```
If IR[11-10] = 00 then          {if bits 10 and 11 in the IR are both 0}
   If AC < 0 then PC ← PC + 1
else If IR[11-10] = 01 then     {if bit 11 = 0 and bit 10 = 1}
   If AC = 0 then PC ← PC + 1
else If IR[11-10] = 10 then     {if bit 11 = 1 and bit 10 = 0}
   If AC > 0 then PC ← PC + 1
```

If the bits in positions ten and eleven are both ones, an error condition results. However, an additional condition could also be defined using these bit values.

Jump *X*

This instruction causes an unconditional branch to the given address, *X*. Therefore, to execute this instruction, *X* must be loaded into the PC.

PC ← *X*

In reality, the lower or least significant 12 bits of the instruction register (or IR[11–0]) reflect the value of *X*. So this transfer is more accurately depicted as:

PC ← IR[11-0]

However, we feel that the notation PC ← *X* is easier to understand and relate to the actual instructions, so we use this instead.

Register transfer notation is a symbolic means of expressing what is happening in the system when a given instruction is executing. RTN is sensitive to the datapath, in that if multiple microoperations must share the bus, they must be executed in a sequential fashion, one following the other.

4.9 INSTRUCTION PROCESSING

Now that we have a basic language with which to communicate ideas to our computer, we need to discuss exactly how a specific program is executed. All computers follow a basic machine cycle: the fetch, decode, and execute cycle.

4.9.1 The Fetch–Decode–Execute Cycle

The **fetch–decode–execute cycle** represents the steps that a computer follows to run a program. The CPU fetches an instruction (transfers it from main memory to the instruction register), decodes it (determines the opcode and fetches any data necessary to carry out the instruction), and executes it (performs the operation[s] indicated by the instruction). Notice that a large part of this cycle is spent copying data from one location to another. When a program is initially loaded, the address of the first instruction must be placed in the PC. The steps in this cycle, which take place in specific clock cycles, are listed below. Note that Steps 1 and

2 make up the fetch phase, Step 3 makes up the decode phase, and Step 4 is the execute phase.

1. Copy the contents of the PC to the MAR: `MAR ← PC`.
2. Go to main memory and fetch the instruction found at the address in the MAR, placing this instruction in the IR; increment PC by 1 (PC now points to the next instruction in the program): `IR ← M[MAR]` and then `PC ← PC+1`. (*Note:* Because MARIE is word addressable, the PC is incremented by 1, which results in the next word's address occupying the PC. If MARIE were byte addressable, the PC would need to be incremented by 2 to point to the address of the next instruction, because each instruction would require 2 bytes. On a byte-addressable machine with 32-bit words, the PC would need to be incremented by 4.)
3. Copy the rightmost 12 bits of the IR into the MAR; decode the leftmost 4 bits to determine the opcode, `MAR ← IR[11-0]`, and decode IR[15–12].
4. If necessary, use the address in the MAR to go to memory to get data, placing the data in the MBR (and possibly the AC), and then execute the instruction `MBR ← M[MAR]` and execute the actual instruction.

This cycle is illustrated in the flowchart in Figure 4.11.

Note that computers today, even with large instruction sets, long instructions, and huge memories, can execute millions of these fetch–decode–execute cycles in the blink of an eye.

4.9.2 Interrupts and the Instruction Cycle

All computers provide a means for the normal fetch–decode–execute cycle to be interrupted. These interruptions may be necessary for many reasons, including a program error (such as division by 0, arithmetic overflow, stack overflow, or attempting to access a protected area of memory); a hardware error (such as a memory parity error or power failure); an I/O completion (which happens when a disk read is requested and the data transfer is complete); a user interrupt (such as hitting Ctrl-C or Ctrl-Break to stop a program); or an interrupt from a timer set by the operating system (such as is necessary when allocating virtual memory or performing certain bookkeeping functions). All of these have something in common: they interrupt the normal flow of the fetch–decode–execute cycle and tell the computer to stop what it is currently doing and go do something else. They are, naturally, called **interrupts**.

The speed with which a computer processes interrupts plays a key role in determining the computer's overall performance. **Hardware interrupts** can be generated by any peripheral on the system, including memory, the hard drive, the keyboard, the mouse, or even the modem. Instead of using interrupts, processors could poll hardware devices on a regular basis to see if they need anything done. However, this would waste CPU time as the answer would more

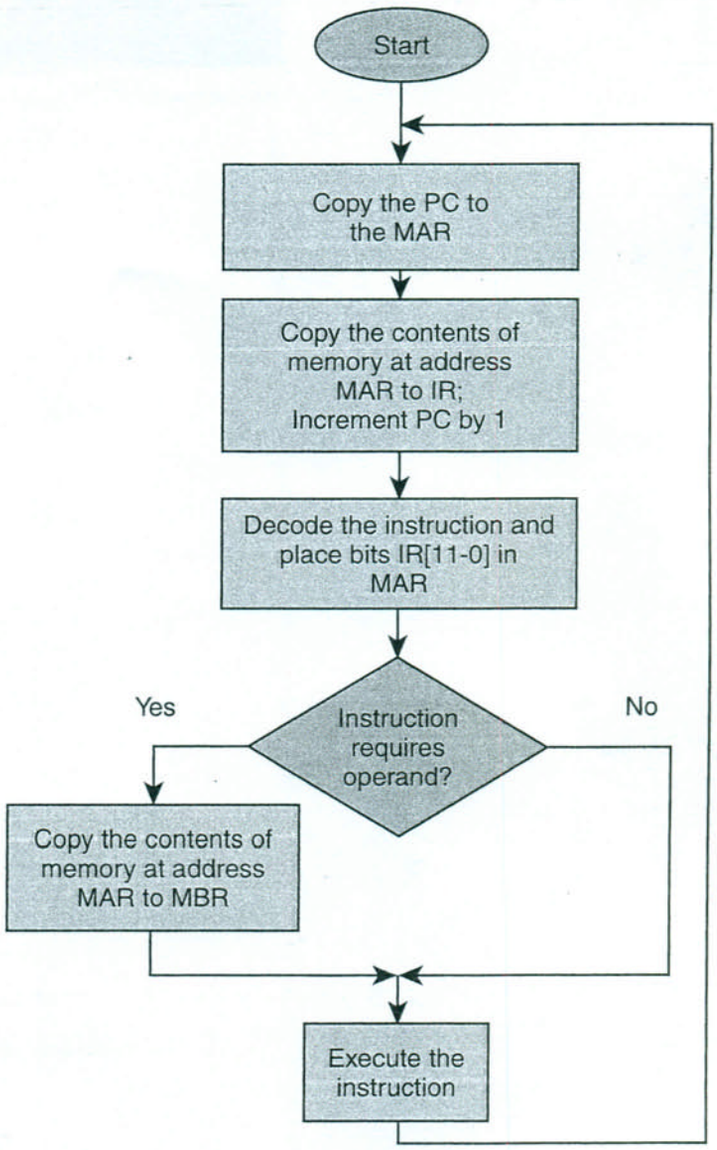

FIGURE 4.11 The Fetch–Decode–Execute Cycle

often than not be "no." Interrupts are nice because they let the CPU know the device needs attention at a particular moment without requiring the CPU to constantly monitor the device. Suppose you need specific information that a friend has promised to acquire for you. You have two choices: call the friend on a regular schedule (polling) and waste his or her time and yours if the information is not ready, or wait for a phone call from your friend once the information has been acquired. You may be in the middle of a conversation with someone else when the phone call "interrupts" you, but the latter approach is by far the more efficient way to handle the exchange of information.

Computers also employ **software interrupts** (also called **traps** or **exceptions**) used by various software applications. Modern computers support both software and hardware interrupts by using **interrupt handlers**. These han-

dlers are simply routines (procedures) that are executed when their respective interrupts are detected. The interrupts, along with their associated **interrupt service routines** (**ISR**s), are stored in an **interrupt vector table**.

How do interrupts fit into the fetch–decode–execute cycle? The CPU finishes execution of the current instruction and checks, at the beginning of every fetch–decode–execute cycle, to see if an interrupt has been generated, as shown in Figure 4.12. Once the CPU acknowledges the interrupt, it must then process the interrupt.

The details of the "Process the Interrupt" block are given in Figure 4.13. This process, which is the same regardless of what type of interrupt has been invoked, begins with the CPU detecting the interrupt signal. Before doing anything else, the system suspends whatever process is executing by saving the program's state and variable information. The device ID or interrupt request number of the device causing the interrupt is then used as an index into the interrupt vector table, which is kept in very low memory. The address of the interrupt service routine (known as its **address vector**) is retrieved and placed into the program counter, and execution resumes (the fetch–decode–execute cycle begins again) within the service routine. After the interrupt service has completed, the system restores the information it saved from the program that was running when the interrupt occurred, and program execution may resume—unless another interrupt is detected, whereupon the interrupt is serviced as described.

It is possible to suspend processing of non-critical interrupts by use of a special interrupt mask bit found in the flag register. This is called **interrupt masking**, and interrupts that can be suspended are called **maskable** interrupts. **Nonmaskable** interrupts cannot be suspended, because to do so, it is possible that the system would enter an unstable or unpredictable state.

Assembly languages provide specific instructions for working with hardware and software interrupts. When writing assembly language programs, one of the

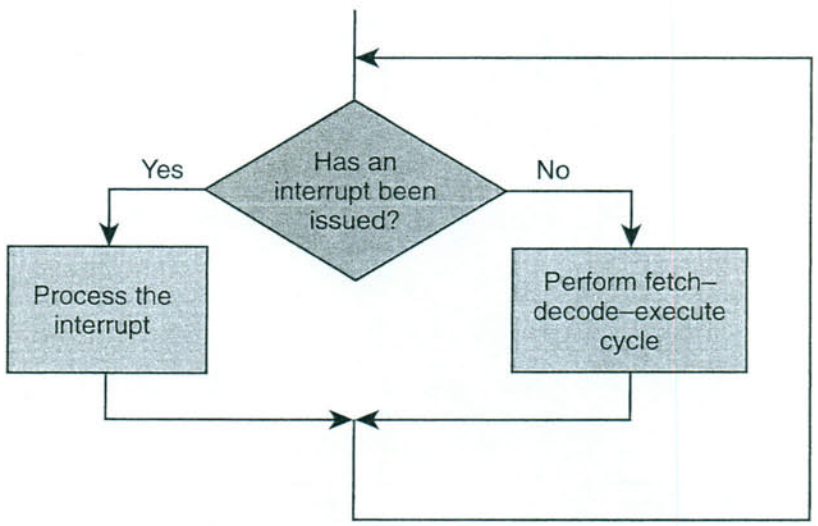

FIGURE 4.12 Fetch–Decode–Execute Cycle with Interrupt Checking

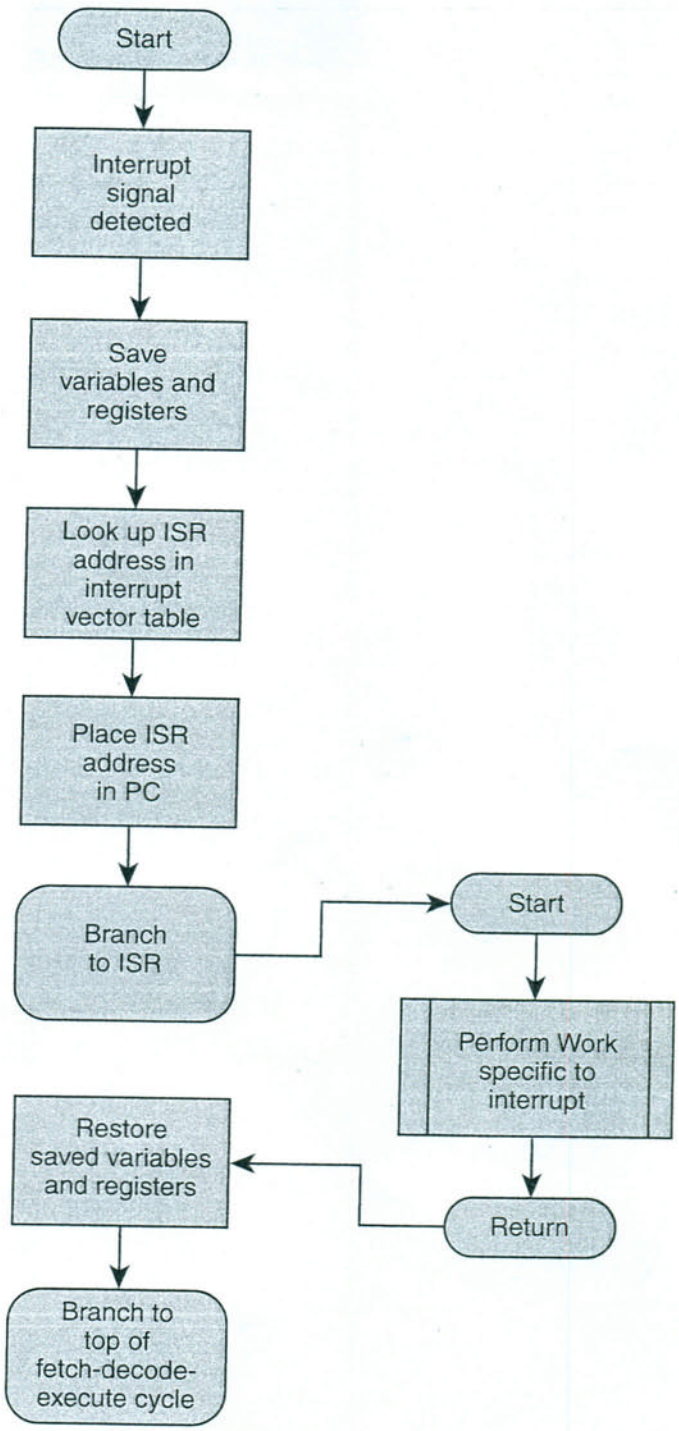

FIGURE 4.13 Processing an Interrupt

most common tasks is dealing with I/O through software interrupts (see Chapter 7 for additional information on interrupt-driven I/O). Indeed, one of the more complicated functions for the novice assembly language programmer is reading input and writing output, specifically because this must be done using interrupts. MARIE simplifies the I/O process for the programmer by avoiding the use of interrupts for I/O.

4.9.3 MARIE's I/O

I/O processing is one of the most challenging aspects of computer system design and programming. Our model is necessarily simplified, and we provide it at this point only to complete MARIE's functionality.

MARIE has two registers to handle input and output. One, the input register, holds data being transferred from an input device into the computer; the other, the output register, holds information ready to be sent to an output device. The timing used by these two registers is very important. For example, if you are entering input from the keyboard and type very fast, the computer must be able to read each character that is put into the input register. If another character is entered into that register before the computer has a chance to process the current character, the current character is lost. It is more likely, because the processor is very fast and keyboard input is very slow, that the processor might read the same character from the input register multiple times. We must avoid both of these situations.

To get around problems like these, MARIE employs a modified type of programmed I/O (discussed in Chapter 7) that places all I/O under the direct control of the programmer. MARIE's output action is simply a matter of placing a value into the OutREG. This register can be read by an output controller that sends it to an appropriate output device, such as a terminal display, printer, or disk. For input, MARIE, being the simplest of simple systems, places the CPU into a wait state until a character is entered into the InREG. The InREG is then copied to the accumulator for subsequent processing as directed by the programmer. We observe that this model provides no concurrency. The machine is essentially idle while waiting for input. Chapter 7 explains other approaches to I/O that make more efficient use of machine resources.

4.10 A SIMPLE PROGRAM

We now present a simple program written for MARIE. In Section 4.12, we present several additional examples to illustrate the power of this minimal architecture. It can even be used to run programs with procedures, various looping constructs, and different selection options.

Our first program adds two numbers together (both of which are found in main memory), storing the sum in memory. (We forgo I/O for now.)

Hex Address	Instruction		Binary Contents of Memory Address	Hex Contents of Memory
100	Load	104	0001000100000100	1104
101	Add	105	0011000100000101	3105
102	Store	106	0010000100000110	2106
103	Halt		0111000000000000	7000
104	0023		0000000000100011	0023
105	FFE9		1111111111101001	FFE9
106	0000		0000000000000000	0000

TABLE 4.3 A Program to Add Two Numbers

Table 4.3 lists an assembly language program to do this, along with its corresponding machine-language program. The list of instructions under the Instruction column constitutes the actual assembly language program. We know that the fetch–decode–execute cycle starts by fetching the first instruction of the program, which it finds by loading the PC with the address of the first instruction when the program is loaded for execution. For simplicity, let's assume our programs in MARIE are always loaded starting at address 100 (in hex).

The list of instructions under the Binary Contents of Memory Address column constitutes the actual machine language program. It is often easier for humans to read hexadecimal as opposed to binary, so the actual contents of memory are displayed in hexadecimal.

This program loads 0023_{16} (or decimal value 35) into the AC. It then adds the hex value FFE9 (decimal -23) that it finds at address 105. This results in a value of 12 in the AC. The `Store` instruction stores this value at memory location 106. When the program is done, the binary contents of location 106 change to 0000000000001100, which is hex 000C, or decimal 12. Figure 4.14 indicates the contents of the registers as the program executes.

The last RTN instruction in Part c places the sum at the proper memory location. The statement "decode IR[15–12]" simply means the instruction must be decoded to determine what is to be done. This decoding can be done in software (using a microprogram) or in hardware (using hardwired circuits). These two concepts are covered in more detail in Section 4.13.

Note that there is a one-to-one correspondence between the assembly language and the machine-language instructions. This makes it easy to convert assembly language into machine code. Using the instruction tables given in this chapter, you should be able to hand assemble any of our example programs. For this reason, we look at only the assembly language code from this point on. Before we present more programming examples, however, a discussion of the assembly process is in order.