1.5 The Binary System

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

Binary Notation

Recall that in the base ten system, each position in a representation is associated with a quantity. In the representation 375, the 5 is in the position associated with the quantity one, the 7 is in the position associated with ten, and the 3 is in the position associated with the quantity one hundred (Figure 1.15a). Each quantity is ten times that of the quantity to its right. The value represented by the entire expression is obtained by multiplying the value of each digit by the quantity associated with that digit's position and then adding those products. To illustrate, the pattern 375 represents (3 × hundred) + (7 × ten) + (5 × one), which, in more technical notation, is $(3 \times 10^2) + (7 \times 10^1) + (5 \times 10^0)$.

The position of each digit in binary notation is also associated with a quantity, except that the quantity associated with each position is twice the quantity associated with the position to its right. More precisely, the rightmost digit in a binary representation is associated with the quantity one (2^0), the next position to the left is associated with two (2^1), the next is associated with four (2^2), the next with eight (2^3), and so on. For example, in the binary representation 1011, the rightmost 1 is in the position associated with the quantity one, the 1 next to it is in the position associated with two, the 0 is in the position associated with four, and the leftmost 1 is in the position associated with eight (Figure 1.15b).

To extract the value represented by a binary representation, we follow the same procedure as in base ten—we multiply the value of each digit by the quantity associated with its position and add the results. For example, the value represented by 100101 is 37, as shown in Figure 1.16. Note that since binary notation uses only the digits 0 and 1, this multiply-and-add process reduces merely to adding the quantities associated with the positions occupied by 1s. Thus the binary pattern 1011 represents the value eleven, because the 1s are found in the positions associated with the quantities one, two, and eight.

In Section 1.4 we learned how to count in binary notation, which allowed us to encode small integers. For finding binary representations of large values, you may prefer the approach described by the algorithm in Figure 1.17. Let us apply this algorithm to the value thirteen (Figure 1.18). We first divide thirteen by two,

Figure 1.15 The base ten and binary systems

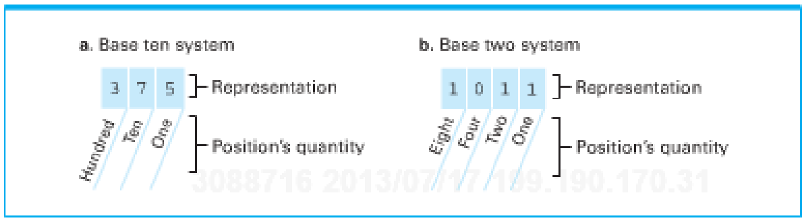


Figure 1.16 Decoding the binary representation 100101

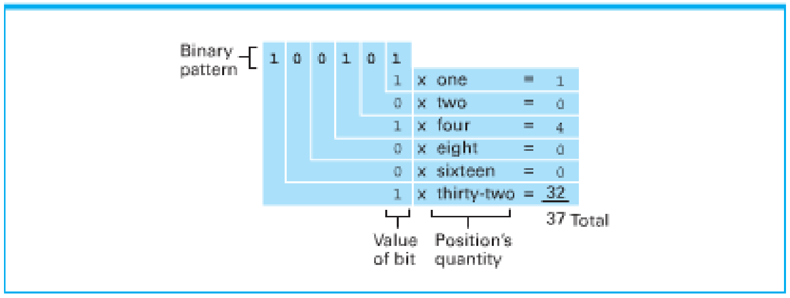


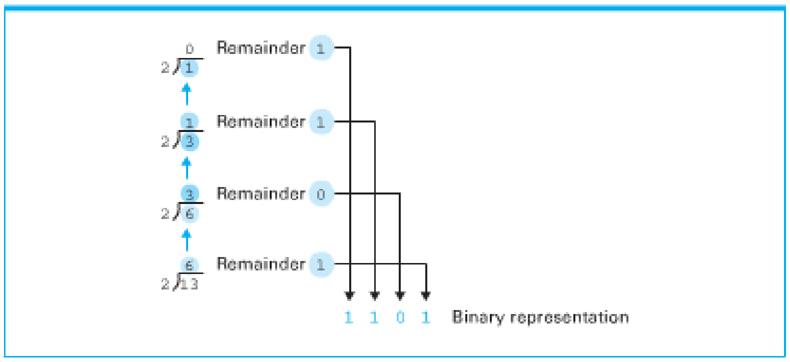
Figure 1.17 An algorithm for finding the binary representation of a positive integer

Step 1. Divide the value by two and record the remainder.

Step 2. As long as the quotient obtained is not zero, continue to divide the newest quotient by two and record the remainder.

Step 3. Now that a quotient of zero has been obtained, the binary representation of the original value consists of the remainders listed from right to left in the order they were recorded.

Figure 1.18 Applying the algorithm in Figure 1.17 to obtain the binary representation of thirteen



obtaining a quotient of six and a remainder of one. Since the quotient was not zero, Step 2 tells us to divide the quotient (six) by two, obtaining a new quotient of three and a remainder of zero. The newest quotient is still not zero, so we divide it by two, obtaining a quotient of one and a remainder of one. Once again, we divide the newest quotient (one) by two, this time obtaining a quotient of zero and a remainder of one. Since we have now acquired a quotient of zero, we move on to Step 3, where we learn that the binary representation of the original value (thirteen) is 1101, obtained from the list of remainders.

Binary Addition

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

```
   58
 + 27
```

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

```
    1
   58
 + 27
    5
```

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

```
   58
 + 27
   85
```

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

To add two integers represented in binary notation, we follow the same procedure except that all sums are computed using the addition facts shown in Figure 1.19 rather than the traditional base ten facts that you learned in elementary school. For example, to solve the problem

```
   111010
 +  11011
```

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

```
      1
   111010
 +  11011
       01
```

Figure 1.19 The binary addition facts

```
  0     1     0     1
 +0    +0    +1    +1
  0     1     1    10
```

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

```
    1
  111010
+  11011
    0101
```

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

```
 1
  111010
+  11011
  010101
```

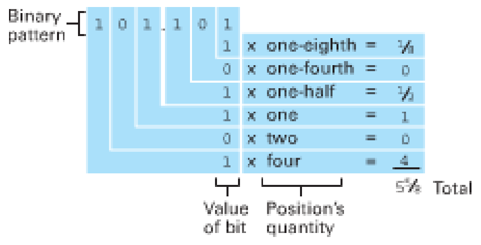
The only entry in the next column is the 1 that we carried from the previous column so we record it in the answer. Our final solution is this:

```
  111010
+  11011
 1010101
```

Fractions in Binary

To extend binary notation to accommodate fractional values, we use a **radix point** in the same role as the decimal point in decimal notation. That is, the digits to the left of the point represent the integer part (whole part) of the value and are interpreted as in the binary system discussed previously. The digits to its right represent the fractional part of the value and are interpreted in a manner similar to the other bits, except their positions are assigned fractional quantities. That is, the first position to the right of the radix is assigned the quantity $\frac{1}{2}$ (which is 2^{-1}), the next position the quantity $\frac{1}{4}$ (which is 2^{-2}), the next $\frac{1}{8}$ (which is 2^{-3}), and so on. Note that this is merely a continuation of the rule stated previously: Each position is assigned a quantity twice the size of the one to its right. With these quantities assigned to the bit positions, decoding a binary representation containing a radix point requires the same procedure as used without a radix point. More precisely, we multiply each bit value by the quantity assigned to that bit's position in the representation. To illustrate, the binary representation 101.101 decodes to $5\frac{5}{8}$, as shown in Figure 1.20.

Figure 1.20 Decoding the binary representation 101.101



Analog Versus Digital

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

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

For addition, the techniques applied in the base ten system are also applicable in binary. That is, to add two binary representations having radix points, we merely align the radix points and apply the same addition process as before. For example, 10.011 added to 100.11 produces 111.001, as shown here:

```
   10.011
+ 100.110
  111.001
```

Questions & Exercises

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

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

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

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

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

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

4. Express the following values in binary notation:

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

Add Note

5. Perform the following additions in binary notation:

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

1.6 Storing Integers

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

Two's Complement Notation

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

Figure 1.21 shows two complete two's complement systems—one based on bit patterns of length three, the other based on bit patterns of length four. Such a

Figure 1.21 Two's complement notation systems

a. Using patterns of length three

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

b. Using patterns of length four

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

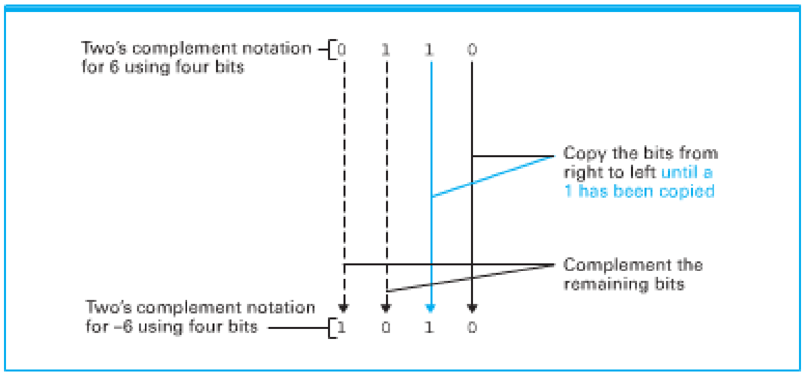
system is constructed by starting with a string of 0s of the appropriate length and then counting in binary until the pattern consisting of a single 0 followed by 1s is reached. These patterns represent the values 0, 1, 2, 3, ... The patterns representing negative values are obtained by starting with a string of 1s of the appropriate length and then counting backward in binary until the pattern consisting of a single 1 followed by 0s is reached. These patterns represent the values −1, −2, −3, ... (If counting backward in binary is difficult for you, merely start at the very bottom of the table with the pattern consisting of a single 1 followed by 0s, and count up to the pattern consisting of all 1s.)

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

In a two's complement system, there is a convenient relationship between the patterns representing positive and negative values of the same magnitude. They are identical when read from right to left, up to and including the first 1. From there on, the patterns are complements of one another. (The **complement** of a pattern is the pattern obtained by changing all the 0s to 1s and all the 1s to 0s; 0110 and 1001 are complements.) For example, in the 4-bit system in Figure 1.21 the patterns representing 2 and −2 both end with 10, but the pattern representing 2 begins with 00, whereas the pattern representing −2 begins with 11. This observation leads to an algorithm for converting back and forth between bit patterns representing positive and negative values of the same magnitude. We merely copy the original pattern from right to left until a 1 has been copied, then we complement the remaining bits as they are transferred to the final bit pattern (Figure 1.22).

Understanding these basic properties of two's complement systems also leads to an algorithm for decoding two's complement representations. If the pattern to be decoded has a sign bit of 0, we need merely read the value as

Figure 1.22 Encoding the value −6 in two's complement notation using 4 bits


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

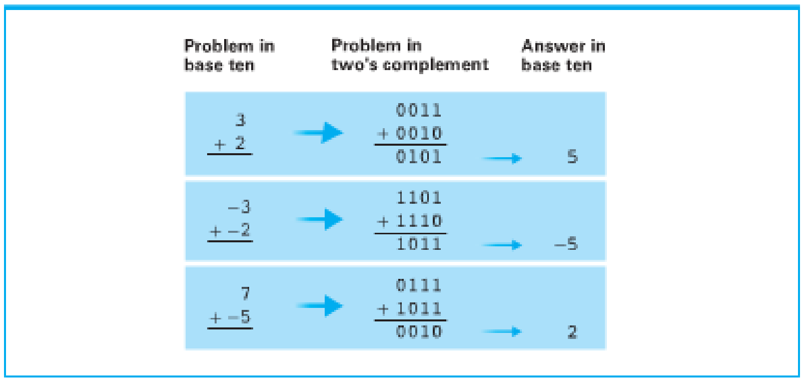
though the pattern were a binary representation. For example, 0110 represents the value 6, because 110 is binary for 6. If the pattern to be decoded has a sign bit of 1, we know the value represented is negative, and all that remains is to find the magnitude of the value. We do this by applying the "copy and complement" procedure in Figure 1.22 and then decoding the pattern obtained as though it were a straightforward binary representation. For example, to decode the pattern 1010, we first recognize that since the sign bit is 1, the value represented is negative. Hence, we apply the "copy and complement" procedure to obtain the pattern 0110, recognize that this is the binary representation for 6, and conclude that the original pattern represents −6.

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

With this understanding, consider the three addition problems in Figure 1.23. In each case, we have translated the problem into two's complement notation (using bit patterns of length four), performed the addition process previously described, and decoded the result back into our usual base ten notation.

Observe that the third problem in Figure 1.23 involves the addition of a positive number to a negative number, which demonstrates a major benefit of two's complement notation: Addition of any combination of signed numbers can be accomplished using the same algorithm and thus the same circuitry. This is in stark contrast to how humans traditionally perform arithmetic computations. Whereas elementary school children are first taught to add and later taught to subtract, a machine using two's complement notation needs to know only how to add.

Figure 1.23 Addition problems converted to two's complement notation



Problem in base ten	Problem in two's complement	Answer in base ten
3 + 2	0011 + 0010 = 0101	5
−3 + −2	1101 + 1110 = 1011	−5
7 + −5	0111 + 1011 = 0010	2

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

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

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

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

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

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

Add Note

Excess Notation

Another method of representing integer values is **excess notation.** As is the case with two's complement notation, each of the values in an excess notation system is represented by a bit pattern of the same length. To establish an excess system, we first select the pattern length to be used, then write down all the different bit patterns of that length in the order they would appear if we were counting in binary. Next, we observe that the first pattern with a 1 as its most significant bit appears approximately halfway through the list. We pick this pattern to represent zero; the patterns following this are used to represent 1, 2, 3, . . .; and the patterns preceding it are used for −1, −2, −3, ... The resulting code, when using patterns of length four, is shown in Figure 1.24. There we see that the value 5 is represented by the pattern 1101 and −5 is represented by 0011. (Note that the difference between an excess system and a two's complement system is that the sign bits are reversed.)

The system represented in Figure 1.24 is known as excess eight notation. To understand why, first interpret each of the patterns in the code using the traditional binary system and then compare these results to the values represented in the excess notation. In each case, you will find that the binary interpretation exceeds the excess notation interpretation by the value 8. For example, the pattern 1100 in binary notation represents the value 12, but in our excess system it represents 4; 0000 in binary notation represents 0, but in the excess system it represents negative 8. In a similar manner, an excess system based on patterns of length five would be called excess 16 notation,

Figure 1.24 An excess eight conversion table

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

Add Note

Figure 1.25 An excess notation system using bit patterns of length three

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

because the pattern 10000, for instance, would be used to represent zero rather than representing its usual value of 16. Likewise, you may want to confirm that the three-bit excess system would be known as excess four notation (Figure 1.25).

Questions & Exercises

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

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

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

a. 6	b. −6	c. −17
d. 13	e. −1	f. 0

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

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

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

a. four	b. six	c. eight

5. In the following problems, each bit pattern represents a value stored in two's complement notation. Find the answer to each problem in two's complement notation by performing the addition process described in

the text. Then check your work by translating the problem and your answer into base ten notation.

a. $\begin{array}{r} 0101 \\ +\ 0010 \\ \hline \end{array}$ b. $\begin{array}{r} 0011 \\ +\ 0001 \\ \hline \end{array}$ c. $\begin{array}{r} 0101 \\ +\ 1010 \\ \hline \end{array}$ d. $\begin{array}{r} 1110 \\ +\ 0011 \\ \hline \end{array}$ e. $\begin{array}{r} 1010 \\ +\ 1110 \\ \hline \end{array}$

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

a. $\begin{array}{r} 0100 \\ +\ 0011 \\ \hline \end{array}$ b. $\begin{array}{r} 0101 \\ +\ 0110 \\ \hline \end{array}$ c. $\begin{array}{r} 1010 \\ +\ 1010 \\ \hline \end{array}$ d. $\begin{array}{r} 1010 \\ +\ 0111 \\ \hline \end{array}$ e. $\begin{array}{r} 0111 \\ +\ 0001 \\ \hline \end{array}$

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

a. $\begin{array}{r} 6 \\ -(-1) \\ \hline \end{array}$ b. $\begin{array}{r} 3 \\ -2 \\ \hline \end{array}$ c. $\begin{array}{r} 4 \\ -6 \\ \hline \end{array}$ d. $\begin{array}{r} 2 \\ -(-4) \\ \hline \end{array}$ e. $\begin{array}{r} 1 \\ -5 \\ \hline \end{array}$

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

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

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

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

a. 5 b. −5 c. 3
d. 0 e. 7 f. −8

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

1.7 Storing Fractions

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

Floating-Point Notation

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

We first designate the high-order bit of the byte as the sign bit. Once again, a 0 in the sign bit will mean that the value stored is nonnegative, and a 1 will mean that the value is negative. Next, we divide the remaining 7 bits of the byte into two groups, or fields: the **exponent field** and the **mantissa field.** Let us designate the 3 bits following the sign bit as the exponent field and the remaining 4 bits as the mantissa field. Figure 1.26 illustrates how the byte is divided.

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

```
.1011
```

Next, we extract the contents of the exponent field (110) and interpret it as an integer stored using the 3-bit excess method (see again Figure 1.25). Thus the pattern in the exponent field in our example represents a positive 2. This tells us to move the radix in our solution to the right by 2 bits. (A negative exponent would mean to move the radix to the left.) Consequently, we obtain

```
10.11
```

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

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

```
.1100
```

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

```
.01100
```

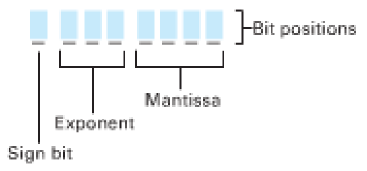
which represents $\frac{3}{8}$. Since the sign bit in the original pattern is 0, the value stored is nonnegative. We conclude that the pattern 00111100 represents $\frac{3}{8}$.

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

```
_ _ _ _ 1 0 0 1
```

We must now fill in the exponent field. To this end, we imagine the contents of the mantissa field with a radix point at its left and determine the number of bits and the direction the radix must be moved to obtain the original binary number.

Figure 1.26 Floating-point notation components



In our example, we see that the radix in .1001 must be moved 1 bit to the right to obtain 1.001. The exponent should therefore be a positive one, so we place 101 (which is positive one in excess four notation as shown in Figure 1.25) in the exponent field. Finally, we fill the sign bit with 0 because the value being stored is nonnegative. The finished byte looks like this:

0 1 0 1 1 0 0 1

There is a subtle point you may have missed when filling in the mantissa field. The rule is to copy the bit pattern appearing in the binary representation from left to right, starting with the leftmost 1. To clarify, consider the process of storing the value ⅜, which is .011 in binary notation. In this case the mantissa will be

_ _ _ _ 1 1 0 0

It will not be

_ _ _ _ 0 1 1 0

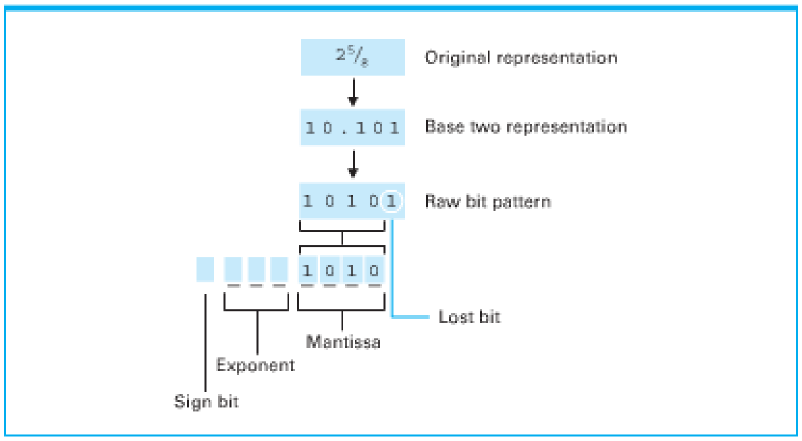
This is because we fill in the mantissa field *starting with the leftmost 1* that appears in the binary representation. Representations that conform to this rule are said to be in **normalized form.**

Using normalized form eliminates the possibility of multiple representations for the same value. For example, both 00111100 and 01000110 would decode to the value ⅜, but only the first pattern is in normalized form. Complying with normalized form also means that the representation for all nonzero values will have a mantissa that starts with 1. The value zero, however, is a special case; its floating-point representation is a bit pattern of all 0s.

Truncation Errors

Let us consider the annoying problem that occurs if we try to store the value 2⅝ with our one-byte floating-point system. We first write 2⅝ in binary, which gives us 10.101. But when we copy this into the mantissa field, we run out of room, and the rightmost 1 (which represents the last ⅛) is lost (Figure 1.27). If we ignore

Figure 1.27 Encoding the value 2⅝



this problem for now and continue by filling in the exponent field and the sign bit, we end up with the bit pattern 01101010, which represents $2\frac{1}{2}$ instead of $2\frac{5}{8}$. What has occurred is called a **truncation error,** or **round-off error**—meaning that part of the value being stored is lost because the mantissa field is not large enough.

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

Another source of truncation errors is a phenomenon that you are already accustomed to in base ten notation: the problem of nonterminating expansions, such as those found when trying to express $\frac{1}{3}$ in decimal form. Some values cannot be accurately expressed regardless of how many digits we use. The difference between our traditional base ten notation and binary notation is that more values have nonterminating representations in binary than in decimal notation. For example, the value one-tenth is nonterminating when expressed in binary. Imagine the problems this might cause the unwary person using floating-point notation to store and manipulate dollars and cents. In particular, if the dollar is used as the unit of measure, the value of a dime could not be stored accurately. A solution in this case is to manipulate the data in units of pennies so that all values are integers that can be accurately stored using a method such as two's complement.

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

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

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

Single Precision Floating Point

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

Many of today's computers support a 32 bit form of this notation called **Single Precision Floating Point.** This format uses 1 bit for the sign, 8 bits for the exponent (in an excess notation), and 23 bits for the mantissa. Thus, single precision floating point is capable of expressing very large numbers (order of 10^{38}) down to very small numbers (order of 10^{-37}) with the precision of 7 decimal digits. That is to say, the first 7 digits of a given decimal number can be stored with very good accuracy (a small amount of error may still be present). Any digits passed the first 7 will certainly be lost by truncation error (although the magnitude of the number is retained). Another form, called **Double Precision Floating Point**, uses 64 bits and provides a precision of 15 decimal digits.

If we add the values in the order listed, we first add 2½ to ⅛ and obtain 2⅝, which in binary is 10.101. Unfortunately, because this value cannot be stored accurately (as seen previously), the result of our first step ends up being stored as 2½ (which is the same as one of the values we were adding). The next step is to add this result to the last ⅛. Here again a truncation error occurs, and our final result turns out to be the incorrect answer 2½.

Now let us add the values in the opposite order. We first add ⅛ to ⅛ to obtain ¼. In binary this is .01; so the result of our first step is stored in a byte as 00111000, which is accurate. We now add this ¼ to the next value in the list, 2½, and obtain 2¾, which we can accurately store in a byte as 01101011. The result this time is the correct answer.

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

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

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

you might get the answer

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

rather than

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

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

Questions & Exercises

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

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

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

   a. 2¾ b. 5¼ c. ¾ d. −3½ e. −4⅜

3. In terms of the floating-point format discussed in the text, which of the patterns 01001001 and 00111101 represents the larger value? Describe a