CHAPTER 2

Data Manipulation

In this chapter we will learn how a computer manipulates data and communicates with peripheral devices such as printers and keyboards. In doing so, we will explore the basics of computer architecture and learn how computers are programmed by means of encoded instructions, called machine language instructions.

**Asterisks indicate suggestions for optional sections.*

In Chapter 1 we studied topics relating to the storage of data inside a computer. In this chapter we will see how a computer manipulates that data. This manipulation consists of moving data from one location to another as well as performing operations such as arithmetic calculations, text editing, and image manipulation. We begin by extending our understanding of computer architecture beyond that of data storage systems.

2.1 Computer Architecture

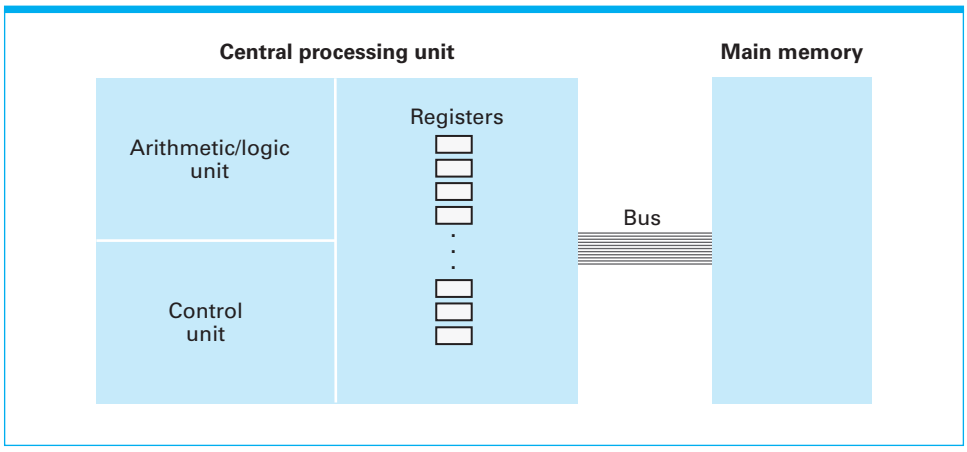
The circuitry in a computer that controls the manipulation of data is called the **central processing unit,** or **CPU** (often referred to as merely the processor). In the machines of the mid-twentieth century, CPUs were large units comprised of perhaps several racks of electronic circuitry that reflected the significance of the unit. However, technology has shrunk these devices drastically. The CPUs found in today's desktop computers and notebooks are packaged as small flat squares (approximately two inches by two inches) whose connecting pins plug into a socket mounted on the machine's main circuit board (called the **motherboard**). In smartphones, mini-notebooks, and other **Mobile Internet Devices (MID),** CPUs are around half the size of a postage stamp. Due to their small size, these processors are called **microprocessors.**

CPU Basics

A CPU consists of three parts (Figure 2.1): the **arithmetic/logic unit,** which contains the circuitry that performs operations on data (such as addition and subtraction); the **control unit,** which contains the circuitry for coordinating the machine's activities; and the **register unit,** which contains data storage cells (similar to main memory cells), called **registers,** that are used for temporary storage of information within the CPU.

Some of the registers within the register unit are considered **general-purpose registers,** whereas others are **special-purpose registers.** We will discuss some of the special-purpose registers in Section 2.3. For now, we are concerned only with the general-purpose registers.

Figure 2.1 CPU and main memory connected via a bus



General-purpose registers serve as temporary holding places for data being manipulated by the CPU. These registers hold the inputs to the arithmetic/logic unit's circuitry and provide storage space for results produced by that unit. To perform an operation on data stored in main memory, the control unit transfers the data from memory into the general-purpose registers, informs the arithmetic/logic unit which registers hold the data, activates the appropriate circuitry within the arithmetic/logic unit, and tells the arithmetic/logic unit which register should receive the result.

For the purpose of transferring bit patterns, a machine's CPU and main memory are connected by a collection of wires called a **bus** (see again Figure 2.1). Through this bus, the CPU extracts (reads) data from main memory by supplying the address of the pertinent memory cell along with an electronic signal telling the memory circuitry that it is supposed to retrieve the data in the indicated cell. In a similar manner, the CPU places (writes) data in memory by providing the address of the destination cell and the data to be stored together with the appropriate electronic signal telling main memory that it is supposed to store the data being sent to it.

Based on this design, the task of adding two values stored in main memory involves more than the mere execution of the addition operation. The data must be transferred from main memory to registers within the CPU, the values must be added with the result being placed in a register, and the result must then be stored in a memory cell. The entire process is summarized by the five steps listed in Figure 2.2.

The Stored-Program Concept

Early computers were not known for their flexibility—the steps that each device executed were built into the control unit as a part of the machine. To gain more flexibility, some of the early electronic computers were designed so that the CPU could be conveniently rewired. This flexibility was accomplished by means of a pegboard arrangement similar to old telephone switchboards in which the ends of jumper wires were plugged into holes.

Figure 2.2 Adding values stored in memory

Step 1. Get one of the values to be added from memory and place it in a register.

Step 2. Get the other value to be added from memory and place it in another register.

Step 3. Activate the addition circuitry with the registers used in Steps 1 and 2 as inputs and another register designated to hold the result.

Step 4. Store the result in memory.

Step 5. Stop.

Cache Memory

It is instructive to compare the memory facilities within a computer in relation to their functionality. Registers are used to hold the data immediately applicable to the operation at hand; main memory is used to hold data that will be needed in the near future; and mass storage is used to hold data that will likely not be needed in the immediate future. Many machines are designed with an additional memory level, called cache memory. **Cache memory** is a portion (perhaps several hundred KB) of high-speed memory located within the CPU itself. In this special memory area, the machine attempts to keep a copy of that portion of main memory that is of current interest. In this setting, data transfers that normally would be made between registers and main memory are made between registers and cache memory. Any changes made to cache memory are then transferred collectively to main memory at a more opportune time. The result is a CPU that can execute its machine cycle more rapidly because it is not delayed by main memory communication.

A breakthrough (credited, apparently incorrectly, to John von Neumann) came with the realization that a program, just like data, can be encoded and stored in main memory. If the control unit is designed to extract the program from memory, decode the instructions, and execute them, the program that the machine follows can be changed merely by changing the contents of the computer's memory instead of rewiring the CPU.

The idea of storing a computer's program in its main memory is called the **stored-program concept** and has become the standard approach used today—so standard, in fact, that it seems obvious. What made it difficult originally was that everyone thought of programs and data as different entities: Data were stored in memory; programs were part of the CPU. The result was a prime example of not seeing the forest for the trees. It is easy to be caught in such ruts, and the development of computer science might still be in many of them today without our knowing it. Indeed, part of the excitement of the science is that new insights are constantly opening doors to new theories and applications.

Questions & Exercises

1. What sequence of events do you think would be required to move the contents of one memory cell in a computer to another memory cell?
2. What information must the CPU supply to the main memory circuitry to write a value into a memory cell?
3. Mass storage, main memory, and general-purpose registers are all storage systems. What is the difference in their use?

2.2 Machine Language

To apply the stored-program concept, CPUs are designed to recognize instructions encoded as bit patterns. This collection of instructions along with the encoding system is called the **machine language.** An instruction expressed in this language is called a machine-level instruction or, more commonly, a **machine instruction.**

The Instruction Repertoire

The list of machine instructions that a typical CPU must be able to decode and execute is quite short. In fact, once a machine can perform certain elementary but well-chosen tasks, adding more features does not increase the machine's theoretical capabilities. In other words, beyond a certain point, additional features may increase such things as convenience but add nothing to the machine's fundamental capabilities.

The degree to which machine designs should take advantage of this fact has led to two philosophies of CPU architecture. One is that a CPU should be designed to execute a minimal set of machine instructions. This approach leads to what is called a **reduced instruction set computer (RISC).** The argument in favor of RISC architecture is that such a machine is efficient, fast, and less expensive to manufacture. On the other hand, others argue in favor of CPUs with the ability to execute a large number of complex instructions, even though many of them are technically redundant. The result of this approach is known as a **complex instruction set computer (CISC).** The argument in favor of CISC architecture is that the more complex CPU can better cope with the ever-increasing complexities

Who Invented What?

Awarding a single individual credit for an invention is always a dubious undertaking. Thomas Edison is credited with inventing the incandescent lamp, but other researchers were developing similar lamps, and in a sense Edison was lucky to be the one to obtain the patent. The Wright brothers are credited with inventing the airplane, but they were competing with and benefited from the work of many contemporaries, all of whom were preempted to some degree by Leonardo da Vinci, who toyed with the idea of flying machines in the fifteenth century. Even Leonardo's designs were apparently based on earlier ideas. Of course, in these cases the designated inventor still has legitimate claims to the credit bestowed. In other cases, history seems to have awarded credit inappropriately—an example is the stored-program concept. Without a doubt, John von Neumann was a brilliant scientist who deserves credit for numerous contributions. But one of the contributions for which popular history has chosen to credit him, the stored-program concept, was apparently developed by researchers led by J. P. Eckert at the Moore School of Electrical Engineering at the University of Pennsylvania. John von Neumann was merely the first to publish work reporting the idea and thus computing lore has selected him as the inventor.

of today's software. With CISC, programs can exploit a powerful rich set of instructions, many of which would require a multi-instruction sequence in a RISC design.

In the 1990s and into the millennium, commercially available CISC and RISC processors were actively competing for dominance in desktop computing. Intel processors, used in PCs, are examples of CISC architecture; PowerPC processors (developed by an alliance between Apple, IBM, and Motorola) are examples of RISC architecture and were used in the Apple Macintosh. As time progressed, the manufacturing cost of CISC was drastically reduced; thus Intel's processors (or their equivalent from AMD—Advanced Micro Devices, Inc.) are now found in virtually all desktop and laptop computers (even Apple is now building computers based on Intel products).

While CISC secured its place in desktop computers, it has an insatiable thirst for electrical power. In contrast, the company Advanced RISC Machine (ARM) has designed a RISC architecture specifically for low power consumption. (Advanced RISC Machine was originally Acorn Computers and is now ARM Holdings.) Thus, ARM-based processors, manufactured by a host of vendors including Qualcomm and Texas Instruments, are readily found in game controllers, digital TVs, navigation systems, automotive modules, cellular telephones, smartphones, and other consumer electronics.

Regardless of the choice between RISC and CISC, a machine's instructions can be categorized into three groupings: (1) the data transfer group, (2) the arithmetic/logic group, and (3) the control group.

Data Transfer The data transfer group consists of instructions that request the movement of data from one location to another. Steps 1, 2, and 4 in Figure 2.2 fall into this category. We should note that using terms such as *transfer* or *move* to identify this group of instructions is actually a misnomer. It is rare that the data being transferred is erased from its original location. The process involved in a transfer instruction is more like copying the data rather than moving it. Thus terms such as *copy* or *clone* better describe the actions of this group of instructions.

While on the subject of terminology, we should mention that special terms are used when referring to the transfer of data between the CPU and main memory. A request to fill a general-purpose register with the contents of a memory cell is

Variable-Length Instructions

To simplify explanations in the text, the machine language used for examples in this chapter (and described in Appendix C) uses a fixed size (two bytes) for all instructions. Thus, to fetch an instruction, the CPU always retrieves the contents of two consecutive memory cells and increments its program counter by two. This consistency streamlines the task of fetching instructions and is characteristic of RISC machines. CISC machines, however, have machine languages whose instructions vary in length. Today's Intel processors, for example, have instructions that range from single-byte instructions to multiple-byte instructions whose length depends on the exact use of the instruction. CPUs with such machine languages determine the length of the incoming instruction by the instruction's op-code. That is, the CPU first fetches the op-code of the instruction and then, based on the bit pattern received, knows how many more bytes to fetch from memory to obtain the rest of the instruction.

commonly referred to as a LOAD instruction; conversely, a request to transfer the contents of a register to a memory cell is called a STORE instruction. In Figure 2.2, Steps 1 and 2 are LOAD instructions, and Step 4 is a STORE instruction.

An important group of instructions within the data transfer category consists of the commands for communicating with devices outside the CPU-main memory context (printers, keyboards, display screens, disk drives, etc.). Since these instructions handle the input/output (I/O) activities of the machine, they are called **I/O instructions** and are sometimes considered as a category in their own right. On the other hand, Section 2.5 describes how these I/O activities can be handled by the same instructions that request data transfers between the CPU and main memory. Thus, we shall consider the I/O instructions to be a part of the data transfer group.

Arithmetic/Logic The arithmetic/logic group consists of the instructions that tell the control unit to request an activity within the arithmetic/logic unit. Step 3 in Figure 2.2 falls into this group. As its name suggests, the arithmetic/logic unit is capable of performing operations other than the basic arithmetic operations. Some of these additional operations are the Boolean operations AND, OR, and XOR, introduced in Chapter 1, which we will discuss in more detail later in this chapter.

Another collection of operations available within most arithmetic/logic units allows the contents of registers to be moved to the right or the left within the register. These operations are known as either SHIFT or ROTATE operations, depending on whether the bits that "fall off the end" of the register are merely discarded (SHIFT) or are used to fill the holes left at the other end (ROTATE).

Control The control group consists of those instructions that direct the execution of the program rather than the manipulation of data. Step 5 in Figure 2.2 falls into this category, although it is an extremely elementary example. This group contains many of the more interesting instructions in a machine's repertoire, such as the family of JUMP (or BRANCH) instructions used to direct the CPU to execute an instruction other than the next one in the list. These JUMP instructions appear in two varieties: **unconditional jumps** and **conditional jumps.** An example of the former would be the instruction "Skip to Step 5"; an example of the latter would be, "If the value obtained is 0, then skip to Step 5." The distinction is that a conditional jump results in a "change of venue" only if a certain condition is satisfied. As an example, the sequence of instructions in Figure 2.3 represents an algorithm for dividing two values where Step 3 is a conditional jump that protects against the possibility of division by zero.

An Illustrative Machine Language

Let us now consider how the instructions of a typical computer are encoded. The machine that we will use for our discussion is described in Appendix C and summarized in Figure 2.4. It has 16 general-purpose registers and 256 main memory cells, each with a capacity of 8 bits. For referencing purposes, we label the registers with the values 0 through 15 and address the memory cells with the values 0 through 255. For convenience we think of these labels and addresses as values represented in base two and compress the resulting bit patterns using hexadecimal notation. Thus, the registers are labeled 0 through F, and the memory cells are addressed 00 through FF.

Figure 2.3 Dividing values stored in memory

Step 1. LOAD a register with a value from memory.

Step 2. LOAD another register with another value from memory.

Step 3. If this second value is zero, JUMP to Step 6.

Step 4. Divide the contents of the first register by the second register and leave the result in a third register.

Step 5. STORE the contents of the third register in memory.

Step 6. STOP.

The encoded version of a machine instruction consists of two parts: the **op-code** (short for operation code) field and the **operand** field. The bit pattern appearing in the op-code field indicates which of the elementary operations, such as STORE, SHIFT, XOR, and JUMP, is requested by the instruction. The bit patterns found in the operand field provide more detailed information about the operation specified by the op-code. For example, in the case of a STORE operation, the information in the operand field indicates which register contains the data to be stored and which memory cell is to receive the data.

The entire machine language of our illustrative machine (Appendix C) consists of only twelve basic instructions. Each of these instructions is encoded using a total of 16 bits, represented by four hexadecimal digits (Figure 2.5). The op-code for each instruction consists of the first 4 bits or, equivalently, the first hexadecimal digit. Note (Appendix C) that these op-codes are represented by the hexadecimal digits 1 through C. In particular, the table in Appendix C

Figure 2.4 The architecture of the machine described in Appendix C

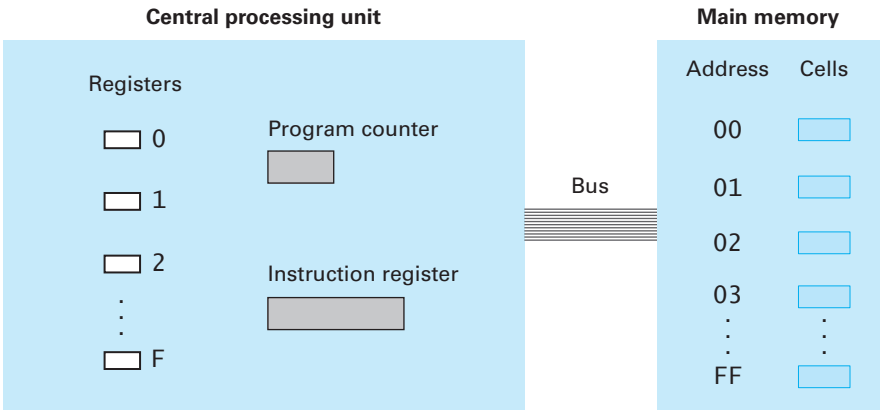
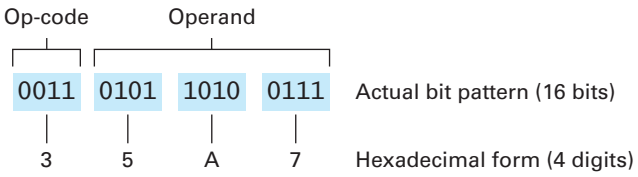


Figure 2.5 The composition of an instruction for the machine in Appendix C

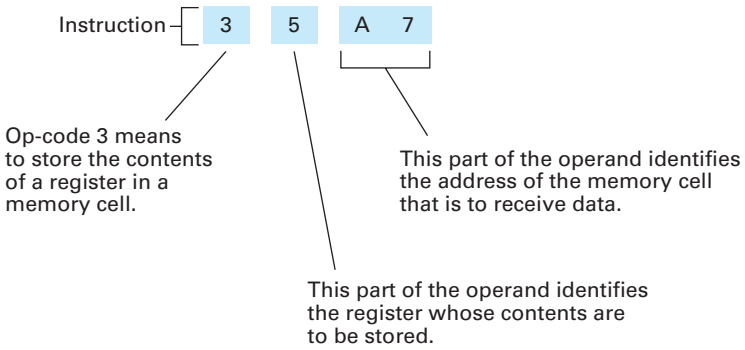


shows us that an instruction beginning with the hexadecimal digit 3 refers to a STORE instruction, and an instruction beginning with hexadecimal A refers to a ROTATE instruction.

The operand field of each instruction in our illustrative machine consists of three hexadecimal digits (12 bits), and in each case (except for the HALT instruction, which needs no further refinement) clarifies the general instruction given by the op-code. For example (Figure 2.6), if the first hexadecimal digit of an instruction were 3 (the op-code for storing the contents of a register), the next hexadecimal digit of the instruction would indicate which register is to be stored, and the last two hexadecimal digits would indicate which memory cell is to receive the data. Thus the instruction 35A7 (hexadecimal) translates to the statement "STORE the bit pattern found in register 5 in the memory cell whose address is A7." (Note how the use of hexadecimal notation simplifies our discussion. In reality, the instruction 35A7 is the bit pattern 0011010110100111.)

The instruction 35A7 also provides an explicit example of why main memory capacities are measured in powers of two. Because 8 bits in the instruction are reserved for specifying the memory cell utilized by this instruction, it is possible to reference exactly 2^8 different memory cells. It behooves us therefore to build main memory with this many cells—addressed from 0 to 255. If main memory had more cells, we would not be able to write instructions that distinguished between them; if main memory had fewer cells, we would be able to write instructions that referenced nonexisting cells.

Figure 2.6 Decoding the instruction 35A7



As another example of how the operand field is used to clarify the general instruction given by an op-code, consider an instruction with the op-code 7 (hexadecimal), which requests that the contents of two registers be ORed. (We will see what it means to OR two registers in Section 2.4. For now we are interested merely in how instructions are encoded.) In this case, the next hexadecimal digit indicates the register in which the result should be placed, while the last two hexadecimal digits indicate which two registers are to be ORed. Thus the instruction 70C5 translates to the statement "OR the contents of register C with the contents of register 5 and leave the result in register 0."

A subtle distinction exists between our machine's two LOAD instructions. Here we see that the op-code 1 (hexadecimal) identifies an instruction that loads a register with the contents of a memory cell, whereas the op-code 2 (hexadecimal) identifies an instruction that loads a register with a particular value. The difference is that the operand field in an instruction of the first type contains an address, whereas in the second type the operand field contains the actual bit pattern to be loaded.

Note that the machine has two ADD instructions: one for adding two's complement representations and one for adding floating-point representations. This distinction is a consequence of the fact that adding bit patterns that represent values encoded in two's complement notation requires different activities within the arithmetic/logic unit from adding values encoded in floating-point notation. We close this section with Figure 2.7, which contains an encoded version of the instructions in Figure 2.2. We have assumed that the values to be added are stored in two's complement notation at memory addresses 6C and 6D and the sum is to be placed in the memory cell at address 6E.

Figure 2.7 An encoded version of the instructions in Figure 2.2

Encoded instructions	Translation
156C	Load register 5 with the bit pattern found in the memory cell at address 6C.
166D	Load register 6 with the bit pattern found in the memory cell at address 6D.
5056	Add the contents of register 5 and 6 as though they were two's complement representation and leave the result in register 0.
306E	Store the contents of register 0 in the memory cell at address 6E.
C000	Halt.

Questions & Exercises

1. Why might the term *move* be considered an incorrect name for the operation of moving data from one location in a machine to another?
2. In the text, JUMP instructions were expressed by identifying the destination explicitly by stating the name (or step number) of the destination within the JUMP instruction (for example, "Jump to Step 6"). A drawback of this technique is that if an instruction name (number) is later changed, we must be sure to find all jumps to that instruction and change that name also. Describe another way of expressing a JUMP instruction so that the name of the destination is not explicitly stated.
3. Is the instruction "If 0 equals 0, then jump to Step 7" a conditional or unconditional jump? Explain your answer.
4. Write the example program in Figure 2.7 in actual bit patterns.
5. The following are instructions written in the machine language described in Appendix C. Rewrite them in English.

   a. 368A **b.** BADE **c.** 803C **d.** 40F4
6. What is the difference between the instructions 15AB and 25AB in the machine language of Appendix C?
7. Here are some instructions in English. Translate each of them into the machine language of Appendix C.

   a. LOAD register number 3 with the hexadecimal value 56.

   b. ROTATE register number 5 three bits to the right.

   c. AND the contents of register A with the contents of register 5 and leave the result in register 0.

2.3 Program Execution

A computer follows a program stored in its memory by copying the instructions from memory into the CPU as needed. Once in the CPU, each instruction is decoded and obeyed. The order in which the instructions are fetched from memory corresponds to the order in which the instructions are stored in memory unless otherwise altered by a JUMP instruction.

To understand how the overall execution process takes place, it is necessary to consider two of the special purpose registers within the CPU: the **instruction register** and the **program counter** (see again Figure 2.4). The instruction register is used to hold the instruction being executed. The program counter contains the address of the next instruction to be executed, thereby serving as the machine's way of keeping track of where it is in the program.

The CPU performs its job by continually repeating an algorithm that guides it through a three-step process known as the **machine cycle.** The steps in the

machine cycle are fetch, decode, and execute (Figure 2.8). During the fetch step, the CPU requests that main memory provide it with the instruction that is stored at the address indicated by the program counter. Since each instruction in our machine is two bytes long, this fetch process involves retrieving the contents of two memory cells from main memory. The CPU places the instruction received from memory in its instruction register and then increments the program counter by two so that the counter contains the address of the next instruction stored in memory. Thus the program counter will be ready for the next fetch.

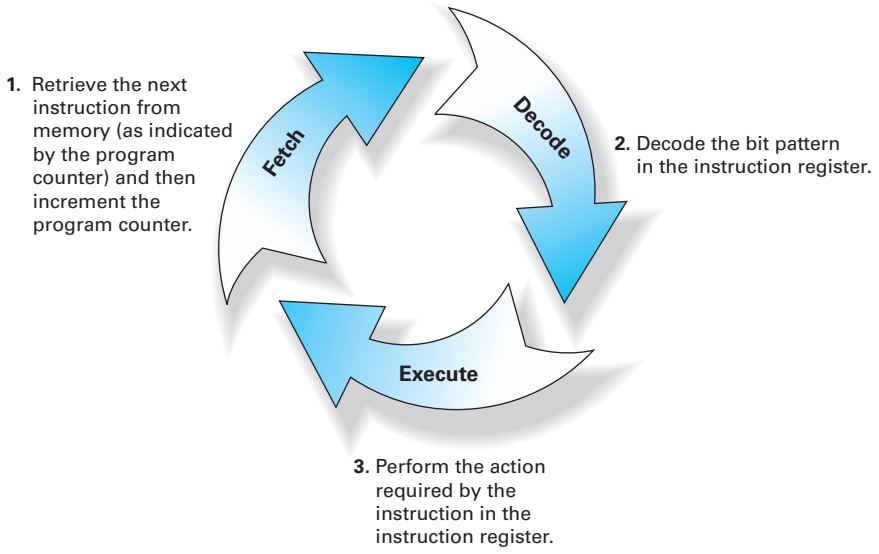
With the instruction now in the instruction register, the CPU decodes the instruction, which involves breaking the operand field into its proper components based on the instruction's op-code.

The CPU then executes the instruction by activating the appropriate circuitry to perform the requested task. For example, if the instruction is a load from memory, the CPU sends the appropriate signals to main memory, waits for main memory to send the data, and then places the data in the requested register. If the instruction is for an arithmetic operation, the CPU activates the appropriate circuitry in the arithmetic/logic unit with the correct registers as inputs and waits for the arithmetic/logic unit to compute the answer and place it in the appropriate register.

Once the instruction in the instruction register has been executed, the CPU again begins the machine cycle with the fetch step. Observe that since the program counter was incremented at the end of the previous fetch, it again provides the CPU with the correct address.

A somewhat special case is the execution of a JUMP instruction. Consider, for example, the instruction B258 (Figure 2.9), which means "JUMP to the instruction at address 58 (hexadecimal) if the contents of register 2 is the same as that of register 0." In this case, the execute step of the machine cycle begins with the comparison of registers 2 and 0. If they contain different bit patterns, the execute

Figure 2.8 The machine cycle



Comparing Computer Power

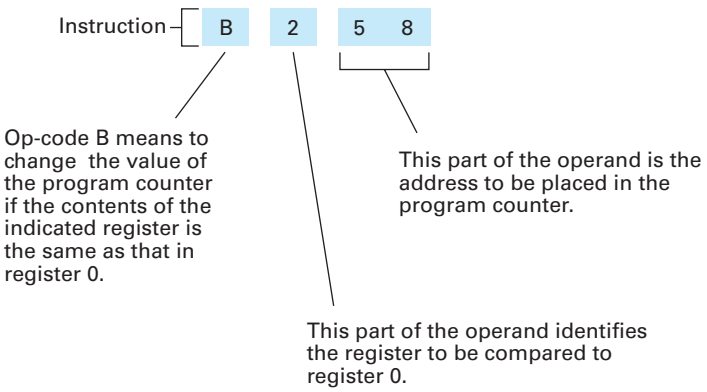
When shopping for a personal computer, you will find that clock speeds are often used to compare machines. A computer's **clock** is a circuit, called an oscillator, that generates pulses that are used to coordinate the machine's activities—the faster this oscillating circuit generates pulses, the faster the machine performs its machine cycle. Clock speeds are measured in hertz (abbreviated as Hz) with one Hz equal to one cycle (or pulse) per second. Typical clock speeds in desktop computers are in the range of a few hundred MHz (older models) to several GHz. (MHz is short for megahertz, which is a million Hz. GHz is short for gigahertz, which is 1000 MHz.)

Unfortunately, different CPU designs might perform different amounts of work in one clock cycle, and thus clock speed alone fails to be relevant in comparing machines with different CPUs. If you are comparing a machine based on an Intel processor to one based on ARM, it would be more meaningful to compare performance by means of **benchmarking,** which is the process of comparing the performance of different machines when executing the same program, known as a benchmark. By selecting benchmarks representing different types of applications, you get meaningful comparisons for various market segments.

step terminates and the next machine cycle begins. If, however, the contents of these registers are equal, the machine places the value 58 (hexadecimal) in its program counter during the execute step. In this case, then, the next fetch step finds 58 in the program counter, so the instruction at that address will be the next instruction to be fetched and executed.

Note that if the instruction had been B058, then the decision of whether the program counter should be changed would depend on whether the contents of register 0 was equal to that of register 0. But these are the same registers and thus must have equal content. In turn, any instruction of the form B0XY will cause a jump to be executed to the memory location XY regardless of the contents of register 0.

Figure 2.9 Decoding the instruction B258



An Example of Program Execution

Let us follow the machine cycle applied to the program presented in Figure 2.7, which retrieves two values from main memory, computes their sum, and stores that total in a main memory cell. We first need to put the program somewhere in memory. For our example, suppose the program is stored in consecutive addresses, starting at address A0 (hexadecimal). With the program stored in this manner, we can cause the machine to execute it by placing the address (A0) of the first instruction in the program counter and starting the machine (Figure 2.10).

The CPU begins the fetch step of the machine cycle by extracting the instruction stored in main memory at location A0 and placing this instruction (156C) in its instruction register (Figure 2.11a). Notice that, in our machine, instructions are 16 bits (two bytes) long. Thus the entire instruction to be fetched occupies the memory cells at both address A0 and A1. The CPU is designed to take this into account so it retrieves the contents of both cells and places the bit patterns received in the instruction register, which is 16 bits long. The CPU then adds two to the program counter so that this register contains the address of the next instruction (Figure 2.11b). At the end of the fetch step of the first machine cycle, the program counter and instruction register contain the following data:

```
Program Counter: A2
Instruction Register: 156C
```

Next, the CPU analyzes the instruction in its instruction register and concludes that it is to load register 5 with the contents of the memory cell at address 6C. This load activity is performed during the execution step of the machine cycle, and the CPU then begins the next cycle.

This cycle begins by fetching the instruction 166D from the two memory cells starting at address A2. The CPU places this instruction in the instruction register

Figure 2.10 The program from Figure 2.7 stored in main memory ready for execution

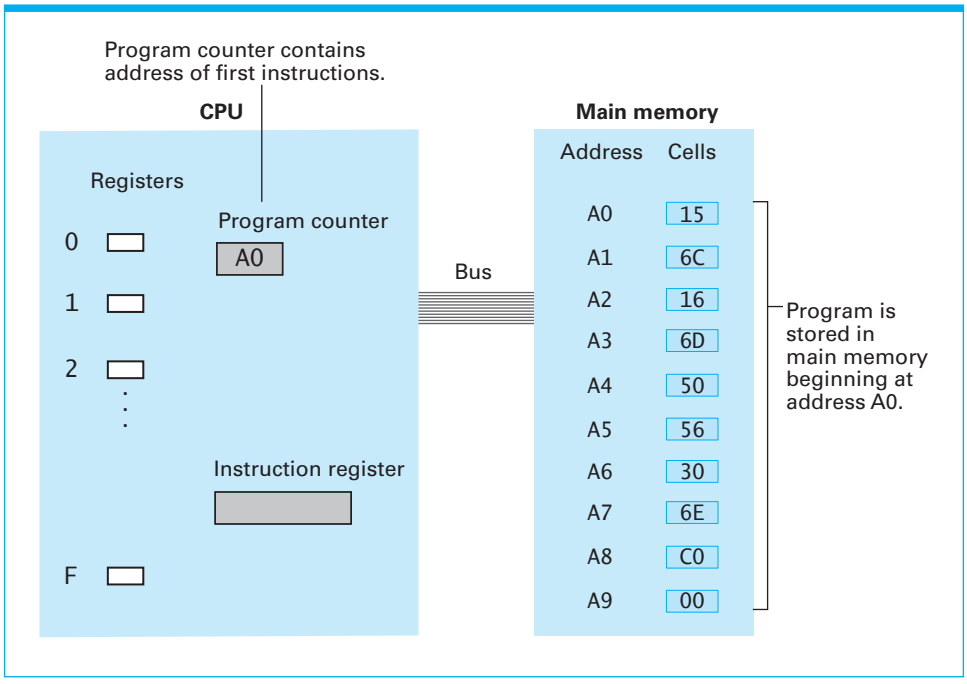
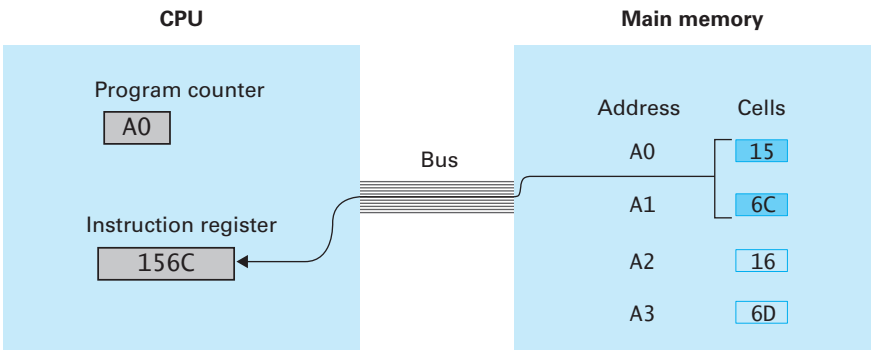
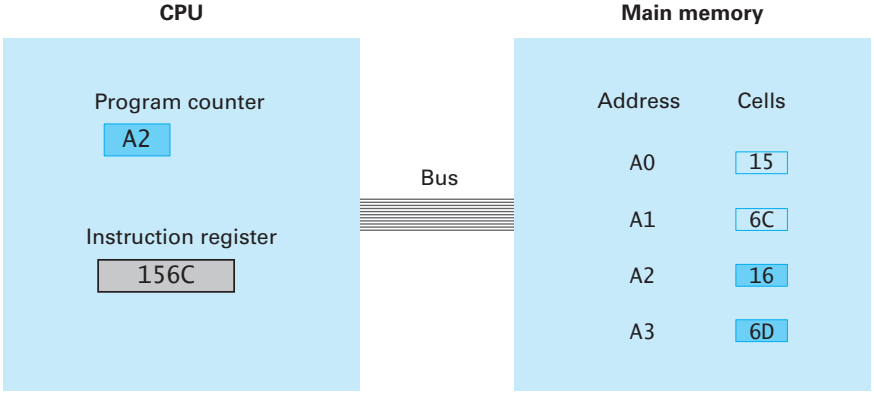


Figure 2.11 Performing the fetch step of the machine cycle



a. At the beginning of the fetch step the instruction starting at address A0 is retrieved from memory and placed in the instruction register.



b. Then the program counter is incremented so that it points to the next instruction.

and increments the program counter to A4. The values in the program counter and instruction register therefore become the following:

```
Program Counter: A4
Instruction Register: 166D
```

Now the CPU decodes the instruction 166D and determines that it is to load register 6 with the contents of memory address 6D. It then executes the instruction. It is at this time that register 6 is actually loaded.

Since the program counter now contains A4, the CPU extracts the next instruction starting at this address. The result is that 5056 is placed in the instruction register, and the program counter is incremented to A6. The CPU now decodes the contents of its instruction register and executes it by activating the two's complement addition circuitry with inputs being registers 5 and 6.

During this execution step, the arithmetic/logic unit performs the requested addition, leaves the result in register 0 (as requested by the control unit), and reports to the control unit that it has finished. The CPU then begins another machine cycle. Once again, with the aid of the program counter, it fetches the

next instruction (306E) from the two memory cells starting at memory location A6 and increments the program counter to A8. This instruction is then decoded and executed. At this point, the sum is placed in memory location 6E.

The next instruction is fetched starting from memory location A8, and the program counter is incremented to AA. The contents of the instruction register (C000) are now decoded as the halt instruction. Consequently, the machine stops during the execute step of the machine cycle, and the program is completed.

In summary, we see that the execution of a program stored in memory involves the same process you and I might use if we needed to follow a detailed list of instructions. Whereas we might keep our place by marking the instructions as we perform them, the CPU keeps its place by using the program counter. After determining which instruction to execute next, we would read the instruction and extract its meaning. Then, we would perform the task requested and return to the list for the next instruction in the same manner that the CPU executes the instruction in its instruction register and then continues with another fetch.

Programs Versus Data

Many programs can be stored simultaneously in a computer's main memory, as long as they occupy different locations. Which program will be run when the machine is started can then be determined merely by setting the program counter appropriately.

One must keep in mind, however, that because data are also contained in main memory and encoded in terms of 0s and 1s, the machine alone has no way of knowing what is data and what is program. If the program counter were assigned the address of data instead of the address of the desired program, the CPU, not knowing any better, would extract the data bit patterns as though they were instructions and execute them. The final result would depend on the data involved.

We should not conclude, however, that providing programs and data with a common appearance in a machine's memory is bad. In fact, it has proved a useful attribute because it allows one program to manipulate other programs (or even itself) the same as it would data. Imagine, for example, a program that modifies itself in response to its interaction with its environment and thus exhibits the ability to learn, or perhaps a program that writes and executes other programs in order to solve problems presented to it.

Questions & Exercises

1. Suppose the memory cells from addresses 00 to 05 in the machine described in Appendix C contain the (hexadecimal) bit patterns given in the following table:

Address	Contents
00	14
01	02
02	34
03	17
04	C0
05	00

If we start the machine with its program counter containing 00, what bit pattern is in the memory cell whose address is hexadecimal 17 when the machine halts?

2. Suppose the memory cells at addresses B0 to B8 in the machine described in Appendix C contain the (hexadecimal) bit patterns given in the following table:

Address	Contents
B0	13
B1	B8
B2	A3
B3	02
B4	33
B5	B8
B6	C0
B7	00
B8	0F

a. If the program counter starts at B0, what bit pattern is in register number 3 after the first instruction has been executed?

b. What bit pattern is in memory cell B8 when the halt instruction is executed?

3. Suppose the memory cells at addresses A4 to B1 in the machine described in Appendix C contain the (hexadecimal) bit patterns given in the following table:

Address	Contents
A4	20
A5	00
A6	21
A7	03
A8	22
A9	01
AA	B1
AB	B0
AC	50
AD	02
AE	B0
AF	AA
B0	C0
B1	00

When answering the following questions, assume that the machine is started with its program counter containing A4.

a. What is in register 0 the first time the instruction at address AA is executed?

b. What is in register 0 the second time the instruction at address AA is executed?

c. How many times is the instruction at address AA executed before the machine halts?

4. Suppose the memory cells at addresses F0 to F9 in the machine described in Appendix C contain the (hexadecimal) bit patterns described in the following table:

Address	Contents
F0	20
F1	C0
F2	30
F3	F8
F4	20
F5	00
F6	30
F7	F9
F8	FF
F9	FF

If we start the machine with its program counter containing F0, what does the machine do when it reaches the instruction at address F8?

2.4 Arithmetic/Logic Instructions

As indicated earlier, the arithmetic/logic group of instructions consists of instructions requesting arithmetic, logic, and shift operations. In this section, we look at these operations more closely.

Logic Operations

We introduced the logic operations AND, OR, and XOR (exclusive or, often pronounced, "ex-or") in Chapter 1 as operations that combine two input bits to produce a single output bit. These operations can be extended to **bitwise operations** that combine two strings of bits to produce a single output string by applying the basic operation to individual columns. For example, the result of ANDing the patterns 10011010 and 11001001 results in

```
  10011010
AND 11001001
  10001000
```

where we have merely written the result of ANDing the two bits in each column at the bottom of the column. Likewise, ORing and XORing these patterns would produce

```
  10011010
OR 11001001
  11011011
```

```
  10011010
XOR 11001001
  01010011
```

One of the major uses of the AND operation is for placing 0s in one part of a bit pattern while not disturbing the other part. There are many applications for this in practice, such as filtering certain colors out of a digital image represented in the RGB format, as described in the previous chapter. Consider, for example, what happens if the byte 00001111 is the first operand of an AND operation.

Without knowing the contents of the second operand, we still can conclude that the four most significant bits of the result will be 0s. Moreover, the four least significant bits of the result will be a copy of that part of the second operand, as shown in the following example:

```
    00001111
AND 10101010
    00001010
```

This use of the AND operation is an example of the process called **masking.** Here one operand, called a **mask,** determines which part of the other operand will affect the result. In the case of the AND operation, masking produces a result that is a partial replica of one of the operands, with 0s occupying the nonduplicated positions. One trivial use of the AND operation in this context would be to mask off all of the bits associated with the red component of the pixels in an image, leaving only the blue and green components. This transformation is frequently available as an option in image manipulation software.

AND operations are useful when manipulating other types of **bit map** besides images, whenever a string of bits is used in which each bit represents the presence or absence of a particular object. As a non-graphical example, a string of 52 bits, in which each bit is associated with a particular playing card, can be used to represent a poker hand by assigning 1s to those five bits associated with the cards in the hand and 0s to all the others. Likewise, a bit map of 52 bits, of which thirteen are 1s, can be used to represent a hand of bridge, or a bit map of 32 bits can be used to represent which of thirty-two ice cream flavors are available.

Suppose, then, that the eight bits in a memory cell are being used as a bit map, and we want to find out whether the object associated with the third bit from the high-order end is present. We merely need to AND the entire byte with the mask 00100000, which produces a byte of all 0s if and only if the third bit from the high-order end of the bit map is itself 0. A program can then act accordingly by following the AND operation with a conditional branch instruction. Moreover, if the third bit from the high-order end of the bit map is a 1, and we want to change it to a 0 without disturbing the other bits, we can AND the bit map with the mask 11011111 and then store the result in place of the original bit map.

Where the AND operation can be used to duplicate a part of a bit string while placing 0s in the nonduplicated part, the OR operation can be used to duplicate a part of a string while putting 1s in the nonduplicated part. For this we again use a mask, but this time we indicate the bit positions to be duplicated with 0s and use 1s to indicate the nonduplicated positions. For example, ORing any byte with 11110000 produces a result with 1s in its most significant four bits while its remaining bits are a copy of the least significant four bits of the other operand, as demonstrated by the following example:

```
   11110000
OR 10101010
   11111010
```

Consequently, whereas the mask 11011111 can be used with the AND operation to force a 0 in the third bit from the high-order end of a byte, the mask 00100000 can be used with the OR operation to force a 1 in that position.

A major use of the XOR operation is in forming the complement of a bit string. XORing any byte with a mask of all 1s produces the complement of the byte. For example, note the relationship between the second operand and the result in the following example:

```
   11111111
XOR 10101010
   01010101
```

The XOR operation can be used to invert all of the bits of an RGB bitmap image, resulting in an "inverted" color image in which light colors have been replaced by dark colors, and vice versa.

In the machine language described in Appendix C, op-codes 7, 8, and 9 are used for the logic operations OR, AND, and XOR, respectively. Each requests that the corresponding logic operation be performed between the contents of two designated registers and that the result be placed in another designated register. For example, the instruction 7ABC requests that the result of ORing the contents of registers B and C be placed in register A.

Rotation and Shift Operations

The operations in the class of rotation and shift operations provide a means for moving bits within a register and are often used in solving alignment problems. These operations are classified by the direction of motion (right or left) and whether the process is circular. Within these classification guidelines are numerous variations with mixed terminology. Let us take a quick look at the ideas involved.

Consider a register containing a byte of bits. If we shift its contents one bit to the right, we imagine the rightmost bit falling off the edge and a hole appearing at the leftmost end. What happens with this extra bit and the hole is the distinguishing feature among the various shift operations. One technique is to place the bit that fell off the right end in the hole at the left end. The result is a **circular shift,** also called a **rotation.** Thus, if we perform a right circular shift on a byte-size bit pattern eight times, we obtain the same bit pattern we started with.

Another technique is to discard the bit that falls off the edge and always fill the hole with a 0. The term **logical shift** is often used to refer to these operations. Such shifts to the left can be used for multiplying two's complement representations by two. After all, shifting binary digits to the left corresponds to multiplication by two, just as a similar shift of decimal digits corresponds to multiplication by ten. Moreover, division by two can be accomplished by shifting the binary string to the right. In either shift, care must be taken to preserve the sign bit when using certain notational systems. Thus, we often find right shifts that always fill the hole (which occurs at the sign bit position) with its original value. Shifts that leave the sign bit unchanged are sometimes called **arithmetic shifts.**

Among the variety of shift and rotate instructions possible, the machine language described in Appendix C contains only a right circular shift, designated by op-code A. In this case the first hexadecimal digit in the operand specifies the register to be rotated, and the rest of the operand specifies the number of bits to be rotated. Thus the instruction A501 means "Rotate the contents of register

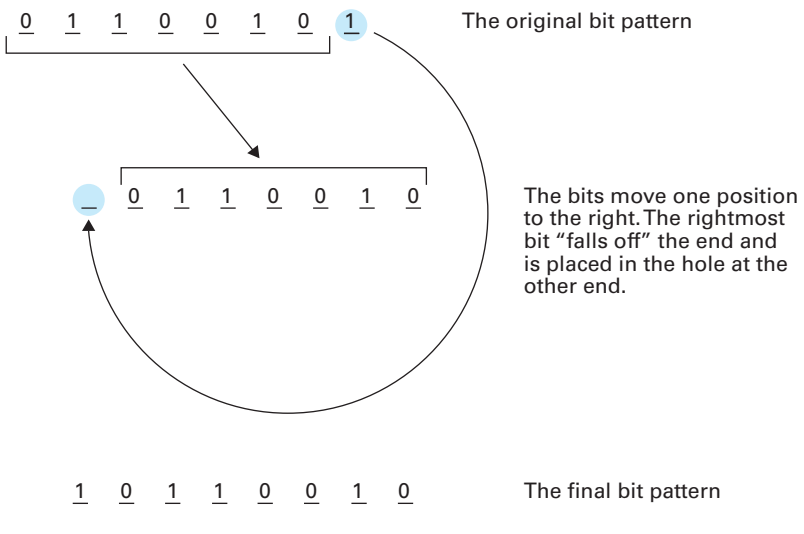
5 to the right by 1 bit." In particular, if register 5 originally contained the bit pattern 65 (hexadecimal), then it would contain B2 after this instruction is executed (Figure 2.12). (You may wish to experiment with how other shift and rotate instructions can be produced with combinations of the instructions provided in the machine language of Appendix C. For example, since a register is eight bits long, a right circular shift of three bits produces the same result as a left circular shift of five bits.)

Arithmetic Operations

Although we have already mentioned the arithmetic operations of add, subtract, multiply, and divide, a few loose ends should still be connected. First, we have already seen that subtraction can be simulated by means of addition and negation. Moreover, multiplication is merely repeated addition and division is repeated subtraction. (Six divided by two is three because three twos can be subtracted from six.) For this reason, some small CPUs are designed with only the add or perhaps only the add and subtract instructions.

We should also mention that numerous variations exist for each arithmetic operation. We have already alluded to this in relation to the add operations available on our machine in Appendix C. In the case of addition, for example, if the values to be added are stored in two's complement notation, the addition process must be performed as a straightforward column by column addition. However, if the operands are stored as floating-point values, the addition process must extract the mantissa of each, shift them right or left according to the exponent fields, check the sign bits, perform the addition, and translate the result into floating-point notation. Thus, although both operations are considered addition, the action of the machine is not the same.

Figure 2.12 Rotating the bit pattern 65 (hexadecimal) one bit to the right



Questions & Exercises

1. Perform the indicated operations.

   a.
   ```
     01001011
   AND 10101011
   ```

   b.
   ```
     100000011
   AND 11101100
   ```

   c.
   ```
     11111111
   AND 00101101
   ```

   d.
   ```
     01001011
   OR  10101011
   ```

   e.
   ```
     10000011
   OR  11101100
   ```

   f.
   ```
     11111111
   OR  00101101
   ```

   g.
   ```
     01001011
   XOR 10101011
   ```

   h.
   ```
     100000011
   XOR 11101100
   ```

   i.
   ```
     11111111
   XOR 00101101
   ```

2. Suppose you want to isolate the middle four bits of a byte by placing 0s in the other four bits without disturbing the middle four bits. What mask must you use together with what operation?

3. Suppose you want to complement the four middle bits of a byte while leaving the other four bits undisturbed. What mask must you use together with what operation?

4. a. Suppose you XOR the first two bits of a string of bits and then continue down the string by successively XORing each result with the next bit in the string. How is your result related to the number of 1s appearing in the string?

   b. How does this problem relate to determining what the appropriate parity bit should be when encoding a message?

5. It is often convenient to use a logical operation in place of a numeric one. For example, the logical operation AND combines two bits in the same manner as multiplication. Which logical operation is almost the same as adding two bits, and what goes wrong in this case?

6. What logical operation together with what mask can you use to change ASCII codes of lowercase letters to uppercase? What about uppercase to lowercase?

7. What is the result of performing a three-bit right circular shift on the following bit strings:

   a. 01101010 b. 00001111 c. 01111111

8. What is the result of performing a one-bit left circular shift on the following bytes represented in hexadecimal notation? Give your answer in hexadecimal form.

   a. AB b. 5C c. B7 d. 35

9. A right circular shift of three bits on a string of eight bits is equivalent to a left circular shift of how many bits?

10. What bit pattern represents the sum of 01101010 and 11001100 if the patterns represent values stored in two's complement notation? What if the

patterns represent values stored in the floating-point format discussed in Chapter 1?

11. Using the machine language of Appendix C, write a program that places a 1 in the most significant bit of the memory cell whose address is A7 without modifying the remaining bits in the cell.

12. Using the machine language of Appendix C, write a program that copies the middle four bits from memory cell E0 into the least significant four bits of memory cell E1, while placing 0s in the most significant four bits of the cell at location E1.

2.5 Communicating with Other Devices

Main memory and the CPU form the core of a computer. In this section, we investigate how this core, which we will refer to as the computer, communicates with peripheral devices such as mass storage systems, printers, keyboards, mice, display screens, digital cameras, and even other computers.

The Role of Controllers

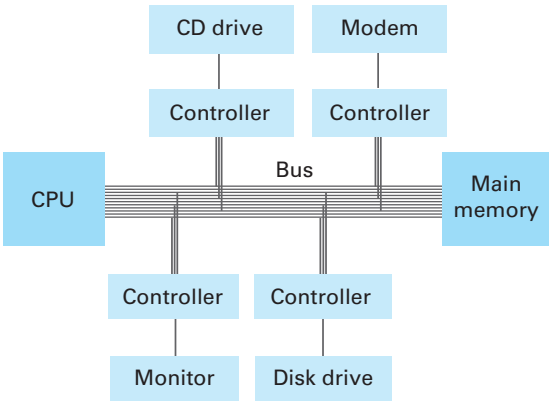
Communication between a computer and other devices is normally handled through an intermediary apparatus known as a **controller.** In the case of a personal computer, a controller may consist of circuitry permanently mounted on the computer's motherboard or, for flexibility, it may take the form of a circuit board that plugs into a slot on the motherboard. In either case, the controller connects via cables to peripheral devices within the computer case or perhaps to a connector, called a **port,** on the back of the computer where external devices can be attached. These controllers are sometimes small computers themselves, each with its own memory circuitry and simple CPU that performs a program directing the activities of the controller.

A controller translates messages and data back and forth between forms compatible with the internal characteristics of the computer and those of the peripheral device to which it is attached. Originally, each controller was designed for a particular type of device; thus, purchasing a new peripheral device often required the purchase of a new controller as well.

Recently, steps have been taken within the personal computer arena to develop standards, such as the **universal serial bus (USB)** and **FireWire,** by which a single controller is able to handle a variety of devices. For example, a single USB controller can be used as the interface between a computer and any collection of USB-compatible devices. The list of devices on the market today that can communicate with a USB controller includes mice, printers, scanners, mass storage devices, digital cameras, and smartphones.

Each controller communicates with the computer itself by means of connections to the same bus that connects the computer's CPU and main memory (Figure 2.13). From this position it is able to monitor the signals being sent between the CPU and main memory as well as to inject its own signals onto the bus.

Figure 2.13 Controllers attached to a machine's bus

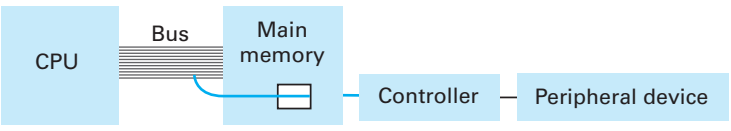


With this arrangement, the CPU is able to communicate with the controllers attached to the bus in the same manner that it communicates with main memory. To send a bit pattern to a controller, the bit pattern is first constructed in one of the CPU's general-purpose registers. Then an instruction similar to a STORE instruction is executed by the CPU to "store" the bit pattern in the controller. Likewise, to receive a bit pattern from a controller, an instruction similar to a LOAD instruction is used.

In some computer designs the transfer of data to and from controllers is directed by the same LOAD and STORE op-codes that are already provided for communication with main memory. In these cases, each controller is designed to respond to references to a unique set of addresses while main memory is designed to ignore references to these locations. Thus when the CPU sends a message on the bus to store a bit pattern at a memory location that is assigned to a controller, the bit pattern is actually "stored" in the controller rather than main memory. Likewise, if the CPU tries to read data from such a memory location, as in a LOAD instruction, it will receive a bit pattern from the controller rather than from memory. Such a communication system is called **memory-mapped I/O** because the computer's input/output devices appear to be in various memory locations (Figure 2.14).

An alternative to memory-mapped I/O is to provide special op-codes in the machine language to direct transfers to and from controllers. Instructions with these op-codes are called I/O instructions. As an example, if the language

Figure 2.14 A conceptual representation of memory-mapped I/O



described in Appendix C followed this approach, it might include an instruction such as F5A3 to mean "STORE the contents of register 5 in the controller identified by the bit pattern A3."

Direct Memory Access

Since a controller is attached to a computer's bus, it can carry on its own communication with main memory during those nanoseconds in which the CPU is not using the bus. This ability of a controller to access main memory is known as **direct memory access (DMA),** and it is a significant asset to a computer's performance. For instance, to retrieve data from a sector of a disk, the CPU can send requests encoded as bit patterns to the controller attached to the disk asking the controller to read the sector and place the data in a specified area of main memory. The CPU can then continue with other tasks while the controller performs the read operation and deposits the data in main memory via DMA. Thus two activities will be performed at the same time. The CPU will be executing a program and the controller will be overseeing the transfer of data between the disk and main memory. In this manner, the computing resources of the CPU are not wasted during the relatively slow data transfer.

The use of DMA also has the detrimental effect of complicating the communication taking place over a computer's bus. Bit patterns must move between the CPU and main memory, between the CPU and each controller, and between each controller and main memory. Coordination of all this activity on the bus is a major design issue. Even with excellent designs, the central bus can become an impediment as the CPU and the controllers compete for bus access. This impediment

USB and FireWire

The universal serial bus (USB) and FireWire are standardized serial communication systems that simplify the process of adding new peripheral devices to a personal computer. USB was developed under the lead of Intel. The development of FireWire was led by Apple. In both cases the underlying theme is for a single controller to provide external ports at which a variety of peripheral devices can be attached. In this setting, the controller translates the internal signal characteristics of the computer to the appropriate USB or FireWire standard signals. In turn, each device connected to the controller converts its internal idiosyncrasies to the same USB or FireWire standard, allowing communication with the controller. The result is that attaching a new device to a PC does not require the insertion of a new controller. Instead, one merely plugs any USB compatible device into a USB port or a FireWire compatible device into a FireWire port.

Of the two, FireWire provides a faster transfer rate, but the lower cost of USB 2.0 technology has made it the leader in the lower-cost mass market arena. A new, faster version of the USB standard, version 3.0, has also begun to appear on the market. USB-compatible devices on the market today include mice, keyboards, printers, scanners, digital cameras, smartphones, and mass storage systems designed for backup applications. FireWire applications tend to focus on devices that require higher transfer rates such as video recorders and online mass storage systems.

is known as the **von Neumann bottleneck** because it is a consequence of the underlying **von Neumann architecture** in which a CPU fetches its instructions from memory over a central bus.

Handshaking

The transfer of data between two computer components is rarely a one-way affair. Even though we may think of a printer as a device that receives data, the truth is that a printer also sends data back to the computer. After all, a computer can produce and send characters to a printer much faster than the printer can print them. If a computer blindly sent data to a printer, the printer would quickly fall behind, resulting in lost data. Thus a process such as printing a document involves a constant two-way dialogue, known as **handshaking,** in which the computer and the peripheral device exchange information about the device's status and coordinate their activities.

Handshaking often involves a **status word,** which is a bit pattern that is generated by the peripheral device and sent to the controller. The status word is a bit map in which the bits reflect the conditions of the device. For example, in the case of a printer, the value of the least significant bit of the status word may indicate whether the printer is out of paper, while the next bit may indicate whether the printer is ready for additional data. Still another bit may be used to indicate the presence of a paper jam. Depending on the system, the controller may respond to this status information itself or make it available to the CPU. In either case, the status word provides the mechanism by which communication with a peripheral device can be coordinated.

Popular Communication Media

Communication between computing devices is handled over two types of paths: parallel and serial. These terms refer to the manner in which signals are transferred with respect to each other. In the case of **parallel communication,** several signals are transferred at the same time, each on a separate "line." Such a technique is capable of transferring data rapidly but requires a relatively complex communication path. Examples include a computer's internal bus where multiple wires are used to allow large blocks of data and other signals to be transferred simultaneously.

In contrast, **serial communication** is based on transferring signals one after the other over a single line. Thus serial communication requires a simpler data path than parallel communication, which is the reason for its popularity. USB and FireWire, which offer relatively high-speed data transfer over short distances of only a few meters, are examples of serial communication systems. For slightly longer distances (within a home or office building), serial communication over Ethernet connections (Section 4.1), either by wire or radio broadcast, are popular.

For communication over greater distances, traditional voice telephone lines dominated the personal computer arena for many years. These communication paths, consisting of a single wire over which tones are transferred one after the other, are inherently serial systems. The transfer of digital data over these lines is accomplished by first converting bit patterns into audible tones by means of a **modem** (short for *modulator-demodulator*), transferring these tones serially over the telephone system, and then converting the tones back into bits by another modem at the destination.

For faster long-distance communication over traditional telephone lines, telephone companies offer a service known as **DSL (Digital Subscriber Line),** which takes advantage of the fact that existing telephone lines are capable of handling a wider frequency range than that used by traditional voice communication. More precisely, DSL uses frequencies above the audible range to transfer digital data while leaving the lower-frequency spectrum for voice communication. Although DSL has been highly successful, telephone companies are rapidly upgrading their systems to fiber-optic lines, which support digital communication more readily than traditional telephone lines.

Cable modems are a competing technology that modulate and demodulate bit patterns to be transmitted over cable television systems. Many cable providers now make use of both fiber-optic lines and traditional coaxial cable to provide both high definition television signals and computer network access.

Satellite links via high-frequency radio broadcast make computer network access possible even in some remote locations far from high speed telephone and cable television networks.

Communication Rates

The rate at which bits are transferred from one computing component to another is measured in **bits per second (bps).** Common units include **Kbps** (kilo-bps, equal to one thousand bps), **Mbps** (mega-bps, equal to one million bps), and **Gbps** (giga-bps, equal to one billion bps). (Note the distinction between bits and bytes—that is, 8 Kbps is equal to 1 KB per second. In abbreviations, a lowercase b usually means *bit* whereas an uppercase B means *byte.*)

For short distance communication, USB 2.0 and FireWire provide transfer rates of several hundred Mbps, which is sufficient for most multimedia applications. This, combined with their convenience and relatively low cost, is why they are popular for communication between home computers and local peripherals such as printers, external disk drives, and cameras.

By combining **multiplexing** (the encoding or interweaving of data so that a single communication path serves the purpose of multiple paths) and data compression techniques, traditional voice telephone systems were able to support transfer rates of 57.6 Kbps. This falls short of the needs of today's multimedia and Internet applications, such as high definition video streaming from sites like Netflix or YouTube. To play MP3 music recordings requires a transfer rate of about 64 Kbps, and to play even low quality video clips requires transfer rates measured in units of Mbps. This is why alternatives such as DSL, cable, and satellite links, which provide transfer rates well into the Mbps range, have replaced traditional audio telephone systems. (For example, DSL offers transfer rates on the order of 54 Mbps.)

The maximum rate available in a particular setting depends on the type of the communication path and the technology used in its implementation. This maximum rate is often loosely equated to the communication path's **bandwidth,** although the term *bandwidth* also has connotations of capacity rather than transfer rate. That is, to say that a communication path has a high bandwidth (or provides **broadband** service) means that the communication path has the ability to transfer bits at a high rate as well as the capacity to carry large amounts of information simultaneously.

Questions & Exercises

1. Assume that the machine described in Appendix C uses memory-mapped I/O and that the address B5 is the location within the printer port to which data to be printed should be sent.
   a. If register 7 contains the ASCII code for the letter A, what machine language instruction should be used to cause that letter to be printed at the printer?
   b. If the machine executes a million instructions per second, how many times can this character be sent to the printer in one second?
   c. If the printer is capable of printing five traditional pages of text per minute, will it be able to keep up with the characters being sent to it in (b)?
2. Suppose that the hard disk on your personal computer rotates at 3000 revolutions a minute, that each track contains 16 sectors, and that each sector contains 1024 bytes. Approximately what communication rate is required between the disk drive and the disk controller if the controller is going to receive bits from the disk drive as they are read from the spinning disk?
3. Estimate how long it would take to transfer a 300-page novel encoded in 16-bit Unicode characters at a transfer rate of 54 Mbps.

2.6 Programming Data Manipulation

One of the essential features of computer programming languages such as Python is that they shield users from the tedious details of working with the lowest levels of the machine. Having just completed much of a chapter on the lowest levels of data manipulation in computer processors, it is instructive to review some of the major details that Python scripts shield the programmer from needing to worry about.

As we will explore in greater detail in Chapter 6, high-level programming language statements are mapped down to low-level machine instructions in order to be executed. A single Python statement might map to a single machine instruction, or to many tens or even hundreds of machine instructions, depending on the complexity of the statement and the efficiency of the machine language. Different implementations of the Python language interpreter, in concert with other elements of the computer's operating system software, take care of this mapping process for each particular computer processor. As a result, the Python programmer does not need to know whether she is executing her Python script on a RISC processor or a CISC processor.

We can recognize many Python operations that correspond closely to the basic machine instructions for modern computers or for the simple machine described in Appendix C. Addition of Python integers and floating-point numbers clearly resembles the ADD op-codes of our simple machine. Assigning values

to variables surely involves the LOAD, STORE, and MOVE op-codes in some arrangement. Python shields us from worrying about which processor registers are in use, but leverages the op-codes of the machine to carry out our instructions. We cannot see the instruction register, program counter, or memory cell addresses, but the Python script executes sequentially, one statement after the other, in the same way as the simple machine language programs.

Logic and Shift Operations

Logic and shift operations can be executed on any kind of numerical data, but because they often deal with individual bits of data, it is easiest to illustrate these operations with binary values. Just as Python uses the `0x` prefix to specify values in hexadecimal, the `0b` prefix can be used to specify values in binary[1].

```
x    = 0b00110011
mask = 0b00001111
```

Note that this is effectively no different from assigning `x` the value 51, (which is 110011 in binary), or `0x33` (which is 51 expressed in hexadecimal), or from assigning `mask` the value 15, (which is 1111 in binary), or `0x0F` (15 in hexadecimal). The representation we use to spell out the integer value in the Python assignment statement does not change how it is represented in the computer, only how human readers understand it.

Built-in Python operators exist for each of the bitwise logical operators described in Section 2.4.

```
print(0b00000101 ^ 0b00000100)      # Prints 5 XOR 4, which is 1
print(0b00000101 | 0b00000100)      # Prints 5 OR  4, which is 5
print(0b00000101 & 0b00000100)      # Prints 5 AND 4, which is 4
```

As a result, we can replicate each of the example problems of Section 2.4 as Python code.

```
                                    #      10011010
print(0b10011010 & 0b11001001)      #    AND 11001001
                                    #      10001000

                                    #      10011010
print(0b10011010 | 0b11001001)      #    OR  11001001
                                    #      11011011

                                    #      10011010
print(0b10011010 ^ 0b11001001)      #    XOR 11001001
                                    #      01010011
```

For all of these examples, Python will print the result in its default output representation, which is base-10. If the user would also like the output to be displayed in binary notation, a built-in function exists to convert any integer value into the string of zero and one characters for the corresponding binary representation.

[1]This syntax is another recent addition to the evolving Python language. Make sure that you are using at least Python 3 to replicate these examples.

```
print(bin(0b10011010 & 0b11001001))   # Prints "0b10001000"
print(bin(0b10011010 | 0b11001001))   # Prints "0b11011011"
print(bin(0b10011010 ^ 0b11001001))   # Prints "0b1010011"
```

Because newer versions of Python can use an arbitrary number of digits for representing numbers, leading zeros are not printed. Thus, the third line above prints only seven digits, rather than eight.

Python's built-in operators for performing logical shift operations consist of dual greater-than and less-than symbols, visually suggesting the direction of shift. The operand on the right of the operator indicates the number of bit positions to shift.

```
print(0b00111100 >> 2)   # Prints  "15", which is 0b00001111
print(0b00111100 << 2)   # Prints "240", which is 0b11110000
```

In addition to shifting bit masks left or right, bit shift operators are also an efficient way to multiply (left shift) or divide (right shift) by powers of 2.

Control Structures

The control group of machine language instructions presented earlier in this chapter affords us a mechanism for jumping from one part of a program to another. In higher-level languages like Python, this enables what are called **control structures,** syntax patterns that allow us to express algorithms more powerfully. One example of this is the `if`-statement, which allows a segment of code to be conditionally skipped if a Boolean value in the script is not true.

```
if (water_temp > 140):
  print('Bath water too hot!')
```

Intuitively, this Python snippet will be mapped to machine instructions that make the comparison between the `water_temp` variable and the integer value 140, probably both previously loaded into registers. A conditional jump instruction will skip over the machine instructions for the `print()` built-in if the `water_temp` value was not 140 or larger.

Another control structure is the looping construct `while`, which allows a segment of code to be executed multiple times, often subject to some condition.

```
while (n < 10):
  print(n)
  n = n + 1
```

Assuming the variable `n` starts with a value less than 10, this loop will continue printing and incrementing `n` until it becomes greater than or equal to 10.

We will spend more time examining these and other control structures in Chapter 5 and beyond. For now, we focus on a mechanism that allows us to jump to another part of the program, carry out a desired task, and then return to the program point we came from.

Functions We have already seen three built-in Python operations that do not follow the same syntactic form as the arithmetic and logic operators. The `print()`,

`str()` and `bin()` operations are invoked using given names instead of symbols, and also involve parentheses wrapped around their operands.

Both of these are examples of a Python language feature called **functions.** The term *function* in mathematics is often used to describe algebraic relationships, such as "f(x) = x^2 + 3x + 4." Upon seeing such a function definition, we understand that in subsequent lines the expression "f(5)" is taken to mean that the value 5 should be plugged in wherever the parameter x occurs in the expression defining f( ). Thus, f(5) = 5^2 + 3*5 + 4 = 25 + 15 + 4 = 44. Programming language functions are quite similar in that they allow us to use a name for a series of operations that should be performed on the given parameter or parameters. Due to the way that this language feature is mapped to lower level machine languages, the appearance of a function in an expression or statement is known as a **function call,** or sometimes **calling** a function.

The occurrences of `print()` and `bin()` in the examples above are two such function calls; they indicate that the Python interpreter will go execute the definition of the named function, and then return to continue with its work. The syntax is to follow the name of the function immediately with an opening parenthesis, and then to give the function **argument value** that will be plugged in for the parameter when the function definition is evaluated, followed by a closing parenthesis. It is important to match opening and closing parentheses—not doing so will cause a Python syntax error and is a common mistake for beginners.

From now on, we will follow the convention of including the parentheses when talking about Python functions, such as `print()`, so as to clearly denote them as distinct from variables or other items.

Functions come in many varieties beyond what we have already seen. Some functions take more than one argument, such as the `max()` function:

```
x = 1034
y = 1056
z = 2078
biggest = max(x, y, z)
print(biggest)                  # Prints "2078"
```

Multiple arguments are separated by commas within the parentheses. Some functions **return** a value, which is to say that the function call itself can appear as part of a more complex expression, or as the right-hand side of an assignment statement. These are sometimes called **fruitful functions.** This is the case for both `max()` (as above) and `bin()`, which takes an integer value as an argument and returns the corresponding string of zeros and ones. Other functions do not return a value, and usually are used as standalone statements, as is the case for `print()`. Functions that do not return a value are sometimes called **void functions,** or **procedures,** although Python makes no distinction in its syntax rules. It makes no sense to assign the result of a void function to a variable, as in

```
x = print('hello world!')      # x is assigned None
```

although this is not an error in Python, per se, and is subtly different from not assigning a value to `x` at all.

Each of the functions we have seen so far is one of the few dozen built-in functions that Python knows about, but there are extensive libraries of additional functions that a more advanced script can refer to. The Python library modules

contain many useful functions that may not normally be required, but that can be called upon when needed.

```
# Calculates the hypotenuse of a right triangle
import math

sideA = 3.0
sideB = 4.0
# Calculate third side via Pythagorean Theorem
hypotenuse = math.sqrt(sideA**2 + sideB**2)

print(hypotenuse)
```

In this example, the `import` statement forewarns the Python interpreter that the script refers to the library called "`math`," which happens to be one of the standard set of library modules that Python comes equipped with. The `sqrt()` function defined within the `math` library module provides the square root of the argument, which in this case was the expression of `sideA` squared plus `sideB` squared. Note that the library function call includes both the module name ("`math`") and the function name ("`sqrt`"), joined by a period.

The Python `math` module includes dozens of useful mathematical functions, including logorithmic, trigonometric, and hyperbolic functions, as well as some familiar constant values such as `math.pi`.

Beyond the built-in and library module functions, Python provides syntax for a script to define its own functions. We will define a few very simple examples at the end of this section, and explore more elaborate variations in a later chapter.

Input and Output

The previous example snippets and scripts have used the built-in Python `print()` function to output results. Many programming languages provide similar mechanisms for achieving input and output, providing programmers with a convenient abstraction to move data in or out of the computer processor. In fact, these I/O built-ins communicate with the hardware controllers and peripheral devices discussed in the previous section.

None of our example scripts thus far have required any input from the user. Simple user input can be accomplished with the built-in Python `input()` function.

```
echo = input('Please enter a string to echo: ')
print(echo * 3)
```

The `input()` function takes as an optional argument a prompt string to present to the user when waiting for input. When run, this script will pause after displaying "`Please enter a string to echo: `", and wait for the user to type something. When the user hits the enter key, the script assigns the string of characters typed (not including the enter key,) to the variable `echo`. The second line of the script then outputs the string repeated three times. (Recall that the `*` operator replicates string operands.)

Armed with the ability to acquire input, let's rewrite our hypotenuse script to prompt a user for the side lengths rather than hardcode the values into assignment statements.

```
# Calculates the hypotenuse of a right triangle
import math

# Inputting the side lengths, first try
sideA = input('Length of side A? ')
sideB = input('Length of side B? ')
# Calculate third side via Pythagorean Theorem
hypotenuse = math.sqrt(sideA**2 + sideB**2)

print(hypotenuse)
```

When run, this script prompts the user with, "`Length of side A?` ", and awaits input. Let us suppose that the user types "3" and enter. The script prompts the user with "`Length of side B?` ", and awaits input. Let us suppose that the user types "4" and enter. At this point, the Python interpreter aborts the script, printing out:

```
hypotenuse = math.sqrt(sideA**2 + sideB**2)
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

This type of error can be easy to create in a dynamically typed language like Python. Our hypotenuse calculation, which worked in the earlier version of the script, now causes an error when the values have been read as input from the user instead. The problem is indeed a "`TypeError`," stemming from the fact that Python no longer knows how to take the square of variable `sideA`, because `sideA` is now a character string in this version of the script, rather than an integer as before. The problem comes not from the line that calculates the hypotenuse, but from earlier in the script, when the values of `sideA` and `sideB` are returned from `input()`. This, too, is common when encountering errors with programming languages. The Python interpreter attempts to provide the line of script responsible for the problem, but the real culprit is actually earlier in the script.

In the string echoing snippet above, it was clear that the value assigned to `echo` should be the string of characters typed in by the user. The `input()` function behaves in the same way in the hypotenuse program, even though the programmer's intent is now to enter integer values. The representation of the ASCII- or UTF-8-encoded string "4" differs from the two's complement representation of the integer 4, and the Python script must explicitly make the conversion from one representation to the other before proceeding to calculations with integers.

Fortunately, another built-in function provides the capability. The `int()` function attempts to convert its argument into an integer representation. If it cannot, an appropriate error message is produced.

There are at least three places that we can use the `int()` function to remove the bug in this script. We can call it before even assigning the result of `input()` to the variable. We can add new lines of script that are only for calling the conversion function. Or, we can make the conversion just before squaring the variable within the call to the `math.sqrt()` function. The revised script below uses the first option; the other two are left as an exercise for the reader.

```
# Calculates the hypotenuse of a right triangle
import math
```

```
# Inputting the side lengths, with integer conversion
sideA = int(input('Length of side A? '))
sideB = int(input('Length of side B? '))
# Calculate third side via Pythagorean Theorem
hypotenuse = math.sqrt(sideA**2 + sideB**2)

print(hypotenuse)
```

The revised script operates as intended and can be used for many right triangles without having to edit the script, as in the pre-input version.

As a final note, the `int()` function performs its conversion by carefully examining the string argument and interpreting it as a number. If the input string is a number, but not an integer, as for example, "3.14," the `int()` function discards the fractional portion and returns only the integer value. This operation is a truncation and will not "round up" as a human might expect. Similar conversion functions exist in Python for all of the other standard value types.

Marathon Training Assistant

The complete Python script below demonstrates many of the concepts introduced in this section. As the popularity of recreational distance running has increased, many participants find themselves pursuing complex training schedules to prepare their bodies for the rigors of running a marathon. This script assists a runner who wishes to calculate how long her training workout will take, based upon the distance and pace she wishes to run. Given a *pace* (number of minutes and seconds to run a single mile), and a total mileage, this script calculates the projected elapsed time to run the workout, as well as a user-friendly speed calculation in miles per hour. Figure 2.15 gives some example data points; in each row, given the first three columns as input, the last three columns would be the expected results. Note that different implementations of Python may print a different number of decimal places from Figure 2.15 for speed values that don't work out to round numbers.

Figure 2.15 Example marathon training data

Time Per Mile				Total Elapsed Time	
Minutes	Seconds	Miles	Speed (mph)	Minutes	Seconds
9	14	5	6.49819494584	46	10
8	0	3	7.5	24	0
7	45	6	7.74193548387	46	30
7	25	1	8.08988764044	7	25

```
# Marathon training assistant.
import math

# This function converts a number of minutes and seconds into just seconds.
def total_seconds(min, sec):
  return min * 60 + sec

# This function calculates a speed in miles per hour given
# a time (in seconds) to run a single mile.
def speed(time):
  return 3600 / time

# Prompt user for pace and mileage.
pace_minutes = int(input('Minutes per mile? '))
pace_seconds = int(input('Seconds per mile? '))
miles = int(input('Total miles? '))

# Calculate and print speed.
mph = speed(total_seconds(pace_minutes, pace_seconds))
print('Your speed is')
print(mph)

# Calculate elapsed time for planned workout.
total = miles * total_seconds(pace_minutes, pace_seconds)
elapsed_minutes = total // 60
elapsed_seconds = total % 60

print('Your total elapsed time is')
print(elapsed_minutes)
print(elapsed_seconds)
```

The script above uses both built-in functions—`input()`, `int()`, and `print()`—as well as user-defined functions—`speed()` and `total_seconds()`. The keyword `def` precedes a user function definition and is followed by the name for the function and a list of parameters to be provided when the function is called. The indented line that follows is called the **body** of a function and expresses the steps that define the function. In a later chapter, we will see examples of functions with more than one statement in their body. The keyword `return` highlights the expression that will be calculated to find the result of the function.

The user-defined functions are defined at the top of the script, but are not actually invoked until the script reaches the lines where they are called as part of a larger expression. Note also the way in which function calls are stacked in this script. The results of the calls to `input()` are immediately passed as arguments to the `int()` function, and the result of the `int()` function is then assigned to a variable. Similarly, the result of `total_seconds()` is immediately passed as an argument to the `speed()` function, whose result is then assigned to the variable `mph`. In each of these cases, it would be permissible to make the function calls one at a time, assign the result to a new variable, and then call the next function

that relies on the first result. However, this more compact form is more succinct and does not require a proliferation of temporary variables to hold intermediate results of the calculation.

Given inputs of 7 minutes and 45 seconds per mile, for 6 miles, this script outputs:

```
Your speed is
7.74193548387
Your total elapsed time is
46
30
```

The format of this output remains quite primitive. It lacks proper units (`7.74193548387 mph`, and `46 minutes`, `30 seconds`), prints an inappropriate number of decimal places for a simple calculation, and breaks lines in too many places. Cleaner output is left as an exercise for the reader.

Questions & Exercises

1. The hypotenuse example script truncates the sides to integers, but outputs a floating-point number. Why? Adapt the script to output an integer.
2. Adapt the hypotenuse script to use floating-point numbers as input, without truncating them. Which is more appropriate, the integer version from the previous question, or the floating-point version?
3. The Python built-in function `str()` will convert a numerical argument into a character string representation, and the `'+'` can be used to concatenate strings together. Use these to modify the marathon script to produce cleaner output, for example:

   ```
   Your speed is 7.74193548387 mph
   Your total elapsed time is 46 minutes, 30 seconds
   ```

4. Use the Python built-in `bin()` to write a script that reads a base-10 integers as input and outputs the corresponding binary representation of that integer in ones and zeros.
5. The XOR operation is often used both for efficiently calculating checksums (see Section 1.9) and encryption (see Section 4.5). Write a simple Python script that reads in a number and outputs that number XORed with a pattern of ones and zeros, such as 0x55555555. The same script will "encrypt" a number into a seemingly unrelated number, but when run again and given the encrypted number as input will return the original number.
6. Explore some of the error conditions that you can create with unexpected inputs to the example scripts from this section. What happens if you enter all zeros for the hypotenuse script or the marathon script? What about negative numbers? Strings of characters instead of numbers?

2.7 Other Architectures

To broaden our perspective, let us consider some alternatives to the traditional machine architecture we have discussed so far.

Pipelining

Electric pulses travel through a wire no faster than the speed of light. Since light travels approximately 1 foot in a nanosecond (one billionth of a second), it requires at least 2 nanoseconds for the CPU to fetch an instruction from a memory cell that is 1 foot away. (The read request must be sent to memory, requiring at least 1 nanosecond, and the instruction must be sent back to the CPU, requiring at least another nanosecond.) Consequently, to fetch and execute an instruction in such a machine requires several nanoseconds—which means that increasing the execution speed of a machine ultimately becomes a miniaturization problem.

However, increasing execution speed is not the only way to improve a computer's performance. The real goal is to improve the machine's **throughput,** which refers to the total amount of work the machine can accomplish in a given amount of time.

An example of how a computer's throughput can be increased without requiring an increase in execution speed involves **pipelining,** which is the technique of allowing the steps in the machine cycle to overlap. In particular, while one instruction is being executed, the next instruction can be fetched, which means that more than one instruction can be in "the pipe" at any one time, each at a different stage of being processed. In turn, the total throughput of the machine is increased even though the time required to fetch and execute each individual instruction remains the same. (Of course, when a JUMP instruction is reached, any gain that would have been obtained by prefetching is not realized because the instructions in "the pipe" are not the ones needed after all.)

Modern machine designs push the pipelining concept beyond our simple example. They are often capable of fetching several instructions at the same time and actually executing more than one instruction at a time when those instructions do not rely on each other.

The Multi-Core CPU

As technology provides ways of placing more and more circuitry on a silicon chip, the physical distinction between a computer's components diminishes. For instance, a single chip might contain a CPU and main memory. This is an example of the "system-on-a-chip (SoC)" approach in which the goal is to provide a complete apparatus in a single device that can be used as an abstract tool in higher level designs. In other cases, multiple copies of the same circuit are provided within a single device. This latter tactic originally appeared in the form of chips containing several independent gates or perhaps multiple flip-flops. Today's state of the art allows for more than one entire CPU to be placed on a single chip. This is the underlying architecture of devices known as multi-core CPUs, which consist of two or more CPUs residing on the same chip along with shared cache memory. (Multi-core CPUs containing two processing units are typically called dual-core CPUs.) Such devices simplify the construction of MIMD systems and are readily available for use in home computers.

Multiprocessor Machines

Pipelining can be viewed as a first step toward **parallel processing,** which is the performance of several activities at the same time. However, true parallel processing requires more than one processing unit, resulting in computers known as multiprocessor or **multi-core** machines.

Most computers today are designed with this idea in mind. One strategy is to attach several processing units, each resembling the CPU in a single-processor machine, to the same main memory. In this configuration, the processors can proceed independently yet coordinate their efforts by leaving messages to one another in the common memory cells. For instance, when one processor is faced with a large task, it can store a program for part of that task in the common memory and then request another processor to execute it. The result is a machine in which different instruction sequences are performed on different sets of data, which is called a **MIMD** (**multiple-instruction stream, multiple-data stream**) architecture, as opposed to the more traditional **SISD** (**single-instruction stream, single-data stream**) architecture.

A variation of multiple-processor architecture is to link the processors together so that they execute the same sequence of instructions in unison, each with its own set of data. This leads to a **SIMD** (**single-instruction stream, multiple-data stream**) architecture. Such machines are useful in applications in which the same task must be applied to each set of similar items within a large block of data. Another approach to parallel processing is to construct large computers as conglomerates of smaller machines, each with its own memory and CPU. Within such an architecture, each of the small machines is coupled to its neighbors so that tasks assigned to the whole system can be divided among the individual machines. Thus if a task assigned to one of the internal machines can be broken into independent subtasks, that machine can ask its neighbors to perform these subtasks concurrently. The original task can then be completed in much less time than would be required by a single-processor machine.

Questions & Exercises

1. Referring back to question 3 of Section 2.3, if the machine used the pipeline technique discussed in the text, what will be in "the pipe" when the instruction at address AA is executed? Under what conditions would pipelining not prove beneficial at this point in the program?
2. What conflicts must be resolved in running the program in question 4 of Section 2.3 on a pipeline machine?
3. Suppose there were two "central" processing units attached to the same memory and executing different programs. Furthermore, suppose that one of these processors needs to add one to the contents of a memory cell at roughly the same time that the other needs to subtract one from the same cell. (The net effect should be that the cell ends up with the same value with which it started.)
   a. Describe a sequence in which these activities would result in the cell ending up with a value one less than its starting value.
   b. Describe a sequence in which these activities would result in the cell ending up with a value one greater than its starting value.

Chapter Review Problems

(Asterisked problems are associated with optional sections.)

1. a. In what way are general-purpose registers and main memory cells similar?
   b. In what way do general-purpose registers and main memory cells differ?

2. Answer the following questions in terms of the machine language described in Appendix C.
   a. Write the instruction 2304 (hexadecimal) as a string of 16 bits.
   b. Write the op-code of the instruction B2A5 (hexadecimal) as a string of 4 bits.
   c. Write the operand field of the instruction B2A5 (hexadecimal) as a string of 12 bits.

3. Suppose a block of data is stored in the memory cells of the machine described in Appendix C from address 98 to A2, inclusive. How many memory cells are in this block? List their addresses.

4. What is the value of the program counter in the machine described in Appendix C immediately after executing the instruction B0CD?

5. Suppose the memory cells at addresses 00 through 05 in the machine described in Appendix C contain the following bit patterns:

Address	Contents
00	22
01	11
02	32
03	02
04	C0
05	00

   Assuming that the program counter initially contained 00, record the contents of the program counter, instruction register, and memory cell at address 02 at the end of each fetch phase of the machine cycle until the machine halts.

6. Suppose three values x, y, and z are stored in a machine's memory. Describe the sequence of events (loading registers from memory, saving values in memory, and so on) that leads to the computation of x + y + z. How about (2x) + y?

7. The following are instructions written in the machine language described in Appendix C. Translate them into English.
   a. 7123 b. 40E1 c. A304
   d. B100 e. 2BCD

8. Suppose a machine language is designed with an op-code field of 4 bits. How many different instruction types can the language contain? What if the op-code field is increased to 6 bits?

9. Translate the following instructions from English into the machine language described in Appendix C.
   a. LOAD register 6 with the hexadecimal value 77.
   b. LOAD register 7 with the contents of memory cell 77.
   c. JUMP to the instruction at memory location 24 if the contents of register 0 equals the value in register A.
   d. ROTATE register 4 three bits to the right.
   e. AND the contents of registers E and 2 leaving the result in register 1.

10. Rewrite the program in Figure 2.7 assuming that the values to be added are encoded using floating-point notation rather than two's complement notation.

11. Classify each of the following instructions (in the machine language of Appendix C) in terms of whether its execution changes the contents of the memory cell at location 3B, retrieves the contents of the memory cell at location 3C, or is independent of the contents of the memory cell at location 3C.
   a. 353C b. 253C c. 153C
   d. 3C3C e. 403C

12. Suppose the memory cells at addresses 00 through 03 in the machine described in Appendix C contain the following bit patterns:

Address	Contents
00	26
01	55
02	C0
03	00

   a. Translate the first instruction into English.
   b. If the machine is started with its program counter containing 00, what bit pattern is in register 6 when the machine halts?

13. Suppose the memory cells at addresses 00 through 02 in the machine described in Appendix C contain the following bit patterns:

Address	Contents
00	12
01	21
02	34

a. What would be the first instruction executed if we started the machine with its program counter containing 00?
b. What would be the first instruction executed if we started the machine with its program counter containing 01?

14. Suppose the memory cells at addresses 00 through 05 in the machine described in Appendix C contain the following bit patterns:

Address	Contents
00	12
01	02
02	32
03	42
04	C0
05	00

When answering the following questions, assume that the machine starts with its program counter equal to 00.
a. Translate the instructions that are executed into English.
b. What bit pattern is in the memory cell at address 42 when the machine halts?
c. What bit pattern is in the program counter when the machine halts?

15. Suppose the memory cells at addresses 00 through 09 in the machine described in Appendix C contain the following bit patterns:

Address	Contents
00	1C
01	03
02	2B
03	03
04	5A
05	BC
06	3A
07	00
08	C0
09	00

Assume that the machine starts with its program counter containing 00.
a. What will be in the memory cell at address 00 when the machine halts?
b. What bit pattern will be in the program counter when the machine halts?

16. Suppose the memory cells at addresses 00 through 07 in the machine described in Appendix C contain the following bit patterns:

Address	Contents
00	2B
01	07
02	3B
03	06
04	C0
05	00
06	00
07	23

a. List the addresses of the memory cells that contain the program that will be executed if we start the machine with its program counter containing 00.
b. List the addresses of the memory cells that are used to hold data.

17. Suppose the memory cells at addresses 00 through 0D in the machine described in Appendix C contain the following bit patterns:

Address	Contents
00	20
01	04
02	21
03	01
04	40
05	12
06	51
07	12
08	B1
09	0C
0A	B0
0B	06
0C	C0
0D	00

Assume that the machine starts with its program counter containing 00.
a. What bit pattern will be in register 0 when the machine halts?

b. What bit pattern will be in register 1 when the machine halts?
c. What bit pattern is in the program counter when the machine halts?

18. Suppose the memory cells at addresses F0 through FD in the machine described in Appendix C contain the following (hexadecimal) bit patterns:

Address	Contents
F0	20
F1	00
F2	22
F3	02
F4	23
F5	04
F6	B3
F7	FC
F8	50
F9	02
FA	B0
FB	F6
FC	C0
FD	00

If we start the machine with its program counter containing F0, what is the value in register 0 when the machine finally executes the halt instruction at location FC?

19. If the machine in Appendix C executes an instruction every microsecond (a millionth of a second), how long does it take to complete the program in Problem 18?

20. Suppose the memory cells at addresses 20 through 28 in the machine described in Appendix C contain the following bit patterns:

Address	Contents
20	12
21	20
22	32
23	30
24	B0
25	21
26	24
27	C0
28	00

Assume that the machine starts with its program counter containing 20.
a. What bit patterns will be in registers 0, 1, and 2 when the machine halts?
b. What bit pattern will be in the memory cell at address 30 when the machine halts?
c. What bit pattern will be in the memory cell at address B0 when the machine halts?

21. Suppose the memory cells at addresses AF through B1 in the machine described in Appendix C contain the following bit patterns:

Address	Contents
AF	B0
B0	B0
B1	AF

What would happen if we started the machine with its program counter containing AF?

22. Suppose the memory cells at addresses 00 through 05 in the machine described in Appendix C contain the following (hexadecimal) bit patterns:

Address	Contents
00	25
01	B0
02	35
03	04
04	C0
05	00

If we start the machine with its program counter containing 00, when does the machine halt?

23. In each of the following cases, write a short program in the machine language described in Appendix C to perform the requested activities. Assume that each of your programs is placed in memory starting at address 00.
a. Move the value at memory location D8 to memory location B3.
b. Interchange the values stored at memory locations D8 and B3.
c. If the value stored in memory location 44 is 00, then place the value 01 in memory location 46; otherwise, put the value FF in memory location 46.

24. A game that used to be popular among computer hobbyists is core wars—a variation of battleship. (The term *core* originates from an early memory technology in which 0s and 1s were represented as magnetic fields in little rings of magnetic material. The rings were called cores.) The game is played between two opposing programs, each stored in different locations of the same computer's memory. The computer is assumed to alternate between the two programs, executing an instruction from one followed by an instruction from the other. The goal of each program is to cause the other to malfunction by writing extraneous data on top of it; however, neither program knows the location of the other.

a. Write a program in the machine language of Appendix C that approaches the game in a defensive manner by being as small as possible.

b. Write a program in the language of Appendix C that tries to avoid any attacks from the opposing program by moving to different locations. More precisely, beginning at location 00, write a program that will copy itself to location 70 and then jump to location 70.

c. Extend the program in (b) to continue relocating to new memory locations. In particular, make your program move to location 70, then to E0 (= 70 + 70) then to 60 (= 70 + 70 + 70) etc.

25. Write a program in the machine language of Appendix C to compute the sum of floating-point values stored at memory locations A0, A1, A2, and A3. Your program should store the total at memory location A4.

26. Suppose the memory cells at addresses 00 through 05 in the machine described in Appendix C contain the following (hexadecimal) bit patterns:

Address	Contents
00	20
01	C0
02	30
03	04
04	00
05	00

What happens if we start the machine with its program counter containing 00?

27. What happens if the memory cells at addresses 08 and 09 of the machine described in Appendix C contain the bit patterns B0 and 08, respectively, and the machine is started with its program counter containing the value 08?

28. Suppose the following program, written in the machine language of Appendix C, is stored in main memory beginning at address 30 (hexadecimal). What task will the program perform when executed?

```
2003
2101
2200
2310
1400
3410
5221
5331
3239
333B
B248
B038
C000
```

29. Summarize the steps involved when the machine described in Appendix C performs an instruction with op-code B. Express your answer as a set of directions as though you were telling the CPU what to do.

***30.** Summarize the steps involved when the machine described in Appendix C performs an instruction with op-code 5. Express your answer as a set of directions as though you were telling the CPU what to do.

***31.** Summarize the steps involved when the machine described in Appendix C performs an instruction with op-code 6. Express your answer as a set of directions as though you were telling the CPU what to do.

***32.** Suppose the registers 4 and 5 in the machine described in Appendix C contain the bit patterns 3A and C8, respectively. What bit pattern is left in register 0 after executing each of the following instructions:

a. 5045 b. 6045 c. 7045
d. 8045 e. 9045

*33. Using the machine language described in Appendix C, write programs to perform each of the following tasks:

a. Copy the bit pattern stored in memory location 44 into memory location AA.

b. Change the least significant 4 bits in the memory cell at location 34 to 0s while leaving the other bits unchanged.

c. Copy the least significant 4 bits from memory location A5 into the least significant 4 bits of location A6 while leaving the other bits at location A6 unchanged.

d. Copy the least significant 4 bits from memory location A5 into the most significant 4 bits of A5. (Thus, the first 4 bits in A5 will be the same as the last 4 bits.)

*34. Perform the indicated operations:

a.
```
  111001
AND 101001
```

b.
```
  000101
AND 101010
```

c.
```
  001110
AND 010101
```

d.
```
  111011
AND 110111
```

e.
```
  111001
OR  101001
```

f.
```
  010100
OR  101010
```

g.
```
  000100
OR  010101
```

h.
```
  101010
OR  110101
```

i.
```
  111001
XOR 101001
```

j.
```
  000111
XOR 101010
```

k.
```
  010000
XOR 010101
```

l.
```
  111111
XOR 110101
```

*35. Identify both the mask and the logical operation needed to accomplish each of the following objectives:

a. Put 1s in the upper 4 bits of an 8-bit pattern without disturbing the other bits.

b. Complement the most significant bit of an 8-bit pattern without changing the other bits.

c. Complement a pattern of 8 bits.

d. Put a 0 in the least significant bit of an 8-bit pattern without disturbing the other bits.

e. Put 1s in all but the most significant bit of an 8-bit pattern without disturbing the most significant bit.

f. Filter out all of the green color component from an RGB bitmap image pixel in which the middle 8 bits of a 24-bit pattern store the green information.

g. Invert all of the bits in a 24-bit RGB bitmap pixel.

h. Set all the bits in a 24-bit RGB bitmap pixel to 1, indicating the color "white".

*36. Write and test short Python scripts to implement each of the parts of the previous question.

*37. Identify a logical operation (along with a corresponding mask) that, when applied to an input string of 8 bits, produces an output string of all 0s if and only if the input string is 10000001.

*38. Write and test a short Python script to implement the previous question.

*39. Describe a sequence of logical operations (along with their corresponding masks) that, when applied to an input string of 8 bits, produces an output byte of all 0s if the input string both begins and ends with 1s. Otherwise, the output should contain at least one 1.

*40. Write and test a short Python script to implement the previous question.

*41. What would be the result of performing a 4-bit left circular shift on the following bit patterns?

a. 10101 b. 11110000 c. 001
d. 101000 e. 00001

*42. What would be the result of performing a 2-bit right circular shift on the following bytes represented in hexadecimal notation (give your answers in hexadecimal notation)?

a. 3F b. 0D
c. FF d. 77

*43. a. What single instruction in the machine language of Appendix C could be used to accomplish a 5-bit right circular shift of register B?

b. What single instruction in the machine language of Appendix C could be used to accomplish a 2-bit left circular shift of register B?

*44. Write a program in the machine language of Appendix C that reverses the contents of the memory cell at address 8C. (That is, the final bit pattern at address 8C when read from left to right should agree with the original pattern when read from right to left.)

*45. Write a program in the machine language of Appendix C that subtracts the value stored at A1 from the value stored at address A2 and places the result at address A0. Assume that the values are encoded in two's complement notation.

*46. High definition video can be delivered at a rate of 30 frames per second (fps) where each frame has a resolution of 1920 x 1080 pixels using 24 bits per pixel. Can an uncompressed video stream of this format be sent over a USB 1.1 serial port? USB 2.0 serial port? USB 3.0 serial port? (Note: The maximum speeds of USB 1.1, USB 2.0, and USB 3.0 serial ports are 12Mbps, 480Mbps, and 5Gbps respectively.)

*47. Suppose a person is typing forty words per minute at a keyboard. (A word is considered to be five characters.) If a machine executes 500 instructions every microsecond (millionth of a second), how many instructions does the machine execute during the time between the typing of two consecutive characters?

*48. How many bits per second must a keyboard transmit to keep up with a typist typing forty words per minute? (Assume each character is encoded in ASCII and each word consists of six characters.)

*49. Suppose the machine described in Appendix C communicates with a printer using the technique of memory-mapped I/O. Suppose also that address FF is used to send characters to the printer, and address FE is used to receive information about the printer's status. In particular, suppose the least significant bit at the address FE indicates whether the printer is ready to receive another character (with a 0 indicating "not ready" and a 1 indicating "ready"). Starting at address 00, write a machine language routine that waits until the printer is ready for another character and then sends the character represented by the bit pattern in register 5 to the printer.

*50. Write a program in the machine language described in Appendix C that places 0s in all the memory cells from address A0 through C0 but is small enough to fit in the memory cells from address 00 through 13 (hexadecimal).

*51. Suppose a machine has 200 GB of storage space available on a hard disk and receives data over a broadband connection at the rate of 15 Mbps. At this rate, how long will it take to fill the available storage space?

*52. Suppose a satellite system is being used to receive a serial data stream at 250 Kbps. If a burst of atmospheric interference lasts 6.96 seconds, how many data bits will be affected?

*53. Suppose you are given 32 processors, each capable of finding the sum of two multidigit numbers in a millionth of a second. Describe how parallel processing techniques can be applied to find the sum of 64 numbers in only six-millionths of a second. How much time does a single processor require to find this same sum?

*54. Summarize the difference between a CISC architecture and a RISC architecture.

*55. Identify two approaches to increasing throughput.

*56. Describe how the average of a collection of numbers can be computed more rapidly with a multiprocessor machine than a single processor machine.

*57. Write and test a Python script that reads in a floating-point radius of a circle and outputs the circumference and area of the circle.

*58. Write and test a Python script that reads in a character string and an integer and outputs the character string repeated the number of times given by the integer.

*59. Write and test a Python script that reads in two floating-point side lengths of a right triangle and outputs the hypotenuse length, perimeter, and area.

Social Issues

The following questions are intended as a guide to the ethical/social/legal issues associated with the field of computing. The goal is not merely to answer these questions. You should also consider why you answered as you did and whether your justifications are consistent from one question to the next.

1. Suppose a computer manufacturer develops a new machine architecture. To what extent should the company be allowed to own that architecture? What policy would be best for society?
2. In a sense, the year 1923 marked the birth of what many now call *planned obsolescence.* This was the year that General Motors, led by Alfred Sloan, introduced the automobile industry to the concept of model years. The idea was to increase sales by changing styling rather than necessarily introducing a better automobile. Sloan is quoted as saying, "We want to make you dissatisfied with your current car so you will buy a new one." To what extent is this marketing ploy used today in the computer industry?
3. We often think in terms of how computer technology has changed our society. Many argue, however, that this technology has often kept changes from occurring by allowing old systems to survive and, in some cases, become more entrenched. For example, would a central government's role in society have survived without computer technology? To what extent would centralized authority be present today had computer technology not been available? To what extent would we be better or worse off without computer technology?
4. Is it ethical for an individual to take the attitude that he or she does not need to know anything about the internal details of a machine because someone else will build it, maintain it, and fix any problems that arise? Does your answer depend on whether the machine is a computer, automobile, nuclear power plant, or toaster?
5. Suppose a manufacturer produces a computer chip and later discovers a flaw in its design. Suppose further that the manufacturer corrects the flaw in future production but decides to keep the original flaw a secret and does not recall the chips already shipped, reasoning that none of the chips already in use are being used in an application in which the flaw will have consequences. Is anyone hurt by the manufacturer's decision? Is the manufacturer's decision justified if no one is hurt and the decision keeps the manufacturer from losing money and possibly having to lay off employees?
6. Does advancing technology provide cures for heart disease or is it a source of a sedentary life style that contributes to heart disease?
7. It is easy to imagine financial or navigational disasters that may occur as the result of arithmetic errors due to overflow and truncation problems. What consequences could result from errors in image storage systems due to loss of image details (perhaps in fields such as reconnaissance or medical diagnosis)?
8. ARM Holdings is a small company that designs the processors for a wide variety of consumer electronic devices. It does not manufacture any of the processors; instead the designs are licensed to semiconductor vendors (such as Qualcomm, Samsung, and Texas Instruments) who pay a royalty for each unit produced. This business model spreads the high cost of research and

development of computer processors across the entire consumer electronic market. Today, over 95 percent of all cellular phones (not just smartphones), over 40 percent of all digital cameras, and 25 percent of digital TVs use an ARM processor. Furthermore, ARM processors are found in mini-notebooks, MP3 players, game controllers, electronic book readers, navigation systems, and the list goes on. Given this, do you consider this company to be a monopoly? Why or why not? Because consumer devices play an ever-increasing role in today's society, is the dependency on this little-known company good, or does it raise concerns?

Additional Reading

Carpinelli, J. D. *Computer Systems Organization and Architecture.* Boston, MA: Addison-Wesley, 2001.

Comer, D. E. *Essentials of Computer Architecture.* Upper Saddle River, NJ: Prentice-Hall, 2005.

Dandamudi, S P. *Guide to RISC Processors for Programmers and Engineers.* New York: Springer, 2005.

Furber, S. *ARM System-on-Chip Architecture,* 2nd ed. Boston, MA: Addison Wesley, 2000.

Hamacher, V. C., Z. G. Vranesic, and S. G. Zaky. *Computer Organization,* 5th ed. New York: McGraw-Hill, 2002.

Knuth, D. E. *The Art of Computer Programming, Vol. 1,* 3rd ed. Boston, MA: Addison-Wesley, 1998.

Murdocca, M. J., and V. P. Heuring. *Computer Architecture and Organization: An Integrated Approach.* New York: Wiley, 2007.

Stallings, W. *Computer Organization and Architecture,* 9th ed. Upper Saddle River, NJ: Prentice-Hall, 2012.

Tanenbaum, A. S. *Structured Computer Organization,* 6th ed. Upper Saddle River, NJ: Prentice-Hall, 2012.