CHAPTER

6

ARITHMETIC

CHAPTER OBJECTIVES

In this chapter you will learn about:

- High-speed adders, implemented in a hierarchical structure, using carry-lookahead logic to generate carry signals in parallel
- The Booth algorithm, used to determine how multiplicand summands are selected by the multiplier bit patterns in performing multiplication of signed numbers
- High-speed multipliers, which use carry-save addition to add summands in parallel
- Circuits that perform division operations
- The representation of floating-point numbers in the IEEE standard format, and how to perform basic arithmetic operations on them

A basic operation in all digital computers is the addition or subtraction of two numbers. Arithmetic operations occur at the machine instruction level. They are implemented, along with basic logic functions such as AND, OR, NOT, and EXCLUSIVE-OR (XOR), in the arithmetic and logic unit (ALU) subsystem of the processor, as discussed in Chapter 1. In this chapter, we present the logic circuits used to implement arithmetic operations. The time needed to perform an addition operation affects the processor's performance. Multiply and divide operations, which require more complex circuitry than either addition or subtraction operations, also affect performance. We present some of the techniques used in modern computers to perform arithmetic operations at high speed.

Compared with arithmetic operations, logic operations are simple to implement using combinational circuitry. They require only independent Boolean operations on individual bit positions of the operands, whereas carry/borrow lateral signals are required in arithmetic operations.

In Section 2.1, we described the representation of signed binary numbers, and showed that 2's-complement is the best representation from the standpoint of performing addition and subtraction operations. The examples in Figure 2.4 show that two, n-bit, signed numbers can be added using n-bit binary addition, treating the sign bit the same as the other bits. In other words, a logic circuit that is designed to add unsigned binary numbers can also be used to add signed numbers in 2's-complement. If overflow does not occur, the sum is correct, and any output carry can be ignored. The first two sections of this chapter present logic circuit networks for addition and subtraction.

6.1 ADDITION AND SUBTRACTION OF SIGNED NUMBERS

Figure 6.1 shows the logic truth table for the sum and carry-out functions for adding equally weighted bits x_i and y_i in two numbers X and Y. The figure also shows logic expressions for these functions, along with an example of addition of the 4-bit unsigned numbers 7 and 6. Note that each stage of the addition process must accommodate a carry-in bit. We use c_i to represent the carry-in to the ith stage, which is the same as the carry-out from the $(i-1)$st stage.

The logic expression for s_i in Figure 6.1 can be implemented with a 3-input XOR gate, used in Figure 6.2a as part of the logic required for a single stage of binary addition. The carry-out function, c_{i+1}, is implemented with a two-level AND-OR logic circuit. A convenient symbol for the complete circuit for a single stage of addition, called a *full adder* (FA), is also shown in the figure.

A cascaded connection of n full adder blocks, as shown in Figure 6.2b, can be used to add two n-bit numbers. Since the carries must propagate, or ripple, through this cascade, the configuration is called an *n-bit ripple-carry adder*.

The carry-in, c_0, into the *least-significant-bit* (LSB) position provides a convenient means of adding 1 to a number. For instance, forming the 2's-complement of a number involves adding 1 to the 1's-complement of the number. The carry signals are also useful for interconnecting k adders to form an adder capable of handling input numbers that are kn bits long, as shown in Figure 6.2c.

x_i	y_i	Carry-in c_i	Sum s_i	Carry-out c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s_i = x_i y_i c_i + x_i y_i c_i + x_i y_i c_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$
$$c_{i+1} = y_i c_i + x_i c_i + x_i y_i$$

Example:

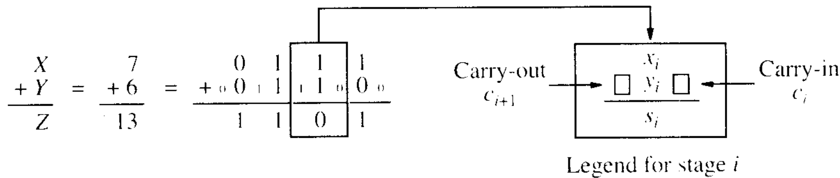


Figure 6.1 Logic specification for a stage of binary addition.

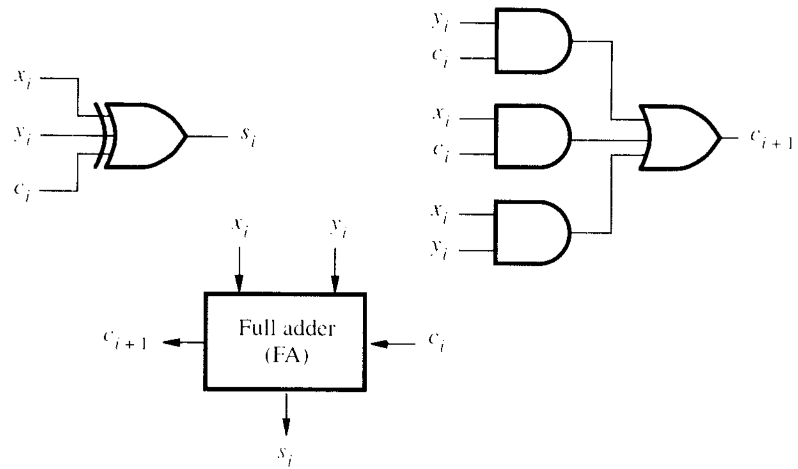
6.1.1 ADDITION/SUBTRACTION LOGIC UNIT

The n-bit adder in Figure 6.2b can be used to add 2's-complement numbers X and Y, where the x_{n-1} and y_{n-1} bits are the sign bits. In this case, the carry-out bit, c_n, is not part of the answer. In section 2.1.4, arithmetic overflow was discussed. Overflow can only occur when the signs of the two operands are the same. In this case, overflow obviously occurs if the sign of the result is different. Therefore, a circuit to detect overflow can be added to the n-bit adder by implementing the logic expression

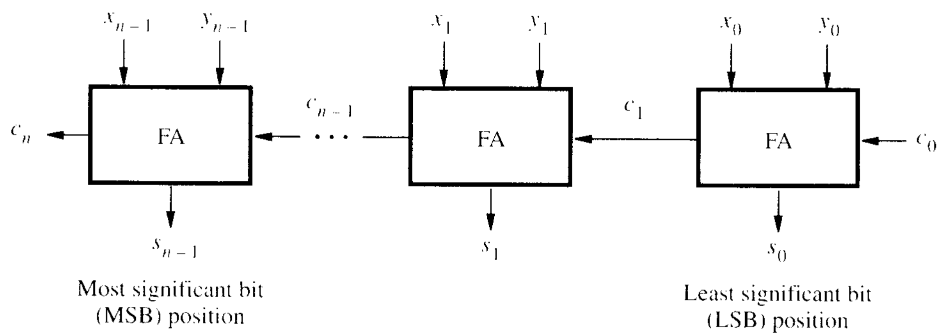
$$\text{Overflow} = x_{n-1} y_{n-1} \overline{s}_{n-1} + \overline{x}_{n-1} \overline{y}_{n-1} s_{n-1}$$

It can also be shown that overflow occurs when the carry bits c_n and c_{n-1} are different. (See Problem 6.9.) Therefore, a simpler alternative circuit for detecting overflow can be obtained by implementing the expression $c_n \oplus c_{n-1}$ with an XOR gate.

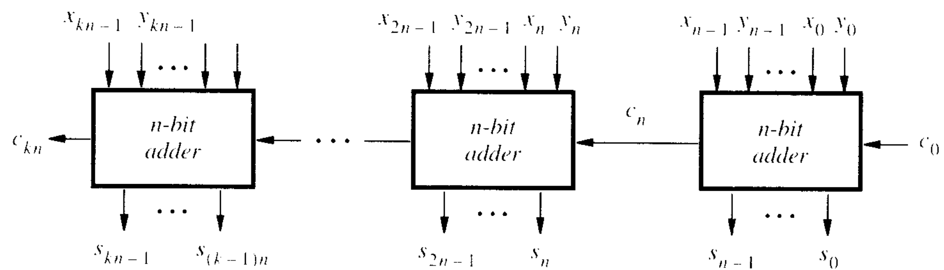
In order to perform the subtraction operation $X - Y$ on 2's-complement numbers X and Y, we form the 2's-complement of Y and add it to X. The logic circuit network shown in Figure 6.3 can be used to perform either addition or subtraction based on the value applied to the Add/Sub input control line. This line is set to 0 for addition, applying the Y vector unchanged to one of the adder inputs along with a carry-in signal, c_0,



(a) Logic for a single stage



(b) An n-bit ripple-carry adder



(c) Cascade of k n-bit adders

Figure 6.2 Logic for addition of binary vectors.

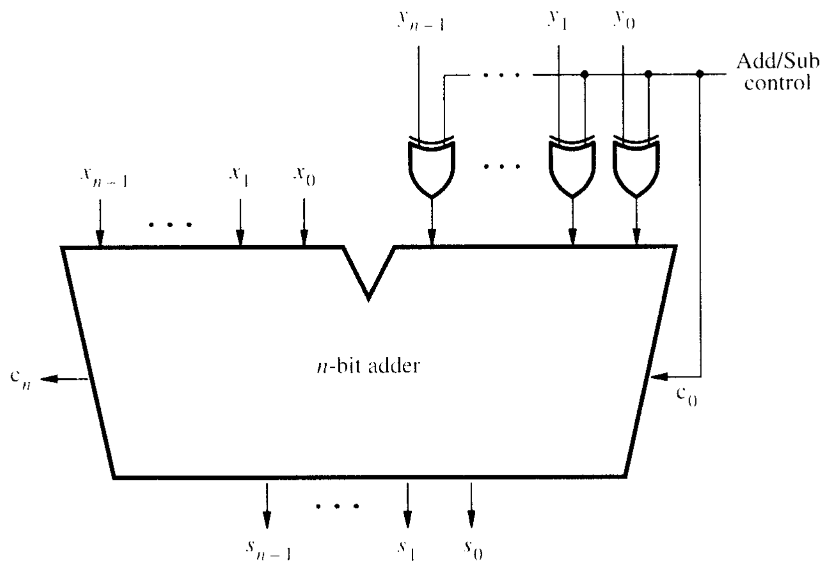


Figure 6.3 Binary addition-subtraction logic network.

of 0. When the Add/Sub control line is set to 1, the Y vector is 1's-complemented (that is, bit complemented) by the XOR gates and c_0 is set to 1 to complete the 2's-complementation of Y. Remember that 2's-complementing a negative number is done in exactly the same manner as for a positive number. An XOR gate can be added to Figure 6.3 to detect the overflow condition $c_n \oplus c_{n-1}$.

6.2 DESIGN OF FAST ADDERS

If an n-bit ripple-carry adder is used in the addition/subtraction unit of Figure 6.3, it may have too much delay in developing its outputs, s_0 through s_{n-1} and c_n. Whether or not the delay incurred is acceptable can be decided only in the context of the speed of other processor components and the data transfer times of registers and cache memories. The delay through a network of logic gates depends on the integrated circuit electronic technology (see Appendix A) used in fabricating the network and on the number of gates in the paths from inputs to outputs. The delay through any combinational logic network constructed from gates in a particular technology is determined by adding up the number of logic-gate delays along the longest signal propagation path through the network. In the case of the n-bit ripple-carry adder, the longest path is from inputs x_0, y_0, and c_0 at the LSB position to outputs c_n and s_{n-1} at the *most-significant-bit* (MSB) position.

Using the logic implementation indicated in Figure 6.2*a*, c_{n-1} is available in $2(n-1)$ gate delays, and s_{n-1} is correct one XOR gate delay later. The final carry-out, c_n, is available after $2n$ gate delays. Therefore, if a ripple-carry adder is used to implement the addition/subtraction unit shown in Figure 6.3, all sum bits are available in $2n$ gate delays,

including the delay through the XOR gates on the Y input. Using the implementation $c_n \oplus c_{n-1}$ for overflow, this indicator is available after $2_n + 2$ gate delays.

Two approaches can be taken to reduce delay in adders. The first approach is to use the fastest possible electronic technology in implementing the ripple-carry logic design or variations of it. The second approach is to use an augmented logic gate network structure that is larger than that shown in Figure 6.2*b*. We will describe an easily understood version of the second approach in the next section. In practice, a number of design techniques have been used to implement high-speed adders. They include electronic circuit designs for fast propagation of carry signals as well as variations on the basic network structure presented in the next section.

6.2.1 Carry-Lookahead Addition

A fast adder circuit must speed up the generation of the carry signals. The logic expressions for s_i (sum) and c_{i+1} (carry-out) of stage i (see Figure 6.1) are

$$s_i = x_i \oplus y_i \oplus c_i$$

and

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

Factoring the second equation into

$$c_{i+1} = x_i y_i + (x_i + y_i)c_i$$

we can write

$$c_{i+1} = G_i + P_i c_i$$

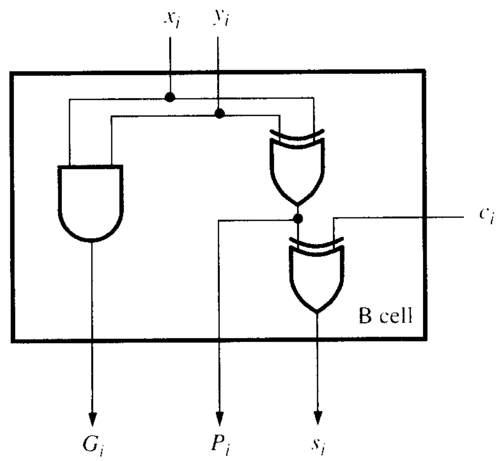
where

$$G_i = x_i y_i \quad \text{and} \quad P_i = x_i + y_i$$

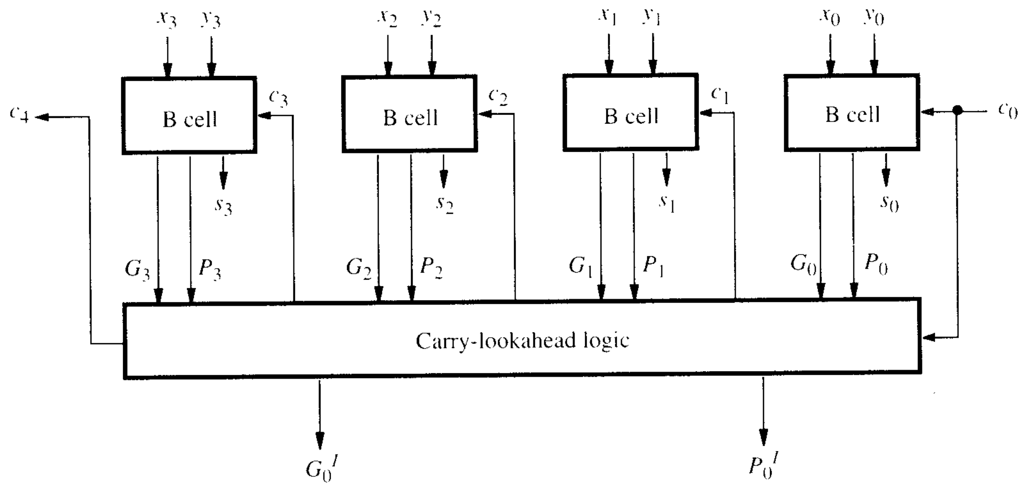
The expressions G_i and P_i are called the *generate* and *propagate* functions for stage i. If the generate function for stage i is equal to 1, then $c_{i+1} = 1$, independent of the input carry, c_i. This occurs when both x_i and y_i are 1. The propagate function means that an input carry will produce an output carry when either x_i is 1 or y_i is 1. All G_i and P_i functions can be formed independently and in parallel in one logic-gate delay after the X and Y vectors are applied to the inputs of an n-bit adder. Each bit stage contains an AND gate to form G_i, an OR gate to form P_i, and a three-input XOR gate to form s_i. A simpler circuit can be derived by observing that an adequate propagate function can be realized as $P_i = x_i \oplus y_i$, which differs from $P_i = x_i + y_i$ only when $x_i = y_i = 1$. But, in this case $G_i = 1$, so it does not matter whether P_i is 0 or 1. Then, using a cascade of two 2-input XOR gates to realize the 3-input XOR function, the basic cell B in Figure 6.4*a* can be used in each bit stage.

Expanding c_i in terms of $i - 1$ subscripted variables and substituting into the c_{i+1} expression, we obtain

$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} c_{i-1}$$



(a) Bit-stage cell



(b) 4-bit adder

Figure 6.4 4-bit carry-lookahead adder.

Continuing this type of expansion, the final expression for any carry variable is

$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \cdots + P_i P_{i-1} \cdots P_1 G_0 + P_i P_{i-1} \cdots P_0 c_0 \quad \text{[6.1]}$$

Thus, all carries can be obtained three gate delays after the input signals X, Y, and c_0 are applied because only one gate delay is needed to develop all P_i and G_i signals,

followed by two gate delays in the AND-OR circuit for c_{i+1}. After a further XOR gate delay, all sum bits are available. Therefore, independent of n, the n-bit addition process requires only four gate delays.

Let us consider the design of a 4-bit adder. The carries can be implemented as

$$c_1 = G_0 + P_0 c_0$$
$$c_2 = G_1 + P_1 G_0 + P_1 P_0 c_0$$
$$c_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0$$
$$c_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$$

The complete 4-bit adder is shown in Figure 6.4*b*. The carries are implemented in the block labeled carry-lookahead logic. An adder implemented in this form is called a *carry-lookahead adder*. Delay through the adder is 3 gate delays for all carry bits and 4 gate delays for all sum bits. In comparison, note that a 4-bit ripple-carry adder requires 7 gate delays for s_3 and 8 gate delays for c_4.

If we try to extend the carry-lookahead adder of Figure 6.4*b* for longer operands, we run into a problem of gate fan-in constraints. From Expression 6.1, we see that the last AND gate and the OR gate require a fan-in of $i + 2$ in generating c_{i+1}. For c_4 in the 4-bit adder, a fan-in of 5 is required. This is about the limit for practical gates. So the adder design shown in Figure 6.4*b* cannot be directly extended to longer operand sizes. However, if we cascade a number of 4-bit adders, as shown in Figure 6.2*c*, it is possible to build longer adders.

Eight 4-bit carry-lookahead adders can be connected as in Figure 6.2*c* to form a 32-bit adder. The delays in generating sum bits $s_{31}, s_{30}, s_{29}, s_{28}$, and c_{32} in the high-order 4-bit adder in this cascade are calculated as follows. The carry-out c_4 from the low-order adder is available 3 gate delays after the input operands X, Y, and c_0 are applied to the 32-bit adder. Then, c_8 is available at the output of the second adder after a further 2 gate delays, c_{12} is available after a further 2 gate delays, and so on. Finally, c_{28}, the carry-in to the high-order 4-bit adder, is available after a total of $(6 \times 2) + 3 = 15$ gate delays. Then, c_{32} and all carries inside the high-order adder are available after a further 2 gate delays, and all 4 sum bits are available after 1 more gate delay, for a total of 18 gate delays. This should be compared to total delays of 63 and 64 for s_{31} and c_{32} if a ripple-carry adder is used.

In the next section, we show how it is possible to improve upon the cascade structure just discussed, leading to further reduction in adder delay. The key idea is to generate the carries $c_4, c_8, \ldots$ in parallel, similar to the way that c_1, c_2, c_3, and c_4, are generated in parallel in the 4-bit carry-lookahead adder.

Higher-Level Generate and Propagate Functions

In the 32-bit adder just discussed, the carries $c_4, c_8, c_{12}, \ldots$ ripple through the 4-bit adder blocks with two gate delays per block, analogous to the way that individual carries ripple through each bit stage in a ripple-carry adder. By using higher-level block generate and propagate functions, it is possible to use the lookahead approach to develop the carries $c_4, c_8, c_{12}, \ldots$ in parallel, in a higher-level carry-lookahead circuit.

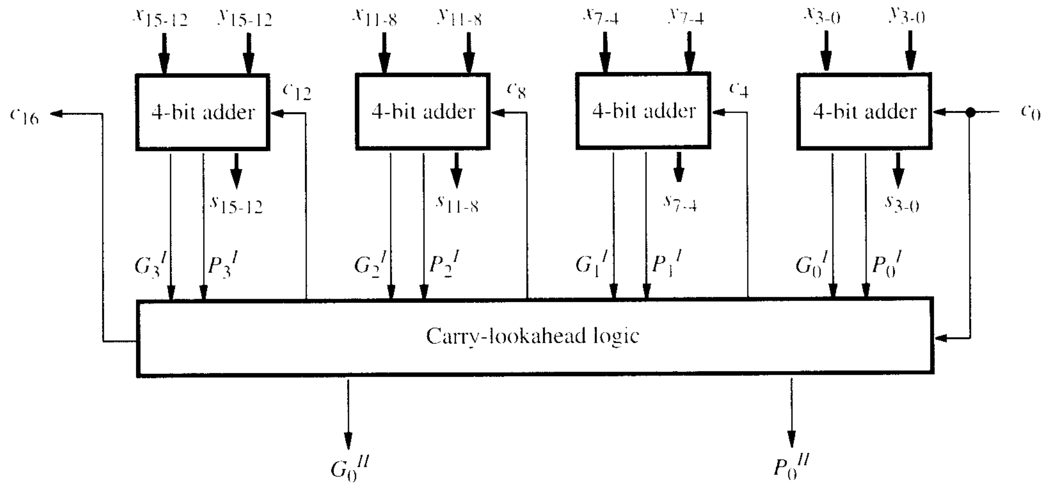


Figure 6.5 16-bit carry-lookahead adder built from 4-bit adders (see Figure 6.4*b*).

Figure 6.5 shows a 16-bit adder built from four 4-bit adder blocks. These blocks provide new output functions defined as G_k^I and P_k^I, where $k = 0$ for the first 4-bit block, as shown in Figure 6.4*b*, $k = 1$ for the second 4-bit block, and so on. In the first block,

$$P_0^I = P_3 P_2 P_1 P_0$$

and

$$G_0^I = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0$$

In words, we say that the first-level G_i and P_i functions determine whether bit stage i generates or propagates a carry, and that the second-level G_k^I and P_k^I functions determine whether block k generates or propagates a carry. With these new functions available, it is not necessary to wait for carries to ripple through the 4-bit blocks. Carry c_{16} is formed by one of the carry-lookahead circuits in Figure 6.5 as

$$c_{16} = G_3^I + P_3^I G_2^I + P_3^I P_2^I G_1^I + P_3^I P_2^I P_1^I G_0^I + P_3^I P_2^I P_1^I P_0^I c_0$$

The input carries to the 4-bit blocks are formed in parallel by similar shorter expressions. These expressions for c_{16}, c_{12}, c_8, and c_4, are identical in form to the expressions for c_4, c_3, c_2, and c_1, respectively, implemented in the carry-lookahead circuits in Figure 6.4*b*. Only the variable names are different. Therefore, the structure of the carry-lookahead circuits in Figure 6.5 is identical to the carry-lookahead circuits in Figure 6.4*b*. We should note, however, that the carries c_4, c_8, c_{12}, and c_{16}, generated internally by the 4-bit adder blocks, are not needed in Figure 6.5 because they are generated by the higher-level carry-lookahead circuits.

Now, consider the delay in producing outputs from the 16-bit carry-lookahead adder. The delay in developing the carries produced by the carry-lookahead circuits is

two gate delays more than the delay needed to develop the G_k^I and P_k^I functions. The latter require two gate delays and one gate delay, respectively, after the generation of G_i and P_i. Therefore, all carries produced by the carry-lookahead circuits are available 5 gate delays after X, Y, and c_0 are applied as inputs. The carry c_{15} is generated inside the high-order 4-bit block in Figure 6.5 in two gate delays after c_{12}, followed by s_{15} in one further gate delay. Therefore, s_{15} is available after 8 gate delays. Note that if a 16-bit adder is built by cascading 4-bit carry-lookahead adder blocks, the delays in developing c_{16} and s_{15} are 9 and 10 gate delays, respectively, as compared to 5 and 8 gate delays for the configuration in Figure 6.5.

Two 16-bit adder blocks can be cascaded to implement a 32-bit adder. In this configuration, the output c_{16} from the low-order block is the carry input to the high-order block. The delay is much lower than the delay through the 32-bit adder that we discussed earlier, which was built by cascading eight 4-bit adders. In that configuration, recall that s_{31} is available after 18 gate delays and c_{32} is available after 17 gate delays. The delay analysis for the cascade of two 16-bit adders is as follows. The carry c_{16} out of the low-order block is available after 5 gate delays, as calculated above. Then, both c_{28} and c_{32} are available in the high-order block after a further 2 gate delays, and c_{31} is available 2 gate delays after c_{28}. Therefore, c_{31} is available after a total of 9 gate delays, and s_{31} is available in 10 gate delays. Recapitulating, s_{31} and c_{32} are available after 10 and 7 gate delays, respectively, compared to 18 and 17 gate delays for the same outputs if the 32-bit adder is built from a cascade of eight 4-bit adders.

The same reasoning used in developing second-level G_k^I and P_k^I functions from first-level G_i and P_i functions can be used to develop third-level G_k^{II} and P_k^{II} functions from G_k^I and P_k^I functions. Two such third-level functions are shown as outputs from the carry-lookahead logic in Figure 6.5. A 64-bit adder can be built from four of the 16-bit adders shown in Figure 6.5 along with additional carry-lookahead logic circuits that produces carries c_{16}, c_{32}, c_{48}, and c_{64}. Delay through this adder can be shown to be 12 gate delays for s_{63} and 7 gate delays for c_{64}, using an extension of the reasoning used above for the 16-bit adder. (See Problem 6.10.)

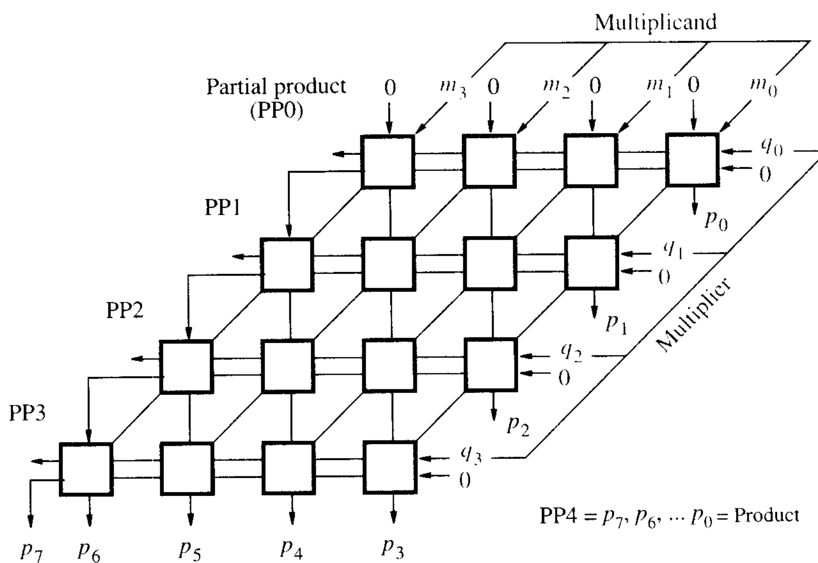
6.3 Multiplication of Positive Numbers

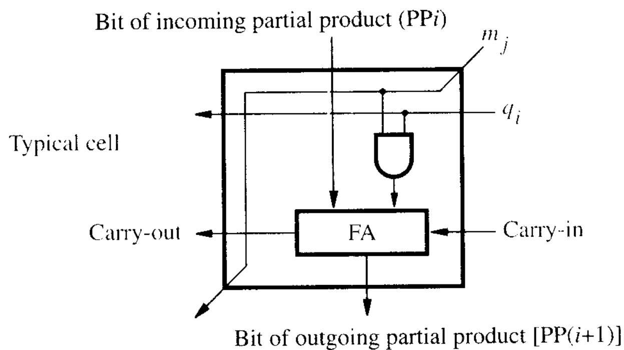
The usual algorithm for multiplying integers by hand is illustrated in Figure 6.6*a* for the binary system. This algorithm applies to unsigned numbers and to positive signed numbers. The product of two n-digit numbers can be accommodated in $2n$ digits, so the product of the two 4-bit numbers in this example fits into 8 bits, as shown. In the binary system, multiplication of the multiplicand by one bit of the multiplier is easy. If the multiplier bit is 1, the multiplicand is entered in the appropriate position to be added to the partial product. If the multiplier bit is 0, then 0s are entered, as in the third row of the example.

Binary multiplication of positive operands can be implemented in a combinational, two-dimensional logic array, as shown in Figure 6.6*b*. The main component in each cell is a full adder FA. The AND gate in each cell determines whether a multiplicand bit, m_j, is added to the incoming partial-product bit, based on the value of the multiplier

```
          1 1 0 1     (13) Multiplicand M
        × 1 0 1 1     (11) Multiplier Q
        ---------
          1 1 0 1
        1 1 0 1
      0 0 0 0
    1 1 0 1
  ---------------
  1 0 0 0 1 1 1 1     (143) Product P
```

(a) Manual multiplication algorithm





(b) Array implementation

Figure 6.6 Array multiplication of positive binary operands.

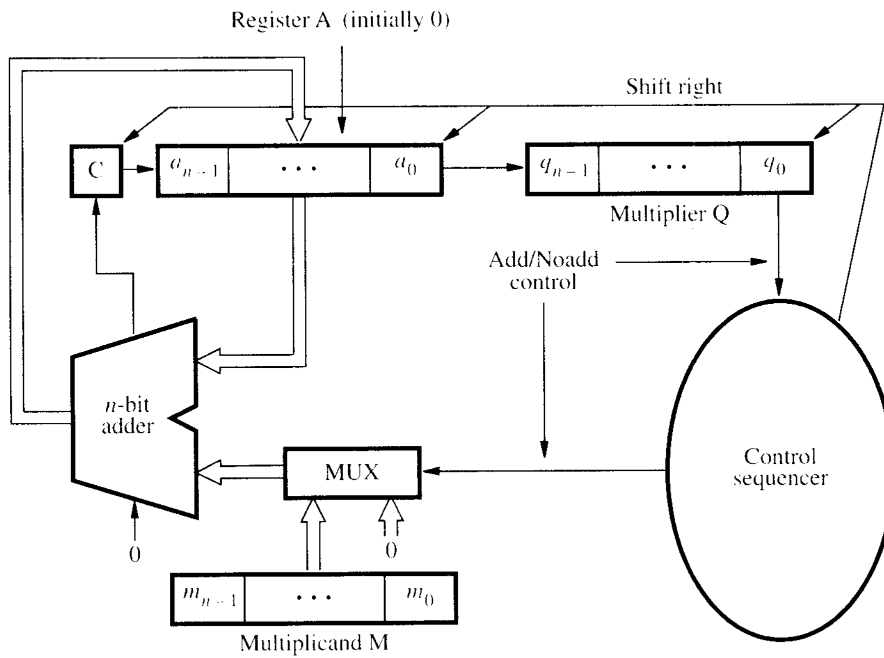
bit, q_i. Each row i, where $0 \leq i \leq 3$, adds the multiplicand (appropriately shifted) to the incoming partial product, PPi, to generate the outgoing partial product, PP$(i + 1)$, if $q_i = 1$. If $q_i = 0$, PPi is passed vertically downward unchanged. PP0 is all 0s, and PP4 is the desired product. The multiplicand is shifted left one position per row by the diagonal signal path.

The worst case signal propagation delay path is from the upper right corner of the array to the high-order product bit output at the bottom left corner of the array. The path consists of the staircase pattern that includes the two cells at the right end of each row, followed by all the cells in the bottom row. Assuming that there are two gate delays from the inputs to the outputs of a full adder block, the path has a total of $6(n - 1) - 1$ gate delays, including the initial AND gate delay in all cells, for the $n \times n$ array. (See Problem 6.12.) Only the AND gates are actually needed in the first row of the array because the incoming partial product PP0 is zero. This has been taken into account in developing the delay expression.

Multiplication is usually provided in the machine instruction set of a processor. High-performance processor chips use an appreciable area of the chip to perform arithmetic functions on both integer and floating-point operands. (Floating-point operations are discussed later in this chapter.) Although the preceding combinational multiplier is easy to understand, it uses many gates for multiplying numbers of practical size, such as 32- or 64-bit numbers. Multiplication can also be performed using a mixture of combinational array techniques, similar to those shown in Figure 6.6, and sequential techniques requiring less combinational logic.

The simplest way to perform multiplication is to use the adder circuitry in the ALU for a number of sequential steps. The block diagram in Figure 6.7*a* shows the hardware arrangement for sequential multiplication. This circuit performs multiplication by using a single n-bit adder n times to implement the spatial addition performed by the n rows of ripple-carry adders of Figure 6.6*b*. Registers A and Q combined hold PPi while multiplier bit q_i generates the signal Add/Noadd. This signal controls the addition of the multiplicand, M, to PPi to generate PP$(i + 1)$. The product is computed in n cycles. The partial product grows in length by one bit per cycle from the initial vector, PP0, of n 0s in register A. The carry-out from the adder is stored in flip-flop C, shown at the left end of register A. At the start, the multiplier is loaded into register Q, the multiplicand into register M, and C and A are cleared to 0. At the end of each cycle, C, A, and Q are shifted right one bit position to allow for growth of the partial product as the multiplier is shifted out of register Q. Because of this shifting, multiplier bit q_i appears at the LSB position of Q to generate the Add/Noadd signal at the correct time, starting with q_0 during the first cycle, q_1 during the second cycle, and so on. After they are used, the multiplier bits are discarded by the right-shift operation. Note that the carry-out from the adder is the leftmost bit of PP$(i + 1)$, and it must be held in the C flip-flop to be shifted right with the contents of A and Q. After n cycles, the high-order half of the product is held in register A and the low-order half is in register Q. The multiplication example of Figure 6.6*a* is shown in Figure 6.7*b* as it would be performed by this hardware arrangement.

Using this sequential hardware structure, it is clear that a Multiply instruction takes much more time to execute than an Add instruction. Several techniques have been used to speed up multiplication; we discuss some of them in the next few sections.



(a) Register configuration

M: 1 1 0 1

C	A	Q		
0	0 0 0 0	1 0 1 1		Initial configuration
0	1 1 0 1	1 0 1 1	Add	First cycle
0	0 1 1 0	1 1 0 1	Shift	
1	0 0 1 1	1 1 0 1	Add	Second cycle
0	1 0 0 1	1 1 1 0	Shift	
0	1 0 0 1	1 1 1 0	No add	Third cycle
0	0 1 0 0	1 1 1 1	Shift	
1	0 0 0 1	1 1 1 1	Add	Fourth cycle
0	1 0 0 0	1 1 1 1	Shift	

Product (A and Q of the final row)

(b) Multiplication example

Figure 6.7 Sequential circuit binary multiplier.

6.4 SIGNED-OPERAND MULTIPLICATION

We now discuss multiplication of 2's-complement signed operands, generating a double-length product. The general strategy is still to accumulate partial products by adding versions of the multiplicand as selected by the multiplier bits.

First, consider the case of a positive multiplier and a negative multiplicand. When we add a negative multiplicand to a partial product, we must extend the sign-bit value of the multiplicand to the left as far as the product will extend. In Figure 6.8, for example, the 5-bit signed operand, -13, is the multiplicand, and it is multiplied by $+11$, the multiplier, to get the 10-bit product, -143. The sign extension of the multiplicand is shown in blue. Thus, the hardware discussed earlier can be used for negative multiplicands if it provides for sign extension of the partial products.

For a negative multiplier, a straightforward solution is to form the 2's-complement of both the multiplier and the multiplicand and proceed as in the case of a positive multiplier. This is possible because complementation of both operands does not change the value or the sign of the product. A technique that works equally well for both negative and positive multipliers, called the Booth algorithm, is described next.

6.4.1 BOOTH ALGORITHM

The Booth algorithm generates a $2n$-bit product and treats both positive and negative 2's-complement n-bit operands uniformly. To understand the essence of this algorithm, consider a multiplication operation in which the multiplier is positive and has a single block of 1s, for example, 0011110. To derive the product, we could add four appropriately shifted versions of the multiplicand, as in the standard procedure. However, we can reduce the number of required operations by regarding this multiplier as the

						1	0	0	1	1	(−13)
					×	0	1	0	1	1	(+11)
	1	1	1	1	1	1	0	0	1	1	
	1	1	1	1	1	0	0	1	1		
Sign extension is shown in blue	0	0	0	0	0	0	0	0			
	1	1	1	0	0	1	1				
	0	0	0	0	0	0					
	1	1	0	1	1	1	0	0	0	1	(−143)

Figure 6.8 Sign extension of negative multiplicand.

							0	1	0	1	1	0	1
							0	0	+1	+1	+1	+1	0
							0	0	0	0	0	0	0
						0	1	0	1	1	0	1	
					0	1	0	1	1	0	1		
				0	1	0	1	1	0	1			
			0	1	0	1	1	0	1				
		0	0	0	0	0	0	0					
	0	0	0	0	0	0	0						
0	0	0	1	0	1	0	1	0	0	0	1	1	0

							0	1	0	1	1	0	1	
							0	+1	0	0	0	−1	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	1	1	1	1	1	1	0	1	0	0	1	1		← 2's complement of the multiplicand
0	0	0	0	0	0	0	0	0	0	0	0			
0	0	0	0	0	0	0	0	0	0	0				
0	0	0	0	0	0	0	0	0	0					
0	0	0	1	0	1	1	0	1						
0	0	0	0	0	0	0	0							
0	0	0	1	0	1	0	1	0	0	0	1	1	0	

Figure 6.9 Normal and Booth multiplication schemes.

difference between two numbers:

$$\begin{array}{rr} 0100000 & (32) \\ -0000010 & (2) \\ \hline 0011110 & (30) \end{array}$$

This suggests that the product can be generated by adding 2^5 times the multiplicand to the 2's-complement of 2^1 times the multiplicand. For convenience, we can describe the sequence of required operations by recoding the preceding multiplier as $0\ {+1}000\ {-1}0$.

In general, in the Booth scheme, −1 times the shifted multiplicand is selected when moving from 0 to 1, and +1 times the shifted multiplicand is selected when moving from 1 to 0, as the multiplier is scanned from right to left. Figure 6.9 illustrates the normal and the Booth algorithms for the example just discussed. The Booth algorithm clearly extends to any number of blocks of 1s in a multiplier, including the situation in which a single 1 is considered a block. See Figure 6.10 for another example of recoding a multiplier. The case when the least significant bit of the multiplier is 1 is handled by assuming that an implied 0 lies to its right. The Booth algorithm can also be used directly for negative multipliers, as shown in Figure 6.11.

To demonstrate the correctness of the Booth algorithm for negative multipliers, we use the following property of negative-number representations in the 2's-complement

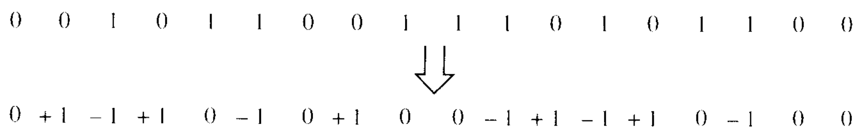


Figure 6.10 Booth recoding of a multiplier.

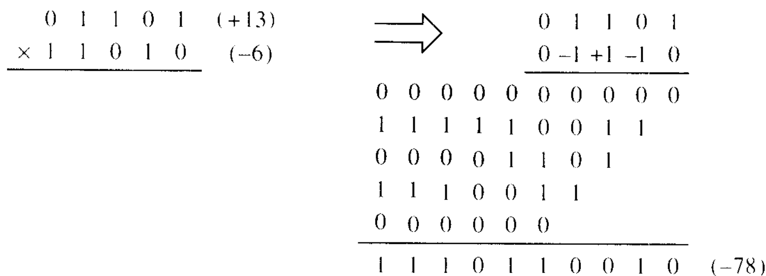


Figure 6.11 Booth multiplication with a negative multiplier.

system: Let the leftmost 0 of a negative number, X, be at bit position k, that is,

$$X = 11\ldots10x_{k-1}\ldots x_0$$

Then the value of X is given by

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

The correctness of this expression for $V(X)$ is shown by observing that if X is formed as the sum of two numbers

$$\begin{array}{rl} & 11\ldots100000\ldots0 \\ + & 00\ldots00x_{k-1}\ldots x_0 \\ \hline & X = 11\ldots10x_{k-1}\ldots x_0 \end{array}$$

then the top number is the 2's-complement representation of -2^{k+1}. The recoded multiplier now consists of the part corresponding to the second number, with -1 added in position $k+1$. For example, the multiplier 110110 is recoded as 0 −1 +1 0 −1 0.

The Booth technique for recoding multipliers is summarized in Figure 6.12. The transformation 011 . . . 110 ⇒ +100 . . . 0 −10 is called *skipping over 1s*. This term is derived from the case in which the multiplier has its 1s grouped into a few contiguous blocks. Only a few versions of the shifted multiplicand (the summands) must be added to generate the product, thus speeding up the multiplication operation. However, in the worst case — that of alternating 1s and 0s in the multiplier — each bit of the multiplier selects a summand. In fact, this results in more summands than if the Booth algorithm were not used. A 16-bit, worst-case multiplier, an ordinary multiplier, and a good multiplier are shown in Figure 6.13.

Multiplier		Version of multiplicand selected by bit i
Bit i	Bit $i-1$	
0	0	$0 \times M$
0	1	$+1 \times M$
1	0	$-1 \times M$
1	1	$0 \times M$

Figure 6.12 Booth multiplier recoding table.

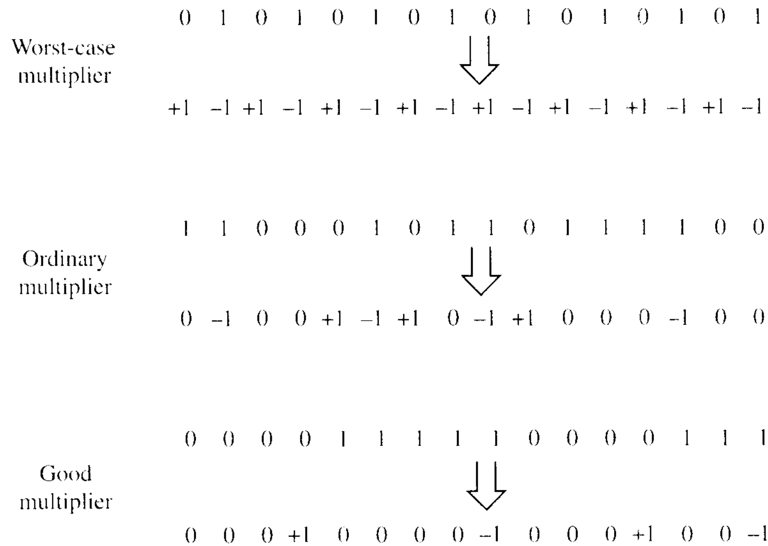


Figure 6.13 Booth recoded multipliers.

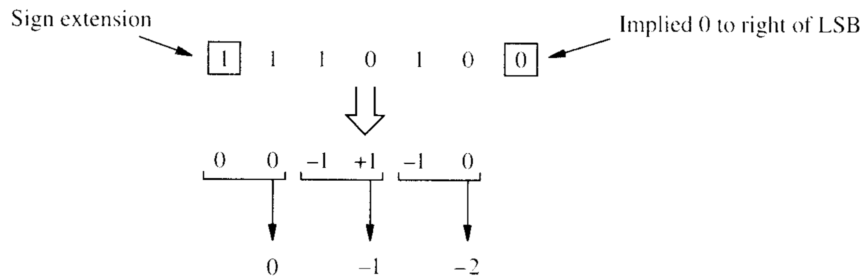
The Booth algorithm has two attractive features. First, it handles both positive and negative multipliers uniformly. Second, it achieves some efficiency in the number of additions required when the multiplier has a few large blocks of 1s. The speed gained by skipping over 1s depends on the data. On average, the speed of doing multiplication with the Booth algorithm is the same as with the normal algorithm.

6.5 FAST MULTIPLICATION

We now describe two techniques for speeding up the multiplication operation. The first technique guarantees that the maximum number of summands (versions of the multiplicand) that must be added is $n/2$ for n-bit operands. The second technique reduces the time needed to add the summands.

6.5.1 BIT-PAIR RECODING OF MULTIPLIERS

A technique called *bit-pair recoding* halves the maximum number of summands. It is derived directly from the Booth algorithm. Group the Booth-recoded multiplier bits in pairs, and observe the following: The pair (+1 −1) is equivalent to the pair (0 +1). That is, instead of adding −1 times the multiplicand M at shift position i to $+1 \times M$ at position $i+1$, the same result is obtained by adding $+1 \times M$ at position i. Other examples are: (+1 0) is equivalent to (0 +2), (−1 +1) is equivalent to (0 −1), and so on. Thus, if the Booth-recoded multiplier is examined two bits at a time, starting from the right, it can be rewritten in a form that requires at most one version of the multiplicand to be added to the partial product for each pair of multiplier bits. Figure 6.14*a* shows an example of bit-pair recoding of the multiplier in Figure 6.11, and



(a) Example of bit-pair recoding derived from Booth recoding

Multiplier bit-pair		Multiplier bit on the right	Multiplicand selected at position i
$i+1$	i	$i-1$	
0	0	0	$0 \times M$
0	0	1	$+1 \times M$
0	1	0	$+1 \times M$
0	1	1	$+2 \times M$
1	0	0	$-2 \times M$
1	0	1	$-1 \times M$
1	1	0	$-1 \times M$
1	1	1	$0 \times M$

(b) Table of multiplicand selection decisions

Figure 6.14 Multiplier bit-pair recoding.

```
                0  1  1  0  1   (+13)
              × 1  1  0  1  0   (-6)
              ---------------

                      ⇩

                0  1  1  0  1
                0 -1 +1 -1  0
              ---------------
 0  0  0  0  0  0  0  0  0  0
 1  1  1  1  1  0  0  1  1
 0  0  0  0  1  1  0  1
 1  1  1  0  0  1  1
 0  0  0  0  0  0
------------------------------
 1  1  1  0  1  1  0  0  1  0   (-78)

                      ⇩

                0  1  1  0  1
                0    -1    -2
              ---------------
 1  1  1  1  1  0  0  1  1  0
 1  1  1  1  0  0  1  1
 0  0  0  0  0  0
------------------------------
 1  1  1  0  1  1  0  0  1  0
```

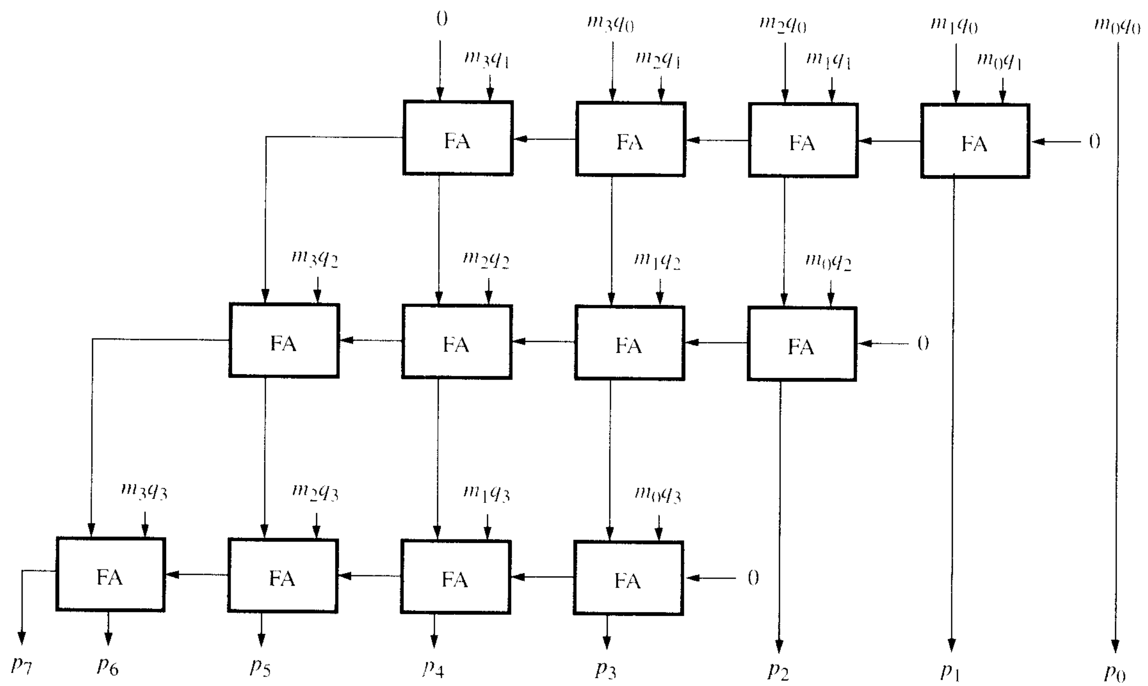
Figure 6.15 Multiplication requiring only $n/2$ summands.

Figure 6.14*b* shows a table of the multiplicand selection decisions for all possibilities. The multiplication operation in Figure 6.11 is shown in Figure 6.15 as it would be computed using bit-pair recoding of the multiplier.

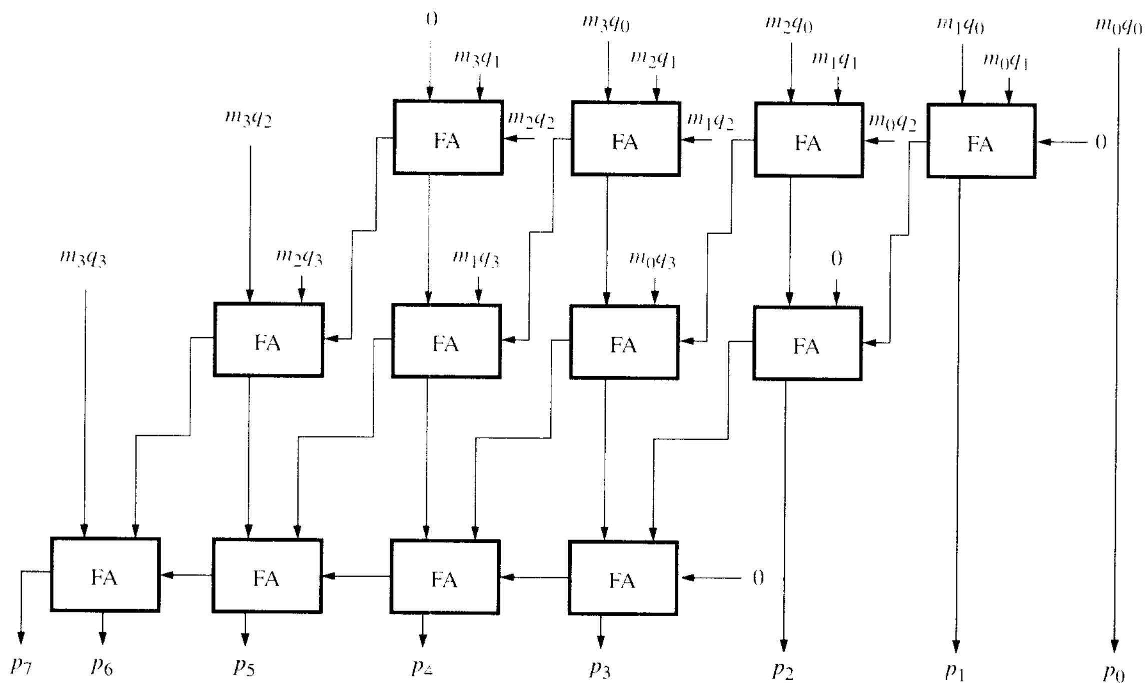
6.5.2 CARRY-SAVE ADDITION OF SUMMANDS

Multiplication requires the addition of several summands. A technique called *carry-save addition* (CSA) speeds up the addition process. Consider the array for 4 × 4 multiplication shown in Figure 6.16*a*. This structure is the general array shown in Figure 6.6, with the first row consisting of just the AND gates that implement the bit products m_3q_0, m_2q_0, m_1q_0, and m_0q_0.

Instead of letting the carries ripple along the rows, they can be "saved" and introduced into the next row, at the correct weighted positions, as shown in Figure 6.16*b*. This frees up an input to three full adders in the first row. These inputs are used to



(a) Ripple-carry array (Figure 6.6 structure)



(b) Carry-save array

Figure 6.16 Ripple-carry and carry-save arrays for the multiplication operation $M \times Q = P$ for 4-bit operands.

						1	0	1	1	0	1	(45)	M
					X	1	1	1	1	1	1	(63)	Q
						1	0	1	1	0	1		A
					1	0	1	1	0	1			B
				1	0	1	1	0	1				C
			1	0	1	1	0	1					D
		1	0	1	1	0	1						E
	1	0	1	1	0	1							F
1	0	1	1	0	0	0	1	0	0	1	1	(2,835)	Product

Figure 6.17 A multiplication example used to illustrate carry-save addition as shown in Figure 6.18.

introduce the third summand bit products m_2q_2, m_1q_2, and m_0q_2. Now, two inputs of each full adder in the second row are fed by sum and carry outputs from the first row. The third input is used to introduce the bit products m_2q_3, m_1q_3, and m_0q_3 of the fourth summand. The high-order bit products m_3q_2 and m_3q_3 of the third and fourth summands are introduced into the remaining free inputs at the left end in the second and third rows. The saved carry bits and the sum bits from the second row are now added in the third row to produce the final product bits.

Delay through the carry-save array is somewhat less than delay through the ripple-carry array. This is because the S and C vector outputs from each row are produced in parallel in one full-adder delay. The amount of reduction in delay is considered in Problem 6.22.

A more significant reduction in delay can be achieved as follows. Consider the addition of many summands, as required in the multiplication of longer operands. We can group the summands in threes and perform carry-save addition on each of these groups in parallel to generate a set of S and C vectors in one full-adder delay. Next, we group all of the S and C vectors into threes, and perform carry-save addition on them, generating a further set of S and C vectors in one more full-adder delay. We continue with this process until there are only two vectors remaining. They can then be added in a ripple-carry or a carry-lookahead adder to produce the desired product.

Consider the example of adding the six shifted versions of the multiplicand for the case of multiplying two 6-bit unsigned numbers where all six bits of the multiplier are equal to 1. Such an example is shown in Figure 6.17. The six summands, $A, B, \ldots, F$ are added by carry-save addition in Figure 6.18. The "blue boxes" in these two figures indicate the same operand bits, and show how they are reduced to sum and carry bits in Figure 6.18 by carry-save addition. Three levels of carry-save addition are performed, as shown schematically in Figure 6.19. It is clear from this figure that the final two vectors S_4 and C_4 are available in three full-adder delays after the six input summands

```
                    1 0 1 1 0 1   M
                  x 1 1 1 1 1 1   Q
                  -------------
                    1 0 1 1 0 1   A
                  1 0 1 1 0 1     B
                1 0 1 1 0 1       C
            -------------------
        1 1 0 0 0 0 1 1   S1
      0 0 1 1 1 1 0 0     C1

      1 0 1 1 0 1         D
    1 0 1 1 0 1           E
  1 0 1 1 0 1             F
  -----------------
  1 1 0 0 0 0 1 1         S2
0 0 1 1 1 1 0 0           C2

        1 1 0 0 0 0 1 1   S1
      0 0 1 1 1 1 0 0     C1
  1 1 0 0 0 0 1 1         S2
  ---------------------
  1 1 0 1 0 1 0 0 0 1 1   S3
0 0 0 0 1 0 1 1 0 0 0     C3
0 0 1 1 1 1 0 0           C2
-----------------------
0 1 0 1 1 1 0 1 0 0 1 1   S4
+ 0 1 0 1 0 1 0 0 0 0 0   C4
-----------------------
1 0 1 1 0 0 0 1 0 0 1 1   Product
```

Figure 6.18 The multiplication example from Figure 6.17 performed using carry-save addition.

are applied to level 1. The final regular addition operation on S_4 and C_4, which produces the product, can be done with either a ripple-carry or a carry-lookahead adder.

Let us compute the total logic gate delay required to perform the 6×6 multiplication as shown in Figures 6.18 and 6.19. After one AND gate delay, used to select the summands based on the multiplier bits, all six summands are available as inputs to CSA level 1. The outputs S_4 and C_4 from the third CSA level are available 6 gate delays later, assuming two gate delays per CSA level. The final two vectors can be added in a further

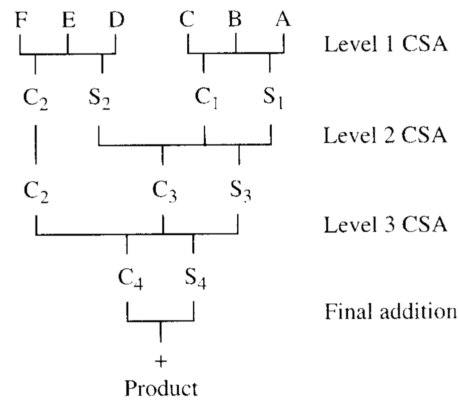


Figure 6.19 Schematic representation of the carry-save addition operations in Figure 6.18.

8 gate delays using a carry-lookahead adder of the form shown in Figure 6.5. The total gate delay is therefore 15. By comparison, the total gate delay in performing multiplication by using an $n \times n$ array of the type shown in Figure 6.6 is $6(n-1)-1$; so the 6×6 case has a gate delay of 29. This halving of delay is a result of using both carry-save addition of summands in parallel and carry-lookahead addition of the final two vectors.

When the number of summands is large, the time saved is proportionally much greater. For example, the addition of 32 numbers using the carry-save addition method, following the pattern shown in Figure 6.19, requires only 8 levels of CSA steps before the final Add operation. In general, it can be shown that approximately $1.7log_2k - 1.7$ levels of CSA steps are needed to reduce k summands to 2 vectors, which, when added, produce the desired sum. (See Problem 6.23.) A 64-bit carry-lookahead adder can be used to add the final 2 vectors. Total delay for 32×32 multiplication is calculated as follows: one gate delay for the initial AND gates, whose outputs produce the 32 summands; 16 gate delays for 8 levels of CSA steps; and 12 gate delays for the longest path (to s_{63}) through the 64-bit adder. The total delay is thus 29 gate delays. In comparison, an array multiplier for the 32×32 case requires 185 gate delays to generate the last bit of the product.

Some issues have been omitted in discussing the use of carry-save addition for speeding up the multiplication operation. First, when negative summands are involved, as they are for signed-operand multiplication using the Booth algorithm, it is necessary to accommodate sign-extension in the CSA logic. Full extension to the double-length product distance is not actually required. Only a few bits of extension at each CSA level are needed. Second, we have assumed that a $2n$-bit carry-lookahead adder is needed to add the final two S and C vectors for $n \times n$ multiplication. Somewhat fewer bits are actually involved in this final addition because some of the low-order product bits are determined earlier. But this is not a big factor; and the delay analysis that we have used is correct because the adder is not significantly shorter. Finally, we have used n summands for an $n \times n$ multiplication. But if bit-pair recoding of the multiplier is done, the number of summands is reduced to $n/2$. This reduces the number of CSA levels required from $1.7log_2n - 1.7$ to $1.7log_2n - 3.4$.

Summary of Fast Multiplication

We now summarize the techniques for high-speed multiplication. Bit-pair recoding of the multiplier, derived from the Booth algorithm, reduces the number of summands by a factor of 2. These summands can then be reduced to only 2 by using a relatively small number of carry-save addition steps. The final product can be generated by an addition operation that uses a carry-lookahead adder. All three of these techniques — bit-pair recoding of the multiplier, carry-save addition of the summands, and lookahead addition — have been used in various ways by the designers of high-performance processors to reduce the time needed to perform multiplication.

6.6 INTEGER DIVISION

In Section 6.4, we discussed positive-number multiplication by relating the way the multiplication operation is done manually to the way it is done in a logic circuit. We use the same approach here in discussing integer division. We discuss positive-number division in detail, and then make some general comments on the signed-operand case.

Figure 6.20 shows examples of decimal division and binary division of the same values. Consider the decimal version first. The 2 in the quotient is determined by the following reasoning: First, we try to divide 13 into 2, and it does not work. Next, we try to divide 13 into 27. We go through the trial exercise of multiplying 13 by 2 to get 26, and, knowing that $27 - 26 = 1$ is less than 13, we enter 2 as the quotient and perform the required subtraction. The next digit of the dividend, 4, is brought down, and we finish by deciding that 13 goes into 14 once, and the remainder is 1. We can discuss binary division in a similar way, with the simplification that the only possibilities for the quotient bits are 0 and 1.

A circuit that implements division by this longhand method operates as follows: It positions the divisor appropriately with respect to the dividend and performs a subtraction. If the remainder is zero or positive, a quotient bit of 1 is determined, the remainder is extended by another bit of the dividend, the divisor is repositioned, and another subtraction is performed. On the other hand, if the remainder is negative, a quotient bit of 0 is determined, the dividend is restored by adding back the divisor, and the divisor is repositioned for another subtraction.

```
      21                    10101
13 ) 274         1101 ) 100010010
     26                  1101
      14                  10000
      13                   1101
       1                    1110
                            1101
                               1
```

Figure 6.20 Longhand division examples.

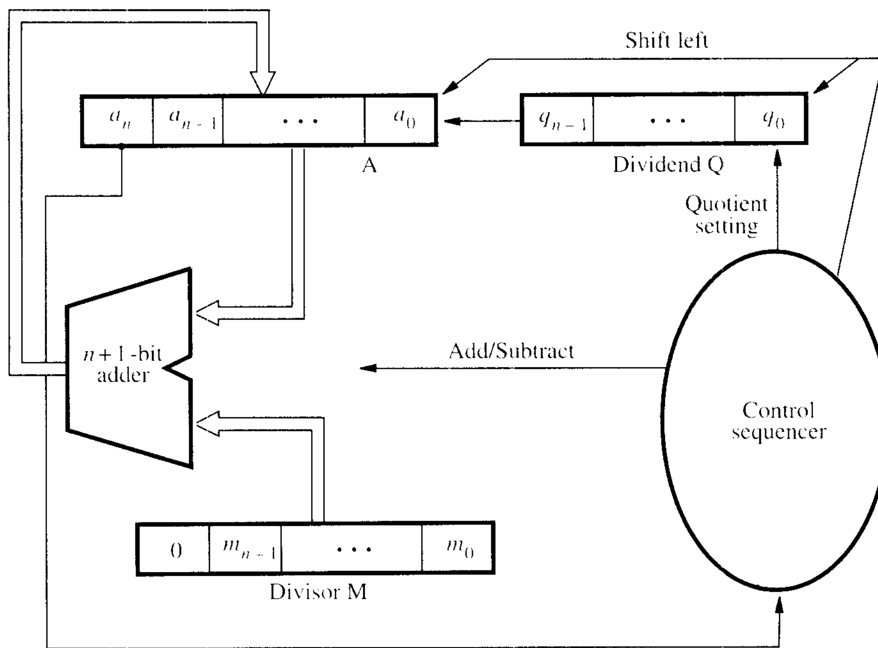


Figure 6.21 Circuit arrangement for binary division.

Restoring Division

Figure 6.21 shows a logic circuit arrangement that implements *restoring division*. Note its similarity to the structure for multiplication that was shown in Figure 6.7. An n-bit positive divisor is loaded into register M and an n-bit positive dividend is loaded into register Q at the start of the operation. Register A is set to 0. After the division is complete, the n-bit quotient is in register Q and the remainder is in register A. The required subtractions are facilitated by using 2's-complement arithmetic. The extra bit position at the left end of both A and M accommodates the sign bit during subtractions. The following algorithm performs restoring division.

Do the following n times:

1. Shift A and Q left one binary position.
2. Subtract M from A, and place the answer back in A.
3. If the sign of A is 1, set q_0 to 0 and add M back to A (that is, restore A); otherwise, set q_0 to 1.

Figure 6.22 shows a 4-bit example as it would be processed by the circuit in Figure 6.21.

Nonrestoring Division

The restoring-division algorithm can be improved by avoiding the need for restoring A after an unsuccessful subtraction. Subtraction is said to be unsuccessful if the result

```
          10
    11 ) 1000
          11
          --
          10

Initially   0 0 0 0 0      1 0 0 0        ⎫
            0 0 0 1 1                     ⎪
Shift       0 0 0 0 1      0 0 0 □        ⎪
Subtract    1 1 1 0 1                     ⎬ First cycle
Set q0     (1)1 1 1 0                     ⎪
Restore           1 1                     ⎪
            0 0 0 0 1      0 0 0 [0]      ⎭
Shift       0 0 0 1 0      0 0 [0] □      ⎫
Subtract    1 1 1 0 1                     ⎪
Set q0     (1)1 1 1 1                     ⎬ Second cycle
Restore           1 1                     ⎪
            0 0 0 1 0      0 0 [0][0]     ⎭
Shift       0 0 1 0 0      0 [0][0] □     ⎫
Subtract    1 1 1 0 1                     ⎬ Third cycle
Set q0     (0)0 0 0 1                     ⎪
Shift       0 0 0 1 0      0 [0][0][1]    ⎭
Subtract    1 1 1 0 1      [0][0][1] □    ⎫
Set q0     (1)1 1 1 1                     ⎬ Fourth cycle
Restore           1 1                     ⎪
            0 0 0 1 0      [0][0][1][0]   ⎭
            Remainder      Quotient
```

Figure 6.22 A restoring-division example.

is negative. Consider the sequence of operations that takes place after the subtraction operation in the preceding algorithm. If A is positive, we shift left and subtract M, that is, we perform 2A − M. If A is negative, we restore it by performing A + M, and then we shift it left and subtract M. This is equivalent to performing 2A + M. The q_0 bit is appropriately set to 0 or 1 after the correct operation has been performed. We can summarize this in the following algorithm for *nonrestoring division*.

Step 1: Do the following n times:

1. If the sign of A is 0, shift A and Q left one bit position and subtract M from A; otherwise, shift A and Q left and add M to A.
2. Now, if the sign of A is 0, set q_0 to 1; otherwise, set q_0 to 0.

Step 2: If the sign of A is 1, add M to A.

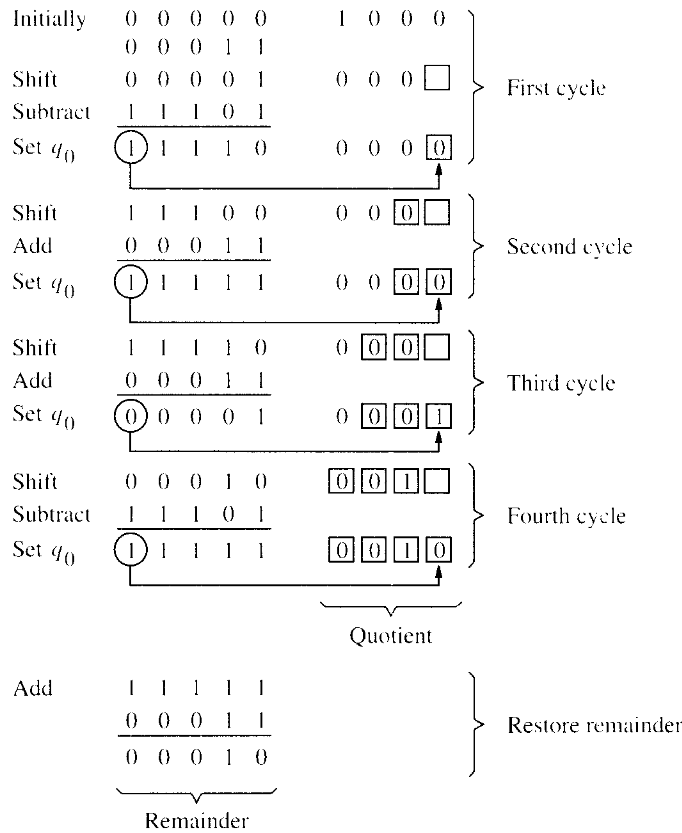


Figure 6.23 A nonrestoring-division example.

Step 2 is needed to leave the proper positive remainder in A at the end of the n cycles of Step 1. The logic circuitry in Figure 6.21 can also be used to perform this algorithm. Note that the Restore operations are no longer needed, and that exactly one Add or Subtract operation is performed per cycle. Figure 6.23 shows how the division example in Figure 6.22 is executed by the nonrestoring-division algorithm.

There are no simple algorithms for directly performing division on signed operands that are comparable to the algorithms for signed multiplication. In division, the operands can be preprocessed to transform them into positive values. After using one of the algorithms just discussed, the results are transformed to the correct signed values, as necessary.

6.7 FLOATING-POINT NUMBERS AND OPERATIONS

Until now, we have dealt exclusively with fixed-point numbers and have considered them as integers, that is, as having an implied binary point at the right end of the number. It is also possible to assume that the binary point is just to the right of the sign bit, thus

representing a fraction. In the 2's-complement system, the signed value F, represented by the n-bit binary fraction

$$B = b_0.b_{-1}b_{-2}\ldots b_{-(n-1)}$$

is given by

$$F(B) = -b_0 \times 2^0 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \cdots + b_{-(n-1)} \times 2^{-(n-1)}$$

where the range of F is

$$-1 \leq F \leq 1 - 2^{-(n-1)}$$

Consider the range of values representable in a 32-bit, signed, fixed-point format. Interpreted as integers, the value range is approximately 0 to $\pm 2.15 \times 10^9$. If we consider them to be fractions, the range is approximately $\pm 4.55 \times 10^{-10}$ to ± 1. Neither of these ranges is sufficient for scientific calculations, which might involve parameters like Avogadro's number (6.0247×10^{23} mole^{-1}) or Planck's constant (6.6254×10^{-27}erg·s). Hence, we need to easily accommodate both very large integers and very small fractions. To do this, a computer must be able to represent numbers and operate on them in such a way that the position of the binary point is variable and is automatically adjusted as computation proceeds. In such a case, the binary point is said to float, and the numbers are called *floating-point numbers*. This distinguishes them from fixed-point numbers, whose binary point is always in the same position.

Because the position of the binary point in a floating-point number is variable, it must be given explicitly in the floating-point representation. For example, in the familiar decimal scientific notation, numbers may be written as 6.0247×10^{23}, 6.6254×10^{-27}, -1.0341×10^2, -7.3000×10^{-14}, and so on. These numbers are said to be given to five *significant digits*. The *scale factors* (10^{23}, 10^{-27}, and so on) indicate the position of the decimal point with respect to the significant digits. By convention, when the decimal point is placed to the right of the first (nonzero) significant digit, the number is said to be *normalized*. Note that the base, 10, in the scale factor is fixed and does not need to appear explicitly in the machine representation of a floating-point number. The sign, the significant digits, and the exponent in the scale factor constitute the representation. We are thus motivated to define a floating-point number representation as one in which a number is represented by its sign, a string of significant digits, commonly called the *mantissa*, and an exponent to an implied base for the scale factor.

6.7.1 IEEE STANDARD FOR FLOATING-POINT NUMBERS

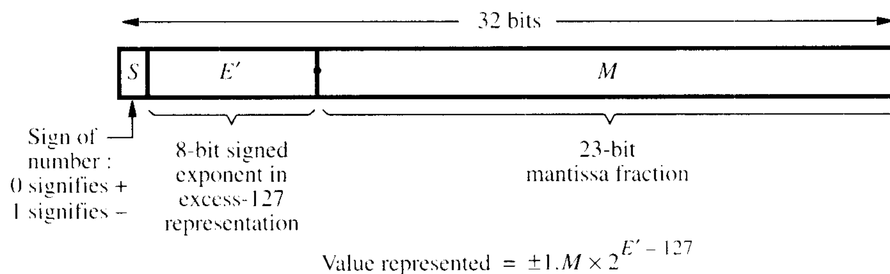
We start with a general form and size for floating-point numbers in the decimal system, and then relate this form to a comparable binary representation. A useful form is

$$\pm X_1.X_2X_3X_4X_5X_6X_7 \times 10^{\pm Y_1Y_2}$$

where X_i and Y_i are decimal digits. Both the number of significant digits (7) and the exponent range (± 99) are sufficient for a wide range of scientific calculations. It is possible to approximate this mantissa precision and scale factor range in a binary representation that occupies 32 bits, which is a standard computer word length. A 24-bit

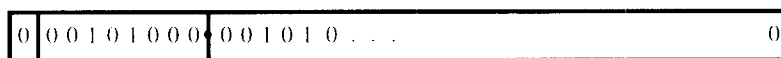
mantissa can approximately represent a 7-digit decimal number, and an 8-bit exponent to an implied base of 2 provides a scale factor with a reasonable range. One bit is needed for the sign of the number. Since the leading nonzero bit of a normalized binary mantissa must be a 1, it does not have to be included explicitly in the representation. Therefore, a total of 32 bits is needed.

This standard for representing floating-point numbers in 32 bits has been developed and specified in detail by the Institute of Electrical and Electronics Engineers (IEEE) [1]. The standard describes both the representation and the way in which the four basic arithmetic operations are to be performed. The 32-bit representation is given in Figure 6.24*a*. The sign of the number is given in the first bit, followed by a representation for the exponent (to the base 2) of the scale factor. Instead of the signed exponent, E, the value actually stored in the exponent field is an unsigned integer $E' = E + 127$.



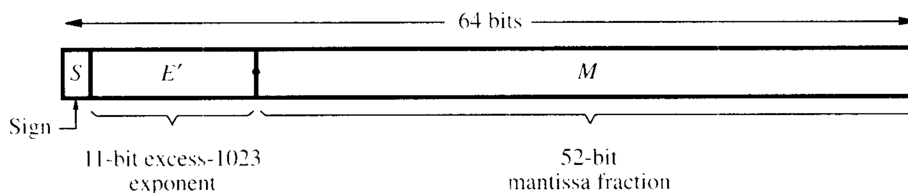
Value represented = $\pm 1.M \times 2^{E'-127}$

(a) Single precision



Value represented = $1.001010\ldots0 \times 2^{-87}$

(b) Example of a single-precision number



Value represented = $\pm 1.M \times 2^{E'-1023}$

(c) Double precision

Figure 6.24 IEEE standard floating-point formats.

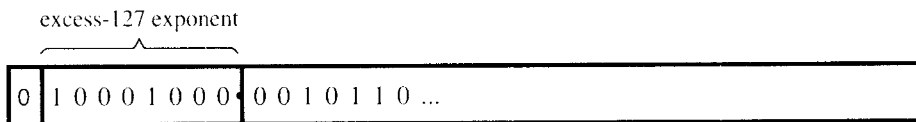
This is called the excess-127 format. Thus, E' is in the range $0 \leq E' \leq 255$. The end values of this range, 0 and 255, are used to represent special values, as described below. Therefore, the range of E' for normal values is $1 \leq E' \leq 254$. This means that the actual exponent, E, is in the range $-126 \leq E \leq 127$. The *excess-x* representation for exponents enables efficient comparison of the relative sizes of two floating-point numbers. (See Problem 6.27.)

The last 23 bits represent the mantissa. Since binary normalization is used, the most significant bit of the mantissa is always equal to 1. This bit is not explicitly represented; it is assumed to be to the immediate left of the binary point. Hence, the 23 bits stored in the M field actually represent the fractional part of the mantissa, that is, the bits to the right of the binary point. An example of a single-precision floating-point number is shown in Figure 6.24*b*.

The 32-bit standard representation in Figure 6.24*a* is called a *single-precision* representation because it occupies a single 32-bit word. The scale factor has a range of 2^{-126} to 2^{+127}, which is approximately equal to $10^{\pm 38}$. The 24-bit mantissa provides approximately the same precision as a 7-digit decimal value. To provide more precision and range for floating-point numbers, the IEEE standard also specifies a *double-precision* format, as shown in Figure 6.24*c*. The double-precision format has increased exponent and mantissa ranges. The 11-bit excess-1023 exponent E' has the range $1 \leq E' \leq 2046$ for normal values, with 0 and 2047 used to indicate special values, as before. Thus, the actual exponent E is in the range $-1022 \leq E \leq 1023$, providing scale factors of 2^{-1022} to 2^{1023} (approximately $10^{\pm 308}$). The 53-bit mantissa provides a precision equivalent to about 16 decimal digits.

A computer must provide at least single-precision representation to conform to the IEEE standard. Double-precision representation is optional. The standard also specifies certain optional extended versions of both of these formats. The extended versions are intended to provide increased precision and increased exponent range for the representation of intermediate values in a sequence of calculations. For example, the dot product of two vectors of numbers can be computed by accumulating the sum of products in extended precision. The inputs are given in a standard precision, either single or double, and the answer is truncated to the same precision. The use of extended formats helps to reduce the size of the accumulated round-off error in a sequence of calculations. Extended formats also enhance the accuracy of evaluation of elementary functions such as sine, cosine, and so on. In addition to requiring the four basic arithmetic operations, the standard requires that the operations of remainder, square root, and conversion between binary and decimal representations be provided.

We note two basic aspects of operating with floating-point numbers. First, if a number is not normalized, it can always be put in normalized form by shifting the fraction and adjusting the exponent. Figure 6.25 shows an unnormalized value, $0.0010110\ldots \times 2^9$, and its normalized version, $1.0110\ldots \times 2^6$. Since the scale factor is in the form 2^i, shifting the mantissa right or left by one bit position is compensated by an increase or a decrease of 1 in the exponent, respectively. Second, as computations proceed, a number that does not fall in the representable range of normal numbers might be generated. In single precision, this means that its normalized representation requires an exponent less than -126 or greater than $+127$. In the first case, we say that *underflow* has occurred, and in the second case, we say that *overflow* has occurred. Both underflow and overflow are arithmetic exceptions that are considered below.



(There is no implicit 1 to the left of the binary point.)

$$\text{Value represented} = +0.0010110\ldots \times 2^9$$

(a) Unnormalized value

$$\text{Value represented} = +1.0110\ldots \times 2^6$$

(b) Normalized version

Figure 6.25 Floating-point normalization in IEEE single-precision format.

Special Values

The end values 0 and 255 of the excess-127 exponent E' are used to represent special values. When $E' = 0$ and the mantissa fraction M is zero, the value exact 0 is represented. When $E' = 255$ and $M = 0$, the value ∞ is represented, where ∞ is the result of dividing a normal number by zero. The sign bit is still part of these representations, so there are ± 0 and $\pm\infty$ representations.

When $E' = 0$ and $M \neq 0$, *denormal* numbers are represented. Their value is $\pm 0.M \times 2^{-126}$. Therefore, they are smaller than the smallest normal number. There is no implied one to the left of the binary point, and M is any nonzero 23-bit fraction. The purpose of introducing denormal numbers is to allow for *gradual underflow*, providing an extension of the range of normal representable numbers that is useful in dealing with very small numbers in certain situations. When $E' = 255$ and $M \neq 0$, the value represented is called *Not a Number* (NaN). A NaN is the result of performing an invalid operation such as $0/0$ or $\sqrt{-1}$.

Exceptions

In conforming to the IEEE Standard, a processor must set *exception* flags if any of the following occur in performing operations: underflow, overflow, divide by zero, inexact, invalid. We have already mentioned the first three. *Inexact* is the name for a result that requires rounding in order to be represented in one of the normal formats. An *invalid* exception occurs if operations such as $0/0$ or $\sqrt{-1}$ are attempted. When exceptions occur, the results are set to special values.

If interrupts are enabled for any of the exception flags, system or user-defined routines are entered when the associated exception occurs. Alternatively, the application

program can test for the occurrence of exceptions, as necessary, and decide how to proceed.

A more detailed discussion of the floating-point issues raised here and in the next two sections is given in Appendix A of Hennessy and Patterson [2].

6.7.2 ARITHMETIC OPERATIONS ON FLOATING-POINT NUMBERS

In this section, we outline the general procedures for addition, subtraction, multiplication, and division of floating-point numbers. The rules we give apply to the single-precision IEEE standard format. These rules specify only the major steps needed to perform the four operations; for example, the possibility that overflow or underflow might occur is not discussed. Furthermore, intermediate results for both mantissas and exponents might require more than 24 and 8 bits, respectively. These and other aspects of the operations must be carefully considered in designing an arithmetic unit that meets the standard. Although we do not provide full details in specifying the rules, we consider some aspects of implementation, including rounding, in later sections.

If their exponents differ, the mantissas of floating-point numbers must be shifted with respect to each other before they are added or subtracted. Consider a decimal example in which we wish to add 2.9400×10^2 to 4.3100×10^4. We rewrite 2.9400×10^2 as 0.0294×10^4 and then perform addition of the mantissas to get 4.3394×10^4. The rule for addition and subtraction can be stated as follows:

Add/Subtract Rule

1. Choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponents.
2. Set the exponent of the result equal to the larger exponent.
3. Perform addition/subtraction on the mantissas and determine the sign of the result.
4. Normalize the resulting value, if necessary.

Multiplication and division are somewhat easier than addition and subtraction, in that no alignment of mantissas is needed.

Multiply Rule

1. Add the exponents and subtract 127.
2. Multiply the mantissas and determine the sign of the result.
3. Normalize the resulting value, if necessary.

Divide Rule

1. Subtract the exponents and add 127.
2. Divide the mantissas and determine the sign of the result.
3. Normalize the resulting value, if necessary.

The addition or subtraction of 127 in the multiply and divide rules results from using the excess-127 notation for exponents.

6.7.3 GUARD BITS AND TRUNCATION

Let us consider some important aspects of implementing the steps in the preceding algorithms. Although the mantissas of initial operands and final results are limited to 24 bits, including the implicit leading 1, it is important to retain extra bits, often called *guard* bits, during the intermediate steps. This yields maximum accuracy in the final results.

Removing guard bits in generating a final result requires that the extended mantissa be *truncated* to create a 24-bit number that approximates the longer version. This operation also arises in other situations, for instance, in converting from decimal to binary numbers. We should mention that the general term rounding is also used for the truncation operation, but we will use a more restrictive definition of rounding as one of the forms of truncation.

There are several ways to truncate. The simplest way is to remove the guard bits and make no changes in the retained bits. This is called *chopping*. Suppose we want to truncate a fraction from six to three bits by this method. All fractions in the range $0.b_{-1}b_{-2}b_{-3}000$ to $0.b_{-1}b_{-2}b_{-3}111$ are truncated to $0.b_{-1}b_{-2}b_{-3}$. The error in the 3-bit result ranges from 0 to 0.000111. In other words, the error in chopping ranges from 0 to almost 1 in the least significant position of the retained bits. In our example, this is the b_{-3} position. The result of chopping is a *biased* approximation because the error range is not symmetrical about 0.

The next simplest method of truncation is *Von Neumann rounding*. If the bits to be removed are all 0s, they are simply dropped, with no changes to the retained bits. However, if any of the bits to be removed are 1, the least significant bit of the retained bits is set to 1. In our 6-bit to 3-bit truncation example, all 6-bit fractions with $b_{-4}b_{-5}b_{-6}$ not equal to 000 are truncated to $0.b_{-1}b_{-2}1$. The error in this truncation method ranges between -1 and $+1$ in the LSB position of the retained bits. Although the range of error is larger with this technique than it is with chopping, the maximum magnitude is the same, and the approximation is *unbiased* because the error range is symmetrical about 0.

Unbiased approximations are advantageous if many operands and operations are involved in generating a result, because positive errors tend to offset negative errors as the computation proceeds. Statistically, we can expect the results of a complex computation to have a high probability of accuracy.

The third truncation method is a *rounding* procedure. Rounding achieves the closest approximation to the number being truncated and is an unbiased technique. The procedure is as follows: A 1 is added to the LSB position of the bits to be retained if there is a 1 in the MSB position of the bits being removed. Thus, $0.b_{-1}b_{-2}b_{-3}1\ldots$ is rounded to $0.b_{-1}b_{-2}b_{-3} + 0.001$, and $0.b_{-1}b_{-2}b_{-3}0\ldots$ is rounded to $0.b_{-1}b_{-2}b_{-3}$. This provides the desired approximation, except for the case in which the bits to be removed are $10\ldots0$. This is a tie situation; the longer value is halfway between the two closest truncated representations. To break the tie in an unbiased way, one possibility is to choose the retained bits to be the nearest even number. In terms of our 6-bit example, the value $0.b_{-1}b_{-2}0100$ is truncated to the value $0.b_{-1}b_{-2}0$, and $0.b_{-1}b_{-2}1100$ is truncated to $0.b_{-1}b_{-2}1 + 0.001$. The descriptive phrase "round to the nearest number or nearest even number in case of a tie" is sometimes used to refer to this truncation technique.

The error range is approximately $-\frac{1}{2}$ to $+\frac{1}{2}$ in the LSB position of the retained bits. Clearly, this is the best method. However, it is also the most difficult to implement because it requires an addition operation and a possible renormalization. This rounding technique is the default mode for truncation specified in the IEEE floating-point standard. The standard also specifies other truncation methods, referring to all of them as rounding modes.

This discussion of errors that are introduced when guard bits are removed by truncation has treated the case of a single truncation operation. When a long series of calculations involving floating-point numbers is performed, the analysis that determines error ranges or bounds for the final results can be a complicated study. We do not discuss this aspect of numerical computation further, except to make a few comments on the way that guard bits and rounding are handled in the IEEE floating-point standard.

Results of single operations must be computed to be accurate within half a unit in the LSB position. In general, this requires that rounding be used as the truncation method. Implementing this rounding scheme requires only three guard bits to be carried along during the intermediate steps in performing the operations described. The first two of these bits are the two most significant bits of the section of the mantissa to be removed. The third bit is the logical OR of all bits beyond these first two bits in the full representation of the mantissa. This bit is relatively easy to maintain during the intermediate steps of the operations to be performed. It should be initialized to 0. If a 1 is shifted out through this position, the bit becomes 1 and retains that value; hence, it is usually called the *sticky bit*.

6.7.4 IMPLEMENTING FLOATING-POINT OPERATIONS

The hardware implementation of floating-point operations involves a considerable amount of logic circuitry. These operations can also be implemented by software routines. In either case, the computer must be able to convert input and output from and to the user's decimal representation of numbers. In most general-purpose processors, floating-point operations are available at the machine-instruction level, implemented in hardware.

An example of the implementation of floating-point operations is shown in Figure 6.26. This is a block diagram of a hardware implementation for the addition and subtraction of 32-bit floating-point operands that have the format shown in Figure 6.24*a*. Following the Add/Subtract rule given in Section 6.7.2, we see that the first step is to compare exponents to determine how far to shift the mantissa of the number with the smaller exponent. The shift-count value, n, is determined by the 8-bit subtractor circuit in the upper left corner of the figure. The magnitude of the difference $E'_A - E'_B$, or n, is sent to the SHIFTER unit. If n is larger than the number of significant bits of the operands, then the answer is essentially the larger operand (except for guard and sticky-bit considerations in rounding), and shortcuts can be taken in deriving the result. We do not explore this in detail.

The sign of the difference that results from comparing exponents determines which mantissa is to be shifted. Therefore, in step 1, the sign is sent to the SWAP network in

Figure 6.26 Floating-point addition-subtraction unit.

the upper right corner of Figure 6.26. If the sign is 0, then $E'_A \geq E'_B$ and the mantissas M_A and M_B are sent straight through the SWAP network. This results in M_B being sent to the SHIFTER, to be shifted n positions to the right. The other mantissa, M_A, is sent directly to the mantissa adder/subtractor. If the sign is 1, then $E'_A < E'_B$ and the mantissas are swapped before they are sent to the SHIFTER.

Step 2 is performed by the two-way multiplexer, MUX, near the bottom left corner of the figure. The exponent of the result, E', is tentatively determined as E'_A if $E'_A \geq E'_B$, or E'_B if $E'_A < E'_B$, based on the sign of the difference resulting from comparing exponents in step 1.

Step 3 involves the major component, the mantissa adder/subtractor in the middle of the figure. The CONTROL logic determines whether the mantissas are to be added or subtracted. This is decided by the signs of the operands (S_A and S_B) and the operation (Add or Subtract) that is to be performed on the operands. The CONTROL logic also determines the sign of the result, S_R. For example, if A is negative ($S_A = 1$), B is positive ($S_B = 0$), and the operation is $A - B$, then the mantissas are added and the sign of the result is negative ($S_R = 1$). On the other hand, if A and B are both positive and the operation is $A - B$, then the mantissas are subtracted. The sign of the result, S_R, now depends on the mantissa subtraction operation. For instance, if $E'_A > E'_B$, then $M_A -$ (shifted M_B) is positive and the result is positive. But if $E'_B > E'_A$, then $M_B -$ (shifted M_A) is positive and the result is negative. This example shows that the sign from the exponent comparison is also required as an input to the CONTROL network. When $E'_A = E'_B$ and the mantissas are subtracted, the sign of the mantissa adder/subtractor output determines the sign of the result. The reader should now be able to construct the complete truth table for the CONTROL network.

Step 4 of the Add/Subtract rule consists of normalizing the result of step 3, mantissa M. The number of leading zeros in M determines the number of bit shifts, X, to be applied to M. The normalized value is truncated to generate the 24-bit mantissa, M_R, of the result. The value X is also subtracted from the tentative result exponent E' to generate the true result exponent, E'_R. Note that only a single right shift might be needed to normalize the result. This would be the case if two mantissas of the form $1.xx\ldots$ were added. The vector M would then have the form $1x.xx\ldots$. This would correspond to an X value of -1 in the figure.

We have not given any details on the guard bits that must be carried along with intermediate mantissa values. In the IEEE standard, only a few bits are needed, as discussed earlier, to generate the 24-bit normalized mantissa of the result.

Let us consider the actual hardware that is needed to implement the blocks in Figure 6.26. The two 8-bit subtractors and the mantissa adder/subtractor can be implemented by combinational logic, as discussed earlier in this chapter. Because their outputs must be in sign-and-magnitude form, we must modify some of our earlier discussions. A combination of 1's-complement arithmetic and sign-and-magnitude representation is often used. Considerable flexibility is allowed in implementing the SHIFTER and the output normalization operation. If a design with a modest logic gate count is required, the operations can be implemented with shift registers. However, they can also be built as combinational logic units for high-performance, but in that case, a significant number of logic gates is needed. In high-performance processors, a significant portion of the chip area is assigned to floating-point operations.

6.8 CONCLUDING REMARKS

Computer arithmetic poses several interesting logic design problems. This chapter discussed some of the techniques that have proven useful in designing binary arithmetic units. The carry-lookahead technique is one of the major ideas in high-performance adder design. In the design of fast multipliers, bit-pair recoding of the multiplier, derived from the Booth algorithm, reduces the number of summands that must be added to generate the product. Carry-save addition substantially reduces the time needed to add the summands.

The IEEE floating-point number representation standard was described, and a set of rules for performing the four standard operations was given. As an example of the circuit complexity required to implement floating-point operations, the block diagram of an addition/subtraction unit was described.

PROBLEMS

6.1 Consider the binary numbers in the following addition and subtraction problems to be signed, 6-bit values in the 2's-complement representation. Perform the operations indicated, specify whether or not arithmetic overflow occurs, and check your answers by converting operands and results to decimal sign-and-magnitude representation.

$$\begin{array}{rrr}
010110 & 101011 & 111111 \\
\underline{+001001} & \underline{+100101} & \underline{+000111} \\
\\
011001 & 110111 & 010101 \\
\underline{+010000} & \underline{+111001} & \underline{+101011} \\
\\
010110 & 111110 & 100001 \\
\underline{-011111} & \underline{-100101} & \underline{-011101} \\
\\
111111 & 000111 & 011010 \\
\underline{-000111} & \underline{-111000} & \underline{-100010}
\end{array}$$

6.2 Signed binary fractions in 2's-complement representation are discussed at the beginning of Section 6.7.

(*a*) Express the decimal values 0.5, −0.123, −0.75, and −0.1 as signed 6-bit fractions. (See Appendix E for decimal-to-binary fraction conversion.)

(*b*) What is the maximum representation error, e, involved in using only 5 significant bits after the binary point?

(*c*) Calculate the number of bits needed after the binary point so that

(*a*) $e < \frac{1}{10}$
(*b*) $e < \frac{1}{100}$
(*c*) $e < \frac{1}{1000}$
(*d*) $e < \frac{1}{10^6}$

6.3 The 1's-complement and 2's-complement binary representation methods are special cases of the $(b-1)$'s-complement and b's-complement representation techniques in base b number systems. For example, consider the decimal system. The sign-and-magnitude values +526, −526, +70, and −70 have 4-digit signed-number representations in each of the two complement systems, as shown in Figure P6.1. The 9's-complement is formed by taking the complement of each digit position with respect to 9. The 10's-complement is formed by adding 1 to the 9's-complement. In each of the latter two representations, the leftmost digit is zero for a positive number and 9 for a negative number.

Representation	Examples			
Sign and magnitude	+526	−526	+70	−70
9's complement	0526	9473	0070	9929
10's complement	0526	9474	0070	9930

Figure P6.1 Signed numbers in base 10 used in Problem 6.3.

Now consider the base-3 (ternary) system, in which the unsigned, 5-digit number $t_4t_3t_2t_1t_0$ has the value $t_4 \times 3^4 + t_3 \times 3^3 + t_2 \times 3^2 + t_1 \times 3^1 + t_0 \times 3^0$, with $0 \leq t_i \leq 2$. Express the ternary sign-and-magnitude numbers +11011, −10222, +2120, −1212, +10, and −201 as 6-digit, signed, ternary numbers in the 3's-complement system.

6.4 Represent each of the decimal values 56, −37, 122, and −123 as signed 6-digit numbers in the 3's-complement ternary format, perform addition and subtraction on them in all possible pairwise combinations, and state whether or not arithmetic overflow occurs for each operation performed. (See Problem 6.3 for a definition of the ternary number system, and use a technique analogous to that given in Appendix E for decimal-to-ternary integer conversion.)

6.5 A half adder is a combinational logic circuit that has two inputs, x and y, and two outputs, s and c, that are the sum and carry-out, respectively, resulting from the binary addition of x and y.

(*a*) Design a half adder as a two-level AND-OR circuit.

(*b*) Show how to implement a full adder, as shown in Figure 6.2*a*, by using two half adders and external logic gates, as necessary.

(*c*) Compare the longest logic delay path through the network derived in Part (*b*) to that of the logic delay of the adder network shown in Figure 6.2*a*.

6.6 Write a 68000 or IA-32 program to transform a 16-bit positive binary number into a 5-digit decimal number in which each digit of the number is coded in the binary-coded decimal (BCD) code. These BCD digit codes are to occupy the low-order 4 bits of

five successive byte locations in the main memory. Use the conversion technique based on successive division by 10. This method is analogous to successive division by 2 when converting decimal-to-binary, as discussed in Appendix E. Consult Appendix C (68000) or D (IA-32) for the format and operation of the Divide instruction.

6.7 Assume that four BCD digits, representing a decimal integer in the range 0 to 9999, are packed into the lower half of a 32-bit memory location DECIMAL. Write an ARM, 68000, or IA-32 subroutine to convert the decimal integer stored at DECIMAL into binary representation and to store it in the memory location BINARY.

6.8 A modulo 10 adder is needed for adding BCD digits. Modulo 10 addition of two BCD digits, $A = A_3A_2A_1A_0$ and $B = B_3B_2B_1B_0$, can be achieved as follows: Add A to B (binary addition). Then, if the result is an illegal code that is greater than or equal to 10_{10}, add 6_{10}. (Ignore overflow from this addition.)

(*a*) When is the output carry equal to 1?

(*b*) Show that this algorithm gives correct results for

$$(1)\ A = 0101 \quad \text{and} \quad B = 0110$$
$$(2)\ A = 0011 \quad \text{and} \quad B = 0100$$

(*c*) Design a BCD digit adder using a 4-bit binary adder and external logic gates as needed. The inputs are $A_3A_2A_1A_0$, $B_3B_2B_1B_0$, and a carry-in. The outputs are the sum digit $S_3S_2S_1S_0$ and the carry-out. A cascade of such blocks can form a ripple-carry BCD adder.

6.9 Show that the logic expression $c_n \oplus c_{n-1}$ is a correct indicator of overflow in the addition of 2's-complement integers, by using an appropriate truth table.

6.10 (*a*) Design a 64-bit adder that uses four of the 16-bit carry-lookahead adders shown in Figure 6.5 along with additional logic to generate c_{16}, c_{32}, c_{48}, and c_{64}, from c_0 and the G_i^{II} and P_i^{II} variables shown in this figure. What is the relationship of the additional logic to the logic inside each lookahead circuit in the figure?

(*b*) Show that the delay through the 64-bit adder is 12 gate delays for s_{63} and 7 gate delays for c_{64}, as claimed at the end of Section 6.2.1.

(*c*) Compare the gate delays to produce s_{31} and c_{32} in the 64-bit adder of part (*a*) to the gate delays for the same variables in the 32-bit adder built from a cascade of two 16-bit adders, as discussed in Section 6.2.1.

6.11 (*a*) How many logic gates are needed to build the 4-bit carry-lookahead adder shown in Figure 6.4?

(*b*) Use appropriate parts of the result from Part (*a*) to calculate how many logic gates are needed to build the 16-bit carry-lookahead adder shown in Figure 6.5.

6.12 Show that the worst case delay through an $n \times n$ array of the type shown in Figure 6.6*b* is $6(n-1)-1$ gate delays, as claimed in Section 6.3.

6.13 Using manual methods, perform the operations $A \times B$ and $A \div B$ on the 5-bit unsigned numbers $A = 10101$ and $B = 00101$.

6.14 Show how the multiplication and division operations in Problem 6.13 would be performed by the hardware in Figures 6.7*a* and 6.21, respectively, by constructing charts similar to those in Figures 6.7*b* and 6.23.

6.15 Write an ARM, 68000, or IA-32 program for the multiplication of two 32-bit unsigned numbers that is patterned after the technique used in Figure 6.7. Assume that the multiplier and multiplicand are in registers R_2 and R_3, respectively. The product is to be developed in registers R_1 (high-order half) and R_2 (low-order half). (*Hint:* Use a combination of Shift and Rotate operations for a double-register shift.)

6.16 Write an ARM, 68000, or IA-32 program for integer division based on the nonrestoring-division algorithm. Assume that both operands are positive, that is, the leftmost bit is zero for both the dividend and the divisor.

6.17 Multiply each of the following pairs of signed 2's-complement numbers using the Booth algorithm. In each case, assume that A is the multiplicand and B is the multiplier.

(*a*) $A = 010111$ and $B = 110110$
(*b*) $A = 110011$ and $B = 101100$
(*c*) $A = 110101$ and $B = 011011$
(*d*) $A = 001111$ and $B = 001111$

6.18 Repeat Problem 6.17 using bit-pairing of the multipliers.

6.19 Indicate generally how to modify the circuit diagram in Figure 6.7*a* to implement multiplication of signed, 2's-complement, n-bit numbers using the Booth algorithm, by clearly specifying inputs and outputs for the Control sequencer and any other changes needed around the adder and A register.

6.20 If the product of two, n-bit, signed numbers in the 2's-complement representation can be represented in n bits, the manual multiplication algorithm shown in Figure 6.6*a* can be used directly, treating the sign bits the same as the other bits. Try this on each of the following pairs of 4-bit signed numbers:

(*a*) Multiplicand = 1110 and Multiplier = 1101
(*b*) Multiplicand = 0010 and Multiplier = 1110

Why does this work correctly?

6.21 An integer arithmetic unit that can perform addition and multiplication of 16-bit unsigned numbers is to be used to multiply two 32-bit unsigned numbers. All operands, intermediate results, and final results are held in 16-bit registers labeled R_0 through R_{15}. The hardware multiplier multiplies the contents of R_i (multiplicand) by R_j (multiplier) and stores the double-length 32-bit product in registers R_j and R_{j+1}, with the low-order half in R_j. When $j = i - 1$, the product overwrites both operands. The hardware adder adds the contents of R_i and R_j and puts the result in R_j. The input carry to an Add operation is 0, and the input carry to an Add-with-carry operation is the contents of a carry flag C. The output carry from the adder is always stored in C.

Specify the steps of a procedure for multiplying two 32-bit operands in registers R_1, R_0, and R_3, R_2, high-order halves first, leaving the 64-bit product in registers

R_{15}, R_{14}, R_{13}, and R_{12}. Any of the registers R_{11} through R_4 may be used for intermediate values, if necessary. Each step in the procedure can be a multiplication, or an addition, or a register transfer operation.

6.22 (*a*) Calculate the delay, in terms of logic gate delays, for the product bit p_7 in each of the arrays in Figure 6.16. Assume that each output from a full adder is available two gate delays after the inputs are available. Include the AND gate delay to generate all $m_i q_j$ products at the beginning.

(*b*) The delay for the extension of Figure 6.16*a* to the $n \times n$ case has been stated as $6(n - 1) - 1$ in Section 6.4. Develop a similar expression for the extension of Figure 6.16*b* to the $n \times n$ case.

6.23 Develop the derivation for the formula $1.7log_2k - 1.7$ for the number of carry-save addition steps needed to reduce k summands to two vectors. (This formula is stated without derivation in Section 6.5.2.)

6.24 (*a*) How many CSA levels are needed to reduce 16 summands to 2 using a pattern similar to that shown in Figure 6.19?

(*b*) Draw the pattern for reducing 32 summands to 2 to prove that the claim of 8 levels in Section 6.5.2 is correct.

(*c*) Compare the exact answers in Parts (*a*) and (*b*) to the results derived from the approximation $1.7log_2k - 1.7$.

6.25 In Section 6.7, we used the practical-sized 32-bit IEEE standard format for floating-point numbers. Here, we use a shortened format that retains all the pertinent concepts but is manageable for working through numerical exercises. Consider that floating-point numbers are represented in a 12-bit format as shown in Figure P6.2. The scale factor has an implied base of 2 and a 5-bit, excess-15 exponent, with the two end values of 0 and 31 used to signify exact 0 and infinity, respectively. The 6-bit mantissa is normalized as in the IEEE format, with an implied 1 to the left of the binary point.

(*a*) Represent the numbers $+1.7$, -0.012, $+19$, and $\frac{1}{8}$ in this format.

(*b*) What are the smallest and largest numbers representable in this format?

(*c*) How does the range calculated in Part (*b*) compare to the ranges of a 12-bit signed integer and a 12-bit signed fraction?

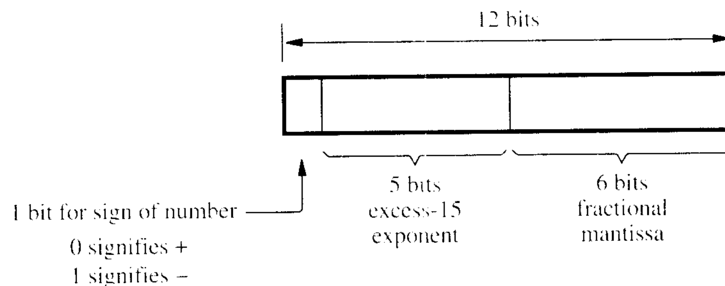


Figure P6.2 Floating-point format used in Problem 6.25.

(*d*) Perform Add, Subtract, Multiply, and Divide operations on the operands

$A =$	0	10001	011011

$B =$	1	01111	101010

6.26 Consider a 16-bit, floating-point number in a format similar to that discussed in Problem 6.25, with a 6-bit exponent and a 9-bit normalized fractional mantissa. The base of the scale factor is 2 and the exponent is represented in excess-31 format.

(*a*) Add the numbers A and B, formatted as follows:

$A =$	0	100001	111111110

$B =$	0	011111	001010101

Give the answer in normalized form. Remember that an implicit 1 is to the left of the binary point but is not included in the A and B formats. Use rounding when producing the final normalized 9-bit mantissa.

(*b*) Using decimal numbers w, x, y, and z, express the magnitude of the largest and smallest (nonzero) values representable in the preceding normalized floating-point format. Use the following form:

$$\text{Largest} = w \times 2^{x}$$

$$\text{Smallest} = y \times 2^{-z}$$

6.27 How does the excess-x representation for exponents of the scale factor in the floating-point number representation of Figure 6.24*a* facilitate the comparison of the relative sizes of two floating-point numbers? (*Hint:* Assume that a combinational logic network that compares the relative sizes of two, 32-bit, unsigned integers is available. Use this network, along with external logic gates, as necessary, to design the required network for the comparison of floating-point numbers.)

6.28 In Problem 6.25*a*, conversion of the simple decimal numbers into binary floating-point format is straightforward. However, if the decimal numbers are given in floating-point format, conversion is not straightforward because we cannot separately convert the mantissa and the exponent of the scale factor because $10^{x} = 2^{y}$ does not, in general, allow both x and y to be integers. Suppose a table of binary, floating-point numbers t_i, such that $t_i = 10^{x_i}$ for x_i in the representable range, is stored in a computer. Give a procedure in general terms for converting a given decimal floating-point number into binary floating-point format. You may use both the integer and floating-point instructions available in the computer.

6.29 Consider the representation of the decimal number 0.1 as a signed, 8-bit, binary fraction in the representation discussed at the beginning of Section 6.7. If the number does not convert exactly into this 8-bit format, approximate the number using all three of the truncation methods discussed in Section 6.7.3.

6.30 Construct an example to show that three guard bits are needed to produce the correct answer when two positive numbers are subtracted.

6.31 Which of the four 6-bit answers to Problem 6.2*a* are not exact? For each of these cases, give the three 6-bit values that correspond to the three types of truncation defined in Section 6.7.3.

6.32 Derive logic equations that specify the Add/Sub and S_R outputs of the combinational CONTROL network in Figure 6.26.

6.33 If gate fan-in is limited to four, how can the SHIFTER in Figure 6.26 be implemented combinationally?

6.34 (*a*) Sketch a logic-gate network that implements the multiplexer MUX in Figure 6.26.
(*b*) Relate the structure of the SWAP network in Figure 6.26 to your solution to Part (*a*).

6.35 How can the leading zeros detector in Figure 6.26 be implemented combinationally?

6.36 The mantissa adder-subtractor in Figure 6.26 operates on positive, unsigned binary fractions and must produce a sign-and-magnitude result. In the discussion accompanying Figure 6.26, we state that 1's-complement arithmetic is convenient because of the required format for input and output operands. When adding two signed numbers in 1's-complement notation, the carry-out from the sign position must be added to the result to obtain the correct signed answer. This is called *end-around carry correction*. Consider the two examples in Figure P6.3, which illustrate addition using signed, 4-bit encodings of operands and answers in the 1's-complement system.

The 1's-complement arithmetic system is convenient when a sign-and-magnitude result is to be generated because a negative number in 1's-complement notation can be converted to sign-and-magnitude form by complementing the bits to the right of the sign-bit position. Using 2's-complement arithmetic, addition of +1 is needed to convert a negative value into sign-and-magnitude notation. If a carry-lookahead adder is used, it is possible to incorporate the end-around carry operation required by 1's-complement arithmetic into the lookahead logic. With this discussion as a guide, give the complete design of the 1's-complement adder-subtractor required in Figure 6.26.

```
  (3)          0   0   1   1            (6)          0   1   1   0
+(-5)     + [0] 1 0 0 1 1 0 0 0       +(-3)     + [1] 1 1 1 1 0 0 0 0
-----     ------------------------    -----     ------------------------
  -2         | 1   1   0   1             3         | 0   0   1   0
             |______________► 0                    |______________► 1
          ------------------------              ------------------------
             1   1   0   1                         0   0   1   1
```

Figure P6.3 1's-complement addition used in Problem 6.36.

REFERENCES

1. Institute of Electrical and Electronics Engineers, *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-1985, August 1985.
2. J.L. Hennessy and D.A. Patterson, *Computer Architecture — A Quantitative Approach*, 2nd ed., Morgan Kaufmann, San Francisco, CA, 1996.