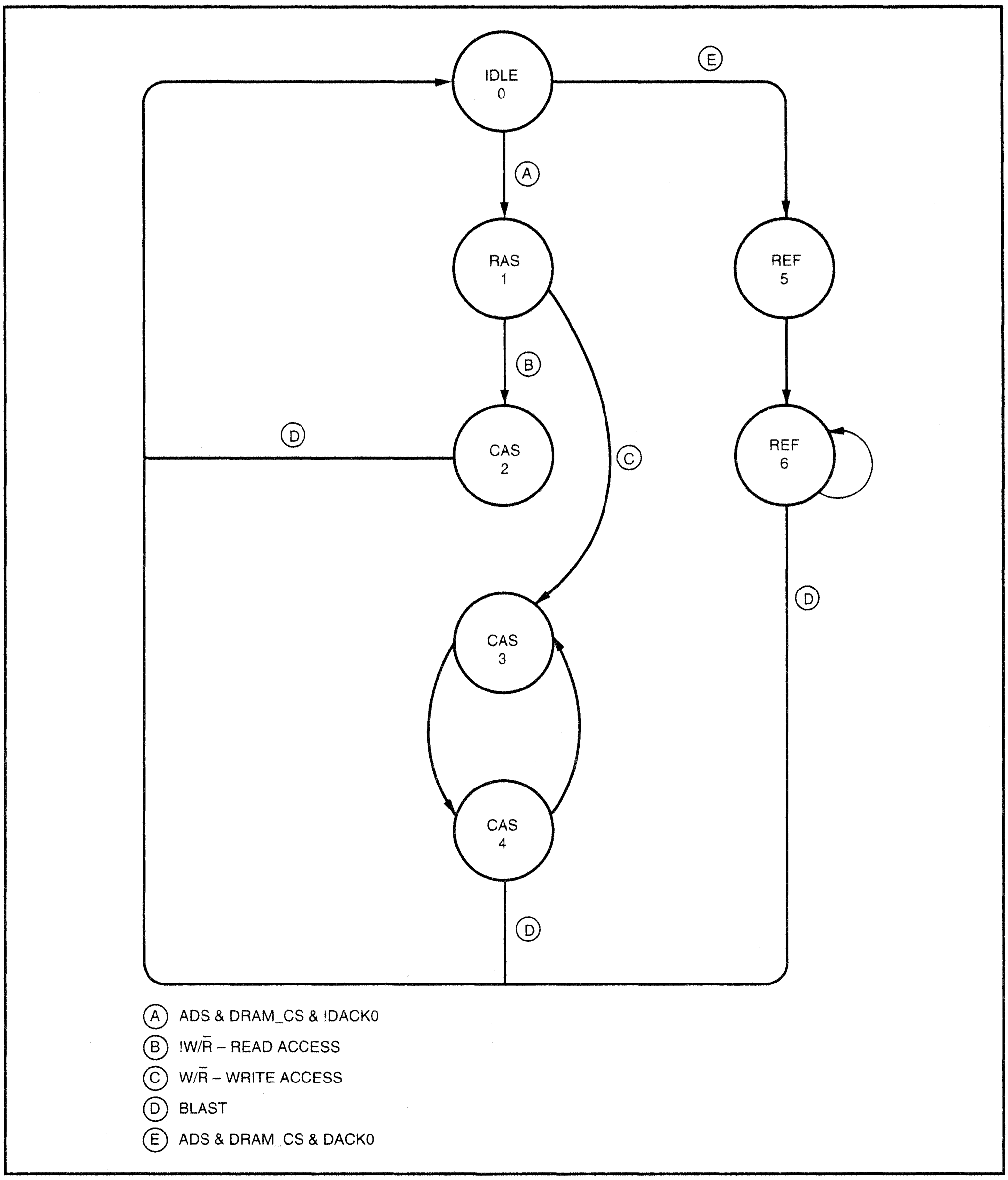


Figure B.19. DRAM Controller State Machine

```
STATE_0:                        /* Idle */
    RAS                         is not asserted;
    CAS3:0                      is not asserted;
    COL_ADR                     is not asserted;

    IF                          /* memory access */
        ADS && DRAM_CS && !DACK0;
    THEN
        the next state is STATE_1;
    ELSE IF                     /* refresh access */
        ADS && DRAM_CS && DACK0;
    THEN
        the next state is STATE_5;
    ELSE
        the next state is STATE_0;
STATE_1:                        /* Assert $\overline{\text{RAS}}$ */
    RAS                         is asserted;
    CAS3:0                      is not asserted;
    COL_ADR                     is not asserted;
    IF
        WRITE;                  /* write */
    THEN
        the next state is STATE_3;
    ELSE                        /* read */
        the next state is STATE_2;
STATE_2:                        /* Static Column Mode Read, Assert $\overline{\text{CAS}}$ */
    RAS                         is asserted;
    CAS3:0                      is asserted;
    COL_ADR                     is asserted;
    IF
        BLAST;
    THEN
        the next state is STATE_0;
    ELSE
        the next state is STATE_2;
STATE_3:                        /* Select Column Address */
    RAS                         is asserted;
    CAS3:0                      is not asserted;
    COL_ADR                     is asserted;
    the next state is STATE_4;
STATE_4:                        /* Assert $\overline{\text{CAS}}$ */
    RAS                         is asserted;
    COL_ADR                     is asserted;
    CAS0 = BE0;
    CAS1 = BE1;
    CAS2 = BE2;
    CAS3 = BE3;
    IF
        WAIT && BLAST;
    THEN
        the next state is STATE_3;
    ELSE IF
        BLAST
    THEN
        the next state is STATE_0;
    ELSE
        the next state is STATE_4;
STATE_5:                        /* REFRESH CYCLE,  RAS  ONLY REFRESH */
    RAS                         is not asserted;
    CAS3:0                      is not asserted;
    COL_ADR                     is asserted;
    the next state is STATE_6;
STATE_6:                        /* REFRESH CYCLE, Assert  RAS   */
    RAS                         is asserted;
    CAS3:0                      is not asserted;
    COL_ADR                     is asserted;
    IF
        BLAST;
    THEN
        the next state is STATE_0;
    ELSE
        the next state is STATE_6;
```

DRAM Refresh Request and Timer Logic

DRAM refresh request and timer logic is responsible for generating DMA requests at an appropriate interval and for removing the DMA request after receiving DMA acknowledge.

Typical DRAMs must be refreshed every 4 ms; refresh cycles must be performed on all 256 rows during this 4 ms interval. If a distributed refresh method is chosen, then a refresh cycle must be performed every 15 µs. The time base can be generated from a counter connected to PCLK, a timer counter chip or any other time base. DMA request and acknowledge signals are shown in Figure B.20.

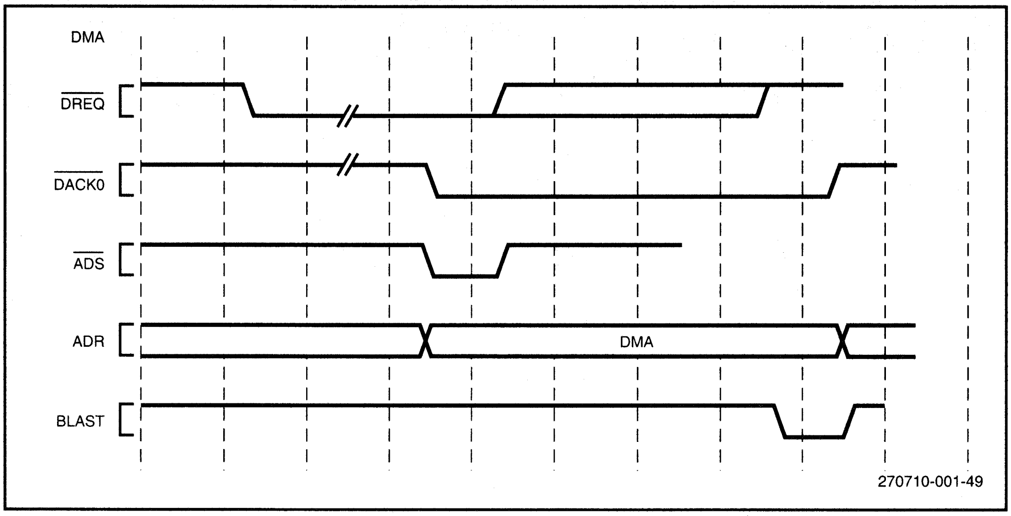


Figure B.20. DMA Request and Acknowledge Signals

DMA Programming for Refresh

DMA should be programmed to perform 32-bit, fly-by, source synchronized demand mode transfers with source chaining. The chaining must be set up to perform an infinite loop of transfers. When all transfers are complete and all rows are refreshed, the cycle begins again. See Figure B.21 for chaining description.

	0xC	0x8	0x4	0x0
ADR: 0XXXXXXXX0	NEXT_PTR	DESTINATION ADR	SOURCE ADR	BYTE COUNT
DRAM_REF_CHAIN	& DRAM_REF_CHAIN	X	DRAM ADR	NUMBER OF ROWS

270710-001-50

Figure B.21. DMA Chaining Description

Memory Ready

The memory ready input to the i960 CA processor ($\overline{\text{READY}}$) indicates the completion of a DRAM read or write cycle. $\overline{\text{READY}}$ must be generated by the DRAM controller and must satisfy setup and hold times specified in the data sheet. If there multiple memory systems are using $\overline{\text{READY}}$, ready signals from these memory systems must be logically ORed together.

Region Table Programming

Region table programming is critical to DRAM operation. N_{RAD} and N_{WAD} wait states must satisfy $\overline{RAS}$, $\overline{CAS}$ 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 $\overline{RAS}$ precharge time. Figures B.22 and B.23 shows typical system waveforms for this design. Note that $\overline{RAS}$ is not asserted until the end of the address cycle; this delay contributes to $\overline{RAS}$ precharge time. In some DRAM designs, it may be possible to remove $\overline{RAS}$ before access is complete. This is especially true for static column reads and multiple world access. If $\overline{RAS}$ can be removed early in the access, $\overline{RAS}$ precharge can occur during the access.

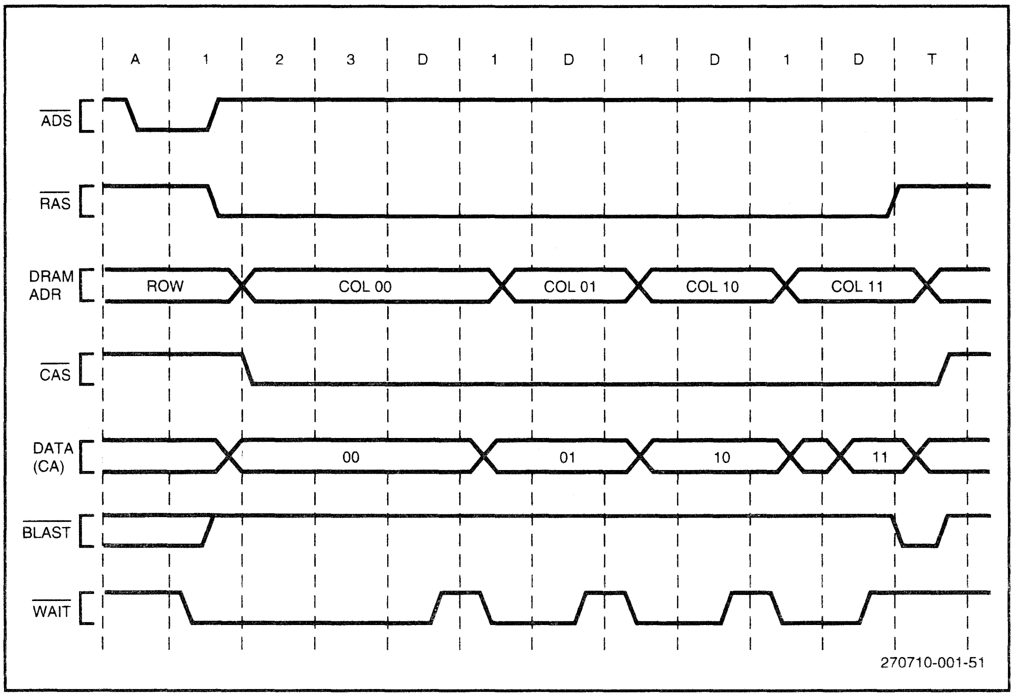


Figure B.22. DRAM System Read Waveform

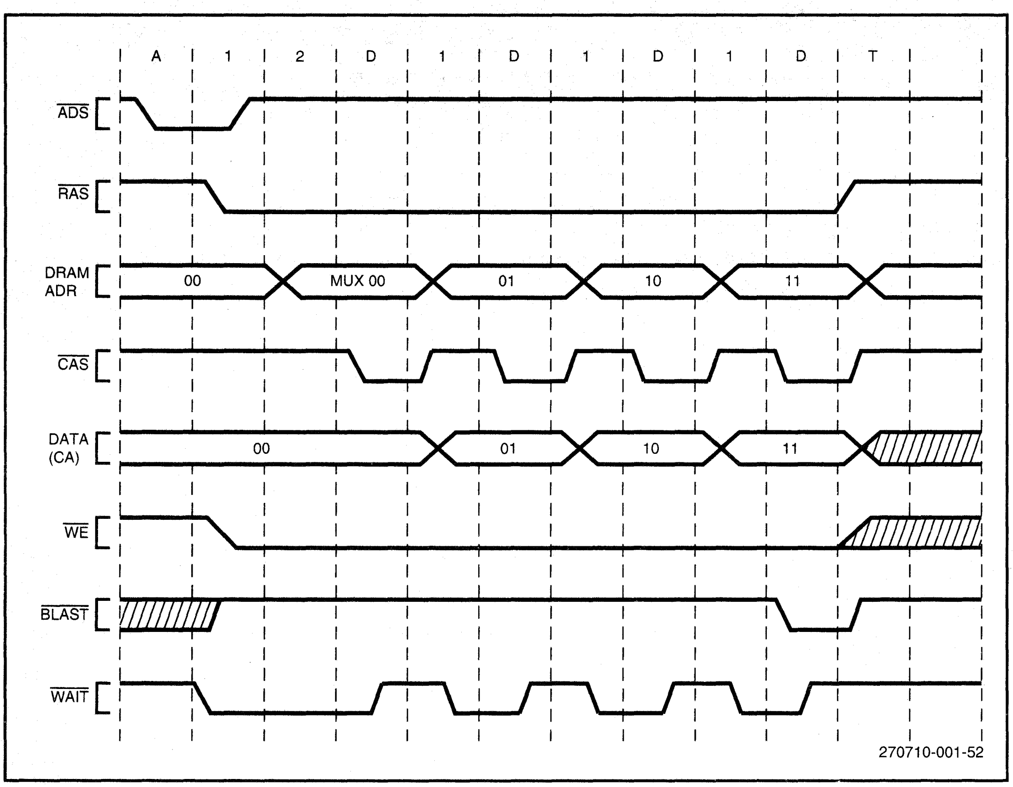


Figure B.23. DRAM System Write Waveform

Design Example: Burst DRAM with Distributed CAS-Before-RAS Refresh using READY Control

This example illustrates a DRAM system design that uses $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ refresh and $\overline{\text{READY}}$ control. $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ refresh uses the internal refresh address generation capabilities of modern DRAMs. The design does not use a DMA channel for refresh. $\overline{\text{READY}}$ must be generated by the DRAM controller to indicate that a data transfer is complete. The controller must arbitrate between access requests and refresh requests, control the address multiplexer and $\overline{\text{RAS}}$ precharge time. The internal wait state generator is not used. DRAM controller must be designed with information about processor and DRAM speed.

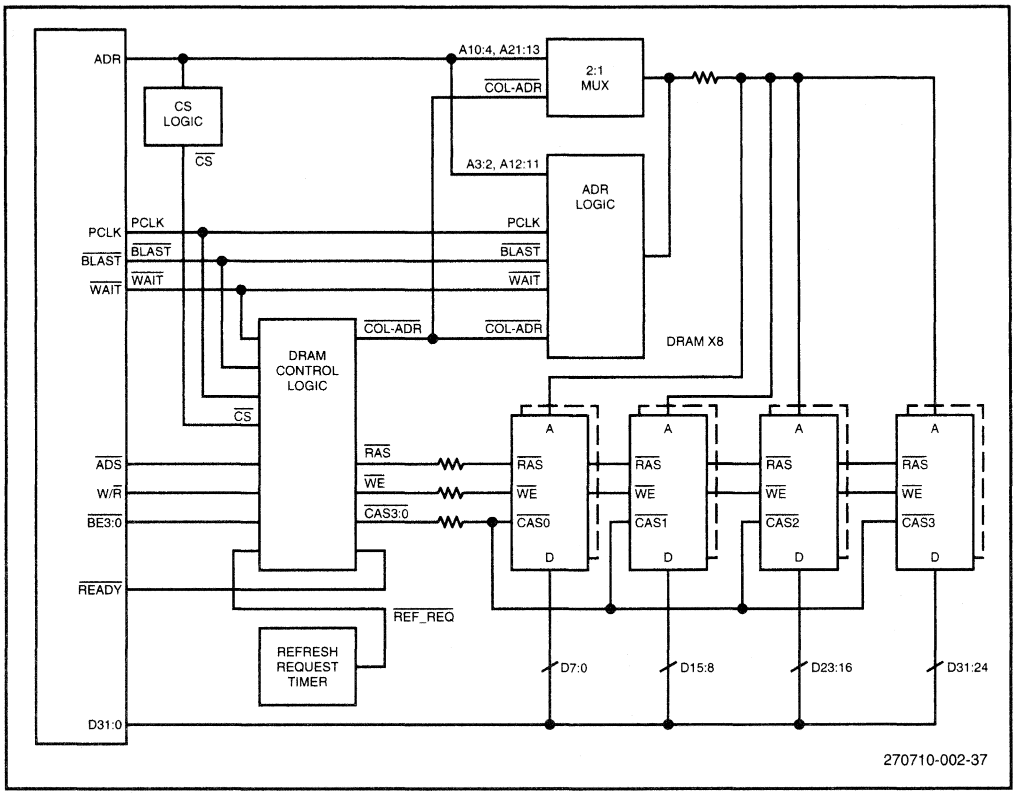


Figure B.24. Block Diagram

The memory system block diagram (Figure B.24) is similar to the schematic for the previous example, except for the absence of the DMA controller connection. The refresh timer indicates it is time to refresh the DRAM.

DRAM Controller State Machine

The state machine in Figure B.25 is more complicated that the state machine in the previous example. This is because the controller works without the help of the internal wait-state generator. There are two advantages of this design over the previous example: a DMA channel is not used and the refresh cycle does not require the processor bus. Not using a DMA channel for DRAM refresh makes the DMA channel available for other applications within the system.

$\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ 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 static column mode DRAM with a 33 MHz clock (PCLK). This DRAM controller does not require the internal wait state generator; as a result, all wait state parameters can be programmed to 0.

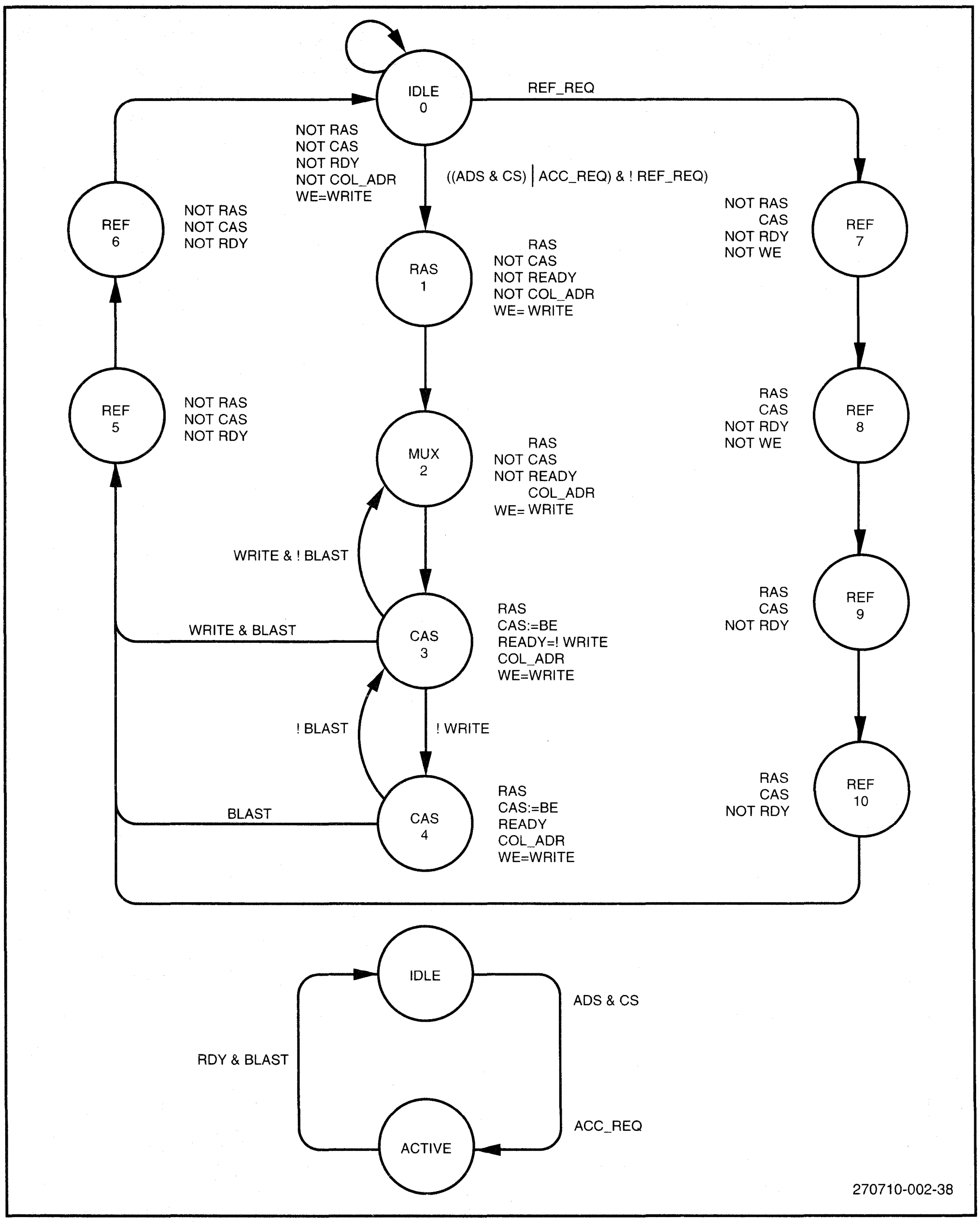


Figure B.25. DRAM State Machine

The refresh request timer generates the refresh request signal ($\overline{\text{REF\_REQ}}$), 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 $\overline{\text{ADS}}$ and $\overline{\text{DRAM\_CS}}$ 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 $\overline{\text{RAS}}$ 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.

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:                          /* 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;
    the next state is STATE_3;
STATE_3:                          /* Assert CAS , write is ready, read is not */
    RAS                           is asserted;
    CAS3:0                        = BE3:0;
    COL_ADR                       is asserted;
    READY                         =!W/R;
    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*/
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                             = 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*/
```

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. For example, 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 B.26 shows the access overlap for a burst access.

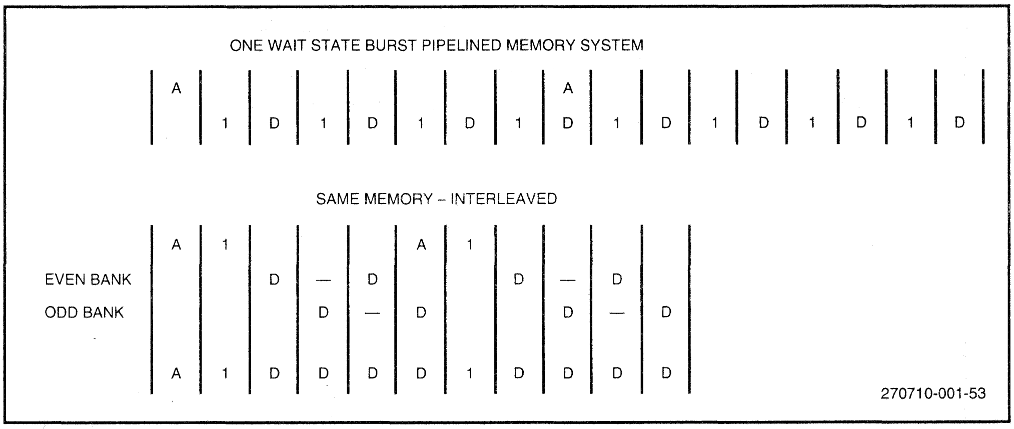

ONE WAIT STATE BURST PIPELINED MEMORY SYSTEM
A 1 D 1 D 1 D 1 D A D 1 D 1 D 1 D 1 D
SAME MEMORY – INTERLEAVED
A 1 A 1
EVEN BANK D — D D — D
ODD BANK D — D D — D
A 1 D D D D 1 D D D D
270710-001-53

Figure B.26. Two-Way Interleaved Read Access Overlap

Figure B.27 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 B.28 shows the read waveform.

The schematic (Figure B.28) illustrates a memory system that interleaves read accesses. Write interleaving requires latching the written data and controlling memory access with the $\overline{\text{READY}}$ 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 CA processor bus controller queues all access; therefore, write interleaving does not significantly benefit most applications.

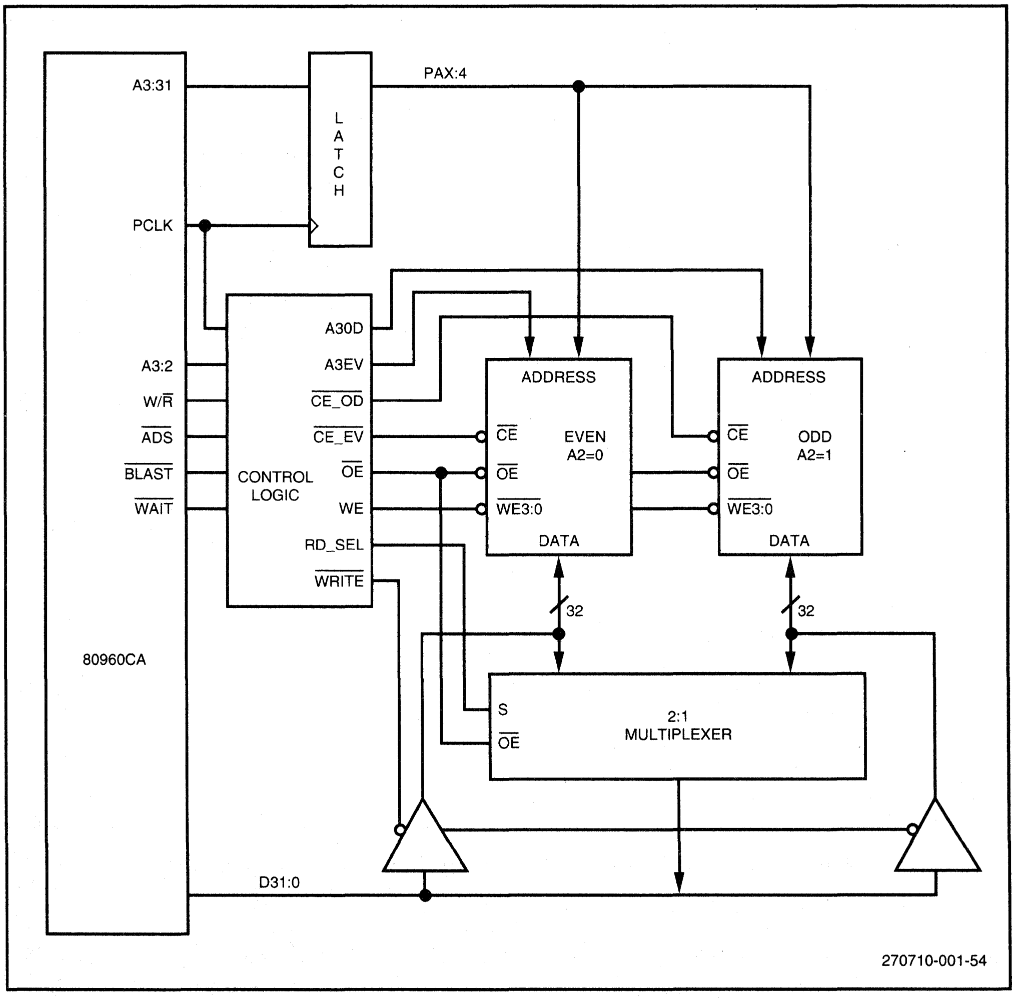


Figure B.27. Two-Way Interleaved Memory System

Memory interleaving can be applied to SRAM, DRAM and even EPROM memory systems. Interleaved SRAM and EPROM memory systems overlap access times for consecutive accesses to improve memory system performance. The i960 CA processor 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 memory systems that require N_{XDA} states or $\overline{\text{READY}}$ control. Interleaved DRAM memory systems can overlap the memory access time and $\overline{\text{RAS}}$ precharge time of consecutive accesses.

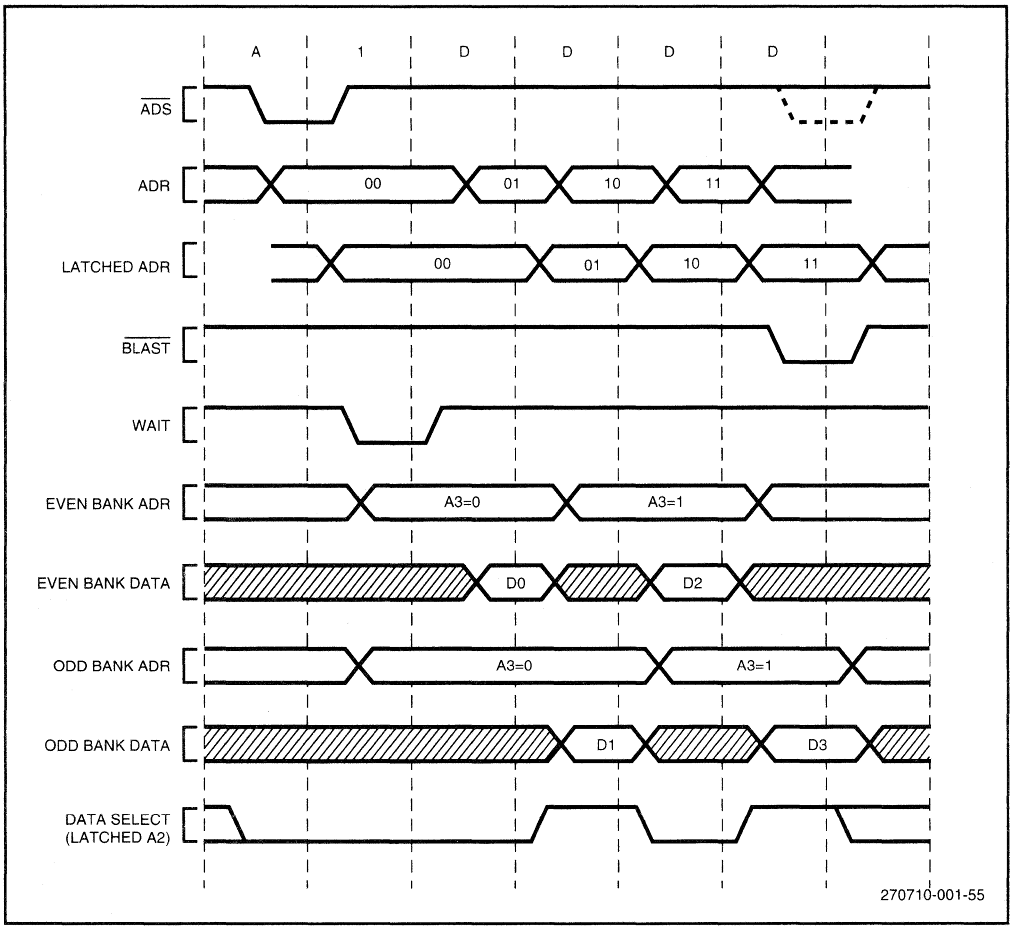


Figure B.28. Two-Way Interleaved Read Waveforms

INTERFACING TO SLOW PERIPHERALS USING THE INTERNAL WAIT STATE GENERATOR

This section illustrates how easy it is to interface slow peripherals to the i960 CA processor. This example shows the interface to an Intel 82C54-2 Timer/Counter and an Intel 82510 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.

Implementation

Both the 82C54-2 Timer/Counter and 82510 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 82C54-2 and 82510 are memory mapped into a memory region programmed for non-burst, non-pipelined reads and an 8-bit data bus.

The $\overline{\text{RD}}$ 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.

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 ($\overline{\text{ADS}}$).

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. A3 address signal determines which peripheral is enabled.

The data transceiver is enabled by the PLD. The transceiver is activated when both the $\overline{\text{CS}}$ and $\overline{\text{DEN}}$ signals are asserted. The equation is:

$$\overline{\text{DATA\_8\_EN}} = \overline{\text{CS}} \mid\mid \overline{\text{DEN}};$$

Transceiver direction control is connected directly to the DT/$\overline{\text{R}}$ signal of the i960 CA device. Data transceiver usage is optional; it is used here to reduce capacitive loading on the data bus. The i960 CA 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.

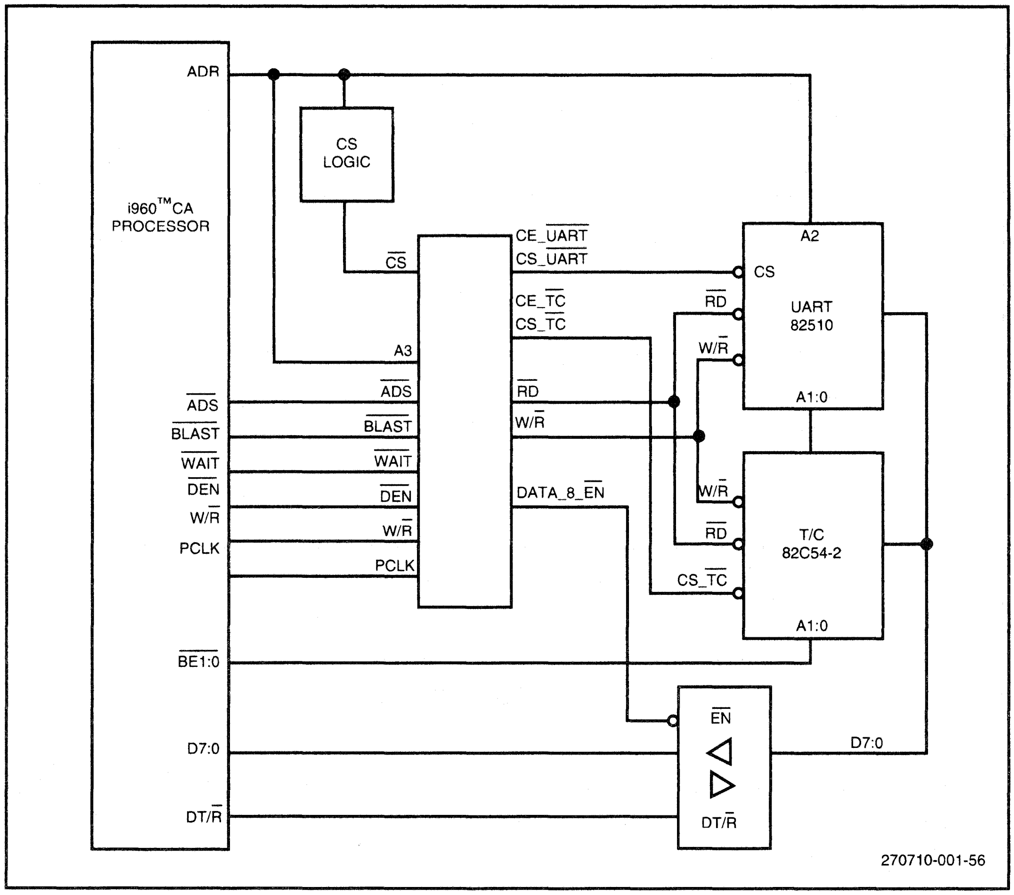


Figure B.29. 8-bit Interface Schematic

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.

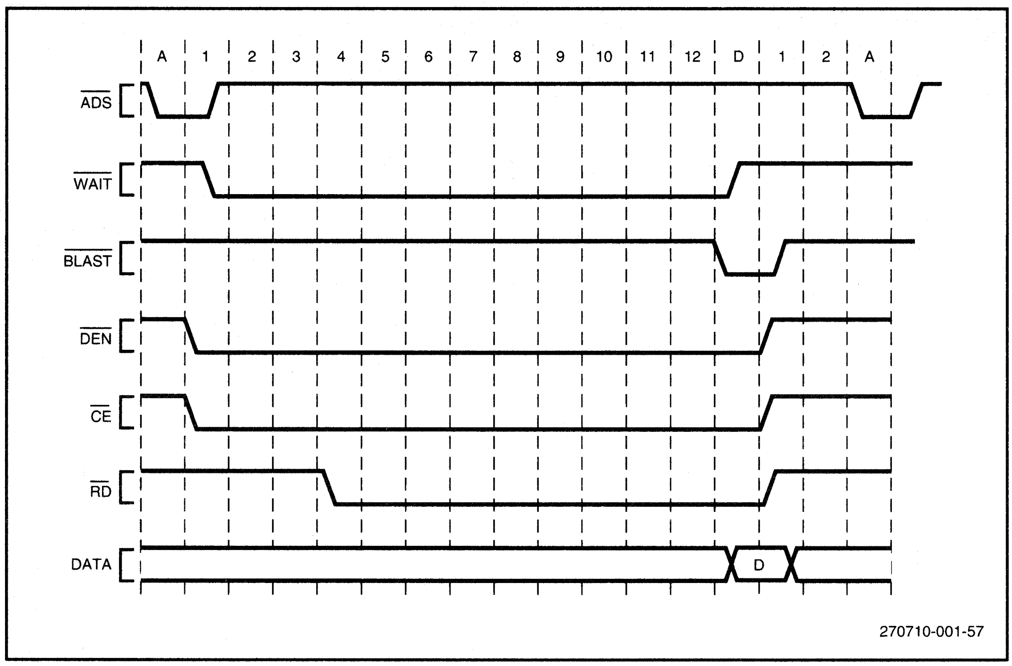


Figure B.30. Read Waveforms

Data lines are not driven during N_{XDA} wait states. This requires gating the W/$\overline{R}$ signal with the $\overline{WAIT}$ signal, so that W/$\overline{R}$ goes high while the data is still asserted. There is a relative timing for output data hold after $\overline{WAIT}$ goes high. The data hold requirement of the peripheral and the delay time to gate the write signal with $\overline{WAIT}$ 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/$\overline{R}$ output of the PLD is a gated version of the $\overline{WAIT}$ signal. This guarantees that the peripheral's write data hold time is satisfied.

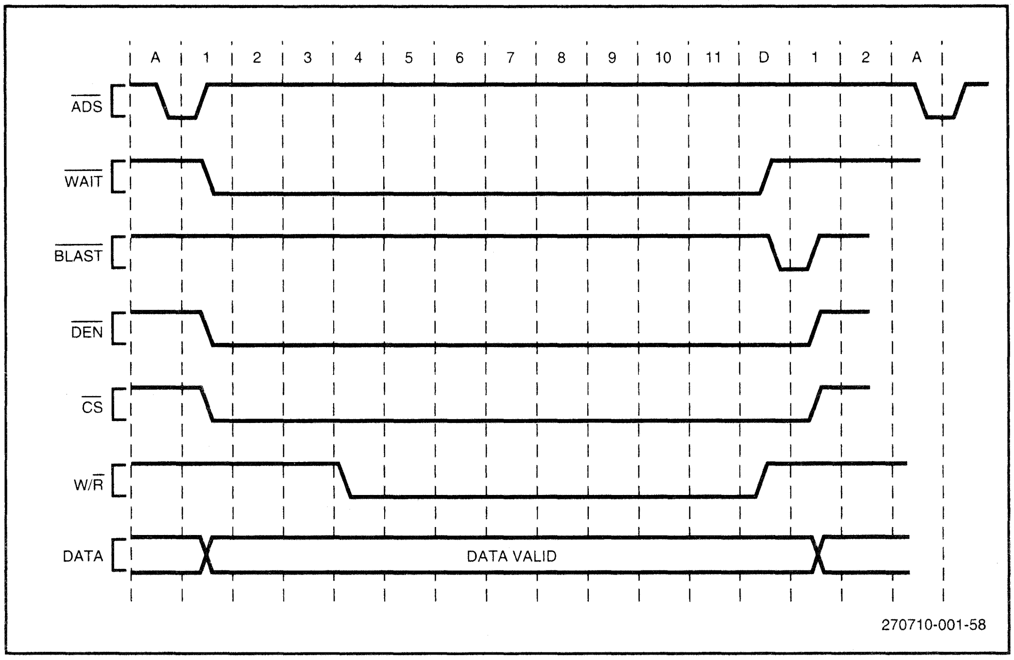


Figure B.31. Write Waveforms

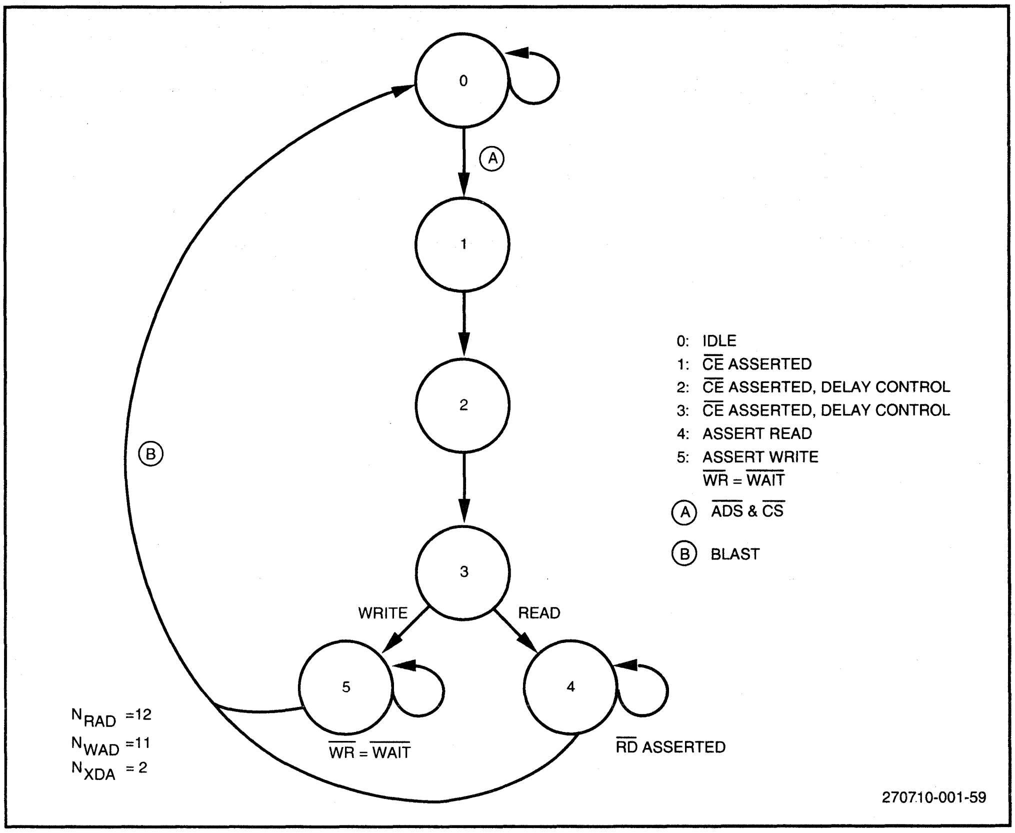

0
1
2
3
4
5
Ⓐ
Ⓑ
WRITE
READ
$\overline{WR} = \overline{WAIT}$
$\overline{RD}$ ASSERTED
0: IDLE
1: $\overline{CE}$ ASSERTED
2: $\overline{CE}$ ASSERTED, DELAY CONTROL
3: $\overline{CE}$ ASSERTED, DELAY CONTROL
4: ASSERT READ
5: ASSERT WRITE
$\overline{WR} = \overline{WAIT}$
Ⓐ $\overline{ADS}$ & $\overline{CS}$
Ⓑ BLAST
N_{RAD} =12
N_{WAD} =11
N_{XDA} =2
270710-001-59

Figure B.32. State Machine Diagram

The pseudo-code description below is provided only to describe the state machine diagram. It is not intended for direct use as PLD equations.

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:                        /*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;
```

INTERFACING TO THE 27960CA BURST EPROM

The 27960CA Burst EPROM offers an integrated, high-performance, pipelined burst interface to the i960 CA processor. The Burst EPROM provides a synchronous interface to the i960 CA processor that requires no external logic. These EPROMs offer higher performance read memory systems than high speed DRAMs.

Overview of the 27960CA Burst EPROM

The 27960CA Burst EPROM is a 128K x 8, high-performance CMOS EPROM with synchronous pipelined burst interface. The 27960CA requires no support circuity and provides a synchronous burst interface to the i960 CA processor's bus. The Burst EPROM can operate in the processor's pipelined or non-pipelined access modes. The highest performance is realized in the pipelined read mode. Internally, the 27960CA Burst EPROM is organized in blocks of four bytes which are sequentially accessed.

A burst access begins by latching the address in the EPROM on the rising edge of PCLK when $\overline{\text{ADS}}$ is asserted. After one or two wait states, depending on the version of the 27960CA Burst EPROM, the first data byte is output. The next three consecutive data bytes can be output without any data-to-data wait states. Burst access is terminated on the rising edge of PCLK when $\overline{\text{BLAST}}$ is asserted. Burst EPROM timing is shown in Figure B.33.

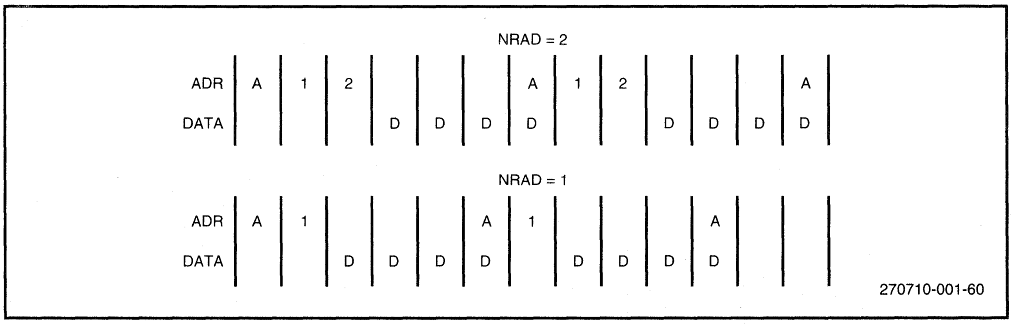

NRAD = 2
ADR | A | 1 | 2 | | | | A | 1 | 2 | | | | A
DATA | | | | D | D | D | D | | | D | D | D | D
NRAD = 1
ADR | A | 1 | | | | A | 1 | | | | A | |
DATA | | | D | D | D | D | | D | D | D | D | |
270710-001-60

Figure B.33. Performance of Burst EPROM Pipelined Read

High performance outputs provide zero wait state, data-to-data burst access. Extra power and ground pins dedicated to the output circuity reduce the effect of fast output switching.

The 27960CA Burst EPROM is a byte-wide device. Systems can be designed with the 27960CA Burst EPROM in 8-, 16- or 32-bit data widths by connecting them to the proper i960 CA processor data pins. The signal definitions below provide an operation description of the 27960CA Burst EPROM. (For programming information, see the 27960CA Burst EPROM data sheet.) 27960CA Burst EPROM signal definitions are:

CLK	Clock (input). EPROM clock. Address (A16:0) is latched internally on the rising edge of CLK. Data (D7:0) is output with respect to CLK. $\overline{\text{ADS}}$, $\overline{\text{CS}}$ and $\overline{\text{BLAST}}$ are all sampled on the rising edge of CLK. This signal may be connected directly to the i960 CA processor PCLK signal.
A16:0	17-bit address bus (input). During a burst operation, A16:2 provides the base address pointing to a block of four consecutive bytes. A1:0 selects the first byte of the burst access. The 27960CA Burst EPROM latches valid addresses in the first clock cycle. An internal address generator increments addresses for subsequent burst bytes.
D7:0	8-bit data bus (output). Data bus drives are enabled when $\overline{\text{CS}}$ and $\overline{\text{ADS}}$ are asserted during the rising edge of CLK. Data bus drivers are disabled when $\overline{\text{BLAST}}$ is asserted and $\overline{\text{ADS}}$ is not asserted on the rising edge of CLK.
$\overline{\text{ADS}}$	Address strobe (input). Indicates the start of a new bus access. It is asserted (low) in the first clock cycle of a bus access. This signal may be connected directly to the i960 CA processor $\overline{\text{ADS}}$ signal.
$\overline{\text{CS}}$	Chip select (input). Master device enable. When asserted (low), data can be read from the device. $\overline{\text{CS}}$ enables the state machine and I/O circuitry. A memory access begins on the first rising edge of CLK in which $\overline{\text{ADS}}$ and $\overline{\text{CS}}$ are asserted. A burst cycle does not terminate if $\overline{\text{CS}}$ goes high.
$\overline{\text{BLAST}}$	Burst last (input). Terminates the current burst access. This signal may be connected directly to the i960 CA processor $\overline{\text{BLAST}}$ signal.
$\overline{\text{RESET}}$	Asynchronous reset (input). Resets the EPROM, disables the data outputs. Reset will abort an active access.

Figure B.36 shows the connections to the 27960CA Burst EPROM.

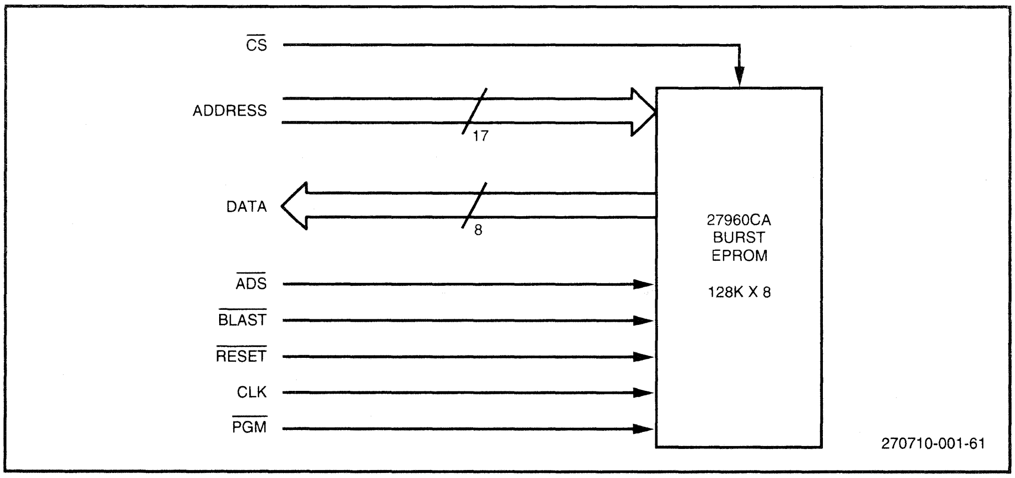


Figure B.34. The 27960CA Burst EPROM

Interfacing to the i960™ CA Microprocessor

The following example demonstrates a 32-bit-wide burst access EPROM interface to the i960 CA processor. The 27960CA Burst EPROM operates at one or two N_{RAD} wait states between the address and the first byte of the burst (depending on the version of the 27960CA Burst EPROM). There are no wait states between sequential data during a burst. Figure B.35 shows a non-buffered, 128K x 32 Burst EPROM system. Chip select logic is the only external logic required for this interface.

Higher order address lines are decoded to generate $\overline{CS}$. Qualification of $\overline{CS}$ with other signals is done by the 27960CA Burst EPROM. Chip select logic can be implemented with standard asynchronous decoders or a PLD. The pipelined read waveform for the Burst EPROM system is shown is Figure B.36.

The wait state configuration must be programmed into the i960 CA processor's Memory Region Configuration Table. N_{RAD} wait states must be programmed to one or two, corresponding to the version of the 27960CA Burst EPROM. N_{RDD} wait states must be programmed to 0. N_{XDA} wait states should be programmed to 0.

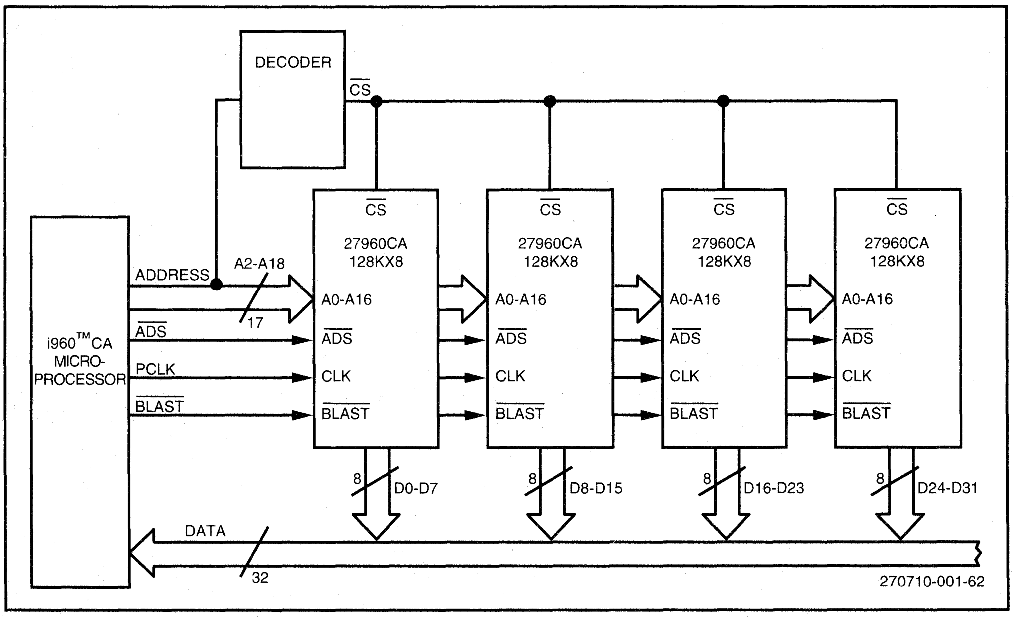


Figure B.35. 128K X 32 Burst EPROM System

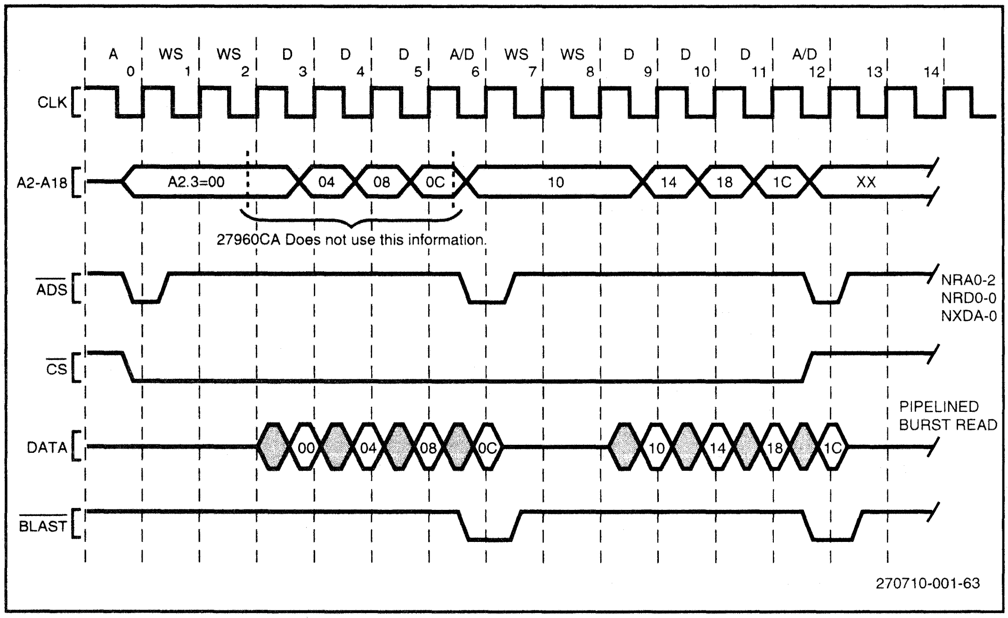


Figure B.36. Burst Pipelined EPROM Read

Booting from the 27960CA Burst EPROM

The i960 CA processor reads four bytes from the Initialization Boot Record (IBR) on initialization. (See *Chapter 14, Initialization and System Requirements.*) The processor's initial bus configuration is encoded in these four bytes. During initialization, before these bytes are read, the memory region configuration table defaults to $N_{RAD} = 31$ and $N_{XDA} = 3$. To facilitate booting from the Burst EPROM, the 27960CA will access normally and then wrap around to the first word (least significant) of the four word burst. This word is held until BLAST is asserted (this is illustrated in Figure B.37). In this way, it is possible to store the IBR in the Burst EPROM.

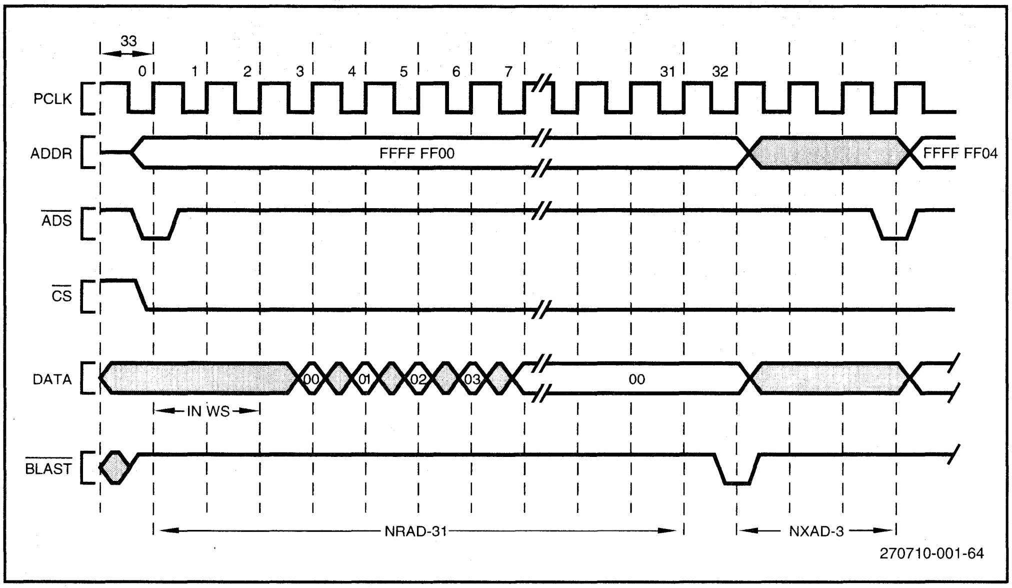


Figure B.37. Booting from the 27960CA Burst EPROM

INTERFACING TO THE 82596CA LOCAL AREA NETWORK COPROCESSOR

The 82596CA LAN coprocessor provides a subset of the i960 CA processor bus interface signals, minimizing bus interface logic. It shares most signals directly with the i960 CA processor. The 82596CA LAN coprocessor's bus cycles (including burst cycles), bus interface timing, bus arbitration method and signal definitions are compatible with the i960 CA processor.

NOTE

In this section, i960 CA microprocessor is generally referred to as "processor" and 82596CA LAN coprocessor is referred to as "coprocessor."

82596CA LAN Coprocessor Overview

The 82956CA coprocessor is a 32-bit multitasking LAN coprocessor which implements the carrier sense, multiple access and collision detect (CSMA/CD) link access protocol (Figure B.38). The coprocessor supports a wide variety of networks. It executes high-level commands and performs command chaining and interprocessor communication via memory shared with the i960 CA processor. This relieves the processor of all time-critical, local network control functions.

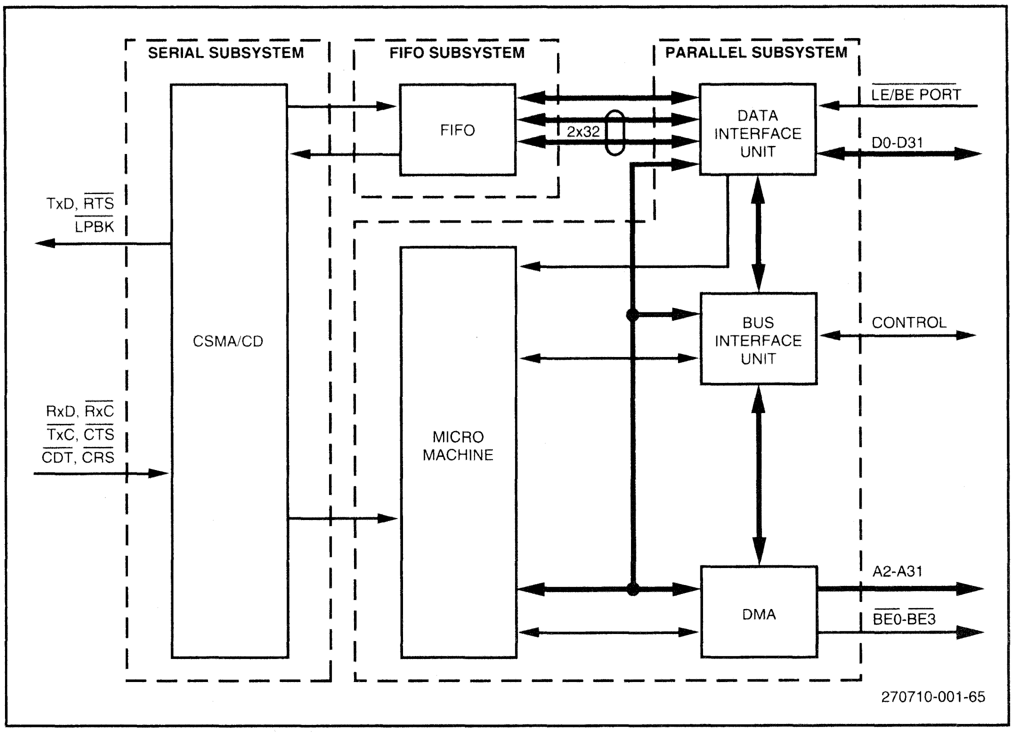


Figure B.38. 82596CA LAN Coprocessor Block Diagram

Coprocessor features include:

- Complete CSMA/CD functions
  - Complete media access control (MAC) functions
  - High level command interface
  - Manchester encoding or NRZ encoding and decoding
  - IEEE 802.3 or HDLC frame delimiting
- Industry-standard network support
  - IEEE 802.3 (Ethernet, Ethernet Twisted Pair, Cheapernet, StarLAN, etc.)
  - IBM PC Network (baseband and broadband)
  - Proprietary CSMA/CD networks up to 20 MBits/sec

- Compatible i960 CA processor interface
  - Optimized bus interface to the i960 CA processor bus
  - Shared i960 CA processor bus signals and memory timing
  - Support for i960 CA processor byte ordering
- Architectural features:
  - On-chip DMA
  - Bus throttle
  - 128-byte receive FIFO, 64-byte transmit FIFO
  - On-chip memory management
  - Network management and diagnostics
  - 82586 software-compatible mode
- Performance features:
  - 9.6 microsecond back-to-back frame transmission and reception
  - 80/105.6 Mbytes/second bus transfer rate (burst transfers) at 25/33 MHz
  - 50/66 Mbyte/second bus transfer rate (non-burst transfers) at 25/33 MHz

Applications

This coprocessor is ideal for interconnect, bridges and high performance embedded communication applications. Its bus interface provides a compatible interface to the i960 CA processor bus, making it very easy to use. Typically, the serial interface is to a physical layer device, such as the Intel 82C501AC Ethernet serial interface chip or the 82521 Twisted Pair Ethernet Serial Super Component.

For burst transfers, the coprocessor's bus occupies only three percent of the total processor-bus bandwidth, under maximum loading conditions for Ethernet. The large FIFOs tolerate long bus latencies – up to 100 μs – which is ideal for systems with multiple bus masters. Programmable bus throttle timers regulate coprocessor's use of the processor bus, allowing the processor bus overhead to be optimized for a given worst-case bus latency. The BREQ signal from the processor can trigger the coprocessor's bus throttle timers when needed or the timers can be controlled by the coprocessor itself.

Processor and Coprocessor Interaction

The coprocessor interacts with the i960 CA processor bus as either a bus master or a slave (port access mode). In normal operation, it is a bus master which moves data between system memory and the coprocessor's control registers or internal FIFOs. The coprocessor can use the same burst cycles, bus hold and bus lock operations as the i960 CA processor.

The coprocessor and processor communicate through shared memory, as shown in Figure B.39. The processor and coprocessor normally use the interrupt (INT/$\overline{\text{INT}}$) and channel attention (CA) signals to initiate communication and use a system control block of memory for command and status storage. INT/$\overline{\text{INT}}$ alerts the processor to a change of contents in the system control block. By asserting CA, the processor causes the coprocessor to examine the system control block contents for the change.

The coprocessor executes its command list from shared memory and simultaneously receives frames from the network and places them in shared memory. The processor manages the shared memory, which contains command chains and bidirectional data chains. The coprocessor executes the command chains. An on-chip DMA controls four channels which allow autonomous transfers of data blocks. Buffers, containing erroneous or collided frames, can be automatically recovered without processor intervention. The processor becomes involved only after a command sequence has finished executing or after a sequence of frames has been received and stored, ready for processing.

In addition to this normal operating mode, the processor can initiate a port access in the coprocessor. This allows the processor to write an alternate system configuration pointer, write an alternate dump command and pointer (used for troubleshooting a no-response problem), perform a software reset or perform a self test.

Bus Interface Signals

The i960 CA processor and 82596CA coprocessor share the bus by floating their respective output and I/O bus signals when bus ownership is not acquired. The following summarizes the input shared bus interface signals between the coprocessor and processor. This interface is also shown in Figure B.39.

Table B.1. Shared i960™ CA Processor and 82596CA Bus Output and I/O Signals

Signal	Definition	Type	Signal state when not bus owner
A31-A2	Address	O	float
$\overline{\text{BE3:0}}$	Byte Enables	O	float
D31-D0	Data Bus	I/O	float
$\overline{\text{LOCK}}$	Bus Lock indicator	O	float
W/$\overline{\text{R}}$	Write/Read indicator	O	float
D/$\overline{\text{C}}$	Data/Control indicator	O*	float

NOTE

* The 82596CA does not have the D/$\overline{\text{C}}$ signal.

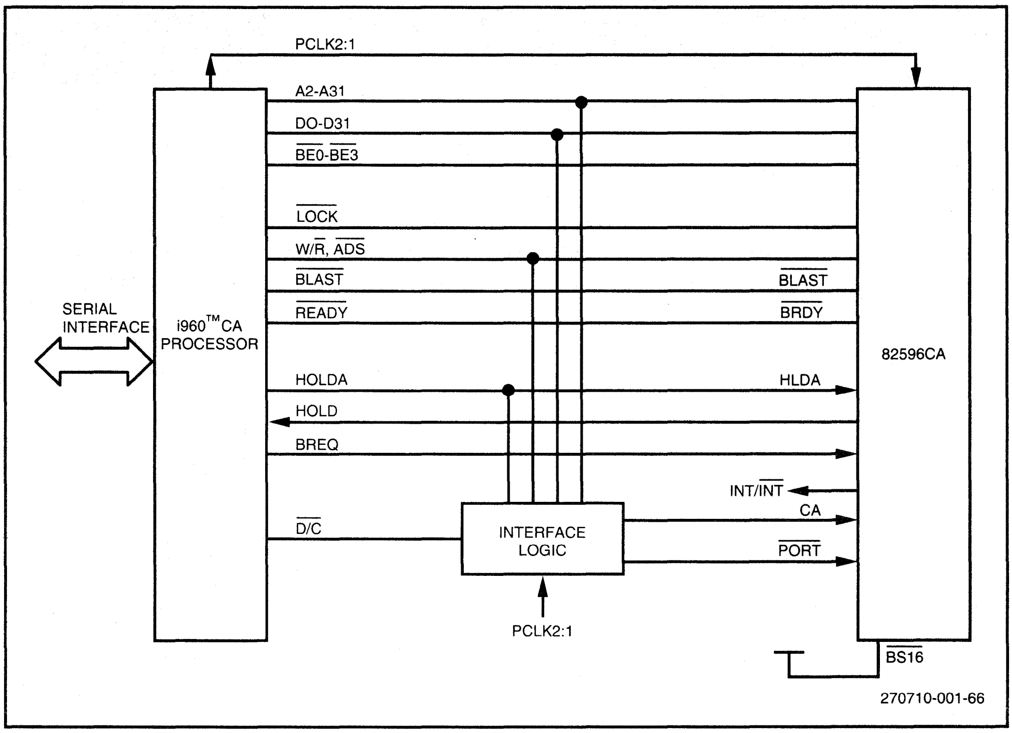


Figure B.39. i960™ CA Processor/82596CA Coprocessor Interface

Table B.2. Shared i960™ CA Processor and 82596CA Bus Input Signals

Signal	Definition	Type
$\overline{\text{BRDY}}$ (82596)/$\overline{\text{READY}}$(i960 CA processor)	Ready	I
$\overline{\text{RDY}}$ (82596)/$\overline{\text{BTERM}}$(i960 CA processor)	Burst terminate	I

Table B.3. Arbitration Signals for i960™ CA Processor/82596CA Interface

Signal	Definition	Type	i960 CA Processor Type	Comments
HOLD	Hold request	I	O	82596CA coprocessor always drives
HLDA (82596 coprocessor) HOLDA (i960 CA processor)	Hold acknowledge	O	I	i960 CA processor always drives
BREQ	Bus Request	O	I	i960 CA processor always drives

Arbitration

Bus arbitration between the i960 CA processor and 82596CA coprocessor is achieved by the hold and hold acknowledge handshake. The coprocessor requests the bus by asserting HOLD to the processor. The processor responds by asserting HOLDA, thus allowing the coprocessor to acquire the bus. The processor's BREQ signal can be used to improve arbitration efficiency. BREQ indicates that an internal cycle is pending. This signal can be tied directly to the coprocessor's BREQ input. When BREQ is asserted, it triggers the coprocessor's bus throttle timers. The bus throttle timers cause the coprocessor to relinquish the bus in a programmable amount of time. This scheme can help improve arbitration efficiency by reducing hold and hold acknowledge handshake delays between the processor and coprocessor.

Interface Logic Requirements

Interface logic between the processor and coprocessor performs the following functions:

- Provides a port that the processor can select, based on an address decode to perform a coprocessor channel attention
- Provides a port that the processor can select, based on an address decoded to perform coprocessor CPU PORT access functions.
- Drives the $D/\overline{C}$ signals when the coprocessor controls the bus; the coprocessor does not have this signal.

82596CA Coprocessor and i960™ CA Processor Interface Considerations

Coprocessor/processor interface provides compatible bus signals and bus operation; however, there are some differences between the two interfaces that should be considered:

- The processor supports read pipelining; the coprocessor does not. Processor read pipelining is programmed through a region table, allowing pipelining for a certain memory region. The processor and coprocessor should share a non-pipelined memory region.
- The coprocessor supports dynamic bus sizing for 32- and 16-bit buses. The processor does not support dynamic bus sizing; it supports bus sizing through a programmable region table. Both the coprocessor and processor have a compatible byte enable encoding scheme for 32-bit buses and should share a 32-bit memory region.
- The processor has a wait state generator built in; the coprocessor does not. The ready signal needs to be properly returned to the coprocessor.
- The processor provides the signals DT/$\overline{\text{R}}$ and $\overline{\text{DEN}}$ and the coprocessor does not. If the external hardware uses these signals, then these signals need to be generated when the coprocessor controls the bus.

Appendix C
Considerations for Writing Portable Code

APPENDIX C
CONSIDERATIONS FOR WRITING PORTABLE CODE

This appendix describes the parts of the i960 CA microprocessor which are implementation dependent. The following information is intended as a guide for writing application code which is directly portable to other implementations of the i960 architecture.

i960™ CORE ARCHITECTURE

The i960 CA component is an implementation of the *i960 core architecture*. All i960 family products are based on the core architecture definition. An i960-based product, such as the i960 CA microprocessor, 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 which are defined by the core architecture.
- On-chip program or data memory.
- Integrated peripherals which 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 parts of the i960 CA microprocessor which are implementation dependent are described below; those parts not described below are part of the core architecture.

ADDRESS SPACE RESTRICTIONS

Address space properties that are implementation-specific to the i960 CA processor are described in the subsections that follow.

Structures in Reserved Memory

Addresses in the range FF00 0000H to FFFF FFFFH are reserved by the i960 architecture. Any uses of reserved memory are implementation specific. The i960 CA processor uses a section of the reserved address space for the initialization boot record. (See *Chapter 14, Initialization and System Requirements.*) The initialization boot record may not exist or may be structured differently for other implementations of the i960 architecture. Code which relies on structures in reserved memory is not portable to all i960-based products.

Internal Data RAM

Internal data RAM — an i960 CA implementation-specific feature — is mapped to the first 1 Kbyte of the processor's address space (0000H - 03FFH). High performance, supervisor-protected data space and the locations assigned for DMA and interrupt functions are special features which are implemented in internal data RAM. Code which relies on these special features is not directly portable to all i960 product implementations.

Instruction Cache

The i960 architecture allows instructions to be cached on-chip in a non-transparent fashion. This means that cache may not detect modification of the program memory by loads, stores or alteration by external agents. (See *Chapter 2, Programming Environment.*) Each implementation of the i960 architecture which uses an integrated instruction cache must provide a mechanism to purge the cache or some other method that forces consistency between external memory and internal cache.

This mechanism is implementation-dependent. Application code which supports modification of the code space must use this implementation-specific feature and, therefore, is not object code portable to all i960 product implementations.

A 1 Kbyte instruction cache is integrated on the i960 CA processor. Its instruction cache does not detect modification of external program memory. This instruction cache is purged using the system control (**sysctl**) instruction, which is specific to the i960 CA processor.

Data and Data Structure Alignment

Not all i960 architecture implementations are required to handle loads and stores to non-aligned addresses. Therefore, code which generates non-aligned addresses is not object-code compatible with all i960 product implementations.

The i960 CA microprocessor, as an implementation-specific feature, automatically handles non-aligned load and store requests. (See *Chapter 10, The Bus Controller.*)

Alignment of architecturally-defined data structures in memory is implementation-dependent. Stack frames are also aligned to implementation-specific boundaries. Data structure alignment is discussed in *Chapter 2, Programming Environment*. Code which relies on specific alignment of data structures in memory is not portable to every implementation of the i960 architecture.

EXTENDED REGISTER SET

i960 architecture defines a way to address 32 additional internal registers in addition to the 16 global and 16 local registers. Register function is implementation-dependent: on the i960 CA device, three extended registers are implemented as special-function registers; on other implementations, these extended registers can be used for other functions or not implemented at all. For example, an implementation can choose to use these registers as general-purpose data registers or as floating point registers. Since the use of the extended register set is not defined, code which addresses these registers is not functionally compatible with all implementations of the i960 architecture.

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. 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.

INSTRUCTION SET

The i960 architecture defines a comprehensive instruction set. Code which 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 sections describe the properties of the an instruction set which are implementation dependent.

Instruction Timing

An objective of the i960 architecture is to allow microarchitectural 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 which depends on instruction execution times, therefore, is not portable to all i960 architecture implementations.

Implementation-Specific Instructions

Most of the i960 CA processor's instruction set is defined by the core architecture. Several instructions are specific to the i960 CA device. These instructions are either functional extensions to the instruction set (e.g., **eshro**) or instructions which control implementation-specific functions (e.g., **sdma**). A box around the instruction mnemonic in *Chapter 9, Instruction Set Reference* denotes an implementation-specific instruction. These instructions are listed below:

• **eshro**	extended shift right ordinal
• **sdma**	set up DMA controller
• **udma**	update DMA data RAM
• **sysctl**	system control

Application code using implementation-specific instructions is not directly portable to the entire i960 family.

INTERRUPT REQUESTS AND POSTING

i960 architecture defines the interrupt servicing mechanism. This includes priority definition, interrupt table structure and interrupt context switching which 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 CA processor's interrupt controller manages external interrupt pins and internal DMA sources. Specific to the i960 CA processor implementation are interrupt controller features, external interrupt pins and NMI pins. Code which configures the interrupt controller — or in other ways interacts with interrupt requestors — is not directly portable to other i960 implementations. On the i960 CA product, interrupts are requested in software with the **sysctl** instruction. This instruction and the software request mechanism are implementation specific.

Posting interrupts is also implementation specific. A pending priorities and pending interrupts field is provided in the interrupt table for interrupt posting. (See *Chapter 6, Interrupts*) An implementation may or may not choose to post all interrupts in the interrupt table in external memory. For example, the i960 CA processor — to minimize latency — posts hardware-requested interrupts internally in the IPND register.

Application code which expects interrupts to be posted in the interrupt table is not object-code portable to all i960-based products. Also, code which requests interrupts by setting bits in the pending priorities and pending interrupts field of the interrupt table is not portable.

INITIALIZATION

The way that an i960-based product is initialized is implementation dependent. For the i960 CA device, pointers to data structures, configuration information and a first instruction pointer are loaded from external memory at initialization. The i960 CA processor defines the initialization boot record, process control block and control table to hold this initial processor state. These structures are implementation dependent. Code which accesses locations in these data structures is not portable to other i960 processor implementations.

OTHER i960™ CA MICROPROCESSOR IMPLEMENTATION-SPECIFIC FEATURES

Subsections that follow describe additional implementation-specific features of the i960 CA microprocessor. These features do not relate directly to application code portability.

Data Control Peripherals

The DMA controller, 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 add or subtract such system integration features.

Implementation-Specific Faults

The architecture defines a subset of fault types and subtypes which apply to all implementations of the architecture. Other fault types and subtypes may be defined by implementations to detect errant conditions which relate to implementation-specific features. For example, the i960 CA microprocessor provides an operation-unaligned fault for detecting non-aligned memory accesses. Future i960 processor implementations which generate this fault will assign the same fault type and subtype number to the fault.

External System Requirements

External system requirements for the i960 CA microprocessor are not defined by the architecture. The external bus, $\overline{\text{RESET}}$ pin, clock input, power and ground requirements and I/O characteristics are all specific to the i960 CA microprocessor implementation.

Appendix D
Instruction Set Reference

APPENDIX D
INSTRUCTION SET REFERENCE

This appendix describes the encoding format for instructions in the i960 CA microprocessor. Included is a description of the four instruction formats and how the addressing modes relate to these formats.

GENERAL INSTRUCTION FORMAT

i960 architecture defines four basic instruction encoding formats, as shown in Figure D.1: REG, COBR, CTRL and MEM. 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 permits an optional second word to hold a displacement value. The following sections describe each format's instruction word fields.

REG FORMAT

REG format is used for operations performed on data contained in global, local or special function registers. Most of the i960 family's instructions use this format.

REG instructions opcode is 12 bits long (three hexadecimal digits) and is split between bits 7 through 10 (low opcode) and bits 24 through 31 (high opcode). For example, **addi** opcode is 591H. Here, 59H is contained in bits 24 through 31; 1H is contained in bits 7 through 10.

src1 and *src2* fields specify the instruction's source operands. Operands can be global or local registers, literals or special-function registers. Mode flags (M1 for *src1* and M2 for *src2*), special-purpose flags (s1 for *src1* and s2 for *src2*) and the instruction type determine what an operand specifies:

- If a mode flag and its associated special-purpose flag are set to 0, the respective *src1* or *src2* field specifies a global or local register.
- If the mode flag is set to 1 and the special-purpose flag is set to 0, the field specifies a literal in the range of 0 to 31.
- If the mode flag is set to 0 and the special-purpose flag is set to 1, the field specifies a special-function register.

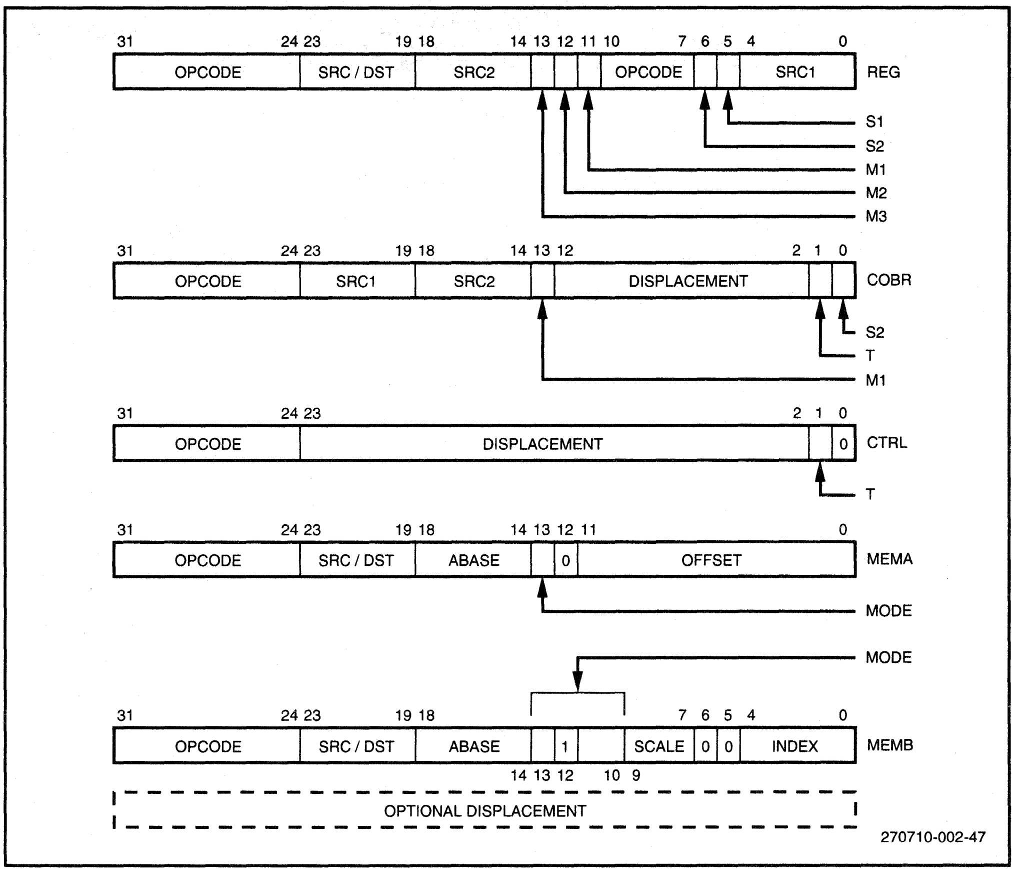


Figure D.1. Instruction Formats

Table D.1 shows the relationship between mode flags, special-purpose flags and *src1* and *src2* operands.

Table D.1. Encoding of *src1* and *src2* Fields in REG Format

M1 or M2	S1 or S2	src1 or src2 Operand Value	Register Number	Literal Value
0	0	00000_2-01111_2 10000_2-11111_2	r0-r15 g0-g15	
1	0	00000_2-11111_2		0-31
0	1	00000_2-11111_2	sf0-sf31	
1	1	Reserved		

NOTE

On the i960 CA processor, the only special function registers implemented are sf0, sf1 and sf2.

The *src/dst* field can specify a source operand, a destination operand or both, depending on the instruction. Here again, mode flag M3 determines how this field is used. Table D.2 shows this relationship.

Table D.2. Encoding of *src/dst* Field in REG Format

M3	*src/dst*	*src* Only	*dst* Only
0	g0 .. g15 r0 .. r15	g0 .. g15 r0 .. r15	g0 .. g15 r0 .. r15
1	Not Allowed	Literal	sf0 .. sf31

If M3 is clear, the *src/dst* operand is a global or local register that is encoded as shown in Table D.1. 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.

COBR FORMAT

The COBR format is used primarily for compare-and-branch instructions; however, test-if instructions also COBR. COBR opcode field is eight bits — two hexadecimal digits. *src1* and *src2* fields specify the instruction's source operands (complete encoding of *src1, src2* and *dst* is the same as is shown in Table D.5):

- *src1* can specify a global or local register or a literal as determined by mode flag M1
- *src2* can specify a global or local register or special function register as determined by special-purpose flag S2

The *T* flag supports branch prediction for conditional instructions: if *T* is set to 0, the condition being tested is likely to be true; if set to 1, the condition is likely to be false. An implementation may choose to ignore this bit.

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 goes as a result of a comparison. The displacement field 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 current instruction's IP.

NOTE

To allow label usage in the assembly-language version of the COBR format instructions, the i960 assembler converts a *targ* (target) operand value in an assembly-language instruction into the displacement value required for the COBR format, using the following calculation:

$$\texttt{displacement} = \frac{\texttt{(targ - IP)}}{4}$$

For the test-if instructions, only the *src1* field is used. Here, this field specifies a destination global or local register; M1 is ignored.

CTRL FORMAT

The CTRL format is used for instructions that branch to a new IP, including the branch, branch-if, **bal** and **call** instructions; **ret** also uses this format. CTRL opcode field is eight bits — two hexadecimal digits.

Branch target address is specified with the displacement field in the same manner as COBR format instructions. The displacement field specifies a word displacement or 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* flag performs the same prediction function for CTRL instructions as it does for COBR instructions.

MEM FORMAT

The MEM format is used for instructions that require a memory address to be computed. These instructions include the load, store and **lda** instructions. Also, the extended versions of the branch, branch-and-link and call instructions (**bx**, **balx** and **callx**) use this format.

The two MEM-format encodings are MEMA and MEMB. MEMB can optionally add a 32-bit displacement (contained in a second word) to the instruction. Bit 12 of the instruction's first word determines whether MEMA (clear) or MEMB (set) is used.

The opcode field is eight bits long for either encoding. The *src/dst* field specifies a global or local register. For load instructions, *src/dst* specifies the destination register for a word loaded into the processor from memory or, for operands larger than one word, the first of successive destination registers. For store instructions, this field specifies the register or group of registers that contain the source operand to be stored in memory.

The *mode* field determines the address mode used for the instruction. Table D.3 summarizes the addressing modes for the two MEM-format encodings. Fields used in these addressing modes are described in the following sections.

Table D.3. Addressing Modes for MEM Format Instructions

Format Bits	Mode	Address Computation
MEMA	00_2	offset
	10_2	(abase) + offset
MEMB	0100_2	(abase)
	0101_2	(IP) + displacement + 8
	0110_2	reserved
	0111_2	(abase) + (index) * 2^{scale}
	1100_2	displacement
	1101_2	(abase) + displacement
	1110_2	(index) * 2^{scale} + displacement
	1111_2	(abase) + (index) * 2^{scale} + displacement

NOTE

In these address computations, a field in parentheses — e.g., (abase) — indicates that the value in the specified register is used in the computation. Usage of a reserved encoding causes generation of an invalid-opcode fault.

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* field is set to 00_2), the processor interprets the *offset* field as an offset from byte 0 of the current process address space; the *abase* field is ignored. Using this addressing mode along with the **lda** instruction allows a constant in the range 0 to 4096 to be loaded into a register.

For the register-indirect-with-offset addressing mode (*mode* field is set to 10_2), *offset* field value is added to the address in the *abase* register. Setting the offset value to zero creates a register indirect addressing mode; however, this operation can generally be carried out faster by using the MEMB version of this addressing mode.

MEMB Format Addressing

The MEMB format provides the following seven addressing modes:

- absolute displacement
- register indirect with displacement
- register indirect with index and displacement
- IP with displacement
- register indirect
- register indirect with index
- index with displacement

The *abase* and *index* fields specify local or global registers, the contents of which are used in address computation. When the *index* field is used in an addressing mode, the processor automatically scales the index register value by the amount specified in the *scale* field. Table D.4 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 D.4. Encoding of Scale Field

Scale	Scale Factor (Multiplier)
000_2	1
001_2	2
010_2	4
011_2	8
100_2	16
101_2 to 111_2	Reserved

NOTE

Usage of a reserved encoding causes generation of an invalid-opcode fault.

For the IP with displacement mode, the value of the displacement field plus eight is added to the address of the current instruction.

INSTRUCTION REFERENCE BY OPCODE

This section lists the instruction encoding for each i960 CA microprocessor instruction. Instructions are grouped by instruction format and listed by opcode within each format. Table D.5 describes the meaning of each M3, M2, M1, S2, S1 and T bit combinations for each format.

Table D.5 Miscellaneous Instruction Encoding Bits

M3	M2	M1	S2	S1	T	Description
						REG Format
x	x	0	x	0	—	*src1* is a global or local register
x	x	1	x	0	—	*src1* is a literal
x	x	0	x	1	—	*src1* is a special function register
x	x	1	x	1	—	reserved
x	0	x	0	x	—	*src2* is a global or local register
x	1	x	0	x	—	*src2* is a literal
x	0	x	1	x	—	*src2* is a special function register
x	1	x	1	x	—	reserved
0	x	x	x	x	—	*src/dst* is a global or local register
1	x	x	x	x	—	*src/dst* is a literal when used as a source or a special function register when used as a destination. M3 may not be 1 when *src/dst* is used both as a source and destination in an instruction (**atmod, modify, extract, modpc**).
						COBR Format
—	—	0	0	—	x	*src1 src2* and *dst* are global or local registers
—	—	1	0	—	x	*src1* is a literal, *src2* and *dst* are global or local registers
—	—	0	1	—	x	*src1* is a global or local register, *src2* and *dst* are special function registers
—	—	1	1	—	0	*src1* is a literal, *src2* and *dst* are special function registers
						COBR Format and CTRL Format
—	—	x	—	x	1	Outcome of conditional test is predicted to be true.
—	—	x	—	x	0	Outcome of conditional test is predicted to be false.

Table D.6. REG Format Instruction Encodings

Opcode	Mnemonic	Opcode (11 - 4)	src/dst	src2	Mode			Opcode (3-0)	Special Flags		src1
		31 24	23....19	18 ... 14	13	12	11	10.......7	6	5	4.........0
58:0	**notbit**	0101 1000	*dst*	*src*	M3	M2	M1	0000	S2	S1	*bitpos*
58:1	**and**	0101 1000	*dst*	*src2*	M3	M2	M1	0001	S2	S1	*src1*
58:2	**andnot**	0101 1000	*dst*	*src2*	M3	M2	M1	0010	S2	S1	*src1*
58:3	**setbit**	0101 1000	*dst*	*src*	M3	M2	M1	0011	S2	S1	*bitpos*
58:4	**notand**	0101 1000	*dst*	*src2*	M3	M2	M1	0100	S2	S1	*src1*
58:6	**xor**	0101 1000	*dst*	*src2*	M3	M2	M1	0110	S2	S1	*src1*
58:7	**or**	0101 1000	*dst*	*src2*	M3	M2	M1	0111	S2	S1	*src1*
58:8	**nor**	0101 1000	*dst*	*src2*	M3	M2	M1	1000	S2	S1	*src1*
58:9	**xnor**	0101 1000	*dst*	*src2*	M3	M2	M1	1001	S2	S1	*src1*
58:A	**not**	0101 1000	*dst*		M3	M2	M1	1010	S2	S1	*src*
58:B	**ornot**	0101 1000	*dst*	*src2*	M3	M2	M1	1011	S2	S1	*src1*
58:C	**clrbit**	0101 1000	*dst*	*src*	M3	M2	M1	1100	S2	S1	*bitpos*
58:D	**notor**	0101 1000	*dst*	*src2*	M3	M2	M1	1101	S2	S1	*src1*
58:E	**nand**	0101 1000	*dst*	*src2*	M3	M2	M1	1110	S2	S1	*src1*
58:F	**alterbit**	0101 1000	*dst*	*src*	M3	M2	M1	1111	S2	S1	*bitpos*
59:0	**addo**	0101 1001	*dst*	*src2*	M3	M2	M1	0000	S2	S1	*src1*
59:1	**addi**	0101 1001	*dst*	*src2*	M3	M2	M1	0001	S2	S1	*src1*
59:2	**subo**	0101 1001	*dst*	*src2*	M3	M2	M1	0010	S2	S1	*src1*
59:3	**subi**	0101 1001	*dst*	*src2*	M3	M2	M1	0011	S2	S1	*src1*
59:8	**shro**	0101 1001	*dst*	*src*	M3	M2	M1	1000	S2	S1	*len*
59:A	**shrdi**	0101 1001	*dst*	*src*	M3	M2	M1	1010	S2	S1	*len*
59:B	**shri**	0101 1001	*dst*	*src*	M3	M2	M1	1011	S2	S1	*len*
59:C	**shlo**	0101 1001	*dst*	*src*	M3	M2	M1	1100	S2	S1	*len*
59:D	**rotate**	0101 1001	*dst*	*src*	M3	M2	M1	1101	S2	S1	*len*
59:E	**shli**	0101 1001	*dst*	*src*	M3	M2	M1	1110	S2	S1	*len*
5A:0	**cmpo**	0101 1010		*src2*	M3	M2	M1	0000	S2	S1	*src1*
5A:1	**cmpi**	0101 1010		*src2*	M3	M2	M1	0001	S2	S1	*src1*
5A:2	**concmpo**	0101 1010		*src2*	M3	M2	M1	0010	S2	S1	*src1*
5A:3	**concmpi**	0101 1010		*src2*	M3	M2	M1	0011	S2	S1	*src1*
5A:4	**cmpinco**	0101 1010	*dst*	*src2*	M3	M2	M1	0100	S2	S1	*src1*
5A:5	**cmpinci**	0101 1010	*dst*	*src2*	M3	M2	M1	0101	S2	S1	*src1*
5A:6	**cmpdeco**	0101 1010	*dst*	*src2*	M3	M2	M1	0110	S2	S1	*src1*
5A:7	**cmpdeci**	0101 1010	*dst*	*src2*	M3	M2	M1	0111	S2	S1	*src1*
5A:C	**scanbyte**	0101 1010		*src2*	M3	M2	M1	1100	S2	S1	*src1*
5A:E	**chkbit**	0101 1010		*src*	M3	M2	M1	1110	S2	S1	*bitpos*
5B:0	**addc**	0101 1011	*dst*	*src2*	M3	M2	M1	0000	S2	S1	*src1*
5B:2	**subc**	0101 1011	*dst*	*src2*	M3	M2	M1	0010	S2	S1	*src1*
5C:C	**mov**	0101 1100	*dst*		M3	M2	M1	1100	S2	S1	*src*
5D:8	**eshro**	0101 1101	*dst*	*src2*	M3	M2	M1	1000	S2	S1	*src1*
5D:C	**movl**	0101 1101	*dst*		M3	M2	M1	1100	S2	S1	*src*
5E:C	**movt**	0101 1110	*dst*		M3	M2	M1	1100	S2	S1	*src*
5F:C	**movq**	0101 1111	*dst*		M3	M2	M1	1100	S2	S1	*src*

Opcode	Mnemonic	Opcode (11 - 4)	src/dst	src2	Mode			Opcode (3-0)	Special Flags		src1
		31 24	23....19	18 ... 14	13	12	11	10.......7	6	5	4..........0
63:0	**sdma**	0110 0011	*src3*	*src2*	M3	M2	M1	0000	S2	S1	*src1*
63:1	**udma**	0110 0011						0001			
64:0	**spanbit**	0110 0100	*dst*		M3	M2	M1	0000	S2	S1	*src*
64:1	**scanbit**	0110 0100	*dst*		M3	M2	M1	0001	S2	S1	*src*
64:5	**modac**	0110 0100	*mask*	*src*	M3	M2	M1	0101	S2	S1	*dst*
65:0	**modify**	0110 0101	*src/dst*	*src*	M3	M2	M1	0000	S2	S1	*mask*
65:1	**extract**	0110 0101	*src/dst*	*len*	M3	M2	M1	0001	S2	S1	*bitpos*
65:4	**modtc**	0110 0101	*mask*	*src*	M3	M2	M1	0100	S2	S1	*dst*
65:5	**modpc**	0110 0101	*src/dst*	*mask*	M3	M2	M1	0101	S2	S1	*src*
65:9	**sysctl**	0110 0101	*src3*	*src2*	M3	M2	M1	1001	S2	S1	*src1*
66:0	**calls**	0110 0110			M3	M2	M1	0000	S2	S1	*src*
66:B	**mark**	0110 0110			M3	M2	M1	1011	S2	S1	
66:C	**fmark**	0110 0110			M3	M2	M1	1100	S2	S1	
66:D	**flushreg**	0110 0110			M3	M2	M1	1101	S2	S1	
66:F	**syncf**	0110 0110			M3	M2	M1	1111	S2	S1	
67:0	**emul**	0110 0111	*dst*	*src2*	M3	M2	M1	0000	S2	S1	*src1*
67:1	**ediv**	0110 0111	*dst*	*src2*	M3	M2	M1	0001	S2	S1	*src1*
70:1	**mulo**	0111 0000	*dst*	*src2*	M3	M2	M1	0001	S2	S1	*src1*
70:8	**remo**	0111 0000	*dst*	*src2*	M3	M2	M1	1000	S2	S1	*src1*
70:B	**divo**	0111 0000	*dst*	*src2*	M3	M2	M1	1011	S2	S1	*src1*
74:1	**muli**	0111 0100	*dst*	*src2*	M3	M2	M1	0001	S2	S1	*src1*
74:8	**remi**	0111 0100	*dst*	*src2*	M3	M2	M1	1000	S2	S1	*src1*
74:9	**modi**	0111 0100	*dst*	*src2*	M3	M2	M1	1001	S2	S1	*src1*
74:B	**divi**	0111 0100	*dst*	*src2*	M3	M2	M1	1011	S2	S1	*src1*

Table D.7. COBR Format Instruction Encodings

Opcode	Mnemonic	Opcode	src1	src2	M	Displacement	T	S2
		31......... 24	23... 19	18....14	13	12.......................................2	1	0
20	**testno**	0010 0000	*dst*		M1		T	S2
21	**testg**	0010 0001	*dst*		M1		T	S2
22	**teste**	0010 0010	*dst*		M1		T	S2
23	**testge**	0010 0011	*dst*		M1		T	S2
24	**testl**	0010 0100	*dst*		M1		T	S2
25	**testne**	0010 0101	*dst*		M1		T	S2
26	**testle**	0010 0110	*dst*		M1		T	S2
27	**testo**	0010 0111	*dst*		M1		T	S2
30	**bbc**	0011 0000	*bitpos*	*src*	M1	*targ*	T	S2
31	**cmpobg**	0011 0001	*src1*	*src2*	M1	*targ*	T	S2
32	**cmpobe**	0011 0010	*src1*	*src2*	M1	*targ*	T	S2
33	**cmpobge**	0011 0011	*src1*	*src2*	M1	*targ*	T	S2
34	**cmpobl**	0011 0100	*src1*	*src2*	M1	*targ*	T	S2
35	**cmpobne**	0011 0101	*src1*	*src2*	M1	*targ*	T	S2
36	**cmpoble**	0011 0110	*src1*	*src2*	M1	*targ*	T	S2
37	**bbs**	0011 0111	*bitpos*	*src*	M1	*targ*	T	S2
38	**cmpibno**	0011 1000	*src1*	*src2*	M1	*targ*	T	S2
39	**cmpibg**	0011 1001	*src1*	*src2*	M1	*targ*	T	S2
3A	**cmpibe**	0011 1010	*src1*	*src2*	M1	*targ*	T	S2
3B	**cmpibge**	0011 1011	*src1*	*src2*	M1	*targ*	T	S2
3C	**cmpibl**	0011 1100	*src1*	*src2*	M1	*targ*	T	S2
3D	**cmpibne**	0011 1101	*src1*	*src2*	M1	*targ*	T	S2
3E	**cmpible**	0011 1110	*src1*	*src2*	M1	*targ*	T	S2
3F	**cmpibo**	0011 1111	*src1*	*src2*	M1	*targ*	T	S2

Table D.8. CTRL Format Instruction Encodings

Opcode	Mnemonic	Opcode	Displacement	T	0
		31 24	23 2	1	0
08	**b**	0000 1000	*targ*	T	0
09	**call**	0000 1001	*targ*	T	0
0A	**ret**	0000 1010		T	0
0B	**bal**	0000 1011	*targ*	T	0
10	**bno**	0001 0000	*targ*	T	0
11	**bg**	0001 0001	*targ*	T	0
12	**be**	0001 0010	*targ*	T	0
13	**bge**	0001 0011	*targ*	T	0
14	**bl**	0001 0100	*targ*	T	0
15	**bne**	0001 0101	*targ*	T	0
16	**ble**	0001 0110	*targ*	T	0
17	**bo**	0001 0111	*targ*	T	0
18	**faultno**	0001 1000		T	0
19	**faultg**	0001 1001		T	0
1A	**faulte**	0001 1010		T	0
1B	**faultge**	0001 1011		T	0
1C	**faultl**	0001 1100		T	0
1D	**faultne**	0001 1101		T	0
1E	**faultle**	0001 1110		T	0
1F	**faulto**	0001 1111		T	0

Table D.9. MEM Format Instruction Encodings

31..........24	23....19	18 ...14	13 12	11 0
Opcode	**src/ dst**	**ABASE**	**Mode**	**Offset**

31..........24	23....19	18 ...14	13 12 11 10	9 ...7	6 5	4 0
Opcode	**src/ dst**	**ABASE**	**Mode**	**Scale**	**00**	**Index**
Displacement						

Effective Address

efa =										
offset	*Opcode*	*dst*		0	0	*offset*				
offset(reg)	*Opcode*	*dst*	*reg*	1	0	*offset*				
(reg)	*Opcode*	*dst*	*reg*	0	1	0	0		00	
disp + 8 (IP)	*Opcode*	*dst*		0	1	0	1		00	
	displacement									
(reg1)[reg2 * scale]	*Opcode*	*dst*	*reg1*	0	1	1	1	*scale*	00	*reg2*
disp	*Opcode*	*dst*		1	1	0	0		00	
	displacement									
disp(reg)	*Opcode*	*dst*	*reg*	1	1	0	1		00	
	displacement									
disp[reg * scale]	*Opcode*	*dst*		1	1	1	0	*scale*	00	*reg*
	displacement									
disp(reg1)[reg2*scale]	*Opcode*	*dst*	*reg1*	1	1	1	1	*scale*	00	*reg2*
	displacement									

Opcode	Mnemonic
80	**ldob**
82	**stob**
84	**bx**
85	**balx**
86	**callx**
88	**ldos**
8A	**stos**
8C	**lda**
90	**ld**
92	**st**

Opcode	Mnemonic
98	**ldl**
9A	**stl**
A0	**ldt**
A2	**stt**
B0	**ldq**
B2	**stq**
C0	**ldib**
C2	**stib**
C8	**ldis**
CA	**stis**

Appendix E
Register and Data Structure Reference

APPENDIX E
REGISTER AND DATA STRUCTURE REFERENCE

OVERVIEW

Registers and data structures, listed alphabetically, are:

Arithmetic Controls Register (AC)

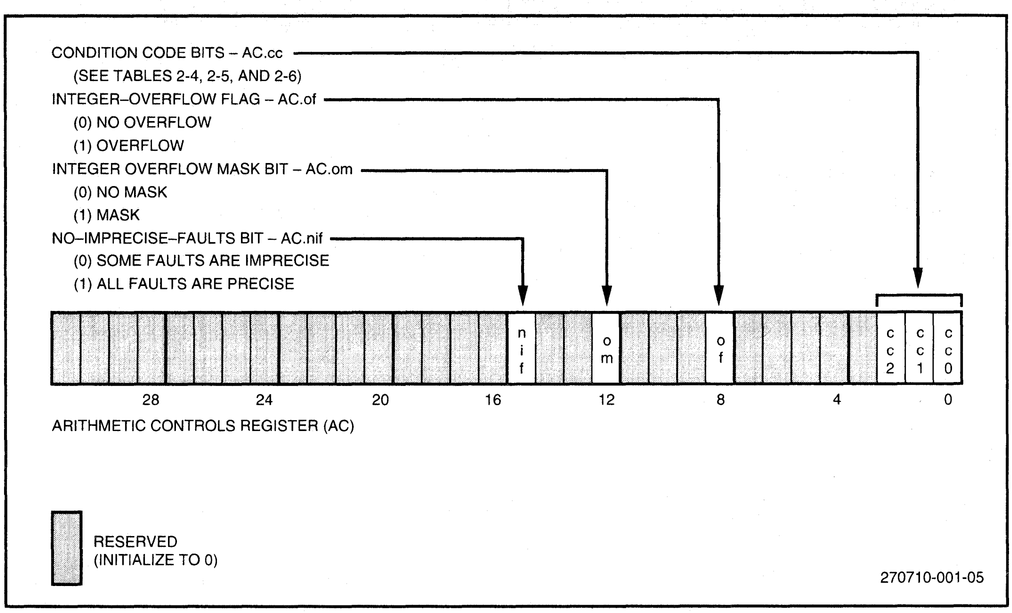
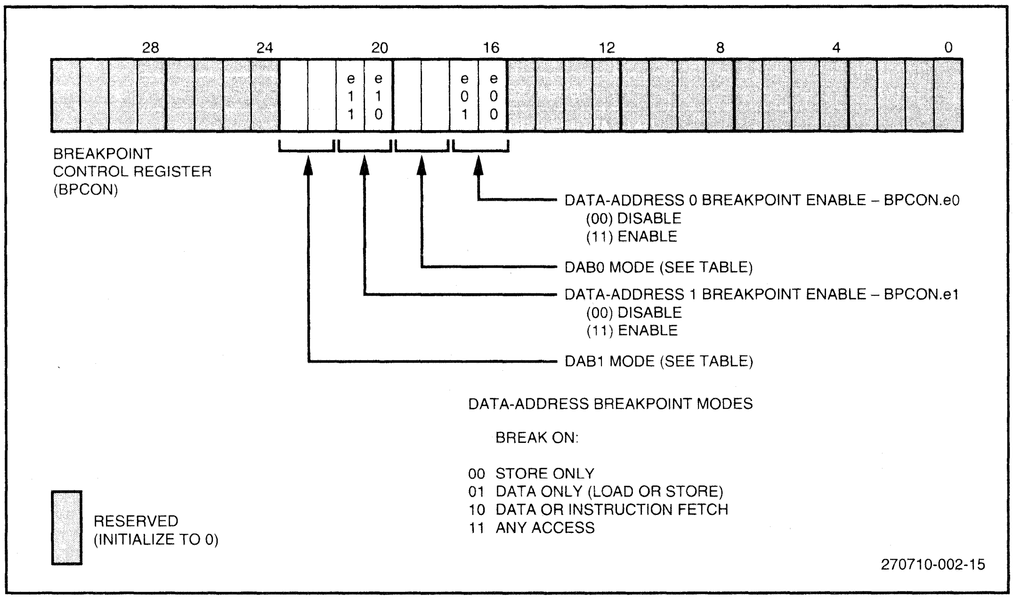

CONDITION CODE BITS – AC.cc
(SEE TABLES 2-4, 2-5, AND 2-6)
INTEGER–OVERFLOW FLAG – AC.of
(0) NO OVERFLOW
(1) OVERFLOW
INTEGER OVERFLOW MASK BIT – AC.om
(0) NO MASK
(1) MASK
NO–IMPRECISE–FAULTS BIT – AC.nif
(0) SOME FAULTS ARE IMPRECISE
(1) ALL FAULTS ARE PRECISE
nif
om
of
cc2
cc1
cc0
28
24
20
16
12
8
4
0
ARITHMETIC CONTROLS REGISTER (AC)
RESERVED (INITIALIZE TO 0)
270710-001-05

Figure E.1. Arithmetic Controls (AC) Register

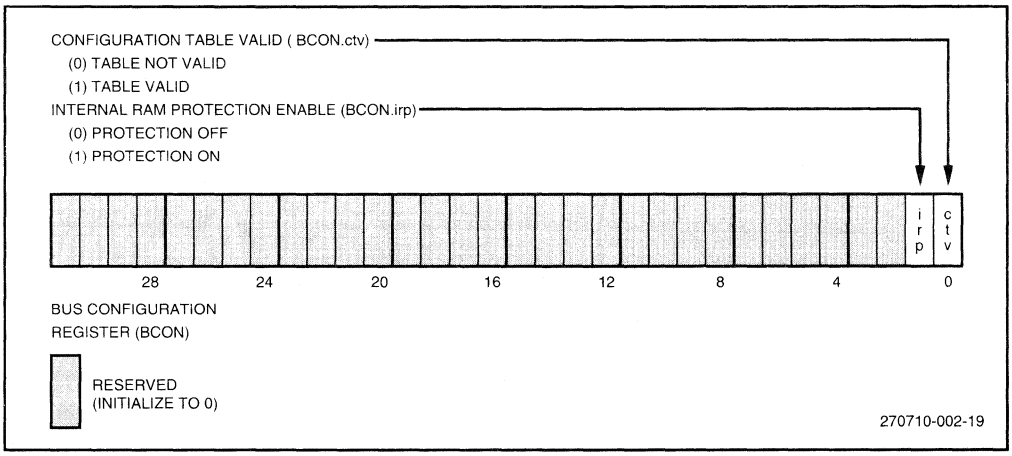
Breakpoint Control Register (BPCON)



28
24
20
16
12
8
4
0
e11
e10
e01
e00
BREAKPOINT CONTROL REGISTER (BPCON)
DATA-ADDRESS 0 BREAKPOINT ENABLE – BPCON.e0
(00) DISABLE
(11) ENABLE
DAB0 MODE (SEE TABLE)
DATA-ADDRESS 1 BREAKPOINT ENABLE – BPCON.e1
(00) DISABLE
(11) ENABLE
DAB1 MODE (SEE TABLE)
DATA-ADDRESS BREAKPOINT MODES
BREAK ON:
00 STORE ONLY
01 DATA ONLY (LOAD OR STORE)
10 DATA OR INSTRUCTION FETCH
11 ANY ACCESS
RESERVED (INITIALIZE TO 0)
270710-002-15

Figure E.2. Hardware Breakpoint Control Register (BPCON)

Bus Configuration Register (BCON)



CONFIGURATION TABLE VALID (BCON.ctv)
(0) TABLE NOT VALID
(1) TABLE VALID
INTERNAL RAM PROTECTION ENABLE (BCON.irp)
(0) PROTECTION OFF
(1) PROTECTION ON
irp
ctv
28
24
20
16
12
8
4
0
BUS CONFIGURATION REGISTER (BCON)
RESERVED (INITIALIZE TO 0)
270710-002-19

Figure E.3. BCON Register

Control Table

31 0

IP BREAKPOINT 0 (IPB0)	0H
IP BREAKPOINT 1 (IPB1)	4H
DATA ADDRESS BREAKPOINT 0 (DAB0)	8H
DATA ADDRESS BREAKPOINT 1 (DAB1)	CH
INTERRUPT MAP 0 (IMAP0)	10H
INTERRUPT MAP 1 (IMAP1)	14H
INTERRUPT MAP 2 (IMAP2)	18H
INTERRUPT CONTROL (ICON)	1CH
MEMORY REGION 0 CONFIGURATION (MCON0)	20H
MEMORY REGION 1 CONFIGURATION (MCON1)	24H
MEMORY REGION 2 CONFIGURATION (MCON2)	28H
MEMORY REGION 3 CONFIGURATION (MCON3)	2CH
MEMORY REGION 4 CONFIGURATION (MCON4)	30H
MEMORY REGION 5 CONFIGURATION (MCON5)	34H
MEMORY REGION 6 CONFIGURATION (MCON6)	38H
MEMORY REGION 7 CONFIGURATION (MCON7)	3CH
MEMORY REGION 8 CONFIGURATION (MCON8)	40H
MEMORY REGION 9 CONFIGURATION (MCON9)	44H
MEMORY REGION 10 CONFIGURATION (MCON10)	48H
MEMORY REGION 11 CONFIGURATION (MCON11)	4CH
MEMORY REGION 12 CONFIGURATION (MCON12)	50H
MEMORY REGION 13 CONFIGURATION (MCON13)	54H
MEMORY REGION 14 CONFIGURATION (MCON14)	58H
MEMORY REGION 15 CONFIGURATION (MCON15)	5CH
RESERVED (INITIALIZE TO 0)	60H
BREAKPOINT CONTROL (BPCON)	64H
TRACE CONTROLS (TC)	68H
BUS CONFIGURATION CONTROL (BCON)	6CH

270710-002-02

Figure E.4. Control Table

Data Address Breakpoint Registers (DAB0-DAB1)

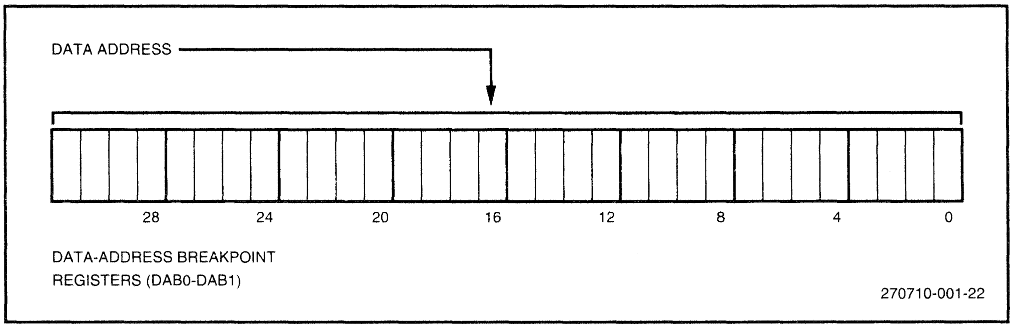


Figure E.5. Data Address Breakpoint Registers (DAB0 - DAB1)

DMA Command Register (DMAC) (sf2)

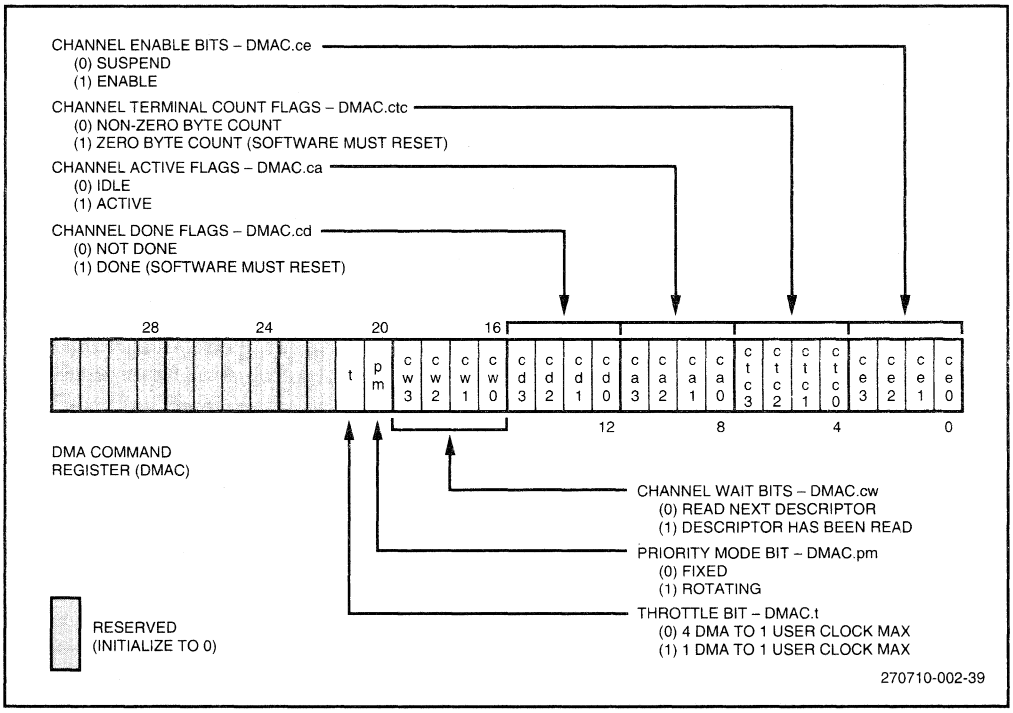

CHANNEL ENABLE BITS – DMAC.ce
(0) SUSPEND
(1) ENABLE
CHANNEL TERMINAL COUNT FLAGS – DMAC.ctc
(0) NON-ZERO BYTE COUNT
(1) ZERO BYTE COUNT (SOFTWARE MUST RESET)
CHANNEL ACTIVE FLAGS – DMAC.ca
(0) IDLE
(1) ACTIVE
CHANNEL DONE FLAGS – DMAC.cd
(0) NOT DONE
(1) DONE (SOFTWARE MUST RESET)
28
24
20
16
t
pm
cw3
cw2
cw1
cw0
cd3
cd2
cd1
cd0
ca3
ca2
ca1
ca0
ctc3
ctc2
ctc1
ctc0
ce3
ce2
ce1
ce0
12
8
4
0
DMA COMMAND
REGISTER (DMAC)
CHANNEL WAIT BITS – DMAC.cw
(0) READ NEXT DESCRIPTOR
(1) DESCRIPTOR HAS BEEN READ
PRIORITY MODE BIT – DMAC.pm
(0) FIXED
(1) ROTATING
THROTTLE BIT – DMAC.t
(0) 4 DMA TO 1 USER CLOCK MAX
(1) 1 DMA TO 1 USER CLOCK MAX
RESERVED
(INITIALIZE TO 0)
270710-002-39

Figure E.6. DMA Command Register (DMAC)

DMA Control Word

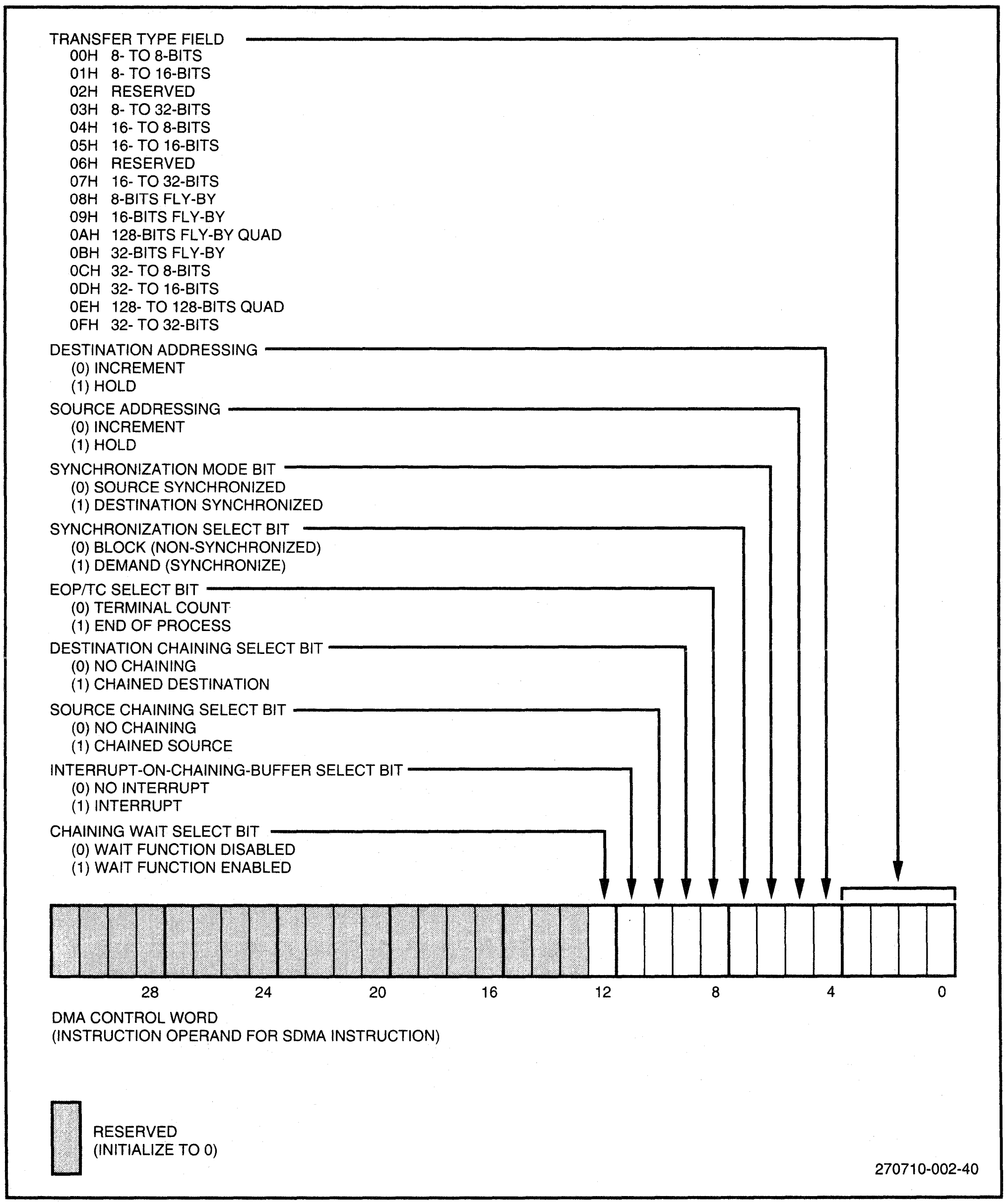


Figure E.7. DMA Control Word

Fault Record

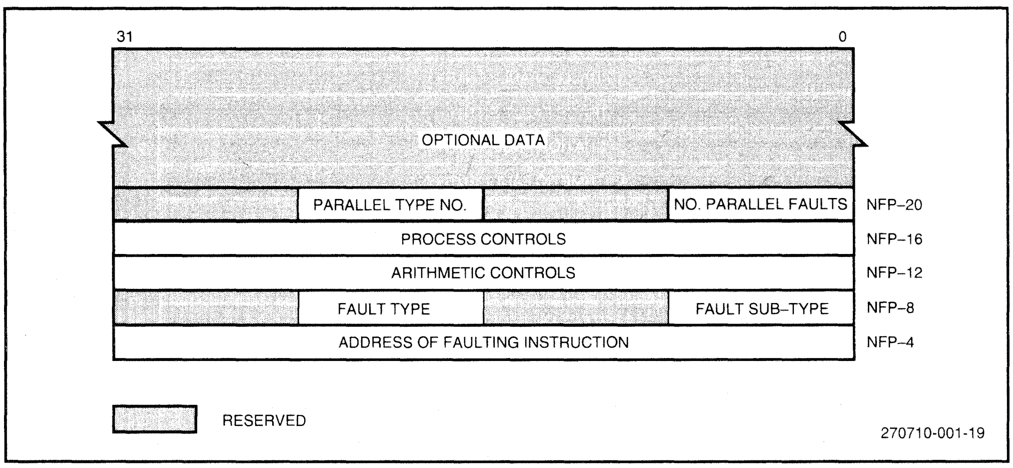


Figure E.8. Fault Record

Fault Table

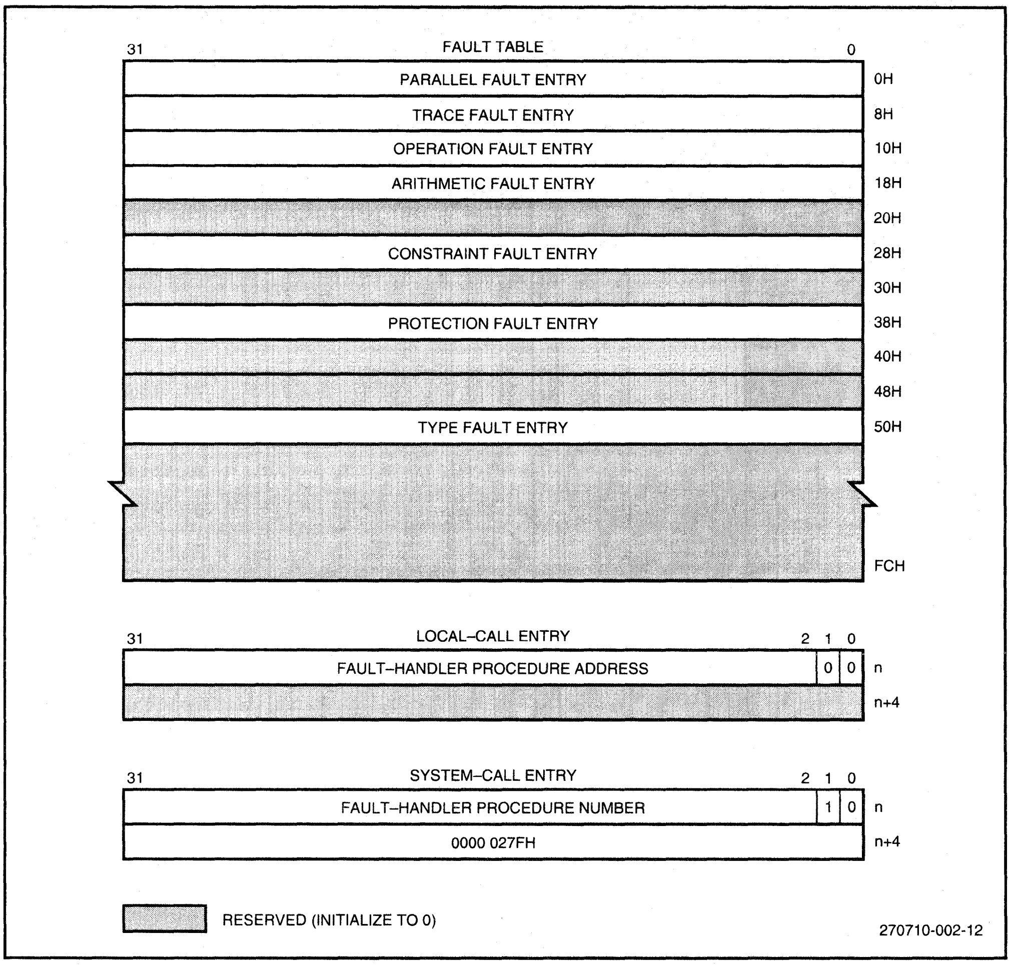


Figure E.9. Fault Table and Fault Table Entries

Global and Local Registers

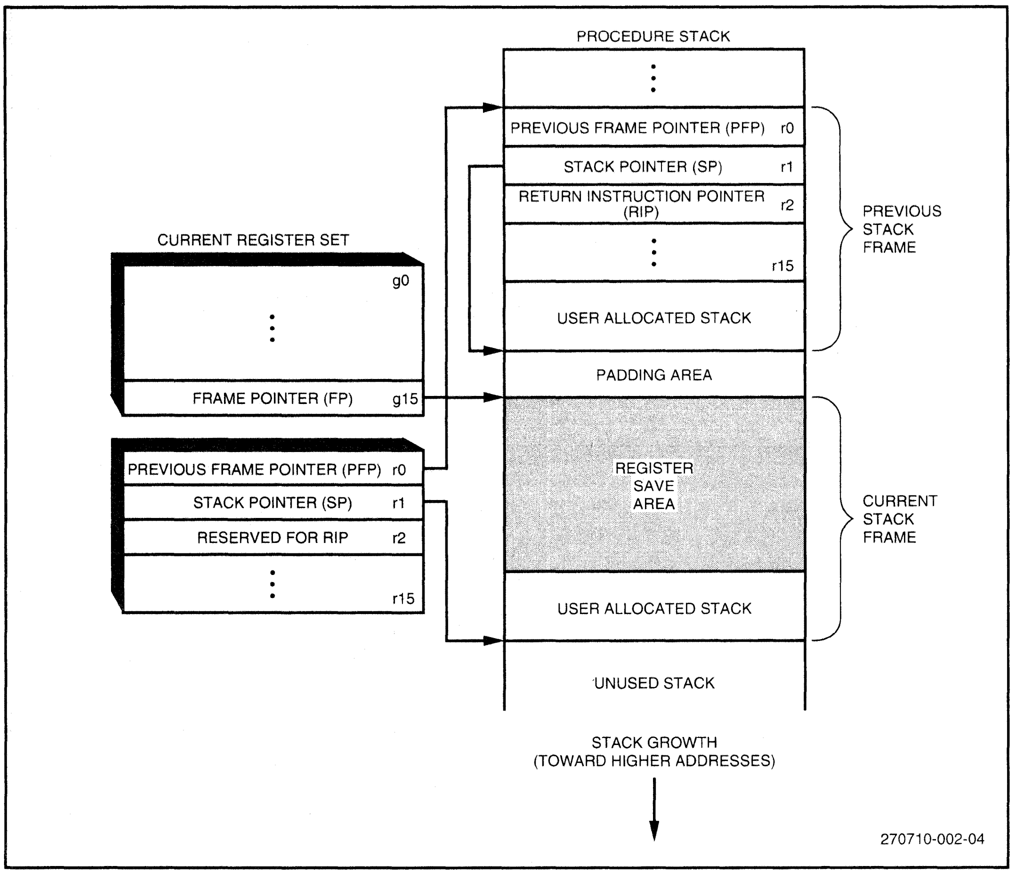


Figure E.10. Procedure Stack Structure and Local Registers

Initialization Boot Record (IBR) and Process Control Block (PRCB)

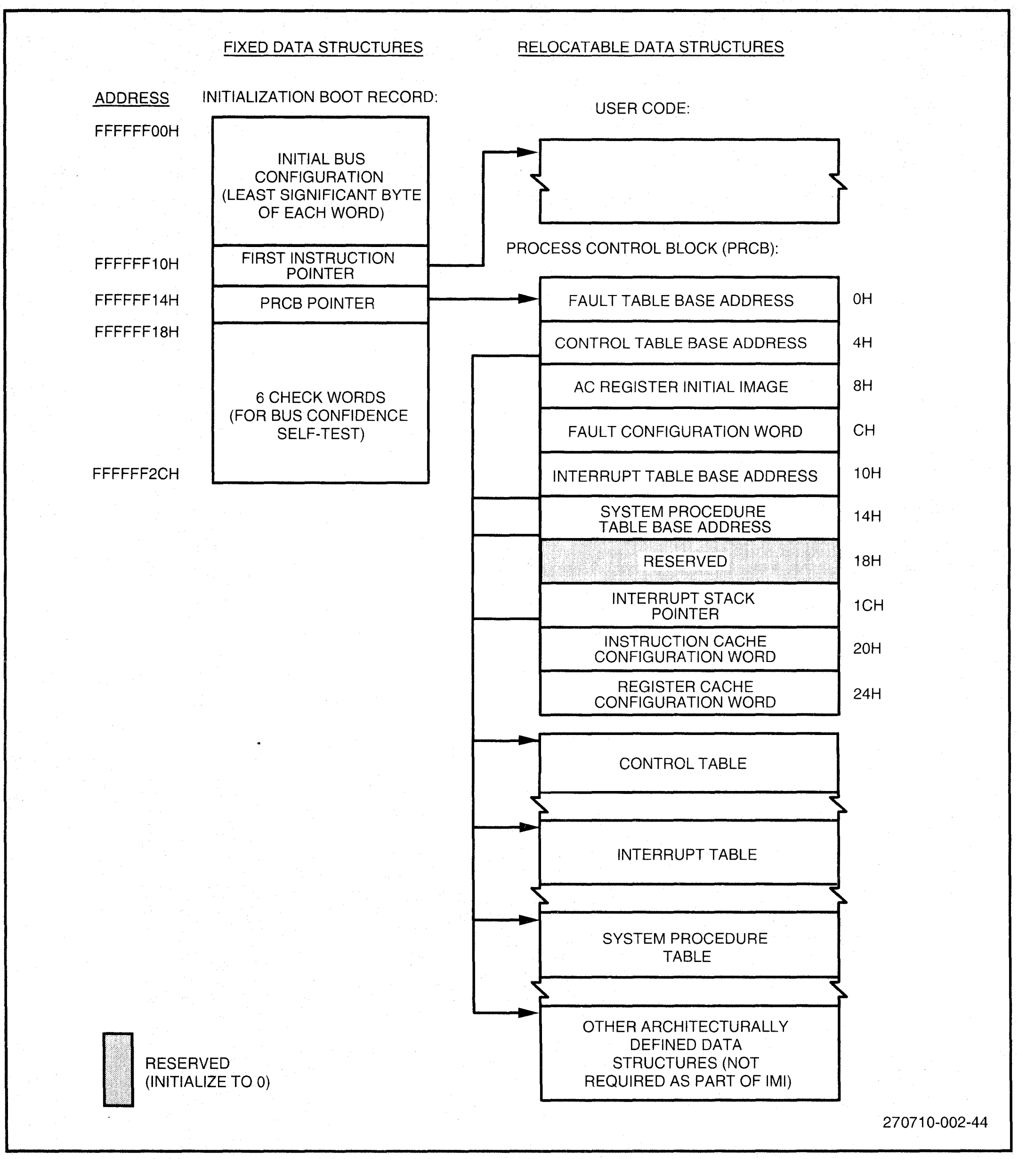

FIXED DATA STRUCTURES
RELOCATABLE DATA STRUCTURES
ADDRESS
INITIALIZATION BOOT RECORD:
FFFFFF00H
INITIAL BUS CONFIGURATION (LEAST SIGNIFICANT BYTE OF EACH WORD)
FFFFFF10H
FIRST INSTRUCTION POINTER
FFFFFF14H
PRCB POINTER
FFFFFF18H
6 CHECK WORDS (FOR BUS CONFIDENCE SELF-TEST)
FFFFFF2CH
USER CODE:
PROCESS CONTROL BLOCK (PRCB):
FAULT TABLE BASE ADDRESS 0H
CONTROL TABLE BASE ADDRESS 4H
AC REGISTER INITIAL IMAGE 8H
FAULT CONFIGURATION WORD CH
INTERRUPT TABLE BASE ADDRESS 10H
SYSTEM PROCEDURE TABLE BASE ADDRESS 14H
RESERVED 18H
INTERRUPT STACK POINTER 1CH
INSTRUCTION CACHE CONFIGURATION WORD 20H
REGISTER CACHE CONFIGURATION WORD 24H
CONTROL TABLE
INTERRUPT TABLE
SYSTEM PROCEDURE TABLE
OTHER ARCHITECTURALLY DEFINED DATA STRUCTURES (NOT REQUIRED AS PART OF IMI)
RESERVED (INITIALIZE TO 0)
270710-002-44

Figure E.11. Initial Memory Image (IMI)

Instruction Address Breakpoint Registers (IPB0-IPB1)

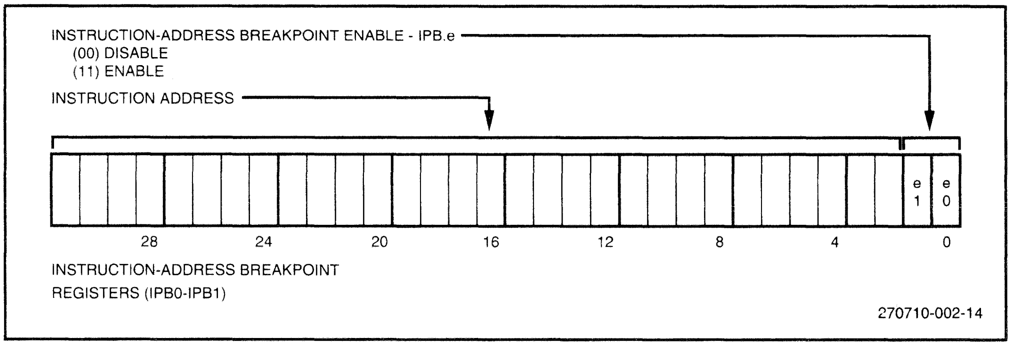

INSTRUCTION-ADDRESS BREAKPOINT ENABLE - IPB.e
(00) DISABLE
(11) ENABLE
INSTRUCTION ADDRESS
e1
e0
28
24
20
16
12
8
4
0
INSTRUCTION-ADDRESS BREAKPOINT REGISTERS (IPB0-IPB1)
270710-002-14

Figure E.12. Instruction Address Breakpoint Registers (IPB0 - IPB1)

E

Interrupt Control Register (ICON)

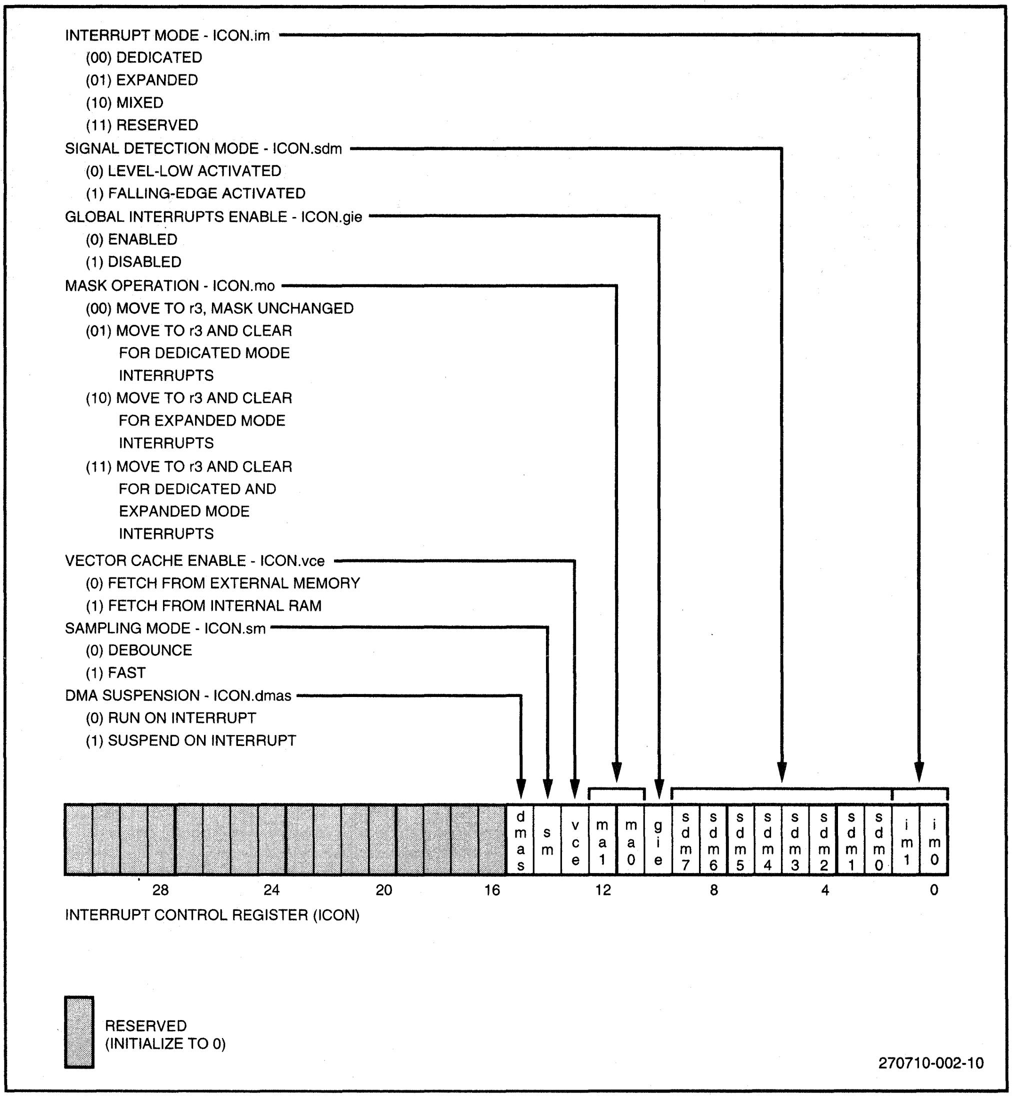

INTERRUPT MODE - ICON.im
(00) DEDICATED
(01) EXPANDED
(10) MIXED
(11) RESERVED
SIGNAL DETECTION MODE - ICON.sdm
(0) LEVEL-LOW ACTIVATED
(1) FALLING-EDGE ACTIVATED
GLOBAL INTERRUPTS ENABLE - ICON.gie
(0) ENABLED
(1) DISABLED
MASK OPERATION - ICON.mo
(00) MOVE TO r3, MASK UNCHANGED
(01) MOVE TO r3 AND CLEAR FOR DEDICATED MODE INTERRUPTS
(10) MOVE TO r3 AND CLEAR FOR EXPANDED MODE INTERRUPTS
(11) MOVE TO r3 AND CLEAR FOR DEDICATED AND EXPANDED MODE INTERRUPTS
VECTOR CACHE ENABLE - ICON.vce
(0) FETCH FROM EXTERNAL MEMORY
(1) FETCH FROM INTERNAL RAM
SAMPLING MODE - ICON.sm
(0) DEBOUNCE
(1) FAST
DMA SUSPENSION - ICON.dmas
(0) RUN ON INTERRUPT
(1) SUSPEND ON INTERRUPT
dmas
sm
vce
ma1
ma0
gie
sdm7
sdm6
sdm5
sdm4
sdm3
sdm2
sdm1
sdm0
im1
im0
28
24
20
16
12
8
4
0
INTERRUPT CONTROL REGISTER (ICON)
RESERVED (INITIALIZE TO 0)
270710-002-10

Figure E.13. Interrupt Control (ICON) Register

Interrupt Pending (IPND) (sf0) and Interrupt Mask Registers (IMSK) (sf1)

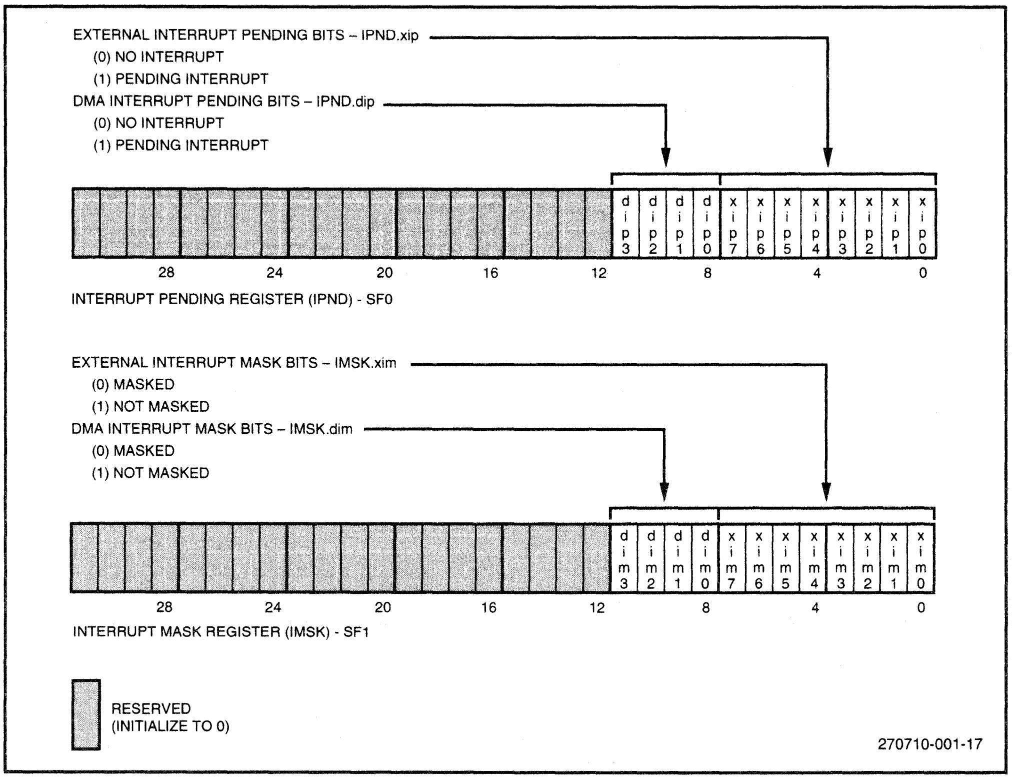

EXTERNAL INTERRUPT PENDING BITS – IPND.xip
(0) NO INTERRUPT
(1) PENDING INTERRUPT
DMA INTERRUPT PENDING BITS – IPND.dip
(0) NO INTERRUPT
(1) PENDING INTERRUPT
dip3 dip2 dip1 dip0 xip7 xip6 xip5 xip4 xip3 xip2 xip1 xip0
28 24 20 16 12 8 4 0
INTERRUPT PENDING REGISTER (IPND) - SF0
EXTERNAL INTERRUPT MASK BITS – IMSK.xim
(0) MASKED
(1) NOT MASKED
DMA INTERRUPT MASK BITS – IMSK.dim
(0) MASKED
(1) NOT MASKED
dim3 dim2 dim1 dim0 xim7 xim6 xim5 xim4 xim3 xim2 xim1 xim0
28 24 20 16 12 8 4 0
INTERRUPT MASK REGISTER (IMSK) - SF1
RESERVED
(INITIALIZE TO 0)
270710-001-17

Figure E.14. Interrupt Mask (IMSK) and Interrupt Pending (IPND) Registers

Interrupt Map Registers (IMAP0-IMAP2)

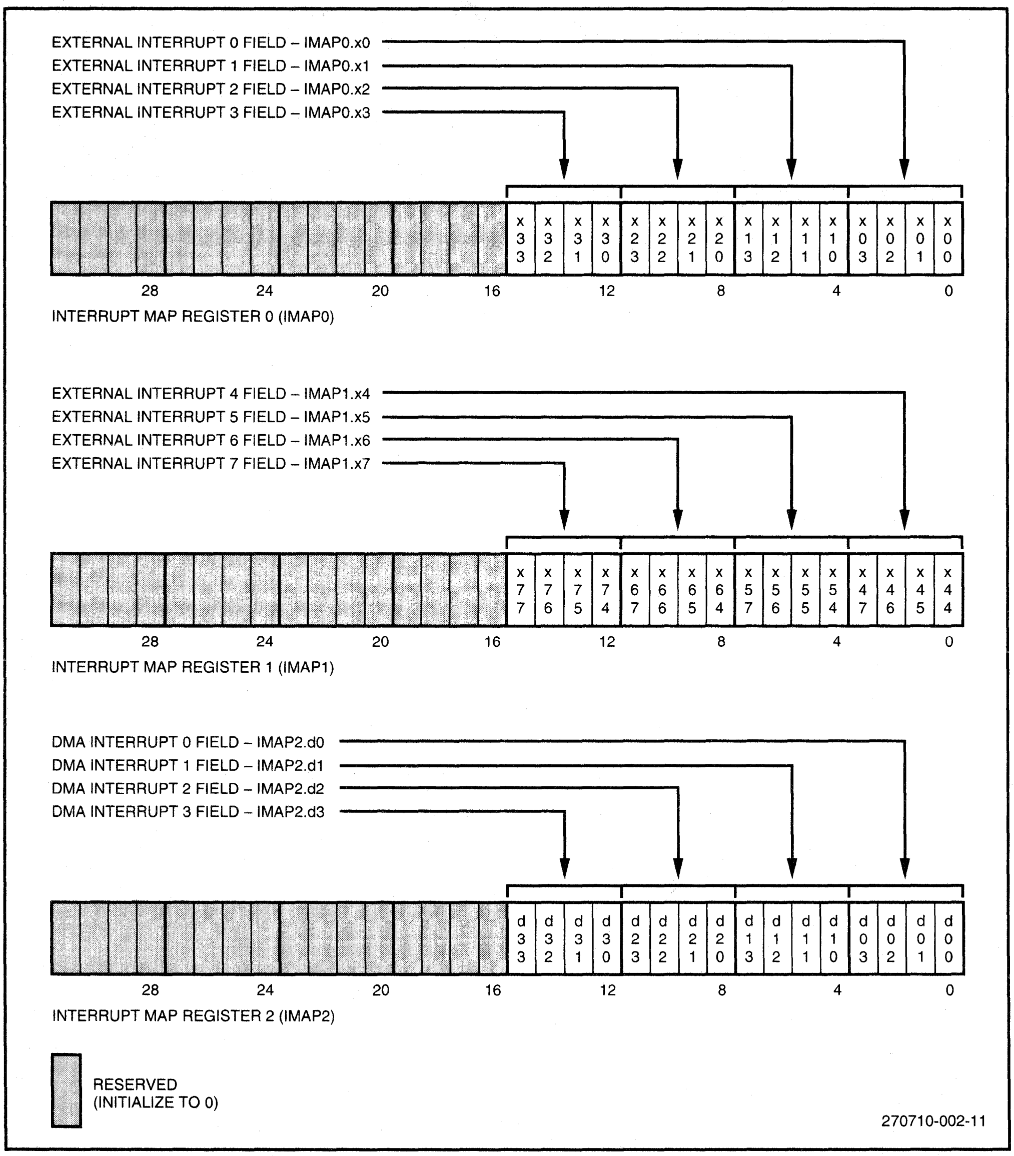

EXTERNAL INTERRUPT 0 FIELD – IMAP0.x0
EXTERNAL INTERRUPT 1 FIELD – IMAP0.x1
EXTERNAL INTERRUPT 2 FIELD – IMAP0.x2
EXTERNAL INTERRUPT 3 FIELD – IMAP0.x3
x33 x32 x31 x30 x23 x22 x21 x20 x13 x12 x11 x10 x03 x02 x01 x00
28 24 20 16 12 8 4 0
INTERRUPT MAP REGISTER 0 (IMAP0)
EXTERNAL INTERRUPT 4 FIELD – IMAP1.x4
EXTERNAL INTERRUPT 5 FIELD – IMAP1.x5
EXTERNAL INTERRUPT 6 FIELD – IMAP1.x6
EXTERNAL INTERRUPT 7 FIELD – IMAP1.x7
x77 x76 x75 x74 x67 x66 x65 x64 x57 x56 x55 x54 x47 x46 x45 x44
28 24 20 16 12 8 4 0
INTERRUPT MAP REGISTER 1 (IMAP1)
DMA INTERRUPT 0 FIELD – IMAP2.d0
DMA INTERRUPT 1 FIELD – IMAP2.d1
DMA INTERRUPT 2 FIELD – IMAP2.d2
DMA INTERRUPT 3 FIELD – IMAP2.d3
d33 d32 d31 d30 d23 d22 d21 d20 d13 d12 d11 d10 d03 d02 d01 d00
28 24 20 16 12 8 4 0
INTERRUPT MAP REGISTER 2 (IMAP2)
RESERVED
(INITIALIZE TO 0)
270710-002-11

Figure E.15. Interrupt Mapping (IMAP2-IMAP0) Registers

Interrupt Record

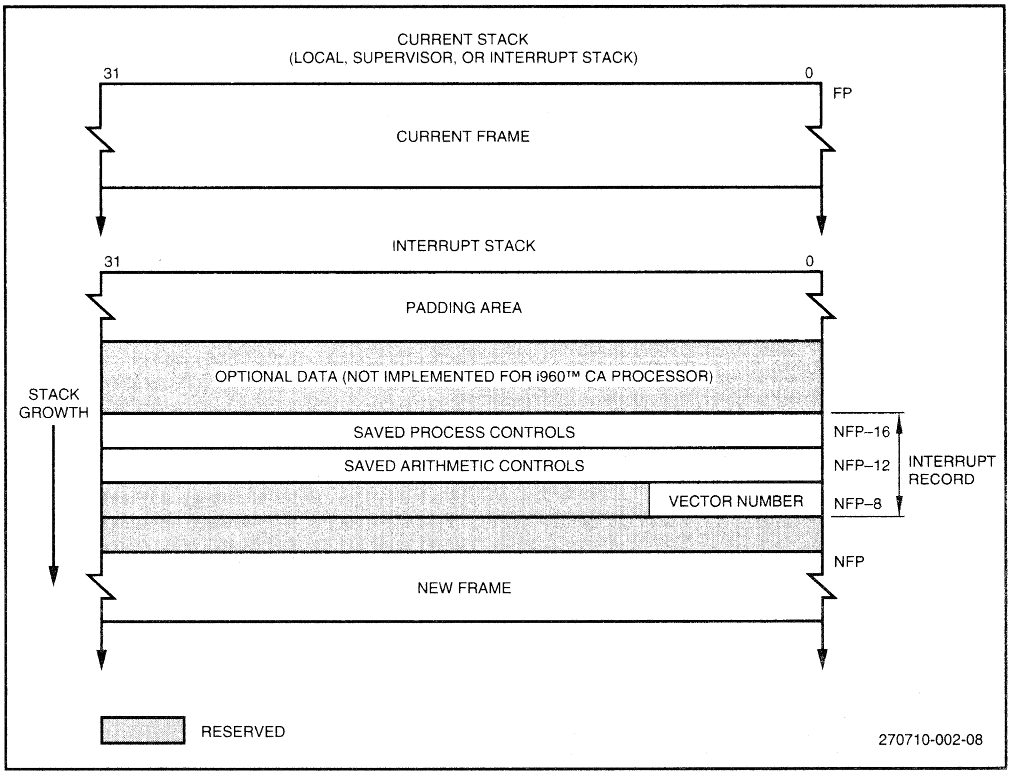


Figure E.16. Storage of an Interrupt Record on the Interrupt Stack

Interrupt Table

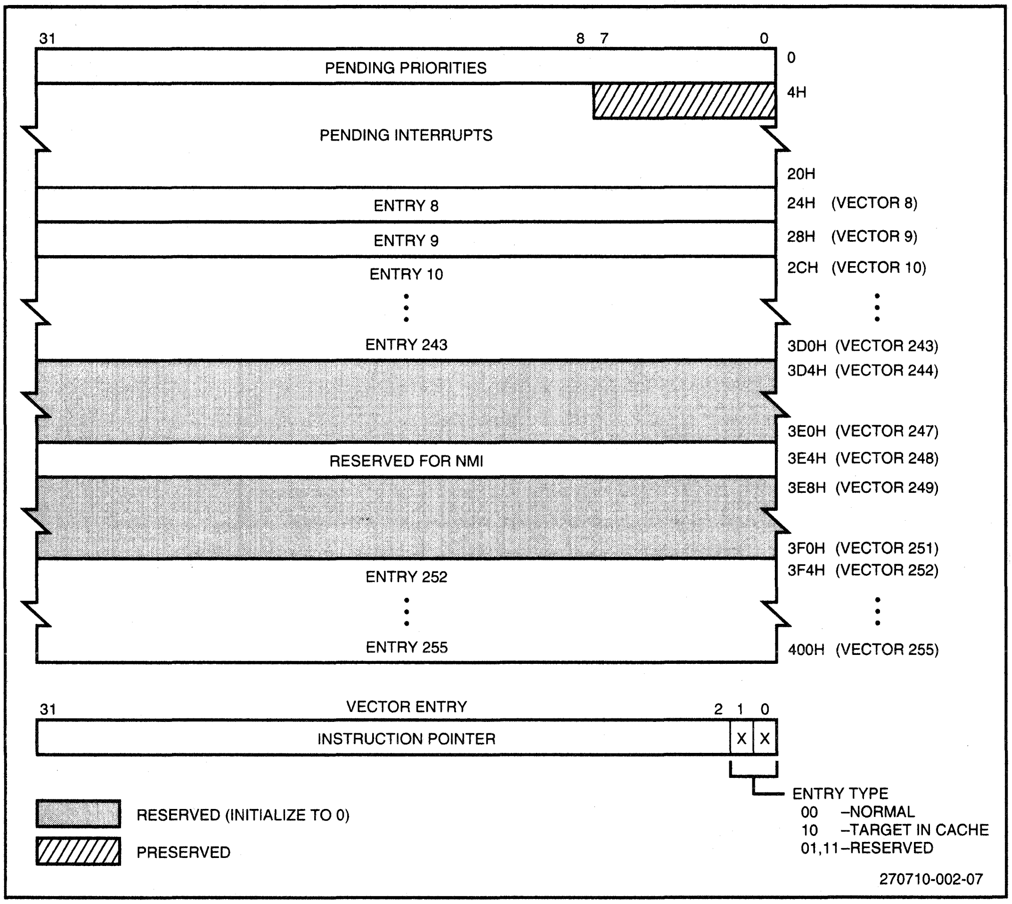


Figure E.17. Interrupt Table

Memory Region Configuration Registers (MCON0-MCON15)

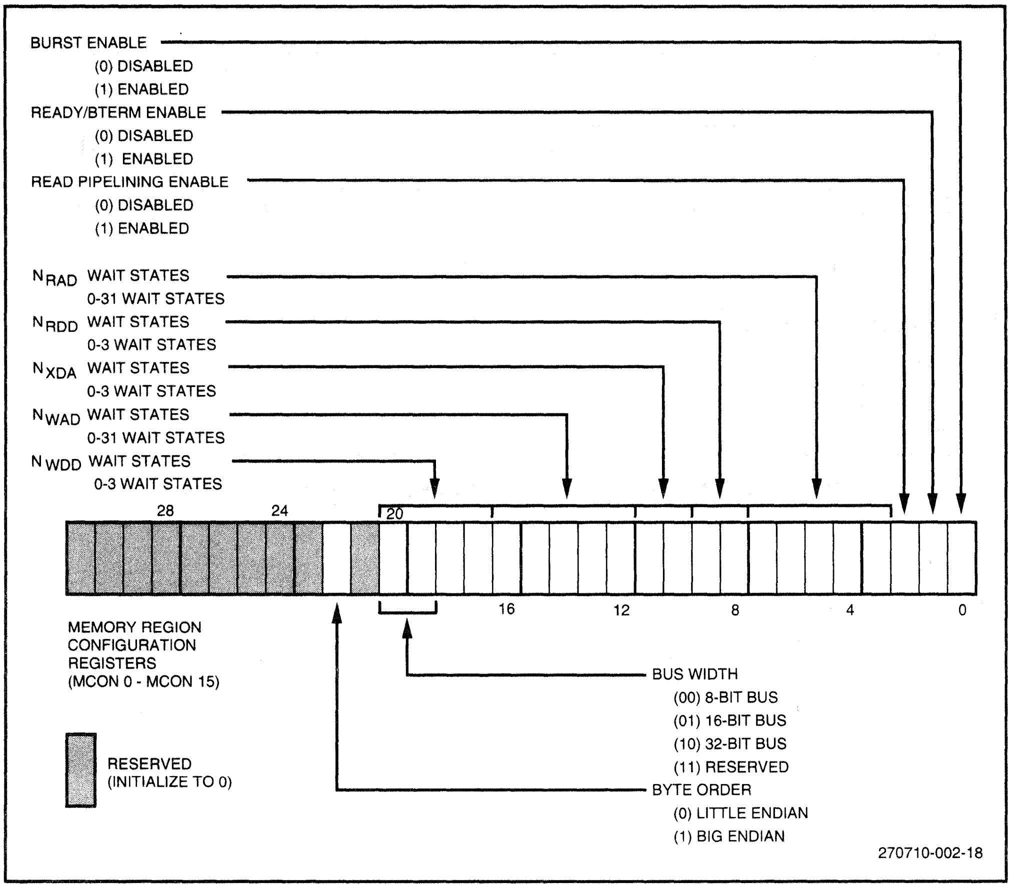


Figure E.18. Memory Region Configuration Register (MCON0-MCON15)

Previous Frame Pointer (PFP) (r0)

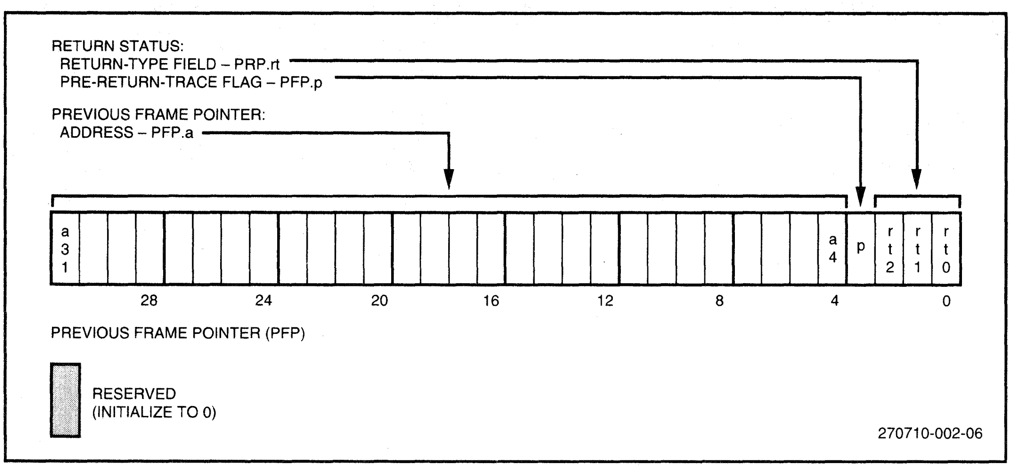

RETURN STATUS:
RETURN-TYPE FIELD – PRP.rt
PRE-RETURN-TRACE FLAG – PFP.p
PREVIOUS FRAME POINTER:
ADDRESS – PFP.a
a31
a4
p
rt2
rt1
rt0
28
24
20
16
12
8
4
0
PREVIOUS FRAME POINTER (PFP)
RESERVED (INITIALIZE TO 0)
270710-002-06

Figure E.19. Previous Frame Pointer Register (PFP) (r0)

Process Control Block Configuration Words (PRCB)

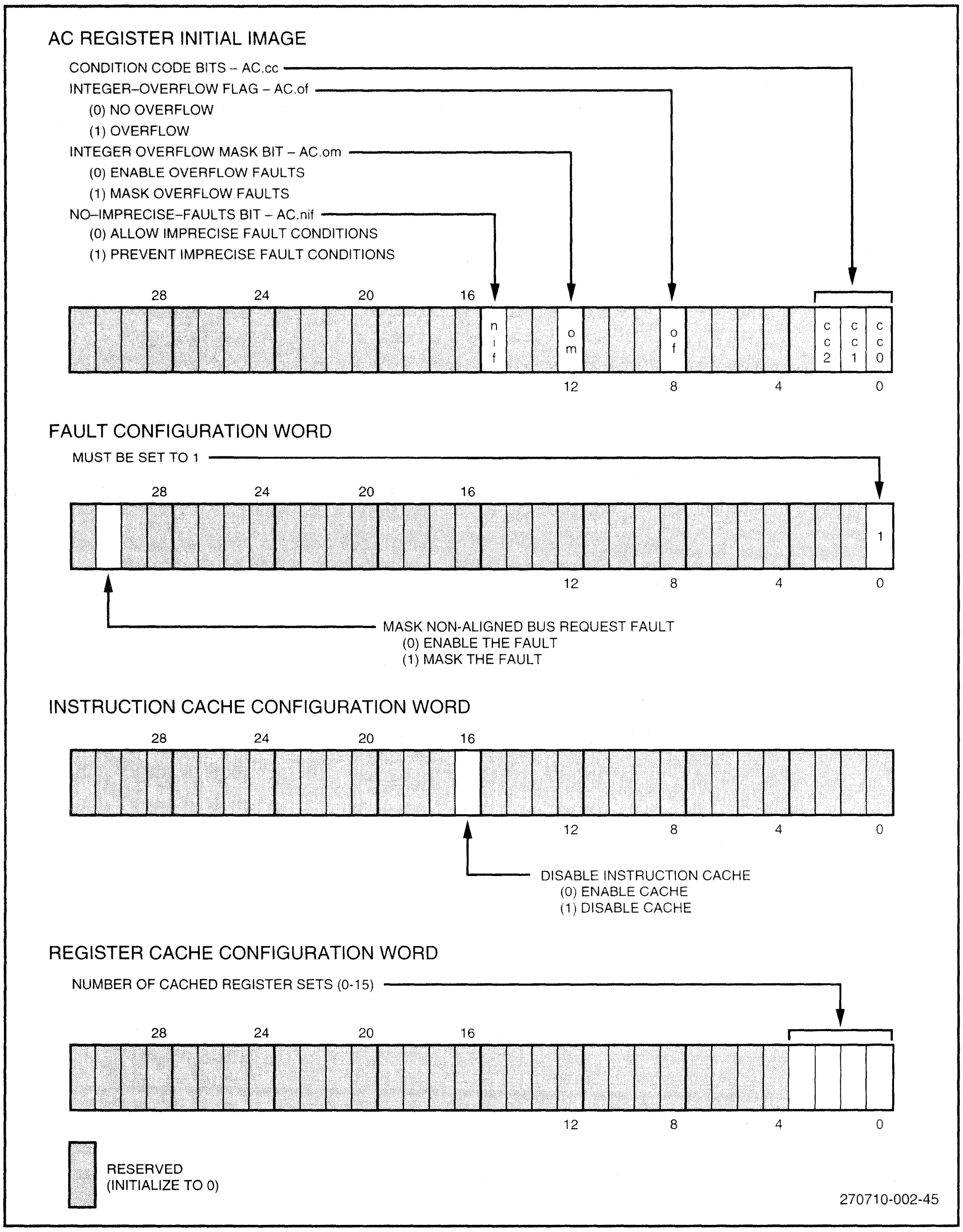

AC REGISTER INITIAL IMAGE
CONDITION CODE BITS – AC.cc
INTEGER–OVERFLOW FLAG – AC.of
(0) NO OVERFLOW
(1) OVERFLOW
INTEGER OVERFLOW MASK BIT – AC.om
(0) ENABLE OVERFLOW FAULTS
(1) MASK OVERFLOW FAULTS
NO–IMPRECISE–FAULTS BIT – AC.nif
(0) ALLOW IMPRECISE FAULT CONDITIONS
(1) PREVENT IMPRECISE FAULT CONDITIONS
28
24
20
16
nif
om
of
cc2
cc1
cc0
12
8
4
0
FAULT CONFIGURATION WORD
MUST BE SET TO 1
1
MASK NON-ALIGNED BUS REQUEST FAULT
(0) ENABLE THE FAULT
(1) MASK THE FAULT
INSTRUCTION CACHE CONFIGURATION WORD
DISABLE INSTRUCTION CACHE
(0) ENABLE CACHE
(1) DISABLE CACHE
REGISTER CACHE CONFIGURATION WORD
NUMBER OF CACHED REGISTER SETS (0-15)
RESERVED
(INITIALIZE TO 0)
270710-002-45

Figure E.20. Configuration Words in the PRCB

Process Controls Register (PC)

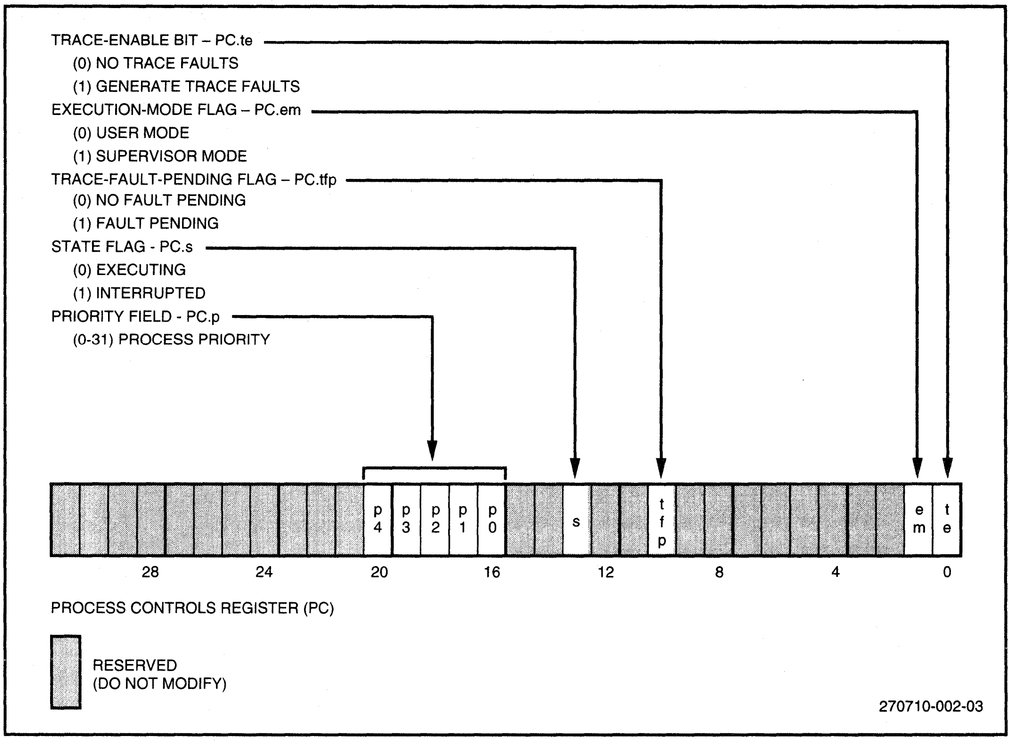


Figure E.21. Process Controls (PC) Register

System Procedure Table

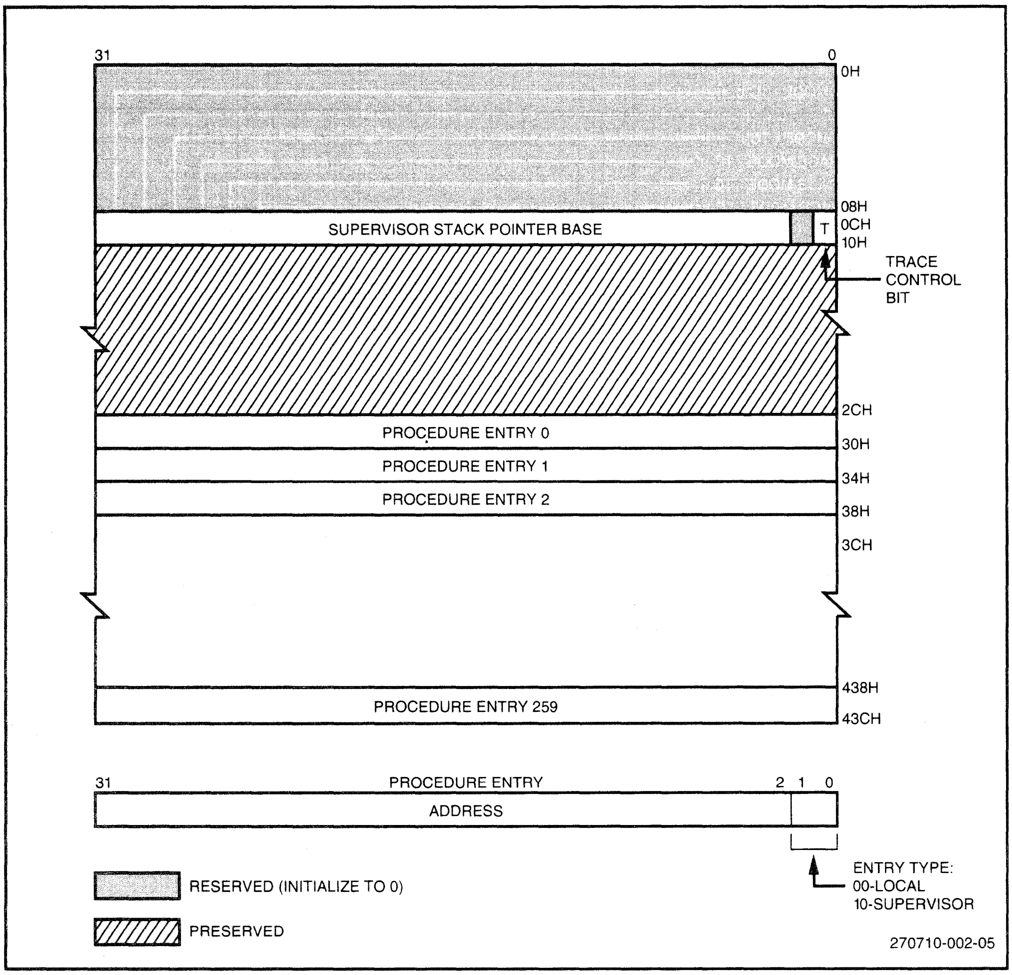


Figure E.22. System Procedure Table

Trace Controls Register (TC)

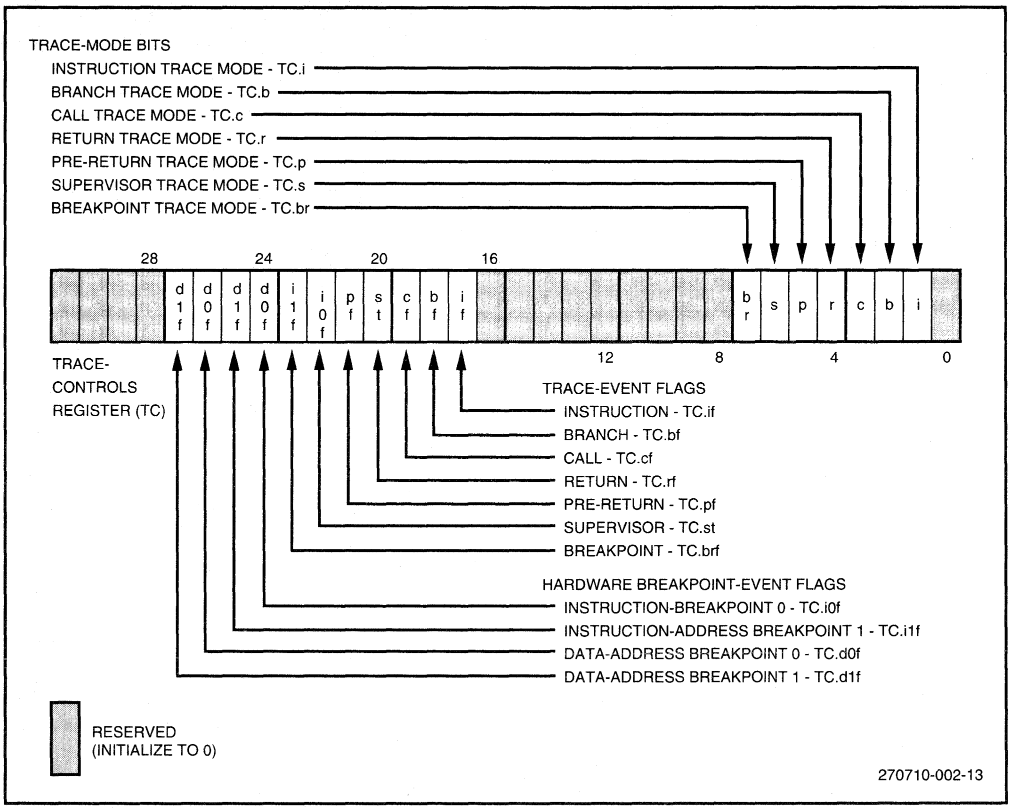

TRACE-MODE BITS
INSTRUCTION TRACE MODE - TC.i
BRANCH TRACE MODE - TC.b
CALL TRACE MODE - TC.c
RETURN TRACE MODE - TC.r
PRE-RETURN TRACE MODE - TC.p
SUPERVISOR TRACE MODE - TC.s
BREAKPOINT TRACE MODE - TC.br
28
24
20
16
12
8
4
0
d1f
d0f
d1f
d0f
i1f
i0f
pf
st
cf
bf
if
br
s
p
r
c
b
i
TRACE-CONTROLS REGISTER (TC)
TRACE-EVENT FLAGS
INSTRUCTION - TC.if
BRANCH - TC.bf
CALL - TC.cf
RETURN - TC.rf
PRE-RETURN - TC.pf
SUPERVISOR - TC.st
BREAKPOINT - TC.brf
HARDWARE BREAKPOINT-EVENT FLAGS
INSTRUCTION-BREAKPOINT 0 - TC.i0f
INSTRUCTION-ADDRESS BREAKPOINT 1 - TC.i1f
DATA-ADDRESS BREAKPOINT 0 - TC.d0f
DATA-ADDRESS BREAKPOINT 1 - TC.d1f
RESERVED (INITIALIZE TO 0)
270710-002-13

Figure E.23. Trace Controls (TC) Register

Glossary

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 *linear* – also called *flat* – 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.

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 which 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.

FP. Frame Pointer (see).

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. Stack Frame (see).

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.

IBR. Initialization Boot Record (see).

IMI. Initial Memory Image (see).

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 will be able to 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 i960 CA microprocessor. 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. An architecturally-defined 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.

Literals. A set of 32 ordinal values ranging from 0 to 31 (5 bits) that can be used as operands in certain instructions.

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 **call**, **callx** and **calls** 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.

NIF. No Imprecise Faults Bit (see).

NMI. Non Maskable Interrupt (see).

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. Process Control Block (see).

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 **syncf** 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 is able to 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.

SP. See Stack Pointer.

Special Function Registers (SFRs). A set of implementation-defined registers that represent an extension to the basic register set of the i960 architecture. They are intended to allow communication between the core processor and specially designed coprocessors. When special function registers are implemented, they can be used as operands in any instruction that accepts a global or local register as an operand.

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 *activation record*. 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 architecturally-defined registers – each 32-bits in length – 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 **calls** instruction) where the entry type of the called procedure is 10_2. 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 be in. The processor uses the supervisor stack when in supervisor mode. Also, while in supervisor mode, software is allowed to execute the **modpc** instruction and any other implementation-defined instructions that are designed to be supervisor mode instructions.

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 **calls** 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 **modpc** 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.

Index

INDEX

intel®

NORTH AMERICAN SALES OFFICES

ALABAMA

Intel Corp.
600 Boulevard South
Suite 104-L
Huntsville 35802
Tel: (205) 883-3507
FAX: (205) 883-3511

ARIZONA

†Intel Corp.
410 North 44th Street
Suite 500
Phoenix 85008
Tel: (602) 231-0386
FAX: (602) 244-0446

CALIFORNIA

†Intel Corp.
21515 Vanowen Street
Suite 116
Canoga Park 91303
Tel: (818) 704-8500
FAX: (818) 340-1144

Intel Corp.
1 Sierra Gate Plaza
Suite 280C
Roseville 95678
Tel: (916) 782-8086
FAX: (916) 782-8153

†Intel Corp.
9665 Chesapeake Dr.
Suite 325
San Diego 92123
Tel: (619) 292-8086
FAX: (619) 292-0628

*†Intel Corp.
400 N. Tustin Avenue
Suite 450
Santa Ana 92705
Tel: (714) 835-9642
TWX: 910-595-1114
FAX: (714) 541-9157

*†Intel Corp.
San Tomas 4
2700 San Tomas Expressway
2nd Floor
Santa Clara 95051
Tel: (408) 986-8086
TWX: 910-338-0255
FAX: (408) 727-2620

COLORADO

*†Intel Corp.
600 S. Cherry St.
Suite 700
Denver 80222
Tel: (303) 321-8086
TWX: 910-931-2289
FAX: (303) 322-8670

CONNECTICUT

†Intel Corp.
301 Lee Farm Corporate Park
83 Wooster Heights Rd.
Danbury 06810
Tel: (203) 748-3130
FAX: (203) 794-0339

FLORIDA

†Intel Corp.
800 Fairway Drive
Suite 160
Deerfield Beach 33441
Tel: (305) 421-0506
FAX: (305) 421-2444

†Intel Corp.
5850 T.G. Lee Blvd.
Suite 340
Orlando 32822
Tel: (407) 240-8000
FAX: (407) 240-8097

GEORGIA

†Intel Corp.
20 Technology Parkway
Suite 150
Norcross 30092
Tel: (404) 449-0541
FAX: (404) 605-9762

ILLINOIS

*†Intel Corp.
Woodfield Corp. Center III
300 N. Martingale Road
Suite 400
Schaumburg 60173
Tel: (708) 605-8031
FAX: (708) 706-9762

INDIANA

†Intel Corp.
8910 Purdue Road
Suite 350
Indianapolis 46268
Tel: (317) 875-0623
FAX: (317) 875-8938

MARYLAND

*†Intel Corp.
10010 Junction Dr.
Suite 200
Annapolis Junction 20701
Tel: (410) 206-2860
FAX: (410) 206-3678

MASSACHUSETTS

*†Intel Corp.
Westford Corp. Center
5 Carlisle Road
2nd Floor
Westford 01886
Tel: (508) 692-0960
TWX: 710-343-6333
FAX: (508) 692-7867

MICHIGAN

†Intel Corp.
7071 Orchard Lake Road
Suite 100
West Bloomfield 48322
Tel: (313) 851-8096
FAX: (313) 851-8770

MINNESOTA

†Intel Corp.
3500 W. 80th St.
Suite 360
Bloomington 55431
Tel: (612) 835-6722
TWX: 910-576-2867
FAX: (612) 831-6497

NEW JERSEY

*†Intel Corp.
Lincroft Office Center
125 Half Mile Road
Red Bank 07701
Tel: (908) 747-2233
FAX: (908) 747-0983

NEW YORK

*Intel Corp.
850 Crosskeys Office Park
Fairport 14450
Tel: (716) 425-2750
TWX: 510-253-7391
FAX: (716) 223-2561

*†Intel Corp.
2950 Express Dr., South
Suite 130
Islandia 11722
Tel: (516) 231-3300
TWX: 510-227-6236
FAX: (516) 348-7939

†Intel Corp.
300 Westage Business Center
Suite 230
Fishkill 12524
Tel: (914) 897-3860
FAX: (914) 897-3125

OHIO

*†Intel Corp.
3401 Park Center Drive
Suite 220
Dayton 45414
Tel: (513) 890-5350
TWX: 810-450-2528
FAX: (513) 890-8658

*†Intel Corp.
25700 Science Park Dr.
Suite 100
Beachwood 44122
Tel: (216) 464-2736
TWX: 810-427-9298
FAX: (804) 282-0673

OKLAHOMA

Intel Corp.
6801 N. Broadway
Suite 115
Oklahoma City 73162
Tel: (405) 848-8086
FAX: (405) 840-9819

OREGON

†Intel Corp.
15254 N.W. Greenbrier Pkwy.
Building B
Beaverton 97006
Tel: (503) 645-8051
TWX: 910-467-8741
FAX: (503) 645-8181

PENNSYLVANIA

*†Intel Corp.
925 Harvest Drive
Suite 200
Blue Bell 19422
Tel: (215) 641-1000
FAX: (215) 641-0785

*†Intel Corp.
400 Penn Center Blvd.
Suite 610
Pittsburgh 15235
Tel: (412) 823-4970
FAX: (412) 829-7578

PUERTO RICO

†Intel Corp.
South Industrial Park
P.O. Box 910
Las Piedras 00671
Tel: (809) 733-8616

SOUTH CAROLINA

Intel Corp.
100 Executive Center Drive
Suite 109, B183
Greenville 29615
Tel: (803) 297-8086
FAX: (803) 297-3401

TEXAS

†Intel Corp.
8911 N. Capital of Texas Hwy.
Suite 4230
Austin 78759
Tel: (512) 794-8086
FAX: (512) 338-9335

*†Intel Corp.
12000 Ford Road
Suite 400
Dallas 75234
Tel: (214) 241-8087
FAX: (214) 484-1180

*†Intel Corp.
7322 S.W. Freeway
Suite 1490
Houston 77074
Tel: (713) 988-8086
TWX: 910-881-2490
FAX: (713) 988-3660

UTAH

†Intel Corp.
428 East 6400 South
Suite 104
Murray 84107
Tel: (801) 263-8051
FAX: (801) 268-1457

WASHINGTON

†Intel Corp.
2800 156th Avenue S.E.
Suite 105
Bellevue 98008
Tel: (206) 643-8086
FAX: (206) 746-4495

Intel Corp.
408 N. Mullan Road
Suite 105
Spokane 99206
Tel: (509) 928-8086
FAX: (509) 928-9467

WISCONSIN

Intel Corp.
400 N. Executive Dr.
Suite 401
Brookfield 53005
Tel: (414) 789-2733
FAX: (414) 789-2746

CANADA

BRITISH COLUMBIA

Intel Semiconductor of Canada, Ltd.
999 Canada Place
Suite 404, #11
Vancouver V6C 3E2
Tel: (604) 844-2823
FAX: (604) 844-2813

ONTARIO

†Intel Semiconductor of Canada, Ltd.
2650 Queensview Drive
Suite 250
Ottawa K2B 8H6
Tel: (613) 829-9714
FAX: (613) 820-5936

†Intel Semiconductor of Canada, Ltd.
190 Attwell Drive
Suite 500
Rexdale M9W 6H8
Tel: (416) 675-2105
FAX: (416) 675-2438

QUEBEC

†Intel Semiconductor of Canada, Ltd.
1 Rue Holiday
Suite 115
Tour East
Pt. Claire H9R 5N3
Tel: (514) 694-9130
FAX: 514-694-0064

†Sales and Service Office
*Field Application Location

CG/SALE/041692

NORTH AMERICAN DISTRIBUTORS

ALABAMA

Arrow/Schweber Electronics
1015 Henderson Road
Huntsville 35806
Tel: (205) 837-6955
FAX: (205) 721-1581

Hamilton/Avnet
4960 Corporate Drive, #135
Huntsville 35805
Tel: (205) 837-7210
FAX: (205) 721-0356

MTI Systems Sales
4950 Corporate Dr., #120
Huntsville 35805
Tel: (205) 830-9526
FAX: (205) 830-9557

Pioneer Technologies Group
4835 University Square, #5
Huntsville 35805
Tel: (205) 837-9300
FAX: (205) 837-9358

ARIZONA

Arrow/Schweber Electronics
2415 W. Erie Drive
Tempe 85282
Tel: (602) 431-0030
FAX: (602) 252-9109

Avnet Computer
30 South McKemy Avenue
Chandler 85226
Tel: (602) 961-6460
FAX: (602) 961-4787

Hamilton/Avnet
30 South McKemy Avenue
Chandler 85226
Tel: (602) 961-6403
FAX: (602) 961-1331

Wyle Laboratories
4141 E. Raymond
Phoenix 85040
Tel: (602) 437-2088
FAX: (602) 437-2124

CALIFORNIA

Arrow Commercial Systems Group
1502 Crocker Avenue
Hayward 94544
Tel: (510) 489-5371
FAX: (510) 489-9393

Arrow Commercial Systems Group
14242 Chambers Road
Tustin 92680
Tel: (714) 544-0200
FAX: (714) 731-8438

Arrow/Schweber Electronics
26707 W. Agoura Road
Calabasas 91302
Tel: (818) 880-9686
FAX: (818) 772-8930

Arrow/Schweber Electronics
9511 Ridgehaven Court
San Diego 92123
Tel: (619) 565-4800
FAX: (619) 279-8062

rrow/Schweber Electronics
180 Murphy Avenue
San Jose 95131
el: (408) 441-9700
AX: (408) 453-4810

rrow/Schweber Electronics
961 Dow Avenue
ustin 92680
el: (714) 838-5422
AX: (714) 838-4151

vnet Computer
170 Pullman Street
osta Mesa 92626
el: (714) 641-4150
AX: (714) 641-4170

Avnet Computer
1361B West 190th Street
Gardena 90248
Tel: (800) 426-7999
FAX: (310) 327-5389

Avnet Computer
755 Sunrise Blvd., #150
Roseville 95661
Tel: (916) 781-2521
FAX: (916) 781-3819

Avnet Computer
1175 Bordeaux Drive, #A
Sunnyvale 94089
Tel: (408) 743-3304
FAX: (408) 743-3348

Hamilton/Avnet
3170 Pullman Street
Costa Mesa 92626
Tel: (714) 641-4100
FAX: (714) 754-6033

Hamilton/Avnet
1175 Bordeaux Drive, #A
Sunnyvale 94089
Tel: (408) 743-3300
FAX: (408) 745-6679

Hamilton/Avnet
4545 Viewridge Avenue
San Diego 92123
Tel: (619) 571-1900
FAX: (619) 571-8761

Hamilton/Avnet
21150 Califa St.
Woodland Hills 91367
Tel: (818) 594-0404
FAX: (818) 594-8234

Hamilton/Avnet
755 Sunrise Avenue, #150
Roseville 95661
Tel: (916) 925-2216
FAX: (916) 925-3478

Pioneer Technologies Group
134 Rio Robles
San Jose 95134
Tel: (408) 954-9100
FAX: (408) 954-9113

Wyle Laboratories
15360 Barranca Pkwy.
Irvine 92713
Tel: (714) 753-9953

Wyle Laboratories
2951 Sunrise Blvd., #175
Rancho Cordova 95742
Tel: (916) 638-5282
FAX: (916) 638-1491

Wyle Laboratories
9525 Chesapeake Drive
San Diego 92123
Tel: (619) 565-9171
FAX: (619) 365-0512

Wyle Laboratories
3000 Bowers Avenue
Santa Clara 95051
Tel: (408) 727-2500
FAX: (408) 727-5896

Wyle Laboratories
17872 Cowan Avenue
Irvine 92714
Tel: (714) 863-9953
FAX: (714) 263-0473

Wyle Laboratories
26010 Mureau Road, #150
Calabasas 91302
Tel: (818) 880-9000
FAX: (818) 880-5510

COLORADO

Arrow/Schweber Electronics
61 Inverness Dr. East, #105
Englewood 80112
Tel: (303) 799-0258
FAX: (303) 373-5760

Hamilton/Avnet
9605 Maroon Circle, #200
Englewood 80112
Tel: (303) 799-7800
FAX: (303) 799-7801

Wyle Laboratories
451 E. 124th Avenue
Thornton 80241
Tel: (303) 457-9953
FAX: (303) 457-4831

CONNECTICUT

Arrow/Schweber Electronics
12 Beaumont Road
Wallingford 06492
Tel: (203) 265-7741
FAX: (203) 265-7988

Avnet Computer
55 Federal Road, #103
Danbury 06810
Tel: (203) 797-2880
FAX: (203) 791-9050

Hamilton/Avnet
55 Federal Road, #103
Danbury 06810
Tel: (203) 743-6077
FAX: (203) 791-9050

Pioneer-Standard
2 Trap Falls Rd., #101
Shelton 06484
Tel: (203) 929-5600
FAX: (203) 838-9901

FLORIDA

Arrow/Schweber Electronics
400 Fairway Drive, #102
Deerfield Beach 33441
Tel: (305) 429-8200
FAX: (305) 428-3991

Arrow/Schweber Electronics
37 Skyline Drive, #3101
Lake Mary 32746
Tel: (407) 333-9300
FAX: (407) 333-9320

Avnet Computer
3343 W. Commercial Blvd.
Bldg. C/D, Suite 107
Ft. Lauderdale 33309
Tel: (305) 979-9067
FAX: (305) 730-0368

Avnet Computer
3247 Tech Drive North
St. Petersburg 33716
Tel: (813) 573-5524
FAX: (813) 572-4324

Hamilton/Avnet
5371 N.W. 33rd Avenue
Ft. Lauderdale 33309
Tel: (305) 484-5016
FAX: (305) 484-8369

Hamilton/Avnet
3247 Tech Drive North
St. Petersburg 33716
Tel: (813) 573-3930
FAX: (813) 572-4329

Hamilton/Avnet
7079 University Boulevard
Winter Park 32791
Tel: (407) 657-3300
FAX: (407) 678-1878

Pioneer Technologies Group
337 Northlake Blvd., #1000
Alta Monte Springs 32701
Tel: (407) 834-9090
FAX: (407) 834-0865

Pioneer Technologies Group
674 S. Military Trail
Deerfield Beach 33442
Tel: (305) 428-8877
FAX: (305) 481-2950

GEORGIA

Arrow Commercial Systems Group
3400 C. Corporate Way
Duluth 30136
Tel: (404) 623-8825
FAX: (404) 623-8802

Arrow/Schweber Electronics
4250 E. Rivergreen Pkwy., #E
Duluth 30136
Tel: (404) 497-1300
FAX: (404) 476-1493

Avnet Computer
3425 Corporate Way, #G
Duluth 30136
Tel: (404) 623-5452
FAX: (404) 476-0125

Hamilton/Avnet
3425 Corporate Way, #G
Duluth 30136
Tel: (404) 446-0611
FAX: (404) 446-1011

Pioneer Technologies Group
4250 C. Rivergreen Parkway
Duluth 30136
Tel: (404) 623-1003
FAX: (404) 623-0665

ILLINOIS

Arrow/Schweber Electronics
1140 W. Thorndale Rd.
Itasca 60143
Tel: (708) 250-0500

Avnet Computer
1124 Thorndale Avenue
Bensenville 60106
Tel: (708) 860-8573
FAX: (708) 773-7976

Hamilton/Avnet
1130 Thorndale Avenue
Bensenville 60106
Tel: (708) 860-7700
FAX: (708) 860-8530

MTI Systems
1140 W. Thorndale Avenue
Itasca 60143
Tel: (708) 250-8222
FAX: (708) 250-8275

Pioneer-Standard
2171 Executive Dr., #200
Addison 60101
Tel: (708) 495-9680
FAX: (708) 495-9831

INDIANA

Arrow/Schweber Electronics
7108 Lakeview Parkway West Dr.
Indianapolis 46268
Tel: (317) 299-2071
FAX: (317) 299-2379

Avnet Computer
485 Gradle Drive
Carmel 46032
Tel: (317) 575-8029
FAX: (317) 844-4964

Hamilton/Avnet
485 Gradle Drive
Carmel 46032
Tel: (317) 844-9333
FAX: (317) 844-5921

Pioneer-Standard
9350 Priority Way West Dr.
Indianapolis 46250
Tel: (317) 573-0880
FAX: (317) 573-0979

intel®

NORTH AMERICAN DISTRIBUTORS (Contd.)

IOWA

Hamilton/Avnet
2335A Blairsferry Rd., N.E.
Cedar Rapids 52402
Tel: (319) 362-4757
FAX: (319) 393-7050

KANSAS

Arrow/Schweber Electronics
9801 Legler Road
Lenexa 66219
Tel: (913) 541-9542
FAX: (913) 541-0328

Avnet Computer
15313 W. 95th Street
Lenexa 61219
Tel: (913) 541-7989
FAX: (913) 541-7904

Hamilton/Avnet
15313 W. 95th
Overland Park 66215
Tel: (913) 888-1055
FAX: (913) 541-7951

KENTUCKY

Hamilton/Avnet
805 A. Newtown Circle
Lexington 40511
Tel: (606) 259-1475
FAX: (606) 252-3238

MARYLAND

Arrow/Schweber Electronics
9800J Patuxent Woods Dr.
Columbia 21046
Tel: (301) 596-7800
FAX: (301) 995-6201

Avnet Computer
7172 Columbia Gateway Dr., #G
Columbia 21045
Tel: (301) 995-3571
FAX: (301) 995-3515

Hamilton/Avnet
7172 Columbia Gateway Dr., #F
Columbia 21045
Tel: (301) 995-3554
FAX: (301) 995-3515

*North Atlantic Industries
Systems Division
7125 River Wood Dr.
Columbia 21046
Tel: (301) 312-5800
FAX: (301) 290-7951

Pioneer Technologies Group
15810 Gaither Road
Gaithersburg 20877
Tel: (301) 921-0660
FAX: (301) 670-6746

MASSACHUSETTS

Arrow/Schweber Electronics
25 Upton Dr.
Wilmington 01887
Tel: (508) 658-0900
FAX: (508) 694-1754

Avnet Computer
10 D Centennial Drive
Peabody 01960
Tel: (508) 532-9886
FAX: (508) 532-9660

Hamilton/Avnet
10 D Centennial Drive
Peabody 01960
Tel: (508) 531-7430
FAX: (508) 532-9802

Pioneer-Standard
44 Hartwell Avenue
Lexington 02173
Tel: (617) 861-9200
FAX: (617) 863-1547

Wyle Laboratories
15 Third Avenue
Burlington 01803
Tel: (617) 272-7300
FAX: (617) 272-6809

MICHIGAN

Arrow/Schweber Electronics
19880 Haggerty Road
Livonia 48152
Tel: (800) 231-7902
FAX: (313) 462-2686

Avnet Computer
2876 28th Street, S.W., #5
Grandville 49418
Tel: (616) 531-9607
FAX: (616) 531-0059

Avnet Computer
41650 Garden Brook Rd. #120
Novi 48375
Tel: (313) 347-1820
FAX: (313) 347-4067

Hamilton/Avnet
2876 28th Street, S.W., #5
Grandville 49418
Tel: (616) 243-8805
FAX: (616) 531-0059

Hamilton/Avnet
41650 Garden Brook Rd., #100
Novi 48375
Tel: (313) 347-4270
FAX: (313) 347-4021

Pioneer-Standard
4505 Broadmoor S.E.
Grand Rapids 49512
Tel: (616) 698-1800
FAX: (616) 698-1831

Pioneer-Standard
13485 Stamford
Livonia 48150
Tel: (313) 525-1800
FAX: (313) 427-3720

MINNESOTA

Arrow/Schweber Electronics
10100 Viking Drive, #100
Eden Prairie 55344
Tel: (612) 941-5280
FAX: (612) 942-7803

Avnet Computer
10000 West 76th Street
Eden Prairie 55344
Tel: (612) 829-0025
FAX: (612) 944-2781

Hamilton/Avnet
12400 Whitewater Drive
Minnetonka 55343
Tel: (612) 932-0600
FAX: (612) 932-0613

Pioneer-Standard
7625 Golden Triange Dr., #G
Eden Prairie 55344
Tel: (612) 944-3355
FAX: (612) 944-3794

MISSOURI

Arrow/Schweber Electronics
2380 Schuetz Road
St. Louis 63141
Tel: (314) 567-6888
FAX: (314) 567-1164

Avnet Computer
739 Goddard Avenue
Chesterfield 63005
Tel: (314) 537-2725
FAX: (314) 537-4248

Hamilton/Avnet
741 Goddard
Chesterfield 63005
Tel: (314) 537-1600
FAX: (314) 537-4248

NEW HAMPSHIRE

Avnet Computer
2 Executive Park Drive
Bedford 03102
Tel: (800) 442-8638
FAX: (603) 624-2402

NEW JERSEY

Arrow/Schweber Electronics
4 East Stow Rd., Unit 11
Marlton 08053
Tel: (609) 596-8000
FAX: (609) 596-9632

Arrow/Schweber Electronics
43 Route 46 East
Pine Brook 07058
Tel: (201) 227-7880
FAX: (201) 538-4962

Avnet Computer
1-B Keystone Ave., Bldg. 36
Cherry Hill 08003
Tel: (609) 424-8961
FAX: (609) 751-2502

Hamilton/Avnet
1 Keystone Ave., Bldg. 36
Cherry Hill 08003
Tel: (609) 424-0110
FAX: (609) 751-2552

Hamilton/Avnet
10 Industrial
Fairfield 07006
Tel: (201) 575-3390
FAX: (201) 575-5839

MTI Systems Sales
6 Century Drive
Parsippany 07054
Tel: (201) 882-8780
FAX: (201) 539-6430

Pioneer-Standard
14-A Madison Rd.
Fairfield 07006
Tel: (201) 575-3510
FAX: (201) 575-3454

NEW MEXICO

Alliance Electronics, Inc.
10510 Research Avenue
Albuquerque 87123
Tel: (505) 292-3360
FAX: (505) 275-6392

Avnet Computer
7801 Academy Road
Bldg. 1, Suite 204
Albuquerque 87109
Tel: (505) 828-9725
FAX: (505) 828-0360

Hamilton/Avnet
7801 Academy Rd. N.E.
Bldg. 1, Suite 204
Albuquerque 87108
Tel: (505) 765-1500
FAX: (505) 243-1395

NEW YORK

Arrow/Schweber Electronics
3375 Brighton Henrietta Townline Rd.
Rochester 14623
Tel: (716) 427-0300
FAX: (716) 427-0735

Arrow/Schweber Electronics
20 Oser Avenue
Hauppauge 11788
Tel: (516) 231-1000
FAX: (516) 231-1072

Avnet Computer
933 Motor Parkway
Hauppauge 11788
Tel: (516) 434-7443
FAX: (516) 434-7426

Avnet Computer
2060 Townline Rd.
Rochester 14623
Tel: (716) 272-9110
FAX: (716) 272-9685

Hamilton/Avnet
933 Motor Parkway
Hauppauge 11788
Tel: (516) 231-9800
FAX: (516) 434-7426

Hamilton/Avnet
2060 Townline Rd.
Rochester 14623
Tel: (716) 292-0730
FAX: (716) 292-0810

Hamilton/Avnet
103 Twin Oaks Drive
Syracuse 13120
Tel: (315) 437-2641
FAX: (315) 432-0740

MTI Systems
1 Penn Plaza
250 W. 34th Street
New York 10119
Tel: (212) 643-1280
FAX: (212) 643-1288

Pioneer-Standard
68 Corporate Drive
Binghamton 13904
Tel: (607) 722-9300
FAX: (607) 722-9562

Pioneer-Standard
60 Crossway Park West
Woodbury, Long Island 11797
Tel: (516) 921-8700
FAX: (516) 921-2143

Pioneer-Standard
840 Fairport Park
Fairport 14450
Tel: (716) 381-7070
FAX: (716) 381-5955

NORTH CAROLINA

Arrow/Schweber Electronics
5240 Greensdairy Road
Raleigh 27604
Tel: (919) 876-3132
FAX: (919) 878-9517

Avnet Computer
2725 Millbrook Rd., #123
Raleigh 27604
Tel: (919) 790-1735
FAX: (919) 872-4972

Hamilton/Avnet
5250-77 Center Dr. #350
Charlotte 28217
Tel: (704) 527-2485
FAX: (704) 527-8058

Hamilton/Avnet
3510 Spring Forest Drive
Raleigh 27604
Tel: (919) 878-0819

Pioneer Technologies Group
9401 L-Southern Pine Blvd.
Charlotte 28210
Tel: (704) 527-8188
FAX: (704) 522-8564

Pioneer Technologies Group
2810 Meridian Parkway, #148
Durham 27713
Tel: (919) 544-5400
FAX: (919) 544-5885

OHIO

Arrow Commercial Systems Group
284 Cramer Creek Court
Dublin 43017
Tel: (614) 889-9347
FAX: (614) 889-9680

Arrow/Schweber Electronics
6573 Cochran Road, #E
Solon 44139
Tel: (216) 248-3990
FAX: (216) 248-1106

Arrow/Schweber Electronics
8200 Washington Village Dr.
Centerville 45458
Tel: (513) 435-5563
FAX: (513) 435-2049

*Self Certified Small Business per Federal Acquisition Regulations

intel®

NORTH AMERICAN DISTRIBUTORS (Contd.)

OHIO (Contd.)

Avnet Computer
7764 Washington Village Dr.
Dayton 45459
Tel: (513) 439-6756
FAX: (513) 439-6719

Avnet Computer
30325 Bainbridge Rd., Bldg. A
Solon 44139
Tel: (216) 349-2505
FAX: (216) 349-1894

Hamilton/Avnet
7760 Washington Village Dr.
Dayton 45459
Tel: (513) 439-6733
FAX: (513) 439-6711

Hamilton/Avnet
30325 Bainbridge
Solon 44139
Tel: (216) 349-4910
FAX: (216) 349-1894

Hamilton/Avnet
2600 Corp Exchange Drive, #180
Columbus 43231
Tel: (614) 882-7004
FAX: (614) 882-8650

MTI Systems Sales
23404 Commerce Park Rd.
Beachwood 44122
Tel: (216) 464-6688
FAX: (216) 464-3564

Pioneer-Standard
4433 Interpoint Boulevard
Dayton 45424
Tel: (513) 236-9900
FAX: (513) 236-8133

Pioneer-Standard
4800 E. 131st Street
Cleveland 44105
Tel: (216) 587-3600
FAX: (216) 663-1004

OKLAHOMA

Arrow/Schweber Electronics
12111 East 51st Street, #101
Tulsa 74146
Tel: (918) 252-7537
FAX: (918) 254-0917

Hamilton/Avnet
12121 E. 51st St., #102A
Tulsa 74146
Tel: (918) 252-7297
FAX: (918) 250-8763

OREGON

Almac/Arrow Electronics
1885 N.W. 169th Place
Beaverton 97006
Tel: (503) 629-8090
FAX: (503) 645-0611

Avnet Computer
9409 Southwest Nimbus Ave.
Beaverton 97005
Tel: (503) 627-0900
FAX: (503) 526-6242

Hamilton/Avnet
9750 Southwest Nimbus Ave.
Beaverton 97005
Tel: (503) 627-0201
FAX: (503) 641-4012

Wyle Laboratories
9640 Sunshine Court
Bldg. G, Suite 200
Beaverton 97005
Tel: (503) 643-7900
FAX: (503) 646-5466

PENNSYLVANIA

Avnet Computer
213 Executive Drive, #320
Mars 16046
Tel: (412) 772-1888
FAX: (412) 772-1890

Hamilton/Avnet
213 Executive, #320
Mars 16045
Tel: (412) 281-4152
FAX: (412) 772-1890

Pioneer-Standard
259 Kappa Drive
Pittsburgh 15238
Tel: (412) 782-2300
FAX: (412) 963-8255

Pioneer Technologies Group
500 Enterprise Road
Keith Valley Business Center
Horsham 19044
Tel: (215) 674-4000
FAX: (215) 674-3107

TEXAS

Arrow/Schweber Electronics
3220 Commander Drive
Carrollton 75006
Tel: (214) 380-6464
FAX: (214) 248-7208

Avnet Computer
4004 Beltline, Suite 200
Dallas 75244
Tel: (214) 308-8181
FAX: (214) 308-8129

Avnet Computer
1235 North Loop West, #525
Houston 77008
Tel: (713) 867-7500
FAX: (713) 861-6851

Hamilton/Avnet
1826-F Kramer Lane
Austin 78758
Tel: (512) 832-4306
FAX: (512) 832-4315

Hamilton/Avnet
4004 Beltline, Suite 200
Dallas 75244
Tel: (214) 308-8111
FAX: (214) 308-8109

Hamilton/Avnet
1235 North Loop West, #521
Houston 77008
Tel: (713) 240-7733
FAX: (713) 861-6541

Pioneer-Standard
1826-D Kramer Lane
Austin 78758
Tel: (512) 835-4000
FAX: (512) 835-9829

Pioneer-Standard
13765 Beta Road
Dallas 75244
Tel: (214) 263-3168
FAX: (214) 490-6419

Pioneer-Standard
10530 Rockley Road, #100
Houston 77099
Tel: (713) 495-4700
FAX: (713) 495-5642

Wyle Laboratories
1810 Greenville Avenue
Richardson 75081
Tel: (214) 235-9953
FAX: (214) 644-5064

Wyle Laboratories
4030 West Braker Lane, #330
Austin 78758
Tel: (512) 345-8853
FAX: (512) 345-9330

Wyle Laboratories
11001 South Wilcrest, #100
Houston 77099
Tel: (713) 879-9953
FAX: (713) 879-6540

UTAH

Arrow/Schweber Electronics
1946 W. Parkway Blvd.
Salt Lake City 84119
Tel: (801) 973-6913

Avnet Computer
1100 E. 6600 South, #150
Salt Lake City 84121
Tel: (801) 266-1115
FAX: (801) 266-0362

Hamilton/Avnet
1100 East 6600 South, #120
Salt Lake City 84121
Tel: (801) 972-2800
FAX: (801) 263-0104

Wyle Laboratories
1325 West 2200 South, #E
West Valley 84119
Tel: (801) 974-9953
FAX: (801) 972-2524

WASHINGTON

Almac/Arrow Electronics
14360 S.E. Eastgate Way
Bellevue 98007
Tel: (206) 643-9992
FAX: (206) 643-9709

Hamilton/Avnet
17761 N.E. 78th Place, #C
Redmond 98052
Tel: (206) 241-8555
FAX: (206) 241-5472

Avnet Computer
17761 Northeast 78th Place
Redmond 98052
Tel: (206) 867-0160
FAX: (206) 867-0161

Wyle Laboratories
15385 N.E. 90th Street
Redmond 98052
Tel: (206) 881-1150
FAX: (206) 881-1567

WISCONSIN

Arrow/Schweber Electronics
200 N. Patrick Blvd., #100
Brookfield 53005
Tel: (414) 792-0150
FAX: (414) 792-0156

Avnet Computer
20875 Crossroads Circle, #400
Waukesha 53186
Tel: (414) 784-8205
FAX: (414) 784-6006

Hamilton/Avnet
28875 Crossroads Circle, #400
Waukesha 53186
Tel: (414) 784-4510
FAX: (414) 784-9509

Pioneer-Standard
120 Bishops Way #163
Brookfield 53005
Tel: (414) 784-3480

ALASKA

Avnet Computer
1400 West Benson Blvd., #400
Anchorage 99503
Tel: (907) 274-9899
FAX: (907) 277-2639

CANADA

ALBERTA

Avnet Computer
2816 21st Street Northeast
Calgary T2E 6Z2
Tel: (403) 291-3284
FAX: (403) 250-1591

Zentronics
6815 8th Street N.E., #100
Calgary T2E 7H
Tel: (403) 295-8838
FAX: (403) 295-8714

BRITISH COLUMBIA

Almac-Arrow Electronics
8544 Baxter Place
Burnaby V5A 4T8
Tel: (604) 421-2333
FAX: (604) 421-5030

Hamilton/Avnet
8610 Commerce Court
Burnaby V5A 4N6
Tel: (604) 420-4101
FAX: (604) 420-5376

Zentronics
11400 Bridgeport Rd., #108
Richmond V6X 1T2
Tel: (604) 273-5575
FAX: (604) 273-2413

ONTARIO

Arrow/Schweber Electronics
36 Antares Dr., Unit 100
Nepean K2E 7W5
Tel: (613) 226-6903
FAX: (613) 723-2018

Arrow/Schweber Electronics
1093 Meyerside, Unit 2
Mississauga L5T 1M4
Tel: (416) 670-7769
FAX: (416) 670-7781

Avnet Computer
151 Superior Blvd.
Mississuaga L5T 2L1
Tel: (416) 795-3835

Avnet Computer
190 Colonade Road
Nepean K2E 7J5
Tel: (613) 727-2000
FAX: (613) 226-1184

Hamilton/Avnet
151 Superior Blvd., Units 1–6
Mississauga L5T 2L1
Tel: (416) 564-6060
FAX: (416) 564-6033

Hamilton/Avnet
190 Colonade Road
Nepean K2E 7J5
Tel: (613) 226-1700
FAX: (613) 226-1184

Zentronics
1355 Meyerside Drive
Mississauga L5T 1C9
Tel: (416) 564-9600
FAX: (416) 564-3127

Zentronics
155 Colonade Rd., South
Unit 17
Nepean K2E 7K1
Tel: (613) 226-8840
FAX: (613) 226-6352

QUEBEC

Arrow/Schweber Electronics
1100 St. Regis Blvd.
Dorval H9P 2T5
Tel: (514) 421-7411
FAX: (514) 421-7430

Arrow/Schweber Electronics
500 Boul. St-Jean-Baptiste Ave.
Quebec H2E 5R9
Tel: (418) 871-7500
FAX: (418) 871-6816

Avnet Computer
2795 Rue Halpern
St. Laurent H4S 1P8
Tel: (514) 335-2483
FAX: (514) 335-2481

Hamilton/Avnet
2795 Halpern
St. Laurent H4S 1P8
Tel: (514) 335-1000
FAX: (514) 335-2481

Zentronics
520 McCaffrey
St. Laurent H4T 1N3
Tel: (514) 737-9700
FAX: (514) 737-5212

intel®

EUROPEAN SALES OFFICES

FINLAND

Intel Finland OY
Ruosilantie 2
00390 Helsinki
Tel: (358) 0 544 644
FAX: (358) 0 544 030

FRANCE

Intel Corporation S.A.R.L.
1, Rue Edison-BP 303
78054 St. Quentin-en-Yvelines Cedex
Tel: (33) (1) 30 57 70 00
FAX: (33) (1) 30 64 60 32

GERMANY

Intel GmbH
Dornacher Strasse 1
8016 Feldkirchen bei Muenchen
Tel: (49) 089/90992-0
FAX: (49) 089/9043948

ISRAEL

Intel Semiconductor Ltd.
Atidim Industrial Park-Neve Sharet
P.O. Box 43202
Tel-Aviv 61430
Tel: (972) 03 498080
FAX: (972) 03 491870

ITALY

Intel Corporation Italia S.p.A.
Milanofiori Palazzo E
20094 Assago
Milano
Tel: (39) (02) 89200950
FAX: (39) (2) 3498464

NETHERLANDS

Intel Semiconductor B.V.
Postbus 84130
3009 CC Rotterdam
Tel: (31) 10 407 11 11
FAX: (31) 10 455 4688

SPAIN

Intel Iberia S.A.
Zubaran, 28
28010 Madrid
Tel: (34) 308 25 52
FAX: (34) 410 7570

SWEDEN

Intel Sweden A.B.
Dalvagen 24
171 36 Solna
Tel: (46) 8 734 01 00
FAX: (46) 8 278085

UNITED KINGDOM

Intel Corporation (U.K.) Ltd.
Pipers Way
Swindon, Wiltshire SN3 1RJ
Tel: (44) (0793) 696000
FAX: (44) (0793) 641440

EUROPEAN DISTRIBUTORS/REPRESENTATIVES

AUSTRIA

Bacher Electronics GmbH
Rotenmuehlgasse 26
A-1120 Wien
Tel: 43 222 81356460
FAX: 43 222 834276

BELGIUM

Inelco Belgium S.A.
Oorlogskruisenlaan 94
B-1120 Bruxelles
Tel: 32 2 244 2811
FAX: 32 2 216 4301

FRANCE

Almex
48, Rue de l'Aubepine
B.P. 102
92164 Antony Cedex
Tel: 33 1 4096 5400
FAX: 33 1 4666 6028

Lex Electronics
Silic 585
60 Rue des Gemeaux
94663 Rungis Cedex
Tel: 33 1 4978 4978
FAX: 33 1 4978 0596

Metrologie
Tour d'Asnieres
4, Avenue Laurent Cely
92606 Asnieres Cedex
Tel: 33 1 4790 6240
FAX: 33 1 4790 5947

Tekelec-Airtronic
Cite Des Bruyeres
Rue Carle Vernet
BP 2
92310 Sevres
Tel: 33 1 4623 2425
FAX: 33 1 4507 2191

GERMANY

E2000 Vertriebs-AG
Stahlgruberring 12
8000 Muenchen 82
Tel: 49 89 420010
FAX: 49 89 42001209

Jermyn GmbH
Im Dachsstueck 9
6250 Limburg
Tel: 49 6431 5080
FAX: 49 6431 508289

Metrologie GmbH
Steinerstrasse 15
8000 Muenchen 70
Tel: 49 89 724470
FAX: 49 89 72447111

Proelectron Vertriebs GmbH
Max-Planck-Strasse 1-3
6072 Dreieich
Tel: 49 6103 304343
FAX: 49 6103 304425

Rein Electronik GmbH
Loetscher Weg 66
4054 Nettetal 1
Tel: 49 2153 7330
FAX: 49 2153 733513

GREECE

Pouliadis Associates Corp.
5 Koumbari Street
Kolonaki Square
10674 Athens
Tel: 30 1 360 3741
FAX: 30 1 360 7501

IRELAND

Micro Marketing
Tany Hall
Eglinton Terrace
Dundrum
Dublin
Tel: 0001 989 400
FAX: 0001 989 8282

ISRAEL

Eastronics Ltd.
Rozanis 11
P.O.B. 39300
Tel Baruch
Tel-Aviv 61392
Tel: 972 3 475151
FAX: 972 3 475125

ITALY

Celdis Spa
Via F.11i Gracchi 36
20092 Cinisello Balsamo
Milano
Tel: 39 2 66012003
FAX: 39 2 6182433

Intesi Div. Della Deutsche
Divisione ITT
Industries GmbH
P.I. 06550110156
Milanofiori Palazzo E5
20094 Assago (Milano)
Tel: 39 2 824701
FAX: 39 2 8242631

Lasi Elettronica S.p.A.
P.I. 00839000155
Viale Fulvio Testi, N.280
20126 Milano
Tel: 39 2 66101370
FAX: 39 2 66101385

Telcom s.r.l.—Divisione MDS
Via Trombetta
Zona Marconi
Strada Cassanese
Segrate—Milano
Tel: 39 2 2138010
FAX: 39 2 216061

NETHERLANDS

Koning en Hartman B.V.
Energieweg 1
2627 AP Delft
Tel: 31 15 609 906
FAX: 31 15 619 194

PORTUGAL

ATD Electronica LDA
Rua Dr. Faria de
Vasconcelos, 3a
1900 Lisboa
Tel: 351 1 8472200
FAX: 351 1 8472197

SPAIN

ATD Electronica SA
Avda de la Industria, 32
Nave 17, 2B
28100 Alcobendas
Madrid
Tel: 1 661 65 51
FAX: 1 661 63 00

Metrologia Iberica
Avenida de la Industria NR 32-2o
Oficina 17
28100 Alcobendas
Madrid
Tel: (1) 661 11 42
FAX: (1) 661 57 55

SCANDINAVIA

OY Fintronic AB
Heikkilantie 2a
SF-00210 Helsinki
Tel: 358 0 6926022
FAX: 358 0 6821251

ITT Multikomponent A/S
Naverland 29
DK-2600 Glostrup
Denmark
Tel: 010 45 42 451822
FAX: 010 45 42 457624

Nordisk Elektronik A/S
Postboks 122
Smedsvingen 4
N-1364 Hvalstad
Norway
Tel: 47 2 846210
FAX: 47 2 846545

Nordisk Electronik AB
Box 36
Torshamnsgatan 39
S-16493 Kista
Sweden
Tel: 46 8 7034630
FAX: 46 8 7039845

SWITZERLAND

Industrade A.G.
Hertistrasse 31
CH-8304 Wallisellen
Tel: 41 1 8328111
FAX: 41 1 8307550

TURKEY

EMPA
80050 Sishane
Refik Saydam Cad No. 89/5
Istanbul
Tel: 90 1 143 6212
FAX: 90 1 143 6547

UNITED KINGDOM

Access Elect Comp Ltd.
Jubilee House
Jubilee Road
Letchworth
Hertfordshire
SG6 1QH
Tel: 0462 480888
FAX: 0462 682467

Bytech Components Ltd.
12a Cedarwood
Chineham Business Park
Crockford Lane
Basingstoke
Hants RG12 1RW
Tel: 0256 707107
FAX: 0256 707162

Bytech Systems
Unit 3
The Western Centre
Western Road
Bracknell
Berks RG12 1RW
Tel: 0344 55333
FAX: 0344 867270

Metrologie
Rapid House
Oxford Road
High Wycombe
Bucks
Herts HP11 2EE
Tel: 0494 474147
FAX: 0494 452144

Jermyn
Vestry Estate
Otford Road
Sevenoaks
Kent TN14 5EU
Tel: 0732 450144
FAX: 0732 451251

MMD
3 Bennet Court
Bennet Road
Reading
Berkshire RG2 0QX
Tel: 0734 313232
FAX: 0734 313255

Rapid Silicon
3 Bennet Court
Bennet Road
Reading
Berks RG2 0QX
Tel: 0734 752266
FAX: 0734 312728

Metro Systems
Rapid House
Oxford Road
High Wycombe
Bucks HP11 2EE
Tel: 0494 474171
FAX: 0494 21860

YUGOSLAVIA

H.R. Microelectronics Corp.
2005 de la Cruz Blvd.
Suite 220
Santa Clara, CA 95050
U.S.A.
Tel: (408) 988-0286
FAX: (408) 988-0306

intel®

INTERNATIONAL SALES OFFICES

AUSTRALIA

Intel Australia Pty. Ltd.
Unit 13
Allambie Grove Business Park
25 Frenchs Forest Road East
Frenchs Forest, NSW, 2086
Sydney
Tel: 61-2-975-3300
FAX: 61-2-975-3375

Intel Australia Pty. Ltd.
711 High Street
1st Floor
East Kw. Vic., 3102
Melbourne
Tel: 61-3-810-2141
FAX: 61-3-819 7200

BRAZIL

Intel Semiconductores do Brazil LTDA
Avenida Paulista, 1159-CJS 404/405
CEP 01311 - Sao Paulo - S.P.
Tel: 55-11-287-5899
TLX: 11-37-557-ISDB
FAX: 55-11-287-5119

CHINA/HONG KONG

Intel PRC Corporation
15/F, Office 1, Citic Bldg.
Jian Guo Men Wai Street
Beijing, PRC
Tel: (1) 500-4850
TLX: 22947 INTEL CN
FAX: (1) 500-2953

Intel Semiconductor Ltd.*
10/F East Tower
Bond Center
Queensway, Central
Hong Kong
Tel: (852) 844-4555
FAX: (852) 868-1989

INDIA

Intel Asia Electronics, Inc.
4/2, Samrah Plaza
St. Mark's Road
Bangalore 560001
Tel: 91-812-215773
TLX: 953-845-2646 INTEL IN
FAX: 091-812-215067

JAPAN

Intel Japan K.K.
5-6 Tokodai, Tsukuba-shi
Ibaraki, 300-26
Tel: 0298-47-8511
FAX: 0298-47-8450

Intel Japan K.K.*
Hachioji ON Bldg.
4-7-14 Myojin-machi
Hachioji-shi, Tokyo 192
Tel: 0426-48-8770
FAX: 0426-48-8775

Intel Japan K.K.*
Bldg. Kumagaya
2-69 Hon-cho
Kumagaya-shi, Saitama 360
Tel: 0485-24-6871
FAX: 0485-24-7518

Intel Japan K.K.*
Kawa-asa Bldg.
2-11-5 Shin-Yokohama
Kohoku-ku, Yokohama-shi
Kanagawa, 222
Tel: 045-474-7660
FAX: 045-471-4394

Intel Japan K.K.*
Ryokuchi-Eki Bldg.
2-4-1 Terauchi
Toyonaka-shi, Osaka 560
Tel: 06-863-1091
FAX: 06-863-1084

Intel Japan K.K.
Shinmaru Bldg.
1-5-1 Marunouchi
Chiyoda-ku, Tokyo 100
Tel: 03-3201-3621
FAX: 03-3201-6850

Intel Japan K.K.
Green Bldg.
1-16-20 Nishiki
Naka-ku, Nagoya-shi
Aichi 460
Tel: 052-204-1261
FAX: 052-204-1285

KOREA

Intel Korea, Ltd.
16th Floor, Life Bldg.
61 Yoido-dong, Youngdeungpo-Ku
Seoul 150-010
Tel: (2) 784-8186
FAX: (2) 784-8096

SINGAPORE

Intel Singapore Technology, Ltd.
101 Thomson Road #08-03/06
United Square
Singapore 1130
Tel: (65) 250-7811
FAX: (65) 250-9256

TAIWAN

Intel Technology Far East Ltd.
Taiwan Branch Office
8th Floor, No. 205
Bank Tower Bldg.
Tung Hua N. Road
Taipei
Tel: 886-2-5144202
FAX: 886-2-717-2455

INTERNATIONAL DISTRIBUTORS/REPRESENTATIVES

ARGENTINA

Dafsys S.R.L.
Chacabuco, 90-6 Piso
1069-Buenos Aires
Tel. & FAX: 54.1334.1871

AUSTRALIA

Email Electronics
15-17 Hume Street
Huntingdale, 3166
Tel: 011-61-3-544-8244
TLX: AA 30895
FAX: 011-61-3-543-8179

NSD-Australia
205 Middleborough Rd.
Box Hill, Victoria 3128
Tel: 03 8900970
FAX: 03 8990819

BRAZIL

Microlinear
Largo do Arouche, 24
01219 Sao Paulo, SP
Tel: 5511-220-2215
FAX: 5511-220-5750

CHILE

Sisteco
Vecinal 40 – Las Condes
Santiago
Tel: 562-234-1644
FAX: 562-233-9895

CHINA/HONG KONG

Novel Precision Machinery Co., Ltd.
Room 728 Trade Square
681 Cheung Sha Wan Road
Kowloon, Hong Kong
Tel: (852) 360-8999
TWX: 32032 NVTNL HX
FAX: (852) 725-3695

GUATEMALA

Abinitio
11 Calle 2 – Zona 9
Guatemala City
Tel: 5022-32-4104
FAX: 5022-32-4123

INDIA

Micronic Devices
Arun Complex
No. 65 D.V.G. Road
Basavanagudi
Bangalore 560 004
Tel: 011-91-812-600-631
011-91-812-611-365
TLX: 9538458332 MDBG

Micronic Devices
No. 516 5th Floor
Swastik Chambers
Sion, Trombay Road
Chembur
Bombay 400 071
TLX: 9531 171447 MDEV

Micronic Devices
25/8, 1st Floor
Bada Bazaar Marg
Old Rajinder Nagar
New Delhi 110 060
Tel: 011-91-11-5723509
011-91-11-589771
TLX: 031-63253 MDND IN

Micronic Devices
6-3-348/12A Dwarakapuri Colony
Hyderabad 500 482
Tel: 011-91-842-226748

S&S Corporation
1587 Kooser Road
San Jose, CA 95118
Tel: (408) 978-6216
TLX: 820281
FAX: (408) 978-8635

JAMAICA

MC Systems
10-12 Grenada Crescent
Kingston 5
Tel: (809) 929-2638
(809) 926-0188
FAX: (809) 926-0104

JAPAN

Asahi Electronics Co. Ltd.
KMM Bldg. 2-14-1 Asano
Kokurakita-ku
Kitakyushu-shi 802
Tel: 093-511-6471
FAX: 093-551-7861

CTC Components Systems Co., Ltd.
4-8-1 Dobashi, Miyamae-ku
Kawasaki-shi, Kanagawa 213
Tel: 044-852-5121
FAX: 044-877-4268

Dia Semicon Systems, Inc.
Flower Hill Shinmachi Higashi-kan
1-23 Shinmachi, Setagaya-ku
Tokyo 154
Tel: 03-3439-1600
FAX: 03-3439-1601

Okaya Koki
2-4-18 Sakae
Naka-ku, Nagoya-shi 460
Tel: 052-204-8315
FAX: 052-204-8380

Ryoyo Electro Corp.
Konwa Bldg.
1-12-22 Tsukiji
Chuo-ku, Tokyo 104
Tel: 03-3546-5011
FAX: 03-3546-5044

KOREA

J-Tek Corporation
Dong Sung Bldg. 9/F
158-24, Samsung-Dong, Kangnam-Ku
Seoul 135-090
Tel: (822) 557-8039
FAX: (822) 557-8304

Samsung Electronics
Samsung Main Bldg.
150 Taepyung-Ro-2KA, Chung-Ku
Seoul 100-102
C.P.O. Box 8780
Tel: (822) 751-3680
TWX: KORSST K 27970
FAX: (822) 753-9065

MEXICO

PSI S.A. de C.V.
Fco. Villa esq. Ajusco s/n
Cuernavaca, MOR 62130
Tel: 52-73-13-9412
52-73-17-5340
FAX: 52-73-17-5333

NEW ZEALAND

Email Electronics
36 Olive Road
Penrose, Auckland
Tel: 011-64-9-591-155
FAX: 011-64-9-592-681

SAUDI ARABIA

AAE Systems, Inc.
642 N. Pastoria Ave.
Sunnyvale, CA 94086
U.S.A.
Tel: (408) 732-1710
FAX: (408) 732-3095
TLX: 494-3405 AAE SYS

SINGAPORE

Electronic Resources Pte, Ltd.
17 Harvey Road
#03-01 Singapore 1336
Tel: (65) 283-0888
TWX: RS 56541 ERS
FAX: (65) 289-5327

SOUTH AFRICA

Electronic Building Elements
178 Erasmus St. (off Watermeyet St.)
Meyerspark, Pretoria, 0184
Tel: 011-2712-803-7680
FAX: 011-2712-803-8294

TAIWAN

Micro Electronics Corporation
12th Floor, Section 3
285 Nanking East Road
Taipei, R.O.C.
Tel: (886) 2-7198419
FAX: (886) 2-7197916

Acer Sertek Inc.
15th Floor, Section 2
Chien Kuo North Rd.
Taipei 18479 R.O.C.
Tel: 886-2-501-0055
TWX: 23756 SERTEK
FAX: (886) 2-5012521

URUGUAY

Interfase
Zabala 1378
11000 Montevideo
Tel: 5982-96-0490
5982-96-1143
FAX: 5982-96-2965

VENEZUELA

Unixel C.A.
4 Transversal de Monte Cristo
Edf. AXXA, Piso 1, of. 1&2
Centro Empresarial Boleita
Caracas
Tel: 582-238-6082
FAX: 582-238-1816

*Field Application Location

intel®

NORTH AMERICAN SERVICE OFFICES

ALASKA

Intel Corp.
c/o TransAlaska Network
1515 Lore Rd.
Anchorage 99507
Tel: (907) 522-1776

Intel Corp.
c/o TransAlaska Data Systems
c/o GCI Operations
520 Fifth Ave., Suite 407
Fairbanks 99701
Tel: (907) 452-6264

ARIZONA

*Intel Corp.
410 North 44th Street
Suite 500
Phoenix 85008
Tel: (602) 231-0386
FAX: (602) 244-0446

*Intel Corp.
500 E. Fry Blvd., Suite M-15
Sierra Vista 85635
Tel: (602) 459-5010

ARKANSAS

Intel Corp.
c/o Federal Express
1500 West Park Drive
Little Rock 72204

CALIFORNIA

*Intel Corp.
21515 Vanowen St., Ste. 116
Canoga Park 91303
Tel: (818) 704-8500

*Intel Corp.
300 N. Continental Blvd.
Suite 100
El Segundo 90245
Tel: (213) 640-6040

*Intel Corp.
1900 Prairie City Rd.
Folsom 95630-9597
Tel: (916) 351-6143

*Intel Corp.
9665 Chesapeake Dr., Suite 325
San Diego 92123
Tel: (619) 292-8086

**Intel Corp.
400 N. Tustin Avenue
Suite 450
Santa Ana 92705
Tel: (714) 835-9642

**Intel Corp.
2700 San Tomas Exp., 1st Floor
Santa Clara 95051
Tel: (408) 970-1747

COLORADO

*Intel Corp.
600 S. Cherry St., Suite 700
Denver 80222
Tel: (303) 321-8086

CONNECTICUT

*Intel Corp.
301 Lee Farm Corporate Park
83 Wooster Heights Rd.
Danbury 06811
Tel: (203) 748-3130

FLORIDA

**Intel Corp.
800 Fairway Dr., Suite 160
Deerfield Beach 33441
Tel: (305) 421-0506
FAX: (305) 421-2444

*Intel Corp.
5850 T.G. Lee Blvd., Ste. 340
Orlando 32822
Tel: (407) 240-8000

GEORGIA

*Intel Corp.
20 Technology Park, Suite 150
Norcross 30092
Tel: (404) 449-0541

5523 Theresa Street
Columbus 31907

HAWAII

**Intel Corp.
Honolulu 96820
Tel: (808) 847-6738

ILLINOIS

**†Intel Corp.
Woodfield Corp. Center III
300 N. Martingale Rd., Ste. 400
Schaumburg 60173
Tel: (708) 605-8031

INDIANA

*Intel Corp.
8910 Purdue Rd., Ste. 350
Indianapolis 46268
Tel: (317) 875-0623

KANSAS

*Intel Corp.
10985 Cody, Suite 140
Overland Park 66210
Tel: (913) 345-2727

KENTUCKY

Intel Corp.
133 Walton Ave., Office 1A
Lexington 40508
Tel: (606) 255-2957

Intel Corp.
896 Hillcrest Road, Apt. A
Radcliff 40160 (Louisville)

LOUISIANA

Hammond 70401
(serviced from Jackson, MS)

MARYLAND

**Intel Corp.
10010 Junction Dr., Suite 200
Annapolis Junction 20701
Tel: (301) 206-2860

MASSACHUSETTS

**Intel Corp.
Westford Corp. Center
3 Carlisle Rd., 2nd Floor
Westford 01886
Tel: (508) 692-0960

MICHIGAN

*Intel Corp.
7071 Orchard Lake Rd., Ste. 100
West Bloomfield 48322
Tel: (313) 851-8905

MINNESOTA

*Intel Corp.
3500 W. 80th St., Suite 360
Bloomington 55431
Tel: (612) 835-6722

MISSISSIPPI

Intel Corp.
c/o Compu-Care
2001 Airport Road, Suite 205F
Jackson 39208
Tel: (601) 932-6275

MISSOURI

*Intel Corp.
3300 Rider Trail South
Suite 170
Earth City 63045
Tel: (314) 291-1990

Intel Corp.
Route 2, Box 221
Smithville 64089
Tel: (913) 345-2727

NEW JERSEY

**Intel Corp.
300 Sylvan Avenue
Englewood Cliffs 07632
Tel: (201) 567-0821

*Intel Corp.
Lincroft Office Center
125 Half Mile Road
Red Bank 07701
Tel: (908) 747-2233

NEW MEXICO

Intel Corp.
Rio Rancho 1
4100 Sara Road
Rio Rancho 87124-1025
(near Albuquerque)
Tel: (505) 893-7000

NEW YORK

*Intel Corp.
2950 Expressway Dr. South
Suite 130
Islandia 11722
Tel: (516) 231-3300

Intel Corp.
300 Westage Business Center
Suite 230
Fishkill 12524
Tel: (914) 897-3860

Intel Corp.
5858 East Molloy Road
Syracuse 13211
Tel: (315) 454-0576

NORTH CAROLINA

*Intel Corp.
5800 Executive Center Drive
Suite 105
Charlotte 28212
Tel: (704) 568-8966

**Intel Corp.
5540 Centerview Dr., Suite 215
Raleigh 27606
Tel: (919) 851-9537

OHIO

**Intel Corp.
3401 Park Center Dr., Ste. 220
Dayton 45414
Tel: (513) 890-5350

*Intel Corp.
25700 Science Park Dr., Ste. 100
Beachwood 44122
Tel: (216) 464-2736

OREGON

**Intel Corp.
15254 N.W. Greenbrier Pkwy.
Building B
Beaverton 97006
Tel: (503) 645-8051

PENNSYLVANIA

*†Intel Corp.
925 Harvest Drive
Suite 200
Blue Bell 19422
Tel: (215) 641-1000
1-800-468-3548
FAX: (215) 641-0785

**†Intel Corp.
400 Penn Center Blvd., Ste. 610
Pittsburgh 15235
Tel: (412) 823-4970

*Intel Corp.
1513 Cedar Cliff Dr.
Camp Hill 17011
Tel: (717) 761-0860

PUERTO RICO

Intel Corp.
South Industrial Park
P.O. Box 910
Las Piedras 00671
Tel: (809) 733-8616

TEXAS

**Intel Corp.
Westech 360, Suite 4230
8911 N. Capitol of Texas Hwy.
Austin 78752-1239
Tel : (512) 794-8086

**†Intel Corp.
12000 Ford Rd., Suite 401
Dallas 75234
Tel: (214) 241-8087

**Intel Corp.
7322 SW Freeway, Suite 1490
Houston 77074
Tel: (713) 988-8086

UTAH

Intel Corp.
428 East 6400 South
Suite 104
Murray 84107
Tel: (801) 263-8051
FAX: (801) 268-1457

VIRGINIA

*Intel Corp.
9030 Stony Point Pkwy.
Suite 360
Richmond 23235
Tel: (804) 330-9393

WASHINGTON

**Intel Corp.
155 108th Avenue N.E., Ste. 386
Bellevue 98004
Tel: (206) 453-8086

CANADA

ONTARIO

**Intel Semiconductor of
Canada, Ltd.
2650 Queensview Dr., Ste. 250
Ottawa K2B 8H6
Tel: (613) 829-9714

**Intel Semiconductor of
Canada, Ltd.
190 Attwell Dr., Ste. 102
Rexdale (Toronto) M9W 6H8
Tel: (416) 675-2105

QUEBEC

**Intel Semiconductor of
Canada, Ltd.
1 Rue Holiday
Suite 115
Tour East
Pt. Claire H9R 5N3
Tel: (514) 694-9130
FAX: 514-694-0064

CUSTOMER TRAINING CENTERS

ARIZONA

2402 W. Beardsley Road
Phoenix 85027
Tel: (602) 869-4288
1-800-468-3548

SYSTEMS ENGINEERING OFFICES

MINNESOTA

3500 W. 80th Street
Suite 360
Bloomington 55431
Tel: (612) 835-6722

NEW YORK

2950 Expressway Dr., South
Islandia 11722
Tel: (506) 231-3300

*Carry-in locations
**Carry-in/mail-in locations

CG/SALE/041692

UNITED STATES
Intel Corporation
2200 Mission College Boulevard
P.O. Box 58119
Santa Clara, CA 95052-8119

JAPAN
Intel Japan K.K.
5-6 Tokodai, Tsukuba-shi
Ibaraki, 300-26

FRANCE
Intel Corporation S.A.R.L.
1, Rue Edison, BP 303
78054 Saint-Quentin-en-Yvelines Çedex

UNITED KINGDOM
Intel Corporation (U.K.) Ltd.
Pipers Way
Swindon
Wiltshire, England SN3 1RJ

GERMANY
Intel Semiconductor GmbH
Dornacher Strasse 1
8016 Feldkirchen bei Muenchen

HONG KONG
Intel Semiconductor Ltd.
10/F East Tower
Bond Center
Queensway, Central

CANADA
Intel Semiconductor of Canada, Ltd.
190 Attwell Drive, Suite 500
Rexdale, Ontario M9W 6H8

Printed in U.S.A./0492/15K/RRD DM
Order Number: 270710-002
© Intel Corporation, 1992