

The AND gates labeled A_1 and A_2 detect the conditions under which the first-level sum matches the patterns $11XX_2$ and $1X1X_2$. These are exactly the cases in which this sum exceeds 9. When carry-out is asserted, the XOR gate and the adders in the second row effectively add 0110_2 to the first row's sum.

There is one further case to consider. The correction factor should also be applied whenever the first-row sum exceeds 15. We saw such an example with the sum of 9 and 7 above. This case is easy to detect: the carry-out of the first-row adders will be asserted.

Thus the sum exceeds 9 if either the first-row carry-out is asserted, or the sum matches the pattern $11XX_2$, or the sum matches the pattern $1X1X_2$. These are precisely the inputs to the OR gate that computes the BCD carry-out.

A BCD adder requires over 50% more hardware than a comparable binary adder. Since faster binary adders are now available, it is no surprise that they have replaced BCD adders in almost all applications.

5.5 Combinational Multiplier

In this section we look at the design of multiplier circuitry. The methods we introduce are combinational, although alternative methods based on circuits with state are also possible. However, the fastest circuits for multiplication use just the techniques we will be discussing here.

Basic Concept Throughout this section, we will look only at multiplication techniques for unsigned numbers. Alternatively, the hardware we present is suitable for sign and magnitude multiplication, but we concentrate on the manipulation of the magnitude part. Recall that the two numbers involved in a multiplication are called the *multiplicand* and the *multiplier*.

The process of binary multiplication is best illustrated with an example. In this case, the multiplicand is 1101_2 (13) and the multiplier is 1011_2 (11):

$$\begin{array}{r}
 \text{multiplicand} \quad 1101 \quad (13) \\
 \text{multiplier} \quad * 1011 \quad (11) \\
 \hline
 \quad \quad \quad 1101 \\
 \quad \quad 1101 \\
 \quad 0000 \\
 \underline{1101} \\
 10001111 \quad (143) \\
 \swarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \searrow \\
 128 + 8 + 4 + 2 + 1 = 143
 \end{array}$$

Each bit of the multiplier is multiplied against the multiplicand, the product is aligned according to the position of the bit within the multiplier, and the resulting products are then summed to form the final result. One attraction of binary multiplication is how easy it is to form these

intermediate products: if the multiplier bit is a 1, the product is an appropriately shifted copy of the multiplicand; if the multiplier bit is a 0, the product is simply 0.

For an n -bit multiplicand and multiplier, the resulting product will be $2n$ bits. Stated differently, the product of 2^n and 2^n is $2^{n+n} = 2^{2n}$, a $2n$ -bit number. Thus, the product of two 4-bit numbers requires 8 bits, of two 8-bit numbers requires 16 bits, and so on.

Partial Product Accumulation We can construct a combinational circuit that directly implements the process described by the preceding example. The method is called *partial product accumulation*.

First, we rewrite the multiplicand bits as A_3, A_2, A_1, A_0 and the multiplier bits as B_3, B_2, B_1, B_0 . The multiplication of A and B becomes

$$\begin{array}{r}
 \begin{array}{cccc}
 A_3 & A_2 & A_1 & A_0 \\
 B_3 & B_2 & B_1 & B_0 \\
 \hline
 A_3 \cdot B_0 & A_2 \cdot B_0 & A_1 \cdot B_0 & A_0 \cdot B_0 \\
 A_3 \cdot B_1 & A_2 \cdot B_1 & A_1 \cdot B_1 & A_0 \cdot B_1 \\
 A_3 \cdot B_2 & A_2 \cdot B_2 & A_1 \cdot B_2 & A_0 \cdot B_2 \\
 A_3 \cdot B_3 & A_2 \cdot B_3 & A_1 \cdot B_3 & A_0 \cdot B_3 \\
 \hline
 S_6 & S_5 & S_4 & S_3 & S_2 & S_1 & S_0
 \end{array}
 \end{array}$$

Each of the ANDed terms is called a *partial product*. The resulting product is formed by accumulating down the columns of partial products, propagating the carries from the rightmost columns to the left.

A combinational circuit for implementing the 4-bit multiplier is shown in Figure 5.28. The first level of 16 AND gates computes the individual partial products. The second- and third-level logic blocks form the accumulation of the products on a column-by-column basis. The column sums are formed by a mixture of cascaded half adders and full adders.

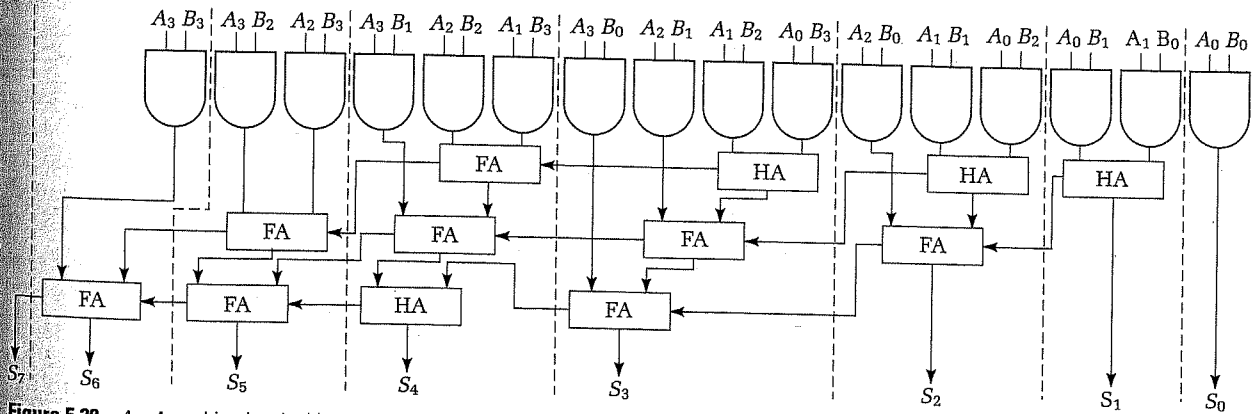


Figure 5.28 4 × 4 combinational adder.

ich
are
is
add
uld
an
the
ed,
ern
the
ible
sur-
eth-
ised
; for
lica-
we
con-
two
and
aple.
(11):
l, the
multi-
esult.
these

In the figure, inputs from the top are the bits to be added and the input from the right is the carry-in. The output from the bottom is the sum and to the left is the carry-out.

To see how the partial products are accumulated, let's look at the circuit of Figure 5.28 in a little more detail. S_1 is the straightforward sum of just two partial products, $A_1 \cdot B_0$ and $A_0 \cdot B_1$. S_2 is the sum of three products. We implement this with two cascaded adders, one of which takes the carry-out from S_1 's column.

S_3 is a little more complicated, because there are two different carry-outs from the previous column. We use three cascaded adders, two full adders and one half adder, to implement the sum. The two carry-outs from S_2 are accumulated through the carry-in inputs of the two full adders.

S_4 is the sum of three products and three possible carry-outs from S_3 . The carries make this case more complicated than the S_2 sum. The solution is to implement the sum with three adders—two full adders and one half adder. The two full adders sum the three products and two of the carry-outs. The half adder adds to this result the third possible carry-in.

The logic for S_5 is similar. Here we must sum two products and three carries. Two full adders do the job. Note that the second full adder sums two of the three carries from the previous column with the result of the first full adder. A similar analysis applies to S_7 and S_6 .

The delay through the multiplier is determined by the ripple carries between the adders. We can use a carry lookahead scheme to reduce these delays.

Clearly, the full combinational multiplier uses a lot of hardware. The dominating costs are the adders—four half adders and eight full adders. To simplify the implementation slightly, a designer may choose to use full adders for all of the adder blocks, setting the carry input to 0 where the half adder function is required. Given the full adder schematic of Figure 5.7, this is 12 adders of six gates each, for a total of 72 gates. When we add to this the 16 gates forming the partial products, the total for the whole circuit is 88 gates. It is easy to see that combinational multipliers can be justified only for the most high performance of applications.

A slightly different implementation of the 4-by-4 combinational multiplier is shown in Figure 5.29. Figure 5.29(a) gives the basic building block, a full adder circuit that sums a locally computed partial product ($X \cdot Y$), an input passed into the block from above (*Sum In*), and a carry passed from a block diagonally above. It generates a carry-out (*Cout*) and a new sum (*Sum Out*). Figure 5.29(b) shows the interconnection of 16 of these blocks to implement the full multiplier function. The A_i values are distributed along block diagonals and the B_j values are passed along rows. This implementation uses the same gate count as the previous one: 16 AND gates and 12 adders (the top row does not need adders).

TTL Multiplier Components The TTL components 74284 and 74285 provide a two-chip implementation of a 4-by-4 parallel binary multiplier. Figure 5.30 illustrates their use. The 74284 component implements the

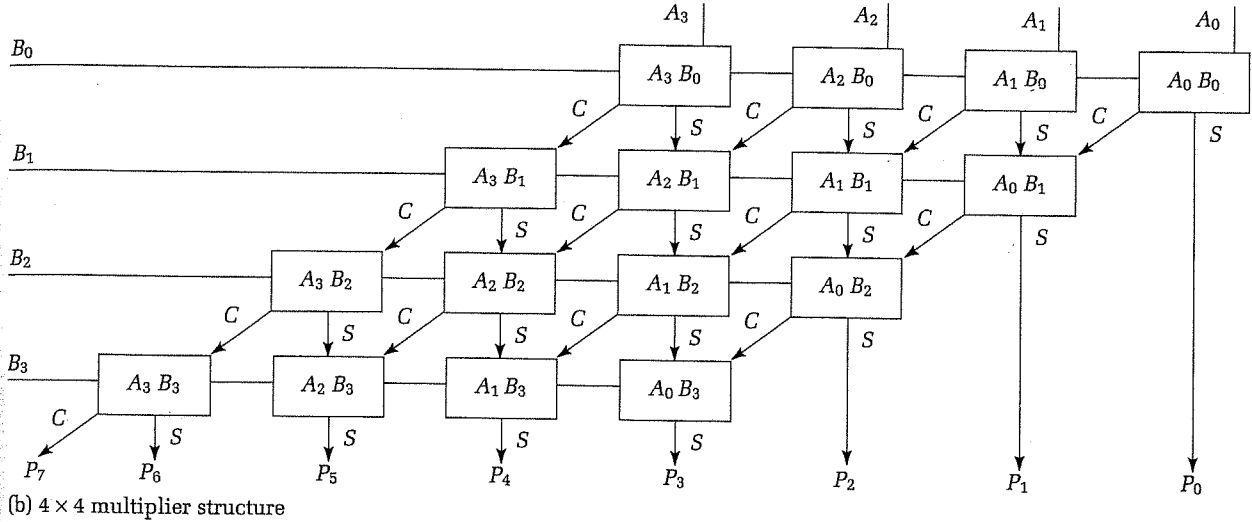
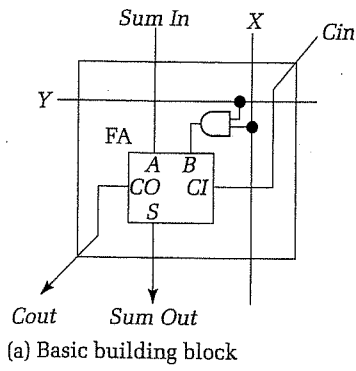


Figure 5.29 4 x 4 combinational multiplier.

high-order 4 bits of the product, while the 74285 implements the low-order bits. Both chips have dual active low output enable signals, \overline{G}_A and \overline{G}_B . When both enables are 0, the outputs are valid. Otherwise, they are in the high-impedance state.

Case Study

5.6 An 8-by-8 Bit Multiplier

In this section, we will see how to apply the principles and components of arithmetic circuits to implement a subsystem of moderate complexity. Our objective is to design a fast 8-by-8 bit multiplier using 4-by-4 bit multipliers as building blocks, along with adders, arithmetic logic, and carry lookahead units.

it
n
r-
n
e
h
y-
ill
m
S₃.
on
alf
ry-
nd
ler
ult
be-
ays.
The
To
full
half
5.7,
d to
cuit
fied
ulti-
ock,
, an
from
sum
ocks
uted
nple-
; and
pro-
plier.
s the