CHAPTER

TWO

BOOLEAN FUNCTIONS AND GATES

In this chapter you'll learn a different technique for generating the same logic signals that were obtained from a ROM in the last chapter. Instead of ROMs you'll use logic circuits called gates. Gates are used instead of ROMs for three reasons:

1. Gates are usually faster than ROMs and are used for highest-speed digital circuitry.
2. Gates may be less expensive than ROMs for very small ASMs.
3. Gates are often used to supplement a ROM, usually reducing the size of ROM required.

Gate circuitry is more difficult to design, more expensive to build and test, and less flexible than ROM circuitry. The operation of gates may be described by *boolean algebra*, which is the next topic we discuss.

Logical Connectives

Logical or boolean connectives are operations performed on variables which may assume the value of 0 or 1. A variable which can assume only two values is called a *boolean variable.* Since digital signals can only assume two values, you can see that we'll be able to make a one-to-one correspondence between boolean variables and digital signals.

The AND Connective

The logical *AND* connective is symbolized by a dot between the boolean variables to be ANDed:

$$A \cdot B$$

An equal sign is used to assign the value of this expression or *function* to another variable:

$$C = A \cdot B$$

The value of boolean variable C is made equal to the AND function of variables A and B. However, we still don't know the value of this function. The expression $A \cdot B$ is 1 only if *both A and B* are 1. This may be expressed in tabular form:

$$C = A \cdot B$$

C	A	B
0	0	0
0	0	1
0	1	0
1	1	1

These tables are often called *truth tables.* More than two variables may be ANDed together:

$$F = A \cdot B \cdot C$$

The value of the resulting expression is 1 only if all the variables are 1. For the example above, F is 1 only if A is 1 *and* B is 1 *and* C is 1. The AND function is often abbreviated by omitting the dots:

$$ABC = A \cdot B \cdot C$$

If a variable has more than one letter, it must be separated by parentheses if the dots are omitted:

$$(RST)BC = RST \cdot B \cdot C$$

The OR Connective

The logical *OR* connective is symbolized by a plus sign between the variables to be ORed:

$$C = A + B$$

The expression $A + B$ is a 1 if A is a 1 *or* B is a 1 *or* both A and B are 1.

$$C = A + B$$

C	A	B
0	0	0
1	0	1
1	1	0
1	1	1

More than two variables may be ORed together:

$$F = A + B + C$$

The value of the resulting expression is 1 if one or more of the variables are one. In the example above, F is 1 when *at least* one of the variables A, B, or C is 1.

Logical Inversion

Logical *inversion* is symbolized by a horizontal line drawn over the variable to be inverted:

$$C = \overline{A}$$

The above expression is read *C equals A inverted* or *C equals A bar* or *C equals not A* or *C is the complement of A*. The value of the expression is 1 when A is 0 and is 0 when A is 1.

$$C = \overline{A}$$

C	A
1	0
0	1

Complex Expressions

The logical connectives may be used together in complex expressions. Parentheses are used to group subexpressions when confusion might exist. For example, the expression

$$F = A + (B \cdot C)$$

is different from the expression

$$F = (A + B) \cdot C$$

A common abbreviation is to omit parentheses when they surround an AND connective. For example, the following are equivalent:

$$A + (B \cdot C) = A + BC$$

As an example, let's evaluate the following function:

$$F = \overline{A + BC}$$

Of course, this is an abbreviation for the expression

$$F = (\overline{A + (B \cdot C)})$$

Evaluate this expression by listing all possible combinations of the three variables A, B, and C in a table. Then, evaluate and tabulate each subexpression, starting from the innermost parentheses.

A	B	C	BC	$A + BC$	F
0	0	0	0	0	1
0	0	1	0	0	1
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	1	0
1	0	1	0	1	0
1	1	0	0	1	0
1	1	1	1	1	0

In this case, we evaluate $B \cdot C$ first, then $A + BC$ and, finally, invert $A + BC$ to get F.

Logical Expressions from Tables

Any boolean expression can be evaluated as described in the previous section. It would be much more useful to be able to find a logical expression corresponding to a given table. Since we can generate a table of ROM contents from an ASM chart, we should like to be able to write the logical expressions corresponding to each ROM data output in that table. As an example, let's use the ROM contents for the simple traffic light controller which is shown in Table 2-1. Notice that we have renamed the state variables in this table. For example, state variable A_3 is now called D. This was done to simplify writing the expressions we shall generate. There are two methods we can use to get different but equivalent expressions for each of these ROM outputs. These methods are called sum of products and product of sums.

Table 2-1 Traffic controller ROM

Current state													
A_3 D	A_2 C	A_1 B	A_0 A	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	1	0	0	1	1	0	0
0	0	0	1	0	0	1	0	0	0	1	1	0	0
0	0	1	0	0	0	1	1	0	0	1	1	0	0
0	0	1	1	0	1	0	0	0	0	1	1	0	0
0	1	0	0	0	1	0	1	0	1	0	1	0	0
0	1	0	1	0	1	1	0	1	0	0	0	0	1
0	1	1	0	0	1	1	1	1	0	0	0	0	1
0	1	1	1	1	0	0	0	1	0	0	0	0	1
1	0	0	0	1	0	0	1	1	0	0	0	0	1
1	0	0	1	0	0	0	0	1	0	0	0	1	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0	0	0	0

Canonical Sum of Products

One boolean expression represents one output of the ROM. In Table 2-2, we've listed the state variables and ROM output D_7 for which we intend to find a logical expression. The expression we will find is called a *canonical sum of products* (SOP), although it has nothing to do with ordinary arithmetic. We start by finding each row in which the output D_7 is a 1. There are four such rows. For each of these rows, we write the AND expression of *all* four input variables that is 1 for that row. For example, the row corresponding to state 2 has an output of 1 for D_7. The variables in that row have the values $D = 0$, $C = 0$, $B = 1$, and $A = 0$. Thus, the AND expression that is 1 for that set of variables' values is:

$$\overline{D} \cdot \overline{C} \cdot B \cdot \overline{A}$$

Note that this expression is 1 for this particular combination of variables' values and *only* for this combination. Since there are four 1s in the D_7 column, we find four AND expressions, as shown in Table 2-3. These expressions are:

$$\begin{gathered}\overline{D} \cdot \overline{C} \cdot \overline{B} \cdot A \\ \overline{D} \cdot \overline{C} \cdot B \cdot \overline{A} \\ \overline{D} \cdot C \cdot \overline{B} \cdot A \\ \overline{D} \cdot C \cdot B \cdot \overline{A}\end{gathered}$$

Each of these expressions will evaluate to a 1 only for that combination of input variable values corresponding to one of the four rows in which D_7 is a 1. You can see we've written four AND expressions each of which generates a single 1 output of D_7. Since we want a *single* expression whose output is 1 whenever *any* of these 4 individual expressions is a 1, we will simply OR all four expressions together. Remember that an OR function is a 1 whenever any of its variables is a 1. We can now write the logical expression for D_7:

$$D_7 = \overline{D}\overline{C}\overline{B}A + \overline{D}\overline{C}B\overline{A} + \overline{D}C\overline{B}A + \overline{D}CB\overline{A}$$

This expression is a 1 for the four input variable combinations for which D_7 should be a 1 and is a 0 otherwise. We can write SOP expressions for each of the 10 ROM outputs, as shown in Table 2-3.

Table 2-2 Sum of products for D_7

D	C	B	A	D_7	AND function
0	0	0	0	0	
0	0	0	1	1	$\overline{D}\overline{C}\overline{B}A$
0	0	1	0	1	$\overline{D}\overline{C}B\overline{A}$
0	0	1	1	0	
0	1	0	0	0	
0	1	0	1	1	$\overline{D}C\overline{B}A$
0	1	1	0	1	$\overline{D}CB\overline{A}$
0	1	1	1	0	
1	0	0	0	0	
1	0	0	1	0	
1	0	1	0	0	
1	0	1	1	0	
1	1	0	0	0	
1	1	0	1	0	
1	1	1	0	0	
1	1	1	1	0	

$$D_7 = \overline{D}\overline{C}\overline{B}A + \overline{D}\overline{C}B\overline{A} + \overline{D}C\overline{B}A + \overline{D}CB\overline{A}$$

Table 2-3 All SOP expressions for traffic controller

$$\begin{aligned}
D_9 &= \overline{D}CBA + D\overline{C}\overline{B}\overline{A} \\
D_8 &= \overline{D}\overline{C}BA + \overline{D}C\overline{B}\overline{A} + \overline{D}C\overline{B}A + \overline{D}CB\overline{A} \\
D_7 &= \overline{D}\overline{C}\overline{B}A + \overline{D}\overline{C}B\overline{A} + \overline{D}C\overline{B}A + \overline{D}CB\overline{A} \\
D_6 &= \overline{D}\overline{C}\overline{B}\overline{A} + \overline{D}\overline{C}B\overline{A} + \overline{D}C\overline{B}\overline{A} + \overline{D}CB\overline{A} + D\overline{C}\overline{B}\overline{A} \\
D_5 &= \overline{D}C\overline{B}A + \overline{D}CB\overline{A} + \overline{D}CBA + D\overline{C}\overline{B}\overline{A} + D\overline{C}\overline{B}A \\
D_4 &= \overline{D}C\overline{B}\overline{A} \\
D_3 &= \overline{D}\overline{C}\overline{B}\overline{A} + \overline{D}\overline{C}\overline{B}A + \overline{D}\overline{C}B\overline{A} + \overline{D}\overline{C}BA \\
D_2 &= \overline{D}\overline{C}\overline{B}\overline{A} + \overline{D}\overline{C}\overline{B}A + \overline{D}\overline{C}B\overline{A} + \overline{D}\overline{C}BA + \overline{D}C\overline{B}\overline{A} \\
D_1 &= D\overline{C}\overline{B}A \\
D_0 &= \overline{D}C\overline{B}A + \overline{D}CB\overline{A} + \overline{D}CBA + D\overline{C}\overline{B}\overline{A}
\end{aligned}$$

Canonical Product of Sums

Another expression we can find from a truth table is the *canonical product of sums* (POS). To find the POS expression, we find all rows in the table that are 0 (rather than 1 as we did for SOP). Next we find the OR function that evaluates to 0 for each input combination for which D_7 is 0. This is shown in Table 2-4. For example, D_7 is 0 for the variable combination $D = 0$, $C = 1$, $B = 0$, and $A = 0$. The OR expression of all four variables that is 0 for these variables' values is:

$$D + \overline{C} + B + A$$

This expression is 0 for this combination of variables' values and only for this combination. We find a similar expression for each row in which D_7 is 0. Now we have 12 expressions, each of which is 0 for a single row of the table for D_7. Since we need a *single* expression which is 0 whenever any of these 12 expressions is 0, we simply AND the 12 expressions together, as shown in Table 2-4. You'll notice that, although this POS expression is equivalent to the SOP expression, it is much more complex. This is because the function D_7 has many more 0s than 1s.

Several techniques can be used to reduce the complexity of logical expressions. In this chapter we will examine *boolean identities* and *Karnaugh maps* (K maps).

Table 2-4 Product of sums for D_7

D	C	B	A	D_7	OR function
0	0	0	0	0	$D+C+B+A$
0	0	0	1	1	
0	0	1	0	1	
0	0	1	1	0	$D+C+\overline{B}+\overline{A}$
0	1	0	0	0	$D+\overline{C}+B+A$
0	1	0	1	1	
0	1	1	0	1	
0	1	1	1	0	$D+\overline{C}+\overline{B}+\overline{A}$
1	0	0	0	0	$\overline{D}+C+B+A$
1	0	0	1	0	$\overline{D}+C+B+\overline{A}$
1	0	1	0	0	$\overline{D}+C+\overline{B}+A$
1	0	1	1	0	$\overline{D}+C+\overline{B}+\overline{A}$
1	1	0	0	0	$\overline{D}+\overline{C}+B+A$
1	1	0	1	0	$\overline{D}+\overline{C}+B+\overline{A}$
1	1	1	0	0	$\overline{D}+\overline{C}+\overline{B}+A$
1	1	1	1	0	$\overline{D}+\overline{C}+\overline{B}+\overline{A}$

$$\begin{aligned} D_7 = {} & (D+C+B+A)\,(D+C+\overline{B}+\overline{A})\,(D+\overline{C}+B+A)\,(D+\overline{C}+\overline{B}+\overline{A}) \\ & (\overline{D}+C+B+A)\,(\overline{D}+C+B+\overline{A})\,(\overline{D}+C+\overline{B}+A)\,(\overline{D}+C+\overline{B}+\overline{A}) \\ & (\overline{D}+\overline{C}+B+A)\,(\overline{D}+\overline{C}+B+\overline{A})\,(\overline{D}+\overline{C}+\overline{B}+A)\,(\overline{D}+\overline{C}+\overline{B}+\overline{A}) \end{aligned}$$

BOOLEAN IDENTITIES

Boolean expressions may be simplified by a formal algebra. Boolean algebra is described by Hill and Peterson.[1] Rather than proceed with a similar, rigorous treatment of boolean algebra, we will present several important boolean identities and demonstrate their use. Important boolean identities are summarized in Table 2-5. Any of these identities may be proved correct by evaluating the expressions on both sides of the equal sign for all possible variable combinations. You'll notice that the identities are arranged in two columns. Identities opposite each other are called *duals.* That is, one identity may be derived from the other by exchanging AND and OR symbols, and 0s and 1s.

According to the distributive laws in Table 2-5, boolean expressions may be factored in two ways. The first way seems familiar because it is analogous to arithmetic factoring (as if the symbols + and · stood for addition and multiplication):

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

The second way seems unfamiliar since it would not be correct if we interpreted the symbols + and · as arithmetic rather than as logical connectives:

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

[1]Fredrick J. Hill and Gerald R. Peterson, *Switching Theory and Logical Design,* 2d ed., John

Table 2-5 Boolean identities

Identities involving one variable	
$\overline{(\overline{A})}=A$	$\overline{(\overline{A})}=A$
$A+0=A$	$A\cdot 1=A$
$A+1=1$	$A\cdot 0=0$
$A+A=A$	$A\cdot A=A$
$A+\overline{A}=1$	$\overline{A}\cdot A=0$
Commutative, associative, and distributive laws	
$A+B=B+A$	$AB=BA$
$A+B+C=(A+B)+C=A+(B+C)$	$ABC=(AB)\,C=A\,(BC)$
$(AB)+(AC)=A\,(B+C)$	$(A+B)\,(A+C)=A+(BC)$
Identities involving two or more variables	
$A+(AB)=A$	$A\,(A+B)=A$
$(A+\overline{B})\,B=AB$	$(A\overline{B})+B=A+B$
$AB+\overline{A}C+BC=AB+\overline{A}C$	$(A+B)\,(\overline{A}+C)\,(B+C)=(A+B)\,(\overline{A}+C)$
De Morgan's laws	
$\overline{(A+B+C\cdots)}=\overline{A}\cdot\overline{B}\cdot\overline{C}\cdot\ldots$	$\overline{ABC\cdots}=\overline{A}+\overline{B}+\overline{C}+\cdots$

Using the first form above, the SOP expression for D_7 may be written as

$$D_7=\overline{D}\overline{C}\,(\overline{B}A+B\overline{A})+\overline{D}C\,(\overline{B}A+B\overline{A})$$

Notice we still have a common factor and can apply the first form of factoring twice to get

$$\begin{aligned} D_7 &= (\overline{D}\overline{C}+\overline{D}C)\,(\overline{B}A+B\overline{A}) \\ &= \overline{D}\,(\overline{C}+C)\,(\overline{B}A+B\overline{A}) \end{aligned}$$

From Table 2-5, we know that $\overline{C}+C$ is always 1, so that

$$D_7=\overline{D}\cdot 1\cdot(\overline{B}A+B\overline{A})$$

Now 1 ANDed with any variable is just that variable so that

$$D_7=\overline{D}\,(\overline{B}A+B\overline{A})$$

As you might expect, considerable practice is required to determine which identities to use to reduce a complex boolean expression. Also, several different but equivalent reductions might be possible. For these reasons, formal boolean manipulations are usually used for quick reduction of simple expressions. More systematic methods are used for complex expressions. One useful systematic method is the Karnaugh map.

KARNAUGH MAPS

Examine the logical expression

$$\overline{A}BC+\overline{A}B\overline{C}$$

Note that it can be factored into

$$\overline{A}B\,(C+\overline{C})$$

which is simply reduced to

$$\overline{A}B$$

In other words, we have been able to eliminate the variable C entirely and combine two terms into one. We were able to eliminate the variable C because all the other variables *except* C were identical in both terms. We could factor out all the identical variables and find a $C+\overline{C}$ term remaining.

An analogous situation exists for the logical expression

$$(\overline{A}+B+C)\,(\overline{A}+B+\overline{C})$$

which can be factored to

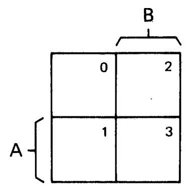
$$(\overline{A}+B)+C\cdot\overline{C}$$

From Table 2-5, we note that $C\cdot\overline{C}$ is always 0, and 0 ORed with any variable is simply that variable. Then, this expression can be reduced to

$$(\overline{A}+B)$$

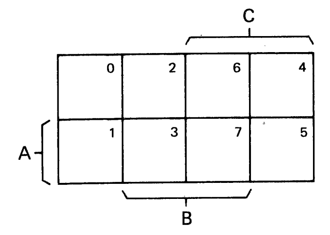
We've again been able to eliminate one variable completely because the original two terms differed in only one variable.

Karnaugh maps (K maps) are figures on which boolean expressions may be plotted. K maps are arranged so that terms that differ by only a single variable are plotted adjacently. We showed above that terms that differ in only one variable may be combined and the differing variable eliminated. We will find that, because adjacent terms on a K map differ by only one variable, simplified expressions may be written directly from these maps. The K maps for two, three, four, and five variables are shown in Figs. 2-1, 2-2, 2-3, and 2-4, respectively. Each square is given a decimal number corresponding to a specific combination of variable values, as listed in each figure. In addition, a bracket and letter denote those areas of the map where a particular variable is 1. For example, in Fig. 2-3, the B and bracket denote that the variable B is a 1 in squares 3, 2, 7, 6, 15, 14, 11, and 10. The variable B is a zero in the remaining squares. As a convention in this book, we'll use the first occurring letter of the alphabet as the least significant variable on a K map. That is, the binary number formed by a particular combination of the variables $DCBA$ corresponds to the decimal numbers used on the



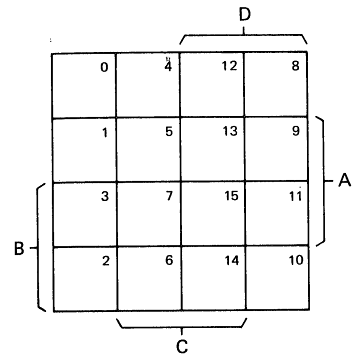
Square number	Variable values B	Variable values A
0	0	0
1	0	1
2	1	0
3	1	1

Figure 2-1 Two-variable Karnaugh map.



Square number	Variable values C	B	A
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Figure 2-2 Three-variable Karnaugh map.



Square number	Variable values D	C	B	A
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

Figure 2-3 Four-variable Karnaugh map.

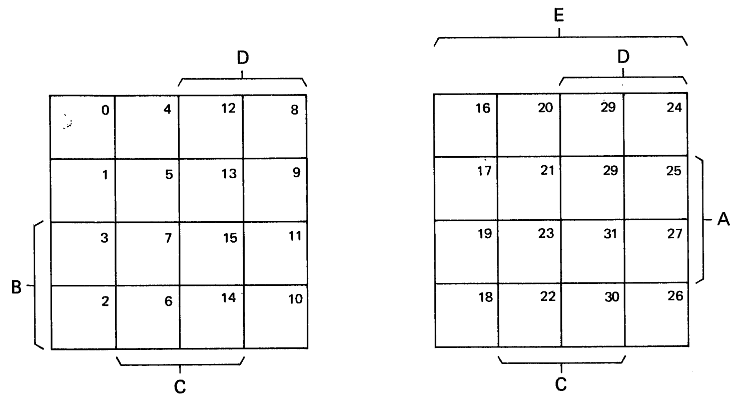


Figure 2-4 Five-variable Karnaugh map.

maps. If $DCBA = 1101$, that variable combination corresponds to square 13 on the K map. We'll use this four-variable K map as an example from here on. Adjacent squares on the map differ by only one variable. Thus, squares 7 and 3 differ only in variable C, squares 7 and 5 in variable B, squares 7 and 15 in variable D, and squares 7 and 6 in variable A. Diagonals are *not* considered adjacent and differ by more than one variable. Thus, squares 7 and 13 are *not* adjacent as they differ in both variables B and D. Squares along the edges of the map are also adjacent to the square along the opposite edge. Thus, square 1 is adjacent to square 9 besides 0, 5, and 3. Also, square 2 is adjacent to squares 0 and 10, as well as 3 and 6.

Using the K Map

Let's use function D_7 from Table 2-1 as an example. We'll enter a 1 into each square on a K map where the function D_7 happens to be 1. That is, each square corresponding to a variable combination that should make D_7 a 1 will have a 1 marked in it, as shown in Fig. 2-5. Since squares 1 and 5 each have a 1 in them and are adjacent, we know that the AND expressions for these 1s will differ by only a single variable. We'll circle these adjacent 1s to indicate that we should be able to combine these terms into a single AND expression with one less variable. Similarly, we'll do the same to squares 2 and 6. Now we can write a reduced expression simply by reading it off the K map! Squares 1 and 5 tell us that D_7 should be 1 if $B = 0$ (both squares are outside the B bracketed area), if $A = 1$ (both squares are inside the A bracketed area), and if $D = 0$ (both squares are outside the D bracketed area). The AND expression that is a 1 for this combination of variable values is:

$$A\bar{B}\bar{D}$$

Note that the variable C does not appear because the area circled (squares 1 and 5) is half inside and half outside of the area bracketed by C. The circle around squares 2 and 6 represents the term

$$\bar{A}B\bar{D}$$

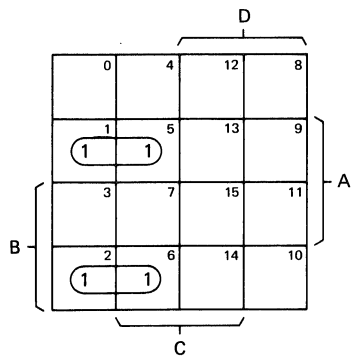


Figure 2-5 Karnaugh map for traffic light D_7.

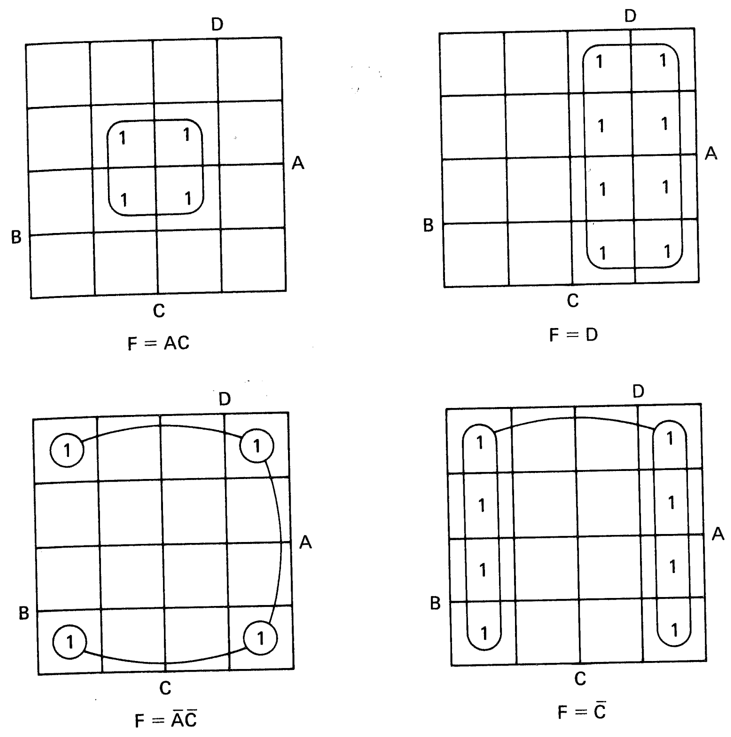


Figure 2-6 Karnaugh maps with groups having more than two terms.

These expressions are ORed together to generate a function that is 1 whenever any of the AND expressions is 1. This is the reduced SOP expression for D_7:

$$D_7 = A\bar{B}\bar{D} + \bar{A}B\bar{D}$$

K maps are even more versatile. Any group of *four* mutually adjacent squares can be combined into one term with two-fewer variables, any group of *eight* mutually adjacent squares into one term with three-fewer variables, etc. Figure 2-6 shows some examples of combining more than two squares. In each case, the boolean expression may be read from the K map by analyzing the circled areas. If the circled area is completely contained in the area bracketed by a variable, that variable will appear in the AND expression. If the circled area is completely outside an area bracketed by a variable, that variable will appear complemented in the AND expression. If a circled area overlaps areas bracketed and not bracketed by a variable, that variable will not appear in the AND expression.

K-Map Product of Sums

The result obtained in the example above was a sum-of-products representation. A product-of-sums representation can be obtained by plotting the 0s on the K map,

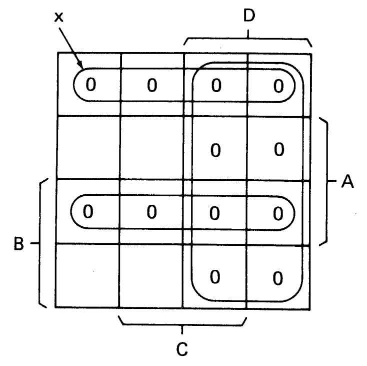


Figure 2-7 Product of sums Karnaugh map for D_7.

rather than the 1s of the function. Figure 2-7 shows the 0s of D_7 plotted and adjacent 0s circled in groups. Note that we've circled each 0 at least once and four of the 0s are in two groups. We wanted to circle as large a group as we possibly could to eliminate the most variables and have the fewest terms. It doesn't hurt to circle a square twice; if two terms of the function are 0, the function will still be 0. Likewise, we may circle 1s twice on a SOP map if it is advantageous to do so.

The group of 0s marked X in Fig. 2-17 tells us that we need an OR function that is 0 whenever $B = 0$ and $A = 0$. This function is:

$$A + B$$

Note that C and D do not appear because this group overlaps both the $C = 0$ and $C = 1$ areas and the $D = 0$ and $D = 1$ areas. Be careful to notice that in the POS solution, a 0 group *inside* a variable area causes that variable to appear in the expression *with* an inversion. This is exactly opposite to the SOP solution, in which a 1 group appearing *inside* a variable area causes that variable to appear *without* an inversion. Complete the expressions for the remaining two groups, and AND them together to get a function that is zero whenever any OR expression is zero. This is the reduced POS expression for D_7:

$$D_7 = (A + B)(\overline{A} + \overline{B})(\overline{D})$$

This is much simpler than the original POS expression for D_7 which had 12 terms, each having four variables.

The Best K Map

Often there are several ways to circle K-map groups. Each way may be equally acceptable, subject to a few rules:

1. Every term must be circled at least once, even if it is the only term in the circle, as shown in Fig. 2-8.
2. Circled groups must contain 2, 4, 8, 16, etc., terms that are mutually adjacent. Some *incorrect* groups are shown in Fig. 2-9.

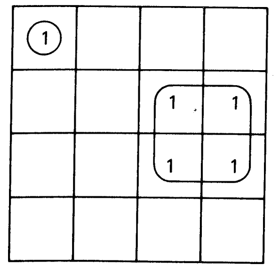


Figure 2-8 Karnaugh map with a group having only one term.

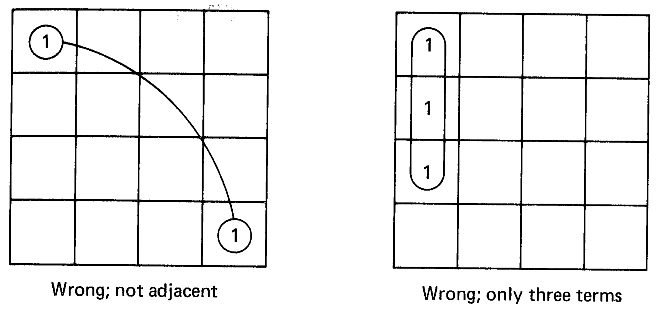


Figure 2-9 Karnaugh maps with incorrect groups.

3. Circled groups should be as large as possible, as shown in Fig. 2-10, even if a term must be in more than one group.
4. Once all the terms are circled in as large a group as possible, do not add additional groups, as shown in Fig. 2-11.

A good starting point is to circle all terms for which only one choice of group exists.

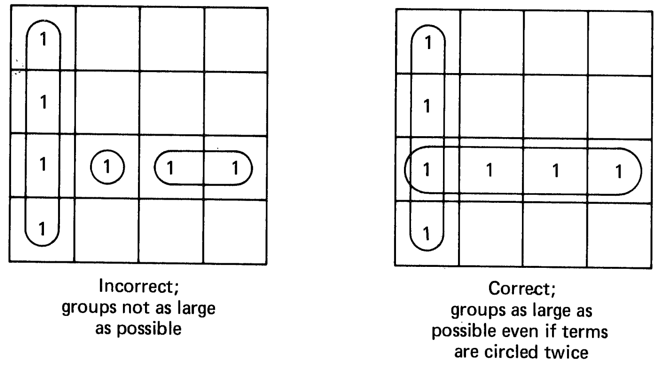
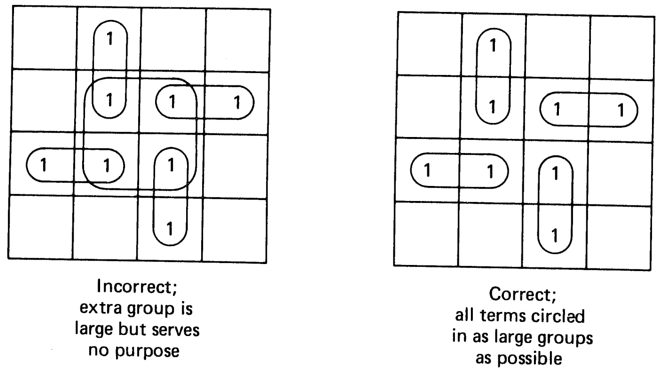


Figure 2-10 Karnaugh map group size.



Incorrect; extra group is large but serves no purpose
Correct; all terms circled in as large groups as possible

Figure 2-11 Redundant group.

Don't Cares on K Maps

You'll remember that, in the traffic controller of Table 2-1, all states were not used. States AH to FH weren't required to implement the traffic light. We arbitrarily assigned the contents of the ROM words associated with these states to be 0s. These table entries are really don't cares, and not 0s, and should be entered on the K map as don't cares. Don't cares are represented by the letter *X* on a K map. This is shown in Fig. 2-12 for D_7. Since we really don't care what these values are, we will assign values to them so that we can enlarge existing groups on the K map.

In Fig. 2-12, we will assign the value 1 to boxes 14 and 10 so that we can expand the bottom group to include four terms. This will eliminate an additional variable from one AND expression. We'll leave the don't cares in boxes 12, 13, and 15 as 0s; so we need not add additional groups to include them. We don't actually change the *X*'s on the map. We just know that we can add them to groups when convenient (10 and 14), but also can leave them uncircled when convenient (12, 13, and 15). The expression for D_7 in Fig. 2-12 is the same as that for D_7 in Fig. 2-5 for those input combinations that are actually used by the traffic light controller. These expressions are *not* equivalent for input combinations 10 and 14.

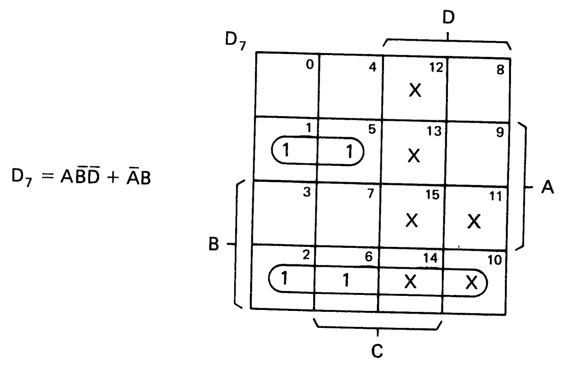


Figure 2-12 Karnaugh map don't cares.

Now we can derive boolean expressions from tables and use K maps to reduce those expressions. However, reduced boolean expressions are little use to us unless we can implement them with logic circuits. We could have simply stored the original table in a ROM.

GATES

Logic circuits exist that implement boolean connectives. In fact, the three basic connectives we've used are available as integrated circuits. The *AND gate* generates a logic signal output that is a 1 only if all its inputs are 1s. The *OR gate* generates a logic signal output that is a 1 whenever one or more of its inputs are a 1. The *inverter* generates a 1 output whenever its single input is a 0 and a 0 output whenever its input is a 1. The symbols for these circuits are shown in Fig. 2-13. Just as the AND and OR connectives could be applied to more than two inputs, AND and OR gates can have two or more inputs. Electrical, physical, and economic factors restrict the number of inputs available on a single gate. For one common logic family, 13 inputs are available on one gate, but two, three, and four input gates are much more common. The number of inputs of a gate is called the *fan in*.

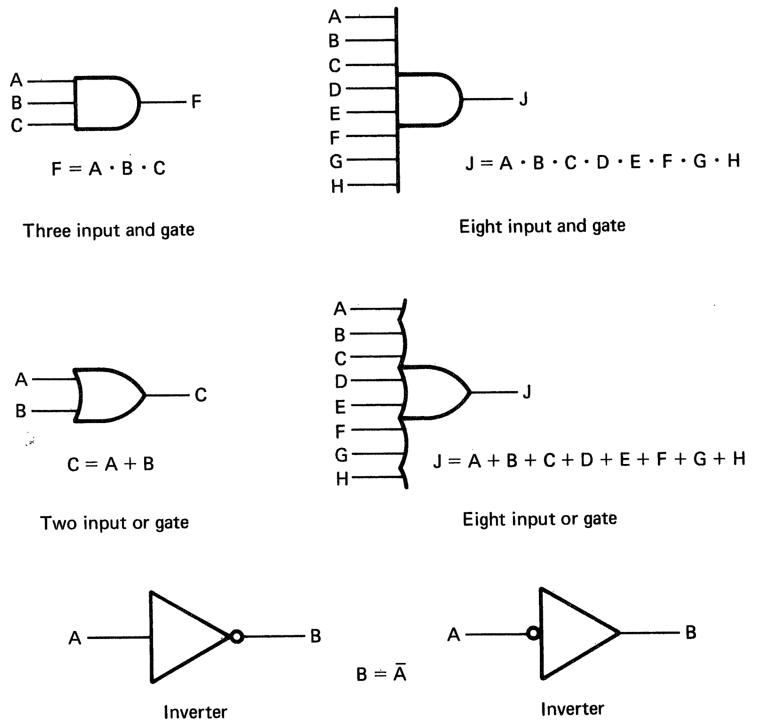


Figure 2-13 Gate symbols.

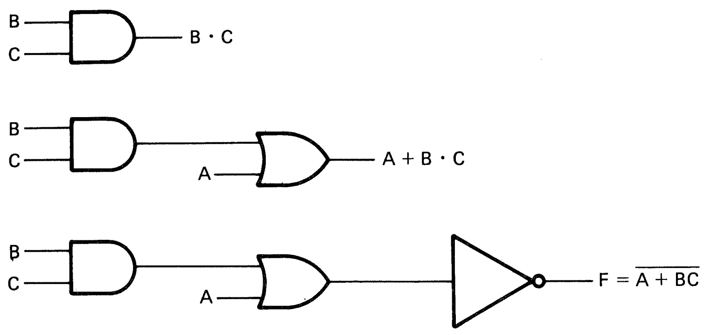


Figure 2-14 Implementing logical expressions with gates.

Gates and inverters can be used to implement logical expressions directly. Consider the expression

$$F = \overline{(A + (BC))}$$

Just as when evaluating logical expressions, we start implementing boolean expressions from the innermost parentheses. In this case, the function BC is implemented first with an AND gate, as shown in Fig. 2-14. Next, this BC output is connected to an OR gate, along with A, to form $A + BC$. Finally, an inverter forms the complement of the output of the OR gate to generate the desired function. Figure 2-15 shows another example. Note several important techniques used in this diagram. First, the inverters for the input variables are drawn to the side to make the drawing more readable. Each inverter output is labeled to indicate to what inputs on the diagram it is connected. Second, even though $\overline{D}$ is used twice, only one inverter is needed as its output may be connected to two inputs. A single output may be connected to several inputs; the maximum number of inputs is specified by the fan out of the logic circuit family. Usually 10 or more inputs may be connected to a single output. Third, two functions F and G are generated. They share a common term $\overline{A}B$, which need be implemented only once.

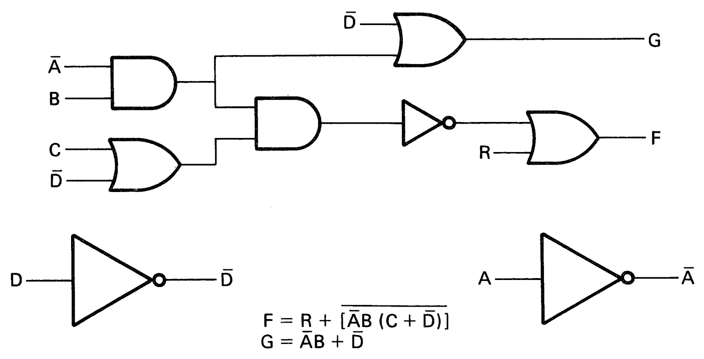

$\overline{D}$
G
$\overline{A}$
B
C
$\overline{D}$
R
F
D
$\overline{D}$
A
$\overline{A}$
F = R + [$\overline{\overline{A}B\,(C + \overline{D})}$]
G = $\overline{A}$B + $\overline{D}$

Figure 2-15 Gate-realization example.

Gate Implementation of a ROM

Since our logical expressions originally were obtained from a ROM table, a gate realization of these expressions should work the same as the ROM specified by the original table. Consider the ROM specified by Table 2-1. To implement this table with gates, draw the K maps corresponding to each of the 10 functions. Each function corresponds to one of the ROM outputs D_0 to D_9. Reduce the K maps using either SOP or POS. The K maps are shown in Fig. 2-16. The SOP functions are:

$$D_9 = ABC\overline{D} + \overline{A}\,\overline{B}\,\overline{C}D$$
$$D_8 = AB\overline{C}\,\overline{D} + \overline{A}C\overline{D} + \overline{B}C\overline{D}$$
$$D_7 = A\overline{B}\,\overline{D} + \overline{A}B\overline{D}$$
$$D_6 = \overline{A}\,\overline{D} + \overline{A}\,\overline{B}\,\overline{C}$$
$$D_5 = BC\overline{D} + AC\overline{D} + \overline{B}\,\overline{C}D$$
$$D_4 = \overline{A}\,\overline{B}C\overline{D}$$
$$D_3 = \overline{C}\,\overline{D}$$
$$D_2 = \overline{C}\,\overline{D} + \overline{A}\,\overline{B}\,\overline{D}$$
$$D_1 = A\overline{B}\,\overline{C}D$$
$$D_0 = BC\overline{D} + AC\overline{D} + \overline{A}\,\overline{B}\,\overline{C}D$$

Draw the gate realizations, shown in Fig. 2-17, corresponding to each of these reduced expressions. Since the gate realization of Fig. 2-17 should behave identically to the ROM, we should be able to connect the gates to inputs, outputs, and flip-flops, as shown in Fig. 2-18, to implement a traffic light controller. The ROM and gate realizations are identical for all input combinations that are actually used. Unused input combinations had 0 outputs in the ROM realization. Unused input combinations could have been specified as don't cares and assigned a value of either 0 or 1, as convenient, in the gate realization. If we had used don't cares, these two "equivalent" circuits would *not* behave identically if a combination of unused state variables were to accidentally occur.

Designing a gate implementation requires many more steps than designing an equivalent ROM implementation. In addition, many logic circuits need to be interconnected in the gate implementation, while only a single ROM circuit may be all that is needed in the alternate implementation. Then, why should gate implementations be used? Several factors enter into this decision. Some are tabulated in Table 2-6. Often, cost is most important. Especially for very simple circuits, a gate realization may be less expensive than a ROM. Computing cost is very difficult and must include fabrication, power supply, cooling, testing, and other costs that are attributable to each implementation. These associated costs may vary widely and are often quite large in relation to the cost of the logic circuits themselves. For this reason, comparing parts costs alone is not sufficient.

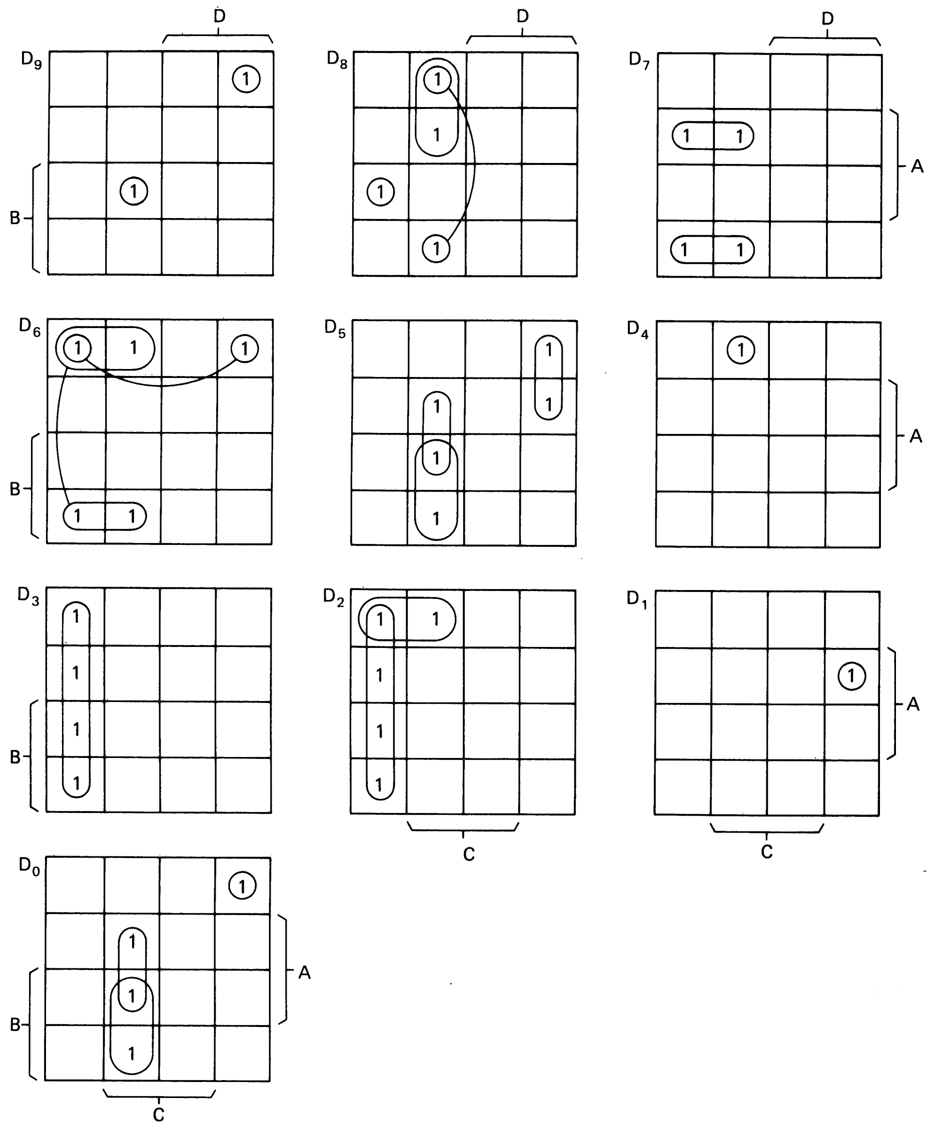


Figure 2-16 Reduced functions for traffic controller.

Trends in most factors affecting the choice between gate and ROM realizations are favoring the use of ROMs. However, gates are often used in conjunction with ROMs to reduce the cost of the ROM-implemented circuit. Gates may be used to reduce the number of ROM outputs required. This is especially important since the number of ROM outputs available is often a multiple of 4 or 8. Thus, a ROM implementation requiring 10 outputs often must be implemented with a 12- or even 16-output ROM. We've seen that the size, and cost, of a ROM are strongly related to the

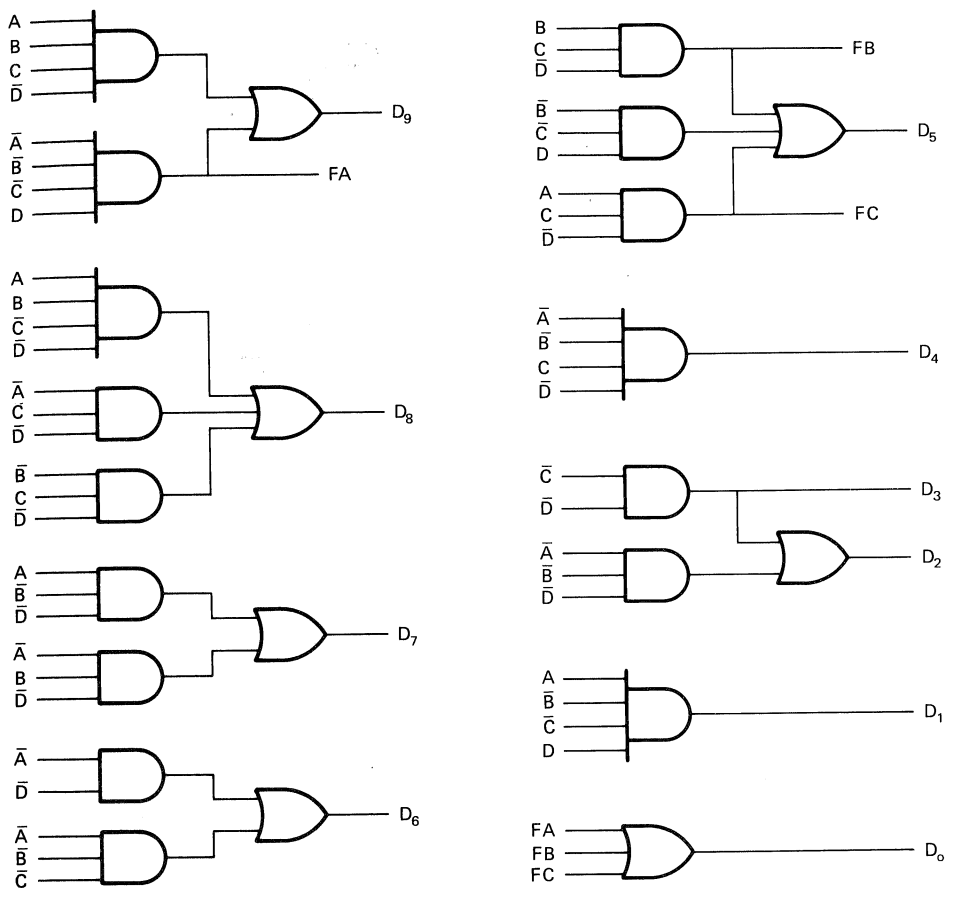


Figure 2-17 Gate realizations of logic expressions for traffic light.

number of inputs or address lines. Eliminating only one address input of a ROM will reduce the size of the ROM by half. Later, we will learn how to use gates to reduce both the number of address inputs and the number of data outputs required of a ROM.

Symbology and Graphical Techniques

A small circle like that used in the inverter symbol may be appended to any logic circuit symbol's input or output. The new circuit performs a new function. The input or output with the circle (called an *inversion bubble*) behaves as if an inverter were connected to that signal. Figure 2-19 shows some examples of this notation. Also, the type D flip-flop shown has an additional output called $\overline{Q}$. Many flip-flops have the complement of the output signal Q available. This often eliminates an additional external inverter. In order to implement a circuit with available parts, it is often necessary to move, add, or delete inverters or inversion bubbles, as shown in Fig. 2-20.

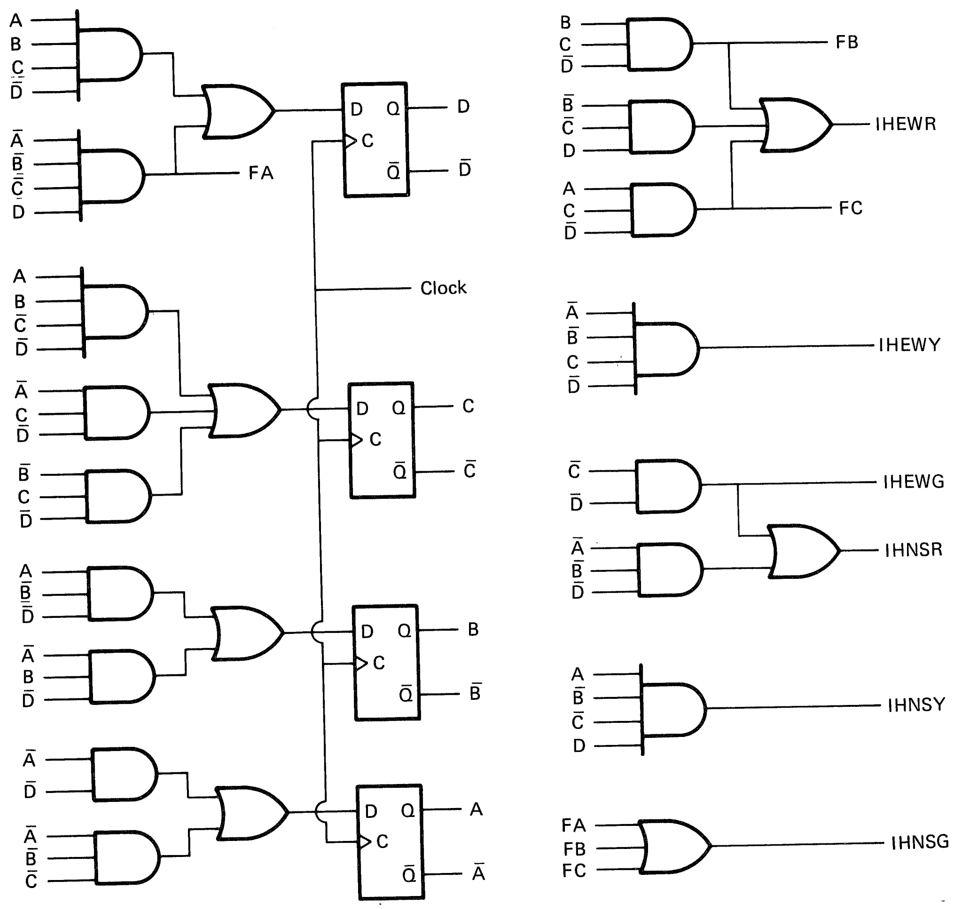


Figure 2-18 Complete traffic-light controller using gates.

In Fig. 2-20*a*, an inversion bubble is moved from a gate output to a gate input (or vice-versa), without changing the function *F* which is generated. As long as one inversion occurs between the AND gate output and the OR gate input connected to it, the function *F* will remain the same. However, the intermediate variable available on the wire connecting the two gates will change, as shown in the drawing.

Table 2-6 ROM vs gate implementation

	ROM	Gate
Design	Simple	Complex
Fabrication	Easy	Difficult
Maintenance	Easy	Difficult
Cost	May be more	May be less
Speed	May be slower	May be faster

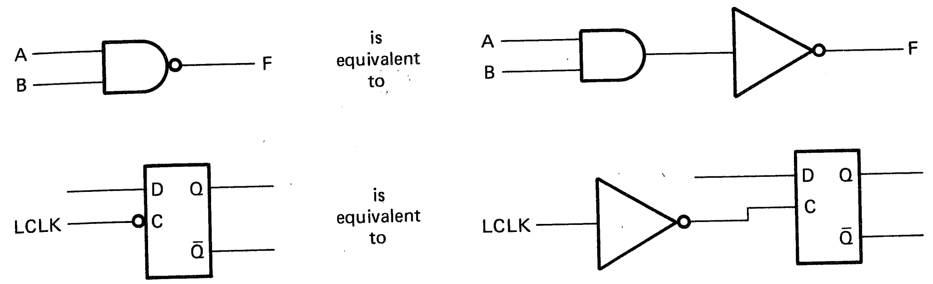


Figure 2-19 Inversion bubbles.

Since a logic variable inverted twice is simply that same original variable, we may add or delete *two* inversions at a time in any logic connection without changing circuit operation. Figure 2-20*b* shows two circuits for generating the function *G* that differ only by a double inversion. Although the function *G* remains unchanged, the intermediate variable *is* different. Finally, anything we've been doing with inversion bubbles may also be done with inverters, as shown in Fig. 2-20*c*.

Two common logic circuits are NAND and NOR gates, as shown in Fig. 2-21. These combinations of an AND and of an OR gate with inverters connected to their outputs are so common that they've been given special names: NAND and NOR gates, respectively. In fact, logic circuits fabricated with this integral inverter are more common than simple AND or OR gates.

Often, it is useful to use an AND gate instead of an OR gate, or vice-versa. For example, it may be desirable to use all of one type of gate to minimize parts stocking

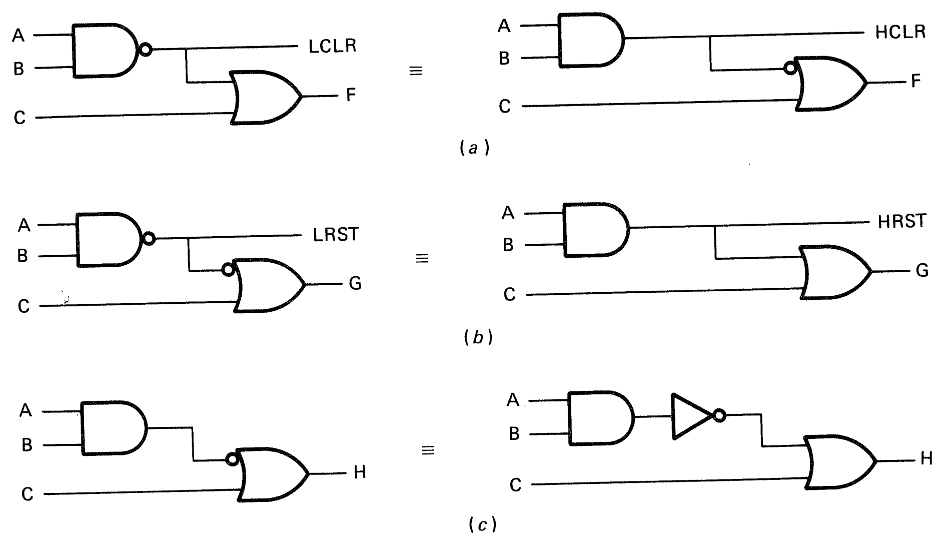


Figure 2-20 (*a*) Moving inversion bubbles. (*b*) Adding/deleting inversion bubbles. (*c*) Adding/deleting inverters.

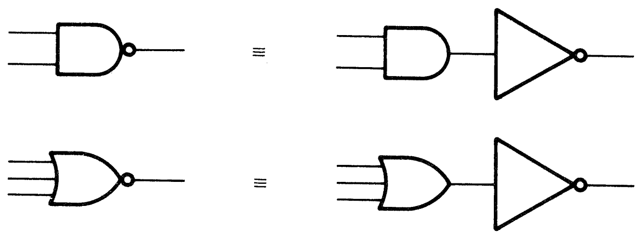


Figure 2-21 NAND and NOR gates.

requirements or simplify assembly or maintenance. Two boolean identities, called *De Morgan's laws*, allow us to substitute AND for OR gates, or vice-versa. These identities are:

$$\overline{(A+B+C+\ldots)} = \bar{A}\cdot\bar{B}\cdot\bar{C}\cdot\ldots$$

$$\overline{ABC\cdots} = \bar{A}+\bar{B}+\bar{C}+\cdots$$

Although these identities may be applied to the logical expressions directly, a simple graphical method exists to use these identities directly on the circuit diagram. To change an AND gate to an OR gate, first change the gate symbol to the OR-gate shape and then add inversion bubbles to the output and all inputs of that gate. To change an OR gate to an AND gate, first change the gate symbol to the AND-gate shape and then add inversion bubbles to the output and all inputs of that gate. After all gates have been changed to the type desired, you can manipulate inversion bubbles and inverters as described previously to obtain the desired circuit.

We desire to implement the standard SOP circuit of Fig. 2-22*a* with all NAND

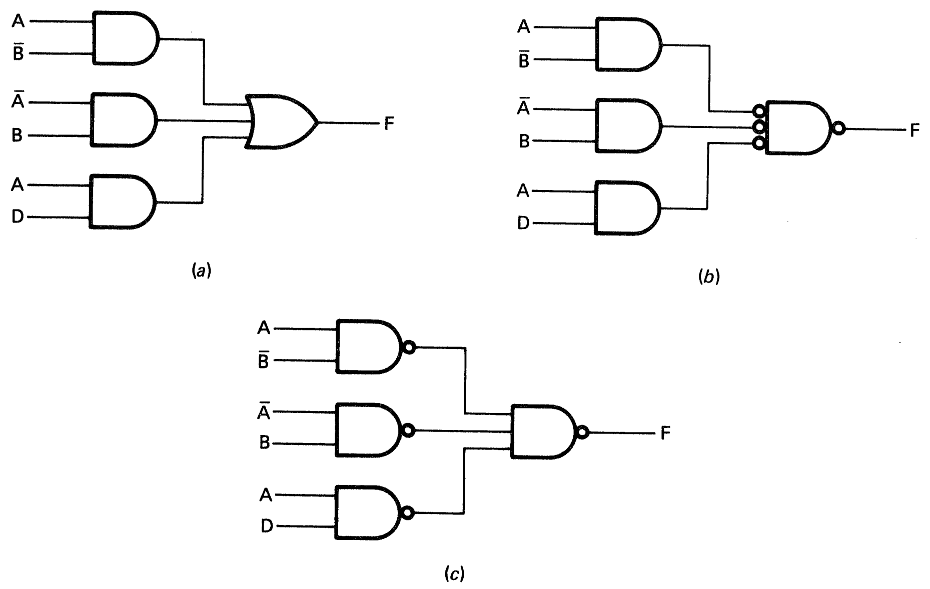


Figure 2-22 All NAND gate sum of products.

gates. Since we know that the OR output gate will have to change to an AND gate, we first apply the graphical version of De Morgan's law to get Fig. 2-22*b*. We change the gate symbol from OR to AND and add inversion bubbles to all inputs and outputs. This *does not* change the function of this gate. Next, notice that to obtain all NAND gates, move three inversion bubbles, as shown in Fig. 2-22*c*. Starting with the same SOP circuit in Fig. 2-23*a*, we obtain an all-NOR-gate realization in Fig. 2-23*c*. Draw this circuit more simply, as in Fig. 2-23*d*, by changing the input variables. Figure 2-23*d* also makes more physical sense since most input variables will be available in both true and complemented forms (e.g., from the Q and $\bar{Q}$ outputs of a flip-flop). Input variables that are not available in the sense desired are best inverted aside in the circuit diagram, as has been described previously. You can see now why NAND and NOR gates are so common. It is easy to change a SOP realization (or POS) to an all-NAND- or all-NOR-gate implementation.

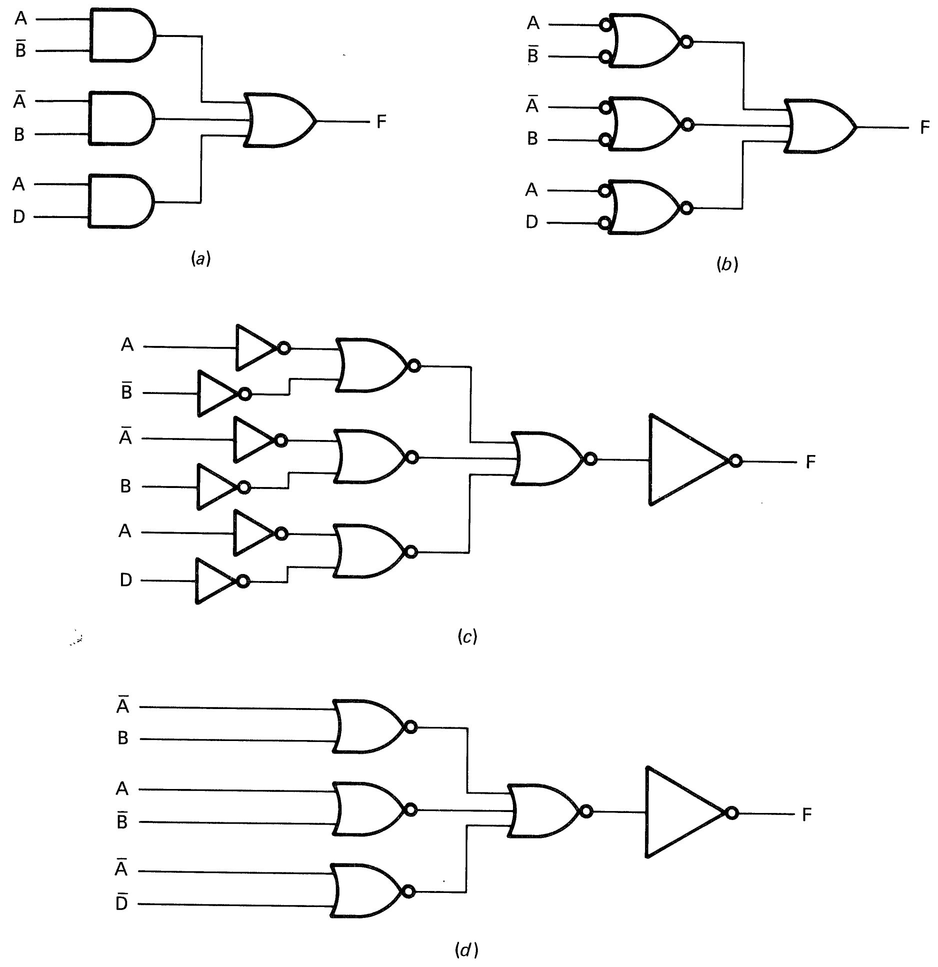


Figure 2-23 All NOR gate sum of products.

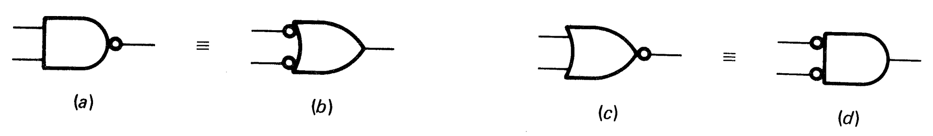


Figure 2-24 Alternate symbols for NAND and NOR gates.

You should also see that every gate circuit may be represented by two different symbols. Alternate symbols for NAND and NOR gates are shown in Fig. 2-24. These symbols are identical in meaning, and either may be used to indicate the same physical circuit. The choice of symbol is usually made to best describe the operation of the circuit. For example, in Fig. 2-24*a*, this NAND symbol tells us that the output will be 0 only when all inputs are 1s. However, the NAND symbol of Fig. 2-24*b* is more descriptive of a circuit whose output should be 1 when either or both inputs are 0. *Both of these circuits perform the same logic function.* The first symbol indicates that the gate generates an active low signal only when two active high signals are both asserted. The second symbol indicates that the gate generates an active high signal whenever either of two active low inputs is asserted.

Figure 2-24*c* and *d* are analogous for the NOR gate. The gate of Fig. 2-24*c* generates an active low signal if either of two active high signals is asserted. The same gate in Fig. 2-24*d* generates an active high signal when both its active low inputs are asserted.

Although either symbol for a gate would be correct on a circuit diagram, a circuit diagram is easier to understand if the symbol corresponding to the action desired is used. For example, consider the circuit of Fig. 2-25*a*. The function of this circuit is not obvious. Figure 2-25*b* describes the function more clearly. The designer had wanted an active high output when all 16 active high inputs were asserted. Since only

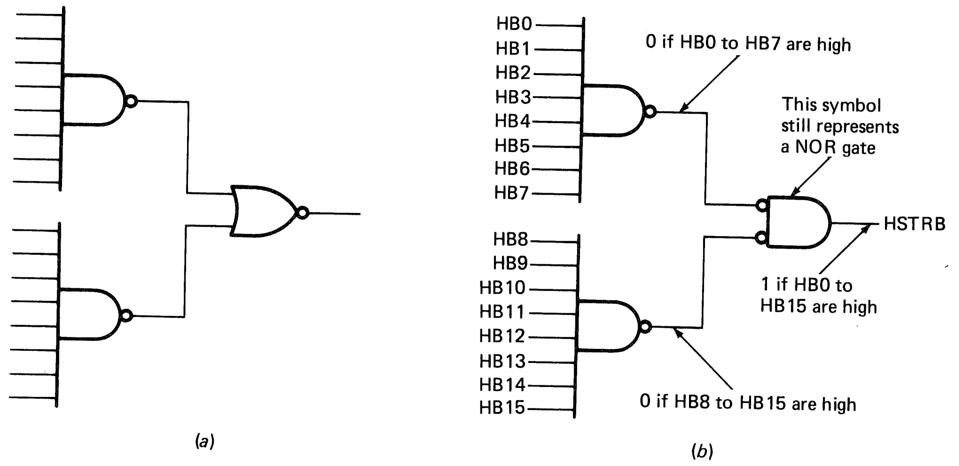

HB0
HB1
HB2
HB3
HB4
HB5
HB6
HB7
HB8
HB9
HB10
HB11
HB12
HB13
HB14
HB15
0 if HB0 to HB7 are high
This symbol still represents a NOR gate
HSTRB
1 if HB0 to HB15 are high
0 if HB8 to HB15 are high
(a)
(b)

Figure 2-25 (*a*) 16-bit coincidence detector. (*b*) More meaningful symbology for coincidence detector.

8-input NAND gates were available, two of these were combined with a NOR gate to perform the same function as a 16-input AND gate. The symbols in Fig. 2-25*b* clearly show that the designer was implementing a 16-input AND function.

Wired OR and Wired AND

Usually, the operation of a gate circuit is undefined if the *outputs* of two or more gates are connected directly together. This is the case for many logic gates: interconnecting their outputs does not generate a meaningful digital signal. However, some logic families (e.g., emitter-coupled logic or ECL) allow two or more outputs to be interconnected to generate a meaningful signal. Two such AND gates are interconnected in Fig. 2-26. The logic function generated by interconnecting these two outputs is the OR function. Thus, this connection is often called the *wired OR.* Other logic families may implement a *wired AND* function when interconnected as shown in Fig. 2-27. Notice that you cannot determine if a wired function is an AND or an OR from the circuit diagram alone. The data sheet for the gates being used must be consulted to determine the function implemented. Most logic families have at least a few types of gates that may be wire ORed or ANDed, even if the majority of gates in the family may not be so connected. For example, the transistor-transistor logic (TTL) family does not allow for interconnection of its outputs. However, a few TTL gates exist which can be interconnected to implement the wired AND function.

Although wired functions may seem very desirable because they save one entire gate, disadvantages sometimes limit their usefulness. Connecting gate outputs together usually *reduces* the fan out below that of *either* gate alone. If two gates, each with a fan out of 10, are wire ANDed, the combined circuit may be able to drive only nine inputs. Also, connecting gate outputs together will slow the propagation of logic signals through those gates. As will be seen in a later chapter, increasing the delay through gating circuits will slow the maximum speed at which an ASM using those gating circuits can operate. Finally, testing a circuit using wired functions may be very difficult. If 10 gate outputs are wired together, it is impossible to determine which of

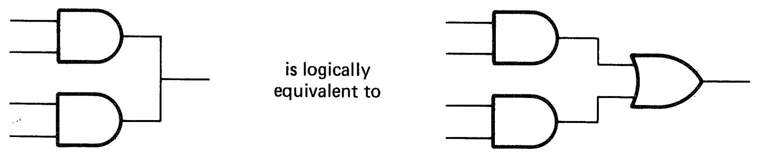


Figure 2-26 Wired OR function.

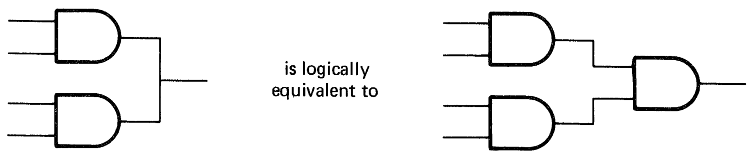
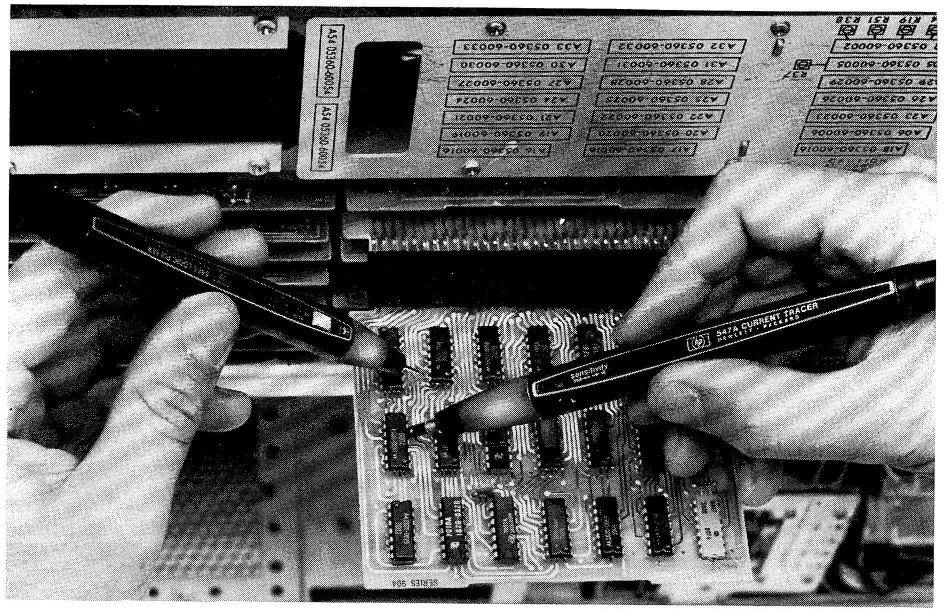


Figure 2-27 Wired AND function.



A current probe is on the right and a logic probe is on the left (*Hewlett-Packard Co.*).

the 10 is malfunctioning from measuring their single output logic voltage level! A *current probe* allows the current in the wire to each individual output to be sensed, even if the outputs are connected together. Even so, wired functions are often implemented so that gate outputs may be individually disconnected to facilitate testing.

Finally, flip-flop outputs are almost never connected together. Connecting flip-flop outputs together is ambiguous. It is not clear if the unchanged flip-flop output state is an input to the wired function or if the wired function actually changes the flip-flop's state.

PROBLEMS

2-1 Evaluate the following logical expressions for all combinations of variables.
- (*a*) $F_1 = \overline{A + B + C}$
- (*b*) $F_2 = (\overline{A})(\overline{B})(\overline{C})$
- (*c*) $F_3 = \overline{A} + \overline{B} + \overline{C}$
- (*d*) $F_4 = \overline{ABC}$
- (*e*) $F_5 = ABC + \overline{(B + C)}$

2-2 Consider the function specified by Table P2-1.
- (*a*) Find the canonical SOP expression for this function.
- (*b*) Find the canonical POS expression for this function.
- (*c*) Find the reduced SOP expression, using a K map.
- (*d*) Find the reduced POS expression, using a K map.

2-3 Reduce the K maps of Fig. P2-1 to the simplest SOP expressions.

2-4 Reduce the K maps of Fig. P2-1 to the simplest POS expressions.

Table P2-1 Function table

D	C	B	A	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

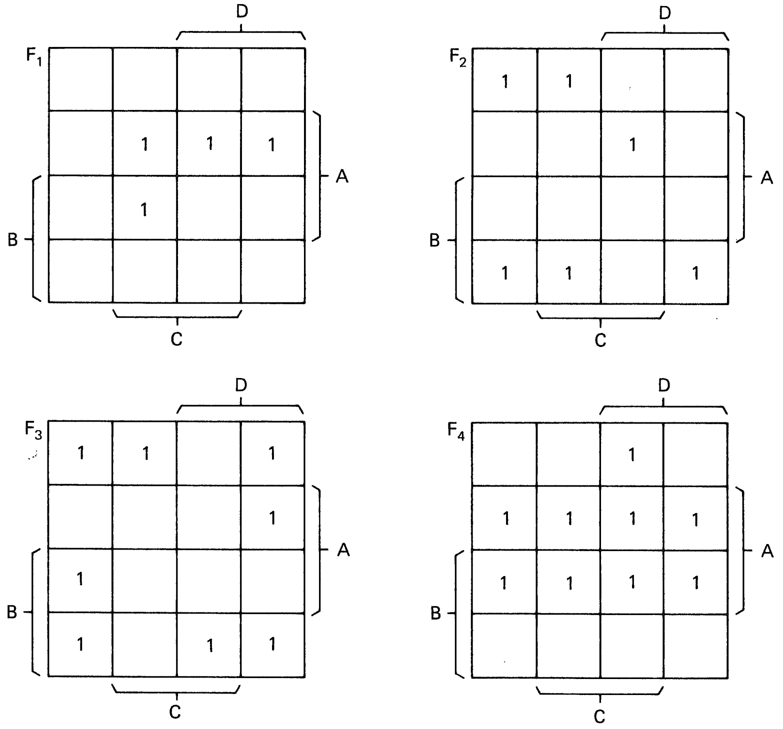


Figure P2-1 Karnaugh map for Prob. 2-6.

2-5 Draw circuit diagrams that implement the following logical expressions with AND gates, OR gates, and inverters.

(*a*) $AB + CD$
(*b*) $\overline{A}B + A\overline{B}$
(*c*) $(\overline{A}\overline{B} + AB)\,D + CE$
(*d*) $(\overline{A}\overline{B} + C)\,(D + E)$

2-6 Repeat Prob. 2-5 using:

(*a*) Only NAND gates and inverters
(*b*) Only NOR gates and inverters

2-7 Design two circuits to implement the flowchart of Fig. P2-2.

(*a*) Draw the circuit for a ROM controller, and specify the ROM contents.
(*b*) Draw the circuit for a gate controller.

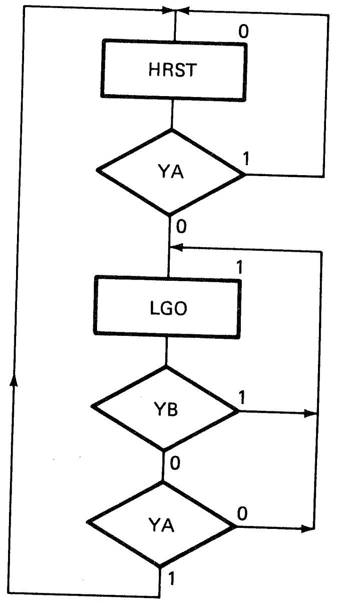


Figure P2-2 ASM chart for Prob. 2-7.

2-8 Design a gate controller for Prob. 1-9.

2-9 The K maps in Fig. 2-16 could have don't cares in positions 10, 11, 12, 13, 14, and 15 because these states were used in the ASM. Determine the minimum SOP representation for these 10 functions if the don't cares are included.

2-10 (*a*) Assume you are using a logic family that has wired ANDs. Repeat Prob. 2-6*a*, using the wired AND function.

(*b*) Assume you are using a logic family that has wired ORs. Repeat Prob. 2-6*b*, using the wired OR function.

CHAPTER

THREE

MSI AND LSI CIRCUITS

Examining many digital circuits, we should notice that similar circuitry and similar algorithms are used repeatedly. These functions are so common that they are manufactured as standard integrated circuits that may be incorporated into any digital system. The economies of scale reduce the cost of these mass-produced parts, regardless of the type of system in which they are used. Thus, the many computers manufactured help reduce the cost of traffic light controllers, and vice-versa. More importantly, these commonly used functions are preengineered. The cost of designing these circuits may be amortized over millions of individual parts. This significantly reduces digital system development costs.

We have already examined three types of mass-produced parts: gates, flip-flops, and ROMs. Gates and flip-flops are called small-scale integrated (SSI) circuits because they contain relatively few electronic components. In this chapter, we'll be primarily concerned with medium- and large-scale integrated (MSI and LSI) circuits. These larger circuits are advantageous because they allow complex functions to be preengineered and mass-produced. Before we examine common MSI circuits and their uses, we shall introduce digital signal busses.

BUS NOTATION

Often digital signals are grouped together for a common purpose. On a circuit diagram they appear as parallel signal lines, running together to various circuits. These signals are collectively called a bus. Because busses are difficult to draw and require a great amount of space on the diagram and their drawings can be confusing and difficult to follow, a shorthand notation has been developed. In Fig. 3-1, both the state variables (*SV0* to *SV3*) and the next-state function (*NS0* to *NS3*) are drawn as busses. Each