Chapter 4 Interrupts, Traps and Exceptions
(Revision number 16)

In the previous chapter, we discussed the implementation of the processor. In this chapter, we will discuss how the processor can handle discontinuities in program execution. In a sense, branch instructions and subroutine calls are discontinuities as well. However, the programmer consciously introduced such discontinuities that are part of the program. The discontinuities we will look into in this chapter are those that are potentially unplanned for and often may not even be part of the program that experiences it.

Let us look at a simple analogy, a *classroom*. The professor is giving a lecture on computer architecture. To encourage class participation, he wants students to ask questions. He could do one of two things: (1) periodically, he could stop lecturing and poll the students to see if they have any questions; or (2) he could tell the students whenever they have a question, they should put up their hands to signal that they have a question. Clearly, the first approach takes away the spontaneity of the students. By the time the professor gets around to polling the students, they may have forgotten that they had a question. Worse yet, they had so many questions one after another that they are now completely lost! The second approach will ensure that the students remain engaged and they do not lose their train of thought. However, there is a slight problem. When should the professor take a student's question? He could take it immediately as soon as someone puts up a hand, but he may be in mid-sentence. Therefore, he should finish his train of thought and then take the question. He has to be careful to remember where he was in his lecture so that after answering the student's question he can return to where he left off in the lecture. What if another student asks a question while the professor is in the middle of answering the first student's question? That will soon spiral out of control, so as a first order principle the professor should not take another question until he is finished with answering the first one. So two things to take away from the classroom analogy: remember where to return in the lecture, and disable further questions.

The processor that we designed executes instructions but unless it can talk to the outside world for I/O it is pretty much useless. Building on the above analogy, we can have the processor poll an input device (such as a keyboard) periodically. On the one hand, polling is error prone since the device could be generating data faster than the polling rate. On the other hand, it is extremely wasteful for the processor to be doing the polling if the device has no data to deliver. Therefore, we can apply the classroom analogy and let the device *interrupt* the processor to let it know that it has something to say to the processor. As in the classroom analogy, the processor should remember where it is in its current program execution and disable further interruptions until it services the current one.

4.1 Discontinuities in program execution

In Chapter 3, we defined the terms synchronous and asynchronous in the context of logic circuit behavior. Consider a real life example. Let's say you walk to the fridge and pick up a soda. That's a *synchronous* event. It is part of your intended activities. On the other hand, while you are working on your homework in your room, your roommate comes in and gives you a soda. That's an *asynchronous* event, since it was not part of your intended activities. Making a phone call is a synchronous event; receiving one is an asynchronous event.

We can generalize the definition of synchronous and asynchronous events observed in a system, be it in hardware or software. A *synchronous* event is one, which occurs (if it does at all) at well-defined points of time aligned with the intended activities of the system. The state changes from one microstate to the next in the sequences we discussed in Chapter 3 are all examples of synchronous events in a hardware system. Similarly, opening a file in your program is a synchronous software event.

An *asynchronous* event is one, which occurs (if it does at all) unexpectedly with respect to other ongoing activities in the system. As we will see shortly, interrupts are asynchronous hardware events. An e-mail arrival notification while you are in the middle of your assignment is an asynchronous software event.

A system may compose synchronous and asynchronous events. For example, the hardware may use polling (which is synchronous) to detect an event and then generate an asynchronous software event.

Now we are ready to discuss discontinuities in program execution that come in three forms: *interrupts*, *exceptions*, and *traps*.

1. Interrupts

An *interrupt* is the mechanism by which devices catch the attention of the processor. This is an unplanned discontinuity for the currently executing program and is asynchronous with the processor execution. Furthermore, the device I/O may be intended for an altogether different program from the current one. For the purposes of clarity of discourse, we will consider only discontinuities caused by external devices as interrupts.

2. Exceptions

Programs may unintentionally perform certain illegal operations (for example *divide by zero*) or follow an execution path unintended in the program specification. In such cases, once again it becomes necessary to discontinue the original sequence of instruction execution of the program and deal with this kind of unplanned discontinuity, namely, *exception*. Exceptions are internally generated conditions and are synchronous with the processor execution. They are usually unintended by the current program and are the result of some erroneous condition encountered during execution. However, programming languages such as Java define an exception mechanism to allow error propagation through layers of software. In this case, the program *intentionally* generates an exception to signal some unexpected program behavior. In either case, we define exception to be some condition (intentional or unintentional) that deviates from normal program execution.

3. Traps

Programs often make *system calls* to read/write files or for other such services from the system. Such system calls are like procedure calls but they need some special handling since, the user program will be accessing parts of the system whose integrity affects a whole community of users and not just this program. Further, the user program may not know where in memory the procedure corresponding to this service exists and may have to discern that information at the point of call. Trap, as the name suggests, allows the program to *fall into* the operating system, which will then decide what the user program wants. Another term often used in computer literature for program generated traps is *software interrupts*. For the purposes of our discussion, we consider software interrupts to be the same as traps. Similar to exceptions, traps are internally generated conditions and are synchronous with the processor execution. Some traps could be intentional, for example, as manifestation of a program making a system call. Some traps could be unintentional as far as the program is concerned. We will see examples of such unintentional traps in later chapters when we discuss memory systems.

While the understanding of the term "interrupt" is fairly standardized, the same cannot be said about the other two terms in the literature. In this book, we have chosen to adopt a particular definition of these three terms, which we use consistently, but acknowledge that the definition and use of these terms may differ in other books. Specifically, in our definition, a "trap" is an internal condition that the currently running program has no way of dealing with on its own, and if anything, the system has to handle it. On the other hand, it is the currently running program's responsibility to handle an "exception".

Table 4.1 summarizes the characteristics of these three types of program discontinuities. The second column characterizes whether it is synchronous or asynchronous; the third column identifies the source of the discontinuity as internal or external to the currently running program; the fourth column identifies if the discontinuity was intentional irrespective of the source; and the last column gives examples of each type of discontinuity.

Type	Sync/Async	Source	Intentional?	Examples
Exception	Sync	Internal	Yes and No	Overflow, Divide by zero, Illegal memory address, Java exception mechanism
Trap	Sync	Internal	Yes and No	System call, Software interrupts, Page fault, Emulated instructions
Interrupt	Async	External	Yes	I/O device completion

Table 4.1: Program Discontinuities

4.2 Dealing with program discontinuities

As it turns out, program discontinuity is a powerful tool. First, it allows the computer system to provide input/output capabilities; second, it allows the computer system to perform resource management among competing activities; third, it allows the computer system to aid the programmer with developing correct programs. Interrupts, traps, and exceptions, serve the three functions, respectively.

Dealing with program discontinuities is a partnership between the processor architecture and the operating system. Let us understand the division of labor. Detecting program discontinuity is the processor's responsibility. Redirecting the processor to execute code to deal with the discontinuity is the operating system's responsibility. As we will see throughout this book, such a partnership exists between the hardware and the system software for handling each subsystem.

Let us now figure out what needs to happen to deal with program discontinuities, specifically the actions that need to happen implicitly in hardware, and those that need to happen explicitly in the operating system.

It turns out that most of what the processor has to do to deal with program discontinuity is the same regardless of the type. Recall that a processor is simply capable of executing instructions. To deal with any of these program discontinuities, the processor has to start

executing a different set of instructions than the ones it is currently executing. A *Handler* is the procedure that is executed when a discontinuity occurs. The code for the handler is very much like any other procedure that you may write. In this sense, such discontinuities are very much like a procedure call (see Figure 4.1). However, it is an unplanned procedure call. Moreover, the control may or may not return to the interrupted program (depending on the nature of the discontinuity). Yet, it is necessary to observe all the formalities (the procedure calling convention) for this unplanned procedure call and subsequent resumption of normal program execution. Most of what needs to happen is straightforward, similar to normal procedure call/return.

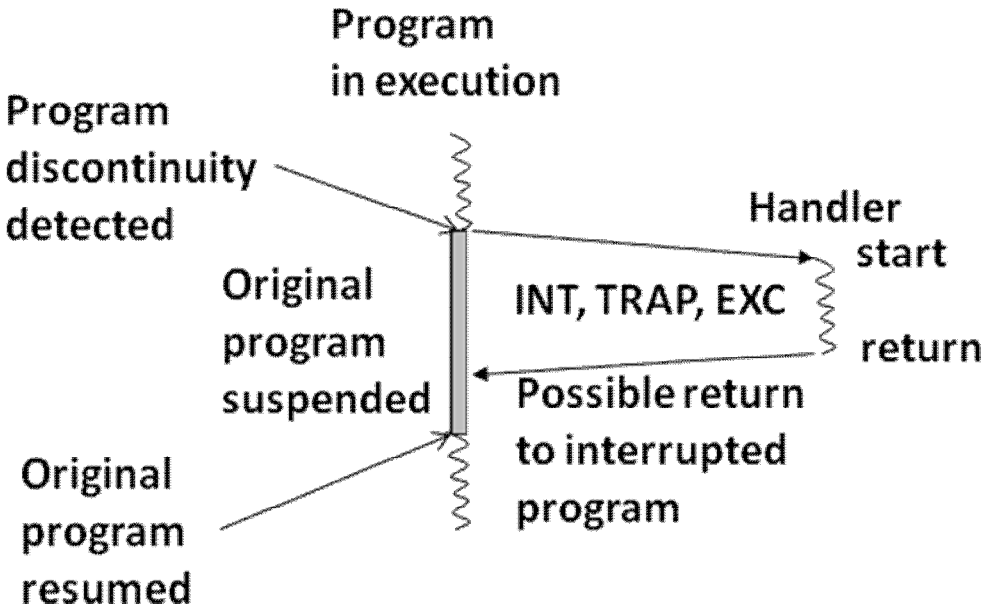

Figure 4.1: Program Discontinuity

Four things are tricky about these discontinuities.

1. They can happen anywhere during the instruction execution. That is the discontinuity can happen in the middle of an instruction execution.
2. The discontinuity is unplanned for and quite possibly completely unrelated to the current program in execution. Therefore, the hardware has to save the program counter value implicitly before the control goes to the handler.
3. At the point of detecting the program discontinuity, the hardware has to determine the address of the handler to transfer control from the currently executing program to the handler.
4. Since the hardware saved the PC implicitly, the handler has to discover how to resume normal program execution.

What makes it possible to address all of the above issues is the partnership between the operating system and the processor architecture. The basis for this partnership is a data structure maintained by the operating system somewhere in memory that is known to the processor. This data structure is a *fixed-size table* of handler addresses, one for each type

of anticipated program discontinuity (see Figure 4.2). The size of the table is architecture dependent. Historically, the name given to this data structure is *interrupt vector table (IVT)*[1]. Each discontinuity is given a *unique* number often referred to as a *vector*. This number serves as a unique index into the IVT. The operating system sets up this table at boot time. This is the explicit part of getting ready for dealing with program discontinuities. Once set up, the processor uses this table during normal program execution to look up a specific handler address upon detecting a discontinuity.

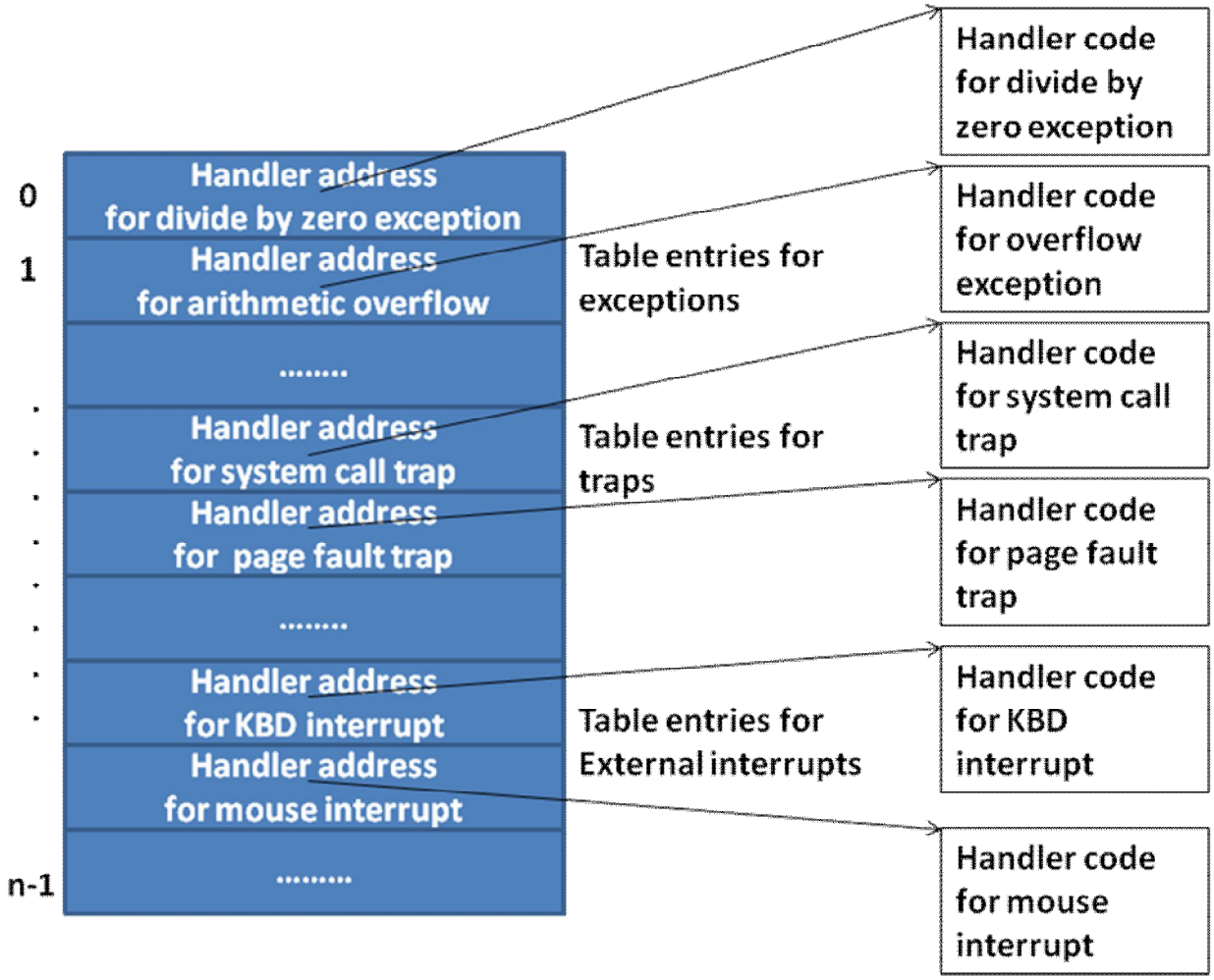

Figure 4.2: Interrupt Vector Table (IVT) – OS sets up this table at boot time

In the case of traps and exceptions, the hardware generates this vector internally. We introduce an *exception/trap register (ETR)*, internal to the processor, for storing this vector (Figure 4.3). Upon an exception or a trap, the unique number associated with that exception or trap will be placed in ETR. For example, upon detecting a "divide by zero" exception, the FSM for divide instruction will place the vector corresponding to this exception in ETR. Similarly, a system call may manifest as a "trap instruction"

[1] Different vendors give this data structure a different name. Intel calls this data structure Interrupt Descriptor Table (IDT), with 256 entries.

supported by the processor architecture. In this case, the FSM for the trap instruction will place the vector corresponding to the specific system call in ETR.

Figure 4.3: Exception/Trap Register – Number set by the processor upon detecting an exception/trap; used by the processor to index into the IVT to get the handler address

To summarize, the essence of the partnership between the operating system and the hardware for dealing with program discontinuities is as follows:

1. The architecture may itself define a set of exceptions and specify the numbers (vector values) associated with them. These are usually due to runtime errors encountered during program execution (such as arithmetic overflow and divide by zero).
2. The operating system may define its own set of exceptions (software interrupts) and traps (system calls) and specify the numbers (vector values) associated with them.
3. The operating system sets up the IVT at boot time with the addresses of the handlers for dealing with different kinds of exceptions, traps, and interrupts.
4. During the normal course of execution, the hardware detects exceptions/traps and stashes the corresponding vector values in ETR.
5. During normal course of execution, the hardware detects external interrupts and receives the vector value corresponding to the interrupting device.
6. The hardware uses the vector value as an index into the IVT to retrieve the handler address to transfer control from the currently executing program.

In the case of external interrupt, the processor has to do additional work to determine vector corresponding to the device that is interrupting to dispatch the appropriate device-handling program. Section 4.3 discusses the enhancements to the processor architecture and instruction-set to deal with program discontinuities. Section 4.4 deals with hardware design considerations for dealing with program discontinuities.

4.3 Architectural enhancements to handle program discontinuities

Let us first understand the architectural enhancements needed to take care of these program discontinuities. Since the processor mechanism is the same regardless of the type of discontinuity, we will henceforth simply refer to these discontinuities as interrupts.

1. When should the processor entertain an interrupt? This is analogous to the classroom example. We need to leave the processor in a clean state before going to the handler. Even if the interrupt happened in the middle of an instruction execution, the processor should wait until the instruction execution is complete before checking for an interrupt.

2. How does the processor know there is an interrupt? We can add a *hardware line* on the datapath bus of the processor. At the end of each instruction execution, the processor samples this line to see if there is an interrupt pending.
3. How do we save the return address? How do we manufacture the handler address? Every instruction execution FSM enters a special macro state, **INT**, at the end of instruction execution if there is an interrupt pending.
4. How do we handle multiple cascaded interrupts? We will discuss the answer to this question in Section 4.3.3.
5. How do we return from the interrupt? We will present ideas for this question in Section 4.3.4.

4.3.1 Modifications to FSM

In Chapter 3, the basic FSM for implementing a processor consisted of three macro states Fetch, Decode, and Execute, as shown below.

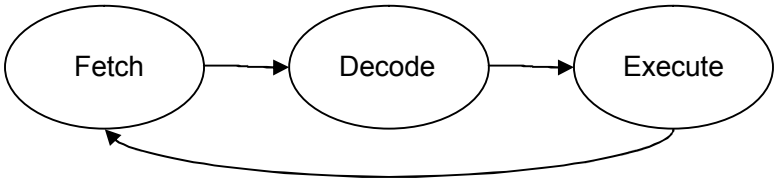

Figure 4.4-(a): Basic FSM of a processor

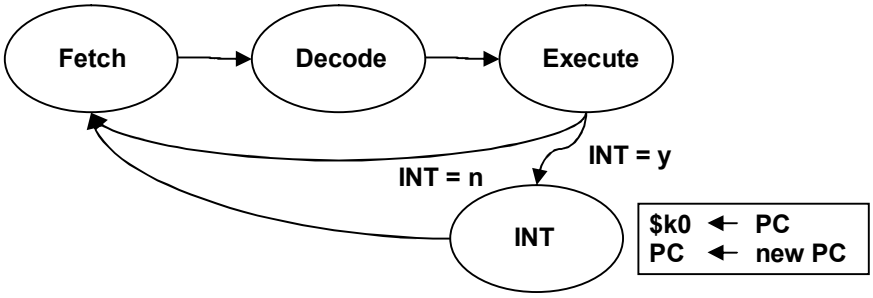

Figure 4.4-(b): Modified FSM for handling interrupts

Figure 4.4-(b) shows the modified FSM that includes a new macro state for handling interrupts. As we mentioned in Section 4.2, an interrupt may have been generated anytime during the execution of the current instruction. The FSM now checks at the point of completion of an instruction if there is a pending interrupt. If there is (INT=y), then the FSM transitions to the INT macro state; if there is no interrupt pending (INT=n) then next instruction execution resumes by returning to the Fetch macro state. A possibility is to check for interrupts after each macro state. This is analogous to the professor completing his/her thought before recognizing a student with a question in the classroom example. However, checking for interrupts after each macro state is problematic. In Chapter 3, we saw that the datapath of a processor includes several internal registers that are not visible at the level of the instruction set architecture of the

processor. We know that once an instruction execution is complete, the values in such internal registers are no longer relevant. Thus, deferring the check for interrupt until the completion of the current instruction leaves the processor in a clean state. To resume the interrupted program execution after servicing the interrupt, the two things that are needed are the state of the program visible registers and the point of program resumption.

Example 1:

Consider the following program:

100 ADD
101 NAND
102 LW
103 NAND
104 BEQ

An interrupt occurs when the processor is executing the ADD instruction. What is the PC value that needs to be preserved to resume this program after the interrupt?

Answer:

Even though the interrupt occurs during the execution of the ADD instruction, the interrupt will be taken only **after** the instruction execution is complete. Therefore, the PC value to be preserved to resume this program after the interrupt is **101**.

Let us now discuss what needs to happen in the INT macro state. To make the discussion concrete, we will make enhancements to the LC-2200 processor for handling interrupts.

1. We have to save the current PC value somewhere. We reserve one of the processor registers $k0 (general purpose register number 12 in the register file) for this purpose. The INT macro state will save PC into $k0.
2. We receive the PC value of the handler address from the device, load it into the PC and go to the FETCH macro state. We will elaborate on the details of accomplishing this step shortly.

4.3.2 A simple interrupt handler

Figure 4.5 shows a simple interrupt handler. The save/restore of processor registers is exactly similar to the procedure calling convention discussed in Chapter 2.

```
Handler:
        save processor registers;
        execute device code;
        restore processor registers;
        return to original program;
```

Figure 4.5: A Simple Interrupt Handler

Example 2:

Consider the following program:

100 ADD
101 NAND

102 LW
103 NAND
104 BEQ

An interrupt occurs when the processor is executing the ADD instruction. At this time, the only registers in use by the program are R2, R3, and R4. What registers are saved and restored by the interrupt handler?

Answer:
Unfortunately, since an interrupt can happen at any time, the interrupt handler has no way of knowing which registers are currently in use by the program. Therefore, it save and restores **ALL** the program visible registers though this program only needs R2, R3, and R4 to be saved and restored.

4.3.3 Handling cascaded interrupts

There is a problem with the simple handler code in Figure 4.5. If there is another interrupt while servicing the current one then we will lose the PC value of the original program (currently in $k0). This essentially would make it impossible to return to the original program that incurred the first interrupt. Figure 4.6 depicts this situation.

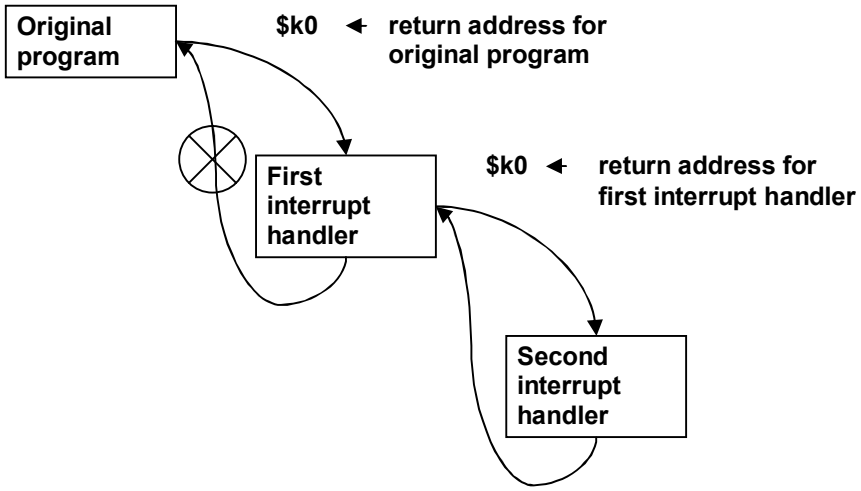

Figure 4.6: Cascaded interrupts

By the time we get to the second interrupt handler, we have lost the return address to the original program. This situation is analogous to the classroom example when the professor has to handle a second question before completing the answer to the first one. In that analogy, we simply took the approach of disallowing a second questioner before the answer to the first questioner was complete. May be the second question needs to be answered right away to help everyone understand the answer to the original question. Not being able to entertain multiple interrupts is just not a viable situation in a computer system. Devices are heterogeneous in their speed. For example, the data rate of a disk is much higher than that of a keyboard or a mouse. Therefore, we cannot always afford to turn off interrupts while servicing one. At the same time, there has to be a window

devoid of interrupts available to any handler wherein it can take the necessary actions to avoid the situation shown in Figure 4.6.

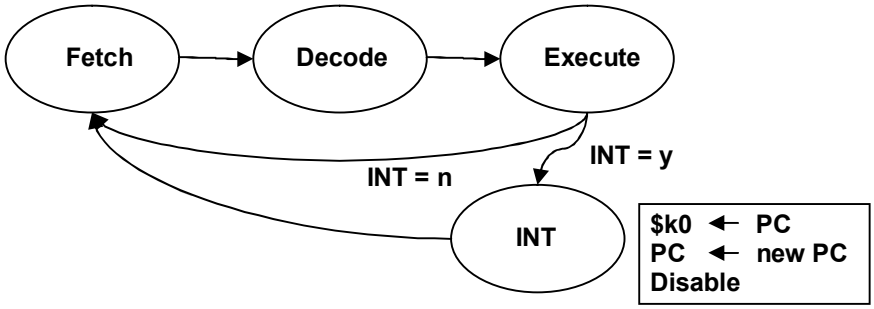

Figure 4.7: Modified FSM with disable interrupts added

Therefore, two things become clear in terms of handling cascaded interrupts.

1. A new instruction to turn off interrupts: disable interrupts
2. A new instruction to turn on interrupts: enable interrupts

Further, the hardware should implicitly turn off interrupts while in the INT state and hand over control to the handler. Figure 4.7 shows the modified FSM with the disable interrupt added to the INT macro state.

Let us now investigate what the handler should do to avoid the situation shown in Figure 4.6. It should save the return address to the original program contained in $k0 while the interrupts are still disabled. Once it does that, it can then enable interrupts to ensure that the processor does not miss interrupts that are more important. Before leaving the handler, it should restore $k0, with interrupts disabled. Figure 4.8 shows this modified interrupt handler. As in the procedure calling convention discussed in Chapter 2, saving/restoring registers is on the stack.

```
Handler:
        /* The interrupts are disabled when we enter */
        save $k0;
        enable interrupts;
        save processor registers;
        execute device code;
        restore processor registers;
        disable interrupts;
        restore $k0;
        return to original program;
```

Figure 4.8: Modified Interrupt Handler

Example 3:

Consider the following program:

```
100    ADD
101    NAND
102    LW
103    NAND
104    BEQ
```

An interrupt occurs when the processor is executing the ADD instruction. The handler code for handling the interrupt is as follows:

```
1000   Save $k0
1001   Enable interrupts
1002   /* next several instructions save processor registers */
……
1020   /* next several instructions execute device code */
……
1102   /* next several instructions restore processor registers */
……
1120   restore $k0
1121   return to original program
```

Assume that a second interrupt occurs at the instruction "restore $k0" (PC = 1120). When will the original program be resumed?

Answer:

The original program **will never be resumed**. Note that the second interrupt will be taken immediately on completion of "restore $k0". Upon completion of this instruction, the handler $k0 = 101, which is the point of resumption of the original program. Now the second interrupt is taken. Unfortunately, the second interrupt (see Figure 4.7) will store the point of resumption of the first handler (memory address = 1121) into $k0. Thus the point of resumption of the original program (memory address = 101) is lost forever. The reason is that the interrupt handler in this example does not have the crucial "disable interrupts" instruction in Figure 4.8.

It should be noted, however, that it might not always be prudent to entertain a second interrupt while servicing the current one. For example, in Section 4.4.1, we will introduce the notion of multiple interrupt levels. Based on their relative speeds, devices will be placed on different interrupt priority levels. For example, a high-speed device such as disk will be placed on a higher priority level compared to a low-speed device such as a keyboard. When the processor is serving an interrupt from the disk, it may temporarily ignore an interrupt coming from a keyboard.

The role of the hardware is to provide the necessary mechanisms for the processor to handle cascaded interrupts correctly. It is a partnership between the handler code (which is part of the operating system) and the processor hardware to determine how best to handle multiple simultaneous interrupts depending on what the processor is doing at that point of time.

Basically, the choice is two-fold:

- ignore the interrupt for a while (if the operating system is currently handling a higher priority interrupt), or
- attend to the interrupt immediately as described in this sub-section.

Ignoring an interrupt temporarily may be implicit using the hardware priority levels or explicit via the "disable interrupt" instruction that is available to the handler program.

4.3.4 Returning from the handler

Once the handler completes execution, it can return to the original program by using the PC value stored in $k0. At first glance, it appears that we should be able to use the mechanism used for returning from a procedure call to return from the interrupt as well. For example, in Chapter 2, to return from a procedure call we introduced the instruction[2],

```
J       r_link
```

Naturally, we are tempted to use the same instruction to return from the interrupt,

```
J       $k0
```

However, there is a problem. Recall that the interrupts should be in the enabled state when we return to the original program. Therefore, we may consider the following sequence of instructions to return from the interrupt:

```
Enable interrupts;
J       $k0;
```

There is a problem with using this sequence of instructions as well to return from an interrupt. Recall that we check for interrupts at the end of each instruction execution. Therefore, between "Enable Interrupts" and "J $k0", we may get a new interrupt that will trash $k0.

Therefore, we introduce a new instruction

```
Return from interrupt (RETI)
```

The semantics of this instruction are as follows:

```
Load PC from $k0;
Enable interrupts;
```

The important point to note is that this instruction is *atomic*, that is, this instruction executes fully before any new interrupts can occur. With this new instruction, Figure 4.9 shows the correct interrupt handler that can handle nested interrupts.

```
Handler:
        /* The interrupts are disabled when we enter */
        save $k0;
        enable interrupts;
        save processor registers;
        execute device code;
        restore processor registers;
        disable interrupts;
        restore $k0;
        return from interrupt;
        /* interrupts will be enabled by return from interrupt */
```

Figure 4.9: Complete Interrupt Handler

[2] Recall that LC-2200 does not have a separate unconditional jump instruction. We can simulate it with the JALR instruction available in LC-2200.

4.3.5 Checkpoint

To summarize, we have made the following architectural enhancements to LC-2200 to handle interrupts:

1. Three new instructions to LC-2200:
   Enable interrupts
   Disable interrupts
   Return from interrupt
2. Upon an interrupt, store the current PC implicitly into a special register $k0.

We now turn to the discussion of the hardware needed to accommodate these architectural enhancements. We already detailed the changes to the FSM at the macro level. We will leave working out the details of the INT macro state in the context of the LC-2200 datapath as an exercise to the reader. To complete this exercise, we will need to figure out the microstates needed to carry out all the actions needed in the INT macro state, and then generate the control signals needed in each microstate.

4.4 Hardware details for handling program discontinuities

As we mentioned earlier in Section 4.2, the architectural enhancements discussed thus far are neutral to the type of discontinuity, namely, exception, trap, or interrupt. In this section, we discuss the hardware details for dealing with program discontinuities in general, and handling external interrupts in particular. We already introduced the interrupt vector table (IVT) and exception/trap register (ETR). We investigate the datapath modifications for interrupts as well as the ability to receive the interrupt vector from an external device. We intentionally keep this discussion simple. The interrupt architecture in modern processors is very sophisticated and we review this briefly in the chapter summary (Section 4.6).

4.4.1 Datapath details for interrupts

We will now discuss the implementation details for handling interrupts. In Chapter 3, we introduced and discussed the concept of a bus for connecting the datapath elements. Let us expand on that concept since it is necessary to understand the datapath extensions for handling interrupts. Specifically, let us propose a bus that connects the processor to the memory and other I/O devices. For the processor to talk to the memory, we need address and data lines. Figure 4.10 shows the datapath with additional lines on the bus for supporting interrupts. There is a wire labeled INT on the bus. Any device that wishes to interrupt the CPU *asserts* this line. In Chapter 3, we emphasized the importance of ensuring that only one entity accesses a shared bus at any point of time. The INT line on the other hand is different. Any number of devices can simultaneously assert this line to indicate their intent to talk to the processor (analogous to multiple students putting up their hands in the classroom example). An electrical circuit concept called *wired-or* logic makes it possible for several devices to assert the INT line simultaneously. The details of

this electrical trick are outside the scope of this course, and the interested reader should refer to a textbook[3] on logic design for more details.

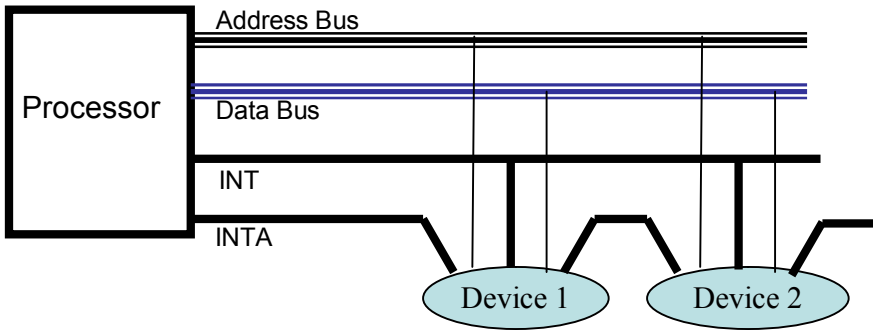

Figure 4.10: Datapath enhancements for handling interrupts

Upon an interrupt, the processor asserts the INTA line (in the INT macro state). Exactly one device should get this acknowledgement, though multiple devices may have signaled their intent to interrupt the processor. Notice the wiring of the INTA. It is not a shared line (like INT) but a chain from one device to another, often referred to as a *daisy chain*. The electrically closest device (Device 1 in Figure 4.10) gets the INTA signal first. If it has requested an interrupt then it knows the processor is ready to talk to it. If it has not requested an interrupt, then it knows that some other device is waiting to talk to the processor and passes the INTA signal down the chain. Daisy chain has the virtue of simplicity but it suffers from latency for the propagation of the acknowledgement signal to the interrupting device, especially in this day and age of very fast processors. For this reason, this method is not used in modern processors.

A generalization of this design principle allows multiple INT and INTA lines to exist on the bus. Each distinct pair of INT and INTA lines corresponds to a *priority level*. Figure 4.11 shows an 8-level priority interrupt scheme. Notice that there is still exactly one device that can talk to the processor at a time (the INTA line from the processor is routed to the INTA line corresponding to the highest priority pending interrupt). Device priority is linked to the speed of the device. The higher the speed of the device the greater the chance of data loss, and hence greater is the need to give prompt attention to that device. Thus, the devices will be arranged on the interrupt lines based on their relative priorities. For example, a disk would be at a higher priority compared to a keyboard. However, despite having multiple interrupt levels, they are insufficient to accommodate all the devices, due to the sheer number, on dedicated interrupt lines. Besides, by definition a device on a higher priority level is more important than the one on a lower priority level. However, there may be devices with similar speed characteristics that belong to the same equivalence class as far as priority goes. Consider a keyboard and a mouse, for instance. Thus, it may still be necessary to place multiple devices on the same priority level as shown in Figure 4.11.

[3] Please see: Yale Patt and Sanjay Patel, "Introduction to Computing Systems," McGraw-Hill.

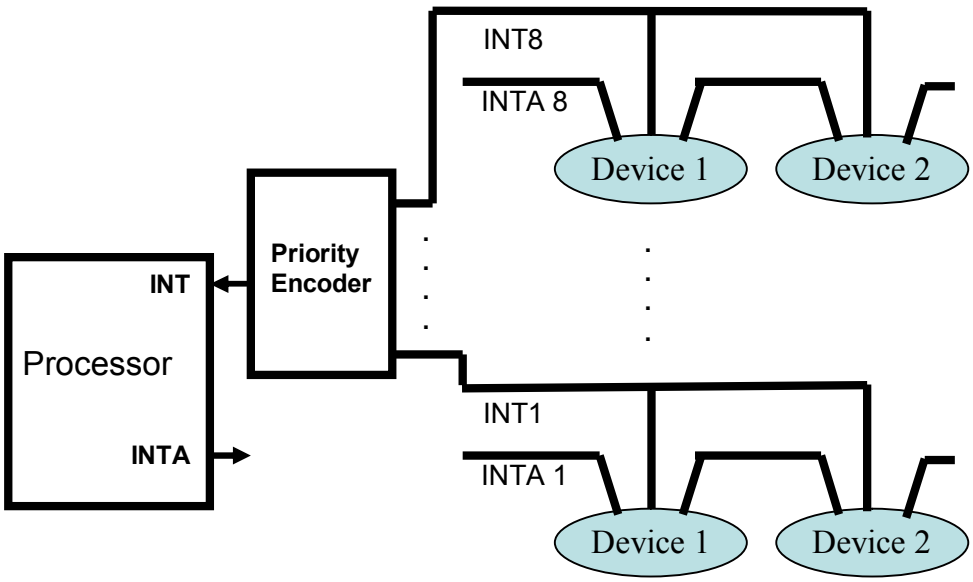

Figure 4.11: Priority Interrupt

As we mentioned earlier, daisy chaining the acknowledgement line through the devices may not be desirable due to the latency associated with the signal propagation. Besides, processor is a precious resource and every attempt is made nowadays to shift unnecessary burden away from the processor to supporting glue logic outside the processor. Therefore, in modern processors, the burden of determining which device needs attention is shifted to an external hardware unit called *interrupt controller*. The interrupt controller takes care of fielding the interrupts, determining the highest priority one to report to the processor, and handling the basic handshake as detailed above for responding to the devices. Instead of daisy chaining the devices in hardware, the operating system chains the interrupt service routines for a given priority level in a linked list. Servicing an interrupt now involves walking through the linked list to determine the first device that has an interrupt pending and choosing that for servicing.

You are probably wondering how all of this relates to the way you plug in a device (say a memory stick or a pair of headphones) into your laptop. Actually, it is no big magic. The position of the device (and hence its priority level) is already pre-determined in the I/O architecture of the system (as in Figure 4.11), and all you are seeing is the external manifestation of the device slot where you have to plug in your device. We will discuss more about computer buses in a later chapter on Input/Output (See Chapter 10).

4.4.2 Details of receiving the address of the handler

Let us now look at how the processor receives the vector from an external device. As we mentioned earlier (see Section 4.2), the interrupt vector table (IVT), set up by the operating system at boot time, contains the handler addresses for all the external interrupts. While a device does not know where in memory its handler code is located, it knows the table entry that will contain it. For example, the keyboard may know its vector

is 80 and the mouse may know that its vector is 82[4]. Upon receiving the INTA signal from the processor (see Figure 4.12), the device puts its vector on the data bus. Recall that the processor is still in the INT macro state. The processor uses this vector as the index to look up in the vector table and retrieve the handler address, which is then loaded into the PC. The operating system reserves a portion of the memory (usually low memory addresses) for housing this vector table. The size of the vector table is a design choice of the operating system. Thus, with this one level of indirection through the interrupt vector table, the processor determines the handler address where the code for executing the procedure associated with this specific interrupt is in memory. Figure 4.12 pictorially shows this interaction between the device and the processor for receiving the handler address.

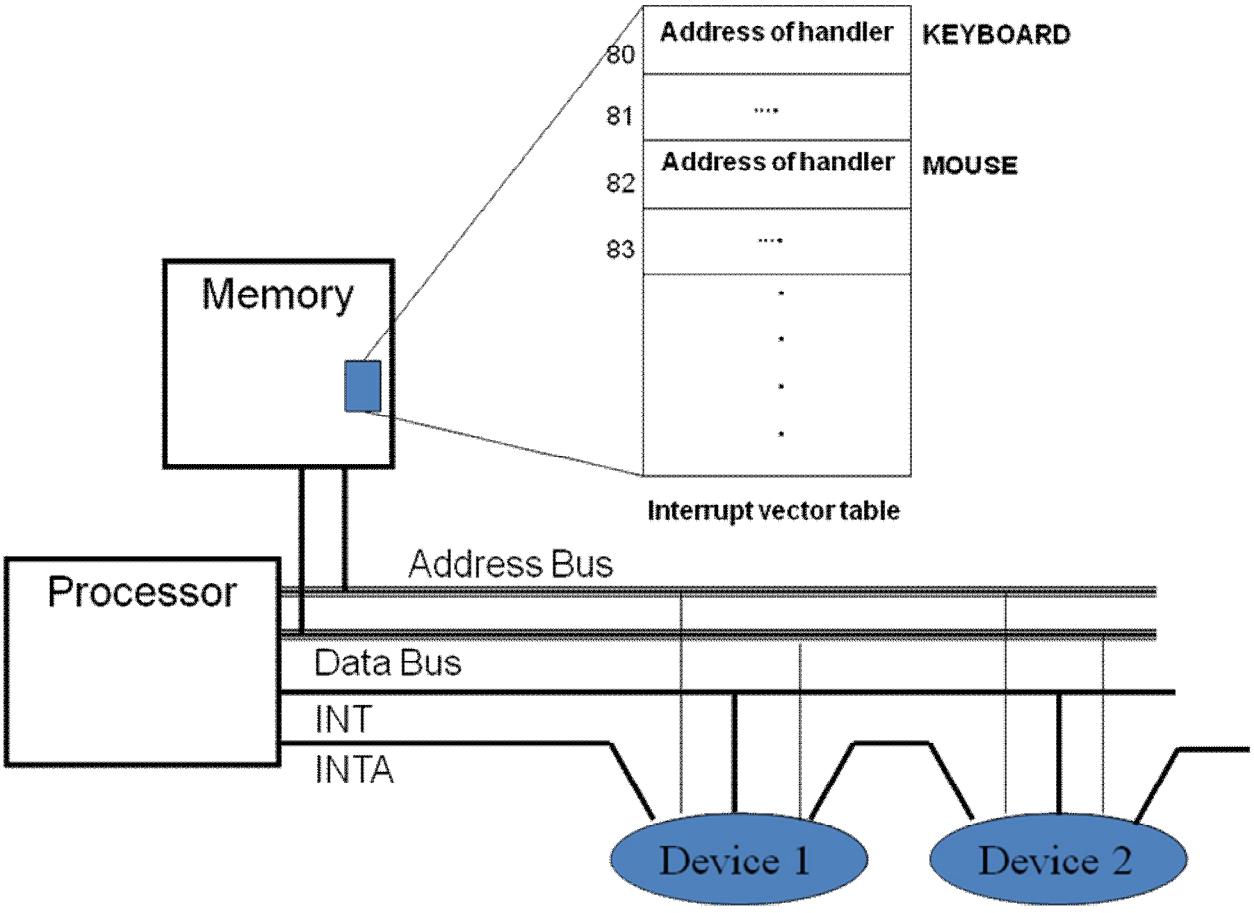

Figure 4.12: Processor-device interaction to receive the interrupt vector

The handshake between the processor and the device is summarized below:

- The device asserts the INT line whenever it is ready to interrupt the processor.

[4] The OS typically decides the table entry for each device that is then "programmed" into the device interface. We will revisit this topic again in Chapter 9 that deals with Input/Output.

- The processor upon completion of the current instruction (Execute macro-state in Figure 4.4-(b)), checks the INT line (in Figure 4.4-(b) we show this check as INT=y/n in the FSM) for pending interrupts.
- If there is a pending interrupt (INT=y in the FSM shown in Figure 4.4-(b)), then the processor enters the INT macro-state and asserts the INTA line on the bus.
- The device upon receiving the INTA from the processor, places its vector (for example, keyboard will put out 80 as its vector) on the data bus.
- The processor receives the vector and looks up the entry in the interrupt vector table corresponding to this vector. Let us say, the address of the handler found in this entry is 0x5000. This is the PC value for the procedure that needs to be executed for handling this interrupt.
- The processor (which is still in the INT macro state), completes the action in the INT macro-state as shown in Figure 4.4-(b), saving the current PC in $k0, and loading the PC with the value retrieved from the interrupt vector table.

4.4.3 Stack for saving/restoring

Figure 4.9 shows the handler code which includes saving and restoring of registers (similar to the procedure calling convention). The stack seems like an obvious place to store the processor registers. However, there is a problem. How does the handler know which part of the memory is to be used as a stack? The interrupt may not even be for the currently running program after all.

For this reason, it is usual for the architecture to have two stacks: *user stack* and *system stack*. Quite often, the architecture may designate a particular register as the stack pointer. Upon entering the INT macro state, the FSM performs *stack switching*.

Let us see what hardware enhancements are needed to facilitate this stack switching.

1. **Duplicate stack pointer:**
   Really, all that needs to be done is to duplicate the register designated by the architecture as the stack pointer. In Chapter 2, we designated an architectural register $sp as the stack pointer. We will duplicate that register: one for use by the user program and the other for use by the system. The state saving in the interrupt handler will use the system version of $sp while the user programs will use the user version of $sp. This way, the state saving will not disturb the user stack since all the saving/restoring happens on the system stack. At system start up (i.e., boot time), the system initializes the system version of $sp with the address of the system stack with sufficient space allocated to deal with interrupts (including nested interrupts).

2. **Privileged mode:**
   Remember that the interrupt handler is simply a program. We need to let the processor know which version of $sp to use at any point of time. For this reason, we introduce a *mode* bit in the processor. The processor is in *user* or *kernel* mode depending on the value of this bit. If the processor is in user mode the hardware

implicitly uses the user version of $sp. If it is in kernel mode it uses the kernel version of $sp. The FSM sets this bit in the INT macro state. Thus, the handler will run in kernel mode and hence use the system stack. Before returning to the user program, RETI instruction sets the mode bit back to "user" to enable the user program to use the user stack when it resumes execution.

The mode bit also serves another important purpose. We introduced three new instructions to support interrupt. It would not be prudent to allow any program to execute these instructions. For example, the register $k0 has a special connotation and any arbitrary program should not be allowed to write to it. Similarly, any arbitrary program should not be allowed to enable and disable interrupts. Only the operating system executes these instructions, referred to as *privileged instructions*. We need a way to prevent normal user programs from trying to execute these privileged instructions, either accidentally or maliciously. An interrupt handler is part of the operating system and runs in the "kernel" mode (the FSM sets the mode bit in the INT macro state to "kernel"). If a user program tries to use these instructions it will result in an illegal instruction trap.

We need to address two more subtle points of detail to complete the interrupt handling. First, recall that interrupts could be nested. Since all interrupt handlers run in the kernel mode, we need to do the mode switching (and the implicit stack switching due to the mode bit) in the INT macro state only when the processor is going from a user program to servicing interrupts. Further, we need to remember the current mode the processor is in to take the appropriate action (whether to return to user or kernel mode) when the handler executes RETI instruction. The system stack is a convenient vehicle to remember the current mode of the processor.

The INT macro state and the RETI instruction push and pop the current mode of the processor on to the system stack, respectively. The INT macro state and the RETI instruction take actions commensurate with the current mode of the processor. Figure 4.13 summarizes all the actions taken in the INT macro state; and Figure 4.14 summarizes the semantics of the RETI instruction.

```
INT macro state:
        $k0 ←PC;
        ACK INT by asserting INTA;
        Receive interrupt vector from device on the data bus;
        Retrieve address of the handler from the interrupt vector table;
        PC ← handler address retrieved from the vector table;
        Save current mode on the system stack;
        mode = kernel;  /* noop if the mode is already kernel */
        Disable interrupts;
```

Figure 4.13: Actions in the INT Macro State

```
RETI:
        Load PC from $k0;
        /* since the handler executes RETI, we are in the kernel mode */
        Restore mode from the system stack; /* return to previous mode */
        Enable interrupts;
```

Figure 4.14: Semantics of RETI instruction

4.5 Putting it all together

4.5.1 Summary of Architectural/hardware enhancements

To deal with program discontinuities, we added the following architectural/hardware enhancements to LC-2200:

1. An interrupt vector table (IVT), to be initialized by the operating system with handler addresses.
2. An exception/trap register (ETR) that contains the vector for internally generated exceptions and traps.
3. A Hardware mechanism for receiving the vector for an externally generated interrupt.
4. User/kernel mode and associated mode bit in the processor.
5. User/system stack corresponding to the mode bit.
6. A hardware mechanism for storing the current PC implicitly into a special register $k0, upon an interrupt, and for retrieving the handler address from the IVT using the vector (either internally generated or received from the external device).
7. Three new instructions to LC-2200:
   Enable interrupts
   Disable interrupts
   Return from interrupt

4.5.2 Interrupt mechanism at work

We present a couple of examples to bring all these concepts together and give the reader an understanding of how the interrupt mechanism works. For clarity of presentation, we will call the system version of $sp as SSP and the user version of $sp as USP. However, architecturally (i.e., from the point of view of the instruction set) they refer to the same register. The hardware knows (via the mode bit) whether to use USP or SSP as $sp.

Example 4:

The example sketched in Figure 4.15 (A-D) illustrates the sequence of steps involved in interrupt handling. Some program **foo** is executing (Figure 4.15-A).

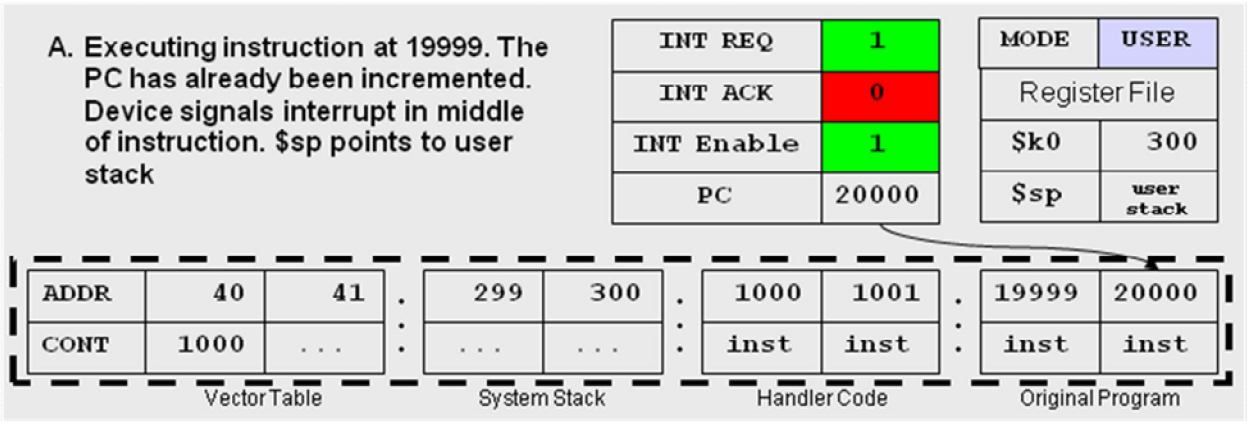

Figure 4.15-A: Interrupt Handling (INT received)

A keyboard device interrupts the processor. The processor is currently executing an instruction at location 19999. It waits until the instruction execution completes. It then goes to the INT macro state (Figure 4.15-B) and saves the current PC (whose value is 20000) in $k0. Upon receipt of INTA from the processor, the device puts out its vector on the data bus. The processor receives the vector from the device on the data bus as shown in Figure 4.15-B.

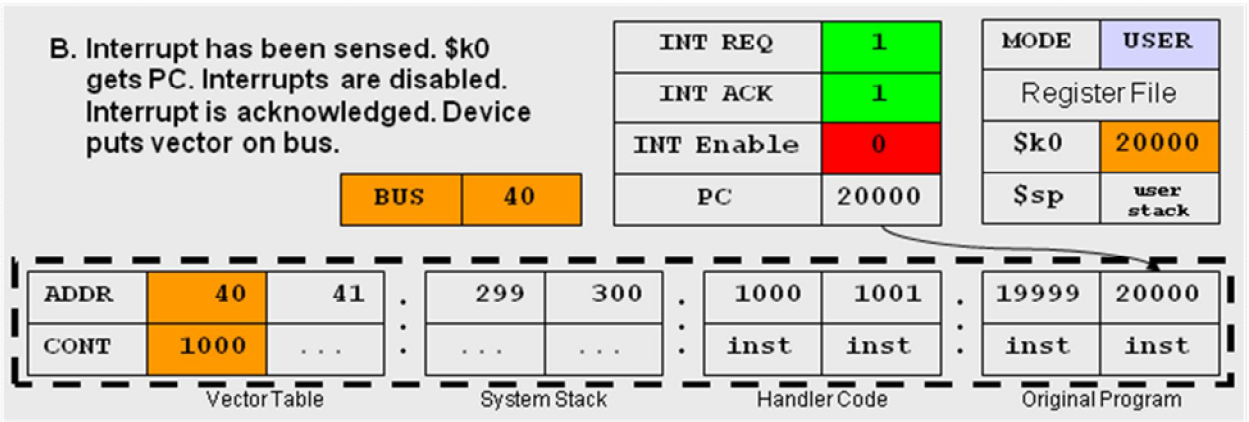

Figure 4.15-B: Interrupt Handling (INT macro state – receive vector)

Let us say the value received is 40. The processor looks up memory location 40 to get the handler address (let this be 1000). Let the contents of the SSP be 300. In the INT macro state, the FSM loads 1000 into PC, 300 into $sp, saves the current mode on the system stack, and goes back to the FETCH macro state. The handler code at location 1000 (similar to Figure 4.9) starts executing using $sp = 299 as its stack (Figure 4.15-C).

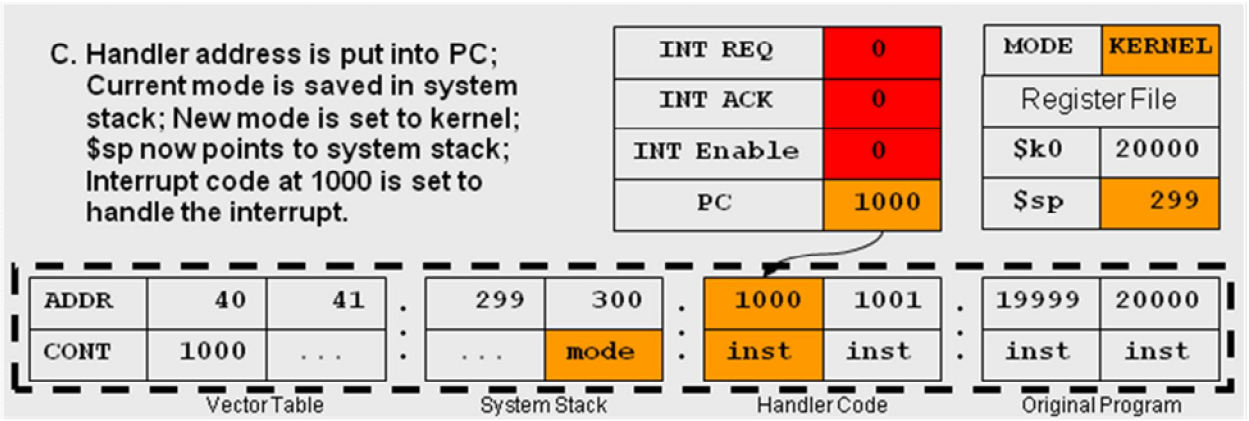

Figure 4.15-C: Interrupt Handling (Transfer control to handler code)

The original program will resume execution at PC = 20000 when the handler executes return from interrupt (Figure 4.15-D).

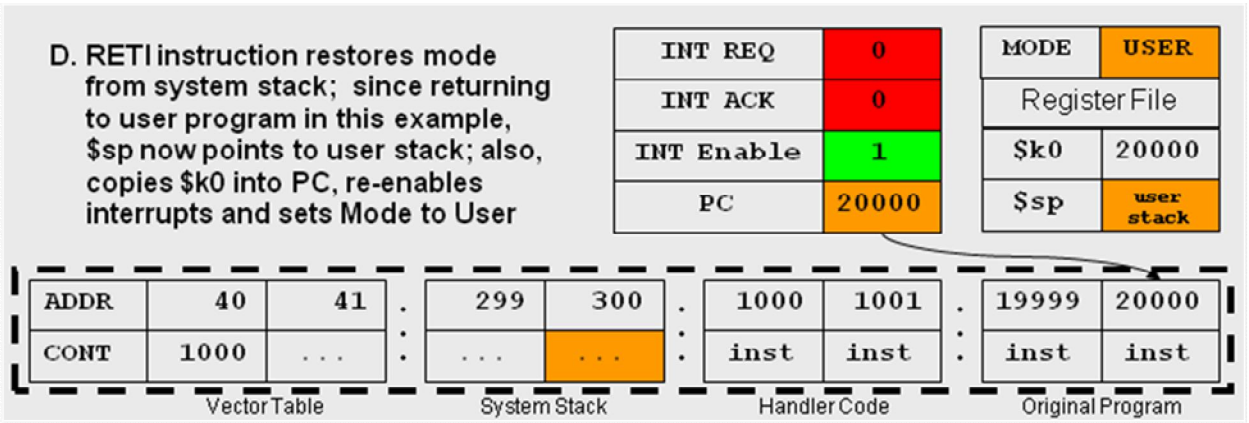

Figure 4.15-D: Interrupt Handling (return to original program)

Example 5:

Consider the following:

Assume memory addresses are consecutive integers
User program executing instruction at memory location 7500
User Stack pointer ($sp) value 18000
SSP value 500
Vector for keyboard = 80
Vector for disk = 50
Handler address for keyboard interrupt = 3000
Handler address for disk interrupt = 5000

(a) Pictorially represent the above information similar to Figure 4.15
(b) An interrupt occurs from the keyboard. Show the relevant state of the processor (similar to Figure 4.15-C) when the interrupt handler for the keyboard is about to start executing.

(c) A higher priority interrupt from the disk occurs while the keyboard handler is running after it has re-enabled interrupts. Assume that it is executing an instruction at memory location 3023, and the stack pointer ($sp) value is 515. Show the relevant state of the processor (similar to Figure 4.15-C) when the interrupt handler for the disk is about to start executing.
(d) Show the relevant state of the processor when the disk handler executes RETI instruction (similar to Figure 4.15-D).
(e) Show the relevant state of the processor when the keyboard handler executes RETI instruction (similar to Figure 4.15-D).

This example is left as an exercise for the reader.

4.6 Summary

In this Chapter, we introduced an important concept, interrupts, that allows a processor to communicate with the outside world. An interrupt is a specific instance of program discontinuity. We discussed the minimal hardware enhancements needed inside the processor as well as at the bus level to handle nested interrupts.

- The processor enhancements included three new instructions, a user stack, a system stack, a mode bit, and a new macro state called INT.
- At the bus level, we introduced special control lines called INT and INTA for the device to indicate to the processor that it wants to interrupt and for the processor to acknowledge the interrupt, respectively.

We reviewed traps and exceptions that are synchronous versions of program discontinuities as well. The interesting thing is that the software mechanism needed to handle all such discontinuities is similar. We discussed how to write a generic interrupt handler that can handle nested interrupts.

We have intentionally simplified the presentation of interrupts in this chapter to make it accessible to students in a first systems course. Interrupt mechanisms in modern processors are considerably more complex. For example, modern processors categorize interrupts into two groups: *maskable* and *non-maskable*.

- The former refers to interrupts that can be temporarily turned off using the disable interrupt mechanism (e.g., a device interrupt).
- The latter corresponds to interrupts that cannot be turned off even with the disable interrupt mechanism (e.g., an internal hardware error detected by the system).

We presented a mechanism by which a processor learns the starting address of the interrupt handler (via the vector table) and the use of a dedicated register for stashing the return address for the interrupted program. We also presented a simple hardware scheme by which a processor may discover the identity of the interrupting device and acknowledge the interrupt. The main intent is give confidence to the reader that designing such hardware is simple and straightforward.

We presented mode as a characterization of the internal state of a processor. This is also an intentionally simplistic view. The processor state may have a number of other attributes available as discrete bits of information (similar to the mode bit). Usually, a processor aggregates all of these bits into one register called *processor status word (PSW)*. Upon an interrupt and its return, the hardware implicitly pushes and pops, respectively, both the PC and the PSW on the system stack[5].

We also presented a fairly simple treatment of the interrupt handler code to understand what needs to be done in the processor architecture to deal with interrupts. The handler would typically do a lot more than save processor registers. In later Chapters, we will revisit interrupts in the context of operating system functionalities such as processor scheduling (Chapter 6) and device drivers (Chapter 10).

Interrupt architecture of modern processors is much more sophisticated. First of all, since the processor is a precious resource, most of the chores associated with interrupt processing except for executing the actual handler code is kept outside the processor. For example, a device called *programmable interrupt controller (PIC)* aids the processor in dealing with many of the nitty-gritty details with handling external interrupts including:

- Dealing with multiple interrupt levels,
- Fielding the actual interrupts from the devices,
- Selecting the highest priority device among the interrupting devices,
- Getting the identity (vector table index) of the device selected to interrupt the processor, and
- Acknowledging the selected device

The PIC provides processor-readable registers, one of which contains the identity of the device selected to interrupt the processor. The use of PIC simplifies what the processor has to do on an interrupt. Upon an interrupt, the return address either is placed on a system stack or is made available in a special processor register, and control is simply transferred to a well-defined address set by the operating system, which corresponds to a generic first level interrupt handler of the operating system. This handler simply saves the return address for the interrupted program, reads the identity of the interrupting device from the PIC, and jumps to the appropriate handler code. Usually this first level operating system handler is non-interruptible and may run in a special mode called *interrupt mode*, so the operating system does not have to worry about nested interrupts. In general, it is important that a device driver – software that actually manipulates a device – do as little work in an interrupt handling code as possible. This is to make sure that the processor is not tied up forever in interrupt processing. Device drivers are written such that only the time sensitive code is in the interrupt handler. For example, Linux operating system defines *top-half* and *bottom-half* handlers. By definition, the bottom-half handlers do not have the same sense of urgency as the top-half handlers. A device that has significant amount of work to do that is not time critical will do it in the bottom-half handler.

[5] In LC-2200, we designate a register $k0 for saving PC in the INT macro state. An alternative approach adopted in many modern processors is to save the PC directly on the system stack.

We have only scratched the surface of issues relating to interrupt architecture of processors and operating system mechanisms for efficiently dealing with them. The interested reader is referred to more advanced textbooks on computer architecture[6] for details on how the interrupt architecture is implemented in modern processors, as well as to books on operating system implementation[7] to get a deeper knowledge on interrupt handling.

4.7 Review Questions

1. Upon an interrupt what has to happen implicitly in hardware before control is transferred to the interrupt handler?

2. Why not use JALR to return from the interrupt handler?

3. Put the following steps in the correct order

   Actual work of the handler
   Disable interrupt
   Enable interrupt
   Restore ko from stack
   Restore state
   Return from interrupt
   Save ko on stack
   Save state

4. How does the processor know which device has requested an interrupt?

5. What instructions are needed to implement interruptible interrupts? Explain the function and purpose of each along with an explanation of what would happen if you didn't have them.

6. In the following interrupt handler code select the **ONES THAT DO NOT BELONG**.

   _____ disable interrupts;
   _____ save PC;
   _____ save $k0;
   _____ enable interrupts;
   _____ save processor registers;
   _____ execute device code;
   _____ restore processor registers;
   _____ disable interrupts;
   _____ restore $k0;

[6] Hennessy and Patterson, "Computer Architecture: A Quantitative Approach," Morgan Kaufmann publishers.
[7] Alessandro Rubini & Jonathan Corbet , "Linux Device Drivers," 2nd Edition, O'Reilly & Associates

_____ disable interrupts;
_____ restore PC;
_____ enable interrupts;
_____ return from interrupt;

7. In the following actions in the INT macro state select the **ONES THAT DO NOT BELONG**.

_____ save PC;
_____ save SP;
_____ $k0 ← PC;
_____ enable interrupts;
_____ save processor registers;
_____ ACK INT by asserting INTA;
_____ Receive interrupt vector from device on the data bus;
_____ Retrieve PC address from the interrupt vector table;
_____ Retrieve SP value from the interrupt vector table;
_____ disable interrupts
_____ PC ← PC retrieved from the vector table;
_____ SP ← SP value retrieved from the vector table;
_____ disable interrupts;