of as "adding the opposite," as in 10 + (−7). Complement notation allows us to simplify subtraction by turning it into addition, but it also gives us a method to represent negative numbers. Because we do not wish to use a special bit to represent the sign (as we did in signed-magnitude representation), we need to remember that if a number is negative, we should convert it to its complement. The result should have a 1 in the leftmost bit position to indicate the number is negative. If the number is positive, we do not have to convert it to its complement. All positive numbers should have a zero in the leftmost bit position. Example 2.16 illustrates these concepts.

EXAMPLE 2.16 Express 23_{10} and -9_{10} in 8-bit binary one's complement form.

$$23_{10} = +(00010111_2) = 00010111_2$$
$$-9_{10} = -(00001001_2) = 11110110_2$$

Suppose we wish to subtract 9 from 23. To carry out a one's complement subtraction, we first express the subtrahend (9) in one's complement, then add it to the minuend (23); we are effectively now adding −9 to 23. The high-order bit will have a 1 or a 0 carry, which is added to the low-order bit of the sum. (This is called **end carry-around** and results from using the diminished radix complement.)

EXAMPLE 2.17 Add 23_{10} to -9_{10} using one's complement arithmetic.

	1 ←	1	1	1		1	1		⇐ carries	
		0	0	0	1	0	1	1	1	(23)
The last	+	1	1	1	1	0	1	1	0	+(–9)
carry is added		0	0	0	0	1	1	0	1	
								+	1	
to the sum.		0	0	0	0	1	1	1	0	14_{10}

EXAMPLE 2.18 Add 9_{10} to -23_{10} using one's complement arithmetic.

The last	0 ←	0	0	0	0	1	0	0	1	(9)
carry is zero	+	1	1	1	0	1	0	0	0	+(–23)
so we are done.		1	1	1	1	0	0	0	1	-14_{10}

How do we know that 11110001_2 is actually -14_{10}? We simply need to take the one's complement of this binary number (remembering it must be negative

because the leftmost bit is negative). The one's complement of 11110001_2 is 00001110_2, which is 14.

The primary disadvantage of one's complement is that we still have two representations for zero: 00000000 and 11111111. For this and other reasons, computer engineers long ago stopped using one's complement in favor of the more efficient two's complement representation for binary numbers.

Two's Complement

Two's complement is an example of a radix complement. Given a number N in base r having d digits, the radix complement of N is defined to be $r^d - N$ for $N \neq 0$ and 0 for $N = 0$. The radix complement is often more intuitive than the diminished radix complement. Using our odometer example, the ten's complement of going forward 2 miles is $10^3 - 2 = 998$, which we have already agreed indicates a negative (backward) distance. Similarly, in binary, the two's complement of the 4-bit number 0011_2 is $2^4 - 0011_2 = 10000_2 - 0011_2 = 1101_2$.

Upon closer examination, you will discover that two's complement is nothing more than one's complement incremented by 1. To find the two's complement of a binary number, simply flip bits and add 1. This simplifies addition and subtraction as well. Since the subtrahend (the number we complement and add) is incremented at the outset, however, there is no end carry-around to worry about. We simply discard any carries involving the high-order bits. Remember, only negative numbers need to be converted to two's complement notation, as indicated in Example 2.19.

EXAMPLE 2.19 Express 23_{10}, -23_{10}, and -9_{10} in 8-bit binary two's complement form.

$$23_{10} = +(00010111_2) = 00010111_2$$
$$-23_{10} = -(00010111_2) = 11101000_2 + 1 = 11101001_2$$
$$-9_{10} = -(00001001_2) = 11110110_2 + 1 = 11110111_2$$

Suppose we are given the binary representation for a number and want to know its decimal equivalent? Positive numbers are easy. For example, to convert the two's complement value of 00010111_2 to decimal, we simply convert this binary number to a decimal number to get 23. However, converting two's complement negative numbers requires a reverse procedure similar to the conversion from decimal to binary. Suppose we are given the two's complement binary value of 11110111_2, and we want to know the decimal equivalent. We know this is a negative number but must remember it is represented using two's complement. We first flip the bits and then add 1 (find the one's complement and add 1). This results in the following: $00001000_2 + 1 = 00001001_2$. This is equivalent to the decimal value 9. However, the original number we started with was negative, so we end up with -9 as the decimal equivalent to 11110111_2.

The following two examples illustrate how to perform addition (and hence subtraction, because we subtract a number by adding its opposite) using two's complement notation.

EXAMPLE 2.20 Add 9_{10} to -23_{10} using two's complement arithmetic.

$$\begin{array}{rcccccccc} & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \quad (9) \\ + & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \quad +(-23) \\ \hline & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \quad -14_{10} \end{array}$$

It is left as an exercise for you to verify that 11110010_2 is actually -14_{10} using two's complement notation.

EXAMPLE 2.21 Find the sum of 23_{10} and -9_{10} in binary using two's complement arithmetic.

$$\begin{array}{lrcccccccc} & 1\leftarrow & 1 & 1 & 1 & & 1 & 1 & 1 & \Leftarrow \text{carries} \\ \text{Discard} & & 0 & 0 & 0 & 1 & 0 & 1 & 1 & (23) \\ \text{carry.} & + & 1 & 1 & 1 & 1 & 0 & 1 & 1 & +(-9) \\ \hline & & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \quad 14_{10} \end{array}$$

Notice that the discarded carry in Example 2.21 did not cause an erroneous result. An overflow occurs if two positive numbers are added and the result is negative, or if two negative numbers are added and the result is positive. It is not possible to have overflow when using two's complement notation if a positive and a negative number are being added together.

Simple computer circuits can easily detect an overflow condition using a rule that is easy to remember. You'll notice in Example 2.21 that the carry going into the sign bit (a 1 is carried from the previous bit position into the sign bit position) is the same as the carry going out of the sign bit (a 1 is carried out and discarded). When these carries are equal, no overflow occurs. When they differ, an overflow indicator is set in the arithmetic logic unit, indicating the result is incorrect.

> **A Simple Rule for Detecting an Overflow Condition in Signed Numbers:** *If the carry into the sign bit equals the carry out of the bit, no overflow has occurred. If the carry into the sign bit is different from the carry out of the sign bit, overflow (and thus an error) has occurred.*

The hard part is getting programmers (or compilers) to consistently check for the overflow condition. Example 2.22 indicates overflow because the carry into the sign bit (a 1 is carried in) is not equal to the carry out of the sign bit (a 0 is carried out).

INTEGER MULTIPLICATION AND DIVISION

Unless sophisticated algorithms are used, multiplication and division can consume a considerable number of computation cycles before a result is obtained. Here, we discuss only the most straightforward approach to these operations. In real systems, dedicated hardware is used to optimize throughput, sometimes carrying out portions of the calculation in parallel. Curious readers will want to investigate some of these advanced methods in the references cited at the end of this chapter.

The simplest multiplication algorithms used by computers are similar to traditional pencil and paper methods used by humans. The complete multiplication table for binary numbers couldn't be simpler: zero times any number is zero, and one times any number is that number.

To illustrate simple computer multiplication, we begin by writing the multiplicand and the multiplier to two separate storage areas. We also need a third storage area for the product. Starting with the low-order bit, a pointer is set to each digit of the multiplier. For each digit in the multiplier, the multiplicand is "shifted" one bit to the left. When the multiplier is 1, the "shifted" multiplicand is added to a running sum of partial products. Because we shift the multiplicand by one bit for each bit in the multiplier, a product requires double the working space of either the multiplicand or the multiplier.

There are two simple approaches to binary division: We can either iteratively subtract the denominator from the divisor, or we can use the same trial-and-error method of long division that we were taught in grade school. As mentioned above with multiplication, the most efficient methods used for binary division are beyond the scope of this text and can be found in the references at the end of this chapter.

Regardless of the relative efficiency of any algorithms that are used, division is an operation that can always cause a computer to crash. This is the

EXAMPLE 2.22 Find the sum of 126_{10} and 8_{10} in binary using two's complement arithmetic.

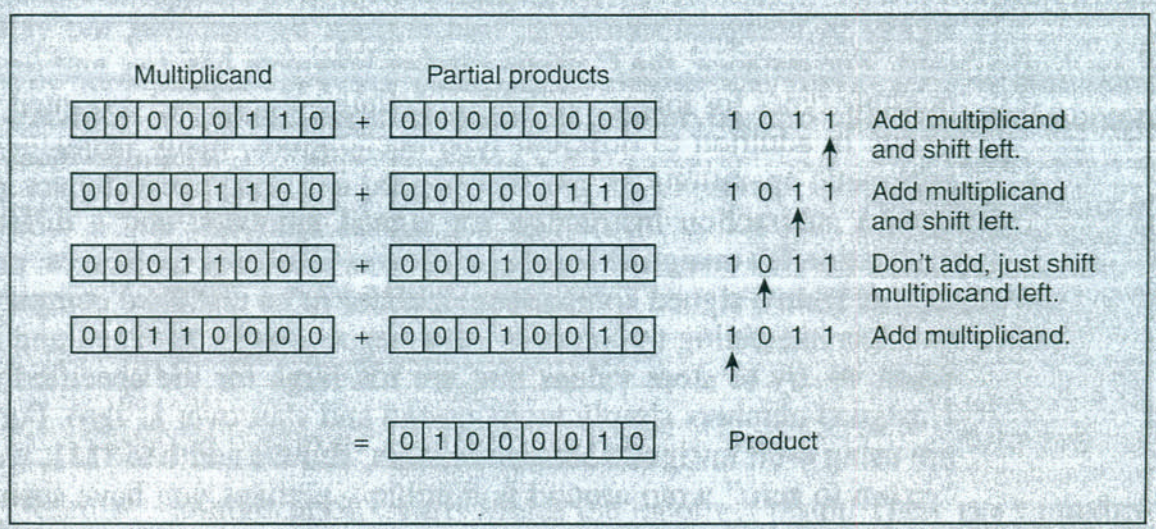
	0←	1	1	1	1					⇐ carries
Discard last		0	1	1	1	1	1	1	0	(126)
carry.	+	0	0	0	0	1	0	0	0	+(8)
		1	0	0	0	0	1	1	0	(–122???)

A one is carried into the leftmost bit, but a zero is carried out. Because these carries are not equal, an overflow has occurred. (We can easily see that two positive numbers are being added but the result is negative.) We return to this topic in Section 2.4.5.

case particularly when division by zero is attempted or when two numbers of enormously different magnitudes are used as operands. When the divisor is much smaller than the dividend, we get a condition known as **divide underflow**, which the computer sees as the equivalent of division by zero, which is impossible.

Computers make a distinction between integer division and floating-point division. With integer division, the answer comes in two parts: a quotient and a remainder. Floating-point division results in a number that is expressed as a binary fraction. These two types of division are sufficiently different from each other as to warrant giving each its own special circuitry. Floating-point calculations are carried out in dedicated circuits called **floating-point units**, or **FPUs**.

EXAMPLE Find the product of 00000110_2 and 00001011_2.



Two's complement is the most popular choice for representing signed numbers. The algorithm for adding and subtracting is quite easy, has the best representation for 0 (all 0 bits), is self-inverting, and is easily extended to larger numbers of bits. The biggest drawback is in the asymmetry seen in the range of values that can be represented by N bits. With signed-magnitude numbers, for example, 4 bits allow us to represent the values -7 through $+7$. However, using two's complement, we can represent the values -8 through $+7$, which is often confusing to anyone learning about complement representations. To see why $+7$ is the largest number we can represent using 4-bit two's complement representation, we need only remember the first bit must be 0. If the remaining bits are all 1s (giving us the largest magnitude possible), we have 0111_2, which is 7. An immediate reaction to this is that the smallest negative number should then be 1111_2, but we can see that

1111_2 is actually -1 (flip the bits, add one, and make the number negative). So how do we represent -8 in two's complement notation using 4 bits? It is represented as 1000_2. We know this is a negative number. If we flip the bits (0111), add 1 (to get 1000, which is 8), and make it negative, we get -8.

2.4.3 Unsigned Versus Signed Numbers

We introduced our discussion of binary integers with unsigned numbers. Unsigned numbers are used to represent values that are guaranteed not to be negative. A good example of an unsigned number is a memory address. If the 4-bit binary value 1101 is unsigned, then it represents the decimal value 13, but as a signed two's complement number, it represents -3. Signed numbers are used to represent data that can be either positive or negative.

A computer programmer must be able to manage both signed and unsigned numbers. To do so, the programmer must first identify numeric values as either signed or unsigned numbers. This is done by declaring the value as a specific type. For instance, the C programming language has `int` and `unsigned int` as possible types for integer variables, defining signed and unsigned integers respectively. In addition to different type declarations, many languages have different arithmetic operations for use with signed and unsigned numbers. A language may have a subtraction instruction for signed numbers, and a different subtraction instruction for unsigned numbers. In most assembly languages, programmers can choose from a signed comparison operator or an unsigned comparison operator.

It is interesting to compare what happens with unsigned and signed numbers when we try to store values that are too large for the specified number of bits. Unsigned numbers simply wrap around and start over at zero. For example, if we are using 4-bit unsigned binary numbers, and we add 1 to 1111, we get 0000. This "return to zero" wrap around is familiar—perhaps you have seen a high mileage car in which the odometer has wrapped back around to zero. However, signed numbers devote half of their space to positive numbers and the other half to negative numbers. If we add 1 to the largest positive 4-bit two's complement number 0111 (+7), we get 1000 (-8). This wrap around with the unexpected change in sign has been problematic to inexperienced programmers, resulting in multiple hours of debugging time. Good programmers understand this condition and make appropriate plans to deal with the situation before it occurs.

2.4.4 Computers, Arithmetic, and Booth's Algorithm

Computer arithmetic as introduced in this chapter may seem simple and straightforward, but it is a field of major study in computer architecture. The basic focus is on the implementation of arithmetic functions, which can be realized in software, firmware, or hardware. Researchers in this area are working toward designing superior central processing units (CPUs), developing high-performance arithmetic circuits, and contributing to the area of embedded systems application-specific circuits.

They are working on algorithms and new hardware implementations for fast addition, subtraction, multiplication, and division, as well as fast floating-point operations. Researchers are looking for schemes that use non-traditional approaches, such as the **fast carry look-ahead** principle, **residue arithmetic**, and **Booth's algorithm**. Booth's algorithm is a good example of one such scheme and is introduced here in the context of signed two's complement numbers to give you an idea of how a simple arithmetic operation can be enhanced by a clever algorithm.

Although Booth's algorithm usually yields a performance increase when multiplying two's complement numbers, there is another motivation for introducing this algorithm. In Section 2.4.2 we covered examples of two's complement addition and saw that the numbers could be treated as unsigned values. We simply perform "regular" addition, as the following example illustrates:

$$\begin{array}{rr} 1001 & (-7) \\ +\ 0011 & (+3) \\ \hline 1100 & (-4) \end{array}$$

The same is true for two's complement subtraction. However, now consider the standard pencil and paper method for multiplying the following two's complement numbers:

$$\begin{array}{rr} 1011 & (-5) \\ \times\ 1100 & (-4) \\ \hline 0000 & \\ 0000 & \\ 1011 & \\ 1011 & \\ \hline 10000100 & (-124) \end{array}$$

"Regular" multiplication clearly yields the incorrect result. There are a number of solutions to this problem, such as converting both values to positive numbers, performing conventional multiplication, and then remembering if one or both values were negative to determine if the result should be positive or negative. Booth's algorithm not only solves this dilemma, but also speeds up multiplication in the process.

The general idea of Booth's algorithm is to increase the speed of a multiplication when there are consecutive zeros or ones in the multiplier. It is easy to see that consecutive zeros help performance. For example, if we use the tried and true pencil and paper method and find 978×1001, the multiplication is much easier than if we take 978×999. This is due to the two zeros found in 1001. However, if we rewrite the two problems as follows:

$$978 \times 1001 = 978 \times (1000 + 1) = 978 \times 1000 + 978$$
$$978 \times 999 = 978 \times (1000 - 1) = 978 \times 1000 - 978$$

we see that the problems are in fact equal in difficulty.