ISBN/ISSN: 0132067110

DIGITAL SYSTEMS AND INFORMATION

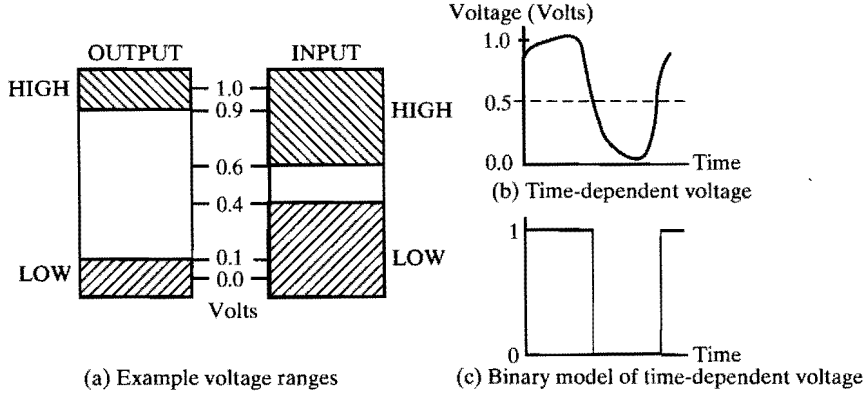
This book deals with logic circuits and digital computers. Early computers were used for computations with discrete numeric elements called *digits* (the Latin word for fingers)—hence the term *digital computer*. The use of "digital" spread from the computer to logic circuits and other systems that use discrete elements of information, giving us the terms *digital circuits* and *digital systems*. The term *logic* is applied to circuits that operate on a set of just two elements with values True (1) and False (0). Since computers are based on logic circuits, they operate on patterns of elements from these two-valued sets, which are used to represent, among other things, the decimal digits. Today, the term "digital circuits" is viewed as synonymous with the term "logic circuits."

The *general-purpose digital computer* is a digital system that can follow a stored sequence of instructions, called a *program*, that operates on data. The user can specify and change the program or the data according to specific needs. As a result of this flexibility, general-purpose digital computers can perform a variety of information-processing tasks, ranging over a very wide spectrum of applications. This makes the digital computer a highly general and very flexible digital system. Also, due to its generality, complexity, and widespread use, the computer provides an ideal vehicle for learning the concepts, methods, and tools of digital system design. To this end, we use the exploded pictorial diagram of a computer of the class commonly referred to as a PC (personal computer) given on the opposite page. We employ this generic computer to highlight the significance of the material covered and its relationship to the overall system. A bit later in this chapter, we will discuss the various major components of the generic computer and see how they relate to a block diagram commonly used to represent a computer. Otherwise, the remainder of the chapter focuses on the digital systems in our daily lives and introduces approaches for representing information in digital circuits and systems.

1-1 INFORMATION REPRESENTATION

Digital systems store, move, and process information. The information represents a broad range of phenomena from the physical and man-made world. The physical world is characterized by parameters such as weight, temperature, pressure, velocity, flow, and sound intensity and frequency. Most physical parameters are *continuous*, typically capable of taking on all possible values over a defined range. In contrast, in the man-made world, parameters can be discrete in nature, such as business records using words, quantities, and currencies, taking on values from an alphabet, the integers, or units of currency, respectively. In general, information systems must be able to represent both continuous and discrete information. Suppose that temperature, which is continuous, is measured by a sensor and converted to an electrical voltage, which is likewise continuous. We refer to such a continuous voltage as an *analog signal*, which is one possible way to represent temperature. But, it is also possible to represent temperature by an electrical voltage that takes on *discrete* values that occupy only a finite number of values over a range, e.g., corresponding to integer degrees centigrade between −40 and +119. We refer to such a voltage as a *digital signal*. Alternatively, we can represent the discrete values by multiple voltage signals, each taking on a discrete value. At the extreme, each signal can be viewed as having only two discrete values, with multiple signals representing a large number of discrete values. For example, each of the 160 values just mentioned for temperature can be represented by a particular combination of eight two-valued signals. The signals in most present-day electronic digital systems use just two discrete values and are therefore said to be *binary*. The two discrete values used are often called 0 and 1, the digits for the binary number system.

We typically represent the two discrete values by ranges of voltage values called HIGH and LOW. Output and input voltage ranges are illustrated in Figure 1-1(a). The HIGH output voltage value ranges between 0.9 and 1.1 volts, and the LOW output voltage value between −0.1 and 0.1 volts. The high input range allows 0.6 to 1.1 volts to be recognized as a HIGH, and the low input range allows −0.1 to 0.4 volts to be recognized as a LOW. The fact that the input ranges are wider than the



(a) Example voltage ranges

(b) Time-dependent voltage

(c) Binary model of time-dependent voltage

□ **FIGURE 1-1**
Examples of Voltage Ranges and Waveforms for Binary Signals

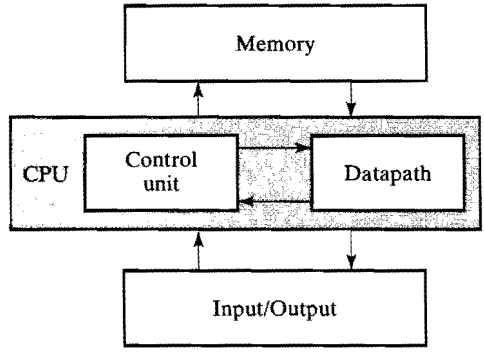
output ranges allows the circuits to function correctly in spite of variations in their behavior and undesirable "noise" voltages that may be added to or subtracted from the outputs.

We give the output and input voltage ranges a number of different names. Among these are HIGH (H) and LOW (L), TRUE (T) and FALSE (F), and 1 and 0. It is natural to associate the higher voltage ranges with HIGH or H, and the lower voltage ranges with LOW or L. For TRUE and 1 and FALSE and 0, however, there is a choice. TRUE and 1 can be associated with either the higher or lower voltage range and FALSE and 0 with the other range. Unless otherwise indicated, we assume that TRUE and 1 are associated with the higher of the voltage ranges, H, and that FALSE and 0 are associated with the lower of the voltage ranges, L. This particular convention is called *positive logic.*

It is interesting to note that the values of voltages for a digital circuit in Figure 1-1(a) are still continuous, ranging from −0.1 to +1.1 volts. Thus, the voltage is actually analog! The actual voltages values for the output of a very high-speed digital circuit are plotted versus time in Figure 1-1(b). Such a plot is referred to as a *waveform.* The interpretation of the voltage as binary is based on a model using voltage ranges to represent discrete values 0 and 1 on the inputs and the outputs. The application of such a model, which redefines all voltage above 0.5 V as 1 and below 0.5 V as 0 in Figure 1-1(b), gives the waveform in Figure 1-1(c). The output has now been interpreted as binary, having only discrete values 1 and 0, with the actual voltage values removed. We note that digital circuits, made up of electronic devices called transistors, are designed to cause the outputs to occupy the two distinct output voltage ranges for 1 (H) and 0 (L) in Figure 1-1, whenever the outputs are not changing. In contrast, analog circuits are designed to have their outputs take on continuous values over their range, whether changing or not.

Since 0 and 1 are associated with the binary number system, they are the preferred names for the signal ranges. A binary digit is called a *bit.* Information is represented in digital computers by groups of bits. By using various coding techniques, groups of bits can be made to represent not only binary numbers, but also other groups of discrete symbols. Groups of bits, properly arranged, can even specify to the computer the program instructions to be executed and the data to be processed.

Why is binary used? In contrast to the situation in Figure 1-1, consider a system with 10 values representing the decimal digits. In such a system, the voltages available—say, 0 to 1.0 volts—could be divided into 10 ranges, each of length 0.1 volt. A circuit would provide an output voltage within each of these 10 ranges. An input of a circuit would need to determine in which of the 10 ranges an applied voltage lies. If we wish to allow for noise on the voltages, then output voltage might be permitted to range over less than 0.05 volt for a given digit representation, and boundaries between inputs could vary by less than 0.05 volt. This would require complex and costly electronic circuits, and the output still could be disturbed by small "noise" voltages or small variations in the circuits occurring during their manufacture or use. As a consequence, the use of such multivalued circuits is very limited. Instead, binary circuits are used in which correct circuit operation can be achieved with significant variations in values of the two output voltages and the



□ **FIGURE 1-2**
Block Diagram of a Digital Computer

two input ranges. The resulting transistor circuit with an output that is either HIGH or LOW is simple, easy to design, and extremely reliable. In addition, this use of binary values makes the results calculated repeatable in the sense that the same set of input values to a calculation always gives exactly the same set of outputs. This is not necessarily the case for multivalued or analog circuits, in which noise voltages and small variations due to manufacture or circuit aging can cause results to differ at different times.

The Digital Computer

A block diagram of a digital computer is shown in Figure 1-2. The memory stores programs as well as input, output, and intermediate data. The datapath performs arithmetic and other data-processing operations as specified by the program. The control unit supervises the flow of information between the various units. A datapath, when combined with the control unit, forms a component referred to as a *central processing unit*, or CPU.

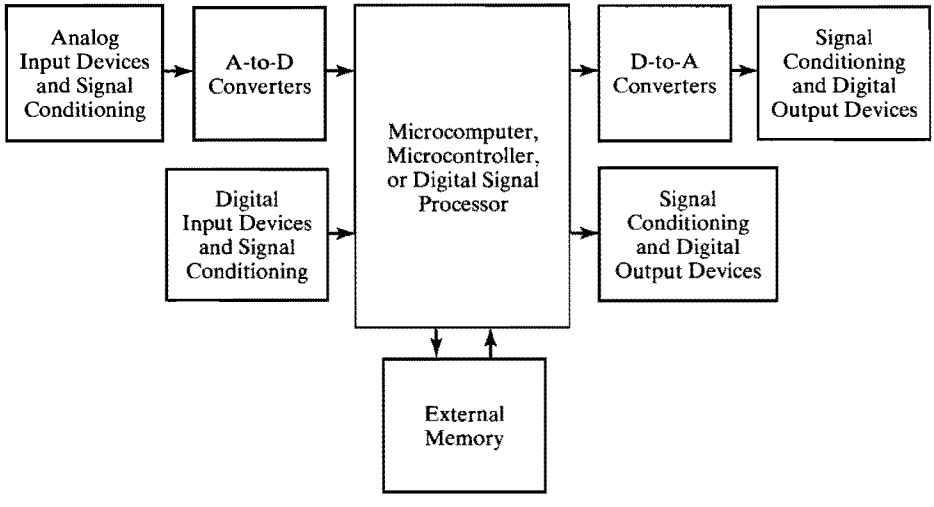
The program and data prepared by the user are transferred into memory by means of an input device such as a keyboard. An output device, such as an LCD (liquid crystal display), displays the results of the computations and presents them to the user. A digital computer can accommodate many different input and output devices, such as CD-ROM and DVD drives, scanners, and printers. These devices use digital logic circuits, but often include analog electronic circuits, optical sensors, LCDs (liquid crystal displays), and electromechanical components.

The control unit in the CPU retrieves the instructions, one by one, from the program stored in the memory. For each instruction, the control unit manipulates the datapath to execute the operation specified by the instruction. Both program and data are stored in memory. A digital computer can perform arithmetic computations, manipulate strings of alphabetic characters, and be programmed to make decisions based on internal and external conditions.

Beyond the Computer

In terms of world impact, computers, such as the PC, are not the end of the story. Smaller, often less powerful, single-chip computers called *microcomputers* or *microcontrollers*, or special-purpose computers called *digital signal processors* (DSPs) actually are more prevalent in our lives. These computers are parts of everyday products and their presence is often not apparent. As a consequence of being integral parts of other products and often enclosed within them, they are called *embedded systems*. A generic block diagram of an embedded system is shown in Figure 1-3. Central to the system is the microcomputer (or its equivalent). It has many of the characteristics of the PC, but differs in the sense that its software programs are often permanently stored to provide only the functions required for the product. This software, which is critical to the operation of the product, is an integral part of the embedded system and referred to as *embedded software*. Also, the human interface of the microcomputer can be very limited or nonexistent. The larger information-storage components such as a hard drive and compact disk or DVD drive frequently are not present. The microcomputer contains some memory; if additional memory is needed, it can be added externally.

With the exception of the external memory, the hardware connected to the embedded microcomputer in Figure 1-3 interfaces with the product and/or the outside world. The input devices transform inputs from the product or outside world into electrical signals, and the output devices transform electrical signals into outputs to the product or outside world. The input and output devices are of two types, those which use analog signals and those which use digital signals. Examples of digital input devices include a limit switch which is closed or open depending on whether a force is applied to it and a keypad having ten decimal integer buttons. Examples of analog input devices include a thermistor which changes its electrical



□ **FIGURE 1-3**
Block Diagram of an Embedded System

resistance in response to the temperature and a crystal which produces a charge (and a corresponding voltage) in response to the pressure applied. Typically, electrical or electronic circuitry is required to "condition" the signal so that it can be read by the embedded system. Examples of digital output devices include relays (switches that are opened or closed by applied voltages), a stepper motor that responds to applied voltage pulses, or an LED digital display. Examples of analog output devices include a loudspeaker and a panel meter with a dial. The dial position is controlled by the interaction of the magnetic fields of a permanent magnet and an electromagnet driven by the voltage applied to the meter.

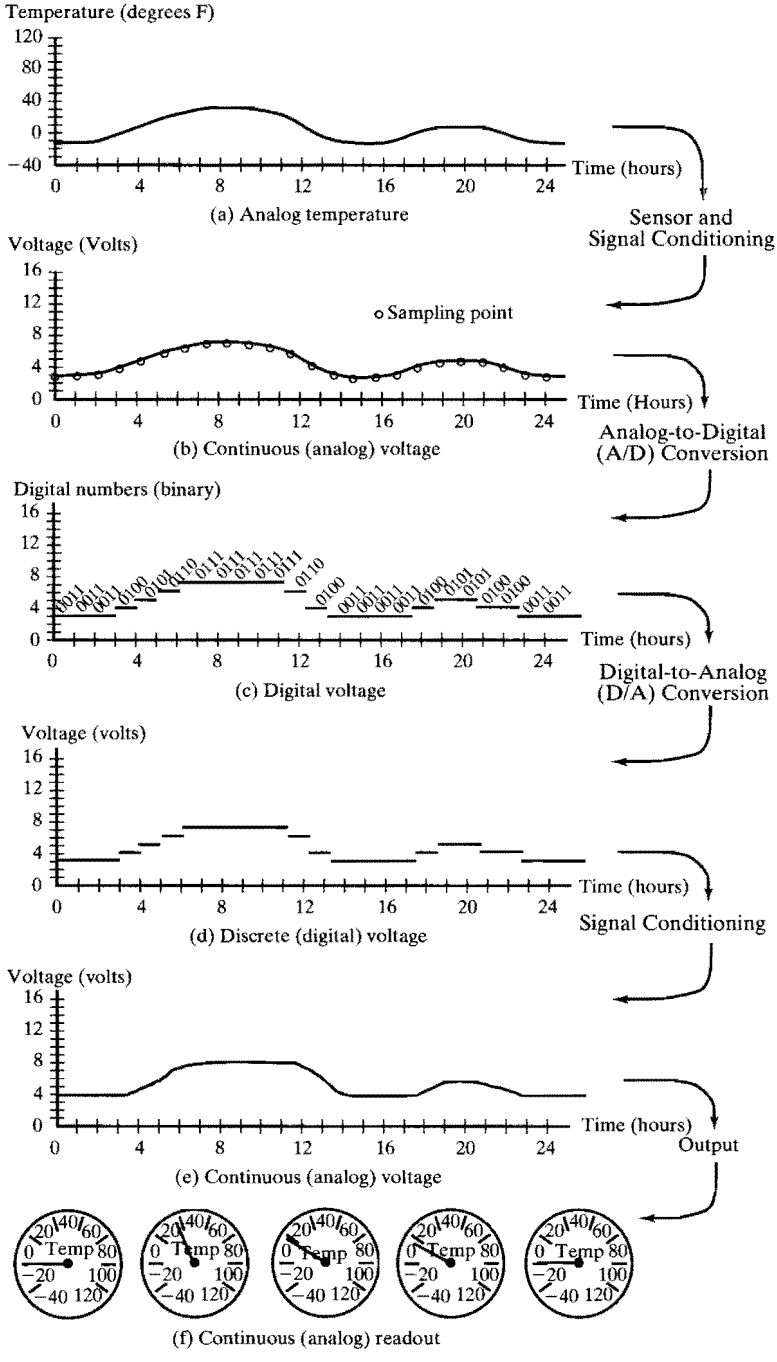
Next, we illustrate embedded systems by considering a temperature measurement performed by using a wireless weather station. In addition, this example also illustrates analog and digital signals, including conversion between the signal types.

EXAMPLE 1-1 Temperature Measurement and Display

A wireless weather station measures a number of weather parameters at an outdoor site and transmits them for display to an indoor base station. Its operation can be illustrated by considering the temperature measurement illustrated in Figure 1-4 with reference to the block diagram in Figure 1-3. Two embedded microprocessors are used, one in the outdoor site and the other in the indoor base station.

The temperature at the outdoor site ranges continuously from −40°F to +115°F. Temperature values over one 24-hour period are plotted as a function of time in Figure 1-4(a). This temperature is measured by a sensor consisting of a thermistor (a resistance that varies with temperature) with a fixed current applied by an electronic circuit. This sensor provides an analog voltage that is proportional to the temperature. Using signal conditioning, this voltage is changed to a continuous voltage ranging between 0 and 15 volts, as shown in Figure 1-4(b).

The analog voltage is sampled at a rate of once per hour (a very slow sampling rate used just for illustration), as shown by the dots in Figure 1-4(b). Each value sampled is applied to an analog-to-digital (A/D) converter, as in Figure 1-3, which replaces the value with a digital number written in binary and having decimal values between 0 and 15, as shown in Figure 1-4(c). A binary number can be interpreted in decimal by multiplying the bits from left to right times the respective weights, 8, 4, 2, and 1, and adding the resulting values. For example, 0101 can be interpreted as $0 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 = 5$. In the process of conversion, the value of the temperature is quantized from an infinite number of values to just 16 values. Examining the correspondence between the temperature in Figure 1-4(a) and the voltage in Figure 1-4(b), we find that the typical digital value of temperature represents an actual temperature range up to 5 degrees above or below the digital value. For example, the analog temperature range between −25 and −15 degrees is represented by the digital temperature value of −20 degrees. This discrepancy between the actual temperature and the digital temperature is called the *quantization error*. In order to obtain greater precision, we would need to increase the number of bits beyond four in the output of the A/D converter. The hardware



□ **FIGURE 1-4**
Temperature Measurement and Display

components for sensing, signal conditioning, and A/D conversion are shown in the upper left corner of Figure 1-3.

Next, the digital value passes through the microcomputer to a wireless transmitter as a digital output device in the lower right corner of Figure 1-3. The digital value is transmitted to a wireless receiver, which is a digital input device in the base station. The digital value enters the microcomputer at the base station, where calculations may be performed to adjust its value based on thermistor properties. The resulting value is to be displayed with an analog meter shown in Figure 1-4(f) as the output device. In order to support this display, the digital value is converted to an analog value by a digital-to-analog converter, giving the quantized, discrete voltage levels shown in Figure 1-4(d). Signal conditioning, such as processing of the output by a low-pass analog filter, is applied to give the continuous signal in Figure 1-4(e). This signal is applied to the analog voltage display, which has been labeled with the corresponding temperature values shown for five selected points over the 24-hour period in Figure 1-4(f). ■

□ **TABLE 1-1**
Embedded System Examples

Application Area	Product
Banking, commerce and manufacturing	Copiers, FAX machines, UPC scanners, vending machines, automatic teller machines, automated warehouses, industrial robots
Communication	Cell phones, routers, satellites
Games and toys	Video games, handheld games, talking stuffed toys
Home appliances	Digital alarm clocks, conventional and microwave ovens, dishwashers
Media	CD players, DVD players, flat panel TVs, Digital cameras, digital video cameras
Medical equipment	Pacemakers, incubators, magnetic resonance imaging
Personal	Digital watches, MP3 players, personal digital assistants
Transportation and navigation	Electronic engine controls, traffic light controllers, aircraft flight controls, global positioning systems

You might ask: "How many embedded systems are there in my current living environment?" Do you have a cell phone? An iPod™? An Xbox™? A digital camera? A microwave oven? An automobile? All of these are embedded systems! In fact, a late-model automobile can contain more than 50 microcontrollers, each controlling a distinct embedded system, such as the engine control unit (ECU), automatic braking system (ABS), and stability control unit (SCU). Further, a significant proportion of these embedded systems communicate with each other through a CAN (controller area network). A new automotive network called FlexRay promises to provide high-speed, reliable communication for safety-critical tasks such as braking-by-wire and steering-by-wire, eliminating primary dependence on mechanical and hydraulic linkages and enhancing the potential for additional safety features such as collision avoidance. Table 1-1 lists examples of embedded systems classified by application area.

Considering the widespread use of personal computers and embedded systems, the impact of digital systems on our lives is truly mind boggling! Digital systems play central roles in our medical diagnosis and treatment, our educational institutions and workplaces, in moving from place to place, in our homes, in interacting with others, and in just having fun! Considering the complexity of many of these systems, it is a wonder that they work at all. Thanks to the invention of the transistor and the integrated circuit and to the ingenuity and perseverance of millions of engineers and programmers, they indeed work and usually work well. In the remainder of this text, we take you on a journey that reveals how digital systems work and provide a detailed look at how to design digital systems and computers.

More on the Generic Computer

At this point, we will briefly discuss the generic computer and relate its various parts to the block diagram in Figure 1-2. At the lower left of the diagram at the beginning of this chapter is the heart of the computer, an integrated circuit called the *processor.* Modern processors such as this one are quite complex and consist of tens to hundreds of millions of transistors. The processor contains four functional modules: the CPU, the FPU, the MMU, and the internal cache.

We have already discussed the CPU. The FPU (*floating-point unit*) is somewhat like the CPU, except that its datapath and control unit are specifically designed to perform floating-point operations. In essence, these operations process information represented in the form of scientific notation (e.g., 1.234×10^7), permitting the generic computer to handle very large and very small numbers. The CPU and the FPU, in relation to Figure 1-2, each contain a datapath and a control unit.

The MMU is the *memory management unit.* The MMU plus the internal cache and the separate blocks near the bottom of the computer labeled "External Cache" and "RAM" (*random-access memory*) are all part of the memory in Figure 1-2. The two caches are special kinds of memory that allow the CPU and FPU to get at the data to be processed much faster than with RAM alone. RAM is what is most commonly referred to as memory. As its main function, the MMU causes the memory

that appears to be available to the CPU to be much, much larger than the actual size of the RAM. This is accomplished by data transfers between the RAM and the hard drive shown at the top of the drawing of the generic computer. So the hard drive, which we discuss later as an input/output device, conceptually appears as a part of both the memory and input/output.

The connection paths shown between the processor, memory, and external cache are the pathways between integrated circuits. These are typically implemented as fine copper conductors on a printed circuit board. The connection paths below the bus interface are referred to as the processor bus. The connections above the bus interface are the input/output (I/O) bus. The processor bus and the I/O bus attached to the bus interface carry data having different numbers of bits and have different ways of controlling the movement of data. They may also operate at different speeds. The bus interface hardware handles these differences so that data can be communicated between the two buses.

All of the remaining structures in the generic computer are considered part of I/O in Figure 1-2. In terms of sheer physical volume, these structures dominate. In order to enter information into the computer, a keyboard is provided. In order to view output in the form of text or graphics, a graphics adapter card and LCD (*liquid crystal display*) screen are provided. The hard drive, discussed previously, is an electromechanical magnetic storage device. It stores large quantities of information in the form of magnetic flux on spinning disks coated with magnetic materials. In order to control the hard drive and transfer information to and from it, a drive controller is used. The keyboard, graphics adapter card, and drive controller card are all attached to the I/O bus. This allows these devices to communicate through the bus interface with the CPU and other circuitry connected to the processor buses.

The generic computer consists mainly of an interconnection of digital modules. To understand the operation of each module, we need a basic knowledge of digital systems and their general behavior. Chapters 1 through 6 of this book deal with logic design of digital circuits in general. Chapters 5 and 7 discuss the primary components of a digital system, their operation, and their design. The operational characteristics of RAM are explained in Chapter 8. Datapath and control for simple computers are introduced in Chapter 9. Chapters 10 through 13 present the basics of computer design. Typical instructions employed in computer instruction-set architectures are presented in Chapter 10. The architecture and design of CPUs are examined in Chapter 11. Input and output devices and the various ways that a CPU can communicate with them are discussed in Chapter 12. Finally, memory hierarchy concepts related to the caches and MMU are introduced in Chapter 13.

To guide the reader through this material and to keep in mind the "forest" as we carefully examine many of the "trees," accompanying discussion appears in a blue box at the beginning of each chapter. Each discussion introduces the topics in the chapter and ties them to the associated components in the generic computer diagram at the start of this chapter. At the completion of our journey, we will have covered most of the various modules of the computer and will have gained an understanding of the fundamentals that underlie both its function and design.

Earlier, we mentioned that a digital computer manipulates discrete elements of information and that all information in the computer is represented in binary

form. Operands used for calculations may be expressed in the binary number system or in the decimal system by means of a binary code. The letters of the alphabet are also converted into a binary code. The remainder of this chapter introduces the binary number system, binary arithmetic, and selected binary codes as a basis for further study in the succeeding chapters. In relation to the generic computer, this material is very important and spans all of the components, excepting some in I/O that involve mechanical operations and analog (as contrasted with digital) electronics.

1-2 Number Systems

The decimal number system is employed in everyday arithmetic to represent numbers by strings of digits. Depending on its position in the string, each digit has an associated value of an integer raised to the power of 10. For example, the decimal number 724.5 is interpreted to represent 7 hundreds plus 2 tens plus 4 units plus 5 tenths. The hundreds, tens, units, and tenths are powers of 10 implied by the position of the digits. The value of the number is computed as follows:

$$724.5 = 7 \times 10^2 + 2 \times 10^1 + 4 \times 10^0 + 5 \times 10^{-1}$$

The convention is to write only the digits and infer the corresponding powers of 10 from their positions. In general, a decimal number with n digits to the left of the decimal point and m digits to the right of the decimal point is represented by a string of coefficients:

$$A_{n-1}A_{n-2}\cdots A_1A_0.A_{-1}A_{-2}\cdots A_{-m+1}A_{-m}$$

Each coefficient A_i is one of 10 digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). The subscript value i gives the position of the coefficient and, hence, the weight 10^i by which the coefficient must be multiplied.

The decimal number system is said to be of *base* or *radix* 10, because the coefficients are multiplied by powers of 10 and the system uses 10 distinct digits. In general, a number in base r contains r digits, 0, 1, 2, ..., $r - 1$, and is expressed as a power series in r with the general form

$$A_{n-1}r^{n-1} + A_{n-2}r^{n-2} + \ldots + A_1r^1 + A_0r^0 + A_{-1}r^{-1} + A_{-2}r^{-2} + \ldots + A_{-m+1}r^{-m+1} + A_{-m}r^{-m}$$

When the number is expressed in positional notation, only the coefficients and the radix point are written down:

$$A_{n-1}A_{n-2}\cdots A_1A_0.A_{-1}A_{-2}\cdots A_{-m+1}A_{-m}$$

In general, the "." is called the *radix point*. A_{n-1} is referred to as the *most significant digit* (*msd*) and A_{-m} as the *least significant digit* (*lsd*) of the number. Note that

if $m = 0$, the lsd is $A_{-0} = A_0$. To distinguish between numbers of different bases, it is customary to enclose the coefficients in parentheses and place a subscript after the right parenthesis to indicate the base of the number. However, when the context makes the base obvious, it is not necessary to use parentheses. The following illustrates a base 5 number with $n = 3$ and $m = 1$ and its conversion to decimal:

$$\begin{aligned}(312.4)_5 &= 3 \times 5^2 + 1 \times 5^1 + 2 \times 5^0 + 4 \times 5^{-1} \\ &= 75 + 5 + 2 + 0.8 = (82.8)_{10}\end{aligned}$$

Note that for all the numbers without the base designated, the arithmetic is performed with decimal numbers. Note also that the base 5 system uses only five digits, and, therefore, the values of the coefficients in a number can be only 0, 1, 2, 3, and 4 when expressed in that system.

An alternative method for conversion to base 10 that reduces the number of operations is based on a factored form of the power series:

$$(\ldots((A_{n-1}r + A_{n-2})r + A_{n-3})r + \cdots + A_1)r + A_0$$

$$+ (A_{-1} + (A_{-2} + (A_{-3} + \cdots + (A_{-m+2} + (A_{-m+1} + A_{-m}r^{-1})r^{-1})r^{-1}\ldots)r^{-1})r^{-1})r^{-1}$$

For the example above,

$$\begin{aligned}(312.4)_5 &= ((3 \times 5 + 1) \times 5) + 2 + 4 \times 5^{-1} \\ &= 16 \times 5 + 2 + 0.8 = (82.8)_{10}\end{aligned}$$

In addition to decimal, three number systems are used in computer work: binary, octal, and hexadecimal. These are base 2, base 8, and base 16 number systems, respectively.

Binary Numbers

The binary number system is a base 2 system with two digits: 0 and 1. A binary number such as 11010.11 is expressed with a string of 1s and 0s and, possibly, a binary point. The decimal equivalent of a binary number can be found by expanding the number into a power series with a base of 2. For example,

$$(11010)_2 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = (26)_{10}$$

As noted earlier, the digits in a binary number are called bits. When a bit is equal to 0, it does not contribute to the sum during the conversion. Therefore, the conversion to decimal can be obtained by adding the numbers with powers of two corresponding to the bits that are equal to 1. For example,

$$(110101.11)_2 = 32 + 16 + 4 + 1 + 0.5 + 0.25 = (53.75)_{10}$$

□ **TABLE 1-2**
Powers of Two

n	2^n	n	2^n	n	2^n
0	1	8	256	16	65,536
1	2	9	512	17	131,072
2	4	10	1,024	18	262,144
3	8	11	2,048	19	524,288
4	16	12	4,096	20	1,048,576
5	32	13	8,192	21	2,097,152
6	64	14	16,384	22	4,194,304
7	128	15	32,768	23	8,388,608

The first 24 numbers obtained from 2 to the power of n are listed in Table 1-2. In digital systems, we refer to 2^{10} as K (kilo), 2^{20} as M (mega), 2^{30} as G (giga), and 2^{40} as T (tera). Thus,

$$4\text{K} = 2^2 \times 2^{10} = 2^{12} = 4096 \quad \text{and} \quad 16\text{M} = 2^4 \times 2^{20} = 2^{24} = 16{,}777{,}216$$

This convention does not necessarily apply in all cases, with more conventional usage of K, M, G, and T as 10^3, 10^6, 10^9 and 10^{12}, respectively, sometimes applied as well. So caution is necessary in interpreting and using this notation.

The conversion of a decimal number to binary can be easily achieved by a method that successively subtracts powers of two from the decimal number. To convert the decimal number N to binary, first find the greatest number that is a power of two (see Table 1-2) and that, subtracted from N, produces a positive difference. Let the difference be designated N_1. Now find the greatest number that is a power of two and that, subtracted from N_1, produces a positive difference N_2. Continue this procedure until the difference is zero. In this way, the decimal number is converted to its powers-of-two components. The equivalent binary number is obtained from the coefficients of a power series that forms the sum of the components. 1s appear in the binary number in the positions for which terms appear in the power series, and 0s appear in all other positions. This method is demonstrated by the conversion of decimal 625 to binary as follows:

$$\begin{aligned}
625 - 512 &= 113 = N_1 & 512 &= 2^9 \\
113 - 64 &= 49 = N_2 & 64 &= 2^6 \\
49 - 32 &= 17 = N_3 & 32 &= 2^5 \\
17 - 16 &= 1 = N_4 & 16 &= 2^4 \\
1 - 1 &= 0 = N_5 & 1 &= 2^0
\end{aligned}$$

$$(625)_{10} = 2^9 + 2^6 + 2^5 + 2^4 + 2^0 = (1001110001)_2$$

Octal and Hexadecimal Numbers

As previously mentioned, all computers and digital systems use the binary representation. The octal (base 8) and hexadecimal (base 16) systems are useful for representing binary quantities indirectly because their bases are powers of two. Since $2^3 = 8$ and $2^4 = 16$, each octal digit corresponds to three binary digits and each hexadecimal digit corresponds to four binary digits.

The more compact representation of binary numbers in either octal or hexadecimal is much more convenient for people than using bit strings in binary that are three or four times as long. Thus, most computer manuals use either octal or hexadecimal numbers to specify binary quantities. A group of 15 bits, for example, can be represented in the octal system with only five digits. A group of 16 bits can be represented in hexadecimal with four digits. The choice between an octal and a hexadecimal representation of binary numbers is arbitrary, although hexadecimal tends to win out, since bits often appear in strings of size divisible by four.

The octal number system is the base 8 system with digits 0, 1, 2, 3, 4, 5, 6, and 7. An example of an octal number is 127.4. To determine its equivalent decimal value, we expand the number in a power series with a base of 8:

$$(127.4)_8 = 1 \times 8^2 + 2 \times 8^1 + 7 \times 8^0 + 4 \times 8^{-1} = (87.5)_{10}$$

Note that the digits 8 and 9 cannot appear in an octal number.

It is customary to use the first r digits from the decimal system, starting with 0, to represent the coefficients in a base r system when r is less than 10. The letters of the alphabet are used to supplement the digits when r is 10 or more. The hexadecimal number system is a base 16 system with the first 10 digits borrowed from the decimal system and the letters A, B, C, D, E, and F used for the values 10, 11, 12, 13, 14, and 15, respectively. An example of a hexadecimal number is

$$(B65F)_{16} = 11 \times 16^3 + 6 \times 16^2 + 5 \times 16^1 + 15 \times 16^0 = (46687)_{10}$$

The first 16 numbers in the decimal, binary, octal, and hexadecimal number systems are listed in Table 1-3. Note that the sequence of binary numbers follows a prescribed pattern. The least significant bit alternates between 0 and 1, the second significant bit between two 0s and two 1s, the third significant bit between four 0s and four 1s, and the most significant bit between eight 0s and eight 1s.

The conversion from binary to octal is easily accomplished by partitioning the binary number into groups of three bits each, starting from the binary point and proceeding to the left and to the right. The corresponding octal digit is then assigned to each group. The following example illustrates the procedure:

$$(010\ 110\ 001\ 101\ 011.\ 111\ 100\ 000\ 110)_2 = (26153.7406)_8$$

The corresponding octal digit for each group of three bits is obtained from the first eight entries in Table 1-3. To make the total count of bits a multiple of three, 0s can be added on the left of the string of bits to the left of the binary point. More importantly, 0s must

□ **TABLE 1-3**
Numbers with Different Bases

Decimal (base 10)	Binary (base 2)	Octal (base 8)	Hexadecimal (base 16)
00	0000	00	0
01	0001	01	1
02	0010	02	2
03	0011	03	3
04	0100	04	4
05	0101	05	5
06	0110	06	6
07	0111	07	7
08	1000	10	8
09	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

be added on the right of the string of bits to the right of the binary point to make the number of bits a multiple of three and obtain the correct octal result.

Conversion from binary to hexadecimal is similar, except that the binary number is divided into groups of four digits, starting at the binary point. The previous binary number is converted to hexadecimal as follows:

$$(0010\ 1100\ 0110\ 1011.\ 1111\ 0000\ 0110)_2 = (2C6B.F06)_{16}$$

The corresponding hexadecimal digit for each group of four bits is obtained by reference to Table 1-3.

Conversion from octal or hexadecimal to binary is done by reversing the procedure just performed. Each octal digit is converted to a 3-bit binary equivalent, and extra 0s are deleted. Similarly, each hexadecimal digit is converted to its 4-bit binary equivalent. This is illustrated in the following examples:

$$(673.12)_8 = 110\ 111\ 011.\ 001\ 010 = (110111011.00101)_2$$
$$(3A6.C)_{16} = 0011\ 1010\ 0110.\ 1100 = (1110100110.11)_2$$

Number Ranges

In digital computers, the range of numbers that can be represented is based on the number of bits available in the hardware structures that store and process information. The number of bits in these structures is most frequently a power of two, such as 8, 16, 32, and 64. Since the numbers of bits is fixed by the structures, the addition of leading or trailing zeros to represent numbers is necessary, and the range of numbers that can be represented is also fixed.

For example, for a computer processing 16-bit unsigned integers, the number 537 is represented as 0000001000011001. The range of integers that can be handled by this representation is from 0 to $2^{16} - 1$, that is, from 0 to 65,535. If the same computer is processing 16-bit unsigned fractions with the binary point to the left of the most significant digit, then the number 0.375 is represented by 0.0110000000000000. The range of fractions that can be represented is from 0 to $(2^{16} - 1)/2^{16}$, or from 0.0 to 0.9999847412.

In later chapters, we will deal with fixed-bit representations and ranges for binary signed numbers and floating-point numbers. In both of these cases, some bits are used to represent information other than simple integer or fraction values.

1-3 ARITHMETIC OPERATIONS

Arithmetic operations with numbers in base r follow the same rules as for decimal numbers. However, when a base other than the familiar base 10 is used, one must be careful to use only r allowable digits and perform all computations with base r digits. Examples of the addition of two binary numbers are as follows (note the names of the operands for addition):

Carries:	00000	101100
Augend:	01100	10110
Addend:	+10001	+10111
Sum:	11101	101101

The sum of two binary numbers is calculated following the same rules as for decimal numbers, except that the sum digit in any position can be only 1 or 0. Also, a carry in binary occurs if the sum in any bit position is greater than 1. (A carry in decimal occurs if the sum in any digit position is greater than 9.) Any carry obtained in a given position is added to the bits in the column one significant position higher. In the first example, since all of the carries are 0, the sum bits are simply the sum of the augend and addend bits. In the second example, the sum of the bits in the second column from the right is 2, giving a sum bit of 0 and a carry bit of 1 (2 = 2 + 0). The carry bit is added with the 1s in the third position, giving a sum of 3, which produces a sum bit of 1 and a carry of 1 (3 = 2 + 1).

The following are examples of the subtraction of two binary numbers; as with addition, note the names of the operands:

Borrows:	00000	00110		00110
Minuend:	10110	10110	10011	11110
Subtrahend:	−10010	−10011	−11110	−10011
Difference:	00100	00011		−01011

The rules for subtraction are the same as in decimal, except that a borrow into a given column adds 2 to the minuend bit. (A borrow in the decimal system adds 10 to the minuend digit.) In the first example shown, no borrows occur, so the difference bits are simply the minuend bits minus the subtrahend bits. In the second example, in the right position, the subtrahend bit is 1 with the minuend bit 0, so it is necessary to borrow from the second position as shown. This gives a difference bit in the first position of 1 (2 + 0 − 1 = 1). In the second position, the borrow is subtracted, so a borrow is again necessary. Recall that, in the event that the subtrahend is larger than the minuend, we subtract the minuend from the subtrahend and give the result a minus sign. This is the case in the third example, in which this interchange of the two operands is shown.

The final operation to be illustrated is binary multiplication, which is quite simple. The multiplier digits are always 1 or 0. Therefore, the partial products are equal either to the multiplicand or to 0. Multiplication is illustrated by the following example:

Multiplicand:	1011
Multiplier:	× 101
	1011
	0000
	1011
Product:	110111

Arithmetic operations with octal, hexadecimal, or any other base r system will normally require the formulation of tables from which one obtains sums and products of two digits in that base. An easier alternative for adding two numbers in base r is to convert each pair of digits in a column to decimal, add the digits in decimal, and then convert the result to the corresponding sum and carry in the base r system. Since addition is done in decimal, we can rely on our memories for obtaining the entries from the familiar decimal addition table. The sequence of steps for adding the two hexadecimal numbers 59F and E46 is shown in Example 1-2.

EXAMPLE 1-2 Hexadecimal Addition

Perform the addition $(59F)_{16} + (E46)_{16}$:

Hexadecimal	Equivalent Decimal Calculation				
	1 ←		1 ←		
5 9 F	5	Carry	9	15	Carry
E 4 6	14		4	6	
1 3 E 5	1 19 = 16 + 3		14 = E	21 = 16 + 5	

The equivalent decimal calculation columns on the right show the mental reasoning that must be carried out to produce each digit of the hexadecimal sum. Instead of adding F + 6 in hexadecimal, we add the equivalent decimals, 15 + 6 = 21. We then convert back to hexadecimal by noting that 21 = 16 + 5. This gives a sum digit of 5 and a carry of 1 to the next higher-order column of digits. The other two columns are added in a similar fashion. ■

In general, the multiplication of two base r numbers can be accomplished by doing all the arithmetic operations in decimal and converting intermediate results one at a time. This is illustrated in the multiplication of two octal numbers shown in Example 1-3.

EXAMPLE 1-3 Octal Multiplication

Perform the multiplication $(762)_8 \times (45)_8$:

Octal	Octal		Decimal		Octal
762	5×2	=	$10 = 8 + 2$	=	12
45	$5 \times 6 + 1$	=	$31 = 24 + 7$	=	37
4672	$5 \times 7 + 3$	=	$38 = 32 + 6$	=	46
3710	4×2	=	$8 = 8 + 0$	=	10
43772	$4 \times 6 + 1$	=	$25 = 24 + 1$	=	31
	$4 \times 7 + 3$	=	$31 = 24 + 7$	=	37

Shown on the right are the mental calculations for each pair of octal digits. The octal digits 0 through 7 have the same value as their corresponding decimal digits. The multiplication of two octal digits plus a carry, derived from the calculation on the previous line, is done in decimal, and the result is then converted back to octal. The left digit of the two-digit octal result gives the carry that must be added to the digit product on the next line. The blue digits from the octal results of the decimal calculations are copied to the octal partial products on the left. For example, $(5 \times 2)_8 = (12)_8$. The left digit, 1, is the carry to be added to the product $(5 \times 6)_8$, and the blue least significant digit, 2, is the corresponding digit of the octal partial product. When there is no digit product to which the carry can be added, the carry is written directly into the octal partial product, as in the case of the 4 in 46. ■

Conversion from Decimal to Other Bases

We convert a number in base r to decimal by expanding it in a power series and adding all the terms, as shown previously. We now present a general procedure for the operation of converting a decimal number to a number in base r that is the reverse of the alternative expansion to base 10 on page 32. If the number includes a radix point, we need to separate the number into an integer part and a fraction part, since the two parts must be converted differently. The conversion of a decimal integer to a number in base r is done by dividing the number and all successive

quotients by r and accumulating the remainders. This procedure is best explained by example.

EXAMPLE 1-4 Conversion of Decimal Integers to Octal

Convert decimal 153 to octal:

The conversion is to base 8. First, 153 is divided by 8 to give a quotient of 19 and a remainder of 1, as shown in blue. Then 19 is divided by 8 to give a quotient of 2 and a remainder of 3. Finally, 2 is divided by 8 to give a quotient of 0 and a remainder of 2. The coefficients of the desired octal number are obtained from the remainders:

$153/8 = 19 + 1/8$	Remainder $= 1$	Least significant digit
$19/8 = 2 + 3/8$	$= 3$	
$2/8 = 0 + 2/8$	$= 2$	Most significant digit

$(153)_{10} = (231)_8$ ■

Note in Example 1-4 that the remainders are read from last to first, as indicated by the arrow, to obtain the converted number. The quotients are divided by r until the result is 0. We also can use this procedure to convert decimal integers to binary, as shown in Example 1-5. In this case, the base of the converted number is 2, and therefore, all the divisions must be done by 2.

EXAMPLE 1-5 Conversion of Decimal Integers to Binary

Convert decimal 41 to binary:

$41/2 = 20 + 1/2$	Remainder $= 1$	Least significant digit
$20/2 = 10$	$= 0$	
$10/2 = 5$	$= 0$	
$5/2 = 2 + 1/2$	$= 1$	
$2/2 = 1$	$= 0$	
$1/2 = 0 + 1/2$	$= 1$	Most significant digit

$(41)_{10} = (101001)_2$

Of course, the decimal number could be converted by the sum of powers of two:

$$(41)_{10} = 32 + 8 + 1 = (101001)_2$$ ■

The conversion of a decimal fraction to base r is accomplished by a method similar to that used for integers, except that multiplication by r is used instead of

division, and integers are accumulated instead of remainders. Again, the method is best explained by example.

EXAMPLE 1-6 Conversion of Decimal Fractions to Binary

Convert decimal 0.6875 to binary:

First, 0.6875 is multiplied by 2 to give an integer and a fraction. The new fraction is multiplied by 2 to give a new integer and a new fraction. This process is continued until the fractional part equals 0 or until there are enough digits to give sufficient accuracy. The coefficients of the binary number are obtained from the integers as follows:

$$
\begin{aligned}
0.6875 \times 2 &= 1.3750 \quad \text{Integer} = 1 \quad \text{Most significant digit} \\
0.3750 \times 2 &= 0.7500 \quad = 0 \\
0.7500 \times 2 &= 1.5000 \quad = 1 \\
0.5000 \times 2 &= 1.0000 \quad = 1 \quad \text{Least significant digit} \\
(0.6875)_{10} &= (0.1011)_2
\end{aligned}
$$

■

Note in the foregoing example that the integers are read from first to last, as indicated by the arrow, to obtain the converted number. In the example, a finite number of digits appear in the converted number. The process of multiplying fractions by r does not necessarily end with zero, so we must decide how many digits of the fraction to use from the conversion. Also, remember that the multiplications are by number r. Therefore, to convert a decimal fraction to octal, we must multiply the fractions by 8, as shown in Example 1-7.

EXAMPLE 1-7 Conversion of Decimal Fractions to Octal

Convert decimal 0.513 to a three-digit octal fraction:

$$
\begin{aligned}
0.513 \times 8 &= 4.104 \quad \text{Integer} = 4 \quad \text{Most significant digit} \\
0.104 \times 8 &= 0.832 \quad = 0 \\
0.832 \times 8 &= 6.656 \quad = 6 \\
0.656 \times 8 &= 5.248 \quad = 5 \quad \text{Least significant digit}
\end{aligned}
$$

The answer, to three significant figures, is obtained from the integer digits. Note that the last integer digit, 5, is used for rounding in base 8 of the second-to-the-last digit, 6, to obtain

$$(0.513)_{10} = (0.407)_8$$

■

The conversion of decimal numbers with both integer and fractional parts is done by converting each part separately and then combining the two answers. Using the results of Example 1-4 and Example 1-7, we obtain

$$(153.513)_{10} = (231.407)_8$$

1-4 Decimal Codes

The binary number system is the most natural one for a computer, but people are accustomed to the decimal system. One way to resolve this difference is to convert decimal numbers to binary, perform all arithmetic calculations in binary, and then convert the binary results back to decimal. This method requires that we store the decimal numbers in the computer in such a way that they can be converted to binary. Since the computer can accept only binary values, we must represent the decimal digits by a code that contains 1s and 0s. It is also possible to perform the arithmetic operations directly with decimal numbers when they are stored in the computer in coded form.

An n-bit *binary code* is a group of n bits that assume up to 2^n distinct combinations of 1s and 0s, with each combination representing one element of the set being coded. A set of four elements can be coded with a 2-bit binary code, with each element assigned one of the following bit combinations: 00, 01, 10, 11. A set of 8 elements requires a 3-bit code, and a set of 16 elements requires a 4-bit code. The bit combinations of an n-bit code can be determined from the count in binary from 0 to $2^n - 1$. Each element must be assigned a unique binary bit combination, and no two elements can have the same value; otherwise, the code assignment is ambiguous.

A binary code will have some unassigned bit combinations if the number of elements in the set is not a power of 2. The ten decimal digits form such a set. A binary code that distinguishes among ten elements must contain at least four bits, but six out of the 16 possible combinations will remain unassigned. Numerous different binary codes can be obtained by arranging four bits into 10 distinct combinations. The code most commonly used for the decimal digits is the straightforward binary assignment listed in Table 1-3 on page 32. This is called *binary-coded decimal* and is commonly referred to as BCD. Other decimal codes are possible, one of which is presented in Chapter 3.

Table 1-4 gives a 4-bit code for each decimal digit. A number with n decimal digits will require $4n$ bits in BCD. Thus, decimal 396 is represented in BCD with 12 bits as

0011 1001 0110

with each group of four bits representing one decimal digit. A decimal number in BCD is the same as its equivalent binary number only when the number is between 0 and 9, inclusive. A BCD number greater than 10 has a representation different from its equivalent binary number, even though both contain 1s and 0s. Moreover, the binary combinations 1010 through 1111 are not used and have no meaning in the BCD code.

Consider decimal 185 and its corresponding value in BCD and binary:

$$(185)_{10} = (0001\ 1000\ 0101)_{BCD} = (10111001)_2$$

□ **TABLE 1-4**
Binary-Coded Decimal (BCD)

Decimal Symbol	BCD Digit
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

The BCD value has 12 bits, but the equivalent binary number needs only 8 bits. It is obvious that a BCD number needs more bits than its equivalent binary value. However, BCD representation of decimal numbers is still important, because computer input and output data used by most people needs to be in the decimal system. BCD numbers are decimal numbers and not binary numbers, even though they are represented using bits. The only difference between a decimal and a BCD number is that decimals are written with the symbols 0, 1, 2, ..., 9, and BCD numbers use the binary codes 0000, 0001, 0010, ..., 1001.

BCD Addition

Consider the addition of two decimal digits in BCD, together with a possible carry of 1 from a previous less significant pair of digits. Since each digit does not exceed 9, the sum cannot be greater than $9 + 9 + 1 = 19$, the 1 being a carry. Suppose we add the BCD digits as if they were binary numbers. Then the binary sum will produce a result in the range from 0 to 19. In binary, this will be from 0000 to 10011, but in BCD, it should be from 0000 to 1 1001, the first 1 being a carry and the next four bits being the BCD digit sum. When the binary sum is less than 1010 (without a carry), the corresponding BCD digit is correct. But when the binary sum is greater than or equal to 1010, the result is an invalid BCD digit. The addition of binary 6, $(0110)_2$, to the sum converts it to the correct digit and also produces a decimal carry as required. The reason is that the difference between a carry from the most significant bit position of the binary sum and a decimal carry is $16 - 10 = 6$. Thus, the decimal carry and the correct BCD sum

digit are forced by adding 6 in binary. Consider the next three-digit BCD addition example.

EXAMPLE 1-8 BCD Addition

110	BCD carry	1	1	
448		0100	0100	1000
+489		+0100	+1000	+1001
937	Binary sum	1001	1101	1 0001
	Add 6		+0110	+0110
	BCD sum		1 0011	1 0111
	BCD result	1001	0011	0111

In each position, the two BCD digits are added as if they were two binary numbers. If the binary sum is greater than 1001, we add 0110 to obtain the correct BCD digit sum and a carry. In the right column, the binary sum is equal to 17. The presence of the carry indicates that the sum is greater than 16 (certainly greater than 9), so a correction is needed. The addition of 0110 produces the correct BCD digit sum, 0111 (7), and a carry of 1. In the next column, the binary sum is 1101 (13), an invalid BCD digit. Addition of 0110 produces the correct BCD digit sum, 0011 (3), and a carry of 1. In the final column, the binary sum is equal to 1001 (9) and is the correct BCD digit. ■

1-5 Alphanumeric Codes

Many applications of digital computers require the handling of data consisting not only of numbers, but also of letters. For instance, an insurance company with thousands of policyholders uses a computer to process its files. To represent the names and other pertinent information, it is necessary to formulate a binary code for the letters of the alphabet. In addition, the same binary code must represent numerals and special characters such as $. Any alphanumeric character set for English is a set of elements that includes the ten decimal digits, the 26 letters of the alphabet, and several (more than three) special characters. If only capital letters are included, we need a binary code of at least six bits, and if both uppercase letters and lowercase letters are included, we need a binary code of at least seven bits. Binary codes play an important role in digital computers. The codes must be in binary because computers can handle only 1s and 0s. Note that binary encoding merely changes the symbols, not the meaning of the elements of information being encoded.

ASCII Character Code

The standard binary code for the alphanumeric characters is called ASCII (American Standard Code for Information Interchange). It uses seven bits to code 128 characters, as shown in Table 1-5. The seven bits of the code are designated by B_1 through B_7, with B_7 being the most significant bit. Note that the most significant three bits of the code determine the column of the table and the least significant four bits the row of the table. The letter A, for example, is represented in ASCII as 1000001 (column 100, row 0001). The ASCII code contains 94 characters that can be printed and 34 nonprinting characters used for various control functions. The printing characters consist of the 26 uppercase letters, the 26 lowercase letters, the 10 numerals, and 32 special printable characters such as %, @, and $.

The 34 control characters are designated in the ASCII table with abbreviated names. They are listed again below the table with their full functional names. The control characters are used for routing data and arranging the printed text into a prescribed format. There are three types of control characters: format effectors, information separators, and communication control characters. Format effectors are characters that control the layout of printing. They include the familiar typewriter controls such as backspace (BS), horizontal tabulation (HT), and carriage return (CR). Information separators are used to separate the data into divisions—for example, paragraphs and pages. They include characters such as record separator (RS) and file separator (FS). The communication control characters are used during the transmission of text from one location to the other. Examples of communication control characters are STX (start of text) and ETX (end of text), which are used to frame a text message transmitted via communication wires.

ASCII is a 7-bit code, but most computers manipulate an 8-bit quantity as a single unit called a *byte*. Therefore, ASCII characters most often are stored one per byte, with the most significant bit set to 0. The extra bit is sometimes used for specific purposes, depending on the application. For example, some printers recognize an additional 128 8-bit characters, with the most significant bit set to 1. These characters enable the printer to produce additional symbols, such as those from the Greek alphabet or characters with accent marks as used in languages other than English.

UNICODE This supplement on Unicode, a 16-bit standard code for representing the symbols and ideographs for the world's languages, is available on the Companion Website (http://www.prenhall.com/mano) for the text.

Parity Bit

To detect errors in data communication and processing, an additional bit is sometimes added to a binary code word to define its parity. A *parity bit* is the extra bit

☐ TABLE 1-5
American Standard Code for Information Interchange (ASCII)

	$B_7B_6B_5$							
$B_4B_3B_2B_1$	000	001	010	011	100	101	110	111
0000	NULL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Control Characters

NULL	NULL	DLE	Data link escape
SOH	Start of heading	DC1	Device control 1
STX	Start of text	DC2	Device control 2
ETX	End of text	DC3	Device control 3
EOT	End of transmission	DC4	Device control 4
ENQ	Enquiry	NAK	Negative acknowledge
ACK	Acknowledge	SYN	Synchronous idle
BEL	Bell	ETB	End of transmission block
BS	Backspace	CAN	Cancel
HT	Horizontal tab	EM	End of medium
LF	Line feed	SUB	Substitute
VT	Vertical tab	ESC	Escape
FF	Form feed	FS	File separator
CR	Carriage return	GS	Group separator
SO	Shift out	RS	Record separator
SI	Shift in	US	Unit separator
SP	Space	DEL	Delete

included to make the total number of 1s in the resulting code word either even or odd. Consider the following two characters and their even and odd parity:

	With Even Parity	With Odd Parity
1000001	01000001	11000001
1010100	11010100	01010100

In each case, we use the extra bit in the most significant position of the code to produce an even number of 1s in the code for *even parity* or an odd number of 1s in the code for *odd parity*. In general, one parity or the other is adopted, with even parity being more common. Parity may be used with binary numbers as well as with codes, including ASCII for characters, and the parity bit may be placed in any fixed position in the code.

EXAMPLE 1-9 Error Detection and Correction for ASCII Transmission

The parity bit is helpful in detecting errors during the transmission of information from one location to another. Assuming that even parity is used, the simplest case is handled as follows: An even (or odd) parity bit is generated at the sending end for all 7-bit ASCII characters; the 8-bit characters that include parity bits are transmitted to their destination. The parity of each character is then checked at the receiving end; if the parity of the received character is not even (odd), it means that at least one bit has changed its value during the transmission. This method detects one, three, or any odd number of errors in each character transmitted. An even number of errors is undetected. Other error-detection codes, some of which are based on additional parity bits, may be needed to take care of an even number of errors. What is done after an error is detected depends on the particular application. One possibility is to request retransmission of the message on the assumption that the error was random and will not occur again. Thus, if the receiver detects a parity error, it sends back a NAK (negative acknowledge) control character consisting of the even-parity eight bits, 10010101, from Table 1-5 on page 45. If no error is detected, the receiver sends back an ACK (acknowledge) control character, 00000110. The sending end will respond to a NAK by transmitting the message again, until the correct parity is received. If, after a number of attempts, the transmission is still in error, an indication of a malfunction in the transmission path is given. ■

1-6 Gray Codes

As we count up or down using binary codes, the number of bits that change from one binary value to the next varies. This is illustrated by the binary code for the octal digits on the left in Table 1-6. As we count from 000 up to 111 and "roll

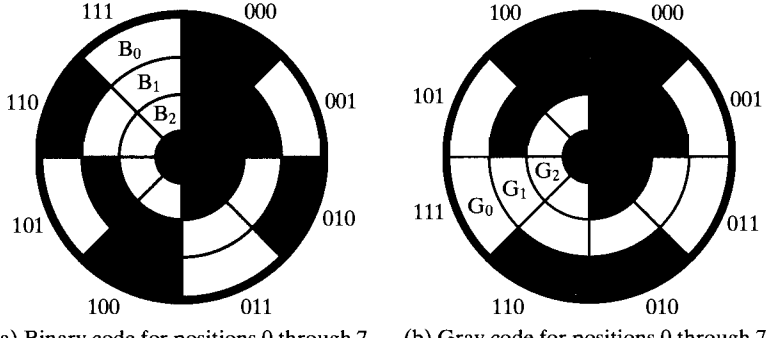
□ **TABLE 1-6**
Gray Code

Binary Code	Bit Changes	Gray Code	Bit Changes
000		000	
	1		1
001		001	
	2		1
010		011	
	1		1
011		010	
	3		1
100		110	
	1		1
101		111	
	2		1
110		101	
	1		1
111		100	
	3		1
000		000	

over" to 000, the number of bits that change between the binary values ranges from 1 to 3.

For many applications, multiple bit changes as the circuit counts is not a problem. There are applications, however, in which a change of more than one bit when counting up or down can cause serious problems. One such problem is illustrated by an optical shaft-angle encoder shown in Figure 1-5(a). The encoder is a disk attached to a rotating shaft for measurement of the rotational position of the shaft. The disk contains areas that are clear for binary 1 and opaque for binary 0. An illumination source is placed on one side of the disk, and optical sensors, one for each of the bits to be encoded, are placed on the other side of the disk. When a clear region lies between the source and a sensor, the sensor responds to the light with a binary 1 output. When an opaque region lies between the source and the sensor, the sensor responds to the dark with a binary 0.

The rotating shaft, however, can be in any angular position. For example, suppose that the shaft and disk are positioned so that the sensors lie right at the



(a) Binary code for positions 0 through 7 (b) Gray code for positions 0 through 7

□ **FIGURE 1-5**
Optical Shaft-Angle Encoder

boundary between 011 and 100. In this case, sensors in positions B_2, B_1 and B_0 have the light partially blocked. In such a situation, it is unclear whether the three sensors will see light or dark. As a consequence, each sensor may produce either a 1 or a 0. Thus, the resulting encoded binary number for a value between 3 and 4 may be 000, 001, 010, 011, 100, 101, 110, or 111. Either 011 or 100 will be satisfactory in this case, but the other six values are clearly erroneous!

To see the solution to this problem, notice that in those cases in which only a single bit changes when going from one value to the next or previous value, this problem cannot occur. For example, if the sensors lie on the boundary between 2 and 3, the resulting code is either 010 or 011, either of which is satisfactory. If we change the encoding of the values 0 through 7 such that only one bit value changes as we count up or down (including rollover from 7 to 0), then the encoding will be satisfactory for all positions. A code having the property that only one bit at a time changes between codes during counting is a *Gray code* named for Frank Gray, who patented its use for shaft encoders in 1953. There are multiple Gray codes for any set of n consecutive integers, with n even.

A specific Gray code for the octal digits, called a *binary reflected Gray code*, appears on the right in Table 1-6. Note that the counting order for binary codes is now 000, 001, 011, 010, 110, 111, 101, 100, and 000. If we want binary codes for processing, then we can build a digital circuit or use software that converts these codes to binary before they are used in further processing of the information.

Figure 1-5(b) shows the optical shaft-angle encoder using the Gray code from Table 1-6. Note that any two segments on the disk adjacent to each other have only one region that is clear for one and opaque for the other.

The optical shaft encoder illustrates one use of the Gray code concept. There are many other similar uses in which a physical variable, such as position or voltage, has a continuous range of values that is converted to a digital representation. A quite different use of Gray codes appears in low-power CMOS (Complementary Metal Oxide Semiconductor) logic circuits that count up or down. In CMOS, power is consumed only when a bit changes. For the example codes given in Table 1-6 with continuous counting (either up or down), there are 14 bit changes for binary counting for every eight bit changes for Gray code counting. Thus, the power consumed at the counter outputs for the Gray code counter is only 57 percent of that consumed at the binary counter outputs.

A Gray code for a counting sequence of n binary code words (n must be even) can be constructed by replacing each of the first $n/2$ numbers in the sequence with a code word consisting of 0 followed by the even parity for each bit of the binary code word and the bit to its left. For example, for the binary code word 0100, the Gray code word is 0, parity(0, 1), parity(1, 0), parity(0, 0) = 0110. Next, take the sequence of numbers formed and copy it in reverse order with the leftmost 0 replaced by a 1. This new sequence provides the Gray code words for the second $n/2$ of the original n code words. For example, for BCD codes, the first five Gray code words are 0000, 0001, 0011, 0010, and 0110. Reversing the order of these codes and replacing the leftmost 0 with a 1, we obtain 1110, 1010, 1011, 1001, and 1000 for the last five Gray codes. For the special cases in which the original binary codes are 0 through $2^n - 1$, each Gray code word may be formed directly from the

corresponding binary code word by copying its leftmost bit and then replacing each of the remaining bits with the even parity of the bit of the number and the bit to its left.

1-7 Chapter Summary

In this chapter, we introduced digital systems and digital computers and showed why such systems use signals having only two values. We briefly introduced the structure of the stored-program digital computer and showed how computers can be applied to a broad range of specialized applications by using embedded systems. We then related the computer structure to a representative example of a personal computer (PC).

Number-system concepts, including base (radix) and radix point, were presented. Because of their correspondence to two-valued signals, binary numbers were discussed in detail. Octal (base 8) and hexadecimal (base 16) were also emphasized, since they are useful as shorthand notation for binary. Arithmetic operations in bases other than base 10 and the conversion of numbers from one base to another were covered. Because of the predominance of decimal in normal use, Binary-Coded Decimal (BCD) was treated. The representation of information in the form of characters instead of numbers by means of the ASCII code for the English alphabet was presented. The parity bit was presented as a technique for error detection, and the Gray code, which is critical to selected applications, was defined.

In subsequent chapters, we treat the representation of signed numbers and floating-point numbers. Although these topics fit well with the topics in this chapter, they are difficult to motivate without associating them with the hardware used to implement the operations performed on them. Thus, we delay their presentation until we examine the associated hardware.

References

1. Gray, F. *Pulse Code Communication.* U.S. Patent 2 632 058, March 17, 1953.
2. Patterson, D. A., and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface,* 3rd ed. San Francisco: Morgan Kaufmann, 2004.
3. White, R. *How Computers Work: Millennium Edition,* 5th ed. Indianapolis: Que, 1999.

Problems

The plus (+) indicates a more advanced problem.

1–1. This problem concerns wind measurements made by the wireless weather station illustrated in Example 1-1. The wind-speed measurement uses a