Module 3
Chapter 1: Combinational Circuit Design and Simulation using Gates.

3.1 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.

3.2DESIGN 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:

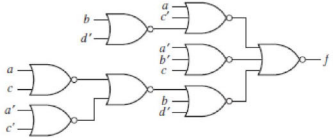
$$f = \left(a' + b' + c\right)\left(a + b' + c'\right)\left(a + c' + d\right)\left(a + b + c + d'\right)\left(a' + b + c' + d'\right)$$

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. $f' = abc' + a'c\left(b + d'\right) + b'd\left(a'c' + ac\right)$

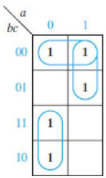
Or $f = \left[\left(a' + b' + c\right)\right]\left[\left(a + c'\right) + \left(b'd\right)\right]\left[\left(b + d'\right) + \left(a + c\right)\left(a' + c'\right)\right]$



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

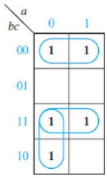
$$f1 = \Sigma m\left(0,2,3,4,5\right)\ f2 = \Sigma m\left(0,2,3,4,7\right)\ f3 = \Sigma m\left(1,2,6,7\right)$$

Solution:



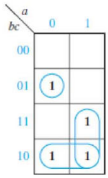
$$f1 = \Sigma m\left(0,2,3,4,5\right)$$

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



$$f2 = \Sigma m\left(0,2,3,4,7\right)$$

$$f2 = b'c' + bc + a'b$$



$$f3 = \Sigma m\left(0,2,3,4,5\right)$$

$$f3 = a'b'c + ab + bc'$$

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

$$f1 = b'\underbrace{\left(a + c'\right)} + \underline{a'b}$$

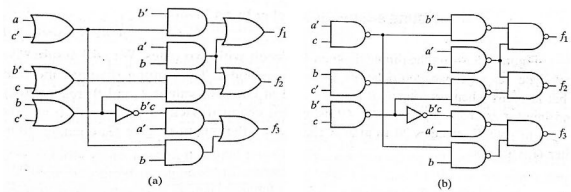
$$f2 = b\left(a' + c\right) + b'c' \quad or \quad f2 = \left(b' + c\right)\left(b + c'\right) + \underline{a'b}$$

$$f3 = a'b'c + b\underbrace{\left(a + c'\right)}$$

The second expression for f2 has term common to $f1$, *so we will choose the second expression. We can eliminate the remaining three-input gate from f3 by noting that*

$$a'b'c = a'\left(b'c\right) = a'\left(b + c'\right)'$$

Figure shows the resulting circuit, using common terms $a'b$ *and* $a + c'$. *because each output gate is an OR, the conversion to NAND gates, as shown in Figure (b), is straightforward.*



(a) (b)

3.3 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:

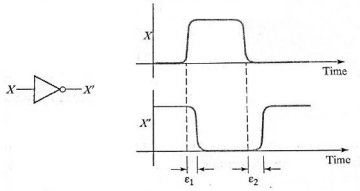


FIGURE: Propagation Delay in an Inverter

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:

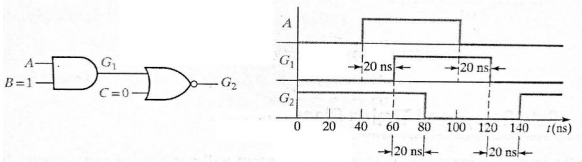


FIGURE: Timing Diagram for AND-NOR Circuit

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.

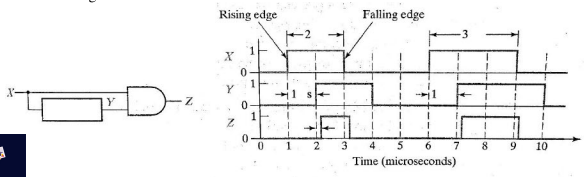


FIGURE: Timing Diagram for Circuit with Delay

3.4 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:

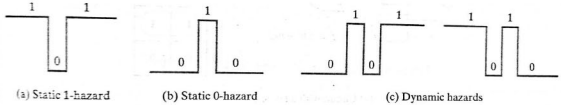
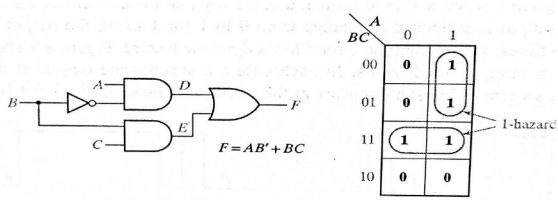


FIGURE: Types of Hazards

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.



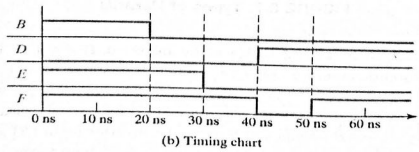


FIGURE: Detection of a 1-Hazard

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.

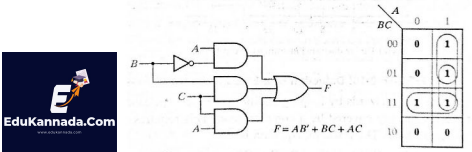
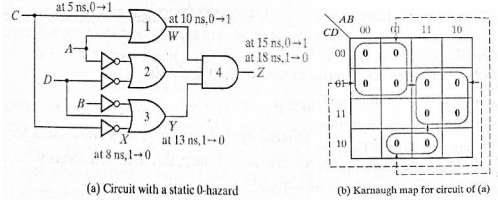


FIGURE: Circuit with Hazard Removed

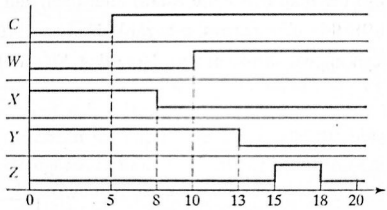
Detection of Static 0 Hazard: The following Figure shows a circuit with several 0-hazards. The product-of-sums representation for the circuit output is

$$F = \left(A + C\right)\left(A' + D\right)\left(B' + C' + D\right)$$

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.

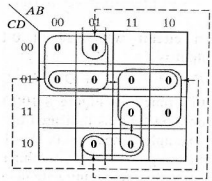
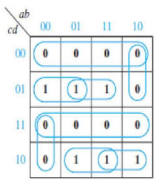


FIGURE: Karnaugh Map Removing Hazards

$$F = \left(A + C\right)\left(A' + D'\right)\left(B' + C' + D\right)\left(C + D'\right)\left(A + B' + D\right)\left(A' + B' + C'\right)$$

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 $\left(F^t\right)$ 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 the 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, DE Morgan's laws, etc. Treat each x_i and x'_i as independent variables to prevent introduction of hazards.

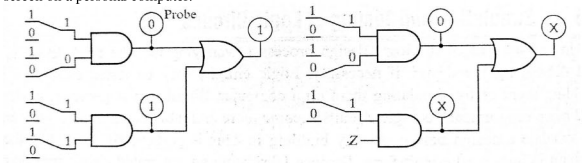
3.5 SIMULATION AND TESTING OF LOGIC CIRCUITS:

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.

TABLE: AND and OR Functions for Four-Valued Simulation

·	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:

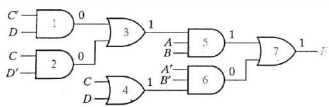


FIGURE: Logic Circuit with Incorrect Output

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 $0 + 0 \neq 1$. 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.

Module 3
Chapter 2: Multiplexers, Decoders, & Programmable Logic Devices.

3.6 INTRODUCTION:

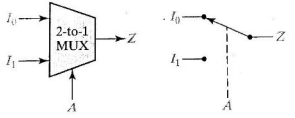
Integrated circuit may be classified as small-scale integration SSI, medium scale integration MSI, large-scale integration LSI, or very large-scale integration VLSI, depending on the number of gates in each integrated circuit package and the type of function performed.

SSI functions include NAND, NOR, AND, and OR gates, inverters, and flip-flops. SSI integrated circuit package is typically contain 1 to 4 gates, six inverters, or one or two flip-flops. The MSI integrated circuits such as adders, multiplexers, decoders, registers, and counters, perform more complex functions. Such integrated circuits typically contain the equivalent of 12 to 100 gates in one package. More complex functions such as memories and microprocessors are classified as LSI or VLSI integrated circuits. An LSI integrated circuit generally contains 100 to a few thousands gates in a single package, and VLSI integrated circuit contain several thousand gates or more.

It is generally uneconomical to design digital systems using only SSI and MSI integrated circuit. By using LSI and VLSI functions, the required number of integrated circuit packages is generally reduced. The cost of mounting and wiring the integrated circuit as well as the cost of designing and maintaining the digital system may be significantly lower than the LSI and VLSI functions are used.

3.7 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.



$$Z = A'I_0 + AI_1$$

FIGURE: 2-to-1 Multiplexer and Switch Analog

When the control input A is 0, the switch is in the upper position and the MUX output is Z = I0; when A is 1, the switch is in the lower position and the MUX output is Z = I1. In other words, a MUX acts like a switch that selects one of the data inputs (I0 or I1) 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 2n-to-1 multiplexer.

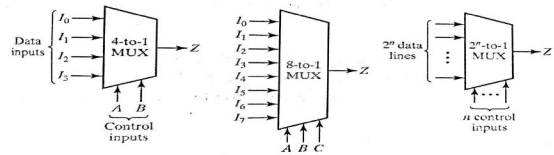


FIGURE : Multiplexers

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 I0; similarly, the control inputs 01, 10, and 11 give outputs of I1, I2, and I3, respectively. The 4-to-1 multiplexer is described by the equation:

$$Z = A'B'I_0 + A'BI_1 + AB'I_2 + ABI_3$$

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

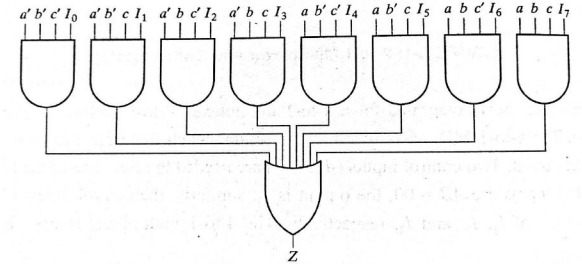


FIGURE : Logic Diagram for 8-to-1 MUX

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 *2n* data inputs. The general equation for the output of a MUX with *n* control inputs and *2n* 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.

There are several other implementations of the 8-to-1 MUX. Each of the gates shown in the figure above can be replaced by NAND gates to obtain NAND Gate implementation. If an NOR gate implementation is wanted, the equation for Z can be written as a product of sums:

$$Z = \begin{pmatrix} A + B + C + I_0 \\ A' + B + C + I_4 \end{pmatrix} \begin{pmatrix} A + B + C' + I_1 \\ A' + B + C' + I_5 \end{pmatrix} \begin{pmatrix} A + B' + C + I_2 \\ A' + B' + C + I_6 \end{pmatrix} \begin{pmatrix} A + B' + C' + I_3 \\ A' + B' + C' + I_7 \end{pmatrix}$$

Implementations with more than two levels of gates can be obtained by factoring the equation for Z. For example if multiple level and NAND-gate implementation is desired, above equation can be factored, One factorization is

$$Z = A'B'\left(C'I_0 + CI_1\right) + A'B\left(C'I_2 + CI_3\right) + AB'\left(C'I_4 + CI_5\right) + AB\left(C'I_6 + CI_7\right)$$

The corresponding NAND gate circuit is shown in the figure below. Note that the data inputs are connected to four 2-to-1 MUXs with C as a select line, and the output of these 2-to-1 MUXs are connected to 4-to-1 MUX with A and B as a select lines. Figure below shows this in a block diagram form.

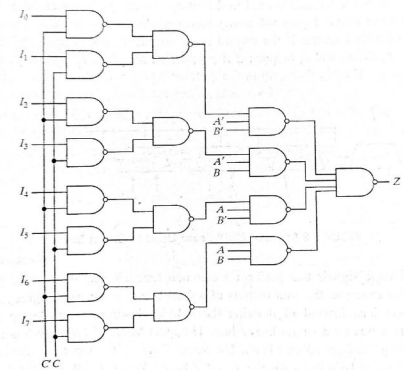


FIGURE : A Multi-Level Implementation of an 8-to-1 MUX

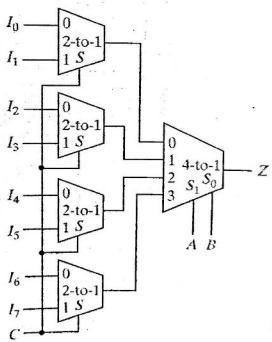


FIGURE : Component MUXs of Figure 9-4

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.

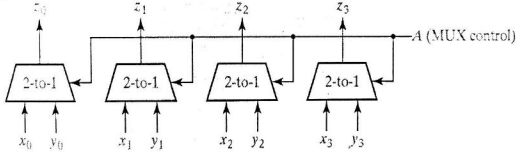


FIGURE: Quad Multiplexer Used to Select Data

Four combinations of multiplexers with an enable are possible. The output can be active high or active low, whereas the enable can be active high or active low. In a block diagram for the MUX, an active low line is indicated by inserting a bubble on the line to indicate the inclusion of an inversion. Figure below shows this combination for 4 to 1 MUX.

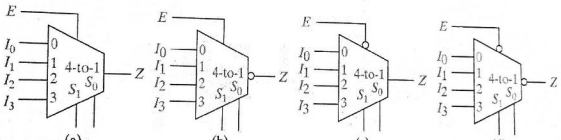


FIGURE: Active-High, Active-Low Enable and Output Combinations

In addition to acting as a data selector, a MUX can implement more general logic functions. In figure below a 4 to 1 mux is used to implement the function

$$Z = C'D'\left(A' + B'\right) + C'D\left(A'\right) + CD'\left(AB' + A'B\right) + CD'\left(0\right)$$
$$= A'C' + A'BD' + AB'D'$$

Given a switching function, a MUX implementation can be obtained using Shannon's expansion of the function. In general the complexity of the implementation will depend upon which function inputs are used as the MUX select inputs, so it is necessary to try different combination to obtain the simplest solution.

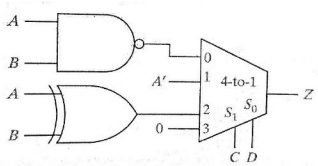


FIGURE: Four-Variable Function Implemented with a 4-to-1 MUX

3.8 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, $F = C$.

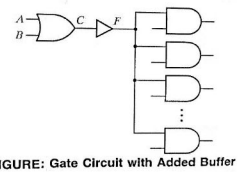


FIGURE: Gate Circuit with Added Buffer

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.

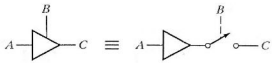
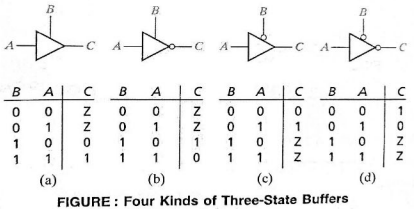


FIGURE: Three-State Buffer

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)

FIGURE : Four Kinds of Three-State Buffers

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.

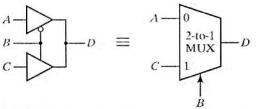
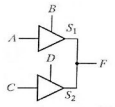


FIGURE: Data Selection Using Three-State Buffers

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.



S_1 \ S_2	X	0	1	Z
X	X	X	X	X
0	X	0	X	0
1	X	X	1	1
Z	X	0	1	Z

FIGURE: Circuit with Two Three-State Buffers

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.

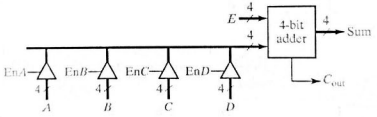


FIGURE: 4-Bit Adder with Four Sources for One Operand

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.

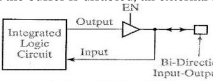
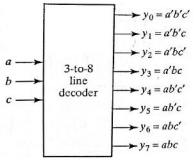


FIGURE: Integrated Circuit with Bi-Directional Input-Output Pin

3.9 Decoders and Encoders:

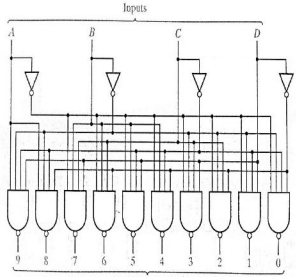
The ***decoder*** is another commonly used type of integrated circuit. The following Figure shows the diagram and truth table for a 3-to-8 line decoder. This decoder generates all of the minterms of the three input variables. Exactly one of the output lines will be 1 for each combination of the values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

FIGURE: A 3-to-8 Line Decoder

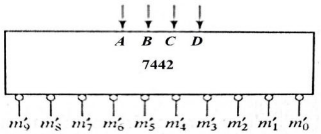
The following Figure illustrates a 4-to-10 decoder. This decoder has inverted outputs (indicated by the small circles). For each combination of the values of the inputs, exactly one of the output lines will be 0. When a binary-coded-decimal digit is used as an input to this decoder, one of the output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



(a) Logic diagram

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



(b) Block diagram

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:

$y_i = m_i$; $i = 0$ to $2^n - 1$ (non-inverted outputs)

or

$y_i = m_i' = M_i$, $i = 0$ to $2^n - 1$ (inverted outputs)

Where m_i is a minterm of the n input variables and M_i is a maxterm.

Because an n-input decoder generate all the minterms of n variables, n-variable functions can be realized by ORing together selected minterm outputs from a decoder. If the decoder outputs are inverted, then then NAND gates can be used to generate the functions, as illustrated in the following example.

Realize $f_1(a,b,c,d) = m_1 + m_2 + m_4$ and $f_2(a,b,c,d) = m_4 + m_7 + m_9$

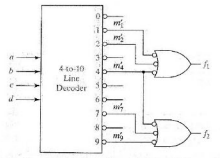
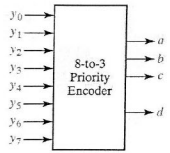


FIGURE: Realization of a Multiple-Output Circuit Using a Decoder

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.



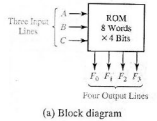
y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	a	b	c	d
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
X	1	0	0	0	0	0	0	0	0	1	1
X	X	1	0	0	0	0	0	0	1	0	1
X	X	X	1	0	0	0	0	0	1	1	1
X	X	X	X	1	0	0	0	1	0	0	1
X	X	X	X	X	1	0	0	1	0	1	1
X	X	X	X	X	X	1	0	1	1	0	1
X	X	X	X	X	X	X	1	1	1	1	1

FIGURE: An 8-to-3 Priority Encoder

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 Y_5 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.

3.10 MEMORIES:

A read only memory ROM consists of an array of Semiconductor devices that are interconnected to store an area of a binary data. Once binary data is stored in the ROM, it can be read out whenever desired, but the data that is stored cannot be changed under normal operating conditions. Figure (a) below shows a ROM which has three input lines and a 4 output lines. Figure (B) shows the typical truth table which relates the ROM inputs and outputs.



(a) Block diagram

A	B	C	F_0	F_1	F_2	F_3
0	0	0	1	0	1	0
0	0	1	1	0	1	0
0	1	0	0	1	1	1
0	1	1	0	1	0	1
1	0	0	1	1	0	0
1	0	1	0	0	0	1
1	1	0	1	1	1	1
1	1	1	0	1	0	1

Typical Data Stored in ROM (2^3 words of 4 bits each)

(b) Truth table for ROM

FIGURE: An 8-Word × 4-Bit ROM

For each combination of input values on the three input lines, the corresponding patterns of 0's and 1's appear on the ROM output lines. For example, if the combination ABC = 010 is applied to the input lines, the patterns of $F_0F_1F_2F_3$ =0111 appears on the output lines. Each of the output patterns that is stored in the ROM is called a word. Because the ROM has

three input lines, we have 23 = 8 different combinations of input values. Each input combinations serves as an *address* which can select one of the eight words stored in the memory. Because there are four output lines, each word is four bits long and the size of this ROM is an 8 words × 4 bits.

A ROM which has *n* input lines and *m* output lines (shown in figure below) contains an array of 2^n words, and each word is m-bits long. The input lines serve as an address to select one of the 2^n words. When an input combination is applied to ROM, the patterns of 0's and 1's which is stored in the corresponding word in the memory appears at the output lines. For the example in figure if 00...11 is applied to the input (address lines) of the ROM, the word 110... 010 will be selected and transferred to the output lines. A $2^n \times m$ ROM can realize *m* functions of *n* variables because it can store a truth table with 2^n rows and *m* columns. Typical sizes of commercially available from range from 32 words × 4 bits to 512k words × 8 bits, or larger.

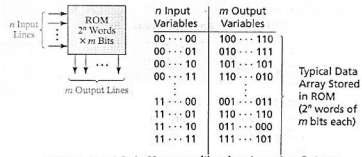


FIGURE: Read-Only Memory with n Inputs and m Outputs

ROM basically consists of decoder and a memory array as shown in the figure below. When a pattern of 0's and 1's is applied to a decoder inputs, exactly one of the 2^n decoder output is one. This decoder output line selects one of the words in the memory array, and the bit pattern stored in this word is transferred to the memory output lines.

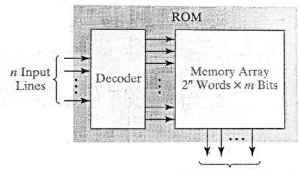


FIGURE: Basic ROM Structure

Figure below illustrates one possible internal structure of 8-word × 4bit ROM.

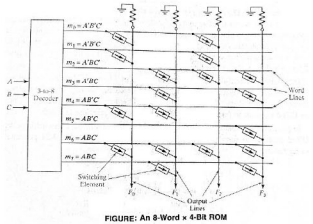


FIGURE: An 8-Word × 4-Bit ROM

The decoder generate the 8 minterms of the three input variables. The memory array forms the four output functions by ORing together selected minterms. A switching element is placed at the intersection of the word line and *output line* if the corresponding mid-term is to be included in the output function; otherwise, the switching element is omitted (or not connected). If switching element connects an output line to a word line which is 1, the output line will be one. Otherwise, the pull down resistor at the top of the figure cause the output line to be 0. So the switching element which are connected in this way in the memory array effectively form an OR gate for each of the output functions. For example m0, m1, m4 and m6 are ORed together to form F_0. Figure below shows the equivalent OR gate.

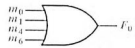


FIGURE: Equivalent OR Gate for F_0

In general, those minterms which are connected to output lines by switching elements ORed together to form the output F_i. Thus the ROM in figure generates the following function:

$$F_0 = \Sigma m\{0,1,4,6\} = A'B' + AC'$$
$$F_1 = \Sigma m\{2,3,4,6,7\} = B + AC'$$
$$F_2 = \Sigma m\{0,1,2,6\} = A'B' + BC'$$
$$F_3 = \Sigma m\{2,3,5,6,7\} = AC + B$$

Figure below shows an internal diagram of the ROM. The switching elements are the intersection of the rows and columns of the memory are indicated using X's. And X indicates that the switching element is present and connected, and no X indicates that corresponding element is absent or not connected.

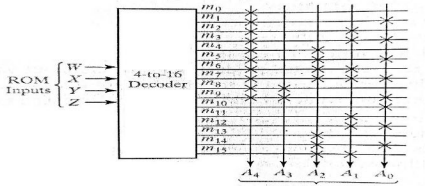


FIGURE: ROM Realization of Code Converter

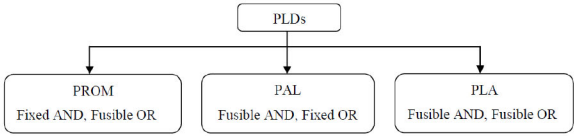
Three common types of ROMs are Mask Programmable ROMs, Programmable ROMs (PROMS), and electrically erasable Programmable ROMs (EEPROMS). At the time of the manufacture the data are is permanently stored in a mask Programmable ROM. This is

accomplished by selectively including our omitting the switching elements at the row column intersections of the memory are. This requires preparation of a special mask, which is used during fabrication of the Integrated circuit. Preparation of this mask is expensive so the use of mask Programmable ROM is economically feasible only if a large quantity (typically several thousand or more) is required with the same data are. If the small quantity of the Rhombus is required with the given data are a, hi proms may be used.

Modification of the data stored in ROM is often necessary during the developmental phases of a digital system, So EEPROMs are used instead of mask Programmable ROMs. EEPROM use a special charge storage mechanism to enable or disable the switching element in the memory array. A PROM programmer is used to provide appropriate voltage pulses to store electronic charges in the memory allocation. Data stored in this manner is generally permanent until erased.

3.11 PROGRAMMABLE LOGIC ARRAYS (PLA):

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.

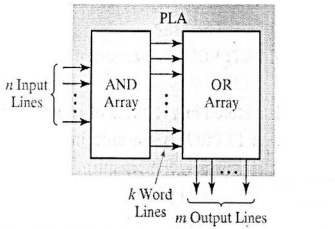


FIGURE: Programmable Logic Array Structure

Example: Realize the following functions using PLA:
F0 = Σm (0, 1, 4, 6) = A'B' + AC'

F1 = Σm (2, 3, 4, 6, 7) = B + AC'
F2 = Σm (0, 1, 2, 6) = A'B' + BC'
F3 = Σm (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.

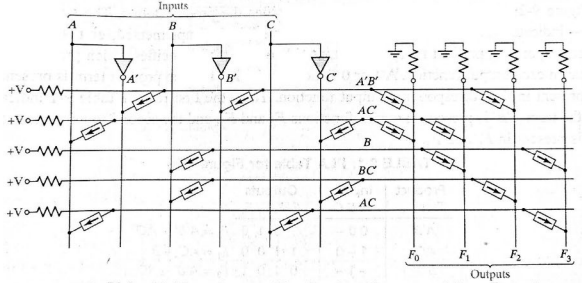


FIGURE: PLA with Three Inputs, Five Product Terms, and Four Outputs

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

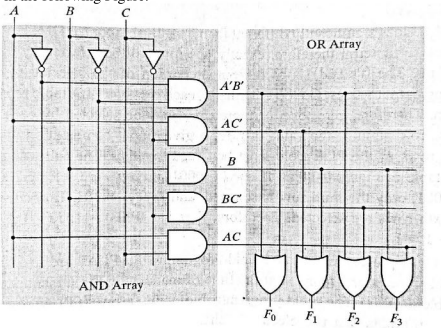


FIGURE: AND-OR Array Equivalent

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.

TABLE: PLA Table

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:

$f_1 = a'bd + abd + ab'c' + bc$

$f_2 = a'bd + c$

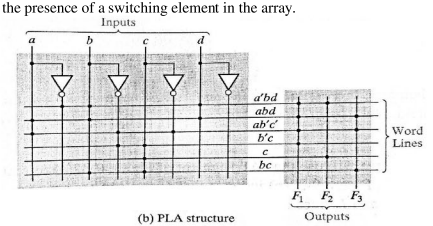
$f_3 = abd + ab'c' + bc$

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:

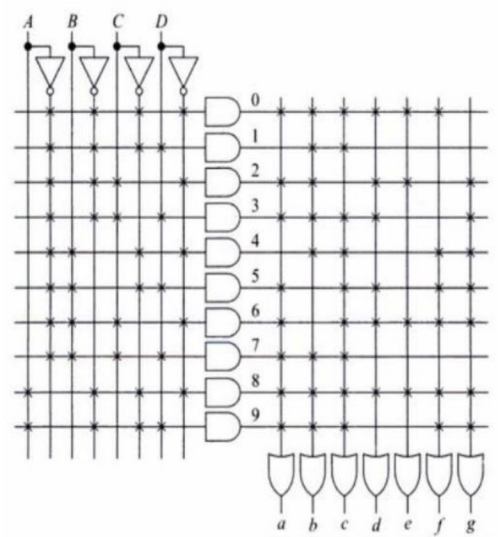
Mask-programmable and field-programmable PLAs are available. The mask-programmable type is programmed at the time of manufacture. The field-programmable logic array (FPLA) has programmable interconnection points that use electronic charges to store a pattern in the AND and OR arrays.

Problem:

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

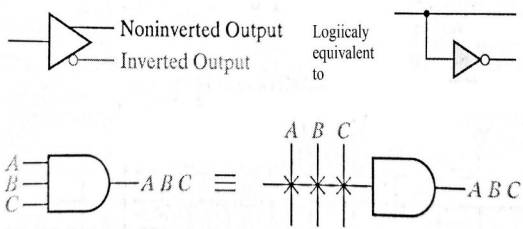
Solution:

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).

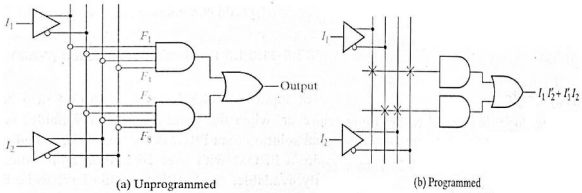


Programmable Array Logic (PAL):

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:
$$sum = x'y'c_{in} + x'yc'_{in} + xy'c'_{in} + xyc_{in}$$
$$C_{out} = xy + yc_{in} + xc_{in}$$

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.

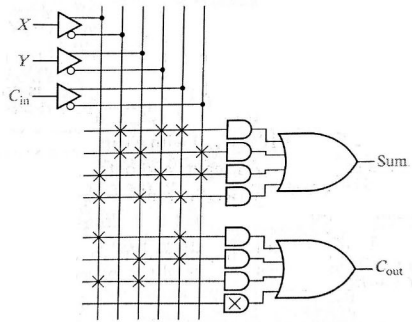


FIGURE: Implementation of a Full Adder Using a PAL