CHAPTER 10

COMPUTER ARITHMETIC

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Understand the distinction between the way in which numbers are represented (the binary format) and the algorithms used for the basic arithmetic operations.
- Explain **twos complement representation**.
- Present an overview of the techniques for doing basic arithmetic operation in two complement notation.
- Understand the use of significand, base, and exponent in the representation of **floating-point numbers**.
- Present an overview of the IEEE 754 standard for floating-point representation.
- Understand some of the key concepts related to floating-point arithmetic, including guard bits, rounding, subnormal numbers, underflow and overflow.

We begin our examination of the processor with an overview of the arithmetic and logic unit (ALU). The chapter then focuses on the most complex aspect of the ALU, computer arithmetic. The implementations of simple logic and arithmetic functions in digital logic are described in Chapter 11, and logic functions that are part of the ALU are described in Chapter 12.

Computer arithmetic is commonly performed on two very different types of numbers: integer and floating point. In both cases, the representation chosen is a crucial design issue and is treated first, followed by a discussion of arithmetic operations.

This chapter includes a number of examples, each of which is highlighted in a shaded box.

10.1 THE ARITHMETIC AND LOGIC UNIT

The ALU is that part of the computer that actually performs arithmetic and logical operations on data. All of the other elements of the computer system—control unit, registers, memory, I/O—are there mainly to bring data into the ALU for it to process and then to take the results back out. We have, in a sense, reached the core or essence of a computer when we consider the ALU.

An ALU and indeed, all electronic components in the computer, are based on the use of simple digital logic devices that can store binary digits and perform simple Boolean logic operations.

Figure 10.1 indicates, in general terms, how the ALU is interconnected with the rest of the processor. Operands for arithmetic and logic operations are presented to the ALU in registers, and the results of an operation are stored in registers. These registers are temporary storage locations within the processor that are connected by signal paths to the ALU (e.g., see Figure 2.3). The ALU may also set flags as the result of an operation. For example, an overflow flag is set to 1 if the result of a computation exceeds the length of the register into which it is to be stored.

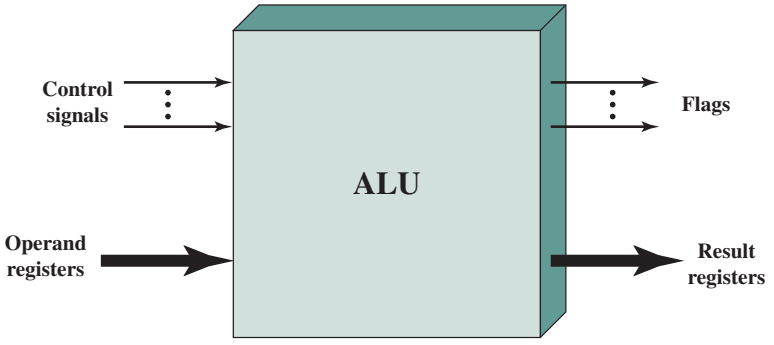


Figure 10.1 ALU Inputs and Outputs

The flag values are also stored in registers within the processor. The processor provides signals that control the operation of the ALU and the movement of the data into and out of the ALU.

10.2 INTEGER REPRESENTATION

In the binary number system,[1] arbitrary numbers can be represented with just the digits zero and one, the minus sign (for negative numbers), and the period, or **radix point** (for numbers with a fractional component).

$$-1101.0101_2 = -13.3125_{10}$$

For purposes of computer storage and processing, however, we do not have the benefit of special symbols for the minus sign and radix point. Only binary digits (0 and 1) may be used to represent numbers. If we are limited to nonnegative integers, the representation is straightforward.

An 8-bit word can represent the numbers from 0 to 255, such as

$$\begin{aligned} 00000000 &= \quad 0 \\ 00000001 &= \quad 1 \\ 00101001 &= \quad 41 \\ 10000000 &= 128 \\ 11111111 &= 255 \end{aligned}$$

In general, if an n-bit sequence of binary digits $a_{n-1}a_{n-2} \ldots a_1a_0$ is interpreted as an unsigned integer A, its value is

$$A = \sum_{i=0}^{n-1} 2^i a_i$$

[1]See Chapter 9 for a basic refresher on number systems (decimal, binary, hexadecimal).

Sign-Magnitude Representation

There are several alternative conventions used to represent negative as well as positive integers, all of which involve treating the most significant (leftmost) bit in the word as a sign bit. If the sign bit is 0, the number is positive; if the sign bit is 1, the number is negative.

The simplest form of representation that employs a sign bit is the sign-magnitude representation. In an n-bit word, the rightmost $n - 1$ bits hold the magnitude of the integer.

$$\begin{aligned} +18 &= 00010010 \\ -18 &= 10010010 \quad \text{(sign magnitude)} \end{aligned}$$

The general case can be expressed as follows:

Sign Magnitude

$$A = \begin{cases} \sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 0 \\ -\sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 1 \end{cases} \tag{10.1}$$

There are several drawbacks to sign-magnitude representation. One is that addition and subtraction require a consideration of both the signs of the numbers and their relative magnitudes to carry out the required operation. This should become clear in the discussion in Section 10.3. Another drawback is that there are two representations of 0:

$$\begin{aligned} +0_{10} &= 00000000 \\ -0_{10} &= 10000000 \quad \text{(sign magnitude)} \end{aligned}$$

This is inconvenient because it is slightly more difficult to test for 0 (an operation performed frequently on computers) than if there were a single representation.

Because of these drawbacks, sign-magnitude representation is rarely used in implementing the integer portion of the ALU. Instead, the most common scheme is twos complement representation.[2]

Twos Complement Representation

Like sign magnitude, twos complement representation uses the most significant bit as a sign bit, making it easy to test whether an integer is positive or negative. It differs from the use of the sign-magnitude representation in the way that the other bits are interpreted. Table 10.1 highlights key characteristics of twos complement representation and arithmetic, which are elaborated in this section and the next.

Most treatments of twos complement representation focus on the rules for producing negative numbers, with no formal proof that the scheme is valid. Instead,

[2]In the literature, the terms *two's complement* or *2's complement* are often used. Here we follow the practice used in standards documents and omit the apostrophe (e.g., IEEE Std 100-1992, *The New IEEE Standard Dictionary of Electrical and Electronics Terms*).

Table 10.1 Characteristics of Twos Complement Representation and Arithmetic

Range	-2^{n-1} through $2^{n-1} - 1$
Number of Representations of Zero	One
Negation	Take the Boolean complement of each bit of the corresponding positive number, then add 1 to the resulting bit pattern viewed as an unsigned integer.
Expansion of Bit Length	Add additional bit positions to the left and fill in with the value of the original sign bit.
Overflow Rule	If two numbers with the same sign (both positive or both negative) are added, then overflow occurs if and only if the result has the opposite sign.
Subtraction Rule	To subtract B from A, take the twos complement of B and add it to A.

our presentation of twos complement integers in this section and in Section 10.3 is based on [DATT93], which suggests that twos complement representation is best understood by defining it in terms of a weighted sum of bits, as we did previously for unsigned and sign-magnitude representations. The advantage of this treatment is that it does not leave any lingering doubt that the rules for arithmetic operations in twos complement notation may not work for some special cases.

Consider an n-bit integer, A, in twos complement representation. If A is positive, then the sign bit, a_{n-1}, is zero. The remaining bits represent the magnitude of the number in the same fashion as for sign magnitude:

$$A = \sum_{i=0}^{n-2} 2^i a_i \qquad \text{for } A \geq 0$$

The number zero is identified as positive and therefore has a 0 sign bit and a magnitude of all 0s. We can see that the range of positive integers that may be represented is from 0 (all of the magnitude bits are 0) through $2^{n-1} - 1$ (all of the magnitude bits are 1). Any larger number would require more bits.

Now, for a negative number $A(A < 0)$, the sign bit, a_{n-1}, is one. The remaining $n - 1$ bits can take on any one of 2^{n-1} values. Therefore, the range of negative integers that can be represented is from -1 to -2^{n-1}. We would like to assign the bit values to negative integers in such a way that arithmetic can be handled in a straightforward fashion, similar to unsigned integer arithmetic. In unsigned integer representation, to compute the value of an integer from the bit representation, the weight of the most significant bit is $+2^{n-1}$. For a representation with a sign bit, it turns out that the desired arithmetic properties are achieved, as we will see in Section 10.3, if the weight of the most significant bit is -2^{n-1}. This is the convention used in twos complement representation, yielding the following expression for negative numbers:

Twos Complement

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i \qquad \textbf{(10.2)}$$

Equation (10.2) defines the twos complement representation for both positive and negative numbers. For $a_{n-1} = 0$, the term $-2^{n-1}a_{n-1} = 0$ and the equation defines

Table 10.2 Alternative Representations for 4-Bit Integers

Decimal Representation	Sign-Magnitude Representation	Twos Complement Representation	Biased Representation
+8	—	—	1111
+7	0111	0111	1110
+6	0110	0110	1101
+5	0101	0101	1100
+4	0100	0100	1011
+3	0011	0011	1010
+2	0010	0010	1001
+1	0001	0001	1000
+0	0000	0000	0111
−0	1000	—	—
−1	1001	1111	0110
−2	1010	1110	0101
−3	1011	1101	0100
−4	1100	1100	0011
−5	1101	1011	0010
−6	1110	1010	0001
−7	1111	1001	0000
−8	—	1000	—

a nonnegative integer. When $a_{n-1} = 1$, the term 2^{n-1} is subtracted from the summation term, yielding a negative integer.

Table 10.2 compares the sign-magnitude and twos complement representations for 4-bit integers. Although twos complement is an awkward representation from the human point of view, we will see that it facilitates the most important arithmetic operations, addition and subtraction. For this reason, it is almost universally used as the processor representation for integers.

A useful illustration of the nature of twos complement representation is a value box, in which the value on the far right in the box is 1 (2^0) and each succeeding position to the left is double in value, until the leftmost position, which is negated. As you can see in Figure 10.2a, the most negative twos complement number that can be represented is -2^{n-1}; if any of the bits other than the sign bit is one, it adds a positive amount to the number. Also, it is clear that a negative number must have a 1 at its leftmost position and a positive number must have a 0 in that position. Thus, the largest positive number is a 0 followed by all 1s, which equals $2^{n-1} - 1$.

The rest of Figure 10.2 illustrates the use of the value box to convert from twos complement to decimal and from decimal to twos complement.

Range Extension

It is sometimes desirable to take an n-bit integer and store it in m bits, where $m > n$. This expansion of bit length is referred to as **range extension**, because the range of numbers that can be expressed is extended by increasing the bit length.

−128	64	32	16	8	4	2	1

(a) An eight-position twos complement value box

−128	64	32	16	8	4	2	1
1	0	0	0	0	0	1	1

−128 +2 +1 = −125

(b) Convert binary 10000011 to decimal

−128	64	32	16	8	4	2	1
1	0	0	0	1	0	0	0

−120 = −128 +8

(c) Convert decimal −120 to binary

Figure 10.2 Use of a Value Box for Conversion between Twos Complement Binary and Decimal

In sign-magnitude notation, this is easily accomplished: simply move the sign bit to the new leftmost position and fill in with zeros.

+18	=	00010010	(sign magnitude, 8 bits)
+18	=	0000000000010010	(sign magnitude, 16 bits)
−18	=	10010010	(sign magnitude, 8 bits)
−18	=	1000000000010010	(sign magnitude, 16 bits)

This procedure will not work for twos complement negative integers. Using the same example,

+18	=	00010010	(twos complement, 8 bits)
+18	=	0000000000010010	(twos complement, 16 bits)
−18	=	11101110	(twos complement, 8 bits)
−32,658	=	1000000001101110	(twos complement, 16 bits)

The next to last line is easily seen using the value box of Figure 10.2. The last line can be verified using Equation (10.2) or a 16-bit value box.

Instead, the rule for twos complement integers is to move the sign bit to the new leftmost position and fill in with copies of the sign bit. For positive numbers, fill in with zeros, and for negative numbers, fill in with ones. This is called sign extension.

−18	=	11101110	(twos complement, 8 bits)
−18	=	1111111111101110	(twos complement, 16 bits)

To see why this rule works, let us again consider an n-bit sequence of binary digits $a_{n-1}a_{n-2} \ldots a_1a_0$ interpreted as a twos complement integer A, so that its value is

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

If A is a positive number, the rule clearly works. Now, if A is negative and we want to construct an m-bit representation, with $m > n$. Then

$$A = -2^{m-1}a_{m-1} + \sum_{i=0}^{m-2} 2^i a_i$$

The two values must be equal:

$$-2^{m-1} + \sum_{i=0}^{m-2} 2^i a_i = -2^{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

$$-2^{m-1} + \sum_{i=n-1}^{m-2} 2^i a_i = -2^{n-1}$$

$$-2^{n-1} + \sum_{i=n-1}^{m-2} 2^i a_i = 2^{m-1}$$

$$1 + \sum_{i=0}^{n-2} 2^i + \sum_{i=n-1}^{m-2} 2^i a_i = 1 + \sum_{i=0}^{m-2} 2^i$$

$$\sum_{i=n-1}^{m-2} 2^i a_i = \sum_{i=n-1}^{m-2} 2^i$$

$$\Rightarrow a_{m-2} = \ldots = a_{n-2} = a_{n-2} = 1$$

In going from the first to the second equation, we require that the least significant $n - 1$ bits do not change between the two representations. Then we get to the next to last equation, which is only true if all of the bits in positions $n - 1$ through $m - 2$ are 1. Therefore, the sign-extension rule works. The reader may find the rule easier to grasp after studying the discussion on twos complement negation at the beginning of Section 10.3.

Fixed-Point Representation

Finally, we mention that the representations discussed in this section are sometimes referred to as fixed point. This is because the radix point (binary point) is fixed and assumed to be to the right of the rightmost digit. The programmer can use the same representation for binary fractions by scaling the numbers so that the binary point is implicitly positioned at some other location.

10.3 INTEGER ARITHMETIC

This section examines common arithmetic functions on numbers in twos complement representation.

Negation

In sign-magnitude representation, the rule for forming the negation of an integer is simple: invert the sign bit. In twos complement notation, the negation of an integer can be formed with the following rules:

1. Take the Boolean complement of each bit of the integer (including the sign bit). That is, set each 1 to 0 and each 0 to 1.
2. Treating the result as an unsigned binary integer, add 1.

This two-step process is referred to as the **twos complement operation**, or the taking of the twos complement of an integer.

$$\begin{array}{rcll} +18 & = & 00010010 & \text{(twos complement)} \\ \text{bitwise complement} & = & 11101101 & \\ & & \underline{+\qquad 1} & \\ & & 11101110 = -18 & \end{array}$$

As expected, the negative of the negative of that number is itself:

$$\begin{array}{rcll} -18 & = & 11101110 & \text{(twos complement)} \\ \text{bitwise complement} & = & 00010001 & \\ & & \underline{+\qquad 1} & \\ & & 00010010 = +18 & \end{array}$$

We can demonstrate the validity of the operation just described using the definition of the twos complement representation in Equation (10.2). Again, interpret an n-bit sequence of binary digits $a_{n-1}a_{n-2} \ldots a_1a_0$ as a twos complement integer A, so that its value is

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

Now form the bitwise complement, $\overline{a_{n-1}a_{n-2}} \ldots \overline{a_0}$, and, treating this as an unsigned integer, add 1. Finally, interpret the resulting n-bit sequence of binary digits as a twos complement integer B, so that its value is

$$B = -2^{n-1}\overline{a_{n-1}} + 1 + \sum_{i=0}^{n-2} 2^i \overline{a_i}$$

Now, we want $A = -B$, which means $A + B = 0$. This is easily shown to be true:

$$\begin{aligned} A + B &= -(a_{n-1} + \overline{a_{n-1}})2^{n-1} + 1 + \left(\sum_{i=0}^{n-2} 2^i (a_i + \overline{a_i})\right) \\ &= -2^{n-1} + 1 + \left(\sum_{i=0}^{n-2} 2^i\right) \\ &= -2^{n-1} + 1 + (2^{n-1} - 1) \\ &= -2^{n-1} + 2^{n-1} = 0 \end{aligned}$$

The preceding derivation assumes that we can first treat the bitwise complement of A as an unsigned integer for the purpose of adding 1, and then treat the result as a twos complement integer. There are two special cases to consider. First, consider $A = 0$. In that case, for an 8-bit representation:

$$\begin{array}{rcl} 0 & = & 00000000 \quad \text{(twos complement)} \\ \text{bitwise complement} & = & 11111111 \\ & & \underline{+\qquad 1} \\ & & 100000000 = 0 \end{array}$$

There is a *carry* out of the most significant bit position, which is ignored. The result is that the negation of 0 is 0, as it should be.

The second special case is more of a problem. If we take the negation of the bit pattern of 1 followed by $n - 1$ zeros, we get back the same number. For example, for 8-bit words,

$$\begin{array}{rcl} +128 & = & 10000000 \quad \text{(twos complement)} \\ \text{bitwise complement} & = & 01111111 \\ & & \underline{+\qquad 1} \\ & & 10000000 = -128 \end{array}$$

Some such anomaly is unavoidable. The number of different bit patterns in an n-bit word is $2n$, which is an even number. We wish to represent positive and negative integers and 0. If an equal number of positive and negative integers are represented (sign magnitude), then there are two representations for 0. If there is only one representation of 0 (twos complement), then there must be an unequal number of negative and positive numbers represented. In the case of twos complement, for an n-bit length, there is a representation for -2^{n-1} but not for $+2^{n-1}$.

Addition and Subtraction

Addition in twos complement is illustrated in Figure 10.3. Addition proceeds as if the two numbers were unsigned integers. The first four examples illustrate successful operations. If the result of the operation is positive, we get a positive number in twos complement form, which is the same as in unsigned-integer form. If the result of the operation is negative, we get a negative number in twos complement form. Note that, in some instances, there is a carry bit beyond the end of the word (indicated by shading), which is ignored.

On any addition, the result may be larger than can be held in the word size being used. This condition is called **overflow**. When overflow occurs, the ALU must signal this fact so that no attempt is made to use the result. To detect overflow, the following rule is observed:

OVERFLOW RULE: If two numbers are added, and they are both positive or both negative, then overflow occurs if and only if the result has the opposite sign.

1001 = −7 +0101 = 5 1110 = −2 (a) (−7) + (+5)	1100 = −4 +0100 = 4 10000 = 0 (b) (−4) + (+4)
0011 = 3 +0100 = 4 0111 = 7 (c) (+3) + (+4)	1100 = −4 +1111 = −1 11011 = −5 (d) (−4) + (−1)
0101 = 5 +0100 = 4 1001 = Overflow (e) (+5) + (+4)	1001 = −7 +1010 = −6 10011 = Overflow (f) (−7) + (−6)

Figure 10.3 Addition of Numbers in Twos Complement Representation

Figures 10.3e and f show examples of overflow. Note that overflow can occur whether or not there is a carry.

Subtraction is easily handled with the following rule:

SUBTRACTION RULE: To subtract one number (subtrahend) from another (minuend), take the twos complement (negation) of the subtrahend and add it to the minuend.

Thus, subtraction is achieved using addition, as illustrated in Figure 10.4. The last two examples demonstrate that the overflow rule still applies.

0010 = 2 +1001 = −7 1011 = −5 (a) M = 2 = 0010 S = 7 = 0111 −S = 1001	0101 = 5 +1110 = −2 10011 = 3 (b) M = 5 = 0101 S = 2 = 0010 −S = 1110
1011 = −5 +1110 = −2 11001 = −7 (c) M = −5 = 1011 S = 2 = 0010 −S = 1110	0101 = 5 +0010 = 2 0111 = 7 (d) M = 5 = 0101 S = −2 = 1110 −S = 0010
0111 = 7 +0111 = 7 1110 = Overflow (e) M = 7 = 0111 S = −7 = 1001 −S = 0111	1010 = −6 +1100 = −4 10110 = Overflow (f) M = −6 = 1010 S = 4 = 0100 −S = 1100

Figure 10.4 Subtraction of Numbers in Twos Complement Representation (M − S)

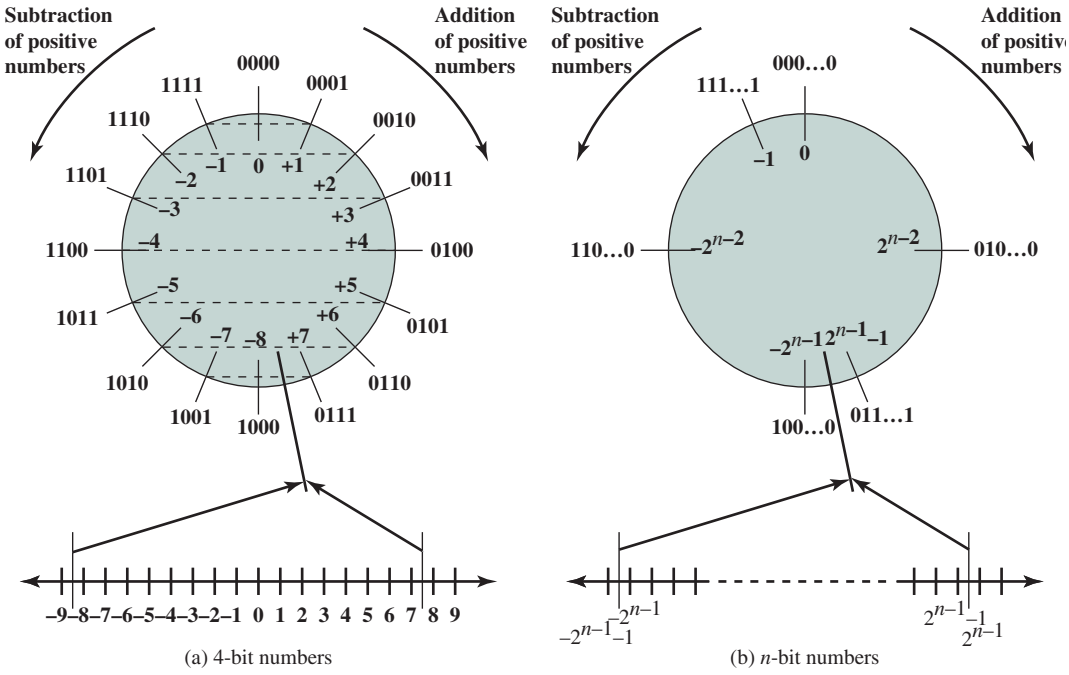


Figure 10.5 Geometric Depiction of Twos Complement Integers

Some insight into twos complement addition and subtraction can be gained by looking at a geometric depiction [BENH92], as shown in Figure 10.5. The circle in the upper half of each part of the figure is formed by selecting the appropriate segment of the number line and joining the endpoints. Note that when the numbers are laid out on a circle, the twos complement of any number is horizontally opposite that number (indicated by dashed horizontal lines). Starting at any number on the circle, we can add positive k (or subtract negative k) to that number by moving k positions clockwise, and we can subtract positive k (or add negative k) from that number by moving k positions counterclockwise. If an arithmetic operation results in traversal of the point where the endpoints are joined, an incorrect answer is given (overflow).

ALL OF the examples of Figures 10.3 and 10.4 are easily traced in the circle of Figure 10.5.

Figure 10.6 suggests the data paths and hardware elements needed to accomplish addition and subtraction. The central element is a binary adder, which is presented two numbers for addition and produces a sum and an overflow indication. The binary adder treats the two numbers as unsigned integers. (A logic implementation of an adder is given in Chapter 11.) For addition, the two numbers are presented to the adder from two registers, designated in this case as **A** and **B** registers. The result may be stored in one of these registers or in a third. The overflow indication is stored in a 1-bit overflow flag (0 = no overflow; 1 = overflow). For subtraction, the subtrahend (**B** register) is passed through a twos complementer so that its twos complement is presented to the adder. Note that Figure 10.6 only shows the

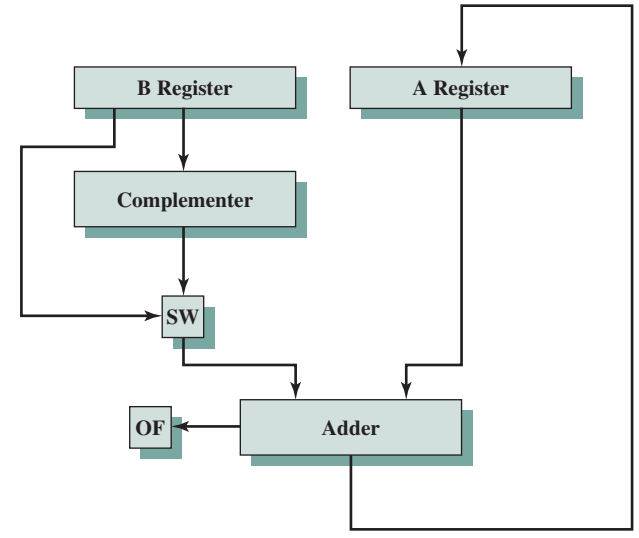


Figure 10.6 Block Diagram of Hardware for Addition and Subtraction

data paths. Control signals are needed to control whether or not the complementer is used, depending on whether the operation is addition or subtraction.

Multiplication

Compared with addition and subtraction, multiplication is a complex operation, whether performed in hardware or software. A wide variety of algorithms have been used in various computers. The purpose of this subsection is to give the reader some feel for the type of approach typically taken. We begin with the simpler problem of multiplying two unsigned (nonnegative) integers, and then we look at one of the most common techniques for multiplication of numbers in twos complement representation.

UNSIGNED INTEGERS Figure 10.7 illustrates the multiplication of unsigned binary integers, as might be carried out using paper and pencil. Several important observations can be made:

1. Multiplication involves the generation of partial products, one for each digit in the multiplier. These partial products are then summed to produce the final product.

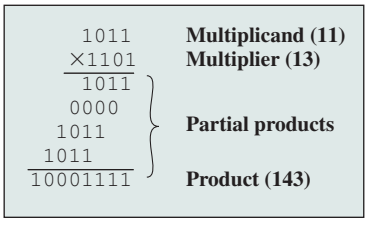
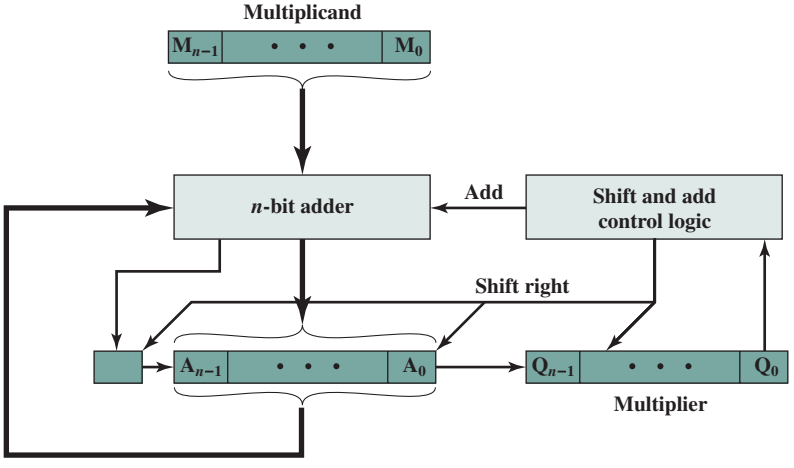


Figure 10.7 Multiplication of Unsigned Binary Integers

2. The partial products are easily defined. When the multiplier bit is 0, the partial product is 0. When the multiplier is 1, the partial product is the multiplicand.
3. The total product is produced by summing the partial products. For this operation, each successive partial product is shifted one position to the left relative to the preceding partial product.
4. The multiplication of two n-bit binary integers results in a product of up to $2n$ bits in length (e.g., $11 \times 11 = 1001$).

Compared with the pencil-and-paper approach, there are several things we can do to make computerized multiplication more efficient. First, we can perform a running addition on the partial products rather than waiting until the end. This eliminates the need for storage of all the partial products; fewer registers are needed. Second, we can save some time on the generation of partial products. For each 1 on the multiplier, an add and a shift operation are required; but for each 0, only a shift is required.

Figure 10.8a shows a possible implementation employing these measures. The multiplier and multiplicand are loaded into two registers (Q and M). A third



(a) Block diagram

C	A	Q	M		
0	0000	1101	1011	**Initial values**	
0	1011	1101	1011	**Add**	**First cycle**
0	0101	1110	1011	**Shift**	
0	0010	1111	1011	**Shift**	**Second cycle**
0	1101	1111	1011	**Add**	**Third cycle**
0	0110	1111	1011	**Shift**	
1	0001	1111	1011	**Add**	**Fourth cycle**
0	1000	1111	1011	**Shift**	

(b) Example from Figure 10.7 (product in A, Q)

Figure 10.8 Hardware Implementation of Unsigned Binary Multiplication

register, the A register, is also needed and is initially set to 0. There is also a 1-bit C register, initialized to 0, which holds a potential carry bit resulting from addition.

The operation of the multiplier is as follows. Control logic reads the bits of the multiplier one at a time. If Q_0 is 1, then the multiplicand is added to the A register and the result is stored in the A register, with the C bit used for overflow. Then all of the bits of the C, A, and Q registers are shifted to the right one bit, so that the C bit goes into A_{n-1}, A_0 goes into Q_{n-1}, and Q_0 is lost. If Q_0 is 0, then no addition is performed, just the shift. This process is repeated for each bit of the original multiplier. The resulting $2n$-bit product is contained in the A and Q registers. A flowchart of the operation is shown in Figure 10.9, and an example is given in Figure 10.8b. Note that on the second cycle, when the multiplier bit is 0, there is no add operation.

TWOS COMPLEMENT MULTIPLICATION We have seen that addition and subtraction can be performed on numbers in twos complement notation by treating them as unsigned integers. Consider

$$\begin{array}{r} 1001 \\ +\,0011 \\ \hline 1100 \end{array}$$

If these numbers are considered to be unsigned integers, then we are adding 9 (1001) plus 3 (0011) to get 12 (1100). As twos complement integers, we are adding -7(1001) to 3 (0011) to get -4(1100).

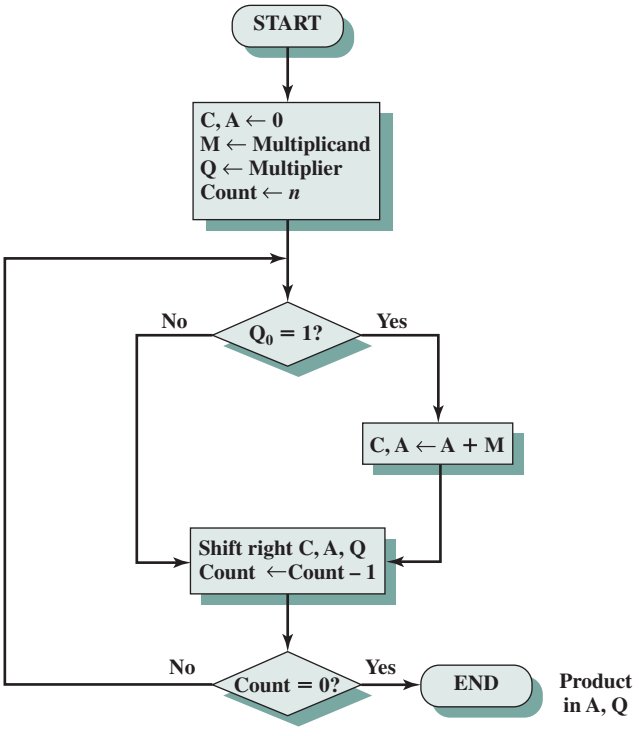


Figure 10.9 Flowchart for Unsigned Binary Multiplication

```
      1011
    × 1101
  --------
  00001011   1011 × 1 × 2^0
  00000000   1011 × 0 × 2^1
  00101100   1011 × 1 × 2^2
  01011000   1011 × 1 × 2^3
  --------
  10001111
```

Figure 10.10 Multiplication of Two Unsigned 4-Bit Integers Yielding an 8-Bit Result

Unfortunately, this simple scheme will not work for multiplication. To see this, consider again Figure 10.7. We multiplied 11 (1011) by 13 (1101) to get 143 (10001111). If we interpret these as twos complement numbers, we have −5(1011) times −3 (1101) equals −113 (10001111). This example demonstrates that straightforward multiplication will not work if both the multiplicand and multiplier are negative. In fact, it will not work if either the multiplicand or the multiplier is negative. To justify this statement, we need to go back to Figure 10.7 and explain what is being done in terms of operations with powers of 2. Recall that any unsigned binary number can be expressed as a sum of powers of 2. Thus,

$$1101 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 2^3 + 2^2 + 2^0$$

Further, the multiplication of a binary number by 2^n is accomplished by shifting that number to the left n bits. With this in mind, Figure 10.10 recasts Figure 10.7 to make the generation of partial products by multiplication explicit. The only difference in Figure 10.10 is that it recognizes that the partial products should be viewed as $2n$-bit numbers generated from the n-bit multiplicand.

Thus, as an unsigned integer, the 4-bit multiplicand 1011 is stored in an 8-bit word as 00001011. Each partial product (other than that for 2^0) consists of this number shifted to the left, with the unoccupied positions on the right filled with zeros (e.g., a shift to the left of two places yields 00101100).

Now we can demonstrate that straightforward multiplication will not work if the multiplicand is negative. The problem is that each contribution of the negative multiplicand as a partial product must be a negative number on a $2n$-bit field; the sign bits of the partial products must line up. This is demonstrated in Figure 10.11, which shows that multiplication of 1001 by 0011. If these are treated as unsigned integers, the multiplication of $9 \times 3 = 27$ proceeds simply. However, if 1001 is interpreted

```
    1001  (9)                  1001  (-7)
  × 0011  (3)                × 0011  (3)
  --------                   --------
  00001001  1001 × 2^0       11111001  (-7) × 2^0 = (-7)
  00010010  1001 × 2^1       11110010  (-7) × 2^1 = (-14)
  --------                   --------
  00011011  (27)             11101011  (-21)
```

(a) Unsigned integers (b) Twos complement integers

Figure 10.11 Comparison of Multiplication of Unsigned and Twos Complement Integers

as the twos complement value −7, then each partial product must be a negative twos complement number of 2*n* (8) bits, as shown in Figure 10.11b. Note that this is accomplished by padding out each partial product to the left with binary 1s.

If the multiplier is negative, straightforward multiplication also will not work. The reason is that the bits of the multiplier no longer correspond to the shifts or multiplications that must take place. For example, the 4-bit decimal number −3 is written 1101 in twos complement. If we simply took partial products based on each bit position, we would have the following correspondence:

$$1101 \leftrightarrow -(1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0) = -(2^3 + 2^2 + 2^0)$$

In fact, what is desired is $-(2^1 + 2^0)$. So this multiplier cannot be used directly in the manner we have been describing.

There are a number of ways out of this dilemma. One would be to convert both multiplier and multiplicand to positive numbers, perform the multiplication, and then take the twos complement of the result if and only if the sign of the two original numbers differed. Implementers have preferred to use techniques that do not require this final transformation step. One of the most common of these is Booth's algorithm [BOOT51]. This algorithm also has the benefit of speeding up the multiplication process, relative to a more straightforward approach.

Booth's algorithm is depicted in Figure 10.12 and can be described as follows. As before, the multiplier and multiplicand are placed in the Q and M registers,

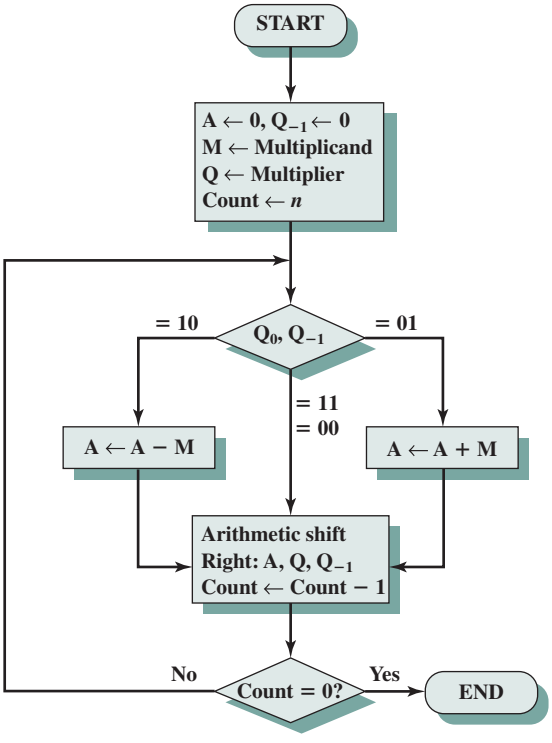


Figure 10.12 Booth's Algorithm for Twos Complement Multiplication

A	Q	Q_{-1}	M		
0000	0011	0	0111	**Initial values**	
1001	0011	0	0111	**A←A – M**	**First cycle**
1100	1001	1	0111	**Shift**	
1110	0100	1	0111	**Shift**	**Second cycle**
0101	0100	1	0111	**A←A + M**	**Third cycle**
0010	1010	0	0111	**Shift**	
0001	0101	0	0111	**Shift**	**Fourth cycle**

Figure 10.13 Example of Booth's Algorithm (7 × 3)

respectively. There is also a 1-bit register placed logically to the right of the least significant bit (Q_0) of the Q register and designated Q_{-1}; its use is explained shortly. The results of the multiplication will appear in the A and Q registers. A and Q_{-1} are initialized to 0. As before, control logic scans the bits of the multiplier one at a time. Now, as each bit is examined, the bit to its right is also examined. If the two bits are the same (1–1 or 0–0), then all of the bits of the A, Q, and Q_{-1} registers are shifted to the right 1 bit. If the two bits differ, then the multiplicand is added to or subtracted from the A register, depending on whether the two bits are 0–1 or 1–0. Following the addition or subtraction, the right shift occurs. In either case, the right shift is such that the leftmost bit of A, namely A_{n-1}, not only is shifted into A_{n-2}, but also remains in A_{n-1}. This is required to preserve the sign of the number in A and Q. It is known as an **arithmetic shift**, because it preserves the sign bit.

Figure 10.13 shows the sequence of events in Booth's algorithm for the multiplication of 7 by 3. More compactly, the same operation is depicted in Figure 10.14a. The rest of Figure 10.14 gives other examples of the algorithm. As can be seen, it works with any combination of positive and negative numbers. Note also the efficiency of the algorithm. Blocks of 1s or 0s are skipped over, with an average of only one addition or subtraction per block.

```
   0111
 × 0011    (0)
11111001   1-0
0000000    1-1
000111     0-1
00010101   (21)
```

(a) (7) × (3) = (21)

```
   0111
 × 1101    (0)
11111001   1-0
0000111    0-1
111001     1-0
11101011   (-21)
```

(b) (7) × (−3) = (−21)

```
   1001
 × 0011    (0)
00000111   1-0
0000000    1-1
111001     0-1
11101011   (-21)
```

(c) (−7) × (3) = (−21)

```
   1001
 × 1101    (0)
00000111   1-0
1111001    0-1
000111     1-0
00010101   (21)
```

(d) (−7) × (−3) = (21)

Figure 10.14 Examples Using Booth's Algorithm

Why does Booth's algorithm work? Consider first the case of a positive multiplier. In particular, consider a positive multiplier consisting of one block of 1s surrounded by 0s (e.g., 00011110). As we know, multiplication can be achieved by adding appropriately shifted copies of the multiplicand:

$$\begin{aligned} \text{M} \times (00011110) &= \text{M} \times (2^4 + 2^3 + 2^2 + 2^1) \\ &= \text{M} \times (16 + 8 + 4 + 2) \\ &= \text{M} \times 30 \end{aligned}$$

The number of such operations can be reduced to two if we observe that

$$2^n + 2^{n-1} + \cdots + 2^{n-K} = 2^{n+1} - 2^{n-K} \tag{10.3}$$

$$\begin{aligned} \text{M} \times (00011110) &= \text{M} \times (2^5 - 2^1) \\ &= \text{M} \times (32 - 2) \\ &= \text{M} \times 30 \end{aligned}$$

So the product can be generated by one addition and one subtraction of the multiplicand. This scheme extends to any number of blocks of 1s in a multiplier, including the case in which a single 1 is treated as a block.

$$\begin{aligned} \text{M} \times (01111010) &= \text{M} \times (2^6 + 2^5 + 2^4 + 2^3 + 2^1) \\ &= \text{M} \times (2^7 - 2^3 + 2^2 - 2^1) \end{aligned}$$

Booth's algorithm conforms to this scheme by performing a subtraction when the first 1 of the block is encountered (1–0) and an addition when the end of the block is encountered (0–1).

To show that the same scheme works for a negative multiplier, we need to observe the following. Let X be a negative number in twos complement notation:

$$\text{Representation of } X = \{1x_{n-2}x_{n-3} \ldots x_1x_0\}$$

Then the value of X can be expressed as follows:

$$X = -2^{n-1} + (x_{n-2} \times 2^{n-2}) + (x_{n-3} \times 2^{n-3}) + \cdots (x_1 \times 2^1) + (x_0 \times 2^0) \tag{10.4}$$

The reader can verify this by applying the algorithm to the numbers in Table 10.2.

The leftmost bit of X is 1, because X is negative. Assume that the leftmost 0 is in the kth position. Thus, X is of the form

$$\text{Representation of } X = \{111 \ldots 10x_{k-1}x_{k-2} \ldots x_1x_0\} \tag{10.5}$$

Then the value of X is

$$X = -2^{n-1} + 2^{n-2} + \cdots + 2^{k+1} + (x_{k-1} \times 2^{k-1}) + \cdots + (x_0 \times 2^0) \tag{10.6}$$

From Equation (10.3), we can say that

$$2^{n-2} + 2^{n-3} + \cdots + 2^{k-1} = 2^{n-1} - 2^{k-1}$$

Rearranging

$$-2^{n-1} + 2^{n-2} + 2^{n-3} + \cdots + 2^{k+1} = -2^{k+1} \quad \textbf{(10.7)}$$

Substituting Equation (10.7) into Equation (10.6), we have

$$X = -2^{k+1} + (x_{k-1} \times 2^{k-1}) + \cdots + (x_0 \times 2^0) \quad \textbf{(10.8)}$$

At last we can return to Booth's algorithm. Remembering the representation of X [Equation (10.5)], it is clear that all of the bits from x_0 up to the leftmost 0 are handled properly because they produce all of the terms in Equation (10.8) but (-2^{k+1}) and thus are in the proper form. As the algorithm scans over the leftmost 0 and encounters the next 1 (2^{k+1}), a 1–0 transition occurs and a subtraction takes place (-2^{k+1}). This is the remaining term in Equation (10.8).

As an example, consider the multiplication of some multiplicand by (-6). In twos complement representation, using an 8-bit word, (-6) is represented as 11111010. By Equation (10.4), we know that

$$-6 = -2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^1$$

which the reader can easily verify. Thus,

$$\text{M} \times (11111010) = \text{M} \times (-2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^1)$$

Using Equation (10.7),

$$\text{M} \times (11111010) = \text{M} \times (-2^3 + 2^1)$$

which the reader can verify is still $\text{M} \times (-6)$. Finally, following our earlier line of reasoning,

$$\text{M} \times (11111010) = \text{M} \times (-2^3 + 2^2 - 2^1)$$

We can see that Booth's algorithm conforms to this scheme. It performs a subtraction when the first 1 is encountered (10), an addition when (01) is encountered, and finally another subtraction when the first 1 of the next block of 1s is encountered. Thus, Booth's algorithm performs fewer additions and subtractions than a more straightforward algorithm.

Division

Division is somewhat more complex than multiplication but is based on the same general principles. As before, the basis for the algorithm is the paper-and-pencil approach, and the operation involves repetitive shifting and addition or subtraction.

Figure 10.15 shows an example of the long division of unsigned binary integers. It is instructive to describe the process in detail. First, the bits of the dividend are examined from left to right, until the set of bits examined represents a number greater than or equal to the divisor; this is referred to as the divisor being able to divide the number. Until this event occurs, 0s are placed in the quotient from left to right. When the event occurs, a 1 is placed in the quotient and the divisor is subtracted from the partial dividend. The result is referred to as a *partial remainder.*

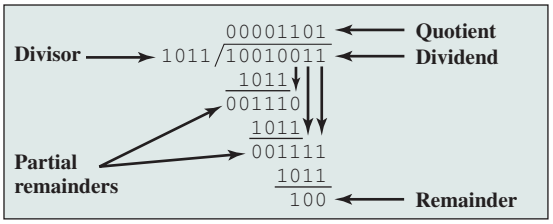


Figure 10.15 Example of Division of Unsigned Binary Integers

From this point on, the division follows a cyclic pattern. At each cycle, additional bits from the dividend are appended to the partial remainder until the result is greater than or equal to the divisor. As before, the divisor is subtracted from this number to produce a new partial remainder. The process continues until all the bits of the dividend are exhausted.

Figure 10.16 shows a machine algorithm that corresponds to the long division process. The divisor is placed in the M register, the dividend in the Q register. At

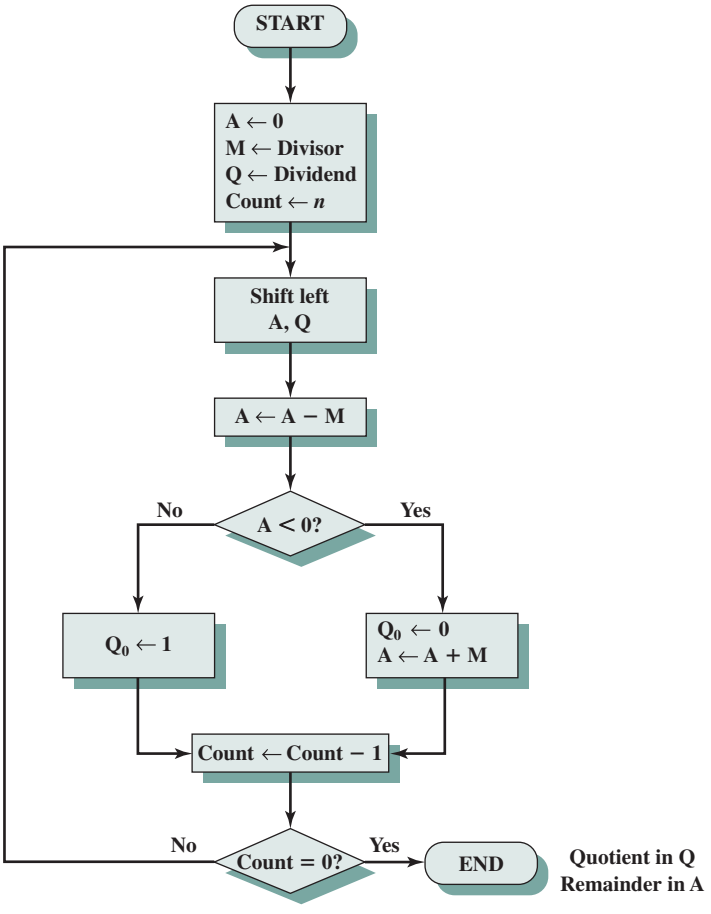


Figure 10.16 Flowchart for Unsigned Binary Division

A	Q	
0000	0111	**Initial value**
0000	1110	**Shift**
1101		**Use twos complement of 0011 for subtraction**
1101		**Subtract**
0000	1110	**Restore, set $Q_0 = 0$**
0001	1100	**Shift**
1101		
1110		**Subtract**
0001	1100	**Restore, set $Q_0 = 0$**
0011	1000	**Shift**
1101		
0000	1001	**Subtract, set $Q_0 = 1$**
0001	0010	**Shift**
1101		
1110		**Subtract**
0001	0010	**Restore, set $Q_0 = 0$**

Figure 10.17 Example of Restoring Twos Complement Division (7/3)

each step, the A and Q registers together are shifted to the left 1 bit. M is subtracted from A to determine whether A divides the partial remainder.[3] If it does, then Q_0 gets a 1 bit. Otherwise, Q_0 gets a 0 bit and M must be added back to A to restore the previous value. The count is then decremented, and the process continues for n steps. At the end, the quotient is in the Q register and the remainder is in the A register.

This process can, with some difficulty, be extended to negative numbers. We give here one approach for twos complement numbers. An example of this approach is shown in Figure 10.17.

The algorithm assumes that the divisor V and the dividend D are positive and that $|V| < |D|$. If $|V| = |D|$, then the quotient $Q = 1$ and the remainder $R = 0$. If $|V| > |D|$, then $Q = 0$ and $R = D$. The algorithm can be summarized as follows:

1. Load the twos complement of the divisor into the M register; that is, the M register contains the negative of the divisor. Load the dividend into the A, Q registers. The dividend must be expressed as a $2n$-bit positive number. Thus, for example, the 4-bit 0111 becomes 00000111.
2. Shift A, Q left 1 bit position.
3. Perform $A \leftarrow A - M$. This operation subtracts the divisor from the contents of A.
4. **a.** If the result is nonnegative (most significant bit of $A = 0$), then set $Q_0 \leftarrow 1$.
   b. If the result is negative (most significant bit of $A = 1$), then set $Q_0 \leftarrow 0$. and restore the previous value of A.
5. Repeat steps 2 through 4 as many times as there are bit positions in Q.
6. The remainder is in A and the quotient is in Q.

[3]This is subtraction of unsigned integers. A result that requires a borrow out of the most significant bit is a negative result.

To deal with negative numbers, we recognize that the remainder is defined by

$$D = Q \times V + R$$

That is, the remainder is the value of R needed for the preceding equation to be valid. Consider the following examples of integer division with all possible combinations of signs of D and V:

$$\begin{array}{llllll} D = 7 & V = 3 & \Rightarrow & Q = 2 & R = 1 \\ D = 7 & V = -3 & \Rightarrow & Q = -2 & R = 1 \\ D = -7 & V = 3 & \Rightarrow & Q = -2 & R = -1 \\ D = -7 & V = -3 & \Rightarrow & Q = 2 & R = -1 \end{array}$$

The reader will note from Figure 10.17 that $(-7)/(3)$ and $(7)/(-3)$ produce different remainders. We see that the magnitudes of Q and R are unaffected by the input signs and that the signs of Q and R are easily derivable from the signs of D and V. Specifically, $\text{sign}(R) = \text{sign}(D)$ and $\text{sign}(Q) = \text{sign}(D) \times \text{sign}(V)$. Hence, one way to do twos complement division is to convert the operands into unsigned values and, at the end, to account for the signs by complementation where needed. This is the method of choice for the restoring division algorithm [PARH10].

10.4 FLOATING-POINT REPRESENTATION

Principles

With a fixed-point notation (e.g., twos complement) it is possible to represent a range of positive and negative integers centered on or near 0. By assuming a fixed binary or radix point, this format allows the representation of numbers with a fractional component as well.

This approach has limitations. Very large numbers cannot be represented, nor can very small fractions. Furthermore, the fractional part of the quotient in a division of two large numbers could be lost.

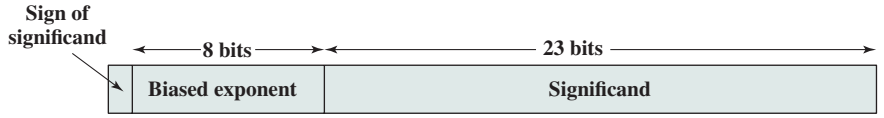
For decimal numbers, we get around this limitation by using scientific notation. Thus, 976,000,000,000,000 can be represented as 9.76×10^{14}, and 0.0000000000000976 can be represented as 9.76×10^{-14}, What we have done, in effect, is dynamically to slide the decimal point to a convenient location and use the exponent of 10 to keep track of that decimal point. This allows a range of very large and very small numbers to be represented with only a few digits.

This same approach can be taken with binary numbers. We can represent a number in the form

$$\pm \mathrm{S} \times \mathrm{B}^{\pm \mathrm{E}}$$

This number can be stored in a binary word with three fields:

- Sign: plus or minus
- Significand S
- Exponent E



(a) Format

$$
\begin{aligned}
1.1010001 \times 2^{10100} &= 0\ 10010011\ 10100010000000000000000 = 1.6328125 \times 2^{20} \\
-1.1010001 \times 2^{10100} &= 1\ 10010011\ 10100010000000000000000 = -1.6328125 \times 2^{20} \\
1.1010001 \times 2^{-10100} &= 0\ 01101011\ 10100010000000000000000 = 1.6328125 \times 2^{-20} \\
-1.1010001 \times 2^{-10100} &= 1\ 01101011\ 10100010000000000000000 = -1.6328125 \times 2^{-20}
\end{aligned}
$$

(b) Examples

Figure 10.18 Typical 32-Bit Floating-Point Format

The **base** B is implicit and need not be stored because it is the same for all numbers. Typically, it is assumed that the radix point is to the right of the leftmost, or most significant, bit of the significand. That is, there is one bit to the left of the radix point.

The principles used in representing binary floating-point numbers are best explained with an example. Figure 10.18a shows a typical 32-bit floating-point format. The leftmost bit stores the **sign** of the number (0 = positive, 1 = negative). The **exponent** value is stored in the next 8 bits. The representation used is known as a **biased representation**. A fixed value, called the bias, is subtracted from the field to get the true exponent value. Typically, the bias equals $(2^{k-1} - 1)$, where k is the number of bits in the binary exponent. In this case, the 8-bit field yields the numbers 0 through 255. With a bias of 127 $(2^7 - 1)$, the true exponent values are in the range -127 to $+128$. In this example, the base is assumed to be 2.

Table 10.2 shows the biased representation for 4-bit integers. Note that when the bits of a biased representation are treated as unsigned integers, the relative magnitudes of the numbers do not change. For example, in both biased and unsigned representations, the largest number is 1111 and the smallest number is 0000. This is not true of sign-magnitude or twos complement representation. An advantage of biased representation is that nonnegative floating-point numbers can be treated as integers for comparison purposes.

The final portion of the word (23 bits in this case) is the **significand**.[4]

Any floating-point number can be expressed in many ways.

The following are equivalent, where the significand is expressed in binary form:

$$0.110 \times 2^5$$
$$110 \times 2^2$$
$$0.0110 \times 2^6$$

To simplify operations on floating-point numbers, it is typically required that they be normalized. A **normal number** is one in which the most significant digit of the

[4]The term **mantissa**, sometimes used instead of *significand,* is considered obsolete. *Mantissa* also means "the fractional part of a logarithm," so is best avoided in this context.

significand is nonzero. For base 2 representation, a normal number is therefore one in which the most significant bit of the significand is one. As was mentioned, the typical convention is that there is one bit to the left of the radix point. Thus, a normal nonzero number is one in the form

$$\pm 1.\text{bbb} \ldots \text{ b} \times 2^{\pm \text{E}}$$

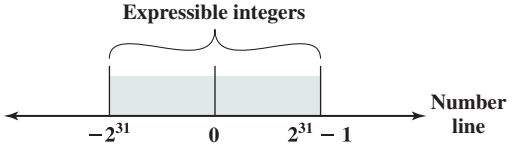
where b is either binary digit (0 or 1). Because the most significant bit is always one, it is unnecessary to store this bit; rather, it is implicit. Thus, the 23-bit field is used to store a 24-bit significand with a value in the half open interval [1, 2). Given a number that is not normal, the number may be normalized by shifting the radix point to the right of the leftmost 1 bit and adjusting the exponent accordingly.

Figure 10.18b gives some examples of numbers stored in this format. For each example, on the left is the binary number; in the center is the corresponding bit pattern; on the right is the decimal value. Note the following features:

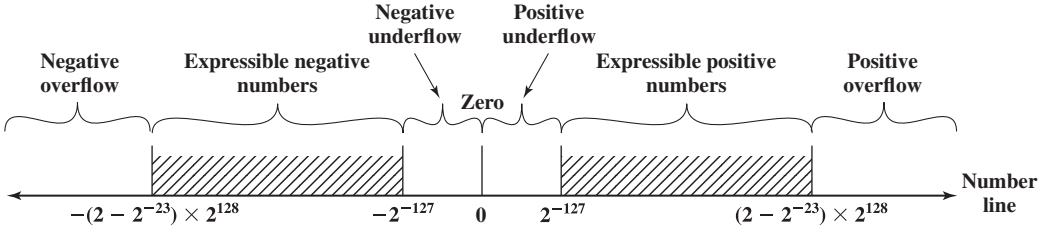
- The sign is stored in the first bit of the word.
- The first bit of the true significand is always 1 and need not be stored in the significand field.
- The value 127 is added to the true exponent to be stored in the exponent field.
- The base is 2.

For comparison, Figure 10.19 indicates the range of numbers that can be represented in a 32-bit word. Using twos complement integer representation, all of the integers from -2^{31} to $2^{31} - 1$ can be represented, for a total of 2^{32} different numbers. With the example floating-point format of Figure 10.18, the following ranges of numbers are possible:

- Negative numbers between $-(2 - 2^{-23}) \times 2^{128}$ and -2^{-127}
- Positive numbers between 2^{-127} and $(2 - 2^{-23}) \times 2^{128}$



(a) Twos complement integers



(b) Floating-point numbers

Figure 10.19 Expressible Numbers in Typical 32-Bit Formats

Five regions on the number line are not included in these ranges:

- Negative numbers less than $-(2 - 2^{-23}) \times 2^{128}$, called **negative overflow**
- Negative numbers greater than 2^{-127}, called **negative underflow**
- Zero
- Positive numbers less than 2^{-127}, called **positive underflow**
- Positive numbers greater than $(2 - 2^{-23}) \times 2^{128}$, called **positive overflow**

The representation as presented will not accommodate a value of 0. However, as we shall see, actual floating-point representations include a special bit pattern to designate zero. Overflow occurs when an arithmetic operation results in an absolute value greater than can be expressed with an exponent of 128 (e.g., $2^{120} \times 2^{100} = 2^{220}$). Underflow occurs when the fractional magnitude is too small (e.g., $2^{-120} \times 2^{-100} = 2^{-220}$). Underflow is a less serious problem because the result can generally be satisfactorily approximated by 0.

It is important to note that we are not representing more individual values with floating-point notation. The maximum number of different values that can be represented with 32 bits is still 2^{32}. What we have done is to spread those numbers out in two ranges, one positive and one negative. In practice, most floating-point numbers that one would wish to represent are represented only approximately. However, for moderate sized integers, the representation is exact.

Also, note that the numbers represented in floating-point notation are not spaced evenly along the number line, as are fixed-point numbers. The possible values get closer together near the origin and farther apart as you move away, as shown in Figure 10.20. This is one of the trade-offs of floating-point math: Many calculations produce results that are not exact and have to be rounded to the nearest value that the notation can represent.

In the type of format depicted in Figure 10.18, there is a trade-off between range and precision. The example shows 8 bits devoted to the exponent and 23 to the significand. If we increase the number of bits in the exponent, we expand the range of expressible numbers. But because only a fixed number of different values can be expressed, we have reduced the density of those numbers and therefore the precision. The only way to increase both range and precision is to use more bits. Thus, most computers offer, at least, single-precision numbers and doubleprecision numbers. For example, a processor could support a single-precision format of 64 bits, and a double-precision format of 128 bits.

So there is a trade-off between the number of bits in the exponent and the number of bits in the significand. But it is even more complicated than that. The implied base of the exponent need not be 2. The IBM S/390 architecture, for example, uses a base of 16 [ANDE67b]. The format consists of a 7-bit exponent and a 24-bit significand.

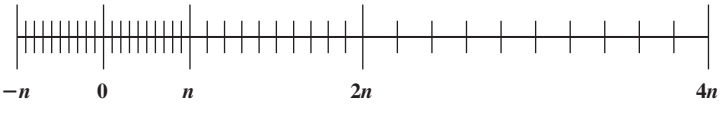


Figure 10.20 Density of Floating-Point Numbers

In the IBM base-16 format,

$$0.11010001 \times 2^{10100} = 0.11010001 \times 16^{101}$$

and the exponent is stored to represent 5 rather than 20.

The advantage of using a larger exponent is that a greater range can be achieved for the same number of exponent bits. But remember, we have not increased the number of different values that can be represented. Thus, for a fixed format, a larger exponent base gives a greater range at the expense of less precision.

IEEE Standard for Binary Floating-Point Representation

The most important floating-point representation is defined in IEEE Standard 754, adopted in 1985 and revised in 2008. This standard was developed to facilitate the portability of programs from one processor to another and to encourage the development of sophisticated, numerically oriented programs. The standard has been widely adopted and is used on virtually all contemporary processors and arithmetic coprocessors. IEEE 754-2008 covers both binary and decimal floating-point representations. In this chapter, we deal only with binary representations.

IEEE 754-2008 defines the following different types of floating-point formats:

- **Arithmetic format:** All the mandatory operations defined by the standard are supported by the format. The format may be used to represent floating-point operands or results for the operations described in the standard.
- **Basic format:** This format covers five floating-point representations, three binary and two decimal, whose encodings are specified by the standard, and which can be used for arithmetic. At least one of the basic formats is implemented in any conforming implementation.
- **Interchange format:** A fully specified, fixed-length binary encoding that allows data interchange between different platforms and that can be used for storage.

The three basic binary formats have bit lengths of 32, 64, and 128 bits, with exponents of 8, 11, and 15 bits, respectively (Figure 10.21). Table 10.3 summarizes the characteristics of the three formats. The two basic decimal formats have bit lengths of 64 and 128 bits. All of the basic formats are also arithmetic format types (can be used for arithmetic operations) and interchange format types (platform independent).

Several other formats are specified in the standard. The binary16 format is only an interchange format and is intended for storage of values when higher precision is not required. The binary{k} format and the decimal{k} format are interchange formats with total length k bits and with defined lengths for the significand and exponent. The format must be a multiple of 32 bits; thus formats are defined for $k = 160, 192$, and so on. These two families of formats are also arithmetic formats.

In addition, the standard defines **extended precision formats**, which extend a supported basic format by providing additional bits in the exponent (extended range) and in the significand (extended precision). The exact format

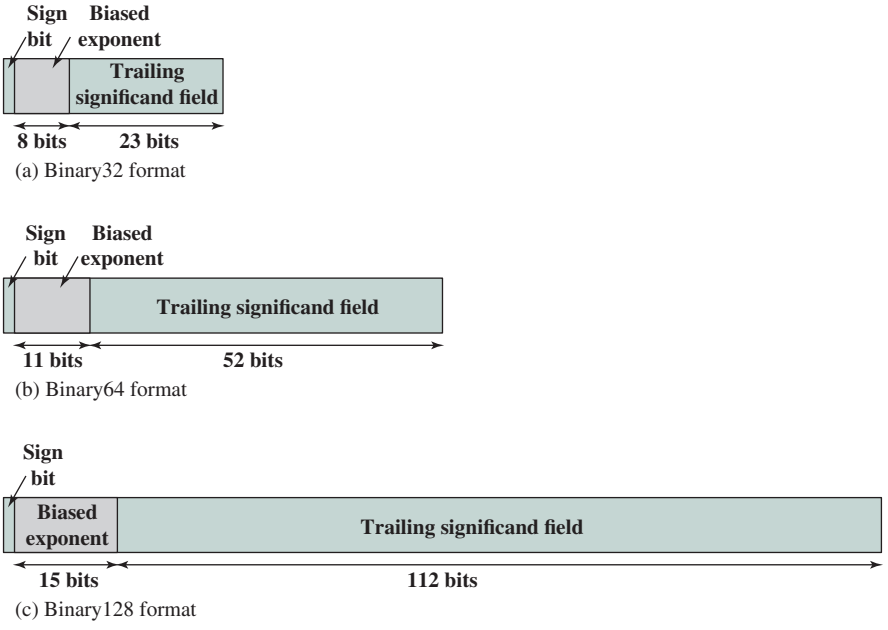


Figure 10.21 IEEE 754 Formats

is implementation dependent, but the standard places certain constraints on the length of the exponent and significand. These formats are arithmetic format types but not interchange format types. The extended formats are to be used for intermediate calculations. With their greater precision, the extended formats lessen the

Table 10.3 IEEE 754 Format Parameters

Parameter	Format		
	Binary32	Binary64	Binary128
Storage width (bits)	32	64	128
Exponent width (bits)	8	11	15
Exponent bias	127	1023	16383
Maximum exponent	127	1023	16383
Minimum exponent	−126	−1022	−16382
Approx normal number range (base 10)	$10^{-38}, 10^{+38}$	$10^{-308}, 10^{+308}$	$10^{-4932}, 10^{+4932}$
Trailing significand width (bits)*	23	52	112
Number of exponents	254	2046	32766
Number of fractions	2^{23}	2^{52}	2^{112}
Number of values	1.98×2^{31}	1.99×2^{63}	1.99×2^{128}
Smallest positive normal number	2^{-126}	2^{-1022}	2^{-16362}
Largest positive normal number	$2^{128} - 2^{104}$	$2^{1024} - 2^{971}$	$2^{16384} - 2^{16271}$
Smallest subnormal magnitude	2^{-149}	2^{-1074}	2^{-16494}

Note: * Not including implied bit and not including sign bit.

chance of a final result that has been contaminated by excessive roundoff error; with their greater range, they also lessen the chance of an intermediate overflow aborting a computation whose final result would have been representable in a basic format. An additional motivation for the extended format is that it affords some of the benefits of a larger basic format without incurring the time penalty usually associated with higher precision.

Finally, IEEE 754-2008 defines an **extendable precision format** as a format with a precision and range that are defined under user control. Again, these formats may be used for intermediate calculations, but the standard places no constraint or format or length.

Table 10.4 shows the relationship between defined formats and format types.

Not all bit patterns in the IEEE formats are interpreted in the usual way; instead, some bit patterns are used to represent special values. Table 10.5 indicates the values assigned to various bit patterns. The exponent values of all zeros (0 bits) and all ones (1 bits) define special values. The following classes of numbers are represented:

- For exponent values in the range of 1 through 254 for 32-bit format, 1 through 2046 for 64-bit format, and 1 through 16382, normal nonzero floating-point numbers are represented. The exponent is biased, so that the range of exponents is −126 through +127 for 32-bit format, and so on. A normal number requires a 1 bit to the left of the binary point; this bit is implied, giving an effective 24-bit, 53-bit, or 113-bit significand. Because one of the bits is implied, the corresponding field in the binary format is referred to as the **trailing significand field**.
- An exponent of zero together with a fraction of zero represents positive or negative zero, depending on the sign bit. As was mentioned, it is useful to have an exact value of 0 represented.

Table 10.4 IEEE Formats

Format	Format Type		
	Arithmetic Format	Basic Format	Interchange Format
binary16			X
binary32	X	X	X
binary64	X	X	X
binary128	X	X	X
binary{*k*} **(*k* = *n* × 32 for *n* > 4)**	X		X
decimal64	X	X	X
decimal128	X	X	X
decimal{*k*} **(*k* = *n* × 32 for *n* > 4)**	X		X
extended precision	X		
extendable precision	X		

Table 10.5 Interpretation of IEEE 754 Floating-Point Numbers

(a) binary32 format

	Sign	Biased Exponent	Fraction	Value
positive zero	0	0	0	0
negative zero	1	0	0	−0
plus infinity	0	all 1s	0	∞
minus infinity	1	all 1s	0	−∞
quiet NaN	0 or 1	all 1s	≠ 0; first bit = 1	qNaN
signaling NaN	0 or 1	all 1s	≠ 0; first bit = 0	sNaN
positive normal nonzero	0	0 < e < 225	f	$2^{e-127}(1.f)$
negative normal nonzero	1	0 < e < 225	f	$-2^{e-127}(1.f)$
positive subnormal	0	0	f ≠ 0	$2^{e-126}(0.f)$
negative subnormal	1	0	f ≠ 0	$-2^{e-126}(0.f)$

(b) binary64 format

	Sign	Biased Exponent	Fraction	Value
positive zero	0	0	0	0
negative zero	1	0	0	−0
plus infinity	0	all 1s	0	∞
minus infinity	1	all 1s	0	−∞
quiet NaN	0 or 1	all 1s	≠ 0; first bit = 1	qNaN
signaling NaN	0 or 1	all 1s	≠ 0; first bit = 0	sNaN
positive normal nonzero	0	0 < e < 2047	f	$2^{e-1023}(1.f)$
negative normal nonzero	1	0 < e < 2047	f	$-2^{e-1023}(1.f)$
positive subnormal	0	0	f ≠ 0	$2^{e-1022}(0.f)$
negative subnormal	1	0	f ≠ 0	$-2^{e-1022}(0.f)$

(c) binary128 format

	Sign	Biased Exponent	Fraction	Value
positive zero	0	0	0	0
negative zero	1	0	0	−0
plus infinity	0	all 1s	0	∞
minus infinity	1	all 1s	0	−∞
quiet NaN	0 or 1	all 1s	≠ 0; first bit = 1	qNaN
signaling NaN	0 or 1	all 1s	≠ 0; first bit = 0	sNaN
positive normal nonzero	0	all 1s	f	$2^{e-16383}(1.f)$
negative normal nonzero	1	all 1s	f	$-2^{e-16383}(1.f)$
positive subnormal	0	0	f ≠ 0	$2^{e-16383}(0.f)$
negative subnormal	1	0	f ≠ 0	$-2^{e-16383}(0.f)$

- An exponent of all ones together with a fraction of zero represents positive or negative infinity, depending on the sign bit. It is also useful to have a representation of infinity. This leaves it up to the user to decide whether to treat overflow as an error condition or to carry the value ∞ and proceed with whatever program is being executed.
- An exponent of zero together with a nonzero fraction represents a subnormal number. In this case, the bit to the left of the binary point is zero and the true exponent is -126 or -1022. The number is positive or negative depending on the sign bit.
- An exponent of all ones together with a nonzero fraction is given the value NaN, which means *Not a Number*, and is used to signal various exception conditions.

The significance of subnormal numbers and NaNs is discussed in Section 10.5.

10.5 FLOATING-POINT ARITHMETIC

Table 10.6 summarizes the basic operations for floating-point arithmetic. For addition and subtraction, it is necessary to ensure that both operands have the same exponent value. This may require shifting the radix point on one of the operands to achieve alignment. Multiplication and division are more straightforward.

A floating-point operation may produce one of these conditions:

- **Exponent overflow:** A positive exponent exceeds the maximum possible exponent value. In some systems, this may be designated as $+\infty$ or $-\infty$.
- **Exponent underflow:** A negative exponent is less than the minimum possible exponent value (e.g., -200 is less than -127). This means that the number is too small to be represented, and it may be reported as 0.

Table 10.6 Floating-Point Numbers and Arithmetic Operations

Floating-Point Numbers	Arithmetic Operations
$X = X_S \times B^{X_E}$ $Y = Y_S \times B^{Y_E}$	$\left.\begin{array}{l} X + Y = (X_S \times B^{X_E - Y_E} + Y_S) \times B^{Y_E} \\ X - Y = (X_S \times B^{X_E - Y_E} - Y_S) \times B^{Y_E} \end{array}\right\} X_E \leq Y_E$ $X \times Y = (X_S \times Y_S) \times B^{X_E + Y_E}$ $\frac{X}{Y} = \left(\frac{X_S}{Y_S}\right) \times B^{X_E - Y_E}$

Examples:

$X = 0.3 \times 10^2 = 30$

$Y = 0.2 \times 10^3 = 200$

$X + Y = (0.3 \times 10^{2-3} + 0.2) \times 10^3 = 0.23 \times 10^3 = 230$

$X - Y = (0.3 \times 10^{2-3} - 0.2) \times 10^3 = (-0.17) \times 10^3 = -170$

$X \times Y = (0.3 \times 0.2) \times 10^{2+3} = 0.06 \times 10^5 = 6000$

$X \div Y = (0.3 \div 0.2) \times 10^{2-3} = 1.5 \times 10^{-1} = 0.15$

- **Significand underflow:** In the process of aligning significands, digits may flow off the right end of the significand. As we will discuss, some form of rounding is required.
- **Significand overflow:** The addition of two significands of the same sign may result in a carry out of the most significant bit. This can be fixed by realignment, as we will explain.

Addition and Subtraction

In floating-point arithmetic, addition and subtraction are more complex than multiplication and division. This is because of the need for alignment. There are four basic phases of the algorithm for addition and subtraction:

1. Check for zeros.
2. Align the significands.
3. Add or subtract the significands.
4. Normalize the result.

A typical flowchart is shown in Figure 10.22. A step-by-step narrative highlights the main functions required for floating-point addition and subtraction. We assume a format similar to those of Figure 10.21. For the addition or subtraction operation, the two operands must be transferred to registers that will be used by the ALU. If the floating-point format includes an implicit significand bit, that bit must be made explicit for the operation.

Phase 1. Zero check: Because addition and subtraction are identical except for a sign change, the process begins by changing the sign of the subtrahend if it is a subtract operation. Next, if either operand is 0, the other is reported as the result.

Phase 2. Significand alignment: The next phase is to manipulate the numbers so that the two exponents are equal.

To see the need for aligning exponents, consider the following decimal addition:

$$(123 \times 10^0) + (456 \times 10^{-2})$$

Clearly, we cannot just add the significands. The digits must first be set into equivalent positions, that is, the 4 of the second number must be aligned with the 3 of the first. Under these conditions, the two exponents will be equal, which is the mathematical condition under which two numbers in this form can be added. Thus,

$$(123 \times 10^0) + (456 \times 10^{-2}) = (123 \times 10^0) + (4.56 \times 10^0) = 127.56 \times 10^0$$

Alignment may be achieved by shifting either the smaller number to the right (increasing its exponent) or shifting the larger number to the left. Because either operation may result in the loss of digits, it is the smaller number that is shifted; any digits that are lost are therefore of relatively small significance. The alignment

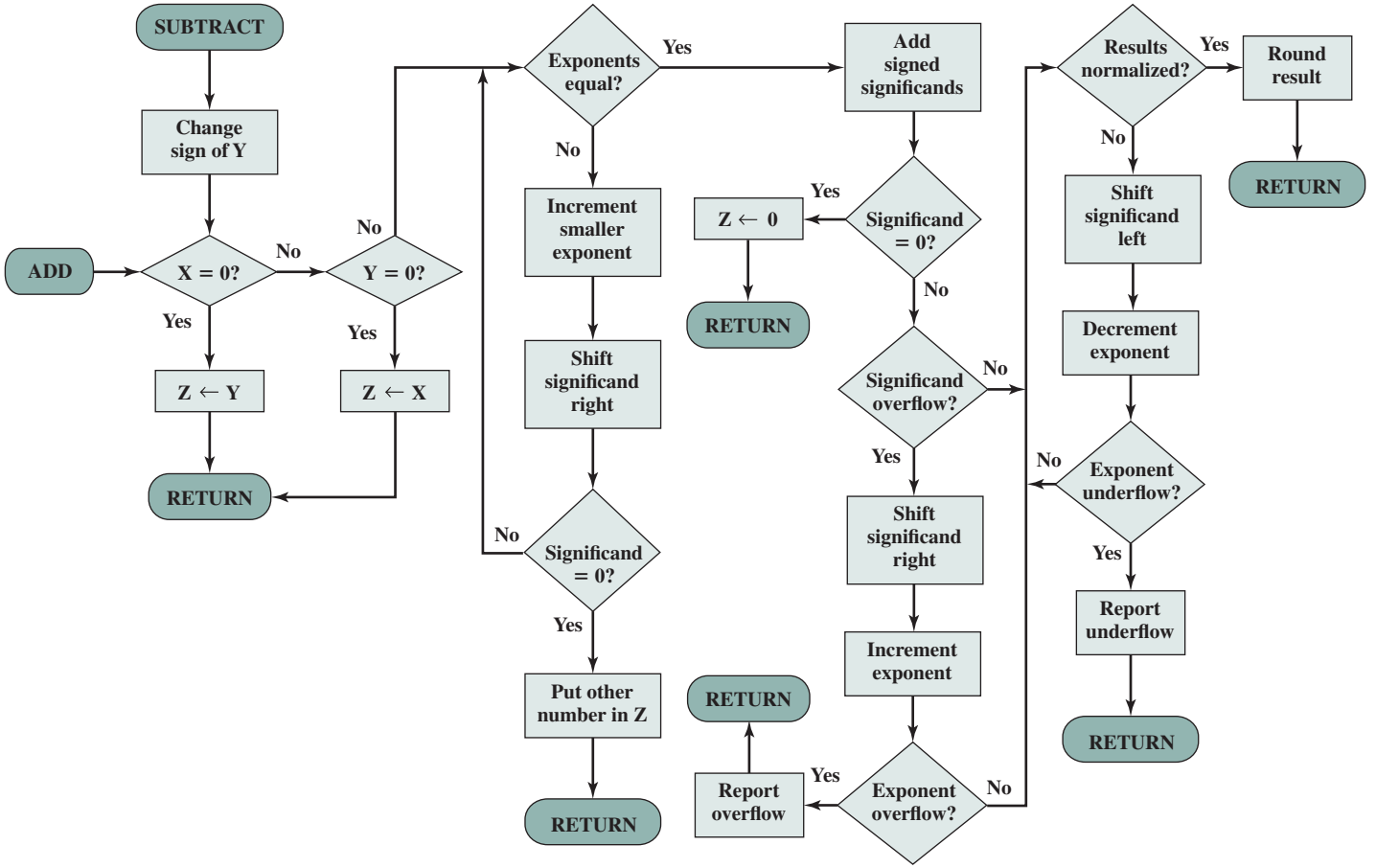


Figure 10.22 Floating-Point Addition and Subtraction ($Z \leftarrow X \pm Y$)

is achieved by repeatedly shifting the magnitude portion of the significand right 1 digit and incrementing the exponent until the two exponents are equal. (Note that if the implied base is 16, a shift of 1 digit is a shift of 4 bits.) If this process results in a 0 value for the significand, then the other number is reported as the result. Thus, if two numbers have exponents that differ significantly, the lesser number is lost.

Phase 3. Addition: Next, the two significands are added together, taking into account their signs. Because the signs may differ, the result may be 0. There is also the possibility of significand overflow by 1 digit. If so, the significand of the result is shifted right and the exponent is incremented. An exponent overflow could occur as a result; this would be reported and the operation halted.

Phase 4. Normalization: The final phase normalizes the result. Normalization consists of shifting significand digits left until the most significant digit (bit, or 4 bits for base-16 exponent) is nonzero. Each shift causes a decrement of the exponent and thus could cause an exponent underflow. Finally, the result must be rounded off and then reported. We defer a discussion of rounding until after a discussion of multiplication and division.

Multiplication and Division

Floating-point multiplication and division are much simpler processes than addition and subtraction, as the following discussion indicates.

We first consider multiplication, illustrated in Figure 10.23. First, if either operand is 0, 0 is reported as the result. The next step is to add the exponents. If the exponents are stored in biased form, the exponent sum would have doubled the bias. Thus, the bias value must be subtracted from the sum. The result could be either an exponent overflow or underflow, which would be reported, ending the algorithm.

If the exponent of the product is within the proper range, the next step is to multiply the significands, taking into account their signs. The multiplication is performed in the same way as for integers. In this case, we are dealing with a sign-magnitude representation, but the details are similar to those for twos complement representation. The product will be double the length of the multiplier and multiplicand. The extra bits will be lost during rounding.

After the product is calculated, the result is then normalized and rounded, as was done for addition and subtraction. Note that normalization could result in exponent underflow.

Finally, let us consider the flowchart for division depicted in Figure 10.24. Again, the first step is testing for 0. If the divisor is 0, an error report is issued, or the result is set to infinity, depending on the implementation. A dividend of 0 results in 0. Next, the divisor exponent is subtracted from the dividend exponent. This removes the bias, which must be added back in. Tests are then made for exponent underflow or overflow.

The next step is to divide the significands. This is followed with the usual normalization and rounding.

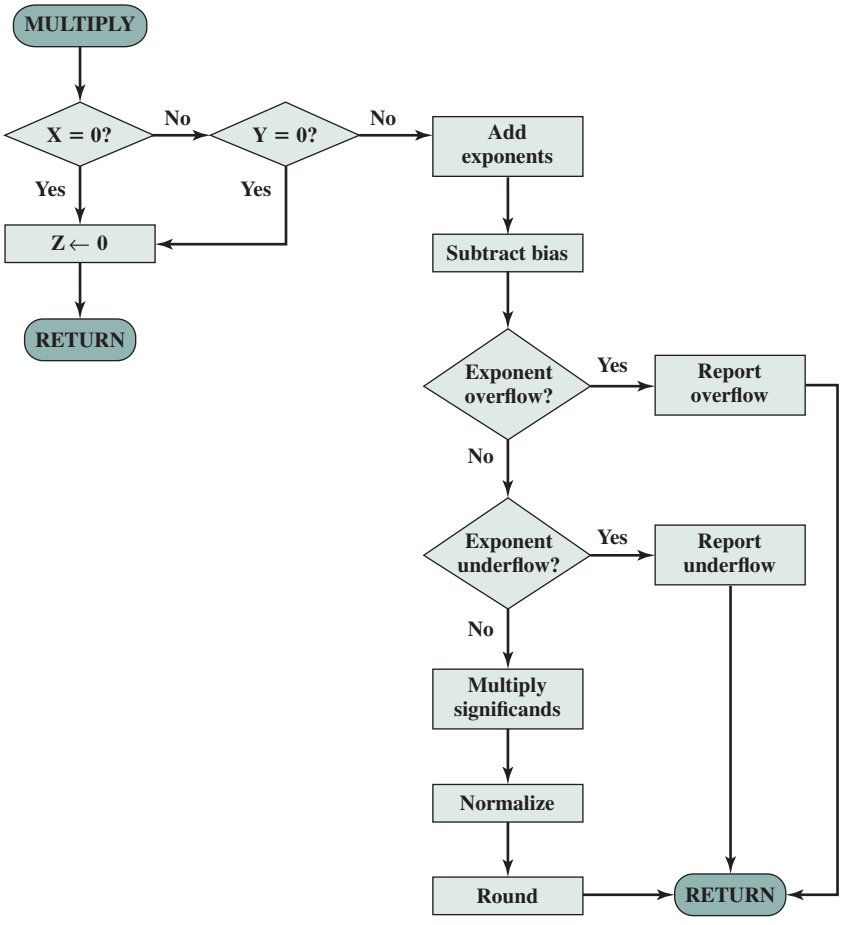


Figure 10.23 Floating-Point Multiplication ($Z \leftarrow X \pm Y$)

Precision Considerations

GUARD BITS We mentioned that, prior to a floating-point operation, the exponent and significand of each operand are loaded into ALU registers. In the case of the significand, the length of the register is almost always greater than the length of the significand plus an implied bit. The register contains additional bits, called guard bits, which are used to pad out the right end of the significand with 0s.

The reason for the use of guard bits is illustrated in Figure 10.25. Consider numbers in the IEEE format, which has a 24-bit significand, including an implied 1 bit to the left of the binary point. Two numbers that are very close in value are $x = 1.00 \cdots 00 \times 2^1$ and $y = 1.11 \cdots 11 \times 2^0$. If the smaller number is to be subtracted from the larger, it must be shifted right 1 bit to align the exponents. This is shown in Figure 10.25a. In the process, y loses 1 bit of significance; the result is 2^{-22}. The same operation is repeated in

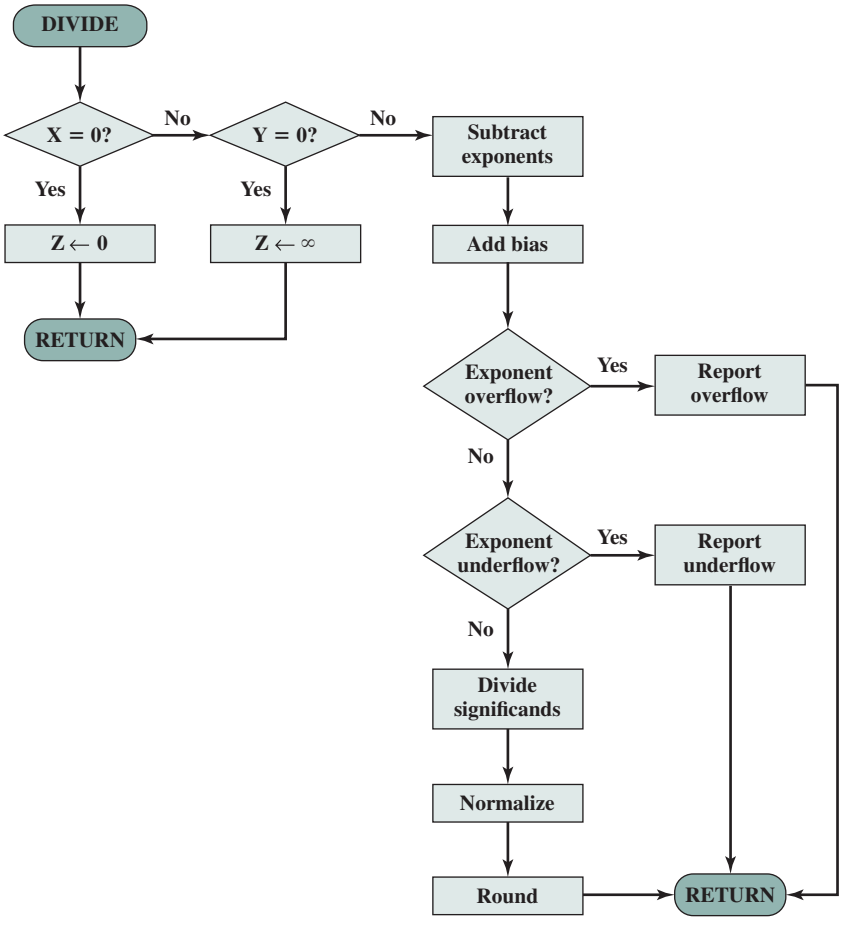


Figure 10.24 Floating-Point Division (Z ← X/Y)

$$
\begin{aligned}
x &= 1.000.....00 \times 2^{1} \\
-y &= 0.111.....11 \times 2^{1} \\
z &= 0.000.....01 \times 2^{1} \\
&= 1.000.....00 \times 2^{-22}
\end{aligned}
$$

(a) Binary example, without guard bits

$$
\begin{aligned}
x &= .100000 \times 16^{1} \\
-y &= .0FFFFF \times 16^{1} \\
z &= .000001 \times 16^{1} \\
&= .100000 \times 16^{-4}
\end{aligned}
$$

(c) Hexadecimal example, without guard bits

$$
\begin{aligned}
x &= 1.000.....00\ 0000 \times 2^{1} \\
-y &= 0.111.....11\ 1000 \times 2^{1} \\
z &= 0.000.....00\ 1000 \times 2^{1} \\
&= 1.000.....00\ 0000 \times 2^{-23}
\end{aligned}
$$

(b) Binary example, with guard bits

$$
\begin{aligned}
x &= .100000\ 00 \times 16^{1} \\
-y &= .0FFFFF\ F0 \times 16^{1} \\
z &= .000000\ 10 \times 16^{1} \\
&= .100000\ 00 \times 16^{-5}
\end{aligned}
$$

(d) Hexadecimal example, with guard bits

Figure 10.25 The Use of Guard Bits

part (b) with the addition of guard bits. Now the least significant bit is not lost due to alignment, and the result is 2^{-23}, a difference of a factor of 2 from the previous answer. When the radix is 16, the loss of precision can be greater. As Figures 10.25c and (d) show, the difference can be a factor of 16.

ROUNDING Another detail that affects the precision of the result is the rounding policy. The result of any operation on the significands is generally stored in a longer register. When the result is put back into the floating-point format, the extra bits must be eliminated in such a way as to produce a result that is close to the exact result. This process is called **rounding**.

A number of techniques have been explored for performing rounding. In fact, the IEEE standard lists four alternative approaches:

- **Round to nearest:** The result is rounded to the nearest representable number.
- **Round toward $+\infty$:** The result is rounded up toward plus infinity.
- **Round toward $-\infty$:** The result is rounded down toward negative infinity.
- **Round toward 0:** The result is rounded toward zero.

Let us consider each of these policies in turn. **Round to nearest** is the default rounding mode listed in the standard and is defined as follows: The representable value nearest to the infinitely precise result shall be delivered.

If the extra bits, beyond the 23 bits that can be stored, are 10010, then the extra bits amount to more than one-half of the last representable bit position. In this case, the correct answer is to add binary 1 to the last representable bit, rounding up to the next representable number. Now consider that the extra bits are 01111. In this case, the extra bits amount to less than one-half of the last representable bit position. The correct answer is simply to drop the extra bits (truncate), which has the effect of rounding down to the next representable number.

The standard also addresses the special case of extra bits of the form 10000.... Here the result is exactly halfway between the two possible representable values. One possible technique here would be to always truncate, as this would be the simplest operation. However, the difficulty with this simple approach is that it introduces a small but cumulative bias into a sequence of computations. What is required is an unbiased method of rounding. One possible approach would be to round up or down on the basis of a random number so that, on average, the result would be unbiased. The argument against this approach is that it does not produce predictable, deterministic results. The approach taken by the IEEE standard is to force the result to be even: If the result of a computation is exactly midway between two representable numbers, the value is rounded up if the last representable bit is currently 1 and not rounded up if it is currently 0.

The next two options, **rounding to plus** and **minus infinity**, are useful in implementing a technique known as interval arithmetic. Interval arithmetic provides an efficient method for monitoring and controlling errors in floating-point computations by producing two values for each result. The two values correspond to the lower and upper endpoints of an interval that contains the true result. The width of the interval, which is the difference between the upper and lower endpoints, indicates the accuracy of the result. If the endpoints of an interval are not representable, then the interval endpoints are rounded down and up, respectively. Although the width of the interval may vary according to implementation, many algorithms have been designed to produce narrow intervals. If the range between the upper and lower bounds is sufficiently narrow, then a sufficiently accurate result has been obtained. If not, at least we know this and can perform additional analysis.

The final technique specified in the standard is **round toward zero**. This is, in fact, simple truncation: The extra bits are ignored. This is certainly the simplest technique. However, the result is that the magnitude of the truncated value is always less than or equal to the more precise original value, introducing a consistent bias toward zero in the operation. This is a serious bias because it affects every operation for which there are nonzero extra bits.

IEEE Standard for Binary Floating-Point Arithmetic

IEEE 754 goes beyond the simple definition of a format to lay down specific practices and procedures so that floating-point arithmetic produces uniform, predictable results independent of the hardware platform. One aspect of this has already been discussed, namely rounding. This subsection looks at three other topics: infinity, NaNs, and subnormal numbers.

INFINITY Infinity arithmetic is treated as the limiting case of real arithmetic, with the infinity values given the following interpretation:

$$-\infty < (\text{every finite number}) < +\infty$$

With the exception of the special cases discussed subsequently, any arithmetic operation involving infinity yields the obvious result.

For example:

$$\begin{array}{ll} 5 + (+\infty) = +\infty & 5 \div (+\infty) = +0 \\ 5 - (+\infty) = -\infty & (+\infty) + (+\infty) = +\infty \\ 5 + (-\infty) = -\infty & (-\infty) + (-\infty) = -\infty \\ 5 - (-\infty) = +\infty & (-\infty) - (+\infty) = -\infty \\ 5 \times (+\infty) = +\infty & (+\infty) - (-\infty) = +\infty \end{array}$$

QUIET AND SIGNALING NANS A NaN is a symbolic entity encoded in floating-point format, of which there are two types: signaling and quiet. A signaling NaN signals an invalid operation exception whenever it appears as an operand. Signaling

Table 10.7 Operations that Produce a Quiet NaN

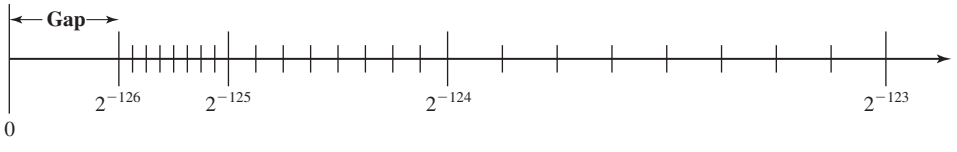
Operation	Quiet NaN Produced By
Any	Any operation on a signaling NaN
Add or subtract	Magnitude subtraction of infinities: $(+\infty) + (-\infty)$ $(-\infty) + (+\infty)$ $(+\infty) - (+\infty)$ $(-\infty) - (-\infty)$
Multiply	$0 \times \infty$
Division	$\frac{0}{0}$ or $\frac{\infty}{\infty}$
Remainder	x REM 0 or ∞ REM y
Square root	$\sqrt{x}$, where $x < 0$

NaNs afford values for uninitialized variables and arithmetic-like enhancements that are not the subject of the standard. A quiet NaN propagates through almost every arithmetic operation without signaling an exception. Table 10.7 indicates operations that will produce a quiet NaN.

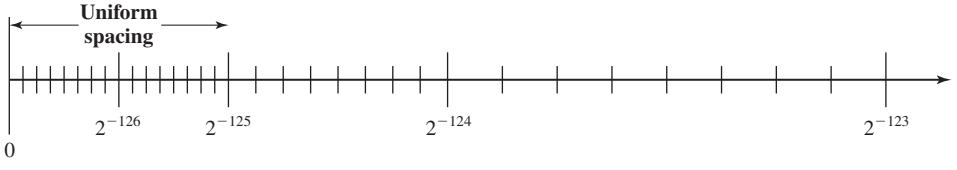
Note that both types of NaNs have the same general format (Table 10.4): an exponent of all ones and a nonzero fraction. The actual bit pattern of the nonzero fraction is implementation dependent; the fraction values can be used to distinguish quiet NaNs from signaling NaNs and to specify particular exception conditions.

SUBNORMAL NUMBERS Subnormal numbers are included in IEEE 754 to handle cases of exponent underflow. When the exponent of the result becomes too small (a negative exponent with too large a magnitude), the result is subnormalized by right shifting the fraction and incrementing the exponent for each shift until the exponent is within a representable range.

Figure 10.26 illustrates the effect of including subnormal numbers. The representable numbers can be grouped into intervals of the form $[2^n, 2^{n+1}]$. Within



(a) 32-bit format without subnormal numbers



(b) 32-bit format with subnormal numbers

Figure 10.26 The Effect of IEEE 754 Subnormal Numbers

each such interval, the exponent portion of the number remains constant while the fraction varies, producing a uniform spacing of representable numbers within the interval. As we get closer to zero, each successive interval is half the width of the preceding interval but contains the same number of representable numbers. Hence the density of representable numbers increases as we approach zero. However, if only normal numbers are used, there is a gap between the smallest normal number and 0. In the case of the 32-bit IEEE 754 format, there are 2^{23} representable numbers in each interval, and the smallest representable positive number is 2^{-126}. With the addition of subnormal numbers, an additional $2^{23} - 1$ numbers are uniformly added between 0 and 2^{-126}.

The use of subnormal numbers is referred to as *gradual underflow* [COON81]. Without subnormal numbers, the gap between the smallest representable nonzero number and zero is much wider than the gap between the smallest representable nonzero number and the next larger number. Gradual underflow fills in that gap and reduces the impact of exponent underflow to a level comparable with roundoff among the normal numbers.

10.6 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

arithmetic and logic unit (ALU)	minuend	radix point
arithmetic shift	multiplicand	range extension
base	multiplier	remainder
biased representation	negative overflow	rounding
dividend	negative underflow	sign bit
divisor	normal number	sign-magnitude representation
exponent	ones complement representation	significand
exponent overflow	overflow	significand overflow
exponent underflow	partial product	significand underflow
fixed-point representation	positive overflow	subnormal number
floating-point representation	positive underflow	subtrahend
guard bits	product	twos complement representation
mantissa	quotient	

Review Questions

10.1 Briefly explain the following representations: sign magnitude, twos complement, biased.

10.2 Explain how to determine if a number is negative in the following representations: sign magnitude, twos complement, biased.

10.3 What is the sign-extension rule for twos complement numbers?

10.4 How can you form the negation of an integer in twos complement representation?

10.5 In general terms, when does the twos complement operation on an n-bit integer produce the same integer?

10.6 What is the difference between the twos complement representation of a number and the twos complement of a number?

10.7 If we treat two twos complement numbers as unsigned integers for purposes of addition, the result is correct if interpreted as a twos complement number. This is not true for multiplication. Why?

10.8 What are the four essential elements of a number in floating-point notation?

10.9 What is the benefit of using biased representation for the exponent portion of a floating-point number?

10.10 What are the differences among positive overflow, exponent overflow, and significand overflow?

10.11 What are the basic elements of floating-point addition and subtraction?

10.12 Give a reason for the use of guard bits.

10.13 List four alternative methods of rounding the result of a floating-point operation.

Problems

10.1 Represent the following decimal numbers in both binary sign/magnitude and twos complement using 16 bits: + 512; − 29.

10.2 Represent the following twos complement values in decimal: 1101011; 0101101.

10.3 Another representation of binary integers that is sometimes encountered is **ones complement**. Positive integers are represented in the same way as sign magnitude. A negative integer is represented by taking the Boolean complement of each bit of the corresponding positive number.

a. Provide a definition of ones complement numbers using a weighted sum of bits, similar to Equations (10.1) and (10.2).

b. What is the range of numbers that can be represented in ones complement?

c. Define an algorithm for performing addition in ones complement arithmetic.

Note: Ones complement arithmetic disappeared from hardware in the 1960s, but still survives checksum calculations for the Internet Protocol (IP) and the Transmission Control Protocol (TCP).

10.4 Add columns to Table 10.1 for sign magnitude and ones complement.

10.5 Consider the following operation on a binary word. Start with the least significant bit. Copy all bits that are 0 until the first bit is reached and copy that bit, too. Then take the complement of each bit thereafter. What is the result?

10.6 In Section 10.3, the twos complement operation is defined as follows. To find the twos complement of X, take the Boolean complement of each bit of X, and then add 1.

a. Show that the following is an equivalent definition. For an n-bit integer X, the twos complement of X is formed by treating X as an unsigned integer and calculating $(2_n - X)$.

b. Demonstrate that Figure 10.5 can be used to support graphically the claim in part (a), by showing how a clockwise movement is used to achieve subtraction.

10.7 The r's complement of an n-digit number N in base r is defined as $r^n - N$ for $N \neq 0$ and 0 for $N = 0$. Find the tens complement of the decimal number 13,250.

10.8 Calculate (72,530 − 13,250) using tens complement arithmetic. Assume rules similar to those for twos complement arithmetic.

10.9 Consider the twos complement addition of two n-bit numbers:

$$z_{n-1}z_{n-2} \cdots z_0 = x_{n-1}x_{n-2} \cdots x_0 + y_{n-1}y_{n-2} \cdots y_0$$

Assume that bitwise addition is performed with a carry bit c_i generated by the addition of x_i, y_i, and c_{i-1}. Let ν be a binary variable indicating overflow when $\nu = 1$. Fill in the values in the table.

Input	x_{n-1}	0	0	0	0	1	1	1	1
	y_{n-1}	0	0	1	1	0	0	1	1
	c_{n-2}	0	1	0	1	0	1	0	1
Output	z_{n-1}								
	v								

10.10 Assume numbers are represented in 8-bit twos complement representation. Show the calculation of the following:

a. $6 + 13$ **b.** $-6 + 13$ **c.** $6 - 13$ **d.** $-6 - 13$

10.11 Find the following differences using twos complement arithmetic:

a. $\begin{array}{r} 111000 \\ -110011 \end{array}$ **b.** $\begin{array}{r} 11001100 \\ -101110 \end{array}$ **c.** $\begin{array}{r} 111100001111 \\ -110011110011 \end{array}$ **d.** $\begin{array}{r} 11000011 \\ -11101000 \end{array}$

10.12 Is the following a valid alternative definition of overflow in twos complement arithmetic?

If the exclusive-OR of the carry bits into and out of the leftmost column is 1, then there is an overflow condition. Otherwise, there is not.

10.13 Compare Figures 10.9 and 10.12. Why is the C bit not used in the latter?

10.14 Given $x = 0101$ and $y = 1010$ in twos complement notation (i.e., $x = 5, y = -6$), compute the product $p = x \times y$ with Booth's algorithm.

10.15 Use the Booth algorithm to multiply 23 (multiplicand) by 29 (multiplier), where each number is represented using 6 bits.

10.16 Prove that the multiplication of two n-digit numbers in base B gives a product of no more than $2n$ digits.

10.17 Verify the validity of the unsigned binary division algorithm of Figure 10.16 by showing the steps involved in calculating the division depicted in Figure 10.15. Use a presentation similar to that of Figure 10.17.

10.18 The twos complement integer division algorithm described in Section 10.3 is known as the restoring method because the value in the A register must be restored following unsuccessful subtraction. A slightly more complex approach, known as nonrestoring, avoids the unnecessary subtraction and addition. Propose an algorithm for this latter approach.

10.19 Under computer integer arithmetic, the quotient J/K of two integers J and K is less than or equal to the usual quotient. True or false?

10.20 Divide -145 by 13 in binary twos complement notation, using 12-bit words. Use the algorithm described in Section 10.3.

10.21 **a.** Consider a fixed-point representation using decimal digits, in which the implied radix point can be in any position (to the right of the least significant digit, to the right of the most significant digit, and so on). How many decimal digits are needed to represent the approximations of both Planck's constant (6.63×10^{-27}) and Avogadro's number (6.02×10^{23})? The implied radix point must be in the same position for both numbers.

b. Now consider a decimal floating-point format with the exponent stored in a biased representation with a bias of 50. A normalized representation is assumed. How many decimal digits are needed to represent these constants in this floating-point format?

10.22 Assume that the exponent e is constrained to lie in the range $0 \leq e \leq X$, with a bias of q, that the base is b, and that the significand is p digits in length.

a. What are the largest and smallest positive values that can be written?

b. What are the largest and smallest positive values that can be written as normalized floating-point numbers?

10.23 Express the following numbers in IEEE 32-bit floating-point format:

a. −5 **b.** −6 **c.** −1.5 **d.** 384 **e.** 1/16 **f.** −1/32

10.24 The following numbers use the IEEE 32-bit floating-point format. What is the equivalent decimal value?

a. 1 10000011 11000000000000000000000
b. 0 01111110 10100000000000000000000
c. 0 10000000 00000000000000000000000

10.25 Consider a reduced 7-bit IEEE floating-point format, with 3 bits for the exponent and 3 bits for the significand. List all 127 values.

10.26 Express the following numbers in IBM's 32-bit floating-point format, which uses a 7-bit exponent with an implied base of 16 and an exponent bias of 64 (40 hexadecimal). A normalized floating-point number requires that the leftmost hexadecimal digit be nonzero; the implied radix point is to the left of that digit.

a. 1.0	**c.** 1/64	**e.** − 15.0	**g.** 7.2×10^{75}
b. 0.5	**d.** 0.0	**f.** 5.4×10^{-79}	**h.** 65,535

10.27 Let 5BCA0000 be a floating-point number in IBM format, expressed in hexadecimal. What is the decimal value of the number?

10.28 What would be the bias value for

a. A base-2 exponent (B = 2) in a 6-bit field?
b. A base-8 exponent (B = 8) in a 7-bit field?

10.29 Draw a number line similar to that in Figure 10.19b for the floating-point format of Figure 10.21b.

10.30 Consider a floating-point format with 8 bits for the biased exponent and 23 bits for the significand. Show the bit pattern for the following numbers in this format:

a. −720 **b.** 0.645

10.31 The text mentions that a 32-bit format can represent a maximum of 2^{32} different numbers. How many different numbers can be represented in the IEEE 32-bit format? Explain.

10.32 Any floating-point representation used in a computer can represent only certain real numbers exactly; all others must be approximated. If A' is the stored value approximating the real value A, then the relative error, r, is expressed as

$$r = \frac{A - A'}{A}$$

Represent the decimal quantity + 0.4 in the following floating-point format: base = 2; exponent: biased, 4 bits; significand, 7 bits. What is the relative error?

10.33 If $A = 1.427$, find the relative error if A is truncated to 1.42 and if it is rounded to 1.43.

10.34 When people speak about inaccuracy in floating-point arithmetic, they often ascribe errors to cancellation that occurs during the subtraction of nearly equal quantities. But when X and Y are approximately equal, the difference $X - Y$ is obtained exactly, with no error. What do these people really mean?

10.35 Numerical values A and B are stored in the computer as approximations A' and B' Neglecting any further truncation or roundoff errors, show that the relative error of the product is approximately the sum of the relative errors in the factors.

10.36 One of the most serious errors in computer calculations occurs when two nearly equal numbers are subtracted. Consider $A = 0.22288$ and $B = 0.22211$. The computer truncates all values to four decimal digits. Thus $A' = 0.2228$ and $B' = 0.2221$.

a. What are the relative errors for A' and B'?
b. What is the relative error for $C' = A' - B'$?

10.37 To get some feel for the effects of denormalization and gradual underflow, consider a decimal system that provides 6 decimal digits for the significand and for which the smallest normalized number is 10^{-99}. A normalized number has one nonzero decimal digit to the left of the decimal point. Perform the following calculations and denormalize the results. Comment on the results.

a. $(2.50000 \times 10^{-60}) \times (3.50000 \times 10^{-43})$
b. $(2.50000 \times 10^{-60}) \times (3.50000 \times 10^{-60})$
c. $(5.67834 \times 10^{-97}) - (5.67812 \times 10^{-97})$

10.38 Show how the following floating-point additions are performed (where significands are truncated to 4 decimal digits). Show the results in normalized form.

a. $5.566 \times 10^{2} + 7.777 \times 10^{2}$
b. $3.344 \times 10^{1} + 8.877 \times 10^{-2}$

10.39 Show how the following floating-point subtractions are performed (where significands are truncated to 4 decimal digits). Show the results in normalized form.

a. $7.744 \times 10^{-3} - 6.666 \times 10^{-3}$ **b.** $8.844 \times 10^{-3} - 2.233 \times 10^{-1}$

10.40 Show how the following floating-point calculations are performed (where significands are truncated to 4 decimal digits). Show the results in normalized form.

a. $(2.255 \times 10^{1}) \times (1.234 \times 10^{0})$ **b.** $(8.833 \times 10^{2}) \div (5.555 \times 10^{4})$