Our goal is to use a string of ones in a binary number to our advantage in much the same way we use a string of zeros to our advantage. We can use the rewriting idea from above. For example, the binary number 0110 can be rewritten $1000 - 0010 = -0010 + 1000$. The two ones have been replaced by a "subtract" (determined by the rightmost 1 in the string) followed by an add (determined by moving one position left of the leftmost 1 in the string).

Consider the following standard multiplication example:

```
    0011
  × 0110
  ------
  + 0000   (0 in multiplier means simple shift)
  + 0011   (1 in multiplier means add multiplicand and shift)
 + 0011    (1 in multiplier means add multiplicand and shift)
+ 0000     (0 in multiplier means simple shift)
 --------
 00010010
```

The idea of Booth's algorithm is to replace the string of ones in the multiplier with an initial subtract when we see the rightmost 1 of the string (or subtract 0011) and then later add for the bit after the last one (or add 001100). In the middle of the string, we can now use simple shifting:

```
    0011
  × 0110
  ------
  + 0000   (0 in multiplier means shift)
  - 0011   (first 1 in multiplier means subtract multiplicand and shift)
 + 0000    (middle of string of 1s means shift)
+ 0011     (prior step had last 1 so add multiplicand)
 --------
 00010010
```

In Booth's algorithm, if the multiplicand and multiplier are n-bit two's complement numbers, the result is a $2n$-bit two's complement value. Therefore, when we perform our intermediate steps, we must extend our n-bit numbers to $2n$-bit numbers. If a number is negative and we extend it, we must extend the sign. For example, the value 1000 (−8) extended to 8 bits would be 11111000. We continue to work with bits in the multiplier, shifting each time we complete a step. However, we are interested in *pairs* of bits in the multiplier and proceed according to the following rules:

1. If the current multiplier bit is 1 and the preceding bit was 0, we are at the beginning of a string of ones, so subtract the multiplicand from the product (or add the opposite).
2. If the current multiplier bit is 0 and the preceding bit was 1, we are at the end of a string of ones, so we add the multiplicand to the product.
3. If it is a 00 pair, or a 11 pair, do no arithmetic operation (we are in the middle of a string of zeros or a string of ones). Simply shift. The power of the algorithm is in this step: we can now treat a string of ones as a string of zeros and do nothing more than shift.

Note: The first time we pick a pair of bits in the multiplier, we should assume a mythical 0 as the "previous" bit. Then we simply move left one bit for our next pair.

Example 2.23 illustrates the use of Booth's algorithm to multiply -3×5 using signed 4-bit two's complement numbers.

EXAMPLE 2.23: Negative 3 in 4-bit two's complement is 1101. Extended to 8 bits, it is 11111101. Its complement is 00000011. When we see the rightmost 1 in the multiplier, it is the beginning of a string of 1s, so we treat it as if it were the string 10:

```
      1101      (for subtracting, we will add −3's complement, or 00000011)
    × 0101
-----------
  +00000011     (10 = subtract 1101 = add 00000011)
 +11111101      (01 = add 11111101 to product—note sign extension)
+00000011       (10 = subtract 1101 = add 00000011)
+11111101       (01 = add multiplicand 11111101 to product)
-----------
 10011110001    (using the 8 rightmost bits we have −3 × 5 = −15)
 ~~
```

Ignore extended sign bits that go beyond 2n.

EXAMPLE 2.24: Let's look at the larger example of 53×126:

```
      00110101  (for subtracting, we will add the complement of 53, or
    × 01111110  11001011)
---------------
+0000000000000000  (00 = simple shift)
+11111111001011    (10 = subtract = add 11001011, extend sign)
+00000000000000    (11 = simple shift)
+0000000000000     (11 = simple shift)
+000000000000      (11 = simple shift)
+00000000000       (11 = simple shift)
+0000000000        (11 = simple shift)
+000110101         (01 = add)
---------------
 10001101000010110 (53 × 126 = 6678)
```

Note we have not shown the extended sign bits that go beyond what we need and use only the 16 rightmost bits. The entire string of ones in the multiplier was replaced by a subtract (adding 11001011) followed by an add. Everything in the middle is simply shifting, something that is very easy for a computer to do (as we will see in Chapter 3). If the time required for a computer to do an add is sufficiently larger than that required to do a shift, Booth's algorithm can provide a considerable increase in performance. This depends somewhat, of course, on the multiplier. If the multiplier has strings of zeros and/or ones, the algorithm works well. If the multiplier consists of an alternating string of zeros and ones (the worst case), using Booth's algorithm might very well require more operations than the standard approach.

Computers perform Booth's algorithm by adding and shifting values stored in registers. A special type of shift called an **arithmetic shift** is necessary to preserve the sign bit. Many books present Booth's algorithm in terms of arithmetic shifts and add operations on registers only, and may appear quite different from the preceding method. We have presented Booth's algorithm so that it more closely resembles the pencil and paper method with which we are all familiar, although it is equivalent to the computer algorithms presented elsewhere.

There have been many algorithms developed for fast multiplication, but many do not hold for signed multiplication. Booth's algorithm not only allows multiplication to be performed faster in most cases, but it also has the added bonus in that it works correctly on signed numbers.

2.4.5 Carry Versus Overflow

The wrap around referred to in the preceding section is really overflow. CPUs often have flags to indicate both carry and overflow. However, the overflow flag is used only with signed numbers and means nothing in the context of unsigned numbers, which use the carry flag instead. If carry (which means *carry out of the leftmost bit*) occurs in unsigned numbers, we know we have overflow (the new value is too large to be stored in the given number of bits) but the overflow bit is not set. Carry out can occur in signed numbers as well; however, its occurrence in signed numbers is neither sufficient nor necessary for overflow. We have already seen that overflow in signed numbers can be determined if the carry in to the leftmost bit and the carry out of the leftmost bit differ. However, carry out of the leftmost bit in unsigned operations always indicates overflow.

To illustrate these concepts, consider 4-bit unsigned and signed numbers. If we add the two unsigned values 0111 (7) and 0001 (1), we get 1000 (8). There is no carry (out), and thus no error. However, if we add the two unsigned values 0111 (7) and 1011 (11), we get 0010 with a carry, indicating there is an error (indeed, 7 + 11 is not 2). This wrap around would cause the carry flag in the CPU to be set. Essentially, carry out in the context of unsigned numbers means an overflow has occurred, even though the overflow flag is not set.

We said carry (out) is neither sufficient nor necessary for overflow in signed numbers. Consider adding the two's complement integers 0101 (+5) and 0011 (+3). The result is 1000 (−8), which is clearly incorrect. The problem is that we have a carry in to the sign bit, but no carry out, which indicates we have an overflow (therefore carry is not necessary for overflow). However, if we now add 0111 (+7) and 1011 (−5), we get the correct result 0010 (+2). We have both a carry in to and a carry out of the leftmost bit so there is no error (so carry is not sufficient for overflow). The carry flag would be set, but the overflow flag would not be set. Thus carry out does not necessarily indicate an error in signed numbers, nor does the lack of carry out indicate the answer is correct.

To summarize, the rule of thumb used to determine when carry indicates an error depends on whether we are using signed or unsigned numbers. For unsigned numbers, a carry (out of the leftmost bit) indicates the total number of bits was not large

Expression	Result	Carry?	Overflow?	Correct result?
0100(+4)+0010(-2)	0110(+6)	No	No	Yes
0100(+4)+0110(+6)	1010(-6)	No	Yes	No
1100(-4)+1110(-2)	1010(-6)	Yes	No	Yes
1100(-4)+1010(-6)	0110(+6)	Yes	Yes	No

TABLE 2.2 Examples of Carry and Overflow in Signed Numbers

enough to hold the resulting value, and overflow has occurred. For signed numbers, if the carry in to the sign bit and the carry (out of the sign bit) differ, then overflow has occurred. The overflow flag is set only when overflow occurs with signed numbers.

Carry and overflow clearly occur independently of each other. Examples using signed numbers are given in Table 2.2. Carry in to the sign bit is not indicated in the table.

2.5 FLOATING-POINT REPRESENTATION

If we wanted to build a real computer, we could use any of the integer representations that we just studied. We would pick one of them and proceed with our design tasks. Our next step would be to decide the word size of our system. If we want our system to be really inexpensive, we would pick a small word size, say 16 bits. Allowing for the sign bit, the largest integer that this system could store is 32,767. So now what do we do to accommodate a potential customer who wants to keep a tally of the number of spectators paying admission to professional sports events in a given year? Certainly, the number is larger than 32,767. No problem. Let's just make the word size larger. Thirty-two bits ought to do it. Our word is now big enough for just about anything that anyone wants to count. But what if this customer also needs to know the amount of money each spectator spends per minute of playing time? This number is likely to be a decimal fraction. Now we're really stuck.

The easiest and cheapest approach to this problem is to keep our 16-bit system and say, "Hey, we're building a cheap system here. If you want to do fancy things with it, get yourself a good programmer." Although this position sounds outrageously flippant in the context of today's technology, it was a reality in the earliest days of each generation of computers. There simply was no such thing as a floating-point unit in many of the first mainframes or microcomputers. For many years, clever programming enabled these integer systems to act as if they were, in fact, floating-point systems.

If you are familiar with scientific notation, you may already be thinking of how you could handle floating-point operations—how you could provide **floating-point emulation**—in an integer system. In scientific notation, numbers are expressed in two parts: a fractional part and an exponential part that indicates the power of ten to which the fractional part should be raised to obtain the value we need. So to express 32,767 in scientific notation, we could write 3.2767×10^4. Scientific notation

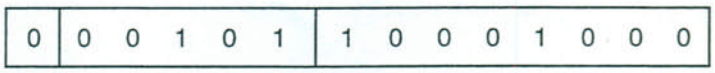
simplifies pencil and paper calculations that involve very large or very small numbers. It is also the basis for floating-point computation in today's digital computers.

2.5.1 A Simple Model

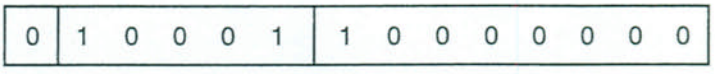
In digital computers, floating-point numbers consist of three parts: a sign bit, an exponent part (representing the exponent on a power of 2), and a fractional part (which has sparked considerable debate regarding appropriate terminology). The term **mantissa** is widely accepted when referring to this fractional part. However, many people take exception to this term because it also denotes the fractional part of a logarithm, which is not the same as the fractional part of a floating point number. The Institute of Electrical and Electronics Engineers (IEEE) introduced the term **significand** to refer to the fractional part of a floating point number combined with the implied binary point and implied 1 (which we discuss at the end of this section). Regrettably, the two terms *mantissa* and *significand* have become interchangeable when referring to the fractional part of a floating point number, even though they are not technically equivalent. Throughout this book, we refer to the fractional part as the significand, regardless of whether or not it includes the implied 1 as intended by IEEE.

The number of bits used for the exponent and significand depends on whether we would like to optimize for range (more bits in the exponent) or precision (more bits in the significand). (We discuss range and precision in more detail in Section 2.5.7.) For the remainder of this section, we will use a 14-bit model with a 5-bit exponent, an 8-bit significand, and a sign bit (see Figure 2.1). More general forms are described in Section 2.5.2.

Let's say that we wish to store the decimal number 17 in our model. We know that $17 = 17.0 \times 10^0 = 1.7 \times 10^1 = 0.17 \times 10^2$. Analogously, in binary, $17_{10} = 10001_2 \times 2^0 = 1000.1_2 \times 2^1 = 100.01_2 \times 2^2 = 10.001_2 \times 2^3 = 1.0001_2 \times 2^4 = 0.10001_2 \times 2^5$. If we use this last form, our fractional part will be 10001000 and our exponent will be 00101, as shown here:



Using this form, we can store numbers of much greater magnitude than we could using a **fixed-point** representation of 14 bits (which uses a total of 14 binary digits plus a binary, or radix, point). If we want to represent $65536 = 0.1_2 \times 2^{17}$ in this model, we have:



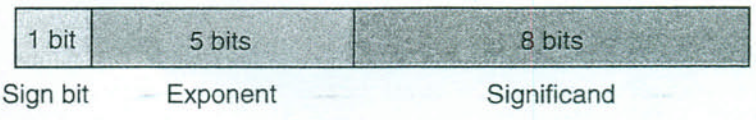
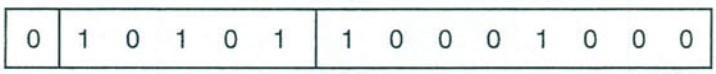


FIGURE 2.1 Simple Model Floating-Point Representation

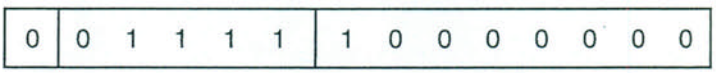
One obvious problem with this model is that we haven't provided for negative exponents. If we wanted to store 0.25, we would have no way of doing so because 0.25 is 2^{-2} and the exponent -2 cannot be represented. We could fix the problem by adding a sign bit to the exponent, but it turns out that it is more efficient to use a **biased** exponent, because we can use simpler integer circuits designed specifically for unsigned numbers when comparing the values of two floating-point numbers.

The idea behind using a bias value is to convert every integer in the range into a non-negative integer, which is then stored as a binary numeral. The integers in the desired range of exponents are first adjusted by adding this fixed bias value to each exponent. The bias value is a number near the middle of the range of possible values that we select to represent zero. In this case, we could select 16 because it is midway between 0 and 31 (our exponent has 5 bits, thus allowing for 2^5 or 32 values). Any number larger than 16 in the exponent field represents a positive value. Values less than 16 indicate negative values. This is called an **excess-16** representation because we have to subtract 16 to get the true value of the exponent. Note that exponents of all zeros or all ones are typically reserved for special numbers (such as zero or infinity). In our simple model, we allow exponents of all zeros and ones.

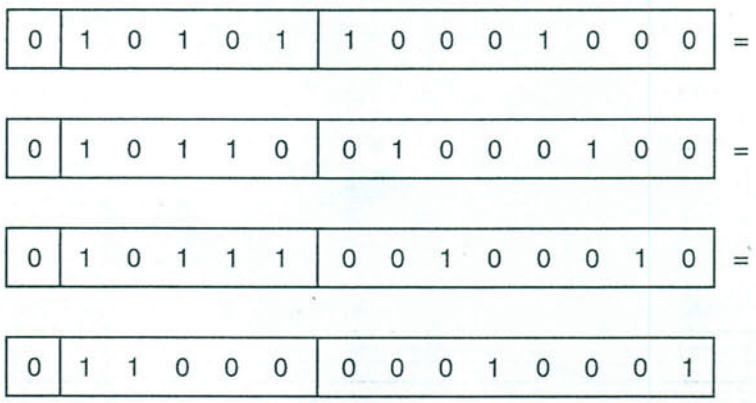
Returning to our example of storing 17, we calculated $17_{10} = 0.10001_2 \times 2^5$. The biased exponent is now $16 + 5 = 21$:



If we wanted to store $0.25 = 0.1 \times 2^{-1}$ we would have:



There is still one rather large problem with this system: We do not have a unique representation for each number. All of the following are equivalent:



Because synonymous forms such as these are not well-suited for digital computers, floating-point numbers must be **normalized**—that is, the leftmost bit of the significand must always be 1. This process is called **normalization**. This convention has the additional advantage that if the 1 is implied, we effectively gain an extra bit of precision in the significand. Normalization works well for every value except zero, which

contains no non-zero bits. For that reason, any model used to represent floating-point numbers must treat zero as a special case. We will see in the next section that the IEEE-754 floating-point standard makes an exception to the rule of normalization.

EXAMPLE 2.25 Express 0.03125_{10} in normalized floating-point form using the simple model with excess-16 bias.

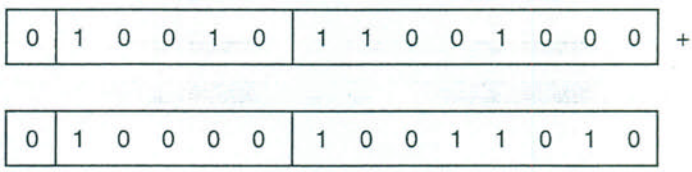
$0.03125_{10} = 0.00001_2 \times 2^0 = 0.0001 \times 2^{-1} = 0.001 \times 2^{-2} = 0.01 \times 2^{-3} = 0.1 \times 2^{-4}$. Applying the bias, the exponent field is $16 - 4 = 12$.

Note that in our simple model we have not expressed the number using the normalization notation that implies the 1, which is introduced in Section 2.5.4.

2.5.2 Floating-Point Arithmetic

If we wanted to add two decimal numbers that are expressed in scientific notation, such as $1.5 \times 10^2 + 3.5 \times 10^3$, we would change one of the numbers so that both of them are expressed in the same power of the base. In our example, $1.5 \times 10^2 + 3.5 \times 10^3 = 0.15 \times 10^3 + 3.5 \times 10^3 = 3.65 \times 10^3$. Floating-point addition and subtraction work the same way, as illustrated below.

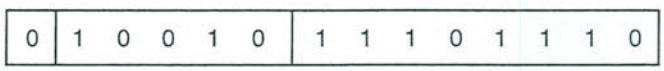
EXAMPLE 2.26 Add the following binary numbers as represented in a normalized 14-bit format, using the simple model with a bias of 16.



We see that the addend is raised to the second power and that the augend is to the zero power. Alignment of these two operands on the binary point gives us:

$$\begin{array}{r} 11.001000 \\ +\,0.10011010 \\ \hline 11.10111010 \end{array}$$

Renormalizing, we retain the larger exponent and truncate the low-order bit. Thus, we have:



However, because our simple model requires a normalized significand, we have no way to represent zero. This is easily remedied by allowing the string of all zeros (a zero sign, a zero exponent, and a zero significand) to represent the value zero. In the next section we will see that IEEE-754 also reserves special meaning for certain bit patterns.