Introduction to i960™ Architecture

2

CHAPTER 2
INTRODUCTION TO i960™ ARCHITECTURE

This chapter provides an overview of the architecture on which the 80960 series of processors is based.

AN EMBEDDED 32-BIT ARCHITECTURE FROM INTEL

The 80960SA/SB processor marks the continuation of the i960 architecture series -- an embedded 32-bit architecture from Intel. This architecture has been designed to meet the needs of embedded applications such as machine control, robotics, process control, avionics, and instrumentation. It represents a renewed commitment from Intel to provide reliable, high-performance processors and controllers for the embedded processor marketplace.

The i960 architecture can best be characterized as a high-performance computing engine. It features high-speed instruction execution and ease of programming. It is also easily extensible, allowing processors and controllers based on this architecture to be conveniently customized to meet the needs of specific processing and control applications.

Some of the important attributes of the i960 architecture include:

- full 32-bit registers
- high-speed, pipelined instruction execution
- a convenient program execution environment with 32 general-purpose registers and a versatile set of special-function registers
- a highly optimized procedure call mechanism that features on-chip caching of local variables and parameters
- extensive facilities for handling interrupts and faults
- extensive tracing facilities to support efficient program debugging and monitoring
- register scoreboarding and write buffering to permit efficient operation with lower performance memory subsystems

The following sections describe those features of the i960 architecture that are provided to streamline code execution and simplify programming. Also described are those features that allow extensions to be added to the architecture. Later sections describe the execution environment and hardware considerations of an 80960SA/SB system.

HIGH PERFORMANCE PROGRAM EXECUTION

Much of the design of the i960 architecture has been aimed at maximizing the processor's computational and data processing speed through increased parallelism. The following paragraphs describe several of the mechanisms and techniques used to accomplish this goal, including:

- an efficient load and store memory-access model
- caching of code
- overlapped execution of instructions
- many one or two clock-cycle instructions

Load and Store Model

One of the more important features of the i960 architecture is that most of its operations are performed on operands in registers, rather than in memory. For example, all the arithmetic, logical, comparison, branching, and bit operations are performed with registers and literals.

This feature provides two benefits. First, it increases program execution speed by minimizing the number of memory accesses required to execute a program. Second, it reduces memory latency encountered when using slower, lower-cost memory parts by permitting concurrent ALU and memory access operations.

To support this concept, the architecture provides a generous supply of general-purpose registers. For each procedure, 32 registers are available (28 of which are available for general use). These registers are divided into two types: global and local. Both these types of registers can be used for general storage of operands. The only difference is that global registers retain their contents across procedure boundaries, whereas the processor allocates a new set of local registers each time a new procedure is called.

The architecture also provides a set of fast, versatile load and store instructions. These instructions allow burst transfers of 1, 2, 4, 8, 12, or 16 bytes of information between memory and the registers.

On-Chip Caching of Code and Data

To further reduce memory accesses, the architecture offers two mechanisms for caching code and data on chip: an instruction cache and multiple sets of local registers. The instruction cache allows pre-fetching of blocks of instruction from memory, which helps insure that the instruction execution pipeline is supplied with a steady stream of instructions. It also reduces the number of memory accesses required when performing iterative operations such as loops. (The size of the instruction cache can vary between components. With the 80960SA/SB processor, it is 512 bytes.)

To optimize the architecture's procedure call mechanism, the processor provides multiple sets of local registers. This allows the processor to perform most procedure calls without having to write the local registers out to the stack in memory.

(The number of local-register sets provided depends on the processor implementation. The 80960SA/SB processor provides four sets of local registers.)

Overlapped Instruction Execution

Another technique that the i960 architecture employs to enhance program execution speed is overlapping the execution of some instructions. This is accomplished through register scoreboarding.

Register scoreboarding permits instruction execution to continue while data is being fetched from memory. When a load instruction is executed, the processor sets one or more scoreboard bits to indicate the target registers to be loaded. After the target registers are loaded, the scoreboard bits are cleared. While the target registers are being loaded, the processor is allowed to execute other instructions that do not use these registers. The processor uses the scoreboard bits to insure that target registers are not used until the loads are complete. (The checking of scoreboard bits is transparent to software.) The net result of using this technique is that code can often be optimized in such a way as to allow some instructions to be executed in parallel.

Single-Clock Instructions

It is the intent of the i960 architecture that a processor be able to execute commonly used instructions such as move, add, subtract, logical operations, compare and branch in a minimum number of clock cycles (preferably one clock cycle). The architecture supports this concept in several ways. For example, the load and store model described earlier in this chapter (with its concentration on register-to-register operations) allows simple operations to be performed without the overhead of memory-to-memory operations.

Also, all the instructions in the i960 architecture are 32 bits or 64 bits long and aligned on 32-bit boundaries. This feature allows instructions to be decoded in one clock cycle. It also eliminates the need for an instruction-alignment stage in the pipeline.

The design of the 80960SA/SB processor takes full advantage of these features of the architecture, resulting in over 50 instructions that can be executed in a single clock-cycle.

Efficient Interrupt Model

The i960 architecture provides an efficient mechanism for servicing interrupts from external sources. To handle interrupts, the processor maintains an interrupt table of 248 interrupt vectors (240 of which are available for general use). When an interrupt is signaled, the processor uses a pointer from the interrupt table to perform an implicit call to an interrupt handler procedure. In performing this call, the processor automatically saves the state of the processor prior to receiving the interrupt; performs the interrupt routine; and then restores the state of the processor. A separate interrupt stack is also provided to segregate interrupt handling from application programs.

The interrupt handling facilities also feature a method of prioritizing interrupts. Using this technique, the processor is able to store interrupts that are lower in priority than the task the processor is currently working on in a pending interrupt section of the interrupt table. At certain defined times, the processor checks the pending interrupts and services them.

SIMPLIFIED PROGRAMMING ENVIRONMENT

Partly as a side benefit of its streamlined execution environment and partly by design, processors based on the i960 architecture are particularly easy to program. For example, the large number of general purpose registers allows relatively complex algorithms to be executed with a minimum number of memory accesses. The following paragraphs describe some of the other features that simplify programming.

Highly Efficient Procedure Call Mechanism

The procedure call mechanism makes procedure calls and parameter passing between procedures simple and compact. Each time a call instruction is issued, the processor automatically saves the current set of local registers and allocates a new set of local registers for the called procedure. Likewise, on a return from a procedure, the current set of local registers is deallocated and the local registers for the procedure being returned to are restored. On a procedure call, the program thus never has to explicitly save and restore those local variables and parameters that are stored in local registers.

Versatile Instruction Set and Addressing

The selection of instructions and addressing modes also simplifies programming. The architecture offers a full set of load, store, move, arithmetic, comparison, and branch instructions, with operations on both integer and ordinal data types. It also provides a complete set of Boolean and bit-field instructions, to simplify operations on bits and bit strings.

The addressing modes are efficient and straightforward, while at the same time providing the necessary indexing and scaling modes required to address complex arrays and record structures.

The large 4-gigabyte address space provides ample room to store programs and data. The availability of 32 addressing lines allows some address lines to be memory-mapped to control hardware functions.

Extensive Fault Handling Capability

To aid in program development, the i960 architecture defines a wide selection of faults that the processor detects, including arithmetic faults, invalid operands, invalid operations, and machine faults. When a fault is detected, the processor makes an implicit call to a fault handler routine, using a mechanism similar to that described above for interrupts. The information collected for each fault allows program developers to quickly correct faulting code. It also allows automatic recovery from some faults.

Debugging and Monitoring

To support debugging systems, the i960 architecture provides a mechanism for monitoring processor activity by means of trace events. The processor can be configured to detect as many as seven different trace events, including branches, calls, supervisor calls, returns, prereturns, breakpoints, and the execution of any instruction. When the processor detects a trace event, it signals a trace fault and calls a fault handler. Intel provides several tools that use this feature, including an in-circuit emulator (ICE) device.

SUPPORT FOR ARCHITECTURAL EXTENSIONS

The i960 architecture described earlier in this chapter provides a high-performance computing engine for use as the computational and data processing core of embedded processors or controllers. The architecture also provides several features that enable processors based on this architecture to be easily customized to meet the needs of specific embedded applications, such as signal processing, array processing, or graphics processing.

The most important of these features is a set of 32 special function registers. These registers provide a convenient interface to circuitry in the processor or to pins that can be connected to external hardware. They can be used to control timers, to perform operations on special data types, or to perform I/O functions.

The special function registers are similar to the global registers. They can be addressed by all the register-access instructions.

EXTENSIONS INCLUDED IN THE 80960SA/SB PROCESSORS

The 80960 series of processors offer a complete implementation of the i960 architecture, plus several extensions to the architecture. These extensions fall into two categories: floating-point processing and intra-agent communication.

On-Chip Floating Point

The 80960SB processor provides a complete implementation of the IEEE standard for binary floating-point arithmetic (IEEE 754-1985). This implementation includes a full set of floating-point operations, including add, subtract, multiply, divide, trigonometric functions, and logarithmic functions. These operations are performed on single precision (32-bit), double precision (64-bit), and extended precision (80-bit) real numbers.

One benefit of this implementation is that the floating-point handling facilities are completely integrated into the normal instruction execution environment. Single- and double-precision floating-point values are stored in the same registers as non-floating point values. Also, four 80-bit floating-point registers are provided to hold extended-precision values.

OVERVIEW OF THE EXECUTION ENVIRONMENT

The next few sections describe how the 80960SA/SB processor executes instructions and how it stores and manipulates data. The parts of the execution environment discussed include the address space, the register model, the instruction pointer, and the arithmetic controls.

The execution environment's procedure stack and procedure-call mechanism are described in Chapter 4.

When the 80960SA/SB processor is initialized, it sets up an execution environment. It then begins executing instructions from a program, using this execution environment to store and manipulate data.

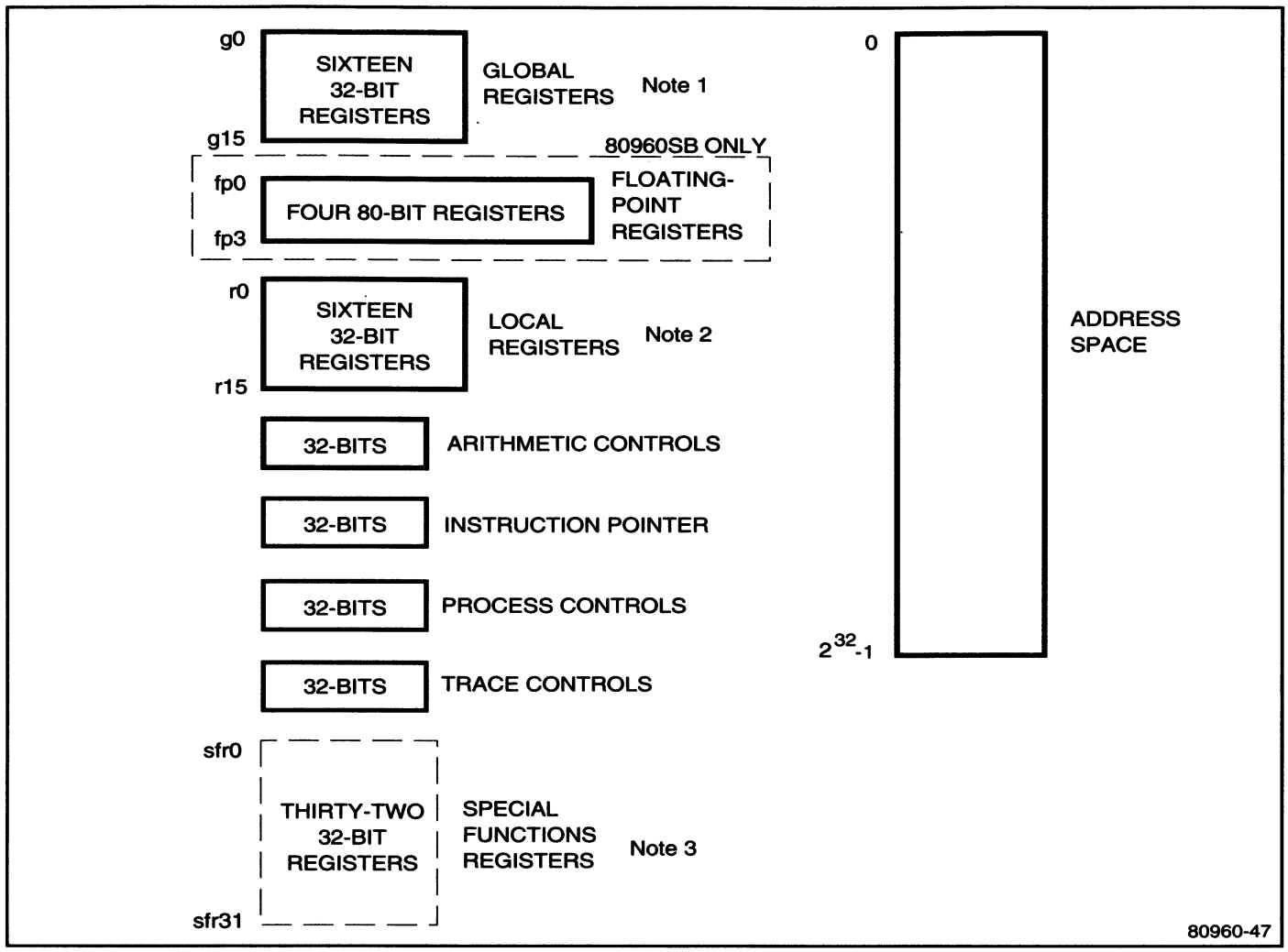
Figure 2-1 shows the part of the execution environment that the processor sets up to execute a procedure within a program. This environment consists of a 2^{32}-byte address space, a set of global and floating-point registers (80960SB only), a set of local registers, a set of arithmetic-control bits, the instruction pointer, a set of process-control bits, and a set of trace-control bits. All of these items, except the address space, reside on the 80960SA/SB chip.

NOTE

The floating-point registers shown in Figure 2-1 are not defined in the i960 architecture. They are extensions to the architecture that have been added to the 80960SB processor to support floating-point operations on the extended-real (floating point) data type.

The 32 special-function registers (shown in Figure 2-1 in a dashed box) are defined in the i960 architecture. These registers are not implemented in the 80960SA/SB processor.

When the instruction stream includes a procedure call, a procedure stack and some additional elements are added to this execution environment. These procedure-call related elements are shown and discussed in Chapter 4.



NOTES:

1. REGISTER g15 IS RESERVED FOR STACK MANAGEMENT FUNCTIONS.
2. REGISTERS r0, r1, AND r2 ARE RESERVED FOR STACK MANAGEMENT FUNCTIONS.
3. SPECIAL FUNCTION REGISTERS ARE NOT IMPLEMENTED IN THIS PROCESSOR.

Figure 2-1: Execution Environment

ADDRESS SPACE

From the point of view of the processor, the address space is flat (unsegmented) and byte addressable, with addresses running contiguously from 0 to 2^{32} - 1. Programs and the kernel can allocate space for data, instructions, and the stack anywhere within this space, with the following exceptions:

- Instructions must be aligned on word boundaries.
- Some of the addresses in the upper 16M Bytes of the address space (addresses $FF000000_{16}$ through $FFFFFFFF_{16}$) are reserved for specific functions. In general, programs and the kernel should not use this section of the address space.

The memory requirements to support this address space are given in Chapter 3, in the section titled "Memory Requirements."

REGISTER MODEL

The processor provides three types of data registers: global, floating-point, and local. The 16 global registers constitute a set of general-purpose registers, the contents of which are preserved across procedure boundaries. (On the 80960SB processor, The 4 floating-point registers are provided to support extended floating-point arithmetic.) Their contents are also preserved across procedure boundaries. The 16 local registers are provided to hold parameters specific to a procedure (i.e., local variables). For each procedure that is called, the processor allocates a separate set of 16 local registers.

For any one procedure within a program, 36 registers are thus available on the 80960SB processor (as shown in Figure 2-2): the 16 global registers, the 4 floating-point registers, and the 16 local registers. All of these registers are maintained on the processor chip. The 32 global and local registers are available on the 80960SA processor.

Global Registers

The 16 global registers (g0 through g15) are 32-bit registers. Each register can thus hold a word (32 bits) of data. Registers g0 through g14 are general-purpose registers; 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 4.)

The general-purpose global registers (g0 through g14) can hold any of the data types that the processor recognizes (i.e., ordinals, integers, reals).

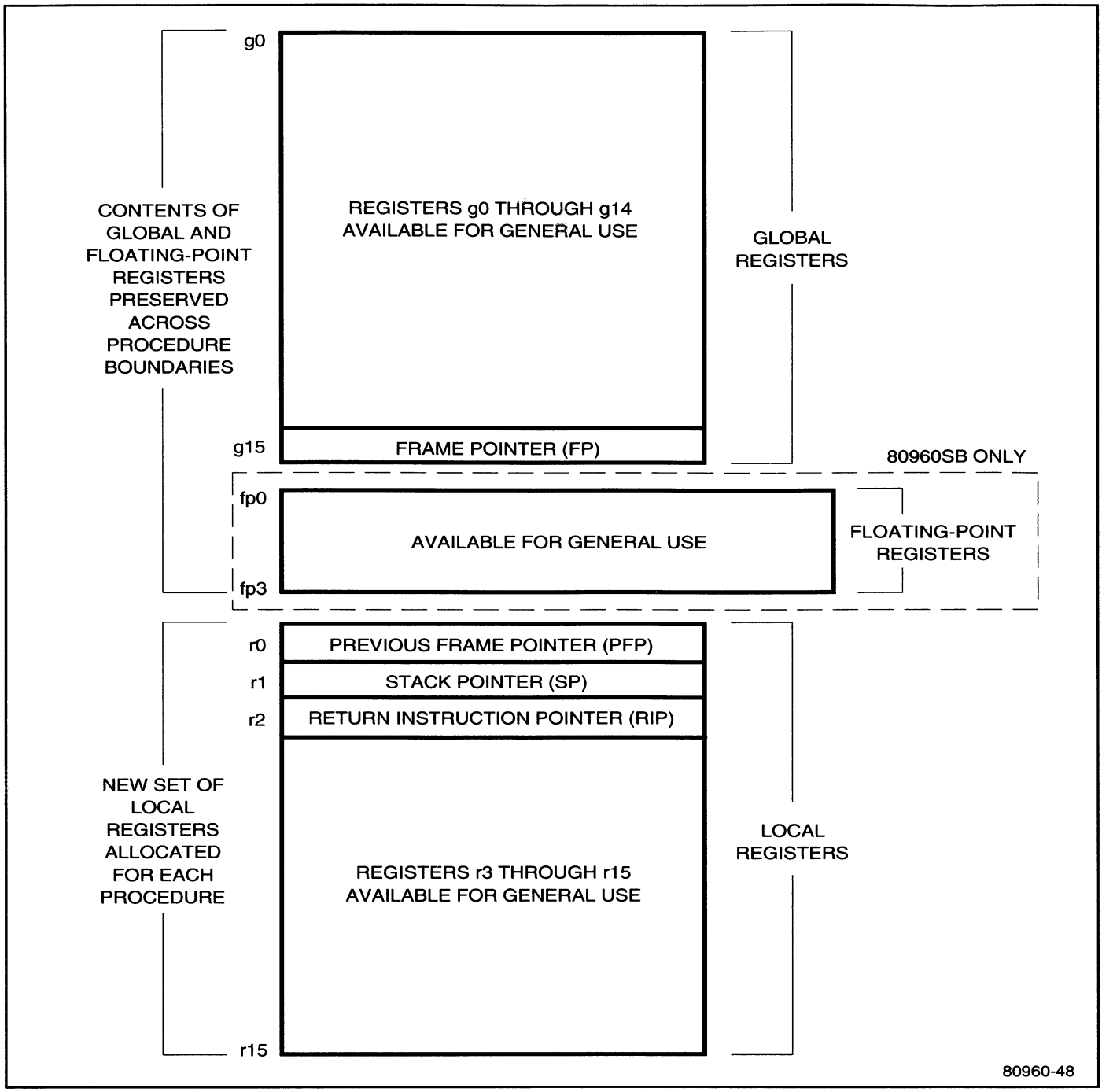


Figure 2-2: Registers Available to a Single Procedure

Floating-Point Registers

The four floating-point registers (fp0 through fp3) are 80-bit registers. These registers can be accessed only as operands of floating-point instructions. All numbers stored in these registers are stored in extended-real format. (This format is described in Chapter 10.) The processor automatically converts floating-point values from real or long-real format into extended-real format when a floating-point register is used as a destination for an instruction.

NOTE

The floating-point registers are defined in the i960 architecture as an option for processors such as the 80960SB that support floating-point operations. These registers are omitted from implementations of the architecture that do not support floating-point operations.

Local Registers

The 16 local registers (r0 through r15) are 32-bit registers, like the global registers. The purpose of the local registers is to provide a separate set of registers, aside from the global and floating-point registers, for each active procedure. Each time a procedure is called, the processor automatically sets up a new set of local registers for that procedure and saves the local registers for 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 4.) The processor accesses the local registers at the same speed as it does the global registers.

Register Alignment

Several of the processor's instructions operate on multiple-word operands. For example, the load-long instruction (**ldl**) loads two words from memory into two consecutive registers. Here, the register number for the least significant word is specified in the instruction and the most-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 value is undefined. If a register reference for a destination value is not properly aligned, the registers that the processor writes to are undefined.

Register Scoreboarding

The 80960SA/SB provides a mechanism called *register scoreboarding* that in certain situations permits instructions to be executed concurrently. This mechanism works as follows. While an instruction is being executed, the processor sets a 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 in some instances is able to execute those instructions before execution of the prior instruction is complete. In effect, the register scoreboarding mechanism allows some instructions to be executed in parallel.

A common application of this feature is to execute one or more fast instructions (instructions that take one to three clock cycles) concurrently with load instructions. A load instruction typically takes 4 to 10 clock cycles or more (depending on the design of system memory and the addressing mode used). Register scoreboarding allows other instructions to be executed concurrently with the load instruction, provided the other instructions do not affect the registers being loaded. For example, the following group of instructions loads a group of local registers while performing some other operations on data in global registers.

```
ld xyz, r6              # r6 ← data from address xyz
addi g4, g6, g7         # g7 ← g4 + g6
addi g9, g10, g11       # g11 ← g9 + g10
ld abc, r8              # r8 ← data from address abc
and g0, 0xffff, g1      # g1 ← g0 AND 0xffff
addi r6, r8, r7         # r7 ← r6 + r8
```

Here, the two **addi** instructions following the first load and the **and** instruction following the second load are performed concurrently with the bus accesses of the two load instructions.

(Appendix C provides a detailed discussion of the processor's instruction-execution pipeline and register scoreboarding.)

INSTRUCTION POINTER

The instruction pointer (IP) is the address (in the address space) of the instruction currently being executed. This address is 32 bits; however, since instructions are required to be aligned on word boundaries in memory, the 2 least-significant bits of the IP are always zero.

Instructions in the processor are one or two words long. The IP gives the address of the lowest order byte of the first word of the instruction.

The IP is stored in the processor and cannot be read directly. However, the IP-with-displacement addressing mode allows the IP to be used as an offset into the address space. This addressing mode can also be used with the **lda** (load address) instruction to read the current value of the IP.

When a break occurs in the instruction stream (due to an interrupt or a procedure call), the IP of the next instruction to be executed (i.e., the RIP) is stored in local register r2, which is then stored on the stack. Refer to Chapter 4 for further discussion of this operation.

ARITHMETIC CONTROLS

The processor's arithmetic controls are made up of a set of 32 bits, which are cached on the processor chip in the arithmetic-controls register. Figures 2-3 and 2-4 show the arrangement of the arithmetic controls bits for each processor. The arithmetic controls bits include condition code flags; floating-point control and status flags and masks; integer control and status flags; and a flag that controls faulting on imprecise faults.

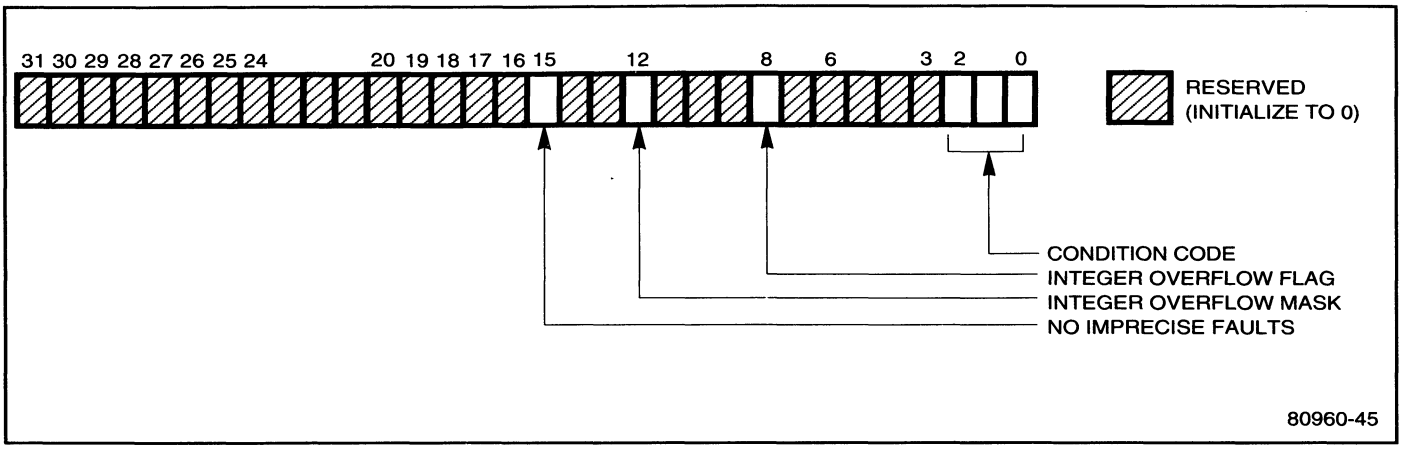


Figure 2-3: Arithmetic Controls (80960SA)

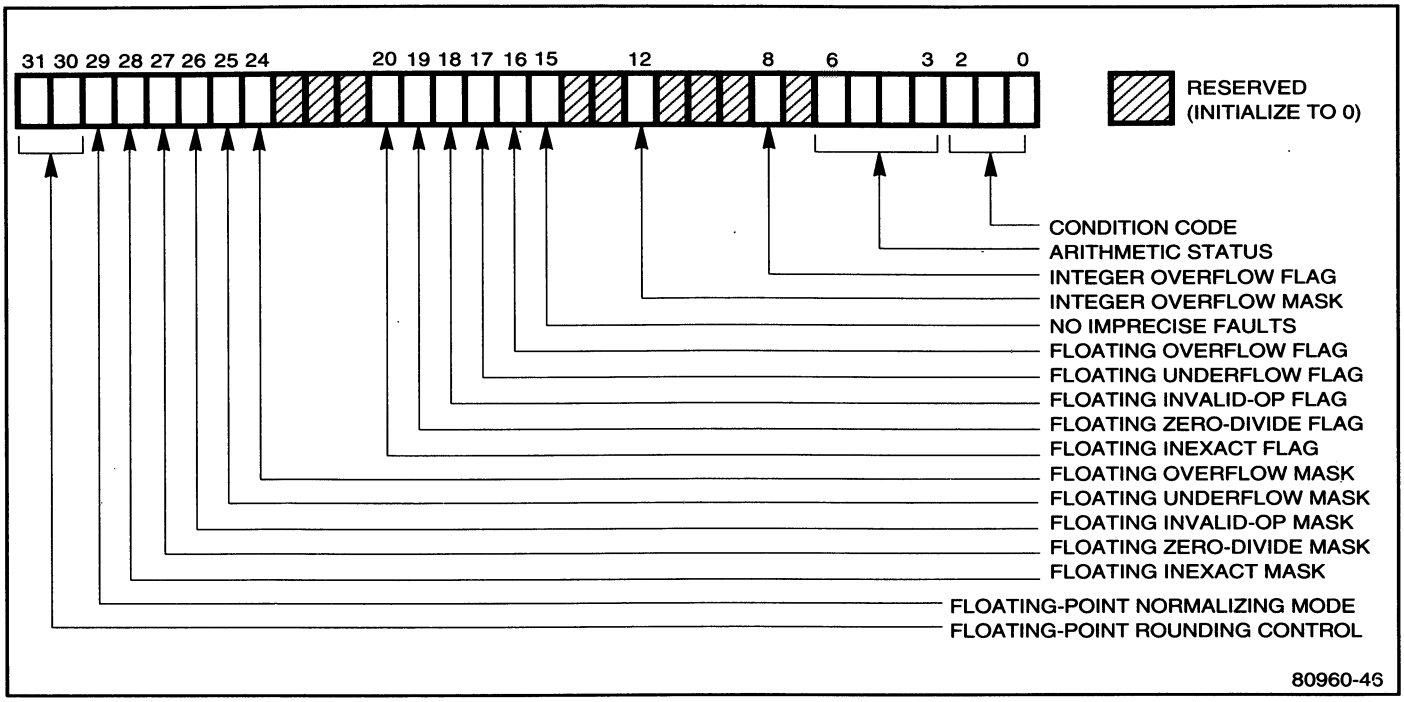


Figure 2-4: Arithmetic Controls (80960SB)

The processor sets or clears these bits to show the results of certain operations. For example, the processor modifies the condition code flags after each comparison operation to show the result of the comparison. Other arithmetic control bits, such as the floating-point fault masks, are set by the currently running program to tell the processor how to respond to certain fault conditions.

NOTE

The arithmetic-status flags and the floating-point flags and masks are not defined in the i960 architecture. They are an extension to the architecture, which is provided in the 80960SA/SB processor to support floating-point operations. For implementations of the architecture that do not support floating-point operations, these flags and masks are reserved bits.

Initializing and Modifying the Arithmetic Controls

The state of the processor's arithmetic controls is undefined at processor initialization or on a processor reinitialize (initiated with a reinitialize processor IAC). Part of the initialization code should thus be to set the arithmetic controls to a specific state.

The arithmetic controls can be examined and modified using the modify arithmetic controls (**modac**) instruction. This instruction uses a mask to allow specific bits to be changed.

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, respectively.

The **modac** instruction can be used to explicitly save and restore the contents of the arithmetic controls.

Functions of the Arithmetic-Controls Bits

The functions of the various arithmetic controls bits are as follows:

NOTE

In the following discussion, some of the arithmetic controls bits are referred to as "sticky flags." A sticky flag is one that the processor never implicitly clears. Once the processor sets a sticky flag to indicate that a particular condition has occurred, the flag remains set until the program explicitly clears it.

Condition-Code Flags

The processor sets the condition-code flags (bits 0-2) to indicate the results of certain instructions (usually compare instructions). Other instructions, such as conditional-branch instructions, examine these flags and perform functions according to their state. Once the processor has set these flags, it leaves them unchanged until it executes another instruction that uses these flags to store results.

These flags are used to show either true or false conditions or inequalities (greater-than, equal, or less-than conditions). To show true or false conditions, the processor sets the flags as shown in Table 2-1.

Table 2-1: Condition Codes for True or False Conditions

Condition Code	Condition
010	true
000	false

To show inequalities, the processor sets the condition-code flags as shown in Table 2-2.

Table 2-2: Condition Codes for Inequality Conditions

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

Certain instructions (such as the branch-if instructions) use a 3-bit mask to evaluate the condition-code flags. For example the branch-if-greater-or-equal instruction (**bge**) uses a mask of 011_2 to determine if the condition code is set to either greater-than or equal. These masks cover the additional conditions of greater-or-equal, less-or-equal (110_2), not-equal (101_2), and ordered (111_2).

The term unordered is 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." Refer to the section in Chapter 10 titled "Comparison, Branching and Classification" for further information about the ordered and unordered conditions.

Arithmetic-Status Flags (80960SB Only)

The processor uses the arithmetic-status field (bits 2-6) in conjunction with the classify instructions (**classr** and **classrl**) to show the class of a floating-point number. When executing these instructions, the processor sets the bits in the arithmetic-status field as shown in Table 2-3, according to the class of the value being classified.

Table 2-3: Encoding of Arithmetic-Status Field

Arithmetic Status	Classification
s000	zero
s001	denormalized number
s010	normal finite number
s011	infinity
s100	quiet NaN
s101	signaling NaN
s110	reserved operand

The "s" bit is set to the sign of the value being classified.

The remainder real instructions (**remr** and **remrl**) also use the arithmetic status field as described in Chapter 9.

No-Imprecise-Faults Flag

The no-imprecise-faults (NIF) flag (located at bit 15 of the arithmetic controls) determines whether or not imprecise faults are allowed to be raised. If set, faults are required to be precise; if clear, certain faults can be imprecise. (Refer to the section in Chapter 6 titled "Precise and Imprecise Faults" for more information about this flag.)

NOTE

In the 80960SA/SB implementations of the i960 architecture, the NIF flag is ignored. These implementations assume that the flag is set to 1, meaning that all faults are required to be precise.

Integer-Overflow Flag and Mask

The integer-overflow flag (bit 8) and the integer-overflow mask (bit 12) are used in conjunction with the arithmetic integer-overflow fault. The mask bit masks the integer-overflow fault. When the fault is 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. The integer-overflow flag is a sticky flag. (Refer to the discussion of the arithmetic integer-overflow fault in Chapter 6 for more information about the integer-overflow mask and flag.)

Floating-Point Flags and Masks (80960SB Only)

The floating-point flags (bits 16 through 20) and masks (bits 24 through 28) perform the same functions as the integer-overflow flag and mask, except they are used for operations on real (floating point) numbers. When a mask is set, its associated floating-point fault is masked. When a mask is clear, the processor sets the flag for the associated fault whenever the fault condition occurs, but does not generate a fault. All the floating-point flags are sticky bits. Refer to the section in Chapter 10 titled "Exceptions and Fault Handling" for a detailed discussion of the floating-point faults and their associated flags and masks in the arithmetic controls.

Floating-Point-Normalizing-Mode Flag (80960SB Only)

The floating-point-normalizing-mode flag (bit 29) determines whether or not floating-point instructions are allowed to operate on denormalized numbers. If set, floating-point instructions are allowed to operate on denormalized numbers; if clear, the processor generates a floating reserved-operand fault when it detects denormalized numbers that are used as operands for floating-point instructions. (Refer to the section in Chapter 10 titled "Normalizing Mode" for more information on the use of this flag.)

Floating-Point-Rounding Control (80960SB Only)

The floating-point-rounding-control field (bits 30-31) indicates which rounding mode is in effect for floating point computations. These bits are set as shown in Table 2-4, depending on the rounding mode to be selected.

Table 2-4: Encoding of Floating-Point-Rounding-Control Panel

Rounding Control	Rounding Mode
00	Round to nearest (even)
01	Round down (toward negative infinity)
10	Round up (toward positive infinity)
11	Truncate (round toward zero)

(Refer to the section in Chapter 10 titled "Rounding Control" for more information on the use of the floating-point-rounding-control field.)

All the unused bits in the arithmetic controls are reserved and must be set to 0.

PROCESS AND TRACE CONTROLS

The processor's process controls and trace controls are also cached on the processor chip. The process controls are a set of 32 bits that control or show the current execution state of the processor. The process controls are described in detail later in this chapter.

The trace controls are a set of 32 bits that control the tracing facilities of the processor. The trace controls are described in Chapter 3.

INSTRUCTION CACHING

The processor provides a 512-byte cache for instructions. When the processor fetches an instruction or group of instructions from memory, they are stored in this cache before being fed into the instruction-execution pipeline. The processor manages this cache transparently from the program being run.

This instruction cache is a read-only cache, meaning that once bytes from the instruction stream are written into the instruction cache, they cannot be changed. Because of this, the processor does not support self-modified programs in a transparent fashion. The only way to change the instruction stream once it has been written into the instruction cache is to purge the instruction cache. The IAC message "Purge Instruction Cache" is provided for this purpose, as described in Chapter 11.

NOTE

The purge instruction cache IAC is not defined in the i960 architecture. It is an implementation-dependent feature of the 80960SA/SB processor.

SYSTEM ARCHITECTURE

The following sections concentrate on hardware considerations of the 80960SA/SB architecture, focusing on system configurations from a general perspective to explain overall design concepts. Subsequent chapters describe the details of hardware system design.

OVERVIEW OF THE 80960SA/SB PROCESSOR SYSTEM ARCHITECTURE

The central processing module, memory module, and I/O module form the natural boundaries for the hardware system architecture. A high bandwidth bus connects the modules together.

Figure 2-5 shows a simplified block diagram of a typical system configuration. The heart of this system is the 80960SA/SB processor, which fetches program instructions, executes code, manipulates stored information, and interacts with I/O devices. The high bandwidth bus connects the 80960SA/SB processor to memory and I/O modules. The 80960SA/SB processor stores system data, instructions, and programs in the memory module. The processor accesses various peripheral devices in the I/O module to directly support communication with other auxiliary subsystems.

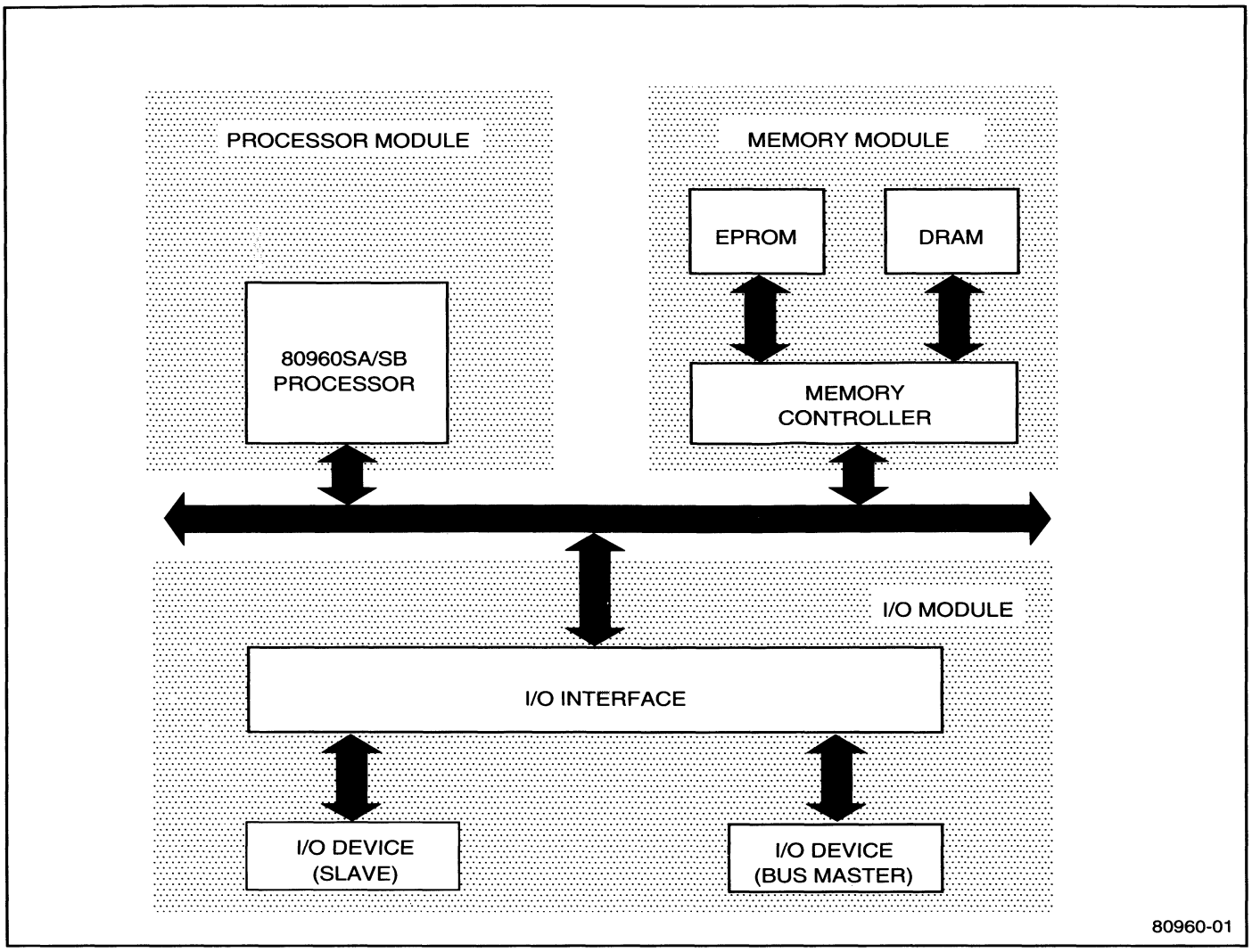


Figure 2-5: Basic 80960 System Configuration

80960SA/SB PROCESSOR AND THE BUS

The 80960SA/SB processor performs bus operations using multiplexed address and data signals and provides all the necessary control signals. The processor provides Intel standard control signals, such as Address Latch Enable (ALE), Address Status (AS), Write/Read command (W/R), Data Transmit/Receive (DT/R), and Data ENable (DEN). The 80960SA/SB processor also generates byte enable signals to specify which bytes on the 32-bit data lines are valid during the transfer.

The bus supports burst transactions, which access up to eight 16-bit data words at a maximum rate of one word per clock cycle. The processor performs burst transactions to load the on-chip 512-byte instruction cache to minimize memory accesses for instruction fetches. The processor also uses burst transactions for data access when more than a single 16-bit value is read/written.

To transfer control of the bus to an external bus master, the 80960SA/SB processor provides two arbitration signals: hold request (HOLD) and hold acknowledge (HLDA). After receiving HOLD, the processor asserts HLDA to grant control of the bus to an external bus master.

The 80960SA/SB processor uses an on-chip interrupt controller or an external interrupt controller (or both) to provide a flexible interrupt structure. An internal interrupt vector register specifies the type of interrupt structure.

MEMORY MODULE

A memory module can consist of a memory controller, Erasable Programmable Read-Only Memory (EPROM), Dynamic Random Access Memory (DRAM), or any other type of memory desired. The memory controller first conditions the bus signals for memory operation. It then demultiplexes the address and data lines, generates the chip select signals from the address, detects the start of the cycle for burst mode operation, and latches A_4-A_{15}.

A memory controller generates the control signals for ROM or RAM. In particular, it provides the control signals, the multiplexed row/column address signals, and the refresh control for dynamic RAMs. The controller may use the static column mode or page mode features of the dynamic RAM to accommodate the burst transaction of the 80960SA/SB processor. In addition to supplying the operation signals, the controller generates the READY signal to indicate that data can transfer to or from the 80960SA/SB processor.

The 80960SA/SB processor directly addresses up to 4-Gigabytes of physical memory. to ease the design of the controller, the processor does not allow burst accesses to cross 16-byte boundaries. Each address specifies a two-byte data word within the block. The processor uses two byte enable signals to access individual data bytes.

I/O MODULE

The I/O module consists of the I/O components and an interface circuit. I/O components allow the 80960SA/SB processor to connect to real world sensors, standard peripheral devices, or output devices.

The interface circuit performs several functions. It demultiplexes the address and data lines, generates the chip select signals from the address, produces the I/O read or I/O write command from the processor's W/R signal, and generates the READY signal. Because these functions are similar to functions of the memory controller, both interfaces can use similar or identical logic.

The 80960SA/SB processor uses memory-mapped addresses to access I/O devices, allowing the CPU to use the same instructions to exchange information for both memory and peripheral devices. This allows the powerful memory instructions to perform 8- and 16-bit data transfers.

Chapter 13 describes the I/O interface.