

No artifact designed by man is so convenient for this kind of functional description as a digital computer. Almost the only ones of its properties that are detectable in its behavior are the organizational properties. Almost no interesting statement that one can make about an operating computer bears any particular relation to the specific nature of the hardware. A computer is an organization of elementary functional components in which, to a high approximation, only the function performed by those components is relevant to the behavior of the whole system.

THE SCIENCES OF THE ARTIFICIAL, HERBERT SIMON

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Describe the basic elements of a computer system and their interrelationship.
- Explain the steps taken by a processor to execute an instruction.
- Understand the concept of interrupts and how and why a processor uses interrupts.
- List and describe the levels of a typical computer memory hierarchy.
- Explain the basic characteristics of multiprocessor and multicore organizations.
- Discuss the concept of locality and analyze the performance of a multilevel memory hierarchy.
- Understand the operation of a stack and its use to support procedure call and return.

An operating system (OS) exploits the hardware resources of one or more processors to provide a set of services to system users. The OS also manages secondary memory and I/O (input/output) devices on behalf of its users. Accordingly, it is important to have some understanding of the underlying computer system hardware before we begin our examination of operating systems.

This chapter provides an overview of computer system hardware. In most areas, the survey is brief, as it is assumed that the reader is familiar with this subject. However, several areas are covered in some detail because of their importance to topics covered later in the book. Further topics are covered in Appendix C.

1.1 BASIC ELEMENTS

At a top level, a computer consists of processor, memory, and I/O components, with one or more modules of each type. These components are interconnected in some fashion to achieve the main function of the computer, which is to execute programs. Thus, there are four main structural elements:

- **Processor:** Controls the operation of the computer and performs its data processing functions. When there is only one processor, it is often referred to as the **central processing unit** (CPU).

- **Main memory:** Stores data and programs. This memory is typically volatile; that is, when the computer is shut down, the contents of the memory are lost. In contrast, the contents of disk memory are retained even when the computer system is shut down. Main memory is also referred to as *real memory* or *primary memory*.
- **I/O modules:** Move data between the computer and its external environment. The external environment consists of a variety of devices, including secondary memory devices (e.g., disks), communications equipment, and terminals.
- **System bus:** Provides for communication among processors, main memory, and I/O modules.

Figure 1.1 depicts these top-level components. One of the processor's functions is to exchange data with memory. For this purpose, it typically makes use of two internal (to the processor) registers: a memory address register (MAR), which specifies the address in memory for the next read or write; and a memory buffer register (MBR), which contains the data to be written into memory or which receives

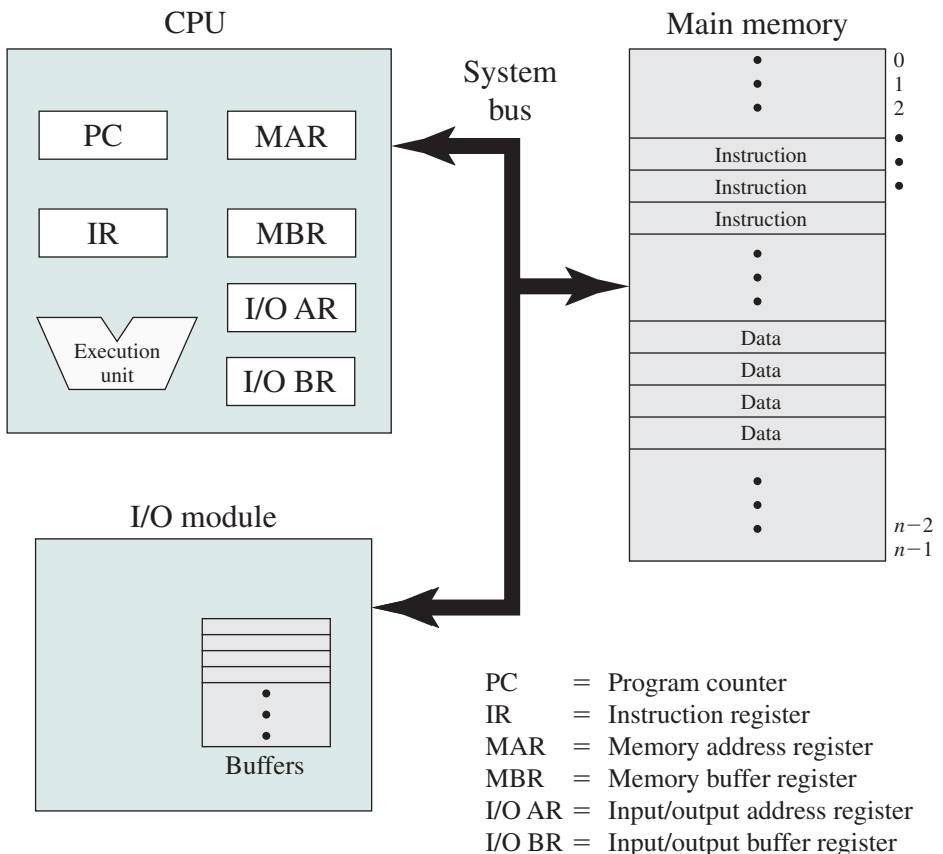


Figure 1.1 Computer Components: Top-Level View

the data read from memory. Similarly, an I/O address register (I/OAR) specifies a particular I/O device. An I/O buffer register (I/OBR) is used for the exchange of data between an I/O module and the processor.

A memory module consists of a set of locations, defined by sequentially numbered addresses. Each location contains a bit pattern that can be interpreted as either an instruction or data. An I/O module transfers data from external devices to processor and memory, and vice versa. It contains internal buffers for temporarily holding data until they can be sent on.

1.2 EVOLUTION OF THE MICROPROCESSOR

The hardware revolution that brought about desktop and handheld computing was the invention of the microprocessor, which contained a processor on a single chip. Though originally much slower than multichip processors, microprocessors have continually evolved to the point that they are now much faster for most computations due to the physics involved in moving information around in sub-nanosecond timeframes.

Not only have microprocessors become the fastest general purpose processors available, they are now multiprocessors; each chip (called a socket) contains multiple processors (called cores), each with multiple levels of large memory caches, and multiple logical processors sharing the execution units of each core. As of 2010, it is not unusual for even a laptop to have 2 or 4 cores, each with 2 hardware threads, for a total of 4 or 8 logical processors.

Although processors provide very good performance for most forms of computing, there is increasing demand for numerical computation. Graphical Processing Units (GPUs) provide efficient computation on arrays of data using Single-Instruction Multiple Data (SIMD) techniques pioneered in supercomputers. GPUs are no longer used just for rendering advanced graphics, but they are also used for general numerical processing, such as physics simulations for games or computations on large spreadsheets. Simultaneously, the CPUs themselves are gaining the capability of operating on arrays of data—with increasingly powerful vector units integrated into the processor architecture of the x86 and AMD64 families.

Processors and GPUs are not the end of the computational story for the modern PC. Digital Signal Processors (DSPs) are also present, for dealing with streaming signals—such as audio or video. DSPs used to be embedded in I/O devices, like modems, but they are now becoming first-class computational devices, especially in handhelds. Other specialized computational devices (fixed function units) co-exist with the CPU to support other standard computations, such as encoding/decoding speech and video (codecs), or providing support for encryption and security.

To satisfy the requirements of handheld devices, the classic microprocessor is giving way to the System on a Chip (SoC), where not just the CPUs and caches are on the same chip, but also many of the other components of the system, such as DSPs, GPUs, I/O devices (such as radios and codecs), and main memory.

1.3 INSTRUCTION EXECUTION

A program to be executed by a processor consists of a set of instructions stored in memory. In its simplest form, instruction processing consists of two steps: The processor reads (*fetches*) instructions from memory one at a time and executes each instruction. Program execution consists of repeating the process of instruction fetch and instruction execution. Instruction execution may involve several operations and depends on the nature of the instruction.

The processing required for a single instruction is called an *instruction cycle*. Using a simplified two-step description, the instruction cycle is depicted in Figure 1.2. The two steps are referred to as the *fetch stage* and the *execute stage*. Program execution halts only if the processor is turned off, some sort of unrecoverable error occurs, or a program instruction that halts the processor is encountered.

At the beginning of each instruction cycle, the processor fetches an instruction from memory. Typically, the program counter (PC) holds the address of the next instruction to be fetched. Unless instructed otherwise, the processor always increments the PC after each instruction fetch so that it will fetch the next instruction in sequence (i.e., the instruction located at the next higher memory address). For example, consider a simplified computer in which each instruction occupies one 16-bit word of memory. Assume that the program counter is set to location 300. The processor will next fetch the instruction at location 300. On succeeding instruction cycles, it will fetch instructions from locations 301, 302, 303, and so on. This sequence may be altered, as explained subsequently.

The fetched instruction is loaded into the instruction register (IR). The instruction contains bits that specify the action the processor is to take. The processor interprets the instruction and performs the required action. In general, these actions fall into four categories:

- **Processor-memory:** Data may be transferred from processor to memory or from memory to processor.
- **Processor-I/O:** Data may be transferred to or from a peripheral device by transferring between the processor and an I/O module.
- **Data processing:** The processor may perform some arithmetic or logic operation on data.
- **Control:** An instruction may specify that the sequence of execution be altered. For example, the processor may fetch an instruction from location 149, which

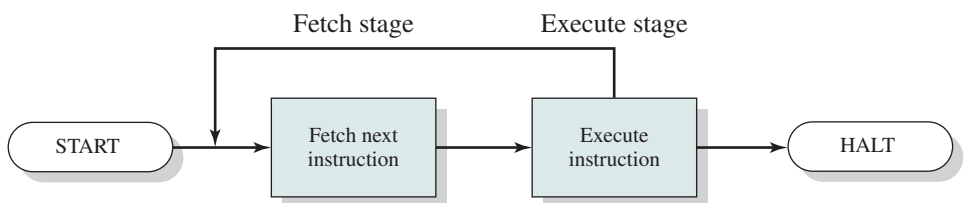


Figure 1.2 Basic Instruction Cycle

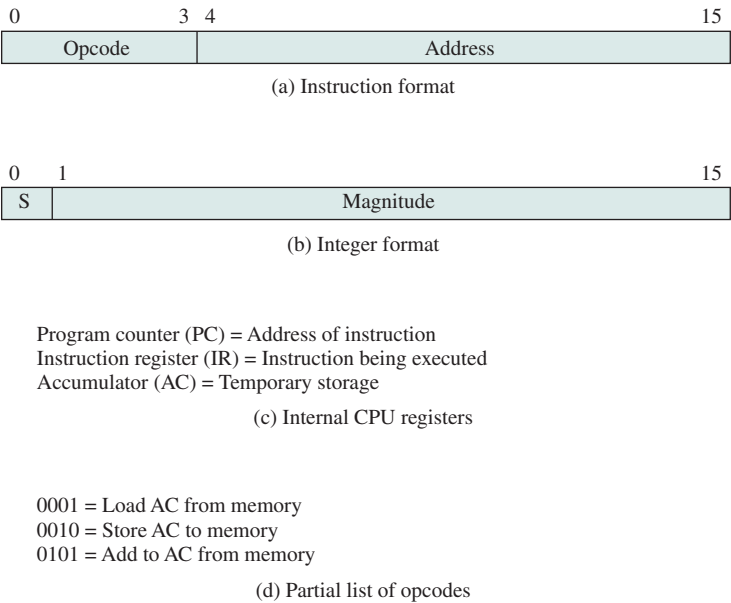


Figure 1.3 Characteristics of a Hypothetical Machine

specifies that the next instruction will be from location 182. The processor sets the program counter to 182. Thus, on the next fetch stage, the instruction will be fetched from location 182 rather than 150.

An instruction’s execution may involve a combination of these actions.

Consider a simple example using a hypothetical processor that includes the characteristics listed in Figure 1.3. The processor contains a single data register, called the accumulator (AC). Both instructions and data are 16 bits long, and memory is organized as a sequence of 16-bit words. The instruction format provides 4 bits for the opcode, allowing as many as $2^4 = 16$ different opcodes (represented by a single hexadecimal¹ digit). The opcode defines the operation the processor is to perform. With the remaining 12 bits of the instruction format, up to $2^{12} = 4,096$ (4K) words of memory (denoted by three hexadecimal digits) can be directly addressed.

Figure 1.4 illustrates a partial program execution, showing the relevant portions of memory and processor registers. The program fragment shown adds the contents of the memory word at address 940 to the contents of the memory word at address 941 and stores the result in the latter location. Three instructions, which can be described as three fetch and three execute stages, are required:

1. The PC contains 300, the address of the first instruction. This instruction (the value 1940 in hexadecimal) is loaded into the IR and the PC is incremented. Note that this process involves the use of a memory address register (MAR)

¹A basic refresher on number systems (decimal, binary, hexadecimal) can be found at the Computer Science Student Resource Site at ComputerScienceStudent.com.

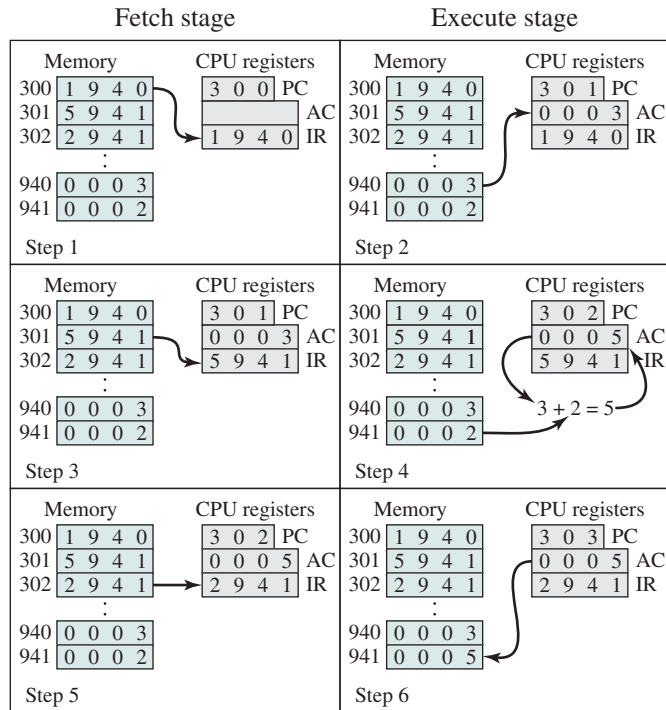


Figure 1.4 Example of Program Execution (contents of memory and registers in hexadecimal)

and a memory buffer register (MBR). For simplicity, these intermediate registers are not shown.

2. The first 4 bits (first hexadecimal digit) in the IR indicate that the AC is to be loaded from memory. The remaining 12 bits (three hexadecimal digits) specify the address, which is 940.
3. The next instruction (5941) is fetched from location 301 and the PC is incremented.
4. The old contents of the AC and the contents of location 941 are added and the result is stored in the AC.
5. The next instruction (2941) is fetched from location 302 and the PC is incremented.
6. The contents of the AC are stored in location 941.

In this example, three instruction cycles, each consisting of a fetch stage and an execute stage, are needed to add the contents of location 940 to the contents of 941. With a more complex set of instructions, fewer instruction cycles would be needed. Most modern processors include instructions that contain more than one address. Thus the execution stage for a particular instruction may involve more than one reference to memory. Also, instead of memory references, an instruction may specify an I/O operation.

1.4 INTERRUPTS

Virtually all computers provide a mechanism by which other modules (I/O, memory) may interrupt the normal sequencing of the processor. Table 1.1 lists the most common classes of interrupts.

Interrupts are provided primarily as a way to improve processor utilization. For example, most I/O devices are much slower than the processor. Suppose that the processor is transferring data to a printer using the instruction cycle scheme of Figure 1.2. After each write operation, the processor must pause and remain idle until the printer catches up. The length of this pause may be on the order of many thousands or even millions of instruction cycles. Clearly, this is a very wasteful use of the processor.

To give a specific example, consider a PC that operates at 1 GHz, which would allow roughly 10^9 instructions per second.² A typical hard disk has a rotational speed of 7200 revolutions per minute for a half-track rotation time of 4 ms, which is 4 million times slower than the processor.

Figure 1.5a illustrates this state of affairs. The user program performs a series of WRITE calls interleaved with processing. The solid vertical lines represent segments of code in a program. Code segments 1, 2, and 3 refer to sequences of instructions that do not involve I/O. The WRITE calls are to an I/O routine that is a system utility and that will perform the actual I/O operation. The I/O program consists of three sections:

- A sequence of instructions, labeled 4 in the figure, to prepare for the actual I/O operation. This may include copying the data to be output into a special buffer and preparing the parameters for a device command.
- The actual I/O command. Without the use of interrupts, once this command is issued, the program must wait for the I/O device to perform the requested function (or periodically check the status, or poll, the I/O device). The program might wait by simply repeatedly performing a test operation to determine if the I/O operation is done.
- A sequence of instructions, labeled 5 in the figure, to complete the operation. This may include setting a flag indicating the success or failure of the operation.

Table 1.1 Classes of Interrupts

Program	Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, attempt to execute an illegal machine instruction, and reference outside a user's allowed memory space.
Timer	Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis.
I/O	Generated by an I/O controller, to signal normal completion of an operation or to signal a variety of error conditions.
Hardware failure	Generated by a failure, such as power failure or memory parity error.

²A discussion of the uses of numerical prefixes, such as giga and tera, is contained in a supporting document at the Computer Science Student Resource Site at ComputerScienceStudent.com.

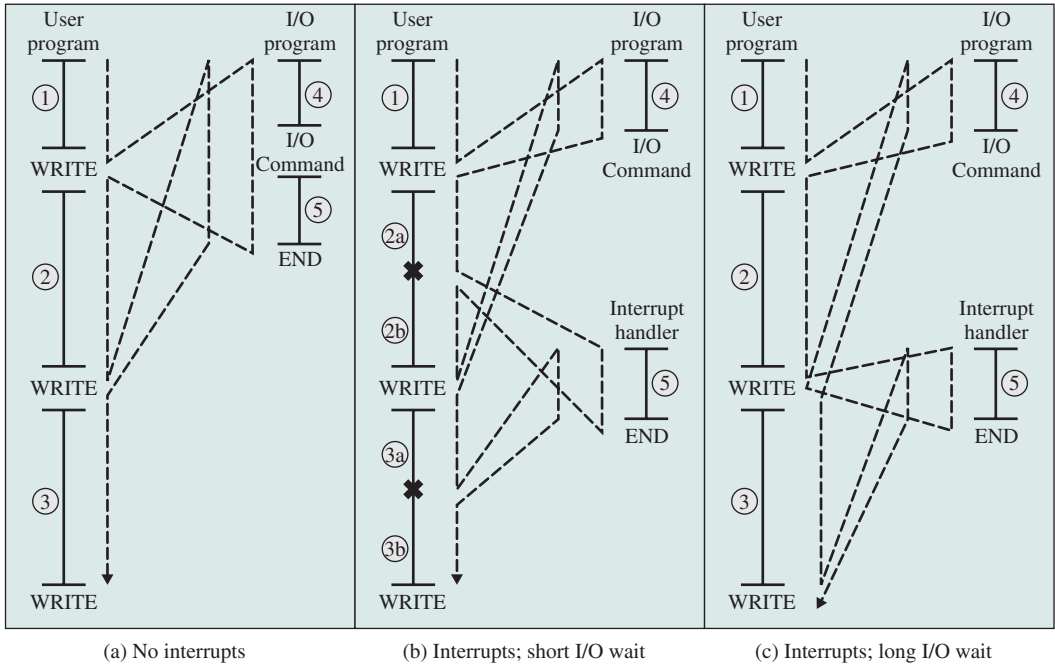


Figure 1.5 Program Flow of Control without and with Interrupts

The dashed line represents the path of execution followed by the processor; that is, this line shows the sequence in which instructions are executed. Thus, after the first WRITE instruction is encountered, the user program is interrupted and execution continues with the I/O program. After the I/O program execution is complete, execution resumes in the user program immediately following the WRITE instruction.

Because the I/O operation may take a relatively long time to complete, the I/O program is hung up waiting for the operation to complete; hence, the user program is stopped at the point of the WRITE call for some considerable period of time.

Interrupts and the Instruction Cycle

With interrupts, the processor can be engaged in executing other instructions while an I/O operation is in progress. Consider the flow of control in Figure 1.5b. As before, the user program reaches a point at which it makes a system call in the form of a WRITE call. The I/O program that is invoked in this case consists only of the preparation code and the actual I/O command. After these few instructions have been executed, control returns to the user program. Meanwhile, the external device is busy accepting data from computer memory and printing it. This I/O operation is conducted concurrently with the execution of instructions in the user program.

When the external device becomes ready to be serviced, that is, when it is ready to accept more data from the processor, the I/O module for that external device sends an *interrupt request* signal to the processor. The processor responds by suspending operation of the current program; branching off to a routine to service

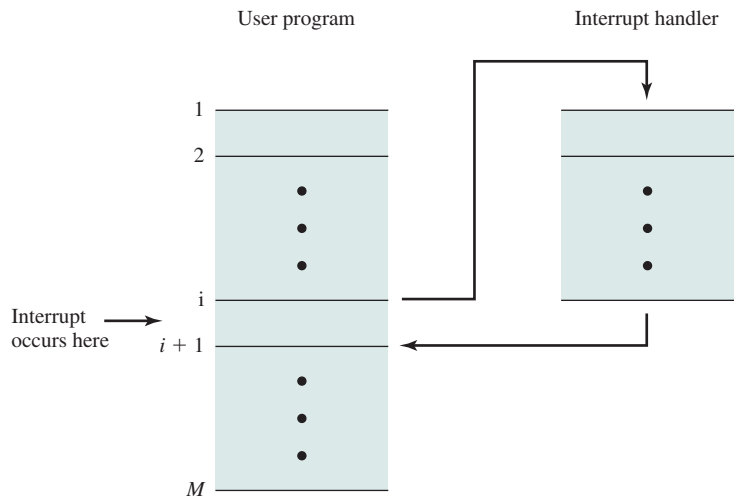


Figure 1.6 Transfer of Control via Interrupts

that particular I/O device, known as an interrupt handler; and resuming the original execution after the device is serviced. The points at which such interrupts occur are indicated by ✕ in Figure 1.5b. Note that an interrupt can occur at any point in the main program, not just at one specific instruction.

For the user program, an interrupt suspends the normal sequence of execution. When the interrupt processing is completed, execution resumes (Figure 1.6). Thus, the user program does not have to contain any special code to accommodate interrupts; the processor and the OS are responsible for suspending the user program and then resuming it at the same point.

To accommodate interrupts, an *interrupt stage* is added to the instruction cycle, as shown in Figure 1.7 (compare Figure 1.2). In the interrupt stage, the processor checks to see if any interrupts have occurred, indicated by the presence of an interrupt signal. If no interrupts are pending, the processor proceeds to the fetch stage and fetches the next instruction of the current program. If an interrupt is pending,

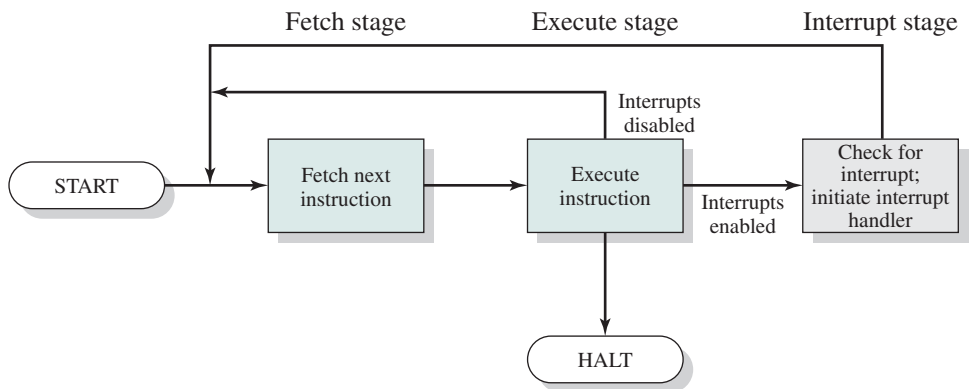


Figure 1.7 Instruction Cycle with Interrupts

the processor suspends execution of the current program and executes an *interrupt-handler* routine. The interrupt-handler routine is generally part of the OS. Typically, this routine determines the nature of the interrupt and performs whatever actions are needed. In the example we have been using, the handler determines which I/O module generated the interrupt and may branch to a program that will write more data out to that I/O module. When the interrupt-handler routine is completed, the processor can resume execution of the user program at the point of interruption.

It is clear that there is some overhead involved in this process. Extra instructions must be executed (in the interrupt handler) to determine the nature of the interrupt and to decide on the appropriate action. Nevertheless, because of the relatively large amount of time that would be wasted by simply waiting on an I/O operation, the processor can be employed much more efficiently with the use of interrupts.

To appreciate the gain in efficiency, consider Figure 1.8, which is a timing diagram based on the flow of control in Figures 1.5a and 1.5b. Figures 1.5b and 1.8

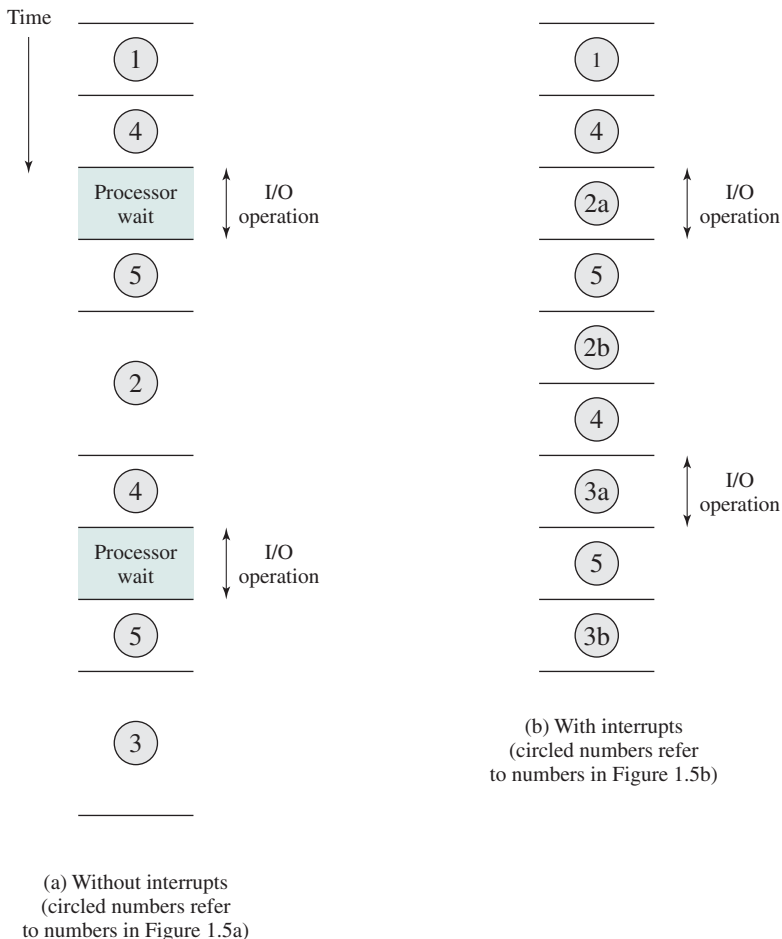


Figure 1.8 Program Timing: Short I/O Wait

assume that the time required for the I/O operation is relatively short: less than the time to complete the execution of instructions between write operations in the user program. The more typical case, especially for a slow device such as a printer, is that the I/O operation will take much more time than executing a sequence of user instructions. Figure 1.5c indicates this state of affairs. In this case, the user program reaches the second WRITE call before the I/O operation spawned by the first call is complete. The result is that the user program is hung up at that point. When the preceding I/O operation is completed, this new WRITE call may be processed, and a new I/O operation may be started. Figure 1.9 shows the timing for this situation with and without the use of interrupts. We can see that there is still a gain in efficiency because part of the time during which the I/O operation is underway overlaps with the execution of user instructions.

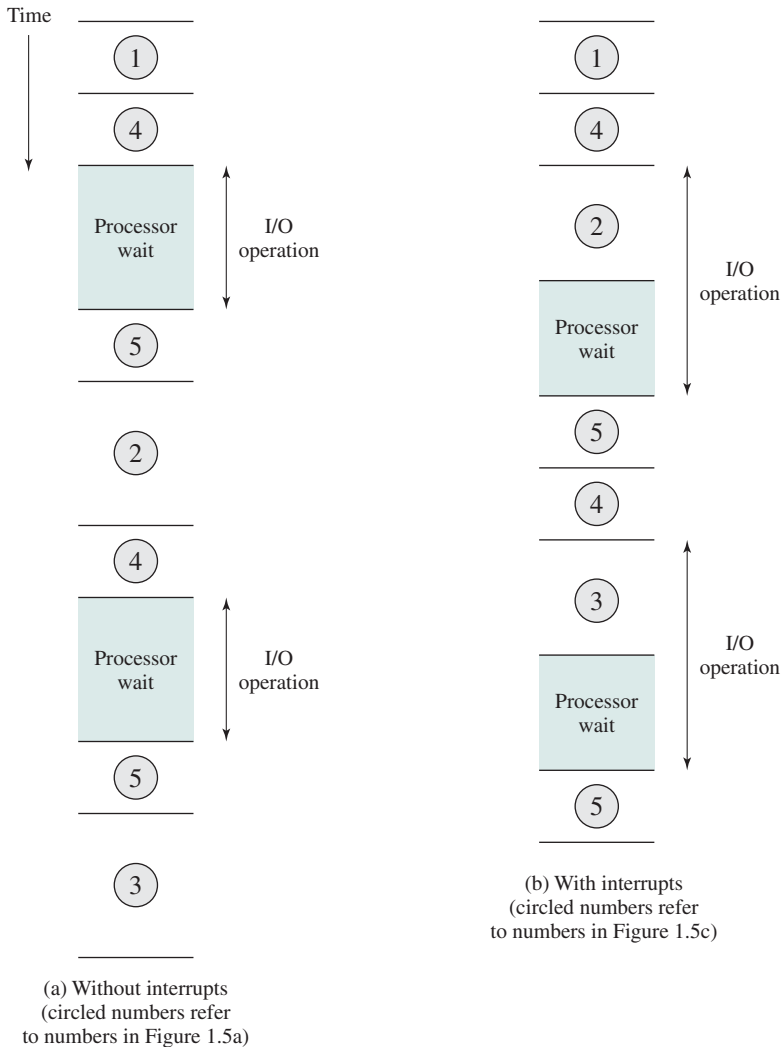


Figure 1.9 Program Timing: Long I/O Wait

Interrupt Processing

An interrupt triggers a number of events, both in the processor hardware and in software. Figure 1.10 shows a typical sequence. When an I/O device completes an I/O operation, the following sequence of hardware events occurs:

1. The device issues an interrupt signal to the processor.
2. The processor finishes execution of the current instruction before responding to the interrupt, as indicated in Figure 1.7.
3. The processor tests for a pending interrupt request, determines that there is one, and sends an acknowledgment signal to the device that issued the interrupt. The acknowledgment allows the device to remove its interrupt signal.
4. The processor next needs to prepare to transfer control to the interrupt routine. To begin, it saves information needed to resume the current program at the point of interrupt. The minimum information required is the program status word³ (PSW) and the location of the next instruction to be executed, which

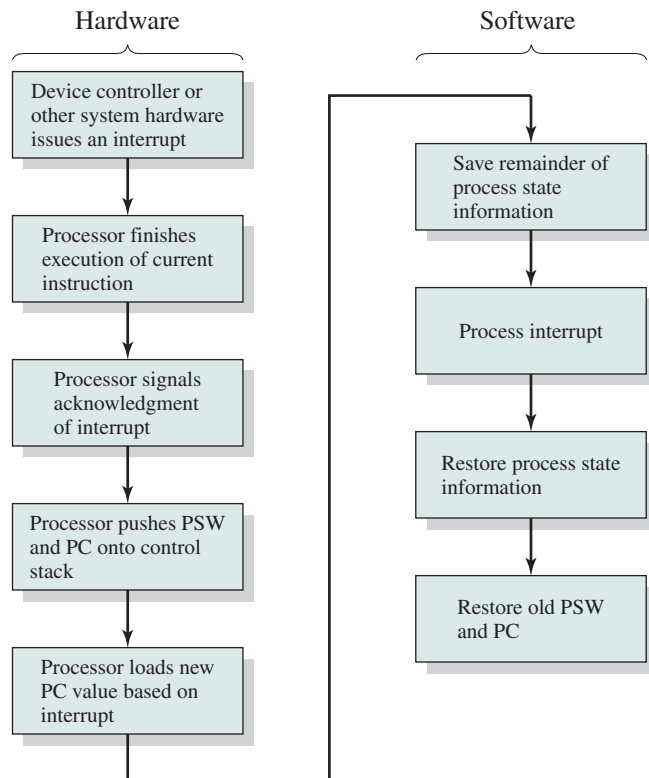


Figure 1.10 Simple Interrupt Processing

³The PSW contains status information about the currently running process, including memory usage information, condition codes, and other status information, such as an interrupt enable/disable bit and a kernel/user mode bit. See Appendix C for further discussion.

is contained in the program counter (PC). These can be pushed onto a control stack (see Appendix P).

5. The processor then loads the program counter with the entry location of the interrupt-handling routine that will respond to this interrupt. Depending on the computer architecture and OS design, there may be a single program, one for each type of interrupt, or one for each device and each type of interrupt. If there is more than one interrupt-handling routine, the processor must determine which one to invoke. This information may have been included in the original interrupt signal, or the processor may have to issue a request to the device that issued the interrupt to get a response that contains the needed information.

Once the program counter has been loaded, the processor proceeds to the next instruction cycle, which begins with an instruction fetch. Because the instruction fetch is determined by the contents of the program counter, control is transferred to the interrupt-handler program. The execution of this program results in the following operations:

6. At this point, the program counter and PSW relating to the interrupted program have been saved on the control stack. However, there is other information that is considered part of the state of the executing program. In particular, the contents of the processor registers need to be saved, because these registers may be used by the interrupt handler. So all of these values, plus any other state information, need to be saved. Typically, the interrupt handler will begin by saving the contents of all registers on the stack. Other state information that must be saved is discussed in Chapter 3. Figure 1.11a shows a simple example. In this case, a user program is interrupted after the instruction at location N . The contents of all of the registers plus the address of the next instruction ($N + 1$), a total of M words, are pushed onto the control stack. The stack pointer is updated to point to the new top of stack, and the program counter is updated to point to the beginning of the interrupt service routine.
7. The interrupt handler may now proceed to process the interrupt. This includes an examination of status information relating to the I/O operation or other event that caused an interrupt. It may also involve sending additional commands or acknowledgments to the I/O device.
8. When interrupt processing is complete, the saved register values are retrieved from the stack and restored to the registers (e.g., see Figure 1.11b).
9. The final act is to restore the PSW and program counter values from the stack. As a result, the next instruction to be executed will be from the previously interrupted program.

It is important to save all of the state information about the interrupted program for later resumption. This is because the interrupt is not a routine called from the program. Rather, the interrupt can occur at any time and therefore at any point in the execution of a user program. Its occurrence is unpredictable.

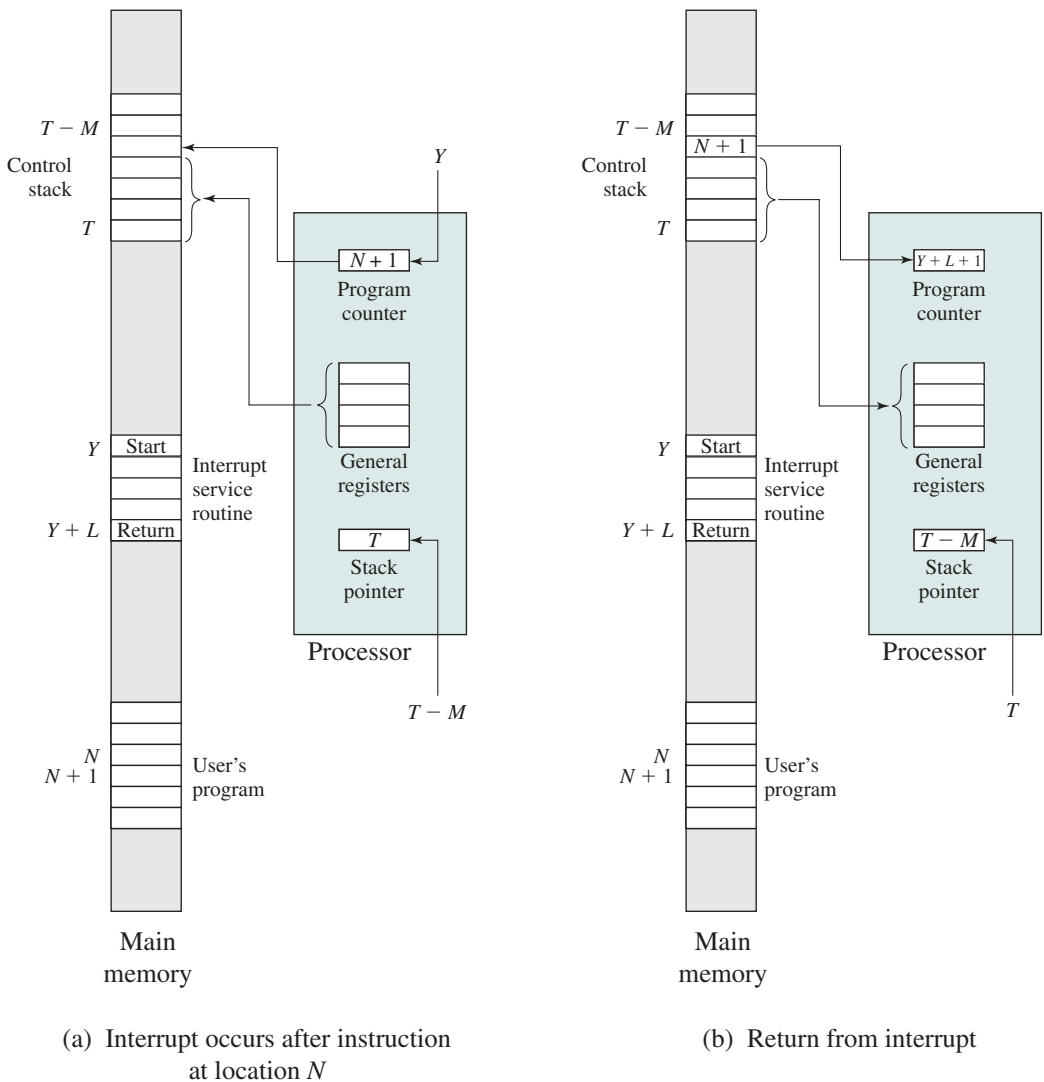
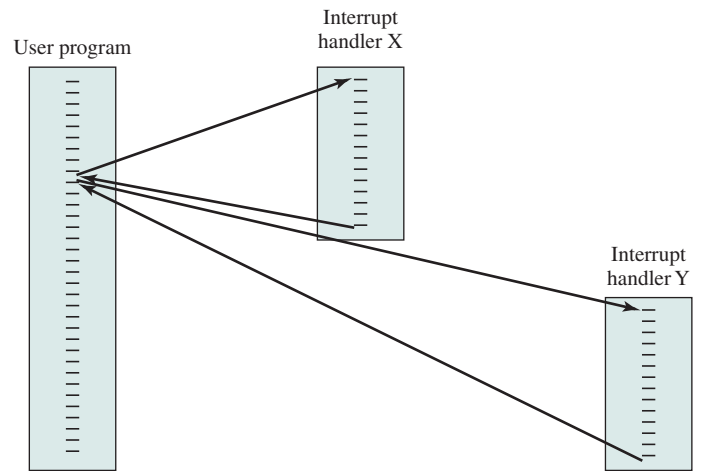


Figure 1.11 Changes in Memory and Registers for an Interrupt

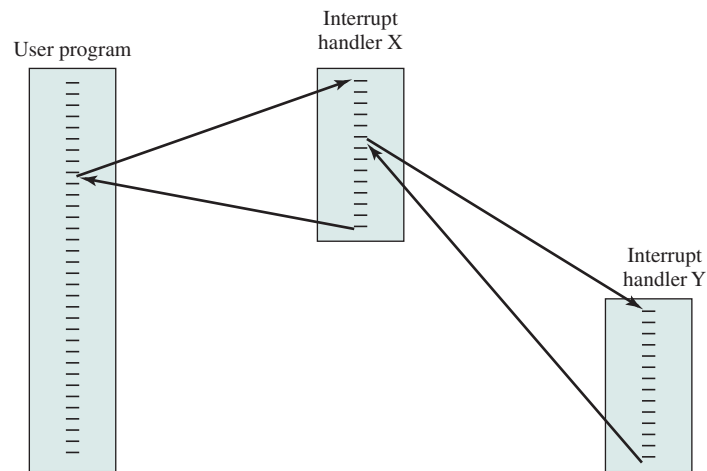
Multiple Interrupts

So far, we have discussed the occurrence of a single interrupt. Suppose, however, that one or more interrupts can occur while an interrupt is being processed. For example, a program may be receiving data from a communications line and printing results at the same time. The printer will generate an interrupt every time that it completes a print operation. The communication line controller will generate an interrupt every time a unit of data arrives. The unit could either be a single character or a block, depending on the nature of the communications discipline. In any case, it is possible for a communications interrupt to occur while a printer interrupt is being processed.

Two approaches can be taken to dealing with multiple interrupts. The first is to disable interrupts while an interrupt is being processed. A *disabled interrupt* simply means that the processor ignores any new interrupt request signal. If an interrupt occurs during this time, it generally remains pending and will be checked by the processor after the processor has reenabled interrupts. Thus, if an interrupt occurs when a user program is executing, then interrupts are disabled immediately. After the interrupt-handler routine completes, interrupts are reenabled before resuming the user program, and the processor checks to see if additional interrupts have occurred. This approach is simple, as interrupts are handled in strict sequential order (Figure 1.12a).



(a) Sequential interrupt processing



(b) Nested interrupt processing

Figure 1.12 Transfer of Control with Multiple Interrupts

The drawback to the preceding approach is that it does not take into account relative priority or time-critical needs. For example, when input arrives from the communications line, it may need to be absorbed rapidly to make room for more input. If the first batch of input has not been processed before the second batch arrives, data may be lost because the buffer on the I/O device may fill and overflow.

A second approach is to define priorities for interrupts and to allow an interrupt of higher priority to cause a lower-priority interrupt handler to be interrupted (Figure 1.12b). As an example of this second approach, consider a system with three I/O devices: a printer, a disk, and a communications line, with increasing priorities of 2, 4, and 5, respectively. Figure 1.13, based on an example in [TANE06], illustrates a possible sequence. A user program begins at $t = 0$. At $t = 10$, a printer interrupt occurs; user information is placed on the control stack and execution continues at the printer interrupt service routine (ISR). While this routine is still executing, at $t = 15$ a communications interrupt occurs. Because the communications line has higher priority than the printer, the interrupt request is honored. The printer ISR is interrupted, its state is pushed onto the stack, and execution continues at the communications ISR. While this routine is executing, a disk interrupt occurs ($t = 20$). Because this interrupt is of lower priority, it is simply held, and the communications ISR runs to completion. When the communications ISR is complete ($t = 25$), the previous processor state is restored, which is the execution of the printer ISR. However, before even a single instruction in that routine can be executed, the processor honors the higher-priority disk interrupt and transfers control to the disk ISR. Only when that routine is complete ($t = 35$) is the printer ISR resumed. When that routine completes ($t = 40$), control finally returns to the user program.

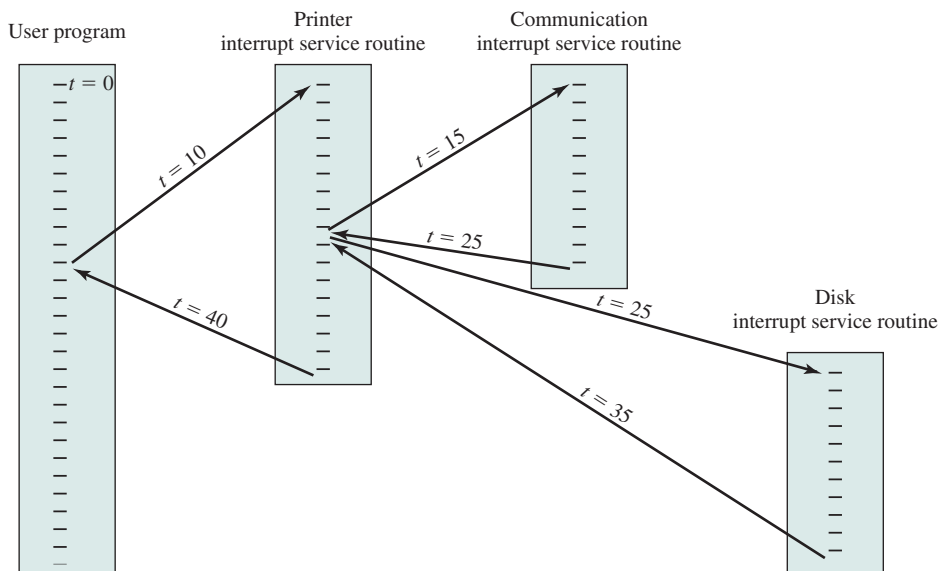


Figure 1.13 Example Time Sequence of Multiple Interrupts