plex because there are fewer cases to learn. Let us first review decimal addition:

$$\begin{array}{r} 3\ 7\ 6 \\ +4\ 6\ 1 \\ \hline 8\ 3\ 7 \end{array} \quad \text{LSD}$$

The least-significant-digit (LSD) position is operated on first, producing a sum of 7. The digits in the second position are then added to produce a sum of 13, which produces a **carry** of 1 into the third position. This produces a sum of 8 in the third position.

The same general steps are followed in binary addition. However, only four cases can occur in adding the two binary digits (bits) in any position. They are:

$$\begin{aligned} 0 + 0 &= 0 \\ 1 + 0 &= 1 \\ 1 + 1 &= 10 = 0 + \text{carry of 1 into next position} \\ 1 + 1 + 1 &= 11 = 1 + \text{carry of 1 into next position} \end{aligned}$$

The last case occurs when the two bits in a certain position are 1 and there is a carry from the previous position. Here are several examples of the addition of two binary numbers (decimal equivalents are in parentheses):

011 (3)	1001 (9)	11.011 (3.375)
+ 110 (6)	+ 1111 (15)	+ 10.110 (2.750)
1001 (9)	11000 (24)	110.001 (6.125)

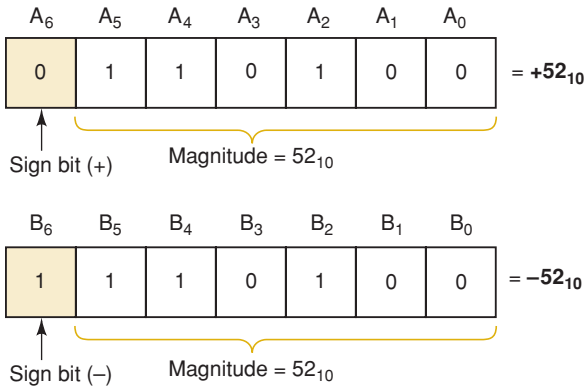
Addition is the most important arithmetic operation in digital systems. As we shall see, the operations of subtraction, multiplication, and division as they are performed in most modern digital computers and calculators actually use only addition as their basic operation.

REPRESENTING SIGNED NUMBERS

The sign bit is used to indicate the positive or negative nature of the stored binary number. The numbers in Figure 6-1 consist of a sign bit and six magnitude bits. The magnitude bits are the true binary equivalent of the decimal value being represented. This is called the **sign-magnitude system** for representing signed binary numbers.

Although the sign-magnitude system is straightforward, calculators and computers do not normally use it because the circuit implementation is more complex than in other systems. The most commonly used system for representing signed binary numbers is the **2's-complement system**. Before we see how this is done, we must first see how to form the 1's complement and 2's complement of a binary number.

FIGURE 6-1
Representation of signed numbers in sign-magnitude form.



1's-Complement Form

The 1's complement of a binary number is obtained by changing each 0 to a 1 and each 1 to a 0. In other words, change each bit in the number to its complement. The process is shown below.

```
1 0 1 1 0 1  original binary number
↓ ↓ ↓ ↓ ↓ ↓
```

Thus, we say that the 1's complement of 101101 is 010010.

2's Complement Form

The 2's complement of a binary number is formed by taking the 1's complement of the number and adding 1 to the least-significant-bit position. The process is illustrated below for $101101_2 = 45_{10}$.

```
  1 0 1 1 0 1    binary equivalent of 45
  0 1 0 0 1 0    complement each bit to form 1's complement
+         1      add 1 to form 2's complement
  0 1 0 0 1 1    2's complement of original binary number
```

Thus, we say that 010011 is the 2's complement representation of 101101.

Here's another example of converting a binary number to its 2's-complement representation:

```
    1 0 1 1 0 0    original binary number
    0 1 0 0 1 1    1's complement
+           1      add 1
    0 1 0 1 0 0    2's complement of original number
```

The 2's-complement system is used to represent signed numbers because, as we shall see, it allows us to perform the operation of subtraction by actually performing addition. This is significant because it means that a digital computer can use the same circuitry both to add and to subtract, thereby realizing a saving in hardware.

EXAMPLE 6-1

Represent each of the following signed decimal numbers as a signed binary number in the 2's-complement system. Use a total of five bits, including the sign bit.

(a) +13 (b) −9

Solution

(a) The number is positive, so the magnitude (13) will be represented in its true-magnitude form, that is, $13 = 1101_2$. Attaching the sign bit of 0, we have

$$\underset{\text{sign bit}\uparrow}{\mathbf{+13 = 01101}}$$

(b) The number is negative, so the magnitude (9) must be represented in 2's-complement form:

$$\begin{array}{rll} 9_{10} = 1001_2 & & \\ \quad 0110 & (1\text{'s complement}) \\ + \quad 1 & (\text{add 1 to LSB}) \\ \hline \quad 0111 & (2\text{'s complement}) \end{array}$$

When we attach the sign bit of 1, the complete signed number becomes

$$\underset{\text{sign bit}\uparrow}{\mathbf{-9 = 10111}}$$

OR

$$\begin{array}{rll} +9 = 01001 & & \\ \quad 10110 & (1\text{'s complement of each bit including sign bit}) \\ + \quad 1 & (\text{add 1 to LSB}) \\ \hline -9 = 10111 & (2\text{'s-complement representation of } -9) \end{array}$$

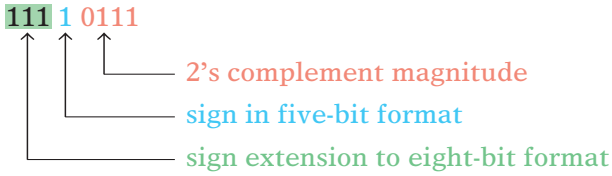
The result is, of course, the same as before.

Sign Extension

Example 6-1 required that we use a total of five bits to represent the signed numbers. The size of a register (*number of flip-flops*) determines the number of binary digits that are stored for each number. Most digital systems today store numbers in registers sized in even multiples of four bits. In other words, the storage registers will be made up of 4, 8, 12, 16, 32, or 64 bits. In a system that stores eight-bit numbers, seven bits represent the magnitude and the MSB represents the sign. If we need to store a positive five-bit number in an eight-bit register, it makes sense to simply add leading zeros. The MSB (sign bit) is still 0, indicating a positive value.

What happens when we try to store five-bit negative numbers in an eight-bit register? In the previous section we found that the five-bit, 2's-complement binary representation for −9 is 10111.

If we appended leading 0s, this would no longer be a negative number in eight-bit format. The proper way to extend a negative number is to append leading 1's. Thus, the value stored for negative 9 is



Special Case in 2's-Complement Representation

we can state that the complete range of values that can be represented in the 2's-complement system having N magnitude bits is

$$-2^N \text{ to } +(2^N - 1)$$

There are a total of 2^{N+1} different values, *including* zero.

For example, Table 6-1 lists all signed numbers that can be represented in

TABLE 6-1

Decimal Value	Signed Binary Using 2's Complement
$+7 = 2^3 - 1$	0111
+6	0110
+5	0101
+4	0100
+3	0011
+2	0010
+1	0001
0	0000
−1	1111
−2	1110
−3	1101
−4	1100
−5	1011
−6	1010
−7	1001
$-8 = -2^3$	1000

6-3 ADDITION IN THE 2's-COMPLEMENT SYSTEM

Case I: Two Positive Numbers. The addition of two positive numbers is straightforward. Consider the addition of +9 and +4:

```
+9 → | 0 | 1001    (augend)
+4 → | 0 | 0100    (addend)
     | 0 | 1101    (sum = +13)
       ↑
       └ sign bits
```

Note that the sign bits of the **augend** and the **addend** are both 0 and the sign bit of the sum is 0, indicating that the sum is positive. Also note that the augend and the addend are made to have the same number of bits. This must *always* be done in the 2's-complement system.

Case II: Positive Number and Smaller Negative Number. Consider the addition of +9 and −4. Remember that the −4 will be in its 2's-complement form. Thus, +4 (00100) must be converted to −4 (11100).

```
          ┌ sign bits
          ↓
+9 → | 0 | 1001    (augend)
−4 → | 1 | 1100    (addend)
   1 | 0 | 0101
   ↑
```

In this case, the sign bit of the addend is 1. Note that the sign bits also participate in the addition process. In fact, a carry is generated in the last position

of addition. *This carry is always disregarded,* so that the final sum is 00101, which is equivalent to +5.

Case III: Positive Number and Larger Negative Number. Consider the addition of −9 and +4:

```
−9 →    10111
+4 →    00100
        11011      (sum = −5)
        ↑
        └── negative sign bit
```

The sum here has a sign bit of 1, indicating a negative number. Because the sum is negative, it is in 2's-complement form, so that the last four bits, 1011, actually represent the 2's complement of the sum. To find the true magnitude of the sum, we must negate (2's-complement) 11011; the result is 00101 = +5. Thus, 11011 represents −5.

Case IV: Two Negative Numbers

```
−9 →    10111
−4 →    11100
      1 10011
      ↑ ↑
      │ └── sign bit
      └──── This carry is disregarded; the result is 10011 (sum = −13).
```

This final result is again negative and in 2's-complement form with a sign bit of 1. Negating (2's-complementing) this result produces 01101 = +13.

Case V: Equal and Opposite Numbers

```
−9 →    10111
+9 →    01001
      0 1 00000
        ↑
        └──── Disregard; the result is 00000 (sum = +0).
```

The result is obviously +0, as expected.

6-4 SUBTRACTION IN THE 2's-COMPLEMENT SYSTEM

The subtraction operation using the 2's-complement system actually involves the operation of addition and is really no different from the various cases for addition considered in Section 6-3. When subtracting one binary number (the **subtrahend**) from another binary number (the **minuend**), use the following procedure:

1. *Negate the subtrahend.* This will change the subtrahend to its equivalent value of opposite sign.
2. *Add this to the minuend.* The result of this addition will represent the *difference* between the subtrahend and the minuend.

Once again, as in all 2's-complement arithmetic operations, it is necessary that both numbers have the same number of bits in their representations.

Let us consider the case where +4 is to be subtracted from +9.

$$\begin{array}{rl} \text{minuend } (+9) \rightarrow & 01001 \\ \text{subtrahend } (+4) \rightarrow & 00100 \end{array}$$

Negate the subtrahend to produce 11100, which represents −4. Now add this to the minuend.

$$\begin{array}{rll} & 01001 & (+9) \\ + & 11100 & (-4) \\ \hline \not{1} & 00101 & (+5) \end{array}$$

↑ Disregard, so the result is 00101 = +5.

When the subtrahend is changed to its 2's complement, it actually becomes −4, so that we are *adding* −4 and +9, which is the same as subtracting +4 from +9. This is the same as case II of Section 6-3. Any subtraction operation, then, actually becomes one of addition when the 2's-complement system is used. This feature of the 2's-complement system has made it the most widely used of the methods available because it allows addition and subtraction to be performed by the same circuitry.

Here's another example showing +9 subtracted from −4:

$$\begin{array}{rll} & 11100 & (-4) \\ - & 01001 & (+9) \end{array}$$

Negate the subtrahend (+9) to produce 10111 (−9) and add this to the minuend (−4).

$$\begin{array}{rll} & 11100 & (-4) \\ + & 10111 & (-9) \\ \hline \not{1} & 10011 & (-13) \end{array}$$

↑ Disregard

The reader should verify the results of using the above procedure for the following subtractions: (a) +9 − (−4); (b) −9 − (+4); (c) −9 − (−4); (d) +4 − (−4). Remember that when the result has a sign bit of 1, it is negative and in 2's-complement form.

Arithmetic Overflow

In each of the previous addition and subtraction examples, the numbers that were added consisted of a sign bit and four magnitude bits. The answers also consisted of a sign bit and four magnitude bits. Any carry into the sixth bit position was disregarded. In all of the cases considered, the

magnitude of the answer was small enough to fit into four bits. Let's look at the addition of +9 and +8.

$$\begin{array}{rcc} +9 \rightarrow & 0 & 1001 \\ +8 \rightarrow & 0 & 1000 \\ \hline & 1 & 0001 \end{array}$$

incorrect sign ↑ ↑ incorrect magnitude

The answer has a negative sign bit, which is obviously incorrect because we are adding two positive numbers. The answer should be +17, but the magnitude 17 requires more than four bits and therefore *overflows* into the sign-bit position. This **overflow** condition can occur only when two positive or two negative numbers are being added, and it always produces an incorrect result. Overflow can be detected by checking to see that the sign bit of the result is the same as the sign bits of the numbers being added.

Subtraction in the 2's-complement system is performed by negating the minuend and *adding* it to the subtrahend, so overflow can occur only when the minuend and subtrahend have different signs. For example, if we are subtracting −8 from +9, the −8 is negated to become +8 and is added to +9, just as shown above, and overflow produces an erroneous negative result because the magnitude is too large.

A computer will have a special circuit to detect any overflow condition when two numbers are added or subtracted. This detection circuit will signal the computer's control unit that overflow has occurred and the result is incorrect. We will examine such a circuit in an end-of-chapter problem.

Number Circles and Binary Arithmetic

The concept of signed arithmetic and overflow can be illustrated by taking the numbers from Table 6-1 and "bending" them into a number circle as shown in Figure 6-3. Notice that there are two ways to look at this circle. It can be thought of as a circle of unsigned numbers (as shown in the outer ring) with minimum value 0 and maximum 15, or as signed 2's-complement numbers (as shown in the inner ring) with maximum value 7 and minimum −8. To add using a number circle, simply start at the value of the augend and

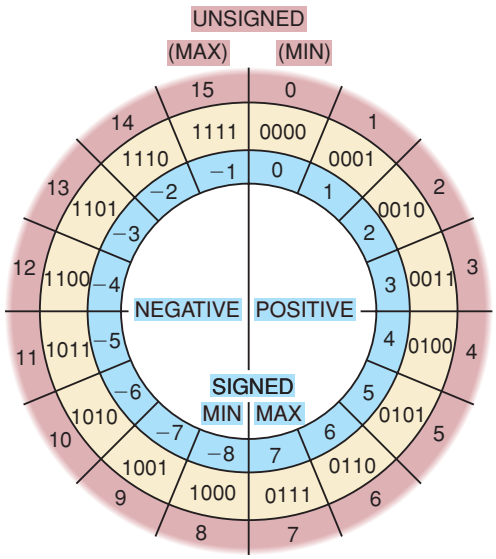


FIGURE 6-3 A four-bit number circle.

6-5 MULTIPLICATION OF BINARY NUMBERS

The multiplication of binary numbers is done in the same manner as the multiplication of decimal numbers. The process is actually simpler because the multiplier digits are either 0 or 1 and so we are always multiplying by 0 or 1 and no other digits. The following example illustrates for unsigned binary numbers:

```
   1001       ← multiplicand = 9₁₀
   1011       ← multiplier = 11₁₀
-------
   1001  ⎫
  1001   ⎬    partial products
 0000    ⎪
1001     ⎭
1100011  }    final product = 99₁₀
```

In this example the multiplicand and the multiplier are in true binary form and no sign bits are used.

6-6 BINARY DIVISION

The process for dividing one binary number (the *dividend*) by another (the *divisor*) is the same as that followed for decimal numbers, that which we usually refer to as "long division." The actual process is simpler in binary because

when we are checking to see how many times the divisor "goes into" the dividend, there are only two possibilities, 0 or 1. To illustrate, consider the following simple division examples:

```
     0011           0010
  11/1001      100/1010
     011            100
     0011            10
       11            10
```

In the first example, we have 1001_2 divided by 11_2, which is equivalent to $9 \div 3$ in decimal. The resulting quotient is $0011_2 = 3_{10}$. In the second example, 1010_2 is divided by 100_2, or $10 \div 4$ in decimal. The result is $0010.1_2 = 2.5_{10}$.

6-8 HEXADECIMAL ARITHMETIC

Hex numbers are used extensively in machine-language computer programming and in conjunction with computer memories (i.e., addresses). When working in these areas, you will encounter situations where hex numbers must be added or subtracted.

Hex Addition

Addition of hexadecimal numbers is done in much the same way as decimal addition, as long as you remember that the largest hex digit is F instead of 9. The following procedure is suggested:

1. Add the two hex digits in decimal, mentally inserting the decimal equivalent for those digits larger than 9.
2. If the sum is 15 or less, it can be directly expressed as a hex digit.
3. If the sum is greater than or equal to 16, subtract 16 and carry a 1 to the next digit position.

The following examples will illustrate the procedure.

EXAMPLE 6-6

Add the hex numbers 58 and 24.

Solution

$$\begin{array}{r} 58 \\ +24 \\ \hline 7C \end{array}$$

Adding the LSDs (8 and 4) produces 12, which is C in hex. There is no carry into the next digit position. Adding 5 and 2 produces 7.

EXAMPLE 6-7

Add the hex numbers 58 and 4B.

Solution

$$\begin{array}{r} 58 \\ +4B \\ \hline A3 \end{array}$$

Start by adding 8 and B, mentally substituting decimal 11 for B. This produces a sum of 19. Because 19 is greater than 16, subtract 16 to get 3; write down the 3 and carry a 1 into the next position. This carry is added to the 5 and 4 to produce a sum of 10_{10}, which is then converted to hexadecimal A.

EXAMPLE 6-8

Add 3AF to 23C.

Solution

$$\begin{array}{r} 3AF \\ +23C \\ \hline 5EB \end{array}$$

The sum of F and C is considered as $15 + 12 = 27_{10}$. Because this is greater than 16, subtract 16 to get 11_{10}, which is hexadecimal B, and carry a 1 into the second position. Add this carry to A and 3 to obtain E. There is no carry into the MSD position.