3.1 BOOLEAN LAWS AND THEOREMS

You should know enough Boolean algebra to make obvious simplifications. What follows is a discussion of the basic laws and theorems of Boolean algebra. Some of them will look familiar from ordinary algebra but others will be distinctly new.

Basic Laws

The commutative laws are

$$A + B = B + A \tag{3.1}$$
$$AB = BA \tag{3.2}$$

These two equations indicate that the order of a logical operation is unimportant because the same answer is arrived at either way. As far as logic circuits are concerned. Figure 3.la shows how to visualize Eq. (3.1). All it amounts to is realizing that the inputs to an OR gate can be transposed without changing the output. Likewise, Fig. 3.1b is a graphical equivalent for Eq. (3.2).

The associative laws are

$$A + (B + C) = (A + B) + C \tag{3.3}$$
$$A(BC) = (AB)C \tag{3.4}$$

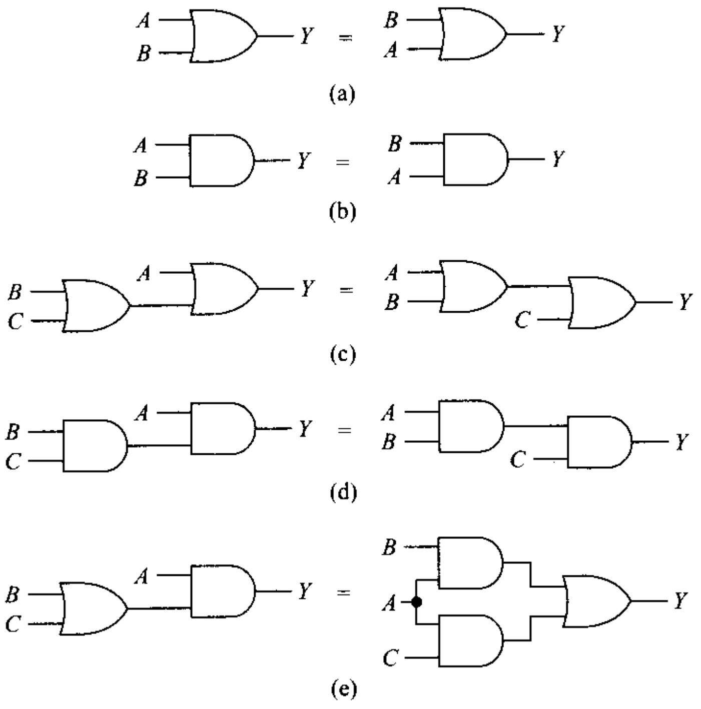


Fig. 3.1 Commutative, associative, and distributive laws

These laws show that the order of combining variables has no effect on the final answer. In terms of logic circuits, Fig. 3.lc illustrates Eq. (3.3), while Fig. 3.1d represents Eq. (3.4).

The distributive law is

$$A(B + C) = AB + AC \tag{3.5}$$

This law is easy to remember because it is identical to ordinary algebra. Figure 3.1e shows the corresponding logic equivalence. The distributive law gives you a hint about the value of Boolean algebra. If you can rearrange a Boolean expression, the corresponding logic circuit may be simpler.

The first five laws present no difficulties because they are identical to ordinary algebra. You can use these laws to simplify complicated Boolean expressions and arrive at simpler logic circuits. But before you begin, you have to learn other Boolean laws and theorems.

OR Operations

The next four Boolean relations are about OR operations. Here is the first:

$$A + 0 = A \tag{3.6}$$

This says that a variable ORed with 0 equals the variable. If you think about it, makes perfect sense. When A is 0,

$$0 + 0 = 0$$

And when A is 1,

$$1 + 0 = 1$$

In either case, Eq. (3.6) is true.

Another Boolean relation is

$$A + A = A \tag{3.7}$$

Again, you can see right through this by substituting the two possible values of A. First when $A = 0$, Eq. (3.7) gives

$$0 + 0 = 0$$

which is true. Next, $A = 1$ results in

$$1 + 1 = 1$$

which is also true because 1 ORed with 1 produces 1. Therefore, any variable ORed with itself equals the variable.

Another Boolean rule worth knowing is

$$A + 1 = 1 \tag{3.8}$$

Why is this valid? When $A = 0$, Eq. (3.8) gives

$$0 + 1 = 1$$

which is true. Also. $A = 1$ gives

$$1 + 1 = 1$$

This is correct because the plus sign implies OR addition, not ordinary addition. In summary, Eq. (3.8) says this, if one input to an OR gate is high, the output is high no matter what the other input.

Finally, we have

$$A + \overline{A} = 1 \tag{3.9}$$

You should see this in a flash. If A is 0, $\overline{A}$ is 1 and the equation is true. Conversely, if A is 1, $\overline{A}$ is 0 and the equation still agrees. In short, a variable ORed with its complement always equals 1.

AND Operations

Here are three AND relations

$$A \cdot 1 = A \tag{3.10}$$
$$A \cdot A = A \tag{3.11}$$
$$A \cdot 0 = 0 \tag{3.12}$$

When A is 0, all the foregoing are true. Likewise, when A is 1, each is true. Therefore, the three equations are valid and can be used to simplify Boolean equations.

One more AND formula is

$$A \cdot \overline{A} = 0 \tag{3.13}$$

This one is easy to understand because you get either

$$0 \cdot 1 = 0$$

or

$$1 \cdot 0 = 0$$

for the two possible values of A. In words, Eq. (3.13) indicates that a variable ANDed with its complement always equals zero.

Double Inversion and De Morgan's Theorems

The *double-inversion rule* is

$$\overline{\overline{A}} = A \tag{3.14}$$

which shows that the double complement of a variable equals the variable. Finally, there are the De Morgan theorems discussed in Chapter 2:

$$\overline{A + B} = \overline{A}\,\overline{B} \tag{3.15}$$
$$\overline{AB} = \overline{A} + \overline{B} \tag{3.16}$$

You already know how important these are. The first says a NOR gate and a bubbled AND gate are equivalent. The second says a NAND gate and a bubbled OR gate are equivalent.

Duality Theorem

The *duality theorem* is one of those elegant theorems proved in advanced mathematics. We will state the theorem without proof. Here is what the duality theorem says. Starting with a Boolean relation, you can derive another Boolean relation by

1. Changing each OR sign to an AND sign
2. Changing each AND sign to an OR sign
3. Complementing any 0 or 1 appearing in the expression

For instance, Eq. (3.6) says that

$$A + 0 = A$$

The dual relation is

$$A \cdot 1 = A$$

This dual property is obtained by changing the OR sign to an AND sign, and by complementing the 0 to get a 1.

The duality theorem is useful because it sometimes produces a new Boolean relation. For example, Eq. (3.5) states that

$$A(B + C) = AB + AC$$

By changing each OR and AND operation, we get the dual relation

$$A + BC = (A + B)(A + C) \qquad (3.17)$$

This is new, not previously discussed. (If you want to prove it, construct the truth table for each side of the equation. The truth tables will be identical, which means the Boolean relation is true.)

Covering and Combination

The covering rule, where one term covers the condition of the other term so that the other term becomes redundant, can be represented in dual form as

$$A + AB = A \qquad (3.18)$$

and

$$A(A + B) = A \qquad (3.19)$$

The above can be easily proved from basic laws because,

$$A + AB = A \cdot 1 + AB = A(1 + B) = A \cdot 1 = A$$

and

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

The combining rules are,

$$AB + A\overline{B} = A \qquad (3.20)$$

and in its dual form

$$(A + B)(A + \overline{B}) = A \qquad (3.21)$$

Eq. (3.20) can easily be proved as $B + \overline{B} = 1$

Expanding left hand side of Eq. (3.21)

$$A \cdot A + A \cdot B + A \cdot \overline{B} + B \cdot \overline{B} = A + A(B + \overline{B}) + 0$$
$$= A + A \cdot 1 = A + A = A = \text{right hand side}$$

Consensus Theorem

The *consensus theorem* finds a redundant term which is a *consensus of two other terms*. The idea is that if the consensus term is true, then any of the other two terms is true and thus it becomes redundant. This can be expressed in dual form as

$$AB + \overline{A}C + BC = AB + \overline{A}C \qquad (3.22)$$

$$(A + B)(\overline{A} + C)(B + C) = (A + B)(\overline{A} + C) \qquad (3.23)$$

In the first expression, BC is the consensus term and thus redundant. This is because if $BC = 1$, then both $B = 1$ and $C = 1$ and any of the other two terms AB or $\overline{A}C$ must be one as either $A = 1$ or $\overline{A} = 1$. Similarly, in the second expression, $(B + C)$ is the consensus term and if this term is 0 then both $B = 0$ and $C = 0$. This makes one of the other two sum terms 0 as either $A = 0$ or $\overline{A} = 0$.

SUMMARY OF BOOLEAN RELATIONS

For future reference, here are some Boolean relations and their duals:

$A + B = B + A$	$AB = BA$
$A + (B + C) = (A + B) + C$	$A(BC) = (AB)C$
$A(B + C) = AB + AC$	$A + BC = (A + B)(A + C)$
$A + 0 = A$	$A \cdot 1 = A$
$A + 1 = 1$	$A \cdot 0 = 0$
$A + A = A$	$A \cdot A = A$
$A + \overline{A} = 1$	$A \cdot \overline{A} = 0$
$\overline{\overline{A}} = A$	$\overline{\overline{A}} = A$
$\overline{A + B} = \overline{A}\,\overline{B}$	$\overline{AB} = \overline{A} + \overline{B}$
$A + AB = A$	$A(A + B) = A$
$A + \overline{A}B = A + B$	$A(\overline{A} + B) = AB$
$AB + A\overline{B} = A$	$(A + B)(A + \overline{B}) = A$
$AB + \overline{A}C + BC = AB + \overline{A}C$	$(A + B)(\overline{A} + C)(B + C) = (A + B)(\overline{A} + C)$

Example 3.1 Prove that, $A(A' + C)(A'B + C)(A'BC + C') = 0$

Solution

LHS $= (AA' + AC)(A'B + C)(A'BC + C')$	: distributive law
$= AC(A'B + C)(A'BC + C')$	: since, $XX' = 0$
$= (AC \cdot A'B + AC \cdot C)(A'BC + C')$	: distributive law
$= AC(A'BC + C')$	: since, $XX' = 0$
$= AC \cdot A'BC + AC \cdot C'$	: distributive law
$= 0 =$ RHS	: since, $XX' = 0$

Example 3.2 Simplify, $Y = (A + B)(A'(B' + C'))' + A'(B + C)$

Solution

$Y = (A + B)((A + (B' + C')') + A'(B + C)$: De Morgan's theorem

$= (A + B)(A + BC) + A'(B + C)$: De Morgan's theorem

$= (AA + ABC + AB + BBC) + A'(B + C)$

$= (A + AB + ABC + BC) + A'(B + C)$

$= A(1 + B + BC) + BC + A'(B + C)$

$= A + BC + A'(B + C)$

$= (A + A'(B + C)) + BC$

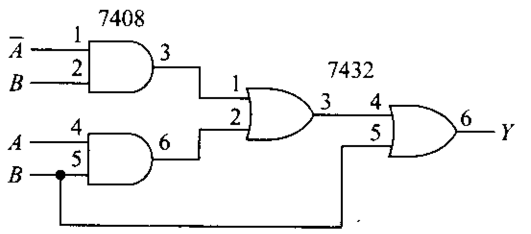
$= A + B + C + BC$

$= A + B + C(1 + B)$

$= A + B + C$

Example 3.3 A *logic clip* is a device that you can attach to a 14- or 16-pin DIP. This troubleshooting tool contains 16 light-emitting diodes (LEDs) that monitor the state of the pins. When a pin voltage is high, the corresponding LED lights up. If the pin voltage is low, the LED is dark.

Suppose you have built the circuit of Fig. 3.2a, but it doesn't work correctly. When you connect a logic clip to the 7408, you get the readings of Fig. 3.2b (a black circle means an LED is off, and a white one means it's on). When you connect the clip to the 7432, you get the indications of Fig. 3.2c. Which of the gates is faulty?



(a)

● 1	14 ○
○ 2	13 ○
● 3	12 ○
○ 4	11 ○
○ 5	10 ○
● 6	9 ○
● 7	8 ○

(b)

● = Off
○ = On

● 1	14 ○
● 2	13 ○
● 3	12 ○
● 4	11 ○
○ 5	10 ○
○ 6	9 ○
● 7	8 ○

(c)

Fig. 3.2

Solution When you use a logic clip, all you have to do is look at the inputs and output to isolate a faulty gate. For instance, Fig. 3.2b applies to a 7408 (quad 2-input AND gate). The First AND gate (pins 1 to 3) is all right because

Pin 1—low
Pin 2—high
Pin 3—low

A 2-input AND gate is supposed to have a low output if any input is low.

The second AND gate (pins 4 to 6) is defective. Why? Because

Pin 4—high
Pin 5—high
Pin 6—low

Something is wrong with this AND gate because it produces a low output even though both inputs are high.

If you check Fig. 3.2c (the 7432), all OR gates are normal. For instance, the first OR gate (pins 1 to 3) is all right because it produces a low output when the 2 inputs are low. The second OR gate (pins 4 to 6) is working correctly since it produces a high output when 1 input is high.

SELF-TEST

1. All the rules for Boolean algebra are exactly the same as for ordinary algebra. (T or F)
2. Expand using the distributive law: $Y = A(B + C)$.
3. Simplify: $Y = \bar{A}Q + AQ$.

3.2 SUM-OF-PRODUCTS METHOD

Figure 3.3 shows the four possible ways to AND two input signals that are in complemented and uncomplemented form. These outputs are called *fundamental products*. Table 3.1 lists each fundamental product next to the input conditions producing a high output. For instance, $\overline{A}\,\overline{B}$ is high when A and B are low; $\overline{A}B$ is high when A is low and B is high; and so on. The fundamental products are also called *minterms*. Products $A'B'$, $A'B$, AB', AB are represented by m_0, m_1, m_2, and m_3 respectively. The suffix i of m_i comes from decimal equivalent of binary values (Table 3.1) that makes corresponding product term high.

Table 3.1 Fundamental Products for Two Inputs

A	B	*Fundamental Product*
0	0	$\overline{A}\,\overline{B}$
0	1	$\overline{A}B$
1	0	$A\overline{B}$
1	1	AB

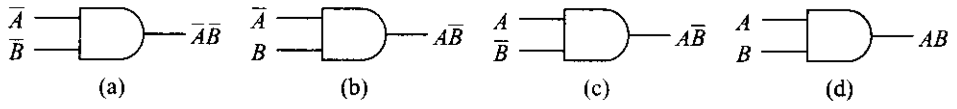


Fig. 3.3 ANDing two variables and their complements

The idea of fundamental products applies to three or more input variables. For example, assume three input variables: A, B, C and their complements. There are eight ways to AND three input variables and their complements resulting in fundamental products of

$$\overline{A}\,\overline{B}\,\overline{C},\ \overline{A}\,\overline{B}C,\ \overline{A}B\overline{C},\ \overline{A}BC,\ A\overline{B}\,\overline{C},\ A\overline{B}C,\ AB\overline{C},\ ABC$$

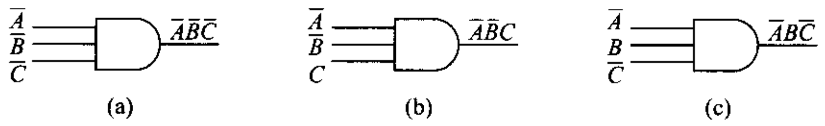


Fig. 3.4 Examples of ANDing three variables and their complements

The above three variable minterms can alternatively be represented by m_0, m_1, m_2, m_3, m_4, m_5, m_6, and m_7 respectively. Note that, for n variable problem there can be 2^n number of minterms. Figure 3.4a shows the first fundamental product, Fig. 3.4b the second, and Fig. 3.4c the third. (For practice, draw the gates for the remaining fundamental products.) for twice variable case.

Table 3.2 Fundamental Products for Three Inputs

A	B	C	*Fundamental Products*
0	0	0	$\overline{A}\,\overline{B}\,\overline{C}$
0	0	1	$\overline{A}\,\overline{B}C$
0	1	0	$\overline{A}B\overline{C}$
0	1	1	$\overline{A}BC$
1	0	0	$A\overline{B}\,\overline{C}$
1	0	1	$A\overline{B}C$
1	1	0	$AB\overline{C}$
1	1	1	ABC

Table 3.2 summarizes the fundamental products by listing each one next to the input condition that results in a high output. For instance, when $A = 1$, $B = 0$ and $C = 0$, the fundamental product results in an output of

$$Y = A\overline{B}\,\overline{C} = 1 \cdot \overline{0} \cdot \overline{0} = 1$$

Sum-of-Products Equation

Here is how to get the sum-of-products solution, given a truth table like Table 3.3. What you have to do is locate each output 1 in the truth table and write down the fundamental product. For instance, the first output 1 appears for an input of $A = 0$, $B = 1$, and $C = 1$. The corresponding fundamental product is $\bar{A}BC$. The next output 1 appears for $A = 1$, $B = 0$, and $C = 1$. The corresponding fundamental product is $A\bar{B}C$. Continuing like this, you can identify all the fundamental products, as shown in Table 3.4. To get the sum-of-products equation, all you have to do is OR the fundamental products of Table 3.4:

$$Y = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC \tag{3.24}$$

Alternate representation of Table 3.3,

$$Y = F(A, B, C) = \Sigma\, m\,(3, 5, 6, 7)$$

where 'Σ' symbolizes summation or logical OR operation that is performed on corresponding minterms and $Y = F(A, B, C)$ means Y is a function of three Boolean variables A, B and C. This kind of representation of a truth table is also known as *canonical sum form*.

Table 3.3 Design Truth Table

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

Table 3.4 Fundamental Products for Table 3.3

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	$1 \rightarrow \bar{A}BC$
1	0	0	0
1	0	1	$1 \rightarrow A\bar{B}C$
1	1	0	$1 \rightarrow AB\bar{C}$
1	1	1	$1 \rightarrow ABC$

Logic Circuit

After you have a sum-of-products equation, you can derive the corresponding logic circuit by drawing an AND-OR network, or what amounts to the same thing, a NAND-NAND network. In Eq. (3.24) each product is the output of a 3-input AND gate. Furthermore, the logical sum Y is the output of a 4-input OR gate. Therefore, we can draw the logic circuit as shown in Fig. 3.5. This AND-OR circuit is one solution to the design problem that we started with. In other words, the AND-OR circuit of Fig. 3.5 has the truth table given by Table 3.3.

We cannot build the circuit of Fig. 3.5 because a 4-input OR gate is not available as a TTL *chip* (a synonym for integrated circuit). But a 4-input NAND gate is. Figure 3.6 shows the logic circuit as a NAND-NAND circuit with TTL pin numbers. Also notice how the inputs come from a *bus*, a

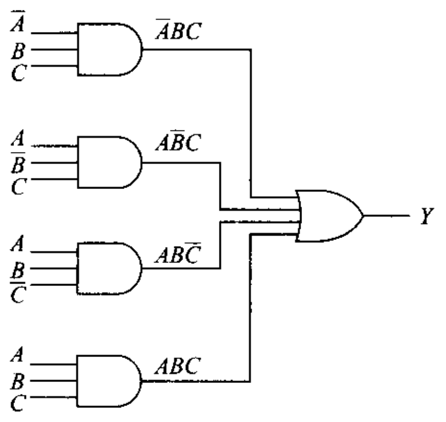


Fig. 3.5 AND-OR solution

group of wires carrying logic signals. In Fig. 3.6, the bus has six wires with logic signals A, B, C, and their complements. Microcomputers are bus-organized, meaning that the input and output signals of the logic circuits are connected to buses.

Example 3.4 Suppose a three-valuable truth table has a high output for these input conditions: 000, 010, 100, and 110. What is the sum-of-products circuit?

Solution Here are the fundamental products:

000 : $\bar{A}\bar{B}\bar{C}$
010 : $\bar{A}B\bar{C}$
100 : $A\bar{B}\bar{C}$
110 : $AB\bar{C}$

When you OR these products, you get

$$Y = \bar{A}\bar{B}\bar{C} + \bar{A}B\bar{C} + A\bar{B}\bar{C} + AB\bar{C}$$

The circuit of Fig. 3.6 will work if we reconnect the input lines to the bus as follows:

$\bar{A}$: pins 1 and 3
$\bar{B}$: pins 2 and 10
$\bar{C}$: pins 13, 5, 11, and 13
A : pins 9 and I
B : pins 4 and 2

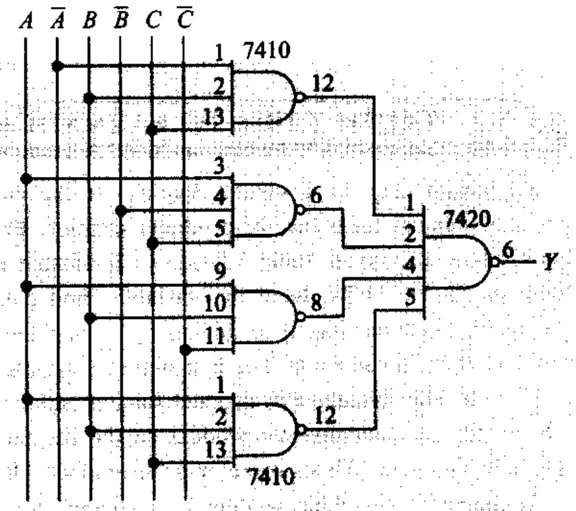


Fig. 3.6 Combinational logic circuit

Example 3.5 Simplify the Boolean equation in Example 3.4 and describe the logic circuit.

Solution The Boolean equation is

$$Y = \bar{A}\bar{B}\bar{C} + \bar{A}B\bar{C} + A\bar{B}\bar{C} + AB\bar{C}$$

Since $\bar{C}$ is common to each term, factor as follows:

$$Y = (\bar{A}\bar{B} + \bar{A}B + A\bar{B} + AB)\bar{C}$$

Again, factor to get

$$Y = [\bar{A}(\bar{B} + B) + A(\bar{B} + B)]\bar{C}$$

Now, simplify the foregoing as follows:

$$Y = [\bar{A}(1) + A(1)]\bar{C} = (\bar{A} + A)\bar{C}$$

or

$$Y = \bar{C}$$

This final equation means that you don't even need a logic circuit. All you need is a wire connecting input $\bar{C}$ to output Y.

The lesson is clear. The AND-OR (NAND-NAND) circuit you get with the sum-of-products method is not necessarily as simple as possible. With algebra, you often can factor and reduce the sum-of-products equation to arrive at a simpler Boolean equation, which means a simpler logic circuit. A simpler logic circuit is preferred because it usually costs less to build and is more reliable.

SELF-TEST

4. How many fundamental products are there for two variables? How many for three variables?
5. The AND-OR or the NAND-NAND circuit obtained with the sum-of-products method is always the simplest possible circuit. (T or F)

3.3 TRUTH TABLE TO KARNAUGH MAP

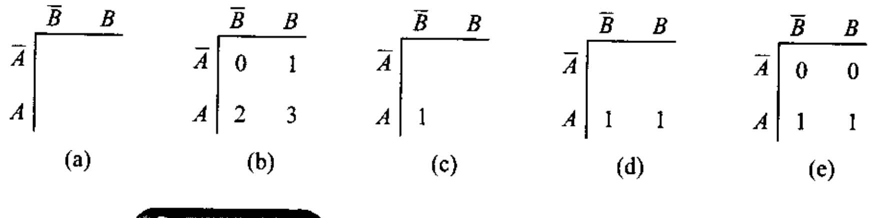
A *Karnaugh map* is a visual display of the fundamental products needed for a sum-of-products solution. For instance, here is how to convert Table 3.5 into its Karnaugh map. Begin by drawing Fig. 3.7a. Note the variables and complements: the vertical column has $\overline{A}$ followed by A, and the horizontal row has $\overline{B}$ followed by B. The first output 1 appears for $A = 1$ and $B = 0$. The fundamental product for this input condition is $A\overline{B}$. Enter this fundamental product on the Karnaugh map as shown in Fig. 3.7b. This 1 represents the product $A\overline{B}$ because the 1 is in row A and column $\overline{B}$.

Table 3.5

A	B	Y
0	0	0
0	1	0
1	0	1
1	1	1

Similarly, Table 3.5 has an output 1 appearing for inputs of $A = 1$ and $B = 1$. The fundamental product is AB, which can be entered on the Karnaugh map as shown in Fig. 3.7c. The final step in drawing the Karnaugh map is to enter 0s in the remaining spaces (see Fig. 3.7d).

In terms of decimal equivalence each position of Karnaugh map can be drawn as shown in Fig. 3.7b. Note that, Table 3.5 can be written using minterms as $Y = \Sigma\, m(2, 3)$ and Fig. 3.7e represents that.



$\overline{B}$ B $\overline{A}$ A
(a)
$\overline{B}$ B $\overline{A}$ 0 1 A 2 3
(b)
$\overline{B}$ B $\overline{A}$ A 1
(c)
$\overline{B}$ B $\overline{A}$ A 1 1
(d)
$\overline{B}$ B $\overline{A}$ 0 0 A 1 1
(e)

Fig. 3.7 Constructing a Karnaugh map

Three-Variable Maps

Here is how to draw a Karnaugh map for Table 3.6 or for logic equation, $Y = F(A, B, C) = \Sigma m(2,6,7)$. First, draw the blank map of Fig. 3.8a. The vertical column is labeled $\overline{A}\overline{B}$, $\overline{A}B$, AB, and $A\overline{B}$. With this order, only one variable changes from complemented to uncomplemented form (or vice versa) as you move downward. In terms of decimal equivalence of each position the Karnaugh map is as shown in Fig. 3.8b. Note how minterms in the equation gets mapped into corresponding positions in the map.

Table 3.6

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

Next, look for output 1s in Table 3.6. Output 1s appear for ABC inputs of 010, 110 and 111. The fundamental products for these input conditions are $\bar{A}B\bar{C}$, $AB\bar{C}$, and ABC Enter 1s for these products on the Karnaugh map (Fig. 3.8b).

The final step is to enter 0s in the remaining spaces (Fig. 3.8c).

	$\bar{C}$	C
$\bar{A}\bar{B}$		
$\bar{A}B$		
AB		
$A\bar{B}$		

(a)

	$\bar{C}$	C
$\bar{A}\bar{B}$	0	1
$\bar{A}B$	2	3
AB	6	7
$A\bar{B}$	4	5

(b)

	$\bar{C}$	C
$\bar{A}\bar{B}$		
$\bar{A}B$	1	
AB	1	1
$A\bar{B}$		

(c)

	$\bar{C}$	C
$\bar{A}\bar{B}$	0	0
$\bar{A}B$	1	0
AB	1	1
$A\bar{B}$	0	0

(d)

Fig. 3.8 Three-variable Karnaugh map

Four-Variable Maps

Many digital computers and systems process 4-bit numbers. For instance, some digital chips will work with nibbles like 0000, 0001, 0010, and so on. For this reason, logic circuits are often designed to handle four input variables (or their complements). This is why you must know how to draw a four-variable Karnaugh map.

Here is an example. Suppose you have a truth table like Table 3.7. Start by drawing a blank map like Fig. 3.9a. Notice the order. The vertical column is $\bar{A}\bar{B}, \bar{A}B, AB,$ and $A\bar{B}$. The horizontal row is $\bar{C}\bar{D}, \bar{C}D, CD,$ and $C\bar{D}$. In terms of decimal equivalence of each position the Karnaugh map is as shown in Fig. 3.9b. In Table 3.7, you have output 1s appearing for $ABCD$ inputs of 0001, 0110, 0111, and 1110. The fundamental products for these input conditions are $\bar{A}\bar{B}\bar{C}D, \bar{A}BC\bar{D}, \bar{A}BCD,$ and $ABC\bar{D}$. After entering 1s on the Karnaugh map, you have Fig. 3.9c. The final step of filling in 0s results in the complete map of Fig. 3.9d.

Table 3.7

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
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	1
1	1	1	1	0

SELF-TEST

6. What is a Karnaugh map?
7. How many entries are there on a four-variable Karnaugh map?

Entered Variable Map

As the name suggests, in *entered variable map* one of the input variable is placed inside Karnaugh map. This is done separately noting how it is related with output. This reduces the Karnaugh map size by one degree,

(a)

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$				
$\bar{A}B$				
AB				
$A\bar{B}$				

(b)

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	1	3	2
$\bar{A}B$	4	5	7	6
AB	12	13	15	14
$A\bar{B}$	8	9	11	10

(c)

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	1	3	2
$\bar{A}B$	4	5	7	6
AB	12	13	15	14
$A\bar{B}$	8	9	11	10

(d)

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	1	0	0
$\bar{A}B$	0	0	1	1
AB	0	0	0	1
$A\bar{B}$	0	0	0	0

Fig. 3.9 Constructing a four-variable Karnaugh map

i.e. a three variable problem that requires $2^3 = 8$ locations in Karnaugh map will require $2^{(3-1)} = 4$ locations in entered variable map. This technique is particularly useful for mapping problems with more than four input variables.

We illustrate the technique by taking a three variable example, truth table of which is shown in Table 3.6. Let's choose C as *map entered variable* and see how output Y varies with C for different combinations of other two variables A and B. Fig. 3.10a shows the relation drawn from Table 3.6. For $AB = 00$ we find $Y = 0$ and is not dependent on C. For $AB = 01$ we find Y is complement of C thus we can write $Y = C'$. Similarly, for $AB = 10$, $Y = 0$ and for $AB = 11$, $Y = 1$. The corresponding entered variable map is shown in Fig. 3.10b. If we choose A as map entered variable we have table shown in Fig. 3.10c showing relation with Y for various combinations of BC; the corresponding entered variable map is shown in Fig. 3.10d.

(a)

A	B	Y
0	0	0
0	1	$\bar{C}$
1	0	0
1	1	1

(b)

	$\bar{B}$	B
$\bar{A}$	0	$\bar{C}$
A	0	1

(c)

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

(d)

	$\bar{C}$	C
$\bar{B}$	0	0
B	1	A

Fig. 3.10 Entered variable map of truth table shown in Table 3.6

3.4 PAIRS, QUADS, AND OCTETS

Look at Fig. 3.11a. The map contains a pair of 1s that are horizontally adjacent (next to each other). The first 1 represents the product $ABCD$; the second 1 stands for the product $ABC\bar{D}$. As we move from the first 1 to the second 1, only one variable goes from uncomplemented to complemented form (D to $\bar{D}$); the other variables don't change form (A, B and C remain uncomplemented). Whenever this happens, you can *eliminate the variable that changes form.*

(a)

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	0	0	0
$\bar{A}B$	0	0	0	0
AB	0	0	1	1
$A\bar{B}$	0	0	0	0

(b)

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	0	0	0
$\bar{A}B$	0	0	0	0
AB	0	0	1	1
$A\bar{B}$	0	0	0	0

Fig. 3.11 Horizontally adjacent 1s

Proof

The sum-of-products equation corresponding to Fig. 3.11a is

$$Y = ABCD + ABC\overline{D}$$

which factors into

$$Y = ABC(D + \overline{D})$$

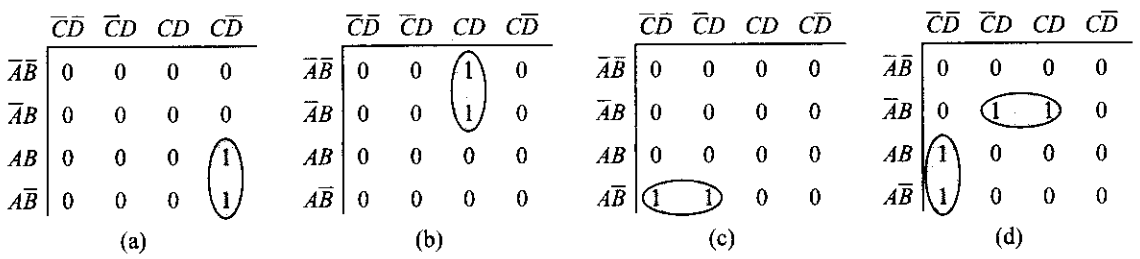
Since D is ORed with its complement, the equation simplifies to

$$Y = ABC$$

In general, a pair of horizontally adjacent 1s like those of Fig. 3.11a means the sum-of-products equation will have a variable and a complement that drop out as shown above.

For easy identification, we will encircle two adjacent 1s as shown in Fig. 3.11b. Two adjacent 1s such as these are called a pair. In this way, we can tell at a glance that one variable and its complement will drop out of the corresponding Boolean equation. In other words, an encircled pair of 1s like those of Fig. 3.11b no longer stand for the ORing of two separate products, $ABCD$ and $ABC\overline{D}$. Rather, the encircled pair is visualized as representing a single reduced product ABC.

Here is another example. Figure 3.12a shows a pair of 1s that are vertically adjacent. These 1s correspond to $ABC\overline{D}$ and $A\overline{B}C\overline{D}$. Notice that only one variable changes from uncomplemented to complemented form (B to $\overline{B}$). Therefore, B and $\overline{B}$ can be factored and eliminated algebraically, leaving a reduced product of $AC\overline{D}$.



$\overline{C}\overline{D}$ $\overline{C}D$ CD $C\overline{D}$
$\overline{A}\overline{B}$
$\overline{A}B$
AB
$A\overline{B}$
(a)
(b)
(c)
(d)

Fig. 3.12 Examples of pairs

More Examples

Whenever you see a pair of horizontally or vertically adjacent 1s, you can eliminate the variable that appears in both complemented and uncomplemented form. The remaining variables (or their complements) will be the only ones appearing in the single-product term corresponding to the pair of 1s. For instance, a glance at Fig. 3.12b indicates that B goes from complemented to uncomplemented form when we move from the upper to the lower 1; the other variables remain the same. Therefore, the encircled pair of 1s in Fig. 3.12b, represents the product $\overline{A}CD$. Likewise, given the pair of 1s in Fig. 3.12c, the only change is from $\overline{D}$ to D. So the encircled pair of 1s stands for the product $A\overline{B}\overline{C}$.

If more than one pair exists on a Karnaugh map, you can OR the simplified products to get the Boolean equation. For instance, the lower pair of Fig. 3.12d represents the simplified product $A\overline{C}\overline{D}$; the upper pair stands for $\overline{A}BD$. The corresponding Boolean equation for this map is

$$Y = A\overline{C}\overline{D} + \overline{A}BD$$

The Quad

A *quad* is a group of four 1s that are horizontally or vertically adjacent. The 1s may be end-to-end, as shown in Fig. 3.13a, or in the form of a square, as in Fig. 3.13b. When you see a quad, always encircle it because it leads to a simpler product. In fact, a quad eliminates *two variables and their complements*.

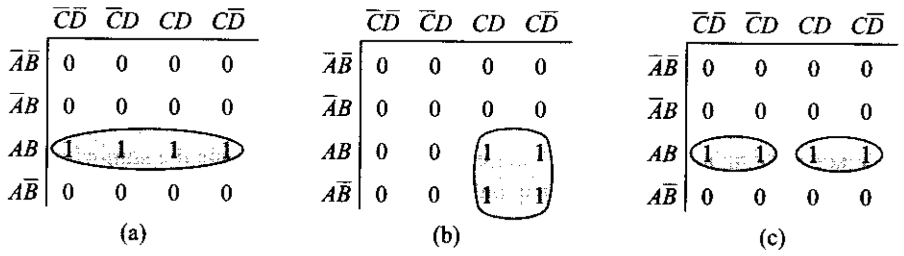


Fig. 3.13 Examples of quads

Here is why a quad eliminates two variables and their complements. Visualize the four 1s of Fig. 3.13a as two pairs (see Fig. 3.13c). The first pair represents $AB\bar{C}$; the second pair stands for ABC. The Boolean equation for these two pairs is

$$Y = AB\bar{C} + ABC$$

This factors into

$$Y = AB(\bar{C} + C)$$

which reduces to

$$Y = AB$$

So, the quad of Fig. 3.13a represents a product whose two variables and their complements have dropped out.

A similar proof applies to any quad. You can visualize it as two pairs whose Boolean equation leads to a single product involving only two variables or their complements. There's no need to go through the algebra each time. Merely step through the different 1s in the quad and determine which two variables go from complemented to uncomplemented form (or vice versa); these are the variables that drop out.

For instance, look at the quad of Fig. 3.13b. Pick any 1 as a starting point. When you move horizontally, D is the variable that changes form. When you move vertically, B changes form. Therefore, the remaining variables (A and C) are the only ones appearing in the simplified product. In other words, the simplified equation for the quad of Fig. 3.13b is

$$Y = AC$$

The Octet

Besides pairs and quads, there is one more group to adjacent 1s to look for: the *octet*. This is a group of eight 1s like those of Fig. 3.14a on the next page. An octet like this eliminates *three variables and their complements*. Here's why. Visualize the octet as two quads (see Fig. 3.14b). The equation for these two quads is

$$Y = A\bar{C} + AC$$

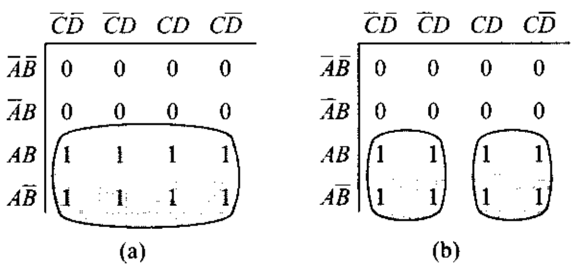


Fig. 3.14 **Example of octet**

After factoring,

$$Y = A(\bar{C} + C)$$

But this reduces to

$$Y = A$$

So the octet of Fig. 3.14a means three variables and their complements drop out of the corresponding product.

A similar proof applies to any octet. From now on don't bother with the algebra. Merely step through the 1s of the octet and determine which three variables change form. These are the variables that drop out.

SELF-TEST

8. **On a Karnaugh map, two adjacent 1s are called a ____.**
9. **On a Karnaugh map, an octet contains how many 1s?**

3.5 KARNAUGH SIMPLIFICATIONS

As you know, a pair eliminates one variable and its complement, a quad eliminates two variables and their complements, and an octet eliminates three variables and their complements. Because of this, after you draw a Karnaugh map, encircle the octets first, the quads second, and the pairs last. In this way, the greatest simplification results.

An Example

Suppose you have translated a truth table into the Karnaugh map shown in Fig. 3.15a. First, look for octets. There are none. Next, look for quads. When you find them, encircle them. Finally, look for and encircle pairs. If you do this correctly, you arrive at Fig. 3.15b.

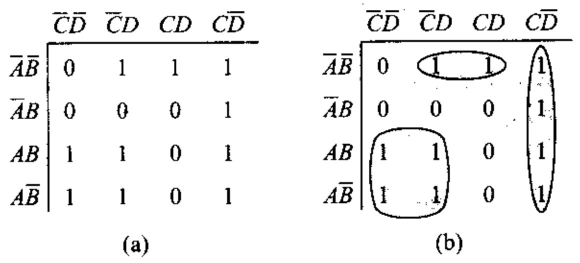


Fig. 3.15 **Encircling octets, quads and pairs**

The pair represents the simplified product $\bar{A}\bar{B}D$, the lower quad stands for $A\bar{C}$, and the quad on the right represents $C\bar{D}$. By ORing these simplified products, we get the Boolean equation corresponding to the entire

Karnaugh map:

$$Y = \bar{A}\bar{B}D + A\bar{C} + C\bar{D} \tag{3.25}$$

Overlapping Groups

You are allowed to use the same 1 more than once. Figure 3.16a illustrates this idea. The I representing the fundamental product $AB\bar{C}D$ is part of the pair and part of the octet. The simplified equation for the overlapping groups is

$$Y = A + B\bar{C}D \tag{3.26}$$

It is valid to encircle the 1s as shown in Fig. 3.16b, but then the isolated 1 results in a more complicated equation:

$$Y = A + \bar{A}B\bar{C}D$$

So, always overlap groups if possible. That is, use the 1s more than once to get the largest groups you can.

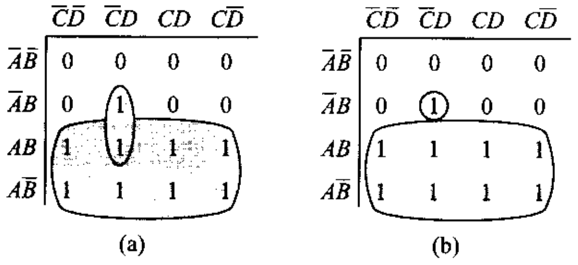


Fig. 3.16 Overlapping groups

Rolling the Map

Another thing to know about is rolling. Look at Fig. 3.17a on the next page. The pairs result in this equation:

$$Y = B\bar{C}\bar{D} + BC\bar{D} \tag{3.27}$$

Visualize picking up the Karnaugh map and rolling it so that the left side touches the right side. If you are visualizing correctly, you will realize the two pairs actually form a quad. To indicate this, draw half circles around each pair, as shown in Fig. 3.17b. From this viewpoint, the quad of Fig. 3.17b has the equation

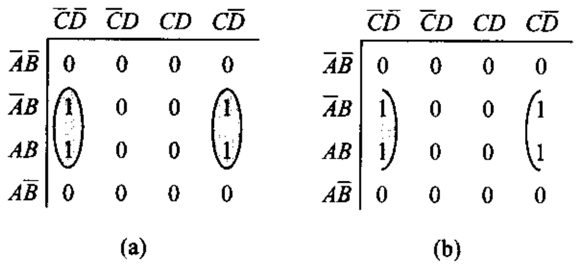


Fig. 3.17 Rolling the Karnaugh map

$$Y = B\bar{D} \tag{3.28}$$

Why is rolling valid? Because Eq. (3.27) can be algebraically simplified to Eq. (3.28). The proof starts with Eq. (3.27):

$$Y = B\bar{C}\bar{D} + BC\bar{D}$$

This factors into

$$Y = B\bar{D}(\bar{C} + C)$$

which reduces to

$$Y = B\bar{D}$$

But this final equation is the one that represents a rolled quad like Fig. 3.17b. Therefore, 1s on the edges of a Karnaugh map can be grouped with 1s on opposite edges.

More Examples

If possible, roll and overlap to get the largest groups you can find. For instance, Fig. 3.18a shows an inefficient way to encircle groups. The octet and pair have a Boolean equation of

$$Y = \overline{C} + BC\overline{D}$$

You can do better by rolling and overlapping as shown in Fig. 3.18b; the Boolean equation now is

$$Y = \overline{C} + B\overline{D}$$

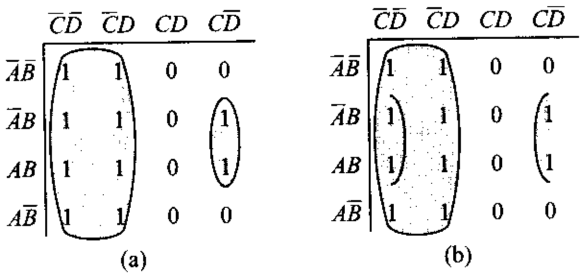


Fig. 3.18 **Rolling and overlapping**

Here is another example. Figure 3.19a shows an inefficient grouping of 1s; the corresponding equation is

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

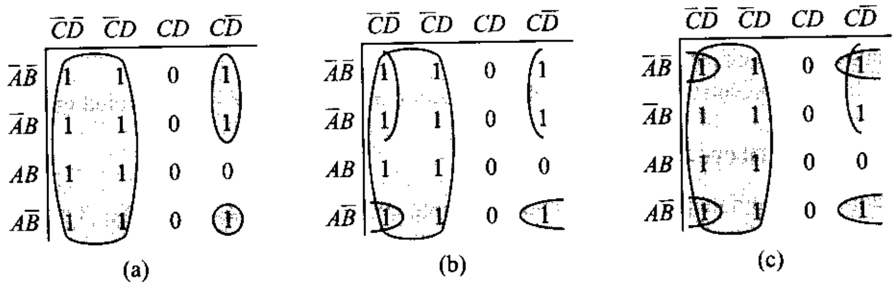


Fig. 3.19 **Different ways of encircling groups**

If we roll and overlap as shown in Fig. 3.19b, the equation is simpler:

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

It is possible to group the 1s as shown in Fig. 3.19c. The equation now becomes

$$Y = \overline{C} + \overline{A}\overline{D} + \overline{B}\overline{D} \tag{3.29}$$

Compare this with the preceding equation. As you can see, the equations are comparable in simplicity. Either grouping (Fig. 3.19b or c) is valid; therefore, you can use whichever you like.

Eliminating Redundant Groups

After you have finished encircling groups, eliminate any *redundant group*. This is a group whose 1s are already used by other groups. Here is an example. Given Fig. 3.20a, encircle the quad to get Fig. 3.20b. Next, group the remaining 1s into pairs by overlapping (Fig. 3.20c). In Fig. 3.20c, all the 1s of the quad are used by the pairs. Because of this, the quad is redundant and can be eliminated to get Fig. 3.20d. As you see, all the 1s are covered by the pairs. Figure 3.20d contains one less product than Fig. 3.20c; therefore, Fig. 3.20d is the most efficient way to group the 1s.

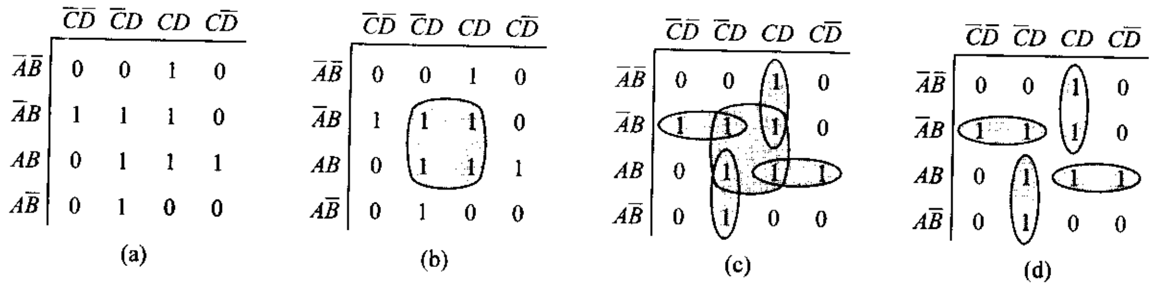


Fig. 3.20 Eliminating an unnecessary group

Conclusion

Here is a summary of the Karnaugh-map method for simplifying Boolean equations:

1. Enter a 1 on the Karnaugh map for each fundamental product that produces a 1 output in the truth table. Enter 0s elsewhere.
2. Encircle the octets, quads, and pairs. Remember to roll and overlap to get the largest groups possible.
3. If any isolated 1s remain, encircle each.
4. Eliminate any redundant group.
5. Write the Boolean equation by ORing the products corresponding to the encircled groups.

Simplification of Entered Variable Map

This is similar to Karnaugh map method. Refer to entered variable maps shown in Fig. 3.10. The groupings for these are as shown in Fig. 3.21a and Fig. 3.21b. Note that in Fig. 3.21a C' is grouped with 1 to get a larger group as 1 can be written as $1 = 1 + C'$. Similarly A is grouped with 1 in Fig. 3.21b.

Next, the product term representing each group is obtained by including map entered variable in the group as an additional ANDed term. Thus, group 1 of Fig. 3.21a gives $B.(C') = BC'$ and group 2 gives $AB.(1) = AB$ resulting $Y = BC' + AB$.

In Fig. 3.21b, group 1 gives product term $B.(A) = AB$ and group 2 gives $BC'.(1) = BC'$ so that $Y = BC' + AB$. The final expression is same for both as they represent the same truth table (Table 3.6).

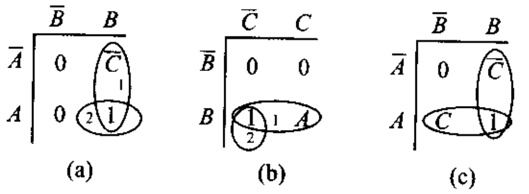


Fig. 3.21 Simplification of entered variable map

Note that, entered variable map shown Fig. 3.21c for a different truth table (Take it as an exercise to prepare that truth table) has only two product terms and doesn't need a separate coverage of 1. This is because one can write $1 = C + C'$ and C is included in one group while C' in other. The output of this map can be written as $Y = AC + BC'$.

Example 3.6 What is the simplified Boolean equation for the following logic equation expressed by minterms?

$$Y = F(A, B, C, D) = \Sigma m(7, 9, 10, 11, 12, 13, 14, 15)$$

Solution We know, each minterm makes corresponding location in Karnaugh map 1 and thus Fig. 3.22a represents the given equation. There are no octets, but there is a quad as shown in Fig. 3.22b. By overlapping, we can find two more quads (see Fig. 3.22c). We can encircle the remaining 1 by making it part of an overlapped pair (Fig. 3.22d). Finally, there are no redundant groups.

The horizontal quad of Fig. 3.22d corresponds to a simplified product AB. The square quad on the right corresponds to AC, while the one on the left stands for AD. The pair represents BCD. By ORing these products, we get the simplified Boolean equation:

$$Y = AB + AC + AD + BCD \tag{3.30}$$

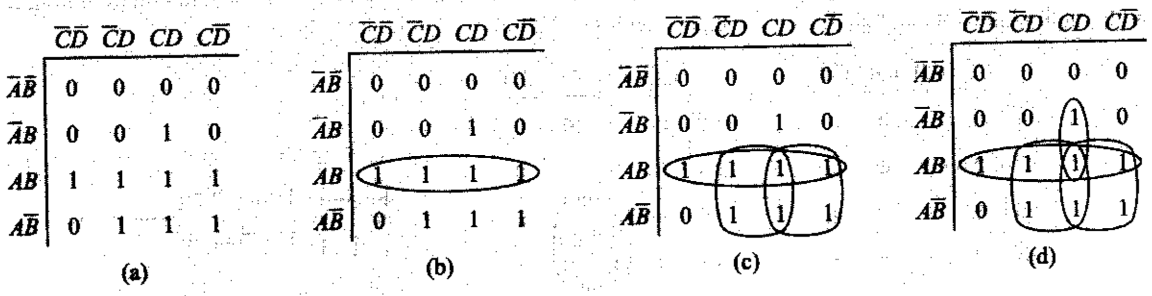


Fig. 3.22 **Using the Karnaugh map**

SELF-TEST

10. Write the sum-of-product terms for the entries in Fig. 3.18. Use Boolean algebra to simplify the expression.

3.6 DON'T-CARE CONDITIONS

In some digital systems, certain input conditions never occur during normal operation; therefore, the corresponding output never appears. Since the output never appears, it is indicated by an X in the truth table. For instance, Table 3.8 on the next page shows a truth table where the output is low for all input entries from 0000 to 1000, high for input entry 1001, and an X for 1010 through 1111. The X is called a *don't-care condition*. Whenever you see an X in a truth table, you can let it equal either 0 or 1, whichever produces a simpler logic circuit.

Figure 3.23a shows the Karnaugh map of Table 3.8 with don't-cares for all inputs from 1010 to 1111. These don't-cares are like wild cards in poker because you can let them stand for whatever you like. Figure 3.23b shows the most efficient way to encircle the 1. Notice two crucial ideas. First, the 1 is included in a

Table 3.8 **Truth Table with Don't-Care Conditions**

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

quad, the largest group you can find if you visualize all X's as 1s. Second, after the 1 has been encircled, all X's outside the quad are visualized as 0s. In this way, the X's are used to the best possible advantage. As already mentioned, you are free to do this because don't-cares correspond to input conditions that never appear.

The quad of Fig. 3.23b results in a Boolean equation of

$$Y = AD$$

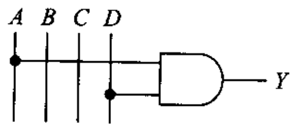
The logic circuit for this is an AND gate with inputs of A and D, as shown in Fig. 3.23c. You can check this logic circuit by examining Table 3.8. The possible inputs are from 0000 to 1001; in this range a high A and a high D produce a high Y only for input condition 1001.

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	0	0	0
$\bar{A}B$	0	0	0	0
AB	×	×	×	×
$A\bar{B}$	0	1	×	×

(a)

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	0	0	0
$\bar{A}B$	0	0	0	0
AB	×	×	×	×
$A\bar{B}$	0	1	×	×

(b)



(c)

Fig. 3.23 Don't-care conditions

Remember these ideas about don't-care conditions:

1. Given the truth table, draw a Karnaugh map with 0s, 1s, and don't-cares.
2. Encircle the actual 1s on the Karnaugh map in the largest groups you can find by treating the don't-cares as 1s.
3. After the actual 1s have been included in groups, disregard the remaining don't cares by visualizing them as 0s.

Example 3.7 Suppose Table 3.8 has high output for an input of 0000, low output, for 0001 to 1001, and don't cares for 1010 to 1111. What is the simplest logic circuit with this truth table?

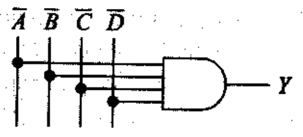
Solution The truth table has a 1 output only for the input condition 0000. The corresponding fundamental product is $\bar{A}\bar{B}\bar{C}\bar{D}$. Figure 3.24a shows the Karnaugh map with a 1 for the fundamental product, 0s for inputs 0001 to 1001, and X's for inputs 1010 to 1111. In this case, the don't-cares are of no help. The best we can do is to encircle the isolated 1, while treating the don't-cares as 0s. So, the Boolean equation is

$$Y = \bar{A}\bar{B}\bar{C}\bar{D}$$

Figure 3.24b shows the logic circuit. The 4-input AND gate produces a high output only for the input condition $A = 0$, $B = 0$, $C = 0$, and $D = 0$.

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	1	0	0	0
$\bar{A}B$	0	0	0	0
AB	×	×	×	×
$A\bar{B}$	0	0	×	×

(a)



(b)

Fig. 3.24 Decoding 0000

Example 3.8 Give the simplest logic circuit for following logic equation where d represents don't-care condition for following locations.

$$F(A, B, C, D) = \Sigma m(7) + d(10, 11, 12, 13, 14, 15)$$

Solution Figure 3.25a is the Karnaugh map. The most efficient encircling is to group the 1s into a pair using the don't-care as shown. Since this is the largest group possible, all remaining don't cares are treated as 0s. The equation for the pair is

$$Y = BCD$$

and Fig. 3.25b is the logic circuit. This 3.input AND gate produces a high output only for an input of $A = 0$, $B = 1$, $C = 1$, and $D = 1$ because the input possibilities range only from 0000 to 1001.

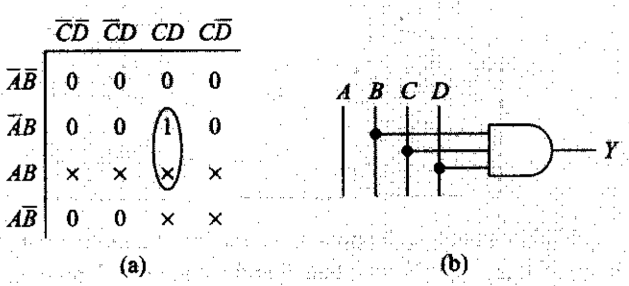


Fig. 3.25 Decoding 0111

SELF-TEST

11. What is meant by a don't-care condition on a Karnaugh map? How is it indicated?
12. How can using don't-cares aid circuit simplification?

3.7 PRODUCT-OF-SUMS METHOD

With the sum-of-products method the design starts with a truth table that summarizes the desired input-output conditions. The next step is to convert the truth table into an equivalent sum-of-products equation. The final step is to draw the AND-OR network or its NAND-NAND equivalent.

The product-of-sums method is similar. Given a truth table, you identify the fundamental sums needed for a logic design. Then by ANDing these sums, you get the product-of-sums equation corresponding to the truth table. But there are some differences between the two approaches. With the sum-of-products method, the fundamental product produces an output 1 for the corresponding input condition. But with the product-of-sums method, the fundamental sum produces an output 0 for the corresponding input condition. The best way to understand this distinction is with an example.

Converting a Truth Table to an Equation

Suppose you are given a truth table like Table 3.9 and you want to get the product-of-sums equation. What you have to do is locate each output 0 in the truth table and write down its fundamental sum. In Table 3.9, the first output 0 appears for $A = 0$, $B = 0$, and $C = 0$. The fundamental sum for these inputs is $A + B + C$. Why? Because this produces an output zero for the corresponding input condition:

$$Y = A + B + C = 0 + 0 + 0 = 0$$

Table 3.9

A	B	C	Y	*Maxterm*
0	0	0	$0 \to A+B+C$	M_0
0	0	1	1	M_1
0	1	0	1	M_2
0	1	1	$0 \to A+\overline{B}+\overline{C}$	M_3
1	0	0	1	M_4
1	0	1	1	M_5
1	1	0	$0 \to \overline{A}+\overline{B}+C$	M_6
1	1	1	1	M_7

The second output 0 appears for the input condition of $A = 0$, $B = 1$, and $C = 1$. The fundamental sum for this is $A + \overline{B} + \overline{C}$. Notice that B and C are complemented because this is the only way to get a logical sum of 0 for the given input conditions:

$$Y = A + \overline{B} + \overline{C} = 0 + \overline{1} + \overline{1} = 0 + 0 + 0 = 0$$

Similarly, the third output 0 occurs for $A = 1$, $B = 1$, and $C = 0$; therefore, its fundamental sum is $\overline{A} + \overline{B} + C$:

$$Y = \overline{A} + \overline{B} + C = \overline{1} + \overline{1} + 0 = 0 + 0 + 0 = 0$$

Table 3.9 shows all the fundamental sums needed to implement the truth table. Notice that each variable is complemented when the corresponding input variable is a 1; the variable is uncomplemented when the corresponding input variable is 0. To get the product-of-sums equation, all you have to do is AND the fundamental sums:

$$Y = (A + B + C)(A + \overline{B} + \overline{C})(\overline{A} + \overline{B} + C) \tag{3.31}$$

This is the product-of-sums equation for Table 3.9.

As each product term was called minterm in SOP representation in POS each sum term is called *maxterm* and is designated by M_i as shown in Table 3.9. Equation 3.31 in terms of maxterm can be represented as

$$Y = F(A, B, C) = \Pi M(0, 3, 6)$$

where 'Π' symbolizes product, i.e. AND operation. This kind of representation of a truth table is also known as *canonical product form*.

Logic Circuit

After you have a product-of-sums equation, you can get the logic circuit by drawing an OR-AND network, or if you prefer, a NOR-NOR network. In Eq. (3.31) each sum represents the output of a 3-input OR gate. Furthermore, the logical product Y is the output of a 3-input AND gate. Therefore, you can draw the logic circuit as shown in Fig. 3.26.

A 3-input OR gate is not available as a TTL chip. So, the circuit of Fig. 3.26 is not practical. With De Morgan's first theorem, however, you can replace the OR-AND circuit of Fig. 3.26 by the NOR-NOR circuit of Fig. 3.27.

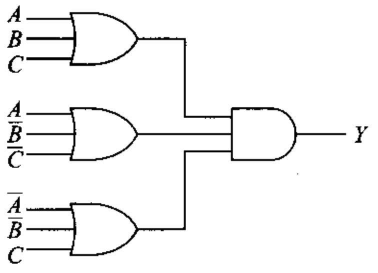


Fig. 3.26 Product-of-sums circuit

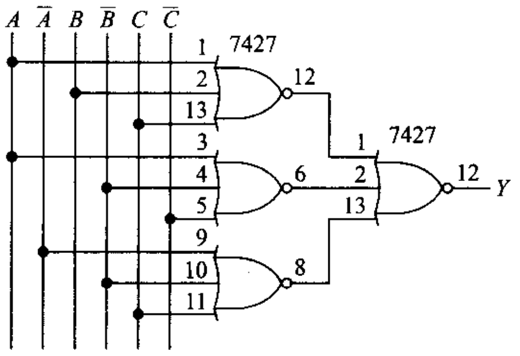


Fig. 3.27

Conversion between SOP and POS

We have seen that SOP representation is obtained by considering ones in a truth table while POS comes considering zeros. In SOP, each one at output gives one AND term which is finally ORed. In POS, each zero gives one OR term which is finally ANDed. Thus SOP and POS occupy complementary locations in a truth table and one representation can be obtained from the other by

(i) identifying complementary locations,
(ii) changing minterm to maxterm or reverse, and finally
(iii) changing summation by product or reverse.

Thus Table 3.9 can be represented as

$$Y = F(A, B, C) = \Pi M(0, 3, 6) = \Sigma m(1, 2, 4, 5, 7)$$

Similarly Table 3.4 can be represented as

$$Y = F(A, B, C) = \Sigma m(3, 5, 6, 7) = \Pi M(0, 1, 2, 4)$$

This is also known as *conversion between canonical forms.*

Example 3.9 Suppose a truth table has a low output for the first three input conditions: 000, 001, and 010. If all other outputs are high, what is the product-of-sums circuit?

Solution **The product-of-sums equation is**

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

The circuit of Fig. 3.27 will work if we reconnect the input lines as follows:

A : pins 1, 3, and 9
B : pins 2 and 4
C : pins 13 and 11
$\overline{B}$: pin 10
$\overline{C}$: pin 5

SELF-TEST

13. A product-of-sums expression leads to what kind of logic circuit?
14. Explain how to convert the complementary NAND-NAND circuit into its dual NOR-NOR circuit.

3.8 PRODUCT-OF-SUMS SIMPLIFICATION

After you write a product-of-sums equation, you can simplify it with Boolean algebra. Alternatively, you may prefer simplification based on the Karnaugh map. There are several ways of using the Karnaugh map. One can use a similar technique as followed in SOP representation but by forming largest group of zeros and then replacing each group by a sum term. The variable going in the formation of sum term is inverted if it remains constant with a value 1 in the group and it is not inverted if that value is 0. Finally, all the sum terms are ANDed to get simplest POS form. We illustrate this in Examples 3.11 and 3.12. In this section we also present an interesting alternative to above technique.

Sum-of-Products Circuit

Suppose the design starts with a truth table like Table 3.10. The first thing to do is to draw the Karnaugh map in the usual way to get Fig. 3.28a. The encircled groups allow us to write a sum-of-products equation:

$$Y = \overline{A}\,\overline{B} + AB + AC$$

Figure 3.28b shows the corresponding NAND-NAND circuit.

Complementary Circuit

To get a product-of-sums circuit, begin by complementing each 0 and 1 on the Karnaugh map of Fig. 3.28a. This results in the complemented map shown in Fig. 3.28c. The encircled 1s allow us to write the following sum-of-products equation:

$$\overline{Y} = \overline{A}B + A\overline{B}\,\overline{C}$$

Why is this $\overline{Y}$ instead of Y? Because complementing the Karnaugh map is the same as complementing the output of the truth table, which means the sum-of-products equation for Fig. 3.28c is for $\overline{Y}$ instead of Y.

Figure 3.28d shows the corresponding NAND-NAND circuit for $\overline{Y}$. This circuit does not produce the desired output; it produces the complement of the desired output.

Finding the NOR-NOR Circuit

What we want to do next is to get the product-of-sums solution, the NOR-NOR circuit that produces the

Table 3.10

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

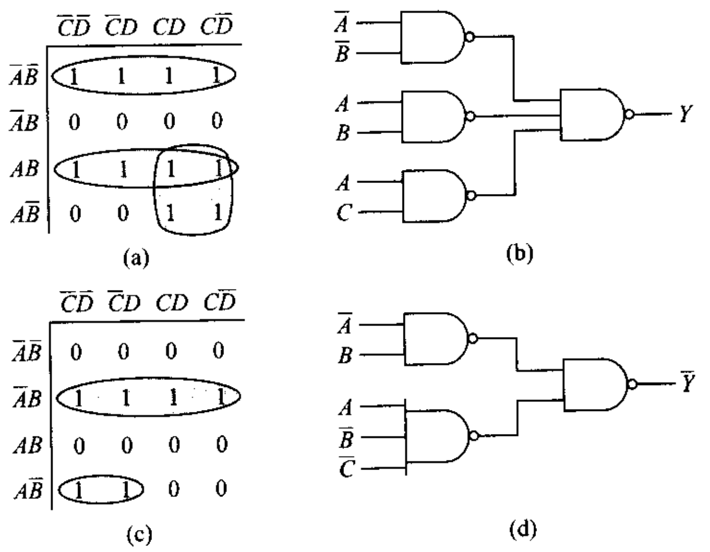


Fig. 3.28 Deriving the sum-of-products circuit

original truth table of Table 3.10. De Morgan's first theorem tells us NAND gates can be replaced by bubbled OR gates; therefore, we can replace Fig. 3.28d by Fig. 3.29a. A bus with each variable and its complement is usually available in a digital system. So, instead of connecting $\overline{A}$ and B to a bubbled OR gate, as shown in Fig. 3.29a, we can connect A and $\overline{B}$ to an OR gate, as shown in Fig. 3.29b. In a similar way, instead of connecting A, $\overline{B}$, and $\overline{C}$ to a bubbled OR gate, we have connected $\overline{A}$, B, and C to an OR gate. In short, Fig. 3.29b is equivalent to Fig. 3.29a.

The next step toward a NOR-NOR circuit is to convert Fig. 3.29b into Fig. 3.29c, which is done by sliding the bubbles to the left from the output gate to the input gates. This changes the input OR gates to NOR gates. The final step is to use a NOR gate on the output to produce Y instead of $\overline{Y}$, as shown in the NOR-NOR circuit of Fig. 3.29d.

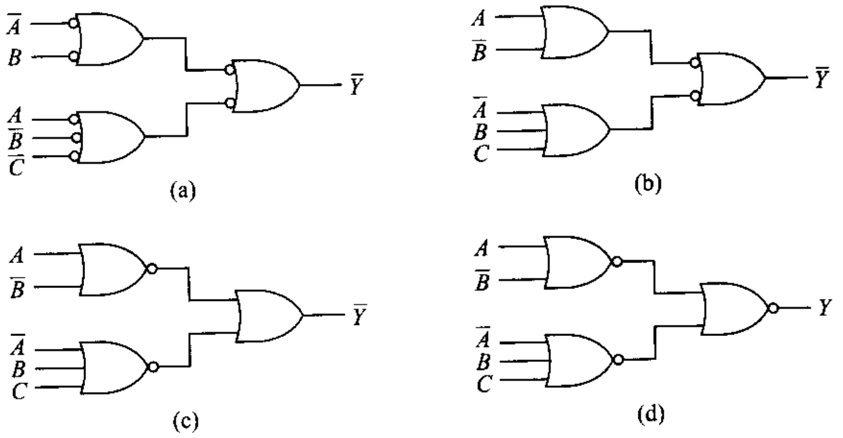


Fig. 3.29 Deriving the product-of-sums circuit

From now on, you don't have to go through every step in changing a complementary NAND-NAND circuit to an equivalent NOR-NOR circuit. Instead, you can apply the duality theorem as described in the following.

Duality

An earlier section introduced the duality theorem of Boolean algebra. Now we are ready to apply this theorem to logic circuits. Given a logic circuit, we can find its dual circuit as follows: Change each AND gate to an OR gate, change each OR gate to an AND gate, and complement all input-output signals. An equivalent statement of duality is this: Change each NAND gate to a NOR gate, change each NOR gate to a NAND gate, and complement all input-output signals.

Compare the NOR-NOR circuit of Fig. 3.29d with the NAND-NAND circuit of Fig. 3.28d. NOR gates have replaced NAND gates. Furthermore, all input and output signals have been complemented. This is an application of the duality theorem. From now on, you can change a complementary NAND-NAND circuit (Fig. 3.28d) into its dual NOR-NOR circuit (Fig. 3.29d) by changing all NAND gates to NOR gates and complementing all signals.

Points to Remember

Here is a summary of the key ideas in the preceding discussion:

1. Convert the truth table into a Karnaugh map. After grouping the 1s, write the sum-of-products equation and draw the NAND-NAND circuit. This is the sum-of-products solution for Y.
2. Complement the Karnaugh map. Group the 1s, write the sum-of-products equation, and draw the NAND-NAND circuit for $\overline{Y}$. This is the complementary NAND-NAND circuit.
3. Convert the complementary NAND-NAND circuit to a dual NOR-NOR circuit by changing all NAND gates to NOR gates and complementing all signals. What remains is the product-of-sums solution for Y.
4. Compare the NAND-NAND circuit (Step 1) with the NOR-NOR circuit (Step 3). You can use whichever circuit you prefer, usually the one with fewer gates.

Example 3.10 Show the sum-of-products and product-of-sums circuits for the Karnaugh map of Fig. 3.30a.

Solution The Boolean equation for Fig. 3.30a on the next page is

$$Y = A + BC\overline{D}$$

Figure 3.30b is the sum-of-products circuit.

After complementing and simplifying the Karnaugh map, we get Fig. 3.30c. The Boolean equation for this is

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

Figure 3.30d is the sum-of-products circuit for the $\overline{Y}$. As shown earlier, we can convert the dual circuit into a NOR-NOR equivalent circuit to get Fig. 3.30e.

The two design choices are Fig. 3.30b and 3.30e. Figure 3.30b is simpler.

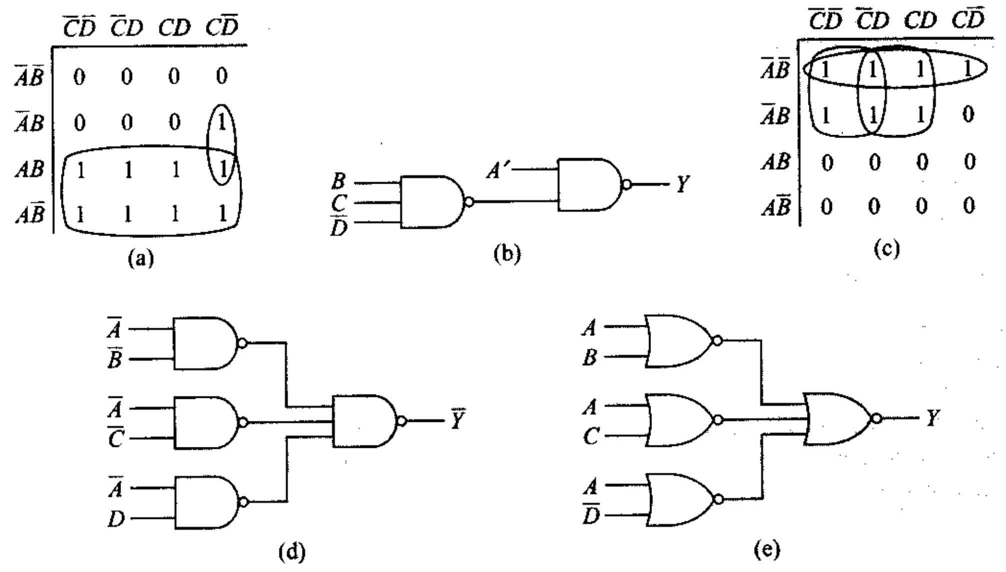


Fig. 3.30

Example 3.11 Give simplest POS form of Karnaugh map shown in Fig. 3.30a by grouping zeros.

Solution Refer to grouping of zeros as shown in Fig. 3.31a. Three groups cover all the zeros that give three sum terms. The first group has A' and C' constant within the group that gives sum term $(A + C)$. Group 2 has A' and D constant giving sum term $(A + D')$. Group 3 has A' and B' constant generating $(A + B)$ as sum term.

The final solution is thus product of these three sum terms and expressed as

$$Y = (A + B)(A + C)(A + D')$$

Note that, the above relation can be realized by OR-AND circuit or NOR-NOR (Fig. 3.30e) circuit.

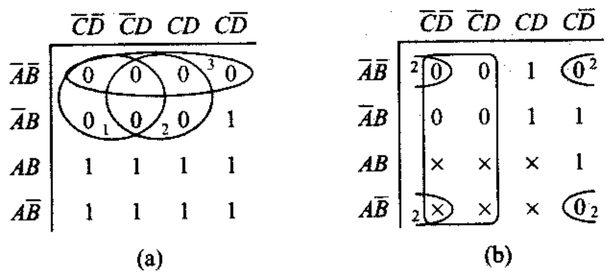


Fig. 3.31 **Simplification by grouping zeros**

Example 3.12 Give simplest POS form of Karnaugh map shown in Fig. 3.31b by grouping zeros.

Solution In a Karnaugh map if don't care conditions exist, we may consider them as zeros if that gives larger group size. This in turn reduces number of literals in the sum term. Refer to grouping of zeros in Fig. 3.31b. We require minimum two groups that includes all the zeros and are also largest in sizes. In group 1, only C' is constant that gives only one literal in sum term as C. Group 2 has B' and D' constant giving sum term $(B + D)$. The final solution is thus product of these two sum terms and expressed as

$$Y = C(B + D)$$

3.9 SIMPLIFICATION BY QUINE-McCLUSKY METHOD

Reduction of logic equation by Karnaugh map method though very simple and intuitively appealing is somewhat subjective. It depends on the user's ability to identify patterns that gives largest size. Also the method becomes difficult to adapt for simplification of 5 or more variables. Quine-McClusky method is a systematic approach for logic simplification that does not have these limitations and also can easily be implemented in a digital computer.

Determination of Prime Implicants

Quine-McClusky method involves preparation of two tables; one determines *prime implicants* and the other selects *essential prime implicants* to get minimal expression. Prime implicants are expressions with least number of literals that represents all the terms given in a truth table. Prime implicants are examined to get essential prime implicants for a particular expression that avoids any type of duplication. We illustrate the method with a 4-variable simplification problem for truth table appearing in Table 3.10. Figure 3.32 shows prime implicant determination table for the problem.

In Stage 1 of the process, we find out all the terms that gives output 1 from truth table (Table 3.10) and put them in different groups depending on how many 1 input variable combinations (*ABCD*) have. For example, first group has no 1 in input combination, second group has only one 1, third two 1s, fourth three 1s and fifth four 1s. We also write decimal equivalent of each combination to their right for convenience.

In Stage 2, we first try to combine first and second group of Stage 1, on a member to member basis. The rule is to see if only one binary digit is differing between two members and we mark that position by

Stage 1 *ABCD*		Stage 2 *ABCD*		Stage 3 *ABCD*	
0 0 0 0	(0)√	0 0 0 -	(0,1)√	0 0 - -	(0,1,2,3)
		0 0 - 0	(0,2)√	0 0 - -	(0,2,1,3)
0 0 0 1	(1)√	0 0 - 1	(1,3)√	- 0 1 -	(2,10,3,11)
0 0 1 0	(2)√	0 0 1 -	(2,3)√		
		- 0 1 0	(2,10)√		
0 0 1 1	(3)√	- 0 1 1	(3,11)√	1 - 1 -	(10,11,14,15)
1 0 1 0	(10)√	1 0 1 -	(10,11)√	1 - 1 -	(10,14,11,15)
1 1 0 0	(12)√	1 - 1 0	(10,14)√	1 1 - -	(12,13,14,15)
		1 1 0 -	(12,13)√	1 1 - -	(12,14,13,15)
		1 1 - 0	(12,14)√		
1 0 1 1	(11)√	1 - 1 1	(11,15)√		
1 1 0 1	(13)√	1 1 - 1	(13,15)√		
1 1 1 0	(14)√	1 1 1 -	(14,15)√		
1 1 1 1	(15)√				

Fig. 3.32 Determination of prime implicants

'–'. This means corresponding variable is not required to represent those members. Thus (0) of first group combines with (1) of second group to form (0,1) in Stage 2 and can be represented by $A'B'C'$ (0 0 0 –). The logic of this representation comes from the fact that minterm $A'B'C'D'$ (0) and $A'B'C'D$ (1) can be combined as $A'B'C'(D' + D) = A'B'C'$. We proceed in the same manner to find rest of the combinations in successive groups of Stage 1 and table them in Fig. 3.32. Note that, we need not look beyond successive groups to find such combinations as groups that are not adjacent, differ by more than one binary digit. Also note that each combination of Stage 2 can be represented by three literals. All the members of particular stage, which finds itself in at least one combination of next stage are tick (√) marked. This is followed for Stage 1 terms as well as terms of other stages.

In Stage 3, we combine members of different groups of Stage 2 in a similar way. Now it will have two '–' elements in each combination. This means each combination requires two literals to represent it. For example (0,1,2,3) is represented by $A'B'$ (0 0 – –). There are three other groups in Stage 3; (2,10,3,11) represented by $B'C$, (10,14,11,15) by AC and (12,13,14,15) by AB. Note that, (0,2,1,3), (10,11,14,15) and (12,14,13,15) get represented by $A'B$, AC and AB respectively and do not give any new term.

There is no Stage 4 for this problem as no two members of Stage 3 has only one digit changing among them. This completes the process of determination of prime implicants. The rule is all the terms that are not ticked at any stage is treated as prime implicants for that problem. Here, we get four of them from Stage 3, namely $A'B'$, $B'C$, AC, AB and none from previous stage as all the terms there are ticked (√).

Selection of Prime Implicants

Once we are able to determine prime implicants that covers all the terms of a truth table we try to select essential prime implicants and remove redundancy or duplication among them. For this, we prepare a table as shown in Table 3.11 that along the row lists all the prime implicants and along columns lists all minterms. The cross-point of a row and column is ticked if the term is covered by corresponding prime implicant. For example, terms 0 and 1 are covered by $A'B'$ only while 2 and 3 are covered by both $A'B'$ and $B'C$ and the corresponding cross-points are ticked. This way we complete the table for rest of the terms.

Table 3.11

	0	1	2	3	10	11	12	13	14	15
$A'B'$ (0,1,2,3)	√	√	√	√						
$B'C$ (2,3,10,11)			√	√	√	√				
AC (10,11,14,15)					√	√			√	√
AB (12,13,14,15)							√	√	√	√

Selection of essential prime implicants from this table is done in the following way. We find minimum number of prime implicants that covers all the minterms. We find $A'B'$ and AB cover terms that are not covered by others and they are essential prime implicants. $B'C$ and AC among themselves cover 10,11 which are not covered by others. So, one of them has to be included in the list of essential prime implicants making it three. And the simplified representation of truth table given in Table 3.10 is one of the following

$$Y = A'B' + B'C + AB \text{ or } Y = A'B' + AC + AB$$

Simplification of the same truth table by Karnaugh map method is shown in Fig. 3.28a and we see the results are the same.

Now, how do you compare the complexity of this approach with Karnaugh map groupings? Yes, this method is more tedious and monotonous compared to Karnaugh map method and people don't prefer it for simplification problems with smaller number of variables. However, as we have mentioned before, for simplification problems with large number of variables Quine-McClusky method can offer solution and Karnaugh map does not.

Example 3.13 Give simplified logic equation of Table 3.6 by Quine-McClusky method.

Solution Tables that determine prime implicants and selects essential prime implicants are shown in Figs. 3.33a and 3.33b respectively. We find both the prime implicants are essential prime implicants. The simplified logic equation thus is expressed as

$$Y = AB + BC'$$

Note that, we got the same expression by simplification entered variable map shown in Figs. 3.22a and 3.22b.

Stage 1		Stage 2	
A B C		*A B C*	
0 1 0	(2)√	- 1 0	(2,6)
1 1 6	(6)√	1 1 -	(6,7)
1 1 1	(7)√		

(a)

	2	6	7
BC' (2,6)	√	√	
AB (6,7)		√	√

(b)

Fig. 3.33 **Simplification by Quine-McClusky method for Example 3.14**

SELF-TEST

15. What is a prime implicant?
16. What are the advantages of Quine-McClusky method?

3.10 HAZARDS AND HAZARD COVERS

In past few sections we have discussed in detail various simplification techniques that give minimal expression for a logic equation which in turn requires minimum hardware for realization of that. It may sound off-beat, but due to some practical problems, in certain cases we may prefer to include more terms than given by simplification techniques. The discussion so far considered gates generating outputs instantaneously. But practical circuits always offer finite propagation delay though very small, in nanosecond order. This gives rise to several *hazards* and *hazard covers* are additional terms in an equation that prevents occurring of them. In this section, we discuss this problem and its solution.

Static-1 Hazard

This type of hazard occurs when $Y = A + A'$ type of situation appears for a logic circuit for certain combination of other inputs and A makes a transition $1 \to 0$. An $A + A'$ condition should always generate 1 at the output, i.e. static-1. But the NOT gate output (Fig. 3.34a) takes finite time to become 1 following $1 \to 0$ transition of A. Thus for the OR gate there are two zeros appearing at its input for that small duration, resulting a 0 at its output (Fig. 3.34b). The width of this zero is in nanosecond order and is called a glitch. For combinational circuits it may go unnoticed but in sequential circuit, more particularly in asynchronous sequential circuit (discussed in Chapter 11) it may cause major malfunctioning.

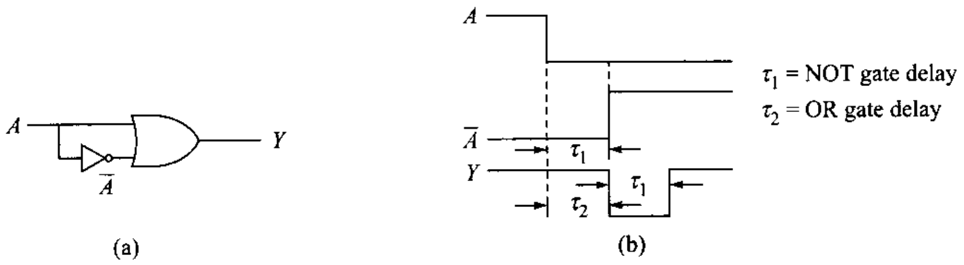


Fig. 3.34 Static-1 hazard

To discuss how we cover static-1 hazard let's look at one example. Refer to Karnaugh map shown in Fig. 3.35a, which is minimally represented by $Y = BC' + AC$. The corresponding circuit is shown in Fig. 3.35b. Consider, for this circuit input $B = 1$ and $A = 1$ and then C makes transition $1 \to 0$. The output shows glitch as discussed above. Consider another grouping for the same map in Fig. 3.35c. This includes one additional term AB and now output $Y = BC' + AC + AB$. The corresponding circuit diagram is shown in Fig. 3.35d. This circuit though require more hardware than minimal representation, is hazard free. The additional term AB ensures $Y = 1$ for $A = 1$, $B = 1$ through the third input of final OR gate and a $1 \to 0$ transition at C does not affect output. Note that, there is no other hazard possibility and inclusion of hazard cover does not alter the truth table in anyway.

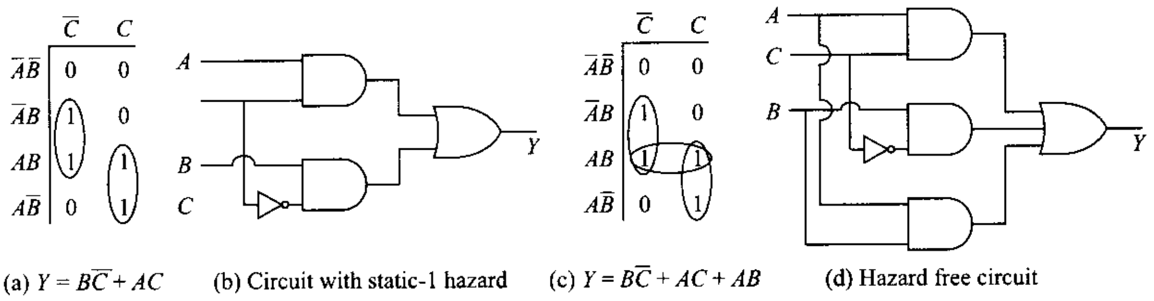


Fig. 3.35 Static-1 hazard and its cover

Again, a NAND gate with A and A' connected at its input for certain input combination will give static-1 hazard when A makes a transition $0 \rightarrow 1$ and requires hazard cover.

Static-0 Hazard

This type of hazard occurs when $Y = A.A'$ kind of situation occurs in a logic circuit for certain combination of other inputs and A makes a transition $0 \rightarrow 1$. An $A.A'$ condition should always generate 0 at the output, i.e. static-0. But the NOT gate output (Fig. 3.36a) takes finite time to become 0 following a $0 \rightarrow 1$ transition of A. Thus for final AND gate there are two ones appearing at its input for a small duration resulting a 1 at its output (Fig. 3.36b). This $Y = 1$ occurs for a very small duration (few nanosecond) but may cause malfunctioning of sequential circuit.

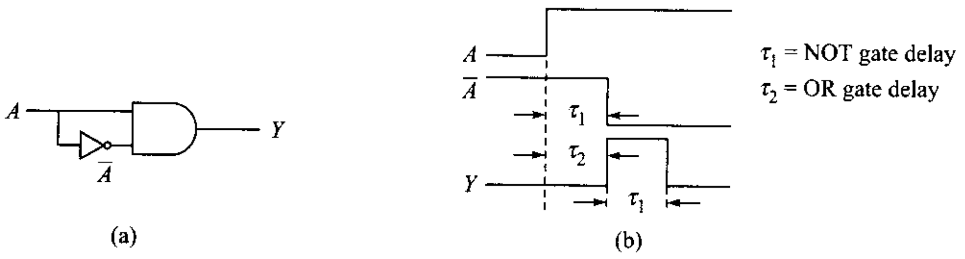
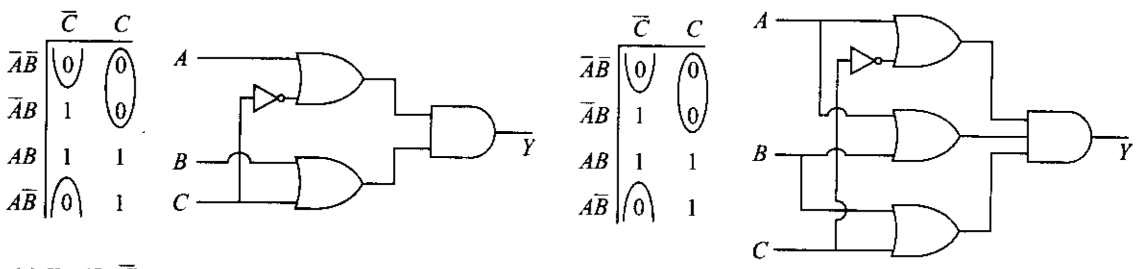


Fig. 3.36 Static-0 hazard

Again, we take an example to discuss how we can prevent static-0 hazard. We use the same truth table as shown in Fig. 3.35a but form group of 0s such that a POS form results. Figure 3.37a shows the minimal cover in POS form that gives $Y = (B + C)(A + C')$ and corresponding circuit in Fig. 3.37b. But if $B = 0, A = 0$ and C makes a transition $0 \rightarrow 1$ there will be static-0 hazard occurring at output. To prevent this we add one additional group, i.e. one more sum term $(A + B)$ as shown in Fig. 3.37c and the corresponding circuit is shown in Fig. 3.37d. The additional term $(A + B)$ ensures $Y = 0$ for $A = 0, B = 0$ through the third input of final AND gate and a $0 \rightarrow 1$ transition at C does not affect output. Again note that for this circuit there is no other hazard possibility and inclusion of hazard cover does not alter the truth table in anyway.



(a) $Y = (B+\bar{C})(A+C)$ (b) Circuit with static-0 hazard (c) $Y = (B+\bar{C})(A+C)(A+B)$ (d) Hazard free circuit

Fig. 3.37 Static-1 hazard and its cover

Also note, a NOR gate with A and A' connected at its input for certain input combination will give static-0 hazard when A makes a transition $1 \rightarrow 0$ and requires hazard cover.

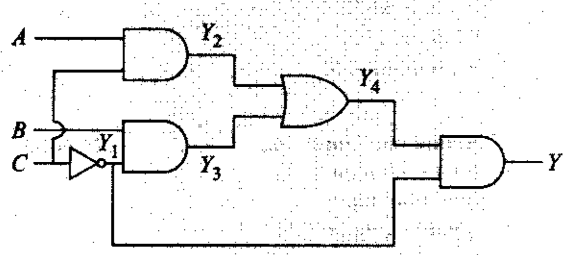
Dynamic Hazard

Dynamic hazard occurs when circuit output makes multiple transitions before it settles to a final value while the logic equation asks for only one transition. An output transition designed as $1 \rightarrow 0$ may give $1 \rightarrow 0 \rightarrow 1 \rightarrow 0$ when such hazard occurs and a $0 \rightarrow 1$ can behave like $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$. The output of logic equation in dynamic hazard degenerates into $Y = A + A'.A$ or $Y = (A + A').A$ kind of relations for certain combinations of the other input variables. As shown by these equations, these occur in multilevel circuits having implicit static-1 and/or static-0 hazards. Providing covers to each one of them dynamic hazard can be prevented.

Example 3.14 Check if the circuit shown in Fig. 3.38a exhibit dynamic hazard. Show how output varies with time if dynamic hazard occurs. Consider all the gates have equal propagation delay of τ nanosecond. Also mention how the hazard can be prevented.

Solution The logic circuit can be written in the form of equation as $Y = (A.C + B.C').C'$. Clearly for $A = 1, B = 1$ we get $Y = (C + C').C'$ which shows potential dynamic hazard with an implicit static-1 hazard. Figure 3.38b shows how a transition $1 \rightarrow 0$ at input C for $AB = 11$ causes dynamic hazard at the output.

The hazard can be prevented by using an additional two input AND gate fed by input A and B and replacing two input OR gate by a three input OR gate. The additional (third) input of OR gate will be fed by output of the new AND gate.



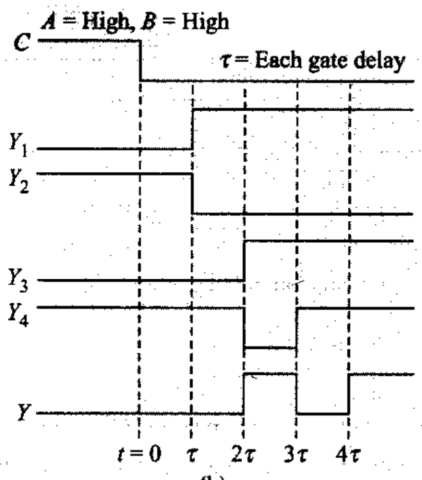


Fig. 3.38 **Example of dynamic hazard**

SELF-TEST

17. What is static-0 hazard?
18. What is dynamic hazard?

3.11 HDL IMPLEMENTATION MODELS

We continue our discussion of Verilog HDL description for a digital logic circuit from Chapter 2, Section 2.5. We have seen how structural gate level modeling easily maps a digital circuit and replicates graphical symbolic representation. We have also seen how a simple test bench can be prepared to test a designed circuit. There, we generated all possible combinations of input variables and passed it to a circuit to be tested by providing realistic gate delays. We'll follow similar test bench but more ways to describe a digital circuit in this and subsequent chapters.

Dataflow Modeling

Gate level modeling, though very convenient to get started with an HDL, consumes more space in describing a circuit and is unsuitable for large, complex design. Verilog provides a keyword **assign** and a set of operators (partial list given in Table 3.11, some operations will be explained in later chapters) to describe a circuit through its behavior or function. Here, we do not explicitly need to define any gate structure using **and**, **or** etc. and it is not necessary to use intermediate variables through **wire** showing gate level interconnections. Verilog compiler handles this while compiling such a model. All **assign** statements are concurrent, i.e. order in which they appear do not matter and also continuous, i.e. any change in a variable in the right hand side will immediately effect left hand side output.

Table 3.12 A Partial List of Verilog Operator

Relational Operation	*Symbol*	*Bit-wise Operation*	*Symbol*
Less than	<	Bit-wise NOT	~
Less than or equal to	<=	Bit-wise AND	&
Greater than	>	Bit-wise OR	|
Equal to	==	Bit-wise Ex-OR	^
Not equal to	!=		
Logical Operation (for expressions)	**Symbol**	**Arithmetic Operation**	**Symbol**
		Binary addition	+
Logical NOT	!	Binary subtraction	–
Logical AND	&&	Binary multiplication	*
Logical OR	||	Binary division	/

Now, we look at data flow model of two circuits shown in Fig. 2.17a and Fig. 2.38. We compare these codes with gate level model code presented in Section 2.5 and note the advantage. We see that data flow model resembles a logic equation and thus gives a more crisp representation.

```
module fig2_24a(A,B,C,D,Y);
  input A,B,C,D;
  output Y;
  assign Y=(A&B)|(C&D);
  //One statement is enough

endmodule
```

```
module testckt(a,b,c,x,y);
  input a,b,c;
  output x,y;
  assign x=~((a|b)|c); // NOR through NOT-OR
  assign y=~((a|b)&(b|c)); /* NAND by NOT-
  AND*/
endmodule
```

Behavioral Modeling

In a behavioral model, statements are executed sequentially following algorithmic description. It is ideally suited to describe a sequential logic circuit. However, it is also possible to describe combinatorial circuits with this but may not be a preferred model in most of the occasions. It always uses **always** keyword followed by a sensitivity list. The procedural statements following **always** is executed only if any variable within sensitivity list changes its value. Procedure assignment or output variables within **always** must be of register type, defined by **reg** which unlike **wire** is not continuously updated but only after a new value is assigned to it. Note that, **wire** variables can only be read and not assigned to in any procedural block, also it cannot store any value and must be continuously driven by output or assign statement.

Now, let us try to write behavioral code for circuit given in Fig. 2.17a. We note that, $Y = AB + CD$, i.e. $Y = 1$ if $AB = 11$ or if $CD = 11$, otherwise $Y = 0$. We use **if...else if...else** construct to describe this circuit. Here, the conditional expression after **if**, if true executes one set of instructions else executes a different set following **else** or none at all.

```
module fig2_24a(A,B,C,D,Y);
  input A,B,C,D;
  output Y;
  reg Y;         /* Y is output after procedural assignment within always
                    block, hence as reg.*/
always @ (A or B or C or D) //A,B,C,D form sensitivity list, note keyword
                                  or
  if ((A==1)&&(B == 1)) // If A,B both are 1
     Y=1;               // Assignment through equal sign not keyword assign
  else if ((C==1)&&(D == 1)) // if C,D both are 1
     Y=1;
  else                  // for all other combinations of A,B,C,D
     Y=0;
endmodule
```

You can compare how logic circuit described in Fig. 2.17a is realized in Verilog HDL following three different models two of which are described in this chapter and one in previous chapter. One might find data flow model more convenient to use for combinatorial circuits. We'll learn more about it in subsequent chapters.

Example 3.15 Realize the truth table shown in Karnaugh Map of Fig. 3.19 using data flow model.

Solution The simplified logic equation of this is given by equation 3.29 as $Y = C' + A'D' + B'D'$. We use this in assign statement to get HDL description.

```
module fig3_20a(A,B,C,D,Y);
  input A,B,C,D;
  output Y;
  assign Y= ~C | (~A & ~D) | (~B & ~D); // ~ operator has higher precedence
endmodule
```

Note that, ~ operator has higher precedence over & and |; while & and | are at same level. To avoid confusion and improve readability it is always advised to use parentheses (...) that has second highest precedence below bit select [...].

The test bench for all the examples described in this chapter can be prepared in a manner similar to what is described in Chapter 2. A simpler HDL representation to prepare a test bench will be discussed in Chapter 6.

PROBLEM SOLVING WITH MULTIPLE METHODS

Problem Get a minimized expression for $Y = F(A, B, C) = \bar{A}\,\bar{B}\,\bar{C} + \bar{A}\,\bar{B}C + \bar{A}BC + A\bar{B}C$

Solution We can solve this using Boolean Algebra, Karnaugh Map, Entered Variable Map and QM Algorithm.

In Method-1 We take help of Boolean Algebra for minimization. We see that $\bar{A}\,\bar{B}C$ can be combined with all three terms using distributive law (Eq. 3.5)

Since, in Boolean algebra $X = X + X + X$ (extending Eq. 3.7) we can write

$$Y = \bar{A}\,\bar{B}\,\bar{C} + (\bar{A}\,\bar{B}C + \bar{A}\,\bar{B}C + \bar{A}\,\bar{B}C) + \bar{A}BC + A\bar{B}C$$

From associative law (Eq. 3.3)

$$Y = (\bar{A}\,\bar{B}\,\bar{C} + \bar{A}\,\bar{B}C) + (\bar{A}\,\bar{B}C + \bar{A}BC) + (\bar{A}\,\bar{B}C + A\bar{B}C)$$

From distributive law (Eq. 3.5)

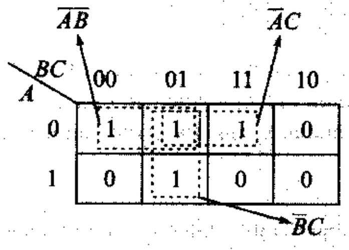
$$Y = \bar{A}\,\bar{B}(\bar{C} + C) + \bar{A}C(\bar{B} + B) + \bar{B}C(\bar{A} + A)$$

From Eq. 3.9, since $X + \bar{X} = 1$

$$Y = \bar{A}\,\bar{B} \cdot 1 + \bar{A}C \cdot 1 + \bar{B}C \cdot 1$$
$$= \bar{A}\,\bar{B} + \bar{A}C + \bar{B}C \quad \text{(since, } X \cdot 1 = X \text{ from Eq. 3.10)}$$

In Method-2, we use Karnaugh Map for minimization. Fig. 3.39 shows the solution by this method.

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



$Y = \bar{A}\bar{B} + \bar{A}C + \bar{B}C$

Fig. 3.39 **Solution using Karnaugh Map**

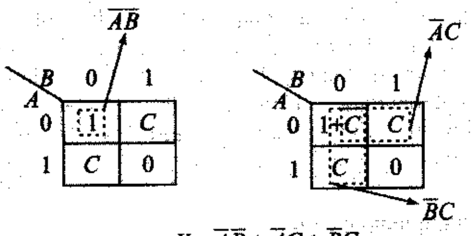
Note how one term is common in three groups formed and the similarity with Method-1 solution.

In Method-3, we use Entered Variable Map for minimization. Figure 3.40 shows the solution by this method.

Since $1 = C + \bar{C}$, we need a separate group for $AB = 00$ as $\bar{C}$ is not explained by other two groups. We use C embedded in 1 to make other two groups bigger and reduce the number of literals, and thus minimize the expression.

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

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



$Y = \overline{A}\,\overline{B} + \overline{A}C + \overline{B}C$

Fig. 3.40 Solution using Entered Variable Map

In Method-4, we use QM algorithm for minimization. Fig. 3.41 shows prime implicants and essential prime implicants. The final solution is arrived at by combining essential prime implicants.

Stage 1 ABC			Stage 2 ABC	
000	(0)	√	00–	(0, 1)
001	(1)	√	0–1	(1, 3)
			–01	(1, 5)
011	(3)	√		
101	(5)	√		

Prime implicants only from stage 2.
They are:
00–($A'B'$), 0–1 ($A'C$) and –01 ($B'C$)

	0	1	3	5
$A'B'$	√	√		
$A'C$		√	√	
$B'C$		√		√

All are essential
$Y = A'B' + A'C + B'C$

Fig. 3.41 Solution using QM Algorithm

SUMMARY

Every Boolean equation has a dual form obtained by changing OR to AND, AND to OR, 0 to 1, and 1 to 0. With Boolean algebra you may be able to simplify a Boolean equation, which implies a simplified logic circuit.

Given a truth table, you can identify the fundamental products that produce output 1s. By ORing these products, you get a sum-of-products equation for the truth table. A sum-of-products equation always results in an AND-OR circuit or its equivalent NAND-NAND circuit.

The Karnaugh method of simplification starts by converting a truth table into a Karnaugh map. Next. You encircle all the octets, quads, and pairs. This allows you to write a simplified Boolean equation and to draw a simplified logic circuit. When a truth table contains don't-cares, you can treat the don't-cares as 0s or 1s, whichever produces the greatest simplification.

One way to get a product-of-sums circuit is to complement the Karnaugh map and write the simplified Boolean equation for $\overline{Y}$. Next, you draw the NAND-NAND circuit for $\overline{Y}$. Finally, you change the NAND-NAND circuit into a NOR-NOR circuit by changing all NAND gates to NOR gates and complementing all signals.

Entered variable map maps a truth table into lower dimension space compared to Karnaugh map though the simplification procedure is similar. Quine-McClusky method provides a step-by-step approach for logic simplification and is a preferred tool that involves large number of variables. Practical digital circuit requires finite propagation delay to transfer information from input to output. This often leads to hazards in the form of unwanted glitches. Hazards are prevented by using additional gates serving as hazard cover.

GLOSSARY

- ***chip*** An integrated circuit. A piece of semiconductor material with a micro-miniature circuit on its surface.
- ***consensus theorem*** A theorem that simplifies a Boolean equation removing a redundant consensus theorem.
- ***don't-care condition*** An input-output condition that never occurs during normal operation. Since the condition never occurs, you can use an X on the Karnaugh map. This X can be a 0 or a 1, whichever you prefer.
- ***dual circuit*** Given a logic circuit, you can find it dual as follows. Change each AND (NAND) gate to an OR (NOR) gate, change each OR (NOR) gate to an AND (NAND) gate, and complement all input-output signals.
- ***Entered variable map*** an alternative to Karnaugh map where a variable is placed as output.
- ***Hazard*** unwanted glitches due to finite propagation delay of logic circuit.
- ***Hazard cover*** additional gates in logic circuit preventing hazard.
- ***Quine-McClusky method*** a tabular method for logic simplification.
- ***logic clip*** A device attached to a 14- or 16-pin DIP. The LEDs in this troubleshooting tool indicate the logic states of the pins.
- ***Karnaugh map*** A drawing that shows all the fundamental products and the corresponding output values of a truth table.
- ***octet*** Eight adjacent 1s in a 2×4 shape on a Karnaugh map.
- ***overlapping groups*** Using the same 1 more than once when looping the 1s of a Karnaugh map.
- ***pair*** Two horizontally or vertically adjacent 1s on a Karnaugh map.
- ***product-of-sums equation*** The logical product of those fundamental sums that produce output 1s in the truth table. The corresponding logic circuit is an OR-AND circuit, or the equivalent NOR-NOR circuit.
- ***quad*** Four horizontal, vertical, or rectangular 1s on a Karnaugh map.
- ***redundant group*** A group of 1s on a Karnaugh map that are all part of other groups. You can eliminate any redundant group.
- ***sum-of-products equation*** The logical sum of those fundamental products that produce output 1s in the truth table. The corresponding logic circuit is an AND-OR circuit, or the equivalent NAND-NAND circuit.

PROBLEMS

Section 3.1

3.1 Draw the logic circuit for

$$Y = A\overline{B}C + ABC$$

Next, simplify the equation with Boolean algebra and draw the simplified logic circuit.

3.2 Draw the logic circuit for

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

Use Boolean algebra to simplify the equation. Then draw the corresponding logic circuit.

3.3 In Fig. 3.42a, the output NAND gate acts like a 2-input gate because pins 10 and 11 are tied together. Suppose a logic clip is connected to the 7410. Which of the three gates is defective if the logic clip displays the data of Fig. 3.42b?

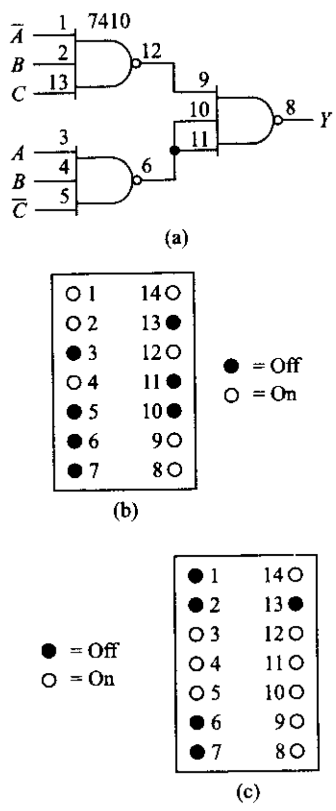


Fig. 3.42

3.4 If a logic clip displays the states of Fig. 3.42c for the circuit of Fig. 3.42a, which of the gates is faulty?

3.5 The circuit of Fig. 3.42a has trouble. If Fig. 3.43 is the timing diagram, which of the following is the trouble:

a. Upper NAND gate is defective.
b. Pin 6 is shorted to +5 V.
c. Pin 9 is grounded.
d. Pin 8 is shorted to +5 V.

Section 3.2

3.6 What is the sum-of-products circuit for the truth table of Table 3.11?

3.7 Simplify the sum-of-products equation in Prob. 3.6 as much as possible and draw the corresponding logic circuit.

3.8 A digital system has a 4-bit input from 0000 to 1111. Design a logic circuit that produces a high output whenever the equivalent decimal input is greater than 13.

3.9 We need a circuit with 2 inputs and 1 output. The output is to be high only when 1 input is high. If both inputs are high, the output is to be low. Draw a sum-of-products circuit for this.

Section 3.3

3.10 Draw the Karnaugh map for Table 3.11.

3.11 Draw the Karnaugh map for Table 3.13.

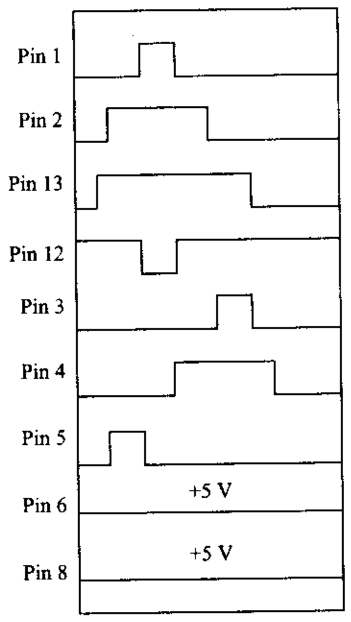


Fig. 3.43

3.12 Show Karnaugh map for equation $Y = F(A, B, C) = \Sigma m(1, 2, 3, 6, 7)$

3.13 Show Karnaugh map for equation $Y = F(A, B, C, D) = \Sigma m(1, 2, 3, 6, 8, 9, 10, 12, 13, 14)$

Section 3.4

3.14 Draw the Karnaugh map for Table 3.11. Then encircle all the octets, quads, and pairs you can find.

3.15 Repeat Prob. 3.14 for Table 3.14.

Section 3.5

3.16 What is the simplified Boolean equation for the Karnaugh map of Table 3.13? The logic circuit?

3.17 Given Table 3.14, use Karnaugh simplification and draw the simplified logic circuit.

3.18 Table 3.15 on the next page shows a special code known as the *Gray code*. For each binary input $ABCD$, there is a corresponding Gray-code output. What is the simplified sum-of-products equation for Y_3? For Y_2? For Y_1? For Y_0? Draw a logic circuit that converts a 4-bit binary input to a Gray-code output.

Section 3.6

3.19 Suppose the last six entries of Table 3.11 are changed to don't-cares. Using the Karnaugh map, show the simplified logic circuit.

3.20 Assume the first six entries of Table 3.13 are changed to don't-cares. What is the simplified logic circuit?

3.21 Suppose the inputs 1010 through 1111 only appear when there is trouble in a digital system. Design a logic circuit that detects the presence of any nibble input from 1010 to 1111.

Section 3.7

3.22 Draw the unsimplified product-of-sums circuit for Table 3.11.

3.23 Repeat Prob. 3.20 for Table 3.13.

3.24 Draw a NOR-NOR circuit for this Boolean expression:

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

3.25 Give SOP form of $Y = F(A, B, C, D) = \Pi\, M(0, 3, 4, 5, 6, 7, 11, 15)$

3.26 Draw Karnaugh map of $Y = F(A, B, C, D) = \Pi\, M(0, 1, 3, 8, 9, 10, 14, 15)$

Table 3.13

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

Table 3.14

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

Table 3.15 **Gray Code**

A	B	C	D	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

Section 3.8

3.27 What is the simplified NOR-NOR circuit for Table 3.11?

3.28 Draw the simplified NOR-NOR circuit for Table 3.13.

3.29 Figure 3.44 shows all the input waveforms for the timing diagram of Fig. 3.30e. Draw the waveform for the output Y.

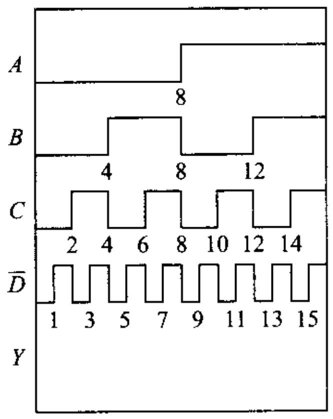


Fig. 3.44

3.30 You are given the following Boolean equation

$$Y = \overline{A}B\overline{C}D + \overline{A}\,\overline{B}C\overline{D} + A\overline{B}C\overline{D}$$

Show the simplified NAND-NAND circuit for this. Also, show the simplified NOR-NOR circuit.

3.31 Table 3.16 is the truth table of *full adder*, a logic circuit with two outputs called the *CARRY* and the *SUM*. What is the simplified NAND-NAND circuit for the *CARRY* output? For the *SUM* output?

3.32 Repeat Prob. 3.27 using NOR-NOR circuits

Table 3.16 **Full-Adder Truth Table**

A	B	C	*Carry*	*Sum*
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

3.33 Simplify to give POS form by grouping zeros in Karnaugh map for equation given in problem 3.27.

3.34 Simplify to give POS form by grouping zeros in Karnaugh map for equation given in problem 3.28.

Sections 3.9 and 3.10

3.35 Get simplified expression of $Y = F(A, B, C, D) = \Sigma\, m(1, 2, 8, 9, 10, 12, 13, 14)$ using Quine-McClusky method.

3.36 Get simplified expression of $Y = F(A, B, C, D, E) = \Sigma\, m(0, 1, 2, 3, 4, 5, 12, 13, 14, 26, 27, 28, 29, 30)$ using Quine-McClusky method.

3.37 For the following Karnaugh map give SOP and POS form that do not show static-0 or static-1 hazard.

	$\overline{C}$	C
$\overline{A}\overline{B}$	1	1
$\overline{A}B$	0	0
AB	1	0
$A\overline{B}$	1	0

3.38 Verify with timing diagram if the following circuit shows dynamic hazard.

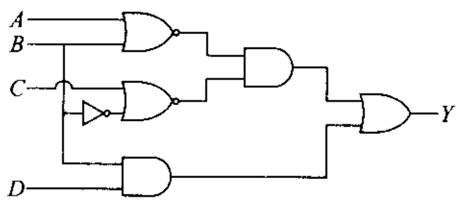


Fig. 3.45

LABORATORY EXPERIMENT

AIM: The aim of this experiment is to verify De Morgan's theorems

Theory: De Morgan's two theorems are

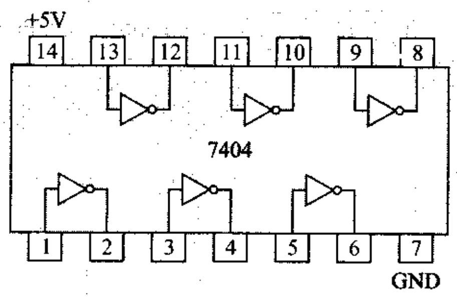
$$(A + B)' = A' \cdot B'$$

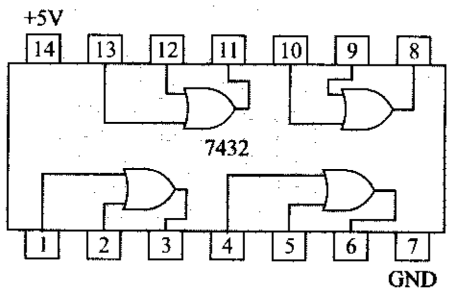
and $$(A \cdot B)' = A' + B'$$

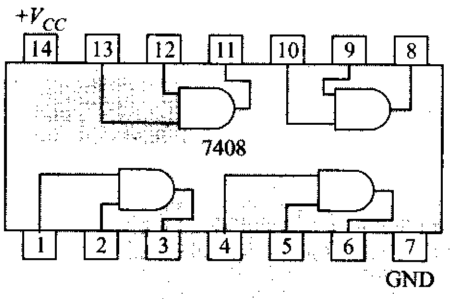
NAND gate and NOR gate can be used to generate the left hand side of the two equations while NOT gate, AND gate and OR gate can be used to generate the right hand side.

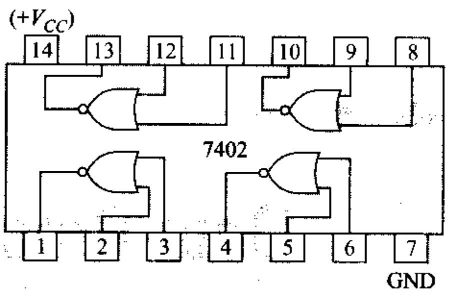
Apparatus: 5 V DC Power supply, Multimeter, and Bread Board

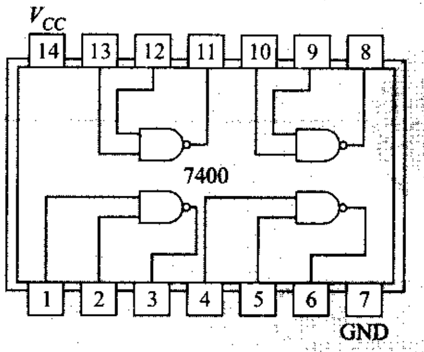
Work element: Verify the truth table of IC 7404, 7408, 7432, 7402 and 7400. Interconnect them in such a manner so that right hand sides of the equations are implemented. Find its truth table. Compare it with truth table of NOR and NAND gates.

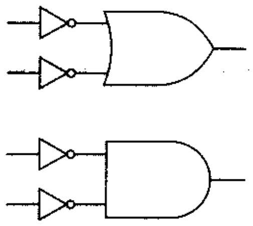












Answers to Self-tests

1. False
2. $Y = AB + AC$
3. $Y = Q$
4. Four, eight
5. False
6. A Karnaugh map is a visual display of the fundamental products needed for a sum-of-products solution.
7. Sixteen
8. Pair
9. Eight
10. $Y = \overline{A}B\overline{C}\overline{D} + AB\overline{C}\overline{D} + \overline{A}BC\overline{D} + ABC\overline{D}$. Simplify as $Y = B\overline{D}$
11. A don't-care condition is an input condition that never occurs during normal operations, and it is indicated with an X.
12. An X can be used to create pairs, quads, octets, etc.
13. A product-of-sums expression leads directly to an OR-AND circuit.
14. Change all NAND gates to NOR gates, and complement all signals (see Example 3.10).
15. Prime implicants are expressions with least number of literals that represents all the terms given in a truth table.
16. Systematic, step-by-step approach that can be implemented in a digital computer and providing solution for any number of variables.
17. A logic high pulse of very short duration when output should be at logic low.
18. Dynamic hazard occurs when circuit output makes multiple transitions before it settles while the logic equation asks for only one transition.

Data-Processing Circuits

4

OBJECTIVES

- **Determine the output of a multiplexer or demultiplexer based on input conditions.**
- **Find, based on input conditions, the output of an encoder or decoder.**
- **Draw the symbol and write the truth table for an exclusive-OR gate.**
- **Explain the purpose of parity checking.**
- **Show how a magnitude comparator works.**
- **Describe a ROM, PROM, EPROM, PAL, and PLA.**

This chapter is about logic circuits that process binary data. We begin with a discussion of multiplexers, which are circuits that can select one of many inputs. Then you will see how multiplexers are used as a design alternative to the sum-of-products solution. This will be followed by an examination of a variety of circuits, such as demultiplexers, decoders, encoders, exclusive-OR gates, parity checkers, magnitude comparator, and read-only memories. The chapter ends with a discussion of programmable logic arrays and relevant HDL concepts.

4.1 MULTIPLEXERS

Multiplex means *many into one*. A *multiplexer* is a circuit with many inputs but only one output. By applying control signals, we can steer any input to the output. Thus it is also called a *data selector* and control inputs are termed select inputs. Figure 4.1a illustrates the general idea. The circuit has n input signals, m control signals and 1 output signal. Note that, m control signals can select at the most 2^m input signals thus $n \leq 2^m$.

The circuit diagram of a 4-to-1 multiplexer is shown in Fig. 4.1c and its truth table in Fig. 4.1b. Depending on control inputs A, B one of the four inputs D_0 to D_3 is steered to output Y.

Let us write the logic equation of this circuit. Clearly, it will give a SOP representation, each AND gate generating a product term, which finally are summed by OR gate. Thus,

$$Y = A'B'.D_0 + A'B.D_1 + AB'.D_2 + AB.D_3$$

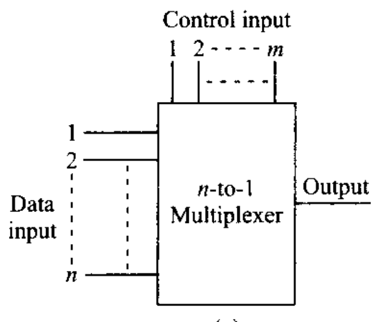
If $A = 0, B = 0$, $\quad Y = 0'0'.D_0 + 0'.0.D_1 + 0.0'.D_2 + 0.0.D_3$

or, $\quad Y = 1.1.D_0 + 1.0.D_1 + 0.1.D_2 + 0.0.D_3$

or, $\quad Y = D_0$

In other words, for $AB = 00$, the first AND gate to which D_0 is connected remains active and equal to D_0 and all other AND gate are inactive with output held at logic 0. Thus, multiplexer output Y is same as D_0. If $D_0 = 0$, $Y = 0$ and if $D_0 = 1$, $Y = 1$.

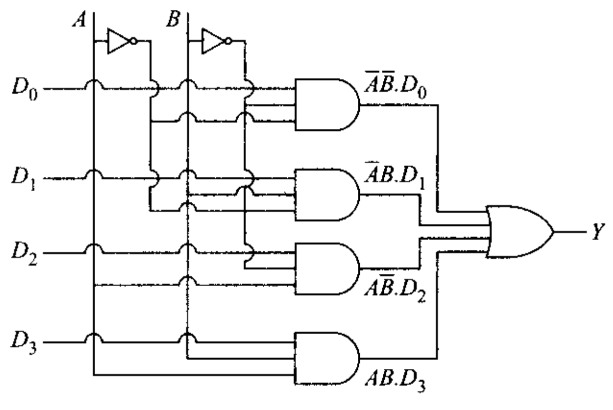
Similarly, for $AB = 01$, second AND gate will be active and all other AND gates remain inactive. Thus, output $Y = D_1$. Following same procedure we can complete the truth table of Fig. 4.1b.



(a)

A	B	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

(b)



(c)

Fig. 4.1 (a) Multiplexer block diagram, (b) 4-to-1 multiplexer truth table, (c) Its logic circuit

Now, if we want 5-to-1 multiplexer how many select lines are required? There is no 5th combination possible with two select lines and hence we need a third select input. Note that, with three we can select up to $2^3 = 8$ data inputs. Commercial multiplexers ICs come in integer power of 2, e.g. 2-to-1, 4-to-1, 8-to-1, 16-to-1 multiplexers. With this background, let us look at a 16-to-1 multiplexer circuit, which may look complex but follows same logic as that of a 4-to-1 multiplexer.

16-to-1 Multiplexer

Figure shows a 16-to-l multiplexer. The input bits are labeled D_0 to D_{15}. Only one of these is transmitted to the output. Which one depends on the value of $ABCD$, the control input. For instance, when

$$ABCD = 0000$$

the upper AND gate is enabled while all other AND gates are disabled. Therefore, data bit D_0 is transmitted to the output, giving

$$Y = D_0$$

If D_0 is low, Y is low; if D_0 is high, Y is high. The point is that Y depends only on the value of D_0.
If the control nibble (group of 4-bits) is changed to

$$ABCD = 1111$$

all gates are disabled except the bottom AND gate. In this case, D_{15} is the only bit transmitted to the output, and

$$Y = D_{15}$$

As you can see, the control nibble determines which of the input data bits is transmitted to the output.
Thus we can write output as

$$Y = A'B'C'D'.D_0 + A'B'C'D.D_1 + A'B'CD'.D_2 + \ldots + ABCD'.D_{14} + ABCD.D_{15}$$

At this point can we answer, how would an 8 to 1 multiplexer circuit look like? First of all we need three select lines for 8 data inputs. And there will be 8 AND gates each one having four inputs; three from select lines and one from data input. The final output is generated from an OR gate which takes input from 8 AND gates. The equation for this can be written as

$$Y = A'B'C'.D_0 + A'B'C.D_1 + A'BC'.D_2 + A'BC.D_3 + AB'C'.D_4 + AB'C.D_5 + ABC'.D_6 + ABC.D_7$$

Thus, for $ABC = 000$, multiplexer output $Y = D_0$; other AND gates and corresponding data inputs D_1 to D_7 remain inactive. Similarly, for $ABC = 001$, multiplexer output $Y = D_1$, for $ABC = 010$, multiplexer output $Y = D_2$ and finally, for $ABC = 111$, multiplexer output $Y = D_7$.

The 74150

Try to visualize the 16-input OR gate of Fig. 4.2 changed to a NOR gate. What effect does this have on the operation of the circuit? Almost none. All that happens is we get the complement of the selected data bit rather than the data bit itself. For instance, when $ABCD = 0111$, the output is

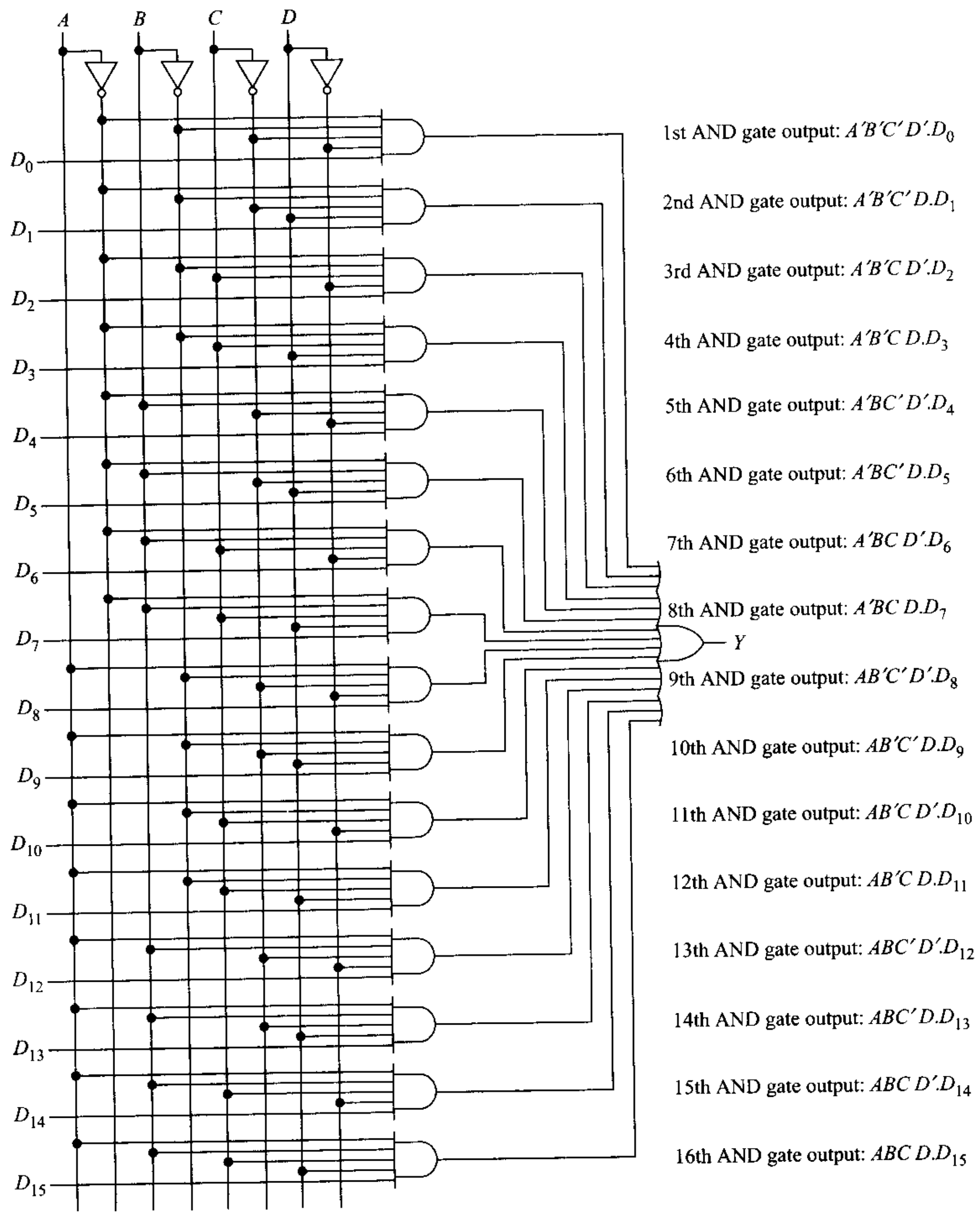
$$Y = \overline{D_7}$$

This is the Boolean equation for a typical transistor-transistor logic (TTL) multiplexer because it has an inverter on the output that produces the complement of the selected data bit.

The 74150 is a 16-to-1 TTL multiplexer with the pin diagram shown in Fig. 4.3. Pins 1 to 8 and 16 to 23 are for the input data bits D_0 to D_{15}. Pins 11, 13, 14, and 15 are for the control bits $ABCD$. Pin 10 is the output; and it equals the complement of the selected data bit. Pin 9 is for the STROBE, an input signal that disables or enables the multiplexer. As shown in Table 4.1, a low strobe enables the multiplexer, so that output Y equals the complement of the input data bit:

$$Y = \overline{D_n}$$

where n is the decimal equivalent of $ABCD$. On the other hand, a high strobe disables the multiplexer and forces the output into the high state. With a high strobe, the value of $ABCD$ doesn't matter.



A
B
C
D
D_0
D_1
D_2
D_3
D_4
D_5
D_6
D_7
D_8
D_9
D_{10}
D_{11}
D_{12}
D_{13}
D_{14}
D_{15}
1st AND gate output: $A'B'C'D'.D_0$
2nd AND gate output: $A'B'C'D.D_1$
3rd AND gate output: $A'B'C\,D'.D_2$
4th AND gate output: $A'B'C\,D.D_3$
5th AND gate output: $A'BC'D'.D_4$
6th AND gate output: $A'BC'D.D_5$
7th AND gate output: $A'BC\,D'.D_6$
8th AND gate output: $A'BC\,D.D_7$
Y
9th AND gate output: $AB'C'D'.D_8$
10th AND gate output: $AB'C'D.D_9$
11th AND gate output: $AB'C\,D'.D_{10}$
12th AND gate output: $AB'C\,D.D_{11}$
13th AND gate output: $ABC'D'.D_{12}$
14th AND gate output: $ABC'D.D_{13}$
15th AND gate output: $ABC\,D'.D_{14}$
16th AND gate output: $ABC\,D.D_{15}$

Fig. 4.2 **Sixteen-to-one multiplexer**

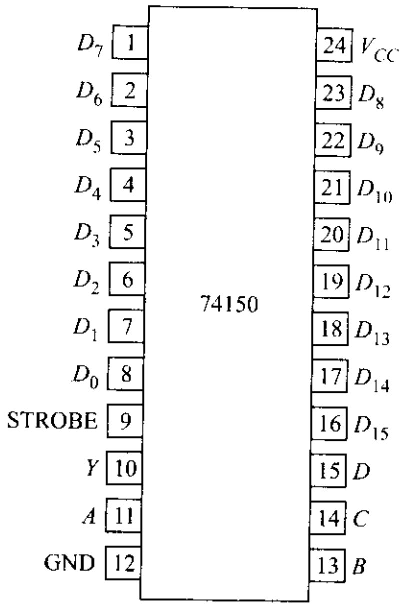


Fig. 4.3 Pinout diagram of 74150

Table 4.1 74150 Truth Table

Strobe	*A*	*B*	*C*	*D*	*Y*
L	*L*	*L*	*L*	*L*	$\bar{D}_0$
L	*L*	*L*	*L*	*H*	$\bar{D}_1$
L	*L*	*L*	*H*	*L*	$\bar{D}_2$
L	*L*	*L*	*H*	*H*	$\bar{D}_3$
L	*L*	*H*	*L*	*L*	$\bar{D}_4$
L	*L*	*H*	*L*	*H*	$\bar{D}_5$
L	*L*	*H*	*H*	*L*	$\bar{D}_6$
L	*L*	*H*	*H*	*H*	$\bar{D}_7$
L	*H*	*L*	*L*	*L*	$\bar{D}_8$
L	*H*	*L*	*L*	*H*	$\bar{D}_9$
L	*H*	*L*	*H*	*L*	$\bar{D}_{10}$
L	*H*	*L*	*H*	*H*	$\bar{D}_{11}$
L	*H*	*H*	*L*	*L*	$\bar{D}_{12}$
L	*H*	*H*	*L*	*H*	$\bar{D}_{13}$
L	*H*	*H*	*H*	*L*	$\bar{D}_{14}$
L	*H*	*H*	*H*	*H*	$\bar{D}_{15}$
H	*X*	*X*	*X*	*X*	*H*

Multiplexer Logic

Digital design usually begins with a truth table. The problem is to come up with a logic circuit that has the same truth table. In Chapter 3, you saw two standard methods for implementing a truth table: the sum-of-products and the product-of-sums solutions. The third method is the *multiplexer solution*. For example, to use a 74150 to implement Table 4.2. Complement each Y output to get the corresponding data input:

$$D_0 = \bar{1} = 0$$
$$D_1 = \bar{0} = 1$$
$$D_2 = \bar{1} = 0$$

and so forth, up to

$$D_{15} = \bar{1} = 0$$

Next, wire the data inputs of 74150 as shown in Fig. 4.4, so that they equal the foregoing values. In other words, D_0 is grounded, D_1 is connected to +5 V, D_2 is grounded, and so forth. In each of these cases, the data input is the complement of the desired Y output of Table 4.2.

Figure 4.4 is the multiplexer design solution. It has the same truth table given in Table 4.2. If in doubt, analyze it as follows for each input condition. When $ABCD$ = 0000, D_0 is the selected input in Fig. 4.4. Since D_0 is low, Y is high. When $ABCD$ = 0001, D_1 is selected. Since D_1 is high, Y is low. If you check the remaining input possibilities, you will see that the circuit has the truth table given in Table 4.2.

Bubbles on Signal Lines

Data sheets often show inversion bubbles on some of the signal lines. For instance, notice the bubble on pin 10, the output of Fig. 4.4. This bubble is a reminder that the output is the complement of the selected data bit.

Table 4.2

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

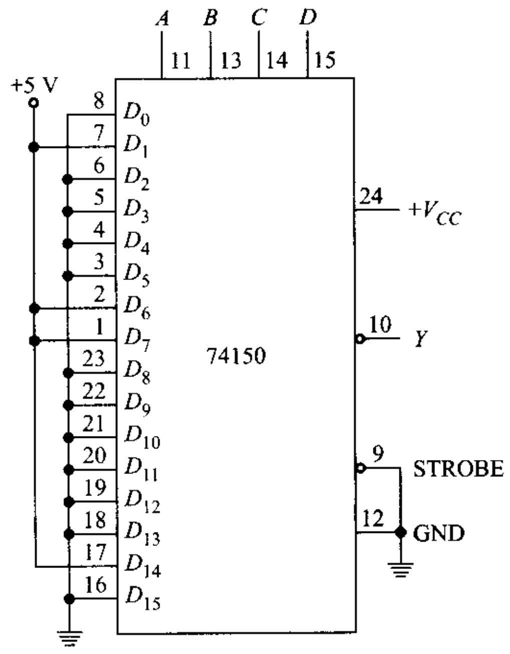


Fig. 4.4 **Using a 74150 for multiplexer logic**

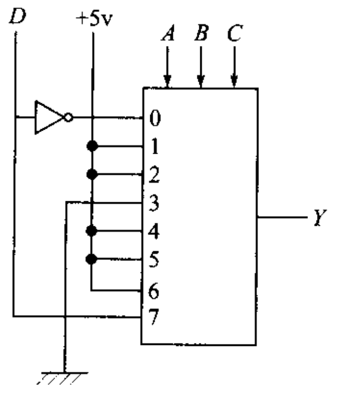
Also notice the bubble on the STROBE input (pin 9). As discussed earlier, the multiplexer is active (enabled) when the STROBE is low and inactive (disabled) when it is high. Because of this, the STROBE is called an *active-low signal*; it causes something to happen when it is low rather than when it is high. Most schematic diagrams use bubbles to indicate active-low signals. From now on, whenever you see a bubble on an input pin, remember that it means the signal is active-low.

Universal Logic Circuit

Multiplexer sometimes is called *universal logic circuit* because a 2^n-to-1 multiplexer can be used as a design solution for any n variable truth table. This we have seen for realization of a 4 variable truth table by 16-to-1 multiplexer in Fig. 4.5. Here, we show how this truth table can be realized using an 8-to-1 multiplexer. Let's consider A,B and C variables to be fed as select inputs. The fourth variable D then has to be present as data input. The method is shown in Fig. 4.5a. The first three rows map the truth table in a different way, similar to the procedure we adopted in entered variable map (Section 3.3). We write all the combinations of 3 select inputs in first row along different columns. Now corresponding to each value of 4th variable D, truth table

ABC	000	001	010	011	100	101	110	111
$D = 0$	1	1	1	0	1	1	1	0
$D = 1$	0	1	1	0	1	1	1	1
Y	D'	1	1	0	1	1	1	D
8-to-1 MUX data input	$D_0 = D'$	$D_1 = 1$	$D_2 = 1$	$D_3 = 0$	$D_4 = 1$	$D_5 = 1$	$D_6 = 1$	$D_7 = D$

(a)



(b)

Fig. 4.5 A four variable truth table realization using 8-to-1 multiplexer

output Y is written in 2nd and 3rd row. The 4th row writes Y as a function of D. In fifth row we assign data input values for 8-to-1 multiplexer simply copying Y values obtained in previous row. This is because for each select variable combination a multiplexer transfers a particular input to its output. In 8-to-1 multiplexer, $ABC = 000$ selects D_0, $ABC = 001$ selects D_1 and so on. The corresponding circuit is shown in Fig. 4.5b.

Note that, we can choose any of the four variables (A,B,C,D) of truth table to feed as input to 8-to-1 multiplexer but then mapping in first three rows of Fig. 4.5a will change. The rest of the procedure will remain same. We show an alternative to this technique for a new problem in Example 4.2.

Nibble Multiplexers

Sometimes we want to select one of two input nibbles. In this case, we can use a nibble multiplexer like the one shown in Fig. 4.6. The input nibble on the left is $A_3A_2A_1A_0$ and the one on the right is $B_3B_2B_1B_0$. The control signal labeled *SELECT* determines which input nibble is transmitted to the output. When SELECT is low, the four NAND gates on the left are activated; therefore,

$$Y_3Y_2Y_1Y_0 = A_3A_2A_1A_0$$

When SELECT is high, the four NAND gates on the right are active, and

$$Y_3Y_2Y_1Y_0 = B_3B_2B_1B_0$$

Figure 4.7a on the next page shows the pinout diagram of a 74157, a nibble multiplexer with a SELECT input as previously described. When SELECT is low, the left nibble is steered to the output. When SELECT

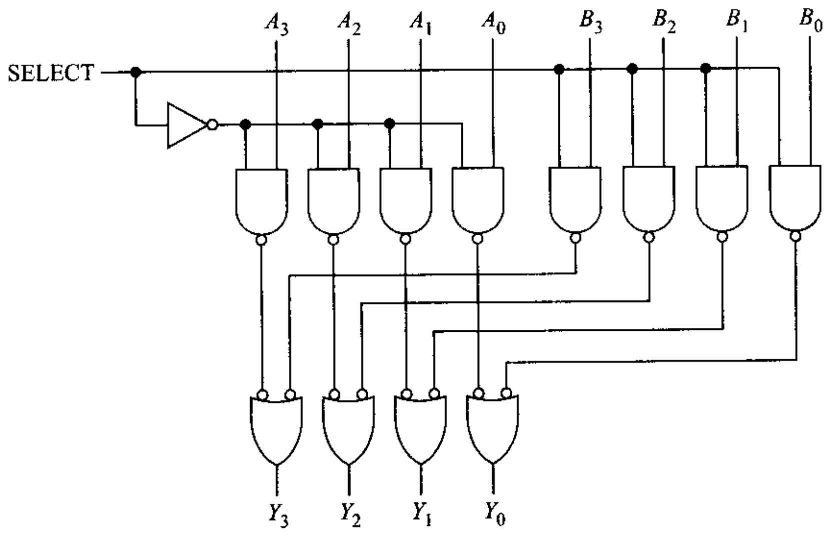


Fig. 4.6 Nibble multiplexer

is high, the right nibble is steered to the output. The 74157 also includes a strobe input. As before, the strobe must be low for the multiplexer to work properly. When the strobe is high, the multiplexer is inoperative.

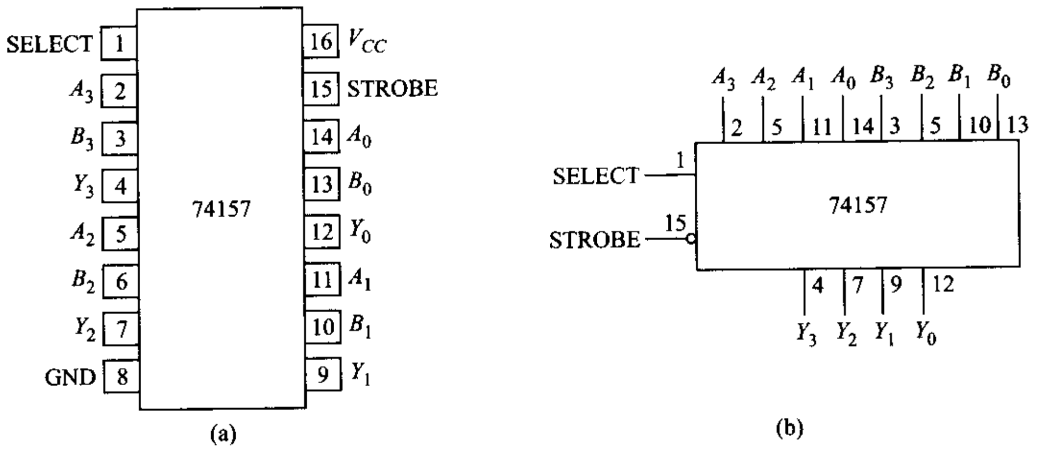


Fig. 4.7 Pinout diagram of 74157

Figure 4.7b shows how to draw a 74157 on a schematic diagram. The bubble on pin 15 tells us that STROBE is an active-low input.

Example 4.1 Show how 4-to-1 multiplexer can be obtained using only 2-to-1 multiplexer.

Solution

Logic equation for 2-to-1 Multiplexer: $Y = A'.D_0 + A.D_1$

Logic equation for 4-to-1 Multiplexer: $Y = A'B'.D_0 + A'B.D_1 + AB'.D_2 + AB.D_3$

This can be rewritten as, $Y = A'(B'.D_0 + B.D_1) + A(B'.D_2 + B.D_3)$

Compare this with equation of 2-to-1 multiplexer. We need two 2-to-1 multiplexer to realize two bracketed terms where B serves as select input. The output of these two multiplexers can be sent to a third multiplexer as data inputs where A serves as select input and we get the 4-to-1 multiplexer. Figure 4.8a shows circuit diagram for this.

Example 4.2

(a) Realize $Y = A'B + B'C' + ABC$ using an 8-to-1 multiplexer. (b) Can it be realized with a 4-to-1 multiplexer?

Solution

(a) First we express Y as a function of minterms of three variables. Thus

$$Y = A'B + B'C' + ABC$$
$$Y = A'B(C' + C) + B'C'(A' + A) + ABC \text{ [As, } X + X' = 1]$$
$$Y = A'B'C' + A'BC' + A'BC + AB'C' + ABC$$

Comparing this with equation of 8 to 1 multiplexer, we find by substituting $D_0 = D_2 = D_3 = D_4 = D_7 = 1$ and $D_1 = D_5 = D_6 = 0$ we get given logic relation.

(b) Let variables A and B be used as selector in 4 to 1 multiplexer and C fed as input. The 4-to-1 multiplexer generates 4 minterms for different combinations of AB. We rewrite given logic equation in such a way that all these terms are present in the equation.

$$Y = A'B + B'C' + ABC$$
$$Y = A'B + B'C'(A' + A) + ABC \text{ [As, } X + X' = 1]$$
$$Y = A'B'.C' + A'B.1 + AB'.C' + AB.C$$

Compare above with equation of a 4-to-1 multiplexer. We see $D_0 = C'$, $D_1 = 1$, $D_2 = C'$ and $D_3 = C$ generate the given logic function.

Example 4.3 Design a 32-to-1 multiplexer using two 16-to-1 multiplexers and one 2-to-1 multiplexer.

Solution The circuit diagram is shown in Fig. 4.8b. A 32-to-1 multiplexer requires $\log_2 32 = 5$ select lines say, $ABCDE$. The lower 4 select lines $BCDE$ chose 16-to-1 multiplexer outputs. The 2-to-1 multiplexer chooses one of the output of two 16-to-1 multiplexers depending on what appears in the 5th select line, A.

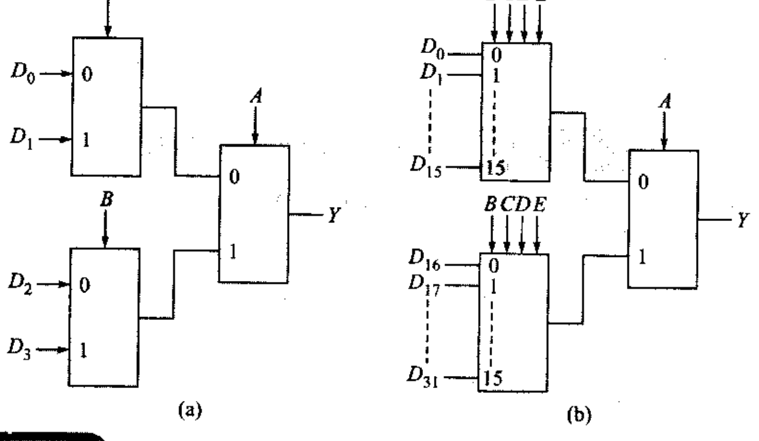


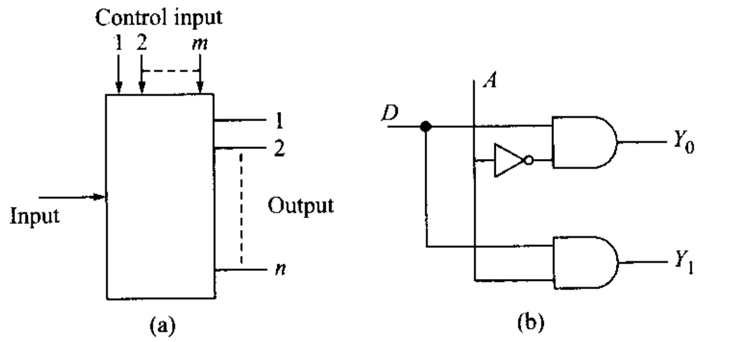
Fig. 4.8 **Realization of higher order multiplexers using lower orders**

SELF-TEST

1. A circuit with many inputs but only one output is called a _____.
2. What is the significance of the bubble on pin 10 of the multiplexer in Fig. 4.5?

4.2 DEMULTIPLEXERS

Demultiplex means *one into many*. A *demultiplexer* is a logic circuit with one input and many outputs. By applying control signals, we can steer the input signal to one of the output lines. Figure 4.9a illustrates the general idea. The circuit has 1 input signal, m control or select signals and n output signals where $n \leq 2^m$. Figure 4.9b shows the circuit diagram of a 1-to-2 demultiplexer. Note the similarity of multiplexer and demultiplexer circuits in generating different combinations of control variables through a bank of AND gates. Figure 4.9c lists some of the commercially available demultiplexer ICs. Note that a demultiplexer IC can also behave like a decoder. More about this will be discussed in next section.



IC No.	DEMUX Type	Decoder Type
74154	1-to-16	4-to-16
74138	1-to-8	3-to-8
74155	1-to-4	2-to-4

(c)

Fig. 4.9 (a) Demultiplexer block diagram, (b) Logic circuit of 1-to-2 demultiplexer, (c) Few commercially available ICs

1-to-16 Demultiplexer

Figure 4.10 shows a l-to-16 demultiplexer. The input bit is labeled D. This data bit (D) is transmitted to the data bit of the output lines. But which one? Again, this depends on the value of $ABCD$, the control input. When $ABCD$ = 0000, the upper AND gate is enabled while all other AND gates are disabled. Therefore, data bit D is transmitted only to the Y_0 output, giving $Y_0 = D$. If D is low, Y_0 is low. If D is high, Y_0 is high. As you can see, the value of Y_0 depends on the value of D. All other outputs are in the low state. If the control nibble is changed to $ABCD$ = 1111, all gates are disabled except the bottom AND gate. Then, D is transmitted only to the Y_{15} output, and $Y_{15} = D$.

The 74154

The 74154 is a 1-to-16 demultiplexer with the pin diagram of Fig. 4.11. Pin 18 is for the input DATA D, and pins 20 to 23 are for the control bits $ABCD$. Pins 1 to 11 and 13 to 17 are for the output bits Y_0 to Y_{15}. Pin 19 is for the STROBE, again an active-low input. Finally, pin 24 is for V_{CC} and pin 12 for ground.

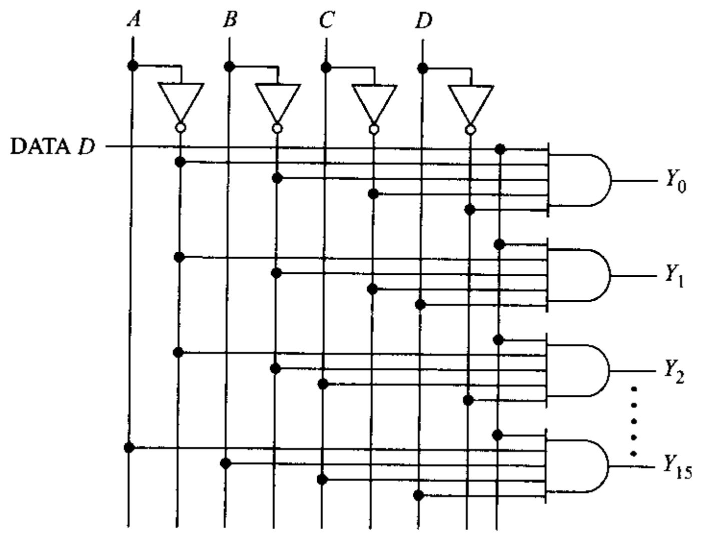


Fig. 4.10 **1-to-16 demultiplexer**

Table 4.3 shows the truth table of a 74154. First, notice the STROBE input. It must be low to activate the 74154. When the STROBE is low, the control input *ABCD* determines which output lines are low when the DATA input is low. When the DATA input is high, all output lines are high. And, when the STROBE is high, all output lines are high.

Table 4.3 **74154 Truth Table**

Strobe	*Data*	*A*	*B*	*C*	*D*	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7	Y_8	Y_9	Y_{10}	Y_{11}	Y_{12}	Y_{13}	Y_{14}	Y_{15}
L	L	L	L	L	L	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
L	L	L	L	L	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H
L	L	L	L	H	L	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H
L	L	L	L	H	H	H	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H
L	L	L	H	L	L	H	H	H	H	L	H	H	H	H	H	H	H	H	H	H	H
L	L	L	H	L	H	H	H	H	H	H	L	H	H	H	H	H	H	H	H	H	H
L	L	L	H	H	L	H	H	H	H	H	H	L	H	H	H	H	H	H	H	H	H
L	L	L	H	H	H	H	H	H	H	H	H	H	L	H	H	H	H	H	H	H	H
L	L	H	L	L	L	H	H	H	H	H	H	H	H	L	H	H	H	H	H	H	H
L	L	H	L	L	H	H	H	H	H	H	H	H	H	H	L	H	H	H	H	H	H
L	L	H	L	H	L	H	H	H	H	H	H	H	H	H	H	L	H	H	H	H	H
L	L	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H	L	H	H	H	H
L	L	H	H	L	L	H	H	H	H	H	H	H	H	H	H	H	H	L	H	H	H
L	L	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	L	H	H
L	L	H	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	L	H
L	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	L
L	H	X	X	X	X	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
H	L	X	X	X	X	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
H	H	X	X	X	X	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H

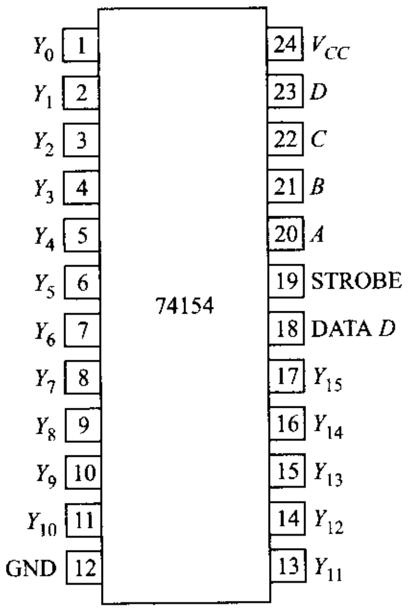


Fig. 4.11 Pinout diagram of 74154

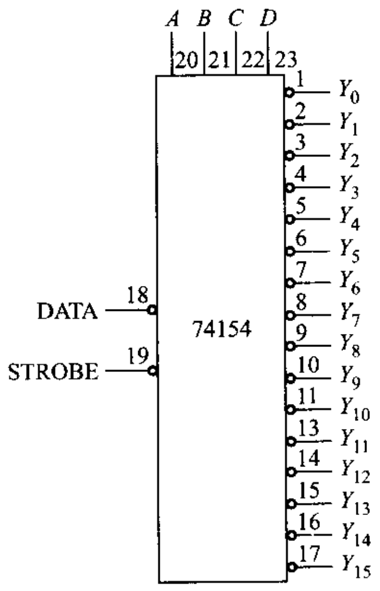


Fig. 4.12 Logic diagram of 74154

Figure 4.12 shows how to draw a 74154 on a schematic diagram. There is one input DATA bit (pin 18) under the control of nibble *ABCD*. The DATA bit is automatically steered to the output line whose subscript is the decimal equivalent of *ABCD*. Again, the bubble on the STROBE pin indicates an active-low input. Notice that DATA is inverted at the input (the bubble on pin 18) and again on any output (the bubble on each output pin). With this double inversion, DATA passes through the 74154 unchanged.

Example 4.4 In Fig. 4.13a, what does the Y_{12} output equal for each of the following conditions:

a. R is high, T is high, $ABCD = 0110$.
b. R is low, T is high, $ABCD = 1100$.
c. R is high, T is high, $ABCD = 1100$.

Solution

a. Since R and T are both high, the STROBE is low and the 74154 is active. Because $ABCD = 0110$, the input data is steered to the Y_6 output line (pin 7). The Y_{12} output remains in the high state (see Table 4.3).
b. Here, the STROBE is high and the 74154 is inactive. The Y_{12} output is high.
c. With R and T both high, the STROBE is low and the 74154 is active. Since $ABCD = 1100$, the two pulses are steered to the Y_{12} output (pin 14).

Example 4.5 Show how two 1-to-16 demultiplexers can be connected to get a 1-to-32 demultiplexer.

Solution Figure 4.13b shows the circuit diagram. A 1-to-32 demultiplexer has 5 select variables *ABCDE*. Four of them (*BCDE*) are fed to two 1-to-16 demultiplexer. And the fifth (A) is used to select one of these two multiplexer through strobe input. If $A = 0$, the top 714154 is chosen and *BCDE* directs data to one of the 15 outputs of that IC. If $A = 1$, the bottom IC is chosen and depending on value of *BCDE* data is directed to one of the 15 outputs this IC.

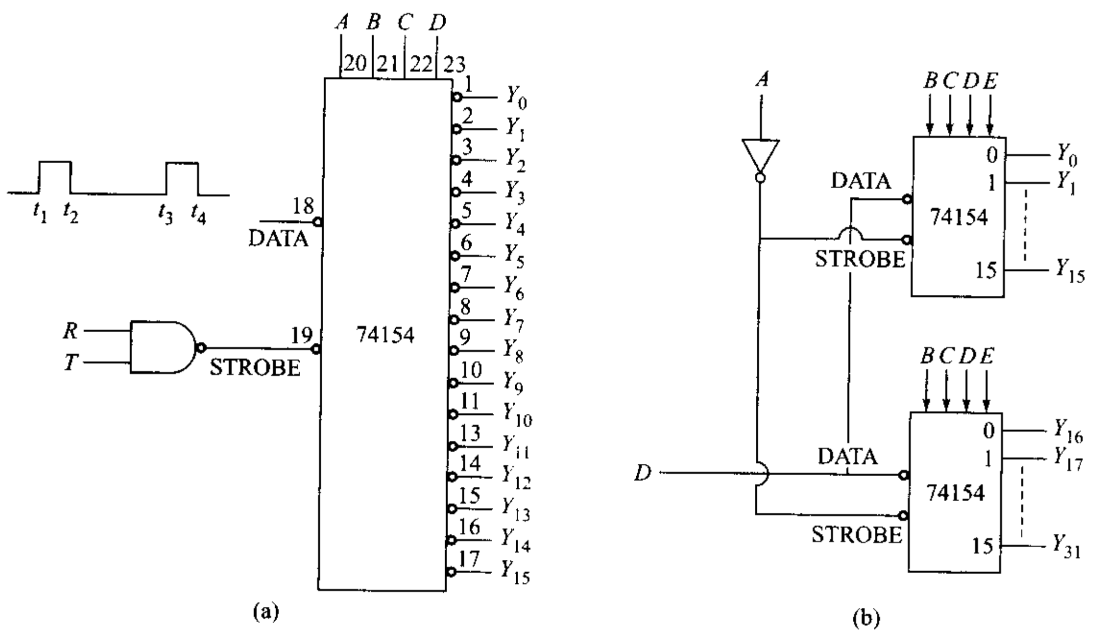

A B C D
20 21 22 23
t_1 t_2 t_3 t_4
18
DATA
R
T
19
STROBE
74154
Y_0 Y_1 Y_2 Y_3 Y_4 Y_5 Y_6 Y_7 Y_8 Y_9 Y_{10} Y_{11} Y_{12} Y_{13} Y_{14} Y_{15}
(a)
A
BCDE
DATA
STROBE
74154
0
1
15
Y_0 Y_1 Y_{15}
D
Y_{16} Y_{17} Y_{31}
(b)

Fig. 4.13

SELF-TEST

3. **A logic circuit with one input and many outputs is called a ____.**
4. **For the 74154 demultiplexer, what must the logic levels *ABCD* be in order to steer the DATA input signal to output line Y_{10}?**
5. If *ABCD* = *LHLH*, DATA = *L*, and STROBE = *H*, what will the logic level be at Y_5 on a 74154?

4.3 1-OF-16 DECODER

A *decoder* is similar to a demultiplexer, with one exception—there is no data input. The only inputs are the control bits *ABCD*, which are shown in Fig. 4.14. This logic circuit is called a *l-of-16 decoder* because only 1 of the 16 output lines is high. For instance, when *ABCD* is 0001, only the Y_1 AND gate has all, inputs high; therefore, only the Y_1 output is high. If *ABCD* changes to 0100 only the Y_4 AND gate has all inputs high; as a result, only the Y_4 output goes high.

If you check the other *ABCD* possibilities (0000 to 1111), you will find that the subscript of the high output always equals the decimal equivalent of *ABCD*. For this reason, the circuit is sometimes called a *binary-to-decimal decoder*. Because it has 4 input lines and 16 output lines, the circuit is also known as a *4-line to 16-line decoder*.

Normally, you would not build a decoder with separate inverters and AND gates as shown in Fig. 4.14. Instead, you would use an IC such as the 74154. The 74154 is called a *decoder-demultiplexer*, because it can be used either as a decoder or as a demultiplexer.

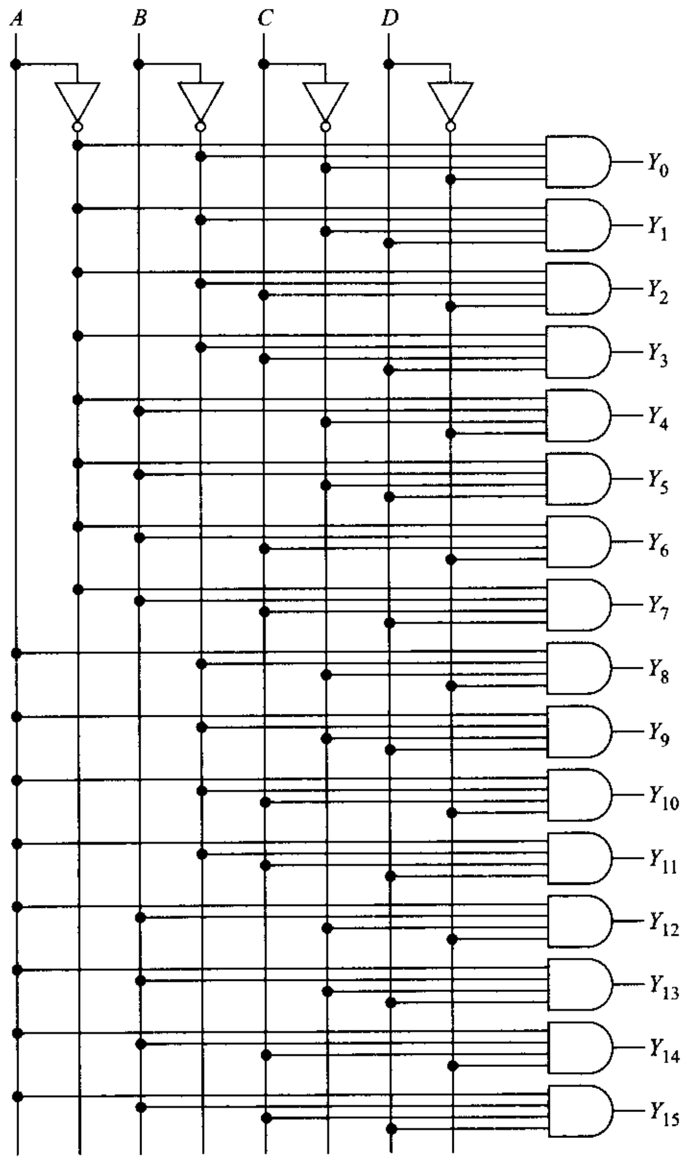


Fig. 4.14 **1-of-16 decoder**

You saw how to use a 74154 as a demultiplexer in Sec. 4.2. To use this same IC as a decoder, all you have to do is ground the DATA and STROBE inputs as shown in Fig. 4.15. Then, the selected output line is in the low state (see Table 4.3). This is why bubbles are shown on the output lines. They remind us that the output line is low when it is active or selected. For instance, if the binary input is

$$ABCD = 0111$$

then the Y_7 output is low, while all other-outputs are high.

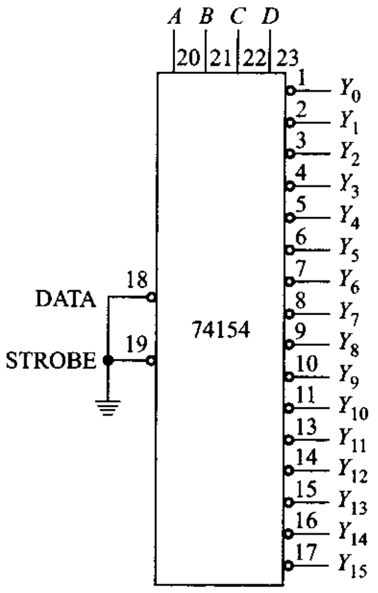


Fig. 4.15 Using 74154 as decoder

Example 4.6 Figure 4.16 illustrates *chip expansion*. We have expanded two 74154s to get a 1-of-32 decoder. Here is the way the circuit works. Bit X drives the first 74154, and the complement of X drives the second 74154. When X is low, the first 74154 is active and the second is inactive.

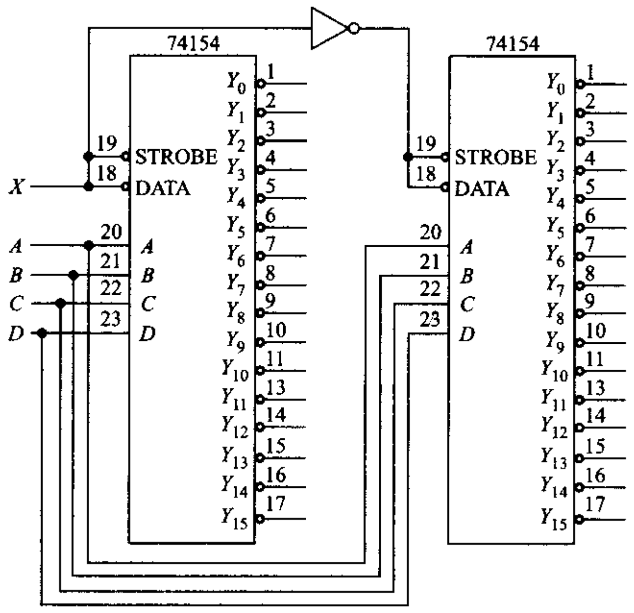


Fig. 4.16 Chip expansion

The *ABCD* input drives both decoders but only the first is active; therefore, only one output line on the first decoder is in the low state.

On the other hand, when *X* is high, the first 74154 is disabled and the second one is enabled. This means the *ABCD* input is decoded into a low output from the second decoder. In effect, the circuit of Fig. 4.16 acts like a 1-of-32 decoder.

In Fig. 4.16, all output lines are high, except the decoded output line. The bubble on each output line tells anyone looking at the schematic diagram that the active output line is in the low state rather than the high state. Similarly, the bubbles on the STROBE and DATA inputs of each 74154 indicate active-low inputs.

Example 4.7 Show how using a 3-to-8 decoder and multi-input OR gates following Boolean expressions can be realized simultaneously.

$$F_1(A, B, C) = \Sigma m(0, 4, 6);\ F_2(A, B, C) = \Sigma m(0, 5);\ F_2(A, B, C) = \Sigma m(1, 2, 3, 7)$$

Solution **Since at the decoder output we get all the minterms we use them as shown in Fig. 4.17 to get the required Boolean functions.**

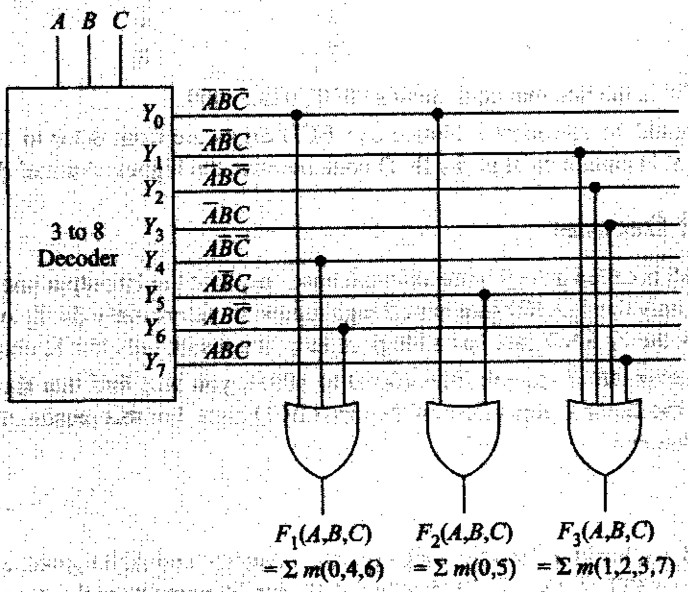


Fig. 4.17 **Solution for Example 4.7**

SELF-TEST

6. What is the significance of the bubbles on the outputs of the 74154 in Fig. 4.15?
7. In Fig. 4.15, *ABCD* = *HLHL*. What are the logic levels of the outputs?

4.4 BCD-TO-DECIMAL DECODERS

BCD is an abbreviation for *binary-coded decimal.* The BCD code expresses each digit in a decimal number by its nibble equivalent. For instance, decimal number 429 is changed to its BCD form as follows:

4	2	9
↓	↓	↓
0100	0010	1001

To anyone using the BCD code, 0100 0010 1001 is equivalent to 429.

As another example, here is how to convert the decimal number 8963 to its BCD form:

8	9	6	3
↓	↓	↓	↓
1000	1001	0110	0011

Again, we have changed each decimal digit to its binary equivalent.

Some early computers processed BCD numbers. This means that the decimal numbers were changed into BCD numbers, which the computer then added, subtracted, etc. The final answer was converted from BCD back to decimal numbers.

Here is an example of how to convert from the BCD form back to the decimal number:

0101	0111	1000
↓	↓	↓
5	7	8

As you can see, 578 is the decimal equivalent of 0101 0111 1000.

One final point should be considered. Notice that BCD digits are from 0000 to 1001. All combinations above this (1010 to 1111) cannot exist in the BCD code because the highest decimal digit being coded is 9.

BCD-to-Decimal Decoder

The circuit of Fig. 4.18 is called a *1-of-10 decoder* because only 1 of the 10 output lines is high. For instance, when *ABCD* is 0011, only the Y_3 AND gate has all high inputs; therefore, only the Y_3 output is high, If *ABCD* changes to 1000, only the Y_8 AND gate has all high inputs; as a result, only the Y_8 output goes high.

If you check the other *ABCD* possibilities (0000 to 1001), you will find that the subscript of the high output always equals the decimal equivalent of the input BCD digit. For this reason, the circuit is also called a *BCD-to-decimal converter.*

The 7445

Typically, you would not build a decoder with separate inverters and AND gates, as shown in Fig. 4.18. Instead, you would use a TTL IC like the 7445 of Fig. 4.19. Pin 16 connects to the supply voltage V_{CC} and pin 8 is grounded. Pins 12 to 15 are for the BCD input (*ABCD*), while pins 1 to 7 and 9 to 11 are for the outputs. This IC is functionally equivalent to the one in Fig. 4.18, except that the active output line is in the low state. All other output lines are in the high state, as shown in Table 4.4. Notice that an invalid BCD input (1010 to 1111) forces all output lines into the high state.

Example 4.8 The decoded outputs of a 7445 can be connected to light-emitting diodes (LEDs), as shown in Fig. 4.20. If each resistance is 1 kΩ and each LED has a forward voltage drop of 2 V, how much current is there through a LED when it is conducting? (See Chapter 13 for a discussion of LEDs.)

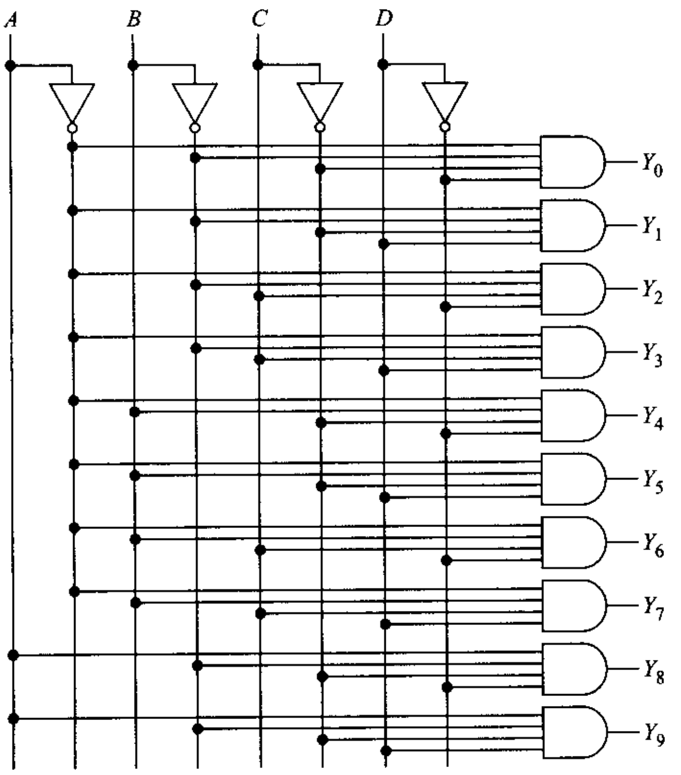


Fig. 4.18 1-of-10 decoder

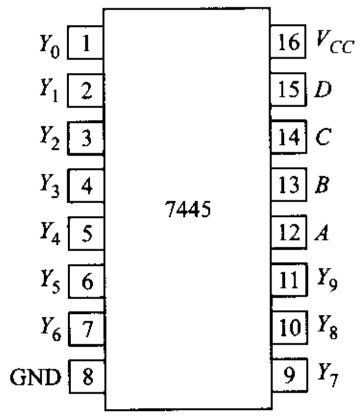


Fig. 4.19 Pinout diagram of 7445

Table 4.4 7445 Truth Table

	Inputs				*Outputs*									
No.	*A*	*B*	*C*	*D*	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7	Y_8	Y_9
0	*L*	*L*	*L*	*L*	*L*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*
1	*L*	*L*	*L*	*H*	*H*	*L*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*
2	*L*	*L*	*H*	*L*	*H*	*H*	*L*	*H*	*H*	*H*	*H*	*H*	*H*	*H*
3	*L*	*L*	*H*	*H*	*H*	*H*	*H*	*L*	*H*	*H*	*H*	*H*	*H*	*H*
4	*L*	*H*	*L*	*L*	*H*	*H*	*H*	*H*	*L*	*H*	*H*	*H*	*H*	*H*
5	*L*	*H*	*L*	*H*	*H*	*H*	*H*	*H*	*H*	*L*	*H*	*H*	*H*	*H*
6	*L*	*H*	*H*	*L*	*H*	*H*	*H*	*H*	*H*	*H*	*L*	*H*	*H*	*H*
7	*L*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*L*	*H*	*H*
8	*H*	*L*	*L*	*L*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*L*	*H*
9	*H*	*L*	*L*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*L*
	H	*L*	*H*	*L*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*
	H	*L*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*
	H	*H*	*L*	*L*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*
	H	*H*	*L*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*
	H	*H*	*H*	*L*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*
	H	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*	*H*

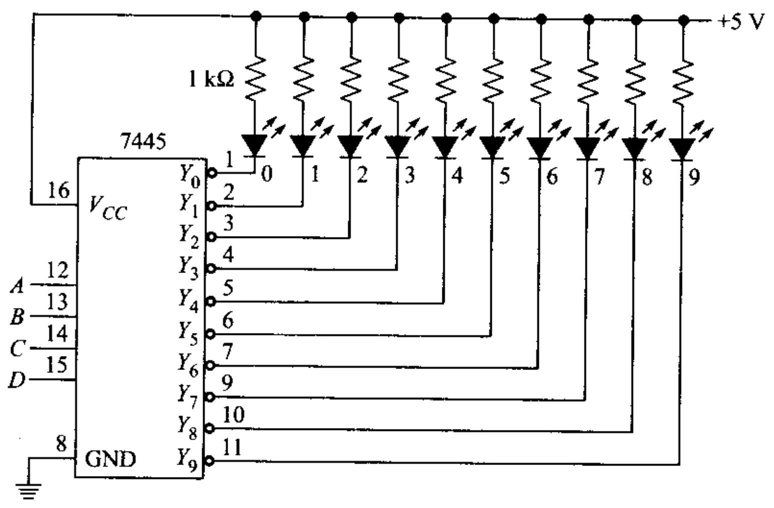


Fig. 4.20 Circuit for Example 4.7

Solution When an output is in the low state, you can approximate the output voltage as zero. Therefore, the current through a LED is

$$I = \frac{5\text{ V} - 2\text{ V}}{1\text{ k}\Omega} = 3\text{ mA}$$

Example 4.9 The LEDs of Fig. 4.20 are numbered 0 through 9. Which of the LEDs is lit for each of the following conditions:

a. $ABCD = 0101$.
b. $ABCD = 1001$.
c. $ABCD = 1100$.

Solution

a. When $ABCD = 0101$, the decoded output line is Y_5. Since Y_5 is approximately grounded, LED 5 lights up. All other LEDs remain off because the other outputs are high.
b. When $ABCD = 1001$, LED 9 is on.
c. $ABCD = 1100$ is an invalid input. Therefore, none of the LEDs is on because all output, lines are high (see Table 4.4).

SELF-TEST

8. What does the abbreviation BCD stand for?
9. What is a LED?

4.5 SEVEN-SEGMENT DECODERS

A LED emits radiation when forward-biased. Why? Because free electrons recombine with holes near the junction. As the free electrons fall from a higher energy level to a lower one, they give up energy in the form of heat and light. By using elements like gallium, arsenic, and phosphorus, a manufacturer can produce LEDs that emit red, green, yellow, blue, orange and infrared (invisible) light. LEDs that produce visible radiation are useful in test instruments, pocket calculators, etc.

Seven-Segment Indicator

Figure 4.21a shows a *seven-segment indicator*, i.e. seven LEDs labeled *a* through *g*. By forward-biasing different LEDs, we can display the digits 0 through 9 (see Fig. 4.21b). For instance, to display a 0, we need to light up segments *a*, *b*, *c*, *d*, *e*, and *f*. To light up a 5, we need segments *a*, *c*, *d*, *f*, and *g*.

Seven-segment indicators may be the common-anode type where all anodes are connected together (Fig. 4.22a) or the common-cathode type where all cathodes are connected together (Fig. 4.22b). With the common-anode type of Fig. 4.22a, you have to connect a current-limiting resistor between each LED and ground. The size of this resistor determines how much current flows through the LED. The typical LED current is between 1 and 50 mA. The common-cathode type of Fig. 4.22b uses a current-limiting resistor between each LED and $+V_{CC}$.

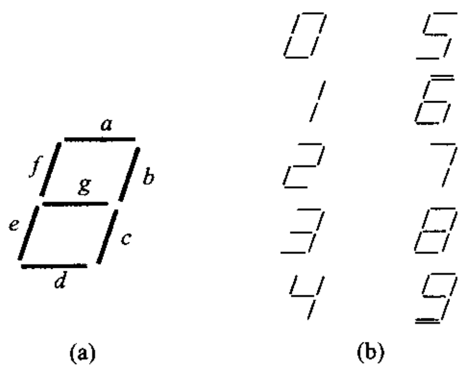


Fig. 4.21 Seven-segment indicator

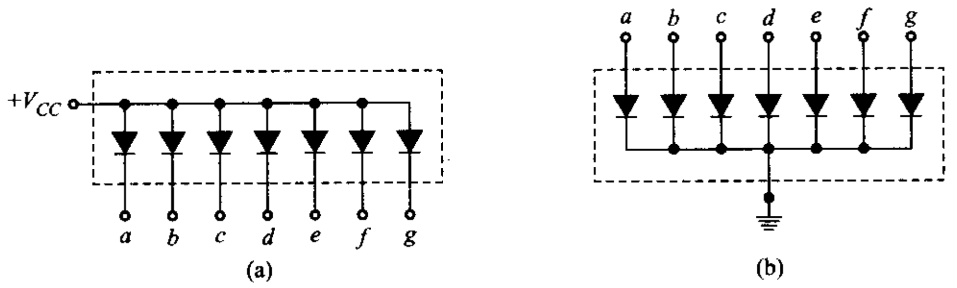


Fig. 4.22 (a) Common-anode type, (b) Common-cathode type

The 7446

A seven-segment *decoder-driver* is an IC decoder that can be used to drive a seven-segment indicator. There are two types of decoder-drivers, corresponding to the common-anode and common-cathode indicators. Each decoder-driver has 4 input pins (the BCD input) and 7 output pins (the *a* through *g* segments).

Figure 4.23a shows a 7446 driving a common-anode indicator. Logic circuits inside the 7446 convert the BCD input to the required output. For instance, if the BCD input is 0111, the internal logic (not shown) of the 7446 will force LEDs *a*, *b*, and *c* to conduct. As a result, digit 7 will appear on the seven-segment indicator.

Notice the current-limiting resistors between the seven-segment indicator and the 7446 of Fig. 4.23a. You have to connect these external resistors to limit the current in each segment to a safe value between 1 and 50 mA, depending on how bright you want the display to be.

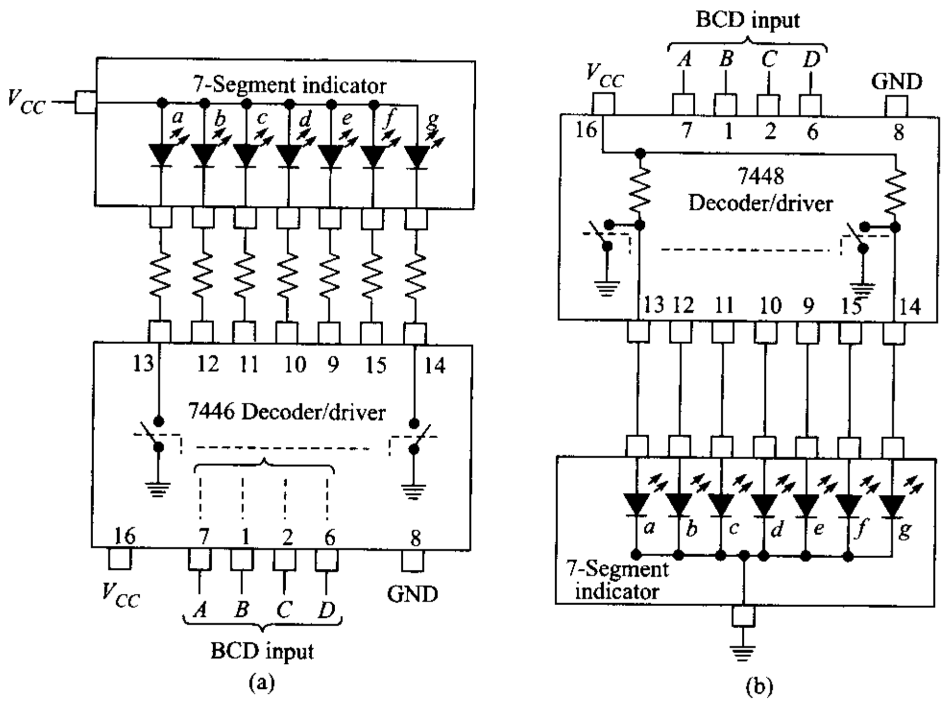


Fig. 4.23 (a) 7446 decoder-driver, (b) 7448 decoder-driver

The 7448

Figure 4.23b is the alternative decoding approach. Here, a 7448 drives a common-cathode indicator. Again, internal logic converts the BCD input to the required output. For example, when a BCD input of 0100 is used, the internal logic forces LEDs *b*, *c*, *f*, and *g* to conduct. The seven-segment indicator then displays a 4. Unlike the 7446 that requires external current-limiting resistors, the 7448 has its own current-limiting resistors on the chip. A switch symbol is used to illustrate operation of the 7446 and 7448 in Fig. 4.23. Switching in the actual IC is of course accomplished using bipolar junction transistors (BJTs).

SELF-TEST

10. Sketch the segments in a seven-segment indicator.
11. Each segment of a seven-segment indicator is what type of device?

4.6 ENCODERS

An encoder converts an active input signal into a coded output signal. Figure 4.24 illustrates the general idea. There are *n* input lines, only one of which is active. Internal logic within the encoder converts this active input to a coded binary output with *m* bits.

Decimal-to-BCD Encoder

Figure 4.25 shows a common type of encoder—*the decimal-to-BCD encoder*. The switches are push-button switches like those of a pocket calculator. When button 3 is pressed, the C and D OR gates have high inputs; therefore, the output is

$$ABCD = 0011$$

If button 5 is pressed, the output becomes

$$ABCD = 0101$$

When switch 9 is pressed,

$$ABCD = 1001$$

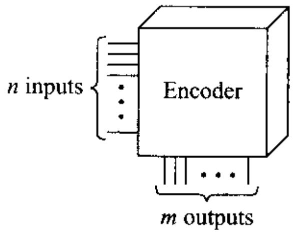


Fig. 4.24 Encoder

The 74147

Figure 4.26a is the pinout diagram for a 74147, a decimal-to-BCD encoder. The decimal input, X_1 to X_9, connect to pins 1 to 5, and 10 to 13. The BCD output comes from pins 14, 6, 7, and 9. Pin 16 is for the supply voltage, and pin 8 is grounded. The label NC on pin 15 means *no connection* (the pin is not used).

Figure 4.26b shows how to draw a 74147 on a schematic diagram. As usual, the bubbles indicate active-low inputs and outputs. Table 4.5 is the truth table of a 74147. Notice the following. When all X inputs are high, all outputs are high. When X_9 is low, the $ABCD$ output is $LHHL$ (equivalent to 9 if you complement the bits). When X_8 is the only low input, $ABCD$ is $LHHH$

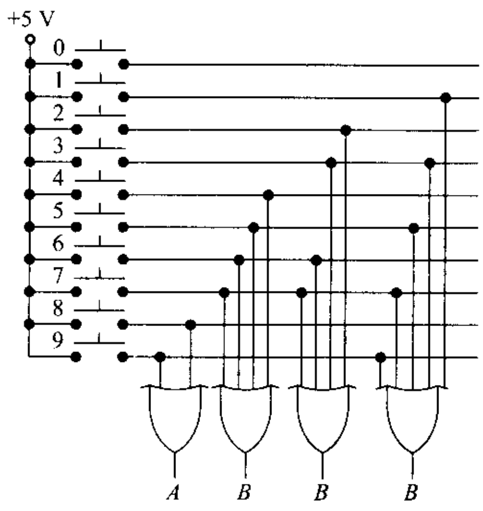


Fig. 4.25 Decimal-to-BCD encoder

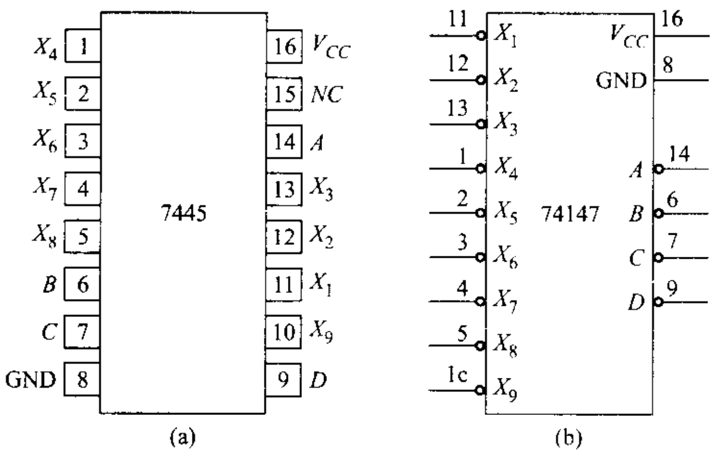


Fig. 4.26 (a) Pinout diagram of 74147, (b) Logic diagram

Table 4.5 **74147 Truth Table**

Inputs									*Outputs*			
X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	A	B	C	D
H	H	H	H	H	H	H	H	H	H	H	H	H
X	X	X	X	X	X	X	X	L	L	H	H	L
X	X	X	X	X	X	X	L	H	L	H	H	H
X	X	X	X	X	X	L	H	H	H	L	L	L
X	X	X	X	X	L	H	H	H	H	L	L	H
X	X	X	X	L	H	H	H	H	H	L	H	L
X	X	X	L	H	H	H	H	H	H	L	H	H
X	X	L	H	H	H	H	H	H	H	H	L	L
X	L	H	H	H	H	H	H	H	H	H	L	H
L	H	H	H	H	H	H	H	H	H	H	H	L

(equivalent to 8 if the bits are complemented). When X_7 is the only low input, $ABCD$ becomes $HLLL$ (equivalent to 7 if the bits are complemented). Continue like this through the rest of the truth table and you can see that an active-low decimal input is being converted to a complemented BCD output.

Incidentally, the 74147 is called a *priority encoder* because it gives priority to the highest-order input. You can see this by looking at Table 4.5. If all inputs X_1 through X_9 are low, the highest of these, X_9, is encoded to get an output of $LHHL$. In other words, X_9 has priority over all others. When X_9 is high, X_8 is next in line of priority and gets encoded if it is low. Working your way through Table 4.5, you can see that the highest active-low from X_9 to X_0 has priority and will control the encoding.

Example 4.10 What is the $ABCD$ output of Fig. 4.27 when button 6 is pressed?

Solution When all switches are open, the X_1 to X_9 inputs are pulled up to the high state (+5 V). A glance at Table 4.5 indicates that the $ABCD$ output is $HHHH$ at this time.

When switch 6 is pressed, the X_6 input is grounded. Therefore, all X inputs are high, except for X_6. Table 4.5 indicates that the $ABCD$ output is $HLLH$, which is equivalent to 6 when the output bits are complemented.

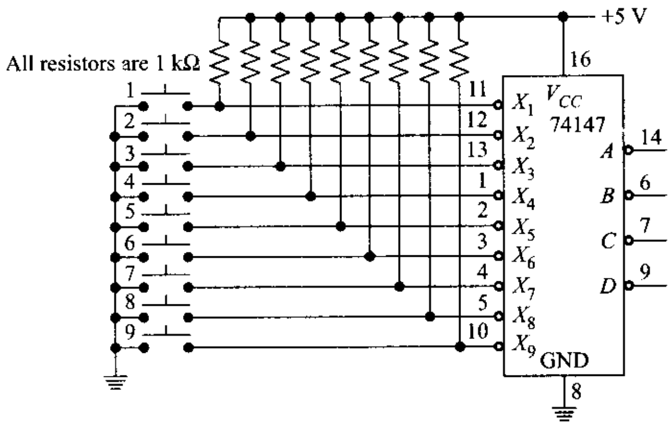


Fig. 4.27

Example 4.11 Design a priority encoder the truth table of which is shown in Fig. 4.28a. The order of priority for three inputs is $X_1 > X_2 > X_3$. However, if the encoder is not enabled by S or all the inputs are inactive the output $AB = 00$.

Solution Figure 4.28b and Fig. 4.28c show the Karnaugh map for output A and B respectively. Note that, we have used a different notation for input variables in these maps. Compare this with notations presented in previous chapters. You will find a variable with prime is presented by 0 and if it is not primed is represented by 1. Then taking groups of 1s we get the design equations as shown in the figure. The logic circuits for output A and B can be directly drawn from these equations.

Intput				Output	
S	X_1	X_2	X_3	A	B
0	×	×	×	0	0
1	1	×	×	0	1
1	0	1	×	1	0
1	0	0	1	1	1
1	0	0	0	0	0

(a)

X_2X_3 \ SX_1	00	01	11	10
00	0	0	0	0
01	0	0	0	1
11	0	0	0	1
10	0	0	0	1

$A = S\bar{X}_1X_3 + S\bar{X}_1X_2$

(b)

X_2X_3 \ SX_1	00	01	11	10
00	0	0	1	0
01	0	0	1	1
11	0	0	1	0
10	0	0	1	0

$B = SX_1 + S\bar{X}_2X_3$

(c)

Fig. 4.28 **Design of a priority encoder**

SELF-TEST

12. What is an encoder?
13. What type of encoder is the TTL 74147?

4.7 EXCLUSIVE-OR GATES

The *exclusive-OR gate* has a high output only when an odd number of inputs is high. Figure 4.29 shows how to build an exclusive-OR gate. The upper AND gate forms the product $\bar{A}B$, while the lower one produces $A\bar{B}$. Therefore, the output of the OR gate is

$$Y = \bar{A}B + A\bar{B}$$

Here is what happens for different inputs. When A and B are low, both AND gates have low outputs; therefore, the final output Y is low. If A is low and B is high, the upper AND gate has a high output, so the OR gate has high output. Likewise, a high A and a low B result in a final output that is high. If both inputs are high, both AND gates have low outputs and the final output is low.

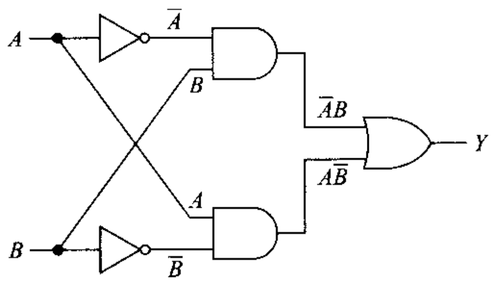


Fig. 4.29 **Exclusive-OR gate**

Table 4.6 shows the truth table for a 2-input exclusive-OR gate. The output is high when A or B is high, but not when both are high. This is why the circuit is known as an exclusive-OR gate. In other words, the output is a 1 *only* when the inputs are different.

Table 4.6 **Exclusive-OR Truth Table**

A	*B*	*Y*
0	0	0
0	1	1
1	0	1
1	1	0

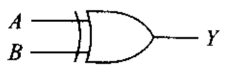


Fig. 4.30 **Logic symbol for exclusive-OR gate**

Figure 4.30 shows the symbol for a 2-input exclusive-OR gate. Whenever you see this symbol, remember the action—the output is high if either input is high, but not when both are high. Stated another way, the inputs must be different to get a high output.

Four Inputs

Figure 4.31a shows a pair of exclusive-OR gates driving an exclusive-OR gate. If all inputs (*A* to *D*) are low, the input gates have low outputs, so the final gate has a low output. If *A* to *C* are low and *D* is high, the upper gate has a low output, the lower gate has a high output, and the output gate has a high output.

If we continue analyzing the circuit operation for the remaining input possibilities, we can work out Table 4.7. Here is an important property of this truth table. Each *ABCD* input with an odd number of 1s produces an output 1. For instance, the first *ABCD* entry to produce an output 1 is 0001; it has an odd

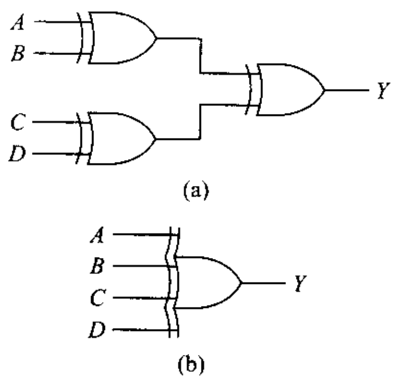


Fig. 4.31 **Four-input exclusive OR gate**

Table 4.7 **4-Input Exclusive-OR Gate**

Comment	*A*	*B*	*C*	*D*	*Y*
Even	0	0	0	0	0
Odd	0	0	0	1	1
Odd	0	0	1	0	1
Even	0	0	1	1	0
Odd	0	1	0	0	1
Even	0	1	0	1	0
Even	0	1	1	0	0
Odd	0	1	1	1	1
Odd	1	0	0	0	1
Even	1	0	0	1	0
Even	1	0	1	0	0
Odd	1	0	1	1	1
Even	1	1	0	0	0
Odd	1	1	0	1	1
Odd	1	1	1	0	1
Even	1	1	1	1	0

number of 1s. The next *ABCD* entry to produce an output 1 is 0010; again, an odd number of 1s. An output 1 also occurs for these *ABCD* inputs: 0100, 0111, 1000, 1011,1101, and 1110, each having an odd number of 1s.

Figure 4.31a illustrates the logic for a 4-input exclusive-OR gate. In this book, we will use the abbreviated symbol given in Fig. 4.31b to represent a 4-input exclusive-OR gate. When you see this symbol, remember the action—the gate produces an output 1 when the *ABCD* input has an odd number of 1s.

Any Number of Inputs

Using 2-input exclusive-OR gates as building blocks, you can produce exclusive-OR gates with any number of inputs. For example, Fig. 4.32a shows a pair of exclusive-OR gates. There are 3 inputs and 1 output. If you analyze this circuit, you will find it produces an output 1 only when the 3-bit input has an odd number of 1s. Figure 4.32b shows an abbreviated symbol for a 3-input exclusive-OR gate.

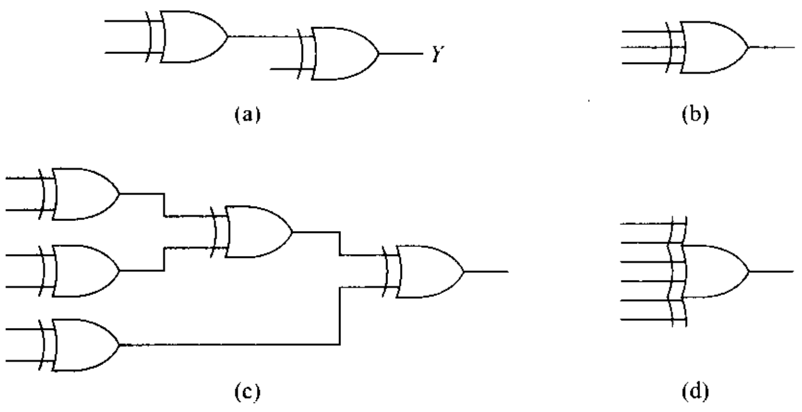


Fig. 4.32 Exclusive-OR gate with several inputs

As another example, Fig. 4.32c shows a circuit with 6 inputs and 1 output. Analysis of the circuit shows that it produces an output 1 only when the 6-bit input has an odd number of 1s. Figure 4.32d shows an abbreviated symbol for a 6-input exclusive-OR gate.

In general, you can build an exclusive-OR gate with any number of inputs. Such a gate always produces an output 1 only when the *n*-bit input has an odd number of 1s.

SELF-TEST

14. When is the output of an exclusive-OR gate high?
15. Draw the logic symbol for an exclusive-OR gate.

4.8 PARITY GENERATORS AND CHECKERS

Even parity means an *n*-bit input has an even number of 1s. For instance, 110011 has even parity because it contains four 1s. *Odd parity* means an *n*-bit input has an odd number of 1s. For example, 110001 has odd parity because it contains three 1s.

Here are two more examples:

1111 0000 1111 0011 even parity
1111 0000 1111 0111 odd parity

The first binary number has even parity because it contains ten 1s; the second binary number has odd parity because it contains eleven 1s. Incidentally, longer binary numbers are much easier to read if they are split into nibbles, or groups of four, as done here.

Parity Checker

Exclusive-OR gates are ideal for checking the parity of a binary number because they produce an output 1 when the input has an odd number of 1s. Therefore, an even-parity input to an exclusive-OR gate produces a low output, while an odd-parity input produces a high output.

For instance, Fig. 4.33 shows a 16-input exclusive-OR gate. A 16-bit number drives the input. The exclusive-OR gate produces an output 1 because the input has odd parity (an odd number of 1s). If the 16-bit input changes to another value, the output becomes 0 for even-parity numbers and 1 for odd-parity numbers.

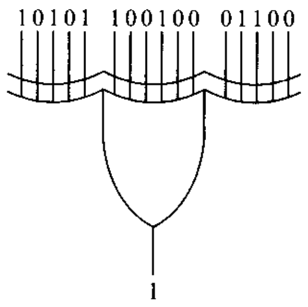


Fig. 4.33 Exclusive-OR gate with 16 inputs

Parity Generation

In a computer, a binary number may represent an instruction that tells the computer to add, subtract, and so on; or the binary number may represent data to be processed like a number, letter, etc. In either case, you sometimes will see an extra bit added to the original binary number to produce a new binary number with even or odd parity.

For instance, Fig. 4.34 shows this 8-bit binary number:

$$X_7X_6X_5X_4 \quad X_3X_2X_1X_0$$

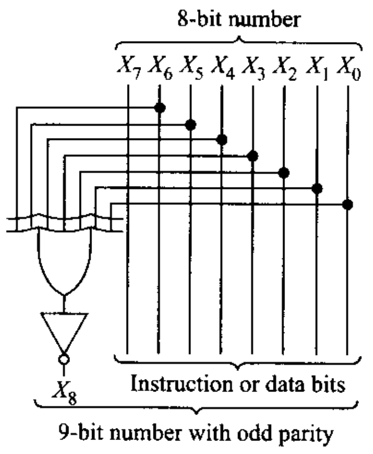
Suppose this number equals 0100 0001. Then, the number has even parity, which means the exclusive-OR gate produces an output of 0. Because of the inverter,

$$X_8 = 1$$

and the final 9-bit output is 1 0100 0001. Notice that this has odd parity.

Suppose we change the 8-bit input to 0110 0001. Now, it has odd parity. In this case, the exclusive-OR gate produces an output 1. But the inverter produces a 0, so that the final 9-bit output is 0 0110 0001. Again, the final output has odd parity.

The circuit given in Fig. 4.34 is called an *odd-parity generator* because it always produces a 9-bit output number with odd parity. If the 8-bit input has even parity, a 1 comes



8-bit number
X_7 X_6 X_5 X_4 X_3 X_2 X_1 X_0
X_8
Instruction or data bits
9-bit number with odd parity

Fig. 4.34 Odd-parity generation

out of the inverter to produce a final output with odd parity. On the other hand, if the 8-bit input has odd parity, a 0 comes out of the inverter, and the final 9-bit output again has odd parity. (To get an even-parity generator, delete the inverter.)

Application

What is the practical application of parity generation and checking? Because of transients, noise, and other disturbances, 1-bit errors sometimes occur when binary data is transmitted over telephon· lines or other communication paths. One way to check for errors is to use an odd-parity generator at the .ı ansmitting end and an odd-parity checker at the receiving end. If no 1-bit errors occur in transmission, the received data will have odd parity. But if one of the transmitted bits is changed by noise or any other disturbance, the received data will have even parity.

For instance, suppose we want to send 0100 0011. With an odd-parity generator like Fig. 4.34, the data to be transmitted will be 0 0100 0011. This data can be sent over telephone lines to some destination. If no errors occur in transmission, the odd-parity checker at the receiving end will produce a high output, meaning the received number has odd parity. On the other hand, if a 1-bit error does creep into the transmitted data, the odd-parity checker will have a low output, indicating the received data is invalid.

One final point should be made. Errors are rare to begin with. When they do occur, they are usually 1-bit errors. This is why the method described here catches almost all of the errors that occur in transmitted data.

The 74180

Figure 4.35 shows the pinout diagram for a 74180, which is a TTL parity generator-checker. The input data bits are X_7 to X_0; these bits may have even or odd parity. The even input (pin 3) and the odd input (pin 4) control the operation of the chip as shown in Table 4.8. The symbol Σ stands for *summation*. In the left input column of Table 4.8, Σ of *H*'s (highs) refers to the parity of the input data X_7 to X_0. Depending on how you set up the values of the even and odd inputs, the Σ even and Σ odd outputs may be low or high.

For instance, suppose even input is high and odd input is low. When the input data has even parity (the first entry of Table 4.8), the Σ even output is high and the Σ odd output is low. When the input data has odd parity, the Σ even output is low and the Σ odd output is high.

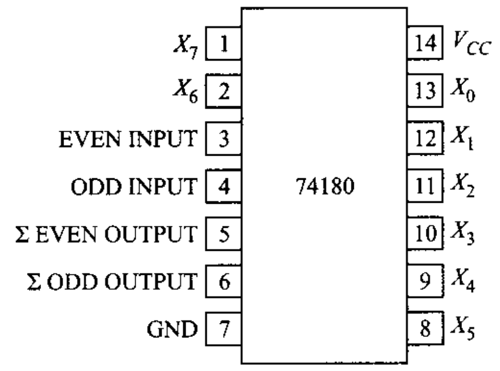


Fig. 4.35 Pinput diagram of 74180

Table 4.8 74180 Truth Table

Inputs			Outputs	
Σ of H's at X_7 to X_0	*Even*	*Odd*	*Σ even*	*Σ odd*
Even	*H*	*L*	*H*	*L*
Odd	*H*	*L*	*L*	*H*
Even	*L*	*H*	*L*	*H*
Odd	*L*	*H*	*H*	*L*
X	*H*	*H*	*L*	*L*
X	*L*	*L*	*H*	*H*

If you change the control inputs, you change the operation. Assume that the even input is low and the odd input is high. When the input data has even parity, the Σ even output is low and the Σ odd output is high. When the input data has odd parity, the Σ even output is high and the Σ odd output is low.

The 74180 can be used to detect even or odd parity. It can also be set up to generate even or odd parity.

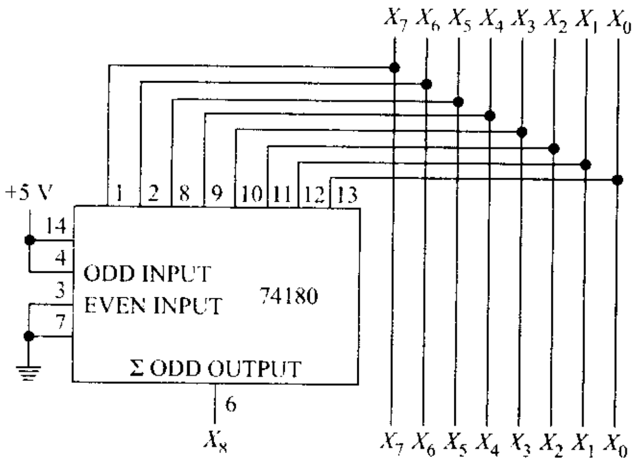

X_7 X_6 X_5 X_4 X_3 X_2 X_1 X_0
+5 V
1
2
8
9
10
11
12
13
14
4
3
7
ODD INPUT
EVEN INPUT
74180
Σ ODD OUTPUT
6
X_8
X_7 X_6 X_5 X_4 X_3 X_2 X_1 X_0

Fig. 4.36 **Using a 74180 to generate odd parity**

Example 4.12 Show how to connect a 74180 to generate a 9-bit output with odd parity.

Solution Figure 4.36 shows one solution. The ODD INPUT (pin 4) is connected to +5 V, and the EVEN INPUT (pin 3) is grounded. Suppose the input data $X_7 \dots X_0$ has even parity. Then, the third entry of Table 4.8 tells us the Σ ODD OUTPUT (pin 6) is high. Therefore, the 9-bit number $X_8 \dots X_0$ coming out of the circuit has odd parity.

On the other hand, suppose $X_7 \dots X_0$ has odd parity. Then the fourth entry of Table 4.8 says that the Σ odd output is low. Again, the 9-bit number $X_8 \dots X_0$ coming out at the bottom of Fig. 4.36 has odd parity.

The following conclusion may be drawn. Whether the input data has even or odd parity, the 9-bit number being generated in Fig. 4.36 always has odd parity.

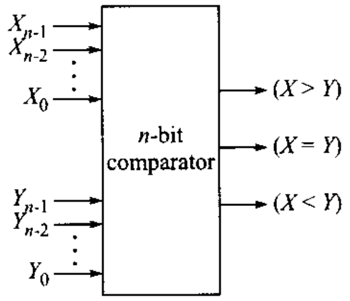
SELF-TEST

16. What does it mean to say that an n-bit binary number has *even* parity?
17. Exclusive-OR gates are useful as parity generators. (T or F)

4.9 MAGNITUDE COMPARATOR

Magnitude comparator compares magnitude two n-bit binary numbers, say X and Y and activates one of these three outputs $X = Y$, $X > Y$ and $X < Y$. Figure 4.37a presents block diagram of such a comparator. Fig. 4.37b presents truth table when two 1-bit numbers are compared and its circuit diagram is shown in Fig. 4.37c. The logic equations for the outputs can be written as follows, where G, L, E stand for greater than, less than and equal to respectively.

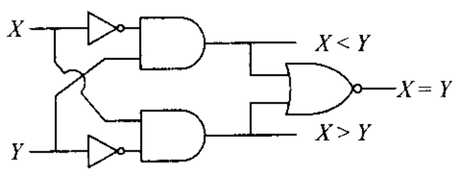
$$(X > Y): G = XY' \quad (X < Y): L = X'Y \quad (X = Y): E = X'Y' + XY = (XY' + X'Y)' = (G + L)'$$



(a)

Input		Output		
X	Y	$X>Y$	$X=Y$	$X<Y$
0	0	0	0	1
0	1	0	1	0
1	0	1	0	0
1	1	0	0	1

(b)



(c)

Fig. 4.37 **(a) Block diagram of Magnitude comparator, (b) Truth table, (c) Circuit for 1-bit comparator**

Now, how can we design a 2-bit comparator? We can form a 4-variable (X: X_1X_0 and Y: Y_1Y_0) truth table and get logic equations through any simplification technique. But this procedure will become very complex if we try to design a comparator for 3-bit numbers or more. Here, we discuss a simple but generic procedure for 2-bit comparator design, which can easily be extended to make any n-bit magnitude comparator. We shall use the truth table of 1-bit comparator that generates greater than, less than and equal terms.

Let's first define bit-wise greater than terms (G):	$G_1 = X_1Y_1'$,	$G_0 = X_0Y_0'$
Then, bit-wise less than term (L):	$L_1 = X_1'Y_1$,	$L_0 = X_0'Y_0$
Therefore, bit-wise equality term (E):	$E_1 = (G_1 + L_1)'$,	$E_0 = (G_0 + L_0)'$

From above definitions we can easily write 2-bit comparator outputs as follows.

$$(X = Y) = E_1.E_0 \qquad (X > Y) = G_1 + E_1.G_0 \qquad (X < Y) = L_1 + E_1.L_0$$

The logic followed in arriving at these equations is this; $X = Y$ when both the bits are equal. $X > Y$ if MSB of X is higher ($G_1 = 1$) than that of Y. If MSB is equal, given by $E_1 = 1$, then LSB of X and Y is checked and if found higher ($G_0 = 1$) the condition $X > Y$ is fulfilled. Similar logic gives us the $X < Y$ term.

Thus for any two n-bit numbers X: $X_{n-1}\, X_{n-2} \ldots X_0$ and Y: $Y_{n-1}\, Y_{n-2} \ldots Y_0$

We can write,
$$(X = Y) = E_{n-1}\, E_{n-2} \ldots E_0$$
$$(X > Y) = G_{n-1} + E_{n-1}G_{n-2} + \ldots + E_{n-1}\, E_{n-2} \ldots E_1G_0$$
$$(X < Y) = L_{n-1} + E_{n-1}L_{n-2} + \ldots + E_{n-1}\, E_{n-2} \ldots E_1L_0$$

where E_i, G_i and L_i represent for ith bit $X_i = Y_i$, $X_i > Y_i$ and $X_i < Y_i$ terms respectively.

The block diagram of IC 7485, which compares two 4-bit numbers is shown in Fig. 4.38a. This is a 16 pin IC and all the pin numbers are mentioned in this functional diagram. Note that the circuit has three additional inputs in the form of $(X = Y)_{in}$, $(X > Y)_{in}$ and $(X < Y)_{in}$. What is the use of them? They are used when we need to connect more than one IC 7485 to compare numbers having more than 4-bits. But these inputs are of lower priority. They can decide the output only when 4-bit numbers fed to this IC are equal. For example, if $X = 0100$ and $Y = 0011$, $(X > Y)_{out}$ will be high and other outputs will be low irrespective of the value appearing at $(X = Y)_{in}$, $(X > Y)_{in}$ and $(X < Y)_{in}$. When IC 7485 is not used in cascade we keep $(X = Y)_{in} = 1$, $(X > Y)_{in} = 0$ and $(X < Y)_{in} = 0$.

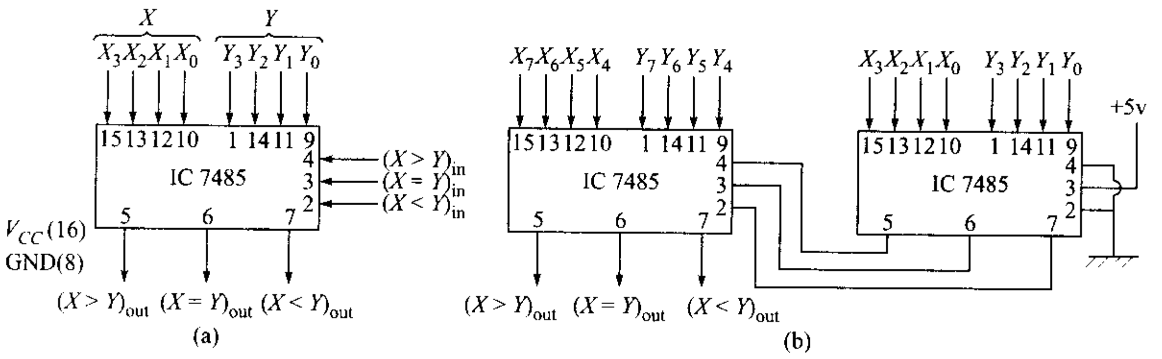


Fig. 4.38 **(a) Functional diagram of IC 7485, (b) 8-bit comparator from two 4-bit comparators**

Example 4.13 Show how two IC 7485 can be used to compare magnitude of two 8-bit numbers.

Solution Refer to Fig. 4.38b for solution. The numbers to compare are X: $X_7 X_6 \ldots X_0$ and Y: $Y_7 Y_6 \ldots Y_0$. We need two IC 7485s each one comparing 4 bits. The most significant bits (suffix 7,6,5,4) are given higher priority and the final output is taken from that IC 7485 which compares them.

SELF-TEST

18. How many outputs a magnitude comparator generates?
19. How many IC 7485s are needed to compare two 12-bit numbers?

4.10 READ-ONLY MEMORY

A *read-only memory* (which is abbreviated ROM and rhymes with Mom) is an IC that can store thousands of binary numbers representing computer instructions and other fixed data. A good example of fixed data is the unchanging information in a mathematical table. Since the numerical data do not change, they can be stored in a ROM, included in a computer system, and used as a "look-up" table when needed. Some of the smaller ROMs are also used to implement truth tables. In other words, we can use a ROM instead of sum-of-products circuit to generate any Boolean function.

Diode ROM

Suppose we want to build a circuit that stores the binary numbers shown in Table 4.9. To keep track of where the numbers are stored, we will assign *addresses*. For instance, we want to store 0111 at address 0, 1000 at address 1, 1011 at address 2, and so forth. Figure 4.39 shows one way to store the nibbles given in Table 4.9. When the switch is in position 0 (address 0), the upper row of diodes are conducting current (they act as closed

Table 4.9 **Diode ROM**

Address	*Nibble*
0	0111
1	1000
2	1011
3	1100
4	0110
5	1001
6	0011
7	1110

switches). (See Chapter 14 for a discussion of diodes.) The output of the ROM is thus

$$Y_3 Y_2 Y_1 Y_0 = 0111$$

When the switch is moved to position 1, the second row is activated and

$$Y_3 Y_2 Y_1 Y_0 = 1000$$

As you move the switch to the remaining positions or addresses, you get a $Y_3 \ldots Y_0$ output that matches the nibbles given in Table 4.9.

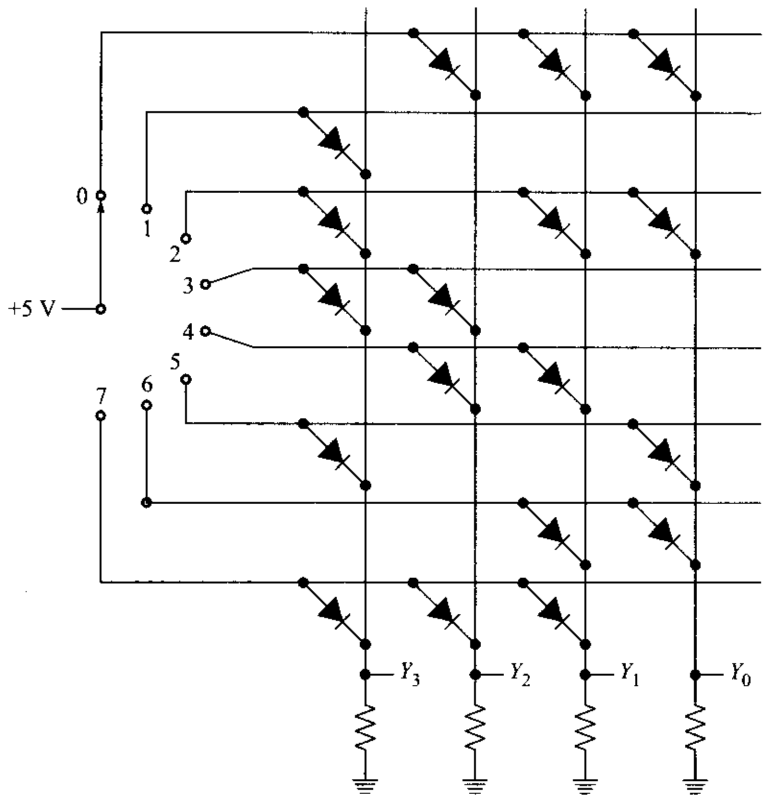


Fig. 4.39 Diode ROM

On-Chip Decoding

Rather than switch-select the addresses as shown in Fig. 4.39, a manufacturer uses *on-chip decoding*. Figure 4.40 illustrates the idea. The 3-input pins (A, B, and C) supply the binary address of the stored number. Then, a l-of-8 decoder produces a high output to one of the diode rows. For instance, if

$$ABC = 100$$

the l-of-8 decoder applies a high voltage to the $A\bar{B}\bar{C}$ line, and the ROM output is

$$Y_3 Y_2 Y_1 Y_0 = 0110$$

If you change the binary address to

$$ABC = 110$$

the ROM output changes to

$$Y_3Y_2Y_1Y_0 = 0011$$

With on-chip decoding, n inputs can select 2^n memory locations (stored numbers). For instance, we need 3 address lines to access 8 memory locations, 4 address lines for 16 memory locations, 8 address lines for 256 memory locations, and so on.

Commercially Available ROMs

A binary number is sometimes called a *word*. In a computer, binary numbers or words represent instructions, alphabet letters, decimal numbers, etc. The circuit given in Fig. 4.40 is a 32-bit ROM organized as 8 words

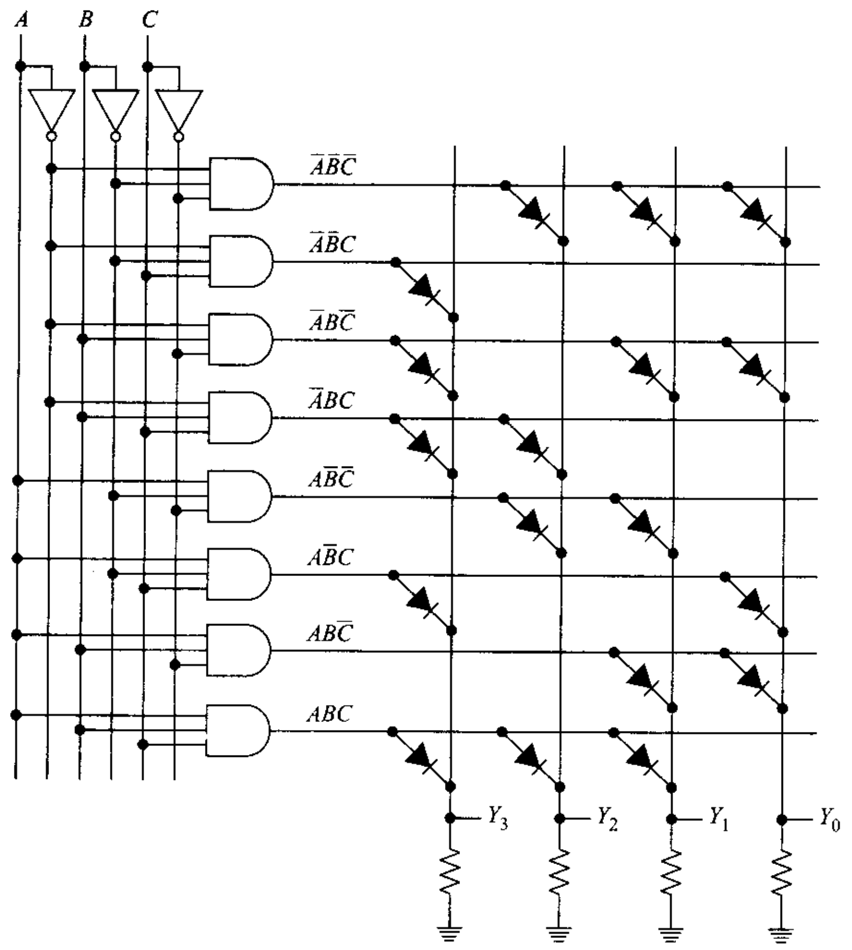


Fig. 4.40 On-chip decoding

with 4 bits at each address (an 8 × 4 ROM). The ROM given in Fig. 4.40 is for instructional purposes only because you would not build this circuit with discrete components. Instead, you would select a commercially available ROM. For instance, here are some TTL ROMs:

7488: 256 bits organized as 32 × 8

74187: 1024 bits organized as 256 × 4

74S370: 2048 bits organized as 512 × 4

As you can see, the 7488 can store 32 words of 8 bits each, the 74187 can store 256 words of 4 bits each, and the 74S370 can store 512 words of 4 bits each. If you want to store bytes (words with 8 bits), then you can parallel the 4-bit ROMs. For example, two parallel 74187s can store 256 words of 8 bits each.

One way to change the stored numbers of a ROM is by adding or removing diodes. With discrete circuits, you would have to solder or unsolder diodes to change the stored nibbles. With integrated circuits, however, you can send a list of the data to be stored to an IC manufacturer, who then produces a *mask* (a photographic template of the circuit). This mask is used in the mass production of your ROMs. As a rule, ROMs are used only for large production runs (thousands or more) because of manufacturing costs.

Generating Boolean Functions

Because the AND gates of Fig. 4.40 produce all the fundamental products and the diodes OR some of these products, the ROM is generating four Boolean functions as follows:

$$Y_3 = \overline{A}\,\overline{B}C + \overline{A}B\overline{C} + \overline{A}BC + A\overline{B}C + ABC \tag{4.1}$$

$$Y_2 = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}BC + A\overline{B}\,\overline{C} + ABC \tag{4.2}$$

$$Y_1 = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}B\overline{C} + A\overline{B}\,\overline{C} + AB\overline{C} + ABC \tag{4.3}$$

$$Y_0 = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}B\overline{C} + A\overline{B}C + AB\overline{C} \tag{4.4}$$

This means that you can use a ROM instead of a logic circuit to implement a truth table.

For instance, suppose you start with a truth table like the one in Table 4.10. There are four outputs: Y_3, Y_2, Y_1, and Y_0. A sum-of-products solution would lead to four AND-OR circuits, one for Eq. (4.1), a second for Eq. (4.2), and so on. The ROM solution is different. With a ROM you have to store the binary numbers of Table 4.9 (same as Table 4.10) at the indicated addresses. When this is done, the ROM given in Fig. 4.40 is equivalent to a sum-of-products circuit. In other words, you can use the ROM instead of an AND-OR circuit to generate the desired truth table.

Table 4.10 **Truth Table**

A	B	C	Y_3	Y_2	Y_1	Y_0
0	0	0	0	1	1	1
0	0	1	1	0	0	0
0	1	0	1	0	1	1
0	1	1	1	1	0	0
1	0	0	0	1	1	0
1	0	1	1	0	0	1
1	1	0	0	0	1	1
1	1	1	1	1	1	0

Programmable ROMs

A *programmable ROM (PROM)* allows the user instead of the manufacturer to store the data. An instrument called a *PROMprogrammer* stores the words by "burning in." Here is an example of how a PROM programmer works. Originally, all diodes are connected at the cross points. For instance, in Fig. 4.40 there would be a total of 32 diodes (8 rows and 4 columns). Each of these diodes has a *fusible link* (a small fuse). The PROM programmer sends destructively high currents through all diodes to be removed. In this way, only the desired diodes remain connected after programming a PROM. Programming like this is permanent because the data cannot be erased after it has been burned in.

Here are some commercially available PROMs:

74S188: 256 bits organized as 32 × 8

74S287: 1024 bits organized as 256 × 4

74S472: 4096 bits organized as 512 × 8

PROMs such as these are useful for small production runs. For instance, if you are building only a few hundred units (or maybe even just one), you would choose a PROM rather than a ROM.

Since PROMs are useful in many applications, manufacturers produce these chips in high volume. Furthermore, the PROM is a universal logic solution. Why? Because the AND gates generate all the fundamental products; the user can then OR these products as needed to generate any Boolean output. One disadvantage of PROMs is the limit on number of input variables; typically, PROMs have 8 inputs or less.

Simplified Drawing of a PROM

It is cumbersome to draw large PROMs as illustrated in Fig. 4.41, because of the large number of diodes. An alternative, streamlined drawing procedure for PROMs like the one in Fig. 4.40 is shown in Fig. 4.41. In this simplified drawing, the solid black bullets indicate connections to the AND-gate inputs. Each bullet represents a *fixed* connection that cannot be changed. Furthermore, each AND gate has 3 inputs, indicated by the bullet on its input line. Similarly, each OR gate has 8 inputs, as indicated by the ×'s on its input line, but each × is a fusible link that can be removed.

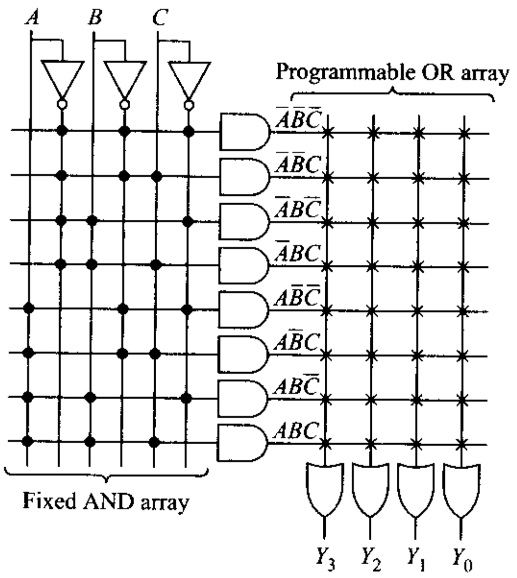


Fig. 4.41 **Streamlined drawing of PROM**

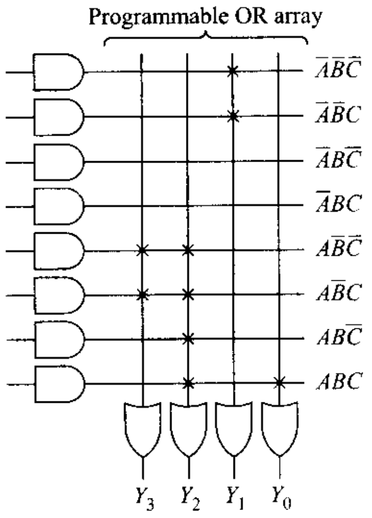
Notice that the input side of Fig. 4.41 is a fixed AND array, meaning the inputs to the AND gates are not programmable in a PROM. On the other hand, the output side of the circuit is programmable because each connection at the input of each OR gate is a fusible link. A fixed AND array and a programmable OR array are characteristic of all PROMs. To begin with, every AND-gate output is connected to every OR-gate input. Since the AND gates produce all eight possible combinations of the input variables A, B and C, it is possible to produce any Boolean function at the OR-gate outputs.

Programming a PROM

Generating a Boolean function at the output of a PROM is accomplished by *fusing* (melting) fusible links at the input to the OR gates in Fig. 4.41. For example, suppose we want to generate the function $Y_0 = ABC$. Simply fuse (melt) 7 of the AND-gate outputs connected to the Y_0 OR-gate input and leave the single AND-gate output ABC connected. A portion of Fig. 4.41 is shown in Fig. 4.43 with the proper fusible link remaining for Y_0.

As a second example, suppose we want to generate the function $Y_1 = \overline{A}\,\overline{B}$. We must include all terms containing $\overline{A}\,\overline{B}$, since

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


Programmable OR array
$\overline{A}\,\overline{B}\,\overline{C}$
$\overline{A}\,\overline{B}C$
$\overline{A}B\overline{C}$
$\overline{A}BC$
$A\overline{B}\,\overline{C}$
$A\overline{B}C$
$AB\overline{C}$
ABC
Y_3
Y_2
Y_1
Y_0
Fig. 4.42 **Boolean function from PROM**

The two top fusible links must be included, while the remaining six are broken, as shown in Fig. 4.42. Continuing in this fashion, you can see that $Y_2 = A$ and $Y_3 = A\overline{B}$.

Erasable PROMs

The *erasable PROM* (*EPROM*) uses metal-oxide-semiconductor field-effect transistors (MOSFETs). Data is stored with an EPROM programmer. Later, data can be erased with ultraviolet light. The light passes through a quartz window in the IC package. When it strikes the chip, the ultraviolet light releases all stored charges. The effect is to wipe out the stored contents. In other words, the EPROM is ultraviolet-light-erasable and electrically reprogrammable.

Here are some commercially available EPROMs:

2716: 16,384 bits organized as 2048 × 8

2732: 32,768 bits organized as 4096 × 8

The EPROM is useful in project development. With an EPROM, the designer can modify the contents until the stored data is perfect. When the design is finalized, the data can be burned into PROMs (small production runs) or sent to an IC manufacturer who produces ROMs (large production runs).

SELF-TEST

20. What is a ROM?
21. What does it mean to say that a particular ROM is "512 × 8"?
22. What is a PROM?

4.11 PROGRAMMABLE ARRAY LOGIC

Programmable array logic (PAL) is a programmable array of logic gates on a single chip. PALs are another design solution, similar to a sum-of-products solution, product-of-sums solution, and multiplexer logic.

Programming a PAL

A PAL is different from a PROM because it has a programmable AND array and a fixed OR array. For instance, Fig. 4.43 shows a PAL with 4 inputs and 4 outputs. The x's on the input side are fusible links, while the solid black bullets on the output side are fixed connections. With a PROM programmer, we can burn in the desired fundamental products, which are then ORed by the fixed output connections.

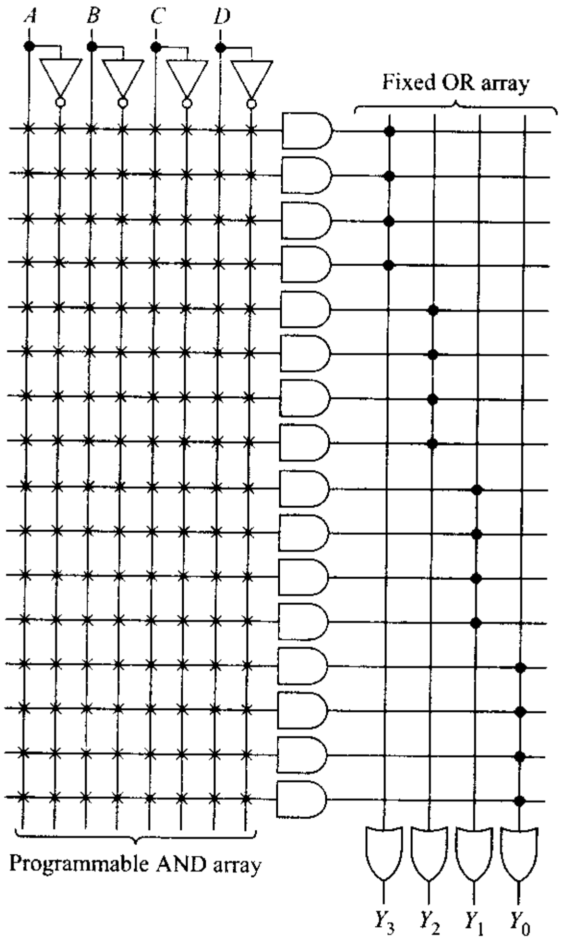


Fig. 4.43 **Structure of PAL**

Here is an example of how to program a PAL. Suppose we want to generate the following Boolean functions:

$$Y_3 = \overline{A}B\overline{C}D + \overline{A}BC\overline{D} + \overline{A}BCD + ABC\overline{D} \tag{4.5}$$

$$Y_2 = \overline{A}BC\overline{D} + \overline{A}BCD + ABCD \tag{4.6}$$

$$Y_1 = \overline{A}B\overline{C} + \overline{A}BC + A\overline{B}C + AB\overline{C} \tag{4.7}$$

$$Y_0 = ABCD \tag{4.8}$$

Start with Eq. (4.5). The first desired product is $\overline{A}B\overline{C}D$. On the top input line of Fig. 4.44 we have to remove the first ×, the fourth ×, the fifth ×, and the eighth ×. Then the top AND gate has an output of $\overline{A}B\overline{C}D$.

By removing ×s on the next three input lines, we can make the top four AND gates produce the fundamental products of Eq. (4.5). The fixed OR connections on the output side imply that the first OR gate produces an output of

$$Y_3 = \overline{A}B\overline{C}D + \overline{A}BC\overline{D} + \overline{A}BCD + ABC\overline{D}$$

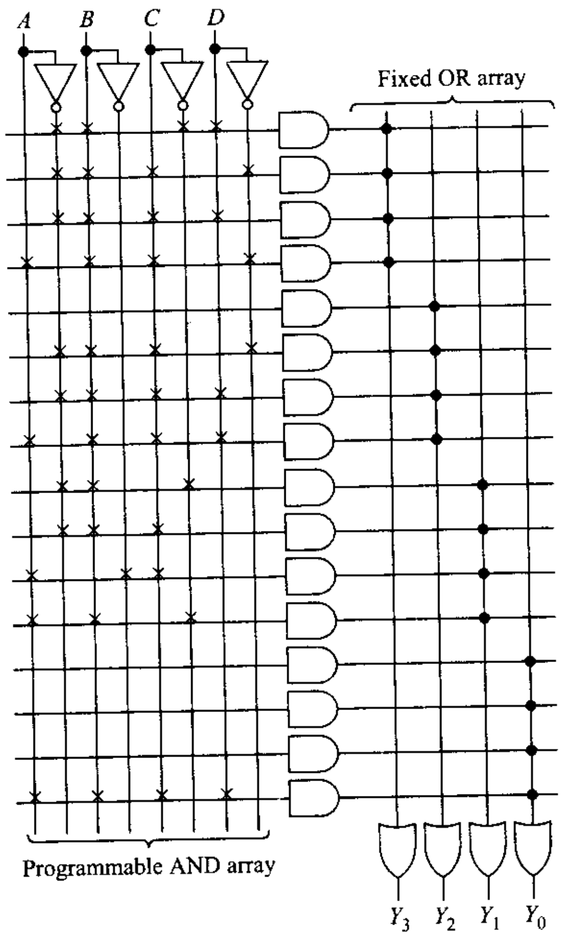


Fig. 4.44 **Example of programming a PAL**

Similarly, we can remove ×s as needed to generate Y_2, Y_1, and Y_0. Figure 4.44 shows how the PAL looks after the necessary ×s have been removed. If you examine this circuit, you will see that it produces the Y outputs given by Eqs. (4.5) to (4.8).

Commercially Available PALs

The PAL given in Fig. 4.43 is hypothetical. Commercially available PALs typically have more inputs. For instance, here is a sample of some TTL PALs available from National Semiconductor Corporation:

10H8; 10 input and 8 output AND-OR

16H2: 6 input and 2 output AND-OR

14L4: 14 input and 4 output AND-OR-INVERT

For these chip numbers, H stands for active-high output and L for active-low output. The 10H8 and the 16H2 produce active-high outputs because they are AND-OR PALs. The 14L4, on the other hand, produces an active-low output because it is an AND-OR-INVERT circuit (one that has inverters at the final outputs).

Unlike PROMs, PALs are not a universal logic solution. Why? Because only some of the fundamental products can be generated and ORed at the final outputs. Nevertheless, PALs have enough flexibility to produce all kinds of complicated logic functions. Furthermore, PALs have the advantage of 16 inputs compared to the typical limit of 8 inputs for PROMs.

SELF-TEST

23. What is a PAL?
24. A PAL has an AND array and an OR array. Which one is fixed and which is programmable?

4.12 PROGRAMMABLE LOGIC ARRAYS

Programmable logic arrays (*PLAs*), along with ROMs and PALs, are included in the more general classification of ICs called *programmable logic devices* (*PLDs*). Figure 4.45 illustrates the basic operation of these three PLDs. In each case, the input signals are presented to an array of AND gates, while the outputs are taken from an array of OR gates.

The input AND-gate array used in a PROM is *fixed* and cannot be altered, while the output OR-gate array is *fusible-linked*, and can thus be programmed. The PAL is just the opposite: The output OR-gate array is fixed, while the input AND-gate array is fusible-linked and thus programmable. The PLA is much more versatile than the PROM or the PAL, since both its AND-gate array and its OR-gate array are fusible-linked and programmable. It is also more complicated to utilize since the number of fusible links are doubled.

A PLA having 3 input variables (ABC) and 3 output variables (XYZ) is illustrated in Fig. 4.46. Eight AND gates are required to decode the 8 possible input states. In this case, there are three OR gates that can be used to generate logic functions at the output. Note that there could be additional OR gates at the output if desired. Programming the PLA is a two-step process that combines procedures used with the PROM and the PAL.

As an example, suppose it is desired to use a PLA to recognize each of the 10 decimal digits represented in binary form and to correctly drive a 7-segment display. The 7-segment indicator was presented in Sec. 4.5. To begin with, the PLA must have 4 inputs, as shown in Fig. 4.47a. Four bits ($ABCD$) are required to represent the 10 decimal numbers (see Table 1.1). There must be 7 outputs ($abcdefg$), 1 output to drive each

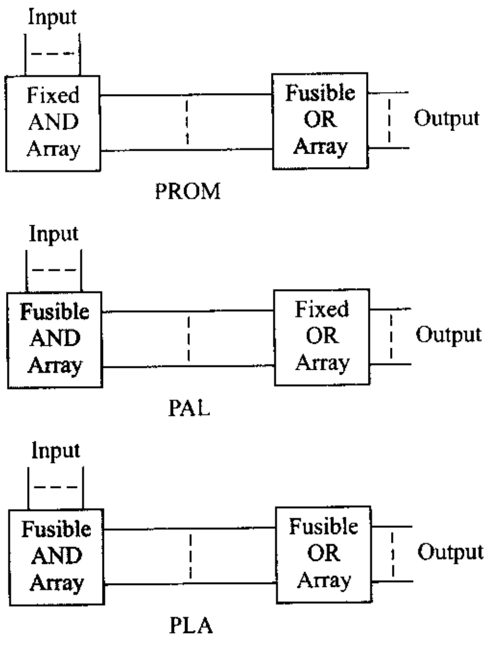


Fig. 4.45

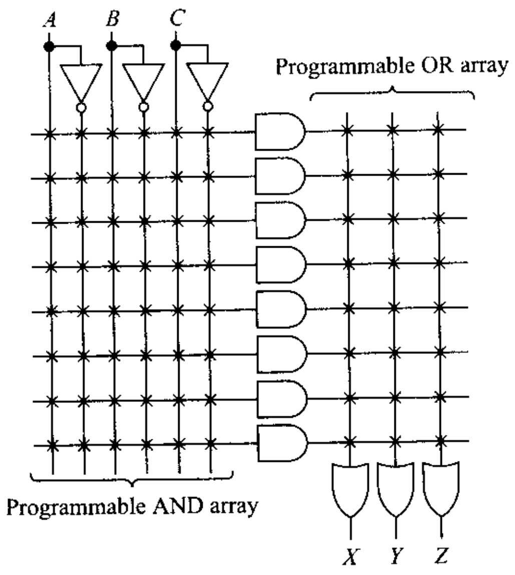


Fig. 4.46

of the 7 segments of the indicator. Let's assume that our PLA is capable of driving the 7-segment indicator directly. (This is not always a valid assumption, and a buffer amplifier may be needed to supply the proper current for the indicator.)

To begin with, all fusible links are good. The circuit in Fig. 4.47b shows the remaining links after programming. The input AND-gate array is programmed (fusible links are removed) such that each AND gate decodes one of the decimal numbers. Then, with the use of Fig. 4.47c, links are removed from the output OR-gate array such that the proper segments of the indicator are illuminated. For instance, when $ABCD$ = $LHLH$, segments *afgcd* are illuminated to display the decimal number 5. You should take the time to examine the other nine digits to confirm proper operation.

One final point. Many PLDs are programmable only at the factory. They must be ordered from the manufacturer with specific programming instructions. There are, however, PLDs that can be programmed by the user. These are said to be *field-programmable*, and the letter F is often used to indicate this fact. For instance, the Texas Instruments TIFPLA840 is a field-programmable PLA with 14 input variables, 32 AND gates, and 6 OR gates; it is described as a 14 × 32 × 6 FPLA.

SELF-TEST

25. **What is a PLA?**
26. **How does a PLA differ from a PAL?**
27. **In Fig. 4.47, *ABCD* = *LLHH*. What segments are activated?**

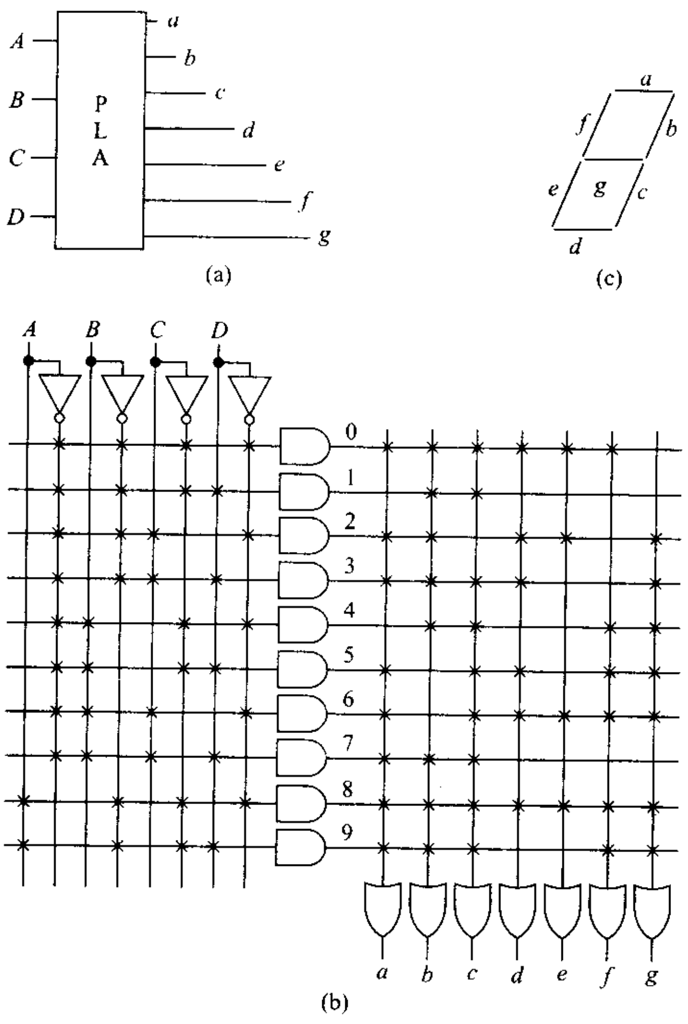


Fig. 4.47 7-segment decoder using PLA

4.13 TROUBLESHOOTING WITH A LOGIC PROBE

Chapter 3 introduced the logic clip, a device that connects to a 14 or 16-pin IC. The logic clip contains 16 LEDs that monitor the state of the pins. When a pin voltage is high, the corresponding LED lights up. When the pin voltage is low, the LED is dark.

Figure 4.48 shows a *logic probe*, which is another troubleshooting tool you will find helpful in diagnosing faulty circuits. When you touch the probe tip to the output node as shown, the device lights up for a high state and goes dark for a low state. For instance, if either A or B, or both, are low, then Y is high and the probe lights up. On the other hand, if A and B are both high, Y is low and the probe is dark.

Among other things, the probe is useful for locating short circuits that occur in manufacturing. For example, during the stuffing and soldering of printed-circuit boards, an undesirable splash of solder may connect two adjacent traces (conducting lines). Known as a *solder bridge*, this kind of trouble can short-circuit a node to the ground or to the supply voltage. The node is then stuck in a low or high state. The probe helps you to find short-circuited nodes because it stays in one state, no matter how the inputs are changing.

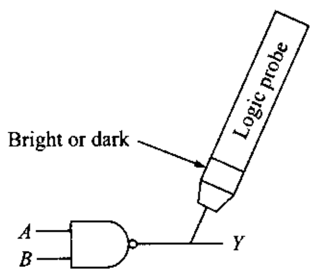


Fig. 4.48 **Using a logic probe**

4.14 HDL IMPLEMENTATION OF DATA PROCESSING CIRCUITS

We start with hardware design of multiplexers using Verilog code. The data flow model provides a different use of keyword **assign** in the form of

assign $X = S \,?\, A : B$;

This statement does following assignment. If, $S = 1, X = A$ and if $S = 0, X = B$. One can use this statement or the logic equation to realize a 2 to 1 multiplexer shown in Fig. 4.2(a) in one of the following ways.

```
module mux2to1(A,D0,D1,Y);
  input A,D0,D1; /* Circuit shown
  in Fig. 4.3(a)*/
  output Y;
  assign Y=(~A&D0)|(A&D1);
endmodule
```

```
module mux2to1(A,D0,D1,Y);
  input A,D0,D1; /* Circuit shown in
  Fig. 4.3(a)*/
  output Y;
  assign Y= A ? D1 : D0; /*Conditional
  assignment*/
endmodule
```

The behavioral model can be used to describe the 2 to 1 multiplexers in following two different ways, one using **if ... else** statement and the other using **case** statement. The **case** evaluates an expression or a variable that can have multiple values each one corresponding to one statement in the following block. Depending on value of the expression, one of those statements get executed. The behavioral model of 2 to 1 multiplexer in both is given below:

```
module mux2to1(A,D0,D1,Y);
   input A,D0,D1; /* Circuit shown
   in Fig. 4.3(a)*/
output Y;
reg Y;
always @ (A or D0 or D1)
   if (A==1) Y=D1;
   else Y=D0;
endmodule
```

```
module mux2to1(A,D0,D1,Y);
   input A,D0,D1; /* Circuit shown
   in Fig. 4.3(a)*/
   output Y;
   reg Y;
   always @ (A or D0 or D1)
       case (A)
          0 : Y=D0;
          1 : Y=D1;
       endcase
endmodule
```

Example 4.14 Design a 4 to 1 multiplexer, shown in Fig. 4.1(c) using conditional **assign** and **case** statements.

Solution The codes are given next. We have used nested condition for **assign** statement. If A = 1, condition (B ? D3 : D2) is evaluated. Then if B = 1, Y = D3. And this is what is given in Fig. 4.1(c). Similarly, the other combinations of A and B are evaluated and Y is assigned a value from D2 to D0. For **case** statement we concatenated A and B by using operator {...} and generated four possible combinations. For a particular value of AB, statement corresponding to one of them gets executed.

```
module mux4to1(A,B,D0,D1,D2,D3,Y);
  input A,B,D0,D1,D2,D3;
  output Y; /* Circuit shown
  in Fig. 4.1(c)*/
  assign  Y = A ?( B ? D3 : D2):(B ?
  D1 : D0);
endmodule
```

```
module mux4to1(A,B,D0,D1,D2,D3,Y);
  input A,B,D0,D1,D2,D3;
  output Y;
  reg Y;
  always @ (A or B or D0 or D1 or D2
  or D3)
    case ({A,B}) /*Concatenation of
    A and B, A is MSB*/
     0: Y=D0; /*Two binary digit can
        generate*/
     1: Y=D1; /*four different values
        0,1,2,3 for*/
     2: Y=D2; /*binary combination
        00,01,10*/
     3: Y=D3; //and 11 respectively
   endcase
 endmodule
```

BUS Representation in HDL

We introduce concept of BUS or vector representation in HDL description through design of a 1 to 4 demultiplexer that can also serve as a 2 to 4 decoder. The data input of former is treated as enable input of later. We consider S as a select input defined by two binary digits S[1] and S[0]. Output Y is 4 bit long, one of which goes high for a particular combination of select inputs if data(enable) input is high. The Verilog code for this demultiplexer/decoder is given below:

```
module demux1to4(S,D,Y);
  input [1:0] S;
  input  D;
  output [3:0] Y;
  reg [3:0] Y;
  always @ (S or D)
     case ({D,S}) //Concatenation of D  and S to give 3 bits, D is MSB
       3'b100 : Y= 4'b0001; /*Binary representation, refer to Section 2-5.
      If D=1, S=00, Y=0001*/
       3'b101 : Y= 4'b0010; // if D=1, S=01, Y=0010
       3'b110 : Y= 4'b0100; // if D=1, S=10, Y=0100
       3'b111 : Y= 4'b1000; // if D=1, S=11, Y=1000
       default : Y= 4'b0000; //For other combinations D=0, then Y=0000
    endcase
endmodule
```

Example 4.15 A verilog HDL code for a digital circuit is given as follows. Can you describe the function it performs? Can it be related to any logic circuit discussed in this chapter?

```
module unknown(A,B,C,Y);
  input [3:0] A,B;
  input  [2:0] C;
  output [2:0] Y;
  reg [2:0] Y;
  always @ (A or B or C)
     if (A<B) Y=3'b001;
     else if (A>B) Y=3'b010;
     else Y=C;
endmodule
```

Solution The circuit described by the HDL compares two 4-bit numbers A and B and generates a 3 bit output Y. It has also a 3 bit input C. If A is less than B, output $Y = 001$ and does not depend on C. Similarly, if A is greater than B, $Y = 010$ irrespective of C. But if these two conditions are not met, i.e. if $A = B$ then $Y = C$.

If we consider three bits of Y represent (starting from MSB) $A = B$, $A > B$ and $A < B$ respectively then, this circuit represents a 4-bit magnitude comparator where C represents comparator output of previous stage that is of lower significance. If numbers of this stage are equal then the value at C that represents equal, greater than, less than condition of previous stage numbers is reflected by Y. This is similar to IC 7485 discussed in Section 4.9.

PROBLEM SOLVING WITH MULTIPLE METHODS

Problem Show how data processing circuits can be used to compare two 2-bit numbers, A_1A_0 and B_1B_0 to generate two outputs, $A > B$ and $A = B$.

Solution We can use multiplexers, decoder or simply a 4-bit comparator. The truth table of the above problem is shown in Fig. 4.49.

In Method-1, we use two 16 to 1 multiplexers to realize $A > B$ and $A = B$ as shown in Fig. 4.50. The numbers A_1A_0 and B_1B_0 are used as selection inputs as shown. For every selection of input, the

A_1A_0	B_1B_0	$A>B$	$A=B$
00	00	0	1
00	01	0	0
00	10	0	0
00	11	0	0
01	00	1	0
01	01	0	1
01	10	0	0
01	11	0	0
10	00	1	0
10	01	1	0
10	10	0	1
10	11	0	0
11	00	1	0
11	01	1	0
11	10	1	0
11	11	0	1

Fig. 4.49 **Truth table**

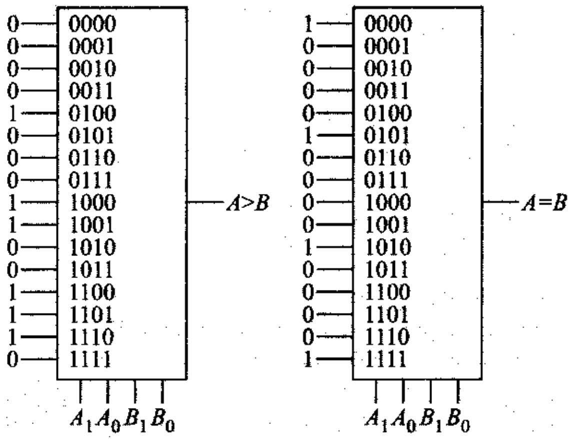


Fig. 4.50 **Solution using 16 to 1 multiplexers**

corresponding data input goes to the output. The input assignment comes straight from the truth table in Fig. 4.49 for the two cases.

In Method-2, we use two 8 to 1 multiplexers to realize $A > B$ and $A = B$ as shown in Fig. 4.51. The numbers A_1A_0 and B_1 are used as selection inputs while B_0 is part of the data input. We form pair of combinations of the truth table for constant $A_1A_0B_1$ and B_0 variable. This helps to find out how output varies with B_0.

In Method-3, we use one 4 to 16 decoder and two multi-input OR gates to realize $A > B$ and $A = B$ as shown in Fig. 4.52. We sum selected minterms, as required from the truth table, from the set of all the minterms generated by the decoder.

$A_1A_0B_1$	B_0	$A>B$	$A=B$
0 0 0	0	0 (0)	1 (B_0')
0 0 0	1	0	0
0 0 1	0	0 (0)	0 (0)
0 0 1	1	0	0
0 1 0	0	1 (B_0')	0 (B_0)
0 1 0	1	0	1
0 1 1	0	0 (0)	0 (0)
0 1 1	1	0	0
1 0 0	0	1 (1)	0 (0)
1 0 0	1	1	0
1 0 1	0	0 (0)	1 (B_0')
1 0 1	1	0	0
1 1 0	0	1 (1)	0 (0)
1 1 0	1	1	0
1 1 1	0	1 (B_0')	0 (B_0)
1 1 1	1	0	1

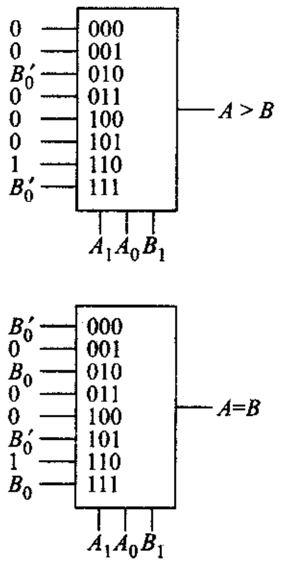


Fig. 4.51 **Solution using 8 to 1 multiplexers**

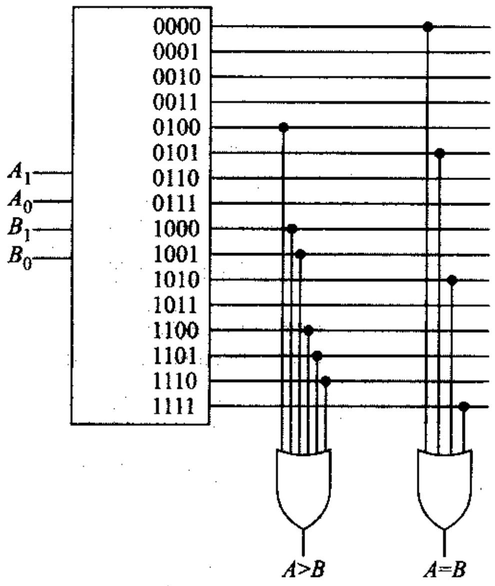


Fig. 4.52 **Solution using 4 to 16 decoder**

Though we are not presenting them as a separate method, the AND bank (inside decoder) and OR bank combination concept presented here can be used to obtain solution from programmable logic devices such as PLA, PAL, etc.

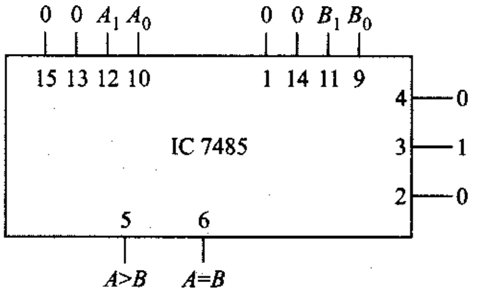


Fig. 4.53 **Solution using 4-bit comparator**

In Method-4, we follow a straightforward approach to use a 4-bit comparator (IC 7485 : Fig. 4.38a) for the purpose as shown in Fig. 4.53. We keep the higher two bits '0' and '$A = B$ input' high so that it essentially becomes a 2-bit comparator generating all three outputs $A > B$, $A = B$ and $A < B$ of which only first two are useful here.

SUMMARY

A multiplexer is a circuit with many inputs but only one output. The 16-to-1 multiplexer has 16 input bits, 4 control bits, and 1 output bit. The 4 control bits select and steer 1 of the 16 inputs to the output. The multiplexer is a universal logic circuit because it can generate any truth table.

A demultiplexer has one input and many outputs. By applying control signals, we can steer the input signal to one of the output lines. A decoder is similar to a demultiplexer, except that there is no data input. The control bits are the only input. They are decoded by activating one of the output lines.

BCD is an abbreviation for binary-coded decimal. The BCD code expresses each digit in a decimal number by its nibble equivalent. A BCD-to-decimal decoder converts a BCD input to its equivalent decimal value. A seven-segment decoder converts a BCD input to an output suitable for driving a seven-segment indicator.

An encoder converts an input signal into a coded output signal. An example is the decimal-to-BCD encoder. An exclusive-OR gate has a high output only when an odd number of inputs are high. Exclusive-OR gates are useful in parity generators-checkers.

Magnitude comparators are useful in comparing two binary numbers. It generates three outputs that give if one number is greater, equal or less than the other number. Cascading magnitude comparators we can compare two numbers of any size.

A ROM is a read-only memory. Smaller ROMs are used to implement truth tables. ROMs are expensive because they require a mask for programming. PROMs are user-programmable and ideal for small production runs. EPROMs are not only user-programmable, but they are also erasable and reprogrammable during the design and development cycle. PALs are chips that are programmable arrays of logic. Unlike the PROM with its fixed AND array and programmable OR array, a PAL has programmable AND array and a fixed OR array. The PAL has the advantage of having up to 16 inputs in commercially available devices. In the PLA both the AND array and the OR array are programmable. The PLA is a much more versatile programmable logic device (PLD) IC than the PROM or the PAL.

GLOSSARY

- ***active low*** The low state is the one that causes something to happen rather than the high state.
- ***BCD*** A binary-coded decimal.
- ***data selector*** A synonym for multiplexer.
- ***decoder*** A circuit that is similar to a demultiplexer, except there is no data input. The control input bits produce one active output line.
- ***demultiplexer*** A circuit with one input and many outputs.
- ***EPROM*** An erasable programmable read-only memory. With this device, the user can erase the stored contents with ultraviolet light and electrically store new data. EPROMs are useful during project development where programs and data are being perfected.
- ***even parity*** A binary number with an even number of 1s.
- ***exclusive-OR gate*** A gate that produces a high output only when an odd number of inputs is high.
- ***LED*** A light-emitting diode.
- ***logic probe*** A troubleshooting device that indicates the state of a signal line.
- ***Magnitude comparator*** compares two binary numbers and signals if one is greater, equal or less than other.
- ***multiplexer*** A circuit with many inputs but only one output.
- ***odd parity*** A binary number with an odd number of 1s.
- ***PAL*** A programmable array logic (sometimes written PLA, which stands for programmable

logic array). In either case, it is a chip with a programmable AND array and a fixed OR array.

- ***parity generation*** An extra bit that is generated and attached to a binary number, so that the new number has either even or odd parity.
- ***PLA*** A programmable logic array.
- ***PLD*** A programmable logic device.
- ***PROM*** A programmable read-only memory. A type of chip that allows the user to program it with a PROM programmer that burns fusible links at the diode cross points. Once the data is stored, the programming is permanent. PROMs are useful for small production runs.
- ***ROM*** A read-only memory. An IC that can store many binary numbers at locations called addresses. ROMs are expensive to manufacture and are used only for large production runs where the cost of the mask can be recovered by sales.
- ***strobe*** An input that disables or enables a circuit.

PROBLEMS

Section 4.1

4.1 In Fig. 4.2, if $ABCD = 1001$, what does Y equal?

4.2 In Fig. 4.4, if $ABCD = 1100$, what does Y equal?

4.3 We want to implement Table 3.12 of the preceding chapter using multiplexer logic. Show a circuit, similar to the one in Fig. 4.4, that can do the job.

4.4 Show how to connect a 74150 to implement this Boolean equation:

$$Y = \overline{A}B\overline{C}D + A\overline{B}C\overline{D} + ABC\overline{D}$$

4.5 Draw a circuit with four 74150s that has a truth table like the one in Table 4.11.

4.6 Table 4.12 shows the Gray code. Show how four 74150s may be connected to convert from binary to Gray code. Show how the same can be realized by four 74151 ICs (8-to-1 multiplexer).

Table 4.11

A	B	C	D	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	1	0
0	0	0	1	1	0	1	0
0	0	1	0	0	0	0	0
0	0	1	1	0	1	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	0	1	0	0	0
0	1	1	1	1	0	0	0
1	0	0	0	0	1	0	1
1	0	0	1	0	0	0	1
1	0	1	0	0	0	0	0
1	0	1	1	0	0	0	0
1	1	0	0	0	0	0	0
1	1	0	1	0	1	0	1
1	1	1	0	1	0	1	0
1	1	1	1	0	0	1	0

Table 4.12 Gray Code

A	B	C	D	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

Section 4.2

4.7 In Fig. 4.12, if $ABCD$ = 0101, which is the active output line when the strobe is high? When it is low?

4.8 Input signals R and T are low in Fig. 4.13. Which is the active output line when $ABCD$ = 0011? To have the Y_9 output line active, what input signals do you need?

4.9 Suppose a logic probe shows that pin 19, given in Fig. 4.13, is always high. Which of the following may possibly cause trouble:

a. Pin 20 is grounded.

b. Pin 18 has a sine wave instead of pulses.

c. The R input is grounded.

d. The T input is connected to +5 V.

Section 4.3

4.10 Are the output signals of Fig. 4.15 active low or active high? For the IC to decode the $ABCD$ input, does the strobe have to be low or high?

4.11 In Fig. 4.16, suppose X = 1 and $ABCD$ = 0110. Which is the active chip and which is the active output line?

4.12 Design a circuit that realizes following two functions using a decoder and two OR gates.

$$F_1(A,B) = \Sigma\, m(0,3) \quad \text{and}$$
$$F_2(A,B) = \Sigma\, m(1,2)$$

4.13 Design a circuit that realizes following three functions using a decoder and three OR gates.

$$F_1(A,B,C) = \Sigma\, m(1,3,7),$$
$$F_2(A,B,C) = \Sigma\, m(2,3,5) \quad \text{and}$$
$$F_3(A,B,C) = \Sigma\, m(0,1,5,7)$$

Section 4.4

4.14 Convert the following decimal numbers into their BCD equivalents:

a. 32 b. 634

c. 4898

4.15 Convert the following BCD numbers into their decimal equivalents:

a. 0110 0111

b. 1000 0001 0011

c. 0111 0010 0101 1001

4.16 In Fig. 4.18, what is the high output line when $ABCD$ = 0101?

4.17 In Fig. 4.20, which is the low output when $ABCD$ = 0111?

4.18 Figure 4.54 shows a group of chips numbered 0 through 9. Each chip has an active-low STROBE input. Which chip is active for each of these conditions:

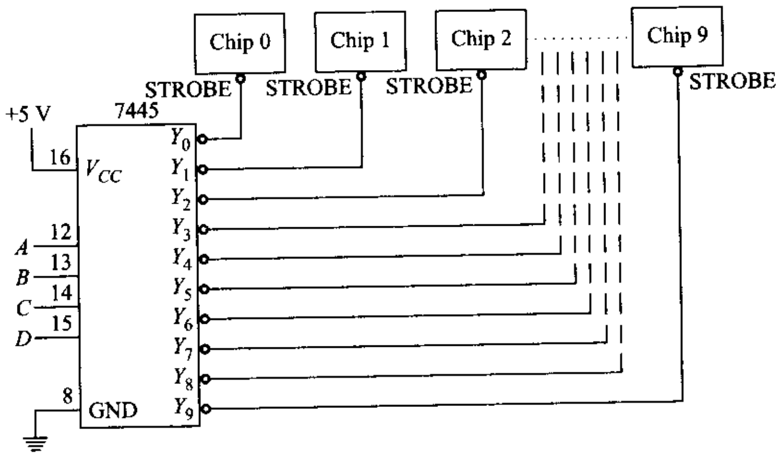


Fig. 4.54

a. $ABCD$ = 0000.
b. $ABCD$ = 0010.
c. $ABCD$ = 1001.

4.19 The $ABCD$ input of Fig. 4.54 initially equals 1111. For this condition, all output waveforms start high in the timing diagram of Fig. 4.55. Another circuit not shown is supposed to produce the following input values of $ABCD$: 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, and 1001.

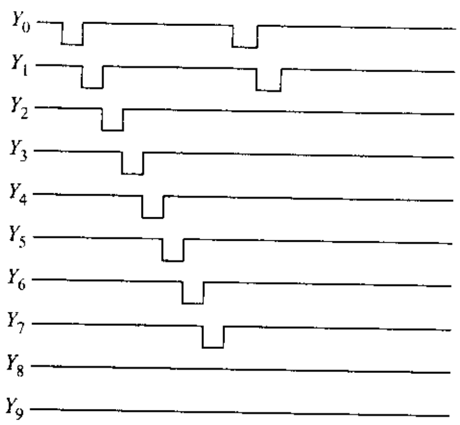


Fig. 4.55

The timing diagram tells us that something is wrong with the logic circuit of Fig. 4.54. Which of the following is a possible trouble:

a. Pin 16 is not connected to the supply voltage. b. Pin 8 is open.
c. Pin 12 is short-circuited to the ground.
d. Pin 15 is short-circuited to +5 V.

Section 4.5

4.20 In Fig. 4.21, which of the segments have to be active to display the following digits:

a. 2 b. 6
c. 8

4.21 In Fig. 4.23a, V_{CC} = +5 V, all resistors are 1 kΩ, and each LED has a voltage drop of 2 V. Approximately how much current is there through an active segment?

Section 4.6

4.22 In Fig. 4.25, what is the output when button 7 is pressed? When button 3 is pressed?

4.23 In Fig. 4.27, if button 8 is pressed, which is the input pin that goes into the low state? What does the $ABCD$ output equal?

Section 4.7

4.24 In Fig. 4.32d, what does Y equal for each of the following inputs:

a. 000110 b. 011001
c. 011111 d. 111100

4.25 In Fig. 4.33, what does Y equal for the following inputs:

a.	1111	0000	1111	0000
b.	0101	1010	1100	0111
c.	1110	1011	1101	0001
d.	0001	0101	0011	0110

4.26 In Fig. 4.56, the 8-bit register is a logic circuit that stores byte $A_7 \ldots A_0$. What does byte $Y_7 \ldots Y_0$ equal for each of these conditions:

a. $A_7 \ldots A_0$ = 1000 0111 and INVERT = 0.
b. $A_7 \ldots A_0$ = 0011 1100 and INVERT = 1.
c. $A_7 \ldots A_0$ = 1111 0000 and INVERT = 0.
d. $A_7 \ldots A_0$ = 1110 0001 and INVERT = 1.

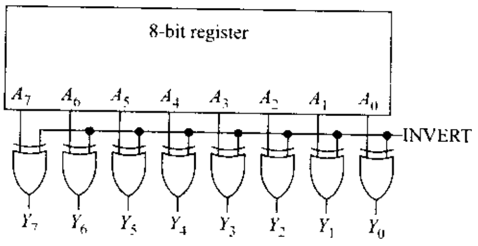
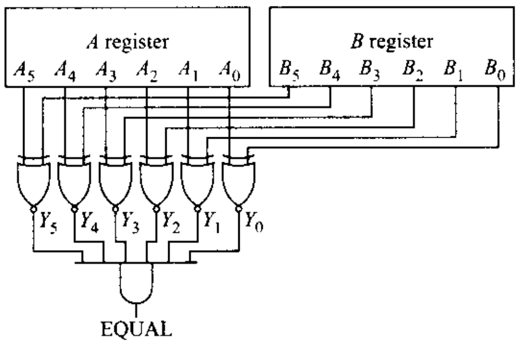

8-bit register
A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0
INVERT
Y_7 Y_6 Y_5 Y_4 Y_3 Y_2 Y_1 Y_0

Fig. 4.56

4.27 In Fig. 4.57 on the next page, each register is a logic circuit that stores a 6-bit number. The left register stores $A_5 \ldots A_0$ and the right register stores $B_5 \ldots B_0$. What value does the output signal labeled $EQUAL$ have for each of these:



A register
A_5 A_4 A_3 A_2 A_1 A_0
B register
B_5 B_4 B_3 B_2 B_1 B_0
Y_5 Y_4 Y_3 Y_2 Y_1 Y_0
EQUAL

Fig. 4.57

a. $A_7 \ldots A_0$ is less than $B_7 \ldots B_0$.
b. $A_7 \ldots A_0$ equals $B_7 \ldots B_0$.
c. $A_7 \ldots A_0$ is greater than $B_7 \ldots B_♦$

Sections 4.8 and 4.9

4.28 In Fig. 4.58, what does X_8 equal for each of the following $X_7 \ldots X_0$ inputs:

a. 0000 1111 b. 1111 0001
c. 1010 1110 d. 1011 1100

4.29 In Fig. 4.58, what changes can you make to get a 9-bit output with even parity?

4.30 In Fig. 4.58, assuming the circuit is working all right, what will the logic probe indicate for each of the following:

a. Input data has even parity.
b. Input data has odd parity.
c. Pins 3 and 4 are grounded.

4.31 Write the $(X > Y)$ equation for a 4-bit comparator.

4.32 Show how magnitude of two 10-bit numbers can be compared using IC 7485.

4.33 Suppose a ROM has 8 input address lines. How many memory locations does it have?

4.34 Two 74S370s are connected in parallel. To address all memory locations, how many bits must the binary address have?

4.35 In Fig. 4.40, if $ABC = 011$, what does $Y_3Y_2Y_1Y_0$ equal?

4.36 Draw a ROM circuit similar to the one in Fig. 4.40 that produces these outputs:

$$Y_3 = \bar{A}B\bar{C}$$
$$Y_2\ A\bar{B}C + ABC$$
$$Y_1 = A\bar{B}C + \bar{A}BC + ABC$$
$$Y_0 = \bar{A}B\bar{C} + \bar{A}BC + AB\bar{C} + ABC$$

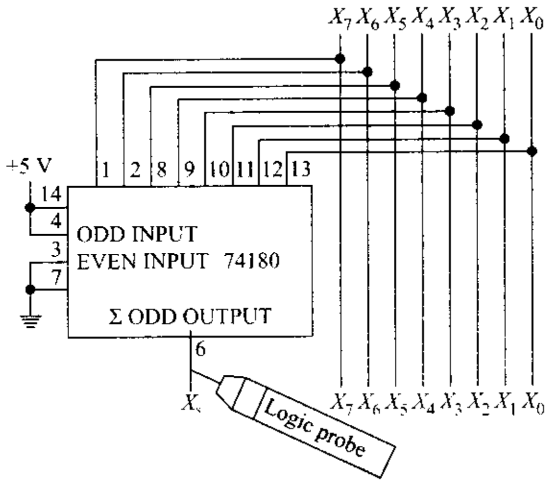


Fig. 4.58

Section 4.10

4.37 Draw a PROM circuit similar to the one in Fig. 4.41 that generates the Y_3 to Y_0 output given in Table 4.11.

4.38 What is the Boolean equation for Y_3 in Fig. 4.59 on the previous page? For Y_2? For Y_1? For Y_0?

4.39 Draw a 4-input and 4-output PAL circuit that has the truth table of Table 4.11.

Section 4.11

4.40 Write the Boolean expression for the output Y_3 in Fig. 4.42.

4.41 The input to the PLA in Fig. 4.47 is $ABCD$ = 0011. What segments of the indicator are illuminated and what decimal number is displayed? What if $ABCD$ = 1001? What about 1111?

4.42 Will there be any ambiguity if segment g of the 7-segment indicator in Fig. 4.47 is defective (burned out)? What numbers are displayed?

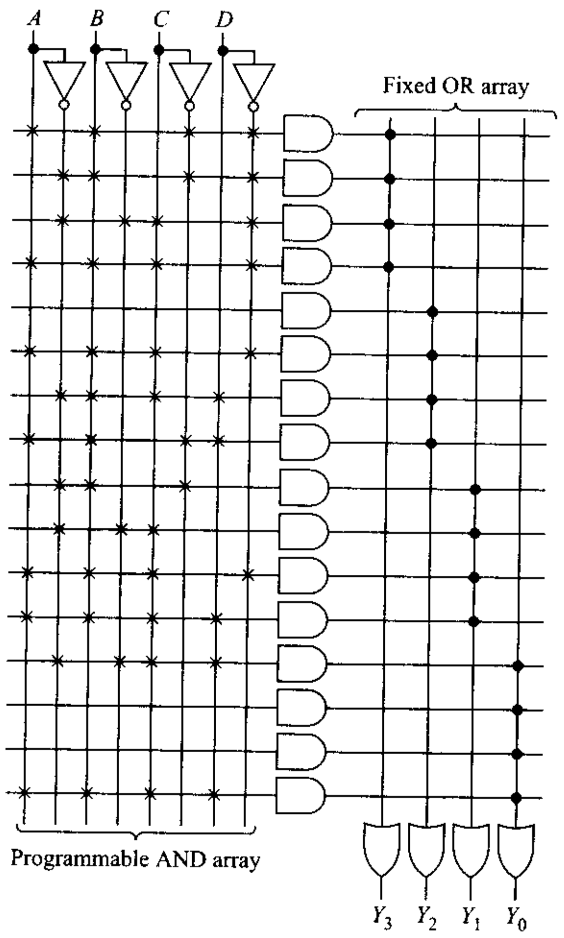
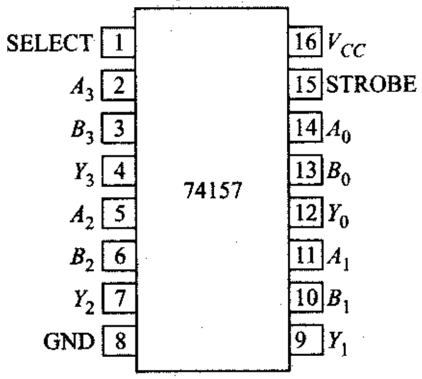


Fig. 4.59

LABORATORY EXPERIMENT

AIM: The aim of this experiment is to display one of two BCD numbers in a 7-segment display.

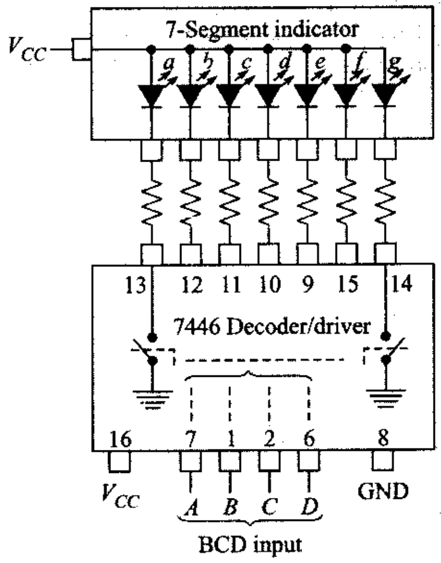
Theory: The two BCD numbers can be selected by activating select line of a multiplexer.



The multiplexer output then is one of the two BCD numbers. These outputs can be connected to four inputs of a 7-segment decoder/driver. The outputs of this driver can be connected to a 7-segment display to display the decimal equivalent of the BCD number selected.

Apparatus: 5 V DC Power supply, Multimeter, and Bread Board

Work element: Verify the truth table of multiplexer IC 74157. Note that STROBE is an input and find its use. The common select line applies to all four 2-to-1 multiplexers. Verify the truth table of IC7446 for BCD inputs. Select



resistance values to be connected in the range 220 to 1000 ohm. This is to ensure that entire power supply voltage does not drop across the LED of display. Interconnect properly all the different units and verify.

Answers to Self-tests

1. Multiplexer
2. It means that Y is active low.
3. Demultiplexer
4. $ABCD = HLHL$.
5. It will be high since STROBE is high.
6. The outputs are active low.
7. Y_{10} is low; all other outputs are high.
8. BCD stands for binary-coded decimal.
9. LED stands for light-emitting diode.
10. See Fig. 4.21.
11. Each segment is an LED.
12. An encoder converts an active input signal into a coded output signal, for instance, decimal to binary.
13. The TTL 74147 is a decimal-to-binary encoder.

14. The output for an exclusive-OR gate is high only when an *odd* number of inputs are high.
15. See Fig. 4.30.
16. There are an *even* number of 1s (highs).
17. True.
18. Three: $X = Y$, $X > Y$ and $X < Y$.
19. Three.
20. ROM stands for read-only memory.
21. A 512×8 ROM is arranged as 512 eight-bit words.
22. PROM stands for programmable read-only memory.
23. PAL stands for programmable array logic.
24. The AND array is programmable; the OR array is fixed.
25. PLA stands for programmable logic array.
26. In a PLA, both the AND array and the OR array are programmable.
27. Decimal 3; segments *abcdg*