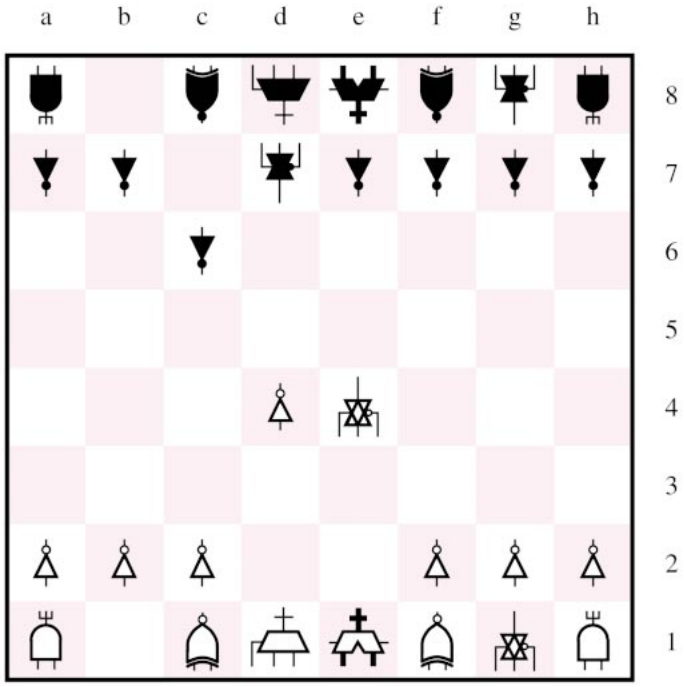
chapter

4

Optimized Implementation of Logic Functions



4. Nc3xe4, Nb8–d7

In Chapter 2 we showed that algebraic manipulation can be used to find the lowest-cost implementations of logic functions. The purpose of that chapter was to introduce the basic concepts in the synthesis process. The reader is probably convinced that it is easy to derive a straightforward realization of a logic function in a canonical form, but it is not at all obvious how to choose and apply the theorems and properties of section 2.5 to find a minimum-cost circuit. Indeed, the algebraic manipulation is rather tedious and quite impractical for functions of many variables.

If CAD tools are used to design logic circuits, the task of minimizing the cost of implementation does not fall to the designer; the tools perform the necessary optimizations automatically. Even so, it is essential to know something about this process. Most CAD tools have many features and options that are under control of the user. To know when and how to apply these options, the user must have an understanding of what the tools do.

In this chapter we will introduce some of the optimization techniques implemented in CAD tools and show how these techniques can be automated. As a first step we will discuss a graphical approach, known as the Karnaugh map, which provides a neat way to manually derive minimum-cost implementations of simple logic functions. Although it is not suitable for implementation in CAD tools, it illustrates a number of key concepts. We will show how both two-level and multilevel circuits can be designed. Then we will describe a cubical representation for logic functions, which is suitable for use in CAD tools. We will also continue our discussion of the VHDL language and CAD tools.

4.1 KARNAUGH MAP

In section 2.6 we saw that the key to finding a minimum-cost expression for a given logic function is to reduce the number of product (or sum) terms needed in the expression, by applying the combining property 14*a* (or 14*b*) as judiciously as possible. The Karnaugh map approach provides a systematic way of performing this optimization. To understand how it works, it is useful to review the algebraic approach from Chapter 2. Consider the function f in Figure 4.1. The canonical sum-of-products expression for f consists of minterms m_0, m_2, m_4, m_5, and m_6, so that

$$f = \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1x_2\bar{x}_3 + x_1\bar{x}_2\bar{x}_3 + x_1\bar{x}_2x_3 + x_1x_2\bar{x}_3$$

The combining property 14*a* allows us to replace two minterms that differ in the value of only one variable with a single product term that does not include that variable at all. For example, both m_0 and m_2 include $\bar{x}_1$ and $\bar{x}_3$, but they differ in the value of x_2 because m_0 includes $\bar{x}_2$ while m_2 includes x_2. Thus

$$\begin{aligned}\bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1x_2\bar{x}_3 &= \bar{x}_1(\bar{x}_2 + x_2)\bar{x}_3 \\ &= \bar{x}_1 \cdot 1 \cdot \bar{x}_3 \\ &= \bar{x}_1\bar{x}_3\end{aligned}$$

Row number	x_1	x_2	x_3	f
0	0	0	0	1
1	0	0	1	0
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

Figure 4.1 The function $f(x_1, x_2, x_3) = \sum m(0, 2, 4, 5, 6)$.

Hence m_0 and m_2 can be replaced by the single product term $\bar{x}_1\bar{x}_3$. Similarly, m_4 and m_6 differ only in the value of x_2 and can be combined using

$$\begin{aligned} x_1\bar{x}_2\bar{x}_3 + x_1x_2\bar{x}_3 &= x_1(\bar{x}_2 + x_2)\bar{x}_3 \\ &= x_1 \cdot 1 \cdot \bar{x}_3 \\ &= x_1\bar{x}_3 \end{aligned}$$

Now the two newly generated terms, $\bar{x}_1\bar{x}_3$ and $x_1\bar{x}_3$, can be combined further as

$$\begin{aligned} \bar{x}_1\bar{x}_3 + x_1\bar{x}_3 &= (\bar{x}_1 + x_1)\bar{x}_3 \\ &= 1 \cdot \bar{x}_3 \\ &= \bar{x}_3 \end{aligned}$$

These optimization steps indicate that we can replace the four minterms m_0, m_2, m_4, and m_6 with the single product term $\bar{x}_3$. In other words, the minterms m_0, m_2, m_4, and m_6 are all *included* in the term $\bar{x}_3$. The remaining minterm in f is m_5. It can be combined with m_4, which gives

$$x_1\bar{x}_2\bar{x}_3 + x_1\bar{x}_2x_3 = x_1\bar{x}_2$$

Recall that theorem 7*b* in section 2.5 indicates that

$$m_4 = m_4 + m_4$$

which means that we can use the minterm m_4 twice—to combine with minterms m_0, m_2, and m_6 to yield the term $\bar{x}_3$ as explained above and also to combine with m_5 to yield the term $x_1\bar{x}_2$.

We have now accounted for all the minterms in f; hence all five input valuations for which $f = 1$ are covered by the minimum-cost expression

$$f = \bar{x}_3 + x_1\bar{x}_2$$

The expression has the product term $\bar{x}_3$ because $f = 1$ when $x_3 = 0$ regardless of the values of x_1 and x_2. The four minterms m_0, m_2, m_4, and m_6 represent all possible minterms for which $x_3 = 0$; they include all four valuations, 00, 01, 10, and 11, of variables x_1 and x_2. Thus if $x_3 = 0$, then it is guaranteed that $f = 1$. This may not be easy to see directly from the truth table in Figure 4.1, but it is obvious if we write the corresponding valuations grouped together:

	x_1	x_2	x_3
m_0	0	0	0
m_2	0	1	0
m_4	1	0	0
m_6	1	1	0

In a similar way, if we look at m_4 and m_5 as a group of two

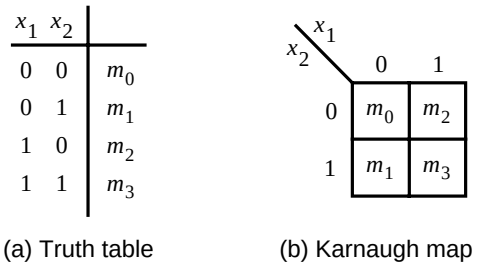
	x_1	x_2	x_3
m_4	1	0	0
m_5	1	0	1

it is clear that when $x_1 = 1$ and $x_2 = 0$, then $f = 1$ regardless of the value of x_3.

The preceding discussion suggests that it would be advantageous to devise a method that allows easy discovery of groups of minterms for which $f = 1$ that can be combined into single terms. The Karnaugh map is a useful vehicle for this purpose.

The *Karnaugh map* [1] is an alternative to the truth-table form for representing a function. The map consists of *cells* that correspond to the rows of the truth table. Consider the two-variable example in Figure 4.2. Part (*a*) depicts the truth-table form, where each of the four rows is identified by a minterm. Part (*b*) shows the Karnaugh map, which has four cells. The columns of the map are labeled by the value of x_1, and the rows are labeled by x_2. This labeling leads to the locations of minterms as shown in the figure. Compared to the truth table, the advantage of the Karnaugh map is that it allows easy recognition of minterms that can be combined using property 14*a* from section 2.5. Minterms in any two cells that are adjacent, either in the same row or the same column, can be combined. For example, the minterms m_2 and m_3 can be combined as

$$\begin{aligned} m_2 + m_3 &= x_1\bar{x}_2 + x_1x_2 \\ &= x_1(\bar{x}_2 + x_2) \\ &= x_1 \cdot 1 \\ &= x_1 \end{aligned}$$



(a) Truth table (b) Karnaugh map

Figure 4.2 Location of two-variable minterms.

The Karnaugh map is not just useful for combining pairs of minterms. As we will see in several larger examples, the Karnaugh map can be used directly to derive a minimum-cost circuit for a logic function.

Two-Variable Map

A Karnaugh map for a two-variable function is given in Figure 4.3. It corresponds to the function f of Figure 2.15. The value of f for each valuation of the variables x_1 and x_2 is indicated in the corresponding cell of the map. Because a 1 appears in both cells of the bottom row and these cells are adjacent, there exists a single product term that can cause f to be equal to 1 when the input variables have the values that correspond to either of these cells. To indicate this fact, we have circled the cell entries in the map. Rather than using the combining property formally, we can derive the product term intuitively. Both of the cells are identified by $x_2 = 1$, but $x_1 = 0$ for the left cell and $x_1 = 1$ for the right cell. Thus if $x_2 = 1$, then $f = 1$ regardless of whether x_1 is equal to 0 or 1. The product term representing the two cells is simply x_2.

Similarly, $f = 1$ for both cells in the first column. These cells are identified by $x_1 = 0$. Therefore, they lead to the product term $\bar{x}_1$. Since this takes care of all instances where $f = 1$, it follows that the minimum-cost realization of the function is

$$f = x_2 + \bar{x}_1$$

Evidently, to find a minimum-cost implementation of a given function, it is necessary to find the smallest number of product terms that produce a value of 1 for all cases where

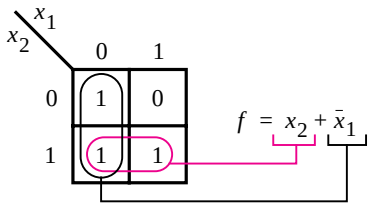


Figure 4.3 The function of Figure 2.15.

$f = 1$. Moreover, the cost of these product terms should be as low as possible. Note that a product term that covers two adjacent cells is cheaper to implement than a term that covers only a single cell. For our example once the two cells in the bottom row have been covered by the product term x_2, only one cell (top left) remains. Although it could be covered by the term $\bar{x}_1\bar{x}_2$, it is better to combine the two cells in the left column to produce the product term $\bar{x}_1$ because this term is cheaper to implement.

Three-Variable Map

A three-variable Karnaugh map is constructed by placing 2 two-variable maps side by side. Figure 4.4 shows the map and indicates the locations of minterms in it. In this case each valuation of x_1 and x_2 identifies a column in the map, while the value of x_3 distinguishes the two rows. To ensure that minterms in the adjacent cells in the map can always be combined into a single product term, the adjacent cells must differ in the value of only one variable. Thus the columns are identified by the sequence of (x_1, x_2) values of 00, 01, 11, and 10, rather than the more obvious 00, 01, 10, and 11. This makes the second and third columns different only in variable x_1. Also, the first and the fourth columns differ only in variable x_1, which means that these columns can be considered as being adjacent. The reader may find it useful to visualize the map as a rectangle folded into a cylinder where the left and the right edges in Figure 4.4*b* are made to touch. (A sequence of codes, or valuations, where consecutive codes differ in one variable only is known as the *Gray code*. This code is used for a variety of purposes, some of which will be encountered later in the book.)

Figure 4.5*a* represents the function of Figure 2.18 in Karnaugh-map form. To synthesize this function, it is necessary to cover the four 1s in the map as efficiently as possible. It is not difficult to see that two product terms suffice. The first covers the 1s in the top row, which are represented by the term $x_1\bar{x}_3$. The second term is $\bar{x}_2x_3$, which covers the 1s in the bottom row. Hence the function is implemented as

$$f = x_1\bar{x}_3 + \bar{x}_2x_3$$

which describes the circuit obtained in Figure 2.19*a*.

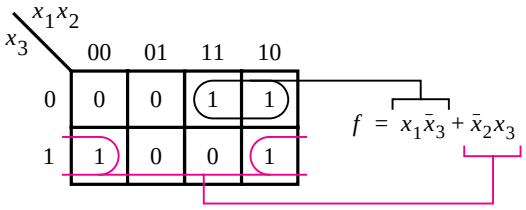
x_1	x_2	x_3	
0	0	0	m_0
0	0	1	m_1
0	1	0	m_2
0	1	1	m_3
1	0	0	m_4
1	0	1	m_5
1	1	0	m_6
1	1	1	m_7

(a) Truth table

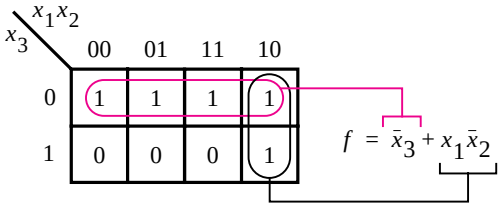
x_3 \ x_1x_2	00	01	11	10
0	m_0	m_2	m_6	m_4
1	m_1	m_3	m_7	m_5

(b) Karnaugh map

Figure 4.4 Location of three-variable minterms.



(a) The function of Figure 2.18



(b) The function of Figure 4.1

Figure 4.5 Examples of three-variable Karnaugh maps.

In a three-variable map it is possible to combine cells to produce product terms that correspond to a single cell, two adjacent cells, or a group of four adjacent cells. Realization of a group of four adjacent cells using a single product term is illustrated in Figure 4.5*b*, using the function from Figure 4.1. The four cells in the top row correspond to the (x_1, x_2, x_3) valuations 000, 010, 110, and 100. As we discussed before, this indicates that if $x_3 = 0$, then $f = 1$ for all four possible valuations of x_1 and x_2, which means that the only requirement is that $x_3 = 0$. Therefore, the product term $\bar{x}_3$ represents these four cells. The remaining 1, corresponding to minterm m_5, is best covered by the term $x_1\bar{x}_2$, obtained by combining the two cells in the right-most column. The complete realization of f is

$$f = \bar{x}_3 + x_1\bar{x}_2$$

It is also possible to have a group of eight 1s in a three-variable map. This is the trivial case where $f = 1$ for all valuations of input variables; in other words, f is equal to the constant 1.

The Karnaugh map provides a simple mechanism for generating the product terms that should be used to implement a given function. A product term must include only those variables that have the same value for all cells in the group represented by this term. If the variable is equal to 1 in the group, it appears uncomplemented in the product term; if it is equal to 0, it appears complemented. Each variable that is sometimes 1 and sometimes 0 in the group does not appear in the product term.

Four-Variable Map

A four-variable map is constructed by placing 2 three-variable maps together to create four rows in the same fashion as we used 2 two-variable maps to form the four columns in a

three-variable map. Figure 4.6 shows the structure of the four-variable map and the location of minterms. We have included in this figure another frequently used way of designating the rows and columns. As shown in blue, it is sufficient to indicate the rows and columns for which a given variable is equal to 1. Thus $x_1 = 1$ for the two right-most columns, $x_2 = 1$ for the two middle columns, $x_3 = 1$ for the bottom two rows, and $x_4 = 1$ for the two middle rows.

Figure 4.7 gives four examples of four-variable functions. The function f_1 has a group of four 1s in adjacent cells in the bottom two rows, for which $x_2 = 0$ and $x_3 = 1$—they are represented by the product term $\bar{x}_2 x_3$. This leaves the two 1s in the second row to be covered, which can be accomplished with the term $x_1 \bar{x}_3 x_4$. Hence the minimum-cost implementation of the function is

$$f_1 = \bar{x}_2 x_3 + x_1 \bar{x}_3 x_4$$

The function f_2 includes a group of eight 1s that can be implemented by a single term, x_3. Again, the reader should note that if the remaining two 1s were implemented separately, the result would be the product term $x_1 \bar{x}_3 x_4$. Implementing these 1s as a part of a group of four 1s, as shown in the figure, gives the less expensive product term $x_1 x_4$.

Just as the left and the right edges of the map are adjacent in terms of the assignment of the variables, so are the top and the bottom edges. Indeed, the four corners of the map are adjacent to each other and thus can form a group of four 1s, which may be implemented by the product term $\bar{x}_2 \bar{x}_4$. This case is depicted by the function f_3. In addition to this group of 1s, there are four other 1s that must be covered to implement f_3. This can be done as shown in the figure.

In all examples that we have considered so far, a unique solution exists that leads to a minimum-cost circuit. The function f_4 provides an example where there is some choice. The groups of four 1s in the top-left and bottom-right corners of the map are realized by the terms $\bar{x}_1 \bar{x}_3$ and $x_1 x_3$, respectively. This leaves the two 1s that correspond to the term $x_1 x_2 \bar{x}_3$. But these two 1s can be realized more economically by treating them as a part of a group of four 1s. They can be included in two different groups of four, as shown in the figure. One

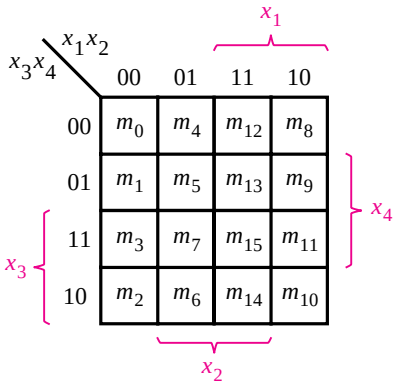


Figure 4.6 A four-variable Karnaugh map.

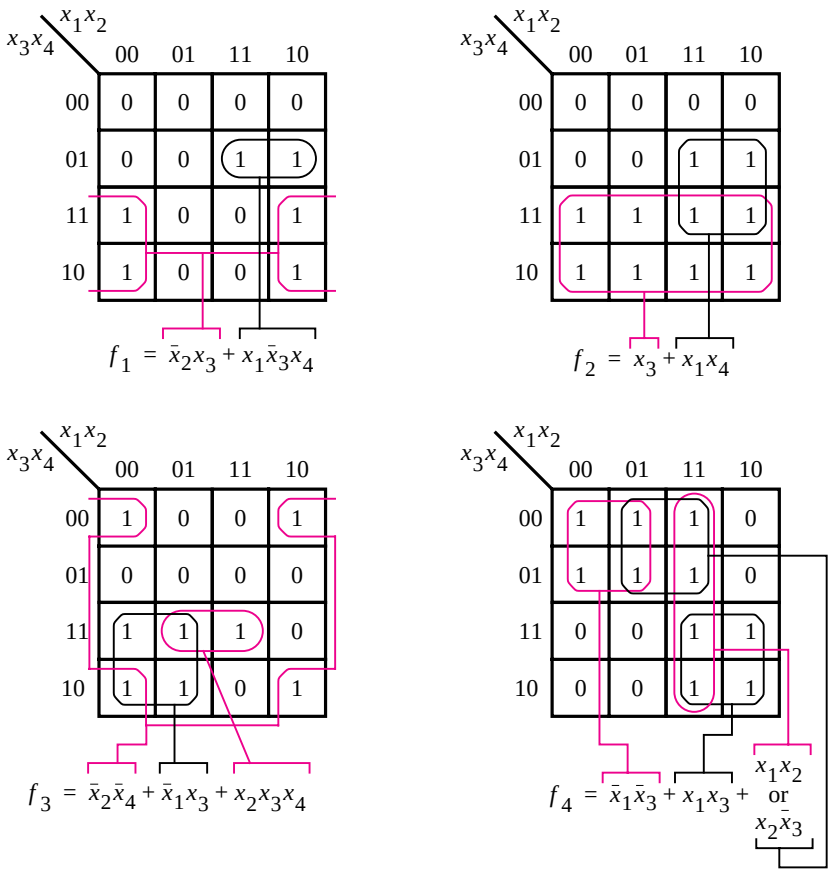


Figure 4.7 Examples of four-variable Karnaugh maps.

choice leads to the product term x_1x_2, and the other leads to $x_2\bar{x}_3$. Both of these terms have the same cost; hence it does not matter which one is chosen in the final circuit. Note that the complement of x_3 in the term $x_2\bar{x}_3$ does not imply an increased cost in comparison with x_1x_2, because this complement must be generated anyway to produce the term $\bar{x}_1\bar{x}_3$, which is included in the implementation.

Five-Variable Map

We can use 2 four-variable maps to construct a five-variable map. It is easy to imagine a structure where one map is directly behind the other, and they are distinguished by $x_5 = 0$ for one map and $x_5 = 1$ for the other map. Since such a structure is awkward to draw, we can simply place the two maps side by side as shown in Figure 4.8. For the logic function given in this example, two groups of four 1s appear in the same place in both four-variable maps; hence their realization does not depend on the value of x_5. The same is true for the two groups of two 1s in the second row. The 1 in the top-right corner appears only in the

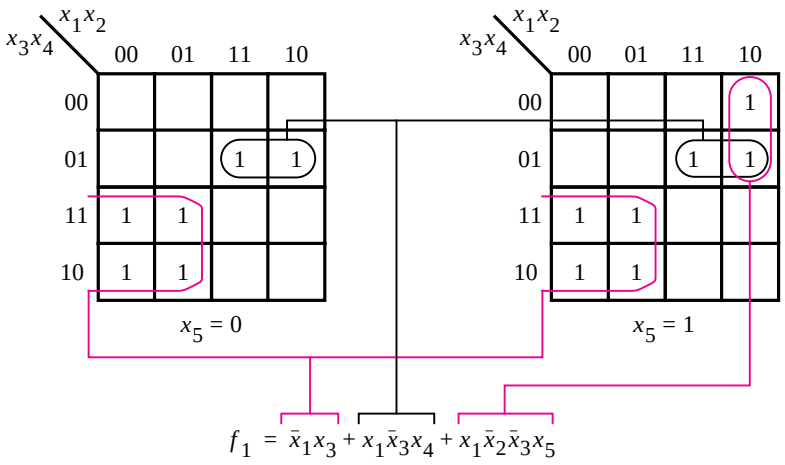


Figure 4.8 A five-variable Karnaugh map.

right map, where $x_5 = 1$; it is a part of the group of two 1s realized by the term $x_1\bar{x}_2\bar{x}_3x_5$. Note that in this map we left blank those cells for which $f = 0$, to make the figure more readable. We will do likewise in a number of maps that follow.

Using a five-variable map is obviously more awkward than using maps with fewer variables. Extending the Karnaugh map concept to more variables is not useful from the practical point of view. This is not troublesome, because practical synthesis of logic functions is done with CAD tools that perform the necessary minimization automatically. Although Karnaugh maps are occasionally useful for designing small logic circuits, our main reason for introducing the Karnaugh maps is to provide a simple vehicle for illustrating the ideas involved in the minimization process.

4.2 STRATEGY FOR MINIMIZATION

For the examples in the preceding section, we used an intuitive approach to decide how the 1s in a Karnaugh map should be grouped together to obtain the minimum-cost implementation of a given function. Our intuitive strategy was to find as few as possible and as large as possible groups of 1s that cover all cases where the function has a value of 1. Each group of 1s has to comprise cells that can be represented by a single product term. The larger the group of 1s, the fewer the number of variables in the corresponding product term. This approach worked well because the Karnaugh maps in our examples were small. For larger logic functions, which have many variables, such intuitive approach is unsuitable. Instead, we must have an organized method for deriving a minimum-cost implementation. In this section we will introduce a possible method, which is similar to the techniques that are automated in CAD tools. To illustrate the main ideas, we will use Karnaugh maps. Later,

in section 4.9, we will describe a different way of representing logic functions, which is used in CAD tools.

4.2.1 TERMINOLOGY

A huge amount of research work has gone into the development of techniques for synthesis of logic functions. The results of this research have been published in numerous papers. To facilitate the presentation of the results, certain terminology has evolved that avoids the need for using highly descriptive phrases. We define some of this terminology in the following paragraphs because it is useful for describing the minimization process.

Literal

A given product term consists of some number of variables, each of which may appear either in uncomplemented or complemented form. Each appearance of a variable, either uncomplemented or complemented, is called a *literal*. For example, the product term $x_1\bar{x}_2x_3$ has three literals, and the term $\bar{x}_1x_3\bar{x}_4x_6$ has four literals.

Implicant

A product term that indicates the input valuation(s) for which a given function is equal to 1 is called an *implicant* of the function. The most basic implicants are the minterms, which we introduced in section 2.6.1. For an n-variable function, a minterm is an implicant that consists of n literals.

Consider the three-variable function in Figure 4.9. There are 11 possible implicants for this function. This includes the five minterms: $\bar{x}_1\bar{x}_2\bar{x}_3$, $\bar{x}_1\bar{x}_2x_3$, $\bar{x}_1x_2\bar{x}_3$, $\bar{x}_1x_2x_3$, and $x_1x_2x_3$. Then there are the implicants that correspond to all possible pairs of minterms that can be combined, namely, $\bar{x}_1\bar{x}_2$ (m_0 and m_1), $\bar{x}_1\bar{x}_3$ (m_0 and m_2), $\bar{x}_1x_3$ (m_1 and m_3), $\bar{x}_1x_2$ (m_2 and m_3), and x_2x_3 (m_3 and m_7). Finally, there is one implicant that covers a group of four minterms, which consists of a single literal $\bar{x}_1$.

Prime Implicant

An implicant is called a *prime implicant* if it cannot be combined into another implicant that has fewer literals. Another way of stating this definition is to say that it is impossible to delete any literal in a prime implicant and still have a valid implicant.

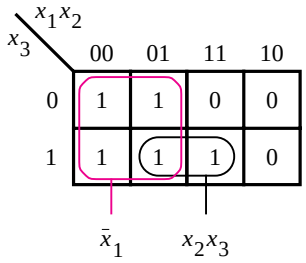


Figure 4.9 Three-variable function $f(x_1, x_2, x_3) = \sum m(0, 1, 2, 3, 7)$.

In Figure 4.9 there are two prime implicants: $\bar{x}_1$ and x_2x_3. It is not possible to delete a literal in either of them. Doing so for $\bar{x}_1$ would make it disappear. For x_2x_3, deleting a literal would leave either x_2 or x_3. But x_2 is not an implicant because it includes the valuation $(x_1, x_2, x_3) = 110$ for which $f = 0$, and x_3 is not an implicant because it includes $(x_1, x_2, x_3) = 101$ for which $f = 0$.

Cover

A collection of implicants that account for all valuations for which a given function is equal to 1 is called a *cover* of that function. A number of different covers exist for most functions. Obviously, a set of all minterms for which $f = 1$ is a cover. It is also apparent that a set of all prime implicants is a cover.

A cover defines a particular implementation of the function. In Figure 4.9 a cover consisting of minterms leads to the expression

$$f = \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 + \bar{x}_1x_2\bar{x}_3 + \bar{x}_1x_2x_3 + x_1x_2x_3$$

Another valid cover is given by the expression

$$f = \bar{x}_1\bar{x}_2 + \bar{x}_1x_2 + x_2x_3$$

The cover comprising the prime implicants is

$$f = \bar{x}_1 + x_2x_3$$

While all of these expressions represent the function f correctly, the cover consisting of prime implicants leads to the lowest-cost implementation.

Cost

In Chapter 2 we suggested that a good indication of the cost of a logic circuit is the number of gates plus the total number of inputs to all gates in the circuit. We will use this definition of cost throughout the book. But we will assume that primary inputs, namely, the input variables, are available in both true and complemented forms at zero cost. Thus the expression

$$f = x_1\bar{x}_2 + x_3\bar{x}_4$$

has a cost of nine because it can be implemented using two AND gates and one OR gate, with six inputs to the AND and OR gates.

If an inversion is needed inside a circuit, then the corresponding NOT gate and its input are included in the cost. For example, the expression

$$g = \overline{x_1\bar{x}_2 + x_3}(\bar{x}_4 + x_5)$$

is implemented using two AND gates, two OR gates, and one NOT gate to complement $(x_1\bar{x}_2 + x_3)$, with nine inputs. Hence the total cost is 14.

4.2.2 MINIMIZATION PROCEDURE

We have seen that it is possible to implement a given logic function with various circuits. These circuits may have different structures and different costs. When designing a logic

circuit, there are usually certain criteria that must be met. One such criterion is likely to be the cost of the circuit, which we considered in the previous discussion. In general, the larger the circuit, the more important the cost issue becomes. In this section we will assume that the main objective is to obtain a minimum-cost circuit.

Having said that cost is the primary concern, we should note that other optimization criteria may be more appropriate in some cases. For instance, in Chapter 3 we described several types of programmable-logic devices (PLDs) that have a predefined basic structure and can be programmed to realize a variety of different circuits. For such devices the main objective is to design a particular circuit so that it will fit into the target device. Whether or not this circuit has the minimum cost is not important if it can be realized successfully on the device. A CAD tool intended for design with a specific device in mind will automatically perform optimizations that are suitable for that device. We will show in section 4.7 that the way in which a circuit should be optimized may be different for different types of devices.

In the previous subsection we concluded that the lowest-cost implementation is achieved when the cover of a given function consists of prime implicants. The question then is how to determine the minimum-cost subset of prime implicants that will cover the function. Some prime implicants may have to be included in the cover, while for others there may be a choice. If a prime implicant includes a minterm for which $f = 1$ that is not included in any other prime implicant, then it must be included in the cover and is called an *essential prime implicant*. In the example in Figure 4.9, both prime implicants are essential. The term x_2x_3 is the only prime implicant that covers the minterm m_7, and $\bar{x}_1$ is the only one that covers the minterms m_0, m_1, and m_2. Notice that the minterm m_3 is covered by both of these prime implicants. The minimum-cost realization of the function is

$$f = \bar{x}_1 + x_2x_3$$

We will now present several examples in which there is a choice as to which prime implicants to include in the final cover. Consider the four-variable function in Figure 4.10. There are five prime implicants: $\bar{x}_1x_3$, $\bar{x}_2x_3$, $x_3\bar{x}_4$, $\bar{x}_1x_2x_4$, and $x_2\bar{x}_3x_4$. The essential ones

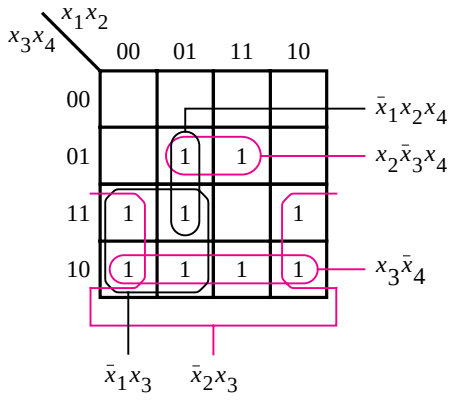


Figure 4.10 Four-variable function $f(x_1, \ldots, x_4) = \sum m(2, 3, 5, 6, 7, 10, 11, 13, 14)$.

(highlighted in blue) are $\bar{x}_2x_3$ (because of m_{11}), $x_3\bar{x}_4$ (because of m_{14}), and $x_2\bar{x}_3x_4$ (because of m_{13}). They must be included in the cover. These three prime implicants cover all minterms for which $f = 1$ except m_7. It is clear that m_7 can be covered by either $\bar{x}_1x_3$ or $\bar{x}_1x_2x_4$. Because $\bar{x}_1x_3$ has a lower cost, it is chosen for the cover. Therefore, the minimum-cost realization is

$$f = \bar{x}_2x_3 + x_3\bar{x}_4 + x_2\bar{x}_3x_4 + \bar{x}_1x_3$$

From the preceding discussion, the process of finding a minimum-cost circuit involves the following steps:

1. Generate all prime implicants for the given function f.
2. Find the set of essential prime implicants.
3. If the set of essential prime implicants covers all valuations for which $f = 1$, then this set is the desired cover of f. Otherwise, determine the nonessential prime implicants that should be added to form a complete minimum-cost cover.

The choice of nonessential prime implicants to be included in the cover is governed by the cost considerations. This choice is often not obvious. Indeed, for large functions there may exist many possibilities, and some *heuristic* approach (i.e., an approach that considers only a subset of possibilities but gives good results most of the time) has to be used. One such approach is to arbitrarily select one nonessential prime implicant and include it in the cover and then determine the rest of the cover. Next, another cover is determined assuming that this prime implicant is not in the cover. The costs of the resulting covers are compared, and the less-expensive cover is chosen for implementation.

We can illustrate the process by using the function in Figure 4.11. Of the six prime implicants, only $\bar{x}_3\bar{x}_4$ is essential. Consider next $x_1x_2\bar{x}_3$ and assume first that it will be included in the cover. Then the remaining three minterms, m_{10}, m_{11}, and m_{15}, will require two more prime implicants to be included in the cover. A possible implementation is

$$f = \bar{x}_3\bar{x}_4 + x_1x_2\bar{x}_3 + x_1x_3x_4 + x_1\bar{x}_2x_3$$

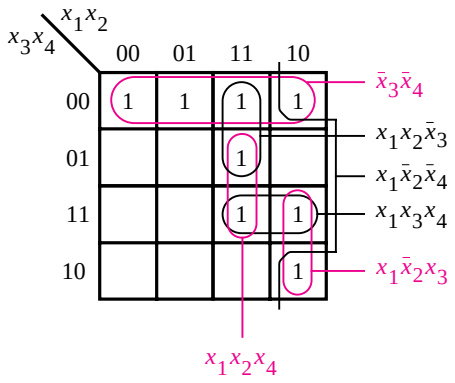


Figure 4.11 The function $f(x_1, \ldots, x_4) = \sum m(0, 4, 8, 10, 11, 12, 13, 15)$.

The second possibility is that $x_1x_2\bar{x}_3$ is not included in the cover. Then $x_1x_2x_4$ becomes essential because there is no other way of covering m_{13}. Because $x_1x_2x_4$ also covers m_{15}, only m_{10} and m_{11} remain to be covered, which can be achieved with $x_1\bar{x}_2x_3$. Therefore, the alternative implementation is

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

Clearly, this implementation is a better choice.

Sometimes there may not be any essential prime implicants at all. An example is given in Figure 4.12. Choosing any of the prime implicants and first including it, then excluding it from the cover leads to two alternatives of equal cost. One includes the prime implicants indicated in black, which yields

$$f = \bar{x}_1\bar{x}_3\bar{x}_4 + x_2\bar{x}_3x_4 + x_1x_3x_4 + \bar{x}_2x_3\bar{x}_4$$

The other includes the prime implicants indicated in blue, which yields

$$f = \bar{x}_1\bar{x}_2\bar{x}_4 + \bar{x}_1x_2\bar{x}_3 + x_1x_2x_4 + x_1\bar{x}_2x_3$$

This procedure can be used to find minimum-cost implementations of both small and large logic functions. For our small examples it was convenient to use Karnaugh maps to determine the prime implicants of a function and then choose the final cover. Other techniques based on the same principles are much more suitable for use in CAD tools; we will introduce one such technique in sections 4.9 and 4.10.

The previous examples have been based on the sum-of-products form. We will next illustrate that the same concepts apply for the product-of-sums form.

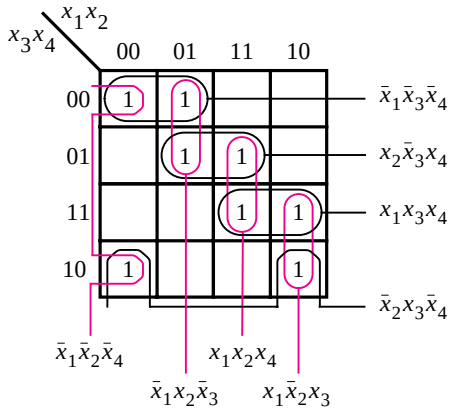


Figure 4.12 The function $f(x_1, \ldots, x_4) = \sum m(0, 2, 4, 5, 10, 11, 13, 15)$.

4.3 MINIMIZATION OF PRODUCT-OF-SUMS FORMS

Now that we know how to find the minimum-cost sum-of-products (SOP) implementations of functions, we can use the same techniques and the principle of duality to obtain minimum-cost product-of-sums (POS) implementations. In this case it is the maxterms for which $f = 0$ that have to be combined into sum terms that are as large as possible. Again, a sum term is considered larger if it covers more maxterms, and the larger the term, the less costly it is to implement.

Figure 4.13 depicts the same function as Figure 4.9 depicts. There are three maxterms that must be covered: M_4, M_5, and M_6. They can be covered by two sum terms shown in the figure, leading to the following implementation:

$$f = (\bar{x}_1 + x_2)(\bar{x}_1 + x_3)$$

A circuit corresponding to this expression has two OR gates and one AND gate, with two inputs for each gate. Its cost is greater than the cost of the equivalent SOP implementation derived in Figure 4.9, which requires only one OR gate and one AND gate.

The function from Figure 4.10 is reproduced in Figure 4.14. The maxterms for which $f = 0$ can be covered as shown, leading to the expression

$$f = (x_2 + x_3)(x_3 + x_4)(\bar{x}_1 + \bar{x}_2 + \bar{x}_3 + \bar{x}_4)$$

This expression represents a circuit with three OR gates and one AND gate. Two of the OR gates have two inputs, and the third has four inputs; the AND gate has three inputs. Assuming that both the complemented and uncomplemented versions of the input variables x_1 to x_4 are available at no extra cost, the cost of this circuit is 15. This compares favorably with the SOP implementation derived from Figure 4.10, which requires five gates and 13 inputs at a total cost of 18.

In general, as we already know from section 2.6.1, the SOP and POS implementations of a given function may or may not entail the same cost. The reader is encouraged to find the POS implementations for the functions in Figures 4.11 and 4.12 and compare the costs with the SOP forms.

We have shown how to obtain minimum-cost POS implementations by finding the largest sum terms that cover all maxterms for which $f = 0$. Another way of obtaining

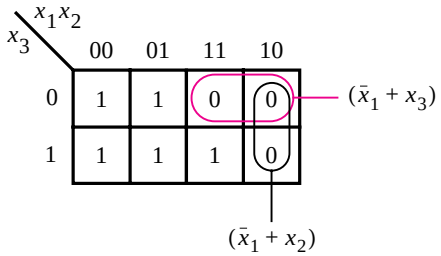


Figure 4.13 POS minimization of $f(x_1, x_2, x_3) = \Pi M(4, 5, 6)$.

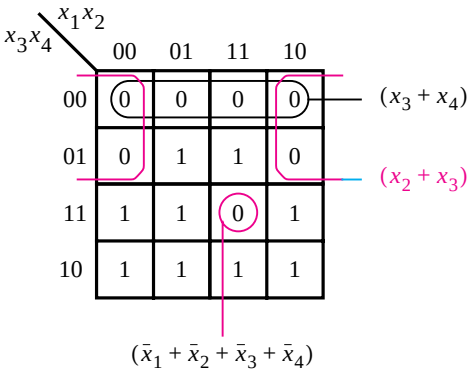


Figure 4.14 POS minimization of $f(x_1, \ldots, x_4) = \Pi M(0, 1, 4, 8, 9, 12, 15)$.

the same result is by finding a minimum-cost SOP implementation of the complement of f. Then we can apply DeMorgan's theorem to this expression to obtain the simplest POS realization because $f = \overline{\overline{f}}$. For example, the simplest SOP implementation of $\overline{f}$ in Figure 4.13 is

$$\overline{f} = x_1\bar{x}_2 + x_1\bar{x}_3$$

Complementing this expression using DeMorgan's theorem yields

$$\begin{aligned} f = \overline{\overline{f}} &= \overline{x_1\bar{x}_2 + x_1\bar{x}_3} \\ &= \overline{x_1\bar{x}_2} \cdot \overline{x_1\bar{x}_3} \\ &= (\bar{x}_1 + x_2)(\bar{x}_1 + x_3) \end{aligned}$$

which is the same result as obtained above.

Using this approach for the function in Figure 4.14 gives

$$\overline{f} = \bar{x}_2\bar{x}_3 + \bar{x}_3\bar{x}_4 + x_1x_2x_3x_4$$

Complementing this expression produces

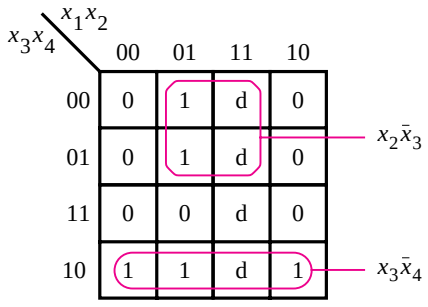
$$\begin{aligned} f = \overline{\overline{f}} &= \overline{\bar{x}_2\bar{x}_3 + \bar{x}_3\bar{x}_4 + x_1x_2x_3x_4} \\ &= \overline{\bar{x}_2\bar{x}_3} \cdot \overline{\bar{x}_3\bar{x}_4} \cdot \overline{x_1x_2x_3x_4} \\ &= (x_2 + x_3)(x_3 + x_4)(\bar{x}_1 + \bar{x}_2 + \bar{x}_3 + \bar{x}_4) \end{aligned}$$

which matches the previously derived implementation.

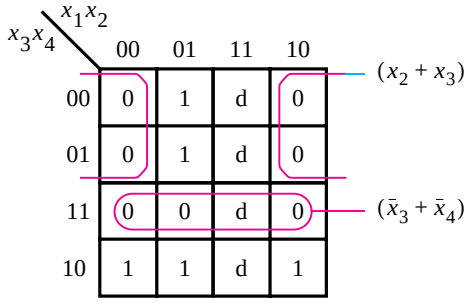
4.4 INCOMPLETELY SPECIFIED FUNCTIONS

In digital systems it often happens that certain input conditions can never occur. For example, suppose that x_1 and x_2 control two interlocked switches such that both switches cannot be closed at the same time. Thus the only three possible states of the switches are that both switches are open or that one switch is open and the other switch is closed. Namely, the input valuations $(x_1, x_2) = 00$, 01, and 10 are possible, but 11 is guaranteed not to occur. Then we say that $(x_1, x_2) = 11$ is a *don't-care condition*, meaning that a circuit with x_1 and x_2 as inputs can be designed by ignoring this condition. A function that has don't-care condition(s) is said to be *incompletely specified*.

Don't-care conditions, or *don't cares* for short, can be used to advantage in the design of logic circuits. Since these input valuations will never occur, the designer may assume that the function value for these valuations is either 1 or 0, whichever is more useful in trying to find a minimum-cost implementation. Figure 4.15 illustrates this idea. The required function has a value of 1 for minterms m_2, m_4, m_5, m_6, and m_{10}. Assuming the above-



(a) SOP implementation



(b) POS implementation

Figure 4.15 Two implementations of the function $f(x_1, \ldots, x_4) = \sum m(2, 4, 5, 6, 10) + D(12, 13, 14, 15)$.

mentioned interlocked switches, the x_1 and x_2 inputs will never be equal to 1 at the same time; hence the minterms m_{12}, m_{13}, m_{14}, and m_{15} can all be used as don't cares. The don't cares are denoted by the letter d in the map. Using the shorthand notation, the function f is specified as

$$f(x_1, \ldots, x_4) = \sum m(2, 4, 5, 6, 10) + D(12, 13, 14, 15)$$

where D is the set of don't cares.

Part (a) of the figure indicates the best sum-of-products implementation. To form the largest possible groups of 1s, thus generating the lowest-cost prime implicants, it is necessary to assume that the don't cares D_{12}, D_{13}, and D_{14} (corresponding to minterms m_{12}, m_{13}, and m_{14}) have the value of 1 while D_{15} has the value of 0. Then there are only two prime implicants, which provide a complete cover of f. The resulting implementation is

$$f = x_2\bar{x}_3 + x_3\bar{x}_4$$

Part (b) shows how the best product-of-sums implementation can be obtained. The same values are assumed for the don't cares. The result is

$$f = (x_2 + x_3)(\bar{x}_3 + \bar{x}_4)$$

The freedom in choosing the value of don't cares leads to greatly simplified realizations. If we were to naively exclude the don't cares from the synthesis of the function, by assuming that they always have a value of 0, the resulting SOP expression would be

$$f = \bar{x}_1x_2\bar{x}_3 + \bar{x}_1x_3\bar{x}_4 + \bar{x}_2x_3\bar{x}_4$$

and the POS expression would be

$$f = (x_2 + x_3)(\bar{x}_3 + \bar{x}_4)(\bar{x}_1 + \bar{x}_2)$$

Both of these expressions have higher costs than the expressions obtained with a more appropriate assignment of values to don't cares.

Although don't-care values can be assigned arbitrarily, an arbitrary assignment may not lead to a minimum-cost implementation of a given function. If there are k don't cares, then there are 2^k possible ways of assigning 0 or 1 values to them. In the Karnaugh map we can usually see how best to do this assignment to find the simplest implementation.

Using interlocked switches to illustrate how don't-care conditions can occur in a real system may seem to be somewhat contrived. However, in Chapters 6, 8, and 9 we will encounter many examples of don't cares that occur in the course of practical design of digital circuits.

4.5 MULTIPLE-OUTPUT CIRCUITS

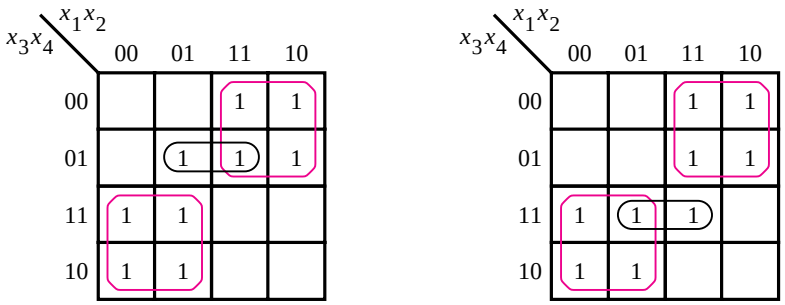
In all previous examples we have considered single functions and their circuit implementations. In practical digital systems it is necessary to implement a number of functions as part of some large logic circuit. Circuits that implement these functions can often be

combined into a less-expensive single circuit with multiple outputs by sharing some of the gates needed in the implementation of individual functions.

Example 4.1 An example of gate sharing is given in Figure 4.16. Two functions, f_1 and f_2, of the same variables are to be implemented. The minimum-cost implementations for these functions are obtained as shown in parts (*a*) and (*b*) of the figure. This results in the expressions

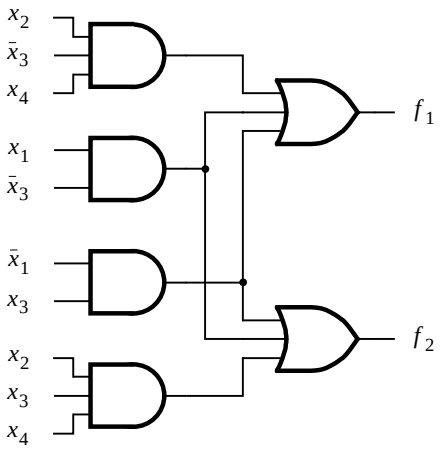
$$f_1 = x_1\bar{x}_3 + \bar{x}_1x_3 + x_2\bar{x}_3x_4$$
$$f_2 = x_1\bar{x}_3 + \bar{x}_1x_3 + x_2x_3x_4$$

The cost of f_1 is four gates and 10 inputs, for a total of 14. The cost of f_2 is the same. Thus the total cost is 28 if both functions are implemented by separate circuits. A less-expensive



(a) Function f_1

(b) Function f_2



(c) Combined circuit for f_1 and f_2

Figure 4.16 An example of multiple-output synthesis.

realization is possible if the two circuits are combined into a single circuit with two outputs. Because the first two product terms are identical in both expressions, the AND gates that implement them need not be duplicated. The combined circuit is shown in Figure 4.16*c*. Its cost is six gates and 16 inputs, for a total of 22.

In this example we reduced the overall cost by finding minimum-cost realizations of f_1 and f_2 and then sharing the gates that implement the common product terms. This strategy does not necessarily always work the best, as the next example shows.

Example 4.2

Figure 4.17 shows two functions to be implemented by a single circuit. Minimum-cost realizations of the individual functions f_3 and f_4 are obtained from parts (*a*) and (*b*) of the figure.

$$f_3 = \bar{x}_1x_4 + x_2x_4 + \bar{x}_1x_2x_3$$
$$f_4 = x_1x_4 + \bar{x}_2x_4 + \bar{x}_1x_2x_3\bar{x}_4$$

None of the AND gates can be shared, which means that the cost of the combined circuit would be six AND gates, two OR gates, and 21 inputs, for a total of 29.

But several alternative realizations are possible. Instead of deriving the expressions for f_3 and f_4 using only prime implicants, we can look for other implicants that may be shared advantageously in the combined realization of the functions. Figure 4.17*c* shows the best choice of implicants, which yields the realization

$$f_3 = x_1x_2x_4 + \bar{x}_1x_2x_3\bar{x}_4 + \bar{x}_1x_4$$
$$f_4 = x_1x_2x_4 + \bar{x}_1x_2x_3\bar{x}_4 + \bar{x}_2x_4$$

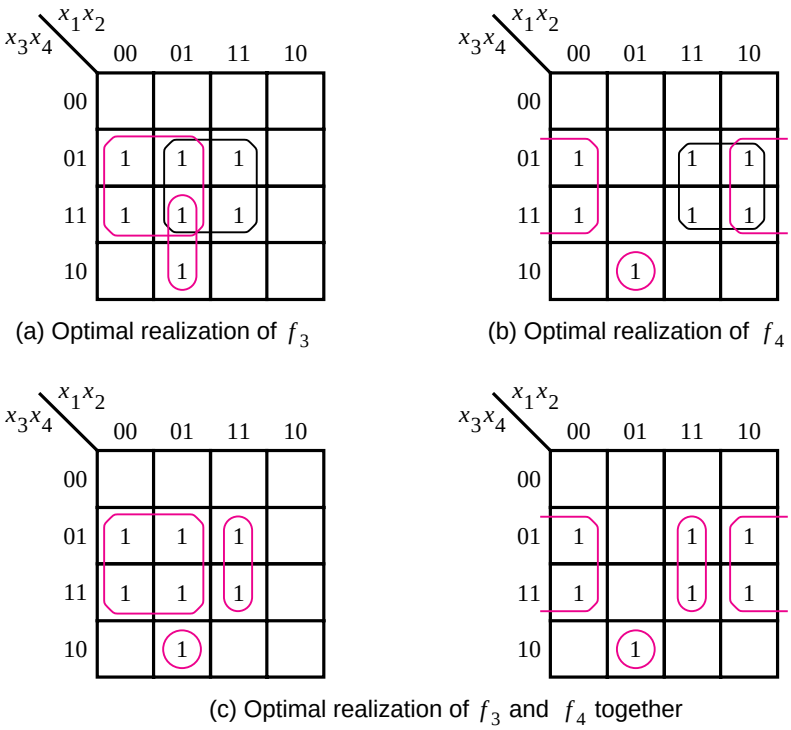
The first two implicants are identical in both expressions. The resulting circuit is given in Figure 4.17*d*. It has the cost of six gates and 17 inputs, for a total of 23.

Example 4.3

In Example 4.1 we sought the best SOP implementation for the functions f_1 and f_2 in Figure 4.16. We will now consider the POS implementation of the same functions. The minimum-cost POS expressions for f_1 and f_2 are

$$f_1 = (\bar{x}_1 + \bar{x}_3)(x_1 + x_2 + x_3)(x_1 + x_3 + x_4)$$
$$f_2 = (x_1 + x_3)(\bar{x}_1 + x_2 + \bar{x}_3)(\bar{x}_1 + \bar{x}_3 + x_4)$$

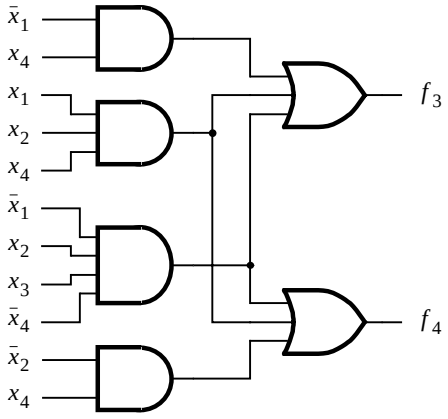
There are no common sum terms in these expressions that could be shared in the implementation. Moreover, from the Karnaugh maps in Figure 4.16, it is apparent that there is no sum term (covering the cells where $f_1 = f_2 = 0$) that can be profitably used in realizing both f_1 and f_2. Thus the best choice is to implement each function separately, according to the preceding expressions. Each function requires three OR gates, one AND gate, and 11 inputs. Therefore, the total cost of the circuit that implements both functions is 30. This realization is costlier than the SOP realization derived in Example 4.1.



(a) Optimal realization of f_3

(b) Optimal realization of f_4

(c) Optimal realization of f_3 and f_4 together



(d) Combined circuit for f_3 and f_4

Figure 4.17 Another example of multiple-output synthesis.

Example 4.4

Consider now the POS realization of the functions f_3 and f_4 in Figure 4.17. The minimum-cost POS expressions for f_3 and f_4 are

$$f_3 = (x_3 + x_4)(x_2 + x_4)(\bar{x}_1 + x_4)(\bar{x}_1 + x_2)$$
$$f_4 = (x_3 + x_4)(x_2 + x_4)(\bar{x}_1 + x_4)(x_1 + \bar{x}_2 + \bar{x}_4)$$

The first three sum terms are the same in both f_3 and f_4; they can be shared in a combined circuit. These terms require three OR gates and six inputs. In addition, one 2-input OR gate and one 4-input AND gate are needed for f_3, and one 3-input OR gate and one 4-input AND gate are needed for f_4. Thus the combined circuit comprises five OR gates, two AND gates, and 19 inputs, for a total cost of 26. This cost is slightly higher than the cost of the circuit derived in Example 4.2.

These examples show that the complexities of the best SOP or POS implementations of given functions may be quite different. For the functions in Figures 4.16 and 4.17, the SOP form gives better results. But if we are interested in implementing the complements of the four functions in these figures, then the POS form would be less costly.

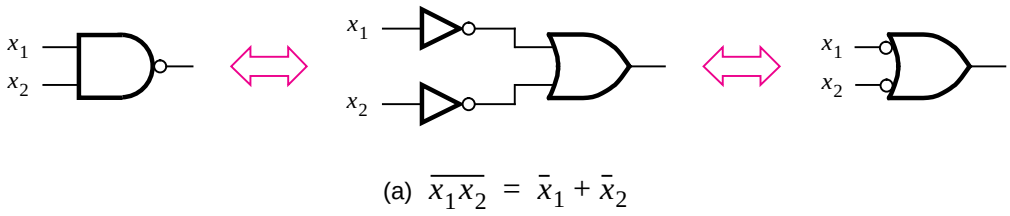
Sophisticated CAD tools used to synthesize logic functions will automatically perform the types of optimizations illustrated in the preceding examples.

4.6 NAND AND NOR LOGIC NETWORKS

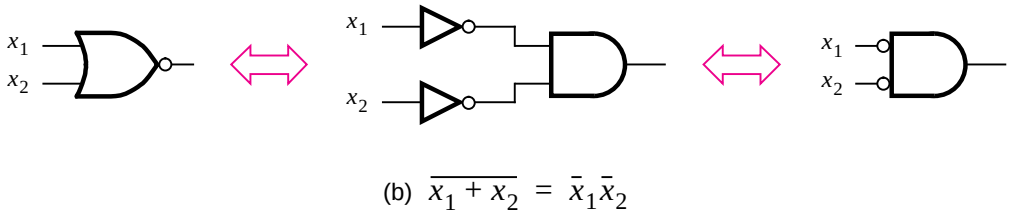
In Chapter 3 we saw that it is possible to design electronic circuits that realize basic logic functions other than AND, OR, and NOT, which have been the focus of our discussion to this point. From Figures 3.6 to 3.9 and Figures 3.13 to 3.15, it is obvious that NAND and NOR gates are simpler to implement than AND and OR gates. Then we should ask whether these gates can be used directly in the synthesis of logic circuits, rather than just being a part of the individual AND and OR gates. In section 2.5 we introduced DeMorgan's theorem. Its logic gate interpretation is shown in Figure 4.18. Identity 15*a* from section 2.5 is interpreted in part (*a*) of the figure. It specifies that a NAND of variables x_1 and x_2 is equivalent to first complementing each of the variables and then ORing them. Notice on the far-right side that we have indicated the NOT gates simply as small circles, which denote inversion of the logic value at that point. The other half of DeMorgan's theorem, identity 15*b*, appears in part (*b*) of the figure. It states that the NOR function is equivalent to first inverting the input variables and then ANDing them.

In previous sections we explained how any logic function can be implemented either in sum-of-products or product-of-sums form, which leads to logic networks that have either an AND-OR or an OR-AND structure, respectively. We will now show that such networks can be implemented using only NAND gates or only NOR gates.

Consider the network in Figure 4.19 as a representative of general AND-OR networks. This network can be transformed into a network of NAND gates as shown in the figure. First, each connection between the AND gate and an OR gate is replaced by a connection



(a) $\overline{x_1 x_2} = \bar{x}_1 + \bar{x}_2$



(b) $\overline{x_1 + x_2} = \bar{x}_1 \bar{x}_2$

Figure 4.18 DeMorgan's theorem in terms of logic gates.

that includes two inversions of the signal: one inversion at the output of the AND gate and the other at the input of the OR gate. Such double inversion has no effect on the behavior of the network, as stated formally in theorem 9 in section 2.5. According to Figure 4.18*a*, the OR gate with inversions at its inputs is equivalent to a NAND gate. Thus we can redraw the network using only NAND gates, as shown in Figure 4.19. This example shows that

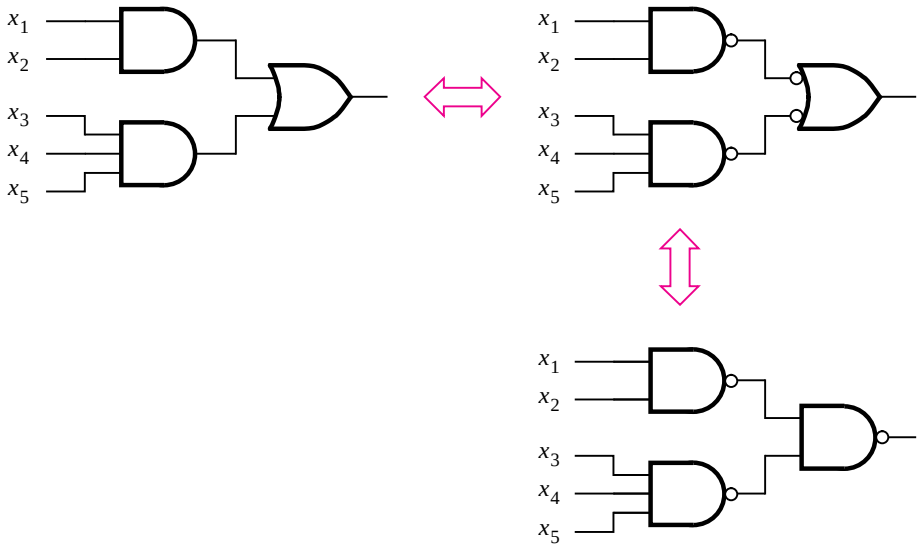


Figure 4.19 Using NAND gates to implement a sum-of-products.

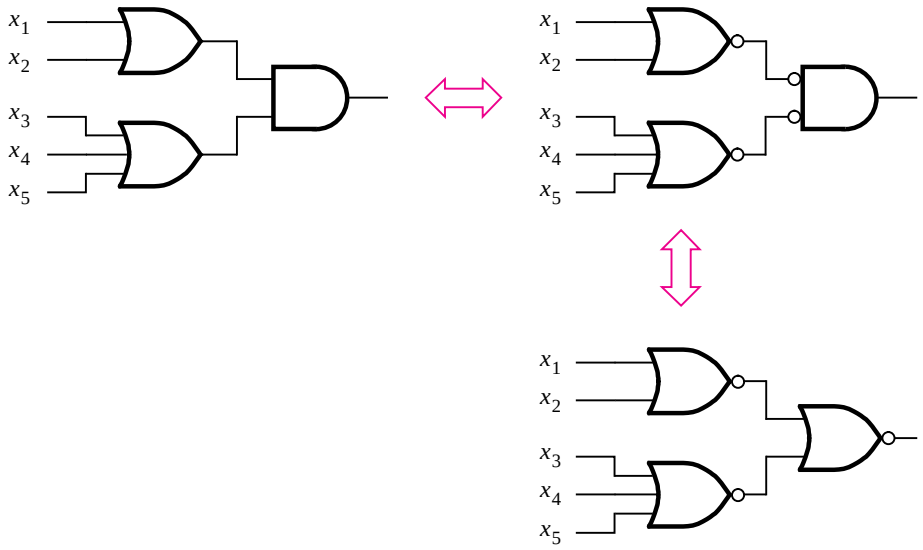


Figure 4.20 Using NOR gates to implement a product-of-sums.

any AND-OR network can be implemented as a NAND-NAND network having the same topology.

Figure 4.20 gives a similar construction for a product-of-sums network, which can be transformed into a circuit with only NOR gates. The procedure is exactly the same as the one described for Figure 4.19 except that now the identity in Figure 4.18*b* is applied. The conclusion is that any OR-AND network can be implemented as a NOR-NOR network having the same topology.

4.7 MULTILEVEL SYNTHESIS

In the preceding sections our objective was to find a minimum-cost sum-of-products or product-of-sums realization of a given logic function. Logic circuits of this type have *two levels* (stages) of gates. In the sum-of-products form, the first level comprises AND gates that are connected to a second-level OR gate. In the product-of-sums form, the first-level OR gates feed the second-level AND gate. We have assumed that both true and complemented versions of the input variables are available so that NOT gates are not needed to complement the variables.

A two-level realization is usually efficient for functions of a few variables. However, as the number of inputs increases, a two-level circuit may result in fan-in problems. Whether or not this is an issue depends on the type of technology that is used to implement the circuit. For example, consider the following function:

$$f(x_1, \ldots, x_7) = x_1x_3\bar{x}_6 + x_1x_4x_5\bar{x}_6 + x_2x_3x_7 + x_2x_4x_5x_7$$

This is a minimum-cost SOP expression. Now consider implementing f in two types of PLDs: a CPLD and an FPGA. Figure 4.21 shows one of the PAL-like blocks from Figure 3.33. The figure indicates in blue the circuitry used to realize the function f. Clearly, the SOP form of the function is well suited to the chip architecture of the CPLD.

Next, consider implementing f in an FPGA. For this example we will use the FPGA shown in Figure 3.39, which contains two-input LUTs. Since the SOP expression for f requires three- and four-input AND operations and a four-input OR, it cannot be directly implemented in this FPGA. The problem is that the fan-in required to implement the function is too high for our target chip architecture.

To solve the fan-in problem, f must be expressed in a form that has more than two levels of logic operations. Such a form is called a *multilevel* logic expression. There are several different approaches for synthesis of multilevel circuits. We will discuss two important techniques known as *factoring* and *functional decomposition*.

4.7.1 FACTORING

The distributive property in section 2.5 allows us to factor the preceding expression for f as follows

$$\begin{aligned} f &= x_1\bar{x}_6(x_3 + x_4x_5) + x_2x_7(x_3 + x_4x_5) \\ &= (x_1\bar{x}_6 + x_2x_7)(x_3 + x_4x_5) \end{aligned}$$

The corresponding circuit has a maximum fan-in of two; hence it can be realized using two-input LUTs. Figure 4.22 gives a possible implementation using the FPGA from Figure 3.39. Note that a two-variable function that has to be realized by each LUT is indicated in the box that represents the LUT.

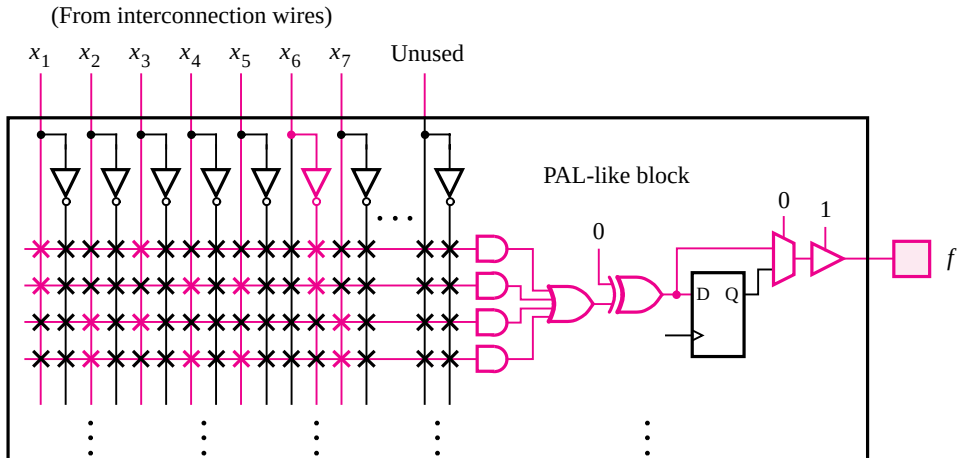


Figure 4.21 Implementation in a CPLD.

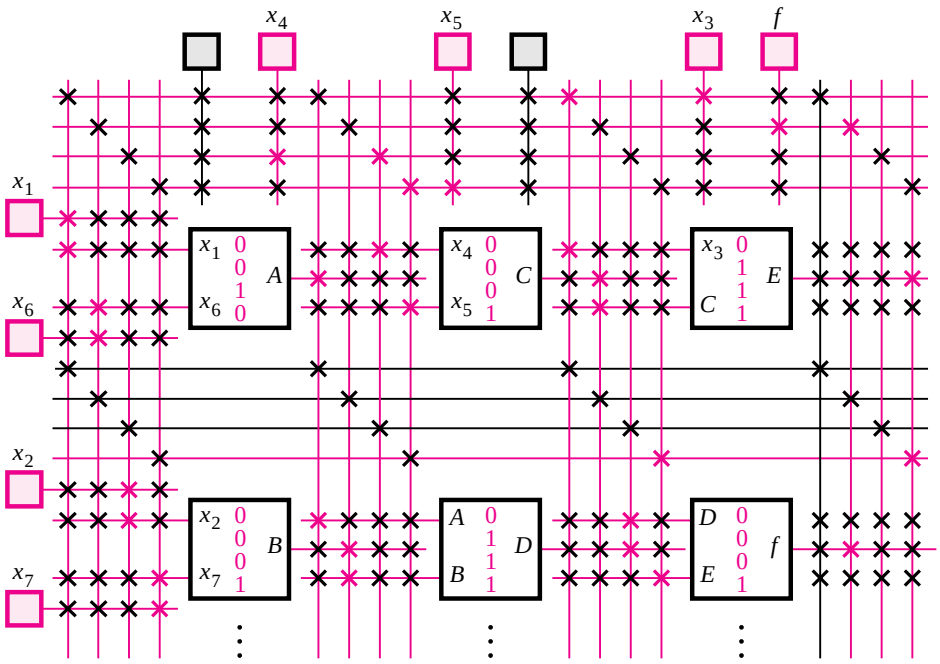


Figure 4.22 Implementation in an FPGA.

Fan-in Problem

In the preceding example, the fan-in restrictions were caused by the fixed structure of the FPGA, where each LUT has only two inputs. However, even when the target chip architecture is not fixed, the fan-in may still be an issue. To illustrate this situation, let us consider the implementation of a circuit in a custom chip. Recall that custom chips usually contain a large number of gates. If the chip is fabricated using CMOS technology, then there will be fan-in limitations as discussed in section 3.8.8. In this technology the number of inputs to a logic gate should be small. For instance, we may wish to limit the number of inputs to an AND gate to be less than five. Under this restriction, if a logic expression includes a seven-input product term, we would have to use 2 four-input AND gates, as indicated in Figure 4.23.

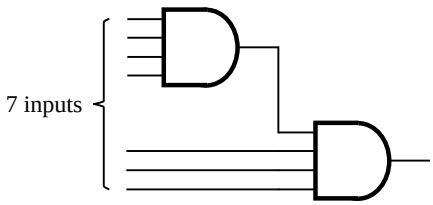


Figure 4.23 Using four-input AND gates to realize a seven-input product term.

Factoring can be used to deal with the fan-in problem. Suppose again that the available gates have a maximum fan-in of four and that we want to realize the function

$$f = x_1\bar{x}_2x_3\bar{x}_4x_5x_6 + x_1x_2\bar{x}_3\bar{x}_4\bar{x}_5x_6$$

This is a minimal sum-of-products expression. Using the approach of Figure 4.23, we will need four AND gates and one OR gate to implement this expression. A better solution is to factor the expression as follows

$$f = x_1\bar{x}_4x_6(\bar{x}_2x_3x_5 + x_2\bar{x}_3\bar{x}_5)$$

Then three AND gates and one OR gate suffice for realization of the required function, as shown in Figure 4.24.

Example 4.5 In practical situations a designer of logic circuits often encounters specifications that naturally lead to an initial design where the logic expressions are in a factored form. Suppose we need a circuit that meets the following requirements. There are four inputs: x_1, x_2, x_3, and x_4. An output, f_1, must have the value 1 if at least one of the inputs x_1 and x_2 is equal to 1 and both x_3 and x_4 are equal to 1; it must also be 1 if $x_1 = x_2 = 0$ and either x_3 or x_4 is 1. In all other cases $f_1 = 0$. A different output, f_2, is to be equal to 1 in all cases except when both x_1 and x_2 are equal to 0 or when both x_3 and x_4 are equal to 0.

From this specification, the function f_1 can be expressed as

$$f_1 = (x_1 + x_2)x_3x_4 + \bar{x}_1\bar{x}_2(x_3 + x_4)$$

This expression can be simplified to

$$f_1 = x_3x_4 + \bar{x}_1\bar{x}_2(x_3 + x_4)$$

which the reader can verify by using a Karnaugh map.

The second function, f_2, is most easily defined in terms of its complement, such that

$$\bar{f}_2 = \bar{x}_1\bar{x}_2 + \bar{x}_3\bar{x}_4$$

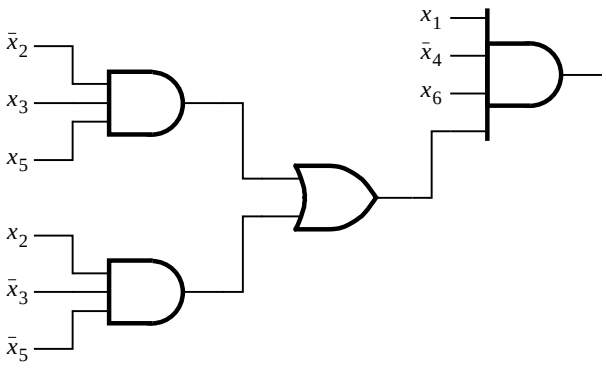


Figure 4.24 A factored circuit.

Then using DeMorgan's theorem gives

$$f_2 = (x_1 + x_2)(x_3 + x_4)$$

which is the minimum-cost expression for f_2; the cost increases significantly if the SOP form is used.

Because our objective is to design the lowest-cost combined circuit that implements f_1 and f_2, it seems that the best result can be achieved if we use the factored forms for both functions, in which case the sum term $(x_3 + x_4)$ can be shared. Moreover, observing that $\overline{x}_1\overline{x}_2 = \overline{x_1 + x_2}$, the sum term $(x_1 + x_2)$ can also be shared if we express f_1 in the form

$$f_1 = x_3x_4 + \overline{x_1 + x_2}(x_3 + x_4)$$

Then the combined circuit, shown in Figure 4.25, comprises three OR gates, three AND gates, one NOT gate, and 13 inputs, for a total of 20.

Impact on Wiring Complexity

The space on integrated circuit chips is occupied by the circuitry that implements logic gates and by the wires needed to make connections among the gates. The amount of space needed for wiring is a substantial portion of the chip area. Therefore, it is useful to keep the wiring complexity as low as possible.

In a logic expression each literal corresponds to a wire in the circuit that carries the desired logic signal. Since factoring usually reduces the number of literals, it provides a powerful mechanism for reducing the wiring complexity in a logic circuit. In the synthesis process the CAD tools consider many different issues, including the cost of the circuit, the fan-in, and the wiring complexity.

4.7.2 FUNCTIONAL DECOMPOSITION

In the preceding examples, which illustrated the factoring approach, multilevel circuits were used to deal with fan-in limitations. However, such circuits may be preferable to their two-level equivalents even if fan-in is not a problem. In some cases the multilevel circuits

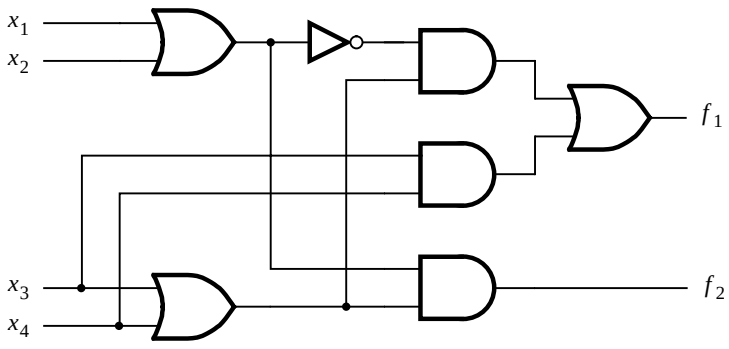


Figure 4.25 Circuit for Example 4.5.

may reduce the cost of implementation. On the other hand, they usually imply longer propagation delays, because they use multiple stages of logic gates. We will explore these issues by means of illustrative examples.

Complexity of a logic circuit, in terms of wiring and logic gates, can often be reduced by *decomposing* a two-level circuit into subcircuits, where one or more subcircuits implement functions that may be used in several places to construct the final circuit. To achieve this objective, a two-level logic expression is replaced by two or more new expressions, which are then combined to define a multilevel circuit. We can illustrate this idea by a simple example.

Example 4.6 Consider the minimum-cost sum-of-products expression

$$f = \bar{x}_1 x_2 x_3 + x_1 \bar{x}_2 x_3 + x_1 x_2 x_4 + \bar{x}_1 \bar{x}_2 x_4$$

and assume that the inputs x_1 to x_4 are available only in their true form. Then the expression defines a circuit that has four AND gates, one OR gate, two NOT gates, and 18 inputs (wires) to all gates. The fan-in is three for the AND gates and four for the OR gate. The reader should observe that in this case we have included the cost of NOT gates needed to complement x_1 and x_2, rather than assume that both true and complemented versions of all input variables are available, as we had done before.

Factoring x_3 from the first two terms and x_4 from the last two terms, this expression becomes

$$f = (\bar{x}_1 x_2 + x_1 \bar{x}_2) x_3 + (x_1 x_2 + \bar{x}_1 \bar{x}_2) x_4$$

Now let $g(x_1, x_2) = \bar{x}_1 x_2 + x_1 \bar{x}_2$ and observe that

$$\begin{aligned}
\bar{g} &= \overline{\bar{x}_1 x_2 + x_1 \bar{x}_2} \\
&= \overline{\bar{x}_1 x_2} \cdot \overline{x_1 \bar{x}_2} \\
&= (x_1 + \bar{x}_2)(\bar{x}_1 + x_2) \\
&= x_1 \bar{x}_1 + x_1 x_2 + \bar{x}_2 \bar{x}_1 + \bar{x}_2 x_2 \\
&= 0 + x_1 x_2 + \bar{x}_1 \bar{x}_2 + 0 \\
&= x_1 x_2 + \bar{x}_1 \bar{x}_2
\end{aligned}$$

Then f can be written as

$$f = g x_3 + \bar{g} x_4$$

which leads to the circuit shown in Figure 4.26. This circuit requires an additional OR gate and a NOT gate to invert the value of g. But it needs only 15 inputs. Moreover, the largest fan-in has been reduced to two. The cost of this circuit is lower than the cost of its two-level equivalent. The trade-off is an increased propagation delay because the circuit has three more levels of logic.

In this example the subfunction g is a function of variables x_1 and x_2. The subfunction is used as an input to the rest of the circuit that completes the realization of the required function f. Let h denote the function of this part of the circuit, which depends on only three

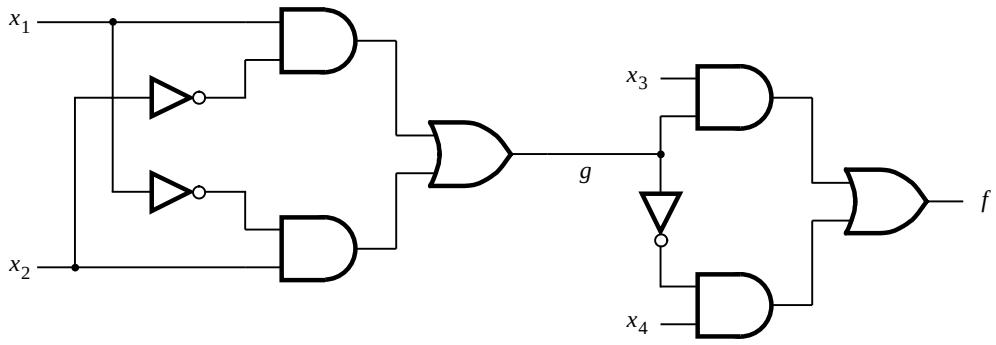


Figure 4.26 Logic circuit for Example 4.6.

inputs: g, x_3, and x_4. Then the decomposed realization of f can be expressed algebraically as

$$f(x_1, x_2, x_3, x_4) = h[g(x_1, x_2), x_3, x_4]$$

The structure of this decomposition can be described in block-diagram form as shown in Figure 4.27.

While not evident from our first example, functional decomposition can lead to great reductions in the complexity and cost of circuits. The reader will get a good indication of this benefit from the next example.

Example 4.7

Figure 4.28*a* defines a five-variable function f in the form of a Karnaugh map. In searching for a good decomposition for this function, it is necessary to first identify the variables that will be used as inputs to a subfunction. We can get a useful clue from the patterns of 1s in

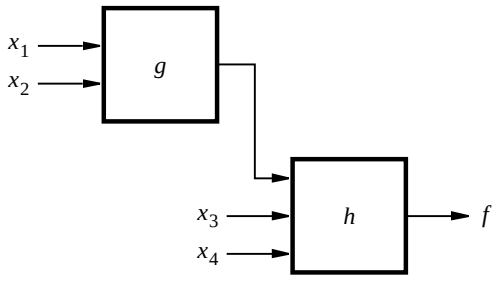
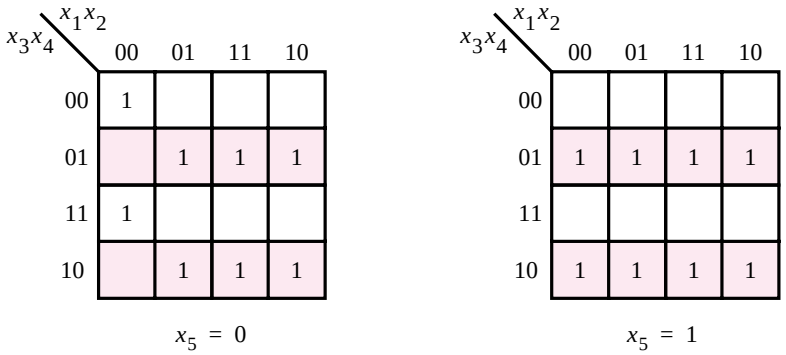
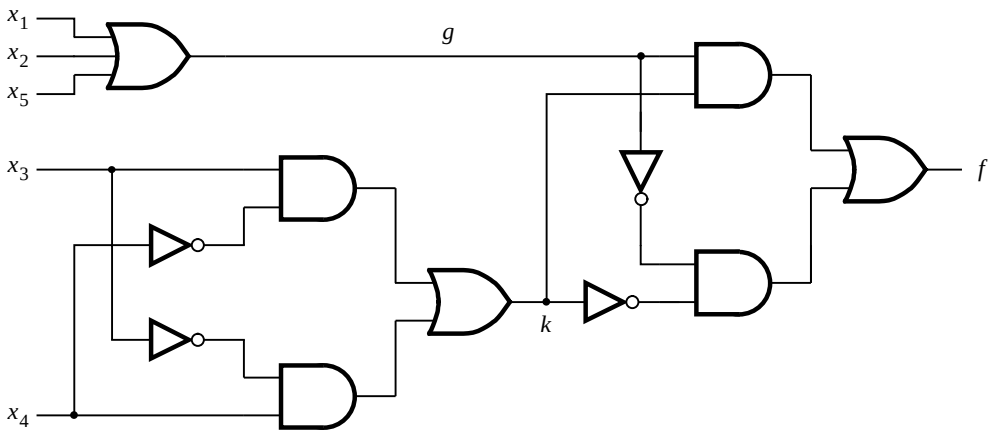


Figure 4.27 The structure of decomposition in Example 4.6.



(a) Karnaugh map for the function *f*



(b) Circuit obtained using decomposition

Figure 4.28 Decomposition for Example 4.7.

the map. Note that there are only two distinct patterns in the rows of the map. The second and fourth rows have one pattern, highlighted in blue, while the first and second rows have the other pattern. Once we specify which row each pattern is in, then the pattern itself depends only on the variables that define columns in each row, namely, x_1, x_2, and x_5. Let a subfunction $g(x_1, x_2, x_5)$ represent the pattern in rows 2 and 4. This subfunction is just

$$g = x_1 + x_2 + x_5$$

because the pattern has a 1 wherever any of these variables is equal to 1. To specify the location of rows where the pattern g occurs, we use the variables x_3 and x_4. The terms $\bar{x}_3x_4$ and $x_3\bar{x}_4$ identify the second and fourth rows, respectively. Thus the expression $(\bar{x}_3x_4 + x_3\bar{x}_4) \cdot g$ represents the part of f that is defined in rows 2 and 4.

Next, we have to find a realization for the pattern in rows 1 and 3. This pattern has a 1 only in the cell where $x_1 = x_2 = x_5 = 0$, which corresponds to the term $\bar{x}_1\bar{x}_2\bar{x}_5$. But we can make a useful observation that this term is just a complement of g. The location of rows 1 and 3 is identified by terms $\bar{x}_3\bar{x}_4$ and x_3x_4, respectively. Thus the expression $(\bar{x}_3\bar{x}_4 + x_3x_4) \cdot \bar{g}$ represents f in rows 1 and 3.

We can make one other useful observation. The expressions $(\bar{x}_3x_4 + x_3\bar{x}_4)$ and $(\bar{x}_3\bar{x}_4 + x_3x_4)$ are complements of each other, as shown in Example 4.6. Therefore, if we let $k(x_3, x_4) = \bar{x}_3x_4 + x_3\bar{x}_4$, the complete decomposition of f can be stated as

$$\begin{aligned} f(x_1, x_2, x_3, x_4, x_5) &= h[g(x_1, x_2, x_5), k(x_3, x_4)] \\ &= kg + \bar{k}\bar{g} \end{aligned}$$

where

$$\begin{aligned} g &= x_1 + x_2 + x_5 \\ k &= \bar{x}_3x_4 + x_3\bar{x}_4 \end{aligned}$$

The resulting circuit is given in Figure 4.28*b*. It requires a total of 11 gates and 19 inputs. The largest fan-in is three.

For comparison, a minimum-cost sum-of-products expression for f is

$$f = x_1\bar{x}_3x_4 + x_1x_3\bar{x}_4 + x_2\bar{x}_3x_4 + x_2x_3\bar{x}_4 + \bar{x}_3x_4x_5 + x_3\bar{x}_4x_5 + \bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4\bar{x}_5 + \bar{x}_1\bar{x}_2x_3x_4\bar{x}_5$$

The corresponding circuit requires a total of 14 gates (including the five NOT gates to complement the primary inputs) and 41 inputs. The fan-in for the output OR gate is eight. Obviously, functional decomposition results in a much simpler implementation of this function.

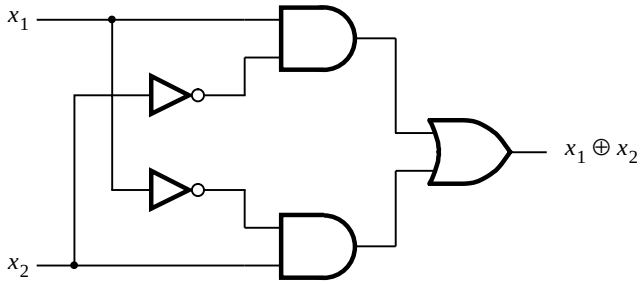
In both of the preceding examples, the decomposition is such that a decomposed subfunction depends on some primary input variables, whereas the remainder of the implementation depends on the rest of the variables. Such decompositions are called *disjoint decompositions* in the technical literature. It is possible to have a *non-disjoint decomposition*, where the variables of the subfunction are also used in realizing the remainder of the circuit. The following example illustrates this possibility.

Example 4.8

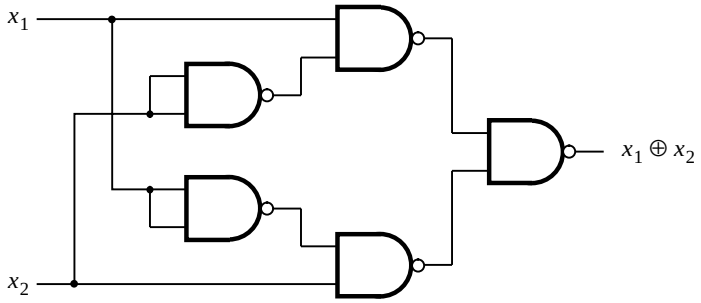
Exclusive-OR (XOR) is a very useful function. In section 3.9.1 we showed how it can be realized using a special circuit. It can also be realized using AND and OR gates as shown in Figure 4.29*a*. In section 4.6 we explained how any AND-OR circuit can be realized as a NAND-NAND circuit that has the same structure.

Let us now try to exploit functional decomposition to find a better implementation of XOR using only NAND gates. Let the symbol $\uparrow$ represent the NAND operation so that $x_1 \uparrow x_2 = \overline{x_1 \cdot x_2}$. A sum-of-products expression for the XOR function is

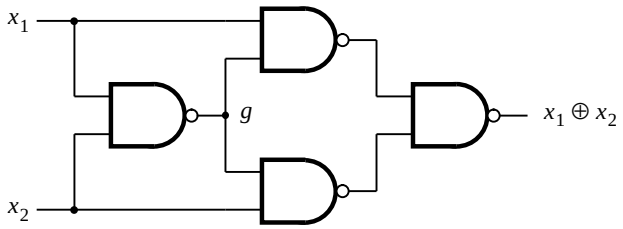
$$x_1 \oplus x_2 = x_1\bar{x}_2 + \bar{x}_1x_2$$



(a) Sum-of-products implementation



(b) NAND gate implementation



(c) Optimal NAND gate implementation

Figure 4.29 Implementation of XOR.

From the discussion in section 4.6, this expression can be written in terms of NAND operations as

$$x_1 \oplus x_2 = (x_1 \uparrow \overline{x}_2) \uparrow (\overline{x}_1 \uparrow x_2)$$

This expression requires five NAND gates, and it is implemented by the circuit in Figure 4.29*b*. Observe that an inverter is implemented using a two-input NAND gate by tying the two inputs together.

To find a decomposition, we can manipulate the term $(x_1 \uparrow \bar{x}_2)$ as follows:

$$(x_1 \uparrow \bar{x}_2) = \overline{(x_1\bar{x}_2)} = \overline{(x_1(\bar{x}_1 + \bar{x}_2))} = (x_1 \uparrow (\bar{x}_1 + \bar{x}_2))$$

We can perform a similar manipulation for $(\bar{x}_1 \uparrow x_2)$ to generate

$$x_1 \oplus x_2 = (x_1 \uparrow (\bar{x}_1 + \bar{x}_2)) \uparrow ((\bar{x}_1 + \bar{x}_2) \uparrow x_2)$$

DeMorgan's theorem states that $\bar{x}_1 + \bar{x}_2 = x_1 \uparrow x_2$; hence we can write

$$x_1 \oplus x_2 = (x_1 \uparrow (x_1 \uparrow x_2)) \uparrow ((x_1 \uparrow x_2) \uparrow x_2)$$

Now we have a decomposition

$$x_1 \oplus x_2 = (x_1 \uparrow g) \uparrow (g \uparrow x_2)$$

$$g = x_1 \uparrow x_2$$

The corresponding circuit, which requires only four NAND gates, is given in Figure 4.29*c*.

Practical Issues

Functional decomposition is a powerful technique for reducing the complexity of circuits. It can also be used to implement general logic functions in circuits that have built-in constraints. For example, in programmable logic devices (PLDs) that were introduced in Chapter 3 it is necessary to "fit" a desired logic circuit into logic blocks that are available on these devices. The available blocks are a target for decomposed subfunctions that may be used to realize larger functions.

A big problem in functional decomposition is finding the possible subfunctions. For functions of many variables, an enormous number of possibilities should be tried. This situation precludes attempts at finding optimal solutions. Instead, heuristic approaches that lead to acceptable solutions are used.

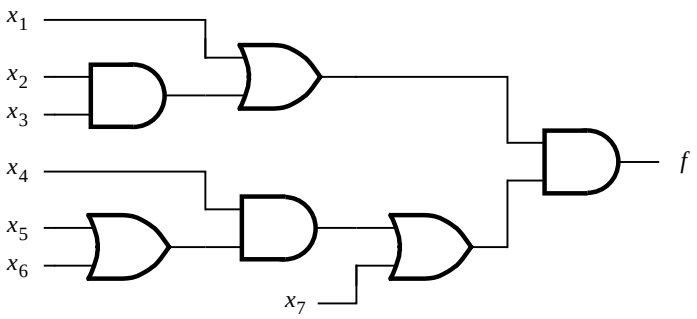
Full discussion of functional decomposition and factoring is beyond the scope of this book. An interested reader may consult other references [2–5]. Modern CAD tools use the concept of decomposition extensively.

4.7.3 MULTILEVEL NAND AND NOR CIRCUITS

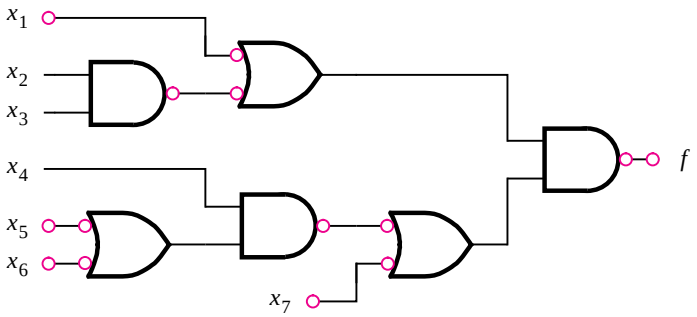
In section 4.6 we showed that two-level circuits consisting of AND and OR gates can be easily converted into circuits that can be realized with NAND and NOR gates, using the same gate arrangement. In particular, an AND-OR (sum-of products) circuit can be realized as a NAND-NAND circuit, while an OR-AND (product-of-sums) circuit becomes a NOR-NOR circuit. The same conversion approach can be used for multilevel circuits. We will illustrate this approach by an example.

Example 4.9

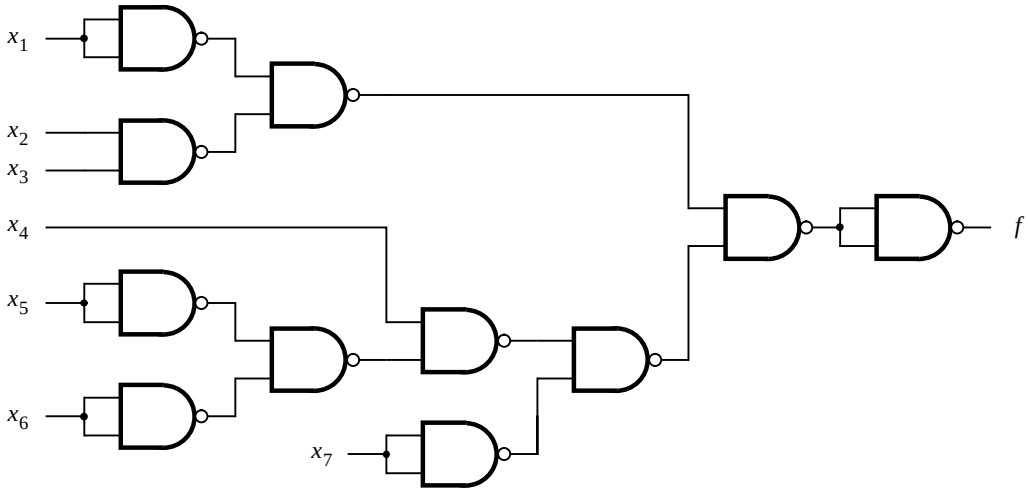
Figure 4.30*a* gives a four-level circuit consisting of AND and OR gates. Let us first derive a functionally equivalent circuit that comprises only NAND gates. Each AND gate is converted to a NAND by inverting its output. Each OR gate is converted to a NAND by



(a) Circuit with AND and OR gates



(b) Inversions needed to convert to NANDs

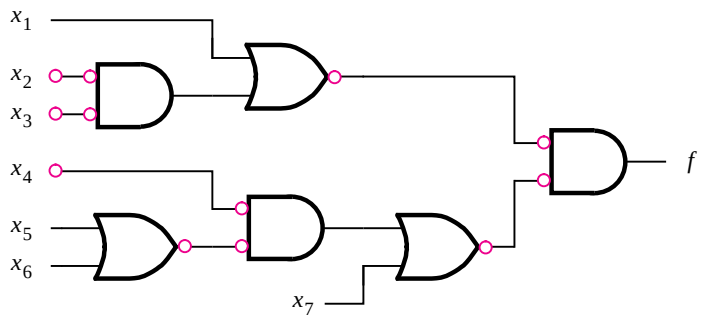


(c) NAND-gate circuit

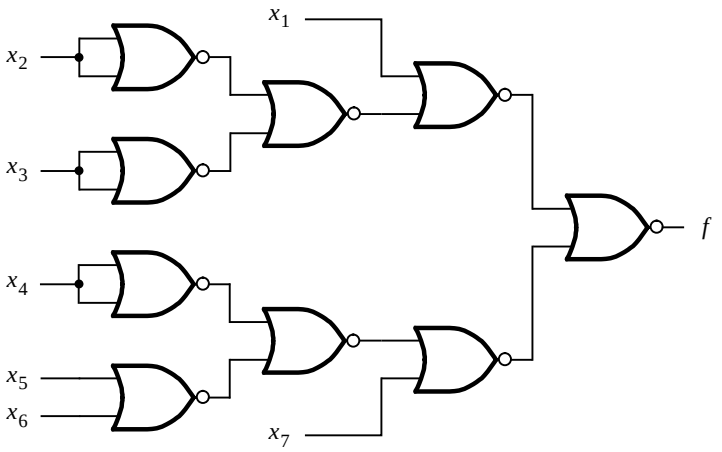
Figure 4.30 Conversion to a NAND-gate circuit.

inverting its inputs. This is just an application of DeMorgan's theorem, as illustrated in Figure 4.18*a*. Figure 4.30*b* shows the necessary inversions in blue. Note that an inversion is applied at both ends of a given wire. Now each gate becomes a NAND gate. This accounts for most of the inversions added to the original circuit. But, there are still four inversions that are not a part of any gate; therefore, they must be implemented separately. These inversions are at inputs x_1, x_5, and x_6 and at the output f. They can be implemented as two-input NAND gates, where the inputs are tied together. The resulting circuit is shown in Figure 4.30*c*.

A similar approach can be used to convert the circuit in Figure 4.30*a* into a circuit that comprises only NOR gates. An OR gate is converted to a NOR gate by inverting its output. An AND becomes a NOR if its inputs are inverted, as indicated in Figure 4.18*b*. Using this approach, the inversions needed for our sample circuit are shown in blue in Figure 4.31*a*.



(a) Inversions needed to convert to NORs



(b) NOR-gate circuit

Figure 4.31 Conversion to a NOR-gate circuit.

Then each gate becomes a NOR gate. The three inversions at inputs x_2, x_3, and x_4 can be realized as two-input NOR gates, where the inputs are tied together. The resulting circuit is presented in Figure 4.31*b*.

It is evident that the basic topology of a circuit does not change substantially when converting from AND and OR gates to either NAND or NOR gates. However, it may be necessary to insert additional gates to serve as NOT gates that implement inversions not absorbed as a part of other gates in the circuit.

4.8 ANALYSIS OF MULTILEVEL CIRCUITS

The preceding section showed that it may be advantageous to implement logic functions using multilevel circuits. It also presented the most commonly used approaches for synthesizing functions in this way. In this section we will consider the task of analyzing an existing circuit to determine the function that it implements.

For two-level circuits the analysis process is simple. If a circuit has an AND-OR (NAND-NAND) structure, then its output function can be written in the SOP form by inspection. Similarly, it is easy to derive a POS expression for an OR-AND (NOR-NOR) circuit. The analysis task is more complicated for multilevel circuits because it is difficult to write an expression for the function by inspection. We have to derive the desired expression by tracing the circuit and determining its functionality. The tracing can be done either starting from the input side and working towards the output, or by starting at the output side and working back towards the inputs. At intermediate points in the circuit, it is necessary to evaluate the subfunctions realized by the logic gates.

Example 4.10 Figure 4.32 replicates the circuit from Figure 4.30*a*. To determine the function f implemented by this circuit, we can consider the functionality at internal points that are the outputs

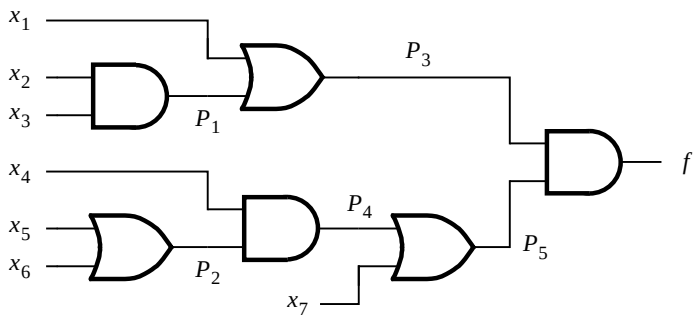


Figure 4.32 Circuit for Example 4.10.

of various gates. These points are labeled P_1 to P_5 in the figure. The functions realized at these points are

$$P_1 = x_2x_3$$
$$P_2 = x_5 + x_6$$
$$P_3 = x_1 + P_1 = x_1 + x_2x_3$$
$$P_4 = x_4P_2 = x_4(x_5 + x_6)$$
$$P_5 = P_4 + x_7 = x_4(x_5 + x_6) + x_7$$

Then f can be evaluated as

$$\begin{aligned} f &= P_3P_5 \\ &= (x_1 + x_2x_3)(x_4(x_5 + x_6) + x_7) \end{aligned}$$

Applying the distributive property to eliminate the parentheses gives

$$f = x_1x_4x_5 + x_1x_4x_6 + x_1x_7 + x_2x_3x_4x_5 + x_2x_3x_4x_6 + x_2x_3x_7$$

Note that the expression represents a circuit comprising six AND gates, one OR gate, and 25 inputs. The cost of this two-level circuit is higher than the cost of the circuit in Figure 4.32, but the circuit has lower propagation delay.

Example 4.11

In Example 4.7 we derived the circuit in Figure 4.28*b*. In addition to AND gates and OR gates, the circuit has some NOT gates. It is reproduced in Figure 4.33, and the internal points are labeled from P_1 to P_{10} as shown. The following subfunctions occur

$$P_1 = x_1 + x_2 + x_5$$
$$P_2 = \overline{x}_4$$
$$P_3 = \overline{x}_3$$
$$P_4 = x_3P_2$$
$$P_5 = x_4P_3$$
$$P_6 = P_4 + P_5$$
$$P_7 = \overline{P}_1$$
$$P_8 = \overline{P}_6$$
$$P_9 = P_1P_6$$
$$P_{10} = P_7P_8$$

We can derive f by tracing the circuit from the output towards the inputs as follows

$$\begin{aligned} f &= P_9 + P_{10} \\ &= P_1P_6 + P_7P_8 \end{aligned}$$

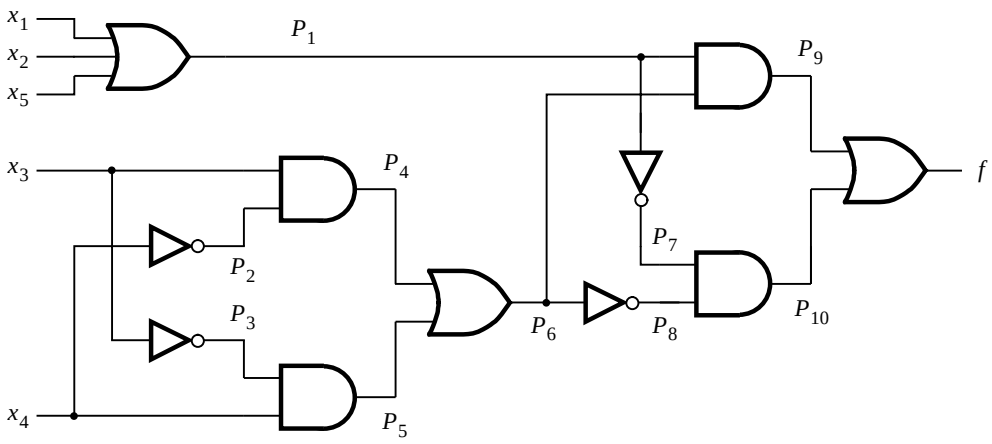


Figure 4.33 Circuit for Example 4.11.

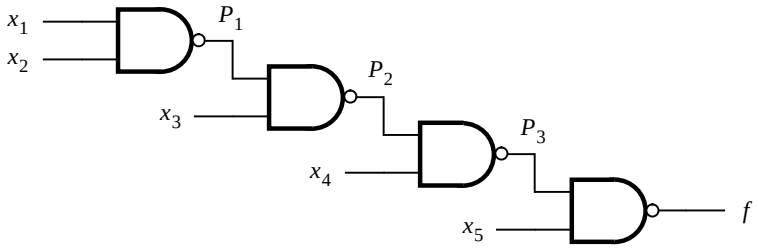
$$= (x_1 + x_2 + x_5)(P_4 + P_5) + \overline{P}_1\overline{P}_6$$
$$= (x_1 + x_2 + x_5)(x_3P_2 + x_4P_3) + \overline{x}_1\overline{x}_2\overline{x}_5\overline{P}_4\overline{P}_5$$
$$= (x_1 + x_2 + x_5)(x_3\overline{x}_4 + x_4\overline{x}_3) + \overline{x}_1\overline{x}_2\overline{x}_5(\overline{x}_3 + \overline{P}_2)(\overline{x}_4 + \overline{P}_3)$$
$$= (x_1 + x_2 + x_5)(x_3\overline{x}_4 + \overline{x}_3x_4) + \overline{x}_1\overline{x}_2\overline{x}_5(\overline{x}_3 + x_4)(\overline{x}_4 + x_3)$$
$$= x_1x_3\overline{x}_4 + x_1\overline{x}_3x_4 + x_2x_3\overline{x}_4 + x_2\overline{x}_3x_4 + x_5x_3\overline{x}_4 + x_5\overline{x}_3x_4 +$$
$$\overline{x}_1\overline{x}_2\overline{x}_5\overline{x}_3\overline{x}_4 + \overline{x}_1\overline{x}_2\overline{x}_5x_4x_3$$

This is the same expression as stated in Example 4.7.

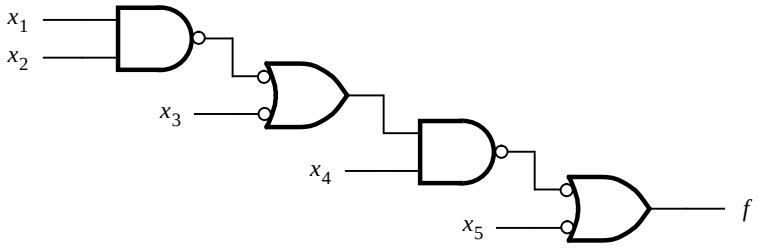
Example 4.12 Circuits based on NAND and NOR gates are slightly more difficult to analyze because each gate involves an inversion. Figure 4.34*a* depicts a simple NAND-gate circuit that illustrates the effect of inversions. We can convert this circuit into a circuit with AND and OR gates using the reverse of the approach described in Example 4.9. Bubbles that denote inversions can be moved, according to DeMorgan's theorem, as indicated in Figure 4.34*b*. Then the circuit can be converted into the circuit in part (*c*) of the figure, which consists of AND and OR gates. Observe that in the converted circuit, the inputs x_3 and x_5 are complemented. From this circuit the function f is determined as

$$f = (x_1x_2 + \overline{x}_3)x_4 + \overline{x}_5$$
$$= x_1x_2x_4 + \overline{x}_3x_4 + \overline{x}_5$$

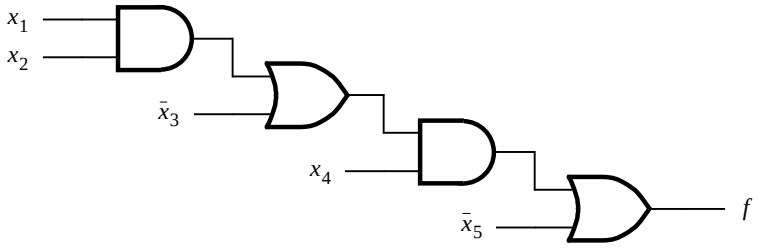
It is not necessary to convert a NAND circuit into a circuit with AND and OR gates to determine its functionality. We can use the approach from Examples 4.10 and 4.11 to



(a) NAND-gate circuit



(b) Moving bubbles to convert to ANDs and ORs



(c) Circuit with AND and OR gates

Figure 4.34 Circuit for Example 4.12.

derive f as follows. Let P_1, P_2, and P_3 label the internal points as shown in Figure 4.34*a*. Then

$$P_1 = \overline{x_1 x_2}$$
$$P_2 = \overline{P_1 x_3}$$
$$P_3 = \overline{P_2 x_4}$$

$$
\begin{aligned}
f &= \overline{P_3 x_5} = \overline{P}_3 + \overline{x}_5 \\
&= \overline{\overline{P_2 x_4}} + \overline{x}_5 = P_2 x_4 + \overline{x}_5 \\
&= \overline{P_1 x_3} x_4 + \overline{x}_5 = (\overline{P}_1 + \overline{x}_3) x_4 + \overline{x}_5 \\
&= (\overline{\overline{x_1 x_2}} + \overline{x}_3) x_4 + \overline{x}_5 \\
&= (x_1 x_2 + \overline{x}_3) x_4 + \overline{x}_5 \\
&= x_1 x_2 x_4 + \overline{x}_3 x_4 + \overline{x}_5
\end{aligned}
$$

Example 4.13 The circuit in Figure 4.35 consists of NAND and NOR gates. It can be analyzed as follows.

$$
\begin{aligned}
P_1 &= \overline{x_2 x_3} \\
P_2 &= \overline{x_1 P_1} = \overline{x}_1 + \overline{P}_1 \\
P_3 &= \overline{x_3 x_4} = \overline{x}_3 + \overline{x}_4 \\
P_4 &= \overline{P_2 + P_3} \\
f &= \overline{P_4 + x_5} = \overline{P}_4 \overline{x}_5 \\
&= \overline{\overline{P_2 + P_3}} \cdot \overline{x}_5 \\
&= (P_2 + P_3)\overline{x}_5 \\
&= (\overline{x}_1 + \overline{P}_1 + \overline{x}_3 + \overline{x}_4)\overline{x}_5 \\
&= (\overline{x}_1 + x_2 x_3 + \overline{x}_3 + \overline{x}_4)\overline{x}_5 \\
&= (\overline{x}_1 + x_2 + \overline{x}_3 + \overline{x}_4)\overline{x}_5 \\
&= \overline{x}_1 \overline{x}_5 + x_2 \overline{x}_5 + \overline{x}_3 \overline{x}_5 + \overline{x}_4 \overline{x}_5
\end{aligned}
$$

Note that in deriving the second to the last line, we used property 16*a* in section 2.5 to simplify $x_2 x_3 + \overline{x}_3$ into $x_2 + \overline{x}_3$.

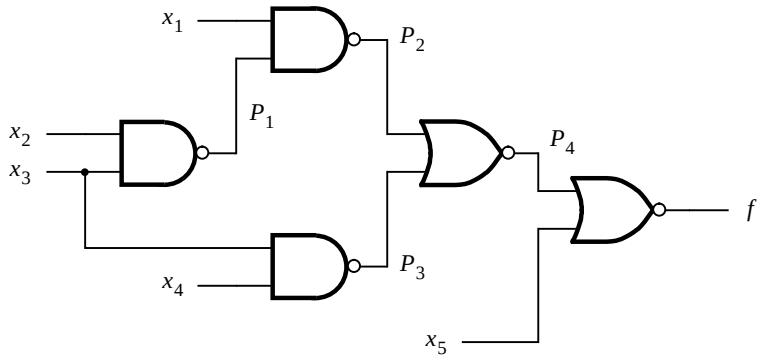


Figure 4.35 Circuit for Example 4.13.

Analysis of circuits is much simpler than synthesis. With a little practice one can develop an ability to easily analyze even fairly complex circuits.

We have now covered a considerable amount of material on synthesis and analysis of logic functions. We have used the Karnaugh map as a vehicle for illustrating the concepts involved in finding optimal implementations of logic functions. We have also shown that logic functions can be realized in a variety of forms, both with two levels of logic and with multiple levels. In a modern design environment, logic circuits are synthesized using CAD tools, rather than by hand. The concepts that we have discussed in this chapter are quite general; they are representative of the strategies implemented in CAD algorithms. As we have said before, the Karnaugh map scheme for representing logic functions is not appropriate for use in CAD tools. In the next section we discuss an alternative representation of logic functions, which is suitable for use in CAD algorithms.

4.9 CUBICAL REPRESENTATION

The Karnaugh map is an excellent vehicle for illustrating concepts, and it is even useful for manual design if the functions have only a few variables. To deal with larger functions it is necessary to have techniques that are algebraic, rather than graphical, which can be applied to functions of any number of variables.

Many algebraic optimization techniques have been developed. As early as the 1950s, a tabular approach proposed by Willard Quine [6] and Edward McCluskey [7] became popular under the name Quine-McCluskey method. Almost all textbooks on logic design discuss this method at length [8–18]. We will not do so because there exist more attractive alternatives that can be incorporated into CAD tools.

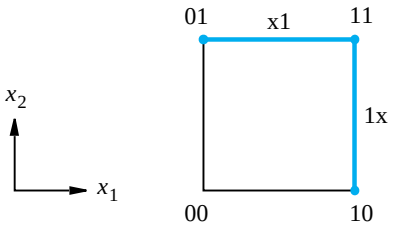
We will not pursue algebraic optimization techniques in great detail, but we will attempt to provide the reader with an appreciation of the tasks involved. This helps in gaining an understanding of what the CAD tools can do and what results can be expected from them. The approach that we will present makes use of a cubical representation of logic functions.

4.9.1 CUBES AND HYPERCUBES

So far in this book, we have encountered four different forms for representing logic functions: truth tables, algebraic expressions, Venn diagrams, and Karnaugh maps. Another possibility is to map a function of n variables onto an n-dimensional cube.

Two-Dimensional Cube

A two-dimensional cube is shown in Figure 4.36. The four corners in the cube are called *vertices*, which correspond to the four rows of a truth table. Each vertex is identified by two coordinates. The horizontal coordinate is assumed to correspond to variable x_1, and vertical coordinate to x_2. Thus vertex 00 is the bottom-left corner, which corresponds to row 0 in the truth table. Vertex 01 is the top-left corner, where $x_1 = 0$ and $x_2 = 1$, which corresponds to row 1 in the truth table, and so on for the other two vertices.



x_1	x_2	f
0	0	0
0	1	1
1	0	1
1	1	1

Figure 4.36 Representation of $f(x_1, x_2) = \sum m(1, 2, 3)$.

We will map a function onto the cube by indicating with blue circles those vertices for which $f = 1$. In Figure 4.36 $f = 1$ for vertices 01, 10, and 11. We can express the function as a set of vertices, using the notation $f = \{01, 10, 11\}$. The function f is also shown in the form of a truth table in the figure.

An edge joins two vertices for which the labels differ in the value of only one variable. Therefore, if two vertices for which $f = 1$ are joined by an edge, then this edge represents that portion of the function just as well as the two individual vertices. For example, $f = 1$ for vertices 10 and 11. They are joined by the edge that is labeled 1x. It is customary to use the letter x to denote the fact that the corresponding variable can be either 0 or 1. Hence 1x means that $x_1 = 1$, while x_2 can be either 0 or 1. Similarly, vertices 01 and 11 are joined by the edge labeled x1, indicating that x_1 can be either 0 or 1, but $x_2 = 1$. The reader must not confuse the use of the letter x for this purpose, in contrast to the subscripted use where x_1 and x_2 refer to the variables.

Two vertices being represented by a single edge is the embodiment of the combining property 14*a* from section 2.5. The edge 1x is the logical sum of vertices 10 and 11. It essentially defines the term x_1, which is the sum of minterms $x_1\bar{x}_2$ and x_1x_2. The property 14*a* indicates that

$$x_1\bar{x}_2 + x_1x_2 = x_1$$

Therefore, finding edges for which $f = 1$ is equivalent to applying the combining property. Of course, this is also analogous to finding pairs of adjacent cells in a Karnaugh map for which $f = 1$.

The edges 1x and x1 define fully the function in Figure 4.36; hence we can represent the function as $f = \{1x, x1\}$. This corresponds to the logic expression

$$f = x_1 + x_2$$

which is also obvious from the truth table in the figure.

Three-Dimensional Cube

Figure 4.37 illustrates a three-dimensional cube. The x_1, x_2, and x_3 coordinates are as shown on the left. Each vertex is identified by a specific valuation of the three variables. The function f mapped onto the cube is the function from Figure 4.1, which was used in Figure 4.5*b*. There are five vertices for which $f = 1$, namely, 000, 010, 100, 101, and 110. These vertices are joined by the five edges shown in blue, namely, x00, 0x0, x10, 1x0,

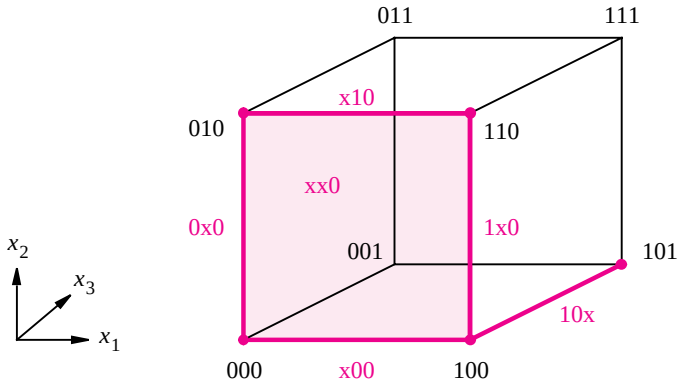


Figure 4.37 Representation of $f(x_1, x_2, x_3) = \sum m(0, 2, 4, 5, 6)$.

and 10x. Because the vertices 000, 010, 100, and 110 include all valuations of x_1 and x_2, when x_3 is 0, they can be specified by the term xx0. This term means that $f = 1$ if $x_3 = 0$, regardless of the values of x_1 and x_2. Notice that xx0 represents the front side of the cube, which is shaded in blue.

From the preceding discussion it is evident that the function f can be represented in several ways. Some of the possibilities are

$$\begin{aligned} f &= \{000, 010, 100, 101, 110\} \\ &= \{0x0, 1x0, 101\} \\ &= \{x00, x10, 101\} \\ &= \{x00, x10, 10x\} \\ &= \{xx0, 10x\} \end{aligned}$$

In a physical realization each of the above terms is a product term implemented by an AND gate. Obviously, the least-expensive circuit is obtained if $f = \{xx0, 10x\}$, which is equivalent to the logic expression

$$f = \bar{x}_3 + x_1\bar{x}_2$$

This is the expression that we derived using the Karnaugh map in Figure 4.5*b*.

Four-Dimensional Cube

Graphical images of two- and three-dimensional cubes are easy to draw. A four-dimensional cube is more difficult. It consists of 2 three-dimensional cubes with their corners connected. The simplest way to visualize a four-dimensional cube is to have one cube placed inside the other cube, as depicted in Figure 4.38. We have assumed that the x_1, x_2, and x_3 coordinates are the same as in Figure 4.37, while $x_4 = 0$ defines the outer cube and $x_4 = 1$ defines the inner cube. Figure 4.38 indicates how the function f_3 of Figure 4.7 is mapped onto the four-dimensional cube. To avoid cluttering the figure with too many labels, we have labeled only those vertices for which $f_3 = 1$. Again, all edges that connect these vertices are highlighted in blue.

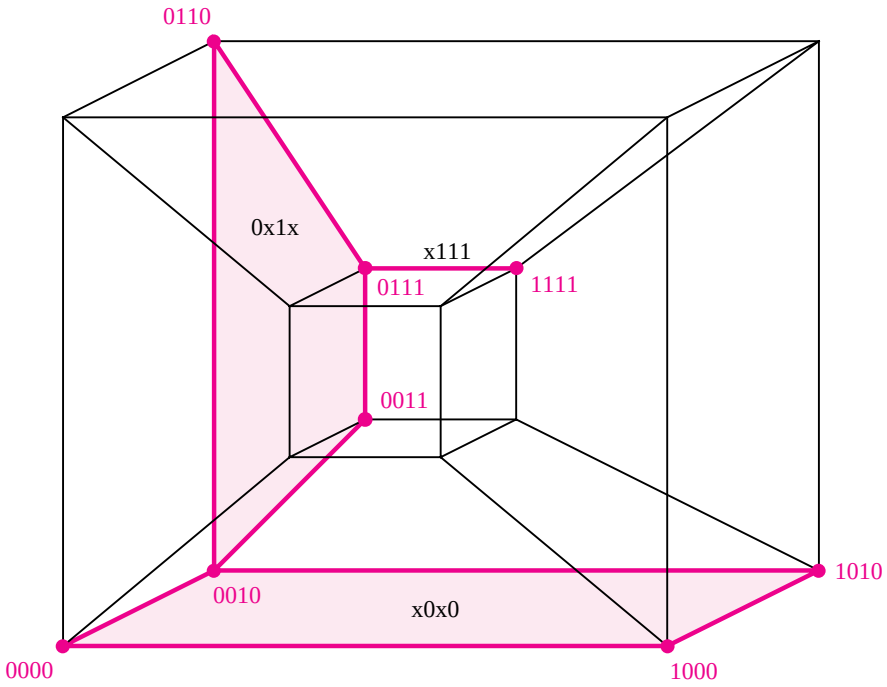


Figure 4.38 Representation of function f_3 from Figure 4.7.

There are two groups of four adjacent vertices for which $f_3 = 1$ that can be represented as planes. The group comprising 0000, 0010, 1000, and 1010 is represented by x0x0. The group 0010, 0011, 0110, and 0111 is represented by 0x1x. These planes are shaded in the figure. The function f_3 can be represented in several ways, for example

$$\begin{aligned} f_3 &= \{0000, 0010, 0011, 0110, 0111, 1000, 1010, 1111\} \\ &= \{00x0, 10x0, 0x10, 0x11, x111\} \\ &= \{x0x0, 0x1x, x111\} \end{aligned}$$

Since each x indicates that the corresponding variable can be ignored, because it can be either 0 or 1, the simplest circuit is obtained if $f = \{x0x0, 0x1x, x111\}$, which is equivalent to the expression

$$f_3 = \overline{x}_2\overline{x}_4 + \overline{x}_1x_3 + x_2x_3x_4$$

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 4.37 and 4.38, it is apparent that the largest possible k-*cubes* that exist for a given function are equivalent to its prime implicants. Next, we will describe a minimization technique that uses the cubical representation of functions.

4.10 MINIMIZATION USING CUBICAL REPRESENTATION

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 outline a relatively simple algorithm, which illustrates the main concepts and indicates some of the problems that arise. 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.

4.10.1 GENERATION OF PRIME IMPLICANTS

As mentioned in section 4.9, 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.

In section 4.2.2 we presented a strategy for finding the minimum-cost sum-of-products form of a logic function. Assume that the initial specification of a function f is given in terms of implicants that are not necessarily either minterms or prime implicants. Then it is necessary to define an operation that will generate other implicants that are not given explicitly in the initial specification, but which will eventually lead to the prime implicants

of f. One such possibility is known as the $*$-*product* operation, which is usually pronounced the "star-product" operation. We will refer to it simply as the $*$-*operation*.

$*$-Operation

The $*$-operation provides a simple way of deriving a new cube by combining two cubes that differ in the value of only one variable. Let $A = A_1A_2 \ldots A_n$ and $B = B_1B_2 \ldots B_n$ be two cubes that are implicants of an n-variable function. Thus each coordinate A_i and B_i is specified as having the value 0, 1, or x. There are two distinct steps in the $*$-operation. First, the $*$-operation is evaluated for each pair A_i and B_i, in coordinates $i = 1, 2, \ldots, n$, according to the table in Figure 4.39. Then based on the results of using the table, a set of rules is applied to determine the overall result of the $*$-operation. The table in Figure 4.39 defines the coordinate $*$-operation, $A_i * B_i$. It specifies the result of $A_i * B_i$ for each possible combination of values of A_i and B_i. This result is the intersection (i.e., the common part) of A and B in this coordinate. Note that when A_i and B_i have the opposite values (0 and 1, or vice versa), the result of the coordinate $*$-operation is indicated by the symbol ø. We say that the intersection of A_i and B_i is empty. Using the table, the complete $*$-operation for A and B is defined as follows:

$$C = A * B, \text{ such that}$$

1. $C = ø$ *if* $A_i * B_i = ø$ for more than one i.
2. Otherwise, $C_i = A_i * B_i$ when $A_i * B_i \neq ø$, and $C_i = \text{x}$ for the coordinate where $A_i * B_i = ø$.

For example, let $A = \{0\text{x}0\}$ and $B = \{111\}$. Then $A_1 * B_1 = 0 * 1 = ø$, $A_2 * B_2 = \text{x} * 1 = 1$, and $A_3 * B_3 = 0 * 1 = ø$. Because the result is ø in two coordinates, it follows from condition 1 that $A * B = ø$. In other words, these two cubes cannot be combined into another cube, because they differ in two coordinates.

As another example, consider $A = \{11\text{x}\}$ and $B = \{10\text{x}\}$. In this case $A_1 * B_1 = 1 * 1 = 1$, $A_2 * B_2 = 1 * 0 = ø$, and $A_3 * B_3 = \text{x} * \text{x} = \text{x}$. According to condition 2 above, $C_1 = 1$, $C_2 = \text{x}$, and $C_3 = \text{x}$, which gives $C = A * B = \{1\text{xx}\}$. A larger 2-cube is created from two 1-cubes that differ in one coordinate only.

The result of the $*$-operation may be a smaller cube than the two cubes involved in the operation. Consider $A = \{1\text{x}1\}$ and $B = \{11\text{x}\}$. Then $C = A * B = \{111\}$. Notice that C is included in both A and B, which means that this cube will not be useful in searching for prime implicants. Therefore, it should be discarded by the minimization algorithm.

A_i \ B_i	0	1	x
0	0	ø	0
1	ø	1	1
x	0	1	x

$A_i * B_i$

Figure 4.39 The coordinate $*$-operation.

As a final example, consider $A = \{x10\}$ and $B = \{0x1\}$. Then $C = A * B = \{01x\}$. All three of these cubes are the same size, but C is not included in either A or B. Hence C has to be considered in the search for prime implicants. The reader may find it helpful to draw a Karnaugh map to see how cube C is related to cubes A and B.

Using the $*$-Operation to Find Prime Implicants

The essence of the $*$-operation is to find new cubes from pairs of existing cubes. In particular, it is of interest to find new cubes that are not included in the existing cubes. A procedure for finding the prime implicants may be organized as follows.

Suppose that a function f is specified by means of a set of implicants that are represented as cubes. Let this set be denoted as the cover C^k of f. Let c^i and c^j be any two cubes in C^k. Then apply the $*$-operation to all pairs of cubes in C^k; let G^{k+1} be the set of newly generated cubes. Hence

$$G^{k+1} = c^i * c^j \quad \text{for all } c^i, c^j \epsilon\, C^k$$

Now a new cover for f may be formed by using the cubes in C^k and G^{k+1}. Some of these cubes may be redundant because they are included in other cubes; they should be removed. Let the new cover be

$$C^{k+1} = C^k \cup G^{k+1} - \text{ redundant cubes}$$

where $\cup$ denotes the logical union of two sets, and the minus sign $(-)$ denotes the removal of elements of a set. If $C^{k+1} \neq C^k$, then a new cover C^{k+2} is generated using the same process. If $C^{k+1} = C^k$, then the cubes in the cover are the prime implicants of f. For an n-variable function, it is necessary to repeat the step at most n times.

Redundant cubes that have to be removed are identified through pairwise comparison of cubes. Cube $A = A_1A_2 \ldots A_n$ should be removed if it is included in some cube $B = B_1B_2 \ldots B_n$, which is the case if $A_i = B_i$ or $B_i = x$ for every coordinate i.

Example 4.14

Consider the function $f(x_1, x_2, x_3)$ of Figure 4.9. Assume that f is initially specified as a set of vertices that correspond to the minterms, m_0, m_1, m_2, m_3, and m_7. Hence let the initial cover be $C^0 = \{000, 001, 010, 011, 111\}$. Using the $*$-operation to generate a new set of cubes, we obtain $G^1 = \{00x, 0x0, 0x1, 01x, x11\}$. Then $C^1 = C^0 \cup G^1$ – redundant cubes. Observe that each cube in C^0 is included in one of the cubes in G^1; therefore, all cubes in C^0 are redundant. Thus $C^1 = G^1$.

The next step is to apply the $*$-operation to the cubes in C^1, which yields $G^2 = \{000, 001, 0xx, 0x1, 010, 01x, 011\}$. Note that all of these cubes are included in the cube 0xx; therefore, all but 0xx are redundant. Now it is easy to see that

$$\begin{aligned} C^2 &= C^1 \cup G^2 - \text{redundant terms} \\ &= \{x11, 0xx\} \end{aligned}$$

since all cubes of C^1, except x11, are redundant because they are covered by 0xx.

Applying the $*$-operation to C^2 yields $G^3 = \{011\}$ and

$$C^3 = C^2 \cup G^3 - \text{redundant terms}$$
$$= \{\text{x11, 0xx}\}$$

Since $C^3 = C^2$, the conclusion is that the prime implicants of f are the cubes {x11, 0xx}, which represent the product terms x_2x_3 and $\bar{x}_1$. This is the same set of prime implicants that we derived using a Karnaugh map in Figure 4.9.

Example 4.15 As another example, consider the four-variable function of Figure 4.10. Assume that this function is initially specified as the cover C^0 = {0101, 1101, 1110, 011x, x01x}. Then successive applications of the $*$-operation and removing the redundant terms gives

$$C^1 = \{\text{x01x, x101, 01x1, x110, 1x10, 0x1x}\}$$
$$C^2 = \{\text{x01x, x101, 01x1, 0x1x, xx10}\}$$
$$C^3 = C^2$$

Therefore, the prime implicants are $\bar{x}_2x_3$, $x_2\bar{x}_3x_4$, $\bar{x}_1x_2x_4$, $\bar{x}_1x_3$, and $x_3\bar{x}_4$.

4.10.2 DETERMINATION OF ESSENTIAL PRIME IMPLICANTS

From a cover that consists of all prime implicants, it is necessary to extract a minimal cover. As we saw in section 4.2.2, all *essential* prime implicants must be included in the minimal cover. To find the essential prime implicants, it is useful to define an operation that determines a part of a cube (implicant) that is *not* covered by another cube. One such operation is called the *#-operation* (pronounced the "sharp operation"), which is defined as follows.

#-Operation

Again, let $A = A_1A_2 \ldots A_n$ and $B = B_1B_2 \ldots B_n$ be two cubes (implicants) of an n-variable function. The sharp operation $A\#B$ leaves as a result "that part of A that is not covered by B." Similar to the $*$-operation, the #-operation has two steps: $A_i\#B_i$ is evaluated for each coordinate i, and then a set of rules is applied to determine the overall result. The sharp operation for each coordinate is defined in Figure 4.40. After this operation is performed for all pairs (A_i, B_i), the complete #-operation is defined as follows:

$$C = A\#B, \text{ such that}$$

1. $C = A$ if $A_i\#B_i = \emptyset$ for some i.
2. $C = \emptyset$ if $A_i\#B_i = \varepsilon$ for all i.
3. Otherwise, $C = \bigcup_i (A_1, A_2, \ldots, \overline{B}_i, \ldots, A_n)$, where the union is for all i for which $A_i = \text{x}$ and $B_i \neq \text{x}$.

The first condition corresponds to the case where cubes A and B do not intersect at all; namely, A and B differ in the value of at least one variable, which means that no part of A is covered by B. For example, let A = 0x1 and B = 11x. The coordinate #-products are $A_1\#B_1 = \emptyset$, $A_2\#B_2 = 0$, and $A_3\#B_3 = \varepsilon$. Then from rule 1 it follows that 0x1 # 11x = 0x1. The second condition reflects the case where A is fully covered by B. For example, 0x1

A_i \ B_i	0	1	x
0	ε	ø	ε
1	ø	ε	ε
x	1	0	ε

$A_i \# B_i$

Figure 4.40 The coordinate #-operation.

0xx = ø. The third condition is for the case where only a part of A is covered by B. In this case the #-operation generates one or more cubes. Specifically, it generates one cube for each coordinate i that is x in A_i, but is not x in B_i. Each cube generated is identical to A, except that A_i is replaced by $\overline{B}_i$. For example, 0xx # 01x = 00x, and 0xx # 010 = {00x, 0x1}.

We will now show how the #-operation can be used to find the essential prime implicants. Let P be the set of all prime implicants of a given function f. Let p^i denote one prime implicant in the set P. Also, let DC denote the don't-care vertices for f. Then p^i is an essential prime implicant if and only if

$$p^i \# (P - p^i) \# DC \neq \text{ø}$$

This means that p^i is essential if there exists at least one vertex for which $f = 1$ that is covered by p^i, but not by any other prime implicant. The #-operation is also performed with the set of don't-care cubes because vertices in p^i that correspond to don't-care conditions are not essential to cover. The meaning of $p^i \# (P - p^i)$ is that the #-operation is applied successively to each prime implicant in P. For example, consider $P = \{p^1, p^2, p^3, p^4\}$ and $DC = \{d^1, d^2\}$. To check whether p^3 is essential, we evaluate

$$((((p^3 \# p^1) \# p^2) \# p^4) \# d^1) \# d^2$$

If the result of this expression is not ø, then p^3 is essential.

Example 4.16

In Example 4.14 we determined that the cubes x11 and 0xx are the prime implicants of the function f in Figure 4.9. We can discover whether each of these prime implicants is essential as follows

$$\text{x11 \# 0xx} = 111 \neq \text{ø}$$

$$\text{0xx \# x11} = \{00\text{x}, 0\text{x}0\} \neq \text{ø}$$

The cube x11 is essential because it is the only prime implicant that covers the vertex 111, for which $f = 1$. The prime implicant 0xx is essential because it is the only one that covers the vertices 000, 001, and 010. This can be seen in the Karnaugh map in Figure 4.9.

Example 4.17

In Example 4.15 we found that the prime implicants of the function in Figure 4.10 are P = {x01x, x101, 01x1, 0x1x, xx10}. Because this function has no don't cares, we compute

$$x01x \# (P - x01x) = 1011 \neq \emptyset$$

This is computed in the following steps: x01x # x101 = x01x, then x01x # 01x1 = x01x, then x01x # 0x1x = 101x, and finally 101x # xx10 = 1011. Similarly, we obtain

$$x101 \# (P - x101) = 1101 \neq \emptyset$$
$$01x1 \# (P - 01x1) = \emptyset$$
$$0x1x \# (P - 0x1x) = \emptyset$$
$$xx10 \# (P - xx10) = 1110 \neq \emptyset$$

Therefore, the essential prime implicants are x01x, x101, and xx10 because they are the only ones that cover the vertices 1011, 1101, and 1110, respectively. This is obvious from the Karnaugh map in Figure 4.10.

When checking whether a cube A is essential, the #-operation with one of the cubes in $P - A$ may generate multiple cubes. If so, then each of these cubes has to be checked using the #-operation with all of the remaining cubes in $P - A$.

4.10.3 COMPLETE PROCEDURE FOR FINDING A MINIMAL COVER

Having introduced the $*$- and #-operations, we can now outline a complete procedure for finding a minimal cover for any n-variable function. Assume that the function f is specified in terms of vertices for which $f = 1$; these vertices are often referred to as the *ON-set* of the function. Also, assume that the don't-care conditions are specified as a *DC-set*. Then the initial cover for f is a union of the ON and DC sets.

Prime implicants of f can be generated using the $*$-operation, as explained in section 4.10.1. Then the #-operation can be used to find the essential prime implicants as presented in section 4.10.2. If the essential prime implicants cover the entire ON-set, then they form the minimum-cost cover for f. Otherwise, it is necessary to include other prime implicants until all vertices in the ON-set are covered.

A nonessential prime implicant p^i should be deleted if there exists a less-expensive prime implicant p^j that covers all vertices of the ON-set that are covered by p^i. If the remaining nonessential prime implicants have the same cost, then a possible heuristic approach is to arbitrarily select one of them, include it in the cover, and determine the rest of the cover. Then an alternative cover is generated by excluding this prime implicant, and the lower-cost cover is chosen for implementation. We already used this approach, which is often referred to as the *branching* heuristic, in section 4.2.2.

The preceding discussion can be summarized in the form of the following minimization procedure:

1. Let $C^0 = ON \cup DC$ be the initial cover of function f and its don't-care conditions.
2. Find all prime implicants of C^0 using the $*$-operation; let P be this set of prime implicants.
3. Find the essential prime implicants using the #-operation. A prime implicant p^i is essential if $p^i \# (P - p^i) \# DC \neq \emptyset$.

If the essential prime implicants cover all vertices of the ON-set, then these implicants form the minimum-cost cover.

4. Delete any nonessential p^i that is more expensive (i.e., a smaller cube) than some other prime implicant p^j if $p^i \# DC \# p^j = \emptyset$.
5. Choose the lowest-cost prime implicants to cover the remaining vertices of the ON-set. Use the branching heuristic on the prime implicants of equal cost and retain the cover with the lowest cost.

Example 4.18

To illustrate the minimization procedure, we will use the function

$$f(x_1, x_2, x_3, x_4, x_5) = \sum m(0, 1, 4, 8, 13, 15, 20, 21, 23, 26, 31) + D(5, 10, 24, 28)$$

To help the reader follow the discussion, this function is also shown in the form of a Karnaugh map in Figure 4.41.

The initial cover C^0 consists of the ON-set and the DC-set:

$C^0 = \{00000, 00001, 00100, 01000, 01101, 01111, 10100, 10101, 10111, 11010, 11111, 00101, 01010, 11000, 11100\}$

Using the $*$-operation, the subsequent covers obtained are

$C^1 = \{0000x, 00x00, 0x000, 00x01, x0100, 0010x, 010x0, x1000, 011x1, 0x101, x1111, 1010x, 1x100, 101x1, x0101, 1x111, x1010, 110x0, 11x00\}$

$C^2 = \{0x000, 011x1, 0x101, x1111, 1x100, 101x1, 1x111, 11x00, 00x0x, x010x, x10x0\}$

$C^3 = C^2$

Therefore, $P = C^2$.

Using the #-operation, we find that there are two essential prime implicants: 00x0x (because it is the only one that covers the vertex 00001) and x10x0 (because it is the only

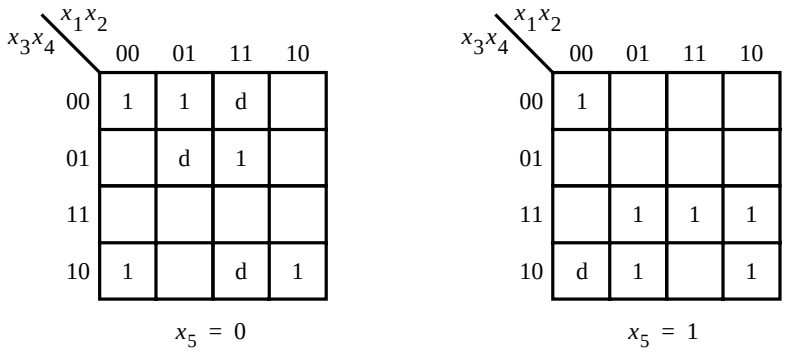


Figure 4.41 The function for Example 4.18.

one that covers the vertex 11010). The minterms of f covered by these two prime implicants are $m(0, 1, 4, 8, 26)$.

Next, we find that 1x100 can be deleted because the only ON-set vertex that it covers is 10100 (m_{20}), which is also covered by x010x and the cost of this prime implicant is lower. Note that having removed 1x100, the prime implicant x010x becomes essential because none of the other remaining prime implicants covers the vertex 10100. Therefore, x010x has to be included in the final cover. It covers $m(20, 21)$.

There remains to find prime implicants to cover $m(13, 15, 23, 31)$. Using the branching heuristic, the lowest-cost cover is obtained by including the prime implicants 011x1 and 1x111. Thus the final cover is

$$C_{minimum} = \{00x0x, x10x0, x010x, 011x1, 1x111\}$$

The corresponding sum-of-products expression is

$$f = \bar{x}_1\bar{x}_2\bar{x}_4 + x_2\bar{x}_3\bar{x}_5 + \bar{x}_2x_3\bar{x}_4 + \bar{x}_1x_2x_3x_5 + x_1x_3x_4x_5$$

Although this procedure is tedious when performed by hand, it is not difficult to write a computer program to implement the algorithm automatically. The reader should check the validity of our solution by finding the optimal realization from the Karnaugh map in Figure 4.41.

4.11 PRACTICAL CONSIDERATIONS

The purpose of the preceding section was to give the reader some idea about how minimization of logic functions may be automated for use in CAD tools. We chose a scheme that is not too difficult to explain. From the practical point of view, this scheme has some drawbacks. The main difficulty is that the number of cubes that must be considered in the process can be extremely large.

If the goal of minimization is relaxed so that it is not imperative to find a minimum-cost implementation, then it is possible to derive heuristic techniques that produce good results in reasonable time. A technique of this type forms the basis of the widely used Espresso program, which is available from the University of California at Berkeley via the World Wide Web. Espresso is a two-level optimization program. Both input to the program and its output are specified in the format of cubes. Instead of using the $*$-operation to find the prime implicants, Espresso uses an implicant-expansion technique. (See problem 4.27 for an illustration of the expansion of implicants.) A comprehensive explanation of Espresso is given in [19], while simplified outlines can be found in [3, 12].

The University of California at Berkeley also provides two software programs that can be used for design of multilevel circuits, called MIS [20] and SIS [21]. They allow a user to apply various multilevel optimization techniques to a logic circuit. The user can experiment with different optimization strategies by applying techniques such as factoring

and decomposition to all or part of a circuit. SIS also includes the Espresso algorithm for two-level minimization of functions, as well as many other optimization techniques.

Numerous commercial CAD systems are on the market. Three companies whose products are widely used are Cadence Design Systems, Mentor Graphics, and Synopsys. Information on their products is available on the World Wide Web. Each company provides logic synthesis software that can be used to target various types of chips, such as PLDs, gate arrays, standard cells, and custom chips. Because there are many possible ways to synthesize a given circuit, as we saw in the previous sections, each commercial product uses a proprietary logic optimization strategy based on heuristics.

To describe CAD tools, some new terminology has been invented. In particular, we should mention two terms that are widely used in industry: *technology-independent logic synthesis* and *technology mapping*. The first term refers to techniques that are applied when optimizing a circuit without considering the resources available in the target chip. Most of the techniques presented in this chapter are of this type. The second term, technology mapping, refers to techniques that are used to ensure that the circuit produced by logic synthesis can be realized using the logic resources available in the target chip. A good example of technology mapping is the transformation from a circuit in the form of logic operations such as AND and OR into a circuit that consists of only NAND operations. This type of technology mapping is done when targeting a circuit to a gate array that contains only NAND gates. Another example is the translation from logic operations to lookup tables, which is done when targeting a design to an FPGA. It should be noted that the terminology is sometimes used inconsistently. For instance, some CAD systems consider factoring, which was discussed in section 4.7.1, to be technology independent, whereas other systems consider it to be a part of the technology mapping. Still other systems, such as MAX+plusII, do not use these two terms at all, even though they clearly implement both types of techniques. We will not rely on these terms in this book and have mentioned them only for completeness.

The next section provides a more detailed discussion of CAD tools. To give an example of the features provided in these tools, we use the MAX+plusII system that accompanies the book. Of course, different CAD systems offer different features. MAX+plusII synthesizes designs for implementation in PLDs. It includes all the optimization techniques introduced in this chapter.

4.12 CAD Tools

In section 2.8 we introduced the concept of a CAD system and described CAD tools for performing design entry, initial synthesis, and functional simulation. In this section we introduce the remaining tools in a typical CAD system, which are used for performing logic synthesis and optimization, physical design, and timing simulation. The principles behind such tools are quite general; the details may vary from one system to another. We will discuss the main aspects of the tools in as general a fashion as possible. However, to provide a sufficient degree of reality, we will use illustrative examples based on the Altera MAX+plusII system that is provided with the book. To fully grasp the concepts presented

in the following discussion, the reader should go through the material in Tutorials 1 and 2, which are presented in Appendices B and C.

A typical CAD system comprises tools for performing the following tasks:

- *Design entry* allows the designer to enter a description of the desired circuit in the form of truth tables, schematic diagrams, or HDL code.
- *Initial synthesis* generates an initial circuit, based on data entered during the design entry stage.
- *Functional simulation* is used to verify the functionality of the circuit, based on inputs provided by the designer.
- *Logic synthesis and optimization* applies optimization techniques to derive an optimized circuit.
- *Physical design* determines how to implement the optimized circuit in a given target technology, for example, in a PLD chip.
- *Timing simulation* determines the propagation delays that are expected in the implemented circuit.
- *Chip configuration* configures the actual chip to realize the designed circuit.

The first three of these tools are discussed in Chapter 2. The rest are described below.

4.12.1 LOGIC SYNTHESIS AND OPTIMIZATION

The optimization techniques described in this chapter are automatically applied by CAD tools when synthesizing logic circuits. Consider the VHDL code in Figure 4.42. It describes the function f from Figure 4.5*a* in the canonical form, which consists of minterms. We used the MAX+plusII system to synthesize f for implementation in a FLEX 10K FPGA. The result obtained was

$$f = \bar{x}_2 x_3 + x_1 \bar{x}_3$$

which is the same minimal sum-of-products expression derived in Figure 4.5*a*. This result was displayed in a *report file*, which is produced by the CAD system. The report file includes a set of logic equations that describe the synthesized circuit.

CAD tools often include many optional features that can be invoked by the user. Figure 4.43 shows some of the logic synthesis options provided by MAX+plusII. Although the reader may not recognize all the options shown, the meaning of terms such as minimization, multilevel synthesis, factoring, and decomposition should be obvious at this point. Detailed explanation of various synthesis procedures can be found in specialized texts [5, 22].

The optimized circuit produced by the logic synthesis tools depends both on the type of logic resources available in the target chip and on the particular CAD system that is used. For example, if the target chip is a CPLD, then each logic function in the circuit is expressed in terms of the gates available in a macrocell. For an FPGA that contains lookup tables (LUTs), the number of inputs to each logic function in the circuit is constrained by the size of the LUTs. If the target chip is a gate array, then the logic functions in the optimized circuit are expressed using only the type of logic cells available in the gate array. Finally,

```
ENTITY func1 IS
     PORT ( x1, x2, x3  : IN     BIT ;
                f       : OUT  BIT ) ;
END func1 ;

ARCHITECTURE LogicFunc OF func1 IS
BEGIN
     f <= (NOT x1 AND NOT x2 AND x3) OR
          (x1 AND NOT x2 AND NOT x3) OR
          (x1 AND NOT x2 AND x3) OR
          (x1 AND x2 AND NOT x3) ;
END LogicFunc ;
```

Figure 4.42 VHDL code for the function in Figure 4.5*a*.

if standard-cell technology is used, then the circuit comprises whatever types of logic cells can be fabricated on the standard-cell chip.

4.12.2 PHYSICAL DESIGN

After logic synthesis the next step in the design flow is to determine exactly how to implement the circuit in the target technology. This step is usually called *physical design*, or *layout synthesis*. There are two main parts to physical design: placement and routing.

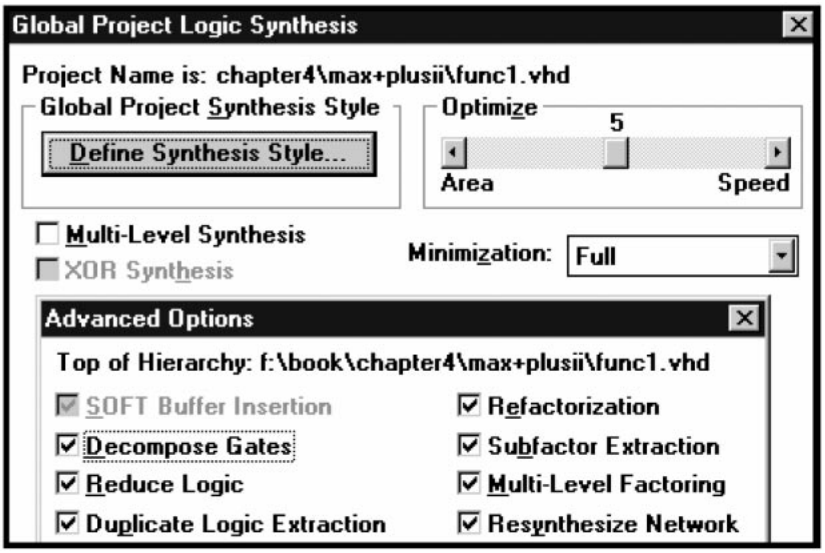


Figure 4.43 Logic synthesis options in MAX+plusII.

A *placement* CAD tool determines where in the target device each logic function in the optimized circuit will be realized. The placement task is highly dependent on the implementation technology. For example, if a PLD is used for implementation, then the structure of the chip is predefined and the placement tool determines which logic resources in the chip are to be used to realize each logic function in the circuit. In the case of a CPLD, the logic functions are assigned to macrocells. For an FPGA each logic function is assigned to a logic cell.

Continuing with our example, the MAX+plusII placement tool realizes the function f from Figure 4.42 in the FLEX 10K FPGA as depicted in Figure 4.44. The figure represents a screen capture of the *Floorplan Editor*, which displays the results generated by the physical design tools. The small squares in the diagram represent the logic cells in the FPGA, which are four-input LUTs (see Appendix E). The logic cell at the top left is used to realize the function f. At the bottom of the window, the Floorplan Editor shows the logic expression contained in the LUT for f. Lines are drawn to indicate the input and output connections of this logic cell. They connect to the I/O cells that are used for inputs x_1, x_2, and x_3, as well as for the output f.

After the placement has been completed, the next step is to decide which of the wires in the chip are to be used to realize the required interconnections. This step is called *routing*. Like the placement task, routing is highly dependent on the implementation technology.

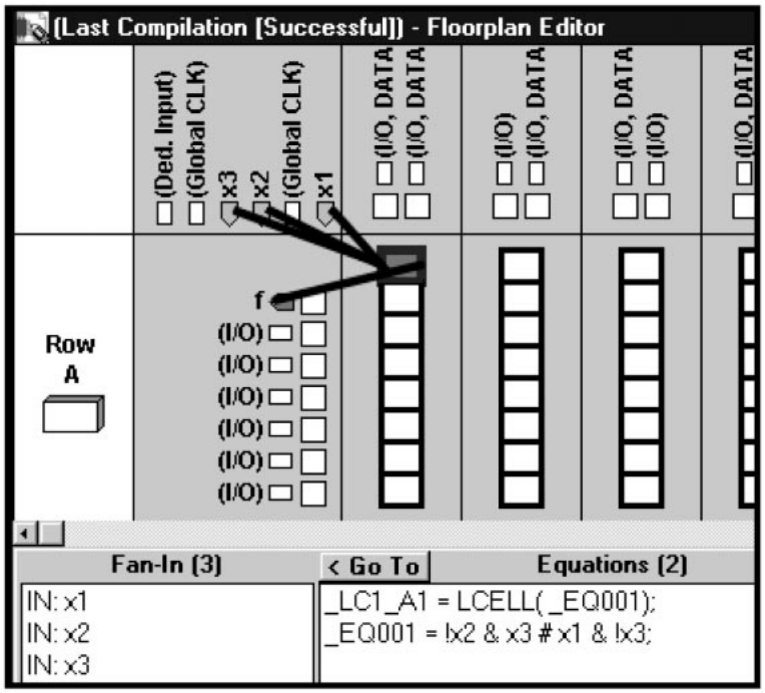


Figure 4.44 The results of physical design for the VHDL code in Figure 4.42.

For a CPLD the programming switches attached to the interconnection wires must be set to connect the macrocells together as needed for the implemented circuit. Similarly, for an FPGA the programming switches are used to connect the logic cells together. If the implementation technology is a gate array or a standard-cell chip, then the routing tool specifies the interconnection wires that are to be fabricated between the rows of logic cells. Some small examples of routing were presented in Chapter 3, in Figures 3.59 and 3.67.

Both the placement and routing tasks can be difficult problems to solve for the CAD tools, especially for the larger devices, such as FPGAs, gate arrays, and standard-cell chips. Much research effort has gone into the development of algorithms for these tasks. Detailed explanations of these algorithms can be found in more specialized books [23, 24].

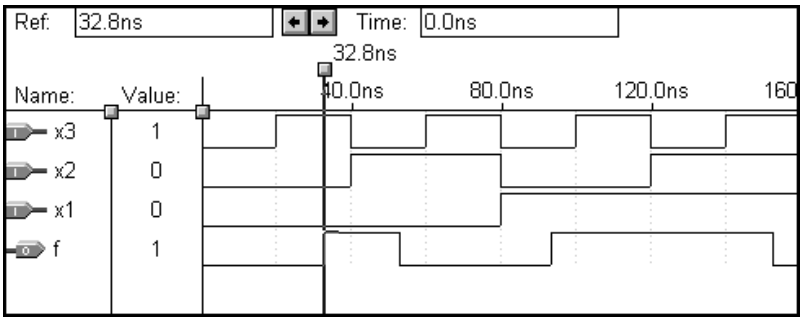
4.12.3 TIMING SIMULATION

In section 2.8.3 we described functional simulation and said that it is used to ensure that a logic circuit description entered into a CAD system functions as expected by the designer. In functional simulation it is assumed that signal propagation delays through logic gates are negligible. In this section we consider *timing simulation*, which simulates the actual propagation delays in the technology chosen for implementation.

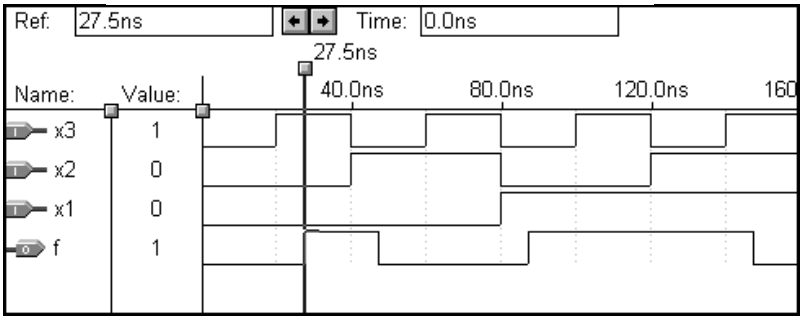
After the physical design tasks are completed, the CAD system has determined exactly how the designed circuit is to be realized in the target technology. It is then possible for the CAD tools to create a model of the circuit that includes all timing aspects of the target chip. The model represents the delays associated with the logic resources in the chip (macrocells or logic cells) and with the interconnection wires.

The results of timing simulation for the function f from Figure 4.42 are shown in Figure 4.45. They were obtained using the timing simulator in MAX+plusII. The simulator allows the designer to specify a waveform for each of the inputs x_1, x_2, and x_3, and the tool generates the corresponding waveform produced at the output f. Part (*a*) of the figure gives the timing expected when the circuit is implemented in the FLEX 10K FPGA. Observe that a heavy vertical line, which is called the *reference line*, is set at the point where f first makes a transition from 0 to 1. The simulator specifies in the box labeled Ref that the reference line is set at 32.8 ns from the start time of the simulation. The change in $x_1x_2x_3$ from 000 to 001 takes place at 20 ns; hence $32.8 - 20 = 12.8$ ns are required for the change in inputs to cause f to change to 1. The reason for the delay at f is that the signals must propagate through the transistor circuits in the FPGA. The timing aspects of transistor circuits are discussed in Chapter 3.

Figure 4.45*b* shows the same simulation for the circuit when it is implemented in a MAX 7000 CPLD. Of course, the circuit implements the same function as when implemented in the FLEX 10K FPGA, but the timing is different. In the MAX 7000 CPLD, f changes 7.5 ns after the inputs change. The speed of a circuit may vary considerably when implemented in different types of chips. Although our example suggests that the CPLD provides much faster speed than the FPGA, the difference is exaggerated because of the small size of the circuit. In general, when larger circuits are implemented, CPLDs and FPGAs provide similar speeds.



(a) Timing in an FPGA



(b) Timing in a CPLD

Figure 4.45 Timing simulation for the VHDL code in Figure 4.42.

4.12.4 SUMMARY OF DESIGN FLOW

Figure 4.46 summarizes the design flow of a complete CAD system. After initial synthesis the logic synthesis tool automatically optimizes the circuit being designed. The physical design tool then determines exactly how to implement the circuit in the chosen technology. Timing simulation ensures that the implemented circuit meets the required performance. Note that if functional correctness has already been ascertained using functional simulation, as discussed in section 2.8, then the functionality of the circuit need not be verified using timing simulation. However, if functional simulation was not done, then timing simulation can be used to check for proper functionality as well. If timing or functional problems are discovered, they are corrected by returning to the previous steps in the design flow. For functional errors it is necessary to revisit the design entry step. For timing errors it may be possible to correct the problems by using the logic synthesis tool. For example, the window displayed in Figure 4.43 shows a sliding bar that can be used to change the

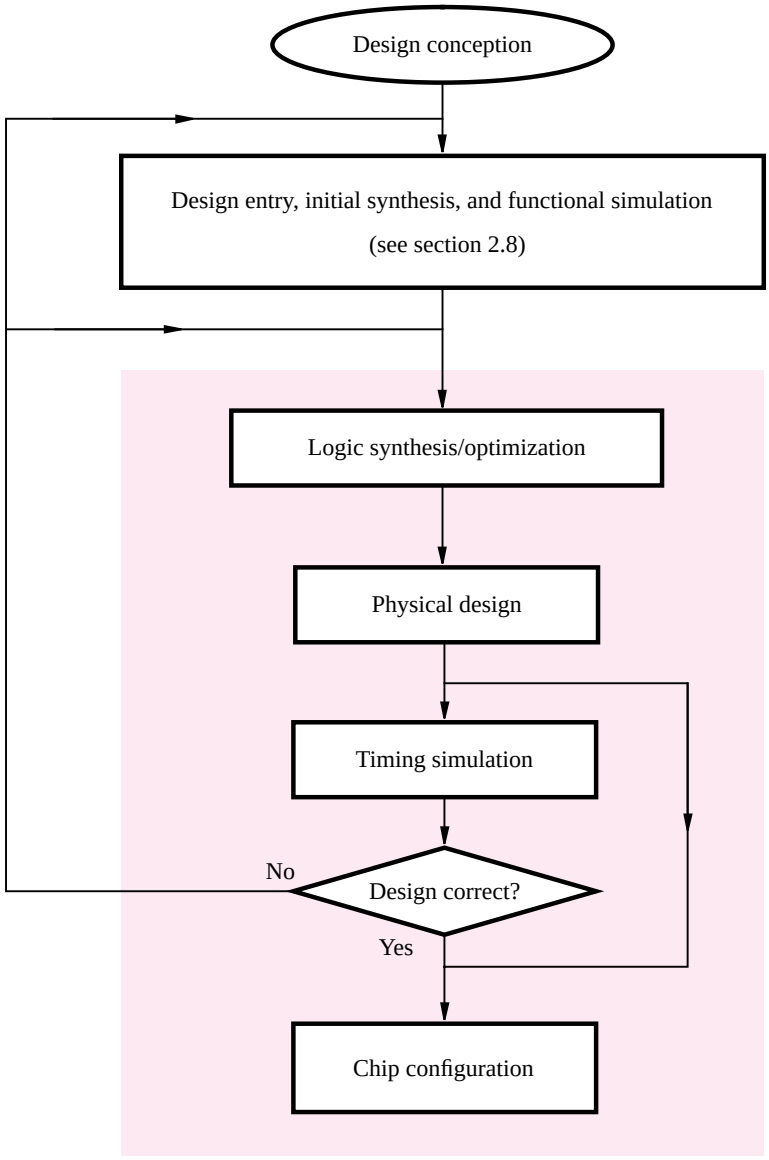


Figure 4.46 A complete CAD system.

emphasis of the logic synthesis algorithms between circuit cost or circuit speed. Cost is optimized by minimizing the amount of area needed on the chip to implement the circuit. Speed is optimized by minimizing the propagation delay of signals in the circuit. It may also be possible to use a faster speed grade of the selected chip or to select a different type of chip that results in a faster circuit, as in the example from Figure 4.45. If the logic synthesis

tool cannot resolve the timing problems, then it is necessary to return to the beginning of the design flow to consider other design alternatives. The final step is to configure the target chip to implement the desired circuit.

4.12.5 EXAMPLES OF CIRCUITS SYNTHESIZED FROM VHDL CODE

In section 2.9 we showed how simple VHDL programs can be written to describe logic functions. This section introduces additional features of VHDL and provides some examples of circuits designed using VHDL code.

Recall that a logic signal is represented in VHDL as a data object, and each data object has an associated type. In the examples in section 2.9, all data objects have the type BIT, which means that they can assume only the values 0 and 1. To give more flexibility, VHDL provides another data type called *STD_LOGIC*. Signals represented using this type can have several different values.

As its name implies, STD_LOGIC is meant to serve as the standard data type for representation of logic signals. An example using the STD_LOGIC type is given in Figure 4.47. The VHDL code shown is the same as that given in Figure 4.42 except that here the type STD_LOGIC is used instead of BIT. The VHDL compiler would synthesize this code in exactly the same way as described for the code in Figure 4.42.

To use the STD_LOGIC type, VHDL code must include the two lines given at the beginning of Figure 4.47. These statements serve as directives to the VHDL compiler. They are needed because the original VHDL standard, IEEE 1076, did not include the STD_LOGIC type. The way that the new type was added to the language, in the IEEE 1164 standard, was to provide the definition of STD_LOGIC as a set of files that can be included with VHDL code when compiled. The set of files is called a *library*. The purpose of the first line in Figure 4.47 is to declare that the code will make use of the IEEE library.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY func2 IS
    PORT ( x1, x2, x3  : IN     STD_LOGIC ;
           f           : OUT  STD_LOGIC ) ;
END func2 ;

ARCHITECTURE LogicFunc OF func2 IS
BEGIN
    f <= (NOT x1 AND NOT x2 AND x3) OR
         (x1 AND NOT x2 AND NOT x3) OR
         (x1 AND NOT x2 AND x3) OR
         (x1 AND x2 AND NOT x3) ;
END LogicFunc ;
```

Figure 4.47 The VHDL code in Figure 4.42 using STD_LOGIC.

In VHDL there are two main aspects to the definition of a new data type. First, the set of values that a data object of the new type can assume must be specified. For STD_LOGIC, there are a number of legal values, but the ones that are the most important for describing logic functions are 0, 1, Z, and −. We introduced the logic value Z, which represents the high-impedance state, in section 3.8.8. The − logic value represents the don't-care condition, which we labeled as *d* in section 4.4. The second requirement is that all legal uses in VHDL code of the new data type must be specified. For example, it is necessary to specify that the type STD_LOGIC is legal for use with Boolean operators.

In the IEEE library one of the files defines the STD_LOGIC data type itself and specifies some basic legal uses, such as for Boolean operations. In Figure 4.47 the second line of code tells the VHDL compiler to use the definitions in this file when compiling the code. The file encapsulates the definition of STD_LOGIC in what is known as a *package*. The package is named std_logic_1164. It is possible to instruct the VHDL compiler to use only a subset of the package, but the normal use is to specify the word *all* to indicate that the entire package is of interest, as we have done in Figure 4.47.

The IEEE library files are plain text files that can be examined with any text editor. Although it is not necessary for purposes of understanding the examples in this book, an interested reader can examine the IEEE library files distributed with the MAX+plusII system that accompanies the book. When the software is installed on a computer running a Microsoft Windows operating system, the IEEE library files are normally installed in the file system in the location C:\maxplus2\max2vhdl\ieee. The file that defines the STD_LOGIC type is named "std1164.vhd."

For the examples of VHDL code given in this book, we will almost always use only the type STD_LOGIC. Besides simplifying the code, using just one data type has another benefit. VHDL is a strongly type-checked language. This means that the VHDL compiler carefully checks all data object assignment statements to ensure that the type of the data object on the left side of the assignment statement is exactly the same as the type of the data object on the right side. Even if two data objects seem compatible from an intuitive point of view, such as an object of type BIT and one of type STD_LOGIC, the VHDL compiler will not allow one to be assigned to the other. Many synthesis tools provide conversion utilities to convert from one type to another, but we will avoid this issue by using only the STD_LOGIC data type in most cases. In the remainder of this section, a few examples of VHDL code are presented. We show the results of synthesizing the code for implementation in two different types of chips, a CPLD and an FPGA.

Example 4.19

Consider the VHDL code in Figure 4.48. The logic expression for f corresponds to the truth table in Figure 4.1. We derived the minimal sum-of-products form, $f = \overline{x}_3 + x_1\overline{x}_2$, using the Karnaugh map in Figure 4.5*b*. If we compile the VHDL code for implementation in a MAX 7000 CPLD, the MAX+plusII tools produce the expression

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

It is easy to show that this expression is not fully minimized. Using the identity 16*a* in section 2.5, the expression can be reduced to $f = \overline{x}_3 + x_1\overline{x}_2$, which is the minimal form that we derived manually. However, because the circuit is being implemented in a CPLD, the

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY func3 IS
    PORT ( x1, x2, x3  : IN     STD_LOGIC ;
           f           : OUT  STD_LOGIC ) ;
END func3 ;

ARCHITECTURE LogicFunc OF func3 IS
BEGIN
    f <= (NOT x1 AND NOT x2 AND NOT x3) OR
         (NOT x1 AND x2 AND NOT x3) OR
         (x1 AND NOT x2 AND NOT x3) OR
         (x1 AND NOT x2 AND x3) OR
         (x1 AND x2 AND NOT x3) ;
END LogicFunc ;
```

Figure 4.48 The VHDL code for the function in Figure 4.1.

extra literal in the product term $x_1\bar{x}_2x_3$ does not increase the cost. Figure 4.49 shows the expression for f realized in a macrocell. Observe that since the XOR gate in the macrocell is not used for the circuit, one input to the XOR gate is connected to 0.

As we have said before, CAD tools include many options that can affect the results of the synthesis procedure. Some of the options available in MAX+plusII are shown in the window in Figure 4.43. One of the options is called *XOR synthesis*, which is a synthesis

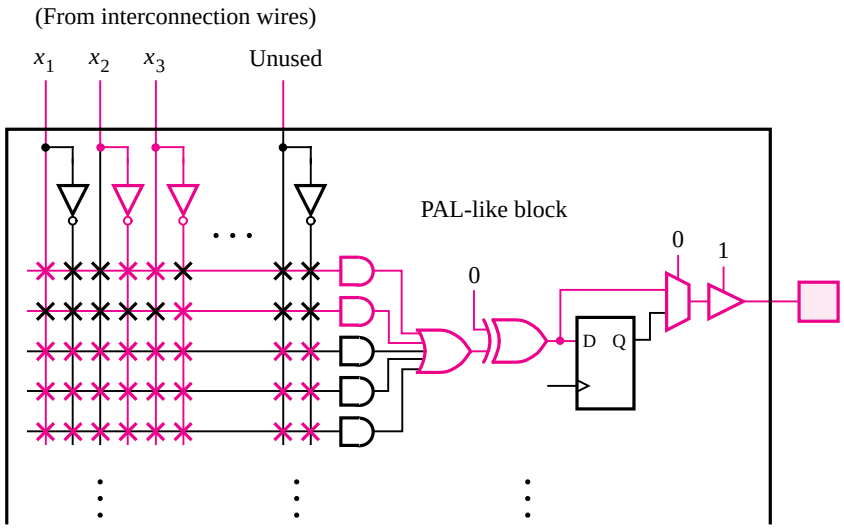


Figure 4.49 Implementation of the VHDL code in Figure 4.48.

technique that attempts to use XOR gates as judiciously as possible. If this option is turned on and the VHDL code in Figure 4.48 is synthesized again, the resulting expression for f becomes

$$f = \overline{x}_3 \oplus x_1\overline{x}_2x_3$$

The reader should verify that this is functionally equivalent to the sum-of-products form given above. The implementation of this expression in a MAX 7000 macrocell is depicted in Figure 4.50. The XOR gate is now used as part of the function, with one input connected to $\overline{x}_3$. Since it occupies a single macrocell, the cost of the implementation is the same as for the circuit in Figure 4.49. Although not true in this example, for some logic functions the XOR gates lead to greatly reduced cost. We should note that it is even possible to realize any arbitrary logic function using only AND and XOR gates [4]. We discuss some typical uses of XOR gates in Chapter 5. As this example illustrates, for any given logic function, several different implementations often have the same cost in a given chip.

Figure 4.51 gives the results of synthesizing the VHDL code in Figure 4.48 into a FLEX 10K FPGA. In this case the compiler generates the same sum-of-products form that we derived manually. Because the logic cells in the FLEX 10K chip are four-input lookup tables, only a single logic cell is needed for this function. The figure shows that the variables x_1, x_2, and x_3 are connected to the LUT inputs called i_2, i_3, and i_4. Input i_1 is not used because the function requires only three inputs. The truth table in the LUT indicates that the unused input is treated as a don't care. Thus only half of the rows in the table are shown, since the other half is identical. The unused LUT input is shown connected to 0 in the figure, but it could just as well be connected to 1.

It is interesting to consider the benefits provided by the optimizations used in logic synthesis. For the implementation in the CPLD, the function was simplified from the original five product terms in the canonical form to just two product terms. However, both

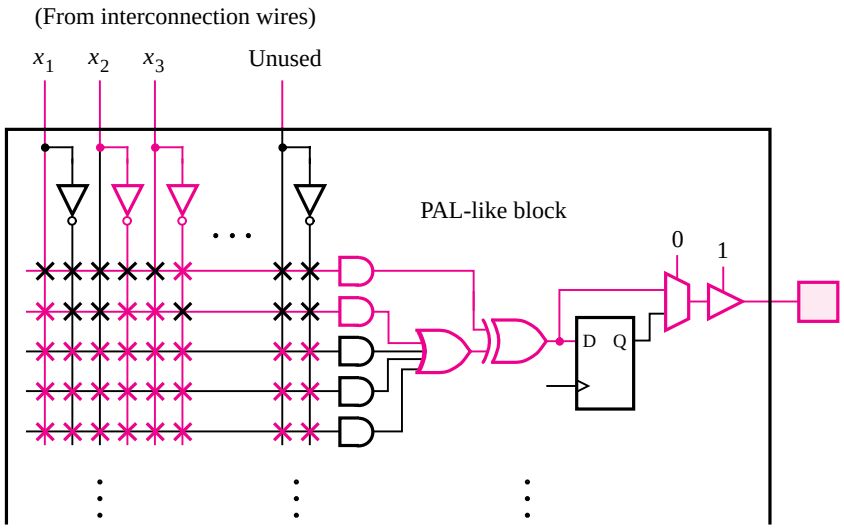


Figure 4.50 Implementation of the VHDL code in Figure 4.48 using XOR synthesis.

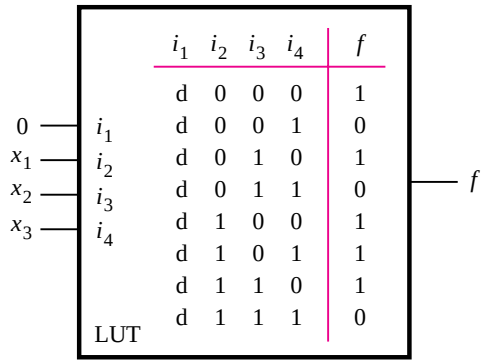


Figure 4.51 The VHDL code in Figure 4.48 implemented in a LUT.

the optimized and nonoptimized forms fit into a single macrocell in the chip, and thus they have the same cost (Appendix E shows that the MAX 7000 CPLD has five product terms in each macrocell). Similarly, for the FPGA, since a LUT is used for implementation, it does not matter whether the function is minimized, because it fits in a single LUT. The reason is that our example circuit is very small. For large circuits it is essential to perform the optimization. Examples 4.20 and 4.21 illustrate logic functions for which the cost of implementation is reduced when optimized.

Example 4.20 The VHDL code in Figure 4.52 corresponds to the function f_1 in Figure 4.7. Because there are six product terms in the canonical form, two macrocells would be needed in a MAX 7000 CPLD. When synthesized by the CAD tools, the resulting expression is

$$f = \bar{x}_2 x_3 + x_1 \bar{x}_3 x_4$$

which is the same as the expression derived in Figure 4.7. Because the optimized expression has only two product terms, it can be realized using just one macrocell and hence results in a lower cost.

When f_1 is synthesized for implementation in a FLEX 10K FPGA, the expression generated is the same as for the CPLD. Since the function has only four inputs, it needs just one LUT.

Example 4.21 In section 4.7 we used a seven-variable logic function as a motivation for multilevel synthesis. This function is given in the VHDL code in Figure 4.53. The logic expression is in minimal sum-of-products form. When it is synthesized for implementation in a MAX 7000 CPLD, no optimizations are performed by the CAD tools. The function requires one macrocell. This function is more interesting when we consider its implementation in the FLEX 10K FPGA. Because there are seven inputs, more than one LUT is required. If the function is implemented directly as given in the VHDL code, then five LUTs are needed, as depicted in Figure 4.54*a*. Rather than showing the truth table programmed in each LUT, we

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY func4 IS
    PORT ( x1, x2, x3, x4  : IN     STD_LOGIC ;
           f               : OUT  STD_LOGIC ) ;
END func4 ;

ARCHITECTURE LogicFunc OF func4 IS
BEGIN
    f <= (NOT x1 AND NOT x2 AND x3 AND NOT x4) OR
         (NOT x1 AND NOT x2 AND x3 AND x4) OR
         (x1 AND NOT x2 AND NOT x3 AND x4) OR
         (x1 AND NOT x2 AND x3 AND NOT x4) OR
         (x1 AND NOT x2 AND x3 AND x4) OR
         (x1 AND x2 AND NOT x3 AND x4) ;
END LogicFunc ;
```

Figure 4.52 The VHDL code for f_1 in Figure 4.7.

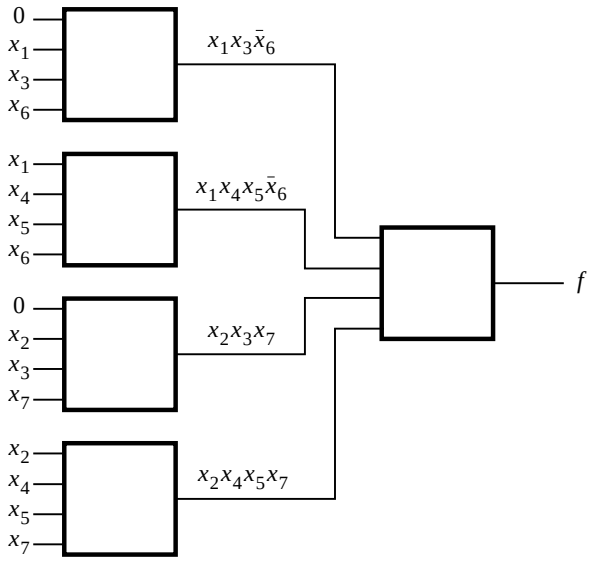
show the logic function that is implemented at the LUT output. Synthesis with MAX+plusII results in the following expression:

$$f = (x_1\bar{x}_6 + x_2x_7)(x_3 + x_4x_5)$$

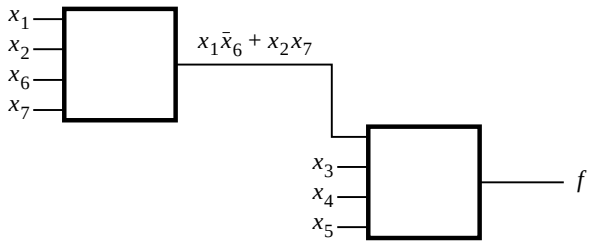
We derived the same expression by using factoring in section 4.7. As illustrated in Figure 4.54*b*, it can be implemented using only two LUTs. One LUT produces the term $S = x_1\bar{x}_6 + x_2x_7$. The other LUT implements the four-input function $f = Sx_3 + Sx_4x_5$.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY func5 IS
    PORT ( x1, x2, x3, x4, x5, x6, x7  : IN     STD_LOGIC ;
           f                           : OUT  STD_LOGIC ) ;
END func5 ;

ARCHITECTURE LogicFunc OF func5 IS
BEGIN
    f <= (x1 AND x3 AND NOT x6) OR
         (x1 AND x4 AND x5 AND NOT x6) OR
         (x2 AND x3 AND x7) OR
         (x2 AND x4 AND x5 AND x7) ;
END LogicFunc ;
```

Figure 4.53 The VHDL code for the function of section 4.7.



(a) Sum-of-products realization



(b) Factored realization

Figure 4.54 Implementation of the VHDL code in Figure 4.53.

4.13 CONCLUDING REMARKS

This chapter has attempted to provide the reader with an understanding of various aspects of synthesis for logic functions and how synthesis is automated using modern CAD tools. Now that the reader is comfortable with the fundamental concepts, we can examine digital circuits of a more sophisticated nature. The next chapter describes circuits that perform arithmetic operations, which are a key part of computers.

PROBLEMS

4.1 Find the minimum-cost SOP and POS forms for the function $f(x_1, x_2, x_3) = \sum m(1, 2, 3, 5)$.

4.2 Repeat problem 4.1 for the function $f(x_1, x_2, x_3) = \sum m(1, 4, 7) + D(2, 5)$.

4.3 Repeat problem 4.1 for the function $f(x_1, \ldots, x_4) = \Pi M(0, 1, 2, 4, 5, 7, 8, 9, 10, 12, 14, 15)$.

4.4 Repeat problem 4.1 for the function $f(x_1, \ldots, x_4) = \sum m(0, 2, 8, 9, 10, 15) + D(1, 3, 6, 7)$.

4.5 Repeat problem 4.1 for the function $f(x_1, \ldots, x_5) = \Pi M(1, 4, 6, 7, 9, 12,15, 17, 20, 21, 22, 23, 28, 31)$.

4.6 Repeat problem 4.1 for the function $f(x_1, \ldots, x_5) = \sum m(0, 1, 3, 4, 6, 8, 9, 11, 13, 14, 16, 19, 20, 21, 22, 24, 25) + D(5, 7, 12, 15, 17, 23)$.

4.7 Repeat problem 4.1 for the function $f(x_1, \ldots, x_5) = \sum m(1, 4, 6, 7, 9, 10, 12, 15, 17, 19, 20, 23, 25, 26, 27, 28, 30, 31) + D(8, 16, 21, 22)$.

4.8 Find 5 three-variable functions for which the product-of-sums form has lower cost than the sum-of-products form.

4.9 A four-variable logic function that is equal to 1 if any three or all four of its variables are equal to 1 is called a *majority* function. Design a minimum-cost circuit that implements this majority function.

4.10 Derive a minimum-cost realization of the four-variable function that is equal to 1 if exactly two or exactly three of its variables are equal to 1; otherwise it is equal to 0.

4.11 Prove or show a counter-example for the statement: If a function f has a unique minimum-cost SOP expression, then it also has a unique minimum-cost POS expression.

4.12 A circuit with two outputs has to implement the following functions

$$f(x_1, \ldots, x_4) = \sum m(0, 2, 4, 6, 7, 9) + D(10, 11)$$

$$g(x_1, \ldots, x_4) = \sum m(2, 4, 9, 10, 15) + D(0, 13, 14)$$

Design the minimum-cost circuit and compare its cost with combined costs of two circuits that implement f and g separately. Assume that the input variables are available in both uncomplemented and complemented forms.

4.13 Repeat problem 4.12 for the following functions

$$f(x_1, \ldots, x_5) = \sum m(1, 4, 5, 11, 27, 28) + D(10, 12, 14, 15, 20, 31)$$

$$g(x_1, \ldots, x_5) = \sum m(0, 1, 2, 4, 5, 8, 14, 15, 16, 18, 20, 24, 26, 28, 31) + D(10, 11, 12, 27)$$

4.14 Implement the logic circuit in Figure 4.26 using NAND gates only.

4.15 Implement the logic circuit in Figure 4.26 using NOR gates only.

4.16 Implement the logic circuit in Figure 4.28 using NAND gates only.

4.17 Implement the logic circuit in Figure 4.28 using NOR gates only.

4.18 Consider the function $f = x_3x_5 + \bar{x}_1x_2x_4 + x_1\bar{x}_2\bar{x}_4 + x_1x_3\bar{x}_4 + \bar{x}_1x_3x_4 + \bar{x}_1x_2x_5 + x_1\bar{x}_2x_5$. Derive a minimum-cost circuit that implements this function using NOT, AND, and OR gates.

4.19 Derive a minimum-cost circuit that implements the function $f(x_1, \ldots, x_4) = \sum m(4, 7, 8, 11) + D(12, 15)$.

4.20 Find the simplest realization of the function $f(x_1, \ldots, x_4) = \sum m(0, 3, 4, 7, 9, 10, 13, 14)$, assuming that the logic gates have a maximum fan-in of two.

4.21 Find the minimum-cost circuit for the function $f(x_1, \ldots, x_4) = \sum m(0, 4, 8, 13, 14, 15)$. Assume that the input variables are available in uncomplemented form only. (Hint: use functional decomposition.)

4.22 Use functional decomposition to find the best implementation of the function $f(x_1, \ldots, x_5) = \sum m(1, 2, 7, 9, 10, 18, 19, 25, 31) + D(0, 15, 20, 26)$. How does your implementation compare with the lowest-cost SOP implementation? Give the costs.

4.23 Show that the following distributive-like rules are valid

$$(A \cdot B)\#C = (A\#C) \cdot (B\#C)$$

$$(A + B)\#C = (A\#C) + (B\#C)$$

4.24 Use the cubical representation and the method discussed in section 4.10 to find a minimum-cost SOP realization of the function $f(x_1, \ldots, x_4) = \sum m(0, 2, 4, 5, 7, 8, 9, 15)$.

4.25 Repeat problem 4.24 for the function $f(x_1, \ldots, x_5) = \bar{x}_1\bar{x}_3\bar{x}_5 + x_1x_2\bar{x}_3 + x_2x_3\bar{x}_4x_5 + x_1\bar{x}_2\bar{x}_3x_4 + x_1x_2x_3x_4\bar{x}_5 + \bar{x}_1x_2x_4\bar{x}_5 + \bar{x}_1\bar{x}_3x_4x_5$.

4.26 Use the cubical representation and the method discussed in section 4.10 to find a minimum-cost SOP realization of the function $f(x_1, \ldots, x_4)$ defined by the ON-set ON = {00x0, 100x, x010, 1111} and the don't-care set DC = {00x1, 011x}.

4.27 In section 4.10.1 we showed how the $*$-product operation can be used to find the prime implicants of a given function f. Another possibility is to find the prime implicants by expanding the implicants in the initial cover of the function. An implicant is *expanded* by removing one literal to create a larger implicant (in terms of the number of vertices covered). A larger implicant is valid only if it does not include any vertices for which $f = 0$. The largest valid implicants obtained in the process of expansion are the prime

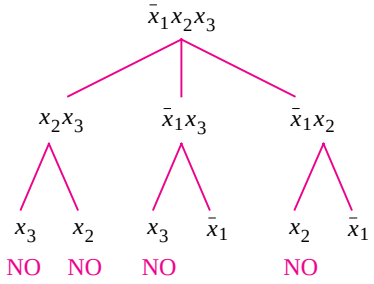


Figure P4.1 Expansion of implicant $\bar{x}_1x_2x_3$.

implicants. Figure P4.1 illustrates the expansion of the implicant $\bar{x}_1x_2x_3$ of the function in Figure 4.9, which is also used in Example 4.14. Note from Figure 4.9 that

$$\bar{f} = x_1\bar{x}_2\bar{x}_3 + x_1\bar{x}_2x_3 + x_1x_2\bar{x}_3$$

In Figure P4.1 the word NO is used to indicate that the expanded term is not valid, because it includes one or more vertices from $\bar{f}$. From the graph it is clear that the largest valid implicants that arise from this expansion are x_2x_3 and $\bar{x}_1$; they are prime implicants of f.

Expand the other four implicants given in the initial cover in Example 4.14 to find all prime implicants of f. What is the relative complexity of this procedure compared to the $*$-product technique?

Note: A technique based on such expansion of implicants is used to find the prime implicants in the Espresso CAD program [19].

4.28 Repeat problem 4.27 for the function in Example 4.15. Expand the implicants given in the initial cover C^0.

4.29 Consider the logic expressions

$$f = x_1\bar{x}_2\bar{x}_5+\bar{x}_1\bar{x}_2\bar{x}_4\bar{x}_5+x_1x_2x_4x_5+\bar{x}_1\bar{x}_2x_3\bar{x}_4+x_1\bar{x}_2x_3x_5+\bar{x}_2\bar{x}_3x_4\bar{x}_5+x_1x_2x_3x_4\bar{x}_5$$

$$g = \bar{x}_2x_3\bar{x}_4+\bar{x}_2\bar{x}_3\bar{x}_4\bar{x}_5+x_1x_3x_4\bar{x}_5+x_1\bar{x}_2x_4\bar{x}_5+x_1x_3x_4x_5+\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_5+x_1x_2\bar{x}_3x_4x_5$$

Prove or disprove that $f = g$.

4.30 Consider the circuit in Figure P4.2, which implements functions f and g. What is the cost of this circuit, assuming that the input variables are available in both true and complemented forms? Redesign the circuit to implement the same functions, but at as low a cost as possible. What is the cost of your circuit?

4.31 Repeat problem 4.30 for the circuit in Figure P4.3. Use only NAND gates in your circuit.

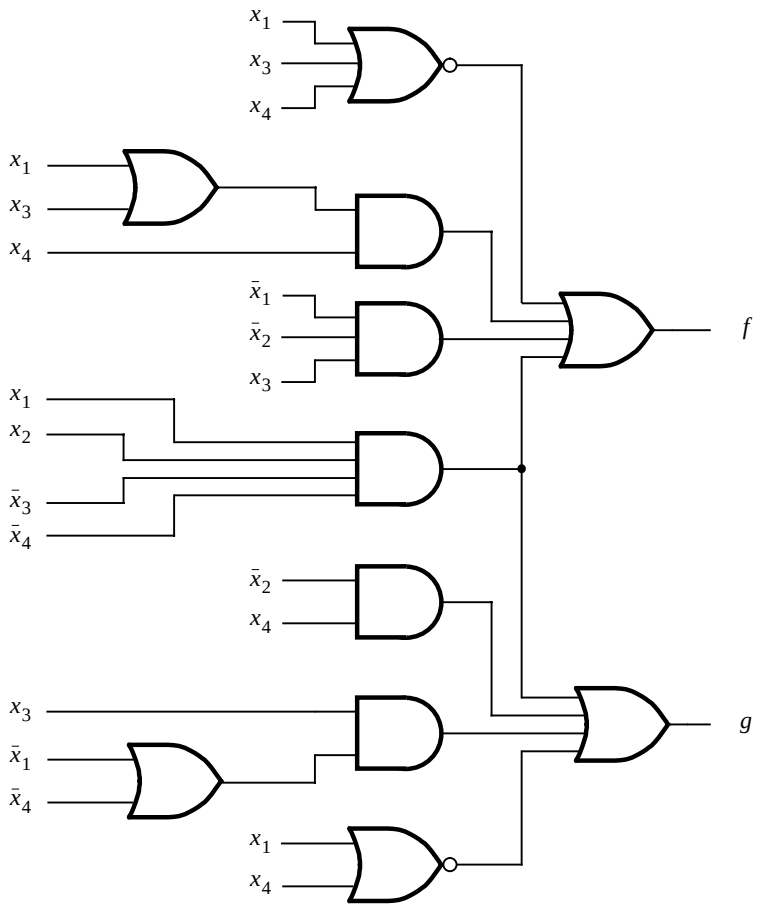


Figure P4.2 Circuit for problem 4.30.

REFERENCES

1. M. Karnaugh, "A Map Method for Synthesis of Combinatorial Logic Circuits," *Transactions of AIEE, Communications and Electronics* 72, part 1, November 1953, pp. 593–599.
2. R. L. Ashenhurst, "The Decomposition of Switching Functions," Proc. of the Symposium on the Theory of Switching, 1957, *Vol. 29 of Annals of Computation Laboratory* (Harvard University: Cambridge, MA, 1959), pp. 74–116.
3. F. J. Hill and G. R. Peterson, *Computer Aided Logical Design with Emphasis on VLSI*, 4th ed. (Wiley: New York, 1993).
4. T. Sasao, *Logic Synthesis and Optimization* (Kluwer: Boston, MA, 1993).

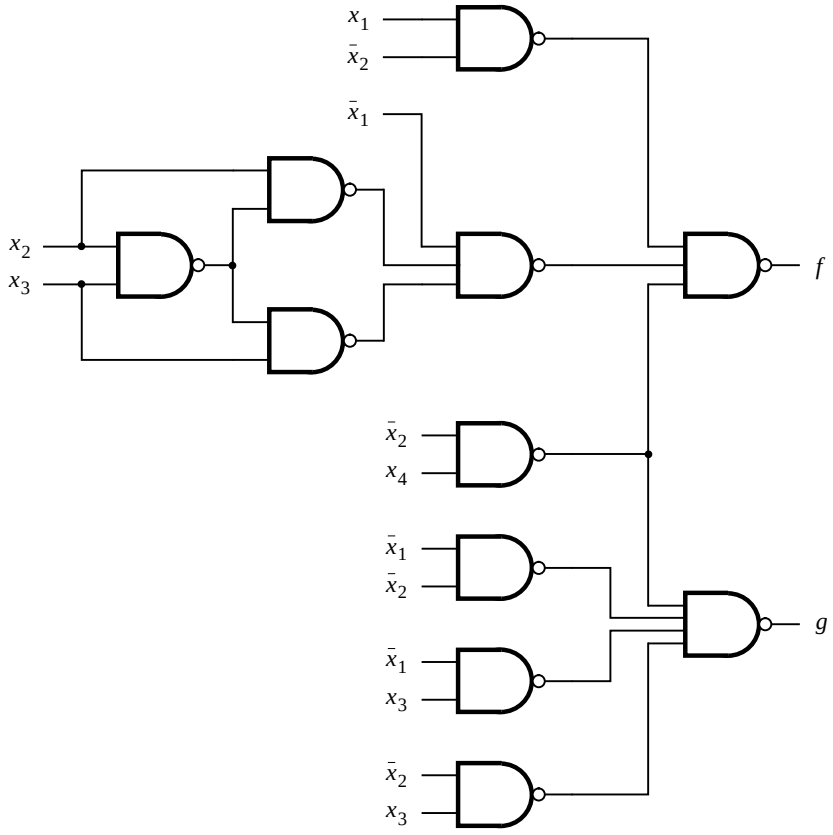


Figure P4.3 Circuit for problem 4.31.

5. S. Devadas, A. Gosh, and K. Keutzer, *Logic Synthesis* (McGraw-Hill: New York, 1994).
6. W. V. Quine, "The Problem of Simplifying Truth Functions," *Amer. Math. Monthly* 59 (1952), pp. 521–31.
7. E. J. McCluskey Jr., "Minimization of Boolean Functions," *Bell System Tech. Journal*, November 1956, pp. 521–31.
8. E. J. McCluskey, *Logic Design Principles* (Prentice-Hall: Englewood Cliffs, NJ, 1986).
9. J. F. Wakerly, *Digital Design Principles and Practices* (Prentice-Hall: Englewood Cliffs, NJ, 1990).
10. J. P. Hayes, *Introduction to Logic Design* (Addison-Wesley: Reading, MA, 1993).
11. C. H. Roth Jr., *Fundamentals of Logic Design*, 4th ed. (West: St. Paul, MN, 1993).
12. R. H. Katz, *Contemporary Logic Design* (Benjamin/Cummings: Redwood City, CA, 1994).

13. V. P. Nelson, H. T. Nagle, B. D. Carroll, and J. D. Irwin, *Digital Logic Circuit Analysis and Design* (Prentice-Hall: Englewood Cliffs, NJ, 1995).
14. J. P. Daniels, *Digital Design from Zero to One* (Wiley: New York, 1996).
15. P. K. Lala, *Practical Digital Logic Design and Testing* (Prentice-Hall: Englewood Cliffs, NJ, 1996).
16. A. Dewey, *Analysis and Design of Digital Systems with VHDL* (PWS Publishing Co.: Boston, MA, 1997).
17. M. M. Mano and C. R. Kime, *Logic and Computer Design Fundamentals* (Prentice-Hall: Upper Saddle River, NJ, 1997).
18. D. D. Gajski, *Principles of Digital Design* (Prentice-Hall: Upper Saddle River, NJ, 1997).
19. R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis* (Kluwer: Boston, MA, 1984).
20. R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A Multiple-Level Logic Synthesis Optimization System," *IEEE Transactions on Computer-Aided Design*, CAD-6, November 1987, pp. 1062–81.
21. E. M. Sentovic, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," Technical Report UCB/ERL M92/41, Electronics Research Laboratory, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1992.
22. G. De Micheli, *Synthesis and Optimization of Digital Circuits* (McGraw-Hill: New York, 1994).
23. N. Sherwani, *Algorithms for VLSI Physical Design Automation* (Kluwer: Boston, MA, 1995).
24. B. Preas and M. Lorenzetti, *Physical Design Automation of VLSI Systems* (Benjamin/Cummings: Redwood City, CA, 1988).