34. Show that the following logical equivalences hold for the Peirce arrow $\downarrow$, where $P \downarrow Q \equiv \sim(P \vee Q)$.

a. $\sim P \equiv P \downarrow P$

b. $P \vee Q \equiv (P \downarrow Q) \downarrow (P \downarrow Q)$

c. $P \wedge Q \equiv (P \downarrow P) \downarrow (Q \downarrow Q)$

***H* d.** Write $P \rightarrow Q$ using Peirce arrows only.

e. Write $P \leftrightarrow Q$ using Peirce arrows only.

ANSWERS FOR TEST YOURSELF

1. the output signal(s) that correspond to all possible combinations of input signals to the circuit **2.** a Boolean expression that represents the input signals as variables and indicates the successive actions of the logic gates on the input signals **3.** outputs a 1 for exactly one particular combination of input signals and outputs 0's for all other combinations **4.** they have the same input/output table **5.** NOT; AND **6.** NOT; OR

2.5 Application: Number Systems and Circuits for Addition

Counting in binary is just like counting in decimal if you are all thumbs. —Glaser and Way

In elementary school, you learned the meaning of decimal notation: that to interpret a string of decimal digits as a number, you mentally multiply each digit by its place value. For instance, 5,049 has a 5 in the thousands place, a 0 in the hundreds place, a 4 in the tens place, and a 9 in the ones place. Thus

$$5{,}049 = 5 \cdot (1{,}000) + 0 \cdot (100) + 4 \cdot (10) + 9 \cdot (1).$$

Using exponential notation, this equation can be rewritten as

$$5{,}049 = 5 \cdot 10^3 + 0 \cdot 10^2 + 4 \cdot 10^1 + 9 \cdot 10^0.$$

More generally, decimal notation is based on the fact that any positive integer can be written uniquely as a sum of products of the form

$$d \cdot 10^n,$$

where each n is a nonnegative integer and each d is one of the decimal digits 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9. The word *decimal* comes from the Latin root *deci*, meaning "ten." Decimal (or base 10) notation expresses a number as a string of digits in which each digit's position indicates the power of 10 by which it is multiplied. The right-most position is the ones place (or 10^0 place), to the left of that is the tens place (or 10^1 place), to the left of that is the hundreds place (or 10^2 place), and so forth, as illustrated below.

Place	10^3 thousands	10^2 hundreds	10^1 tens	10^0 ones
Decimal Digit	5	0	4	9

Binary Representation of Numbers

There is nothing sacred about the number 10; we use 10 as a base for our usual number system because we happen to have ten fingers. In fact, any integer greater than 1 can serve as a base for a number system. In computer science, **base 2 notation**, or **binary notation**, is of special importance because the signals used in modern electronics are always in one of only two states. (The Latin root *bi* means "two.")

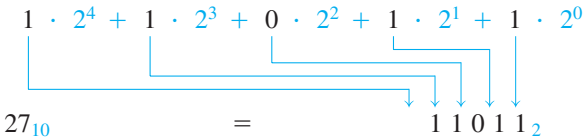
In Section 5.4, we show that any integer can be represented uniquely as a sum of products of the form

$$d \cdot 2^n,$$

where each n is an integer and each d is one of the binary digits (or bits) 0 or 1. For example,

$$\begin{aligned} 27 &= 16 + 8 + 2 + 1 \\ &= 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0. \end{aligned}$$

In binary notation, as in decimal notation, we write just the binary digits, and not the powers of the base. In binary notation, then,



where the subscripts indicate the base, whether 10 or 2, in which the number is written. The places in binary notation correspond to the various powers of 2. The right-most position is the ones place (or 2^0 place), to the left of that is the twos place (or 2^1 place), to the left of that is the fours place (or 2^2 place), and so forth, as illustrated below.

Place	2^4 sixteens	2^3 eights	2^2 fours	2^1 twos	2^0 ones
Binary Digit	1	1	0	1	1

As in the decimal notation, leading zeros may be added or dropped as desired. For example,

$$003_{10} = 3_{10} = 1 \cdot 2^1 + 1 \cdot 2^0 = 11_2 = 011_2.$$

Example 2.5.1 **Binary Notation for Integers from 1 to 9**

Derive the binary notation for the integers from 1 to 9.

Solution

$$\begin{aligned}
1_{10} &= 1 \cdot 2^0 = 1_2 \\
2_{10} &= 1 \cdot 2^1 + 0 \cdot 2^0 = 10_2 \\
3_{10} &= 1 \cdot 2^1 + 1 \cdot 2^0 = 11_2 \\
4_{10} &= 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 100_2 \\
5_{10} &= 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 101_2 \\
6_{10} &= 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 110_2 \\
7_{10} &= 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 111_2 \\
8_{10} &= 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 1000_2 \\
9_{10} &= 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 1001_2
\end{aligned}$$

■

A list of powers of 2 is useful for doing binary-to-decimal and decimal-to-binary conversions. See Table 2.5.1.

TABLE 2.5.1 Powers of 2

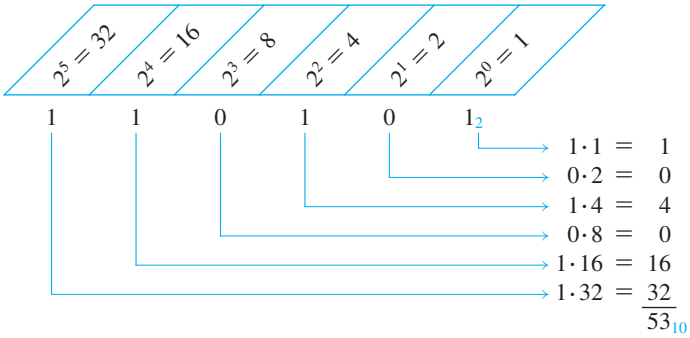
Power of 2	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Decimal Form	1024	512	256	128	64	32	16	8	4	2	1

Example 2.5.2 **Converting a Binary to a Decimal Number**

Represent 110101_2 in decimal notation.

Solution
$$\begin{aligned} 110101_2 &= 1\cdot 2^5 + 1\cdot 2^4 + 0\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 \\ &= 32 + 16 + 4 + 1 \\ &= 53_{10} \end{aligned}$$

Alternatively, the schema below may be used.



■

Example 2.5.3 **Converting a Decimal to a Binary Number**

Represent 209 in binary notation.

Solution Use Table 2.5.1 to write 209 as a sum of powers of 2, starting with the highest power of 2 that is less than 209 and continuing to lower powers.

Since 209 is between 128 and 256, the highest power of 2 that is less than 209 is 128. Hence

$$209_{10} = 128 + \text{a smaller number.}$$

Now $209 - 128 = 81$, and 81 is between 64 and 128, so the highest power of 2 that is less than 81 is 64. Hence

$$209_{10} = 128 + 64 + \text{a smaller number.}$$

Continuing in this way, you obtain

$$\begin{aligned} 209_{10} &= 128 + 64 + 16 + 1 \\ &= 1\cdot 2^7 + 1\cdot 2^6 + 0\cdot 2^5 + 1\cdot 2^4 + 0\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0. \end{aligned}$$

For each power of 2 that occurs in the sum, there is a 1 in the corresponding position of the binary number. For each power of 2 that is missing from the sum, there is a 0 in the corresponding position of the binary number. Thus

$$209_{10} = 11010001_2$$

■

Another procedure for converting from decimal to binary notation is discussed in Section 5.1.

Caution! Do not read 10_2 as "ten"; it is the number two. Read 10_2 as "one oh base two."

Binary Addition and Subtraction

The computational methods of binary arithmetic are analogous to those of decimal arithmetic. In binary arithmetic the number 2 (which equals 10_2 in binary notation) plays a role similar to that of the number 10 in decimal arithmetic.

Example 2.5.4 **Addition in Binary Notation**

Add 1101_2 and 111_2 using binary notation.

Solution Because $2_{10} = 10_2$ and $1_{10} = 1_2$, the translation of $1_{10} + 1_{10} = 2_{10}$ to binary notation is

$$\begin{array}{r} 1_2 \\ +\ 1_2 \\ \hline 10_2 \end{array}$$

It follows that adding two 1's together results in a carry of 1 when binary notation is used. Adding three 1's together also results in a carry of 1 since $3_{10} = 11_2$ ("one one base two").

$$\begin{array}{r} 1_2 \\ +\ 1_2 \\ +\ 1_2 \\ \hline 11_2 \end{array}$$

Thus the addition can be performed as follows:

$$\begin{array}{rcccc} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & \\ & 1 & 1 & 0 & 1_2 \\ + & & 1 & 1 & 1_2 \\ \hline 1 & 0 & 1 & 0 & 0_2 \end{array} \quad \leftarrow \text{carry row}$$

■

Example 2.5.5 **Subtraction in Binary Notation**

Subtract 1011_2 from 11000_2 using binary notation.

Solution In decimal subtraction the fact that $10_{10} - 1_{10} = 9_{10}$ is used to borrow across several columns. For example, consider the following:

$$\begin{array}{rcccc} & {\scriptstyle 9} & {\scriptstyle 9} & & \\ & & {\scriptstyle 1} & {\scriptstyle 1} & \leftarrow \text{borrow row} \\ & 1 & \not{0} & 0 & 0_{10} \\ - & & 5 & 8_{10} & \\ \hline & 9 & 4 & 2_{10} & \end{array}$$

In binary subtraction it may also be necessary to borrow across more than one column. But when you borrow a 1_2 from 10_2, what remains is 1_2.

$$\begin{array}{r} 10_2 \\ -\ 1_2 \\ \hline 1_2 \end{array}$$

Thus the subtraction can be performed as follows:

$$\begin{array}{rcccc} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & \\ & & {\scriptstyle 1} & {\scriptstyle 1} & \leftarrow \text{borrow row} \\ 1 & \not{1} & \not{0} & 0 & 0_2 \\ - & 1 & 0 & 1 & 1_2 \\ \hline & 1 & 1 & 0 & 1_2 \end{array}$$

■

Circuits for Computer Addition

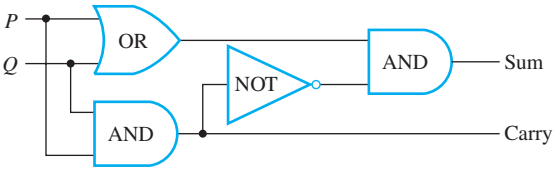
Consider the question of designing a circuit to produce the sum of two binary digits P and Q. Both P and Q can be either 0 or 1. And the following facts are known:

$$\begin{aligned} 1_2 + 1_2 &= 10_2, \\ 1_2 + 0_2 = 1_2 &= 01_2, \\ 0_2 + 1_2 = 1_2 &= 01_2, \\ 0_2 + 0_2 = 0_2 &= 00_2. \end{aligned}$$

It follows that the circuit must have two outputs—one for the left binary digit (this is called the **carry**) and one for the right binary digit (this is called the **sum**). The carry output is 1 if both P and Q are 1; it is 0 otherwise. Thus the carry can be produced using the AND-gate circuit that corresponds to the Boolean expression $P \wedge Q$. The sum output is 1 if either P or Q, but not both, is 1. The sum can, therefore, be produced using a circuit that corresponds to the Boolean expression for *exclusive or*: $(P \vee Q) \wedge \sim(P \wedge Q)$. (See Example 2.4.3(a).) Hence, a circuit to add two binary digits P and Q can be constructed as in Figure 2.5.1. This circuit is called a **half-adder**.

HALF-ADDER

Circuit



Input/OutputTable

P	*Q*	Carry	Sum
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	0

FIGURE 2.5.1 Circuit to Add $P + Q$, Where P and Q Are Binary Digits

Now consider the question of how to construct a circuit to add two binary integers, each with more than one digit. Because the addition of two binary digits may result in a carry to the next column to the left, it may be necessary to add three binary digits at certain points. In the following example, the sum in the right column is the sum of two binary digits, and, because of the carry, the sum in the left column is the sum of three binary digits.

$$\begin{array}{rr} & \overset{1}{} \quad \leftarrow \text{carry row} \\ & 1\ 1_2 \\ + & 1\ 1_2 \\ \hline & 1\ 1\ 0_2 \end{array}$$

Thus, in order to construct a circuit that will add multidigit binary numbers, it is necessary to incorporate a circuit that will compute the sum of three binary digits. Such a circuit is called a **full-adder**. Consider a general addition of three binary digits P, Q, and R that results in a carry (or left-most digit) C and a sum (or right-most digit) S.

$$\begin{array}{r} P \\ + \ Q \\ + \ R \\ \hline CS \end{array}$$

The operation of the full-adder is based on the fact that addition is a binary operation: Only two numbers can be added at one time. Thus P is first added to Q and then the result

is added to R. For instance, consider the following addition:

$$\begin{array}{r} 1_2 \\ +\ 0_2 \\ +\ 1_2 \\ \hline 10_2 \end{array} \quad \left.\begin{array}{l} 1_2 \\ +\ 0_2 \end{array}\right\} 1_2 + 0_2 = 01_2 \quad \left.\begin{array}{l} 01_2 \\ +\ 1_2 \end{array}\right\} 1_2 + 1_2 = 10_2$$

The process illustrated here can be broken down into steps that use half-adder circuits.

Step 1: Add P and Q using a half-adder to obtain a binary number with two digits.

$$\begin{array}{r} P \\ +\ Q \\ \hline C_1 S_1 \end{array}$$

Step 2: Add R to the sum $C_1\ S_1$ of P and Q.

$$\begin{array}{r} C_1 S_1 \\ +\ R \\ \hline \end{array}$$

To do this, proceed as follows:

Step 2a: Add R to S_1 using a half-adder to obtain the two-digit number $C_2 S$.

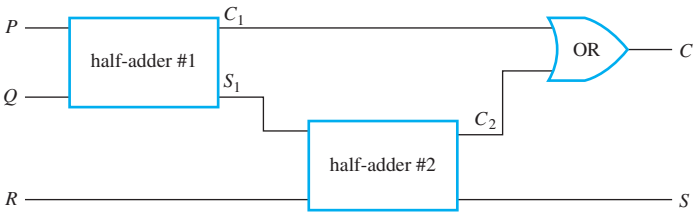
$$\begin{array}{r} S_1 \\ +\ R \\ \hline C_2 S \end{array}$$

Then S is the right-most digit of the entire sum of P, Q, and R.

Step 2b: Determine the left-most digit, C, of the entire sum as follows: First note that it is impossible for both C_1 and C_2 to be 1's. For if $C_1 = 1$, then P and Q are both 1, and so $S_1 = 0$. Consequently, the addition of S_1 and R gives a binary number $C_2 S_1$ where $C_2 = 0$. Next observe that C will be a 1 in the case that the addition of P and Q gives a carry of 1 or in the case that the addition of S_1 (the right-most digit of $P + Q$) and R gives a carry of 1. In other words, $C = 1$ if, and only if, $C_1 = 1$ or $C_2 = 1$. It follows that the circuit shown in Figure 2.5.2 will compute the sum of three binary digits.

FULL-ADDER

Circuit



Input/Output Table

P	*Q*	*R*	*C*	*S*
1	1	1	1	1
1	1	0	1	0
1	0	1	1	0
1	0	0	0	1
0	1	1	1	0
0	1	0	0	1
0	0	1	0	1
0	0	0	0	0

FIGURE 2.5.2 Circuit to Add $P + Q + R$, Where P, Q, and R Are Binary Digits

Two full-adders and one half-adder can be used together to build a circuit that will add two three-digit binary numbers PQR and STU to obtain the sum $WXYZ$. This is illustrated in Figure 2.5.3. Such a circuit is called a **parallel adder**. Parallel adders can be constructed to add binary numbers of any finite length.

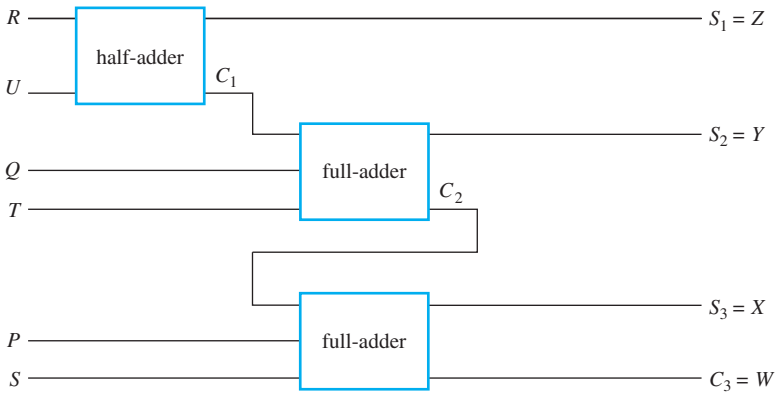


FIGURE 2.5.3 A Parallel Adder to Add PQR and STU to Obtain $WXYZ$

Two's Complements and the Computer Representation of Signed Integers

Typically a fixed number of bits is used to represent integers on a computer. One way to do this is to select a particular bit, normally the left-most, to indicate the sign of the integer, and to use the remaining bits for its absolute value in binary notation. The problem with this approach is that the procedures for adding the resulting numbers are somewhat complicated and the representation of 0 is not unique. A more common approach is to use "two's complements," which makes it possible to add integers quite easily and results in a unique representation for 0. Bit lengths of 64 and (sometimes) 32 are most often used in practice, but, for simplicity and because the principles are the same for all bit lengths, this discussion will focus on a bit length of 8.

We will show how to use eight bits to represent the 256 integers from -128 through 127 and how to perform additions and subtractions within this system of numbers. When the more realistic 32-bit two's complements system is used, more than 4 billion integers can be represented.

Definition

The 8-bit two's complement for an integer a between -128 and 127 is the 8-bit binary representation for $\begin{cases} a & \text{if } a \geq 0 \\ 2^8 - |a| & \text{if } a < 0. \end{cases}$

Thus the 8-bit representation for a nonnegative integer is the same as its 8-bit binary representation. As a concrete example for the negative integer -46, observe that

$$(2^8 - |-46|)_{10} = (256 - 46)_{10} = 210_{10} = (128 + 64 + 16 + 2)_{10} = 11010010_2,$$

and so the 8-bit two's complement for -46 is 11010010.

For negative integers, however, there is a more convenient way to compute two's complements, which involves less arithmetic than applying the definition directly.

The 8-Bit Two's Complement for a Negative Integer

The 8-bit two's complement for a negative integer a that is at least -128 can be obtained as follows:

- Write the 8-bit binary representation for $|a|$.
- Switch all the 1's to 0's and all the 0's to 1's. (This is called flipping, or complementing, the bits.)
- Add 1 in binary notation.

Example 2.5.6 **Finding a Two's Complement**

Use the method described above to find the 8-bit two's complement for -46.

Solution Write the 8-bit binary representation for $|-46|$ $(=46)$, switch all the 1's to 0's and all the 0's to 1's, and then add 1.

$$|-46|_{10} = 46_{10} = (32+8+4+2)_{10} = 00101110_2 \xrightarrow{\text{flip the bits}} 11010001 \xrightarrow{\text{add 1}} 11010010.$$

Note that this is the same result as was obtained directly from the definition. ■

The fact that the method for finding 8-bit two's complements works in general depends on the following facts:

1. The binary representation of $2^8 - 1$ is 11111111_2.
2. Subtracting an 8-bit binary number a from 11111111_2 switches all the 1's to 0's and all the 0's to 1's.
3. $2^8 - |a| = [(2^8 - 1) - |a|] + 1$ for any number a.

Here is how the facts are used when $a = -46$:

	1	1	1	1	1	1	1	1	$\leftrightarrow 2^8 - 1$
	0	0	1	0	1	1	1	0	$\leftrightarrow \lvert -46 \rvert$
0's and 1's are switched	1	1	0	1	0	0	0	1	$\leftrightarrow (2^8 - 1) - \lvert -46 \rvert$
1 is added +	0	0	0	0	0	0	0	1	$\leftrightarrow +1$
	1	1	0	1	0	0	1	0	$\leftrightarrow 2^8 - \lvert -46 \rvert$

Because 127 is the largest integer represented in the 8-bit two's complement system and because $127_{10} = 01111111_2$, all the 8-bit two's complements for nonnegative integers have a leading bit of 0. Moreover, because the bits are switched, the leading bit for all the negative integers is 1. Table 2.5.2 illustrates the 8-bit two's complement representations for the integers from -128 through 127.

TABLE 2.5.2

Integer	8-Bit Two's Complement	Decimal Form of Two's Complement for Negative Integers
127	01111111	
126	01111110	
⋮	⋮	
2	00000010	
1	00000001	
0	00000000	
−1	11111111	$2^8 - 1$
−2	11111110	$2^8 - 2$
−3	11111101	$2^8 - 3$
⋮	⋮	⋮
−127	10000001	$2^8 - 127$
−128	10000000	$2^8 - 128$

Observe that if the two's complement procedure is used on 11010010, which is the two's complement for −46, the result is

$$11010010 \xrightarrow{\text{flip the bits}} 00101101 \xrightarrow{\text{add 1}} 00101110,$$

which is the two's complement for 46. In general, if the two's complement procedure is applied to a positive or negative integer in two's complement form, the result is the negative (or opposite) of that integer. The only exception is the number −128. (See exercise 37a.)

To find the decimal representation of the negative integer with a given 8-bit two's complement:

- Apply the two's complement procedure to the given two's complement.
- Write the decimal equivalent of the result.

Example 2.5.7 **Finding a Number with a Given Two's Complement**

What is the decimal representation for the integer with two's complement 10101001?

Solution Since the left-most digit is 1, the integer is negative. Applying the two's complement procedure gives the following result:

$$10101001 \xrightarrow{\text{flip the bits}} 01010110 \xrightarrow{\text{add 1}} 01010111_2$$
$$= (64 + 16 + 4 + 2 + 1)_{10} = 87_{10} = |-87|_{10}.$$

So the answer is −87. You can check its correctness by deriving the two's complement of −87 directly from the definition:

$$(2^8 - |-87|)_{10} = (256 - 87)_{10} = 169_{10} = (128 + 32 + 8 + 1)_{10} = 10101001_2. \quad \blacksquare$$

Addition and Subtraction with Integers in Two's Complement Form

The main advantage of a two's complement representation for integers is that the same computer circuits used to add nonnegative integers in binary notation can be used for both additions and subtractions of integers in a two's complement system of numeration. First note that because of the algebraic identity

$$a - b = a + (-b) \text{ for all real numbers,}$$

any subtraction problem can be changed into an addition one. For example, suppose you want to compute $78 - 46$. This equals $78 + (-46)$, which should give an answer of 32. To see what happens when you add the numbers in their two's complement forms, observe that the 8-bit two's complement for 78 is the same as the ordinary binary representation for 78, which is 01001110 because $78 = 64 + 8 + 4 + 2$, and, as previously shown, the 8-bit two's complement for -46 is 11010010. Adding the numbers using binary addition gives the following:

	0	1	0	0	1	1	1	0	$\leftrightarrow 78$
+	1	1	0	1	0	0	1	0	$\leftrightarrow -46$
1	0	0	1	0	0	0	0	0	$\leftrightarrow 32?$

The result has a carry bit of 1 in the ninth, or 2^8th, position, but if you discard it, you obtain 00100000, which *is* the correct answer in 8-bit two's complement form because, since $32 = 2^8$,

$$32_{10} = 00100000_2.$$

In general, if you add numbers in 8-bit two's complement form and get a carry bit of 1 in the ninth, or 2^8th position, you should discard it. Using this procedure is equivalent to reducing the sum of the numbers "modulo 2^8," and it gives results that are correct in ordinary decimal arithmetic as long as the sum of the two numbers is within the fixed-bit-length system of integer representations you are using, in this case those between -128 and 127. The fact that this method produces correct results follows from general properties of modular arithmetic, which is discussed at length in Section 8.4.

General Procedure for Using 8-Bit Two's Complements to Add Two Integers

To add two integers in the range -128 through 127 whose sum is also in the range -128 through 127:

- Convert both integers to their 8-bit two's complement representations.
- Add the resulting integers using ordinary binary addition, discarding any carry bit of 1 that may occur in the 2^8th position.
- Convert the result back to decimal form.

When integers are restricted to the range -128 through 127, you can easily imagine adding two integers and obtaining a sum outside the range. For instance,

$(-87)+(-46) = -133$, which is less than -128 and, therefore, requires more than eight bits for its representation. Because this result is outside the 8-bit fixed-length register system imposed by the architecture of the computer, it is often labeled "overflow error." In the more realistic environment where integers are represented using 64 bits, they can range from less than -10^{19} to more than 10^{19}. So a vast number of integer calculations can be made without producing overflow error. And even if a 32-bit fixed integer length is used, nearly 4 billion integers are represented within the system.

Detecting overflow error turns out to be quite simple. The 8-bit two's complement sum of two integers will be outside the range from -128 through 127 if, and only if, the integers are both positive and the sum computed using 8-bit two's complements is negative, or if the integers are both negative and the sum computed using 8-bit two's complement is positive. To see a concrete example for how this works, consider trying to add (-87) and (-46). Here is what you obtain:

	1	0	1	0	1	0	0	1	$\leftrightarrow -87$
+	1	1	0	1	0	0	1	0	$\leftrightarrow -46$
1	0	1	1	1	1	0	1	1	

When you discard the 1 in the 2^8th position, you find that the leading digit of the result is 0, which would mean that the number with the two's complement representation for the sum of two negative numbers would be positive. So the computer signals an overflow error.*

Hexadecimal Notation

It should now be obvious that numbers written in binary notation take up much more space than numbers written in decimal notation. Yet many aspects of computer operation can best be analyzed using binary numbers. **Hexadecimal notation** is even more compact than decimal notation, and it is much easier to convert back and forth between hexadecimal and binary notation than it is between binary and decimal notation. The word *hexadecimal* comes from the Greek root *hex-*, meaning "six," and the Latin root *deci-*, meaning "ten." Hence *hexadecimal* refers to "sixteen," and hexadecimal notation is also called **base 16 notation**. Hexadecimal notation is based on the fact that any integer can be uniquely expressed as a sum of numbers of the form

$$d \cdot 16^n,$$

where each n is a nonnegative integer and each d is one of the integers from 0 to 15. In order to avoid ambiguity, each hexadecimal digit must be represented by a single symbol. The integers 10 through 15 are represented by the symbols A, B, C, D, E, and F. The 16 hexadecimal digits are shown in Table 2.5.3, together with their decimal equivalents and, for future reference, their 4-bit binary equivalents.

*If the carry bit had not been discarded and if the resulting 9 bits could be processed using a "9-bit two's complement conversion procedure," the result of 101111011 would convert to -133, which is the correct answer. However, the computer signals an error because -133 is not representable within its 8-bit two's complement system.

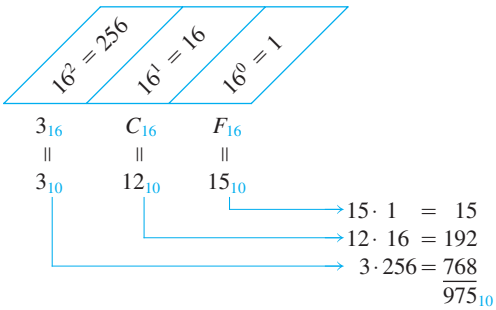
TABLE 2.5.3

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

Example 2.5.8 **Converting from Hexadecimal to Decimal Notation**

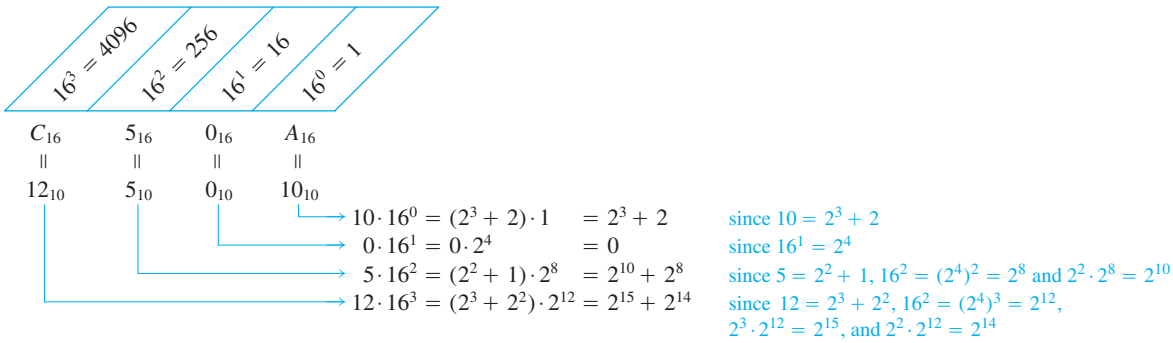
Convert $3CF_{16}$ to decimal notation.

Solution A schema similar to the one introduced in Example 2.5.2 can be used here.



So $3CF_{16} = 975_{10}$. ■

Now consider how to convert from hexadecimal to binary notation. In the example below the numbers are rewritten using powers of 2, and the laws of exponents are applied. The result suggests a general procedure.



But

$$(2^{15} + 2^{14}) + (2^{10} + 2^8) + 0 + (2^3 + 2)$$
$$= 1100\,0000\,0000\,0000_2 + 0101\,0000\,0000_2 + 0000\,0000_2 + 1010_2$$

by the rules for writing binary numbers.

So

$$C50A_{16} = \underbrace{1100}_{C_{16}}\ \underbrace{0101}_{5_{16}}\ \underbrace{0000}_{0_{16}}\ \underbrace{1010_2}_{A_{16}}$$

by the rules for adding binary numbers.

The procedure illustrated in this example can be generalized. In fact, the following sequence of steps will always give the correct answer.

To convert an integer from hexadecimal to binary notation:

- Write each hexadecimal digit of the integer in 4-bit binary notation.
- Juxtapose the results.

Example 2.5.9 **Converting from Hexadecimal to Binary Notation**

Convert $B09F_{16}$ to binary notation.

Solution $B_{16} = 11_{10} = 1011_2$, $0_{16} = 0_{10} = 0000_2$, $9_{16} = 9_{10} = 1001_2$, and $F_{16} = 15_{10} = 1111_2$. Consequently,

B	0	9	F
↕	↕	↕	↕
1011	0000	1001	1111

and the answer is 1011000010011111_2. ■

To convert integers written in binary notation into hexadecimal notation, reverse the steps of the previous procedure. Note that the commonly used computer representation for integers uses 32 bits. When these numbers are written in hexadecimal notation only eight characters are needed.

To convert an integer from binary to hexadecimal notation:

- Group the digits of the binary number into sets of four, starting from the right and adding leading zeros as needed.
- Convert the binary numbers in each set of four into hexadecimal digits. Juxtapose those hexadecimal digits.

Example 2.5.10 **Converting from Binary to Hexadecimal Notation**

Convert 100110110101001_2 to hexadecimal notation.

Solution First group the binary digits in sets of four, working from right to left and adding leading 0's if necessary.

0100 1101 1010 1001.

Convert each group of four binary digits into a hexadecimal digit.

0100	1101	1010	1001
↕	↕	↕	↕
4	D	A	9

Then juxtapose the hexadecimal digits.

$$4DA9_{16}$$ ■

Example 2.5.11 **Reading a Memory Dump**

The smallest addressable memory unit on most computers is one byte, or eight bits. In some debugging operations a dump is made of memory contents; that is, the contents of each memory location are displayed or printed out in order. To save space and make the output easier on the eye, the hexadecimal versions of the memory contents are given, rather than the binary versions. Suppose, for example, that a segment of the memory dump looks like

A3 BB 59 2E.

What is the actual content of the four memory locations?

Solution

$$A3_{16} = 10100011_2$$
$$BB_{16} = 10111011_2$$
$$59_{16} = 01011001_2$$
$$2E_{16} = 00101110_2$$ ■

TEST YOURSELF

1. To represent a nonnegative integer in binary notation means to write it as a sum of products of the form _____, where _____.
2. To add integers in binary notation, you use the facts that $1_2 + 1_2 =$ _____ and $1_2 + 1_2 + 1_2 =$ _____.
3. To subtract integers in binary notation, you use the facts that $10_2 - 1_2 =$ _____ and $11_2 - 1_2 =$ _____.
4. A half-adder is a digital logic circuit that _____, and a full-adder is a digital logic circuit that _____.
5. If a is an integer with $-128 \le a \le 127$, the 8-bit two's complement of a is _____ if $a \ge 0$ and is _____ if $a < 0$.
6. To find the 8-bit two's complement of a negative integer a that is at least -128, you _____, _____, and _____.
7. To add two integers in the range -128 through 127 whose sum is also in the range -128 through 127, you _____, _____, _____, and _____.
8. To represent a nonnegative integer in hexadecimal notation means to write it as a sum of products of the form _____, where _____.
9. To convert a nonnegative integer from hexadecimal to binary notation, you _____ and _____.

EXERCISE SET 2.5

Represent the decimal integers in 1–6 in binary notation.

1. 19
2. 55
3. 287
4. 458
5. 1609
6. 1424

Represent the integers in 7–12 in decimal notation.

7. 1110_2
8. 10111_2
9. 110110_2
10. 1100101_2
11. 1000111_2
12. 1011011_2

Perform the arithmetic in 13–20 using binary notation.

13. $1011_2 + 101_2$

14. $1001_2 + 1011_2$

15. $101101_2 + 11101_2$

16. $110111011_2 + 1001011010_2$

17. $10100_2 - 1101_2$

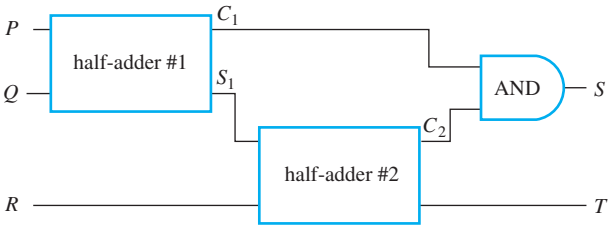
18. $11010_2 - 1101_2$

19. $101101_2 - 10011_2$

20. $1010100_2 - 10111_2$

21. Give the output signals S and T for the circuit shown below if the input signals P, Q, and R are as specified. Note that this is *not* the circuit for a full-adder.

a. $P = 1, Q = 1, R = 1$
b. $P = 0, Q = 1, R = 0$
c. $P = 1, Q = 0, R = 1$



22. Add $11111111_2 + 1_2$ and convert the result to decimal notation, to verify that $11111111_2 = (2^8 - 1)_{10}$.

Find the 8-bit two's complements for the integers in 23–26.

23. -23 **24.** -67 **25.** -4 **26.** -115

Find the decimal representations for the integers with the 8-bit two's complements given in 27–30.

27. 11010011 **28.** 10011001

29. 11110010 **30.** 10111010

Use 8-bit two's complements to compute the sums in 31–36.

31. $57 + (-118)$ **32.** $62 + (-18)$

33. $(-6) + (-73)$ **34.** $89 + (-55)$

35. $(-15) + (-46)$ **36.** $123 + (-94)$

37. a. Show that when you apply the 8-bit two's complement procedure to the 8-bit two's complement for -128, you get the 8-bit two's complement for -128.

***b.** Show that if a, b, and $a + b$ are integers in the range 1 through 128, then

$$(2^8 - a) + (2^8 - b) = (2^8 - (a + b)) + 2^8 \geq 2^8 + 2^7.$$

Explain why it follows that if integers a, b, and $a + b$ are all in the range 1 through 128, then the 8-bit two's complement of $(-a) + (-b)$ is a negative number.

Convert the integers in 38–40 from hexadecimal to decimal notation.

38. $A2BC_{16}$ **39.** $E0D_{16}$ **40.** $39EB_{16}$

Convert the integers in 41–43 from hexadecimal to binary notation.

41. $1C0ABE_{16}$ **42.** $B53DF8_{16}$ **43.** $4ADF83_{16}$

Convert the integers in 44–46 from binary to hexadecimal notation.

44. 00101110_2 **45.** 1011011111000101_2

46. 11001001011100_2

47. Octal Notation: In addition to binary and hexadecimal, computer scientists also use *octal notation* (base 8) to represent numbers. Octal notation is based on the fact that any integer can be uniquely represented as a sum of numbers of the form $d \cdot 8^n$, where each n is a nonnegative integer and each d is one of the integers from 0 to 7. Thus, for example, $5073_8 = 5 \cdot 8^3 + 0 \cdot 8^2 + 7 \cdot 8^1 + 3 \cdot 8^0 = 2619_{10}$.

a. Convert 61502_8 to decimal notation.
b. Convert 20763_8 to decimal notation.
c. Describe methods for converting integers from octal to binary notation and the reverse that are similar to the methods used in Examples 2.5.9 and 2.5.10 for converting back and forth from hexadecimal to binary notation. Give examples showing that these methods result in correct answers.

ANSWERS FOR TEST YOURSELF

1. $d \cdot 2^n$; $d = 0$ or $d = 1$, and n is a nonnegative integer **2.** 10_2; 11_2 **3.** 1_2; 10_2 **4.** outputs the sum of any two binary digits; outputs the sum of any three binary digits **5.** the 8-bit binary representation of a; the 8-bit binary representation of $2^8 - a$ **6.** write the 8-bit binary representation of a; flip the bits; add 1 in binary notation **7.** convert both integers to their 8-bit two's complements; add the results using binary notation; truncate any leading 1; convert back to decimal form **8.** $d \cdot 16^n$; $d = 0, 1, 2, \ldots 9, A, B, C, D, E, F$, and n is a nonnegative integer **9.** write each hexadecimal digit in 4-bit binary notation; juxtapose the results

The Two's Complement

Sign-and-Magnitude Method

Consider the problem of representing both positive AND negative integers over a given range in terms of only ones and zeroes. A straight-forward approach would be to deal with the sign and the magnitude (or, the absolute value) separately. For example, suppose we have 8 bits with which to work. The first bit could represent the sign of our number ("0" for a positive number, "1" for a negative number), while the other 7 bits could be a binary representation of the magnitude of the number. With only 7 bits to express it, the magnitude must range from zero to 127 (i.e., 0000000 to 1111111). Consider the following examples:

Value	Representation in 8 Bits		Value	Representation in 8 Bits
3	00000011		-3	10000011
14	00001110		-14	10001110
82	01010010		-82	11010010
127	01111111		-127	11111111

There are problems with this approach, however. Consider the case of zero. Should it be encoded as 00000000 or 10000000?

As another, more significant issue, consider how one is forced to add values. Our process depends on our context. If both numbers are positive, one can simply use normal binary addition:

```
  5     0 0000101
 +3     0 0000011
----   -----------
  8     0 0001000 (which represents 8)
```

But if there is a mixture of signs, we need to recognize this requires subtraction, as otherwise we get an incorrect result:

```
    5      00000101
 +(-3)     10000011
-------   ----------
    2      10001000 (which represents -8)
```

In the interests of efficiency, it would be nice if the process we use internally to add two numbers didn't depend on their values! Enter the two's complement...

The Two's Complement Method

In the (8-bit) two's complement representation of a integer value between -127 and 127, positive integers are represented in the normal binary manner (with a natural leading zero as the largest number is 127). Representations for negative numbers are obtained from the representations for positive numbers in the following way:

	Example 1: (value = -41)	Example 2: (value = -44)
1. Starting from the right, find the first '1'	00101001	00101100
2. Invert all of the bits to the left of that one	**11010111**	**11010100**

Here are some more examples:

Value	Two's Complement Representation		Value	Two's Complement Representation
3	00000011		-3	11111101
14	00001110		-14	11110010
82	01010010		-82	10101110
127	01111111		-127	10000001

Amazingly, representing negative numbers in this way allows us to compute the sum of two numbers, regardless of their signs, in the SAME way -- via normal binary addition!

```
  5     00000101
 +3     00000011
----   ----------
  8     00001000 (which represents 8)
```

```
    5      00000101
 +(-3)     11111101
-------   ----------
    2      00000010 (which represents 2)
```

If that wasn't enough, note that we get the added bonuses of now having only one representation for zero (00000000), and we can actually include one more number in our range (-128 = 10000000), without causing any problems. Pretty cool, eh? Makes one wonder how they came up with this process for negation, doesn't it?

Think about it though... The negative of a number is defined as the value that can be added to the original number to produce a zero. For example, consider -84. This is defined to be the number that can be added to 84 to produce a sum of zero. We know that 84 written in binary is given by 01010010 after adding the leading zero so that we have 8 bits total. Let $d_7d_6d_5d_4d_3d_2d_1d_0$ be the binary expansion of -84. Then we require that

$$\begin{array}{cccccccccc} & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ + & d_7 & d_6 & d_5 & d_4 & d_3 & d_2 & d_1 & d_0 \\ \hline \ldots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

By duplicating the right-most 1 (i.e., letting $d_2 = 1$), and all of the zeros to the right of the right-most 1 (i.e., $d_1 = d_0 = 0$), we assure sums of zeros there. Most of these will be of the form $0 + 0 = 0$, while the sum involving the right-most 1 will produce 0 and a carried 1 (recall in binary, $1 + 1 = 10$).

$$\begin{array}{cccccccccc} & & & & & & 1 & & & \\ & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ + & d_7 & d_6 & d_5 & d_4 & d_3 & 1 & 0 & 0 \\ \hline & & & & & & 0 & 0 & 0 \end{array}$$

Then, by inverting all of the bits to the left of the right-most 1 (i.e., d_2 through d_7), we assure each of these columns sum to 1 before considering any "carries". Now, throw in the carried 1 produced from the right-most 1 and its duplicate, and one produces a 0 and another carried 1 in each column working to the left. In essence, the carried 1 simply propagates down the line -- until it disappears when we run out of bits (remember, we restricted ourselves to 8 bits).

$$\begin{array}{cccccccccc} & 1 & 1 & 1 & 1 & 1 & & & & \\ & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ + & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ \hline & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

The result is a bit string consisting of 8 zeros, which is the bit string for zero -- which was exactly what was desired! Cool, isn't it?