intel®

80960CA
User's Manual

Order Number: 270710-001

80960CA User's Manual

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 58130
SANTA CLARA, CA 95052-8130

In the U.S. and Canada
call toll free
(800) 548-4725

CURRENT HANDBOOKS

Product line handbooks contain data sheets, application notes, article reprints and other design information.

TITLE	LITERATURE ORDER NUMBER
COMPLETE SET OF HANDBOOKS (Available in U.S. and Canada only)	**231003**
AUTOMOTIVE PRODUCTS HANDBOOK (Not included in handbook set)	231792
COMPONENTS QUALITY/RELIABILITY HANDBOOK	210997
EMBEDDED CONTROL APPLICATIONS HANDBOOK	270648
8-BIT EMBEDDED CONTROLLER HANDBOOK	270645
16-BIT EMBEDDED CONTROLLER HANDBOOK	270646
32-BIT EMBEDDED CONTROLLER HANDBOOK	270647
MEMORY COMPONENTS HANDBOOK	210830
MICROCOMMUNICATIONS HANDBOOK	231658
MICROCOMPUTER PROGRAMMABLE LOGIC HANDBOOK	296083
MICROPROCESSOR AND PERIPHERAL HANDBOOK (2 volume set)	230843
MILITARY PRODUCTS HANDBOOK (2 volume set. Not included in handbook set)	210461
OEM BOARDS AND SYSTEMS HANDBOOK	280407
PRODUCT GUIDE (Overview of Intel's complete product lines)	210846
SYSTEMS QUALITY/RELIABILITY HANDBOOK	231762
INTEL PACKAGING OUTLINES AND DIMENSIONS (Packaging types, number of leads, etc.)	231369
LITERATURE PRICE LIST (U.S. and Canada) (Comprehensive list of current Intel Literature)	210620
INTERNATIONAL LITERATURE GUIDE	E00029

CG/LIT/100188

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 _____

Postage: add 10% of subtotal → Postage _____

Total _____

Pay by check, money order, or include company purchase order with this form ($100 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 58130
Santa Clara, CA 95052-8130

International Customers outside the U.S. and Canada should use the International order form 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 12/31/89.

Source HB

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.

intel®

80960CA
USER'S MANUAL

1989

Intel Corporation makes no warranty for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein.

Intel retains the right to make changes to these specifications at any time, without notice.

Contact your local sales office to obtain the latest specifications before placing your order.

The following are trademarks of Intel Corporation and may only be used to identify Intel products:

376, 386, 387, 486, 4-SITE, Above, ACE51, ACE96, ACE186, ACE196, ACE960, BITBUS, COMMputer, CREDIT, Data Pipeline, ETOX, Genius, $\hat{i}$, i486, i860, ICE, iCEL, ICEVIEW, iCS, iDBP, iDIS, I^2ICE, iLBX, iMDDX, iMMX, Inboard, Insite, Intel, int$_e$l, Intel386, int$_e$lBOS, Intel Certified, Intelevision, int$_e$ligent Identifier, int$_e$ligent Programming, Intellec, Intellink, iOSP, iPDS, iPSC, iRMK, iRMX, iSBC, iSBX, iSDM, iSXM, Library Manager, MAPNET, MCS, Megachassis, MICROMAINFRAME, MULTIBUS, MULTICHANNEL, MULTIMODULE, MultiSERVER, ONCE, OpenNET, OTP, PROMPT, Promware, QUEST, QueX, Quick-Erase, Quick-Pulse Programming, Ripplemode, RMX/80, RUPI, Seamless, SLD, SugarCube, UPI, and VLSiCEL, and the combination of ICE, iCS, iRMX, iSBC, iSBX, iSXM, MCS, or UPI and a numerical suffix.

MDS is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Mohawk Data Sciences Corporation.

MULTIBUS is a patented Intel bus.

CHMOS and HMOS are patented processes of Intel Corp.

Intel Corporation and Intel's FASTPATH are not affiliated with Kinetics, a division of Excelan, Inc. or its FASTPATH trademark or products.

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

Intel Corporation
Literature Sales
P.O. Box 58130
Santa Clara, CA 95052-8130

©INTEL CORPORATION 1989

PREFACE

The Intel 80960CA is a high-performance 32-bit embedded processor which is part of the 80960 processor family. This *80960CA User's Manual* provides detailed programming and hardware design information for the 80960CA. It is written for programmers and hardware designers who understand the basic operating principles of integrated processors and their systems.

RELATED PUBLICATIONS

The user's manual does not provide electrical specifications for the device, such as the DC and AC parametrics, operating conditions, and packaging specifications. This information is found in the *80960CA Data Sheet.*

The 80960CA is based on the 80960 embedded processor architecture. The 80960CA is only a single member of a family of 80960-based processors. Other 80960 family members span a range of performance and integration requirements for embedded applications. For information on other 80960 family members or the 80960 architecture in general refer to the following publications:

- 32-Bit Embedded Controller Handbook, Intel, Order No. 270647
- G. Meyers, D. Budde, The 80960 Microprocessor Architecture, Wiley, 1988.

MANUAL STRUCTURE

This manual is organized in three parts. The following is a synopsis of the contents of each part of the user's manual:

- *Part I-Programming the 80960CA* details the programming environment for the 80960CA. The processor's registers, instruction set, data types, addressing modes, interrupt mechanism, external interrupt interface, and fault mechanism are described in detail in this part of the user's manual.
- *Part II-System Implementation* describes the requirements for designing a system around the 80960CA. The external bus interface, and the integrated DMA controller are described in this part of the manual. The programming requirements for the DMA controller, bus controller, and for initialization of the 80960CA are also described.
- *Part III-Appendices* include quick references for hardware design and programming of the 80960CA. Appendices are also provided which describe the 80960CA's internal architecture, writing assembly-level code to exploit the parallelism of the processor, and the consideration for writing software which is compatible for all members of the 80960 family of embedded processors.

NOTATION AND TERMINOLOGY

The following paragraphs describe the 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 fields or preserved fields. A reserved field is one that may be used by specific implementations of the 80960 architecture. To ensure that a current software design is compatible with all processor implementations based on the 80960 architecture, the bits in reserved fields should be set to 0 when the data structure is initially created. Thereafter, software should not modify or rely on the value of these fields.

Some bits or fields in the architecturally-defined data structures are shown as requiring specific encodings. 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.

A preserved field is one that the processor does not use. Software may use preserved fields for any function.

Specifying Bit and Signal Values

The terms "set" and "clear" are used in this manual to refer to the value of bits 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 given 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 the 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 80960CA'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 generic names for these registers are g0 through g15 for the globals, r0 through r15 for the locals, and sf0, sf1, and sf2 for the special function registers. The list below shows descriptive names for registers along with the commonly used abbreviation.

r0 – Previous Frame Pointer (PFP)
r1 – Stack Pointer (SP)
r2 – Return Instruction Pointer (RIP)
g15 – Frame Pointer (FP)
sf0 – Interrupt Pending (IPND) Register
sf1 – Interrupt Mask (IMSK) Register
sf2 – DMA Command (DMAC) Register

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. A bit controls a processor function and is programmed by the user; a flag indicates status and is generally set by the processor. (The user may also program certain flags.) A field is 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".

TABLE OF CONTENTS

PART II - SYSTEM IMPLEMENTATION

CHAPTER 12
BUS INTERFACE EXAMPLES

CHAPTER 14
INITIALIZATION AND SYSTEM REQUIREMENTS

PART III - APPENDICES

FIGURES

TABLES

EXAMPLES

Introduction 1

CHAPTER 1
INTRODUCTION TO THE 80960CA

The 80960CA is the second-generation addition to Intel's 80960 family of embedded processors. This processor combines sustained multiple-instruction per clock execution in addition to high-speed DMA, interrupt, and bus control functions on a single CHMOS device. The 80960CA represents Intel's continuing commitment to provide a spectrum of reliable, high-performance processors and controllers to satisfy the needs of embedded applications.

The 80960CA member of the 80960 family is designed for those applications which require greater performance on a single chip than is found in an entire embedded RISC system. While the sheer speed of the 80960CA enriches traditional 80960 applications, the processor's integration of performance-sensitive system functions brings its parallel-computing performance to cost- and space-sensitive embedded applications.

As shown in Figure 1-1, the single-chip 80960CA integrates 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/s DMA controller, and a high-speed interrupt controller.

This chapter introduces the 80960CA implementation of the 80960 architecture, the 80960CA integrated system peripherals, and the 80960 product family.

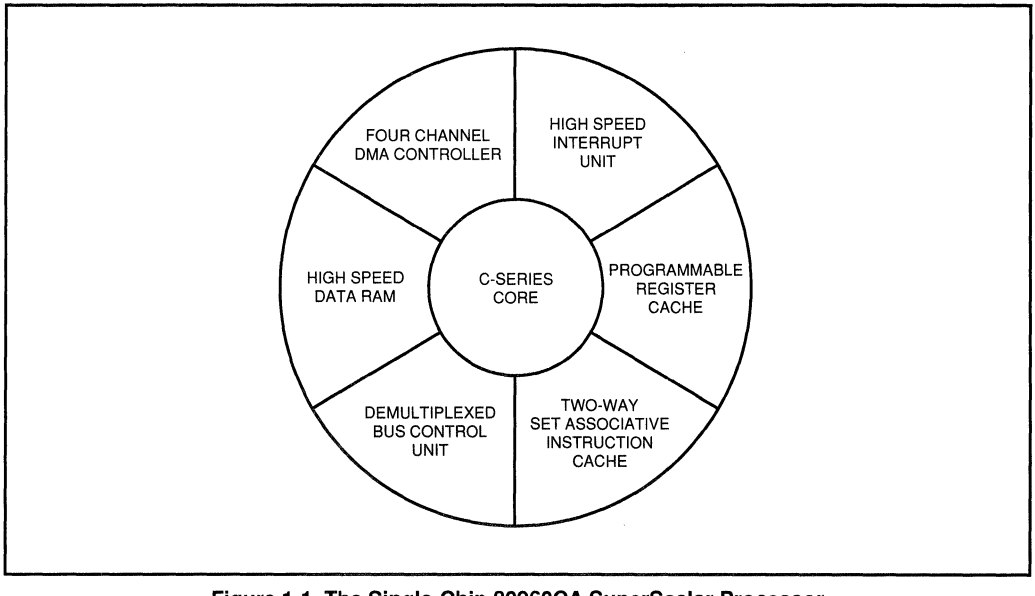


Figure 1-1. The Single-Chip 80960CA SuperScalar Processor

THE 80960CA EMBEDDED PROCESSOR ARCHITECTURE

The 80960 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 every clock, 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. The SuperScalar 80960CA implements sustained parallel decode, issue, and execution.

The on-chip instruction cache enables parallel decode by constantly providing the next four unexecuted instructions to the processor's instruction scheduler. The scheduler inspects all four instructions in a single clock, and issues one, two or three of these instructions in the same clock. 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 an instruction, or group of instructions, is issued by the scheduler, one of the 80960CA's 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. Each unit executes its instructions in parallel with other operations on the processor.

The 80960CA's 32 general purpose 32-bit registers are each six-ported to allow unimpeded parallel access to the 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.

Sustained execution of multiple instructions per clock from a sequential instruction stream is a consequence of the 80960CA's SuperScalar ability to decode multiple instructions at once and issue them to independent processing units where they are executed in parallel.

Full Procedure Call Model

Two types of procedure calls are supported on the 80960CA, 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 80960CA 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 80960CA 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 bits strings.

Most 80960 instructions are typical RISC operations. However, several commonly used complex instructions are also part of the 80960's instruction set. Performance can be optimized by implementing these commonly used functions with parallel hardware. For instance, the 32x32 multiply operation, which is a single instruction, takes less than 5 clocks to execute on the 80960CA (150ns or less at 33 MHz). Furthermore, the multiplier is a parallel unit. This allows instructions after a multiply to execute before the multiplication is complete. In fact, if several unrelated instructions follow a multiply, the multiplication will consume only one clock of execution.

Integrated Priority Interrupt Model

The 80960CA 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 is parallel to the program execution environment, this reduces interrupt latency and overhead, and provides flexible interrupt handling control.

Complete Fault Handling and Debug Capabilities

To aid in program development, the 80960CA detects faults (exceptions). When a fault is detected, the processor makes an implicit call to a fault-handling routine. The 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 80960 architecture provides a mechanism for monitoring processor activities through a software tracing facility. The 80960CA 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 80960CA also provides four breakpoint registers that allow break decisions to be made based upon instruction or data addresses.

80960CA SYSTEM INTEGRATION

The 80960CA is based on the C-series core, which is object code compatible with the 32-bit 80960 Core Architecture. Additionally, the 80960CA integrates three data control peripherals around the core: a bus control unit, a DMA controller, and an interrupt controller.

Pipelined, Burst Bus Control Unit

The 80960CA integrates a 32-bit high-performance bus controller to interface to external memory and peripherals. The 80960CA bus control unit incorporates full wait state logic and bus width control to provide high system performance with minimal system design complexity. The Bus Controller Unit features a maximum transfer rate of 132 MBytes per second (at 33MHz). 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. The source and destination can be any combination of internal RAM or external memory or peripherals. The DMA channels perform single-cycle or two-cycle transfers, and can perform data packing and unpacking on two-cycle transfers. Block transfers, in addition to source or destination synchronized transfers are provided. The DMA supports various types of transfers 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 32 priority levels with an interrupt task switch (latency) of 750 ns. The Interrupt Controller handles the prioritization of software interrupts, hardware interrupts and the process priority. In addition, it also manages four internal sources from the DMA controller, and a single non-maskable interrupt input.

80960 EMBEDDED PROCESSOR FAMILY

The benefit of a standard core architecture is that it allows software designers to develop building-block software, such as real-time kernels, or libraries of functions that have been optimized for the 80960 core architecture. These building blocks will be portable to any implementation of the 80960 architecture.

As shown in Figure 1-2, each compatible 80960 family product is a specialized applications device, which consists of an implementation of the core architecture plus a set of specific building blocks or peripherals. Present members of the 80960 family include the 80960KB, which has an on-chip floating point unit designed for applications which require intensive floating point calculations. Also available is the 80960MC, a military-grade version of the processor that has a memory-management unit (MMU) and supports Ada tasking. The 80960KA, a commercial version of the 80960KB without floating point, focuses on applications demanding pure integer performance. 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 80960 will feature different attributes to meet the price performance demands of embedded-processor applications.

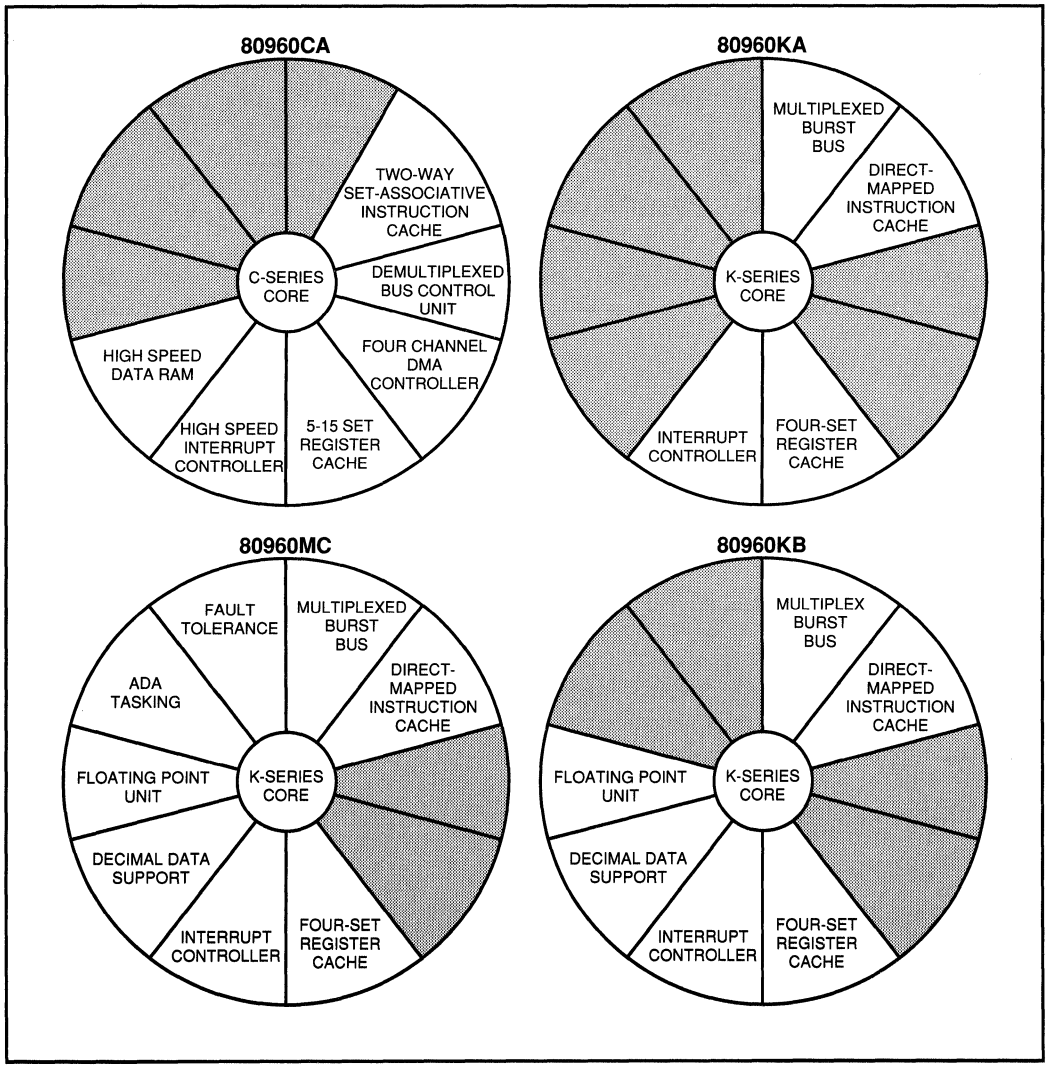


Figure 1-2. 80960 Family

PROGRAMMING **PART I**

Programming Environment 2

CHAPTER 2
PROGRAMMING ENVIRONMENT

This chapter describes the programming environment of the 80960CA, which includes the global and local registers, special function registers, control registers, literals, processor-state registers, and the address space.

OVERVIEW OF THE PROGRAMMING ENVIRONMENT

The architecture defines a programming environment in which programs are executed and data is stored and manipulated. Figure 2-1 shows the elements of the programming environment. These elements include a 4 GBytes (2^{32}-byte) 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 80960CA defines several data structures located in memory as part of the programming environment. These data structures are used to handle procedure calls, interrupts, faults, and to provide configuration information at initialization. These data structures are the local stack, supervisor stack, interrupt table, interrupt stack, fault table, control table, and system-procedure table.

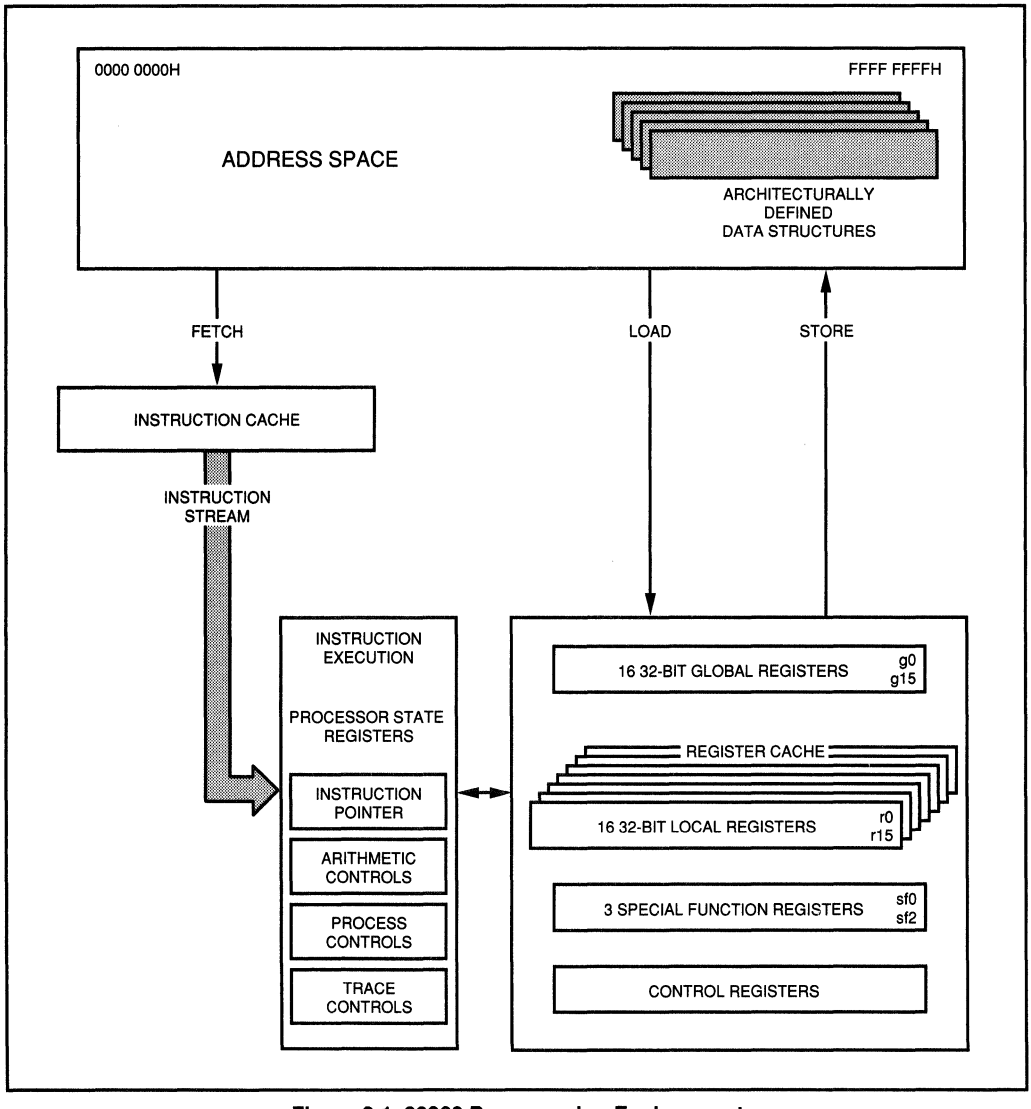


Figure 2-1. 80960 Programming Environment

REGISTERS AND LITERALS AS INSTRUCTION OPERANDS

Since the 80960CA uses only simple load and store instructions to access memory, operations take place at the register level. The 80960CA uses 16 global, 16 local, and three special-functions registers as instruction operands, as well as 32 literals (constants 0-31). These operands are listed in Table 2-1.

Table 2-1. Registers and Literals Used as Instruction Operands

Instruction Operand	Name	Function
g0 - g14	global 0-14	general purpose
fp (g15)	global 15	frame pointer
pfp (r0)	local 0	previous frame pointer
sp (r1)	local 1	stack pointer
rip (r2)	local 2	return instruction pointer
r3 - r15	local 3-15	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

Global Registers

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

The 80960 supplies 16 global registers, designated g0 through g15. Register g15 is reserved for the current Frame Pointer (FP). The FP contains the address of the first byte in the current (topmost) stack frame. (The FP and the procedure stack are discussed in detail in *Chapter 5, Procedure Calls.*)

Local Registers

The 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 automatically allocates a new set of local registers for that procedure and saves the local registers of the calling procedure. The program does not have to explicitly save and restore these registers.

Local registers r3 through r15 are general-purpose registers. Registers r0 through r2 are reserved for special functions, as follows: register r0 contains the Previous-Frame Pointer (PFP); r1 contains the Stack Pointer (SP); and r2 contains the Return-Instruction Pointer (RIP). The PFP, SP, and RIP are discussed in detail in *Chapter 5, Procedure Calls.*

Special Function Registers (SFRs)

The 80960 architecture provides a mechanism to expand its architectural register set with up to 32 additional 32-bit registers. On the 80960CA, three special-function registers (SFRs) are provided as an extension to the architectural register model. These registers are designated sf0, sf1, and sf2 (Table 2-1). Other possible additional registers are not defined for the 80960CA.

In general, the special-function registers provide a means to configure and monitor the status of the interrupt controller and DMA controller. The function of the 80960CA's SFRs is described in detail in *Chapter 6, Interrupts* and *Chapter 13, DMA Controller* in this manual.

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 an SFR operand is executed in user mode.

SFRs are not used as operands for instructions whose machine-level instruction formats are of type MEM or CTRL. Instruction with these formats include loads, stores, and instructions which cause program redirection (call, return, and branches). (See *Appendix D, Instruction Encoding Reference* for a description of the machine-level encoding for operands.) Table 2-2 summarizes the use of SFRs as instruction operands.

Registers sf3-sf31 are not implemented on the 80960CA. Reading or modifying unimplemented registers causes the operation-unimplemented fault to occur.

Register Scoreboarding

Register scoreboarding allows concurrent execution of sequential instructions. When an instruction is executed, 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 is able to execute those instructions before execution of the prior instruction is complete.

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. The example also illustrates overlapping instructions which do not have register dependencies.

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     r6,r7,r8        # and instruction is executed concurrently with multiply
```


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

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, and 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 four if three or four registers are accessed (e.g., g0, g4). If a register reference for a source value is not properly aligned, the source value is undefined. If a register reference for a destination value is not properly aligned, the registers that the processor writes to and the values which are written are undefined. 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 the instruction operands for each machine level instruction format and the positions which can be filled by each register or literal.

Table 2-2. Allowable Register Operands

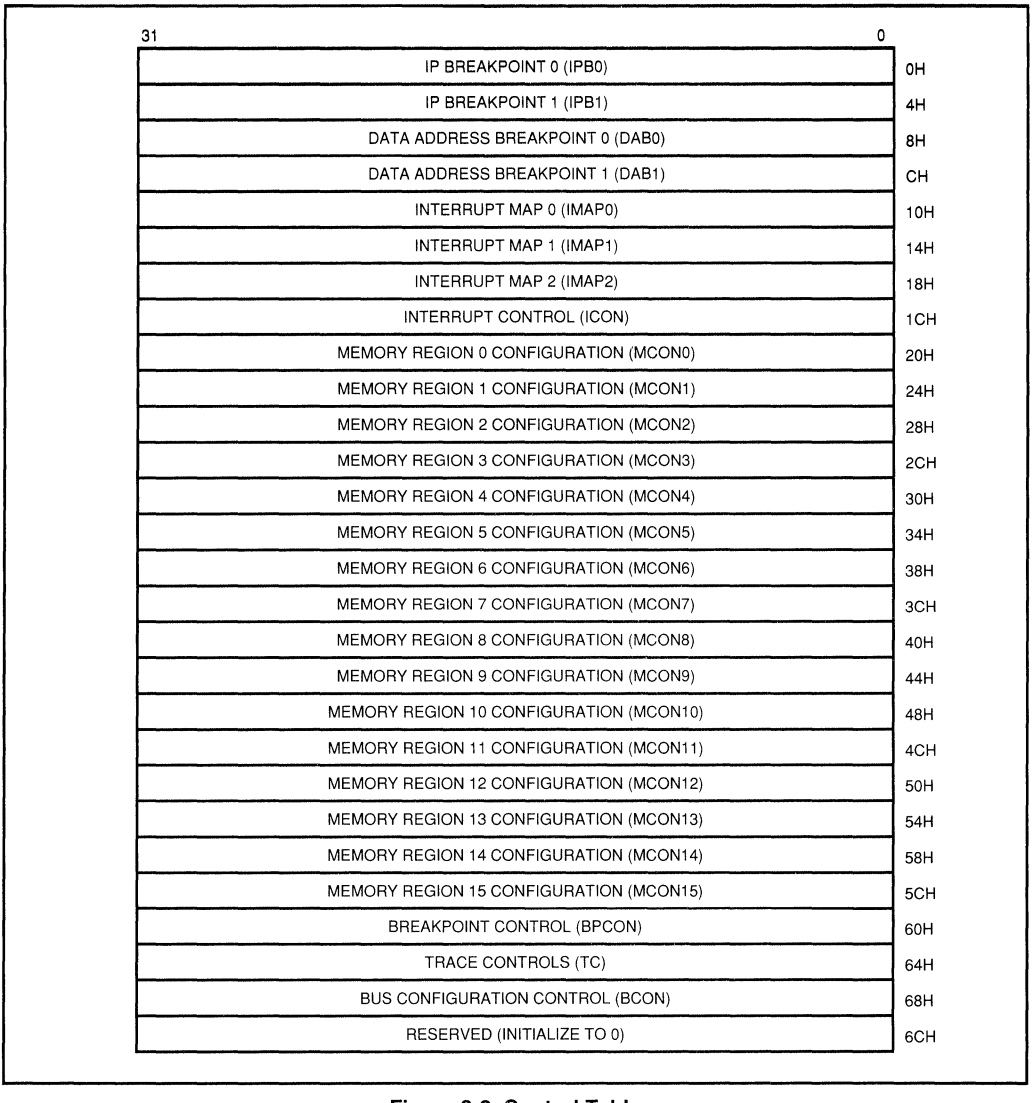
Instruction Format	Operand Field	$Operand_1$			
		Local Register	Global Register	Extended Register (SFR)	Literal
REG	src1	X	X	X	X
	src2	X	X	X	X
	src/dst (as src)	X	X		X
	src/dst (as dst)	X	X	X	
	src/dst (as both)	X	X	(2)	
MEM	src/dst	X	X		
	abase	X	X		
	index	X	X		
COBR	src1	X	X	X	X
	src2	X	X	X	

Notes:
1. X denotes that a 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**).

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, the control registers are loaded from a data structure called the control table (Figure 2-2).

The system control (**sysctl**) instruction is used to move the control table values to the on-chip control register. The control table is divide into seven quad-word groups. Each group is assigned a group number from one to seven. When the **sysctl** instruction is executed, the load-control-register message type along with the group number is specified. The **sysctl** instruction 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.)



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
BREAKPOINT CONTROL (BPCON)	60H
TRACE CONTROLS (TC)	64H
BUS CONFIGURATION CONTROL (BCON)	68H
RESERVED (INITIALIZE TO 0)	6CH

Figure 2-2. Control Table

At initialization, the control table is automatically loaded into the on-chip control registers. This action simplifies the user's start-up code by providing a transparent setup of the 80960CA's peripherals at initialization. (See *Chapter 14, Initialization and System Requirements.*)

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. The function of these data structures is described below.

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

The *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 a system call, know 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.* The system-procedure table structure and system-call mechanism is described in *Chapter 5, Procedure Calls*; the user-supervisor protection model is described in the section titled *User-Supervisor Model* in this chapter.

The *interrupt table* contains vectors (pointers) to interrupt-handling procedures. When an interrupt is serviced, a particular entry in the interrupt table is specified. To ensure that the handling of interrupts does not interfere with application programs, a separate interrupt stack is provided. The interrupt-handling mechanism is described in *Chapter 6, Interrupts.*

The *fault table* contains pointers to fault-handling procedures. When the processor detects a fault, a particular entry in the fault table is selected by the processor. The architecture does not require a separate fault-handling stack. Instead a fault-handling procedure uses the supervisor stack, user stack, or interrupt stack, depending on the execution mode of the processor when the fault occurred and the type of call made to the fault-handling procedure. The fault-handling mechanism is described in *Chapter 7, Faults.*

The 80960CA defines two initialization data structures: the *initialization boot record*, and the *process control block (PRCB).* These structures provide initialization data and pointers to other data structures in memory. When the processor is initialized, these pointers are read from the initialization data structures and cached for internal use.

Pointers to the system-procedure table, interrupt table, interrupt stack, fault table, and control table are specified in the process control block. The location of the supervisor stack is specified in the system-procedure table. The location of the user stack is specified in the user's start-up 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 is sometimes required to write into part of this table.)

The *control table* contains the values for the on-chip control registers. The control table values can be moved to the on-chip registers at initialization, or with the **sysctl** instruction.

MEMORY ADDRESS SPACE

The 80960'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. The 80960CA's address space is shown in Figure 2-3.

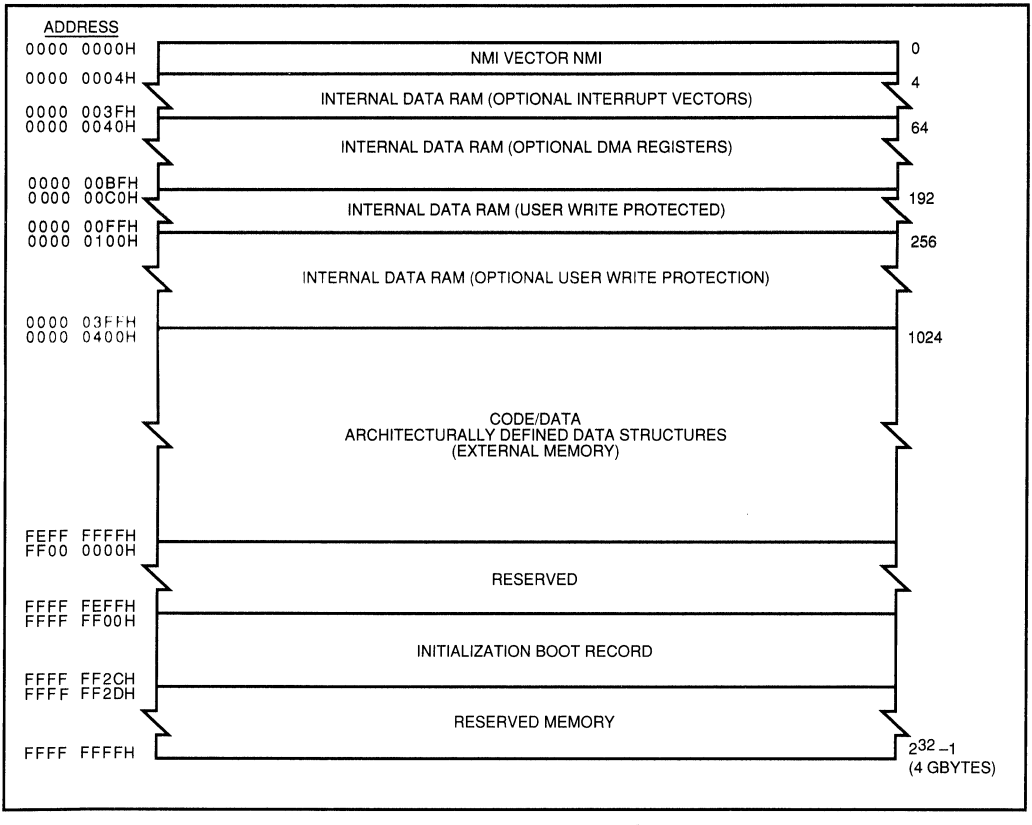


Figure 2-3. 80960CA Address Space

The 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 the kernel's code, data, and stack. But from the point of view of the processor, the address space is linear.)

An address in memory is a 32-bit value in the range 0 to FFFFFFFFH. It can be used to reference a single byte, a half-word (2 bytes), a word (4 bytes), a double-word (8 bytes), a triple-word (12 bytes), or a quad-word (16 bytes) in memory, depending on the instruction being used. (Refer to the descriptions of the load and store instructions in *Chapter 9, Instruction Reference* for information on multiple-byte addressing.)

Memory Requirements

The architecture requires that the external memory has the following properties:

- It must be byte addressable.
- No memory is mapped at reserved addresses unless specified by an implementation.
- It must guarantee *indivisible access* (read or write) for memory addresses that fall within 16-byte boundaries.
- It must guarantee *atomic access* for memory 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.

An 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.

An atomic access is 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 to access 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, Bus Controller.*)

The upper 16 MBytes of the address space (addresses FF000000H through FFFFFFFFH) are reserved for implementation-specific functions. In general, programs should not access this section of the address space unless an implementation specifically requires it. The initialization boot record is located in reserved memory of the 80960CA. (See Figure 2-3.) The 80960CA

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, and instruction fetches from these addresses are not allowed for the 80960CA. (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 the non-reserved address space while adhering to the following alignment requirements:

- Instructions must be aligned on word boundaries.
- All architecture-defined data structures must be aligned on the boundaries specified in Table 2-3.
- Instruction operands for the atomic instructions (**atadd**, **atmod**) must be aligned to word boundaries in memory.

The 80960CA does not require that load and store data be aligned in memory. The 80960CA can handle a non-aligned load or store request by either of two methods. The 80960CA is able to automatically service a non-aligned memory access with microcode assistance. (See *Chapter 10, Bus Controller*) Alternatively, an operation-unimplemented fault can be generated when a non-aligned access is detected. The method for handling non-aligned accesses by the 80960CA 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 blocks of data of various lengths from memory to registers (load) and from registers to memory (store). The allowable sizes for blocks are bytes, half-words (2 bytes), words (4 bytes), double words, triple words, and quad words. For example, the **stl** (store long) instruction stores an 8-byte (double-word) block of data in memory.

The most efficient way to move blocks of data greater than 16 bytes in length is to move them in quad-word increments, using the quad-word instructions (**ldq** and **stq**).

When a block of data is stored in memory, normally the least-significant byte of the block 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 80960CA also provides the option for ordering bytes in an opposite manner in memory. The most-significant byte of the block 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 blocks of data which are short words, or words. (For more details on byte ordering, see *Chapter 10, Bus Controller.*)

When loading a byte, half word, or word from memory to a register, the least-significant bit of the block is always loaded in bit 0 of the register. 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 and bit 31 is the most-significant bit.

Internal Data RAM

Internal data RAM is mapped to the lower 1 KByte of the 80960CA's address space (0000H to 03FFH). The 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 locations in the internal data RAM 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.

The size of the local register cache 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 a description of the PRCB.) 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 which have been set aside for the register cache. (See *Chapter 5, Procedure Calls.*)

The first 256 bytes of internal RAM (0000H to 00FFH) are user mode protected. This data RAM can be read while executing in user or supervisor mode. The RAM can only be modified in supervisor mode. Writes to these locations while in user mode causes a type-mismatch fault to be generated. This feature provides supervisor protection for the DMA and Interrupt functions which use the internal RAM. (See *User-Supervisor Protection Model* in this chapter.) User mode protection is optionally selected for the rest of the data RAM (0100H to 03FFH) by setting the RAM protection bit in the Bus Configuration Register (BCON). The protection option is disabled by default. (See *Chapter 10, Bus Controller.*)

Instruction Cache

The 80960CA's instruction cache enhances the processor's 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 cache is a 1 KByte two-way set associative cache, organized in lines of 4 words. To optimize the cache updates when branches are executed, two valid bits are used for each line. Depending on the target address for the branch, either two or four words of the line may be updated if a cache miss occurs.

The instruction cache is a read-only cache, and does not detect modification to the program memory by loads, stores, or the actions of other processors sharing the code space. In other words, the processor does not transparently support run-time modification of program memory.

Several situations may require modification of the program space. Uploading code to volatile memory at initialization or uploading code from a backplane bus or a disk, require that the processor force a consistency between external memory and internal cache. To achieve this consistency, the contents of the instruction cache can be invalidated after the code modification is complete. The **sysctl** instruction is used to invalidate the instruction cache for the 80960CA. The instruction 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 with the invalidation of the cache. In general, a program should ensure that modified code space is not accessed until the modification and the cache-invalidate is completed.

The 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, the **sysctl** instruction is executed with the *configure-instruction-cache* message. (See *System Control Function* later in this chapter.)

The processor can be directed to load a block of instructions into the cache and then disable all normal updates to the portion of the cache which was loaded. 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 portion of the cache. No external accesses are required for these instructions when the interrupt is serviced

Only interrupts can be directed to fetch instructions from the locked portion of the instruction cache. 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 locked portion of the instruction cache. Execution continues from the locked cache until a miss occurs (e.g., 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 of the cache acts as a normal two-way set associative cache. Normally, an application locks only 512 bytes of the cache. Locking the full 1 KByte cache means that all instruction fetches come from external memory except for the interrupts which are directed to the locked cache.

The **sysctl** instruction 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 block of memory 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 are the instruction-pointer register, arithmetic-controls register, process-controls register, and trace-controls register. The functions of these registers are defined in the following sections.

Instruction Pointer

The instruction pointer (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 2 least-significant bits of the IP are always zero.

All 80960 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 can not 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 value of the IP.

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 7, Procedure Calls* for further discussion of this operation.

Arithmetic-Controls Register

The arithmetic-controls (AC) register (shown in Figure 2-4) contains the condition-code flags; integer-overflow flag and mask bit; and a bit that controls faulting on imprecise faults. All the unused bits in the arithmetic controls are reserved and must be set to 0 at initialization.

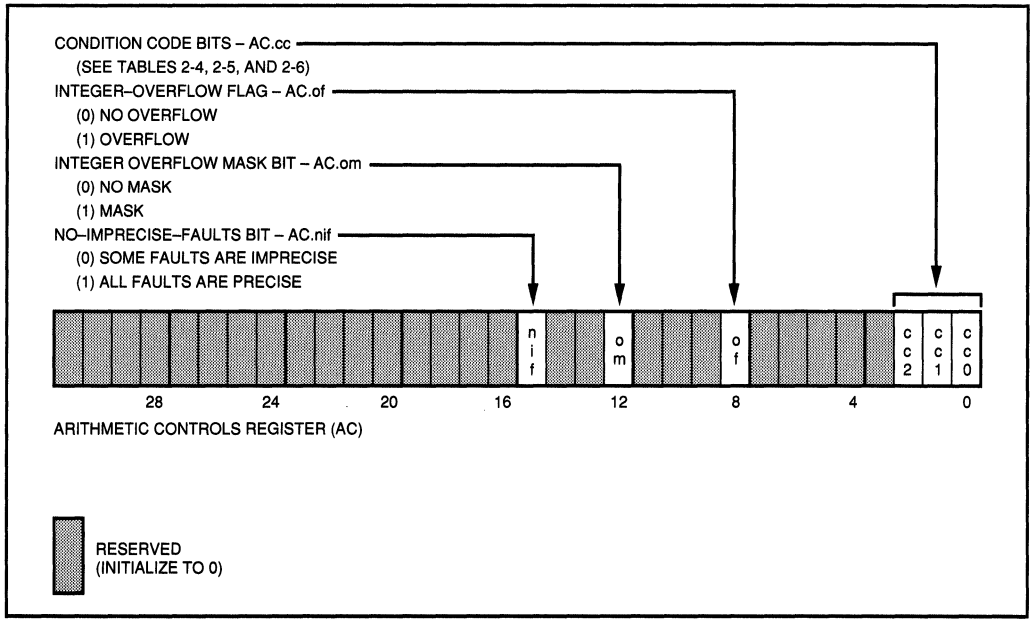


Figure 2-4. Arithmetic-Controls Register

Initializing and Modifying the Arithmetic Controls

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.*) After initialization, the modify-arithmetic-controls (**modac**) instruction allows any of the bits in the register to be examined and modified. (This instruction provides a mask operand that can be used to limit access to specific bits or groups of bits in the register.)

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

Condition Code

The processor sets the *condition-code field* (bits 0-2) in the arithmetic-controls register to indicate the results of certain instructions (usually compare instructions). Other instructions, such as conditional-branch instructions, examine this field and perform functions according to the state of the condition code. Once the processor has set the condition-code field, it remains unchanged until another instruction is executed that modifies the field.

The condition-code field is used to 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 field as shown in Table 2-4.

Table 2-4. Condition Codes for True or False Conditions

Condition Code	Condition
010_2	true
000_2	false

To show equality and inequalities, the processor sets the condition-code field as shown in Table 2-5.

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

Notes: Some implementations of the 80960 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 80960CA does not implement the floating-point processor on-chip.

To show carry out and overflow, the processor sets the condition-code field 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 field. 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 op-code and the instruction performs a bit wise AND of the mask and condition code.

Integer-Overflow

The *integer-overflow flag* (bit 8) and the *integer-overflow mask bit* (bit 12) in the arithmetic-controls register are used in conjunction with the arithmetic-integer-overflow fault. The mask bit disables generation of the fault. 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.

The integer-overflow flag is a "sticky flag". This means that once the processor sets this flag, it never implicitly clears it; the flag remains set until cleared by the program.

(Refer to the discussion of the arithmetic integer-overflow fault in *Chapter 7, Faults* for more information about the integer-overflow mask 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 precises and imprecise faults.)

Process-Controls Register

The process-controls (PC) register (Figure 2-5) contains information to control processor activity and show the current state of the processor. The various functions of this register are described in the following sections.

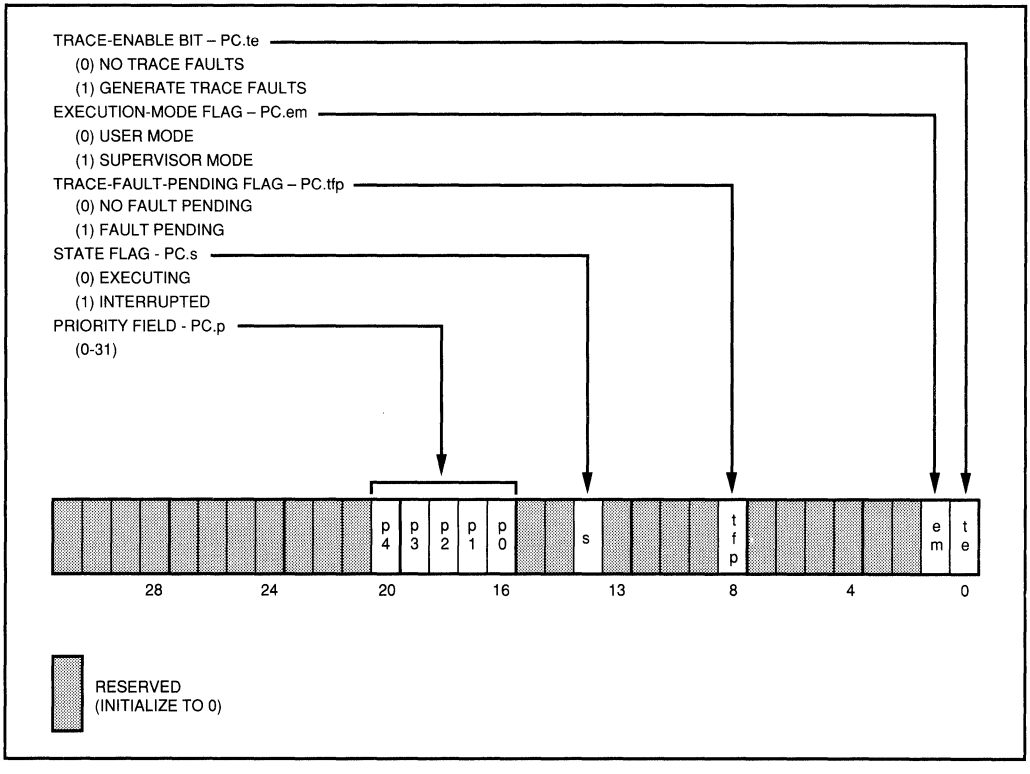


Figure 2-5. Process-Controls Register

INITIALIZING AND MODIFYING THE PROCESS CONTROLS

Any of the following three methods can be used to change bits in the process-controls 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.

The **modpc** instruction reads and modifies the process-controls register directly. The processor must be in the supervisor mode to execute this instruction. A type-mismatch fault is generated if **modpc** is executed in user mode. As with the **modac** instruction, the **modpc** instruction 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 the process controls in the interrupt or fault record that is saved on the stack. On the return from the interrupt or fault handler, the modified process controls are copied into the process-controls register. The processor must be in the supervisor mode prior to the return for the modified process controls to be copied into the process-controls register.

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

Bits 2 through 8, 11, 12, 14, 15, and 21 through 31 are reserved. These bits should be set to 0 at initialization. After initialization or after the processor is reinitialized, the process controls reflect the following conditions: priority=31, execution mode=supervisor, trace enable=off, state=interrupted.

Execution Mode

The *execution-mode flag* (bit 1) indicates that the processor is operating in the 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. (The user and supervisor modes are described in *User and Supervisor Protection Model.*)

Program State

The *state flag* (bit 13) of the process-controls register (Figure 2-5) indicates the state of the processor: 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

The *priority field* (bits 16 through 20) of the process-controls register (Figure 2-5) reflects the current executing or interrupted priority of the processor.

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

The *trace-enable bit* (bit 0) and *trace-fault-pending* (bit 10) *flag* 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 detail in *Chapter 8, Tracing and Debug.*

Trace-Controls Register

The trace-controls (TC) register, in conjunction with the process-controls register, controls the tracing facilities of the processor. It contains trace-mode enable bits and trace event flags, which are used to enable specific tracing modes and record trace events, respectively. The trace controls are described in *Chapter 8, Tracing and Debug.*

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. The primary function of this protection mechanism is to prevent application software from inadvertently altering the kernel.

Supervisor Mode Resources

The processor can be in either of two execution modes: user or supervisor. The supervisor mode is a privileged mode which provides several additional capabilities over the user mode.

- When the processor switches to the supervisor mode, it also switches to the supervisor stack. Switching to the supervisor stack helps maintain the integrity of a kernel. 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 ($\overline{SUP}$) is asserted. The supervisor pin 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 the supervisor mode, the processor is allowed access to a set of supervisor-only functions and instructions. For example, the processor uses the supervisor mode to handle interrupts and trace faults. Operations which can modify the behavior of the DMA controller, the interrupt controller, or which can reconfigure the characteristics of the bus controller, can only be performed in supervisor mode. These functions include modification of SFRs, control registers, or modification of the internal data RAM which is dedicated to the DMA controller and interrupt controller. If supervisor-only operations are attempted while the processor is in the user mode, a fault is generated. (*See Chapter 7, Faults.*) Table 2-7 lists the supervisor-only operations, and the fault which is generated if the operation is attempted in user mode.

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) and **udma** (update DMA)	constraint-privileged
SFR as instruction operand	type-mismatch
Protected internal data RAM write	type-mismatch

The execution mode flag in the process-controls register specifies the execution mode of the processor. The processor automatically sets and clears this flag when it switches between the two execution modes.

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.

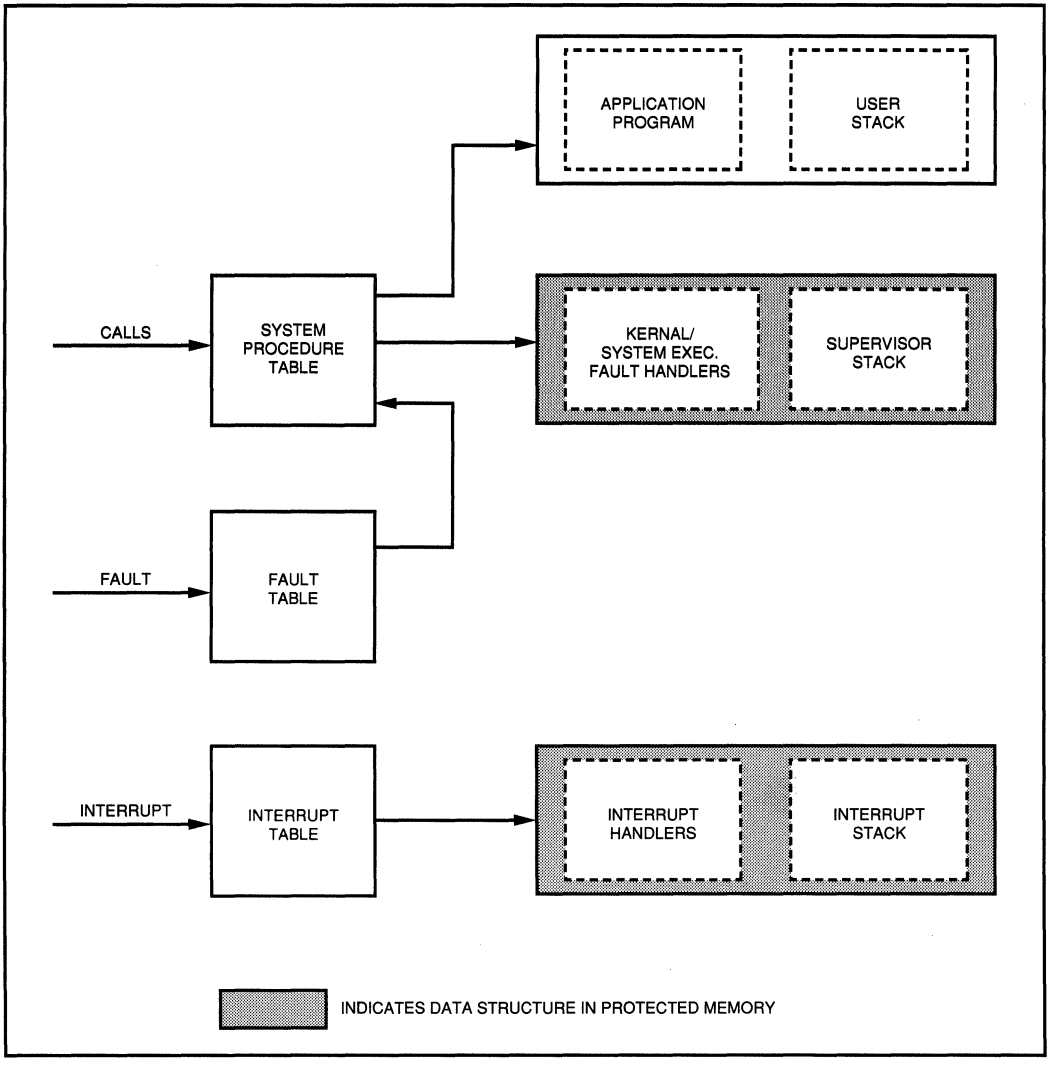


Figure 2-6. Example Application of the User-Supervisor Protection Model.

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 the 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 the execution mode to user mode (i.e., using the **modpc** instruction to change the execution mode bit of the process-controls register to 0). 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 80960CA. 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.

SYSCTL Instruction Syntax

The syntax of the **sysctl** instruction is generalized because the function of the operands differ, depending on the selection of the message type. The instruction takes three source operands (Figure 2-7). The message type field is always the second byte of the source 1 operand. The generalized operand fields of the instruction, 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).

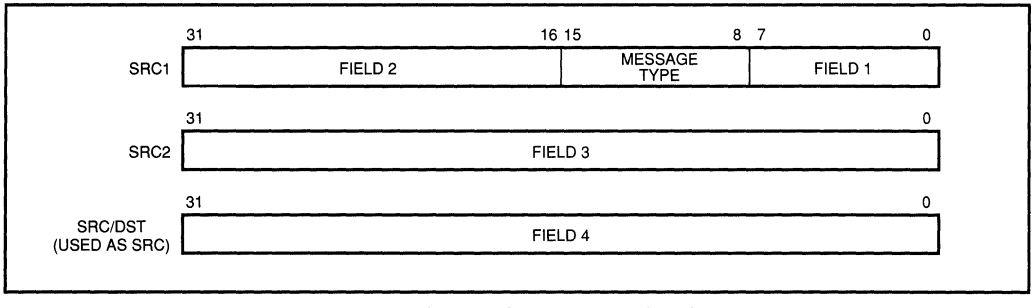


Figure 2-7. Source Operands for SYSCTL

The **sysctl** instruction 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

Message	Source1			Source 2	Source 3
	Type	Field 1	Field 2	Field 3	Field 4
Request Interrupt	00H	Vector No.	N/U	N/U	N/U
Invalidate Cache	01H	N/U	N/U	N/U	N/U
Configure Cache	02H	Mode (Table 2-9)	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 No.	N/U	N/U	N/U

Note: Sources and fields which are not used (designated N/U) are ignored.

System Control Messages

Five system control messages are defined. 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. Each message is described in detail below.

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 the **sysctl** instruction with a message type of 01H causes all cache entries to invalidate. This mode clears all of the valid bits in the cache. 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, and 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 the 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, 512 bytes of normal cache. The particular configure cache operation performed is determined by the value of field 1 in the **sysctl** instruction (Table 2-9). Field 3 of the instruction is a word-aligned 32-bit address when a load and lock mode is selected, otherwise this field is ignored.

Table 2-9. Cache Configuration Modes

Mode Field(1)	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 is initialized by programming the Cache Configuration Word in the PRCB. (See *Chapter 14, Initialization and System Requirements.*) The modes are provided to allow the cache to be turned off temporarily to aid in debugging.

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 causes the 80960CA to reinitialize. Field 3 and field 4 of **sysctl** must contain, respectively, the First Instruction Pointer and the PRCB Pointer. Reinitialization bypasses the 80960CA'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 steps taken when the 80960CA is reinitialized.)

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 pending interrupts and priorities fields of the table. The reinitialize message, in this case, specifies a new PRCB which contains a pointer to the new interrupt table in RAM. (See *Chapter 14, Initialization and System Requirements* for a detailed 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 invocation of the instruction 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 of the instruction must contain the number of the register group to be loaded. The register group number, and the registers represented in the Control Register Table are given in Table 2-10.

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	Breakpoint Control Register (BPCON)
	64H	Trace Controls Register (TC)
	68H	Bus Configuration Control (BCON)
	6CH	Reserved

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

Data Types and Memory Addressing Modes **3**

CHAPTER 3
DATA TYPES AND MEMORY ADDRESSING MODES

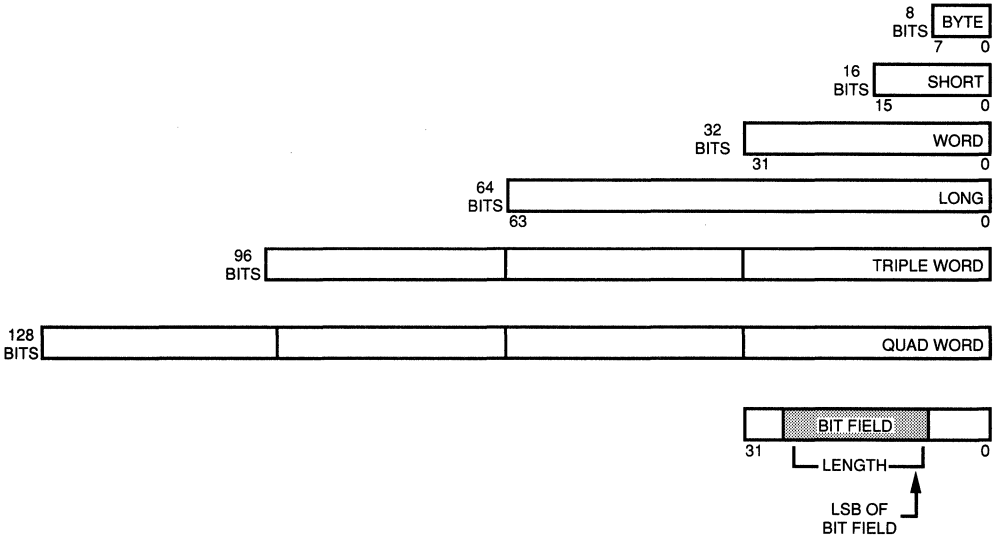
This chapter describes the architecturally-defined data types and memory addressing modes.

DATA TYPES

Several data lengths and data formats are referenced by the instruction set or are produced by instructions. The 80960 architecture defines the following types of data:

- 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 summarizes the data types defined by the 80960 architecture and shows the length and numeric range of each type.



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

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. The architecture defines four sizes of integers: 8-bit (byte integers), 16-bit (short integers), 32-bit (integers), and 64-bit (long integers).

Most integer instructions operate on the 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 80960CA's instructions reference or produce the long-integer data type.

Note: HLL compilers may define long integer types differently than defined by the 80960 architecture.

The size of an integer load or store (byte, short, or word) determines how sign extension or truncation of data is performed when data is moved between registers and memory. For the **ldib** (load integer byte) and **ldis** (load integer short) instructions, 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 the **stib** (store integer byte) and **stis** (store integer short) instruction, a 32-bit two's complement number in a register is stored to memory as a byte or short-word. If the data in the register is too large to be stored as a byte or short-word, the value is truncated, and the integer-overflow condition is signalled. A flag in the arithmetic-controls register is set, or the integer-overflow fault is generated when overflow occurs. (See *Chapter 7, Faults,* for details of the integer overflow fault.) For the **ld** (load word), and **st** (store word) instructions, data is moved directly between memory and a register with no sign extension or truncation of data.

Ordinals

Ordinals are an unsigned integer data type. These values are stored and operated on as positive binary values. The processor recognizes four sizes of ordinals: 8-bit (byte ordinals), 16-bit (short ordinals), 32-bit (ordinals), and 64-bit (long ordinals).

The large number of instructions which perform logical, bit manipulation, and unsigned arithmetic operations reference 32-bit ordinal operands. 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 fields of bits within register operands. An individual bit is specified for a bit operation by giving its bit number and its 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. A bit field is defined by giving its length in bits and the bit number of its lowest-numbered bit.

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 value, a byte or short integer load can result in sign extension of data in a register, and 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 the moving of blocks of data. No manipulation (sign extension, zero extension, or truncation) of the data is performed in these instructions.

The triple- and quad-word data types can be considered a superset (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. Long-word operands in registers must be aligned to double-register boundaries. Triple- and quad-word operands in registers must be aligned to quad-register boundaries. For the 80960CA, alignment of data 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 on the alignment requirements for data and instructions.

MEMORY ADDRESSING MODES

The processor provides 9 modes for addressing operands in memory. These modes are grouped as follows:

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

Each addressing mode is used to reference a byte in the processor's address space. Table 3-1 shows all the memory addressing modes, a brief description of the elements of the address in each mode, and the assembly-code syntax for each mode.

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 the size of the offset. For the absolute-offset addressing mode, the offset is an ordinal number ranging from 0 to 2047. For the absolute-displacement addressing mode the offset is an integer (called a displacement) ranging from -2^{31} to $2^{31}-1$. The absolute-offset addressing mode is encoded in the MEMA machine-instruction format, and the absolute-displacement addressing mode is encoded in the MEMB format. (The encoding-level addressing modes and instruction formats are described in *Appendix D, Instruction-Encoding Reference.*)

At the assembly-language level the two absolute-addressing modes are combined into one (i.e., the assembler syntax for the two addressing modes is the same). The Intel 80960 assembler allows 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

The register-indirect addressing modes use a 32-bit value in a register as a base for the address calculation. The value in the register is referred to as the address base (designated *abase* in table 3-1). An optional scaled-index and offset can be added to this address base, depending on the addressing mode.

In the 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. The allowable scale factors are 1, 2, 4, 8, and 16.

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

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

The 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, which is encoded in the MEMB instruction format.

The register indirect addressing modes are useful for addressing elements of an array or record structure. When addressing elements in an array, the abase value gives the address of the first element in the array and an offset (or displacement) selects a particular element in the array.

Index with Displacement

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

IP with Displacement

The IP-with-displacement addressing mode references the address of the next instruction plus the displacement. This addressing mode is often used with load and store instructions to make them IP relative. At the instruction-encoding level, the displacement plus a constant of 8 is added to the IP of the instruction.

Addressing-Mode Examples

The following examples show how the 80960's addressing modes are encoded in assembly language.

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 register 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 registers r4
                            # through r7.

st      g3,xyz(g4)[g5*2]    # register indirect with index and
                            # displacement; word in g3 loaded into
                            # memory 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.
```

The example below illustrates the usefulness of the scaled index and the scaled index plus displacement addressing modes. In this example, a procedure named array_op uses these addressing modes to fill two contiguous blocks of memory which are separated by a constant offset. A pointer to the top of the block is passed to the procedure in g0, the size of the block in g1, and the fill data in g2.

Example 3-2. Use of the 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
                                        # element to be filled.
        b       .I33
.I34:
        st      g2,(r4)[r3*4]           # fill array at index
        st      g2,0x30(r4)[r3*4]       # fill array at index plus constant offset
        subi    1,r3,r3                 # decrement index
.I33:
        cmpible 0,r3,.I34               # store the next array elements if index
                                        # is not 0
        ret
```

Using the iC960 compiler with the C function shown below produces this code. The array assignments are compiled as indirect-plus-scaled-index stores. The constant defined as "OFFSET" causes the array assignment to compile with an absolute offset.

Example 3-3. Compiler Generated Addressing Modes

```
# define OFFSET 12

void array_op(array, block_size, data)

int *array, block_size, data;
{
        int i;
                for (i=block_size-1; i >= 0; i--) {
                        array[i] = data;
                        array[i+OFFSET] = data;
                        }
        return;
}
```

Instruction Set Summary 4

CHAPTER 4
INSTRUCTION SET SUMMARY

This chapter provides an overview of the 80960 Core Architecture instruction set and the 80960CA-specific instruction set extensions. Included is a description of the assembly-language and instruction-encoding formats; an overview of the various instruction groups; and brief descriptions of the instructions in each group.

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

INSTRUCTION FORMATS

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

Assembly-Language Format

Throughout most of this manual, the instructions are referred to by their assembly-language mnemonics. For example, the add ordinal instruction is referred to as the **addo** instruction.

Examples in the manual use the assembly language syntax of the Intel 80960 assembler, consisting of the instruction mnemonic, followed by zero to three operands, separated by commas. Following is an example of an assembly-language statement for the **addo** instruction:

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

In this example, the ordinal operands in global registers g5 and g9 are added together, and the result is stored in g7.

In the assembly-languages listing in this chapter registers are denoted as shown earlier in this manual, a "g" denotes a global register, an "r" denotes a local register, and an "sf" denotes a special function register. A "#" sign denotes a comment. All numbers used as literals or in address expressions are assumed to be decimal. Hexadecimal numbers are denoted with a 0x prefix (for example, 0xffff0012). Here are several other examples of assembly-language instruction statements:

```
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
```

Additional assembly-language examples are given in *Chapter 3, Data Types and Addressing Modes*. Further information about the assembly-language syntax can be found in the *Intel 80960 Assembler Manual.*

Branch Prediction

Since the actions of branch instructions are dependent 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 opword. The Intel 80960 Assembler encodes the prediction with a mnemonic suffix of ".t" for true and ".f" for false. Use the .t suffix to speed up execution when an instruction usually takes a branch. Use the .f suffix to speed up execution when an instruction usually does not take a branch.

Because test and conditional-fault instructions also use the condition codes, prediction suffixes are also implemented on these instructions. See *Appendix B, Optimizing Code for the 80960CA* for a complete discussion of prediction.

Note: Branch prediction is an implementation-specific feature of the 80960CA. Not every implementation of the 80960 architecture will use the branch prediction bit.

Instruction-Encoding Formats

All instructions are encoded in one 32-bit opword, which must be word aligned in memory. The most significant eight bits of an opword contain the opcode field. The opcode field determines the instruction to be performed and how the remainder of the opword is interpreted. Instructions are encoded in opwords in one of four formats: register (REG); compare and branch (COBR); control (CTRL), and memory (MEM). (See Figure 4-1.)

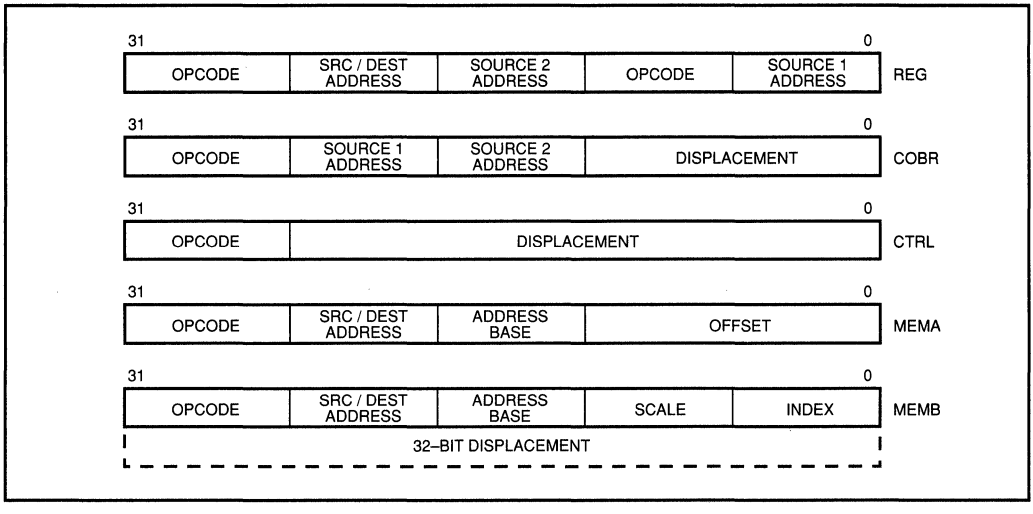


Figure 4-1. Machine-Level Instruction Formats

Most instructions are encoded in the REG format. This format is used primarily for instructions which perform register-to-register operations. The CTRL format is used for branches and calls that do not depend on registers for address calculation. The COBR format is 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.

The MEM format is used for referencing an operand which is a memory address. The load and store instructions, and some branch and call instructions use this format. The MEM format has two encodings: MEMA or MEMB. The MEM format which is used 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. (The instruction encoding formats are described in detail in *Appendix D, Instruction Encoding Reference.*)

Instruction Operands

The REG format has three possible operands: *src1*, *src2*, and *src/dst*. The *src1* and *src2* operands can be global registers, local registers, special-function registers, or literals. The *src/dst* operand is either a global, local or special function register.

The CTRL format is used for branch and call instructions. This format has only one operand, a *displacement* value that indicates the target instruction of the branch or call.

The COBR format has two source operands (*src1* and *src2*) to indicate the values to be compared, and a *displacement* operand to indicate the branch target. The *src1* operand can specify a global register, local register, special-function register, or a literal. The *src2* operand can specify a global, local or special-function register. (Refer to *Chapter 2, Programming Environment* for a detailed discussion of special-function registers.)

The MEM format specifies a source or destination register and an effective address formed by using any of the processor's addressing modes described in *Chapter 3, Data Types and Memory Addressing Modes*. The registers specified in a MEM format instruction must be either a global or local register.

INSTRUCTION GROUPS

The 80960 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 different instructions are

provided to handle different operand sizes, data types, or branch conditions. The following sections give a brief overview of the instructions in each of these groups.

Table 4-1. Summary of the 80960CA Instruction Set

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

Note: Instructions marked by (*) are 80960CA extensions to the 80960 instruction set.

DATA MOVEMENT

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

Load

The load instructions (listed below) copy bytes or words from memory to local or global register, or to a group of registers:

ld	load word
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

All of these instructions use the MEM format.

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

The **ld**, **ldl**, **ldt**, and **ldq** instructions copy 4, 8, 12, and 16 bytes from memory into successive registers.

Store

For each load instruction there is a corresponding store instruction (listed below), which copies bytes or words from a selected local or global register, or group of registers, to memory:

st	store word
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

All of these instructions use the MEM format.

For the **st**, **stob**, **stos**, **stib**, and **stis** instructions, a register and memory address are specified in the instruction, and the value in the register is copied into memory.

For the byte and short instructions, the processor automatically reformats the source register's 32-bit value for the shorter memory location. For the **stib** and **stis** instructions, this reformatting can lead to integer overflow if the register value is too large to be represented in the shorter memory location. When integer overflow occurs, either an integer-overflow fault is generated, or the integer-overflow flag in the arithmetic-controls register is set, depending on the setting of the integer-overflow mask bit in the arithmetic controls register. For the **stob** and **stos** instructions, the processor truncates the operand, and will not create a fault if the truncation resulted in the loss of significant bits.

The **st**, **stl**, **stt**, and **stq** instructions copy 4, 8, 12, and 16 bytes from successive registers into memory.

Move

The move instructions (listed below) copy data from a local, global or special-function register, or group of registers, to another register or group of registers.

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

These instructions use the REG format.

Load Address

The **lda** instruction 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 80960CA, the **lda** instruction is useful for performing simple arithmetic operations. The parallelism of the 80960CA allows an **lda** instruction to execute in the same clock as another arithmetic or logical operation.

ARITHMETIC

Table 4-2 lists the arithmetic operations and data types for which the 80960CA provides instructions. An "X" in this table indicates that the 80960CA provides an instruction for the specified operation and data type. The extended shift-right operation is an 80960CA extension to the 80960 instruction set. All arithmetic operations are carried out on operands in registers. (Refer to the section titled *Atomic Instructions* later in this chapter for the instructions which handle the 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 the arithmetic instructions for ordinal and integer data types.

Table 4-2. Arithmetic Operations

Data Types	Integer	Ordinal
Arithmetic Operations		
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	

Note: Instruction marked with (*) are 80960CA extensions 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

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

The **addc** instruction adds two word operands (literals, or contained in registers) plus bit 1 of the condition code (used here as a carry bit) in the arithmetic-controls register. If the result has a carry, bit 1 of the condition code is set; otherwise, it is cleared. The description of this instruction in *Chapter 9* gives an example of how this instruction can be used to add two long-word (64-bit) operands together.

The **subc** instruction is similar to the **addc** instruction, except it is used to subtract extended-precision values.

Although the **addc** and **subc** instructions 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.

The **emul** instruction multiplies two ordinals (each contained in a register), producing a long ordinal result (stored in two registers). The **ediv** instruction 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 the **remi** and **remo** instructions, the result has the same sign as the dividend; for the **modi** instruction, 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 to the left or to the 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 the bits shifted beyond the register boundary.

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

The **shli** instruction 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.

The **shri** instruction 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. (The discarding of the bits shifted out has the effect of rounding the result toward negative.)

The **shrdi** instruction 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.

The **shli** and **shrdi** instructions are equivalent to **muli** and **divi** by the power of 2, respectively.

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

The **eshro** instruction is an 80960CA extension to the 80960 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 the equivalent of 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	NOT (*src2* OR *src1*)
nor	*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*)

All of these instructions 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 the storage and creation of bitwise masks.

BIT AND BIT FIELD

These instructions perform operations on a specified bit, or field of bits, in an ordinal operand. All of these instructions 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

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

The **alterbit** instruction 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.

The **chkbit** instruction (described later in this chapter in the section titled *Comparison*) can be used to check the value of an individual bit in an ordinal.

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

Bit-Field Operations

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

The **modify** instruction copies bits from one register, under control of a mask, into another register. Only the unmasked bits in the destination register are modified. This instruction is equivalent to a bit-field move.

BYTE OPERATIONS

The **scanbyte** instruction 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 arithmetic-controls register according to the results of the comparison.

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

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

The **cmpi** and **cmpo** instructions simply compare the two operands and set the condition-code bits accordingly.

The **concmpi** and **concmpo** instructions first check the status of bit 2 of the condition code. If it is not set, the operands are compared as with the **cmpi** and **cmpo** instructions. If bit 2 is set, no comparison is performed, and the condition-code bits are not changed.

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

The **chkbit** instruction checks a specified bit in a register and sets the condition-code bits according to the state of the bit. 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 bits 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 instructions cause a TRUE (01H) to be stored in a destination register if the condition code matches the condition specified with the instruction. Otherwise, a FALSE (00H) is stored in the register. These instructions use the COBR format and can operate upon local, global, and special-function registers.

Since the actions of the test instructions are dependent upon 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. The Intel 80960 Assembler encodes the prediction with a mnemonic suffix of ".t" for true and ".f" for false. (See *Appendix B, Optimizing Code for the 80960CA* for a complete discussion of branch prediction.)

BRANCH

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

- unconditional branch
- conditional branch
- compare and branch

Most of the branch instructions specify the target IP by specifying a signed displacement to be added to the current IP. Other branch instructions specify the memory address of the target IP, 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 the actions of branch instructions are dependent upon 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 B, Optimizing Code for the 80960CA* 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

The **b** and **bal** instructions use the CTRL format. The **bx** and **balx** instructions use the MEM format and can specify local or global registers as operands.

The **b** and **bx** instructions 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.

The **bal** and **balx** instructions store the address of the next instruction in a specified register, then jump to the specified target IP. (For the **bal** instruction, the RIP is automatically stored in register g14; for the **balx** instruction the location of the RIP 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 condition-code flags in the arithmetic-controls register. If these bits 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

These instructions use the CTRL format. The **bo** and **bno** instructions are used when working with real numbers. (Refer to *Chapter 2, Programming Environment* for a discussion of meanings of the condition code for conditional operations.)

Compare and Branch

The compare and branch instructions compare two operands, then branch according to the result of the comparison. There are three subtypes of instructions in this group: 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 of these instructions use the COBR machine-instruction format and can specify literals, local, global, and special-function registers as operands.

With the compare-ordinal-and-branch and compare-integer-and-branch instructions, two operands are compared, and the condition-code bits are set, as with the compare instructions described earlier in this chapter. A conditional branch is then executed as with the conditional branch (branch if) instructions.

With the check-bit-and-branch instructions, one operand specifies a bit to be checked in the other operand. The condition-code bits are set according to the state of the specified bit (i.e., 010_2 if the bit is set, and 000_2 if the bit is clear). A conditional branch is then executed according to the setting of the condition-code bits.

These instructions are provided as a performance optimization. When it is not possible to separate adjacent compare and branch instructions with other unrelated instructions, replacing the two instructions with the single compare-and-branch instruction will increase 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 detail 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

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

The **call** and **callx** instructions make local calls to procedures. A local call is a call that does not require a switch to another stack. The **call** and **callx** instructions 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 IP of the **call**. The **callx** instruction 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.

The **calls** instruction is used to make calls to system procedures (procedures that provide a kernel or system-executive services). This instruction operates similarly to the **call** and **callx** instructions, 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, the **calls** instruction 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. (The supervisor mode is described in detail in *Chapter 5, Procedure Calls.*)

The **ret** instruction performs a return from a called procedure to the calling procedure (the procedure that made the call). This instruction obtains its target IP (return IP) from linkage information that was saved for the calling procedure. The **ret** instruction 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 the various types of faults without explicit intervention by the currently running program. (Faults are discussed in detail in *Chapter 7, Faults.*)

The following conditional-fault instructions permit a fault to be generated explicitly (by the program) according to the state of the condition-code bit.

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 of these instructions 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 B, Optimizing Code for the 80960CA* 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:

modtc	modify trace controls
mark	mark
fmark	force mark

These three instructions use the REG format.

The 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 detail in *Chapter 8, Tracing and Debug.*)

The **modtc** instruction permits the trace controls to be modified.

The **mark** instruction causes a breakpoint trace event to be generated if the breakpoint trace mode is enabled. The **fmark** instruction generates a breakpoint trace independent of the state of the breakpoint trace-mode bits.

The 80960CA-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 80960CA's breakpoint registers. (See *Chapter 8, Tracing and Debug* for a detailed description of breakpoints.)

ATOMIC INSTRUCTIONS

The 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 will be completed 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.

There are two atomic instructions: atomic add (**atadd**) and atomic modify (**atmod**). The **atadd** instruction causes an operand to be added to the value in the specified memory location. The **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 processor provides the following instructions for use in controlling processor-related functions.

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

All of these instructions use the REG format and can specify literals, or local, global, or special-function registers.

The **modpc** instruction provides a method of reading and modifying the contents of the process-controls register. Only programs operating in supervisor mode may modify the process-controls register; although, any program may read the register.

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 arithmetic-controls register to be copied to a register, and/or be modified under the control of a mask. The arithmetic-controls register cannot be explicitly addressed with any other instruction, although instructions that use the condition codes, or set the integer-overflow flag, implicitly access the arithmetic-controls register.

The **sysctl** instruction is an 80960CA-specific extension to the 80960 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. the sysctl instruction may only be executed by programs operating is supervisor mode. (See *Chapter 2, Programming Environment* for a complete description.)

The **sdma** and **udma** instructions are the 80960CA-specific extensions to the 80960 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 detailed description of these instructions.)

Procedure Calls 5

CHAPTER 5
PROCEDURE CALLS

This chapter describes the mechanisms for making procedure calls, which include the branch-and-link instructions, the built-in call and return mechanism, the call instructions (**call**, **callx**, **calls**), return instruction (**ret**), and the call actions caused by interrupts and faults.

OVERVIEW

The 80960 architecture supports two methods for making procedure calls: a RISC-style branch-and-link and an integrated call and return mechanism. A branch-and-link is a fast call best suited for calling procedures that do not call other procedures. The call and return mechanism 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** and **balx** instructions), 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** instructions 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 80960 architecture provides a fully-integrated call 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 80960, 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 mechanism is used in two ways in the 80960 architecture: 1) explicit calls to procedures in a user's program, and 2) implicit calls to interrupt and fault handlers. The remainder of this chapter explains the generalized call mechanism used for explicit and implicit calls, and the call and return instructions.

There are two call actions which the processor performs: local and supervisor. When a local call is made, the 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. When a supervisor call is made, the 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. The local call instructions, **call** and **callx**, perform a local call action. With the **call** and **callx** instructions, the IP for the called procedure is included as an operand in the instruction.

A system call is made with the **calls** instruction. This instruction is similar to the **call** and **callx** instructions, except that the processor obtains the IP for the called procedure from the *system-procedure table*. A system call, when executed, is directed to perform either the local or the 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 80960 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 current local register set is saved by the processor, 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 the 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. The 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 the speed of procedure calls, the architecture allows an implementation to cache 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 of the relationship of the local registers and the procedure stack.

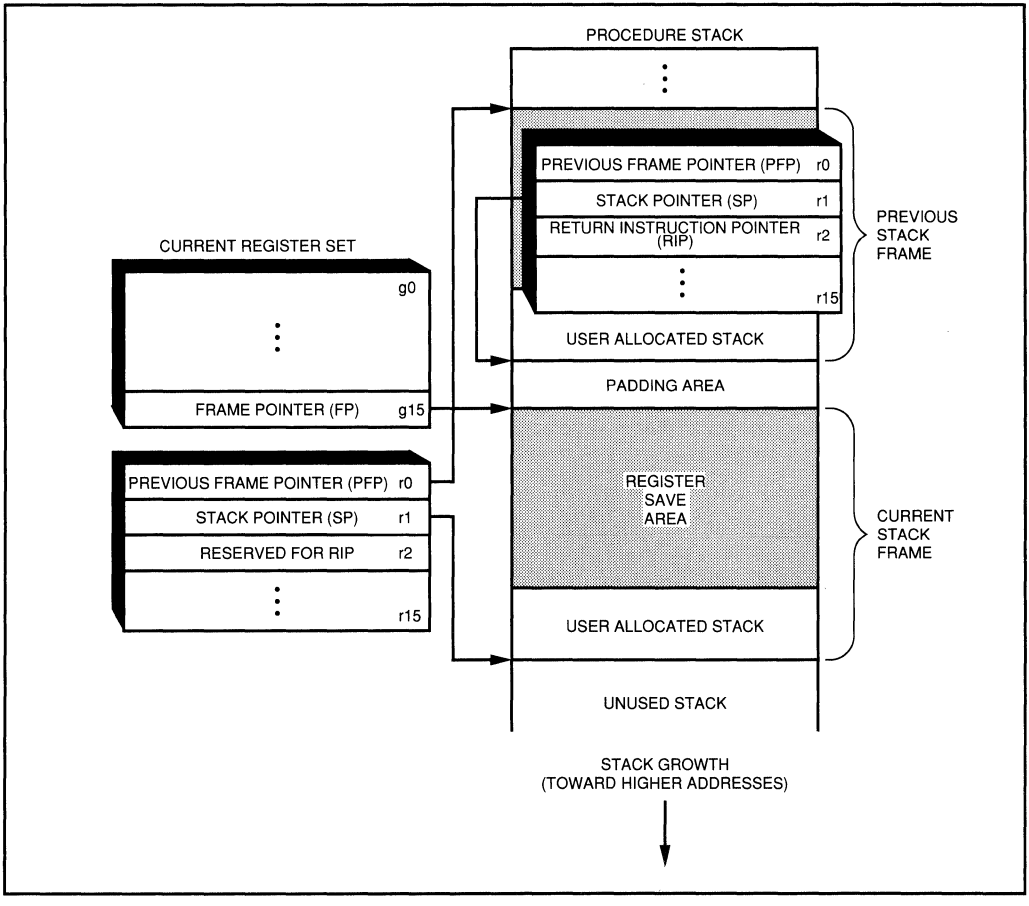


Figure 5-1. Procedure Stack Structure and Local Registers

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 to link the local registers to the procedure stack (Figure 5-1). The following paragraphs describe this linkage information.

Frame Pointer

The frame pointer is the address of the first byte of the current stack frame. It is stored in global register g15, the frame pointer (FP) register. The FP register is always reserved for the frame pointer and should not be used for general storage. In the 80960CA, 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.

Note: The alignment of stack frames is defined for each implementation of the 80960 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 80960CA, SALIGN = 1.)

Stack Pointer

The stack pointer is the byte aligned address of the next unused byte of the stack frame. The value of the stack pointer 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 value of the frame pointer 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 value of the SP register when data is stored or removed from the stack. The 80960 does not provide an explicit push or pop instruction to perform this action.

Previous-Frame Pointer

The previous-frame pointer is the address of the first byte of the previous stack frame. The upper 28 bits of this address is 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

Bits 0 through 3 of the PFP register 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 describe 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 and should not be used 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.

Call Operation

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

1) The processor stores the current instruction pointer in the RIP register of the current stack.
2) The current local register set is saved, freeing these registers for use by the called procedure. Recall that the local registers are saved in the on-chip register cache.
3) The frame pointer for the calling procedure is stored in the PFP register.
4) The return-type field in the PFP register is set according to the type of call which is performed. (See *Returns* in this chapter.)
5) A new stack frame is allocated by using the value of the stack pointer which was saved in step 2. This value is first rounded to the next 16-byte boundary to create a new frame pointer, and is stored in the FP register. Next, 64 bytes is added to create the register save area for the new frame. This value is stored in the stack pointer register.
6) 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 of the types of calls (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 register are usually read from the local register cache; however, in some cases, these registers have been flushed from the 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 begins executing the procedure returned to.

Caching of Local Register Sets

The 80960 architecture supports on-chip caching of local-register sets as a means of optimizing the call and return actions. Movement of data between the register cache and the local register set is typically accomplished in only four clocks with no external bus traffic. The other overhead associated with the call or return action is performed in parallel with this data movement. The result is a high-performance implementation of these common operations.

The size of the register cache is specified at initialization by the value of the register cache configuration word in the PRCB. (See *Chapter 14, Initialization and System Requirements* for more information about initialization and the PRCB.)

Up to five local register sets are cached internally. When the cache is configured for six or more register sets, part of the internal data RAM is used to expand the cache. Each additional set consumes 64 bytes of internal RAM beginning at the highest address of internal RAM (03FFH) and growing downward. The user program is responsible for preventing any corruption to the areas of internal RAM which have been set aside for the register cache.

When a call is made and the register cache is full, a register set in the cache must be flushed to external memory to make room for the current set of local registers. For this purpose, a save area for the local registers is reserved when a stack frame is created.

The register cache is a first-in-first-out (FIFO) cache. In other words, the first register set cached is the first register set which will be flushed to the save area in memory. In a typical program, most procedure calls and returns cause procedure depth to oscillate a few levels around a fixed, median call depth. A FIFO cache tends to be partially filled at the median call depth. Cache flushes occur when the oscillations around the median depth are larger than the cache size can accommodate (Figure 5-2). Configuring the local register cache to hold five sets of local registers is suitable for most applications and does not use any of the data RAM which is available for general data storage. The user should always configure the cache for a minimum of five register sets.

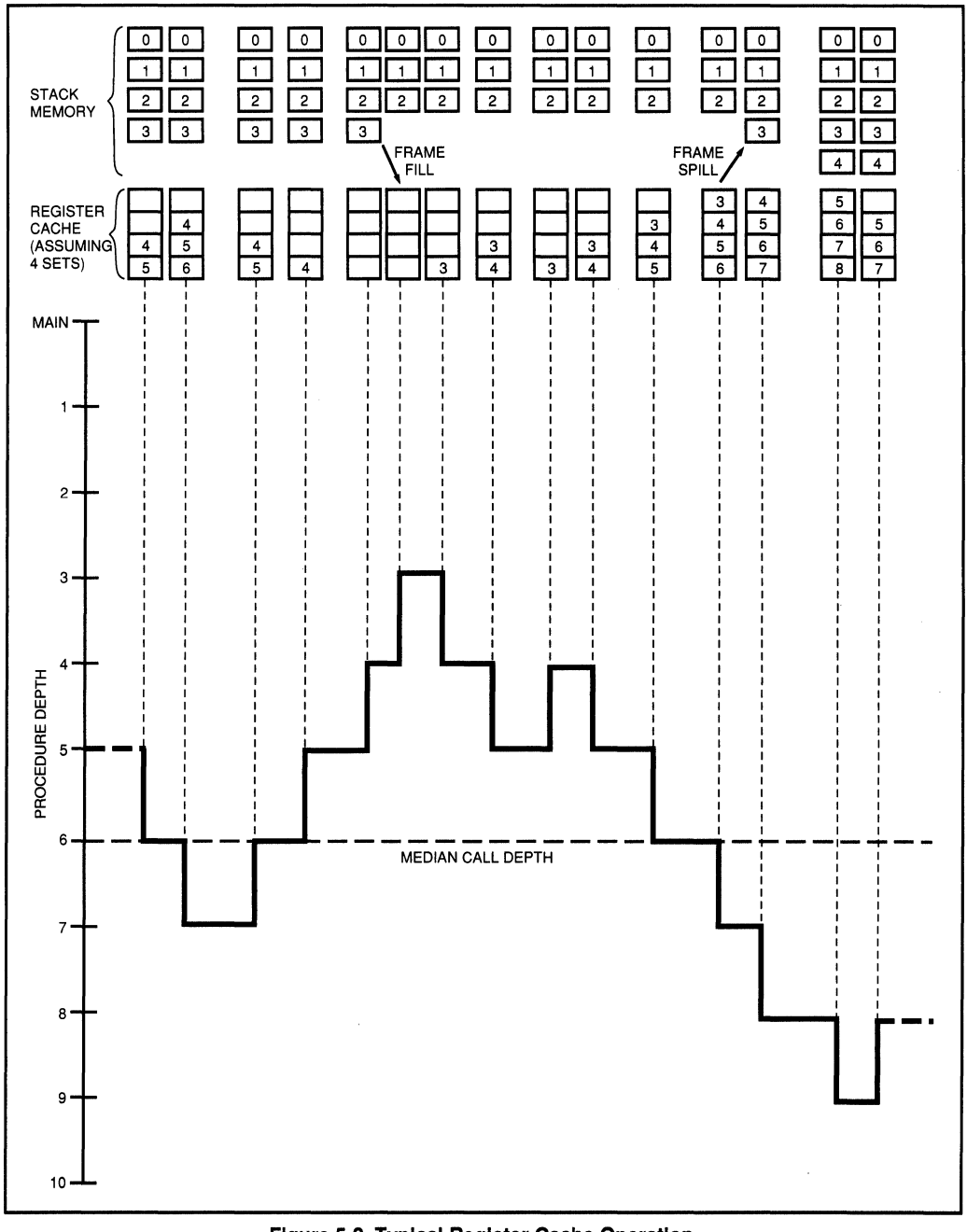


Figure 5-2. Typical Register Cache Operation

Mapping the Local Registers to the Procedure Stack

Each set of local registers is mapped to a register save area of its respective frame in the stack. As described, saved local register sets are frequently cached on-chip rather than saved to memory. This caching is performed non-transparently. In other words, the contents of the local register set will not be saved automatically to the save area in memory when the register set is cached. (This would cause a significant performance loss for calls.) Additionally, no automatic update policy for the register cache is implemented. If the register save area 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 has to have been flushed to memory (because of a cache spill) for the modification to be valid.

The processor provides the **flushreg** (flush local registers) instruction to allow explicit flushing of the local registers. This instruction causes the contents of all the local-register sets, except the current set, to be written to their associated stack frames in memory. The register cache is then invalidated, meaning that all of the flushed register sets will be restored from their save areas in memory. Occasionally, it is necessary to flush the register cache. For example, in debugging software, it may be necessary to trace the call history back through the nested procedures.

When a set of local registers is assigned to a new procedure, the processor may not clear or initialize these registers. The initial contents of these registers are therefore unpredictable. Also, the processor does not initialize the local register save area in the newly created stack frame for the procedure, so its contents are equally unpredictable.

PARAMETER PASSING

Parameters are passed between procedures by value and by reference. Passing parameters by value means that the parameters are passed directly to the calling procedure as part of the call and return mechanism. Passing parameters by reference means that the parameters are stored in an argument list in memory and a pointer is maintained to the argument list.

Parameter passing by value is performed with the global registers and is the fastest method of passing parameters. Here, 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 the 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 value of the SP register. 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 should agree on the use of the global registers. Table 5-1 summarizes the global register model used by the iC-960 C-language compiler. (Refer to the *iC-960 User's Guide* for a detailed discussion of the register allocation model.) This example is described to illustrate a typical implementation of parameter passing between procedures and the use of the global and local registers in this scheme.

Table 5-1. Global Register Function with iC-960 Compiler

Register	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, registers g8 through g11 must be preserved by the called procedure

The parameter registers pass values into a function. Up to 12 parameters are passed by value or reference in the global registers. If the number of parameters exceeds 12, the additional parameters are passed on the stack of the calling procedure, 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 the parameter passing by value and reference.

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 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, the global parameter registers are moved to working local registers at the beginning of the procedure. In this way, the parameter registers are freed, and the nested calls are easily managed. The register move

instruction necessary to perform this action is very fast, and the working parameters, now in local register, are saved efficiently when the nested calls are made.

2) When other procedures are nested within an interrupt or fault procedure, the procedure must preserve all of the normally non-preserved parameter registers. This must be done because the interrupt or fault occurs at any point in the user's program, and the return from the interrupt or fault must restore the exact state of the processor. The interrupt or fault procedure can move the non-preserved global registers to the local registers before the nested call.

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. (The system-local call is described in the following section titled *System Calls*.) The **call** instruction specifies the address of the called procedures as the IP plus a signed, 24-bit displacement (i.e., -2^{23} to $2^{23} - 4$). The **callx** instruction 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 the **call** and **callx** instructions.)

When a local call is made with a **call** or **callx** instruction, the processor performs the same operation as is 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. (The algorithms that the **call** and **callx** instructions use are described in greater detail in *Chapter 9, Instruction Set Reference.*)

SYSTEM CALLS

A system call is a call made through the system-procedure table. It can be used to make a system-local call (similar to a local call made with the **call** and **callx** instructions) or a system-supervisor call.

A system call is initiated with the **calls** instruction, 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 Instruction Reference for a further description of the **calls** instruction.)

Using the ASM-960 language assembler, a system procedure is directly declared using the .sysproc directive. At link time, the optimized call directive, callj, will be replaced with a **calls** when a system procedure target is specified. (See the *ASM-960 User's Guide* for a description of the .sysproc and the callj directives.)

The system-call mechanism offers two benefits. First, it supports portability for application software. 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 have to be changed each time the implementation of the kernel services is modified. Only the entries in the system-procedure table need to 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 described in more detail in *Chapter 2, Programming Environment.*)

System-Procedure Table

The system-procedure table is a data structure for storing IPs to system procedures. These procedures 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. (The use of the system-procedure table to store IPs for fault-handling procedures is described in *Chapter 7, Faults.*)

The structure of the system-procedure table is shown in Figure 5-3. 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 the fields in this table.

Procedure Entries

A procedure entry in the system-procedure table specifies the location of a procedure and its type. Each entry is one word in length and is made up 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 30 most-significant bits of the entry are used for the address. The two least-significant bits of the entry are used to specify the entry type. The procedure-entry type field indicates the 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

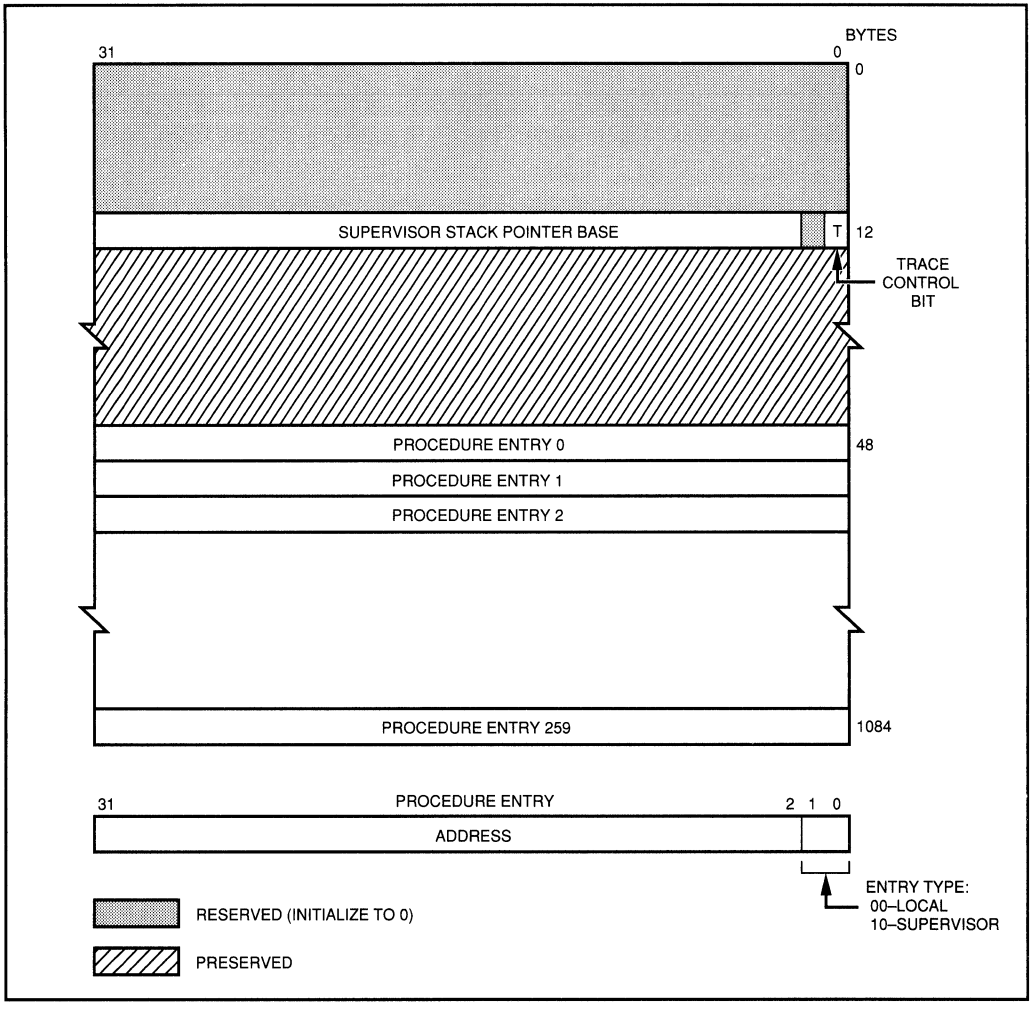


Figure 5-3. System-Procedure-Table

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 5-3). 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 process controls 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 was described earlier in this chapter in the section titled *Call Operation*. The target IP for the call is taken from the system-procedure table, and the new stack frame is allocated on the current stack. (The algorithm that the **calls** instruction uses is described in greater detail 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 target IP for the call is taken from the system-procedure table. The processor performs the same action as is described earlier in this chapter in the 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 process-controls 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 the supervisor mode, either the local call instructions (**call** and **callx**) or the **calls** instruction can be used to call procedures.

(The user-supervisor protection model and its relationship to the supervisor call are described in detail 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 the user mode, and the other stack (the supervisor stack) is for procedures executed in the supervisor mode.

The structure of the user stack and supervisor stack is identical (Figure 5-1). The base stack pointer for the supervisor stack is read from the system-procedure table each time a user-to-supervisor mode switch occurs. 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 should be aligned to a 16-byte boundary. If they are not aligned, the first frame pointer in the stack is rounded up to the next 16-byte boundary.

INTERRUPT AND FAULT CALLS

The architecture defines two type 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 5-4) to determine the type of return action to take. The return-type field is described below.

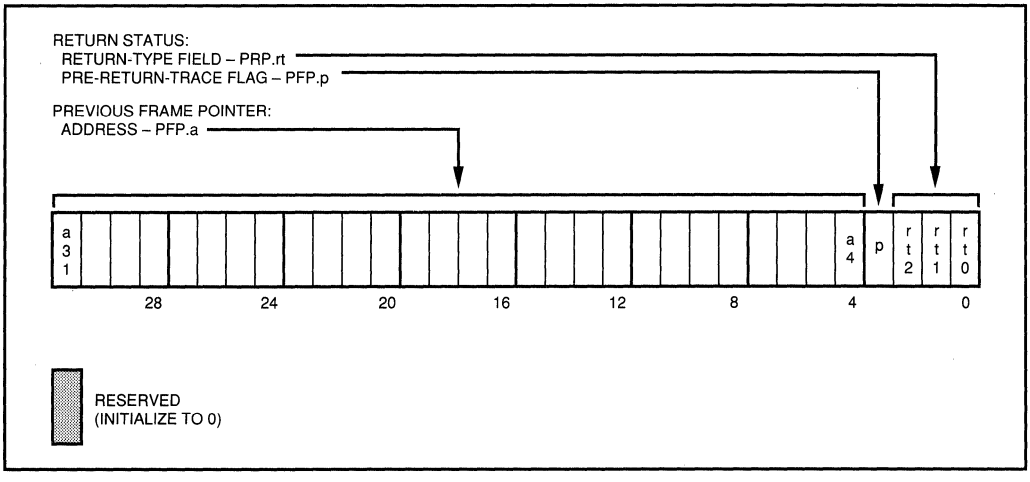


Figure 5-4. PFP Register and the Return Status Field

The *return-type field* indicates the type of call which was made. Table 5-3 shows the encoding of the return-type field for the various calls: local call; supervisor call; interrupt call; and fault call.

The *trace-on-return flag* stores the value of the trace enable bit when a system-supervisor call is made from user mode. When the call is made, the trace-enable bit in the PC register 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 original value of the trace-enable bit 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*.)

The *prereturn-trace flag* (PC.p) is used in conjunction with the call-trace and prereturn-trace modes. If the 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 the 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 detailed discussion of the interaction of the call-trace and prereturn-trace modes with the prereturn-trace flag.)

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 the 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	Interrupt return (See *Chapter 6, Interrupts.*)
p101	reserved	
p110	reserved	
p111	Interrupt call	

Note: "p" is PFP.p (prereturn trace flag)

BRANCH-AND-LINK

A branch-and-link is executed using either the branch-and-link instruction (**bal**) or the branch-and-link-extended instruction (**balx**). When either of these instructions 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 the **bal** instruction, the return IP is automatically saved in global register g14; for the **balx** instruction, 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 the 80960CA's interrupt-handling facilities. Discussed are: descriptions of the data structures used for interrupt handling; the actions that the processor takes when handling an interrupt; the facilities for requesting and posting interrupts; the programmer's interface to the on-chip interrupt controller; the implementation of the interrupt controller; and interrupt latency.

OVERVIEW

An interrupt is an event that causes a temporary break in the execution of a program so that the processor can handle another chore. Interrupts are commonly used to request I/O services or to synchronize the processor with the activities of external hardware. To allow interrupt handlers to be portable across 80960 implementations, the architecture defines a consistent interrupt state and interrupt-priority-handling mechanism. To manage and prioritize interrupt requests in parallel with processor execution, the 80960CA provides an on-chip programmable interrupt controller.

Requests for interrupt service may 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 the priority 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 an entry in a data structure called the interrupt table. 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 register is allocated to the interrupt procedure. The state of the interrupted program is also automatically saved when the call is made.

On the return from the interrupt procedure, the processor restores the state of the interrupted program, switches back to the stack that the processor was using prior to the interrupt, and resumes execution of the program.

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.

On the 80960CA, 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 an instruction (**sysctl**) which signals interrupts. To manage and prioritize all possible interrupts, the 80960CA integrates an on-chip programmable interrupt controller.

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 both 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, and 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 is used to post hardware-requested interrupts. The interrupt mask (IMSK) special function register is used to selectively mask hardware-requested interrupts.

THE 80960 INTERRUPT MODEL

The architecture defines two data structures to support interrupt processing: the interrupt table and the interrupt stack. These data structures are shown in Figure 6-1. The interrupt table contains 248 vectors to interrupt-handling procedures and an area for posting software-requested interrupts. The interrupt stack is provided to prevent interrupt-handling procedures from overwriting the stack being used 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).

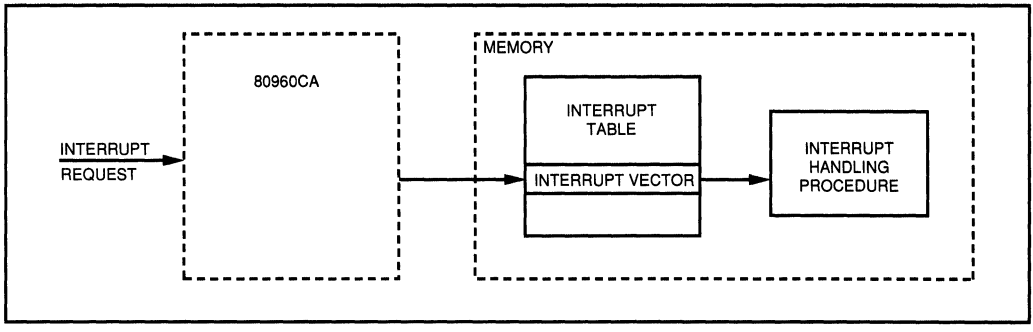


Figure 6-1. Interrupt-Handling Data Structures

Interrupt Priority

To provide transparent prioritization of the 248 possible interrupts, the 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, which is assigned the lowest priority, to vector number 255, which is the highest priority. Since there are 31 priority levels, the priority of each vector is determined by the upper five bits of the vector number. 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 priority of an interrupt request to determine whether to service the interrupt immediately or to delay service. If the priority of the interrupt request is higher than the processor's current priority (the priority of the program or interrupt the processor is executing), the interrupt is serviced immediately. 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 that are waiting to be serviced are called *pending interrupts*, and are discussed later in this chapter.

NOTE: On the 80960CA, the non-maskable interrupt (NMI) will interrupt priority-31 execution, and no interrupt will interrupt an NMI handler.

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

Interrupt Table

The interrupt table (Figure 6-2) is 1028 bytes in length. It can be located anywhere in the non-reserved address space, but it must be aligned on a word boundary. The processor reads a pointer to byte 0 of the interrupt table during initialization. The interrupt table must be located in RAM since the processor must be able to read and write the pending-interrupt section of the table.

The interrupt table is divided into two sections: the vector-entries section and the pending-interrupts section.

Vector Entries

The vector-entry section contains 248 one-word entries, one entry for each possible interrupt-handling procedure. Each interrupt request is associated with an 8-bit vector number which points to a vector entry in the interrupt table. Of the 256 possible values of the vector number, 0 through 7 are not defined and do not have associated entries in the interrupt table.

The vector entry contains the address of a specific interrupt handler. Vector entry 248 contains the address for the NMI handler. Vectors entries 244 through 247 and 249 through 251 are reserved and should not be used.

The structure of a vector entry is given at the bottom of Figure 6-2. Each interrupt procedure must begin on a word boundary, so the processor assumes that the two least-significant bits of the vector are 0. Bits 0 and 1 of an entry indicate the entry type. Entry type 00_2 indicates that the interrupt procedure should be fetched normally. The 80960CA also supports an entry type of $1X_2$, which indicates that the interrupt procedure should be fetched from the locked partition of the instruction cache. (See *Caching of Interrupt-Handling Procedures* in this chapter.) The other possible entry types are reserved and must not be used.

Pending Interrupts Section

The pending-interrupts section comprises the first 36 bytes of the interrupt table. This section is divided into two fields: pending priorities (byte-offset 0 through 3) and pending interrupts (byte-offset 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 256 bits in the pending-interrupts field represents 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 to determine the vector number of the interrupt with the highest priority.

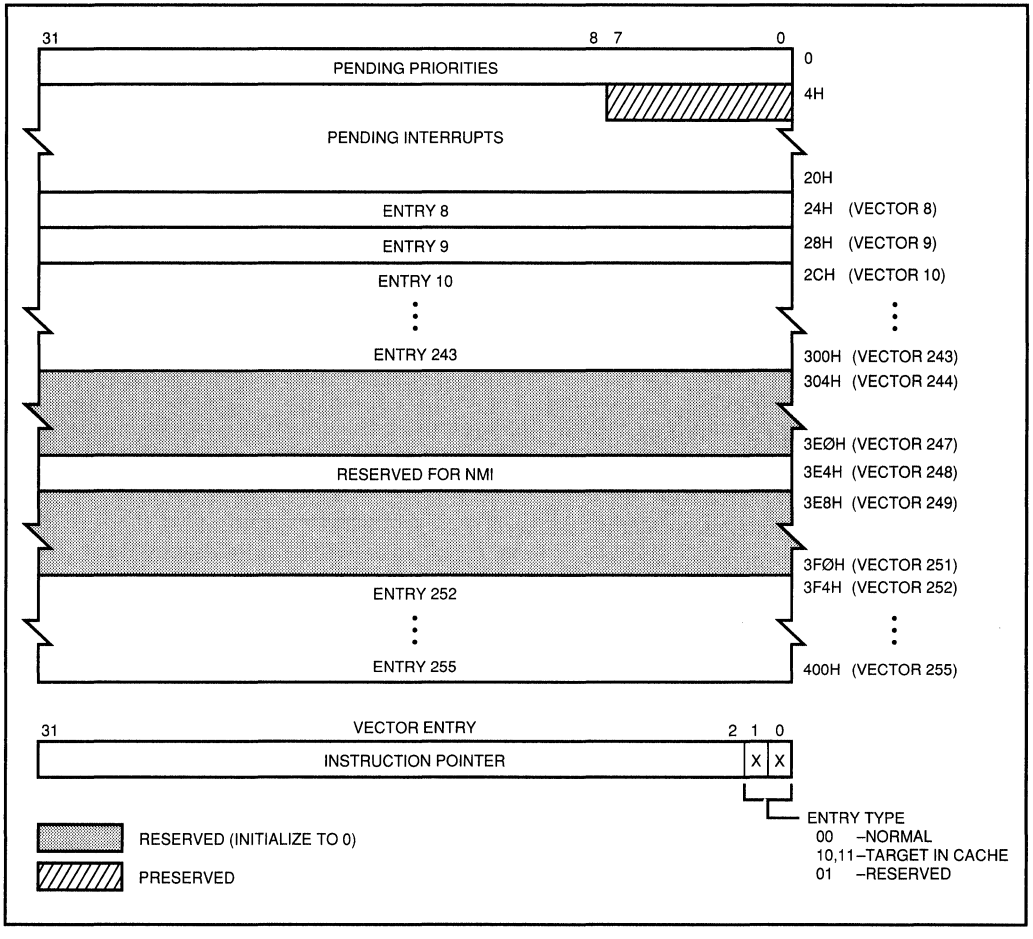


Figure 6-2. Interrupt Table

Posting Interrupts In the Interrupt Table

For the 80960CA, only interrupts which are requested by software are posted in the interrupt table. Hardware-requested interrupts are posted in the IPND register. The IPND register and the mechanism for requesting and posting hardware interrupts is described later in this chapter in the section titled *Managing Interrupt Requests*. 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 a system that uses two or more processors, both processors can post and service interrupts from a shared interrupt table. This interrupt-table sharing allows the processors to share the interrupt-handling load, or provide a communication mechanism between the processors.

To post a pending interrupt in the interrupt table in memory, the processor must perform the following atomic read/write operation that locks the interrupt table until the posting operation has been 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. Posting software interrupts is performed by the **sysctl** instruction on the 80960CA. (See *Software-Generated Interrupt Requests* later in this chapter).

Note: The 80960 architecture does not define a portable method for posting interrupts. Different implementations may implement optimized interrupt-posting mechanisms. The 80960CA records pending interrupts differently depending upon the type of interrupt and the configuration of the interrupt controller. (See the sections in this chapter titled Interrupt Modes, and Software-Generated Interrupts for more detail.)

An external I/O agent or a coprocessor can post pending interrupts to the interrupt table in memory in the same manner described above, providing it has the capability to perform atomic operations on memory.

In some implementations, software may be able to post a pending interrupt explicitly in the interrupt table in memory, using the following algorithm:

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

However, this method of posting an interrupt will only work if the application guarantees that: 1) no external I/O agent is allowed to post a pending interrupt simultaneously with the software, and 2) an interrupt cannot occur after one bit (e.g., the pending priority bit) is set but before the other bit (e.g., the pending priorities bit) is set.

The processor automatically checks the memory-based interrupt table for pending interrupts when executing a modify-process-controls instruction (**modpc**), if the instruction caused the program's priority to be lowered.

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.

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 80960CA caches the following:

- The value of the highest priority posted in the pending priorities field.
- A predefined subset of the interrupt vectors (that is, interrupt vector entries from the interrupt table).
- The pending interrupts that are received from the external interrupt pins and the 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*). (The facilities for caching vectors are described in greater detail in the sections of this chapter titled *Vector Caching Option* and *Interrupts Mask and Pending Register*.)

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 7, 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.

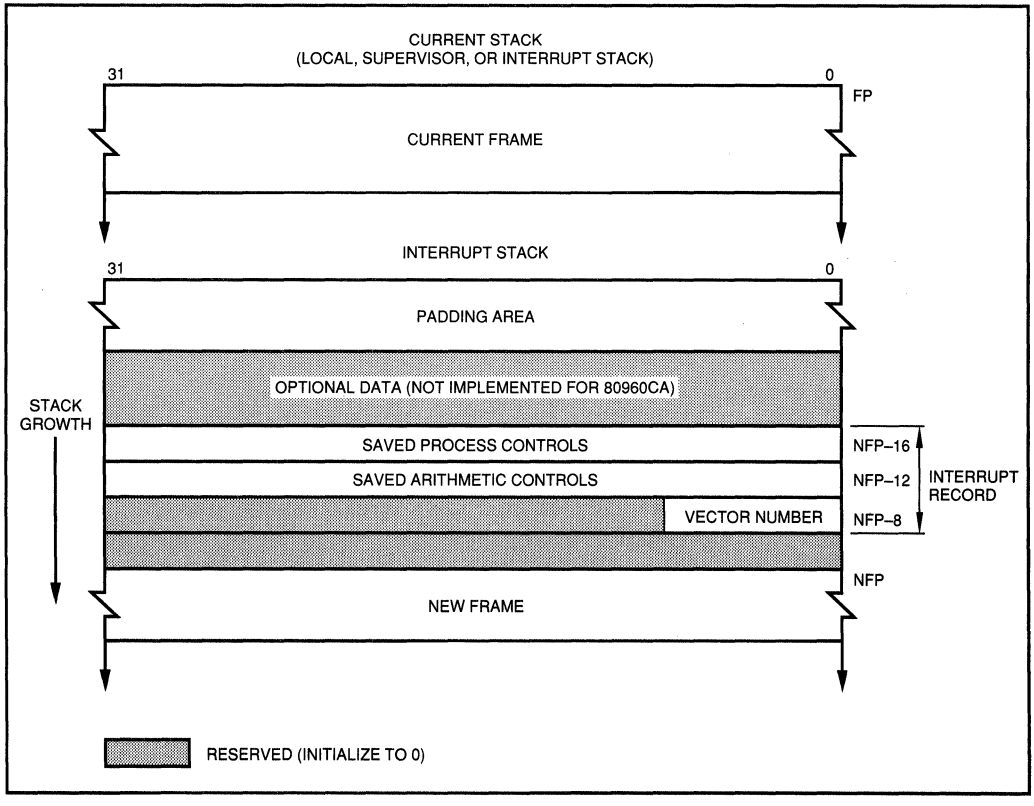


Figure 6-3. Storage of an Interrupt Record on the Interrupt Stack

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 arithmetic-controls and process-controls 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 arithmetic-controls register is located at address NFP-12, and the saved process-controls register is located at address NFP-16.

Note: The interrupt record may also contain a resumption record which stores the context of instructions which had begun, but were not completed when the interrupt was serviced. Although the 80960CA 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 job for an interrupt handler 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 80960 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 arithmetic-controls and process-controls registers for the interrupted program. The interrupt procedure shares the remainder of the execution-environment resources (namely the global registers, special function registers and the address space) with the interrupted program. Thus, interrupt procedures must preserve and restore the state of any resources shared with a non-cooperating program. For example, an interrupt procedure 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 the interrupt latency to critical interrupt routines, interrupt handlers may be locked into the instruction cache. (See the section of this chapter titled *Caching of Interrupt-Handling Procedures* for a complete description.)

Interrupt Context Switch

When the processor services an interrupt, it automatically saves the state of the interrupted program (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 state the processor is in when it receives the interrupt. 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 the interrupt-handling actions for an executing-state interrupt and an interrupted-state interrupt. In both cases, it is assumed that the interrupt priority is higher than that of the processor, and thus will be serviced immediately after the processor receives it.

Executing-State Interrupt

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

- The processor performs the actions of a call, as described in the *Chapter 7, Procedure Calls*, with the exception that the new frame pointer (FP) for the interrupt handler is set to point to the interrupt stack, and is incremented by 16 to create space for an interrupt record (as shown in Figure 6-3). (The interrupt record is described earlier in this chapter in the section titled *Interrupt Record.*)
- The current state of the arithmetic-controls register, the process-controls register, and the interrupt vector number are saved in the interrupt record.
- The processor gets the IP for the first instruction in the called interrupt-handling procedure from the selected entry in the interrupt table.
- The processor stores the interrupt-return (111_2) in the return-status field of the new local-register r0, then changes the following fields and flags in the process-controls register:
  - Sets the state flag (bit 13) to interrupted.
  - Sets the execution-mode flag (bit 1) to supervisor (and switches to supervisor mode).
  - Sets the 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 can not interrupt the servicing of the current interrupt.)
  - Sets the trace-fault-pending flag (bit 10) and the trace-enable bit (bit 0) to 0. (Clearing these bits allows the interrupt to be handled without trace faults being raised.)

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

- The arithmetic-controls and process-controls fields from the interrupt record are copied into the arithmetic-controls and process-controls registers, respectively. (Restoring the process-controls 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 the execution mode to user prior to the return, the process-controls register is not restored on the return.

- If there are pending interrupts that need to be handled (i.e., pending interrupts that are higher then the priority of the program being returned to), they 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 will be handled at this time.
- The processor then performs a return operation, as described in *Chapter 7, Procedure Calls.* This will cause the processor to switch back to the local stack or the supervisor stack (whichever one it was using when it was 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 receives an interrupt while it is servicing another interrupt, and the new interrupt has a higher priority than the interrupt currently being serviced, the current interrupt-handler routine is interrupted. Here, the processor performs the same action to save the state of the interrupted interrupt-handler routine, 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 use in servicing the new interrupt.

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

MANAGING INTERRUPT REQUESTS

The 80960 architecture provides a consistent interrupt model which is needed for interrupt handler compatibility between different 80960 implementations. The architecture, however, leaves the interrupt request management strategy to the specific 80960 implementations. In the 80960CA, the programmable on-chip interrupt controller transparently manages all interrupt requests (Figure 6-4). These requests originate from the 8-bit external interrupt pins ($\overline{\text{XINT7}}$-$\overline{\text{XINT0}}$); the non-maskable interrupt pin ($\overline{\text{NMI}}$); the four DMA controller channels; and the execution of the **sysctl** instruction.

The 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 on the 80960 (expanded mode). Dedicated-mode requests are posted in the Interrupt Pending Register (IPND). Expanded-mode requests are not posted by the processor.

The 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. The 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 requested in the instruction is serviced immediately, or posted in the pending interrupts section of the interrupt table, depending upon the current processor priority and the priority of the request. 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 priority of the processor. When a pending interrupt request is priority-31, or is higher than the processor priority, the core is interrupted. 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.

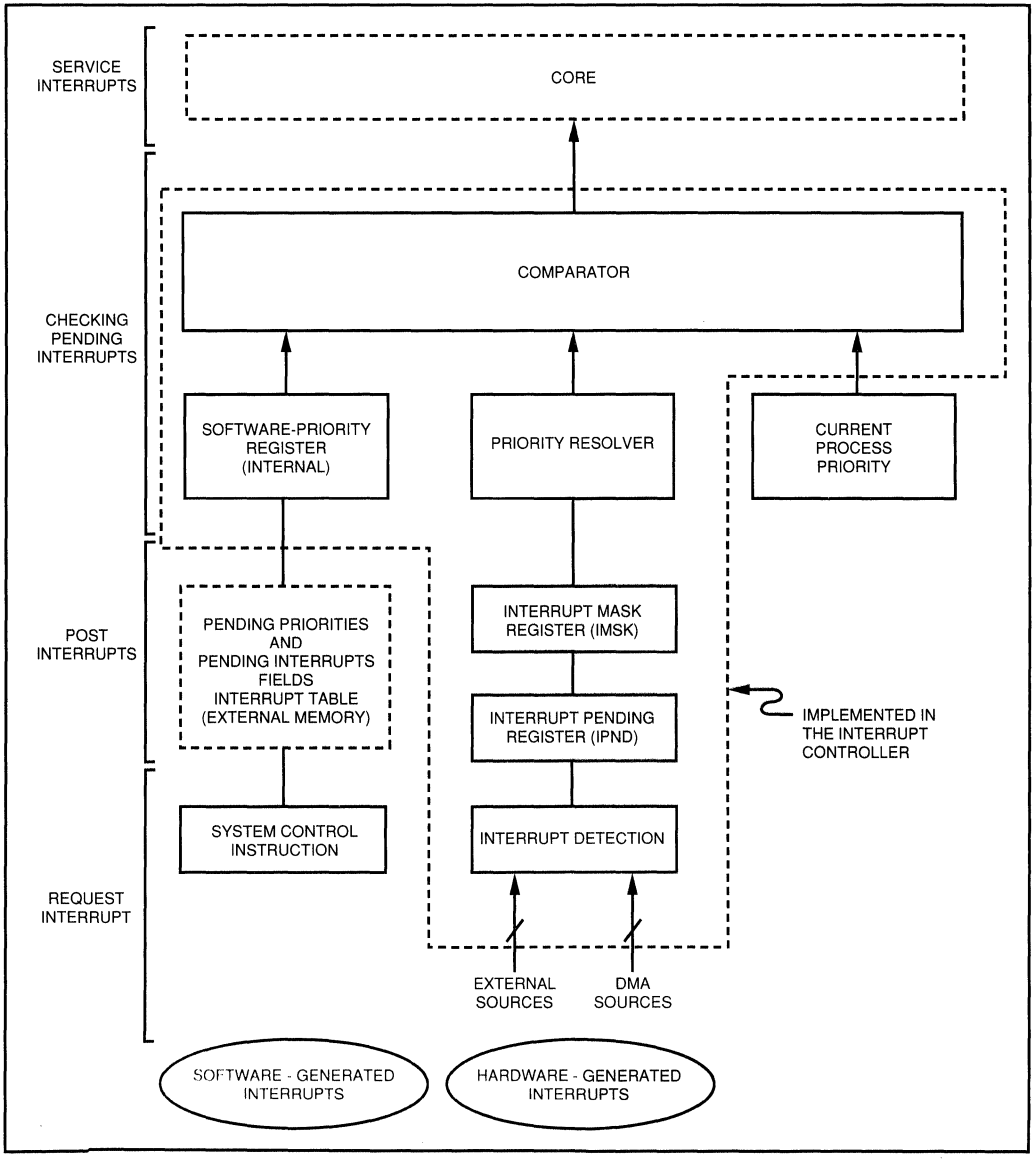


Figure 6-4. 80960CA Interrupt Controller

The following sections describe the interrupt-controller modes; the interrupt-request pins and inputs; the user's interface to the interrupt controller; the method for posting software-generated interrupt requests; and the methods for controlling interrupt latency.

Interrupt Controller Modes

The eight external interrupt pins can be configured for one of three modes: expanded mode, dedicated mode, and mixed mode.

Dedicated Mode

In dedicated mode, each of the external-interrupt pins is assigned a vector number. The vector numbers that may be assigned to a pin are those with the encoding PPPP 0010_2 (Figure 6-5;), where the bits marked P are programmed with bits in the interrupt map (IMAP) registers. This encoding of programmable bits and preset bits can be used to designate 15 different vector numbers, each with a different, even-numbered priority. (Vector $0000\ 0010_2$ is undefined, because it has a priority of 0.)

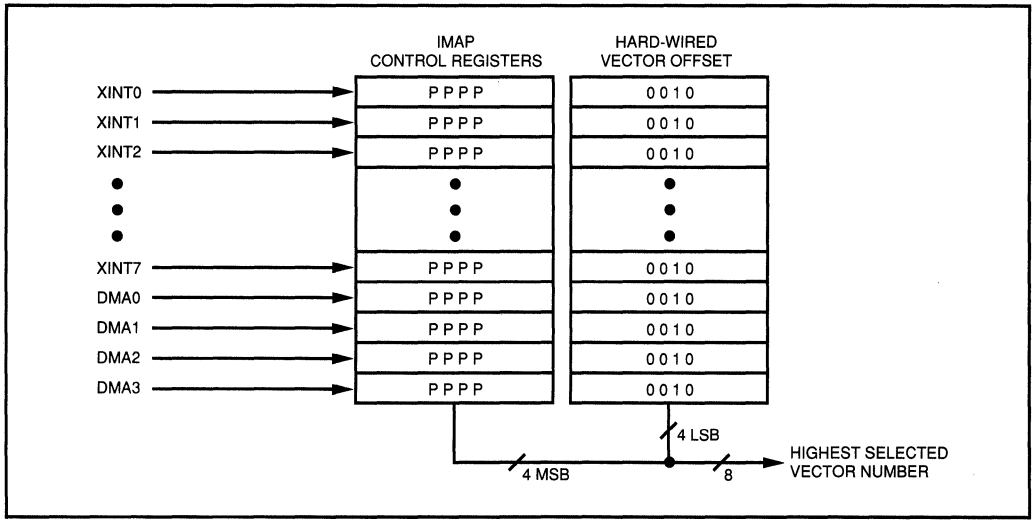


Figure 6-5. Dedicated Mode

Dedicated-mode interrupts are posted in the interrupt-pending (IPND) register. A single bit in the IPND register corresponds to none 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.)

Interrupt vectors are assigned to the DMA inputs in the same way the 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 6-6), with the XINT0 pin representing the least-significant bit, and XINT7 the most significant bit of the number. Note that the external interrupt pins are low-level active. Therefore, the inverse of the vector number is actually applied to the pins.

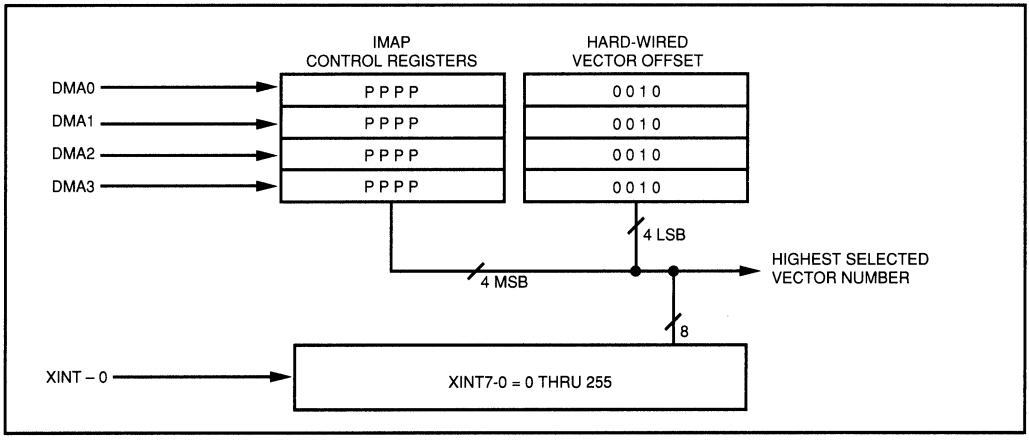


Figure 6-6. Expanded Mode

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 6-7, simple, combinational logic can handle prioritization of the external sources when more than one expanded interrupt is pending. The interrupt source, in this example, must remain asserted until the processor services the interrupt and clears the source.

The external-interrupt pins in expanded mode are always level-low activated. The interrupt controller ignores vector numbers 0 though 7. The output of the external priority encoders in Figure 6-7 can use the 0 vector to indicated that there are no external interrupts pending.

Bit 0 of the IMSK register provides a global mask for all expanded interrupts. The remaining bits (bits 1-7) of the register 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.) The bits 0-7 of the IPND register, in expanded mode, have no function since external logic is responsible for posting interrupts.

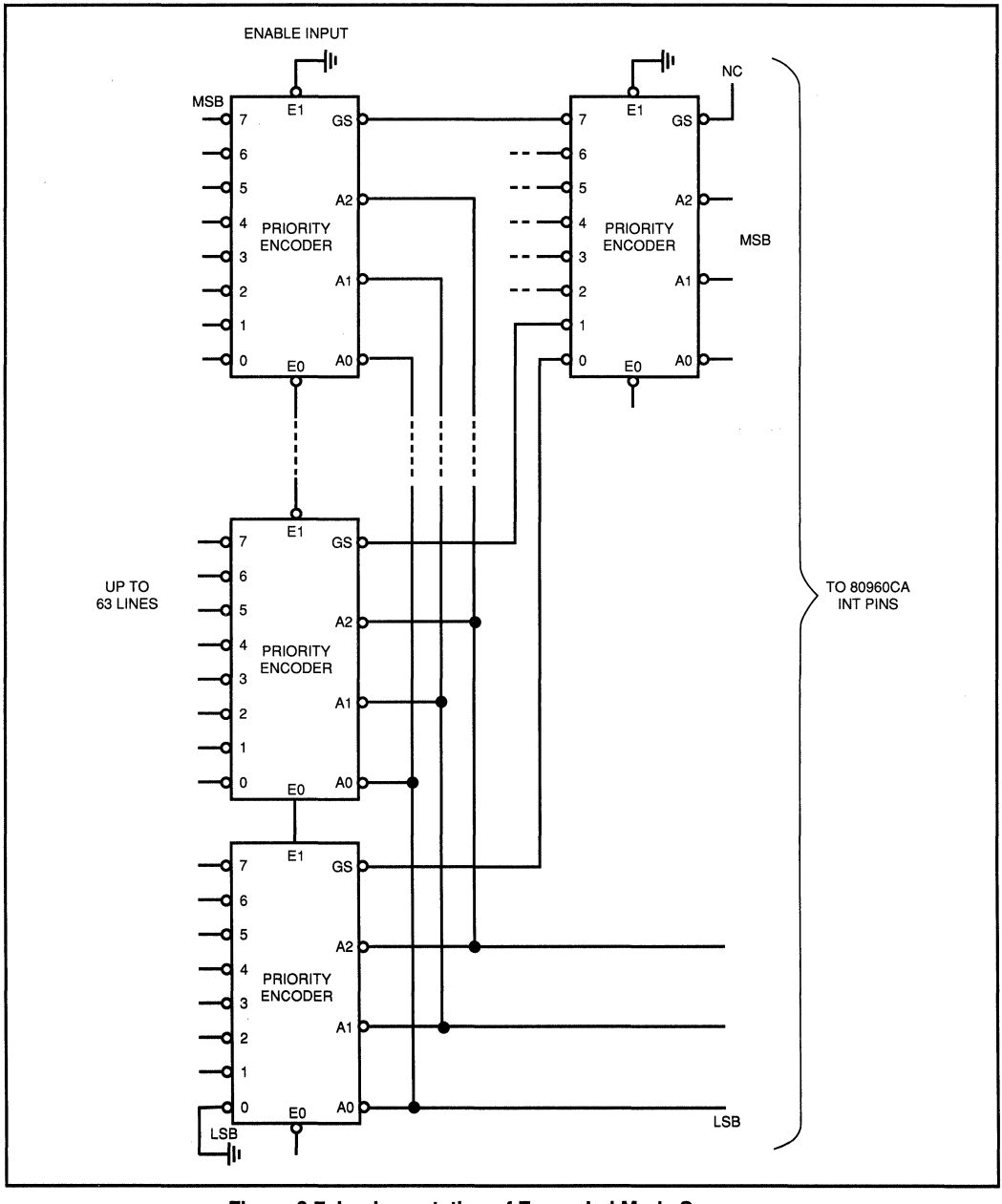


Figure 6-7. Implementation of Expanded Mode Sources

Mixed Mode

In the 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.

Bit 0 of the IMSK register is a global mask for the expanded-mode interrupts, and bits 5 through 7 mask the dedicated interrupts from pins $\overline{\text{XINT5}}$ through $\overline{\text{XINT7}}$, respectively. Bits 1-4 of the IMSK register must be set to 0 in mixed mode. The IPND register posts interrupts from the dedicated-mode pins ($\overline{\text{XINT5}}$-$\overline{\text{XINT7}}$). The bits in the IPND register corresponding to expanded-mode inputs are not used.

Non-Maskable Interrupt

The $\overline{\text{NMI}}$ 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 the priority-31 interrupts and the 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.

When the core receives an NMI request, it is serviced immediately. While servicing an NMI, the core will not respond to any other interrupt requests (even another NMI request) until it has returned 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 value of the IMSK register is saved, the interrupt procedure can restore the value before returning. The option of clearing the mask is selected by programming the ICON register. Several options are provided for clearing the interrupt mask:

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 provided for use 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.

One example of the 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.

Another example of the use of the mask options is for handling priority-31 interrupts. Remember that priority-31 interrupts are interrupted by priority-31 interrupts. If level-activated inputs request a priority-31 interrupt, it is necessary to save and clear the IMSK register when the interrupt is taken. For interrupt inputs which are level activated, an interrupt-handler is typically responsible for acknowledging the interrupt source, thus signalling the source to deactivate. If this is not done, the interrupt controller will continue to detect the active level. Each time the interrupt is detected, another priority-31 interrupt will be nested, preventing the first instruction of the interrupt-handling procedure from being reached.

Optimizing Interrupt Latency and Throughput

The 80960CA controller provides several features aimed at reducing the time required to respond to and service interrupts. The following section describes three features for reducing interrupt latency: caching interrupt vectors on-chip; DMA suspension while servicing interrupts, and caching of interrupt-handling-procedure code.

Vector Caching Option

To reduce interrupt latency, the 80960CA allows some vector entries in the interrupt table to be cached in internal data RAM. When the caching option is selected, all interrupts with a vector number with the four least-significant bit 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.

To use this feature, 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 6-1 shows the required mapping of the vectors to specific locations in the 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 6-1, (remember this vector is always cached in internal data RAM at location 0000H). The processor automatically loads this location at initialization with the value of vector number 248 in the interrupt table.

Table 6-1. 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

Note that the vectors that can be cached coincide with the vector numbers that can be selected with the mapping registers and assigned to dedicated-mode inputs.

DMA Suspension on Interrupts

The core resources required to execute a DMA operation may affect the 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 it is 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 the DMA operation by clearing the channel enable field in the DMA command (DMAC) register. (See *Chapter 15, DMA Controller* for more information on this subject.)

CACHING OF INTERRUPT-HANDLING PROCEDURES

The time required to fetch the first instructions of an interrupt-handling procedure affects the interrupt response time and throughput. The controller allows this fetch time to be reduced by caching interrupt procedures, or portions of the procedures, in the 80960CA's instruction cache. This caching of interrupt procedures is done in the following way.

The instruction cache is divided into two 512-byte halves (Figure 6-8). One or both of these halves can be used for storing interrupt-procedure code. Typically, one half is used as a normal instruction cache, and the other half for caching of interrupt procedures.

Those sections of the interrupt-handling procedures that are to be cached must be placed in a contiguous block of memory. 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 the address space. The maximum size of this block is 512 bytes 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. The **sysctl** instruction 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 two least-significant bits of the interrupt vector must be set $1X_2$ to specify that the interrupt procedure will be fetched from the locked cache rather than the normal memory hierarchy. If the procedure is in the cache, it begins executing it. If a miss at the locked cache occurs, the interrupt procedure will be executed from the normal memory hierarchy. (See *Chapter 5, Programming Environment* for a detailed description of the **sysctl** instruction, and how to configure the load-and-lock features of the instruction cache.)

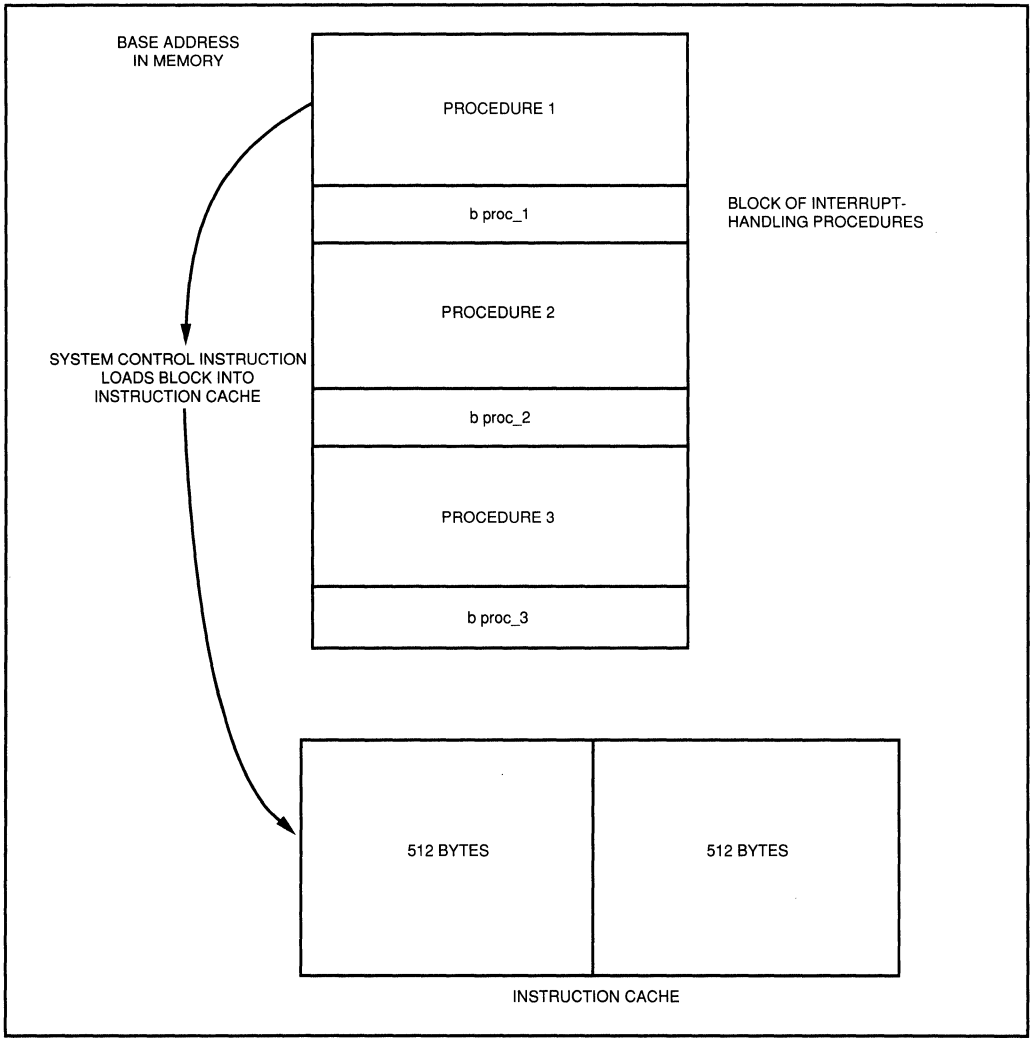


Figure 6-8. Caching Interrupt-Handling Code

Characterizing Interrupt Latency

The time required to perform an interrupt task switch is referred to as the *interrupt latency*. The latency is the time measured between the activation of an interrupt source and the execution of the first instruction for the interrupt-handling procedure for the source.

The interrupt controller in the 80960CA is designed to minimize latency for hardware-generated interrupts. This is achieved by implementing the interrupt detection and the priority-evaluation logic directly in hardware. The latency on the 80960CA depends on several components which are inherent to the internal hardware and the architecture-defined interrupt mechanism. The latency also depends on the hardware and software environment of a particular application. The components of interrupt latency are as follows:

$T_{(detect)}$ - This latency component is the time taken by the interrupt controller for detecting a valid interrupt and for signalling the core that the interrupt is to be serviced. This component depends on the value of the sample-detection mode bit in the ICON register. The fast sample mode results in several clocks less latency than the debounce sample mode.

$T_{(uflow)}$ - This is the latency caused by the execution of the interrupt micro-flow routines. The interrupt microcode performs a complete context switch and allocates a new set of local registers. This latency component is reduced by selecting the vector caching option and the lowest for the NMI, since this is a highly time-critical interrupt.

$T_{(prior\_operations)}$ - This is the latency component due to all non-interruptable instruction or microcode sequences in an application program and the time for finishing pending bus requests. This latency component depends on a system's external-bus performance, and the use of microcoded instructions and functions in an application. Typically, the component is 0 if a single-cycle instruction is interrupted, and the bus request queue is not full when the interrupt occurs.

$T_{(frame\_flush)}$ - If an interrupt occurs when the on-chip local register cache is full, it is necessary to flush local registers to memory. The time required to flush the registers is characterized by $T_{(frame\_flush)}$. Typically, no frame flush occurs for an interrupt, and this component is 0.

$T_{(vector\_fetch)}$ - This is the time required for the processor to fetch the interrupt vector from external memory. This component is always 0 if an interrupt vector is cached in internal data RAM.

$T_{(instruction\_fetch)}$ - This is the latency component for fetching the first instruction for an interrupt handler. It is 0 if the instruction is fetched from the on-chip instruction cache. The instruction cache locking mechanism is provided to guarantee that the instruction will be fetched from cache.

Table 6-2 lists the values for the dominant components of interrupt latency, the detect latency ($T_{(detect)}$) and the micro-flow latency ($T_{(uflow)}$). The other latency components listed above are typically 0, or have the value 0 for certain interrupt controller configurations.

Table 6-2. Components of Interrupt Latency

Latency Component	Latency (PCLK cycles)	Condition
$T_{(detect)}$	5	Fast sample mode, NMI
	9	Debounce sample mode, NMI
	10	Fast sample mode, non-NMI
	14	Debounce sample mode, non-NMI
$T_{(uflow)}$	20	NMI
	20	Non-NMI, cached vector
	24	non-NMI, non-cached vector

Note: The detection latency ($T_{(detect)}$) for a dedicated interrupt is further reduced by 2 clocks when the input priority is at least 16 and all other hardware interrupt priorities are less than 16.

The latency for an interrupt is the sum of the latency components described above. For example, consider a typical dedicated-mode interrupt. The interrupt controller is configured for fast-sample mode, cached-interrupt vectors; and the interrupt procedure is locked in the instruction cache. In this case, the interrupt latency is 10+20=30 clocks. (This is a typical latency of 0.9 µs at 33 MHz.) A similar calculation yields a latency of 5+20=25 clocks for the NMI input. (This is a typical NMI latency of .75 µs at 33MHz.)

External Interface Description

This section describes the physical characteristics of the interrupt inputs. The 80960CA 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:

$\overline{\text{XINT7}}$-$\overline{\text{XINT0}}$ — External Interrupt (Input) - The external interrupt pins cause interrupts to be requested. These pins are software configurable for three different modes: dedicated mode, expanded mode, and mixed mode. 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.

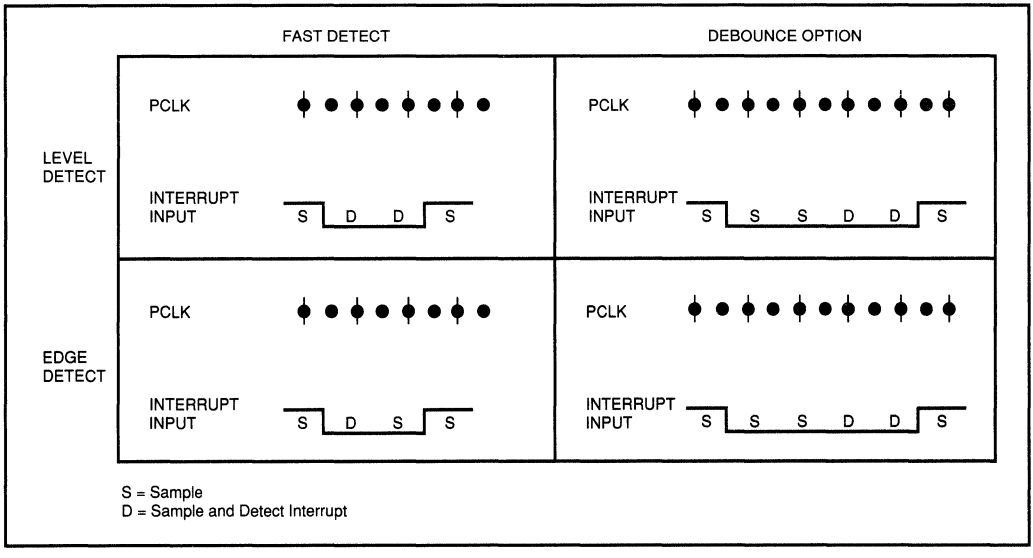
$\overline{\text{NMI}}$ — Non-Maskable Interrupt (Input) This input causes a non-maskable interrupt event to occur. NMI is the highest priority interrupt recognized. The $\overline{\text{NMI}}$ pin is an edge-activated input. A debouncing mode for $\overline{\text{NMI}}$ can be selected under program control. These pins are internally synchronized.

The functions of the external-interrupt pins depend on the mode of operation (expanded, dedicated, or mixed) and on several other options selected by setting bits in the ICON register.

Interrupt Detection Options

When an external-interrupt pin is programmed for dedicated-mode operation, it can be programmed for level-low detection or falling-edge detection in the ICON register. Also, all of the dedicated inputs and the $\overline{\text{NMI}}$ pin can be programmed (globally) for fast sampling or debounce sampling. Expanded-mode inputs are always sampled in the debounce mode.

When a pin is programmed for low-level detection, the input is detected whenever a logic 0 is present on the pin. When a pin is programmed for edge detection, the input is detected only when a 1 to 0 transition occurs. The debounce sampling mode requires that a low level is detected for three consecutive samples before the input is detected. The inputs are sampled internally once every two PCLK cycles. This feature provides a built-in filtering of noisy or slow-falling inputs. Figure 6-9 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.



S = Sample
D = Sample and Detect Interrupt

Figure 6-9. Level and Edge Detection Options

Programmer's Interface

The programmer interfaces to the interrupt controller with four control registers, and two special function registers: the ICON control register; the IMAP0-IMAP2 control registers; the IMSK special-function register; and the IPND special function register. These registers are described in the following sections.

Interrupt Control Register (ICON)

The ICON register (Figure 6-10) is a 32-bit control register that is used to set 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.

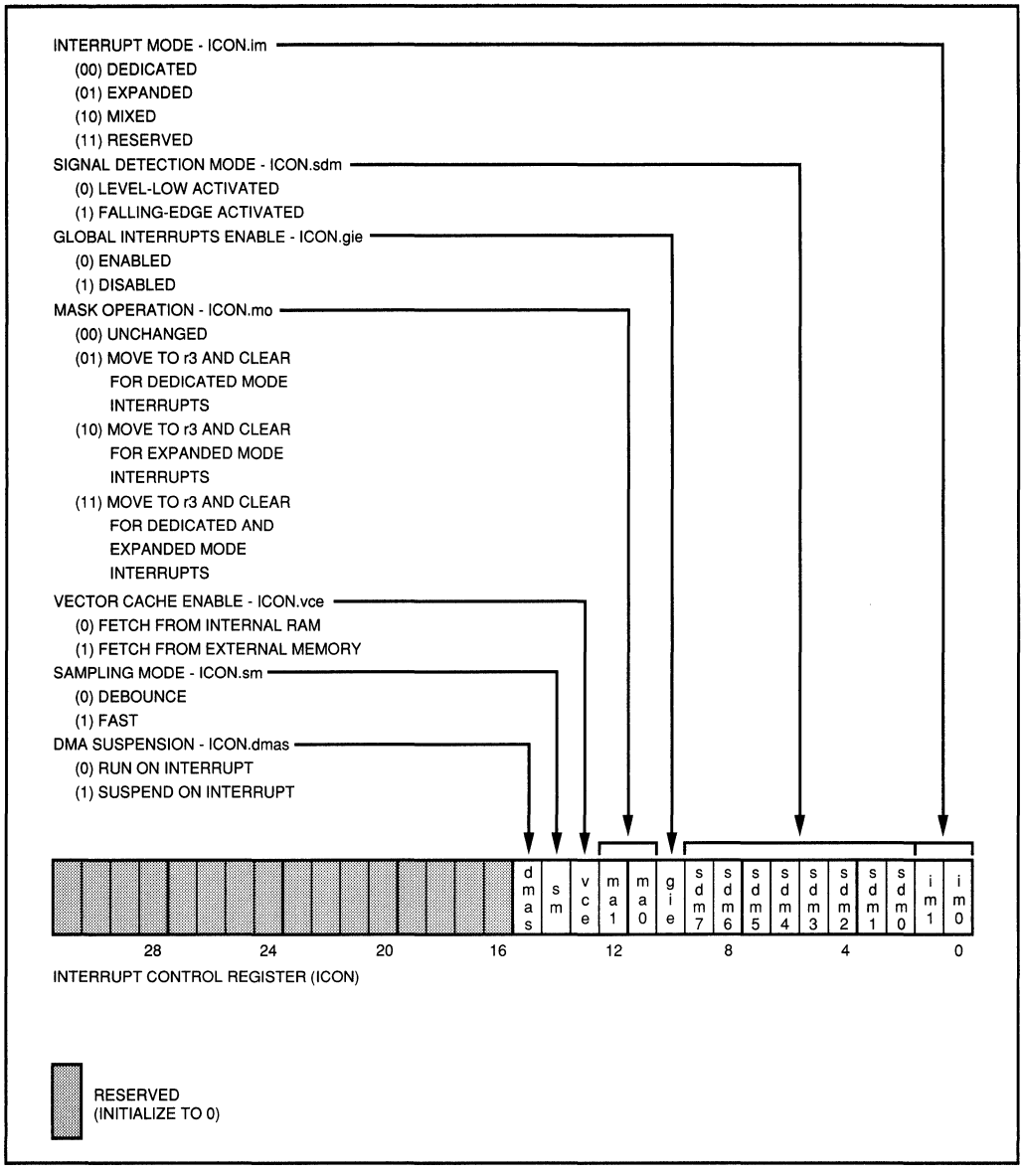


Figure 6-10. Interrupt Control (ICON) Register

The *interrupt-mode field* (bits 0 and 1) of the ICON register determines the mode of operation for the external-interrupt pins (XINT0 through XINT7) - dedicated, expanded, or mixed.

The *signal-detection-mode bits* (bits 2 through 9) determine whether the signals on the individual external-interrupt pins ($\overline{\text{XINT0}}$ - $\overline{\text{XINT7}}$) are level-low activated or falling-edge activated Expanded-mode inputs are always level-detected, and the NMI input is always edge-detected regardless of the value of this bit.

The *global-interrupts enable bit* (bit 10) globally enables or disables the external-interrupt pins and the DMA inputs. It does not affect the$\overline{\text{NMI}}$ pin. This bit performs the same function as clearing the mask register.

The *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.

The *vector cache enable bit* (bit 13) determines whether or not the interrupt table vector entries will be fetched from the interrupt table or from internal data RAM. Only the vectors with four least-significant bits equal to 0010_2 may be cached in internal data RAM.

The *sampling-mode bit* (bit 14) determines whether the dedicated inputs and $\overline{\text{NMI}}$ pin are sampled using debounce sampling or fast sampling. Expanded-mode inputs are always detected using the debounce mode.

The *DMA-suspension bit* (bit 15) determines whether or not DMA continues running or is suspended while an interrupt procedure is being called.

Bits 16 through 31 are reserved and should be set to 0 at initialization.

Interrupt Mapping Registers (IMAP0-IMAP2)

The IMAP registers (Figure 6-11) are three 32-bit registers ($\overline{\text{IMAP0}}$ through $\overline{\text{IMAP2}}$). The bits in these registers are used to program the vector number associated with the interrupt source when the source is connected to a dedicated-mode input. Registers IMAP0 and IMAP1 contain mapping information for the external-interrupt pins (four bits per pin), and register IMAP2 contains mapping information for the DMA-interrupt inputs (four bits per input).

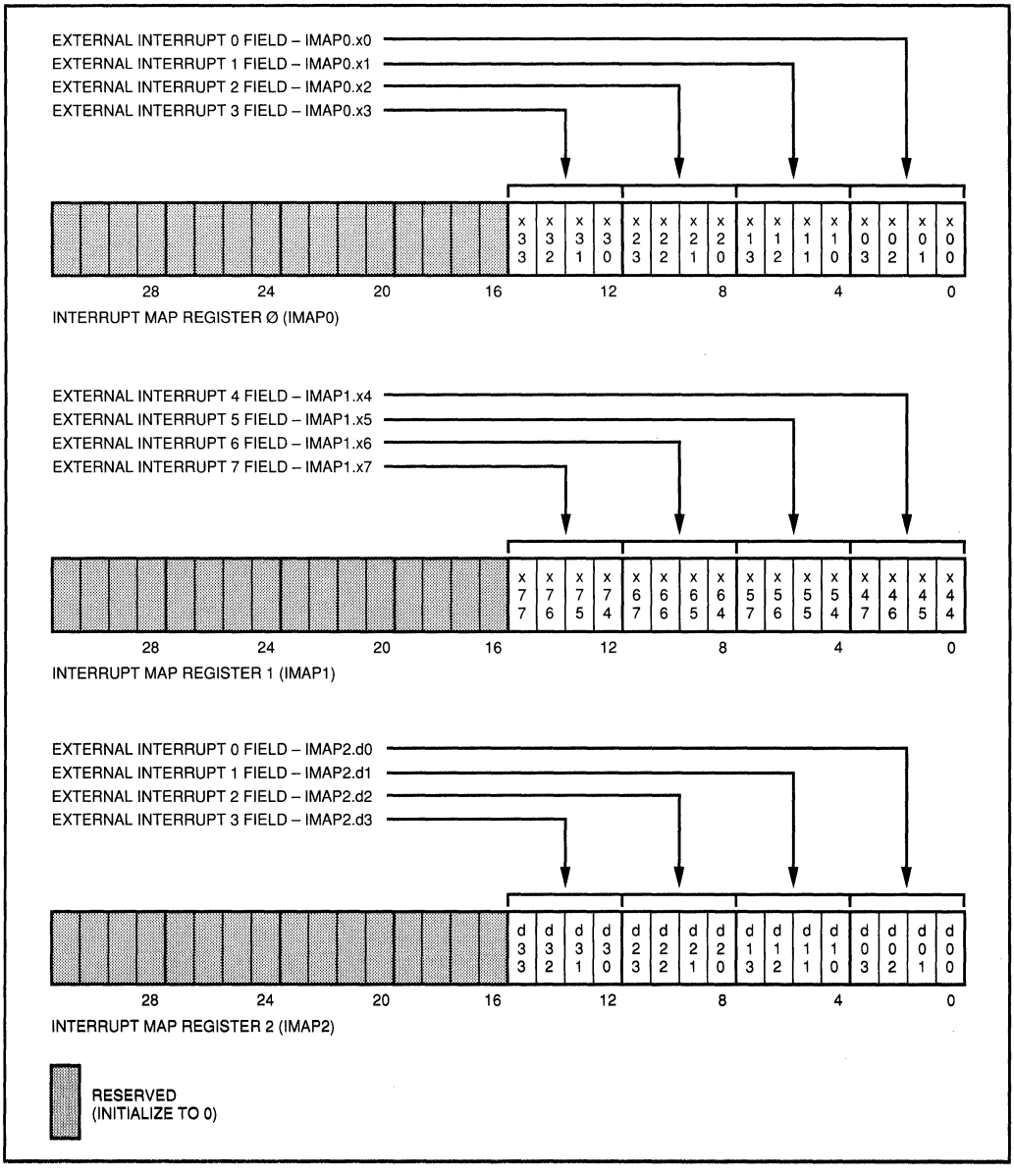


Figure 6-11. Interrupt Mapping (IMAP2-IMAP0) Registers

Each set of four bits contains the four most-significant bits of a vector number; 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, bits 4 through 7 of IMAP0 contain mapping information for the $\overline{\text{XINT1}}$ 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. 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 6-12) are special-function registers (sf1 and sf0, respectively). Bits 0 through 7 of these registers are associated with the external-interrupt pins ($\overline{\text{XINT0}}$ through $\overline{\text{XINT7}}$), 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.

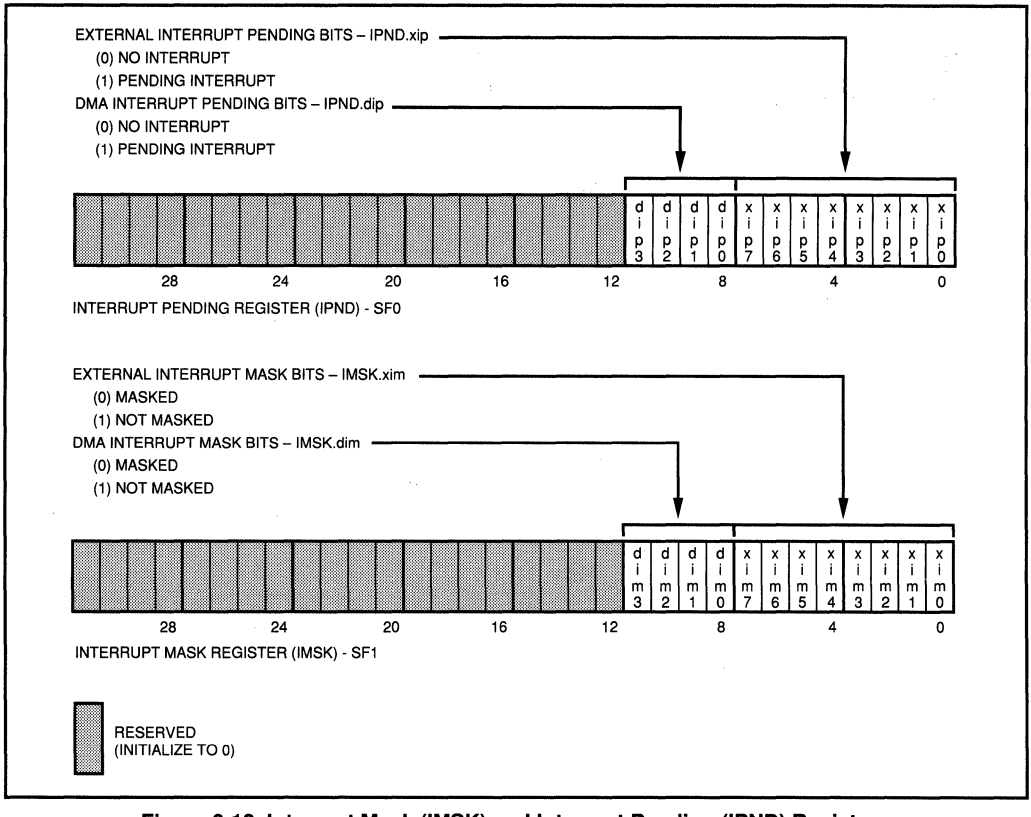


Figure 6-12. Interrupt Mask (IMSK) and Interrupt Pending (IPND) Registers

The IPND register is provided to post dedicated-mode interrupts originating from the eight external-dedicated sources (when configured in dedicated mode), or the four DMA sources. Asserting one of these inputs causes a 1 to be latched into its associated bit in the IPND register. Bits 0 through 7 of this register are not used in expanded mode; bits 0 through 4 are not used in mixed mode.

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.

Bit 0 of the mask register has two functions. It masks interrupt pin $\overline{\text{XINT0}}$ in the dedicated mode, and it globally masks all expanded-mode interrupts in the expanded and mixed modes. Bits 1 through 7 of this register are not used in expanded mode; bits 1 through 4 are not used in mixed mode. When these bits are not being used, they must be cleared.

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 clears the bit that has been 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 will remain set. Because of this, the interrupt routine for a level-detected interrupt must clear the external interrupt source.

An alternative method of posting interrupts in the IPND register (other than through the external-interrupt pins and DMA-interrupt inputs) is to sets 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, provided that the bit set in the IPND register is associated with an interrupt source that is programmed for dedicated-mode operation.

Default and Reset Register Values

The ICON and IMAP2-0 registers are control registers and are loaded from the control table in external memory when the processor is initialized or reinitialized. (The control table is described in *Chapter 2, Programming Environment*.) The IMSK register is set to 0 when the processor is initialized ($\overline{\text{RESET}}$ is deasserted). The value of the IPND register is undefined after a power-up initialization. The user is responsible for clearing this register before any bits in the mask register are set; otherwise, unwanted interrupts may be triggered. For a reset while power is on, the value of the pending register is retained.

Setting Up the Interrupt Controller

This section provides some examples of setting up the interrupt controller. Recall that the IMAP and ICON registers are control registers. The entire control table is automatically read at initialization, and the ICON and IMAP registers are loaded at that time with the values pre-programmed in the table. In many applications, setting the values of these registers in the initial control table is all of the setup required. The following examples describe how the interrupt controller can be dynamically configured after initialization.

Example 6-1 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 16, Initialization and System Requirements* for a description of relocating data structures after initialization.)

Example 6-1. Programming the Interrupt Controller for Expanded Mode

```
# Example expanded mode setup . . .

        mov     0,sf1                   # clear the IMSK register (mask all interrupts)
        ldconst 0x01, g0
        st      g0,ctrl_table_ICON      # store the mode information to the control table
        ldconst 0x301,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
```

Software-Generated Interrupt Requests

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 80960CA, or another processor, posting an interrupt in the pending-interrupts, and pending-priorities fields of the interrupt table.

SYSCTL Instruction

The **sysctl** instruction is typically used to request an interrupt in a program (Example 6-2). 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 5, Programming Environment* for a complete discussion of the **sysctl** instruction.)

Example 6-2. 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 the bits in the pending-interrupts and pending-priorities fields that correspond to the requested interrupt.
2) 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 that was just posted.

The interrupt controller continuously compares the value in the software-priority register with the current process priority and the priority of the highest pending, hardware-generated interrupt. When the value in the software-priority register 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) 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 is not visible to the user. The core only updates the value in this register when a **sysctl** instruction requests an interrupt, and when a software-generated interrupt is serviced.

Posting Interrupts Directly to the Interrupt Table

The 80960CA, or an external agent that is sharing memory with the 80960CA (such as an I/O processor or another 80960CA), 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.

Note: The **sysctl** instruction 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, the **sysctl** instruction could be executed periodically to guarantee recognition of pending interrupts which were posted in the table by the external agent.

Faults 7

CHAPTER 7
FAULTS

This chapter describes the fault-handling facilities of the 80960CA. The subjects covered include the fault-handling data structures and the fault-handling mechanism. A reference section that contains detailed information on each fault type is provided at the end of the chapter.

OVERVIEW OF THE FAULT-HANDLING FACILITIES

The architecture defines various conditions in code or in 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 conditions 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.

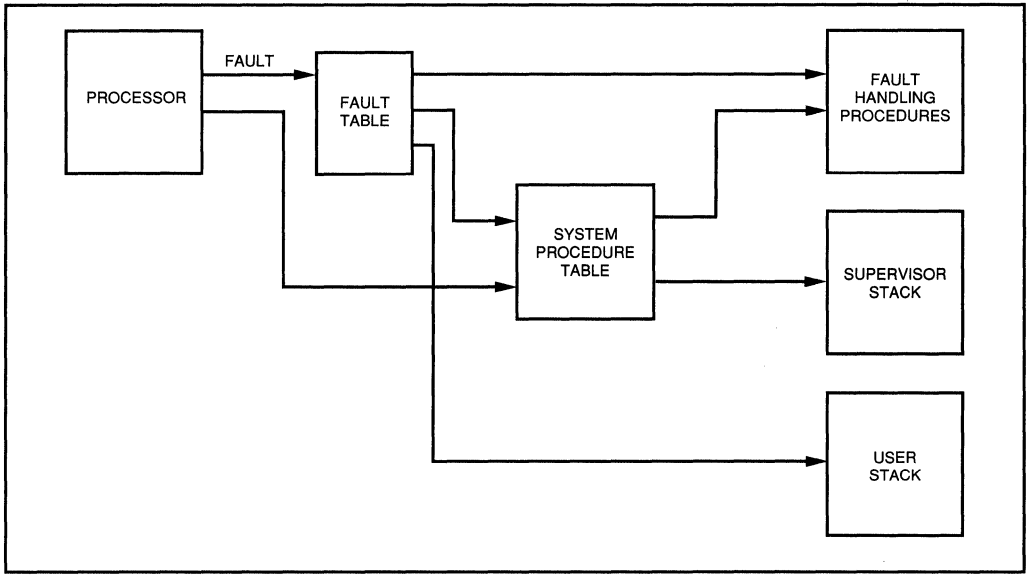


Figure 7-1. Fault-Handling Data Structures

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 (either a user stack or a supervisor stack or both) to handle faults generated by the processor.

The fault table contains pointers to the fault-handling procedures. The system-procedure table is optionally used to provide an interface to any of the fault-handling procedures, and to allow faults to be handled in supervisor mode. The stack frames for fault-handling procedures are created on either the user or the supervisor stack, depending in which mode the fault is handled.

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

The processor can detect a fault at any time while it is executing instructions, including a program, an interrupt-handling procedure, or a fault-handling. If a fault occurs when the processor is 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); or 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 (i.e., the stack being used by the fault-handling procedure). This record includes information on the fault and on the state of the processor when the fault was generated.

Following the creation of the fault record, the processor begins executing the selected fault-handling procedure. If the fault-handling procedure is able to recover from the fault, the processor can then restore itself to its state prior to the fault and resume work on the program with no break in the control flow of the program. 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 80960 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 80960CA recognizes faults defined in the 80960 architecture and a new fault subtype for detecting unaligned memory accesses. Table 7-1 lists all the faults that the 80960CA detects, arranged by type and subtype.

Table 7-1. 80960CA Fault Types and Subtypes

Fault Type		Fault Subtype		Fault Record
Number	Name	Number/Bit Position	Name	
0H	Parallel	2H-FFH	Indicates number of faults that occur in parallel	XX00 XX02H XX00 XXFFH
1H	Trace	Bit 1 Bit 2 Bit 3 Bit 4 Bit 5 Bit 6 Bit 7	Instruction Trace Branch Trace Call Trace Return Trace Prereturn Trace Supervisor Trace Breakpoint Trace	XX01 XX02H XX01 XX04H XX01 XX08H XX01 XX10H XX01 XX20H XX01 XX40H XX01 XX80H
2H	Operation	1H 2H 3H 4H	Invalid Opcode Unimplemented Unaligned (see note) Invalid Operand	XX02 XX01H XX02 XX02H XX02 XX03H XX02 XX04H
3H	Arithmetic	1H 2H	Integer Overflow Arithmetic Zero-Divide	XX03 XX01H XX03 XX02H
4H	Reserved (Floating Point)			
5H	Constraint	1H 2H	Constraint Range Privileged	XX05 XX01H XX05 XX02H
6H	Reserved			
7H	Protection	Bit 1	Length	XX07 XX01H
8H - 9H	Reserved			
AH	Type	1H	Type Mismatch	XX0A XX01H
BH - FH	Reserved			

Note: The operation-unaligned fault is an 80960CA-specific extension.

The first column of Table 7-1 gives the fault type numbers in hexadecimal, and the second column gives the name of the fault type.

The third column gives the fault subtype number, either 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 the trace faults, where it is possible for two or more fault subtypes to be generated simultaneously.

The fourth column gives the name of the fault subtype. 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.

Note: Other 80960 implementations may provide different extensions that recognize different fault conditions. The encoding of fault types and subtypes allows any of these additional faults to be included in the fault table along with the basic faults. Space in the fault table will be 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 processor based on the 80960 architecture that provides floating-point operations will 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 has been 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 types of fault-table entries are allowed: a local-call entry, and a 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 (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.

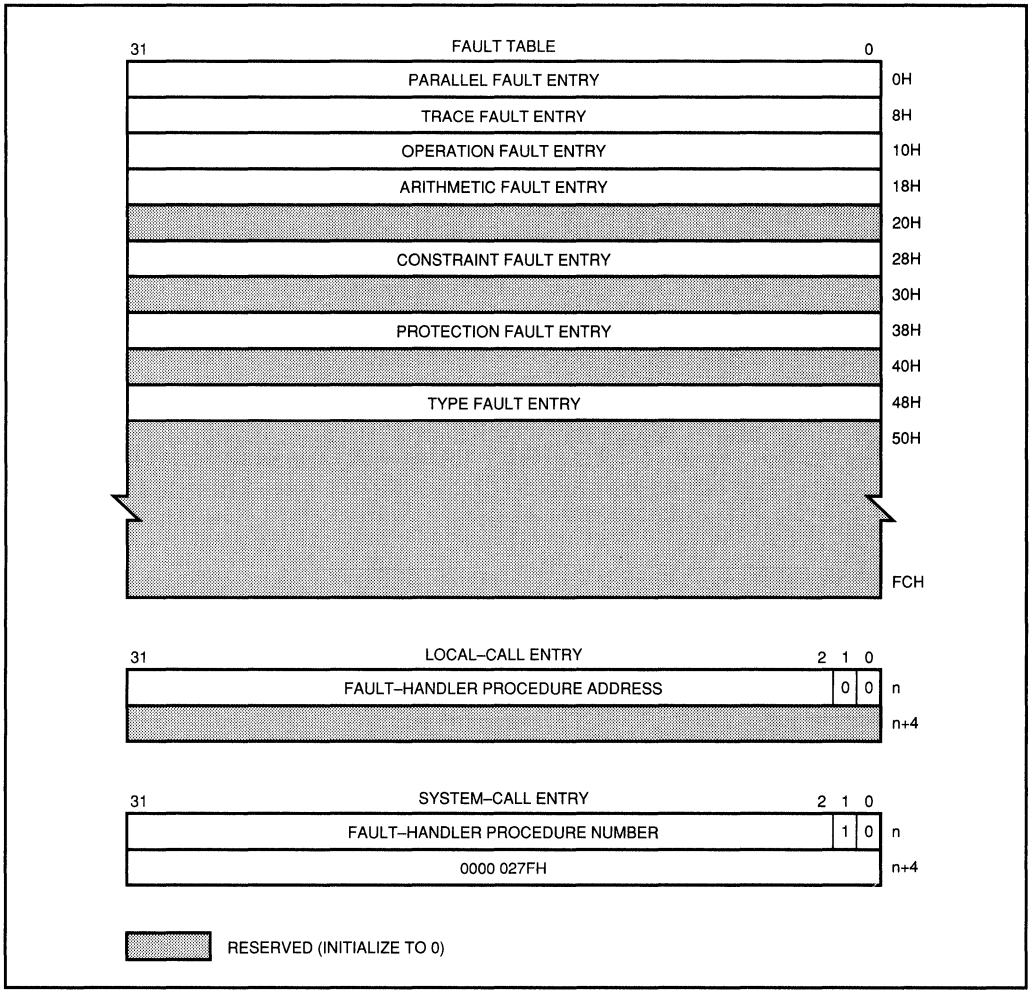


Figure 7-2. Fault Table and Fault-Table Entries

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.

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 (the user stack, the interrupt stack, or the 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 execution of the program. 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 that the processor faulted upon.

The values of the process-controls register and arithmetic-controls register at the time that a fault is generated are stored in their respective fields in the fault record. This information is used to resume work on the program after the fault has been 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.

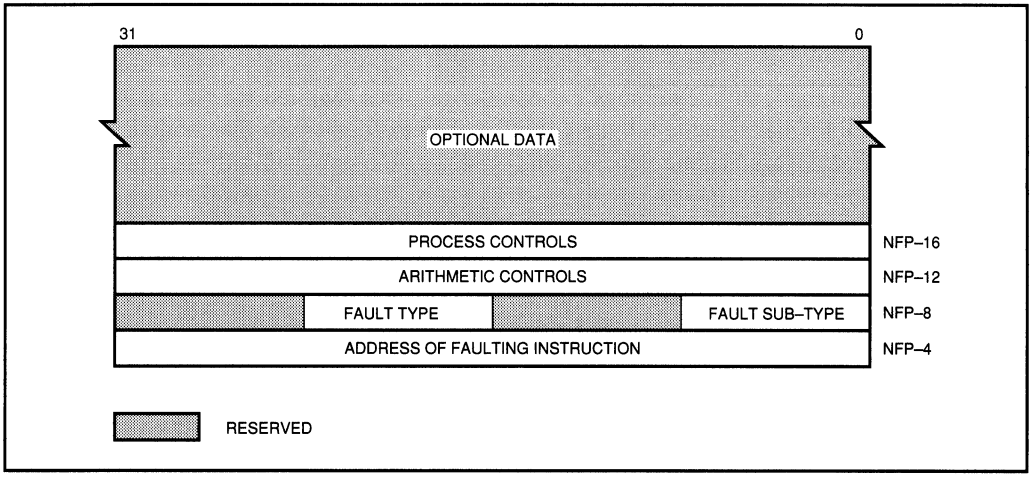


Figure 7-3. Fault Record

The optional-data field is defined for certain faults, and contains additional information about the faulting conditions, usually to assist resumption. The parallel fault type is the fault that contains optional data on the 80960CA. The processor can generate parallel faults when instructions are executed in parallel. Parallel faults and the contents of the optional-data field for this fault type, are described later in the section titled *Multiple Fault Conditions*. All of the bytes not used in the fault record are reserved.

Return Instruction Pointer

When a fault-handling procedure is called, like 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 control flow of the program. 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 execution of the program.

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, the supervisor stack, or the interrupt stack. The fault record begins at the 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. The size of the fault record is variable and depends on the size of the optional fault-data portion of the fault record.

Note: The alignment of stack frames is defined for each implementation of the 80960 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 80960CA, SALIGN = 1)

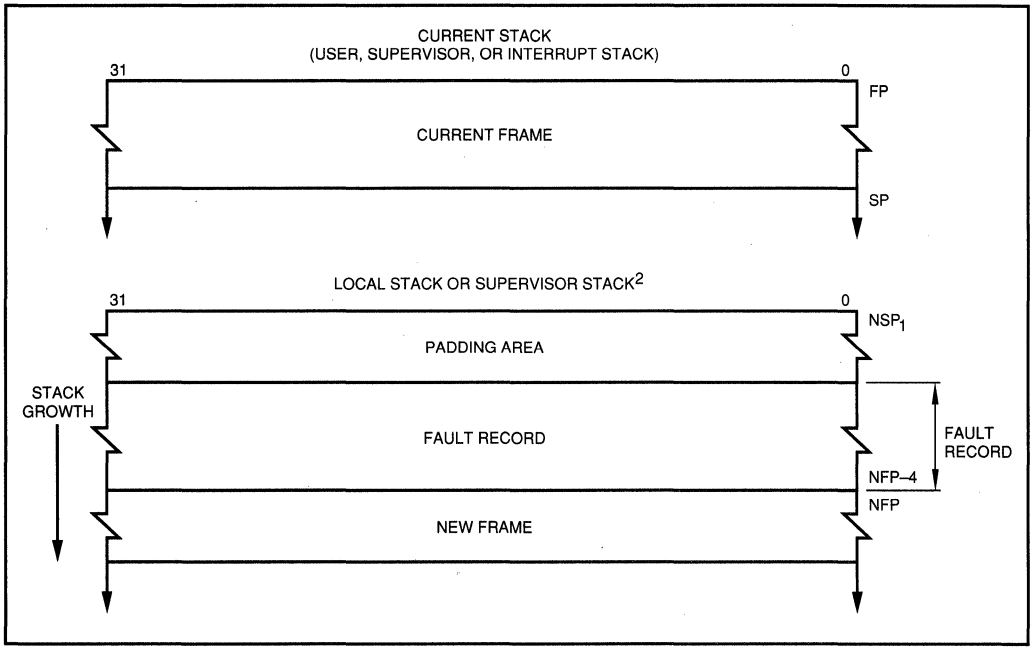


Figure 7-4. Storage of the Fault Record on the Stack

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

MULTIPLE AND PARALLEL FAULTS

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

Multiple Faults

Multiple fault conditions can occur during the execution of a single instruction. 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 of the multiple fault conditions and may not report all multiple faults that it detects.

In a multiple fault situation, the fault condition which is reported 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 which are generated by the execution of a single instruction will be 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 the occurrence of multiple faults of the same type (see 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 Debug* for a detailed 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, 80960CA Internal Architecture*, the 80960CA 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 of the execution units has completed executing its respective instruction and all out-of-order instructions have been executed.

For example, if an integer overflow occurs during the addition in the following code example, the fault is detected before the load and the multiply have completed execution. Before invoking the integer-overflow fault-handling procedure, the processor waits for the load and multiply to complete.

```
ld      (g0), g1;
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 in the section titled *Precise and Imprecise Faults*.) Unless precise 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.

Note: 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.

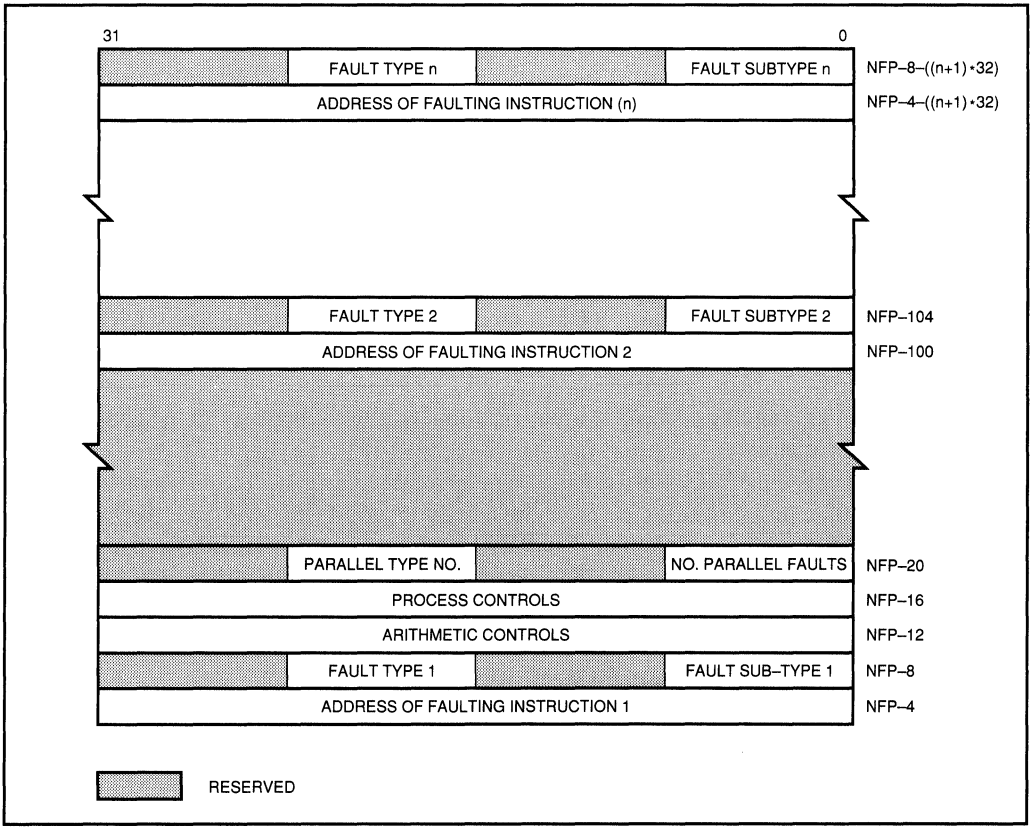


Figure 7-5. Fault Record for Parallel Faults

Note: To calculate the byte offsets, "n" indicates the number of the fault. Thus, for the second fault recorded (n = 2), the relationship (NFP-4 ((n+1)*32)) reduces to NFP-100.

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.* The information for the remaining parallel faults is then written to the optional-data field of the fault record, and the fault-handling procedure for parallel faults is invoked.

The first word in the optional-data field of the fault record (NFP-20) contains information about the parallel faults. The byte at offset NFP-17 contains 00H (the encoding for the parallel-fault type), and 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. (The values of the arithmetic-controls and process-controls registers 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 the 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

Many of the faults that occur can be recovered from easily. When recovery from the fault is possible, the processor's fault-handling mechanism allows the processor to automatically resume work on the program or interrupt that it was working on when the fault occurred. The resumption action 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 a shut-down of the processor, or system.

- Perform a shut down of the processor, or system, without explicitly saving the processor state or the fault information.

When working with the processor at the development level, a common action of the fault-handling procedure 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 80960CA's multi-stage execution pipeline, faults can occur prior to the execution of the faulting instruction (i.e., the instruction that causes the fault), during the instruction execution, or immediately following the execution. When the fault occurs before the faulting instruction is executed, the faulting instruction may be re-executed on the 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 change in the state of the program such that the execution of the program can not be resumed after the fault has been 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
- Arithmetic Zero-Divide
- All Constraint Subtypes
- All Trace Subtypes
- Length

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 so that the control flow of the program 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 process-controls register on the return from the fault-handling procedure. The processor only performs this action if the processor is in the 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 and the internal state field 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 the conditions which cause faults, and the facilities for controlling faults which are optionally generated.

Implicit Generation of Faults

Most faults are generated implicitly. That is, they occur as a side effect of the execution of an instruction which has encountered difficulty. The following paragraphs summarize the 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 – When the result of an integer operation will not fit in the specified destination, an integer-overflow fault is signaled. The integer-overflow fault-handling procedure is invoked if the integer-overflow mask bit in the arithmetic-controls register is set to enable these faults.

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

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

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

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

sdma **udma**
sysctl

If the application attempts to modify a supervisor-only resource while not in supervisor mode, the type-mismatch fault is generated. On the 80960CA, the process-control register, on-chip data RAM, and special-function registers are supervisor-only resources. The following actions if attempted when the processor is not in supervisor mode generate a type-mismatch fault:

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

Out-of-bounds System-Procedure Call – 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, the protection-length fault is generated.

Invalid Instruction Encodings – If the processor encounters an invalid opcode, or an invalid encoding of the addressing mode of a MEM-format instruction, it generates the operation-invalid-opcode fault.

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

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

Referencing a Non-existent SFR – If the processor executes an instruction which references a non-existent special-function register, the invalid-operand fault is generated. On the 80960CA, only sf0, sf1 and sf2 are implemented.

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

Execution from Internal Data RAM – An attempt to execute an instruction which was fetched from the 80960CA's on-chip data RAM generates the operation-unimplemented fault.

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

Breakpoint Detected –When the processor executes an instruction at an instruction pointer which matches one of the programmed instruction-address breakpoints and trace-faults are enabled, a trace fault is generated.

When the processor issues a memory request that matches one of the programmed data-address breakpoints and trace faults are enabled, a trace fault generated.

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

Explicit Generation of Faults

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

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

Fault Controls

Certain fault types and subtypes have masks 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, the data structures in which they are located, the fault subtypes they affect, and where more information about them may be found.

Table 7-2. Fault Flags or Masks

Flag or Mask Name	Location	Faults Affected
Integer-Overflow Mask Bit	Arithmetic-Controls Register	Integer Overflow
No-Imprecise Faults Bit	Arithmetic-Controls Register	All Imprecise Faults
Trace-Enable Bit	Process-Controls Register	All Trace Faults
Trace-Mode Flags	Trace-Controls Register	All Trace Faults
Unaligned-Fault Mask	Process-Control Block	Unaligned fault

Note: The unaligned fault, unaligned-fault mask and the processor control block are 80960CA extensions to the 80960 architecture.

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 in the section titled *Precise and Imprecise Faults.*

The trace-mode bits (in the trace-controls register) and trace-enable bit (in the process-controls register) support trace faults. The trace-mode bits enable trace modes; the trace-enable bit enables the generation of trace faults. 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.*

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

FAULT-HANDLING ACTION

Once a fault has occurred, the processor saves the program state; calls the fault-handling procedure; and restores the program state (if this is possible) once the fault recovery action has been 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 the 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 *Generalized 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, the supervisor stack, or the 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 return-type field in the PFP.

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 a recovery from the fault, a **ret** instruction in the fault-handling procedure allows processor control to return to the program that was being worked on when the fault occurred. On the return, the processor performs the action described in the section of *Chapter 5, Procedure Calls* titled *Generalized Return Operation*, except that the arithmetic-controls field from the fault record is copied into the arithmetic-controls register. Since the call made is local, the process-controls field from the fault record is not copied back to the process-controls 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 address of the fault-handling procedure 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 as is described in the section of *Chapter 5, Procedure Calls* titled *Generalized Call Operation*, with the following exceptions:

- If the processor is in user mode when the fault occurs, it switches to supervisor mode, reads the supervisor-stack pointer from the system-procedure table, and switches to the supervisor stack. A new frame is then created on the supervisor stack.
- If the processor is already in supervisor mode when the fault occurs, the processor creates a new frame on the current stack.

Note: 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 return-type field in the PFP register. If the fault is not a trace fault, it copies the state of the trace-control flag (byte 12, bit 0) of the system-procedure table into the trace-enable bit of the process-controls register. 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 *Generalized Return Operation*, with the following exceptions:

- The arithmetic-controls field of the fault record is copied into the arithmetic-controls register. If the processor is in supervisor mode prior to the return from the fault-handling procedure (which it should be), the process-controls field in the fault record is copied into the process-controls register. (Restoring the process-controls register causes the trace-fault-pending flag and trace-enable bit to return their value before the fault occurred.) 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.

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

Faults and Interrupts

If an interrupt occurs during an instruction that will fault, an instruction that has already faulted, or during the selection of the fault-handling procedure, 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. On returning from the interrupt, the fault is handled. 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 80960 architecture allows some faults to be generated together and not in sequence to support parallel and out-of-order execution of instructions. When this situation occurs, it may be impossible to recover from some of the 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. These mechanisms are the no-imprecise-faults bit (NIF bit) in the arithmetic-controls register 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 that are intended to be recoverable by software. 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 execution of the program without breaking the control flow of the program. The following faults are always precise:

- trace
- protection

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 address of the faulting instruction is correct, but the state of execution of the 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. The faults that the architecture allows to be imprecise include the following:

- operation
- arithmetic
- constraint
- type

Whether a specific fault is always precise, or not, is described later in the *Fault Reference* section of this chapter.

Asynchronous faults are those whose occurrence has no direct relationship to the instruction pointer. The 80960 architecture does not define any faults in this category, and the 80960CA generates no such faults.

The NIF bit controls whether or not imprecise faults can be generated. 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 (that is, the address of the faulting instruction 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 (that is, in parallel and out of order, and with an imprecise RIP). Here, the following conditions hold true:

1) When an imprecise fault occurs, the address of the faulting instruction 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 source operands of the faulting instructions 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 the **syncf** instruction and to generate all faults, before it begins work on instructions that occur after the **syncf** instruction.

This instruction has two uses. One use is to force faults to be precise when the NIF bit is clear. The other use is to ensure that all instructions are complete and all faults generated in one block of code before the execution of another block of code begins. One example of where **syncf** might be used is between the boundaries of ADA blocks, where the blocks have different exception handlers.

Compiled code should execute with the NIF bit clear, using the **syncf** instruction 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.

If recovery from one or more of the imprecise faults is required the NIF bit should be set. For example, if a program needs to handle unmasked integer-overflow faults and recover from them, and the fault-handling procedure cannot be closely coupled with the application to perform imprecise fault recovery the NIF bit should be set.

FAULT REFERENCE

This section describes each of the fault types and subtypes 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 — The fault-type section gives the number which appears in the fault-type field of the fault record when the fault is generated. The fault-subtype section lists the fault subtypes and their associated number for each fault sub-type.

Function — The function section gives a general description of the purpose of the fault type, then describes the purpose of each of the fault subtypes in detail. It also describes how the processor handles each fault subtype.

RIP — The RIP section describes what value is saved in the RIP register of the stack frame that the processor was using when the fault occurred.

Program State Changes — The program-state-changes section describes the effects that the fault subtypes have on the control flow of a program.

Arithmetic Faults

Fault Type: 3H

Fault Subtype:

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

Function: Indicates that there is a problem with an operand or the result of an arithmetic instruction.

The integer-overflow fault is generated when the result of an integer instruction overflows the destination, and the integer-overflow mask in the arithmetic-controls register is cleared. Here, the n least-significant bits of the result are stored in the destination, where n is the destination size. The following instructions can generate this fault:

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

The arithmetic zero-divide fault is generated when the divisor operand of an ordinal or integer divide instruction is zero. The following instructions can generate this fault:

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

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

Program State Changes: These faults may be imprecise when executing with the NIF bit cleared.

The integer-overflow fault may not be recoverable, because the result is stored in the destination before the fault is generated. For example, the faulting instruction can not be re-executed if the destination register was also a source register for the instruction. The arithmetic zero-divide fault is generated before the 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 that a program or procedure has violated an architectural constraint.

The constraint-range fault is generated when a fault-if instruction is executed, and the condition code field in the arithmetic-controls register matches the condition required by the instruction.

The privileged fault is also generated when a program or procedure attempts to use a privileged (supervisor-mode only) instruction while the processor is in user mode. The privileged instructions on the 80960CA are:

sdma udma sysctl

RIP: No defined value.

Program State Changes: These faults may be imprecise when executing with the NIF bit cleared.

No changes in the program's control flow accompany these faults. The constraint-range fault is generated after the fault-if instruction has been executed, but the program state is not affected. The privileged fault is generated before the faulting instruction is executed.

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 that the processor cannot execute the current instruction because of invalid instruction syntax or operand semantics.

The invalid-opcode fault is generated when the processor attempts to execute an instruction that contains an undefined opcode or addressing mode.

The unimplemented fault is generated when the processor attempts to execute an instruction which was fetched from the on-chip data RAM.

The unaligned fault is generated when the processor attempts to access an unaligned word or group of words in memory, and the fault is enabled by the unaligned-fault mask bit in the fault configuration word of the PRCB.

The invalid-operand fault is generated when the processor attempts to execute an instruction for which one or more of the operands have special requirements which are not satisfied. Specifying a non-existent special-function register, or a **sysctl** command which is not defined, causes this fault. Referencing an unaligned long-, triple-, or quad-register group will also cause this fault.

RIP: No defined value.

Program State Changes:

These faults may be imprecise when executing with the NIF bit cleared.

A change in the program's control flow does not accompany the operation faults, because the faults occur before the execution of the instruction.

Parallel Faults

Fault Type: 0H (See the section titled *Parallel Faults* in this chapter.)

Fault Subtype: No subtypes. (See Figure 7-5.)

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

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

RIP: IP of the instruction that would have executed next if faults had not been generated.

Program State Changes: The precision of faults recorded in a parallel fault record depends on the type of faults detected.

A change in the program's control flow may or may not accompany parallel faults, depending on the type of faults detected.

Protection Faults

Fault Type: 7H

Fault Subtype:

Bit Number	Name
Bit 0H	Reserved
Bit 1H	Length
Bit 2H-FH	Reserved

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

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

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

Program State Changes: This fault type is always precise, regardless of the value of the NIF bit.

A change in the program's control flow does not accompany the length fault, because the fault is generated before the faulting instruction.

Trace Faults

Fault Type: 1H

Fault Subtype:

Bit 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 that the processor has detected one or more trace events. The architecture's event tracing mechanism is described in detail in *Chapter 8, Tracing and Debugging.*

A trace event is the occurrence of a particular instruction or type of instruction in the instruction stream. The processor recognizes seven different trace events (instruction, branch, call, return, prereturn, supervisor, and breakpoint). It detects these events, however, only if a mode bit is set for the event in the trace-controls register. If, in addition, the trace-enable bit in the process-controls register is set, the processor generates a fault when a trace event is detected.

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

The following trace modes are available:

Instruction – Generates a trace event following every instruction.

Branch – Generates a trace event following any branch instruction when the branch is taken. (The branch trace event does not occur on branch-and-link or call instructions.)

Call – Generates a trace event following any call or branch-and-link instruction, or any implicit procedure call (i.e., fault- or interrupt-call).

Return – Generates a trace event following any **ret** (return) instruction.

Prereturn – Generates a trace event prior to any **ret** (return) instruction, providing the prereturn-trace flag in the PFP register is set. (The processor sets this flag automatically when prereturn tracing is enabled.)

Supervisor – Generates a trace event following any **calls** instruction that references a supervisor procedure entry in the system-procedure table, and on a return from a supervisor procedure where the return-status type in the PFP register is 010_2 or 011_2.

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).

There is a trace-fault subtype and a bit in the fault-subtype field associated with each of these modes. Multiple fault subtypes can occur simultaneously, with the fault-subtype bit set for each subtype that occurs.

When a fault type other than a trace fault is generated during the execution of an instruction that causes a trace event, the non-trace-fault is handled before the trace fault. An exception to this rule is the prereturn-trace fault. The prereturn-trace fault will occur before the processor has a chance to detect a non-trace-fault, so it is handled first.

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

The 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. 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 of the value of the NIF bit.

A change in the program's control flow accompanies all the trace faults (except the prereturn trace fault), because the 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 that a program or procedure has attempted to perform an illegal operation on an architecture-defined data type or a typed data structure.

The type-mismatch fault is generated when an attempt is made to modify the process-controls register with the **modpc** instruction while the processor is in the user mode.

The type-mismatch fault is also generated when an attempt is made to write to the on-chip data RAM while the processor is in the user mode.

The type-mismatch fault is also generated when an attempt is made to access a special-function register while the processor is in the user mode.

RIP: No defined value.

Program State Changes: These faults may be imprecise when executing with the NIF bit cleared.

A change in the program's control flow does not accompany the type-mismatch fault, because the fault occurs before the execution of the faulting instruction.

Tracing and Debugging 8

CHAPTER 8
TRACING AND DEBUGGING

This chapter describes the facilities of the 80960CA that allow monitoring of the run-time activity of the processor.

OVERVIEW OF THE TRACE-CONTROL FACILITIES

The 80960 architecture provides facilities for monitoring the activity of the processor through the generation of trace events. 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 state of the processor when the trace event occurred. This analysis can be used to locate software or hardware bugs or for general system monitoring during the development of a program.

Tracing is enabled by means of the trace-enable bit in the process-controls register and a set of trace-mode bits in the trace-controls register. Alternately, the **mark** and force mark (**fmark**) instructions can be used to generate trace events explicitly in the instruction stream.

The 80960CA 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 to enable or disable tracing in general. These controls are described in the following sections.

- Trace-controls register mode bits
- Trace-enable bit in the process-controls register.
- Trace-fault-pending flag in the process-controls register.
- Prereturn-trace flag (bit 0) in the return-status field of the previous frame pointer register, (PFP).

- Trace-control bit in the supervisor-stack-pointer field of the system-procedure table.
- The breakpoint mode bits and enable bits in the breakpoint-controls (BPCON) register, located in the control table.
- The address field in the instruction-address breakpoint (IPB0-IPB1) registers, located in the control table.
- The address field and the enable bit in the data-address breakpoint (DAB0 - DAB1) registers, located in the control table.

Trace-Controls Register

The trace-controls register (Figure 8-1) allows software to define the conditions under which trace events are generated.

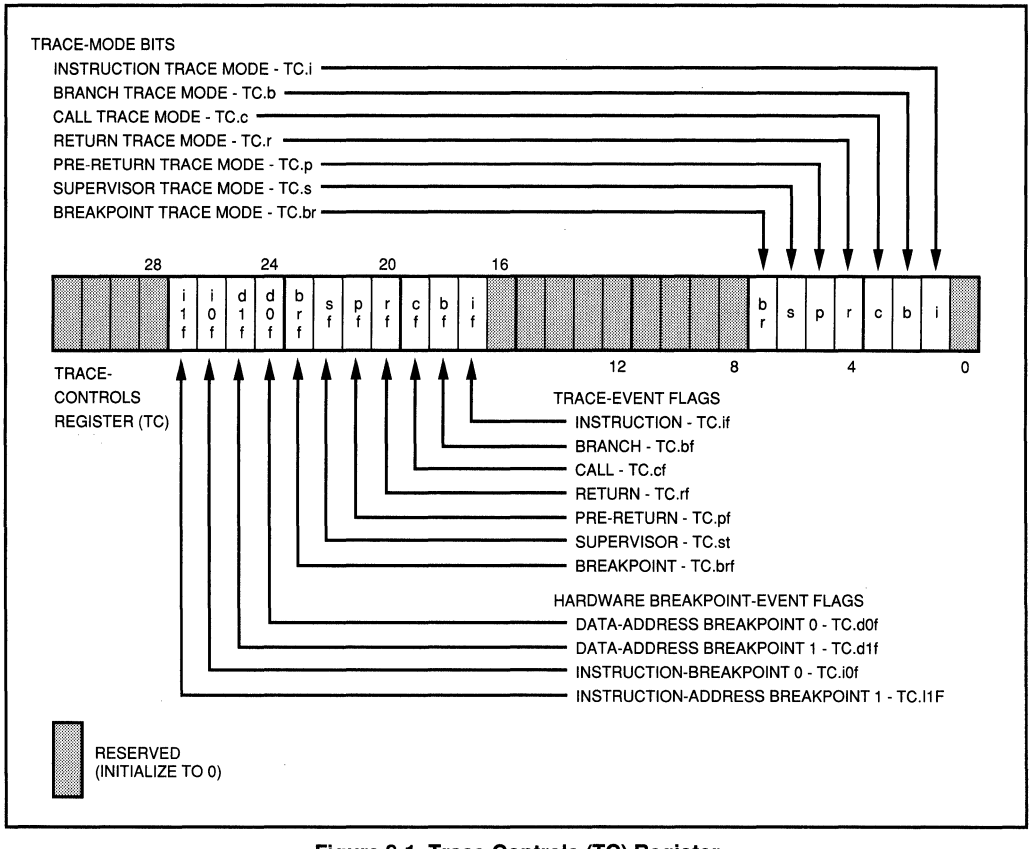


Figure 8-1. Trace-Controls (TC) Register

This register contains the mode bits and the event flags. The 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 is executed. (The trace modes are described later in this chapter in the section titled *Trace Modes.*) The processor uses the 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 in the TC register. On initialization, all the bits and flags in the TC register are cleared. The **modtc** instruction can then be used to set or clear trace-mode bits as required.

Software can also access the event flags using the **modtc** instruction; however, there is generally no reason to. The processor sets and clears these flags automatically as part of its trace-handling mechanism.

Bits 0, 8 through 16, and 28 through 31 of the trace-controls register are reserved. Software should initialize these bits to zero and not modify them afterwards.

Trace-Enable Bit and Trace-Fault-Pending Flag

The trace-enable bit and the trace-fault-pending flag, located in the process-controls register control tracing. The trace-enable bit enables the processor's tracing facilities. When this bit is 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 of the call and return operations that the processor carries out, as described at the end of this chapter in the section titled *Tracing and Interrupt Procedures.*

The trace-fault-pending flag allows the processor to keep track of the fact that a trace event has been detected for an enabled trace condition. The processor uses this flag as follows. When the processor detects a trace event and tracing is enabled, it sets the flag. Before executing an instruction, the processor checks the flag. If the flag is set, and tracing is enabled, it signals a trace fault. By providing a means of recording the occurrence of a trace event, 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**) is executed which results in a switch to supervisor mode. This action occurs independent of whether or not tracing is enabled prior to the call.

When a supervisor call is executed (**calls** instruction that references an entry in the system-procedure table with an entry type 110_2), the processor saves the current state of the trace-enable bit (from the process-controls register) in bit 0 of the return-type field of the PFP register. Then, it sets the trace-enable bit in the process-controls register to the state of the trace-control bit in the system-procedure table. The trace-control bit is located at bit 0 of byte 12 of the system-procedure table. When the trace-control bit is set, tracing is enabled on supervisor calls; when it is clear, tracing is disabled on supervisor calls.

On a return from the supervisor procedure, the trace-enable bit in the process-controls register is restored to the value saved in the return-type field of the PFP register.

TRACE MODES

The following trace modes can be enabled through the trace-controls register:

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

These modes can be enabled individually or several modes can be enabled at once. Some of these modes overlap, such as the call-trace mode and the supervisor-trace mode. The section later in this chapter titled *Handling Multiple Trace Events* describes what the processor does when multiple trace events occur.

The following sections describe each of the trace modes.

Instruction Trace

When the instruction-trace mode is enabled, the processor generates an instruction-trace event each time an instruction is executed. 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 is executed 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**) is executed. 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 (bit 3 of the PFP register) in the new frame created by the call operation or in the current frame if a branch-and-link operation was performed. The processor uses this flag to determine whether or not to signal a prereturn-trace event on a **ret** instruction.

Return Trace

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

Prereturn Trace

The prereturn-trace mode causes the processor to generate a prereturn-trace event prior to the execution of any **ret** instruction, providing the prereturn-trace flag in the PFP register 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 the 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 the supervisor-trace mode is enabled, the processor generates a supervisor-trace event any time 1) a call-system instruction (**calls**) is executed, where the procedure table entry is for a system-supervisor call, or 2) when a **ret** instruction is executed and the return-type field is set to 010_2 or 011_2 (i.e., return from supervisor mode).

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

Breakpoint Trace

The 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 the **mark** and force-mark (**fmark**) instructions, and the hardware breakpoint registers.

Software Breakpoints

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

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 least significant bit in IPB0 or IPB1. Figure 8-2 describes the instruction address breakpoint registers.

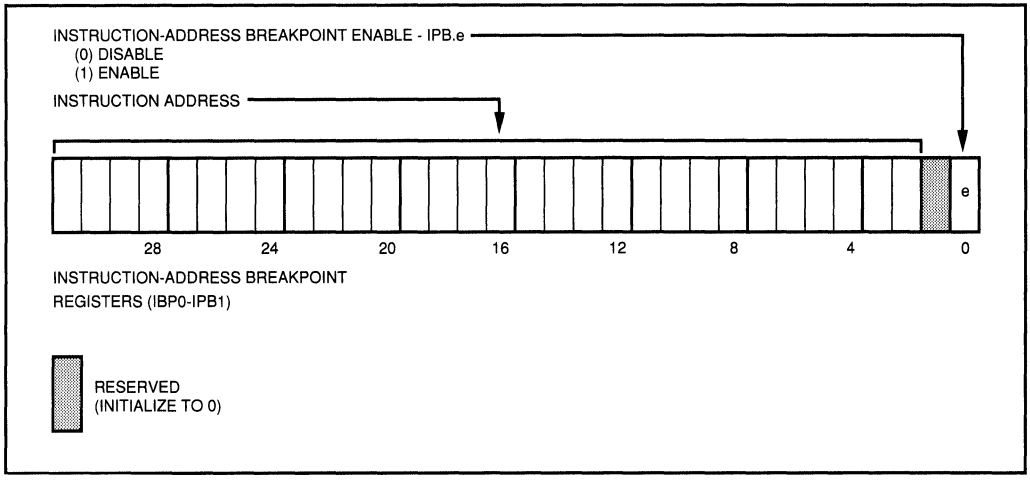


Figure 8-2. Instruction Address Breakpoint Registers (IPB0 - IPB1)

Breakpoint-trace events may also be generated when a memory access is issued which matches the conditions programmed in one of the 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.

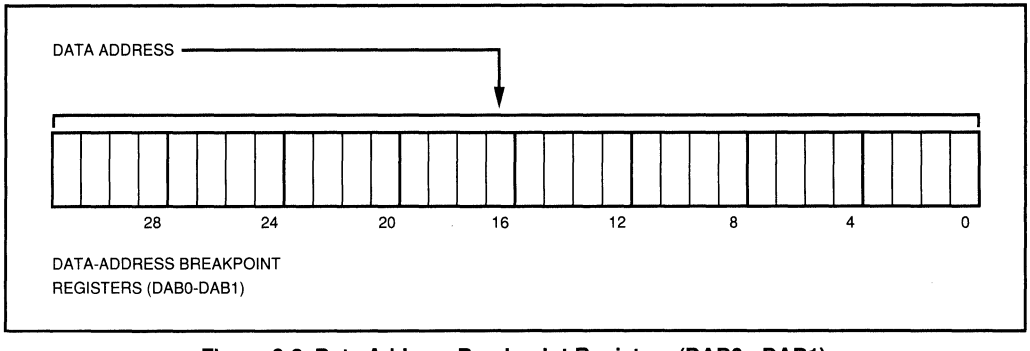


Figure 8-3. Data Address Breakpoint Registers (DAB0 - DAB1)

The programmer configures the breakpoint control (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 enable bits in BPCON.

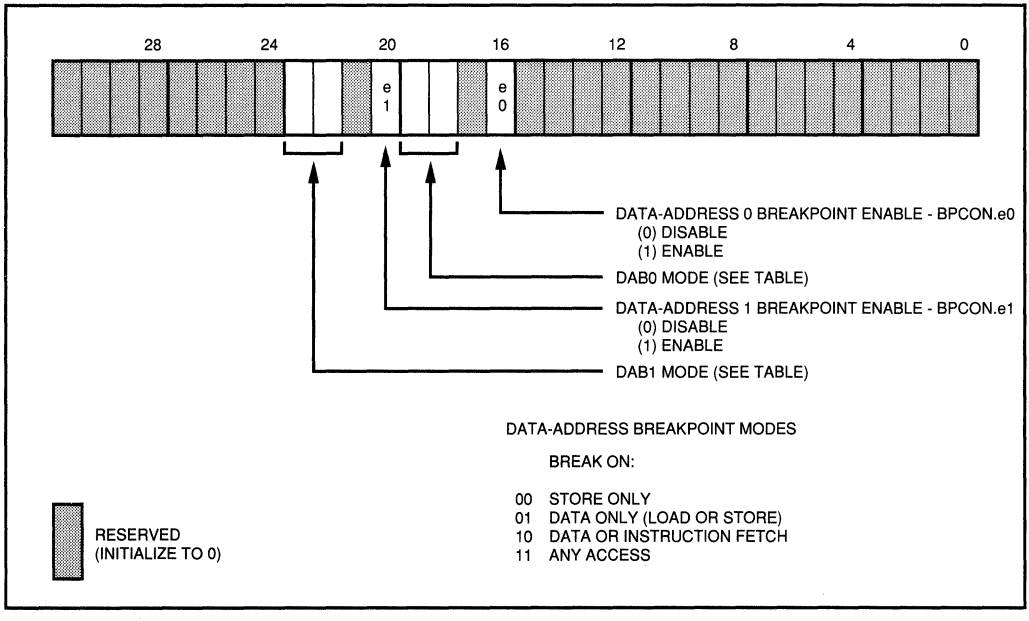


Figure 8-4. Hardware Breakpoint Control Register (BPCON)

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 the **sysctl** instruction. Control registers are described in detail in *Chapter 2, Programming Environment.*

When the processor attempts an access which is set for detection (instruction or data breakpoint), a breakpoint-trace event is signaled. If breakpoint-trace is enabled by the breakpoint trace mode bit in the trace-controls register (TC.br), the appropriate hardware breakpoint trace-event flags in the trace controls register are set. If tracing is enabled, a trace fault is generated after the faulting instruction completes execution.

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 is executed or about to be executed (in the case of a prereturn trace event) and the trace mode for that instruction is enabled.
- An implicit call operation has been executed and the call-trace mode is enabled.
- A **mark** instruction has been executed and the breakpoint-trace mode is enabled.
- An **fmark** instruction has been 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 trace-enable bit in the process-controls register is set, the processor performs the following action:

1) The processor sets the appropriate trace-event flag in the trace-controls register. 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 trace-fault-pending flag in the process-controls register.

Note: The processor may set a trace-event flag and the trace-fault-pending flag before it has completed execution of the instruction that caused the event. However, the processor only handles trace events between the execution of instructions.

If, when the processor detects a trace event, the trace-enable bit in the process-controls register is clear, the processor sets the appropriate event flags, but does not set the 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 will at least signal 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. The general requirements for fault-handling procedures are given in *Chapter 7, Faults.*

The trace-fault-handling procedure must be involved in a specific way and is handled slightly differently than other faults. A trace-fault-handler must be involved with an implicit system-supervisor call. When the call is made, the trace-enable bit in the process-control register 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 trace-control bit located in the system-procedure table. The exceptional handling of the 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 has been signaled, the processor determines how to handle the trace event, according to the setting of the trace-enable bit and trace-fault-pending flags in the process-controls register and to other events that might occur simultaneously with the trace event such as an interrupt or a non-trace fault.

The following sections 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. If this flag is clear, the processor begins execution of the next instruction. If the flag is set, the processor performs the following actions.
2) The processor checks the state of the trace-enable bit in the process-controls register. If the trace-enable bit is clear, the processor clears any trace-event flags that have been set, prior to starting execution of the next instruction.
3) If the trace-enable bit is set, the processor signals a trace fault and begins the 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 prereturn-trace flag in the PFP register. If this flag is 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 current state of the process-controls register, then clearing the trace-enable bit and trace-fault-pending flag in that register.

On returning from the interrupt-handling procedure, the processor restores the process-controls register to the state it was in prior to handling the interrupt, which restores the state of the trace-enable bit and trace-fault-pending flag. If these two flags were set prior to calling the interrupt procedure, a trace fault will be signaled on the 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.

The action described above 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 of the instructions for the processor. To provide quick access to information on a particular instruction, the instructions are listed alphabetically by assembly-language mnemonic. An explanation of the format and notation used in this chapter is given in the following section titled *Notation.*

INTRODUCTION

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

- Alphabetic reference
- Assembly-language mnemonic and name
- Assembly-language 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* - Summary of the instruction set by group and description of the assembly-language instruction format
- *Appendix I, Instruction Set Quick Reference* - A tabular quick-reference of each instruction's operation and side-effects.
- *Appendix D, Instruction Encoding Reference* - A complete description of the opword encodings of the instruction set. A quick-reference listing of instruction encodings is also provided to assist debug with a logic analyzer.

NOTATION

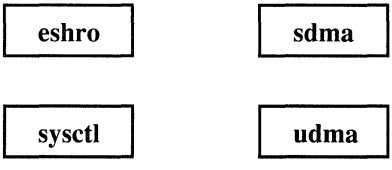
To simplify the presentation of information about the instructions, a simple notation has been adopted in this chapter. The following paragraphs describe this notation.

Alphabetic Reference

The 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 reference at the top of each page gives the assembly-language mnemonics for the instructions covered on that page (e.g., **subc**). Occasionally it is not practical to give all the mnemonics in the page reference. In these cases, the name of the instruction group is given in capital letters (e.g., BRANCH or FAULT IF)

The 80960CA extensions to the 80960 instruction set are indicated with a box around the instruction's alphabetic reference. The following 80960CA instructions are extensions:



Instruction set extensions are generally not portable to other 80960 implementations.

Mnemonic

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

subi Subtract Integer

The CTRL and COBR format instructions also allow the optional **.t** or **.f** mnemonic suffixes for branch prediction. The .t suffix indicates to the processor that the condition the instruction is testing for is likely to be true. The .f suffix 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 a detailed discussion, see the section of *Chapter 2, Programming Environment* titled *Branch Prediction*, and *Appendix B, Optimizing Code for the 80960CA*.

When the programmer provides no suffix with an instruction which supports a suffix, the assembler is free to make its own prediction.

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

be{.t|.f} Branch If Equal

Format

The *Format* section gives the assembly-language format of the instruction and the type of operands allowed. The format is given in two or three lines. The following is an example of a two line format:

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

The first line gives the assembly-language mnemonic (bold-face type) and the operands (italics). When the format is used for two or more instructions, an abbreviated form of the mnemonic is used. The "*" sign at the end of the mnemonic indicates that the mnemonic has been abbreviated.

The operand names are designed to describe the functions of the operands (e.g., *src*, *len*, *mask*).

The second line of the format shows what is allowed to be entered for each operand. The notation used on this line 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 80960 architecture allows up to a total of 32 Special Function Registers. However, sf0, sf1 and sf2 are the only Specific Function Registers implemented on the 80960CA.

In some cases, a third line will be added to show specifically what will be in a register or memory location. 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, in detail, the direct effects and the possible side effects of executing and instruction. The algorithms document the net effect of the instruction on the programming environment; they do not necessarily describe how the processor actually implements the instruction. For example, the **shli** instruction 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 80960CA implementation 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 means the condition-code field in the arithmetic controls register. AC.cc1 means bit 1 of this field. The symbol "^" indicates an exponent; for example, 2^(*bitpos* **mod** 32) is equivalent to $2^{(bitpos \bmod 32)}$.

Table 9-1 lists the meaning of 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 may change independent of instruction execution, the following distinctions are important when interpreting the algorithm of any instruction which references an *sfr*.

(1) When a source operand is an *sfr*, and it is 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 will operate 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 an *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 field (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

←	Assignment
=,≠, <, >, <=, >=	Comparison
<<, >>	Logical Shift
^	Exponentiation
and,or, not, xor	Bitwise Logical Operations
mod	Modulo
+, -	Addition, Subtraction
*	Multiplication (Integer or Ordinal)
/	Division (Integer or Ordinal)
#. .	Comments
memory()	Memory access of specified width memory_{byte|short|word|long|triple|quad}() memory() Width implied by context

Faults

The *Faults* section lists the faults that can be signaled as a direct result of execution of the instruction.

Two possible faulting conditions are common to the entire instruction set. These faults could directly result from any instruction:

Fault Type	Sub-type/Description
Trace	*Instruction.* An Instruction Trace-Event is signaled after completion of the instruction. A Trace fault is generated if PC.te and TC.i are both set to 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 set to 1.
Operation	*Unimplemented.* An attempt to execute any instruction fetched from internal data-ram will cause an operation unimplemented fault.

The above descriptions apply to the entire instruction set, and are abbreviated in the instruction reference.

The following three possible faulting conditions are common to large subsets of the instruction set.

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

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 Encoding Reference* for more information on the formats.

See Also

The *See Also* section gives the mnemonics of related instructions, which can be found listed alphabetically 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,*	*src2,*	*dst*
	reg/lit/sfr	reg/lit/sfr	reg/sfr

Description:

Adds the *src2* and *src1* values, and bit 1 of the condition code (used here as a carry in), and stores the result in *dst*. If the ordinal addition results in a carry, bit 1 of the condition code is set; otherwise, bit 1 is cleared. If integer addition results in an overflow, bit 0 of the condition code is set; otherwise, bit 0 is cleared. Regardless of the results of the addition, bit2 of the condition codes always written to a 0.

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

An integer overflow fault is never signaled with this instruction.

Action:

dst ← *src2* + *src1* + AC.cc1;
AC.cc0 ← $0CV_2$;
V is 1 if integer addition would have generated an overflow.
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 an *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 the *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 an sfr.
Arithmetic	*Integer Overflow*. Result is too large for the destination register (**addi** only). If overflow occurs, and AC.om is a 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:

```
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*,	*src*,	*dst*
	reg/lit/sfr	reg/lit/sfr	reg/sfr

Description: Copies the *src* value to *dst* with one bit altered. The *bitpos* operand specifies the bit to be changed; the condition code determines the value the bit is to be changed to. If condition code bit 1 is a 1, the 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 an 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 the *src2* and *src1* values and stores the result in *dst*. Note in the action expressions below, the *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 an 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	*src/dst*, reg/sfr addr	*src*, reg/lit/sfr	*dst* reg/sfr

Description:

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

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

The memory location in *src/dst* is the address of the first byte (least significant byte) of the word. The address is automatically aligned to a word boundary. (Note that the *src/dst* operand maps to the *src1* operand of the REG format.)

Action:

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

temp ← memory_word (tempa);  # LOCK asserted at beginning of read

memory_word (tempa) ← temp + src; # ordinal addition
                                  # 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 an *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*, reg/sfr addr	*mask*, reg/lit/sfr	*src/dst* reg/sfr

Description:

Moves selected bits of the *src/dst* value into the memory location specified in *src*. The bits set in the *mask* operand select the bits to be modified in memory. The initial value from memory is stored in *src/dst*.

The read and write of memory 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 the operation has been completed).

The memory location in *src* is the address of the first byte (least significant byte) of the word to be modified. The 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 an 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, the IP specified with the *targ* operand can be no farther than -2^{23} to (2^{23} - 4) bytes from the current IP. When using the Intel 80960 assembler, the *targ* operand must be a label, which specifies the IP of the target instruction.

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

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 completion of the instruction. If PC.te is a 1, and, TC.i or TC.b is set to 1, a Trace fault is generated.
	Operation	*Unimplemented.* Execution from on-chip data RAM.
		Operand. An invalid operand value was encountered. (**bx** only)
		Opcode. An invalid operand encoding was 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 the address of the instruction following the **bal** or **balx** then branches to the specified target.

With the **bal** instruction, the address of the next instruction is stored in register g14. The *targ* operand value can be no farther than -2^{23} to (2^{23} - 4) bytes from the current IP. When using the Intel 80960 assembler, the *targ* operand must be a label, which specifies the IP of the target instruction.

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

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

Action:

bal:
```
g14 ← IP + 4;                    # destination of next IP is always g14
IP ← IP + displacement;          # resume execution at the 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 completion of the instruction. If PC.te is a 1, and, TC.i or TC.br is set to 1, a Trace fault is generated.

Operation — *Unimplemented.* Execution from on-chip data RAM.

Operand. An invalid operand value was encountered.

Opcode. An invalid operand encoding was 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		
bb*{.t|.f}	*bitpos*,	*src*,	*targ*
	reg/lit/sfr	reg	disp

Description:

Checks the bit in *src* (designated by *bitpos*) and sets the condition code in the arithmetic controls according to the value found. The processor then performs a conditional branch to the instruction specified with the *targ* operand, according on the state of the condition code.

The optional **.t** or **.f** suffix may be appended to the mnemonic. Use the **.t** suffix to speed-up execution when these instructions usually take the branch. Use the **.f** suffix to speed-up execution when these instructions usually do not take the branch. If a suffix is not provided, the assembler is free to provide one.

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

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

The *targ* operand can be no farther than -2^{12} to $(2^{12} - 4)$ bytes from the current IP. When using the Intel 80960 assembler, the *targ* operand must be a label, which specifies the IP of the target instruction.

Action:

bbc:

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

bbs:

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

Faults:

Trace	*Instruction. Branch* (if taken). *Breakpoint.* Instruction and Branch Trace-Events are signaled after completion of the instruction. If PC.te is a 1, and, TC.i or TC.b is set to 1, a Trace fault is generated.
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of an *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 the instruction specified with the *targ* operand, according to the state of the condition code in the arithmetic controls.

The optional **.t** or **.f** suffix may be appended to the mnemonic. Use the **.t** suffix to speed-up execution when these instructions usually take the branch. Use the **.f** suffix to speed-up execution when these instructions usually do not take the branch. If a suffix is not provided, the assembler is free to provide one.

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

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

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

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

The *targ* operand value or absolute addresses can be no farther than -2^{23} to $(2^{23} - 4)$ bytes from the current IP. When using the Intel 80960 assembler, the *targ* operand must be a label, which specifies the IP of the target instruction.

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

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 completion of the instruction. If PC.te is a 1, and, TC.i or TC.b is set to 1, a Trace fault is generated.
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. The *targ* operand specifies the IP of the first instruction of the called procedure. When using the Intel 80960 assembler, the *targ* operand must be a label.

In executing this instruction, the processor performs a local call operation as described in the section of *Chapter 5, Procedure Calls* titled *Local 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. The processor then goes to the instruction specified with the *targ* argument and begins execution of the new procedure.

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

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 the
                                    # 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 completion of the instruction. If PC.te is a 1, and, TC.i or TC.c is set to 1, a Trace fault is generated.
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. The *targ* operand gives the number of the procedure being called.

For this instruction, the processor performs the system call operation described in the section of *Chapter 5, Procedure Calls* titled *System Calls*. The *targ* operand provides an index to an entry in the system-procedure table. From this entry, the processor gets the IP of the called procedure.

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

As part of this operation, the processor allocates a new set of local registers and a new stack frame for the called procedure. If the processor switches to the 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
        temp_FP ← (SP + 0x10) and not(0xf);      # round to next
        boundary,
        temp_rt ← 000₂;                          # return type is
                                                 local
        }
else
        { # stack switch to supervisor stack required
        temp_FP ← memory_word(SPT.base + 12); # read
                                                 supervisor
                                                 # 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;
        PC.te ← temp_FP.T;                          # Trace enable bit of the
                                                    supervisor
                                                    # stack pointer is written
                                                    to PC.te
        }
memory(FP) ← r0:15 # These accesses are cached in the local register
cache.
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 completion of the instruction. If PC.te is a 1, and, TC.i or TC.c or TC.s is set to 1, a Trace fault is generated.
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of an *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 a new procedure. The *efa* operand specifies the IP of the first instruction of the called procedure.

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

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

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

Refer to *Chapter 3, Data Types and Memory Addressing Modes* for a complete discussion of the 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 the
                                    # 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 completion of the instruction. If PC.te is a 1, and, TC.i or TC.c is set to 1, a Trace fault is generated.
Operation	*Unimplemented.* Execution from on-chip data RAM.
	Operand. An invalid operand value was encountered.
	Opcode. An invalid operand encoding was 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*,	*src*
	reg/lit/sfr	reg/lit/sfr

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

Action:

if ((*src* **and** 2^(*bitpos* **mod** 32)) = 0) AC.cc ← 000_2;
else AC.cc ← 010_2;

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of an *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*,	*src*,	*dst*
	reg/lit/sfr	reg/lit/sfr	reg/sfr

Description: Copies the *src* value to *dst* with one bit cleared. The *bitpos* operand specifies the 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 an *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 the *src2* and *src1* values and sets the condition code according to the results of the comparison. The *src2* operand is then decremented by one and the result is stored in *dst*.

The following table shows the setting of the condition code 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 the **cmpdeci** instruction, integer overflow is ignored to allow looping down through the minimum integer values.

Action:

if (*src1* < *src2*) AC.cc ← 100_2;
else if (*src1* = *src2*) AC.cc ← 010_2;
else AC.cc ← 001_2;
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 an *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*,	*src2*
	reg/lit/sfr	reg/lit/sfr

Description:

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

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

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

Action:

if (*src1* < *src2*) AC.cc ← 100_2;
else if (*src1* = *src2*) AC.cc ← 010_2;
else AC.cc ← 001_2;

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of an *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 the *src2* and *src1* values and sets the condition code according to the results of the comparison. The *src2* operand is then incremented by one and the result is stored in *dst*.

The following table shows the setting of the condition code 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 the **cmpinci** instruction, integer overflow is ignored to allow looping up through the maximum integer values.

Action:

if (*src1* < *src2*) AC.cc ← 100_2;
else if (*src1* = *src2*) AC.cc ← 010_2;
else AC.cc ← 001_2;
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 an *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 Unordered
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/sfr	reg/sfr	disp
cmpob*{.t|.f}	*src1*,	*src2*,	*targ*
	reg/lit/sfr	reg/sfr	disp

Description:

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

The optional **.t** or **.f** suffix may be appended to the mnemonic. Use the **.t** suffix to speed-up execution when these instructions usually take the branch. Use the **.f** suffix to speed-up execution when these instructions usually do not take the branch. If a suffix is not provided, the assembler is free to provide one.

The *targ* operand can be no farther than -2^{12} to $(2^{12} - 4)$ bytes from the current IP. When using the Intel 80960 assembler, the *targ* operand must be a label, which specifies the IP of the target instruction.

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: The **cmpibo** instruction always branches; the **cmpibno** instruction never branches.

The functions that these instructions perform can be duplicated with a **cmpi** or **cmpo** instruction followed by a branch-if instruction, as described in the reference page for the **cmpi** and **cmpo** instructions in this chapter.

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 completion of the instruction. If PC.te is a 1, and, TC.i or TC.br is set to 1, a Trace fault is generated.
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of an *sfr*.

Example:

```
# assume g3 < g9
cmpibl g3, g9, xyz    # g9 is compared with g3;
                      # IP ← xyz.
# assume r7 ≥19
cmpobge r7, 19, 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 the *src2* and *src1* values if bit 2 of the condition code is not set. If the comparison is performed, the condition code is set according to the results of the comparison. Otherwise, the 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 the value in g3 is between the values in g5 and g6, 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), the 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 an *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 the *src2* value by the *src1* value and stores the quotient of the result in *dst*. The remainder (if there is one) is discarded.

For the **divi** instruction, and 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 an *sfr*.
Arithmetic	*Zero Divide.* The *src1* operand is 0.
	Integer Overflow. Result is too large for the destination register (**divi** only). If overflow occurs, and AC.om is a 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:

```
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 the result in *dst*. The *src2* value is a long ordinal (i.e., 64 bits), which is contained in two adjacent registers. The *src2* operand specifies the lower numbered register, which contains the least significant bits of the operand. The *src2* operand must be an even numbered register (i.e., r0, r2, r4, ... or g0, g2, ... or sf0, sf2, ...). The *src1* value is a normal ordinal (i.e., 32 bits).

The result consists of a one-word remainder and a one-word quotient. The remainder is stored in the register designated by *dst* and the quotient is stored in the next highest numbered register. The *dst* operand 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 (i.e., the 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 an *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*,	*src2*,	*dst*
	reg/lit/sfr	reg/lit/sfr	reg/sfr

Description: Multiplies *src2* by *src1* and stores the result in *dst*. The result is a long ordinal (i.e., 64 bits), which is stored in two adjacent registers. The *dst* operand specifies the lower numbered register, which receives the least significant bits of the result. The *dst* operand 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 an *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*,	*src2*,	*dst*
	reg/lit/sfr	reg/lit/sfr	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.

The *src2* value is a long ordinal (i.e., 64 bits), which is contained in two adjacent registers. The *src2* operand specifies the lower numbered register, which contains the least significant bits of the operand. The *src2* operand must be an even numbered register (i.e., r0, r2, r4, ... or g0, g2, ... or sf0, sf2, ...).

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

The least-significant 32-bits of the result of the shift operation 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 an *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*,	*len*,	*src/dst*
	reg/lit/sfr	reg/lit/sfr	reg/sfr

Description: Shifts a specified bit field in *src/dst* right and fills the bits to the left of the shifted bit field with zeros. The *bitpos* value specifies the least significant bit of the bit field to be shifted, and the *len* value specifies the length of the bit field.

Action: *src/dst* ← (*src/dst* >> (*bitpos* **mod** 32)) **and** (2^(*len* **mod** 32) - 1);

Faults:

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

Example:

```
extract 5, 12, g4    # g4 ← g4 with bits 5 through 16
                     # shifted right
```

Opcode: **extract** 65111 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 Unordered

Format:

fault*{.t|.f}

Description:

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

The optional **.t** or **.f** suffix may be appended to the mnemonic. Use the **.t** suffix to speed-up execution when these instructions usually fault. Use the **.f** suffix to speed-up execution when these instructions usually do not fault. If a suffix is not provided, the assembler is free to provide one.

The **faulto** and **faultno** instructions are provided for use by implementations that include 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 in bits 0-2 of the opcode.

Instruction	Msk	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) $\neq 000_2$) Constraint-range fault;

faultno:

if (AC.cc = 000_2) 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.

The **flushreg** instruction is provided to allow a compiler or applications program to circumvent the normal call/return mechanism of the processor. For example, a compiler may need to back-up 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. This instruction causes a breakpoint trace-event to be generated, regardless of the setting of the breakpoint trace mode flag, providing the trace-enable bit (bit 0) in the process-controls register is set.

When a breakpoint-trace event is detected, the trace-fault-pending flag (bit 10) in the process-controls register and the breakpoint-trace-event flag (bit 23) in the trace-controls register are set. Then, before the next instruction is executed, a breakpoint-trace fault is generated.

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 completion of the instruction. If PC.te is a 1 a Trace fault is generated.
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 a byte or string of bytes from memory into a register or group of successive registers. The *efa* operand specifies the address of the first byte to be loaded. 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 of the addressing modes.)

The *dst* operand specifies a register or the first (lowest numbered) register of successive registers.

The **ldob** and **ldib**, and **ldos** and **ldis** instructions load a byte and half word, respectively, and convert it to a full 32-bit word. The data being loaded is sign-extended during the integer loads, and zero-extended during the ordinal loads.

The **ld**, **ldl**, **ldt**, and **ldq** instructions copy 4, 8, 12, and 16 bytes, respectively, from memory into successive registers.

For the **ldl** instruction, *dst* must specify an even numbered register (e.g., g0, g2, ... or r0, r2, ...). For the **ldt** and **ldq** instructions, *dst* must specify a register number that is a multiple of four (e.g., g0, g4, g8, ... or r0, r4, r8, ...). If the registers are not aligned on the required boundary, the results are unpredictable. If the data extends beyond register g15 or r15 for the **ldl**, **ldt**, or **ldq** instruction, the results are unpredictable.

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 was referenced, and unaligned support was disabled.
	Operand. An invalid operand value was encountered.
	Opcode. An invalid opcode encoding was 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*. The value computed 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, the move instruction (**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. An invalid operand value was encountered.
	Opcode. An invalid opcode encoding was 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 a breakpoint-trace event if the breakpoint-trace mode has been enabled. The breakpoint-trace mode is enabled if the trace-enable bit (bit 0) in the process-controls register and the breakpoint-trace mode bit (bit 7) in the trace-controls register have been set.

When a breakpoint-trace event is detected, the trace-fault-pending flag (bit 10) in the process-controls register and the breakpoint-trace-event flag (bit 23) in the trace-controls register are set. Then, before the next instruction is executed, a breakpoint-trace fault is generated.

If the breakpoint-trace mode has not been enabled, the **mark** instruction 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 completion of the instruction. If PC.te is a 1, and, TC.i or TC.br is set to 1, a Trace fault is generated.
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*,	*src*,	*dst*
	reg/lit/sfr	reg/lit/sfr	reg/sfr

Description: Reads and modifies the arithmetic-controls register. The *src* operand contains the value to be placed in the arithmetic-controls register and the *mask* operand specifies the bits that may be changed. Only the bits set in *mask* are modified. Once the arithmetic-controls register has been 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 an *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 an *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*,	*src*,	*src/dst*
	reg/lit/sfr	reg/lit/sfr	reg/sfr

Description: Modifies selected bits in *src/dst* with bits from *src*. The *mask* operand selects the bits to be modified: only the 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 an *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/sfr

Description:

Reads and modifies the process-controls register as specified with *mask* and *src/dst*. The *src/dst* operand contains the value to be placed in the process-controls register and the *mask* operand specifies the bits that may be changed. Only the bits set in the mask are modified. Once the process-controls register has been 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 the supervisor mode to use this instruction with a non-zero *mask* value. If the *mask* operand is set to 0, this instruction can be used to read the process controls, without the processor being in the supervisor mode.

If the action of this instruction results in the priority of the processor being lowered, the interrupt table is checked for pending interrupts.

Changing the reserved fields of the process-controls register 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 an *sfr*.
		Mismatch. Attempted to execute this instruction with a 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 the trace-controls register as specified with *mask* and *src*. The *src* operand contains the value to be placed in the trace-controls register and the *mask* operand specifies the bits that may be changed. Only the bits set in the mask are modified. The *mask* operand must not enable modification of reserved bits. Once the trace-controls register has been 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 an *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**

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 content of one or more source registers (specified with the *src* operand) to one or more destination registers (specified with the *dst* operand).

For the **movl**, **movt**, and **movq** instructions, the *src* and *dst* operands specify the first (lowest numbered) register of several successive registers. The *src* and *dst* registers must be even numbered (e.g., g0, g2, ... or r0, r2, ... or sf0, sf2, ...) for the **movl** instruction and an integral multiple of four (e.g., g0, g4, ... or r0, r4, ... or sf0, sf4, ...) for the **movt** and **movq** instructions.

When the *src* and *dst* operands overlap, the value moved is unpredictable. Also, if registers are not properly aligned, the value moved is unpredictable.

Action:

dst ← *src*;

Faults:

Trace	*Instruction. Breakpoint.*
Operation	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of an *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 an *sfr*.
Arithmetic	*Integer Overflow.* Result is too large for the destination register (**muli** only). If overflow occurs, and AC.om is a 1, the fault is and AC.io is set to a 1. The least significant 32-bits of the result 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*,	*src2*,	*dst*
	reg/lit/sfr	reg/lit/sfr	reg/sfr

Description: Performs a bitwise NAND operation on the *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 an *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*,	*src2*,	*dst*
	reg/lit/sfr	reg/lit/sfr	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 an *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*, reg/lit/sfr	*dst* reg/sfr	
notand	*src1*, reg/lit/sfr	*src2*, reg/lit/sfr	*dst* 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 an *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 an *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*,	*src2*,	*dst*
	reg/lit/sfr	reg/lit/sfr	reg/sfr

Description: Performs a bitwise NOT OR operation on the *src2* and *src1* values and stores the 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 an *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*, reg/lit/sfr	*src2*, reg/lit/sfr	*dst* reg/sfr
ornot	*src1*, reg/lit/sfr	*src2*, reg/lit/sfr	*dst* 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 an *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 an *sfr*.
Arithmetic	*Zero Divide.* The *src1* operand is 0
	Integer Overflow. Result is too large for the destination register (**remi** only). If overflow occurs, and AC.om is a 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:

```
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 stack frame of the calling procedure. 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 the **ret** instruction.

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 Operation Unimplemented fault;
FP ← PFP;
r0:15 ← memory(FP);          # these accesses are cached in the local
                               register cache
IP ← RIP;
```

Faults:	Trace	*Instruction. Return. Pre-Return. Breakpoint.* Instruction, Return and Pre-Return Trace-Events are signaled after completion of the instruction. If PC.te is a 1, and, TC.i or TC.r or TC.p is set to 1, a Trace fault is generated.
	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*,	*src*,	*dst*
	reg/lit/sfr	reg/lit/sfr	reg/sfr

Description: Copies *src* to *dst* and rotates the bits in the resulting *dst* operand to the left (toward higher significance). (The bits shifted off the left end of the word are inserted at the right end of the word.) The *len* operand specifies the number of bits that the *dst* operand is rotated. The *len* operand 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 an *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*
	reg/lit/sfr	reg/sfr

Description: Searches the *src* value for the most-significant set bit (1 bit). If a most-significant 1 bit is found, its bit number is stored in *dst* and the condition code is set to 010_2. If the *src* value is zero, all 1's are stored in *dst* and the 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 an *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*,	*src2*
	reg/lit/sfr	reg/lit/sfr

Description: Performs a byte-by-byte comparison of *src1* and *src2* and sets the condition code to 010_2 if any two corresponding bytes are equal. If no corresponding bytes are equal, the 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 an *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/sfr

Description:

The control for the DMA channel specified by *src1* is written with the value in *src2*. The dedicated data RAM for the specified DMA channel is written with the *src3* value. The first two-bits of *src1* specify the channel. The *src2* operand specifies the control parameters as a literal, or as a single 32-bit register. The *src3* operand specifies a single 32-bit register if the channel is data-chaining. If the channel is not data-chaining, the *src3* operand specifies a quad-word, contained in *src3*, *src3+1*, *src3+2* and *src3+3*. When not data chaining, *src3* must specify a register with a register number divisible by four.

Action:

```
dma_control_for_channel[src1 mod 4] ← src2;
if (not chaining mode)
        dma_ram[src1 mod 4] ← src3;     # quad-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
                                    #and byte count from memory
sdma r6, r7, r8                     # 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 the *src* value to *dst* with one bit set. The *bitpos* operand specifies the 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 an *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 the register boundary are discarded. For values of *len* greater than 32, the processor interprets the value as 32.

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

The **shli** instruction shifts zeros in from the least-significant bit. If the bits shifted out are not the same as the most-significant bit (bit 31), an overflow fault is generated. If overflow occurs, *dst* will equal *src* shifted left as much as possible without overflowing.

The **shri** instruction performs a conventional arithmetic shift-right operation by shifting in the most-significant bit (bit 31). When this instruction is used to divide an negative integer operand by the power of 2, it produces an incorrect quotient. (The discarding of the bits shifted out has the effect of rounding the result toward negative.)

The **shrdi** instruction 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.

The **shli** and **shrdi** instructions are equivalent to **muli** and **divi** by the power of 2.

The **eshro** instruction 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 an *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*
	reg/lit/sfr	reg/sfr

Description: Searches the *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 the condition code is set to 010_2. If the *src* value is all 1's, all 1's are stored in *dst* and the 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 an *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. The *src* operand specifies a register or the first (lowest numbered) register of successive registers.

The *efa* operand specifies the address of the memory location where the byte or the 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.)

The **stob** and **stib**, and **stos** and **stis** instructions store a byte and half word, respectively, from the low order bytes of the *src* register. The 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, and Arithmetic Integer Overflow fault is signaled.

The **st**, **stl**, **stt**, and **stq** instructions copy 4, 8, 12, and 16 bytes, respectively, from successive registers to memory.

For the **stl** instruction, *src* must specify an even numbered register (e.g., g0, g2, ... or r0, r2, ...). For the **stt** and **stq** instructions, *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 was referenced, and unaligned support was disabled.
	Operand. An invalid operand value was encountered.
	Opcode. An invalid opcode encoding was encountered.
Arithmetic	*Integer Overflow.* Result is too large for the destination (**stib** and **stis** 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, the destination will contain 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 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, bit 0 of the condition code is set.

The **subc** instruction does not distinguish between ordinals and integers: it sets bits 0 and 1 of the condition code regardless of the data type.

Action:

dst ← *src2* - *src1* - **not**(AC.cc1);
AC.cc ← $0CV_2$;
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 an *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 an *sfr*.
Arithmetic	*Integer Overflow.* Result is too large for the destination register (**subi** only). The least-significant 32-bits of the result is stored in *dst*. If overflow occurs, and AC.om is a 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

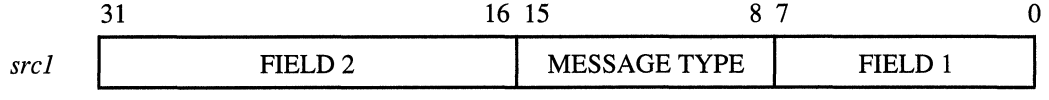
Mnemonic: **sysctl** System Control

Format:

sysctl	*src1*, reg/lit/sfr *message, type*	*src2*, reg/lit/sfr	*src3*; reg/lit/

Description: The processor control function specified by the *message* field of *src1* is executed. The *type* field of *src1* is interpreted depending upon the command. The remaining bits of *src1* are reserved. The *src2* and *src3* operands are also interpreted depending upon the command.

The *src1* operand is interpreted as follows:



The following table lists the commands implemented on the 80960CA.

Message	*Src1*			*Src 2*	*Src 3*
	Type	Field 1	Field 2	Field 3	Field 4
Request Interrupt	00H	Vector No.	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 No.	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; i≤3, 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

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 Unordered

Format:

test*{.t|.f}	*dst*
	reg/sfr

Description:

Stores a true (01H) in *dst* if the logical AND of the condition code and the mask-part of the opcode is not zero. Otherwise, the instruction stores a false (00H) in *dst*. For the **testno** instruction (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 the **.t** suffix to speed-up execution when these instructions usually store a true (1) condition is *dst*. Use the **.f** suffix 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	*Unimplemented.* Execution from on-chip data RAM.
Type	*Mismatch.* Non-supervisor reference of an 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.
Constraint	*Privileged.* Attempt to execute while not in supervisor mode.

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*, reg/lit/sfr	*src2*, reg/lit/sfr	*dst* reg/sfr
xor	*src1*, reg/lit/sfr	*src2*, reg/lit/sfr	*dst* 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 an *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

SYSTEM IMPLEMENTATION

PART II

Bus Controller 10

CHAPTER 10
THE BUS CONTROLLER

This chapter discusses the attributes and programming of the 80960CA high performance, integrated 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 and stores; DMA-transfer requests; and accesses to unaligned memory. Management of the bus is accomplished by queuing bus requests which effectively decouples instruction execution speed from external memory access time.

Load and store instructions are the programs interface to the bus controller and work on ordinal (unsigned) or integer (signed) data. A single load or store instruction can move from one to 16 bytes of data. Instruction fetches are also handled by the bus controller. An instruction fetch reads either the 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 the 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) long. The purpose of configurable memory regions is to provide system hardware interface support. Regions are transparent to the software. The upper four bits of the address (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 the encoding of the byte enable signals and the physical location of the data on the 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. Wait states can be specified between the address and data cycles, between consecutive data cycles of burst accesses, and between the last data cycle and the address cycle of the next access for each of the memory regions. These wait states are programmable independently by region.

An external, memory-ready input is provided to permit 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.

The pipelined read mode provides the highest data bandwidth for reads and instruction fetches. When a region is programmed for pipelined reads, the address cycle for the next read overlaps the data cycle of the current read.

The bus controller supports two byte orders, big endian and little endian. A little-endian region stores the least significant byte of a data word at the lowest byte address within the word. A big-endian region stores the least significant byte of a data word at the highest byte address within the word. Each memory region is independently configured for either byte order.

BUS TERMINOLOGY: REQUESTS, TRANSFERS, AND ACCESSES

It is necessary to define bus requests, bus transfers and bus accesses as they apply to the 80960CA. These terms are all used to describe the operation of the Bus Controller Unit. (See Figure 10-1.)

Transfer A bus transfer is the movement of code or data between a memory system and the 80960CA. A write transfer occurs when a memory system is the destination of the transfer; a read transfer occurs when a memory system is the source of the transfer; a single access contains one to four transfers.

Access A bus access is an address cycle followed by one or more data transfers. Burst accesses may consist of an address cycle followed by one or more data transfers. Non-burst accesses consist of an address cycle and a single-data transfer. A single request may result in multiple accesses.

Request A bus request is issued by the core or the DMA controller to the Bus Controller. It is generated by a load or store instruction, an instruction fetch, or a DMA-transfer request.

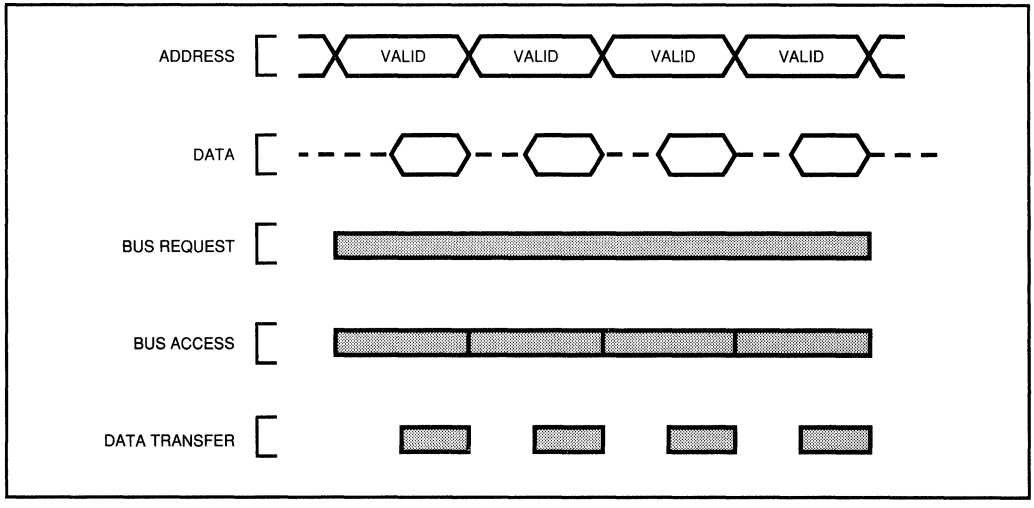


Figure 10-1. Terminology

ACCESSING INTERNAL DATA RAM

One KByte of user-visible internal data RAM is integrated on the 80960CA. This internal data RAM is mapped into the first 1K of the address space (addresses 00H - 3FFH). Data RAM is accessed only by loads, stores, or DMA transfers. Instruction fetches directed to these addresses will cause an operation-unimplemented fault to occur. Some of the 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 the application software. A detailed discussion of the internal data RAM can be found in *Chapter 2, Programming Environment.*

The internal data RAM interfaces directly to an internal 128-bit bus. This bus is the pathway between the registers and the data RAM. Because of the wide internal path, a quad word read or write is usually performed in a single clock.

MEMORY REGION CONFIGURATION

Programmable memory-region configurations simplify the external memory system design and reduce the system parts count. As noted before, the upper four bits of the address (A31:28) indicate which of the 16 regions is currently selected. The memory-region configuration effects all accesses to the addressed memory region. Loads, stores, DMA transfers, and instruction fetches all use the parameters defined in the region table. The memory-region configuration table options are:

- Selectable 8-, 16- , or 32-bit-wide data bus
- Programmable high-performance burst access

- A 5-parameter wait-state generator
- Memory-ready and burst-cycle terminate for dynamic access control
- Programmable pipelined reads
- Selectable big- or little-endian byte ordering

Wait States

The 80960CA bus controller translates a bus request into one or more memory accesses. A memory access consists of an address cycle, one or more data cycles, and the associated wait states. If the burst mode is enabled, an access consists of 1, 2, 3, or 4 data cycles. The number of data cycles depends on the data-bus width and the width of the request (byte, short, word, triple, or quad).

The 80960CA can generate wait states internally. Five parameters for each region define the wait-state-generator operation. Figures 10-2 and 10-3. along with the following text, describe each parameter.

N_{RAD} (**N**umber of wait cycles for **R**ead **A**ddress-to-**D**ata). The number of wait states between the address cycle and the first-read data cycle. N_{RAD} can be programmed for 0-31 wait states.

N_{RDD} (**N**umber 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} (**N**umber 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.

N_{WDD} (**N**umber 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.

N_{XDA} (**N**umber of wait cycles for **X** (read or write) **D**ata to **A**ddress). The number of wait states between the last data cycle of an access to the address cycle of the next access. N_{XDA} applies to both read and write accesses. N_{XDA} is considered the bus turn-around time. The external device's ability to relinquish the bus on a read access (read deasserted to data-float) determines the number of N_{XDA} cycles. DRAM designs can make use of N_{XDA} to guaranteed RAS pre-charge time. N_{XDA} can be programmed for 0-3 clocks.

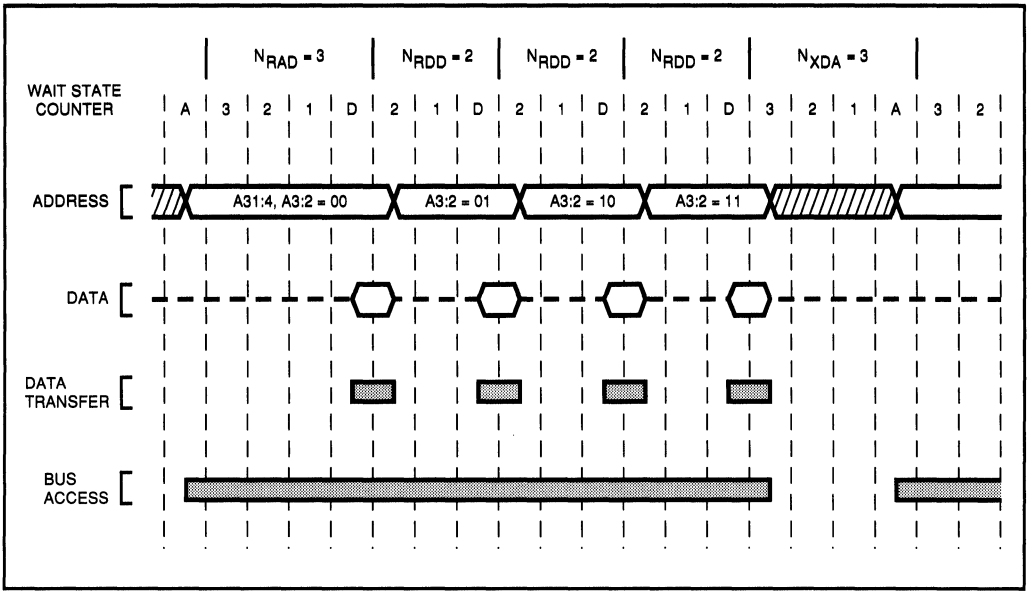


Figure 10-2. Burst Read Wait States

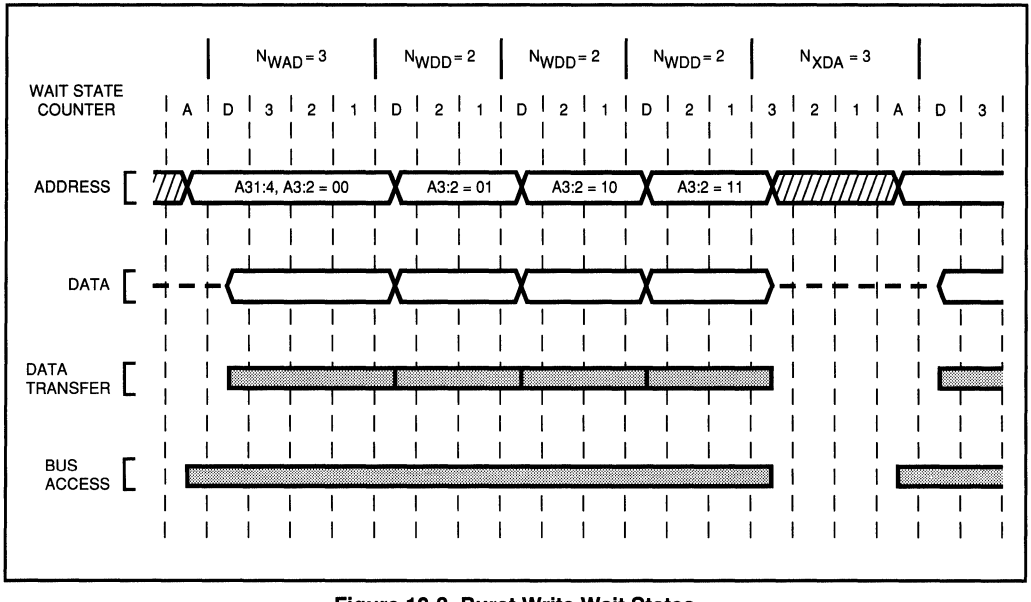


Figure 10-3. Burst Write Wait States

Memory Ready and Burst Terminate Control

The 80960CA provides two control signal inputs that can dynamically alter memory accesses. The ready input (READY) extends accesses by forcing wait states. The burst-terminate input (BTERM) breaks a burst access into a series of non-burst accesses. The memory-region table can be programmed to enable or disable these inputs for each region. These options work with the programmed internal wait states and burst mode. If READY and BTERM are enabled in a region, these pins are sampled only after the programmed number of wait states has expired (Figure 10-4).

The READY input signal indicates to the 80960CA that read data on the bus is valid or that a write data transfer has completed. If the READY pin is asserted (low), wait states continue to be inserted until READY is deasserted (high). This is true for the N_{RAD}, N_{RDD}, N_{WAD}, and N_{WDD} wait states. The N_{XDA} wait states cannot be extended by READY.

The burst-terminate signal (BTERM) breaks up a burst access. Asserting BTERM (low) invokes another address cycle. This allows a burst access to be dynamically broken into smaller accesses which may still be bursts, with no data lost. Read data is accepted on the clock edge that BTERM is asserted, and write data is assumed written. When BTERM is asserted the READY input is ignored, therefore the memory is assumed to be ready.

If BTERM is asserted after the first word of a quad-word burst, the bus controller issues another address cycle. The new address is the address of the second word of the burst access (A3:2 = 01_2). The bus controller then attempts to burst the remaining three words. The BLAST (Burst Last) signal indicates the last data transfer of an access. This signal is asserted at the end of the complete burst access.

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	1 01	1 01	2 00010	OFF 0	1 01	Enabled 1

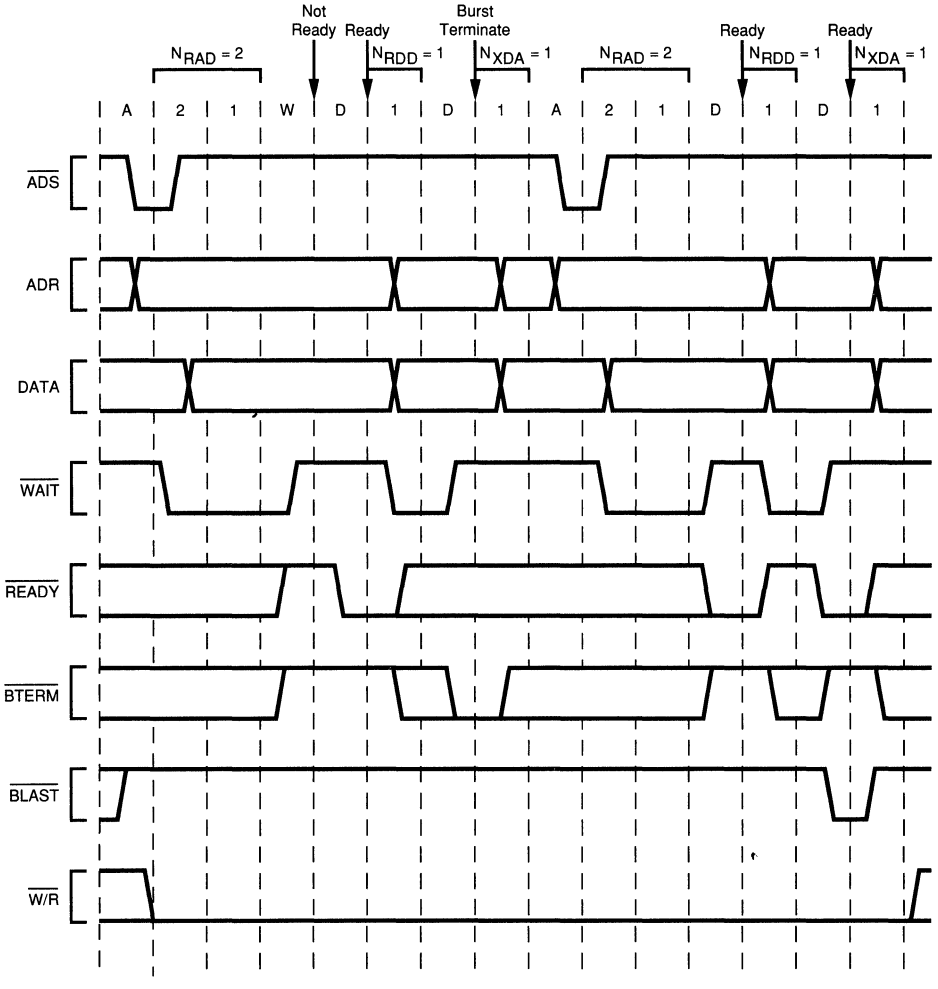


Figure 10-4. BTERM and READY

Data Bus Width

The data-bus width of each region is programmed in the memory region configuration table. The 80960CA allows an 8-, 16-, or 32-bit-wide data bus for each region. The 80960CA justifies the 8- and 16-bit data on the proper data pins. This greatly simplifies the interface to external devices. Eight-bit data is placed on D7:0; 16-bit data is placed on D15:0; and 32-bit data is placed on D31:0 (Figure 10-5).

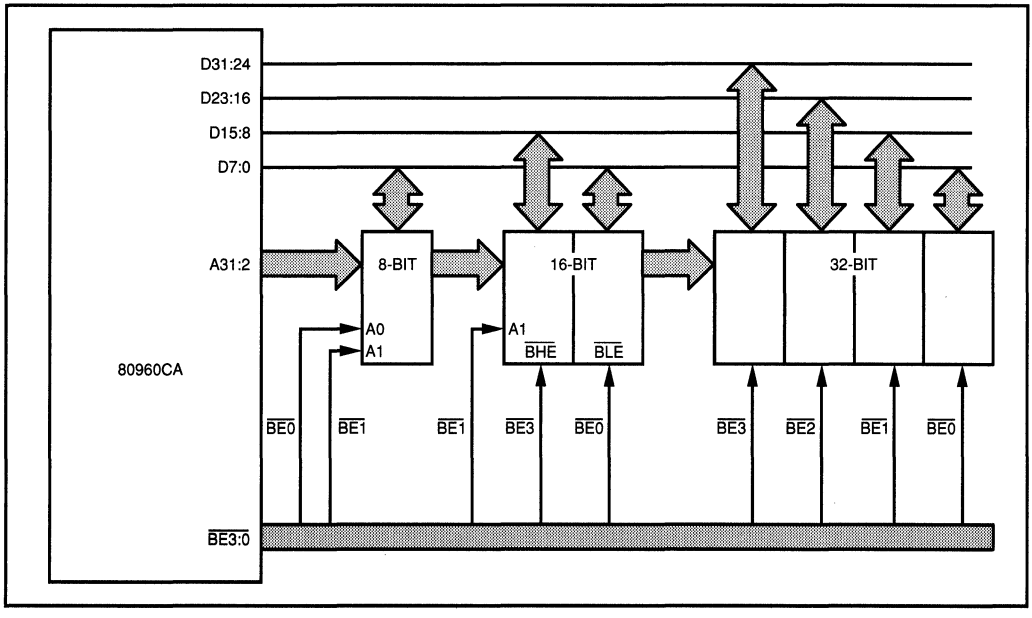


Figure 10-5. Data Width and Byte Enable Encodings

The four byte-enable signals are encoded in each region to generate the proper address signals for 8-, 16-, or 32-bit memory systems. In an 8-bit region, $\overline{BE0}$ is address line A0; and $\overline{BE1}$ is address line A1. In a 16-bit region, $\overline{BE1}$ is address line A1; $\overline{BE3}$ is the byte high-enable signal ($\overline{BHE}$); and $\overline{BE0}$ is the byte low-enable signal ($\overline{BLE}$). In a 32-bit region, the 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 10-1.)

Table 10-1. Byte Enable Encoding

8-Bit Bus Width

BYTE	$\overline{BE3}$ (X)	$\overline{BE2}$ (X)	$\overline{BE1}$ (A1)	$\overline{BE0}$ (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	$\overline{BE3}$ ($\overline{BHE}$)	$\overline{BE2}$ (X)	$\overline{BE1}$ (A1)	$\overline{BE0}$ (A0)
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	$\overline{BE3}$	$\overline{BE2}$	$\overline{BE1}$	$\overline{BE0}$
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

Burst Enable Control

A burst access is an address cycle followed by two to four data transfers. Burst accesses may be enabled or disabled in each region, and they are always aligned. (See the section titled *Data Alignment* later in this chapter.) The two least-significant address signals automatically increment during the burst access. The maximum burst size is four data transfers. 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 burst size of four words. If a quad-word load request (e.g., **ldq**) is made to an 8-bit data region it results in four, four-byte burst accesses. (See Table 10-2.)

Table 10-2. 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

Pipelined Reads

The 80960CA provides an address pipelining mode that improves read-data bandwidth. 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. Pipelining may be enabled or disabled in each memory-region.

If pipelining is enabled, the $\overline{\text{READY}}$ and $\overline{\text{BTERM}}$ inputs are ignored during read accesses to that region. The N_{XDA} wait states are forced to 0 for pipelined reads. $\overline{\text{READY}}$, $\overline{\text{BTERM}}$ and the N_{XDA} wait-states behave as programmed for write accesses. (See *Chapter 11, External Bus Description* for more information about pipelined reads.)

Byte Ordering

The bus controller supports big-endian and little-endian byte ordering for memory operations. Byte ordering determines which memory location contains the least significant byte of an operand. Little-endian systems store the least significant byte of an operand at the lowest byte address in memory. Big-endian systems store the least significant byte at the highest address in memory.

The 80960CA uses little-endian byte ordering internally, and data-in registers follow the little-endian convention. Data-in memory (including the internal data RAM) is arranged as either little or big endian. Both byte ordering arrangements are supported for short-word and word data-types. Any region may be defined to be a big- or little-endian region. Data and instructions can be located in either big- or little-endian regions.

Multiple word bus requests (bursts) to a big-endian region are handled as individual words. The bytes in each word are swapped, but the relationship between the words remains the same. Big-endian data types of greater than 32-bits are not supported.

PROGRAMMING THE BUS CONTROLLER

The Bus Controller is programmed using 17 control registers. Sixteen of these control registers make up the region table; the remaining register is the Bus Configuration (BCON) Register. Control registers are modified by using the load-control-registers message of the system-control (**sysctl**) instruction. (See *Chapter 2, Programming Environment* for a thorough description of control registers.)

The Region Table

The region table contains 16 entries, with each entry stored in a control register. The region table entry specifies the number of wait states; data bus width; byte ordering; burst mode; pipeline mode; byte order; and the external-ready mode for the region that it controls. The four most-significant bits of an address indicate which region is being accessed. A region-table entry is 32-bits wide (Figures 10-6 and 10-7); however, not all of the bits are currently used.

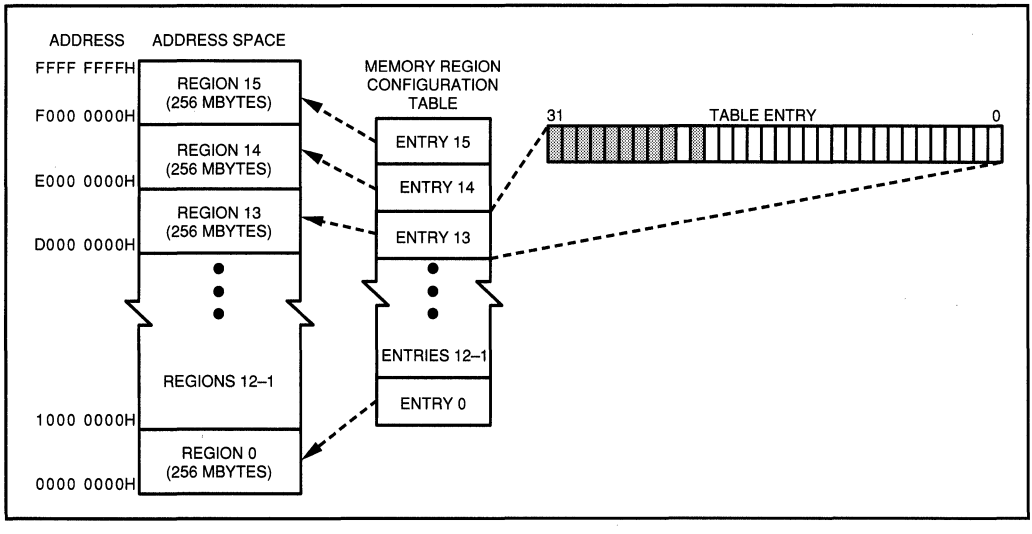


Figure 10-6. Region Table Configures External Memory

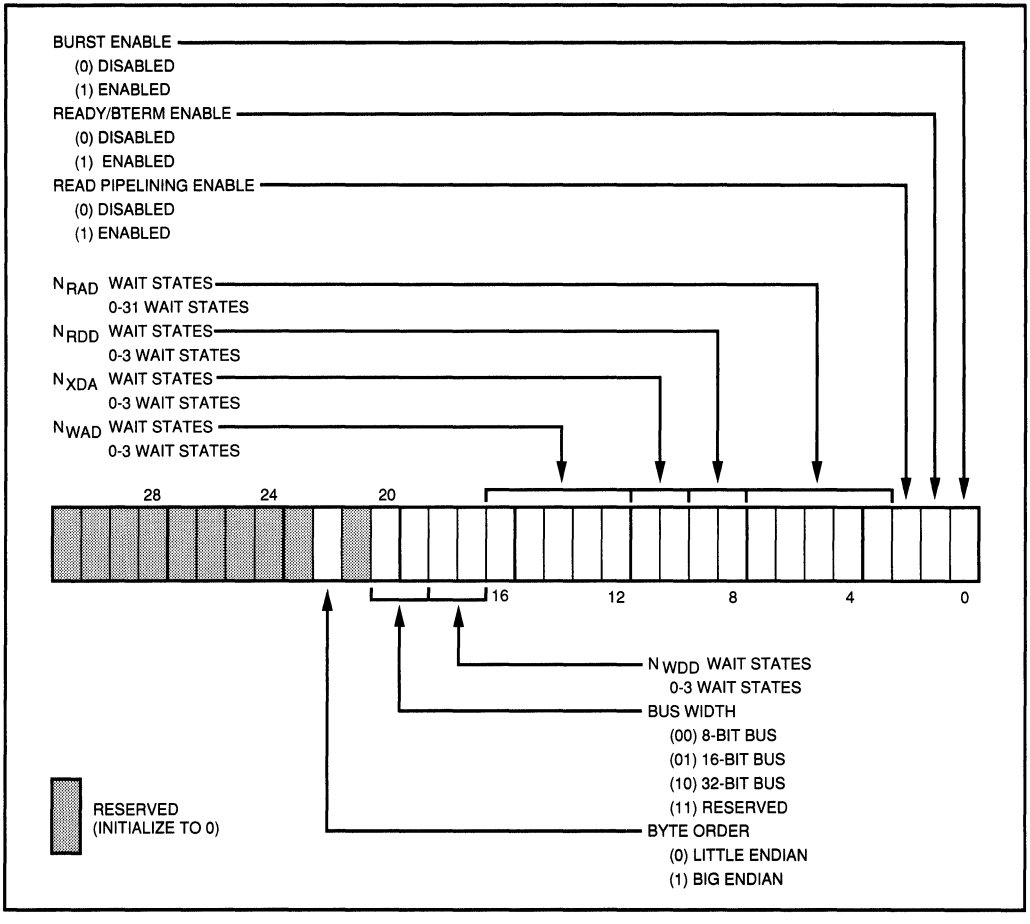


Figure 10-7. Region Table Entry

An entry in the memory-region configuration table is described below and shown in Figure 10-7.

Burst Enable (Bit 0) - Enables or disables burst accesses for the region.

READY/BTERM Enable (Bit 1) - Enables or disables the $\overline{\text{READY}}$ and $\overline{\text{BTERM}}$ inputs for the region. If disabled, the $\overline{\text{READY}}$ and $\overline{\text{BTERM}}$ signals are ignored.

Read Pipelining Enable Bit (Bit 2) - Enables or disables address pipelining of read accesses for the region. $\overline{\text{READY}}$ and $\overline{\text{BTERM}}$ are ignored during pipelined reads.

N_{RAD} Wait States (Bits 3-7) - Number of Read Address-to-Data wait states in the region. (Programmed for 0-31 Wait States)

N_{RDD} Wait States (Bits 8-9) - Number of Read Data-to-Data wait states in the region. (Programmed for 0-3 Wait States)

N_{XDA} Wait States (Bits 10-11) - Number of X (read or write) Data-to-Address wait states in the region. (Programmed for 0-3 Wait States)

N_{WAD} Wait States (Bits 12-16) - Number of Write Address-to-Data wait states in the region. (Programmed for 0-31 Wait States)

N_{WDD} Wait States (Bits 17-18) - Number of Write Data-to-Data wait states in the region. (Programmed for 0-3 Wait States)

Bus Width (Bits 19-20) - Determines the data-bus width for the region. Effects the encoding of the byte-enable signals ($\overline{BE3:0}$).

Byte Ordering (Bit 22) - Selects the byte ordering for the region, little endian or big endian.

The Bus Configuration Register BCON

The Bus Configuration Register (BCON) is a 32-bit register. It controls the region-configuration table and the internal data RAM protection.

The bus configuration register is described below and shown in Figure 10-8.

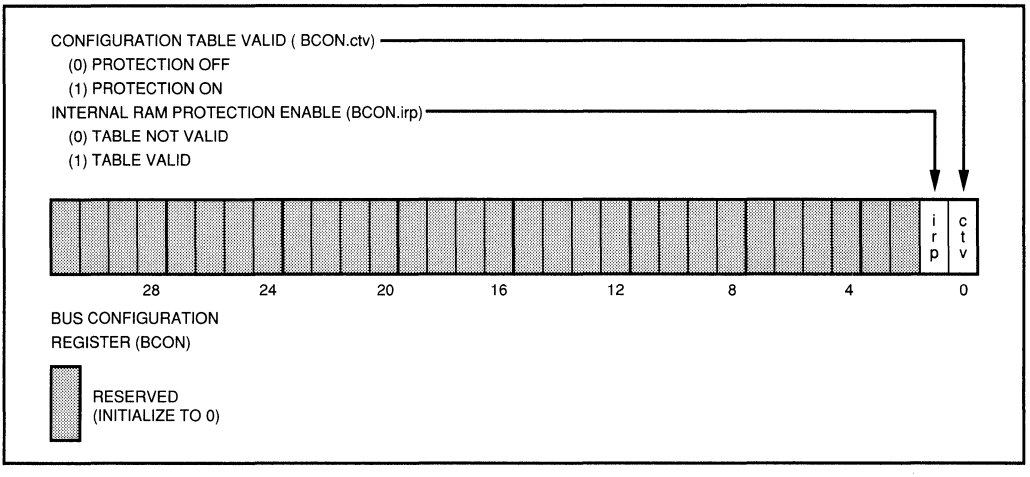


Figure 10-8. BCON Register

Configuration Table Valid (Bit 0) - When the BCON.ctv bit is clear, all memory is accessed as defined by Region Table Entry 0. When the BCON.ctv bit is set, the entire region table is used.

Internal RAM Protection (Bit 1) - Enables supervisor write protection for internal data RAM at address 100H to 3FFH.

10

Configuring the Bus Controller

The bus controller is configured automatically when the processor is initialized. All values for the region table are loaded from the control table, and the BCON.ctv bit is set (table valid) before the first instruction of application code is executed. 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 value in the region table may be altered after initialization by using 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 data transfers are accesses made to data types which fall on a data type's natural boundaries. Quad words and triple words are aligned on 16-byte boundaries; double words are aligned on 8-byte boundaries; words are aligned on 4-byte boundaries; short words are aligned on 2-byte boundaries; and bytes are aligned on 1-byte boundaries.

Unaligned requests to little-endian memory regions are allowed but cause a performance penalty. Unaligned requests can generate a maskable fault. If the fault is masked, the bus controller breaks the non-aligned access into a number of smaller-aligned accesses using microcode. CPU cycles are used to align the unaligned request, and more bus cycles are used to transfer the unaligned data.

If the fault is not masked, an unaligned request will result in an operation-unaligned fault. The alignment fault can be used as a debug feature. Removing unaligned memory accesses from an application increases performance. The unaligned bus-access fault mask is located in the fault-configuration word of the processor control block (PRCB).

If a little-endian word read request from address XXXXXXX0H is issued, a single word-load bus cycle occurs. If a little-endian word read request from address XXXXXXX1H (unaligned) is issued, and the alignment fault is masked, then four single-byte load bus cycles occur. All aligned and unaligned little-endian transfers are summarized in Figure 10-9.

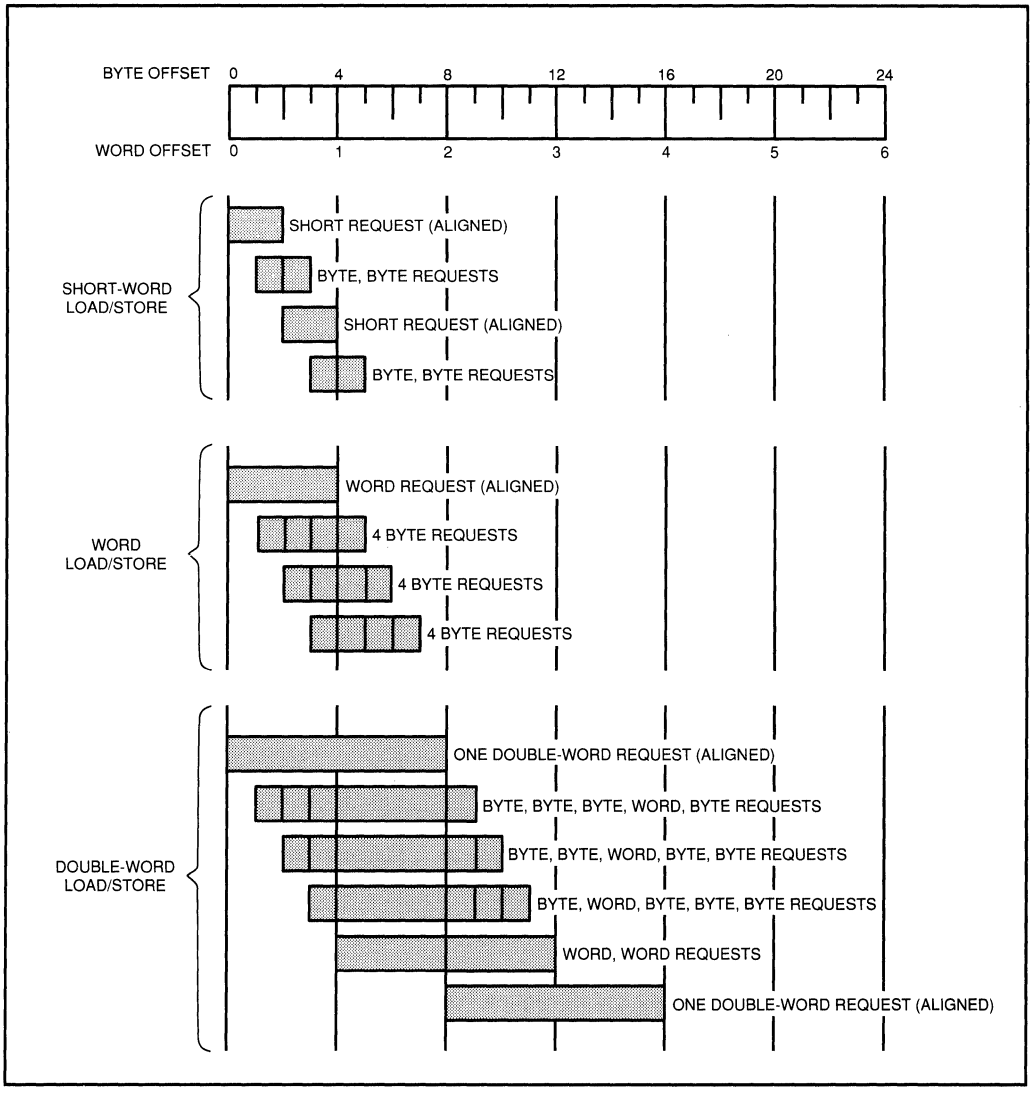


Figure 10-9. A Summary of Aligned and Unaligned Transfers for Little Endian Regions

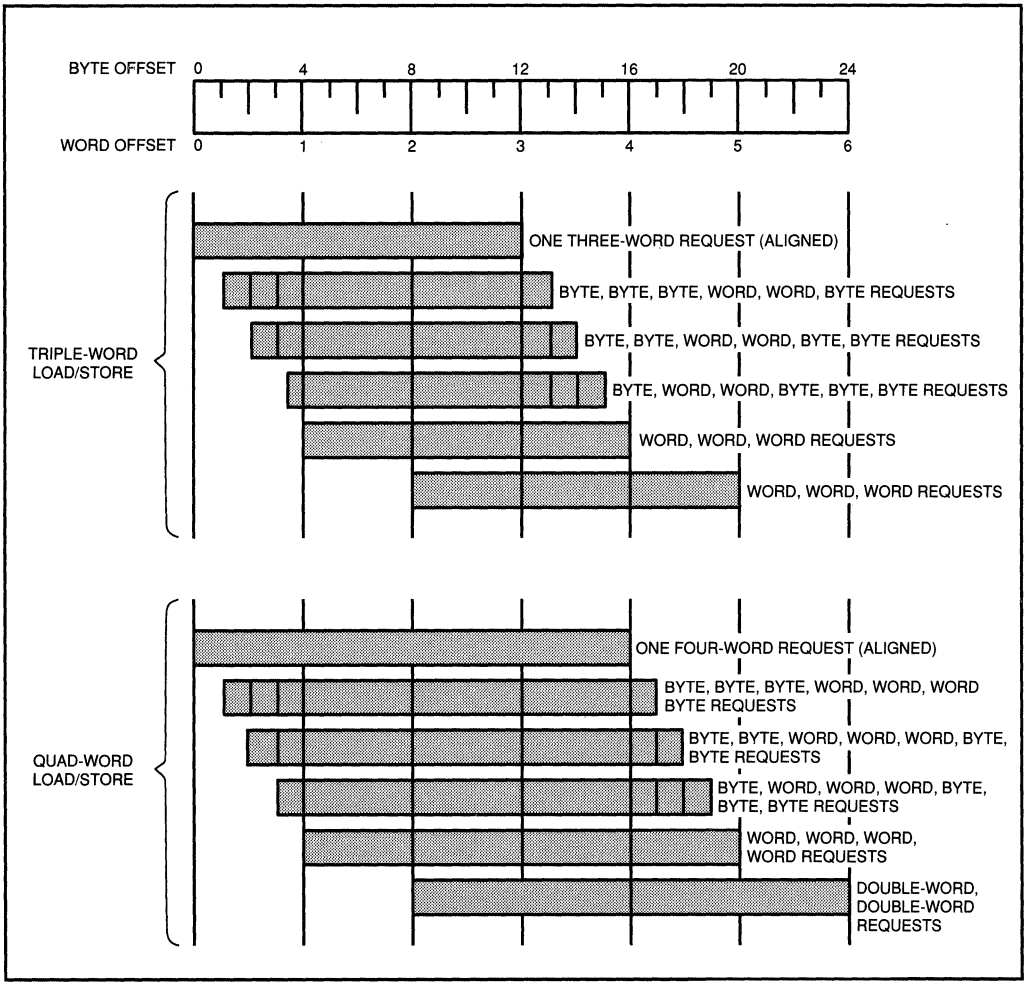


Figure 10-9. A Summary of Aligned and Unaligned Transfers for Little Endian Regions (cont..)

Unaligned Transfers and Big Endian Byte Ordering

Unaligned little-endian requests are either aligned by microcode or cause a fault. Unaligned big-endian requests can only cause a fault. They will not be properly aligned by microcode. If both types of byte ordering are present in a system and microcode is to be used for unaligned little-endian accesses, unaligned accesses to big-endian regions must not be allowed.

BUS CONTROLLER IMPLEMENTATION

The bus controller consists of four units: the queue; the packing unit; the translation unit; and the sequencer (Figure 10-10). The 80960CA's instruction fetch unit, execution unit and DMA unit all pass memory-access requests to the bus controller unit. The bus controller arbitrates, queues and executes these requests.

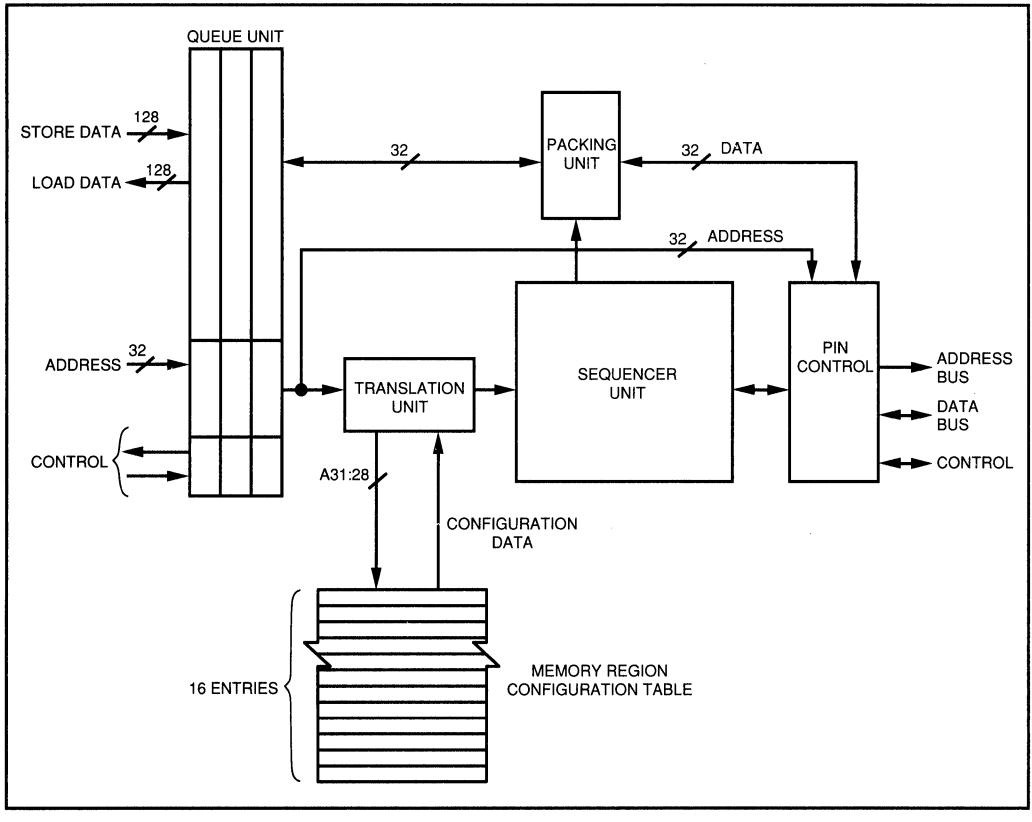


Figure 10-10. Block Diagram of the Bus Controller

The 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 the high bandwidth (128-bit-wide data) internal data busses from the lower bandwidth (32-bit-wide data) external bus.

Two of the queue entries are reserved for bus requests generated from the user's 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 the user program. The 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 process to execute concurrently while sharing the external bus.

The depth of the queue effects the latency of bus requests and the interrupt latency. The queued requests must be serviced before the pending request can be serviced. If an interrupt occurs when all three bus entries are full, the three outstanding requests must be serviced before the first interrupt instruction may be fetched from memory.

The Data Packing Unit

The packing unit handles data movement between the queues and the external bus. The packing unit controls data alignment and data packing. Data is unpacked when the width of data-store request exceeds the width of the physical bus; data is packed when the width of data-load request exceeds the width of the physical bus.

If a word load is issued to an 8-bit bus, the bus controller issues four 1-byte reads, and the packing unit assembles the 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 packing unit unpacks the outgoing data.

The 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 address of the bus request. The bus request and the data from the region table are passed to the bus sequencer when the external bus is available. The sequencer then breaks the request into a set of bus accesses, generating the signals on the external bus pins.

External Bus Description 11

CHAPTER 11
EXTERNAL BUS DESCRIPTION

This chapter discusses the 80960CA bus pins, bus transactions, and bus arbitration. It also describes waveforms and the effects of the programmable memory region configuration table.

OVERVIEW

The 80960CA's integrated bus controller and external bus provide a flexible, easy-to-use interface to memory and peripherals. All of the bus transactions are synchronized with the processor clock output (PCLK); 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 the system component count and ease the design task.

PIN DESCRIPTIONS

The 80960CA bus signals consist of 30 address signals, four byte enables, 32 data lines, and various control signals. A description of the bus signals is given below. All bus signals are TTL compatible.

PCLK2, PCLK1 Processor Output Clocks (outputs) - PCLK1 and PCLK2 are identical clock outputs for the processor's synchronous bus. All other bus operations are synchronized to PCLK1 and PCLK2. Two output pins are provided for additional drive capability, PCLK is one-half the frequency of CLKIN (the input clock).

D31:D0 32-Bit Data Bus (input/output) - 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 floats during HOLDA, ONCE, and read operations.

A31:2 30-Bit Address (outputs) - The 30-bit address bus identifies all external addresses to word (4-Byte) boundaries. The byte-enable lines indicate the selected byte in each word. A3:2 increment during 32-bit burst accesses. The address bus floats during HOLDA and ONCE operations.

$\overline{BE3:0}$ — Byte Enable (outputs) - The byte enables 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, $\overline{BE0}$ and $\overline{BE1}$ act as the lower two bits of the address (A1:0). For a 16-bit memory region, $\overline{BE3}$, $\overline{BE1}$, and $\overline{BE0}$ are encoded as $\overline{BHE}$, A1, and $\overline{BLE}$ respectively. The byte enables are put into a high-impedance state during HOLDA and ONCE operations.

W/$\overline{R}$ — Write/Read (output) - This signal is low for read accesses and high for write accesses. The W/$\overline{R}$ signal changes in the same clock cycle as $\overline{ADS}$ and it remains valid for the entire access in non-pipelined regions. In pipelined regions, W/$\overline{R}$ is not valid in the last cycle of a read access. W/$\overline{R}$ is put into a high-impedance state during HOLDA and ONCE operations.

$\overline{ADS}$ — Address Strobe (output) - This signal indicates valid address and the start of a new bus access. $\overline{ADS}$ is asserted for the first clock of a bus access. $\overline{ADS}$ is put into a high-impedance state for HOLDA and ONCE operations.

DT/$\overline{R}$ — Data Transmit/Receive (output) - DT/$\overline{R}$ is used for direction control for data transceivers. DT/$\overline{R}$ is low when the 80960CA is reading data, and high when it is writing data. DT/$\overline{R}$ is guaranteed not to change while DEN is asserted. DT/$\overline{R}$ is put into a high impedance state for HOLDA and ONCE operations.

$\overline{DEN}$ — Data Enable (output) - $\overline{DEN}$ is asserted (low) after the first address cycle of a bus access and it is deasserted at the end of the last data-cycle of the access (before the N_{XDA} cycles). $\overline{DEN}$ is used to control external data transceivers. $\overline{DEN}$ will remain asserted for sequential accesses to pipelined regions. $\overline{DEN}$ is is put into a high impedance state during HOLDA and ONCE operations.

$\overline{WAIT}$ — Wait (output) - This signal indicates that during a bus access, the internal wait-state generator is inserting either N_{RAD}, N_{RDD}, N_{WAD}, or N_{WDD} wait states. $\overline{WAIT}$ is not asserted during N_{XDA} wait states. WAIT is put into a high-impedance state during HOLDA and ONCE.

$\overline{\text{READY}}$

Ready (input) - The $\overline{\text{READY}}$ input signal indicates that read data on the bus is valid, or that a write-data transfer has completed. The $\overline{\text{READY}}$ signal works in conjunction with the internally programmed wait-state generator. If $\overline{\text{READY}}$ and $\overline{\text{BTERM}}$ are enabled in a region, the $\overline{\text{READY}}$ pin is sampled after the programmed number of wait states has expired. If the $\overline{\text{BTERM}}$ pin is not asserted (high), and the $\overline{\text{READY}}$ pin is asserted (low), wait states will continue to be inserted until $\overline{\text{READY}}$ is deasserted (high). This is true for the N_{RAD}, N_{RDD}, N_{WAD}, and N_{WDD} wait-states. The N_{XDA} wait states cannot be extended by $\overline{\text{READY}}$. To satisfy the $\overline{\text{READY}}$ setup and hold times found in the data sheet, $\overline{\text{READY}}$ must be externally synchronized.

$\overline{\text{BTERM}}$

Burst Terminate (input) - The burst-terminate signal breaks-up a burst access and causes another address cycle to occur. The $\overline{\text{BTERM}}$ signal works in conjunction with the internally programmed wait-state generator. If $\overline{\text{READY}}$ and $\overline{\text{BTERM}}$ are enabled in a region, the $\overline{\text{BTERM}}$ pin is sampled after the programmed number of wait states has expired. When $\overline{\text{BTERM}}$ is asserted, $\overline{\text{READY}}$ is ignored. $\overline{\text{BTERM}}$ must be externally synchronized to satisfy the $\overline{\text{BTERM}}$ setup and hold times found in the data sheet.

D/$\overline{\text{C}}$

Data/Code (output) - D/$\overline{\text{C}}$ indicates that the current access is a data transfer or a code fetch. It has the same timing as W/$\overline{\text{R}}$. D/$\overline{\text{C}}$ is put into a high-impedance state during HOLDA and ONCE operations.

$\overline{\text{DMA}}$

CPU/DMA (output) - This signal indicates that the current bus access was requested by the DMA. It has the same timing as W/$\overline{\text{R}}$ $\overline{\text{DMA}}$ is put into a high-impedance state during HOLDA and ONCE operations.

$\overline{\text{SUP}}$

User/Supervisor (output) - $\overline{\text{SUP}}$ indicates that the current bus access was requested by a supervisor-mode process. It has the same timing as W/$\overline{\text{R}}$. $\overline{\text{SUP}}$ is put into a high-impedance state during HOLDA and ONCE operations.

$\overline{\text{BLAST}}$

Burst Last (output) - $\overline{\text{BLAST}}$ indicates that the data transfers of an access are complete. This signal is asserted in the last data transfer of every bus access, burst and non-burst. $\overline{\text{BLAST}}$ is put into a high-impedance state during HOLDA and ONCE operations.

HOLD Hold (input) - HOLD is used by an external bus master to request access to the bus. The processor asserts HOLDA and relinquishes the bus after the current bus request has completed. 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 *80960CA Data Sheet.*

HOLDA Hold Acknowledge (output) - HOLDA indicates to a bus requester that the processor has relinquished control of the bus. 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.

$\overline{\text{LOCK}}$ Lock (output) - This signal indicates that an atomic memory operation (**atadd**, **atmod**) is in progress. Atomic memory operations are read-modify-write operations. The $\overline{\text{LOCK}}$ signal 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. The $\overline{\text{LOCK}}$ signal is deasserted after the write portion of an atomic access. A HOLD request will be 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.

BREQ Bus Request (output) - While HOLDA is asserted, BREQ indicates that the 80960CA bus controller wishes to perform an external memory operation. BREQ can be used with external bus-arbitration logic to regain control of the bus. It is put into a high-impedance state during ONCE operation.

$\overline{\text{ONCE}}$ On Circuit Emulation (input) - This signal is pulled-up internally and should be left unconnected by the user for normal operation. If $\overline{\text{ONCE}}$ is asserted (low) while $\overline{\text{RESET}}$ is asserted (low), all the output pins float, and all the 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.

Access-Request Status Signals

The D/$\overline{C}$, $\overline{DMA}$, and $\overline{SUP}$ signals provide information about the source of an access request. D/$\overline{C}$ indicates that the current access is data or code. The $\overline{DMA}$ signal indicates that the current access is a DMA access. The $\overline{SUP}$ signal indicates that the current access was requested by a supervisor mode process. When used with a logic analyzer, these signals aid in software debugging.

The D/$\overline{C}$ signal can also be used to implement separate external data and instruction caches, while the $\overline{SUP}$ pin can be used to protect hardware from accesses while the processor is not in the supervisor mode.

BUS TRANSACTIONS

All bus accesses begin by asserting $\overline{ADS}$ and end by asserting $\overline{BLAST}$. The bus is synchronous with PCLK, so all address, data, and control signals (with the exception of DT/$\overline{R}$) are qualified by the rising edge of PCLK.

In between bus accesses, the bus is in the idle state. The idle bus state begins one cycle after the $\overline{BLAST}$ signal is asserted, and ends when $\overline{ADS}$ is asserted. The $\overline{ADS}$ and $\overline{LOCK}$ signals are guaranteed to remain deasserted during the idle state. The following bus signal are invalid during idle: D31:0; A31:2; $\overline{BE3:0}$; $\overline{BLAST}$; W/$\overline{R}$; $\overline{WAIT}$; DT/$\overline{R}$; $\overline{DEN}$; $\overline{DMA}$; $\overline{SUP}$; and BREQ. HOLD/HOLDA bus arbitration is independent of the idle state and will function properly.

All accesses on the bus are aligned by the bus controller. Non-aligned accesses are translated into a series of smaller-aligned accesses by the bus controller. A detailed description of alignment can be found in *Chapter 10, Bus Controller.*

Non-Burst Accesses

A basic (non-burst, non-pipelined) access (Figure 11-1) is an address cycle followed by a single data-cycle, including any optional wait states associated with the access. The wait states may be generated internally by the wait-state generator or externally, using the 80960CA's $\overline{\text{READY}}$ input.

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.

Basic accesses are controlled by the N_{RAD}, N_{WAD} and N_{XDA} wait-state fields of a region table entry. N_{RAD} specifies the number of wait states between the address cycle and the data cycle for read accesses. N_{WAD} specifies the number of wait states between the address cycle and the data cycle for write accesses. N_{XDA} specifies the number of wait states between the data cycle and the next address cycle. Data-to-data wait states (N_{RDD}, N_{WDD}) are not used if burst accesses are not enabled.

The read access begins by asserting the proper address and status signals ($\overline{\text{ADS}}$ A31:2; $\overline{\text{BE}}$0:3; $\overline{\text{SUP}}$; D/$\overline{\text{C}}$; $\overline{\text{DMA}}$; W/$\overline{\text{R}}$) on the rising edge that begins the address cycle (marked as "A" on the figures). The assertion of $\overline{\text{ADS}}$ indicates the beginning of an access.

DT/$\overline{\text{R}}$ is driven on the next falling edge of the clock. 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 next rising edge of the clock (the rising edge that $\overline{\text{ADS}}$ is deasserted and the address cycle ends). The $\overline{\text{DEN}}$ signal can be used to control external data transceivers.

The cycles that follow are the N_{RAD} wait states. The $\overline{\text{WAIT}}$ signal is asserted while the internal wait-state generator is counting. If $\overline{\text{READY}}$/$\overline{\text{BTERM}}$ control is enabled in this region, and $\overline{\text{READY}}$ is not asserted after the wait-state generator has finished counting, wait states will continue to be inserted until the $\overline{\text{READY}}$ signal is asserted.

The assertion of $\overline{\text{BLAST}}$ indicates the end of the data-transfer cycles for this access. The $\overline{\text{DEN}}$ signal is deasserted.

The N_{XDA} wait states (turn-around wait states) follow $\overline{\text{BLAST}}$, and a new address cycle may start after the N_{XDA} cycles have expired. The N_{XDA} states allow slow devices time to get off the bus.

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	X xx	X xx	X xxxxx	1 01	X xx	3 00011	Off 0	Disabled 0	Disabled 0

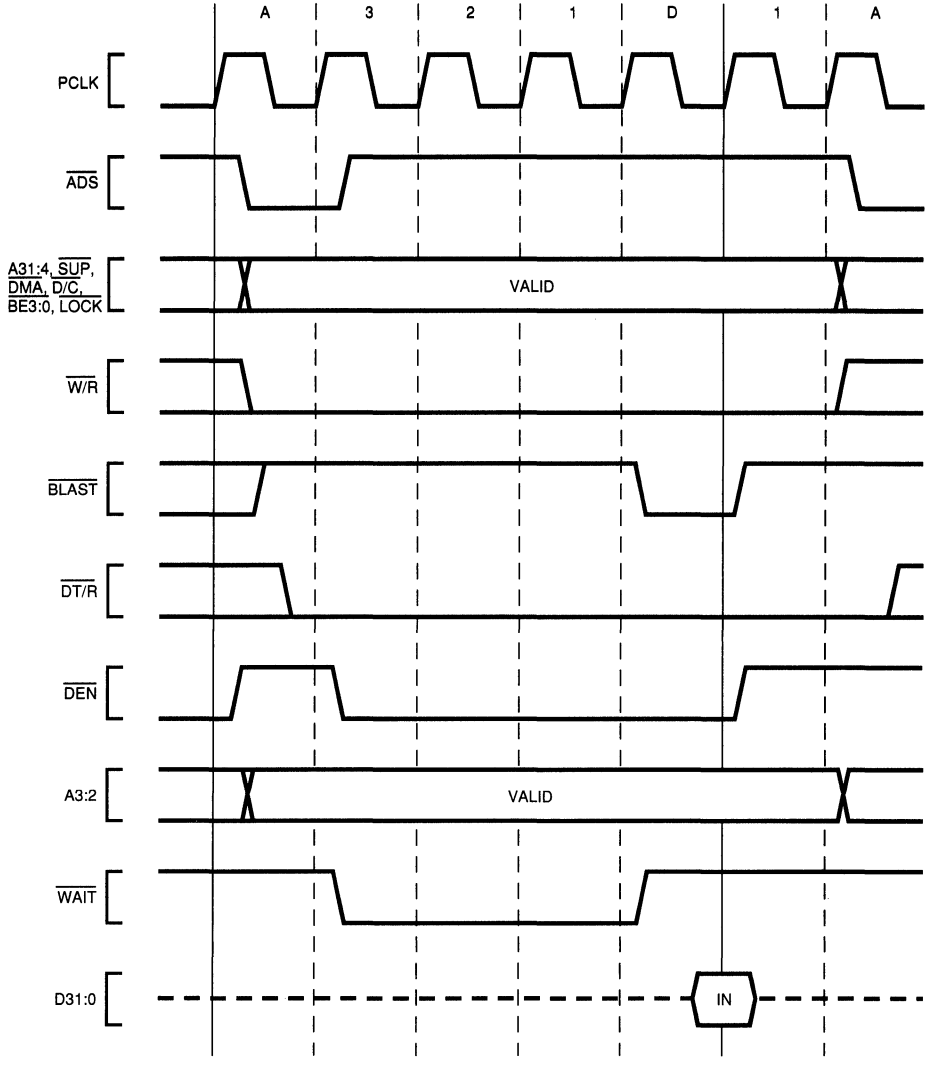


Figure 11-1. Basic Read Access, Non-Pipelined, Non-Burst, Wait-States

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 0...0	X x	0 0	X xx	X xx	0 00000	0 00	X xx	0 00000	Off 0	Disabled 0	Disabled 0

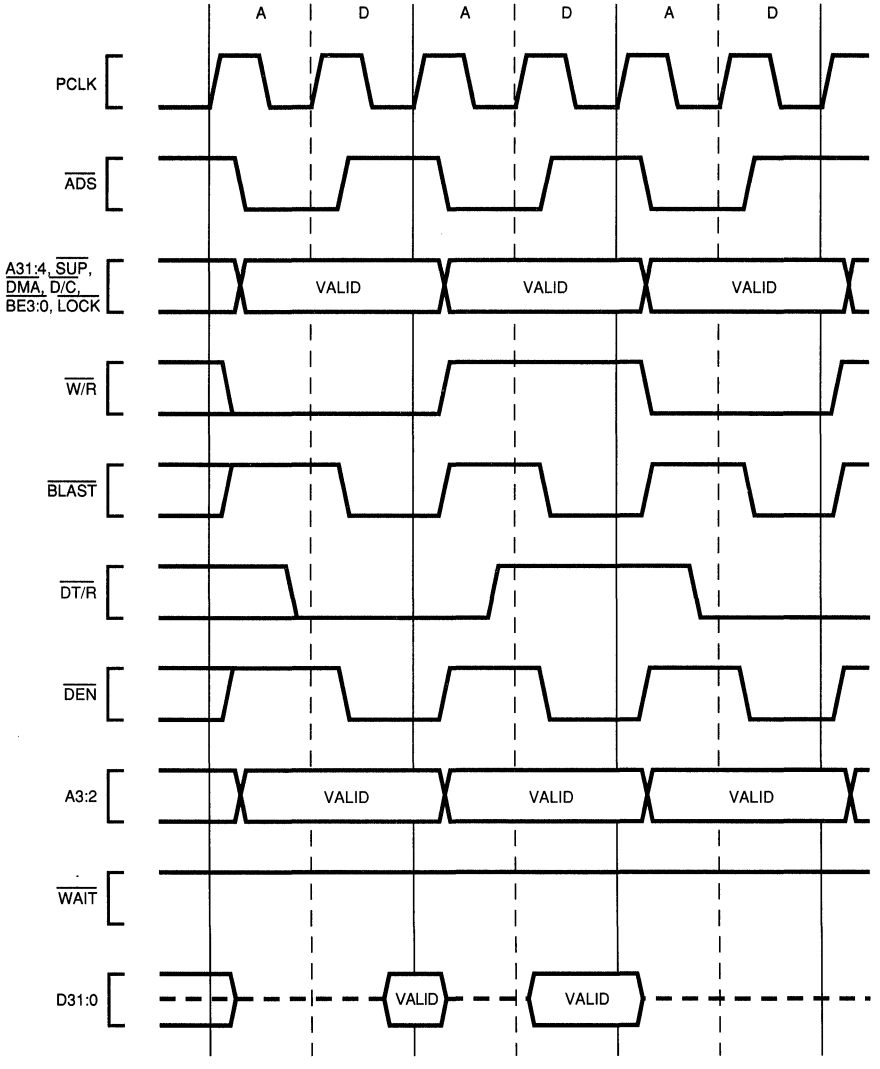


Figure 11-2. Basic Read and Write Accesses, Non-Pipelined, Non-Burst, No Wait States

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 0...0	X x	0 0	X xx	X xx	3 00011	1 01	X xx	X xxxxx	Off 0	Disabled 0	Disabled 0

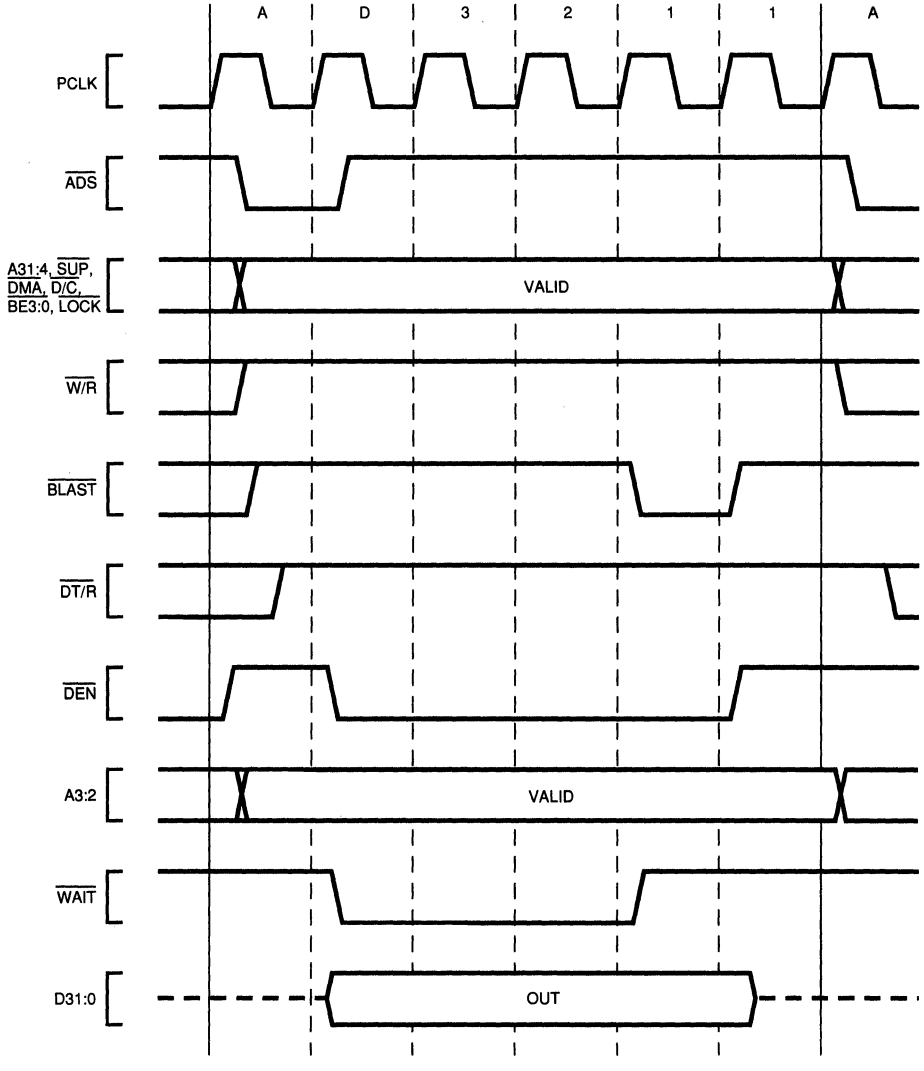


Figure 11-3. Basic Write Access, Non-Pipelined, Non-Burst, Wait States

Burst Accesses

Burst accesses increase the bus bandwidth over non-burst accesses. The 80960CA burst access allows up to four consecutive data cycles to follow a single address cycle. Burst-mode memory systems can get greater performance out of slower memory than non-burst mode memory. SRAM, interleaved SRAM, Static-Column-Mode DRAM, and Fast-Page-Mode DRAM may be easily designed into burst-mode memory systems.

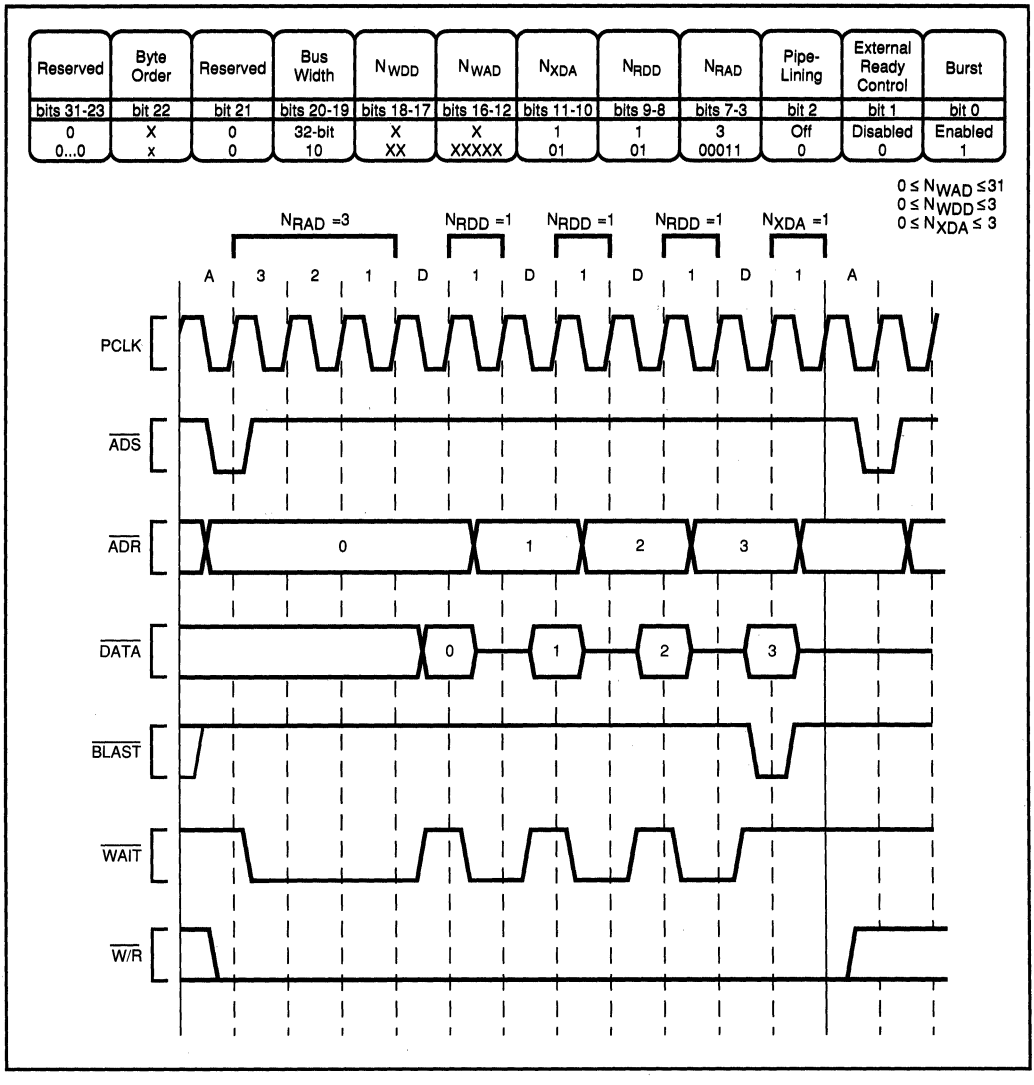


Figure 11-4. Burst-Read Access, Non-Pipelined, with Wait States

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}); one to four data cycles, separated by zero to three data-to-data wait states (N_{RDD} or N_{WDD}); and zero to three turn-around wait states (N_{XDA}). If $\overline{READY}/\overline{BTERM}$ control is enabled in the region, the N_{RAD}, N_{WAD}, N_{RDD} and N_{WDD} wait states may all be extended by not asserting the $\overline{READY}$ signal. $\overline{BTERM}$ may be used to break a burst access into smaller accesses.

The two least-significant bits of the address automatically increment for each consecutive burst data access. 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, $\overline{BE1}$ is encoded as A1, and address pins A2 and A1 increment. When a memory region is configured for an 8-bit data bus width, $\overline{BE0}$ and $\overline{BE1}$ acting as the lower two bits of the address increment. The maximum burst size is four data transfers per access.

Multiple data-cycle 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); and single-word transfers occur on single word boundaries. (See Figure 11-5.)

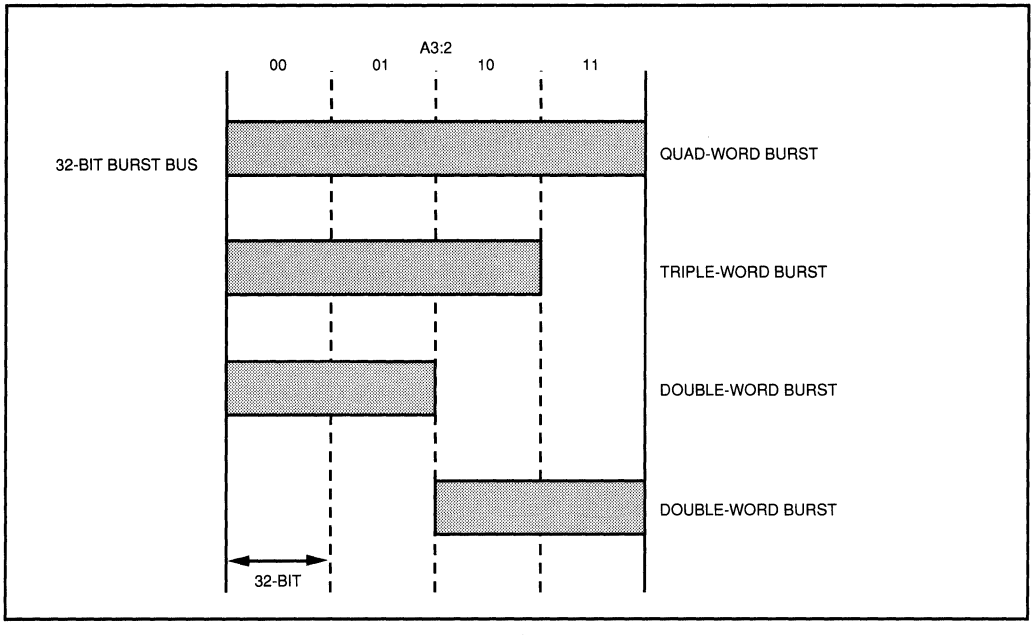


Figure 11-5. 32-Bit-Wide Data-Bus Bursts

Multiple data-cycle burst accesses for a 16-bit bus are always aligned to even short-word boundaries. Data is only transferred on data pins D15:0 for a 16-bit bus. 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 11-6.)

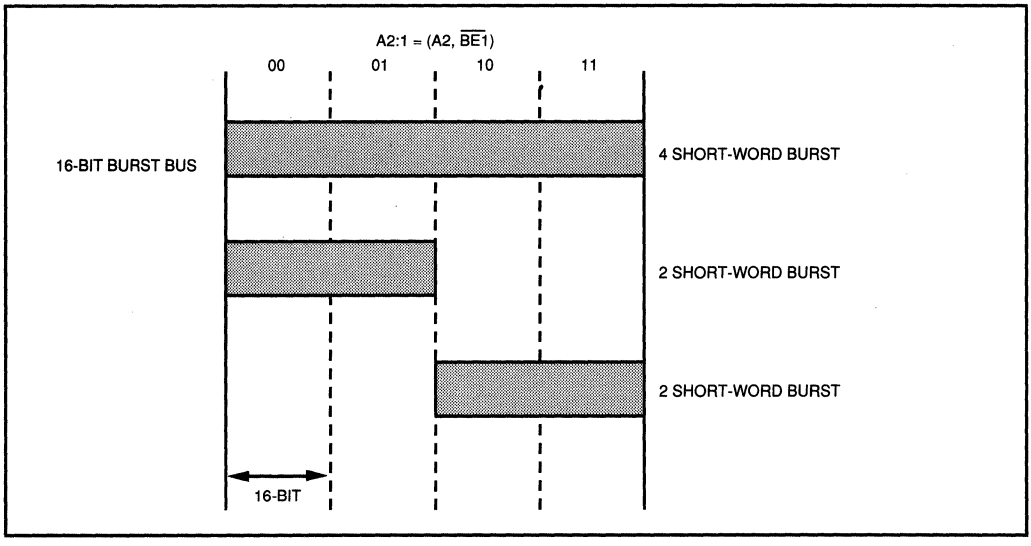


Figure 11-6. 16-Bit Wide Data-Bus Bursts

Multiple data-cycle burst accesses for an 8-bit bus are always aligned to even byte boundaries. Data is only transferred on data pins D7:0 for an 8-bit bus. 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 11-7.)

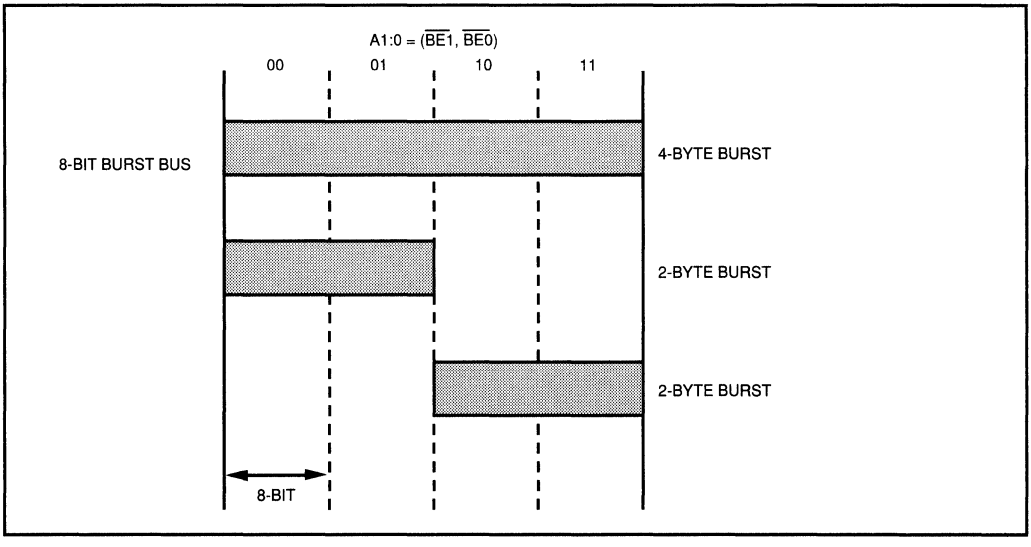


Figure 11-7. 8-Bit Wide Data-Bus Bursts

Figure 11-8 shows a quad-word read on a 32-bit bus. The 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). The word read asserts all the byte enable signals ($\overline{BE3:0}$). The assertion of $\overline{ADS}$ indicates the beginning of an access.

DT/$\overline{R}$ is driven on the next falling edge of the clock to ensure that DT/$\overline{R}$ does not change while $\overline{DEN}$ is asserted.

$\overline{DEN}$ is asserted on the next rising edge of the clock (the rising edge that ends the address cycle). $\overline{ADS}$ is deasserted on this clock edge. The $\overline{DEN}$ signal is used to control external data transceivers. $\overline{DEN}$ and DT/$\overline{R}$ remain asserted throughout the burst access.

The wait-state cycles that follow are the address and the N_{RAD} wait states. The $\overline{WAIT}$ signal 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 will continue to be inserted until the $\overline{READY}$ signal is asserted. If $\overline{BTERM}$ is asserted, then $\overline{READY}$ is ignored. The data is then read, and a new address cycle is generated. (See the section title *Ready and Burst Terminate Control* later in this chapter.)

The data cycle is followed by the N_{RDD} wait states. These wait states separate burst data cycles. They can be used to extend the data access time of reads, and the data setup and hold times, for writes.

The assertion of $\overline{BLAST}$ indicates the end of the data-transfer cycles for this access. At this time, the $\overline{DEN}$ signal is deasserted.

The N_{XDA} wait states (turn-around wait states) follow $\overline{BLAST}$. A new address cycle may start after the N_{XDA} cycles have expired. The N_{XDA} states allow slow devices to get off the bus.

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	1 01	1 01	2 00010	Off 0	Disabled 0	Enabled 1

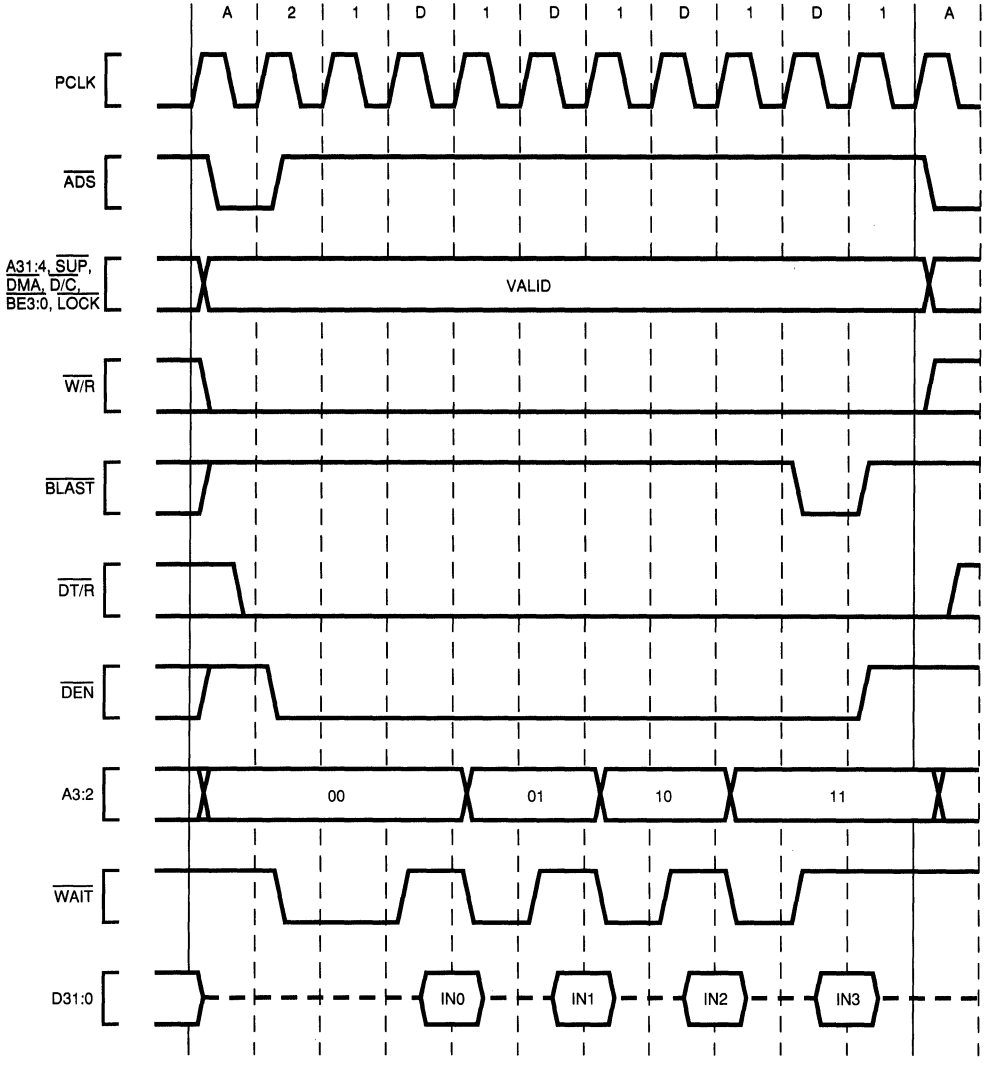


Figure 11-8. Detailed Waveforms 32-bit bus, Burst, Non-Pipelined, Read with wait states

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 0...0	X x	0 0	32-bit 10	1 01	3 00011	1 01	X xx	X xxxxx	Off 0	Disabled 0	Enabled 1

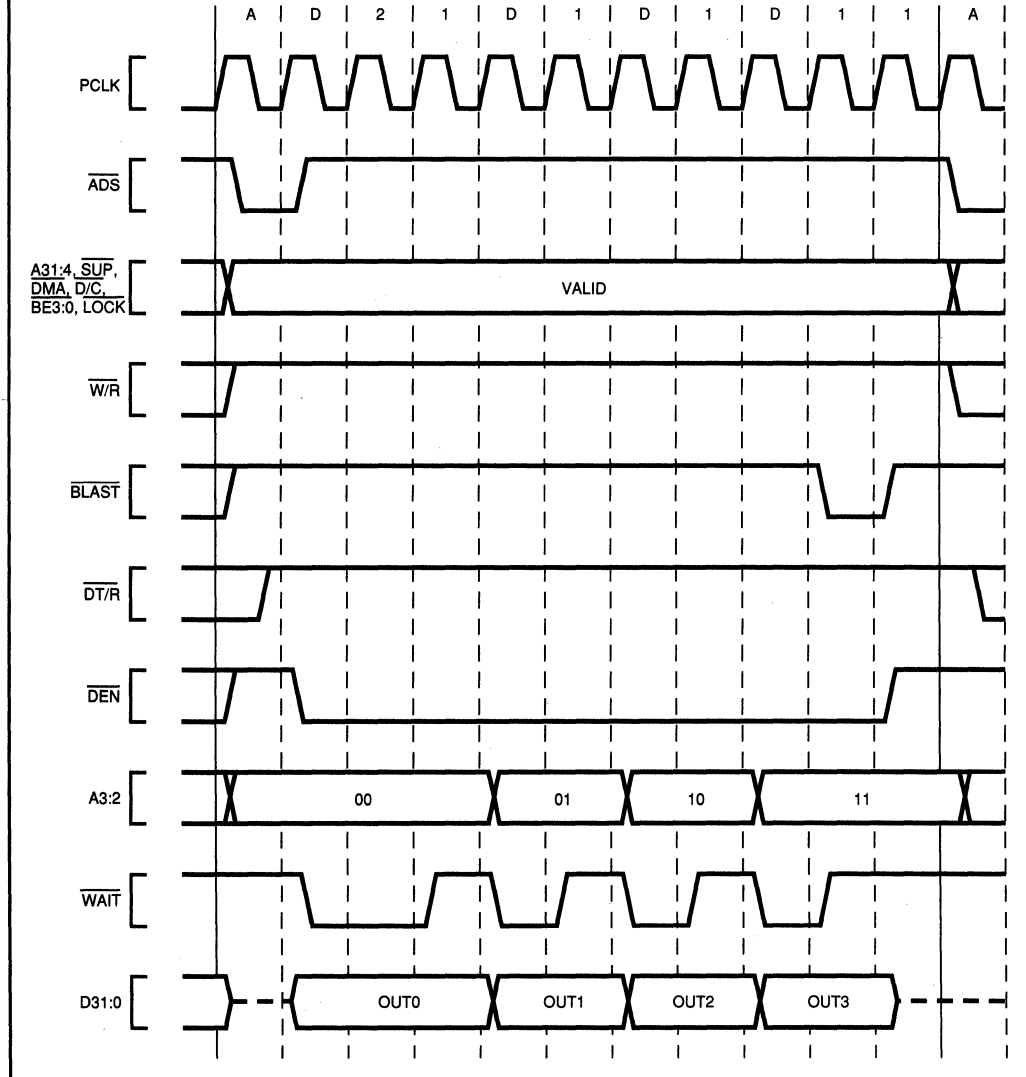


Figure. 11-9. Detailed Waveforms 32-Bit Burst, Non-Pipelined, Write with wait states

Pipelined Reads

Pipelined-read accesses provide 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 11-11.) 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 the address and data cycles improves the data bandwidth .

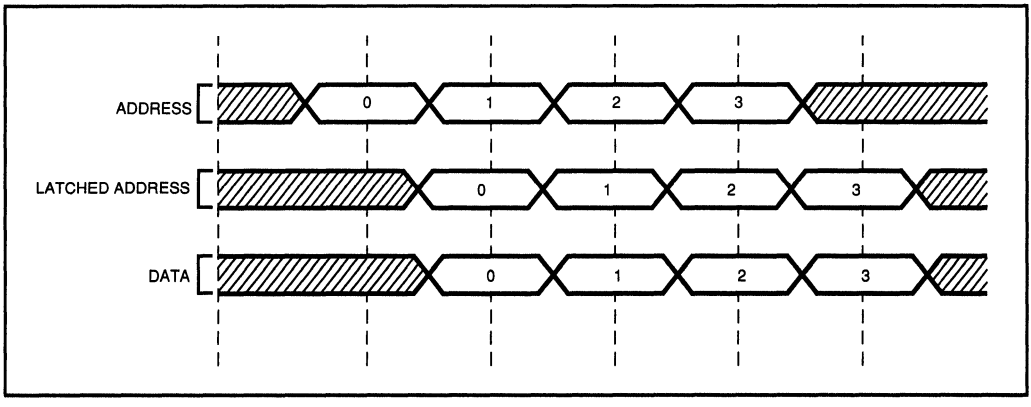


Figure. 11-10. Simple Pipelined-Read Waveform

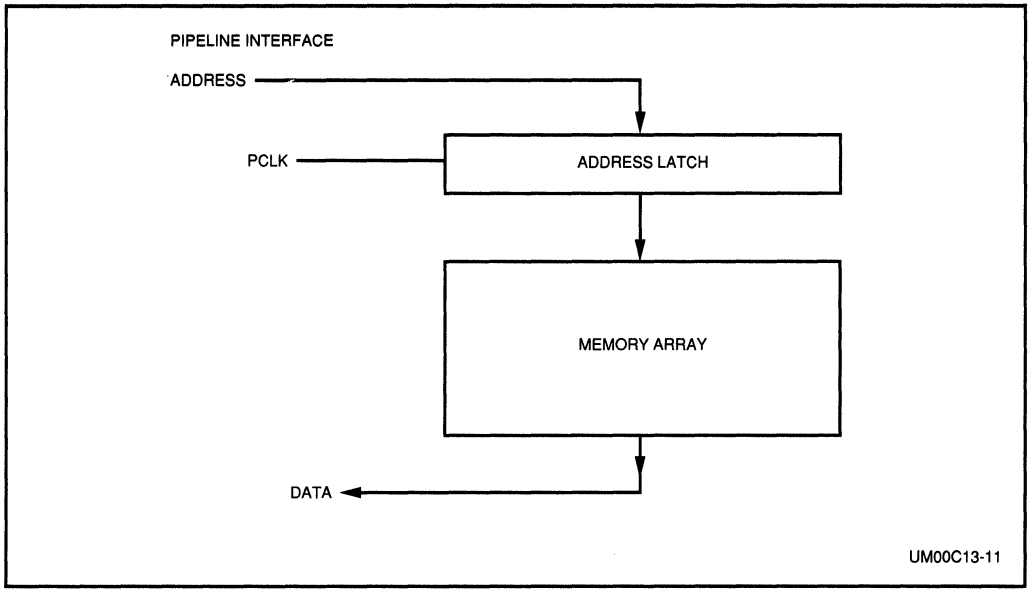


Figure 11-11. Pipelined-Read Memory System

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 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

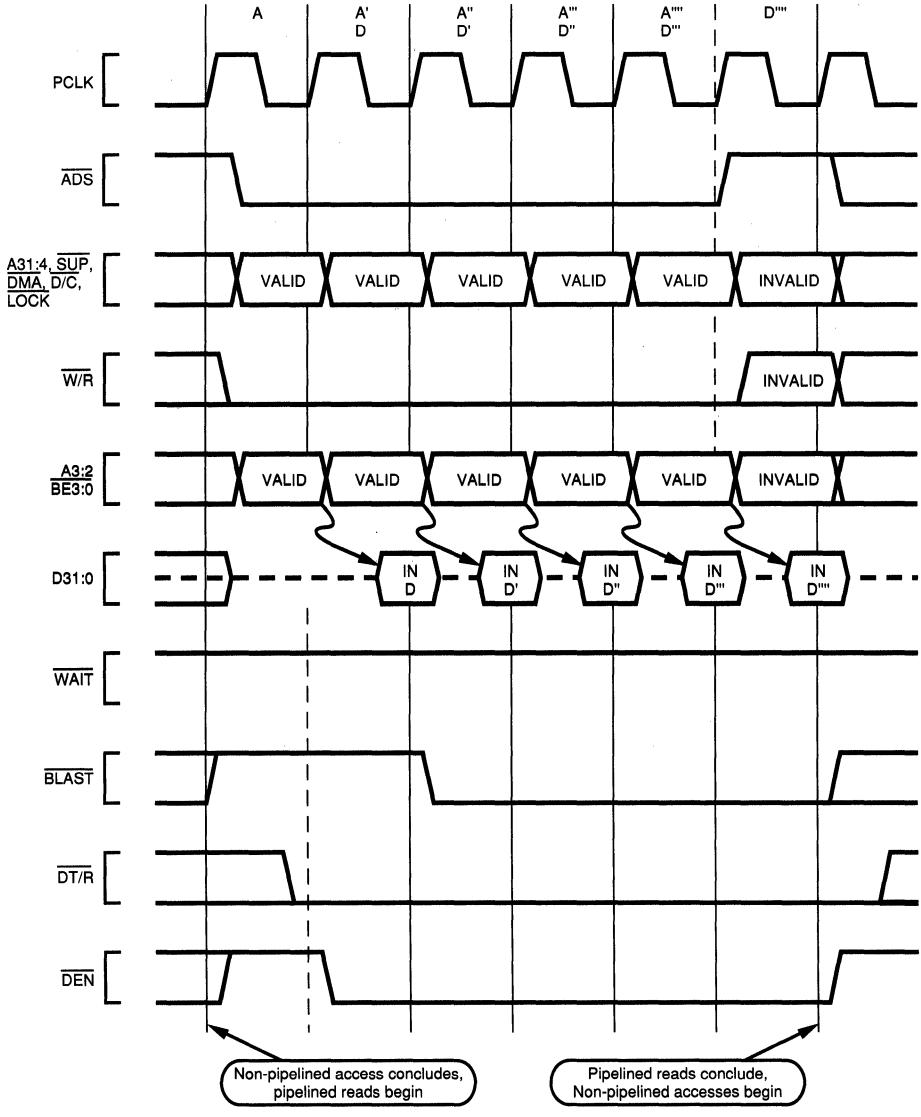


Figure 11-12. Non-Burst Pipeline-Read Waveform

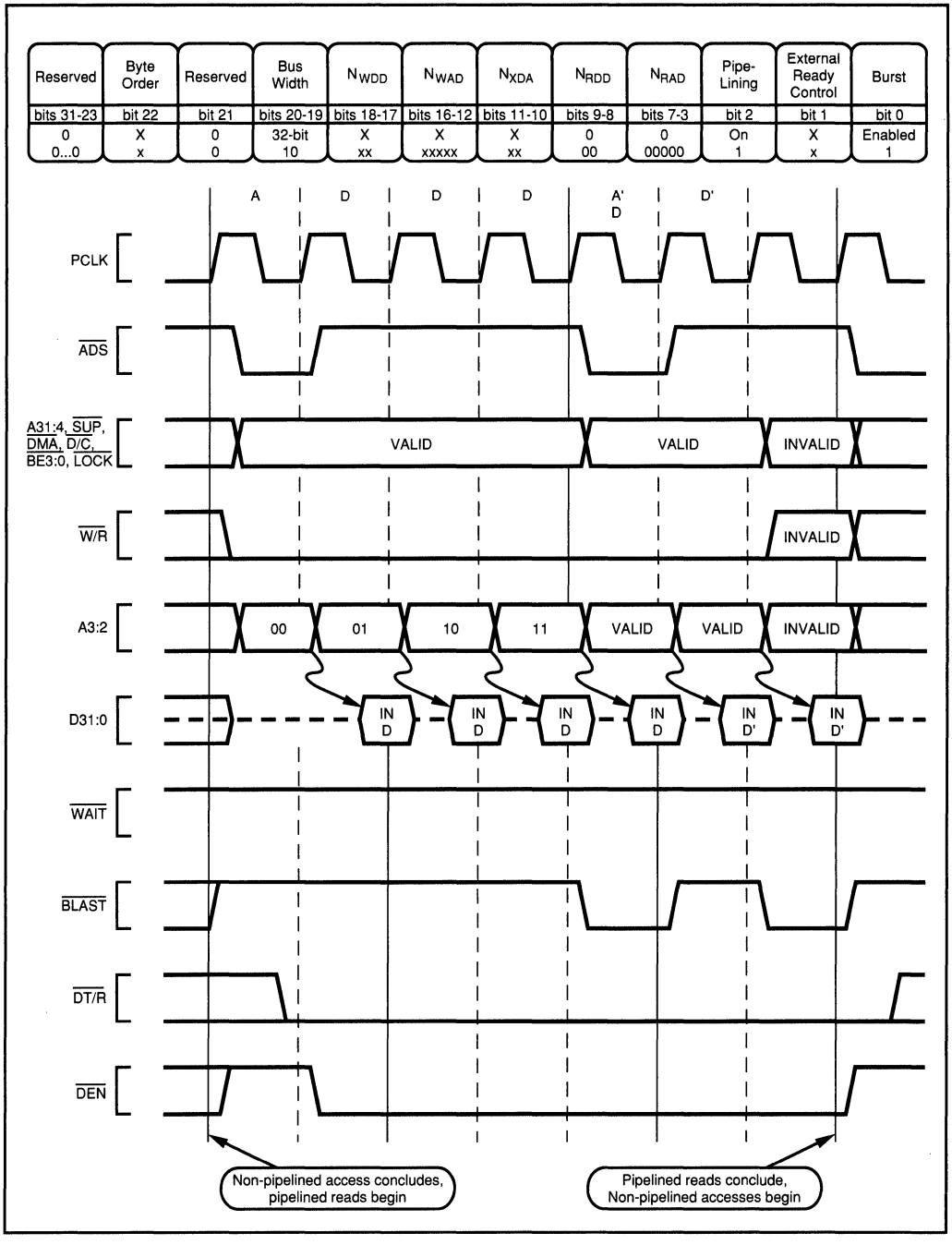


Figure 11-13. Burst Pipelined-Read Waveform

When pipelining is enabled in a region, the $\overline{\text{READY}}$ and $\overline{\text{BTERM}}$ inputs are ignored during read accesses to that region. The N_{XDA} wait states are forced to 0 for read accesses. $\overline{\text{READY}}$, $\overline{\text{BTERM}}$ and N_{XDA} behave as programmed for write accesses.

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.

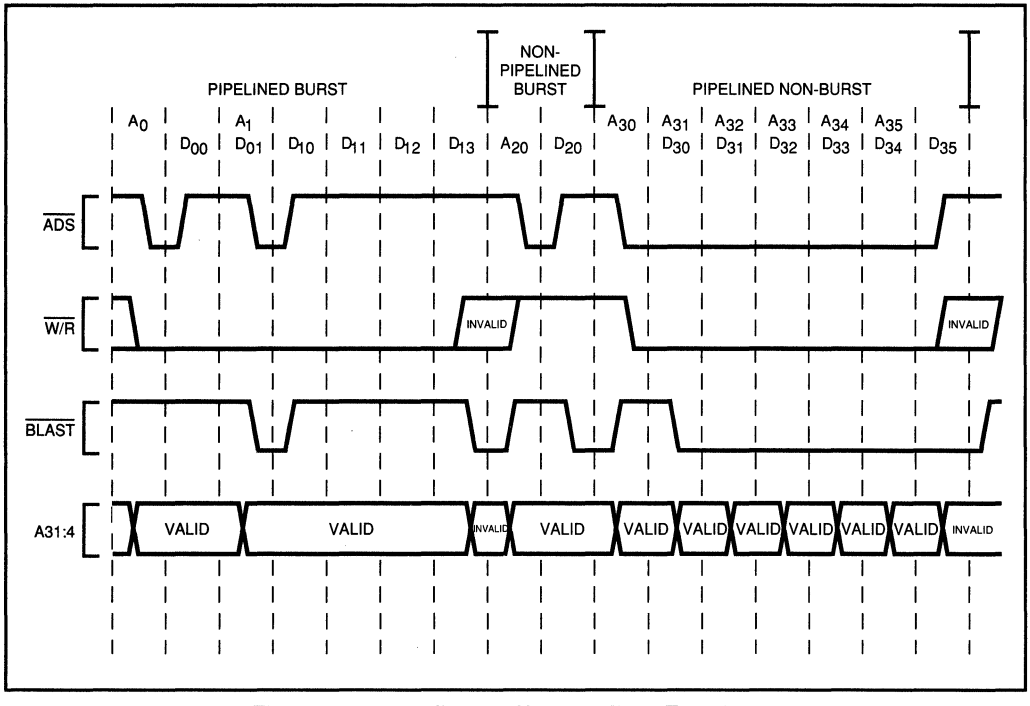


Figure 11-14. Pipelined to Non-Pipelined Transitions

Ready and Burst Terminate Control

The $\overline{\text{READY}}$ (memory ready) and $\overline{\text{BTERM}}$ (burst terminate) inputs allow the external memory system to dynamically control the memory access. $\overline{\text{READY}}$ extends the number of wait states in a data transfer. The assertion of $\overline{\text{READY}}$ indicates that either valid read data is on the bus, or a write transfer has completed. $\overline{\text{BTERM}}$ terminates a burst access by causing another address cycle ($\overline{\text{ADS}}$ asserted) to be generated. The $\overline{\text{BTERM}}$ input is provided for peripherals that need to control bursting dynamically. Most systems will not need to use the $\overline{\text{BTERM}}$ 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 0...0	X x	0 0	32-bit 10	X XX	X XXXXX	1 01	1 01	2 00010	OFF 0	1 01	Enabled 1

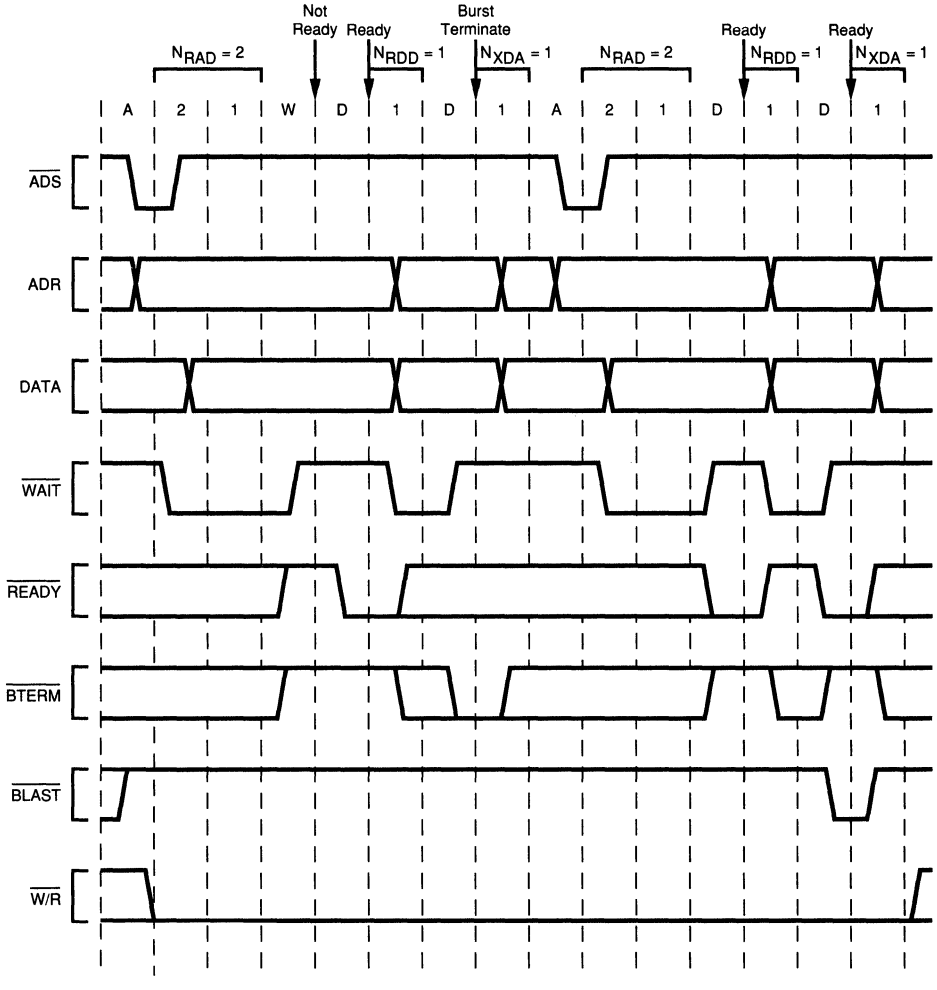


Figure 11-15. $\overline{\text{READY}}$ and $\overline{\text{BTERM}}$ Waveform

Both $\overline{\text{READY}}$ and $\overline{\text{BTERM}}$ work with the internal wait-state generator. They are not sampled until the programmed number of wait states has expired. $\overline{\text{READY}}$ may be used to extend these wait states: N_{RAD}, N_{RDD}, N_{WAD}, or N_{WDD}. $\overline{\text{READY}}$ cannot be used to extend the N_{XDA} wait states; the desired number of N_{XDA} wait states must be programmed in the region table.

If $\overline{\text{READY}}$ is not asserted when the internal wait-state counter expires, additional wait states will be inserted until $\overline{\text{READY}}$ is asserted. The behavior is the same for burst and non-burst regions. When pipelining is enabled in a region, the $\overline{\text{READY}}$ and $\overline{\text{BTERM}}$ inputs are ignored during read accesses to that region.

$\overline{\text{BLAST}}$ is asserted when the programmed number of wait states prior to the last data cycle of an access has expired. When $\overline{\text{READY}}$ and $\overline{\text{BTERM}}$ are enabled, $\overline{\text{BLAST}}$ will remain asserted until $\overline{\text{READY}}$ is asserted, and the data transfer completes.

The $\overline{\text{BTERM}}$ input is used to break up, or terminate a burst access. When $\overline{\text{BTERM}}$ is asserted, $\overline{\text{READY}}$ is ignored, and $\overline{\text{ADS}}$ is asserted. The assertion of $\overline{\text{ADS}}$ indicates the beginning of a new access. Data is not lost if a burst access is terminated. (Read data is accepted on the clock edge that $\overline{\text{BTERM}}$ is asserted. Writes are assumed to be complete.) The bus controller will continue to issue burst accesses for the remaining data. This may result in a non-aligned burst access, such as a three-word burst, starting at an odd address.

External Bus Arbitration

The 80960CA provides a shared bus protocol (HOLD, HOLDA) for another bus master to gain access to the 80960CA's bus. The HOLD input signal indicates that another processor or peripheral wishes to take control of the bus. The HOLDA (Hold Acknowledge) output signal is the acknowledgment that the 80960CA has relinquished the bus. The 80960CA will signal the other processor or peripherals that it needs to access the bus by using the bus request signal (BREQ). The 80960CA can also generate an atomic read-modify-write access ($\overline{\text{LOCK}}$).

Hold and Hold Acknowledge Handshaking

When another bus master wishes to gain control of the bus, it must get permission from the 80960CA. The external master must first assert the HOLD signal. This forces the 80960CA to relinquish control of the bus at the end of the current bus access. The HOLD request will be acknowledged between internal-DMA load and store operations, atomic requests (read-modify-write accesses that assert $\overline{\text{LOCK}}$), and if the bus controller queue is not empty.

The 80960CA acknowledges that it has relinquished the bus by asserting HOLDA, which is asserted in the same cycle that the bus enters the high-impedance state. The processor will continue to run until it needs to access the external bus. The 80960CA will assert the bus request signal (BREQ) when it needs to access the bus.

When HOLD is removed, HOLDA is deasserted on the following cycle, and the bus and control signals are driven. The HOLD signal is a synchronous input and must be synchronized with PCLK in order to meet the required set up and hold times.

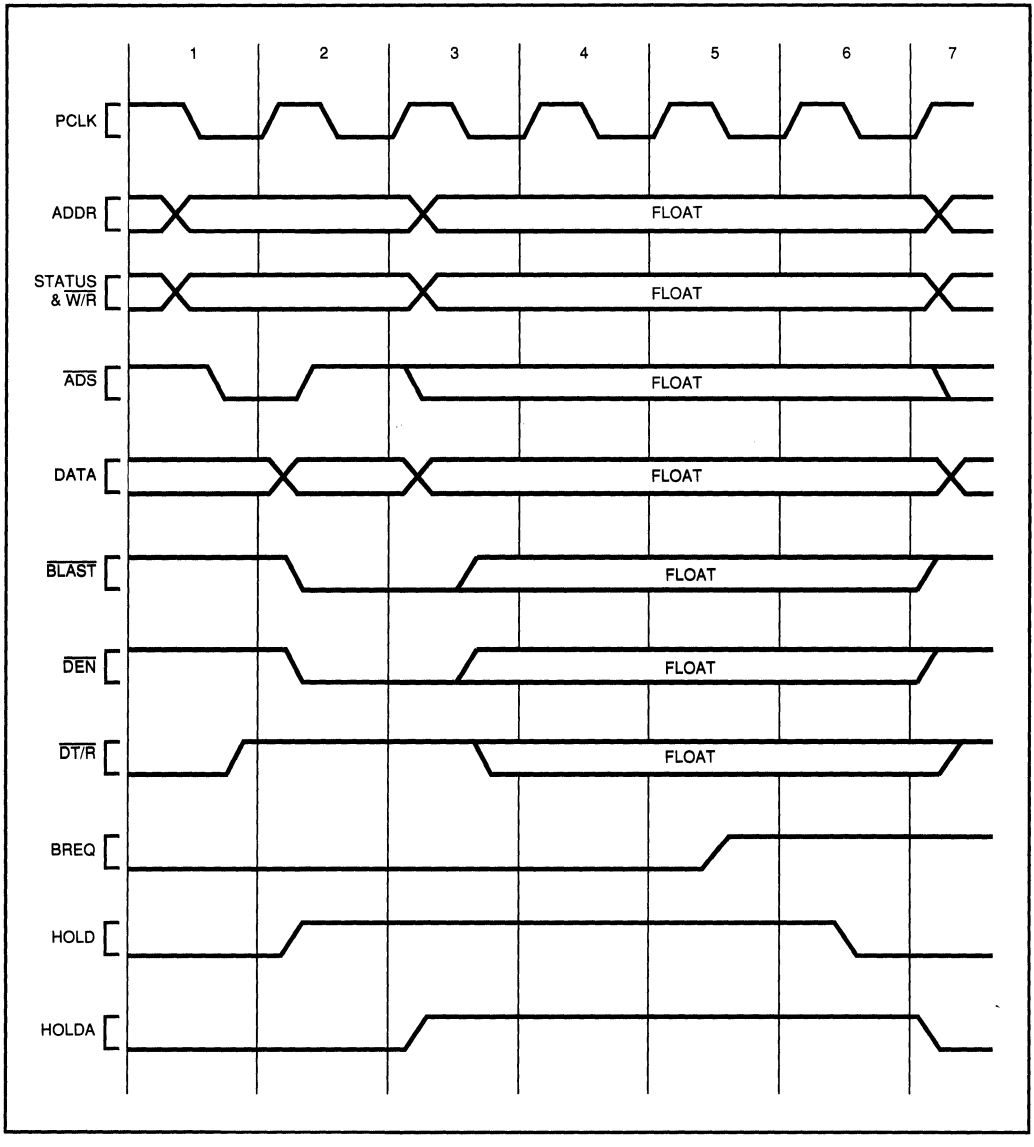


Figure 11-16. HOLD, HOLDA and BREQ Bus Arbitration

THE BUS REQUEST SIGNAL

The bus request signal (BREQ) indicates that the 80960CA needs to regain control of the external bus. Once the 80960CA has transferred control of the bus to an external bus master, there is a danger that the control of the bus may not be regained by the 80960CA for a long time. The BREQ output indicates that the 80960CA has pending bus access. These accesses may be the result of load or store instructions, pending instruction fetches, or an internal-DMA operation. BREQ is provided so that an external bus arbitrator can evaluate the system priorities and then give the bus to the appropriate bus master.

RESET CONSIDERATIONS

The HOLD and HOLDA arbitration can function even during the reset state. The bus controller will acknowledge HOLD while $\overline{\text{RESET}}$ is asserted. This effects the timing of $\overline{\text{BLAST}}$, which is first asserted as a result of asserting $\overline{\text{RESET}}$. Then if the bus is relinquished while $\overline{\text{RESET}}$ is asserted, $\overline{\text{BLAST}}$ floats with the assertion of HOLDA.

If $\overline{\text{RESET}}$ is asserted while the 80960CA has asserted HOLDA (acknowledged the HOLD), the processor will remain in the HOLDA state. The processor will not go into the reset state until HOLD is removed, and the processor removes HOLDA.

The Lock Signal

The assertion of the $\overline{\text{LOCK}}$ output indicates that the 80960CA is executing an atomic read-modify-write operation. These atomic instructions (**atadd, atmod**) require indivisible memory access.

$\overline{\text{LOCK}}$ is asserted in the address cycle of the read transfer. It is deasserted in the first cycle after the write transfer has completed. HOLD and DMA bus requests will be acknowledged while $\overline{\text{LOCK}}$ is asserted. The $\overline{\text{LOCK}}$ signal indicates that other agents should not write data to any address falling within the quad-word boundary of the address that was on the bus when $\overline{\text{LOCK}}$ was asserted. The $\overline{\text{LOCK}}$ signal 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 11-17.)

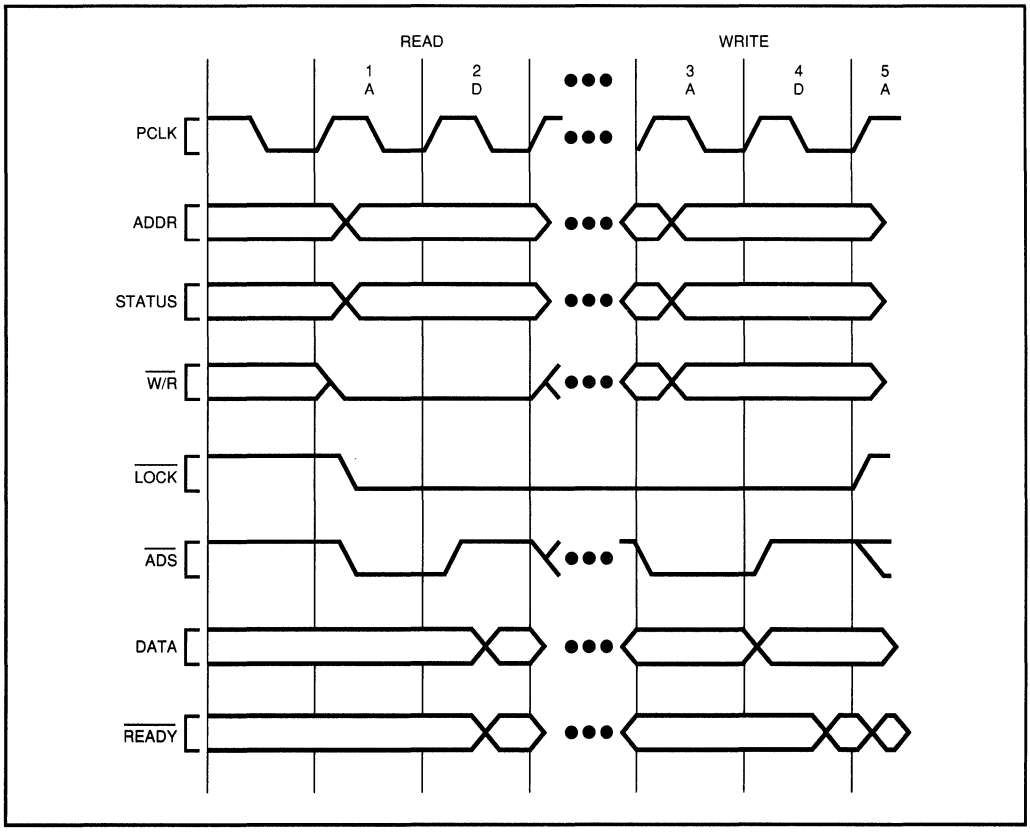


Figure 11-17. The $\overline{\text{LOCK}}$ Signal

Bus Interface Examples 12

CHAPTER 12
BUS INTERFACE EXAMPLES

This chapter describes how to interface the 80960CA 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 in this chapter assume a 33 MHz bus, and issues discussed in each example are independent of operating frequency.

NON-PIPELINED BURST SRAM INTERFACE

This chapter uses a simple SRAM design to demonstrate how the 80960CA 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 both 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 come in a wide variety of packages and densities. The SRAM's address pins are always dedicated as inputs. The 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. The 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.

When $\overline{CE}$ and $\overline{OE}$ are asserted, and $\overline{WE}$ is not asserted, the memory is read. When $\overline{CE}$ and $\overline{WE}$ are asserted, the memory is written. The $\overline{OE}$ input becomes don't care when $\overline{WE}$ is asserted. However, it is recommended that the $\overline{OE}$ signal is not asserted at the beginning or end of a write cycle, because this can lead to bus contention.

Implementation

The following example illustrates a 32-bit-wide burst-access SRAM interface. The design may be simplified if burst-access modes are not required, and the design is easily modified for 8-bit, or 16-bit-wide buses.

The $\overline{WAIT}$ signal, 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 the mid-burst addresses. External address generation improves performance in burst accesses.

Block Diagram

The 32-bit burst SRAM interface consists of the chip-select logic; a state-machine PLD; and the write-enable logic.

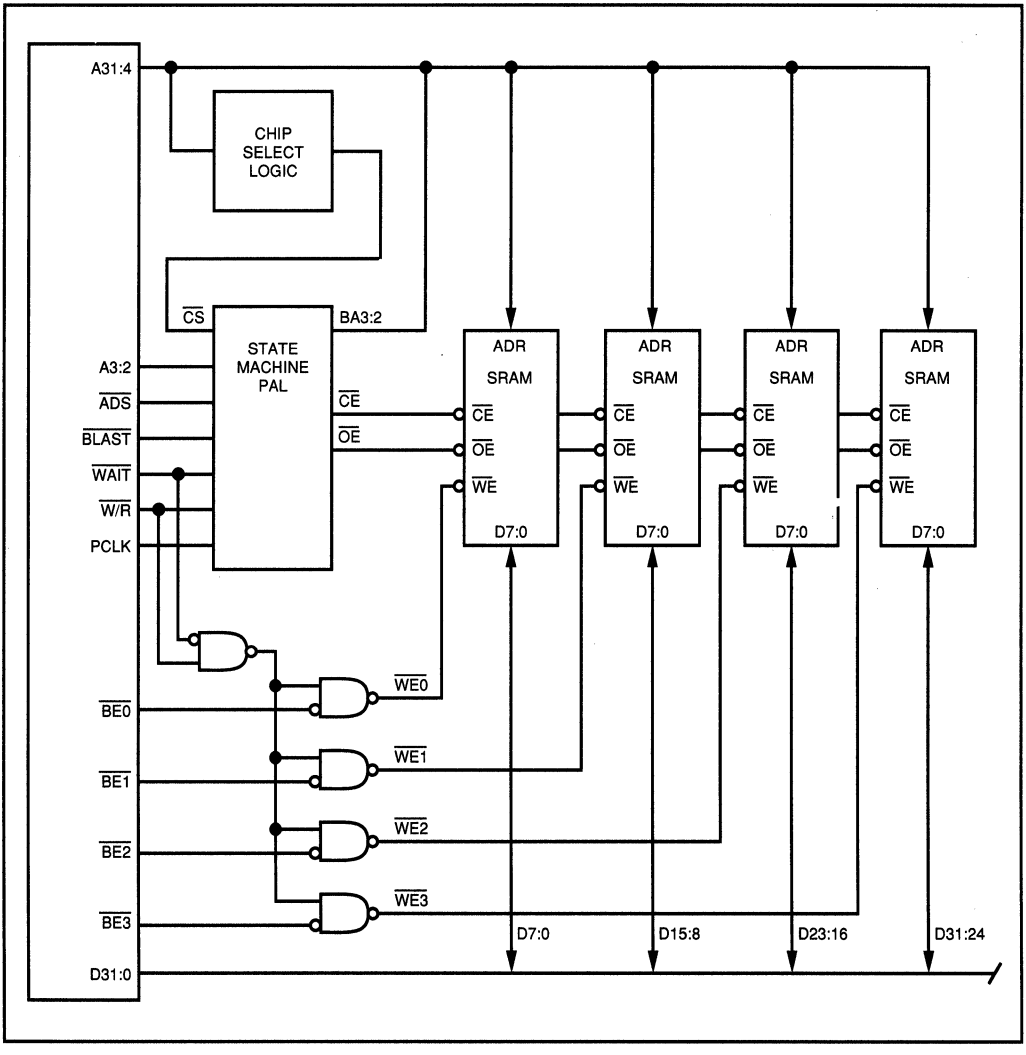


Figure 12-1. Non-Pipelined Burst SRAM Interface

CHIP SELECT LOGIC

The chip-select logic is a simple asynchronous data selector; it can be implemented with a demultiplexer or a PLD. The 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 $\overline{CE}$ and $\overline{OE}$ signals to the SRAM. This PLD also contains the next-address generation logic. The next-address generation logic improves the performance of burst accesses. This improvement occurs because the worst-case address valid delay of the 80960CA is longer than the worst-case delay of the PLD.

WRITE-ENABLE GENERATION LOGIC

The write-enable generation logic generates the $\overline{WE}$ signal to the SRAM. The $\overline{WE}$ signals are conditioned on the 80960CA byte-enables ($\overline{BE3}$-$\overline{BE0}$), the write-read signal (W/$\overline{R}$), and the wait signal ($\overline{WAIT}$).

There is a write-enable signal, $\overline{WE3}$-$\overline{WE0}$, for each byte position corresponding to the byte enable signals, $\overline{BE3}$-$\overline{BE0}$, respectively; this allows byte, short-word and word-wide writes. Read-accesses to this memory system always result in word reads. The 80960CA, in the case of byte- or short-word reads, reads the data from the correct place on the data bus.

CHIP SELECT GENERATION

The assertion of the $\overline{ADS}$ signal during the rising edge of PCLK indicates that the address is valid. The address setup time to this clock edge is the period of PCLK (T_{PP}), minus the address output-delay (T_{OV}). The $\overline{CS}$-signal generation time ($T_{\overline{CS}\_gen}$) must satisfy the input setup-time of the State-Machine PLD(T_{PLD_setup}). Therefore:

$$T_{\overline{CS}\_gen} = T_{PP} - T_{OV} - T_{PLD\_setup}$$

Waveforms

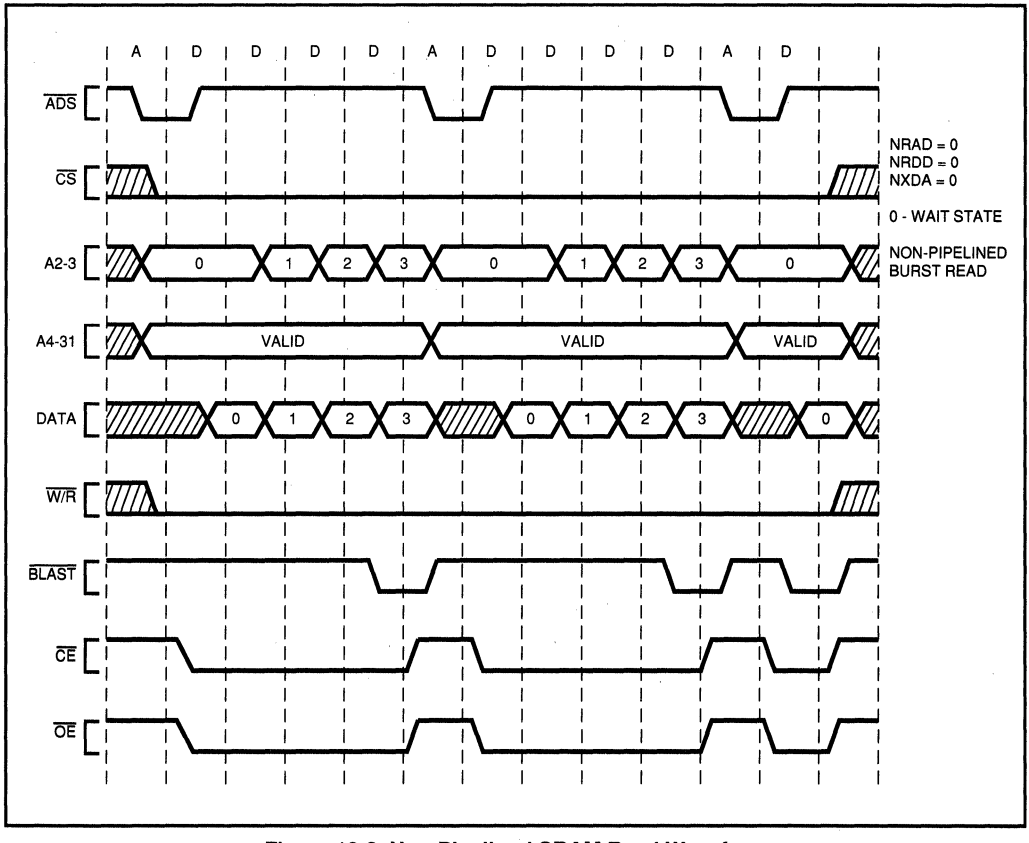


Figure 12-2. Non-Pipelined SRAM Read Waveform

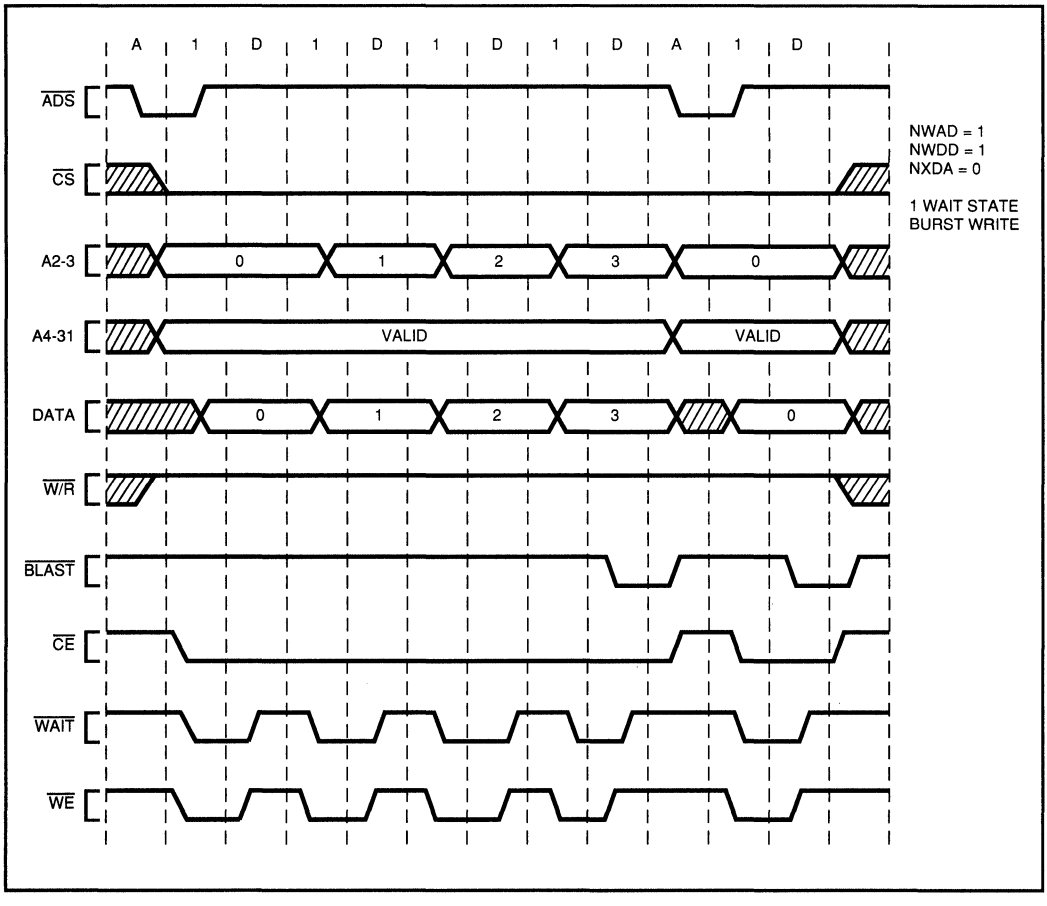


Figure 12-3. Non-Pipelined SRAM Write Waveform

WAIT STATE SELECTION

The 80960CA incorporates an internal wait-state generator; the selection of wait states is dictated by the memory system.

The number of N_{RAD} wait states required is a function of the output-enable access time, the chip-enable access time, or the address access time. N_{RAD} must be selected so that the wait-states and the data cycle accommodate the longest of these times. It is important to consider the PLD output delay.

The number of N_{RDD} wait states required is a function of the address access time. N_{RDD} must be selected so that the wait states and the data cycle accommodate the address to data time of the memory system. If the memory system is using the burst-addresses provided by the 80960CA, then it is important to consider the address output delay from the 80960CA. If external address generation is used, the PLD delay is important.

The number of N_{WAD} and N_{WDD} wait states required is a function of the memory write-cycle time.

The number of N_{XDA} wait-states required is a function of the output-to-float time of the memory system. N_{XDA} determines how soon the 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 80960CA.

THE OUTPUT-ENABLE AND WRITE-ENABLE LOGIC

The output-enable signal is simply:

$$\overline{OE} = W/\overline{R}$$

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:

$$\overline{WE} = !(WRITE \ \& \ WAIT \ \& \ BE);$$

or

$$\overline{WE0} = !W/\overline{R} \ || \ \overline{WAIT} \ || \ \overline{BE0};$$
$$\overline{WE1} = !W/\overline{R} \ || \ \overline{WAIT} \ || \ \overline{BE1};$$
$$\overline{WE2} = !W/\overline{R} \ || \ \overline{WAIT} \ || \ \overline{BE2};$$
$$\overline{WE3} = !W/\overline{R} \ || \ \overline{WAIT} \ || \ \overline{BE3};$$

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 80960CA 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 the 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 the $\overline{BSRAM\_CS}$ are asserted, the $\overline{CE}$ signal is asserted. $\overline{CE}$ remains asserted until $\overline{BLAST}$ is asserted; $\overline{BLAST}$ indicates the access is complete. The $\overline{CE}$ signal is the output of the state register; therefore, the $\overline{CE}$ output delay is the clock-to-output time of the PLD. Minimizing the $\overline{CE}$ delay provides more memory access time.

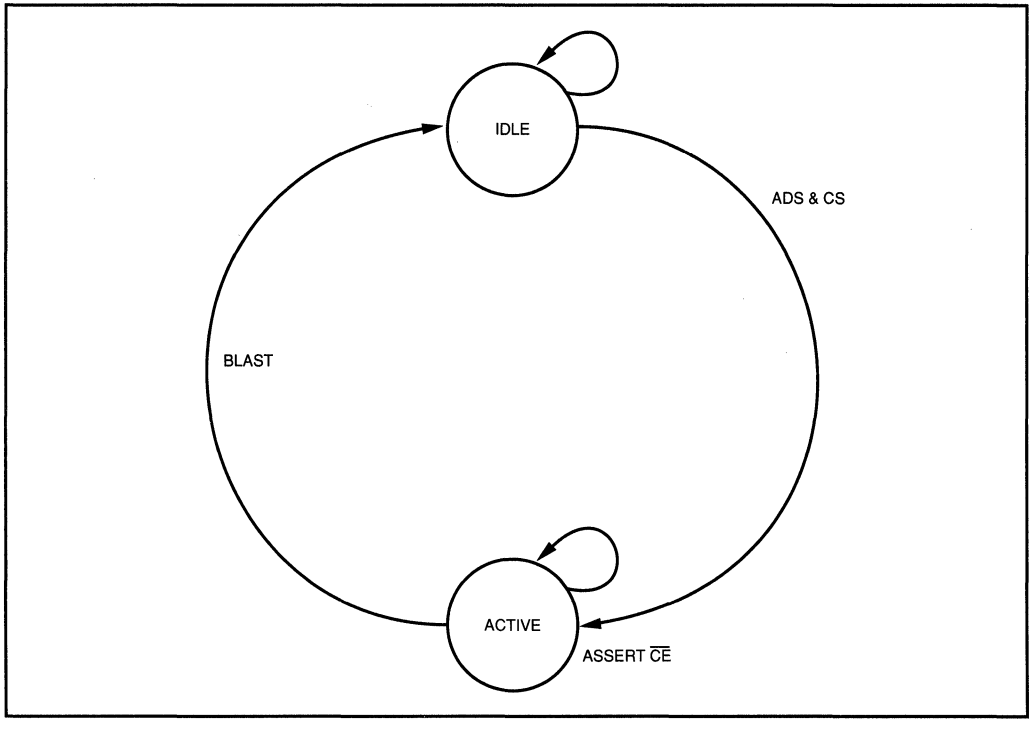


Figure 12-4. Chip-Enable State Machine

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 the registers within the PLD. The address generation time then corresponds to the clock-to-output time of the PLD. The BA3:2 signals are forced to 0 when $\overline{\text{BLAST}}$ is asserted.

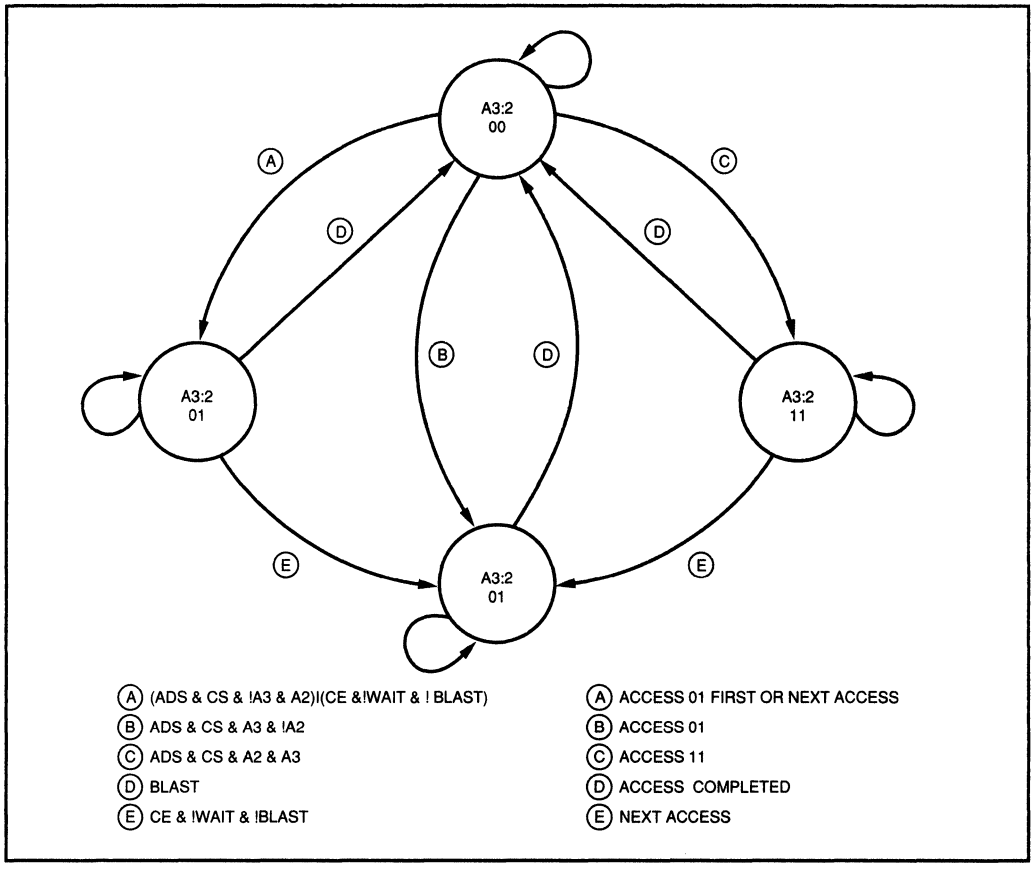


Figure 12-5. A3:2 Address Generation State Machine

The pseudo-code description below 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.

```
        Pseudo-code key:
                !       logical NOT;
                &&      logical AND;
                || logical OR;
                ==      equality test;
                = value assignment;

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.

Trade-offs and Alternatives

The SRAM example which was 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 80960CA 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 the memory-access times. The memory-access parameters are entered into the Memory Region Configuration Table via software.

PIPELINED-READ SRAM INTERFACE

Interfacing to pipelined SRAMThe 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 80960CA. 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 12-6.)

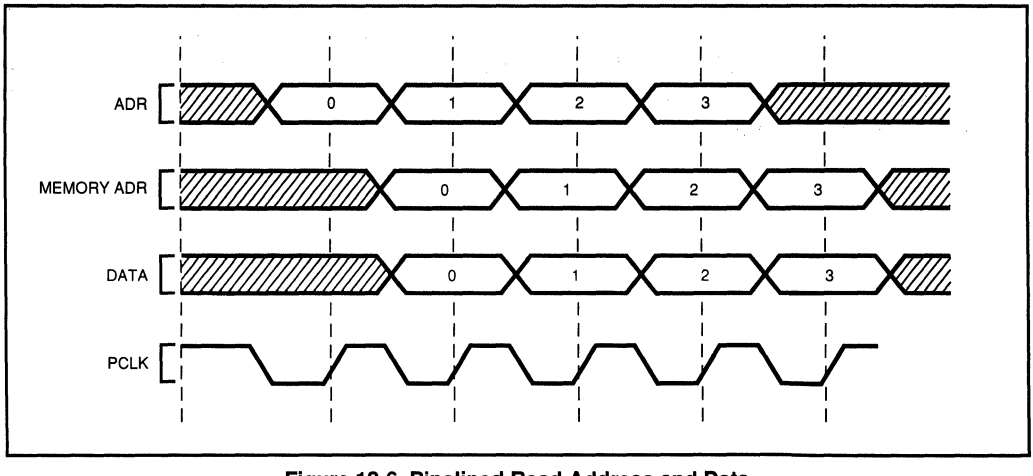


Figure 12-6. Pipelined Read Address and Data

Block Diagram

The same SRAM used in a non-pipelined-read memory system is used in a pipelined-read memory system. Figure 12-7 shows a 32-bit-wide burst read-pipelined memory system. Burst mode is used to speed write accesses.

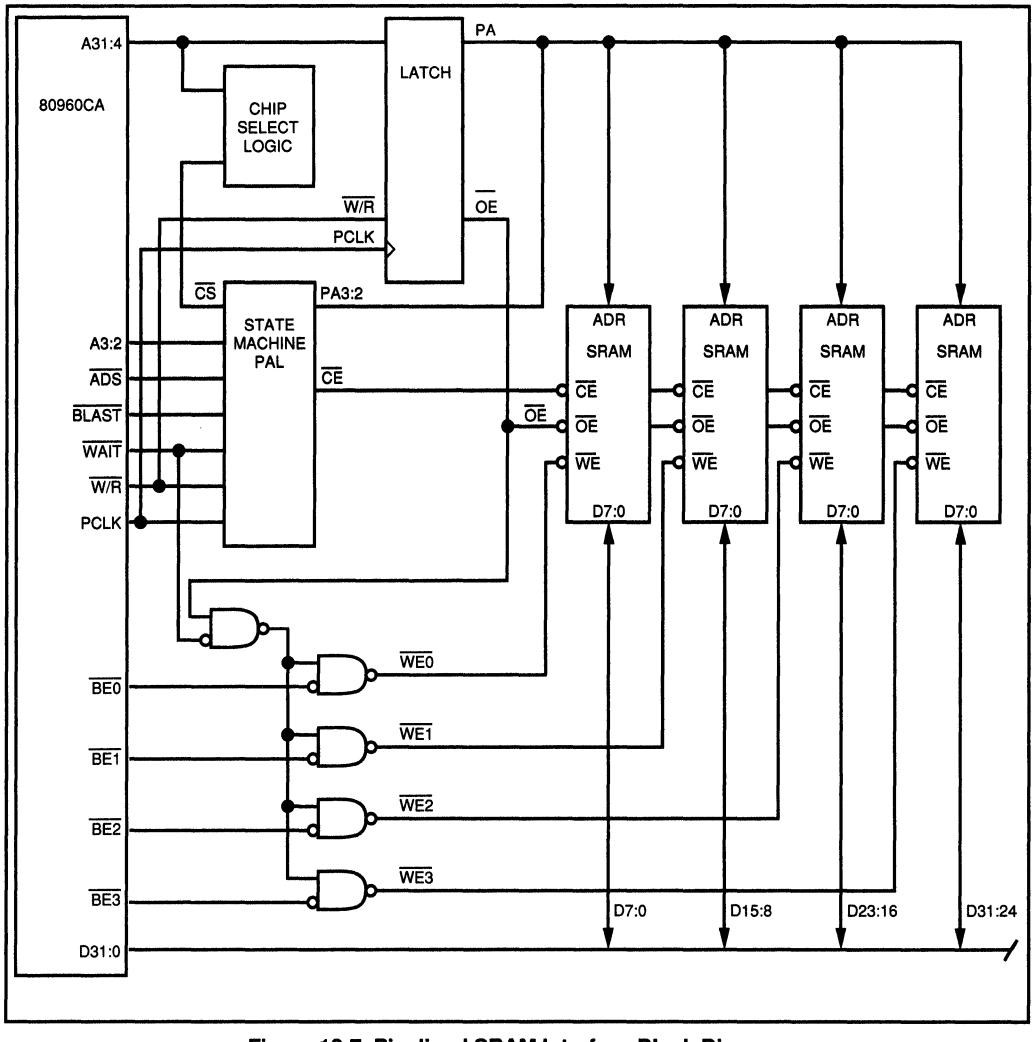


Figure 12-7. Pipelined SRAM Interface Block Diagram

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.

The chip-select logic is a simple asynchronous data selector. The chip select ($\overline{CS}$) is based only on the address and is not qualified with any other signals. (See the section in this chapter titled *Non-Pipelined 1-Burst SRAM* example for more information on chip-select generation.)

THE ADDRESS LATCH

During pipelined-reads, the 80960CA 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.

THE STATE MACHINE PLD

The state-machine PLD contains logic to control the $\overline{CE}$ signal and the address signals A3:2. The $\overline{CE}$ signal is controlled by a simple state-machine; A3:A2 automatically increment during burst accesses. The A3:2 signals are pipelined, so they must be latched for read accesses. Write accesses are not pipelined; therefore it is necessary to latch A3:2 on reads and pass A3:2 through, for burst writes. The A3:2 generation is implemented as a state machine to achieve the minimum address delay out of the PLD. The 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.

THE WRITE ENABLE LOGIC

The write-enable logic uses the byte-enable signals ($\overline{BE3:0}$); the $\overline{WAIT}$ signal; and a latched version of the W/$\overline{R}$ signal ($\overline{OE}$). Therefore:

$$\overline{WE} = !(!OE \;\&\; WAIT \;\&\; BE);$$

or:

$$\overline{WE0} = !\overline{OE} \;||\; \overline{WAIT} \;||\; \overline{BE0};$$
$$\overline{WE1} = !\overline{OE} \;||\; \overline{WAIT} \;||\; \overline{BE1};$$
$$\overline{WE2} = !\overline{OE} \;||\; \overline{WAIT} \;||\; \overline{BE2};$$
$$\overline{WE3} = !\overline{OE} \;||\; \overline{WAIT} \;||\; \overline{BE3};$$

Waveforms

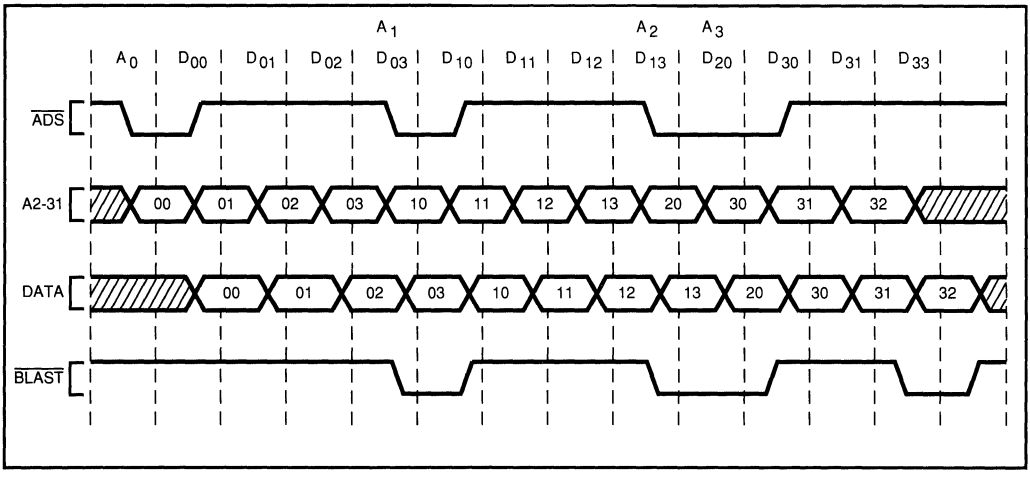


Figure 12-8. Pipelined Read waveform

The DEN signal remains asserted as long as consecutive pipelined-read accesses continue. $\overline{DEN}$ and DT/$\overline{R}$ are related to the data, not the address; therefore, $\overline{DEN}$ and DT/$\overline{R}$ 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 the $\overline{ADS}$ signal remaining asserted for several clock cycles. Similarly, $\overline{BLAST}$ remains asserted for several clock cycles.

The W/$\overline{R}$ signal behaves slightly differently for pipelined reads than for non-pipelined reads. The W/$\overline{R}$ signal is not valid for the last cycle of a pipelined read. This requires that the W/$\overline{R}$ signal be latched for pipelined reads similar to A31:2. The following signals are pipelined during pipelined-read accesses: A31:2,$\overline{BE3:0}$, $\overline{SUP}$, $\overline{DMA}$, and D/$\overline{C}$. All of these pipelined signals are invalid during the last cycle of a pipelined read.

The 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 the address delay maximizes access time.

THE STATE MACHINES

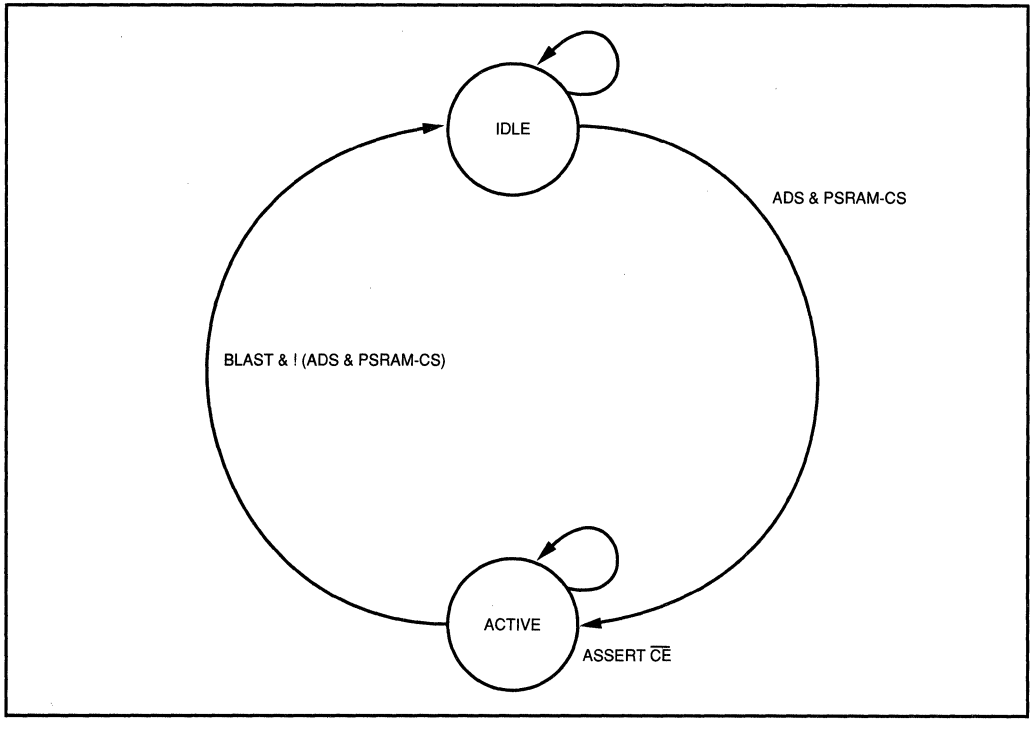


Figure 12-9. Pipelined-Read Chip-Enable State Machine

The chip enable signal ($\overline{CE}$) is controlled by a simple state machine. The state machine is normally in the idle state and $\overline{CE}$ is not asserted. When $\overline{ADS}$ and $\overline{PSRAM\_CS}$ are asserted, the $\overline{CE}$ state machine goes to the active state. $\overline{CE}$ remains active until $\overline{BLAST}$ is asserted.

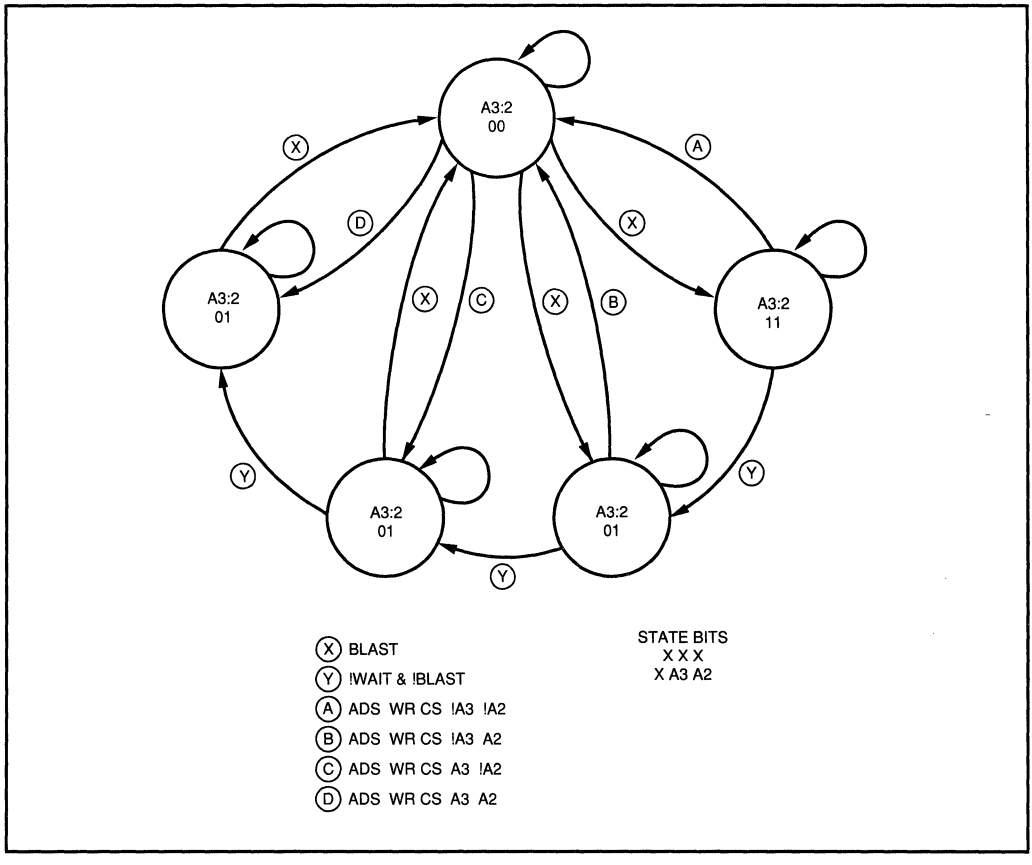


Figure 12-10. Pipelined Read PA3:2 State-Machine Diagram

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:
                !       logical NOT;
                &&      logical AND;
                || logical OR;
                ==      equality test;
                :=      clocked assignment;
                = value assignment;

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.

Trade-offs and Alternatives

The example described above demonstrates a burst pipelined-read SRAM memory interface. The burst mode is used to improve the write performance. If the 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 banks of SRAM.

INTERFACING DYNAMIC RAM (DRAM) WITH THE 80960CA

This section provides an overview of DRAM and DRAM access modes; details of an 80960CA-specific DRAM interface; and two specific design examples are also included. One design uses the integrated DMA unit to refresh the DRAM; the other example uses the $\overline{CAS}$-before-$\overline{RAS}$ method of refresh. Both designs illustrate the advantage of the burst bus on the 80960CA and the fast-column address access times available on many modern DRAMs.

The burst-bus and the memory-region configuration tables simplify interfacing DRAM to the 80960CA. The DRAM systems can be designed in many ways; there are memory access options, memory system configuration options, and refresh mode options.

OVERVIEW OF DRAM

DRAM's offer high-data density, fast access times, and low cost per bit. DRAM's are available in a wide variety of packages, making it easy to pack a lot of memory into a small space. The DRAM features described here are provided as general information. (See specific data sheets for detailed information.)

The burst mode bus of the 80960CA is well suited to the high-speed multiple-column access modes found in DRAM. The nibble, fast-page, and static-column modes of DRAM can easily be exploited to improve 80960CA memory-system performance.

All DRAM's have a multiplexed address bus; two address strobes ($\overline{RAS}$, $\overline{CAS}$); and a write-enable input ($\overline{WE}$). Some DRAM's also have an output-enable input ($\overline{OE}$). DRAM's are accessed by placing a valid-row address on the address input pins and asserting $\overline{RAS}$; then the column address is driven onto the DRAM address pins and $\overline{CAS}$ asserted. The write-enable ($\overline{WE}$) input on the DRAM determines whether the access is a read or a write. The output-enable input ($\overline{OE}$), found on some DRAM's, is used to control the DRAM output buffers, and can be useful for multi-banked 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 $\overline{RAS}$ (row-address strobe). Strobing the $\overline{CAS}$ (column-address strobe) input accesses the consecutive column data. The input address is ignored after the first column access.

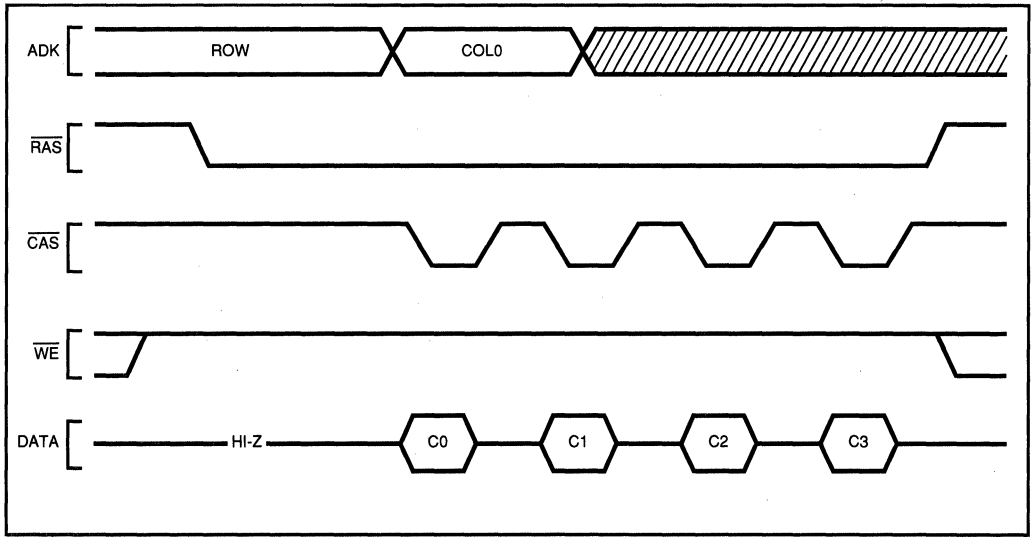


Figure 12-11. Nibble Mode Read

Fast-page mode DRAM is similar to nibble-mode DRAM, but 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 $\overline{RAS}$. Strobing the $\overline{CAS}$ input accesses the selected column data. During reads, the falling edge of $\overline{CAS}$ latches the address (internal to the DRAM) and enables the output. The 80960CA four-word burst bus can easily take advantage of the faster column-access times provided by fast-page mode DRAM.

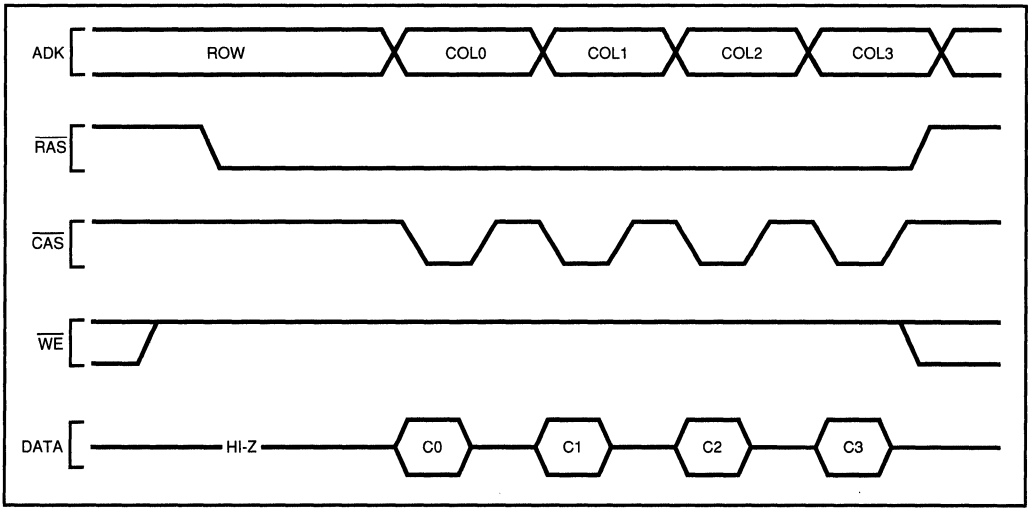


Figure 12-12. Fast-Page Mode DRAM Read

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 of the columns 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 80960CA 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.

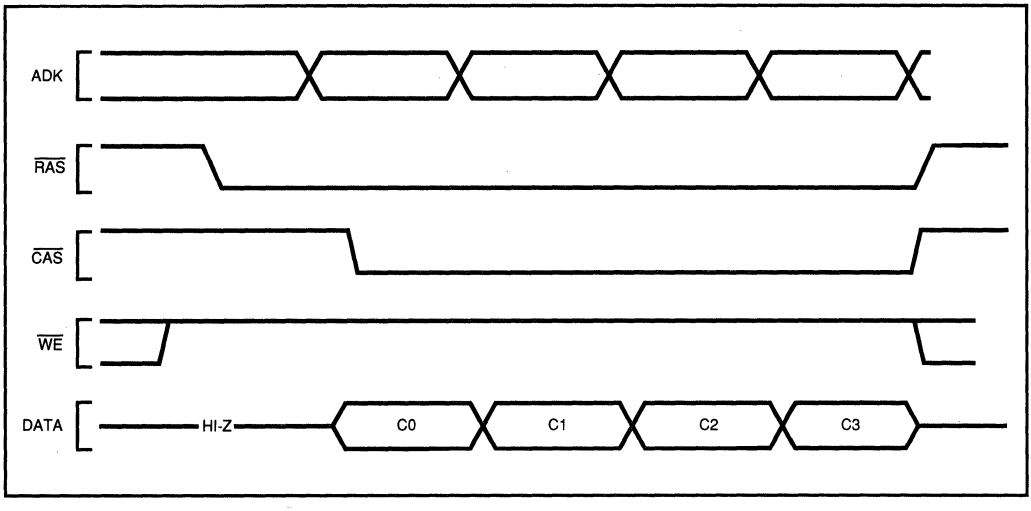


Figure 12-13. Static Column Mode DRAM read

DRAM REFRESH MODES

All DRAM's require periodic refreshing in order to retain data. DRAM's may be refreshed in one of two ways: RAS-only refresh, or CAS-before-RAS refresh. RAS-only refresh is realized by asserting a row address on the address pins and asserting RAS. CAS is not asserted. A single, RAS-only refresh cycle refreshes all the columns within the selected row. CAS-before-RAS refreshes do not require an address to be generated; the DRAM generates the row address with an internal counter.

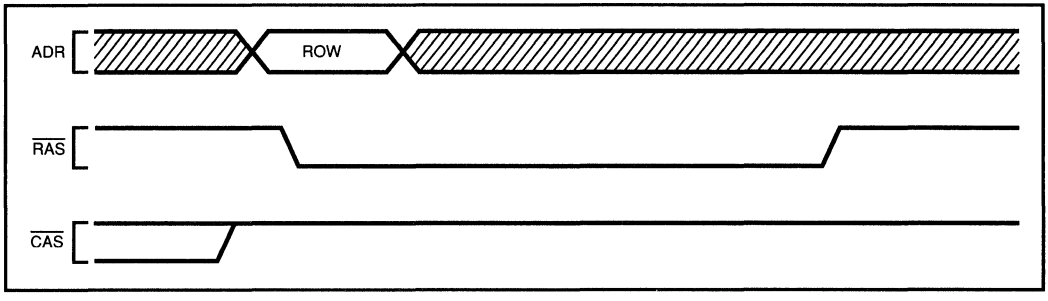


Figure 12-14. RAS only DRAM Refresh

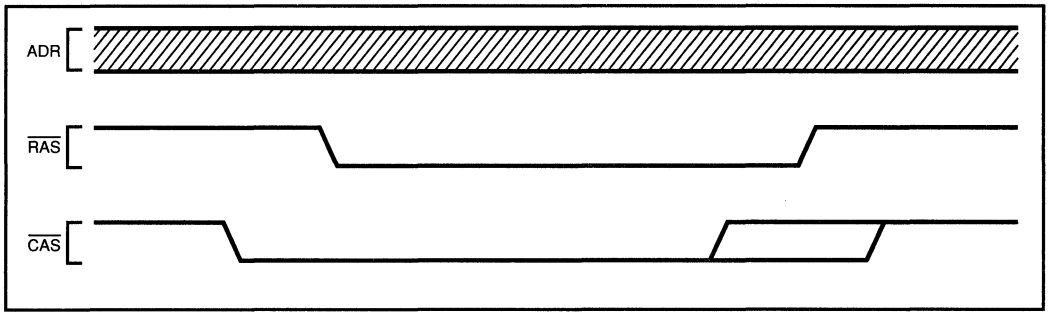


Figure 12-15. CAS-before-RAS DRAM Refresh

DRAM's 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 the memory rows are sequentially accessed when the refresh interval time expires. Distributed refresh implies that the 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, so may be inappropriate for some applications. Burst refreshing, however, guarantees that no refresh activity will occur between the refresh intervals. Some application may be able to make use 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

The address multiplexer inputs can be ordered so that 256 KByte through 4 MByte DRAM can be supported. Interleaving the upper address signals provides compatibility with all these memory densities. (Figure 12-16 illustrates this arrangement.) The availability of DRAM modules with standard pin-outs makes this an attractive way to ensure future memory expansion.

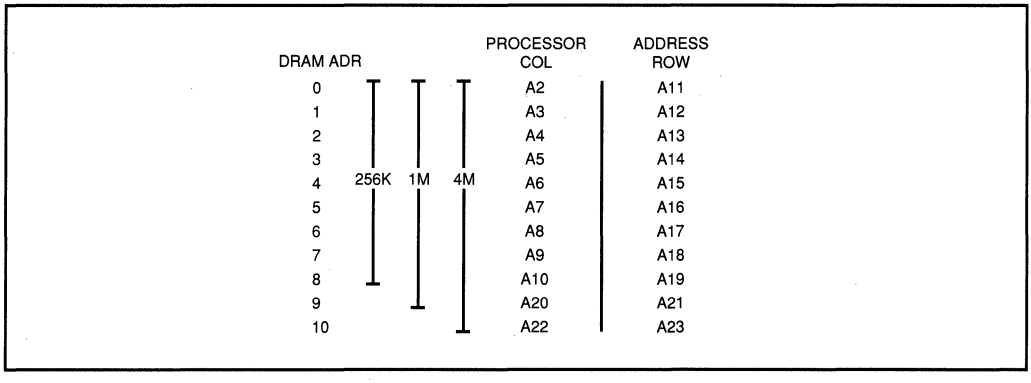


Figure 12-16. Address Multiplexer Inputs

SERIES-DAMPING RESISTORS

Series-damping resistors are recommended on all the DRAM control and address inputs. Series-damping resistors prevent overshoot and undershoot on the DRAM input lines. Damping is required because of the large capacitive load present when many DRAM's are connected together, combined with the inductance of the circuit-board traces. The values of damping resistors 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 over-damped, extending the memory-cycle time. If the damping resistor value is to low, overshoot or undershoot will not be sufficiently damped.

SYSTEM LOADING

The 80960CA can drive a large capacitive load. However, systems that have many banks of DRAM may require data buffers (and multiplexers for interleaved designs) in order to isolate the DRAM load from the 80960CA, or other system components with less drive capability (e.g., high-speed SRAM).

The $\overline{RAS}$ and $\overline{CAS}$ control inputs to the DRAM should also be designed with consideration for capacitive load. When many DRAM's are connected to a common $\overline{RAS}$ or $\overline{CAS}$ signal, the capacitive load can become considerable.

Design Example: Burst DRAM with Distributed 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 80960CA bus signals are used to develop a transparent DRAM controller that does not require any information about the memory subsystem.

Figure 12-17 shows the DRAM system design. The DRAM is configured as a single byte-accessible, 32-bit-wide bank. The RAS signal is common to the entire bank; the CAS0 - CAS3 serve as byte selects within the bank. 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 dram chips.

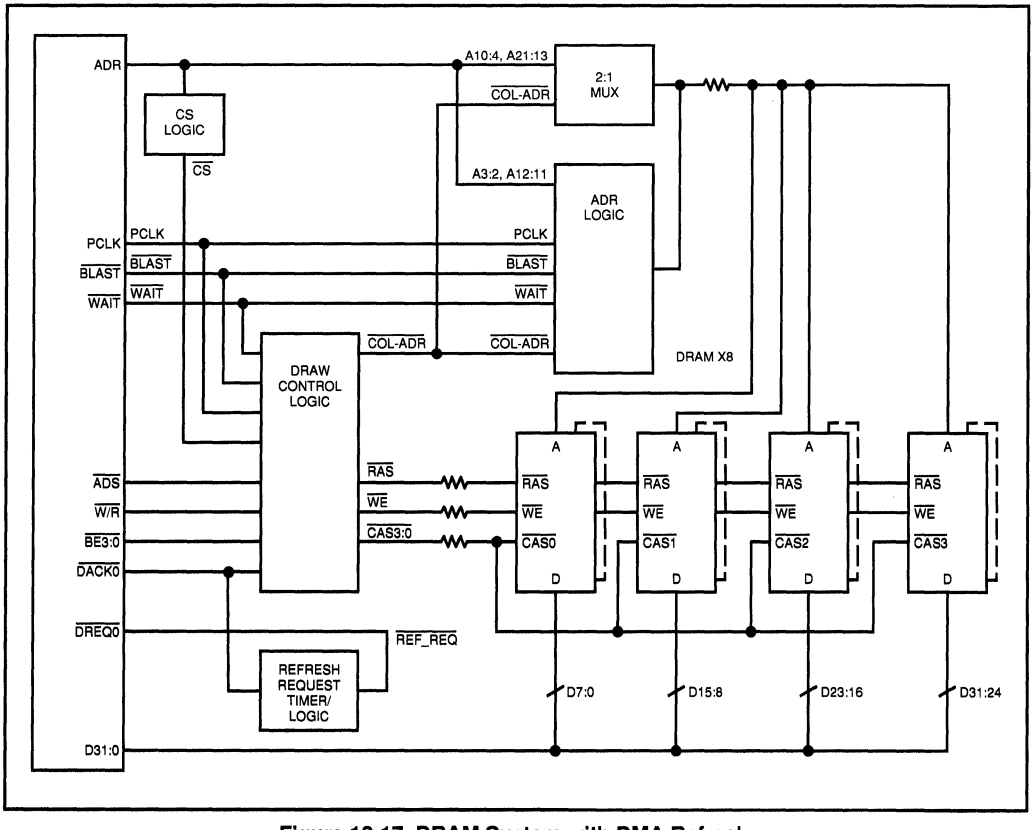


Figure 12-17. DRAM System with DMA Refresh

The control logic is divided down into three logical blocks: the DRAM control logic, the DRAM address-generation logic, and the refresh request timer logic. The DRAM control logic is the main controller. It controls the address multiplexer and all of the DRAM control lines during normal and refresh accesses. The 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

The 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 80960CA address-valid delay time (T_{OV}), plus the worst-case propagation delay through the input address multiplexer.

The DRAM address-generation logic must control the two least-significant bits of the DRAM address. During the initial DRAM access, the address-generation logic acts like a multiplexer. During column accesses within a burst, the address-generation logic generates consecutive addresses. Therefore, the DRAM address-generation logic is designed to function as a multiplexer and an address generator.

If an address-generator is used, the address-valid delay time is equal to the address generation time. The address-generation delay time consists of the clock-to-feedback and feedback-to-output delays for the device selected.

The following state-machine description illustrates the requirements for the 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 80960CA; and $\overline{\text{COL\_ADR}}$ from the DRAM controller logic. The signal $\overline{\text{COL\_ADR}}$ indicates if the DRAM controller is requesting the row address ($\overline{\text{COL\_ADR}}$ not asserted), or the column address ($\overline{\text{COL\_ADR}}$ asserted). The signals output from the DRAM address-generation logic are the two least-significant bits of the DRAM address, DRAM_ADR2:3.

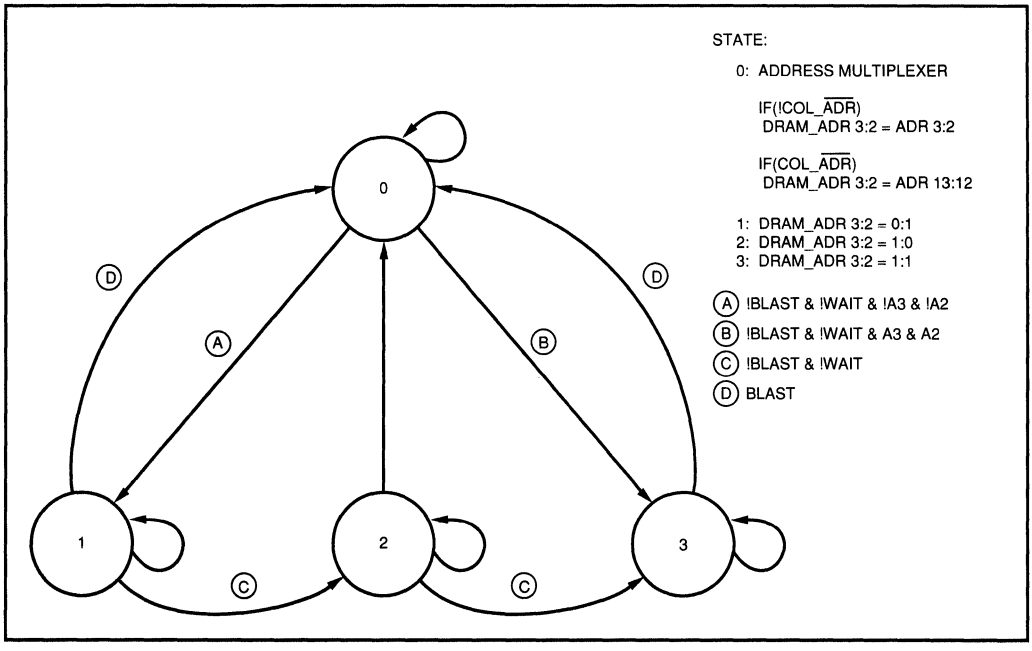


Figure 12-18. DRAM Address Generation State Machine

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:
                !       logical NOT;
                &&      logical AND;
                ==      equality test;
                = value assignment;

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 12-19 is a state machine describing the DRAM control logic. The state machine or subsets of the state machine shown 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.

The signals going into the DRAM control logic are: $\overline{ADS}$; PCLK; W/$\overline{R}$; $\overline{BLAST}$; $\overline{WAIT}$; $\overline{BE0}$-$\overline{BE3}$ from the bus controller; $\overline{DACK0}$, the DMA acknowledge signal; and $\overline{DRAM\_CS}$, a system generated chip select that indicates a DRAM access. The DRAM control logic generates $\overline{RAS}$, $\overline{CAS0:3}$, $\overline{WE}$ and $\overline{COL\_ADR}$. The control signal for the address multiplexer is $\overline{COL\_ADR}$.

The controller logic relies on the wait-state region table and the DMA controller. Programming these on chip peripherals is described later. The DMA acknowledge signal, $\overline{DACK0}$, is used to indicate a DRAM refresh cycle. The DRAM $\overline{WE}$ signal is generated with combinatorial logic ($\overline{WE}$ = !(W/$\overline{R}$)).

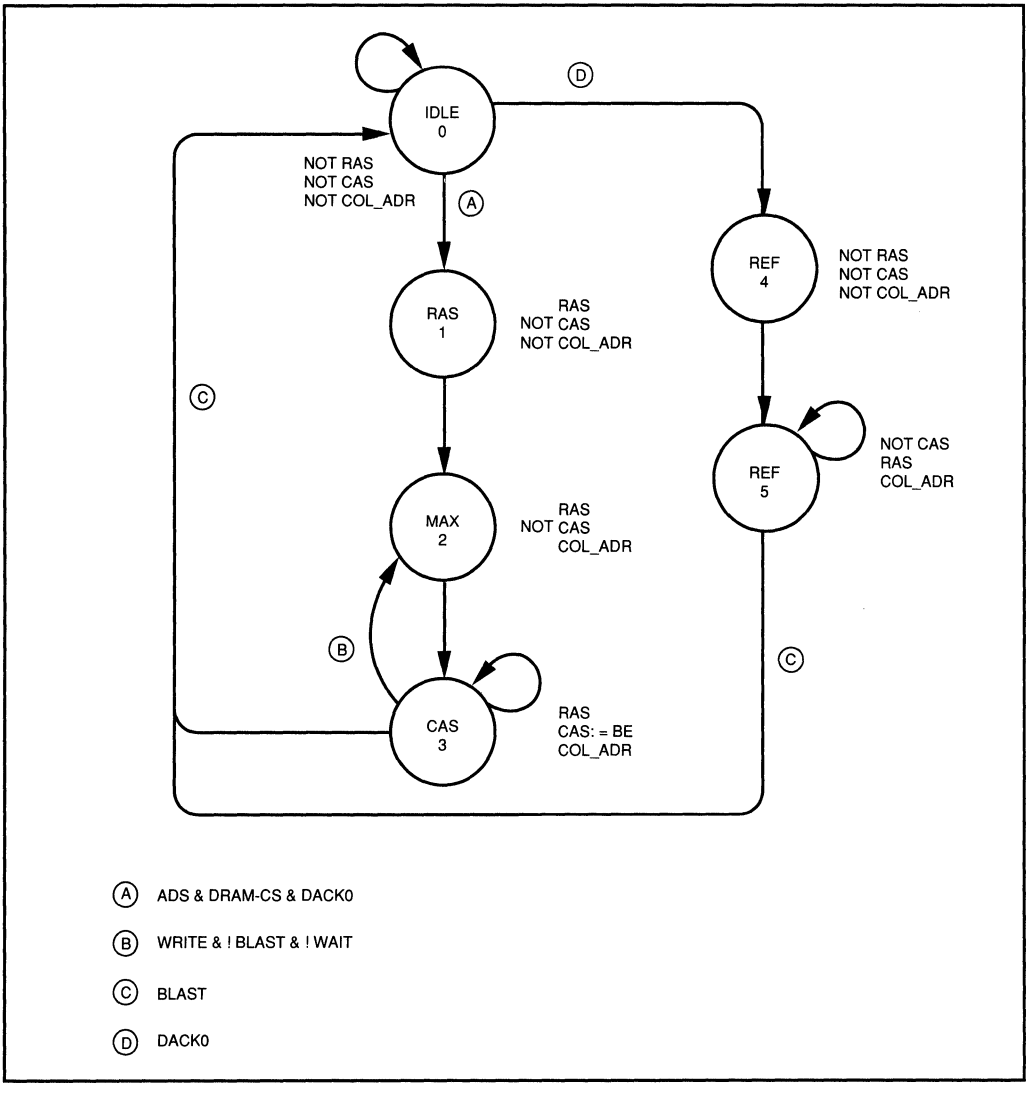


Figure 12-19. DRAM Controller State Machine

```
STATE_0: /* Idle */
        RAS             is not asserted;
        CAS0:3          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 RAS */
        RAS             is asserted;
        CAS0:3          is not asserted;
        COL_ADR         is not asserted;
        IF
                W/R; /* write */
        THEN
                the next state is STATE_3;
        ELSE /* read */
                the next state is STATE_2;
STATE_2: /* Static Column Mode Read, Assert CAS */
        RAS             is asserted;
        CAS0:3          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;
        CAS0:3          is not asserted;
        COL_ADR         is asserted;
        the next state is STATE_4;
```

STATE_4: /* Assert $\overline{CAS}$ */

$\overline{RAS}$ is asserted;
$\overline{COL\_ADR}$ is asserted;
$\overline{CAS0} = \overline{BE0}$;
$\overline{CAS1} = \overline{BE1}$;
$\overline{CAS2} = \overline{BE2}$;
$\overline{CAS3} = \overline{BE3}$;

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, $\overline{RAS}$ ONLY REFRESH */

$\overline{RAS}$ is not asserted;
$\overline{CAS0:3}$ is not asserted;
$\overline{COL\_ADR}$ is asserted;

the next state is STATE_6;

STATE_6: /* REFRESH CYCLE, Assert $\overline{RAS}$ */

$\overline{RAS}$ is asserted;
$\overline{CAS0:3}$ is not asserted;
$\overline{COL\_ADR}$ is asserted;

IF
    BLAST;
THEN
    the next state is STATE_0;
ELSE
    the next state is STATE_6;

DRAM REFRESH REQUEST AND TIMER LOGIC

The DRAM refresh request and timer logic is responsible for generating DMA requests at an appropriate interval, and for removing the DMA request after receiving a DMA acknowledge.

DRAM must be refreshed at a rate of 256 cycles/4 ms. 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 12-20.

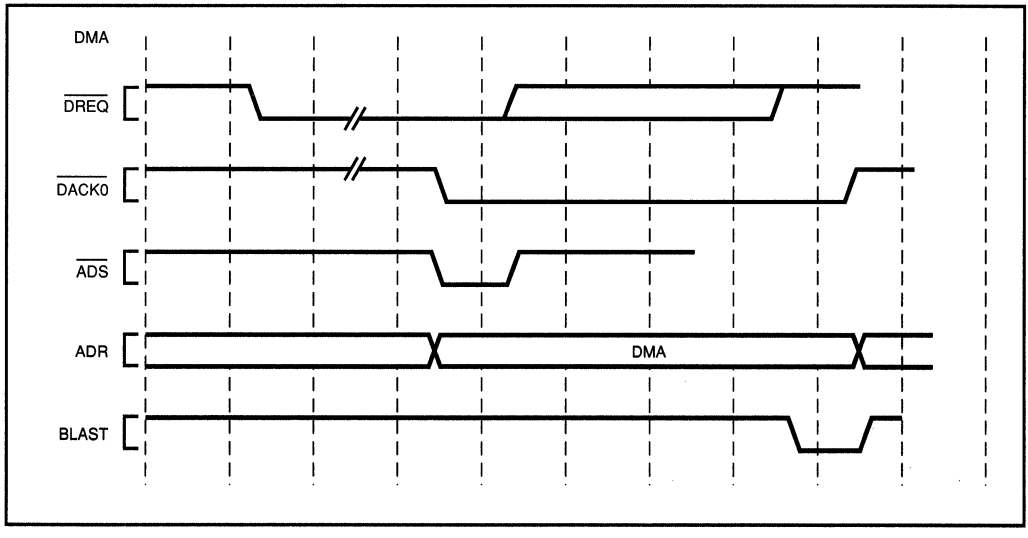


Figure 12-20. DMA Request and Acknowledge Signals

DMA PROGRAMMING FOR REFRESH

The 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 12-21 for chaining description.

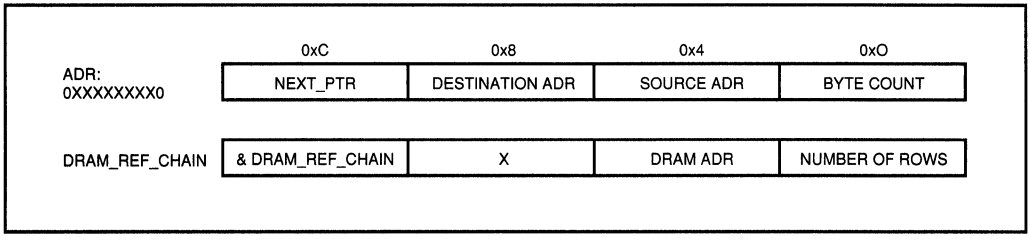


Figure 12-21. DMA Chaining Description

MEMORY READY

The memory-ready input to the 80960CA ($\overline{\text{READY}}$) is used to indicate the completion of a DRAM read or write cycle. The $\overline{\text{READY}}$ signal must be generated by the DRAM controller and must satisfy the setup and hold times specified in the data sheet. If there are multiple memory systems using the $\overline{\text{READY}}$ input, the ready signals from these memory systems must be logically ORed together.

REGION TABLE PROGRAMMING

Region table programming is critical to DRAM operation. The N_{RAD} and N_{WAD} wait states must satisfy the $\overline{\text{RAS}}$, $\overline{\text{CAS}}$ and address-valid times for the DRAM. The N_{RDD} and N_{WDD} times must satisfy the column-address to data-access times. The N_{XDA} time must satisfy the $\overline{\text{RAS}}$ pre-charge time. Figures 12-22, 23 shows typical system waveforms for this design. Note that $\overline{\text{RAS}}$ is not asserted until the end of the address cycle; this delay contributes to the $\overline{\text{RAS}}$ pre-charge time. In some DRAM designs, it may be possible to remove $\overline{\text{RAS}}$ before the access is complete. This is especially true for static column reads and multiple-world access. If $\overline{\text{RAS}}$ can be removed early in the access, the $\overline{\text{RAS}}$ pre-charge can occur during the access.

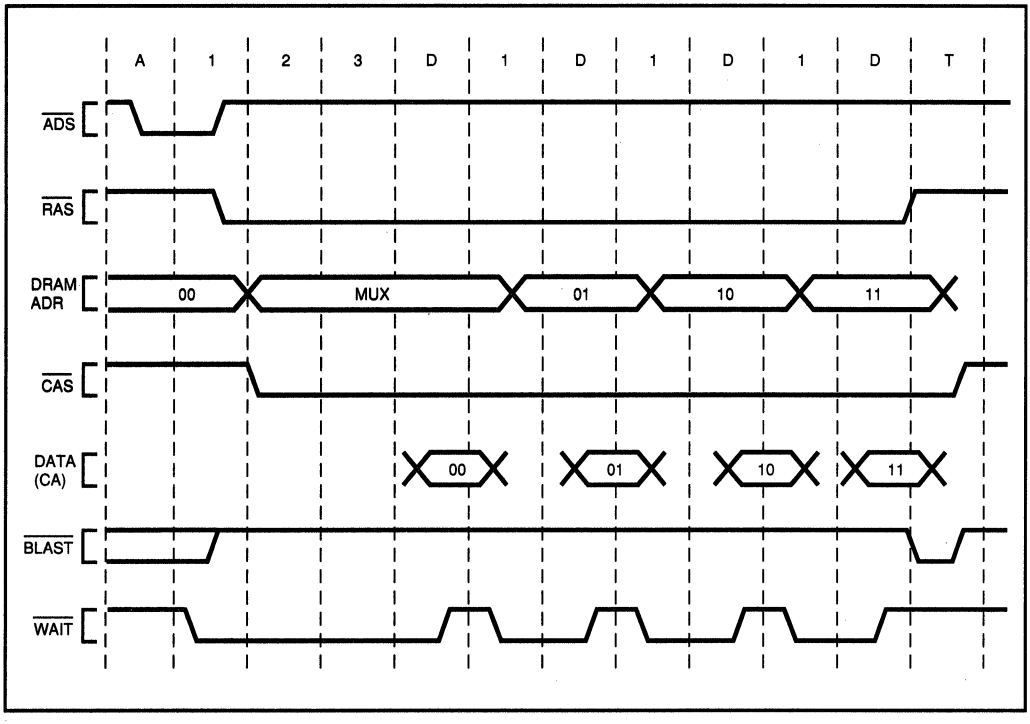


Figure 12-22. DRAM System Read Waveform

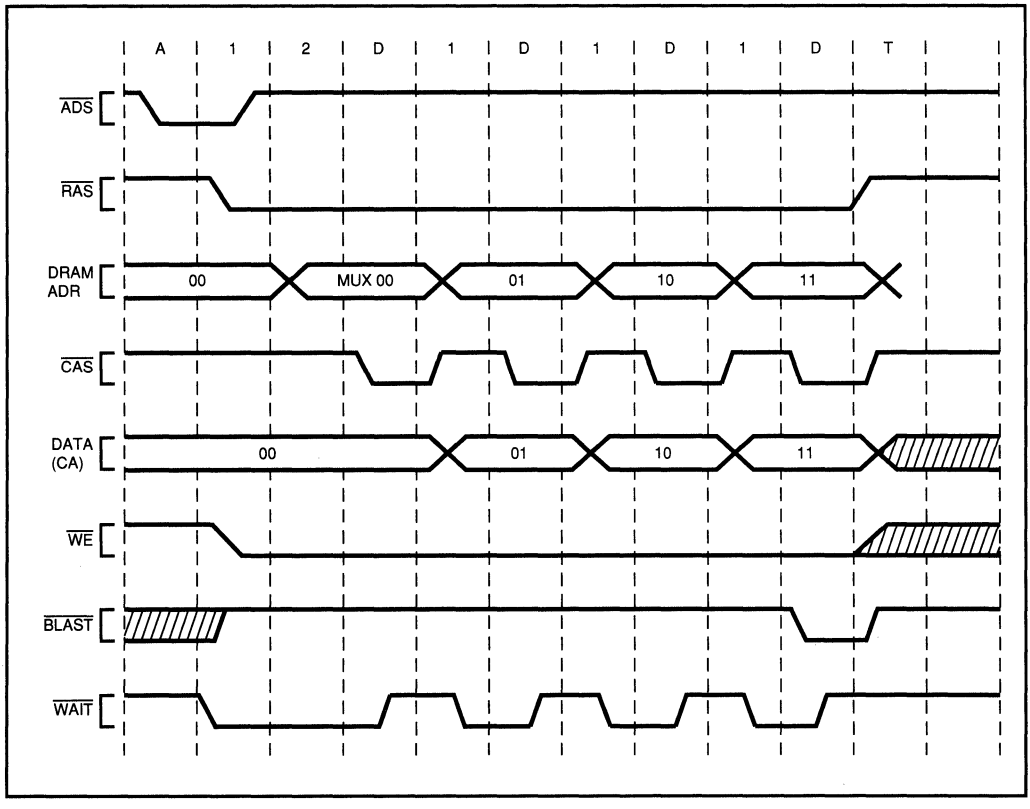


Figure 12-23. DRAM System Write Waveform

DESIGN EXAMPLE: BURST DRAM WITH DISTRIBUTED $\overline{\text{CAS}}$-BEFORE-$\overline{\text{RAS}}$ REFRESH USING $\overline{\text{READY}}$ CONTROL.

This example illustrates a design of a DRAM system 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 DRAM's. The design does not use a DMA channel for refresh. A $\overline{\text{READY}}$ signal 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 the $\overline{\text{RAS}}$ pre-charge time. The internal wait-state generator is not used. The DRAM controller must be designed with information about the processor speed and the DRAM speed.

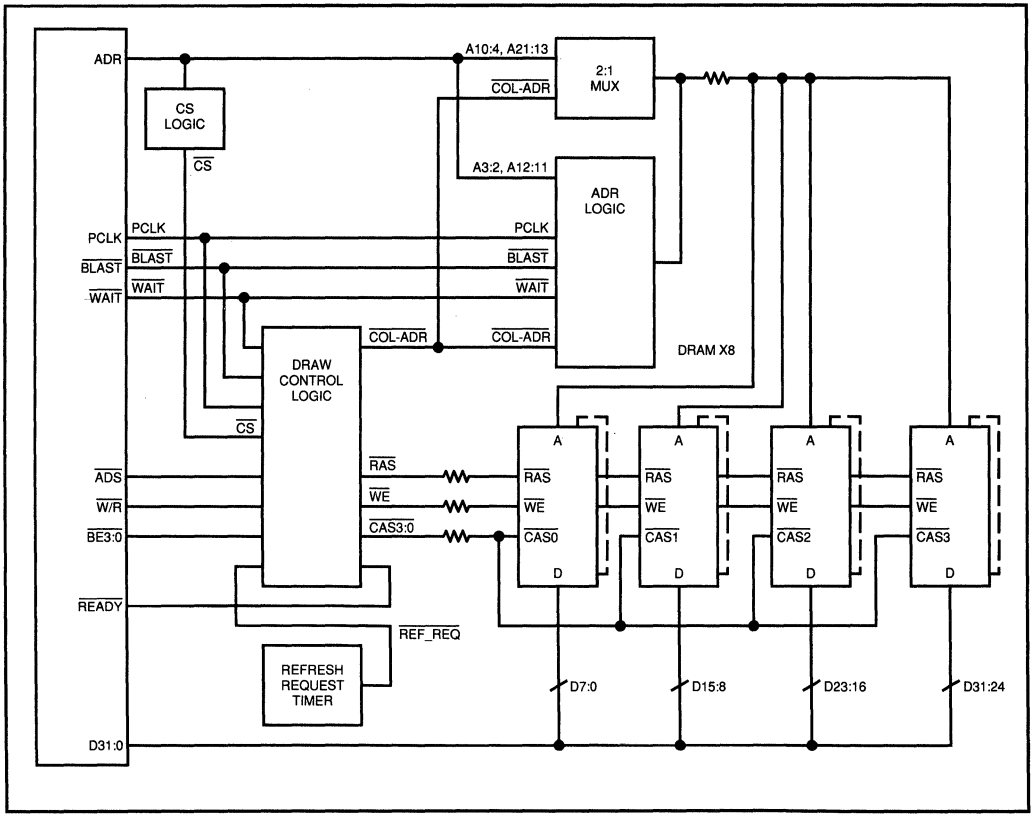


Figure 12-24. Block Diagram

The block diagram for the memory system (Figure 12-24) is similar to the schematic for the previous example, except for the absence of the connection to the DMA controller. The refresh timer indicates it is time to refresh the DRAM.

DRAM CONTROLLER STATE MACHINE

The state machine in Figure 12-25 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. The $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ mode of refresh does not require the bus or any processor intervention; so 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; therefore, all wait-state parameters can be programmed to 0.

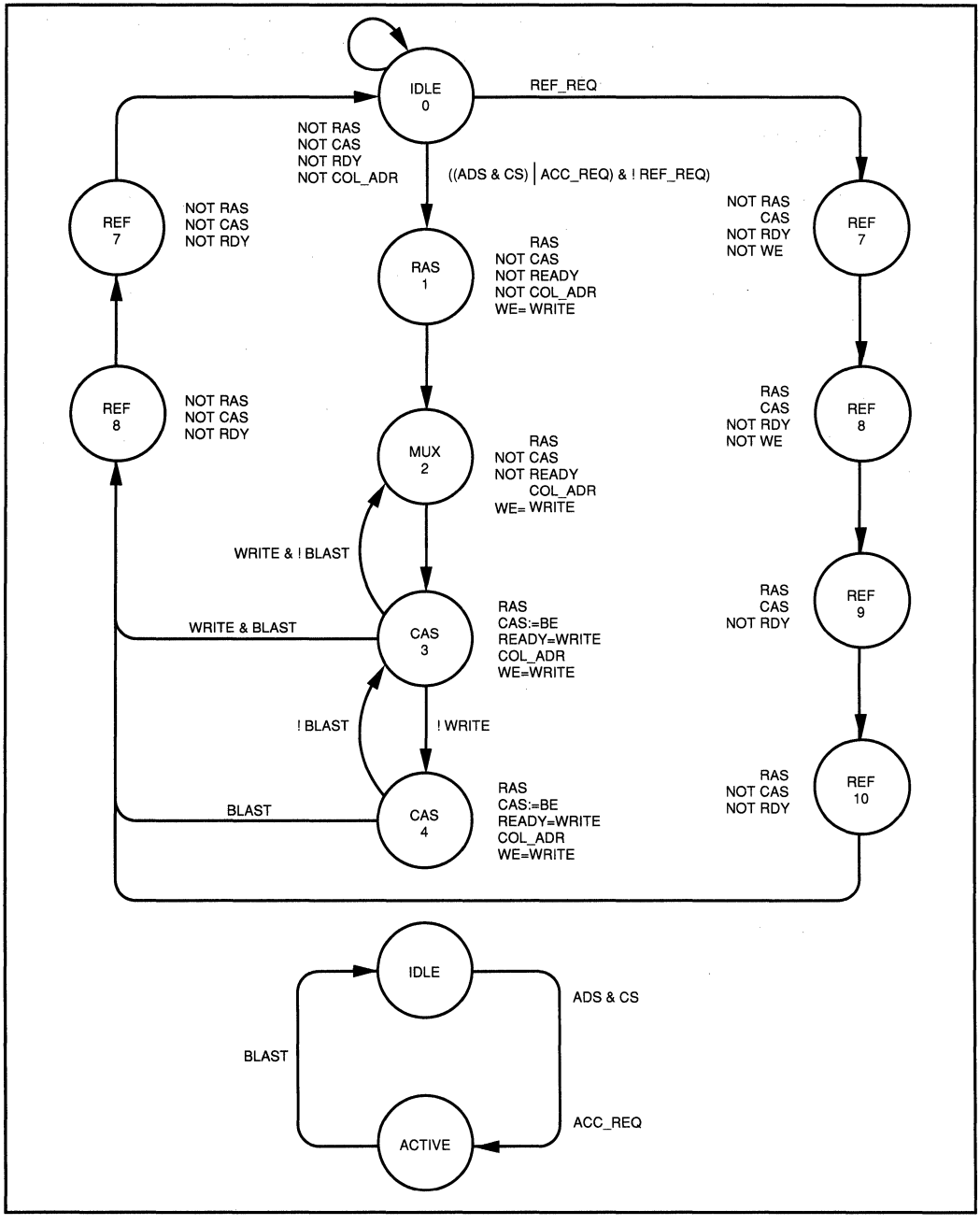


Figure 12-25. DRAM State Machine

The refresh request timer generates the refresh request signal ($\overline{REF\_REQ}$), indicating that it is time to refresh the DRAM. The controller gives preference to refresh requests over access requests. This is done to ensure 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 $\overline{ADS}$ and $\overline{DRAM\_CS}$ are both asserted. ACC_REQ is deasserted when $\overline{BLAST}$ is asserted. It is necessary to latch the access request because the controller could be in a refresh or $\overline{RAS}$ 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:
                !       logical NOT;
                &&      logical AND;
                ||      logical OR
                ==      equality test;
                = value assignment;
                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 nest 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 ready */
        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 Pre-charge*/
        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 Pre-charge*/

STATE_5: /* RAS Pre-charge */
        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 Pre-charge */
        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              = X;
                the next state is STATE_5; /*RAS Pre-Charge*/
```

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 and 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 12-26 shows the access overlap for a burst access.

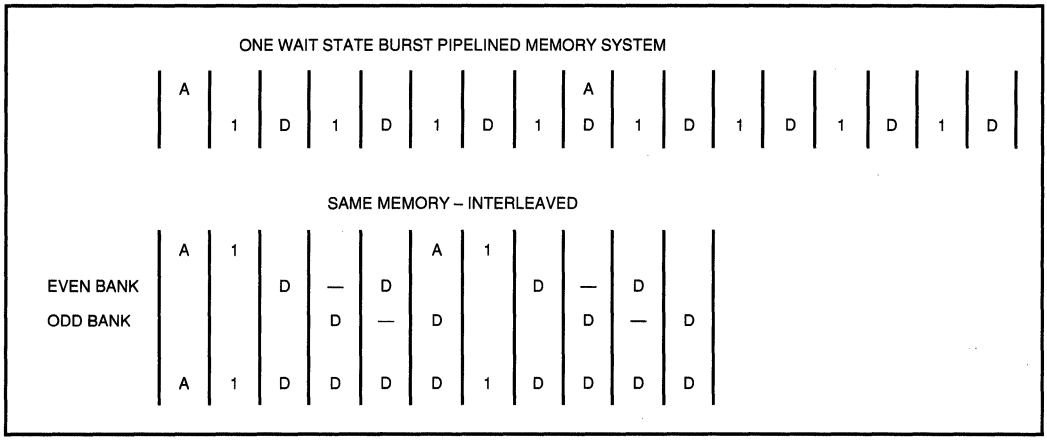


Figure 12-26. 2-Way Interleave Read Access Overlap

Figure 12-27 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; and the low address bit to the memory devices is A3. The A2 address is used to determine which bank (even or odd word) is selected. Figure 12-28 shows the read waveform.

The schematic illustrates a memory system that interleaves the 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 80960CA bus controller queues all access; therefore, write interleaving would not significantly benefit most applications.

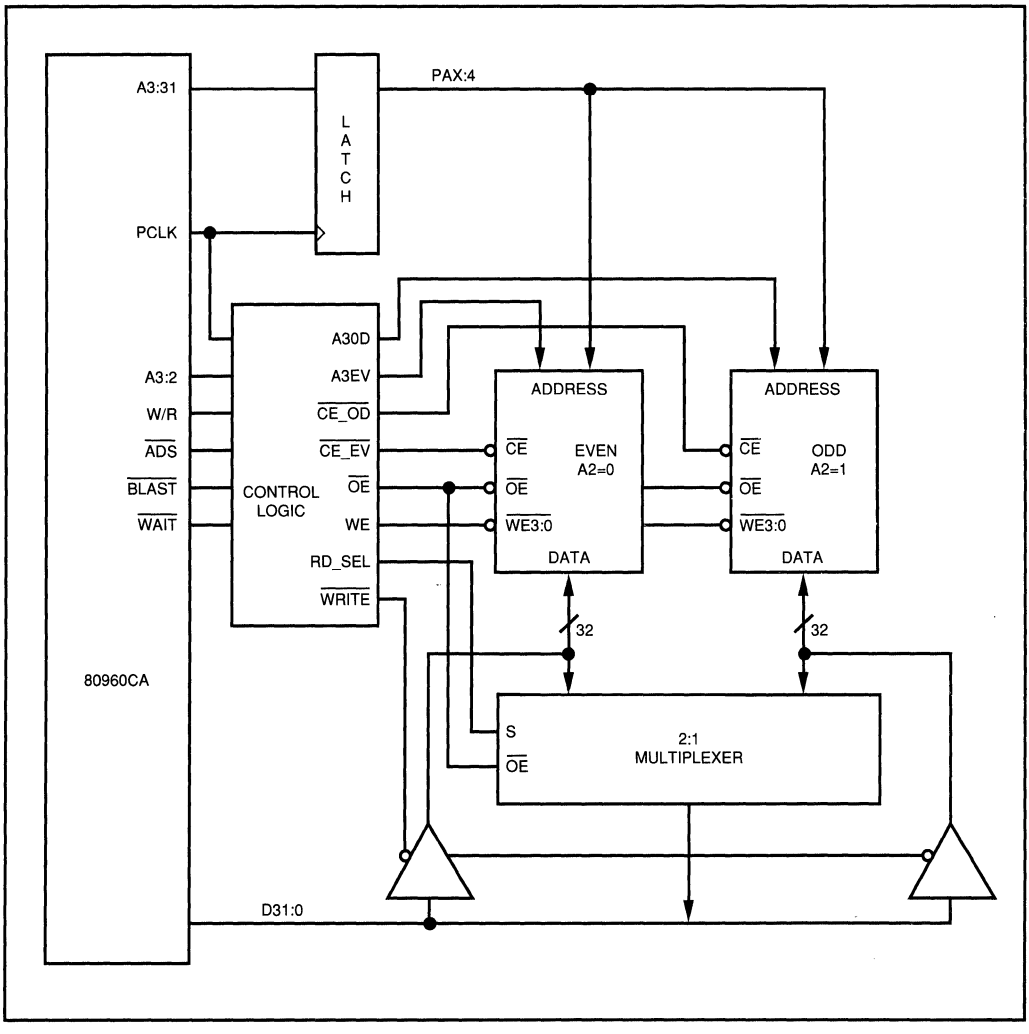


Figure 12-27. 2-Way Interleaved Memory System

Memory interleaving can be applied to SRAM, DRAM, and even EPROM memory systems. Interleaved SRAM and EPROM memory systems overlap the access times for consecutive accesses to improve the memory-system performance. The 80960CA pipelined-read mode can be used on the 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 $\overline{READY}$ control. Interleaved DRAM memory systems can overlap the memory access time and the $\overline{RAS}$ pre-charge time of consecutive accesses.

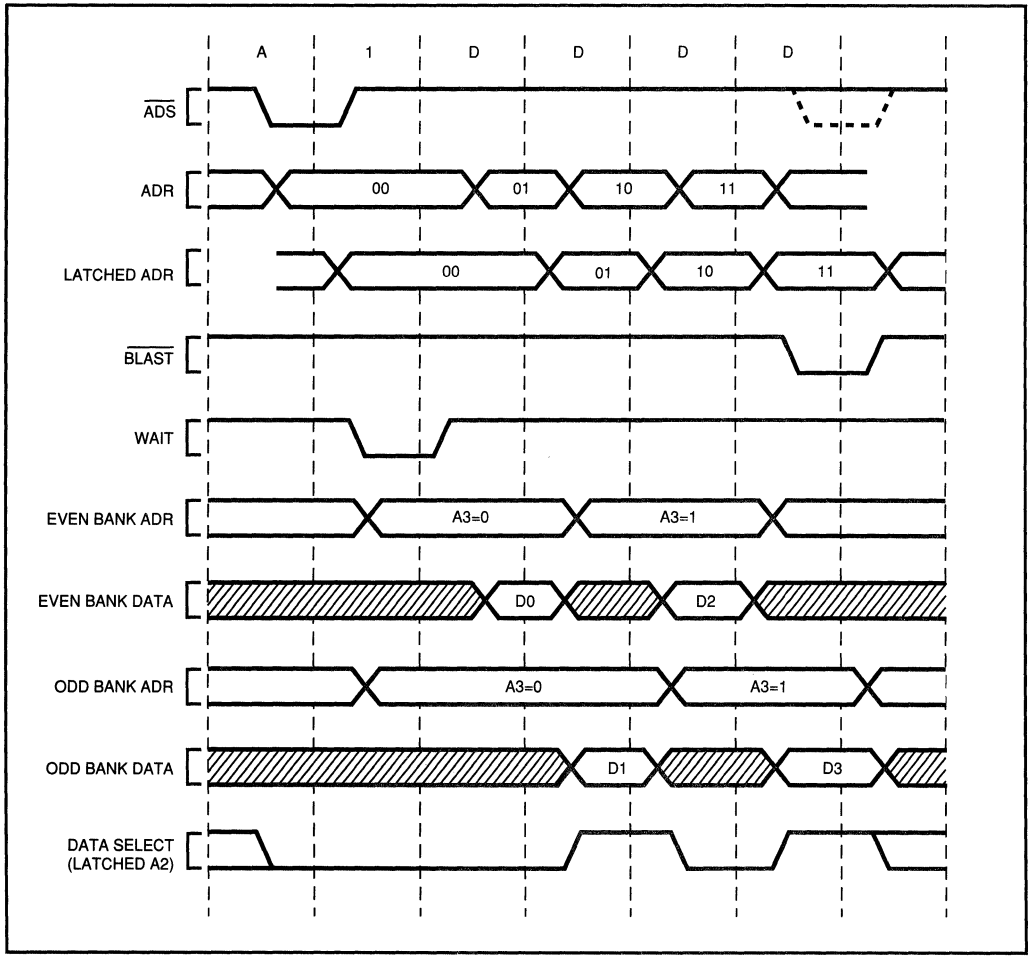


Figure 12-28. 2-Way interleave Read Waveforms

INTERFACING TO SLOW PERIPHERALS USING THE INTERNAL WAIT-STATE GENERATOR

This section illustrates how easy it is to interface slow peripherals to the 80960CA. 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. UART's, Timer/Counters, and 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 bi-directional data bus. The slow peripherals example considers only the memory-mapped interface to these 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 $\overline{RD}$ high to data float time dictates the number of N_{XDA} wait states required. The 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 the chip-select logic; a registered PLD with at least two combinatorial outputs; and a data-transceiver.

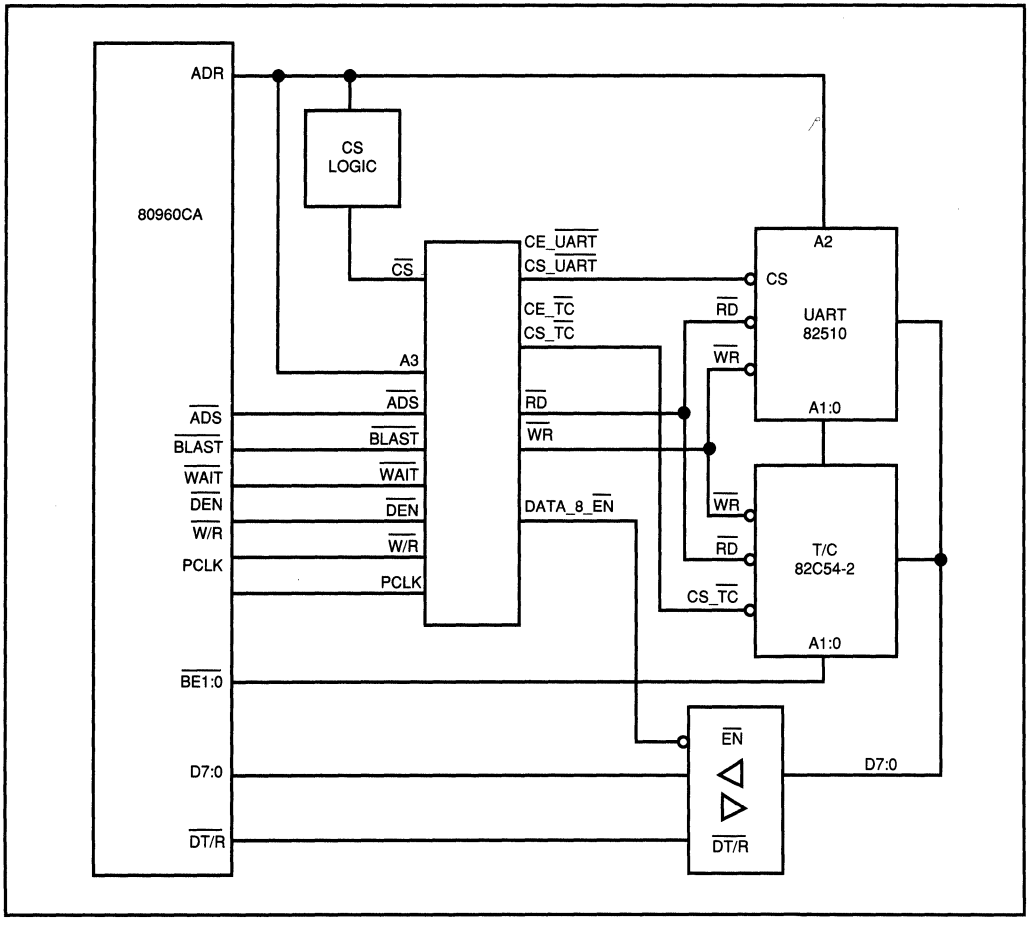


Figure 12-29. 8-bit Interface Schematic

The chip-select logic is the same as in the previous examples. A simple demultiplexer is based only on the address. The PLD that controls the access will qualify this signal with the address strobe ($\overline{ADS}$).

The state-machine PLD generates the chip-enable, read, and write signals for the UART and the Timer/Counter. It also generates the data-enable control for the data-transceiver.
The A3 address signal is used to determine which peripheral is enabled.

The data-transceiver is enabled by the PLD. The transceiver is activated when both the $\overline{CS}$ and $\overline{DEN}$ signals are asserted. The equation is:

$$\overline{DATA\_8\_EN} = \overline{CS} \,||\, \overline{DEN};$$

The direction control of the transceiver is connected directly to the DT/$\overline{R}$ signal of the 80960CA.

The use of the data transceiver is optional; it is used here to reduce the capacitive loading on the data bus. The 80960CA 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.

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 the 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.

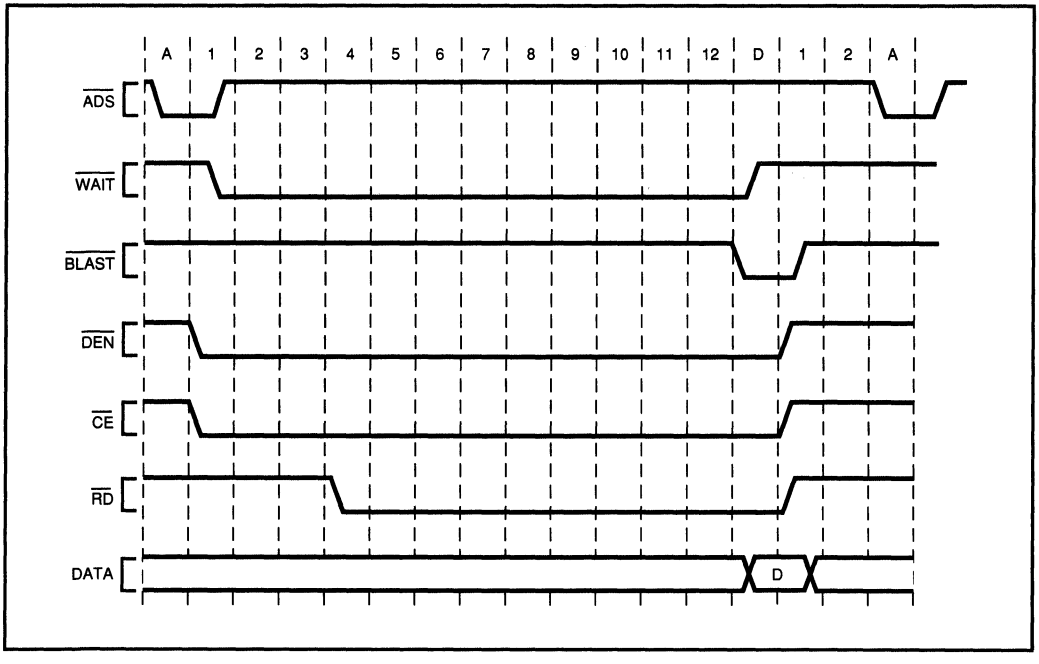


Figure 12-30. Read Waveforms

The data lines are not driven during the N_{XDA} wait states. This requires gating the $\overline{WR}$ signal with the $\overline{WAIT}$ signal, so that $\overline{WR}$ 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.

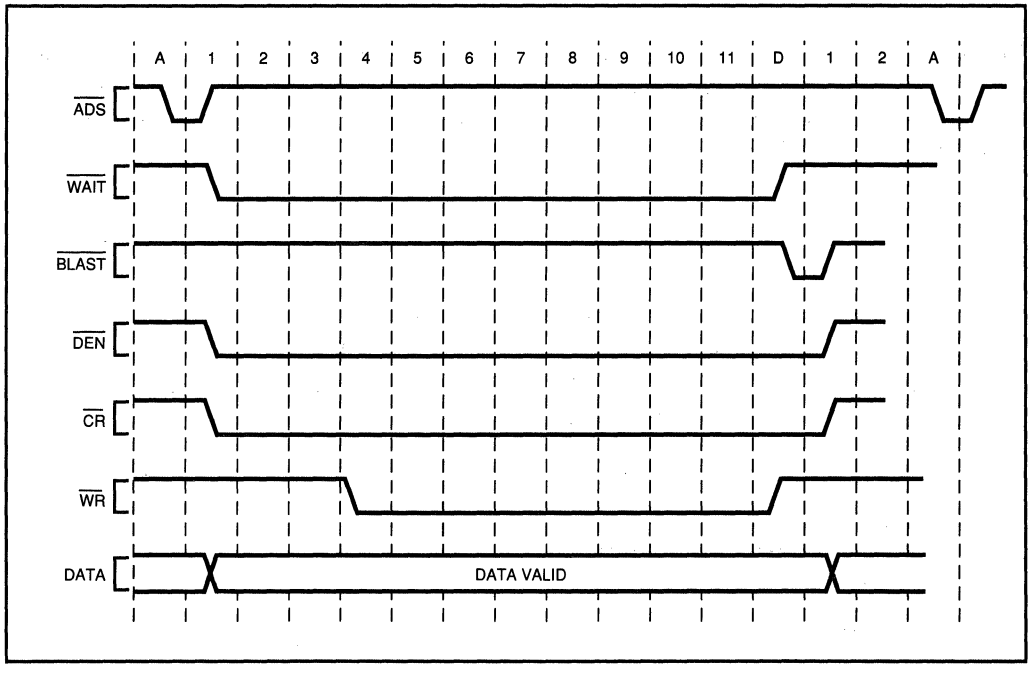


Figure 12-31. Write Waveforms

The state-machine simply delays the read or write signal so that back-to-back commands to the peripheral satisfy the command recovery time of the peripheral. When the write state is entered, the $\overline{WR}$ output of the PLD is a gated version of the $\overline{WAIT}$ signal. This guarantees that the write data hold time of the peripherals is satisfied.

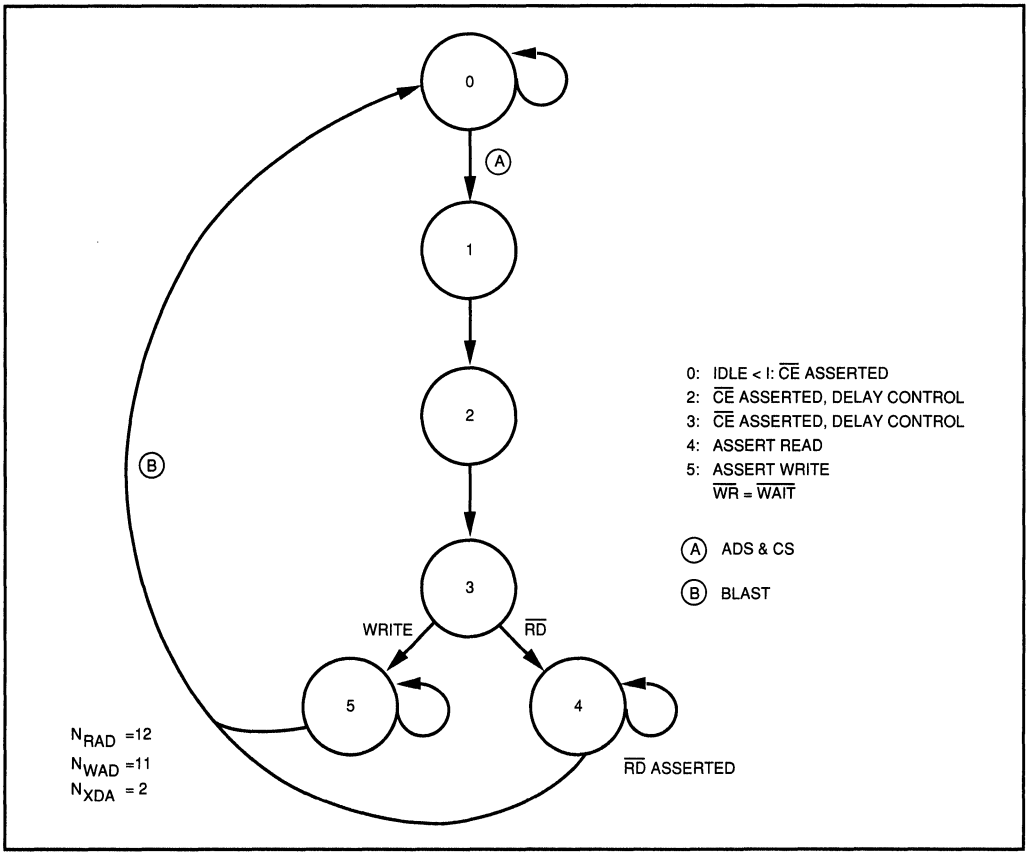


Figure 12-32. State Machine Diagram

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:

```
!       logical NOT;
&&      logical AND;
==      equality test;
= value assignment;
```

```
STATE_0: /*idle */
        CE_UART is not asserted;
        CE_TC is not asserted;
        RD is not asserted;
        WR 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;
        WR 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;
        WR 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;
        WR is not asserted;

        IF
                !W/R /* 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;
        WR is not asserted;

        IF
                BLAST /* Done */
        THEN
                next state is STATE_0;
        ELSE /* write */
                next state is STATE_4;

STATE_5: /* Read asserted to selected peripheral */
        CE_UART = A3;
        CE_TC  = !A3;
        RD is not asserted;
        WR = 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 80960CA. The Burst EPROM provides a synchronous interface to the 80960CA that requires no external logic. These EPROM's offer higher-performance read memory systems than high-speed DRAM's.

Overview of the 27960CA Burst EPROM

The 27960CA 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 80960CA'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 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 $\overline{ADS}$ is asserted. After one or two wait states (depending on the version of the 27960CA), the first data byte is output. The next three consecutive data bytes can be output without any data-to-data wait states. A burst access is terminated on the rising edge of PCLK when $\overline{BLAST}$ is asserted. Timing of the burst EPROM is shown in Figure 12-33.

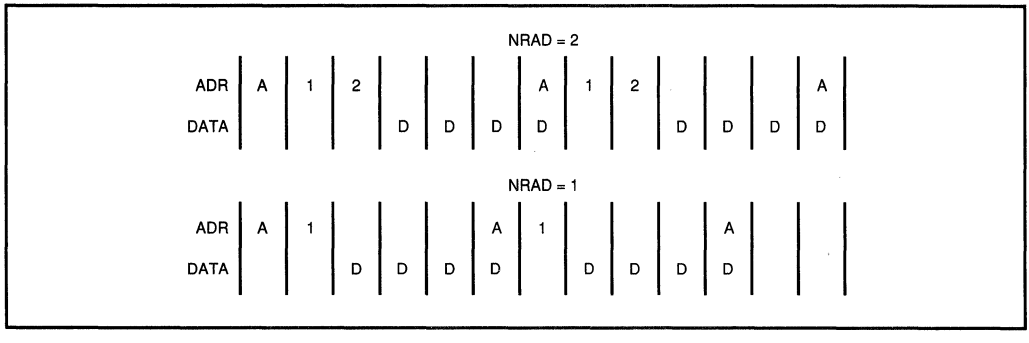


Figure 12-33. Performance of Burst EPROM Pipelined Read

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 is a byte-wide device. Systems can be designed with the 27960CA in 8-, 16- or 32-bit data widths by connecting them to the proper 80960CA data pins. The signal definitions below provide an operation description of the 27960CA. (For programming information, see the 27960CA data sheet.) Definitions of the 27960CA signals are:

CLK Clock (input) - The clock for the EPROM. The address (A16:0) is latched internally on the rising edge of CLK. Data (D7:0) is output with respect to CLK. $\overline{ADS}$, $\overline{CS}$ and $\overline{BLAST}$ are all sampled on the rising edge of CLK. This signal may be connected directly to the 80960CA 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 latches valid addresses in the first clock cycle. An internal address generator increments addresses for subsequent bytes of the burst.

D7:0 8-bit data bus (output) - The data bus drives are enabled when $\overline{CS}$ and $\overline{ADS}$ are asserted during the rising edge of CLK. The data bus drivers are disabled when $\overline{BLAST}$ is asserted and $\overline{ADS}$ is not asserted on the rising edge of CLK.

$\overline{ADS}$ 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 80960CA $\overline{ADS}$ signal.

$\overline{CS}$ Chip select (input) - Master device enable. When asserted (low), data can be read from the device. $\overline{CS}$ enables the state machine and the I/O circuitry. A memory access begins on the first rising edge of CLK that $\overline{ADS}$ and $\overline{CS}$ are asserted. A burst cycle does not terminate if $\overline{CS}$ goes high.

$\overline{BLAST}$ Burst last (input) - Terminates the current burst access. This signal may be connected directly to the 80960CA $\overline{BLAST}$ signal.

$\overline{RESET}$ Asynchronous reset (input) - Resets the EPROM, disables the data outputs. Reset will abort an active access.

Figure 12-36 shows the connections to the 27960CA Burst EPROM.

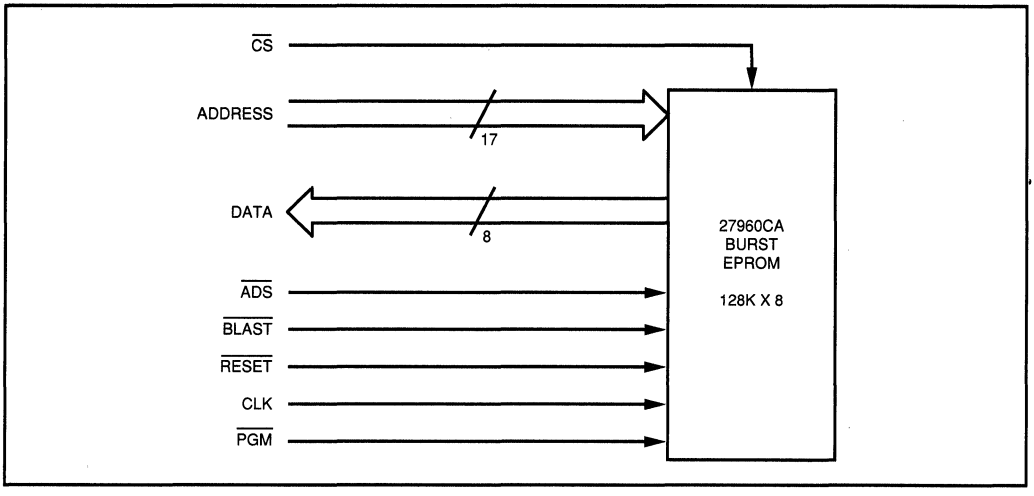


Figure 12-34. The 27960CA EPROM

Interfacing to the 80960CA

The following example demonstrates a 32-bit-wide burst access EPROM interface to the 80960CA. The 27960CA 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). There are no wait states between sequential data during a burst. Figure 12-35 shows a non-buffered, 128K x 32 Burst EPROM system. Chip-select logic is the only external logic that is 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. The 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 12-36.

The wait-state configuration must be programmed into the 80960CA Memory Region Configuration Table. The N_{RAD} wait-states must be programmed to one or two, corresponding to the version of the 27960CA. The N_{RDD} wait states must be programmed to 0. The N_{XDA} wait states should be programmed to 0.

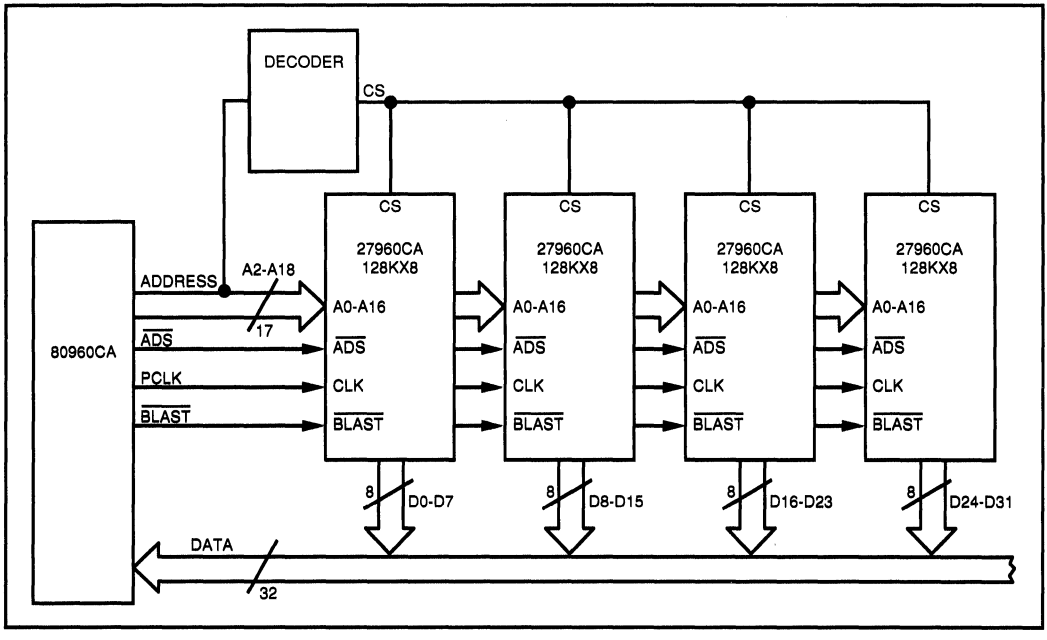


Figure 12-35. 128K X 32 Burst EPROM SYSTEM

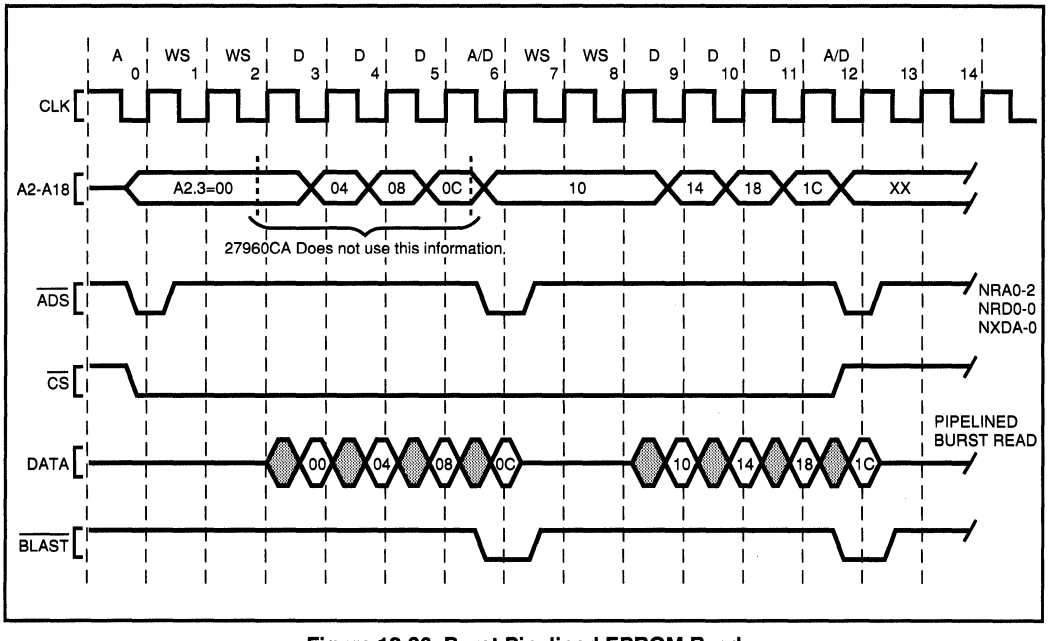


Figure 12-36. Burst Pipelined EPROM Read

12

Booting from the 27960CA

The 80960CA reads 4 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 $\overline{BLAST}$ is asserted (this is illustrated in Figure 12-37); In this way, it is possible to store the IBR in the Burst EPROM.

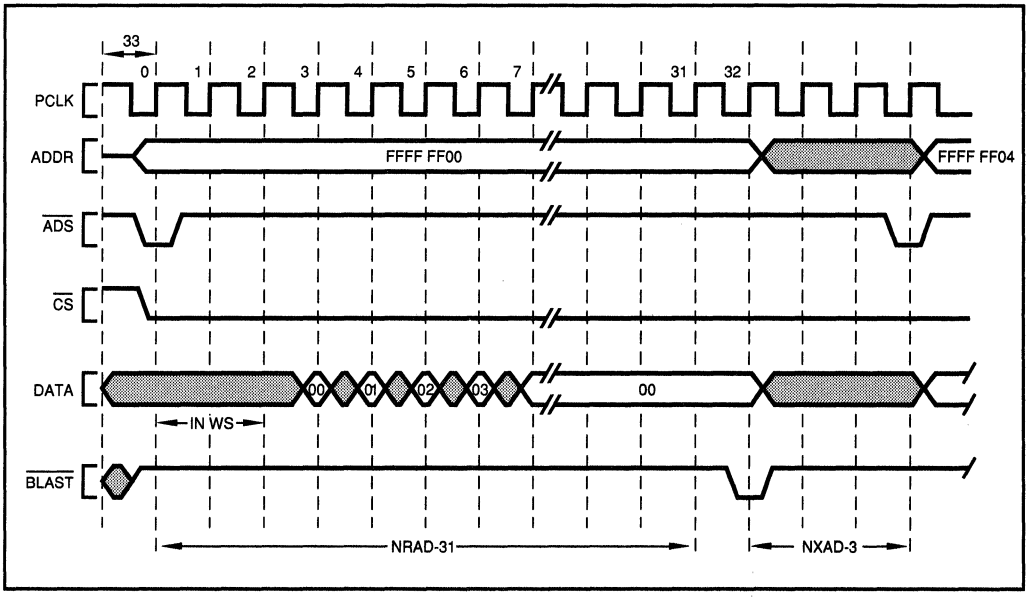


Figure 12-37. Booting from the 27960CA Burst EPROM

INTERFACING TO THE 82596CA LOCAL AREA NETWORK COPROCESSOR

The 82596CA provides a subset of the 80960CA bus interface signals, minimizing bus interface logic. It shares most signals directly with the 80960CA processor. The 82596CA's bus cycles (including burst cycles), bus interface timing, bus arbitration method, and signal definitions are compatible with the 80960CA processor.

Overview of the 82596CA

The 82596CA coprocessor (hereafter referred to generically as the "82596" coprocessor) is a 32-bit multi-tasking LAN coprocessor which implements the carrier-sense, multiple-access and collision-detect (CSMA/CD) link access protocol (Figure 12-38). The coprocessor supports a wide variety of networks. It executes high-level commands, and performs command chaining and inter-processor communication via memory shared with the 80960CA processor. This relieves the processor of all time-critical, local-network control functions.

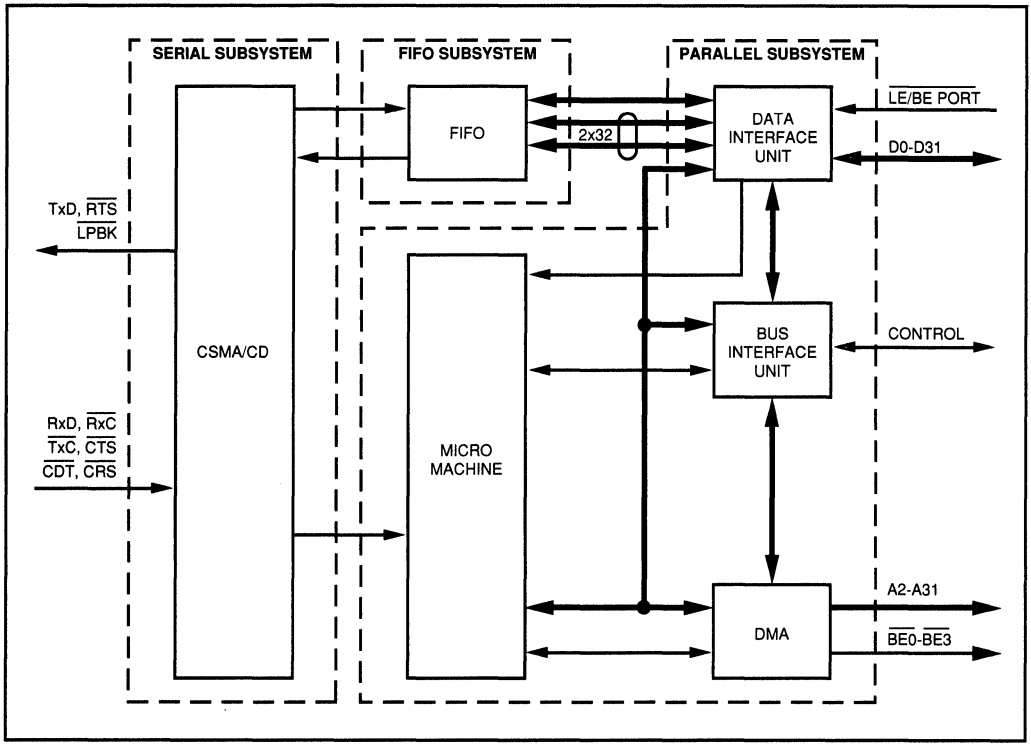


Figure 12-38. 82596 Block Diagram

The coprocessor's 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 80960CA Processor Interface
  - Optimized bus interface to the 80960CA bus
  - Shared 80960CA processor bus signals and memory timing
  - Support for 80960CA 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

The 82596 coprocessor is ideal for interconnect, bridges, and high-performance embedded communication applications. The 82596 bus interface provides a compatible interface to the 80960CA bus, making it is very easy to use. The serial interface is typically 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 use of the processor bus by the coprocessor, 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 82596 coprocessor interacts with the 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 the 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 80960CA.

The coprocessor and the processor communicate through shared memory, as shown in Figure 12-39. The processor and the 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 bi-directional 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 80960CA processor and 82596CA 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 82596CA and 80960CA. This interface is also shown in Figure 12-39.

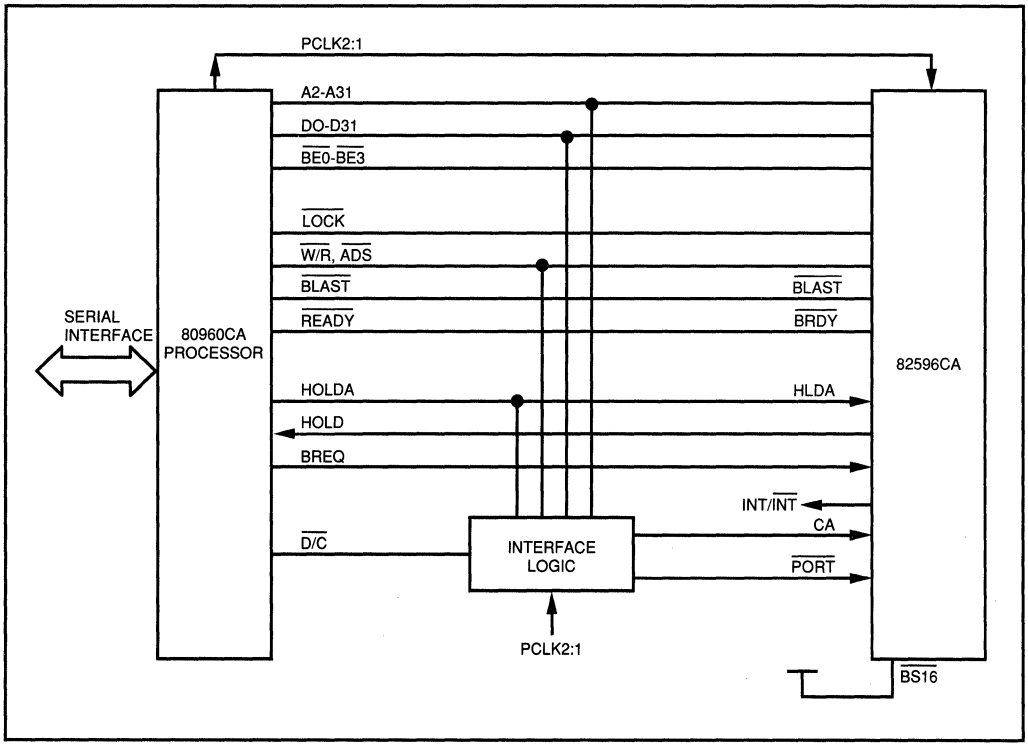


Figure 12-39. 80960CA/82596CA Interface

Table 12-1. Shared 80960CA and 82596CA Bus Output and I/O Signals

Signal	Definition	Type	State of Signal when not owner of the bus
A31-A2	Address	O	float
$\overline{BE3}$-$\overline{BE0}$	Byte Enables	O	float
D31-D0	Data Bus	I/O	float
$\overline{LOCK}$	Bus Lock indicator	O	float
W/$\overline{R}$	Write/Read indicator	O	float
D/$\overline{C}$	Data/Control indicator	O*	float

Note: * The 82596CA does not have the D/$\overline{C}$ signal.

Table 12-2. Shared 80960CA and 82596CA Bus Input Signals

Signal	Definition	Type
$\overline{BRDY}$(82596)/$\overline{READY}$(80960CA)	Ready	I
$\overline{RDY}$(82596)/BTERM(80960CA)	Burst terminate	I

Table 12-3. Arbitration Signals for 80960CA/82596CA Interface

Signal	Definition	Type	80960CA Processor Type	82596CA Comments
HOLD	Hold request	I	O	82596CA always drives
HLDA(82596) HOLDA(80960CA)	Hold acknowledge	O	I	80960CA always drives
BREQ	Bus Request	O	I	80960CA always drives

Arbitration

Bus arbitration between the 80960CA processor and 82596CA is achieved by the hold and hold-acknowledge handshake. The 82596CA requests the bus by asserting HOLD to the 80960CA processor. The processor responds by asserting HOLDA, thus allowing the 82596CA to acquire the bus. The 80960CA's BREQ signal can be used to improve arbitration efficiency. The BREQ signal indicates that there is an internal cycle pending. This signal can be tied directly to the 82596CA BREQ input. When BREQ is asserted, it triggers the 82596CA bus throttle timers. The bus throttle timers cause the 82596 to relinquish the bus in a programmable amount of time. This scheme can help improve arbitration efficiency by reducing the hold and hold-acknowledge handshake delays between the 80960CA processor and the 82596CA.

Interface Logic Requirements

The interface logic between the 80960CA processor and the 82596CA performs the following functions:

- Provides a port that the 80960CA can select, based on an address decode to perform an 82596CA channel attention
- Provides a port that the 80960CA can select, based on an address decoded to perform 82596CA CPU PORT access functions.
- Drives the D/$\overline{C}$ signals when the 82596CA controls the bus; the 82596CA does not have this signal.

82596CA and 80960CA Interface Considerations

The interface between the 82596CA and 80960CA provides compatible bus signals and bus operation; however, there are some differences between the two interfaces that should be considered. They are as follows:

- The 80960CA supports read pipelining, and the 82596CA does not. The 80960CA read pipelining is programmed through a region table, allowing pipelining for a certain memory region. The 80960CA and 82596CA should share a non-pipelined memory region.
- The 82596CA supports dynamic bus sizing for 32- and 16-bit busses. The 80960CA does not support dynamic bus sizing, but supports bus sizing through a programmable region table. Both the 82596CA and 80960CA have a compatible byte-enable encoding scheme for 32-bit busses and should share a 32-bit memory region.
- The 80960CA has a wait-state generator built in. The 82596CA does not have a wait-state generator, and the ready signal needs to be properly returned to the 82596CA.
- The 80960CA provides the signals DT/$\overline{R}$ and $\overline{DEN}$, and the 82596CA does not. If the external hardware uses these signals, then these signals need to be generated when the 82596CA controls the bus.

DMA Controller 13

CHAPTER 13
DMA CONTROLLER

This chapter describes the 80960CA's integrated Direct Memory Access (DMA) Controller. The operation modes, setup, external interface, and implementation of the DMA controller are detailed in this chapter.

OVERVIEW

The DMA controller can manage four independent channels of DMA concurrently. Each channel supports memory to memory transfers where the source and destination can be any combination of internal RAM or external memory. The DMA mechanism provides two different methods for performing DMA transfers: demand-mode transfers (synchronized) and block-mode transfers (non-synchronized). Demand-mode transfers are typically used for transfers between an external device and memory; block-mode transfers are typically used to move blocks of data within memory. In demand mode, external hardware signals for each channel are provided to synchronize DMA transfers with external requesting devices.

To perform a DMA operation the DMA controller uses microcode, multi-process resources of the core, the bus controller, and internal hardware which is dedicated to the DMA controller. Loads and stores are executed in the DMA microcode to execute each DMA transfer. The bus controller, directed by the DMA microcode, handles the data transactions in external memory. The DMA controller hardware is responsible for synchronizing transfers with external devices or memory, providing the programmers interface to the DMA controller and managing the priority for servicing the four DMA channels.

The DMA controller uses multi-process resources designed into the core to enable DMA transfers 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 the DMA processes and the user process enables the user code and up to four DMA's (one per channel) to run at the same time.

The DMA controller is configurable to best exploit the core's processing capabilities and the performance of the external bus. Source and destination data lengths are programmed for each DMA channel. The DMA controller, based on the data length, optimizes the performance of transfers between a source and a destination with different external data-bus widths. A DMA can be programmed for quad-word transfers, taking best advantage of the burst capabilities of the external bus. The DMA controller can also efficiently execute transfers of non-aligned 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 DMA's 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 is used to set up a DMA operation. Addressing, transfer type, and DMA modes are specified in a single issue of this instruction. 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 performance, DMA latency, and processor performance. 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 (synchronized) or a block mode (non-synchronized) channel. Synchronization refers to the mechanism for requesting each DMA transfer with an external device.

Demand-mode DMA's are typically used to move data between memory and an external device. When a channel is configured for demand mode, an external device requests a DMA transfer with a request input ($\overline{\text{DREQ3:0}}$). The DMA controller acknowledges the requesting device with an acknowledge signal ($\overline{\text{DACK3:0}}$) when the requesting device is accessed. A request and acknowledge pin is provided for each DMA channel. These pins are designated with a number suffix which corresponds to the DMA channel number. A DMA transfer is synchronized with either the device or memory which is the source or the destination for the transfer. These transfers are referred to as source or destination synchronized.

Block mode DMA's are typically used to move blocks of data from memory to memory. In block mode, DMA transfers are not requested by an external agent. The DMA operation is initiated by software and continues until terminated or suspended. The DMA is started when the channel enable bit in the DMAC register is set.

SOURCE AND DESTINATION ADDRESSING

When a DMA operation is set up, it is described with a source address, a destination address, and a byte count. For each channel, an address is either held fixed or incremented after each transfer. A fixed address is best suited for addressing memory-mapped peripherals, and an address which increments is suited for the memory side of a transfer. When a channel is set up, address increment or hold is selected separately for the source address and the destination address.

Source address, destination address, and byte count are 32-bit values. This implies that source and destination are byte addressable over the entire address space, and that the length of the DMA operation can be as long as 4 GBytes (2^{32} Bytes). Source address, destination address, and byte count are specified when the **sdma** instruction is executed.

DMA TRANSFERS

The following sections explain the characteristics of a DMA transfer, especially the properties of the transfer which can be affected by channel setup. Intelligent selection of the transfer characteristics works to balance DMA performance and functionality with the performance of the user's program.

The *transfer type* is specified when a channel is set up using the **sdma** instruction. The options for transfer type (Table 13-1) determine if a DMA transfer is performed as a *standard (multiple bus cycle) transfer*, or as a *fly-by (1 bus cycle) transfer*. A standard transfer is made up of multiple bus requests; a fly-by transfer is performed with a single bus request. Fly-by and standard transfers are described in the following sections.

The transfer type options also select the *source/destination data length* for a DMA operation. The data length simply specifies the types of bus requests which are executed to perform each DMA transfer. Combinations of byte, short-word, word, and quad-word load and store requests are issued to perform a transfer.

Table 13-1. Transfer Type Options

Source Data Length (bits)	Destination Data Length (bits)	Transfer Type
Byte (8 bits)	Byte (8 bits)	Standard
Byte (8 bits)	Byte (8 bits)	Fly-by
Byte (8 bits)	Short (16 bits)	Standard
Byte (8 bits)	Word (32 bits)	Standard
Short (16 bits)	Byte (8 bits)	Standard
Short (16 bits)	Short (16 bits)	Standard
Short (16 bits)	Short (16 bits)	Fly-by
Word (32 bits)	Byte (8 bits)	Standard
Word (32 bits)	Short (16 bits)	Standard
Word (32 bits)	Word (32 bits)	Standard
Word (32 bits)	Word (32 bits)	Fly-by
Quad-Word (128 bits)	Quad-Word (128 bits)	Standard
Quad-Word (128 bits)	Quad-Word (128 bits)	Fly-by

Standard Multi-Cycle Transfers

A standard DMA transfer is made up of multiple bus requests. For these transfers, loads from a source address are followed by stores to a destination address. The DMA microcode issues the proper combination of bus requests to execute the transfer. For example, a typical standard DMA transfer could appear as a single byte load request followed by a single byte store request.

For a standard transfer, the 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.

The DMA controller does not perform standard transfers automatically. That is, no mechanism is explicitly provided to protect (or lock) the source or destination from an access by the user program while the source data is internally buffered. However, the user program can poll the progress of a DMA and prevent an access during a transfer. (Monitoring the progress of a DMA is described later in this chapter in the section titled, *DMA Data RAM.*)

Fly-By Single-Cycle Transfers

Fly-by transfers are executed with only a single load or store request. The 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. 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 store.)

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, and the processor ignores the data from the load.

A fly-by DMA in block mode is started by software like any block-mode operation. All block-mode, fly-by transfers are performed as store requests with the acknowledge ($\overline{DACK3:0}$) pin asserted during the bus access. The request pin ($\overline{DREQ3:0}$) is always ignored in block mode. Fly-by block-mode DMA's can be used to implement high-performance memory to memory transfers, where the source and destination addresses are fixed at block boundaries. The acknowledge pin, in this case, must be used in conjunction with external hardware to uniquely address the source and destination for the transfer.

Source/Destination Data Length

The source and destination data length is selected when a channel is set up. The data length determines the size of bus requests which are issued by the DMA microcode. Byte, short-word, or quad-word data lengths are possible. In demand mode, the data length of the requestor is the amount of data moved for each requested transfer. For demand or block mode transfers, the source and destination data lengths are usually selected to match the external data-bus widths for the source and destination. Matching data length to bus width results in the most efficient use of the external bus. For fly-by transfers, a single data length is selected, since in this mode only one load, or store, is issued for each transfer.

Quad-word source and destination data lengths are an option for standard or fly-by DMA's. Quad transfers use the external bus most efficiently when the source or destination memory regions support burst accesses. Using quad-word data lengths may increase bus latency for loads, stores, and instruction fetches generated by the user's program. The user process must wait for each quad-word DMA bus request to complete before the user bus request is serviced. Recall that the service of DMA and user bus requests are alternated by the bus controller.

The data length is the desired, or maximum length, of the DMA-issued bus requests. In cases where source address, destination address, or byte count are non-aligned, requests shorter than the selected length will be issued to align the transfers. (Refer to the section in this chapter titled *Data Alignment.*)

Note: The data length selected for a DMA operation (byte, short-word, word, or quad-word) is not to be confused with the external data-bus width, or other characteristics which are programmed in the memory-region configuration table. The source data length, for example, can be selected as byte and the source address can be in a memory region with a 32-bit data bus. In this case, the DMA microcode simply issues byte load requests to the 32-bit memory. (See *Chapter 10, Bus Controller* for a description of the external bus and the memory region configuration table.)

Assembly and Disassembly

The DMA controller will internally assemble or disassemble data between unequal source and destination data lengths. Byte assembly or disassembly is performed automatically when a channel is set up with unequal source and destination data lengths. Assembly refers to packing of narrow data into wider data. Disassembly refers to unpacking of wide data into narrow data. Assembly and disassembly is supported for all aligned transfers configured with combinations of byte, short-word, and word data lengths. It is not supported for quad-word transfers. Quad-word data lengths must be equal for the source and destination.

Figure 13-1 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 a byte source and a word destination data length is selected for this DMA, the data from four source requests will be buffered before a 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.

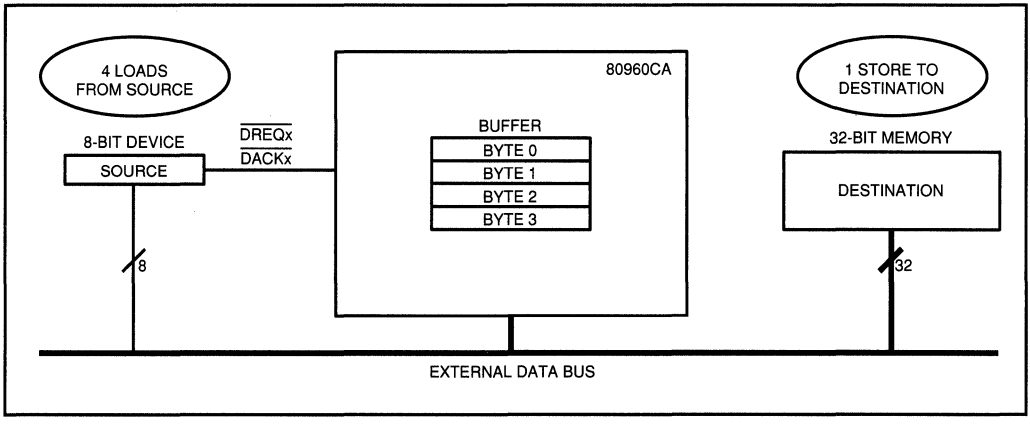


Figure 13-1. Byte to Word Assembly

It is important to understand that the microcode algorithms which perform assembly and disassembly are less efficient than the algorithms which perform transfers between source and destination with equal data lengths. Assembly and disassembly by the DMA controller are provided primarily for making the most efficient use of the external bus, and thus providing maximum throughput for the user's program while the DMA is executing. For example, the system shown in Figure 13-1 functions the same when the source and destination data lengths are both byte-long. In this case, each transfer is performed with a byte load followed by a byte store. The DMA throughput is increased; however, the DMA makes more bus requests to transfer the same amount of data.

Specifying the source and destination transfer lengths and assembly and disassembly by the DMA controller introduce a number of variables which affect performance of a DMA and the user's system in general. Bus loading, bus latency, DMA performance, and performance of the user process are all a function of these variables.

Data Alignment

The DMA Controller performs operations on source and destination data which are not aligned in memory. In other words, the source and destination addresses for a DMA do not need to be aligned to a module memory boundary, or aligned with respect to one another. This section describes the utility as well as the trade-offs and limitation of data alignment by the DMA controller.

In DMA's where both source and destination address is incremented (typically block-mode DMA's), there are no alignment restrictions except for the quad-word transfer lengths and for fly-by transfers. For quad-word transfers, source, destination, and byte count must be aligned to quad-word boundaries. For fly-by transfers, the fly-by address and byte count must be aligned. Alignment for incrementing source and destination address is described in detail in Table 13-2. An incrementing address is designated in this table by regular type, and a fixed address is underlined.

If a DMA source or destination is at a fixed address (typical for demand mode), the fixed address must be aligned. The byte count, in this case must also be a multiple of the data length selected for the fixed address. These constraints simply mean that the device at the fixed address always receives a complete data transfer. Demand-mode alignment is described in detail in Table 13-2.

Aligned DMA's, in general, perform better than non-aligned. Many non-aligned cases execute byte-long bus requests to load or store data at the non-aligned address. The DMA controller, in these cases, reverts to a different transfer mode to perform the DMA. For example, a transfer with a non-aligned 16-bit source and an 8-bit destination reverts to an 8-bit to 8-bit transfer type to perform the DMA. Table 13-2 shows the bus requests which will be executed for every possible case of data alignment.

Table 13-2. Data Alignment

Transfer Type Source/Dest	Source Address(1) (4 LSB)	Dest. Address(1) (4 LSB)	Byte Count (4 LSB)	Load Requests	Store Requests
8/8 Standard	XXXX	XXXX	XXXX (≥1)	Byte	Byte
	XXXX	XXXX	XXXX (≥1)	Byte	Byte
	XXXX	XXXX	XXXX (≥1)	Byte	Byte
	XXXX	XXXX	XXXX (≥1)	Byte	Byte
8/16 Standard	XXXX	XXX0	XXXX (≥1)	Byte	Short
	XXXX	XXX0	XXXX (≥1)	Byte	Short
	XXXX	XXX0	XXX0 (≥2)	Byte	Short
	XXXX	XXX0	XXX0 (≥2)	Byte	Short
	XXXX	XXX1	XXXX (≥1)	Byte	Byte
	XXXX	XXX1	XXXX (≥1)	Byte	Byte
8/32 Standard (optimized alignment)	XXXX	XX00	XXXX (≥1)	Byte	Word
	XXXX	XX00	XXXX (≥1)	Byte	Word
	XXXX	XX00	XX00 (≥4)	Byte	Word
	XXXX	XX00	XX00 (≥4)	Byte	Word
	XXXX	XX01	XXXX (≥1)	Byte	Word
	XXXX	XX01	XXXX (≥1)	Byte	Word
	XXXX	XX10	XXXX (≥1)	Byte	Word
	XXXX	XX10	XXXX (≥1)	Byte	Word
	XXXX	XX11	XXXX (≥1)	Byte	Word
	XXXX	XX11	XXXX (≥1)	Byte	Word
16/8 Standard	XXX0	XXXX	XXXX (≥1)	Short	Byte
	XXX0	XXXX	XXX0 (≥2)	Short	Byte
	XXX0	XXXX	XXXX (≥1)	Short	Byte
	XXX0	XXXX	XXX0 (≥2)	Short	Byte
	XXX1	XXXX	XXXX (≥1)	Byte	Byte
	XXX1	XXXX	XXXX (≥1)	Byte	Byte
16/16 Standard	XXX0	XXX0	XXXX (≥1)	Short	Short
	XXX0	XXX0	XXX0 (≥2)	Short	Short
	XXX0	XXX0	XXX0 (≥2)	Short	Short
	XXX0	XXX0	XXX0 (≥2)	Short	Short
	XXX0	XXX1	XXXX (≥1)	Short	Byte
	XXX0	XXX1	XXX0 (≥2)	Short	Byte
	XXX1	XXX0	XXXX (≥1)	Byte	Short
	XXX1	XXX0	XXX0 (≥2)	Byte	Short
	XXX1	XXX1	XXXX (≥1)	Short	Short

(cont.)

Table 13-2. Data Alignment (cont.)

Transfer Type Source/Dest	Source Address(1) (4 LSB)	Dest. Address(1) (4 LSB)	Byte Count (4LSB)	Load Requests	Store Requests
16/32 Standard	XXX0	XX00	XXXX (≥1)	Short	Word
	XXX0	XX00	XXX0 (≥2)	Short	Word
	XXX0	XX00	XX00 (≥4)	Short	Word
	XXX0	XX00	XX00 (≥4)	Short	Word
	XXX0	XX01	XXXX (≥1)	Short	Byte
	XXX0	XX01	XXX0 (≥2)	Short	Byte
	XXX0	XX10	XXXX (≥1)	Short	Short
	XXX0	XX10	XXX0 (≥2)	Short	Short
	XXX0	XX11	XXXX (≥1)	Short	Byte
	XXX0	XX11	XXX0 (≥2)	Short	Byte
	XXX1	XX00	XXXX (≥1)	Byte	Word
	XXX1	XX00	XX00 (≥4)	Byte	Word
	XXX1	XX01	XXXX (≥1)	Byte	Byte
	XXX1	XX10	XXXX (≥1)	Byte	Short
	XXX1	XX11	XXXX (≥1)	Byte	Byte
32/8 Standard (optimized alignment)	XX00	XXXX	XXXX (≥1)	Word	Byte
	XX00	XXXX	XX00 (≥4)	Word	Byte
	XX00	XXXX	XXXX (≥1)	Word	Byte
	XX00	XXXX	XX00 (≥4)	Word	Byte
	XX01	XXXX	XXXX (≥1)	Word	Byte
	XX01	XXXX	XXXX (≥1)	Word	Byte
	XX10	XXXX	XXXX (≥1)	Word	Byte
	XX10	XXXX	XXXX (≥1)	Word	Byte
	XX11	XXXX	XXXX (≥1)	Word	Byte
	XX11	XXXX	XXXX (≥1)	Word	Byte

(cont.)

Notes: 1) Fixed addresses are underlined; incrementing addresses are designated by regular type.
2) For fly-by transfers, either a load or store request is issued, not both.

Table 13-2. Data Alignment (cont.)

Transfer Type Source/Dest	Source Address(1) (4 LSB)	Dest. Address(1) (4 LSB)	Byte Count (4LSB)	Load Requests	Store Requests
32/16 Standard	XX00	XXX0	XXXX (≥1)	Word	Short
	XX00	XXX0	XX00 (≥4)	Word	Short
	XX00	XXX0	XXX0 (≥2)	Word	Short
	XX00	XXX0	XX00 (≥4)	Word	Short
	XX00	XXX1	XXXX (≥1)	Word	Byte
	XX00	XXX1	XX00 (≥4)	Word	Byte
	XX01	XXX0	XXXX (≥1)	Byte	Short
	XX01	XXX0	XXX0 (≥2)	Byte	Short
	XX01	XXX1	XXXX (≥1)	Byte	Byte
	XX10	XXX0	XXXX (≥1)	Short	Short
	XX10	XXX0	XXX0 (≥2)	Short	Short
	XX10	XXX1	XXXX (≥1)	Short	Byte
	XX11	XXX0	XXXX (≥1)	Byte	Short
	XX11	XXX0	XXX0 (≥2)	Byte	Short
	XX11	XXX1	XXXX (≥1)	Byte	Byte

(cont.)

Table 13-2. Data Alignment (cont.)

Transfer Type Source/Dest	Source Address(1) (4 LSB)	Dest. Address(1) (4 LSB)	Byte Count (4LSB)	Load Requests	Store Requests
32/32 Standard (optimized alignment)	XX00	XX00	XXXX (≥1)	Word	Word
	XX00	XX00	XX00 (≥4)	Word	Word
	XX00	XX00	XX00 (≥4)	Word	Word
	XX00	XX00	XX00 (≥4)	Word	Word
	XX00	XX01	XXXX (≥1)	Word	Word
	XX00	XX01	XX00 (≥4)	Word	Word
	XX00	XX10	XXXX (≥1)	Word	Word
	XX00	XX10	XX00 (≥4)	Word	Word
	XX00	XX11	XXXX (≥1)	Word	Word
	XX00	XX11	XX00 (≥4)	Word	Word
	XX01	XX00	XXXX (≥1)	Word	Word
	XX01	XX00	XX00 (≥4)	Word	Word
	XX01	XX01	XXXX (≥1)	Word	Word
	XX01	XX10	XXXX (≥1)	Word	Word
	XX01	XX11	XXXX (≥1)	Word	Word
	XX10	XX00	XXXX (≥1)	Word	Word
	XX10	XX00	XX00 (≥4)	Word	Word
	XX10	XX01	XXXX (≥1)	Word	Word
	XX10	XX10	XXXX (≥1)	Word	Word
	XX10	XX11	XXXX (≥1)	Word	Word
	XX11	XX00	XXXX (≥1)	Word	Word
	XX11	XX00	XX00 (≥4)	Word	Word
	XX11	XX01	XXXX (≥1)	Word	Word
	XX11	XX10	XXXX (≥1)	Word	Word
	XX11	XX11	XXXX (≥1)	Word	Word
128/128 Standard	0000	0000	0000 (≥16)	Quad	Quad
	0000	0000	0000 (≥16)	Quad	Quad
	0000	0000	0000 (≥16)	Quad	Quad
	0000	0000	0000 (≥16)	Quad	Quad
8/8 Fly-by(2)	XXXX	Not used	XXXX (≥1)	Byte	Byte
	XXXX	Not used	XXXX (≥1)	Byte	Byte
16/16 Fly-by(2)	XXX0	Not used	XXX0 (≥2)	Short	Short
	XXX0	Not used	XXX0 (≥2)	Short	Short
32/32 Fly-by(2)	XX00	Not used	XX00 (≥4)	Word	Word
	XX00	Not used	XX00 (≥4)	Word	Word
128/128 Fly-by(2)	0000	Not used	0000 (≥16)	Quad	Quad
	0000	Not used	0000 (≥16)	Quad	Quad

Notes: 1) Fixed addresses are underlined; incrementing addresses are designated by regular type.
2) For fly-by transfers, either a load or store request is issued, not both.

Several of the commonly-used transfer modes are optimized to perform non-aligned DMA's almost as well as aligned cases. This is done by performing byte transfers until alignment is enforced. At this time, aligned source and destination requests are executed. At the end of the transfer, the DMA may revert to byte transfers to complete the DMA. This alignment mechanism is shown in Figure 13-2. The overhead for the alignment occurs at the beginning and end of the DMA operation and, depending on the byte count for the DMA, may be negligible. These optimized cases are noted in Table 13-2.

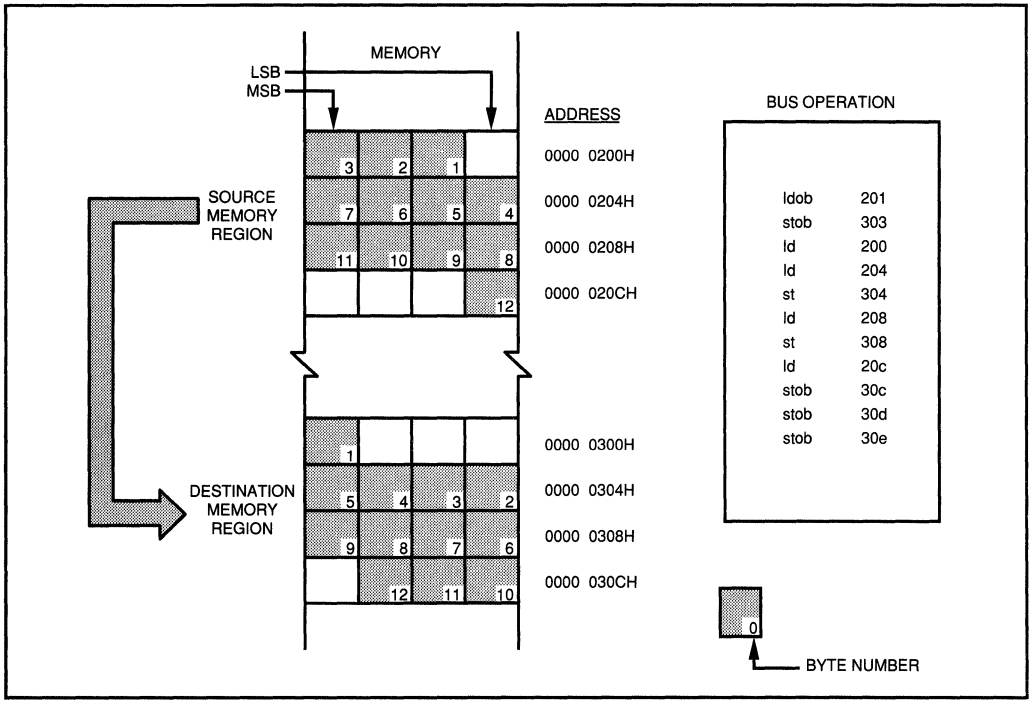


Figure 13-2. Optimization of a Non-aligned DMA

DATA CHAINING

Data Chaining is provided to simplify several complex data manipulations which are commonly performed by DMA controllers. A description of chained DMA operations follows.

Chaining Descriptors

When a channel is configured for chaining mode, chaining descriptors in memory specify the source and destination addressing for the DMA. A chaining descriptor (Figure 13-3) supplies the source address (SA), destination address (DA), and byte count (BC) for a portion of a chained DMA operation. These three values describe a single source and destination *chaining buffer*.

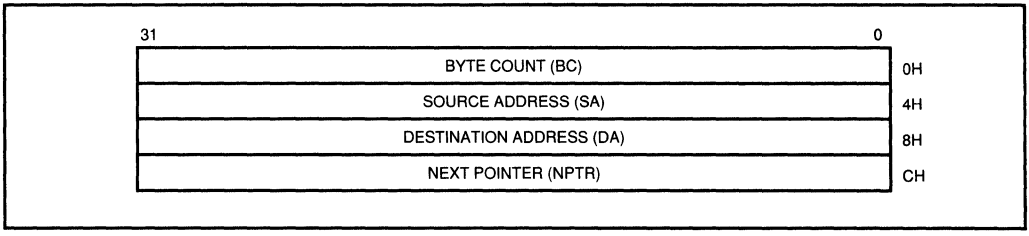


Figure 13-3. DMA Chaining Descriptor

A chained DMA is made up of one or more buffer transfers. A linked set of chaining descriptors describes the full operation. The next pointer (NPTR) field in the chaining descriptor points to descriptors which describe successive buffer transfers. When an NPTR of 0 (null pointer) is encountered, the DMA operation ends. Repetative sequences of DMA transfers are easily implemented by having the NPTR field point back to its own descriptor or to a previous descriptor.

A chained DMA is started by specifying a pointer to the first chaining descriptor. The pointer is specified when a channel is configured in chaining mode.

A chained DMA channel is source-chained, destination-chained, or both source- and destination-chained. The chaining mode determines whether or not the source or destination addresses are read from a new descriptor for each buffer transfer. For example, if a channel is configured for source chaining only (Figure 13-4), 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.)

Each buffer transfer is handled by the DMA controller as if it were a single non-chained DMA. The 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 non-aligned DMA's.

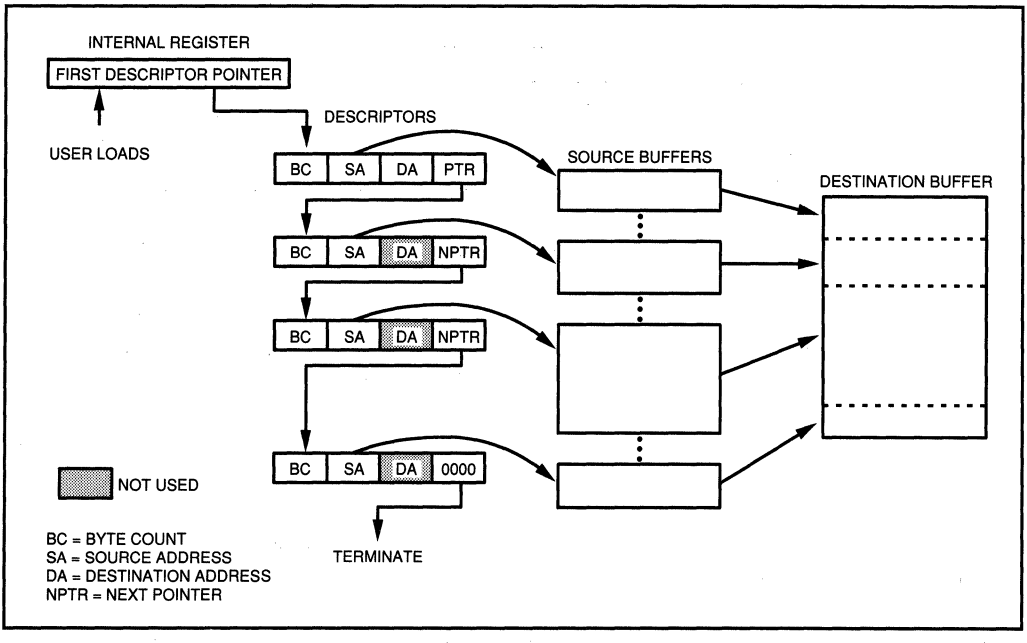


Figure 13-4. Source Chaining

Depending on the configuration of a DMA channel and the chaining mode selected, certain fields in the chaining descriptor are currently 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, and 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, and 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, and the DA field is ignored.

When descriptors are read from external memory, bus latency and memory speed affect the *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.

Channel Wait and Interrupt on Buffer Complete

Two mechanisms are provided which enable user code to synchronize with a buffer transfer. These mechanism are the channel wait and the interrupt on buffer-complete options. Each of these options is separately enabled or disabled when a channel is configured.

The channel-wait function provides a way for the user to setup or modify chaining descriptors while a chained DMA is in progress. When the channel-wait function is enabled, a channel-wait bit in the DMA command register (DMAC) is set when a descriptor is read by the DMA controller. The channel wait bit must be reset by user code before the DMA controller reads the next descriptor.

An interrupt can be generated when a buffer transfer has completed. An interrupt procedure can modify or create a chaining descriptor, and then reset the channel-wait bit in DMAC to continue the chained DMA operation. The interrupt is generated when the end of a buffer is reached.

TERMINATING OR SUSPENDING A DMA

A DMA operation is normally terminated when an $\overline{\text{EOP3:0}}$ pin (programmed as an input) becomes active for a channel, or when the byte count for the DMA reaches 0. These two conditions are described below.

The End of Process/Terminal Count pin $\overline{\text{EOP/TC3:0}}$, pin can be programmed either as an input (EOP3:0) or as an output (TC3:0). When configured as an output, the pin is driven for one clock cycle after a DMA is terminated because of byte count reaching 0. When configured as an input, asserting $\overline{\text{EOP3:0}}$ causes the DMA to terminate regardless of the progress of the DMA. The EOP3:0 pin must be asserted for a minimum of two clock cycles to be detected.

When $\overline{\text{EOP3:0}}$ is detected, the DMA controller performs the following actions:

- Pending requests for the channel are completed, and the DMA is terminated.
- The channel-done bit in DMAC is set.
- The channel interrupt pending bit in the Interrupt Pending Register (IPND) is set.
- The channel-active bit in DMAC is reset.

A DMA operation ends when the internal byte count for the DMA decrements to 0. When byte count reaches 0, the DMA operation terminates and the DMA controller performs the following actions:

- The channel done bit in DMAC is set.
- The channel terminal count bit in DMAC is set.
- The channel interrupt pending bit in the Interrupt Pending Register (IPND) is set.
- The $\overline{\text{EOP}}$/TC3:0 pin is driven for one clock cycle if this pin is configured as an output.
- The channel active bit in DMAC is reset.

A user's program code can distinguish between the two termination mechanisms described by reading the value of the channel terminal-count bit in DMAC. If the terminal count bit along with the done bit for a channel is set, the DMA has ended because of byte count reaching 0. If only the done bit is set for the channel, the DMA has ended because of an active $\overline{\text{EOP}}$x input.

When data chaining is selected, the $\overline{\text{EOP}}$x input terminates the DMA operation as described above with one exception. If both source and destination buffers are chained, asserting $\overline{\text{EOP}}$x causes only the current buffer to terminate. The channel interrupt pending bit is set in the IPND register, and chaining continues with the next buffer transfer, or terminates if a null pointer is reached.

A DMA operation can be suspended at any time by clearing the channel-enable bit in DMAC. When a DMA is suspended, pending requests for the channel are serviced before the channel is suspended. The 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.

All DMA's can be suspended when an interrupt is serviced to reduce the interrupt's latency. This option is selected in the Interrupt Control Register (ICON). When the option is selected, all DMA operations are suspended during the time that the interrupt context switch is processed by the core. DMA's are restarted before the first instruction of the interrupt procedure is encountered. This option effectively 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 channel enable field in the DMAC register. (See *DMA Command Register* in this chapter.) The interrupt procedure should re-enable all suspended channels before returning.

Issuing the setup DMA (**sdma**) instruction for an active channel will cause a DMA transfer to abort. The current DMA operation is terminated, and the channel is set up with the newly-issued **sdma** instruction. A DMA operation should not be terminated with the **sdma** instruction. 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 the **sdma** instruction.

CHANNEL PRIORITY

Each DMA channel is assigned a priority. When more than one DMA channel is enabled, the channel priority determines the order that 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, with Channel 3 having the lowest priority. In this mode, low-priority DMA's, assigned to Channels 1-3, can be locked out while a time-critical DMA, assigned to channel 0, receives all of the processor'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 every enabled channel. When fixed mode is enable, the latency for any channel is dependent on the activity of all channels of higher priority.

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 in the interrupt controller. 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 interrupts.)

The interrupt-pending bit for a DMA channel is set when one of the following conditions occur:

1) A non-chained DMA terminates because byte count reaches 0, or a chained DMA terminates because the null pointer is reached.
2) The $\overline{\text{EOP}}$/TC3:0 is programmed as an input, and the pin is 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.

When any of the conditions listed above occur, the current DMA request is completed before the pending bit in the IPND register is set.

CHANNEL SETUP, STATUS, AND CONTROL

The DMA controller uses the DMA command register (DMAC), and the setup-DMA instruction (**sdma**) to configure and control the four DMA channels. The update DMA instruction (**udma**) is used to monitor the status of a DMA operation in progress.

The DMAC register is a special function register (sf2). This register is used to enable or disable each channel and is the location of frequently-accessed status and control bits for the DMA controller, including idle or active status, and for the termination conditions for a channel.

The **sdma** instruction is used to configure each channel. Source address, destination address, byte count, transfer type, chained or non-chained operation are specified in the **sdma** instruction.

When a channel is set up using **sdma**, an eight-word block of internal data RAM is allocated for the channel. This section of data RAM is used to store the state of the channel when it 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. The **udma** instruction is provided to copy the state of an active DMA channel to the internal RAM.

The following action is usually taken to set up and start a DMA operation on the 80960CA. First, a channel is set up using the **sdma** instruction. Next, the DMAC register is modified to enable the DMA. The DMAC register is then read to monitor the activity of the DMA operation. The **udma** instruction can be issued and the DMA Data RAM examined for the current status of the DMA.

DMA Command Register (DMAC)

The DMA command register is a 32-bit SFR (Figure 13-5) specified as sf2 in the Intel ASM960 assembler. Bits 21-0 are used for DMA status and configuration. The remaining bits (bits 31-22) are reserved and must be programmed to 0 at initialization and at any time that DMAC is modified. The reserved bits are not actually implemented on the 80960CA chip. Clearing these bits at initialization is only required for portability to other 80960CX implementations. The function of DMAC is described below.

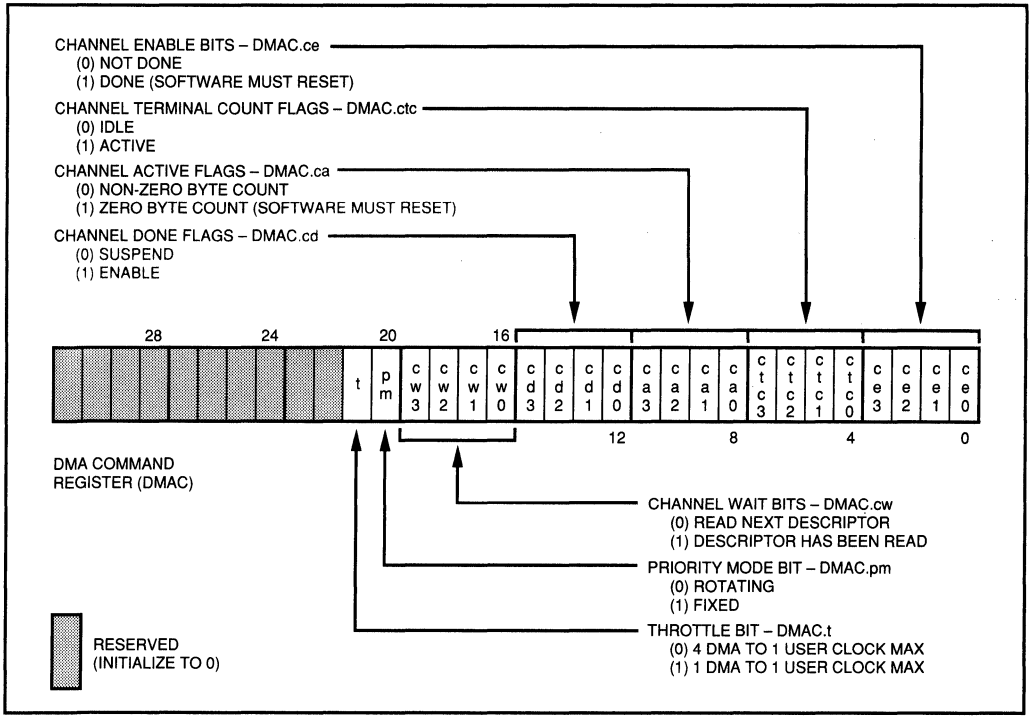


Figure 13-5. DMA Command Register (DMAC)

The *channel enable bits* (bits 3-0) enable or suspend a DMA operation 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 will be suspended after pending requests for the channel are serviced. The DMA operation will resume normally when the bit is reset. The channel enable bits are not reset by the DMA controller when an operation is completed. The channel enable bits must be cleared and then reset by the user to again enable the channel.

The *channel terminal count flags* (bits 7-4) indicate that a DMA operation has stopped because the byte count has reached zero. Flags 4 through 7 indicate terminal count for channels 0, through 3, respectively. A terminal count flag is set only after byte count has reached 0, and the last request for the channel has completed. A clear flag indicates a non-zero byte count. The terminal count flags are not cleared by the DMA controller when a channel is set up or enabled. 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.

The *channel active flags* (bits 11-8) indicate that a channel is either idle or active. Bits 8 through 11 indicate active channels 0 though 3, respectively. An active channel refers to a channel which is either in the middle of a transfer, or has a transfer pending (i.e. the requesting device has not been accessed). An idle channel refers to a disabled channel, or an enabled channel which is not responding to a DMA request. The channel active flags indicate status only. These flags can not be modified by software.

The *channel done flags* (bits 13-12) 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; or an $\overline{\text{EOP3:0}}$ signal is asserted which ends the DMA. The channel done flags are not cleared by the DMA controller when a channel is set up or enabled. This action must be performed by software. The channel done flags indicate status only. Modifying these flags does not affect the operation of the DMA controller.

The *channel wait bits* (bits 19-16) signals that a chaining descriptor has been read, and enables the reading of an ensuing chaining descriptor in memory. The channel wait bits only enable the descriptor read when the channel has been 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 always 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* for more detail.)

The *priority mode bit* (bit 20) selects fixed or rotating priority mode. The priority mode determines the order that DMA channels will be serviced if more than one request is pending. (See *Channel Priority.*)

The *throttle bit* 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.

Set Up DMA Instruction (sdma)

The set up DMA instruction (**sdma**) is used to configure a DMA channel. A single instruction for this purpose is provided to streamline the DMA set up procedure. A channel can be configured by simply issuing **sdma** with the instruction's operands in place.

The **sdma** instruction has the following format:

sdma	op1,	op2,	op3
	reg/lit/sfr	*reg/lit/sfr*	*reg*

The three operands are defined below:

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 in one of two ways depending on whether a chaining, or non-chaining mode is selected. The op3 register must be a quad-aligned register (i.e. the register number must be a multiple of 4).

When non-chaining modes are selected, op3 is the first of three consecutive 32-bit registers (op3, op[3+1], and op[3+2]). The user program must load these registers with the addresses for a DMA before **sdma** is executed. op3 contains the byte count for the DMA operation. op[3+1] contains the source address and op[3+2] the destination address for the DMA.

When chaining modes are selected, op3 is a 32-bit address. This address is the location of the first word of the first chaining descriptor. op[3+1] and op[3+2] are ignored.)See *Data Chaining* for more information on chaining descriptors.)

When a fly-by mode is selected, op3 is the byte count, and op[3+1] is the fly-by address; op[3+2] is ignored.

The channel setup mechanism, started with the **sdma** instruction, is two-part. The **sdma** instruction is a multi-cycle instruction. When **sdma** is issued, the instruction first executes, reading the register operands for the DMA operation; then completes, freeing these registers for use by other instructions. Second, 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 has completed. In this case, the DMA controller will simply wait for the setup to complete before the DMA operation begins. The result is the potential for additional latency on the first DMA request. The additional latency is decreased by issuing 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 has completed. The second **sdma** instruction 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 will not adversely increase interrupt latency.

DMA Control Word

The *DMA control word* (Figure 13-6) specifies modes and options for a DMA. The control word is an operand (op2) of the **sdma** instruction.

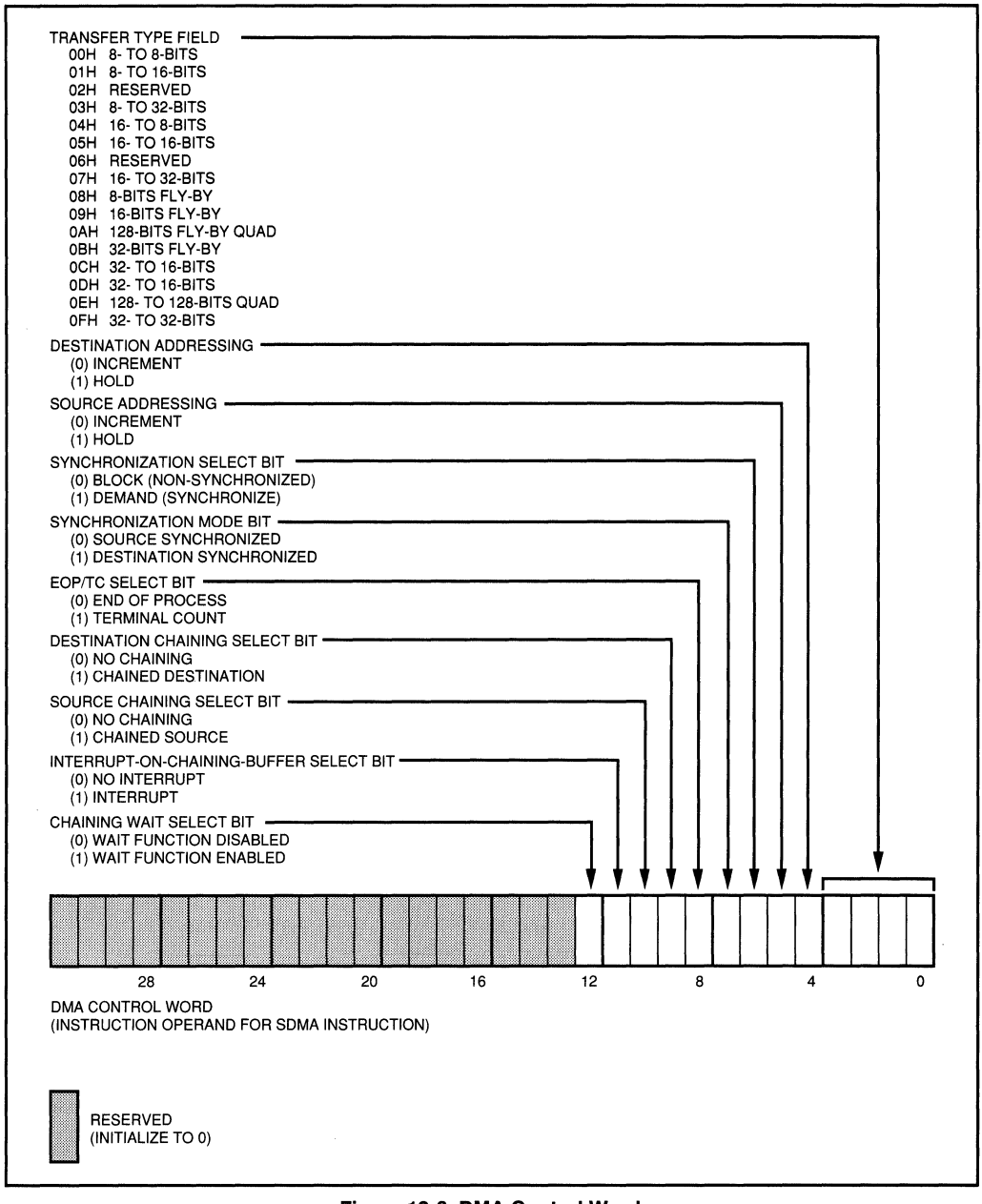


Figure 13-6. DMA Control Word

13

The *transfer type field* (bits 3-0) specifies the data length of bus requests issued by the DMA controller and selects between standard and fly-by transfers.

The *source/destination addressing bits* (bits 4 and 5) determine if the source or the destination address for a channel is held fixed or incremented during a DMA. Bit 5 controls the source address and bit 4 controls the destination address.

The *synchronization select bit* (bit 6) determines if a transfer is a demand mode (synchronized) or block mode (non-synchronized) transfer.

The *synchronization mode bit* (bit 7) specifies that a demand mode transfer is synchronized with the source, or the destination. In fly-by mode, the bit specifies that fly-by stores (0), or fly-by loads (1), are performed. Fly-by stores are source synchronized, and fly-by loads are destination synchronized. In block mode, this bit ignored.

The *EOP/TC select bit* selects the function of the $\overline{\text{EOP/TC3:0}}$ pin. If this bit is set, the pins are configured as inputs ($\overline{\text{EOP3:0}}$). Forcing the pin active terminates a non-chained DMA or source and destination-chained buffer to complete. If the EOP/TC select bit is clear, the pin is configured as an output ($\overline{\text{TC3:0}}$). When the byte count for a DMA reaches zero, the pin is driven active for one clock. (See *Terminating or Suspending a DMA* in this chapter.)

The following bits in the DMA control word control data chaining. If chaining mode is not used, these bits are set to 0.

The *source/destination chaining select bits* (bits 9 and 10) enable chaining modes for a channel. Bit 9 enables source chaining. Bit 10 enables destination chaining. Non-chaining mode is selected if both bits are clear. (See *Data Chaining* in this chapter.)

The *interrupt-on-chaining-buffer select bit* (bit 11) determines if a DMA interrupt is generated when the byte count for a chained buffer reaches 0. The option is selected when the bit is set. This bit is ignored when a non-chaining mode is selected.

The *chaining-wait select bit* (bit 12) enables the channel-wait function. When the wait-enable function is selected, the channel-wait bits in the DMAC register 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, eight words of data RAM are dedicated for use by that channel (Figure 13-7). When channel service swaps between channels, the state of last active channel is saved in data RAM. The state is retrieved when the channel is again serviced.

Note: Channel swapping occurs when the 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.)

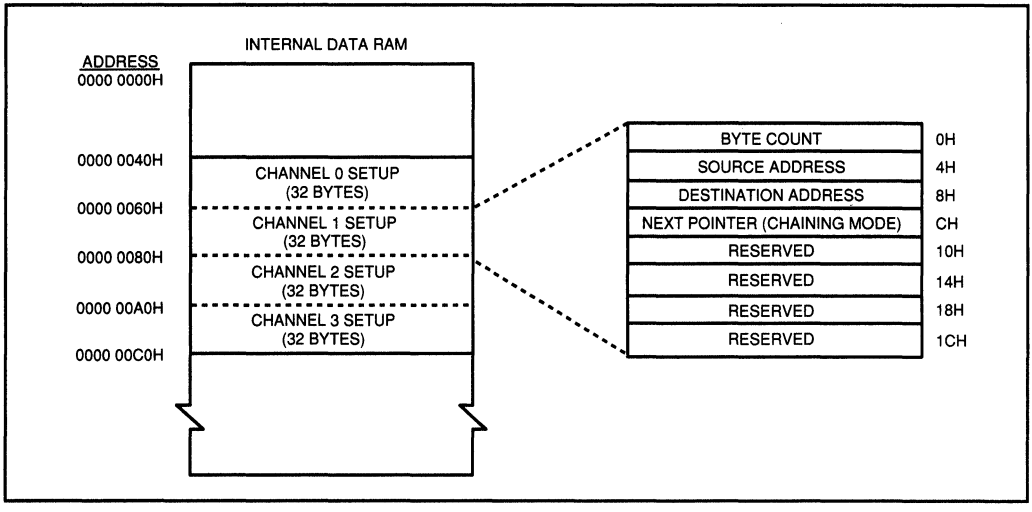


Figure 13-7. DMA Data RAM

The **udma** instruction is provided to flush the state of a currently-executing transfer to data RAM. The DMA state in data RAM may or may not be current, depending on how frequently a channel has been swapped. The **udma** instruction, therefore, should always be issued before accessing the DMA data RAM to ensure that the DMA state is current.

After **udma** is executed, the current byte count, source address, and destination address can be read. Additionally, the next descriptor pointer (NPTR) is provided when a data chaining is enabled.

If a transfer is in progress when **udma** is executed, the information written to data RAM may be different from the actual progress of the DMA by one transfer. This happens because DMA transfers are decoupled from the user code which issued the **udma** instruction.

The DMA data RAM is 128 bytes of the internal RAM located at addresses 0000 0040H to 0000 00BFH (Figure 13-7). This memory is read/write in supervisor mode, and read only in user mode. (See *Chapter 2, Programming Environment.*) This supervisor protection is provided to prevent errant modification of the DMA RAM by a program.

The 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 the data RAM dedicated for a channel which is already set up and is awaiting activity. In general, any modification of the DMA Data RAM for an active or idle channel may cause unpredictable operation of the DMA controller. Conversely, executing the **sdma** instruction 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 address for channel 0
        ldconst         0xf,g3          # DMA control word (32/32 standard - 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 control word (32/32 standard - source
                                        # hold - dest. inc. - demand source sync. -
                                        # dest. chaining, 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
```

13

EXTERNAL INTERFACE DESCRIPTION

The characteristics of the DMA signals (DACK3:0 DREQ3:0, and EOP/TC3:0) and the timing requirements for DMA synchronization are described in the following sections.

Pin Description

Twelve pins are dedicated to the DMA controller. Three pins are associated with each channel.

DREQ3:0 DMA Request (input) - The 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 and DREQ3:0 has the lowest priority. A request is generated by activating the DREQ3:0 pin for a channel.

DACK3:0 DMA Acknowledge (output) - This output is used to notify an external requestor that a requested transfer is taking place. The pin is active for the duration of the access of the requestor.

EOP/TC3:0 End of Process (input) or Terminal Count) (output) - This pin functions as either an input (EOP3:O) or as an output (TC3:0). Programmed as an output, the pin is driven active (low) for one clock when byte count reaches zero and after the last transfer for a DMA has completed. Programmed as an input, an asynchronous active (low) signal on the pin, for a minimum of two clock cycles, will cause one or both of the following events to occur. 1) The DMA terminates, or 2) the DMA terminates the current chaining buffer and continues with chaining if both the source and destination are chained. In each case, a pending bit in the IPND register is set when the pin is asserted.

Demand Mode Request/Acknowledge Timing

Demand-mode transfers require a request from an external requestor before the transfer is started. This DMA transfer should satisfy two basic requirements. First, after a transfer is requested, the DMA controller must be fast in responding to the request by asserting an acknowledge signal for the transfer and accessing the requestor. This characteristic is referred to as latency. Second, the requesting device must be given enough time to deassert the request signal to prevent an unwanted transfer. The timing for source and destination synchronization is described in the following sections.

DMA transfers are requested by asserting (active low) one of the DMA request pins. The acknowledge pin is asserted during the access of the requestor and is used in the external system to select the DMA requestor. The timing of the DACK3:0 outputs are shown in Figure 13-8. (For timing specifications refer to the *80960CA Data Sheet.*)

To start a synchronized DMA, the DREQ3:0 signal must be asserted and held asserted until the requestor is accessed with the DACK3:0 pin. When the bus access for the requestor is in progress, the DREQ3:0 signal may be held active to initiate further DMA transfers, or DREQ3:0 may be driven inactive to prevent further transfers. The DREQ3:0 pin is sampled at a specific time during the access of the requestor (Figure 13-8). This time is different for the following conditions.

1) The bus access for the DMA transfer is immediately followed by the same kind of access. For example, a load access is immediately followed by a load, or a store access by a store. These adjacent loads or stores are caused by assembly or disassembly, or by fly-by transfers.
2) The bus access for the DMA transfer is immediately followed by an opposite type of access. For example, a store access is followed by a load, or a load access by a store. The alternating loads and stores occur when the source and destination transfer lengths for a standard transfer are equal.

Figure 13-8 shows when DREQ3:0 is detected relative to the access of the requesting device. For condition 1, DREQ3:0 is first sampled 1 clock cycle before the end of the device access. For condition 2, DREQ3:0 is first sampled 2 clocks cycles after the device access. (See *Chapter 11, External Bus Description* for a detailed description of the 80690CA's external bus characteristics.)

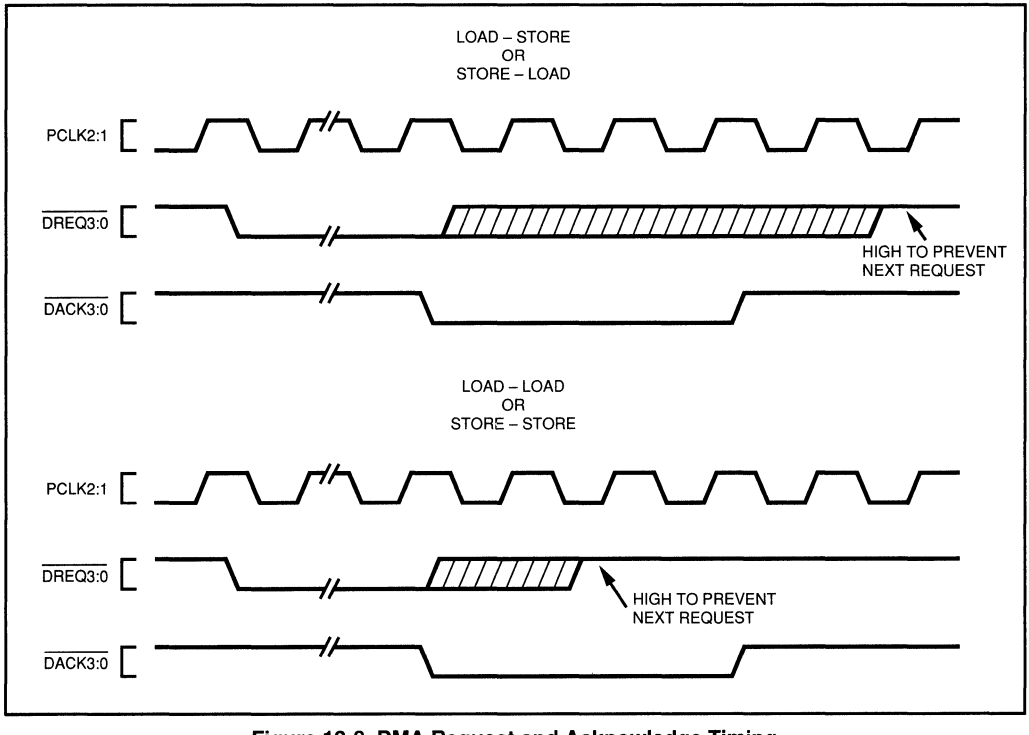


Figure 13-8. DMA Request and Acknowledge Timing

The timing for the DREQ3:0 signals is specified with respect to the end of the bus access (deassertion of DACK3:0). This only applies for bus cycles which are controlled by the internal wait-state generator. When bus cycles are controlled by the external ready inputs, the specification is to the end of the bus access if the ready inputs were immediately returned active.

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, the 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

The EOP/TC3:0 pin can be programmed as an input (EOP3:O) or as an output (TC3:0) for each channel. When programmed as an output, the TC3:0 pins indicate that byte count for a DMA has reached zero for a particular channel. A TC3:0 pin for a channel is driven active after the byte count reaches zero, and the last access of the synchronizing device is complete. The TC3:0 pins are driven active (logic 0) for one PCLK2:1 cycle.

To ensure that a particular device access does not occur after the EOP3:O inputs are asserted, the same timing is enforced for these pins as for the DREQ3:0 inputs.

Block Mode Transfers

Block mode DMA's require no synchronization with a source or a destination device. The DREQ3:0 inputs are ignored during block mode DMA's. The acknowledge signal (DACK3:0) is driven active when the source is accessed. The EOP/TC3:0 pins have the same function as described above in the section *End of Process/Terminal Count Timing*.

The DMA access pin (DMA) indicates that a bus access was initiated by the DMA controller. The pin is asserted (low) for any DMA load or store access.DMA is deasserted (high) for other access. The DMA pin has the same timing as the W/R pin. (See *Chapter 11, External Bus Description* for a complete timing description of the DMA pin.)

DMA CONTROLLER IMPLEMENTATION

The DMA controller is implemented with dedicated logic, microcode, multi-process core hardware, and the resources of the bus controller. The DMA controller uses microcode to perform DMA transfers. DMA features, including data chaining, data alignment, and byte assembly and disassembly are implemented in microcode. The DMA logic arbitrates channel requests, handles the DMA external interface, and manages the microcode execution of transfers.

DMA and User Program Process

The 80960CA is designed to allow DMA operations to be executed in microcode while still 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's (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 80960CA to switch processes on clock boundaries (i.e., no instruction overhead is necessary to switch the process). With this switching mechanism, the DMA microcode and the user program can frequently alternate execution with absolutely no performance loss caused by the process switching.

A process switch from the user process to the DMA process occurs as a result of a *DMA event.* A DMA event is signalled when a DMA channel requires service, or is in the process of setting up a channel. Signalling the DMA event is controlled by the DMA logic.

During a DMA event , 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 will take four clock cycles to every single-user process clock. The ratio of DMA to core cycles can also be selected as 1:1 to maximize core response while a DMA is in progress. The user to DMA cycles 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. The actual cycle allocation between the user process and the DMA process depends on the type of DMA operation which is performed, the activity of a DMA channel, and the loading and performance of the external bus.

Bus Controller

The bus controller unit (BCU) performs the base function of accessing 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.

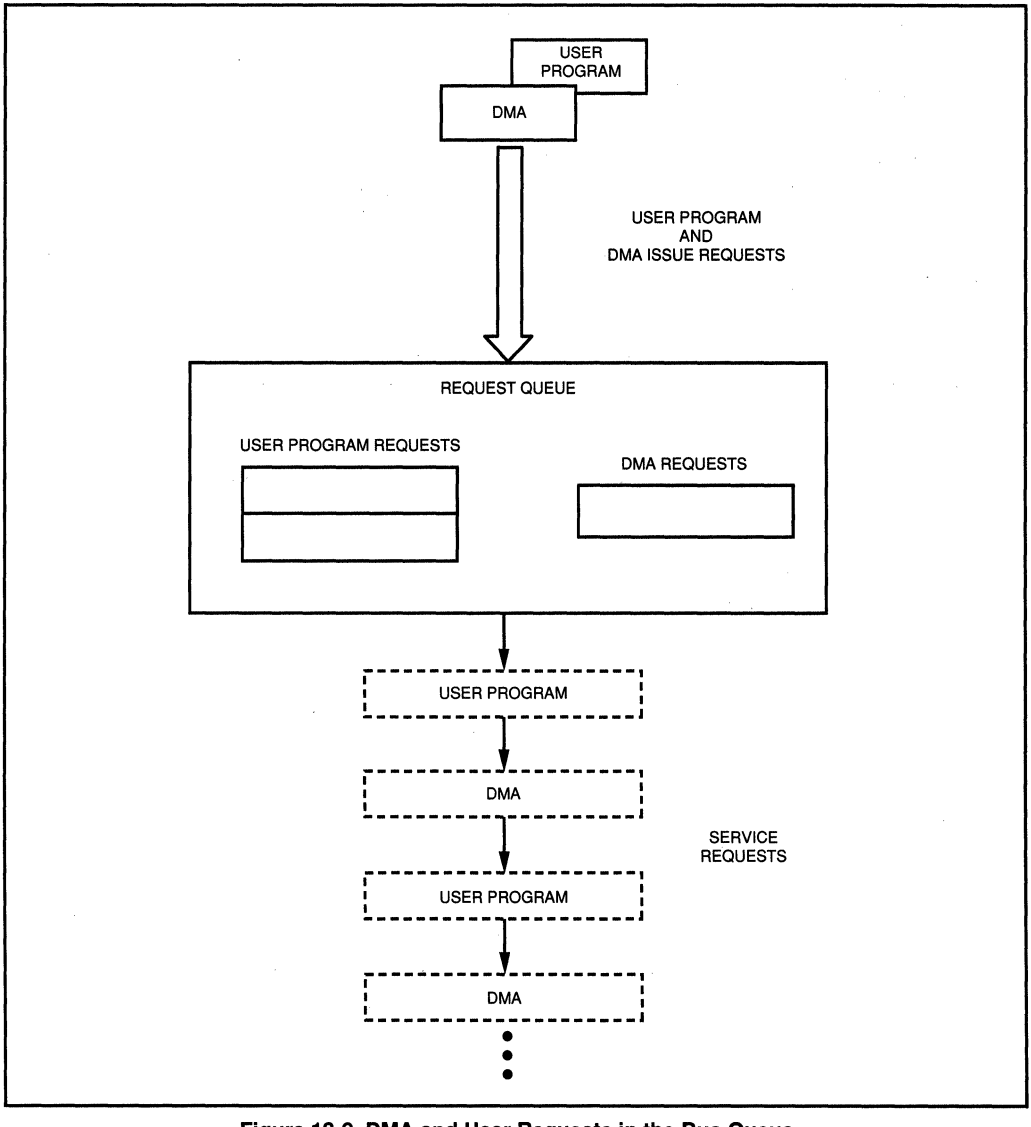


Figure 13-9. DMA and User Requests in the Bus Queue

The BCU contains a queue which accepts up to three pending requests for bus transactions (Figure 13-9). When a DMA channel is set up, the queue is divided so that one slot is dedicated for DMA-process requests, and two slots are dedicated for user-process requests. The DMA and core entries are arranged in such a way that when both a user slot and a DMA slot are filled, the servicing of bus requests alternates between requests issued by the user process and those issued by the DMA process.

DMA Controller Logic

The 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

A DMA transfer is characterized by two values: its throughput and its latency (Figure 13-10). Throughput describes how fast data is moved by a DMA operation. A block mode DMA is described by its throughput. Latency is the time required in demand-mode transfers for the DMA controller to respond to a request by an external device. The latency of a DMA, in general, is the amount of time required to start the transfer.

The throughput of a DMA transfer depends on the configuration of the DMA controller as well as the configuration and activity of the user's system. Several of the items which determine DMA throughput are listed below:

- DMA transfer type
- Data alignment for the source/destination address and the byte count
- The performance of the external bus
- The bus activity generated by the user process when a DMA is in progress
- The value of the DMA throttle bit

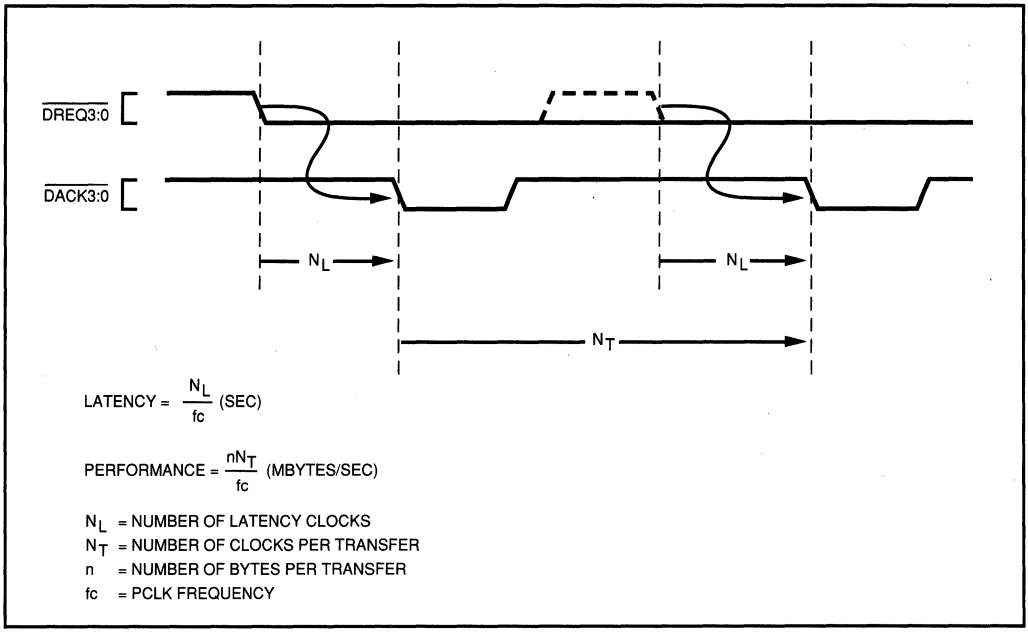


Figure 13-10. DMA Throughput and Latency

Table 13-4 lists DMA throughput for each of the DMA transfer types. The table assumes that source and destination for the DMA are aligned and located in 0 wait-state memory. The throttle bit for the transfer is set to 0. This means, at most, the DMA controller will be given four clocks to every one clock given to the user process. The values in Table 13-4 also assume that bus requests generated by the user process do not interfere with the requests which are issued by the DMA process.

Table 13-4. DMA Throughput

Transfer Type	Bus Mode	Performance (in PCLK cycle N_T)		
		DMA Clocks	User Clocks	Total Clocks
8/8 Standard	All	4	6	10
8/16 Standard	All	12	8	20
8/32 Standard	All	25	15	40
16/8 Standard	All	10	8	18
16/16 Standard	All	4	6	10
16/32 Standard	All	12	8	20
32/8 Standard	All	23	13	36
32/16 Standard	All	10	8	18
32/32 Standard (aligned)	All	4	6	10
32/32 Standard (non-aligned)	All	8	6	14
128/128 Standard	Non-burst, Non-pipe	6	14	20
128/128 Standard	Burst, Non-pipe	6	11	17
128/128 Standard	Non-burst, Pipe	6	11	17
128/128 Standard	Burst, Pipe	6	11	17
8/8 Fly-by	All	3	3	6
16/16 Fly-by	All	3	3	6
32/32 Fly-by	All	3	3	6
128/128 Fly-by	Burst	3	3	6

Note: Values assume a 0 wait state bus and no user generated bus activity.

There are several noteworthy items to mention about the values in Table 13-4. Notice that total clock cycles for a DMA transfer are given as the DMA clocks and user clocks for each transfer. (Recall that the DMA operation and user program are executed concurrently.) At the maximum bandwidth DMA, the user process is still given 50% of the available processor bandwidth. For other transfer types, even more of the processor's bandwidth is available for the user process. Also, note from the table that transfer types which involve packing or unpacking use more DMA clocks per byte transfer than the other transfer types. This is due to the additional microcode which is required to handle the data manipulation internally. The optimized, non-aligned word-to-word transfers are also lower in throughput because of the microcode required to perform the alignment.

The memory speed for the source or destination of a DMA naturally impacts the throughput. The throughput decreases, at least linearly, with addition of wait states to the source and destination memory. Although, the DMA efficiency decreases, the total number of DMA cycles needed to execute the transfer remains almost invariant with the addition of bus wait states. The additional clock cycles for each transfer are instead available as user process clocks.

The bus activity which is generated by the user process may also affect DMA throughput. Bus requests issued by the DMA process and user process are given alternating service by the bus controller. If the user process issues a request to slow memory, the DMA controller must wait for bus activity to finish before the next DMA-issued bus request is serviced.

DMA latency is the sum of several components which are related to the characteristics of the DMA transfer. Table 13-5, along with the following descriptions of each component, can be used to estimate latency for a typical DMA transfer.

Set up the DMA channel ($N_{(setup)}$) - This component of latency describes the number of clocks required to set up a channel with the **sdma** instruction. $N_{(setup)}$ only affects the first transfer of a DMA operation. The maximum set up latency occurs if a transfer is requested at the same time that the **sdma** instruction is issued. Typically, the **sdma** instruction is issued in advance of the first DMA request, reducing this latency component to 0.

Enable the DMA channel ($N_{(enable)}$) - This latency component describes the number of clocks required to enable a DMA channel by setting the channel enable bit in the DMAC register (DMAC.ce). $N_{(enable)}$ only affects the first transfer of a DMA operation. The maximum enable latency occurs if a transfer is requested at the same time that the channel is enabled. Typically, a channel is enable in advance of the first request, reducing this latency component to 0.

Chaining latency ($N_{(chain)}$) - The chaining latency, $N_{(chain)}$, is the latency from arbitrating another buffer transfer in chaining mode and from reading a chaining descriptor from memory. This latency only occurs at the boundaries of buffer transfers when in chaining mode. When chaining modes are not selected, Nchain equals 0.

Swap the DMA channel ($N_{(swap)}$) - This time is defined as the amount of time that it takes a higher priority channel to preempt a lower priority channel, and the time required to copy the associated DMA working registers to RAM. The value of $N_{(swap)}$ should be multiplied by the number of channels which will be preempted when a DMA is requested.

Bus latency ($N_{(bus)}$) - This component of latency is caused by activity on the external bus when a DMA transfer is requested. This activity is caused by transfers which are executing or pending when a DMA request occurs. When a requested DMA is the highest priority, the value of $N_{(bus)}$ is, at most, the number of clocks needed to perform a single transfer. (This is the throughput of the transfer.) When a DMA is requested on a low priority channel, the maximum value of $N_{(bus)}$ is the sum of the number of clocks needed to perform the current transfer and the latencies for other channels which are higher priority. The bus latency can also be extended if an external agent controls the bus with the Hold/Hold Acknowledge function.

Table 13-5. DMA Latency

DMA Latency Component	Value (in PCLK cycles NL)	Condition
$N_{(setup)}$	0-25 0	First transfer after setup All except first transfer
$N_{(enable)}$	0-5 0	First transfer after setup All except first transfer
$N_{(chain)}$	8-10 0	Chaining, enabled, descriptor in internal data RAM Chaining disabled
$N_{(swap)}$	9-12 0	Single channel pre-empted No channel pre-empted
$N_{(bus)}$	0-N_T 0	Transfer in progress on same channel No transfer in progress

Note: Values assume that no bus activity is generated by the user program when the DMA is in progress.

14 Initialization and System Requirements

CHAPTER 14
INITIALIZATION AND SYSTEM REQUIREMENTS

This chapter describes the steps that the 80960CA takes during its initialization. The $\overline{\text{RESET}}$ pin, the reset state of the processor, the built-in self-test features, and on-circuit emulation function (ONCE™) are discussed. The chapter also describes the basic system requirements for the 80960CA, including power, ground, and clock requirements, 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 80960CA is in a quiescent reset state. All external pins are inactive, and the internal state of the processor is forced to a known condition. The processor begins its initialization when the $\overline{\text{RESET}}$ pin is deasserted.

When initialization begins, the 80960CA uses an Initial Memory Image (IMI) to establish its state. The IMI contains the Initialization Boot Record (IBR), the Process Control Block (PRCB), and the system data structures. The IBR contains the addresses of the first instruction of the user's code and the PRCB. The PRCB contains pointers to system data structures. Several data-structure pointers are cached internally at initialization. The PRCB also contains information used to configure the processor at initialization.

The 80960CA may be reinitialized by software. When a reinitialization takes place, a new PRCB and a reinitialization instruction pointer are specified. Reinitialization is useful for relocating data structures from ROM to RAM after initialization.

The 80960CA supports several facilities to assist in system testing and start-up diagnostics. The ONCE mode electrically removes the 80960CA from a system. This feature is useful for system-level testing where a remote tester exercises the 80960CA system. During initialization, the 80960CA performs an internal functional self-test and an external bus self-test. These features are useful for system diagnostics to ensure the base functionality of the 80960CA and the system bus.

The 80960CA 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 80960CA 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 the RESET pin is deasserted. At this time, the processor automatically configures itself with information specified in the IMI, and performs its built-in self-test. The processor then branches to the first instruction of the user's code.

The objective of the initialization sequence on the 80960CA is to provide a complete, working initial state when the first instruction of the user's code is encountered. The user's start-up code has only to perform several base functions to place the processor in a configuration for executing application code.

Reset Operation (RESET)

The RESET pin, when asserted (active low), causes the chip 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 RESET pin is deasserted, the chip begins its initialization as described later in this chapter. RESET is a level-sensitive, asynchronous input. The pin is internally synchronized on the rising edge of PCLK2:1.

The RESET pin must be asserted when power is applied to the 80960CA. The processor will then stabilize in the reset state. This power-up reset is referred to as *cold reset*. To ensure that all of the 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 RESET pin can be deasserted.

The 80960CA may also be cycled through the reset state after execution has started. This is referred to as *warm reset*. The RESET pin, for a warm reset, must be held for a minimum number of clock cycles before the RESET pin can be deasserted. The specifications for a cold and warm reset can be found in the *80960CA 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 80960CA. The 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.)

Table 14-1. Pin Reset State

Pins(1)	Reset State	Pins(1)	Reset State
A31:2	Floating	$\overline{DMA}$	Floating
D31:0	Floating	$\overline{SUP}$	Floating
$\overline{BE3:0}$	High (inactive)	$\overline{FAIL}$	Low (active)
W/$\overline{R}$	High (inactive)	$\overline{DACK3}$	High (inactive)
$\overline{ADS}$	High (inactive)	$\overline{DACK2}$	High (inactive)
$\overline{WAIT}$	High (inactive)	$\overline{DACK1}$	High (inactive)
$\overline{BLAST}$	High (inactive)	$\overline{DACK0}$	High (inactive)
DT/$\overline{R}$	High (inactive)	$\overline{EOP/TC3}$	Floating (input)
$\overline{DEN}$	High (inactive)	$\overline{EOP/TC2}$	Floating (input)
$\overline{LOCK}$	High (inactive)	$\overline{EOP/TC1}$	Floating (input)
BREQ	Low (inactive)	$\overline{EOP/TC0}$	Floating (input)
$\overline{D/C}$	Floating		

Note: 1) The pin states shown assume that the HOLD and the $\overline{ONCE}$ pins are not asserted. If HOLD is asserted during reset, the hold is acknowledged by asserting the HOLDA pin, and the processor pins are configured in the Hold Acknowledge state (See *Chapter 10, Bus Controller.*) If the $\overline{ONCE}$ pin is asserted, the processor pin 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

Notes: 1) All control registers (not listed) are configured with their respective values from the control table after reset.

Self-Test Function (STEST, $\overline{\text{FAIL}}$)

The 80960CA executes a bus-confidence self-test and, optionally, an internal self-test program as part of its initialization. The self test (STEST) pin is provided to enable or disable the internal self-test feature. The failure ($\overline{\text{FAIL}}$) pin is provided to indicate that either of the self-tests passed or failed.

The internal self-test is designed to check the basic functionality of the internal data paths, registers, and memory arrays on-chip. The internal self-test is not intended for a full validation of the processor's functionality. The internal self-test is provided to detect catastrophic internal failures and to compliment a user's system diagnostics by ensuring a confidence level in the processor before any system diagnostics are executed.

The 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 will execute the internal self-test. If the pin is deasserted, the processor will bypass the internal self-test. The external bus-confidence test is always performed regardless of the value of the STEST pin.

The external bus-confidence self-test is designed to check the functionality of the external bus. 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* in this chapter for a description of the IBR and the check sum words.)

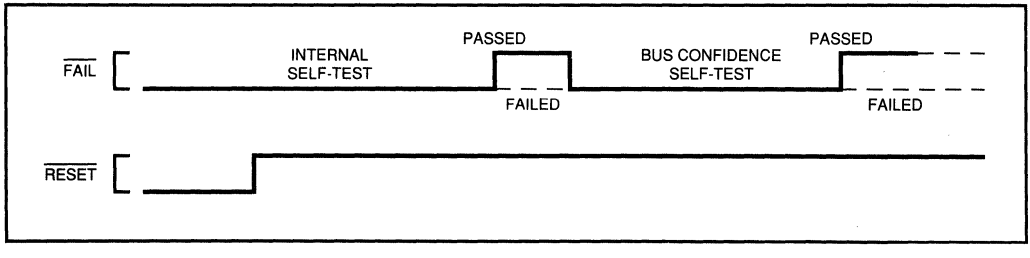


Figure 14-1. $\overline{\text{FAIL}}$ Timing

The FAIL pin is used to signal errors in either the internal self-test or the bus-confidence self-test. The FAIL pin 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, the FAIL pin is deasserted. When the internal self-test is disabled (with the STEST pin), the FAIL pin still toggles at the point where the internal self-test would occur even though the internal self-test is not executed. The FAIL pin becomes undefined after the processor begins the fetch of the first user instruction following initialization.

On-Circuit Emulation (ONCE)

On-circuit emulation is provided to aid board-level testing. The on-circuit emulation feature allows a mounted 80960CA to electrically remove itself from a circuit board. In the ONCE mode, the processor presents a high impedance on every pin, nearly eliminating the 80960CA's power demands on the circuit board. Once the 80960CA is electrically removed, a functional tester can take the place of (emulate) the mounted 80960CA and execute a test of the 80960CA system.

The on-circuit emulation mode is entered by asserting (low) the ONCE pin while the 80960CA is in the reset state. The value of the ONCE pin is latched on the rising edge of the RESET signal.

The ONCE pin should be left unconnected in an 80960CA system. The pin is internally connected to VCC through an internal pull-up resistor, causing the unconnected pin to normally remain in the inactive state. To enter the on-circuit emulation mode, an external tester simply drives the ONCE pin low (overcoming the pull-up resistor) and initiates a reset cycle. To exit the on-circuit emulation mode, the reset cycle must be repeated, with the ONCE pin deasserted prior to the rising edge of RESET . (See the *80960CA Data Sheet* for specific timing of the ONCE pin and the characteristics of the on-circuit emulation mode.)

Initial Memory Image

The Initial Memory Image (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 the integrated peripherals; 2) it provides pointers to the system data structures and the first instruction to be executed after the processor's initialization; and 3) it provides check-sum words that the processor uses in its self-test routine at start-up.

The IMI is made up of three components: the initialization boot record (IBR), the process control block (PRCB), and system data structures. Figure 14-2 shows the components of the IMI. The IBR is fixed in memory; the other components are referenced directly or indirectly by pointers in the IBR and the PRCB.

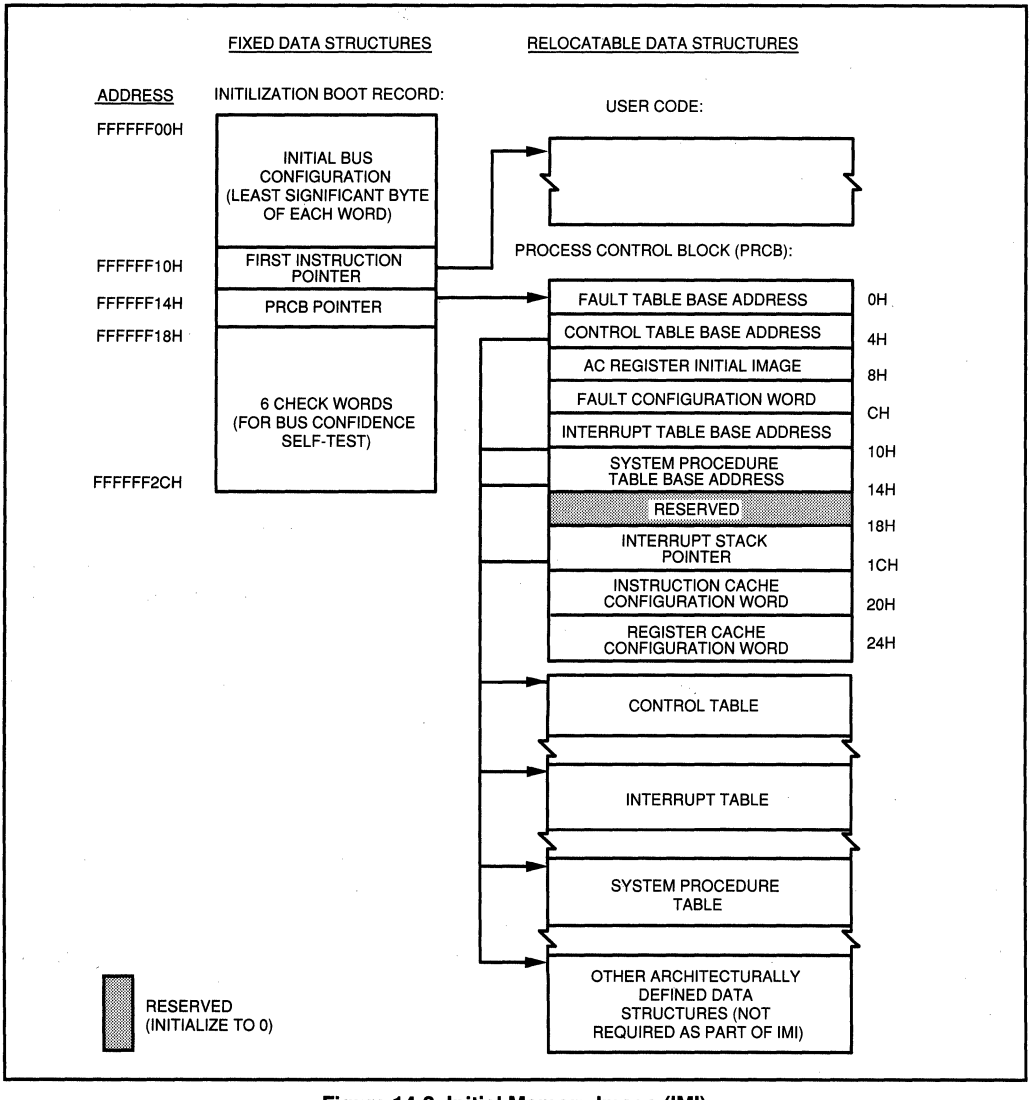


Figure 14-2. Initial Memory Image (IMI)

INITIALIZATION BOOT RECORD

The IBR is the primary data structure required to initialize the 80960CA. 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 is reading the IMI during initialization, it must know the bus characteristics of the external memory where the IMI is located. This bus configuration is read from the first four words of the IBR. At initialization, the processor will perform loads from the lower order byte of the first four words of the IBR. These four 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 word 0 of the IBR is loaded into the lowest byte position of the MCON0 register, and the next three bytes are loaded into successively higher byte positions. The most significant byte (byte 3) of the register is reserved, and must be set to 00H. (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 will take 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 will allow the bus configuration data in the IBR to be loaded regardless of the 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 bytes of configuration data are then read with requests which use the new value of N_{RAD}. Once all four bytes are read, MCON0 is re-written, and initialization continues. This is done to reduce the number of clocks necessary 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, consisting of the first instruction pointer, the PRCB pointer, and six check-sum words. The PRCB pointer and the first instruction pointer are internally cached. The six check-sum words, along with the PRCB pointer and the first instruction pointer, are used in a check-sum calculation which implements a confidence test of the external bus. The sum of these eight words plus FFFF FFFFH must equal 0. The processor uses the following algorithm to calculate the checksum.

```
Begin _checksum:
        lda     0xffff ffff, r4     # initialize checksum
        lda     0xffff ff00, r5     # load IBR address
        addo    0x10, r5, r5        # adjust to point to first instruction pointer
        addc    0, 0, r6            # clear carry
        ld      (r5), r6            # load first word
        addc    r4, r6, r7          # accumulate
        mov     7, r8               # set loop count
loop:   addo    4, r5, r5           # increment memory pointer
        ld      (r5), r6            # load
        addc    r6, r7, r7          # accumulate
        cmpdeco 1, r8, r8           # loop control
        bne     loop                # loop control
        cmpo    0, r7               # check for correct checksum
self:   bne     self                # stop here if wrong
        .                           # go on otherwise
        .
        .
        .
```

PROCESS CONTROL BLOCK

The PRCB contains base addresses for system data structures, and contains 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, the no imprecise faults bit, and the condition code bits to be selected at initialization.

The condition code bits in the AC initial image can be used to specify the source of an initialization or reinitialization when a single instruction entry point to the user start-up code is desirable. This is accomplished by programming the condition code in the AC initial image to a different value for each different entry point. The user start-up code can detect the condition code values, and thus the source of the reinitialization, by using the compare or compare-and-branch instructions.

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 will not be generated when a non-aligned bus request is issued. The 80960CA, in this case, will automatically perform 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 a 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. The instruction cache will remain disabled, until one of two operations is performed: 1) The processor is reinitialized with a new value in the instruction-cache configuration word; or 2) the **sysctl** instruction is issued with the configure instruction-cache message type and a cache configuration mode other than disable cache. (See *Chapter 2, Programming Environment* for a description of the **sysctl** instruction.)

The *register cache configuration word* is used to specify the number of register sets which are cached on-chip. The integrated procedure call mechanism saves the local register set when a call is executed. The 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 least significant four bits of the register-cache configuration word specifies the number of local register sets which are 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 the internal data RAM for the register cache. (See *Chapter 7, Procedure Calls* for a complete description of the register caching mechanism.)

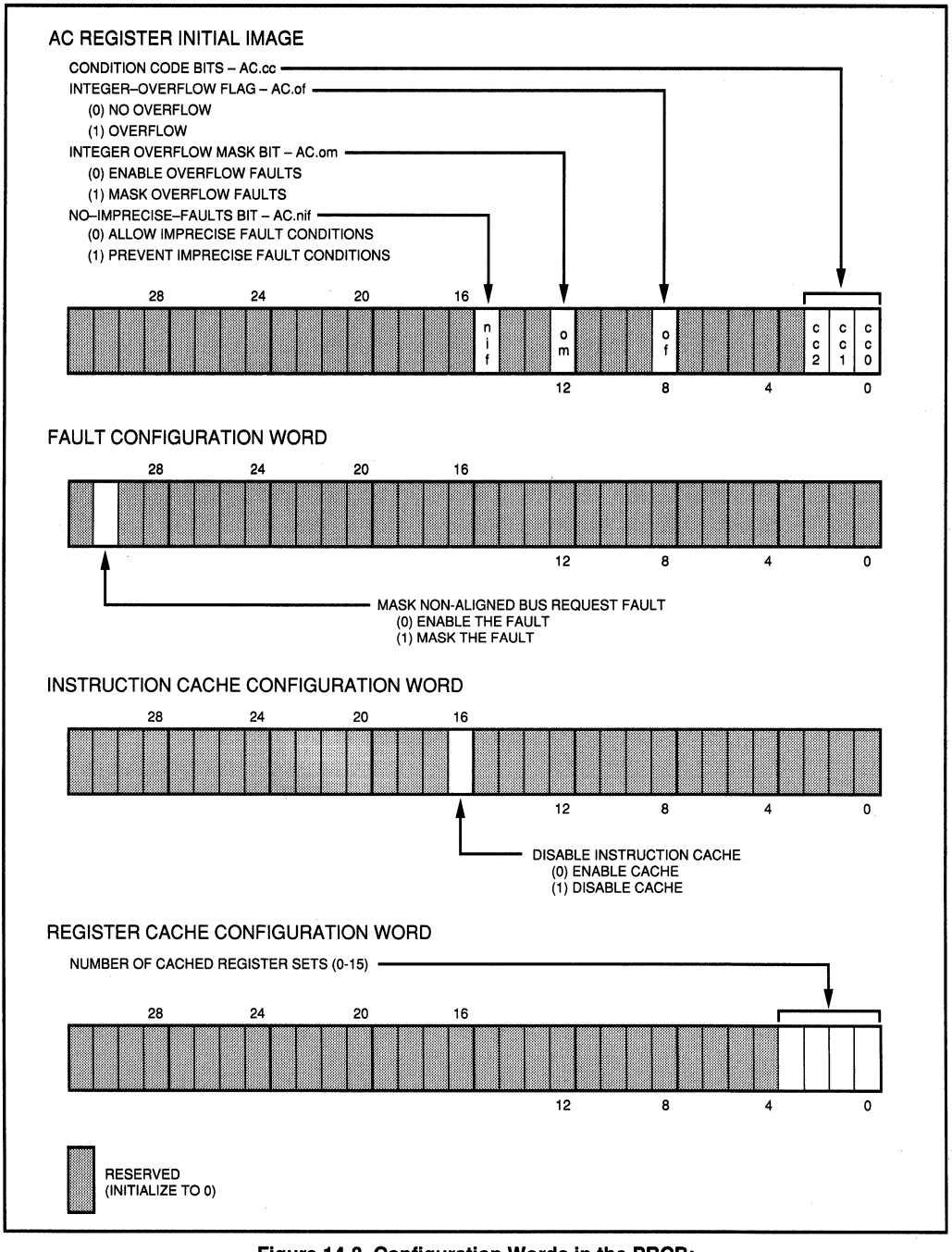


Figure 14-3. Configuration Words in the PRCB;

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. The required data structures are the PRCB; IBR; system-procedure table; the control table; and the interrupt table. These data structures are usually programmed in the system's boot ROM, located in memory region-15 of the address space.

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 base address of the system-procedure table 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 containing the values of the on-chip control registers. At initialization, the control table is automatically loaded. The control table 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, the user stack, the supervisor stack, and the interrupt stack. The necessary stacks must be located in a system's RAM. The fault table is typically located in the boot ROM. If it is necessary to locate the fault table in RAM, the processor must be reinitialized.

Reinitialization and Relocating Data Structures

The processor can be reconfigured, and pointers to data structures can be changed by reinitialization. The 80960CA is reinitialized by issuing the **sysctl** instruction with the reinitialize processor message type. (See *Chapter 2, Programming Environment* for a description of the **sysctl** instruction.) 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 because the processor writes to the pending priorities and pending interrupts fields in this table to post software-generated interrupts. It may also be necessary to relocate the control table to RAM. The control table must be in RAM if the values of the control registers 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 the description of 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.

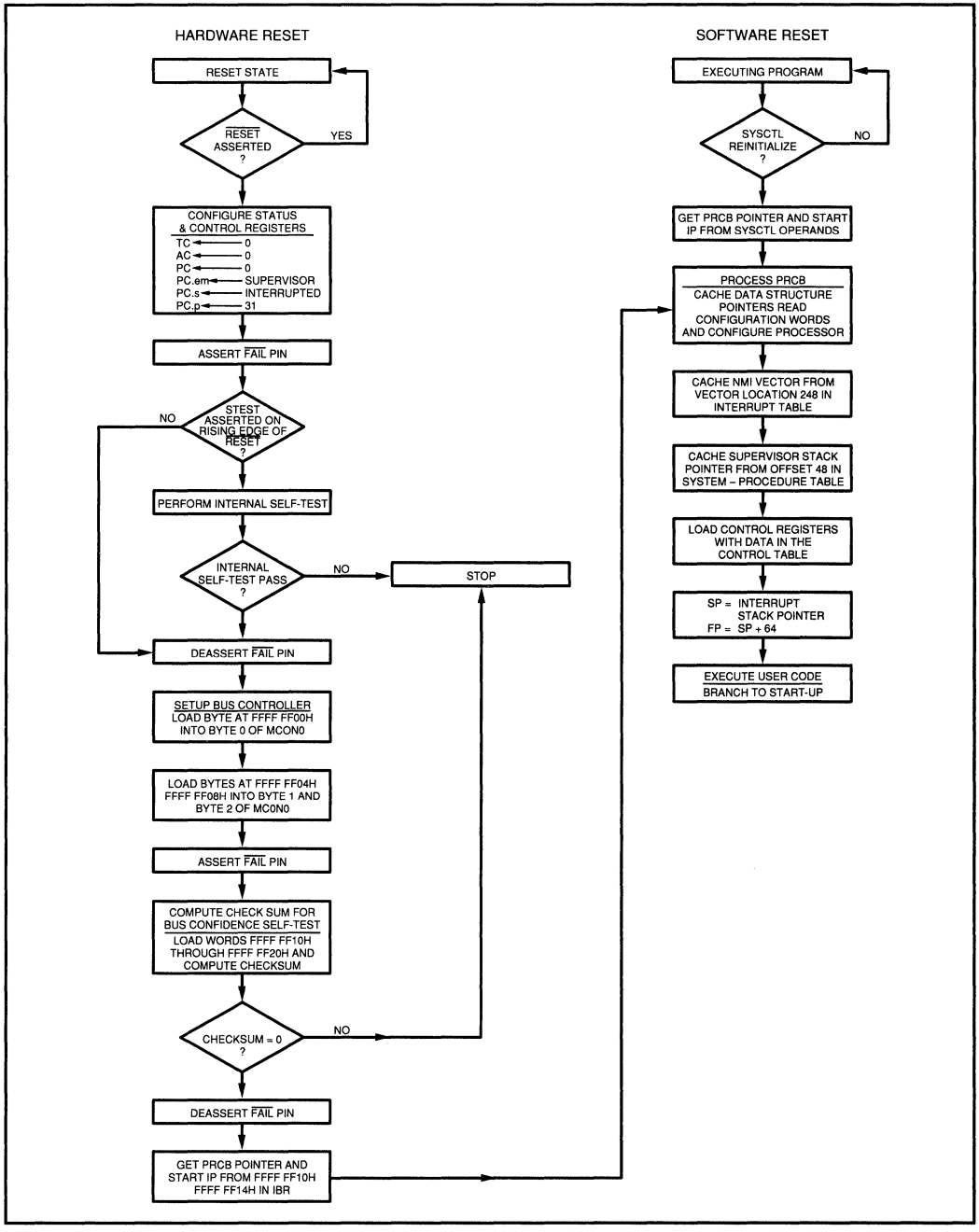


Figure 14-4. Processor Initialization Flow

Start-up Code Example

After initialization is complete, the user start-up 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.

The `startup.s` routine is presented first in Example 14-1. Example 14-2 presents 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 5 provide useful header files for configuring the bus controller and interrupt controller, respectively. Files from both Example 14-4; and 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:
        cmpibg      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
```

(cont.)

Example 14-1. Startup Routine (cont.)

```
_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
```

(cont.)

Example 14-2. Linker Directives File (cont.)

```
        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 that there will be two carry outs when */
cs2 = 0x0                       /*  _start and _rom_prcb are added with the processor's */
cs3 = 0x0                       /*  checksum algorithm since both addresses have most of */
cs4 = 0x0                       /*  the high order address bits set.  We put -2 here to */
cs5 = 0x0                       /*  reduce the checksum subtotal by two to remove the */
                                /*  addition of the two carries. */

cs6 = -( _rom_prcb + _start);   /*  this equation causes the checksum to go
                                    to zero. */
```

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 Check sum 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)
```

(cont.)

Example 14-3. Boot-up Data Declarations (cont.)

```
/*------------------------------------------------------------------------------ */
/* 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))

/* 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
```

(cont.)

Example 14-3. Boot-up Data Declarations (cont.)

```
/*-------------------------------------------------------------------------------- */
/* 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
```

(cont.)

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   0x40000000              /* 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 */
```

(cont.)

Example 14-3. Boot-up Data Declarations (cont.)

```
/*------------------------------------------------------------------------------ */
/* 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                       /* BPCON0 Breakpoint control reg 0 */
        .word   0                       /* BPCON1 Breakpoint control reg 1 */
        .word   0                       /* TC Trace Controls Initial Image */
        .word   0x00000001              /* BCON Bus Controller Mode  */
/*------------------------------------------------------------------------------ */
/* End boot_ca.s                                                                 */
/*------------------------------------------------------------------------------ */
```

(cont.)

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    */
```

(cont.)

14

Example 14-4. Bus Controller Header File (cont.)

```
/*-------------------------------------------------------------------------------- */
/* 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)
```

(cont.)

Example 14-5. Interrupt Controller Header File (cont.)

```
#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)
*/


/*---------------------------------------------------------------------------- */
/* 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)
```

(cont.)

Example 14-5. Interrupt Controller Header File (cont.)

```
/*------------------------------------------------------------------------------ */
/* 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 the generic hardware requirements for a system built around the 80960CA controller. The electrical characteristics of the 80960CA's interface to the external circuit is described in this section. The CLKIN,$\overline{\text{RESET}}$, STEST, $\overline{\text{FAIL}}$, $\overline{\text{ONCE}}$, VSS, and VCC pins are described in detail. The 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 execution rate and timing of the 80960CA controller. The clock input is internally divided by two, or used directly, to produce the external processor clock outputs, PCLK1 and PCLK2.

The state of the CLKMODE pin selects whether the input clock is in two-X mode, or one-X mode. When CLKMODE is tied to ground, or left floating, the CLKIN input will be internally divided by two to produce PCLK2:1 (Two-X Mode). When CLKMODE is pulled to a logic 1 (high), the CLKIN input will be used to create PCLK2:1 at the same frequency, using an internal phase-locked loop circuit (One-X Mode). The data sheet should be consulted for CLKIN specifications in either mode.

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 *80960CA Data Sheet*. The input capacitance for CLKIN 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 reduce the impedance of the power and ground connections to the chip, and reduce the transient noise induced by the current surges. The 80960CA controller is implemented in CHMOS IV technology. Unlike NMOS processes, the 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 80960CA employs 24 VCC and 24 VSS pins to ensure clean on-chip power distribution.

The 80960CA's power consumption depends mostly on frequency. It also depends on voltage and the capacitive bus load. (See the *80960CA Data Sheet).*

POWER AND GROUND PLANES

Power and ground planes must be used in 80960CA systems to minimize noise. The justification for these power and ground planes is the same as for the multiple VSS and VCC pins. Power and ground lines have inherent inductance and capacitance, therefore an impedance $Z=(L/C)^{1/2}$. The total characteristic impedance for the power supply can be reduced by adding more lines. This effect is illustrated in Figure 14-5, which shows that two lines in parallel have half the impedance of one. To reduce the impedance even further, the user should add more lines. In the limit, an infinite number of parallel lines, or a plane, results in the lowest impedance.

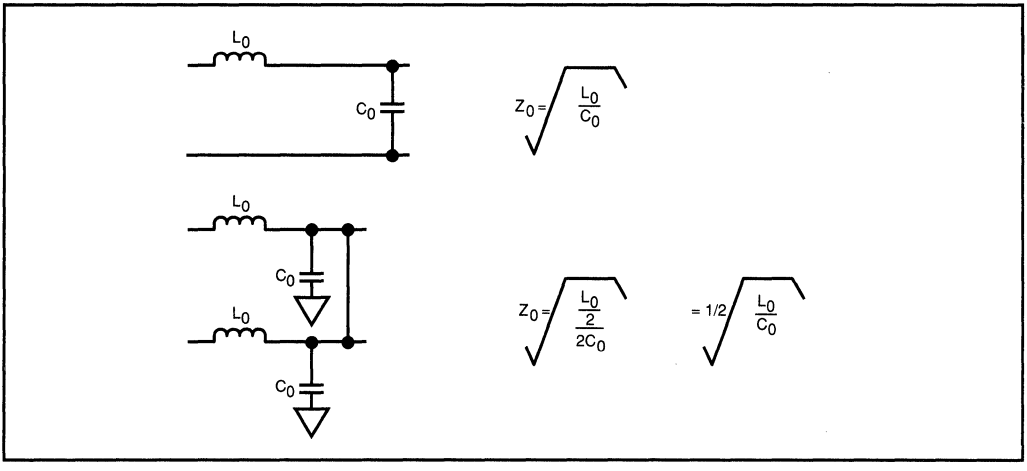


Figure 14-5. Reducing Characteristic Impedance

All power and ground pins must be connected to a plane. Ideally, the 80960CA is located at the center of the board to take full advantage of these planes, simplify layout and reduce noise.

DECOUPLING CAPACITORS

Decoupling capacitors placed across the device between VCC and VSS reduce voltage spikes by supplying the extra current needed during switching. These capacitors should be placed close to their devices because the inductance of connection lines negate their effect. Also, for this reason, the capacitors themselves 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 80960CA controller interfaces to its system through its pins. This section describes the general characteristics of the input and output pins.

OUTPUT PINS

All of the output pins on the 80960CA 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 both VCC and VSS). Each pin is capable of driving an appreciable external load. (The *80960CA Data Sheet* describes the drive capability of each pin and provides timing and derating information to calculate the output delays based on the pin loading.)

The output drivers on the 80960CA are specially designed to provide a uniform drive current over the entire range of operating temperatures and voltages. This feature eliminates the excess noise which is typically produced by output drivers under adverse operating conditions.

INPUT PINS

All of the 80960CA inputs are designed to detect TTL thresholds, providing compatibility with the vast quantity of available random logic and peripheral devices which use TTL outputs.

Most of the 80960CA's 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 the internal logic. If the value is not valid, it is possible for a bi-stable condition to be produced internally. The bi-stable condition is avoided by qualifying the synchronous inputs with the rising edge of PCLK2:1, or a derivative of PCLK2:1. (The *80960CA Data Sheet* specifies input valid setup and hold times relative to PCLK for the synchronized inputs.)

Table 14-3. 80960CA Input Pins

Synchronous Inputs			
D31:0	$\overline{\text{READY}}$	$\overline{\text{BTERM}}$	HOLD
Asynchronous Inputs (sampled by PCLK2:1)			
$\overline{\text{RESET}}$	$\overline{\text{XINT7:0}}$	$\overline{\text{NMI}}$	$\overline{\text{DREQ3:0}}$
Asynchronous Inputs (sampled by $\overline{\text{RESET}}$)			
STEST	$\overline{\text{ONCE}}$		CLKMODE

The inputs on the 80960CA 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 be held off until a following cycle. (The *80960CA Data Sheet* provides the setup and hold requirements relative to PCLK2:1 which will ensure recognition of an asynchronous input on a particular clock. The *80960CA 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. If the round-trip signal path delay is greater than the rise time or fall time of the signal, terminate the line. 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 80960CA, two methods of termination are attractive: A.C. and series. An A.C. 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 14-6. The resistor increases the rise and fall times of the signal 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. The series resistor should be placed 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.

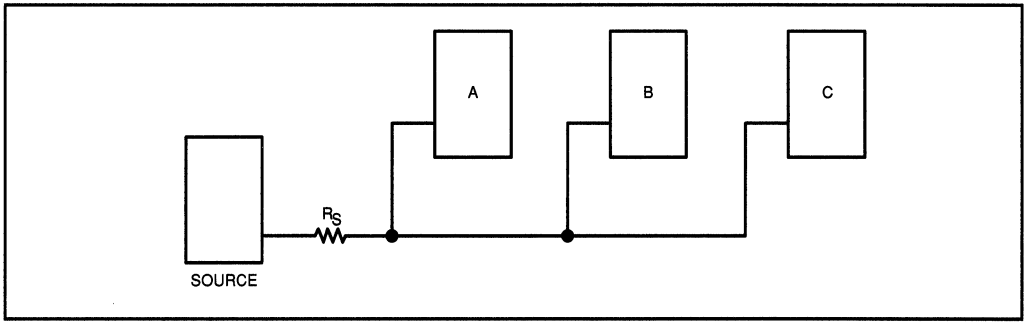


Figure 14-6. Series Termination

A.C. termination is effective in reducing signal reflection (ringing). This termination is accomplished by the addition of an RC combination at the signal's destination, as shown in Figure 14-7. While the termination provides no D.C. 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.

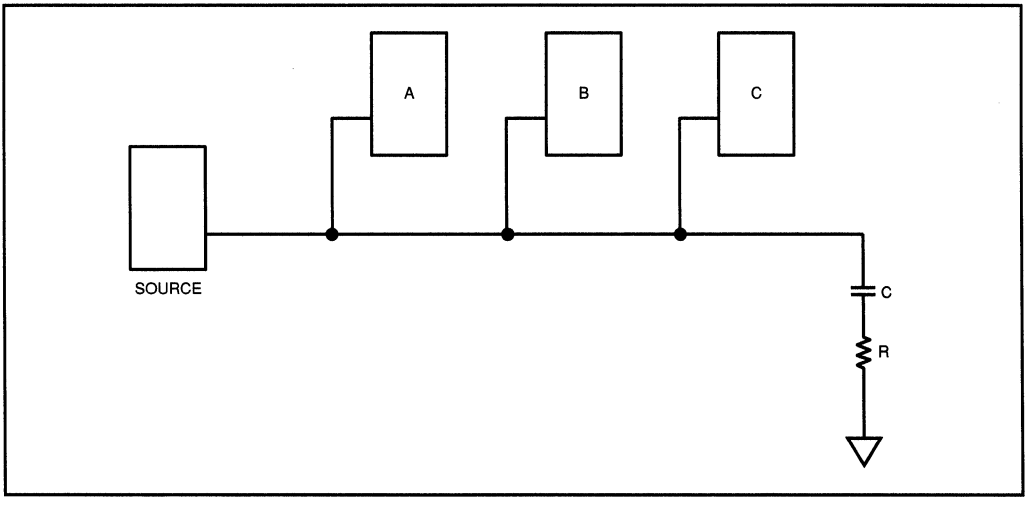


Figure 14-7. A.C. Termination

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:

- Observing the maximum rating for input voltage on I/O pins.
- Never applying power to an 80960CA pin or a device connected to an 80960CA pin before applying power to the 80960CA itself.
- Preventing 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 causing 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

There are two types of interference to consider 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:

- Running 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.
- Running ground lines between the lines of an address bus or a data bus if either of the following conditions exists:
  - The bus is on an external layer of the board.
  - The bus is on an internal layer but not sandwiched between power and ground planes that are at most 10 mils away.

- Avoiding closed loops in signal paths (Figure 14-8). Closed loops cause excessive current and create inductive noise, especially in the circuitry enclosed by a loop.

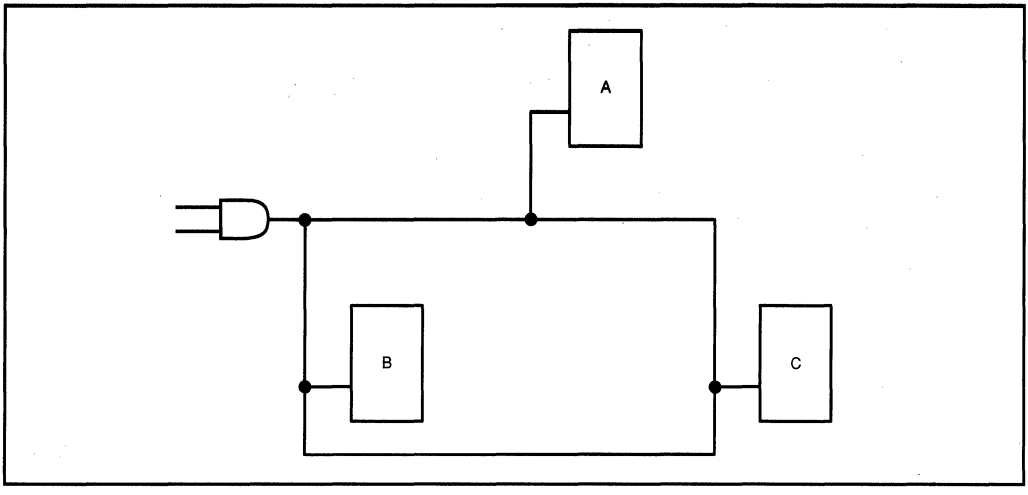


Figure 14-8. Avoid Closed-Loop Signal Paths

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:

- Separating signal lines so that capacitive coupling becomes negligible.
- Running a ground line between two lines to cancel the electrostatic fields.

APPENDICES PART III

80960CA Internal Architecture

A

APPENDIX A
80960CA INTERNAL ARCHITECTURE

This appendix describes the internal construction (microarchitecture) of the 80960CA core, and the features of the core microarchitecture which promote the performance and parallelism of the 80960CA.

OVERVIEW

The 80960CA is based on the 80960 core architecture. The core architecture defines the programming environment, basic interrupt mechanism, and fault mechanism for all 80960-based processors. The 80960CA's implementation of the core architecture is referred to as the C-series core. This core is a high-performance, and highly parallel implementation of the 80960 core architecture. The 80960CA integrates a bus controller, DMA controller, and interrupt controller around the core architecture (Figure A-1).

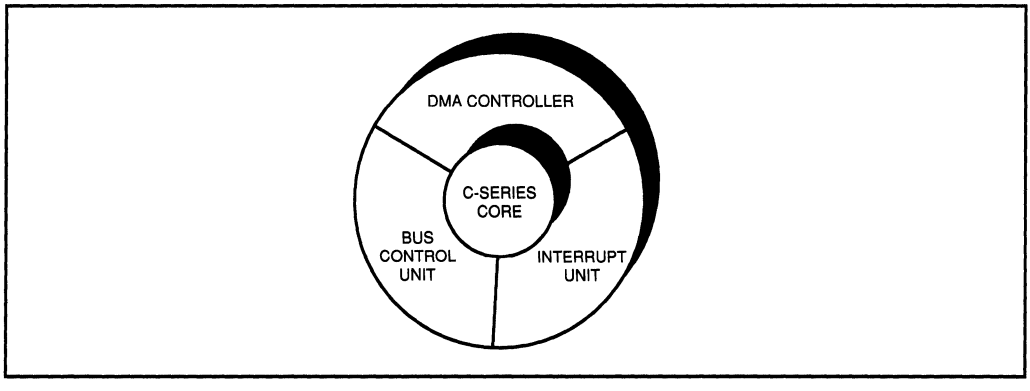


Figure A-1. 80960CA Core and Peripherals

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 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.

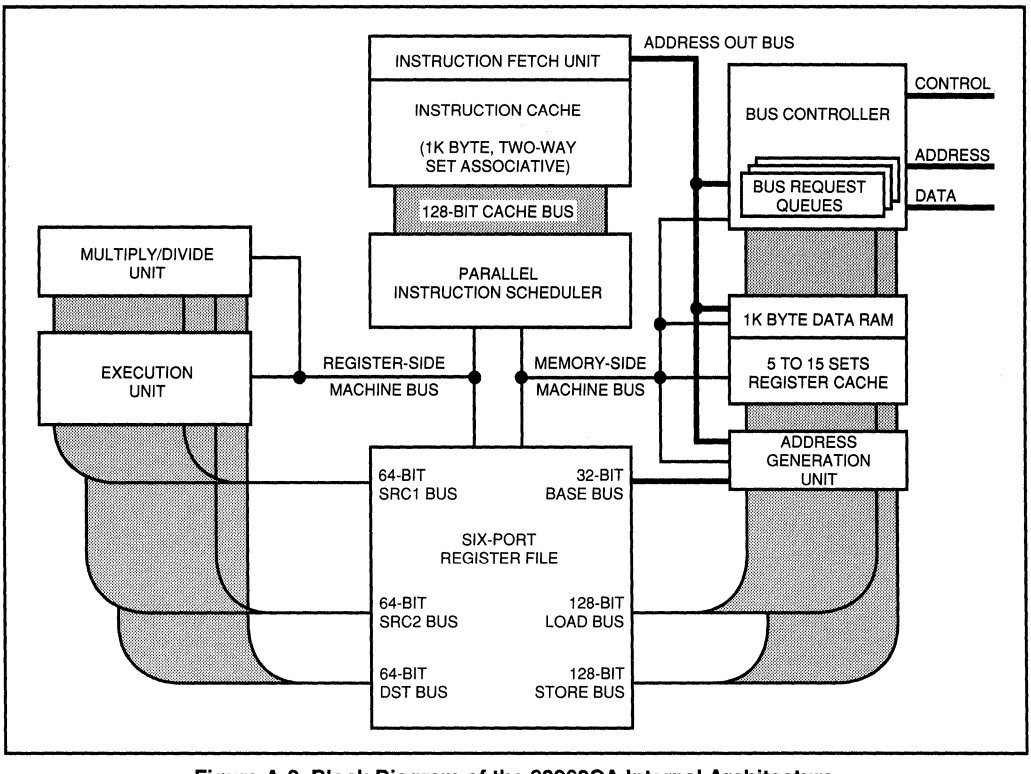


Figure A-2. Block Diagram of the 80960CA Internal Architecture

BASIC STRUCTURE OF THE 80960CA CORE

The 80960CA core contains the following main functional units:

- Instruction Scheduler
- Register File
- Execution Unit
- Multiply/Divide Unit
- Address Generation Unit
- Data RAM/Local Register Cache

Figure A-2 is a block diagram of the 80960CA. The heart of the processor is the instruction scheduler and register file. The other functional units of the core, referred to as coprocessors, interface to these scheduler and register file, connecting to either the register (REG) side or the memory (MEM) side of the processor. The instruction scheduler issues directives, via the REG and MEM interfaces, which target a specific coprocessor. That coprocessor then executes an express function virtually decoupled from the instruction scheduler (IS) and the other coprocessors. The REG and MEM data busses are used to transfer data between the common register file and the coprocessors.

The 80960CA is designed for expandibility 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 80960CA.

Instruction Scheduler

The *instruction scheduler* (IS) decodes the instruction stream and drives the decoded instructions onto the machine bus (the major control bus). It 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: the instruction fetch unit, the instruction cache, and the microcode ROM.

The *instruction fetch unit* provides the instruction scheduler with up to four words of instructions each cycle. It extracts instructions from the instruction cache, the microcode ROM, and its instruction fetch queue for presentation to the scheduler. The instruction fetch unit requests external fetch operations from the bus controller whenever a cache miss occurs.

The *instruction cache* is a 1 KByte, two-way set associative non-transparent cache. The cache delivers up to four instructions per clock to the instruction scheduler. The cache also allows inner loops of code to execute with no external instruction fetches, this maximizes the core's performance.

The 80960CA 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 RISC environment of the 80960CA. Unlike conventional microcode, the 80960CA microcode uses a RISC subset of the instruction set in addition to specific micro-instructions. The microcode, therefore, can be thought of as a RISC program containing operational routines for the complex instructions. When the instruction pointer references a microcoded instruction, the instruction fetch unit automatically branches to the appropriate microcode routine. The 80960CA 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. The decode stage calculates the next address used to fetch the next instruction from the instruction cache. Additionally, this stage starts decoding the instruction. The issue stage completes decoding the instruction and sends it to the appropriate execution unit. During the execute stage, the operation is performed and the result is returned to the register file.

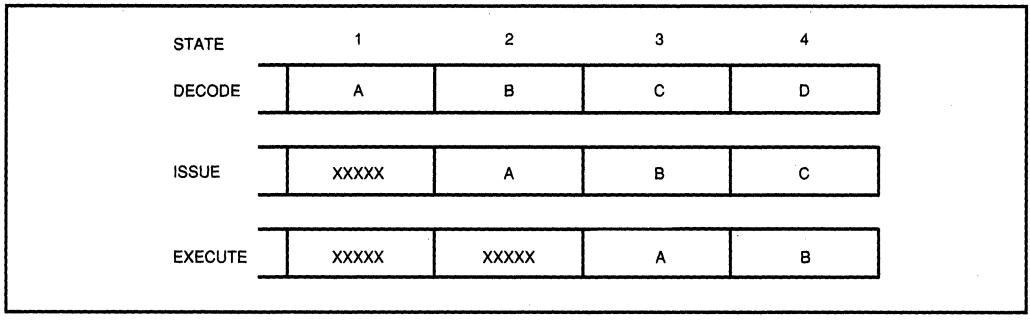


Figure A-3. Instruction Pipeline

The decode stage is when the instruction scheduler decodes the instruction and calculates the next instruction address. This could be a macro-instruction or micro-instruction address. It is either the next sequential address or the target of a branch. For conditional branches, the IS uses the condition codes or internal hardware flags to tell which way to branch. If the 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.

The issue stage is when the instructions are emitted or issued to the rest of the machine. The instructions are transmitted to the other units via the machine bus. The machine bus consists of three parts: 1) the REG format instruction portion, 2) the MEM format instruction portion, and 3) the CTRL format portion. Each part of the machine bus goes to the coprocessor that executes the appropriate instruction. The register file supplies the operands, and stores results for both the

REG and MEM format instructions. For this reason, the register file 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.

When an instruction is issued several things happen. First, the information is driven onto the machine bus. Then, the source operands are read and the resources needed to execute the instruction are checked to see if they are all available. If any resource needed by the instruction is busy (reserved by a previous incomplete instruction or already working on an instruction), the instruction is cancelled. The IS will then attempt to re-issue the instruction on the next clock so the same sequence of events will repeat. This mechanism which is used to manage the processors resources is called *resource scoreboarding*.

A specific form of resource scoreboarding is register scoreboarding. When the computation stage of an instruction takes more than one clock, the result registers are scoreboarded. A subsequent operation needing that particular register will be delayed until the multi-clock operation is completed. 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 busses. In this stage, the coprocessor has received operands from the register file and has recognized an opcode which targets and directs the actions of the coprocessor. Execution begins and a result is returned in this stage.

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 3-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

The Register File (RF) contains the 16 local and 16 global registers. The register file has six ports (Figure A-4), allowing parallel access of the register set by several 80960CA coprocessors. This parallel access results in an ability to execute one simple logic or arithmetic instruction, one memory operation (LOAD/STORE), and one address calculation per clock.

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 an address or address reduction operand to be simultaneously fetched by the Address Generation Unit. The wide load and store data paths enable up to four words of source data and four words of destination data to simultaneously pass between the RF and a MEM coprocessor in a single clock. The wide paths provide a high-bandwidth path between the data RAM and local register cache to implement high-speed calls, returns, and operations in data RAM. The wide paths also 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 busses and a single 64-bit destination bus. The source and result from different REG coprocessors can access the register file simultaneously using this bus structure. The 64-bit source and destination busses 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. Whenever a source register of an instruction is the same as the destination register of the previous instruction, the register bypassing mechanism is activated. 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 register file forwards the result data from the return bus directly to the source bus without reading the source register.

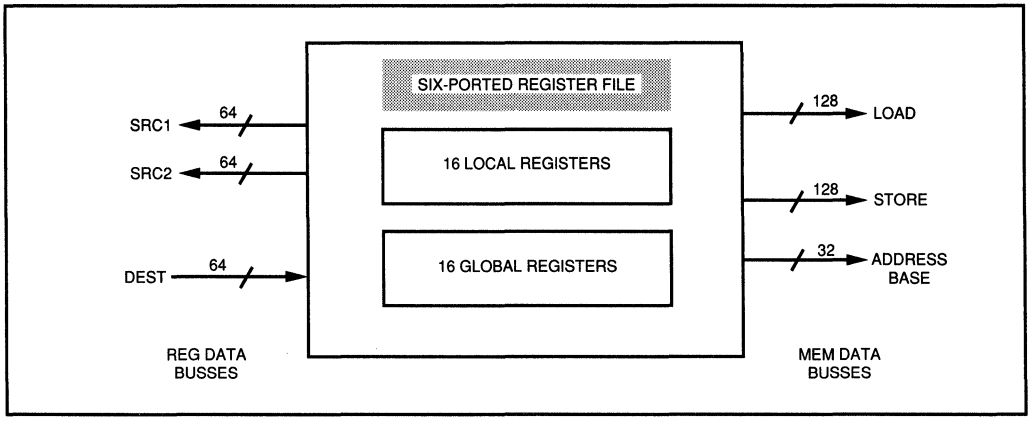


Figure A-4. Six-ported Register File

Execution Unit

The Execution Unit (EU) is the 32-bit arithmetic and logic unit of the 80960CA core. 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 the integer and ordinal arithmetic instructions, the logic and shift instructions, the move instructions, the bit and bit-field instructions, and the compare operations. The EU performs any arithmetic or logical instructions in a single clock.

Multiply/Divide Unit

The Multiply and Divide Unit (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 4 to 5 clocks. The MDU performs multiplies and divides in parallel with the main execution unit.

Address Generation Unit

The Address Generation Unit (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 is part of a 1.5 KByte block of on-chip Static RAM (SRAM). One KByte of this SRAM is mapped into the 80960CA'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.

Optimizing Code for the 80960CA

B

APPENDIX B
OPTIMIZING CODE FOR THE 80960CA

This appendix describes parallel instruction execution on the 80960CA, and describes assembly-language techniques for achieving the highest instruction-stream performance.

MICROARCHITECTURE REVIEW

At the center of the 80960CA core (Figure B-1) is a set of parallel processing units capable of executing multiple single-clock instructions every clock. To support this rate, the instruction scheduler can initiate (i.e. issue) up to three new instructions every clock. The other processing units execute the instructions independently, and in parallel, since each has access to multiple ports of the chip's six-ported register file.

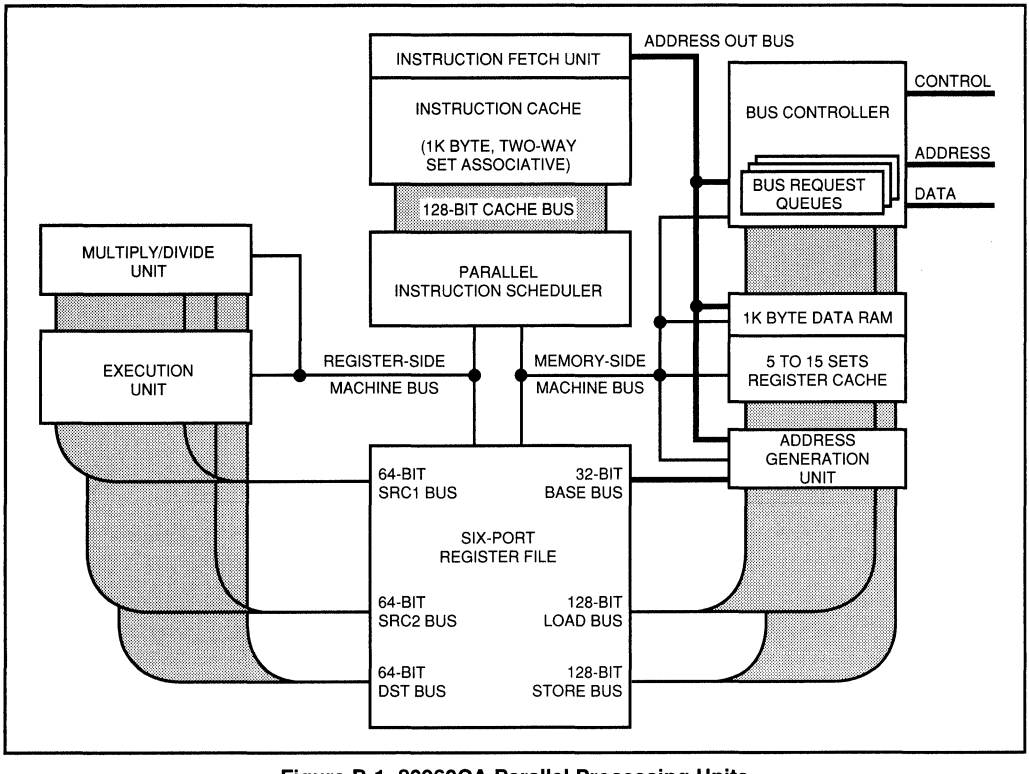


Figure B-1. 80960CA Parallel Processing Units

Parallel Issue

To keep the processing units busy, the instruction scheduler investigates a rolling quad-word group of unexecuted instructions every clock, and dispatches all instructions which may be executed in that clock. The scheduler can issue up to three instructions every clock to the processing units, and can sustain an issue rate of two instructions per clock.

To give the instruction scheduler the best chance of issuing instructions in parallel, the instruction cache is organized so that it can provide three or four instructions per clock to the scheduler. To keep the cost of a cache miss to a minimum, the instruction fetch unit constantly checks whether there will be a cache miss 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. Three units are attached to the Memory-side (MEM-side) of the machine, while two units are attached to the Register-side (REG-side) of the machine. The sixth parallel processing unit is the instruction scheduler, which directly executes the control instructions (the CTRL-side of the machine).

The MEM-side processing units are: the Bus Control Unit (BCU), Data-RAM (DR), and the Address Generation Unit (AGU). The BCU executes memory reads and writes for instructions which reference an operand in external memory. The DR handles memory reads and writes for instructions which reference on-chip data-RAM. The AGU executes the **lda** instruction, and assists address calculation for all loads and stores, and the **callx**, **bx**, and **balx** instructions. MEM-side instructions are dispatched over the MEM machine-bus.

The REG-side processing units are: the Multiply/Divide Unit (MDU), and the Execution Unit (EU). The MDU executes the multiply, divide, remainder, modulo, and extended multiply and divide instructions. The EU executes all other arithmetic, logical, shift, comparison, bit, bit field, move instructions and the **scanbyte** instruction. REG-side instructions are dispatched over the REG machine-bus.

The instruction scheduler executes CTRL-side instructions directly by modifying the next instruction pointer given to the instruction cache.

The processor supports instructions not directly executed by one of the parallel processing units by using an on-chip ROM. This ROM contains a sequence of simple (RISC) instructions for each complex instruction not directly executable in one of the parallel processing units. When a complex instruction is encountered by the scheduler, the appropriate ROMed sequence of RISC instructions is issued for execution. This sequence of instructions is called a micro-flow (μ), since when taken as a whole they perform a complex function, or macro.

Appendix I, Instruction Quick Reference, lists the machine type of each instruction as implemented on the 80960CA.

Optimizations

In general, the register file, instruction scheduler, cache and fetch unit will 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 the details of instruction execution on the 80960CA with the goal of instruction-stream optimization in mind. The *Instruction Stream Optimization* section describes specific code optimization techniques applicable to the 80960CA.

PARALLEL INSTRUCTION ISSUE

Every instruction begins execution after being *issued* by the instruction scheduler. It is the instruction scheduler's responsibility to keep the parallel processing units busy by issuing as many new instructions as possible every clock. To perform this task, the scheduler looks at the next three or four unexecuted words of the instruction stream every clock and sorts out which instructions it can issue in parallel. To achieve this parallelism, the scheduler detects to which "side" (REG, MEM, or CTRL) of the machine each instruction in the current quad-word group belongs, and ensures that there are no register dependencies between the instructions.

When the instruction scheduler issues a group of instructions, the appropriate parallel processing units acknowledge receipt of the instructions and begin executing them. However, register dependencies and resource dependencies could delay instruction execution. These interactions are managed transparently by the processor 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 80960CA's instruction scheduler can issue multiple instructions every clock when the instructions decoded in that clock can be executed by different sides of the machine. For example, an **add** instruction can begin in the same clock as a **ld** instruction since the addition is performed by the EU on the REG-side, while the load is executed by the BCU on the MEM-side of the machine. Furthermore, a branch can be issued in the same clock as the add and load since it is executed on the CTRL-side of the machine (3 instructions/clock).

Figure B-2 shows the paths that the instruction scheduler has available for dispatching each word of the rolling quad-word to the three sides of the machine. The scheduler is not implemented to fully exploit every possible combination of three instruction types in four consecutive words, as this would have been prohibitive, and many of the possible cases are meaningless.

Table B-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.

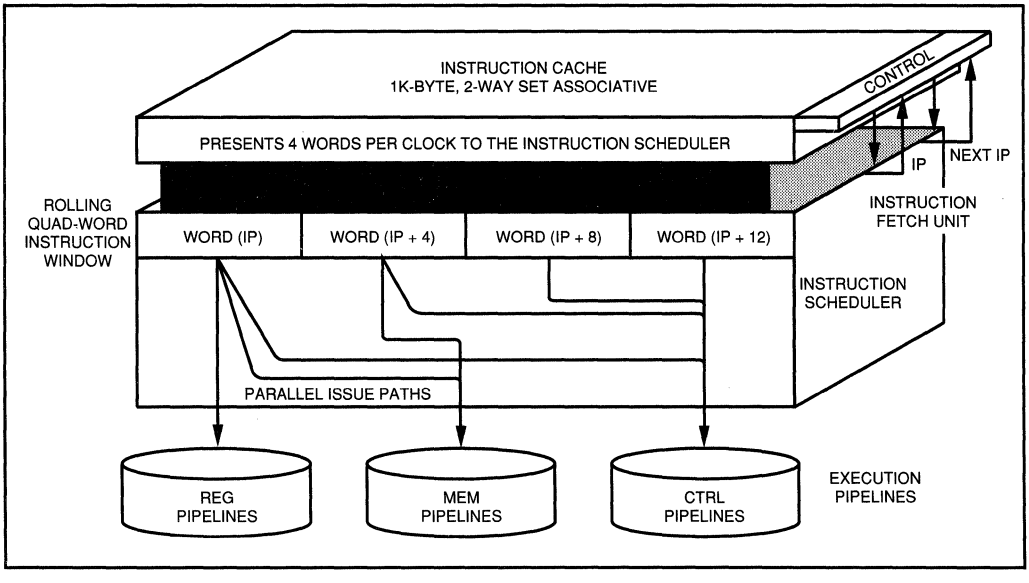


Figure B-2. Issue Paths

Table B-1. Machine Type Sequences Which Can Be Issued in Parallel

Machine-Type 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	REG-side followed by a CTRL instruction in the same rolling quad-word
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	MEM-side followed by a CTRL instruction in the same rolling quad-word
M x x C	MEM-side followed by a CTRL instruction in the same rolling quad-word

Instruction Independence

The scheduler also checks for register dependencies between instructions before issuing them in parallel. The scheduler will not issue a group of instructions if: (1) the same register is specified as a destination more than once, or (2) the same register is specified as a destination in one instruction and a source in a subsequent instruction. A single register may, however, be specified as a source in multiple instructions, or as a source in one instruction and a destination in a subsequent instruction. The 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, the store must
                          # wait for the addo to complete
```

or,

```
addo        g0, g1, g2    # g2 is a destination
ld          (g3), g2      # g2 is also a destination, the
                          # load
                          # must wait for the 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 the addo, and
ld          (g3), g0      # a destination for the load
```

In all cases of parallel issue of instructions, the instruction scheduler 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 moves to 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 target of a prior multi-clock instruction (e.g. a load) which is not completed, the instruction will be delayed. The scheduler will attempt 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 B-2 summarizes the conditions which cause a delay due to a scoreboarded register.

Table B-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	The condition codes in the arithmetic controls register are not valid. Correct branch prediction eliminates dead clocks due to condition code dependencies.

SCOREBOARDED RESOURCE

A scoreboarded resource will also thwart 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 B-3 lists cases which will cause an instruction to be delayed due to a scoreboarded resource.

The following section describes what happens to an instruction once it is issued to a processing unit.

Table B-3. Scoreboarded Resource Conditions

Condition	Description
BCU Queue Full	The Bus Controller queues are full, and the scheduler is attempting to issue a memory request.
MDU Busy	The Multiply/Divide Unit is busy executing a previously issued instruction, and the scheduler is attempting to issue another instruction for which the MDU is responsible.
DR Busy	The 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 will stall execution if a new request is issued to it, and it has not been allowed to return the data from a prior instruction. For example, if the DR and the BCU attempt to return results over the load bus in the same clock, the BCU will win the arbitration. This delays the DR's result by one clock. If, simultaneously, the scheduler is attempting to issue another instruction to the data RAM, the DR will stall the processor for one clock.

REGISTER SCOREBOARDING AND BYPASSING

To maintain the logical intent of the sequential instruction stream the 80960CA implements register scoreboarding and register bypassing. These mechanisms eliminate possible pipeline stalls due to parallel register access dependencies, and are described to provide an understanding how the processor operates. It is not necessary to perform any code optimizations to take advantage of this parallel support hardware.

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 scheduler issues an instruction which will take 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 33'rd 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. The register bypassing logic constantly monitors all register addresses being written, and all register addresses being read. If, in any clock, the same register is being read and written, the bypass logic routes the data coming in over the write port directly to the read port, instead of delaying the read.

Register scoreboarding and bypassing can be seen throughout the descriptions and examples in this appendix.

PARALLEL EXECUTION

Once the instruction scheduler issues a group of instructions, the appropriate processing units begin execution of the instructions, in parallel with all other processor operations. The following sections describe the pipelines of each unit and the execution times of the instructions which they process.

Execution Unit

The Execution Unit (EU) performs the arithmetic, logical, move, comparison, and bit and bit-field operations. The EU receives its instructions over the REG-machine bus, receives its source operands over the *src1* and *src2* buses, and returns its result over the *dst* bus.

The EU pipeline is shown in Figure B-3. In the clock that an EU instruction is issued, the unit latches the source operands and begins performing the operation. The instruction will complete and the result will be written to the destination register in the following clock. When an instruction immediately follows an EU operation which references the EU's destination register, the new instruction will not be issued in the same clock as the EU instruction. As seen in the figure, the new instruction will be issued in the clock following the EU operation.

The EU directly executes the instructions listed in Table B-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 B-3. EU Execution Pipeline

Table B-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 execution of the **shrdi** instruction, the processor executes a four clock micro-flow. See *Micro-flows* in this chapter.

Multiply/Divide Unit

The Multiply/Divide Unit (MDU) performs the multiplication, division, remainder and modulo operations. The MDU receives its instructions over the REG-machine bus, receives its source operands over the *src1* and *src2* buses, and returns its result over the *dst* bus. Once the instruction scheduler issues an MDU instruction, the unit performs its operations in parallel with all other execution.

The MDU pipeline for the **mulo** instruction is shown in Figure B-4. In the clock that the multiply is issued, the unit latches the source operands and begins the operation. The multiply will complete and the result will be 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 will not be issued until the clock in which the multiply result is returned. For example, an **addo** instruction which immediately follows a multiply, and references the destination of the multiply, will be delayed until the fourth clock after the multiply is issued. This five clock multiply latency is easily hidden, since four to eight instructions could be placed between the multiply and the 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		

Figure B-4. MDU Execution Pipeline

The MDU incorporates a one-clock pipeline so that the instruction scheduler can issue a new MDU instruction one clock before the previous result is written. For example, the back-to-back multiply throughput is four clocks per multiply versus a five clock multiply latency. Figure B-5 shows the execution pipeline for back-to-back multiplies, where the adjacent instructions do not have a register dependency between them. (This one-clock pipelining of MDU operations will 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
```

INSTRUCTION SCHEDULER	Issue	addo	mulo	— —	— —	— —	mulo	addo		
EU PIPELINE	Read src1, src2	g0, g1						g8, g9		
	Execute and Write dst		g2←g0+g1						g10←g8+g9	
MDU PIPELINE	Read src1, src2		g2, g3				g5, g6			
	Execute									
	Write dst							g4←g3*g4		

Figure B-5. MDU Pipelined Back-To-Back Operations

The MDU directly executes the instructions listed in Table B-5. The scheduler issues an MDU instruction in one clock. The table lists the length of the execution stage for each instruction (Latency). Subsequent instructions which are not dependent upon the MDU results will be 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 will improve throughput by one clock per instruction.

Table B-5. MDU Instructions

Mnemonic	Issue Clocks	Result Latency	Back-to-Back Throughput (AC.om = 1)	Back-to-Back Throughput (AC.om = 0)
muli **mulo** **emul**	1	5	4	5
divi	1	39	38	38
divo **ediv** **remi** **remo** **modi**	1	35	34	34

B

Data RAM

The on-chip data RAM is described in *Chapter 2, Programming Environment*. The data RAM is single-ported and 128-bits wide to support accesses up to one quad-load or quad-store in size every clock.

The DR 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 DR returns data over the 128-bit Load bus.

The one-clock DR pipeline for reads is shown in Figure B-6. When the instruction scheduler issues a load from the DR, the 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 destination of the load, can not execute in the same clock as the load. As shown in the figure, the instruction will be issued in the clock that the load data is returning.

Table B-6 lists the instructions executed directly using the DR. As seen in Figure B-6, if these instructions are issued back-to-back, they will execute at a one-clock sustained rate, with or without register dependencies.

```
addo   16, 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					
DR PIPELINE	AddressOut bus / St bus		g0		g7	g8				
	Ld bus			Quad g4←(g0)		Triple g8←(g7)	Quad g0←(g8)			

Figure B-6. The Data RAM Execution Pipeline

Table B-6. DR 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: For the offset, displacement, and indirect memory addressing modes. For other addressing modes, see the *Micro-flows* section of this chapter.

Address Generation Unit

The Address Generation Unit (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 written to a destination register in the case of an **lda** instruction, or is 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 instruction scheduler. 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

When an **lda** instruction is issued, the AGU returns the *efa* to the destination register in the following clock, for six of the nine 80960 addressing modes. An instruction which immediately follows the **lda**, and references the destination of the **lda**, will not be issued in the same clock as the **lda**. As shown in Figure B-7, the instruction will be issued in the clock that the **lda** is writing the destination register.

Table B-7 lists the **lda** addressing mode combinations directly executed by the AGU. As seen in the figure, if **lda** instructions are issued back-to-back using one of the addressing modes in the table, the instructions will 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+g16		g6←g4+g5			
AGU PIPELINE	Read over Base bus		g0		g7	g8		
	Execute and Write over Ldbus			g4←g0+g16		g8←(g7*4)+16	g0←g8+16	

Figure B-7. The lda Pipeline

Table B-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 the other memory addressing modes, see the *Micro-flows* section of this chapter.

EFA CALCULATIONS FOR OTHER OPERATIONS

When an instruction is issued which requires an effective address calculation (*efa*) 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 B-6), the load and store instructions begin immediately for these addressing modes with no delay for address generation. See the section in this chapter titled *Micro-flows* for a description of how the other addressing modes are handled.

Bus Controller

The Bus Control Unit (BCU) is described in *Chapter 10, Bus Controller*. The BCU 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 the on-chip request queuing, the BCU is capable of accepting a load or store from the instruction scheduler 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

Although the BCU executes the memory operations for load and store instructions, instruction fetches, micro-flows and DMA operations, its execution pipeline can be easily understood by looking at simple load and store requests. Figure B-8 shows the execution of a load instruction assuming that there were no prior requests stored in the BCU queues, and the worst case that the instruction following the load references the destination of the load.

```
ld      (g0), g1
addo    g1, g2, g3
```

INSTRUCTION SCHEDULER Issue	ld	— —	— —	addo					
AddressOut bus St bus	g0								
External Address Bus		g0							
BCU PIPELINE External Data Bus			(g0)						
Ld Bus				g1←(g0)					
Read src1, src2				g1, g2					
EU PIPELINE Execute and Write dst					g3←g1+g2				

Figure B-8. 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 had been advanced in the code such that it was separated from the instruction which used the data, the load delay could have been completely overlapped with the execution of other instructions, even when the system has wait states.

The execution of a store instruction would proceed as did the load, except that there would be no return clock, and no instructions could get stalled due to a scoreboarded register.

Table B-8 lists the instructions executed directly by the BCU. For each instruction that requires multiple reads on the external bus (e.g. 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 will execute at the bus bandwidth. Figure B-9 shows back-to-back loads being executed.

Table B-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: For the offset, displacement, and indirect memory addressing modes over an external bus with the following characteristics: $N_{XAD} = N_{XDD} = N_{XDA} = 0$, Burst On, Pipelining On, Ready Disabled. For other addressing modes, see the *Micro-flows* section of this chapter.

```
ld      (g0), g1
ld      (g2), g3
ld      (g4), g5
addo    g1, g6, g7
```

INSTRUCTION SCHEDULER	Issue	ld	ld	ld	addo					
BCU PIPELINE	AddressOut bus St bus	g0	g2	g4						
	External Address Bus		g0	g2	g4					
	External Data Bus			(g0)	(g2)	(g4)				
	LD Bus				g1← (g0)	g3← (g2)	g5← (g4)			
EU PIPELINE	Read src1, src2				g1, g6					
	Execute and Write dst					g7← g1+g6				

Figure B-9. Back-to-Back BCU Accesses

BCU QUEUES

To allow programs to issue load requests prior to when 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 loads, the source address, destination register number and load type is stored for each request. For stores, the destination address, store type and the store data is queued for each request. 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.

The 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.

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 instruction scheduler directly executes program-flow control instructions. Branches take two clocks to execute in the CTRL-pipeline, however, the instruction scheduler 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 B-9 lists the instructions which are directly executed by the instruction scheduler, without the aid of micro-flows. For information on other control-flow instructions, see *Micro-flows* later in this chapter.

Table B-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

UNCONDITIONAL BRANCHES

Figure B-10 shows the instruction scheduler issue stage, and the CTRL pipeline for the case where branches branch to branches, essentially disabling the instruction scheduler's ability to look ahead. The scheduler issues the branch in one clock, and the branch is executed in the next clock. The target of the branch is another branch, which the scheduler issues immediately. Hence, branch instructions have a two-clock sustained rate when issued back to back.

```
w: b    x
 ...
x: b    y
 ...
y: b    z
 ...
z: b    w
```

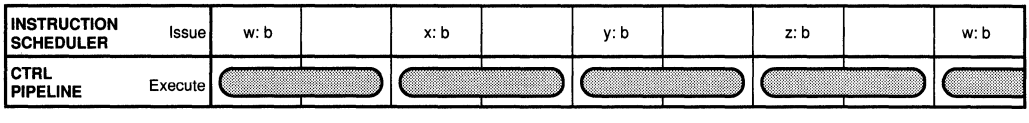


Figure B-10. CTRL Pipeline for Branches to Branches

Figures B-11, 12 and 13 show the instruction scheduler issue stage, and the CTRL pipeline for each case of possible branch lookahead detection by the instruction scheduler. Assuming that the scheduler can see four instructions every clock from the instruction cache, the branch can be in:

- the first executable group of instructions seen,
- the second executable group of instructions seen, or
- the third executable 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 *Parallel Instruction Issue*, earlier in this chapter.

Figure B-11 shows the cases where a branch, when it is first seen by the instruction scheduler, is in the first executable group of instructions. The instruction scheduler 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 instruction scheduler's ability to issue instructions occurs. On the next clock, the instruction scheduler 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. If the branch had been the first instruction in the group (i.e. the branch was a branch target), no other instructions would have been issued in parallel with the branch, and it would have taken a full two clocks to execute (as seen in Figure B-10.).

```
    b       x
    ...
x:  addo    g0, g1, g2
    lda     2(g3), g4
    b       y
    ...
y:  addo    g5, g6, g7
    lda     2(g8), g9
```

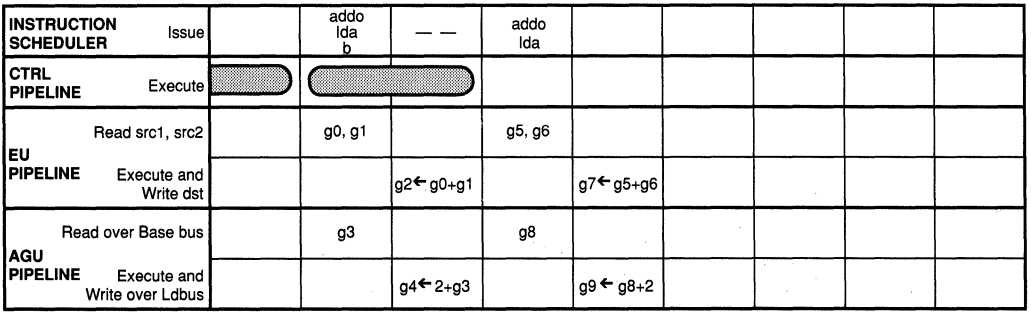


Figure B-11. Branch in First Executable Group

Figure B-12 shows the case where a branch, when it is first seen by the instruction scheduler, is in the second executable group (B) of instructions of the rolling quad-word, not the first executable group (A) which is about to be issued. The instruction scheduler 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 instruction scheduler's ability to issue instructions. On the next clock, the instruction scheduler 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 that the branch was executing. Hence, this branch could be said to have taken zero clocks to execute.

```
    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			

Figure B-12. Branch in Second Executable Group

Figure B-13 shows the case where a branch, when it is first seen by the instruction scheduler, 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 instruction scheduler 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 a no break in the instruction scheduler's ability to issue instructions. On the clock following the issuance of group B, the instruction scheduler issues a new group of instructions from the branch target.

```
    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		

Figure B-13. 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 instruction scheduler sees the branch instruction. For example, a conditional branch which immediately follows a compare instruction can not be allowed to *complete* execution until the result of the comparison is known. However, the processor will *begin* to execute the branch based upon the branch prediction indication given by the programmer for that branch.

When one or more executable groups of instructions 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, and thus it knows 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 B-13.

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 will not be valid when the instruction scheduler sees the branch, and a guess will have to be made. 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 was 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 instruction scheduler's issue stage if it is in the same executable group as the instruction which modified the condition codes; or one clock if it is 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 3 or 4 consecutive opwords to the instruction scheduler on every clock. This capability is critical to the ability to dispatch multiple instructions from the 80960'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 1K-byte 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 3 or 4 words to the instruction scheduler. 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 3 or 4 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 B-10 describes the fetch decision.

Table B-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 are different from actual instruction stream loads in two ways: the load destination, and how load data is buffered.

First, the load destination of an instruction fetch is the Fetch Unit queue, not the register file. Since the fetch data goes directly from the BCU to the fetch queue and the instruction scheduler, the scheduler can issue the fetched instructions during the clock after they are read from external memory.

Second, the BCU buffers fetch data differently than a regular load instruction to reduce fetch latency. Instead of buffering four words of instructions before sending the data to the fetch unit, the BCU sends each word as it is received over the bus. If the fetches are coming from 8-, or 16-bit memory, the BCU will collect 32-bits before sending the word to each fetch unit.

Figure B-14 shows the execution of a two-word fetch that resulted from a cache miss. At the end of the clock in which instructions would have been 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 that it is received from the memory system, and the instruction scheduler 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).

```
    b       y
    ...
y:  addo    g0, g1, g2 ←Cache Miss
    subo    g3, g4, g5
```

INSTRUCTION SCHEDULER	Issue		y: — —	— —	— —	— —	addo	subo		
CTRL PIPELINE	Execute		Cache Miss							
BCU PIPELINE	AddressOut bus St bus			Fetch Address						
	External Address Bus				A					
	External Data Bus					D addo	D subo			
	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	

Figure B-14. Fetch Execution

B

If the fetch request was the result of a prefetch decision, the instruction scheduler would not be stalled unless it needed an instruction from the prefetch request.

If the processor is executing straight-line code which always misses the cache, the instruction scheduler will only be 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 will limit the application's performance.

Cache Replacement

The data fetched as a result of cache miss is written to the cache when, and if, the fetched data is requested by the instruction scheduler. This optimization keeps prefetched data which was never executed 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 instruction scheduler requests one or more instructions which are in the queue, the fetch unit satisfies the request from the queue. If the queue entry that the scheduler requested contained 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 80960CA's parallel processing units are able to directly execute about half of the 80960 instruction encodings. The processor services the remaining complex encodings by executing a sequence of simple instructions from an on-board ROM.

The instruction sequences stored in the ROM are written to utilize the parallel processing units to perform the required function as fast as possible. The micro-flows use the instructions described in the 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 can not 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 the execution times associated with each instruction.

Detection

To keep the support of micro-flowed instructions from impacting the depth of the processor's speed or pipeline depth, complex instructions are detected in the clock that they are fetched. This information becomes part of the instruction encoding which is stored in the Instruction Fetch Unit queue, and or Instruction Cache.

Invocation

The invocation of a micro-flow for a complex instruction 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 instruction scheduler is issuing one group of instructions, parallel decode circuitry is checking to see if a μ instruction will be the *next* executable instruction (Figure B-15). If so, the opwords presented to the Instruction Scheduler in the *next* clock will come from the location in the on-chip ROM that contains the micro-flow for detected complex instruction. The Instruction Scheduler actually never attempts to issue a complex encoding. The encodings are detected when the opword is fetched, and trapped-out during the clock that they are presented to the scheduler.

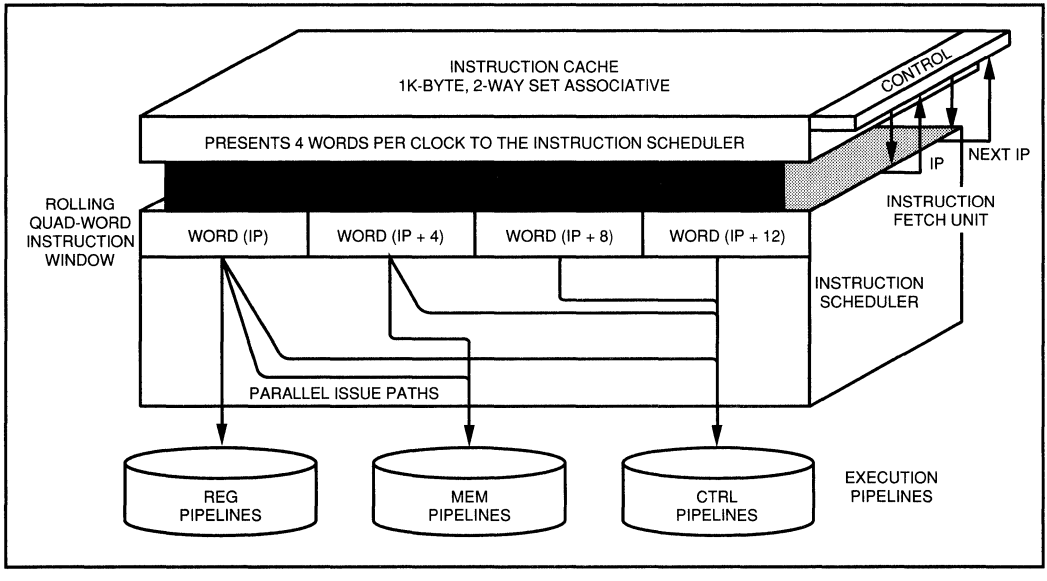


Figure B-15. Micro-flow Invocation

Although there are generally no clocks lost for switching to a micro-flow, 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 the on-chip ROM causes a one-clock break in the Instruction Scheduler'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 will not see a one-clock performance loss even when the lookahead logic is defeated. Furthermore,

micro-flows return to general execution with no overhead, and back-to-back micro-flows will not incur the one-clock defeated lookahead penalty.

Execution

When micro-flows execute, they consume the Instruction Scheduler. From the first clock through the last clock of a micro-flow the scheduler is typically issuing two instructions per clock. No instructions will be issued in parallel with the first or last instructions of a micro-flow. Therefore, the performance of the micro-flowed instructions is described by the number of clocks taken to issue instructions.

The following sections describe the performance of the micro-flowed instructions by functional group. *Appendix I* provides a quick reference of the entire instruction set.

DATA MOVEMENT

The 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.

The **movt** and **movq** instructions each take 2 clocks to execute.

The **lda** instruction takes 2 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. The **lda** instruction using the *disp(IP)* addressing mode takes 4 clocks to execute and can be issued in parallel with an instruction of machine type R. The AGU executes the **lda** directly for all other addressing modes.

The load and store instructions are summarized in Tables B-11, and 12. The number of clocks shown is the *additional* number of issue clocks that are consumed for address calculation prior to the load or store being issued to the BCU, or the DR. These instructions can be issued in parallel with an instruction of machine type R. To find the result latency of the BCU or the DR, see the appropriate section earlier in this appendix.

Table B-11. Load Micro-flow Instruction Issue Clocks

	The following load instructions consume **n** additional issue clocks for address calculation prior to 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: The *offset*, *disp*, and *(reg)* memory addressing modes incur no address calculation overhead. See the *Bus Controller* and *Data-RAM* sections of this appendix.

Table B-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: The *offset*, *disp*, and *(reg)* memory addressing modes incur no address calculation overhead. See the *Bus Controller* and *Data-RAM* sections of this appendix.

B

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

The **scanbit**, **spanbit**, **extract** and **modify** instructions are executed as micro-flows. Table B-13 lists their execution times. For these instructions, the instruction scheduler issues **n**-clocks of instructions in place of the single-word 80960 Instruction encoding, where **n** is shown in the table.

Table B-13. Bit and Bit Field Micro-flow Instructions

Mnemonic	Execution Clocks
scanbit	1
spanbit	2
extract	4
modify	3

BYTE OPERATIONS

The **scanbyte** instruction is directly executed by the EU parallel processing unit.

COMPARISON

The **test*** instructions are implemented with a micro-flow. Execution time depends upon the validity of the condition codes and the settings of the prediction bits. When the condition codes are valid, or the prediction bit is set correctly, the **test*** instructions will take one issue clock if the correct result of the instruction is a 1, and two issue clocks if the correct result is a 0. Otherwise, the instructions will take three issue clocks to execute.

BRANCH

The compare_and_branch, extended branch, branch_and_link, and extended branch and link instructions are implemented with micro-flows.

The **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.

The **bal** instruction takes two issue clocks to execute, assuming a cache hit.

The **bx** and **balx** instructions are summarized in Table B-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 an instruction of machine-type R.

Table B-14. bx and balx Performance

	The following instructions consume **n** issue clocks prior to the target code being 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	2	3	6
balx	2	3	6

Notes: The times are shown assuming instruction cache hits, and a DR-based link target for the **balx** instruction.

CALL AND RETURN

The procedure call, return, and system procedure call instructions are implemented as micro-flows.

The **call** instruction consumes four issue clocks when the target is cached and there is an available register cache location. When a frame spill is required, an additional 36 issue clocks are consumed in a zero-wait state system prior to the target code beginning 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 will directly impact call speed, while wait states in the frame stores will be decoupled from internal execution by the BCU queues.

The **ret** instruction consumes four issue clocks when the target and the previous register set are both cached. When a frame fill is required, an additional 36 issue clocks are consumed in a zero-wait-state system prior to the target code beginning 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 will directly impact return speed.

The **calls** instruction consumes 20 issue clocks if the call is to a supervisor procedure. If the call is to a non-supervisor procedure, the **calls** takes 19 issue clocks. These times assume an available register cache location and a cached target. During the **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 will be 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.

CONDITIONAL FAULTS

The **fault*** instructions are implemented with micro-flows. These instructions require one issue clock if the prediction bit was correct and no fault occurred. If the prediction bit was 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

The **mark** and **fmark** instructions are implemented with micro-flows. The **mark** instruction takes one issue clock if no trace fault is signaled. If a trace fault is signaled, or the **fmark** instruction is executed, the processor switches to the trace fault handler.

ATOMIC

The atomic instructions are implemented with micro-flows. The **atadd** and **atmod** instructions each take 5 issue clocks to execute with an idle bus in a zero-wait state system. Wait states in the memory accessed by these instructions will directly affect execution speed.

PROCESSOR MANAGEMENT

The processor management instructions implemented as micro-flows include: **modpc**, **modtc**, **modac**, **syncf**, **flushreg**, **sdma**, **udma** and **sysctl**.

The **modpc** instruction requires 10 clocks to execute. The **modac** instruction requires 11 clocks to execute. The **modtc** instruction requires 15 clocks to execute.

The **syncf** instruction takes one issue clock if there are no possible outstanding faults. Otherwise, the instruction locks the instruction scheduler until it is certain that no prior instruction that could fault, will fault.

The **flushreg** instruction requires 14 clocks to execute, plus 23 clocks for each frame that is flushed. Wait states in the memory being written will affect this instruction's performance.

The **sdma** instruction executes in 15 clocks. The **udma** instruction executes in 4 clocks.

The **sysctl** instruction timings are listed in Table B-15. The table lists the times assuming a zero wait-state memory system.

Table B-15. sysctl Performance

Message	Message Type	Issue Clocks
Request Interrupt	00H	20-30
Invalidate Cache	01H	13
Configure Cache	02H	20 + bus activity
Reinitialize	03H	195 + bus wait states
Load Control Register Group	04H	33 + 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 80960CA's 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 80960CA.

Note that the examples in this section are constructed to isolate particular optimization tricks for illustration. In general, every example could be further optimized by applying several techniques instead of one.

Advancing "Long" Operations

A few operations on the processor 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 (< 1 clock) if they are sufficiently separated from the instructions that use their results.

LOADS AND STORES

Separate load instructions from the instructions that use the load data. Also 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 will slow down the load. Reorder to issue the load first. The example shows a simple change that saved one clock from a five clock loop.

Example B-1. Overlap Loads (Checksum)

```
loop:
        ldob        (g0), g1
        addo        g1, g2, g2
        cmpinco     g0, g3, g0
        bl.t        loop
```

```
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		|	
3		|	
4	addo		bl.t
5	cmpinco		|
6		ld	

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 the instructions that use their results. Also remember to use shift instructions to replace multiplication and division by powers of 2. The following example shows overlapping pointer math and a comparison with the multiply time in a simple multiply-accumulate loop.

Example B-2. Overlap MDU Operations (Multiply-Accumulate)

```
loop:
        ld          (g0), g2
        ld          (g1), g3
        muli        g2, g3, g4
        addi        g4, g5, g5
        addo        4, g0, g0
        addo        4, g1, g1
        cmpobl.t    g0, g6, loop
```

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	

```
opt_loop:
        ld      (g0), g2
        ld      (g1), g3
        muli    g2, g3, g4
        addo    4, g0, g0
        cmpo    g0, g6
        addo    4, g1, g1
        addi    g4, g5, g5
        bl.t    opt_loop
```

Execution (from DR):

Clock	REGop	MEMop	CTRLop
1		ld	
2		ld	
3	muli		
4	|addo		
5	|cmpo		
6	|addo		
7	|		bl.t
8	addi		|
9		ld	

B

ADVANCING COMPARISONS

Where possible, instructions which change the condition codes should be separated from the 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, the 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 will not take extra clocks since the condition codes are known when the branch is encountered.

In a situation where the comparison and a branch can not be separated to achieve a performance advantage, use the combined compare_and_branch instructions. This will likely lead to faster execution since the two instructions are encoded in a single word. Not only does this code economy provide another space in the cache, but the instruction scheduler may be able to see the upcoming branch earlier since it's encoded in a the same opword as the comparison.

Unroll Loops to Use all the 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 will be used for the same operations if the algorithm can be done 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 B-3. Unroll Loops (Checksum)

```
        -- initialize --
loop:
        ldob        (g0), g1
        addo        g1, g2, g2
        cmpinco     g0, g3, g0
        bl.t        loop
        ret
```

```
                -- 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	
2		|	
3		|	
4	addo		bl.t
5	cmpinco		|
6		ldob	

Execution:

Clock	REGop	MEMop	CTRLop
1		ldob g1	
2	cmpinco	|	bge.f
3	addo g4	|	|
4		ldob g4	
5	cmpinco	|	bl.t
6	addo g1	|	|
7		ldob g1	

B

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 (RM, RMC, or MC). For example, the checksum loop is repeated with a another clock eliminated from code reordering for parallel issue.

Example B-4. Order for Parallelism (Checksum)

```
        -- initialize --
loop:
        ldob        (g0), g1
        addo        g1, g2, g2
        cmpinco     g0, g3, g0
        bl.t        loop
        ret
```

```
                -- 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		ldob	
2		|	
3		|	
4	addo		bl.t
5	cmpinco		|
6		ldob	

Execution:

Clock	REGop	MEMop	CTRLop
1	addo g4	ldob g1	bge.f
2	cmpinco	|	|
3		ldob g4	
4	cmpinco	|	bl.t
5	addo g1	|	|
6	addo g4	ldob g1	

Migrating from Side to Side

Since the 80960CA can sustain execution of two instructions per clock, an attempt should be made to start instructions in two of the three pipelines each clock. Parallelism can be increased by moving an instruction from executing in a unit which has become a critical path to a unit with available clocks. The AGU performs shifts, additions and moves that can replace EU operations. The literal addressing mode, in combination with EU or AGU operations, provides some freedom in deciding which side will load constants into registers. Remember to use the addressing modes that the AGU executes directly (machine type M, not μ).

Table B-16 lists several conversions that can move an instruction from either the EU, or the MDU, to the AGU. Example B-5 exploits the **lda** instruction to increase the performance of a 3x3 low-pass filter of an image by approximately 30 percent.

Table B-16. Creative Uses for the lda Instruction

Operation		**Done as an 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 addo 5, g2, g2	# shift/add combination	lda 5[g1 * 4], g2
mov g0, g1	# register move	lda (g0), g1

Example B-5. Change the Type of Instruction Used (3x3 Lowpass Mask)

$$Y[\ ] = X[\ ] \circledast M[\ ]$$

$$M[\ ] = \begin{pmatrix} \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \\ \frac{2}{16} & \frac{4}{16} & \frac{2}{16} \\ \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \end{pmatrix}$$

B

Example B-5. Change the Type of Instruction Used (3x3 Lowpass Mask) (cont.)

```
# 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
```

```
# 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
```

Example B-5. Change the Type of Instruction Used (3x3 Lowpass Mask) (cont.)

```
# 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

# 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
```

```
# 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
  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
```

B

Example B-5. Change the Type of Instruction Used (3x3 Lowpass Mask) (cont.)

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		

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	

Branching Optimizations

Conditional branches will execute faster if the actual branch direction is correctly predicted using the 80960 branch prediction bits on conditional instructions. Both conditional and unconditional branch-target code will 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). The branches, and 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 will improve throughput, especially when the comparison related to the conditional instruction can not be separated from the test. When the prediction is correct, branches will generally execute in parallel with other execution. If prediction is not correct, the worst case branch time for cached execution is still 2 clocks.

Although prediction bits will most likely be set to gain maximum throughput, different strategies can exist for setting the prediction bits. For example, a code sequence which is dominated by a jungle of comparisons and conditional branches might see large differences between the of execution time of the fastest path to slowest path. The prediction bits could be set to provide the best average throughput, to ensure the fastest worst case execution, or to minimize the deviation between slowest and fastest times.

BRANCH TARGET ALIGNMENT

Since the instruction scheduler is able to see 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, branch targets which are aligned will 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 will pay-off.

The instruction scheduler will take 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. The example 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 the 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 B-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

The **bal** instruction takes one or two 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 the **bal** to reduce the code size of a critical procedure.

When porting optimized algorithms originally written on other processor architectures, the code will often be expanded in a straight-line fashion due the branch speed penalties of the original target and the lack of on-chip caching. On the 80960CA 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 prodigious 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 will generally execute 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 it is not capable of executing two instructions per clock due to memory or resource dependencies, keep unrolling the loop and pipelining operations until the cache is full. In general, for a loop which iterates many times and performs memory operations, unrolling until all registers are used and/or the cache is full will increase performance.

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 will 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 THE ON-CHIP REGISTER CACHE

The 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 80960CA's register cache.

Since the register-cache/data-RAM partition is programmable, 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. Keeping the register cache depth to a minimum frees-up the most on-chip data RAM for variable caching.

There are some situations where the **flushreg** instruction can be used to 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, but may in fact cause a greater total number of spills to occur than would have otherwise occurred without the premature flush.

The **flushreg** instruction could also be used to 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 will reduce interrupt latency within that code piece since there is no possibility of the interrupt's frame spill getting lodged behind slow memory operations.

Although the usage of this premature flush tactic is very application specific, it probably always makes sense to flush the register cache at the beginning of the application's main loop (i.e., after all initialization).

UTILIZING THE ON-CHIP DATA RAM

On every clock, 128-bits of data can be loaded from the DR or stored to the DR. This is a 528 Mb/s 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 can be used to dramatically increase the performance of critical application algorithms. Locations in 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 the data RAM: the coefficients for a continuously operating filter (e.g. FIR), and/or standard DMA descriptor templates from which run-time descriptors are built.

SUMMARY

Table B-17 summarizes the 80960CA code optimization tactics presented in the previous sections. Figure B-16 is a copy of the execution pipeline template used to create the pipeline examples in this appendix.

Table B-17. 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 the processor executes the loop with two instructions per clock, the bus is saturated with quad operations, there are no registers left, or the loop does not fit in the cache.
Order for Parallelism	Alternate REG-side instructions with MEM-side instructions to they may be issued in parallel.
Migrate the Operation	Move EU and MDU operations to the AGU, or vice versa, to enable parallelism.
Use Branch Prediction	Set the 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 the loop does not fit in the cache, use branches, branch-and-links or calls to compress the code size so it fits. Use code size optimization instructions (e.g. **cmpobe**) where possible.
Use Data RAM	Use the 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								

Figure B-16. Execution Pipeline Template

Considerations for Writing Portable Code

C

APPENDIX C
CONSIDERATION FOR WRITING PORTABLE CODE

This appendix describes the parts of the 80960CA 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 80960 architecture.

80960 ARCHITECTURE

The 80960CA is an implementation of the *80960 core architecture*. All 80960 family products are based on the core architecture definition. An 80960-based product, such as the 80960CA, 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:

- The programming environment (global and local registers, literals, processor state registers, data types, memory addressing modes, etc.)
- The implementation-independent instruction set
- The procedure call mechanism
- The mechanism for servicing interrupts and the interrupt and process priority structure
- The mechanism for handling faults and the implementation-independent fault types and subtypes

Implementation-specific features are one or all of the following:

- 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 80960CA which are implementation dependent are described below. The parts of the 80960CA which are not described below are part of the core architecture.

ADDRESS SPACE RESTRICTIONS

Some properties of the address space are implementation-specific for the 80960CA.

Structures in Reserved Memory

All addresses in the range FF00 0000H to FFFF FFFFH are reserved by the 80960 architecture. Any uses of reserved memory are implementation specific. The 80960CA, 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 80960 architecture. Code which relies on structures in reserved memory is not portable to all 80960-based products.

Internal Data RAM

Internal data RAM is mapped to the first 1 KByte of the 80960CA's address space (0000H - 03FFH). The data RAM is an implementation-specific feature. High performance, supervisor-protected data space, and the locations assigned for DMA and interrupt functions are special features which are implemented in the data RAM on the 80960CA. Code which relies on these special features is not directly portable to all 80960 implementations.

Instruction Cache

The 80960 architecture allows instructions to be cached on-chip in a non-transparent fashion. This means that the cache may not detect modification of the program memory by loads, stores, or alteration by external agents. (See *Chapter 2, Programming Environment.*) Each implementation of the 80960 architecture which uses an integrated instruction cache must provide a mechanism to purge the cache, or in another way, force consistency between external memory and the 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 80960 implementations.

A 1 KByte instruction cache is integrated on the 80960CA. The 80960CA's instruction cache does not detect modification of the external program memory. This instruction cache is purged using the system control (**sysctl**) instruction. The **sysctl** instruction is specific to the 80960CA implementation.

Data and Data Structure Alignment

Implementations of the 80960 architecture are not required to handle loads and stores to non-aligned addresses. Code which generates non-aligned addresses, therefore, is not object-code compatible with all 80960 product implementations. The 80960CA, as an implementation-specific feature, automatically handles non-aligned load and store requests. (See *Chapter 10, Bus Controller.*)

The 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 80960 architecture.

EXTENDED REGISTER SET

The 80960 architecture defines a way to address 32 additional internal registers in addition to the 16 global and 16 local registers. (See *Chapter 2, Programming Environment.*) Depending on the implementation, these registers can have different functions. On the 80960CA, 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 80960 architecture.

RESERVED LOCATIONS IN REGISTERS AND DATA STRUCTURES

Some fields in registers and data structures are defined as reserved locations. A reserved field may be used by future implementations of the 80960 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 80960 architecture defines a comprehensive instruction set. Code which uses only the architecturally-defined instruction set is object-level portable to other implementations of the 80960 architecture. Some implementations may favor a particular ordering of code 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 80960 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 implementation of the 80960 architecture.

Implementation-Specific Instructions

Most of the 80960CA's instruction set is defined by the core architecture. Several instructions are specific to the 80960CA. 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 also 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 all 80960-based products.

INTERRUPT REQUESTS AND POSTING

The 80960 architecture defines the mechanism for servicing interrupts. This includes the definition of priorities, the structure of the interrupt table, and the 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 80960CA's interrupt controller manages the external interrupt pins and the internal DMA sources. The interrupt controller features, the external interrupt pins, and the NMI pins are specific to the 80960CA implementation. Code which configures the interrupt controller or in other ways interacts with interrupt requestors is not directly portable to other 80960 implementations. Interrupts are requested in software with the **sysctl** instruction on the 80960CA. 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. The 80960CA, for example, posts hardware-requested interrupts internally in the IPND register in order to minimize latency. Application code which expects interrupts to be posted in the interrupt table is not object-code portable to all 80960-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 80960-based product is initialized is implementation dependent. For the 80960CA, pointers to data structures, configuration information, and a first instruction pointer are loaded from external memory at initialization. The 80960CA defines the initialization boot record, the process control block, and the 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 80960 implementations.

OTHER IMPLEMENTATION-SPECIFIC FEATURES OF THE 80960CA

The following sections describe additional implementation-specific features of the 80960CA. These features do not relate directly to the portability of application code.

Data Control Peripherals

The DMA controller, bus controller, and interrupt controller are implementation-specific extensions to the core architecture. The operation, setup, and control of these units is not a part of the core architecture. Other implementations of the 80960 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 80960CA provides an operation-unaligned fault for detecting non-aligned memory accesses. Future 80960 implementations which generate this fault will assign the same fault type and subtype number to the fault.

External System Requirements

The external system requirements for the 80960CA is 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 80960CA implementation.

Instruction Encoding Reference

D

APPENDIX D
INSTRUCTION ENCODING REFERENCE

This appendix describes the encoding format for instructions in the 80960CA. Included is a description of the four instruction formats and how the addressing modes relate to these formats.

GENERAL INSTRUCTION FORMAT

The 80960 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 opcode field of the instruction. All instructions are one word long and begin on word boundaries. The MEM format instructions are encoded in one of two sub-formats, MEMA or MEMB. The MEMB encoding of the MEM-format permits an optional second word to hold a displacement value. The following sections describe the fields in the instruction word for each format.

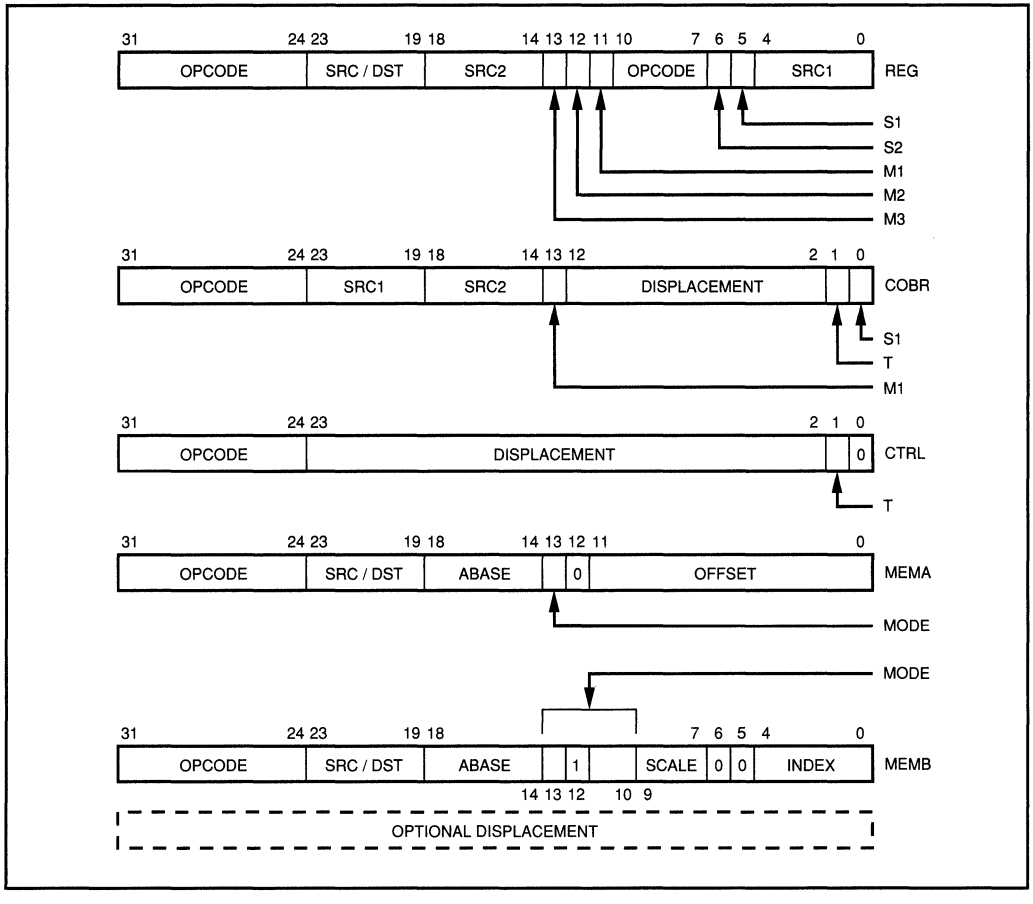


Figure D-1. Instruction Formats

D

REG FORMAT

The REG format is used for operations that are performed on data contained in the global, local, or special function registers. Most of the 80960's instructions use this format.

The opcode for the REG instructions is 12 bits long (3 hexadecimal digits) and is split between bits 7 through 10 (low opcode) and bits 24 through 31 (high opcode). For example, the opcode for the **addi** instruction is 591H. Here, 59H is contained in bits 24 through 31 and 1H is contained in bits 7 through 10.

The *src1* and *src2* fields specify source operands for the instruction. The operands can be global or local registers, literals, or special-function registers. The mode flags (*MI* 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. Table D-1 shows the relationship between the mode flags, the special-purpose flags, and the *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		

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.

Note: sf0, sf1 and sf2 are the only special-function registers implemented on the 80960CA.

The *src/dst* field can specify either a source operand or a destination operand or both, depending on the instruction. Here again, the 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	NA	Literal	sf0 .. sf31

Note: NA means not allowed

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 a literal, or a destination-only operand that is a special function register.

COBR FORMAT

The COBR format is used primarily for compare-and-branch instructions. (The test-if instructions also use this format.) The opcode field for this format is 8 bits (two hexadecimal digits).

The *src1* and *src2* fields specify source operands for the instruction. The *src1* field can specify a global or local register, a literal, or a special-function register, as determined by mode flag (*M1*) and special-purpose flag (*S1*). The encoding of the *src1* field is the same as is shown in Table D-1. The *src2* field can only specify a local or global register.

The *T* flag supports branch prediction for conditional instructions. If the *T* flag is set to 0, the condition being tested is likely to be true; if the flag is 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 that the processor goes to as a result of a comparison. The displacement field can range from -2^{10} to (2^{10} -1). To determine the IP of the target instruction, the processor converts the displacement value to a byte displacement (i.e., multiplies the value by 4). It then adds the resulting byte displacement to the IP of the current instruction.

Note: To allow labels to be used in the assembly-language version of the COBR format instructions, the 80960 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:

displacement = (*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. The **ret** instruction also uses this format. The opcode field for this format is 8 bits (two hexadecimal digits).

The target address for a branch is specified with the displacement field in the same manner as the COBR format instructions. The displacement field specifies a word displacement (also a signed, two's complement number) that can range from -2^{21} to 2^{21} -1. The processor ignores the displacement field for the **ret** instruction.

The *T* flag performs the same prediction function for the CTRL format instructions as it does for the 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.

There are two MEM-format encodings: MEMA and MEMB. The MEMB encoding offers the option of including a 32-bit displacement (contained in a second word) to the instruction. Bit 12 of the first word of the instruction determines whether the MEMA (clear) or MEMB (set) encoding is used.

For both encodings the opcode field is 8 bits long. The *src/dst* field specifies a global or local register. For load instructions, the *src/dst* field 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. The 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

Notes: In the address computations above, a field in parentheses (e.g., (abase)) indicates that the value in the specified register is used in the computation. The use of a reserved encoding causes an invalid-opcode fault to be generated.

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 of from 0 to 4096 to be loaded into a register.

For the register-indirect-with-offset addressing mode (*mode* field is set to 10_2), the value in the *offset* field 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
- register indirect with displacement
- register indirect with index
- register indirect with index and displacement
- index with displacement
- IP with displacement

The *abase* and *index* fields specify local or global registers, the contents of which are used in the address computation. When the *index* field is used in an addressing mode, the processor automatically scales the value in the index register 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: The use of a reserved encoding causes an invalid-opcode fault to be generated.

For the IP with displacement mode, the value of the displacement field plus 8 is added to the address of the current instruction.

INSTRUCTION REFERENCE BY OPCODE

This section lists the instruction encoding for each 80960CA instruction. The 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 (**atadd**, **atmod**).
						COBR Format
—	—	0	—	0	x	*src1* and *src2* are both global or local registers
—	—	1	—	0	x	*src1* is a literal, *src2* is a global or local register
—	—	0	—	1	x	*src1* is a special function register, *src2* is a global or local register
—	—	1	—	x	0	reserved
						COBR Format and CTRL Format
—	—	x	—	x	1	The outcome of the conditional test is predicted to be true.
—	—	x	—	x	0	The outcome of the conditional test is predicted to be false.

Table D-6. REG Format Instruction Encodings

Opcode	Mnemonic	Opcode (11 - 4)	src/ dst	src2	Mode			Op-code (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*

Table D-6. REG Format Instruction Encodings (cont.)

Opcode	Mnemonic	Opcode (11 - 4)	src/ dst	src2	Mode			Op-code (3-0)	Special Flags		src1
		31.....24	23.....19	18.....14	13	12	11	10.....7	6	5	4.....0
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*
61:0	**atmod**	0110 0001	*src/dst*	*mask*	M3	M2	M1	0000	S2	S1	*src*
61:2	**atadd**	0110 0001	*src/dst*	*src*	M3	M2	M1	0010	S2	S1	*dst*
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	*mask*	*src*	M3	M2	M1	0000	S2	S1	*src/dst*
65:1	**extract**	0110 0101	*bitpos*	*len*	M3	M2	M1	0001	S2	S1	*src/dst*
65:4	**modtc**	0110 0101	*mask*	*src*	M3	M2	M1	0100	S2	S1	*dst*
65:5	**modpc**	0110 0101	*src*	*mask*	M3	M2	M1	0101	S2	S1	*src/dst*
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*

D

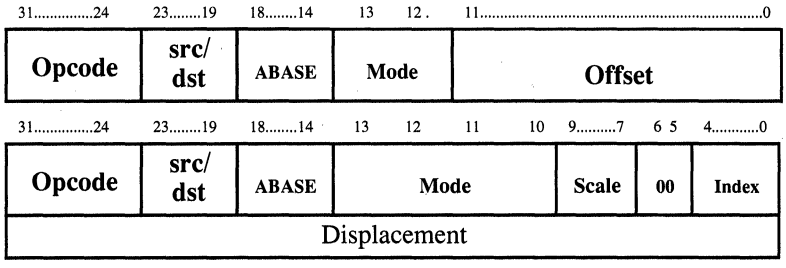
Table D-7. COBR Format Instruction Encodings

Opcode	Mnemonic	Opcode	src1	src2	M	Displacement	T	0
		31.....24	23.....19	18.....14	13	12 2	1	0
20	**testno**	0010 0000	*dst*		M1		T	S1
21	**testg**	0010 0001	*dst*		M1		T	S1
22	**teste**	0010 0010	*dst*		M1		T	S1
23	**testge**	0010 0011	*dst*		M1		T	S1
24	**testl**	0010 0100	*dst*		M1		T	S1
25	**testne**	0010 0101	*dst*		M1		T	S1
26	**testle**	0010 0110	*dst*		M1		T	S1
27	**testno**	0010 0111	*dst*		M1		T	S1
30	**bbc**	0011 0000	*bitpos*	*src*	M1	*targ*	T	S1
31	**cmpobg**	0011 0001	*src1*	*src2*	M1	*targ*	T	S1
32	**cmpobe**	0011 0010	*src1*	*src2*	M1	*targ*	T	S1
33	**cmpobge**	0011 0011	*src1*	*src2*	M1	*targ*	T	S1
34	**cmpobl**	0011 0100	*src1*	*src2*	M1	*targ*	T	S1
35	**cmpobne**	0011 0101	*src1*	*src2*	M1	*targ*	T	S1
36	**cmpoble**	0011 0110	*src1*	*src2*	M1	*targ*	T	S1
37	**bbs**	0011 0111	*bitpos*	*src*	M1	*targ*	T	S1
38	**cmpibno**	0011 1000	*src1*	*src2*	M1	*targ*	T	S1
39	**cmpibg**	0011 1001	*src1*	*src2*	M1	*targ*	T	S1
3A	**cmpibe**	0011 1010	*src1*	*src2*	M1	*targ*	T	S1
3B	**cmpibge**	0011 1011	*src1*	*src2*	M1	*targ*	T	S1
3C	**cmpibl**	0011 1100	*src1*	*src2*	M1	*targ*	T	S1
3D	**cmpibne**	0011 1101	*src1*	*src2*	M1	*targ*	T	S1
3E	**cmpible**	0011 1110	*src1*	*src2*	M1	*targ*	T	S1
3F	**cmpibo**	0011 1111	*src1*	*src2*	M1	*targ*	T	S1

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



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									

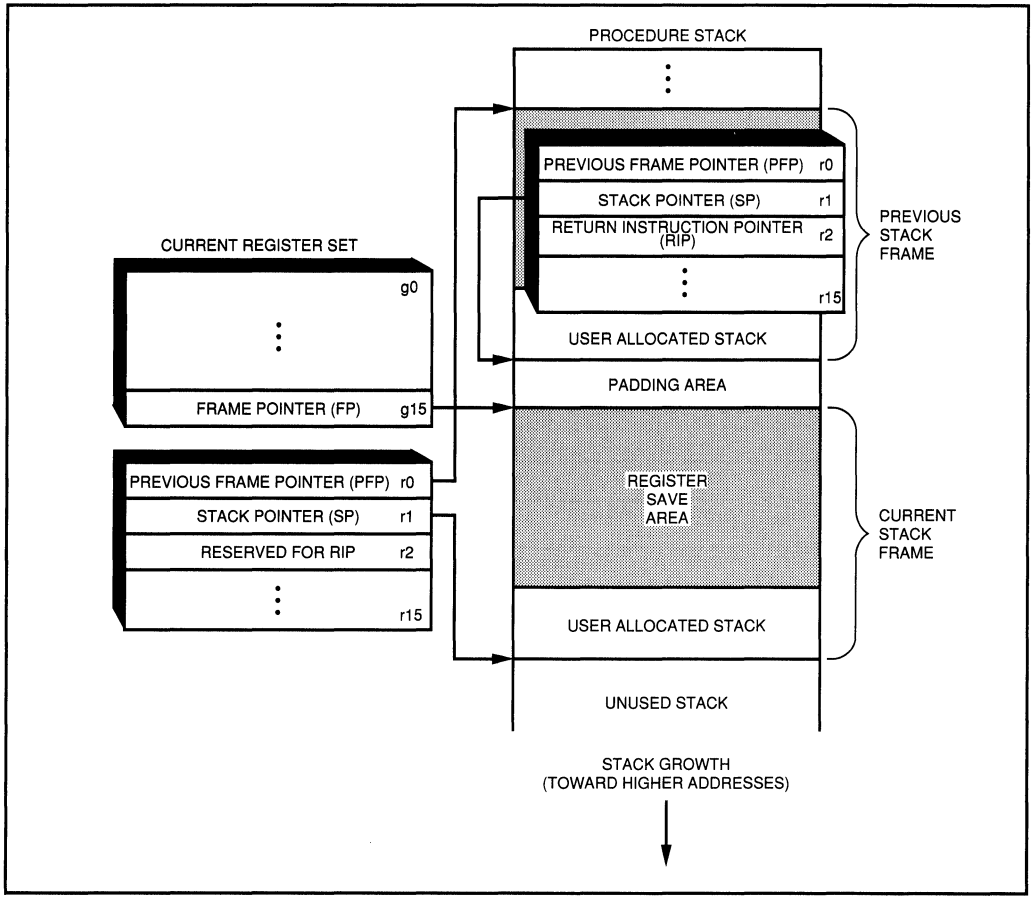
Opcode	Mnemonic	Opcode	Mnemonic
80	**ldob**	98	**ldl**
82	**stob**	9A	**stl**
84	**bx**	A0	**ldt**
85	**balx**	A2	**stt**
86	**callx**	B0	**ldq**
88	**ldos**	B2	**stq**
8A	**stos**	C0	**ldib**
8C	**lda**	C2	**stib**
90	**ld**	C8	**ldis**
92	**st**	CA	**stis**

Register and Data Structure Reference

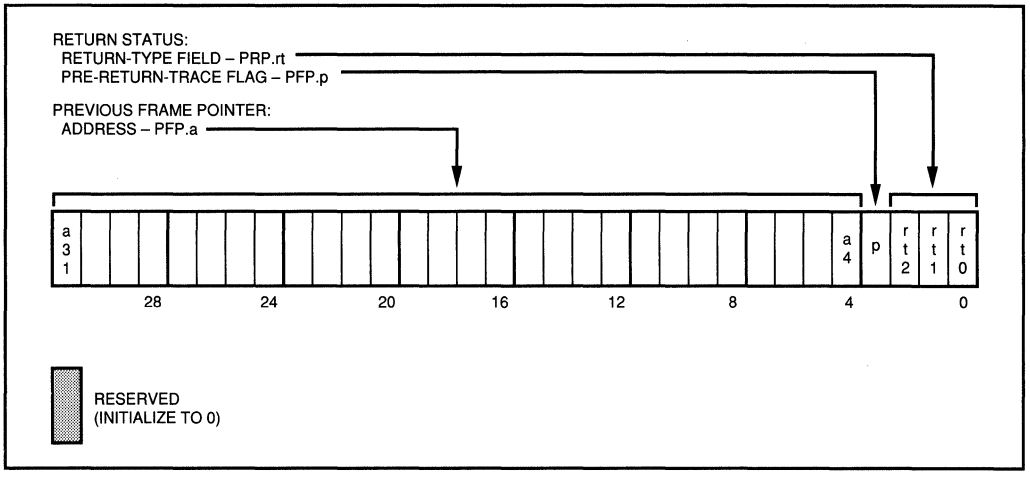
E

APPENDIX E
REGISTER AND DATA STRUCTURE REFERENCE

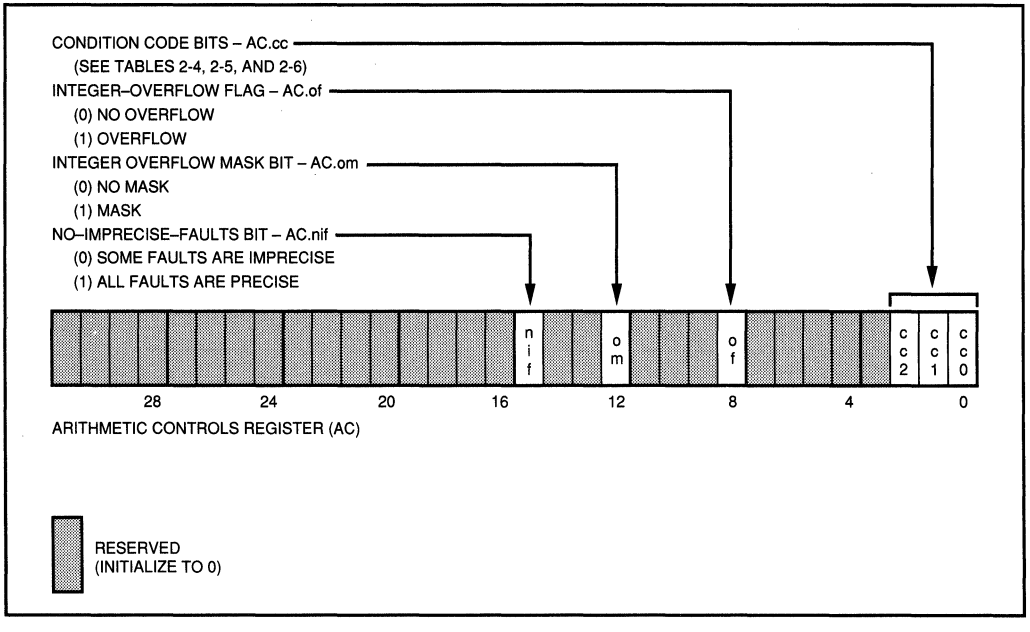
GLOBAL AND LOCAL REGISTERS



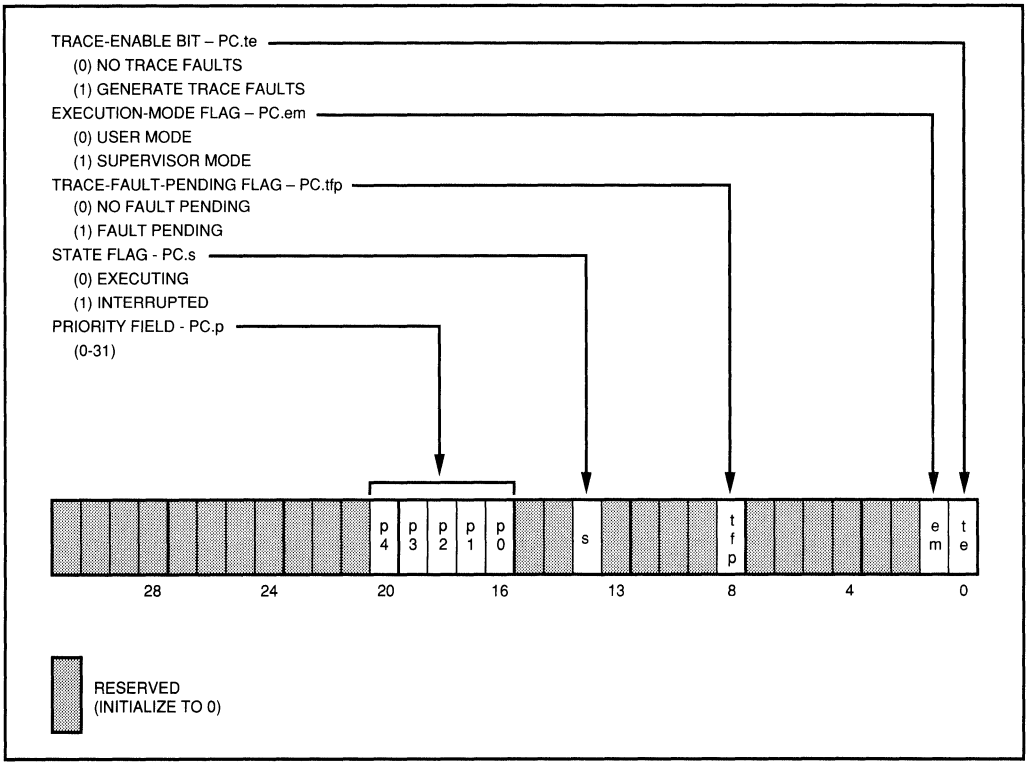
PREVIOUS FRAME POINTER (r0)



ARITHMETIC CONTROLS (AC) REGISTER

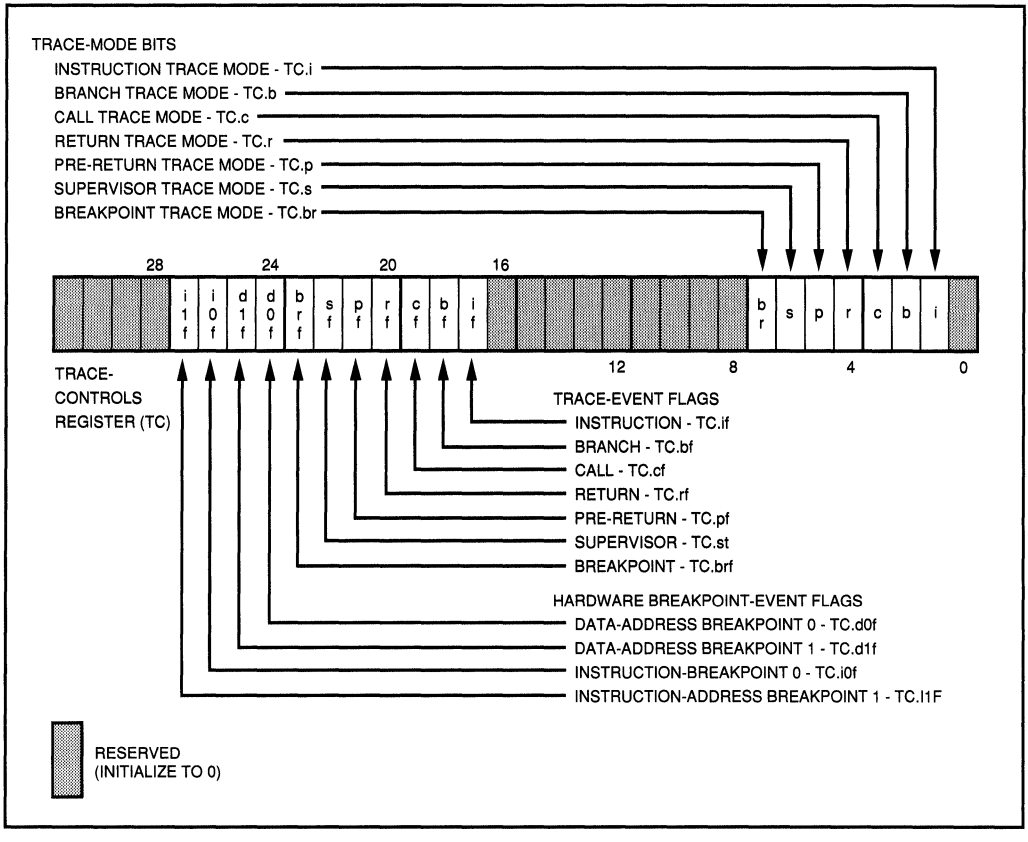


PROCESS CONTROLS (PC) REGISTER

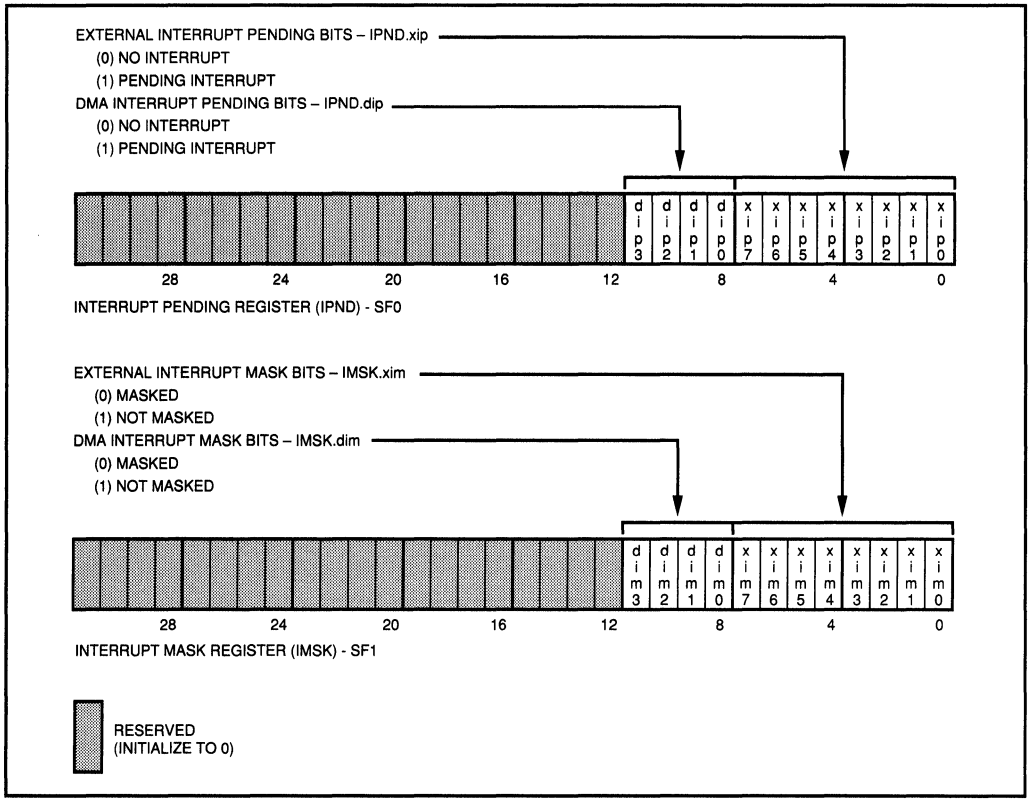


E

TRACE CONTROLS (TC) REGISTER

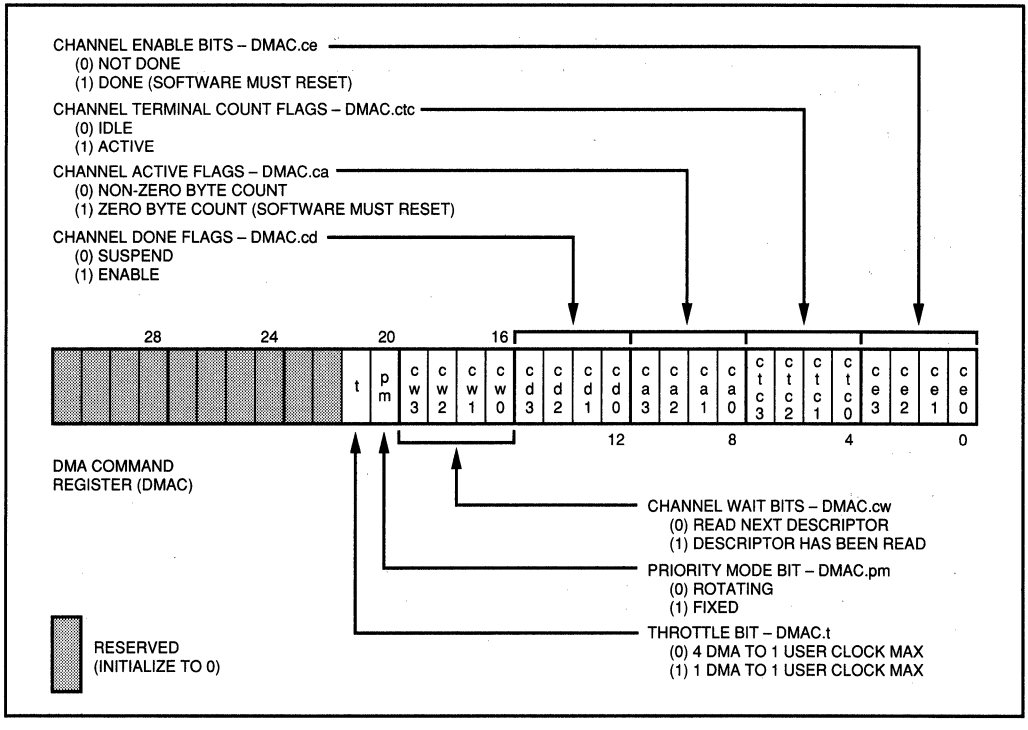


INTERRUPT PENDING (IPND) REGISTER (sf0) AND INTERRUPT MASK (IMSK) REGISTER (sf1)

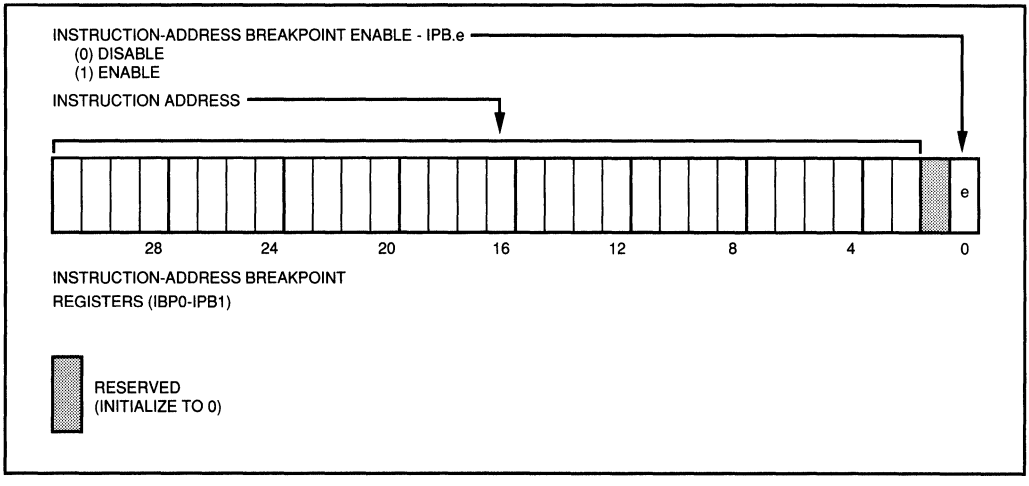


E

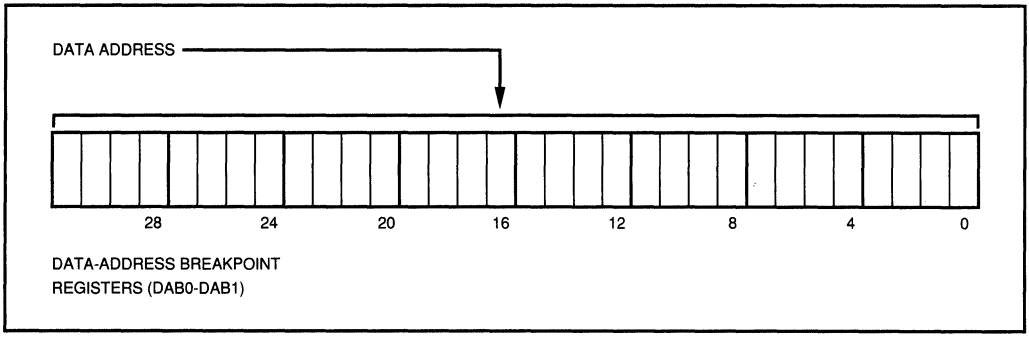
DMA COMMAND (DMAC) REGISTER (sf2)



INSTRUCTION-ADDRESS BREAKPOINT (IPB0-IPB1) REGISTERS

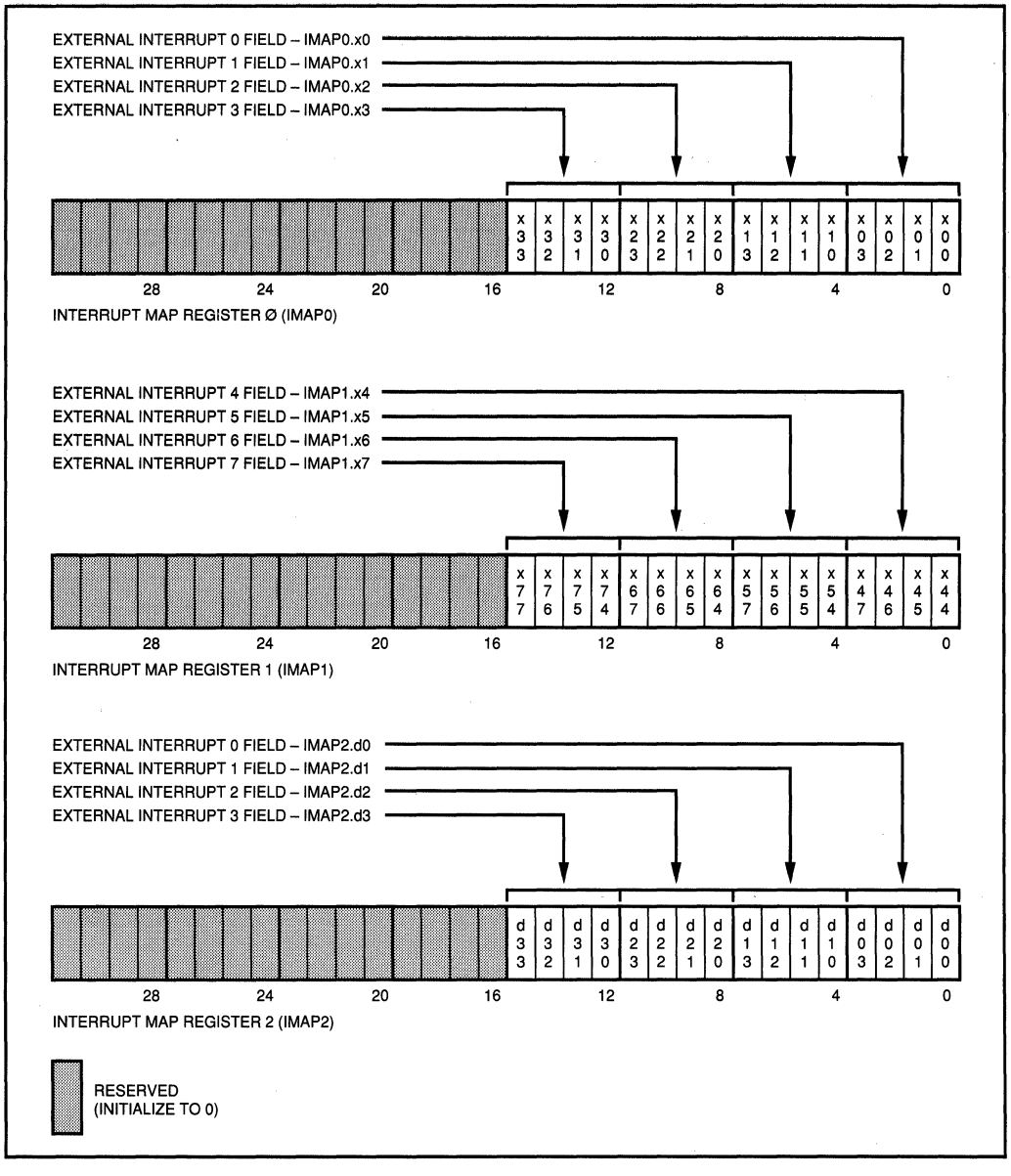


DATA ADDRESS BREAKPOINT (DAB0-DAB1) REGISTERS

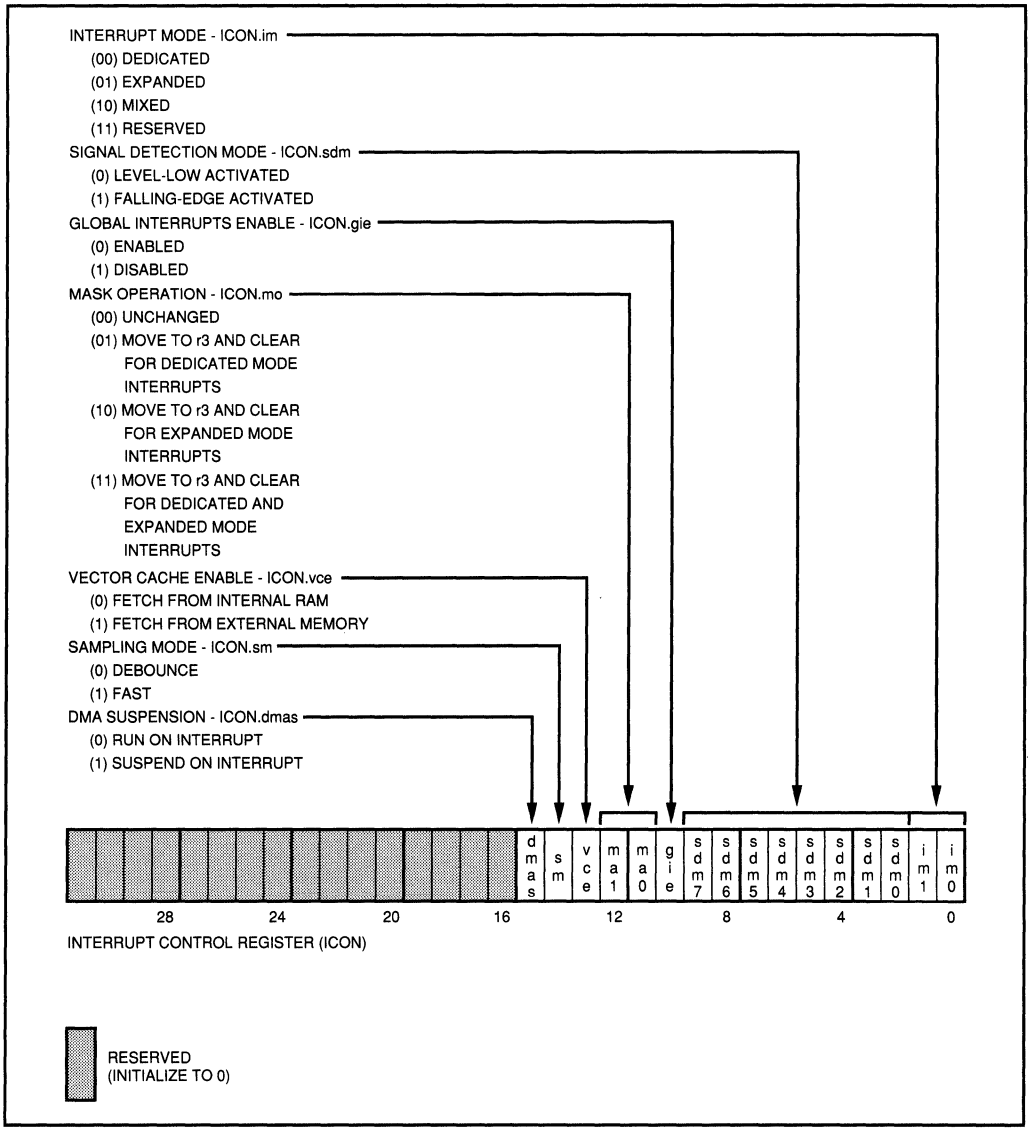


E

INTERRUPT MAP (IMAP0-IMAP2) REGISTERS

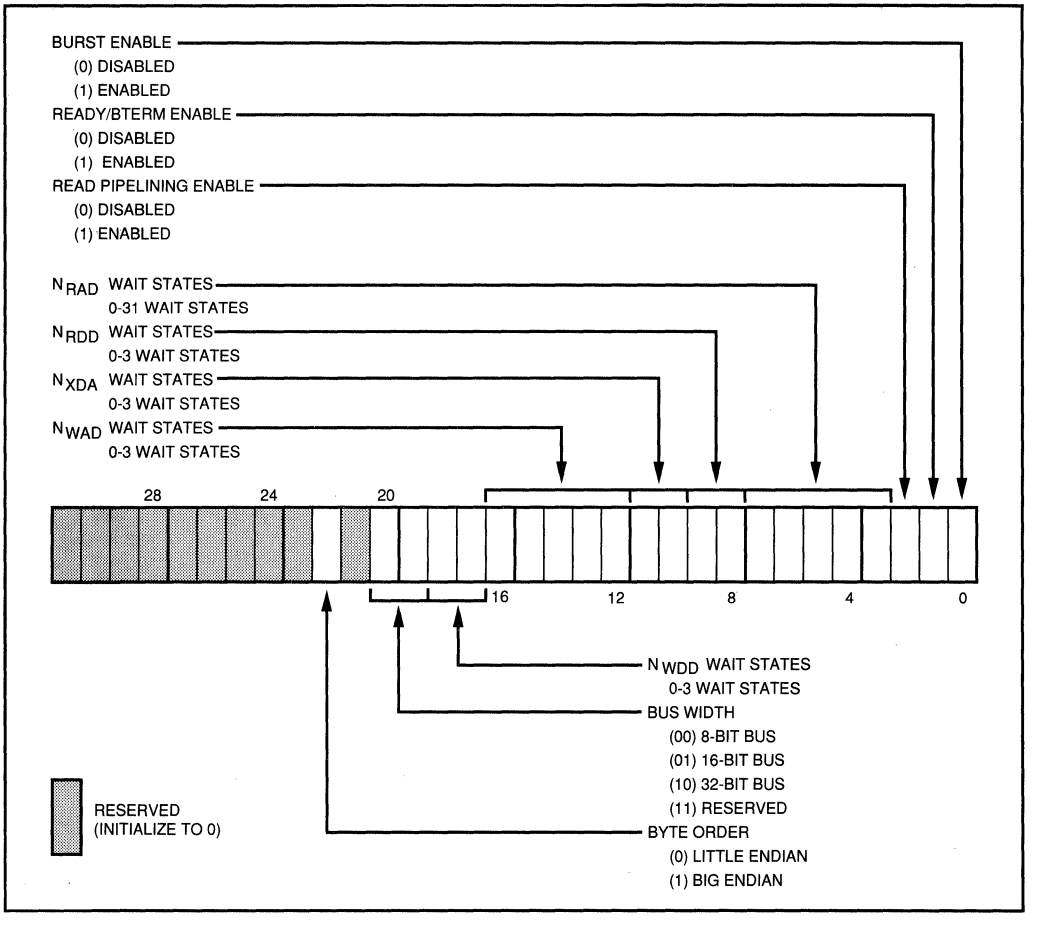


INTERRUPT CONTROL (ICON) REGISTER

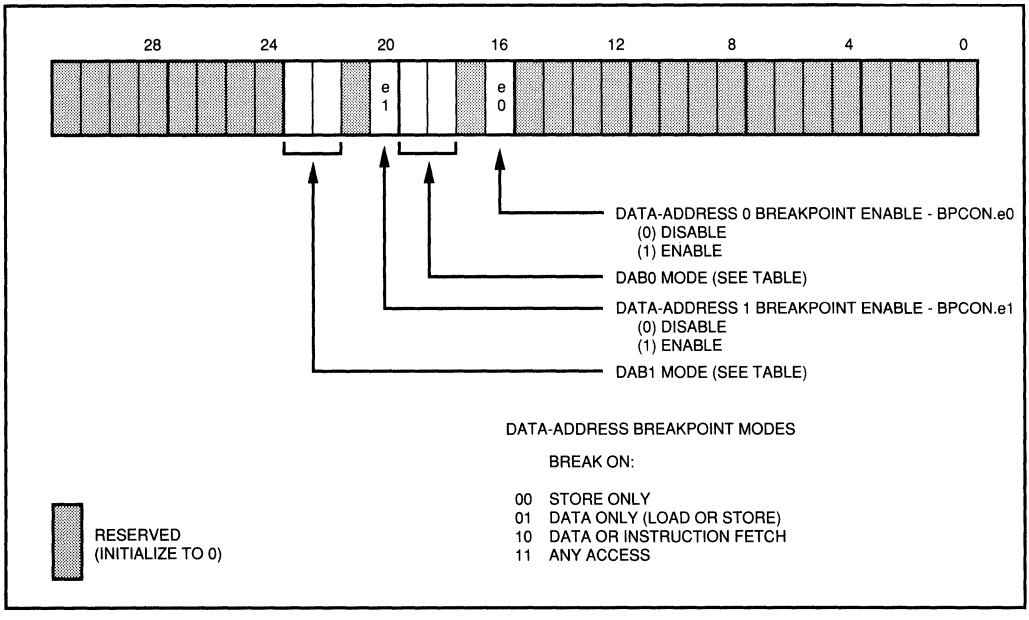


E

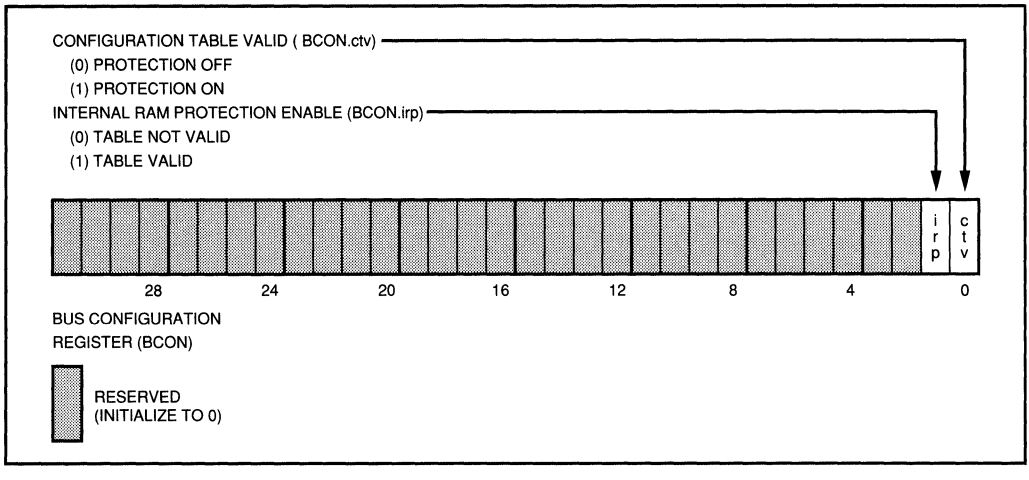
MEMORY REGION CONFIGURATION (MCON0-MCON15) REGISTERS



BREAKPOINT CONTROL (BPCON) REGISTER

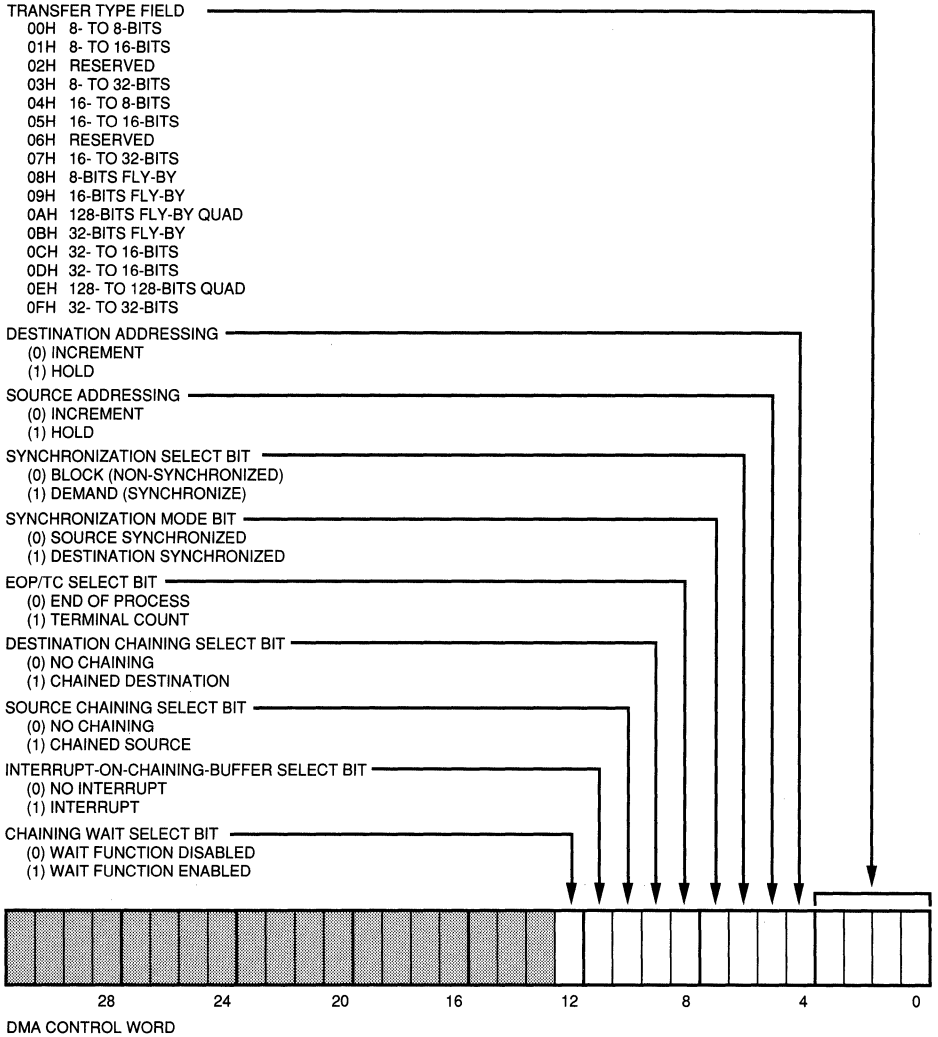


BUS CONFIGURATION (BCON) REGISTER



E

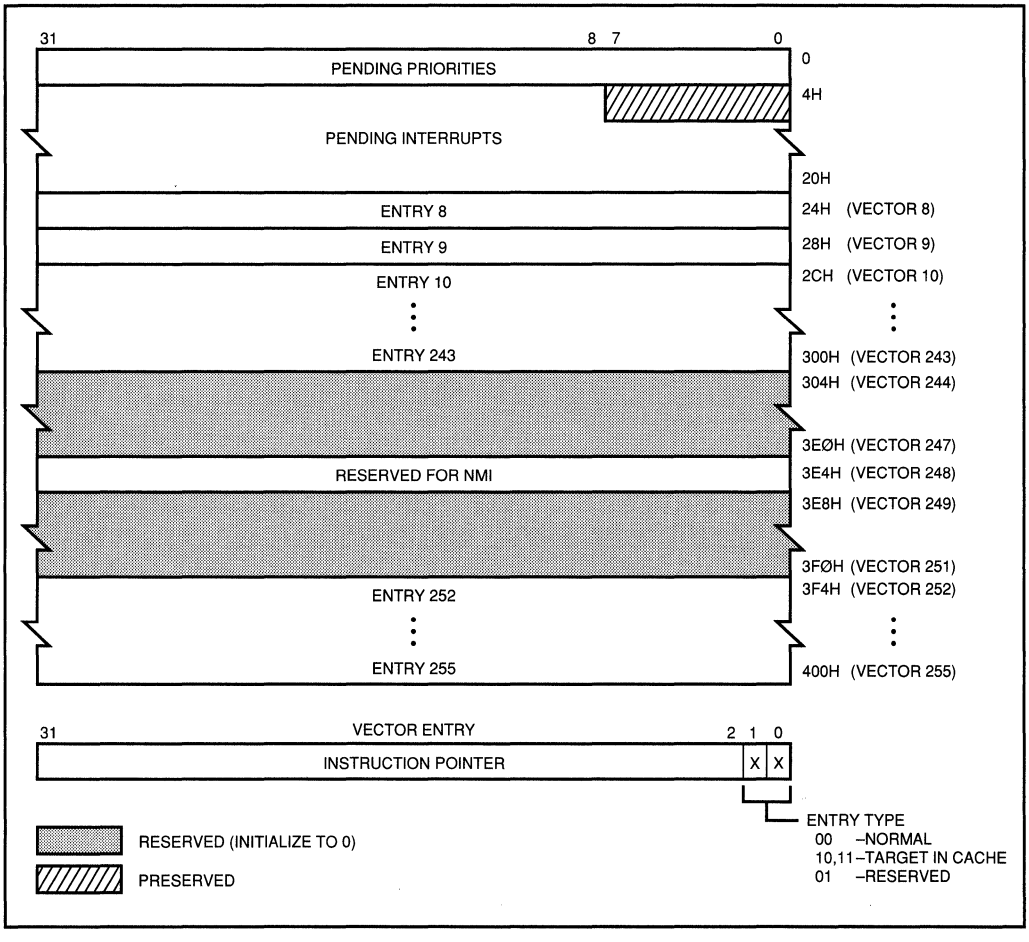
DMA CONTROL WORD



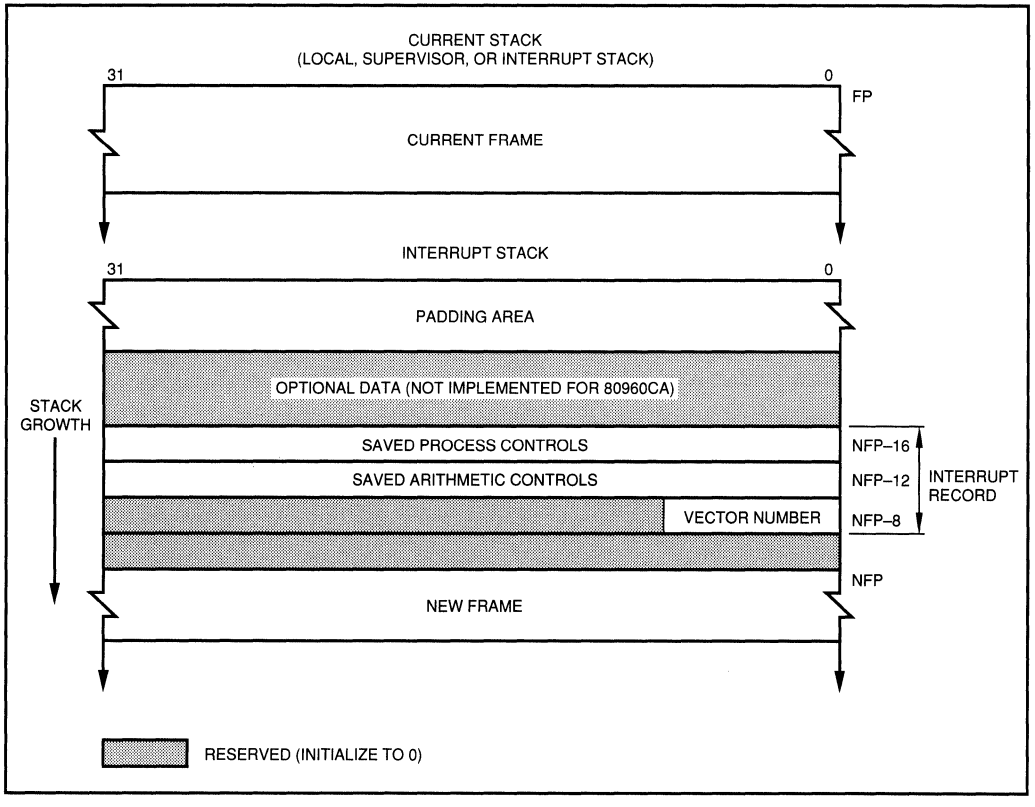
CONTROL TABLE

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
BREAKPOINT CONTROL (BPCON)	60H
TRACE CONTROLS (TC)	64H
BUS CONFIGURATION CONTROL (BCON)	68H
RESERVED (INITIALIZE TO 0)	6CH

INTERRUPT TABLE

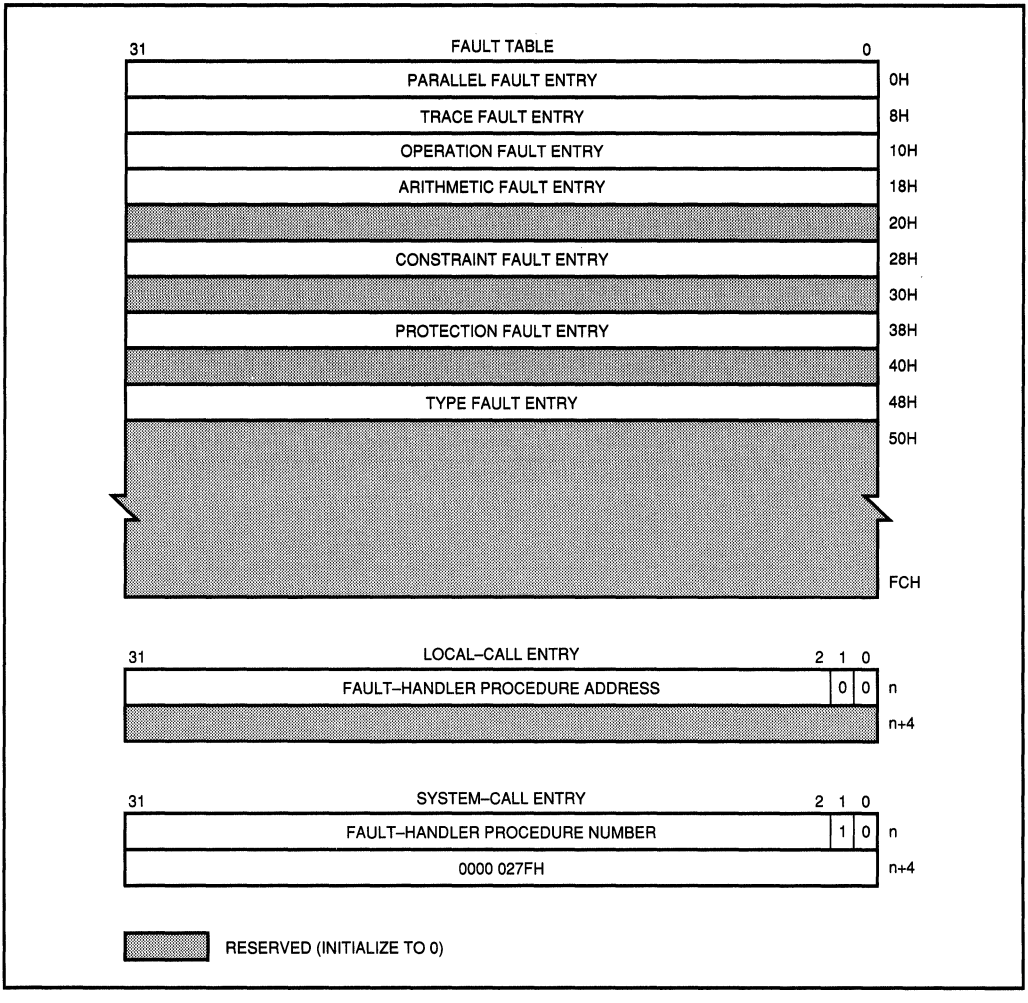


INTERRUPT RECORD

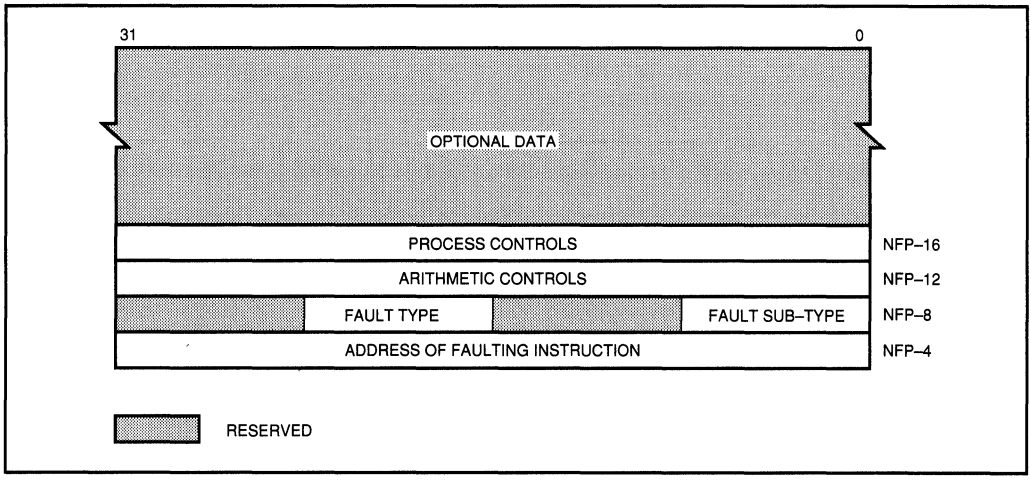


E

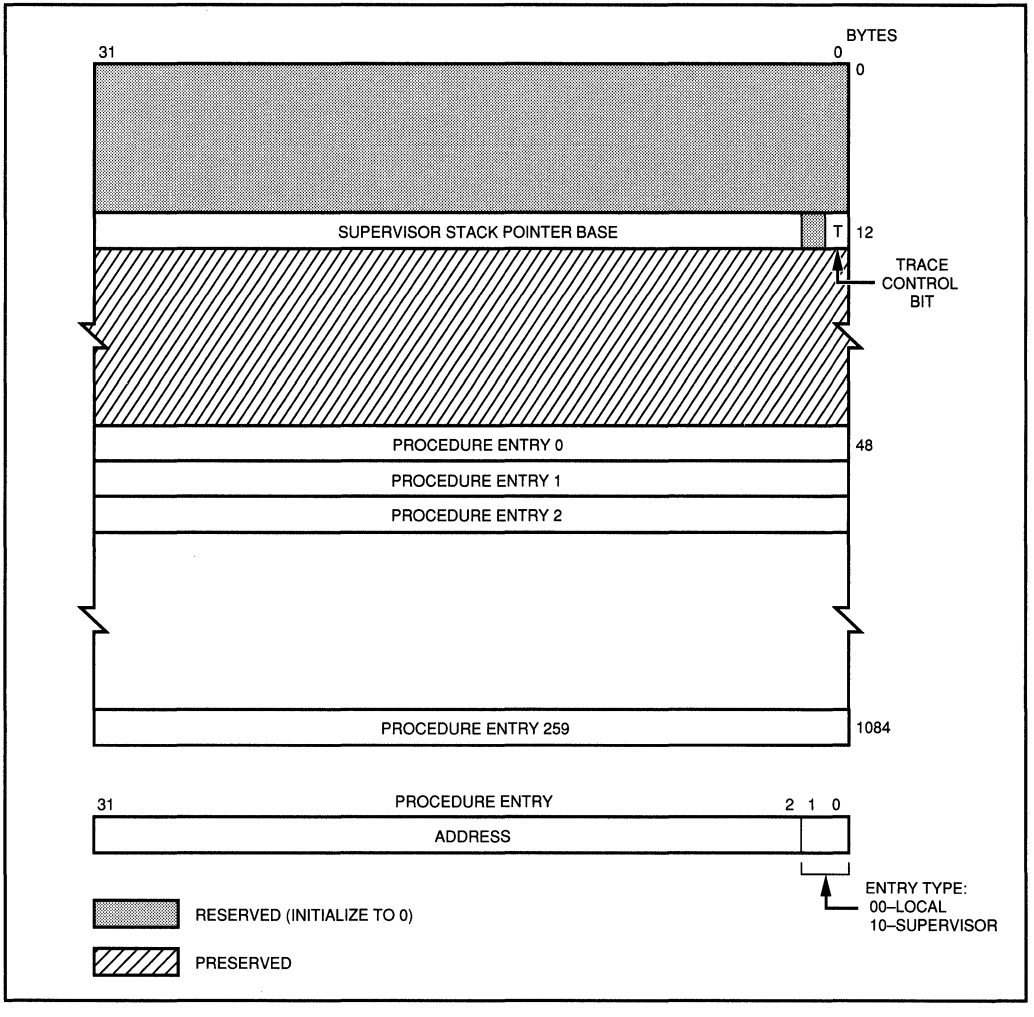
FAULT TABLE


31
FAULT TABLE
0
PARALLEL FAULT ENTRY
0H
TRACE FAULT ENTRY
8H
OPERATION FAULT ENTRY
10H
ARITHMETIC FAULT ENTRY
18H
20H
CONSTRAINT FAULT ENTRY
28H
30H
PROTECTION FAULT ENTRY
38H
40H
TYPE FAULT ENTRY
48H
50H
FCH
31
LOCAL–CALL ENTRY
2 1 0
FAULT–HANDLER PROCEDURE ADDRESS
0 0
n
n+4
31
SYSTEM–CALL ENTRY
2 1 0
FAULT–HANDLER PROCEDURE NUMBER
1 0
n
0000 027FH
n+4
RESERVED (INITIALIZE TO 0)

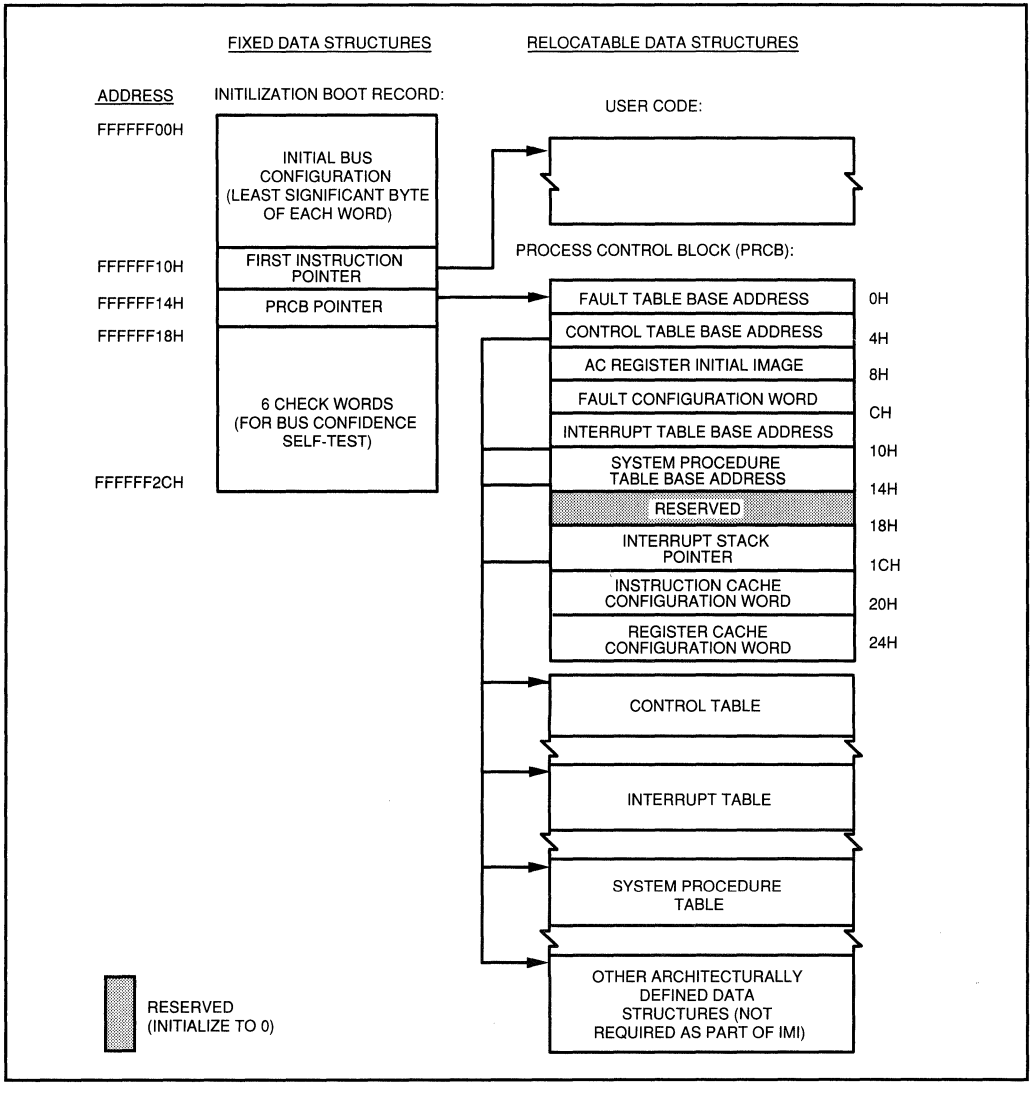
FAULT RECORD



SYSTEM-PROCEDURE TABLE



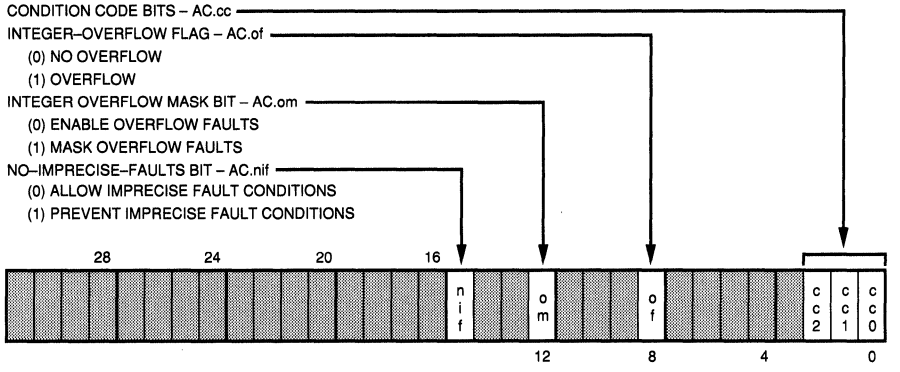
INITIALIZATION BOOT RECORD (IBR) AND PROCESS CONTROL BLOCK (PRCB)



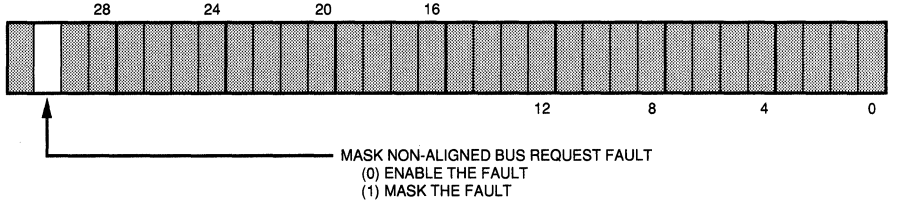
E

CONFIGURATION WORDS IN THE PRCB

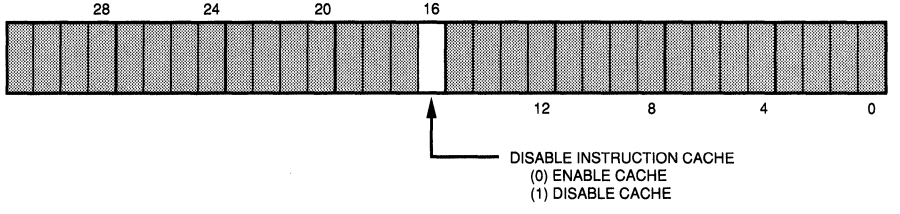
AC REGISTER INITIAL IMAGE



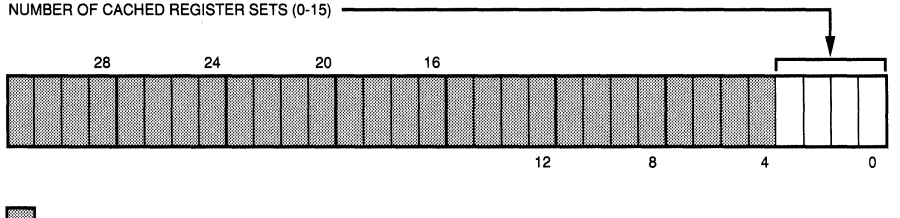
FAULT CONFIGURATION WORD



INSTRUCTION CACHE CONFIGURATION WORD



REGISTER CACHE CONFIGURATION WORD



RESERVED
(INITIALIZE TO 0)

Pin Reference

F

APPENDIX F
PIN REFERENCE

This appendix is a quick-reference which describes the function of the 80960CA's pins. Table F-1 presents the legend for interpreting the pin descriptions in the following tables.

The pins associated with the 32-bit demultiplexed external bus are described in Table F-3. The pins associated with basic processor configuration and control are described in Table F-4. The pins associated with the 80960CA DMA and Interrupt Controller are described in Table F-5. Although not explicitly stated in the tables, all pins float in ONCE™ mode, and all outputs are synchronous.

Table F-1. Pin Description Nomenclature

Symbol	Description
I	Input only pin
O	Output only pin
I/O	Pin can be either an input or output
-	Pins "must be" connected as described
S(. . .)	Synchronous. Inputs must meet setup and hold times relative to PCLK2:1 for proper operation of the processor. All outputs are synchronous to PCLK2:1. S(E) edge sensitive S(L) level sensitive
A(. . .)	Asynchronous. Inputs may be asynchronous to PCLK2:1. A(E) edge sensitive A(L) level sensitive
H(. . .)	While the processor's bus is in the Hold Acknowledge state, the pin: H(1) is driven to VCC H(0) is driven to VSS H(Z) floats H(Q) continues to be a valid output
R(. . .)	While the processor's $\overline{\text{RESET}}$ pin is low, the pin R(1) is driven to VCC R(0) is driven to VSS R(Z) floats R(Q) continues to be a valid output

Table F-2 provides an an example pin description table entry. The "I/O" signifies that the data pins are input-output. The "S(L)" indicates the pins are synchronous and level sensitive to PCLK2:1. The "H(Z)" indicates that these pins float while the processor bus is in a Hold Acknowledge state. The "R(Z)" notation indicates that the pins also float while $\overline{\text{RESET}}$ is low.

Table F-2. Example Pin Description Entry

Name	Type	Description
D31:D0	**I / O** **S(L)** **H(Z)** **R(Z)**	**DATA BUS** carries 32-, 16-, or 8-bit data quantities depending on bus width configuration. The least significant bit of the data is carried on D0 and the most significant on D31. When the bus is configured for 8 bit data, the lower 8 data lines, D7:0 are used. For 16-bit data widths, D15:0 are used. For 32-bit data the full data bus is used.

Table F-3. 80960CA Pin Description — External Bus Signals

Name	Type	Description
A31:2	**O** **S** **H(Z)** **R(Z)**	**ADDRESS BUS** carries the upper 30 bits of the physical address. A31 is the most significant address bit and A2 is the least significant. During a bus access, A31-A2 identify all external addresses to word (4-byte) boundaries. The byte enable signals ($\overline{BE3:0}$) indicate the selected byte in each word. During burst accesses, A3 and A2 increment to indicate successive data cycles.
$\overline{BE3}$ **$\overline{BE2}$** **$\overline{BE1}$** **$\overline{BE0}$**	**O** **S** **H(Z)** **R(Z)**	**BYTE ENABLES** select which of the four bytes addressed by A31:2 are active during an access to a memory region configured for a 32-bit data-bus width. $\overline{BE3}$ applies to D31:24; $\overline{BE2}$ applies to D23:16; $\overline{BE1}$ applies to D15:8; and $\overline{BE0}$ applies to D7:0. 32-bit bus: $\overline{BE3}$ — Byte Enable 3 — enable D31:24 $\overline{BE2}$ — Byte Enable 2 — enable D23:16 $\overline{BE1}$ — Byte Enable 1 — enable D15:8 $\overline{BE0}$ — Byte Enable 0 — enable D7:0 For accesses to a memory region configured for a 16-bit data-bus width, the processor directly encodes $\overline{BE3}$, $\overline{BE1}$ and $\overline{BE0}$ to provided $\overline{BHE}$, A1 and $\overline{BLE}$ respectively. 16-bit bus: $\overline{BE3}$ — Byte High Enable ($\overline{BHE}$) — enable D15:8 $\overline{BE2}$ — Not used (is driven high or low) $\overline{BE1}$ — Address Bit 1 (A1) $\overline{BE0}$ — Byte Low Enable ($\overline{BLE}$) — enable D7:0 For accesses to a memory region configured for an 8-bit data bus width, the processor directly encodes $\overline{BE1}$ and $\overline{BE0}$ to provide A1 and A0 respectively. 8-bit bus: $\overline{BE3}$ — Not used (is driven high or low) $\overline{BE2}$ — Not used (is driven high or low) $\overline{BE1}$ — Address Bit 1 (A1) $\overline{BE0}$ — Address Bit 0 (A0)

D31:0	**I / O** **S(L)** **H(Z)** **R(Z)**	**DATA BUS** carries 32, 16, or 8-bit data quantities depending on bus width configuration. The least significant bit of the data is carried on D0 and the most significant on D31. When the bus is configured for 8 bit data, the lower 8 data lines, D7:0 are used. For 16 bit data bus width, D15:0 are used. For 32 bit data bus width the full data bus is used.
W/$\overline{R}$	**O** **S** **H(Z)** **R(Z)**	**WRITE/READ** is low (0) for read accesses and high (1) for write accesses. The W/$\overline{R}$ signal changes in the same clock cycle as $\overline{ADS}$. It remains valid for the entire access in non-pipelined regions. In pipelined regions, W/$\overline{R}$ may not be valid in the last cycle of a read accesses.
$\overline{ADS}$	**O** **S** **H(Z)** **R(1)**	**ADDRESS STROBE** indicates valid address and the start of a new bus access. $\overline{ADS}$ is asserted for the first clock of a bus access.
$\overline{READY}$	**I** **S(L)** **H(Z)** **R(Z)**	**READY** is an input which signals the termination of a data transfer. $\overline{READY}$ is used to indicate that read data on the bus is valid, or that a write-data transfer has completed. The $\overline{READY}$ signal works in conjunction with the internally programmed wait-state generator. If $\overline{READY}$ is enabled in a region, the pin is sampled after the programmed number of wait-states has expired. If the $\overline{READY}$ pin is deasserted (high), wait states will continue to be inserted until $\overline{READY}$ becomes asserted (low). This is true for the N_{RAD}, N_{RDD}, N_{WAD}, and N_{WDD} wait states. The N_{XDA} wait states cannot be extended.
$\overline{BTERM}$	**I** **S(L)** **H(Z)** **R(Z)**	**BURST TERMINATE** is an input which signals the termination of an access. The assertion of $\overline{BTERM}$ causes another address cycle to occur. The $\overline{BTERM}$ signal works in conjunction with the internally programmed wait-state generator. If $\overline{READY}$ and $\overline{BTERM}$ are enabled in a region, the $\overline{BTERM}$ pin is sampled after the programmed number of wait states has expired. When $\overline{BTERM}$ is asserted (low), $\overline{READY}$ is ignored.

$\overline{\text{WAIT}}$	O S H(Z) R(1)	**WAIT** indicates the status of the internal wait state generator. $\overline{\text{WAIT}}$ is active when wait states are being caused by the internal wait state generator and not by the $\overline{\text{READY}}$ or $\overline{\text{BTERM}}$ inputs. $\overline{\text{WAIT}}$ can be used to derive a write-data strobe. $\overline{\text{WAIT}}$ can also be thought of as a $\overline{\text{READY}}$ output that the processor provides when it is inserting wait states.
$\overline{\text{BLAST}}$	O S H(Z) R(0)	**BURST LAST** indicates the last transfer in a bus access. $\overline{\text{BLAST}}$ is asserted in the last data transfer of burst and non-burst accesses after the wait state counter reaches zero. $\overline{\text{BLAST}}$ remains active until the clock following the last cycle of the last data transfer of a bus access. If the $\overline{\text{READY}}$ or $\overline{\text{BTERM}}$ input is used to extend wait states, the $\overline{\text{BLAST}}$ signal remains active until $\overline{\text{READY}}$ or $\overline{\text{BTERM}}$ terminates the access.
DT/$\overline{\text{R}}$	O S H(Z) R(1)	**DATA TRANSMIT/RECEIVE** indicates direction for data transceivers. DT/$\overline{\text{R}}$ is used in conjunction with $\overline{\text{DEN}}$ to provide control for data transceivers attached to the external bus. When DT/$\overline{\text{R}}$ is low, the signal indicates that the processor will receive data. Conversely, when high the processor will send data. DT/$\overline{\text{R}}$ changes only while $\overline{\text{DEN}}$ is high.
DEN	O S H(Z) R(1)	**DATA ENABLE** indicates data cycles in a bus access. $\overline{\text{DEN}}$ is asserted (low) at the start of the first data cycle of a bus access and is deasserted (high) at the end of the last data cycle. $\overline{\text{DEN}}$ is used in conjunction with DT/$\overline{\text{R}}$ to provide control for data transceivers attached to the external bus. $\overline{\text{DEN}}$ remains asserted for sequential reads from pipelined memory regions. $\overline{\text{DEN}}$ is high when DT/$\overline{\text{R}}$ changes.

F

$\overline{\text{LOCK}}$	O S H(Z) R(1)	**BUS LOCK** indicates that an atomic read-modify-write operation is in progress. $\overline{\text{LOCK}}$ may be used to prevent external agents from accessing memory which is currently involved in an atomic operation. $\overline{\text{LOCK}}$ is asserted (0) in the first clock of an atomic operation, and deasserted (1) in the clock cycle following the last bus access for the atomic operation. The processor acknowledges a bus hold request when $\overline{\text{LOCK}}$ is asserted. The processor also performs DMA transfers while $\overline{\text{LOCK}}$ is active. These actions are allowed to occur when $\overline{\text{LOCK}}$ is asserted so the external memory system has the flexibility to enforce or not to enforce locked accesses.
HOLD	I S(L) H(Z) R(Z)	**HOLD REQUEST** signals that an external agent requests access to the external bus. The processor asserts HOLDA after completing the current bus request. HOLD, HOLDA, and BREQ are used together to arbitrate access to the processor's external bus by external bus agents.
HOLDA	O S H(1) R(Q)	**HOLD ACKNOWLEDGE** indicates to a bus requestor that the processor has relinquished control of the external bus. When HOLDA is asserted, the external address bus, data bus, and bus control signals are floated. HOLD, HOLDA, and BREQ are used together to arbitrate access to the processor's external bus by external bus agents. Since the processor will grant HOLD requests and enter the Hold Acknowledge state even while $\overline{\text{RESET}}$ is active, the state of the HOLDA pin will be independent of the $\overline{\text{RESET}}$ pin.
BREQ	O S H(Q) R(0)	**BUS REQUEST** indicates that the processor wishes to perform a bus request. BREQ can be used by external bus arbitration logic in conjunction with HOLD and HOLDA to determine when to return mastership of the external bus to the processor.
D/$\overline{\text{C}}$	O S H(Z) R(0)	**DATA OR CODE** indicates that a bus access is a data access (1) or a instruction access (0). D/$\overline{\text{C}}$ has the same timing as W/$\overline{\text{R}}$

$\overline{\text{DMA}}$	O S H(Z) R(1)	**DMA ACCESS** indicates that a bus access is initiated by the DMA controller. $\overline{\text{DMA}}$ is asserted (low) for any DMA access. $\overline{\text{DMA}}$ is deasserted (high) for all other accesses.
$\overline{\text{SUP}}$	O S H(Z) R(0)	**SUPERVISOR ACCESS** indicates that a bus access originates from a request issued while in supervisor mode. $\overline{\text{SUP}}$ is asserted (low) when the access has supervisor privileges, and is deasserted (high) otherwise. $\overline{\text{SUP}}$ can be used to isolate supervisor code and data structures from non-supervisor access.

Table F-4. 80960CA Pin Description — Processor Control Signals

Name	Type	Description
$\overline{\text{RESET}}$	I A(L) H(Z) R(Z)	**RESET** causes the chip to reset. When $\overline{\text{RESET}}$ is asserted (low), all external signals return to the reset state. When $\overline{\text{RESET}}$ is deasserted, initialization begins. $\overline{\text{RESET}}$ must be asserted during power-up. When the two-X clock mode is selected, $\overline{\text{RESET}}$ must remain asserted for 16 PCLK cycles before being deasserted in order to guarantee correct initialization of the processor. When the one-x clock mode is selected, $\overline{\text{RESET}}$ must remain asserted for 10,000 PCLK cycles before being deasserted in order to guarantee correct initialization of the processor. The CLKMODE pin selects one-x or two-x input-clock division of the CLKIN pin. The processor's Hold Acknowledge bus state functions while the chip is reset. If the processor's bus is in the Hold Acknowledge state when $\overline{\text{RESET}}$ is activated, the processor internally resets, but maintains the Hold Acknowledge state on external pins until the Hold request is removed. If a hold request is made while the processor is in the reset state, the processor bus grants HOLDA and enters the Hold Acknowledge state.
$\overline{\text{FAIL}}$	O S H(Q) R(0)	**FAIL** indicates failure of the processor's self-test performed at initialization. When $\overline{\text{RESET}}$ is deasserted and the processor begins initialization, the $\overline{\text{FAIL}}$ pin is asserted (0). An internal self-test is performed as part of the initialization process. If this self-test passes, the $\overline{\text{FAIL}}$ pin is deasserted (1) otherwise it remains asserted. The $\overline{\text{FAIL}}$ pin is reasserted while the processor performs and external bus self-confidence test. If this self-test passes, the processor deasserts the $\overline{\text{FAIL}}$ pin and branches to the users initialization routine, otherwise the $\overline{\text{FAIL}}$ pin remains asserted. Internal self-test and the use of the $\overline{\text{FAIL}}$ pin can be disabled with the STEST pin.

STEST	**I** **S(L)** **H(Z)** **R(Z)**	**SELF TEST** causes the processor's internal self-test feature to be enabled or disabled at initialization. STEST is read on the rising edge of RESET. When asserted (high) the processor's internal self-test and external bus confidence tests are performed during processor initialization. When deasserted (low), no self-tests are performed during initialization.
ONCE	**I** **A** **H(Z)** **R(Z)**	**ON CIRCUIT EMULATION** causes all outputs to be floated when asserted (low). ONCE is continuously sampled while RESET is low, and is latched on the rising edge of RESET To place the processor in the ONCE state: 1) assert RESET and ONCE (order does not matter) 2) wait for at least 16 clocks in two-x mode (10,000 clocks in one-x mode) after VCC and CLKIN are within operating specifications 3) deassert RESET 4) wait at least 16 clocks The processor will now be latched in the ONCE state as long as RESET is high. To exit the ONCE state, bring VCC and CLKIN to operating conditions, then assert RESET and bring ONCE high prior to deasserting RESET. CLKIN must operate within the specified operating conditions of the processor until step 4 above has been completed. The CLKIN may then be changed to D.C. to achieve the the lowest possible ONCE mode leakage current. ONCE mode can be used by emulator products or for board testers to effectively make an installed processor transparent in the board.

CLKIN	**I** **A(E)** **H(Z)** **R(Z)**	**CLOCK INPUT** is an input for the external clock needed to run the processor. The external clock is internally divided as prescribed by the CLKMODE pin to produce PCLK2:1.
CLKMODE	**I** **A(L)** **H(Z)** **R(Z)**	**CLOCK MODE** selects the division factor applied to the external clock input (CLKIN). When CLKMODE is high, CLKIN is divided by one to create PCLK2:1 and the processor's internal clock. When CLKMODE is low, CLKIN is divided by two to create PCLK2:1. The clock mode is not latched by the processor. If left unconnected, the processor internally pulls the CLKMODE pin low, enabling the two-x clock mode.
PCLK2 **PCLK1**	**O** **S** **H(Q)** **R(Q)**	**PROCESSOR OUTPUT CLOCKS** provide a timing reference for all inputs and outputs of the processor. All inputs and output timings are specified in relation to PCLK2 and PCLK1. PCLK2 and PCLK1 are identical signals. Two output pins are provided to allow flexibility in the system's allocation of capacitive loading on the clock. PCLK2:1 may also be connected at the processor to form a single clock signal.
VSS	-	**GROUND** connections consist of 24 pins which must be shorted externally to a VSS board plane.
VCC	-	**POWER** connections consist of 24 pins which must be shorted externally to a VCC board plane.
N/C	-	**NO CONNECT** pins must not be connected in a system.

Table F-5. 80960CA Pin Description — DMA and Interrupt Controller Signals

Name	Type	Description
DREQ3 DREQ2 DREQ1 DREQ0	I A(L) H(Z) R(Z)	**DMA REQUEST** causes a DMA transfer to be requested. Each of the four signals requests a transfer on a single channel. DREQ0 requests channel 0, DREQ1 requests channel 1, etc. When two or more channels are requested simultaneously, the channel with the highest priority is serviced first. The channel priority mode is programmable.
DACK3 DACK2 DACK1 DACK0	O S H(Z) R(1)	**DMA ACKNOWLEDGE** indicates that a DMA transfer is being executed. Each of the four signals acknowledges a transfer for a single channel. DACK0 acknowledges channel 0, DACK1 acknowledges channel 1, etc. DACK3:0 are active (0) when the requesting device of a DMA is accessed.
EOP3/TC3 EOP2/TC2 EOP1/TC1 EOP0/TC0	I / O A(L) H(Z) R(Z)	**END OF PROCESS/TERMINAL COUNT** can be programmed as either an input (EOP3:0) or as an output (TC3:0), but not both. Each pin is individually programmable. When programmed as an input, EOP3:0 causes the termination of a current DMA transfer for the channel corresponding to the pin. EOP0 corresponds to channel 0, EOP1 corresponds to channel 1, etc. When a channel is configured for source *and* destination chaining, the EOP3:0 pin for that channel causes termination of only the current buffer transferred and causes the next buffer to be read. EOP3:0 are asynchronous inputs. When programmed as an output, the channel's TC3:0 pin indicates that the channel byte count has reached 0 and a DMA has terminated. TC3:0 is driven active (0) for a single clock cycle after the last DMA transfer is completed on the external bus. TC3:0 are synchronous outputs.

F

$\overline{\text{XINT7}}$ $\overline{\text{XINT6}}$ $\overline{\text{XINT5}}$ $\overline{\text{XINT4}}$ $\overline{\text{XINT3}}$ $\overline{\text{XINT2}}$ $\overline{\text{XINT1}}$ $\overline{\text{XINT0}}$	I A(E/L) H(Z) R(Z)	**EXTERNAL INTERRUPT PINS** cause interrupts to be requested. These pins can be configured in three modes. In the Dedicated Mode, each pin is a dedicated external interrupt source. Dedicated inputs can be individually programmed to be level (low) or edge (falling) activated. In the Expanded Mode, the 8 pins act together as an 8-bit vectored interrupt source. The interrupt pins in this mode are level activated. Since the interrupt pins are active low, the vector number requested is the one's complement of the positive logic value place on the port. This eliminates glue logic to interface to combinational priority encoders which output negative logic. In the Mixed Mode, $\overline{\text{XINT7:5}}$ are dedicated sources and $\overline{\text{XINT4:0}}$ act as the 5 most significant bits of an expanded mode vector. The least significant bits are set to 0010_2 internally.
$\overline{\text{NMI}}$	I A(E) H(Z) R(Z)	**NON-MASKABLE INTERRUPT** causes a non-maskable interrupt event to occur. $\overline{\text{NMI}}$ is the highest priority interrupt recognized. $\overline{\text{NMI}}$ is an edge (falling) activated source.

Glossary **G**

APPENDIX G
GLOSSARY

This appendix contains a glossary of terms commonly used in this manual.

Address Space. An array of bytes used to store the program code, data, stacks, and system data structures that are required to execute a program. The address space is linear (or 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. (The 80960 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 a single byte, a half-word (2 bytes), a word (4 bytes), a double-word (8 bytes), a triple-word (12 bytes) or a quad-word (16 bytes) in memory (depending on the instruction being 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. The flags and masks contained in this register include the condition code flags; the integer-overflow flag and mask bit; and the no-imprecise-faults (NIF) bit. All the 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. (The 80960 core architecture does not define any fault types that are asynchronous.)

Condition-Code Flags. Bits 0, 1, and 2 of the arithmetic-controls register. 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 has set the condition-code flags, it leaves them unchanged until it executes another instruction that uses these flags to store results.

Execution-Mode Flag. Bit 1 of the process-controls register. This flag determines whether the processor is operating in the 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 entry in the fault table 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. (See Frame Pointer.)

Frame Pointer (FP). The address of the first byte in the current (topmost) stack frame of the procedure stack. The FP is contained in global-register g15.

Frame. (See Stack Frame.)

Global Registers. A set of 16 general-purpose registers (g0 through g15), the contents of which are preserved across procedure boundaries. Global registers are used for general storage of data and addresses and for passing parameters between procedures.

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).

Instruction Cache. A memory array used for temporary storage of instructions that have been fetched from main memory. The purpose of the instruction cache 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 2 least-significant bits of the IP are always zero.

Integer-Overflow Flag. Bit 8 of the arithmetic-controls register. 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. Bit 12 of the arithmetic-controls register. 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. The stack that 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 pending-interrupt fields of the interrupt table contains bits that indicate the priorities and vector numbers of interrupts that are waiting to be serviced.

Interrupt Vector. A pointer to an interrupt-handling procedure. In the 80960 architecture, interrupts vectors are stored in the interrupt table.

Interrupt. An event that causes the execution of a program to be suspended temporarily so that the processor can 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), the contents of which are associated with the procedure currently being executed. Local registers are used to 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. The memory array that the address space is mapped to. It can be read-write memory, read-only memory, or a combination of the two. A memory address is generally synonymous with an address in the address space.

NIF. (See No-Imprecise Faults Bit.)

No-Imprecise Faults (NIF) Bit. Bit 15 of the arithmetic-controls register. 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.

Parallel Faults. A condition which occurs when multiple execution units, which are 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 has saved so that it can service it 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 priority and vector number of the interrupt 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 is generally used to refer to the first byte of a procedure, the first byte of a data structure, or a specific byte-location in a stack.

Precise Faults. Faults that are generated in the order that they occur in the instruction stream and with sufficient fault information to allow software to recover from the faults without altering the control flow of the program. The NIF bit in the arithmetic-controls register and the **syncf** instruction allow software to force all faults to be precise.

Previous Frame Pointer (PFP). The address of the first byte of the previous stack frame. It is contained in bits 4 through 31 of local-register r0.

Priority Field. Bits 16 through 20 of the process-controls register. This field determines the priority (from 0 to 31) of the processor. 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. The 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 process-controls register.

Process-Controls Register. A 32-bit register that contains miscellaneous pieces of information used to control processor activity and show the current state of the processor. The flags and fields contained in this register include the trace-enable bit; the execution-mode flag; the trace-fault-pending flag, the state flag; the priority field; and the internal-state field. All the 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 being used 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. A set of implementation-defined registers that represent an extension to the basic register set of the 80960 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 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 the 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. The 80960 architecture defines three stacks: local stack, supervisor stack, and interrupt stack.

State flag. Bit 10 of the process-controls register. 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 state flag in the process-controls register to indicate its current state.

Status and Control Registers. A set of four architecturally-defined registers (each 32-bits long) that contain status and control information used in controlling program flow. These registers include the instruction pointer (IP), the arithmetic-controls register, the process-controls register, and the trace-controls 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. When the processor is in supervisor mode, it uses the supervisor stack. 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 it is in the supervisor mode.

System Call. An explicit procedure call made with the **calls** instruction. There are two types of system calls: a system-local call and a system-supervisor call. On a system call, the processor gets a pointer to the system procedure through the system-procedure 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 the tracing facilities of the processor. 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. Bit 0 of the process-controls register. This bit determines whether trace faults are to be generated (1) or not-generated (0).

Trace-Fault-Pending Flag. Bit 0 of the process-controls register. 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 the return from the fault procedure for the non-trace fault, the processor checks the trace-fault-pending flag. If it is set, it generates the trace fault and handles it.

Tracing. The ability of the processor to detect the execution of certain types of instructions, such as branch instructions, call instructions, and return instructions. 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.)

APPENDIX H
INDEX

B

C

H

D

H

E

F

G

H

I

H

J, K

L

M

N

H

O

P

Q

R

H

S

T

H

U

V, W, X, Y, Z

Instruction Set Quick Reference

I

APPENDIX I
INSTRUCTION-SET QUICK REFERENCE

This appendix provides a quick reference listing for each 80960CA instruction. The reference lists every instruction's mnemonic, name, assembler syntax, action, opcode, instruction format, machine type and execution time.

Mnemonic – The acronym recommended for use by 80960 assemblers.

Name – A descriptive name for the instruction.

Assembler Syntax – The recommended operand ordering and syntax for 80960 assemblers.

Action – An abbreviated algorithmic description of the action performed by the instruction, including any modification of the Arithmetic Controls, Process Controls or Trace Controls registers. Any possible faults generated are also listed. Table I-1 describes the meaning of the shorthand used in the register and fault sections of the reference.

Table I-1 Action Shorthand

—	Unchanged by the instruction, or, is not generated by the instruction.
0	Set to 0 by the instruction under any condition.
1	Set to 1 by the instruction under any condition.
√	May be set or cleared by the instruction.
↓	May be cleared by the instruction, but will never be set by the instruction.
↑	May be set by the instruction, but will never be cleared by the instruction.
T	Trace Fault Type
O	Operation Fault Type
A	Arithmetic Fault Type
C	Constraint Fault Type
P	Privilege Fault Type
Y	Type Fault Type
I	Instruction
B	Branch
C	Call
S	Supervisor
BR	Breakpoint

R	Return
P	Prereturn
U	Unimplemented
OP	Invalid Operand
OC	Invalid Opcode
IO	Integer Overflow
ZD	Zero Divide
M	Machine
R	Range
?	Undefined

Opcode – The 80960 opcode for the instruction.

Instruction Format – The encoding format of the instruction. Appendix D, titled *Instruction Set Encoding Reference*, details the complete encoding for each instruction.

Machine Type – The machine type indicates which group of the 80960CA parallel processing units are used to execute the instruction. Table I-2 lists the possible machine types.

Table I-2 Machine Type Shorthand

R	Register – The instruction is executed in parallel by a processing unit on the Register side of the processor.
M	Memory – The instruction is executed in parallel by a processing unit on the Memory side of the processor.
C	Control – The instruction is executed by the Instruction Scheduler, in parallel with other R- or M-type instructions.
μ	Micro-flow – The processor performs this instruction by issuing a sequence of R-, M- and/or C-type instructions stored in its internal ROM.

Execution Time – The execution time for the instruction is listed in two ways: Instruction Issue and Result Latency. The Instruction Issue time is the number of clocks taken by the instruction when there are no register or resource dependencies to slow it down. Back-to-back instructions with no dependencies will execute at the instruction issue rate. The Result Latency is the length of time that an instruction takes to complete once it begins. Back-to-back instructions which are dependent upon each other will execute at the Result Latency rate. Where a range of numbers is shown for an execution time, the range is due to either the degree of parallel instruction issue achieved, or conditions specific to the instruction's run-time execution (such as branch taken or not taken). Table I-3 describes the shorthand for additive factors that appear in the execution time columns.

not taken). Table I-3 describes the shorthand for additive factors that appear in the execution time columns.

Table I-3 Execution Times Shorthand for Additive Factors

efa	The time for effective address calculation. For the **lda** instruction, *efa* = #clocks — Addressing Mode 0 offset 0 disp 0 (reg) 0 offset(reg) 0 disp(reg) 0 disp[reg * scale] 1 (reg)[reg * scale] 1 disp(reg)[reg * scale] 3 disp+8(IP) For all other references, *efa* = #clocks — Addressing Mode 0 offset 0 disp 0 (reg) 1 offset(reg) 1 disp(reg) 1 disp[reg * scale] 2 (reg)[reg * scale] 2 disp(reg)[reg * scale] 4 disp+8(IP)
bus	The time necessary to perform the external memory operations associated with the instruction. The additive factor *bus* equals 0 when memory operations associated with the instruction are in the on-chip data RAM. *Bus* is also equal to zero for branches and calls where the target is in the instruction cache.
spill	The time required to write one cached register set to its reserved frame on the stack. Although *spill* is a function of *bus*, *spill* equals 36 when the stack is in external zero wait state memory.
fill	The time required to read one register set from the previous frame of the stack. Although *fill* is a function of *bus*, *fill* equals 36 when the stack is in external zero wait state memory.
frames	The number of register sets written flushed.
fixup	When the **shrdi** instruction concludes, a four clock micro-flow executes if any bits shifted out were set, and the source operand was negative. *Fixup* is four clocks for this case. *Fixup* is zero clocks for positive operands, and negative operands where only zeros are shifted out.

Mnemonic	Description	Arithmetic Controls						Process Controls				Trace Controls		Faults						Opcode	Opcode Format	Instruction Execution		
		nif	om	of	cc 2	cc 1	cc 0	p	tfp	em	te	Events	Modes	T	O	A	C	P	Y			Mach. Type	Instruction Issue	Result Latency
addc	Add Ordinal with Carry *src1*, *src2*, *dst* reg/lit/sfr reg/lit/sft reg/sfr *dst* ← *src2* + *src1* + AC.cc1 AC.cc0 ← integer overflow AC.cc1 ← carry out	—	—	—	0	√	√	—	√	—	—	I	—	I	U	—	—	—	M	5B:0	REG	R	0.5 - 1	1
addi	Add Integer *src1*, *src2*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr *dst* ← *src2* + *src1*	—	—	√	—	—	—	—	√	—	—	I	—	I	U	IO	—	—	M	59:1	REG	R	0.5 - 1	1
addo	Add Ordinal *src1*, *src2*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr *dst* ← *src2* + *src1*	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	59:0	REG	R	0.5 - 1	1
alterbit	Alter Bit *bitpos*, *src*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr if (AC.cc1 = 1) *dst* ← *src* or 2^(*bitpos* mod 32) else *dst* ← *src* and not (2^(*bitpos* mod 32)	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	58:F	REG	R	0.5 - 1	1
and	And *src1*, *src2*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr *dst* ← *src2* and *src1*	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	58:1	REG	R	0.5 - 1	1
andnot	And Not *src1*, *src2*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr *dst* ← *src2* and not(*src1*)	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	58:2	REG	R	0.5 - 1	1
atadd	Atomic Add Ordinal *sre/dst*, *src*, *dst* reg reg/lit/sfr reg/sfr *dst* ← memory(*src/dst* and not(0x3)) memory(*src/dst* and not(0x3)) ← *dst* + *src* *LOCK is asserted during the read, and deasserted after the write is completed.*	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	61:2	REG	μ	5 + bus	5 + bus
atmod	Atomic Modify *src*, *mask*, *src/dst* reg/lit/sfr reg/lit/sffr reg temp ← memory(*src* and not(0x3)) memory(*src* and not(0x3)) ← (*src/dst* and *mask*) or (temp and not(*mask*)) *src/dst* ← temp *LOCK is asserted during the read, and deasserted after the write is completed.*	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	61:0	REG	M	5 + bus	5 + bus
b	Branch *targ* IP ← *targ*	—	—	—	—	—	—	—	√	—	—	I B	—	I B	U	—	—	—	—	08	CTRL	C	0 - 2	2
bal	Branch and Link *targ* g14 ← next IP IP ← *targ*	—	—	—	—	—	—	—	√	—	—	I B	—	I B	U	—	—	—	—	0B	CTRL	C	1 - 2	2
balx	Branch And Link Extended *efa*, *dst* addr reg *dst* ← next IP IP ← *efa*	—	—	—	—	—	—	—	√	—	—	I B	—	I	OP U OC	—	—	—	—	85	MEM	μ	2 + efa + bus	2 + efa + bus
bbc	Check Bit and Branch If Clear *bitpos*, *src*, *targ* reg/lit/sfr reg AC.cc1 ← not (*src* and 2^(*bitpos* mod 32)) if (AC.cc1 = 1) IP ← *targ*	—	—	—	0	√	0	—	√	—	—	I B	—	I B	U	—	—	—	M	30	COBR	C	1 - 3	3
bbs	Check Bit and Branch If Set *bitpos*, *src*, *targ* reg/lit/sfr reg AC.cc1 ← (*src* and 2^(*bitpos* mod 32)) if (AC.cc1 = 1) IP ← *targ*	—	—	—	0	√	0	—	√	—	—	I B	—	I B	U	—	—	—	M	37	COBR	C	1 - 3	3
be	Branch If Equal *targ* if ((AC.cc and 010) ≠ 0) IP ← *targ*	—	—	—	—	—	—	—	√	—	—	I B	—	I B	U	—	—	—	—	12	CTRL	C	0 - 2	2
bg	Branch If Greater *targ* if ((AC.cc and 001) ≠ 0) IP ← *targ*	—	—	—	—	—	—	—	√	—	—	I B	—	I B	U	—	—	—	—	11	CTRL	C	0 - 2	2
bge	Branch If Greater Or Equal *targ* if ((AC.cc and 011) ≠ 0) IP ← *targ*	—	—	—	—	—	—	—	√	—	—	I B	—	I B	U	—	—	—	—	13	CTRL	C	0 - 2	2
bl	Branch If Less *targ* if ((AC.cc and 100) ≠ 0) IP ← *targ*	—	—	—	—	—	—	—	√	—	—	I B	—	I B	U	—	—	—	—	14	CTRL	C	0 - 2	2
ble	Branch If Less Or Equal *targ* if ((AC.cc and 110) ≠ 0) IP ← *targ*	—	—	—	—	—	—	—	√	—	—	I B	—	I B	U	—	—	—	—	16	CTRL	C	0 - 2	2
bne	Branch If Not Equal *targ* if ((AC.cc and 101) ≠ 0) IP ← *targ*	—	—	—	—	—	—	—	√	—	—	I B	—	I B	U	—	—	—	—	15	CTRL	C	0 - 2	2
bno	Branch If Not Ordered *targ* if (AC.cc = 000) IP ← *targ*	—	—	—	—	—	—	—	√	—	—	I B	—	I B	U	—	—	—	—	10	CTRL	C	0 - 2	2
bo	Branch If Ordered *targ* if (AC.cc ≠ 0) IP ← *targ*	—	—	—	—	—	—	—	√	—	—	I B	—	I B	U	—	—	—	—	17	CTRL	C	0 - 2	2
bx	Branch Extended *efa* addr IP ← *efa*	—	—	—	—	—	—	—	√	—	—	I B	—	I B	OP U OC	—	—	—	—	84	MEM	μ	2 + efa	2 + efa
call	Call *targ* disp RIP ← next IP temp ← (sp + 0x10) and not (0xf) memory (fp) ← r0:15 /* these accesses are cached in the local register cache */ PFP ← FP PFP.rt ← 000 FP ← temp SP ← FP + 64 IP ← *targ*	—	—	—	—	—	—	—	√	—	—	I C	—	I C	—	—	—	—	—	09	CTRL	μ	4 + spill	4 + spill
calls	Call System *src* reg/lit/sfr if (*src* > 259) Protection-length fault if (stack switch required) { Perform Local Call using SPT } else { Perform Supervisor Call using SPT }	—	—	—	—	—	—	—	√	√	√	I C S	—	I C S	U	—	—	L	M	66:0	REG	μ	19 - 20 + spill	19 - 20 + spill
callx	Call Extended *efa* addr rip ← next IP temp ← (sp + 0x10) and not (0xf) memory (fp) ← r0:15 /* these accesses are cached in the local register cache */ PFP ← FP PFP.rt ← 000 FP ← temp SP ← fp + 64 IP ← *efa*	—	—	—	—	—	—	—	√	—	—	I C	—	I C	OP U OC	—	—	—	—	86	MEM	μ	6 + efa + spill	6 + efa + spill
chkbit	Check Bit *bitpos*, *src* reg/lit/sfr reg/lit/sfr if (*src* and 2^(*bitpos* mod 32) = 0) AC.cc1 ← 0; else AC.cc1 ← 1;	—	—	—	0	√	0	—	√	—	—	I	—	I	U	—	—	—	M	5A:E	REG	R	0.5 - 1	1
clrbit	Clear Bit *bitpos*, *src*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr *dst* ← *src* and not(2^(*bitpos* mod 32))	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	58:C	REG	R	0.5 - 1	1
cmpdeci	Compare and Decrement Integer *src1*, *src2*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr if (*src1* < *src2*) AC.cc ← 100; else if (*src1* = *src2*) AC.cc ← 010; else AC.cc ← 001; *dst* ← *src2* -1; /* overflow is ignored */	—	—	—	√	√	√	—	√	—	—	I	—	I	U	—	—	—	M	5A:7	REG	R	0.5 - 1	1

Mnemonic	Description	Arithmetic Controls						Process Controls				Trace Controls		Faults						Opcode	Opcode Format	Instruction Execution		
		nif	om	of	cc 2	cc 1	cc 0	p	tfp	em	te	Events	Modes	T	O	A	C	P	Y			Mach. Type	Instruction Issue	Result Latency
cmpdeco	Compare and Decrement Ordinal *src1*, *src2*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr if (*src1* < *src2*) AC.cc ← 100; else if (*src1* = *src2*) AC.cc ← 010; else AC.cc ← 001; *dst* ← *src2* -1;	—	—	—	√	√	√	—	√	—	—	I	—	I	U	—	—	—	M	5A:6	REG	R	0.5 - 1	1
cmpi	Compare Integer *src1*, *src2*, reg/lit/sfr reg/lit/sfr if (*src1* < *src2*) AC.cc ← 100; else if (*src2* = *src2*) AC.cc ← 010; else AC.cc ← 001;	—	—	—	√	√	√	—	√	—	—	I	—	I	U	—	—	—	M	5A:1	REG	R	0.5 - 1	1
cmpibe	Compare Integer and Branch If Equal *src1*, *src2*, *targ* reg/lit/sfr reg if (*src1* < *src2*) AC.cc ← 100; else if (*src1* = *src2*) AC.cc ← 010, IP ← *targ*; else AC.cc ← 001;	—	—	—	√	√	√	—	√	—	—	I B	—	I B	U	—	—	—	M	3A	COBR	C	1 - 3	3
cmpibg	Compare Integer and Branch If Greater *src1*, *src2*, *targ* reg/lit/sfr reg if (*src1* < *src2*) AC.cc ← 100; else if (*src1* = *src2*) AC.cc ← 010; else AC.cc ← 001, IP ← *targ*;	—	—	—	√	√	√	—	√	—	—	I B	—	I B	U	—	—	—	M	39	COBR	C	1 - 3	3
cmpibl	Compare Integer and Branch If Less *src1*, *src2*, *targ* reg/lit/sfr reg if (*src1* < *src2*) AC.cc ← 100, IP ← *targ*; else if (*src1* = *src2*) AC.cc ← 010; else AC.cc ← 001;	—	—	—	√	√	√	—	√	—	—	I B	—	I B	U	—	—	—	M	3C	COBR	C	1 - 3	3
cmpible	Compare Integer and Branch If Less Or Equal *src1*, *src2*, *targ* reg/lit/sfr reg if (*src1* < *src2*) AC.cc ← 100, IP ← *targ*; else if (*src1* = *src2*) AC.cc ← 010, IP ← *targ*; else AC.cc ← 001;	—	—	—	√	√	√	—	√	—	—	I B	—	I B	U	—	—	—	M	3E	COBR	C	1 - 3	3
cmpibne	Compare Integer and Branch If Not Equal *src1*, *src2*, *targ* reg/lit/sfr reg if (*src1* < *src2*) AC.cc ← 100, IP ← *targ*; else if (*src1* = *src2*) AC.cc ← 010; else AC.cc ← 001, IP ← *targ*;	—	—	—	√	√	√	—	√	—	—	I B	—	I B	U	—	—	—	M	3D	COBR	C	1 - 3	3
cmpibno	Compare Integer and Branch If Not Ordered *src1*, *src2*, *targ* reg/lit/sfr reg if (*src1* < *src2*) AC.cc ← 100; else if (*src1* = *src2*) AC.cc ← 010; else AC.cc ← 001;	—	—	—	√	√	√	—	√	—	—	I B	—	I B	U	—	—	—	M	38	COBR	C	1 - 3	3
cmpibo	Compare Integer and Branch If Ordered *src1*, *src2*, *targ* reg/lit/sfr reg if (*src1* < *src2*) AC.cc ← 100; else if (*src1* = *src2*) AC.cc ← 010; else AC.cc ← 001; IP ← *targ*	—	—	—	√	√	√	—	√	—	—	I B	—	I B	U	—	—	—	M	3F	COBR	C	1 - 3	3
cmpinci	Compare and Increment Integer *src1*, *src2*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr if (*src1* < *src2*) AC.cc ← 100; else if (*src1* = *src2*) AC.cc ← 010; else AC.cc ← 001; *dst* ← *src1* +1; /* overflow is ignored */	—	—	—	√	√	√	—	√	—	—	I	—	I	U	—	—	—	M	5A:5	REG	R	0.5 - 1	1
cmpinco	Compare and Increment Ordinal *src1*, *src2*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr if (*src1* < *src2*) AC.cc ← 100; else if (*src1* = *src2*) AC.cc ← 010; else AC.cc ← 001; *dst* ← *src1* +1;	—	—	—	√	√	√	—	√	—	—	I	—	I	U	—	—	—	M	5A:4	REG	R	0.5 - 1	1
cmpo	Compare Ordinal *src1*, *src2*, reg/lit/sfr reg/lit/sfr if (*src1* < *src2*) AC.cc ← 100; else if (*src1* = *src2*) AC.cc ← 010; else AC.cc ← 001;	—	—	—	√	√	√	—	√	—	—	I	—	I	U	—	—	—	M	5A:0	REG	R	0.5 - 1	1
cmpobe	Compare Ordinal and Branch If Equal *src1*, *src2*, *targ* reg/lit/sfr reg if (*src1* < *src2*) AC.cc ← 100; else if (*src1* = *src2*) AC.cc ← 010, IP ← *targ*; else AC.cc ← 001;	—	—	—	√	√	√	—	√	—	—	I B	—	I B	U	—	—	—	M	32	COBR	C	1 - 3	3
cmpobg	Compare Ordinal and Branch If Greater *src1*, *src2*, *targ* reg/lit/sfr reg if (*src1* < *src2*) AC.cc ← 100; else if (*src1* = *src2*) AC.cc ← 010; else AC.cc ← 001, IP ← *targ*;	—	—	—	√	√	√	—	√	—	—	I B	—	I B	U	—	—	—	M	31	COBR	C	1 - 3	3
cmpobl	Compare Ordinal and Branch If Less *src1*, *src2* *targ* reg/lit/sfr reg if (*src1* < *src2*) AC.cc ← 100, IP ← *targ*; else if (*src1* = *src2*) AC.cc ← 010; else AC.cc ← 001;	—	—	—	√	√	√	—	√	—	—	I B	—	I B	U	—	—	—	M	34	COBR	C	1 - 3	3
cmpoble	Compare Ordinal and Branch If Less Or Equal *src1*, *src2*, *targ* reg/lit/sfr reg if (*src1* < *src2*) AC.cc ← 100, IP ← *targ*; else if (*src1* = *src2*) AC.cc ← 010, IP ← *targ*; else AC.cc ← 001;	—	—	—	√	√	√	—	√	—	—	I B	—	I B	U	—	—	—	M	36	COBR	C	1 - 3	3
cmpobne	Compare Ordinal and Branch If Not Equal *src1*, *src2*, *targ* reg/lit/sfr reg if (*src1* < *src2*) AC.cc ← 100, IP ← *targ*; else if (*src1* = *src2*) AC.cc ← 010; else AC.cc ← 001, IP ← *targ*;	—	—	—	√	√	√	—	√	—	—	I B	—	I B	U	—	—	—	M	35	COBR	C	1 - 3	3
concmpi	Conditional Compare Integer *src1*, *src2*, reg/lit/sfr reg/lit/sfr if (AC.cc2 = 0) { if (*src1* ≤ *src2*) AC.cc ← 010; else AC.cc ← 001; }	—	—	—	√	√	√	—	√	—	—	I	—	I	U	—	—	—	M	5A:3	REG	R	0.5 - 1	1
concmpo	Conditional Compare Ordinal *src1*, *src2*, reg/lit/sfr reg/lit/sfr if (AC.cc2 = 0) { if (*src1* ≤ *src2*) AC.cc ← 010; else AC.cc ← 001; }	—	—	—	√	√	√	—	√	—	—	I	—	I	U	—	—	—	M	5A:2	REG	R	0.5 - 1	1
divi	Divide Integer *src1*, *src2*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr if (*src2* = 0) Arithmetic Zero Divide fault *dst* ← quotient (*src2*/*src1*) /* *src2*, *src1* and *dst* are 32-bits*/	—	—	↑	—	—	—	—	√	—	—	I	—	I	U	IO ZD	—	—	M	74:B	REG	R	0.5 - 1	39
divo	Divide Ordinal *src1*, *src2*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr if (*src2* = 0) Arithmetic Zero Divide fault *dst* ← quotient (*src2*/*src1*) /* *src2*, *src1* and *dst* are 32-bits */	—	—	—	—	—	—	—	√	—	—	I	—	I	U	ZD	—	—	M	70:B	REG	R	0.5 - 1	34,35
ediv	Extended Divide *src1*, *src2*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr if (*src2* = 0) Arithmetic Zero Divide fault *dst* ← remainder (*src2*/*src1*) *dst* + *1* ← quotient (*src2*/*src1*) /* *src2* is 64-bits, *src1*, *dst* and *dst* + 1 are 32-bits */	—	—	—	—	—	—	—	√	—	—	I	—	I	U	ZD	—	—	M	67:1	REG	R	0.5 - 1	34,35
emul	Extended Multiply *src1*, *src2*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr *dst* ← *src2* * *src* /* *src2* and *src1* are 32-bits*/ /* *dst* is 64-bits */	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	67:0	REG	R	0.5 - 1	4,6

Mnemonic	Description	Arithmetic Controls						Process Controls				Trace Controls		Faults						Opcode	Opcode Format	Instruction Execution		
		nif	om	of	cc 2	cc 1	cc 0	p	tfp	em	te	Events	Modes	T	O	A	C	P	Y			Mach. Type	Instruction Issue	Result Latency
eshro	Extended Shift Right Ordinal *src1*, reg/lit/sfr; *src2*, reg/lit/sfr; *dst*, rreg/sfr *dst* ← *src2* >> (*src1* mod 32)	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	5D:8	REG	R	0.5 - 1	1
extract	Extract *bitpos*, reg/lit/sfr; *len*, reg/lit/sfr; *src/dst*, reg/sfr *src/dst* ← (*src/dst* >> (*bitpos* mod 32)) and (2^(len mod 32) -1)	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	65:1	REG	μ	4	4
faulte	Fault If Equal if ((AC.cc and 010) ≠ 0) Constraint Range fault	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	R	—	—	1A	CTRL	μ	1 - 2	2
faultg	Fault If Greater if ((AC.cc and 001) ≠ 0) Constraint Range fault	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	R	—	—	19	CTRL	μ	1 - 2	2
faultge	Fault If Greater Or Equal if ((AC.cc and 011) ≠ 0) Constraint Range fault	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	R	—	—	1B	CTRL	μ	1 - 2	2
faultl	Fault If Less if ((AC.cc and 100) ≠ 0) Constraint Range fault	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	R	—	—	1C	CTRL	μ	1 - 2	2
faultle	Fault If Less Or Equal if ((AC.cc and 110) ≠ 0) Constraint Range fault	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	R	—	—	1E	CTRL	μ	1 - 2	2
faultne	Fault If Not Equal if ((AC.cc and 101) ≠ 0) Constraint Range fault	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	R	—	—	1D	CTRL	μ	1 - 2	2
faultno	Fault If Not Ordered if (AC.cc = 000) Constraint Range fault	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	R	—	—	18	CTRL	μ	1 - 2	2
faulto	Fault If Ordered if ((AC.cc and 111) ≠ 0) Constraint Range fault	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	R	—	—	1F	CTRL	μ	1 - 2	2
flushreg	Flush Local Registers Write all cached local register sets to memory Invalidate all local register cache locations	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	66:D	REG	μ	14 +23*# frames	14 +23*# frames
fmark	Force Mark if (PC.te = 1) { PC.tfp ← 1; TC.bte ← 1; Trace Breakpoint fault }	—	—	—	—	—	—	—	√	—	—	I BR	—	I BR	—	—	—	—	—	66:C	REG	μ		
ld	Load *efa*, addr; *dst*, reg *dst* ← memory_word(*efa*)	—	—	—	—	—	—	—	√	—	—	I	—	I	OP U OC	—	—	—	—	90	MEM	M or μ	1 + efa	1 + efa + bus
lda	Load Address *efa*, addr; *dst*, reg *dst* ← *efa*	—	—	—	—	—	—	—	√	—	—	I	—	I	OP OC	—	—	—	—	8C	MEM	M or μ	0.5 - 1 + efa	1 + efa
ldib	Load Integer Byte *efa*, addr; *dst*, reg *dst* ← memory_byte(*efa*) sign-extended	—	—	—	—	—	—	—	√	—	—	I	—	I	OP U OC	—	—	—	—	C0	MEM	M or μ	1 + efa	1 + efa + bus
ldis	Load Integer Short *efa*, addr; *dst*, reg *dst* ← memory_short(*efa*) sign-extended	—	—	—	—	—	—	—	√	—	—	I	—	I	OP U OC	—	—	—	—	C8	MEM	M or μ	1 + efa	1 + efa + bus
ldl	Load Long *efa*, addr; *dst*, reg *dst*, *dst*+1 ← memory_long(*efa*)	—	—	—	—	—	—	—	√	—	—	I	—	I	OP U OC	—	—	—	—	98	MEM	M or μ	1 + efa	1 + efa + bus
ldob	Load Ordinal Byte *efa*, addr; *dst*, reg *dst* ← memory_byte(*efa*) zero-extended	—	—	—	—	—	—	—	√	—	—	I	—	I	OP U OC	—	—	—	—	80	MEM	M or μ	1 + efa	1 + efa + bus
ldos	Load Ordinal Short *efa*, addr; *dst*, reg *dst* ← memory_short(*efa*) zero-extended	—	—	—	—	—	—	—	√	—	—	I	—	I	OP U OC	—	—	—	—	88	MEM	M or μ	1 + efa	1 + efa + bus
ldq	Load Quad *efa*, addr; *dst*, reg *dst*, *dst*+1, *dst*+2, *dst*+3 ← memory_quad(*efa*)	—	—	—	—	—	—	—	√	—	—	I	—	I	OP U OC	—	—	—	—	B0	MEM	M or μ	1 + efa	1 + efa + bus
ldt	Load Triple *efa*, addr; *dst*, reg *dst*, *dst*+1, *dst*+2 ← memory_triple(*efa*)	—	—	—	—	—	—	—	√	—	—	I	—	I	OP U OC	—	—	—	—	A0	MEM	M or μ	1 + efa	1 + efa + bus
mark	Mark if ((PC.te = 1) and (TC.btm = 1)) { PC.tfp ← 1; TC.bte ← 1; Trace Breakpoint fault }	—	—	—	—	—	—	—	√	—	—	I BR	—	I BR	U	—	—	—	—	66:B	REG	μ	1	1
modac	Modify AC *mask*, reg/lit/sfr; *src*, reg/lit/sfr; *dst*, reg/sfr temp ← AC AC ← (*src* and *mask*) or (AC and not (*mask*)) *dst* ← temp	√	√	√	√	√	√	—	√	—	—	I	—	I	U	—	—	—	M	64:5	REG	μ	11	11
modi	Modulo Integer *src1*, reg/lit/sfr; *src2*, reg/lit/sfr; *dst*, reg/sfr if (*src2* = 0) Arithmetic Zero Divide fault *dst* ← *src2* mod *src* /* *src2*, *src1*, and *dst* are 32 bits*/	—	—	↑	—	—	—	—	√	—	—	I	—	I	U	IO ZD	—	—	M	74:9	REG	R	0.5 - 1	34,35
modify	Modify *mask*, reg/lit/sfr; *src*, reg/lit/sfr; *src/dst*, reg *src/dst* ← (*src* and *mask*) or (*src/dst* and not(*mask*))	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	65:0	REG	μ	3	3
modpc	Modify PC *src*, reg/lit/sfr; *mask*, reg/lit/sfr; *src/dst*, reg if ((*mask* ≠ 0) and (PC.em = Supervisor)) Type Mismatch fault temp ← PC PC ← (*mask* and *src/dst*) or (PC and not (*mask*)) *src/dst* ← temp	—	—	—	—	—	—	√	√	√	√	I	—	I	U	—	—	—	M	65:5	REG	μ	10	10
modtc	Modify TC *mask*, reg/lit/sfr; *src*, reg/lit/sfr; *dst*, reg/sfr temp ← TC TC ← (*mask* and *src*) or (TC and not(*mask*)) *dst* ← temp	—	—	—	—	—	—	—	√	—	—	I, √	√	I	U	—	—	—	M	65:4	REG	μ	15	15
mov	Move *src*, reg/lit/sfr; *dst*, reg/sfr *dst* ← *src* /* 32-bits */	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	5C:C	REG	R	0.5 - 1	1
movl	Move Long *src*, reg/lit/sfr; *dst*, reg/sfr *dst* ← *src* /* 64-bits */	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	5D:C	REG	R	0.5 - 1	1
movq	Move Quad *src*, reg/lit/sfr; *dst*, reg/sfr *dst* ← *src* /* 128-bits */	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	5F:C	REG	μ	2	2
movt	Move Triple *src*, reg/lit/sfr; *dst*, reg/sfr *dst* ← *src* /* 96-bits */	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	5E:C	REG	μ	2	2
muli	Multiply Integer *src1*, reg/lit/sfr; *src2*, reg/lit/sfr; *dst*, reg/sfr *dst* ← *src2* * *src* /* *dst*, *src2*, *src1* are 32-bits*/	—	—	↑	—	—	—	—	√	—	—	I	—	I	U	IO	—	—	M	74:1	REG	R	0.5 - 1	4,5

Mnemonic	Description	Arithmetic Controls						Process Controls				Trace Controls		Faults						Opcode	Opcode Format	Instruction Execution		
		nif	om	of	cc 2	cc 1	cc 0	p	tfp	em	te	Events	Modes	T	O	A	C	P	Y			Mach. Type	Instruction Issue	Result Latency
mulo	Multiply Ordinal *src1*, *src2*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr *dst* ← *src2* * *src1* /* *dst*, *src2*, *src1* are 32-bits*/	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	70:1	REG	R	0.5 - 1	4,5
nand	Nand *src1*, *src2*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr *dst* ← not(*src2* and *src1*)	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	58:E	REG	R	0.5 - 1	1
nor	Nor *src1*, *src2*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr *dst* ← not(*src2* or *src1*)	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	58:8	REG	R	0.5 - 1	1
not	Not *src*, *dst* reg/lit/sfr reg/sfr *dst* ← not(*src*)	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	58:A	REG	R	0.5 - 1	1
notand	Not And *src1*, *src2*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr *dst* ← (not (*src2*)) and *src1*	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	58:4	REG	R	0.5 - 1	1
notbit	Not Bit *bitpos*, *src*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr *dst* ← *src* xor 2^(*bitpos* mod 32)	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	58:0	REG	R	0.5 - 1	1
notor	Not Or *src1*, *src2*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr *dst* ← (not (*src2*)) or *src1*	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	58:D	REG	R	0.5 - 1	1
or	Or *src1*, *src2*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr *dst* ← *src2* or *src1*	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	58:7	REG	R	0.5 - 1	1
ornot	Or Not *src1*, *src2*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr *dst* ← *src2* or (not(*src1*))	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	58:B	REG	R	0.5 - 1	1
remi	Remainder Integer *src1*, *src2*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr if (*src2* = 0) Arithmetic Zero Divide fault *dst* ← remainder (*src2*/*src1*) /* *src2*, *src1* and *dst* are 32-bits*/	—	—	↑	—	—	—	—	√	—	—	I	—	I	U	IO ZD	—	—	M	74:8	REG	R	0.5 - 1	34, 35
remo	Remainder Ordinal *src1*, *src2*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr if (*src2* = 0) Arithmetic Zero Divide fault *dst* ← remainder (*src2* ÷ *src1*) /* *src2*, *src1* and *dst* are 32-bits*/	—	—	—	—	—	—	—	√	—	—	I	—	I	U	ZD	—	—	M	70:8	REG	R	0.5 - 1	34, 35
ret	Return if ((PFP.rt = 001) or (PFP.rt = 111)) { /* ret from fault, or interrupt handler */ AC ← memory (FP -12) if (PC.em = Supervisor) PC ← memory(FP-16) } else if ((PFP.rt = 010) or (PFP.rt = 011) { /* ret to user procedure*/ PC.te ← PFP.rt0 PC.em ← User } FP ← PFP r0-15 ← memory (FP) /* these accesses are cached in the local register cache */ IP ← RIP	√	√	√	√	√	√	√	√	↓	√	I R P S	—	I R P S	U	—	—	—	—	0A	CTRL	μ	4 + fill	4 + fill
rotate	Rotate *len*, *src*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr *dst* ← *src* rotate_left (len mod 32)	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	59:D	REG	R	0.5 - 1	1
scanbit	Scan for Bit *src*, *dst* reg/lit/sfr reg/sfr if(*src* = 0) { *dst* ← 0xffff ffff AC.cc ← 000 } else { for (i =31; (*src* and 2^i) = 0; i ← i-1) *dst* ← i ; AC.cc ← 010 }	—	—	—	0	√	0	—	√	—	—	I	—	I	U	—	—	—	M	64:1	REG	μ	1	1
scanbyte	Scan Byte Equal *src1*, *src2*, reg/lit/sfr reg/lit/sfr if (((*src1* and 0x0000 00ff) = (*src2* and 0x0000 00ff)) or ((*src1* and 0x0000 ff00) = (*src2* and 0x0000 ff00)) or ((*src1* and 0x00ff 0000) = (*src2* and 0x00ff 0000)) or ((*src1* and 0xff00 0000) = (*src2* and 0xff00 0000))) AC.cc ← 010 else AC.cc ← 000	—	—	—	0	√	0	—	√	—	—	I	—	I	U	—	—	—	M	5A:C	REG	R	0.5 - 1	1
sdma	Setup DMA Channel *src1*, *src2*, *dst* reg/lit/sfr reg/lit/sfr reg/lit dma_channel_control (*src1*) ← *src* if (not chaining mode) DMA_RAM (*src1*) ← *src* /* quad store */ else DMA_RAM (*src1*) ← *src* /*word store */	—	—	—	—	, —	—	—	√	—	—	I	—	I	U	—	P	—	M	63:0	REG	μ	12 - 15	12 - 15
setbit	Set Bit *bitpos*, *src*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr *dst* ← *src* or 2^(*bitpos* mod 32)	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	58:3	REG	R	0.5 - 1	1
shli	Shift Left Integer *len*, *src*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr if (len > 32) i ← 32 else i ← len temp ← *src* while ((temp.31 = temp.30) and (i = 0)) { temp << 1 i-- } *dst* ← temp	—	—	↑	—	—	—	—	√	—	—	I	—	I	U	IO	—	—	M	59:E	REG	R	0.5 - 1	1
shlo	Shift Left Ordinal *len*, *src*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr if (len < 32) *dst* ← *src* << len else *dst* ← 0	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	59:C	REG	R	0.5 - 1	1
shrdi	Shift Right Dividing Integer *len*, *src*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr if (len > 32) i ← 32; else i ← *len*; temp ← *sre*; s_sign ← temp.bit31 lost_bit ← 0 while (i ≠ 0) { lost_bit ← lost_bit or temp.bit0; temp ← temp >> 1; temp.bit31 ← temp.bit30; i ← i - 1; } if ((s_sign = 1) and (lost_bit = 1)) temp ← temp + 1; *dst* ← temp;	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	59:A	REG	R	0.5 - 1	1 + 4 if fixup needed

Mnemonic	Description	Arithmetic Controls						Process Controls				Trace Controls		Faults						Opcode	Opcode Format	Instruction Execution		
		nif	om	of	cc 2	cc 1	cc 0	p	tfp	em	te	Events	Modes	T	O	A	C	P	Y			Mach. Type	Instruction Issue	Result Latency
shri	Shift Right Integer *len*, *src*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr if (len > 32) i ← 32; else i ← *len*; temp ← *sre*; while ((temp.31 = temp.30) and (i ≠ 0)) { temp ← temp >> 1; temp.bit31 ← temp.bit30; i ← i - 1; } *dst* ← temp;	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	59:B	REG	R	0.5 - 1	1
shro	Shift Right Ordinal *len*, *src*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr if (len < 32) *dst* ← *src* >> len else *dst* ← 0	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	59:8	REG	R	0.5 - 1	1
spanbit	Span Over Bit *src*, *dst* reg/lit/sfr reg/sfr if(*src* = 0xffff ffff) { *dst* ← 0xffff ffff AC.cc ← 000 } else { for (i =31; (*src* and 2^i) ≠ 0; i ← i - 1); *dst* ← i AC.cc ← 010 }	—	—	—	0	√	0	—	√	—	—	I	—	I	U	—	—	—	M	64:0	REG	μ	2	2
st	Store *src*, *efa* reg/lit addr memory_word(*efa*) ← *src*	—	—	—	—	—	—	—	√	—	—	I	—	I	OP U OC	—	—	—	M	92	MEM	M or μ	0.5 - 1 + efa	
stib	Store Integer Byte *src*, *efa* reg/lit addr memory_byte(*efa*) ← *src* /* truncated to 8 bits */	—	—	↑	—	—	—	—	√	—	—	I	—	I	OP U OC	IO	—	—	M	C2	MEM	M or μ	0.5 - 1 + efa	
stis	Store Integer Short *src*, *efa* reg/lit addr memory_short(*efa*) ← *src* /* truncated to 16-bits */	—	—	↑	—	—	—	—	√	—	—	I	—	I	OP U OC	IO	—	—	M	CA	MEM	M or μ	0.5 - 1 + efa	
stl	Store Long *src*, *efa* reg/lit addr memory_long(*efa*) ← *src*, *src*+1	—	—	—	—	—	—	—	√	—	—	I	—	I	OP U OC	—	—	—	M	9A	MEM	M or μ	0.5 - 1 + efa	
stob	Store Ordinal Byte *src*, *efa* reg/lit addr memory_byte(*efa*) ← *src* /*truncated to 8-bits */	—	—	—	—	—	—	—	√	—	—	I	—	I	OP U OC	—	—	—	M	82	MEM	M or μ	0.5 - 1 + efa	
stos	Store Ordinal Short *src*, *efa* reg/lit addr memory_short(*efa*) ← *src* /* truncated to 16-bits */	—	—	—	—	—	—	—	√	—	—	I	—	I	OP U OC	—	—	—	M	8A	MEM	M or μ	0.5 - 1 + efa	
stq	Store Quad *src*, *efa* reg/lit addr memory_quad(*efa*) ← *src*, *src*+1, *src*+2, *src*+3	—	—	—	—	—	—	—	√	—	—	I	—	I	OP U OC	—	—	—	M	B2	MEM	M or μ	0.5 - 1 + efa	
stt	Store Triple *src*, *efa* reg/lit addr memory_triple(*efa*) ← *src*, *src*+1, *src*+2	—	—	—	—	—	—	—	√	—	—	I	—	I	OP U OC	—	—	—	M	A2	MEM	M or μ	0.5 - 1 + efa	
subc	Subtract Ordinal With Carry *src1*, *src2*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr *dst* ← *src2* - *src1* - not(AC.cc1) AC.cc0 ← integer overflow AC.cc1 ← carry out	—	—	—	0	√	√	—	√	—	—	I	—	I	U	—	—	—	M	5B:2	REG	R	0.5 - 1	1
subi	Subtract Integer *src1*, *src2*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr *dst* ← *src2* - *src1*	—	—	↑	—	—	—	—	√	—	—	I	—	I	U	IO	—	—	M	59:3	REG	R	0.5 - 1	1
subo	Subtract Ordinal *src1*, *src2*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr *dst* ← *src2* - *src1*	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	59:2	REG	R	0.5 - 1	1
syncf	Synchronize Faults if (AC.nif ≠ 1) { wait until no imprecise fault could occur associated with instructions which have begun, but are not completed. }	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	—	66:F	REG	μ	1	1
sysctl	System Control *src1*, *src2*, *src3* reg/lit/sfr reg/lit/sfr reg/lit i ← (*src1* and 0xff) >> 8 switch (i)																			65:9	REG	μ		
	case 0: Post an Interrupt break;	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M				20-30	20-30
	case 1: Purge the Instruction Cache break;	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M				13	13
	case 2: Configure Instruction Cache break;	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M				20 + bus	20 + bus
	case 3: Software Reset break;	0	0	0	0	0	0	31	√	1	0	I	0	I	U	—	—	—	M				195 + bus	195 + bus
	case 4: Load Control Register Group break;	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M				33 + bus	33 + bus
	default: Operation Invalid Operand fault return	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?				?	?
teste	Test For Equal *dst* reg/sfr if ((AC.cc and 010) ≠ 0) *dst* ← 1 else *dst* ← 0	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	22	COBR	μ	1 - 2	1 - 2
testg	Test For Greater *dst* reg/sfr if ((AC.cc and 001) ≠ 0) *dst* ← 1 else *dst* ← 0	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	21	COBR	μ	1 - 2	1 - 2
testge	Test For Greater Or Equal *dst* reg/sfr if ((AC.cc and 011) ≠ 0) *dst* ← 1 else *dst* ← 0	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	23	COBR	μ	1 - 2	1 - 2
testl	Test For Less *dst* reg/sfr if ((AC.cc and 100) ≠ 0) *dst* ← 1 else *dst* ← 0	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	24	COBR	μ	1 - 2	1 - 2
testle	Test For Less Or Equal *dst* reg/sfr if ((AC.cc and 110) ≠ 0) *dst* ← 1 else *dst* ← 0	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	26	COBR	μ	1 - 2	1 - 2
testne	Test For Not Equal *dst* reg/sfr if ((AC.cc and 101) ≠ 0) *dst* ← 1 else *dst* ← 0	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	25	COBR	μ	1 - 2	1 - 2
testno	Test For Not Ordered *dst* reg/sfr if ((AC.cc = 000) ≠ 0) *dst* ← 1 else *dst* ← 0	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	20	COBR	μ	1 - 2	1 - 2

Mnemonic	Description	Arithmetic Controls						Process Controls				Trace Controls		Faults						Opcode	Opcode Format	Instruction Execution		
		nif	om	of	cc 2	cc 1	cc 0	p	tfp	em	te	Events	Modes	T	O	A	C	P	Y			Mach. Type	Instruction Issue	Result Latency
testo	Test For Ordered *dst* reg/sfr if ((AC.cc and 111) ≠ 0) *dst* ← 1 else *dst* ← 0	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	27	COBR	μ	1 - 2	1 - 2
udma	Update DMA Channel Copy DMA working registers to on-chip DMA ram	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	P	—	—	63:1	REG	μ	1	1
xnor	Exclusive Nor *src1*, *src2*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr *dst* ← not(*src2* or *src1*) or (*src2* and *src1*)	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	58:9	REG	R	0.5 - 1	1
xor	Exclusive Or *src1*, *src2*, *dst* reg/lit/sfr reg/lit/sfr reg/sfr *dst* ← (*src2* or *src1*) and not (*src2* and *src1*)	—	—	—	—	—	—	—	√	—	—	I	—	I	U	—	—	—	M	58:6	REG	R	0.5 - 1	1

UNITED STATES
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

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 Cedex

UNITED KINGDOM
Intel Corporation (U.K.) Ltd.
Pipers Way
Swindon
Wiltshire, England SN3 1RJ

WEST 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./0989/25K/RRD JB
Embedded Processors

ISBN 1-55512-099-7