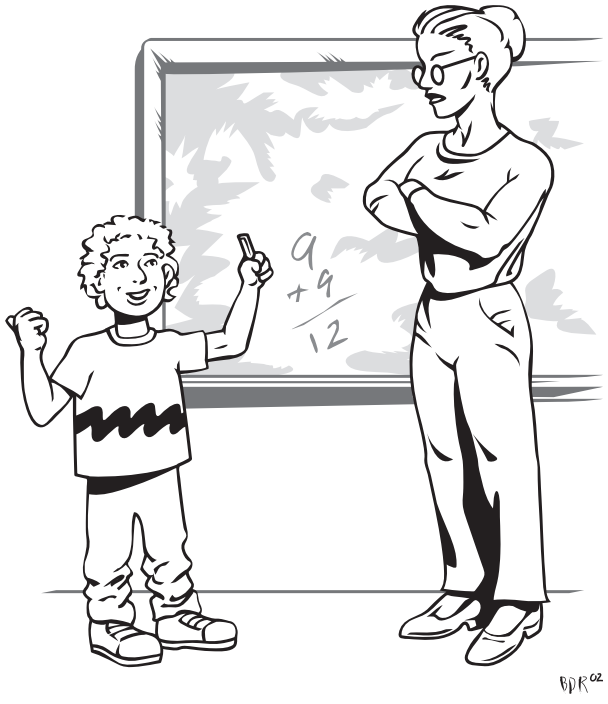
CHAPTER 5

REPRESENTING NUMERICAL DATA



"It's OK, Mrs. Grumpworthy,
my brother's teaching me arithmetic
on our computer at home."

Thomas Sperling, adapted by Benjamin Reece

5.0 INTRODUCTION

As we have noted previously, the computer stores all data and program instructions in binary form, using only groups of zeros and ones. No special provision is made for the storage of the algebraic sign or decimal point that might be associated with a number; all responsibility for interpretation of those zeros and ones is left to the programmers whose programs store and manipulate those binary numbers. Thus the binary numbers in a computer might represent characters, numbers, graphics images, video, audio, program instructions, or something else.

The ability of a computer to manipulate numbers of various kinds is, of course, of particular importance to users. In Chapter 4, we observed that nearly every high-level computing language provides a method for storage, manipulation, and calculation of signed integer and real numbers. This chapter discusses methods of representing and manipulating these numbers within the zeros-and-ones constraint of the computer.

We saw in Chapter 3 that unsigned **integer numbers** can be represented directly in binary form, and this provides a clue as to how we might represent the integer data type in the computer. There is a significant limitation, however: we have yet to show you a sign-free way of handling negative numbers that is compatible with the capabilities of the computer. In this chapter, we explore several different methods of storing and manipulating integers that may encompass both positive and negative numbers.

Also, as you know, it is not always possible to express numbers in integer form. **Real**, or floating point, **numbers** are used in the computer when the number to be expressed is outside of the integer range of the computer (too large or too small) or when the number contains a decimal fraction.

Floating point numbers allow the computer to maintain a limited, fixed number of digits of precision together with a power that shifts the point left or right within the number to make the number larger or smaller, as necessary. The range of numbers that the computer can handle in this way is huge. In a personal computer, for example, the range of numbers that may be expressed this way may be $\pm[10^{-38} < \text{number} < 10^{+38}]$ or more.

Performing calculations with floating point numbers provide an additional challenge. There are trade-offs made for the convenience of using floating point numbers: a potential loss of precision, as measured in terms of significant digits, larger storage requirements, and slower calculations. In this chapter we will also explore the properties of floating point numbers, show how they are represented in the computer, consider how calculations are performed, and learn how to convert between integer and floating point representations. We also investigate the importance of the trade-offs required for the use of floating point numbers and attempt to come up with some reasonable ground rules for deciding what number format to specify in various programming situations.

We remind you that numbers are usually input as characters and must be converted to a numerical format before they may be used in calculations. Numbers that will not be used in calculations, such as zip codes or credit card numbers, are simply manipulated as characters.

5.1 UNSIGNED BINARY AND BINARY-CODED DECIMAL REPRESENTATIONS

In conventional notation, numbers can be represented as a combination of a value, or magnitude, a sign, plus or minus, and, if necessary, a decimal point. As a first step in our discussion, let's consider two different approaches to storing just the value of the number in the computer.

The most obvious approach is simply to recognize that there is a direct binary equivalent for any decimal integer. We can simply store any positive or unsigned whole number as its binary representation. This is the approach that we already discussed in Chapter 3. The range of integers that we can store this way is determined by the number of bits available. Thus, an 8-bit storage location can store any **unsigned integer** of value between 0 and 255, a 16-bit storage location, 0–65535. If we must expand the range of integers to be handled, we can provide more bits. A common way to do this is to use multiple storage locations. In Figure 5.1, for example, four consecutive 1-byte storage locations are used to provide 32 bits of range. Used together, these four locations can accept 2^{32}, or 4,294,967,296 different values.

The use of multiple storage locations to store a single binary number may increase the difficulty of calculation and manipulation of these numbers because the calculation may have to be done one part at a time, possibly with carries or borrows between the parts, but the additional difficulty is not unreasonable. Most modern computers provide built-in

FIGURE 5.1

Storage of a 32-bit Data Word

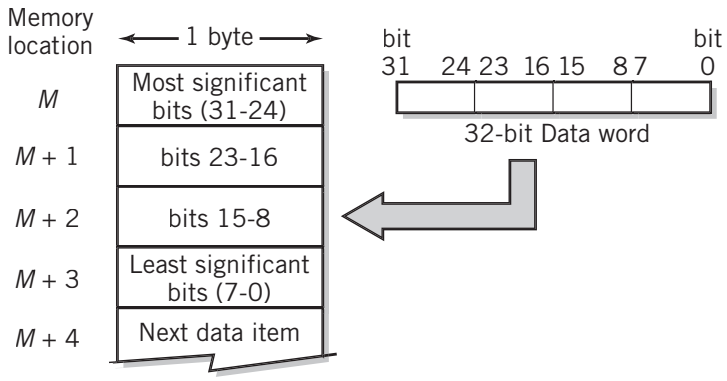


FIGURE 5.2

Value Range for Binary Versus Binary-coded Decimal

No. of Bits	BCD range		Binary range	
4	0–9	1 digit	0–15	1+ digit
8	0–99	2 digits	0–255	2+ digits
12	0–999	3 digits	0–4,095	3+ digits
16	0–9,999	4 digits	0–65,535	4+ digits
20	0–99,999	5 digits	0–1 Million	6 digits
24	0–999,999	6 digits	0–16 Million	7+ digits
32	0–99,999,999	8 digits	0–4 Billion	9+ digits
64	$0–(10^{16}–1)$	16 digits	0–16 Quintillion	19+ digits

instructions that perform data calculations 32 bits or 64 bits at a time, storing the data automatically in consecutive bytes. For other number ranges, and for computers without this capability, these calculations can be performed using software procedures within the computer.

An alternative approach known as **binary-coded decimal** (**BCD**), may be used in some applications. In this approach, the number is stored as a digit-by-digit binary representation of the original decimal integer. Each decimal digit is individually converted to binary. This requires 4 bits per digit. Thus, an 8-bit storage location could hold two binary-coded decimal digits—in other words, one of one hundred different values from 00 to 99. For example, the decimal value 68 would be represented in BCD as 01101000. (Of course you remember that $0110_2 = 6_{10}$ and $1000_2 = 8_{10}$.) Four bits can hold sixteen different values, numbered 0 to F in hexadecimal notation, but with BCD the values A to F are simply not used. The hexadecimal and decimal values for 0 through 9 are equivalent.

The table in Figure 5.2 compares the decimal range of values that can be stored in binary and BCD forms. Notice that for a given number of bits the range of values that can be held using the binary-coded decimal method is substantially less than the range using conventional binary representation. You would expect this because the values A–F are being thrown away for each group of 4 bits. The larger the total number of bits, the more pronounced the difference. With 20 bits, the range for binary is an entire additional decimal digit over the BCD range.

Calculations in BCD are also more difficult, since the computer must break the number into the 4-bit binary groupings corresponding to individual decimal digits and use base 10 arithmetic translated into base 2 to perform calculations. In other words, the calculation for each 4-bit grouping must be treated individually, with arithmetic carries moving from grouping to grouping. Any product or sum of any two BCD integers that exceeds 9 must be reconverted to BCD each time to perform the carries from digit to digit.

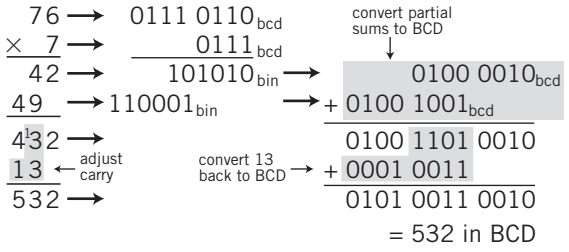
EXAMPLE

One method of performing a ''simple'' one- by two-digit multiplication is shown as an example in Figure 5.3. In the first step, each digit in the multiplicand is multiplied by the single-digit multiplier. This yields the result $7 \times 6 = 42$ in the units place and the result $7 \times 7 = 49$ in the 10's place. Numerically, this corresponds to the result achieved performing the multiplication in decimal, as is shown at the left of the diagram.

To continue, the binary values for 42 and 49 must be converted back to BCD. This is done in the second step. Now the BCD addition takes place. As in the decimal version, the sum of 9 and 4 results in a carry. The binary value 13 must be converted to BCD 3, and the 1 added to the value 4 in the hundreds place. The final result is BCD value 532.

FIGURE 5.3

A Simple BCD Multiplication



If the number contains a decimal point, the same approach can be used, but the application program must keep track of the decimal point's location. For example, many business applications have to maintain full accuracy for real numbers. In many cases, the real numbers being used represent dollars and cents figures. You are aware from Chapter 3 that rational decimal real numbers do not necessarily remain so when converted into binary form. Thus, it is possible that a number converted from decimal to binary and back again may not be exactly the same as the original number. You would not want to add two financial numbers and have the result off by a few cents. (In fact, this was a problem with early versions of spreadsheet programs!)

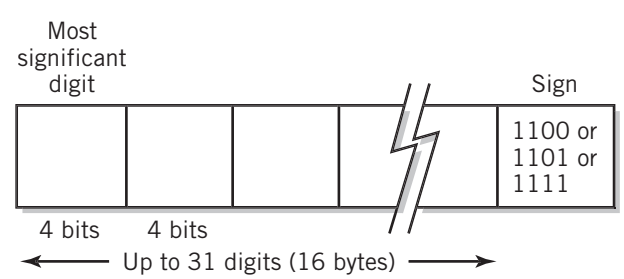
For this reason, business-oriented high-level languages such as COBOL provide formats that allow the user to specify the number of desired decimal places exactly. Large computers support these operations by providing additional instructions for converting, manipulating, and performing arithmetic operations on numbers that are stored in a BCD format.

EXAMPLE

IBM zSeries computers support numbers stored in a BCD format called packed decimal format, shown in Figure 5.4. Each decimal digit is stored in BCD form, two digits to a byte. The most significant digit is stored first, in the high-order bits of the first byte. The sign is stored in the low-order bits of the last byte. Up to thirty-one digits may be stored. The binary values 1100 and 1101 are used for the sign, representing ''+'' and ''−'' respectively. The value 1111 can be used to indicate that the number is unsigned. Since these values do not represent any valid decimal number, it is easy to detect an error, as well as to determine the end of the number. As we noted earlier, the location of the decimal point is not stored and must be maintained by the application program. Intel CPUs provide a more limited packed format that holds two digits (00–99) in a single byte. As an example, the decimal number -324.6 would be stored in packed decimal form as

0000 0011 0010 0100 0110 1101

FIGURE 5.4



The leading 0s are required to make the number fit exactly into 3 bytes. IBM provides additional formats that store data one digit to a byte, but provides no instructions for performing calculations in this format. This format is used primarily as a convenience for conversion between text and packed decimal format. IBM also provides a compressed version of its packed decimal format to save storage space.

Even with computer instructions that perform BCD arithmetic, BCD arithmetic is nearly always much slower. As an alternative, some computers convert each BCD number to binary form, perform the calculation, and then convert the result back to BCD.

Despite its drawbacks, binary-coded decimal representation is sometimes useful, especially in business applications, where it is often desirable to have an exact digit-for-digit decimal equivalent in order to mimic decimal arithmetic, as well as to maintain decimal rounding and decimal precision. Translation between BCD and character form is also easier, since the last 4 bits of ASCII, EBCDIC, and Unicode numeric character forms correspond exactly to the BCD representation of that digit. Thus, to convert from alphanumeric form to BCD you simply chop off everything but the rightmost 4 bits of the character to get its BCD value. This makes BCD an attractive option when the application involves a lot of input and output, but limited calculation. Many business applications fit this description. In most cases, though, binary representation is preferred and used.

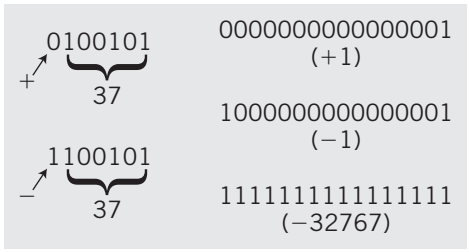
5.2 REPRESENTATIONS FOR SIGNED INTEGERS

With the shortcomings of BCD, it shouldn't surprise you that integers are nearly always stored as binary numbers. As you have already seen, unsigned integers can be converted directly to binary numbers and processed without any special care. The addition of a sign, however, complicates the problem, because there is no obvious direct way to represent the sign in binary notation. In fact, there are several different ways used to represent negative numbers in binary form, depending on the processing that is to take place. The most common of these is known as **2's complement representation**. Before we discuss 2's complement representation, we will take a look at two other, simpler methods: **sign-and-magnitude representation** and **1's complement representation**. Each of these latter methods has some serious limitations for computer use, but understanding these methods and their limitations will clarify the reasoning behind the use of 2's complementation.

Sign-and-magnitude Representation

FIGURE 5.5

Examples of Sign-and-Magnitude Representation



In daily usage, we represent **signed integers** by a plus or minus sign and a value. This representation is known, not surprisingly, as *sign-and-magnitude* representation.

In the computer we cannot use a sign, but must restrict ourselves to 0's and 1's. We could select a particular bit, however, and assign to it values that we agree will represent the plus and minus signs. For example, we could select the leftmost bit and decide that a 0 in this place represents a plus sign and a 1 represents a minus. This selection is entirely arbitrary, but if used consistently, it is as reasonable as any other selection. In fact, this is the representation usually selected. Figure 5.5 shows examples of this representation.

Note that since the leftmost digit is being used as a sign, it cannot represent any value. This means that the positive range of the signed integer using this technique is one-half as large as the corresponding unsigned integer of the same number of bits. On the other hand, the signed integer also has a negative range of equal size to its positive range, so we really haven't lost any capability, but have simply shifted it to the negative region. The total range remains the same, but is redistributed to represent numbers both positive and negative, though in magnitude only half as large.

Suppose 32 bits are available for storage and manipulation of the number. In this case, we will use 1 bit for the sign and 31 bits for the magnitude of the number. By convention, the leftmost, or most significant, bit is usually used as a sign, with 0 corresponding to a plus sign and 1 to a minus sign. The binary range for 32 bits is 0 to 4,294,967,295; we can represent the numbers −2, 147, 483, 647 to +2,147,483,647 this way.

There are several inherent difficulties in performing calculations when using sign-and-magnitude representation. Many of these difficulties arise because the value of the result of an addition depends upon the signs and relative magnitudes of the inputs. This can be easily seen from the following base 10 examples. Since the numbers are exactly equivalent, the same problem of course occurs with binary addition.

EXAMPLE

Consider the base 10 sum of 4 and 2:

$$\begin{array}{r} 4 \\ +\,2 \\ \hline 6 \end{array}$$

The sum of 4 and −2, however, has a different numerical result:

$$\begin{array}{r} 4 \\ -\,2 \\ \hline 2 \end{array}$$

Notice that the addition method used depends on the signs of the operands. One method is used if both signs agree; a different method is used if the signs differ. Even worse,

the presence of a second digit that can result in a carry or borrow changes the result yet again:

$$\begin{array}{r} 2 \\ -\,4 \\ \hline -\,2 \end{array}$$

But

$$\begin{array}{r} 12 \\ -4 \\ \hline 8 \end{array}$$

Interestingly enough, we have been so well trained that we alter our own mental algorithm to fit the particular case without even thinking about it, so this situation might not even have crossed your mind. The computer requires absolute definition of every possible condition, however, so the algorithm must include every possibility; unfortunately, sign-and-magnitude calculation algorithms are complex and difficult to implement in hardware.

In addition to the foregoing difficulty, there are two different binary values for 0,

$$00000000 \text{ and } 10000000$$

representing +0 and −0, respectively. This seems like a minor annoyance, but the system must test at the end of every calculation to assure that there is only a single value for 0. This is necessary to allow program code that compares values or tests a value for 0 to work correctly. Positive 0 is preferred because presenting −0 as an output result would also be confusing to the typical user.

The one occurrence where sign-and-magnitude is a useful representation is when binary-coded decimal is being used. Even though the calculation algorithms are necessarily complex, other algorithms for representing signed integers that you will be introduced to in this chapter are even more impractical when using BCD. Furthermore, as we have already discussed, BCD calculation is complex in any case, so the additional complexity that results from handling sign-and-magnitude representations is just more of the same.

With BCD, the leftmost bit can be used as a sign, just as in the case of binary. With binary, however, using a sign bit cuts the range in half; the effect on range is much less pronounced with BCD. (Remember, though, that BCD already has a much smaller range than binary for the same number of bits.) The leftmost bit in an unsigned BCD integer only represents the values 8 or 9; therefore, using this bit as a sign bit still allows the computer 3 bits to represent the leftmost digit as a number within the range 0–7.

As an example, the range for a signed 16-bit BCD integer would be

$$-7999 \leq \text{value} \leq +7999.$$

Nine's Decimal and 1's Binary Complementary Representations

For most purposes, computers use a different method of representing signed integers known as complementary representation. With this method, the sign of the number is a natural result of the method and does not need to be handled separately. Also, calculations using complementary representation are consistent for all different signed combinations

of input numbers. There are two forms of complementary representation in common use. One, known as the radix complement, is discussed in the next section. In this section, we will introduce a representation known as *diminished radix* complementary representation, so called because the value used as a basis for the complementary operation is *diminished* by one from the radix, or base. Thus, base 10 diminished radix complementary representation uses the value 9 as its basis, and binary uses 1. Although the computer obviously uses the 1's representation, we will introduce the 9's representation first, since we have found that it is easier for most students to understand these concepts in the more familiar decimal system.

NINE'S DECIMAL REPRESENTATION Let us begin by considering a different means of representing negative and positive integers in the decimal number system. Suppose that we manipulate the range of a three-digit decimal number system by splitting the three-digit decimal range down the middle at 500. Arbitrarily, we will allow any number between 0 and 499 to be considered positive. Positive numbers will simply represent themselves. This will allow the value of positive numbers to be immediately identified. Numbers that begin with 5, 6, 7, 8, or 9 in the most significant digit will be treated as representations of negative numbers. Figure 5.6 shows the shift in range.

One convenient way to assign a value to the negative numbers is to allow each digit to be subtracted from the largest numeral in the radix. Thus, there is no carry, and each digit can be converted independently of all others. Subtracting a value from some standard basis value is known as taking the **complement** of the number. Taking the complement of a number is almost like using the basis value as a mirror. In the case of base 10 radix, the largest numeral is 9; thus, this method is called 9's complementary representation.

The facing page shows several examples of this technique.

If we now use the 9's complement technique to assign the negative values to the chart in Figure 5.6, you see that 998 corresponds to a value of −1 and 500 to the value −499. This results in the relationship shown in Figure 5.7.

An important consideration in the choice of a representation is that it is consistent with the normal rules of arithmetic. For the representation to be valid, it is necessary that, for any value within the range,

$$-(-\text{value}) = \text{value}$$

Simply stated, this says that if we complement the value twice, it should return to its original value. Since the complement is just

$$\text{comp} = \text{basis} - \text{value}$$

then complementing twice,

$$\text{basis} - (\text{basis} - \text{value}) = \text{value}$$

which confirms that this requirement is met.

FIGURE 5.6

Range Shifting Decimal Integers

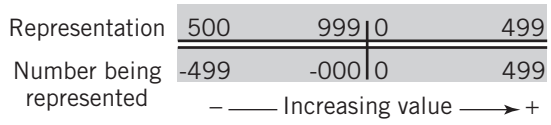
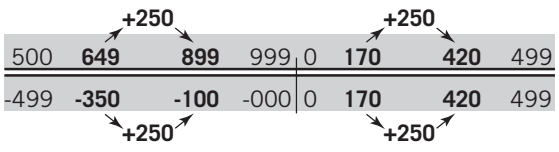


FIGURE 5.7

Addition as a Counting Process



EXAMPLES

Find the 9's complementary representation for the three-digit number −467.

$$\begin{array}{r} 999 \\ -\,467 \\ \hline 532 \end{array}$$

532 represents the value for −467. Notice that the three-digit value range is limited to 0–499, since any larger number would start with a digit of 5 or greater, which is the indicator for a negative number.

■ ■ ■

Find the 9's complementary representation for the four-digit number −467.

$$\begin{array}{r} 9999 \\ -467 \\ \hline 9532 \end{array}$$

Notice that in this system, it is necessary to specify the number of digits, or *word size*, being used. In a four-digit representation, the number (0)532 represents a positive integer, since it is less than 4999 in value. Care is required in maintaining the correct number of digits.

■ ■ ■

What is the sign-and-magnitude value of the four-digit number represented in 9's complement by 3789?

In this case, the leftmost digit is in the range 0–4. Therefore, the number is positive, and is already in correct form. The answer is +3789.

This example emphasizes the difference between the representation of a number in complementary form and the operation of taking the complement of a number. The representation just tells us what the number looks like in complementary form. The operation of finding the complement of a number consists of performing the steps that are necessary to change the number from one sign to the other. Note that if the value represents a negative number, it is necessary to perform the operation if we wish to convert the number into sign-and-magnitude form.

■ ■ ■

What is the sign-and-magnitude value of the four-digit number represented by 9990?

This value is negative. To get the sign-and-magnitude representation for this number, we take the 9's complement:

$$\begin{array}{r} 9999 \\ -\,9990 \\ \hline 9 \end{array}$$

Therefore, 9990 represents the value −9.

Next, let's consider the operation of addition when the numbers being added are in 9's complementary form. When you studied programming language, you learned that modular arithmetic could be used to find the remainder of an integer division. You recall that in modular arithmetic, the count repeats from 0 when a limit, called the *modulus*, is exceeded. Thus, as an example, 4 mod 4 has the value 0 and 5 mod 4 has the value 1.

The 9's complement scale shown in Figure 5.6 shares the most important characteristic of modular arithmetic; namely, in counting upward (from left to right on the scale), when 999 is reached, the next count results in a modular rotation to a value of 0. (Notice that when you reach the right end of the scale, it continues by flowing around to the left end.)

Counting corresponds to addition; thus, to add a number to another is simply to count upward from one number by the other. This idea is illustrated in Figure 5.7. As you can see from the examples in this diagram, simple additions are straightforward and work correctly. To understand how this process works in a "wraparound" situation, consider the example shown in Figure 5.8. As you can see in this case, adding 699 to the value 200 leads to the position 899 by wrapping around the right end. Since 699 is equivalent to −300 and 899 is equivalent to −100, 699 + 200 is equivalent to (−300) +200, and the result of the addition is correct.

The reason this technique works can also be seen in the diagram. The **wraparound** is equivalent to extending the range to include the addition on the scale.

The same final point should also be reached by moving to the left 300 units, which is equivalent to subtracting 300. In fact, the result is off by 1. This occurs because we have again picked a scale with two values for 0, namely, 0 for +0 and 999 for −0. This means that any count that crosses the modulus will be short one count, since 0 will be counted twice. In this particular example, the count to the right, which is the a addition 200 + 699, yielded the correct result, since the modulus was not crossed. The count to the left, the subtraction 200 − 300, is off by one because of the double zero. We could correct for this situation on the chart by moving left an additional count any time the subtraction requires "borrowing" from the modulus. For example, subtracting 200 − 300 requires treating the value 200 as though it were 1200 to stay within the 0–999 range. The borrow can be used to indicate that an additional unit should be subtracted.

Next, consider the situation shown in Figure 5.9. In this case, counting to the right, or adding, also results in crossing the modulus, so an additional count must be added to obtain the correct result. This is an easier situation, however. Since the result of any sum that crosses the modulus must initially contain a carry digit (the 1 in 1099 in the diagram), which is then dropped in the modular addition, it is easy to tell when the modulus has been crossed to the right. We can then simply add the extra count in such cases.

FIGURE 5.8

Addition with Wraparound

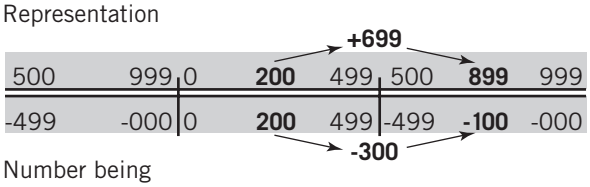


FIGURE 5.9

Addition with Modulus Crossing Representation

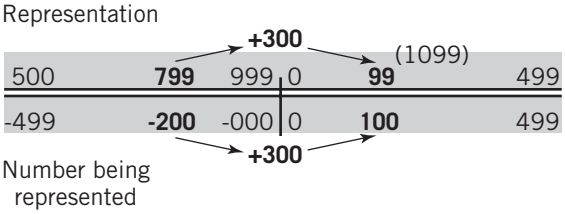
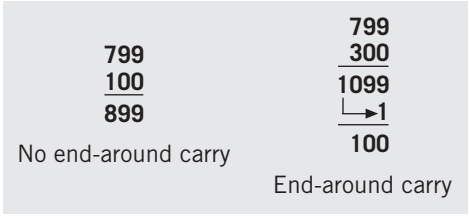


FIGURE 5.10

End-around Carry Procedure



This leads to a simple procedure for adding two numbers in 9's complementary arithmetic: Add the two numbers. If the result flows into the digit beyond the specified number of digits, add the carry into the result. This is known as **end-around carry**. Figure 5.10 illustrates the procedure. Notice that the result is now correct for both examples.

Although we could design a similar algorithm for subtraction, there is no practical reason to do so. Instead, subtraction is performed by taking the complement of the subtrahend (the item being subtracted) and adding to the minuend (the item being subtracted from). In this way, the computer can use a single addition algorithm for all cases.

There is one further consideration. A fixed word size results in a range of some particular fixed size; it is always possible to have a combination of numbers that adds to a result outside the range. This condition is known as **overflow**. If we have a three-digit plus sign word size in a sign-and-magnitude system, and add 500 to 500, the result overflows, since 1000 is outside the range. The fourth digit would be evidence of overflow.

It is just as easy to detect overflow in a 9's complement system, even though the use of modular arithmetic assures that an extra digit will never occur. In complementary arithmetic, numbers that are out of range represent the opposite sign. Thus, if we add

$$300 + 300 = 600\,(\text{i.e.}, -399)$$

both inputs represent positive numbers, but the result is negative. Then the test for overflow is this: *If both inputs to an addition have the same sign, and the output sign is different, overflow has occurred.*

ONE'S COMPLEMENT The computer can use the binary version of the same method of representation that we have just discussed. In base 2, the largest digit is 1. Splitting the range down the middle, as we did before, numbers that begin with 0 are defined to be positive; numbers that begin with 1 are negative. Since

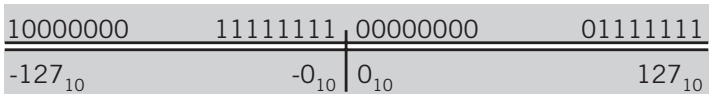
$$\begin{array}{r} 1 \\ -\,0 \\ \hline 1 \end{array} \quad \text{and} \quad \begin{array}{r} 1 \\ -\,1 \\ \hline 0 \end{array}$$

the 1's complement of a number is performed simply by changing every 0 to a 1 and every 1 to a 0. How elegantly simple! This exchange of 0s and 1s is also known as **inversion**. (Of course, this means that both 000 . . . and 111 . . . represent 0, specifically, +0 and −0, respectively.) The 1's complement scale for 8-bit binary numbers is shown in Figure 5.11.

Addition also works in the same way. To add two numbers, regardless of the implied sign of either input, the computer simply adds the numbers as though they were unsigned integers. If there is a carryover into the next bit beyond the leftmost specified bit, 1 is added to the result, following the usual end-around carry rule. Subtraction is done by inverting the subtrahend (i.e., changing every 0 to 1 and every 1 to 0) and adding. Overflows are

FIGURE 5.11

One's Complement Representation



detected in the same way as previously discussed: if both inputs are of the same sign, and the sign of the result is different, overflow has occurred; the result is outside the range. Notice that this test can be performed simply by looking at the leftmost bit of the two inputs and the result.

An important comment about conversion between signed binary and decimal integers in their complementary form: although the technique used is identical between 9's complement decimal and 1's complement binary, the modulus used in the two systems is obviously *not the same!* For example, the modulus in three-digit decimal is 999, with a positive range of 499. The modulus in 8-bit binary is 11111111, or 255_{10}, with a positive range of 01111111, or 127_{10}.

This means that you *cannot* convert directly between 9's complement decimal and 1's complement binary. Instead, you must change the number to sign-and-magnitude representation, convert, and then change the result to the new complementary form. Of course, if the number is positive, this process is trivial, since the complementary form is the same as the sign-and-magnitude form. But you must remember to follow this procedure if the sign is negative. Remember, too, that you must check for overflow to make sure that your number is still in range in the new base.

Here are several examples of 1's complement addition and subtraction, together with the equivalent decimal results:

EXAMPLES

Add

```
00101101 =  45
00111010 =  58
--------------
01100111 = 103
```

■ ■ ■

Add the 16- bit numbers

```
0000000000101101 =   45
1111111111000101 = −58
----------------------
1111111111110010 = −13
```

Note that the addend 1111111111000101 is the inversion of the value in the previous example with eight additional 0s required to fill up 16 bits. The decimal result, −13, is found by inverting 1111111111110010 to 0000000000001101 to get a positive magnitude and adding up the bits.

EXAMPLES

Add

```
                 01101010 = 106
                 11111101 =  -2
               (1)01100111
(end-around carry)  └─────→ +1
                 01101000 = 104
```

■ ■ ■

Subtract

```
  01101010 = 106
−(01011010 =  90)
```

Changing the sign of the addend by inverting

```
                 01101010
                 10100101
               (1)00001111
(end-around carry)  └─────→ +1
                 00010000 = 16
```

■ ■ ■

Add

```
  01000000 =  64
+ 01000001 =  65
  10000001 = −126
```

This is an obvious example of overflow. The correct positive result, 129, exceeds the range for 8 bits. Eight bits can store 256 numbers; splitting the range only allows positive values 0–127.

The overflow situation shown in the last example occurs commonly in the computer,and some high-level languages do not check adequately. In some versions of BASIC, for example, the sum

$$16384 + 16386$$

will show an incorrect result of −32765 or −32766. (The latter result comes from use of a different complementary representation, discussed in the next section.) What has happened is that overflow has occurred in a system that uses 16 bits for integer calculations. The positive range limit for 16 bits is +32767 (a 0 for the sign plus fifteen 1s). Since the sum of 16384 and 16386 is 32770, the calculation overflows. Unfortunately, the user may never notice, especially if the overflowing calculation is buried in a long series of calculations. A good programmer takes such possibilities into account when the program is written. (This type of error caused some embarrassment when it showed up in a recent version of Microsoft Excel.)

Ten's Complement and 2's Complement

FIGURE 5.12

Ten's Complement Scale

Representation	500	999 | 0		499
Number being represented	-500	-001 | 0		499
		–	+	

TEN'S COMPLEMENT You have seen that complementary representation can be effective for the representation and calculation of signed integer numbers. As you are also aware, the system that we have described,which uses the largest number in the base as its complementary reflection point, suffers from some disadvantages that result from the dual zero on its scale.

By shifting the negative scale to the right by one, we can create a complementary system that has only a single zero. This is done by using the radix as a basis for the complementary operation. In decimal base, this is known as the 10's complement representation. The use of this representation will simplify calculations. The trade-off in using 10's complement representation is that it is slightly more difficult to find the complement of a number. A three-digit decimal scale is shown in Figure 5.12. Be sure to notice the differences between this diagram and Figure 5.6.

The theory and fundamental technique for 10's complement is the same as that for 9's complement. The 10's complement representation uses the modulus as its reflection point. The modulus for a three-digit decimal representation is 1000, which is one larger than the largest number in the system, 999.

Complements are found by subtracting the value from the modulus, in this case, 1000. This method assures a single zero, since (1000 − 0) mod 1000 is zero. Again, as with the previously discussed complementary methods, notice that the complement of the complement results in the original value. See the examples on the facing page.

There is an alternative method for complementing a 10's complement number. First, observe that

$$1000 = 999 + 1$$

You recall that the 9's complement was found by subtracting each digit from 9:

$$9\text{'s comp} = 999 - \text{value}$$

From the previous equation, the 10's complement can be rewritten as

$$10\text{'s comp} = 1000 - \text{value} = 999 + 1 - \text{value} = 999 - \text{value} + 1$$

or, finally,

$$10\text{'s comp} = 9\text{'s comp} + 1$$

This gives a simple alternative method for computing the 10's complement value: find the 9's complement, which is easy, and add 1 to the result. Either method gives the same result. You can use whichever method you find more convenient. This alternative method is usually easier computationally, especially when working with binary numbers, as you will see.

Addition in 10's complement is particularly simple. Since there is only a single zero in 10's complement, sums that cross the modulus are unaffected. Thus, the carry that results

EXAMPLES

Find the 10's complement of 247.

As a reminder, note that the question asks for the 10's *complement* of 247, not the 10's *complement representation*. Since 247 represents a positive number, its 10's complement representation is, of course, 247.

The 10's complement of 247 is

$$1000 - 247 = 753$$

Since 247 is a positive representation, 753 represents the value −247.

■ ■ ■

Find the 10's complement of 17.

As in the 9's complement work, we always have to be conscious of the number of specified digits. Since all the work so far has assumed that the numbers contain three digits, let's solve this problem from that assumption:

$$1000 - 017 = 983$$

■ ■ ■

Find the sign and magnitude of the three-digit number with 10's complement representation: 777

Since the number begins with a 7, it must be negative. Therefore,

$$1000 - 777 = 223$$

The sign-magnitude value is −223.

when the addition crosses the zero point is simply ignored. To add two numbers in 10's complement, one simply adds the digits; any carry beyond the specified number of digits is thrown away. (Actually, in the computer, the carry is saved in a special "carry bit," just in case it is to be used to extend the addition to another group of bits for multiple-word additions.) Subtraction is again performed by inverting the subtrahend and adding.

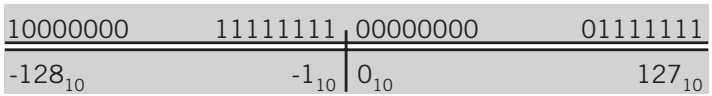
The range of numbers in 10's complement for three digits can be seen in Figure 5.12. Of particular interest is the fact that the positive and negative regions are of different size: there is one negative number, 500, that cannot be represented in the positive region. (The 10's complement of 500 is itself.) This peculiarity is a consequence of the fact that the total range of numbers is even for any even-numbered base, regardless of word size (in this case, 10^W). Since one value is reserved for zero, the number of remaining values to be split between positive and negative is odd and, thus, could not possibly be equal.

TWO'S COMPLEMENT. Two's complement representation for binary is, of course, similar to 10's complement representation for decimal. In binary form, the modulus consists of a base 2 "1" followed by the specified number of 0s. For 16 bits, for example, the modulus is

10000000000000000

FIGURE 5.13

Two's Complement Representation



As was true for the 10's complement, the 2's complement of a number can be found in one of two ways: either subtract the value from the modulus or find the 1's complement by inverting every 1 and 0 and adding 1 to the result.

The second method is particularly well suited to implementation in the computer, but you can use whichever method you find more convenient.

Figure 5.13 shows an 8-bit scale for 2's complement representation.

Two's complement addition, like 10's complement addition in decimal, consists of adding the two numbers *mod* <the modulus>. This is particularly simple for the computer, since it simply means eliminating any 1s that don't fit into the number of bits in the word. Subtraction and overflow are handled as we have already discussed.

As in 10's complement, the range is unevenly divided between positive and negative. The range for 16 bits, for example, is $-32768 \leq \text{value} \leq 32767$.

There are many 2's complement problems at the end of the chapter for you to practice on.

The use of 2's complement is more common in computers than is 1's complement, but both methods are in use. The trade-off is made by the designers of a particular computer: 1's complement makes it easier to change the sign of a number, but addition requires an extra end-around carry step. One's complement has the additional drawback that the algorithm must test for and convert −0 to 0 at the end of each operation. Two's complement simplifies the addition operation at the expense of an additional add operation any time the sign change operation is required.

As a final note, before we conclude our discussion of binary complements, it is useful to be able to predict approximate sizes of integers that are represented in complementary form without going through the conversion. A few hints will help:

1. Positive numbers are always represented by themselves. Since they always start with 0, they are easily identified.
2. Small negative numbers, that is, negative numbers close to 0, have representations that start with large numbers of 1s. The number −2 in 8-bit 2's complement, for example, is represented by

   11111110

   whereas −128, the largest negative 2's complement number, is represented by

   10000000

   This is evident from the scale in Figure 5.13.
3. Since there is only a difference in value of 1 between 1's and 2's complement representations of negative numbers (positive numbers are, of course, identical

in both representations), you can get a quick idea of the value in either representation simply by inverting all the 1s and 0s and approximating the value from the result.

Overflow and Carry Conditions

We noted earlier in this discussion that overflows occur when the result of a calculation does not fit into the fixed number of bits available for the result. In 2's complement, an addition or subtraction overflow occurs when the result overflows into the sign bit. Thus, overflows can occur only when both operands have the same sign and can be detected by the fact that the sign of the result is opposite that of the operands.

Computers provide a flag that allows a programmer to test for an overflow condition. The overflow flag is set or reset each time a calculation is performed by the computer. In addition, the computer provides a **carry flag** that is used to correct for carries and borrows that occur when large numbers must be separated into parts to perform additions and subtractions. For example, if the computer has instructions that are capable of adding two 32-bit numbers, it would be necessary to separate a 64-bit number into two parts, add the least significant part of each, then add the most significant parts, together with any carry that was generated by the previous addition. For normal, single precision 2's complement addition and subtraction the carry bit is ignored.

Although overflow and carry procedures operate similarly, they are not quite the same, and can occur independently of each other. The carry flag is set when the result of an addition or subtraction exceeds the fixed number of bits allocated, without regard to sign. It is perhaps easiest to see the difference between overflow and carry conditions with an example. This example shows each of the four possible outcomes that can result from the addition of two 4-bit 2's complement numbers.

EXAMPLE

```
(+4) + (+2)                          (+4) + (+6)
0100          no overflow,           0100          overflow,
0010          no carry               0110          nocarry
0110 = (+6)   the result is correct  1010 = (−6)   the result is incorrect

(−4) + (−2)                          (−4) + (−6)
 1100         no overflow,            1100         overflow,
 1110         carry                   1010         Carry
110106 = (−6) ignoring carry,        10110 = (+6)   ignoring the carry,
              the result is correct                 the result is incorrect
```

If an overflow occurs on any but the most significant part of a multiple part addition, it is ignored (see exercise 5.13).

Other Bases

Any even-numbered base can be split the same way to represent signed integers in that base. Either the modulus or the largest-digit value can be used as a mirror for the complementary

representation. Odd bases are more difficult: either the range must be split unevenly to use the leftmost digit as an indicator, or the second left digit must be used together with the first to indicate whether the represented number is positive or negative. We will not consider odd bases any further.

Of particular interest are the corresponding 7's and 8's complements in octal and 15's and 16's complements in hexadecimal. These correspond exactly to 1's and 2's complement in binary, so you can use calculation in octal or hexadecimal as a shorthand for binary.

As an example, consider four-digit hexadecimal as a substitute for 16-bit binary. The range will be split down the middle, so that numbers starting with $0–7_{16}$ are positive and those starting with 8–F are negative. But note that hex numbers starting with 8–F all have a binary equivalent with 1 in the leftmost place, whereas 0–7 all start with 0. Therefore, they conform exactly to the split in 16-bit binary.

You can carry the rest of the discussion by yourself, determining how to take the complement, and how to add, from the foregoing discussions. There are practice examples at the end of the chapter.

Finally, note that since binary-coded decimal is essentially a base 10 form, the use of complementary representation for BCD would require algorithms that analyze the first digit to determine the sign and then perform 9's or 10's complement procedures. Since the purpose of BCD representation is usually to simplify the conversion process, it is generally not practical to use complementary representation for signed integers in BCD.

Summary of Rules for Complementary Numbers

The following points summarize the rules for the representation and manipulation of complementary numbers, both radix and diminished radix, in any even number base. For most purposes you will be interested only in 2's complement and 16's complement:

1. Remember that the word "complement" is used in two different ways. To complement a number, or take the complement of a number, means to change its sign. To find the complementary representation of a number means to translate or identify the representation of the number just as it is given.
2. Positive numbers are represented the same in complementary form as they would be in sign and magnitude form. These numbers will start with 0, 1, . . . N/2–1. For binary numbers, positive numbers start with 0, negative with 1.
3. To go from negative sign-and-magnitude to complementary form, or to change the sign of a number, simply subtract each number from the largest number in the base (diminished radix) or from the value 100 . . . , where each zero corresponds to a number position (radix). Remember that implied zeros must be included in the procedure. Alternatively, the radix form may be calculated by adding 1 to the diminished radix form. For 2's complement, it is usually easiest to invert every digit and add 1 to the result.
4. To get the sign-and-magnitude representation for negative numbers, use the procedure in (2) to get the magnitude. The sign will, of course, be negative. Remember that the word size is fixed; there may be one or more implied 0s at the beginning of a number that mean the number is really positive.

5. To add two numbers, regardless of sign, simply add in the usual way. Carries beyond the leftmost digit are ignored in radix form, added to the result in diminished radix form. To subtract, take the complement of the subtrahend and add.
6. If we add two complementary numbers of the same sign and the result is of opposite sign, the result is incorrect. Overflow has occurred.

5.3 REAL NUMBERS

A Review of Exponential Notation

Real numbers add an additional layer of complexity. Because the number contains a radix point (decimal in base 10, binary in base 2), the use of complementary arithmetic must be modified to account for the fractional part of the number. The representation of real numbers in exponential notation simplifies the problem by separating the number into an integer, with a separate **exponent** that places the **radix point** correctly. As before, we first present the techniques in base 10, since working with decimal numbers is more familiar to you. Once you have seen the methods used for the storage and manipulation of floating point numbers, we will then extend our discussion to the binary number system. This discussion will include the conversion of floating point numbers between the decimal and binary bases (which requires some care) and the consideration of floating point formats used in actual computer systems.

Consider the whole number

$$12345$$

If we allow the use of exponents, there are many different ways in which we can represent this number. Without changing anything, this number can be represented as

$$12345 \times 10^0$$

If we introduce decimals, we can easily create other possible representations. Each of these alternative representations is created by shifting the decimal point from its original location. Since each single-place shift represents a multiplication or division of the value by the base, we can decrease or increase the exponent to compensate for the shift. For example, let us write the number as a decimal fraction with the decimal point at the beginning:

$$0.12345 \times 10^5$$

or, as another alternative,

$$123450000 \times 10^{-4}$$

or even,

$$0.0012345 \times 10^7$$

Of course, this last representation will be a poor choice if we are limited to five digits of magnitude,

$$0.00123 \times 10^7$$

since we will have sacrificed two digits of precision in exchange for the two zeros at the beginning of the number which do not contribute anything to the precision of the number. (You may recall from previous math courses that they are known as insignificant digits.) The other representations do retain full precision, and any one of these representations would be theoretically as good as any other. Thus, our choice of representation is somewhat arbitrary and will be based on more practical considerations.

The way of representing numbers described here is known as **exponential notation** or, alternatively, as scientific notation. Using the exponential notation for numbers requires the specification of four separate components to define the number. These are

1. The sign of the number ("+", for our original example)
2. The magnitude of the number, known as the **mantissa** (12345)
3. The sign of the exponent ("+")
4. The magnitude of the exponent (3, see below)

Two additional pieces of information are required to complete the picture:

5. The base of the exponent (in this case, 10)
6. The location of the decimal (or other base) radix point

Both these latter factors are frequently unstated, yet extremely important. In the computer, for example, the base of the exponent is usually, but not always, specified to be 2. In some computers, 16 or 10 may be used instead, and it is obviously important to know which is being used if you ever have to read the numbers in their binary form. The location of the decimal point (or *binary* point, if we're working in base 2) is also an essential piece of information. In the computer, the binary point is set at a particular location in the number, most commonly the beginning or the end of the number. Since its location never changes, it is not necessary to actually store the point. Instead, the location of the binary point is implied.

Knowing the location of the point is, of course, essential. In the example that accompanies the rules just given, the location of the decimal point was not specified. Reading the data suggests that the number might be

$$+12345 \times 10^{+3}$$

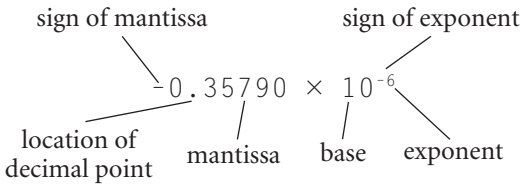
which, of course, is not correct if we're still using the number from our original example. The actual placement of the decimal point should be

$$12.345 \times 10^{3}$$

Let us summarize these rules by showing another example, with each component specifically marked. Assume that the number to be represented is

$$-0.0000003579$$

One possible representation of this number is



Floating Point Format

As was the case with integers, floating point numbers will be stored and manipulated in the computer using a "standard", predefined format. For practical reasons, a multiple of 8 bits is usually selected as the word size. This will simplify the manipulation and arithmetic that is performed with these numbers.

In the case of integers, the entire word is allocated to the magnitude of the integer and its sign. For floating point numbers, the word must be divided: part of the space is reserved for the exponent and its sign; the remainder is allocated to the mantissa and its sign. The base of the exponent and the implied location of the binary point are standardized as part of the format and, therefore, do not have to be stored at all.

You can understand that the format chosen is somewhat arbitrary, since you have already seen that there are many different ways to represent a floating point number. Among the decisions made by the designer of the format are the number of digits to use, the implied location of the binary or decimal point, the base of the exponent, and the method of handling the signs for the mantissa and the exponent.

For example, suppose that the standard word consists of space for seven decimal digits and a sign:

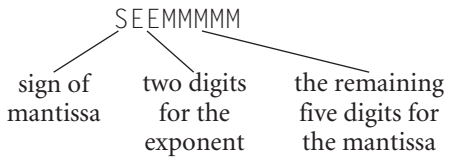
SMMMMMMM

This format would allow the storage of any integer in the range

$$-9,999,999 < I < +99,999,999$$

with full, seven-digit precision. Numbers of magnitude larger than 9,999,999 result in overflow. Numbers of magnitude less than 1 cannot be represented at all, except as 0.

For floating point numbers, we might assign the digits as follows:



In addition we have to specify the implied location for the decimal point.

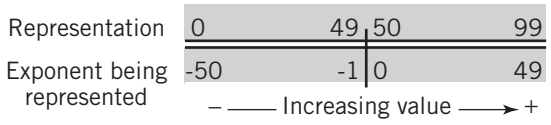
In this example we have "traded" two digits of exponent in exchange for the loss of two digits of precision. We emphasize that we have not increased the number of values that can be represented by seven digits. Seven digits can represent exactly 10,000,000 different values, no matter how they are used. We have simply chosen to use those digits differently—to increase the expressible range of values by giving up precision throughout the range. If we wish to increase the precision, one option is to increase the number of digits.

There are other possible trades. We could, for example, increase precision by another digit by limiting the exponent to a single digit. This might not be as limiting as it first appears. Since each increment or decrement of the exponent changes the number by a factor equivalent to the base (in this case, 10), a fairly substantial range of numbers can be accommodated with even a single digit, in this case 10^9 to 10^0, or 1 billion to 1.

The sign digit will be used to store the sign of the mantissa. Any of the methods shown earlier in this chapter for storing the sign and magnitude of integers could be used for

FIGURE 5.14

Excess-50 Representation



the mantissa. Most commonly, the mantissa is stored using sign-magnitude format. A few computers use complementary notation.

Notice that we have made no specific provision for the sign of the exponent within the proposed format. We must therefore use some method that includes the sign of the exponent within the digits of the exponent itself. One method that you have already seen for doing this is the complementary representation. (Since the exponent and mantissa are independent of each other, and are used differently in calculations, there is no reason to assume that the same representation would be used for both.)

The manipulations used in performing exponential arithmetic allow us to use a simple method for solving this problem. If we pick a value somewhere in the middle of the possible values for the exponent, for example, 50 when the exponent can take on values 0 to 99, and declare that value to correspond to the exponent 0, then every value lower than that will be negative and those above will be positive. Figure 5.14 shows the scale for this offset technique.

What we have done is *offset*, or bias, the value of the exponent by our chosen amount. Thus, to convert from exponential form to the format used in our example, we add the offset to the exponent, and store it in that form. Similarly, the stored form can be returned to our usual exponential notation by subtracting the offset.

This method of storing the exponent is known as **excess-*N* notation**, where N is the chosen midvalue. It is simpler to use for exponents than the complementary form, and appropriate to the calculations required on exponents. In our example we have used excess-50 notation. This allows us to store an exponential range of -50 to $+49$, corresponding to the stored values 00 to 99. We could, if we wished, pick a different offset value, which would expand our ability to handle larger numbers at the expense of smaller numbers, or vice versa.

If we assume that the implied decimal point is located at the beginning of the five-digit mantissa, excess-50 notation allows us a magnitude range of

$$0.00001 \times 10^{-50} < \text{number} < 0.99999 \times 10^{+49}$$

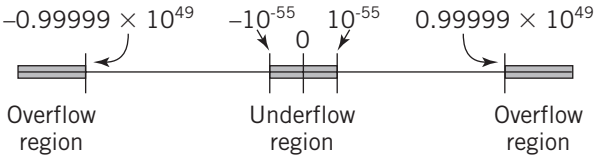
This is an obviously much larger range than that possible using integers, and at the same time gives us the ability to express decimal fractions. In practice, the range may be slightly more restricted, since many format designs require that the most significant digit not be 0, even for very small numbers. In this case, the smallest expressible number becomes 0.10000×10^{-50}, not a great limitation. The word consisting of all 0s is frequently reserved to represent the special value 0.0.

If we were to pick a larger (or smaller) value for the offset, we could skew the range to store smaller (or larger) numbers. Generally, values somewhere in the midrange seem to satisfy the majority of users, and there seems little reason to choose any other offset value.

Notice that, like the integer, it is still possible, although very difficult, to create an overflow by using a number of magnitude too large to be stored. With floating point numbers it is also possible to have **underflow**, where the number is a decimal fraction of magnitude too small to be stored. The diagram in Figure 5.15 shows the regions of underflow and overflow for our example. Note that in the diagram 0.00001×10^{-50} is expressed equivalently as 10^{-55}.

FIGURE 5.15

Regions of Overflow and Underflow



There is one more consideration. As you are already aware, the computer is actually capable of storing only numbers, no signs or decimal points. We have already handled the decimal point by establishing a fixed, implied point. We must also represent the sign of the number in a way that takes this limitation into account.

Here are some examples of floating point decimal representations. The format used is that shown on page 157: a sign, two digits of exponent stored excess-50, and five digits of mantissa. The value 0 is used to represent a "+" sign; 5 has been arbitrarily chosen to represent a "–" sign, just as 1 is usually chosen within the computer for the same purpose. The base is, of course, 10; the implied decimal point is at the beginning of the mantissa. You should look at these examples carefully to make sure that you understand all the details of the **floating point format**.

EXAMPLES

$$05324657 = 0.24657 \times 10^{3} = 246.57$$
$$54810000 = -0.10000 \times 10^{-2} = -0.0010000$$

(Note that five significant digits are maintained.)

$$55555555 = -0.55555 \times 10^{5} = -55555$$
$$04925000 = 0.25000 \times 10^{-1} = 0.025000$$

Normalization and Formatting of Floating Point Numbers

The number of digits used will be determined by the desired precision of the numbers. To maximize the precision for a given number of digits, numbers will be stored whenever possible with no leading zeros. This means that, when necessary, numbers are shifted left by increasing the exponent until leading zeros are eliminated. This process is called **normalization**.

Our standard format, then, will consist of a mantissa of fixed, predetermined size with a decimal point placed at a fixed, predetermined location. The exponent will be adjusted so that numbers will be stored in this format with no leading zeros.

As an example, let us set up a standard format that reflects the storage capabilities suggested in the previous section. Our format will consist of a sign and five digits, with the decimal point located at the beginning of the number:

$$.MMMMM \times 10^{EE}$$

There are four steps required to convert any decimal number into this standard format:

1. Provide an exponent of 0 for the number, if an exponent was not already specified as part of the number.
2. Shift the decimal point left or right by increasing or decreasing the exponent, respectively, until the decimal point is in the proper position.

3. Shift the decimal point right, if necessary, by decreasing the exponent, until there are no leading zeros in the mantissa.
4. Correct the precision by adding or discarding digits as necessary to meet the specification. We discard or round any digits in excess of the specified precision by eliminating the least significant digits. If the number has fewer than the specified number of digits, we supply zeros at the end.

Once we have normalized the number and put it into a standard exponential form, we can perform a fifth step to convert the result into the desired word format. To do this, we change the exponent into excess-50 notation and place the digits into their correct locations in the word.

Conversions between integer and floating point format are similar. The integer is treated as a number with an implied radix point at the end of the number. In the computer, an additional step may be required to convert the integer between complementary and sign-magnitude format to make it compatible with floating point format.

Here are some examples of a decimal to floating point format conversion:

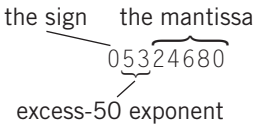
EXAMPLES

Convert the number

$$246.8035$$

into our standard format.

1. Adding an exponent makes the number
$$246.8035 \times 10^0$$
2. We shift the decimal to the left three places, thereby increasing the exponent by 3:
$$0.2468035 \times 10^3$$
3. Since the number is already normalized (no leading zeros), there is no adjustment required.
4. There are seven digits, so we drop the two least significant digits, and the final exponential representation is
$$0.24680 \times 10^3$$
5. The exponent is 3, which in excess-50 notation is represented as 53. If we represent a "+" sign with the digit 0, and a "–" sign with the digit 5 (this choice is totally arbitrary, but we needed to select some digits since the sign itself cannot be stored), the final stored result becomes



■ ■ ■

Assume that the number to be converted is

$$1255 \times 10^{-3}$$

1. The number is already in exponential form.
2. We must shift the decimal to the left four places, so the number becomes

   $$0.1255 \times 10^{+1}$$

   The positive exponent results from adding 4 to the original −3 exponent.
3. The number is normalized, so no additional adjustment is required.
4. A zero is added to provide five digits of precision. The final result in exponential form is

   $$0.12550 \times 10^{1}$$

5. The exponent in excess-50 notation becomes 51, and the result in word format is

   $$05112550$$

■ ■ ■

Assume that the number to be converted is

$$-0.00000075$$

1. Converting to exponential notation, we have

   $$-0.00000075 \times 10^{0}$$

2. The decimal point is already in its correct position, so no modification is necessary.
3. Normalizing, the number becomes

   $$-0.75 \times 10^{-6}$$

4. And the final exponential result,

   $$-0.75000 \times 10^{-6}$$

5. In our word format, this becomes

   $$54475000$$

Although the technique is simple and straightforward, it will still require some practice for you to feel comfortable with it. We suggest that you practice with a friend, inventing numbers for each other to put into a standard format.

Some students have a bit of difficulty remembering whether to increase or decrease the exponent when shifting the number left or right. There is a simple method that may help you to remember which way to go: when you shift the decimal to the right, it makes the resulting number larger. (For example, 1.5 becomes 15.) Thus, the exponent must become smaller to keep the number the same as it was originally.

A Programming Example

Perhaps representing the steps as a pseudocode procedure will clarify these concepts even further. The procedure in Figure 5.16 converts numbers in normal decimal format to the floating point format

SEEMMMMM

The implied decimal point is at the beginning of the mantissa, and the sign is stored as 0 for positive, 5 for negative. The mantissa is stored in sign-magnitude format. The exponent is stored in excess-50 format. The number 0.0 is treated as a special case, with an all-zero format.

We suggest that you trace through the procedure carefully, until you understand each step.

FIGURE 5.16

A Procedure to Convert Decimal Numbers to Floating Point Format

```
function ConvertToFloat();
//variables used:
real decimalin; //decimal number to be converted
//components of the output
integer sign,  exponent,  integermantissa;
float mantissa; //used for normalization
integer floatout; //final form of output
{
     if (decimalin == 0.0) floatout = 0;
     else {
          if (decimalin > 0.0) sign = 0;
          else sign = 50000000;
          exponent = 50;
          StandardizeNumber;
          floatout = sign + exponent * 100000 + integermantissa;
          } //end else

function StandardizeNumber(); {
     mantissa = abs (mantissa);
     //adjust the decimal to fall between 0.1 and 1.0.
          while (mantissa >= 1.00) {
               mantissa = mantissa / 10.0;
               exponent = exponent + 1;
          } //end while
          while (mantissa < 0.1) {
               mantissa = mantissa * 10.0;
               exponent = exponent - 1;
          } //end while
         integermantissa = round (10000.0 * mantissa)
     } //end function StandardizeNumber
} //end ConvertToFloat
```

Floating Point Calculations

Floating point arithmetic is obviously more complex than integer arithmetic. First, the exponent and the mantissa have to be treated separately. Therefore, each has to be extracted from each number being manipulated.

ADDITION AND SUBTRACTION You recall that in order to add or subtract numbers that contain decimal fractions, it is necessary that the decimal points line up. When using exponential notation, it is thus a requirement that the implied decimal point in both numbers be in the same position; the exponents of both numbers must agree.

The easiest way to align the two numbers is to shift the number with the smaller exponent to the right a sufficient number of spaces to increase the exponent to match the larger exponent. This process inserts insignificant zeros at the beginning of the number. Note that this process protects the precision of the result by maintaining all the digits of the larger number. It is the least significant digits of the smaller number that will disappear.

Once alignment is complete, addition or subtraction of the mantissas can take place. It is possible that the addition or subtraction may result in an overflow of the most significant digit. In this case, the number must be shifted right and the exponent incremented to accommodate the overflow. Otherwise, the exponent remains unchanged.

It is useful to notice that the exponent can be manipulated directly in its excess form, since it is the difference in the two exponents that is of interest rather than the value of the exponent itself. It is thus not necessary to change the exponents to their actual values in order to perform addition or subtraction.

EXAMPLE

Add the two floating point numbers

```
05199520
04967850
```

Assume that these numbers are formatted using sign-and-magnitude notation for the mantissa and excess-50 notation for the exponent. The implied decimal point is at the beginning of the mantissa, and base 10 is used for the exponent.

Shifting the lower mantissa right two places to align the exponent, the two numbers become

```
05199520
0510067850
```

Adding the mantissas, the new mantissa becomes

```
(1)0019850
```

We have put the 1 in parentheses to emphasize the fact that it is a carry beyond the original left position of the mantissa. Therefore, we must again shift the mantissa right one place and increment the exponent to accommodate this digit:

```
05210019(850)
```

Rounding the result to five places of precision, we finally get

```
05210020
```

Checking the result,

$$05199520 = 0.99520 \times 10^{1} = 9.9520$$
$$04967850 = 0.67850 \times 10^{-1} = \underline{0.06785}$$
$$10.01985 = 0.1001985 \times 10^{2}$$

which converts to the result that we previously obtained.

MULTIPLICATION AND DIVISION Alignment is not necessary in order to perform multiplication or division. Exponential numbers are multiplied (or divided) by multiplying (dividing) the two mantissas and adding (subtracting) the two exponents. The sign is dealt with separately in the usual way. This procedure is relatively straightforward. There are two special considerations that must be handled, however:

1. Multiplication or division frequently results in a shifting of the decimal point (e.g., $0.2 \times 0.2 = 0.04$) and normalization must be performed to restore the location of the decimal point and to maintain the precision of the result.
2. We must adjust the excess value of the resulting exponent. Adding two exponents, each of which contains an excess value, results in adding the excess value to itself, so the final exponent must be adjusted by subtracting the excess value from the result. Similarly, when we subtract the exponents, we subtract the excess value from itself, and we must restore the excess value by adding it to the result.

EXAMPLE

This is seen easily with an example. Assume we have two numbers with exponent 3. Each is represented in excess-50 notation as 53. Adding the two exponents,

$$\begin{array}{r} 53 \\ \underline{53} \\ 106 \end{array}$$

We have added the value 50 twice, and so we must subtract it out to get the correct excess-50 result:

$$\begin{array}{r} 106 \\ \underline{-50} \\ 56 \end{array}$$

3. The multiplication of two five-digit normalized mantissas yields a ten-digit result. Only five digits of this result are significant, however. To maintain full, five-digit precision, we must first normalize and then round the normalized result back to five digits.

EXAMPLE

Multiply the two numbers

$$\begin{array}{r} 05220000 \\ \underline{\times 04712500} \end{array}$$

Adding the exponents and subtracting the offset results in a new, excess-50 exponent of

$$52 + 47 - 50 = 49$$

Multiplying the two mantissas,

$$0.20000 \times 0.12500 = 0.025000000$$

Normalizing the result by shifting the point one space to the right decreases the exponent by one, giving a final result

04825000

Checking our work,

05220000 is equivalent to 0.20000×10^{2},
04712500 is equivalent to 0.12500×10^{-3}

which multiplies out to

$$0.0250000000 \times 10^{-1}$$

Normalizing and rounding,

$$0.0250000000 \times 10^{-1} = 0.25000 \times 10^{-2}$$

which corresponds to our previous result.

Floating Point in the Computer

The techniques discussed in the previous section can be applied directly to the storage of floating point numbers in the computer simply by replacing the digits with bits. Typically, 4, 8, or 16 bytes are used to represent a floating point number. In fact, the few differences that do exist result from "tricks" that can be played when "0" and "1" are the only options.

A typical floating point format might look like the diagram in Figure 5.17. In this example, 32 bits (4 bytes) are used to provide a range of approximately 10^{-38} to 10^{+38}. With 8 bits, we can provide 256 levels of exponent, so it makes sense to store the exponent in excess-128 notation.

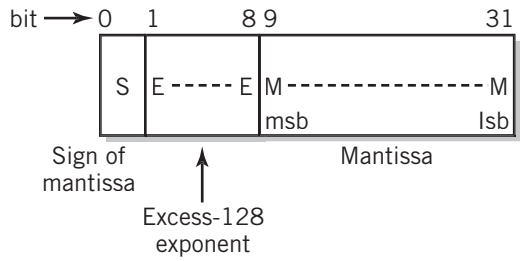
EXAMPLE

Here are some examples of binary floating point format using this notation. Again we have assumed that the binary point is at the start of the mantissa. The base of the exponent is 2.

0 10000001 11001100000000000000000 =
+1.10011000000000000000
1 10000100 10000111100000000000000 =
−1000.01111000000000000
1 01111110 10101010101010101010101 =
−0.00101010101010101010101

FIGURE 5.17

Typical Floating Point Format



Thanks to the nature of the binary system, the 23 bits of mantissa can be stretched to provide 24 bits of precision, which corresponds to approximately seven decimal digits of precision. Since the leading bit of the mantissa must be "1" if the number is normalized, there is no need to store the most significant bit explicitly. Instead, the leading bit can be treated implicitly, similar to the binary point.

There are three potential disadvantages to using this trick. First, the assumption that the leading bit is always a "1" means that we cannot store numbers too small to be normalized, which slightly limits the small end of the range. Second, any format that may require a "0" in the most significant bit for any reason cannot use this method. Finally, this method requires that we provide a separate way to store the number 0.0, since the requirement that the leading bit be a "1" makes a mantissa of 0.0 an impossibility!

Since the additional bit doubles the available precision of the mantissa in all numbers, the slightly narrowed range is usually considered an acceptable trade-off. The number 0.0 is handled by selecting a particular 32-bit word and assigning it the value 0.0. Twenty-four bits of mantissa corresponds to approximately seven decimal digits of precision.

Don't forget that the base and implied binary point must also be specified.

There are many variations, providing different degrees of precision and exponential range, but the basic techniques for handling floating point numbers in the computer are identical to those that we have already discussed in the previous sections of this chapter.

IEEE 754 STANDARD Most current computers conform to IEEE 754 standard formats. The IEEE Computer Society is a society of computer professionals. Among its tasks, the IEEE Computer Society develops technical standards for use by the industry. The IEEE 754 standard defines formats for 32-bit and 64-bit floating point arithmetic. Instructions built into modern computers utilize the standard to perform floating point arithmetic, normalization, and conversion between integer and floating point representations internally under program command. The standard also facilitates the portability of programs between different computers that support the standard.

In addition to the IEEE 754 format, there are a number of older, machine-specific formats still in use for legacy data. The Macintosh also provides an additional 80-bit format. Sun UltraSparc and IBM mainframes systems include additional 128-bit formats. The Intel IA-64 architecture conforms to the IEEE 754 format, but also provides 64-bit significand/17-bit exponent range capability; the programmer can set individual precision control and widest-range exponent values in the floating point status register for additional flexibility.

The standard defines a **single-precision floating point format** consisting of 32 bits, divided into a sign, 8 bits of exponent, and 23 bits of mantissa. Since normalized numbers must always start with a 1, the leading bit is not stored, but is instead implied; this bit is located to the left of the implied binary point. Thus, numbers are normalized to the form

```
1.MMMMMMM...
```

FIGURE 5.18

IEEE Standard 32-bit Floating Point Value Definition

Exponent	Mantissa	Value
0	± 0	0
0	not 0	$\pm 2^{-126} \times 0.M$
1-254	any	$\pm 2^{E-127} \times 1.M$
255	± 0	$\pm \infty$
255	not 0	NaN (Not a Number)

The exponent is formatted using excess-127 notation, with an implied base of 2. This would theoretically allow an exponent range of 2^{-127} to 2^{128}. In actuality, the stored exponent values 0 and 255 are used to indicate special values, and the exponential range of this format is thus restricted to

$$2^{-126} \text{ to } 2^{127}$$

The number 0.0 is defined by a mantissa of 0 together with the special exponential value 0. The IEEE standard also allows the values $\pm\infty$, very small denormalized numbers, and various other special conditions. Overall, the standard allows approximately seven significant decimal digits and an approximate value range of 10^{-45} to 10^{38}.

The double-precision floating point format standard works similarly. Sixty-four bits (8 bytes) are divided into a sign, 11 bits of exponent, and 52 bits of mantissa. The same format is used, with excess-1023 notation for the exponent, an implied base of 2, and an implied most significant bit to the left of the implied binary point. The double-precision standard supports approximately fifteen significant decimal digits and a range of more than 10^{-300} to 10^{300}!

The values defined for all possible 32-bit words are shown in Figure 5.18. The 64-bit table is similar, except for the limiting exponent of 2047, which results in an excess 1023 offset.

Conversion between Base 10 and Base 2

On occasion, you may find it useful to be able to convert real numbers between decimal and binary representation. This task must be done carefully. There are two major areas that sometimes cause students (and others!) difficulty:

1. The whole and fractional parts of numbers with an embedded decimal or binary point must be converted separately.
2. Numbers in exponential form must be reduced to a pure decimal or binary mixed number or fraction before the conversion can be performed.

We dealt with the first issue in Section 3.8. Recall from that section that in converting from one base to another that one must deal with the different multipliers associated with each successive digit. To the left of the radix point, the multipliers are integer, and there is a direct relationship between the different bases. To the right of the point, the multipliers are fractional, and there may or may not be a rational relationship between the multipliers in the different bases.

The solution is to convert each side of the radix point separately using the techniques discussed in Chapter 3. As an alternative, you can multiply the entire number in one base by whatever number is required to make the entire number an integer, and then convert the number in integer form. When this is complete, however, you must divide the converted result by *that same multiplier* in the new base. It is not correct to simply shift the radix point back, since each shift has a different value in the new base! Thus, if you shift a binary point

right by seven places, you have effectively multiplied the number by 128, and you must divide the converted number by 128 in the new base. This latter method is best illustrated with an example.

EXAMPLE

Convert the decimal number 253.75 to binary floating point form.

Begin by multiplying the number by 100 to form the integer value 25375. This is converted to its binary equivalent 110001100011111, or $1.10001100011111 \times 2^{14}$. The IEEE 754 floating point equivalent representation for this integer would be

One more step is required to complete the conversion. The result must be divided by the binary floating point equivalent of 100_{10} to restore the original decimal value. 100_{10} converts to binary 1100100_2, or 010000101100100 in IEEE 754 form. The last step is to divide the original result by this value, using floating point division. We will omit this step, as it is both difficult and irrelevant to this discussion. Although this method looks more difficult than converting the number directly as a mixed fraction, it is sometimes easier to implement within the computer.

The problem with converting floating point numbers expressed in exponential notation is essentially the same problem; however, the difficulty is more serious because it looks as though it should be possible to convert a number, keeping the same exponent, and this is of course not true.

If you always remember that the exponent actually represents a multiplier of value B^e, where B is the base and e is the actual exponent, then you will be less tempted to make this mistake. Obviously it is incorrect to assume that this multiplier would have the same value for a different B.

Instead, it is necessary to follow one of the two solutions just outlined: either reduce the exponential notation to a standard mixed fraction and convert each side separately, or use the value B^e as a multiplier to be divided in the new base at the end of the conversion.

5.4 PROGRAMMING CONSIDERATIONS

In this chapter you have been exposed to a number of different ways of storing and manipulating numeric values. It should be of interest to you to consider how a programmer might make an intelligent choice between the many different options available.

The trade-offs between integer and floating point are clear. Integer calculations are easier for the computer to perform, have the potential to provide higher precision, and are obviously much faster to execute. Integer values usually take up fewer storage locations. As you will see later, it takes a certain amount of time to access each storage location; thus, the use of fewer storage locations saves time, as well as space.

Clearly, the use of integer arithmetic is preferred whenever possible. Most modern high-level languages provide two or more different integer word sizes, usually at least a

"short" integer of 16 bits and a "long" integer of 64 bits. Now that you understand the range limitations of integer arithmetic, you are in a position to determine whether a particular variable or constant can use the integer format, and whether special error checking may be required in your program.

The longer integer formats may require multiple-word calculation algorithms, and as such are slower to execute than short formats. The short format is preferable when it is sufficient for the values that you expect. It may also be necessary to consider the limitations of other systems that the same program may have to operate on.

The use of real numbers is indicated whenever the variable or constant has a fractional part, whenever the number can take on very large or very small values that are outside of integer range, or whenever the required precision exceeds the number of different values that are possible in the longest integer format available to you. (As you've seen, most systems provide a floating point format of very high precision.) Of course, it is sometimes possible to multiply a mixed number by some multiplier to make it integer, perform the calculations in integer form, and then divide back. If the number of calculations is large, and the numbers can be adjusted to operate as integers, this can be a worthwhile option to consider, especially for the gain in execution speed.

As with integers, it is desirable to use the real number with the least precision that is sufficient for the task. Higher-precision formats require more storage and usually must use multiple-word floating point or packed decimal calculation algorithms that are much slower than the lower-precision formats.

Recall that decimal fractions may convert into irrational binary fractions. For those languages that provide the capability, the use of packed decimals represents an attractive alternative to floating point for those business applications where exact calculations involving mixed decimal numbers are required.

SUMMARY AND REVIEW

Computers store all data as binary numbers. There are a number of different ways to format these binary numbers to represent the various types of numbers required for computer processing. Conceptually, the simplest formats are sign-and-magnitude and binary-coded decimal. Although BCD is sometimes used for business programming, both of these formatting methods have shortcomings in terms of number manipulation and calculation.

Unsigned integers can of course be directly represented by their binary equivalents. Complementary arithmetic is usually the method of choice for signed integers. Nine's decimal complement, and its binary equivalent 1's complement, split the number range in two, using the upper half of the range to represent negative numbers. Positive numbers represent themselves. These representations are convenient and especially simple to use, since the complement is found by subtracting the number from a row of the largest digits in the base. Binary complements may be found by simply inverting the 0s and 1s in the number. Calculations are a bit more difficult due to the existence of both positive and negative values for zero, but end-around carry addition may be used for this purpose.

Ten's complement and 2's complement split the range similarly, but use a single value 0 for zero. This requires the use of a complement based on a value one larger than the largest

number in the base for the given number of digits. This ''base value'' will always consist of a 1 followed by *N* zeros, where *N* is the number of digits being used. Complementation may be taken by inverting the number as before, and adding 1 to the result, or by subtracting the number from the base value. Calculation is straightforward, using modulo arithmetic. Most computer arithmetic instructions are based on 2's complement arithmetic.

Both 1's and 2's complement representations have the additional convenience that the sign of a number may be readily identified, since a negative number always begins with a ''1.'' Small negative numbers have large values, and vice versa. Complementary representations for other even-numbered bases can be built similarly.

Numbers with a fractional part and numbers that are too large to fit within the constraints of the integer data capacity are stored and manipulated in the computer as real, or floating point, numbers. In effect, there is a trade-off between accuracy and range of acceptable numbers.

The usual floating point number format consists of a sign bit, an exponent, and a mantissa. The sign and value of the exponent are usually represented in an excess-*N* format. The base of the exponent is 2 for most systems, but some systems use a different base for the exponent. The radix point is implied. When possible, the mantissa is normalized.

In some systems the leading bit is also implied, since normalization requires that the leading bit of the mantissa be a 1.

Floating point numbers are subject to overflow or underflow, where the exponent of the number is too large or too small to represent, respectively. Zero is treated as a special case. Sometimes there is also a special representation for ∞.

Addition and subtraction require that the exponents in each number be equal. This is equivalent to lining up the decimal point in conventional decimal arithmetic. In multiplication and division, the exponents are added or subtracted, respectively. Special care must be taken with exponents that are expressed in excess-*N* notation.

Most computers conform to the format defined in IEEE Standard 754. Other formats in use include extra-precision formats and legacy formats.

FOR FURTHER READING

The representation and manipulation of integers and real numbers within the computer is discussed in most computer architecture texts. A particularly effective discussion is found in Stallings [STAL05]. This discussion presents detailed algorithms and hardware implementations for the various integer operations. A simpler discussion, with many examples, is found in Lipschutz [LIPS82]. More comprehensive treatments of computer arithmetic can be found in the two-volume collection of papers edited by Swartzlander [SWAR90] and in various textbooks on the subject, including those by Kulisch and Maranker [KUL81] and Spaniol [SPAN81]. A classical reference on computer algorithms, which includes a substantial discussion on computer arithmetic, is the book by Knuth [KNUT97]. One additional article of interest is the article titled ''What Every Computer Scientist Should Know About Floating-Point Arithmetic'' [GOLD91].

KEY CONCEPTS AND TERMS

binary-coded decimal (BCD)
carry flag
complement
end-around carry
excess-*N* notation
exponent
exponential notation
floating point format
floating point numbers
integer numbers
integer representation
inversion
mantissa
normalization
1's complement representation
overflow
radix point
real numbers
sign-and-magnitude representation
signed integers
single-precision floating point format
2's complement representation
underflow
unsigned integer
wraparound

READING REVIEW QUESTIONS

5.1 What is the largest unsigned integer that can be stored as a 16-bit number?

5.2 What does *BCD* stand for? Explain at least two important disadvantages of storing numbers in BCD format. Offer one advantage for using a BCD format for storing numbers.

5.3 Give an example that shows the disadvantage of using a sign-and-magnitude format for manipulating signed integers.

5.4 What is a quick way to identify negative numbers when using 1's complement arithmetic?

5.5 How do you change the sign of an integer stored in 1's complement form? As an example, the 8-bit representation for the value 19 is 00010011_2. What is the 1's complement representation for −19?

5.6 How do you identify an *overflow* condition when you add two numbers in 1's complement form?

5.7 Explain the procedure for adding two numbers in 1's complement form. As an example, convert +38 and −24 to 8-bit 1's complement form and add them. Convert your result back to decimal and confirm that your answer is correct.

5.8 If you see a 2's complement number whose value is 11111110_2, what rough estimate can you make about the number?

5.9 How do you change the sign of an integer stored in 2's complement form? As an example, the 8-bit representation for the value 19 is 00010011_2. What is the 2's complement representation for −19?

5.10 How do you detect overflow when adding two numbers in 2's complement form?

5.11 Explain the procedure for adding two numbers in 2's complement form. As an example, convert +38 and −24 to 8-bit 2's complement form and add them. Convert your result back to decimal and confirm that your answer is correct.

5.12 What is the relationship between complementary representation and sign-and-magnitude representation for positive numbers?

5.13 Real numbers in a computer (or *float*, if you prefer), are most often represented in exponential notation. Four separate components are needed to represent numbers in this form. Identify each component in the number 1.2345×10^{-5}. What is the advantage of this type of representation, rather than storing the number as 0.000012345?

5.14 To represent a number in exponential form in the computer, two additional assumptions must be made about the number. What are those assumptions?

5.15 Exponents are normally stored in *excess-N* notation. Explain excess-*N* notation. If a number is stored in excess-31 notation and the actual exponent is 2^{+12}, what value is stored in the computer for the exponent?

5.16 When adding two floating point numbers, what must be true for the exponents of the two numbers?

5.17 The IEEE provides a standard 32-bit format for floating point numbers. The format for a number is specified as $\pm 1.M \times 2^{E-127}$. Explain each part of this format.

EXERCISES

5.1 Data was stored in the Digital PDP-9 computer using six-digit octal notation. Negative numbers were stored in 8's complement form.

- **a.** How many bits does six-digit octal represent? Show that 8's complement octal and 2's complement binary are exactly equivalent.
- **b.** What is the largest positive octal number that can be stored in this machine?
- **c.** What does the number in (b) correspond to in decimal?
- **d.** What is the largest possible negative number? Give your answer in both octal and decimal form.

5.2

- **a.** Find the 16-bit 2's complementary binary representation for the decimal number 1987.
- **b.** Find the 16-bit 2's complementary binary representation for the decimal number −1987.
- **c.** From your answer in (b) find the six-digit 16's complement hexadecimal representation for the decimal number −1987.

5.3 Data is stored in the R4-D4 computer using eight-digit base 4 notation. Negative numbers are stored using 4's complement.

- **a.** What is the sign-and-magnitude value of the following 4's complement number?

$$33333210_4$$

  Leave your answer in base 4.

- **b.** Add the following eight-digit 4's complement numbers. Then, show the sign-and-magnitude values (in base 4) for each of the input numbers and for your result.

$$\begin{array}{r} 13220231 \\ \underline{\quad 120000} \end{array}$$

5.4 Convert the decimal number −19575 to a 15-bit 2's complement binary number. What happens when you perform this conversion? After the conversion is complete, what values (base 2 and base 10) does the computer think it has?

5.5 What are the 16-bit 1's and 2's complements of the following binary numbers?

a. 10000

b. 100111100001001

c. 0100111000100100

5.6 Add the following decimal numbers by converting each to five-digit 10's complementary form, adding, and converting back to sign and magnitude.

a.

```
 24379
  5098
```

b.

```
 24379
−5098
```

c.

```
−24379
  5098
```

5.7 Subtract the second number from the first by taking the six-digit 10's complement of the second number and adding. Convert the result back to sign and magnitude if necessary.

a.

```
  37968
(−) 24109
```

b.

```
  37968
(−) −70925
```

c.

```
 −10255
(−) −7586
```

5.8 The following decimal numbers are already in six-digit 10's complementary form. Add the numbers. Convert each number and your result to sign and magnitude, and confirm your results.

a.

```
  1250
772950
```

b.

```
899211
999998
```

c.

```
970000
 30000
```

5.9 Add the following two 12-bit binary 2's complement numbers. Then convert each number to decimal and check the results.

a.

```
 11001101101
111010111011
```

b.

```
101011001100
111111111100
```

5.10 Given the positive number 2468, what is the largest positive digit that you can add that will not cause overflow in a four-digit decimal, 10's complement number system?

5.11 In 12's complement base 12, how would you know if a number is positive or negative?

5.12 Most computers provide separate instructions for performing unsigned additions and complementary additions. Show that for unsigned additions, carry and overflow are the same. (Hint: Consider the definition of overflow.)

5.13 Consider a machine that performs calculations 4 bits at a time. Eight-bit 2's complement numbers can be added by adding the four least significant bits, followed by the four most significant bits. The leftmost bit is used for the sign, as usual. With 8 bits for each number, add -4 and -6, using 4-bit binary 2's complement arithmetic. Did overflow occur? Did carry occur? Verify your numerical result.

5.14 Add the following 16's complement hexadecimal numbers

```
4F09
D3A5
```

Is your result positive or negative? How do you know? Convert each number to binary and add the binary numbers. Convert the result back to hexadecimal. Is the result the same?

5.15 In the Pink-Lemon-8 computer, real numbers are stored in the format

$$SEEMMMM_8$$

where all the digits, including the exponent, are in octal. The exponent is stored excess-40_8. The mantissa is stored as sign and magnitude, where the sign is 0 for a positive number and 4 for a negative number. The implied octal point is at the end of the mantissa: MMMM.
Consider the real number stored in this format as

```
4366621
```

a. What real number is being represented? Leave your answer in octal.

b. Convert your answer in part (a) to decimal. You may leave your answer in fractional form if you wish.

c. What does changing the original exponent from 36 to 37 do to the magnitude of the number? (Stating that it moves the octal point one place to the right or left is not a sufficient answer.) What would be the new magnitude in decimal?

5.16 **a.** Convert the decimal number 19557 to floating point. Use the format SEEMMMM. All digits are decimal. The exponent is stored excess-40 (not excess-50). The implied decimal point is at the *beginning* of the mantissa. The sign is 1 for a positive number, 7 for a negative number. Hint: Note carefully the number of digits in the mantissa!

b. What is the range of numbers that can be stored in this format?

c. What is the floating point representation for −19557?

d. What is the six-digit 10's complement representation for −19557?

e. What is the floating point representation for 0.0000019557?

5.17 **a.** Convert the number 123.57×10^{15} to the format SEEMMMM, with the exponent stored excess-49. The implied decimal point is to the right of the first mantissa digit.

b. What is the smallest number you can use with this format before underflow occurs?

5.18 Real numbers in the R4-D4 computer are stored in the format

$$\text{SEEMMMM}_4$$

where all the digits, including the exponent, are in base 4. The mantissa is stored as sign and magnitude, where the sign is 0 for a positive number and 3 for a negative number. The implied quadrinary (base 4!) point is at the beginning of the mantissa:

.MMMM

a. If you know that the exponent is stored in an excess-something format, what would be a good choice of value for "something?"

b. Convert the real, decimal number 16.5 to base 4, and show its representation in the format of the R4-D4 computer. Use the excess value that you determined in part (a).

5.19 Convert the following binary and hexadecimal numbers to floating point format. Assume a binary format consisting of a sign bit (negative = 1), a base 2, 8-bit, excess-128 exponent, and 23 bits of mantissa, with the implied binary point to the right of the first bit of the mantissa.

a. 110110.011011_2

b. -1.1111001_2

c. $-4F7F_{16}$

d. 0.00000000111111_2

e. 0.1100×2^{36}

f. 0.1100×2^{-36}

5.20 For the format used in Exercise 5.5, what decimal number is represented by each of the following numbers in floating point format?

a. $C2F00000_{16}$

b. $3C540000_{16}$

5.21 Represent the decimal number 171.625 in IEEE 754 format.

5.22 Show the packed decimal format for the decimal number −129975.

5.23 The following decimal numbers are stored in excess-50 floating point format, with the decimal point to the left of the first mantissa digit. Add them. A 9 is used as a negative sign. Present your result in standard decimal sign-and-magnitude notation.

a.

```
05225731
04833300
```

b.

```
05012500
95325750
```

5.24 Using the same notation as in Exercise 5.9, multiply the following numbers. Present your answer in standard decimal notation.

a.

```
05452500
04822200
```

b.

```
94650000
94450000
```

5.25 Using the same format found in Exercise 5.19, add and multiply the following floating point numbers. Present your answers in both floating point and sign-and-magnitude formats.

$$\begin{array}{l} 3\text{DEC}0000_{16} \\ \underline{\text{C24C}0000_{16}} \end{array}$$

5.26 Write a program in your favorite language that converts numbers represented in the decimal floating point format

```
SEEMMMMM
```

into 10's complementary integer form. Round any fractional decimal value.

5.27 What base is the student in the chapter cartoon using to perform his addition?