bers, the computer must have some means of distinguishing a positive from a negative number. And, as previously explained, the computer word usually contains a sign bit, generally adjacent to the most significant bit in the computer word. In the systems to be described, a 1 in the sign bit will indicate a negative number and a 0 in the sign bit a positive number.

We have examined the representation of numbers in Sec. 5.1 by using a signed-integer magnitude representation system. Two other representation systems, however, are used more often—the 1s and 2s complement systems. (The 2s complement system is the most frequently used at present.) The advantage of these systems is that both positive and negative numbers can be added or subtracted by using only an adder of the type already explained.

Here are the three basic systems.

1 Negative numbers may be stored in their *true magnitude form*. Thus the binary number −0011 will be stored as 1̲0011, where the 1̲ indicates that the number stored is negative and the 0011 indicates the magnitude of the number.[5]

2 The *1s complement* of the magnitude may be used to represent a negative number. The binary number −0111 will, therefore, be represented as 1̲1000, where the 1̲ indicates that the number is negative and 1000 is the 1s complement of the magnitude. (The 1s complement is formed by simply complementing each bit of the positive magnitude.)

3 The *2s complement* may be used to represent a negative binary number. For instance, −0111 would be stored as 1̲1001, where the 1 in the sign bit indicates that the number is negative and the 1001 is the 2s complement of the magnitude of the number. (The 2s complement is formed by 1s-complementing the magnitude part 0111, giving 1000, and then adding 1 to the least significant digit, giving 1001.)

ADDITION IN THE 1S COMPLEMENT SYSTEM

5.7 The 1s complement system for representing negative numbers is often used in parallel binary machines. The main reason is the ease with which the 1s complement of a binary number may be formed, since only complementing each bit of a binary number stored in a flip-flop register is required. Before we discuss the implementations of an adder for the 1s complement system, we note the four possible basic situations which may arise in adding combinations of positive and negative numbers in the 1s complement system:

1 When a positive number is added to another positive number, the addition of all bits, including the sign bit, is straightforward. Since both sign bits will be

[5] Again we note that an underscore __ is used to separate the sign bit from the magnitude bits. Thus 0̲011 is +3. The sign bit is simply stored in a flip-flop in the computer as are the other bits; so this is simply a notational convention and is used to indicate signed numbers.

0s, no sum or carry will be generated in the sign-bit adder and the output will remain 0. Here is an example of the addition of two 4-bit positive numbers.[6]

$$\begin{array}{cc}
\text{NORMAL NOTATION} & \text{COMPUTER WORD} \\
+0011 & \underline{0}0011 \\
+0100 & \underline{0}0100 \\
\hline
+0111 & \underline{0}0111
\end{array}$$

2 When a positive and a negative number are added, the sum may be either positive or negative. If the positive number has a greater magnitude, the sum will be positive; and if the negative number is greater in magnitude, the sum will be negative. In the 1s complement system, the answer will be correct as is if the sum of the two numbers is negative in value. In this case no overflow will be generated when the numbers are added. For instance,

$$\begin{array}{rr}
+0011 & \underline{0}0011 \\
-1100 & \underline{1}0011 \\
\hline
-1001 & \underline{1}0110
\end{array}$$

In this case, the output of the adder will be 10110, the last 4 bits of which are the 1s complement of 1001, the correct magnitude of the sum. The 1 in the sign bit is also correct, indicating a negative number.

3 If the positive number is larger than the negative number, the sum before the end-around carry is added will be incorrect. The addition of the end-around carry will correct this sum. There will be a 0 in the sign bit, indicating that the sum is positive.

$$\begin{array}{rcr}
+1001 & = & \underline{0}1001 \\
-0100 & = & \underline{1}1011 \\
\hline
+0101 & & \underline{0}0100 \\
 & & \hookrightarrow 1 \\
\hline
 & & \underline{0}0101
\end{array}
\qquad
\begin{array}{rcr}
+0011 & = & \underline{0}0011 \\
-0010 & = & \underline{1}1101 \\
\hline
+0001 & & \underline{0}0000 \\
 & & \hookrightarrow 1 \\
\hline
 & & \underline{0}0001
\end{array}$$

Notice what happens when two numbers of equal magnitude but opposite signs are added:

$$\begin{array}{rcr}
+1011 & = & \underline{0}1011 \\
-1011 & = & \underline{1}0100 \\
\hline
0000 & & \underline{1}1111
\end{array}
\qquad
\begin{array}{rcr}
+0000 & = & \underline{0}0000 \\
-0000 & = & \underline{1}1111 \\
\hline
0000 & & \underline{1}1111
\end{array}$$

The result in these cases will be a negative zero ($\underline{1}1111$), which is correct.

4 When two negative numbers are added, an end-around carry will always be generated, as will a carry from the adder for the first bits of the magnitudes of the numbers. This will place a 1 in the sign bit.

[6] In this, and in all discussions that follow, we assume that the result (sum) does not exceed the capacity of the number of digits being used. This is discussed later.

$$\begin{array}{rcr} -0011 & = & \underline{1}1100 \\ \underline{-1011} & = & \underline{\underline{1}0100} \\ -1110 & \ulcorner & \underline{1}0000 \\ & \llcorner\!\!\longrightarrow & \underline{1} \\ & & \underline{1}0001 \end{array} \qquad \begin{array}{rcr} -0100 & = & \underline{1}1011 \\ \underline{-0111} & = & \underline{\underline{1}1000} \\ -1011 & \ulcorner & \underline{1}0011 \\ & \llcorner\!\!\longrightarrow & \underline{1} \\ & & \underline{1}0100 \end{array}$$

The output of the adder will be in 1s complement form in each case, with a 1 in the sign-bit position.

From the above we see that in order to implement an adder which will handle 4-bit magnitude signed 1s complement numbers, we can simply add another full-adder to the configuration in Fig. 5.5. The sign inputs will be labeled X_4 and Y_4, and the C_o output from the adder connected to X_3 and Y_3 will be connected to the C_i input of the new full-adder for X_4 and Y_4. The C_o output from the adder for X_4 and Y_4 will be connected to the C_i input for the adder for X_0 and Y_0. The S_4 output from the new adder will give the sign digit for the sum. (Overflow will not be detected in this adder; additional gates are required.)

ADDITION IN THE 2S COMPLEMENT SYSTEM

5.8 When negative numbers are represented in the 2s complement system, the operation of addition is very similar to that in the 1s complement system. In parallel machines, the 2s complement of a number stored in a register may be formed by first complementing the register and then adding 1 to the least significant bit of the register. This process requires two steps and so is more time-consuming[7] than the 1s complement system. However, the 2s complement system has the advantage of not requiriıg an end-around carry during addition.

Four situations may occur in adding two numbers when the 2s complement system is used:

1 When both numbers are positive, the situation is completely identical with that in case 1 in the 1s complement system.

2 When one number is positive and the other negative, and the large number is the positive number, a carry will be generated through the sign bit. This carry may be discarded, since the outputs of the adder are correct, as shown below:

$$\begin{array}{rcl} +0111 & = & \underline{0}0111 \\ \underline{-0011} & = & \underline{+\underline{1}1101} \\ +0100 & \ulcorner & \underline{0}0100 \\ & \llcorner\!\!\rightarrow & \text{carry is discarded} \end{array} \qquad \begin{array}{rcl} +1000 & = & \underline{0}1000 \\ \underline{-0111} & = & \underline{+\underline{1}1001} \\ +0001 & \ulcorner & \underline{0}0001 \\ & \llcorner\!\!\rightarrow & \text{carry is discarded} \end{array}$$

3 When a positive and negative number are added and the negative number is the larger, no carry will result in the sign bit, and the answer will again be correct as it stands:

[7] Generally this 1 is "sneaked in" during calculation, as will be shown.

$$\begin{array}{rcr}
+0011 & = & \underline{0}0011 \\
-0100 & = & \underline{1}1100 \\
\hline
-0001 & & \underline{1}1111
\end{array}
\qquad
\begin{array}{rcr}
+0100 & = & \underline{0}0100 \\
-1000 & = & \underline{1}1000 \\
\hline
-0100 & & \underline{1}1100
\end{array}$$

Note: A 1 must be added to the least significant bit of a 2s complement negative number in converting it to a magnitude. For example,

$$\begin{array}{rcrl}
\underline{1}0011 & = & 1100 & \text{form the 1s complement} \\
 & & 0001 & \text{add 1} \\
\cline{3-3}
 & & -1101 &
\end{array}$$

When both numbers are the same magnitude, the result is as follows:

$$\begin{array}{rcr}
+0011 & = & \underline{0}0011 \\
-0011 & = & \underline{1}1101 \\
\hline
0000 & & \underline{0}0000
\end{array}$$

When a positive and a negative number of the same magnitude are added, the result will be a positive zero.

4 When two negative numbers are added, a carry will be generated in the sign bit and also in the bit to the right of the sign bit. This will cause a 1 to be placed in the sign bit, which is correct, and the carry from the sign bit may be discarded.

$$\begin{array}{rcl}
-0011 & = & \underline{1}1101 \\
-0100 & = & \underline{1}1100 \\
\hline
-0111 & & \ulcorner\underline{1}1001 \\
 & & \llcorner\!\rightarrow \text{carry is discarded}
\end{array}
\qquad
\begin{array}{rcl}
-0011 & = & \underline{1}1101 \\
-1011 & = & \underline{1}0101 \\
\hline
-1110 & & \ulcorner\underline{1}0010 \\
 & & \llcorner\!\rightarrow \text{carry is discarded}
\end{array}$$

For parallel machines, addition of positive and negative numbers is quite simple, since any overflow from the sign bit is simply discarded. Thus for the parallel adder in Fig. 5.5 we simply add another full-adder, with X_4 and Y_4 as inputs and with the CARRY line C_o from the full-adder, which adds X_3 and Y_3, connected to the carry input C_i to the full-adder for X_4 and Y_4. A 0 is placed on the C_i input to the adder connected to X_0 and Y_0.

This simplicity in adding and subtracting has made the 2s complement system the most popular for parallel machines. In fact, when signed-magnitude systems are used, the numbers generally are converted to 2s complement before addition of negative numbers or subtraction is performed. Then the numbers are changed back to signed magnitude.

ADDITION AND SUBTRACTION IN A PARALLEL ARITHMETIC ELEMENT

5.9 We now examine the design of a gating network which will either add or subtract two numbers. The network is to have an ADD input line and a SUBTRACT input line as well as the lines that carry the representation of the numbers to be added or subtracted. When the ADD line is a 1, the sum of the numbers is to be on the output lines; and when the SUBTRACT line is a 1, the difference is to be on the output lines. If both ADD and SUBTRACT are 0s, the output is to be 0.

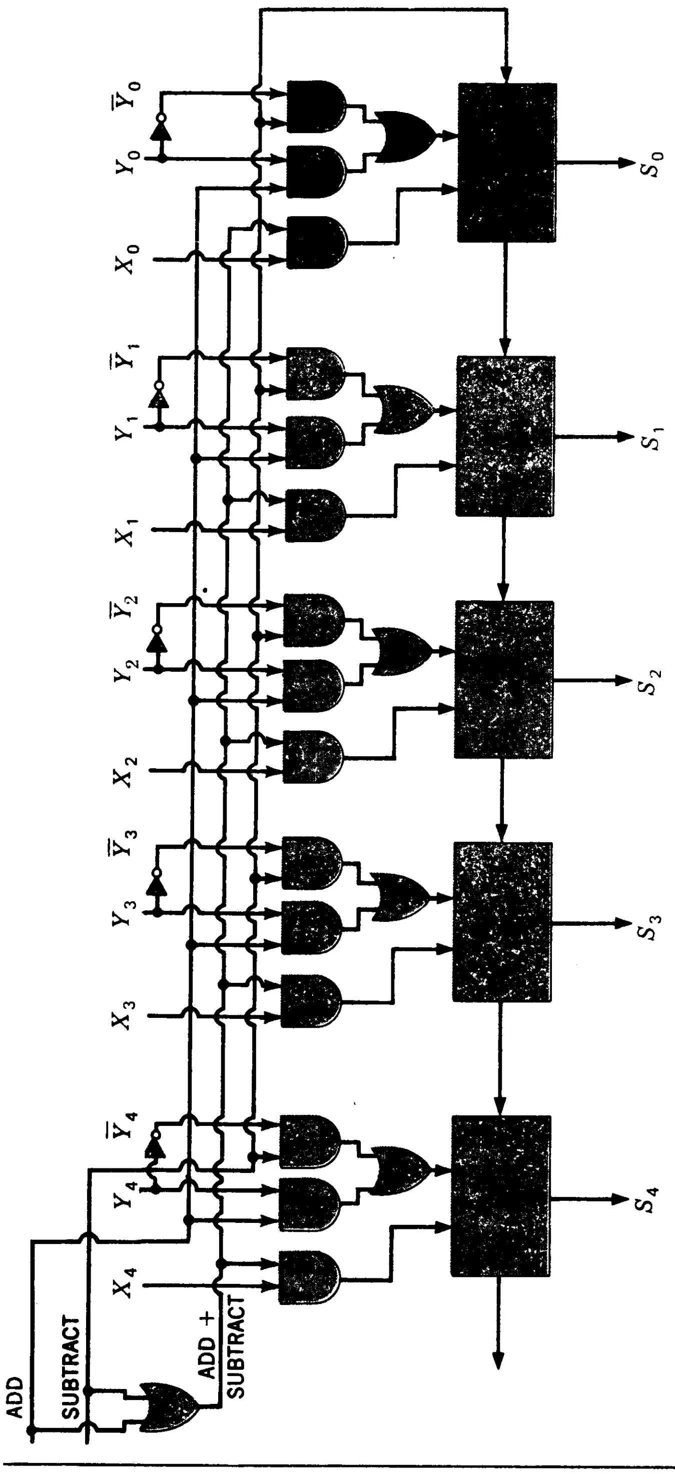
First we note that if the computer is capable of adding both positive and negative numbers, subtraction may be performed by complementing the subtrahend and then adding. For instance, 8 − 4 yields the same result as 8 + (−4), and 6 − (−2) yields the same result as 6 + 2. Subtraction therefore may be performed by an arithmetic element capable of adding, by forming the complement of the subtrahend and then adding. For instance, in the 1s complement system, four cases may arise:

```
                      TWO POSITIVE NUMBERS
 00011                                                  00011
-00001      complementing the subtrahend                11110
            and adding                             ┌──  00001
                                                   └─►carry 1
                                                        00010

                      TWO NEGATIVE NUMBERS
 11101                                                  11101
-11011              complementing                       00100
                                                   ┌──  00001
                                                   └─►carry 1
                                                        00010

   POSITIVE MINUEND,                  NEGATIVE MINUEND,
  NEGATIVE SUBTRAHEND                POSITIVE SUBTRAHEND
   00010   00010                       10101   10101
  -11101 = 00010                      -00010 = 11101
           00100                             ┌── 10010
                                             └─►carry 1
                                                 10011
```

The same basic rules apply to subtraction in the 2s complement system, except that any carry generated in the sign-bit adders is simply dropped. In this case the 2s complement of the subtrahend is formed, and then the complemented number is added to the minuend with no end-around carry.

We now examine the implementation of a combined adder and subtracter network. The primary problem is to form the complement of the number to be subtıacted. This complementation of the subtrahend may be performed in several ways. For the 1s complement system, if the storage register is composed of flip-flops, the 1s complement can be formed by simply connecting the complement of each input to the adder. The 1 which must be added to the least significant position to form a 2s complement may be added when the two numbers are added by connecting a 1 at the CARRY input of the adder for the least significant bits.

A complete logical circuit capable of adding or subtracting two signed 2s complement numbers is shown in Fig. 5.6. One number is represented by X_4, X_3, X_2, X_1, and X_0, and the other number by Y_4, Y_3, Y_2, Y_1, and Y_0. There are two



To add: the ADD line is made A1

To subtract: the SUBTRACT line is made A1

Numbers are to be in 2s complement form

FIGURE 5.6

Parallel addition and subtraction.

control signals, ADD and SUBTRACT. If neither control signal is a 1 (that is, both are 0s), then the outputs from the five full-adders, which are S_4, S_3, S_2, S_1, and S_0, will all be 0s. If the ADD control line is made a 1, the sum of the number X and the number Y will appear as S_4, S_3, S_2, S_1, and S_0. If the SUBTRACT line is made a 1, the difference between X and Y (that is, $X - Y$) will appear on S_4, S_3, S_2, S_1, and S_0.

Notice that the AND-to-OR gate network connected to each Y input selects either Y or $\overline{Y}$, so that, for instance, an ADD causes Y_1 to enter the appropriate full-adder, while a SUBTRACT causes $\overline{Y}_1$ to enter the full-adder.

To either add or subtract, each X input is connected to the appropriate full-adder. When a subtraction is called for, the complement of each Y flip-flop is gated into the full-adder, and a 1 is added by connecting the SUBTRACT signal to the C_i input of the full-adder for the lowest order bits X_0 and Y_0. Since the SUBTRACT line will be a 0 when we add, a 0 carry will be on this line when addition is performed.

The simplicity of the operation of Fig. 5.6 makes 2s complement addition and subtraction very attractive for computer use, and it is the most frequently used system.[8]

The configuration in Fig. 5.6 is the most frequently used for addition and subtraction because it provides a simple, direct means for either adding or subtracting positive or negative numbers. Quite often the S_4, S_3, . . . , S_0 lines are gated back into the X flip-flops, so that the sum or difference of the numbers X and Y replaces the original value of X.

An important consideration is overflow. In digital computers, an *overflow* is said to occur when the performance of an operation results in a quantity beyond the capacity of the register (or storage register) which is to receive the result. Since the registers in Fig. 5.6 have a sign bit plus 4 magnitude bits, they can store from $+15$ to -16 in 2s complement form. Therefore, if the result of an addition or a subtraction were greater than $+15$ or less than -16, we would say that an overflow had occurred. Suppose we add $+8$ to $+12$; the result should be $+20$, and this cannot be represented (fairly) in 2s complement on the lines S_4, S_3, . . . , S_0. The same thing happens if we add -13 and -7 or if we subtract -8 from $+12$. In each case, logical circuitry is used to detect the overflow condition and signal the computer control element. Various options are then available, and what is done can depend on the type of instruction being executed. (Deliberate overflows are sometimes used in double-precision routines. Multiplication and division use the results as are.) We defer this topic to Chap 9. except that one of the questions at the end of the chapter asks for a circuit to test for overflow.

The parallel adder-subtracter configuration in Fig. 5.6 is quite important, and it is instructive to try adding and subtracting several numbers in 2s complement form, using pencil and paper and this logic circuit.

[8] A 1s complement parrallel adder-subtracter can be made by connecting the CARRY-OUT line for the X_0, Y_0 adder to the CARRY-IN line for the X_4, Y_4 adder (disconnecting the SUBTRACT line to this full-adder, of course).

FULL-ADDER DESIGNS

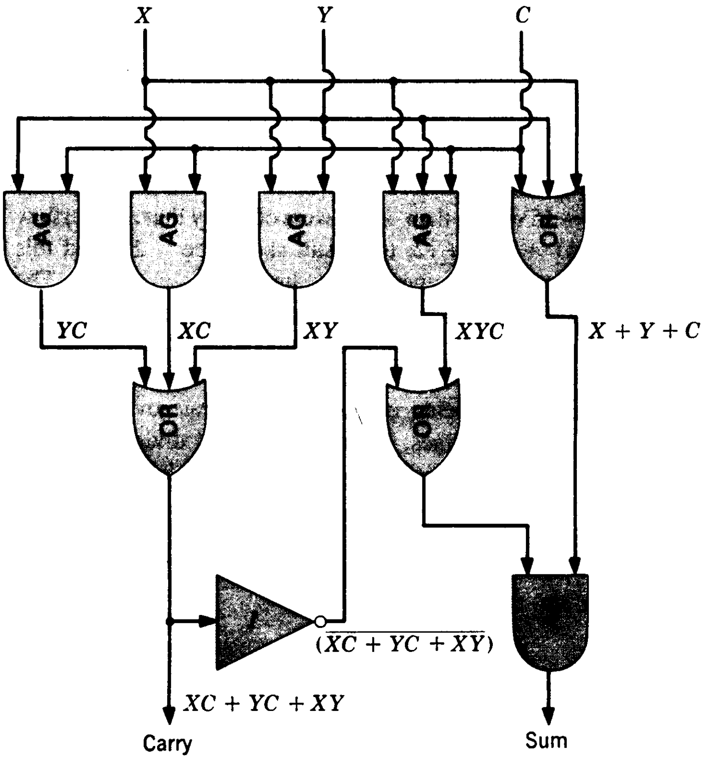
5.10 The full-adder is a basic component of an arithmetic element. Figure 5.3 illustrated the block diagram symbol for the full-adder, along with a table of combinations for the input-output values and the expressions describing the SUM and CARRY lines. Succeeding figures and text described the operation of the full-adder. Notice that a parallel addition system requires one full-adder for each bit in the basic word.

There are, of course, many gate configurations for full binary adders. Examples of an IBM adder and an MSI package containing two full-adders follow.

1 *Full binary adder* Figure 5.7 illustrates the full binary adder configuration used in several IBM general-purpose digital computers. There are three inputs to the circuit: The X input is from one of the storage devices in the accumulator, the Y input is from the corresponding storage device in the register to be added to the accumulator register, and the third input is the CARRY input from the adder for the next least significant bit. The two outputs are the SUM output and the CARRY output. The SUM output will contain the sum value for this particular digit of the output. The CARRY output will be connected to the CARRY input of the next most significant bit's adder (refer to Fig. 5.5).

FIGURE 5.7

Full-adder used in IBM machines.



$$[\overline{(XC + YC + XY)} + XYC]\,(X + Y + C) = \bar{X}\bar{Y}C + \bar{X}Y\bar{C} + X\bar{Y}\bar{C} + XYC$$

The outputs from the three AND gates connected directly to the X, Y, and C inputs are logically added by the OR gate circuit directly beneath. If either the X and Y, X and C, or Y and C input lines contain a 1, there should be a carry output. The output of this circuit, written in logical equation form, is shown on the figure. This may be compared with the expression derived in Fig. 5.3.

The derivation of the SUM output is not so straightforward. The CARRY output expression $XY + XC + YC$ is first inverted (complemented), yielding $(\overline{XY + XC + YC})$. The logical product of X, Y, and C is formed by an AND gate and is logically added to this, forming $(\overline{XY + XC + YC}) + XYC$. The logical sum of X, Y, and C is then multiplied times this, forming the expression

$$[(\overline{XY + XC + YC}) + XYC](X + Y + C)$$

When it is multiplied out and simplified, this expression will be $\bar{X}\bar{Y}C + \bar{X}Y\bar{C} + X\bar{Y}\bar{C} + XYC$, the expression derived in Fig. 5.3. Tracing through the logical operation of the circuit for various values will indicate that the SUM output will be 1 when only one of the input values is equal to 1 or when all three input values are equal to 1. For all other combinations of inputs, the output value will be a 0.

2 *Two full-adders in an IC container* Figure 5.8 shows two full-adders. This package was developed for integrated circuits using transistor-transistor logic (TTL). The entire circuitry is packaged in one IC container. The maximum delay from an

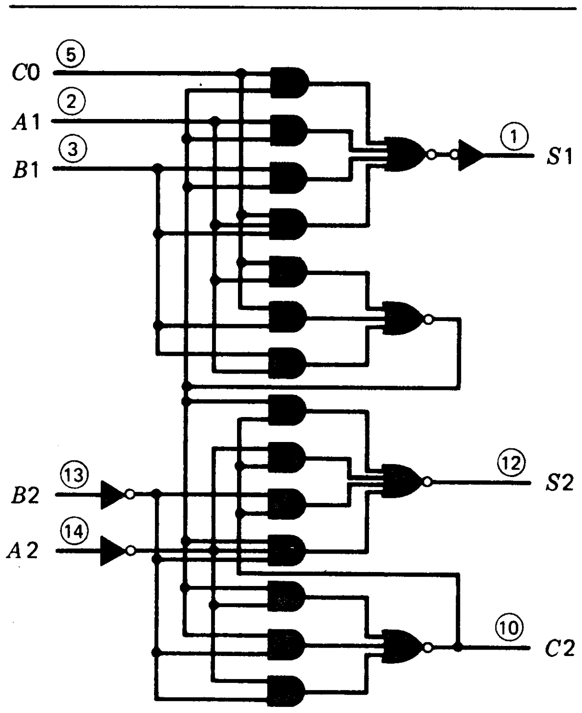


FIGURE 5.8

Two full-adders in an IC container. (*Texas Instruments.*)

input change to an output change for an S output is on the order of 8 nanoseconds (ns).[9] The maximum delay from any input to the $C2$ output is about 6 ns.

The amount of delay associated with each carry is an important figure in evaluating a full-adder for a parallel system, because the time required to add two numbers is determined by the maximum time it takes for a carry to propagate through the adders. For instance, if we add 01111 to 10001 in the 2s complement system, the carry generated by the 1s in the least significant digit of each number must propagate through four carry stages and a sum stage before we can safely gate the sum into the accumulator. A study of the addition of these two numbers by using the configuration in Fig. 5.5 will make this clear. The problem is called the *carry-ripple problem.*

A number of techniques are used in high-speed machines to alleviate this problem. The most used is a bridging, or carry-look-ahead, circuit which calculates the carry-out of a number of stages simultaneously and then delivers this carry to the succeeding stages. (This is covered in a later section.)

BINARY-CODED-DECIMAL ADDER

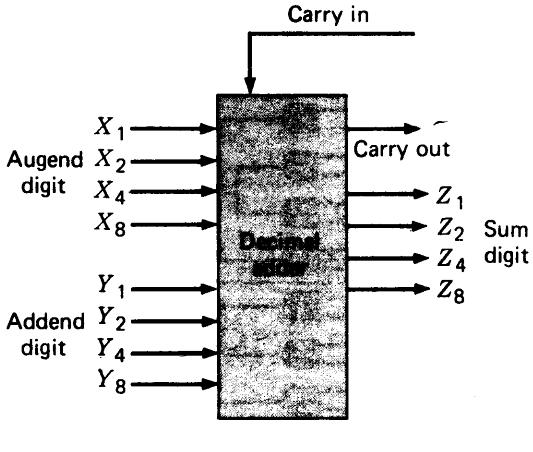
5.11 Arithmetic units which perform operations on numbers stored in BCD form must have the ability to add 4-bit representations of decimal digits. To do this, a BCD adder is used. A block diagram symbol for an adder is shown in Fig. 5.9. The adder has an augend digit input consisting of four lines, an addend digit input of four lines, a carry-in and a carry-out, and a sum digit with four output lines. The augend digit, addend digit, and sum digit are each represented in 8, 4, 2, 1 BCD form.

The purpose of the BCD adder in Fig. 5.9 is to add the augend and addend digits and the carry-in and produce a sum digit and carry-out. This adder could be designed by using the techniques described in Chap. 3 and the rules for decimal

[9] 1 ns = 10^{-9} s.

FIGURE 5.9

Serial-parallel addition.



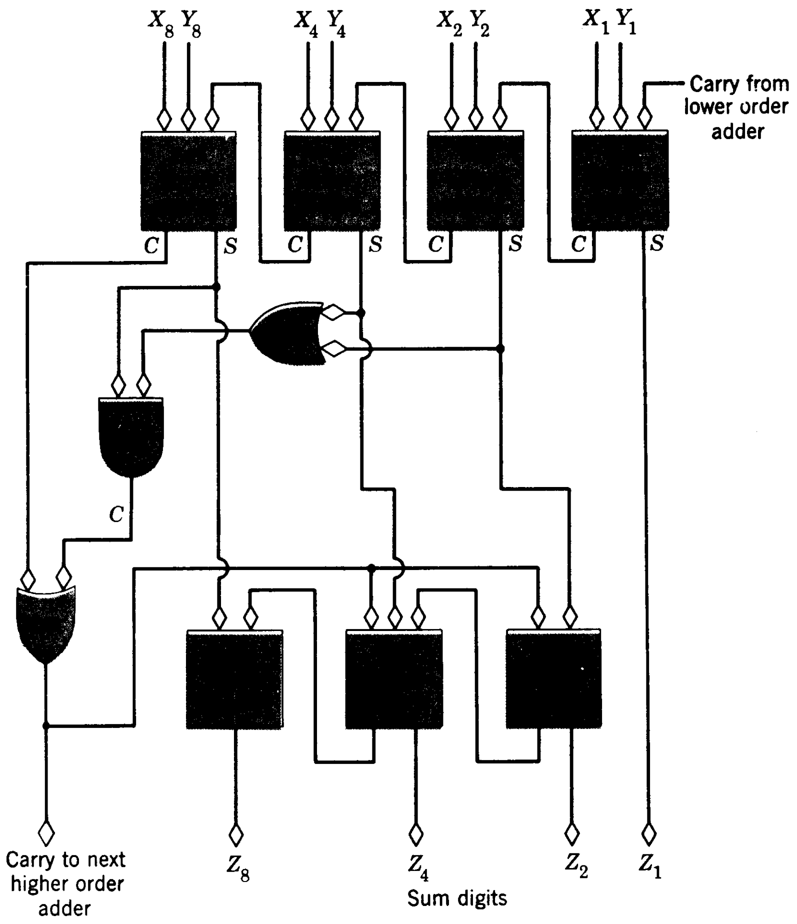


FIGURE 5.10

BCD adder.

addition. It is also possible to make a BCD adder by using full-adders and AND or OR gates. An adder made in this way is shown in Fig. 5.10.

There are eight inputs to the BCD adder; four X_i, or augend, inputs; and four Y_i, or addend, digits. Each input will represent a 0 or a 1 during a given addition. If 3 (0011) is to be added to 2 (0010), then $X_8 = 0$, $X_4 = 0$, $X_2 = 1$, and $X_1 = 1$; $Y_8 = 0$, $Y_4 = 0$, $Y_2 = 1$, and $Y_1 = 0$.

The basic adder in Fig. 5.10 consists of the four binary adders at the top of the figure and performs base-16 addition when the intent is to perform base-10 addition. Thus some provision must be made to (1) generate carries and (2) correct sums greater than 9. For instance, if 3_{10} (0011) is added to 8_{10} (1000), the result should be 1_{10} (0001) with a carry generated.

The actual circuitry which determines when a carry is to be transmitted to the next most significant digits to be added consists of both the full binary adder to which sum (S) outputs from the adders for the 8, 4, 2 inputs are connected and the OR gate to which the carry (C) from the eight-position bits is connected. An examination of the addition process indicates that a carry should be generated when the 8 AND 4, or 8 AND 2, or 8 AND 4 AND 2 sum outputs from the base-16 adder represent 1s, or when the CARRY output from the eight-position adder contains a 1. (This occurs when 8s or 9s are added.)

Whenever the sum of two digits exceeds 9, the CARRY TO NEXT HIGHER ORDER ADDER line contains a 1 for the adder in Fig. 5.10.

A further difficulty arises when a carry is generated. If 7_{10} (0111) is added to 6_{10} (0110), a carry will be generated, but the output from the base-16 adder will be 1101. This 1101 does not represent any decimal digit in the 8, 4, 2, 1 system and must be corrected. The method used to correct this is to add 6_{10} (0110) to the sum from the base-16 adders whenever a carry is generated. This addition is performed by adding 1s to the weight 4 and weight 2 position output lines from the base-16 adder when a carry is generated. The two half-adder and the full-adders at the bottom of Fig. 5.10 perform this function. Essentially, then, the adder performs base-16 addition and corrects the sum, if it is greater than 9, by adding 6. Several examples of this are shown below.

$$8 + 7 = 15 \qquad 1000 + 0111 = \quad \begin{array}{cccccl} & (8) & (4) & (2) & (1) & \\ & 1 & 1 & 1 & 1 & \\ & +0 & 1 & 1 & 0 & \\ \cline{2-5} 1 & 0 & 1 & 0 & 1 & = 5 \end{array}$$

└—with a carry generated

$$9 + 5 = 14 \qquad \begin{array}{cccccl} & (8) & (4) & (2) & (1) & \\ & 1 & 0 & 0 & 1 & \\ & 0 & 1 & 0 & 1 & \\ \cline{2-5} & 1 & 1 & 1 & 0 & \\ & +0 & 1 & 1 & 0 & \\ \cline{2-5} 1 & 0 & 1 & 0 & 0 & = 4 \end{array}$$

└—with a carry generated

Figure 5.11 shows a complete BCD adder in an IC package.[10] The inputs are digits A and digits B, and the outputs are S. A carry-in and a carry-out are included. The circuit line used is CMOS.

POSITIVE AND NEGATIVE BCD NUMBERS

5.12 The techniques for handling BCD numbers greatly resemble those for handling binary numbers. A sign bit is used to indicate whether the number is positive or negative, and there are three methods of representing negative numbers which must be considered. The first and most obvious method is, of course, to represent a negative number in true magnitude form with a sign bit, so that −645 is rep-

[10] The IC packages in Figs. 5.11, 5.13, 5.14, and 5.15 are typical BCD MSI packages. The notation $A1$, $A2$, $A3$, $A4$ (instead of X_1, X_2, X_4, X_8) is often used for the 4 bits of a BCD digit, and the weights 1, 2, 4, 8 are understood. Thus a BCD digit in $B1$, $B2$, $B3$, $B4$ would have weight 1 on $B1$, weight 2 on $B2$, weight 4 on $B3$, and weight 8 on $B4$.

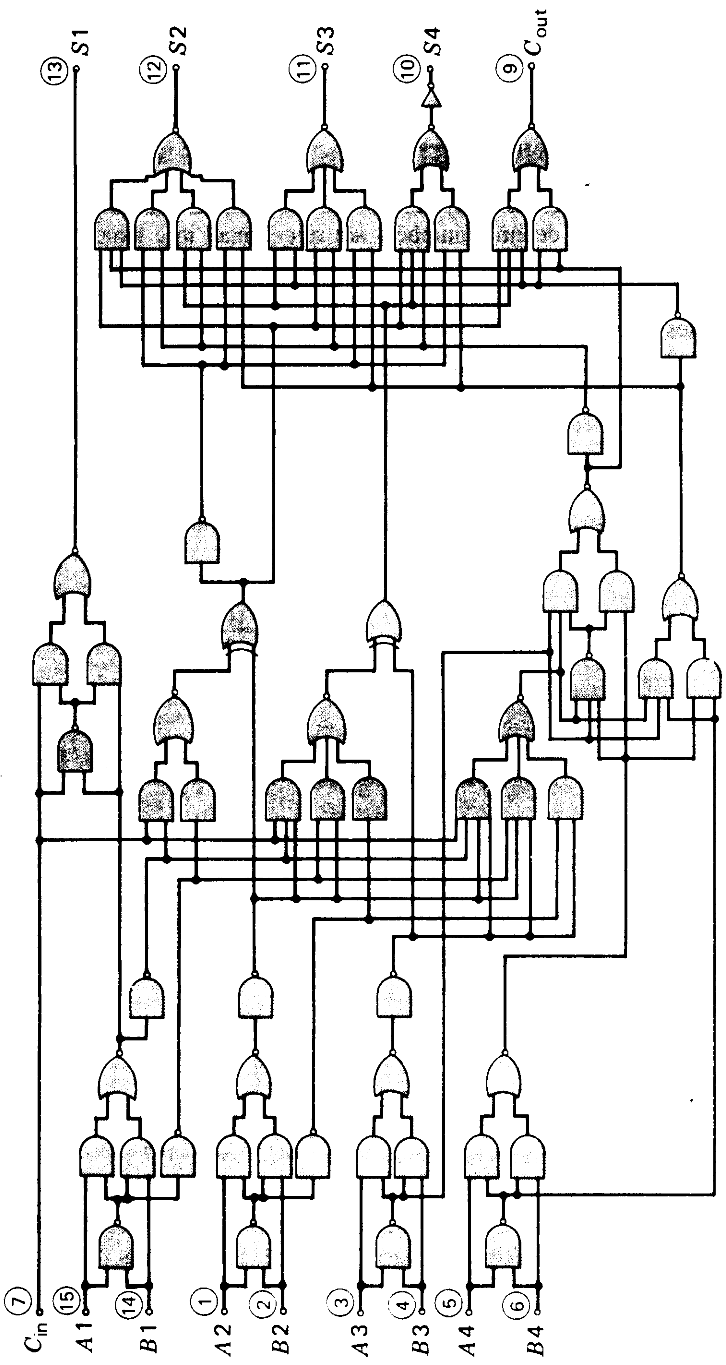


FIGURE 5.11

Complete BCD adder in an IC package.

resented as $\underline{1}645$. The other two possibilities are to represent negative numbers in a 9s or a 10s complement form, which resembles the binary 1s and 2s complement forms.

ADDITION AND SUBTRACTION IN THE 9S COMPLEMENT SYSTEM

5.13 When decimal numbers are represented in a binary code in which the 9s complement is formed when the number is complemented, the situation is roughly the same as when the 1s complement is used to represent a binary number. Four cases may arise: Two positive numbers may be added; a positive and negative number may be added, yielding a positive result; a positive and negative number may be added, yielding a negative result; and two negative numbers may be added. Since there is no problem when two positive numbers are added, we illustrate the three latter situations.

NEGATIVE AND POSITIVE NUMBER—POSITIVE SUM

$$\begin{array}{rcr} +692 & = & \underline{0}692 \\ \underline{-342} & = & \underline{\underline{1}657} \\ +350 & & \underline{0}349 \\ & & \underline{1} \\ & & \underline{0}350 \end{array}$$

POSITIVE AND NEGATIVE NUMBER—NEGATIVE SUM

$$\begin{array}{rcrcr} -631 & = & \underline{1}368 & & \\ \underline{+342} & = & \underline{\underline{0}342} & & \\ -289 & & \underline{1}710 & = & -289 \end{array}$$

TWO NEGATIVE NUMBERS

$$\begin{array}{rcrcr} -248 & = & \underline{1}751 & & \\ \underline{-329} & = & \underline{\underline{1}670} & & \\ -577 & & \underline{1}421 & & \\ & & \underline{1} & & \\ & & \underline{1}422 & = & -577 \end{array}$$

The rules for handling negative numbers in the 10s complement system are the same as those for the binary 2s complement system in that no carry must be ended-around. Therefore, a parallel BCD adder may be constructed by using only the full BCD adder as the basic component, and all combinations of positive and negative numbers may be handled thus.

There is an additional complexity in BCD addition, however, because the 9s complement of a BCD digit cannot be formed by simply complementing each bit in the representation. As a result, a gating block called a *complementer* must be used.

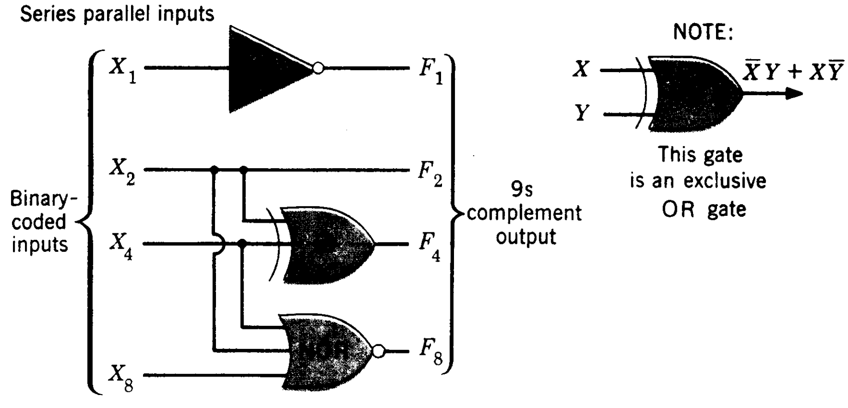


FIGURE 5.12

Logic circuit for forming 9s complement of 8, 4, 2, 1 BCD digits.

To illustrate the type of circuit which may be used to form complements of the code groups for BCD numbers, a block diagram of a logical circuit which will form the 9s complement of a code group representing a decimal number in 8, 4, 2, 1 BCD form is shown in Fig. 5.12. There are four inputs to the circuit, X_1, X_2, X_4, and X_8. Each input carries a different weight: X_1 has weight 1, X_2 has weight 2, X_4 has weight 4, and X_8 has weight 8. If the inputs represent a decimal digit of the number to be complemented, the outputs will represent the 9s complement of the input digit. For instance, if the input is 0010 (decimal 2), the output will be 0111 (decimal 7), the 9s complement of the input.

Figure 5.13 shows a complete 9s complementer in an IC package. When the COMP input is a 1, the outputs $F1$–$F4$ represent the complement of the digit on $A1$–$A4$; but if COMP is a 0, the $A1$–$A4$ inputs are simply placed in $F1$–$F4$ without change.

By connecting the IC packages in Figs. 5.11 and 5.13, a BCD adder-subtracter can be formed as shown in Fig. 5.14. This shows a two-digit adder-subtracter IC package. To add the digits on the inputs, the ADD-SUBTRACT input is made a 1; to subtract, this signal is made a 0. (Making the ZERO input a 1 will cause the value of B to pass through unchanged.)

BCD numbers may be represented in parallel form, as we have shown, but a mode of operation called *series-parallel* is often used. If a decimal number is written in binary-coded form, the resulting number consists of a set of code groups, each of which represents a single decimal digit. For instance, decimal 463 in a BCD 8, 4, 2, 1 code is 0100 0110 0011. Each group of 4 bits represents one decimal digit. It is convenient to handle each code group which represents a decimal digit as a unit, that is, in parallel. At the same time, since the word lengths for decimal computation are apt to be rather long, it is desirable to economize in the amount of equipment used.

The *series-parallel* system provides a compromise in which each code group is handled in parallel, but the decimal digits are handled sequentially. This requires four lines for each 8, 4, 2, 1 BCD character, each input line of which carries a different weight. The block diagram for an adder operating in this system is shown

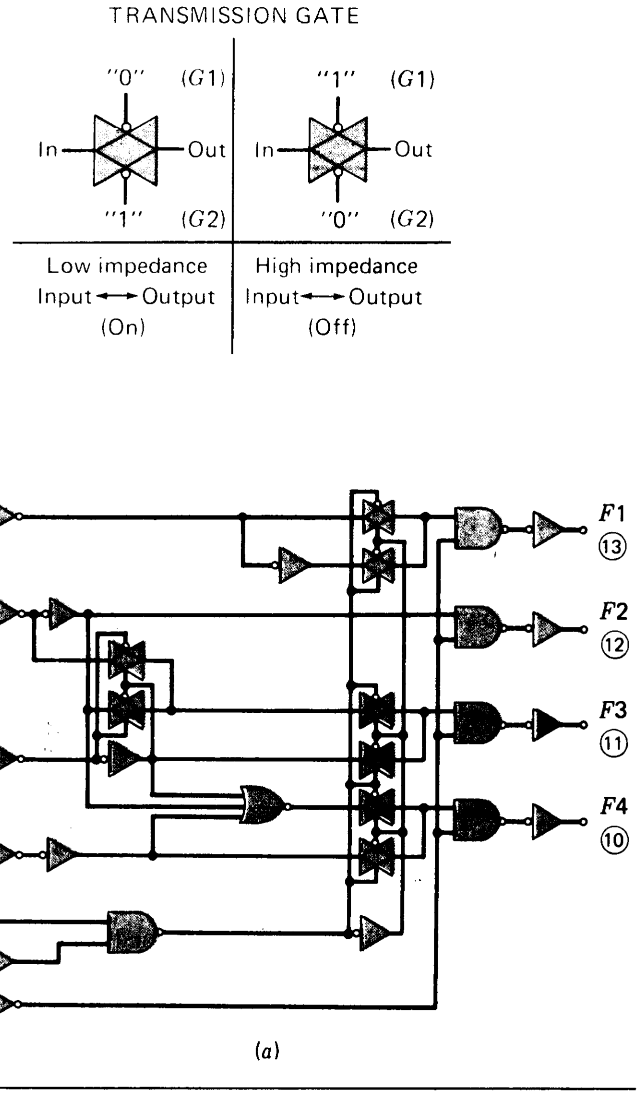


FIGURE 5.13

9s complementer in IC package. (*a*) Logic diagram. (*b*) Table of combinations.

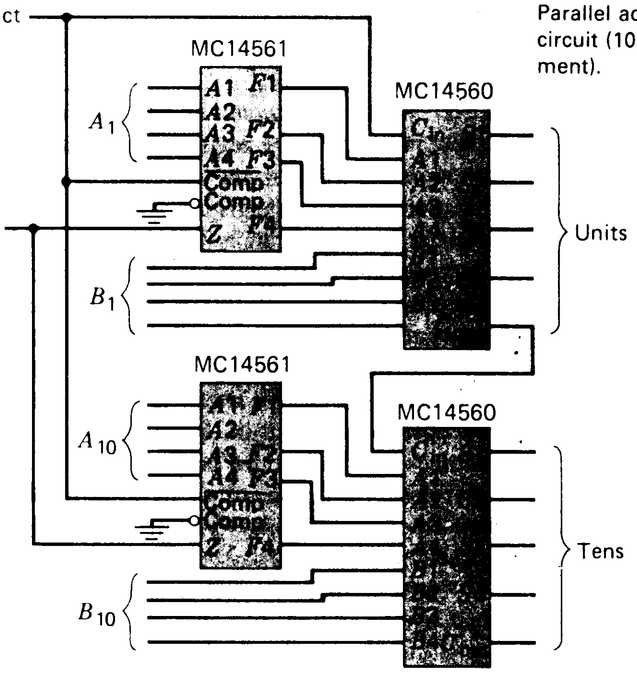
in Fig. 5.15. There are two sets of inputs to the adder; one consists of the four input lines which carry the coded digit for the addend, and the other four input lines carry a coded augend digit. The sets of inputs arrive sequentially from the *A* and *B* registers, each of which consists of four shift registers; the least significant addend and augend BCD digits arrive first, followed by the more significant decimal digits.

If the 8, 4, 2, 1 code is used, let 324 represent the augend and 238 the addend. The ADD signal will be a 0. First the adder will receive 0100 on the augend lines, and at the same time it will receive 1000 on the addend lines. After the first clock pulse, these inputs will be replaced by 0010 on the augend lines and

	DECIMAL EQUIVALENT INPUT	INPUTS				DECIMAL EQUIVALENT OUTPUT	OUTPUTS			
		$A4$	$A3$	$A2$	$A1$		$F1$	$F2$	$F3$	$F4$
	0	0	0	0	0	9	1	0	0	1
	1	0	0	0	1	8	1	0	0	0
	2	0	0	1	0	7	0	1	1	1
	3	0	0	1	1	6	0	1	1	0
	4	0	1	0	0	5	0	1	0	1
	5	0	1	0	1	4	0	1	0	0
	6	0	1	1	0	3	0	0	1	1
	7	0	1	1	1	2	0	0	1	0
	8	1	0	0	0	1	0	0	0	1
	9	1	0	0	1	0	0	0	0	0
Illegal BCD input codes	10	1	0	1	0	7	0	1	1	1
	11	1	0	1	1	6	0	1	1	0
	12	1	1	0	0	5	0	1	0	1
	13	1	1	0	1	4	0	1	0	0
	14	1	1	1	0	3	0	0	1	1
	15	1	1	1	1	2	0	0	1	0

(*b*)

FIGURE 5.13 (*Cont.*)



ZERO	ADD/SUBTRACT	RESULT
0	0	*B* plus *A*
0	1	*B* minus *A*
1	*X*	*B*

X = don't care

FIGURE 5.14

Parallel add-subtract circuit (10s complement).

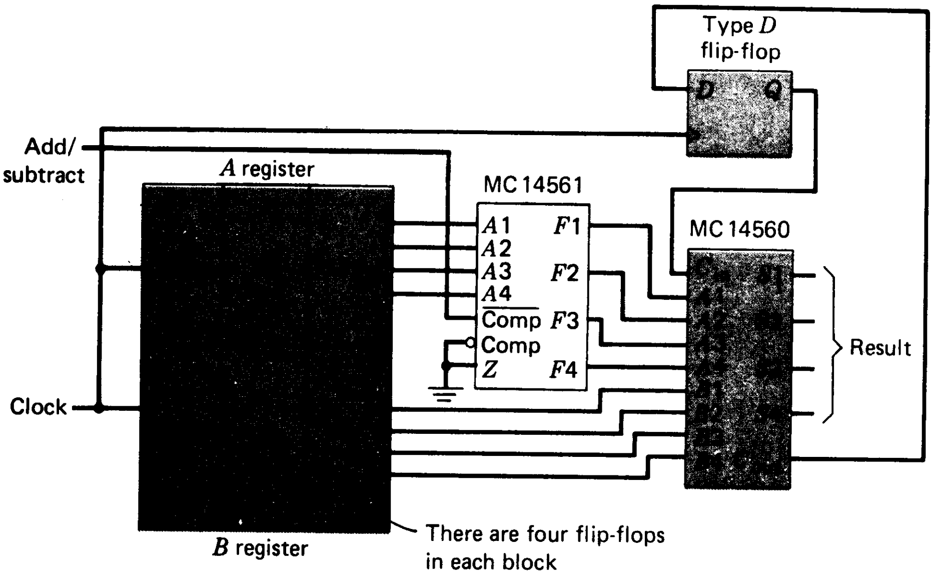


FIGURE 5.15

Series-parallel BCD adder-subtracter using shift register.

0011 on the addend lines. Before the first clock signal, the sum lines should contain 0010; and before the second, 0110. A carry will be generated during the addition of the first two digits; this will be delayed and added in using the *D* flip-flop. The process will continue until each of the three digits has been added. To subtract *B* from *A*, we have only to make the ADD-SUBTRACT input a 1 and then apply the clocks.

SHIFT OPERATION

5.14 A *shift operation* is an operation which moves the digits *stored* in a register to new positions in the register. There are two distinct shift operations: a shift-left operation and a shift-right operation. A *shift-left operation* moves each bit of information stored in a register to the left by some specified number of digits. Consider the six binary digits 000110, which we assume to be stored in a parallel binary register. If the contents of the register are shifted left 1, afterward the shift register will contain 001100. If a shift right of 1 is performed on the word 000110, afterward the shift register will contain 000011. The shifting process in a decimal register is similar: if the register contains $\underline{0}01234$, after a right shift of 1 the register will contain $\underline{0}00123$, or after a left shift of 1 the register will contain $\underline{0}12340$. The shift operation is used in the MULTIPLY and the DIVIDE instructions of most machines and is provided as an instruction which may be used by programmers. For instance, a machine may have instructions SHR and SHL, where the letters represent in mnemonic form the order for SHIFT RIGHT and SHIFT LEFT instructions.

A block diagram of logic circuitry for a single stage (flip-flop) in a register which can be shifted either left or right is shown in Fig. 5.16. As can be seen, the

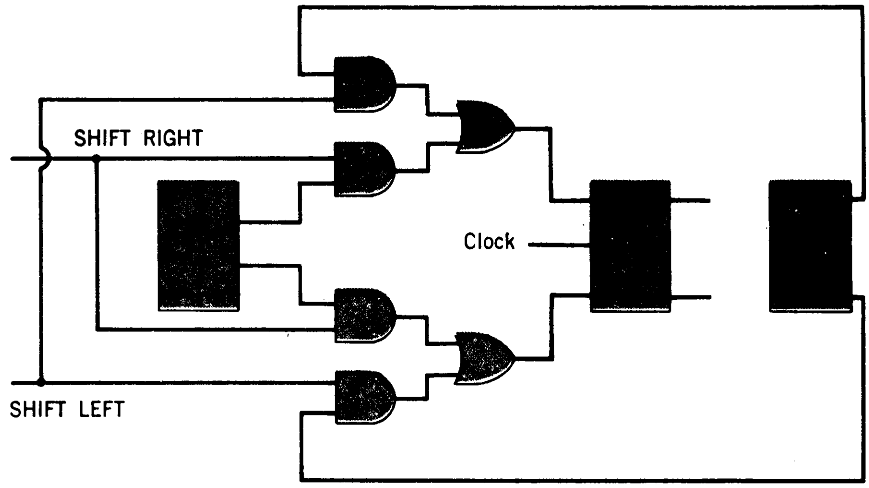


FIGURE 5.16

Shift-left and shift-right stages of register.

bit to the left is shifted into X when SHIFT RIGHT is a 1, and the bit to the right is shifted into X when SHIFT LEFT is a 1.

Figure 5.17 shows an MIS package which contains four flip-flops and gating circuitry so that the register can be shifted right or left and so that the four flip-flops can be parallel-loaded from four input lines W, X, Y, and Z. The circuits are TTL circuits and are clocked in parallel. By combining modules such as this one, a register of a chosen length can be formed which can be shifted left or right or parallel-loaded.

BASIC OPERATIONS

5.15 The arithmetic-logic unit of a digital computer consists of a number of registers in which information can be stored and a set of logic circuits which make it possible to perform certain operations on the information stored in the registers and between registers.

As we have seen, the data stored in a given flip-flop register may be operated on in the following ways:

1 The register may be reset to all 0s.

2 The contents of a register may be complemented to either 1s or 2s complement form for binary or to 9s or 10s complement form for decimal.

3 The contents of a register may be shifted right or left.

4 The contents of a register can be incremented or decremented.

Several operations between registers have been described. These include

1 Transferring the contents of one register to another register

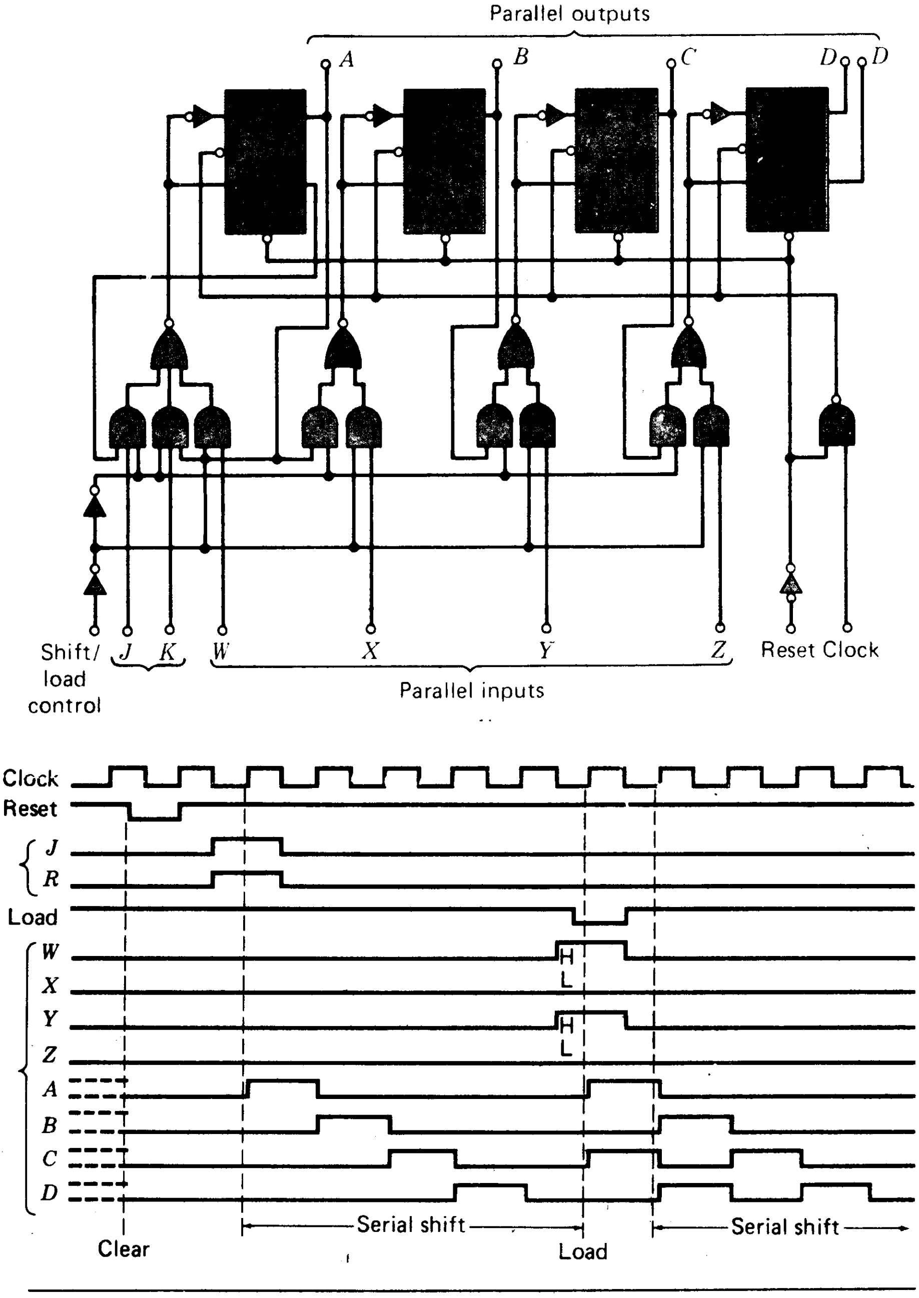


FIGURE 5.17

Shift register (model SN74195) with parallel-load ability. (*Texas Instruments.*)

2 Adding to or subtracting from the contents of one register the contents of anothèr register

Most arithmetic operations which an ALU performs consist of these or sequenced sets of these two types of operations. Complicated instructions, such as multiplication and division, can require a large number of these operations, but these instructions may be performed by using only sequences of the simple operations already described.

One other important point needs to be made. Certain operations which occur within instructions are *conditional;* that is, a given operation may or may not take place, depending on the value of certain bits of the numbers stored. For instance, it may be desirable to multiply using only positive numbers. In this case, the sign bits of the two numbers to be multiplied will be examined by control circuitry and if either is a 1, the corresponding number will be complemented before the multiplication begins. This operation, complementing of the register, is a conditional one.

Many different sequences of operations can yield the same result. For instance, two numbers could be multiplied by simply adding the multiplicand to itself the number of times indicated by the multiplier. If this were done with pencil and paper, 369 × 12 would be performed by adding 369 to itself 12 times. This would be a laborious process compared with the easier algorithm which we have developed for multiplying, but we would get the same result. The same principle applies to computer multiplication. Two numbers could be multiplied by transferring one of the numbers into a counter which counted downward each time an addition was performed, and then adding the other number to itself until the counter reached zero. This technique has been used, but much faster techniques are also used and will be explained.

Many algorithms have been used to multiply and divide numbers in digital computers. Division, especially, is a complicated process; and in decimal computers in particular, many different techniques are used. The particular technique used by a computer is generally based on the cost of the computer and the premium on speed for the computer. As in almost all operations, speed is expensive, and a faster division process generally means a more expensive computer.

To explain the operations of binary multiplication and division, we use a block diagram of a generalized binary computer. Figure 5.18 illustrates, in block diagram form, the registers of an ALU. The computer has three basic registers: an accumulator, a Y register, and a B register. The operations which can be performed have been described:

1 The accumulator can be cleared.

2 The contents of the accumulator can be shifted right or left. Further, the accumulator and the B register may be formed into one long shift register. If we then shift this register right two digits, the two least significant digits of the accumulator will be shifted into the first two places of the B register. Several left shifts will shift the most significant digits of the B register into the accumulator. Since there are 5 bits in the basic computer word, there are five binary storage devices in each register. A right shift of five places will transfer the contents of the accumulator into the B register, and a left shift of five places will shift the contents of the B register into the accumulator.

3 The contents of the Y register can be either added to or subtracted from the accumulator. The sum or difference is stored in the accumulator register.

4 Words from memory may be read into the Y register. To read a word into the accumulator, it is necessary to clear the accumulator, to read the word from memory into the Y register, and to add the Y register to the accumulator.

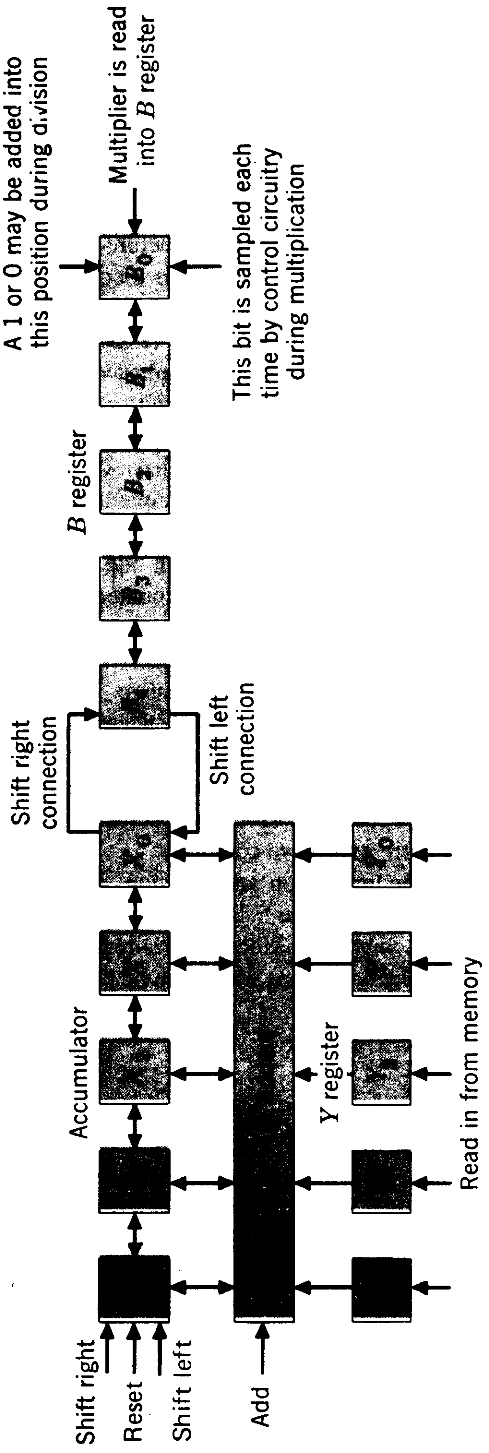

A 1 or 0 may be added into this position during division
Multiplier is read into B register
This bit is sampled each time by control circuitry during multiplication
B register
B_0
B_1
B_2
B_3
Shift right connection
Shift left connection
Accumulator
Y register
Read in from memory
Shift right
Reset
Shift left
Add

FIGURE 5.18

Generalized parallel arithmetic element.

An arithmetic element which can perform these operations on its registers can be sequenced to perform all arithmetic operations. It is, in fact, possible to construct a machine using fewer operations than these, but most general-purpose computers usually have an arithmetic element with at least these capabilities.

BINARY MULTIPLICATION

5.16 The process of multiplying binary numbers may be best examined by writing out the multiplication of two binary numbers:

```
   1001 = multiplicand
   1101 = multiplier
-------
   1001 )
  0000  |
 1001   } partial products
1001    )
-------
1110101 = product
```

The important thing to notice in this process is that there are really only two rules for multiplying a single binary *number* by a binary *digit:* (1) If the multiplier digit is a 1, the multiplicand is simply copied. (2) If the multiplier digit is a 0, the product is 0. The above example illustrates these rules as follows: The first digit to the right of the multiplier is a 1; therefore, the multiplicand is copied as the first partial product. The next digit of the multiplier to the left is a 0; so the partial product is a 0. Each time a partial product is formed, it is shifted one place to the left of the previous partial product. Even if the partial product is a 0, the next partial product is shifted one place to the left of the previous partial product. This process is continued until all the multiplier digits have been used, and then the partial products are summed.

The three operations which the computer must be able to perform to multiply in this manner are, therefore, (1) to sense whether a multiplier bit is either a 1 or a 0, (2) to shift partial products, and (3) to add the partial products.

It is not necessary to wait until all the partial products have been formed before they are summed. They may be summed two at a time. For instance, starting with the first two partial products in the above example, we have

```
 1001
0000
-----
01001
```

Then the next partial product may be added to this sum, displacing it one position to the left:

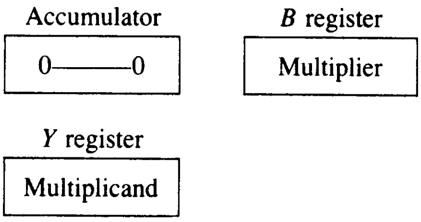
```
 01001
1001
------
101101
```

And finally,

$$\begin{array}{r} 101101 \\ \underline{1001} \\ 1110101 \end{array}$$

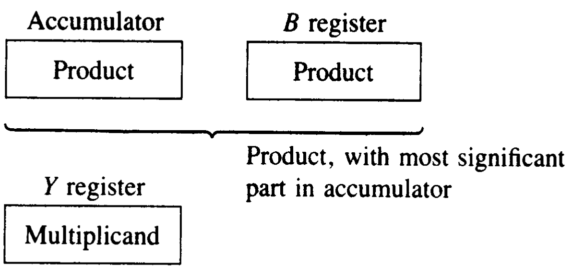
A multiplier can be constructed in just this fashion. By sampling each bit of the multiplier in turn, adding the multiplicand into some register, and then shifting the multiplicand left each time a new multiplier bit is sampled, a product could be formed of the sum of the partial products. In fact, the process of multiplying in most binary machines is performed in a manner very similar to this.

To examine the typical technique for multiplying, the generalized arithmetic elements in Fig. 5.18 are used. Let the multiplier be stored in the *B* register, and the multiplicand in the *Y* register; the accumulator contains all 0s as shown:



Let us also assume that both multiplier and multiplicand are positive. If either is negative, it must be converted to positive form before the multiplication begins. The sign bits will therefore be 0s.

The desired result format is shown, with the product being the combined accumulator and *B* register:



A multiplication requires *n basic steps,* where *n* is the number of bits in the magnitude of the numbers to be multiplied, and a final right shift to position the product. Each basic step is initiated by the control circuitry examining the rightmost bit in the *B* register. The basic step is as follows.

BASIC STEP

If the rightmost bit in the *B* register is a 0, the combined accumulator and *B* register is shifted right one place. If the rightmost bit in the *B* register is a 1, the number in the *Y* register is added to the contents of the accumulator, and then the combined accumulator and *B* register is shifted right one place.

After each basic step, the new rightmost bit of the *B* register is examined again, and the next of the *n* steps is initiated.

Let us consider the same multiplication that was used in the previous example, that is, 1101 × 1001; where 1101 is the multiplier. In the beginning the accumulator contains 00000, the *B* register 01101, and the *Y* register 01001 (the leftmost 0s are sign bits). Four steps and a final shift will be required.

1 Since the rightmost bit of the *B* register contains a 1 (the least significant bit of the multiplier), during the first step the contents of the *Y* register are added to the accumulator, and the combined accumulator and *B* register are shifted to the right. The second least significant bit of the multiplier now occupies the rightmost bit of the *B* register and controls the next operation. The *Y* register still contains the multiplicand 01001, the contents of the accumulator are 00100, and the contents of the *B* register are 10110.

2 The rightmost bit of the *B* register is a 0, and since it controls the next operation, a SHIFT RIGHT signal is initiated and the accumulator and *B* register are shifted right, giving 00010 in the accumulator and 01011 in the *B* register.

3 A 1 is now in the rightmost bit of the *B* register. So the *Y* register is added to the accumulator again, and the combined accumulator and *B* register are shifted right, giving 00101 in the accumulator and 10101 in the *B* register.

4 The least significant bit of the *B* register is another 1; so the *Y* register is added to the accumulator and the accumulator is shifted right. After the above shift right, the combined accumulator and *B* register contain 0011101010. A final right shift gives 0001110101, the correct product for our integer number system. The most significant digits are stored in the accumulator, and the least significant digits in the *B* register.

ACCUMULATOR	*B* REGISTER	
00000	01101	At beginning
00100	10110	After step 1
00010	01011	After step 2
00101	10101	After step 3
00111	01010	After step 4
00011	10101	After shift right

Now the reason for the combined accumulator and *B* register can be seen. The product of two 5-bit signed numbers can contain up to nine significant digits (including the sign bit); and so two 5-bit registers, not one, are required to hold the product. The final product is treated like a 10-bit number extending through the two registers with the leftmost bits (most significant bits) in the left register, the rightmost bits (least significant bits) in the right register, and the least significant binary digit in the rightmost bit. Thus our result in the two combined registers is 0001110101, which is + 117 in decimal.

The control circuitry is designed to perform the examination of the multiplier bits, then either shift or add and shift the correct number of times, and stop. In this case, the length of the multiplier, or *Y* register, is 4 bits plus a sign bit; so four such steps are performed. The general practice is to examine each bit of the computer word except the sign bit, in turn. For instance, if the basic computer word is 25 bits (that is, 24 bits in which the magnitude of a number is stored plus a sign bit), each time a multiplication is performed, the computer will examine 24 bits, each in turn, performing the add-and-shift or just the shift operation 24 times. This makes the multiplication operation longer than such operations as add or subtract. Some parallel computers double their normal rate of operation during multiplication: if the computer performs such operations as addition, complementation, transfers, etc., at a rate of 4 MHz/s for ordinary instructions, the rate will be increased to 8 MHz for the add-and-shift combinations performed during multiplying. Some computers are able to shift right while adding; that is, the sum of the accumulator and *Y* register appears shifted one place to the right each time, and the shift-right operation after each addition may be omitted.

The sign bits of the multiplier and multiplicand may be handled in a number of ways. For instance, the sign of the product can be determined by means of control circuitry before the multiplication procedure is initiated. This sign bit is stored during the multiplication process, after which it is placed into the sign bit of the accumulator, and then the accumulator is complemented, if necessary. Therefore, the sign bits of the multiplier and multiplicand are examined first. If they are both 0s, the sign of the product should be 0; if both are 1s, the sign of the product should be 0; and if either but not both are a 1, the sign of the product should be 1. This information, retained in a flip-flop while the multiplication is taking place, may be transferred into the sign bit afterward. If the computer handles numbers in the 1s or 2s complement system, both multiplier and multiplicand may be handled as positive magnitudes during the multiplication. And if the sign of either number is negative, the number is complemented to a positive magnitude before the multiplication begins. Sometimes the multiplication is performed on complemented numbers by using more complicated algorithms. These are described in the Bibliography.

DECIMAL MULTIPLICATION

5.17 Decimal multiplication is a more involved process than binary multiplication. Whereas the product of a binary digit and a binary number is either the number or 0, the product of a decimal digit and decimal number involves the use of a multiplication table plus carrying and adding. For instance,

$$7 \times 24 = 7 \times 4 + 7 \times 20 = 28 + 140 = 168$$

Even the multiplying of two decimal digits may involve two output digits; for instance, 7 × 8 equals 56. In the following discussion we call the two digits which may result when a decimal digit is multiplied by a decimal digit the *left-hand* and the *right-hand digits*. Thus for 3 × 6 we have 1 for the left-hand digit and 8 for the right-hand digit. For 2 × 3 we have 0 for the left-hand digit and 6 for the right-hand digit.

Except for simply adding the multiplicand to itself the number of times indicated by the multiplier, a simple but time-consuming process, the simplest method for decimal multiplication involves loading the rightmost digit of the multiplier into a counter that counts downward and then adding the multiplicand to itself and simultaneously indexing the counter until the counter reaches 0. The partial product thus formed may be shifted right one decimal digit, the next multiplier digit loaded into the counter, and the process repeated until all the multiplier digits have been used. This is a relatively slow but straightforward technique.

The process may be speeded up by forming products using the multiplicand and the rightmost digit of the multiplier as in the previous scheme, except by actually forming the left-hand and right-hand partial products obtained when a digit is multiplied by a number and then summing them. For instance, 6 × 7164 would yield 2664 for the right-hand product digits and 4032 for the left-hand product digits. The sum would be

$$\begin{array}{r} 2664 \\ +\ 4032 \\ \hline 42984 \end{array}$$

Decimal-machine multiplication is, in general, a complicated process if speed is desired, and there are almost as many techniques for multiplying BCD numbers as there are types of machines.[11] IC packages are produced that contain a gate network having two BCD characters as inputs which produce the two-digit output required. The Questions and Bibliography explore this in more detail.

DIVISION

***5.18[12]** The operation of division is the most difficult and time-consuming that the ALU of most general-purpose computers performs. Although division may appear no more difficult than multiplication, several problems in connection with the division process introduce time-consuming extra steps.

Division, using pencil and paper, is a trial-and-error process. For instance, if we are to divide 77 into 4610, we first notice that 77 will not "go" into 46; so we attempt to divide 77 into 461. We may guess that it will go six times; however,

[11] Several systems use table look-up techniques for forming products, where the product of each pair of digits is stored in the memory.

[12] Sections with asterisks can be omitted on a first reading with no loss in continuity.

$$\begin{array}{r} 6 \\ 77\overline{)4610} \\ \underline{462} \\ -1 \end{array}$$

Therefore we have guessed too high and must reduce the first digit of the quotient, which we will develop, to 5.

The same problem confronts the computer when it attempts to divide in this manner. It must "try" a subtraction each step of the process and then see whether the remainder is negative. Consider the division of 1111 by 11:

$$\begin{array}{r} 101 \\ 11\overline{)1111} \\ \underline{11} \\ 0011 \\ \underline{11} \\ 00 \end{array}$$

It is easy to determine visually at any step of the process whether the quotient is to be a 1 or a 0, but the computer cannot determine this without making a trial subtraction each time. After a trial quotient has been tried and the divisor subtracted, if the result is negative, either the current dividend must be "restored" or some other technique for dividing used.

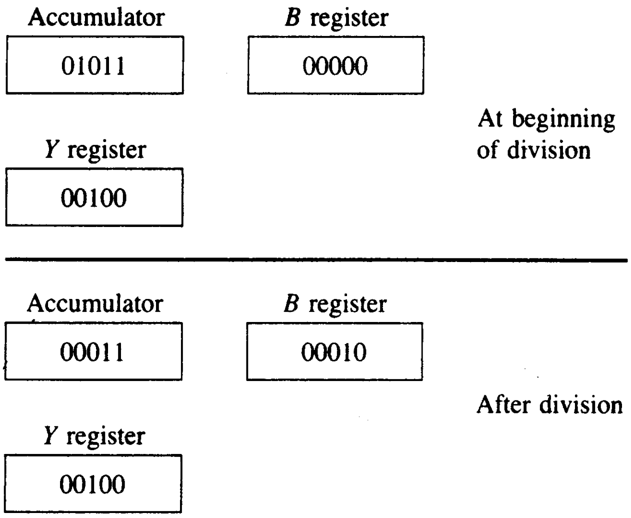
There are several points to be noted concerning binary fixed-point integer-value division. The division is generally performed with two signed binary integers of the same fixed length. The result, or quotient, is stored as a number, with as many digits as the divisor or dividend, and the remainder is also stored as a number of the same length.[13]

Using the registers shown in Fig. 5.18, we show how to divide a number stored in the accumulator by a number in the *Y* register. Then the quotient is stored in the *B* register and the remainder in the accumulator. This is the most common division format.

Assume the *B* and *Y* registers in Fig. 5.18 are 5 bits in length (4 bits plus a sign bit) and the accumulator is also 5 bits in length. Before we start the procedure, the dividend is read into the accumulator, and the divisor into the *Y* register. After the division, the quotient is stored in the *B* register, and the remainder is in the accumulator. Both divisor and dividend are to be positive.

The following shows an example. The accumulator (dividend) originally contains 11 (decimal) and the *Y* register (divisor) contains 4. The desired result then gives the quotient 2 in the *B* register and the remainder 3 in the accumulator.

[13] If we divide one integer into another, both the quotient and remainder will be integers. The rule is as follows: If a is the dividend, y the divisor, b the quotient, and r the remainder, then $a = y \times b + r$.



There are two general techniques for division for binary machines: the *restoring* and the *nonrestoring* techniques. Our first example illustrates the restoring technique.

Just as in multiplication, the restoring technique for division requires that a *basic step* be performed repeatedly (in this case as many times as there are significant bits in the subtrahend).

BASIC STEP

A "trial division" is made by subtracting the *Y* register from the accumulator. After the subtraction, one of the following is executed:

1 If the result is negative, the divisor will not "go," so a 0 is placed in the rightmost bit of the *B* register, and the dividend (accumulator) is restored by adding the divisor to the result of the subtraction. The combined *B* register and accumulator are then shifted left.

2 If the result of a subtraction is positive or zero, there is no need to restore the partial dividend in the accumulator for the trial division has succeeded. The accumulator and *B* register are both shifted left, and then a 1 is placed in the rightmost bit of the *B* register.

The computer determines whether the result of a trial division is positive or negative by examining the sign bit of the accumulator after each subtraction.

TABLE 5.2

B REGISTER	ACCUMULATOR	*Y* REGISTER	REMARKS
00000	00110	00011	We divide 6 by 3.
00000	00110	00110	*Y* register is shifted left once, aligning 1s in accumulator and *Y* register. The basic step must be performed two times.
00000	00000	00110	*Y* register has been subtracted from accumulator. The result is 0, so *B* register and accumulator are shifted left, and a 1 is placed in the rightmost bit of *B* register.
00001	00000	00110	
00001	11010	00110	*Y* register is subtracted from accumulator.
00010	00000	00110	*Y* register is added to accumulator, and *B* register and accumulator are shifted left 1. A 0 is placed in *B* register's last bit.
00010	00000	00110	Accumulator must now be shifted right two times, but it is 0 so no change results. The quotient in *B* register is 2, and the remainder in accumulator is 0.

To demonstrate the entire procedure, first it is necessary to explain how to initiate the division and how to start and stop performing the basic steps. Unfortunately these are complicated procedures, just as determining the position of the decimal point and how to start and stop the division is complicated for ordinary division.

1 As described above, if the divisor is larger than the dividend, then the quotient should be 0, and the remainder is the value of the dividend. (For instance, if we attempt to divide 7 by 17, the quotient is 0, and the remainder is 7.) To test this, the dividend in *Y* can be subtracted from the accumulator. If the result is negative, all that remains is to restore the accumulator by adding the *Y* register to the accumulator. The *B* register now has value 0 which is right for the quotient, and the accumulator has the original value which is the remainder.

2 After the above test is made, it is necessary both to align the leftmost 1 bit in the divisor with the leftmost 1 bit in the dividend by shifting the divisor left and then recording the number of shifts required to make this alignment. If the number of shifts is M, then the basic step must be performed $M + 1$ times.[14]

3 The basic step is now performed the necessary $M + 1$ times.

[14] This can be accomplished by making *Y* a shift register and providing a counter to count the shifts until the first 1 bit of *Y* is aligned with the 1 bit of the accumulator. Both the accumulator and *Y* could be shifted left until there is a 1 bit in their first position, but the remainder will have to be adjusted by moving it right in the accumulator.

TABLE 5.3

B REGISTER	ACCUMULATOR	*Y* REGISTER	REMARKS
00000	01101	00011	We divide 13 by 3.
00000	01101	00110	Shift *Y* register left.
00000	01101	01100	Shift *Y* register left. Leftmost 1 bits in accumulator and *Y* register are aligned. Basic step will be performed three times.
00000	00001	01100	*Y* register has been subtracted from accumulator. Result is positive.
00001	00010	01100	*B* register and accumulator are shifted left, and 1 is placed in *B* register.
00001	10110	01100	*Y* register is subtracted from accumulator. Result is negative.
00010	00100	01100	*Y* register is added to accumulator. Then both are shifted left, and a 0 is placed in *B* register.
00010	11000	01100	*Y* register is subtracted from accumulator. Result is negative.
00100	01000	01100	*Y* register is added to accumulator. Accumulator and *B* register are shifted left. A 0 is placed in *B* register's bit.
00100	00001	01100	Accumulator has been shifted right three times. The quotient is 4, and the remainder is 1.

4 Finally, to adjust the remainder, the accumulator must be shifted right $M + 1$ times after the last basic step is performed. Examples are shown in Tables 5.2 and 5.3. Step 1, testing for a zero quotient, is not shown in the two examples.

Figure 5.19 shows a flowchart of the algorithm. Flowcharts are often used to represent algorithms. A more detailed flowchart would separate some of the steps, such as "shift the accumulator right $M + 1$ times," into single shifts performed in a loop which is controlled by a counter. Often, when'algorithms are reasonably complicated, as this algorithm is, it is convenient to draw a flowchart of the algorithm before attempting to implement the control circuitry.

During division, the sign bits are handled in much the same way as during multiplication. The first step is to convert both the divisor and the dividend to positive magnitude form. The value of the sign bit for the quotient must be stored while the division is taking place. The rule is that if the signs of the dividend and divisor are both either 0s or 1s, the quotient will be positive. If either but not both of their signs are a 1, the quotient will be negative. The relationship of the sign bit of the quotient to the sign bit of the divisor and dividend is, therefore, the quarter-adder, or exclusive OR, relationship, that is, $S = X\overline{Y} + \overline{X}Y$. The value for the correct sign of the quotient may be read into a flip-flop while the division is taking place, and this value may then be placed in the sign bit of the register containing the quotient after the division of magnitudes has been completed.

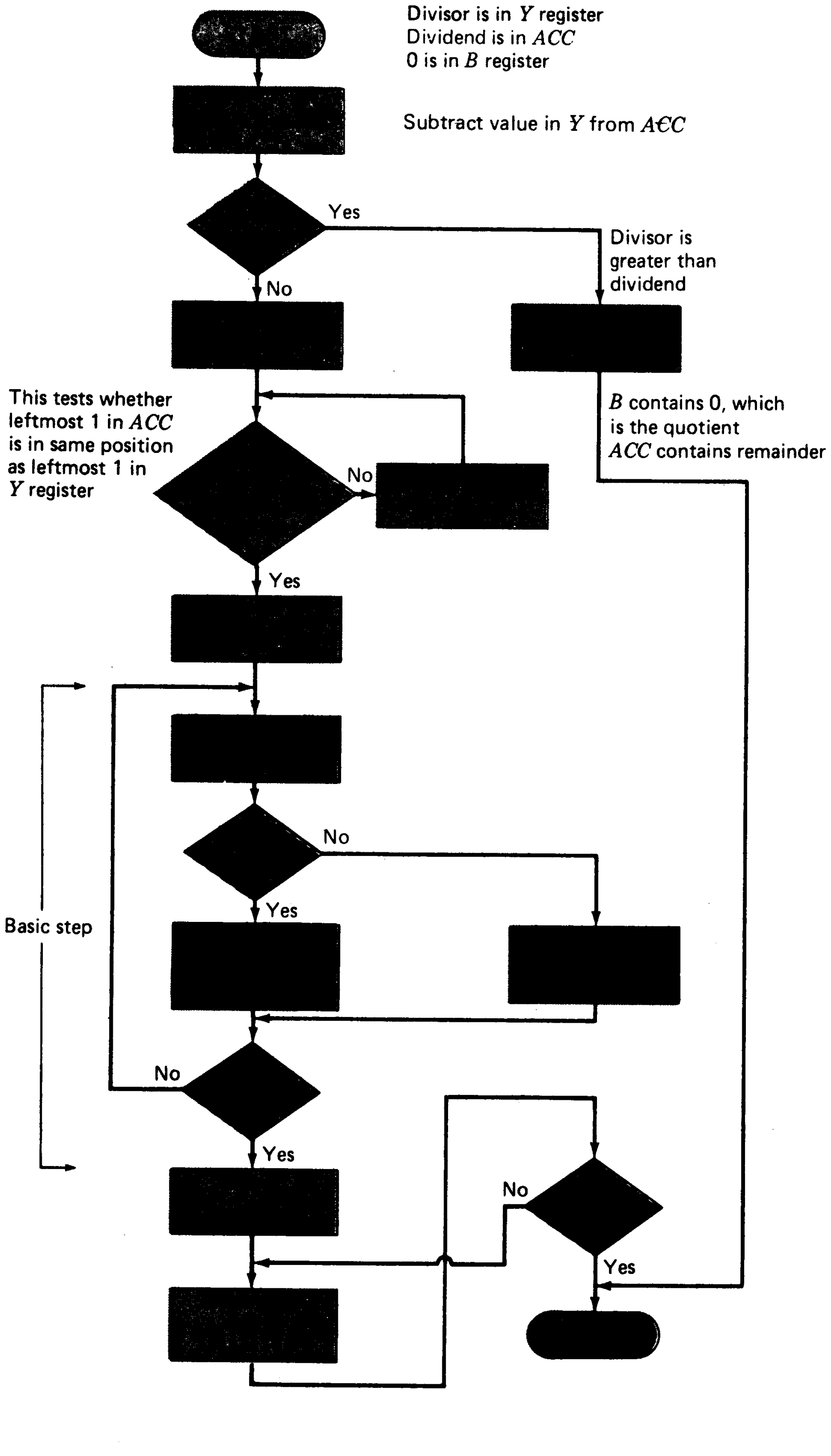

Divisor is in Y register
Dividend is in ACC
0 is in B register
Subtract value in Y from $A€C$
Yes
Divisor is greater than dividend
No
This tests whether leftmost 1 in ACC is in same position as leftmost 1 in Y register
No
B contains 0, which is the quotient
ACC contains remainder
Yes
No
Yes
Basic step
No
Yes
No
Yes

FIGURE 5.19

Flowchart of division algorithm.

There are several techniques for nonrestoring division. One widely used algorithm employs a procedure in which the divisor is alternately subtracted and added. Another uses a technique in which the divisor is compared to the dividend at each trial division. This material is covered in detail in the Bibliography.

LOGICAL OPERATIONS

5.19 In addition to the arithmetic operations, many logical operations are performed by ALUs. Three logical operations are described here: logical multiplication, logical addition, and *sum modulo 2* addition (the exclusive OR operation). Each of these will be operations between registers, where the operation specified will be performed on each of the corresponding digits in the two registers. The result will be stored in one of the registers.

The first operation, logical multiplication, is often referred to as an *extract, masking,* or AND *operation*. The rules for logical multiplication are defined in the chapter on logical algebra. The rules are 0·0 = 0, 0·1 = 0, 1·0 = 0, and 1·1 = 1. Suppose that the contents of the accumulator register are "logically multiplied" by another register. Let each register be five binary digits in length. If the accumulator contains 01101 and the other register contains 00111, the contents of the accumulator after the operation will be 00101.

The masking, or extracting, operation is useful in "packaging" computer words. To save space in memory and keep associated data together, several pieces of information may be stored in the same word. For instance, a word may contain an item number, wholesale price, and retail price, packaged as follows:

S	1→6	7→15	16→24
	item number	wholesale price	retail price

To extract the retail price, the programmer will simply logically multiply the word above by a word containing 0s in the sign digit through digit 15 and 1s in positions 16 through 24. After the operation, only the retail price will remain in the word.

The logical addition operation and the sum modulo 2 operation are also provided in most computers. The rules for these operations are as follows:

LOGICAL ADDITION	MODULO 2 ADDITION
$0 + 0 = 0$	$0 \oplus 0 = 0$
$0 + 1 = 1$	$0 \oplus 1 = 1$
$1 + 0 = 1$	$1 \oplus 0 = 1$
$1 + 1 = 1$	$1 \oplus 1 = 0$

Figure 5.20 shows how a single accumulator flip-flop and *B* flip-flop can be gated so that all three of these logical operations can be performed. The circuit in Fig. 5.20 would be repeated for each stage of the accumulator register.

There are three control signals: LOGICAL MULTIPLY, LOGICAL ADD, and MOD 2 ADD. If one of these is a 1, when a clock pulse arrives, this operation

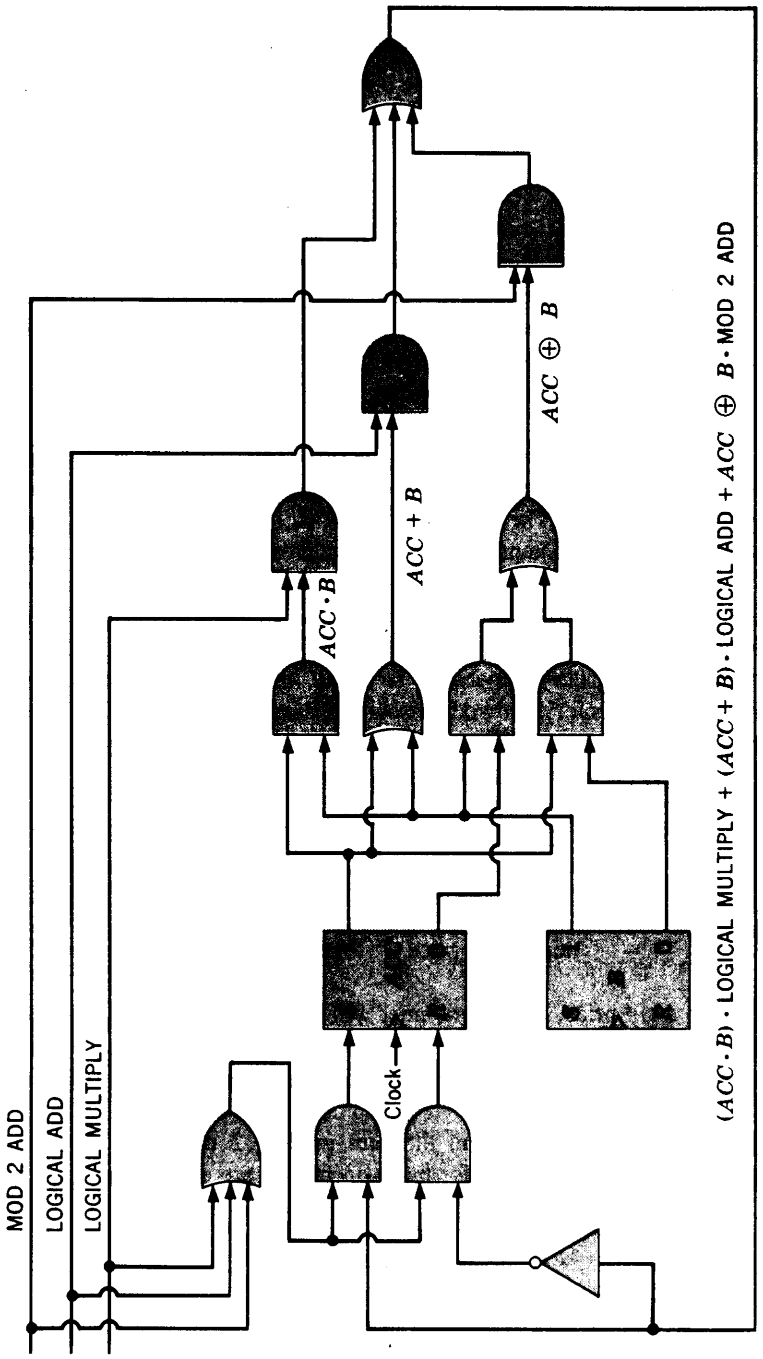

MOD 2 ADD
LOGICAL ADD
LOGICAL MULTIPLY
Clock
ACC·B
ACC + B
ACC ⊕ B
(ACC·B)·LOGICAL MULTIPLY + (ACC + B)·LOGICAL ADD + ACC ⊕ B·MOD 2 ADD

FIGURE 5.20

Circuit for gating logic operations into accumulator flip-flop.

is performed and the result placed in the ACC (accumulator) flip-flop. If none of the control signals is a 1, nothing happens, and the ACC remains as it is.

The actual values desired are formed by three sets of gates; that is, ACC·*B*, ACC + *B*, and ACC ⊕ *B* are all formed first. Each is then AND-gated with the appropriate control signal. Finally the three control signals are ORed, and this signal is used to gate the appropriate value into the ACC flip-flop when one of the control signals is a 1.

Figure 5.20 shows how a choice of several different function values can be gated into a single flip-flop using control signals. We could include an ADD signal and a SHIFT RIGHT and a SHIFT LEFT by simply adding more gates.

Figure 5.21 shows an example of the logic circuitry used to form sections of an ALU. All the gates are contained in a single chip (package) with 24 pins. There is a 7-ns maximum delay through the package.

This chip is called a *4-bit arithmetic-logic unit* and can add, subtract, AND, OR, etc., two 4-bit register sections. Two chips could be used for the logic in an 8-bit accumulator, four chips would form a 16-bit accumulator, etc.

The function performed by this chip is controlled by the mode input M and four function select inputs S_0, S_1, S_2, and S_3. When the mode input M is low (a 0), the 74S181 performs such arithmetic operations as ADD or SUBTRACT. When the mode input M is high (a 1), the ALU does logic operations on the A and B inputs "a bit at a time." (Notice in Fig. 5.21 that the carry-generating gates are disabled by $M = 1$.) For instance, if M is a 0, S_1 and S_2 are also 0s, and S_0 and S_3 are 1s, then the 74S181 performs arithmetic addition. If M is a 1, S_0 and S_3 are 1s, and S_1 and S_2 are 0s, the 74S181 chip exclusive-ORs (mod 2 adds) A and B. (It forms $A_0 \oplus B_0$, $A_1 \oplus B_1$, $A_2 \oplus B_2$, and $A_3 \oplus B_3$.)

The table in Fig. 5.21 further describes the operation of this chip. Questions at the end of the chapter develop some operational characteristics of this 4-bit ALU section.

MULTIPLEXERS

5.20 The function of a *multiplexer* is to select from several inputs a single input. Control lines are used to make this selection.

Figure 5.22 shows an eight-input multiplexer on a single IC chip. The eight inputs are labeled $I_0, I_1, \ldots, I_7$. There are three control wires, S_2, S_1, and S_0. These three control lines can take eight different values (from 000 to 111), and for each value a different input is selected. The value of the input selected appears on Z. An examination of this multiplexer shows that if $S_2S_1S_0$ are all 0s, then input I_0 is selected. If $S_2S_1S_0$ are 001, then I_1 is selected; if $S_2S_1S_0$ are 010, then I_2 is selected; etc.

For example, if $S_2S_1S_0 = 010$, then the output Z will be 0 if I_2 is a 0 and a 1 if I_2 is a 1. In this case, the input values on I_0, I_1, I_3, I_4, I_5, I_6 and I_7 will not affect the output value on Z. The E input enables the multiplexer.

Multiplexers are useful in many ways. Suppose we are to select as inputs to a gate network a single register from four flip-flop registers with two flip-flops in each register. Figure 5.23 shows a dual four-input multiplexer which will accomplish this. The two multiplexers each have four inputs, and the inputs selected from each are in the same respective position. There are two control inputs, S_1 and S_0. If S_1 and S_0 are both 0s, then A_0 and B_0 are selected and placed on the A and B outputs; if S_1 is a 0 and S_0 is a 1, then the values of A_1 and B_1 are placed on the A and B outputs; etc.

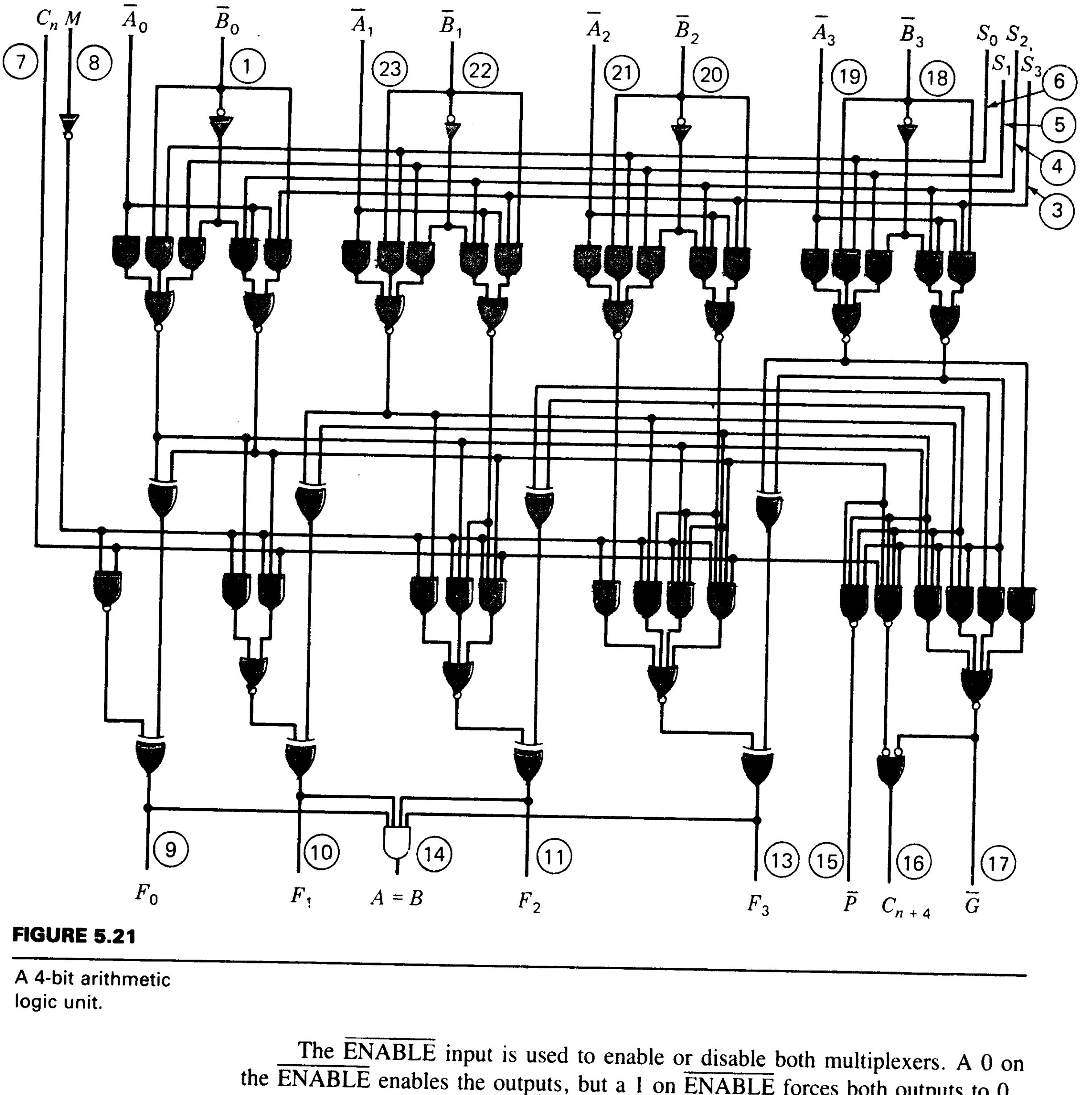


FIGURE 5.21

A 4-bit arithmetic logic unit.

The $\overline{\text{ENABLE}}$ input is used to enable or disable both multiplexers. A 0 on the $\overline{\text{ENABLE}}$ enables the outputs, but a 1 on $\overline{\text{ENABLE}}$ forces both outputs to 0.

Figure 5.24 shows four flip-flop registers W, X, Y, and Z, each with two flip-flops. The control lines S_0 and S_1 select from each of the four sets of inputs a single two-flip-flop register whose outputs are then placed on the output lines. This shows how multiplexers can be used to select a single register from a set of registers.

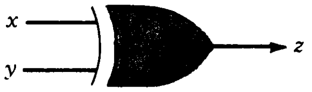
If each register contained more than two flip-flops, then another dual four-input multiplexer would be needed for each additional two flip-flops.

Multiplexers are useful in many ways, and Figs. 5.22 and 5.23 should be examined carefully.

MODE SELECT INPUTS				ACTIVE LOW INPUTS & OUTPUTS	
S_3	S_2	S_1	S_0	LOGIC $(M = H)$	ARITHMETIC $(M = L)$ $(C_n = L)$
L	L	L	L	$\overline{A}$	$A - 1$
L	L	L	L	$\overline{AB}$	$AB - 1$
L	L	H	L	$\overline{A} + B$	$A\overline{B} - 1$
L	L	H	H	Logical 1	-1
L	H	L	L	$\overline{A + B}$	$A \mp (A + \overline{B})$
L	H	L	H	$\overline{B}$	$AB \mp (A + \overline{B})$
L	H	H	L	$\overline{A \oplus B}$	$A - B - 1$
L	H	H	H	$A + \overline{B}$	$A + \overline{B}$
H	L	L	L	$A\overline{B}$	$A \mp (A + B)$
H	L	L	H	$A \oplus B$	$A \mp B$
H	L	H	L	B	$AB \mp (A + B)$
H	L	H	H	$A + B$	$A + B$
H	H	L	L	Logical 0	$A \mp A$
H	H	L	H	$A\overline{B}$	$AB \mp A$
H	H	H	L	AB	$AB \mp A$
H	H	H	H	A	A

*L = 0; H = 1.

Note:



is the symbol for a mod 2 adder (exclusive OR gate)
$z = x \oplus y$

$\mp$ is the sign for arithmetic addition

FIGURE 5.21 (*Cont.*)

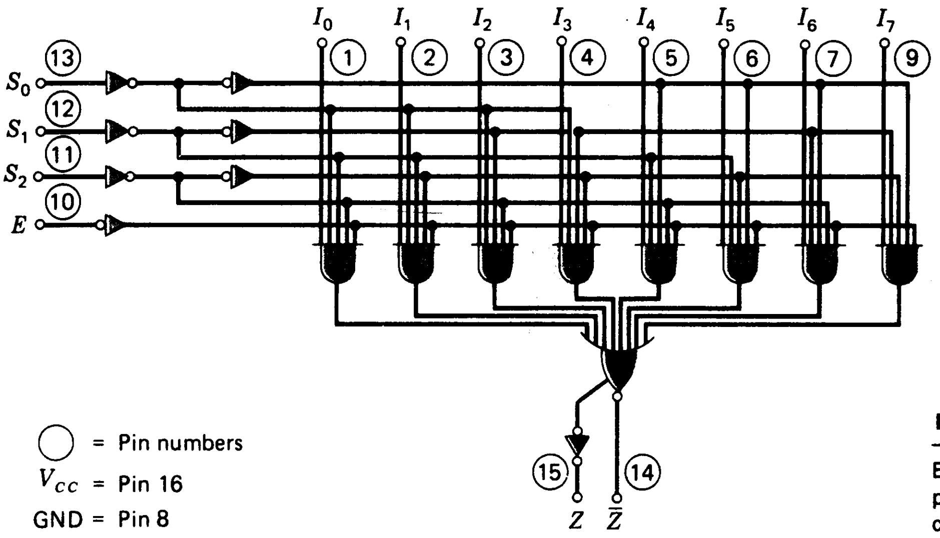


FIGURE 5.22

Eight-input multiplexer in a single IC container.

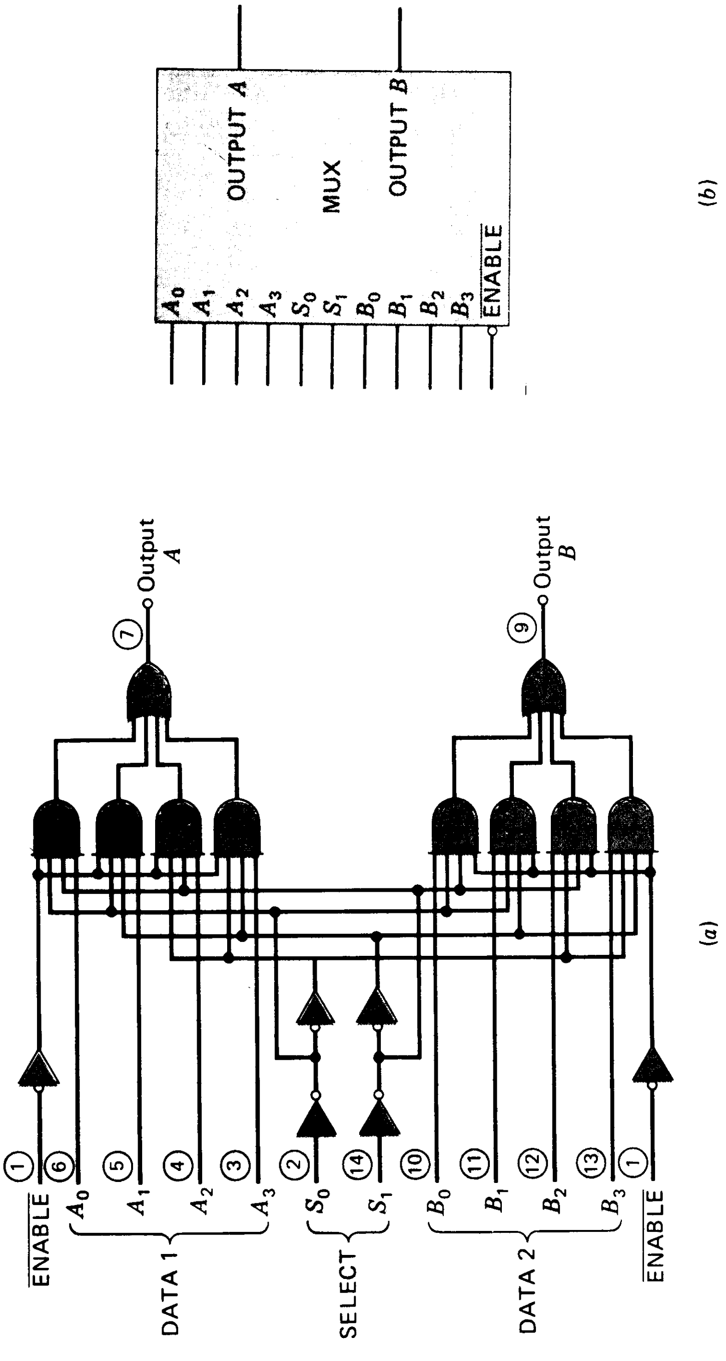


FIGURE 5.23

Two multiplexers in a single IC container (SN54153 and SN74153). (*a*) Block diagram showing gates. (*b*) Block diagram symbol.

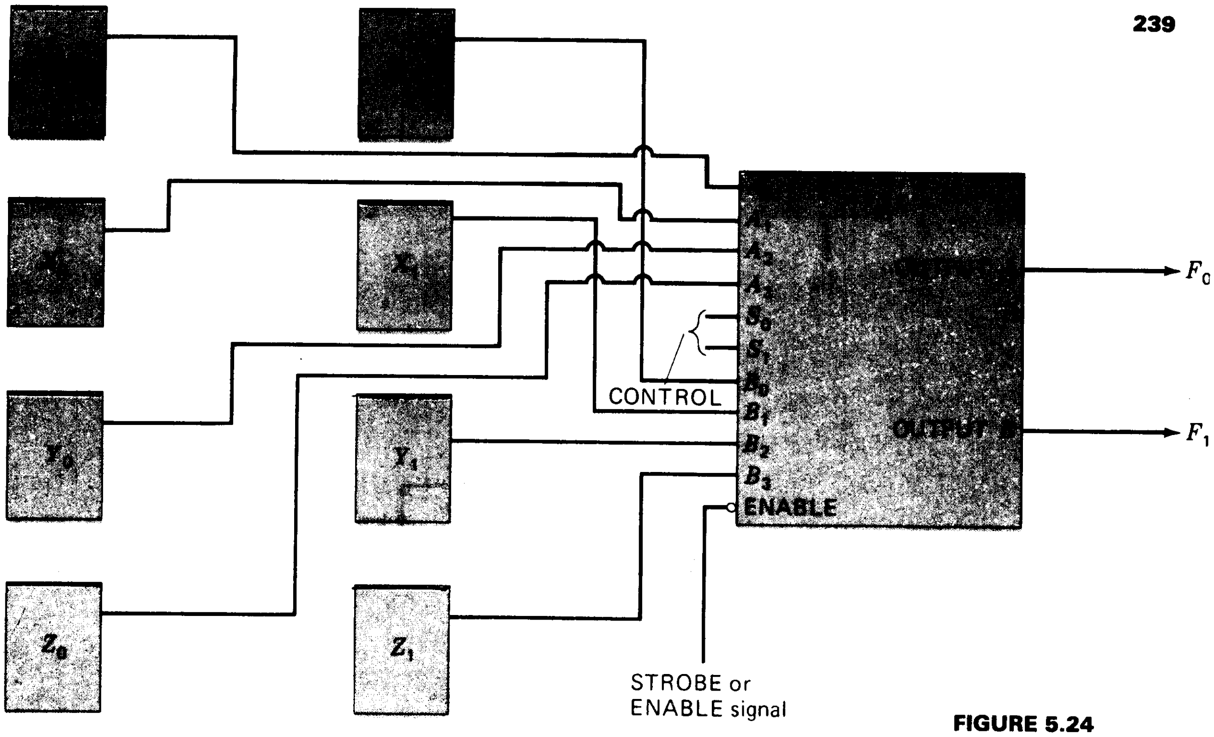


FIGURE 5.24

Using a dual four-input multiplexer IC to select from two four-input flip-flop registers.

HIGH-SPEED ARITHMETIC—SPEEDING UP ADDITION

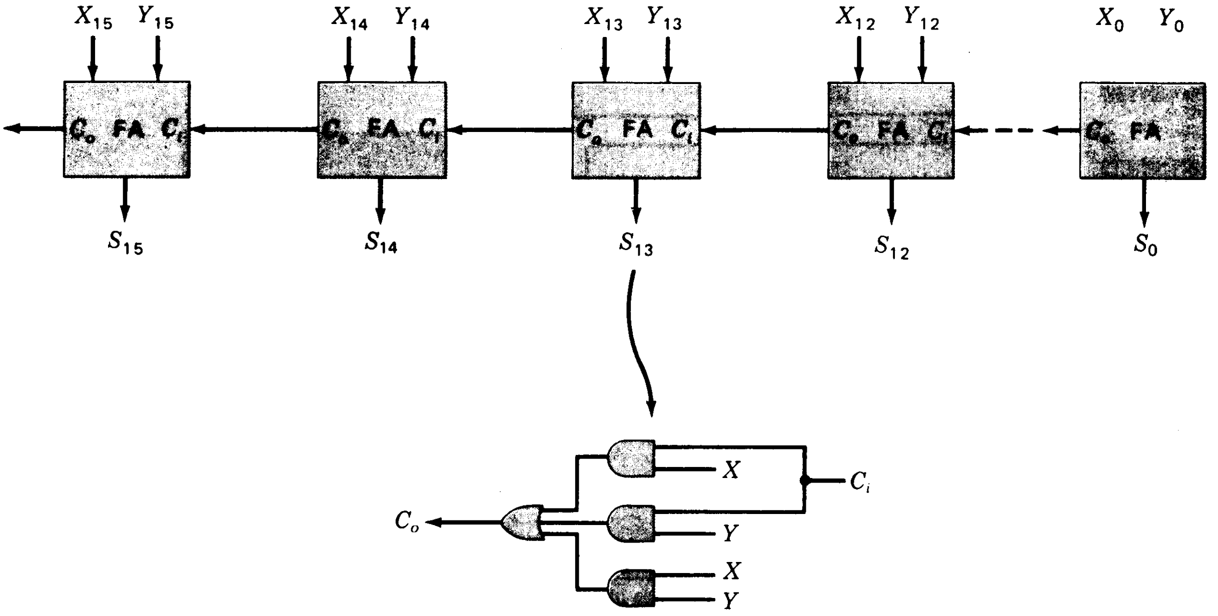
***5.21** Since additions and subtractions are often performed in computers, it is desirable to perform them quickly. In this section we describe how the time required may be shortened. This also will speed up multiplication and division since in most cases these involve a number of additions or subtractions.

Figure 5.25 shows a set of full-adders as they might be interconnected to form a 16-bit full-adder for two registers of 16 flip-flops each (for two 16-bit numbers). Note that a carry arising in the rightmost adder (if X_0 and Y_0 are both 1s) will propagate all the way through the leftmost adder (for X_{15} and Y_{15}) if there is a 1 input at each adder in the chain.

Each gate in a network delays a signal by some time period. Thus if a set of new inputs is placed on the inputs to the adder configuration, it is necessary to wait until the signals have passed through all gates before the outputs, in this case S_{15} through S_0, can be safely used. If each gate has a delay of D ns, then for Fig. 5.25 it is necessary to wait for $32 \times D$ ns from the time the inputs are changed before we can be sure the value of S_{15} is correct.

This is called the carry propagation delay. This delay can be considerable for long registers if the configuration in Fig. 5.25 is used without modification. Fortunately, there are several ways to shorten the carry propagation delay, as we show.

Figure 5.26 shows an IC chip layout which contains gates to add two 4-bit inputs plus a carry to the group. Note here that the C_o, or carry output, has a



CARRY logic in each stage looks like this

FIGURE 5.25

Chain of full-adders.

maximum delay path of three gates for any input. That is, the maximum delay path from any input to the output C_o is a three-gate delay.

Figure 5.27 shows how four of these IC containers can be interconnected to form an adder which will handle 16-bit inputs. The maximum delay through this network is shorter than that for the layout in Fig. 5.25 because of the shorter carry delays. The maximum path length in number of gates for the C_i input to the leftmost four-adder package is nine gates, or 9 × D ns if a gate delay is D ns. The delay through the final package, to S_{15}, for example, is four gate delays, however, because a carry input to C_i must pass through an AND gate, a NOR gate, and exclusive OR gates to reach S_{15}; and exclusive OR gates require two delays. (Exclusive OR gates are often made from a two-level network of conventional gates.)

The reduction of adder carry propagation delay using the adder in Fig. 5.26 is due to the development of the C_o output directly from the eight inputs A_1, B_1, . . . , A_4, B_4, and C_i. For example, if we wish to put a 3-bit adder in a single container, with inputs A_1, B_1, A_2, B_2, A_3, B_3, and C_i, then the C_o output can be written[15] as follows:

$$C_o = C_i(A_3 + B_3)(A_2 + B_2)(A_1 + B_1) \\ + A_3B_3(A_2 + B_2)(A_1 + B_1) + A_2B_2(A_1 + B_1) + A_1B_1$$

[15] This is the expression for a "carry look-ahead" or "carry bridging" net. Placing carry look-aheads every few adders will speed up adder operation.

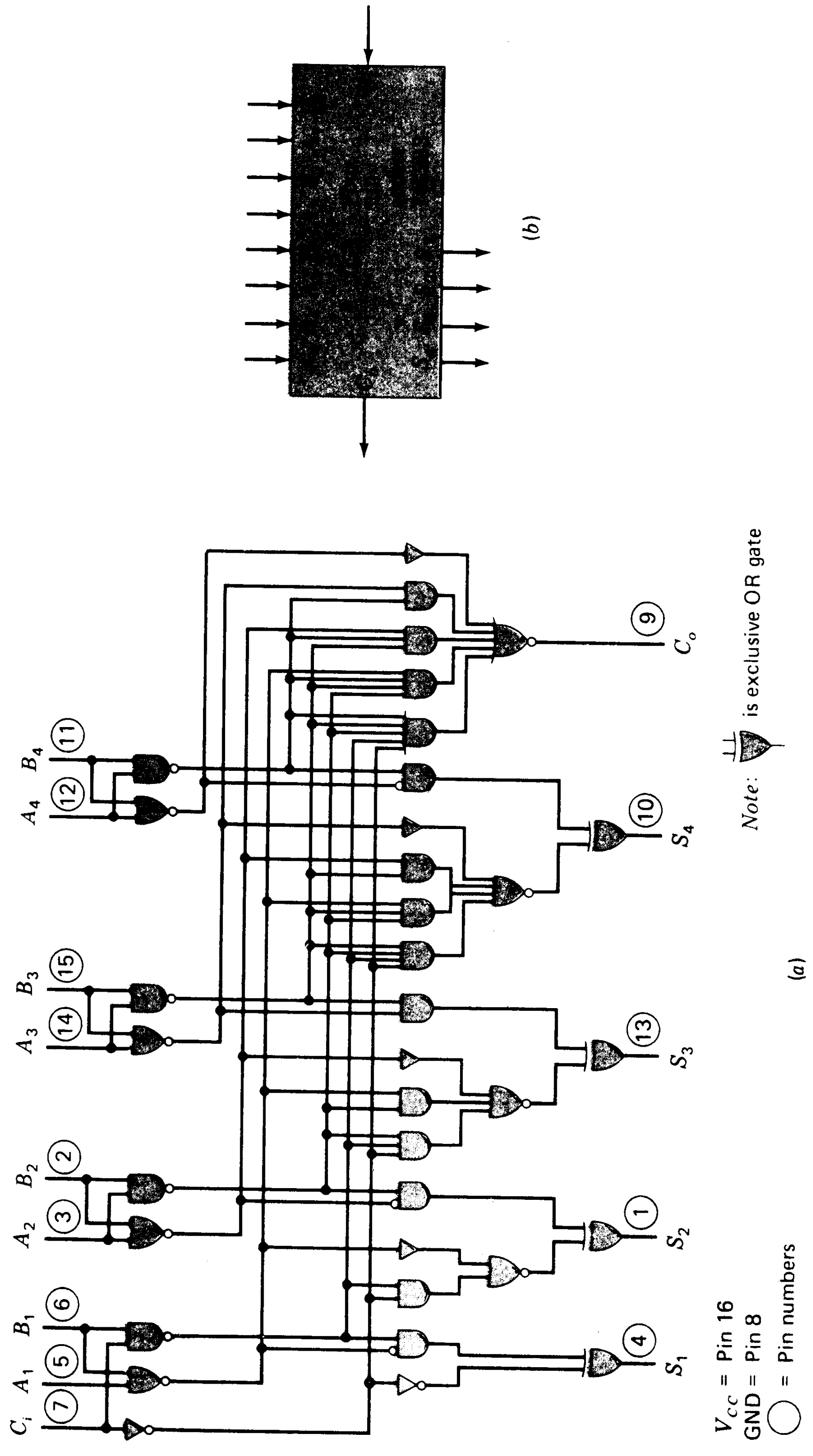


FIGURE 5.26

Full adders with carry bridge. (*a*) A 74S283 chip with four full-adders. (*b*) Block diagram for (*a*).

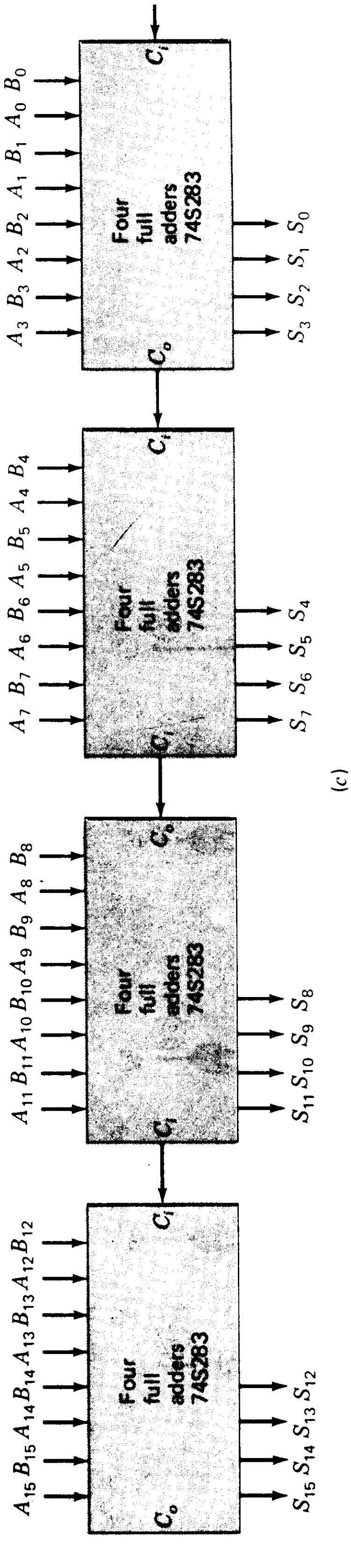
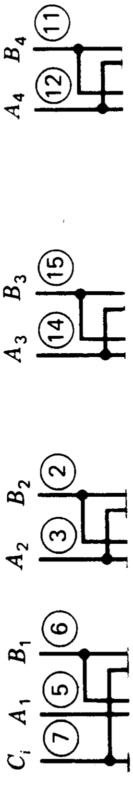

A_3 B_3 A_2 B_2 A_1 B_1 A_0 B_0
Four full adders 74S283
C_i
C_o
S_0 S_1 S_2 S_3
A_7 B_7 A_6 B_6 A_5 B_5 A_4 B_4
S_4 S_5 S_6 S_7
A_{11} B_{11} A_{10} B_{10} A_9 B_9 A_8 B_8
S_8 S_9 S_{10} S_{11}
A_{15} B_{15} A_{14} B_{14} A_{13} B_{13} A_{12} B_{12}
S_{12} S_{13} S_{14} S_{15}

FIGURE 5.27

A 16-bit adder made from IC chips.

(c)



Turning this expression directly into a gate network results in a three-level network and thus three-gate delays, but the expression can be "multiplied out" and a two-level net will result.

The amount of reduction of delay in adders depends on the complexity (and therefore cost) of the gating network used. As an example, Fig. 5.21 has a reasonably fast carry, but this chip also develops a P and G output which can be used with another chip, shown in the Questions, to further speed up adder operation.

HIGH-SPEED ARITHMETIC—PARALLEL MULTIPLIERS

***5.22** The multiplication technique described earlier is called the *add-and-shift algorithm*. This technique is used in most smaller computers because it is direct to implement. Also it is often used in programs which implement multiplication because many smaller computers (microcomputers, in particular) have no multiplication instruction.

In larger computers and in signal-processing computers, there is a need for high-speed multiplication. To achieve this, arrays of gates are used which multiply several binary digits at the same time. These arrays are of various sizes and range from moderately inexpensive to quite expensive. Their use is based on economic considerations and the requirements for the system.

We illustrate how two binary numbers can be multiplied in a gating network. Suppose the numbers are a_1a_0 and b_1b_0, two binary 2-bit numbers. For example, if $a_1 = 1$ and $a_0 = 0$, then $a_1a_0 = 10$, which is 2 in decimal. Similarly, if $b_1 = 1$ and $b_0 = 1$, then $b_1b_0 = 11$, which is 3 in decimal.

If these numbers are multiplied using our familiar technique, this array is formed:

$$\begin{array}{rrrr}
 & & b_1 & b_0 \\
 & \times & a_1 & a_0 \\
\hline
 & & a_0b_1 & a_0b_0 \\
 & a_1b_1 & a_1b_0 & \\
\hline
p_3 & p_2 & p_1 & p_0
\end{array}$$

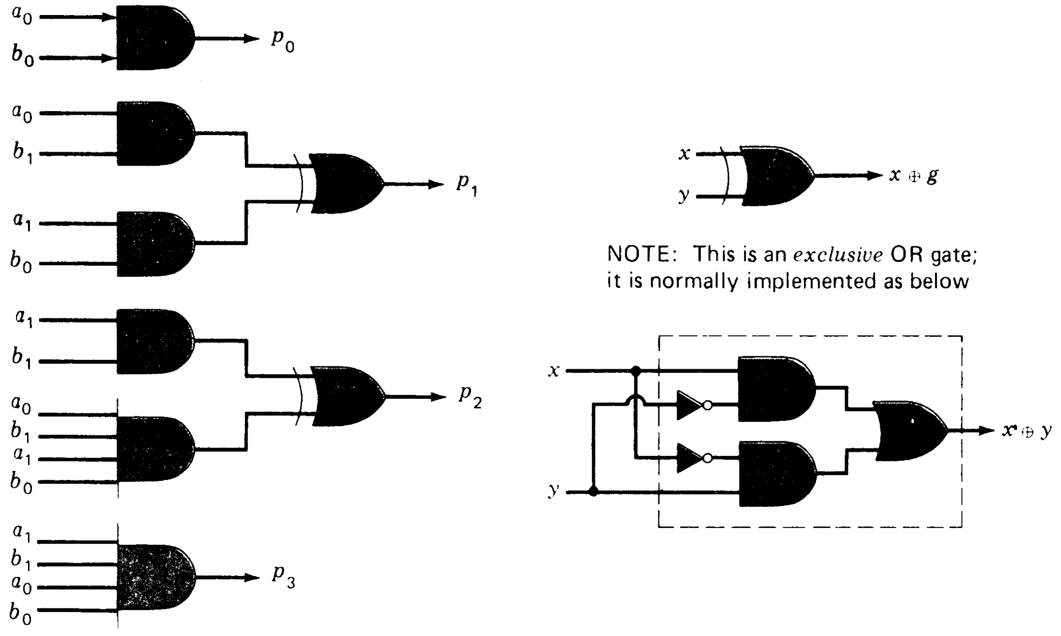
where

$$\begin{aligned}
p_0 &= a_0b_0 \\
p_1 &= a_0b_1 \oplus a_1b_0 \\
p_2 &= a_1b_1 \oplus a_0b_1a_1b_0 \\
p_3 &= a_1b_1a_0a_1b_1a_0b_0
\end{aligned}$$

Here

$$\begin{array}{ll}
0 \oplus 0 = 0 & 1 \oplus 0 = 1 \\
0 \oplus 1 = 1 & 1 \oplus 1 = 1
\end{array}$$

Figure 5.28 shows the boolean algebra expression for the product bits p_3, p_2, p_1, and p_0 realized in gate network form. If inputs for a_1a_0 and b_1b_0 are input to



NOTE: This is an *exclusive* OR gate; it is normally implemented as below

FIGURE 5.28

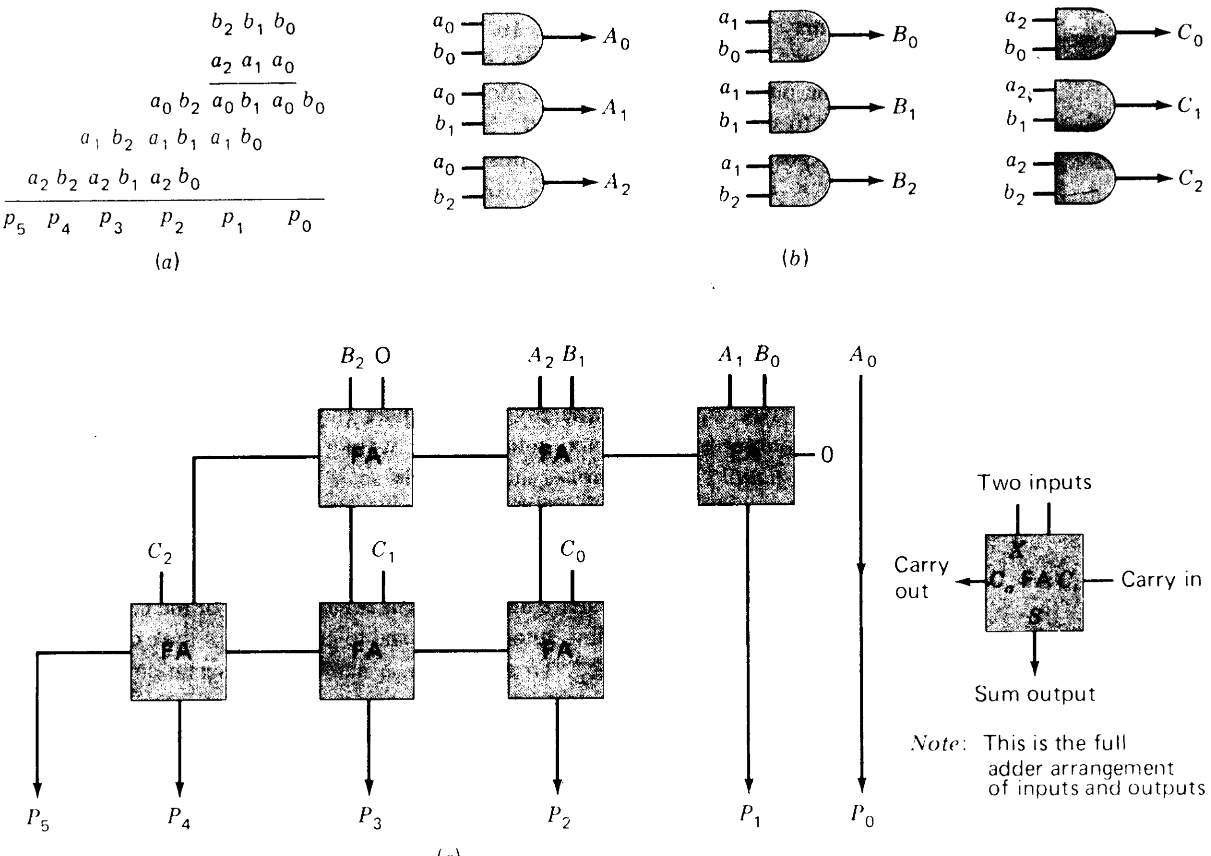
A two-digit parallel binary multiplier.

the network, then $p_3p_2p_1p_0$ will give the product in binary integer form. The net in Fig. 5.28 is called a *parallel multiplier.*

The above technique for deriving boolean algebra expressions for the product bits can be used for multiplications involving more digits. Unfortunately, for 8-bit or even 16-bit multipliers and multiplicands, the expressions become very large and costly to implement. However, this technique works well for small numbers.

To make parallel multipliers for larger numbers of inputs and outputs, often an array of full-adders is used. Consider the multiplication of two 3-bit numbers shown in Fig. 5.29(*a*). The partial products are written with the product bits $p_5p_4p_3p_2p_1p_0$ immediately below. In Fig. 5.29(*b*) a set of nine AND gates is used to produce all the a_ib_j terms in the multiplier in Fig. 5.29(*a*). In Fig. 5.29(*c*) an arrangement of full-adders is shown which have as inputs the outputs from the AND gates in Fig. 5.29(*b*) and which will implement the multiplication shown in Fig. 5.29(*a*). It is instructive to see how the multiplication is performed by the full-adders. The operation of this circuit should be carefully examined.

The maximum length of carry path for Fig. 5.29(*b*) is along the top row to the p_5 output. If the full-adders have two-gate delays for each carry, then Fig. 5.29(*c*) has a worst-case delay of eight-gate delays and another delay arises from the AND gates in Fig. 5.29(*b*). As the number of bits in the numbers being multiplied increases, so does the size of the array of full-adders and so does the delay through the array. There are several ways to rearrange the adders slightly and to add more gates to reduce this delay, which are covered in the Bibliography. [The paper "High-Speed Monolithic Multipliers for Real-Time Digital Signal Processing" by S. Waser in the October 1979 issue of *Computer* is very instructive. The array in Fig. 5.29(*c*) is fundamental, however.]



Note: This is the full adder arrangement of inputs and outputs

FIGURE 5.29

A parallel multiplier. (a) Multiplication of 3-bit numbers. (b) Forming prevalent terms. (c) Parallel multiplier made of full-adders.

Parallel multiplier arrays are packaged in IC containers by several manufacturers. The largest array in a single IC container now multiplies two 16-bit numbers. The delay through this package is about 100 ns. Eight-bit parallel multipliers are common. Several of these multipliers can be grouped along with some full-adders to form multipliers for even larger numbers. Parallel multipliers also can be used to shorten multiplication time by using an add-and-shift algorithm and multiplying several bits at each step.

FLOATING-POINT NUMBER SYSTEMS

5.23 Earlier we described number representation systems in which positive and negative integers are stored in binary words. In the representation system used, the binary point is "fixed" in that it lies at the end of each word, and so each value represented is an integer. When computers calculate with binary numbers in this format, the operations are called *fixed-point arithmetic*.

In science it is often necessary to calculate with very large or very small numbers. So scientists have adopted a convenient notation in which a *mantissa* and an *exponent* represent a number. For instance, 4,900,000 may be written as

0.49×10^7, where 0.49 is the mantissa and 7 is the value of the exponent; or 0.00023 may be written as 0.23×10^{-3}. The notation is based on the relation $y = a \times r^p$, where y is the number to be represented, a is the mantissa, r is the base of the number system ($r = 10$ for decimal and $r = 2$ for binary), and p is the power to which the base is raised.

It is possible to calculate with this representation system. To multiply $a \times 10^n$ and $b \times 10^m$, we form $a \times b \times 10^{m+n}$. To divide $a \times 10^m$ by $b \times 10^n$, we form $a/b \times 10^{m-n}$. To add $a \times 10^m$ to $b \times 10^n$, we must first make m equal to n. If $m = n$, then $a \times 10^m + b \times 10^n = a + b \times 10^m$. The process of making m equal to n is called *scaling* the numbers.

Considerable bookkeeping can be involved in scaling numbers, and there can be difficulty in maintaining precision during computations when numbers vary over a very wide range of magnitudes. For computer usage these problems are alleviated by means of two techniques whereby the computer (not the programmer) keeps track of the radix (decimal) point, automatically scaling the numbers. In the first, programmed *floating-point routines* automatically scale the numbers used during the computations while maintaining the precision of the results and keeping track of thc scale factors. These routines are used with small computers having only fixed-point operations. A second technique lies in building what are called *floating-point operations* into the computer's hardware. The logic circuitry of the computer is then used to perform the scaling automatically and to keep track of the exponents when calculations are performed. To effect this, a number representation system called the *floating-point system* is used.

A floating-point number in a computer uses the exponential notation system described, and during calculations the computer keeps track of the exponent as well as the mantissa. A computer number word in a floating-point system may be divided into three pieces: the first is the sign bit, indicating whether the number is negative or positive; the second part contains the exponent for the number to be represented; and the third part is the mantissa.

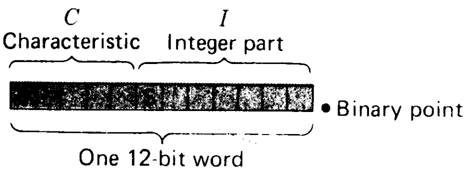
As an example, let us consider a 12-bit word length computer with a floating-point word. Figure 5.30 shows this. It is common practice to call the exponent part of the word the *characteristic* and the mantissa section the *integer part.*

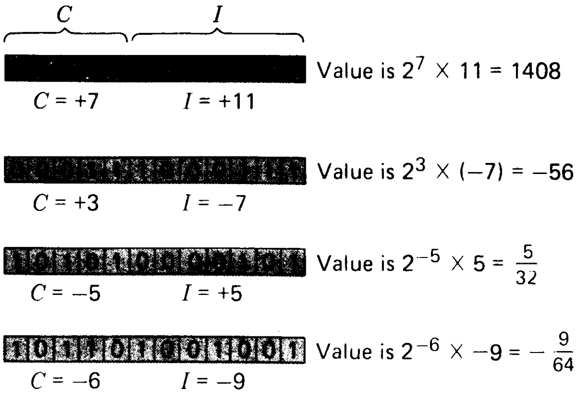
The integer part of the floating-point word shown represents its value in signed-magnitude form (rather than 2s complement, although this has been used). The characteristic is also in signed-magnitude form. The value of the number expressed is $I \times 2^C$, where I is the value of the integer part and C is the value of the characteristic.

Figure 5.31 shows several values of floating-point numbers both in binary form and after they are converted to decimal. Since the characteristic has 5 bits and is in signed-magnitude form, the C in $I \times 2^C$ can have values from -15 to

FIGURE 5.30

A 12-bit floating-point word.




C
I
Value is $2^7 \times 11 = 1408$
$C = +7$
$I = +11$
Value is $2^3 \times (-7) = -56$
$C = +3$
$I = -7$
Value is $2^{-5} \times 5 = \frac{5}{32}$
$C = -5$
$I = +5$
Value is $2^{-6} \times -9 = -\frac{9}{64}$
$C = -6$
$I = -9$

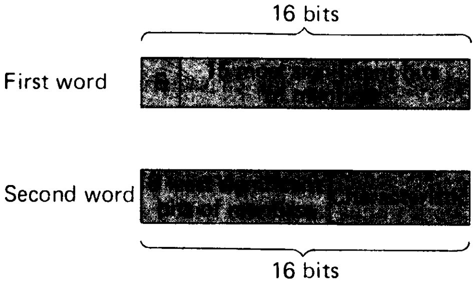


FIGURE 5.31

Values of floating-point numbers in 12-bit all-integer systems.

+15. The value of I is a sign-plus-magnitude binary integer of 7 bits, and so I can have values from -63 to $+63$. The largest number represented by this system would have a maximum I and would be 63×2^{15}.

This example shows the use of a floating-point number representation system to store "real" numbers of considerable range in a binary word.

One other widely followed practice is to express the mantissa of the word as a fraction instead of as an integer. This is in accord with common scientific usage since we commonly say that 0.93×10^4 is in "normal" form for exponential notation (and not 93×10^2). In this usage a mantissa in decimal normally has a value from 0.1 to 0.999. . . . Similarly, a binary mantissa in normal form would have a value from 0.5 (decimal) to less than 1. Most computers maintain their mantissa sections in normal form, continually adjusting words so that a significant (1) bit is always in the leftmost mantissa position.

When the mantissa is in fraction form, this section is called the *fraction*. For our 12-bit example, we can express floating-point numbers with characteristic and fraction by simply supposing the binary point to be to the left of the magnitude (and not to the right, as in integer representation). In this system a number to be represented has value $F \times 2^C$, where F is the binary fraction and C is the characteristic.

For the 12-bit word considered before, fractions would have values from $1 - 2^{-6}$, which is 0111111, to $-(1 - 2^{-6})$, which is 1111111, where the leftmost bit in each number is the sign bit. Thus numbers from $(1 - 2^{-6}) \times 2^{15}$

to $-(1 - 2^{-6}) \times 2^{15}$ can be represented, or about +32,000 to −32,000. The smallest value the fraction part could have is now the fraction 01000000, which is 2^{-1}, and the smallest characteristic, which is 2^{-15}, so the smallest positive number representable is $2^{-1} \times 2^{-15}$, or 2^{-16}. Most computers use this fractional system for the mantissa, although Burroughs and NCR use the integer system.

The Univac series of computers represents single-precision floating-point numbers in this format:

1	2 9	10 36	bit number
S	C	F	
↑	↑	↑	
sign bit	characteristic 8 bits	fraction part 27 bits	

For positive numbers, the characteristic C is treated as a binary integer, the sign bit is a 0, and the fraction part is a binary fraction with value $0.5 \leq F < 1$. The value of the number represented is $2^{C-128} \times F$. This is called an *offset system* because the value of the characteristic is simply the integer value in that portion of the word minus an offset, which in this case is 128. So the exponent can range from −128 to +127, since the integer in the characteristic section is 8 bits long.

As an example, the binary word

```
  0  10000001    1100...0
sign characteristic  fraction
```

has value $2^{129-128} \times \frac{3}{4} = 2 \times \frac{3}{4} = 1.5$. The representation for a negative number can be derived by forming the representation for the positive number with the same magnitude and then forming the 1s complement of this representation (considering all 36 bits as a single binary number).

Another example of computers with internal circuitry which performs floating-point operations and uses a single computer word representation of floating-point numbers is the IBM series.

IBM calls the exponent part the *characteristic* and the mantissa part the *fraction*. In the IBM series, floating-point data words can be either 32 or 64 bits in length. The basic formats are as follows:

Short or single-word floating-point number:

S	characteristic	fraction
0	1→7	8→31

Long or double-word floating-point number:

S	characteristic	fraction
0	1→7	8→63

In both cases, the sign bit S is in the leftmost position and gives the sign of the number. The characteristic part of the word then comprises bits 1 to 7 and is simply a binary integer, which we call C, ranging from 0 to 127. The actual value of the scale factor is formed by subtracting 64 from this integer C and raising 16 to this power. Thus the value 64 in bits 1 to 7 gives a scale factor of $16^{C-64} = 16^{64-64} = 16^0$; a 93 (decimal) in bits 1 to 7 gives a scale factor of $16^{C-64} = 16^{93-64}$, which is 16^{29}; and a 24 in bits 1 to 7 gives a 16^{-40}.

The magnitude of the actual number represented in a given floating-point word is equal to this scale factor times the fraction contained in bits 8 to 31 for the short number or 8 to 63 for a long number. The radix point is assumed to be to the left of bit 8 in either case. So if bits 8 to 31 contain 1000 . . . 00, the fraction has value $\frac{1}{2}$ (decimal); that is, the fraction is .1000 . . . 000 in binary. Similarly, if bits 8 to 31 contain 11000 . . . 000, the fraction value is $\frac{3}{4}$ decimal, or .11000 . . . 000 binary.

The actual number represented then has magnitude equal to the value of the fraction times the value determined by the characteristic. Consider a short number:

	sign	characteristic	fraction
Floating-point number:	0	1 0 0 0 0 0 1	1 1 1 0 0 . . . 0
Bit position:	0	1 2 3 4 5 6 7	8 9 10 11 12 . . . 31

The sign bit is a 0, and so the number represented is positive. The characteristic has binary value 1000001, which is 65 decimal, and so the scale factor is 16^1. The fraction part has value .111 binary or $\frac{7}{8}$ decimal, and so the number represented is $\frac{7}{8} \times 16$, or 14 decimal.

Again, consider the following number:

	sign	characteristic	fraction
Floating-point number:	1	1 0 0 0 0 0 1	1 1 1 0 0 . . . 0
Bit position:	0	1 2 3 4 5 6 7	8 9 10 11 12 . . . 31

This has value -14 since every bit is the same as before, except for the sign bit. (The number representation system is signed magnitude.)

As further examples:

sign	characteristic	fraction	
0	1 0 0 0 0 1 1	1 1 0 . . . 0	$16^3 \times \frac{3}{4} = 3072$
0	0 1 1 1 1 1 1	1 1 0 . . . 0	$16^{-1} \times \frac{3}{4} = \frac{3}{64}$

Clearly a number of floating-point number systems exist, and each manufacturer has virtually a unique system. This can present a problem to system users since programs in high-level languages may yield different results on one computer versus another. To alleviate this and several other problems, there is now a movement underway to standardize on a floating-point number system which will be made available by all manufacturers. The major effort in this area has resulted in the IEEE Proposed Standard for Binary Floating-Point Arithmetic. This standard is the result of work by several organizations (not just IEEE) and is widely supported, particularly by microcomputer manufacturers, several of which already provide actual systems conforming to the standard.

The principal feature of the standard probably is the "hidden 1" principle. Floating-point numbers generally have their fraction (magnitude) part stored with a leading 1 in the leftmost position. This is called *normalized form;* it ensures that the maximum number of significant bits is carried in the number. The reasoning behind the hidden 1 principle is that if the leftmost bit in the fraction (magnitude) section is always a 1, why carry it? Instead, this section of the floating-point number is shifted left one more bit and the 1 is discarded. However, in any reconstruction of the number for external use or during calculations, the 1 is replaced.

The IEEE standard for floating point uses the hidden 1 principle. There is a single and double format. Here is the single format.

Single format:	1	←8→	←23 bits→
	S	*E*	*F*

where S is the sign bit, E is a binary integer, and F is a binary fraction of length 23. However, the value of F is formed by adding 1 to this fraction. Thus, if F in this format is stored as 11000 . . . 00, the value of F is 1.11000 . . . 00, which is $1\frac{3}{4}$ in decimal. The value of a floating-point number in this system is

$$V = (-1)^S \times 2^{E-127} \times 1.F$$

Notice that this system uses an offset of 127 for the exponent (characteristic) value.

Here are three examples of the system:

FLOATING-POINT FORMAT (HEXADECIMAL)	$(-1)^S \times 2^{E-127} \times 1.F$	DECIMAL VALUE
3F800000	$1 \times 2^0 \times 1.0$	+1
BF800000	$-1 \times 2^0 \times 1.0$	−1
40400000	$1 \times 2^1 \times 1.5$	+3

Note that fraction values for F range from 1 to slightly less than 2 ($1 \leq$ value of $F < 2$).

Double format:	1	←11→	←52 bits→
	S	*E*	*F*

where S is the sign bit, E is an 11-bit binary integer, and F is a 52-bit binary fraction with binary point to the far left. However, the value for F is formed by