There are 10 kinds of people in the world—those who understand binary and those who don't.

—Anonymous

CHAPTER 2

Data Representation in Computer Systems

2.1 INTRODUCTION

The organization of any computer depends considerably on how it represents numbers, characters, and control information. The converse is also true: Standards and conventions established over the years have determined certain aspects of computer organization. This chapter describes the various ways in which computers can store and manipulate numbers and characters. The ideas presented in the following sections form the basis for understanding the organization and function of all types of digital systems.

The most basic unit of information in a digital computer is called a **bit**, which is a contraction of **binary digit**. In the concrete sense, a bit is nothing more than a state of "on" or "off " (or "high" and "low") within a computer circuit. In 1964, the designers of the IBM System/360 mainframe computer established a convention of using groups of 8 bits as the basic unit of addressable computer storage. They called this collection of 8 bits a **byte**.

Computer **words** consist of two or more adjacent bytes that are sometimes addressed and almost always are manipulated collectively. The **word size** represents the data size that is handled most efficiently by a particular architecture. Words can be 16 bits, 32 bits, 64 bits, or any other size that makes sense in the context of a computer's organization (including sizes that are not multiples of eight). An 8-bit byte can be divided into two 4-bit halves called **nibbles** (or **nybbles**). Because each bit of a byte has a value within a positional numbering system, the nibble containing the least-valued binary digit is called the *low-order nibble*, and the other half the *high-order nibble*.

2.2 POSITIONAL NUMBERING SYSTEMS

At some point during the middle of the sixteenth century, Europe embraced the decimal (or base 10) numbering system that the Arabs and Hindus had been using for nearly a millennium. Today, we take for granted that the number 243 means two hundreds, plus four tens, plus three units. Notwithstanding the fact that zero means "nothing," virtually everyone knows that there is a substantial difference between having 1 of something and having 10 of something.

The general idea behind positional numbering systems is that a numeric value is represented through increasing powers of a **radix** (or base). This is often referred to as a **weighted numbering system** because each position is weighted by a power of the radix.

The set of valid numerals for a positional numbering system is equal in size to the radix of that system. For example, there are 10 digits in the decimal system, 0 through 9, and 3 digits for the ternary (base 3) system, 0, 1, and 2. The largest valid number in a radix system is one smaller than the radix, so 8 is not a valid numeral in any radix system smaller than 9. To distinguish among numbers in different radices, we use the radix as a subscript, such as in 33_{10} to represent the decimal number 33. (In this text, numbers written without a subscript should be assumed to be decimal.) Any decimal integer can be expressed exactly in any other integral base system (see Example 2.1).

EXAMPLE 2.1 Three numbers are represented as powers of a radix.

$$243.51_{10} = 2 \times 10^2 + 4 \times 10^1 + 3 \times 10^0 + 5 \times 10^{-1} + 1 \times 10^{-2}$$
$$212_3 = 2 \times 3^2 + 1 \times 3^1 + 2 \times 3^0 = 23_{10}$$
$$10110_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 22_{10}$$

The two most important radices in computer science are binary (base two) and hexadecimal (base 16). Another radix of interest is octal (base 8). The binary system uses only the digits 0 and 1; the octal system, 0 through 7. The hexadecimal system allows the digits 0 through 9 with A, B, C, D, E, and F being used to represent the numbers 10 through 15. Table 2.1 shows some of the radices.

2.3 CONVERTING BETWEEN BASES

Gottfried Leibniz (1646–1716) was the first to generalize the idea of the (positional) decimal system to other bases. Being a deeply spiritual person, Leibniz attributed divine qualities to the binary system. He correlated the fact that any integer could be represented by a series of 1s and 0s with the idea that God (1) created the universe out of nothing (0). Until the first binary digital computers were built in the late 1940s, this system remained nothing more than a

Powers of 2
$2^{-2} = \frac{1}{4} = 0.25$
$2^{-1} = \frac{1}{2} = 0.5$
$2^{0} = 1$
$2^{1} = 2$
$2^{2} = 4$
$2^{3} = 8$
$2^{4} = 16$
$2^{5} = 32$
$2^{6} = 64$
$2^{7} = 128$
$2^{8} = 256$
$2^{9} = 512$
$2^{10} = 1024$
$2^{15} = 32,768$
$2^{16} = 65,536$

Decimal	4-Bit Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

TABLE 2.1 Some Numbers to Remember

mathematical curiosity. Today, it lies at the heart of virtually every electronic device that relies on digital controls.

Because of its simplicity, the binary numbering system translates easily into electronic circuitry. It is also easy for humans to understand. Experienced computer professionals can recognize smaller binary numbers (such as those shown in Table 2.1) at a glance. Converting larger values and fractions, however, usually requires a calculator or pencil and paper. Fortunately, the conversion techniques are easy to master with a little practice. We show a few of the simpler techniques in the sections that follow.

2.3.1 Converting Unsigned Whole Numbers

We begin with the base conversion of unsigned numbers. Conversion of signed numbers (numbers that can be positive or negative) is more complex, and it is important that you first understand the basic technique for conversion before continuing with signed numbers.

Conversion between base systems can be done by using either repeated subtraction or a division-remainder method. The subtraction method is cumbersome and requires a familiarity with the powers of the radix being used. Because it is the more intuitive of the two methods, however, we will explain it first.

As an example, let's say we want to convert 538_{10} to base 8. We know that $8^3 = 512$ is the highest power of 8 that is less than 538, so our base 8 number will be 4 digits wide (one for each power of the radix: 0 through 3). We make note that 512 goes once into 538 and subtract, leaving a difference of 26.

We know that the next power of 8, $8^2 = 64$, is too large to subtract, so we note the zero "placeholder" and look for how many times $8^1 = 8$ divides 26. We see that it goes three times and subtract 24. We are left with 2, which is 2×8^0. These steps are shown in Example 2.2.

EXAMPLE 2.2 Convert 538_{10} to base 8 using subtraction.

$$\begin{array}{rl} 538 & \\ -\ 512 & = 8^3 \times 1 \\ \hline 26 & \\ -\ 0 & = 8^2 \times 0 \\ \hline 26 & \\ -\ 24 & = 8^1 \times 3 \\ \hline 2 & \\ -\ 2 & = 8^0 \times 2 \\ \hline 0 & \end{array}$$

$538_{10} = 1032_8$

The division-remainder method is faster and easier than the repeated subtraction method. It employs the idea that successive divisions by the base are in fact successive subtractions by powers of the base. The remainders that we get when we sequentially divide by the base end up being the digits of the result, which are read from bottom to top. This method is illustrated in Example 2.3.

EXAMPLE 2.3 Convert 538_{10} to base 8 using the division-remainder method.

$8\overline{\vert 538}$	2	8 divides 538 67 times with a remainder of 2.
$8\overline{\vert 67}$	3	8 divides 67 8 times with a remainder of 3.
$8\overline{\vert 8}$	0	8 divides 8 1 time with a remainder of 0.
$8\overline{\vert 1}$	1	8 divides 1 0 times with a remainder of 1.
0		

Reading the remainders from *bottom to top*, we have: $538_{10} = 1032_8$.

This method works with any base, and because of the simplicity of the calculations, it is particularly useful in converting from decimal to binary. Example 2.4 shows such a conversion.

EXAMPLE 2.4 Convert 147_{10} to binary.

$2\lfloor\underline{147}$	1	2 divides 147 73 times with a remainder of 1.
$2\lfloor\underline{73}$	1	2 divides 73 36 times with a remainder of 1.
$2\lfloor\underline{36}$	0	2 divides 36 18 times with a remainder of 0.
$2\lfloor\underline{18}$	0	2 divides 18 9 times with a remainder of 0.
$2\lfloor\underline{9}$	1	2 divides 9 4 times with a remainder of 1.
$2\lfloor\underline{4}$	0	2 divides 4 2 times with a remainder of 0.
$2\lfloor\underline{2}$	0	2 divides 2 1 time with a remainder of 0.
$2\lfloor\underline{1}$	1	2 divides 1 0 times with a remainder of 1.
0		

Reading the remainders from bottom to top, we have: $147_{10} = 10010011_2$.

A binary number with N bits can represent unsigned integers from 0 to $2^N - 1$. For example, 4 bits can represent the decimal values 0 through 15, whereas 8 bits can represent the values 0 through 255. The range of values that can be represented by a given number of bits is extremely important when doing arithmetic operations on binary numbers. Consider a situation in which binary numbers are 4 bits in length, and we wish to add 1111_2 (15_{10}) to 1111_2. We know that 15 plus 15 is 30, but 30 cannot be represented using only 4 bits. This is an example of a condition known as **overflow**, which occurs in unsigned binary representation when the result of an arithmetic operation is outside the range of allowable precision for the given number of bits. We address overflow in more detail when discussing signed numbers in Section 2.4.

2.3.2 Converting Fractions

Fractions in any base system can be approximated in any other base system using negative powers of a radix. **Radix points** separate the integer part of a number from its fractional part. In the decimal system, the radix point is called a *decimal point*. Binary fractions have a binary point.

Fractions that contain repeating strings of digits to the right of the radix point in one base may not necessarily have a repeating sequence of digits in another base. For instance, $\frac{2}{3}$ is a repeating decimal fraction, but in the ternary system, it terminates as $0.2_3 (2 \times 3^{-1} = 2 \times \frac{1}{3})$.

We can convert fractions between different bases using methods analogous to the repeated subtraction and division-remainder methods for converting integers. Example 2.5 shows how we can use repeated subtraction to convert a number from decimal to base 5.

EXAMPLE 2.5 Convert 0.4304_{10} to base 5.

$$\begin{array}{rl} 0.4304 & \\ -\ 0.4000 & = 5^{-1} \times 2 \\ \hline 0.0304 & \\ -\ 0.0000 & = 5^{-2} \times 0 \quad \text{(A placeholder)} \\ \hline 0.0304 & \\ -\ 0.0240 & = 5^{-3} \times 3 \\ \hline 0.0064 & \\ -\ 0.0064 & = 5^{-4} \times 4 \\ \hline 0.0000 & \end{array}$$

Reading from top to bottom, we have: $0.4304_{10} = 0.2034_5$.

Because the remainder method works with positive powers of the radix for conversion of integers, it stands to reason that we would use multiplication to convert fractions, because they are expressed in negative powers of the radix. However, instead of looking for remainders, as we did above, we use only the integer part of the product after multiplication by the radix. The answer is read from top to bottom instead of bottom to top. Example 2.6 illustrates the process.

EXAMPLE 2.6 Convert 0.4304_{10} to base 5.

$$\begin{array}{rl} .4304 & \\ \times\quad 5 & \\ \hline 2.1520 & \text{The integer part is 2. Omit from subsequent multiplication.} \\ .1520 & \\ \times\quad 5 & \\ \hline 0.7600 & \text{The integer part is 0. We'll need it as a placeholder.} \\ .7600 & \\ \times\quad 5 & \\ \hline 3.8000 & \text{The integer part is 3. Omit from subsequent multiplication.} \\ .8000 & \\ \times\quad 5 & \\ \hline 4.0000 & \text{The fractional part is now zero, so we are done.} \end{array}$$

Reading from top to bottom, we have $0.4304_{10} = 0.2034_5$.

This example was contrived so that the process would stop after a few steps. Often things don't work out quite so evenly, and we end up with repeating fractions. Most computer systems implement specialized rounding algorithms to

provide a predictable degree of accuracy. For the sake of clarity, however, we will simply discard (or truncate) our answer when the desired accuracy has been achieved, as shown in Example 2.7.

EXAMPLE 2.7 Convert 0.34375_{10} to binary with 4 bits to the right of the binary point.

$$\begin{array}{rl} .34375 & \\ \times \quad 2 & \\ \hline 0.68750 & \text{(Another placeholder)} \\ .68750 & \\ \times \quad 2 & \\ \hline 1.37500 & \\ .37500 & \\ \times \quad 2 & \\ \hline 0.75000 & \\ .75000 & \\ \times \quad 2 & \\ \hline 1.50000 & \text{(This is our fourth bit. We will stop here.)} \end{array}$$

Reading from top to bottom, $0.34375_{10} = 0.0101_2$ to four binary places.

The methods just described can be used to directly convert any number in any base to any other base, say from base 4 to base 3 (as in Example 2.8). However, in most cases, it is faster and more accurate to first convert to base 10 and then to the desired base. One exception to this rule is when you are working between bases that are powers of two, as you'll see in the next section.

EXAMPLE 2.8 Convert 3121_4 to base 3.

First, convert to decimal:

$$\begin{aligned} 3121_4 &= 3 \times 4^3 + 1 \times 4^2 + 2 \times 4^1 + 1 \times 4^0 \\ &= 3 \times 64 + 1 \times 16 + 2 \times 4 + 1 = 217_{10} \end{aligned}$$

Then convert to base 3:

$$\begin{array}{rl} 3\overline{)217} & 1 \\ 3\overline{)72} & 0 \\ 3\overline{)24} & 0 \\ 3\overline{)8} & 2 \\ 3\overline{)2} & 2 \\ 0 & \text{We have } 3121_4 = 22{,}001_3. \end{array}$$

2.3.3 Converting Between Power-of-Two Radices

Binary numbers are often expressed in hexadecimal—and sometimes octal—to improve their readability. Because $16 = 2^4$, a group of 4 bits (called a **hextet**) is easily recognized as a hexadecimal digit. Similarly, with $8 = 2^3$, a group of 3 bits (called an **octet**) is expressible as one octal digit. Using these relationships, we can therefore convert a number from binary to octal or hexadecimal by doing little more than looking at it.

EXAMPLE 2.9 Convert 110010011101_2 to octal and hexadecimal.

110	010	011	101	Separate into groups of 3 bits for the octal conversion.
6	2	3	5	

$110010011101_2 = 6235_8$

1100	1001	1101	Separate into groups of 4 for the hexadecimal conversion.
C	9	D	

$110010011101_2 = C9D_{16}$

If there are too few bits, leading 0s can be added.

2.4 SIGNED INTEGER REPRESENTATION

We have seen how to convert an unsigned integer from one base to another. Signed numbers require that additional issues be addressed. When an integer variable is declared in a program, many programming languages automatically allocate a storage area that includes a sign as the first bit of the storage location. By convention, a 1 in the high-order bit indicates a negative number. The storage location can be as small as an 8-bit byte or as large as several words, depending on the programming language and the computer system. The remaining bits (after the sign bit) are used to represent the number itself.

How this number is represented depends on the method used. There are three commonly used approaches. The most intuitive method, signed magnitude, uses the remaining bits to represent the magnitude of the number. This method and the other two approaches, which both use the concept of **complements**, are introduced in the following sections.

2.4.1 Signed Magnitude

Up to this point, we have ignored the possibility of binary representations for negative numbers. The set of positive and negative integers is referred to as the set of **signed integers**. The problem with representing signed integers as binary values is the sign—how should we encode the actual sign of the number? **Signed-magnitude representation** is one method of solving this problem. As its name implies, a signed-magnitude number has a sign as its leftmost bit (also referred

to as the *high-order* bit or the *most significant* bit), whereas the remaining bits represent the magnitude (or absolute value) of the numeric value. For example, in an 8-bit word, −1 would be represented as 10000001, and +1 as 00000001. In a computer system that uses signed-magnitude representation and 8 bits to store integers, 7 bits can be used for the actual representation of the magnitude of the number. This means that the largest integer an 8-bit word can represent is $2^7 - 1$, or 127 (a 0 in the high-order bit, followed by 7 1s). The smallest integer is 8 1s, or −127. Therefore, N bits can represent $-(2^{(N-1)} - 1)$ to $2^{(N-1)} - 1$.

Computers must be able to perform arithmetic calculations on integers that are represented using this notation. Signed-magnitude arithmetic is carried out using essentially the same methods that humans use with pencil and paper, but it can get confusing very quickly. As an example, consider the rules for addition: (1) If the signs are the same, add the magnitudes and use that same sign for the result; (2) If the signs differ, you must determine which operand has the larger magnitude. The sign of the result is the same as the sign of the operand with the larger magnitude, and the magnitude must be obtained by subtracting (not adding) the smaller one from the larger one. If you consider these rules carefully, this is the method you use for signed arithmetic by hand.

We arrange the operands in a certain way based on their signs, perform the calculation without regard to the signs, and then supply the sign as appropriate when the calculation is complete. When modeling this idea in an 8-bit word, we must be careful to include only 7 bits in the magnitude of the answer, discarding any carries that take place over the high-order bit.

EXAMPLE 2.10 Add 01001111_2 to 00100011_2 using signed-magnitude arithmetic.

				1	1	1	1		⇐ carries
0		1	0	0	1	1	1	1	(79)
0	+	0	1	0	0	0	1	1	+ (35)
0		1	1	1	0	0	1	0	(114)

The arithmetic proceeds just as in decimal addition, including the carries, until we get to the seventh bit from the right. If there is a carry here, we say that we have an overflow condition and the carry is discarded, resulting in an incorrect sum. There is no overflow in this example.

We find that $01001111_2 + 00100011_2 = 01110010_2$ in signed-magnitude representation.

Sign bits are segregated because they are relevant only after the addition is complete. In this case, we have the sum of two positive numbers, which is positive. **Overflow** (and thus an erroneous result) in signed numbers occurs when the sign of the result is incorrect.

In signed magnitude, the sign bit is used only for the sign, so we can't "carry into" it. If there is a carry emitting from the seventh bit, our result will be truncated as the seventh bit overflows, giving an incorrect sum. (Example 2.11 illustrates this overflow condition.) Prudent programmers avoid "million-dollar" mistakes by checking for overflow conditions whenever there is the slightest possibility they could occur. If we did not discard the overflow bit, it would carry into the sign, causing the more outrageous result of the sum of two positive numbers being negative. (Imagine what would happen if the next step in a program were to take the square root or log of that result!)

EXAMPLE 2.11 Add 01001111_2 to 01100011_2 using signed-magnitude arithmetic.

Last carry overflows and is discarded.

1	←			1	1	1	1		⇐ carries
0		1	0	0	1	1	1	1	(79)
0	+	1	1	0	0	0	1	1	+ (99)
0		0	1	1	0	0	1	0	(50)

We obtain the erroneous result of $79 + 99 = 50$.

Dabbling on the Double

The fastest way to convert a binary number to decimal is a method called **double-dabble** (or **double-dibble**). This method builds on the idea that a subsequent power of two is double the previous power of two in a binary number. The calculation starts with the leftmost bit and works toward the rightmost bit. The first bit is doubled and added to the next bit. This sum is then doubled and added to the following bit. The process is repeated for each bit until the rightmost bit has been used.

EXAMPLE 1

Convert 10010011_2 to decimal.

Step 1: Write down the binary number, leaving space between the bits.

1 0 0 1 0 0 1 1

Step 2: Double the high-order bit and copy it under the next bit.

1 0 0 1 0 0 1 1
  2

×2
2

Step 3 : Add the next bit and double the sum. Copy this result under the next bit.

1	0	0	1	0	0	1	1
	2	4					
	+0						
	2						
	×2	×2					
	2	4					

Step 4 : Repeat Step 3 until you run out of bits.

1	0	0	1	0	0	1	1
	2	4	8	18	36	72	146
	+0	+0	+1	+0	+0	+1	+1
	2	4	9	18	36	73	147
×2	×2	×2	×2	×2	×2	×2	
2	4	8	18	36	72	146	

° The answer: $10010011_2 = 147_{10}$

When we combine hextet grouping (in reverse) with the double-dabble method, we find that we can convert hexadecimal to decimal with ease.

EXAMPLE 2

Convert $02CA_{16}$ to decimal.

First, convert the hex to binary by grouping into hextets.

0	2	C	A
0000	0010	1100	1010

• • •

Then apply the double-dabble method on the binary form:

1	0	1	1	0	0	1	0	1	0
	2	4	10	22	44	88	178	356	714
	+0	+1	+1	+0	+0	+1	+0	+1	+0
	2	5	11	22	44	89	178	357	714
×2	×2	×2	×2	×2	×2	×2	×2	×2	
2	4	10	22	44	88	178	356	714	

$02CA_{16} = 1011001010_2 = 714_{10}$

As with addition, signed-magnitude subtraction is carried out in a manner similar to pencil-and-paper decimal arithmetic, where it is sometimes necessary to borrow from digits in the **minuend**.

EXAMPLE 2.12 Subtract 01001111$_2$ from 01100011$_2$ using signed-magnitude arithmetic.

									⇐ borrows
			0	1	1	2			
0		1	~~1~~	~~0~~	~~0~~	~~0~~	1	1	(99)
0	−	1	0	0	1	1	1	1	− (79)
0		0	0	1	0	1	0	0	(20)

We find that $01100011_2 - 01001111_2 = 00010100_2$ in signed-magnitude representation.

EXAMPLE 2.13 Subtract 01100011$_2$ (99) from 01001111$_2$ (79) using signed-magnitude arithmetic.

By inspection, we see that the subtrahend, 01100011, is larger than the minuend, 01001111. With the result obtained in Example 2.12, we know that the difference of these two numbers is 0010100_2. Because the subtrahend is larger than the minuend, all we need to do is change the sign of the difference. So we find that $01001111_2 - 01100011_2 = 10010100_2$ in signed-magnitude representation.

We know that subtraction is the same as "adding the opposite," which equates to negating the value we wish to subtract and then adding instead (which is often much easier than performing all the borrows necessary for subtraction, particularly in dealing with binary numbers). Therefore, we need to look at some examples involving both positive and negative numbers. Recall the rules for addition: (1) If the signs are the same, add the magnitudes and use that same sign for the result; (2) If the signs differ, you must determine which operand has the larger magnitude. The sign of the result is the same as the sign of the operand with the larger magnitude, and the magnitude must be obtained by subtracting (not adding) the smaller one from the larger one.

EXAMPLE 2.14 Add 10010011$_2$ (−19) to 00001101$_2$ (+13) using signed-magnitude arithmetic.

The first number (the augend) is negative because its sign bit is set to 1. The second number (the addend) is positive. What we are asked to do is in fact a subtraction. First, we determine which of the two numbers is larger in

magnitude and use that number for the augend. Its sign will be the sign of the result.

$$\begin{array}{cccccccccc} & & & & 0 & 1 & 2 & & & \Leftarrow \text{borrows} \\ 1 & & 0 & 0 & \cancel{1} & \cancel{0} & \cancel{0} & 1 & 1 & (-19) \\ 0 & - & 0 & 0 & 0 & 1 & 1 & 0 & 1 & +\ (13) \\ \hline 1 & & 0 & 0 & 0 & 0 & 1 & 1 & 0 & (-6) \end{array}$$

With the inclusion of the sign bit, we see that $10010011_2 - 00001101_2 = 10000110_2$ in signed-magnitude representation.

EXAMPLE 2.15 Subtract 10011000_2 (–24) from 10101011_2 (–43) using signed-magnitude arithmetic.

We can convert the subtraction to an addition by negating –24, which gives us 24, and then we can add this to –43, giving us a new problem of –43 + 24. However, we know from the addition rules above that because the signs now differ, we must actually subtract the smaller magnitude from the larger magnitude (or subtract 24 from 43) and make the result negative (because 43 is larger than 24).

$$\begin{array}{cccccccc} & & 0 & 2 & & & & \\ & 0 & \cancel{1} & 0 & 1 & 0 & 1 & 1 & (43) \\ - & 0 & 0 & 1 & 1 & 0 & 0 & 0 & -\ (24) \\ \hline & 0 & 0 & 1 & 0 & 0 & 1 & 1 & (19) \end{array}$$

Note that we are not concerned with the sign until we have performed the subtraction. We know the answer must be negative. So we end up with $10101011_2 - 10011000_2 = 10010011_2$ in signed-magnitude representation.

While reading the preceding examples, you may have noticed how many questions we had to ask ourselves: Which number is larger? Am I subtracting a negative number? How many times do I have to borrow from the minuend? A computer engineered to perform arithmetic in this manner must make just as many decisions (though a whole lot faster). The logic (and circuitry) is further complicated by the fact that signed magnitude has two representations for 0, 10000000 and 00000000. (And mathematically speaking, this simply shouldn't happen!) Simpler methods for representing signed numbers would allow for simpler and less expensive circuits. These simpler methods are based on radix complement systems.

2.4.2 Complement Systems

Number theorists have known for hundreds of years that one decimal number can be subtracted from another by adding the difference of the subtrahend from all nines and adding back a carry. This is called *taking the nine's complement of the subtrahend* or, more formally, finding the **diminished radix complement** of the subtrahend. Let's say we wanted to find $167 - 52$. Taking the difference of 52 from 999, we have 947. Thus, in nine's complement arithmetic, we have $167 - 52 = 167 + 947 = 1114$. The "carry" from the hundreds column is added back to the units place, giving us a correct $167 - 52 = 115$. This method was commonly called *casting out 9s* and has been extended to binary operations to simplify computer arithmetic. The advantage that complement systems give us over signed magnitude is that there is no need to process sign bits separately, but we can still easily check the sign of a number by looking at its high-order bit.

Another way to envision complement systems is to imagine an odometer on a bicycle. Unlike cars, when you go backward on a bike, the odometer will go backward as well. Assuming an odometer with three digits, if we start at zero and end with 700, we can't be sure whether the bike went forward 700 miles or backward 300 miles! The easiest solution to this dilemma is simply to cut the number space in half and use 001–500 for positive miles and 501–999 for negative miles. We have, effectively, cut down the distance our odometer can measure. But now if it reads 997, we know the bike has backed up 3 miles instead of riding forward 997 miles. The numbers 501–999 represent the **radix complements** (the second of the two methods introduced below) of the numbers 001–500 and are being used to represent negative distance.

One's Complement

As illustrated above, the diminished radix complement of a number in base 10 is found by subtracting the subtrahend from the base minus one, which is 9 in decimal. More formally, given a number N in base r having d digits, the diminished radix complement of N is defined to be $\left(r^d - 1\right) - N$. For decimal numbers, $r = 10$, and the diminished radix is $10 - 1 = 9$. For example, the nine's complement of 2468 is $9999 - 2468 = 7531$. For an equivalent operation in binary, we subtract from one less the base (2), which is 1. For example, the one's complement of 0101_2 is $1111_2 - 0101 = 1010$. Although we could tediously borrow and subtract as discussed above, a few experiments will convince you that forming the one's complement of a binary number amounts to nothing more than switching all of the 1s with 0s and vice versa. This sort of bit-flipping is very simple to implement in computer hardware.

It is important to note at this point that although we can find the nine's complement of any decimal number or the one's complement of any binary number, we are most interested in using complement notation to represent negative numbers. We know that performing a subtraction, such as $10 - 7$, can also be thought of as "adding the opposite," as in $10 + (-7)$. Complement notation allows us to simplify subtraction by turning it into addition, but it also gives us a method to represent negative numbers. Because we do not wish to use a special bit to represent the sign (as we did in signed-magnitude representation), we need to

remember that if a number is negative, we should convert it to its complement. The result should have a 1 in the leftmost bit position to indicate that the number is negative.

Although the one's complement of a number is technically the value obtained by subtracting that number from a large power of two, we often refer to a computer using one's complement for negative numbers as a *one's complement system* or as a *computer that uses one's complement arithmetic*. This can be somewhat misleading, as positive numbers do not need to be complemented; we only complement negative numbers so we can get them into a format the computer will understand. Example 2.16 illustrates these concepts.

EXAMPLE 2.16 Express 23_{10} and -9_{10} in 8-bit binary, assuming a computer is using one's complement representation.

$$23_{10} = +(00010111_2) = 00010111_2$$
$$-9_{10} = -(00001001_2) = 11110110_2$$

Unlike signed magnitude, in one's complement addition there is no need to maintain the sign bit separate from the other bits. The sign takes care of itself. Compare Example 2.17 with Example 2.10.

EXAMPLE 2.17 Add 01001111_2 to 00100011_2 using one's complement addition.

				1	1	1	1		⇐ carries
	0	1	0	0	1	1	1	1	(79)
+	0	0	1	0	0	0	1	1	+ (35)
	0	1	1	1	0	0	1	0	(114)

Suppose we wish to subtract 9 from 23. To carry out a one's complement subtraction, we first express the subtrahend (9) in one's complement, then add it to the minuend (23); we are effectively now adding –9 to 23. The high-order bit will have a 1 or a 0 carry, which is added to the low-order bit of the sum. (This is called **end carry-around** and results from using the diminished radix complement.)

EXAMPLE 2.18 Add 23_{10} to -9_{10} using one's complement arithmetic.

1 ←	1	1	1		1	1			⇐ carries
	0	0	0	1	0	1	1	1	(23)
+	1	1	1	1	0	1	1	0	+ (–9)
	0	0	0	0	1	1	0	1	
							+	1	
	0	0	0	0	1	1	1	0	14_{10}

The last carry is added to the sum.

EXAMPLE 2.19 Add 9_{10} to -23_{10} using one's complement arithmetic.

The last carry is 0 so we are done.	0	←	0	0	0	0	1	0	0	1	(9)
		+	1	1	1	0	1	0	0	0	+ (−23)
			1	1	1	1	0	0	0	1	-14_{10}

How do we know that 11110001_2 is actually -14_{10}? We simply need to take the one's complement of this binary number (remembering it must be negative because the leftmost bit is negative). The one's complement of 11110001_2 is 00001110_2, which is 14.

The primary disadvantage of one's complement is that we still have two representations for 0: 00000000 and 11111111. For this and other reasons, computer engineers long ago stopped using one's complement in favor of the more efficient two's complement representation for binary numbers.

Two's Complement

Two's complement is an example of a radix complement. Given a number N in base r having d digits, the radix complement of N is defined as $r^d - N$ for $N \neq 0$ and 0 for $N = 0$. The radix complement is often more intuitive than the diminished radix complement. Using our odometer example, the ten's complement of going forward 2 miles is $10^3 - 2 = 998$, which we have already agreed indicates a negative (backward) distance. Similarly, in binary, the two's complement of the 4-bit number 0011_2 is $2^4 - 0011_2 = 10000_2 - 0011_2 = 1101_2$.

Upon closer examination, you will discover that two's complement is nothing more than one's complement incremented by 1. To find the two's complement of a binary number, simply flip bits and add 1. This simplifies addition and subtraction as well. Because the subtrahend (the number we complement and add) is incremented at the outset, however, there is no end carry-around to worry about. We simply discard any carries involving the high-order bits. Just as with one's complement, *two's complement* refers to the complement of a number, whereas a computer using this notation to represent negative numbers is said to be a *two's complement system*, or a *computer that uses two's complement arithmetic*. As before, positive numbers can be left alone; we only need to complement negative numbers to get them into their two's complement form. Example 2.20 illustrates these concepts.

EXAMPLE 2.20 Express 23_{10}, -23_{10}, and -9_{10} in 8-bit binary, assuming a computer is using two's complement representation.

$$23_{10} = +(00010111_2) = 00010111_2$$
$$-23_{10} = -(00010111_2) = 11101000_2 + 1 = 11101001_2$$
$$-9_{10} = -(00001001_2) = 11110110_2 + 1 = 11110111_2$$

Because the representation of positive numbers is the same in one's complement and two's complement (as well as signed-magnitude), the process of adding two positive binary numbers is the same. Compare Example 2.21 with Example 2.17 and Example 2.10.

Null Pointers: Tips and Hints

You should have a fairly good idea by now of why we need to study the different formats. If numeric values will never be negative (only positive or zero values are used, such as for someone's age), unsigned numbers are used. The advantage to using an unsigned format is that the number of positive values that can be represented is larger (no bits are used for the sign). For example, if we are using 4 bits to represent unsigned integers, we can represent values from 0 to 15. However, if we are using signed magnitude representation, the bits are interpreted differently, and the range becomes –7 to +7.

If the numeric values can be positive or negative, we need to use a format that can represent both. Converting unsigned whole numbers tends to be relatively straightforward, but for signed numbers, in particular, complement systems can be confusing. Because we talk about converting a binary number into a signed two's complement by padding it with 0s to the proper length, and then flipping the bits and adding one, people often think that all numbers are converted this way. For example, if someone is asked how a computer, using two's complement representation, would represent –6 (assuming 4-bit values), they answer: 1010 (take 0110, toggle each bit, and add 1). This is, of course, the correct answer. However, if that same person is asked how the computer, using two's complement representation, would represent +6, they will often do exactly the same thing, giving the same answer, which is incorrect. The value +6 using 4 bits is 0110. The same is true if we are converting from two's complement to decimal: If the binary value is 0111, then its equivalent decimal value is simply 7 (no bit flipping required!). However, if the value is 1010, this represents a negative value, so we would flip the bits (0101), add 1 (0110), and see that this binary value represents –6.

It is important to realize that representing positive numbers does not require any "conversion" at all! It is only when a number is negative that two's complement, signed magnitude, and one's complement representations are necessary. A positive number on a computer that uses signed magnitude, a positive number on a computer that uses one's complement, and a positive number on a computer that uses two's complement will be exactly the same. If a computer uses n bits to represent integers, it can represent 2^n different values (because there are 2^n different bit patterns possible); the computer (and even you!) have no way to know whether a particular bit pattern is unsigned or signed simply by looking at the number. It is up to the programmer to make sure the bit string is interpreted correctly.

EXAMPLE 2.21 Add 01001111_2 to 00100011_2 using two's complement addition.

				1	1	1	1		⇐ carries
	0	1	0	0	1	1	1	1	(79)
+	0	0	1	0	0	0	1	1	+ (35)
	0	1	1	1	0	0	1	0	(114)

Suppose we are given the binary representation for a number and we want to know its decimal equivalent. Positive numbers are easy. For example, to convert the two's complement value of 00010111_2 to decimal, we simply convert this binary number to a decimal number to get 23. However, converting two's complement negative numbers requires a reverse procedure similar to the conversion from decimal to binary. Suppose we are given the two's complement binary value of 11110111_2, and we want to know the decimal equivalent. We know this is a negative number but must remember it is represented using two's complement notation. We first flip the bits and then add 1 (find the one's complement and add 1). This results in the following: $00001000_2 + 1 = 00001001_2$. This is equivalent to the decimal value 9. However, the original number we started with was negative, so we end up with –9 as the decimal equivalent to 11110111_2.

The following two examples illustrate how to perform addition (and hence subtraction, because we subtract a number by adding its opposite) using two's complement notation.

EXAMPLE 2.22 Add 9_{10} to -23_{10} using two's complement arithmetic.

	0	0	0	0	1	0	0	1	(9)
+	1	1	1	0	1	0	0	1	+ (–23)
	1	1	1	1	0	0	1	0	-14_{10}

It is left as an exercise for you to verify that 11110010_2 is actually -14_{10} using two's complement notation.

EXAMPLE 2.23 Find the sum of 23_{10} and -9_{10} in binary using two's complement arithmetic.

	1 ←	1	1	1		1	1	1		⇐ carries
		0	0	0	1	0	1	1	1	(23)
+		1	1	1	1	0	1	1	1	+ (–9)
		0	0	0	0	1	1	1	0	14_{10}

Discard carry

In two's complement, the addition of two negative numbers produces a negative number, as we might expect.

EXAMPLE 2.24 Find the sum of 11101001_2 (–23) and 11110111_2 (–9) using two's complement addition.

Discard carry

1 ←	1	1	1	1	1	1	1		⇐ carries
	1	1	1	0	1	0	0	1	(–23)
+	1	1	1	1	0	1	1	1	+ (–9)
	1	1	1	0	0	0	0	0	(–32)

Notice that the discarded carries in Examples 2.23 and 2.24 did not cause an erroneous result. An overflow occurs if two positive numbers are added and the result is negative, or if two negative numbers are added and the result is positive. It is not possible to have overflow when using two's complement notation if a positive and a negative number are being added together.

Simple computer circuits can easily detect an overflow condition using a rule that is easy to remember. You'll notice in both Examples 2.23 and 2.24 that the carry going into the sign bit (a 1 is carried from the previous bit position into the sign bit position) is the same as the carry going out of the sign bit (a 1 is carried out and discarded). When these carries are equal, no overflow occurs. When they differ, an overflow indicator is set in the arithmetic logic unit, indicating the result is incorrect.

> **A Simple Rule for Detecting an Overflow Condition in Signed Numbers:** If the carry into the sign bit equals the carry out of the bit, no overflow has occurred. If the carry into the sign bit is different from the carry out of the sign bit, overflow (and thus an error) has occurred.

The hard part is getting programmers (or compilers) to consistently check for the overflow condition. Example 2.25 indicates overflow because the carry into the sign bit (a 1 is carried in) is not equal to the carry out of the sign bit (a 0 is carried out).

EXAMPLE 2.25 Find the sum of 126_{10} and 8_{10} in binary using two's complement arithmetic.

Discard last carry

0 ←	1	1	1	1					⇐ carries
	0	1	1	1	1	1	1	0	(126)
+	0	0	0	0	1	0	0	0	+ (8)
	1	0	0	0	0	1	1	0	(–122???)

A 1 is carried into the leftmost bit, but a 0 is carried out. Because these carries are not equal, an overflow has occurred. (We can easily see that two positive numbers are being added but the result is negative.) We return to this topic in Section 2.4.6.

Two's complement is the most popular choice for representing signed numbers. The algorithm for adding and subtracting is quite easy, has the best representation for 0 (all 0 bits), is self-inverting, and is easily extended to larger numbers

INTEGER MULTIPLICATION AND DIVISION

Unless sophisticated algorithms are used, multiplication and division can consume a considerable number of computation cycles before a result is obtained. Here, we discuss only the most straightforward approach to these operations. In real systems, dedicated hardware is used to optimize throughput, sometimes carrying out portions of the calculation in parallel. Curious readers will want to investigate some of these advanced methods in the references cited at the end of this chapter.

The simplest multiplication algorithms used by computers are similar to traditional pencil-and-paper methods used by humans. The complete multiplication table for binary numbers couldn't be simpler: zero times any number is zero, and one times any number is that number.

To illustrate simple computer multiplication, we begin by writing the multiplicand and the multiplier to two separate storage areas. We also need a third storage area for the product. Starting with the low-order bit, a pointer is set to each digit of the multiplier. For each digit in the multiplier, the multiplicand is "shifted" one bit to the left. When the multiplier is 1, the "shifted" multiplicand is added to a running sum of partial products. Because we shift the multiplicand by one bit for each bit in the multiplier, a product requires double the working space of either the multiplicand or the multiplier.

There are two simple approaches to binary division: We can either iteratively subtract the denominator from the divisor, or we can use the same trial-and-error method of long division that we were taught in grade school. As with multiplication, the most efficient methods used for binary division are beyond the scope of this text and can be found in the references at the end of this chapter.

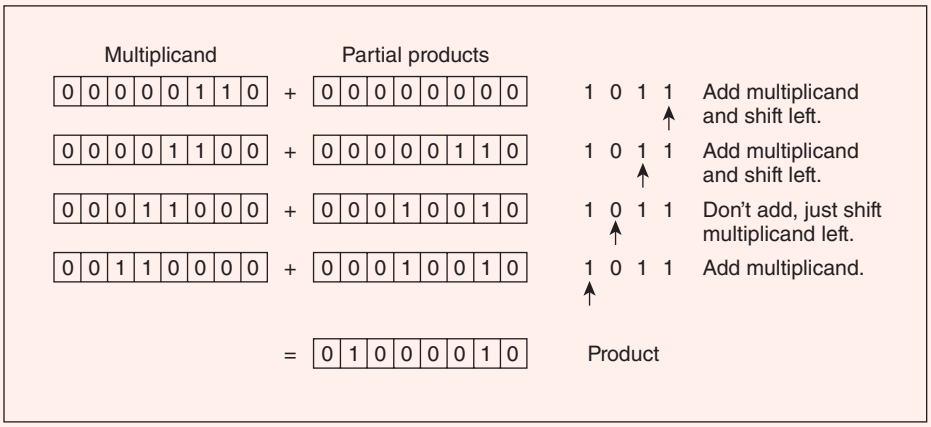
Regardless of the relative efficiency of any algorithms that are used, division is an operation that can always cause a computer to crash. This is the case

of bits. The biggest drawback is in the asymmetry seen in the range of values that can be represented by N bits. With signed-magnitude numbers, for example, 4 bits allow us to represent the values −7 through +7. However, using two's complement, we can represent the values −8 through +7, which is often confusing to anyone learning about complement representations. To see why +7 is the largest number we can represent using 4-bit two's complement representation, we need only remember that the first bit must be 0. If the remaining bits are all 1s (giving us the largest magnitude possible), we have 0111_2, which is 7. An immediate reaction to this is that the smallest negative number should then be 1111_2, but we can see that 1111_2 is actually −1 (flip the bits, add 1, and make the number negative). So how do we represent −8 in two's complement notation using 4 bits? It is represented as 1000_2. We know this is a negative number. If we flip the bits (0111), add 1 (to get 1000, which is 8), and make it negative, we get −8.

particularly when division by zero is attempted or when two numbers of enormously different magnitudes are used as operands. When the divisor is much smaller than the dividend, we get a condition known as **divide underflow**, which the computer sees as the equivalent of division by zero, which is impossible.

Computers make a distinction between integer division and floating-point division. With integer division, the answer comes in two parts: a quotient and a remainder. Floating-point division results in a number that is expressed as a binary fraction. These two types of division are sufficiently different from each other as to warrant giving each its own special circuitry. Floating-point calculations are carried out in dedicated circuits called **floating-point units**, or **FPUs**.

EXAMPLE Find the product of 00000110_2 and 00001011_2.



2.4.3 Excess-M Representation for Signed Numbers

Recall the bicycle example that we discussed when introducing complement systems. We selected a particular value (500) as the cutoff for positive miles, and we assigned values from 501 to 999 to negative miles. We didn't need signs because we used the range to determine whether the number was positive or negative. **Excess-M representation** (also called **offset binary** representation) does something very similar; unsigned binary values are used to represent signed integers. However, excess-M representation, unlike signed magnitude and the complement encodings, is more intuitive because the binary string with all 0s represents the smallest number, whereas the binary string with all 1s represents the largest value; in other words, ordering is preserved.

The unsigned binary representation for integer M (called the **bias**) represents the value 0, whereas all 0s in the bit pattern represents the integer –M. Essentially, a decimal integer is "mapped" (as in our bicycle example) to an unsigned

binary integer, but interpreted as positive or negative depending on where it falls in the range. If we are using n bits for the binary representation, we need to select the bias in such a manner that we split the range equally. We typically do this by choosing a bias of $2^{n-1} - 1$. For example, if we were using 4-bit representation, the bias should be $2^{4-1} - 1 = 7$. Just as with signed magnitude, one's complement, and two's complement, there is a specific range of values that can be expressed in n bits.

The unsigned binary value for a signed integer using excess-M representation is determined simply by adding M to that integer. For example, assuming that we are using excess-7 representation, the integer 0_{10} would be represented as $0 + 7 = 7_{10} = 0111_2$; the integer 3_{10} would be represented as $3 + 7 = 10_{10} = 1010_2$; and the integer -7 would be represented as $-7 + 7 = 0_{10} = 0000_2$. Using excess-7 notation and given the binary number 1111_2, to find the decimal value it represents, we simply subtract 7: $1111_2 = 15_{10}$, and $15 - 7 = 8$; therefore, the value 1111_2, using excess-7 representation, is $+8_{10}$.

Let's compare the encoding schemes we have seen so far, assuming 8-bit numbers:

Integer		Binary Strings Representing the Signed Integer			
Decimal	Binary (for absolute value)	Signed Magnitude	One's Complement	Two's Complement	Excess-127
2	00000010	00000010	00000010	00000010	10000001
–2	00000010	10000010	11111101	11111110	01111101
100	01100100	01100100	01100100	01100100	11100011
–100	01100100	11100100	10011011	10011100	00011011

Excess-M representation allows us to use unsigned binary values to represent signed integers; it is important to note, however, that two parameters must be specified: the number of bits being used in the representation and the bias value itself. In addition, a computer is unable to perform addition on excess-M values using hardware designed for unsigned numbers; special circuits are required. Excess-M representation is important because of its use in representing integer exponents in floating-point numbers, as we will see in Section 2.5.

2.4.4 Unsigned Versus Signed Numbers

We introduced our discussion of binary integers with unsigned numbers. Unsigned numbers are used to represent values that are guaranteed not to be negative. A good example of an unsigned number is a memory address. If the 4-bit binary value 1101 is unsigned, then it represents the decimal value 13, but as a signed two's complement number, it represents –3. Signed numbers are used to represent data that can be either positive or negative.

A computer programmer must be able to manage both signed and unsigned numbers. To do so, the programmer must first identify numeric values as either signed or unsigned numbers. This is done by declaring the value as a specific

type. For instance, the C programming language has `int` and `unsigned int` as possible types for integer variables, defining signed and unsigned integers, respectively. In addition to different type declarations, many languages have different arithmetic operations for use with signed and unsigned numbers. A language may have one subtraction instruction for signed numbers and a different subtraction instruction for unsigned numbers. In most assembly languages, programmers can choose from a signed comparison operator or an unsigned comparison operator.

It is interesting to compare what happens with unsigned and signed numbers when we try to store values that are too large for the specified number of bits. Unsigned numbers simply wrap around and start over at 0. For example, if we are using 4-bit unsigned binary numbers, and we add 1 to 1111, we get 0000. This "return to 0" wraparound is familiar—perhaps you have seen a high-mileage car in which the odometer has wrapped back around to 0. However, signed numbers devote half their space to positive numbers and the other half to negative numbers. If we add 1 to the largest positive 4-bit two's complement number 0111 (+7), we get 1000 (–8). This wraparound with the unexpected change in sign has been problematic to inexperienced programmers, resulting in multiple hours of debugging time. Good programmers understand this condition and make appropriate plans to deal with the situation before it occurs.

2.4.5 Computers, Arithmetic, and Booth's Algorithm

Computer arithmetic as introduced in this chapter may seem simple and straightforward, but it is a field of major study in computer architecture. The basic focus is on the implementation of arithmetic functions, which can be realized in software, firmware, or hardware. Researchers in this area are working toward designing superior central processing units (CPUs), developing high-performance arithmetic circuits, and contributing to the area of embedded systems application-specific circuits. They are working on algorithms and new hardware implementations for fast addition, subtraction, multiplication, and division, as well as fast floating-point operations. Researchers are looking for schemes that use nontraditional approaches, such as the **fast carry look-ahead** principle, **residue arithmetic**, and **Booth's algorithm**. Booth's algorithm is a good example of one such scheme and is introduced here in the context of signed two's complement numbers to give you an idea of how a simple arithmetic operation can be enhanced by a clever algorithm.

Although Booth's algorithm usually yields a performance increase when multiplying two's complement numbers, there is another motivation for introducing this algorithm. In Section 2.4.2, we covered examples of two's complement addition and saw that the numbers could be treated as unsigned values. We simply perform "regular" addition, as the following example illustrates:

$$\begin{array}{rll} & 1\ 0\ 0\ 1 & (-7) \\ + & 0\ 0\ 1\ 1 & (+3) \\ \hline & 1\ 1\ 0\ 0 & (-4) \end{array}$$

The same is true for two's complement subtraction. However, now consider the standard pencil-and-paper method for multiplying the following two's complement numbers:

```
    1011      (–5)
  × 1100      (–4)
  ------
    0000
   0000
  1011
 1011
 --------
 10000100     (–124)
```

"Regular" multiplication clearly yields the incorrect result. There are a number of solutions to this problem, such as converting both values to positive numbers, performing conventional multiplication, and then remembering if one or both values were negative to determine whether the result should be positive or negative. Booth's algorithm not only solves this dilemma, but also speeds up multiplication in the process.

The general idea of Booth's algorithm is to increase the speed of multiplication when there are consecutive 0s or 1s in the multiplier. It is easy to see that consecutive 0s help performance. For example, if we use the tried and true pencil-and-paper method and find 978×1001, the multiplication is much easier than if we take 978×999. This is because of the two 0s found in 1001. However, if we rewrite the two problems as follows:

$$978 \times 1001 = 978 \times (1000 + 1) = 978 \times 1000 + 978$$

$$978 \times 999 = 978 \times (1000 - 1) = 978 \times 1000 - 978$$

we see that the problems are in fact equal in difficulty.

Our goal is to use a string of 1s in a binary number to our advantage in much the same way that we use a string of 0s to our advantage. We can use the rewriting idea from above. For example, the binary number 0110 can be rewritten $1000 - 0010 = -0010 + 1000$. The two 1s have been replaced by a "subtract" (determined by the rightmost 1 in the string) followed by an "add" (determined by moving one position left of the leftmost 1 in the string).

Consider the following standard multiplication example:

```
    0011
  × 0110
  ------
  + 0000   (0 in multiplier means simple shift)
  + 0011   (1 in multiplier means add multiplicand and shift)
  + 0011   (1 in multiplier means add multiplicand and shift)
  + 0000   (0 in multiplier means simple shift)
  --------
  00010010
```

The idea of Booth's algorithm is to replace the string of 1s in the multiplier with an initial subtract when we see the rightmost 1 of the string (or subtract 0011) and then later add for the bit after the last 1 (or add 001100). In the middle of the string, we can now use simple shifting:

```
    0011
  ×0000
  +0000   (0 in multiplier means shift)
  −0011   (first 1inmultiplier means subtract multiplicand and shift)
 +0000    (middle of string of 1s means shift)
+0011     (prior step had last1 so addmultiplicand)
 00010010
```

In Booth's algorithm, if the multiplicand and multiplier are n-bit two's complement numbers, the result is a $2n$-bit two's complement value. Therefore, when we perform our intermediate steps, we must extend our n-bit numbers to $2n$-bit numbers. If a number is negative and we extend it, we must extend the sign. For example, the value 1000 (−8) extended to 8 bits would be 11111000. We continue to work with bits in the multiplier, *shifting each time we complete a step*. However, we are interested in *pairs* of bits in the multiplier and proceed according to the following rules:

1. If the current multiplier bit is 1 and the preceding bit was 0, we are at the beginning of a string of 1s so we subtract the multiplicand from the product (or add the opposite).
2. If the current multiplier bit is 0 and the preceding bit was 1, we are at the end of a string of 1s so we add the multiplicand to the product.
3. If it is a 00 pair, or a 11 pair, do no arithmetic operation (we are in the middle of a string of 0s or a string of 1s). Simply shift. The power of the algorithm is in this step: We can now treat a string of 1s as a string of 0s and do nothing more than shift.

Note: The first time we pick a pair of bits in the multiplier, we should assume a mythical 0 as the "previous" bit. Then we simply move left one bit for our next pair.

Example 2.26 illustrates the use of Booth's algorithm to multiply -3×5 using signed 4-bit two's complement numbers.

EXAMPLE 2.26 Negative 3 in 4-bit two's complement is 1101. Extended to 8 bits, it is 11111101. Its complement is 00000011. When we see the rightmost 1 in the multiplier, it is the beginning of a string of 1s, so we treat it as if it were the string 10:

```
          1 1 0 1   (for subtracting, we will add −3's complement, or 00000011)
        × 0 1 0 1
  + 0 0 0 0 0 0 1 1   (10 = subtract 1101 = add 00000011)
+ 1 1 1 1 1 1 0 1     (01 = add 11111101 to product—note sign extension)
+ 0 0 0 0 0 0 1 1     (10 = subtract 1101 = add 00000011)
+ 1 1 1 1 1 1 0 1     (01 = add multiplicand 11111101 to product)
1 0 0 1 1 1 1 1 0 0 0 1   (using the 8 rightmost bits, we have −3 × 5 = −15)
↑
```

Ignore extended sign bits that go beyond 2n.

EXAMPLE 2.27 Let's look at the larger example of 53×126:

00110101	(for subtracting, we will add the complement of 53, or
$\times$ 01111110	11001011)
+ 0000000000000000	(00 = simple shift)
+ 111111111001011	(10 = subtract = add 11001011, extend sign)
+ 00000000000000	(11 = simple shift)
+ 0000000000000	(11 = simple shift)
+ 000000000000	(11 = simple shift)
+ 00000000000	(11 = simple shift)
+ 0000000000	(11 = simple shift)
+ 000110101	(01 = add)
10001101000010110	(53 $\times$ 126 = 6678)

Note that we have not shown the extended sign bits that go beyond what we need and use only the 16 rightmost bits. The entire string of 1s in the multiplier was replaced by a subtract (adding 11001011) followed by an add. Everything in the middle is simply shifting—something that is very easy for a computer to do (as we will see in Chapter 3). If the time required for a computer to do an add is sufficiently larger than that required to do a shift, Booth's algorithm can provide a considerable increase in performance. This depends somewhat, of course, on the multiplier. If the multiplier has strings of 0s and/or 1s the algorithm works well. If the multiplier consists of an alternating string of 0s and 1s (the worst case), using Booth's algorithm might very well require more operations than the standard approach.

Computers perform Booth's algorithm by adding and shifting values stored in registers. A special type of shift called an **arithmetic shift** is necessary to preserve the sign bit. Many books present Booth's algorithm in terms of arithmetic shifts and add operations on registers only, so it may appear quite different from the preceding method. We have presented Booth's algorithm so that it more closely resembles the pencil-and-paper method with which we are all familiar, although it is equivalent to the computer algorithms presented elsewhere.

There have been many algorithms developed for fast multiplication, but many do not hold for signed multiplication. Booth's algorithm not only allows multiplication to be performed faster in most cases, but it also has the added bonus of working correctly on signed numbers.

2.4.6 Carry Versus Overflow

The wraparound referred to in the preceding section is really overflow. CPUs often have flags to indicate both carry and overflow. However, the overflow flag is used only with signed numbers and means nothing in the context of unsigned numbers, which use the carry flag instead. If carry (which means *carry out of the leftmost bit*) occurs in unsigned numbers, we know we have overflow (the new value is too large to be stored in the given number of bits) but the overflow bit is not set. Carry out can occur in signed numbers as well; however, its occurrence in signed numbers is neither sufficient nor necessary for overflow. We have already

Expression	Result	Carry?	Overflow?	Correct Result?
0100(+4)+0010(+2)	0110(+6)	No	No	Yes
0100(+4)+0110(+6)	1010(−6)	No	Yes	No
1100(−4)+1110(−2)	1010(−6)	Yes	No	Yes
1100(−4)+1010(−6)	0110(+6)	Yes	Yes	No

TABLE 2.2 Examples of Carry and Overflow in Signed Numbers

seen that overflow in signed numbers can be determined if the carry in to the leftmost bit and the carry out of the leftmost bit differ. However, carry out of the leftmost bit in unsigned operations always indicates overflow.

To illustrate these concepts, consider 4-bit unsigned and signed numbers. If we add the two unsigned values 0111 (7) and 0001 (1), we get 1000 (8). There is no carry (out), and thus no error. However, if we add the two unsigned values 0111 (7) and 1011 (11), we get 0010 with a carry, indicating that there is an error (indeed, 7 + 11 is not 2). This wraparound would cause the carry flag in the CPU to be set. Essentially, carry out in the context of unsigned numbers means an overflow has occurred, even though the overflow flag is not set.

We said carry (out) is neither sufficient nor necessary for overflow in signed numbers. Consider adding the two's complement integers 0101 (+5) and 0011 (+3). The result is 1000 (−8), which is clearly incorrect. The problem is that we have a carry in to the sign bit, but no carry out, which indicates that we have an overflow (therefore, carry is not necessary for overflow). However, if we now add 0111 (+7) and 1011 (−5), we get the correct result: 0010 (+2). We have both a carry in to and a carry out of the leftmost bit, so there is no error (so carry is not sufficient for overflow). The carry flag would be set, but the overflow flag would not be set. Thus carry out does not necessarily indicate an error in signed numbers, nor does the lack of carry out indicate that the answer is correct.

To summarize, the rule of thumb used to determine when carry indicates an error depends on whether we are using signed or unsigned numbers. For unsigned numbers, a carry (out of the leftmost bit) indicates the total number of bits was not large enough to hold the resulting value, and overflow has occurred. For signed numbers, if the carry in to the sign bit and the carry (out of the sign bit) differ, then overflow has occurred. The overflow flag is set only when overflow occurs with signed numbers.

Carry and overflow clearly occur independently of each other. Examples using signed two's complement representation are given in Table 2.2. Carry in to the sign bit is not indicated in the table.

2.4.7 Binary Multiplication and Division Using Shifting

Shifting a binary number simply means moving the bits left or right by a certain amount. For example, the binary value 00001111 shifted left one place results in 00011110 (if we fill with a 0 on the right). The first number is equivalent to decimal value 15; the second is decimal 30, which is exactly double the first value. This is no coincidence!

When working with signed two's complement numbers, we can use a special type of shift, called an *arithmetic shift*, to perform quick and easy multiplication and division by 2. Recall that the leftmost bit in a two's complement number

determines its sign, so we must be careful when shifting these values that we don't change the sign bit, as multiplying or dividing by 2 should not change the sign of the number.

We can perform a left arithmetic shift (which multiples a number by 2) or a right arithmetic shift (which divides a number by 2). Assuming that bits are numbered right to left beginning with 0, we have the following definitions for left and right arithmetic shifts.

A **left arithmetic shift** inserts a 0 in for bit b_0 and shifts all other bits left one position, resulting in bit b_{n-1} being replaced by bit b_{n-2}. Because bit b_{n-1} is the sign bit, if the value in this bit changes, the operation has caused overflow. Multiplication by 2 always results in a binary number with the rightmost bit equal to 0, which is an even number, and thus explains why we pad with a 0 on the right. Consider the following examples:

EXAMPLE 2.28 Multiply the value 11 (expressed using 8-bit signed two's complement representation) by 2.

We start with the binary value for 11:

0 0 0 0 1 0 1 1

and we shift left one place, resulting in:

0 0 0 1 0 1 1 0

which is decimal $2 = 11 \times 2$. No overflow has occurred, so the value is correct.

EXAMPLE 2.29 Multiply the value 12 (expressed using 8-bit signed two's complement representation) by 4.

We start with the binary value for 12:

0 0 0 0 1 1 0 0

and we shift left two places (each shift multiplies by 2, so two shifts is equivalent to multiplying by 4), resulting in:

0 0 1 1 0 0 0 0

which is decimal $48 = 12 \times 4$. No overflow has occurred, so the value is correct.

EXAMPLE 2.30 Multiply the value 66 (expressed using 8-bit signed two's complement representation) by 2.

We start with the binary value for 66:

0 1 0 0 0 0 1 0

and we shift left one place, resulting in:

1 0 0 0 0 1 0 0

but the sign bit has changed, so overflow has occurred ($66 \times 2 = 132$, which is too large to be expressed using 8 bits in signed two's complement notation).

A **right arithmetic shift** moves all bits to the right, but carries (copies) the sign bit from bit b_{n-1} to b_{n-2}. Because we copy the sign bit from right to left, overflow is not a problem. However, division by 2 may have a remainder of 1; division using this method is strictly integer division, so the remainder is not stored in any way. Consider the following examples:

EXAMPLE 2.31 Divide the value 12 (expressed using 8-bit signed two's complement representation) by 2.

We start with the binary value for 12:

0 0 0 0 1 1 0 0

and we shift right one place, copying the sign bit of 0, resulting in:

0 0 0 0 0 1 1 0

which is decimal $6 = 12 \div 2$.

EXAMPLE 2.32 Divide the value 12 (expressed using 8-bit signed two's complement representation) by 4.

We start with the binary value for 12:

0 0 0 0 1 1 0 0

and we shift right two places, resulting in:

0 0 0 0 0 0 1 1

which is decimal $3 = 12 \div 4$.

EXAMPLE 2.33 Divide the value −14 (expressed using 8-bit signed two's complement representation) by 2.

We start with the two's complement representation for −14:

1 1 1 1 0 0 1 0

and we shift right one place (carrying across the sign bit), resulting in:

1 1 1 1 1 0 0 1

which is decimal $-7 = -14 \div 2$.

Note that if we had divided −15 by 2 (in Example 2.33), the result would be 11110001 shifted one to the left to yield 11111000, which is −8. Because we are doing integer division, −15 divided by 2 is indeed equal to −8.

2.5 FLOATING-POINT REPRESENTATION

If we wanted to build a real computer, we could use any of the integer representations that we just studied. We would pick one of them and proceed with our design tasks. Our next step would be to decide the word size of our system. If

we want our system to be really inexpensive, we would pick a small word size, say, 16 bits. Allowing for the sign bit, the largest integer this system could store is 32,767. So now what do we do to accommodate a potential customer who wants to keep a tally of the number of spectators paying admission to professional sports events in a given year? Certainly, the number is larger than 32,767. No problem. Let's just make the word size larger. Thirty-two bits ought to do it. Our word is now big enough for just about anything that anyone wants to count. But what if this customer also needs to know the amount of money each spectator spends per minute of playing time? This number is likely to be a decimal fraction. Now we're really stuck.

The easiest and cheapest approach to this problem is to keep our 16-bit system and say, "Hey, we're building a cheap system here. If you want to do fancy things with it, get yourself a good programmer." Although this position sounds outrageously flippant in the context of today's technology, it was a reality in the earliest days of each generation of computers. There simply was no such thing as a floating-point unit in many of the first mainframes or microcomputers. For many years, clever programming enabled these integer systems to act as if they were, in fact, floating-point systems.

If you are familiar with scientific notation, you may already be thinking of how you could handle floating-point operations—how you could provide **floating-point emulation**—in an integer system. In scientific notation, numbers are expressed in two parts: a fractional part and an exponential part that indicates the power of ten to which the fractional part should be raised to obtain the value we need. So to express 32,767 in scientific notation, we could write 3.2767×10^4. Scientific notation simplifies pencil-and-paper calculations that involve very large or very small numbers. It is also the basis for floating-point computation in today's digital computers.

2.5.1 A Simple Model

In digital computers, floating-point numbers consist of three parts: a sign bit, an exponent part (representing the exponent on a power of 2), and a fractional part (which has sparked considerable debate regarding appropriate terminology). The term **mantissa** is widely accepted when referring to this fractional part. However, many people take exception to this term because it also denotes the fractional part of a logarithm, which is not the same as the fractional part of a floating-point number. The Institute of Electrical and Electronics Engineers (IEEE) introduced the term **significand** to refer to the fractional part of a floating-point number combined with the implied binary point and implied 1 (which we discuss at the end of this section). Regrettably, the terms *mantissa* and *significand* have become interchangeable when referring to the fractional part of a floating-point number, even though they are not technically equivalent. Throughout this text, we refer to the fractional part as the *significand*, regardless of whether it includes the implied 1 as intended by IEEE (see Section 2.5.4).

The number of bits used for the exponent and significand depends on whether we would like to optimize for range (more bits in the exponent) or precision

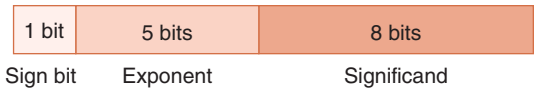


FIGURE 2.1 A Simple Model Floating-Point Representation

(more bits in the significand). (We discuss range and precision in more detail in Section 2.5.7.) For the remainder of this section, we will use a 14-bit model with a 5-bit exponent, an 8-bit significand, and a sign bit (see Figure 2.1). More general forms are described in Section 2.5.2.

Let's say that we wish to store the decimal number 17 in our model. We know that $17 = 17.0 \times 10^0 = 1.7 \times 10^1 = 0.17 \times 10^2$. Analogously, in binary, $17_{10} = 10001_2 \times 2^0 = 1000.1_2 \times 2^1 = 100.01_2 \times 2^2 = 10.001_2 \times 2^3 = 1.0001_2 \times 2^4 = 0.10001_2 \times 2^5$. If we use this last form, our fractional part will be 10001000 and our exponent will be 00101, as shown here:

0	0	0	1	0	1	1	0	0	0	1	0	0	0

Using this form, we can store numbers of much greater magnitude than we could using a **fixed-point** representation of 14 bits (which uses a total of 14 binary digits plus a binary, or radix, point). If we want to represent $65536 = 0.1_2 \times 2^{17}$ in this model, we have:

0	1	0	0	0	1	1	0	0	0	0	0	0	0

One obvious problem with this model is that we haven't provided for negative exponents. If we wanted to store 0.25, we would have no way of doing so because 0.25 is 2^{-2} and the exponent -2 cannot be represented. We could fix the problem by adding a sign bit to the exponent, but it turns out that it is more efficient to use a **biased** exponent, because we can use simpler integer circuits designed specifically for unsigned numbers when comparing the values of two floating-point numbers.

Recall from Section 2.4.3 that the idea behind using a bias value is to convert every integer in the range into a nonnegative integer, which is then stored as a binary numeral. The integers in the desired range of exponents are first adjusted by adding this fixed bias value to each exponent. The bias value is a number near the middle of the range of possible values that we select to represent 0. In this case, we would select 15 because it is midway between 0 and 31 (our exponent has 5 bits, thus allowing for 2^5 or 32 values). Any number larger than 15 in the exponent field represents a positive value. Values less than 15 indicate negative values. This is called an **excess-15** representation because we have to subtract 15 to get the true value of the exponent. Note that exponents of all 0s or all 1s are typically reserved for special numbers (such as 0 or infinity). In our simple model, we allow exponents of all 0s and 1s.

Returning to our example of storing 17, we calculated $17_{10} = 0.10001_2 \times 2^5$. If we update our model to use a biased exponent, the biased exponent is $15 + 5 = 20$:

0	1 0 1 0 0	1 0 0 0 1 0 0 0

To calculate the value of this number, we would take the exponent field (binary value of 20) and subtract the bias $(20 - 4)$, giving us a "real" exponent of 5, resulting in 0.10001000×2^5.

If we wanted to store $0.25 = 0.1 \times 2^{-1}$, we would have:

0	0 1 1 1 0	1 0 0 0 0 0 0 0

There is still one rather large problem with this system: We do not have a unique representation for each number. All of the following are equivalent:

0	1 0 1 0 1	1 0 0 0 1 0 0 0	=

0	1 0 1 1 0	0 1 0 0 0 1 0 0	=

0	1 0 1 1 1	0 0 1 0 0 0 1 0	=

0	1 1 0 0 0	0 0 0 1 0 0 0 1

Because synonymous forms such as these are not well-suited for digital computers, floating-point numbers must be **normalized**—that is, the leftmost bit of the significand must always be 1. This process is called **normalization**. This convention has the additional advantage that if the 1 is implied, we effectively gain an extra bit of precision in the significand. Normalization works well for every value except 0, which contains no nonzero bits. For that reason, any model used to represent floating-point numbers must treat 0 as a special case. We will see in the next section that the IEEE-754 floating-point standard makes an exception to the rule of normalization.

EXAMPLE 2.34 Express 0.03125_{10} in normalized floating-point form using the simple model with excess-15 bias.

$0.03125_{10} = 0.00001_2 \times 2^0 = 0.0001 \times 2^{-1} = 0.001 \times 2^{-2} = 0.01 \times 2^{-3} = 0.1 \times 2^{-4}$.

Applying the bias, the exponent field is $15 - 4 = 11$.

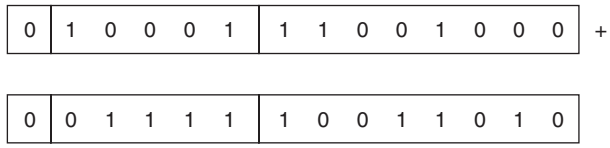
0	0 1 0 1 1	1 0 0 0 0 0 0 0

Note that in our simple model we have not expressed the number using the normalization notation that implies the 1, which is introduced in Section 2.5.4.

2.5.2 Floating-Point Arithmetic

If we wanted to add two decimal numbers that are expressed in scientific notation, such as $1.5 \times 10^2 + 3.5 \times 10^3$, we would change one of the numbers so that both of them are expressed in the same power of the base. In our example, $1.5 \times 10^2 + 3.5 \times 10^3 = 0.15 \times 10^3 + 3.5 \times 10^3 = 3.65 \times 10^3$. Floating-point addition and subtraction work the same way, as illustrated below.

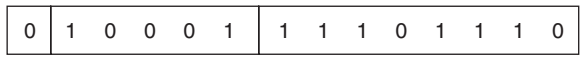
EXAMPLE 2.35 Add the following binary numbers as represented in a normalized 14-bit format, using the simple model with a bias of 15.



We see that the addend is raised to the second power and that the augend is to the 0 power. Alignment of these two operands on the binary point gives us:

$$\begin{array}{r} 11.001000 \\ +\,0.10011010 \\ \hline 11.10111010 \end{array}$$

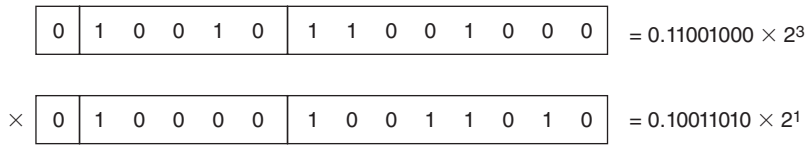
Renormalizing, we retain the larger exponent and truncate the low-order bit. Thus, we have:



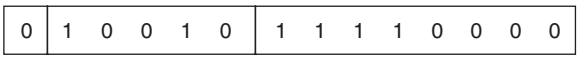
However, because our simple model requires a normalized significand, we have no way to represent 0. This is easily remedied by allowing the string of all 0 (a 0 sign, a 0 exponent, and a 0 significand) to represent the value 0. In the next section, we will see that IEEE-754 also reserves special meaning for certain bit patterns.

Multiplication and division are carried out using the same rules of exponents applied to decimal arithmetic, such as $2^{-3} \times 2^4 = 2^1$, for example.

EXAMPLE 2.36 Assuming a 15-bit bias, multiply:



Multiplication of 0.11001000 by 0.10011010 yields a product of 0.011110000 1010000, and then multiplying by $2^3 \times 2^1 = 2^4$ yields 111.10000101. Renormalizing and supplying the appropriate exponent, the floating-point product is:



2.5.3 Floating-Point Errors

When we use pencil and paper to solve a trigonometry problem or compute the interest on an investment, we intuitively understand that we are working in the system of real numbers. We know that this system is infinite, because given any pair of real numbers, we can always find another real number that is smaller than one and greater than the other.

Unlike the mathematics in our imaginations, computers are finite systems, with finite storage. When we call upon our computers to carry out floating-point calculations, we are modeling the infinite system of real numbers in a finite system of integers. What we have, in truth, is an *approximation* of the real number system. The more bits we use, the better the approximation. However, there is always some element of error, no matter how many bits we use.

Floating-point errors can be blatant, subtle, or unnoticed. The blatant errors, such as numeric overflow or underflow, are the 1s that cause programs to crash. Subtle errors can lead to wildly erroneous results that are often hard to detect before they cause real problems. For example, in our simple model, we can express normalized numbers in the range of $-.11111111_2 \times 2^{16}$ through $+.11111111 \times 2^{16}$. Obviously, we cannot store 2^{-19} or 2^{128}; they simply don't fit. It is not quite so obvious that we cannot accurately store 128.5, which is well within our range. Converting 128.5 to binary, we have 10000000.1, which is 9 bits wide. Our significand can hold only eight. Typically, the low-order bit is dropped or rounded into the next bit. No matter how we handle it, however, we have introduced an error into our system.

We can compute the relative error in our representation by taking the ratio of the absolute value of the error to the true value of the number. Using our example of 128.5, we find:

$$\frac{128.5 - 128}{128.5} = 0.00389105 \approx 0.39\%.$$

If we are not careful, such errors can propagate through a lengthy calculation, causing substantial loss of precision. Table 2.3 illustrates the error propagation as we iteratively multiply 16.24 by 0.91 using our 14-bit simple model. Upon converting these numbers to 8-bit binary, we see that we have a substantial error from the outset.

As you can see, in six iterations, we have more than tripled the error in the product. Continued iterations will produce an error of 100% because the product eventually goes to 0. Although this 14-bit model is so small that it exaggerates the error, all

Multiplier		Multiplicand	14-Bit Product	Real Product	Error
10000.001 (16.125)	×	0.11101000 = (0.90625)	1110.1001 (14.5625)	14.7784	1.46%
1110.1001 (14.5625)	×	0.11101000 =	1101.0011 (13.1885)	13.4483	1.94%
1101.0011 (13.1885)	×	0.11101000 =	1011.1111 (11.9375)	12.2380	2.46%
1011.1111 (11.9375)	×	0.11101000 =	1010.1101 (10.8125)	11.1366	2.91%
1010.1101 (10.8125)	×	0.11101000 =	1001.1100 (9.75)	10.1343	3.79%
1001.1100 (9.75)	×	0.11101000 =	1000.1101 (8.8125)	8.3922	4.44%

TABLE 2.3 Error Propagation in a 14-Bit Floating-Point Number

floating-point systems behave the same way. There is always some degree of error involved when representing real numbers in a finite system, no matter how large we make that system. Even the smallest error can have catastrophic results, particularly when computers are used to control physical events such as in military and medical applications. The challenge to computer scientists is to find efficient algorithms for controlling such errors within the bounds of performance and economics.

2.5.4 The IEEE-754 Floating-Point Standard

The floating-point model we have been using in this section is designed for simplicity and conceptual understanding. We could extend this model to include whatever number of bits we wanted. Until the 1980s, these kinds of decisions were purely arbitrary, resulting in numerous incompatible representations across various manufacturers' systems. In 1985, the IEEE published a floating-point standard for both single- and double-precision floating-point numbers. This standard is officially known as IEEE-754 (1985) and includes two formats: **single precision** and **double precision**. The IEEE-754 standard not only defines binary floating-point representations, but also specifies basic operations, exception conditions, conversions, and arithmetic. Another standard, IEEE 854-1987, provides similar specifications for decimal arithmetic. In 2008, IEEE revised the 754 standard, and it became known as IEEE 754-2008. It carried over the single and double precision from 754, and added support for decimal arithmetic and formats, superseding both 754 and 854. We discuss only the single and double representation for floating-point numbers.

The IEEE-754 single-precision standard uses an excess 127 bias over an 8-bit exponent. The significand assumes an implied 1 *to the left* of the radix point and is 23 bits. This implied 1 is referred to as the **hidden bit** or **hidden 1** and allows an actual significand of 23 + 1 = 24 bits. With the sign bit included, the total word size is 32 bits, as shown in Figure 2.2.

FIGURE 2.2 IEEE-754 Single-Precision Floating-Point Representation

We mentioned earlier that IEEE-754 makes an exception to the rule of normalization. Because this standard assumes an implied 1 to the left of the radix point, the leading bit in the significand can indeed be 0. For example, the number $5.5 = 101.1 = .1011 \times 2^3$. IEEE-754 assumes an implied 1 to the left of the radix point and thus represents 5.5 as 1.011×2^2. Because the 1 is implied, the significand is 011 and does not begin with a 1.

Table 2.4 shows the single-precision representation of several floating-point numbers, including some special 1s. Basically, the value represented by the bit sequence depends on the value of the exponent. The exponent can represent **normalized** numbers (the most common), **denormalized** values (those with all 0s in the exponent field, explained below), and special values (indicated when the exponent field is all 1s). These different cases are required to represent necessary values. For example, note that 0 is not directly representable in the given format, because of a required hidden bit in the significand. Therefore, 0 is a special value denoted using an exponent of all 0s and a significand of all 0s. IEEE-754 does allow for both –0 and +0, although they are equal values. For this reason, programmers should use caution when comparing a floating-point value to 0.

When the exponent is 255 (all 1s), the quantity represented is ± infinity (which has a 0 significand and is used to represent values that overflow because the resulting value is too large to be represented in this format) or "not a number" (which has a nonzero significand and is used to represent a value that is not a real number, such as the square root of a negative number, or as an error indicator, such as in a "division by 0" error).

Under the IEEE-754 standard, most numeric values are normalized and have an implicit leading 1 in their significands (that is assumed to be to the left of the radix point) and the exponent is not all 0s or all 1s. As an example, consider

Floating-Point Number	Single-Precision Representation
1.0	0 01111111 00000000000000000000000
0.5	0 01111110 00000000000000000000000
19.5	0 10000011 00111000000000000000000
–3.75	1 10000000 11100000000000000000000
Zero	0 00000000 00000000000000000000000
± Infinity	0/1 11111111 00000000000000000000000
NaN	0/1 11111111 any nonzero significand
Denormalized Number	0/1 00000000 any nonzero significand

TABLE 2.4 Some Example IEEE-754 Single-Precision Floating-Point Numbers

the value $15212_{10} = 11101101101100_2 = 1.1101101101100_2 \times 2^{13}$. This is represented as:

Another important convention is when the exponent is all 0s but the significand is nonzero. This represents a **denormalized** number in which there is no hidden bit assumed. These numbers are typically used to represent values that are close to 0.

The largest magnitude value we can represent (forget the sign for the time being) with the single-precision floating-point format is $2^{127} \times 1.11111111111\ 111111111111_2$ (let's call this value *MAX*). We can't use an exponent of all 1s because that is reserved for NaN. The smallest magnitude number we can represent is $2^{-127} \times .00000000000000000000001_2$ (let's call this value *MIN*). We can use an exponent of all 0s (which means the number is denormalized) because the significand is nonzero (and represents 2^{-23}). Due to the preceding special values and the limited number of bits, there are four numerical ranges that single-precision floating-point numbers cannot represent: negative numbers less than MAX (negative overflow); negative numbers greater than MIN (negative underflow); positive numbers less than +MIN (positive underflow); and positive numbers greater than +MAX (positive overflow). Figure 2.3 represents a way to visualize the overall scheme of how the floating-point encoding works.

Double-precision numbers use a signed 64-bit word consisting of an 11-bit exponent and a 52-bit significand. The bias is 1023. The range of numbers that can be represented in the IEEE double-precision model is shown in Figure 2.4. NaN is indicated when the exponent is 2047. Representations for 0 and infinity correspond to the single-precision model.

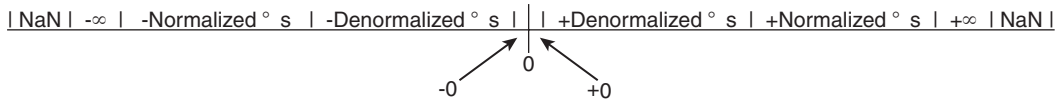


FIGURE 2.3 How Floating Point Encoding Works

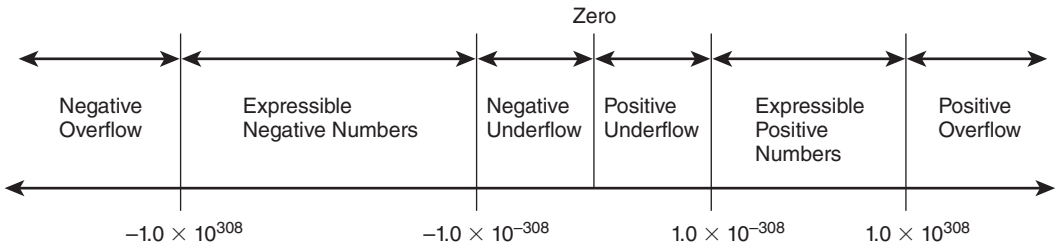


FIGURE 2.4 Range of IEEE-754 Double-Precision Numbers

At a slight cost in performance, most FPUs use only the 64-bit model so that only one set of specialized circuits needs to be designed and implemented.

Virtually every recently designed computer system has adopted the IEEE-754 floating-point model. Unfortunately, by the time this standard came along, many mainframe computer systems had established their own floating-point systems. Changing to the newer system has taken decades for well-established architectures such as IBM mainframes, which now support both their traditional floating-point system and IEEE-754. Before 1998, however, IBM systems had been using the same architecture for floating-point arithmetic that the original System/360 used in 1964. One would expect that both systems will continue to be supported, owing to the substantial amount of older software that is running on these systems.

2.5.5 Range, Precision, and Accuracy

When discussing floating-point numbers it is important to understand the terms *range, precision,* and *accuracy*. Range is very straightforward, because it represents the interval from the smallest value in a given format to the largest value in that same format. For example, the range of 16-bit two's complement integers is –32768 to +32767. The range of IEEE-754 double-precision floating-point numbers is given in Figure 2.4. Even with this large range, we know there are infinitely many numbers that do not exist within the range specified by IEEE-754. The reason floating-point numbers work at all is that there will always be a number in this range that is *close to* the number you want.

People have no problem understanding range, but accuracy and precision are often confused with each other. *Accuracy* refers to how close a number is to its true value; for example, we can't represent 0.1 in floating point, but we can find a number in the range that is relatively close, or reasonably accurate, to 0.1. *Precision*, on the other hand, deals with how much information we have about a value and the amount of information used to represent the value. 1.666 is a number with four decimal digits of precision; 1.6660 is the same exact number with five decimal digits of precision. The second number is not more accurate than the first.

Accuracy must be put into context—to know how accurate a value is, one must know how close it is to its intended target or "true value." We can't look at two numbers and immediately declare that the first is more accurate than the second simply because the first has more digits of precision.

Although they are separate, accuracy and precision are related. Higher precision often allows a value to be more accurate, but that is not always the case. For example, we can represent the value 1 as an integer, a single-precision floating point, or a double-precision floating point, but each is equally (exactly) accurate. As another example, consider 3.13333 as an estimate for pi. It has six digits of precision, yet is accurate to only two digits. Adding more precision will do nothing to increase the accuracy.

On the other hand, when multiplying 0.4×0.3, our accuracy depends on our precision. If we allow only one decimal place for precision, our result is 0.1 (which is close to, but not exactly, the product). If we allow two decimal places of precision, we get 0.12, which accurately reflects the answer.

It is important to note that because any floating-point representation has a limited number of values that can be represented, when performing floating-point arithmetic the result is often the "closest value" that the machine can represent in the given format. This closest value is found by rounding. IEEE has five different rounding modes that it uses to perform rounding. The default mode, **rounding to nearest**, finds the closest value; the other modes can be used for upper and lower bounds. Rounding to nearest has two options: round to nearest, ties to even (if the number is halfway between values, it is rounded to an even value) and round to nearest, ties away from 0 (if the number is halfway between two values, it is rounded to the value furthest away from 0). The three remaining modes are **directed modes**, which allow directed rounding. Rounding can be done toward 0, toward positive infinity, or toward negative infinity. These modes are shown in Figure 2.5.

2.5.6 Additional Problems with Floating-Point Numbers

We have seen that floating-point numbers can overflow and underflow. In addition, we know that a floating-point number may not exactly represent the value we wish, as is the case with the rounding error that occurs with the binary floating-point representation for the decimal number 0.1. As we have seen, these rounding errors can propagate, resulting in substantial problems.

Although rounding is undesirable, it is understandable. In addition to this rounding problem, however, floating-point arithmetic differs from real number arithmetic in two relatively disturbing, and not necessarily intuitive, ways. First, floating-point arithmetic is not always associative. This means that for three floating-point numbers a, b, and c,

$$(a + b) + c \circ a + (b + c)$$

The same holds true for associativity under multiplication. Although in many cases the left-hand side will equal the right-hand side, there is no guarantee. Floating-point arithmetic is also not distributive:

$$a \times (b + c) \neq ab + ac$$

Although results can vary depending on compiler (we used Gnu C), declaring the doubles $a = 0.1$, $b = 0.2$, and $c = 0.3$ illustrates the above inequalities nicely. We encourage you to find three additional floating-point numbers to illustrate that floating-point arithmetic is neither associative nor distributive.

	Value			
Rounding Mode	**+9.5**	**+10.5**	**–9.5**	**–10.5**
Nearest ties to even	+10.0	+10.0	–10.0	–10.0
Nearest ties away from 0	+10.0	+11.0	–10.0	–11.0
Directed toward 0	+9.0	+10.0	–9.0	–10.0
Directed toward $+\infty$	+10.0	+11.0	–9.0	–10.0
Directed toward $-\infty$	+9.0	+10.0	–10.0	–11.0

FIGURE 2.5 Floating Point Rounding Modes

What does this all mean to you as a programmer? Programmers should use extra care when using the equality operator on floating-point numbers. This implies that they should be avoided in controlling looping structures such as `do...while` and `for` loops. It is good practice to declare a "nearness to x" epsilon (e.g., `epsilon` = 1.0×10^{-20}) and then test an absolute value.

For example, instead of using:

```
if x = 2 then...
```

it is better to use:

```
if(abs(x - 2) < epsilon) then...\\ It's close enough if we've
                                 \\ defined epsilon correctly!
```

Floating-Point Ops or Oops?

In this chapter, we have introduced floating-point numbers and the means by which computers represent them. We have touched upon floating-point rounding errors (studies in numerical analysis will provide further depth on this topic) and the fact that floating-point numbers don't obey the standard associative and distributive laws. But just how serious are these issues? To answer this question, we discuss three major floating-point blunders.

In 1994, when Intel introduced the Pentium microprocessor, number crunchers around the world noticed something weird was happening. Calculations involving double-precision divisions and certain bit patterns were producing incorrect results. Although the flawed chip was slightly inaccurate for some pairs of numbers, other instances were more extreme. For example, if $x = 4{,}195{,}835$ and $y = 3{,}145{,}727$, finding $z = x - (x/y) \times y$ should produce a z of 0. The Intel 286, 386, and 486 chips gave exactly that result. Even taking into account the possibility of floating-point round-off error, the value of z should have been about 9.3×10^{-10}. But on the new Pentium, z was equal to 256!

Once Intel was informed of the problem, research and testing revealed the flaw to be an omission in the chip's design. The Pentium was using the radix-4 SRT algorithm for speedy division, which necessitated a 1066-element table. Once implemented in silicon, 5 of those table entries were 0 and should have been +2.

Although the Pentium bug was a public relations debacle for Intel, it was not a catastrophe for those using the chip. In fact, it was a minor thing compared to the programming mistakes with floating-point numbers that have resulted in disasters in areas from offshore oil drilling to stock markets to missile defense. The list of actual disasters that resulted from floating-point errors is very long. The following two instances are among the worst of them.

During the Persian Gulf War of 1991, the United States relied on Patriot missiles to track and intercept cruise missiles and Scud missiles. One of these missiles failed to track an incoming Scud missile, allowing the Scud to hit an American army barracks,

killing 28 people and injuring many more. After an investigation, it was determined that the failure of the Patriot missile was due to using too little precision to allow the missile to accurately determine the incoming Scud velocity.

The Patriot missile uses radar to determine the location of an object. If the internal weapons control computer identifies the object as something that should be intercepted, calculations are performed to predict the air space in which the object should be located at a specific time. This prediction is based on the object's known velocity and time of last detection.

The problem was in the clock, which measured time in tenths of seconds. But the time since boot was stored as an integer number of seconds (determined by multiplying the elapsed time by 1/10). For predicting where an object would be at a specific time, the time and velocity needed to be real numbers. It was no problem to convert the integer to a real number; however, using 24-bit registers for its calculations, the Patriot was limited in the precision of this operation. The potential problem is easily seen when one realizes that 1/10 in binary is:

```
0.0001100110011001100110011001100...
```

When the elapsed time was small, this "chopping error" was insignificant and caused no problems. The Patriot was designed to be on for only a few minutes at a time, so this limit of 24-bit precision would be of no consequence. The problem was that during the Gulf War, the missiles were on for days. The longer a missile was on, the larger the error became, and the more probable that the inaccuracy of the prediction calculation would cause an unsuccessful interception. And this is precisely what happened on February 25, 1991, when a failed interception resulted in 28 people killed—a failed interception caused by loss of precision (required for accuracy) in floating-point numbers. It is estimated that the Patriot missile had been operational for about 100 hours, introducing a rounding error in the time conversion of about 0.34 seconds, which translates to approximately half a kilometer of travel for a Scud missile.

Designers were aware of the conversion problem well before the incident occurred. However, deploying new software under wartime conditions is anything but trivial. Although the new software would have fixed the bug, field personnel could have simply rebooted the systems at specific intervals to keep the clock value small enough so that 24-bit precision would have been sufficient.

One of the most famous examples of a floating-point numeric disaster is the explosion of the Ariane 5 rocket. On June 4, 1996, the unmanned Ariane 5 was launched by the European Space Agency. Forty seconds after liftoff, the rocket exploded, scattering a $500 million cargo across parts of French Guiana. Investigation revealed perhaps one of the most devastatingly careless but efficient software bugs in the annals of computer science—a floating-point conversion error. The rocket's inertial reference system converted a 64-bit floating-point number (dealing with the horizontal velocity of the rocket) to a 16-bit signed integer. However, the particular 64-bit floating-point number to be converted was larger than

32,767 (the largest integer that can be stored in 16-bit signed representation), so the conversion process failed. The rocket tried to make an abrupt course correction for a wrong turn that it had never taken, and the guidance system shut down. Ironically, when the guidance system shut down, control reverted to a backup unit installed in the rocket in case of just such a failure, but the backup system was running the same flawed software.

It seems obvious that a 64-bit floating-point number could be much larger than 32,767, so how did the rocket programmers make such a glaring error? They decided the velocity value would never get large enough to be a problem. Their reasoning? It had never gotten too large before. Unfortunately, this rocket was faster than all previous rockets, resulting in a larger velocity value than the programmers expected. One of the most serious mistakes a programmer can make is to accept the old adage "But we've always done it that way."

Computers are everywhere—in our washing machines, our televisions, our microwaves, even our cars. We certainly hope the programmers who work on computer software for our cars don't make such hasty assumptions. With approximately 15 to 60 microprocessors in all new cars that roll off the assembly line and innumerable processors in commercial aircraft and medical equipment, a deep understanding of floating-point anomalies can quite literally be a lifesaver.

2.6 CHARACTER CODES

We have seen how digital computers use the binary system to represent and manipulate numeric values. We have yet to consider how these internal values can be converted to a form that is meaningful to humans. The manner in which this is done depends on both the coding system used by the computer and how the values are stored and retrieved.

2.6.1 Binary-Coded Decimal

For many applications, we need the exact binary equivalent of the decimal system, which means we need an encoding for individual decimal digits. This is precisely the case in many business applications that deal with money—we can't afford the rounding errors that occur when we convert real numbers to floating point during financial transactions!

Binary-coded decimal (BCD) is very common in electronics, particularly those that display numerical data, such as alarm clocks and calculators. BCD encodes each digit of a decimal number into a 4-bit binary form. Each decimal digit is individually converted to its binary equivalent, as seen in Table 2.5. For example, to encode 146, the decimal digits are replaced by 0001, 0100, and 0110, respectively.

Because most computers use bytes as the smallest unit of access, most values are stored in 8 bits, not 4. That gives us two choices for storing 4-bit BCD digits. We can ignore the cost of extra bits and pad the high-order nibbles

with 0s (or 1s), forcing each decimal digit to be replaced by 8 bits. Using this approach, padding with 0s, 146 would be stored as 00000001 00000100 00000110. Clearly, this approach is quite wasteful. The second approach, called **packed BCD**, stores two digits per byte. Packed decimal format allows numbers to be signed, but instead of putting the sign at the beginning, the sign is stored at the end. The standard values for this "sign digit" are 1100 for +, 1101 for −, and 1111 to indicate that the value is unsigned (see Table 2.5). Using packed decimal format, +146 would be stored as 00010100 01101100. Padding would still be required for an even number of digits. Note that if a number has a decimal point (as with monetary values), this is not stored in the BCD representation of the number and must be retained by the application program.

Another variation of BCD is **zoned-decimal format**. Zoned-decimal representation stores a decimal digit in the low-order nibble of each byte, which is exactly the same as unpacked decimal format. However, instead of padding the high-order nibbles with 0s, a specific pattern is used. There are two choices for the high-order nibble, called the numeric **zone**. **EBCDIC zoned-decimal format** requires the zone to be all 1s (hexadecimal F). **ASCII zoned-decimal format** requires the zone to be 0011 (hexadecimal 3). (See the next two sections for detailed explanations of EBCDIC and ASCII.) Both formats allow for signed numbers (using the sign digits found in Table 2.5) and typically expect the sign to be located in the high-order nibble of the least significant byte (although the sign could be a completely separate byte). For example, +146 in EBCDIC zoned-decimal format is 11110001 11110100 11000110 (note that the high-order nibble of the last byte is the sign). In ASCII zoned-decimal format, +146 is 00110001 00110100 11000110.

Digit	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
Zones	
1111	Unsigned
1100	Positive
1101	Negative

TABLE 2.5 Binary-Coded Decimal

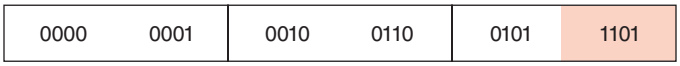
Note from Table 2.5 that six of the possible binary values are not used—1010 through 1111. Although it may appear that nearly 40% of our values are going to waste, we are gaining a considerable advantage in accuracy. For example, the number 0.3 is a repeating decimal when stored in binary. Truncated to an 8-bit fraction, it converts back to 0.296875, giving us an error of approximately 1.05%. In EBCDIC zoned-decimal BCD, the number is stored directly as 1111 0011 (we are assuming the decimal point is implied by the data format), giving no error at all.

EXAMPLE 2.37 Represent −1265 using packed BCD and EBCDIC zoned decimal.

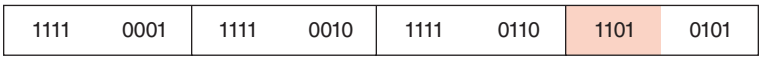
The 4-bit BCD representation for 1265 is:

0001 0010 0110 0101

Adding the sign after the low-order digit and padding the high-order bit with 0000, we have:



The EBCDIC zoned-decimal representation requires 4 bytes:



The sign bit is shaded in both representations.

2.6.2 EBCDIC

Before the development of the IBM System/360, IBM had used a 6-bit variation of BCD for representing characters and numbers. This code was severely limited in how it could represent and manipulate data; in fact, lowercase letters were not part of its repertoire. The designers of the System/360 needed more information processing capability as well as a uniform manner in which to store both numbers and data. To maintain compatibility with earlier computers and peripheral equipment, the IBM engineers decided that it would be best to simply expand BCD from 6 bits to 8 bits. Accordingly, this new code was called **Extended Binary Coded Decimal Interchange Code (EBCDIC)**. IBM continues to use EBCDIC in IBM mainframe and midrange computer systems; however, IBM's AIX operating system (found on the RS/6000 and its successors) and operating systems for the IBM PC use ASCII. The EBCDIC code is shown in Table 2.6 in zone-digit form. Characters are represented by appending digit bits to zone bits. For example, the character *a* is 1000 0001 and the digit 3 is 1111 0011 in EBCDIC. Note that the only difference between uppercase and lowercase characters is in bit position 2, making a translation from uppercase to lowercase (or vice versa) a simple matter of flipping one bit. Zone bits also make it easier for a programmer to test the validity of input data.

2.6.3 ASCII

While IBM was busy building its iconoclastic System/360, other equipment makers were trying to devise better ways for transmitting data between systems. The **American Standard Code for Information Interchange (ASCII)** is one outcome of those efforts. ASCII is a direct descendant of the coding schemes used for decades by teletype (telex) devices. These devices used a 5-bit (Murray) code that was derived from the Baudot code, which was invented in the 1880s. By the early 1960s, the limitations of the 5-bit codes were becoming apparent. The International Organization for Standardization devised a 7-bit coding scheme that it called *International Alphabet Number 5*. In 1967, a derivative of this alphabet became the official standard that we now call *ASCII*.

As you can see in Table 2.7, ASCII defines codes for 32 control characters, 10 digits, 52 letters (uppercase and lowercase), 32 special characters (such as $ and #), and the space character. The high-order (eighth) bit was intended to be used for parity.

Parity is the most basic of all error-detection schemes. It is easy to implement in simple devices like teletypes. A parity bit is turned "on" or "off" depending on whether the sum of the other bits in the byte is even or odd. For example, if we decide to use even parity and we are sending an ASCII *A*, the lower 7 bits are 100 0001. Because the sum of the bits is even, the parity bit would be set to "off" and we would transmit 0100 0001. Similarly, if we transmit an ASCII *C*, 100 0011, the parity bit would be set to "on" before we sent the 8-bit byte, 1100 0011. Parity can be used to detect only single-bit errors. We will discuss more sophisticated error-detection methods in Section 2.7.

To allow compatibility with telecommunications equipment, computer manufacturers gravitated toward the ASCII code. As computer hardware became more reliable, however, the need for a parity bit began to fade. In the early 1980s, microcomputer and microcomputer-peripheral makers began to use the parity bit to provide an "extended" character set for values between 128_{10} and 255_{10}.

Depending on the manufacturer, the higher-valued characters could be anything from mathematical symbols to characters that form the sides of boxes to foreign-language characters such as ñ. Unfortunately, no number of clever tricks can make ASCII a truly international interchange code.

2.6.4 Unicode

Both EBCDIC and ASCII were built around the Latin alphabet. As such, they are restricted in their abilities to provide data representation for the non-Latin alphabets used by the majority of the world's population. As all countries began using computers, each was devising codes that would most effectively represent their native languages. None of these was necessarily compatible with any others, placing yet another barrier in the way of the emerging global economy.

	Digit															
Zone	**0000**	**0001**	**0010**	**0011**	**0100**	**0101**	**0110**	**0111**	**1000**	**1001**	**1010**	**1011**	**1100**	**1101**	**1110**	**1111**
0000	NUL	SOH	STX	ETX	PF	HT	LC	DEL		RLF	SMM	VT	FF	CR	SO	SI
0001	DLE	DC1	DC2	TM	RES	NL	BS	IL	CAN	EM	CC	CU1	IFS	IGS	IRS	IUS
0010	DS	SOS	FS		BYP	LF	ETB	ESC			SM	CU2		ENQ	ACK	BEL
0011			SYN		PN	RS	UC	EOT				CU3	DC4	NAK		SUB
0100	SP										[	.	<	(	+	!
0101	&										]	$	*	)	;	^
0110	–	/								'	:	#	@	'	=	"
0111																
1000		a	b	c	d	e	f	g	h	i						
1001		j	k	l	m	n	o	p	q	r						
1010		~	s	t	u	v	w	x	y	z						
1011																
1100	{	A	B	C	D	E	F	G	H	I						
1101	}	J	K	L	M	N	O	P	Q	R						
1110	/		S	T	U	V	W	X	Y	Z						
1111	0	1	2	3	4	5	6	7	8	9						

Abbreviations					
NUL	Null	TM	Tape mark	ETB	End of transmission block
SOH	Start of heading	RES	Restore	ESC	Escape
STX	Start of text	NL	New line	SM	Set mode
ETX	End of text	BS	Backspace	CU2	Customer use 2
PF	Punch off	IL	Idle	ENQ	Enquiry
HT	Horizontal tab	CAN	Cancel	ACK	Acknowledge
LC	Lowercase	EM	End of medium	BEL	Ring the bell (beep)
DEL	Delete	CC	Cursor control	SYN	Synchronous idle
RLF	Reverse line feed	CU1	Customer use 1	PN	Punch on
SMM	Start manual message	IFS	Interchange file separator	RS	Record separator
VT	Vertical tab	IGS	Interchange group separator	UC	Uppercase
FF	Form feed	IRS	Interchange record separator	EOT	End of transmission
CR	Carriage return	IUS	Interchange unit separator	CU3	Customer use 3
SO	Shift out	DS	Digit select	DC4	Device control 4
SI	Shift in	SOS	Start of significance	NAK	Negative acknowledgment
DLE	Data link escape	FS	Field separator	SUB	Substitute
DC1	Device control 1	BYP	Bypass	SP	Space
DC2	Device control 2	LF	Line feed		

TABLE 2.6 The EBCDIC Code (Values Given in Binary Zone-Digit Format)

0	NUL	16	DLE	32		48	0	64	@	80	P	96	`	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	'	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	HT	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	FS	44	’	60	<	76	L	92	\	108	l	124	|
13	CR	29	GS	45	–	61	–	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

Abbreviations:

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

TABLE 2.7 The ASCII Code (Values Given in Decimal)

In 1991, before things got too far out of hand, a consortium of industry and public leaders was formed to establish a new international information exchange code called *Unicode*. This group is appropriately called the Unicode Consortium.

Unicode is a 16-bit alphabet that is downward compatible with ASCII and the Latin-1 character set. It is conformant with the ISO/IEC 10646-1 international alphabet. Because the base coding of Unicode is 16 bits, it has the capacity to encode the majority of characters used in every language of the world. If this weren't enough, Unicode also defines an extension mechanism that will allow for the coding of an additional million characters. This is sufficient to provide codes for every written language in the history of civilization.

The Unicode codespace consists of five parts, as shown in Table 2.8. A full Unicode-compliant system will also allow formation of composite characters from the individual codes, such as the combination of ´ and A to form Á. The algorithms used for these composite characters, as well as the Unicode extensions, can be found in the references at the end of this chapter.

Although Unicode has yet to become the exclusive alphabet of American computers, most manufacturers are including at least some limited support for it in their systems. Unicode is currently the default character set of the Java programming language. Ultimately, the acceptance of Unicode by all manufacturers will depend on how aggressively they wish to position themselves as international players and how inexpensively disk drives can be produced to support an alphabet with double the storage requirements of ASCII or EBCDIC.

Character Types	Character Set Description	Number of Characters	Hexadecimal Values
Alphabets	Latin, Cyrillic, Greek, etc.	8192	0000 to 1FFF
Symbols	Dingbats, mathematical, etc.	4096	2000 to 2FFF
CJK	Chinese, Japanese, and Korean phonetic symbols and punctuation	4096	3000 To 3FFF
Han	Unified Chinese, Japanese, and Korean	40,960	4000 to DFFF
	Expansion or spillover from Han	4096	E000 to EFFF
User defined		4095	F000 to FFFE

TABLE 2.8 Unicode Codespace

2.7 ERROR DETECTION AND CORRECTION

No communications channel or storage medium can be completely error-free. It is a physical impossibility. As transmission rates are increased, bit timing gets tighter. As more bits are packed per square millimeter of storage, magnetic flux densities increase. Error rates increase in direct proportion to the number of bits per second transmitted, or the number of bits per square millimeter of magnetic storage.

In Section 2.6.3, we mentioned that a parity bit could be added to an ASCII byte to help determine whether any of the bits had become corrupted during transmission. This method of error detection is limited in its effectiveness: Simple parity can detect only an odd number of errors per byte. If two errors occur, we are helpless to detect a problem. Nonsense could pass for good data. If such errors occur in sending financial information or program code, the effects can be disastrous.

As you read the sections that follow, you should keep in mind that just as it is impossible to create an error-free medium, it is also impossible to detect or correct 100% of all errors that *could* occur in a medium. Error detection and correction is yet another study in the trade-offs that one must make in designing computer systems. The well-constructed error control system is therefore a system where a "reasonable" number of the "reasonably" expected errors can be detected or corrected within the bounds of "reasonable" economics. (Note: The word *reasonable* is implementation-dependent.)

2.7.1 Cyclic Redundancy Check

Checksums are used in a wide variety of coding systems, from bar codes to International Standard Book Numbers. These are self-checking codes that will quickly indicate whether the preceding digits have been misread. A **cyclic redundancy check (CRC)** is a type of checksum used primarily in data communications that determines whether an error has occurred within a large block or stream of information bytes. The larger the block to be checked, the larger the checksum must be to provide adequate protection. Checksums and CRCs are types of **systematic error detection** schemes, meaning that the error-checking bits are appended to the original information byte. The group of error-checking bits is called a **syndrome**. The original information byte is unchanged by the addition of the error-checking bits.

The word *cyclic* in *cyclic redundancy check* refers to the abstract mathematical theory behind this error control system. Although a discussion of this theory is beyond the scope of this text, we can demonstrate how the method works to aid in your understanding of its power to economically detect transmission errors.

Arithmetic Modulo 2

You may be familiar with integer arithmetic taken over a modulus. Twelve-hour clock arithmetic is a modulo 12 system that you use every day to tell time. When

we add 2 hours to 11:00, we get 1:00. Arithmetic modulo 2 uses two binary operands with no borrows or carries. The result is likewise binary and is also a member of the modulo 2 system. Because of this closure under addition, and the existence of identity elements, mathematicians say that this modulo 2 system forms an **algebraic field**.

The addition rules are as follows:

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 0$$

EXAMPLE 2.38 Find the sum of 1011_2 and 110_2 modulo 2.

$$\begin{array}{r} 1011 \\ +\,110 \\ \hline 1101_2 \pmod 2 \end{array}$$

This sum makes sense only in modulo 2.

Modulo 2 division operates through a series of partial sums using the modulo 2 addition rules. Example 2.39 illustrates the process.

EXAMPLE 2.39 Find the quotient and remainder when 1001011_2 is divided by 1011_2.

$1011\overline{)1001011}$	1. Write the divisor directly beneath the first bit of the dividend.
$\underline{1011}$	2. Add these numbers using modulo 2.
0010	3. Bring down bits from the dividend so that the first 1 of the difference can align with the first 1 of the divisor.
001001	4. Copy the divisor as in Step 1.
$\underline{1011}$	5. Add as in Step 2.
0010	6. Bring down another bit.
00101	7. 101_2 is not divisible by 1011_2, so this is the remainder.

The quotient is 1010_2.

Arithmetic operations over the modulo 2 field have polynomial equivalents that are analogous to polynomials over the field of integers. We have seen how positional number systems represent numbers in increasing powers of a radix, for example:

$$1011_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0.$$

By letting $X = 2$, the binary number 1011_2 becomes shorthand for the polynomial:

$$1 \times X^3 + 0 \times X^2 + 1 \times X^1 + 1 \times X^0.$$

The division performed in Example 2.39 then becomes the polynomial operation:

$$\frac{X^6 + X^3 + X + 1}{X^3 + X + 1}$$

Calculating and Using CRCs

With that lengthy preamble behind us, we can now proceed to show how CRCs are constructed. We will do this by example:

1. Let the information byte $I = 1001011_2$. (Any number of bytes can be used to form a message block.)
2. The sender and receiver agree upon an arbitrary binary pattern, say, $P = 1011_2$. (Patterns beginning and ending with 1 work best.)
3. Shift I to the left by one less than the number of bits in P, giving a new $I = 1001011000_2$.
4. Using I as a dividend and P as a divisor, perform the modulo 2 division (as shown in Example 2.39). We ignore the quotient and note that the remainder is 100_2. The remainder is the actual CRC checksum.
5. Add the remainder to I, giving the message M:

   $$1001011000_2 + 100_2 = 1001011100_2$$

6. M is decoded and checked by the message receiver using the reverse process. Only now P divides M exactly:

```
      1010100
1011)1001011100
     1011
     001001
        1011
        0010
        001011
           1011
           0000
```

Note: The reverse process would include appending the remainder.

A remainder other than 0 indicates that an error has occurred in the transmission of M. This method works best when a large prime polynomial is used. There are four standard polynomials used widely for this purpose:

- CRC-CCITT (ITU-T): $X^{16} + X^{12} + X^5 + 1$
- CRC-12: $X^{12} + X^{11} + X^3 + X^2 + X + 1$
- CRC-16 (ANSI): $X^{16} + X^{15} + X^2 + 1$
- CRC-32: $X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^8 + X^7 + X^5 + X^4 + X + 1$

CRC-CCITT, CRC-12, and CRC-16 operate over pairs of bytes; CRC-32 uses 4 bytes, which is appropriate for systems operating on 32-bit words. It has been proven that CRCs using these polynomials can detect more than 99.8% of all single-bit errors.

CRCs can be implemented effectively using lookup tables as opposed to calculating the remainder with each byte. The remainder generated by each possible input bit pattern can be "burned" directly into communications and storage electronics. The remainder can then be retrieved using a 1-cycle lookup as compared to a 16- or 32-cycle division operation. Clearly, the trade-off is in speed versus the cost of more complex control circuitry.

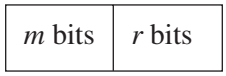
2.7.2 Hamming Codes

Data communications channels are simultaneously more error-prone and more tolerant of errors than disk systems. In data communications, it is sufficient to have only the ability to detect errors. If a communications device determines that a message contains an erroneous bit, all it has to do is request retransmission. Storage systems and memory do not have this luxury. A disk can sometimes be the sole repository of a financial transaction or other collection of nonreproducible real-time data. Storage devices and memory must therefore have the ability to not only detect but to correct a reasonable number of errors.

Error-recovery coding has been studied intensively over the past century. One of the most effective codes—and the oldest—is the Hamming code. **Hamming codes** are an adaptation of the concept of parity, whereby error detection and correction capabilities are increased in proportion to the number of parity bits added to an information word. Hamming codes are used in situations where random errors are likely to occur. With random errors, we assume that each bit failure has a fixed probability of occurrence independent of other bit failures. It is common for computer memory to experience such errors, so in our following discussion, we present Hamming codes in the context of memory bit error detection and correction.

We mentioned that Hamming codes use parity bits, also called **check bits** or **redundant bits**. The memory word itself consists of m bits, but r redundant bits are added to allow for error detection and/or correction. Thus, the final word, called a **code word**, is an n-bit unit containing m data bits and r check bits.

There exists a unique code word consisting of $n = m + r$ bits for each data word as follows:



The number of bit positions in which two code words differ is called the **Hamming distance** of those two code words. For example, if we have the following two code words:

```
1 0 0 0 1 0 0 1
1 0 1 1 0 0 0 1
    * * *
```

we see that they differ in 3 bit positions (marked by *), so the Hamming distance of these two code words is 3. (Please note that we have not yet discussed how to create code words; we will do that shortly.)

The Hamming distance between two code words is important in the context of error detection. If two code words are a Hamming distance d apart, d single-bit errors are required to convert one code word to the other, which implies that this type of error would not be detected. Therefore, if we wish to create a code that guarantees detection of all single-bit errors (an error in only 1 bit), all pairs of code words must have a Hamming distance of at least 2. If an n-bit word is not recognized as a legal code word, it is considered an error.

Given an algorithm for computing check bits, it is possible to construct a complete list of legal code words. The smallest Hamming distance found among all pairs of the code words in this code is called the **minimum Hamming distance** for the code. The minimum Hamming distance of a code, often signified by the notation ***D(min)***, determines its error detecting and correcting capability. Stated succinctly, for any code word X to be received as another valid code word Y, at least $D(min)$ errors must occur in X. So, to detect k (or fewer) single-bit errors, the code must have a Hamming distance of $D(min) = k + 1$. Hamming codes can always detect $D(min) - 1$ errors and correct $\lfloor (D(min) - 1)/2 \rfloor$ errors.[1] Accordingly, the Hamming distance of a code must be at least $2k + 1$ in order for it to be able to correct k errors.

Code words are constructed from information words using r parity bits. Before we continue the discussion of error detection and correction, let's consider a simple example. The most common error detection uses a single parity bit appended to the data (recall the discussion on ASCII character representation). A single-bit error in any bit of the code word produces the wrong parity.

[1]The $\lfloor\ \rfloor$ brackets denote the integer floor function, which is the largest integer that is smaller than or equal to the enclosed quantity. For example, $\lfloor 8.3 \rfloor = 8$ and $\lfloor 8.9 \rfloor = 8$.

EXAMPLE 2.40 Assume a memory with 2 data bits and 1 parity bit (appended at the end of the code word) that uses even parity (so the number of 1s in the code word must be even). With 2 data bits, we have a total of four possible words. We list here the data word, its corresponding parity bit, and the resulting code word for each of these four possible words:

Data Word	Parity Bit	Code Word
00	0	000
01	1	011
10	1	101
11	0	110

The resulting code words have 3 bits. However, using 3 bits allows for 8 different bit patterns, as follows (valid code words are marked with an *):

000*	100
001	101*
010	100*
011*	111

If the code word 001 is encountered, it is invalid and thus indicates that an error has occurred somewhere in the code word. For example, suppose the correct code word to be stored in memory is 011, but an error produces 001. This error can be detected, but it cannot be corrected. It is impossible to determine exactly how many bits have been flipped and exactly which 1s are in error. Error-correcting codes require more than a single parity bit, as we see in the following discussion.

What happens in the above example if a valid code word is subject to two-bit errors? For example, suppose the code word 011 is converted into 000. This error is not detected. If you examine the code in the above example, you will see that *D(min)* is 2, which implies that this code is guaranteed to detect only single-bit errors.

We have already stated that the error detecting and correcting capabilities of a code are dependent on *D(min)*, and from an error detection point of view, we have seen this relationship exhibited in Example 2.40. Error correction requires the code to contain additional redundant bits to ensure a minimum Hamming distance $D(min) = 2k + 1$ if the code is to detect and correct k errors. This Hamming distance guarantees that all legal code words are far enough apart that even with k changes, the original invalid code word is closer to one unique valid code word. This is important because the method used in error correction is to change the invalid code word into the valid code word that differs in the fewest number of bits. This idea is illustrated in Example 2.41.

EXAMPLE 2.41 Suppose we have the following code (do not worry at this time about how this code was generated; we will address this issue shortly):

0 0 0 0 0
0 1 0 1 1
1 0 1 1 0
1 1 1 0 1

First, let's determine *D(min)*. By examining all possible pairs of code words, we discover that the minimum Hamming distance *D(min)* = 3. Thus, this code can detect up to two errors and correct one single-bit error. How is correction handled? Suppose we read the invalid code word 10000. There must be at least one error because this does not match any of the valid code words. We now determine the Hamming distance between the observed code word and each legal code word: It differs in 1 bit from the first code word, 4 from the second, 2 from the third, and 3 from the last, resulting in a **difference vector** of [1,4,2,3]. To make the correction using this code, we automatically correct to the legal code word closest to the observed word, resulting in a correction to 00000. Note that this "correction" is not necessarily correct! We are assuming that the minimum number of possible errors has occurred, namely, 1. It is possible that the original code word was supposed to be 10110 and was changed to 10000 when two errors occurred.

Suppose two errors really did occur. For example, assume we read the invalid code word 11000. If we calculate the distance vector of [2,3,3,2], we see there is no "closest" code word, and we are unable to make the correction. The minimum Hamming distance of 3 permits correction of one error only, and cannot ensure correction, as evidenced in this example, if more than one error occurs.

In our discussion up to this point, we have simply presented you with various codes, but have not given any specifics as to how the codes are generated. There are many methods that are used for code generation; perhaps one of the more intuitive is the Hamming algorithm for code design, which we now present. Before explaining the actual steps in the algorithm, we provide some background material.

Suppose we wish to design a code with words consisting of m data bits and r check bits, which allows for single-bit errors to be corrected. This implies that there are 2^m legal code words, each with a unique combination of check bits. Because we are focused on single-bit errors, let's examine the set of invalid code words that are a distance of 1 from all legal code words.

Each valid code word has n bits, and an error could occur in any of these n positions. Thus, each valid code word has n illegal code words at a distance of 1. Therefore, if we are concerned with each legal code word and each invalid code word consisting of one error, we have $n + 1$ bit patterns associated with each code word (1 legal word and n illegal words). Because each code word consists of n bits, where $n = m + r$, there are 2^n total bit patterns possible. This results in the following inequality:

$$(n + 1) \times 2^m \leq 2^n$$

where $n + 1$ is the number of bit patterns per code word, 2^m is the number of legal code words, and 2^n is the total number of bit patterns possible. Because $n = m + r$, we can rewrite the inequality as:

$$(m + r + 1) \times 2^m \leq 2^{m+r}$$

or

$$(m + r + 1) \circ 2^r$$

This inequality is important because it specifies the lower limit on the number of check bits required (we always use as few check bits as possible) to construct a code with m data bits and r check bits that corrects all single-bit errors.

Suppose we have data words of length $m = 4$. Then:

$$(4 + r + 1) \circ 2^r$$

which implies that r must be greater than or equal to 3. We choose $r = 3$. This means to build a code with data words of 4 bits that should correct single-bit errors, we must add 3 check bits.

The Hamming Algorithm

The Hamming algorithm provides a straightforward method for designing codes to correct single-bit errors. To construct error-correcting codes for any size memory word, we follow these steps:

1. Determine the number of check bits, r, necessary for the code and then number the n bits (where $n = m + r$), right to left, starting with 1 (not 0).
2. Each bit whose bit number is a power of 2 is a parity bit—the others are data bits.
3. Assign parity bits to check bit positions as follows: Bit b is checked by those parity bits $b_1, b_2, \ldots, b_j$ such that $b_1 + b_2 + \ldots + b_j = b$ (where "+" indicates the modulo 2 sum).

We now present an example to illustrate these steps and the actual process of error correction.

EXAMPLE 2.42 Using the Hamming code just described and even parity, encode the 8-bit ASCII character *K*. (The high-order bit will be 0.) Induce a single-bit error and then indicate how to locate the error.

We first determine the code word for *K*.

Step 1: Determine the number of necessary check bits, add these bits to the data bits, and number all n bits.

Because $m = 8$, we have: $(8 + r + 1) \circ 2^r$, which implies that r must be greater than or equal to 4. We choose $r = 4$.

Step 2: Number the n bits right to left, starting with 1, which results in:

12	11	10	9	8	7	6	5	4	3	2	1
				□				□		□	□

The parity bits are marked by boxes.

Step 3: Assign parity bits to check the various bit positions.

To perform this step, we first write all bit positions as sums of those numbers that are powers of 2:

1 = 1	5 = 1 + 4	9 = 1 + 8
2 = 2	6 = 2 + 4	10 = 2 + 8
3 = 1 + 2	7 = 1 + 2 + 4	11 = 1 + 2 + 8
4 = 4	8 = 8	12 = 4 + 8

The number 1 contributes to 1, 3, 5, 7, 9, and 11, so this parity bit will reflect the parity of the bits in these positions. Similarly, 2 contributes to 2, 3, 6, 7, 10, and 11, so the parity bit in position 2 reflects the parity of this set of bits. Bit 4 provides parity for 4, 5, 6, 7, and 12, and bit 8 provides parity for bits 8, 9, 10, 11, and 12. If we write the data bits in the nonboxed blanks, and then add the parity bits, we have the following code word as a result:

0	1	0	0	[1]	1	0	1	[0]	1	[1]	[0]
12	11	10	9	8	7	6	5	4	3	2	1

Therefore, the code word for K is 010011010110.

Let's introduce an error in bit position b_9, resulting in the code word 010111010110. If we use the parity bits to check the various sets of bits, we find the following:

Bit 1 checks 1, 3, 5, 7, 9, and 11: With even parity, this produces an error.
Bit 2 checks 2, 3, 6, 7, 10, and 11: This is okay.
Bit 4 checks 4, 5, 6, 7, and 12: This is okay.
Bit 8 checks 8, 9, 10, 11, and 12: This produces an error.

Parity bits 1 and 8 show errors. These two parity bits both check 9 and 11, so the single-bit error must be in either bit 9 or bit 11. However, because bit 2 checks bit 11 and indicates no error has occurred in the subset of bits it checks, the error must occur in bit 9. (We know this because we created the error; however, note that even if we have no clue where the error is, using this method allows us to determine the position of the error and correct it by simply flipping the bit.)

Because of the way the parity bits are positioned, an easier method to detect and correct the error bit is to add the positions of the parity bits that indicate an error. We found that parity bits 1 and 8 produced an error, and 1 + 8 = 9, which is exactly where the error occurred.

EXAMPLE 2.43 Use the Hamming algorithm to find all code words for a 3-bit memory word, assuming odd parity.

We have 8 possible words: 000, 001, 010, 011, 100, 101, 110, and 111. We first need to determine the required number of check bits. Because $m = 3$, we have $(3 + r + 1) \circ 2^r$, which implies that r must be greater than or equal to 3.

We choose $r = 3$. Therefore, each code word has 6 bits, and the check bits are in positions 1, 2, and 4, as shown here:

6	5	4	3	2	1
_	_	□	_	□	□

From our previous example, we know that:

- Bit 1 checks the parity over bits 1, 3, and 5.
- Bit 2 check the parity over bits 2, 3, and 6.
- Bit 4 checks the parity over bits 4, 5, and 6.

Therefore, we have the following code words for each memory word:

Memory Word	Code Word						
000	Bit position	6	5	4	3	2	1
		0	0	1	0	1	1
001	Bit position	6	5	4	3	2	1
		0	0	1	1	0	0
010	Bit position	6	5	4	3	2	1
		0	1	0	0	1	0
011	Bit position	6	5	4	3	2	1
		0	1	0	1	0	1
100	Bit position	6	5	4	3	2	1
		1	0	0	0	0	1
101	Bit position	6	5	4	3	2	1
		1	0	0	1	1	0
110	Bit position	6	5	4	3	2	1
		1	1	1	0	0	0
111	Bit position	6	5	4	3	2	1
		1	1	1	1	1	1

Our set of code words is 001011, 001100, 010010, 010101, 100001, 100110, 111000, and 111111. If a single bit in any of these words is flipped, we can determine exactly which one it is and correct it. For example, to send 111, we actually send the code word 111111 instead. If 110111 is received, parity bit 1 (which checks bits 1, 3, and 5) is okay, and parity bit 2 (which checks bits 2, 3, and 6) is okay, but parity bit 4 shows an error, as only bits 5 and 6 are 1s, violating odd parity. Bit 5 cannot be incorrect, because parity bit 1 checked out okay. Bit 6 cannot be wrong because parity bit 2 checked out okay. Therefore, it must be bit 4 that is wrong, so it is changed from a 0 to a 1, resulting in the correct code word 111111.

In the next chapter, you will see how easy it is to implement a Hamming code using simple binary circuits. Because of its simplicity, Hamming code protection can be added inexpensively and with minimal effect on performance.

2.7.3 Reed-Solomon

Hamming codes work well in situations where one can reasonably expect errors to be rare events. Fixed magnetic disk drives have error ratings on the order of 1 bit in 100 million. The 3-bit Hamming code that we just studied will easily correct this type of error. However, Hamming codes are useless in situations where there is a likelihood that multiple adjacent bits will be damaged. These kinds of errors are called **burst errors**. Because of their exposure to mishandling and environmental stresses, burst errors are common on removable media such as magnetic tapes and compact discs.

If we expect errors to occur in blocks, it stands to reason that we should use an error-correcting code that operates at a block level, as opposed to a Hamming code, which operates at the bit level. A **Reed-Solomon (RS)** code can be thought of as a CRC that operates over entire characters instead of only a few bits. RS codes, like CRCs, are systematic: The parity bytes are appended to a block of information bytes. RS(n, k) codes are defined using the following parameters:

- s = The number of bits in a character (or "symbol")
- k = The number of s-bit characters comprising the data block
- n = The number of bits in the code word

RS(n, k) can correct $\frac{(n-k)}{2}$ errors in the k information bytes.

The popular RS(255, 223) code, therefore, uses 223 8-bit information bytes and 32 syndrome bytes to form 255-byte code words. It will correct as many as 16 erroneous bytes in the information block.

The generator polynomial for an RS code is given by a polynomial defined over an abstract mathematical structure called a **Galois field**. (A lucid discussion of Galois mathematics is beyond the scope of this text. See the references at the end of the chapter.) The RS-generating polynomial is:

$$g(x) = \left(x - a^{i}\right)\left(x - a^{i+1}\right) \ldots \left(x - a^{i+2t}\right)$$

where $t = n - k$ and x is an entire byte (or symbol) and $g(x)$ operates over the field $GF(2^s)$. (Note: This polynomial expands over the Galois field, which is considerably different from the integer fields used in ordinary algebra.)

The n-byte RS code word is computed using the equation:

$$c(x) = g(x) \times i(x)$$

where $i(x)$ is the information block.

Despite the daunting algebra behind them, RS error-correction algorithms lend themselves well to implementation in computer hardware. They are implemented in high-performance disk drives for mainframe computers as well as compact discs used for music and data storage. These implementations will be described in Chapter 7.

CHAPTER SUMMARY

We have presented the essentials of data representation and numerical operations in digital computers. You should master the techniques described for base conversion and memorize the smaller hexadecimal and binary numbers. This knowledge will be beneficial to you as you study the remainder of this text. Your knowledge of hexadecimal coding will be useful if you are ever required to read a core (memory) dump after a system crash or if you do any serious work in the field of data communications.

You have also seen that floating-point numbers can produce significant errors when small errors are allowed to compound over iterative processes. There are various numerical techniques that can be used to control such errors. These techniques merit detailed study but are beyond the scope of this text.

You have learned that most computers use ASCII or EBCDIC to represent characters. It is generally of little value to memorize any of these codes in their entirety, but if you work with them frequently, you will find yourself learning a number of "key values" from which you can compute most of the others that you need.

Unicode is the default character set used by Java and recent versions of Windows. It is likely to replace EBCDIC and ASCII as the basic method of character representation in computer systems; however, the older codes will be with us for the foreseeable future, owing both to their economy and their pervasiveness.

Error detecting and correcting codes are used in virtually all facets of computing technology. Should the need arise, your understanding of the various error control methods will help you to make informed choices among the various options available. The method that you choose will depend on a number of factors, including computational overhead and the capacity of the storage and transmission media available to you.

FURTHER READING

A brief account of early mathematics in Western civilization can be found in Bunt et al. (1988).

Knuth (1998) presents a delightful and thorough discussion of the evolution of number systems and computer arithmetic in Volume 2 of his series on computer algorithms. (*Every* computer scientist should own a set of the Knuth books.)

A definitive account of floating-point arithmetic can be found in Goldberg (1991). Schwartz et al. (1999) describe how the IBM System/390 performs floating-point operations in both the older form and the IEEE standard. Soderquist and Leeser (1996) provide an excellent and detailed discussion of the problems surrounding floating-point division and square roots.

Detailed information about Unicode can be found at the Unicode Consortium website, www.unicode.org, as well as in the *Unicode Standard, Version 4.0* (2003).

The International Standards Organization website can be found at www.iso.ch. You will be amazed at the span of influence of this group. A similar trove of

information can be found at the American National Standards Institute website: www.ansi.org.

After you master the concepts of Boolean algebra and digital logic, you will enjoy reading Arazi's book (1988). This well-written text shows how error detection and correction are achieved using simple digital circuits. Arazi's appendix gives a remarkably lucid discussion of the Galois field arithmetic that is used in Reed-Solomon codes.

If you'd prefer a rigorous and exhaustive study of error-correction theory, Pretzel's (1992) book is an excellent place to start. The text is accessible, well-written, and thorough.

Detailed discussions of Galois fields can be found in the (inexpensive!) books by Artin (1998) and Warner (1990). Warner's much larger book is a clearly written and comprehensive introduction to the concepts of abstract algebra. A study of abstract algebra will be helpful should you delve into the study of mathematical cryptography, a fast-growing area of interest in computer science.

REFERENCES

Arazi, B. *A Commonsense Approach to the Theory of Error Correcting Codes.* Cambridge, MA: The MIT Press, 1988.

Artin, E. *Galois Theory*. New York: Dover Publications, 1998.

Bunt, L. N. H., Jones, P. S., & Bedient, J. D. *The Historical Roots of Elementary Mathematics.* New York: Dover Publications, 1988.

Goldberg, D. "What Every Computer Scientist Should Know about Floating-Point Arithmetic." *ACM Computing Surveys 23*:1, March 1991, pp. 5–47.

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

Pretzel, O. *Error-Correcting Codes and Finite Fields.* New York: Oxford University Press, 1992.

Schwartz, E. M., Smith, R. M., & Krygowski, C. A. "The S/390 G5 Floating-Point Unit Supporting Hex and Binary Architectures." *IEEE Proceedings from the 14th Symposium on Computer Arithmetic,* 1999, pp. 258–265.

Soderquist, P., & Leeser, M. "Area and Performance Tradeoffs in Floating-Point Divide and Square-Root Implementations." *ACM Computing Surveys 28*:3, September 1996, pp. 518–564.

The Unicode Consortium. *The Unicode Standard, Version 4.0.* Reading, MA: Addison-Wesley, 2003.

Warner, S. *Modern Algebra.* New York: Dover Publications, 1990.

REVIEW OF ESSENTIAL TERMS AND CONCEPTS

1. The word *bit* is a contraction for what two words?
2. Explain how the terms *bit*, *byte*, *nibble*, and *word* are related.
3. Why are binary and decimal called *positional numbering systems*?
4. Explain how base 2, base 8, and base 16 are related.
5. What is a radix?

6. How many of the "numbers to remember" (in all bases) from Table 2.1 can you remember?
7. What does *overflow* mean in the context of unsigned numbers?
8. Name the four ways in which signed integers can be represented in digital computers, and explain the differences.
9. Which one of the four representations for signed integers is used most often by digital computer systems?
10. How are complement systems similar to the odometer on a bicycle?
11. Do you think that double-dabble is an easier method than the other binary-to-decimal conversion methods explained in this chapter? Why?
12. With reference to the previous question, what are the drawbacks of the other two conversion methods?
13. What is overflow, and how can it be detected? How does overflow in unsigned numbers differ from overflow in signed numbers?
14. If a computer is capable only of manipulating and storing integers, what difficulties present themselves? How are these difficulties overcome?
15. What are the goals of Booth's algorithm?
16. How does carry differ from overflow?
17. What is arithmetic shifting?
18. What are the three component parts of a floating-point number?
19. What is a biased exponent, and what efficiencies can it provide?
20. What is normalization, and why is it necessary?
21. Why is there always some degree of error in floating-point arithmetic when it is performed by a binary digital computer?
22. How many bits long is a double-precision number under the IEEE-754 floating-point standard?
23. What is EBCDIC, and how is it related to BCD?
24. What is ASCII, and how did it originate?
25. Explain the difference between ASCII and Unicode.
26. How many bits does a Unicode character require?
27. Why was Unicode created?
28. How do cyclic redundancy checks work?
29. What is systematic error detection?
30. What is a Hamming code?
31. What is meant by *Hamming distance*, and why is it important? What is meant by minimum Hamming distance?
32. How is the number of redundant bits necessary for code related to the number of data bits?
33. What is a burst error?
34. Name an error-detection method that can compensate for burst errors.

EXERCISES

◆**1.** Perform the following base conversions using subtraction or division-remainder:

a) 458_{10} = _____ 3

b) 677_{10} = _____ 5

c) 1518_{10} = _____ 7

d) 4401_{10} = _____ 9

2. Perform the following base conversions using subtraction or division-remainder:

a) 588_{10} = _____ 3

b) 2254_{10} = _____ 5

c) 652_{10} = _____ 7

d) 3104_{10} = _____ 9

3. Perform the following base conversions using subtraction or division-remainder:

a) 137_{10} = _____ 3

b) 248_{10} = _____ 5

c) 387_{10} = _____ 7

d) 633_{10} = _____ 9

◆**4.** Perform the following base conversions:

a) 20101_3 = _____ 10

b) 2302_5 = _____ 10

c) 1605_7 = _____ 10

d) 687_9 = _____ 10

5. Perform the following base conversions:

a) 20012_3 = _____ 10

b) 4103_5 = _____ 10

c) 3236_7 = _____ 10

d) 1378_9 = _____ 10

6. Perform the following base conversions:

a) 21200_3 = _____ 10

b) 3244_5 = _____ 10

c) 3402_7 = _____ 10

d) 7657_9 = _____ 10

◆**7.** Convert the following decimal fractions to binary with a maximum of six places to the right of the binary point:

a) 26.78125

b) 194.03125

c) 298.796875

d) 16.1240234375

8. Convert the following decimal fractions to binary with a maximum of six places to the right of the binary point:

a) 25.84375

b) 57.55

c) 80.90625

d) 84.874023

◆**9.** Convert the following decimal fractions to binary with a maximum of six places to the right of the binary point:

a) 27.59375

b) 105.59375

c) 241.53125

d) 327.78125

10. Convert the following binary fractions to decimal:

a) 10111.1101

b) 100011.10011

c) 1010011.10001

d) 11000010.111

11. Convert the following binary fractions to decimal:

a) 100001.111

b) 111111.10011

c) 1001100.1011

d) 10001001.0111

12. Convert the following binary fractions to decimal:

a) 110001.10101

b) 111001.001011

c) 1001001.10101

d) 11101001.110001

13. Convert the hexadecimal number $AC12_{16}$ to binary.

14. Convert the hexadecimal number $7A01_{16}$ to binary.

15. Convert the hexadecimal number DEAD $BEEF_{16}$ to binary.

16. Represent the following decimal numbers in binary using 8-bit signed magnitude, one's complement, two's complement, and excess-127 representations.

◆**a)** 77

◆**b)** –42

c) 119

d) –107

17. Represent the following decimal numbers in binary using 8-bit signed magnitude, one's complement, two's complement, and excess-127 representations:

a) 60

b) –60

c) 20

d) –20

18. Represent the following decimal numbers in binary using 8-bit signed magnitude, one's complement, two's complement, and excess-127 representations:

a) 97

b) –97

c) 44

d) –44

19. Represent the following decimal numbers in binary using 8-bit signed magnitude, one's complement, two's complement, and excess-127 representations:

a) 89

b) –89

c) 66

d) –66

20. What decimal value does the 8-bit binary number 10011110 have if:

a) It is interpreted as an unsigned number?

b) It is on a computer using signed-magnitude representation?

c) It is on a computer using one's complement representation?

d) It is on a computer using two's complement representation?

e) It is on a computer using excess-127 representation?

◆**21.** What decimal value does the 8-bit binary number 00010001 have if:

a) It is interpreted as an unsigned number?

b) It is on a computer using signed-magnitude representation?

c) It is on a computer using one's complement representation?

d) It is on a computer using two's complement representation?

e) It is on a computer using excess-127 representation?

22. What decimal value does the 8-bit binary number 10110100 have if:

a) It is interpreted as an unsigned number?

b) It is on a computer using signed-magnitude representation?

c) It is on a computer using one's complement representation?

d) It is on a computer using two's complement representation?

e) It is on a computer using excess-127 representation?

23. Given the two binary numbers 11111100 and 01110000:

a) Which of these two numbers is the larger unsigned binary number?

b) Which of these two is the larger when it is being interpreted on a computer using signed two's complement representation?

c) Which of these two is the smaller when it is being interpreted on a computer using signed-magnitude representation?

24. Using a "word" of 3 bits, list all the possible signed binary numbers and their decimal equivalents that are representable in:

a) Signed magnitude

b) One's complement

c) Two's complement

25. Using a "word" of 4 bits, list all the possible signed binary numbers and their decimal equivalents that are representable in:

a) Signed magnitude

b) One's complement

c) Two's complement

26. From the results of the previous two questions, generalize the range of values (in decimal) that can be represented in any given x number of bits using:

a) Signed magnitude

b) One's complement

c) Two's complement

27. Fill in the following table to indicate what each binary pattern represents using the various formats.

Unsigned Integer	4-Bit Binary Value	Signed Magnitude	One's Complement	Two's Complement	Excess-7
0	0000				
1	0001				
2	0010				
3	0011				
4	0100				
5	0101				
6	0110				
7	0111				
8	1000				
9	1001				
10	1010				
11	1011				
12	1100				
13	1101				
14	1110				
15	1111				

28. Given a (very) tiny computer that has a word size of 6 bits, what are the smallest negative numbers and the largest positive numbers that this computer can represent in each of the following representations?

◆**a)** One's complement

b) Two's complement

29. To add two two's complement numbers together, what must be true?

30. What is the most common representation used in most computers to store signed integer values and why?

31. You have stumbled on an unknown civilization while sailing around the world. The people, who call themselves Zebronians, do math using 40 separate characters (probably because there are 40 stripes on a zebra). They would very much like to use computers, but would need a computer to do Zebronian math, which would mean a computer that could represent all 40 characters. You are a computer designer and decide to help them. You decide the best thing is to use BCZ, Binary-Coded Zebronian (which is like BCD except it codes Zebronian, not Decimal). How many bits will you need to represent each character if you want to use the minimum number of bits?

◆**32.** Add the following unsigned binary numbers as shown.

a)
$$\begin{array}{r} 01110101 \\ +\ 00111011 \\ \hline \end{array}$$

b)
$$\begin{array}{r} 00010101 \\ +\ 00011011 \\ \hline \end{array}$$

c)
$$\begin{array}{r} 01101111 \\ +\ 00010001 \\ \hline \end{array}$$

33. Add the following unsigned binary numbers as shown.

a)
$$\begin{array}{r} 01000100 \\ +\ 10111011 \\ \hline \end{array}$$

b)
$$\begin{array}{r} 01011011 \\ +\ 00011111 \\ \hline \end{array}$$

c)
$$\begin{array}{r} 10101100 \\ +\ 00100100 \\ \hline \end{array}$$

◆**34.** Subtract the following signed binary numbers as shown using two's complement arithmetic.

a)
$$\begin{array}{r} 01110101 \\ -\ 00111011 \\ \hline \end{array}$$

b)
$$\begin{array}{r} 00110101 \\ -\ 00001011 \\ \hline \end{array}$$

c)
$$\begin{array}{r} 01101111 \\ -\ 00010001 \\ \hline \end{array}$$

35. Subtract the following signed binary numbers as shown using two's complement arithmetic.

a)
$$\begin{array}{r} 11000100 \\ -\ 00111011 \\ \hline \end{array}$$

b)
$$\begin{array}{r} 01011011 \\ -\ 00011111 \\ \hline \end{array}$$

c)
$$\begin{array}{r} 10101100 \\ -\ 00100100 \\ \hline \end{array}$$

36. Perform the following binary multiplications, assuming unsigned integers:

◆**a)**
$$\begin{array}{r} 1100 \\ \times\ 101 \\ \hline \end{array}$$

b)
$$\begin{array}{r} 10101 \\ \times\ 111 \\ \hline \end{array}$$

c)
$$\begin{array}{r} 11010 \\ \times\ 1100 \\ \hline \end{array}$$

37. Perform the following binary multiplications, assuming unsigned integers:

a)
$$\begin{array}{r} 1011 \\ \times\ 101 \\ \hline \end{array}$$

b)
$$\begin{array}{r} 10011 \\ \times\ 1011 \\ \hline \end{array}$$

c)
$$\begin{array}{r} 11010 \\ \times\ 1011 \\ \hline \end{array}$$

38. Perform the following binary divisions, assuming unsigned integers:

◆**a)** 101101 ÷ 101

b) 10000001 ÷ 101

c) 1001010010 ÷ 1011

39. Perform the following binary divisions, assuming unsigned integers:

a) 11111101 ÷ 1011

b) 110010101 ÷ 1001

c) 1001111100 ÷ 1100

◆**40.** Use the double-dabble method to convert 10212_3 directly to decimal. (Hint: You have to change the multiplier.)

41. Using signed-magnitude representation, complete the following operations:

$+0 + (-0) =$

$(-0) + 0 =$

$0 + 0 =$

$(-0) + (-0) =$

◆**42.** Suppose a computer uses 4-bit one's complement representation. Ignoring overflows, what value will be stored in the variable *j* after the following pseudocode routine terminates?

```
0 → j  // Store 0 in j.
-3 → k // Store -3 in k.
while k ≠ 0
  j = j + 1
  k = k - 1
end while
```

43. Perform the following binary multiplications using Booth's algorithm, assuming signed two's complement integers:

a)
$$\begin{array}{r} 1011 \\ \times\ 0101 \\ \hline \end{array}$$

b)
$$\begin{array}{r} 0011 \\ \times\ 1011 \\ \hline \end{array}$$

c)
$$\begin{array}{r} 1011 \\ \times\ 1100 \\ \hline \end{array}$$

44. Using arithmetic shifting, perform the following:

a) Double the value 00010101_2.

b) Quadruple the value 01110111_2.

c) Divide the value 11001010_2 in half.

45. If the floating-point number representation on a certain system has a sign bit, a 3-bit exponent, and a 4-bit significand:

a) What is the largest positive and the smallest positive number that can be stored on this system if the storage is normalized? (Assume that no bits are implied, there is no biasing, exponents use two's complement notation, and exponents of all 0s and all 1s are allowed.)

b) What bias should be used in the exponent if we prefer all exponents to be non-negative? Why would you choose this bias?

◆**46.** Using the model in the previous question, including your chosen bias, add the following floating-point numbers and express your answer using the same notation as the addend and augend:

0	1	1	1	1	0	0	0
0	1	0	1	1	0	0	1

Calculate the relative error, if any, in your answer to the previous question.

47. Assume we are using the simple model for floating-point representation as given in the text (the representation uses a 14-bit format, 5 bits for the exponent with a bias of 15, a normalized mantissa of 8 bits, and a single sign bit for the number):

a) Show how the computer would represent the numbers 100.0 and 0.25 using this floating-point format.

b) Show how the computer would add the two floating-point numbers in Part a by changing one of the numbers so they are both expressed using the same power of 2.

c) Show how the computer would represent the sum in Part b using the given floating-point representation. What decimal value for the sum is the computer actually storing? Explain.

48. What causes divide underflow, and what can be done about it?

49. Why do we usually store floating-point numbers in normalized form? What is the advantage of using a bias as opposed to adding a sign bit to the exponent?

50. Let $a = 1.0 \times 2^9$, $b = -1.0 \times 2^9$ and $c = 1.0 \times 2^1$. Using the simple floating-point model described in the text (the representation uses a 14-bit format, 5 bits for the exponent with a bias of 15, a normalized mantissa of 8 bits, and a single sign bit for the number), perform the following calculations, paying close attention to the order of operations. What can you say about the algebraic properties of floating-point arithmetic in our finite model? Do you think this algebraic anomaly holds under multiplication as well as addition?

$$b + (a + c) =$$
$$(b + a) + c =$$

51. Show how each of the following floating-point values would be stored using IEEE-754 single precision (be sure to indicate the sign bit, the exponent, and the significand fields):

a) 12.5 **b)** −1.5 **c)** 0.75 **d)** 26.625

52. Show how each of the following floating-point values would be stored using IEEE-754 double precision (be sure to indicate the sign bit, the exponent, and the significand fields):

a) 12.5 **b)** −1.5 **c)** 0.75 **d)** 26.625

53. Suppose we have just found yet another representation for floating-point numbers. Using this representation, a 12-bit floating-point number has 1 bit for the sign of the number, 4 bits for the exponent, and 7 bits for the mantissa, which is normalized as in the simple model so that the first digit to the right of the radix points must be a 1. Numbers in the exponent are in signed two's complement representation. No bias is used, and there are no implied bits. Show the representation for the smallest positive number this machine can represent using the following format (simply fill in the squares provided). What decimal number does this equate to?

54. Find three floating-point values to illustrate that floating-point addition is not associative. (You will need to run a program on specific hardware with a specific compiler.)

55. a) Given that the ASCII code for *A* is 1000001, what is the ASCII code for *J*?

b) Given that the EBCDIC code for *A* is 1100 0001, what is the EBCDIC code for *J*?

56. a) The ASCII code for the letter *A* is 1000001, and the ASCII code for the letter *a* is 1100001. Given that the ASCII code for the letter *G* is 1000111, without looking at Table 2.7, what is the ASCII code for the letter *g*?

b) The EBCDIC code for the letter *A* is 1100 0001, and the EBCDIC code for the letter *a* is 1000 0001. Given that the EBCDIC code for the letter *G* is 1100 0111, without looking at Table 2.6, what is the EBCDIC code for the letter *g*?

c) The ASCII code for the letter *A* is 1000001, and the ASCII code for the letter *a* is 1100001. Given that the ASCII code for the letter *Q* is 1010001, without looking at Table 2.7, what is the ASCII code for the letter *q*?

d) The EBCDIC code for the letter *J* is 1101 0001, and the EBCDIC code for the letter *j* is 1001 0001. Given that the EBCDIC code for the letter *Q* is 1101 1000, without looking at Table 2.6, what is the EBCDIC code for the letter *q*?

e) In general, if you were going to write a program to convert uppercase ASCII characters to lowercase, how would you do it? Looking at Table 2.6, could you use the same algorithm to convert uppercase EBCDIC letters to lowercase?

f) If you were tasked with interfacing an EBCDIC-based computer with an ASCII or Unicode computer, what would be the best way to convert the EBCDIC characters to ASCII characters?

◆**57.** Assume a 24-bit word on a computer. In these 24 bits, we wish to represent the value 295.

a) How would the computer represent the decimal value 295?

b) If our computer uses 8-bit ASCII and even parity, how would the computer represent the string 295?

c) If our computer uses packed BCD with zero padding, how would the computer represent the number +295?

58. Decode the following ASCII message, assuming 7-bit ASCII characters and no parity:
1001010 1001111 1001000 1001110 0100000 1000100 1001111 1000101

59. Why would a system designer wish to make Unicode the default character set for their new system? What reason(s) could you give for not using Unicode as a default? (Hint: Think about language compatibility versus storage space.)

60. Assume we wish to create a code using 3 information bits, 1 parity bit (appended to the end of the information), and odd parity. List all legal code words in this code. What is the Hamming distance of your code?

61. Suppose we are given the following subset of code words, created for a 7-bit memory word with one parity bit: 11100110, 00001000, 10101011, and 11111110. Does this code use even or odd parity? Explain.

62. Are the error-correcting Hamming codes systematic? Explain.

63. Compute the Hamming distance of the following code:
0011010010111100
0000011110001111
0010010110101101
0001011010011110

64. Compute the Hamming distance of the following code:
0000000101111111
0000001010111111
0000010011011111
0000100011101111
0001000011110111
0010000011111011
0100000011111101
1000000011111110

65. In defining the Hamming distance for a code, we choose to use the minimum (Hamming) distance between any two encodings. Explain why it would not be better to use the maximum or average distance.

66. Suppose we want an error-correcting code that will allow all single-bit errors to be corrected for memory words of length 10.

a) How many parity bits are necessary?

b) Assuming we are using the Hamming algorithm presented in this chapter to design our error-correcting code, find the code word to represent the 10-bit information word:

1 0 0 1 1 0 0 1 1 0.

67. Suppose we want an error-correcting code that will allow all single-bit errors to be corrected for memory words of length 12.

a) How many parity bits are necessary?

b) Assuming we are using the Hamming algorithm presented in this chapter to design our error-correcting code, find the code word to represent the 12-bit information word:

1 0 0 1 0 0 0 1 1 0 1 0.

◆**68.** Suppose we are working with an error-correcting code that will allow all single-bit errors to be corrected for memory words of length 7. We have already calculated that we need 4 check bits, and the length of all code words will be 11. Code words are created according to the Hamming algorithm presented in the text. We now receive the following code word:

1 0 1 0 1 0 1 1 1 1 0

Assuming even parity, is this a legal code word? If not, according to our error-correcting code, where is the error?

69. Repeat exercise 68 using the following code word:

0 1 1 1 1 0 1 0 1 0 1

70. Suppose we are working with an error-correcting code that will allow all single-bit errors to be corrected for memory words of length 12. We have already calculated that we need 5 check bits, and the length of all code words will be 17. Code words are created according to the Hamming algorithm presented in the text. We now receive the following code word:

0 1 1 0 0 1 0 1 0 0 1 0 0 1 0 0 1

Assuming even parity, is this a legal code word? If not, according to our error-correcting code, where is the error?

71. Name two ways in which Reed-Solomon coding differs from Hamming coding.

72. When would you choose a CRC code over a Hamming code? A Hamming code over a CRC?

◆**73.** Find the quotients and remainders for the following division problems modulo 2.

a) $1010111_2 \div 1101_2$

b) $1011111_2 \div 11101_2$

c) $1011001101_2 \div 10101_2$

d) $111010111_2 \div 10111_2$

74. Find the quotients and remainders for the following division problems modulo 2.

a) $1111010_2 \div 1011_2$

b) $1010101_2 \div 1100_2$

c) $1101101011_2 \div 10101_2$

d) $1111101011_2 \div 101101_2$

75. Find the quotients and remainders for the following division problems modulo 2.

a) $11001001_2 \div 1101_2$

b) $1011000_2 \div 10011_2$

c) $11101011_2 \div 10111_2$

d) $111110001_2 \div 1001_2$

76. Find the quotients and remainders for the following division problems modulo 2.

a) $1001111_2 \div 1101_2$

b) $1011110_2 \div 1100_2$

c) $1001101110_2 \div 11001_2$

d) $111101010_2 \div 10011_2$

◆**77.** Using the CRC polynomial 1011, compute the CRC code word for the information word 1011001. Check the division performed at the receiver.

78. Using the CRC polynomial 1101, compute the CRC code word for the information word 01001101. Check the division performed at the receiver.

79. Using the CRC polynomial 1101, compute the CRC code word for the information word 1100011. Check the division performed at the receiver.

80. Using the CRC polynomial 1101, compute the CRC code word for the information word 01011101. Check the division performed at the receiver.

81. Pick an architecture (such as 80486, Pentium, Pentium IV, SPARC, Alpha, or MIPS). Do research to find out how your architecture approaches the concepts introduced in this chapter. For example, what representation does it use for negative values? What character codes does it support?

82. We have seen that floating-point arithmetic is neither associative nor distributive. Why do you think this is the case?