APPENDIX

A

LOGIC CIRCUITS

Information in digital computers is represented and processed by electronic networks called *logic circuits*. These circuits operate on *binary variables* that assume one of two distinct values, usually called 0 and 1. In this appendix we will give a concise presentation of logic functions and circuits for their implementation, including a brief review of integrated circuit technology.

A.1 BASIC LOGIC FUNCTIONS

It is helpful to introduce the topic of binary logic by examining a practical problem that arises in all homes. Consider a lightbulb whose on/off status is controlled by two switches, x_1 and x_2. Each switch can be in one of two possible positions, 0 or 1, as shown in Figure A.1*a*. It can thus be represented by a binary variable. We will let the switch names serve as the names of the associated binary variables. The figure also shows an electrical power supply and a lightbulb. The way the switch terminals are interconnected determines how the switches control the light. The light will be on only if a closed path exists from the power supply through the switch network to the lightbulb. Let a binary variable f represent the condition of the light. If the light is on, $f = 1$, and if the light is off, $f = 0$. Thus, $f = 1$ means that there is at least one closed path through the network, and $f = 0$ means that there is no closed path. Clearly, f is a function of the two variables x_1 and x_2.

Let us consider some possibilities for controlling the light. First, suppose that the light is to be on if either switch is in the 1 position, that is, $f = 1$ if

$$x_1 = 1 \quad \text{and} \quad x_2 = 0$$

or

$$x_1 = 0 \quad \text{and} \quad x_2 = 1$$

or

$$x_1 = 1 \quad \text{and} \quad x_2 = 1$$

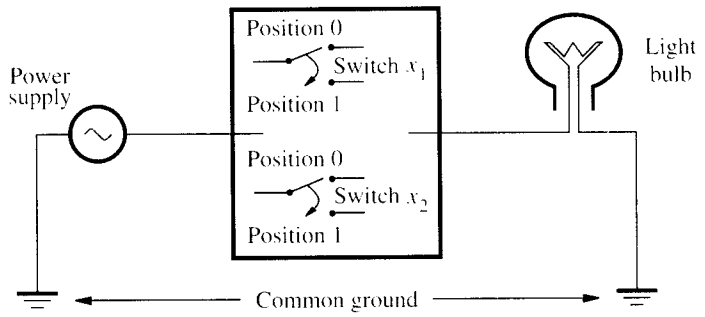
The connections that implement this type of control are shown in Figure A.1*b*. A logic *truth table* that represents this situation is shown beside the wiring diagram. The table lists all possible switch settings along with the value of f for each setting. In logic terms, this table represents the OR function of the two variables x_1 and x_2. The operation is represented algebraically by a "+" sign or a "$\vee$" sign, so that

$$f = x_1 + x_2 = x_1 \vee x_2$$

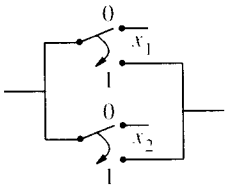
We say that x_1 and x_2 are the *input* variables and f is the *output* function.

We should point out some basic properties of the OR operation. It is commutative, that is,

$$x_1 + x_2 = x_2 + x_1$$

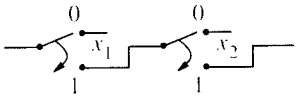


(a) Light bulb controlled by two switches



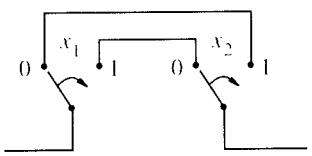
x_1	x_2	$f(x_1, x_2) = x_1 + x_2$
0	0	0
0	1	1
1	0	1
1	1	1

(b) Parallel connection (OR control)



x_1	x_2	$f(x_1, x_2) = x_1 \cdot x_2$
0	0	0
0	1	0
1	0	0
1	1	1

(c) Series connection (AND control)



x_1	x_2	$f(x_1, x_2) = x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

(d) EXCLUSIVE-OR connection (XOR control)

Figure A.1 Light switch example.

It can be extended to n variables, so that

$$f = x_1 + x_2 + \cdots + x_n$$

has the value 1 if any variable x_i has the value 1. This represents the effect of connecting more switches in parallel with the two switches in Figure A.1*b*. Also, inspection of the truth table shows that

$$1 + x = 1$$

and

$$0 + x = x$$

Now, suppose that the light is to be on only when both switches are in the 1 position. The connections for this, along with the corresponding truth-table representation, are shown in Figure A.1*c*. This is the AND function, which uses the symbol "$\cdot$" or "$\wedge$" and is denoted as

$$f = x_1 \cdot x_2 = x_1 \wedge x_2$$

Some basic properties of the AND operation are

$$x_1 \cdot x_2 = x_2 \cdot x_1$$

$$1 \cdot x = x$$

and

$$0 \cdot x = 0$$

The AND function also extends to n variables, with

$$f = x_1 \cdot x_2 \cdot \cdots \cdot x_n$$

having the value 1 only if all the x_i variables have the value 1. This represents the case in which more switches are connected in series with the two switches in Figure A.1*c*.

The final possibility that we will discuss for the way the switches determine the light status is another common situation. If we assume that the switches are at the two ends of a stairway, it should be possible to turn the light on or off from either switch. That is, if the light is on, changing either switch position should turn it off; and if it is off, changing either switch position should turn it on. Assume that the light is off when both switches are in the 0 position. Then changing either switch to the 1 position should turn the light on. Now suppose that the light is on with $x_1 = 1$ and $x_2 = 0$. Switching x_1 back to 0 will obviously turn the light off. Furthermore, it must be possible to turn the light off by changing x_2 to 1, that is, $f = 0$ if $x_1 = x_2 = 1$. The connections to implement this type of control are shown in Figure A.1*d*. The corresponding logic operation is called the EXCLUSIVE-OR (XOR) function, which is represented by the symbol "$\oplus$". Some of its properties are

$$x_1 \oplus x_2 = x_2 \oplus x_1$$

$$1 \oplus x = \overline{x}$$

and

$$0 \oplus x = x$$

where $\bar{x}$ denotes the NOT function of the variable x. This single-variable function, $f = \bar{x}$, has the value 1 if $x = 0$ and the value 0 if $x = 1$. We say that the input x is being *inverted* or *complemented*.

A.1.1 ELECTRONIC LOGIC GATES

The use of switches, closed or open electrical paths, and lightbulbs to illustrate the idea of logic variables and functions is convenient because of their familiarity and simplicity. The logic concepts that have been introduced are equally applicable to the electronic circuits used to process information in digital computers. The physical variables are electrical voltages and currents instead of switch positions and closed or open paths. For example, consider a circuit that is designed to operate on inputs that are at either +5 or 0 volts. The circuit outputs are also at either +5 or 0 V. Now, if we say that +5 V represents logic 1 and 0 V represents logic 0, then we can describe what the circuit does by specifying the truth table for the logic operation that it performs.

With the help of transistors, it is possible to design simple electronic circuits that perform logic operations such as AND, OR, XOR, and NOT. It is customary to use the name *gates* for these basic logic circuits. Standard symbols for these gates are shown in Figure A.2. A somewhat more compact graphical notation for the NOT operation

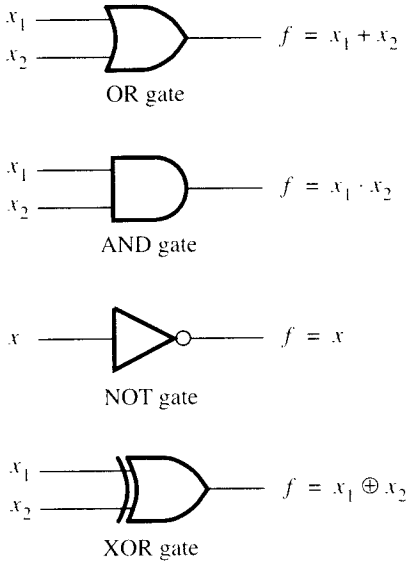


Figure A.2 Standard logic gate symbols.

is used when inversion is applied to a logic gate input or output. In such cases, the inversion is denoted by a small circle.

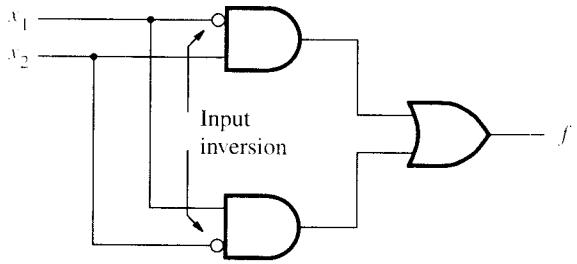
The electronic implementation of logic gates will be discussed in Section A.5. We will now proceed to discuss how basic gates can be used to construct logic networks that implement more complex logic functions.

A.2 SYNTHESIS OF LOGIC FUNCTIONS

Consider the network composed of two AND gates and one OR gate that is shown in Figure A.3*a*. It can be represented by the expression

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

The construction of the truth table for this expression is shown in Figure A.3*b*. First, the values of the AND terms are determined for each input valuation. Then the values of the function f are determined using the OR operation. The truth table for f is identical



$f = x_1 \cdot x_2 + x_1 \cdot x_2$

(a) Network for the XOR function

x_1	x_2	$x_1 \cdot x_2$	$x_1 \cdot x_2$	$f = x_1 \cdot x_2 + x_1 \cdot x_2$ $= x_1 \oplus x_2$
0	0	0	0	0
0	1	1	0	1
1	0	0	1	1
1	1	0	0	0

(b) Truth table construction of $x_1 \cdot x_2 + x_1 \cdot x_2$

Figure A.3 Implementation of the XOR function using AND, OR, and NOT gates.

Table A.1 Two 3-variable functions

x_1	x_2	x_3	f_1	f_2
0	0	0	1	1
0	0	1	1	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	1
1	1	0	0	0
1	1	1	1	0

to the truth table for the XOR function, so the three-gate network in Figure A.3*a* is an implementation of the XOR function using AND, OR, and NOT gates. The logic expression $\overline{x}_1 \cdot x_2 + x_1 \cdot \overline{x}_2$ is called a *sum-of-products* form because the OR operation is sometimes called the "sum" function and the AND operation the "product" function.

We should note that it would be more proper to write

$$f = ((\overline{x}_1) \cdot x_2) + (x_1 \cdot (\overline{x}_2))$$

to indicate the order of applying the operations in the expression. To simplify the appearance of such expressions, we define a hierarchy among the three operations AND, OR, and NOT. In the absence of parentheses, operations in a logic expression should be performed in the following order: NOT, AND, and then OR. Furthermore, it is customary to omit the "·" operator when there is no ambiguity.

Returning to the sum-of-products form, we will now explain how any logic function can be synthesized in this form directly from its truth table. Consider the truth table of Table A.1 and suppose we wish to synthesize the function f_1 using AND, OR, and NOT gates. For each row of the table in which $f_1 = 1$, we include a product (AND) term in the sum-of-products form. The product term includes all three input variables. The NOT operator is applied to these variables individually so that the term is 1 only when the variables have the particular valuation that corresponds to that row of the truth table. This means that if $x_i = 0$, then $\overline{x}_i$ is entered in the product term, and if $x_i = 1$, then x_i is entered. For example, the fourth row of the table has the function entry 1 for the input valuation

$$(x_1, x_2, x_3) = (0, 1, 1)$$

The product term corresponding to this is $\overline{x}_1 x_2 x_3$. Doing this for all rows in which the function f_1 has the value 1 leads to

$$f_1 = \overline{x}_1 \overline{x}_2 \overline{x}_3 + \overline{x}_1 \overline{x}_2 x_3 + \overline{x}_1 x_2 x_3 + x_1 x_2 x_3$$

The logic network corresponding to this expression is shown on the left side in Figure A.4. As another example, the sum-of-products expression for the XOR function

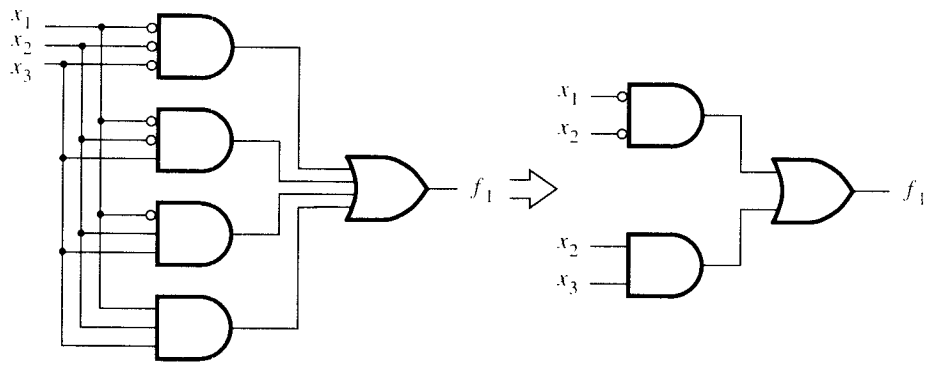


Figure A.4 A logic network for f_1 of Table A.1 and an equivalent minimal network.

can be derived from its truth table using this technique. This approach can be used to derive sum-of-products expressions and the corresponding logic networks for truth tables of any size.

A.3 MINIMIZATION OF LOGIC EXPRESSIONS

We have shown how to derive one sum-of-products expression for each truth table. In fact, there are many equivalent expressions and logic networks for any particular truth table. Two logic expressions or logic gate networks are equivalent if they have identical truth tables. An expression that is equivalent to the sum-of-products expression we derived for f_1 in the previous section is

$$\overline{x}_1\overline{x}_2 + x_2x_3$$

To prove this, we construct the truth table for the simpler expression and show that it is identical to the truth table for f_1 in Table A.1. This is done in Table A.2. The construction of the table for $\overline{x}_1\overline{x}_2 + x_2x_3$ is done in three steps. First, the value of the product term $\overline{x}_1\overline{x}_2$ is computed for each valuation of the inputs. Then x_2x_3 is evaluated. Finally, these two columns are ORed together to obtain the truth table for the expression. This truth table is identical to the truth table for f_1 given in Table A.1.

To simplify logic expressions we will perform a series of algebraic manipulations. The new logic rules that we will use in these manipulations are the distributive rule

$$w(y + z) = wy + wz$$

and the identity

$$w + \overline{w} = 1$$

Table A.2 Evaluation of the expression $\bar{x}_1\bar{x}_2 + x_2x_3$

x_1	x_2	x_3	$\bar{x}_1\bar{x}_2$	x_2x_3	$\bar{x}_1\bar{x}_2 + x_2x_3 = f_1$
0	0	0	1	0	1
0	0	1	1	0	1
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	0	0	0
1	0	1	0	0	0
1	1	0	0	0	0
1	1	1	0	1	1

Table A.3 Truth-table technique for proving equivalence of expressions

				Left-hand side			Right-hand side
w	y	z	$y+z$	$w(y+z)$	wy	wz	$wy + wz$
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

Table A.3 shows the truth-table proof of the distributive rule. It should now be clear that rules such as this can always be proved by constructing the truth tables for the left-hand side and the right-hand side to show that they are identical. Logic rules, such as the distributive rule, are sometimes called *identities*. Although we will not need to use it here, another form of distributive rule that we should include for completeness is

$$w + yz = (w + y)(w + z)$$

The objective in logic minimization is to reduce the cost of implementation of a given logic function according to some criterion. More particularly, we wish to start with a sum-of-products expression derived from a truth table and simplify it to an equivalent *minimal sum-of-products* expression. To define the criterion for minimization, it is necessary to introduce a size or cost measure for a sum-of-products expression. The

usual cost measure is a count of the total number of gates and gate inputs required in implementing the expression in the form shown in Figure A.4. For example, the larger expression in this figure has a cost of 21, composed of a total of 5 gates and 16 gate inputs. Input inversions are ignored in this counting process. The cost of the simpler expression is 9, composed of 3 gates and 6 inputs. We are now in a position to state that a sum-of-products expression is minimal if there is no other equivalent sum-of-products expression with a lower cost. In the simple examples that we will introduce, it is usually reasonably clear when we have arrived at a minimal expression. Thus, we will not give rigorous proofs of minimality.

The general strategy in performing algebraic manipulations to simplify a given expression is as follows. First, group product terms in pairs that differ only in that some variable appears complemented ($\overline{x}$) in one term and true (x) in the other. When the common subproduct consisting of the other variables is factored out of the pair by the distributive rule, we are left with the term $x + \overline{x}$, which has the value 1. Applying this procedure to the first expression for f_1, we obtain

$$\begin{aligned} f_1 &= \overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_1\overline{x}_2x_3 + \overline{x}_1x_2x_3 + x_1x_2x_3 \\ &= \overline{x}_1\overline{x}_2(\overline{x}_3 + x_3) + (\overline{x}_1 + x_1)x_2x_3 \\ &= \overline{x}_1\overline{x}_2 \cdot 1 + 1 \cdot x_2x_3 \\ &= \overline{x}_1\overline{x}_2 + x_2x_3 \end{aligned}$$

This expression is minimal. The network corresponding to it is shown in Figure A.4.

The grouping of terms in pairs so that minimization can lead to the simplest expression is not always as obvious as it is in the preceding example. A rule that is often helpful is

$$w + w = w$$

This allows us to repeat product terms so that a particular term can be combined with

Table A.4 Rules of binary logic

Name	Algebraic identity	
Commutative	$w + y = y + w$	$wy = yw$
Associative	$(w + y) + z = w + (y + z)$	$(wy)z = w(yz)$
Distributive	$w + yz = (w + y)(w + z)$	$w(y + z) = wy + wz$
Idempotent	$w + w = w$	$ww = w$
Involution	$\overline{\overline{w}} = w$	
Complement	$w + \overline{w} = 1$	$w\overline{w} = 0$
de Morgan	$\overline{w + y} = \overline{w}\,\overline{y}$	$\overline{wy} = \overline{w} + \overline{y}$
	$1 + w = 1$	$0 \cdot w = 0$
	$0 + w = w$	$1 \cdot w = w$

more than one other term in the factoring process. As an example of this, consider the function f_2 in Table A.1. The sum-of-products expression that can be derived for it directly from the truth table is

$$f_2 = \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 + \bar{x}_1x_2\bar{x}_3 + x_1\bar{x}_2\bar{x}_3 + x_1\bar{x}_2x_3$$

By repeating the first product term $\bar{x}_1\bar{x}_2\bar{x}_3$ and interchanging the order of terms (by the commutative rule), we obtain

$$f_2 = \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 + x_1\bar{x}_2\bar{x}_3 + x_1\bar{x}_2x_3 + \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1x_2\bar{x}_3$$

Grouping the terms in pairs and factoring yields

$$\begin{aligned} f_2 &= \bar{x}_1\bar{x}_2(\bar{x}_3 + x_3) + x_1\bar{x}_2(\bar{x}_3 + x_3) + \bar{x}_1(\bar{x}_2 + x_2)\bar{x}_3 \\ &= \bar{x}_1\bar{x}_2 + x_1\bar{x}_2 + \bar{x}_1\bar{x}_3 \end{aligned}$$

The first pair of terms is again reduced by factoring, and we obtain the minimal expression

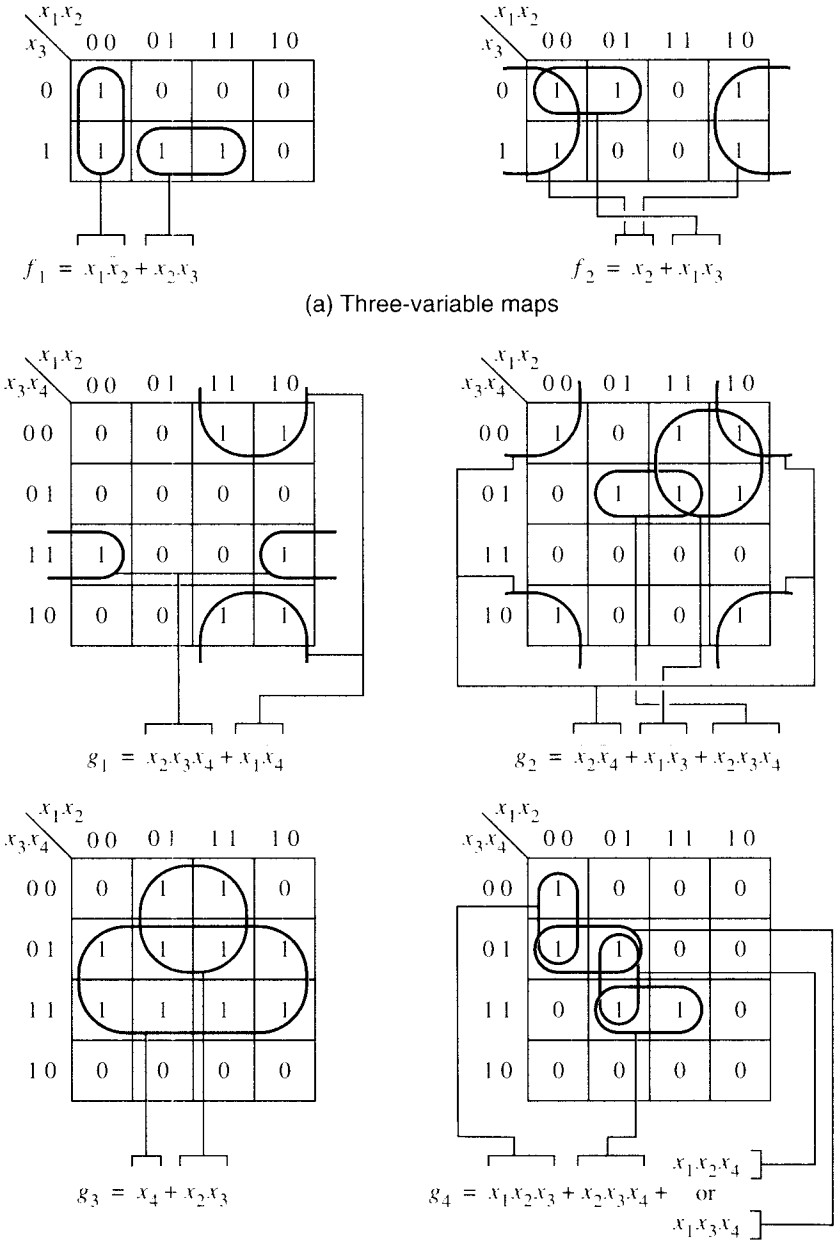
$$f_2 = \bar{x}_2 + \bar{x}_1\bar{x}_3$$

This completes our discussion of algebraic simplification of logic expressions. The obvious practical application of this mathematical exercise stems from the fact that networks with fewer gates and inputs are cheaper and easier to implement. Therefore, it is of economic interest to be able to determine the minimal expression that is equivalent to a given expression. The rules that we have used in manipulating logic expressions are summarized in Table A.4. They are arranged in pairs to show their symmetry as they apply to both the AND and OR functions. So far, we have not had occasion to use either involution or de Morgan's rules, but they will be found to be useful in the next section.

A.3.1 MINIMIZATION USING KARNAUGH MAPS

In our algebraic minimization of the functions f_1 and f_2 of Table A.1, it was necessary to guess the best way to proceed at certain points. For instance, the decision to repeat the term $\bar{x}_1\bar{x}_2\bar{x}_3$ as the first step in minimizing f_2 is not obvious. There is a geometric technique that can be used to quickly derive the minimal expression for a logic function of a few variables. The technique depends on a different form for presentation of the truth table, a form called the *Karnaugh map*. For a three-variable function, the map is a rectangle composed of eight squares arranged in two rows of four squares each, as shown in Figure A.5*a*. Each square of the map corresponds to a particular valuation of the input variables. For example, the third square of the top row represents the valuation $(x_1, x_2, x_3) = (1, 1, 0)$. Because there are eight rows in a three-variable truth table, the map obviously requires eight squares. The entries in the squares are the function values for the corresponding input valuations.

The key idea in the formation of the map is that horizontally and vertically adjacent squares correspond to input valuations that differ in only one variable. When two adjacent squares contain 1s, they indicate the possibility of an algebraic simplification.



(a) Three-variable maps

(b) Four-variable maps

Figure A.5 Minimization using Karnaugh maps.

In the map for f_2 in Figure A.5*a*, the 1 values in the leftmost two squares of the top row correspond to the product terms $\overline{x}_1\overline{x}_2\overline{x}_3$ and $\overline{x}_1x_2\overline{x}_3$. The simplification

$$\overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_1x_2\overline{x}_3 = \overline{x}_1\overline{x}_3$$

was performed earlier in minimizing the algebraic expression for f_2. This simplification can be obtained directly from the map by grouping the two 1s as shown. The product term that corresponds to a group of squares is the product of the input variables whose values are constant on these squares. If the value of input variable x_i is 0 for all 1s of a group, then $\overline{x}_i$ is entered in the product, but if x_i has the value 1 for all 1s of the group, then x_i is entered in the product. Adjacency of two squares includes the property that the left-end squares are adjacent to the right-end squares. Continuing with our discussion of f_2, the group of four 1s consisting of the left-end column and the right-end column simplifies to the single-variable term $\overline{x}_2$ because x_2 is the only variable whose value remains constant over the group. All four possible combinations of values of the other two variables occur in the group.

Karnaugh maps can be used for more than three variables. A Karnaugh map for four variables can be obtained from two 3-variable maps. Examples of four-variable maps are shown in Figure A.5*b*, along with minimal expressions for the functions represented by the maps. In addition to two- and four-square groupings, it is now possible to form eight-square groupings. Such a grouping is illustrated in the map for g_3. Note that the four corner squares constitute a valid group of four and are represented by the product term $\overline{x}_2\overline{x}_4$ in g_2. As in the case of three-variable maps, the term that corresponds to a group of squares is the product of the variables whose values do not change over the group. For example, the grouping of four 1s in the upper right-hand corner of the map for g_2 is represented by the product term $x_1\overline{x}_3$ because $x_1 = 1$ and $x_3 = 0$ over the group. The variables x_2 and x_4 have all the possible combinations of values over this group. It is also possible to use Karnaugh maps for five-variable functions. In this case, two 4-variable maps are used, one of them corresponding to the 0 value for the fifth variable and the other corresponding to the 1 value.

The general procedure for forming groups of two, four, eight, and so on in Karnaugh maps is readily derived. Two adjacent pairs of 1s can be combined to form a group of four. Similarly, two adjacent groups of four can be combined to form a group of eight. In general, the number of squares in any valid group must be equal to 2^k, where k is an integer.

We will now consider a procedure for using Karnaugh maps to obtain minimal sum-of-products expressions. As can be seen in the maps of Figure A.5, a large group of 1s corresponds to a small product term. Thus, a simple gate implementation results from covering all the 1s in the map with as few groups as possible. In general, we should choose the smallest set of groups, picking large ones wherever possible, that cover all the 1s in the map. Consider, for example, the function g_2 in Figure A.5*b*. As we have already seen, the 1s in the four corners constitute a group of four that is represented by the product term $\overline{x}_2\overline{x}_4$. Another group of four exists in the upper right-hand corner and is represented by the term $x_1\overline{x}_3$. This covers all the 1s in the map except for the 1 in the square where $(x_1, x_2, x_3, x_4) = (0, 1, 0, 1)$. The largest group of 1s that includes this square is the two-square group represented by the term $x_2\overline{x}_3x_4$. Therefore, the minimal

expression for g_2 is

$$g_2 = \overline{x}_2\overline{x}_4 + x_1\overline{x}_3 + x_2\overline{x}_3x_4$$

Minimal expressions for the other functions shown in the figure can be derived in a similar manner. Note that in the case of g_4 there are two possible minimal expressions, one including the term $\overline{x}_1x_2x_4$ and the other including the term $\overline{x}_1\overline{x}_3x_4$. It is often the case that a given function has more than one minimal expression.

In all our examples, it is relatively easy to derive minimal expressions. In general, there are formal algorithms for this process, but we will not consider them here.

A.3.2 DON'T-CARE CONDITIONS

In many situations, some valuations of the inputs to a digital circuit never occur. For example, consider the binary-coded decimal (BCD) number representation. Four binary variables b_3, b_2, b_1, and b_0 represent the decimal digits 0 through 9, as shown in Figure A.6. These four variables have a total of 16 distinct valuations, only 10 of which are used for representing the decimal digits. The remaining valuations are not used. Therefore, any logic circuit that processes BCD data will never encounter any of these six valuations at its inputs.

Figure A.6 gives the truth table for a particular function that may be performed on a BCD digit. We do not care what the function values are for the unused input valuations; hence, they are called *don't-cares* and are denoted as such by the letter "d" in the truth table. To obtain a circuit implementation, the function values corresponding to don't-care conditions can be arbitrarily assigned to be either 0 or 1. The best way to assign them is in a manner that will lead to a minimal logic gate implementation. We should interpret don't-cares as 1s whenever they can be used to enlarge a group of 1s. Because larger groups correspond to smaller product terms, minimization is enhanced by the judicious inclusion of don't-care entries.

The function in Figure A.6 represents the following processing on a decimal digit input: The output f is to have the value 1 whenever the inputs represent a nonzero digit that is evenly divisible by 3. Three groups are necessary to cover the three 1s of the map, and don't-cares have been used to enlarge these groups as much as possible.

A.4 SYNTHESIS WITH NAND AND NOR GATES

We will now consider two other basic logic gates called NAND and NOR, which are extensively used in practice because of their simple electronic realizations. The truth table for these gates is shown in Figure A.7. They implement the equivalent of the AND and OR functions followed by the NOT function, which is the motivation for the names and standard logic symbols for these gates. Letting the arrows "↑" and "↓" denote the NAND and NOR operators, respectively, and using de Morgan's rule in Table A.4,

Decimal digit represented	Binary coding $b_3\ b_2\ b_1\ b_0$	f
0	0 0 0 0	0
1	0 0 0 1	0
2	0 0 1 0	0
3	0 0 1 1	1
4	0 1 0 0	0
5	0 1 0 1	0
6	0 1 1 0	1
7	0 1 1 1	0
8	1 0 0 0	0
9	1 0 0 1	1
unused	1 0 1 0	d
	1 0 1 1	d
	1 1 0 0	d
	1 1 0 1	d
	1 1 1 0	d
	1 1 1 1	d

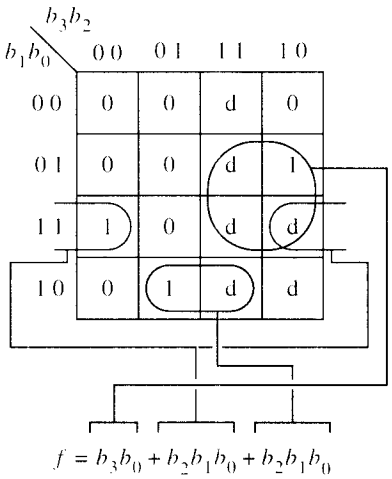


Figure A.6 Four-variable Karnaugh map illustrating don't cares.

we have

$$x_1 \uparrow x_2 = \overline{x_1 x_2} = \overline{x}_1 + \overline{x}_2$$

and

$$x_1 \downarrow x_2 = \overline{x_1 + x_2} = \overline{x}_1 \overline{x}_2$$

NAND and NOR gates with more than two input variables are available, and they

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

$f = x_1 \uparrow x_2 = \overline{x_1 x_2} = \bar{x}_1 + x_2$

(a) NAND

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

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

(b) NOR

Figure A.7 NAND and NOR gates.

operate according to the obvious generalization of de Morgan's law as

$$x_1 \uparrow x_2 \uparrow \cdots \uparrow x_n = \overline{x_1 x_2 \cdots x_n} = \bar{x}_1 + \bar{x}_2 + \cdots + \bar{x}_n$$

and

$$x_1 \downarrow x_2 \downarrow \cdots \downarrow x_n = \overline{x_1 + x_2 + \cdots + x_n} = \bar{x}_1 \bar{x}_2 \cdots \bar{x}_n$$

Logic design with NAND and NOR gates is not as straightforward as with AND, OR, and NOT gates. One of the main difficulties in the design process is that the associative rule is not valid for NAND and NOR operations. We will expand on this problem later. First, however, let us describe a simple, general procedure for synthesizing any logic function using only NAND gates. There is a direct way to translate a logic network expressed in sum-of-products form into an equivalent network composed only of NAND gates. The procedure is easily illustrated with the aid of an example. Consider the following algebraic manipulation of a logic expression corresponding to a four-input network composed of three 2-input NAND gates:

$$\begin{aligned}(x_1 \uparrow x_2) \uparrow (x_3 \uparrow x_4) &= \overline{\overline{(x_1 x_2)(x_3 x_4)}} \\ &= \overline{\overline{x_1 x_2}} + \overline{\overline{x_3 x_4}} \\ &= x_1 x_2 + x_3 x_4\end{aligned}$$

We have used de Morgan's rule and the involution rule in this derivation. Figure A.8 shows the logic network equivalent of this derivation. Since any logic function can be synthesized in a sum-of-products (AND-OR) form and because the preceding derivation is obviously reversible, we have the result that any logic function can be synthesized in NAND-NAND form. We can see that this result is true for functions of any number of variables. The required number of inputs to the NAND gates is obviously the same as the number of inputs to the corresponding AND and OR gates.

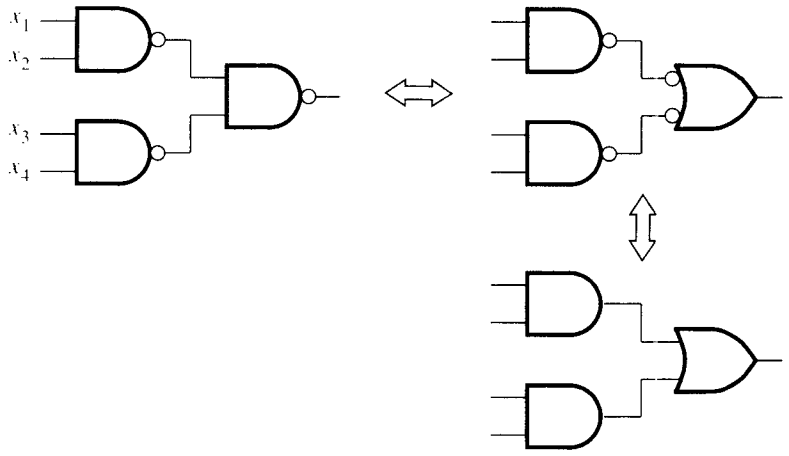


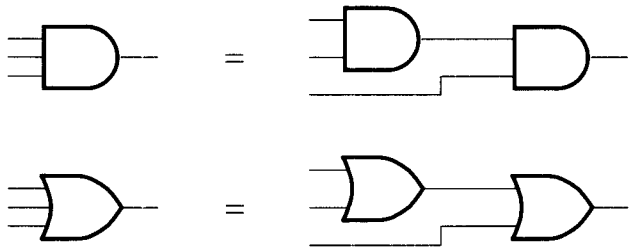
Figure A.8 Equivalence of NAND-NAND and AND-OR networks.

Let us return to the comment that the nonassociativity of the NAND operator can be an annoyance. In designing logic networks with NAND gates using the procedure illustrated in Figure A.8, a requirement for a NAND gate with more inputs than can be found on standard commercially available gates may arise. If this happens when one is using AND and OR gates, there is no problem because the AND and OR operators are associative, and a straightforward cascade of limited fan-in gates can be used. The case of implementing three-input AND and OR functions with two-input gates is shown in Figure A.9*a*. The solution is not as simple in the case of NAND gates. For example, a three-input NAND function cannot be implemented by a cascade of 2 two-input NAND gates. Three gates are needed, as shown in Figure A.9*b*.

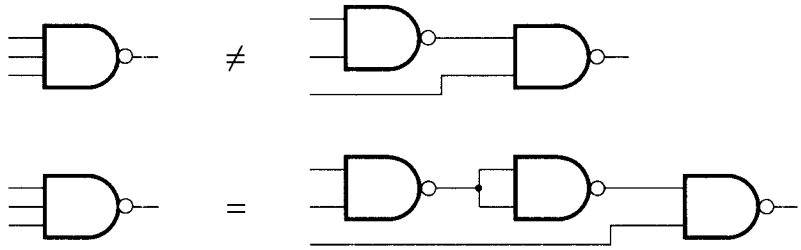
A discussion of the implementation of logic functions using only NOR gates proceeds in a similar manner. Any logic function can be synthesized in a product-of-sums (OR-AND) form. Such networks can be implemented by equivalent NOR-NOR networks.

The preceding discussion introduced some basic concepts in logic design. Detailed discussion of the subject can be found in any of a number of textbooks (see References 1, 3, 7–11).

It is important for the reader to appreciate that many different realizations of a given logic function are possible. For practical reasons, it is useful to find realizations that minimize the cost of implementation. It is also often necessary to minimize the propagation delay through a logic network. We introduced the concept of minimization in the previous sections to give an indication of the nature of logic synthesis and the reductions in cost that may be achieved. For example, Karnaugh maps graphically show the manipulation possibilities that lead to optimal solutions. Although it is important to understand the principles of optimization of logic networks, it is not necessary to do the optimization by hand. Sophisticated *computer-aided design* (CAD) programs exist for such synthesis. The designer needs to specify only the desired functional behavior, and the CAD software generates a cost-effective network that implements the required functionality.



(a) Implementing three-input AND and OR functions with two-input gates



(b) Implementing a three-input NAND function with two-input gates

Figure A.9 Cascading of gates.

A.5 PRACTICAL IMPLEMENTATION OF LOGIC GATES

Let us now turn our attention to the means by which logic variables can be represented and logic functions can be implemented in practice. The choice of a physical parameter to represent logic variables is obviously technology-dependent. In electronic circuits, either voltage or current levels can be used for this purpose.

To establish a correspondence between voltage levels and logic values or states, the concept of a *threshold* is used. Voltages above a given threshold are taken to represent one logic value, with voltages below that threshold representing the other. In practical situations, the voltage at any point in an electronic circuit undergoes small random variations for a variety of reasons. Because of this "noise," the logic state corresponding to a voltage level near the threshold cannot be reliably determined. To avoid such ambiguity, a "forbidden range" should be established, as shown in Figure A.10. In this case, voltages below $V_{0,max}$ represent the 0 value, and voltages above $V_{1,min}$ represent the 1 value. In subsequent discussion, we will often use the terms "low" and "high" to represent the voltage levels corresponding to logic values 0 and 1, respectively.

We will begin our discussion of electronic circuits that implement basic logic functions by considering simple circuits consisting of resistors and transistors that act as switches. Consider the circuits in Figure A.11. When switch S in Figure A.11*a* is closed, the output voltage V_{out} is equal to 0 (ground). When S is open, the output

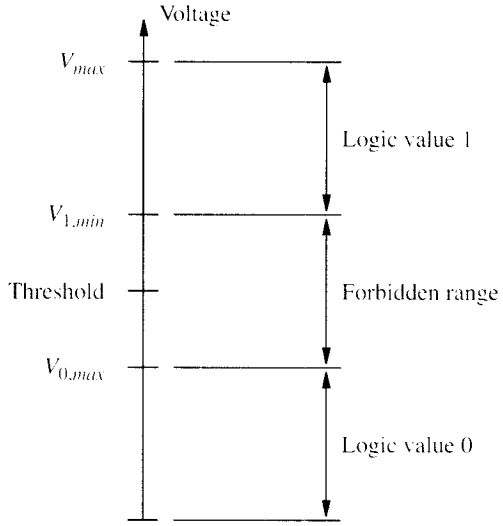


Figure A.10 Representation of logic values by voltage levels.

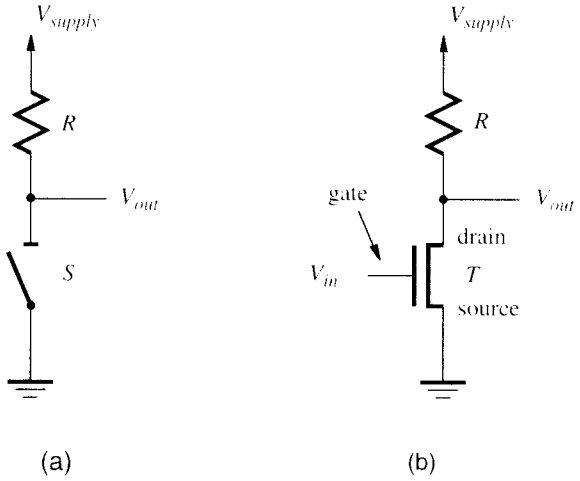


Figure A.11 An inverter circuit.

voltage V_{out} is equal to the supply voltage, V_{supply}. The same effect can be obtained in Figure A.11*b*, in which a transistor T is used to replace the switch S. When the input voltage applied to the gate of the transistor is 0 (that is, when $V_{in} = 0$), the transistor is equivalent to an open switch, and $V_{out} = V_{supply}$. When V_{in} changes to V_{supply}, the transistor acts as a closed switch and the output voltage V_{out} is very close to 0. Thus, the circuit performs the function of a logic NOT gate.

We can now discuss the implementation of more complex logic functions. Figure A.12 shows a circuit realization for a NOR gate. In this case, V_{out} in Figure A.12*a* is high only when both switches S_a and S_b are open. Similarly, V_{out} in Figure A.12*b* is high only when both inputs V_a and V_b are low. Thus, the circuit is equivalent to a NOR gate in which V_a and V_b correspond to two logic variables x_1 and x_2. We can easily verify that a NAND gate can be obtained by connecting the transistors in series as shown in Figure A.13. The logic functions AND and OR can be implemented using NAND and NOR gates, respectively, followed by the inverter of Figure A.11.

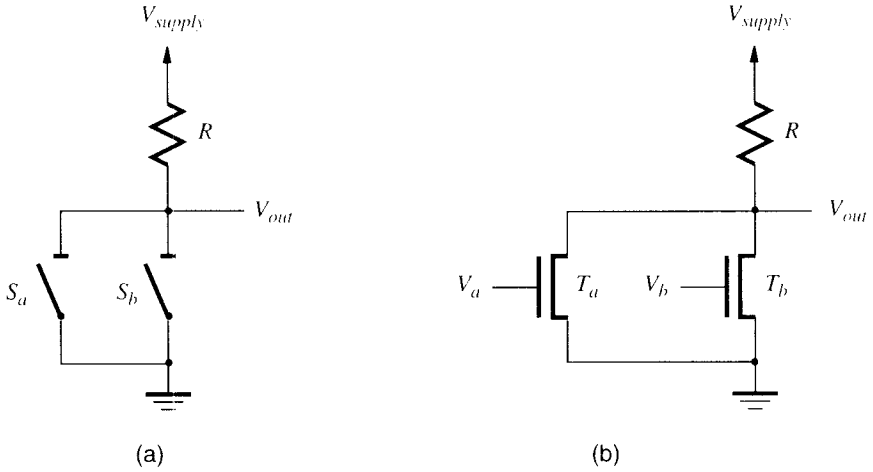


Figure A.12 A transistor circuit implementation of a NOR gate.

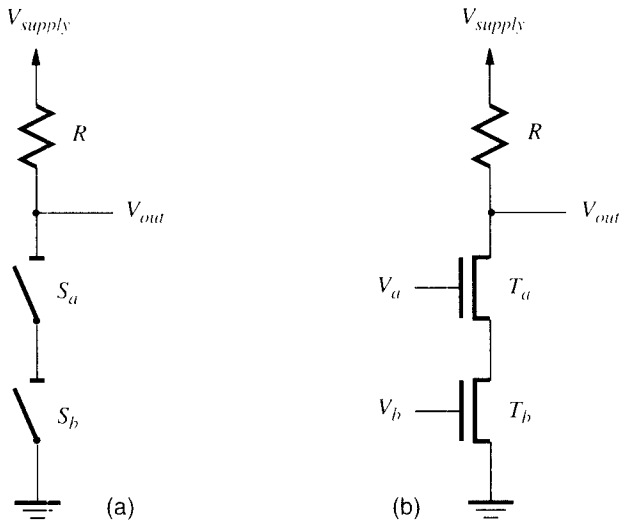
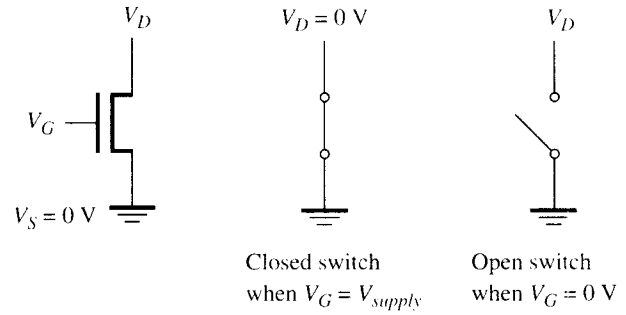


Figure A.13 A transistor circuit implementation of a NAND gate.

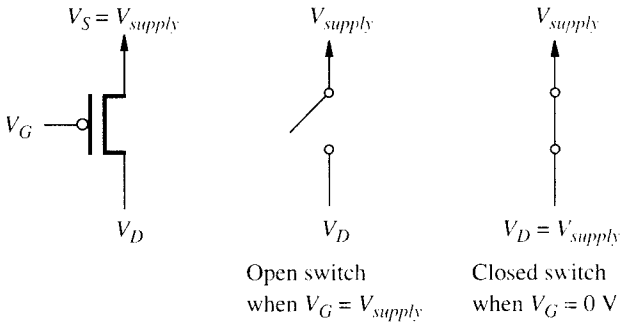
Note that NAND and NOR gates are simpler in their circuit implementations than AND and OR gates. Hence, it is not surprising to find that practical realizations of logic functions use NAND and NOR gates extensively. Many of the examples given in this book show circuits consisting of AND, OR, and NOT gates for ease of understanding. In practice, logic circuits contain all five types of gates.

A.5.1 CMOS CIRCUITS

Figures A.11 through A.13 illustrate the general structure of circuits implemented using *NMOS technology*. The name derives from the fact that the transistors used to realize the logic functions are of NMOS type. Two types of *metal-oxide semiconductor* (MOS) transistors are available for use as switches. An n-channel transistor is said to be of NMOS-type, and it behaves as a closed switch when its gate input is raised to the positive power supply voltage, V_{supply}, as indicated in Figure A.14*a*. The opposite behavior is achieved with a p-channel transistor, which is said to be of PMOS type. It acts as an open switch when the gate voltage, V_G, is equal to V_{supply}, and as a closed switch



(a) NMOS transistor



(b) PMOS transistor

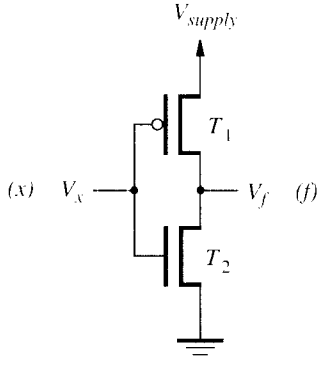
Figure A.14 NMOS and PMOS transistors in logic circuits.

when $V_G = 0$, as indicated in Figure A.14*b*. Note that the graphical symbol for a PMOS transistor has a bubble on the gate input to indicate that its behavior is complementary to that of an NMOS transistor. Note also that the names source and drain are associated with the opposite terminals of PMOS transistors in comparison with NMOS transistors. The source of an NMOS transistor is connected to ground, while the source of a PMOS transistor is connected to V_{supply}. This naming convention is due to the nature of the current that flows through these transistors.

A drawback of the circuits in Figures A.11 through A.13 is their power consumption. In the state in which the switches are closed to provide a path between ground and the pull-up resistor R, there is current flowing from V_{supply} to ground. In the opposite state, in which switches are open, there is no path to ground and there is no current flowing. (In MOS transistors no current flows through the gate terminal.) Thus, depending on the states of its gates, there may be significant power consumption in a logic circuit.

An effective solution to the power consumption problem lies in using both NMOS and PMOS transistors to implement circuits that do not dissipate power when in a steady state. This approach leads to the CMOS (complementary metal-oxide semiconductor) technology. The basic idea of CMOS circuits is illustrated by the inverter circuit in Figure A.15. When $V_x = V_{supply}$, which corresponds to the input x having the logic value 1, transistor T_1 is turned off and T_2 is turned on. Thus, T_2 pulls the output voltage V_f down to 0. When V_x changes to 0, transistor T_1 turns on and T_2 turns off. Thus, T_1 pulls the output voltage V_f up to V_{supply}. Therefore, the logic values of x and f are complements of each other, and the circuit implements a NOT gate.

A key feature of this circuit is that transistors T_1 and T_2 operate in a complementary fashion; when one is on, the other is off. Hence, there is always a closed path from the output point f to either V_{supply} or ground. There is no closed path between V_{supply} and ground at any time except during a very short transition period when the transistors are changing their states. This means that the circuit does not dissipate appreciable power when it is in a steady state. It dissipates power only when it is switching from one logic



(a) Circuit

x	V_x	T_1	T_2	V_f	f
0	low	on	off	high	1
1	high	off	on	low	0

(b) Truth table and transistor states

Figure A.15 CMOS realization of a NOT gate.

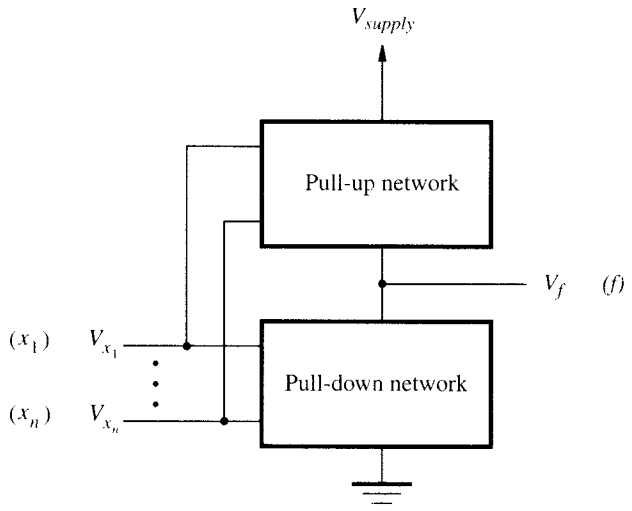


Figure A.16 Structure of a CMOS circuit.

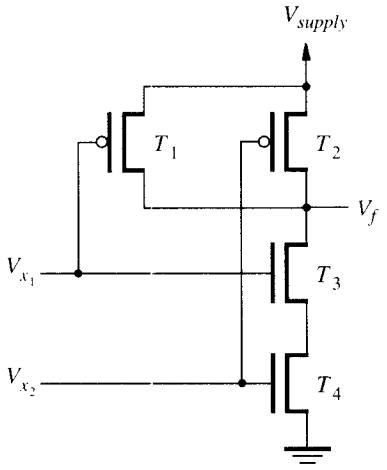
state to another. Therefore, power dissipation in this circuit is dependent on the rate at which state changes take place.

We can now extend the CMOS concept to circuits that have n inputs, as shown in Figure A.16. NMOS transistors are used to implement the pull-down network, such that a closed path is established between the output point f and ground when a desired function $F(x_1, \ldots, x_n)$ is equal to 0. The pull-up network is built with PMOS transistors, such that a closed path is established between the output f and V_{supply} when $F(x_1, \ldots, x_n)$ is equal to 1. The pull-up and pull-down networks are functional complements of each other, so that in steady state there exists a closed path only between the output f and either V_{supply} or ground, but not both.

The pull-down network is implemented in the same way as shown in Figures A.11 through A.13. Figure A.17 gives the implementation of a NAND gate, and Figure A.18 gives a NOR gate. Figure A.19 shows how an AND gate is realized by inverting the output of a NAND gate.

In addition to low power dissipation, CMOS circuits have the advantage that MOS transistors can be implemented in very small sizes and thus occupy a very small area on an integrated circuit chip. This results in two significant benefits. First, it is possible to fabricate chips containing millions of transistors, which has led to the realization of modern microprocessors and large memory chips. Second, the smaller the transistor, the faster it can be switched from one state to another. Thus, CMOS circuits can now be operated at speeds in the gigahertz range.

Different CMOS circuits have been developed to operate with power supply voltages in the range from 1.5 to 15 V. The most commonly used power supplies are 5 V and 3.3 V. Circuits that use lower power supply voltages dissipate much less power (power dissipation is proportional to V^2_{supply}), which means that more transistors can be placed on a chip without causing overheating. A drawback of lower power supply voltage is reduced noise immunity.

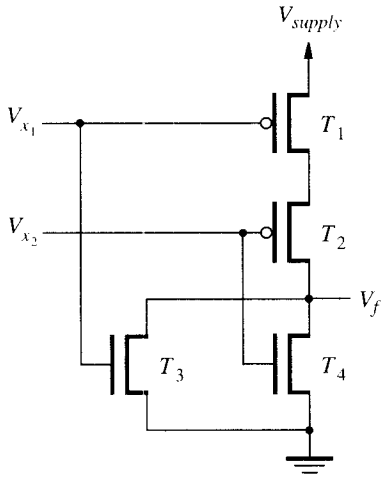


(a) Circuit

x_1	x_2	T_1	T_2	T_3	T_4	f
0	0	on	on	off	off	1
0	1	on	off	off	on	1
1	0	off	on	on	off	1
1	1	off	off	on	on	0

(b) Truth table and transistor states

Figure A.17 CMOS realization of a NAND gate.



(a) Circuit

x_1	x_2	T_1	T_2	T_3	T_4	f
0	0	on	on	off	off	1
0	1	on	off	off	on	0
1	0	off	on	on	off	0
1	1	off	off	on	on	0

(b) Truth table and transistor states

Figure A.18 CMOS realization of a NOR gate.

Transitions between low and high signal levels in a CMOS inverter are illustrated in more detail in Figure A.20. The blue curve, known as the *transfer characteristic*, shows the output voltage as a function of the input voltage. The curve indicates that a rather sharp transition in output voltage takes place when the input voltage passes through the value of about $V_{supply}/2$. There is a *threshold* voltage, V_t, and a small value δ such that

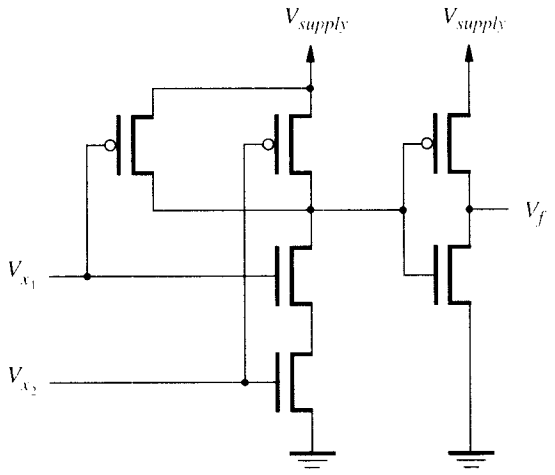


Figure A.19 CMOS realization of an AND gate.

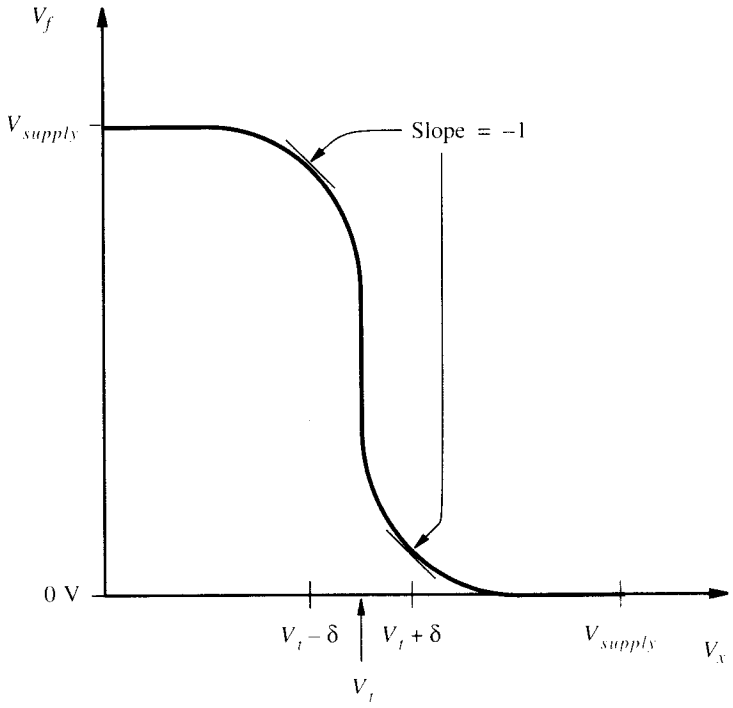


Figure A.20 The voltage transfer characteristic for the CMOS inverter.

$V_{out} \approx V_{supply}$ if $V_{in} < V_t - \delta$ and $V_{out} \approx 0$ if $V_{in} > V_t + \delta$. This means that the input signal need not be exactly equal to the nominal value of either 0 or V_{supply} to produce the correct output signal. There is room for some error, called *noise*, in the input signal that will not cause adverse effects. The amount of noise that can be tolerated is called the *noise margin*. This margin is $V_{supply} - (V_t + \delta)$ volts when the logic value of the input is 1, and it is $V_t - \delta$ when the logic value of the input is 0. CMOS circuits have excellent noise margins.

In this section, we have introduced the basic features of CMOS circuits. For a more detailed discussion of this technology the reader may consult References [1] and [8].

A.5.2 PROPAGATION DELAY

Logic circuits do not switch instantaneously from one state to another. Speed is measured by the rate at which state changes can take place. A related parameter is *propagation delay*, which is defined in Figure A.21. When a state change takes place at the input, a delay is encountered before the corresponding change at the output is observed. This propagation delay is usually measured between the 50-percent points of the transitions, as shown in the figure. Another important parameter is the *transition time*, which is normally measured between the 10- and 90-percent points of the signal swing, as shown. The maximum speed at which a logic circuit can be operated decreases as the

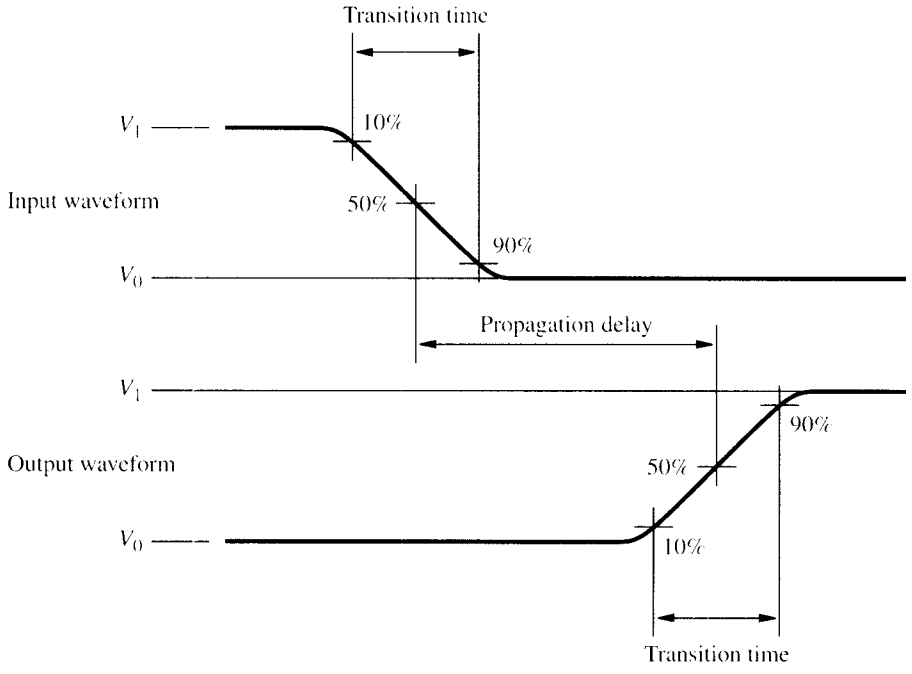


Figure A.21 Definition of propagation delay and transition time.

propagation delay through different paths within that circuit increases. The delay along any path in a logic circuit is the sum of individual gate delays along this path.

A.5.3 FAN-IN AND FAN-OUT CONSTRAINTS

The number of inputs to a logic gate is called its *fan-in*. The number of gate inputs that the output of a logic gate drives is called its *fan-out*. Practical circuits do not allow large fan-in and fan-out because they both have an adverse effect on the propagation delay and hence the speed of the circuit.

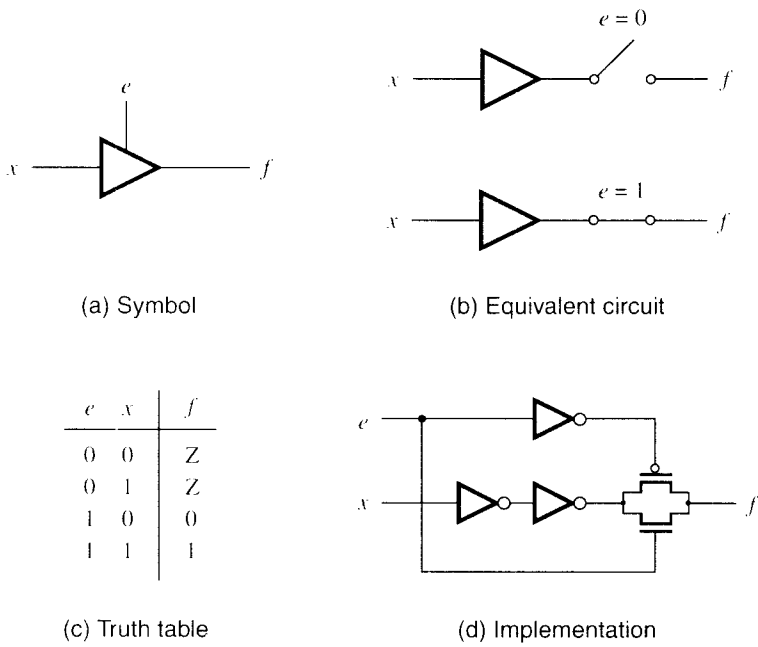
Each transistor in a CMOS gate contributes a certain amount of capacitance. As the capacitance increases, the circuit becomes slower and its signal levels and noise margins become worse. Therefore, it is necessary to limit the fan-in and fan-out, typically to a number less than ten. If the number of desired inputs exceeds the maximum fan-in, it is necessary to use an additional gate of the same type. Figure A.9*a* shows how two gates of the same type can be cascaded. If the number of outputs that have to be driven by a particular gate exceeds the acceptable fan-out, it is possible to use two copies of that gate.

A.5.4 TRI-STATE BUFFERS

In the logic gates discussed so far, it is not possible to connect the outputs of two gates together. This would make no sense from the logic point of view because if one gate generated an output value of 1 and the other an output of 0, it would be uncertain what the combined output signal would be. More importantly, in CMOS circuits, the gate that generates the output of 1 establishes a direct path from the output terminal to V_{supply}, while the gate that generates 0 establishes a path to ground. Thus, the two gates would provide a short circuit across the power supply, which would damage the gates.

Yet, in the design of computer systems, there are many cases where an input signal to a circuit may be derived from one of a number of different sources. This can be done using multiplexer logic circuits, which are discussed in Section A.10. It can also be done using special gates called *tri-state buffers*. A tri-state buffer has three states. Two of the states produce the normal 0 and 1 signals. The third state places the output terminal of the buffer into a high-impedance state in which the output is electrically disconnected from the input it is supposed to drive.

Figure A.22 depicts a tri-state buffer. The buffer has two inputs and one output. The *enable* input, e, controls the operation of the buffer. When $e = 1$, the output f has the same logic value as the input x. When $e = 0$, the output is placed in the high-impedance state, Z. An equivalent circuit is shown in Figure A.22*b*. The triangular symbol in this figure represents a noninverting driver. This is a circuit that performs no logic operation because its output merely replicates the input signal. Its purpose is to provide additional electrical driving capability. When combined with the output switch shown in the figure, it behaves according to the truth table given in Figure A.22*c*. This table describes the required tri-state behavior. Figure A.22*d* shows a circuit implementation of the tri-state buffer. One NMOS and one PMOS transistor are connected in parallel to implement the switch, which is connected to the output of the driver. Because the two transistor



(a) Symbol (b) Equivalent circuit

(c) Truth table (d) Implementation

Figure A.22 Tri-state buffer.

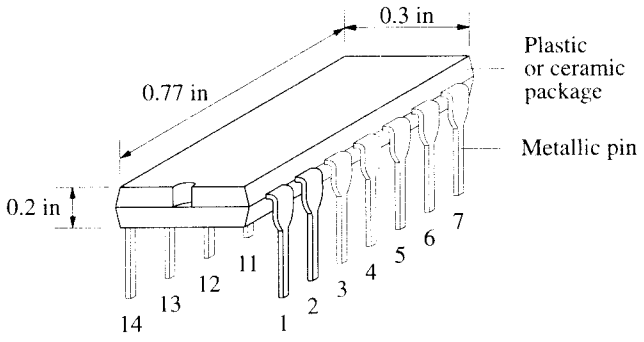
types require complementary control signals at their gate inputs, an inverter is used as shown. When $e = 0$, both transistors are turned off, resulting in an open switch. When $e = 1$, both transistors are turned on, resulting in a closed switch.

The driver circuit has to be able to drive a number of inputs of other gates whose combined capacitance may exceed the drive capability of an ordinary logic gate circuit. To provide a sufficient drive capability, the driver circuit needs larger transistors. Hence, the two cascaded NOT gates that realize the driver are implemented with transistors of larger size than in regular logic gates.

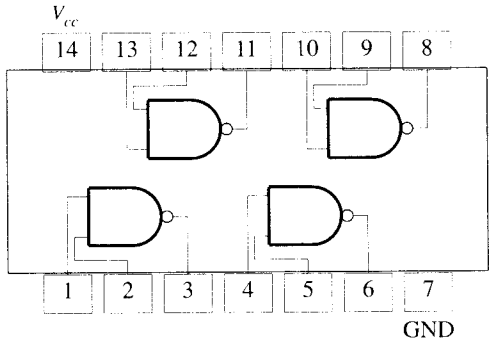
The reader may wonder why is it necessary to use the PMOS transistor in the output switch because from the logic function point of view the same behavior could be achieved using just the NMOS transistor. The reason is that these transistors have to "pass" the logic value generated by the driver circuit to the output f, and it turns out that NMOS transistors pass the logic value 0 well but the logic value 1 poorly, while PMOS transistors pass 1 well and 0 poorly. The parallel arrangement of NMOS and PMOS transistors passes both 1s and 0s well. For a more detailed discussion of this issue and tri-state buffers in general, the reader may consult Reference [1].

A.5.5 INTEGRATED CIRCUIT PACKAGES

The main features of electronic circuits used to implement logic functions were discussed in previous sections. In practical design, it is necessary to use integrated circuits (ICs) that are commercially available. When ICs became available in the 1960s, there



(a) Physical appearance



(b) Schematic of an integrated circuit providing four 2-input NAND gates

Figure A.23 A 14-pin integrated circuit package.

quickly developed a trend to provide logic gates in the form of standardized IC chips. An IC chip is mounted inside a sealed protective package with a number of metallic pins for external connections. Standard IC packages are available with different numbers of pins. A simple package containing four NAND gates is shown in Figure A.23. The four gates utilize common power supply and ground pins. Such ICs comprising only a few logic gates are referred to as *small-scale integrated* (SSI) *circuits.*

The SSI circuits provide too little functionality for the physical space that they require. Moreover, their performance is inferior because of the electrical characteristics of the pins on an IC package. In general, it is necessary to use larger transistors to provide the signals needed to drive external wires connected to pins. This increases both propagation delay and power dissipation.

A CMOS NAND gate provided as part of an IC package like the one illustrated in Figure A.23 may have a propagation delay of 5 nanoseconds. However, the delay of a NAND circuit inside a large CMOS network implemented on a single chip may be 0.2 ns or less, depending on the manufacturing technology used.

Much larger ICs are available today, and almost all logic circuits are realized with such chips. A chip may implement a useful functional block such as an adder, multiplier, register, encoder, or decoder. But it may also provide just an assortment of gates and programmable interconnection switches that can be configured by the designer to realize a variety of arbitrary functions. In subsequent sections, we will discuss some commonly used functional blocks as well as general user-programmable logic devices.

A.6 FLIP-FLOPS

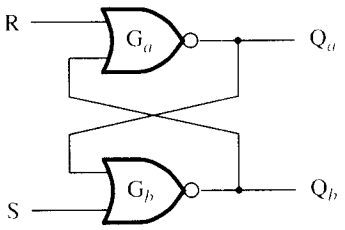
The majority of applications of digital logic require the storage of information. For example, a circuit that controls a combination lock must remember the sequence in which the digits are dialed in order to determine whether to open the lock. Another important example is the storage of programs and data in the memory of a digital computer.

The basic electronic element for storing binary information is termed a *latch*. Consider the two cross-coupled NOR gates in Figure A.24*a*. Let us examine this circuit, starting with the situation in which R = 1 and S = 0. Simple analysis shows that $Q_a = 0$ and $Q_b = 1$. Under this condition, both inputs to gate G_a are equal to 1. Thus, if R is changed to 0, no change will take place at the outputs Q_a and Q_b. If S is set to 1 with R equal to 0, Q_a and Q_b will become 1 and 0, respectively, and will remain in this state after S is returned to 0. Hence, this logic circuit constitutes a memory element, or a latch, that remembers which of the two inputs S and R was most recently equal to 1. A truth table for this latch is given in Figure A.24*b*. Some typical waveforms that characterize the latch are shown in Figure A.24*c*. The arrows in Figure A.24*c* indicate the cause-effect relationships among the signals. Note that when the R and S inputs change from 1 to 0 at the same time, the resulting state is undefined. In practice, the latch will assume one of its two stable states at random. The input valuation R = S = 1 is not used in most applications of such latches.

Because of the nature of the operation of the preceding circuit, the S and R lines are referred to as the *set* and *reset* inputs. Since the valuation R = S = 1 is normally not used, the Q_a and Q_b are usually represented by Q and $\overline{Q}$, respectively. However, $\overline{Q}$ should be regarded merely as a symbol representing the second output of the latch rather than as the complement of Q, because the input valuation R = S = 1 yields $Q = \overline{Q} = 0$.

A.6.1 GATED LATCHES

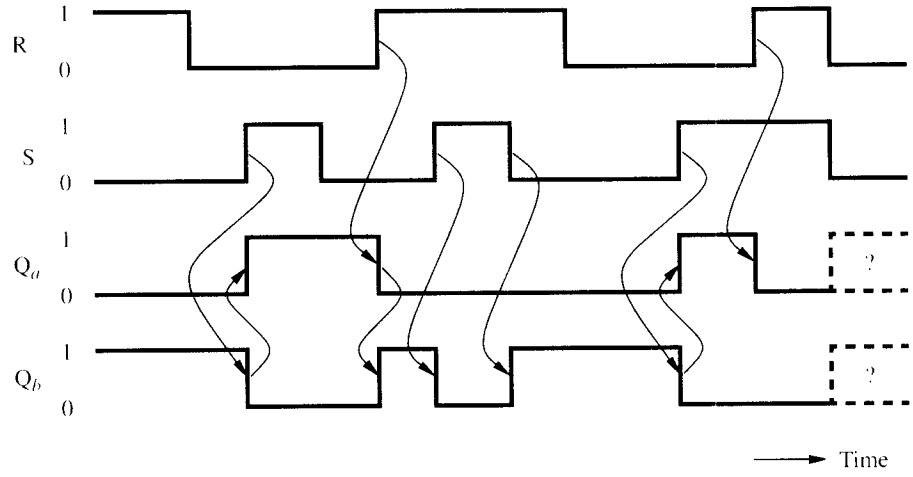
Many applications require that the time at which a latch is set or reset be controlled from an input other than R and S, termed a *clock* input. The resulting configuration is called a *gated SR latch*. A logic circuit, truth table, characteristic waveforms, and a graphical symbol for such a latch are given in Figure A.25. When the clock, *Clk*, is equal to 1, points S′ and R′ follow the inputs S and R, respectively. On the other hand, when *Clk* = 0, the S′ and R′ points are equal to 0, and no change in the state of the latch can take place.



(a) Network

S	R	Q_a	Q_b	
0	0	0/1	1/0	(No change)
0	1	0	1	
1	0	1	0	
1	1	0	0	

(b) Truth table

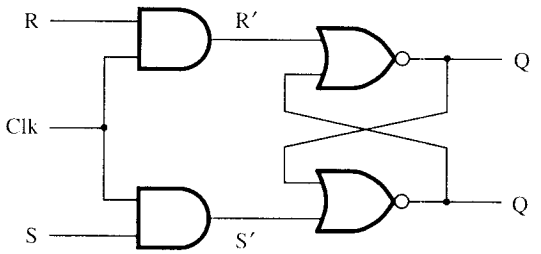


(c) Timing diagram

Figure A.24 A basic latch implemented with NOR gates.

So far we have used truth tables to describe the behavior of logic circuits. A truth table gives the output of a network for various input valuations. Logic circuits whose outputs are uniquely defined for each input valuation are referred to as *combinational circuits*. This is the class of circuits discussed in Sections A.1 to A.4. When memory elements are present, a different class of circuits is obtained. The output of such circuits is a function not only of the present valuation of the input variables but also of their previous behavior. An example of this is shown in Figure A.24. Circuits of this type are called *sequential circuits*.

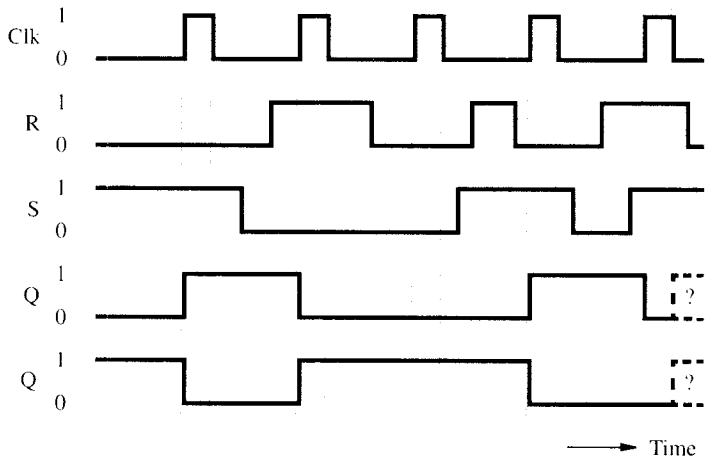
Because of the memory property, the truth table for the latch has to be modified to show the effect of its present state. Figure A.25*b* describes the behavior of the gated SR latch, where $Q(t)$ denotes its present state. The transition to the next state, $Q(t + 1)$, occurs following a clock pulse. Note that for the input valuation $S = R = 1$, $Q(t + 1)$ is undefined for reasons discussed earlier.



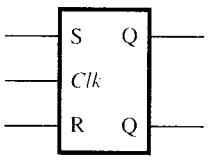
(a) Circuit

Clk	S	R	Q(t + 1)
0	x	x	Q(t) (no change)
1	0	0	Q(t) (no change)
1	0	1	0
1	1	0	1
1	1	1	x

(b) Truth table



(c) Timing diagram



(d) Graphical symbol

Figure A.25 Gated SR latch.

The gated SR latch can be implemented using NAND gates as shown in Figure A.26. It is a useful exercise to show that this circuit is functionally equivalent to the circuit in Figure A.25*a* (see Problem A.20).

A second type of gated latch, called the *gated D latch*, is shown in Figure A.27. In this case, the two signals S and R are derived from a single input D. At a clock pulse,

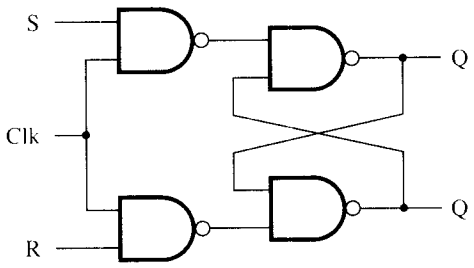
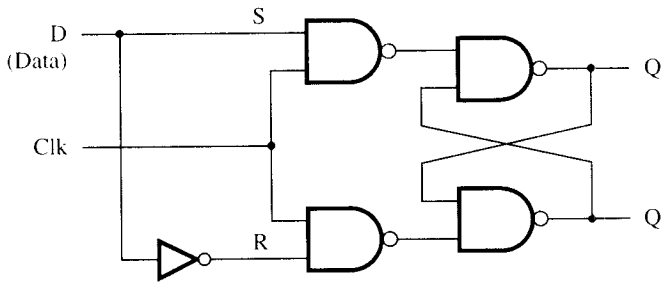


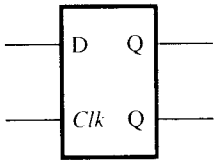
Figure A.26 Gated SR latch implemented with NAND gates.



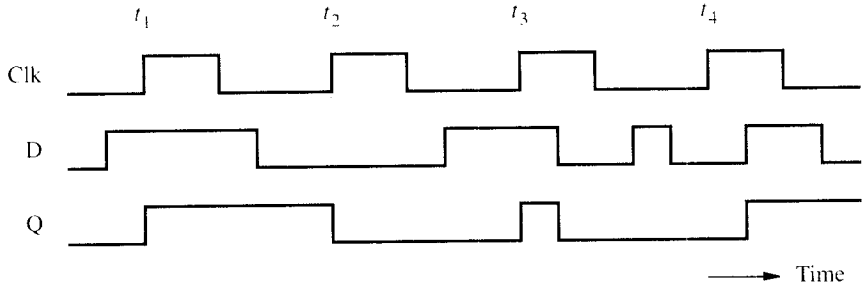
(a) Circuit

Clk	D	Q(t + 1)
0	x	Q(t)
1	0	0
1	1	1

(b) Truth table



(c) Graphical symbol



(d) Timing diagram

Figure A.27 Gated D latch.

the Q output is set to 1 if D = 1 or is reset to 0 if D = 0. This means that the D flip-flop samples the D input at the time the clock is high and stores that information until a subsequent clock pulse arrives.

A.6.2 MASTER-SLAVE FLIP-FLOP

In the circuit of Figure A.25, we assumed that while $Clk = 1$, the inputs S and R do not change. Inspection of the circuit reveals that the outputs will respond immediately to any change in the S or R input during this time. Similarly, for the circuit of Figure A.27, Q = D while $Clk = 1$. This is undesirable in many cases, particularly in circuits involving counters and shift registers, which will be discussed later. In such circuits, immediate propagation of logic conditions from the data inputs (R, S, and D) to the latch outputs may lead to incorrect operation. The concept of a *master-slave* organization eliminates this problem. Two gated D latches can be connected to form a *master-slave D flip-flop*, as shown in Figure A.28*a*. The first, referred to as the master, is connected to the input line D when Clock = 1. A 1-to-0 transition of the clock isolates the master from the input and transfers the contents of the master stage to the slave stage. We can see that no direct path ever exists from the input D to the output Q.

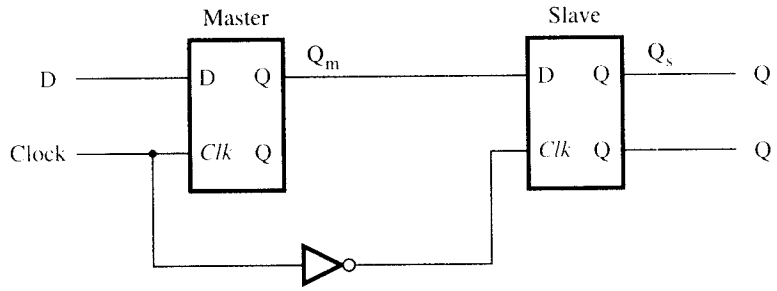
It should be noted that while Clock = 1, the state of the master stage is immediately affected by changes in the input D. The function of the slave stage is to hold the value at the output of the flip-flop while the master stage is being set up to the next-state value determined by the D input. The new state is transferred from the master to the slave after the 1-to-0 transition on Clock. At this point, the master stage is isolated from the inputs so that further changes in the D input will not affect this transfer. Examples of state transitions are shown in the form of a timing diagram in Figure A.28*b*.

The term *flip-flop* refers to a storage element that changes its output state at the edge of a controlling clock signal. In the above master-slave D flip-flop, the observable change takes place at the negative (1-to-0) edge of the clock. The change is observable when it reaches the Q terminal of the slave stage. Note that in the circuit in Figure A.28 we could have used the complement of Clock signal to control the master stage and the uncomplemented Clock to control the slave stage. In that case, the changes in the flip-flop output Q would occur at the positive edge of the clock.

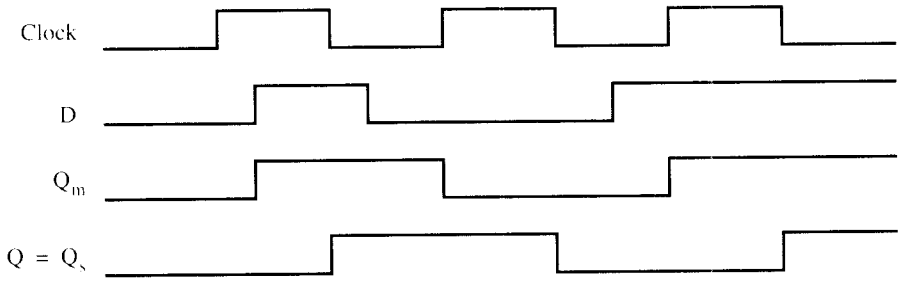
A graphical symbol for a flip-flop is given in Figure A.28*c*. We have used an arrowhead, instead of the label *Clk*, to denote the clock input to the flip-flop. This is a standard way of denoting that the positive edge of the clock causes changes in the state of the flip-flop. In our figure it is the negative edge which causes changes, so a small circle is used (in addition to the arrowhead) on the clock input.

A.6.3 EDGE TRIGGERING

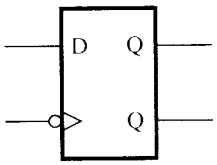
A flip-flop is said to be *edge triggered* if data present at the input are transferred to the output only at a transition in the clock signal. The input and output are isolated from each other at all other times. The terms *positive (leading) edge triggered* and *negative (trailing) edge triggered* describe flip-flops in which data transfer takes place at the



(a) Circuit



(b) Timing diagram

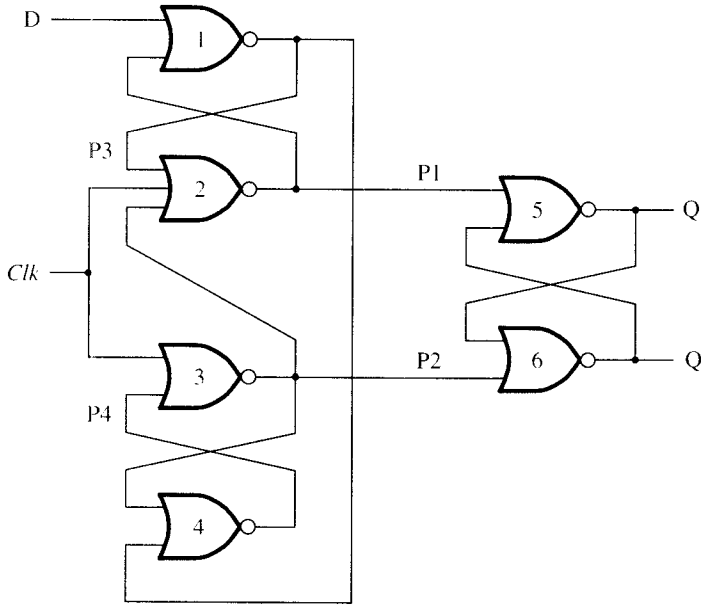


(c) Graphical symbol

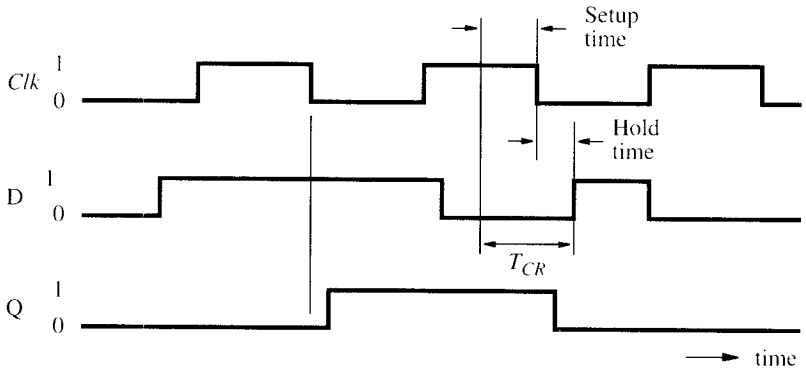
Figure A.28 Master-slave D flip-flop.

0-to-1 and the 1-to-0 clock transitions, respectively. For proper operation, edge-triggered flip-flops require the triggering edge of the clock pulse to be well defined and to have a very short transition time. The master-slave flip-flop in Figure A.28 is negative-edge triggered.

A different implementation for a negative edge-triggered D flip-flop is given in Figure A.29*a*. Let us consider the operation of this flip-flop. If $Clk = 1$, the outputs



(a) Network



(b) Example of timing

Figure A.29 A negative edge-triggered D flip-flop.

of gates 2 and 3 are both 0. Therefore, the flip-flop outputs Q and $\overline{Q}$ maintain the current state of the flip-flop. It is easy to verify that during this period, points P3 and P4 immediately respond to changes at D. Point P3 is kept equal to $\overline{D}$, and P4 is maintained equal to D. When *Clk* drops to 0, these values are transmitted to P1 and P2 by gates 2 and 3, respectively. Thus, the output latch, consisting of gates 5 and 6, acquires the new state to be stored.

We now verify that while $Clk = 0$, further changes at D do not change points P1 and P2. Consider two cases. First, suppose D = 0 at the negative edge of *Clk*. The 1 at P2 maintains an input of 1 at each of the gates 2 and 4, holding P1 and P2 at 0 and 1, respectively, independent of further changes in D. Second, suppose D = 1 at the negative edge of *Clk*. The 1 at P1 means that further changes at D cannot affect the output of gate 1, which is maintained at 0.

When *Clk* goes to 1 at the start of the next clock pulse, points P1 and P2 are again forced to 0, isolating the output from the remainder of the circuit. Points P3 and P4 then follow changes at D, as we have previously described.

An example of the operation of this type of D flip-flop is shown in Figure A.29*b*. The state acquired by the flip-flop upon the 1 to 0 transition of *Clk* is equal to the value on the D input immediately preceding this transition. However, there is a critical time period T_{CR} around the negative edge of *Clk* during which the value on D should not change. This region is split into two parts, the *setup time* before the clock edge and the *hold time* after the clock edge, as shown in the figure. The timing diagram shows that the output Q changes slightly after the negative edge of the clock. This is the effect of the propagation delay through the NOR gates.

A.6.4 T FLIP-FLOP

The most commonly used flip-flops are the D flip-flops because they are useful for temporary storage of data. However, there are applications for which other types of flip-flops are convenient. Counter circuits, discussed in Section A.8, are implemented efficiently using T flip-flops. A *T flip-flop* changes its state every clock cycle if its input T is equal to 1. We say that it "toggles" its state.

Figure A.30 presents the T flip-flop. Its circuit is derived from a D flip-flop as shown in Figure A.30*a*. Its truth table, graphical symbol, and a timing diagram example are also given in the figure. Note that we have assumed a positive edge-triggered flip-flop.

A.6.5 JK FLIP-FLOP

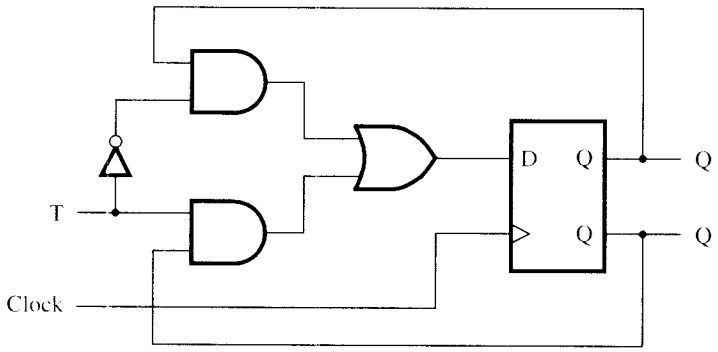
Another flip-flop that is sometimes encountered in practice is the *JK flip-flop*, which combines the behaviors of SR and T flip-flops. It is presented in Figure A.31. Its operation is defined by the truth table in Figure A.31*b*. The first three entries in this table define the same behavior as those in Figure A.25*b* (when $Clk = 1$), so that J and K correspond to S and R. For the input valuation J = K = 1, the next state is defined as the complement of the present state of the flip-flop. That is, when J = K = 1, the flip-flop functions as a *toggle*, reversing its present state.

A JK flip-flop can be implemented using a D flip-flop connected such that

$$D = J\overline{Q} + \overline{K}Q$$

The corresponding circuit is shown in Figure A.31*a*.

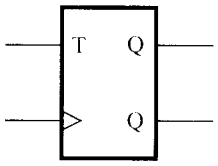
The JK flip-flop is versatile. It can be used to store data, just like the D flip-flop. It can also be used to build counters, because it behaves like the T flip-flop if its J and K input terminals are connected together.



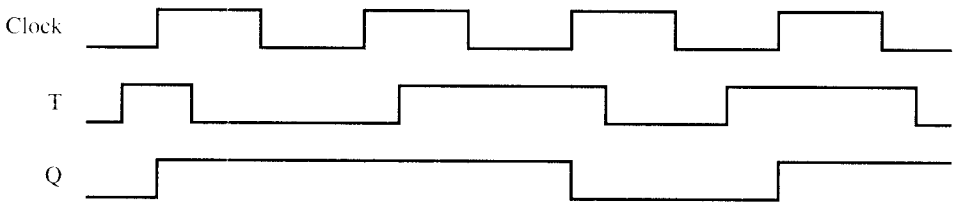
(a) Circuit

T	Q(t + 1)
0	Q(t)
1	Q(t)

(b) Truth table



(c) Graphical symbol

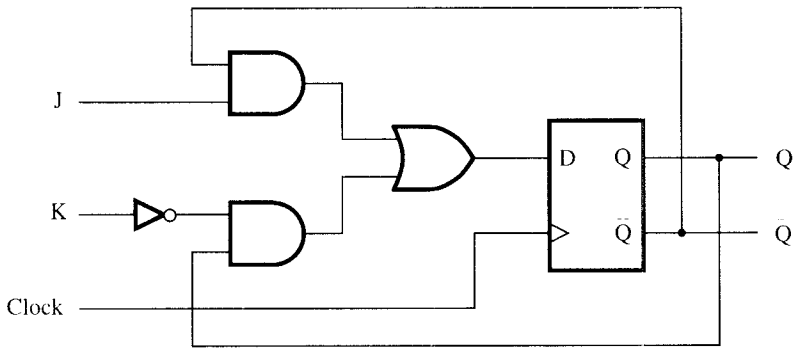


(d) Timing diagram

Figure A.30 T flip-flop.

A.6.6 FLIP-FLOPS WITH PRESET AND CLEAR

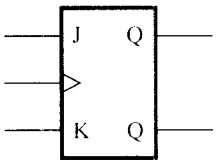
The state of a flip-flop is determined by its present state and the logic values on its input terminals. Sometimes it is desirable to force a flip-flop into a particular state, either 0 or 1, regardless of its present state and the values of the normal inputs. For example, when a computer is powered on, it is necessary to place all flip-flops into a known state. Usually, this means resetting their outputs to state 0. In some cases it is desirable to preset some flip-flops into state 1.



(a) Circuit

J	K	$Q(t+1)$
0	0	$Q(t)$
0	1	0
1	0	1
1	1	$Q(t)$

(b) Truth table



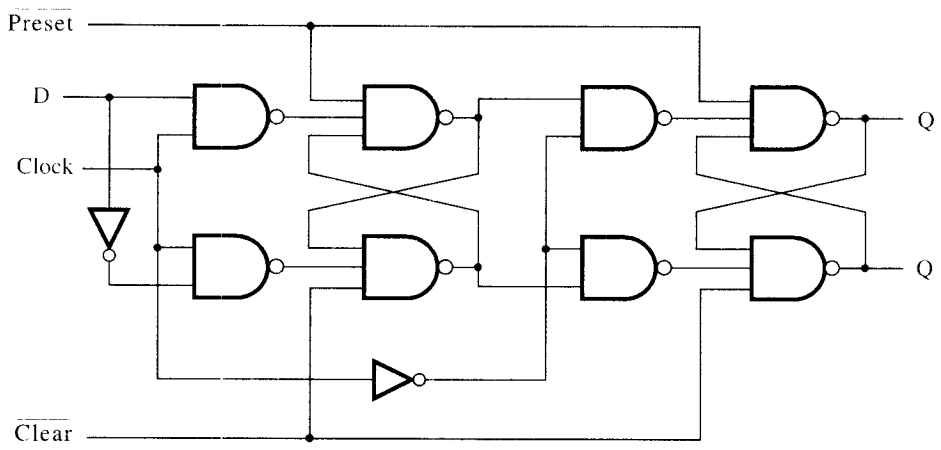
(c) Graphical symbol

Figure A.31 JK flip-flop.

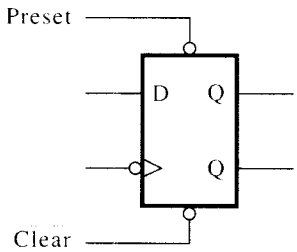
Figure A.32 illustrates how preset and clear control inputs can be added to a master-slave D flip-flop, to force the flip-flop into state 1 or 0 independent of the D input and the clock. These inputs are active low, as indicated by the overbars and bubbles in the figure. When both the $\overline{\text{Preset}}$ and $\overline{\text{Clear}}$ inputs are equal to 1, the flip-flop is controlled by the clock and D input in the normal way. When $\overline{\text{Preset}} = 0$, the flip-flop is forced to the 1 state, and when $\overline{\text{Clear}} = 0$, the flip-flop is forced to the 0 state. The preset and clear controls are also often incorporated in the other flip-flop types.

A.7 REGISTERS AND SHIFT REGISTERS

An individual flip-flop can be used to store one bit. However, in machines in which data are handled in words consisting of many bits (perhaps as many as 64), it is convenient to arrange a number of flip-flops into a common structure called a *register*. The operation of all flip-flops in a register is synchronized by a common clock. Thus, data are written (loaded) into or read from all flip-flops at the same time.



(a) Circuit



(b) Graphical symbol

Figure A.32 Master-slave D flip-flop with Preset and Clear.

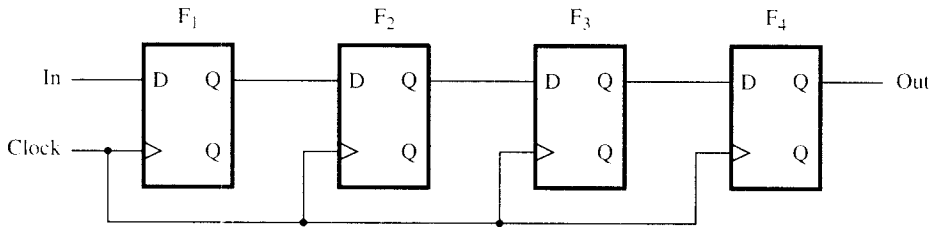


Figure A.33 A simple shift register.

Processing of digital data often requires the capability to shift and rotate the data, so it is necessary to provide the hardware with this facility. A simple mechanism for realizing both operations is a register whose contents may be shifted to the right or left one bit position at a time. As an example, consider the 4-bit shift register in Figure A.33. It consists of D flip-flops connected so that each clock pulse will cause the transfer of

the contents (state) of F_i to F_{i+1}, effecting a "right shift." Data are shifted serially into and out of the register. A rotation of the data can be implemented by connecting Out to In.

Proper operation of a shift register requires that its contents be shifted exactly one position for each clock pulse. This places a constraint on the type of storage elements that can be used. Gated latches, depicted in Figure A.27, are not suitable for this purpose. While the clock is high, the value on D input quickly propagates to the output. From there, the value propagates through the next gated latch in the same manner. Hence, there is no control over the number of shifts that will take place during a single clock pulse. This number depends on the propagation delays of the gated latches and the duration of the clock pulse. The solution to the problem is to use either the master-slave or the edge-triggered flip-flops.

A particularly useful form of a shift register is one that can be loaded and read in parallel. This can be accomplished with some additional gating as illustrated in Figure A.34, which shows a 4-bit register constructed with D flip-flops. The register can be loaded either serially or in parallel. When the register is clocked, a shift takes place if $\overline{\text{Shift}}/\text{Load} = 0$; otherwise, a parallel load is performed.

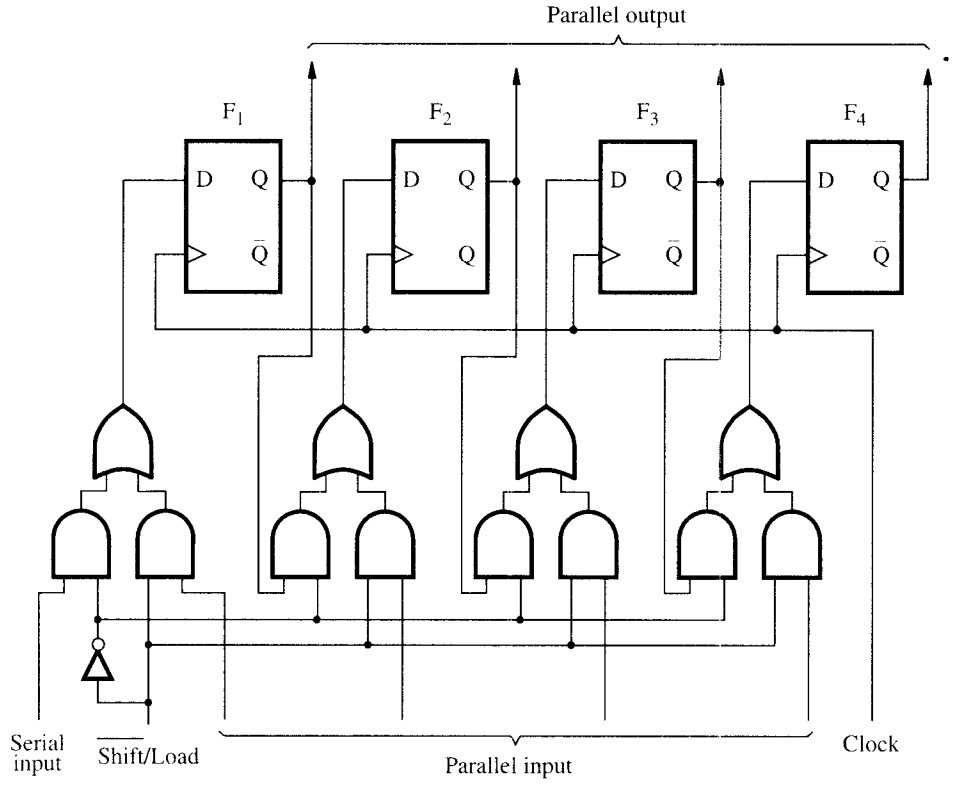
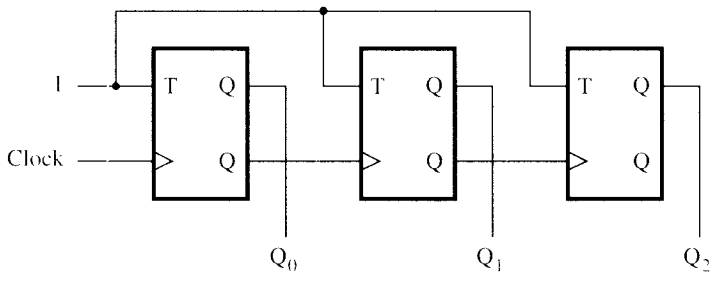


Figure A.34 Parallel-access shift register.

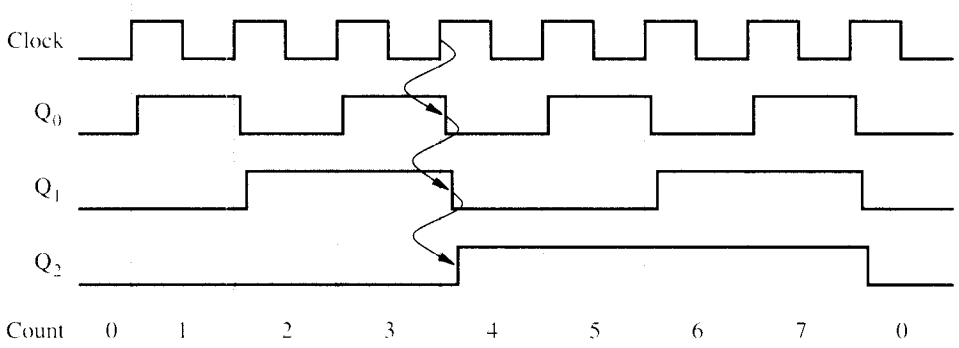
A.8 COUNTERS

In the preceding section, we discussed the applicability of flip-flops in the construction of shift registers. They are equally useful in the implementation of *counter* circuits. It is hardly necessary to justify the need for counters in digital machines. In addition to being hardware mechanisms for realizing ordinary counting functions, counters are also used to generate control and timing signals. A counter driven by a high-frequency clock can be used to produce signals whose frequencies are submultiples of the original clock frequency. In such applications a counter is said to be functioning as a *scaler*.

A simple three-stage (or 3-bit) counter constructed with T flip-flops is shown in Figure A.35. Recall that when the T input is equal to 1, the flip-flop acts as a toggle, that is, its state changes with each successive clock pulse. Thus, two clock pulses will cause Q_0 to change from the 1 state to the 0 state and back to the 1 state or from 0 to 1 to 0. This means that the output waveform of Q_0 has half the frequency of the clock. Similarly, because the second flip-flop is driven by Q_0, the waveform at Q_1 has half the frequency of Q_0, or one-fourth the frequency of the clock. Note that we have assumed that the positive edge of the clock input to each flip-flop triggers the change of its state.



(a) Circuit



(b) Timing diagram

Figure A.35 A 3-bit up-counter.

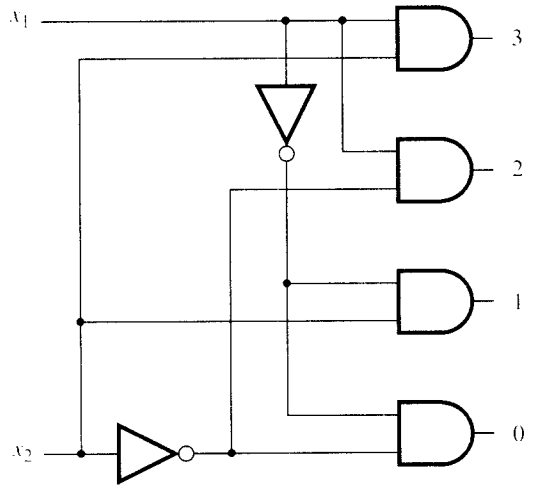
Such a counter is often called a *ripple counter* because the effect of an input clock pulse ripples through the counter. For example, the positive edge of pulse 4 will change the state of Q_0 from 1 to 0. This change in Q_0 will then force Q_1 from 1 to 0, which in turn forces Q_2 from 0 to 1. If each flip-flop introduces some delay Δ, then the delay in setting Q_2 is 3Δ. Such delays can be a problem when very fast operation of counter circuits is required. In many applications, however, these delays are small in comparison with the clock period and can be neglected.

With the addition of some extra logic gates, it is possible to construct a "synchronous" counter in which each stage is under the control of the common clock so that all flip-flops can change their states simultaneously. Such counters are capable of operation at higher speed because the total propagation delay is reduced considerably. In contrast, the counter in Figure A.35 is said to be "asynchronous."

A.9 DECODERS

Much of the information in computers is handled in a highly encoded form. In an instruction, an n-bit field may be used to denote 1 out of 2^n possible choices for the action to be taken. To perform the desired action, the encoded instruction must first be decoded. A circuit capable of accepting an n-variable input and generating the corresponding output signal on one out of 2^n output lines is called a *decoder*. A simple example of a two-input to four-output decoder is given in Figure A.36. One of the four output lines is selected by the inputs x_1 and x_2, as indicated in the figure. The selected output has the logic value 1, and the remaining outputs have the value 0.

Other useful types of decoders exist. For example, using information in BCD form often requires decoding circuits in which a four-variable BCD input is used to select



x_1	x_2	Active output
0	0	0
0	1	1
1	0	2
1	1	3

Figure A.36 A two-input to four-output decoder.

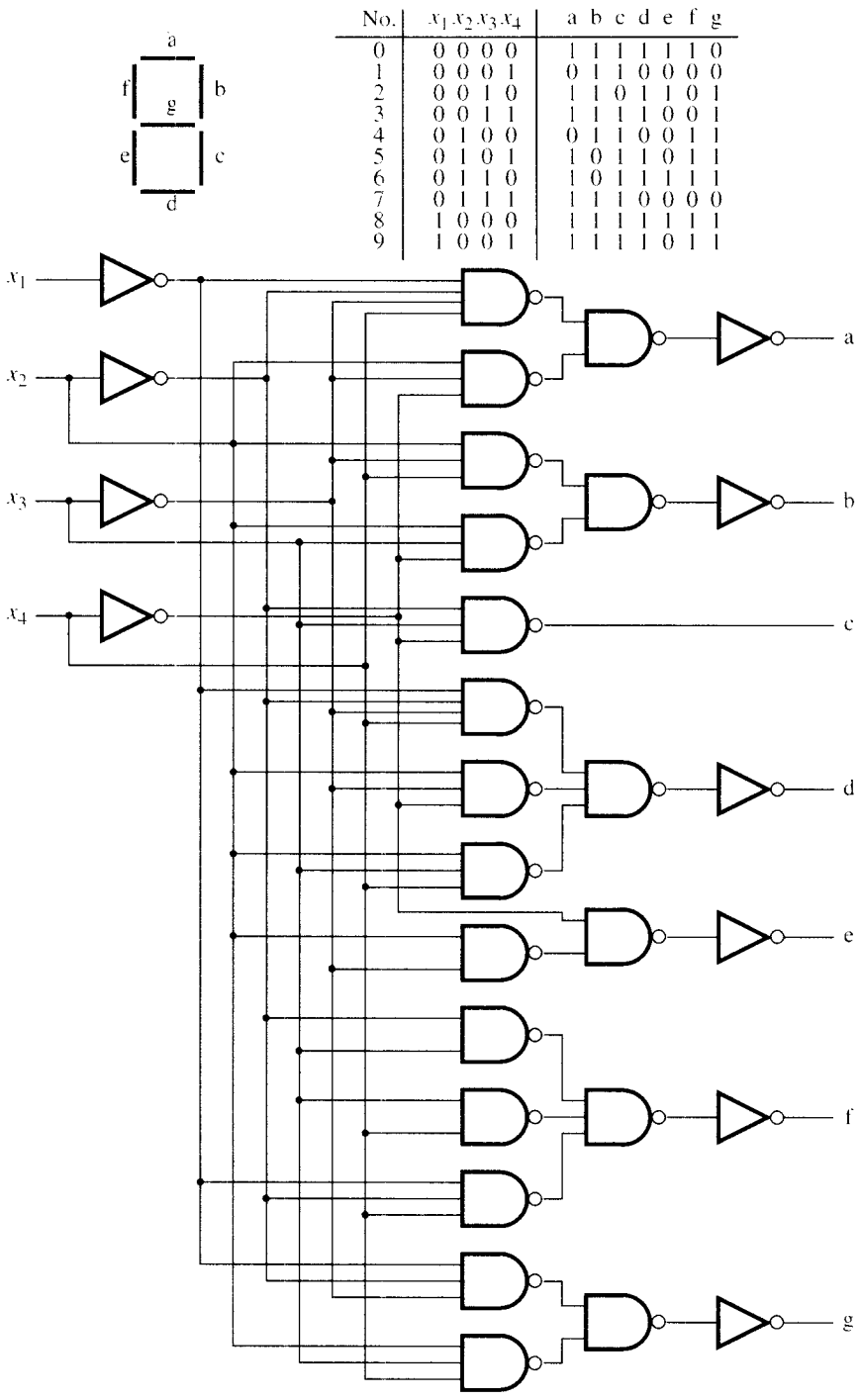


Figure A.37 A BCD to seven-segment display decoder.

1 out of 10 possible outputs. As another specific example, let us consider a decoder suitable for driving a seven-segment display. Figure A.37 shows the structure of a seven-segment element used for display purposes. We can easily see that any decimal number from zero to nine can be displayed with this element simply by turning some segments on (light) while leaving others off (dark). The necessary functions are indicated in the table. They can be realized using the decoding circuit shown in the figure. Note that the circuit is constructed with NAND gates. We encourage the reader to verify that the circuit implements the required functions.

A.10 MULTIPLEXERS

In the preceding section, we saw that decoders select one output line on the basis of input signals. The selected output line has logic value 1, while the other outputs have the value 0. Another class of very useful selector circuits exists in which any one of n data inputs can be selected to appear as the output. The choice is governed by a set of "select" inputs. Such circuits are called *multiplexers*. An example of a multiplexer circuit is shown in Figure A.38. It has two select inputs, w_1 and w_2. Their four possible valuations are used to select one of four inputs, x_1, x_2, x_3, or x_4, to appear as the output z. A simple logic circuit that can implement the required operation is also given. Obviously, the same structure can be used to realize larger multiplexers, in which k select inputs are used to connect one of the 2^k data inputs to the output.

The obvious application of multiplexers is in the gating of data that may come from a number of different sources. For example, loading a 16-bit data register from one of four distinct sources can be accomplished with sixteen 4-input multiplexers.

Multiplexers are also very useful as basic elements for implementing logic functions. Consider a function f defined by the truth table of Figure A.39. It can be represented as shown in the figure by factoring out the variables x_1 and x_2. Note that for each valuation of x_1 and x_2, the function f corresponds to one of four terms: 0, 1, x_3, or $\overline{x}_3$. This suggests the possibility of using a four-input multiplexer circuit, in which x_1 and x_2 are the two select inputs that choose one of the four data inputs. Then, if the data inputs are connected to 0, 1, x_3, or $\overline{x}_3$ as required by the truth table, the output of the multiplexer will correspond to the function f. The approach is completely general. Any function of three variables can be realized with a single four-input multiplexer. Similarly, any function of four variables can be implemented with an eight-input multiplexer, and so on.

A.11 PROGRAMMABLE LOGIC DEVICES (PLDS)

Sections A.2 and A.3 showed how a given switching function can be represented in terms of sum-of-products expressions and implemented by corresponding AND-OR gate networks. Section A.10 showed how multiplexers can be used to realize switching functions. In this section we will consider another class of circuits that can be used for

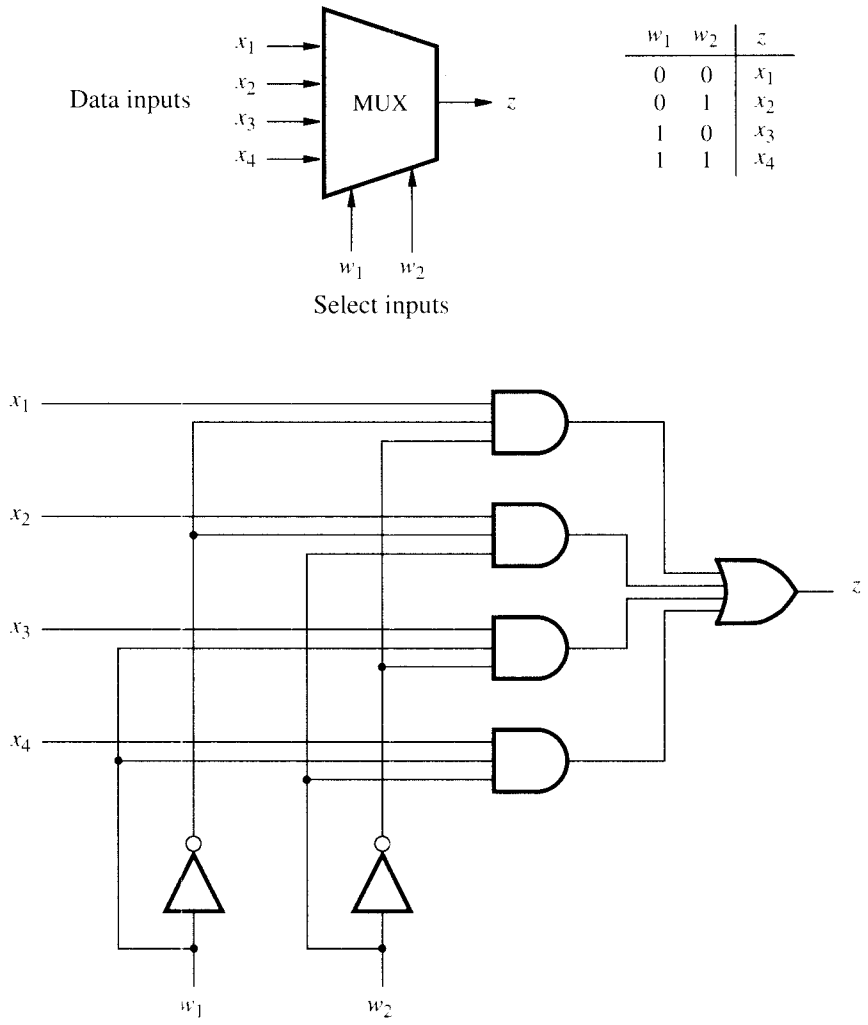


Figure A.38 A four-input multiplexer.

the same purpose. These circuits consist of arrays of switching elements that can be programmed to allow implementation of sum-of-products expressions. They are called *programmable logic devices* (PLDs).

Figure A.40 shows the block diagram of a PLD. It has n input variables ($x_1, \ldots, x_n$) and m output functions ($f_1, \ldots, f_m$). Each function f_i is realized as a sum of product terms that involve the input variables. The variables $x_1, \ldots, x_n$ are presented in true and complemented form to the AND array, where up to k product terms are formed. These are then gated into the OR array, where the output functions are formed. Two commonly used types of PLDs are described in the remainder of this section.

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

⇒

x_1	x_2	f
0	0	0
0	1	x_3
1	0	1
1	1	x_3

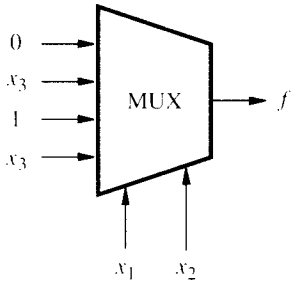


Figure A.39 Multiplexer implementation of a logic function.

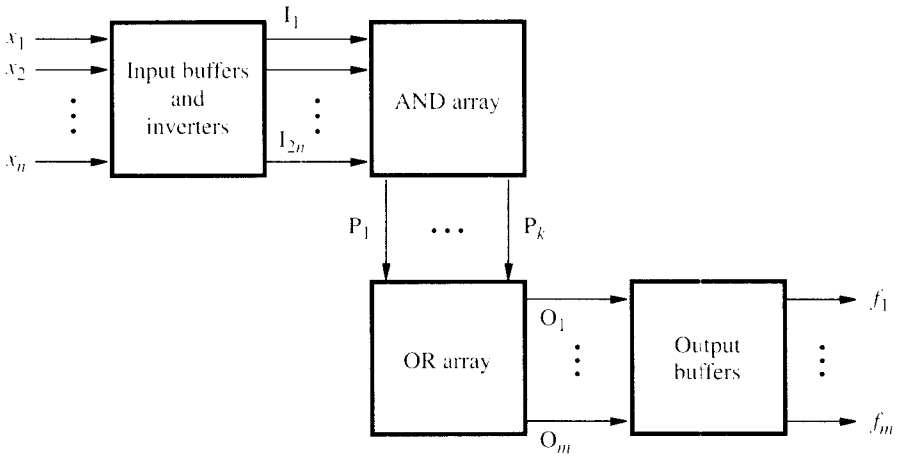
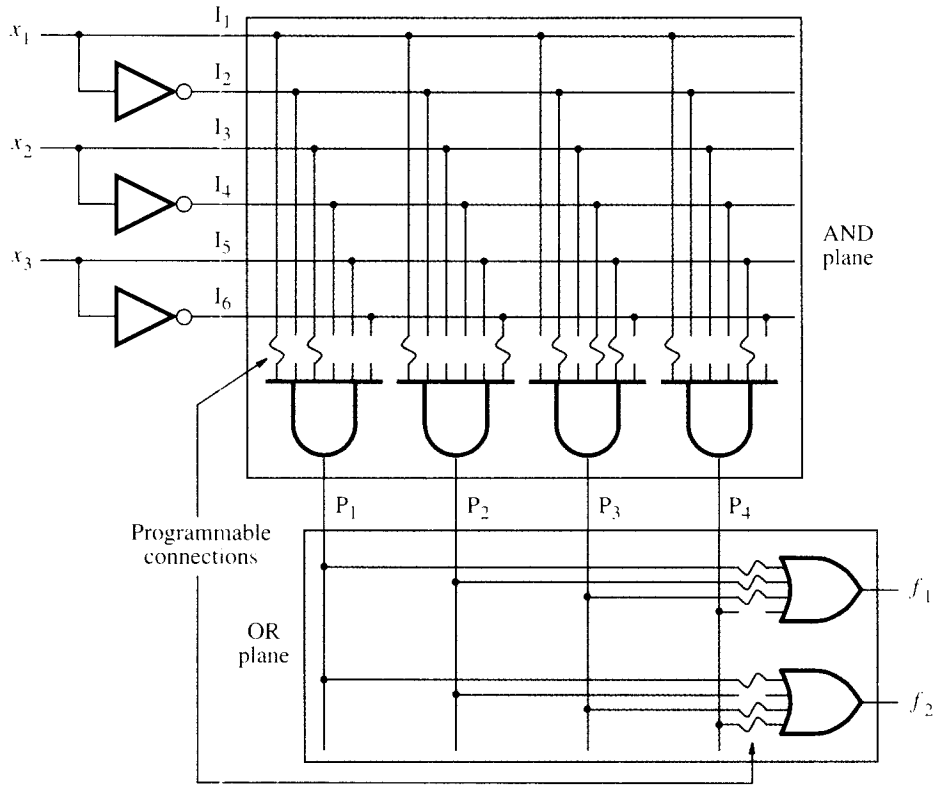


Figure A.40 A block diagram for a PLD.

A.11.1 PROGRAMMABLE LOGIC ARRAY (PLA)

A circuit in which connections to both the AND and the OR arrays can be programmed is called a *programmable logic array* (PLA). Figure A.41 illustrates the functional structure of a PLA using a simple example. The programmable connections must be



$$f_1 = x_1x_2 + x_1x_3 + x_1x_2x_3$$

$$f_2 = x_1x_2 + x_1x_2x_3 + x_1x_3$$

Figure A.41 Functional structure of a PLA.

such that if no connection is made to a given input of an AND gate, the input behaves as if a logic value of 1 is driving it (that is, this input does not contribute to the product term realized by this gate). Similarly, if no connection is made to a given input of an OR gate, this input must have no effect on the output of the gate (that is, the input must behave as if a logic value of 0 is driving it).

Programmed connections may be realized in different ways. In one method, programming consists of blowing fuses in positions where connections are not required. This is done by applying higher-than-normal current. Another possibility is to use transistor switches controlled by erasable memory elements (see Section 5.3 on EPROM memory circuits) to provide the connections as desired. This allows the PLA to be reprogrammable.

The simple PLA in Figure A.41 can generate up to four product terms from three input variables. Two output functions may be implemented using these product terms. Some of the product terms may be used in more than one output function. The PLA is

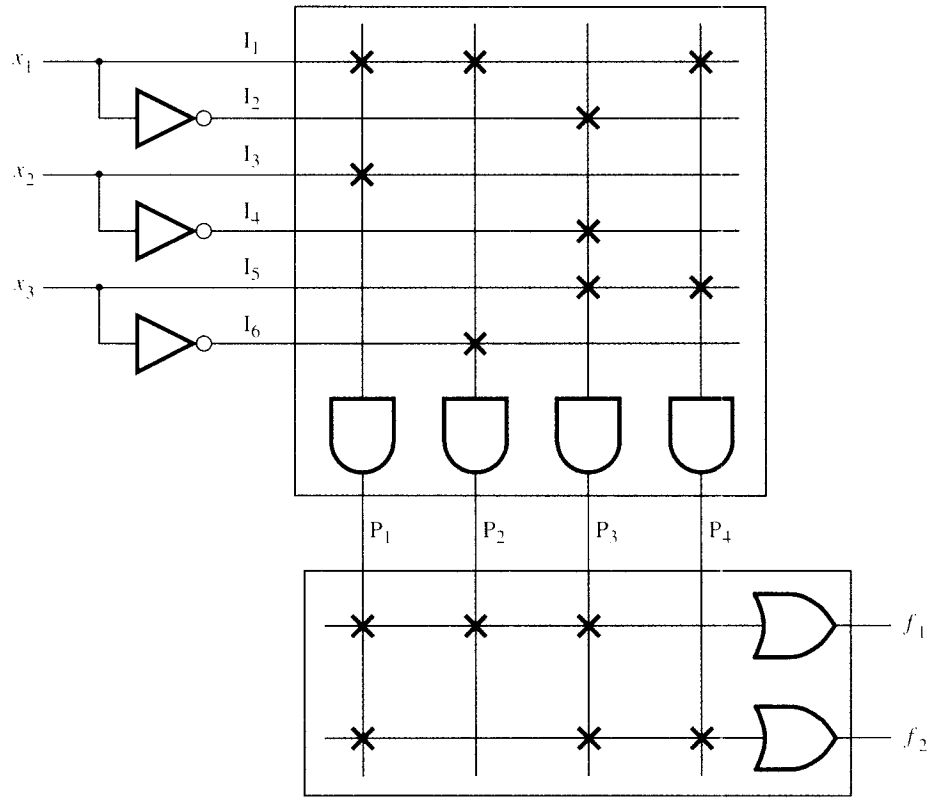


Figure A.42 A simplified sketch of the PLA in Figure A.41.

configured to realize the following two functions:

$$f_1 = x_1x_2 + x_1\bar{x}_3 + \bar{x}_1\bar{x}_2x_3$$
$$f_2 = x_1x_2 + x_1x_3 + \bar{x}_1\bar{x}_2x_3$$

Only four product terms are needed, because two terms can be shared by both functions. Practical PLAs come in much larger sizes.

Although Figure A.41 depicts clearly the basic functionality of a PLA, this style of presentation is awkward for describing a larger PLA. It has become customary in technical literature to represent the product and sum terms by means of corresponding gate symbols that have only one symbolic input line. An $\times$ is placed on this line to represent each programmed connection. This drawing convention is used in Figure A.42 to represent the PLA example from Figure A.41. In general, a programmable connection can be made at any crossing of a vertical line and a horizontal line in the diagram, to implement arbitrary functions of input variables.

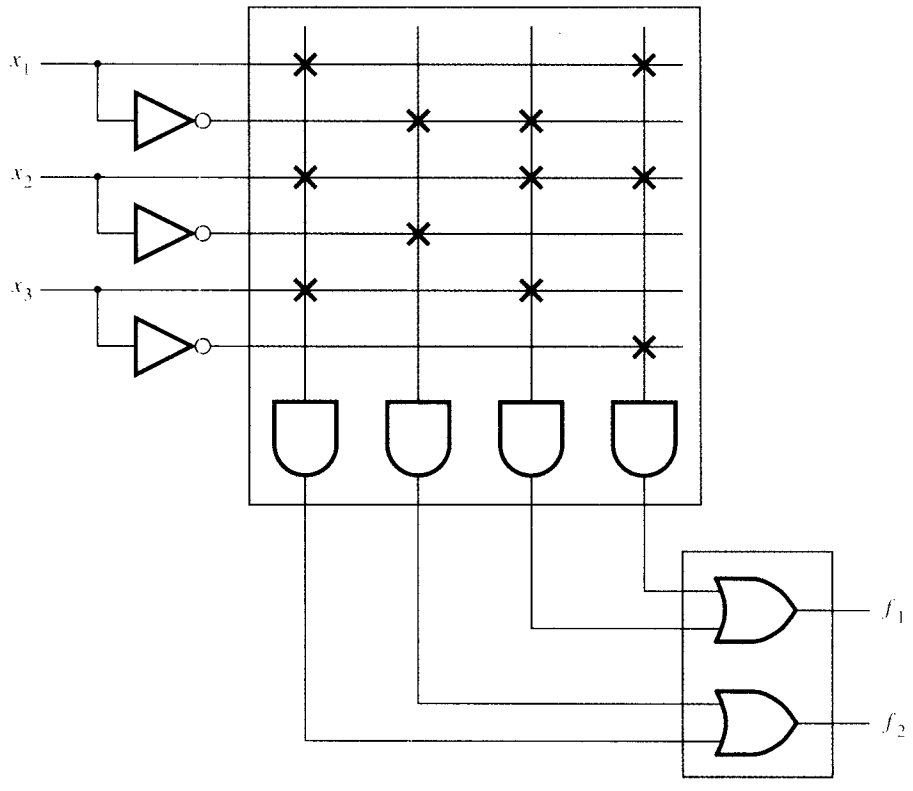
The PLA structure is very efficient in terms of the area needed for its implementation on an integrated circuit chip. For this reason, such structures are often used for

implementing control circuits in processor chips. In this case, the desired connections are put in place as the last step in the manufacturing process, rather than making them programmable after the chip has been fabricated.

A.11.2 PROGRAMMABLE ARRAY LOGIC (PAL)

In a PLA, the inputs to both the AND array and the OR array are programmable. A similar device, in which the inputs to the AND array are programmable but the connections to the OR gates are fixed, has found great popularity in practical applications. Such devices are known as *programmable array logic* (PAL) chips.

Figure A.43 shows a simple example of a PAL that can implement two functions. The number of AND gates connected to each OR gate in a PAL determines the maximum number of product terms that can be realized in a sum-of-products representation of a



$$f_1 = x_1x_2x_3 + x_1x_2x_3$$

$$f_2 = x_1x_2 + x_1x_2x_3$$

Figure A.43 An example of a PAL.

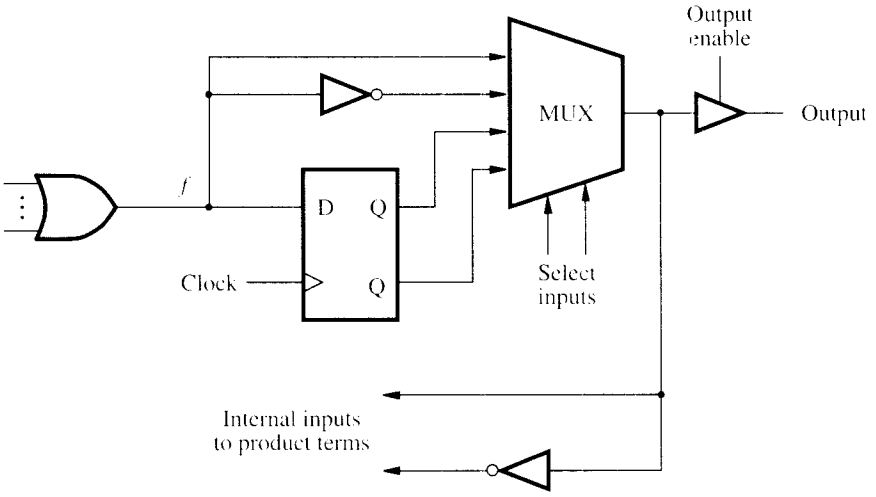


Figure A.44 An example of the output of a PAL element.

given function. The AND gates are permanently connected to specific OR gates, which means that a particular product term cannot be shared among output functions.

PAL chips are available in various configurations. A substantial number of input variables and output functions can be provided, allowing large functions to be realized. The versatility of a PAL may be enhanced further by including flip-flops in the outputs from the OR gates. Such PAL chips enable the designer of a digital system to implement a relatively complex logic network using a single chip.

Figure A.44 indicates the kind of flexibility that can be provided. A multiplexer is used to choose whether a true, complemented, or stored (from the previous clock cycle) value of f is to be presented at the output pin of the PAL chip. The select inputs to the multiplexer can be set as programmable connections. The output pin is driven by a tri-state driver under control of the Output-enable signal. Note that the signal from the output of the multiplexer is also made available as an internal input that can be used in product terms that feed other OR gates in the PAL. This facilitates the realization of circuits that have several levels (stages) of logic gates.

A.11.3 COMPLEX PROGRAMMABLE LOGIC DEVICES (CPLDS)

PALs are useful devices, but their relatively small size means that many such chips may be needed to implement a typical digital system. Larger devices of a similar type have been developed to deal with this issue. They are known as *complex programmable logic devices* (CPLDs). They comprise two or more PAL-like blocks and programmable interconnection wires. Figure A.45 indicates the structure of a CPLD chip. Each PAL-like block is connected to a number of input/output pins. Connections between PAL-like blocks are established by programming the switches associated with the interconnection wires.

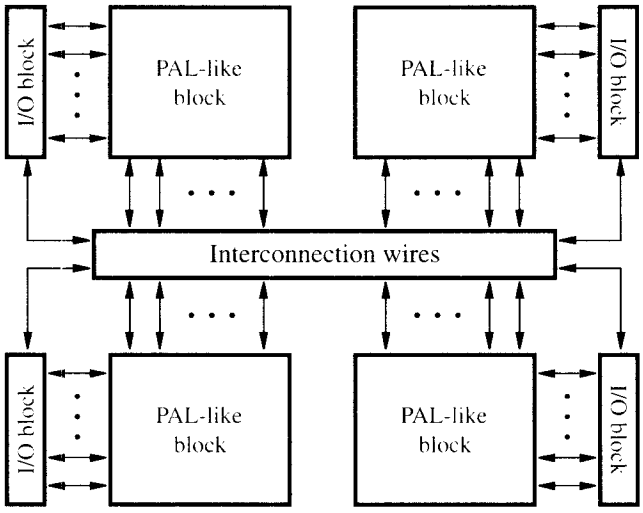


Figure A.45 Structure of a complex programmable logic device (CPLD).

The interconnection resource consists of horizontal and vertical wires. Each horizontal wire can be connected to some of the vertical wires by programming the corresponding switches. It is impractical to provide full connectivity, where each horizontal wire can be connected to any of the vertical wires, because the number of required switches would be large. Satisfactory connectivity can be achieved with a much smaller number of switches.

Commercial CPLDs come in different sizes, ranging from 2 to more than 100 PAL-like blocks. A CPLD chip is programmed by loading the programming information into it via a *JTAG port*. This is a 4-pin port that conforms to an IEEE standard developed by the Joint Test Action Group.

A.12 FIELD-PROGRAMMABLE GATE ARRAYS

PAL chips provide general functionality but are somewhat limited in size because an output pin is provided for each sum-of-products circuit. A more powerful class of programmable devices has been developed to overcome these size limitations. They are known as *field-programmable gate arrays* (FPGAs). Figure A.46 shows a conceptual block diagram of an FPGA. It consists of an array of logic blocks (indicated as black boxes) that can be connected by general interconnection resources. The *interconnect*, shown in blue, consists of segments of wire and programmable switches. The switches are used to connect the logic blocks to the wire segments and to establish connections between different wire segments as desired. This allows a large degree of routing flexibility on the chip. Input and output buffers are provided for access to the pins of the chip.

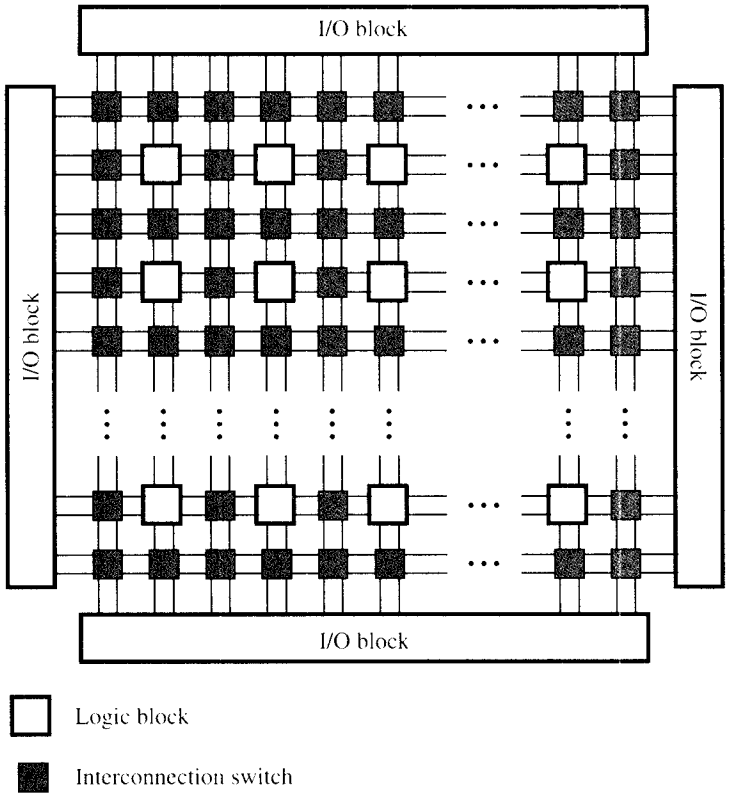


Figure A.46 A conceptual block diagram of an FPGA.

There are a variety of designs for the logic blocks and the interconnect structure. A logic block may be just a simple multiplexer-based circuit capable of implementing logic functions as discussed in Section A.10. Another popular design uses a simple lookup table as a logic block. For example, a four-input lookup table can be implemented in the form of a 16-bit memory circuit in which the truth table of a logic function is stored. Each memory bit corresponds to one combination of the input variables. Such a lookup table can be programmed to implement any function of four variables. The logic blocks may contain flip-flops to provide additional flexibility of the type encountered in Figure A.44.

In addition to the logic blocks, many FPGA chips include a substantial number of memory cells (not shown in Figure A.46), which may be used to implement structures such as first-in first-out (FIFO) queues or RAM and ROM components in system-on-a-chip applications, which are discussed in Chapter 9.

From the user's point of view, there are two major differences between FPGAs and CPLDs. The FPGA chips have much greater functionality and can be used to implement rather large logic networks. An FPGA chip may implement a circuit that requires over a million logic gates. The second important consideration is the speed of these devices.

Since programmable switches are used to establish all connections in the interconnect, an FPGA will inevitably have significantly longer propagation delays compared with a less flexible device such as a PAL or a CPLD.

The growing popularity of FPGAs is due to the fact that they allow a designer to implement very complex logic networks on a single chip without having to design and fabricate a custom VLSI chip, which is both expensive and time-consuming. Using CAD tools, it is possible to generate an FPGA design in a matter of days, rather than the months needed to produce a custom-designed VLSI chip. The FPGA implementations are also attractive in terms of cost. Even the largest FPGAs cost only a few hundred dollars, and the cost associated with the design time is very small compared to the cost of designing a custom chip.

An introductory discussion of programmable logic devices can be found in many modern books on logic design. For a more extensive treatment of these devices, the reader may consult other books [1, 3–6] and manufacturers' literature.

A.13 SEQUENTIAL CIRCUITS

A combinational circuit is one whose output is determined entirely by its present inputs. Examples of such circuits are the decoders and multiplexers presented in Sections A.9 and A.10. A different class of circuits are those whose outputs depend on both the present inputs and on the sequence of previous inputs. They are called *sequential circuits*. Such circuits can be in different *states*, depending on what the sequence of inputs has been up to a given time. The state of a circuit determines the behavior when various input patterns are applied to the circuit. We encountered two specific forms of such circuits in Sections A.7 and A.8, called shift registers and counters. In this section, we will introduce more examples of sequential circuits, provide a general form for them, and give a brief introduction to the design of these circuits.

A.13.1 AN EXAMPLE OF AN UP/DOWN COUNTER

Figure A.35 shows the configuration of an up counter, implemented with three T flip-flops, which counts in the sequence 0, 1, 2, . . . , 7, 0, ... A similar circuit can be used to count in the down direction, that is, 0, 7, 6, . . . , 1, 0, . . . (see Problem A.26). These simple circuits are made possible by the toggle feature of T flip-flops.

We now consider the possibility of implementing such counters with D flip-flops. As a specific example, we will design a counter that counts either up or down, depending on the value of an external control input. To keep the example small, let us restrict the size to a mod-4 counter, which requires only two state bits to represent the four possible count values. We will show how this counter can be designed using general techniques for the synthesis of sequential circuits. The desired circuit will count up if an input signal x is equal to 0 and down if x is 1. The count will change on the negative edge of the clock signal. Let us assume that we are particularly interested in the state when the count is equal to 2. Thus, an output signal, z, should be asserted when the count is equal to 2; otherwise $z = 0$.

The desired counter can be implemented as a sequential circuit. In order to determine what the new count will be when a clock pulse is applied, it is sufficient to know the value of x and the present count. It is not necessary to know what the actual sequence of previous input values has been, as long as we know the present count that has been reached. This count value is said to determine the *present state* of the circuit, which is all that the circuit remembers about previous input values. If the present count is 2 and $x = 0$, the next count will be 3. It makes no difference whether the count of 2 was reached counting down from 3 or up from 1.

Before we show a circuit implementation, let us depict the desired behavior of the counter by means of a state diagram. The counter has four distinct states: S0, S1, S2, and S3. A *state diagram* is a graph in which states are represented as circles (sometimes called nodes). Transitions between states are indicated by labeled arrows. The label associated with an arrow specifies the value of the input x that will cause this particular transition to occur and the value of the output produced as a result. Figure A.47 shows the state diagram of our up/down counter. For example, the arrow emanating from state S1 (count = 1) for an input $x = 0$ points to state S2, thus specifying the transition to state S2. It also indicates that the output z must be equal to 0 while the circuit is in state S1 and the value of x is 0. An arrow from S2 to S3 specifies that when $x = 0$ the next clock pulse will cause a transition from S2 to S3, and that the output z should be 1 while the circuit is in state S2.

Note that the state diagram describes the functional behavior of the counter without any reference to how it is implemented. Figure A.47 can be used to describe an electronic digital circuit, a mechanical counter, or a computer program that behaves in this way. Such diagrams are a powerful means of describing any system that exhibits sequential behavior.

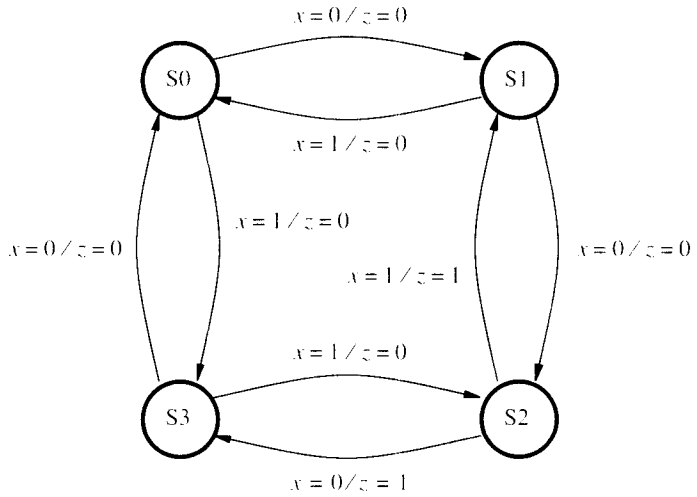


Figure A.47 State diagram of a mod-4 up/down counter that detects the count of 2.

Present state	Next state		Output z	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
S0	S1	S3	0	0
S1	S2	S0	0	0
S2	S3	S1	1	1
S3	S0	S2	0	0

Figure A.48 State table for the example of the up/down counter.

A different way of presenting the information in a state diagram is to use a *state table*. Figure A.48 gives the state table for the example in Figure A.47. The table indicates transitions from all present states to the *next states*, as required by the applied input x. The output signal, z, is determined by the present state of the circuit and the value of the applied input, x.

Having specified the desired up/down counter in general terms, we will now consider its physical realization. Two bits are needed to encode the four states that indicate the count. Let these bits be y_2 (high-order) and y_1 (low-order). The states of the counter are determined by the values of y_2 and y_1, which we will write in the form y_2y_1. We will assign values to y_2y_1 for each of the four states as follows: S0 = 00, S1 = 01, S2 = 10, and S3 = 11. We have chosen the assignment such that the binary number y_2y_1 represents the count in an obvious way. The variables y_2 and y_1 are called the *state variables* of the sequential circuit. Using this *state assignment*, the state table for our example is as shown in Figure A.49. Note that we are using the variables Y_1 and Y_2 to denote the next state in the same manner as y_1 and y_2.

It is important to note that we could have chosen a different assignment of y_2y_1 values to the various states. For example, a possible state assignment is: S0 = 10, S1 = 11, S2 = 01, and S3 = 00. For a counter circuit, this assignment is less intuitive than the one in Figure A.49, but the resultant circuit will work properly. Different state assignments usually lead to different costs in implementing the circuit (see Problem A.32).

Our intention in this example is to use D flip-flops to store the values of the two state variables between successive clock pulses. The output, Q, of a flip-flop is the present-state variable y_i, and the input, D, is the next-state variable Y_i. Note that Y_i is a function of y_2, y_1, and x, as indicated in Figure A.49. From the figure, we see that

$$\begin{aligned} Y_2 &= \overline{y}_2 y_1 \overline{x} + y_2 \overline{y}_1 \overline{x} + \overline{y}_2 \overline{y}_1 x + y_2 y_1 x \\ &= y_2 \oplus y_1 \oplus x \\ Y_1 &= \overline{y}_2 \overline{y}_1 \overline{x} + y_2 \overline{y}_1 \overline{x} + \overline{y}_2 \overline{y}_1 x + y_2 \overline{y}_1 x \\ &= \overline{y}_1 \end{aligned}$$

Present state	Next state		Output z	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
$y_2 y_1$	$Y_2\ Y_1$	$Y_2\ Y_1$		
0 0	0 1	1 1	0	0
0 1	1 0	0 0	0	0
1 0	1 1	0 1	1	1
1 1	0 0	1 0	0	0

Figure A.49 State assignment for the example in Figure A.48.

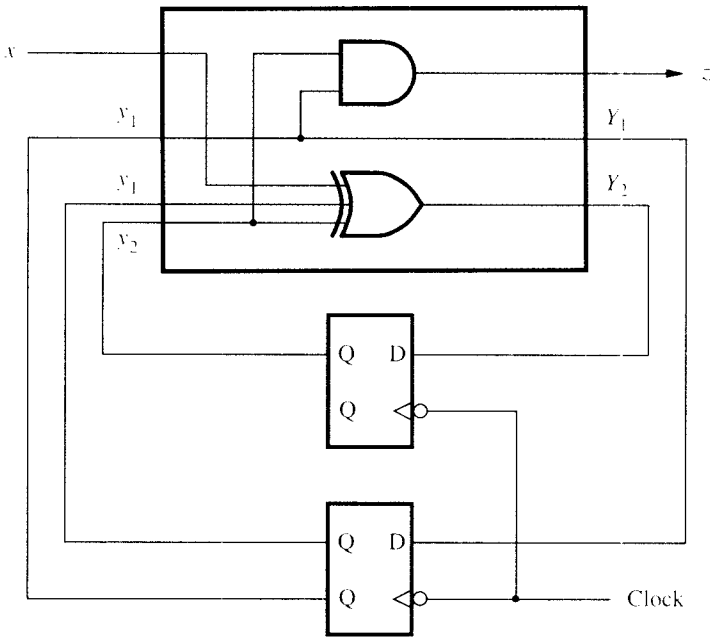


Figure A.50 Implementation of the up/down counter.

The output z is determined as

$$z = y_2 \bar{y}_1$$

These expressions lead to the circuit shown in Figure A.50.

A.13.2 TIMING DIAGRAMS

To fully understand the operation of the counter circuit, it is useful to consider its timing diagram. Figure A.51 gives an example of a possible sequence of events. It assumes that state transitions (changes in flip-flop values) occur on the negative edge of the clock and that the counter starts in state S0. Since $x = 0$, the counter advances to state S1 at t_0, then to S2 at t_1, and to S3 at t_2. The output changes from 0 to 1 when the counter enters state S2. It goes back to 0 when state S3 is reached. At the end of S3, at t_3, the counter goes to S0. We have assumed that at this time the input x changes to 1, causing the counter to count in the down sequence. When the count again reaches S2, at t_5, the output z goes to 1.

Note that all signal changes occur just after the negative edge of the clock, and signals do not change again until the negative edge of the next clock pulse. The delay from the clock edge to the time at which variables y_i change is the propagation delay of the flip-flops used to implement the counter circuit. It is important to note that the input x is also assumed to be controlled by the same clock, and it changes only near the beginning of a clock period. These are essential features of circuits where all changes are controlled by a clock. Such circuits are called *synchronous sequential circuits*.

Another important observation concerns the relationship between the labels used in the state diagram in Figure A.47 and the timing diagram. For example, consider the clock period between t_1 and t_2. During this clock period, the machine is in state S2 and the input value is $x = 0$. This situation is described in the state diagram by the arrow

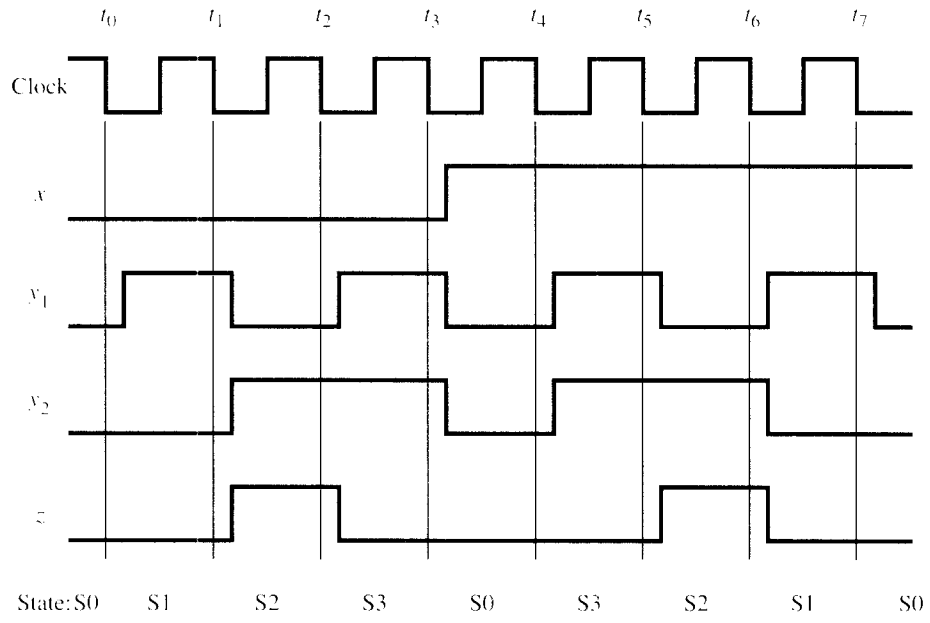


Figure A.51 Timing diagram for the circuit in Figure A.50.

emanating from state S2 labeled $x = 0$. Since this arrow points to state S3, the timing diagram shows y_2 and y_1 changing to the values corresponding to state S3 at the next clock edge, t_2. The output value associated with the arrow gives the value of z while the counter is in state S2.

A.13.3 THE FINITE STATE MACHINE MODEL

The specific example of the up/down counter implemented as a synchronous sequential circuit with flip-flops and combinational logic gates, as shown in Figure A.50, is easily generalized to the formal *finite state machine* model given in Figure A.52. In this model, the time delay through the delay elements is equal to the duration of the clock cycle. This is the time that elapses between changes in Y_i and the corresponding changes in y_i. The model assumes that the combinational logic block has no delay; hence, the outputs z, Y_1, and Y_2 are instantaneous functions of the inputs x, y_1, and y_2. In an actual circuit, some delay will be introduced by the circuit elements, as shown in Figure A.51. The circuit will work properly if the delay through the combinational logic block is short with respect to the clock cycle. The next-state outputs Y_i must be available in time to cause the flip-flops to change to the desired next state at the end of the clock cycle. Also, while the output z may not be at the desired value during all of the clock cycle, it must reach this value well before the end of the cycle.

Inputs to the combinational logic block consist of the flip-flop outputs, y_i, which represent the present state, and the external input, x. The outputs of the block are the

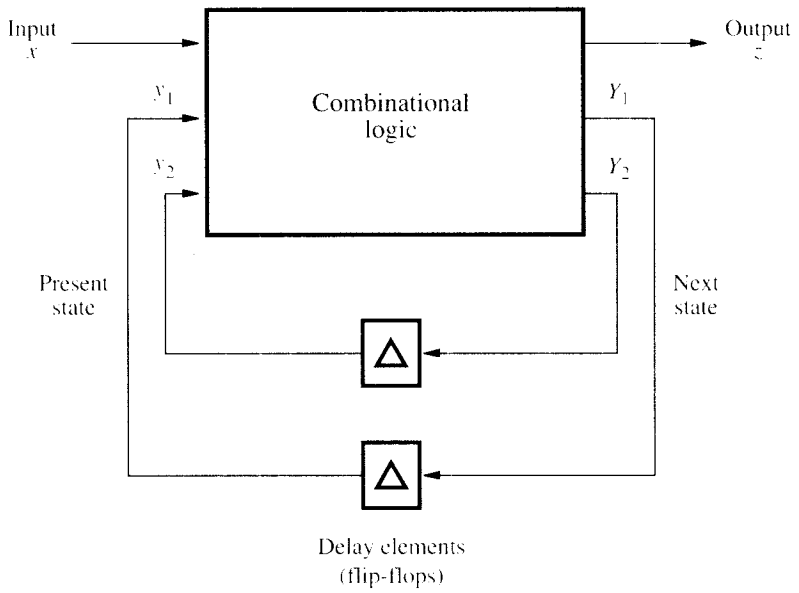


Figure A.52 A formal model of a finite state machine.

inputs to the flip-flops, which we have called Y_i, and the external output, z. When the active clock edge arrives marking the end of the present clock cycle, the values on the Y_i lines are loaded into the flip-flops. They become the next set of values of the state variables, y_i. Since these signals are connected to the input of the combinational block, they, along with the next value of the external input x, will produce new z and Y_i values. A clock cycle later, the new Y_i values are transferred to y_i, and the process repeats. In other words, the flip-flops constitute a feedback path from the output to the input of the combinational block, introducing a delay of one clock period.

Although we have shown only one external input, one external output, and two state variables in Figure A.52, it is clear that multiple versions are possible for any of the three types of variables.

A.13.4 SYNTHESIS OF FINITE STATE MACHINES

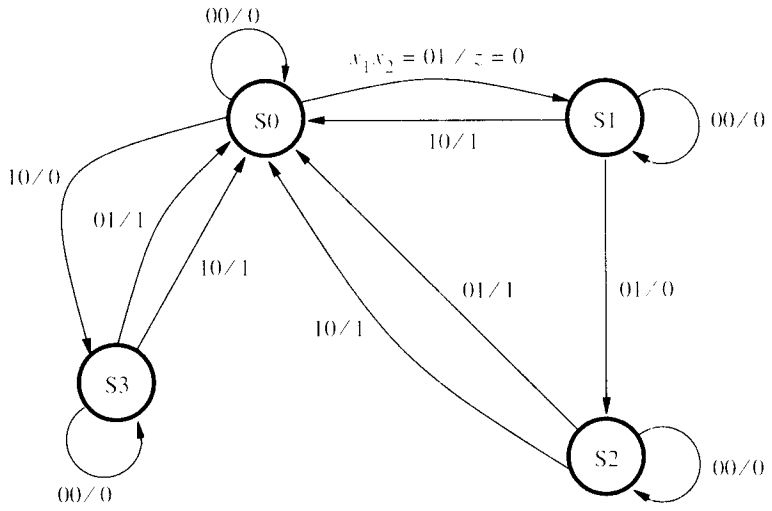
Let us summarize how to design a synchronous sequential circuit having the general organization in Figure A.52, based on a state diagram like that in Figure A.47. The design, or synthesis, process involves the following steps:

1. Develop an appropriate state diagram or state table.
2. Determine the number of flip-flops needed, and choose a suitable type of flip-flop.
3. Determine the values to be stored in these flip-flops for each state in the state diagram. This is referred to as state assignment.
4. Develop the state-assigned state table.
5. Derive a truth table for the combinational logic block.
6. Find a suitable circuit implementation for the combinational logic block.

Example

As a further example of a finite state machine that has both inputs and outputs, consider a coin-operated vending machine. For simplicity, let us assume that the machine accepts only quarters and dimes. The quarters or dimes are applied as inputs until a total of 30 cents or more is deposited. When this total is reached, an output (merchandise) is provided. No change is provided if more than 30 cents is deposited. Let binary inputs x_1 and x_2 represent coins being deposited, such that $x_1 = 1$ or $x_2 = 1$ if a quarter or a dime is deposited, respectively. Otherwise, these inputs are equal to 0. Only one coin is deposited at a time, so that input combination $x_1x_2 = 11$ never occurs. Also, let a binary output z represent merchandise provided by the machine, such that $z = 0$ for no merchandise and $z = 1$ for merchandise provided.

The first task in designing a logic circuit for the vending machine is to draw a state diagram or a state table. It is best to give a word description of each state needed and then decide later how many flip-flops will be needed to represent the required number of states. The states represent the total amount of money deposited at any point in the process. Based on the fact that dimes or quarters can be deposited in any order until the



$x_1 = 1$ – quarter deposited
$x_2 = 1$ – dime deposited
$z = 1$ – dispense merchandise
(i.e., a total of 30 cents deposited)
Input combination $x_1x_2 = 11$ cannot occur

Figure A.53 State diagram for the vending machine example.

total is equal to or greater than 30 cents, the states needed are:

S0 = nothing deposited (the "start" state)
S1 = 10 cents
S2 = 20 cents
S3 = 25 cents

We do not need any more states because when the present state is either S2 or S3, either a dime or a quarter will suffice as the present input to generate the $z = 1$ output and move to state S0 to start again.

A state diagram description of the desired behavior for the vending machine is given in Figure A.53. Note that the input $x_1x_2 = 11$ does not appear because both a quarter and a dime cannot be deposited at the same time. Also notice that each state has an arrow looping back to itself labeled 00/0. This indicates that if no coins are being deposited during a clock cycle, the circuit stays in its present state.

Only four states are needed for this machine. This will require two flip-flops. If we label them y_2 and y_1, and assign their values to represent the states as S0 = 00, S1 = 01, S2 = 10, and S3 = 11, Figure A.54 shows the resultant assigned state table.

	Present state	Next state				Output z			
		$x_1x_2 = 00$	$x_1x_2 = 01$	$x_1x_2 = 10$	$x_1x_2 = 11$	$x_1x_2 = 00$	$x_1x_2 = 01$	$x_1x_2 = 10$	$x_1x_2 = 11$
	y_2y_1	Y_2Y_1	Y_2Y_1	Y_2Y_1	Y_2Y_1				
S0	0 0	0 0	0 1	1 1	-	0	0	0	-
S1	0 1	0 1	1 0	0 0	-	0	0	1	-
S2	1 0	1 0	0 0	0 0	-	0	1	1	-
S3	1 1	1 1	0 0	0 0	-	0	1	1	-

Figure A.54 Assigned state table for the vending machine example.

We have used dashes in the table to indicate that the input combination of $x_1x_2 = 11$ does not appear. These entries are don't-care conditions, which we can take advantage of in the design of the combinational logic block, which will be discussed next.

This completes the first four steps in the synthesis procedure. We now go to step 5. The assigned state table in Figure A.54 leads directly to the truth table in Figure A.55, which specifies the functions of the combinational logic block. From the table, it is easy to derive the following expressions that give the implementation of the logic block:

$$Y_2 = \bar{x}_1\bar{x}_2y_2 + x_2\bar{y}_2y_1 + x_1\bar{y}_2\bar{y}_1$$

$$Y_1 = \bar{x}_1\bar{x}_2y_1 + \bar{y}_2\bar{y}_1(x_1 + x_2)$$

$$z = y_2(x_1 + x_2) + x_1y_1$$

Observe that the logic terms $\bar{x}_1\bar{x}_2$, $\bar{y}_1\bar{y}_2$, and $(x_1 + x_2)$ appear in more than one of the expressions. This leads to a cost saving in the implementation of the block.

Sequential circuits can easily be implemented with PALs, CPLDs and FPGAs because these devices contain flip-flops as well as combinational logic gates. Modern computer-aided design tools can be used to synthesize sequential circuits directly from a specification given in terms of a state diagram.

Note that the next-state and output entries in Figure A.54 are the same for states S2 and S3 for all input combinations where a change of state occurs. This implies that two different states are not really needed to represent the totals 20 cents and 25 cents. One state would be sufficient, because from either of these total deposits, the next coin deposited will cause merchandise to be provided ($z = 1$) and will cause a return to the starting state S0. Thus, states S2 and S3 are *equivalent* and can be replaced by a single state. This means that only three states are needed to implement the machine. Two flip-flops are still required. However, in more general situations, a reduction in the number of states through state equivalences often leads to fewer flip-flops and simpler circuits.

Another economy that can be achieved in implementing sequential circuits is in the combinational logic required. Different state assignments will lead to different logic

x_1	x_2	y_2	y_1	Y_2	Y_1	z
0	0	0	0	0	0	0
0	0	0	1	0	1	0
0	0	1	0	1	0	0
0	0	1	1	1	1	0
0	1	0	0	0	1	0
0	1	0	1	1	0	0
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	1	1	0
1	0	0	1	0	0	1
1	0	1	0	0	0	1
1	0	1	1	0	0	1
1	1	0	0	d	d	d
1	1	0	1	d	d	d
1	1	1	0	d	d	d
1	1	1	1	d	d	d

Figure A.55 Combinational logic specification for the vending machine circuit.

specifications, some of which may require fewer gates than others. We will not develop these ideas further, but the reader should appreciate that there are many interesting aspects to the economical design and implementation of sequential circuits.

Finally, we should note that other types of flip-flops can be used to represent state variables. We have used the D flip-flops here to keep the presentation as simple as possible. By using other, more flexible types, such as JK flip-flops, the required combinational logic can sometimes be reduced. See Problems A.35 and A.36 for an exploration of this possibility.

The preceding introduction to sequential circuits is based on the type of circuits that operate under the control of a clock. It is also possible to implement sequential circuits without using a clock. Such circuits are called *asynchronous sequential circuits*. Their design is not as straightforward as that of the synchronous sequential circuits. For a

complete treatment of both types of sequential circuits, consult one of many books that specialize in logic design [1, 3, 7–11].

A.14 CONCLUDING REMARKS

The main purpose of this appendix is to acquaint the reader with the basic concepts in logic design and to provide an indication of the circuit configurations commonly used in the construction of computer systems. Familiarity with this material will lead to a much better understanding of the architectural concepts discussed in the main chapters of the book. As we have said in several places, the detailed design of logic networks is done with the help of CAD tools. These tools take care of many details and can be used very effectively by a knowledgeable designer.

IC technology and CAD tools have revolutionized logic design. A variety of IC components are commercially available at ever-decreasing costs, and new developments and technological improvements are constantly occurring. In this appendix, we introduced some of the basic components that are useful in the design of digital systems.

From the designer's point of view, the important parameters are the cost and speed of the resultant circuits. Both of these measures are improved by making the number of IC packages used as low as possible. This can be achieved if large chips are used, which are capable of implementing complex logic networks on a single chip. In particular, the CPLD and FPGA devices offer effective solutions in many applications.

Two other design objectives are becoming increasingly important. The ability to easily test the resultant circuits simplifies both the task of proving that newly produced equipment works correctly and the task of repairing it when it fails. Furthermore, it is often desirable to increase the reliability of a system with the help of additional, redundant logic circuits (for example, by duplicating some parts). Both of these objectives are likely to lead to increased component cost. It is the designer's job to arrive at a satisfactory trade-off between these considerations. A number of books are available that deal with the subject of testing and fault tolerance [1, 12–17].

PROBLEMS

A.1 Implement the COINCIDENCE function in sum-of-products form, where COINCIDENCE = $\overline{\text{XOR}}$.

A.2 Prove the following identities by using algebraic manipulation and also by using truth tables.

(*a*) $\overline{a \oplus b} \oplus c = \bar{a}\bar{b}\bar{c} + ab\bar{c} + \bar{a}bc + a\bar{b}c$

(*b*) $x + w\bar{x} = x + w$

(*c*) $x_1\bar{x}_2 + \bar{x}_2x_3 + x_3\bar{x}_1 = x_1\bar{x}_2 + x_3\bar{x}_1$

A.3 Derive minimal sum-of-products forms for the four 3-variable functions f_1, f_2, f_3, and f_4 given in Figure PA.1. Is there more than one minimal form for any of these functions? If so, derive all of them.

x_1	x_2	x_3	f_1	f_2	f_3	f_4
0	0	0	1	1	d	0
0	0	1	1	1	1	1
0	1	0	0	1	0	1
0	1	1	0	1	1	d
1	0	0	1	0	d	d
1	0	1	0	0	0	d
1	1	0	1	0	1	1
1	1	1	1	1	1	0

Figure PA.1 Logic functions for Problem A.3.

A.4 Find the simplest sum-of-products form for the function f using the don't-care condition d, where

$$f = x_1(x_2\bar{x}_3 + x_2x_3 + \bar{x}_2\bar{x}_3x_4) + x_2\bar{x}_4(\bar{x}_3 + x_1)$$

and

$$d = x_1\bar{x}_2(x_3x_4 + \bar{x}_3\bar{x}_4) + \bar{x}_1\bar{x}_3x_4$$

A.5 Consider the function

$$f(x_1, \ldots, x_4) = (x_1 \oplus x_3) + (x_1x_3 + \bar{x}_1\bar{x}_3)x_4 + x_1\bar{x}_2$$

(*a*) Use a Karnaugh map to find a minimum cost sum-of-products (SOP) expression for f.

(*b*) Find a minimum cost SOP expression for $\bar{f}$, which is the complement of f. Then, complement (using de Morgan's rule) this SOP expression to find an expression for f. The resulting expression will be in the product-of-sums (POS) form. Compare its cost with the SOP expression derived in Part *a*. Can you draw any general conclusions from this result?

A.6 Find a minimum cost implementation of the function $f(x_1, x_2, x_3, x_4)$, where $f = 1$ if either one or two of the input variables have the logic value 1. Otherwise, $f = 0$.

A.7 Figure A.6 defines the 4-bit encoding of BCD digits. Design a circuit that has four inputs labeled $b_3, \ldots, b_0$, and an output f, such that $f = 1$ if the 4-bit input pattern is a valid BCD digit; otherwise $f = 0$. Give a minimum cost implementation of this circuit.

A.8 Two 2-bit numbers $A = a_1a_0$ and $B = b_1b_0$ are to be compared by a four-variable function $f(a_1, a_0, b_1, b_0)$. The function f is to have the value 1 whenever

$$v(A) \leq v(B)$$

where $v(X) = x_1 \times 2^1 + x_0 \times 2^0$ for any 2-bit number. Assume that the variables A and B are such that $|v(A) - v(B)| \leq 2$. Synthesize f using as few gates as possible.

A.9 Repeat Problem A.8 for the requirement that $f = 1$ whenever

$$v(A) > v(B)$$

subject to the input constraint

$$v(A) + v(B) \leq 4$$

A.10 Prove that the associative rule does not apply to the NAND operator.

A.11 Implement the following function with no more than six NAND gates, each having three inputs.

$$f = x_1x_2 + x_1x_2x_3 + \bar{x}_1\bar{x}_2\bar{x}_3x_4 + \bar{x}_1\bar{x}_2x_3\bar{x}_4$$

Assume that both true and complemented inputs are available.

A.12 Show how to implement the following function using six or fewer two-input NAND gates. Complemented input variables are not available.

$$f = x_1x_2 + \bar{x}_3 + \bar{x}_1x_4$$

A.13 Implement the following function as economically as possible using only NAND gates. Assume that complemented input variables are not available.

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

A.14 A number code in which consecutive numbers are represented by binary patterns that differ only in one bit position is called a Gray code. A truth table for a 3-bit Gray code to binary code converter is shown in Figure PA.2*a*.

(*a*) Implement the three functions f_1, f_2, and f_3 using only NAND gates.

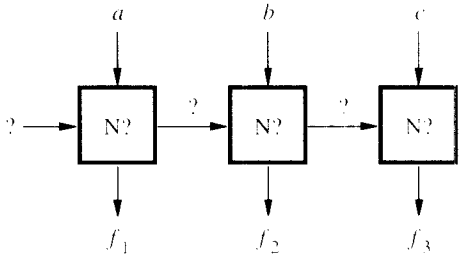
(*b*) A lower-cost network for performing this code conversion can be derived by noting the following relationships between the input and output variables.

$$f_1 = a$$
$$f_2 = f_1 \oplus b$$
$$f_3 = f_2 \oplus c$$

Using these relationships, specify the contents of a combinational network N that can be repeated, as shown in Figure PA.2*b*, to implement the conversion. Compare the total number of NAND gates required to implement the conversion in this form to the number required in Part *a*.

3-bit Gray code inputs			Binary code outputs		
a	b	c	f_1	f_2	f_3
0	0	0	0	0	0
0	0	1	0	0	1
0	1	1	0	1	0
0	1	0	0	1	1
1	1	0	1	0	0
1	1	1	1	0	1
1	0	1	1	1	0
1	0	0	1	1	1

(a) Three-bit Gray code to binary code conversion



(b) Code conversion network

Figure PA.2 Gray code conversion example for Problem A.14.

A.15 Implement the XOR function using only 4 two-input NAND gates.

A.16 Figure A.37 defines a BCD to seven-segment display decoder. Give an implementation for this truth table using AND, OR, and NOT gates. Verify that the same functions are correctly implemented by the NAND gate circuits shown in the figure.

A.17 In the logic network shown in Figure PA.3, gate 3 fails and produces the logic value 1 at its output F1 regardless of the inputs. Redraw the network, making simplifications

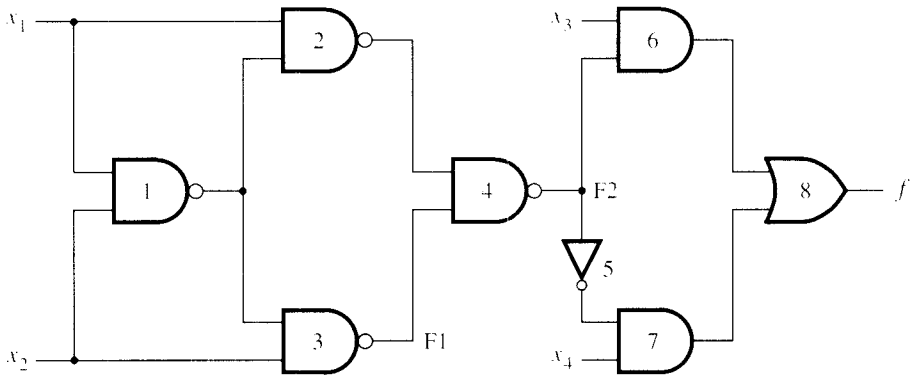


Figure PA.3 A faulty network.

wherever possible, to obtain a new network that is equivalent to the given faulty network and that contains as few gates as possible. Repeat this problem, assuming that the fault is at position F2, which is stuck at a logic value 0.

A.18 Figure A.16 shows the structure of a general CMOS circuit. Derive a CMOS circuit that implements the function

$$f(x_1, \ldots, x_4) = \bar{x}_1\bar{x}_2 + \bar{x}_3\bar{x}_4$$

Use as few transistors as possible. (Hint: Consider series/parallel networks of transistors. Note the complementary series and parallel structure of the pull-up and pull-down networks in Figures A.17 and A.18.)

A.19 Draw the waveform for the output Q in the JK circuit of Figure A.31, using the input waveforms shown in Figure PA.4 and assuming that the flip-flop is initially in the 0 state.

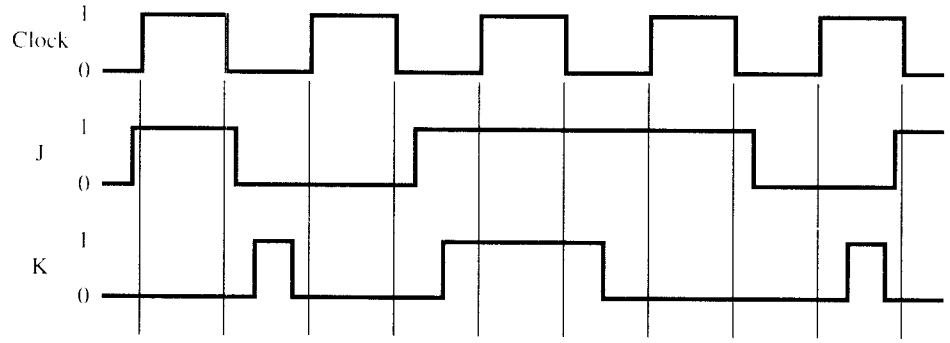


Figure PA.4 Input waveforms for a JK flip-flop.

A.20 Derive the truth table for the NAND gate circuit in Figure PA.5. Compare it to the truth table in Figure A.24*b* and then verify that the circuit in Figure A.26 is equivalent to the circuit in Figure A.25*a*.

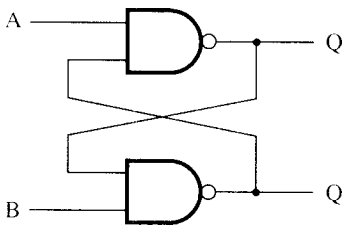


Figure PA.5 NAND latch.

A.21 Compute both the setup time and the hold time in terms of NOR gate delays for the negative edge-triggered D flip-flop shown in Figure A.29.

A.22 In the circuit of Figure A.27*a*, replace all NAND gates with NOR gates. Derive a truth table for the resulting circuit. How does this circuit compare with the circuit in Figure A.27*a*?

A.23 Figure A.33 shows a shift register network that shifts the data to the right one place at a time under the control of a clock signal. Modify this shift register to make it capable of shifting data either one or two places at a time under the control of the clock and an additional control input ONE/TWO.

A.24 A 4-bit shift register that has two control inputs -- INITIALIZE and RIGHT/LEFT — is required. When INITIALIZE is set to 1, the binary number 1000 should be loaded into the register independently of the clock input. When INITIALIZE = 0, pulses at the clock input should rotate this pattern. The pattern rotates right or left when the RIGHT/LEFT input is equal to 1 or 0, respectively. Give a suitable design for this register using D flip-flops that have preset and clear inputs as shown in Figure A.32.

A.25 Derive a three-input to eight-output decoder network, with the restriction that the gates to be used cannot have more than two inputs.

A.26 Figure A.35 shows a 3-bit up counter. A counter that counts in the opposite direction (that is, 7, 6, . . . , 1, 0, 7, . . .) is called a down counter. A counter capable of counting in both directions under the control of an UP/DOWN signal is called an up/down counter. Show a logic diagram for a 3-bit up/down counter that can also be preset to any state through parallel loading of its flip-flops from an external source. A LOAD/COUNT control is used to determine whether the counter is being loaded or is operating as a counter.

A.27 Figure A.35 shows an asynchronous 3-bit up-counter. Design a 4-bit synchronous up-counter, which counts in the sequence 0, 1, 2, . . . , 15, 0 . . . . Use T flip-flops in your circuit. In the synchronous counter all flip-flops have to be able to change their states at the same time. Hence, the primary clock input has to be connected directly to the clock inputs of all flip-flops.

A.28 A switching function to be implemented is described by the expression

$$f(x_1, x_2, x_3, x_4) = x_1x_3\bar{x}_4 + \bar{x}_1\bar{x}_3x_4 + \bar{x}_2\bar{x}_3\bar{x}_4$$

(*a*) Show an implementation of f in terms of an eight-input multiplexer circuit.

(*b*) Can f be realized with a four-input multiplexer circuit? If so, show how.

A.29 Repeat Problem A.28 for

$$f(x_1, x_2, x_3, x_4) = x_1\bar{x}_2x_3 + x_2x_3x_4 + \bar{x}_1\bar{x}_4$$

A.30 (*a*) What is the total number of distinct functions, $f(x_1, x_2, x_3)$, of three binary variables?

(*b*) How many of these functions are implementable with one PAL circuit of the type shown in Figure A.43?

(*c*) What is the smallest change in the circuit in Figure A.43 that should be made to allow any three-variable function to be implemented with a single PAL circuit?

A.31 Consider the PAL circuit in Figure A.43. Suppose that the circuit is modified by adding a fourth input variable, x_4, whose uncomplemented and complemented forms can be connected to all four AND gates in the same way as the variables x_1, x_2, and x_3.

(*a*) Can this modified PAL be used to implement the function

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

If so, show how.

(*b*) How many functions of three variables cannot be implemented with this PAL?

A.32 Complete the design of the up/down counter in Figure A.47 by using the state assignment S0 = 10, S1 = 11, S2 = 01, and S3 = 00. How does this design compare with the one given in Section A.13.1?

A.33 Design a 2-bit synchronous counter of the general form shown in Figure A.50 that counts in the sequence . . . , 0, 3, 1, 2, 0, . . . , using D flip-flops. This circuit has no external inputs, and the outputs are the flip-flop values themselves.

A.34 Repeat Problem A.33 for a 3-bit counter that counts in the sequence . . . , 0, 1, 2, 3, 4, 5, 0, . . . , taking advantage of the unused count values 6 and 7 as don't-care conditions in designing the combinational logic.

A.35 In Section A.13, D flip-flops were used in the design of synchronous sequential circuits. This is the simplest choice in the sense that the logic function values for a D input are directly determined by the desired next-state values in the state table. Suppose that JK flip-flops are to be used instead of D flip-flops. Describe, by the construction of a table, how to determine the binary value for each of the J and K inputs for a flip-flop as a function of each possible required transition from present state to next state for that flip-flop. (*Hint:* The table should have four rows, one for each of the transitions $0 \rightarrow 0$, $0 \rightarrow 1$, $1 \rightarrow 0$, and $1 \rightarrow 1$; and each J and K entry is to be 0, 1, or "don't care," as required.) Apply the information in your table to the design of individual combinational logic functions for each J and K input for each of the two flip-flops of the 2-bit binary counter of Problem A.33. How does the simplicity of the logic required compare to that needed for the design of the counter using D flip-flops?

A.36 Repeat Problem A.34 using JK flip-flops instead of D flip-flops. The general procedure for doing this is provided by the answer to Problem A.35.

A.37 In the vending machine example used in Section A.13.4 to illustrate the finite state machine model, a single binary output, z, was used to indicate the dispensing of merchandise. Change was not provided as an output. The purpose of this problem is to expand the output to include providing proper change. Assume that the only input sequences of dimes and quarters are: 10-10-10, 10-25, 25-10, and 25-25. Coincident with the last coin input, the outputs to be provided for these sequences are 0, 5, 5, and 20, respectively. Use two new binary outputs, z_2 and z_3, to represent the three distinct outputs. (This does not correspond directly to coins in use, but it keeps the problem simple.)

(*a*) Specify the new state table that incorporates the new outputs.

(*b*) Develop the logic expressions for the new outputs z_2 and z_3.

(*c*) Are there any equivalent states in the new state table?

A.38 Finite state machines can be used to detect the occurrence of certain subsequences in the sequence of binary inputs applied to the machine. Such machines are called *finite state recognizers*. Suppose that a machine is to produce a 1 as its output coincident with the second 1 in the pattern 011 whenever that subsequence occurs in the input sequence applied to the machine.

(*a*) Draw the state diagram for this machine.

(*b*) Make a state assignment for the required number of flip-flops and construct the assigned state table, assuming that D flip-flops are to be used.

(*c*) Derive the logic expressions for the output and the next-state variables.

A.39 Repeat Part *a* only of Problem A.38 for a machine that is to recognize the occurrence of either of the subsequences 011 and 010 in the input sequence, including the cases where overlap occurs. For example, the input sequence 110101011... is to produce the output sequence 000010101....

REFERENCES

1. S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design*, McGraw-Hill, Burr Ridge, IL, 2000.
2. A.S. Sedra and K.C. Smith, *Microelectronic Circuits*, 4th ed., Oxford, New York, 1998.
3. J.F. Wakerley, *Digital Design Principles and Practices*, Prentice Hall, Upper Saddle River, NJ, 2000.
4. J.H. Jenkins, *Designing with FPGAs and CPLDs*, Prentice-Hall, Englewood Cliffs, N.J., 1994.
5. S.M. Trimberger, *Field-Programmable Gate Array Technology*, Kluwer, Boston, 1994.
6. S.D. Brown, R.J. Francis, J. Rose, and Z.G. Vranesic, *Field-Programmable Gate Arrays*, Kluwer, Boston, 1992.
7. R.H. Katz, *Contemporary Logic Design*, Benjamin Cummings, Redwood City, Calif., 1994.
8. J.P. Hayes, *Digital Logic Design*, Addison-Wesley, Reading, Mass., 1993.
9. F.H. Hill and G.R. Peterson, *Computer Aided Logical Design with Emphasis on VLSI*, 4th ed., Wiley, New York, 1993.
10. C.H. Roth, *Fundamentals of Logic Design*, 4th ed., West, St. Paul, Minn., 1992.
11. M.M. Mano and C.R. Kime, *Logic and Computer Design Fundamentals*, Prentice-Hall, Upper Saddle River, N.J., 1997.
12. M. Abramovici, M.A. Breuer, and A.D. Friedman, *Digital Systems Testing and Testable Design*, revised edition, IEEE Press, New York, 1995.

13. V.N. Yarmolik, *Fault Diagnosis of Digital Circuits,* Wiley, Chichester, England, 1994.

14. P.K. Lala, *Digital System Design Using Programmable Logic Devices,* Prentice-Hall, Englewood Cliffs, N.J., 1990.

15. B.W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems,* Addison-Wesley, Reading, Mass., 1989.

16. A.J. Miczo, *Digital Logic Testing and Simulation,* Wiley, New York, 1986.

17. D.K. Pradhan, *Fault-Tolerant Computing,* Prentice-Hall, Englewood Cliffs, N.J., 1986.