Interrupts 5

CHAPTER 5
INTERRUPTS

This chapter describes the 80960SA/SB processor's interrupt handling facilities. It also describes how interrupts are signaled.

OVERVIEW OF THE INTERRUPT FACILITIES

An interrupt is a temporary break in the control stream of a program so that the processor can handle another chore. Interrupts are generally requested from an external source. The interrupt request either contains a vector number or else points to a vector that tells the processor what chore to do while in the interrupted state. When the processor has finished servicing the interrupt, it generally returns to the program that it was working on when the interrupt occurred and resumes execution where it left off.

The processor provides a mechanism for servicing interrupts, which uses an implicit procedure call to a selected interrupt-handling procedure, called an *interrupt handler.*

When an interrupt occurs, the current state of the program is saved. If the interrupt occurs during an instruction that requires many machine cycles, the instruction state is also saved and execution of the instruction is suspended. See Chapter 3, "Instruction Suspension", for details.

The processor then creates a new frame on the interrupt stack and executes an implicit call to the interrupt handler selected with the interrupt vector.

Upon returning from the interrupt handler, the processor switches back to the program that was running when the interrupt occurred, restores it to the state it was in when the interrupt occurred, and resumes work on it.

Another feature of this interrupt handling mechanism is that it allows interrupts to be prioritized. If an interrupt is signaled that has the same or a lower priority than the processor's current priority, the processor stores the interrupt and services it at a later time. Interrupts that are waiting to be serviced are called *pending interrupts.*

SOFTWARE REQUIREMENTS FOR INTERRUPT HANDLING

To use the processor's interrupt handling facilities, software must provide the following items in memory:

- Interrupt Table
- Interrupt Handler Routines
- Interrupt Stack

These items are generally established in memory as part of the initialization procedure. Once these items are present in memory and pointers to them have been entered in the appropriate system data structures, the processor then handles interrupts automatically and independently from software.

The requirements for these items are given in following sections of this chapter.

VECTORS AND PRIORITY

Each interrupt vector is 8 bits in length, which allows up to 256 unique vectors to be defined. In practice, vectors 0 through 7 cannot be used, and vectors 244 through 251 are reserved and should not be used by software.

Each vector has a predefined priority, which is defined by the following expression:

$$\text{priority} = \text{vector}/8$$

Thus, at each priority level, there are 8 possible vectors (e.g., vectors 8 through 15 have a priority of 1, vectors 16 through 23 a priority of 2, and so on to vectors 246 through 255, which have a priority of 31).

The processor uses the priority of an interrupt to determine whether or not to service the interrupt immediately or to delay service. If the interrupt priority is greater than the processor's current priority, the processor services the interrupt immediately; if the interrupt priority is equal to or less than the processor's current priority, the processor saves the interrupt vector as a pending interrupt so that it can be serviced at a later time.

A priority-31 interrupt is always serviced immediately.

Note that 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 contains instruction pointers (addresses in the address space) to interrupt handlers. It must be aligned on a word boundary. The processor determines the location of the interrupt table by means of a pointer in the IMI.

As shown in Figure 5-1, the interrupt table contains one entry (i.e., one pointer) for each allowable vector. The structure of an interrupt-table entry is given at the bottom of Figure 5-1. Each interrupt procedure must begin on a word boundary, so the two least-significant bits of the entry are set to 0.

The first 36 bytes of the interrupt table are used to record pending interrupts. This section of the table is divided into two fields: pending priorities (byte-offset 0 through 3) and pending interrupts (byte-offset 4 through 35).

The pending priorities field contains a 32-bit string in which each bit represents an interrupt priority. The bit number in the string represents the priority number. 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.

The pending interrupts field contains a 256-bit string in which each bit represents an interrupt vector. For example, byte-offset 4 is reserved, byte-offset 5 is for vectors 8 through 15, byte-offset 6 is for vectors 16 through 23, and so on. When a pending interrupt is logged, 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. Software should set these fields to 0 at initialization and not access these fields after that.

NOTE

Refer to the section later in this chapter titled "Handling Pending Interrupts" for a description of the processor's pending interrupt mechanism.

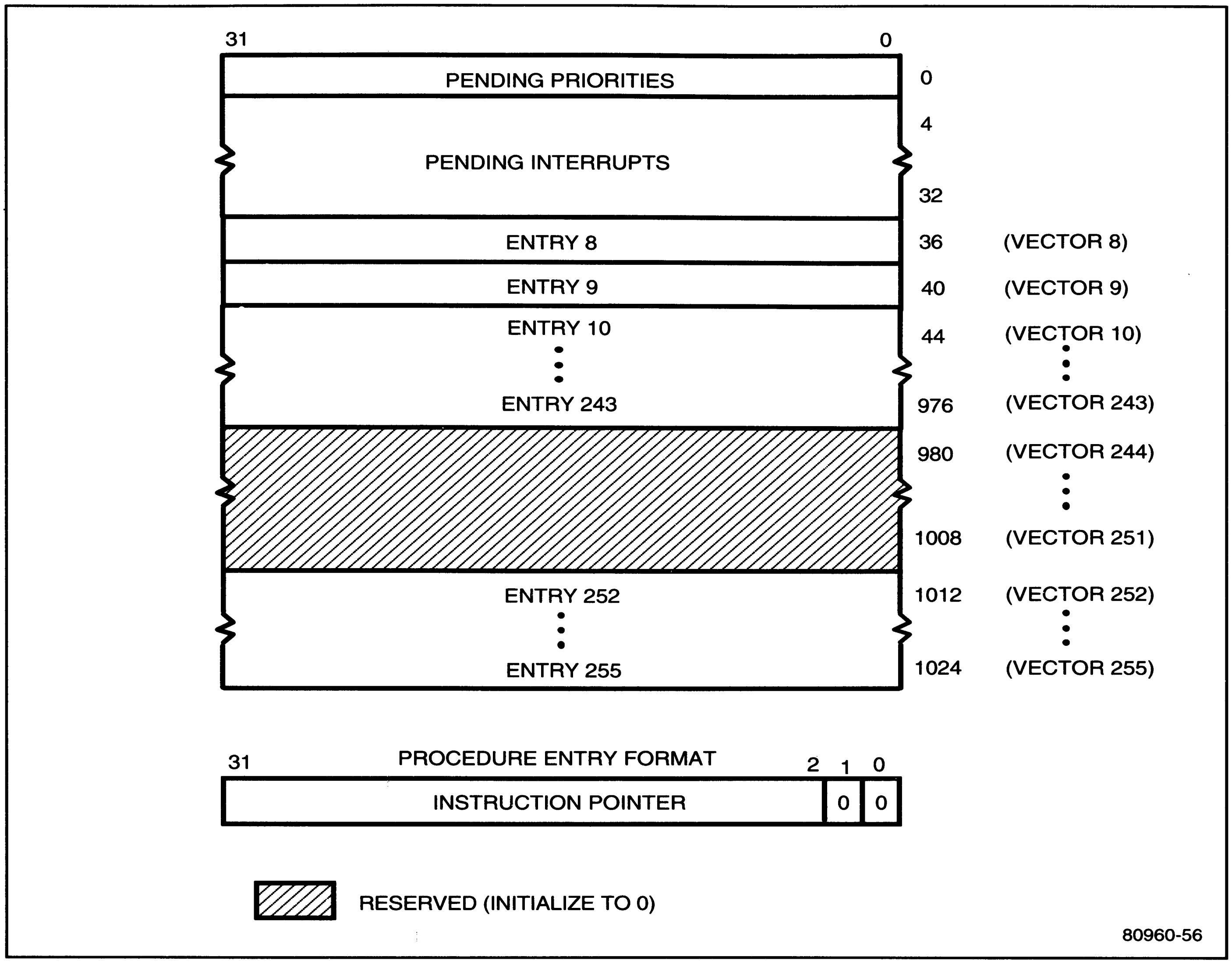

Figure 5-1: Interrupt Table

INTERRUPT HANDLER PROCEDURES

An interrupt handler is a procedure that performs a specific action that has been associated with a particular interrupt vector. For example, a typical job for an interrupt handler is to read a character from a keyboard.

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

The processor execution mode is always switched to supervisor while an interrupt is being handled.

When an interrupt-handler procedure is called, the states of the process controls and arithmetic controls for the interrupted program are saved. However, the interrupt handler shares the other resources of the interrupted program, in particular the global registers and the address space. This sharing of resources imposes one important restriction on the interrupt handler procedures.

The interrupt handler procedures must preserve and restore the state of any of the resources that it uses. For example, the processor allocates a set of local registers to the interrupt handler, just as it does on a local procedure call. If the interrupt handler needs to use the global or floating-point registers, however, it should save their contents before using them and restore them before returning from the interrupt handler.

INTERRUPT STACK

The interrupt stack can be located anywhere in the address space. The processor determines the location of the interrupt stack by means of a pointer in the IMI.

The interrupt stack has the same structure as the local procedure stack described in Chapter 4 in the section titled "Local Registers and the Procedure Stack."

INTERRUPT HANDLING ACTIONS

When the processor receives an interrupt, it handles it automatically. The processor takes care of saving the processor state, calling the interrupt-handler routine, and restoring the processor state once the interrupt has been serviced. Software support is not required.

The following section describes the actions the processor takes while handling interrupts. It is not necessary to read this section to use the interrupt mechanism or write an interrupt handler routine. This discussion is provided for those readers who wish to know the details of the interrupt handling mechanism.

Receiving an Interrupt

Whenever the processor receives an interrupt signal, it performs the following action:

1. It temporarily stops work on its current task, whether it is working on a program or another interrupt procedure.
2. It reads the interrupt vector.
3. It compares the priority of the vector with the processor's current priority.
4. If the interrupt priority is higher than that of the processor, the processor services the interrupt immediately as described in the next sections.
5. If the interrupt priority is equal to or less than that of the processor, the processor records the new interrupt in the pending interrupt record and continues work on its current task.

Servicing an Interrupt

The method that the processor uses to service an interrupt depends on the state the processor is in when it receives the interrupt. The following sections describe the interrupt handling actions for the executing and interrupted states of the processor. (The processor cannot respond to an interrupt while it is in the stopped state.)

In the two situations described in following sections, it is assumed that the interrupt priority is higher than that of the processor and will thus be serviced immediately after the processor receives it. The handling of lower priority interrupts is described later in this chapter in the section titled "Pending Interrupts."

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 to service the interrupt; this procedure is the same regardless of whether the processor is in the user or the supervisor mode when the interrupt occurs:

1. The processor switches to the interrupt stack (as shown in Figure 5-2). The interrupt stack pointer becomes the new stack pointer (NSP) for the processor.
2. The processor saves the current state of process controls and arithmetic controls in an interrupt record on the interrupt stack. (The interrupt record is described later in this chapter in the section titled "Interrupt Record".)
3. If the execution of an instruction was suspended, the processor includes a resumption record for the instruction in the interrupt record and sets the resume flag in the saved process controls. (Refer to the section in Chapter 4 titled "Instruction Suspension" for a discussion of the criteria for suspending instructions.)
4. The processor allocates a new frame on the interrupt stack and loads the new frame pointer (NFP) in global register g15.

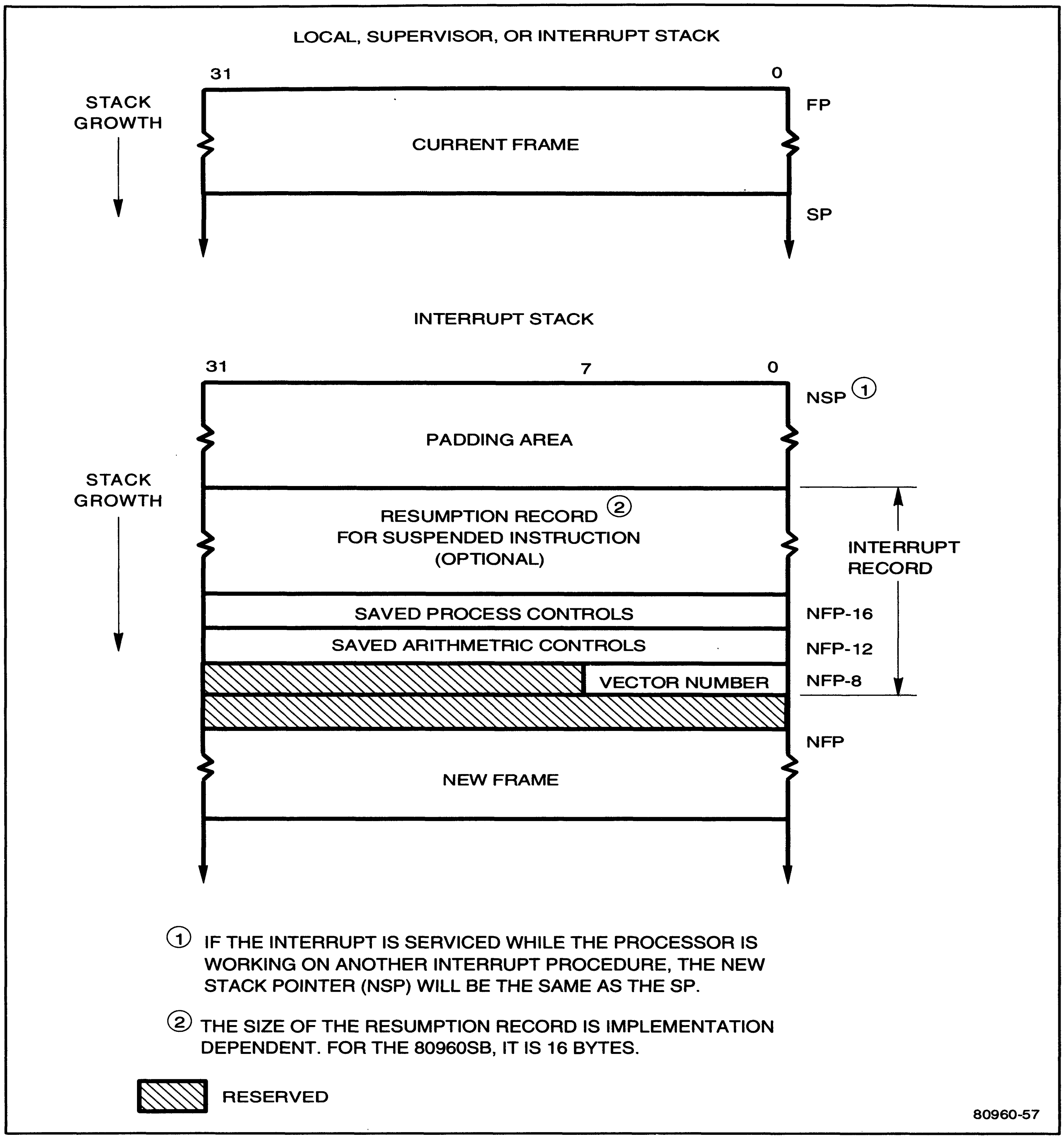
LOCAL, SUPERVISOR, OR INTERRUPT STACK
31
0
STACK GROWTH
CURRENT FRAME
FP
SP
INTERRUPT STACK
31
7
0
NSP ①
PADDING AREA
RESUMPTION RECORD ②
FOR SUSPENDED INSTRUCTION
(OPTIONAL)
INTERRUPT RECORD
SAVED PROCESS CONTROLS
NFP-16
SAVED ARITHMETIC CONTROLS
NFP-12
VECTOR NUMBER
NFP-8
NFP
NEW FRAME
① IF THE INTERRUPT IS SERVICED WHILE THE PROCESSOR IS WORKING ON ANOTHER INTERRUPT PROCEDURE, THE NEW STACK POINTER (NSP) WILL BE THE SAME AS THE SP.
② THE SIZE OF THE RESUMPTION RECORD IS IMPLEMENTATION DEPENDENT. FOR THE 80960SB, IT IS 16 BYTES.
▨ RESERVED
80960-57

Figure 5-2: Storage of an Interrupt Record on the Stack

5. The processor switches to the interrupted state.
6. The processor sets the state flag in its internal process controls to interrupted, its execution mode to supervisor, and its priority to the priority of the interrupt. Setting the processor's priority to that of the interrupt insures that lower priority interrupts can not interrupt the servicing of the current interrupt.
7. Also in its internal process controls, the processor clears the trace-fault-pending and trace-enable flags. Clearing these flags allows the interrupt to be handled without trace faults being raised.
8. The processor sets the frame return status field (associated with the PFP in register r0) to 111_2.
9. The processor performs an implicit call-extended operation (similar to that performed for the **callx** instruction). The address for the procedure that is called is that which is specified in the interrupt table for the specified interrupt vector.

Once the processor has completed the interrupt procedure, it performs the following action on the return:

1. The processor copies the arithmetic controls field from the interrupt record into its arithmetic controls register.
2. The processor copies the process controls field from the interrupt record into its internal process controls.
3. If the resume flag of the process controls is set, the processor copies the resumption record from the interrupt record to the scratch space field of the PRCB.
4. The processor deallocates the current stack frame and interrupt record from the interrupt stack and switches to the local stack or the supervisor stack (whichever one it was using when it was interrupted).
5. The processor checks the interrupt table for pending interrupts that are higher then the priority of the program being returned to. If a higher-priority pending interrupt is found, it is handled as if the interrupt occurred at this point.
6. Assuming that there are no pending interrupts to be serviced, the processor switches to the executing state and resumes work on the program.

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 is described at the beginning of this 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.

Interrupt Record

The processor saves the state of an interrupted program (or interrupt-handler) routine in an interrupt record. Figure 5-2 shows the structure of this interrupt record.

The interrupt record is always stored on the interrupt stack adjacent to the new frame that is created for use by the interrupt handler procedure. The resumption record within the interrupt record is used to save the state of a suspended instruction. If no instruction is suspended, the resumption record is not created.

Pending Interrupts

As is described earlier in this chapter, the processor provides a mechanism for evaluating interrupts according to their priority. If the interrupt priority is equal to or lower than the processor's current priority, the processor does not service the interrupt immediately. Instead, it posts the interrupt in the pending interrupt section of the interrupt table. The processor checks the interrupt table at specific times and services those interrupts that have a higher priority than its current priority. This pending interrupt mechanism provides two benefits:

1. The ability to delay the servicing of low priority interrupts (by posting them in the pending interrupt section of the interrupt table) allows the processor to concentrate its processing activity on higher priority tasks.
2. In a system that uses two or more 80960SA/SB processors, both processors can share the same interrupt table. This interrupt-table sharing allows the processors to share the interrupt handling load.

The following paragraphs describe how the processor handles pending interrupts.

NOTE

The i960 architecture defines the section of the interrupt table for storing pending interrupts and a mechanism for checking the interrupt table for pending interrupts. The method used for posting interrupts to the interrupt table and circumstances under which the processor checks the interrupt table for pending interrupts is not defined.

In the following description of the pending interrupt mechanism, the information given in the sections titled "Posting Pending Interrupts" and "Checking for Pending Interrupts" is specific to the 80960SA/SB processor. The information given in the section titled "Handling Pending Interrupts" is defined in the i960 architecture and should be common in all processors that implement this part of the architecture.

Posting Pending Interrupts

An interrupt can be posted in the pending-interrupt record of the interrupt table in either of the following two ways:

1. The processor receives an interrupt with a priority equal to or lower than that of the program the processor is currently working on. The processor then automatically posts the interrupt in the pending-interrupt record.
2. The kernel can set the desired pending-interrupt and pending-priority bits in the interrupt table.

Using the first method, the processor performs an atomic read/write operation that locks the interrupt table until the posting operation has been completed. Locking the interrupt table prevents other agents on the bus from accessing the interrupt table during this time.

The second method of posting an interrupt is risky, because it does not use this locking technique. (The processor's atomic instructions are not able to perform a locking operation that spans several instructions.) This method will work only if the kernel can insure the following:

- that no external I/O agent will attempt to post a pending interrupt simultaneously with the processor, and
- that an interrupt cannot occur after one bit (e.g., the pending priority bit) of the pending-interrupt record is set but before the other bit (the pending interrupt vector) is set.

Checking for Pending Interrupts

The processor automatically checks the interrupt table for pending interrupts at the following times:

- After returning from an interrupt-handler procedure
- While executing a modify-process-controls instruction (**modpc**), if the instruction causes the program's priority to be lowered.
- After receiving a test pending interrupts IAC message.

Handling Pending Interrupts

The processor uses the same type of atomic read/write operation to check the interrupt table for pending interrupts as it does for posting pending interrupts. Again, this technique prevents other agents on the bus from accessing the interrupt table until the pending-interrupt check has been completed.

When the processor finds a pending interrupt, it handles it as if it had just received the interrupt. The handling mechanism is the same as is described earlier in this chapter for interrupts that are serviced as soon as they are received.

If the processor finds two pending interrupts at the same priority, it services the interrupt with the highest vector number first.

SIGNALING INTERRUPTS

NOTE

The i960 architecture does not define a mechanism for signaling interrupts to the processor. The methods of signaling interrupts described in the following section are specific to the 80960SA/SB processor.

The 80960SA/SB processor can be interrupted in any of the following four ways:

- Signal on its interrupt pins
- Signal on its interrupt pins from an external interrupt controller
- An IAC message from a program
- A pending interrupt (described earlier in this chapter)

Interrupts From Interrupt Pins

The processor has four interrupt pins, called $\overline{\text{INT0}}$, INT1, INT2/INTR, and $\overline{\text{INT3/INTA}}$. These pins can be configured in either of the following ways:

- as four interrupt-signal inputs
- as two interrupt inputs and two pins for handshaking with an interrupt controller such as the Intel 8259A Programmable Interrupt Controller

A 32-bit, interrupt-control register in the processor determines how these pins are used. Each interrupt pin is associated with one 8-bit field in the register, as shown in Figure 5-3.

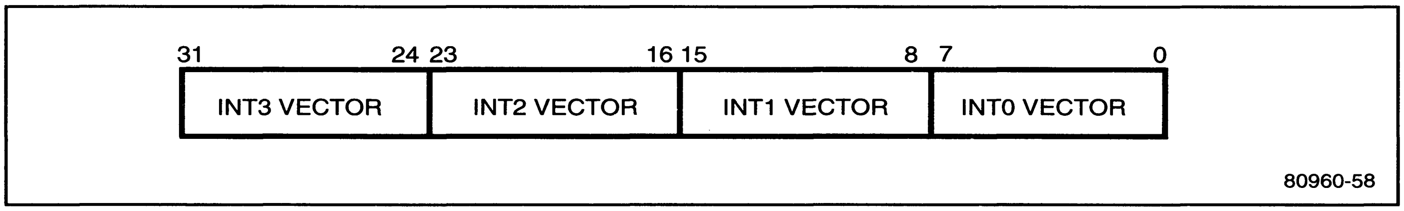

Figure 5-3: Interrupt-Control Register

If the interrupt pins are to be used as four inputs, a different interrupt vector is stored in each of the four fields in the interrupt-control register. Then, when an interrupt is signaled on one of the pins, the processor reads the vector from the pin's associated field in the register. For example, if an interrupt is signaled on pin INT_0, the processor reads the vector from bits 0 through 7.

Asserting any of the four interrupt input signals ($\overline{INT_0}$, INT_1, INT_2, $\overline{INT_3}$), will interrupt the 80960SA/SB processor. If the signals are simultaneously asserted, the 80960SA/SB assumes that INT_0 has the highest priority, followed by INT_1, INT_2, and $\overline{INT_3}$. Software should follow this convention when programming the Interrupt Control Register. When the interrupt input signals are asserted, the 80960SA/SB processor utilizes a vector number that the interrupt control register specifies as an index to an entry in the interrupt table located in memory.

If the INT_2 vector field is set to 0, the functions of the INT_2 and $\overline{INT_3}$ pins are changed to INTR and $\overline{INTA}$, respectively. Here, the INTR pin is used to receive signals from an interrupt controller and the INTA pin is used to send acknowledge signals back to the controller. When the processor receives a signal on the INTR pin, it reads an interrupt vector from the least-significant 8 bits of its bus, then sends an acknowledge signal to the controller through $\overline{INTA}$. When the INT_2 and $\overline{INT_3}$ pins are configured in this manner, the processor ignores the INT_3 vector field.

The interrupt-control register is memory mapped to addresses $FF000004_{16}$ through $FF000007_{16}$. Only the processor can read or write this register using the synchronous load (**synld**) and synchronous move (**synmov**) instructions. External agents on the bus cannot access this register.

The value in the interrupt-control register after the processor is initialized is $FF000000_{16}$. This setting specifies the four interrupt pins as $\overline{INTA}$, INTR, INT, and $\overline{INT_0}$.

IAC Interrupts

The processor can also receive an interrupt request by means of the IAC mechanism. (The IAC mechanism is described in detail in Chapter 11.) The interrupt IAC message can be sent to the processor internally as part of the currently running program. The interrupt vector is contained in the interrupt IAC message.

A program running on the processor can signal an interrupt through an internal interrupt-IAC message. An internal IAC is sent to the processor by means of a synchronous move instruction. When the processor executes a synchronous move to its IAC message space, it signals an IAC message internally.

Interrupt Latency

The 80960SA/SB interrupt controller manages the interrupt mechanism automatically and therefore deals with many cases. Depending on the situation, latency may vary. The interrupt latencies consist of a base latency and special case latencies added to it. These special cases consist of such things as using an 8259A interrupt controller, the local cache being full, or an interrupt occurring while the processor is already in the interrupted state.

The base interrupt latency is 90 cycles as shown in Table 5-1. Table 5-2 describes the breakdown of the base interrupt latency. Notice that it only takes six cycles for the 80960SA/SB to respond to the interrupt; four cycles for hardware recognition of the interrupt, and a minimum of one cycle to respond if the interrupt occurs on an instruction boundary. The table indicates that it takes two cycles for interrupt signals at the beginning of a RISC instruction. This value differs depending on the instruction being interrupted and the point at which the interrupt occurs in the instruction. Table 5-3 gives values for integer execution, floating point, and transcendental floating-point instruction interrupt boundaries.

Table 5-1: Interrupt Latencies

Type of Latency	Cycles
Base Interrupt Latency	90
Return	80
Interrupt immediately followed by another interrupt. Second interrupt posted to interrupt table.	157
Return with a Pending Interrupt Posted	157
Pending Interrupt	0

Table 5-2: Constituent Parts of the Base Latency

Constituent Latencies	**Cycle**
Hardware Recognition	4
Stop Current Instruction Flow Assuming a RISC Instruction	2
Determine Next IP and Save	8
Read Interrupt Vector Number	18
Check Interrupt Priority	8
Read Interrupt Table Vector	15
Check if Processor Already Interrupted	6
Save Process Control and Write Interrupt Record	14
Compute Interrupt Record Address of New Local Register Set	10
Allocate New Local Register Set	3
Fetch New Instruction and Start Decoding	2

Other situations that add to the latency are interrupts which signal the start of a multicycle instruction or multiple interrupts which signal at the same time. The first may cause a resumption record to store on the stack. This records all the necessary information the 80960SB needs to resume executing the interrupted instruction. Not all interruptible instructions cause a resumption record to be created. If an instruction executes for over 100 cycles, then the processor creates a resumption record. Any cycle less that that will simply restart the instruction upon return from interrupt. This provides an engineering trade-off between saving states after less than 100 cycles and restarting the instruction. Restarting the instruction requires fewer cycles (in most cases).

Table 5-3: Special Case Latencies

Special Case Latencies	Cycles
8259A Interrupt Expansion	10 MHz = 14 16 MHz = 16
Frame Cache Full	40
Current Process in "Interrupt"	14
RISC Instructions (Worst Case)	3-4
Integer Execution	10-40
Floating Point	12-96
Transcendental Floating Point	90
Instruction Cache Miss (2 Wait State)	7

Multiple interrupts signalled at various times are handled on a first come, first serve basis. Interrupts occurring at the same time are handled on a priority scheme with INT_3, INT_2, INT_1, and INT_0. The first interrupt is handled as soon as the 80960SA/SB reaches an interruptible state (e.g., at an end of instruction) and subsequent interrupts are read from the interrupt control register and posted in the interrupt table as soon as the microcode routine re-enables interrupts. While interrupts are enabled, the event (another interrupt) stores in the 4-bit register as described earlier. Posting a pending interrupt to the interrupt table adds about 60 cycles to the interrupt latency. This consists of comparing the priorities of the processor and interrupt, and writing a *one* to the appropriate bits in the pending interrupt field in the interrupt table. The index vector from the interrupt control register or an 8259A vector points to the positions in the fields.

The minimum interrupt latency is 90 clocks (or 5.63 microseconds at 16 Mhz). This latency assumes the instruction handler is in the cache. If there is an instruction cache miss, seven clocks for caching the instructions must be added to the base latency (assuming a two wait state memory system). In most cases, the instruction will be cached already. A program's typical latency would add about 3 more clocks for non-RISC instructions. A local register cache miss adds 40 cycles (or 2.50 microseconds) to the interrupt latency. The worst case latency would be 207 cycles (or 12.94 microseconds). This assumes the interrupt signals at the beginning of an ediv instruction (40 cycles), a local cache register miss (40 cycles) occurs, the current process is in the *interrupt* state (14 cycles), and an 8259A controller is used with 4 wait states (18 cycles).

It is important to note that during the microcode routine, all of the stack manipulations, saving state, checking priorities, and allocating new registers is accomplished automatically. When the 80960SA/SB enters the user interrupt handler, this routine does not need to do any housekeeping chores, and it can begin immediately with useful code. The benefit is that this work is handled in microcode quickly and efficiently. Also note that the 80960SA/SB responds to an interrupt in as little as 6 clocks. This is from the point of interrupt pin assertion to the point when the instruction flow stops and the microcode routine handles the housekeeping tasks. Normally, processors do not include any of the housekeeping activities in the interrupt latency, so care should be taken when comparing latencies. Table 5-3 lists the latencies based on special cases that occur. These values must be added to the base latency from Table 5-1.