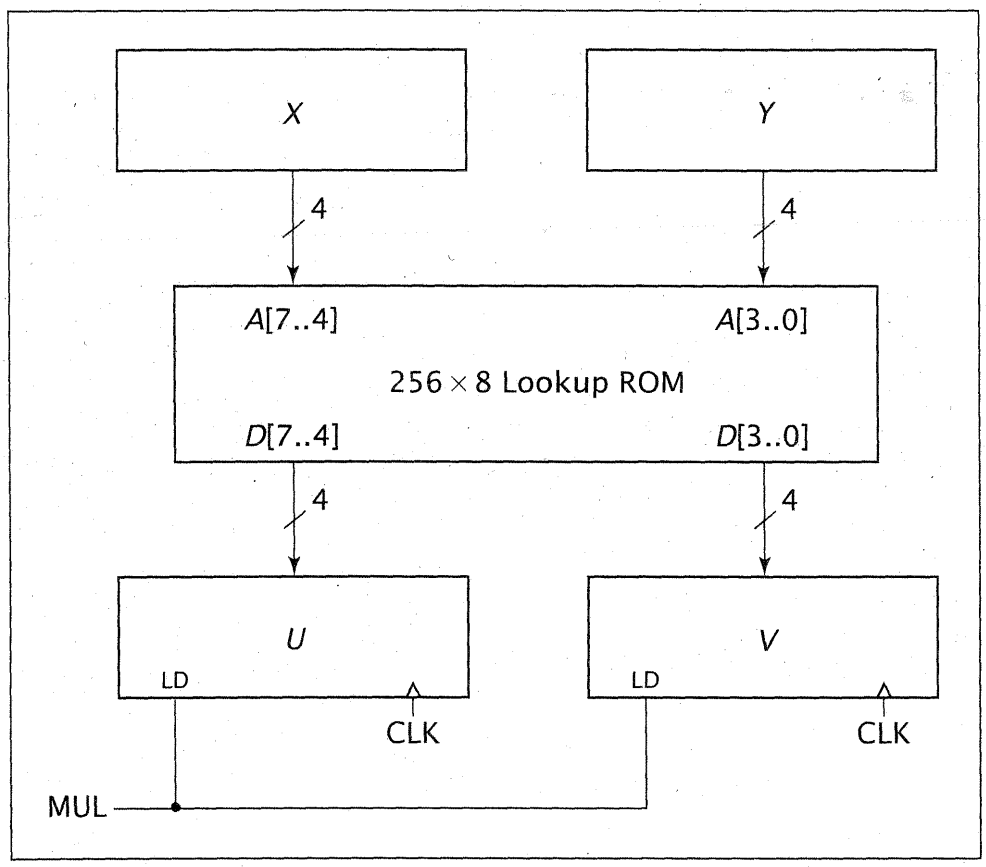
FIGURE 8.16
A multiplier implemented using a lookup ROM



sented in this chapter. For example, the hardware shown in Figure 8.16 may be less complex than that of the original shift-add implementation shown in Figure 8.5. It can also multiply numbers more quickly than the shift-add hardware. However, the size of the lookup ROM grows rapidly as the size of the operands increases. Although the 4-bit multiplier requires a ROM of size 256 × 8, an equivalent 8-bit multiplier would use a 64K × 16 ROM.

8.4.3 Wallace Trees

A **Wallace tree** is a combinatorial circuit used to multiply two numbers. Although it requires more hardware than shift-add multipliers, it produces a product in far less time. Instead of performing additions using standard parallel adders, Wallace trees use carry-save adders and only one parallel adder.

A carry-save adder can add three values simultaneously, instead of just two. However, it does not output a single result. Instead, it outputs both a sum and a set of carry bits. To illustrate this, consider the carry-save adder shown in Figure 8.17. Each bit S_i is the binary sum of bits X_i, Y_i, and Z_i. Carry bit C_{i+1} is the carry generated by this sum. To

FIGURE 8.17
A carry-save adder

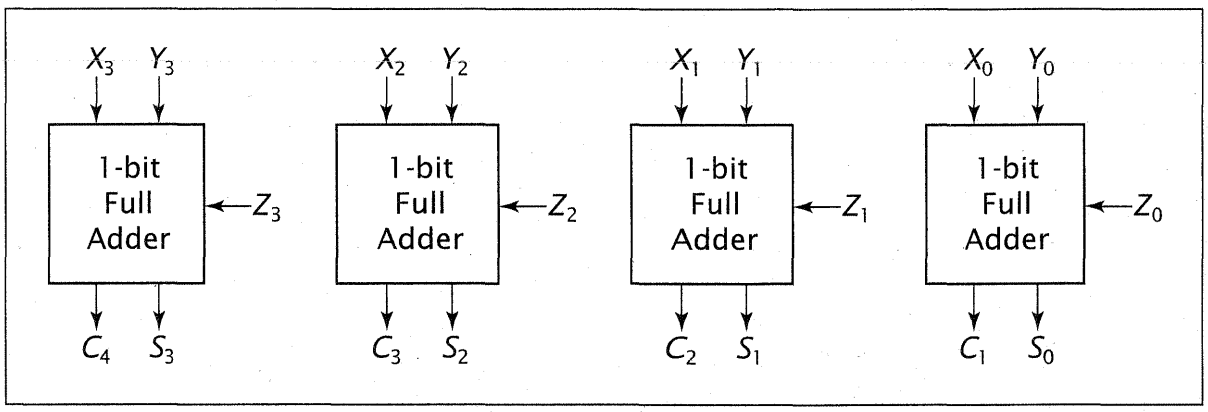
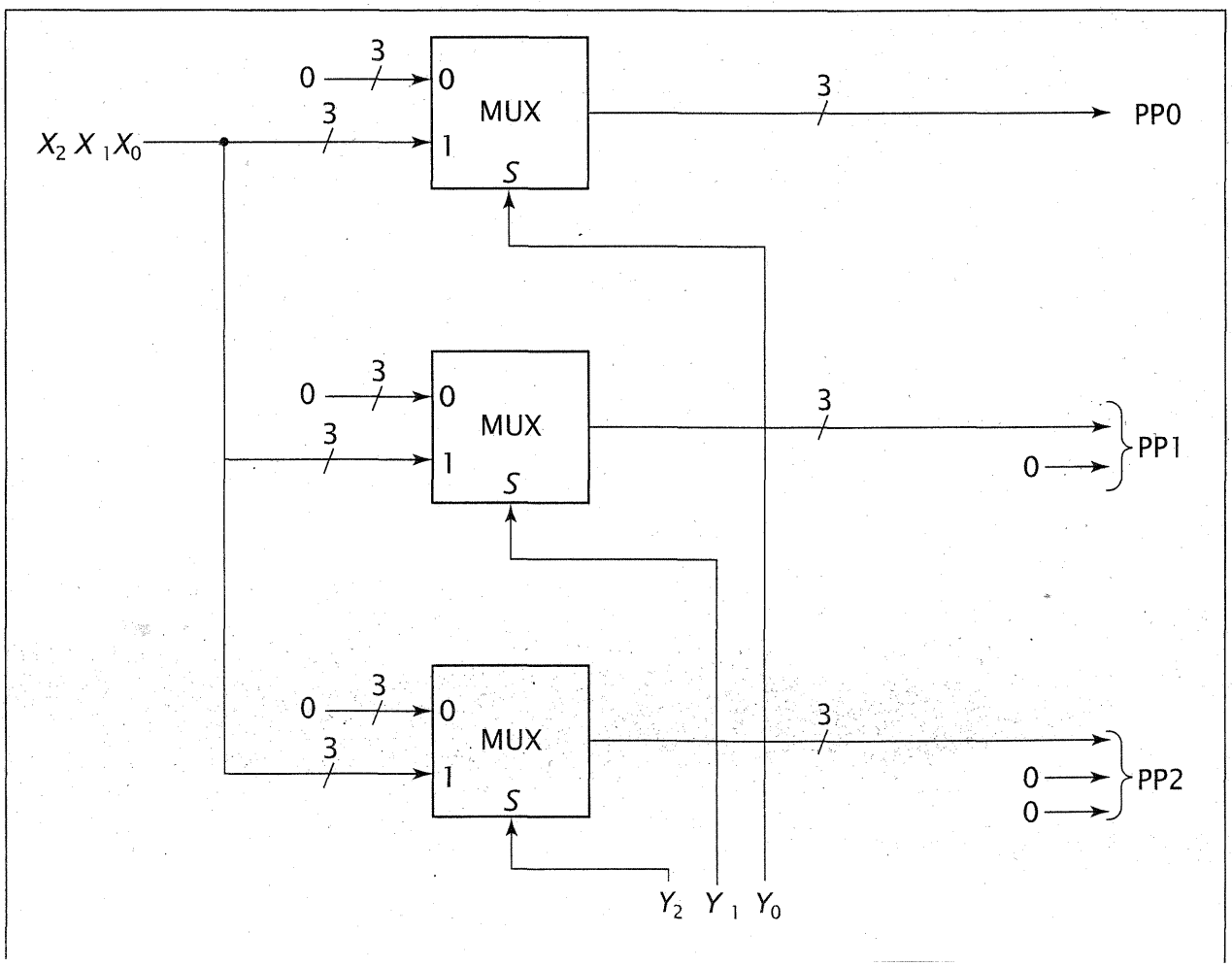


FIGURE 8.18
Generating partial products for multiplication using Wallace trees



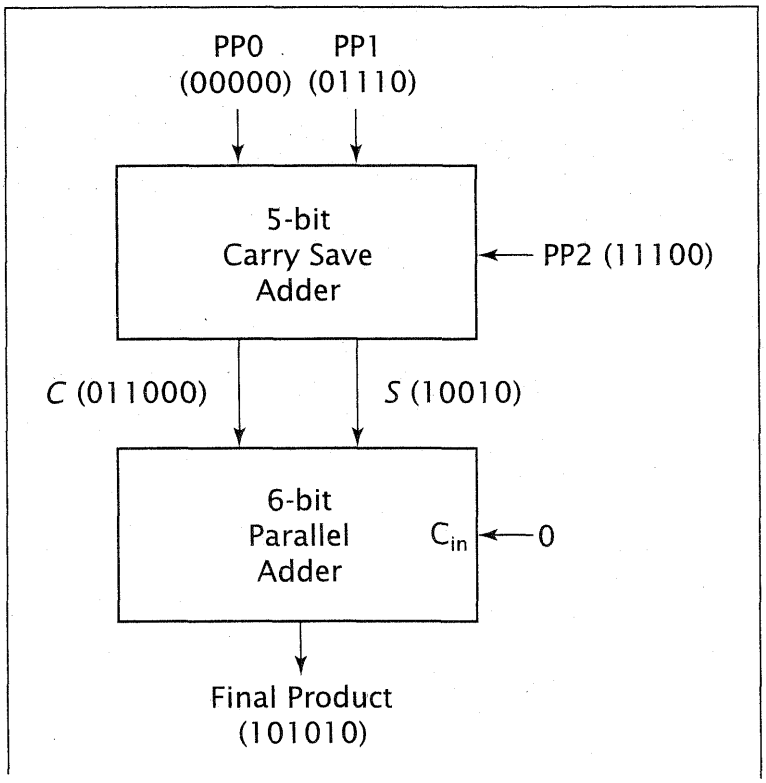
form a final sum, C and S must be added together. Because carry bits do not propagate through the adder, it is faster than a parallel adder. In a parallel adder, adding 1111 and 0001 generates a carry that propagates from the least significant bit, through each bit of the sum, to the output carry. Unlike the parallel adder, though, the carry-save adder does not produce a final sum.

Consider a numeric example. For $X = 0111$, $Y = 1011$, and $Z = 0010$, this carry-save adder would output $S = 1110$ and $C = 00110$. Note that C appears to be shifted one position to the left of S since the adder that generates S_i also generates C_{i+1}. Adding S and C using a parallel adder produces the result 10100 (20), which is 0111 (7) + 1011 (11) + 0010 (2).

To use a carry-save adder to perform multiplication, we first calculate the partial products of the multiplication, then input them to the carry-save adder. For example,

$$
\begin{array}{rl}
x = & 111 \\
y = & \underline{110} \\
 & 000 \quad \leftarrow PP0 \\
 & 111 \quad \leftarrow PP1 \\
 & \underline{111} \quad \leftarrow PP2 \\
 & 101010 \quad \leftarrow \textit{Final sum calculated}
\end{array}
$$

FIGURE 8.19
A 3 × 3 multiplier constructed using a carry-save adder



The partial products can be calculated by using the bits of y to select either the value x or 0, shifting the result to the left to properly align the partial products. In this example, *PP2* is actually 11100, the 111 value of x shifted two positions to the left. It is shifted two places because bit y_2 was used to generate this partial product. Similarly, *PP1* was set by y_1, so it is only shifted one position to the left. Figure 8.18 on page 354 shows one way to generate the partial products for this example.

We can use a 5-bit carry save adder to add the partial products *PP0*, *PP1*, and *PP2*. We then add its *S* and *C* outputs using a parallel adder to produce the final product. The hardware to perform this multiplication is shown in Figure 8.19 on page 355. Note that we incorporate the leading and trailing zeroes into the partial products to align the numbers properly.

Although this is technically a minimal Wallace tree, it does not fully explain the design principle. Figure 8.20 shows a Wallace tree for multiplying two 4-bit numbers. The first carry-save adder adds the first three partial products. The second carry-save adder adds the fourth partial product to the *S* and *C* outputs of the first adder. Finally, a parallel adder generates the product.

FIGURE 8.20
A 4 × 4 Wallace tree multiplier

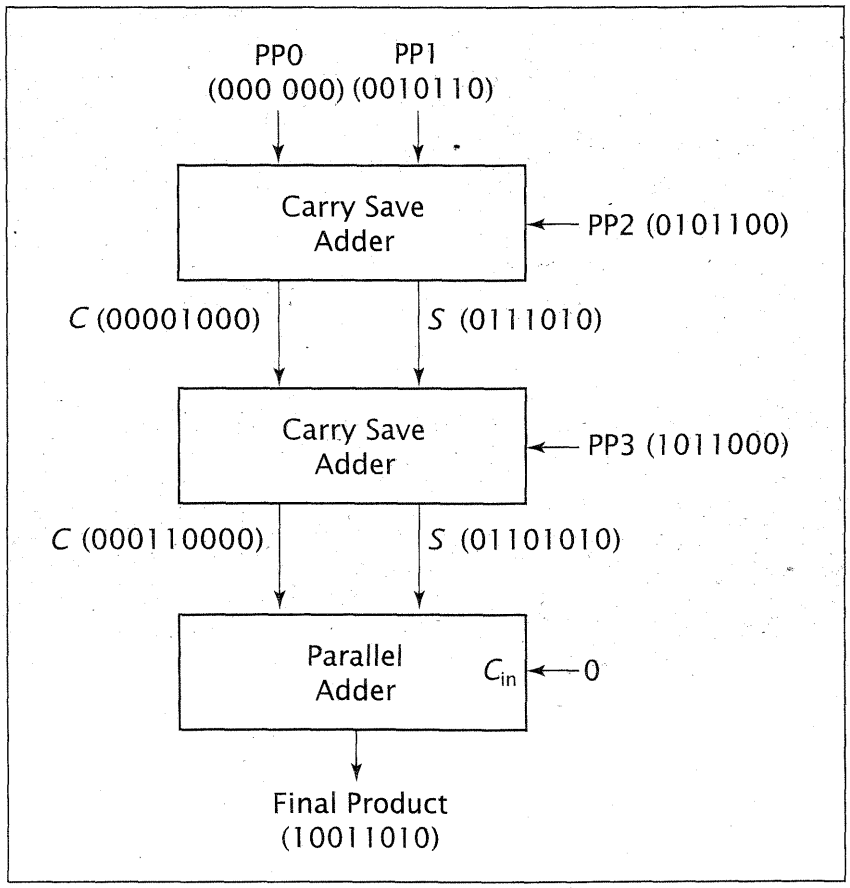
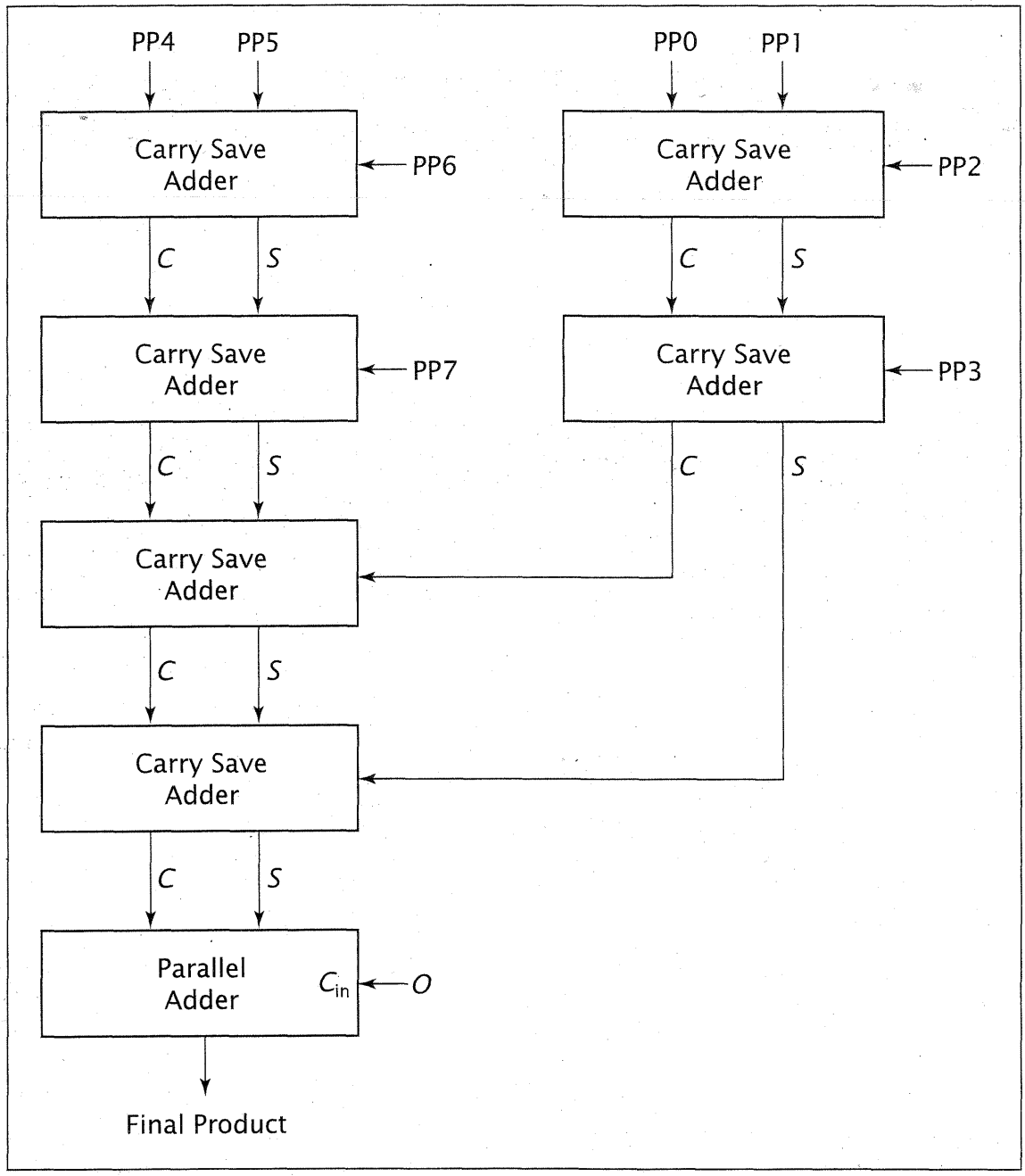


FIGURE 8.21
An 8 × 8 Wallace tree multiplier



Consider the multiplication 1011 · 1110, which has partial products *PP0* = 000 0000, *PP1* = 001 0110, *PP2* = 010 1100, and *PP3* = 101 1000. The outputs of the first carry-save adder are S = 011 1010 and C = 0000 1000. The second carry-save adder generates S = 0110 1010 and C = 0 0011 0000, and the parallel adder outputs the final product 1001 1010.

The Wallace tree makes use of parallel operations for larger numbers of partial products, which occur when multiplying numbers with

more bits. Figure 8.21 shows a Wallace tree for multiplying two 8-bit numbers.

8.5 Floating Point Numbers

All of the numeric formats presented in this chapter so far are fixed-point formats. In practice, they only represent integers, not fractions. However, a general purpose computer must be able to work with fractions with varying numbers of bits, or it will not be very useful. Clearly, it is desirable to allow a computer to use fractions for certain applications. This is the purpose of **floating point numbers**.

In this section we examine floating point numbers. We review their formats and how special numbers, such as 0, are handled. Characteristics of floating point numbers, such as precision and range, are studied. Finally, we present arithmetic algorithms for this numeric format.

8.5.1 Numeric Format

Floating point format is very similar to scientific notation. Recall that a number expressed in scientific notation has a sign, a fraction or *significand* (often erroneously called a *mantissa*), and an exponent. For example, the number -1234.5678 could be expressed as -1.2345678×10^3, where the sign is negative, the significand is 1.2345678, and the exponent is 3. This example uses base 10, but any base can be used to represent floating point numbers. In this section, all algorithms use base two because computers generally use base two to represent floating point numbers.

One disadvantage of scientific notation is that most numbers can be expressed in many different ways. For example, $-1234.5678 = -1.2345678 \times 10^3 = -1234567.8 \times 10^{-3}$, as well as several other representations. Computers are more efficient, and have much simpler hardware, if each number can have only one unique representation.

For this reason, floating point numbers must be **normalized**—that is, each number's significand is a fraction with no leading zeroes. Thus the only valid floating point representation for -1234.5678 is $-.12345678 \times 10^4$. (There is one exception to this rule; see the description of the IEEE 754 floating point standard in Section 8.6.)

This normalized representation works well for every possible number except zero. Since the number zero has only zeroes in its significand, it cannot be normalized. For this reason, a special value is assigned to zero; arithmetic algorithms must explicitly check for zero values and treat them as special cases. Positive and negative infinity also have special representations and require special treatment.

Another value that requires a special representation is **Not a Number**, or **NaN**. This value represents the result of illegal operations, such as $\infty \div \infty$ or taking the square root of a negative number. Also, compilers may assign the value NaN to uninitialized variables.