Chapter

4

The Building Blocks: Binary Numbers, Boolean Logic, and Gates

Chapter Introduction

r studying this chapter, you will be able to:

Translate between base-ten and base-two numbers, and represent negative numbers using both sign-magnitude and two's complement representations
Explain how fractional numbers, characters, sounds, and images are represented inside the computer
Build truth tables for Boolean expressions and determine when they are true or false
Describe the relationship between Boolean logic and electronic gates
Construct circuits using the sum-of-products circuit design algorithm, and analyze simple circuits to determine their truth tables
Explain how large, complex circuits, like 32-bit adder or compare-for-equality circuits, are constructed from simpler, 1-bit components

Describe the purpose and workings of multiplexer and decoder control circuits

4.1 Introduction

Level 1 of this text investigated the algorithmic foundations of computer science. It developed algorithms for searching lists, finding largest and smallest values, locating patterns, sorting lists, and cleaning up bad data. It also showed how to analyze and evaluate algorithms to demonstrate that they are not only correct but efficient and useful as well.

Our discussion assumed that these algorithms would be executed by something called a *computing agent*, an abstract concept representing any entity capable of understanding and executing our instructions. At the time we didn't care what that computing agent was—person, mathematical model, computer, or robot. However, in this section of the text we *do*care what our computing agent looks like and how it is able to execute instructions and produce results.

In this chapter, we introduce the fundamental building blocks of all computer systems—binary representation, Boolean logic, gates, and circuits.

4.2 The Binary Numbering System

Our first concern with learning how to build computers is understanding how computers represent information. Their internal storage techniques are quite different from the way you and I represent information when we write a note or do a quick calculation on paper.

4.2.1 Binary Representation of Numeric and Textual Information

People generally represent numeric and textual information (language differences aside) by using the following notational conventions:

1. The 10 decimal digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 for numeric values such as 459
2. *Sign/magnitude notation* for signed numbers—that is, a + or − sign placed immediately to the left of the digits; +31 and −789 are examples
3. *Decimal notation* for real numbers, with a decimal point separating the whole number part from the fractional part; an example is 12.34
4. The 26 letters A, B, C, …, Z for textual information (as well as lowercase letters and a few special symbols for punctuation)

You might suppose that these well-known schemes are the same conventions that computers use to store information in memory. Surprisingly, this is not true.

There are two types of information representation: The *external* representation of information is the way information is represented by humans and the way it is entered at a keyboard or virtual keypad or displayed on a printer or screen.
The *internal* representation of information is the way it is stored in the memory of a computer. This difference is diagrammed in Figure 4.1.

Figure 4.1 Distinction between external and internal representation of information

Externally, computers do use decimal digits, sign/magnitude notation, and the 26-character alphabet. However, virtually every computer ever built stores data—numbers, letters, graphics, images, sound—internally using the **binary numbering system**.

Binary is a base-2 **positional numbering system** not unlike the more familiar decimal, or base-10, system used in everyday life. In these systems, the value or "worth" of a digit depends not only on its absolute value but also on its specific position within a number. In the decimal system, there are 10 unique digits (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9), and the value of the positions in a decimal number is based on powers of 10. Moving from right to left in a number, the positions represent ones , tens , hundreds , thousands , and so on. Therefore, the decimal number 2,359 is evaluated as follows:
The same concepts apply to binary numbers except that there are only two digits, 0 and 1, and the value of the positions in a binary number is based on powers of 2. Moving from right to left, the positions represent ones , twos , fours , eights , sixteens , and so on. The two digits, 0 and 1, are frequently referred to as **bits**, a contraction of the two words *binary* digits.
For example, the six-digit binary number 111001 is evaluated as follows:

The five-digit binary quantity 10111 is evaluated in the following manner:

Evaluating a binary number is quite easy, because 1 times any value is simply that value, and 0 times any value is always 0. Thus, when evaluating a binary number, use the following *binary-to-decimal algorithm*: Whenever there is a 1 in a column, add the positional value of that column to a running sum, and whenever there is a 0 in a column, add nothing. The final sum is the decimal value of this binary number. This is the procedure we followed in the previous two examples.

A binary-to-decimal conversion table for the values 0–31 is shown in Figure 4.2. You might want to evaluate a few of the binary values using this algorithm to confirm their decimal equivalents.

Figure 4.2

Binary-to-decimal conversion table for the decimal digits 0–31

Binary	Decimal	Binary	Decimal
0	0	10000	16

Binary	Decimal	Binary	Decimal
1	1	10001	17
10	2	10010	18
11	3	10011	19
100	4	10100	20
101	5	10101	21
110	6	10110	22
111	7	10111	23
1000	8	11000	24
1001	9	11001	25
1010	10	11010	26
1011	11	11011	27
1100	12	11100	28
1101	13	11101	29
1110	14	11110	30
1111	15	11111	31

Any whole number that can be represented in base 10 can also be represented in base 2, although it may take more digits because a single decimal digit contains more information than a single binary digit. Note that in the first example shown on the previous page it takes only two decimal digits (5 and 7) to represent the quantity 57 in base 10, but it takes six binary digits (1, 1, 1, 0, 0, and 1) to express the same value in base 2.

To go in the reverse direction—that is, to convert a decimal value into its binary equivalent—we use the *decimal-to-binary algorithm*, which is based on successive divisions by 2. Dividing the original decimal value by 2 produces a quotient and a remainder, which must be either a 0 or a 1. Record the remainder digit and then divide the quotient by 2, getting a new quotient and a second remainder digit. The process of dividing by 2, saving the quotient, and writing down the remainder is repeated until the quotient equals 0. The sequence of remainder digits, when written left to right from the last remainder digit to the first, is the binary representation of the original decimal value. For example, here is the conversion of the decimal value 19 into binary:

In this example, the remainder digits, when written left to right from the last one to the first, are 10011. This is the binary form of the decimal value 19. To confirm this, we can convert this value back to decimal form using the binary-to-decimal algorithm.

In every computer, there is a maximum number of binary digits that can be used to store an integer. Typically, this value is 16, 32, or 64 bits. Once we have fixed this maximum number of bits (as part of the design of the computer), we also have fixed the largest unsigned whole number that can be represented in this computer. For example, Figure 4.2 used at most 5 bits to represent binary numbers. The largest value that could be represented is 11111, not unlike the number 99999, which is the maximum mileage value that can be represented on a five-digit decimal odometer. 11111 is the binary representation for the decimal integer 31. If there were 16 bits available on a given computer, rather than 5, then the largest integer that could be represented is

This quantity is . Unsigned integers larger than this cannot be represented with 16 binary digits. Any operation on this computer that produces an unsigned value greater than 65,535 results in the error condition called **arithmetic overflow**. This is an attempt to represent an integer that exceeds the maximum allowable value. The computer could be designed to use more than 16 bits to represent integers, but no matter how many bits are ultimately used, there is always a maximum value beyond which the computer cannot correctly represent any integer. This characteristic is one of the major differences between the disciplines of mathematics and computer science. In mathematics, a quantity may usually take on any value, no matter how large. Computer science must deal with a finite—and sometimes quite limited—set of possible representations, and it must handle the errors that occur when those limits are exceeded.

Arithmetic in binary is quite easy because we have only 2 digits to deal with rather than 10. Therefore, the rules that define arithmetic operations such as addition and subtraction have only entries, rather than the entries for decimal digits. For example, here are the four rules that define binary addition:

The last rule says that , which has the decimal value 2.

To add two binary numbers, you use the same technique first learned in grade school. Add each column one at a time from right to left, using the binary addition rules shown above. In the column being added, you write the sum digit under the line and any carry digit produced is written above the next column to the left. For example, addition of the two binary values 7 (00111) and 14 (01110) proceeds as follows:

Start by adding the two digits in the rightmost column—the 1 and 0. This produces a sum of 1 and a carry digit of 0; the carry digit gets “carried” to the second column.

Now add the carry digit from the previous column to the two digits in the second column, which gives 0 + 1 + 1. From the rules above, we see that the (0 + 1) produces a 1. When this is added to the value 1, it produces a sum of 0 and a new carry digit of 1.

A Not So Basic Base

The decimal system has been in use for so long that most people cannot imagine using a number system other than base 10. Tradition says it was chosen because we have 10 fingers and 10 toes. However, the discussion of the past few pages should convince you that there is nothing unique or special about decimal numbering, and the basic operations of arithmetic (addition, subtraction, multiplication, and division) work just fine in other bases, such as base 2. In addition to binary, computer science makes frequent use of *octal* (base 8) and *hexadecimal* (base 16). Furthermore, it is not only computers that utilize nondecimal bases. For example, the Native American Yuki tribe of northern California reportedly used base 4, or *quaternary* numbers, counting using the spaces between fingers rather than on the fingers themselves. The pre-Columbian Mayans of Mexico and Central America used a *vigesimal* system, or base 20, whereas ancient Babylonians employed *sexagesimal*, or base 60 (and we are quite sure that members of both cultures had the same number of fingers and toes as 21st-century human beings!).

Adding the two digits in the third column plus the carry digit from the second column produces 1 + 1 + 1, which is 11, or a sum of 1 and a new carry digit of 1.

Continuing in this right-to-left manner until we reach the leftmost column produces the final result, 10101 in binary, or 21 in decimal.

Signed Numbers. Binary digits can represent not only whole numbers but also other forms of data, including signed integers, decimal numbers, and characters. For example, to represent signed integers, we can use the leftmost bit of a number to represent the sign, with 0 meaning positive (+) and 1 meaning negative (−). The remaining bits are used to represent the magnitude of the value. This form of signed integer representation is termed **sign/magnitude notation**, and it is one of a number of different techniques for representing positive and negative whole numbers. For example, to represent the quantity −49 in sign/magnitude, we could use seven binary digits with 1 bit for the sign and 6 bits for the magnitude:

The value + 3 would be stored like this:

You might wonder how a computer knows that the seven-digit binary number 1110001 in the first example above represents the signed integer value −49 rather than the unsigned whole number 113.

The answer to this question is that a computer does *not* know. A sequence of binary digits can have many different interpretations, and there is no fixed, predetermined interpretation given to any binary value. A binary number stored in the memory of a computer takes on meaning only because it is used in a certain way. If we use the value 1110001 as though it were a signed integer, then it will be interpreted that way and will take on the value −49. If it is used, instead, as an unsigned whole number, then that is what it will become, and it will be interpreted as the value 113. The meaning of a binary number stored in memory is based solely on the context in which it is used.

Initially, this might seem strange, but we deal with this type of ambiguity all the time in natural languages. For example, in the Hebrew language, letters of the alphabet are also used as numbers. Thus the Hebrew character aleph (א) can stand for either the letter A or the number 1. The only way to tell which meaning is appropriate is to consider the context in which the character is used. Similarly, in English, the word *ball* can mean

either a round object used to play games or an elegant formal party. Which interpretation is correct? We cannot say without knowing the context in which the word is used. The same is true for values stored in the memory of a computer system. It is the context that determines the meaning of a binary string.

Sign/magnitude notation is quite easy for people to work with and understand, but, surprisingly, it is used rather infrequently in real computer systems. The reason is the existence of the very "messy" and unwanted signed number: 10000 … 0000. Because the leftmost bit is a 1, this value is treated as negative. The magnitude is 0000 … 0000. Thus this bit pattern represents the numerical quantity "negative zero," a value that has no real mathematical meaning and should not be distinguished from the other representation for zero, 00000 … 0000. The existence of two distinct bit patterns for a single numerical quantity causes some significant problems for computer designers.

For example, assume we are executing the following algorithmic operation on two signed numbers *a* and *b*:

when *a* has the value 0000 … 0 and *b* has the value 1000 … 0. Should they be considered equal to each other? Numerically, the value −0 does equal +0, so maybe we should do operation 1. However, the two bit patterns are not identical, so maybe these two values are not equal, and we should do operation 2. This situation can result in programs that execute in different ways on different machines.

Therefore, computer designers tend to favor signed integer representations that do not suffer from the problem of two zeros. One of the most widely used is called **two's complement representation**. To understand how this method works, you need to write down, in circular form, all binary patterns from 000 … 0 to 111 … 1 in increasing order. Here is what that circle might look like using three-digit numbers:

In this diagram, the positive numbers begin at 000 and proceed in order around the circle to the right. Negative numbers begin at 111 and proceed in order around the circle to the left. The leftmost digit specifies whether the number is to be given a positive interpretation or a negative interpretation .

Bit Pattern	***Decimal Value***
0 0 0	0
0 0 1	+1
0 1 0	+2
0 1 1	+3
1 0 0	−4
1 0 1	−3
1 1 0	−2
1 1 1	−1

In this representation, if we add, for example, 3 + (−3), we get 0, as expected:

Note that in the two's complement representation, there is only a single zero, the binary number 000 … 0. However, the existence of a single pattern for zero leads to another unusual situation. The total number of values that can be represented with *n* bits is , which is always an even number. In the previous example, , so there were possible values. One of these is used for 0, leaving seven remaining values, which is an odd number. It is impossible to divide these seven patterns equally between the positive and negative numbers, and in this example we ended up with four negative values but only three positive ones. The pattern that was previously "negative zero" (100) now represents the value −4, but there is no equivalent number on the positive side, that is, there is no binary pattern that represents +4. In the two's complement representation of signed integers, you can always represent one more negative number than positive. This is not as severe a problem as having two zeros, though, and two's complement is widely used for representing signed numbers inside a computer.
This has been only a brief introduction to the two's complement representation. A Challenge Work problem at the end of this chapter invites you to investigate further the underlying mathematical foundations of this interesting representational technique.

Fractional Numbers. Fractional numbers, such as 12.34 and −0.001275, can also be represented in binary by using the signed-integer techniques we have just described. To do that, however, we must first convert the number to **scientific notation**:

where *M* is the *mantissa*, *B* is the *exponent base* (usually 2), and *E* is the *exponent*. For example, assume we want to represent the decimal quantity +5.75. In addition, assume that we will use 16 bits to represent the number, with 10 bits allocated for representing the mantissa and 6 bits for the exponent. (The exponent base *B* is assumed to be 2 and is not explicitly stored.) Both the mantissa and the exponent are signed integer numbers, so we can use either the sign/magnitude or two's complement notations that we just learned to represent each of these two fields. (In all the following examples, we have chosen to use sign/magnitude notation.)

In binary, the value 5 is 101. To represent the fractional quantity 0.75, we need to remember that the bits to the right of the decimal point (or binary point in our case) have the positional values , , , and so on, where *r* is the base of the numbering system used to represent the number. When using decimal notation, these position values are the tenths , hundredths , thousandths , and so on. Because *r* is 2 in our case, the positional values of the digits to the right of the binary point are halves , quarters , eighths , sixteenths , and so on. Thus,

Therefore, in binary . Using scientific notation, and an exponent base , we can write this value as
Next, we must *normalize* the number so that its first significant digit is immediately to the right of the binary point. As we move the binary point, we adjust the value of the exponent so that the overall value of the number remains unchanged. If we move the binary point to the left one place (which makes the value smaller by a factor of 2), then we add 1 to the exponent (which makes it larger by a factor of 2). We do the reverse when we move the binary point to the right.

We now have the number in the desired format and can put all the pieces together. We separately store the mantissa (excluding the binary point, which is assumed to be to the left of the first significant digit) and the exponent, both of which are signed integers and can be represented in sign/magnitude notation. The mantissa is stored with its sign—namely, 0, because it is a positive quantity—followed by the assumed binary point, followed by the magnitude of the mantissa, which in this case is 10111. Next we store the exponent, which is +3, or 000011 in sign/magnitude. The overall representation, using 16 bits, is

For another example, let's determine the internal representation of the fraction −5/16.

Textual Information. To represent textual material in binary, the system assigns to each printable letter or symbol in our alphabet a unique number (this assignment is called a *code mapping*), and then it stores that symbol internally using the binary equivalent of that number. For example, here is one possible mapping of characters to numbers, which uses 8 bits to represent each character.

To store the four-character string "BAD!" in memory, the computer would store the binary representation of each individual character using the above 8-bit code.

We have indicated above that the 8-bit numeric quantity 10000001 is interpreted as the character "!". However, as we mentioned earlier, the only way a computer knows that the 8-bit value 10000001 represents the symbol "!" and not the unsigned integer value 129 (128 + 1) or the signed integer value −1 (, magnitude is 1) is by the context in which it is used. If these 8 bits are sent to a display device that expects to be given characters, then this value will be interpreted as an "!". If, on the other hand, this 8-bit value is sent to an arithmetic unit that adds unsigned numbers, then it will be interpreted as 129 in order to make the addition operation meaningful. It is critical to remember that every pattern of binary digits has multiple interpretations. The correct interpretation is determined only when that binary pattern is used in a specific way.
To facilitate the exchange of textual information, such as Facebook posts or email, between computer systems, it would be most helpful if everyone used the same code mapping. Fortunately, this is pretty much the case. Initially, the most widely used code set for representing characters internally in a computer system was **ASCII**, an acronym for the American Standard Code for Information Interchange. ASCII is an international standard for representing textual information that uses 8 bits per character, so it is able to encode a total of different symbols. These are assigned the integer values 0 to 255. However, only the numbers 32 to 126 have been assigned to printable characters. The remainder either are unassigned or are used for representing nonprinting control characters such as tab, form feed, and return. Figure 4.3 shows the ASCII conversion table for the numerical values 32–126.

Figure 4.3

ASCII conversion table

Keyboard Character	Binary ASCII Code	Integer Equivalent	Keyboard Character	Binary ASCII Code	Integer Equivalent
(blank)	00100000	32	P	01010000	80
!	00100001	33	Q	01010001	81
"	00100010	34	R	01010010	82
#	00100011	35	S	01010011	83
$	00100100	36	T	01010100	84
%	00100101	37	U	01010101	85
&	00100110	38	V	01010110	86
'	00100111	39	W	01010111	87
(	00101000	40	X	01011000	88
)	00101001	41	Y	01011001	89
*	00101010	42	Z	01011010	90
+	00101011	43	[	01011011	91
’	00101100	44	\	01011100	92
–	00101101	45	]	01011101	93
.	00101110	46	^	01011110	94
/	00101111	47	_	01011111	95
0	00110000	48	`	01100000	96
1	00110001	49	a	01100001	97
2	00110010	50	b	01100010	98
3	00110011	51	c	01100011	99

Keyboard Character	Binary ASCII Code	Integer Equivalent	Keyboard Character	Binary ASCII Code	Integer Equivalent
4	00110100	52	d	01100100	100
5	00110101	53	e	01100101	101
6	00110110	54	f	01100110	102
7	00110111	55	g	01100111	103
8	00111000	56	h	01101000	104
9	00111001	57	i	01101001	105
:	00111010	58	j	01101010	106
;	00111011	59	k	01101011	107
<	00111100	60	l	01101100	108
=	00111101	61	m	01101101	109
>	00111110	62	n	01101110	110
?	00111111	63	o	01101111	111
@	01000000	64	p	01110000	112
A	01000001	65	q	01110001	113
B	01000010	66	r	01110010	114
C	01000011	67	s	01110011	115
D	01000100	68	t	01110100	116
E	01000101	69	u	01110101	117
F	01000110	70	v	01110110	118
G	01000111	71	w	01110111	119
H	01001000	72	x	01111000	120

Keyboard Character	Binary ASCII Code	Integer Equivalent	Keyboard Character	Binary ASCII Code	Integer Equivalent
I	01001001	73	y	01111001	121
J	01001010	74	z	01111010	122
K	01001011	75	{	01111011	123
L	01001100	76	:	01111100	124
M	01001101	77	]	01111101	125
N	01001110	78	~	01111110	126
O	01001111	79			

However, the code set called **Unicode**, developed in the early 1990s, has gained in popularity because it uses, at a minimum, a 16-bit representation for characters rather than the 8-bit format of ASCII. This means that it is able to represent at least unique characters instead of the of ASCII. (*Note*: Unicode is a superset of ASCII. The decimal values 0–127 represent exactly the same characters in both ASCII and Unicode.) It might seem as though 256 characters should be more than enough to represent all the textual symbols that we would ever need—for example, 26 uppercase letters, 26 lowercase letters, 10 digits, and a few dozen special symbols, such as + = − { }] [\: ” ? > < . , ; % $ # @. Add that all together and it still totals only about 100 symbols, far less than the 256 that can be represented in ASCII. However, that is true only if we limit our text to Arabic numerals and the Roman alphabet. The world is growing more connected all the time—helped along by computers, networks, and the web—and it is critically important that computers are able to represent and exchange textual information using the widest possible range of alphabets. When we start assigning codes to symbols drawn from alphabets such as Russian, Arabic, Chinese, Hebrew, Greek, Thai, Bengali, Sanskrit, Cherokee, Inuit, and Braille, as well as mathematical symbols, linguistic marks such as the tilde, umlaut, and accent grave, and numerous graphical symbols, it quickly becomes clear that ASCII does not have enough room to represent them all. Unicode, with its 16 bits and space for over 65,000 symbols, was initially considered large enough to accommodate this enormous range of text. However, it turned out that even 16 bits proved insufficient to handle the exploding range of textual information being produced around the world, and Unicode now includes a 32-bit variant in which each separate character is represented using 32 binary digits, allowing for the possibility of distinct textual symbols, or about 4 billion! Currently Unicode has assigned formal mappings to more than 128,000 characters drawn from over 135 modern and historical alphabets, including such rarities as Egyptian hieroglyphics. The Unicode home page, which gives all the current standard mappings, is located at www.unicode.org.

4.2.2 Binary Representation of Sound and Images

During the first 30 to 40 years of computing, the overwhelming majority of applications, such as word processing and spreadsheets, were text based and limited to the manipulation of characters, words, and numbers. However, sound and images are now as important a form of representation as text and numbers because of the popularity of digitally encoded music, the rapid emergence of digital photography, the popularity of streaming videos, and the almost universal availability of online digital movies. Most of us, whether computer specialists or not, have had the experience of playing MP3 sound files, emailing vacation pictures to friends and family, or enjoying a YouTube video clip. In this section, we take a brief look at how sounds and images are represented in computers, using the same binary numbering system that we have been discussing.

Sound is analog information, unlike the digital format used to represent text and numbers discussed in the previous section. In a **digital representation**, the allowable values for a given object are drawn from a finite set, such as letters {A, B, C, …, Z} or a subset of integers {0, 1, 2, 3, …, MAX}.

In an **analog representation**, objects can take on any value. For example, in the case of sound, a tone is a continuous sinusoidal waveform that varies in a regular periodic fashion over time, as shown in Figure 4.4. (*Note*: This diagram shows only a single tone. Complex sounds, such as symphonic music, are composed of multiple overlapping waveforms. However, the basic ideas are the same.)

Figure 4.4 Example of sound represented as a waveform

The **amplitude** (height) of the wave is a measure of its loudness—the greater the amplitude, the louder the sound. The **period** of the wave, designated as T, is the time it takes for the wave to make one complete cycle. The **frequency** f is the total number of cycles per unit time measured in cycles/second, also called *hertz*, and defined as . The frequency is a measure of the *pitch*, the highness or lowness of a sound. The higher the frequency, the higher the perceived tone. A healthy, young human ear can generally detect sounds in the range of 20 to 20,000 hertz.
To store a waveform (such as the one in Figure 4.4) in a computer, the analog signal first must be **digitized**, that is, converted to a digital representation. This can be done using a technique known as **sampling**. At fixed time intervals, the amplitude of the signal is measured and stored as an integer value. The wave is thus represented in the computer in digital form as a sequence of sampled numerical amplitudes. For example, Figure 4.5(a)shows the sampling of the waveform of Figure 4.4.

Figure 4.5 Digitization of an analog signal

Sampling the original signal

Recreating the signal from the sampled values

This signal can now be stored inside the computer as the series of signed integer values 3, 7, 7, 5, 0, −3, −6, −6, …, where each numerical value is encoded in binary using the techniques described in the previous section. From these stored digitized values, the

computer can recreate an approximation to the original analog wave. It would first generate an amplitude level of 3, then an amplitude level of 7, then an amplitude level of 7, and so on, as shown in Figure 4.5(b). These values would be sent to a sound-generating device, such as stereo speakers, which would produce the actual sounds based on the numerical values received.

The accuracy with which the original sound can be reproduced is dependent on two key parameters—the sampling rate and the bit depth. The **sampling rate** measures how many times per second we sample the amplitude of the sound wave. Obviously, the more often we sample, the more accurate the reproduction. Note, for example, that the sampling shown in Figure 4.5(a) appears to have missed the peak value of the wave because the peak occurred between two sampling intervals. Furthermore, the more often we sample, the greater the range of frequencies that can be captured; if the frequency of a wave is greater than or equal to the sampling rate, we might not sample any points whatsoever on an entire waveform. For example, Figure 4.6 shows the sampling interval *t*, which is exactly equal to the period *T* of the wave being measured.

Figure 4.6 Sampling interval *t* is the same as period *T*

This rate of sampling produces a constant amplitude value, totally distorting the original sound. In general, a sampling rate of *R* samples/second allows you to reproduce all frequencies up to about *R*/2 hertz. Because the human ear can normally detect sound up to about 20,000 hertz, a sampling rate of at least 40,000 samples per second is necessary to capture all audible frequencies.

The **bit depth** is the number of bits used to encode each sample. In the previous section, you learned that ASCII is an 8-bit character code, allowing for 256 unique symbols. Unicode uses a minimum of 16 bits, allowing for more than 65,000 symbols and greatly increasing the number of symbols that can be represented. The same trend can be seen in sound reproduction. Initially, 8 bits per sample was the standard, but the 256 levels of amplitude that could be represented turned out to be insufficient for the sophisticated high-end sound systems produced and marketed today. Most audio encoding schemes today use either 16 or 24 bits per sample level, allowing for either 65,000 or 16,000,000 distinct amplitude levels.

There are many audio-encoding formats in use today, including AAC (Advanced Audio Coding), which is the standard audio format for Apple's iPhone and iPad, WMA (Windows Media Audio), WAV (Waveform Audio File Format), and MIDI (Musical Instrument Digital Interface). Another popular and widely used digital audio format is *MP3*, an acronym for MPEG-1 (later updated to MPEG-2), Audio Level 3 Encoding. This is a digital audio encoding standard established by the Motion Picture Experts Group (MPEG), a committee of the International Organization for Standardization (ISO) of the United Nations. MP3 samples sound signals at the rate of 44,100 samples/ second, using 16 bits per sample. This produces high-quality sound reproduction, which is why MP3 is the most widely used format for rock, opera, and classical music.

An image, such as a photograph, is also analog data but can also be stored using binary representation. An image is a continuous set of intensity and color values that can be digitized by sampling the analog information, just as is done for sound. The sampling process, often called *scanning*, consists of measuring the intensity values of distinct

points located at regular intervals across the image's surface. These points are called *pixels*, short for picture elements, and the more pixels used, the more accurate the encoding of the image. The average human eye cannot accurately discern components closer together than about 0.05–0.1 mm, so if the pixels, or dots, are sufficiently dense, they appear to the human eye as a single, contiguous image. For example, a high-quality digital camera stores about 10–15 million pixels per photograph. For a 3 in. × 5 in. image, this is about , or 900 pixels per linear inch. This means the individual pixels are separated by about 1/900th of an inch, or roughly 0.02 mm—much too close together to be individually visualized. (By comparison, an iPhone 7 camera contains two 12-million-pixel cameras.) Figure 4.7 enlarges a small section of a digitized photograph to better show how it is stored internally as a set of discrete picture elements.

Figure 4.7 Example of a digitized photograph

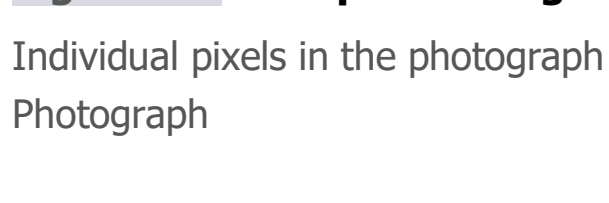


Photo by Maris Sidenstecker

One of the key questions we need to answer is how much information is stored for each pixel. Suppose we want to store a representation of a black-and-white image. The easiest and most space-efficient approach is to mark each pixel as either white, stored as a binary 0, or black, stored as a binary 1. The only problem is that this produces a stark *black-and-white image*, with a highly sharp and unpleasant visual contrast. A much better way, although it takes more storage, is to represent black-and-white images using a *gray scale* of varying intensity. For example, if we use 3 bits per pixel, we can represent shades of intensity from level 0, pure white, to level 7, pure black. An example of this eight-level gray scale is shown in Figure 4.8. If we wanted more detail than is shown there, we could use 8 bits per pixel, giving us distinct shades of gray.

Figure 4.8 An eight-level gray scale

We now can encode our image as a sequence of numerical pixel values, storing each row of pixels completely, from left to right, before moving down to store the next row. Each pixel is encoded as an unsigned binary value representing its gray scale intensity. This form of image representation is called **raster graphics**, and it is used by such well-known graphics standards as JPEG (Joint Photographer Experts Group), GIF (Graphics Interchange Format), and BMP (bitmap).

Today, most images are not black and white, but are in color. To digitize color images, we still measure the intensity value of the image at a discrete set of points, but we need to store more information about each pixel. The most common format for storing color images is the **RGB encoding scheme**, RGB being an acronym for red-green-blue. This technique describes a specific color by capturing the individual contribution to a pixel's color of each of the three colors, red, green, and blue. It uses one **byte**, or 8 bits, for each color, allowing us to represent an intensity range of 0 to 255 for each color. The value 0 means that there is no contribution from this color, whereas the value 255 means a full contribution of this color.

For example, the color magenta is an equal mix of pure red and blue, which would be RGB encoded as (255, 0, 255):

Red	*Green*	*Blue*
255	0	255

The color hot pink is produced by setting the three RGB values to

Red	*Green*	*Blue*
255	105	180

and harvest gold is rendered as

Red	*Green*	*Blue*
218	165	32

Using 3 bytes of information per pixel—24 bits—allows us to represent distinct colors, about 16.7 million. This 24-bit color-encoding scheme is often referred to as *True Color*, and it provides an enormous range of shades and an extremely accurate color image reproduction. That is why it is the encoding scheme used in the JPEG color imaging format. Newer high-resolution video standards are increasing the theoretical number of colors that can be represented on a screen. For example, the HDMI (High-Definition Multimedia Interface) 1.4 standard allows pixel bit depths of up to 48 bits, allowing for distinct colors, over 280 trillion!

However, representing 280 trillion, or even 16 million, colors requires a huge amount of memory space, and some image representation techniques reduce that value by using what is called a *color palette*. While theoretically supporting 16+ million different colors, they only allow you to use 256 (or some other small number) at any one time, just as a painter may have a lot of colors in his or her studio but puts only a few on the palette at a time. With a palette size of 256, we can encode each pixel using only 8 bits rather than 24, because , thus reducing storage space demands by almost 67%. Each of these 256 values does not represent an explicit RGB color value but rather an index into a palette, or a color table. This index specifies which color on the palette is to be used to draw this pixel. This is the technique used, for example, in GIF, which uses a palette that can hold as few as 2 colors or as many as 256.

Sound and image data typically require huge amounts of storage, far more than is required for the numbers and text discussed in Section 4.2.1. For example, a 300-page novel contains about 100,000 words. Each word has on average about five characters and, as discussed in the previous section, each individual character can be encoded into Unicode using 16 bits. Thus, the total number of bits needed to represent this book is roughly

By comparison, 1 minute of sound recording encoded using the MP3 standard, which samples 44,100 times per second using a bit depth of 16 bits per sample, requires

It takes more than five times the space to store the information in 1 minute of music as it does to store an entire 300-page book! Similarly, to store a single photograph taken using the iPhone 7 digital camera with 12 million pixels using 24-bit True-Color raster graphics requires:

A single iPhone color photograph could require 36 times more storage than an entire novel.

As these examples clearly show, the storage of analog information, such as sound, images, voice, and video, is enormously space intensive, and an important area of computer science research—**data compression**—is directed at addressing just this issue. Data compression algorithms attempt to represent information in ways that preserve accuracy while using significantly less space.

For example, a simple compression technique that can be used on almost any form of data is *run-length encoding*. This method replaces a sequence of identical values , , …, by a pair of values (*v*, *n*), which indicates that the value *v* is replicated *n* times. If both *v* and *n*require 1 byte of storage, then we have reduced the total number of bytes required to store this sequence from *n* down to 2. Using this method, we could encode the following 5 × 3 image of the letter *E*, where , and :

255	255	255
255	0	0
255	255	255
255	0	0
255	255	255

like this:

Run-length encoding reduces the number of bytes needed to store this image from 15, using the raster graphics representation, to the 10 bytes shown above. Compression schemes are usually evaluated by their **compression ratio**, which measures how much they reduce the storage requirements of the data:

For the example shown above, this ratio is

meaning the scheme reduces the amount of space needed to store the image by 33%. Applied to a larger image, this might mean that a 4-million-bit representation could be reduced to about 2.7 million bits, a significant savings.

Another popular compression technique is the use of *variable-length code sets*, which are often used to compress text but can also be used with other forms of data. In Section 4.2.1, we showed that textual symbols, such as *A*, *z*, and *#*, are represented internally by a code mapping that uses exactly the same number of bits for every symbol, either 8 (ASCII) or 16 (Unicode). That is a wasteful approach because some symbols occur much more frequently than others. (For example, in English, the letters *E* and *A* are much more common than *J*, *Q*, *X*, and *Z*.) If the codes representing commonly used symbols were shorter than the codes representing the less-common symbols, this could result in a significant saving of space.

Assume that we want to encode the Hawaiian alphabet, which only contains the five vowels *A*, *E*, *I*, *O*, and *U*, and the seven consonants *H*, *K*, *L*, *M*, *N*, *P*, and *W*. If we were to store these characters using a fixed-length code set, we would need at least 4 bits/symbol, because . Figure 4.9(a) shows one possible encoding of these 12 letters using a fixed-length, 4-bit encoding. However, if we know that *A* and *I* are the most commonly used letters in the Hawaiian alphabet, with *H* and *W* next, we could

represent *A* and *I* using 2 bits, *H* and *W*using 3 bits, and the remaining letters using either 4, 5, 6, or 7 bits, depending on their frequency. However, we must be sure that if the 2-bit sequence is used to represent an A, for example, then no other symbol representation can start with the same 2-bit sequence. Otherwise, if we saw the sequence , we would not know if it was an *A* or the beginning of another character. One possible variable-length encoding for the Hawaiian alphabet is shown in Figure 4.9(b).

Figure 4.9

Using variable-length code sets

Fixed length

Variable length

Letter	4-Bit Encoding	Variable-Length Encoding
A	0000	00
I	0001	10
H	0010	010
W	0011	110
E	0100	0110
O	0101	0111
M	0110	11100
K	0111	11101
U	1000	11110
N	1001	111110
P	1010	1111110
L	1011	1111111
	(a)	(b)

Representing the six-character word HAWAII using the fixed-length 4-bit encoding scheme of Figure 4.9(a) requires bits. Representing it with the variable-length encoding shown in Figure 4.9(b) produces the following:

H	A	W	A	I	I
010	00	110	00	10	10

This is a total of 14 bits, producing a compression ratio of , a reduction in storage demands of about 42%.

These two techniques are examples of what are called **lossless compression** schemes. This means that no information is lost in the compression, and it is possible to exactly reproduce the original data. **Lossy compression** schemes compress data in a way that does not guarantee that all of the information in the original data can be fully and completely recreated. They trade a possible loss of accuracy for a higher compression ratio because the small inaccuracies in sounds or images are often undetectable to the human ear or eye. Many of the compression schemes in widespread use today, including MP3 and JPEG, use lossy techniques, which permit significantly greater compression ratios than would otherwise be possible. Using lossy JPEG, for example, it is possible to achieve compression ratios of 10:1, 20:1, or more, depending on how much loss of detail we are willing to tolerate. This compares with the values of 1.5 and 1.7 in the earlier described lossless schemes. Using these lossy compression schemes, that 288-million bit, high-resolution image mentioned earlier could be reduced to only 15 or 30 million bits, certainly a much more manageable value. Data compression schemes are an essential component in allowing us to represent multimedia information in a concise and manageable way.

4.2.3 The Reliability of Binary Representation

At this point, you might be wondering: Why are we bothering to use binary? Because we use a decimal numerical system for everyday tasks, wouldn't it be more convenient to use a base-10 representation for both the external and the internal representation of information? Then there would be no need to go through the time-consuming conversions diagrammed in Figure 4.1 or to learn the binary representation techniques discussed in the previous two sections.

As we stated in the Special Interest Box, "A Not So Basic Base," there is absolutely no theoretical reason why one could not build a "decimal" computer or, indeed, a computer that stored numbers using base 3 (ternary), base 8 (octal), or base 16 (hexadecimal). The techniques described in the previous two sections apply to information represented in *any*base of a positional numbering system, including base 10.

Computers use binary representation not for any theoretical reasons but for reasons of *reliability*. As we will see shortly, computers store information using electronic devices, and the internal representation of information must be implemented in terms of electronic quantities such as currents and voltage levels.

Building a base-10 "decimal computer" requires finding a device with 10 distinct and stable energy states that can be used to represent the 10 unique digits (0, 1, …, 9) of the decimal system. For example, assume there exists a device that can store electrical charges in the range 0 to +45 volts. We could use it to build a decimal computer by letting certain voltage levels correspond to specific decimal digits:

Voltage Level	Corresponds to This Decimal Digit
+0	0
+5	1

+10	2
+15	3
+20	4
+25	5
+30	6
+35	7
+40	8
+45	9

Storing the two-digit decimal number 28 requires two of these devices, one for each of the digits in the number. The first device would be set to +10 volts to represent the digit 2, and the second would be set to +40 volts to represent the digit 8.

Although this is theoretically feasible, it is certainly not recommended. As electrical devices age, they become unreliable, and they may *drift*, or change their energy state, over time. What if the device representing the value 8 (the one set to +40 volts) lost 6% of its voltage (not a huge amount for an old, well-used piece of equipment)? The voltage would drop from+40 volts to about +37.5 volts. The question is whether the value +37.5 represents the digit 7 (+35) or the digit 8 (+40). It is impossible to say. If that same device lost another 6% of its voltage, it would drop from +37.5 volts to about +35 volts. Our 8 has now become a 7, and the original value of 28 has unexpectedly changed to 27. Building a reliable decimal machine would be an engineering nightmare.

The problem with a base-10 representation is that it needs to store 10 unique symbols, and, therefore, it needs devices that have 10 stable states. Such devices are extremely rare. Electrical systems tend to operate best in a *bistable environment*, in which there are only two (rather than 10) stable states separated by a huge energy barrier. Examples of these bistable states include the following:

- Full on/full off
- Fully charged/fully discharged
- Charged positively/charged negatively
- Magnetized/nonmagnetized
- Magnetized clockwise/magnetized counterclockwise

In the binary numbering system, there are only two symbols (0 and 1), so we can let one of the two stable states of our bistable device represent a 0 and the other a 1. This is a much more reliable way to represent information inside a computer.

For example, if we use binary rather than decimal to store data in our hypothetical electronic device that stores voltages in the range from 0 to +45 volts, the representational scheme becomes much simpler:

0 volts	= 0 (full off)
+45 volts	= 1 (full on)

Now a 6% or even a 12% drift doesn't affect the interpretation of the value being represented. In fact, it takes an almost 50% change in voltage level to create a problem

in interpreting a stored value. The use of binary for the internal representation of data significantly increases the inherent reliability of a computer. This single advantage is worth all the time it takes to convert from decimal to binary for internal storage and from binary to decimal for the external display of results.

4.2.4 Binary Storage Devices

As you learned in the previous section, binary computers can be built out of any bistable device. This idea can be expressed more formally by saying that it is possible to construct a binary computer and its internal components using any hardware device that meets the following four criteria:

1. The device has two stable energy states (one for a 0, one for a 1).
2. These two states are separated by a large energy barrier (so that a 0 does not accidentally become a 1, or vice versa).
3. It is possible to sense which state the device is in (to see whether it is storing a 0 or a 1) without permanently destroying the stored value.
4. It is possible to switch the state from a 0 to a 1, or vice versa, by applying a sufficient amount of energy.

There are many devices that meet these conditions, including some surprising ones such as a light switch. A typical light switch has two stable states (ON and OFF). These two states are separated by a large energy barrier so that a switch that is in one state will not accidentally change to the other. We can determine what state the switch is in by looking to see whether the label says ON or OFF (or just by looking at the light), and we can change the state of the switch by applying a sufficient amount of energy via our fingertips. Thus it would be possible to build a reliable (albeit very slow and bulky) binary computing device out of ordinary light switches and fingertips!

As you might imagine, computer systems are not built from light switches, but they have been built using a wide range of devices. This section describes two of these devices.

Magnetic cores were used to construct computer memories for about 20 years. From roughly 1955 to 1975, this was by far the most popular storage technology—even today, the memory unit of a computer is sometimes referred to as *core memory* even though it has been decades since magnetic cores have been used.

A *core* is a small, magnetizable, iron oxide-coated "doughnut," about 1/50 of an inch in inner diameter, with wires strung through its center hole. The two states used to represent the binary values 0 and 1 are based on the *direction* of the magnetic field of the core. When electrical current is sent through the wire in one specific direction, say left to right, the core is magnetized in a counterclockwise direction.* This state could represent the binary value 0. Current sent in the opposite direction produces a clockwise magnetic field that could represent the binary value 1. These scenarios are diagrammed in Figure 4.10. Because magnetic fields do not change much over time, these two states are highly stable, and they form the basis for the construction of memory devices that store binary numbers.

Figure 4.10 Using magnetic cores to represent binary values

In the early 1970s, core memories began to be replaced by smaller, cheaper technologies that required less power and were easier to manufacture. One-fiftieth of an inch in diameter and a few grams of weight might not seem like much, but it can produce a bulky and unworkable structure when memory units must contain millions or billions of bits. For example, a typical core memory from the 1950s or 1960s had a density of about . The memory in a modern computer typically has at least 16 GB (1 **gigabyte** = 1 billion 8-bit bytes), which is more than 128 billion bits. If we had to construct a modern memory unit using cores, it would require of space, which is a square about 416,000 inches, or 1,330 feet, on a side. Built from cores, our memory unit would stand more than 100 stories high!

Today, the elementary building block for all modern computer systems is no longer the core but the transistor. A **transistor** is much like the light switch mentioned earlier. It can be in an OFF state, which does not allow electricity to flow, or in an ON state, in which electricity can pass unimpeded. However, unlike the light switch, a transistor is a solid-state device that has no mechanical or moving parts. The switching of a transistor from the OFF to the ON state, and vice versa, is done electronically rather than mechanically. This allows the transistor to be fast as well as extremely small. A typical transistor can switch states in a billionth of a second, and at current technology levels about 1–2 billion transistors can fit into a space only . Furthermore, hardware technology is changing so rapidly that both these numbers might be out of date by the time you read these words.

Transistors are constructed from special materials called *semiconductors*, such as silicon and gallium arsenide. A large number of transistors, as well as the electrical conducting paths that connect them, can be printed photographically on a wafer of silicon to produce a device known as an *integrated circuit* or, more commonly, a *chip*. The chip is mounted on a *circuit board*, which interconnects all the different chips (e.g., memory, processor, and communications) needed to run a computer system. This circuit board is then plugged into the computer using a set of connectors located on the end of the board. The relationships among transistors, chips, and circuit boards are diagrammed in Figure 4.11. The use of photographic rather than mechanical production techniques has numerous advantages. Because light can be focused very sharply, these integrated circuits can be manufactured in very high densities—high numbers of transistors per square centimeter—and with a very high degree of accuracy. The more transistors that can be packed into a fixed amount of space, the greater the processing power of the computer and the greater the amount of information that can be stored in memory.

Figure 4.11 Relationships among transistors, chips, and circuit boards

Another advantage of photographic production techniques is that it is possible to make a standard template, called a *mask*, which describes the circuit. This mask can be used to produce a virtually unlimited number of copies of that chip, much as a photographic negative can be used to produce an unlimited number of prints.

Together, these characteristics can result in very small and very inexpensive high-speed circuits. Whereas the first computers of the early 1940s (as seen in Chapter 1, Figure 1.6) filled huge rooms and cost millions of dollars, the processor inside a modern workstation contains billions of transistors on a tiny chip just a few centimeters square,

is thousands of times more powerful than those early machines, and can cost less than a few hundred dollars.

The theoretical concepts underlying the physical behavior of semiconductors and transistors, as well as the details of chip manufacture, are well beyond the scope of this book. They are usually discussed in courses in physics or electrical engineering. Instead, we will examine a transistor in terms of the simplified model shown in Figure 4.12 and then use this model to explain its behavior. (Here is another example of the importance of abstraction in computer science.)

Figure 4.12 Simplified model of a transistor

In the model shown in Figure 4.12, each transistor contains three lines—two input lines (control and collector) and one output line (emitter), with each line either in the 1-state, with a high positive voltage, or in the 0-state, with a voltage close to 0. The first input line, called the *control* or the *base*, is used to open or close the switch inside the transistor. If we set the control line to a 1 by applying a sufficiently high positive voltage, the switch closes and the transistor enters the ON state. In this state, current from the input line called the *collector* can flow directly to the single output line called the *emitter*, and the associated voltage can be detected by a measuring device. This ON state could be used to represent the binary value 1. If instead we set the control line to a 0 by applying a voltage close to zero, the switch opens, and the transistor enters the OFF state. In this state, the flow of current through the transistor is blocked and no voltage is detected on the emitter line. The OFF state could be used to represent the binary value 0. This is diagrammed in Figure 4.13.

Figure 4.13 Binary ON and OFF states

This type of solid-state switching device forms the basis for the construction of virtually all computers built today, and it is the fundamental building block for all high-level components described in the upcoming chapters. Remember, however, that there is no theoretical reason why we must use transistors as our "elementary particles" when designing computer systems. Just as cores were replaced by transistors, transistors may ultimately be replaced by some newer, perhaps molecular or biological, technology that is faster, smaller, and cheaper. (Researchers are beginning to investigate the possibility of using DNA molecules as the basic building blocks for computer systems, just as they are the basic building blocks for human "systems.") The only requirements for our building blocks are those given in the beginning of this section—that they be able to represent reliably the two binary values 0 and 1.

Moore's Law and the Limits of Chip Design

Since the development of the first integrated circuits in the 1950s, the number of transistors on a circuit board has been doubling roughly every 24 months. This observation was first reported in a 1965 paper by Gordon E. Moore, the cofounder of Intel, and is now universally referred to as "Moore's law." This doubling has continued unabated for the last 50 years and represents a rate of improvement unequaled in any other technical field. More transistors on a chip means more speed and more power and is the reason for the enormous increase in performance (and decrease in size) of computers in the last 50 years.

The following table details this growth from 1971, when chips held just a few thousand transistors, to today's microprocessors that hold billions.

Processor	Transistor Count	Date
Intel 404	2,300	1971
Intel 8080	4,500	1974
Intel 8088	29,000	1979
Intel 80286	134,000	1982
Intel 80386	275,000	1985
Intel 80486	1,200,000	1989
Pentium	3,100,000	1993
Pentium II	7,500,000	1997
Pentium 4	42,000,000	2000
Itanium 2	220,000,000	2003
Dual-Core Itanium 2	1,400,000,000	2006
Quad-Core Itanium Tukwila	2,000,000,000	2008
nVidiaGeForce 6800 Ultra	3,000,000,000	2011
62-core Xeon Phi	5,000,000,000	2015

It is impossible to maintain this type of exponential growth for an indefinitely extended period of time, and industry analysts have been predicting the demise of Moore's law for the last 10–15 years. However, the ongoing development of new materials and new manufacturing technologies has allowed the industry to continue this phenomenal rate of improvement. But there is a physical limit looming on the horizon that will be the most difficult hurdle yet. As more and more transistors are packed onto a single chip, distances between them get smaller and smaller, and experts estimate that in about 10–20 years intertransistor distances will approach the space between individual atoms. For example, transistors on today's chips are separated by about 25–50 nanometers , only about 250–500 times greater than the diameter of a single atom of silicon, which is about meters. In a few generations, these atomic distances will be reached, and a totally new approach to computer design will be required, perhaps one based on the emerging fields of nanotechnology and quantum computing.

4.3 Boolean Logic and Gates

4.3.1 Boolean Logic

The construction of computer circuits is based on the branch of mathematics and symbolic logic called **Boolean logic**. This area of mathematics deals with rules for manipulating the two logical values *true* and *false*, and it is used to construct circuits that perform operations such as adding numbers, comparing numbers, and fetching instructions. These ideas are part of the branch of computer science known as **hardware design**, also called **logic design**.

It is easy to see the relationship between Boolean logic and computer design: The truth value *true* could represent the binary value 1, and the truth value *false* could represent the binary value 0. Thus anything stored internally as a sequence of binary digits (which, as we saw in earlier sections, is everything stored inside a computer) can also be viewed as a sequence of the logical values true and false, and these values can be manipulated by the operations of Boolean logic.

Let us define a **Boolean expression** as any expression that evaluates to either true or false. For example, the expression is a Boolean expression because it is true if x is 1, and it is false if x has any other value. Similarly, both and ($c > 5.23$) are Boolean expressions. In "traditional" mathematics (the mathematics of real numbers), the operations used to construct arithmetic expressions are +, −, ×, ÷, and , which map real numbers into real numbers. In Boolean logic, the three basic operations used to construct Boolean expressions are AND, OR, and NOT, and they map a set of (true, false) values into a single (true, false) result. (*Note*: There are other Boolean operations, such as XOR, NOR, and NAND, that we mention later in this chapter.)
The rule for performing the AND operation is as follows: If a and b are Boolean expressions, then the value of the expression (a AND b), also written as ($a \cdot b$), is *true* if and only if both $a$and b have the value *true;* otherwise, the expression (a AND b) has the value *false*. Informally, this rule says that the AND operation produces the value *true* if and only if both of its components are true. This idea can be expressed using a structure called a **truth table**, shown in Figure 4.14.

Figure 4.14

Truth table for the AND operation

Inputs		Output *a* AND *b*
a	***b***	**(also written *a* · *b*)**
False	False	False
False	True	False
True	False	False
True	True	True

The two columns labeled Inputs in the truth table of Figure 4.14 list the four possible combinations of true/false values of *a* and *b*. The column labeled Output specifies the value of the expression (*a* AND *b*) for the corresponding values of *a* and *b*.

To illustrate the AND operation, imagine that we want to check whether a test score *S* is in the range 90–100 inclusive. We want to develop a Boolean expression that is true if the score is in the desired range and false otherwise. We cannot do this with a single comparison. If we test only that ($S \geq 90$), then a score of 105, which is greater than or equal to 90, will produce the result *true*, even though it is out of range. Similarly, if we test only that ($S \leq 100$), then a score of 85, which is less than or equal to 100, will also produce a *true*, even though it too is not in the range 90–100.

Instead, we need to determine whether the score *S* is greater than or equal to 90 *and*whether it is less than or equal to 100. Only if both conditions are true can we say that *S* is in the desired range. We can express this idea using the following Boolean expression:

Each of the two expressions in parentheses can be either true or false depending on the value of *S*. However, only if both conditions are true does the expression evaluate to *true*. For example, a score of causes the first expression to be false (70 is not greater than or equal to 90), whereas the second expression is true (70 is less than or equal to 100). The truth table in Figure 4.14 shows that the result of evaluating (*false* AND *true*) is false. Thus, the overall expression is false, indicating (as expected) that 70 is not in the range 90–100.

The second Boolean operation is OR. The rule for performing the OR operation is as follows: If *a* and *b* are Boolean expressions, then the value of the Boolean expression (*a* OR *b*), also written as (*a* + *b*), is *true* if *a* is *true*, if *b* is *true*, or if both are *true*. Otherwise, (*a* OR *b*) has the value *false*. The truth table for OR is shown in Figure 4.15.

Figure 4.15

Truth table for the OR operation

Inputs		Output *a* OR *b*
a	***b***	**(also written *a* + *b*)**
False	False	False
False	True	True
True	False	True
True	True	True

To see the OR operation at work, imagine that a variable called *major* specifies a student's college major. If we want to know whether a student is majoring in either math or computer science, we cannot accomplish this with a single comparison. The test (*major* = math) omits computer science majors, whereas the test (*major* = computer

science) leaves out themathematicians. Instead, we need to determine whether the student is majoring in *either*math or computer science (or perhaps in both). This can be expressed as follows:

If the student is majoring in either one or both of the two disciplines, then one or both of the two terms in the expression are true. Referring to the truth table in Figure 4.15, we see that (*true* OR *false*), (*false* OR *true*), and (*true* OR *true*) all produce the value *true*, which indicates that the student is majoring in at least one of these two fields. However, if the student is majoring in English, both conditions are false. As Figure 4.15 illustrates, the value of the expression (*false* OR *false*) is *false*, meaning that the student is not majoring in either math or computer science.

The final Boolean operator that we examine here is NOT. Unlike AND and OR, which require two operands and are, therefore, called *binary operators*, NOT requires only one operand and is called a *unary operator*, like the square root operation in arithmetic. The rule for evaluating the NOT operation is as follows: If *a* is a Boolean expression, then the value of the expression (NOT *a*), also written as $\bar{a}$, is *true* if *a* has the value *false*, and it is *false* if *a* has the value *true*. The truth table for NOT is shown in Figure 4.16.

Figure 4.16

Truth table for the NOT operation

Input *a*	Output NOT *a* (also written $\bar{a}$)
False	True
True	False

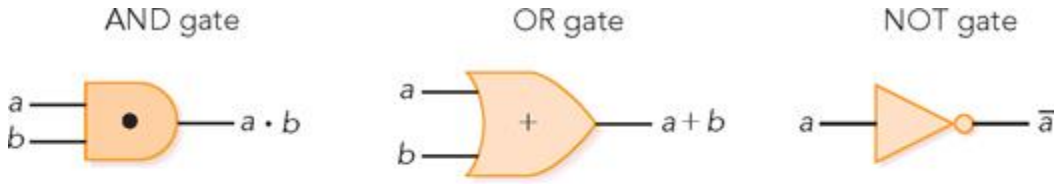
Informally, we say that the NOT operation reverses, or *complements*, the value of a Boolean expression, making it true if currently false, and vice versa. For example, the expression (GPA > 3.5) is true if your grade point average is greater than 3.5, and the expression NOT (GPA > 3.5) is true only under the reverse conditions, that is when your grade point average is less than or equal to 3.5.

AND, OR, and NOT are the three operations of Boolean logic that we use in this chapter. Why have we introduced these Boolean operations in the first place? The previous section discussed hardware concepts such as energy states, electrical currents, transistors, and integrated circuits. Now it appears that we have changed direction and are discussing highly abstract ideas drawn from the discipline of symbolic logic. However, as we hinted earlier and will see in detail in the next section, there is a very close relationship between the hardware concepts of Section 4.2.4 and the operations of Boolean logic. In fact, the fundamental building blocks of a modern computer system (the objects with which engineers actually design) are not the transistors introduced in Section 4.2.4 but the gates that implement the Boolean operations AND, OR, and NOT. Surprisingly, it is the rules of logic—a discipline developed by the Greeks 2,300 years ago and expanded by George Boole (see the Special Interest Box) 160 years ago—that provide the theoretical foundation for constructing modern computer hardware.

4.3.2 Gates

A **gate** is an electronic device that operates on a collection of binary inputs to produce a binary output. That is, it transforms a set of (0,1) input values into a single (0,1) output value according to a specific transformation rule. Although gates can implement a wide range of different transformation rules, the ones we are concerned with in this section are those that implement the Boolean operations AND, OR, and NOT introduced in the previous section. As shown in Figure 4.17, these gates can be represented symbolically, along with the truth tables that define their transformation rules.

Figure 4.17 The three basic gates and their symbols



a	b	a · b
0	0	0
0	1	0
1	0	0
1	1	1

a	b	a + b
0	0	0
0	1	1
1	0	1
1	1	1

a	ā
0	1
1	0

Comparing Figures 4.14, 4.15, 4.16 with Figure 4.17 shows that if the value 1 is equivalent to *true* and the value 0 is equivalent to *false*, then these three electronic gates directly implement the corresponding Boolean operation. For example, an AND gate has its output line set to 1 (set to some level of voltage that represents a binary 1) if and only if both of its inputs are 1. Otherwise, the output line is set to 0 (set to some level of voltage that represents a binary 0). This is functionally identical to the rule that says the result of (*a* AND *b*) is *true* if and only if both *a* and *b* are *true*; otherwise, (*a* AND *b*) is *false.* Similar arguments hold for OR and NOT.

A NOT gate can be constructed from a single transistor, as shown in Figure 4.18, in which the collector is connected to the power supply (Logical-1) and the emitter is connected to the ground (Logical-0). If the control line of the transistor (labeled Input) is set to 1, then the transistor is in the ON state, and it passes current through to the ground. In this case, the voltage on the line labeled Output is 0. However, if Input is set to 0, the transistor is in the OFF state, and it blocks passage of current to the ground. Instead, the current is transmitted to the Output line, producing a value of 1. Thus, the value appearing on the output line of Figure 4.18 is the complement—the NOT—of the value appearing on the input line.

Figure 4.18 Construction of a NOT gate

To construct an AND gate, we begin by connecting two transistors *in series*, as shown in Figure 4.19(a), with the collector line of transistor 1 connected to the power supply (Logical-1) and the emitter line of transistor 2 connected to ground (Logical-0). If both control lines, called Input-1 and Input-2 in Figure 4.19(a), are set to 1, then both transistors are in the ON state, and the current will be connected to ground, resulting in

a value of 0 on the output line. If either (or both) Input-1 or Input-2 is 0, then the corresponding transistor is in the OFF state and does not allow current to pass, resulting in a 1 on the output line. Thus, the output of the gate in Figure 4.19(a) is a 0 if and only if both inputs are a 1; otherwise, it is a 1. This is the exact *opposite* of the definition of AND, and Figure 4.19(a) represents a gate called NAND, an acronym for *NOT AND*. It produces the complement of the AND operation, and it is an important and widely used gate in hardware design.

Figure 4.19 Construction of NAND and AND gates

A two-transistor NAND gate

A three-transistor AND gate

If, however, we want to build an AND gate, then all we have to do is add a NOT gate (of the type shown in Figure 4.18) to the output line. This complements the NAND output and produces the AND truth table of Figure 4.14. This gate is shown in Figure 4.19(b). Note that the NAND of Figure 4.19(a) requires two transistors, whereas the AND of Figure 4.19(b)requires three. This is one reason why NAND gates are widely used to build computer circuits.

To construct an OR gate, we again start with two transistors. However, this time they are connected *in parallel* rather than in series, as shown in Figure 4.20(a).

Figure 4.20 Construction of NOR and OR gates

A two-transistor NOR gate

A three-transistor OR gate

In Figure 4.20(a) if either, or both, of the lines Input-1 and Input-2 are set to 1, then the corresponding transistor is in the ON state, and the current is connected to the ground, producing an output line value of 0. Only if both input lines are 0, effectively shutting off both transistors, will the output line contain a 1. Again, this is the exact opposite to the definition of OR given in Figure 4.15. Figure 4.20(a) is an implementation of a NOR gate, an acronym for NOT *OR*. To convert this to an OR gate, we do the same thing we did earlier—add a NOT gate to the output line. This gate is diagrammed in Figure 4.20(b).

Gates of the type shown in Figures 4.18, 4.19, 4.20 are not abstract entities that exist only in textbooks and classroom discussions. They are actual electronic devices that serve as the building blocks in the design and construction of modern computer systems. The reason for using gates rather than transistors is that a transistor is too elementary a device to act as the fundamental design component. It requires a designer to deal with such low-level issues as currents, voltages, and the laws of physics. Transistors, grouped together to form more powerful building blocks called gates, allow us to think and design at a higher level. Instead of dealing with the complex physical rules associated with discrete electrical devices, we can use the power and expressiveness of mathematics and logic to build computers.

This seemingly minor shift (from transistors to gates) has a profound effect on how computer hardware is designed and built. From this point on in our discussion of hardware design, we no longer need deal with anything electrical. Instead, our building blocks are AND, OR, and NOT gates, and our circuit construction rules are the rules of Boolean logic. This is another example of the importance of abstraction in computer science.

George Boole (1815–1864)

George Boole was a mid-19th-century English mathematician, philosopher, and logician. He was the son of a shoemaker and had little formal education, having attended only elementary school. He taught himself mathematics and logic and mastered French, German, Italian, Latin, and Greek. He avidly studied the works of the great Greek and Roman philosophers such as Aristotle, Plato, and Euclid. He built on their work in logic, argumentation, and reasoning and, in 1854, produced a book titled *Introduction into the Laws of Thought*. This seminal work attempted to apply the formal laws of algebra and arithmetic to the principles of logic. That is, it treated reasoning as simply another branch of mathematics containing operators, variables, and transformation rules. He created a new form of logic containing the values *true* and *false* and the operators AND, OR, and NOT. He also developed a set of rules describing how to interpret and manipulate expressions that contain these values.

At the time of its development, the importance of this work was not apparent, and it languished in relative obscurity. However, 100 years later, Boole's ideas became the theoretical framework underlying the design of all computer systems. In his honor, these true/false expressions became known as *Boolean expressions*, and this branch of mathematics is called *Boolean logic* or *Boolean algebra*.

Even though he had very little formal schooling, Boole was eventually appointed professor of mathematics at Queens College in Cork, Ireland. He received a gold medal from the Royal Mathematical Society and in 1857 was awarded an honorary doctoral degree from Oxford University. He is now universally recognized as one of the greatest mathematicians of the 19th century.

4.4 Building Computer Circuits

4.4.1 Introduction

A **circuit** is a collection of logic gates that transforms a set of binary inputs into a set of binary outputs and in which the values of the outputs depend only on the current values of the inputs. (Actually, this type of circuit is more properly called a **combinational circuit**. We use the simpler term *circuit* in this discussion.) A circuit C with *m* binary inputs and *n* binary outputs is represented as shown in Figure 4.21.

Figure 4.21 Diagram of a typical computer circuit

Internally, the circuit shown in Figure 4.21 is constructed from the AND, OR, and NOT gates introduced in the previous section. (*Note*: We do not use the NAND and NOR gates diagrammed in Figure 4.19(a) and Figure 4.20(a), respectively.) These gates can be interconnected in any way so long as the connections do not violate the constraints on the proper number of inputs and outputs for each gate. Each AND and OR gate must have exactly two inputs and one output. (Multiple-input AND and OR gates do exist, but

we do not use them in our examples.) Each NOT gate must have exactly one input and one output. For example, Figure 4.22 diagrams a circuit with two inputs labeled *a* and *b* and two outputs labeled *c* and d. It contains one AND gate, one OR gate, and two NOT gates.

Figure 4.22 Circuit with two inputs and two outputs

There is a direct relationship between Boolean expressions and *circuit diagrams* of this type. Every Boolean expression can be represented pictorially as a circuit diagram, and every output value in a circuit diagram can be written as a Boolean expression. For example, in the diagram shown, the two output values labeled *c* and *d* are equivalent to the following two Boolean expressions:

The choice of which representation to use, a circuit diagram or a Boolean expression, depends on what we want to do. The pictorial view better allows us to visualize the overall structure of the circuit, and is often used during the design stage. A Boolean expression may be better for performing mathematical or logical operations, such as verification and optimization, on the circuit. We use both representations in the following sections.

The value appearing on any output line of a circuit can be determined if we know the current input values and the transformations produced by each logic gate. (*Note*: There are circuits, called **sequential circuits**, which contain feedback loops in which the output of a gate is fed back as input to an earlier gate. The output of these circuits depends not only on the current input values but also on *previous* inputs. These circuits are typically used to build memory units because, in a sense, they can "remember" inputs. We do not discuss sequential circuits here.)

In the previous example, if and , then the value on the *c* output line is 1, and the value on the *d* output line is 0. These values can be determined as shown in Figure 4.23.

Figure 4.23 Input values and output values

Note that it is perfectly permissible to "split" or "tap" a line and send its value to two different gates. Here the input value *b* was split and sent to two separate gates.

The next section presents an algorithm for designing and building circuits from the three fundamental gate types AND, OR, and NOT. This enables us to move to yet a higher level of abstraction. Instead of thinking in terms of transistors and electrical voltages (as in Section 4.2.4) or in terms of logic gates and truth values (as in Section 4.3.2), we can think and design in terms of circuits for high-level operations such as addition and comparison. This makes understanding computer hardware much more manageable.

4.4.2 A Circuit Construction Algorithm

The circuit shown at the end of the previous section is simply an example and is not meant to carry out any meaningful operation. To create circuits that perform useful arithmetic and logical functions, we need a way to convert a description of a circuit's

desired behavior into a circuit diagram, composed of AND, OR, and NOT gates, that does exactly what we want it to do.

There are a number of **circuit construction algorithms** to accomplish this task, and the remainder of this section describes one such technique, called the *sum-of-products algorithm*, that allows us to design circuits. Section 4.4.3 demonstrates how this algorithm works by constructing actual circuits that all computer systems need.

Step 1: Truth Table Construction. First, determine how the circuit should behave under all possible circumstances. That is, determine the binary value that should appear on each output line of the circuit for every possible combination of inputs. This information can be organized as a truth table. If a circuit has *N* input lines, and if each input line can be either a 0 or a 1, then there are combinations of input values, and the truth table has rows. For each output of the circuit, we must specify the desired output value for every row in the truth table.
For example, if a circuit has three inputs and two outputs, then a truth table for that circuit has input combinations and might look something like the following. (In this example, the output values are completely arbitrary.)

This circuit has two outputs labeled Output-1 and Output-2. The truth table specifies the value of each of these two output lines for every one of the eight possible combinations of inputs. We will use this example to illustrate the subsequent steps in the algorithm.

Step 2: Subexpression Construction Using AND and NOT Gates. Choose any one output column of the truth table built in Step 1 and scan down that column. Every place that you find a 1 in that output column, you build a Boolean *subexpression* that produces the value 1 (i.e., is true) for exactly that combination of input values and no other. To build this subexpression, you examine the value of each input for this specific case. If the input is a 1, use that input value directly in your subexpression. If the input is a 0, first take the NOT of that input, changing it from a 0 to a 1, and then use that *complemented* input value in your subexpression. You now have an input sequence of all 1s, and if all of these modified inputs are ANDed together (two at a time, of course), then the output value is a 1. For example, let's look at the output column labeled Output-1 in the truth table below.

There are two 1s in the column labeled Output-1; they are referred to as case 1 and case 2. We thus need to construct two subexpressions, one for each of these two cases.

In case 1, the inputs *a* and *c* have the value 0 and the input *b* has the value 1. Thus we apply the NOT operator to both *a* and *c*, changing them from 0 to 1. Because the value of *b* is 1, we can use *b* directly. We now have three modified input values, all of which have the value 1. ANDing these three values together yields the Boolean expression This expression produces a 1 only when the input is exactly , , . In any other case, at least one of the three factors in the expression is 0, and when the AND operation is carried out, it produces a 0. (Check this yourself by trying some other input values and seeing what is produced.) Thus the desired subexpression for case 1 is
The subexpression for case 2 is developed in an identical manner, and it results in

This subexpression produces a 1 only when the input is exactly , , .

Step 3: Subexpression Combination Using OR Gates. Take each of the subexpressions produced in Step 2 and combine them, two at a time, using OR gates. Each of the individual subexpressions produces a 1 for exactly one particular case where the truth table output is a 1, so the OR of the output of all of them produces a 1 in each case where the truth table has a 1 and in no other case. Consequently, the Boolean expression produced in Step 3 implements exactly the function described in the output column of the truth table on which we are working. In the current example, the final Boolean expression produced during Step 3 is

Step 4: Circuit Diagram Production. Construct the final circuit diagram. To do this, convert the Boolean expression produced at the end of Step 3 into a circuit diagram, using AND, OR, and NOT gates to implement the AND, OR, and NOT operators appearing in the Boolean expression. This circuit diagram produces the output described in the corresponding column of the truth table created in Step 1. The circuit diagram for the Boolean expression developed in Step 3 is shown in Figure 4.24.

Figure 4.24 Circuit diagram for the output labeled Output-1

We have successfully built the part of the circuit that produces the output for the column labeled Output-1 in the truth table shown in Step 1. We now repeat Steps 2, 3, and 4 for any additional output columns contained in the truth table. (In this example, there is a second column labeled Output-2. We leave the construction of that circuit as a practice exercise.) When we have constructed a circuit diagram for every output of the circuit, we are finished. The sum-of-products algorithm is summarized in Figure 4.25.

Figure 4.25 The sum-of-products circuit construction algorithm

This has been a formal introduction to one particular circuit construction algorithm. The algorithm is not easy to comprehend in an abstract sense. The next section clarifies this technique by using it to design two circuits that perform the operations of comparison and addition, respectively. Seeing it used to design actual circuits will make the steps of the algorithm easier to understand and follow.

We end this section by noting that the circuit construction algorithm just described does not always produce an optimal circuit, where *optimal* means that the circuit accomplishes its desired function using the smallest number of logic gates. For example, using the truth table shown in Step 2: Subexpression Construction Using AND and NOT Gates, our sum-of-products algorithm produced the seven-gate circuit shown in Figure 4.24. This is a correct answer in the sense that the circuit does produce the correct values for Output-1 for all combinations of inputs. However, it is possible to do much better.

The circuit shown in Figure 4.26 also produces the correct result using only two gates instead of seven. This difference is very important because each AND, OR, and NOT gate is a physical entity that costs real money, takes up space on the chip, requires power to operate, and generates heat that must be dissipated.

Figure 4.26 A more efficient circuit

Eliminating five unnecessary gates produces a real savings. The fewer gates we use, the cheaper, more efficient, and more compact are our circuits and hence the resulting computer. Algorithms for **circuit optimization**—that is, for reducing the number of gates needed to implement a circuit—are an important part of hardware design. Challenge Work problem 1 at the end of the chapter invites you to investigate this interesting topic in more detail.

4.4.3 Examples of Circuit Design and Construction

Let's use the algorithm described in Section 4.4.2 to construct two circuits important to the operation of any real-world computer: a compare-for-equality circuit and an addition circuit.

A Compare-for-Equality Circuit. The first circuit we will construct is a *compare-for-equality circuit*, or CE circuit, which tests two unsigned binary numbers for exact equality. The circuit produces the value 1 (*true*) if the two numbers are equal and the value 0 (*false*) if they are not. Such a circuit could be used in many situations. For example, in the shampooing algorithm in Chapter 1, Figure 1.3(a), there is an instruction that says,

Repeat Steps 4 through 6 until the value of *WashCount* equals 2

Our CE circuit could accomplish the comparison between the number currently stored in *WashCount* and the value 2 and return true or false, depending on whether these two values were equal or not equal.

Let's start by using the sum-of-products algorithm in Figure 4.25 to construct a simpler circuit called 1-CE, short for 1-bit compare for equality. A 1-CE circuit compares two 1-bit values a and b for equality. That is, the circuit 1-CE produces a 1 as output if both its inputs are 0 or both its inputs are 1. Otherwise, 1-CE produces a 0. After designing 1-CE, we will use it to create a full-blown comparison circuit that can handle numbers of any size.

Step 1 of the algorithm says to construct the truth table that describes the behavior of the desired circuit. The truth table for the 1-CE circuit is

In the output column of the truth table, there are two 1 values, labeled case 1 and case 2, so Step 2 of the algorithm is to construct two subexpressions, one for each of these two cases. The subexpression for case 1 is because this produces the value 1 only when and . The subexpression for case 2 is $(a \cdot b)$, which produces a 1 only when and . We now combine the outputs of these two subexpressions with an OR gate, as described in Step 3, to produce the Boolean expression

Finally, in Step 4, we convert this expression to a circuit diagram, which is shown in Figure 4.27. The circuit shown in Figure 4.27 correctly compares two 1-bit quantities and determines if they are equal. If they are equal, it outputs a 1. If they are unequal, it outputs a 0.

Figure 4.27 One-bit compare-for-equality circuit

However, the numbers compared for equality by a computer are usually much larger than a single binary digit. We want a circuit that correctly compares two numbers that contain *N* binary digits. To build this "*N*-bit compare-for-equality" circuit, we use *N* of the 1-CE circuits shown in Figure 4.27, one for each bit position in the numbers to be compared. Each 1-CE circuit produces a 1 if the two binary digits in its specific location are identical and produces a 0 if they are not. If every circuit produces a 1, then the two numbers are identical in every bit position, and they are equal. To check whether all our 1-CE circuits produce a 1, we simply AND together (two at a time) the outputs of all *N* 1-CE circuits. Remember that an AND gate produces a 1 if and only if both of its inputs are a 1. Thus the final output of the *N*-bit compare circuit is a 1 if and only if every pair of bits in the corresponding location of the two numbers is identical—that is, the two numbers are equal.

Figure 4.28 shows the design of a complete *N-bit compare-for-equality circuit* called CE. Each of the two numbers being compared, *a* and *b*, contains *N* bits, and they are labeled and . The box labeled 1-CE in Figure 4.28 is the 1-bit compare-for-equality circuit shown in Figure 4.27. Looking at these figures, you can see that we have designed a very complex electrical circuit without the specification of a single electrical device. The only "devices" in those diagrams are gates to implement the logical operations AND, OR, and NOT, and the only "rules" we need to know to understand the diagrams are the transformation rules of Boolean logic. George Boole's work in algebraic logic from the 1850s is now the starting point for the design of every circuit found inside a modern 21st-century computer.

Figure 4.28 *N*-bit compare-for-equality circuit

An Addition Circuit. Our second example of circuit construction is an addition circuit called ADD that performs binary addition on two unsigned *N*-bit integers. Typically, this type of circuit is called a *full adder*. For example, assuming , our ADD circuit would be able to perform the following 6-bit addition operation:

Just as we did with the CE circuit, we carry out the design of the ADD circuit in two stages. First, we use the circuit construction algorithm of Figure 4.25 to build a circuit called 1-ADD that adds a single pair of binary digits, along with a carry digit. We then interconnect *N* of these 1-ADD circuits to produce the complete *N*-bit full adder circuit ADD.

Looking at the addition example just shown, we see that summing the values in any column *i* requires us to add three binary values—the two binary digits in that column, and , and the carry digit from the previous column, called . Furthermore, the circuit must produce two binary outputs: a sum digit and a new carry digit that propagates to the next column. The pictorial representation of the 1-bit adder circuit 1-ADD and its accompanying truth table are shown in Figure 4.29.

Figure 4.29 The 1-ADD circuit and truth table

Because the 1-ADD circuit being constructed has two outputs, and , we must use Steps 2, 3, and 4 of the circuit construction algorithm twice, once for each output. Let's work on the sum output first.

The output column of Figure 4.29 contains four 1s, so we need to construct four subexpressions. In accordance with the guidelines given in Step 2 of the construction algorithm, these four subexpressions are

- Case 1:
- Case 2:
- Case 3:
- Case 4:

Step 3 says to combine the outputs of these four subexpressions using three OR gates to produce the output labeled in the truth table of Figure 4.29. The final Boolean expression for the sum output is

The logic circuit to produce the output whose expression is given above is shown in Figure 4.30. (This circuit diagram has been labeled to highlight the four separate subexpressions created during Step 2, as well as the combining of the subexpressions in Step 3 of the construction algorithm.)

Figure 4.30 Sum output for the 1-ADD circuit

We are not yet finished, because the 1-ADD circuit in Figure 4.29 has a second output—the carry into the next column. That means the circuit construction algorithm must be repeated for the second output column, labeled .

The column also contains four 1s, so we again need to build four separate subcircuits, just as for the sum output, and combine them using OR gates. The construction proceeds in a fashion similar to the first part, so we leave the details as an exercise for the reader.

The Boolean expression describing the carry output of the 1-ADD circuit is

We have now built the two parts of the 1-ADD circuit that produce the sum and the carry outputs. The complete 1-ADD circuit is constructed by simply putting these two pieces together. Figure 4.31 shows the complete (and admittedly quite complex) 1-ADD circuit to implement 1-bit addition. To keep the diagram from becoming an incomprehensible tangle of lines, we have drawn it in a slightly different orientation from Figures 4.27 and 4.30. Everything else is exactly the same.

Figure 4.31 Complete 1-ADD circuit for 1-bit binary addition

When looking at this rather imposing diagram, do not become overly concerned with the details of every gate, every connection, and every operation. Figure 4.31 more importantly illustrates the *process* by which we design such a complex and intricate circuit: by transforming the idea of 1-bit binary addition into an electrical circuit using the tools of algorithmic problem solving and symbolic logic.

How is the 1-ADD circuit shown in Figure 4.31 used to add numbers that contain *N* binary digits rather than just one? The answer is simple if we think about the way numbers are added by hand. (We discussed exactly this topic when developing the addition algorithm of Figure 1.2 in Chapter 1.) We add numbers one column at a time, moving from right to left, generating the sum digit, writing it down, and sending any carry to the next column. The same thing can be done in hardware. We use *N* of the 1-ADD circuits shown in Figure 4.31, one for each column. Starting with the rightmost circuit, each 1-ADD circuit adds a single column of digits, generates a sum digit that is part of the final answer, and sends its carry digit to the 1-ADD circuit on its left, which replicates this process. After *N* repetitions of this process, all sum digits have been generated, and the *N* circuits have correctly added the two numbers.

The complete full adder circuit called ADD is shown in Figure 4.32. It adds the two *N*-bit numbers , … and to produce the (*N* + 1)-bit sum . Because addition is one of the most common arithmetic operations, the circuit shown in Figure 4.32 (or something equivalent) is one of the most important and most frequently used arithmetic components. Addition circuits are found in every computer, tablet, smartphone, and handheld calculator in the marketplace. They are even found in computer-controlled thermostats, clocks, and microwave ovens, where they enable us, for example, to add 30 minutes to the cooking time.

Figure 4.32 The complete full adder ADD circuit

Figure 4.32 is, in a sense, the direct hardware implementation of the addition algorithm shown in Figure 1.2. Although Figure 1.2 and Figure 4.32 are quite different, both represent essentially the same algorithm: the column-by-column addition of two *N*-bit numerical values. This demonstrates quite clearly that there are many different ways to express the same algorithm—in this case, pseudocode (Figure 1.2) and hardware circuits (Figure 4.32). Later chapters show additional ways to represent algorithms, such as machine language programs and high-level language programs. However, regardless of whether we use English, pseudocode, mathematics, or transistors to describe an algorithm, its fundamental properties are the same, and the central purpose of computer science—algorithmic problem solving—remains the same.

It may also be instructive to study the size and complexity of the ADD circuit just designed. Figure 4.32 shows that the addition of two *N*-bit integer values requires *N* separate 1-ADD circuits. Let's assume that , a typical value for modern computers. Referring to Figure 4.31, we see that each 1-ADD circuit uses 3 NOT gates, 16 AND gates, and 6 OR gates, a total of 25 logic gates. Thus the total number of logic gates used to implement 32-bit binary addition is gates. Figure 4.18 shows that each NOT gate requires one transistor, and Figures 4.19(b) and 4.20(b) show that each AND and OR gate requires three transistors. Therefore, more than 2,200 transistors are needed to build a 32-bit adder circuit:

(*Note*: Optimized 32-bit addition circuits can be constructed using as few as 500–600 transistors. However, this does not change the fact that it takes many, many transistors to accomplish this addition task.)

Dr. William Shockley (1910–1989)

Dr. William Shockley was the inventor (along with John Bardeen and Walter Brattain) of the transistor. His discovery has probably done as much to shape our modern world as any scientific advancement of the 20th century. He received the 1956 Nobel Prize in Physics and, at his death, was a distinguished professor at Stanford University.

Shockley and his team developed the transistor in 1947 while working at Bell Laboratories. He left there in 1954 to set up the Shockley Semiconductor Laboratory in California—a company that was instrumental in the birth of the high-technology region called Silicon Valley. The employees of this company eventually went on to develop other fundamental advances in computing, such as the integrated circuit and the microprocessor and started a number of important high-technology companies, including Fairchild Semiconductor, Advanced Micro Devices (AMD), and Intel.

Although Shockley's work has been compared with that of Pasteur, Salk, and Einstein in importance, his reputation and place in history have been forever tarnished by his outrageous and controversial racial theories. His education and training were in physics and electrical engineering, but Shockley spent the last years of his life trying to convince people of the genetic inferiority of black people, even though he was ridiculed and shunned by colleagues who abandoned all contact with him. By the time of his death in 1989 he had become estranged from virtually all his friends and family. It is said that his children learned about his death only through stories in the newspaper. His intense racial bigotry prevented him from receiving the recognition that would otherwise have been his for monumental contributions in physics, engineering, and computer science.

This computation emphasizes the importance of the continuing research into the miniaturization of electrical components. (See the Special Interest Box on Moore's Lawearlier in this chapter.) If vacuum tubes were used instead of transistors, as was done in computers from about 1940 to 1955, the adder circuit shown in Figure 4.32 would be extraordinarily bulky; 2,208 vacuum tubes would occupy a space about the size of a large refrigerator. It would also generate huge amounts of heat, necessitating sophisticated cooling systems, and it would be very difficult to maintain. (Imagine the time it would take to locate a single burned-out vacuum tube from a cluster of 2,000.) Using something on the scale of the magnetic core technology described in Section 4.2.4 and shown in Figure 4.9, the adder circuit would fit into an area a few inches square. However, modern circuit technology can now achieve transistor densities greater than . At this level, the entire ADD circuit of Figure 4.32 would easily fit in an area much, much smaller than the size of the period at the end of this sentence. That is why it is now possible to put powerful computer-processing facilities not only in a room or on a desk but also inside a watch, a thermostat, or even inside the human body.

4.5 Control Circuits

The previous section described the design of circuits for implementing arithmetic and logical operations. However, there are other, quite different, types of circuits that are also essential to the proper functioning of a computer system. This section briefly describes one of these important circuit types, **control circuits**, which are used not to implement arithmetic operations but to determine the order in which operations are carried out and to select the correct data values to be processed. In a sense, they are the sequencing and decision-making circuits inside a computer. These circuits are essential to the proper function of a computer because, as we noted in Chapter 1, algorithms and

programs must be well ordered and must always know which operation to do next. The two major types of control circuits are called *multiplexers* and *decoders*, and, like everything else described in this chapter, they can be completely described in terms of gates and the rules of logic.

A **multiplexer** is a circuit that has 2^n *input lines* and 1 *output line*. Its function is to select exactly one of its input lines and copy the binary value on that input line onto its single output line. A multiplexer chooses one specific input by using an additional set of *N* lines called *selector lines*. (Thus the total number of inputs to the multiplexer circuit is .) The input lines of a multiplexer are numbered . Each of the *N* selector lines can be set to either a 0 or a 1, so we can use the *N* selector lines to represent all binary values from 000 … 0 (*N* zeros) to 111 … 1 (*N* ones), which represent all integer values from 0 to . These numbers correspond exactly to the numbers of the input lines. Thus the binary number that appears on the selector lines can be interpreted as the identification number of the input line that is to be selected. Pictorially, a multiplexer looks like the diagram in Figure 4.33.

Figure 4.33 Multiplexer circuit

For example, if we had four input lines (i.e.,) coming into our multiplexer, numbered 0, 1, 2, and 3, then we would need two selector lines. The four binary combinations that can appear on this pair of selector lines are 00, 01, 10, and 11, which correspond to the decimal values 0, 1, 2, and 3, respectively (refer to Figure 4.2). The multiplexer selects the one input line whose identification number corresponds to the value appearing on the selector lines and copies the value on that input line to the output line. If, for example, the two selector lines were set to 1 and 0, then a multiplexer circuit would pick input line 2 because 10 in binary is 2 in decimal notation.
Implementing a multiplexer using logic gates is not difficult. Figure 4.34 shows a simple multiplexer circuit with . This is a multiplexer with two input lines and a single selector line.

Figure 4.34 A two-input multiplexer circuit

In Figure 4.34 if the value on the selector line is 0, then the bottom input line to AND gate 2 is always 0, so its output is always 0. Looking at AND gate 1, we see that the NOT gate changes its bottom input value to a 1. Because (1 AND *a*) is always *a*, the output of the top AND gate is equal to the value of *a*, which is the value of the input from line 0. Thus the two inputs to the OR gate are 0 and *a*. Because the value of the expression (0 OR *a*) is identical to *a*, by setting the selector line to 0 we have, in effect, selected as our output the value that appears on line 0. You should confirm that if the selector line has the value 1, then the output of the circuit in Figure 4.34 is *b*, the value appearing on line 1. We can design multiplexers with more than two inputs in a similar fashion, although they rapidly become more complex.

The second type of control circuit is called a **decoder** (Figure 4.35) and it operates in the opposite way from a multiplexer. A decoder has *N* input lines numbered and output lines numbered .

Figure 4.35 Decoder circuit

Each of the *N* input lines of the decoder can be set to either a 0 or a 1, and when these *N*values are interpreted as a single binary number, they can represent all integer values from 0 to . It is the job of the decoder to determine the value represented on its *N* input lines and then send a signal (i.e., a 1) on the single output line that has that identification number. All other output lines are set to 0.

For example, if our decoder has three input lines, it has eight output lines numbered 0 to 7. These three input lines can represent all binary values from 000 to 111, which is from 0 to 7 in decimal notation. If, for example, the binary values on the three input lines are 101, which is a 5, then a signal (a binary 1) would be sent out by the decoder on output line 5. All other output lines would contain a 0.

Figure 4.36 shows the design of a 2-to-4 decoder circuit with two input lines and four output lines. These four output lines are labeled 0, 1, 2, and 3, and the only output line that carries a signal value of 1 is the line whose identification number is identical to the value appearing on the two input lines. For example, if the two inputs are 11, then line 3 should be set to a 1 (11 in binary is 3 in decimal). This is, in fact, what happens because the AND gate connected to line 3 is the only one whose two inputs are equal to a 1. You should confirm that this circuit behaves properly when it receives the inputs 00, 01, and 10 as well.

Figure 4.36A 2-to-4 decoder circuit

Together, decoder and multiplexer circuits enable us to build computer systems that execute the correct instructions using the correct data values. For example, assume we have a computer that can carry out four different types of arithmetic operations—add, subtract, multiply, and divide. Furthermore, assume that these four instructions have code numbers 0, 1, 2, and 3, respectively. We could use a decoder circuit to ensure that the computer performs the correct instruction. We need a decoder circuit with two input lines. It receives as input the two-digit code number (in binary) of the instruction that we want to perform: 00 (add), 01 (subtract), 10 (multiply), or 11 (divide). The decoder interprets this value and sends out a signal on the correct output line. This signal is used to select the proper arithmetic circuit and cause it to perform the desired operation. This behavior is diagrammed in Figure 4.37.

Figure 4.37Example of the use of a decoder circuit

Whereas a decoder circuit can be used to select the correct instruction, a multiplexer can help ensure that the computer executes this instruction using the correct data. For example, suppose our computer has four special registers called R0, R1, R2, and R3. (For now, just consider a register to be a place to store a data value. We describe registers in more detail in the next chapter.) Assume that we have built a circuit called *test-if-zero* that can test whether any of these four registers contains the value 0. (This is actually quite similar to the CE circuit of Figure 4.28.) We can use a multiplexer circuit to select the register that we want to send to the test-if-zero circuit. This is shown in Figure 4.38. If we want to test if register R2 in Figure 4.38 is 0, we simply put the binary value 10 (2 in decimal notation) on the two selector lines. This selects register R2, and only its value passes through the multiplexer and is sent to the test circuit.

Figure 4.38Example of the use of a multiplexer circuit

There are many more examples of the use of control circuits in Chapter 5, which examines the execution of programs and the overall organization of a computer system.

4.6 Conclusion

We began our discussion on the representation of information and the design of computer circuits with the most elementary component, bistable electronic devices such as transistors. We showed how they can be used to construct logic gates that in turn can be used to implement circuits to carry out useful functions. Our purpose here was not to make you an expert in specifying and designing computer circuits but to demonstrate how it is possible to implement high-level arithmetic operations using only low-level electronic components such as transistors. We also demonstrated how it is possible to reorient our viewpoint and raise our level of abstraction. We changed the level of discussion from electricity to arithmetic, from hardware devices to mathematical behavior, from form to function. This is one of the first steps up the hierarchy of abstractions introduced in Figure 1.9.

Chapter 5 continues this "upward climb" to yet higher levels of abstraction. It shows how arithmetic circuits, such as compare-for-equality and addition (Section 4.4.3), and control circuits, such as multiplexers and decoders (Section 4.5), can be used to construct entire computer systems.

After reading this chapter, you might have the feeling that although you understand the individual concepts that are covered, you still don't understand, in the grand sense, what computers are or how they work. You might feel that you can follow the details but can't see the "big picture." One possible reason is that this chapter looks at computers from a very elementary viewpoint, by studying different types of specialized circuits. This is analogous to studying the human body as a collection of millions of cells of different types—blood cells, brain cells, skin cells, and so on. Cytology (the study of cells) is certainly an important part of the field of biology, but understanding only the cellular structure of the human body provides no intuitive understanding of what people are and how we do such characteristic things as walk, eat, and breathe. Understanding these complex actions derives not from a study of molecules, genes, or cells but from a study of higher-level organs and their interactions, such as the lungs, heart, and muscles.

That is exactly what happens in Chapter 5, in which we examine higher-level computer components such as processors, memory, and instructions, and begin our study of the topic of computer organization.