Step	Action	F	Accumulator A	Register Q
0	Initialize registers	0	00000000	10110011 = multiplier X
			11010101	= multiplicand $Y = M$
1			11010101	10110011
	Add M to A	1	11101010	11011001
	Right-shift F.A.Q	1	11010101	
2			10111111	11011001
	Add M to A	1	11011111	11101100
	Right-shift F.A.Q	1	00000000	
3			11011111	11101100
	Add zero to A	1	11101111	11110110
	Right-shift F.A.Q	1	00000000	
4			11101111	11110110
	Add zero to A	1	11110111	11111011
	Right-shift F.A.Q	1	11010101	
5			11001100	11111011
	Add M to A	1	11100110	01111101
	Right-shift F.A.Q	1	11010101	
6			10111011	01111101
	Add M to A	1	11011101	10111110
	Right-shift F.A.Q	1	00000000	
7			11011101	10111110
	Add zero to A	1	11101110	11011111
	Right-shift F.A.Q	1	11010101	
8			00011001	11011111
	Subtract M from A	1	00011001	11011110 = product P
	Set Q[0] to 0			

Figure 4.14
Illustration of the Robertson multiplication algorithm for twos-complement fractions.

Here y_7 is the sign of the multiplicand stored in M[7], and x_i is the current multiplier bit being tested in Q[0]. Thus F is set to 1 if Y is negative and at least one nonzero x_i is encountered. Once set to 1, it remains at that value. A negative Y and a positive or negative X therefore produce a series of negative partial products. This situation is to be expected, since bits x_6:x_0 of the multiplier X are always treated as if they were positive. A positive Y, or $X = 0$, causes F to remain permanently at 0. Note that the sign p_{15} of the product P requires no separate computational step. As in Example 2.7, the least significant bit p_0 of P is set to 0 to make the result exactly 16 bits long.

Figure 4.13 presents an HDL description of the twos-complement multiplication algorithm, which summarizes the foregoing analysis; compare the corresponding sign-magnitude algorithm in Figure 2.39. An application of the present algorithm to the case X = 10110011 and Y = 11010101 appears in Figure 4.14. The sign bit x_7 of the multiplier X is underlined to show its passage through Q. Observe how F becomes 1 in step 1, when the negative multiplicand is first added to the accumulator. F continues to supply leading 1s to the A register until step 8. Then because Q[7] = x_7 = 1, a subtraction is performed that produces the proper sign $p_{15} = 0$ in A(0). Setting Q[0] = p_0 to 0 completes the multiplication process.

Booth's algorithm. Another interesting and widely used scheme for twos-complement multiplication was proposed by Andrew D. Booth in the 1950s

[Booth 1951]. Like Robertson's method in Example 4.2, Booth's algorithm employs both addition and subtraction, but it treats positive and negative operands uniformly—no special actions are required for negative numbers. Booth's algorithm can also be readily extended in various ways to speed up the multiplication process; see problems 4.16 and 4.17. A version of this algorithm implements the ARM6's multiply instruction.

The multiplication algorithms we have considered so far involve scanning the multiplier X from right to left and using the value of the current multiplier bit x_i to determine which of the following operations to perform: add the multiplicand Y, subtract Y, or add zero, that is, no operation. In Booth's approach two adjacent bits $x_i x_{i-1}$ are examined in each step. If $x_i x_{i-1} = 01$, then Y is added to the current partial product P_i, while if $x_i x_{i-1} = 10$, Y is subtracted from P_i. If $x_i x_{i-1} = 00$ or 11, then neither addition or subtraction is performed; only the subsequent right shift of P_i takes place. Thus Booth's algorithm effectively skips over runs of 1s and runs of 0s that it encounters in X. This skipping reduces the average number of add-subtract steps and allows faster multipliers to be designed, although at the expense of more complex timing and control circuitry.

The validity of Booth's method can be seen as follows. Suppose that X is a positive integer and contains a subsequence X^* consisting of a run of k 1s flanked by two 0s.

$$\begin{aligned} X^* &= x_i x_{i-1} x_{i-2} \ldots x_{i-k+1} x_{i-k} x_{i-k-1} \\ &= 0\,1\,1 \ldots 1\,1\,0 \end{aligned}$$

In a direct add-and-shift multiplication algorithm such as Robertson's, Y is multiplied by each bit of X^* in sequence and the results are summed so that X^*'s contribution to the product $P = X \times Y$ is

$$\sum_{j=i-k}^{i-1} 2^j Y \tag{4.22}$$

Now when Booth's algorithm is applied to X^*, it performs an addition when it encounters $x_i x_{i-1} = 01$, which contributes $2^i Y$ to P. It performs a subtraction at $x_{i-k} x_{i-k-1} = 10$, which contributes $-2^{i-k} Y$ to P. Thus the net contribution of X^* to the product P in this case is

$$\begin{aligned} 2^i Y - 2^{i-k} Y &= 2^{i-k} Y (2^k - 1) Y \\ &= 2^{i-k} \sum_{m=0}^{k-1} 2^m Y \\ &= \sum_{m=0}^{k-1} 2^{m+i-k} Y \end{aligned} \tag{4.23}$$

Suppose the index m is replaced by $j = m + i - k$. Then the upper and lower limits of the summation in (4.23) change from $k - 1$ and 0 to $i - 1$ and $i - k$, respectively, implying that (4.22) and (4.23) are, in fact, the same. It follows that Booth's algorithm correctly computes the contribution of X^*, and hence of the entire multiplier X, to the product P. Equation (4.20) implies that the contribution of a negative X^*

```
BoothMult       (in: INBUS; out: OUTBUS);
                register A[7:0], M[7:0], Q[7:–1], COUNT[2:0],
                bus INBUS[7:0], OUTBUS[7:0];
BEGIN:          A := 0, COUNT := 0,
INPUT:          M := INBUS;
                Q[7:0] := INBUS, Q[–1] := 0;
SCAN:           if Q[1] Q[0] = 01 then A[7:0] := A[7:0] + M[7:0], go to
                TEST;
                  else if Q[1] Q[0] = 10 then A[7:0] := A[7:0] – M[7:0];
TEST:           if COUNT = 7 then go to OUTPUT,
RSHIFT:         A[7] := A[7], A[6:0].Q = A.Q[7:0],
INCREMENT:      COUNT := COUNT + 1, go to SCAN;
OUTPUT:         OUTBUS := A, Q[0] := 0;
                OUTBUS := Q[7:0];
end BoothMult;
```

Figure 4.15
HDL description of an 8-bit multiplier implementing the basic Booth algorithm.

to P can also be expressed in the formats of (4.20) and (4.23); a similar argument demonstrates the correctness of the algorithm for negative multipliers. The argument for fractions is essentially the same as that for integers.

The twos-complement multiplication circuit of Figure 4.12 can easily be modified to implement Booth's algorithm. Figure 4.15 describes a straightforward implementation of the Booth algorithm using the above approach with $n = 8$ and a circuit based on Figure 4.12. An extra flip-flop Q[–1] is appended to the right end of the multiplier register Q, and the sign logic for A is reduced to the simple sign extension A[7] := A[7]. In each step the two adjacent bits Q[0]Q[–1] of Q are examined, instead of Q[0] alone as in Robertson's algorithm, to decide the operation (add Y, subtract Y, or no operation) to be performed in that step. For comparison with Robertson's method in Figure 4.13, the operands are assumed to be fractions. The application of this algorithm to the example solved by Robertson's method in Figure 4.14 appears in Figure 4.16, where the bits stored in Q[0]Q[–1] in each step are underlined.

Combinational array multipliers. Advances in VLSI technology have made it possible to build combinational circuits that perform $n \times n$-bit multiplication for fairly large values of n. An example is the Integrated Device Technology IDT721CL multiplier chip, which can multiply two 16-bit numbers in 16 *ns* [Integrated Device Technology 1995]. These multipliers resemble the n-step sequential multipliers discussed above but have roughly n times more logic to allow the product to be computed in one step instead of in n steps. They are composed of arrays of simple combinational elements, each of which implements an add/subtract-and-shift operation for small slices of the multiplication operands.

Suppose that two binary numbers $X = x_{n-1}x_{n-2}\ldots x_1x_0$ and $Y = y_{n-1}y_{n-2}\ldots y_1y_0$ are to be multiplied. For simplicity, assume that X and Y are unsigned integers. The product $P = X \times Y$ can therefore be expressed as

Step	Action	Accumulator	Register Q
0	Initialize registers	00000000	10110011 = multiplier X
	Set Q[−1] to 0	00000000	101100110
		11010101	= mulitplicand Y = M
1	Subtract M from A	00101011	
	Right-shift A.Q	00010101	10110011**0**
	Skip add/subtract	00010101	110110011
2	Right-shift A.Q	00001010	110110011
		11010101	111011001
3	Add M to A	11011111	
	Right-shift A.Q	11101111	11101100**1**
	Skip add/subtract	11101111	111101100
4	Right-shift A.Q	11110111	111101100
		11010101	111110110
5	Subtract M from A	00100010	
	Right-shift A.Q	00010001	11111011**0**
	Skip add/subtract	00010001	011111011
6	Right-shift A.Q	00001000	011111011
		11010101	101111101
7	Add M to A	11011101	
	Right-shift A.Q	11101110	10111110**1**
		11010101	110111110
8	Subtract M from A	00011001	
	Set Q[0] to 0	00011001	11011110**0**
			110111100 = product P

Figure 4.16
Illustration of the Booth multiplication algorithm.

$$P = \sum_{i=0}^{n-1} 2^i x_i Y \tag{4.24}$$

corresponding to the bit-by-bit multiplication style of Figure 4.10. Now (4.24) can be rewritten as

$$P = \sum_{i=0}^{n-1} 2^i \left(\sum_{j=0}^{n-1} x_i y_j 2^j \right) \tag{4.25}$$

Each of the n^2 1-bit product terms $x_i y_j$ appearing in (4.25) can be computed by a two-input AND gate—observe that the arithmetic and logical products coincide in the 1-bit case. Hence an $n \times n$ array of two-input ANDs of the type shown in Figure 4.17 can compute all the $x_i y_j$ terms simultaneously. The terms are summed according to (4.25) by an array of $n(n-1)$ 1-bit full adders as shown in Figure 4.18; this circuit is a kind of two-dimensional ripple-carry adder. The shifts implied by the 2^i and 2^j factors in (4.25) are implemented by the spatial displacement of the adders along the x and y dimensions. Note the similarities between the circuit of Figure 4.17 and the multiplication examples of Figures 4.10 and 4.11.