CHAPTER 4

Datapath Design

An instruction-set processor consists of datapath (data processing) and control units. This chapter addresses the register-level design of the datapath unit, while Chapter 5 covers the control unit. The focus is on the arithmetic algorithms and circuits needed to process numerical data. These circuits are examined first for fixed-point numbers (integers) and then for floating-point numbers. The use of pipelining to speed up data processing is also discussed.

4.1 FIXED-POINT ARITHMETIC

The design of circuits to implement the four basic arithmetic instructions for fixed-point numbers—addition, subtraction, multiplication, and division—is the main topic of this section. It also discusses the implementation of logic instructions and ALU design.

4.1.1 Addition and Subtraction

Add and subtract instructions for fixed-point binary numbers are found in the instruction set of every computer. In smaller machines such as microcontrollers they are the only available arithmetic instructions. As we have seen in earlier chapters, addition and subtraction hardware (Example 2.7) or software (Example 3.1) can be used to implement multiplication and, in fact, any arithmetic operation. Beginning with Charles Babbage, computer designers have devoted considerable effort to the design of high-speed adders and subtracters. As we will see, these basic circuits can be designed in many different ways that involve various trade-offs between operating speed and hardware cost.

Basic adders. First consider the design of a circuit to add two n-bit unsigned binary numbers, a topic discussed in section 2.1.3. The fastest such adder is, in principle, a two-level combinational circuit in which each of the n sum bits is expressed as a (logical) sum of products or product of sums of the n input variables. In practice, such a circuit is feasible for very small values of n only, as it requires $c(n)$ gates with fan-in $f(n)$, where both $c(n)$ and $f(n)$ grow exponentially with n. Practical adders take the form of multilevel combinational or, occasionally, sequential circuits. They sacrifice operating speed for a reduction in circuit complexity as measured by the number and size of the components used. In general, the addition of two n-bit numbers X and Y is performed by subdividing the numbers into stages X_i and Y_i of length n_i, where $n > n_i > 1$. X_i and Y_i are added separately, and the resulting partial sums are combined to form the overall sum. The formation of this sum involves assimilation of carry bits generated by the partial additions.

The sum z_i,c_i of two 1-bit numbers x_i and y_i can be expressed by the *half-adder* logic equations

$$z_i = x_i \oplus y_i$$

$$c_i = x_i y_i$$

where z_i is the sum bit, c_i is the carry-out bit, $\oplus$ denotes EXCLUSIVE-OR, and juxtaposition denotes AND. If we introduce a third input bit c_{i-1} denoting a carry-in signal, we obtain the following *full-adder* equations:

$$z_i = x_i \oplus y_i \oplus c_{i-1}$$

$$c_i = x_i y_i + x_i c_{i-1} + y_i c_{i-1} \tag{4.1}$$

(Note that + denotes logical OR—not plus—here.) A full adder, also called a *1-bit adder,* can be directly implemented from these equations in various ways, as demonstrated by Figure 2.9 (section 2.1.1). Figure 4.1 shows a fast AND-OR realization of a 1-bit adder, along with an appropriate circuit symbol for use in register-level designs.

The least expensive circuit in terms of hardware cost for adding two n-bit binary numbers is a serial adder, the design of which was covered in Example 2.2. A serial adder adds the numbers bit by bit and so requires n clock cycles to compute the complete sum of two n-bit numbers. As Figure 4.2 indicates, a serial adder consists of a full adder realizing Equations (4.1) and a flip-flop to store c_i. One sum bit is generated in each clock cycle; a carry is also computed and stored for use during the next clock cycle. Figure 4.2 presents a high-level view of a serial adder that has a D flip-flop as the carry store. Although this adder is slow, its circuit size is very small and is independent of n.

Circuits that, in one clock cycle, add all bits of two n-bit numbers, as well as an external carry-in signal c_{in}, are called n-bit *parallel* adders or simply n-bit adders. The simplest such adder is formed by connecting n full adders as in Figure 4.3. Each 1-bit adder stage supplies a carry bit to the stage on its left. A 1 appearing on the carry-in line of a 1-bit adder can cause it to generate a 1 on its carry-out line. Hence carry signals propagate through the adder from right to left, giving rise to the name *ripple-carry adder*. In the worst case a carry signal can ripple through all n stages of the adder. The input carry signal c_{in} is normally set to 0 for addition. The maximum signal propagation delay of an n-bit ripple-carry adder, which in synchronous circuit design determines the operating speed, is nd,

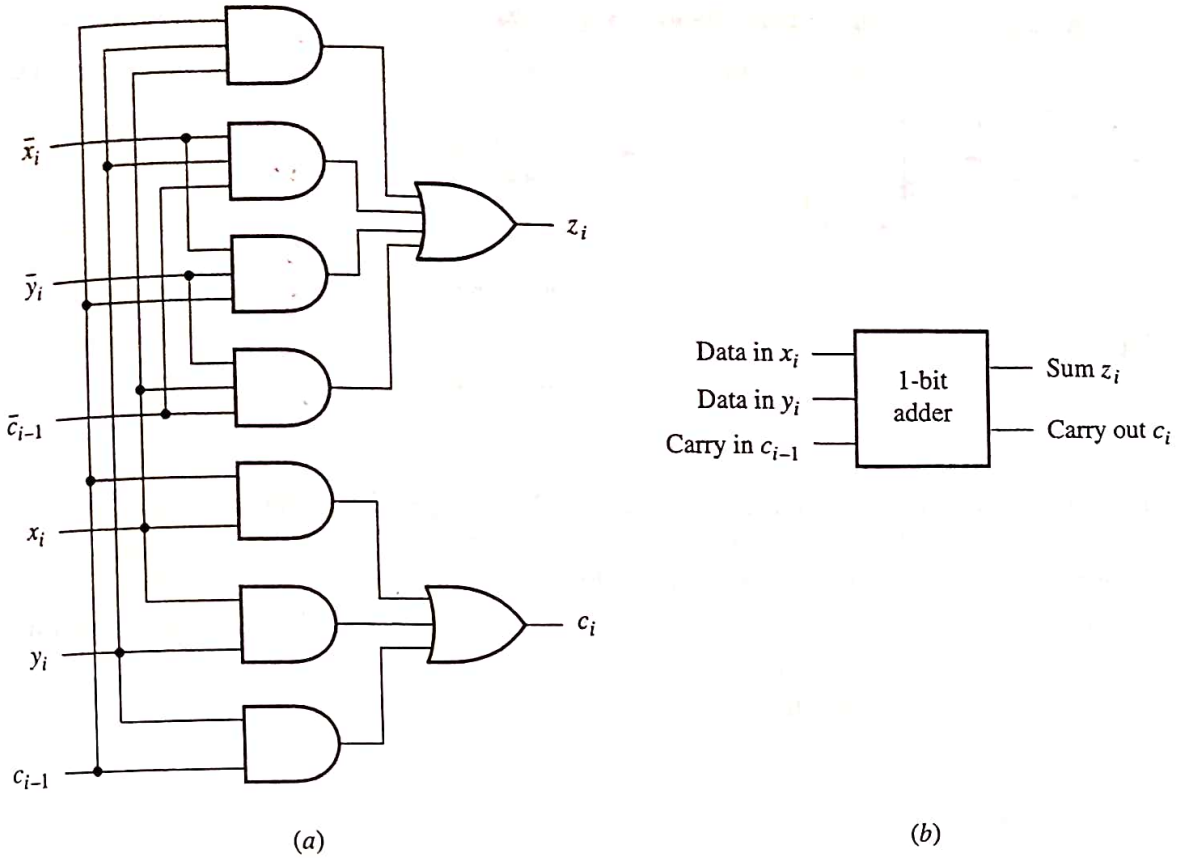


Figure 4.1
A 1-bit (full) adder: (*a*) two-level AND-OR logic circuit and (*b*) symbol.

where d is the delay of a full-adder stage. Unlike a serial adder, the amount of hardware in a ripple-carry adder increases linearly with n, the word size of the numbers being added.

Subtracters. Adders like those of Figures 4.2 and 4.3 operate correctly on both unsigned and positive numbers because the 0 sign bit of a positive number has the same effect as a leading zero in an unsigned number. The best way to add

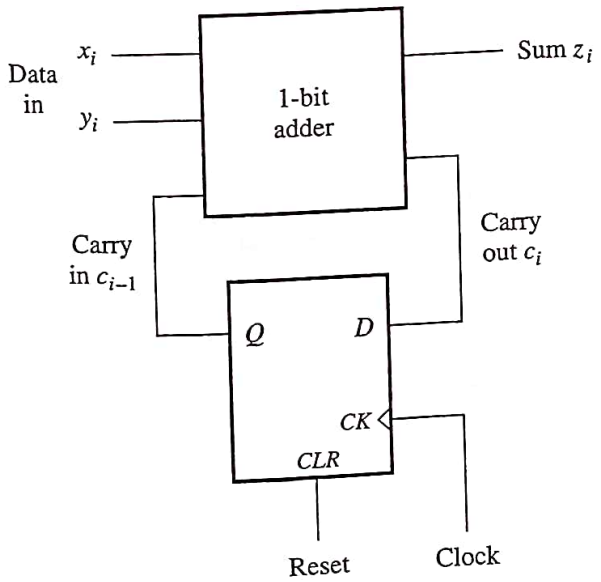


Figure 4.2
A serial binary adder.

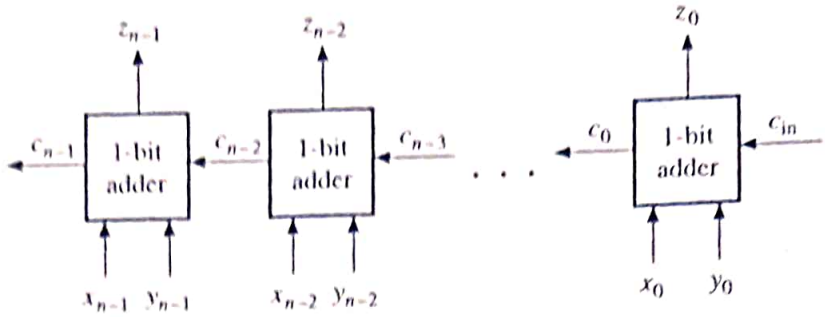


Figure 4.3
An n-bit ripple-carry adder composed of n 1-bit (full) adders.

negative numbers—these have 1 as the sign bit—depends on the number code in use. Adding $-X$ to Y is equivalent to subtracting X from Y, so the ability to add negative numbers implies the ability to do subtraction.

Subtraction is relatively simple with twos-complement code because negation (changing X to $-X$) is very easy to implement. As discussed in section 3.2.2, if $X = x_{n-1}x_{n-2}\ldots x_0$ is a twos-complement integer, then negation is realized by

$$-X = \bar{x}_{n-1}\bar{x}_{n-2}\ldots\bar{x}_0 + 1 \tag{4.2}$$

where + denotes addition modulo 2^n. An efficient way to obtain the ones-complement portion $\bar{X} = \bar{x}_{n-1}\bar{x}_{n-2}\ldots\bar{x}_0$ of $-X$ in (4.2) uses the word-based EXCLUSIVE-OR function $X \oplus s$ with a control variable s. When $s = 0$, $X \oplus s = X$, but when $s = 1$, $X \oplus s = \bar{X}$. Suppose that Y and $X \oplus s$ are now applied to the inputs of an n-bit adder. The addition of 1 required by (4.2) to change $\bar{X}$ to $-X$ can be realized by applying s to the carry input line of the adder. In the resulting circuit shown in Figure 4.4, the control line s selects the addition operation $Y + X$ when $s = 0$ and the subtraction operation $Y - X = Y + \bar{X} + 1$ when $s = 1$. Thus extending a parallel adder to perform twos-complement subtraction as well as addition merely requires connecting n two-input EXCLUSIVE-OR gates to the adder's inputs; these gates are represented by a single n-bit word gate in Figure 4.4.

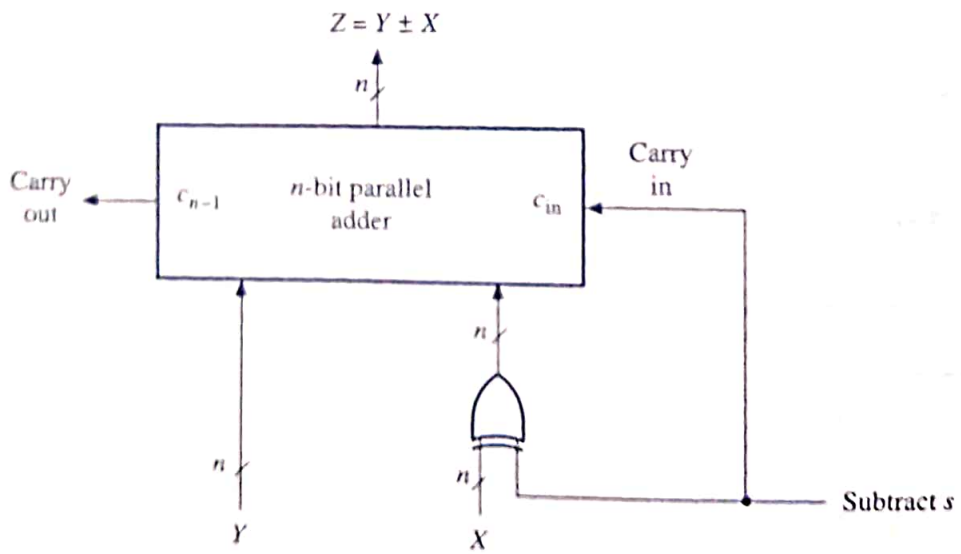


Figure 4.4
An n-bit twos-complement adder-subtracter.

As an example, let $X = 11101011$ and $Y = 00101000$, denoting -21_{10} and 40_{10}, respectively, in twos-complement code. Bit-by-bit addition produces

$$Z = X + Y = 11101011 + 00101000 = 00010011 \tag{4.3}$$

which corresponds to $-21_{10} + 40_{10} = +19_{10}$. (Observe that the output carry $c_{n-1} = 1$ in (4.3) is ignored.) To subtract X from Y, we first compute

$$-X = \bar{1}\,\bar{1}\,\bar{1}\,\bar{0}\,\bar{1}\,\bar{0}\,\bar{1}\,\bar{1} + 1 = 00010101$$

and then the sum

$$Z = (-X) + Y = 00010101 + 00101000 = 00111101$$

which corresponds to $21_{10} + 40_{10} = +61_{10}$.

Subtraction is not so readily implemented in the case of unsigned or sign-magnitude numbers. It is sometimes useful to construct a subtracter for such numbers based on the *full (1-bit) subtracter* function $z_i = y_i - x_i - b_{i-1}$. This operation is defined by the logic equations:

$$z_i = x_i \oplus y_i \oplus b_{i-1}$$

$$b_i = x_i\bar{y}_i + x_i b_{i-1} + \bar{y}_i b_{i-1}$$

Here z_i is the difference bit, while b_{i-1} and b_i are the borrow-in and borrow-out bits, respectively. n-bit serial or parallel binary subtracters are constructed in essentially the same way as the corresponding adders with carry signals replaced by borrows. Subtracters are of minor interest compared with adders, because, as we have just seen, an adder suffices for both addition and subtraction when twos-complement number code is used.

Overflow. When the result of an arithmetic operation exceeds the standard word size n, overflow occurs. With n-bit *un*signed numbers, overflow is indicated by an output carry bit $c_{n-1} = 1$. For example, adding the unsigned numbers $X = 11101011 = 235_{10}$ and $Y = 00101010 = 42_{10}$ using an adder like that of Figure 4.3 yields

$$Z = X + Y = 11101011 + 00101010 = 00010101 \tag{4.4}$$

with $c_{n-1} = c_7 = 1$. Now Z corresponds to 21_{10}, which is $235_{10} + 42_{10}$ (modulo 256) and is the result of addition that "wraps around" when the largest number $2^n - 1$, in this case $11111111 = 255_{10}$, is exceeded. On appending c_7 to Z, we get $c_7Z = 100010101 = 277_{10} = 256_{10} + 21_{10}$, which is the sum in ordinary (modulo infinity) arithmetic. Unsigned arithmetic operations are often viewed as modulo-2^n operations only, and overflow is not explicitly detected. This is the case when computing memory addresses in a computer, for instance, where addresses simply wrap around to zero after the highest address is reached.

Overflow is indicated by a flag bit v in operations involving signed numbers; this flag is found in CPU status (condition code) registers. If we reinterpret the numbers in the preceding example as twos-complement rather than as unsigned, then $X = 11101011$ denotes -21_{10}, while $Y = 00101010$ denotes $+42_{10}$. The result Z computed in (4.4) now denotes $+21_{10}$, and the fact that $c_{n-1} = 1$ does not indicate overflow. In fact, we can never have overflow on adding a positive to a negative number. Overflow in modulo-2^n twos-complement addition can only result from adding two positive numbers or two negative numbers. In the first case overflow

is indicated by a carry bit *into* the sign position, that is, by $c_{n-2} = 1$, since this indicates that the magnitude of the sum exceeds the $n - 1$ bits allocated to it. A little thought shows that overflow from adding two negative numbers is indicated by $c_{n-2} = 0$. We can thus conclude (as we did earlier in section 3.2.2) that the overflow condition is specified by the logic expression

$$v = \bar{x}_{n-1}\bar{y}_{n-1}c_{n-2} + x_{n-1}y_{n-1}\bar{c}_{n-2} \tag{4.5}$$

Now c_{n-1}, the carry output signal from the sign position, is defined by $x_{n-1}y_{n-1} + x_{n-1}c_{n-2} + y_{n-1}c_{n-2}$, from which it follows that

$$v = c_{n-1} \oplus c_{n-2} \tag{4.6}$$

Either (4.5) or (4.6) can be used to design overflow detection logic for twos-complement addition or subtraction. Overflow detection in the case of sign-magnitude numbers is similar and is left as an exercise (problem 4.6).

High-speed adders. The general strategy for designing fast adders is to reduce the time required to form carry signals. One approach is to compute the input carry needed by stage i directly from carrylike signals obtained from all the preceding stages $i - 1, i - 2, \ldots, 0$, rather than waiting for normal carries to ripple slowly from stage to stage. Adders that use this principle are called *carry-lookahead adders*. An n-bit carry-lookahead adder is formed from n stages, each of which is basically a full adder modified by replacing its carry output line c_i by two auxiliary signals called g_i and p_i, or *generate* and *propagate*, respectively, which are defined by the following logic equations:

$$g_i = x_i y_i \qquad p_i = x_i + y_i \tag{4.7}$$

The name *generate* comes from the fact that stage i generates a carry of 1 ($c_i = 1$) independent of the value of c_{i-1} if both x_i and y_i are 1; that is, if $x_i y_i = 1$. Stage i propagates c_{i-1}; that is, it makes $c_i = 1$ in response to $c_{i-1} = 1$ if x_i or y_i is 1—in other words, if $x_i + y_i = 1$.

Now the usual equation $c_i = x_i y_i + x_i c_{i-1} + y_i c_{i-1}$, denoting the carry signal c_i to be sent to stage $i + 1$, can be rewritten in terms of g_i and p_i.

$$c_i = g_i + p_i c_{i-1} \tag{4.8}$$

Similarly, c_{i-1} can be expressed in terms of g_{i-1}, p_{i-1}, and c_{i-2}.

$$c_{i-1} = g_{i-1} + p_{i-1}c_{i-2} \tag{4.9}$$

On substituting (4.9) into (4.8) we obtain

$$c_i = g_i + p_i g_{i-1} + p_i p_{i-1} c_{i-2}$$

Continuing in this way, c_i can be expressed as a sum-of-products function of the p and g outputs of all the preceding stages. For example, the carries in a four-stage carry-lookahead adder are defined as follows:

$$\begin{aligned}
c_0 &= g_0 + p_0 c_{in} \\
c_1 &= g_1 + p_1 g_0 + p_1 p_0 c_{in} \\
c_2 &= g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_{in} \\
c_3 &= g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_{in}
\end{aligned} \tag{4.10}$$

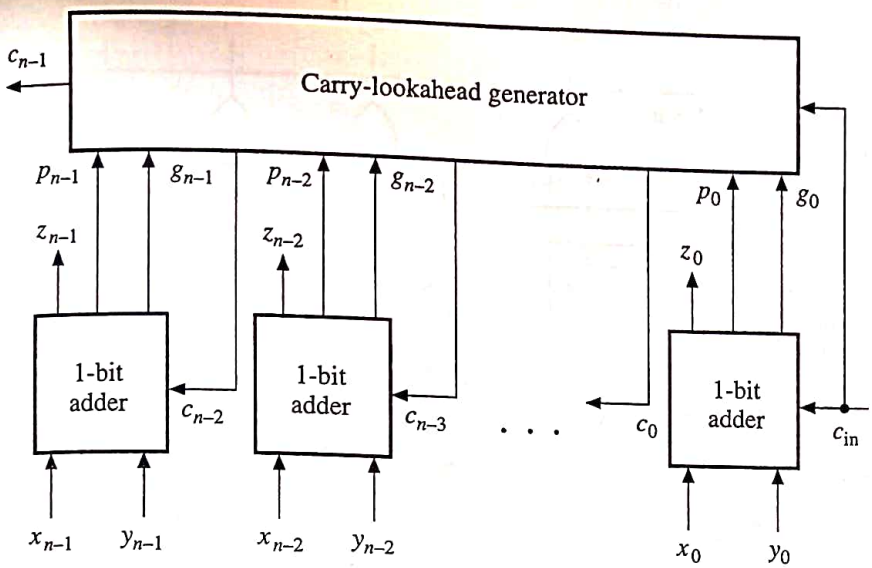


Figure 4.5
Overall structure of carry-lookahead adder.

Figure 4.5 shows the general form of a carry-lookahead adder circuit designed in this way.

We can further simplify the design by noting that the sum equation for stage i

$$z_i = x_i \oplus y_i \oplus c_{i-1}$$

is equivalent to

$$z_i = p_i \oplus g_i \oplus c_{i-1} \tag{4.11}$$

Combining the pg equations (4.7), the carry-lookahead equations (4.10), and the modified sum equations (4.11) for $0 \le i \le 3$, we obtain the 4-bit carry-lookahead adder depicted in Figure 4.6. This design is found in practical adders such as the 74283 IC [Texas Instruments 1988]. It has four levels of logic gates, so the adder's maximum delay is $4d$, where d is the (average) gate delay. This delay is independent of the number of inputs n as long as carry generation is defined by two-level logic as in (4.10). However, the number of gates grows in proportion to n^2 as n increases. In contrast, the number of gates in a two-level adder of the sum-of-products type grows exponentially with n, while the number of gates in a ripple-carry adder grows linearly with n. The complexity of the carry-generation logic in the carry-lookahead adder, including its gate count, its maximum fan-in, and its maximum fan-out, increases steadily with n. Such practical cost considerations limit n in a single carry-lookahead adder module to four or so.

Adder expansion. The methods of handling carry signals in the two main combinational adder designs considered so far, namely, ripple-carry propagation (Figure 4.3) and carry-lookahead (Figure 4.5), can be extended to larger adders of the kind needed to execute add instructions in, say, a 64-bit computer. If we replace the n 1-bit (full) adder stages in the n-bit ripple-carry design of Figure 4.3 with n k-bit adders, we obtain an nk-bit adder. Four 4-bit adders such as the 4-bit carry-lookahead circuit of Figure 4.6 can be connected in this way to form the 16-bit adder appearing in Figure 4.7. This design represents a compromise between a 16-stage ripple-carry adder, which is cheap but slow, and a single-stage 16-bit

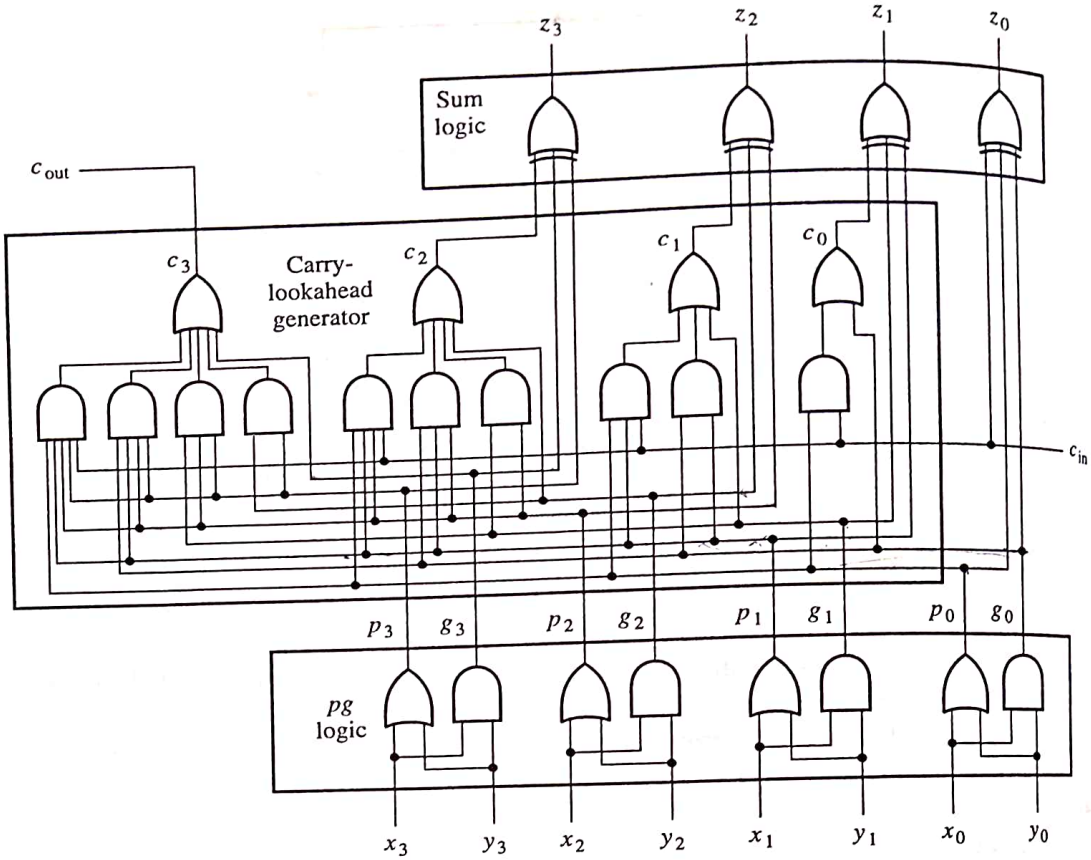


Figure 4.6
A 4-bit carry-lookahead adder.

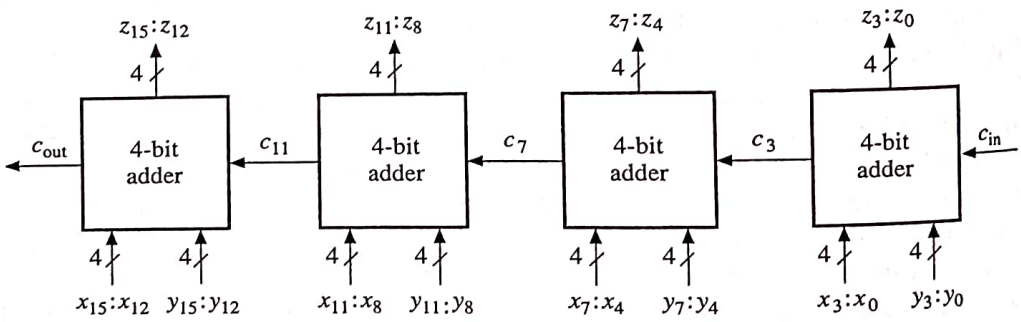


Figure 4.7
A 16-bit adder composed of 4-bit adders linked by ripple-carry propagation.

carry-lookahead adder, which is fast, expensive, and impractical because of the complexity of its carry-generation logic. The circuit of Figure 4.7 effectively combines sets of four $x_i y_i$ inputs into groups that are added via carry lookahead; the results computed by the various groups are then linked via ripple carries.

Comparing Figures 4.3 and 4.7, we see that we have effectively replaced components designed for 1-bit addition with similar but larger components intended for 4-bit addition. If we apply the same principle to the carry-lookahead circuit of Figure 4.5, we get the expanded design of Figure 4.8. Again we are replacing 1-bit

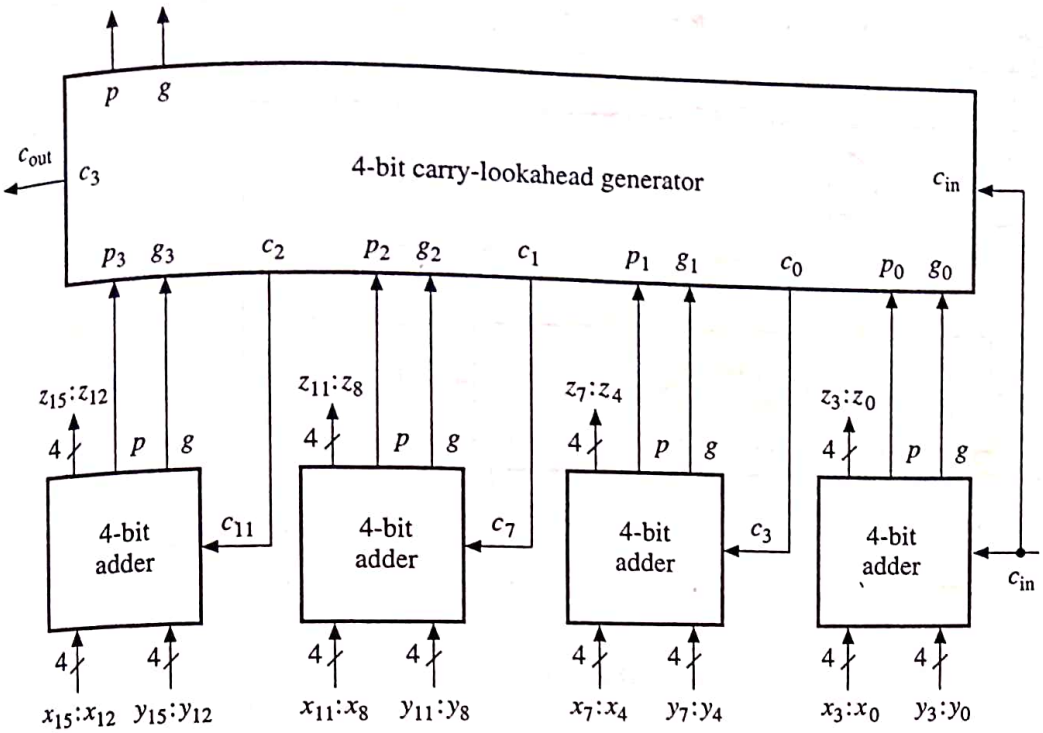


Figure 4.8
A 16-bit adder composed of 4-bit adders linked by carry lookahead.

adders with 4-bit adders, but now each adder stage produces a propagate-generate signal pair pg instead of c_{out}, and a carry-lookahead generator converts the four sets of pg signals to the carry inputs required by the four stages. The "group" g and p signals produced by each 4-bit stage are defined by

$$\begin{aligned} g &= x_i y_i + x_{i-1} y_{i-1}(x_i + y_i) + x_{i-2} y_{i-2}(x_i + y_i)(x_{i-1} + y_{i-1}) \\ &\quad + x_{i-3} y_{i-3}(x_i + y_i)(x_{i-1} + y_{i-1})(x_{i-2} + y_{i-2}) \\ p &= (x_i + y_i)(x_{i-1} + y_{i-1})(x_{i-2} + y_{i-2})(x_{i-3} + y_{i-3}) \end{aligned} \tag{4.12}$$

which directly extend (4.7). It is not hard to show that the logic to generate the group carry signals c_{out}, c_{11}, c_7, and c_3 in Figure 4.8 is exactly the same as that of the carry-lookahead generator of Figure 4.6 and is therefore defined by Equations (4.10).

EXAMPLE 4.1 DESIGN OF A COMPLETE TWOS-COMPLEMENT ADDER-SUBTRACTER. To illustrate the preceding concepts, we will design a twos-complement adder-subtracter that computes the three quantities $X + Y$, $X - Y$, and $Y - X$, as well as overflow and zero flags. The design goal is to minimize the number of gates used; operating speed is not of concern. The circuit is required in several versions that handle different data word sizes, including 4, 8, and 16 bits. We will assume that we have standard gate-level and 4-bit register-level components available as building blocks.

The lowest cost adders employ ripple-carry propagation and can easily provide access to the internal signals needed by the flags. Recall that overflow detection uses c_{n-2}, the input carry to the sign position. Zero detection requires access to all the sum outputs and poses no special problems. Figure 4.9*a* shows the logic diagram of an appropriate 4-bit ripple-carry adder. The overflow flag is defined by Equation (4.6) as $v = c_3 \oplus c_2$ and is realized here by an XOR gate. The zero flag is defined by $z =$

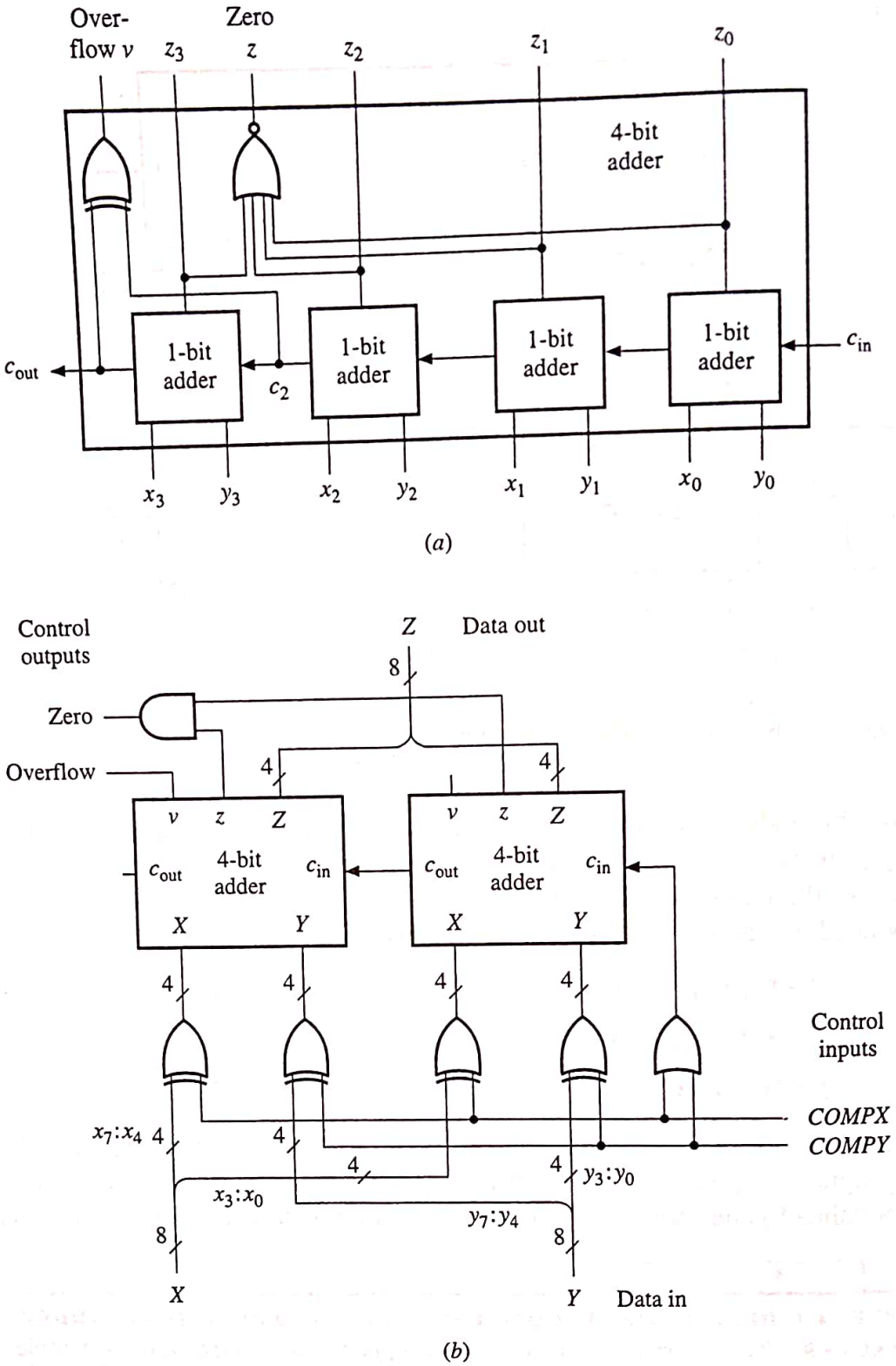


Figure 4.9
Low-cost addition and subtraction of twos-complement numbers: (*a*) 4-bit adder module and (*b*) 8-bit adder-subtracter.

$z_3 + z_2 + z_1 + z_0$ and implemented by a NOR gate. We can use k copies of this adder to produce a $4k$-bit ripple-carry adder in the usual way. The overflow flag for the entire circuit is taken from the v output of the left-most (most significant) stage, while the z outputs of all the stages are ANDed to produce the zero flag.

To extend the adder to an adder-subtracter, the design of Figure 4.4 is a good starting point. It uses an XOR word gate to complement the X input, thereby enabling the circuit to compute $X + Y$ and $Y - X$. To implement the third operation $X - Y$, we could

insert a two-way 4-bit multiplexer into each of the data-in buses so that both X and Y can be applied to each of the adder-subtracter's data inputs. A cheaper solution is to insert a second XOR word gate into the Y bus, enabling Y to be complemented independently. We can then compute $X - Y$ in the form $X + \bar{Y} + 1$.

The complete design of an 8-bit adder-subtracter along the foregoing lines is depicted in Figure 4.9*b*. It contains two 4-bit adders of the type in Figure 4.9*a* linked by their carry lines. Two lines *COMPX* and *COMPY* control the XOR gates that change X and Y to $\bar{X}$ and $\bar{Y}$, respectively. The OR gate sets the adder's carry-in line to 1 during subtraction. A two-input AND gate combines the two z outputs to produce the zero flag, which is 1 if and only if the entire 8-bit result $Z = 0$.

Three of the four signal combinations on *COMPX* and *COMPY* control lines implement the desired three arithmetic functions. The fourth combination 11 implements the sum $\bar{X} + \bar{Y} + 1$, which is an arithmetic function implemented by our design that has no obvious uses.[1] Such superfluous functions are common in the design of data processing circuits.

4.1.2 Multiplication

Fixed-point multiplication requires substantially more hardware than fixed-point addition and, as a result, is not included in the instruction sets of some smaller processors. Multiplication is usually implemented by some form of repeated addition. A simple but slow method to compute $X \times Y$ is to add the multiplicand Y to itself X times, where X is the multiplier. (A version of this technique using counters is discussed in problem 2.4.) Often multiplication is implemented by multiplying Y by X k bits at a time and adding the resulting terms. Figure 4.10 shows this process for unsigned binary numbers in pencil-and-paper calculations with $k = 1$. The main operations involved are shifting and addition. The algorithm of Figure 4.10 is inefficient in that the 1-bit products $x_j 2^i Y$ must be stored until the final addition step is completed. In machine implementations it is desirable to add each $x_j 2^i Y$ term as it is generated to the sum of the preceding terms to form a number P_{i+1} called a *partial product*. Figure 4.11 shows the calculation in Figure 4.10 implemented in this way. The computation involved in processing one multiplier bit x_j can be described by a register-transfer statement of the form

$$P_{i+1} := P_i + x_j 2^i Y \tag{4.13}$$

1010	Multiplicand Y
1101	Multiplier $X = x_3x_2x_1x_0$
1010	x_0Y
0000	$x_1 2Y$
1010	$x_2 2^2 Y$
1010	$x_3 2^3 Y$
10000010	Product $P = \sum_{j=0}^{3} x_j 2Y^j$

Figure 4.10
Typical pencil-and-paper method for multiplication of unsigned binary numbers.

[1]On the other hand, it has been observed, that "there is no feature of a machine, however pathological, which cannot be exploited by a programmer." [Kampe 1960].

1010	Multiplicand Y
1101	Multiplier $X = x_3x_2x_1x_0$
00000000	$P_0 = 0$
1010	x_0Y
00001010	$P_1 = P_0 + x_0Y$
0000	$x_1 2Y$
00001010	$P_2 = P_1 + x_1 2Y$
1010	$x_2 2^2 Y^j$
00110010	$P_3 = P_2 + x_2 2^2 Y$
1010	$x_3 2^3 Y$
10000010	$P_4 = P_3 + x_3 2^3 Y = P$

Figure 4.11
The multiplication of Figure 4.10 modified for machine implementation.

where 2^iY is equivalent to Y shifted i positions to the left. In the version of this multiplication algorithm presented in Example 2.7 (section 2.2.3), P_i is shifted right with respect to a fixed multiplicand Y so that (4.13) is replaced by the equivalent two operations

$$P_i := P_i + x_jY; \quad P_{i+1} := 2^{-j}P_i; \tag{4.14}$$

The multiplication of sign-magnitude numbers requires a straightforward extension of the unsigned case discussed above. The magnitude part of the product $P = X \times Y$ is computed by the unsigned shift-and-add multiplication algorithm, and the sign p_s of P is computed separately from the signs of X and Y as follows: $p_s := x_s \oplus y_s$. The implementation of sign-magnitude multiplication using this sequential method is covered in Example 2.7.

Twos-complement multipliers. The multiplication of twos-complement numbers presents some difficulties in the case of negative operands. For example, when a negative P_i is right-shifted as in (4.14), leading 1s rather than leading 0s must be introduced at the left end of the number. More seriously, the multiplication process must treat positive and negative operands differently.

A conceptually simple approach to twos-complement multiplication is to negate all negative operands at the beginning, perform unsigned multiplication on the resulting (positive) numbers, and then negate the result if necessary. Twos-complement negation for an integer $X = x_{n-1}x_{n-2}x_{n-3}\ldots x_1x_0$ is specified by

$$-X = \bar{x}_{n-1}\bar{x}_{n-2}\bar{x}_{n-3}\ldots\bar{x}_1\bar{x}_0 + 000\ldots01 \quad (\text{modulo } 2^n) \tag{4.15}$$

and can easily be implemented by an adder and an EXCLUSIVE-OR word gate, as shown in Figure 4.4. However, up to four extra clock cycles are needed to negate X and Y and the double-length product P. Several faster schemes have been proposed to handle negative operands. Since these hinge on certain properties of the twos-complement representation, we consider the latter first.

Clearly $\bar{x}_i = 1 - x_i$ (modulo 2), so we can rewrite (4.15) as follows:

$$-X = 111\ldots11 - x_{n-1}x_{n-2}x_{n-3}\ldots x_1x_0 + 000\ldots01 \quad (\text{modulo } 2^n) \tag{4.16}$$

Since $2^n = 111\ldots11 + 000\ldots01$, this equation is equivalent to $-X = 2^n - X$, which, incidentally, indicates the origin of the term twos-complement. Now if X is positive

($x_{n-1} = 0$), we can express its value as

$$X = \sum_{i=0}^{n-2} 2^i x_i \tag{4.17}$$

If X is negative ($x_{n-1} = 1$), then (4.17) does not hold. However, we can rewrite (4.16) as

$$\begin{aligned} -X &= 111\ldots11 - (0x_{n-2}x_{n-3}\ldots x_1x_0 + 100\ldots00) + 000\ldots01 \\ &= 2^{n-1} - x_{n-2}x_{n-3}\ldots x_1x_0 \end{aligned} \tag{4.18}$$

because $2^{n-1} = 111\ldots11 - 100\ldots00 + 000\ldots0$. Hence for negative X,

$$\begin{aligned} X &= -2^{n-1} + x_{n-2}x_{n-3}\ldots x_1x_0 \\ &= -2^{n-1} + \sum_{i=0}^{n-2} 2^i x_i \end{aligned} \tag{4.19}$$

Finally, we combine (4.17) and (4.19) into a single formula

$$X = -2^{n-1}x_{n-1} + \sum_{i=0}^{n-2} 2^i x_i \tag{4.20}$$

which is valid for both positive and negative n-bit integers. For example, suppose that $n = 6$ and $X = 101101$. Evaluating X according to (4.20) yields

$$\begin{aligned} X &= -2^5 \times 1 + 2^4 \times 0 + 2^3 \times 1 + 2^2 \times 1 + 2^1 \times 0 + 2^0 \times 1 \\ &= -32 + 8 + 4 + 1 = -19 \end{aligned}$$

Equation (4.20) implies that we can treat bits $x_{n-2}x_{n-3}\ldots x_1x_0$ of a negative twos-complement integer in the same way as the corresponding (magnitude) bits of a positive number; each bit x_i has the positive weight 2^i. Weight $+2^{n-1}$ is assigned to the sign bit x_{n-1} of a positive number; however, since $x_{n-1} = 0$, its contribution to the number is zero. In the negative case, the sign x_{n-1} is assigned weight -2^{n-1}; this adds -2^{n-1} to the number, ensuring that it is negative.

If $X = x_{n-1}x_{n-2}\ldots x_1x_0$ is a twos-complement fraction instead of an integer, then the negation formula (4.15) remains valid, but because bit i now has weight 2^{i-n+1} instead of 2^i, Equation (4.20) is replaced by

$$X = -2^0 x_{n-1} + \sum_{i=0}^{n-2} 2^{i-n+1} x_i \tag{4.21}$$

In effect we have multiplied (4.20) by the scaling factor $2^{-(n-1)}$. For example, let $n = 4$ and $X = 1011$, which represents the fraction -0.625_{10}. Application of (4.21) yields

$$\begin{aligned} X &= -2^0 \times 1 + 2^{-1} \times 0 + 2^{-2} \times 1 + 2^{-3} \times 1 \\ &= -1.000 + 0.250 + 0.125 = -0.625 \end{aligned}$$

Suppose that X is the multiplier operand in a shift-and-add multiplication algorithm to compute $P = X \times Y$ for twos-complement numbers. Equations (4.20) and

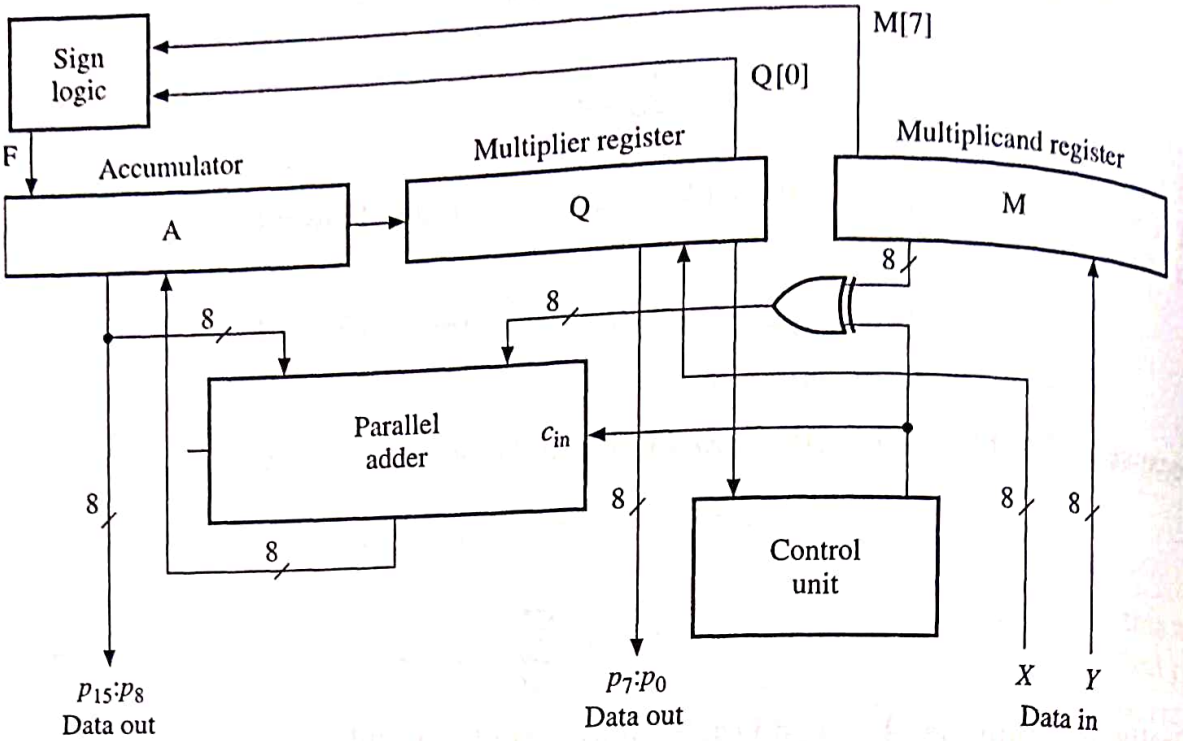


Figure 4.12
The datapath of the twos-complement multiplier.

(4.21) suggest that we can use an unsigned multiplication technique like those illustrated in Figures 4.12 and 4.13 with one change: When multiplying by the sign bit, perform subtraction rather than addition in the final step if a minus sign $x_{n-1} = 1$ is encountered. This observation is the basis of a twos-complement multiplication algorithm developed by James E. Robertson, which has been widely used in computer design [Robertson 1955; Cavanagh 1984]. We now show one way to adapt the circuit developed in Example 2.7 for sign-magnitude multiplication to deal with the twos-complement case.

```
2Cmultiplier   (in: INBUS; out: OUTBUS);
               register A[7:0], M[7:0], Q[7:0], COUNT[2:0], F;
               bus INBUS[7:0], OUTBUS[7:0];
BEGIN:         A := 0, COUNT := 0, F := 0,
INPUT:         M := INBUS;
               Q := INBUS;
ADD:           A[7:0] := A[7:0] + M[7:0] × Q[0],
               F := (M[7] and Q[0]) or F;
RSHIFT:        A[7] := F, A[6:0].Q := A.Q[7:1], COUNT := COUNT + 1;
TEST:          if COUNT ≠ 7 then go to ADD;
SUBTRACT:      A[7:0] := A[7:0] − M[7:0] × Q[0], Q[0] := 0;
OUTPUT:        OUTBUS := Q;
               OUTBUS := A;
end 2Cmultiplier;
```

Figure 4.13
HDL description of the multiplier for 8-bit twos-complement fractions.

EXAMPLE 4.2 DESIGN OF A MULTIPLIER FOR TWOS-COMPLEMENT FRACTIONS. Consider again the task of multiplying two 8-bit binary fractions $X = x_7x_6x_5x_4x_3x_2x_1x_0$ and $Y = y_7y_6y_5y_4y_3y_2y_1y_0$ to form the product $P = Y \times X$, this time using twos-complement code. (Example 2.7 analyzed this problem for the sign-magnitude case.) Assume that the multiplier will have a register-level structure similar to that in Figure 2.41, with registers A, M, and Q storing the various operands and A.Q forming a right-shift register. Since sign bits will be included in additions and subtractions, we need an 8-bit adder-subtracter, rather than the 7-bit magnitude-only adder used in the earlier design. Figure 4.12 shows the datapath of the proposed design at the register level.

To develop the required twos-complement multiplication algorithm for this machine, we consider the four possible cases determined by the signs of X and Y.

1. $x_7 = y_7 = 0$; that is, both X and Y are positive. The computation in this case is effectively unsigned multiplication with the product P computed in a series of add-and-shift steps of the form

$$P_i := P_i + x_iY; \quad P_{i+1} := 2^{-1}P_i;$$

All partial products P_i are nonnegative, so leading 0s are introduced into A during the right-shift operation indicated by the factor 2^{-1}.

2. $x_7 = 0$, $y_7 = 1$; that is, X is positive and Y is negative. The partial product P_i will be zero, and leading 0s should be shifted into A as before, until the first 1 in X is encountered. Multiplication of Y by this 1 and addition of the result to A causes P_i to become negative, from which point on leading 1s rather than 0s must be shifted into A. These rules ensure that a right shift corresponds to division by 2 in twos-complement code.

3. $x_7 = 1$, $y_7 = 0$; that is, X is negative and Y is positive. This follows case 1 for the first seven add-and-shift steps yielding the partial product

$$P_7 = \sum_{i=0}^{6} 2^{i-7} x_i Y$$

For the final step, often referred to as a correction step, the subtraction $P := P_7 - Y$ is performed. The result P is then given by

$$P = -Y + \sum_{i=0}^{6} 2^{i-7} x_i Y = \left(-x_7 + \sum_{i=0}^{6} 2^{i-7} x_i\right) Y$$

which is $X \times Y$ by (4.21).

4. $x_7 = y_7 = 1$; that is, both X and Y are negative. The procedure used here follows case 2, with leading 0s (1s) being introduced into the accumulator whenever its contents are zero (negative). The correction (subtraction) step of case 3 is also performed, which ensures that the final product in A.Q is nonnegative.

Each addition/subtraction step can be performed in the usual twos-complement fashion by treating the sign bits like any other and ignoring overflow. Care is needed in the shift step to ensure that the correct new value is placed in the accumulator's sign position A[7]. This value must be a leading 0 if the current partial product in A.Q is positive or zero, and 1 if it is negative. We introduce a flip-flop F to control the values assigned to A[7]. F is initially set to 0, and is subsequently defined by

$$F := (y_7 \textit{ and } x_i) \textit{ or } F$$

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

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.

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

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
1		11010101	= mulitplicand Y = M
	Subtract M from A	00101011	101100110
	Right-shift A.Q	00010101	110110011
2	Skip add/subtract	00010101	110110011
	Right-shift A.Q	00001010	111011001
3		11010101	
	Add M to A	11011111	111011001
	Right-shift A.Q	11101111	111101100
4	Skip add/subtract	11101111	111101100
	Right-shift A.Q	11110111	111110110
5		11010101	
	Subtract M from A	00100010	111110110
	Right-shift A.Q	00010001	011111011
6	Skip add/subtract	00010001	011111011
	Right-shift A.Q	00001000	101111101
7		11010101	
	Add M to A	11011101	101111101
	Right-shift A.Q	11101110	110111110
8		11010101	
	Subtract M from A	00011001	110111110
	Set Q[0] to 0	00011001	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.

The AND and add functions of the array multiplier can be combined into a single component (cell) as illustrated in Figure 4.19. This cell realizes the arithmetic expression

$$c_{out}s = a \textbf{ plus } b \textbf{ plus } xy \tag{4.26}$$

An $n \times n$-bit multiplier can be built using n^2 copies of this cell as the sole component, although, as in Figure 4.18, some cells on the periphery of the array have inputs set to 0 or 1, effectively reducing their operation from (4.26) a ***plus*** b ***plus*** xy to a ***plus*** b (a half adder). The multiplication time for this multiplier is determined by the worst-case carry propagation and, ignoring the differences between the internal and peripheral cells, is $(2n-1)D$, where D is the delay of the basic cell.

Multiplication algorithms for twos-complement numbers, such as Robertson's and Booth's, can also be realized by arrays of combinational cells as the next example shows.

EXAMPLE 4.3 ARRAY IMPLEMENTATION OF THE BOOTH MULTIPLICATION ALGORITHM [KOREN 1993]. Implementing the Booth method by a combinational array requires a multifunction cell capable of addition, subtraction, and no operation (skip). Such a cell B is shown in Figure 4.20a. Its various functions are selected by a pair of control lines H and D as indicated. It is easily seen that the required functions of B are defined by the following logic equations.

$$z = a \oplus bH \oplus cH$$

$$c_{out} = (a \oplus D)(b + c) + bc$$

When $HD = 10$, these equations reduce to the usual full-adder equations (4.1); when $HD = 11$, they reduce to the corresponding full-subtracter equations

$$z = a \oplus b \oplus c$$

$$c_{out} = \bar{a}b + \bar{a}c + bc$$

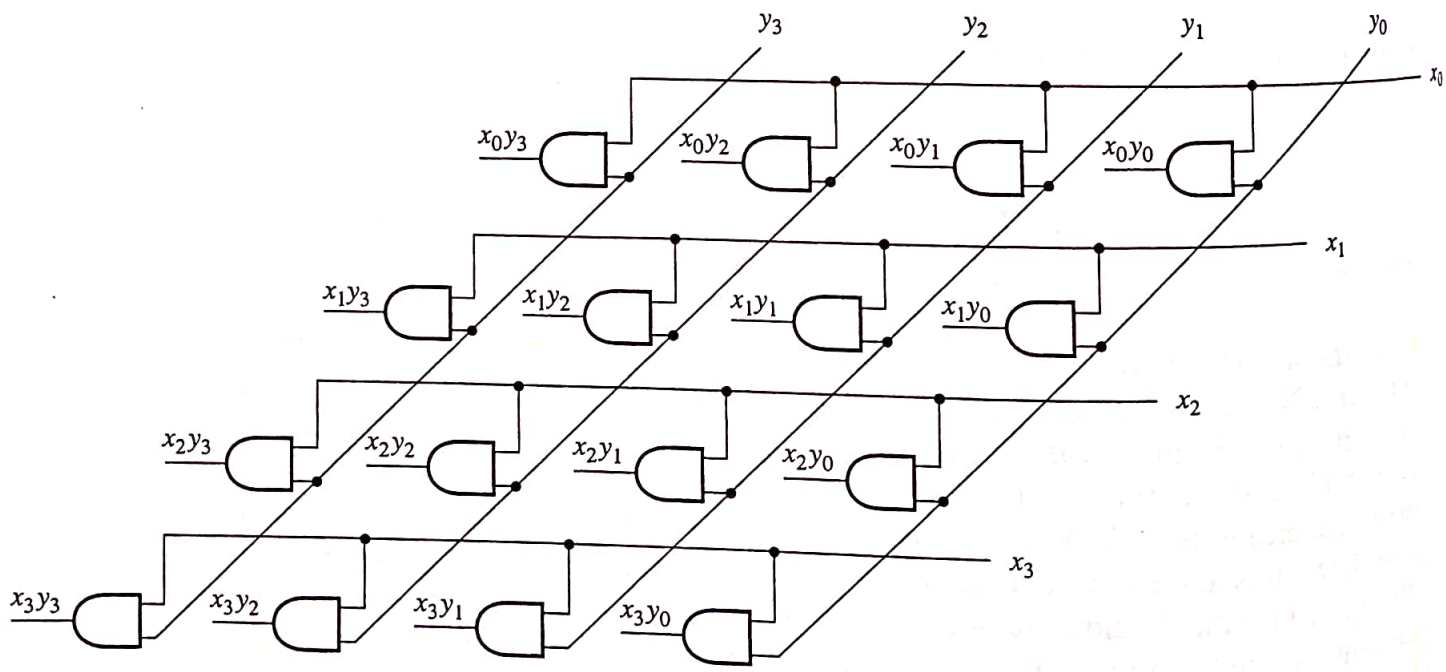


Figure 4.17
AND array for 4×4-bit unsigned multiplication.

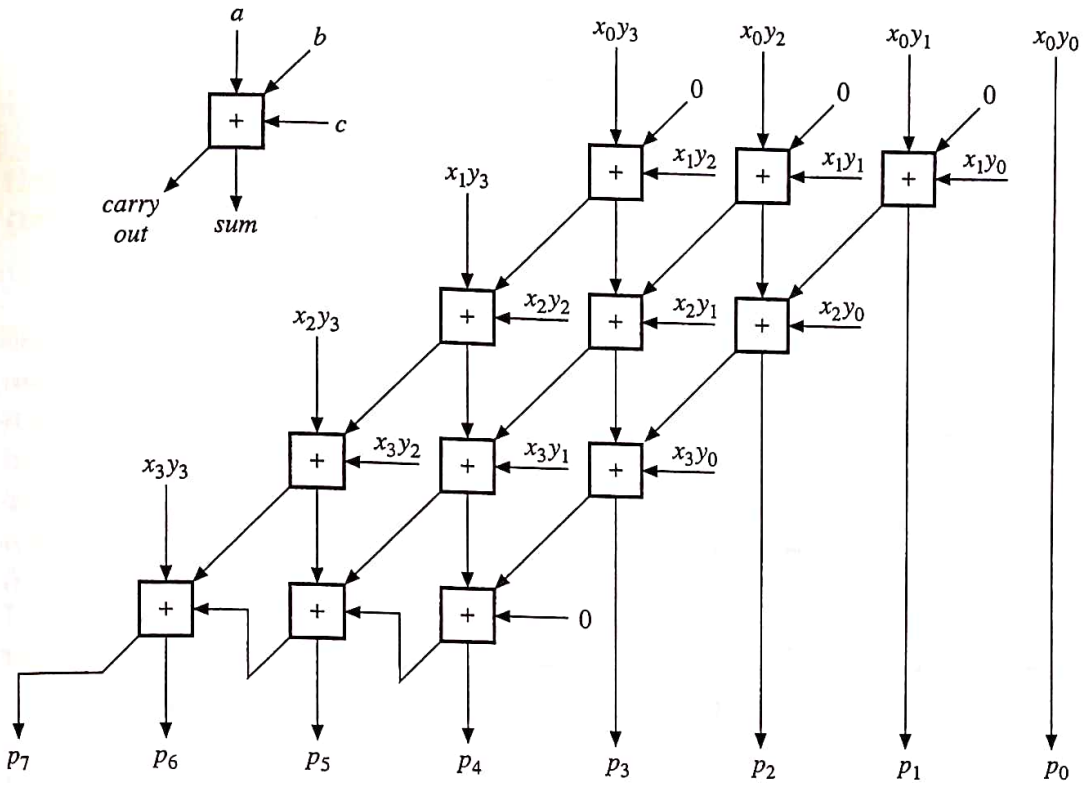


Figure 4.18
Full-adder array for 4 × 4-bit unsigned multiplication.

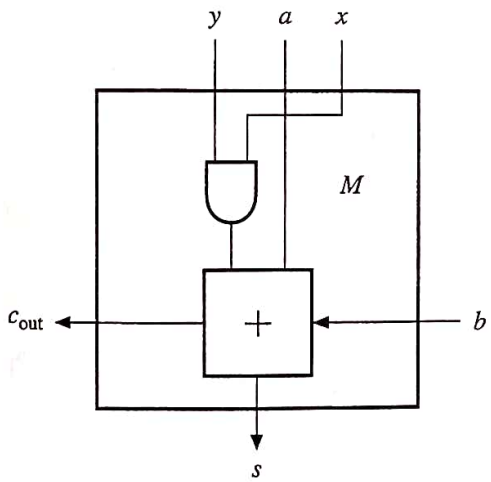
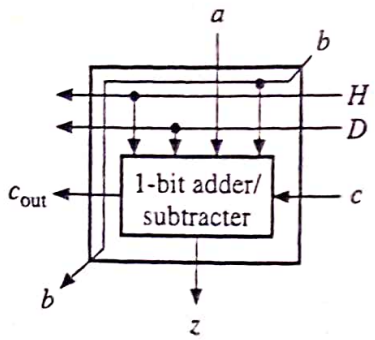


Figure 4.19
Cell M for an unsigned array multiplier.

in which c and c_{out} assume the roles of borrow-in and borrow-out, respectively. When $H = 0$, z becomes a, and the carry lines play no role in the final result.

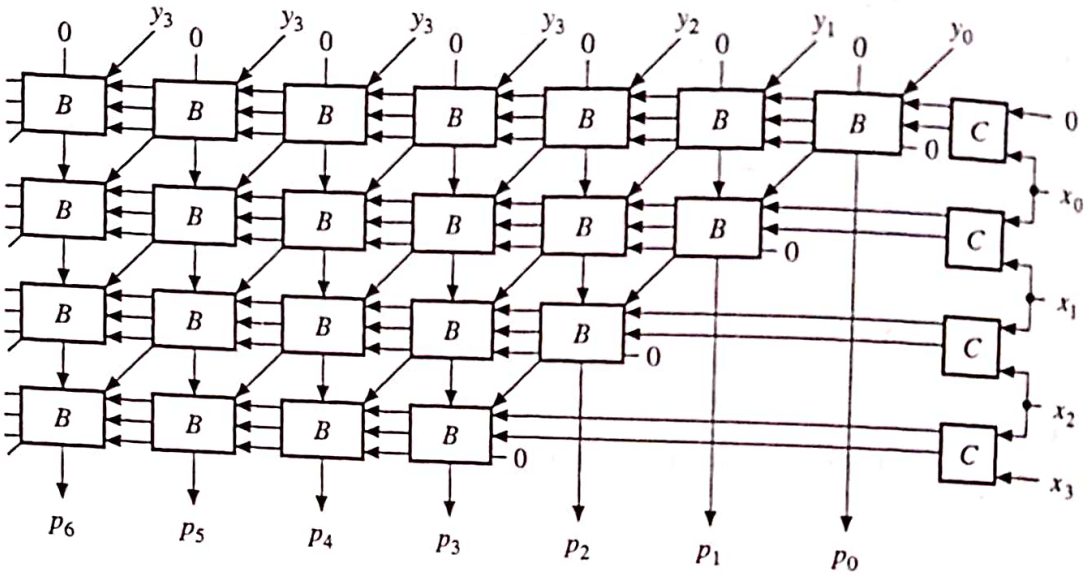
An n-bit multiplier is constructed from $n^2 + n(n-1)/2$ copies of the B cell connected as shown in Figure 4.20*b*. The extra cells at the top left change the array's shape from the parallelogram of Figure 4.18 to a trapezium and are employed to sign-extend the multiplicand Y for addition and subtraction. Note how the diagonal lines marked b deliver the sign-extended Y directly to every row of B cells. When Y is positive, it is sign-extended by leading 0s; this is implicit in the array of Figure 4.18. In the present case, when Y is negative, it must be explicitly sign-extended by leading 1s.

The operation to be performed by each row i of B cells is decided by bits x_ix_{i-1} of the operand X. To allow each possible x_ix_{i-1} pair to control row operations, we



H D	Function
0 X	$z = a$ (no operation)
1 0	$c_{out}z = a$ *plus* b *plus* c (add)
1 1	$c_{out}z = a - b - c$ (subtract)

(*a*)



(*b*)

Figure 4.20
Combinational array implementing Booth's algorithm: (*a*) main cell *B* and (*b*) array multiplier for 4 × 4-bit numbers.

introduce a second cell type denoted *C* in Figure 4.20*b* to generate the control input signal *H* and *D* required by the *B* cells. Cell *C* compares x_i with x_{i-1} and generates the values of *HD* required by Figure 4.20*a*; these values are as follows:

$$H = \overline{x_i \oplus x_{i-1}}$$
$$D = x_i \bar{x}_{i-1}$$

4.1.3 Division

In fixed-point division two numbers, a divisor *V* and a dividend *D*, are given. The object is to compute a third number *Q*, the quotient, such that $Q \times V$ equals or is very close to *D*. For example, if unsigned integer formats are being used, *Q* is computed so that

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

where R, the remainder, is required to be less than V, that is, $0 \le R < V$. We can then write

$$D/V = Q + R/V \tag{4.27}$$

Here R/V is a small quantity representing the error in using Q alone to represent D/V; this error is zero if $R = 0$.

Preliminaries. The relationship $D \approx Q \times V$ suggests that a close correspondence exists between division and multiplication, specifically the dividend, quotient, and divisor correspond to the product, multiplicand, and multiplier, respectively. This correspondence means that similar algorithms and circuits can be used for multiplication and division. In multiplication the shifted multiplier is added to the multiplicand to form the product. In division the shifted divisor is subtracted from the dividend to form the quotient. Just as multiplication ends with a double-length product, division often begins with a double-length dividend. Despite these similarities, division is a more difficult operation than multiplication because to determine a particular quotient bit q_i, we have to answer the question: How many multiples is the divisor V of the current partial dividend D_i? This question is typically answered by trial and error: Multiply V by a trial value for q_i, subtract the result from D_i, and check the value of the remainder. Note too that the next quotient bit q_{i+1} cannot be determined until q_i is known. Thus division has an element of uncertainty not found in multiplication.

One of the simpler binary division methods is a sequential digit-by-digit algorithm similar to that used in pencil-and-paper methods with decimal numbers. Figure 4.21 illustrates this approach for a 3-bit divisor $V = 101$ and a 6-bit dividend $D = 100110$. The dividend is scanned from left to right, and the quotient is computed bit by bit. In each step divisor V is compared to the current partial dividend D_i, referred to here as the partial remainder R_i.[2] The current quotient bit q_i is either 0 or 1, and is determined by comparing V with R_i; this comparison is the hard part of division. Note that decimal division is harder than binary in this

	0111	Quotient $Q = q_3q_2q_1q_0$
Divisor $V = 101$	100110	Dividend $D = R_0$
	000	q_3V
	100110	R_1
	101	$q_2 2^{-1}V$
	10010	R_2
	101	$q_1 2^{-2}V$
	1000	R_3
	101	$q_0 2^{-3}V$
	011	R_4 = remainder R

Figure 4.21
Typical pencil-and-paper method for division of unsigned numbers.

[2]We use the terms *partial dividend* and *partial remainder* interchangeably because the remainder from step i is used as the dividend in step $i + 1$.

regard because q_i must be selected from 10 possible digit values instead of from two. If the numbers appearing in the division calculation of Figure 4.21 are unsigned binary integers of length six, then (4.27) becomes

$$100110. / 000101. = 000111. + 000011./ 000101.$$

corresponding to the decimal division 38/5 = 7 + 3/5. If the numbers are unsigned 6-bit fractions, then Figure 4.21 is interpreted as

$$.100110/.101000 = .111000 + .000011/.101000$$

corresponding to .59375/.625 = .875 + .046875/.625.

In integer arithmetic Q and R are always integers of the standard word size. If fraction formats are used, however, the number of bits of Q is not necessarily bounded. For example, .2000/.3000 = .66666..., a repeating fraction. It is necessary, therefore, to limit the number of quotient bits generated by the division process. Division of .2000 by .3000 might be required to yield a four-digit quotient Q with truncation or rounding determining the final digit of Q. Several other difficulties occur in division. If D is too large relative to V, then Q will not fit in the standard word size, resulting in *quotient overflow*. For instance, the four-digit fraction division .2000/.0100 produces a nonfraction six-digit result 20.0000. When $V = 0$, the quotient Q is treated as undefined or infinity and a *divide-by-zero error* is said to occur. Special circuits are employed to check for, and flag, quotient overflow and zero divisors before division begins.

Basic algorithms. Suppose that the divisor V and dividend D are unsigned integers and the quotient $Q = q_{n-1}q_{n-2}q_{n-3}\ldots$ is to be computed one bit at a time. At each step i, $2^{-i}V$, which represents the divisor shifted i bits to the right, is compared with the current partial remainder R_i. The quotient bit q_i is set to 1 (0) if $2^{-i}V$ is less (greater) than R_i, and a new partial remainder R_{i+1} is computed according to the relation

$$R_{i+1} := R_i - q_i 2^{-i} V \tag{4.28}$$

In machine implementations it is more convenient to shift the partial remainder to the left relative to a fixed divisor, in which case (4.28) is replaced by

$$R_{i+1} := 2R_i - q_i V$$

Figure 4.22 shows the calculation of Figure 4.21 modified in this way. The final partial remainder R_4 is now the overall remainder R shifted three bits to the left, so that $R = 2^{-3}R_4$.

As observed above, the central problem in division is finding the quotient digit q_i. If radix-r numbers are being represented, then q_i must be chosen from among r possible values. When $r = 2$, q_i can be generated by comparing V and $2R_i$ in the ith step, as is done implicitly in Figure 4.22. If $V > 2R_i$, then $q_i = 0$; otherwise, $q_i = 1$. If V is long, a combinational magnitude comparator circuit may be impractical, in which case q_i is usually determined by subtracting V from $2R_i$ and examining the sign of $2R_i - V$. If $2R_i - V$ is negative, $q_i = 0$; otherwise, $q_i = 1$.

The circuit used for multiplication in Example 4.2 (Figure 4.12) is easily modified to perform division, as shown in Figure 4.23. The $2n$-bit shift register A.Q stores the partial remainders. Initially the dividend (which can contain up to $2n$

Divisor V			Quotient Q
101	100110	Dividend $D = 2R_0$	
	000	q_3V	0
	100110	R_1	
	1001100	$2R_1$	
	101	q_2V	01
	100100	R_2	
	1001000	$2R_2$	
	101	q_1V	011
	100000	R_3	
	1000000	$2R_3$	
	101	q_0V	0111
	011000	$R_4 = 2^3R$	

Figure 4.22
The division of Figure 4.21 modified for machine implementation.

bits) is placed in A.Q. The divisor V is placed in the M register where it remains throughout the division process. In each step A.Q is shifted to the left. The positions vacated at the right-most end of the Q register can be used to store the quotient bits as they are generated. When the division process terminates, Q contains the quotient, while A contains the (shifted) remainder.

As noted already, the quotient bit q_i can be determined by a trial subtraction of the form $2R_i - V$. This subtraction also yields the new partial remainder R_{i+1} when $2R_i - V$ is positive; that is, when $q_i = 1$. Clearly, the process of determining q_i and R_{i+1} can be integrated. Two major division algorithms are distinguished by the way they combine the computation of q_i and R_{i+1}. If $q_i = 0$, then the result of the trial

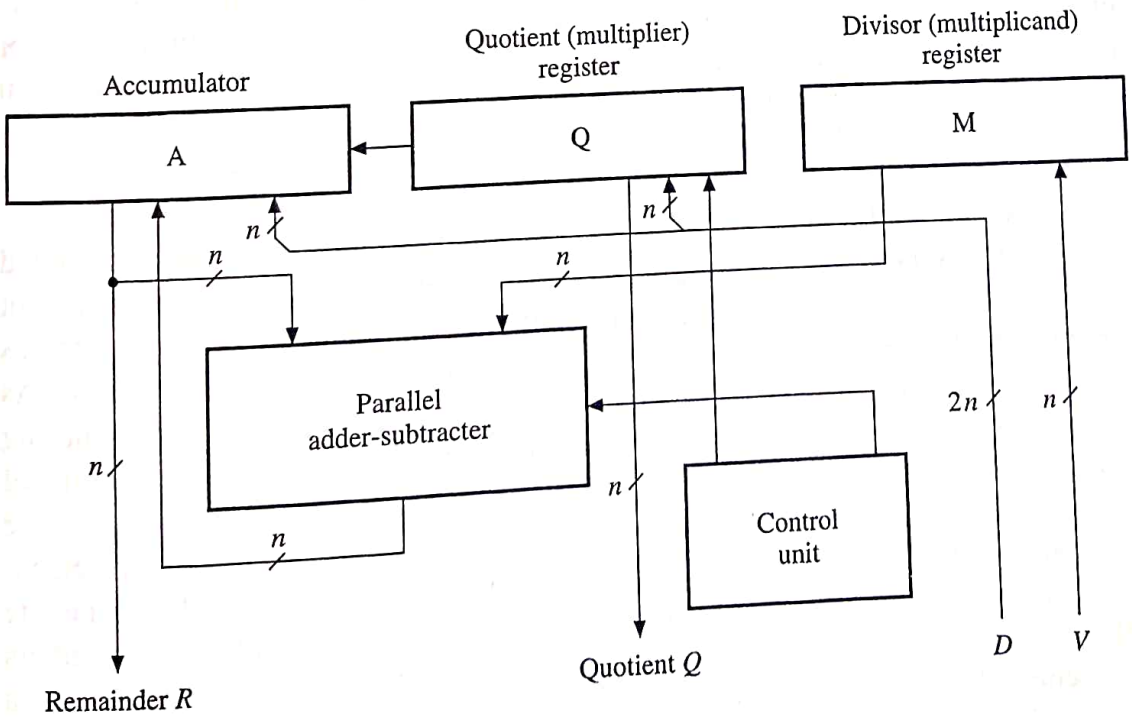


Figure 4.23
The datapath of a sequential n-bit binary divider.

subtraction is $2R_i - V$; however, the required new partial remainder R_{i+1} is $2R_i$. The partial remainder R_{i+1} can be obtained by adding V back to the result of the trial subtraction. This straightforward technique is called *restoring division*. In every step the operation

$$R_{i+1} := 2R_i - V \tag{4.29}$$

is performed. When the result of the subtraction is negative, a restoring addition is performed as follows:

$$R_{i+1} := R_{i+1} + V$$

If the probability of $q_i = 1$ is 1/2, then this algorithm requires n subtractions and an average of $n/2$ additions.

The restoration step of the preceding algorithm is eliminated in a slightly different technique called *nonrestoring division*. This method is based on the observation that a restoration of the form

$$R_i := R_i + V \tag{4.30}$$

is followed in the next step by the subtraction (4.29). Operations (4.29) and (4.30) can be merged into the single operation

$$R_{i+1} := 2R_i + V \tag{4.31}$$

Thus when $q_i = 1$, which is indicated by a positive value of R_i, R_{i+1} is computed using (4.29). When $q_i = 0$, R_{i+1} is computed using (4.31). The calculation of each quotient bit involves either an addition or a subtraction, but not both. Nonrestoring division therefore requires n additions or subtractions, whereas restoring division requires an average of $3n/2$ additions and subtractions.

Figure 4.24 presents a nonrestoring division algorithm designed for the circuit of Figure 4.23 with unsigned integers. The divisor V and quotient Q are n bits long (with leading 0s if necessary), while the dividend D is up to $2n - 1$ bits long, which is the maximum length of the product of two n-bit integers. The flip-flop S is appended to the accumulator A to record the sign of the result of an addition or subtraction and to determine the quotient bit. Each new quotient bit is placed in $Q[0]$, and the final values of the quotient Q and the remainder R are in the Q and A registers, respectively. An application of this algorithm when $n = 4$ appears in Figure 4.25 with $D = 1100001_2 = 97_{10}$ and $V = 1010_2 = 10_{10}$.

The restoring and nonrestoring division techniques can be extended to signed numbers in much the same way as multiplication. Sign-magnitude numbers present few difficulties; the magnitudes of the quotient and remainder can be computed as in the unsigned number case, while their signs are determined separately. As remarked in [Cavanagh 1984], there are no simple division algorithms for handling negative numbers directly in twos-complement code because of the difficulty of selecting the quotient bits so that the quotient has the correct positive or negative representation. The most direct approach to signed division is to negate any negative operands, perform division on the resulting positive numbers, and then negate the results, as needed. A fast division algorithm for twos-complement numbers based on the nonrestoring approach was devised independently in 1958 by Dura W. Sweeney, James E. Robertson, and Keith D. Tocher and is called the SRT method in their honor; see [Cavanagh 1984; Koren 1993] for details.

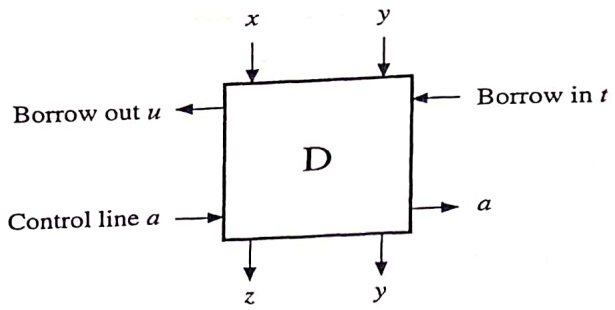
```
NRdivider        (in: INBUS; out: OUTBUS);
                 register S, A[n–1:0], M[n–1:0], Q[n–1:0], COUNT[⌈log2n⌉:0];
                 bus INBUS[n–1:0], OUTBUS[n–1:0];
BEGIN:           COUNT := 0, S := 0,
INPUT:           A := INBUS; {Input the left half of the dividend D}
                 Q := INBUS; {Input the right half of the dividend D}
                 M := INBUS; {Input the divisor V}
SUBTRACT:        S.A := S.A – M; {S is the sign of the result}
TEST:            if S = 0 then
                   begin Q[0] := 1;
                   if COUNT = n – 1 then go to CORRECTION; else
                     begin COUNT := COUNT + 1, S.A.Q[n–1:1] := A.Q; end
                   S.A := S.A – M, go to TEST; end
                 else {if S = 1}
                   begin Q[0] := 0;
                   if COUNT = n – 1 then go to CORRECTION; else
                     begin COUNT := COUNT + 1, S.A.Q[n–1:1] := A.Q; end
                 S.A := S.A + M, go to TEST; end
CORRECTION:      if S = 1 then S.A := S.A + M;
OUTPUT:          OUTBUS := Q; {Output the quotient Q}
                 OUTBUS := A; {Output the remainder R}
end NRdivider;
```

Figure 4.24
Nonrestoring division algorithm for unsigned integers.

Step	Action	S	A	Q
0	Initialize registers	0	1100	0010 = dividend D
			1010	= divisor V = M
1				
	Subtract M from A	0	0010	0010
	Reset Q[0]	0	0010	0011
	Left shift S.A.Q	0	0100	0110
			1010	
2				
	Subtract M from A	1	1010	0110
	Set Q[0]	1	1010	0110
	Left shift S.A.Q	1	0100	1100
			1010	
3				
	Add M to A	1	1110	1100
	Set Q[0]	1	1110	1100
	Left shift S.A.Q	1	1101	1000
			1010	
4				
	Add M to A	0	0111	1000
	Reset Q[3]	0	0111	1001
				1001 = quotient Q
			0111	= remainder R

Figure 4.25
Illustration of the nonrestoring division algorithm for unsigned integers.

गंधाला।



a	Function
0	$uz = x$ ***minus*** y ***minus*** t
1	$z = x$

Figure 4.26
A cell D for array implementation of restoring division.

Combinational array dividers. Combinational array circuits can be used for division as well as for multiplication. Figure 4.26 shows a cell D suitable for implementing a version of the restoring division algorithm. This cell is basically a full subtracter with t and u being the borrow-in and borrow-out bits, respectively. The main output z is controlled by input a. When $a = 1$, z is the difference bit defined by the arithmetic equation

$$z = x \textbf{ minus } y \textbf{ minus } t$$

When $a = 0$, $z = x$. Thus the behavior of the cell D is given by the logic equations

$$z = x \oplus \bar{a}\,(y \oplus t)$$

$$u = \bar{x}y + \bar{x}t + yt$$

Figure 4.27 shows an array of D cells to divide 3-bit unsigned integers and generate a 4-bit quotient. Each row of the array subtracts the divisor V from the shifted partial remainder $2R_i$ generated by the row above. The sign of the result, and therefore of the quotient bit, is indicated by the borrow-out signal from the left-most cell in the row. This signal u_i is connected to the control inputs a of all cells in the same row. If $u_i = 0$, then the output from the row is $2R_i - V$ and $q_i = \bar{u}_i = 1$. If $u_i = 1$, then the output from the row is restored to $2R_i$, and again $q_i = \bar{u}_i = 0$. Thus the output of each row is initially $2R_i - V$, but it is restored to $2R_i$ when required. Restoration is achieved by overriding the subtraction performed by the row rather than by explicitly adding back the divisor.

Let d and d' be the carry (borrow) propagation and restore times of a cell, respectively. Let the divisor and dividend be n bits long. Each row of the divider array functions as an n-bit ripple-borrow subtracter, so the maximum time required to compute one quotient bit is $nd + d'$. The time required to compute an m-bit quotient and the corresponding remainder is therefore $m(nd + d')$, and the number of cells needed is $m(n + 1) - 1$.

Division by repeated multiplication. In systems containing a high-speed multiplier, division can be performed efficiently and at low cost using repeated multiplication. In each iteration a factor F_i is generated and used to multiply both the divisor V and the dividend D. Therefore

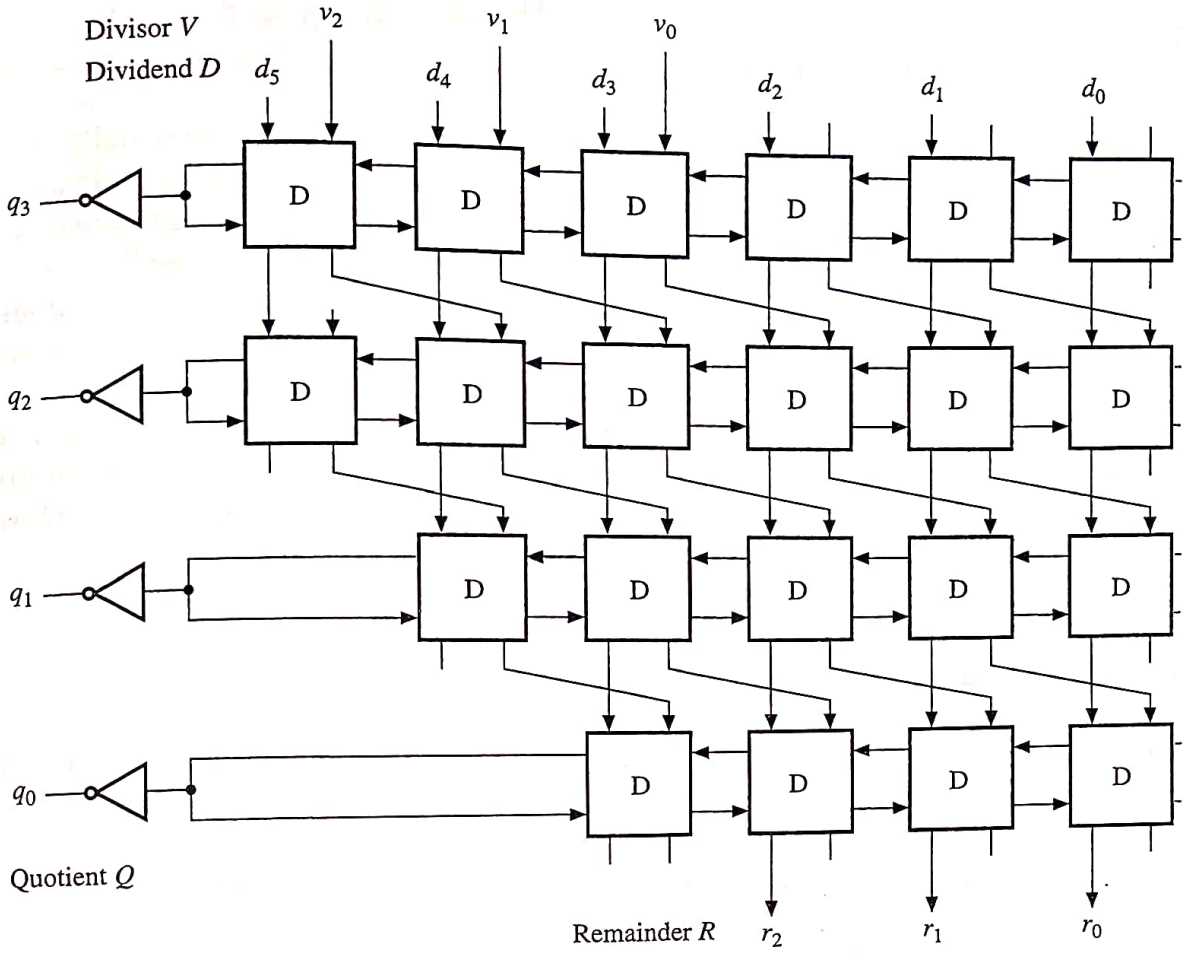


Figure 4.27
A divider array for 3-bit unsigned numbers using the cell D of Figure 4.26.

$$Q = \frac{D \times F_0 \times F_1 \times F_2 \times \ldots}{V \times F_0 \times F_1 \times F_2 \times \ldots}$$

F_i is chosen so that the sequence $V \times F_0 \times F_1 \times F_2 \ldots$ converges rapidly toward one. Hence $D \times F_0 \times F_1 \times F_2 \ldots$ must converge toward the desired quotient.

The convergence of the method depends on the selection of the F_i's. For simplicity, assume that D and V are positive normalized fractions so that $V = 1 - x$, where $x < 1$. Set $F_0 = 1 + x$. We can now write

$$V \times F_0 = (1 - x)(1 + x) = 1 - x^2$$

Clearly $V \times F_0$ is closer to one than to V. Next set $F_1 = 1 + x^2$. Hence

$$V \times F_0 \times F_1 = (1 - x^2)(1 + x^2) = 1 - x^4$$

and so on. Let V_i denote $V \times F_0 \times F_1 \times \ldots \times F_i$. The multiplication factor at each stage is computed as $F_i = 2 - V_{i-1}$, which is simply the twos-complement of V_{i-1}. Hence

$$F_i = 1 + x^{2^i} \text{ and } V_i = 1 - x^{2^{i+1}}$$

As i increases, V_i converges quickly toward one. The process terminates when $V_i =$ 0.11...11, the number closest to one for the given word size.