CHAPTER 1

Data Storage

In this chapter, we consider topics associated with data representation and the storage of data within a computer. The types of data we will consider include text, numeric values, images, audio, and video. Much of the information in this chapter is also relevant to fields other than traditional computing, such as digital photography, audio/video recording and reproduction, and long-distance communication.

**Asterisks indicate suggestions for optional sections.*

We begin our study of computer science by considering how information is encoded and stored inside computers. Our first step is to discuss the basics of a computer's data storage devices and then to consider how information is encoded for storage in these systems. We will explore the ramifications of today's data storage systems and how such techniques as data compression and error handling are used to overcome their shortfalls.

1.1 Bits and Their Storage

Inside today's computers information is encoded as patterns of 0s and 1s. These digits are called **bits** (short for *binary digits*). Although you may be inclined to associate bits with numeric values, they are really only symbols whose meaning depends on the application at hand. Sometimes patterns of bits are used to represent numeric values; sometimes they represent characters in an alphabet and punctuation marks; sometimes they represent images; and sometimes they represent sounds.

Boolean Operations

To understand how individual bits are stored and manipulated inside a computer, it is convenient to imagine that the bit 0 represents the value *false* and the bit 1 represents the value *true.* Operations that manipulate true/false values are called **Boolean operations,** in honor of the mathematician George Boole (1815–1864), who was a pioneer in the field of mathematics called logic. Three of the basic Boolean operations are AND, OR, and XOR (exclusive or) as summarized in Figure 1.1. (We capitalize these Boolean operation names to distinguish them from their English word counterparts.) These operations are similar to the arithmetic operations TIMES and PLUS because they combine a pair of values (the operation's input) to produce a third value (the output). In contrast to arithmetic operations, however, Boolean operations combine true/false values rather than numeric values.

The Boolean operation AND is designed to reflect the truth or falseness of a statement formed by combining two smaller, or simpler, statements with the conjunction *and.* Such statements have the generic form

P AND *Q*

where *P* represents one statement, and *Q* represents another—for example,

Kermit is a frog AND Miss Piggy is an actress.

The inputs to the AND operation represent the truth or falseness of the compound statement's components; the output represents the truth or falseness of the compound statement itself. Since a statement of the form *P* AND *Q* is true only when both of its components are true, we conclude that 1 AND 1 should be 1, whereas all other cases should produce an output of 0, in agreement with Figure 1.1.

In a similar manner, the OR operation is based on compound statements of the form

P OR *Q*

where, again, *P* represents one statement and *Q* represents another. Such statements are true when at least one of their components is true, which agrees with the OR operation depicted in Figure 1.1.

Figure 1.1 The possible input and output values of Boolean operations AND, OR, and XOR (exclusive or)

The AND operation

$$\begin{array}{r} 0 \\ \underline{\text{AND}\quad 0} \\ 0 \end{array} \qquad \begin{array}{r} 0 \\ \underline{\text{AND}\quad 1} \\ 0 \end{array} \qquad \begin{array}{r} 1 \\ \underline{\text{AND}\quad 0} \\ 0 \end{array} \qquad \begin{array}{r} 1 \\ \underline{\text{AND}\quad 1} \\ 1 \end{array}$$

The OR operation

$$\begin{array}{r} 0 \\ \underline{\text{OR}\quad 0} \\ 0 \end{array} \qquad \begin{array}{r} 0 \\ \underline{\text{OR}\quad 1} \\ 1 \end{array} \qquad \begin{array}{r} 1 \\ \underline{\text{OR}\quad 0} \\ 1 \end{array} \qquad \begin{array}{r} 1 \\ \underline{\text{OR}\quad 1} \\ 1 \end{array}$$

The XOR operation

$$\begin{array}{r} 0 \\ \underline{\text{XOR}\quad 0} \\ 0 \end{array} \qquad \begin{array}{r} 0 \\ \underline{\text{XOR}\quad 1} \\ 1 \end{array} \qquad \begin{array}{r} 1 \\ \underline{\text{XOR}\quad 0} \\ 1 \end{array} \qquad \begin{array}{r} 1 \\ \underline{\text{XOR}\quad 1} \\ 0 \end{array}$$

There is not a single conjunction in the English language that captures the meaning of the XOR operation. XOR produces an output of 1 (true) when one of its inputs is 1 (true) and the other is 0 (false). For example, a statement of the form *P* XOR *Q* means "either *P* or *Q* but not both." (In short, the XOR operation produces an output of 1 when its inputs are different.)

The operation NOT is another Boolean operation. It differs from AND, OR, and XOR because it has only one input. Its output is the opposite of that input; if the input of the operation NOT is true, then the output is false, and vice versa. Thus, if the input of the NOT operation is the truth or falseness of the statement

Fozzie is a bear.

then the output would represent the truth or falseness of the statement

Fozzie is not a bear.

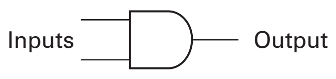
Gates and Flip-Flops

A device that produces the output of a Boolean operation when given the operation's input values is called a **gate.** Gates can be constructed from a variety of technologies such as gears, relays, and optic devices. Inside today's computers, gates are usually implemented as small electronic circuits in which the digits 0 and 1 are represented as voltage levels. We need not concern ourselves with such details, however. For our purposes, it suffices to represent gates in their symbolic form, as shown in Figure 1.2. Note that the AND, OR, XOR, and NOT gates are represented by distinctively shaped symbols, with the input values entering on one side, and the output exiting on the other.

Gates provide the building blocks from which computers are constructed. One important step in this direction is depicted in the circuit in Figure 1.3. This is a particular example from a collection of circuits known as a **flip-flop.** A flip-flop

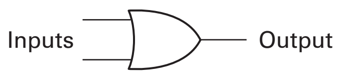
Figure 1.2 A pictorial representation of AND, OR, XOR, and NOT gates as well as their input and output values

AND



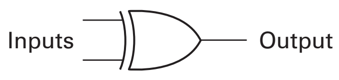
Inputs	Output
0 0	0
0 1	0
1 0	0
1 1	1

OR



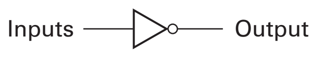
Inputs	Output
0 0	0
0 1	1
1 0	1
1 1	1

XOR



Inputs	Output
0 0	0
0 1	1
1 0	1
1 1	0

NOT

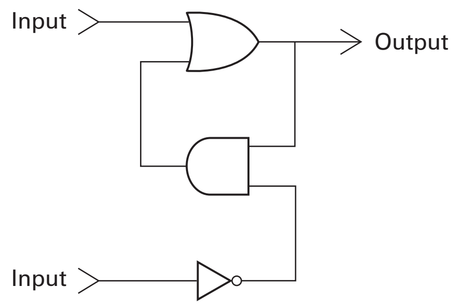


Inputs	Output
0	1
1	0

is a fundamental unit of computer memory. It is a circuit that produces an output value of 0 or 1, which remains constant until a pulse (a temporary change to a 1 that returns to 0) from another circuit causes it to shift to the other value. In other words, the output can be set to "remember" a zero or a one under control of external stimuli. As long as both inputs in the circuit in Figure 1.3 remain 0, the output (whether 0 or 1) will not change. However, temporarily placing a 1 on the upper input will force the output to be 1, whereas temporarily placing a 1 on the lower input will force the output to be 0.

Let us consider this claim in more detail. Without knowing the current output of the circuit in Figure 1.3, suppose that the upper input is changed to 1 while the lower input remains 0 (Figure 1.4a). This will cause the output of the OR gate to be 1, regardless of the other input to this gate. In turn, both inputs to the AND gate will now be 1, since the other input to this gate is already 1 (the output produced by the NOT gate whenever the lower input of the flip-flop is at 0). The output of the AND gate will then become 1, which means that the second input to the OR gate will now be 1 (Figure 1.4b). This guarantees that the output of the OR gate will remain 1, even when the upper input to the flip-flop is changed back to 0 (Figure 1.4c). In summary, the flip-flop's output has become 1, and this output value will remain after the upper input returns to 0.

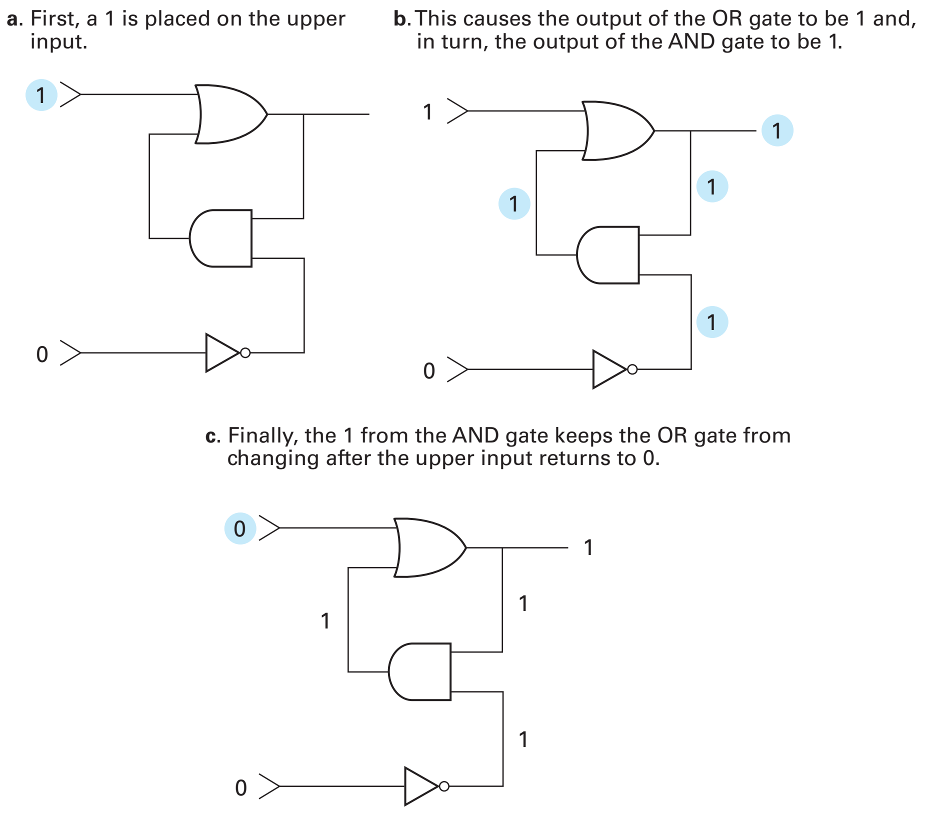
Figure 1.3 A simple flip-flop circuit



In a similar manner, temporarily placing the value 1 on the lower input will force the flip-flop's output to be 0, and this output will persist after the input value returns to 0.

Our purpose in introducing the flip-flop circuit in Figures 1.3 and 1.4 is threefold. First, it demonstrates how devices can be constructed from gates, a process known as digital circuit design, which is an important topic in computer

Figure 1.4 Setting the output of a flip-flop to 1


a. First, a 1 is placed on the upper input.
1
0
b. This causes the output of the OR gate to be 1 and, in turn, the output of the AND gate to be 1.
1
1
1
1
1
0
c. Finally, the 1 from the AND gate keeps the OR gate from changing after the upper input returns to 0.
0
1
1
1
1
0

engineering. Indeed, the flip-flop is only one of many circuits that are basic tools in computer engineering.

Second, the concept of a flip-flop provides an example of abstraction and the use of abstract tools. Actually, there are other ways to build a flip-flop. One alternative is shown in Figure 1.5. If you experiment with this circuit, you will find that, although it has a different internal structure, its external properties are the same as those of Figure 1.3. A computer engineer does not need to know which circuit is actually used within a flip-flop. Instead, only an understanding of the flip-flop's external properties is needed to use it as an abstract tool. A flip-flop, along with other well-defined circuits, forms a set of building blocks from which an engineer can construct more complex circuitry. In turn, the design of computer circuitry takes on a hierarchical structure, each level of which uses the lower level components as abstract tools.

The third purpose for introducing the flip-flop is that it is one means of storing a bit within a modern computer. More precisely, a flip-flop can be set to have the output value of either 0 or 1. Other circuits can adjust this value by sending pulses to the flip-flop's inputs, and still other circuits can respond to the stored value by using the flip-flop's output as their inputs. Thus, many flip-flops, constructed as very small electrical circuits, can be used inside a computer as a means of recording information that is encoded as patterns of 0s and 1s. Indeed, technology known as **very large-scale integration (VLSI),** which allows millions of electrical components to be constructed on a wafer (called a **chip**), is used to create miniature devices containing millions of flip-flops along with their controlling circuitry. Consequently, these chips are used as abstract tools in the construction of computer systems. In fact, in some cases VLSI is used to create an entire computer system on a single chip.

Hexadecimal Notation

When considering the internal activities of a computer, we must deal with patterns of bits, which we will refer to as a string of bits, some of which can be quite long. A long string of bits is often called a **stream.** Unfortunately, streams are difficult for the human mind to comprehend. Merely transcribing the pattern 101101010011 is tedious and error prone. To simplify the representation of such bit patterns, therefore, we usually use a shorthand notation called **hexadecimal notation,** which takes advantage of the fact that bit patterns within a machine

Figure 1.5 Another way of constructing a flip-flop

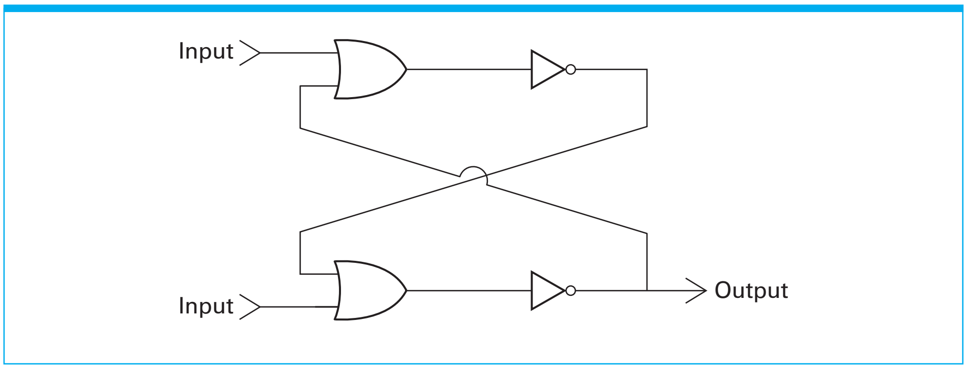


Figure 1.6 The hexadecimal encoding system

Bit pattern	Hexadecimal representation
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

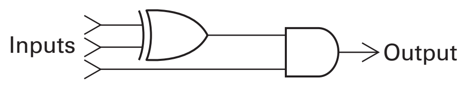
tend to have lengths in multiples of four. In particular, hexadecimal notation uses a single symbol to represent a pattern of four bits. For example, a string of twelve bits can be represented by three hexadecimal symbols.

Figure 1.6 presents the hexadecimal encoding system. The left column displays all possible bit patterns of length four; the right column shows the symbol used in hexadecimal notation to represent the bit pattern to its left. Using this system, the bit pattern 10110101 is represented as B5. This is obtained by dividing the bit pattern into substrings of length four and then representing each substring by its hexadecimal equivalent—1011 is represented by B, and 0101 is represented by 5. In this manner, the 16-bit pattern 1010010011001000 can be reduced to the more palatable form A4C8.

We will use hexadecimal notation extensively in the next chapter. There you will come to appreciate its efficiency.

Questions & Exercises

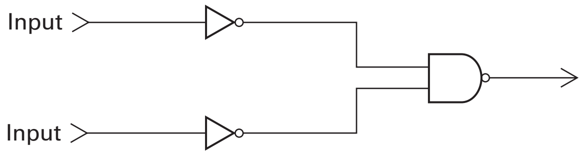
1. What input bit patterns will cause the following circuit to produce an output of 1?



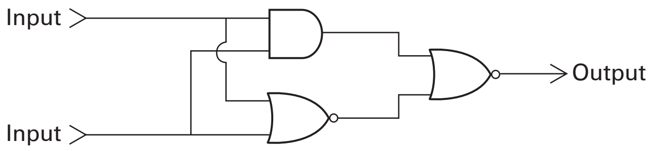
2. In the text, we claimed that placing a 1 on the lower input of the flip-flop in Figure 1.3 (while holding the upper input at 0) will force the flip-flop's output to be 0. Describe the sequence of events that occurs within the flip-flop in this case.

3. Assuming that both inputs to the flip-flop in Figure 1.5 begin as 0, describe the sequence of events that occurs when the upper input is temporarily set to 1.

4. a. If the output of an AND gate is passed through a NOT gate, the combination computes the Boolean operation called NAND, which has an output of 0 only when both its inputs are 1. The symbol for a NAND gate is the same as an AND gate except that it has a circle at its output. The following is a circuit containing a NAND gate. What Boolean operation does the circuit compute?



b. If the output of an OR gate is passed through a NOT gate, the combination computes the Boolean operation called NOR that has an output of 1 only when both its inputs are 0. The symbol for a NOR gate is the same as an OR gate except that it has a circle at its output. The following is a circuit containing an AND gate and two NOR gates. What Boolean operation does the circuit compute?



5. Use hexadecimal notation to represent the following bit patterns:

a. 0110101011110010 b. 111010000101010100010111

c. 01001000

6. What bit patterns are represented by the following hexadecimal patterns?

a. 5FD97 b. 610A c. ABCD d. 0100

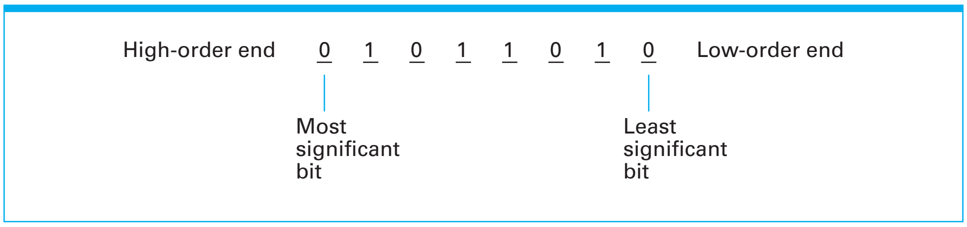
1.2 Main Memory

For the purpose of storing data, a computer contains a large collection of circuits (such as flip-flops), each capable of storing a single bit. This bit reservoir is known as the machine's **main memory.**

Memory Organization

A computer's main memory is organized in manageable units called **cells,** with a typical cell size being eight bits. (A string of eight bits is called a **byte.** Thus, a typical memory cell has a capacity of one byte.) Small computers embedded in such household devices as microwave ovens may have main memories consisting

Figure 1.7 The organization of a byte-size memory cell



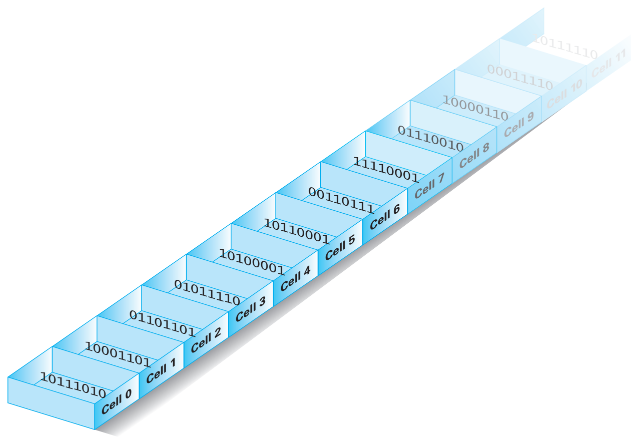
of only a few hundred cells, whereas large computers may have billions of cells in their main memories.

Although there is no left or right within a computer, we normally envision the bits within a memory cell as being arranged in a row. The left end of this row is called the **high-order end,** and the right end is called the **low-order end.** The left-most bit is called either the high-order bit or the **most significant bit** in reference to the fact that if the contents of the cell were interpreted as representing a numeric value, this bit would be the most significant digit in the number. Similarly, the rightmost bit is referred to as the low-order bit or the **least significant bit.** Thus we may represent the contents of a byte-size memory cell as shown in Figure 1.7.

To identify individual cells in a computer's main memory, each cell is assigned a unique "name," called its **address.** The system is analogous to the technique of identifying houses in a city by addresses. In the case of memory cells, however, the addresses used are entirely numeric. To be more precise, we envision all the cells being placed in a single row and numbered in this order starting with the value zero. Such an addressing system not only gives us a way of uniquely identifying each cell but also associates an order to the cells (Figure 1.8), giving us phrases such as "the next cell" or "the previous cell."

An important consequence of assigning an order to both the cells in main memory and the bits within each cell is that the entire collection of bits within a computer's main memory is essentially ordered in one long row. Pieces of this long row can therefore be used to store bit patterns that may be longer than the

Figure 1.8 Memory cells arranged by address



length of a single cell. In particular, we can still store a string of 16 bits merely by using two consecutive memory cells.

To complete the main memory of a computer, the circuitry that actually holds the bits is combined with the circuitry required to allow other circuits to store and retrieve data from the memory cells. In this way, other circuits can get data from the memory by electronically asking for the contents of a certain address (called a read operation), or they can record information in the memory by requesting that a certain bit pattern be placed in the cell at a particular address (called a write operation).

Because a computer's main memory is organized as individual, addressable cells, the cells can be accessed independently as required. To reflect the ability to access cells in any order, a computer's main memory is often called **random access memory (RAM).** This random access feature of main memory is in stark contrast to the mass storage systems that we will discuss in the next section, in which long strings of bits are manipulated as amalgamated blocks.

Although we have introduced flip-flops as a means of storing bits, the RAM in most modern computers is constructed using analogous, but more complex technologies that provide greater miniaturization and faster response time. Many of these technologies store bits as tiny electric charges that dissipate quickly. Thus these devices require additional circuitry, known as a refresh circuit, that repeatedly replenishes the charges many times a second. In recognition of this volatility, computer memory constructed from such technology is often called **dynamic memory,** leading to the term **DRAM** (pronounced "DEE–ram") meaning Dynamic RAM. Or, at times the term **SDRAM** (pronounced "ES-DEE-ram"), meaning Synchronous DRAM, is used in reference to DRAM that applies additional techniques to decrease the time needed to retrieve the contents from its memory cells.

Measuring Memory Capacity

As we will learn in the next chapter, it is convenient to design main memory systems in which the total number of cells is a power of two. In turn, the size of the memories in early computers were often measured in 1024 (which is 2^{10}) cell units. Since 1024 is close to the value 1000, the computing community adopted the prefix *kilo* in reference to this unit. That is, the term *kilobyte* (abbreviated KB) was used to refer to 1024 bytes. Thus, a machine with 4096 memory cells was said to have a 4KB memory ($4096 = 4 \times 1024$). As memories became larger, this terminology grew to include MB (megabyte), GB (gigabyte), and TB (terabyte). Unfortunately, this application of prefixes *kilo-, mega-,* and so on, represents a misuse of terminology because these are already used in other fields in reference to units that are powers of a thousand. For example, when measuring distance, *kilometer* refers to 1000 meters, and when measuring radio frequencies, *megahertz* refers to 1,000,000 hertz. In the late 1990s, international standards organizations developed specialized terminology for powers of two: *kibi*-, *mebi*-, *gibi*-, and *tebi*-bytes denote powers of 1024, rather than powers of a thousand. However, while this distinction is the law of the land in many parts of the world, both the general public and many computer scientists have been reluctant to abandon the more familiar, yet ambiguous "megabyte." Thus, a word of caution is in order when using this terminology. As a general rule, terms such as *kilo-, mega-,* etc. refer to powers of two when used in the context of computer measurements, but they refer to powers of a thousand when used in other contexts.

Questions & Exercises

1. If the memory cell whose address is 5 contains the value 8, what is the difference between writing the value 5 into cell number 6 and moving the contents of cell number 5 into cell number 6?
2. Suppose you want to interchange the values stored in memory cells 2 and 3. What is wrong with the following sequence of steps:

   Step 1. Move the contents of cell number 2 to cell number 3.

   Step 2. Move the contents of cell number 3 to cell number 2.

   Design a sequence of steps that correctly interchanges the contents of these cells. If needed, you may use additional cells.
3. How many bits would be in the memory of a computer with 4KB memory?

1.3 Mass Storage

Due to the volatility and limited size of a computer's main memory, most computers have additional memory devices called **mass storage** (or secondary storage) systems, including magnetic disks, CDs, DVDs, magnetic tapes, flash drives, and solid-state disks (all of which we will discuss shortly). The advantages of mass storage systems over main memory include less volatility, large storage capacities, low cost, and in many cases, the ability to remove the storage medium from the machine for archival purposes.

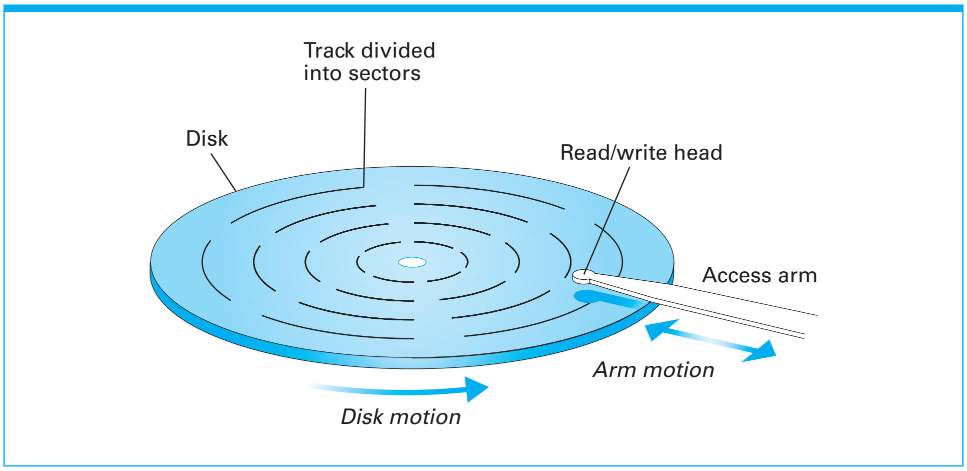
A major disadvantage of magnetic and optical mass storage systems is that they typically require mechanical motion and therefore require significantly more time to store and retrieve data than a machine's main memory, where all activities are performed electronically. Moreover, storage systems with moving parts are more prone to mechanical failures than solid state systems.

Magnetic Systems

For years, magnetic technology has dominated the mass storage arena. The most common example in use today is the **magnetic disk** or **hard disk drive (HDD),** in which a thin spinning disk with magnetic coating is used to hold data (Figure 1.9). Read/write heads are placed above and/or below the disk so that as the disk spins, each head traverses a circle, called a **track.** By repositioning the read/write heads, different concentric tracks can be accessed. In many cases, a disk storage system consists of several disks mounted on a common spindle, one on top of the other, with enough space for the read/write heads to slip between the platters. In such cases, the read/write heads move in unison. Each time the read/write heads are repositioned, a new set of tracks—which is called a **cylinder**—becomes accessible.

Since a track can contain more information than we would normally want to manipulate at any one time, each track is divided into small arcs called **sectors** on which information is recorded as a continuous string of bits. All sectors on a disk contain the same number of bits (typical capacities are in the range of 512 bytes to a few KB), and in the simplest disk storage systems each track contains the same

Figure 1.9 A disk storage system



number of sectors. Thus, the bits within a sector on a track near the outer edge of the disk are less compactly stored than those on the tracks near the center, since the outer tracks are longer than the inner ones. In contrast, in high-capacity disk storage systems, the tracks near the outer edge are capable of containing significantly more sectors than those near the center, and this capability is often used by applying a technique called **zoned-bit recording.** Using zoned-bit recording, several adjacent tracks are collectively known as zones, with a typical disk containing approximately 10 zones. All tracks within a zone have the same number of sectors, but each zone has more sectors per track than the zone inside of it. In this manner, efficient use of the entire disk surface is achieved. Regardless of the details, a disk storage system consists of many individual sectors, each of which can be accessed as an independent string of bits.

The capacity of a disk storage system depends on the number of platters used and the density in which the tracks and sectors are placed. Lower-capacity systems may consist of a single platter. High-capacity disk systems, capable of holding many gigabytes, or even terabytes, consist of perhaps three to six platters mounted on a common spindle. Furthermore, data may be stored on both the upper and lower surfaces of each platter.

Several measurements are used to evaluate a disk system's performance: (1) **seek time** (the time required to move the read/write heads from one track to another); (2) **rotation delay** or **latency time** (half the time required for the disk to make a complete rotation, which is the average amount of time required for the desired data to rotate around to the read/write head once the head has been positioned over the desired track); (3) **access time** (the sum of seek time and rotation delay); and (4) **transfer rate** (the rate at which data can be transferred to or from the disk). (Note that in the case of zone-bit recording, the amount of data passing a read/write head in a single disk rotation is greater for tracks in an outer zone than for an inner zone, and therefore the data transfer rate varies depending on the portion of the disk being used.)

A factor limiting the access time and transfer rate is the speed at which a disk system rotates. To facilitate fast rotation speeds, the read/write heads in these systems do not touch the disk but instead “float” just off the surface. The spacing is so close that even a single particle of dust could become jammed between the head and disk surface, destroying both (a phenomenon known as a head crash). Thus, disk systems are typically housed in cases that are sealed at the factory. With this construction, disk systems are able to rotate at speeds of several hundred times per second, achieving transfer rates that are measured in MB per second.

Since disk systems require physical motion for their operation, these systems suffer when compared to speeds within electronic circuitry. Delay times within an electronic circuit are measured in units of nanoseconds (billionths of a second) or less, whereas seek times, latency times, and access times of disk systems are measured in milliseconds (thousandths of a second). Thus the time required to retrieve information from a disk system can seem like an eternity to an electronic circuit awaiting a result.

Magnetic storage technologies that are now less widely used include **magnetic tape,** in which information is recorded on the magnetic coating of a thin plastic tape wound on reels, and **floppy disk drives,** in which single platters with a magnetic coating are encased in a portable cartridge designed to be readily removed from the drive. Magnetic tape drives have extremely long seek times, just as their cousins, audio cassettes, suffer from long rewind and fast-forward times. Low cost and high data capacities still make magnetic tape suitable for applications where data is primarily read or written linearly, such as archival data backups. The removable nature of floppy disk platters came at the cost of much lower data densities and access speeds than hard disk platters, but their portability was extremely valuable in earlier decades, prior to the arrival of flash drives with larger capacity and higher durability.

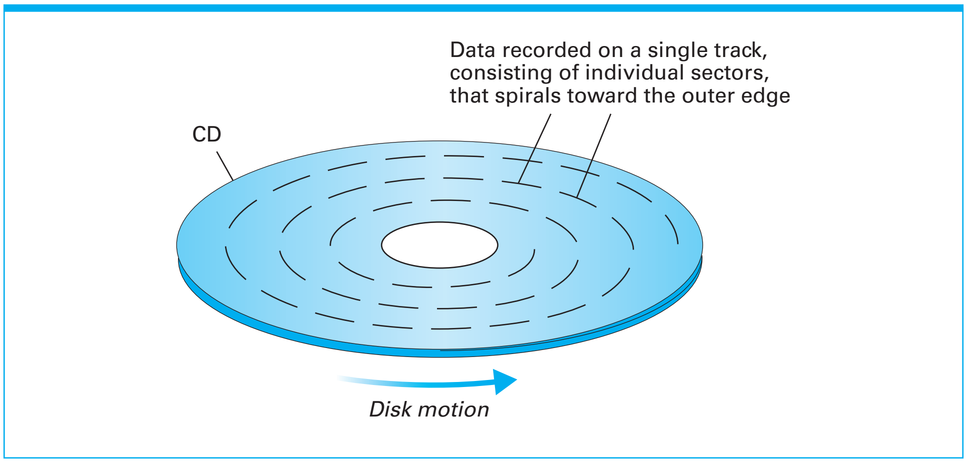
Optical Systems

Another class of mass storage systems applies optical technology. An example is the **compact disk (CD).** These disks are 12 centimeters (approximately 5 inches) in diameter and consist of reflective material covered with a clear protective coating. Information is recorded on them by creating variations in their reflective surfaces. This information can then be retrieved by means of a laser that detects irregularities on the reflective surface of the CD as it spins.

CD technology was originally applied to audio recordings using a recording format known as **CD-DA (compact disk-digital audio),** and the CDs used today for computer data storage use essentially the same format. In particular, information on these CDs is stored on a single track that spirals around the CD like a groove in an old-fashioned phonograph record, however, unlike old-fashioned phonograph records, the track on a CD spirals from the inside out (Figure 1.10). This track is divided into units called sectors, each with its own identifying markings and a capacity of 2KB of data, which equates to 1/75 of a second of music in the case of audio recordings.

Note that the distance around the spiraled track is greater toward the outer edge of the disk than at the inner portion. To maximize the capacity of a CD, information is stored at a uniform linear density over the entire spiraled track, which means that more information is stored in a loop around the outer portion of the spiral than in a loop around the inner portion. In turn, more sectors will be read in a single revolution of the disk when the laser is scanning the outer portion of the spiraled track than when the laser is scanning the inner portion of the track. Thus, to obtain a uniform rate of data transfer, CD-DA players are designed

Figure 1.10 CD storage format


Data recorded on a single track, consisting of individual sectors, that spirals toward the outer edge
CD
Disk motion

to vary the rotation speed depending on the location of the laser. However, most CD systems used for computer data storage spin at a faster, constant speed and thus must accommodate variations in data transfer rates.

As a consequence of such design decisions, CD storage systems perform best when dealing with long, continuous strings of data, as when reproducing music. In contrast, when an application requires access to items of data in a random manner, the approach used in magnetic disk storage (individual, concentric tracks divided into individually accessible sectors) outperforms the spiral approach used in CDs.

Traditional CDs have capacities in the range of 600 to 700MB. However, **DVDs (Digital Versatile Disks),** which are constructed from multiple, semi-transparent layers that serve as distinct surfaces when viewed by a precisely focused laser, provide storage capacities of several GB. Such disks are capable of storing lengthy multimedia presentations, including entire motion pictures. Finally, Blu-ray technology, which uses a laser in the blue-violet spectrum of light (instead of red), is able to focus its laser beam with very fine precision. As a result, **BDs (Blu-ray Disks)** provides over five times the capacity of a DVD. This seemingly vast amount of storage is needed to meet the demands of high definition video.

Flash Drives

A common property of mass storage systems based on magnetic or optic technology is that physical motion, such as spinning disks, moving read/write heads, and aiming laser beams, is required to store and retrieve data. This means that data storage and retrieval is slow compared to the speed of electronic circuitry. **Flash memory** technology has the potential of alleviating this drawback. In a flash memory system, bits are stored by sending electronic signals directly to the storage medium where they cause electrons to be trapped in tiny chambers of silicon dioxide, thus altering the characteristics of small electronic circuits. Since these chambers are able to hold their captive electrons for many years without external power, this technology is excellent for portable, nonvolatile data storage.

Although data stored in flash memory systems can be accessed in small byte-size units as in RAM applications, current technology dictates that stored data be erased in large blocks. Moreover, repeated erasing slowly damages the silicon dioxide chambers, meaning that current flash memory technology is not suitable for general main memory applications where its contents might be altered many times a second. However, in those applications in which alterations can be controlled to a reasonable level, such as in digital cameras and smartphones, flash memory has become the mass storage technology of choice. Indeed, since flash memory is not sensitive to physical shock (in contrast to magnetic and optic systems), it is now replacing other mass storage technologies in portable applications such as laptop computers.

Flash memory devices called **flash drives,** with capacities of hundreds of GBs, are available for general mass storage applications. These units are packaged in ever smaller plastic cases with a removable cap on one end to protect the unit's electrical connector when the drive is offline. The high capacity of these portable units as well as the fact that they are easily connected to and disconnected from a computer make them ideal for portable data storage. However, the vulnerability of their tiny storage chambers dictates that they are not as reliable as optical disks for truly long-term applications.

Larger flash memory devices called **SSDs (solid-state disks)** are explicitly designed to take the place of magnetic hard disks. SSDs compare favorably to hard disks in their resilience to vibrations and physical shock, their quiet operation (due to no moving parts), and their lower access times. SSDs remain more expensive than hard disks of comparable size and thus are still considered a high-end option when buying a computer. SSD sectors suffer from the more limited lifetime of all flash memory technologies, but the use of **wear-leveling** techniques can reduce the impact of this by relocating frequently altered data blocks to fresh locations on the drive.

Another application of flash technology is found in **SD (Secure Digital) memory cards** (or just SD Card). These provide up to two GBs of storage and are packaged in a plastic rigged wafer about the size a postage stamp (SD cards are also available in smaller mini and micro sizes), **SDHC (High Capacity)** memory cards can provide up to 32 GBs and the next generation **SDXC (Extended Capacity) memory cards** may exceed a TB. Given their compact physical size, these cards conveniently slip into slots of small electronic devices. Thus, they are ideal for digital cameras, smartphones, music players, car navigation systems, and a host of other electronic appliances.

Questions & Exercises

1. What is gained by increasing the rotation speed of a disk or CD?
2. When recording data on a multiple-disk storage system, should we fill a complete disk surface before starting on another surface, or should we first fill an entire cylinder before starting on another cylinder?
3. Why should the data in a reservation system that is constantly being updated be stored on a magnetic disk instead of a CD or DVD?

4. What factors allow CD, DVD, and Blu-ray disks all to be read by the same drive?
5. What advantage do flash drives have over the other mass storage systems introduced in this section?
6. What advantages continue to make magnetic hard disk drives competitive?

1.4 Representing Information as Bit Patterns

Having considered techniques for storing bits, we now consider how information can be encoded as bit patterns. Our study focuses on popular methods for encoding text, numerical data, images, and sound. Each of these systems has repercussions that are often visible to a typical computer user. Our goal is to understand enough about these techniques so that we can recognize their consequences for what they are.

Representing Text

Information in the form of text is normally represented by means of a code in which each of the different symbols in the text (such as the letters of the alphabet and punctuation marks) is assigned a unique bit pattern. The text is then represented as a long string of bits in which the successive patterns represent the successive symbols in the original text.

In the 1940s and 1950s, many such codes were designed and used in connection with different pieces of equipment, producing a corresponding proliferation of communication problems. To alleviate this situation, the **American National Standards Institute** (**ANSI,** pronounced "AN–see") adopted the **American Standard Code for Information Interchange** (**ASCII,** pronounced "AS–kee"). This code uses bit patterns of length seven to represent the upper- and lowercase letters of the English alphabet, punctuation symbols, the digits 0 through 9, and certain control information such as line feeds, carriage returns, and tabs. ASCII is extended to an eight-bit-per-symbol format by adding a 0 at the most significant end of each of the seven-bit patterns. This technique not only produces a code in which each pattern fits conveniently into a typical byte-size memory cell but also provides 128 additional bit patterns (those obtained by assigning the extra bit the value 1) that can be used to represent symbols beyond the English alphabet and associated punctuation.

A portion of ASCII in its eight-bit-per-symbol format is shown in Appendix A. By referring to this appendix, we can decode the bit pattern

```
01001000    01100101    01101100    01101100    01101111    00101110
```

as the message "Hello." as demonstrated in Figure 1.11.

The **International Organization for Standardization** (also known as **ISO,** in reference to the Greek word *isos,* meaning equal) has developed a number of extensions to ASCII, each of which was designed to accommodate a major language group. For example, one standard provides the symbols needed to express the text of most Western European languages. Included in its 128 additional patterns are symbols for the British pound and the German vowels ä, ö, and ü.

Figure 1.11 The message "Hello." in ASCII or UTF-8 encoding

01001000	01100101	01101100	01101100	01101111	00101110
H	e	l	l	o	.

The ISO-extended ASCII standards made tremendous headway toward supporting all of the world's multilingual communication; however, two major obstacles surfaced. First, the number of extra bit patterns available in extended ASCII is simply insufficient to accommodate the alphabet of many Asian and some Eastern European languages. Second, because a given document was constrained to using symbols in just the one selected standard, documents containing text of languages from disparate language groups could not be supported. Both proved to be a significant detriment to international use. To address this deficiency, **Unicode** was developed through the cooperation of several of the leading manufacturers of hardware and software and has rapidly gained the support of the computing community. This code uses a unique pattern of up to 21 bits to represent each symbol. When the Unicode character set is combined with the **Unicode Transformation Format 8-bit (UTF-8)** encoding standard, the original ASCII characters can still be represented with 8 bits, while the thousands of additional characters from such languages as Chinese, Japanese, and Hebrew can be represented by 16 bits. Beyond the characters required for all of the world's commonly used languages, UTF-8 uses 24- or 32-bit patterns to represent more obscure Unicode symbols, leaving ample room for future expansion.

A file consisting of a long sequence of symbols encoded using ASCII or Unicode is often called a **text file.** It is important to distinguish between simple text files that are manipulated by utility programs called **text editors** (or often simply editors) and the more elaborate files produced by **word processors** such as Microsoft's Word. Both consist of textual material. However, a text file contains only a character-by-character encoding of the text, whereas a file produced by a word processor contains numerous proprietary codes representing changes in fonts, alignment information, and other parameters.

Representing Numeric Values

Storing information in terms of encoded characters is inefficient when the information being recorded is purely numeric. To see why, consider the problem of storing the value 25. If we insist on storing it as encoded symbols in ASCII using one byte per symbol, we need a total of 16 bits. Moreover, the largest number we could store using 16 bits is 99. However, as we will shortly see, by using **binary notation** we can store any integer in the range from 0 to 65535 in these 16 bits. Thus, binary notation (or variations of it) is used extensively for encoded numeric data for computer storage.

Binary notation is a way of representing numeric values using only the digits 0 and 1 rather than the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 as in the traditional decimal, or base 10, system. We will study the binary system more thoroughly in Section 1.5. For now, all we need is an elementary understanding of the system. For this purpose consider an old-fashioned car odometer whose display wheels

The American National Standards Institute

The American National Standards Institute (ANSI) was founded in 1918 by a small consortium of engineering societies and government agencies as a nonprofit federation to coordinate the development of voluntary standards in the private sector. Today, ANSI membership includes more than 1300 businesses, professional organizations, trade associations, and government agencies. ANSI is headquartered in New York and represents the United States as a member body in the ISO. The website for the American National Standards Institute is at http://www.ansi.org.

Similar organizations in other countries include Standards Australia (Australia), Standards Council of Canada (Canada), China State Bureau of Quality and Technical Supervision (China), Deutsches Institut für Normung (Germany), Japanese Industrial Standards Committee (Japan), Dirección General de Normas (Mexico), State Committee of the Russian Federation for Standardization and Metrology (Russia), Swiss Association for Standardization (Switzerland), and British Standards Institution (United Kingdom).

contain only the digits 0 and 1 rather than the traditional digits 0 through 9. The odometer starts with a reading of all 0s, and as the car is driven for the first few miles, the rightmost wheel rotates from a 0 to a 1. Then, as that 1 rotates back to a 0, it causes a 1 to appear to its left, producing the pattern 10. The 0 on the right then rotates to a 1, producing 11. Now the rightmost wheel rotates from 1 back to 0, causing the 1 to its left to rotate to a 0 as well. This in turn causes another 1 to appear in the third column, producing the pattern 100. In short, as we drive the car we see the following sequence of odometer readings:

```
0000
0001
0010
0011
0100
0101
0110
0111
1000
```

This sequence consists of the binary representations of the integers zero through eight. Although tedious, we could extend this counting technique to discover that the bit pattern consisting of 16 1s represents the value 65535, which confirms our claim that any integer in the range from 0 to 65535 can be encoded using 16 bits.

Due to this efficiency, it is common to store numeric information in a form of binary notation rather than in encoded symbols. We say “a form of binary notation” because the straightforward binary system just described is only the basis for several numeric storage techniques used within machines. Some of these variations of the binary system are discussed later in this chapter. For now, we merely note that a system called **two’s complement** notation (see Section 1.6) is common for storing whole numbers because it provides a convenient method for representing negative numbers as well as positive. For representing numbers with fractional parts such as 4-1/2 or 3/4, another technique, called **floating-point notation** (see Section 1.7), is used.

Representing Images

One means of representing an image is to interpret the image as a collection of dots, each of which is called a **pixel,** short for "picture element." The appearance of each pixel is then encoded and the entire image is represented as a collection of these encoded pixels. Such a collection is called a **bit map.** This approach is popular because many display devices, such as printers and display screens, operate on the pixel concept. In turn, images in bit map form are easily formatted for display.

The method of encoding the pixels in a bit map varies among applications. In the case of a simple black-and-white image, each pixel can be represented by a single bit whose value depends on whether the corresponding pixel is black or white. This is the approach used by most facsimile machines. For more elaborate black-and-white photographs, each pixel can be represented by a collection of bits (usually eight), which allows a variety of shades of grayness to be represented. In the case of color images, each pixel is encoded by more complex system. Two approaches are common. In one, which we will call RGB encoding, each pixel is represented as three color components—a red component, a green component, and a blue component—corresponding to the three primary colors of light. One byte is normally used to represent the intensity of each color component. In turn, three bytes of storage are required to represent a single pixel in the original image.

An alternative to simple RGB encoding is to use a "brightness" component and two color components. In this case the "brightness" component, which is called the pixel's luminance, is essentially the sum of the red, green, and blue components. (Actually, it is considered to be the amount of white light in the pixel, but these details need not concern us here.) The other two components, called the blue chrominance and the red chrominance, are determined by computing the difference between the pixel's luminance and the amount of blue or red light, respectively, in the pixel. Together these three components contain the information required to reproduce the pixel.

The popularity of encoding images using luminance and chrominance components originated in the field of color television broadcast because this approach provided a means of encoding color images that was also compatible with older black-and-white television receivers. Indeed, a gray-scale version of an image can be produced by using only the luminance components of the encoded color image.

ISO—The International Organization for Standardization

The International Organization for Standardization (more commonly called ISO) was established in 1947 as a worldwide federation of standardization bodies, one from each country. Today, it is headquartered in Geneva, Switzerland, and has more than 100 member bodies as well as numerous correspondent members. (A correspondent member is usually a standardization body from a country that does not have a nationally recognized standardization body. Such members cannot participate directly in the development of standards but are kept informed of ISO activities.) ISO maintains a website at http://www.iso.org.

A disadvantage of representing images as bit maps is that an image cannot be rescaled easily to any arbitrary size. Essentially, the only way to enlarge the image is to make the pixels bigger, which leads to a grainy appearance. (This is the technique called "digital zoom" used in digital cameras as opposed to "optical zoom" that is obtained by adjusting the camera lens.)

An alternate way of representing images that avoids this scaling problem is to describe the image as a collection of geometric structures, such as lines and curves, that can be encoded using techniques of analytic geometry. Such a description allows the device that ultimately displays the image to decide how the geometric structures should be displayed rather than insisting that the device reproduce a particular pixel pattern. This is the approach used to produce the scalable fonts that are available via today's word processing systems. For example, TrueType (developed by Microsoft and Apple) is a system for geometrically describing text symbols. Likewise, PostScript (developed by Adobe Systems) provides a means of describing characters as well as more general pictorial data. This geometric means of representing images is also popular in **computer-aided design (CAD)** systems in which drawings of three-dimensional objects are displayed and manipulated on computer display screens.

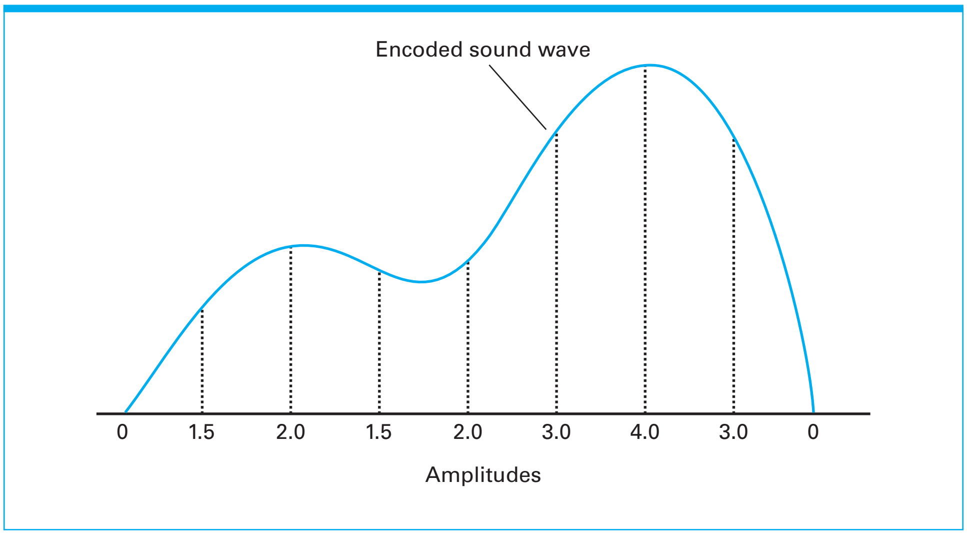
The distinction between representing an image in the form of geometric structures as opposed to bit maps is evident to users of many drawing software systems (such as Microsoft's Paint utility) that allow the user to draw pictures consisting of pre-established shapes such as rectangles, ovals, and elementary curves. The user simply selects the desired geometric shape from a menu and then directs the drawing of that shape via a mouse. During the drawing process, the software maintains a geometric description of the shape being drawn. As directions are given by the mouse, the internal geometric representation is modified, reconverted to bit map form, and displayed. This allows for easy scaling and shaping of the image. Once the drawing process is complete, however, the underlying geometric description is discarded and only the bit map is preserved, meaning that additional alterations require a tedious pixel-by-pixel modification process. On the other hand, some drawing systems preserve the description as geometric shapes that can be modified later. With these systems, the shapes can be easily resized, maintaining a crisp display at any dimension.

Representing Sound

The most generic method of encoding audio information for computer storage and manipulation is to sample the amplitude of the sound wave at regular intervals and record the series of values obtained. For instance, the series 0, 1.5, 2.0, 1.5, 2.0, 3.0, 4.0, 3.0, 0 would represent a sound wave that rises in amplitude, falls briefly, rises to a higher level, and then drops back to 0 (Figure 1.12). This technique, using a sample rate of 8000 samples per second, has been used for years in long-distance voice telephone communication. The voice at one end of the communication is encoded as numeric values representing the amplitude of the voice every eight-thousandth of a second. These numeric values are then transmitted over the communication line to the receiving end, where they are used to reproduce the sound of the voice.

Although 8000 samples per second may seem to be a rapid rate, it is not sufficient for high-fidelity music recordings. To obtain the quality sound reproduction obtained by today's musical CDs, a sample rate of 44,100 samples per second

Figure 1.12 The sound wave represented by the sequence 0, 1.5, 2.0, 1.5, 2.0, 3.0, 4.0, 3.0, 0



is used. The data obtained from each sample are represented in 16 bits (32 bits for stereo recordings). Consequently, each second of music recorded in stereo requires more than a million bits.

An alternative encoding system known as Musical Instrument Digital Interface (MIDI, pronounced "MID–ee") is widely used in the music synthesizers found in electronic keyboards, for video game sound, and for sound effects accompanying websites. By encoding directions for producing music on a synthesizer rather than encoding the sound itself, MIDI avoids the large storage requirements of the sampling technique. More precisely, MIDI encodes what instrument is to play which note for what duration of time, which means that a clarinet playing the note D for two seconds can be encoding in three bytes rather than more than two million bits when sampled at a rate of 44,100 samples per second.

In short, MIDI can be thought of as a way of encoding the sheet music read by a performer rather than the performance itself, and in turn, a MIDI "recording" can sound significantly different when performed on different synthesizers.

Questions & Exercises

1. Here is a message encoded in ASCII using 8 bits per symbol. What does it say? (See Appendix A)

   ```
   01000011 01101111 01101101 01110000 01110101 01110100 01100101
   01110010 00100000 01010011 01100011 01101001 01100101 01101110
   01100011 01100101
   ```

2. In the ASCII code, what is the relationship between the codes for an uppercase letter and the same letter in lowercase? (*See* Appendix A.)

3. Encode these sentences in ASCII:

   a. "Stop!" Cheryl shouted.
   b. Does 2 + 3 = 5?

4. Describe a device from everyday life that can be in either of two states, such as a flag on a flagpole that is either up or down. Assign the symbol 1 to one of the states and 0 to the other, and show how the ASCII representation for the letter b would appear when stored with such bits.

5. Convert each of the following binary representations to its equivalent base 10 form:

   a. 0101
   b. 1001
   c. 1011
   d. 0110
   e. 10000
   f. 10010

6. Convert each of the following base 10 representations to its equivalent binary form:

   a. 6
   b. 13
   c. 11
   d. 18
   e. 27
   f. 4

7. What is the largest numeric value that could be represented with three bytes if each digit were encoded using one ASCII pattern per byte? What if binary notation were used?

8. An alternative to hexadecimal notation for representing bit patterns is **dotted decimal notation** in which each byte in the pattern is represented by its base 10 equivalent. In turn, these byte representations are separated by periods. For example, 12.5 represents the pattern 0000110000000101 (the byte 00001100 is represented by 12, and 00000101 is represented by 5), and the pattern 100010000001000000000111 is represented by 136.16.7. Represent each of the following bit patterns in dotted decimal notation.

   a. 0000111100001111
   b. 001100110000000010000000
   c. 0000101010100000

9. What is an advantage of representing images via geometric structures as opposed to bit maps? What about bit map techniques as opposed to geometric structures?

10. Suppose a stereo recording of one hour of music is encoded using a sample rate of 44,100 samples per second as discussed in the text. How does the size of the encoded version compare to the storage capacity of a CD?

1.5 The Binary System

In Section 1.4 we saw that binary notation is a means of representing numeric values using only the digits 0 and 1 rather than the 10 digits 0 through 9 that are used in the more common base 10 notational system. It is time now to look at binary notation more thoroughly.

Binary Notation

Recall that in the base 10 system, each position in a representation is associated with a quantity. In the representation 375, the 5 is in the position associated with the quantity one, the 7 is in the position associated with ten, and the 3 is in the position associated with the quantity one hundred (Figure 1.13a). Each quantity is 10 times that of the quantity to its right. The value represented by the entire expression is obtained by multiplying the value of each digit by the quantity associated with that digit's position and then adding those products. To illustrate, the pattern 375 represents (3 × hundred) + (7 × ten) + (5 × one), which, in more technical notation, is $(3 \times 10^2) + (7 \times 10^1) + (5 \times 10^0)$.

The position of each digit in binary notation is also associated with a quantity, except that the quantity associated with each position is twice the quantity associated with the position to its right. More precisely, the rightmost digit in a binary representation is associated with the quantity one (2^0), the next position to the left is associated with two (2^1), the next is associated with four (2^2), the next with eight (23), and so on. For example, in the binary representation 1011, the rightmost 1 is in the position associated with the quantity one, the 1 next to it is in the position associated with two, the 0 is in the position associated with four, and the leftmost 1 is in the position associated with eight (Figure 1.13b).

To extract the value represented by a binary representation, we follow the same procedure as in base 10—we multiply the value of each digit by the quantity associated with its position and add the results. For example, the value represented by 100101 is 37, as shown in Figure 1.14. Note that since binary notation uses only the digits 0 and 1, this multiply-and-add process reduces merely to adding the quantities associated with the positions occupied by 1s. Thus the binary pattern 1011 represents the value eleven, because the 1s are found in the positions associated with the quantities one, two, and eight.

In Section 1.4 we learned how to count in binary notation, which allowed us to encode small integers. For finding binary representations of large values, you may prefer the approach described by the algorithm in Figure 1.15. Let us apply this algorithm to the value thirteen (Figure 1.16). We first divide thirteen by two, obtaining a quotient of six and a remainder of one. Since the quotient was not zero, Step 2 tells us to divide the quotient (six) by two, obtaining a new quotient of three and a remainder of zero. The newest quotient is still not zero, so we divide it by two, obtaining a quotient of one and a remainder of one. Once again, we divide the newest quotient (one) by two, this time obtaining a quotient of zero and a remainder of one. Since we have now acquired a quotient of zero, we move on to Step 3, where we learn that the binary representation of the original value (thirteen) is 1101, obtained from the list of remainders.

Figure 1.13 The base 10 and binary systems

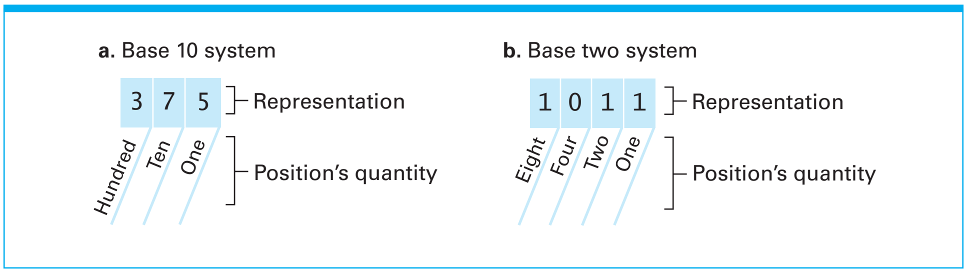


Figure 1.14 Decoding the binary representation 100101

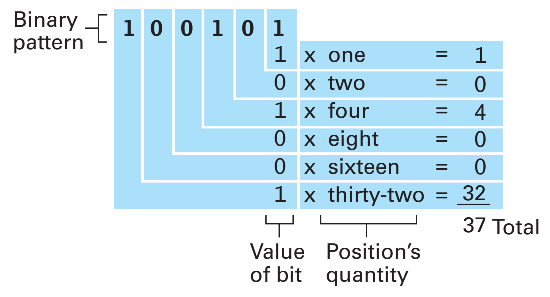
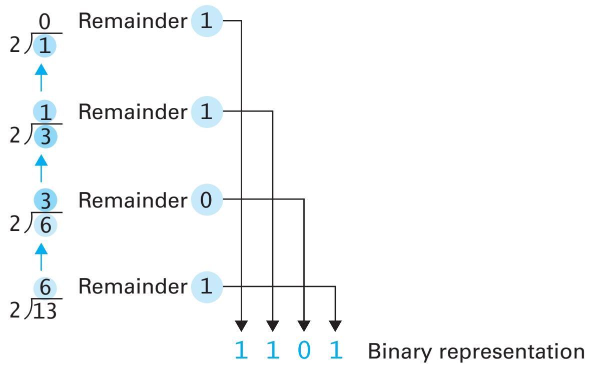


Figure 1.15 An algorithm for finding the binary representation of a positive integer

Step 1.	Divide the value by two and record the remainder.
Step 2.	As long as the quotient obtained is not zero, continue to divide the newest quotient by two and record the remainder.
Step 3.	Now that a quotient of zero has been obtained, the binary representation of the original value consists of the remainders listed from right to left in the order they were recorded.

Figure 1.16 Applying the algorithm in Figure 1.15 to obtain the binary representation of thirteen



Binary Addition

To understand the process of adding two integers that are represented in binary, let us first recall the process of adding values that are represented in traditional base 10 notation. Consider, for example, the following problem:

```
   58
+  27
```

We begin by adding the 8 and the 7 in the rightmost column to obtain the sum 15. We record the 5 at the bottom of that column and carry the 1 to the next column, producing

$$\begin{array}{r} 1 \\ 58 \\ +\ 27 \\ \hline 5 \end{array}$$

We now add the 5 and 2 in the next column along with the 1 that was carried to obtain the sum 8, which we record at the bottom of the column. The result is as follows:

$$\begin{array}{r} 58 \\ +\ 27 \\ \hline 85 \end{array}$$

In short, the procedure is to progress from right to left as we add the digits in each column, write the least significant digit of that sum under the column, and carry the more significant digit of the sum (if there is one) to the next column.

To add two integers represented in binary notation, we follow the same procedure except that all sums are computed using the addition facts shown in Figure 1.17 rather than the traditional base 10 facts that you learned in elementary school. For example, to solve the problem

$$\begin{array}{r} 111010 \\ +\ \ 11011 \\ \hline \end{array}$$

we begin by adding the rightmost 0 and 1; we obtain 1, which we write below the column. Now we add the 1 and 1 from the next column, obtaining 10. We write the 0 from this 10 under the column and carry the 1 to the top of the next column. At this point, our solution looks like this:

$$\begin{array}{r} 1 \\ 111010 \\ +\ \ 11011 \\ \hline 01 \end{array}$$

We add the 1, 0, and 0 in the next column, obtain 1, and write the 1 under this column. The 1 and 1 from the next column total 10; we write the 0 under the column and carry the 1 to the next column. Now our solution looks like this:

$$\begin{array}{r} 1 \\ 111010 \\ +\ \ 11011 \\ \hline 0101 \end{array}$$

Figure 1.17 The binary addition facts

$$\begin{array}{r} 0 \\ +0 \\ \hline 0 \end{array} \quad \begin{array}{r} 1 \\ +0 \\ \hline 1 \end{array} \quad \begin{array}{r} 0 \\ +1 \\ \hline 1 \end{array} \quad \begin{array}{r} 1 \\ +1 \\ \hline 10 \end{array}$$

The 1, 1, and 1 in the next column total 11 (binary notation for the value three); we write the low-order 1 under the column and carry the other 1 to the top of the next column. We add that 1 to the 1 already in that column to obtain 10. Again, we record the low-order 0 and carry the 1 to the next column. We now have

```
 1
  111010
+  11011
  010101
```

The only entry in the next column is the 1 that we carried from the previous column so we record it in the answer. Our final solution is this:

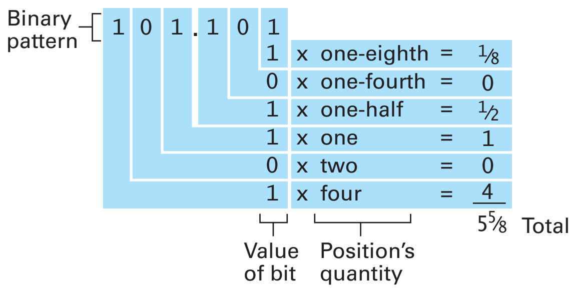
```
  111010
+  11011
 1010101
```

Fractions in Binary

To extend binary notation to accommodate fractional values, we use a **radix point** in the same role as the decimal point in decimal notation. That is, the digits to the left of the point represent the integer part (whole part) of the value and are interpreted as in the binary system discussed previously. The digits to its right represent the fractional part of the value and are interpreted in a manner similar to the other bits, except their positions are assigned fractional quantities. That is, the first position to the right of the radix is assigned the quantity 1/2 (which is 2^{-1}), the next position the quantity 1/4 (which is 2^{-2}), the next 1/8 (which is 2^{-3}), and so on. Note that this is merely a continuation of the rule stated previously: Each position is assigned a quantity twice the size of the one to its right. With these quantities assigned to the bit positions, decoding a binary representation containing a radix point requires the same procedure as used without a radix point. More precisely, we multiply each bit value by the quantity assigned to that bit's position in the representation. To illustrate, the binary representation 101.101 decodes to 5-5/8, as shown in Figure 1.18.

For addition, the techniques applied in the base 10 system are also applicable in binary. That is, to add two binary representations having radix points, we

Figure 1.18 Decoding the binary representation 101.101



Analog versus Digital

Prior to the twenty-first century, many researchers debated the pros and cons of digital versus analog technology. In a digital system, a value is encoded as a series of digits and then stored using several devices, each representing one of the digits. In an analog system, each value is stored in a single device that can represent any value within a continuous range.

Let us compare the two approaches using buckets of water as the storage devices. To simulate a digital system, we could agree to let an empty bucket represent the digit 0 and a full bucket represent the digit 1. Then we could store a numeric value in a row of buckets using floating-point notation (see Section 1.7). In contrast, we could simulate an analog system by partially filling a single bucket to the point at which the water level represented the numeric value being represented. At first glance, the analog system may appear to be more accurate since it would not suffer from the truncation errors inherent in the digital system (again see Section 1.7). However, any movement of the bucket in the analog system could cause errors in detecting the water level, whereas a significant amount of sloshing would have to occur in the digital system before the distinction between a full bucket and an empty bucket would be blurred. Thus the digital system would be less sensitive to error than the analog system. This robustness is a major reason why many applications that were originally based on analog technology (such as telephone communication, audio recordings, and television) are shifting to digital technology.

merely align the radix points and apply the same addition process as before. For example, 10.011 added to 100.11 produces 111.001, as shown here:

```
   10.011
+ 100.110
  -------
  111.001
```

Questions & Exercises

1. Convert each of the following binary representations to its equivalent base 10 form:

   a. 101010 b. 100001 c. 10111 d. 0110 e. 11111

2. Convert each of the following base 10 representations to its equivalent binary form:

   a. 32 b. 64 c. 96 d. 15 e. 27

3. Convert each of the following binary representations to its equivalent base 10 form:

   a. 11.01 b. 101.111 c. 10.1 d. 110.011 e. 0.101

4. Express the following values in binary notation:

   a. $4\frac{1}{2}$ b. $2\frac{3}{4}$ c. $1\frac{1}{8}$ d. $\frac{5}{16}$ e. $5\frac{5}{8}$

5. Perform the following additions in binary notation:

```
a.  11011     b.  1010.001     c.  11111     d.  111.11
   +1100        +    1.101       + 0001        + 00.01
```

1.6 Storing Integers

Mathematicians have long been interested in numeric notational systems, and many of their ideas have turned out to be very compatible with the design of digital circuitry. In this section we consider two of these notational systems, two's complement notation and excess notation, which are used for representing integer values in computing equipment. These systems are based on the binary system but have additional properties that make them more compatible with computer design. With these advantages, however, come disadvantages as well. Our goal is to understand these properties and how they affect computer usage.

Two's Complement Notation

The most popular system for representing integers within today's computers is **two's complement** notation. This system uses a fixed number of bits to represent each of the values in the system. In today's equipment, it is common to use a two's complement system in which each value is represented by a pattern of 32 bits. Such a large system allows a wide range of numbers to be represented but is awkward for demonstration purposes. Thus, to study the properties of two's complement systems, we will concentrate on smaller systems.

Figure 1.19 shows two complete two's complement systems—one based on bit patterns of length three, the other based on bit patterns of length four. Such a system is constructed by starting with a string of 0s of the appropriate length and then counting in binary until the pattern consisting of a single 0 followed by 1s is reached. These patterns represent the values 0, 1, 2, 3, ... The patterns representing negative values are obtained by starting with a string of 1s of the appropriate length and then counting backward in binary until the pattern consisting of a single 1 followed by 0s is reached. These patterns represent the values −1, −2, −3, ... (If counting backward in binary is difficult for you, merely start at the very bottom of the table with the pattern consisting of a single 1 followed by 0s, and count up to the pattern consisting of all 1s.)

Note that in a two's complement system, the leftmost bit of a bit pattern indicates the sign of the value represented. Thus, the leftmost bit is often called the **sign bit.** In a two's complement system, negative values are represented by the patterns whose sign bits are 1; nonnegative values are represented by patterns whose sign bits are 0.

In a two's complement system, there is a convenient relationship between the patterns representing positive and negative values of the same magnitude. They are identical when read from right to left, up to and including the first 1. From there on, the patterns are complements of one another. (The **complement** of a pattern is the pattern obtained by changing all the 0s to 1s and all the 1s to 0s; 0110 and 1001 are complements.) For example, in the 4-bit system in Figure 1.19

Figure 1.19 Two's complement notation systems

a. Using patterns of length three

Bit pattern	Value represented
011	3
010	2
001	1
000	0
111	-1
110	-2
101	-3
100	-4

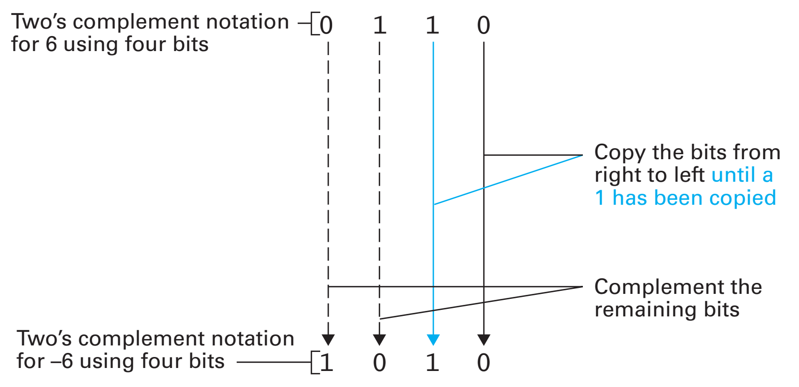
b. Using patterns of length four

Bit pattern	Value represented
0111	7
0110	6
0101	5
0100	4
0011	3
0010	2
0001	1
0000	0
1111	-1
1110	-2
1101	-3
1100	-4
1011	-5
1010	-6
1001	-7
1000	-8

the patterns representing 2 and −2 both end with 10, but the pattern representing 2 begins with 00, whereas the pattern representing −2 begins with 11. This observation leads to an algorithm for converting back and forth between bit patterns representing positive and negative values of the same magnitude. We merely copy the original pattern from right to left until a 1 has been copied, then we complement the remaining bits as they are transferred to the final bit pattern (Figure 1.20).

Understanding these basic properties of two's complement systems also leads to an algorithm for decoding two's complement representations. If the pattern

Figure 1.20 Encoding the value—6 in two's complement notation using 4 bits


Two's complement notation for 6 using four bits
0 1 1 0
Copy the bits from right to left until a 1 has been copied
Complement the remaining bits
Two's complement notation for –6 using four bits
1 0 1 0

to be decoded has a sign bit of 0, we need merely read the value as though the pattern were a binary representation. For example, 0110 represents the value 6, because 110 is binary for 6. If the pattern to be decoded has a sign bit of 1, we know the value represented is negative, and all that remains is to find the magnitude of the value. We do this by applying the “copy and complement” procedure in Figure 1.20 and then decoding the pattern obtained as though it were a straightforward binary representation. For example, to decode the pattern 1010, we first recognize that since the sign bit is 1, the value represented is negative. Hence, we apply the “copy and complement” procedure to obtain the pattern 0110, recognize that this is the binary representation for 6, and conclude that the original pattern represents −6.

Addition in Two’s Complement Notation To add values represented in two’s complement notation, we apply the same algorithm that we used for binary addition, except that all bit patterns, including the answer, are the same length. This means that when adding in a two’s complement system, any extra bit generated on the left of the answer by a final carry must be truncated. Thus “adding” 0101 and 0010 produces 0111, and “adding” 0111 and 1011 results in 0010 (0111 + 1011 = 10010, which is truncated to 0010).

With this understanding, consider the three addition problems in Figure 1.21. In each case, we have translated the problem into two’s complement notation (using bit patterns of length four), performed the addition process previously described, and decoded the result back into our usual base 10 notation.

Observe that the third problem in Figure 1.21 involves the addition of a positive number to a negative number, which demonstrates a major benefit of two’s complement notation: Addition of any combination of signed numbers can be accomplished using the same algorithm and thus the same circuitry. This is in stark contrast to how humans traditionally perform arithmetic computations. Whereas elementary school children are first taught to add and later taught to subtract, a machine using two’s complement notation needs to know only how to add.

Figure 1.21 Addition problems converted to two’s complement notation

Problem in base 10	Problem in two's complement	Answer in base 10
$\begin{array}{r} 3 \\ +\ 2 \\ \hline \end{array}$ →	$\begin{array}{r} 0011 \\ +\ 0010 \\ \hline 0101 \end{array}$ →	5
$\begin{array}{r} -3 \\ +\ -2 \\ \hline \end{array}$ →	$\begin{array}{r} 1101 \\ +\ 1110 \\ \hline 1011 \end{array}$ →	-5
$\begin{array}{r} 7 \\ +\ -5 \\ \hline \end{array}$ →	$\begin{array}{r} 0111 \\ +\ 1011 \\ \hline 0010 \end{array}$ →	2

For example, the subtraction problem 7 − 5 is the same as the addition problem 7 + (−5). Consequently, if a machine were asked to subtract 5 (stored as 0101) from 7 (stored as 0111), it would first change the 5 to −5 (represented as 1011) and then perform the addition process of 0111 + 1011 to obtain 0010, which represents 2, as follows:

$$\begin{array}{rcrcrcr} 7 & & 0111 & & 0111 & & \\ \underline{-5} & \rightarrow & \underline{-\ 0101} & \rightarrow & \underline{+\ 1011} & & \\ & & & & 0010 & \rightarrow & 2 \end{array}$$

We see, then, that when two's complement notation is used to represent numeric values, a circuit for addition combined with a circuit for negating a value is sufficient for solving both addition and subtraction problems. (Such circuits are shown and explained in Appendix B.)

The Problem of Overflow One problem we have avoided in the preceding examples is that in any two's complement system there is a limit to the size of the values that can be represented. When using two's complement with patterns of 4 bits, the largest positive integer that can be represented is 7, and the most negative integer is −8. In particular, the value 9 cannot be represented, which means that we cannot hope to obtain the correct answer to the problem 5 + 4. In fact, the result would appear as −7. This phenomenon is called **overflow.** That is, overflow is the problem that occurs when a computation produces a value that falls outside the range of values that can be represented. When using two's complement notation, this might occur when adding two positive values or when adding two negative values. In either case, the condition can be detected by checking the sign bit of the answer. An overflow is indicated if the addition of two positive values results in the pattern for a negative value or if the sum of two negative values appears to be positive.

Of course, because most computers use two's complement systems with longer bit patterns than we have used in our examples, larger values can be manipulated without causing an overflow. Today, it is common to use patterns of 32 bits for storing values in two's complement notation, allowing for positive values as large as 2,147,483,647 to accumulate before overflow occurs. If still larger values are needed, longer bit patterns can be used or perhaps the units of measure can be changed. For instance, finding a solution in terms of miles instead of inches results in smaller numbers being used and might still provide the accuracy required.

The point is that computers can make mistakes. So, the person using the machine must be aware of the dangers involved. One problem is that computer programmers and users become complacent and ignore the fact that small values can accumulate to produce large numbers. For example, in the past it was common to use patterns of 16 bits for representing values in two's complement notation, which meant that overflow would occur when values of $2^{15} = 32{,}768$ or larger were reached. On September 19, 1989, a hospital computer system malfunctioned after years of reliable service. Close inspection revealed that this date was 32,768 days after January 1, 1900, and the machine was programmed to compute dates based on that starting date. Thus, because of overflow, September 19, 1989, produced a negative value—a phenomenon the computer's program was not designed to handle.

Excess Notation

Another method of representing integer values is **excess notation.** As is the case with two's complement notation, each of the values in an excess notation system is represented by a bit pattern of the same length. To establish an excess system, we first select the pattern length to be used, then write down all the different bit patterns of that length in the order they would appear if we were counting in binary. Next, we observe that the first pattern with a 1 as its most significant bit appears approximately halfway through the list. We pick this pattern to represent zero; the patterns following this are used to represent 1, 2, 3, . . .; and the patterns preceding it are used for −1, −2, −3, ... The resulting code, when using patterns of length four, is shown in Figure 1.22. There we see that the value 5 is represented by the pattern 1101 and −5 is represented by 0011. (Note that one difference between an excess system and a two's complement system is that the sign bits are reversed.)

The system represented in Figure 1.22 is known as excess eight notation. To understand why, first interpret each of the patterns in the code using the traditional binary system and then compare these results to the values represented in the excess notation. In each case, you will find that the binary interpretation exceeds the excess notation interpretation by the value 8. For example, the pattern 1100 in binary notation represents the value 12, but in our excess system it represents 4; 0000 in binary notation represents 0, but in the excess system it represents negative 8. In a similar manner, an excess system based on patterns of length five would be called excess 16 notation, because the pattern 10000, for instance, would be used to represent zero rather than representing its usual value of 16. Likewise, you may want to confirm that the three-bit excess system would be known as excess four notation (Figure 1.23).

Figure 1.22 An excess eight conversion table

Bit pattern	Value represented
1111	7
1110	6
1101	5
1100	4
1011	3
1010	2
1001	1
1000	0
0111	-1
0110	-2
0101	-3
0100	-4
0011	-5
0010	-6
0001	-7
0000	-8

Figure 1.23 An excess notation system using bit patterns of length three

Bit pattern	Value represented
111	3
110	2
101	1
100	0
011	-1
010	-2
001	-3
000	-4

Questions & Exercises

1. Convert each of the following two's complement representations to its equivalent base 10 form:

   a. 00011 b. 01111 c. 11100
   d. 11010 e. 00000 f. 10000

2. Convert each of the following base 10 representations to its equivalent two's complement form using patterns of 8 bits:

   a. 6 b. −6 c. −17
   d. 13 e. −1 f. 0

3. Suppose the following bit patterns represent values stored in two's complement notation. Find the two's complement representation of the negative of each value:

   a. 00000001 b. 01010101 c. 11111100
   d. 11111110 e. 00000000 f. 01111111

4. Suppose a machine stores numbers in two's complement notation. What are the largest and smallest numbers that can be stored if the machine uses bit patterns of the following lengths?

   a. four b. six c. eight

5. In the following problems, each bit pattern represents a value stored in two's complement notation. Find the answer to each problem in two's complement notation by performing the addition process described in the text. Then check your work by translating the problem and your answer into base 10 notation.

   a. 0101 + 0010 b. 0011 + 0001 c. 0101 + 1010
   d. 1110 + 0011 e. 1010 + 1110

6. Solve each of the following problems in two's complement notation, but this time watch for overflow and indicate which answers are incorrect because of this phenomenon.

a. 0100 + 0011	b. 0101 + 0110	c. 1010 + 1010
d. 1010 + 0111	e. 0111 + 0001	

7. Translate each of the following problems from base 10 notation into two's complement notation using bit patterns of length four, then convert each problem to an equivalent addition problem (as a machine might do), and perform the addition. Check your answers by converting them back to base 10 notation.

a. $6 - (-1)$	b. $3 - (-2)$	c. $4 - 6$
d. $2 - (-4)$	e. $1 - 5$	

8. Can overflow ever occur when values are added in two's complement notation with one value positive and the other negative? Explain your answer.

9. Convert each of the following excess eight representations to its equivalent base 10 form without referring to the table in the text:

a. 1110	b. 0111	c. 1000
d. 0010	e. 0000	f. 1001

10. Convert each of the following base 10 representations to its equivalent excess eight form without referring to the table in the text:

a. 5	b. −5	c. 3
d. 0	e. 7	f. −8

11. Can the value 9 be represented in excess eight notation? What about representing 6 in excess four notation? Explain your answer.

1.7 Storing Fractions

In contrast to the storage of integers, the storage of a value with a fractional part requires that we store not only the pattern of 0s and 1s representing its binary representation but also the position of the radix point. A popular way of doing this is based on scientific notation and is called **floating-point** notation.

Floating-Point Notation

Let us explain floating-point notation with an example using only one byte of storage. Although machines normally use much longer patterns, this 8-bit format is representative of actual systems and serves to demonstrate the important concepts without the clutter of long bit patterns.

We first designate the high-order bit of the byte as the sign bit. Once again, a 0 in the sign bit will mean that the value stored is nonnegative, and a 1 will mean that the value is negative. Next, we divide the remaining 7 bits of the byte into two groups, or fields: the **exponent field** and the **mantissa field.** Let us designate the 3 bits following the sign bit as the exponent field and the remaining 4 bits as the mantissa field. Figure 1.24 illustrates how the byte is divided.

We can explain the meaning of the fields by considering the following example. Suppose a byte consists of the bit pattern 01101011. Analyzing this pattern with the preceding format, we see that the sign bit is 0, the exponent is 110, and the mantissa is 1011. To decode the byte, we first extract the mantissa and place a radix point on its left side, obtaining

```
.1011
```

Next, we extract the contents of the exponent field (110) and interpret it as an integer stored using the 3-bit excess method (see again Figure 1.24). Thus the pattern in the exponent field in our example represents a positive 2. This tells us to move the radix in our solution to the right by 2 bits. (A negative exponent would mean to move the radix to the left.) Consequently, we obtain

```
10.11
```

which is the binary representation for $2\frac{3}{4}$. (Recall the representation of binary fractions from Figure 1.18.) Next, we note that the sign bit in our example is 0; the value represented is thus nonnegative. We conclude that the byte 01101011 represents $2\frac{3}{4}$. Had the pattern been 11101011 (which is the same as before except for the sign bit), the value represented would have been $2\frac{3}{4}$.

As another example, consider the byte 00111100. We extract the mantissa to obtain

```
.1100
```

and move the radix 1 bit to the left, since the exponent field (011) represents the value −1. We therefore have

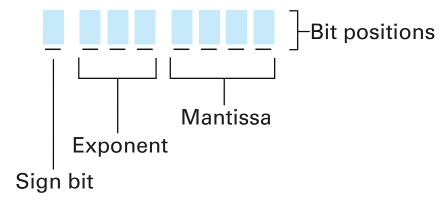
```
.01100
```

which represents 3/8. Since the sign bit in the original pattern is 0, the value stored is nonnegative. We conclude that the pattern 00111100 represents $\frac{3}{8}$.

To store a value using floating-point notation, we reverse the preceding process. For example, to encode $1\frac{1}{8}$, first we express it in binary notation and obtain 1.001. Next, we copy the bit pattern into the mantissa field from left to right, starting with the leftmost 1 in the binary representation. At this point, the byte looks like this:

_ _ _ _ $\underline{1}$ $\underline{0}$ $\underline{0}$ $\underline{1}$

Figure 1.24 Floating-point notation components



We must now fill in the exponent field. To this end, we imagine the contents of the mantissa field with a radix point at its left and determine the number of bits and the direction the radix must be moved to obtain the original binary number. In our example, we see that the radix in .1001 must be moved 1 bit to the right to obtain 1.001. The exponent should therefore be a positive one, so we place 101 (which is positive one in excess four notation as shown in Figure 1.23) in the exponent field. Finally, we fill the sign bit with 0 because the value being stored is nonnegative. The finished byte looks like this:

$\underline{0}\ \underline{1}\ \underline{0}\ \underline{1}\ \underline{1}\ \underline{0}\ \underline{0}\ \underline{1}$

There is a subtle point you may have missed when filling in the mantissa field. The rule is to copy the bit pattern appearing in the binary representation from left to right, starting with the leftmost 1. To clarify, consider the process of storing the value $^{3}/_{8}$, which is .011 in binary notation. In this case the mantissa will be

$\underline{\ }\ \underline{\ }\ \underline{\ }\ \underline{\ }\ \underline{1}\ \underline{1}\ \underline{0}\ \underline{0}$

It will not be

$\underline{\ }\ \underline{\ }\ \underline{\ }\ \underline{\ }\ \underline{0}\ \underline{1}\ \underline{1}\ \underline{0}$

This is because we fill in the mantissa field starting with the leftmost 1 that appears in the binary representation. Representations that conform to this rule are said to be in **normalized form.**

Using normalized form eliminates the possibility of multiple representations for the same value. For example, both 00111100 and 01000110 would decode to the value $^{3}/_{8}$, but only the first pattern is in normalized form. Complying with normalized form also means that the representation for all nonzero values will have a mantissa that starts with 1. The value zero, however, is a special case; its floating-point representation is a bit pattern of all 0s.

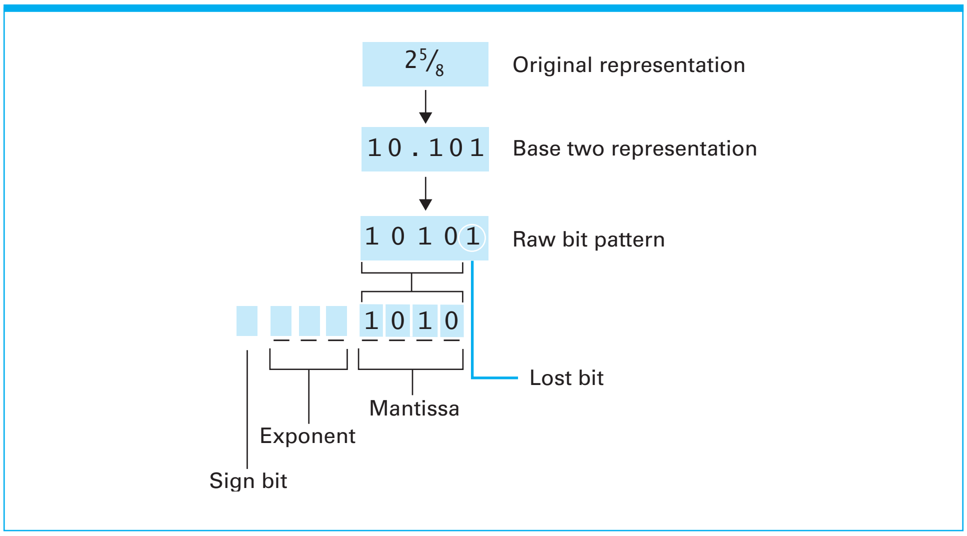
Truncation Errors

Let us consider the annoying problem that occurs if we try to store the value $2^{5}/_{8}$ with our one-byte floating-point system. We first write $2^{5}/_{8}$ in binary, which gives us 10.101. But when we copy this into the mantissa field, we run out of room, and the rightmost 1 (which represents the last $^{1}/_{8}$) is lost (Figure 1.25). If we ignore this problem for now and continue by filling in the exponent field and the sign bit, we end up with the bit pattern 01101010, which represents $2^{1}/_{2}$ instead of $2^{5}/_{8}$. What has occurred is called a **truncation error,** or **round-off error**—meaning that part of the value being stored is lost because the mantissa field is not large enough.

The significance of such errors can be reduced by using a longer mantissa field. In fact, most computers manufactured today use at least 32 bits for storing values in floating-point notation instead of the 8 bits we have used here. This also allows for a longer exponent field at the same time. Even with these longer formats, however, there are still times when more accuracy is required.

Another source of truncation errors is a phenomenon that you are already accustomed to in base 10 notation: the problem of nonterminating expansions, such as those found when trying to express $^{1}/_{3}$ in decimal form. Some values cannot be accurately expressed regardless of how many digits we use. The difference between our traditional base 10 notation and binary notation is that more values have nonterminating representations in binary than in decimal notation. For example, the value 1/10 is nonterminating when expressed in binary. Imagine the problems this might cause the unwary person using floating-point notation to

Figure 1.25 Encoding the value $2\frac{5}{8}$



store and manipulate dollars and cents. In particular, if the dollar is used as the unit of measure, the value of a dime could not be stored accurately. A solution in this case is to manipulate the data in units of pennies so that all values are integers that can be accurately stored using a method such as two's complement.

Truncation errors and their related problems are an everyday concern for people working in the area of numerical analysis. This branch of mathematics deals with the problems involved when doing actual computations that are often massive and require significant accuracy.

The following is an example that would warm the heart of any numerical analyst. Suppose we are asked to add the following three values using our one-byte floating-point notation defined previously:

$2\frac{1}{2} + \frac{1}{8} + \frac{1}{8}$

Single Precision Floating Point

The floating-point notation introduced in this chapter (Section 1.7) is far too simplistic to be used in an actual computer. After all, with just 8 bits, only 256 numbers out of the set of all real numbers can be expressed. Our discussion has used 8 bits to keep the examples simple, yet still cover the important underlying concepts.

Many of today's computers support a 32-bit form of this notation called **Single Precision Floating Point.** This format uses 1 bit for the sign, 8 bits for the exponent (in an excess notation), and 23 bits for the mantissa. Thus, single precision floating point is capable of expressing very large numbers (order of 10^{38}) down to very small numbers (order of 10^{-37}) with the precision of 7 decimal digits. That is to say, the first 7 digits of a given decimal number can be stored with very good accuracy (a small amount of error may still be present). Any digits passed the first 7 will certainly be lost by truncation error (although the magnitude of the number is retained).

Another form, called **Double Precision Floating Point,** uses 64 bits and provides a precision of 15 decimal digits.

If we add the values in the order listed, we first add $2\frac{1}{2}$ to $\frac{1}{8}$ and obtain $2\frac{5}{8}$, which in binary is 10.101. Unfortunately, because this value cannot be stored accurately (as seen previously), the result of our first step ends up being stored as $2\frac{1}{2}$ (which is the same as one of the values we were adding). The next step is to add this result to the last 1/8. Here again a truncation error occurs, and our final result turns out to be the incorrect answer $2\frac{1}{2}$.

Now let us add the values in the opposite order. We first add $\frac{1}{8}$ to $\frac{1}{8}$ to obtain $\frac{1}{4}$. In binary this is .01; so the result of our first step is stored in a byte as 00111000, which is accurate. We now add this $\frac{1}{4}$ to the next value in the list, $2\frac{1}{2}$, and obtain $2\frac{3}{4}$, which we can accurately store in a byte as 01101011. The result this time is the correct answer.

To summarize, in adding numeric values represented in floating-point notation, the order in which they are added can be important. The problem is that if a very large number is added to a very small number, the small number may be truncated. Thus, the general rule for adding multiple values is to add the smaller values together first, in hopes that they will accumulate to a value that is significant when added to the larger values. This was the phenomenon experienced in the preceding example.

Designers of today's commercial software packages do a good job of shielding the uneducated user from problems such as this. In a typical spreadsheet system, correct answers will be obtained unless the values being added differ in size by a factor of 10^{16} or more. Thus, if you found it necessary to add one to the value

```
10,000,000,000,000,000
```

you might get the answer

```
10,000,000,000,000,000
```

rather than

```
10,000,000,000,000,001
```

Such problems are significant in applications (such as navigational systems) in which minor errors can be compounded in additional computations and ultimately produce significant consequences, but for the typical PC user the degree of accuracy offered by most commercial software is sufficient.

Questions & Exercises

1. Decode the following bit patterns using the floating-point format discussed in the text:

   a. 01001010 **b.** 01101101 **c.** 00111001 **d.** 11011100 **e.** 10101011

2. Encode the following values into the floating-point format discussed in the text. Indicate the occurrence of truncation errors.

   a. $2\frac{3}{4}$ **b.** $5\frac{1}{4}$ **c.** $\frac{3}{4}$ **d.** $-3\frac{1}{2}$ **e.** $-4\frac{3}{8}$

3. In terms of the floating-point format discussed in the text, which of the patterns 01001001 and 00111101 represents the larger value? Describe a simple procedure for determining which of two patterns represents the larger value.
4. When using the floating-point format discussed in the text, what is the largest value that can be represented? What is the smallest positive value that can be represented?

1.8 Data and Programming

While humans have devised the data representations and basic operations that comprise modern computers, few people are very good at working with computers directly at this level. People prefer to reason about computational problems at a higher level of abstraction, and they rely on the computer to handle the lowest levels of detail. A *programming language* is a computer system created to allow humans to precisely express algorithms to the computer using a higher level of abstraction.

In the twentieth century, programming computers was considered to be the province of a few highly trained experts; to be sure, there remain many problems in computing that require the attention of experienced computer scientists and software engineers. However, in the twenty-first century, as computers and computing have become increasingly intertwined in every aspect of our modern lives, it has grown steadily more difficult to identify career fields that do not require at least some degree of programming skill. Indeed, some have identified programming or *coding* to be the next foundational pillar of modern literacy, alongside reading, writing, and arithmetic.

In this section, and in programming supplement sections in subsequent chapters, we look at how a programming language reflects the main ideas of the chapter and allows humans to more easily solve problems involving computation.

Getting Started with Python

Python is a programming language that was created by Guido van Rossum in the late 1980s. Today it is one of the top ten most-used languages and remains popular in developing web applications, in scientific computation, and as an introductory language for students. Organizations that use Python range from Google to NASA, DropBox to Industrial Light & Magic, and across the spectrum of casual, scientific, and artistic computer users. Python emphasizes readability and includes elements of the imperative, object-oriented, and functional programming paradigms, which will be explored in Chapter 6.

The software for editing and running programs written in Python is freely available from `www.python.org`, as are many other resources for getting started. The Python language has evolved and continues to evolve over time. All of the examples in this book will use a version of the language called Python 3. Earlier versions of Python are capable of running very similar programs, but there have been many minor changes, such as punctuation, since Python 2.

Python is an *interpreted language,* which for beginners means that Python instructions can be typed into an interactive prompt or can be stored in a plain text file (called a "script") and run later. In the examples below, either mode can be used, but exercise and chapter review problems will generally ask for a Python script.

Hello Python

By longstanding tradition, the first program described in many programming language introductions is "Hello, World." This simple program outputs a nominal greeting, demonstrating how a particular language produces a result, and also how a language represents text. In Python[1], we write this program as

```
print('Hello, World!')
```

Type this statement into Python's interactive interpreter, or save it as a Python script and execute it. In either case, the result should be:

```
Hello, World!
```

Python parrots the text between the quotation marks back to the user.

There are several aspects to note even in this simple Python script. First, `print` is a built-in function, a predefined operation that Python scripts can use to produce *output,* a result of the program that will be made visible to the user. The print is followed by opening and closing parentheses; what comes between those parentheses is the value to be printed.

Second, Python can denote strings of text using single quotation marks. The quotation marks in front of the capital `H` and after the exclamation point denote the beginning and end of a string of characters that will be treated as a value in Python.

Programming languages carry out their instructions very precisely. If a user makes subtle changes to the message between the starting and finishing quotation marks within the print statement, the resultant printed text will change accordingly. Take a moment to try different capitalizations, punctuation, and even different words within the print statement to see that this is so.

Variables

Python allows the user to name values for later use, an important abstraction when constructing compact, understandable scripts. These named storage locations are termed *variables,* analogous to the mathematical variables often seen in algebra courses. Consider the slightly enhanced version of Hello World below:

```
message = 'Hello, World!'
print(message)
```

In this script, the first line is an *assignment statement.* The use of the = can be misleading to beginners, who are accustomed to the algebraic usage of the equal sign. This assignment statement should be read, "variable `message` is assigned

[1]This Python code is for version 3 of the language, which will be referred to only as "Python" for the remainder of the book. Earlier versions of Python do not always require the opening and closing parentheses.

the string value `'Hello, World!'`". In general, an assignment statement will have a variable name on the left side of the equal sign and a value to the right.

Python is a *dynamically typed* language, which means that our script need not establish ahead of time that there will be a variable called `message`, or what type of value should be stored in `message`. In the script, it is sufficient to state that our text string will be assigned to `message`, and then to refer to that variable `message` in the subsequent `print` statement.

The naming of variables is largely up to the user in Python. Python's simple rules are that variable names must begin with an alphabet letter and may consist of an arbitrary number of letters, digits, and the underscore character, _. While a variable named `m` may be sufficient for a two-line example script, experienced programmers strive to give meaningful, descriptive variable names in their scripts.

Python variable names are *case-sensitive,* meaning that capitalization matters. A variable named `size` is treated as distinct from variables named `Size` or `SIZE`. A small number of *keywords,* names that are reserved for special meaning in Python, cannot be used as variable names. You can view this list by accessing the built-in Python help system.

```
help('keywords')
```

Variables can be used to store all of the types of values that Python is able to represent.

```
my_integer = 5
my_floating_point = 26.2
my_Boolean = True
my_string = 'characters'
```

Observe that the types of values we see here correspond directly to the representations covered earlier in this chapter: Boolean trues and falses (Section 1.1), text (Section 1.4), integers (Section 1.6), and floating point numbers (Section 1.7). With additional Python code (beyond the scope of our simple introduction in this text) we could store image and sound data (Section 1.4) with Python variables, as well.

Python expresses hexadecimal values using a `0x` prefix, as in

```
my_integer = 0xFF
print(my_integer)
```

Specifying a value in hexadecimal does not alter the representation of that value in the computer's memory, which stores integer values as a collection of ones and zeros regardless of the numerical base used in the programmer's reasoning. Hexadecimal notation remains a shortcut for humans, used in situations where that representation may aid in understanding the script. The `print` statement above thus prints `255`, the base 10 interpretation of hexadecimal `0xFF`, because that is the default behavior for `print`. More complex adjustments to the `print` statement can be used to output values in other representations, but we confine our discussion here to the more familiar base 10.

Unicode characters, including those beyond the ubiquitous ASCII subset, can be included directly in strings when the text editor supports them,

```
print('₹1000')          # Prints ₹1000, one thousand Indian Rupees
```

or can be specified using four hexadecimal digits following a '\u' prefix.

```
print('\u00A31000')       # Prints £1000, one thousand British
                          # Pounds Sterling
```

The portion of the string '\u00A3' encodes the Unicode representation of the British pound symbol. The '1000' follows immediately so that there will be no space between the currency symbol and the amount in the final output: £1000.

These example statements introduce another language feature, in addition to Unicode text strings. The # symbol denotes the beginning of a *comment,* a human-readable notation to the Python code that will be ignored by the computer when executed. Experienced programmers use comments in their code to explain difficult segments of the algorithm, include history or authorship information, or just to note where a human should pay attention when reading the code. All of the characters to the right of the # until the end of the line are ignored by Python.

Operators and Expressions

Python's built-in operators allow values to be manipulated and combined in a variety of familiar ways.

```
print(3 + 4)       # Prints "7", which is 3 plus 4.
print(5 - 6)       # Prints "-1", which is 5 minus 6
print(7 * 8)       # Prints "56", which is 7 times 8
print(45 / 4)      # Prints "11.25", which is 45 divided by 4
print(2 ** 10)     # Prints "1024", which is 2 to the 10th power
```

When an operation such as forty-five divided by four produces a non-integer result, such as 11.25, Python implicitly switches to a floating-point representation. When purely integer answers are desired, a different set of operators can be used.

```
print(45 // 4)     # Prints "11", which is 45 integer divided by 4
print(45 % 4)      # Prints "1", because 4 * 11 + 1 = 45
```

The double slash signifies the *integer floor division* operator, while the percentage symbol signifies the *modulus,* or remainder operator. Taken together, we can read these calculations as, "four goes into forty-five eleven times, with a remainder of one." In the earlier example, we used ** to signify exponentiation, which can be somewhat surprising given that the caret symbol, ^, is often used for this purpose in typewritten text and even some other programming languages. In Python, the caret operator belongs to the group of *bitwise Boolean operations*, which will be discussed in the next chapter.

String values also can be combined and manipulated in some intuitive ways.

```
s = 'hello' + 'world'
t = s * 4

print(t)    # Prints "helloworldhelloworldhelloworldhelloworld"
```

The plus operator *concatenates* string values, while the multiplication operator *replicates* string values.

The multiple meanings of some of the built-in operators can lead to confusion. This script will produce an error:

```
print('USD$' + 1000)     # TypeError: Can't convert 'int' to str implicitly
```

The error indicates that the string concatenation operator doesn't know what to do when the second operand is not also a string. Fortunately, Python provides functions that allow values to be converted from one type of representation to another. The `int()` function will convert a floating-point value back to an integer representation, discarding the fractional part. It will also convert a string of text digits into an integer representation, provided that the string correctly spells out a valid number. Likewise, the `str()` function can be used to convert numeric representations into UTF-8 encoded text strings. Thus, the following modification to the `print` statement above corrects the error.

```
print('USD$' + str(1000))     # Prints "USD$1000"
```

Currency Conversion

The complete Python script example below demonstrates many of the concepts introduced in this section. Given a set number of U.S. dollars, the script produces monetary conversions to four other currencies.

```
# A converter for international currency exchange.
USD_to_GBP = 0.66   # Today's rate, US dollars to British Pounds
USD_to_EUR = 0.77   # Today's rate, US dollars to Euros
USD_to_JPY = 99.18  # Today's rate, US dollars to Japanese Yen
USD_to_INR = 59.52  # Today's rate, US dollars to Indian Rupees

GBP_sign    = '\u00A3' # Unicode values for non-ASCII currency symbols.
EUR_sign    = '\u20AC'
JPY_sign    = '\u00A5'
INR_sign    = '\u20B9'

dollars     = 1000 # The number of dollars to convert

pounds      = dollars * USD_to_GBP   # Conversion calculations
euros       = dollars * USD_to_EUR
yen         = dollars * USD_to_JPY
rupees      = dollars * USD_to_INR

print('Today, $' + str(dollars))    # Printing the results
print('converts to ' + GBP_sign + str(pounds))
print('converts to ' + EUR_sign + str(euros))
print('converts to ' + JPY_sign + str(yen))
print('converts to ' + INR_sign + str(rupees))
```

When executed, this script outputs the following:

```
Today, $1000
converts to £660.0
converts to €770.0
converts to ¥99180.0
converts to ₹59520.0
```

Debugging

Programming languages are not very forgiving for beginners, and a great deal of time learning to write software can be spent trying to find **bugs,** or errors in the code. There are three major classes of bug that we create in software: **syntax errors** (mistakes in the symbols that have been typed), **semantic errors** (mistakes in the meaning of the program), and **runtime errors** (mistakes that occur when the program is executed.)

Syntax errors are the most common for novices and include simple errors such as forgetting one of the quote marks at the beginning or ending of a text string, failing to close open parentheses, or misspelling the function name `print`. The Python interpreter will generally try to point these errors out when it encounters them, displaying an offending line number and a description of the problem. With some practice, a beginner can quickly learn to recognize and interpret common error cases. As examples:

```
print(5 + )
SyntaxError: invalid syntax
```

This expression is missing a value between the addition operator and the closing parenthesis.

```
print(5.e)
SyntaxError: invalid token
```

Python expects digits to follow the decimal point, not a letter.

```
pront(5)
NameError: name 'pront' is not defined
```

Like calling someone by the wrong name, misspelling the name of a known function or variable can result in confusion and embarrassment.

Semantic errors are flaws in the algorithm, or flaws in the way the algorithm is expressed in a language. Examples might include using the wrong variable name in a calculation or getting the order of arithmetic operations wrong in a complex expression. Python follows the standard rules for operator precedence, so in an expression like `total_pay = 40 + extra_hours * pay_rate`, the multiplication will be performed before the addition, incorrectly calculating the total pay. (Unless your pay rate happens to be $1/hour.) Use parenthesis to properly specify the order of operations in complex expressions, thereby avoiding both semantic errors and code that may be harder to understand (e.g., `total_pay = (40 + extra_hours) * pay_rate`).

Finally, runtime errors at this level might include unintentionally dividing by zero or using a variable before you have defined it. Python reads statements from top to bottom; it and must see an assignment statement to a variable before that variable is used in an expression.

Testing is an integral part of writing Python scripts—or really any kind of program—effectively. Run your script frequently as you write it, perhaps as often as after you complete each line of code. This allows syntax errors to be identified and fixed early and helps focus the author's attention on what should be happening at each step of the script.

Questions & Exercises

1. What makes Python an *interpreted* programming language?
2. Write Python statements that print the following:
   a. The words "Computer Science Rocks", followed by an exclamation point
   b. The number 42
   c. An approximation of the value of Pi to 4 decimal places
3. Write Python statements to make the following assignments to variables:
   a. The word "programmer" to a variable called, `rockstar`
   b. The number of seconds in an hour to a variable called `seconds_per_hour`
   c. The average temperature of the human body to a variable called `bodyTemp`
4. Write a Python statement that given an existing variable called `bodyTemp` in degrees Fahrenheit stores the equivalent temperature in degrees Celsius to a new variable called `metricBodyTemp`.

1.9 Data Compression

For the purpose of storing or transferring data, it is often helpful (and sometimes mandatory) to reduce the size of the data involved while retaining the underlying information. The technique for accomplishing this is called **data compression.** We begin this section by considering some generic data compression methods and then look at some approaches designed for specific applications.

Generic Data Compression Techniques

Data compression schemes fall into two categories. Some are **lossless,** others are **lossy.** Lossless schemes are those that do not lose information in the compression process. Lossy schemes are those that may lead to the loss of information. Lossy techniques often provide more compression than lossless ones and are therefore popular in settings in which minor errors can be tolerated, as in the case of images and audio.

In cases where the data being compressed consist of long sequences of the same value, the compression technique called **run-length encoding,** which is a lossless method, is popular. It is the process of replacing sequences of identical data elements with a code indicating the element that is repeated and the number of times it occurs in the sequence. For example, less space is required to indicate that a bit pattern consists of 253 ones, followed by 118 zeros, followed by 87 ones than to actually list all 458 bits.

Another lossless data compression technique is **frequency-dependent encoding,** a system in which the length of the bit pattern used to represent a data item is inversely related to the frequency of the item's use. Such codes are examples of variable-length codes, meaning that items are represented by patterns of different lengths. David Huffman is credited with discovering an algorithm that is commonly used for developing frequency-dependent codes, and it is common practice to refer to codes developed in this manner as **Huffman codes.** In turn, most frequency-dependent codes in use today are Huffman codes.

As an example of frequency-dependent encoding, consider the task of encoded English language text. In the English language the letters *e, t, a,* and i are used more frequently than the letters *z, q,* and *x.* So, when constructing a code for text in the English language, space can be saved by using short bit patterns to represent the former letters and longer bit patterns to represent the latter ones. The result would be a code in which English text would have shorter representations than would be obtained with uniform-length codes.

In some cases, the stream of data to be compressed consists of units, each of which differs only slightly from the preceding one. An example would be consecutive frames of a motion picture. In these cases, techniques using **relative encoding,** also known as **differential encoding,** are helpful. These techniques record the differences between consecutive data units rather than entire units; that is, each unit is encoded in terms of its relationship to the previous unit. Relative encoding can be implemented in either lossless or lossy form depending on whether the differences between consecutive data units are encoded precisely or approximated.

Still other popular compression systems are based on **dictionary encoding** techniques. Here the term *dictionary* refers to a collection of building blocks from which the message being compressed is constructed, and the message itself is encoded as a sequence of references to the dictionary. We normally think of dictionary encoding systems as lossless systems, but as we will see in our discussion of image compression, there are times when the entries in the dictionary are only approximations of the correct data elements, resulting in a lossy compression system.

Dictionary encoding can be used by word processors to compress text documents because the dictionaries already contained in these processors for the purpose of spell checking make excellent compression dictionaries. In particular, an entire word can be encoded as a single reference to this dictionary rather than as a sequence of individual characters encoded using a system such as UTF-8. A typical dictionary in a word processor contains approximately 25,000 entries, which means an individual entry can be identified by an integer in the range of 0 to 24,999. This means that a particular entry in the dictionary can be identified by a pattern of only 15 bits. In contrast, if the word being referenced consisted of six letters, its character-by-character encoding would require 48 bits using UTF-8.

A variation of dictionary encoding is **adaptive dictionary encoding** (also known as dynamic dictionary encoding). In an adaptive dictionary encoding system, the dictionary is allowed to change during the encoding process. A popular example is **Lempel-Ziv-Welsh (LZW) encoding** (named after its creators, Abraham Lempel, Jacob Ziv, and Terry Welsh). To encode a message using LZW, one starts with a dictionary containing the basic building blocks from which the message is constructed, but as larger units are found in the message, they are added to the dictionary—meaning that future occurrences of those units can be encoded as single, rather than multiple, dictionary references. For example, when encoding English text, one could start with a dictionary containing individual characters, digits, and punctuation marks. But as words in the message are identified, they could be added to the dictionary. Thus, the dictionary would grow as the message is encoded, and as the dictionary grows, more words (or recurring patterns of words) in the message could be encoded as single references to the dictionary.

The result would be a message encoded in terms of a rather large dictionary that is unique to that particular message. But this large dictionary would not have to be present to decode the message. Only the original small dictionary would be needed. Indeed, the decoding process could begin with the same small dictionary with which the encoding process started. Then, as the decoding process continues, it would encounter the same units found during the encoding process, and thus be able to add them to the dictionary for future reference just as in the encoding process.

To clarify, consider applying LZW encoding to the message

```
xyx xyx xyx xyx
```

starting with a dictionary with three entries, the first being *x,* the second being y, and the third being a space. We would begin by encoding *xyx* as 121, meaning that the message starts with the pattern consisting of the first dictionary entry, followed by the second, followed by the first. Then the space is encoded to produce 1213. But, having reached a space, we know that the preceding string of characters forms a word, and so we add the pattern *xyx* to the dictionary as the fourth entry. Continuing in this manner, the entire message would be encoded as 121343434.

If we were now asked to decode this message, starting with the original three-entry dictionary, we would begin by decoding the initial string 1213 as *xyx* followed by a space. At this point we would recognize that the string *xyx* forms a word and add it to the dictionary as the fourth entry, just as we did during the encoding process. We would then continue decoding the message by recognizing that the 4 in the message refers to this new fourth entry and decode it as the word *xyx,* producing the pattern

```
xyx xyx
```

Continuing in this manner we would ultimately decode the string 121343434 as

```
xyx xyx xyx xyx
```

which is the original message.

Compressing Images

In Section 1.4, we saw how images are encoded using bit map techniques. Unfortunately, the bit maps produced are often very large. In turn, numerous compression schemes have been developed specifically for image representations.

One system known as **GIF** (short for **Graphic Interchange Format** and pronounced "Giff" by some and "Jiff" by others) is a dictionary encoding system that was developed by CompuServe. It approaches the compression problem by reducing the number of colors that can be assigned to a pixel to only 256. The red-green-blue combination for each of these colors is encoded using three bytes, and these 256 encodings are stored in a table (a dictionary) called the palette. Each pixel in an image can then be represented by a single byte whose value indicates which of the 256 palette entries represents the pixel's color. (Recall that a single byte can contain any one of 256 different bit patterns.) Note that GIF is a lossy compression system when applied to arbitrary images because the colors in the palette may not be identical to the colors in the original image.

GIF can obtain additional compression by extending this simple dictionary system to an adaptive dictionary system using LZW techniques. In particular, as patterns of pixels are encountered during the encoding process, they are added to the dictionary so that future occurrences of these patterns can be encoded more efficiently. Thus, the final dictionary consists of the original palette and a collection of pixel patterns.

One of the colors in a GIF palette is normally assigned the value "transparent," which means that the background is allowed to show through each region assigned that "color." This option, combined with the relative simplicity of the GIF system, makes GIF a logical choice in simple animation applications in which multiple images must move around on a computer screen. On the other hand, its ability to encode only 256 colors renders it unsuitable for applications in which higher precision is required, as in the field of photography.

Another popular compression system for images is **JPEG** (pronounced "JAY-peg"). It is a standard developed by the **Joint Photographic Experts Group** (hence the standard's name) within ISO. JPEG has proved to be an effective standard for compressing color photographs and is widely used in the photography industry, as witnessed by the fact that most digital cameras use JPEG as their default compression technique.

The JPEG standard actually encompasses several methods of image compression, each with its own goals. In those situations that require the utmost in precision, JPEG provides a lossless mode. However, JPEG's lossless mode does not produce high levels of compression when compared to other JPEG options. Moreover, other JPEG options have proven very successful, meaning that JPEG's lossless mode is rarely used. Instead, the option known as JPEG's baseline standard (also known as JPEG's lossy sequential mode) has become the standard of choice in many applications.

Image compression using the JPEG baseline standard requires a sequence of steps, some of which are designed to take advantage of a human eye's limitations. In particular, the human eye is more sensitive to changes in brightness than to changes in color. So, starting from an image that is encoded in terms of luminance and chrominance components, the first step is to average the chrominance values over two-by-two pixel squares. This reduces the size of the chrominance information by a factor of four while preserving all the original brightness information. The result is a significant degree of compression without a noticeable loss of image quality.

The next step is to divide the image into eight-by-eight pixel blocks and to compress the information in each block as a unit. This is done by applying a mathematical technique known as the discrete cosine transform, whose details need not concern us here. The important point is that this transformation converts the original eight-by-eight block into another block whose entries reflect how the pixels in the original block relate to each other rather than the actual pixel values. Within this new block, values below a predetermined threshold are then replaced by zeros, reflecting the fact that the changes represented by these values are too subtle to be detected by the human eye. For example, if the original block contained a checkerboard pattern, the new block might reflect a uniform average color. (A typical eight-by-eight pixel block would represent a very small square within the image so the human eye would not identify the checkerboard appearance anyway.)

At this point, more traditional run-length encoding, relative encoding, and variable-length encoding techniques are applied to obtain additional compression. All together, JPEG's baseline standard normally compresses color images by a factor of at least 10, and often by as much as 30, without noticeable loss of quality.

Still another data compression system associated with images is **TIFF** (short for **Tagged Image File Format**). However, the most popular use of TIFF is not as a means of data compression but instead as a standardized format for storing photographs along with related information such as date, time, and camera settings. In this context, the image itself is normally stored as red, green, and blue pixel components without compression.

The TIFF collection of standards does include data compression techniques, most of which are designed for compressing images of text documents in facsimile applications. These use variations of run-length encoding to take advantage of the fact that text documents consist of long strings of white pixels. The color image compression option included in the TIFF standards is based on techniques similar to those used by GIF and are therefore not widely used in the photography community.

Compressing Audio and Video

The most commonly used standards for encoding and compressing audio and video were developed by the **Motion Picture Experts Group (MPEG)** under the leadership of ISO. In turn, these standards themselves are called MPEG.

MPEG encompasses a variety of standards for different applications. For example, the demands for high definition television (HDTV) broadcast are distinct from those for video conferencing, in which the broadcast signal must find its way over a variety of communication paths that may have limited capabilities. Both of these applications differ from that of storing video in such a manner that sections can be replayed or skipped over.

The techniques employed by MPEG are well beyond the scope of this text, but in general, video compression techniques are based on video being constructed as a sequence of pictures in much the same way that motion pictures are recorded on film. To compress such sequences, only some of the pictures, called I-frames, are encoded in their entirety. The pictures between the I-frames are encoded using relative encoding techniques. That is, rather than encode the entire picture, only its distinctions from the prior image are recorded. The I-frames themselves are usually compressed with techniques similar to JPEG.

The best known system for compressing audio is **MP3,** which was developed within the MPEG standards. In fact, the acronym *MP3* is short for

MPEG layer 3. Among other compression techniques, MP3 takes advantage of the properties of the human ear, removing those details that the human ear cannot perceive. One such property, called **temporal masking,** is that for a short period after a loud sound, the human ear cannot detect softer sounds that would otherwise be audible. Another, called **frequency masking,** is that a sound at one frequency tends to mask softer sounds at nearby frequencies. By taking advantage of such characteristics, MP3 can be used to obtain significant compression of audio while maintaining near CD quality sound.

Using MPEG and MP3 compression techniques, video cameras are able to record as much as an hour's worth of video within 128MB of storage, and portable music players can store as many as 400 popular songs in a single GB. But, in contrast to the goals of compression in other settings, the goal of compressing audio and video is not necessarily to save storage space. Just as important is the goal of obtaining encodings that allow information to be transmitted over today's communication systems fast enough to provide timely presentation. If each video frame required a MB of storage and the frames had to be transmitted over a communication path that could relay only one KB per second, there would be no hope of successful video conferencing. Thus, in addition to the quality of reproduction allowed, audio and video compression systems are often judged by the transmission speeds required for timely data communication. These speeds are normally 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). Using MPEG techniques, video presentations can be successfully relayed over communication paths that provide transfer rates of 40 Mbps. MP3 recordings generally require transfer rates of no more than 64 Kbps.

Questions & Exercises

1. List four generic compression techniques.
2. What would be the encoded version of the message

   ```
   xyx yxxxy xyx yxxxy yxxxy
   ```

   if LZW compression, starting with the dictionary containing *x*, *y*, and a space (as described in the text), were used?
3. Why would GIF be better than JPEG when encoding color cartoons?
4. Suppose you were part of a team designing a spacecraft that will travel to other planets and send back photographs. Would it be a good idea to compress the photographs using GIF or JPEG's baseline standard to reduce the resources required to store and transmit the images?
5. What characteristic of the human eye does JPEG's baseline standard exploit?
6. What characteristic of the human ear does MP3 exploit?
7. Identify a troubling phenomenon that is common when encoding numeric information, images, and sound as bit patterns.

1.10 Communication Errors

When information is transferred back and forth among the various parts of a computer, or transmitted from the earth to the moon and back, or, for that matter, merely left in storage, a chance exists that the bit pattern ultimately retrieved may not be identical to the original one. Particles of dirt or grease on a magnetic recording surface or a malfunctioning circuit may cause data to be incorrectly recorded or read. Static on a transmission path may corrupt portions of the data. In the case of some technologies, normal background radiation can alter patterns stored in a machine's main memory.

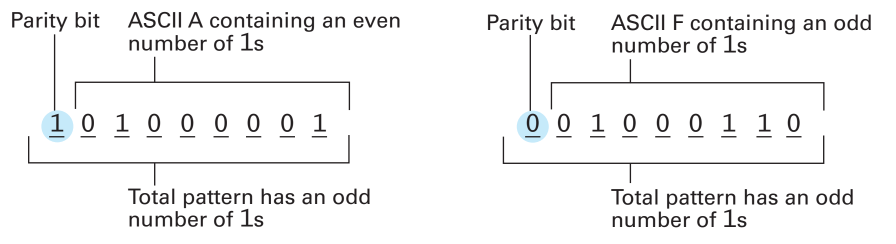
To resolve such problems, a variety of encoding techniques have been developed to allow the detection and even the correction of errors. Today, because these techniques are largely built into the internal components of a computer system, they are not apparent to the personnel using the machine. Nonetheless, their presence is important and represents a significant contribution to scientific research. It is fitting, therefore, that we investigate some of these techniques that lie behind the reliability of today's equipment.

Parity Bits

A simple method of detecting errors is based on the principle that if each bit pattern being manipulated has an odd number of 1s and a pattern with an even number of 1s is encountered, an error must have occurred. To use this principle, we need an encoding system in which each pattern contains an odd number of 1s. This is easily obtained by first adding an additional bit, called a **parity bit,** to each pattern in an encoding system already available (perhaps at the high-order end). In each case, we assign the value 1 or 0 to this new bit so that the entire resulting pattern has an odd number of 1s. Once our encoding system has been modified in this way, a pattern with an even number of 1s indicates that an error has occurred and that the pattern being manipulated is incorrect.

Figure 1.26 demonstrates how parity bits could be added to the ASCII codes for the letters A and F. Note that the code for A becomes 101000001 (parity bit 1) and the ASCII for F becomes 001000110 (parity bit 0). Although the original 8-bit pattern for A has an even number of 1s and the original 8-bit pattern for F has an odd number of 1s, both the 9-bit patterns have an odd number of 1s. If this technique were applied to all the 8-bit ASCII patterns, we would obtain a 9-bit encoding system in which an error would be indicated by any 9-bit pattern with an even number of 1s.

Figure 1.26 The ASCII codes for the letters A and F adjusted for odd parity


Parity bit
ASCII A containing an even number of 1s
1 0 1 0 0 0 0 0 1
Total pattern has an odd number of 1s
Parity bit
ASCII F containing an odd number of 1s
0 0 1 0 0 0 1 1 0
Total pattern has an odd number of 1s

The parity system just described is called **odd parity,** because we designed our system so that each correct pattern contains an odd number of 1s. Another technique is called **even parity.** In an even parity system, each pattern is designed to contain an even number of 1s, and thus an error is signaled by the occurrence of a pattern with an odd number of 1s.

Today it is not unusual to find parity bits being used in a computer's main memory. Although we envision these machines as having memory cells of 8-bit capacity, in reality each has a capacity of 9 bits, 1 bit of which is used as a parity bit. Each time an 8-bit pattern is given to the memory circuitry for storage, the circuitry adds a parity bit and stores the resulting 9-bit pattern. When the pattern is later retrieved, the circuitry checks the parity of the 9-bit pattern. If this does not indicate an error, then the memory removes the parity bit and confidently returns the remaining 8-bit pattern. Otherwise, the memory returns the 8 data bits with a warning that the pattern being returned may not be the same pattern that was originally entrusted to memory.

The straightforward use of parity bits is simple, but it has its limitations. If a pattern originally has an odd number of 1s and suffers two errors, it will still have an odd number of 1s, and thus the parity system will not detect the errors. In fact, straightforward applications of parity bits fail to detect any even number of errors within a pattern.

One means of minimizing this problem is sometimes applied to long bit patterns, such as the string of bits recorded in a sector on a magnetic disk. In this case the pattern is accompanied by a collection of parity bits making up a **checkbyte.** Each bit within the checkbyte is a parity bit associated with a particular collection of bits scattered throughout the pattern. For instance, one parity bit may be associated with every eighth bit in the pattern starting with the first bit, while another may be associated with every eighth bit starting with the second bit. In this manner, a collection of errors concentrated in one area of the original pattern is more likely to be detected, since it will be in the scope of several parity bits. Variations of this checkbyte concept lead to error detection schemes known as **checksums** and **cyclic redundancy checks (CRC).**

Error-Correcting Codes

Although the use of a parity bit allows the detection of an error, it does not provide the information needed to correct the error. Many people are surprised that **error-correcting codes** can be designed so that errors can be not only detected but also corrected. After all, intuition says that we cannot correct errors in a received message unless we already know the information in the message. However, a simple code with such a corrective property is presented in Figure 1.27.

To understand how this code works, we first define the term **Hamming distance,** which is named after R. W. Hamming, who pioneered the search for error-correcting codes after becoming frustrated with the lack of reliability of the early relay machines of the 1940s. The Hamming distance between two bit patterns is the number of bits in which the patterns differ. For example, the Hamming distance between the patterns representing A and B in the code in Figure 1.27 is four, and the Hamming distance between B and C is three. The

Figure 1.27 An error-correcting code

Symbol	Code
A	000000
B	001111
C	010011
D	011100
E	100110
F	101001
G	110101
H	111010

important feature of the code in Figure 1.27 is that any two patterns are separated by a Hamming distance of at least three.

If a single bit is modified in a pattern from Figure 1.27, the error can be detected since the result will not be a legal pattern. (We must change at least 3 bits in any pattern before it will look like another legal pattern.) Moreover, we can also figure out what the original pattern was. After all, the modified pattern will be a Hamming distance of only one from its original form but at least two from any of the other legal patterns.

Thus, to decode a message that was originally encoded using Figure 1.27, we simply compare each received pattern with the patterns in the code until we find one that is within a distance of one from the received pattern. We consider this to be the correct symbol for decoding. For example, if we received the bit pattern 010100 and compared this pattern to the patterns in the code, we would obtain the table in Figure 1.28. Thus, we would conclude that the character transmitted must have been a D because this is the closest match.

Figure 1.28 Decoding the pattern 010100 using the code in Figure 1.27

Character	Code	Pattern received	Distance between received pattern and code	
A	0 0 0 0 0 0	0 **1** 0 **1** 0 0	2	
B	0 0 1 1 1 1	0 **1** **0** 1 **0** **0**	4	
C	0 1 0 0 1 1	0 1 0 **1** **0** **0**	3	
D	0 1 1 1 0 0	0 1 **0** 1 0 0	**1**	Smallest distance
E	1 0 0 1 1 0	**0** **1** 0 1 **0** 0	3	
F	1 0 1 0 0 1	**0** **1** **0** **1** 0 **0**	5	
G	1 1 0 1 0 1	**0** 1 0 1 0 **0**	2	
H	1 1 1 0 1 0	**0** 1 **0** 1 **0** 0	4	

You will observe that using this technique with the code in Figure 1.27 actually allows us to detect up to two errors per pattern and to correct one error. If we designed the code so that each pattern was a Hamming distance of at least five from each of the others, we would be able to detect up to four errors per pattern and correct up to two. Of course, the design of efficient codes associated with large Hamming distances is not a straightforward task. In fact, it constitutes a part of the branch of mathematics called algebraic coding theory, which is a subject within the fields of linear algebra and matrix theory.

Error-correcting techniques are used extensively to increase the reliability of computing equipment. For example, they are often used in high-capacity magnetic disk drives to reduce the possibility that flaws in the magnetic surface will corrupt data. Moreover, a major distinction between the original CD format used for audio disks and the later format used for computer data storage is in the degree of error correction involved. CD-DA format incorporates error-correcting features that reduce the error rate to only one error for two CDs. This is quite adequate for audio recordings, but a company using CDs to supply software to customers would find that flaws in 50 percent of the disks would be intolerable. Thus, additional error-correcting features are employed in CDs used for data storage, reducing the probability of error to one in 20,000 disks.

Questions & Exercises

1. The following bytes were originally encoded using odd parity. In which of them do you know that an error has occurred?

   a. 100101101 b. 100000001 c. 000000000
   d. 111000000 e. 011111111

2. Could errors have occurred in a byte from question 1 without your knowing it? Explain your answer.

3. How would your answers to questions 1 and 2 change if you were told that even parity had been used instead of odd?

4. Encode these sentences in ASCII using odd parity by adding a parity bit at the high-order end of each character code:

   a. "Stop!" Cheryl shouted. b. Does 2 + 3 = 5?

5. Using the error-correcting code presented in Figure 1.27, decode the following messages:

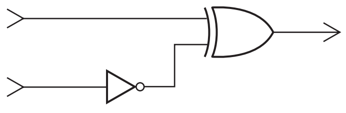
   a. 001111 100100 001100 b. 010001 000000 001011
   c. 011010 110110 100000 011100

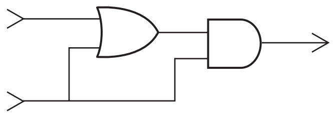
6. Construct a code for the characters A, B, C, and D using bit patterns of length five so that the Hamming distance between any two patterns is at least three.

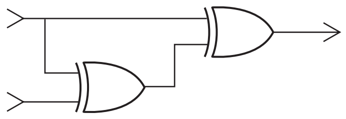
Chapter Review Problems

(Asterisked problems are associated with optional sections.)

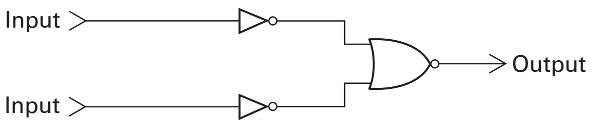
1. Determine the output of each of the following circuits, assuming that the upper input is 1 and the lower input is 0. What would be the output when upper input is 0 and the lower input is 1?

a. 

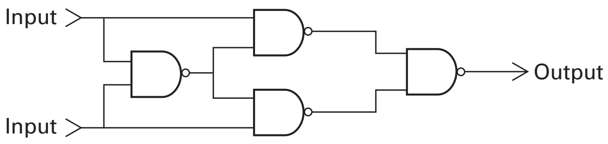
b. 

c. 

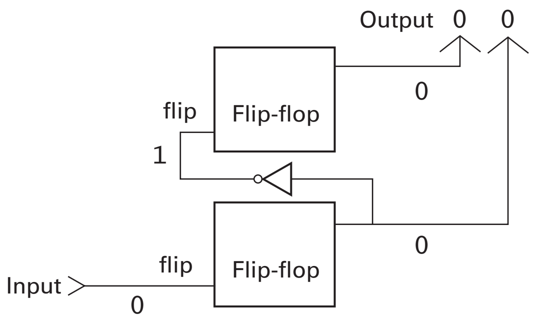
2. a. What Boolean operation does the circuit compute?



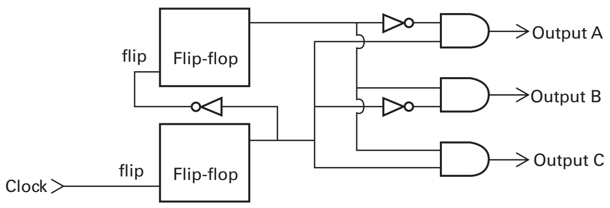
b. What Boolean operation does the circuit compute?



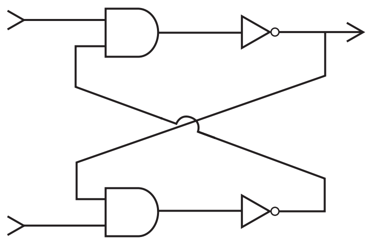
*3. a. If we were to purchase a flip-flop circuit from an electronic component store, we may find that it has an additional input called *flip*. When this input changes from a 0 to 1, the output flips state (if it was 0 it is now 1 and vice versa). However, when the flip input changes from 1 to a 0, nothing happens. Even though we may not know the details of the circuitry needed to accomplish this behavior, we could still use this device as an abstract tool in other circuits. Consider the circuitry using two of the following flip-flops. If a pulse were sent on the circuit's input, the bottom flip-flop would change state. However, the second flip-flop would not change, since its input (received from the output of the NOT gate) went from a 1 to a 0. As a result, this circuit would now produce the outputs 0 and 1. A second pulse would flip the state of both flip-flops, producing an output of 1 and 0. What would be the output after a third pulse? After a fourth pulse?



b. It is often necessary to coordinate activities of various components within a computer. This is accomplished by connecting a pulsating signal (called a clock) to circuitry similar to part a. Additional gates (as shown) send signals in a coordinated fashion to other connected circuits. On studying this circuit, you should be able to confirm that on the 1st, 5th, 9th . . . pulses of the clock, a 1 will be sent on output A. On what pulses of the clock will a 1 be sent on output B? On what pulses of the clock will a 1 be sent on output C? On which output is a 1 sent on the 4th pulse of the clock?



4. Assume that both of the inputs in the following circuit are 1. Describe what would happen if the upper input were temporarily changed to 0. Describe what would happen if the lower input were temporarily changed to 0. Redraw the circuit using NAND gates.



5. The following table represents the addresses and contents (using hexadecimal notation) of some cells in a machine's main memory. Starting with this memory arrangement, follow the sequence of instructions and record the final contents of each of these memory cells:

Address	Contents
00	AB
01	53
02	D6
03	02

Step 1. Move the contents of the cell whose address is 03 to the cell at address 00.

Step 2. Move the value 01 into the cell at address 02.

Step 3. Move the value stored at address 01 into the cell at address 03.

6. How many cells can be in a computer's main memory if each cell's address can be represented by two hexadecimal digits? What if four hexadecimal digits are used?

7. What bit patterns are represented by the following hexadecimal notations?

a. 8A9　b. DCB　c. EF3
d. A01　e. C99

8. What is the value of the least significant bit in the bit patterns represented by the following hexadecimal notations?

a. 9A　b. 90
c. 1B　d. 6E

9. Express the following bit patterns in hexadecimal notation:

a. 101101001011010010 11

b. 000111100001

c. 1111111011011011

10. Suppose a digital camera has a storage capacity of 500MB. How many black-and-white photographs could be stored in the camera if each consisted of 512 pixels per row and 512 pixels per column if each pixel required one bit of storage?

11. Suppose an image is represented on a display screen by a square array containing 256 columns and 256 rows of pixels. If for each pixel, 3 bytes are required to encode the color and 8 bits to encode the intensity, how many byte-size memory cells are required to hold the entire picture?

12. a. What are the advantages, if any, of using zoned-bit recording?

b. What is the difference between seek time and access time?

13. Suppose that you want to create a backup of your entire data which is around 10GB. Would it be reasonable to use DVDs for the purpose of creating this backup? What about BDs (Blu-ray Disks)?

14. If each sector on a magnetic disk can store 512 bytes of data, how many sectors are required to store two pages of integers (perhaps 10 lines of 100 integers in each page) if every integer is represented by using four bytes?

15. How many bytes of storage space would be required to store a 20-page document containing details of employees, in which each page contains 100 records and every record is of 200 characters, if two byte Unicode characters were used?

16. In zoned-bit recording, why does the rate of data transfer vary, depending on the portion of the disk being used?

17. What is the average access time for a hard disk which has a rotation delay of 10 milliseconds and a seek time of 9 milliseconds?

18. Suppose a disk storage system consists of 5 platters with 10 tracks on each side and

8 sectors in each track. What is the capacity of the system? Assume every sector contains 512 bytes and data can be stored on both surfaces of each platter.

19. Here is a message in ASCII. What does it say?
```
01000011 01101111 01101101 01110000
01110101 01110100 01100101 01110010
00100000 01010011 01100011 01101001
01100101 01101110 01100011 01100101
00100001
```

20. The following two messages are encoded in ASCII using one byte per character and then represented in hexadecimal notation. Are both the messages same?
```
436F6D7075746572    436F6D7075736572
```

21. Encode the following sentences in ASCII using one byte per character.
a. Is 1 byte = 8 bits?
b. Yes, a byte contains 8 bits!

22. Combine the two sentences of the previous problem and express it in hexadecimal notation.

23. List the hexadecimal representations of the integers from 20 to 27.

24. a. Write the number 100 by representing 1 and 0 in ASCII.
b. Write the number 255 in binary representation.

25. What values have binary representations in which only one of the bits is 1? List the binary representations for the smallest six values with this property.

***26.** Convert each of the following hexadecimal representations to binary representation and then to its equivalent base 10 representation:
a. A   b. 14   c. 1E
d. 28   e. 32   f. 3C
g. 46   h. 65   i. CA
j. 12F   k. 194   l. 1F9

***27.** Convert each of the following base 10 representations to its equivalent binary representation:
a. 110   b. 99   c. 72
d. 81   e. 36

***28.** Convert each of the following excess 32 representations to its equivalent base 10 representation:
a. 011111   b. 100110   c. 111000
d. 000101   e. 010101

***29.** Convert each of the following base 10 representations to its equivalent excess sixteen representation:
a. −12   b. 0   c. 10
d. −8   e. 9

***30.** Convert each of the following two's complement representations to its equivalent base 10 representation:
a. 010101   b. 101010   c. 110110
d. 011011   e. 111001

***31.** Convert each of the following base 10 representations to its equivalent two's complement representation in which each value is represented in 8 bits:
a. −27   b. 3   c. 21
d. 8   e. −18

***32.** Perform each of the following additions assuming the bit strings represent values in two's complement notation. Identify each case in which the answer is incorrect because of overflow.
a. 00101 + 01000   b. 11111 + 00001
c. 01111 + 00001   d. 10111 + 11010
e. 11111 + 11111   f. 00111 + 01100

***33.** Solve each of the following problems by translating the values into two's complement notation (using patterns of 5 bits), converting any subtraction problem to an equivalent addition problem, and performing that addition. Check your work by converting your answer to base 10 notation. (Watch out for overflow.)
a. 5 + 1   b. 5 − 1   c. 12 − 5
d. 8 − 7   e. 12 + 5   f. 5 − 11

***34.** Convert each of the following binary representations into its equivalent base 10 representation:
a. 11.11   b. 100.0101   c. 0.1101
d. 1.0   e. 10.01

***35.** Express each of the following values in binary notation:
a. $5\frac{3}{4}$ b. $15\frac{15}{16}$ c. $5\frac{3}{8}$
d. $1\frac{1}{4}$ e. $6\frac{5}{8}$

***36.** Decode the following bit patterns using the floating-point format described in Figure 1.24:
a. 01011001   b. 11001000
c. 10101100   d. 00111001

***37.** Encode the following values using the 8-bit floating-point format described in Figure 1.24. Indicate each case in which a truncation error occurs.
a. $-7\frac{1}{2}$ b. $\frac{1}{2}$ c. $-3\frac{3}{4}$
d. $\frac{7}{32}$ e. $\frac{31}{32}$

***38.** Assuming you are not restricted to using normalized form, list all the bit patterns that could be used to represent the value 3/8 using the floating-point format described in Figure 1.24.

***39.** What is the best approximation to the square root of 2 that can be expressed in the 8-bit floating-point format described in Figure 1.24? What value is actually obtained if this approximation is squared by a machine using this floating-point format?

***40.** What is the best approximation to the value one-tenth that can be represented using the 8-bit floating-point format described in Figure 1.24?

***41.** Explain how errors can occur when measurements using the metric system are recorded in floating-point notation. For example, what if 110 cm was recorded in units of meters?

***42.** One of the bit patterns 01011 and 11011 represents a value stored in excess 16 notation and the other represents the same value stored in two's complement notation.
a. What can be determined about this common value?
b. What is the relationship between a pattern representing a value stored in two's complement notation and the pattern representing the same value stored in excess notation when both systems use the same bit pattern length?

***43.** The three bit patterns 10000010, 01101000, and 00000010 are representations of the same value in two's complement, excess, and the 8-bit floating-point format presented in Figure 1.24, but not necessarily in that order. What is the common value, and which pattern is in which notation?

***44.** Which of the following values cannot be represented accurately in the floating-point format introduced in Figure 1.24?
a. $6\frac{1}{2}$ b. $\frac{13}{16}$ c. 9
d. $\frac{17}{32}$ e. $\frac{15}{16}$

***45.** If you changed the length of the bit strings being used to represent integers in binary from 4 bits to 6 bits, what change would be made in the value of the largest integer you could represent? What if you were using two's complement notation?

***46.** What would be the hexadecimal representation of the largest memory address in a memory consisting of 4MB if each cell had a one-byte capacity?

***47.** What would be the encoded version of the message

```
xxy yyx xxy xxy yyx
```

if LZW compression, starting with the dictionary containing x, y, and a space (as described in Section 1.8), were used?

***48.** The following message was compressed using LZW compression with a dictionary whose first, second, and third entries are x, y, and space, respectively. What is the decompressed message?

```
22123113431213536
```

***49.** If the message

```
xxy yyx xxy xxyy
```

were compressed using LZW with a starting dictionary whose first, second, and third entries were x, y, and space, respectively, what would be the entries in the final dictionary?

***50.** As we will learn in the next chapter, one means of transmitting bits over traditional telephone systems is to convert the bit patterns into sound, transfer the sound over the telephone lines, and then convert the sound back into bit patterns. Such techniques are limited to transfer rates of 57.6 Kbps. Is this sufficient for teleconferencing if the video is compressed using MPEG?

***51.** Encode the following sentences in ASCII using even parity by adding a parity bit at the high-order end of each character code:
a. Does 100/5 = 20?
b. The total cost is $7.25.

***52.** The following message was originally transmitted with odd parity in each short bit string. In which strings have errors definitely occurred?

```
11001 11011 10110 00000 11111 10001
10101 00100 01110
```

*53. Suppose a 24-bit code is generated by representing each symbol by three consecutive copies of its ASCII representation (for example, the symbol A is represented by the bit string 010000010100000101000001). What error-correcting properties does this new code have?

*54. Using the error-correcting code described in Figure 1.28, decode the following words:
a. 111010 110110
b. 101000 100110 001100
c. 011101 000110 000000 010100
d. 010010 001000 001110 101111
   000000 110111 100110
e. 010011 000000 101001 100110

*55. International currency exchange rates change frequently. Investigate current exchange rates, and update the currency converter script from Section 1.8 accordingly.

*56. Find another currency not already included in the currency converter from Section 1.8. Acquire its current conversion rate and find its Unicode currency symbol on the web. Extend the script to convert this new currency.

*57. If your web browser and text editor properly support Unicode and UTF-8, copy/paste the actual international currency symbols into the converter script of Section 1.8, in place of the cumbersome codes like, '\u00A3'. (If your software has trouble handling Unicode, you may get strange symbols in your text editor when you try to do this.)

*58. The currency converter script of Section 1.8 uses the variable `dollars` to store the amount of money to be converted before performing each of the multiplications. This made the script one line longer than simply typing the integer quantity 1000 directly into each of the multiplication calculations. Why is it advantageous to create this extra variable ahead of time?

*59. Write and test a Python script that given a number of bytes outputs the equivalent number of kilobytes, megabytes, gigabytes, and terabytes. Write and test a complementary script that given a number of terabytes outputs the equivalent number of GB, MB, KB, and bytes.

*60. Write and test a Python script that given a number of minutes and seconds for a recording calculates the number of bits used to encode uncompressed, CD-quality stereo audio data of that length. (Review Section 1.4 for the necessary parameters and equations.)

*61. Identify the error(s) in this Python script.
```
days_per_week  = 7
weeks_per_year  = 52
days_per_year  = days_per_week **
                weeks_per_year
PRINT(days_per_year)
```

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. A truncation error has occurred in a critical situation, causing extensive damage and loss of life. Who is liable, if anyone? The designer of the hardware? The designer of the software? The programmer who actually wrote that part of the program? The person who decided to use the software in that particular application? What if the software had been corrected by the company that originally developed it, but that update had not been purchased and applied in the critical application? What if the software had been pirated?

2. Is it acceptable for an individual to ignore the possibility of truncation errors and their consequences when developing his or her own applications?
3. Was it ethical to develop software in the 1970s using only two digits to represent the year (such as using 76 to represent the year 1976), ignoring the fact that the software would be flawed as the turn of the century approached? Is it ethical today to use only three digits to represent the year (such as 982 for 1982 and 015 for 2015)? What about using only four digits?
4. Many argue that encoding information often dilutes or otherwise distorts the information, since it essentially forces the information to be quantified. They argue that a questionnaire in which subjects are required to record their opinions by responding within a scale from one to five is inherently flawed. To what extent is information quantifiable? Can the pros and cons of different locations for a waste disposal plant be quantified? Is the debate over nuclear power and nuclear waste quantifiable? Is it dangerous to base decisions on averages and other statistical analysis? Is it ethical for news agencies to report polling results without including the exact wording of the questions? Is it possible to quantify the value of a human life? Is it acceptable for a company to stop investing in the improvement of a product, even though additional investment could lower the possibility of a fatality relating to the product's use?
5. Should there be a distinction in the rights to collect and disseminate data depending on the form of the data? That is, should the right to collect and disseminate photographs, audio, or video be the same as the right to collect and disseminate text?
6. Whether intentional or not, a report submitted by a journalist usually reflects that journalist's bias. Often by changing only a few words, a story can be given either a positive or negative connotation. (Compare, "The majority of those surveyed opposed the referendum." to "A significant portion of those surveyed supported the referendum.") Is there a difference between altering a story (by leaving out certain points or carefully selecting words) and altering a photograph?
7. Suppose that the use of a data compression system results in the loss of subtle but significant items of information. What liability issues might be raised? How should they be resolved?

Additional Reading

Drew, M., and Z. Li. *Fundamentals of Multimedia.* Upper Saddle River, NJ: Prentice-Hall, 2004.

Halsall, F. *Multimedia Communications.* Boston, MA: Addison-Wesley, 2001.

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. 2, 3rd ed. Boston, MA: Addison-Wesley, 1998.

Long, B. *Complete Digital Photography,* 3rd ed. Hingham, MA: Charles River Media, 2005.

Miano, J. *Compressed Image File Formats.* New York: ACM Press, 1999.

Petzold, C. *CODE: The Hidden Language of Computer Hardware and Software.* Redman, WA: Microsoft Press, 2000.

Salomon, D. *Data Compression: The Complete Reference,* 4th ed. New York: Springer, 2007.

Sayood, K. *Introduction to Data Compression,* 3rd ed. San Francisco, CA: Morgan Kaufmann, 2005.