MODULE – 3

COMBINATIONAL LOGIC CIRCUITS

COMBINATIONAL CIRCUIT DESIGN & SIMULATION USING GATES

REVIEW OF COMBINATIONAL CIRCUIT DESIGN:

Steps involved in the design of a combinational switching circuit:

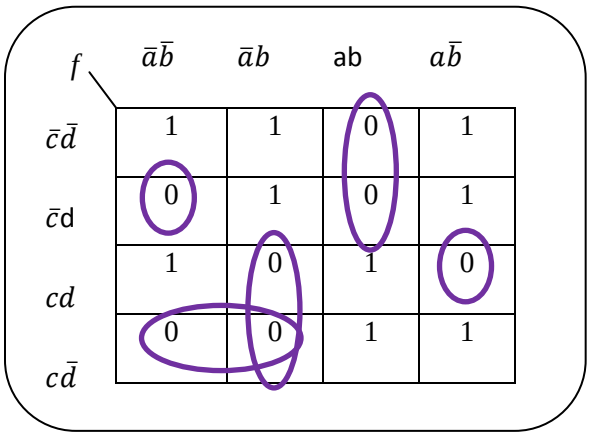
1. Set up a truth table which specifies the output(s) as a function of the input variables. If a given combination of values for the input variables can never occur at the circuit inputs, the corresponding output values are don't-cares.
2. Derive simplified algebraic expressions for the output functions using Karnaugh Maps, or Quine-McCluskey method, or any other similar procedure. The resulting algebraic expressions are then manipulated into the proper form, depending on the type of gates to be used in realizing the circuit.
3. When a circuit has two or more outputs, common terms in the output functions can often be used to reduce the total number of gates or gate inputs.
4. Minimum two-level AND-OR, or NAND-NAND circuits can be realized using the minimum sum-of-products. Minimum two-level OR-AND, or NOR-NOR circuits can be realized using the minimum product-of-sums.

DESIGN OF CIRCUITS WITH LIMITED GATE FAN-IN:

In practical logic design problems, the maximum number of inputs on each gate (or the fan-in) is limited. Depending on the type of gates used, this limit may be two, three, four, eight, or some other number. If a two-level realization of a circuit requires more gate inputs than allowed, factoring the logic expression to obtain a multi-level realization is necessary.

Example: *Realize* $f(a,b,c,d) = \Sigma m(0,3,4,5,8,9,10,14,15)$ *using three input NOR gates.*

Solution:



The product-of-sum equation is:

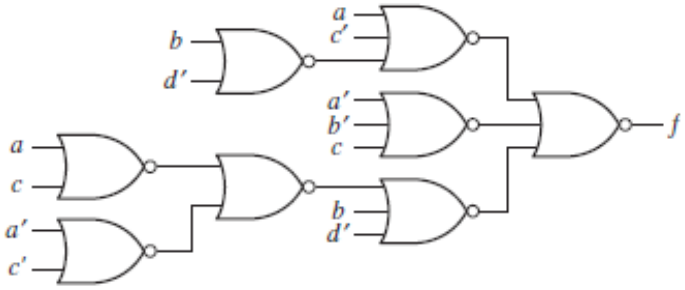
$$f = (a' + b' + c)(a + b' + c')(a + c' + d)(a + b + c + d')(a' + b + c' + d')$$

As can be seen from the preceding expression, a two-level realization requires three-three input gates, two four-input gates and one five-input gate. The expression for f' is factored to reduce the maximum number of gate inputs to three and, then, it is complemented.

Or $f' = abc' + a'bc + a'cd' + a'b'c'd + ab'cd$

i.e., $f' = abc' + a'c(b + d') + b'd(a'c' + ac)$

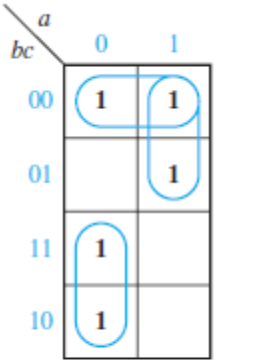
Or $f = [(a' + b' + c)][(a + c') + (b'd)][(b + d') + (a + c)(a' + c')]$



Example: *Realize the following functions using only two-input NAND gates and inverters.*

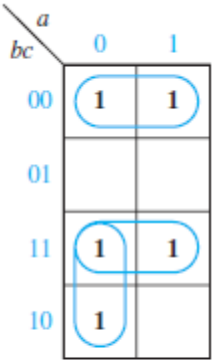
$$f1 = \Sigma m(0,2,3,4,5) \qquad f2 = \Sigma m(0,2,3,4,7) \qquad f3 = \Sigma m(1,2,6,7)$$

Solution:



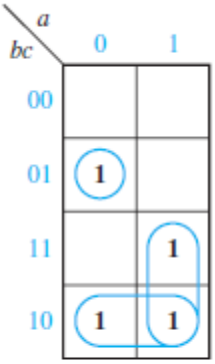
$f_1 = \Sigma\, m(0, 2, 3, 4, 5)$

$f_1 = b'c' + ab' + a'b$



$f_2 = \Sigma\, m(0, 2, 3, 4, 7)$

$f_2 = b'c' + bc + a'b$



$f_3 = \Sigma\, m(1, 2, 6, 7)$

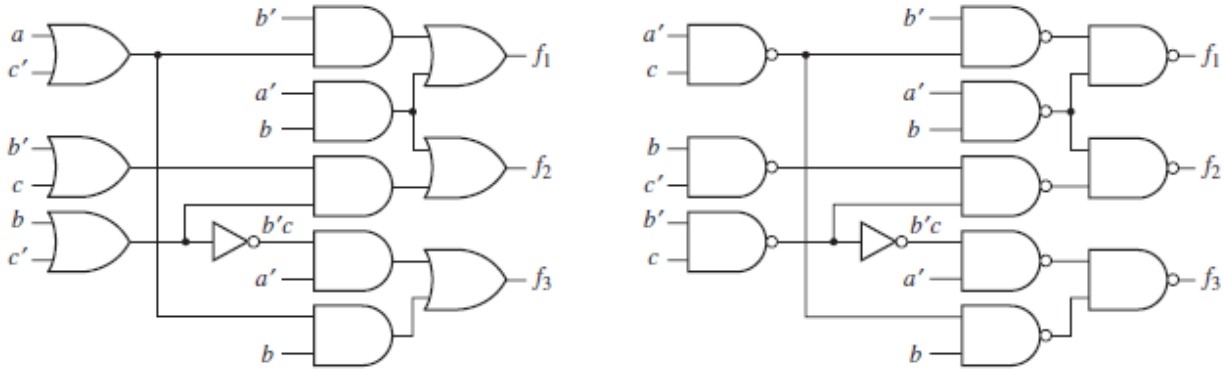
$f_3 = a'b'c + ab + bc'$

Each function requires a three-input OR gate; so we will factor to reduce the number of gate inputs:

$$f_1 = b'(\underline{a + c'}) + \underline{a'b}$$
$$f_2 = b(a' + c) + b'c' \qquad \text{or} \qquad f_2 = (b' + c)(b + c') + \underline{a'b}$$
$$f_3 = a'b'c + b(\underline{a + c'})$$

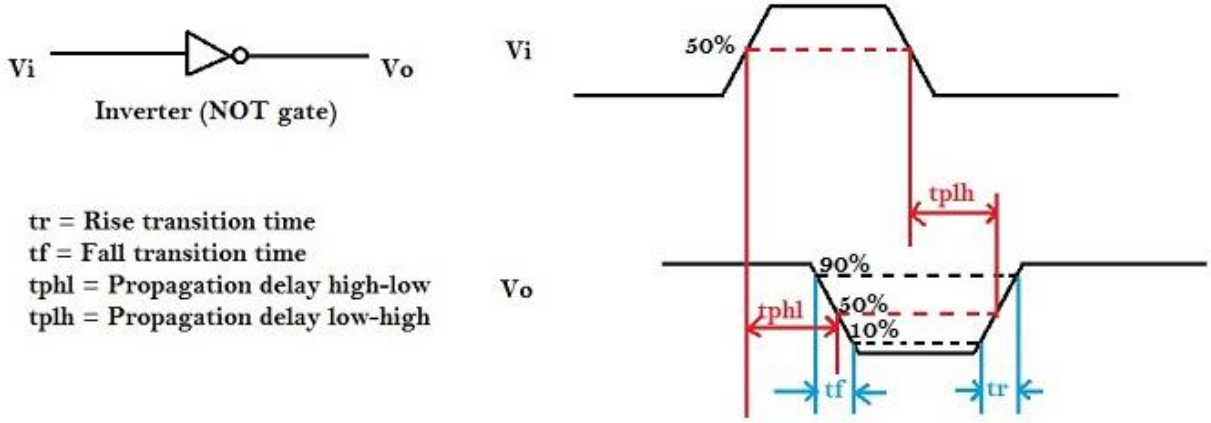
The second expression for f2 has a term common to f1, so we will choose the second expression. Also, we can eliminate the remaining three-input gate from f3 by noting that; $a'b'c = a'(b'c) = a'(b + c')'$.

The following Figures show the resulting circuits:



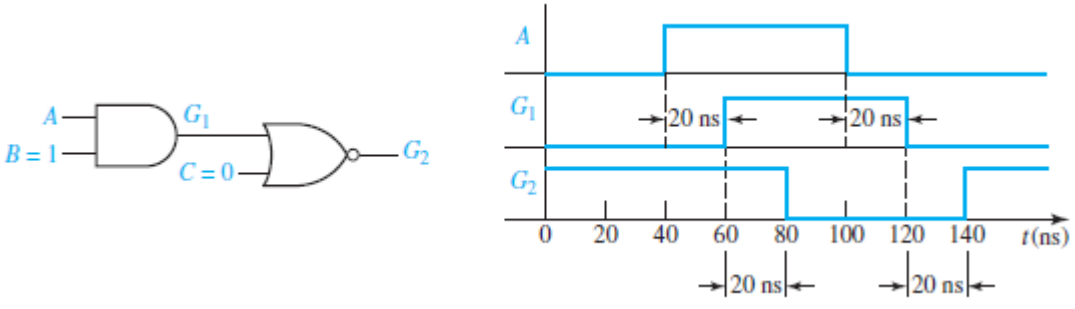
GATE DELAYS AND TIMING DIAGRAMS:

When the input to a logic gate is changed, the output will not change instantaneously. The gates take a finite time to react to a change in input, so that the change in the gate output is delayed with respect to the input change. The following Figure shows possible input and output waveforms for an inverter:



If the change in output is delayed by time, ε, with respect to the input, we say that, the gate has a propagation delay of ε. In practice, the propagation delay for a 0 to 1 output change may be different than the delay for a 1 to 0 change. In many cases these delays can be neglected. However, in the analysis of some types of sequential circuits, even short delays may be important.

The following Figure shows the timing diagram for a circuit with two gates:



Assume that, each gate has a propagation delay of 20 ns (nanoseconds). This timing diagram indicates what happens when gate inputs B and C are held at constant values 1 and 0, respectively, and input A is changed to 1 at $t = 40$ *ns* and then changed back to 0 at $t = 100$ *ns*. The output of gate G1 changes 20 ns after A changes, and the output of gate G2 changes 20 ns after G1 changes.

HAZARDS IN COMBINATIONAL LOGIC:

When the input to a combinational circuit changes, unwanted switching transients may appear in the output. These transients occur when different paths from input to output have different propagation delays.

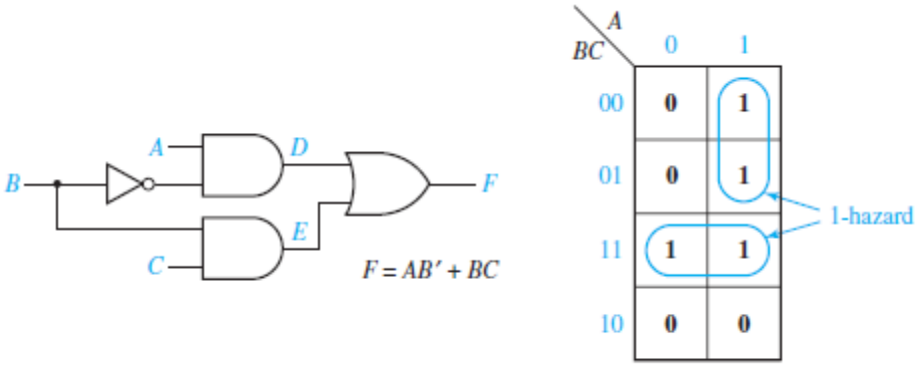
- If, in response to any single input change and for some combination of propagation delays, a circuit output may momentarily go to 0 when it should remain a constant 1, we say that the circuit has a ***static 1-hazard***.
- Similarly, if the output may momentarily go to 1 when it should remain a 0, we say that the circuit has a ***static 0-hazard***.
- If, when the output is supposed to change from 0 to 1 (or 1 to 0), the output may change three or more times, we say that the circuit has a ***dynamic hazard***.

The following Figure shows possible outputs from a circuit with hazards:

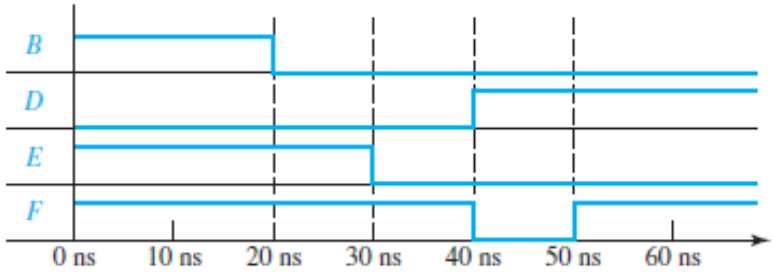
(a) Static 1-hazard (b) Static 0-hazard (c) Dynamic hazards

Note that hazards are properties of the circuit and are independent of the delays existing in the circuit.

The following Figure illustrates a circuit with a static 1-hazard.



(a) Circuit with a static 1-hazard



(b) Timing chart

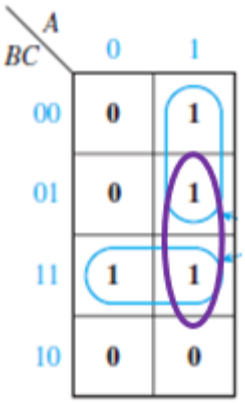
If $A = C = 1$, then $F = B + B' = 1$, so the F output should remain a constant 1 when B changes from 1 to 0. However, as shown in Figure (b), if each gate has a propagation delay of 10 ns, E will go to 0 before D goes to 1, resulting in a momentary 0 (a glitch caused by the 1-hazard) appearing at the output F. Note that right after B changes to 0, both the inverter input (B) and output (B') are 0 until the propagation delay has elapsed. During this period, both terms in the equation for F are 0, so F momentarily goes to 0.

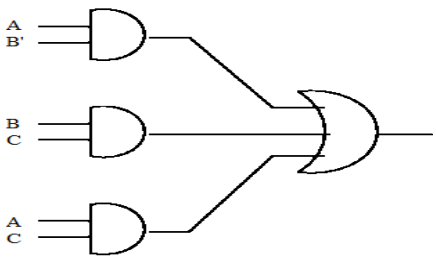
Detection of Static 1 Hazard: Hazards can be detected using a Karnaugh map (see above Figure). As seen on the map, no loop covers both minterms ABC and ABC'. So if A = C = 1 and B changes, both terms can momentarily go to 0, resulting in a glitch in F.

We can detect hazards in a two-level AND-OR circuit, using the following procedure:

1. Write down the sum-of-products expression for the circuit.
2. Plot each term on the map and loop it.
3. If any two adjacent 1's are not covered by the same loop, a 1-hazard exists for the transition between the two 1's. For an n-variable map, this transition occurs when one variable changes and the other $n - 1$ variables are held constant.

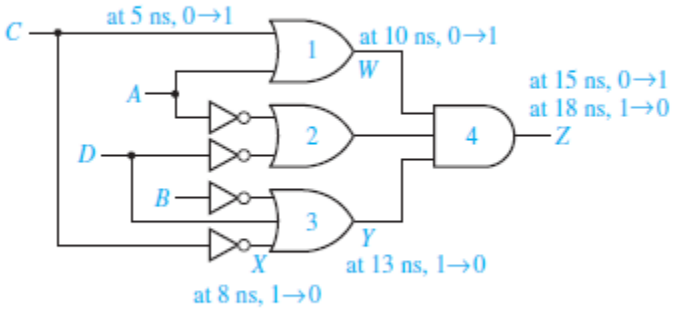
To Eliminate Static 1 Hazard: If we add a loop to the map of above Figure and, then, add the corresponding gate to the circuit (as shown in the following Figure), this eliminates the hazard. The term AC remains 1 while B is changing, so no glitch can appear in the output. Note that F is no longer a minimum sum of products.



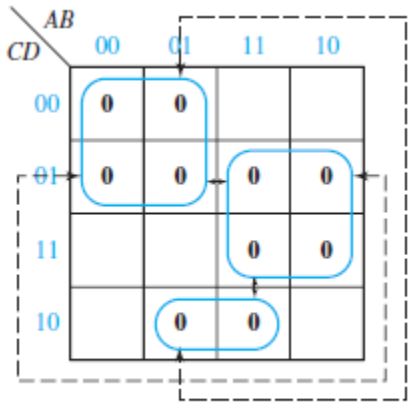


Detection of Static 0 Hazard: The following Figure shows a circuit with several 0-hazards. The product-of-sums representation for the circuit output is $F = (A + C)(A' + D')(B' + C' + D)$

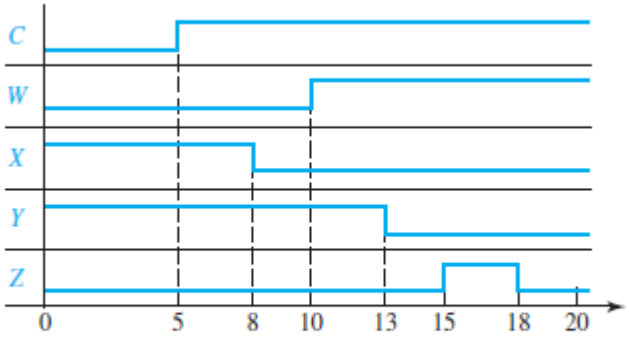
The Karnaugh map for this function (see following Figure) shows four pairs of adjacent *0*'s that are not covered by a common loop as indicated by the arrows. Each of these pairs corresponds to a 0-hazard. For example, when A = 0, B = 1, D = 0, and C changes from 0 to 1, a spike may appear at the Z output for some combination of gate delays. The timing diagram of (shown below) illustrates this assuming gate delays of 3 ns for each inverter, and of 5 ns for each AND gate and each OR gate.



(a) Circuit with a static 0-hazard

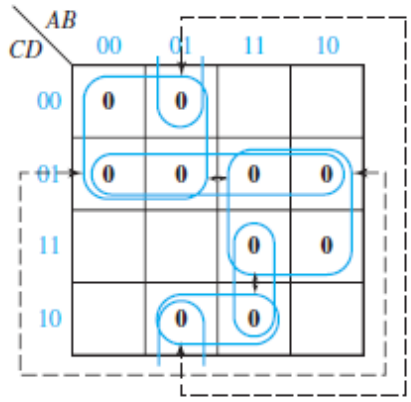


(b) Karnaugh map for circuit of (a)



(c) Timing diagram illustrating 0-hazard of (a)

To Eliminate Static 0 Hazard: We can eliminate the 0-hazards by looping additional prime implicants that cover the adjacent 0's that are not already covered by a common loop. This requires three additional loops as shown in shown in the following Figure. The resulting circuit requires seven gates in addition to the inverters, as given by below expression.

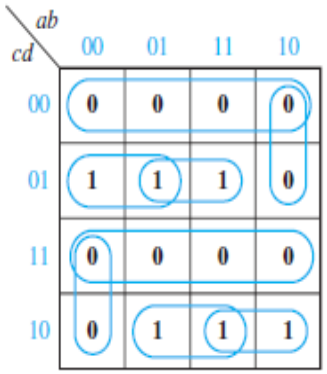


$$F = (A + C)(A' + D')(B' + C' + D)(C + D')(A + B' + D)(A' + B' + C')$$

Hazards in circuits with more than two levels can be determined by deriving either a SOP or POS expression for the circuit that represents a two-level circuit containing the same hazards as the original circuit. The SOP or POS expression is derived in the normal manner except that the complementation laws are not used, i.e., $xx' = 0$ and $x + x' = 1$ are not used. Consequently, the resulting SOP (POS) expression may contain products (sums) of the form $xx'\alpha$ $(x + x' + \beta)$. (α is a product of literals or it may be null; β is a sum of literals or it may be empty).

Dynamic Hazard: A dynamic hazard exists if there is a term of the form $xx'\alpha$ and two conditions are satisfied: (1) There are adjacent input combinations on the Karnaugh map differing in the value of x, with $\alpha = 1$ and with opposite function values, and (2) for these input combinations the change in x propagates over at least three paths through the circuit.

Consider the following expression and its Karnaugh map;



$f = a'c'd + bc'd + bcd' + acd'$

$$f = (c' + ad' + bd')(c + a'd + bd)$$
$$= cc' + acd' + bcd' + a'c'd + aa'dd' + a'bdd' + bc'd + abdd' + bdd'$$
$$= cc' + acd' + bcd' + a'c'd + aa'dd' + bc'd + bdd'$$

The circuit does not contain any static 1-hazards because each pair of adjacent 1's are covered by one of the product terms. Potentially, the terms cc' and bdd' may cause either static 0- or dynamic hazards or both; the first for c changing and the second for d changing.

To design a circuit which is free of static and dynamic hazards, the following procedure may be used:

1. Find a sum-of-products expression (F^t) for the output in which every pair of adjacent 1's is covered by a 1-term. (The sum of all prime implicants will always satisfy this condition.) A two-level AND-OR circuit based on this F^t will be free of 1-, 0-, and dynamic hazards.
2. If a different form of the circuit is desired, manipulate F^t to the desired form by simple factoring, DeMorgan's laws, etc. Treat each x_i and x_i' as independent variables to prevent introduction of hazards.

SIMULATION AND TESTING OF LOGIC CIRCUITS:

An important part of the logic design process is verifying that the final design is correct and debugging the design if necessary. Logic circuits may be tested either by actually building them or by simulating them on a computer. Simulation is generally easier, faster, and more economical. As logic circuits become more and more complex, it is very important to simulate a design before actually building it. This is particularly true when the design is built in integrated circuit form, because fabricating an integrated circuit may take a long time and correcting errors may be very expensive.

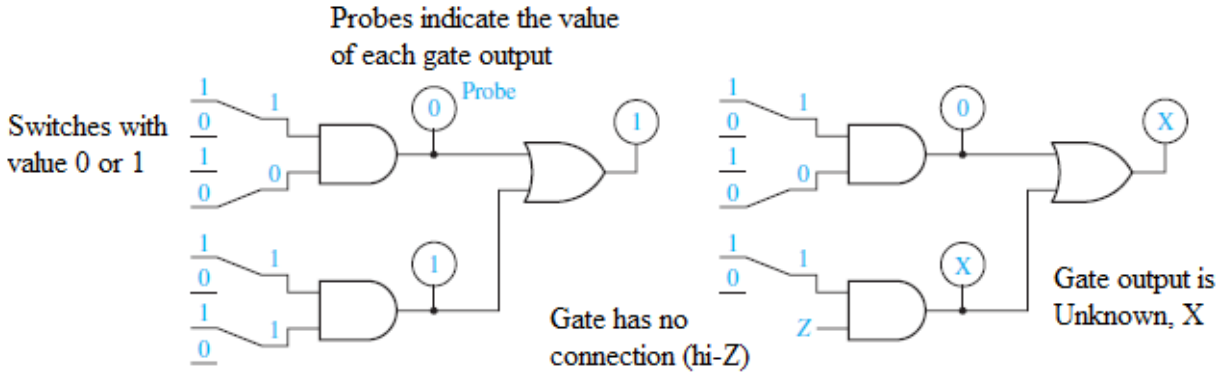
Simulation is done for following reasons: (1) Verification that the design is logically correct, (2) Verification that the timing of the logic signals is correct, and (3) Simulation of faulty components in the circuit as an aid to finding tests for the circuit.

A simple simulator for combinational logic works as follows:

1. The circuit inputs are applied to the first set of gates in the circuit, and the outputs of those gates are calculated.
2. The outputs of the gates which changed in the previous step are fed into the next level of gate inputs. If the input to any gate has changed, then the output of that gate is calculated.
3. Step 2 is repeated until no more changes in gate inputs occur. The circuit is then in a steady-state condition, and the outputs may be read.
4. Steps 1 through 3 are repeated every time a circuit input changes.

Four-Valued Logic Simulator: The two logic values, ***0*** and ***1***, are not sufficient for simulating logic circuits. At times, the value of a gate input or output may be ***unknown***, and we will represent this unknown value by ***X***. At other times we may have no logic signal at an input, as in the case of an open circuit when an input is not connected to any output. We use the logic value ***Z*** to represent an ***open circuit***, or ***high impedance*** (hi-Z) connection.

The following Figure shows a typical simulation screen on a personal computer.



(a) Simulation screen showing switches (b) Simulation screen with missing gate input

The following Table shows AND and OR functions for four-valued logic simulation. These functions are defined in a manner similar to the way real gates work.

•	0	1	X	Z
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

+	0	1	X	Z
0	0	1	X	X
1	1	1	1	1
X	X	1	X	X
Z	X	1	X	X

For an AND gate,

- If one of the inputs is *0*, the output is always *0* regardless of the other input
- If one input is *1* and the other input is *X* (we do not know what the other input is), then the output is *X* (we do not know what the output is)
- If one input is *1* and the other input is *Z* (it has no logic signal), then the output is *X* (we do not know what the hardware will do).

For an OR gate,

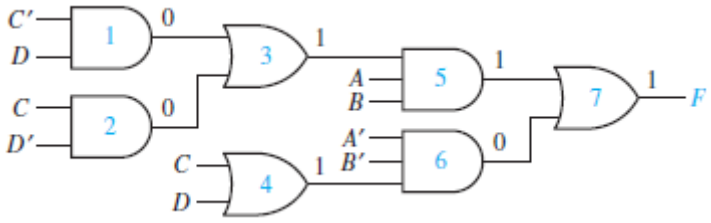
- If one of the inputs is *1*, the output is **1** regardless of the other input
- If one input is 0 and the other input is *X* or *Z*, the output is unknown.

If a circuit output is wrong for some set of input values, this may be due to several possible causes:

1. Incorrect design
2. Gates connected wrong
3. Wrong input signals to the circuit

   If the circuit is built in lab, other possible causes include
4. Defective gates
5. Defective connecting wires.

Example: The function $F = AB(C'D + CD') + (A'B'(C + D)$ is realized by the circuit shown below:

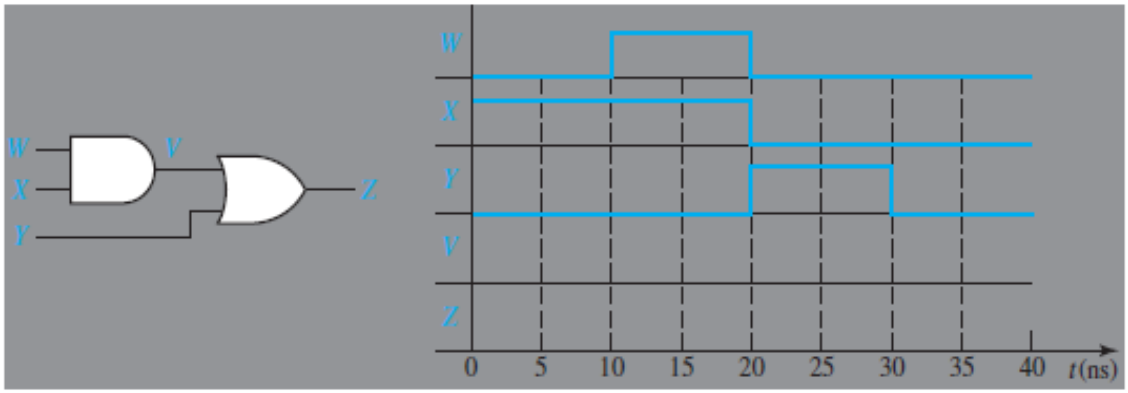


When a student builds the circuit in a lab, he finds that when A = B = C = D = 1, the output F has the wrong value, and that the gate outputs are as shown in above Figure. The reason for the incorrect value of F can be determined as follows:

1. The output of gate 7 (F) is wrong, but this wrong output is consistent with the inputs to gate 7, that is, 1 + 0 = 1.Therefore, one of the inputs to gate 7 must be wrong.
2. In order for gate 7 to have the correct output (F = 0), both inputs must be 0. Therefore, the output of gate 5 is wrong. However, the output of gate 5 is consistent with its inputs because 1.1.1 = 1. Therefore, one of the inputs to gate 5 must be wrong.
3. Either the output of gate 3 is wrong, or the A or B input to gate 5 is wrong. Because *C'D + CD'= 0*, the output of gate 3 is wrong.
4. The output of gate 3 is not consistent with the outputs of gates 1 and 2 because 0 + 0 ≠ 1. Therefore, either one of the inputs to gate 3 is connected wrong, or gate 3 is defective, or one of the input connections to gate 3 is defective.

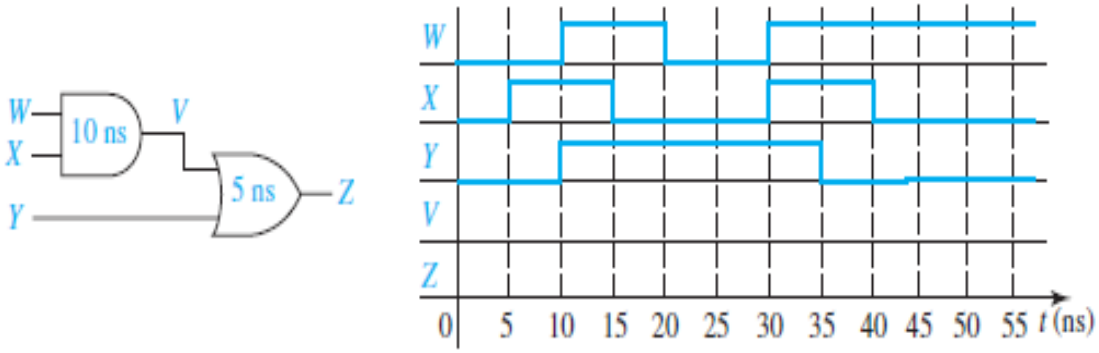
Problem: *Complete the timing diagram for the given circuit. Assume that both gates have a propagation delay of 5 ns.*

Solution:



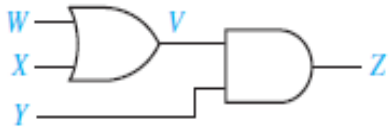
Problem: *Draw the timing diagram for V and Z for the circuit. Assume that the AND gate has a delay of 10 ns and the OR gate has a delay of 5 ns.*

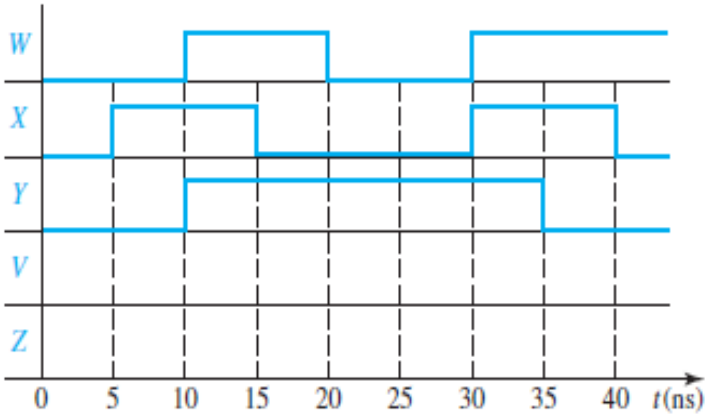
Solution:



Problem: *Complete the timing diagram for the given circuit. Assume that both gates have a propagation delay of 5 ns.*

Solution:





Problem: *Consider the logic function: F (A, B, C, D) = ∑m (0, 4, 5, 10, 11, 13, 14, 15).*

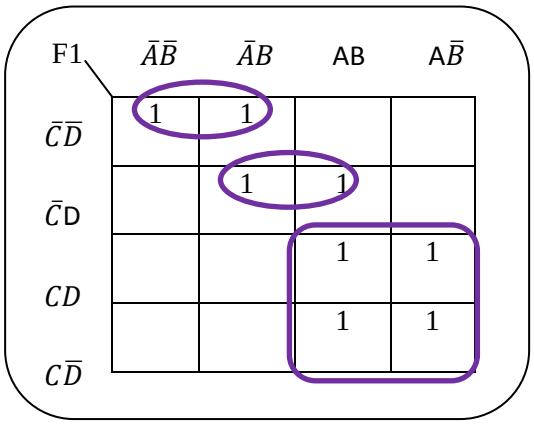
(a) Find two different minimum circuits which implement F using AND and OR Gates. Identify two hazards in each circuit.

(b) Find an AND-OR circuit for F which has no hazard.

(c) Find an OR-AND circuit for F which has no hazard.

Solution: Given, F (A, B, C, D) = ∑m (0, 4, 5, 10, 11, 13, 14, 15)

(a) K-Map1:



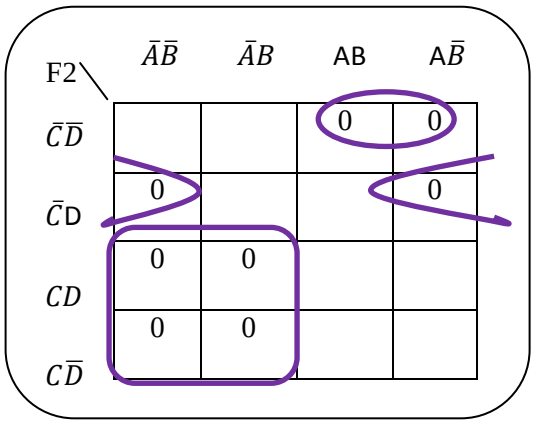
F1 =

Hazards: adjacent 1's are not covered by the same loop

1] A'BC'

2] ABD

K-Map2:



F2 =

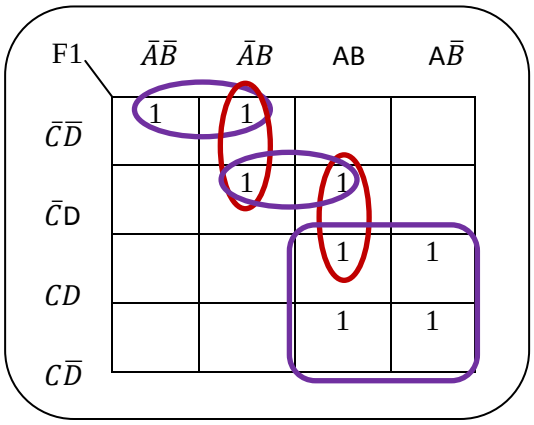
Hazards: adjacent 0's are not covered by the same loop

1] (A + B + D')

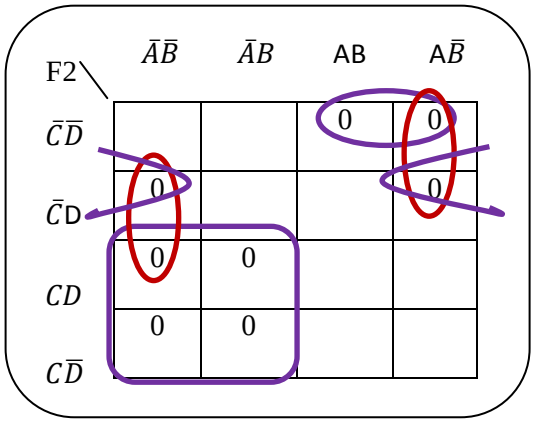
2] (A' + B + C')

(b) & (c) AND-OR & OR-AND circuit for F which has no hazard.

K-Map (AND-OR):



K-Map (OR-AND):



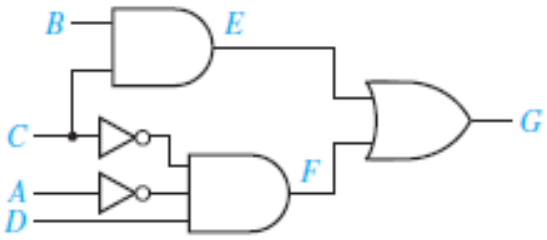
F1 =

Hazard free AND-OR Network:

F2 =

Hazard Free OR-AND Network:

Problem: *For the following circuit:*



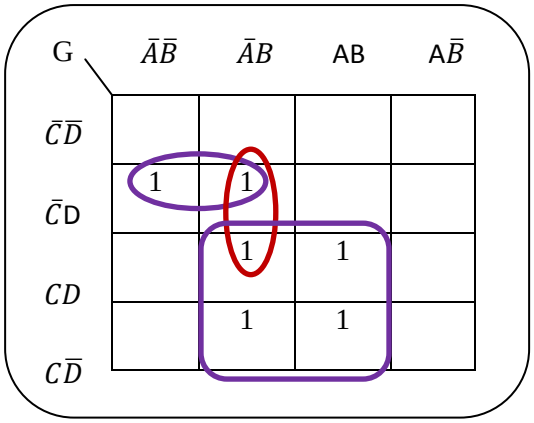
(a) Assume that the inverters have a delay of 1 ns and the other gates have a delay of 2 ns. Initially, A = 0 and B = C = D = 1, and C changes to 0 at time = 2 ns. Draw a timing diagram and identify the transient that occurs.

(b) Modify the circuit to eliminate the hazard.

Solution: (a) Timing Diagram:

(b) Expression for $G\ (A, B, C, D) = BC + A'C'D$

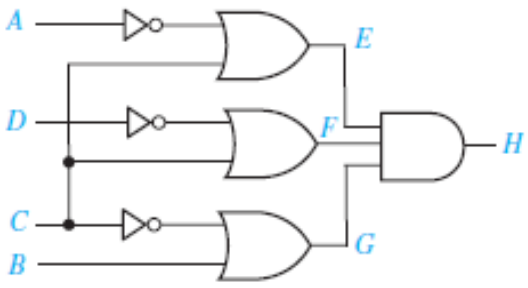
K-Map (AND-OR):



G =

Hazard free Circuit Diagram:

Problem: *For the following circuit:*



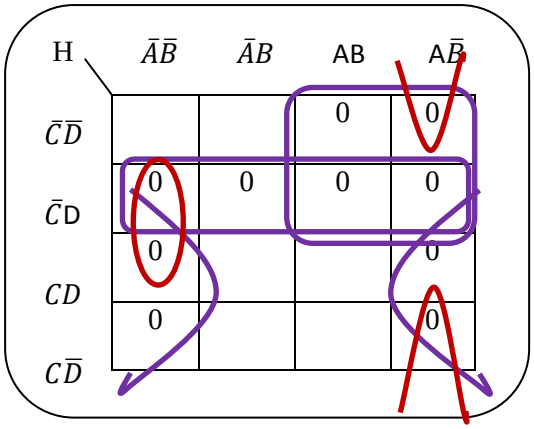
(a) *Assume the inverters have a delay of 1 ns and the other gates have a delay of 2 ns. Initially A = B = 0 and C = D = 1; C changes to 0 at time 2 ns. Draw a timing diagram showing the glitch corresponding to the hazard.*

(b) *Modify the circuit so that it is hazard free. (Leave the circuit as a two-level, OR-AND circuit.).*

Solution: (a) Timing Diagram:

(b) Expression for $H\ (A, B, C, D) = (A' + C)(C + D')(B + C')$

K-Map (OR-AND):



H =

Hazard free Circuit Diagram:

Homework:

1] F (A, B, C, D) = ∑ m (0, 2, 3, 5, 6, 7, 8, 9, 13, 15).

(a) Find three different minimum AND-OR circuits that implement F. Identify two hazards in each circuit. Then find an AND-OR circuit for F that has no hazards.

(b) There are two minimum OR-AND circuits for F; each has one hazard. Identify the hazard in each circuit, and then find an OR-AND circuit for F that has no hazards.

2] Consider the following logic function: F(A, B, C, D) = ∑ m(0, 2, 5, 6, 7, 8, 9, 12, 13, 15).

(a) Find two different minimum AND-OR circuits which implement F. Identify two hazards in each circuit. Then find an AND-OR circuit for F that has no hazards.

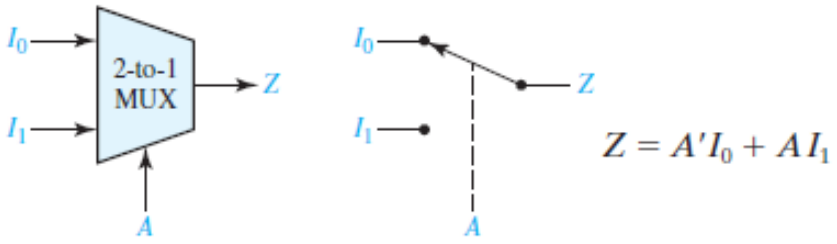
(b) The minimum OR-AND circuit for F has one hazard. Identify it, and then find an OR-AND circuit for F that has no hazards.

ANALOG AND DIGITAL ELECTRONICS

MULTIPLEXERS, DECODERS, AND PROGRAMMABLE LOGIC DEVICES

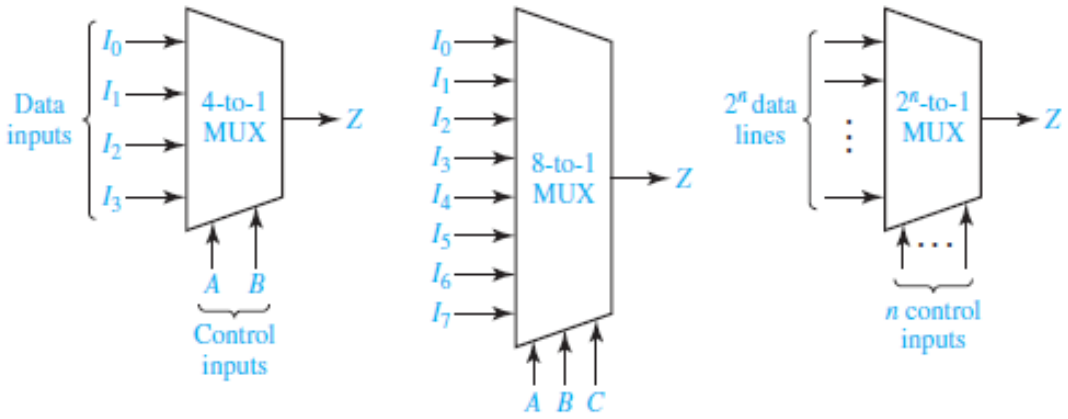
MULTIPLEXERS:

A *multiplexer* (or *data selector*, abbreviated as *MUX*) has a group of data inputs and a group of control inputs. The control inputs are used to select one of the data inputs and connect it to the output terminal. The following Figure shows a 2-to-1 multiplexer.



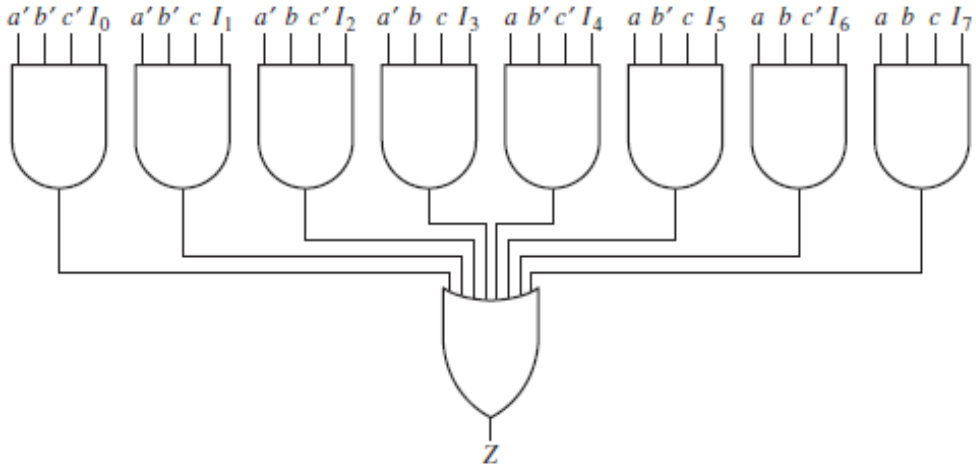
When the control input A is 0, the switch is in the upper position and the MUX output is $Z = I_0$; when A is 1, the switch is in the lower position and the MUX output is $Z = I_1$. In other words, a MUX acts like a switch that selects one of the data inputs (I_0 or I_1) and transmits it to the output. The logic equation for the 2-to-1 MUX is therefore: $Z = A' I_0 + AI_1$

The following Figure shows diagrams for a 4-to-1 multiplexer, 8-to-1 multiplexer, and 2^n-to-1 multiplexer.



The 4-to-1 MUX acts like a four-position switch that transmits one of the four inputs to the output. Two control inputs (A and B) are needed to select one of the four inputs. If the control inputs are AB = 00, the output is I_0; similarly, the control inputs 01, 10, and 11 give outputs of I_1, I_2, and I_3, respectively. The 4-to-1 multiplexer is described by the equation: $Z = A' B' I_0 + A' B I_1 + A B' I_2 + A B I_3$

Similarly, the 8-to-1 MUX selects one of eight data inputs using three control inputs.



It is described by the equation: $Z = A'B'C'I_0 + A'B'CI_1 + A'BC'I_2 + A'BCI_3 + AB'C'I_4 + AB'CI_5 + ABC'I_6 + ABCI_7$. Multiplexers can also have an additional input called an *enable* input.

If the OR gate in the above Figure is replaced by a NOR gate, then the 8-to-1 MUX inverts the selected input. To distinguish between these two types of multiplexers, we will say that the multiplexers without the inversion have *active high* outputs, and the multiplexers with the inversion have *active low* outputs.

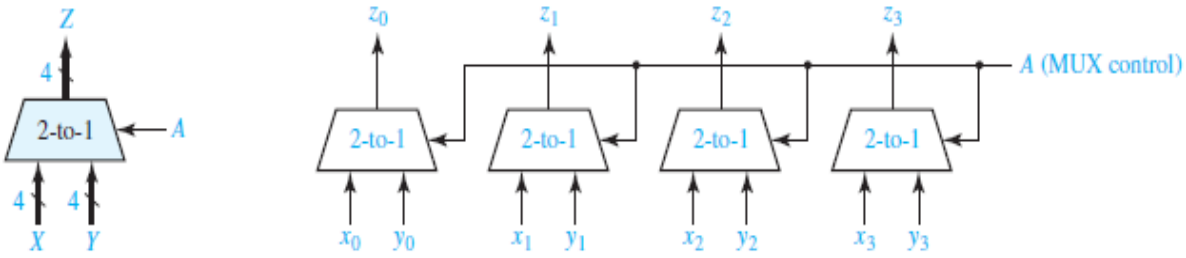
In general, a multiplexer with *n* control inputs can be used to select any one of 2^n data inputs. The general equation for the output of a MUX with *n* control inputs and 2^n data inputs is:

$$Z = \sum_{k=0}^{2^n-1} m_k I_k$$

Where m_k is a minterm of the *n* control variables and I_k is the corresponding data input.

Multiplexers are frequently used in digital system design to select the data which is to be processed or stored.

The following Figure shows how a quadruple 2-to-1 MUX is used to select one of two 4-bit data words. If the control A = 0, the values of x0, x1, x2,and x3 will appear at the z0, z1, z2, and z3 outputs; if A = 1, the values of y0, y1, y2, and y3 will appear at the outputs.

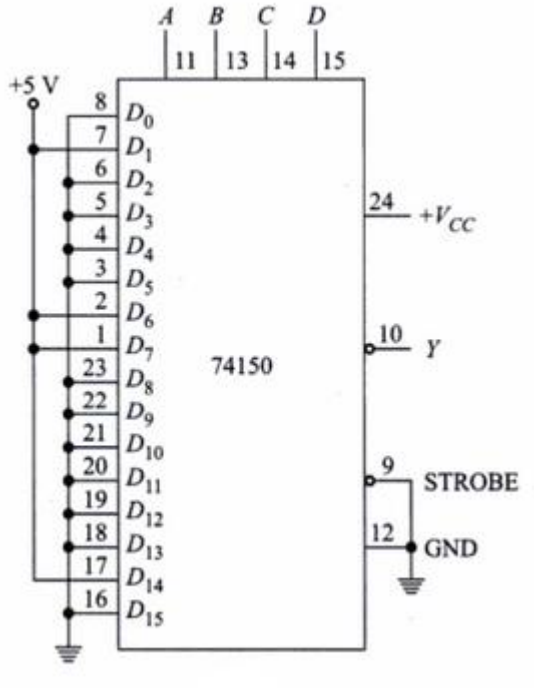


Multiplexer Logic: A digital design usually begins with a truth table. The problem is to come up with a logic circuit that has the same truth table. We have two standard methods for implementing a truth table – the SOP and the POS solution. The third method is the *multiplexer solution.*

Problem: *Implement Y (A, B, C, D) = ∑m (0, 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 15) using 16-to-1 multiplexer (IC 74150) & 8-to-1 multiplexer.*

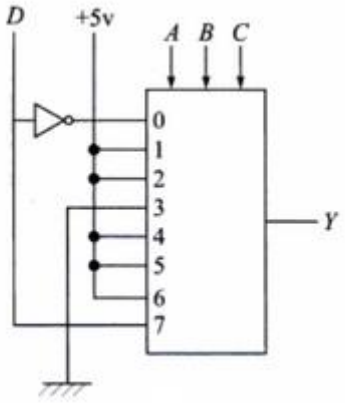
Solution: Notice that, output is an active low signal in IC 74150.

A	B	C	D	Y	8-to-1 MUX Data Inputs
0	0	0	0	1	$\bar{D}$
0	0	0	1	0	
0	0	1	0	1	1
0	0	1	1	1	
0	1	0	0	1	1
0	1	0	1	1	
0	1	1	0	0	0
0	1	1	1	0	
1	0	0	0	1	1
1	0	0	1	1	
1	0	1	0	1	1
1	0	1	1	1	
1	1	0	0	1	1
1	1	0	1	1	
1	1	1	0	0	D
1	1	1	1	1	



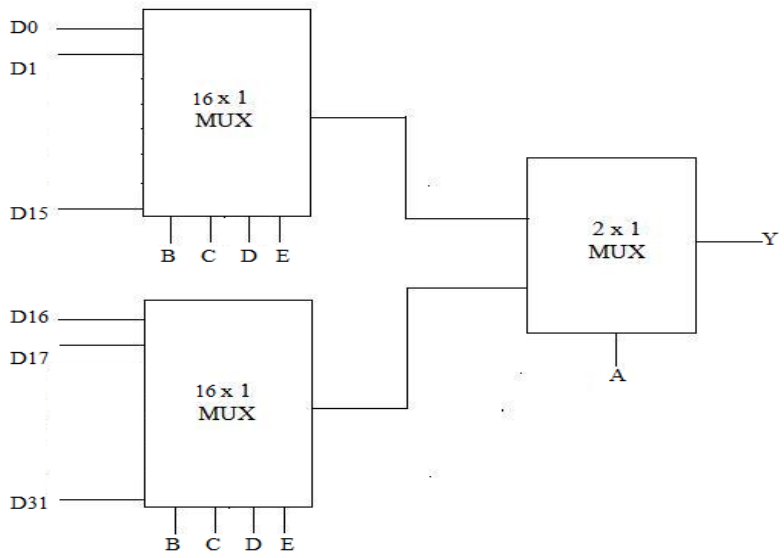
We follow a procedure that is similar to the one that we adopted in Entered Variable Map method to implement Y using 8-to-1 MUX.

A	B	C	8-to-1 MUX Data Inputs
0	0	0	$\bar{D}$
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	D



Problem: *Design a 32-to-1 multiplexer using two 16-to-1 multiplexer and one 2-to-1 multiplexer.*

Solution: The circuit diagram is shown in the following Fig. A 32-to-1 multiplexer required 5 ($\log_2 32$) select lines (say, ABCDE). The lower four 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 the 5th select line (A).



Problem: *Realize* $Y = \bar{A}B + B\bar{C} + ABC$ *using an 8-to-1 multiplexer. Also, realize the same with a 4-to-1 multiplexer.*

Solution: Given, $Y = \bar{A}B + \bar{B}\bar{C} + ABC$

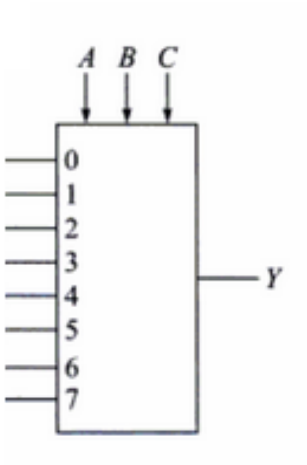
$Y = \bar{A}B(\bar{C} + C) + \bar{B}\bar{C}(\bar{A} + A) + ABC$

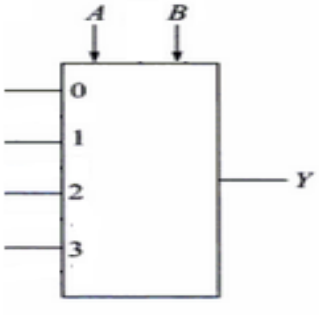
$Y = \bar{A}B\bar{C} + \bar{A}BC + \bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + ABC$

Y = ∑m (0, 2, 3, 4, 7).

Hence, to generate the given logic function, using *8-to-1 multiplexer*,

we find $D_0 = D_2 = D_3 = D_4 = D_7 = 1$ and $D_1 = D_5 = D_6 = 0$.





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

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

Hence, for a *4-to-1 multiplexer*,

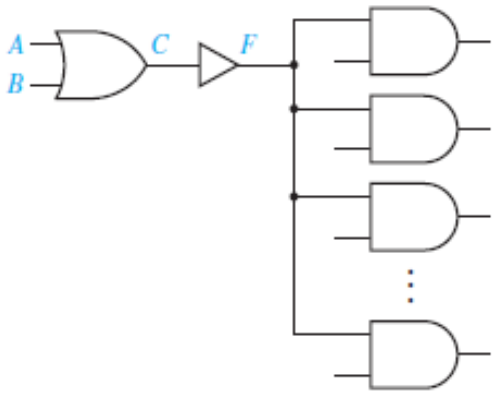
we find $D_0 = C'$, $D_1 = 1$, $D_2 = C'$, and $D_3 = C$ generates the given function.

Alternate Method:

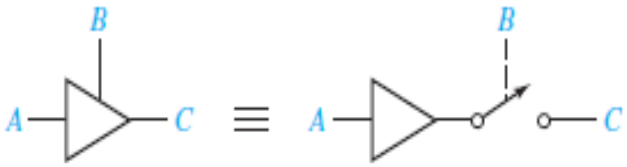
A	B	C	8-to-1 MUX Data Inputs	4-to-1 MUX Data Inputs
0	0	0	1 = D_0	$\bar{C}$ = D_0
0	0	1	0 = D_1	
0	1	0	1 = D_2	1 = D_1
0	1	1	1 = D_3	
1	0	0	1 = D_4	$\bar{C}$ = D_2
1	0	1	0 = D_5	
1	1	0	0 = D_6	C = D_3
1	1	1	1 = D_7	

THREE STATE BUFFERS:

A gate output can only be connected to a limited number of other device inputs without degrading the performance of a digital system. A simple buffer may be used to increase the driving capability of a gate output. The following Figure shows a buffer connected between a gate output and several gate inputs. Because no bubble is present at the buffer output, this is a non-inverting buffer, and the logic values of the buffer input and output are the same, that is, $F = C$.

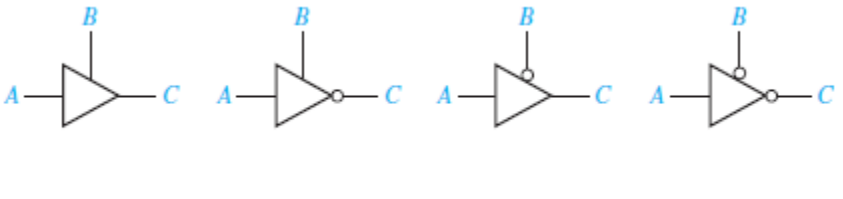


Normally, a logic circuit will not operate correctly if the outputs of two or more gates or other logic devices are directly connected to each other. Use of three-state logic permits the outputs of two or more gates or other logic devices to be connected together. The following Figure shows a three-state buffer and its logical equivalent.



When the enable input B is 1, the output C equals A; when B is 0, the output C acts like an open circuit. In other words, when B is 0, the output C is effectively disconnected from the buffer output so that no current can flow. This is often referred to as a Hi-Z (high-impedance) state of the output because the circuit offers a very high resistance or impedance to the flow of current. Three-state buffers are also called *tri-state buffers*.

The following Figure shows the truth tables for four types of three-state buffers.



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

(a)

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

(b)

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

(c)

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

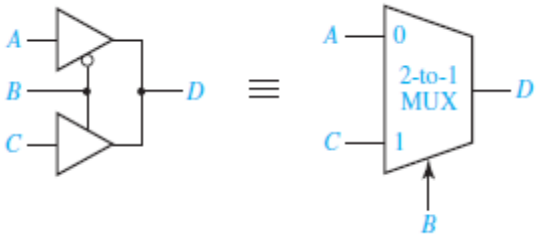
(d)

In Figures (a) and (b), the enable input B is not inverted, so the buffer output is enabled when B = 1 and disabled when B = 0. That is, the buffer operates normally when B = 1, and the buffer output is effectively an open circuit when B = 0. We use the symbol *Z* to represent this high-impedance state.

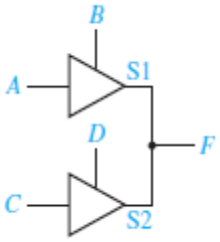
In Figure (b), the buffer output is inverted so that $C = A'$ when the buffer is enabled.

The buffers in Figures (c) and (d) operate the same as in (a) and (b) except that the enable input is inverted, so the buffer is enabled when B = 0.

In the following Figure, the outputs of two three-state buffers are tied together. When B = 0, the top buffer is enabled, so that D = A; when B = 1, the lower buffer is enabled, so that D = C. Therefore, $D = B'A + BC$. This is logically equivalent to using a 2-to-1 multiplexer to select the A input when B = 0 and the C input when B = 1.

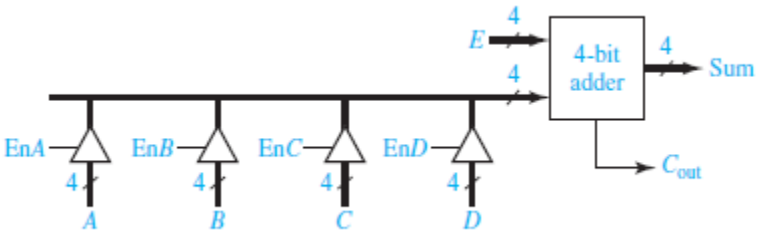


When we connect two three-state buffer outputs together, as shown in the following Figure, if one of the buffers is disabled (output = Z), the combined output F is the same as the other buffer output. If both buffers are disabled, the output is Z. If both buffers are enabled, a conflict can occur. If A = 0 and C = 1, we do not know what the hardware will do, so the F output is unknown (X). If one of the buffer inputs is unknown, the F output will also be unknown. The table in the following Figure summarizes the operation of the circuit. S1 and S2 represent the outputs the two buffers would have if they were not connected together. When a bus is driven by three-state buffers, we call it a three-state bus. The signals on this bus can have values of 0, 1, Z, and perhaps X.

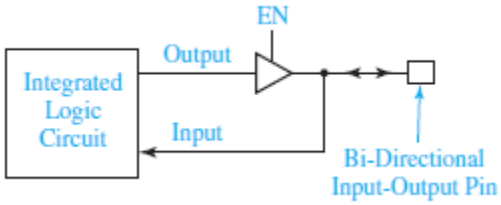


	S2			
S1	X	0	1	Z
X	X	X	X	X
0	X	0	X	0
1	X	X	1	1
Z	X	0	1	Z

A multiplexer may be used to select one of several sources to drive a device input. For example, if an adder input must come from four different sources; a 4-to-1 MUX may be used to select one of the four sources. An alternative is to set up a three-state bus, using three-state buffers to select one of the sources (see the following Figure). In this circuit, each buffer symbol actually represents four three-state buffers that have a common enable signal.

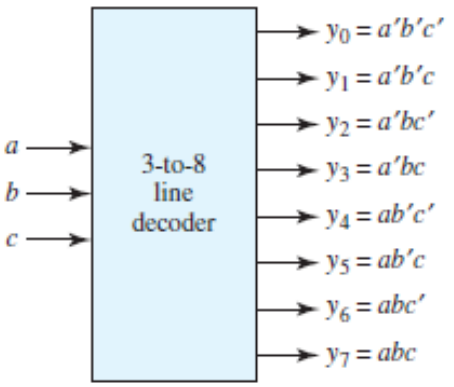


Integrated circuits are often designed using bi-directional pins for input and output. Bi-directional means that the same pin can be used as an input pin and as an output pin, but not both at the same time. To accomplish this, the circuit output is connected to the pin through a three-state buffer, as shown in the following Figure. When the buffer is enabled, the pin is driven with the output signal. When the buffer is disabled, an external source can drive the input pin.



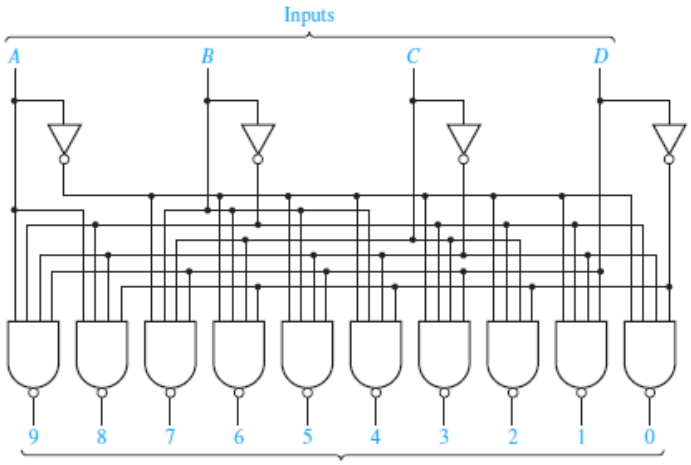
DECODERS AND ENCODERS:

The ***decoder*** is another commonly used type of integrated circuit. The following Figure shows the diagram and truth table for a 3-to-8 line decoder. This decoder generates all of the minterms of the three input variables. Exactly one of the output lines will be 1 for each combination of the values of the input variables.

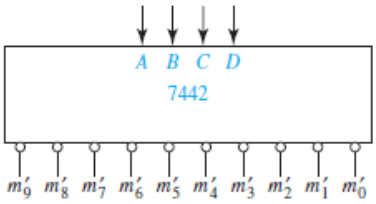


a	b	c	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

The following Figure illustrates a 4-to-10 decoder. This decoder has inverted outputs (indicated by the small circles). For each combination of the values of the inputs, exactly one of the output lines will be 0. When a binary-coded-decimal digit is used as an input to this decoder, one of the output lines will go low to indicate which of the 10 decimal digits is present.



(a) Logic diagram



(b) Block diagram

BCD Input				Decimal Output									
A	B	C	D	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	1	1	1	1	1	1	1	1	1
0	0	0	1	1	0	1	1	1	1	1	1	1	1
0	0	1	0	1	1	0	1	1	1	1	1	1	1
0	0	1	1	1	1	1	0	1	1	1	1	1	1
0	1	0	0	1	1	1	1	0	1	1	1	1	1
0	1	0	1	1	1	1	1	1	0	1	1	1	1
0	1	1	0	1	1	1	1	1	1	0	1	1	1
0	1	1	1	1	1	1	1	1	1	1	0	1	1
1	0	0	0	1	1	1	1	1	1	1	1	0	1
1	0	0	1	1	1	1	1	1	1	1	1	1	0
1	0	1	0	1	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	1	1	1	1
1	1	0	0	1	1	1	1	1	1	1	1	1	1
1	1	0	1	1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1

(c) Truth Table

In general, an n-to-2^n line decoder generates all 2^n minterms (or maxterms) of the n input variables. The outputs are defined by the equations:

$$y_i = m_i, \qquad i = 0 \text{ to } 2^n - 1 \text{ (non-inverted outputs)}$$

or

$$y_i = m_i' = M_i, \qquad i = 0 \text{ to } 2^n - 1 \text{ (inverted outputs)}$$

where m_i is a minterm of the n input variables and Mi is a maxterm.

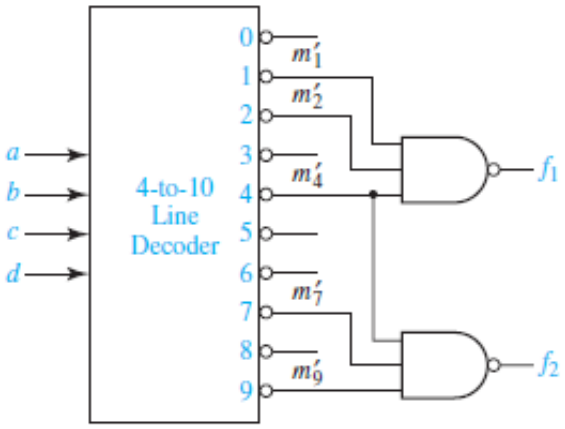
Example: *Realize the following functions using a 4-to-10 decoder.*

f1 (a, b, c, d) = m1 + m2+ m4 and f2 (a, b, c, d) = m4 + m7 +m9

Solution: An n-input decoder generates all of the minterms of n variables. Hence, n-variable functions can be realized by ORing together selected minterm outputs from a decoder.

Rewriting given f1 and f2; we have: f1 = (m1' m2'm4')' and f2 = (m4' m7'm9')'

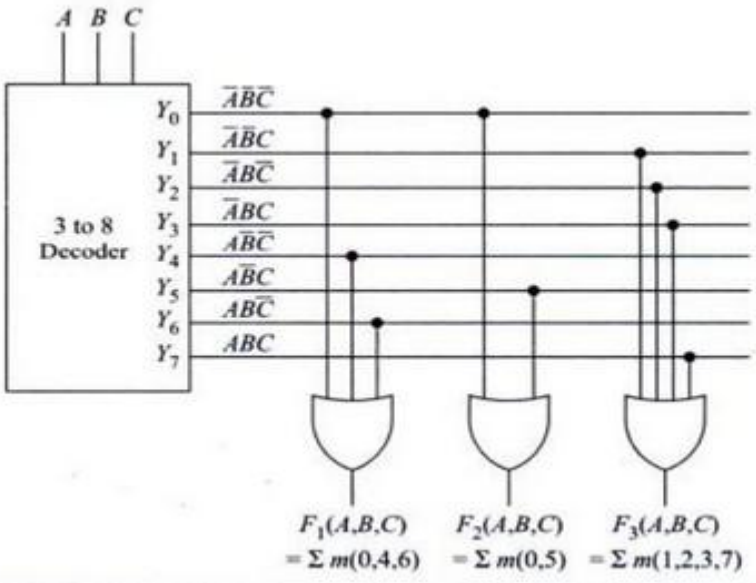
Now, f1 and f2 can be generated using NAND gates, as shown in the following Figure.



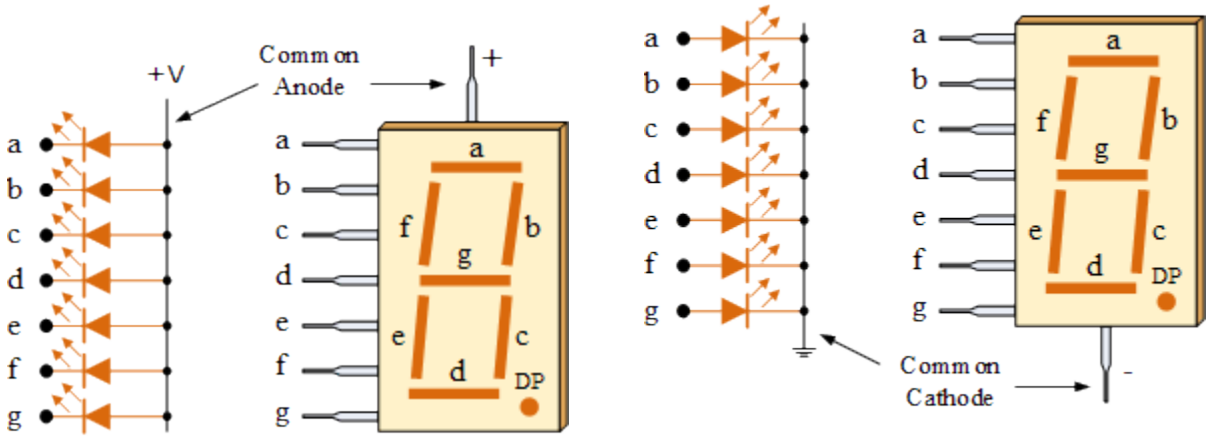
Problem: *Show how using a 3-to8 decoder and multi-input OR gates following Boolean expression can be realized simultaneously.*

$F_1 (A, B, C) = \sum m (0, 4, 6)$ $F_2 (A, B, C) = \sum m (0, 5)$ $F_3 (A, B, C) = \sum m (1, 2, 3, 7)$.

Solution: Since, at the decoder output, we get all the min-terms, we use them as shown in the following Fig, to get the required Boolean expression:



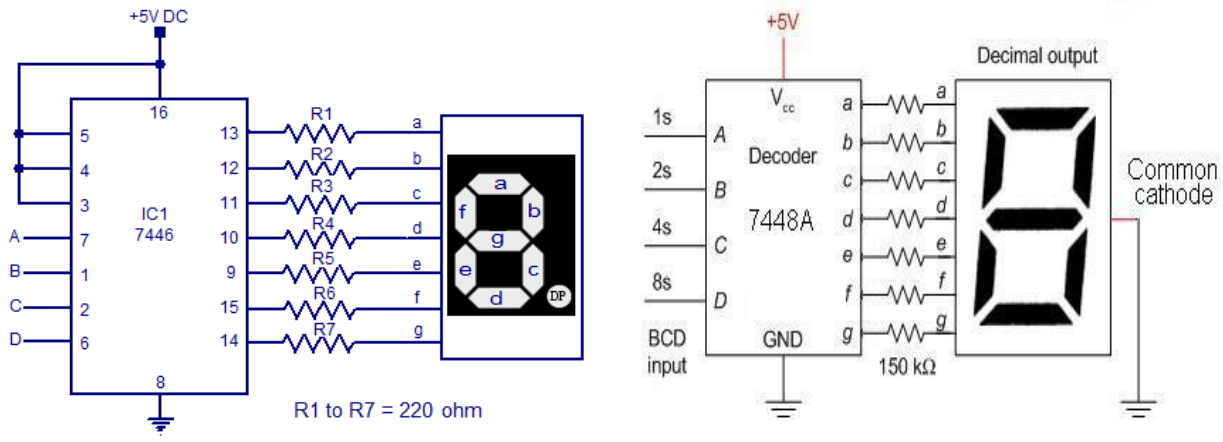
Seven-Segment Decoders: The following Fig shows a *seven-segment indicator*, i.e. seven LEDs labeled *a* through *g* (actually, eight LEDs labeled through *a* through *h*). By forward biasing the LEDs, we can display the digits 0 through 9. For example, to display the digit 0, we need to light-up the segments *a*, *b*, *c*, *d*, *e*, and *f*. Similarly, to light-up the digit 5, we need segments *a*, *c*, *d*, *f*, and *g*.



Seven-Segment Indicator: Common Anode Type & Common Cathode Type

Seven-segment indicators may be *common-anode* type; where all anodes are connected together (as shown above) or *common-cathode* type; where all cathodes are connected together (as shown above).

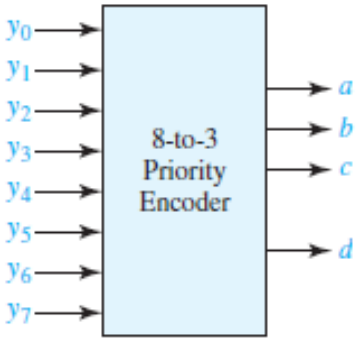
The 7446 & The 7448: A *seven-segment decoder-driver* is an IC decoder that can be used to drive a seven-segment indicator. There are two types of decoder-drivers, corresponding to common-anode (IC 7446) and common cathode (IC 7448) indicators. Each decoder driver has 4 input pins (the BCD input) and 7 output pins (*a* through *h* segments), as shown in the following Fig.



7446 Decoder-driver & 7448 Decoder-driver

The logic circuits inside 7446 / 7448 convert the BCD input to the required output. For Example, if the BCD input is 0111, the internal logic of the 7446 / 7448 will force segments *a*, *b*, and *c* to conduct. As a result, digit 7 will appear on the seven-segment indicator.

An ***encoder*** (*converts an active input signal to a coded output signal*) performs the inverse function of a decoder. The following Figure shows a *8-to-3 priority encoder* with inputs y0 through y7. If input yi is 1 and the other inputs are 0, then the abc outputs represent a binary number equal to i. For example, if y3 = 1, then abc = 011.

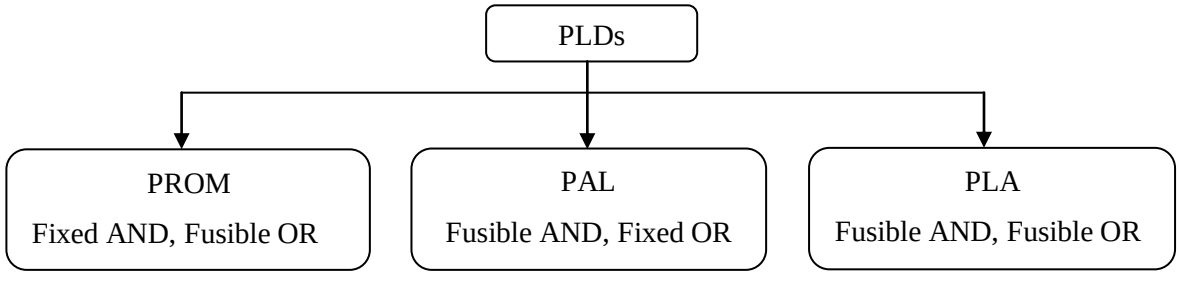


y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	a	b	c	d
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
X	1	0	0	0	0	0	0	0	0	1	1
X	X	1	0	0	0	0	0	0	1	0	1
X	X	X	1	0	0	0	0	0	1	1	1
X	X	X	X	1	0	0	0	1	0	0	1
X	X	X	X	X	1	0	0	1	0	1	1
X	X	X	X	X	X	1	0	1	1	0	1
X	X	X	X	X	X	X	1	1	1	1	1

If more than one input is 1 at the same time, the output can be defined using a priority scheme. The truth table in the above Figure uses the following scheme: If more than one input is 1, the highest numbered input determines the output. For example, if inputs y1, y4, and y5 are 1, the output is abc = 101.The X's in the table are don't-cares; for example, if y5 is 1, we do not care what inputs y0 through y4 are. Output d is 1 if any input is 1, otherwise, d is 0. This signal is needed to distinguish the case of all 0 inputs from the case where only y0 is 1.

PROGRAMMABLE LOGIC DEVICES (PLDs):

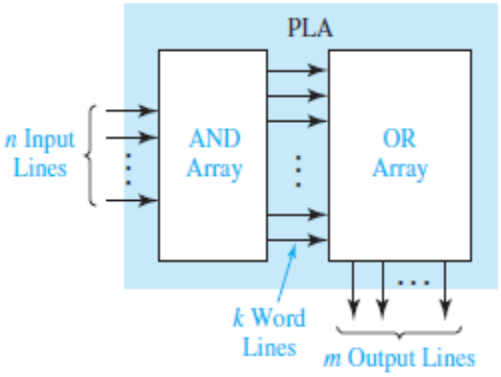
A *programmable logic device* (or *PLD*) is a general name for a digital integrated circuit capable of being programmed to provide a variety of different logic functions. *PLD*s are electronic components, used to build reconfigurable digital circuits. Programmable Read Only Memory (PROM), Programmable Array Logic (PAL), and Programmable Logic Array (PLA) are included in the general classification.



General Classification of PLDs

Programmable Logic Arrays (PLA):

A *PLA* with *n* inputs and *m* outputs (see the following Figure) can realize *m* functions of *n* variables. In PLA, the product terms of the input variables is realized by an AND array; and the OR array ORs together the product terms needed to form the output functions. Hence, a PLA implements a sum-of-products expression.



Example: *Realize the following functions using PLA:*

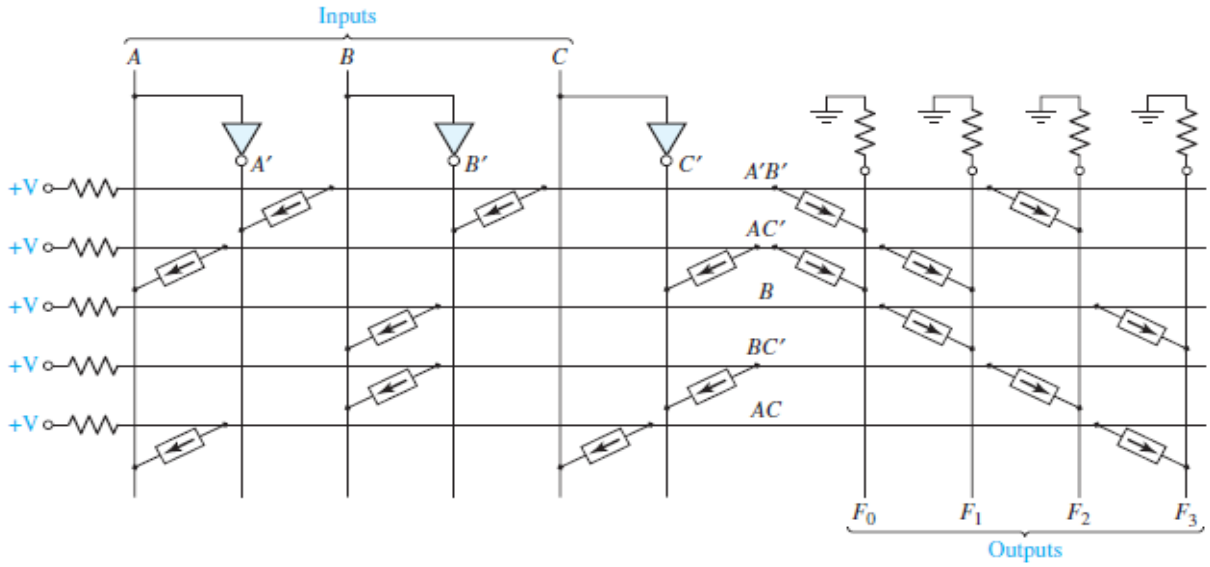
$F0 = \sum m\ (0, 1, 4, 6) = A'B' + AC'$ $\qquad$ $F1 = \sum m\ (2, 3, 4, 6, 7) = B + AC'$

$F2 = \sum m\ (0, 1, 2, 6) = A'B' + BC'$ $\qquad$ $F3 = \sum m\ (2, 3, 5, 6, 7) = AC + B$

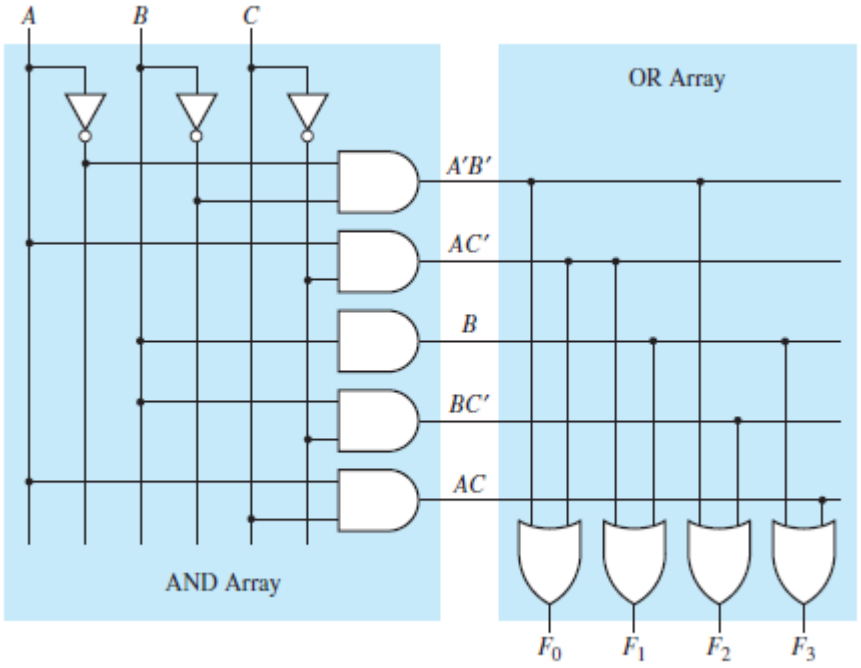
Solution: The following Figure shows a PLA which realizes the said functions.

Product terms are formed in the AND array by connecting switching elements at appropriate points in the array. For example, to form A'B', switching elements are used to connect the first word line with the A' and B' lines.

Switching elements are connected in the OR array to select the product terms needed for the output functions. For example, because F0 = A'B' + AC', switching elements are used to connect the A'B' and AC' lines to the F0 line.



The connections in the AND and OR arrays of this PLA make it equivalent to the AND-OR array shown in the following Figure.



The contents of a PLA can be specified by a PLA table. The following Table specifies the PLA shown in the above Figure. The input side of the table specifies the product terms. The symbols 0, l, and – indicate whether a variable is complemented, not complemented, or not present in the corresponding product term. The output side of the table specifies which product terms appear in each output function. A 1 or 0 indicates whether a given product term is present or not present in the corresponding output function. Thus, the first row of Table indicates that the term A'B' is present in output functions F0 and F2, and the second row indicates that AC' is present in F0 and F1.

Product Term	Inputs A B C	Outputs F_0 F_1 F_2 F_3	
$A'B'$	0 0 –	1 0 1 0	$F_0 = A'B' + AC'$
AC'	1 – 0	1 1 0 0	$F_1 = AC' + B$
B	– 1 –	0 1 0 1	$F_2 = A'B' + BC'$
BC'	– 1 0	0 0 1 0	$F_3 = B + AC$
AC	1 – 1	0 0 0 1	

Example: *Realize the following functions using PLA:*

$$f1 = a'bd + abd + ab'c' + b'c$$

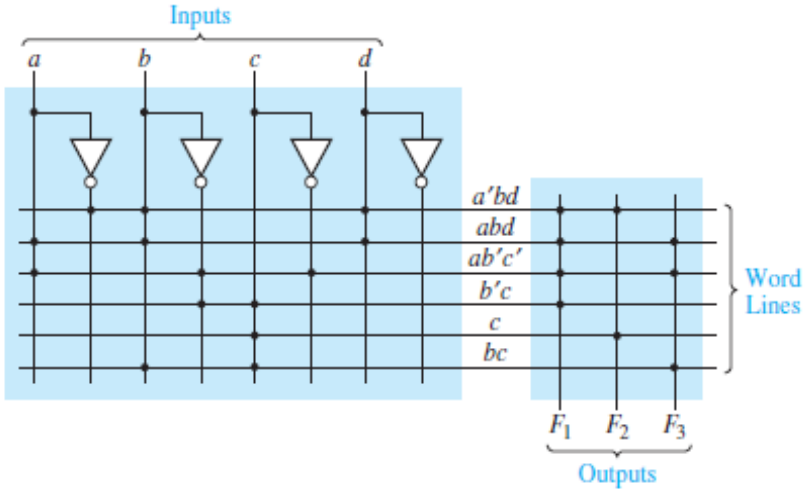
$$f2 = c + a'bd$$

$$f3 = bc + ab'c' \; abd$$

Solution: Based on the given expressions, we can construct a PLA table (see Figure (a)), with one row for each distinct product term. Figure (b) shows the corresponding PLA structure, which has four inputs, six product terms, and three outputs. A dot at the intersection of a word line and an input or output line indicates the presence of a switching element in the array.

a b c d	f_1 f_2 f_3
0 1 – 1	1 1 0
1 1 – 1	1 0 1
1 0 0 –	1 0 1
– 0 1 –	1 0 0
– – 1 –	0 1 0
– 1 1 –	0 0 1

(a) PLA table

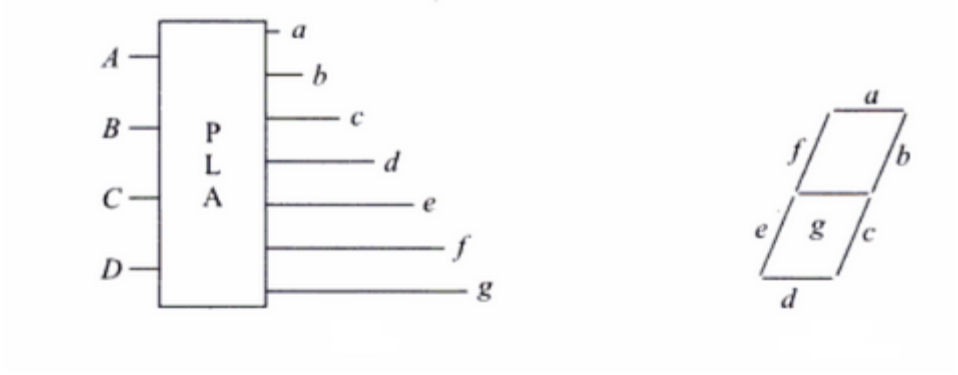


(b) PLA structure

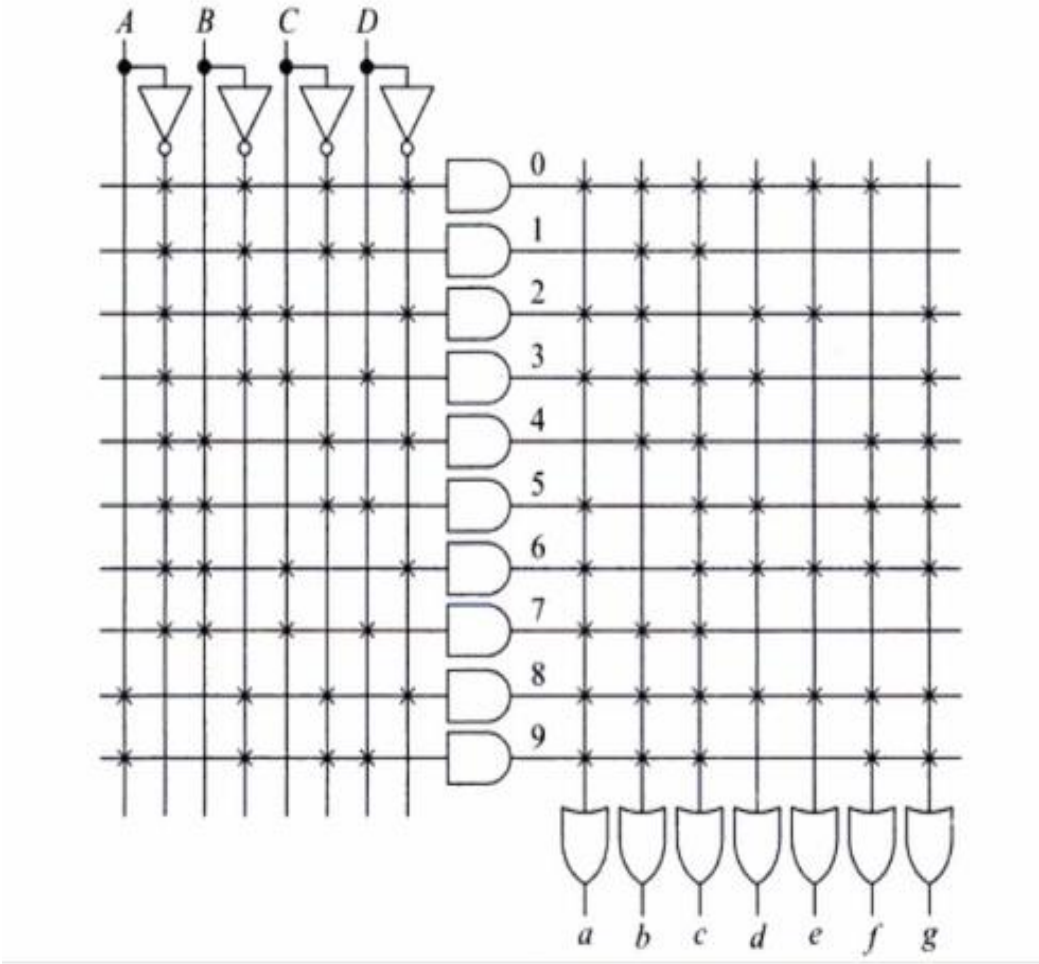
NOTE: Mask-programmable and field-programmable PLAs are available. The mask-programmable type is programmed at the time of manufacture. The field-programmable logic array (FPLA) has programmable interconnection points that use electronic charges to store a pattern in the AND and OR arrays.

Problem: *Design a PLA to recognize each of the 10 decimal digits represented in binary form and to correctly drive a 7-segment display.*

Solution: The PLA must have 4 inputs, as shown in the following Fig. Four bits are required to represent the 10 decimal numbers. There must be 7 outputs (abcdefg).

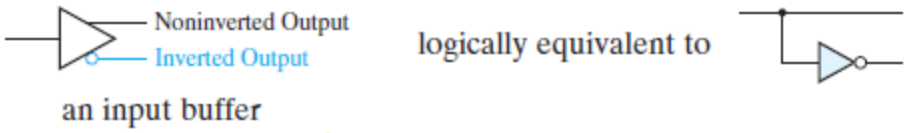


The circuit given in the following Figure shows the links after programming. The input AND-gate array is programmed such that, each AND gate decodes one of the decimal numbers. Then, links are removed from the output OR-gate array, such that proper segments of the indicator are illuminated. For example, when ABCD = LHLH, segments *afgcd* are illuminated to display the decimal number 5.

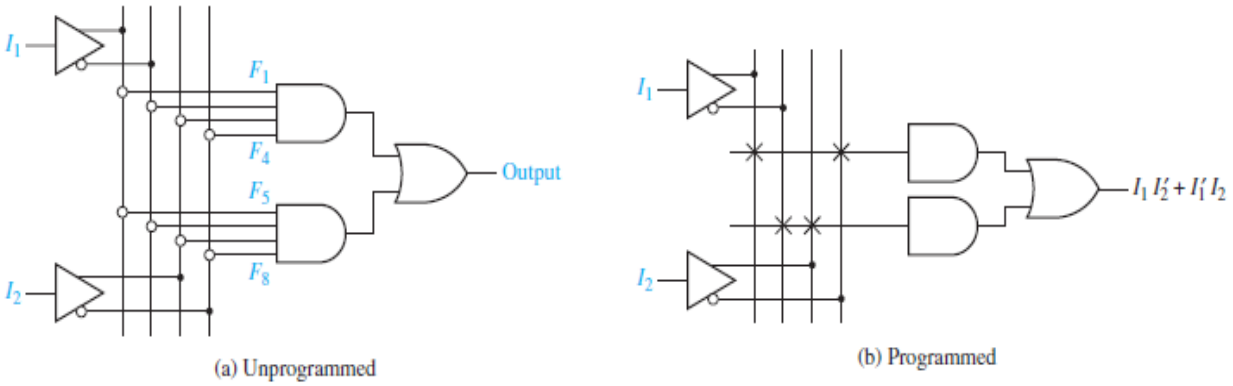


Programmable Array Logic (PAL):

A *PAL* is a special case of the programmable logic array (PLA) in which the AND array is programmable and the OR array is fixed. The following Figure represents a segment of an un-programmed PAL.



Consider the PAL segment of the following Figure (a), used to realize the function $I_1 I'_2 + I'_1 I_2$. The X's in the following Figure (b) indicate that I_1 *and* I'_2 lines are connected to the first AND gate, and the I'_1*and* I_2 lines are connected to the other gate.



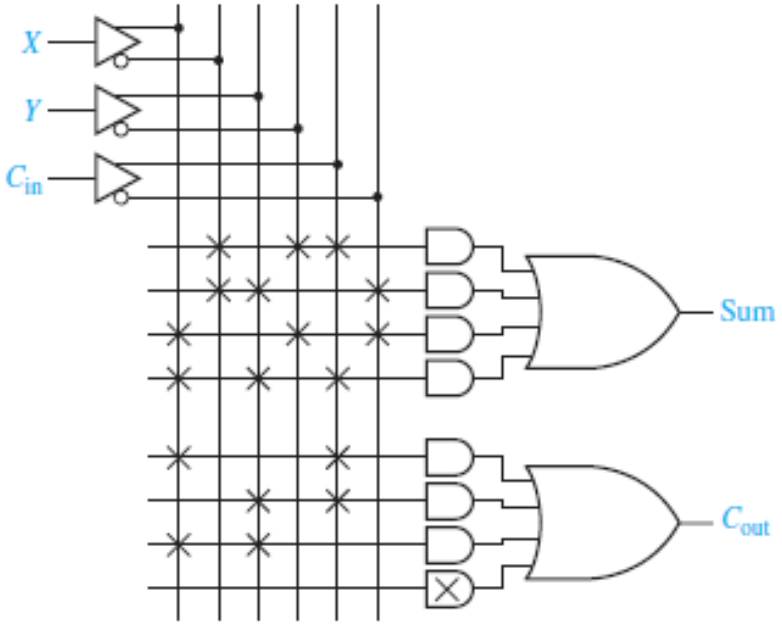
Example: *Implement Full Adder using PAL.*

Solution: The logic equations for the full adder are:

$$Sum = X'Y'Cin + X'YC'in + XY'C'in + XYCin$$

$$Cout = XY + YCin + XCin$$

The following Figure shows PAL where each OR gate is driven by four AND gates. The X's on the diagram show the connections that are programmed into the PAL to implement the full adder equations.



Problem: *Generate the following Boolean function using PAL.*

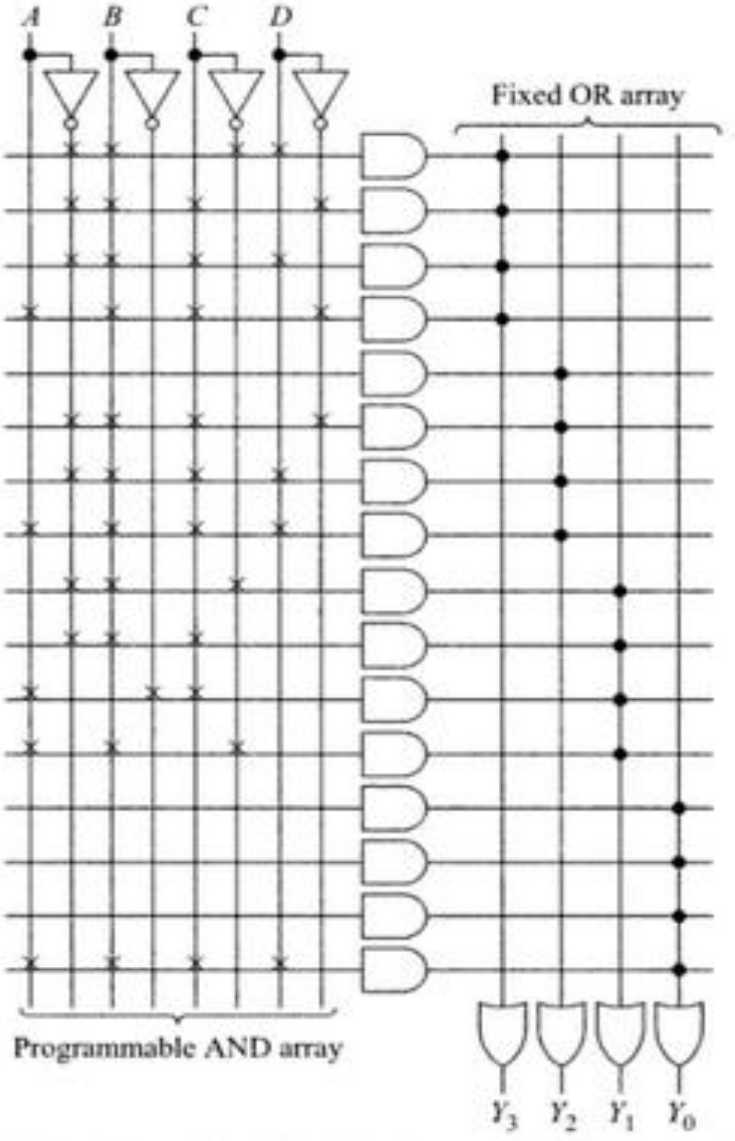
$$Y_3 = \bar{A}B\bar{C}D + \bar{A}BC\bar{D} + \bar{A}BCD + ABC\bar{D}$$

$$Y_2 = \bar{A}BC\bar{D} + \bar{A}BCD + ABCD$$

$$Y_1 = \bar{A}B\bar{C} + \bar{A}BC + A\bar{B}C + AB\bar{C}$$

$$Y_0 = ABCD$$

Solution: Start with first equation. The first desired product is A'BC'D, which is marked as shown in the following Figure. The fixed OR connections on the output side imply that the first OR gate produces an output of first equation.



Homework:

1] Realize a full adder using a 3-to-8 line decoder and

(a) two OR gates, (b) two NOR gates.

2] Derive the logic equations for a 4-to-2 priority encoder.

3] Implement the following equations using PLA –

$$X = AB'D + AC' + BC + C'D'$$

$$Y = A'C' + AC + C'D'$$

$$Z = CD + A'C' + AB'D$$

4] Implement a full subtracter using a PAL.

5] Use a 4-to-1 multiplexer and a minimum number of external gates to realize the function

$F(w, x, y, z) = \sum m\,(3, 4, 5, 7, 10, 14) + \sum d\,(1, 6, 15)$.

Try these –

1] Show how

(a) two 2-to-1 multiplexers (with no added gates) could be connected toform a 3-to-1 MUX. Input selection should be as follows:

If AB = 00, select I_0

If AB = 01, select I_1

If AB = 1– (B is a don't-care), select I_2

(b) two 4-to-1 and one 2-to-1 multiplexers could be connected to form an 8-to-1 MUX with three control inputs.

(c) four 2-to-1 and one 4-to-1 multiplexers could be connected to form an 8-to-1 MUX with three control inputs.

(d) to implement a 32-to-1 multiplexer using two 16-to-1 multiplexers and a 2-to-1 multiplexer.

2] Implement the function $R = ab'h' + bch' + eg'h + fgh$ using –

(a) 2-to-1 MUXes

(b) only tri-state buffers.

3] Implement a full adder

(a) using two 8-to-1 MUXes. Connect X, Y, and Cin to the control inputs of the MUXes and connect 1 or 0 to each data input.

(b) using two 4-to-1 MUXes and one inverter. Connect X and Y to the control inputs of the MUXes, and connect 1's, 0's, Cin, or Cin to each data input.