to the expression

$$f_3 = \overline{x}_2\overline{x}_4 + \overline{x}_1 x_3 + x_2 x_3 x_4$$

We derived the same expression in Figure 4.7.

### *n*-Dimensional Cube

A function that has $n$ variables can be mapped onto an $n$-dimensional cube. Although it is impractical to draw graphical images of cubes that have more than four variables, it is not difficult to extend the ideas introduced above to a general $n$-variable case. Because visual interpretation is not possible and because we normally use the word *cube* only for a three-dimensional structure, many people use the word *hypercube* to refer to structures with more than three dimensions. We will continue to use the word *cube* in our discussion.

It is convenient to refer to a cube as being of a certain *size* that reflects the number of vertices in the cube. Vertices have the smallest size. Each variable has a value of 0 or 1 in a vertex. A cube that has an x in one variable position is larger because it consists of two vertices. For example, the cube 1x01 consists of vertices 1001 and 1101. A cube that has two x's consists of four vertices, and so on. A cube that has $k$ x's consists of $2^k$ vertices.

An $n$-dimensional cube has $2^n$ vertices. Two vertices are adjacent if they differ in the value of only one coordinate. Because there are $n$ coordinates (axes in the $n$-dimensional cube), each vertex is adjacent to $n$ other vertices. The $n$-dimensional cube contains cubes of lower dimensionality. Cubes of the lowest dimension are vertices. Because their dimension is zero, we will call them 0-*cubes*. Edges are cubes of dimension 1; hence we will call them 1-*cubes*. A side of a three-dimensional cube is a 2-*cube*. An entire three-dimensional cube is a 3-*cube*, and so on. In general, we will refer to a set of $2^k$ adjacent vertices as a $k$-*cube*.

From the examples in Figures 4.34 and 4.35, it is apparent that the largest possible $k$-*cubes* that exist for a given function are equivalent to its prime implicants. Next, we will discuss minimization techniques that use the cubical representation of functions.

## 4.9 A Tabular Method for Minimization

Cubical representation of logic functions is well suited for implementation of minimization algorithms that can be programmed and run efficiently on computers. Such algorithms are included in modern CAD tools. While the CAD tools can be used effectively without detailed knowledge of how their minimization algorithms are implemented, the reader may find it interesting to gain some insight into how this may be accomplished. In this section we will describe a relatively simple tabular method, which illustrates the main concepts and indicates some of the problems that arise.

A tabular approach for minimization was proposed in the 1950s by Willard Quine [6] and Edward McCluskey [7]. It became popular under the name *Quine-McCluskey method*. While it is not efficient enough to be used in modern CAD tools, it is a simple method that illustrates the key issues. We will present it using the cubical notation discussed in section 4.8.

### 4.9.1 Generation of Prime Implicants

As mentioned in section 4.8, the prime implicants of a given logic function $f$ are the largest possible $k$-cubes for which $f = 1$. For incompletely specified functions, which include a set of don't-care vertices, the prime implicants are the largest $k$-cubes for which either $f = 1$ or $f$ is unspecified.

Assume that the initial specification of $f$ is given in terms of minterms for which $f = 1$. Also, let the don't-cares be specified as minterms. This allows us to create a list of vertices for which either $f = 1$ or it is a don't-care condition. We can compare these vertices in pairwise fashion to see if they can be combined into larger cubes. Then we can attempt to combine these new cubes into still larger cubes and continue the process until we find the prime implicants.

The basis of the method is the combining property of Boolean algebra

$$x_i x_j + x_i \overline{x}_j = x_i$$

which we used in section 4.8 to develop the cubical representation. If we have two cubes that are identical in all variables (coordinates) except one, for which one cube has the value 0 and the other has 1, then these cubes can be combined into a larger cube. For example, consider $f(x_1, \ldots, x_4) = \{1000, 1001, 1010, 1011\}$. The cubes 1000 and 1001 differ only in variable $x_4$; they can be combined into a new cube 100x. Similarly, 1010 and 1011 can be combined into 101x. Then we can combine 100x and 101x into a larger cube 10xx, which means that the function can be expressed simply as $f = x_1\overline{x}_2$.

Figure 4.36 shows how we can generate the prime implicants for the function, $f$, in Figure 4.11. The function is defined as

$$f(x_1, \ldots, x_4) = \sum m(0, 4, 8, 10, 11, 12, 13, 15)$$

There are no don't-care conditions. Since larger cubes can be generated only from the minterms that differ in just one variable, we can reduce the number of pairwise comparisons by placing the minterms into groups such that the cubes in each group have the same number

List 1

| | | |
|---|---|---|
| 0 | 0 0 0 0 | ✓ |
| 4 | 0 1 0 0 | ✓ |
| 8 | 1 0 0 0 | ✓ |
| 10 | 1 0 1 0 | ✓ |
| 12 | 1 1 0 0 | ✓ |
| 11 | 1 0 1 1 | ✓ |
| 13 | 1 1 0 1 | ✓ |
| 15 | 1 1 1 1 | ✓ |

List 2

| | | |
|---|---|---|
| 0,4 | 0 x 0 0 | ✓ |
| 0,8 | x 0 0 0 | ✓ |
| 8,10 | 1 0 x 0 | |
| 4,12 | x 1 0 0 | ✓ |
| 8,12 | 1 x 0 0 | ✓ |
| 10,11 | 1 0 1 x | |
| 12,13 | 1 1 0 x | |
| 11,15 | 1 x 1 1 | |
| 13,15 | 1 1 x 1 | |

List 3

| | |
|---|---|
| 0,4,8,12 | x x 0 0 |

**Figure 4.36** Generation of prime implicants for the function in Figure 4.11.

of 1s, and sort the groups by the number of 1s. Thus, it will be necessary to compare each cube in a given group only with all cubes in the immediately preceding group. In Figure 4.36, the minterms are ordered in this way in list 1. (Note that we indicated the decimal equivalents of the minterms as well, to facilitate our discussion.) The minterms, which are also called 0-cubes as explained in section 4.8, can be combined into 1-cubes shown in list 2. To make the entries easily understood we indicated the minterms that are combined to form each 1-cube. Next, we check if the 0-cubes are included in the 1-cubes and place a check mark beside each cube that is included. We now generate 2-cubes from the 1-cubes in list 2. The only 2-cube that can be generated is xx00, which is placed in list 3. Again, the check marks are placed against the 1-cubes that are included in the 2-cube. Since there exists just one 2-cube, there can be no 3-cubes for this function. The cubes in each list without a check mark are the prime implicants of $f$. Therefore, the set, $P$, of prime implicants is

$$\begin{aligned} P &= \{10x0,\ 101x,\ 110x,\ 1x11,\ 11x1,\ xx00\} \\ &= \{p_1, p_2, p_3, p_4, p_5, p_6\} \end{aligned}$$

## 4.9.2 Determination of a Minimum Cover

Having generated the set of all prime implicants, it is necessary to choose a minimum-cost subset that covers all minterms for which $f = 1$. As a simple measure we will assume that the cost is directly proportional to the number of inputs to all gates, which means to the number of literals in the prime implicants chosen to implement the function.

To find a minimum-cost cover, we construct a *prime implicant cover table* in which there is a row for each prime implicant and a column for each minterm that must be covered. Then we place check marks to indicate the minterms covered by each prime implicant. Figure 4.37*a* shows the table for the prime implicants derived in Figure 4.36. If there is a single check mark in some column of the cover table, then the prime implicant that covers the minterm of this column is *essential* and it must be included in the final cover. Such is the case with $p_6$, which is the only prime implicant that covers minterms 0 and 4. The next step is to remove the row(s) corresponding to the essential prime implicants and the column(s) covered by them. Hence we remove $p_6$ and columns 0, 4, 8, and 12, which leads to the table in Figure 4.37*b*.

Now, we can use the concept of *row dominance* to reduce the cover table. Observe that $p_1$ covers only minterm 10 while $p_2$ covers both 10 and 11. We say that $p_2$ *dominates* $p_1$. Since the cost of $p_2$ is the same as the cost of $p_1$, it is prudent to choose $p_2$ rather than $p_1$, so we will remove $p_1$ from the table. Similarly, $p_5$ dominates $p_3$, hence we will remove $p_3$ from the table. Thus, we obtain the table in Figure 4.37*c*. This table indicates that we must choose $p_2$ to cover minterm 10 and $p_5$ to cover minterm 13, which also takes care of covering minterms 11 and 15. Therefore, the final cover is

$$\begin{aligned} C &= \{p_2, p_5, p_6\} \\ &= \{101x,\ 11x1,\ xx00\} \end{aligned}$$

| Prime implicant | Minterm | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 4 | 8 | 10 | 11 | 12 | 13 | 15 |
| $p_1$ = 1 0 x 0 | | | ✓ | ✓ | | | | |
| $p_2$ = 1 0 1 x | | | | ✓ | ✓ | | | |
| $p_3$ = 1 1 0 x | | | | | | ✓ | ✓ | |
| $p_4$ = 1 x 1 1 | | | | | ✓ | | | ✓ |
| $p_5$ = 1 1 x 1 | | | | | | | ✓ | ✓ |
| $p_6$ = x x 0 0 | ✓ | ✓ | ✓ | | | ✓ | | |

(a) Initial prime implicant cover table

| Prime implicant | Minterm | | | |
|---|---|---|---|---|
| | 10 | 11 | 13 | 15 |
| $p_1$ | ✓ | | | |
| $p_2$ | ✓ | ✓ | | |
| $p_3$ | | | ✓ | |
| $p_4$ | | ✓ | | ✓ |
| $p_5$ | | | ✓ | ✓ |

(b) After the removal of essential prime implicants

| Prime implicant | Minterm | | | |
|---|---|---|---|---|
| | 10 | 11 | 13 | 15 |
| $p_2$ | ✓ | ✓ | | |
| $p_4$ | | ✓ | | ✓ |
| $p_5$ | | | ✓ | ✓ |

(c) After the removal of dominated rows

**Figure 4.37** Selection of a cover for the function in Figure 4.11.

which means that the minimum-cost implementation of the function is

$$f = x_1\overline{x}_2x_3 + x_1x_2x_4 + \overline{x}_3\overline{x}_4$$

This is the same expression as the one derived in section 4.2.2.

In this example we used the concept of row dominance to reduce the cover table. We removed the dominated rows because they cover fewer minterms and the cost of their prime implicants is the same as the cost of the prime implicants of the dominating rows. However, a dominated row should not be removed if the cost of its prime implicant is less than the cost of the dominating row's prime implicant. An example of this situation can be found in problem 4.25.

The tabular method can be used with don't-care conditions as illustrated in the following example.

---

**Example 4.14**

The don't-care minterms are included in the initial list in the same way as the minterms for which $f = 1$. Consider the function

$$f(x_1, \ldots, x_4) = \sum m(0, 2, 5, 6, 7, 8, 9, 13) + D(1, 12, 15)$$

We encourage the reader to derive a Karnaugh map for this function as an aid in visualizing the derivation that follows. Figure 4.38 depicts the generation of prime implicants, producing the result

$$\begin{aligned} P &= \{00x0, 0x10, 011x, x00x, xx01, 1x0x, x1x1\} \\ &= \{p_1, p_2, p_3, p_4, p_5, p_6, p_7\} \end{aligned}$$

The initial prime implicant cover table is shown in Figure 4.39*a*. The don't-care minterms are not included in the table because they do not have to be covered. There are no essential prime implicants. Examining this table, we see that column 8 has check marks in the same rows as column 9. Moreover, column 9 has an additional check mark in row $p_5$. Hence column 9 dominates column 8. We refer to this as the concept of *column dominance*. When one column dominates another, we can remove the dominating column, which is

List 1

| | | |
|---|---|---|
| 0 | 0 0 0 0 | ✓ |
| 1 | 0 0 0 1 | ✓ |
| 2 | 0 0 1 0 | ✓ |
| 8 | 1 0 0 0 | ✓ |
| 5 | 0 1 0 1 | ✓ |
| 6 | 0 1 1 0 | ✓ |
| 9 | 1 0 0 1 | ✓ |
| 12 | 1 1 0 0 | ✓ |
| 7 | 0 1 1 1 | ✓ |
| 13 | 1 1 0 1 | ✓ |
| 15 | 1 1 1 1 | ✓ |

List 2

| | | |
|---|---|---|
| 0,1 | 0 0 0 x | ✓ |
| 0,2 | 0 0 x 0 | |
| 0,8 | x 0 0 0 | ✓ |
| 1,5 | 0 x 0 1 | ✓ |
| 2,6 | 0 x 1 0 | |
| 1,9 | x 0 0 1 | ✓ |
| 8,9 | 1 0 0 x | ✓ |
| 8,12 | 1 x 0 0 | ✓ |
| 5,7 | 0 1 x 1 | ✓ |
| 6,7 | 0 1 1 x | |
| 5,13 | x 1 0 1 | ✓ |
| 9,13 | 1 x 0 1 | ✓ |
| 12,13 | 1 1 0 x | ✓ |
| 7,15 | x 1 1 1 | ✓ |
| 13,15 | 1 1 x 1 | ✓ |

List 3

| | |
|---|---|
| 0,1,8,9 | x 0 0 x |
| 1,5,9,13 | x x 0 1 |
| 8,9,12,13 | 1 x 0 x |
| 5,7,13,15 | x 1 x 1 |

**Figure 4.38** Generation of prime implicants for the function in Example 4.14.

| Prime implicant | Minterm | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 2 | 5 | 6 | 7 | 8 | 9 | 13 |
| $p_1$ = 0 0 x 0 | ✓ | ✓ | | | | | | |
| $p_2$ = 0 x 1 0 | | ✓ | | ✓ | | | | |
| $p_3$ = 0 1 1 x | | | | ✓ | ✓ | | | |
| $p_4$ = x 0 0 x | ✓ | | | | | ✓ | ✓ | |
| $p_5$ = x x 0 1 | | | ✓ | | | | ✓ | ✓ |
| $p_6$ = 1 x 0 x | | | | | | ✓ | ✓ | ✓ |
| $p_7$ = x 1 x 1 | | | ✓ | | ✓ | | | ✓ |

(a) Initial prime implicant cover table

| Prime implicant | Minterm | | | | | |
|---|---|---|---|---|---|---|
| | 0 | 2 | 5 | 6 | 7 | 8 |
| $p_1$ = 0 0 x 0 | ✓ | ✓ | | | | |
| $p_2$ = 0 x 1 0 | | ✓ | | ✓ | | |
| $p_3$ = 0 1 1 x | | | | ✓ | ✓ | |
| $p_4$ = x 0 0 x | ✓ | | | | | ✓ |
| $p_5$ = x x 0 1 | | | ✓ | | | |
| $p_6$ = 1 x 0 x | | | | | | ✓ |
| $p_7$ = x 1 x 1 | | | ✓ | | ✓ | |

(b) After the removal of columns 9 and 13

| Prime implicant | Minterm | | | | | |
|---|---|---|---|---|---|---|
| | 0 | 2 | 5 | 6 | 7 | 8 |
| $p_1$ | ✓ | ✓ | | | | |
| $p_2$ | | ✓ | | ✓ | | |
| $p_3$ | | | | ✓ | ✓ | |
| $p_4$ | ✓ | | | | | ✓ |
| $p_7$ | | | ✓ | | ✓ | |

(c) After the removal of rows $p_5$ and $p_6$

| Prime implicant | Minterm | |
|---|---|---|
| | 2 | 6 |
| $p_1$ | ✓ | |
| $p_2$ | ✓ | ✓ |
| $p_3$ | | ✓ |

(d) After including $p_4$ and $p_7$ in the cover

**Figure 4.39** Selection of a cover for the function in Example 4.14.

column 9 in this case. Note that this is in contrast to rows where we remove dominated (rather than dominating) rows. The reason is that when we choose a prime implicant to cover the minterm that corresponds to the dominated column, this prime implicant will also cover the minterm corresponding to the dominating column. In our example, choosing either $p_4$ or $p_6$ covers both minterms 8 and 9. Similarly, column 13 dominates column 5, hence column 13 can be deleted.

After removing columns 9 and 13, we obtain the reduced table in Figure 4.39*b*. In this table row $p_4$ dominates $p_6$ and row $p_7$ dominates $p_5$. This means that $p_5$ and $p_6$ can be removed, giving the table in Figure 4.39*c*. Now, $p_4$ and $p_7$ are essential to cover minterms 8 and 5, respectively. Thus, the table in Figure 4.39*d* is obtained, from which it is obvious that $p_2$ covers the remaining minterms 2 and 6. Note that row $p_2$ dominates both rows $p_1$ and $p_3$.

The final cover is

$$\begin{aligned} C &= \{p_2, p_4, p_7\} \\ &= \{\text{0x10, x00x, x1x1}\} \end{aligned}$$

and the function is implemented as

$$f = \overline{x}_1 x_3 \overline{x}_4 + \overline{x}_2 \overline{x}_3 + x_2 x_4$$

---

In Figures 4.37 and 4.39, we used the concept of row and column dominance to reduce the cover table. This is not always possible, as illustrated in the following example.

---

**Example 4.15**

Consider the function

$$f(x_1, \ldots, x_4) = \sum m(0, 3, 10, 15) + D(1, 2, 7, 8, 11, 14)$$

The prime implicants for this function are

$$\begin{aligned} P &= \{\text{00xx, x0x0, x01x, xx11, 1x1x}\} \\ &= \{p_1, p_2, p_3, p_4, p_5\} \end{aligned}$$

The initial prime implicant cover table is shown in Figure 4.40*a*. There are no essential prime implicants. Also, there are no dominant rows or columns. Moreover, all prime implicants have the same cost because each of them is implemented with two literals. Thus, the table does not provide any clues that can be used to select a minimum-cost cover.

A good practical approach is to use the concept of *branching*, which was introduced in section 4.2.2. We can choose any prime implicant, say $p_3$, and first choose to include this prime implicant in the final cover. Then we can determine the rest of the final cover in the usual way and compute its cost. Next we try the other possibility by excluding $p_3$ from the final cover and determine the resulting cost. We compare the costs and choose the less expensive alternative.

Figure 4.40*b* gives the cover table that is left if $p_3$ is included in the final cover. The table does not include minterms 3 and 10 because they are covered by $p_3$. The table indicates

| Prime implicant | Minterm 0 | 3 | 10 | 15 |
|---|---|---|---|---|
| $p_1$ = 0 0 x x | ✓ | ✓ | | |
| $p_2$ = x 0 x 0 | ✓ | | ✓ | |
| $p_3$ = x 0 1 x | | ✓ | ✓ | |
| $p_4$ = x x 1 1 | | ✓ | | ✓ |
| $p_5$ = 1 x 1 x | | | ✓ | ✓ |

(a) Initial prime implicant cover table

| Prime implicant | Minterm 0 | 15 |
|---|---|---|
| $p_1$ | ✓ | |
| $p_2$ | ✓ | |
| $p_4$ | | ✓ |
| $p_5$ | | ✓ |

(b) After including $p_3$ in the cover

| Prime implicant | Minterm 0 | 3 | 10 | 15 |
|---|---|---|---|---|
| $p_1$ | ✓ | ✓ | | |
| $p_2$ | ✓ | | ✓ | |
| $p_4$ | | ✓ | | ✓ |
| $p_5$ | | | ✓ | ✓ |

(c) After excluding $p_3$ from the cover

**Figure 4.40** Selection of a cover for the function in Example 4.15.

that a complete cover must include either $p_1$ or $p_2$ to cover minterm 0 and either $p_4$ or $p_5$ to cover minterm 15. Therefore, a complete cover can be

$$C = \{p_1, p_3, p_4\}$$

The alternative of excluding $p_3$ leads to the cover table in Figure 4.40*c*. Here, we see that a minimum-cost cover requires only two prime implicants. One possibility is to choose $p_1$

and $p_5$. The other possibility is to choose $p_2$ and $p_4$. Hence a minimum-cost cover is just

$$\begin{aligned} C_{min} &= \{p_1, p_5\} \\ &= \{00\text{xx}, 1\text{x}1\text{x}\} \end{aligned}$$

The function is realized as

$$f = \bar{x}_1\bar{x}_2 + x_1x_3$$

---

## 4.9.3 Summary of the Tabular Method

The tabular method can be summarized as follows:

1. Starting with a list of cubes that represent the minterms where $f = 1$ or a don't-care condition, generate the prime implicants by successive pairwise comparisons of the cubes.
2. Derive a cover table which indicates the minterms where $f = 1$ that are covered by each prime implicant.
3. Include the essential prime implicants (if any) in the final cover and reduce the table by removing both these prime implicants and the covered minterms.
4. Use the concept of row and column dominance to reduce the cover table further. A dominated row is removed only if the cost of its prime implicant is greater than or equal to the cost of the dominating row's prime implicant.
5. Repeat steps 3 and 4 until the cover table is either empty or no further reduction of the table is possible.
6. If the reduced cover table is not empty, then use the branching approach to determine the remaining prime implicants that should be included in a minimum cost cover.

The tabular method illustrates how an algebraic technique can be used to generate the prime implicants. It also shows a simple approach for dealing with the covering problem, to find a minimum-cost cover. The method has some practical limitations. In practice, functions are seldom defined in the form of minterms. They are usually given either in the form of algebraic expressions or as sets of cubes. The need to start the minimization process with a list of minterms means that the expressions or sets have to be expanded into this form. This list may be very large. As larger cubes are generated, there will be numerous comparisons performed and the computation will be slow. Using the cover table to select the optimal set of prime implicants is also computationally intensive when large functions are involved.

Many algebraic techniques have been developed, which aim to reduce the time that it takes to generate the optimal covers. While most of these techniques are beyond the scope of this book, we will briefly discuss one possible approach in the next section. A reader who intends to use the CAD tools, but is not interested in the details of automated minimization, may skip this section without loss of continuity.

Another interesting feature of XOR gates is that a two-input XOR gate can be thought of as using one input as a control signal that determines whether the true or complemented value of the other input will be passed through the gate as the output value. This is clear from the definition of XOR, where $x_i \oplus y_i = \bar{x}y + x\bar{y}$. Consider $x$ to be the control input. Then if $x = 0$, the output will be equal to the value of $y$. But if $x = 1$, the output will be equal to the complement of $y$. In the derivation above, we used algebraic manipulation to derive $s_i = (x_i \oplus y_i) \oplus c_i$. We could have obtained the same expression immediately by making the following observation. In the top half of the truth table in Figure 5.4*a*, $c_i$ is equal to 0, and the sum function $s_i$ is the XOR of $x_i$ and $y_i$. In the bottom half of the table, $c_i$ is equal to 1, while $s_i$ is the complemented version of its top half. This observation leads directly to our expression using 2 two-input XOR operations. We will encounter an important example of using XOR gates to pass true or complemented signals under the control of another signal in section 5.3.3.

In the preceding discussion we encountered the complement of the XOR operation, which we denoted as $\overline{x \oplus y}$. This operation is used so commonly that it is given the distinct name *XNOR*. A special symbol, $\odot$, is often used to denote the XNOR operation, namely

$$x \odot y = \overline{x \oplus y}$$

The XNOR is sometimes also referred to as the *coincidence* operation because it produces the output of 1 when its inputs coincide in value; that is, they are both 0 or both 1.

## 5.2.1 Decomposed Full-Adder

In view of the names used for the circuits, one can expect that a full-adder can be constructed using half-adders. This can be accomplished by creating a multilevel circuit of the type discussed in section 4.6.2. The circuit is given in Figure 5.5. It uses two half-adders to form a full-adder. The reader should verify the functional correctness of this circuit.

## 5.2.2 Ripple-Carry Adder

To perform addition by hand, we start from the least-significant digit and add pairs of digits, progressing to the most-significant digit. If a carry is produced in position $i$, then this carry is added to the operands in position $i + 1$. The same arrangement can be used in a logic circuit that performs addition. For each bit position we can use a full-adder circuit, connected as shown in Figure 5.6. Note that to be consistent with the customary way of writing numbers, the least-significant bit position is on the right. Carries that are produced by the full-adders propagate to the left.

When the operands $X$ and $Y$ are applied as inputs to the adder, it takes some time before the output sum, $S$, is valid. Each full-adder introduces a certain delay before its $s_i$ and $c_{i+1}$ outputs are valid. Let this delay be denoted as $\Delta t$. Thus the carry-out from the first stage, $c_1$, arrives at the second stage $\Delta t$ after the application of the $x_0$ and $y_0$ inputs. The carry-out from the second stage, $c_2$, arrives at the third stage with a $2\Delta t$ delay, and so on. The signal $c_{n-1}$ is valid after a delay of $(n - 1)\Delta t$, which means that the complete sum is available after a delay of $n\Delta t$. Because of the way the carry signals "ripple" through the full-adder stages, the circuit in Figure 5.6 is called a *ripple-carry adder*.

The delay incurred to produce the final sum and carry-out in a ripple-carry adder depends on the size of the numbers. When 32- or 64-bit numbers are used, this delay may become unacceptably high. Because the circuit in each full-adder leaves little room for a drastic reduction in the delay, it may be necessary to seek different structures for implementation of $n$-bit adders. We will discuss a technique for building high-speed adders in section 5.4.

So far we have dealt with unsigned integers only. The addition of such numbers does not require a carry-in for stage 0. In Figure 5.6 we included $c_0$ in the diagram so that the ripple-carry adder can also be used for subtraction of numbers, as we will see in section 5.3.

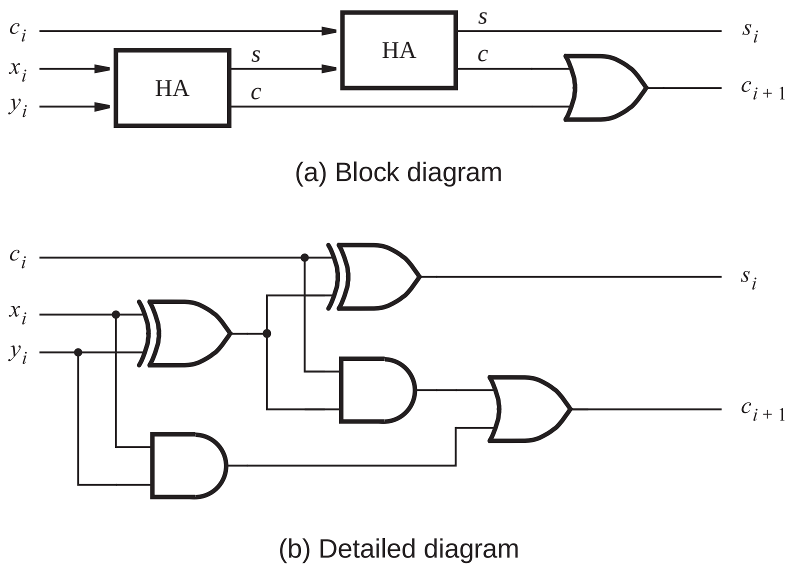


**Figure 5.5** A decomposed implementation of the full-adder circuit.

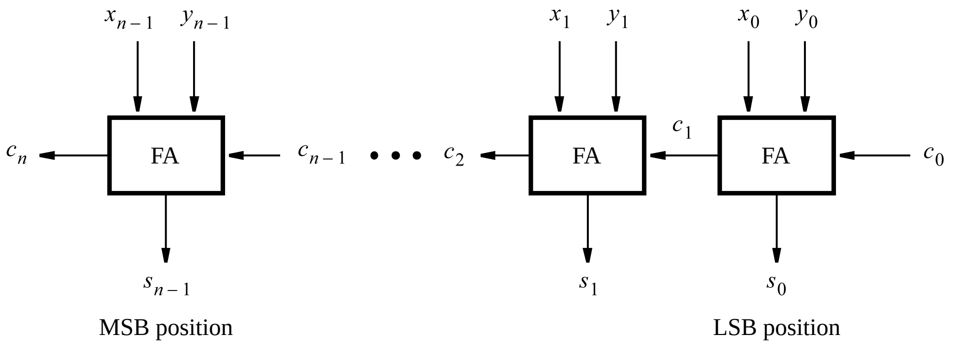


**Figure 5.6** An $n$-bit ripple-carry adder.

### 5.2.3 Design Example

Suppose that we need a circuit that multiplies an eight-bit unsigned number by 3. Let $A = a_7a_6 \cdots a_1a_0$ denote the number and $P = p_9p_8 \cdots p_1p_0$ denote the product $P = 3A$. Note that 10 bits are needed to represent the product.

A simple approach to design the required circuit is to use two ripple-carry adders to add three copies of the number $A$, as illustrated in Figure 5.7$a$. The symbol that denotes each adder is a commonly used graphical symbol for adders. The letters $x_i$, $y_i$, $s_i$, and $c_i$ indicate the meaning of the inputs and outputs according to Figure 5.6. The first adder produces $A + A = 2A$. Its result is represented as eight sum bits and the carry from the most-significant bit. The second adder produces $2A + A = 3A$. It has to be a nine-bit adder to be able to handle the nine bits of 2A, which are generated by the first adder. Because the $y_i$ inputs have to be driven only by the eight bits of $A$, the ninth input $y_8$ is connected to a constant 0.

This approach is straightforward, but not very efficient. Because $3A = 2A + A$, we can observe that $2A$ can be generated by shifting the bits of $A$ one bit-position to the left, which gives the bit pattern $a_7a_6a_5a_4a_3a_2a_1a_00$. According to equation 5.1, this pattern is equal to $2A$. Then a single ripple-carry adder suffices for implementing $3A$, as shown in Figure 5.7$b$. This is essentially the same circuit as the second adder in part ($a$) of the figure. Note that the input $x_0$ is connected to a constant 0. Note also that in the second adder in part ($a$) the value of $x_0$ is always 0, even though it is driven by the least-significant bit, $s_0$, of the sum of the first adder. Because $x_0 = y_0 = a_0$ in the first adder, the sum bit $s_0$ will be 0, whether $a_0$ is 0 or 1.

## 5.3 Signed Numbers

In the decimal system the sign of a number is indicated by a + or − symbol to the left of the most-significant digit. In the binary system the *sign* of a number is denoted by the left-most bit. For a positive number the left-most bit is equal to 0, and for a negative number it is equal to 1. Therefore, in signed numbers the left-most bit represents the sign, and the remaining $n - 1$ bits represent the magnitude, as illustrated in Figure 5.8. It is important to note the difference in the location of the most-significant bit (MSB). In unsigned numbers all bits represent the magnitude of a number; hence all $n$ bits are *significant* in defining the magnitude. Therefore, the MSB is the left-most bit, $b_{n-1}$. In signed numbers there are $n - 1$ significant bits, and the MSB is in bit position $b_{n-2}$.

### 5.3.1 Negative Numbers

Positive numbers are represented using the positional number representation as explained in the previous section. Negative numbers can be represented in three different ways: sign-and-magnitude, 1's complement, and 2's complement.

the two operands. Therefore, it should be possible to use the same adder circuit to perform both addition and subtraction.

## 5.3.3 Adder and Subtractor Unit

The only difference between performing addition and subtraction is that for subtraction it is necessary to use the 2's complement of one operand. Let $X$ and $Y$ be the two operands, such that $Y$ serves as the subtrahend in subtraction. From section 5.3.1 we know that a 2's complement can be obtained by adding 1 to the 1's complement of $Y$. Adding 1 in the least-significant bit position can be accomplished simply by setting the carry-in bit $c_0$ to 1. A 1's complement of a number is obtained by complementing each of its bits. This could be done with NOT gates, but we need a more flexible circuit where we can use the true value of $Y$ for addition and its complement for subtraction.

In section 5.2 we explained that two-input XOR gates can be used to choose between true and complemented versions of an input value, under the control of the other input. This idea can be applied in the design of the adder/subtractor unit as follows. Assume that there exists a control signal that chooses whether addition or subtraction is to be performed. Let this signal be called $\overline{\text{Add}}/\text{Sub}$. Also, let its value be 0 for addition and 1 for subtraction. To indicate this fact, we placed a bar over Add. This is a commonly used convention, where a bar over a name means that the action specified by the name is to be taken if the control signal has the value 0. Now let each bit of $Y$ be connected to one input of an XOR gate, with the other input connected to $\overline{\text{Add}}/\text{Sub}$. The outputs of the XOR gates represent $Y$ if $\overline{\text{Add}}/\text{Sub} = 0$, and they represent the 1's complement of $Y$ if $\overline{\text{Add}}/\text{Sub} = 1$. This leads to the circuit in Figure 5.13. The main part of the circuit is an $n$-bit adder, which can be implemented using the ripple-carry structure of Figure 5.6. Note that the control signal

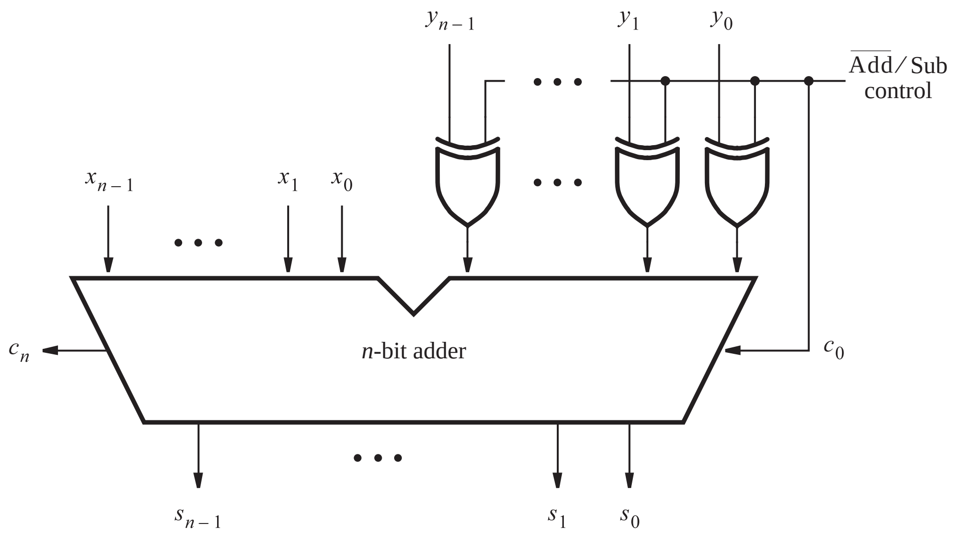


**Figure 5.13** Adder/subtractor unit.

$\overline{\text{Add}}$/Sub is also connected to the carry-in $c_0$. This makes $c_0 = 1$ when subtraction is to be performed, thus adding the 1 that is needed to form the 2's complement of $Y$. When the addition operation is performed, we will have $c_0 = 0$.

The combined adder/subtractor unit is a good example of an important concept in the design of logic circuits. It is useful to design circuits to be as flexible as possible and to exploit common portions of circuits for as many tasks as possible. This approach minimizes the number of gates needed to implement such circuits, and it reduces the wiring complexity substantially.

### 5.3.4 Radix-Complement Schemes

The idea of performing a subtraction operation by addition of a complement of the subtrahend is not restricted to binary numbers. We can gain some insight into the workings of the 2's complement scheme by considering its counterpart in the decimal number system. Consider the subtraction of two-digit decimal numbers. Computing a result such as $74 - 33 = 41$ is simple because each digit of the subtrahend is smaller than the corresponding digit of the minuend; therefore, no borrow is needed in the computation. But computing $74 - 36 = 38$ is not as simple because a borrow is needed in subtracting the least-significant digit. If a borrow occurs, the computation becomes more complicated.

Suppose that we restructure the required computation as follows

$$\begin{aligned} 74 - 36 &= 74 + 100 - 100 - 36 \\ &= 74 + (100 - 36) - 100 \end{aligned}$$

Now two subtractions are needed. Subtracting 36 from 100 still involves borrows. But noting that $100 = 99 + 1$, these borrows can be avoided by writing

$$\begin{aligned} 74 - 36 &= 74 + (99 + 1 - 36) - 100 \\ &= 74 + (99 - 36) + 1 - 100 \end{aligned}$$

The subtraction in parentheses does not require borrows; it is performed by subtracting each digit of the subtrahend from 9. We can see a direct correlation between this expression and the one used for 2's complement, as reflected in the circuit in Figure 5.13. The operation $(99 - 36)$ is analogous to complementing the subtrahend $Y$ to find its 1's complement, which is the same as subtracting each bit from 1. Using decimal numbers, we find the *9's complement* of the subtrahend by subtracting each digit from 9. In Figure 5.13 we add the carry-in of 1 to form the 2's complement of $Y$. In our decimal example we perform $(99 - 36) + 1 = 64$. Here 64 is the 10's complement of 36. For an $n$-digit decimal number, $N$, its *10's complement*, $K_{10}$, is defined as $K_{10} = 10^n - N$, while its 9's complement, $K_9$, is $K_9 = (10^n - 1) - N$.

Thus the required subtraction $(74 - 36)$ can be performed by addition of the 10's complement of the subtrahend, as in

$$\begin{aligned} 74 - 36 &= 74 + 64 - 100 \\ &= 138 - 100 \\ &= 38 \end{aligned}$$

The subtraction $138 - 100$ is trivial because it means that the leading digit in 138 is simply deleted. This is analogous to ignoring the carry-out from the circuit in Figure 5.13, as discussed for the subtraction examples in Figure 5.11.

---

**Example 5.1** Suppose that $A$ and $B$ are $n$-digit decimal numbers. Using the above 10's-complement approach, $B$ can be subtracted from $A$ as follows:

$$A - B = A + (10^n - B) - 10^n$$

If $A \geq B$, then the operation $A + (10^n - B)$ produces a carry-out of 1. This carry is equivalent to $10^n$; hence it can be simply ignored.

But if $A < B$, then the operation $A + (10^n - B)$ produces a carry-out of 0. Let the result obtained be $M$, so that

$$A - B = M - 10^n$$

We can rewrite this as

$$10^n - (B - A) = M$$

The left side of this equation is the 10's complement of $(B - A)$. The 10's complement of a positive number represents a negative number that has the same magnitude. Hence $M$ correctly represents the negative value obtained from the computation $A - B$ when $A < B$. This concept is illustrated in the examples that follow.

---

**Example 5.2** When dealing with binary signed numbers we use 0 in the left-most bit position to denote a positive number and 1 to denote a negative number. If we wanted to build hardware that operates on signed decimal numbers, we could use a similar approach. Let 0 in the left-most digit position denote a positive number and let 9 denote a negative number. Note that 9 is the 9's complement of 0 in the decimal system, just as 1 is the 1's complement of 0 in the binary system.

Thus, using three-digit signed numbers, $A = 045$ and $B = 027$ are positive numbers with magnitudes 45 and 27, respectively. The number $B$ can be subtracted from $A$ as follows

$$\begin{aligned}
A - B &= 045 - 027 \\
&= 045 + 1000 - 1000 - 027 \\
&= 045 + (999 - 027) + 1 - 1000 \\
&= 045 + 972 + 1 - 1000 \\
&= 1018 - 1000 \\
&= 018
\end{aligned}$$

This gives the correct answer of +18.

Next consider the case where the minuend has lower value than the subtrahend. This is illustrated by the computation

$$\begin{aligned}
B - A &= 027 - 045 \\
&= 027 + 1000 - 1000 - 045
\end{aligned}$$

The speed of any circuit is limited by the longest delay along the paths through the circuit. In the case of the circuit in Figure 5.13, the longest delay is along the path from the $y_i$ input, through the XOR gate and through the carry circuit of each adder stage. The longest delay is often referred to as the *critical-path delay*, and the path that causes this delay is called the *critical path*.

## 5.4 Fast Adders

The performance of a large digital system is dependent on the speed of circuits that form its various functional units. Obviously, better performance can be achieved using faster circuits. This can be accomplished by using superior (usually newer) technology in which the delays in basic gates are reduced. But it can also be accomplished by changing the overall structure of a functional unit, which may lead to even more impressive improvement. In this section we will discuss an alternative for implementation of an $n$-bit adder, which substantially reduces the time needed to add numbers.

### 5.4.1 Carry-Lookahead Adder

To reduce the delay caused by the effect of carry propagation through the ripple-carry adder, we can attempt to evaluate quickly for each stage whether the carry-in from the previous stage will have a value 0 or 1. If a correct evaluation can be made in a relatively short time, then the performance of the complete adder will be improved.

From Figure 5.4*b* the carry-out function for stage $i$ can be realized as

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

If we factor this expression as

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

then it can be written as

$$c_{i+1} = g_i + p_i c_i \qquad \textbf{[5.3]}$$

where

$$g_i = x_i y_i$$
$$p_i = x_i + y_i$$

The function $g_i$ is equal to 1 when both inputs $x_i$ and $y_i$ are equal to 1, regardless of the value of the incoming carry to this stage, $c_i$. Since in this case stage $i$ is guaranteed to generate a carry-out, $g$ is called the *generate* function. The function $p_i$ is equal to 1 when at least one of the inputs $x_i$ and $y_i$ is equal to 1. In this case a carry-out is produced if $c_i = 1$. The effect is that the carry-in of 1 is propagated through stage $i$; hence $p_i$ is called the *propagate* function.

Expanding the expression 5.3 in terms of stage $i-1$ gives

$$\begin{aligned} c_{i+1} &= g_i + p_i(g_{i-1} + p_{i-1}c_{i-1}) \\ &= g_i + p_i g_{i-1} + p_i p_{i-1} c_{i-1} \end{aligned}$$

The same expansion for other stages, ending with stage 0, gives

$$c_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \cdots + p_i p_{i-1} \cdots p_2 p_1 g_0 + p_i p_{i-1} \cdots p_1 p_0 c_0 \qquad \textbf{[5.4]}$$

This expression represents a two-level AND-OR circuit in which $c_{i+1}$ is evaluated very quickly. An adder based on this expression is called a *carry-lookahead adder*.

To appreciate the physical meaning of expression 5.4, it is instructive to consider its effect on the construction of a fast adder in comparison with the details of the ripple-carry adder. We will do so by examining the detailed structure of the two stages that add the least-significant bits, namely, stages 0 and 1. Figure 5.15 shows the first two stages of a ripple-carry adder in which the carry-out functions are implemented as indicated in expression 5.3. Each stage is essentially the circuit from Figure 5.4*c* except that an extra

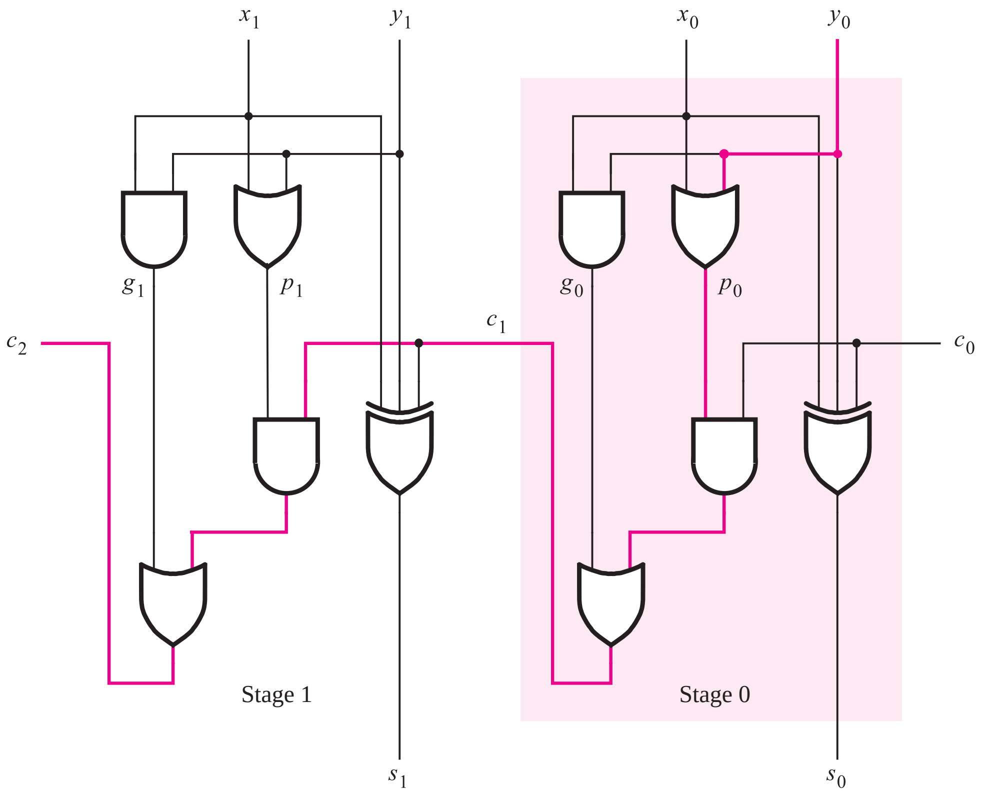


**Figure 5.15** A ripple-carry adder based on expression 5.3.

OR gate is used (which produces the $p_i$ signal), instead of an AND gate because we factored the sum-of-products expression for $c_{i+1}$.

The slow speed of the ripple-carry adder is caused by the long path along which a carry signal must propagate. In Figure 5.15 the critical path is from inputs $x_0$ and $y_0$ to the output $c_2$. It passes through five gates, as highlighted in blue. The path in other stages of an $n$-bit adder is the same as in stage 1. Therefore, the total delay along the critical path is $2n + 1$.

Figure 5.16 gives the first two stages of the carry-lookahead adder, using expression 5.4 to implement the carry-out functions. Thus

$$c_1 = g_0 + p_0 c_0$$
$$c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$$

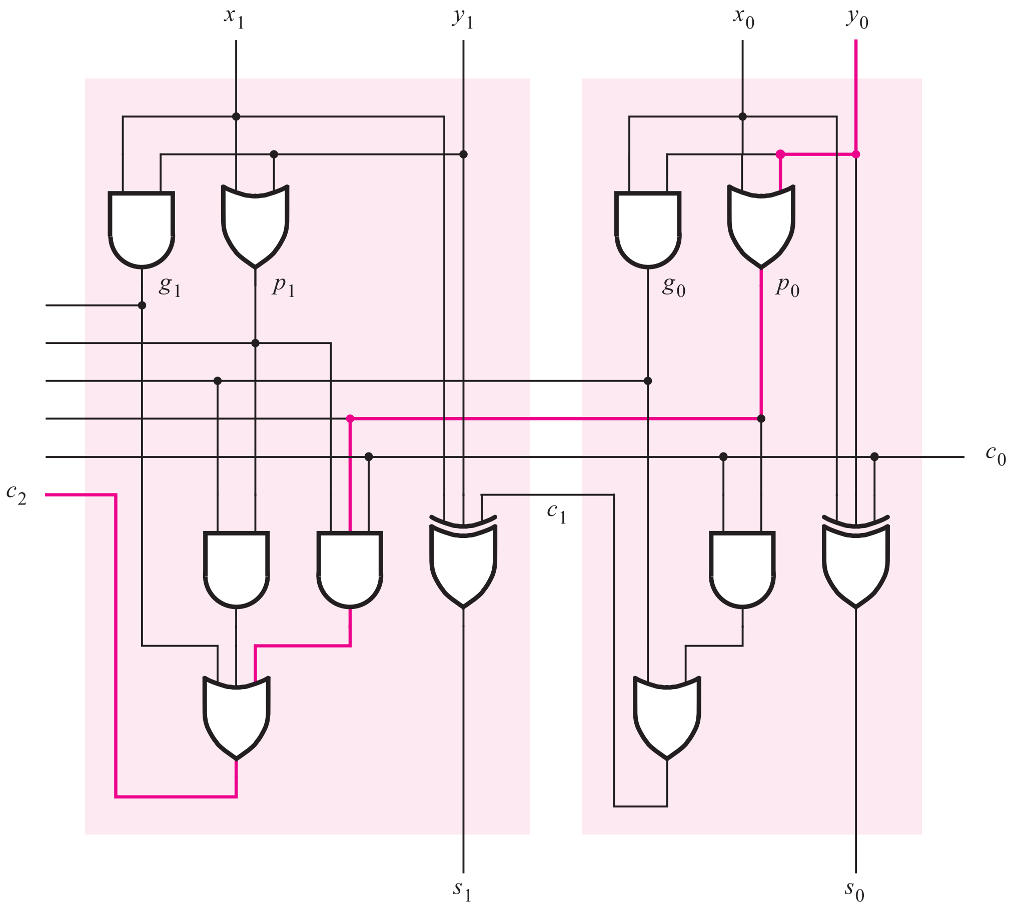


**Figure 5.16** The first two stages of a carry-lookahead adder.

The critical path for producing the $c_2$ signal is highlighted in blue. In this circuit, $c_2$ is produced just as quickly as $c_1$, after a total of three gate delays. Extending the circuit to $n$ bits, the final carry-out signal $c_n$ would also be produced after only three gate delays because expression 5.4 is just a large two-level (AND-OR) circuit.

The total delay in the $n$-bit carry-lookahead adder is four gate delays. The values of all $g_i$ and $p_i$ signals are determined after one gate delay. It takes two more gate delays to evaluate all carry signals. Finally, it takes one more gate delay (XOR) to generate all sum bits. The key to the good performance of the adder is quick evaluation of carry signals.

The complexity of an $n$-bit carry-lookahead adder increases rapidly as $n$ becomes larger. To reduce the complexity, we can use a *hierarchical* approach in designing large adders. Suppose that we want to design a 32-bit adder. We can divide this adder into 4 eight-bit blocks, such that bits $b_{7-0}$ are block 0, bits $b_{15-8}$ are block 1, bits $b_{23-16}$ are block 2, and bits $b_{31-24}$ are block 3. Then we can implement each block as an eight-bit carry-lookahead adder. The carry-out signals from the four blocks are $c_8$, $c_{16}$, $c_{24}$, and $c_{32}$. Now we have two possibilities. We can connect the four blocks as four stages in a ripple-carry adder. Thus while carry-lookahead is used within each block, the carries ripple between the blocks. This circuit is illustrated in Figure 5.17.

Instead of using a ripple-carry approach between blocks, a faster circuit can be designed in which a second-level carry-lookahead is performed to produce quickly the carry signals between blocks. The structure of this "hierarchical carry-lookahead adder" is shown in Figure 5.18. Each block in the top row includes an eight-bit carry-lookahead adder, based on generate signals, $g_i$, and propagate signals, $p_i$, for each stage in the block, as discussed before. However, instead of producing a carry-out signal from the most-significant bit of the block, each block produces generate and propagate signals for the entire block. Let $G_j$ and $P_j$ denote these signals for each block $j$. Now $G_j$ and $P_j$ can be used as inputs to a second-level carry-lookahead circuit, at the bottom of Figure 5.18, which evaluates all carries between blocks. We can derive the block generate and propagate signals for block 0 by examining the expression for $c_8$

$$\begin{aligned} c_8 = {} & g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4 + p_7p_6p_5p_4g_3 + p_7p_6p_5p_4p_3g_2 \\ & + p_7p_6p_5p_4p_3p_2g_1 + p_7p_6p_5p_4p_3p_2p_1g_0 + p_7p_6p_5p_4p_3p_2p_1p_0c_0 \end{aligned}$$

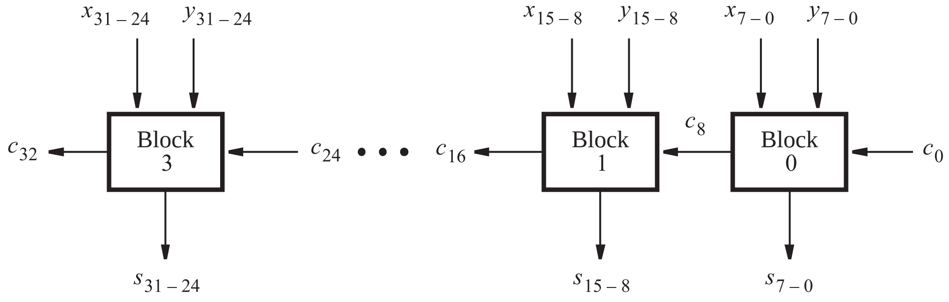


**Figure 5.17** A hierarchical carry-lookahead adder with ripple-carry between blocks.

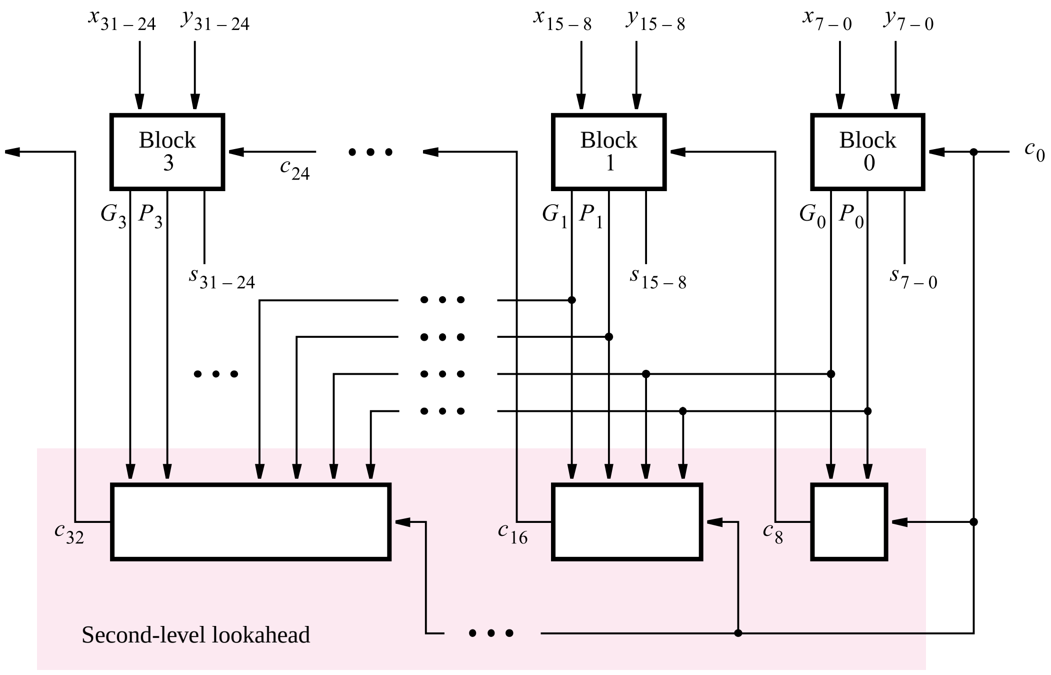


**Figure 5.18** A hierarchical carry-lookahead adder.

The last term in this expression specifies that, if all eight propagate functions are 1, then the carry-in $c_0$ is propagated through the entire block. Hence

$$P_0 = p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0$$

The rest of the terms in the expression for $c_8$ represent all other cases when the block produces a carry-out. Thus

$$G_0 = g_7 + p_7 g_6 + p_7 p_6 g_5 + \cdots + p_7 p_6 p_5 p_4 p_3 p_2 p_1 g_0$$

The expression for $c_8$ in the hierarchical adder is given by

$$c_8 = G_0 + P_0 c_0$$

For block 1 the expressions for $G_1$ and $P_1$ have the same form as for $G_0$ and $P_0$ except that each subscript $i$ is replaced by $i + 8$. The expressions for $G_2$, $P_2$, $G_3$, and $P_3$ are derived in the same way. The expression for the carry-out of block 1, $c_{16}$, is

$$\begin{aligned} c_{16} &= G_1 + P_1 c_8 \\ &= G_1 + P_1 G_0 + P_1 P_0 c_0 \end{aligned}$$

Similarly, the expressions for $c_{24}$ and $c_{32}$ are

$$\begin{aligned} c_{24} &= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0 \\ c_{32} &= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0 \end{aligned}$$

Using this scheme, it takes two more gate delays to produce the carry signals $c_8$, $c_{16}$, and $c_{24}$ than the time needed to generate the $G_j$ and $P_j$ functions. Therefore, since $G_j$ and $P_j$ require three gate delays, $c_8$, $c_{16}$, and $c_{24}$ are available after five gate delays. The time needed to add two 32-bit numbers involves these five gate delays plus two more to produce the internal carries in blocks 1, 2, and 3, plus one more gate delay (XOR) to generate each sum bit. This gives a total of eight gate delays.

In section 5.3.5 we determined that it takes $2n + 1$ gate delays to add two numbers using a ripple-carry adder. For 32-bit numbers this implies 65 gate delays. It is clear that the carry-lookahead adder offers a large performance improvement. The trade-off is much greater complexity of the required circuit.

### Technology Considerations

The preceding delay analysis assumes that gates with any number of inputs can be used. We know from Chapters 3 and 4 that the technology used to implement the gates limits the fan-in to a rather small number of inputs. Therefore the reality of fan-in constraints must be taken into account. To illustrate this problem, consider the expressions for the first eight carries:

$$\begin{aligned}
c_1 &= g_0 + p_0c_0 \\
c_2 &= g_1 + p_1g_0 + p_1p_0c_0 \\
&\ \ \vdots \\
c_8 &= g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4 + p_7p_6p_5p_4g_3 + p_7p_6p_5p_4p_3g_2 \\
&\quad + p_7p_6p_5p_4p_3p_2g_1 + p_7p_6p_5p_4p_3p_2p_1g_0 + p_7p_6p_5p_4p_3p_2p_1p_0c_0
\end{aligned}$$

Suppose that the maximum fan-in of the gates is four inputs. Then it is impossible to implement all of these expressions with a two-level AND-OR circuit. The biggest problem is $c_8$, where one of the AND gates requires nine inputs; moreover, the OR gate also requires nine inputs. To meet the fan-in constraint, we can rewrite the expression for $c_8$ as

$$\begin{aligned}
c_8 = {} & (g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4) + [(p_7p_6p_5p_4)(g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0)] \\
& + (p_7p_6p_5p_4)(p_3p_2p_1p_0)c_0
\end{aligned}$$

To implement this expression we need ten AND gates and three OR gates. The propagation delay in generating $c_8$ consists of one gate delay to develop all $g_i$ and $p_i$, two gate delays to produce the sum-of-products terms in parentheses, one gate delay to form the product term in square brackets, and one delay for the final ORing of terms. Hence $c_8$ is valid after five gate delays, rather than the three gates delays that would be needed without the fan-in constraint.

Because fan-in limitations reduce the speed of the carry-lookahead adder, some devices that are characterized by low fan-in include dedicated circuitry for implementation of fast adders. Examples of such devices include FPGAs whose logic blocks are based on lookup tables.

Before we leave the topic of the carry-lookahead adder, we should consider an alternative implementation of the structure in Figure 5.16. The same functionality can be achieved by using the circuit in Figure 5.19. In this case stage 0 is implemented using the circuit of Figure 5.5 in which 2 two-input XOR gates are used to generate the sum bit, rather than

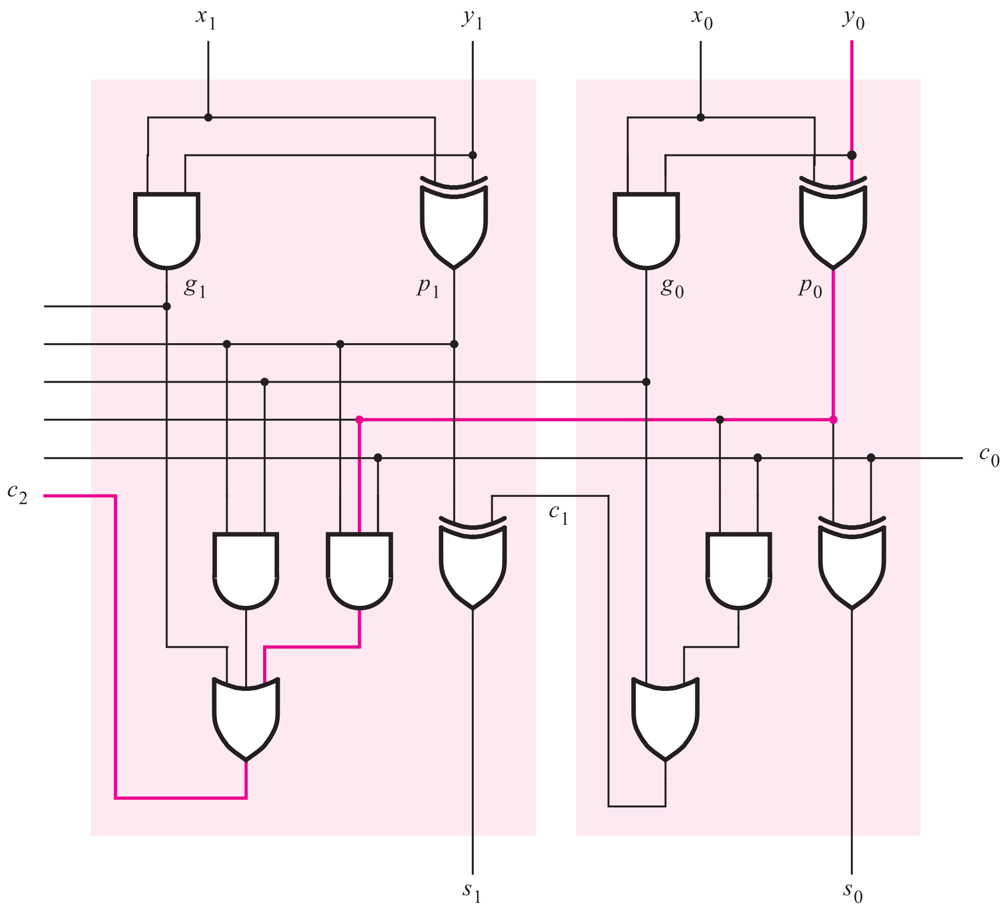


**Figure 5.19** An alternative design for a carry-lookahead adder.

having 1 three-input XOR gate. The output of the first XOR gate can also serve as the propagate signal $p_0$. Thus the corresponding OR gate in Figure 5.16 is not needed. Stage 1 is constructed using the same approach.

The circuits in Figures 5.16 and 5.19 require the same number of gates. But is one of them better in some way? The answer must be sought by considering the specific aspects of the technology that is used to implement the circuits. If a CPLD or an FPGA is used, such as those in Figures 3.33 and 3.39, then it does not matter which circuit is chosen. A three-input XOR function can be realized by one macrocell in the CPLD, using the sum-of-products expression

$$s_i = x_i\overline{y}_i\overline{c}_i + \overline{x}_i y_i \overline{c}_i + \overline{x}_i\overline{y}_i c_i + x_i y_i c_i$$

because the macrocell allows for implementation of four product terms.

In the FPGA any three-input function can be implemented in a single logic cell; hence it is easy to realize a three-input XOR. However, suppose that we want to build a carry-lookahead adder on a custom chip. If the XOR gate is constructed using the approach discussed in section 3.9.1, then a three-input XOR would actually be implemented using 2 two-input XOR gates, as we have done for the sum bits in Figure 5.19. Therefore, if the first XOR gate realizes the function $x_i \oplus y_i$, which is also the propagate function $p_i$, then it is obvious that the alternative in Figure 5.19 is more attractive. The important point of this discussion is that optimization of logic circuits may depend on the target technology. The CAD tools take this fact into account.

The carry-lookahead adder is a well-known concept. There exist standard chips that implement a portion of the carry-lookahead circuitry. They are called *carry-lookahead generators*. CAD tools often include predesigned subcircuits for adders, which designers can use to design larger units.

## 5.5 Design of Arithmetic Circuits Using CAD Tools

In this section we show how the arithmetic circuits can be designed by using CAD tools. Two different design methods are discussed: using schematic capture and using VHDL code.

### 5.5.1 Design of Arithmetic Circuits Using Schematic Capture

An obvious way to design an arithmetic circuit via schematic capture is to draw a schematic that contains the necessary logic gates. For example, to create an $n$-bit adder, we could first draw a schematic that represents a full-adder. Then an $n$-bit ripple-carry adder could be created by drawing a higher-level schematic that connects together *n instances* of the full-adder. A hierarchical schematic created in this manner would look like the circuit shown in Figure 5.6. We could also use this methodology to create an adder/subtractor circuit, such as the circuit depicted in Figure 5.13.

The main problem with this approach is that it is cumbersome, especially when the number of bits is large. This problem is even more apparent if we consider creating a schematic for a carry-lookahead adder. As shown in section 5.4.1, the carry circuitry in each stage of the carry-lookahead adder becomes increasingly more complex. Hence it is necessary to draw a separate schematic for each stage of the adder. A better approach for creating arithmetic circuits via schematic capture is to use predefined subcircuits.

We mentioned in section 2.9.1 that schematic capture tools provide a library of graphical symbols that represent basic logic gates. These gates are used to create schematics of relatively simple circuits. In addition to basic gates, most schematic capture tools also provide a library of commonly used circuits, such as adders. Each circuit is provided as a module that can be imported into a schematic and used as part of a larger circuit. In some CAD systems the modules are referred to as *macrofunctions*, or *megafunctions*.

chapter

# 6

# Combinational-Circuit Building Blocks

## Chapter Objectives

In this chapter you will learn about:

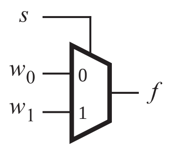
- Commonly used combinational subcircuits
- Multiplexers, which can be used for selection of signals and for implementation of general logic functions
- Circuits used for encoding, decoding, and code-conversion purposes
- Key VHDL constructs used to define combinational circuits

Previous chapters have introduced the basic techniques for design of logic circuits. In practice, a few types of logic circuits are often used as building blocks in larger designs. This chapter discusses a number of these blocks and gives examples of their use. The chapter also includes a major section on VHDL, which describes several key features of the language.

## 6.1 MULTIPLEXERS

Multiplexers were introduced briefly in Chapters 2 and 3. A multiplexer circuit has a number of data inputs, one or more select inputs, and one output. It passes the signal value on one of the data inputs to the output. The data input is selected by the values of the select inputs. Figure 6.1 shows a 2-to-1 multiplexer. Part (*a*) gives the symbol commonly used. The *select* input, $s$, chooses as the output of the multiplexer either input $w_0$ or $w_1$. The multiplexer's functionality can be described in the form of a truth table as shown in part (*b*) of the figure. Part (*c*) gives a sum-of-products implementation of the 2-to-1 multiplexer, and part (*d*) illustrates how it can be constructed with transmission gates.

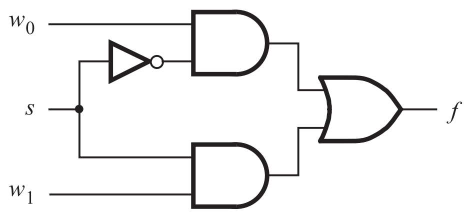
Figure 6.2*a* depicts a larger multiplexer with four data inputs, $w_0, \ldots, w_3$, and two select inputs, $s_1$ and $s_0$. As shown in the truth table in part (*b*) of the figure, the two-bit number represented by $s_1 s_0$ selects one of the data inputs as the output of the multiplexer.



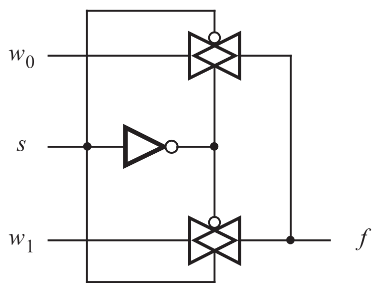
(a) Graphical symbol

| $s$ | $f$ |
|---|---|
| 0 | $w_0$ |
| 1 | $w_1$ |

(b) Truth table

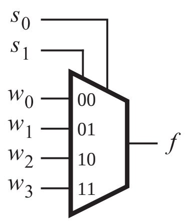


(c) Sum-of-products circuit



(d) Circuit with transmission gates

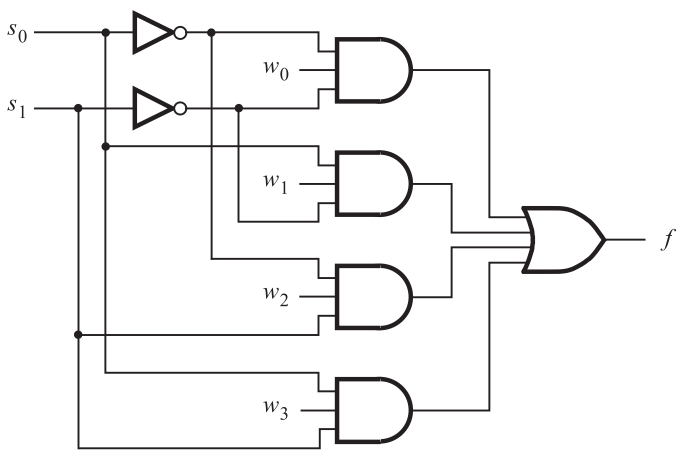
**Figure 6.1** A 2-to-1 multiplexer.



(a) Graphical symbol

| $s_1$ | $s_0$ | $f$ |
|---|---|---|
| 0 | 0 | $w_0$ |
| 0 | 1 | $w_1$ |
| 1 | 0 | $w_2$ |
| 1 | 1 | $w_3$ |

(b) Truth table



(c) Circuit

**Figure 6.2** A 4-to-1 multiplexer.

A sum-of-products implementation of the 4-to-1 multiplexer appears in Figure 6.2*c*. It realizes the multiplexer function

$$f = \overline{s}_1\overline{s}_0w_0 + \overline{s}_1s_0w_1 + s_1\overline{s}_0w_2 + s_1s_0w_3$$

It is possible to build larger multiplexers using the same approach. Usually, the number of data inputs, $n$, is an integer power of two. A multiplexer that has $n$ data inputs, $w_0, \ldots, w_{n-1}$, requires $\lceil \log_2 n \rceil$ select inputs. Larger multiplexers can also be constructed from smaller multiplexers. For example, the 4-to-1 multiplexer can be built using three 2-to-1 multiplexers as illustrated in Figure 6.3. If the 4-to-1 multiplexer is implemented using transmission gates, then the structure in this figure is always used. Figure 6.4 shows how a 16-to-1 multiplexer is constructed with five 4-to-1 multiplexers.

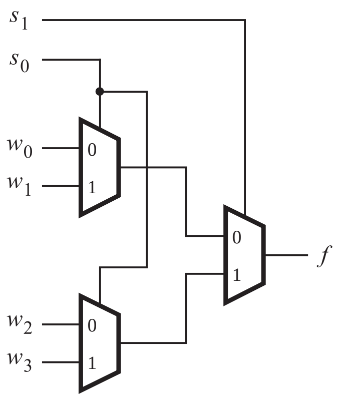


**Figure 6.3** Using 2-to-1 multiplexers to build a 4-to-1 multiplexer.

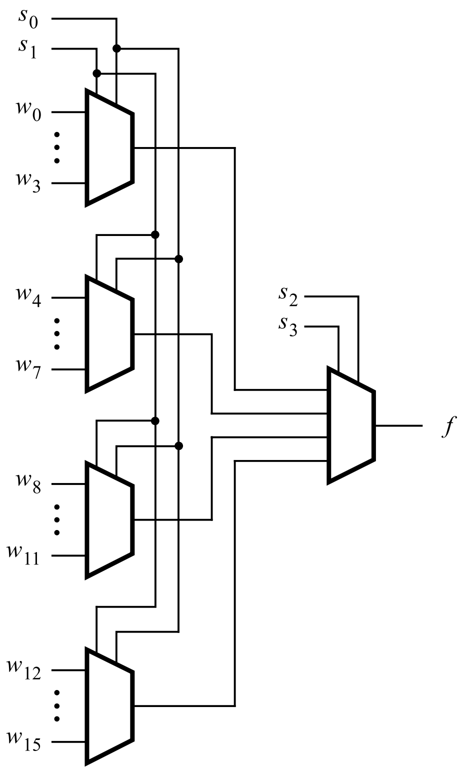


**Figure 6.4** A 16-to-1 multiplexer.

**Example 6.1**

Figure 6.5 shows a circuit that has two inputs, $x_1$ and $x_2$, and two outputs, $y_1$ and $y_2$. As indicated by the blue lines, the function of the circuit is to allow either of its inputs to be connected to either of its outputs, under the control of another input, $s$. A circuit that has $n$ inputs and $k$ outputs, whose sole function is to provide a capability to connect any input to any output, is usually referred to as an $n \times k$ crossbar switch. Crossbars of various sizes can be created, with different numbers of inputs and outputs. When there are two inputs and two outputs, it is called a $2 \times 2$ crossbar.

Figure 6.5*b* shows how the $2 \times 2$ crossbar can be implemented using 2-to-1 multiplexers. The multiplexer select inputs are controlled by the signal $s$. If $s = 0$, the crossbar connects $x_1$ to $y_1$ and $x_2$ to $y_2$, while if $s = 1$, the crossbar connects $x_1$ to $y_2$ and $x_2$ to $y_1$. Crossbar switches are useful in many practical applications in which it is necessary to be able to connect one set of wires to another set of wires, where the connection pattern changes from time to time.

**Example 6.2**

We introduced field-programmable gate array (FPGA) chips in section 3.6.5. Figure 3.39 depicts a small FPGA that is programmed to implement a particular circuit. The logic blocks in the FPGA have two inputs, and there are four tracks in each routing channel. Each of the programmable switches that connects a logic block input or output to an interconnection wire is shown as an X. A small part of Figure 3.39 is reproduced in Figure 6.6*a*. For clarity,

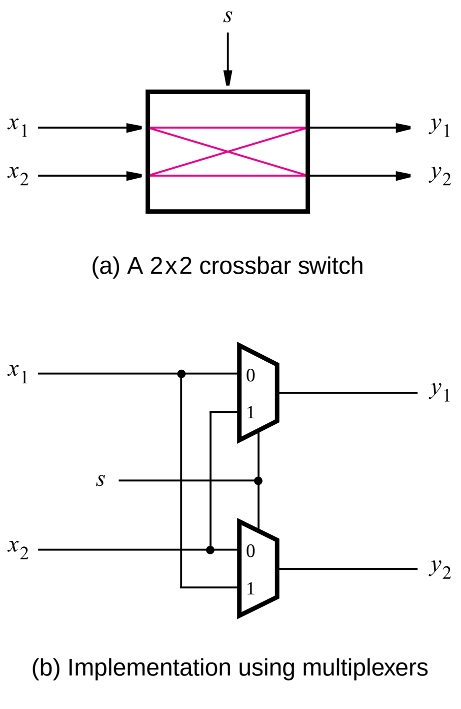
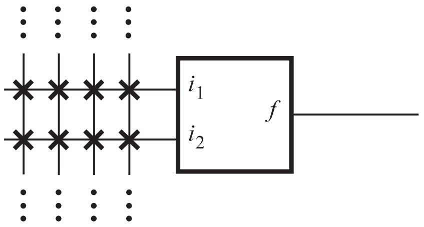
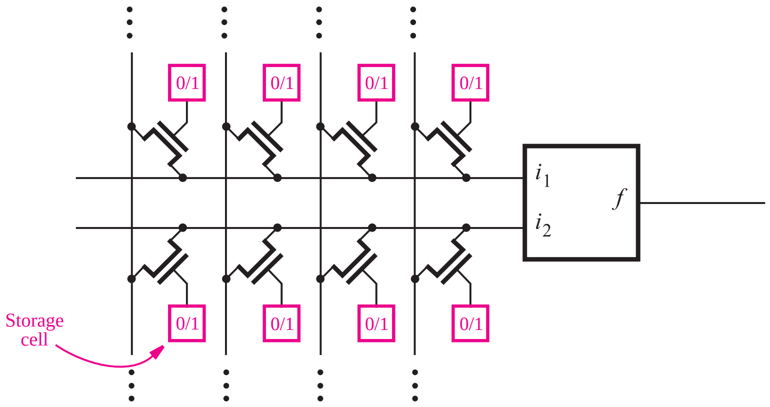

s
$x_1$
$x_2$
$y_1$
$y_2$
(a) A 2x2 crossbar switch
$x_1$
0
1
$y_1$
s
$x_2$
0
1
$y_2$
(b) Implementation using multiplexers

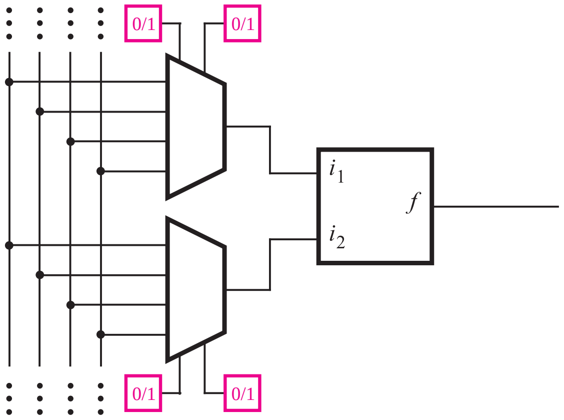
**Figure 6.5** A practical application of multiplexers.



(a) Part of the FPGA in Figure 3.39



(b) Implementation using pass transistors



(c) Implementation using multiplexers

**Figure 6.6** Implementing programmable switches in an FPGA.

the figure shows only a single logic block and the interconnection wires and switches associated with its input terminals.

One way in which the programmable switches can be implemented is illustrated in Figure 6.6*b*. Each X in part (*a*) of the figure is realized using an NMOS transistor controlled by a storage cell. This type of programmable switch was also shown in Figure 3.68. We described storage cells briefly in section 3.6.5 and will discuss them in more detail in section 10.1. Each cell stores a single logic value, either 0 or 1, and provides this value as the output of the cell. Each storage cell is built by using several transistors. Thus the eight cells shown in the figure use a significant amount of chip area.

The number of storage cells needed can be reduced by using multiplexers, as shown in Figure 6.6*c*. Each logic block input is fed by a 4-to-1 multiplexer, with the select inputs controlled by storage cells. This approach requires only four storage cells, instead of eight. In commercial FPGAs the multiplexer-based approach is usually adopted.

---

### 6.1.1 Synthesis of Logic Functions Using Multiplexers

Multiplexers are useful in many practical applications, such as those described above. They can also be used in a more general way to synthesize logic functions. Consider the example in Figure 6.7*a*. The truth table defines the function $f = w_1 \oplus w_2$. This function can be implemented by a 4-to-1 multiplexer in which the values of $f$ in each row of the truth table are connected as constants to the multiplexer data inputs. The multiplexer select inputs are driven by $w_1$ and $w_2$. Thus for each valuation of $w_1 w_2$, the output $f$ is equal to the function value in the corresponding row of the truth table.

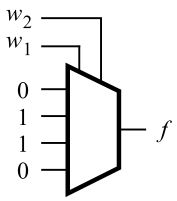
The above implementation is straightforward, but it is not very efficient. A better implementation can be derived by manipulating the truth table as indicated in Figure 6.7*b*, which allows $f$ to be implemented by a single 2-to-1 multiplexer. One of the input signals, $w_1$ in this example, is chosen as the select input of the 2-to-1 multiplexer. The truth table is redrawn to indicate the value of $f$ for each value of $w_1$. When $w_1 = 0$, $f$ has the same value as input $w_2$, and when $w_1 = 1$, $f$ has the value of $\overline{w}_2$. The circuit that implements this truth table is given in Figure 6.7*c*. This procedure can be applied to synthesize a circuit that implements any logic function.

---

**Example 6.3**

Figure 6.8*a* gives the truth table for the three-input majority function, and it shows how the truth table can be modified to implement the function using a 4-to-1 multiplexer. Any two of the three inputs may be chosen as the multiplexer select inputs. We have chosen $w_1$ and $w_2$ for this purpose, resulting in the circuit in Figure 6.8*b*.

| $w_1$ | $w_2$ | $f$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

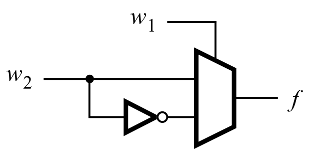


(a) Implementation using a 4-to-1 multiplexer

| $w_1$ | $w_2$ | $f$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

| $w_1$ | $f$ |
|---|---|
| 0 | $w_2$ |
| 1 | $\overline{w}_2$ |

(b) Modified truth table

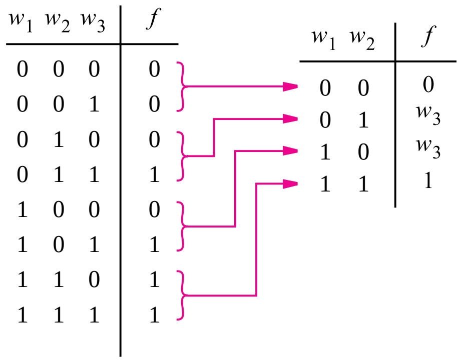


(c) Circuit

**Figure 6.7** Synthesis of a logic function using mutiplexers.

**Example 6.4** Figure 6.9*a* indicates how the function $f = w_1 \oplus w_2 \oplus w_3$ can be implemented using 2-to-1 multiplexers. When $w_1 = 0, f$ is equal to the XOR of $w_2$ and $w_3$, and when $w_1 = 1, f$ is the XNOR of $w_2$ and $w_3$. The left multiplexer in the circuit produces $w_2 \oplus w_3$, using the result from Figure 6.7, and the right multiplexer uses the value of $w_1$ to select either $w_2 \oplus w_3$ or its complement. Note that we could have derived this circuit directly by writing the function as $f = (w_2 \oplus w_3) \oplus w_1$.

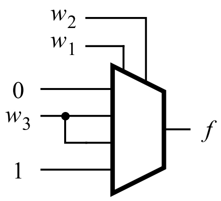
Figure 6.10 gives an implementation of the three-input XOR function using a 4-to-1 multiplexer. Choosing $w_1$ and $w_2$ for the select inputs results in the circuit shown.



| $w_1$ | $w_2$ | $w_3$ | $f$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

| $w_1$ | $w_2$ | $f$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | $w_3$ |
| 1 | 0 | $w_3$ |
| 1 | 1 | 1 |

(a) Modified truth table



(b) Circuit

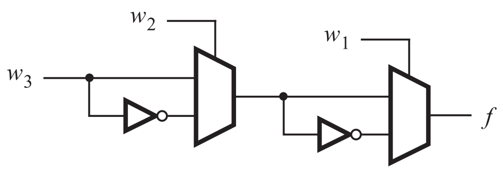
**Figure 6.8** Implementation of the three-input majority function using a 4-to-1 multiplexer.

| $w_1$ | $w_2$ | $w_3$ | $f$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

$w_2 \oplus w_3$

$\overline{w_2 \oplus w_3}$

(a) Truth table

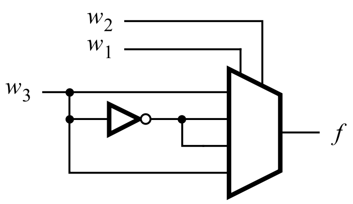


(b) Circuit

**Figure 6.9** Three-input XOR implemented with 2-to-1 multiplexers.

| $w_1$ | $w_2$ | $w_3$ | $f$ | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | $w_3$ |
| 0 | 0 | 1 | 1 | |
| 0 | 1 | 0 | 1 | $\overline{w}_3$ |
| 0 | 1 | 1 | 0 | |
| 1 | 0 | 0 | 1 | $\overline{w}_3$ |
| 1 | 0 | 1 | 0 | |
| 1 | 1 | 0 | 0 | $w_3$ |
| 1 | 1 | 1 | 1 | |

(a) Truth table



(b) Circuit

**Figure 6.10** Three-input XOR implemented with a 4-to-1 multiplexer.

## 6.1.2 Multiplexer Synthesis Using Shannon's Expansion

Figures 6.8 through 6.10 illustrate how truth tables can be interpreted to implement logic functions using multiplexers. In each case the inputs to the multiplexers are the constants 0 and 1, or some variable or its complement. Besides using such simple inputs, it is possible to connect more complex circuits as inputs to a multiplexer, allowing functions to be synthesized using a combination of multiplexers and other logic gates. Suppose that we want to implement the three-input majority function in Figure 6.8 using a 2-to-1 multiplexer in this way. Figure 6.11 shows an intuitive way of realizing this function. The truth table can be modified as shown on the right. If $w_1 = 0$, then $f = w_2w_3$, and if $w_1 = 1$, then $f = w_2 + w_3$. Using $w_1$ as the select input for a 2-to-1 multiplexer leads to the circuit in Figure 6.11*b*.

This implementation can be derived using algebraic manipulation as follows. The function in Figure 6.11*a* is expressed in sum-of-products form as

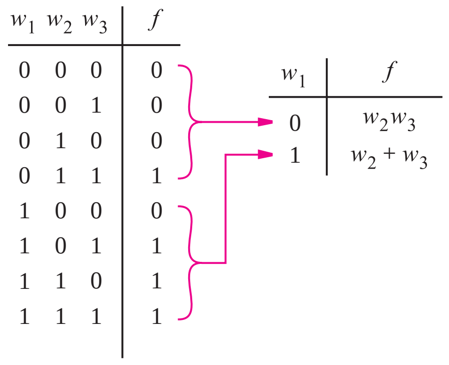
$$f = \overline{w}_1w_2w_3 + w_1\overline{w}_2w_3 + w_1w_2\overline{w}_3 + w_1w_2w_3$$

It can be manipulated into

$$\begin{aligned} f &= \overline{w}_1(w_2w_3) + w_1(\overline{w}_2w_3 + w_2\overline{w}_3 + w_2w_3) \\ &= \overline{w}_1(w_2w_3) + w_1(w_2 + w_3) \end{aligned}$$

which corresponds to the circuit in Figure 6.11*b*.

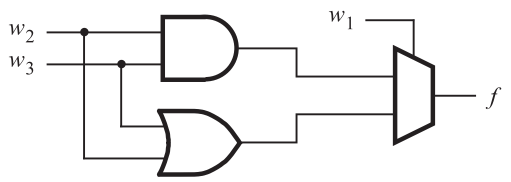
Multiplexer implementations of logic functions require that a given function be decomposed in terms of the variables that are used as the select inputs. This can be accomplished by means of a theorem proposed by Claude Shannon [1].



| $w_1$ | $w_2$ | $w_3$ | $f$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

| $w_1$ | $f$ |
|---|---|
| 0 | $w_2 w_3$ |
| 1 | $w_2 + w_3$ |

(a) Truth table



(b) Circuit

**Figure 6.11** The three-input majority function implemented using a 2-to-1 multiplexer.

## Shannon's Expansion Theorem

Any Boolean function $f(w_1, \ldots, w_n)$ can be written in the form

$$f(w_1, w_2, \ldots, w_n) = \overline{w}_1 \cdot f(0, w_2, \ldots, w_n) + w_1 \cdot f(1, w_2, \ldots, w_n)$$

This expansion can be done in terms of any of the $n$ variables. We will leave the proof of the theorem as an exercise for the reader (see problem 6.9).

To illustrate its use, we can apply the theorem to the three-input majority function, which can be written as

$$f(w_1, w_2, w_3) = w_1 w_2 + w_1 w_3 + w_2 w_3$$

Expanding this function in terms of $w_1$ gives

$$f = \overline{w}_1(w_2 w_3) + w_1(w_2 + w_3)$$

which is the expression that we derived above.

For the three-input XOR function, we have

$$\begin{aligned} f &= w_1 \oplus w_2 \oplus w_3 \\ &= \overline{w}_1 \cdot (w_2 \oplus w_3) + w_1 \cdot (\overline{w_2 \oplus w_3}) \end{aligned}$$

which gives the circuit in Figure 6.9*b*.

In Shannon's expansion the term $f(0, w_2, \ldots, w_n)$ is called the *cofactor* of $f$ with respect to $\overline{w}_1$; it is denoted in shorthand notation as $f_{\overline{w}_1}$. Similarly, the term $f(1, w_2, \ldots, w_n)$ is called the cofactor of $f$ with respect to $w_1$, written $f_{w_1}$. Hence we can write

$$f = \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1}$$

In general, if the expansion is done with respect to variable $w_i$, then $f_{w_i}$ denotes $f(w_1, \ldots, w_{i-1}, 1, w_{i+1}, \ldots, w_n)$ and

$$f(w_1, \ldots, w_n) = \overline{w}_i f_{\overline{w}_i} + w_i f_{w_i}$$

The complexity of the logic expression may vary, depending on which variable, $w_i$, is used, as illustrated in Example 6.5.

**Example 6.5** For the function $f = \overline{w}_1 w_3 + w_2 \overline{w}_3$, decomposition using $w_1$ gives

$$\begin{aligned} f &= \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1} \\ &= \overline{w}_1 (w_3 + w_2) + w_1 (w_2 \overline{w}_3) \end{aligned}$$

Using $w_2$ instead of $w_1$ produces

$$\begin{aligned} f &= \overline{w}_2 f_{\overline{w}_2} + w_2 f_{w_2} \\ &= \overline{w}_2 (\overline{w}_1 w_3) + w_2 (\overline{w}_1 + \overline{w}_3) \end{aligned}$$

Finally, using $w_3$ gives

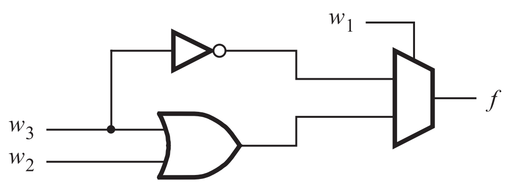
$$\begin{aligned} f &= \overline{w}_3 f_{\overline{w}_3} + w_3 f_{w_3} \\ &= \overline{w}_3 (w_2) + w_3 (\overline{w}_1) \end{aligned}$$

The results generated using $w_1$ and $w_2$ have the same cost, but the expression produced using $w_3$ has a lower cost. In practice, the CAD tools that perform decompositions of this type try a number of alternatives and choose the one that produces the best result.

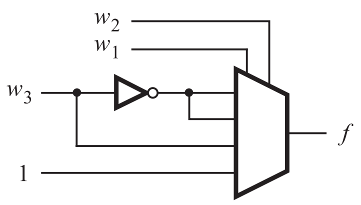
Shannon's expansion can be done in terms of more than one variable. For example, expanding a function in terms of $w_1$ and $w_2$ gives

$$\begin{aligned} f(w_1, \ldots, w_n) = {} & \overline{w}_1 \overline{w}_2 \cdot f(0, 0, w_3, \ldots, w_n) + \overline{w}_1 w_2 \cdot f(0, 1, w_3, \ldots, w_n) \\ & + w_1 \overline{w}_2 \cdot f(1, 0, w_3, \ldots, w_n) + w_1 w_2 \cdot f(1, 1, w_3, \ldots, w_n) \end{aligned}$$

This expansion gives a form that can be implemented using a 4-to-1 multiplexer. If Shannon's expansion is done in terms of all $n$ variables, then the result is the canonical sum-of-products form, which was defined in section 2.6.1.



(a) Using a 2-to-1 multiplexer



(b) Using a 4-to-1 multiplexer

**Figure 6.12** The circuits synthesized in Example 6.6.

---

**Example 6.6**

Assume that we wish to implement the function

$$f = \overline{w}_1\overline{w}_3 + w_1w_2 + w_1w_3$$

using a 2-to-1 multiplexer and any other necessary gates. Shannon's expansion using $w_1$ gives

$$\begin{aligned} f &= \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1} \\ &= \overline{w}_1(\overline{w}_3) + w_1(w_2 + w_3) \end{aligned}$$

The corresponding circuit is shown in Figure 6.12*a*. Assume now that we wish to use a 4-to-1 multiplexer instead. Further decomposition using $w_2$ gives

$$\begin{aligned} f &= \overline{w}_1\overline{w}_2 f_{\overline{w}_1\overline{w}_2} + \overline{w}_1 w_2 f_{\overline{w}_1 w_2} + w_1\overline{w}_2 f_{w_1\overline{w}_2} + w_1 w_2 f_{w_1 w_2} \\ &= \overline{w}_1\overline{w}_2(\overline{w}_3) + \overline{w}_1 w_2(\overline{w}_3) + w_1\overline{w}_2(w_3) + w_1 w_2(1) \end{aligned}$$

The circuit is shown in Figure 6.12*b*.

---

**Example 6.7**

Consider the three-input majority function

$$f = w_1w_2 + w_1w_3 + w_2w_3$$

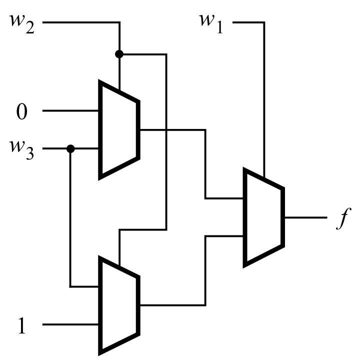


**Figure 6.13** The circuit synthesized in Example 6.7.

We wish to implement this function using only 2-to-1 multiplexers. Shannon's expansion using $w_1$ yields

$$\begin{aligned} f &= \overline{w}_1(w_2w_3) + w_1(w_2 + w_3 + w_2w_3) \\ &= \overline{w}_1(w_2w_3) + w_1(w_2 + w_3) \end{aligned}$$

Let $g = w_2w_3$ and $h = w_2 + w_3$. Expansion of both $g$ and $h$ using $w_2$ gives

$$\begin{aligned} g &= \overline{w}_2(0) + w_2(w_3) \\ h &= \overline{w}_2(w_3) + w_2(1) \end{aligned}$$

The corresponding circuit is shown in Figure 6.13. It is equivalent to the 4-to-1 multiplexer circuit derived using a truth table in Figure 6.8.

---

**Example 6.8** In section 3.6.5 we said that most FPGAs use lookup tables for their logic blocks. Assume that an FPGA exists in which each logic block is a three-input lookup table (3-LUT). Because it stores a truth table, a 3-LUT can realize any logic function of three variables. Using Shannon's expansion, any four-variable function can be realized with at most three 3-LUTs. Consider the function

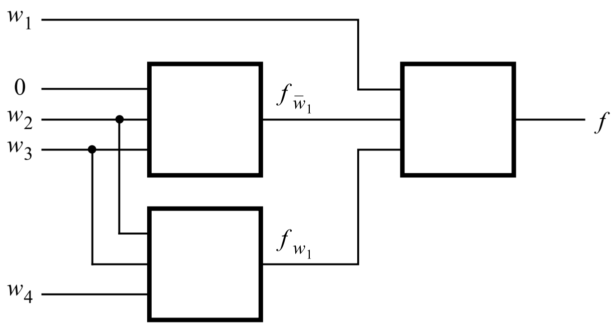
$$f = \overline{w}_2w_3 + \overline{w}_1w_2\overline{w}_3 + w_2\overline{w}_3w_4 + w_1\overline{w}_2\overline{w}_4$$

Expansion in terms of $w_1$ produces

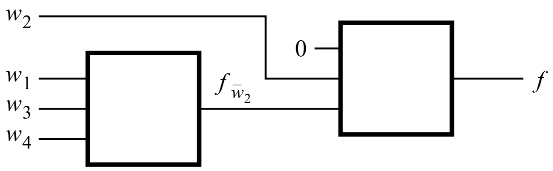
$$\begin{aligned} f &= \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1} \\ &= \overline{w}_1(\overline{w}_2w_3 + w_2\overline{w}_3 + w_2\overline{w}_3w_4) + w_1(\overline{w}_2w_3 + w_2\overline{w}_3w_4 + \overline{w}_2\overline{w}_4) \\ &= \overline{w}_1(\overline{w}_2w_3 + w_2\overline{w}_3) + w_1(\overline{w}_2w_3 + w_2\overline{w}_3w_4 + \overline{w}_2\overline{w}_4) \end{aligned}$$

A circuit with three 3-LUTs that implements this expression is shown in Figure 6.14*a*. Decomposition of the function using $w_2$, instead of $w_1$, gives

$$\begin{aligned} f &= \overline{w}_2 f_{\overline{w}_2} + w_2 f_{w_2} \\ &= \overline{w}_2(w_3 + w_1\overline{w}_4) + w_2(\overline{w}_1\overline{w}_3 + \overline{w}_3w_4) \end{aligned}$$



(a) Using three 3-LUTs



(b) Using two 3-LUTs

**Figure 6.14** Circuits synthesized in Example 6.8.

Observe that $\overline{f}_{\overline{w}_2} = f_{w_2}$; hence only two 3-LUTs are needed, as illustrated in Figure 6.14*b*. The LUT on the right implements the two-variable function $\overline{w}_2 f_{\overline{w}_2} + w_2 \overline{f}_{\overline{w}_2}$.

Since it is possible to implement any logic function using multiplexers, general-purpose chips exist that contain multiplexers as their basic logic resources. Both Actel Corporation [2] and QuickLogic Corporation [3] offer FPGAs in which the logic block comprises an arrangement of multiplexers. Texas Instruments offers gate array chips that have multiplexer-based logic blocks [4].

## 6.2 Decoders

Decoder circuits are used to decode encoded information. A binary decoder, depicted in Figure 6.15, is a logic circuit with $n$ inputs and $2^n$ outputs. Only one output is asserted at a time, and each output corresponds to one valuation of the inputs. The decoder also has an enable input, *En*, that is used to disable the outputs; if $En = 0$, then none of the decoder outputs is asserted. If $En = 1$, the valuation of $w_{n-1} \cdots w_1 w_0$ determines which of the outputs is asserted. An $n$-bit binary code in which exactly one of the bits is set to 1 at a

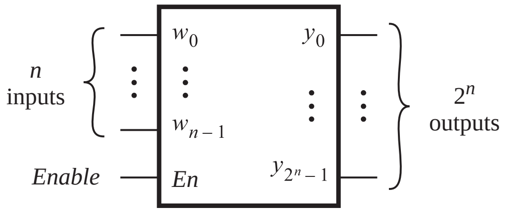


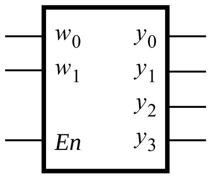
**Figure 6.15** An $n$-to-$2^n$ binary decoder.

time is referred to as *one-hot encoded*, meaning that the single bit that is set to 1 is deemed to be "hot." The outputs of a binary decoder are one-hot encoded.

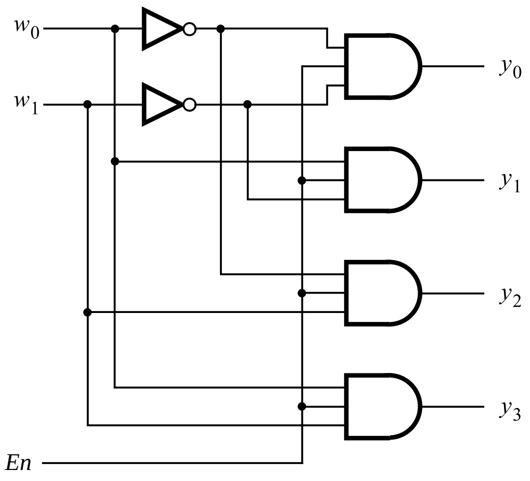
A 2-to-4 decoder is given in Figure 6.16. The two data inputs are $w_1$ and $w_0$. They represent a two-bit number that causes the decoder to assert one of the outputs $y_0, \dots, y_3$. Although a decoder can be designed to have either active-high or active-low outputs, in

| $En$ | $w_1$ | $w_0$ | $y_0$ | $y_1$ | $y_2$ | $y_3$ |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 0 | x | x | 0 | 0 | 0 | 0 |

(a) Truth table



(b) Graphical symbol



(c) Logic circuit

**Figure 6.16** A 2-to-4 decoder.

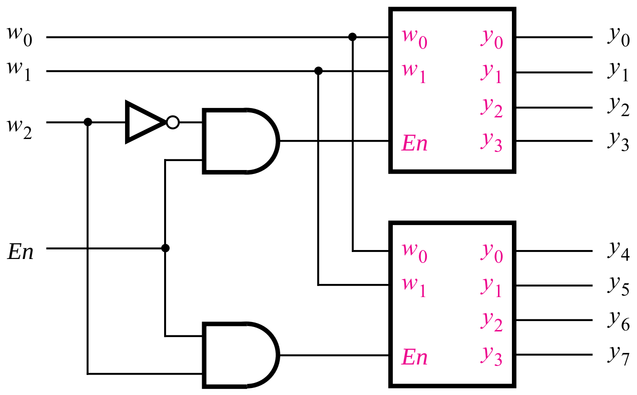


**Figure 6.17** A 3-to-8 decoder using two 2-to-4 decoders.

Figure 6.16 active-high outputs are assumed. Setting the inputs $w_1w_0$ to 00, 01, 10, or 11 causes the output $y_0$, $y_1$, $y_2$, or $y_3$ to be set to 1, respectively. A graphical symbol for the decoder is given in part (*b*) of the figure, and a logic circuit is shown in part (*c*).

Larger decoders can be built using the sum-of-products structure in Figure 6.16*c*, or else they can be constructed from smaller decoders. Figure 6.17 shows how a 3-to-8 decoder is built with two 2-to-4 decoders. The $w_2$ input drives the enable inputs of the two decoders. The top decoder is enabled if $w_2 = 0$, and the bottom decoder is enabled if $w_2 = 1$. This concept can be applied for decoders of any size. Figure 6.18 shows how five 2-to-4 decoders can be used to construct a 4-to-16 decoder. Because of its treelike structure, this type of circuit is often referred to as a *decoder tree*.

**Example 6.9**

Decoders are useful for many practical purposes. In Figure 6.2*c* we showed the sum-of-products implementation of the 4-to-1 multiplexer, which requires AND gates to distinguish the four different valuations of the select inputs $s_1$ and $s_0$. Since a decoder evaluates the values on its inputs, it can be used to build a multiplexer as illustrated in Figure 6.19. The enable input of the decoder is not needed in this case, and it is set to 1. The four outputs of the decoder represent the four valuations of the select inputs.

**Example 6.10**

In Figure 3.59 we showed how a 2-to-1 multiplexer can be constructed using two tri-state buffers. This concept can be applied to any size of multiplexer, with the addition of a decoder. An example is shown in Figure 6.20. The decoder enables one of the tri-state buffers for each valuation of the select lines, and that tri-state buffer drives the output, $f$, with the selected data input. We have now seen that multiplexers can be implemented in various ways. The choice of whether to employ the sum-of-products form, transmission gates, or tri-state buffers depends on the resources available in the chip being used. For instance, most FPGAs that use lookup tables for their logic blocks do not contain tri-state

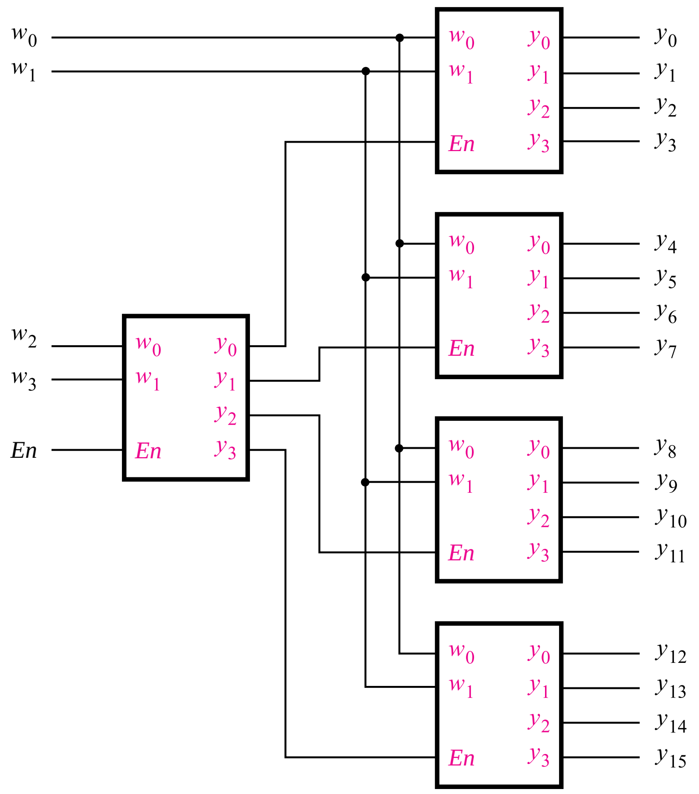


**Figure 6.18** A 4-to-16 decoder built using a decoder tree.

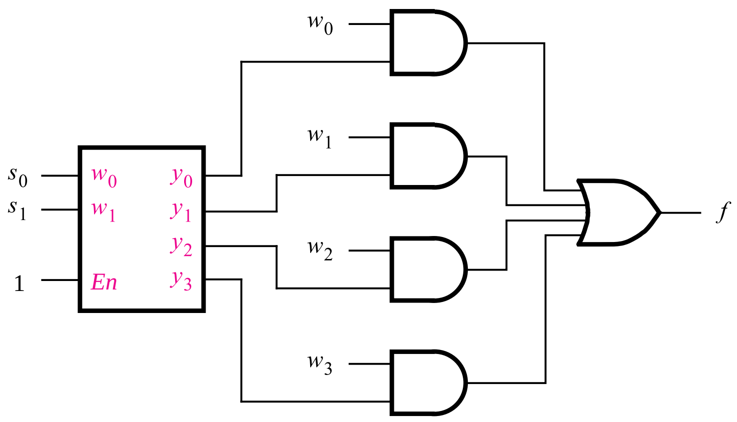


**Figure 6.19** A 4-to-1 multiplexer built using a decoder.

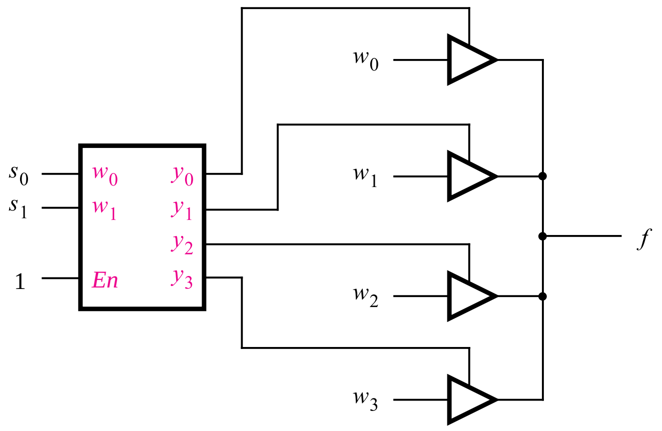


**Figure 6.20** A 4-to-1 multiplexer built using a decoder and tri-state buffers.

buffers. Hence multiplexers must be implemented in the sum-of-products form using the lookup tables (see Example 6.30).

---

## 6.2.1 Demultiplexers

We showed in section 6.1 that a multiplexer has one output, $n$ data inputs, and $\lceil \log_2 n \rceil$ select inputs. The purpose of the multiplexer circuit is to *multiplex* the $n$ data inputs onto the single data output under control of the select inputs. A circuit that performs the opposite function, namely, placing the value of a single data input onto multiple data outputs, is called a *demultiplexer*. The demultiplexer can be implemented using a decoder circuit. For example, the 2-to-4 decoder in Figure 6.16 can be used as a 1-to-4 demultiplexer. In this case the $En$ input serves as the data input for the demultiplexer, and the $y_0$ to $y_3$ outputs are the data outputs. The valuation of $w_1w_0$ determines which of the outputs is set to the value of $En$. To see how the circuit works, consider the truth table in Figure 6.16*a*. When $En = 0$, all the outputs are set to 0, including the one selected by the valuation of $w_1w_0$. When $En = 1$, the valuation of $w_1w_0$ sets the appropriate output to 1.

In general, an $n$-to-$2^n$ decoder circuit can be used as a 1-to-$n$ demultiplexer. However, in practice decoder circuits are used much more often as decoders rather than as demultiplexers. In many applications the decoder's $En$ input is not actually needed; hence it can be omitted. In this case the decoder always asserts one of its data outputs, $y_0, \ldots, y_{2^n-1}$, according to the valuation of the data inputs, $w_{n-1} \cdots w_0$. Example 6.11 uses a decoder that does not have the $En$ input.

**Example 6.11** One of the most important applications of decoders is in memory blocks, which are used to store information. Such memory blocks are included in digital systems, such as computers, where there is a need to store large amounts of information electronically. One type of memory block is called a *read-only memory* (ROM). A ROM consists of a collection of storage cells, where each cell permanently stores a single logic value, either 0 or 1. Figure 6.21 shows an example of a ROM block. The storage cells are arranged in $2^m$ rows with $n$ cells per row. Thus each row stores $n$ bits of information. The location of each row in the ROM is identified by its *address*. In the figure the row at the top of the ROM has address 0, and the row at the bottom has address $2^m - 1$. The information stored in the rows can be accessed by asserting the select lines, $\text{Sel}_0$ to $\text{Sel}_{2^m-1}$. As shown in the figure, a decoder with $m$ inputs and $2^m$ outputs is used to generate the signals on the select lines. Since the inputs to the decoder choose the particular address (row) selected, they are called the *address* lines. The information stored in the row appears on the data outputs of the ROM, $d_{n-1}, \ldots, d_0$, which are called the *data* lines. Figure 6.21 shows that each data line has an associated tri-state buffer that is enabled by the ROM input named *Read*. To access, or *read*, data from the ROM, the address of the desired row is placed on the address lines and *Read* is set to 1.

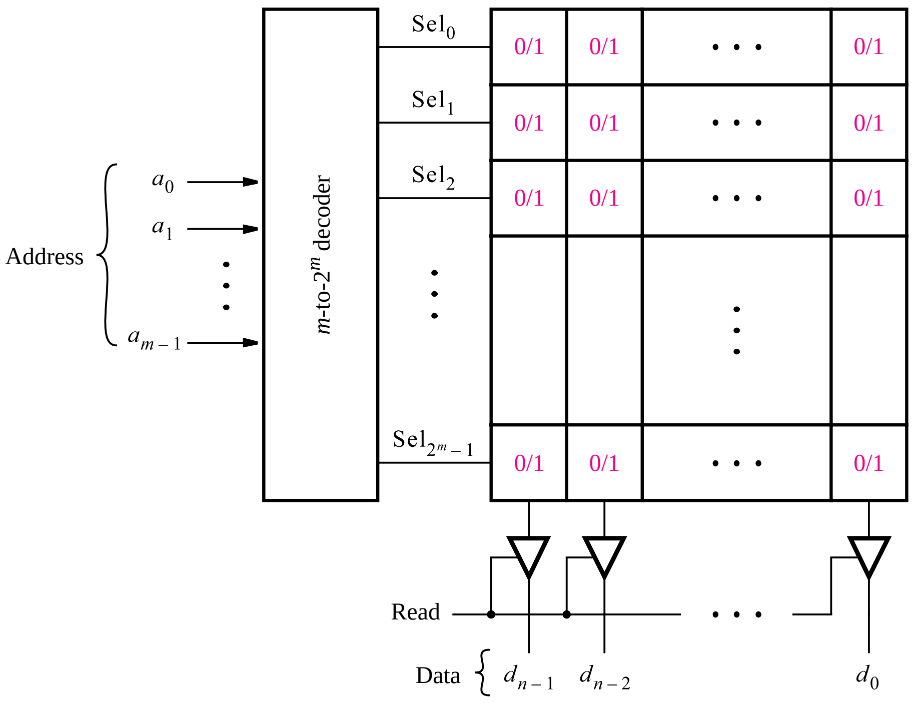


**Figure 6.21** A $2^m \times n$ read-only memory (ROM) block.

Many different types of memory blocks exist. In a ROM the stored information can be read out of the storage cells, but it cannot be changed (see problem 6.32). Another type of ROM allows information to be both read out of the storage cells and stored, or *written*, into them. Reading its contents is the normal operation, whereas writing requires a special procedure. Such a memory block is called a programmable ROM (PROM). The storage cells in a PROM are usually implemented using EEPROM transistors. We discussed EEPROM transistors in section 3.10 to show how they are used in PLDs. Other types of memory blocks are discussed in section 10.1.

## 6.3 Encoders

An encoder performs the opposite function of a decoder. It encodes given information into a more compact form.

### 6.3.1 Binary Encoders

A *binary encoder* encodes information from $2^n$ inputs into an $n$-bit code, as indicated in Figure 6.22. Exactly one of the input signals should have a value of 1, and the outputs present the binary number that identifies which input is equal to 1. The truth table for a 4-to-2 encoder is provided in Figure 6.23*a*. Observe that the output $y_0$ is 1 when either input $w_1$ or $w_3$ is 1, and output $y_1$ is 1 when input $w_2$ or $w_3$ is 1. Hence these outputs can be generated by the circuit in Figure 6.23*b*. Note that we assume that the inputs are one-hot encoded. All input patterns that have multiple inputs set to 1 are not shown in the truth table, and they are treated as don't-care conditions.

Encoders are used to reduce the number of bits needed to represent given information. A practical use of encoders is for transmitting information in a digital system. Encoding the information allows the transmission link to be built using fewer wires. Encoding is also useful if information is to be stored for later use because fewer bits need to be stored.

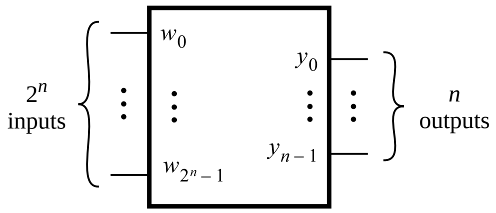
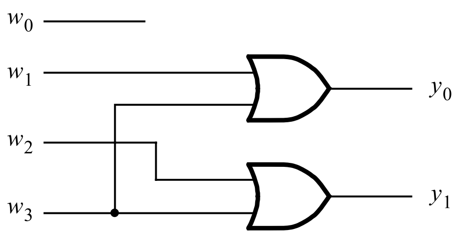


**Figure 6.22** A $2^n$-to-$n$ binary encoder.

| $w_3$ | $w_2$ | $w_1$ | $w_0$ | $y_1$ | $y_0$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 |

(a) Truth table



(b) Circuit

**Figure 6.23** A 4-to-2 binary encoder.

## 6.3.2 Priority Encoders

Another useful class of encoders is based on the priority of input signals. In a *priority encoder* each input has a priority level associated with it. The encoder outputs indicate the active input that has the highest priority. When an input with a high priority is asserted, the other inputs with lower priority are ignored. The truth table for a 4-to-2 priority encoder is shown in Figure 6.24. It assumes that $w_0$ has the lowest priority and $w_3$ the highest. The outputs $y_1$ and $y_0$ represent the binary number that identifies the highest priority input set to 1. Since it is possible that none of the inputs is equal to 1, an output, $z$, is provided to indicate this condition. It is set to 1 when at least one of the inputs is equal to 1. It is set to

| $w_3$ | $w_2$ | $w_1$ | $w_0$ | $y_1$ | $y_0$ | z |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | d | d | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | x | 0 | 1 | 1 |
| 0 | 1 | x | x | 1 | 0 | 1 |
| 1 | x | x | x | 1 | 1 | 1 |

**Figure 6.24** Truth table for a 4-to-2 priority encoder.

0 when all inputs are equal to 0. The outputs $y_1$ and $y_0$ are not meaningful in this case, and hence the first row of the truth table can be treated as a don't-care condition for $y_1$ and $y_0$.

The behavior of the priority encoder is most easily understood by first considering the last row in the truth table. It specifies that if input $w_3$ is 1, then the outputs are set to $y_1y_0 = 11$. Because $w_3$ has the highest priority level, the values of inputs $w_2$, $w_1$, and $w_0$ do not matter. To reflect the fact that their values are irrelevant, $w_2$, $w_1$, and $w_0$ are denoted by the symbol x in the truth table. The second-last row in the truth table stipulates that if $w_2 = 1$, then the outputs are set to $y_1y_0 = 10$, but only if $w_3 = 0$. Similarly, input $w_1$ causes the outputs to be set to $y_1y_0 = 01$ only if both $w_3$ and $w_2$ are 0. Input $w_0$ produces the outputs $y_1y_0 = 00$ only if $w_0$ is the only input that is asserted.

A logic circuit that implements the truth table can be synthesized by using the techniques developed in Chapter 4. However, a more convenient way to derive the circuit is to define a set of intermediate signals, $i_0, \ldots, i_3$, based on the observations above. Each signal, $i_k$, is equal to 1 only if the input with the same index, $w_k$, represents the highest-priority input that is set to 1. The logic expressions for $i_0, \ldots, i_3$ are

$$
\begin{aligned}
i_0 &= \overline{w}_3\overline{w}_2\overline{w}_1w_0\\
i_1 &= \overline{w}_3\overline{w}_2w_1\\
i_2 &= \overline{w}_3w_2\\
i_3 &= w_3
\end{aligned}
$$

Using the intermediate signals, the rest of the circuit for the priority encoder has the same structure as the binary encoder in Figure 6.23, namely

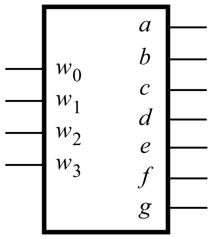
$$
\begin{aligned}
y_0 &= i_1 + i_3\\
y_1 &= i_2 + i_3
\end{aligned}
$$

The output $z$ is given by

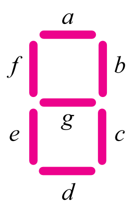
$$z = i_0 + i_1 + i_2 + i_3$$

## 6.4 CODE CONVERTERS

The purpose of the decoder and encoder circuits is to convert from one type of input encoding to a different output encoding. For example, a 3-to-8 binary decoder converts from a binary number on the input to a one-hot encoding at the output. An 8-to-3 binary encoder performs the opposite conversion. There are many other possible types of code converters. One common example is a BCD-to-7-segment decoder, which converts one binary-coded decimal (BCD) digit into information suitable for driving a digit-oriented display. As illustrated in Figure 6.25*a*, the circuit converts the BCD digit into seven signals that are used to drive the segments in the display. Each segment is a small light-emitting diode (LED), which glows when driven by an electrical signal. The segments are labeled from $a$ to $g$ in the figure. The truth table for the BCD-to-7-segment decoder is given in Figure 6.25*c*. For each valuation of the inputs $w_3, \ldots, w_0$, the seven outputs are set to



(a) Code converter



(b) 7-segment display

| $w_3$ | $w_2$ | $w_1$ | $w_0$ | $a$ | $b$ | $c$ | $d$ | $e$ | $f$ | $g$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |

(c) Truth table

**Figure 6.25** A BCD-to-7-segment display code converter.

display the appropriate BCD digit. Note that the last 6 rows of a complete 16-row truth table are not shown. They represent don't-care conditions because they are not legal BCD codes and will never occur in a circuit that deals with BCD data. A circuit that implements the truth table can be derived using the synthesis techniques discussed in Chapter 4. Finally, we should note that although the word *decoder* is traditionally used for this circuit, a more appropriate term is *code converter*. The term *decoder* is more appropriate for circuits that produce one-hot encoded outputs.

## 6.5 Arithmetic Comparison Circuits

Chapter 5 presented arithmetic circuits that perform addition, subtraction, and multiplication of binary numbers. Another useful type of arithmetic circuit compares the relative sizes of two binary numbers. Such a circuit is called a *comparator*. This section considers the design of a comparator that has two $n$-bit inputs, $A$ and $B$, which represent unsigned binary numbers. The comparator produces three outputs, called $AeqB$, $AgtB$, and $AltB$. The $AeqB$ output is set to 1 if $A$ and $B$ are equal. The $AgtB$ output is 1 if $A$ is greater than $B$, and the $AltB$ output is 1 if $A$ is less than $B$.

The desired comparator can be designed by creating a truth table that specifies the three outputs as functions of $A$ and $B$. However, even for moderate values of $n$, the truth table is large. A better approach is to derive the comparator circuit by considering the bits of $A$ and $B$ in pairs. We can illustrate this by a small example, where $n = 4$.

Let $A = a_3a_2a_1a_0$ and $B = b_3b_2b_1b_0$. Define a set of intermediate signals called $i_3$, $i_2$, $i_1$, and $i_0$. Each signal, $i_k$, is 1 if the bits of $A$ and $B$ with the same index are equal. That is, $i_k = \overline{a_k \oplus b_k}$. The comparator's $AeqB$ output is then given by

$$AeqB = i_3i_2i_1i_0$$

An expression for the $AgtB$ output can be derived by considering the bits of $A$ and $B$ in the order from the most-significant bit to the least-significant bit. The first bit-position, $k$, at which $a_k$ and $b_k$ differ determines whether $A$ is less than or greater than $B$. If $a_k = 0$ and $b_k = 1$, then $A < B$. But if $a_k = 1$ and $b_k = 0$, then $A > B$. The $AgtB$ output is defined by

$$AgtB = a_3\overline{b}_3 + i_3a_2\overline{b}_2 + i_3i_2a_1\overline{b}_1 + i_3i_2i_1a_0\overline{b}_0$$

The $i_k$ signals ensure that only the first digits, considered from the left to the right, of $A$ and $B$ that differ determine the value of $AgtB$.

The $AltB$ output can be derived by using the other two outputs as

$$AltB = \overline{AeqB + AgtB}$$

A logic circuit that implements the four-bit comparator circuit is shown in Figure 6.26. This approach can be used to design a comparator for any value of $n$.

Comparator circuits, like most logic circuits, can be designed in different ways. Another approach for designing a comparator circuit is presented in Example 5.10 in Chapter 5.

## 6.6 VHDL for Combinational Circuits

Having presented a number of useful circuits that can be used as building blocks in larger circuits, we will now consider how such circuits can be described in VHDL. Rather than relying on the simple VHDL statements used in previous examples, such as logic expressions, we will specify the circuits in terms of their behavior. We will also introduce a number of new VHDL constructs.

### 6.6.1 Assignment Statements

VHDL provides several types of statements that can be used to assign logic values to signals. In the examples of VHDL code given so far, only simple assignment statements have been used, either for logic or arithmetic expressions. This section introduces other types of