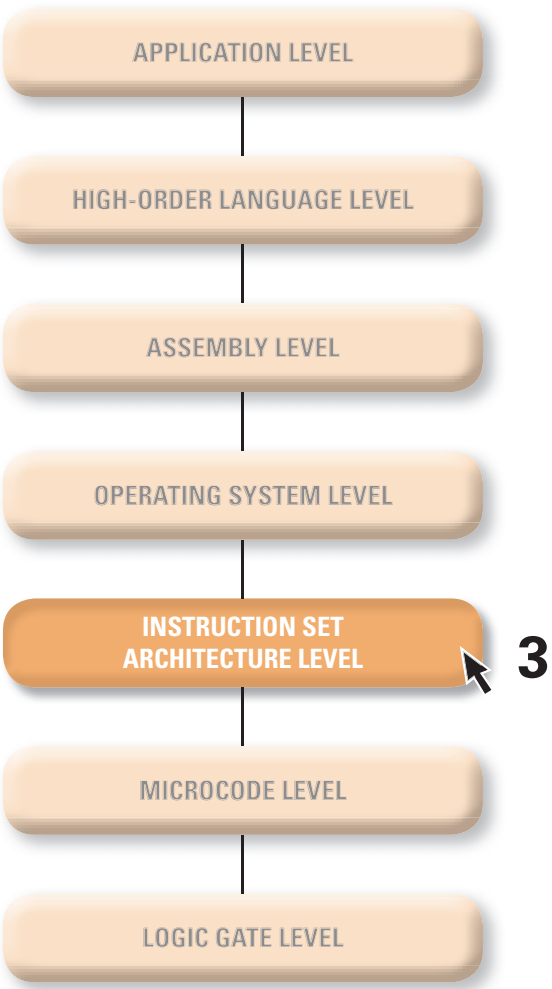
LEVEL
3

Instruction Set Architecture



CHAPTER

3

Information Representation

TABLE OF CONTENTS

One of the most significant inventions of mankind is the printed word. The words on this page represent information stored on paper, which is conveyed to you as you read. Like the printed page, computers have memories for storing information. The central processing unit (CPU) has the ability to retrieve information from its memory much as you take information from words on a page.

Reading and writing, words and pages

Some computer terminology is based on this analogy. The CPU *reads* information from memory and *writes* information into memory. The information itself is divided into *words*. In some computer systems, large sets of words, usually anywhere from a few hundred to a few thousand, are grouped into *pages*.

Information representation at Level ISA3

In C, at Level HOL6, information takes the form of values that you store in a variable in main memory or in a file on disk. This chapter shows how the computer stores that information at Level ISA3. Information representation at the machine level differs significantly from that at the high-order languages level. At Level ISA3, information representation is less human-oriented. Later chapters discuss information representation at the intermediate levels, Levels Asmb5 and OS4, and show how they relate to Levels HOL6 and ISA3.

3.1 Unsigned Binary Representation

The Mark I computer

Early computers were electromechanical. That is, all their calculations were performed with moving switches called *relays*. The Mark I computer, built in 1944 by Howard H. Aiken of Harvard University, was such a machine. Aiken had procured financial backing for his project from Thomas J. Watson, president of International Business Machines (IBM). The relays in the Mark I computer could compute much faster than the mechanical gears that were used in adding machines at that time.

The ENIAC computer

Even before the completion of Mark I, John V. Atanasoff, working at Iowa State University, had finished the construction of an electronic computer to solve systems of linear equations. In 1941 John W. Mauchly visited Atanasoff's laboratory and in 1946, in collaboration with J. Presper Eckert at the University of Pennsylvania, built the famous Electronic Numerical Integrator and Calculator (ENIAC). ENIAC's 19,000 vacuum tubes could perform 5000 additions per second compared to 10 additions per second with the relays of the Mark I. Like the ENIAC, present-day computers are electronic, although their calculations are performed with integrated circuits (ICs) instead of with vacuum tubes.

Binary Storage

Electronic computer memories cannot store numbers and letters directly. They can store only electrical signals. When the CPU reads information

from memory, it is detecting a signal whose voltage is about equal to that produced by two flashlight batteries.

Computer memories are designed with a most remarkable property. Each storage location contains either a high-voltage signal or a low-voltage signal—never anything in between. The storage location is like being pregnant. Either you are or you are not. There is no halfway.

The word *digital* means that the signal stored in memory can have only a fixed number of values. *Binary* means that only two values are possible. Practically all computers on the market today are binary. Hence, each storage location contains either a high voltage or a low voltage. The state of each location is also described as being either on or off, or, alternatively, as containing either a 1 or a 0.

Each individual storage unit is called a *binary digit* or *bit*. A bit can be only 1 or 0—never anything else, such as 2, 3, A, or Z. This is a fundamental concept. Every piece of information stored in the memory of a computer, whether it is the amount you owe on your credit card or your street address, is stored in binary as 1's and 0's.

In practice, the bits in a computer system are grouped together into *cells*. A seven-bit cell, for example, would store its information in groups of seven bits, as FIGURE 3.1(a) shows. You can think of a cell as a group of boxes, each box containing a 1 or a 0, and nothing else. Part (b) shows some possible binary values in a seven-bit cell. The values in part (c) are impossible because the digits in some boxes differ from 0 or 1.

Different parts of a computer system have different numbers of bits in each cell. Practically all computers today have eight bits per cell in their main memories and disks. An eight-bit cell is a *byte*. Other parts of the system have different sizes. This chapter shows examples with several different cell sizes to illustrate the general principle.

Information such as numbers and letters must be represented in binary form to be stored in memory. The representation scheme used to store information is called a *code*. This section examines a code for storing unsigned integers. The remainder of this chapter describes codes for storing other kinds of data. The next chapter examines codes for storing program commands in memory.

FIGURE 3.1
A seven-bit memory cell in main memory.

(a) A seven-bit cell.

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

(b) Some possible values in a seven-bit cell.

6	8	0	7	2	5	1
J	A	N	U	A	R	Y

(c) Some impossible values in a seven-bit cell.

Integers

Numbers must be represented in binary form to be stored in a computer's memory. The particular code depends on whether the number has a fractional part or is an integer. If the number is an integer, the code depends on whether it is always nonnegative or whether it can be negative as well.

Unsigned binary

The *unsigned binary* representation is for integers that are always nonnegative. Before learning the binary system, we will review our own

base 10 (*decimal*, or *dec* for short) system, and then work our way down to the binary system.

Our decimal system was probably invented because we have 10 fingers with which we count and add. A book of arithmetic using this elegant system was written in India in the eighth century AD. It was translated into Arabic and was eventually carried by merchants to Europe, where it was translated from Arabic into Latin. The numbers came to be known as Arabic numerals because at the time it was thought that they originated in Arabia. But Hindu-Arabic numerals would be a more appropriate name because they actually originated in India.

Counting with Arabic numerals in base 10 looks like this (reading down, of course):

Counting in decimal

0	7	14	21	28	35
1	8	15	22	29	36
2	9	16	23	30	37
3	10	17	24	31	38
4	11	18	25	32	⋮
5	12	19	26	33	
6	13	20	27	34	

Starting from 0, the Indians simply invented a symbol for the next number, 1, then 2, and so on until they got to the symbol 9. At that point they looked at their hands and thought of a fantastic idea. On their last finger they did not invent a new symbol. Instead they used the first two symbols, 1 and 0, together to represent the next number, 10.

You know the rest of the story. When they got to 19 they saw that the 9 was as high as they could go with the symbols they had invented. So they dropped it down to 0 and increased the 1 to 2, creating 20. They did the same for 29 to 30 and, eventually, 99 to 100. On and on it went.

What if we only had 8 fingers instead of 10? What would have happened? At 7, the next number would be on our last finger, and we would not need to invent a new symbol. The next number would be represented as 10. Counting in base 8 (*octal*, or *oct* for short) looks like this:

Counting in octal

0	7	16	25	34	43
1	10	17	26	35	44
2	11	20	27	36	45
3	12	21	30	37	46
4	13	22	31	40	⋮
5	14	23	32	41	
6	15	24	33	42	

The next number after 77 is 100 in octal.

Comparing the decimal and octal schemes, notice that 5 (oct) is the same number as 5 (dec), but that 21 (oct) is not the same number as 21 (dec). Instead, 21 (oct) is the same number as 17 (dec). Numbers have a tendency to look larger than they actually are when written in octal.

But what if we only had 3 fingers instead of 10 or 8? The pattern is the same. Counting in base 3 looks like this:

Counting in base 3

0	21	112	210	1001	1022
1	22	120	211	1002	1100
2	100	121	212	1010	1101
10	101	122	220	1011	1102
11	102	200	221	1012	⋮
12	110	201	222	1020	
20	111	202	1000	1021	

Finally, we have arrived at unsigned binary representation. Computers have only two fingers. Counting in base 2 (*binary*, or *bin* for short) follows the exact same method as counting in octal and base 3:

Counting in binary

0	111	1110	10101	11100	100011
1	1000	1111	10110	11101	100100
10	1001	10000	10111	11110	100101
11	1010	10001	11000	11111	100110
100	1011	10010	11001	100000	⋮
101	1100	10011	11010	100001	
110	1101	10100	11011	100010	

Binary numbers look a lot larger than they actually are. The number 10110 (bin) is only 22 (dec).

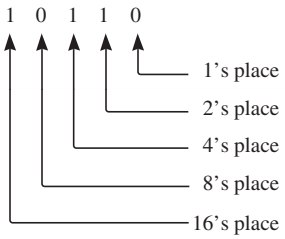
Base Conversions

Given a number written in binary, there are several ways to determine its decimal equivalent. One way is to simply count up to the number in binary and in decimal. That method works well for small numbers. Another method is to add up the place values of each 1 bit in the binary number.

Example 3.1 FIGURE 3.2(a) shows the place values for 10110 (bin). Starting with the 1's place on the right (called the *least significant bit*), each place has a value twice as great as the previous place value. Figure 3.2(b) shows the addition that produces the 22 (dec) value. ∎

Example 3.2 The unsigned binary number system is analogous to our familiar decimal system. FIGURE 3.3 shows the place values for 58,036 (dec). The figure 58,036 represents six 1's, three 10's, no 100's, eight 1000's, and

FIGURE 3.2
Converting from binary to decimal.

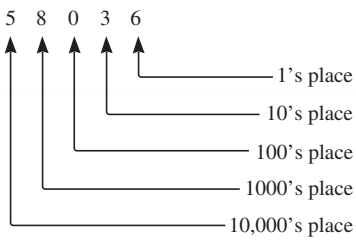


(a) The place values for 10110 (bin).

0 1's place = 0
1 2's place = 2
1 4's place = 4
0 8's place = 0
1 16's place = 16
22 (dec)

(b) Converting 10110 (bin) to decimal.

FIGURE 3.3
The place values for 58,036 (dec).



five 10,000's. Starting with the 1's place from the right, each place value is 10 times greater than the previous place value. In binary, each place value is 2 times greater than the previous place value.

The value of an unsigned number can be conveniently represented as a polynomial in the base of the number system. (The base is also called the *radix* of the number system.) FIGURE 3.4 shows the polynomial representation of 10110 (bin) and 58,036 (dec). The value of the least significant place is

FIGURE 3.4
The polynomial representation of unsigned numbers.

$$1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

(a) The binary number 10110.

$$5 \times 10^4 + 8 \times 10^3 + 0 \times 10^2 + 3 \times 10^1 + 6 \times 10^0$$

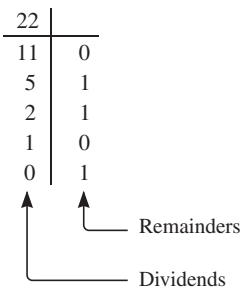
(b) The decimal number 58,036.

always the base to the zeroth power, which is always 1. The next significant place is the base to the first power, which is the value of the base itself. You can see from the structure of the polynomial that the value of each place is the base times the value of the previous place.

In binary, the only place with an odd value is the 1's place. All the other places (2's, 4's, 8's, and so on) are even. If there is a 0 in the 1's place, the value of the binary number will come from adding several even numbers, and it therefore will be even. On the other hand, if there is a 1 in the 1's place of a binary number, its value will come from adding one to several even numbers, and it will be odd. As in the decimal system, you can tell whether a binary number is even or odd simply by inspecting the digit in the 1's place.

Determining the binary equivalent of a number written in decimal is a bit tricky. One method is to successively divide the original number by 2, keeping track of the remainders, which will form the binary number when listed in reverse order from how they were obtained.

FIGURE 3.5
Converting from decimal to binary.



Example 3.3 FIGURE 3.5 converts 22 (dec) to binary. The number 22 divided by 2 is 11 with a remainder of 0, which is written in the right column. Then, 11 divided by 2 is 5, with a remainder of 1. Continuing until the number gets down to 0 produces a column of remainders, which, when read from the bottom up, form the binary number 10110. ∎

Notice that the least significant bit is the remainder when you divide the original value by 2. This fact is consistent with the observation that you can determine whether a binary number is even or odd by inspecting only the least significant bit. If the original value is even, the division will produce a remainder of 0, which will be the least significant bit. Conversely, if the original value is odd, the least significant bit will be 1.

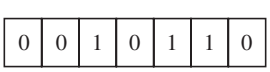
Range for Unsigned Integers

All these counting schemes based on Arabic numerals let you represent arbitrarily large numbers. A real computer, however, has a finite number of bits in each cell. FIGURE 3.6 shows how a seven-bit cell would store the number 22 (dec). Notice the two leading 0's, which do not affect the value of the number, but which are necessary for specifying the contents of the memory location. In dealing with a seven-bit cell, you should write the number without showing the boxes as

001 0110

The two leading 0's are still necessary. This text displays bit strings with a space (for legibility) between each group of four bits starting from the right.

FIGURE 3.6
The number 22 (dec) in a seven-bit cell.



The range of unsigned values depends on the number of bits in a cell. A sequence of all 0's represents the smallest unsigned value, and a sequence of all 1's represents the largest.

Example 3.4 The smallest unsigned integer a seven-bit cell can store is

000 0000 (bin)

and the largest is

111 1111 (bin)

The smallest is 0 (dec) and the largest is 127 (dec). A seven-bit cell cannot store an unsigned integer greater than 127. ∎

Unsigned Addition

Addition with unsigned binary numbers works like addition with unsigned decimal numbers. But it is easier because you only need to learn the addition rules for 2 bits instead of 10 digits. The rules for adding bits are

Binary addition rules

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 10$$

The carry technique in binary

The carry technique that you are familiar with in the decimal system also works in the binary system. If two numbers in a column add to a value greater than 1, you must carry 1 to the next column.

Example 3.5 Suppose you have a six-bit cell. To add the two numbers 01 1010 and 01 0001, simply write one number above the other and start at the least significant column:

```
      01 1010
ADD   01 0001
      -------
      10 1011
```

Notice that when you get to the fifth column from the right, 1 + 1 equals 10. You must write down the 0 and carry the 1 to the next column, where 1 + 0 + 0 produces the leftmost 1 in the sum.

To verify that this carry technique works in binary, convert the two numbers and their sum to decimal:

01 1010 (bin) = 26 (dec)
01 0001 (bin) = 17 (dec)
10 1011 (bin) = 43 (dec)

Sure enough, 26 + 17 = 43 in decimal. ∎

Example 3.6 These examples show how the carry can propagate along several consecutive columns:

```
      00 0011          00 1111
ADD   01 0001    ADD   00 1001
      01 0100          01 1000
```

In the second example, when you get to the fourth column from the right, you have a carry from the previous column. Then 1 + 1 + 1 equals 11. You must write down 1 and carry 1 to the next column. ∎

The Carry Bit

The range for the six-bit cell of the previous examples is 00 0000 to 11 1111 (bin), or 0 to 63 (dec). It is possible for two numbers to be in range but for their sum to be out of range. In that case, the sum is too large to fit into the six bits of the storage cell.

The carry bit in addition

To flag this condition, the CPU contains a special bit called the *carry bit*, denoted by the letter C. When two binary numbers are added, if the sum of the leftmost column (called the *most significant bit*) produces a carry, then C is set to 1. Otherwise C is cleared to 0. In other words, C always contains the carry from the leftmost column of the cell. In all the previous examples, the sum was in range. Hence the carry bit was cleared to 0.

Example 3.7 Here are two examples showing the effect on the carry bit:

```
      01 0110          10 1010
ADD   10 0010    ADD   01 1010
C = 0 11 1000    C = 1 00 0100
```

In the second example, the CPU adds 42 + 26. The correct result, which is 68, is too large to fit into the six-bit cell. Remember that the range is from 0 to 63. So the lowest order (that is, the rightmost) six bits are stored, giving an incorrect result of 4. The carry bit is also set to 1 to indicate that a carry occurred from the highest-order column. ∎

3.2 Two's Complement Binary Representation

The unsigned binary representation works for nonnegative integers only. If a computer is to process negative integers, it must use a different representation.

Suppose you have a six-bit cell and you want to store the number −5 (dec). Because 5 (dec) is 101 (bin), you might try the pattern shown in FIGURE 3.7. But this is impossible because all bits, including the first, must

FIGURE 3.7
An attempt to store a negative number in binary.

−	0	0	1	0	1

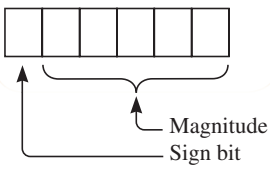


FIGURE 3.8
The structure of a signed integer.

be 0 or 1. Remember that computers are binary. The above storage value would require each box to be capable of storing a 0, or a 1, or a dash. Such a computer would have to be ternary instead of binary.

The solution to this problem is to reserve the first box in the cell to indicate the sign. Thus, the six-bit cell will have two parts—a one-bit sign and a five-bit magnitude, as FIGURE 3.8 shows. Because the sign bit must be 0 or 1, one possibility is to let a 0 sign bit indicate a positive number and a 1 sign bit indicate a negative number. Then +5 could be represented as

00 0101

and −5 could be represented as

10 0101

In this code the magnitudes for +5 and −5 would be identical. Only the sign bits would differ.

Few computers use the previous code, however. The problem is that if you add +5 and −5 in decimal, you get 0, but if you add 00 0101 and 10 0101 in binary (sign bits and all), you get

```
      00 0101
ADD   10 0101
------------
C = 0 10 1010
```

A convenient property of negative numbers

which is definitely not 0. It would be much more convenient if the hardware of the CPU could add the numbers for +5 and −5, complete with sign bits using the ordinary rules for unsigned binary addition, and get 0.

The *two's complement* binary representation has that property. The positive numbers have a 0 sign bit and a magnitude as in the unsigned binary representation. For example, the number +5 (dec) is still represented as 00 0101.

But the representation of −5 (dec) is not 10 0101. Instead, it is 11 1011, because adding +5 and −5 gives

```
      00 0101
ADD   11 1011
------------
C = 1 00 0000
```

Note that the six-bit sum is all 0's, as advertised.

The NEG operation

Under the rules of binary addition for a six-bit cell, the number 11 1011 is called the *additive inverse* of 00 0101. The operation of finding the additive inverse is referred to as *negation*, abbreviated NEG. To negate a number is also called *taking its two's complement*.

All we need now is the rule for taking the two's complement of a number. A simple rule is based on the *ones' complement*, which is simply the binary

sequence with all the 1's changed to 0's and all the 0's changed to 1's. The ones' complement is also called the NOT operation.

The NOT operation

Example 3.8 The ones' complement of 00 0101 is

NOT 00 0101 = 11 1010

assuming a six-bit cell. ∎

A clue to finding the rule for two's complement is to note the effect of adding a number to its ones' complement. Because 1 plus 0 is 1, and 0 plus 1 is 1, any number, when added to its ones' complement, will produce a sequence of all 1's. But then, adding a single 1 to a number of all 1's produces a number of all 0's.

Example 3.9 Adding 00 0101 to its ones' complement produces

```
        00 0101
ADD     11 1010
C = 0   11 1111
```

which is all 1's. Adding 1 to this produces

```
        11 1111
ADD     00 0001
C = 1   00 0000
```

which is all 0's. ∎

In other words, adding a number to its ones' complement plus 1 gives all 0's. So the two's complement of a binary number must be found by adding 1 to its ones' complement.

Example 3.10 To find the two's complement of 00 0101, add 1 to its ones' complement.

NOT 00 0101 = 11 1010

```
        11 1010
ADD     00 0001
        11 1011
```

The two's complement of 00 0101 is therefore 11 1011. That is,

NEG 00 0101 = 11 1011

Recall that 11 1011 is indeed the negative of 00 0101 because they add to 0 as shown. ∎

The general rule for negating a number regardless of how many bits the number contains is

The two's complement rule

› The two's complement of a number is 1 plus its ones' complement.

Or, in terms of the NEG and NOT operations,

› NEG x = 1 + NOT x

In our familiar decimal system, if you take the negative of a value that is already negative, you get a positive value. Algebraically,

$$-(-x) = x$$

where x is some positive value. If the rule for taking the two's complement is to be useful, the two's complement of a negative value should be the corresponding positive value.

Example 3.11 What happens if you take the two's complement of −5 (dec)?

```
NOT   11 1011 = 00 0100

          00 0100
ADD       00 0001
          -------
          00 0101
```

Voilà! You get +5 (dec) back again, as you would expect. ∎

Two's Complement Range

Suppose you have a four-bit cell to store integers in two's complement representation. What is the range of integers for this cell?

The positive integer with the greatest magnitude is 0111 (bin), which is +7 (dec). It cannot be 1111 as in unsigned binary because the first bit is reserved for the sign and must be 0. In unsigned binary, you can store numbers as high as +15 (dec) with four bits. All four bits are used for the magnitude. In two's complement representation, you can store numbers only as high as +7 (dec), because only three bits are reserved for the magnitude.

What is the negative number with the greatest magnitude? The answer to this question might not be obvious. FIGURE 3.9 shows the result of taking the two's complement of each positive number up to +7. What pattern do you see in the figure?

Notice that the two's complement operation automatically produces a 1 in the sign bit of the negative numbers, as it should. Even numbers still end in 0, and odd numbers end in 1.

Also, −5 is obtained from −6 by adding 1 to −6 in binary, as you would expect. Similarly, −6 is obtained from −7 by adding 1 to −7 in binary. We can squeeze one more negative integer out of our four bits by including −8.

FIGURE 3.9
The result of taking the two's complement in a four-bit cell.

Decimal	Binary
−7	1001
−6	1010
−5	1011
−4	1100
−3	1101
−2	1110
−1	1111

When you add 1 to −8 in binary, you get −7. The number −8 should therefore be represented as 1000. FIGURE 3.10 shows the complete table for signed integers assuming a four-bit memory cell.

The number −8 (dec) has a peculiar property not shared by any of the other negative integers. If you take the two's complement of −7, you get +7, as follows:

```
NOT   1001 = 0110

      0110
ADD   0001
      0111
```

But if you take the two's complement of −8, you get −8 back again:

```
NOT   1000 = 0111

      0111
ADD   0001
      1000
```

This property exists because there is no way to represent +8 with only four bits.

We have determined the range of numbers for a four-bit cell with two's complement binary representation. It is

1000 to 0111

as written in binary, or

−8 to +7

as written in decimal.

The same patterns hold regardless of how many bits are contained in the cell. The largest positive integer is a single 0 followed by all 1's. The negative integer with the largest magnitude is a single 1 followed by all 0's. Its magnitude is 1 greater than the magnitude of the largest positive integer. The number −1 (dec) is represented as all 1's.

Example 3.12 The range for six-bit two's complement representation is

10 0000 to 01 1111

as written in binary, or

−32 to 31

as written in decimal. Unlike all the other negative integers, the two's complement of 10 0000 is itself, 10 0000. Also notice that −1 (dec) = 11 1111 (bin). ∎

FIGURE 3.10
The signed integers for a four-bit cell.

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

Base Conversions

Converting from decimal to binary

To convert a negative number from decimal to binary is a two-step process. First, convert its magnitude from decimal to binary as in unsigned binary representation. Then negate it by taking the two's complement.

Example 3.13 How is −7 (dec) stored in a 10-bit cell? First, find the binary value of +7.

+7 (dec) = 00 0000 0111 (bin)

Then, take its two's complement.

NOT 00 0000 0111 = 11 1111 1000

```
        11 1111 1000
ADD     00 0000 0001
        -------------
        11 1111 1001
```

So −7 (dec) is 11 1111 1001 (bin). ∎

Converting from binary to decimal

To convert a number from binary to decimal in a computer that uses two's complement representation, always check the sign bit first. If it is 0, the number is positive and you may convert, as in unsigned representation. If it is 1, the number is negative and you can choose one of two methods. One method is to make the number positive by negating it. Then convert to decimal as in unsigned representation.

Example 3.14 Say you have a 10-bit cell that contains 11 1101 1010. What decimal number does it represent? The sign bit is 1, so the number is negative. First negate the number:

NOT 11 1101 1010 = 00 0010 0101

```
        00 0010 0101
ADD     00 0000 0001
        -------------
        00 0010 0110
```

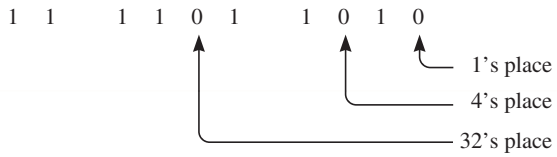
00 0010 0110 (bin) = 32 + 4 + 2 = 38 (dec)

So the original binary number must have been the negative of 38. That is,

11 1101 1010 (bin) = −38 (dec) ∎

The other method is to convert directly without taking the two's complement. Simply add 1 to the sum of the place values of the 0's in the original binary number. This method works because the first step in taking

FIGURE 3.11
The place values of the 0's in 11 1101 1010 (bin).



the two's complement of a positive integer is to invert the bits. Those bits that were 1's, and thus contributed to the magnitude of the positive integer, become 0's. The 0's, not the 1's, of a negative integer contribute to its magnitude.

Example 3.15 FIGURE 3.11 shows the place values of the 0's in 11 1101 1010 (bin). Adding 1 to their sum gives

$$11\ 1101\ 1010\ (\text{bin}) = -(1 + 32 + 4 + 1) = -38\ (\text{dec})$$

which is the same result as with the previous method. ∎

The Number Line

Another way of viewing binary representation is with the number line. FIGURE 3.12 shows the number line for a three-bit cell with unsigned binary representation. Eight numbers are represented.

You add by moving to the right on the number line. For example, to add 4 and 3, start with 4 and move three positions to the right to get 7. If you try to add 6 and 3 on the number line, you will fall off the right end. If you do it in binary, you will get an incorrect result because the answer is out of range:

```
      110
ADD   011
-------
C = 1 001
```

The two's complement number line comes from the unsigned number line by breaking it between 3 and 4 and shifting the right part to the left side.

FIGURE 3.12
The number line for a three-bit unsigned system.

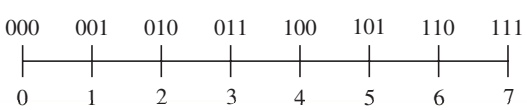
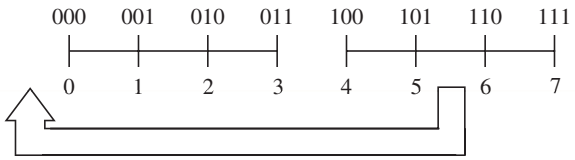
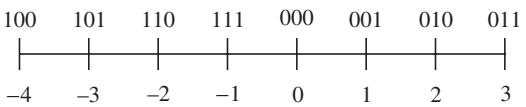


FIGURE 3.13
The number line for a three-bit two's complement system.



(a) Breaking the number line in the middle.



(b) Shifting the right part to the left side.

FIGURE 3.13 shows that the binary number 111 is now adjacent to 000, and what used to be +7 (dec) is now −1 (dec).

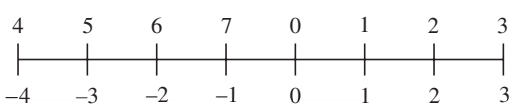
Addition is still performed by moving to the right on the number line, even if you pass through 0. To add −2 and 3, start with −2 and move three positions to the right to get 1. If you do it in binary, the answer is in range and correct:

	110
ADD	011
C = 1	001

These bits are identical to those for 6 + 3 in unsigned binary. Notice that the carry bit is 1, even though the answer is in range. With two's complement representation, the carry bit no longer indicates whether the result of the addition is in range.

Sometimes you can avoid the binary representation altogether by considering the shifted number line entirely in decimal. FIGURE 3.14 shows the two's complement number line with the binary number replaced by its

FIGURE 3.14
The two's complement number line with unsigned decimals.



unsigned decimal equivalent. In this example, there are three bits in each memory location. Thus, there are 2^3, or 8, possible numbers.

Now the unsigned and signed numbers are the same from 0 up to 3. Furthermore, you can get the signed negative numbers from the unsigned numbers by subtracting 8:

$$7 - 8 = -1$$
$$6 - 8 = -2$$
$$5 - 8 = -3$$
$$4 - 8 = -4$$

Example 3.16 Suppose you have an eight-bit cell. There are 2^8, or 256, possible integer values. The nonnegative numbers go from 0 to 127. Assuming two's complement binary representation, what do you get if you add 97 and 45? In unsigned binary, the sum is

$$97 + 45 = 142 \text{ (dec, unsigned)}$$

But in two's complement binary, the sum is

$$142 - 256 = -114 \text{ (dec, signed)}$$

Notice that we get this result by avoiding the binary representation altogether. To verify the result, first convert 97 and 45 to binary and add:

$$97 \text{ (dec)} = 0110\ 0001 \text{ (bin)}$$
$$45 \text{ (dec)} = 0010\ 1101 \text{ (bin)}$$

$$\begin{array}{ll} & 0110\ 0001 \\ \text{ADD} & 0010\ 1101 \\ \hline \text{C} = 0 & 1000\ 1110 \end{array}$$

This is a negative number because of the 1 in the sign bit. And now, to determine its magnitude:

$$\text{NEG } 1000\ 1110 = 0111\ 0010 \text{ (bin)}$$
$$= 114 \text{ (dec)}$$

This produces the expected result. ∎

The Overflow Bit

An important characteristic of binary storage at Level ISA3 is the absence of a type associated with a value. In the previous example, the sum 1000 1110, when interpreted as an unsigned number, is 142 (dec), but when interpreted in two's complement representation is −114 (dec). Although the value of the bit pattern depends on its type, whether unsigned or two's complement, the

hardware makes no distinction between the two types. It stores only the bit pattern.

When the CPU adds the contents of two memory cells, it uses the rules for binary addition on the bit sequences, regardless of their types. In unsigned binary, if the sum is out of range, the hardware simply stores the (incorrect) result, sets the C bit accordingly, and goes on. It is up to the software to examine the C bit after the addition to see if a carry out occurred from the most significant column and to take appropriate action if necessary.

The C bit detects overflow for unsigned integers.

We noted above that in two's complement binary representation, the carry bit no longer indicates whether a sum is in range or out of range. An *overflow condition* occurs when the result of an operation is out of range. To flag this condition for signed numbers, the CPU contains another special bit called the *overflow bit*, denoted by the letter V. When the CPU adds two binary integers, if their sum is out of range when interpreted in the two's complement representation, then V is set to 1. Otherwise V is cleared to 0.

The V bit detects overflow for signed integers.

The CPU performs the same addition operation regardless of the interpretation of the bit pattern. As with the C bit, the CPU does not stop if a two's complement overflow occurs. It sets the V bit and continues with its next task. It is up to the software to examine the V bit after the addition.

Example 3.17 Here are some examples with a six-bit cell, showing the effects on the carry bit and on the overflow bit:

Adding two positives:		00 0011		01 0110
	ADD	01 0101	ADD	00 1100
	V = 0	01 1000	V = 1	10 0010
	C = 0		C = 0	
Adding a positive and a negative:		00 0101		00 1000
	ADD	11 0111	ADD	11 1010
	V = 0	11 1100	V = 0	00 0010
	C = 0		C = 1	
Adding two negatives:		11 1010		10 0110
	ADD	11 0111	ADD	10 0010
	V = 0	11 0001	V = 1	00 1000
	C = 1		C = 1	

Notice that all combinations of values are possible for V and C. ∎

How can you tell if an overflow condition will occur? One way would be to convert the two numbers to decimal, add them, and see if their sum is outside the range as written in decimal. If so, an overflow has occurred.

The hardware detects an overflow by comparing the carry into the sign bit with the C bit. If they are different, an overflow has occurred, and V gets 1. If they are the same, V gets 0.

Instead of comparing the carry into the sign bit with C, you can tell directly by inspecting the signs of the numbers and the sum. If you add two positive numbers and get a negative sum, or if you add two negative numbers and get a positive sum, then an overflow occurred. It is not possible to get an overflow by adding a positive number and a negative number.

The Negative and Zero Bits

In addition to the C bit, which detects an overflow condition for unsigned integers, and the V bit, which detects an overflow condition for signed integers, the CPU maintains two other bits that the software can test after it performs an operation. They are the N bit, for detecting a negative result, and the Z bit, for detecting a zero result. In summary, the function of these four status bits is

- N = 1 if the result is negative.

  N = 0 otherwise.
- Z = 1 if the result is all zeros.

  Z = 0 otherwise.
- V = 1 if a signed integer overflow occurred.

  V = 0 otherwise.
- C = 1 if an unsigned integer overflow occurred.

  C = 0 otherwise.

The N bit is easy for the hardware to determine, as it is simply a copy of the sign bit. It takes a little more work for the hardware to determine the Z bit, because it must determine if every bit of the result is zero. Chapter 10 shows how the hardware computes the status bits from the result.

Example 3.18 Here are three examples of addition that show the effect of all four status bits on the result.

	01	0110			00	1000			00	1101	
ADD	00	1100		ADD	11	1010		ADD	11	0011	
N = 1	10	0010		N = 0	00	0010		N = 0	00	0000	
Z = 0				Z = 0				Z = 1			
V = 1				V = 0				V = 0			
C = 0				C = 1				C = 1			∎

The default behavior for C and most other HOL6 languages is to ignore the V bit on integer overflow. FIGURE 3.15 is a program that initializes `n`

FIGURE 3.15
An integer overflow in C.

```
#include <stdio.h>
#include <limits.h>

int main() {
   int n = INT_MAX - 2;
   for (int i = 0; i < 6; i++) {
      printf("n == %d\n", n);
      n++;
   }
   return 0;
}
```

Output

```
n == 2147483645
n == 2147483646
n == 2147483647
n == -2147483648
n == -2147483647
n == -2147483646
```

to two less than its maximum value, using `INT_MAX` from the `limits.h` library header file. It executes a loop six times, each time incrementing `n` by 1. The output is from execution on a computer that stores integers in 16-bit cells, making the two's complement range −2,147,483,648 to 2,147,483,647 as written in decimal or 1000 0000 0000 0000 to 0111 1111 1111 1111 as written in binary. When the program adds 1 to the maximum value, it sets C to 0 and V to 1. An overflow occurs, but the program keeps on executing, interpreting the incorrect sum of 1000 0000 0000 0000 as a negative number. Section 6.2 shows how to test the V bit at the Asmb5 level.

3.3 Operations in Binary

Because all information in a computer is stored in binary form, the CPU processes it with binary operations. The previous sections present the binary operations NOT, ADD, and NEG. NOT is a logical operator; ADD and NEG are arithmetic operators. This section describes some other logical and arithmetic operators that are available in the CPU of the computer.

FIGURE 3.16
The truth tables for the AND, OR, and XOR operators at Level ISA3.

p	*q*	*p* AND *q*
0	0	0
0	1	0
1	0	0
1	1	1

(a) ISA3 table for AND.

p	*q*	*p* OR *q*
0	0	0
0	1	1
1	0	1
1	1	1

(b) ISA3 table for OR.

p	*q*	*p* XOR *q*
0	0	0
0	1	1
1	0	1
1	1	0

(c) ISA3 table for XOR.

Logical Operators

You are familiar with the logical operations AND and OR. Another logical operator is the exclusive or, denoted XOR. The exclusive or of logical values *p* and *q* is true if *p* is true, or if *q* is true, but not both. That is, *p* must be true exclusive of *q*, or *q* must be true exclusive of *p*.

One interesting property of binary digits is that you can interpret them as logical quantities. At Level ISA3, a 1 bit can represent true, and a 0 bit can represent false. FIGURE 3.16 shows the truth tables for the AND, OR, and XOR operators at Level ISA3.

At Level HOL6, AND and OR operate on Boolean expressions whose values are either true or false. They are used in `if` statements and loops to test conditions that control the execution of statements. An example of the AND operator is the C phrase

```
if ((ch >= 'a') && (ch <= 'z'))
```

FIGURE 3.17 shows the truth tables for AND, OR, and XOR at Level HOL6. They are identical to Figure 3.16, with 1 at Level ISA3 corresponding to true at Level HOL6, and 0 at Level ISA3 corresponding to false at Level HOL6.

Logical operations are easier to perform than addition because no carries are involved. The operation is applied bitwise to the corresponding bits in the sequence. Neither the carry bit nor the overflow bit is affected by logical operations.

Example 3.19 Some examples for a six-bit cell are

	01 1010		01 1010		01 1010
AND	01 0001	OR	01 0001	XOR	01 0001
N = 0	01 0000	N = 0	01 1011	N = 0	00 1011
Z = 0		Z = 0		Z = 0	

Note that when you take the AND of 1 and 1, the result is 1 with no carry. ∎

FIGURE 3.17
The truth tables for the AND, OR, and XOR operators at Level HOL6.

p	*q*	*p* AND *q*
true	true	true
true	false	false
false	true	false
false	false	false

(a) HOL6 table for AND.

p	*q*	*p* OR *q*
true	true	true
true	false	true
false	true	true
false	false	false

(b) HOL6 table for OR.

p	*q*	*p* XOR *q*
true	true	false
true	false	true
false	true	true
false	false	false

(c) HOL6 table for XOR.

Each of the operations AND, OR, and XOR combines two groups of bits to produce its result. But NOT and NEG operate on only a single group of bits. They are, therefore, called *unary operations*.

Register Transfer Language

The purpose of register transfer language (RTL) is to specify precisely the effect of a hardware operation. The RTL symbols might be familiar to you from your study of logic. FIGURE 3.18 shows the symbols.

The AND and OR operations are known as *conjunction* and *disjunction* in logic. The NOT operator is negation. The implies operator can be

FIGURE 3.18
The register transfer language operations and their symbols.

Operation	RTL Symbol
AND	$\wedge$
OR	$\vee$
XOR	$\oplus$
NOT	$\neg$
Implies	$\Rightarrow$
Transfer	$\leftarrow$
Bit index	$\langle\rangle$
Informal description	{ }
Sequential separator	;
Concurrent separator	,

translated into English as "if/then." The transfer operator is the hardware equivalent of the assignment operator = in C. The memory cell on the left of the operator gets the quantity on the right of the operator. The bit index operator treats the memory cell as an array of bits starting with an index of 0 for the leftmost bit, the same way C indexes an array of elements. The braces enclose an informal English description when a more formal specification would not be helpful.

There are two separators. The sequential separator (semicolon) separates two actions that occur one after the other. The concurrent separator (comma) separates two actions that occur simultaneously.

Example 3.20 In the third computation of Example 3.19, suppose the first six-bit cell is denoted *a*, the second six-bit cell is denoted *b*, and the result is denoted *c*. An RTL specification of the exclusive OR operation is

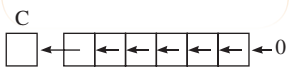
$$c \leftarrow a \oplus b \,;\, \mathrm{N} \leftarrow c < 0 \,,\, \mathrm{Z} \leftarrow c = 0$$

First, *c* gets the exclusive OR of *a* and *b*. After that action, two things happen simultaneously—N gets a Boolean value and Z gets a Boolean value. The Boolean expression $c < 0$ is 1 when *c* is less than zero and 0 when it is not. ■

Arithmetic Operators

Two other unary operations are ASL, which stands for *arithmetic shift left*, and ASR, which stands for *arithmetic shift right*. As the name *ASL* implies, each bit in the cell shifts one place to the left. The bit that was on the leftmost end shifts into the carry bit. The rightmost bit gets 0. FIGURE 3.19 shows the action of the ASL operation for a six-bit cell.

FIGURE 3.19
The action of the ASL operation for a six-bit cell.



Example 3.21 Three examples of the arithmetic shift left operation are

ASL 11 1100 = 11 1000, N = 1, Z = 0, V = 0, C = 1
ASL 00 0011 = 00 0110, N = 0, Z = 0, V = 0, C = 0
ASL 01 0110 = 10 1100, N = 1, Z = 0, V = 1, C = 0 ■

The operation is called an *arithmetic shift* because of the effect it has when the bits represent an integer. Assuming unsigned binary representation, the three integers in the previous example before the shift are

60 3 22 (dec, unsigned)

After the shift, they are

56 6 44 (dec, unsigned)

ASL doubles the number.

The effect of ASL is to double the number. ASL could not double the 60 because 120 is out of range for a six-bit unsigned integer. If the carry bit is 1

after the shift, an overflow has occurred when you interpret the binary sequence as an unsigned integer.

In the decimal system, a left shift produces the same effect, but the integer is multiplied by 10 instead of by 2. For example, a decimal ASL applied to 356 would give 3560, which is 10 times the original value.

What if you interpret the numbers in two's complement representation? Then the three integers before the shift are

$$-4 \quad 3 \quad 22 \qquad \text{(dec, signed)}$$

After the shift, they are

$$-8 \quad 6 \quad -20 \qquad \text{(dec, signed)}$$

Again, the effect of the ASL is to double the number, even if it is negative. This time ASL could not double the 22 because 44 is out of range when you assume two's complement representation. This overflow condition causes the V bit to be set to 1. The situation is similar to the ADD operation, where the C bit detects overflow of unsigned values, but the V bit is necessary to detect overflow of signed values.

The RTL specification for an arithmetic shift left on a six-bit cell r is

$$C \leftarrow r\langle 0\rangle \,,\; r\langle 0..4\rangle \leftarrow r\langle 1..5\rangle \,,\; r\langle 5\rangle \leftarrow 0 \,;$$
$$N \leftarrow r < 0 \,,\; Z \leftarrow r = 0 \,,\; V \leftarrow \{\text{overflow}\}$$

Simultaneously, C gets the leftmost bit of r, the leftmost five bits of r get the values of the bits immediately to their right, and the last bit on the right gets 0. After the values are shifted, the N, Z, and V status bits are set according to the new values in r. It is important to distinguish between the semicolon, which separates two events (each of which has three parts), and the comma, which separates simultaneous events within the parts. The braces indicate less formally that the V bit is set according to whether the result overflowed when you interpret the value as a signed integer.

In the ASR operation, each bit in the group shifts one place to the right. The least significant bit shifts into the carry bit, and the most significant bit remains unchanged. FIGURE 3.20 shows the action of the ASR operation for a six-bit cell. The ASR operation does not affect the V bit because an overflow is impossible when you divide a number by 2.

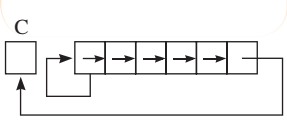


FIGURE 3.20
The action of the ASR operation for a six-bit cell.

Example 3.22 Four examples of the arithmetic shift right operation are

$$\begin{aligned}
&\text{ASR } 01\ 0100 = 00\ 1010, && N = 0, Z = 0, C = 0\\
&\text{ASR } 01\ 0111 = 00\ 1011, && N = 0, Z = 0, C = 1\\
&\text{ASR } 11\ 0010 = 11\ 1001, && N = 1, Z = 0, C = 0\\
&\text{ASR } 11\ 0101 = 11\ 1010, && N = 1, Z = 0, C = 1
\end{aligned}$$

The ASR operation is designed specifically for the two's complement representation. Because the sign bit does not change, negative numbers remain negative and positive numbers remain positive.

ASR halves the number.

Shifting to the left multiplies an integer by 2, whereas shifting to the right divides it by 2. Before the shift, the four integers in the previous example are

20 23 −14 −11 (dec, signed)

After the shift, they are

10 11 −7 −6 (dec, signed)

The even integers can be divided by 2 exactly, so there is no question about the effect of ASR on them. When odd integers are divided by 2, the result is always rounded down. For example, $23 \div 2 = 11.5$, and 11.5 rounded down is 11. Similarly, $-11 \div 2 = -5.5$, and −5.5 rounded down is −6. Note that −6 is less than −5.5 because it lies to the left of −5.5 on the number line.

Rotate Operators

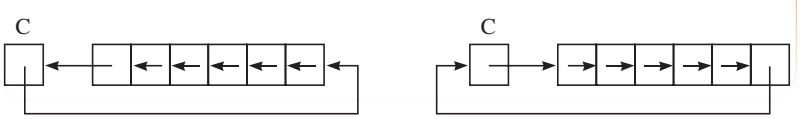
In contrast to the arithmetic operators, the rotate operators do not interpret a binary sequence as an integer. Consequently, the rotate operations do not affect the N, Z, or V bits, but only the C bit. There are two rotate operators—rotate left, denoted ROL, and rotate right, denoted ROR. FIGURE 3.21 shows the actions of the rotate operators for a six-bit cell. Rotate left is similar to arithmetic shift left, except that the C bit is rotated into the rightmost bit of the cell instead of 0 shifting into the rightmost bit. Rotate right does the same thing but in the opposite direction.

The RTL specification for a rotate left on a six-bit cell is

$$C \leftarrow r\langle 0\rangle\,,\ r\langle 0\,..4\rangle \leftarrow r\langle 1..5\rangle\,,\ r\langle 5\rangle \leftarrow C$$

Although the carry bit is the only status bit affected for the rotate left operation at Level ISA3, it affects the V bit at Level Mc2 (as discussed in Chapter 10).

FIGURE 3.21
The action of the rotate operators.



(a) The rotate left operation. **(b)** The rotate right operation.

Example 3.23 Four examples of the rotate operation are

C = 1, ROL 01 1101 = 11 1011, C = 0
C = 0, ROL 01 1101 = 11 1010, C = 0
C = 1, ROR 01 1101 = 10 1110, C = 1
C = 0, ROR 01 1101 = 00 1110, C = 1

where the value of C before the rotate is on the left and the value of C after the rotate is on the right. ∎

3.4 Hexadecimal and Character Representations

The binary representations in the previous sections are integer representations. This section deals with yet another number base, which will be used with the computer introduced in the next chapter. It also shows how that computer stores alphabetic information.

Hexadecimal

Suppose humans had 16 fingers instead of 10. What would have happened when Arabic numerals were invented? Remember the pattern. With 10 fingers, you start from 0 and keep inventing new symbols—1, 2, and so on until you get to your penultimate finger, 9. Then on your last finger you combine 1 and 0 to represent the next number, 10.

With 16 fingers, when you get to 9 you still have plenty of fingers left. You must go on inventing new symbols. These extra symbols are usually represented by the letters at the beginning of the English alphabet. So counting in base 16 (*hexadecimal*, or *hex* for short) looks like this:

Counting in hexadecimal

0	7	E	15	1C	23
1	8	F	16	1D	24
2	9	10	17	1E	25
3	A	11	18	1F	26
4	B	12	19	20	⋮
5	C	13	1A	21	
6	D	14	1B	22	

When the hexadecimal number contains many digits, counting can be a bit tricky. Consider counting the next five numbers in hexadecimal, starting with 8BE7, C9D, or 9FFE:

8BE7	C9D	9FFE
8BE8	C9E	9FFF
8BE9	C9F	A000
8BEA	CA0	A001
8BEB	CA1	A002
8BEC	CA2	A003

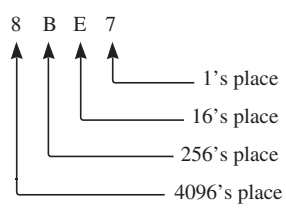
When written in octal, numbers have a tendency to look larger than they actually are. In hexadecimal, the effect is the opposite. Numbers have a tendency to look smaller than they actually are. Comparing the list of hexadecimal numbers with the list of decimal numbers shows that 18 (hex) is 24 (dec).

Base Conversions

In hexadecimal, each place value is 16 times greater than the previous place value. To convert from hexadecimal to decimal, simply multiply the place value by its digit and add.

Example 3.24 FIGURE 3.22 shows how to convert 8BE7 from hexadecimal to decimal. The decimal value of B is 11, and the decimal value of E is 14.

The procedure for converting from decimal to hexadecimal is analogous to the procedure for converting from decimal to binary. Instead of successively dividing the number by 2, you divide it by 16 and keep track of the remainders, which are the hexadecimal digits of the converted number.



(a) The place values for 8BE7.

$$
\begin{aligned}
7 \times \quad 1 &= \quad 7 \\
14 \times \quad 16 &= \quad 224 \\
11 \times \quad 256 &= \quad 2816 \\
8 \times 4096 &= 32{,}768 \\
&\quad\overline{35{,}815}
\end{aligned}
$$

(b) Converting 8BE7 to decimal.

FIGURE 3.22 Converting from hexadecimal to decimal.

FIGURE 3.23
The hexadecimal conversion chart.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0_	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1_	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2_	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3_	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4_	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5_	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6_	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7_	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8_	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9_	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A_	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B_	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C_	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D_	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E_	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F_	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

For numbers up to 255 (dec) or FF (hex), converting either way is easily done with the table in FIGURE 3.23. The body of the table contains decimal numbers. The left column and top row contain hexadecimal digits.

Example 3.25 To convert 9C (hex) to decimal, look up row 9 and column C to find 156 (dec). To convert 125 (dec), look it up in the body of the table and read off 7D (hex) from the left column and top row. ∎

Hexadecimal as a shorthand for binary

If computers store information in binary format, why learn the hexadecimal system? The answer lies in the special relationship between hexadecimal and binary, as FIGURE 3.24 shows. There are 16 possible combinations of four bits, and there are exactly 16 hexadecimal digits. Each hexadecimal digit, therefore, represents four bits.

Bit patterns are often written in hexadecimal notation to save space on the printed page. A computer manual for a 16-bit machine might state that a memory location contains 01D3. That is shorter than saying it contains 0000 0001 1101 0011.

To convert from unsigned binary to hexadecimal, partition the bits into groups of four starting from the rightmost end, and use the hexadecimal from Figure 3.23 for each group. To convert from hexadecimal to unsigned binary, simply reverse the procedure.

Example 3.26 To write the 10-bit unsigned binary number 10 1001 1100 in hexadecimal, start with the rightmost four bits, 1100:

10 1001 1100 (bin) = 29C (hex)

Because 10 bits cannot be partitioned into groups of four exactly, you must assume two additional leading 0's when looking up the leftmost digit in Figure 3.23. The leftmost hexadecimal digit comes from

10 (bin) = 0010 (bin) = 2 (hex)

in this example. ∎

Example 3.27 For a 14-bit cell,

0D60 (hex) = 00 1101 0110 0000 (bin)

Note that the last hexadecimal 0 represents four binary 0's, but the first hexadecimal 0 represents only two binary 0's. ∎

To convert from decimal to unsigned binary, you may prefer to use the hexadecimal table as an intermediate step. You can avoid any computation by looking up the hexadecimal value in Figure 3.22 and then converting each digit to binary according to Figure 3.23.

Example 3.28 For a six-bit cell,

29 (dec) = 1D (hex) = 01 1101 (bin)

where each step in the conversion is a simple table lookup. ∎

In machine language program listings or program traces, numbers are rarely written in hexadecimal notation with negative signs. Instead, the sign bit is implicit in the bit pattern represented by the hexadecimal digits. You must remember that hexadecimal is only a convenient shorthand for a binary sequence. The hardware stores only binary values.

Example 3.29 If a 10-bit memory location contains 37A (hex), then the number in decimal is found by considering the following bit pattern:

37A (hex) = 11 0111 1010 (bin)

FIGURE 3.24
The relationship between hexadecimal and binary.

Hexadecimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

The sign bit is 1, so the number is negative. Converting to decimal gives

37A (hex) = −134 (dec)

Notice that the hexadecimal number is not written with a negative sign, even though it may be interpreted as a negative number.

ASCII Characters

ASCII

Because computer memories are binary, alphabetic characters must be coded to be stored in memory. A widespread binary code for alphabetic characters is the *American Standard Code for Information Interchange*, also known as *ASCII* (pronounced *askey*).

ASCII contains all the uppercase and lowercase English letters, the 10 numeric digits, and special characters such as punctuation signs. Some of its symbols are nonprintable and are used mainly to transmit information between computers or to control peripheral devices.

ASCII is a seven-bit code. Because there are $2^7 = 128$ possible combinations of seven bits, there are 128 ASCII characters. FIGURE 3.25 shows all these characters. The first column of the table shows the nonprintable characters, whose meanings are listed at the bottom. The rest of the table lists the printable characters.

Example 3.30 The sequence 000 0111, which stands for *bell*, causes a terminal to beep. Two other examples of nonprintable characters are *ACK* for *acknowledge* and *NAK* for *negative acknowledge*, which are used by some data transmission protocols. If the sender sends a packet of information over the channel that is detected error-free, the receiver sends an ACK back to the sender, which then sends the next packet. If the receiver detects an error, it sends a NAK back to the sender, which then resends the packet that was damaged in the initial transmission.

Example 3.31 The name

```
Tom
```

would be stored in ASCII as

101 0100
110 1111
110 1101

If that sequence of bits were sent to an output terminal, the word "Tom" would be displayed.

FIGURE 3.25
The American Standard Code for Information Interchange (ASCII).

Char	Bin	Hex	Char	Bin	Hex	Char	Bin	Hex	Char	Bin	Hex
NUL	000 0000	00	SP	010 0000	20	@	100 0000	40	`	110 0000	60
SOH	000 0001	01	!	010 0001	21	A	100 0001	41	a	110 0001	61
STX	000 0010	02	"	010 0010	22	B	100 0010	42	b	110 0010	62
ETX	000 0011	03	#	010 0011	23	C	100 0011	43	c	110 0011	63
EOT	000 0100	04	$	010 0100	24	D	100 0100	44	d	110 0100	64
ENQ	000 0101	05	%	010 0101	25	E	100 0101	45	e	110 0101	65
ACK	000 0110	06	&	010 0110	26	F	100 0110	46	f	110 0110	66
BEL	000 0111	07	'	010 0111	27	G	100 0111	47	g	110 0111	67
BS	000 1000	08	(	010 1000	28	H	100 1000	48	h	110 1000	68
HT	000 1001	09	)	010 1001	29	I	100 1001	49	i	110 1001	69
LF	000 1010	0A	*	010 1010	2A	J	100 1010	4A	j	110 1010	6A
VT	000 1011	0B	+	010 1011	2B	K	100 1011	4B	k	110 1011	6B
FF	000 1100	0C	,	010 1100	2C	L	100 1100	4C	l	110 1100	6C
CR	000 1101	0D	–	010 1101	2D	M	100 1101	4D	m	110 1101	6D
SO	000 1110	0E	.	010 1110	2E	N	100 1110	4E	n	110 1110	6E
SI	000 1111	0F	/	010 1111	2F	O	100 1111	4F	o	110 1111	6F
DLE	001 0000	10	0	011 0000	30	P	101 0000	50	p	111 0000	70
DC1	001 0001	11	1	011 0001	31	Q	101 0001	51	q	111 0001	71
DC2	001 0010	12	2	011 0010	32	R	101 0010	52	r	111 0010	72
DC3	001 0011	13	3	011 0011	33	S	101 0011	53	s	111 0011	73
DC4	001 0100	14	4	011 0100	34	T	101 0100	54	t	111 0100	74
NAK	001 0101	15	5	011 0101	35	U	101 0101	55	u	111 0101	75
SYN	001 0110	16	6	011 0110	36	V	101 0110	56	v	111 0110	76
ETB	001 0111	17	7	011 0111	37	W	101 0111	57	w	111 0111	77
CAN	001 1000	18	8	011 1000	38	X	101 1000	58	x	111 1000	78
EM	001 1001	19	9	011 1001	39	Y	101 1001	59	y	111 1001	79
SUB	001 1010	1A	:	011 1010	3A	Z	101 1010	5A	z	111 1010	7A
ESC	001 1011	1B	;	011 1011	3B	[	101 1011	5B	{	111 1011	7B
FS	001 1100	1C	<	011 1100	3C	\	101 1100	5C	|	111 1100	7C
GS	001 1101	1D	=	011 1101	3D	]	101 1101	5D	}	111 1101	7D
RS	001 1110	1E	>	011 1110	3E	^	101 1110	5E	~	111 1110	7E
US	001 1111	1F	?	011 1111	3F	_	101 1111	5F	DEL	111 1111	7F

Abbreviations for Control Characters

NUL	null, or all zeros	**FF**	form feed	**CAN**	cancel
SOH	start of heading	**CR**	carriage return	**EM**	end of medium
STX	start of text	**SO**	shift out	**SUB**	substitute
ETX	end of text	**SI**	shift in	**ESC**	escape
EOT	end of transmission	**DLE**	data link escape	**FS**	file separator
ENQ	enquiry	**DC1**	device control 1	**GS**	group separator
ACK	acknowledge	**DC2**	device control 2	**RS**	record separator
BEL	bell	**DC3**	device control 3	**US**	unit separator
BS	backspace	**DC4**	device control 4	**SP**	space
HT	horizontal tabulation	**NAK**	negative acknowledge	**DEL**	delete
LF	line feed	**SYN**	synchronous idle		
VT	vertical tabulation	**ETB**	end of transmission block		

Example 3.32 The street address

```
52 Elm
```

would be stored in ASCII as

011 0101
011 0010
010 0000
100 0101
110 1100
110 1101

The blank space between `2` and `E` is a separate ASCII character. ■

The End of the Line

The ASCII standard was developed in the early 1960s and was intended for use on teleprinter machines of that era. A popular device that used the ASCII code was the Teletype Model 33, a mechanical printer with a continuous roll of paper that wrapped around a cylindrical carriage similar to a typewriter. The teleprinter received a stream of ASCII characters over a telephone line and printed the characters on paper.

The nonprintable characters are also known as *control characters* because they were originally used to control the mechanical aspects of a teleprinter. In particular, the ASCII LF control character stands for *line feed*. When the teleprinter received the LF character, it would rotate the carriage enough to advance the paper by one line. Another control character, CR, which stands for *carriage return*, would move the print head to the leftmost position of the page. Because these two mechanical operations were necessary to make the printer mechanism start at the beginning of a new line, the convention was to always use CR-LF to mark the beginning of a new line in a message that was to be sent to a teleprinter.

When early computer companies, notably Digital Equipment Corporation, adopted the ASCII code, they kept this CR-LF convention to denote the end of a line of text. It was convenient because many of those early machines used teleprinters as output devices. The convention was picked up by IBM and Microsoft when they developed the PC DOS and MS-DOS operating systems. MS-DOS eventually became Microsoft Windows, and the CR-LF convention has stuck to this day, despite the disappearance of the old teleprinter for which it was necessary.

Multics was an early operating system that was the forerunner of Unix. To simplify storage and processing of textual data, it adopted the convention of using only the LF character to denote the end of a line. This convention was picked up by Unix and continued by Linux, and is also the convention for OS X because it is Unix.

Figure 2.13 is a C program that reads a stream of characters from the input device and outputs the same stream of characters but substitutes the newline character, denoted `\n` in the string, for each space. The newline character corresponds to the ASCII LF control character. The program in Figure 2.13 works even if you run it in a Windows environment. The C standard specifies that if you output the `\n` character in a `printf()` string, the system will convert it to the convention for the operating system on which you are executing the program. In a Windows system, the `\n` character is converted to two characters, CR-LF, in the output stream. In a Unix system, it remains LF. If you ever need to process the CR character explicitly in a C program, you can write it as `\r`.

Unicode Characters

The first electronic computers were developed to perform mathematical calculations with numbers. Eventually, they processed textual data as well, and the ASCII code became a widespread standard for processing text with the Latin alphabet. As computer technology spread around the world, text processing in languages with different alphabets produced many incompatible systems. The Unicode Consortium was established to collect and catalog all the alphabets of all the spoken languages in the world, both current and ancient, as a first step toward a standard system for the worldwide interchange, processing, and display of texts in these natural languages.

Strictly speaking, the standard organizes characters into scripts, not languages. It is possible for one script to be used in multiple languages. For example, the extended Latin script can be used for many European and American languages. Version 7.0 of the Unicode standard has 123 scripts for natural language and 15 scripts for other symbols. Examples of natural language scripts are Balinese, Cherokee, Egyptian Hieroglyphs, Greek, Phoenician, and Thai. Examples of scripts for other symbols are Braille Patterns, Emoticons, Mathematical Symbols, and Musical Symbols.

Each character in every script has a unique identifying number, usually written in hexadecimal, and is called a *code point*. The hexadecimal number is preceded by "U+" to indicate that it is a Unicode code point. Corresponding to a code point is a *glyph*, which is the graphic representation of the symbol on the page or screen. For example, in the Hebrew script, the code point U+05D1 has the glyph ב.

FIGURE 3.26 shows some example code points and glyphs in the Unicode standard. The CJK Unified script is for the written languages of China, Japan,

Unicode Script	Code Point	Glyphs							
		0	1	2	3	4	5	6	7
Arabic	U+063_	ذ	ر	ز	س	ش	ص	ض	ط
Armenian	U+054_	Ձ	Ղ	Ճ	Մ	Յ	Ն	Շ	Ո
Braille Patterns	U+287_	⡠	⡡	⡢	⡣	⡤	⡥	⡦	⡧
CJK Unified	U+4EB_	京	亱	亲	亳	亴	亵	亶	亷
Cyrillic	U+041_	А	Б	В	Г	Д	Е	Ж	З
Egyptian Hieroglyphs	U+1300_	𓀀	𓀁	𓀂	𓀃	𓀄	𓀅	𓀆	𓀇
Emoticons	U+1F61_	😐	😑	😒	😓	😔	😕	😖	😗
Hebrew	U+05D_	א	ב	ג	ד	ה	ו	ז	ח
Basic Latin (ASCII)	U+004_	@	A	B	C	D	E	F	G
Latin-1 Supplement	U+00E_	à	á	â	ã	ä	å	æ	ç

FIGURE 3.26
A few code points and glyphs from the Unicode character set.

and Korea, which share a common character set with some variations. There are tens of thousands of characters in these Asian writing systems, all based on a common set of Han characters. To minimize unnecessary duplication, the Unicode Consortium merged the characters into a single set of unified characters. This Han unification is an ongoing process carried out by a group of experts from the Chinese-speaking countries, North and South Korea, Japan, Vietnam, and other countries.

Code points are backward compatible with ASCII. For example, from the ASCII table in Figure 3.25, the Latin letter S is stored with seven bits as 101 0011 (bin), which is 53 (hex). So, the Unicode code point for S is U+0053. The standard requires at least four hex digits following U+, padding the number with leading zeros if necessary.

A single code point can have more than one glyph. For example, an Arabic letter may be displayed with different glyphs depending on its position in a word. On the other hand, a single glyph might be used to represent two code points. The consecutive Latin code points U+0066 and U+0069 for f and i are frequently rendered with the ligature glyph fi.

The range of the Unicode code space is 0 to 10FFFF (hex), or 0 to 1 0000 1111 1111 1111 1111 (bin), or 0 to 1,114,111 (dec). About one-fourth of these million-plus code points have been assigned. Some values are reserved for private use, and each Unicode standard revision assigns a few more values to code points. It is theoretically possible to represent each code point with a single 21-bit number. Because computer memory is normally organized into eight-bit bytes, it would be possible to use three bytes to store each code point with the leading three bits unused.

However, most computers process information in chunks of either 32 bits (4 bytes) or 64 bits (8 bytes). It follows that the most effective method for processing textual information is to store each code point in a 32-bit cell, even though the leading 11 bits would be unused and always set to zeros. This method of encoding is called *UTF-32*, where *UTF* stands for *Unicode Transformation Format*. UTF-32 always requires eight hexadecimal characters to represent its four bytes.

Example 3.33 To determine how the letter z is stored in UTF-32, look up its value in the ASCII table as 7A (hex). Because Unicode code points are backward compatible with ASCII, the code point for the letter z is U+007A. The UTF-32 encoding in binary is obtained by prefixing zeros for a total of 32 bits as follows:

0000 0000 0000 0000 0000 0000 0111 1010

So, U+007A is encoded as 0000 007A (UTF-32). ∎

Example 3.34 To determine the UTF-32 encoding of the emoticon 😮 with code point U+1F617, simply prefix the correct number of zeros. The encoding is 0001 F617 (UTF-32). ∎

Although UTF-32 is effective for processing textual information, it is inefficient for storing and transmitting textual information. If you have a file that stores mostly ASCII characters, three-fourths of the file space will be occupied by zeros. UTF-8 is a popular encoding standard that is able to represent every Unicode character. It uses one to four bytes to store a single character and therefore takes less storage space than UTF-32. The 64 Ki code points in the range U+0000 to U+FFFF, known as the *Basic Multilingual Plane*, contain characters for almost all modern languages. UTF-8 can represent each of these code points with one to three bytes and uses only a single byte for an ASCII character.

FIGURE 3.27 shows the UTF-8 encoding scheme. The first column, labeled *Bits*, represents the upper limit of the number of bits in the code point, excluding all leading zeros. The x's in the code represent the rightmost bits from the code point, which are spread out over one to four bytes.

The first row in the table corresponds to the ASCII characters, which have an upper limit of seven bits. An ASCII character is stored as a single byte whose first bit is 0 and whose last seven bits are identical to seven-bit ASCII. The first step in decoding a UTF-8 string is to inspect the first bit of the first byte. If it is zero, the first character is an ASCII character, which can be determined from the ASCII table, and the following byte is the first byte of the next character.

If the first bit of the first byte is 1, the first character is outside the range U+0000 to U+007F—that is, it is not an ASCII character, and it occupies more than one byte. In this case, the number of leading 1's in the first byte

FIGURE 3.27
The UTF-8 encoding scheme.

Bits	First Code Point	Last Code Point	Byte 1	Byte 2	Byte 3	Byte 4
7	U+0000	U+007F	0xxxxxxx			
11	U+0080	U+07FF	110xxxxx	10xxxxxx		
16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
21	U+10000	U+1FFFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

is equal to the total number of bytes occupied by the character. Some of the bits from the code point are stored in the first byte and some are stored in the remaining continuation bytes. Every continuation byte begins with the string 10 and stores six bits from the code point.

Example 3.35 To determine the UTF-8 encoding of the emoticon ☺ with code point U+1F617, first determine the upper limit of the number of bits in the code point. From Figure 3.27, it is in the range U+10000 to U+1FFFF; thus, the rightmost 21 bits from the code point are spread out over 4 bytes. The rightmost 21 bits from 1F617 (hex) are

0 0001 1111 0110 0001 0111

where enough leading zeros are added to total 21 bits. The last row in Figure 3.27 shows the first three bits stored in Byte 1, the next six stored in Byte 2, the next six stored in Byte 3, and the last six stored in Byte 4. Regrouping the 21 bits accordingly yields

000 011111 011000 010111

The format of Byte 1 from the table is 11110xxx, so insert the first three zeros in place of the x's to yield 11110000, and do the same for Bytes 3 and 4. The resulting bit pattern of the four bytes is

11110000 10011111 10011000 10010111

So, U+1F617 is encoded as F09F 9897 (UTF-8), which is different from the four bytes of the UTF-32 encoding in Example 3.34. ■

Example 3.36 To determine the sequence of code points from the UTF-8 byte sequence 70 C3 A6 6F 6E, first write the byte sequence in binary as

01110000 11000011 10100110 01101111 01101110

You can immediately determine that the first, fourth, and fifth bytes are ASCII characters because the leading bit is zero in those bytes. From the ASCII table, these bytes correspond to the letters p, o, and n, respectively. The leading 110 in the second byte indicates that 11 bits are spread out over 2 bytes per the second row in the body of the table in Figure 3.27. The leading 10 in the third byte is consistent, because that prefix denotes a continuation byte. Extracting the rightmost 5 bits from the second byte (first byte of the pair) and the rightmost 6 bytes from the third byte (second byte of the pair) yields the 11 bits:

00011 100110

Prefixing this pattern with leading zeros and regrouping yields

0000 0000 1110 0110

which is the code point U+00E6 corresponding to Unicode character æ. So, the original five-byte sequence is a UTF-8 encoding of the four code points U+0070, U+00E6, U+006F, U+006E and represents the string "pæon". ■

Figure 3.27 shows that UTF-8 does not allow all possible bit patterns. For example, it is illegal to have the bit pattern

11100011 01000001

in a UTF-8–encoded file because the 1110 prefix of the first byte indicates that it is the first byte of a three-byte sequence, but the leading zero of the second byte indicates that it is a single ASCII character and not a continuation byte. If such a pattern is detected in a UTF-8–encoded file, the data is corrupted.

A major benefit of UTF-8 is its self-synchronization property. A decoder can uniquely identify the type of any byte in the sequence by inspection of the prefix bits. For example, if the first two bits are 10, it is a continuation byte. Or, if the first four bits are 1110, it is the first byte of a three-byte sequence. This self-synchronization property makes it possible for a UTF-8 decoder to recover most of the text when data corruption does occur.

UTF-8 is by far the most common encoding standard on the World Wide Web. It has become the default standard for multilingual applications. Operating systems are incorporating UTF-8 so that documents and files can be named in the user's native language. Modern programming languages such as Python and Swift have UTF-8 built in so a programmer can, for example, name a variable pæon or even 😐😑. Text editors that have traditionally processed only pure ASCII text, as opposed to word processors that have always been format friendly, are increasingly able to process UTF-8–encoded text files.

3.5 Floating-Point Representation

The numeric representations described in previous sections of this chapter are for integer values. C has three numeric types that have fractional parts:

- `float` single-precision floating point
- `double` double-precision floating point
- `long double` extended-precision floating point

Values of these types cannot be stored at Level ISA3 with two's complement binary representation because provisions must be made for locating the decimal point within the number. Floating-point values are stored using a binary version of scientific notation.

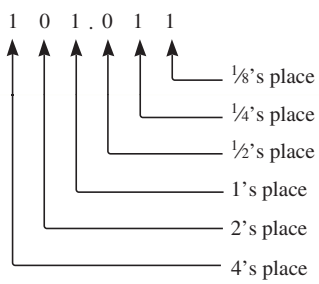
Binary Fractions

Binary fractions have a binary point, which is the base 2 version of the base 10 decimal point.

Example 3.37 FIGURE 3.28(a) shows the place values for 101.011 (bin). The bits to the left of the binary point have the same place values as the corresponding bits in unsigned binary representation, as in Figure 3.2. Starting with the 1/2's place to the right of the binary point, each place has a value one-half as great as the previous place value. Figure 3.28(b) shows the addition that produces the 5.375 (dec) value. ∎

FIGURE 3.29 shows the polynomial representation of numbers with fractional parts. The value of the bit to the left of the radix point is always the base to the zeroth power, which is always 1. The next significant place to the left is the base to the first power, which is the value of the base itself. The value of the bit to the right of the radix point is the base to the power −1. The next significant place to the right is the base to the power −2. The value of each place to the right is 1/base times the value of the place on its left.

FIGURE 3.28
Converting from binary to decimal.



(a) The place values for 101.011 (bin).

1 ⅛'s place = 0.125
1 ¼'s place = 0.25
0 ½'s place = 0.0
1 1's place = 1.0
0 2's place = 0.0
1 4's place = 4.0
5.375 (dec)

(b) Converting 101.011 (bin) to decimal.

FIGURE 3.29
The polynomial representation of floating-point numbers.

$$1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$$

(a) The binary number 101.011.

$$5 \times 10^2 + 0 \times 10^1 + 6 \times 10^0 + 7 \times 10^{-1} + 2 \times 10^{-2} + 1 \times 10^{-3}$$

(b) The decimal number 506.721.

Determining the decimal value of a binary fraction requires two steps. First, convert the bits to the left of the binary point using the technique of Example 3.3 for converting unsigned binary values. Then, use the algorithm of successive doubling to convert the bits to the right of the binary point.

Example 3.38 FIGURE 3.30 shows the conversion of 6.5859375 (dec) to binary. The conversion of the whole part gives 110 (bin) to the left of the binary point. To convert the fractional part, write the digits to the right of the decimal point in the heading of the right column of the table. Double the fractional part, writing the digit to the left of the decimal point in the column on the left and the fractional part in the column on the right. The next time you double, do not include the whole number part. For example, the value 0.34375 comes from doubling 0.171875, not from doubling 1.171875. The digits on the left from top to bottom are the bits of the binary fractional part from left to right. So, 6.5859375 (dec) = 110.1001011 (bin). ∎

FIGURE 3.30
Converting from decimal to binary.

6.5859375

6 (dec) = 110 (bin)

(a) Convert the whole part.

	.5859375
1	.171875
0	.34375
0	.6875
1	.375
0	.75
1	.5
1	.0

(b) Convert the fractional part.

The algorithm for converting the fractional part from decimal to binary is the mirror image of the algorithm for converting the whole part, from decimal to binary. Figure 3.5 shows that to convert the whole part, you use the algorithm of successive division by two. The bits you generate are the remainders of the division, and you generate them from right to left starting at the binary point. To convert the fractional part, you use the algorithm of successive multiplication by two. The bits you generate are the whole part of the multiplication, and you generate them from left to right starting at the binary point.

A number that can be represented with a finite number of digits in decimal may require an endless representation in binary.

Example 3.39 FIGURE 3.31 shows the conversion of 0.2 (dec) to binary. The first doubling produces 0.4. A few more doublings produce 0.4 again. It is clear that the process will never terminate and that 0.2 (dec) = 0.001100110011 . . . (bin) with the bit pattern 0011 endlessly repeating. ∎

FIGURE 3.31
A decimal value with an unending binary representation.

	.2
0	.4
0	.8
1	.6
1	.2
0	.4
0	.8
1	.6
⋮	⋮

Because all computer cells can store only a finite number of bits, the value 0.2 (dec) cannot be stored exactly and must be approximated. You should realize that if you add 0.2 + 0.2 in a Level HOL6 language like C, you will probably not get 0.4 exactly because of the roundoff error inherent in the binary representation of the values. For that reason, good numeric software rarely tests two floating point numbers for strict equality. Instead, the software maintains a small but nonzero tolerance that represents how close two floating point values must be to be considered equal. If the tolerance is, say, 0.0001, then the numbers 1.38264 and 1.38267 would be

considered equal because their difference, which is 0.00003, is less than the tolerance.

Excess Representations

Floating-point numbers are represented with a binary version of the scientific notation common with decimal numbers. A nonzero number is normalized if it is written in scientific notation with the first nonzero digit immediately to the left of the radix point. The number zero cannot be normalized because it does not have a first nonzero digit.

Example 3.40 The decimal number −328.4 is written in normalized form in scientific notation as -3.284×10^2. The effect of the exponent 2 as the power of 10 is to shift the decimal point two places to the right. Similarly, the binary number −10101.101 is written in normalized form in scientific notation as -1.0101101×2^4. The effect of the exponent 4 as the power of 2 is to shift the binary point four places to the right. ▮

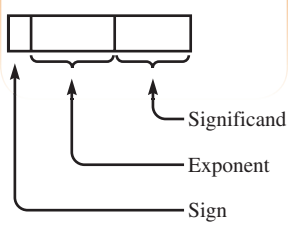
Example 3.41 The binary number 0.00101101 is written in normalized form in scientific notation as 1.01101×2^{-3}. The effect of the exponent −3 as the power of 2 is to shift the binary point three places to the left. ▮

In general, a floating point number can be positive or negative, and its exponent can be a positive or negative integer. FIGURE 3.32 shows a cell in memory that stores a floating point value. The cell is divided into three fields. The first field stores one bit for the sign of the number. The second field stores the bits representing the exponent of the normalized binary number. The third field, called the *significand*, stores bits that represent the magnitude of the value.

The more bits stored in the exponent, the wider the range of floating point values. The more bits stored in the significand, the higher the precision of the representation. A common representation is an 8-bit cell for the exponent and a 23-bit cell for the significand. To present the concepts of the floating point format, the examples in this section use a 3-bit cell for the exponent and a 4-bit cell for the significand. These are unrealistically tiny cell sizes, but they help to illustrate the format without an unwieldy number of bits.

Any signed representation for integers could be used to store the exponent. You might think that two's complement binary representation would be used, because that is the representation that most computers use to store signed integers. However, two's complement is not used. Instead, a biased representation is used for a reason that will be explained later.

FIGURE 3.32
Storage for a floating-point value.



An example of a biased representation for a five-bit cell is excess 15. The range of numbers for the cell is −15 to 16 as written in decimal and 00000 to 11111 as written in binary. To convert from decimal to excess 15, you add 15 to the decimal value and then convert to binary as you would an unsigned number. To convert from excess 15 to decimal, you write the decimal value as if it were an unsigned number and subtract 15 from it. In excess 15, the first bit denotes whether a value is positive or negative. But unlike two's complement representation, 1 signifies a positive value, and 0 signifies a negative value.

Example 3.42 For a five-bit cell, to convert 5 from decimal to excess 15, add 5 + 15 = 20. Then convert 20 to binary as if it were unsigned, 20 (dec) = 10100 (bin). Therefore, 5 (dec) = 10100 (excess 15). The first bit is 1, indicating a positive value. ∎

Example 3.43 To convert 00011 from excess 15 to decimal, convert 00011 as an unsigned value, 00011 (bin) = 3 (dec). Then subtract decimal values 3 − 15 = −12. So, 00011 (excess 15) = −12 (dec). ∎

FIGURE 3.33 shows the bit patterns for a three-bit cell that stores integers with excess 3 representation compared to two's complement representation. Each representation stores eight values. The excess 3 representation has a range of −3 to 4 (dec), while the two's complement representation has a range of −4 to 3 (dec).

Decimal	Excess 3	Two's Complement
−4		100
−3	000	101
−2	001	110
−1	010	111
0	011	000
1	100	001
2	101	010
3	110	011
4	111	

FIGURE 3.33
The signed integers for a three-bit cell.

The Hidden Bit

Figure 3.32 shows one bit reserved for the sign of the number but no bit reserved for the binary point. A bit for the binary point is unnecessary because numbers are stored normalized, so the system can assume that the first 1 is to the left of the binary point. Furthermore, because there will always be a 1 to the left of the binary point, there is no need to store the leading 1 at all. To store a decimal value, first convert it to binary, write it in normalized scientific notation, store the exponent in excess representation, drop the leading 1, and store the remaining bits of the magnitude in the significand. The bit that is assumed to be to the left of the binary point but that is not stored explicitly is called the *hidden bit*.

Example 3.44 Assuming a three-bit exponent using excess 3 and a four-bit significand, how is the number 3.375 stored? Converting the whole number part gives 3 (dec) = 11 (bin). Converting the fractional part gives 0.375 (dec) = 0.011. The complete binary number is 3.375 (dec) = 11.011 (bin), which is 1.1011×2^1 in normalized binary scientific notation. The number is positive, so the sign bit is 0. The exponent is 1 (dec) = 100 (excess 3) from Figure 3.33. Dropping the leading 1, the four bits to the right of the binary point are .1011. So, 3.375 is stored as 0100 1011. ■

Of course, the hidden bit is assumed, not ignored. When you read a decimal floating point value from memory, the compiler assumes that the hidden bit is not stored. It generates code to insert the hidden bit before it performs any computation with the full number of bits. The floating point hardware even adds a few extra bits of precision called *guard digits* that it carries throughout the computation. After the computation, the system discards the guard digits and the assumed hidden bit and stores as many bits to the right of the binary point as the significand will hold.

Not storing the leading 1 allows for greater precision. In the previous example, the bits for the magnitude are 1.1011. Using a hidden bit, you drop the leading 1 and store .1011 in the four-bit significand. In a representation without a hidden bit, you would store the most significant bits, 1.011, in the four-bit significand and be forced to discard the least significant 0.0001 value. The result would be a value that only approximates the decimal value 3.375.

Because every memory cell has a finite number of bits, approximations are unavoidable even with a hidden bit. The system approximates by rounding off the least significant bits it must discard using a rule called "round to nearest, ties to even." FIGURE 3.34 shows how the rule works for decimal and binary numbers. You round off 23.499 to 23 because 23.499 is closer to 23 than it is to 24. Similarly, 23.501 is closer to 24 than it is to 23. However, 23.5 is just as close to 23 as it is to 24, which is a tie. It rounds to 24 because 24

FIGURE 3.34
Round to nearest, ties to even.

Decimal	Decimal Rounded	Binary	Binary Rounded
23.499	23	1011.011	1011
23.5	24	1011.1	1100
23.501	24	1011.101	1100
24.499	24	1100.011	1100
24.5	24	1100.1	1100
24.501	25	1100.101	1101

is even. Similarly, the binary number 1011.1 is just as close to 1011 as it is to 1100, which is a tie. It rounds to 1100 because 1100 is even.

Example 3.45 Assuming a three-bit exponent using excess 3 and a four-bit significand, how is the number −13.75 stored? Converting the whole number part gives 13 (dec) = 1101 (bin). Converting the fractional part gives 0.75 (dec) = 0.11. The complete binary number is 13.75 (dec) = 1101.11 (bin), which is 1.10111×2^3 in normalized binary scientific notation. The number is negative, so the sign bit is 1. The exponent is 3 (dec) = 110 (excess 3). Dropping the leading 1, the five bits to the right of the binary point are .10111. However, only four bits can be stored in the significand. Furthermore, .10111 is just as close to .1011 as it is to .1100, and the tie rule is in effect. Because 1011 is odd and 1100 is even, round to .1100. So, −13.75 is stored as 1110 1100. ∎

Special Values

Some real values require special treatment. The most obvious is zero, which cannot be normalized because there is no 1 bit in its binary representation. You must set aside a special bit pattern for zero. Standard practice is to put all 0's in the exponent field and all 0's in the significand as well. What do you put for the sign? Most common is to have two representations for zero, one positive and one negative. For a three-bit exponent and four-bit significand, the bit patterns are

1 000 0000 (bin) = −0.0 (dec)
0 000 0000 (bin) = +0.0 (dec)

This solution for storing zero has ramifications for some other bit patterns. If the bit pattern for +0.0 were not special, then 0 000 0000 would be interpreted with the hidden bit as 1.0000×2^{-3} (bin) = 0.125, the smallest positive value that could be stored had the value not been reserved for zero. If this pattern is reserved for zero, then the smallest positive value that can be stored is

$$0\ 000\ 0001 = 1.0001 \times 2^{-3}\ (\text{bin}) = 0.1328125$$

which is slightly larger. The negative number with the smallest possible magnitude is identical but with a 1 in the sign bit. The largest positive number that can be stored is the bit pattern with the largest exponent and the largest significand. The bit pattern for the largest value is

$$0\ 111\ 1111\ (\text{bin}) = +31.0\ (\text{dec})$$

FIGURE 3.35 shows the number line for the representation where zero is the only special value. As with integer representations, there is a limit to how large a value you can store. If you try to multiply 9.5 times 12.0, both of which are in range, the true value is 114.0, which is in the positive overflow region.

Unlike integer values, however, the real number line has an underflow region. If you try to multiply 0.145 times 0.145, which are both in range, the true value is 0.021025, which is in the positive underflow region. The smallest positive value that can be stored is 0.132815.

Numeric calculations with approximate floating point values need to have results that are consistent with what would be expected when calculations are done with exact precision. For example, suppose you multiply 9.5 and 12.0. What should be stored for the result? Suppose you store the largest possible value, 31.0, as an approximation. Suppose further that this is an intermediate value in a longer computation. If you later need to compute half of the result, you will get 15.5, which is far from what the correct value would have been.

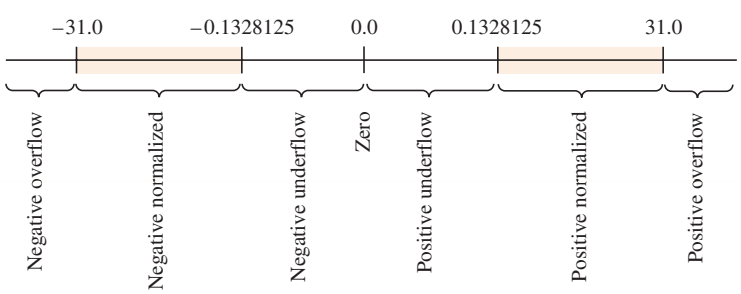


FIGURE 3.35
The real number line with zero as the only special value.

Special Value	Exponent	Significand
Zero	All zeros	All zeros
Denormalized	All zeros	Nonzero
Infinity	All ones	All zeros
Not a number	All ones	Nonzero

FIGURE 3.36
The special values in floating-point representation.

The same problem occurs in the underflow region. If you store 0.0 as an approximation of 0.021025, and you later want to multiply the value by 12.0, you will get 0.0. You risk being misled by what appears to be a reasonable value.

The problems encountered with overflow and underflow are alleviated somewhat by introducing more special values for the bit patterns. As is the case with zero, you must use some bit patterns that would otherwise be used to represent other values on the number line. In addition to zero, three special values are common—infinity, not a number (NaN), and denormalized numbers. FIGURE 3.36 lists the four special values for floating-point representation and their bit patterns.

Infinity

Infinity is used for values that are in the overflow regions. If the result of an operation overflows, the bit pattern for infinity is stored. If further operations are done on this bit pattern, the result is what you would expect for an infinite value. For example, $3/\infty = 0$, $5 + \infty = \infty$, and the square root of infinity is infinity. You can produce infinity by dividing by 0. For example, $3/0 = \infty$, and $-4/0 = -\infty$. If you ever do a computation with real numbers and get infinity, you know that an overflow occurred somewhere in your intermediate results.

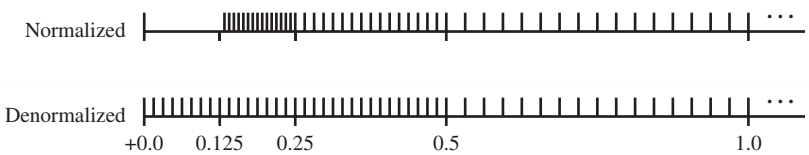
Not a number

A bit pattern for a value that is not a number is called a *NaN* (rhymes with *plan*). NaNs are used to indicate floating point operations that are illegal. For example, taking the square root of a negative number produces NaN, and so does dividing 0/0. Any floating point operation with at least one NaN operand produces NaN. For example, 7 + NaN = NaN, and 7/NaN = NaN.

Both infinity and NaN use the largest possible value of the exponent for their bit patterns. That is, the exponent field is all 1's. The significand is all 0's for infinity and can be any nonzero pattern for NaN. Reserving these bit patterns for infinity and NaN has the effect of reducing the range of values that can be stored. For a three-bit exponent and four-bit significand, the bit patterns for the largest magnitudes and their decimal values are

1 111 0000 (bin) = $-\infty$
1 110 1111 (bin) = −15.5 (dec)
0 110 1111 (bin) = +15.5 (dec)
0 111 0000 (bin) = $+\infty$

FIGURE 3.37
The real number line with and without denormalized numbers.



Denormalized numbers

There is no single infinitesimal value for the underflow region in Figure 3.35 that corresponds to the infinite value in the overflow region. Instead, there is a set of values called *denormalized values* that alleviate the problem of underflow. FIGURE 3.37 is a drawing to scale of the floating point values for a binary representation without denormalized special values (top) and with denormalized values (bottom) for a system with a three-bit exponent and four-bit significand. The figure shows three complete sequences of values for exponent fields of 000, 001, and 010 (excess 3), which represent −3, −2, and −1 (dec), respectively.

For normalized numbers in general, the gap between successive values doubles with each unit increase of the exponent. For example, in the number line on the top, the group of 16 values between 0.125 and 0.25 corresponds to numbers written in binary scientific notation with a multiplier of 2^{-3}. The 16 numbers between 0.25 and 0.5 are spaced twice as far apart and correspond to numbers written in binary scientific notation with a multiplier of 2^{-2}.

Without denormalized special values, the gap between +0.0 and the smallest positive value is excessive compared to the gaps in the smallest sequence. Denormalized special values make the gap between successive values for the first sequence equal to the gap between successive values for the second sequence. It spreads these values out evenly as they approach +0.0 from the right. On the left half of the number line, not shown in the figure, the negative values are spread out evenly as they approach −0.0 from the left.

Gradual underflow

This behavior of denormalized values is called *gradual underflow*. With gradual underflow, the gap between the smallest positive value and zero is reduced considerably. The idea is to take the nonzero values that would be stored with an exponent field of all 0's (in excess representation) and distribute them evenly in the underflow gap.

Because the exponent field of all 0's is reserved for denormalized numbers, the smallest positive normalized number becomes

$$0\ 001\ 0000 = 1.000 \times 2^{-2}\ (\text{bin}) = 0.25\ (\text{dec})$$

It might appear that we have made matters worse because the smallest positive normalized number with 000 in the exponent field is 0.1328125.

But, the denormalized values are spread throughout the gap in such a way as to actually reduce it.

When the exponent field is all 0's and the significand contains at least one 1, special rules apply to the representation. Assuming a three-bit exponent and a four-bit significand,

Representation rules for denormalized numbers

- The hidden bit to the left of the binary point is assumed to be 0 instead of 1.
- The exponent is assumed to be stored in excess 2 instead of excess 3.

Example 3.46 For a representation with a three-bit exponent and four-bit significand, what decimal value is represented by 0 000 0110? Because the exponent is all 0's and the significand contains at least one 1, the number is denormalized. Its exponent is 000 (excess 2) $= 0 - 2 = -2$ (dec), and its hidden bit is 0, so its binary scientific notation is 0.0110×2^{-2}. The exponent is in excess 2 instead of excess 3 because this is the special case of a denormalized number. Converting to decimal yields 0.09375. ∎

To see how much better the underflow gap is, compute the values having the smallest possible magnitudes, which are denormalized.

1 000 0001 (bin) = −0.015625 (dec)
1 000 0000 (bin) = −0.0
0 000 0000 (bin) = +0.0
0 000 0001 (bin) = +0.015625 (dec)

Without denormalized numbers, the smallest positive number is 0.1328125, so the gap has been reduced considerably.

FIGURE 3.38 shows some of the key values for a three-bit exponent and a four-bit significand using all four special values. The values are listed in numeric order from smallest to largest. The figure shows why an excess representation is common for floating point exponents. Consider all the positive numbers from +0.0 to +∞, ignoring the sign bit. You can see that if you treat the rightmost seven bits to be a simple unsigned integer, the successive values increase by one all the way from 000 0000 for 0 (dec) to 111 0000 for ∞. To do a comparison of two positive floating point values, say in a C statement like

```
if (x < y)
```

the computer does not need to extract the exponent field or insert the hidden bit. It can simply compare the rightmost seven bits as if they represented an integer to determine which floating point value has the larger magnitude. The circuitry for integer operations is considerably faster than that for

FIGURE 3.38
Floating-point values for a three-bit exponent and four-bit significand.

	Binary	Scientific Notation	Decimal
Not a number	1 111 nonzero		
Negative infinity	1 111 0000		$-\infty$
Negative normalized	1 110 1111 1 110 1110 . . . 1 011 0001 1 011 0000 1 010 1111 . . . 1 001 0001 1 001 0000	-1.1111×2^{3} -1.1110×2^{3} . . . -1.0001×2^{0} -1.0000×2^{0} -1.1111×2^{-1} . . . -1.0001×2^{-2} -1.0000×2^{-2}	-15.5 -15.0 . . . -1.0625 -1.0 -0.96875 . . . -0.265625 -0.25
Negative denormalized	1 000 1111 1 000 1110 . . . 1 000 0010 1 000 0001	-0.1111×2^{-2} -0.1110×2^{-2} . . . -0.0010×2^{-2} -0.0001×2^{-2}	-0.234375 -0.21875 . . . -0.03125 -0.015625
Negative zero	1 000 0000		-0.0
Positive zero	0 000 0000		$+0.0$
Positive denormalized	0 000 0001 0 000 0010 . . . 0 000 1110 0 000 1111	0.0001×2^{-2} 0.0010×2^{-2} . . . 0.1110×2^{-2} 0.1111×2^{-2}	0.015625 0.03125 . . . 0.21875 0.234375
Positive normalized	0 001 0000 0 001 0001 . . . 0 010 1111 0 011 0000 0 011 0001 . . . 0 110 1110 0 110 1111	1.0000×2^{-2} 1.0001×2^{-2} . . . 1.1111×2^{-1} 1.0000×2^{0} 1.0001×2^{0} . . . 1.1110×2^{3} 1.1111×2^{3}	0.25 0.265625 . . . 0.96875 1.0 1.0625 . . . 15.0 15.5
Positive infinity	0 111 0000		$+\infty$
Not a number	0 111 nonzero		

floating point operations, so using an excess representation for the exponent really improves performance.

The same pattern occurs for the negative numbers. The rightmost seven bits can be treated like an unsigned integer to compare magnitudes of the negative quantities. Floating-point quantities would not have this property if the exponents were stored using two's complement representation.

Figure 3.38 shows that −0.0 and +0.0 are distinct. At this low level of abstraction, negative zero is stored differently from positive zero. However, programmers at a higher level of abstraction expect the set of real number values to have only one zero, which is neither positive nor negative. For example, if the value of `x` has been computed as −0.0 vand `y` as +0.0, then the programmer should expect `x` to have the value 0 and `y` to have the value 0, and the expression `(x < y)` to be false. Computers must be programmed to return false in this special case, even though the bit patterns indicate that `x` is negative and `y` is positive. The system hides the fact that there are two representations of zero at a low level of abstraction from the programmer at a higher level of abstraction.

With denormalization, to convert from decimal to binary you must first check if a decimal value is in the denormalized range to determine its representation. From Figure 3.38, for a three-bit exponent and a four-bit significand, the smallest positive normalized value is 0.25. Any value less than 0.25 is stored with the denormalized format.

Example 3.47 For a representation with a three-bit exponent and four-bit significand, how is the decimal value −0.078 stored? Because 0.078 is less than 0.25, the representation is denormalized, the exponent is all zeros, and the hidden bit is 0. Converting to binary, 0.078 (dec) = 0.000100111.... Because the exponent is all zeros and the exponent is stored in excess 2 representation, the multiplier must be 2^{-2}. In binary scientific notation with a multiplier of 2^{-2}, 0.000100111 ... = 0.0100111 ... × 2^{-2}. As expected, the digit to the left of the binary point is 0, which is the hidden bit. The bits to be stored in the significand are the first four bits of .0100111 . . ., which rounds off to .0101. So the floating point representation for −0.078 is 1000 0101. ∎

The IEEE 754 Floating-Point Standard

The Institute of Electrical and Electronic Engineers, Inc. (IEEE), is a professional society supported by its members that provides services in various engineering fields, one of which is computer engineering. The society has various groups that propose standards for the industry. Before the IEEE proposed its standard for floating point numbers, every computer manufacturer designed its own representation for floating point values, and they all differed from each other. In the early days before networks became

prevalent and little data was shared between computers, this arrangement was tolerated.

Even without the widespread sharing of data, however, the lack of a standard hindered research and development in numerical computations. It was possible for two identical programs to run on two separate machines with the same input and produce different results because of the different approximations of the representations.

The IEEE set up a committee to propose a floating point standard, which it did in 1985. There are two standards: number 854, which is more applicable to handheld calculators than to other computing devices, and number 754, which was widely adopted for computers. The standard was revised with little change in 2008. Virtually every computer manufacturer now provides floating point numbers for their computers that conform to the IEEE 754 standard.

The floating point representation described earlier in this section is identical to the IEEE 754 standard except for the number of bits in the exponent field and in the significand. FIGURE 3.39 shows the two formats for the standard. The single-precision format has an 8-bit cell for the exponent using excess 127 representation (except for denormalized numbers, which use excess 126) and 23 bits for the significand. It corresponds to C type `float`. The double-precision format has an 11-bit cell for the exponent using excess 1023 representation (except for denormalized numbers, which use excess 1022) and a 52-bit cell for the significand. It corresponds to C type double.

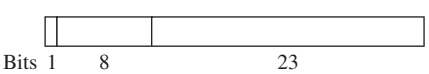
The single-precision format has the following bit values. Positive infinity is

0 1111 1111 000 0000 0000 0000 0000 0000

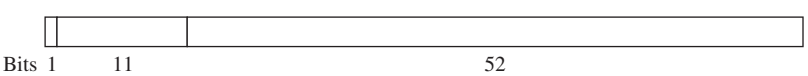
The hexadecimal abbreviation for the full 32-bit pattern arranges the bits into groups of four as

0111 1111 1000 0000 0000 0000 0000 0000

FIGURE 3.39
The IEEE 754 floating-point standard.



(a) Single precision.



(b) Double precision.

which is written 7F80 0000 (hex). The largest positive value is

0 1111 1110 111 1111 1111 1111 1111 1111

or 7F7F FFFF (hex). It is exactly $2^{128} - 2^{104}$, which is approximately 2^{128} or 3.4×10^{38}. The smallest positive normalized number is

0 0000 0001 000 0000 0000 0000 0000 0000

or 0080 0000 (hex). It is exactly 2^{-126}, which is approximately 1.2×10^{-38}. The smallest positive denormalized number is

0 0000 0000 000 0000 0000 0000 0000 0001

or 0000 0001 (hex). It is exactly 2^{-149}, which is approximately 1.4×10^{-45}.

Example 3.48 What is the hexadecimal representation of −47.25 in single-precision floating point? The integer 47 (dec) = 101111 (bin), and the fraction 0.25 (dec) = 0.01 (bin). So, 47.25 (dec) = 101111.01 = 1.0111101×2^5. The number is negative, so the first bit is 1. The exponent 5 is converted to excess 127 by adding 5 + 127 = 132 (dec) = 1000 0100 (excess 127). The significand stores the bits to the right of the binary point, 0111101. So, the bit pattern is

1 1000 0100 011 1101 0000 0000 0000 0000

which is C23D 0000 (hex). ∎

Example 3.49 What is the number, as written in binary scientific notation, whose hexadecimal representation is 3CC8 0000? The bit pattern is

0 0111 1001 100 1000 0000 0000 0000 0000

The sign bit is zero, so the number is positive. The exponent is 0111 1001 (excess 127) = 121 (unsigned) = 121 − 127 = −6 (dec). From the significand, the bits to the right of the binary point are 1001. The hidden bit is 1, so the number is 1.1001×2^{-6}. ∎

Example 3.50 What is the number, as written in binary scientific notation, whose hexadecimal representation is 0050 0000? The bit pattern is

0 0000 0000 101 0000 0000 0000 0000 0000

The sign bit is 0, so the number is positive. The exponent field is all 0's, so the number is denormalized. The exponent is 0000 0000 (excess 126) = 0 (unsigned) = 0 − 126 = −126 (dec). The hidden bit is 0 instead of 1, so the number is 0.101×2^{-126}. ∎

The double-precision format has both wider range and greater precision because of the larger exponent and significand fields. The largest double value is approximately 2^{1023}, or 1.8×10^{308}. The smallest positive normalized number is approximately 2.2×10^{-308}, and the smallest denormalized number is approximately 4.9×10^{-324}.

Figure 3.37, which shows the denormalized special values, applies to IEEE 754 values with a few modifications. For single precision, the exponent field has eight bits. Thus, the three sequences in the top figure correspond to multipliers of 2^{-127}, 2^{-126}, and 2^{-125}. Because the significand has 23 bits, each of the three sequences has $2^{23} = 8{,}388{,}608$ values instead of 16 values. It is still the case that the spacing between successive values in each sequence is double the spacing between successive values in the preceding sequence.

For double precision, the exponent field has 11 bits. So, the three sequences in the top figure correspond to multipliers of 2^{-1023}, 2^{-1022}, and 2^{-1021}. Because the significand has 52 bits, each of the three sequences has $2^{52} = 4{,}503{,}599{,}627{,}370{,}496$ values instead of 16 values. With denormalization, each of the 4.5 quadrillion values in the left group are spread out evenly as they approach +0.0 from the right.

3.6 Models

A model is a simplified representation of some physical system. Workers in every scientific discipline, including computer science, construct models and investigate their properties. Consider some models of the solar system that astronomers have constructed and investigated.

Aristotle, who lived in Greece about 350 BC, proposed a model in which the earth was at the center of the universe. Surrounding the earth were 55 celestial spheres. The sun, moon, planets, and stars were each carried around the heavens on one of these spheres.

How well did this model match reality? It was successful in explaining the appearance of the sky, which looks like the top half of a sphere. It was also successful in explaining the approximate motion of the planets. Aristotle's model was accepted as accurate for hundreds of years.

Then in 1543 the Polish astronomer Copernicus published *De Revolutionibus*. In it he modeled the solar system with the sun at the center. The planets revolved around the sun in circles. This model was a better approximation to the physical system than the earth-centered model.

In the latter part of the 16th century, the Danish astronomer Tycho Brahe made a series of precise astronomical observations that showed a discrepancy in Copernicus's model. Then in 1609 Johannes Kepler proposed a model in which the earth and all the planets revolved around the sun not in circles, but

in flattened circles called *ellipses*. This model was successful in explaining in detail the intricate motion of the planets as observed by Tycho Brahe.

Each of these models is a simplified representation of the solar system. None of the models is a completely accurate description of the real physical world. We know now, in light of Einstein's theories of relativity, that even Kepler's model is an approximation. No model is perfect. Every model is an approximation to the real world.

Models as approximations of reality

When information is represented in a computer's memory, that representation is only a model as well. Just as each model of the solar system describes some aspects of the underlying real system more accurately than other aspects, so does a representation scheme describe some property of the information more accurately than other properties.

For example, one property of positive integers is that there is an infinite number of them. No matter how large an integer you write down, someone else can always write down a larger integer. The unsigned binary representation in a computer does not describe that property very accurately. There is a limit to the size of the integer when stored in memory.

You may be aware that

$$\sqrt{2} = 1.4142135\ldots$$

The digits go on forever, never repeating. The representation scheme for storing real numbers is a model that only approximates numbers such as the square root of 2. It cannot represent the square root of 2 exactly. Any time a computer solves a problem, approximations are always involved because of limitations in the models.

Modeling the computer itself

All sorts of physical systems are commonly modeled with computers—inventories, molecules, accounting systems, and biological population systems, to name a few. In computer science, it is often the computer itself that is modeled.

The only physically real part of the computer is at Level LG1. Ultimately, a computer is just a complicated, organized mass of circuits and electrical signals. At Level ISA3, the high signals are modeled as 1's and the low signals as 0's. The programmer at Level ISA3 does not need to know anything about electrical circuits and signals to work with his model. Remember that at Level ISA3, the 1's and 0's represent the word `Tom` as

```
101 0100
110 1111
110 1101
```

The programmer at Level HOL6 does not need to know anything about bits to work with his model. In fact, programming the computer at any level requires only a knowledge of the model of the computer at that level.

A programmer at Level HOL6 can model the computer as a C machine. This model accepts C programs and uses them to process data. When the programmer instructs the machine to

```
printf("Tom");
```

he need not be concerned with how the computer is modeled as a binary machine at Level ISA3. Similarly, when a programmer at Level ISA3 writes a sequence of bits, he need not be concerned with how the computer is modeled as a combination of circuits at Level LG1.

This modeling of computer systems at successively higher levels is an idea that is not unique to computer science. Consider a large corporation with six divisions throughout the country. The president's model of the corporation is six divisions, with a vice president of each division reporting to him. He views the overall performance of the company in terms of the performance of each of the divisions. When he tells the vice president of the Widget Division to increase earnings, he does not need to be concerned with the vice president's model of the Widget Division. Likewise, when the vice president goes to each department manager within the Widget Division with an order, she does not need to be concerned with the department manager's model of his department. To have the president himself deal with the organization at the department level would be just about impossible. There are simply too many details at the department level of the entire corporation for one person to manage.

The computer user at Level App7 is like the president. He gives an instruction such as "compute the grade point average of all the sophomores" to a program originally written by a programmer at Level HOL6. He need not be concerned with the Level HOL6 model to issue the instruction. Eventually this command at Level App7 is transformed through successively lower levels to Level LG1. The end result is that the user at Level App7 can control the mass of electrical circuitry and signals with a very simplified model of the computer.

Chapter Summary

A binary quantity is restricted to one of two values. At the machine level, computers store information in binary. A bit is a binary digit whose value can be either 0 or 1. Nonnegative integers use unsigned binary representation. The rightmost bit is in the 1's place, the next bit to the left is in the 2's place, the next bit to the left is in the 4's place, and so on, with each place value double the preceding place value. Signed integers

use two's complement binary representation in which the first bit is the sign bit and the remaining bits determine the magnitude. For positive numbers, the two's complement representation is identical to the unsigned representation. For negative numbers, however, the two's complement of a number is obtained by taking 1 plus the ones' complement of the corresponding positive number.

Every binary integer, signed or unsigned, has a range that is determined by the number of bits in the memory cell. The smaller the number of bits in the cell, the more limited the range. The carry bit, C, is used to flag an out-of-range condition for an unsigned integer, and the overflow bit, V, is used to flag an out-of-range condition for an integer in two's complement representation. Operations on binary integers include ADD, AND, OR, XOR, and NOT. ASL, which stands for *arithmetic shift left*, multiplies a binary value by 2, and ASR, which stands for *arithmetic shift right*, divides a binary value by 2.

The hexadecimal number system, which is a base 16 system, provides a compact notation for expressing bit patterns. The 16 hexadecimal digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. One hexadecimal digit represents four bits. The American Standard Code for Information Interchange, abbreviated ASCII, is a common code for storing characters. It is a seven-bit code with 128 characters, including the uppercase and lowercase letters of the English alphabet, the decimal digits, punctuation marks, and nonprintable control characters. The Unicode character set extends ASCII to cover all the world's languages.

A floating point number is stored in a cell with three fields—a one-bit sign field, a field for the exponent, and a field for the significand. Except for special values, numbers are stored in binary scientific notation with a hidden bit to the left of the binary point that is assumed to be 1. The exponent is stored in an excess representation. Four special values are zero, infinity, NaN, and denormalized numbers. The IEEE 754 standard defines the number of bits in the exponent and significand fields to be 8 and 23 for single precision, and 11 and 52 for double precision.

A basic problem at all levels of abstraction is the mismatch between the form of the information to be processed and the language to represent it. A program in machine language processes bits. A program in a high-order language processes items such as arrays and records. Regardless of the level in which the program is written, the information must be cast into a format that the language can recognize. Matching the information to the language is a basic problem at all levels of abstraction and is a source of approximation in the modeling process of problem solving.

Exercises

Section 3.1

*1. Count the next 10 numbers **(a)** in octal starting from 267, **(b)** in base 3 starting from 2102, **(c)** in binary starting from 10101, and **(d)** in base 5 starting from 2433.

2. Count the next 10 numbers **(a)** in octal starting from 466, **(b)** in base 3 starting from 1201, **(c)** in binary starting from 11011, and **(d)** in base 5 starting from 3434.

*3. Convert the following numbers from binary to decimal, assuming unsigned binary representation:
(a) 10010 **(b)** 110 **(c)** 1011
(d) 1000 **(e)** 11111 **(f)** 1010101

4. Convert the following numbers from binary to decimal, assuming unsigned binary representation:
(a) 10110 **(b)** 10 **(c)** 10101
(d) 10000 **(e)** 1111 **(f)** 11110000

*5. Convert the following numbers from decimal to binary, assuming unsigned binary representation:
(a) 25 **(b)** 16 **(c)** 1 **(d)** 14 **(e)** 5 **(f)** 41

6. Convert the following numbers from decimal to binary, assuming unsigned binary representation:
(a) 12 **(b)** 35 **(c)** 3 **(d)** 0 **(e)** 27 **(f)** 16

7. With unsigned binary representation, what is the range of numbers as written in binary and in decimal for the following cells?
*(a) a two-bit cell *(b) a three-bit cell **(c)** a four-bit cell
(d) a five-bit cell **(e)** an *n*-bit cell in general

*8. Perform the following additions on unsigned integers, assuming a seven-bit cell. Show the effect on the carry bit:

(a)
```
        010 1011
ADD     100 1001
C =
```

(b)
```
        101 1001
ADD     011 0111
C =
```

(c)
```
        111 1111
ADD     111 1111
C =
```

(d)
```
        111 1111
ADD     000 0001
C =
```

9. Perform the following additions on unsigned integers, assuming a nine-bit cell. Show the effect on the carry bit:

(a)
```
        0 0100 1011
ADD     0 1101 0001
C =
```

(b)
```
        1 0001 1101
ADD     0 1110 1000
C =
```

(c)

	1 1111 1111
ADD	0 0000 0001
C =	

(d)

	1 1111 1111
ADD	1 1111 1111
C =	

10. Section 3.1 states that you can tell whether a binary number is even or odd only by inspecting the digit in the 1's place. Is that always possible for an arbitrary base? Explain.

11. Converting between octal and decimal is analogous to the technique of converting between binary and decimal. ***(a)** Write the polynomial representation of the octal number 70146 as in Figure 3.4. **(b)** Use the technique of Figure 3.5 to convert 7291 (dec) to octal.

12. Why do programmers at Level ISA3 confuse Halloween and Christmas? Hint: What does 31 (oct) equal?

Section 3.2

***13.** Convert the following numbers from decimal to binary, assuming seven-bit two's complement binary representation:

(a) 49 **(b)** −27 **(c)** 0
(d) −64 **(e)** −1 **(f)** −2
(g) What is the range for this computer as written in binary and in decimal?

14. Convert the following numbers from decimal to binary, assuming nine-bit two's complement binary representation:

(a) 51 **(b)** −29 **(c)** −2
(d) 0 **(e)** −256 **(f)** −1
(g) What is the range for this cell as written in binary and in decimal?

***15.** Convert the following numbers from binary to decimal, assuming seven-bit two's complement binary representation:

(a) 001 1101 **(b)** 101 0101 **(c)** 111 1100
(d) 000 0001 **(e)** 100 0000 **(f)** 100 0001

16. Convert the following numbers from binary to decimal, assuming nine-bit two's complement binary representation:

(a) 0 0001 1010 **(b)** 1 0110 1010 **(c)** 1 1111 1100
(d) 0 0000 0001 **(e)** 1 0000 0000 **(f)** 1 0000 0001

***17.** Perform the following additions, assuming seven-bit two's complement binary representation. Show the effect on the status bits:

(a)

	010 1011
ADD	000 1110
N =	
Z =	
V =	
C =	

(b)

	111 1001
ADD	000 1101
N =	
Z =	
V =	
C =	

(c)
	100 0110
ADD	101 0101
N =	
Z =	
V =	
C =	

(d)
	110 0001
ADD	111 0101
N =	
Z =	
V =	
C =	

(e)
	000 1101
ADD	011 0100
N =	
Z =	
V =	
C =	

(f)
	100 1001
ADD	010 1011
N =	
Z =	
V =	
C =	

18. Perform the following additions, assuming nine-bit two's complement binary representation. Show the effect on the status bits:

(a)
	0 1010 1100
ADD	0 0011 1010
N =	
Z =	
V =	
C =	

(b)
	1 1110 0101
ADD	0 0011 0101
N =	
Z =	
V =	
C =	

(c)
	1 0001 1011
ADD	1 0101 0100
N =	
Z =	
V =	
C =	

(d)
	1 1000 0101
ADD	1 1101 0110
N =	
Z =	
V =	
C =	

(e)
	0 0011 0100
ADD	0 1101 0010
N =	
Z =	
V =	
C =	

(f)
	1 0010 0111
ADD	0 1010 0111
N =	
Z =	
V =	
C =	

19. With two's complement binary representation, what is the range of numbers as written in binary and in decimal notation for the following cells?

*(a) a two-bit cell *(b) a three-bit cell **(c)** a four-bit cell
(d) a five-bit cell **(e)** an *n*-bit cell in general

Section 3.3

*20. Perform the following logical operations, assuming a seven-bit cell:

(a)
```
        010 1100
AND     110 1010
N =
Z =
```

(b)
```
        000 1111
AND     101 0101
N =
Z =
```

(c)
```
        010 1100
OR      110 1010
N =
Z =
```

(d)
```
        000 1111
OR      101 0101
N =
Z =
```

(e)
```
        010 1100
XOR     110 1010
N =
Z =
```

(f)
```
        000 1111
XOR     101 0101
N =
Z =
```

(g) NEG 010 1100

(h) NOT 110 1010

21. Perform the following logical operations, assuming a nine-bit cell:

(a)
```
        0 1001 0011
AND     1 0111 0101
N =
Z =
```

(b)
```
        0 0000 1111
AND     1 0111 0101
N =
Z =
```

(c)
```
        0 1001 0011
OR      1 0111 0101
N =
Z =
```

(d)
```
        0 0000 1111
OR      1 0111 0101
N =
Z =
```

(e)
```
        0 1001 0011
XOR     1 0111 0101
N =
Z =
```

(f)
```
        0 0000 1111
XOR     1 0111 0101
N =
Z =
```

(g) NEG 1 1001 0011

(h) NOT 1 0111 0101

*22. Assuming seven-bit two's complement representation, convert each of the following decimal numbers to binary, show the effect of the ASL operation on it, and then convert the result back to decimal. Repeat with the ASR operation:

(a) 24 **(b)** 37 **(c)** −26

(d) 1 **(e)** 0 **(f)** −1

23. Assuming nine-bit two's complement representation, convert each of the following decimal numbers to binary, show the effect of the ASL

operation on it, and then convert the result back to decimal. Repeat with the ASR operation:

(a) 94 **(b)** 135 **(c)** −62
(d) 1 **(e)** 0 **(f)** −1

24. **(a)** Write the RTL specification for an arithmetic shift right on an eight-bit cell. **(b)** Write the RTL specification for an arithmetic shift left on an eight-bit cell.

***25.** Assuming a seven-bit cell, show the effect of the rotate operation on each of the following values with the given initial value of C:

(a) C = 1, ROL 010 1101 **(b)** C = 0, ROL 010 1101
(c) C = 1, ROR 010 1101 **(d)** C = 0, ROR 010 1101

26. Assuming a nine-bit cell, show the effect of the rotate operation on each of the following values with the given initial value of C:

(a) C = 1, ROL 0 0110 1101 **(b)** C = 0, ROL 0 0110 1101
(c) C = 1, ROR 0 0110 1101 **(d)** C = 0, ROR 0 0110 1101

27. **(a)** Write the RTL specification for a rotate right on an eight-bit cell. **(b)** Write the RTL specification for a rotate left on an eight-bit cell.

Section 3.4

28. Count the next five numbers in hexadecimal, starting with the following:

***(a)** 3AB7 **(b)** 6FD **(c)** B9E

29. Convert the following numbers from hexadecimal to decimal:

***(a)** 2D5E **(b)** 2F **(c)** 7

30. This chapter mentions the method of converting from decimal to hexadecimal but gives no examples. Use the method to convert the following decimal numbers to hexadecimal:

***(a)** 26,831 **(b)** 4096 **(c)** 9

31. The technique for converting from decimal to any base will work, with some modification, for bases other than binary. **(a)** Explain the method to convert from decimal to octal. **(b)** Explain the method to convert from decimal to base *n* in general.

***32.** Assuming seven-bit two's complement binary representation, convert the following numbers from hexadecimal to decimal. Remember to check the sign bit:

(a) 5D **(b)** 2F **(c)** 40

33. Assuming nine-bit two's complement binary representation, convert the following numbers from hexadecimal to decimal. Remember to check the sign bit:

(a) 1B4 **(b)** 0F5 **(c)** 100

*34. Assuming seven-bit two's complement binary representation, write the bit patterns for the following decimal numbers in hexadecimal:
(a) −27 **(b)** 63 **(c)** −1

35. Assuming nine-bit two's complement binary representation, write the bit patterns for the following decimal numbers in hexadecimal:
(a) −73 **(b)** −1 **(c)** 94

*36. Decode the following secret ASCII message (reading across):

```
100 1000    110 0001    111 0110    110 0101
010 0000    110 0001    010 0000    110 1110
110 1001    110 0011    110 0101    010 0000
110 0100    110 0001    111 1001    010 0001
```

37. Decode the following secret ASCII message (reading across):

```
100 1101    110 0101    110 0101    111 0100
010 0000    110 0001    111 0100    010 0000
110 1101    110 1001    110 0100    110 1110
110 1001    110 0111    110 1000    111 0100
010 1110
```

*38. How is the following string of nine characters stored in ASCII?

```
Pay $0.92
```

39. How is the following string of 13 characters stored in ASCII?

```
(321)497-0015
```

40. Convert the following Unicode code points to UTF-8:
*(a) U+0542, Armenian Ղ. **(b)** U+2873, Braille Pattern ⡳
(c) U+4EB6, CJK Unified 亶 **(d)** U+13007, Egyptian Hieroglyphics 𓀇

41. Decode the following UTF-8 words:
(a) 56 6F 69 C3 A0 **(b)** 4B C3 A4 73 65 **(c)** 70 C3 A2 74 65

42. You are the chief communications officer for the Lower Slobovian army at war with the Upper Slobovians. Your spies will infiltrate the enemy's command headquarters in an attempt to gain the "upper" hand. You know the Uppers are planning a major assault, and you also know the following: (1) It will be at either sunrise or sunset, (2) it will come by land, air, or sea, and (3) it will occur on March 28, 29, 30, or 31, or on April 1. Your spies must communicate with you in binary. Devise a suitable binary code for transmitting the information. Try to use the fewest number of bits possible.

43. Octal numbers are sometimes used instead of hexadecimal numbers to represent bit sequences.
*(a) How many bits does one octal number represent?

How would you represent the decimal number −13 in octal with the following cells?

(b) A 15-bit cell **(c)** A 16-bit cell **(d)** An 8-bit cell

Section 3.5

*44. Convert the following numbers from binary to decimal:
(a) 110.101001 **(b)** 0.000011 **(c)** 1.0

45. Convert the following numbers from binary to decimal:
(a) 101.101001 **(b)** 0.000101 **(c)** 1.0

*46. Convert the following numbers from decimal to binary:
(a) 13.15625 **(b)** 0.0390625 **(c)** 0.6

47. Convert the following numbers from decimal to binary:
(a) 12.28125 **(b)** 0.0234375 **(c)** 0.7

48. Construct a table similar to Figure 3.33 that compares all the values with a four-bit cell for excess 7 and two's complement representation.

49. **(a)** With excess 7 representation, what is the range of numbers as written in binary and in decimal for a four-bit cell? **(b)** With excess 15 representation, what is the range of numbers as written in binary and in decimal for a five-bit cell? **(c)** With excess $2^{n-1} - 1$ representation, what is the range of numbers as written in binary and in decimal for an n-bit cell in general?

50. Assuming a three-bit exponent field and a four-bit significand, write the bit pattern for the following decimal values:
*(a) −12.5 **(b)** 13.0 **(c)** 0.43 **(d)** 0.1015625

51. Assuming a three-bit exponent field and a four-bit significand, what decimal values are represented by the following bit patterns?
*(a) 0 010 1101 **(b)** 1 101 0110 **(c)** 1 111 1001
(d) 0 001 0011 **(e)** 1 000 0100 **(f)** 0 111 0000

52. For IEEE 754 single-precision floating point, write the hexadecimal representation for the following decimal values:
(a) 27.1015625 **(b)** −1.0 **(c)** −0.0
(d) 0.5 **(e)** 0.6 **(f)** 256.015625

53. For IEEE 754 single-precision floating point, what is the number, as written in binary scientific notation, whose hexadecimal representation is the following?
*(a) 4280 0000 **(b)** B350 0000 **(c)** 0061 0000
(d) FF80 0000 **(e)** 7FE4 0000 **(f)** 8000 0000

54. For IEEE 754 single-precision floating point, write the hexadecimal representation for
- **(a)** positive zero
- **(b)** the smallest positive denormalized number
- **(c)** the largest positive denormalized number
- **(d)** the smallest positive normalized number
- **(e)** 1.0
- **(f)** the largest positive normalized number
- **(g)** positive infinity

55. For IEEE 754 double-precision floating point, write the hexadecimal representation for
- **(a)** positive zero
- **(b)** the smallest positive denormalized number
- **(c)** the largest positive denormalized number
- **(d)** the smallest positive normalized number
- **(e)** 1.0
- **(f)** the largest positive normalized number
- **(g)** positive infinity

Problems

Section 3.1

56. Write a program in C that takes as input a 4-digit octal number and prints the next 10 octal numbers. Define an octal number as

```
int octNum[4];
```

Use `octNum[0]` to store the most significant (i.e., leftmost) octal digit and `octNum[3]` to store the least significant octal digit. Test your program with interactive input.

57. Write a program in C that takes as input an 8-bit binary number and prints the next 10 binary numbers. Define a binary number as

```
int binNum[8];
```

Use `binNum[0]` to store the most significant (i.e., leftmost) bit and `binNum[7]` to store the least significant bit. Ask the user to input the first binary number with each bit separated by at least one space.

58. Defining a binary number as in Problem 57, write the function

```
int binToDec(const int bin[])
```

to convert an eight-bit unsigned binary number to a nonnegative decimal integer. Do not output the decimal integer in the function. Test your function with interactive input.

59. Defining a binary number as in Problem 57, write the function

```
void decToBin(int bin[], int dec)
```

to convert a nonnegative decimal integer to an eight-bit unsigned binary number. Do not output the binary number in the function. Test your function with interactive input.

60. Defining `sum`, `bin1`, and `bin2` as binary numbers as in Problem 57, write the void function

```
void binaryAdd(int sum[], int *cBit,
               const int bin1[], const int bin2[])
```

to compute `sum` as the sum of the two binary numbers, `bin1` and `bin2`. `cBit` should be the value of the carry bit after the addition. Do not output the carry bit or the sum in the function. Test your void function with interactive input.

Section 3.2

61. Defining a binary number as in Problem 57, write the function

```
int binToDec (const int bin[])
```

to convert an eight-bit two's complement binary number to a signed decimal integer. Do not output the decimal integer in the function. Test your function with interactive input.

62. Defining a binary number as in Problem 57, write the function

```
void decToBin(int bin[], int dec)
```

to convert a signed decimal integer to an eight-bit two's complement binary number. Do not output the binary number in the function. Test your function with interactive input.

Section 3.3

63. Defining `bAnd`, `bin1`, and `bin2` as binary numbers as in Problem 57, write the void function

```
void binaryAnd(int bAnd[],
               const int bin1[], const int bin2[])
```

to compute `bAnd` as the AND of the two binary numbers `bin1` and `bin2`. Do not output the binary number in the function. Test your function with interactive input.

64. Write the void function for Problem 63 renamed as `binaryOr()` using the OR operation.

65. Defining a binary number as in Problem 57, write the function

```
void shiftLeft(int binNum[], int *cBit)
```

to perform an arithmetic shift left on `binNum`. `cBit` should be the value of the carry bit after the shift. Do not output the shifted number or the carry bit in the function. Test your function with interactive input.

66. Write the function for Problem 65 renamed `shiftRight()`, using the arithmetic shift right operation.

Section 3.4

67. Write a program in C that takes as input a four-digit hexadecimal number and prints the next 10 hexadecimal numbers. Define a hexadecimal number as

```
int hexNum[4]
```

Allow upper- or lowercase letters for input and use uppercase letters for the hexadecimal output. For example, 3C6f should be valid input and should produce output 3C6F, 3C70, 3C71, ...

68. Defining a hexadecimal number as in Problem 67, write the function

```
int hexToDec(const int hexNum[])
```

to convert a four-digit hexadecimal number to a nonnegative decimal integer. Do not output the decimal integer in the function. Test your function with interactive input.

69. Defining a hexadecimal number as in Problem 67, write the function

```
void decToHex(int hexNum[], int decNum)
```

to convert a nonnegative decimal integer to a four-digit hexadecimal number. Do not output the hexadecimal number in the function. Test your function with interactive input.

70. Defining a hexadecimal number as in Problem 67, write the function

```
int hexToDec(const int hexNum[])
```

to convert a four-digit hexadecimal number to a signed decimal integer. Do not output the decimal integer in the function. Test your function with interactive input.

71. Defining a hexadecimal number as in Problem 67, write the function

```
void decToHex(int hex[], int dec)
```

to convert a signed decimal integer to a four-digit hexadecimal number. Do not output the hexadecimal number in the function. Test your function with interactive input.

72. Write a program in C to convert an unsigned number in an arbitrary base to a nonnegative decimal integer. For four-digit base 6 numbers, for example, declare

```
const int base = 6;
const int numDigits = 4;
int number[numDigits];
```

Write the function

```
void getNumber(int num[])
```

to input the unsigned number in the arbitrary base. Use the uppercase letters of the alphabet for input if required by the value of `base`. Write the function

```
int baseToDec(const int num[])
```

to convert the number in the arbitrary base to a nonnegative decimal value. You must be able to modify your program for operation with a different base by changing only the constant `base`. You must be able to modify the program for a different number of digits by changing only the constant `numDigits`.

73. Using the declarations as in Problem 72, write the function

```
void decToBase(int baseNum[], int decNum)
```

to convert the nonnegative decimal integer to the unsigned number in the arbitrary base. Write the function

```
void putNumber(const int baseNum[])
```

to output the unsigned number in the arbitrary base. Use the uppercase letters of the alphabet for output if required by the value of `base`. You must be able to modify your program for operation with a different base by changing only the constant base. You must be able to modify the program for a different number of digits by changing only the constant `numDigits`.