Catalog

4.1 Asynchronous Sequential Circuits

Asynchronous sequential circuits do not use clock signals as synchronous circuits do. Instead, the circuit is driven by the pulses of the inputs which means the state of the circuit changes when the inputs change. Also, they don't use clock pulses. The change of internal state occurs when there is a change in the input variable. Their memory elements are either un-clocked flip-flops or time-delay elements. They are similar to combinational circuits with feedback.

Advantages –

- No clock signal, hence no waiting for a clock pulse to begin processing inputs, therefore fast. Their speed is faster and theoretically limited only by propagation delays of the logic gates.
- Robust handling. Higher performance function units, which provide average-case completion rather than worst-case completion. **Lower power consumption** because no transistor transitions when it is not performing useful computation. The absence of clock drivers reduces power consumption. Less severe electromagnetic interference (EMI).
- More tolerant to process variations and external voltage fluctuations. Achieve high performance while gracefully handling variable input and output rates and mismatched pipeline stage delays. Freedom from difficulties of distributing a high-fan-out, timing-sensitive clock signal. Better modularity.
- **Less assumptions** about the manufacturing process. Circuit speed adapts to changing temperature and voltage conditions. Immunity to transistor-to-transistor variability in the manufacturing process, which is one of the most serious problems faced by the semiconductor industry

Disadvantages –

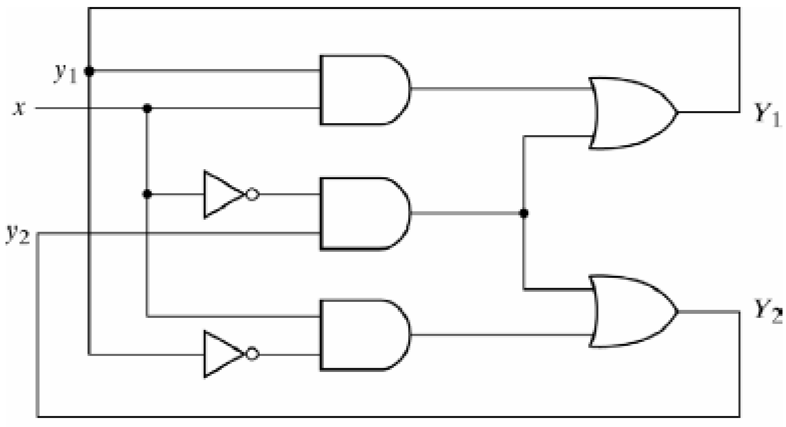
- Some asynchronous circuits may require extra power for certain operations.
- More **difficult to design** and subject to problems like sensitivity to the relative arrival times of inputs at gates. If transitions on two inputs arrive at almost the same time, the circuit can go into the wrong state depending on slight differences in the propagation delays of the gates which are known as **race condition**.

- The number of circuit elements (transistors) maybe double that of synchronous circuits. Fewer people are trained in this style compared to synchronous design. Difficult to test and debug. Their **output** is **uncertain**.
- The performance of asynchronous circuits may be reduced in architectures that have a complex data path. Lack of dedicated, asynchronous design-focused commercial EDA tools.

4.2 Analysis Procedure

Transition Table

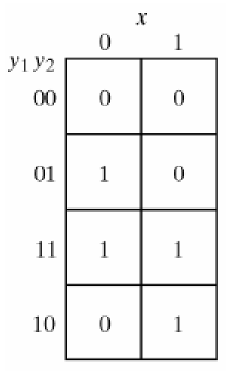
An example of an asynchronous sequential circuit is shown below:



The analysis of the circuit starts by considering the excitation variables (*Y*1 and *Y*2) as outputs and the secondary variables (*y*1 and *y*2) as inputs.

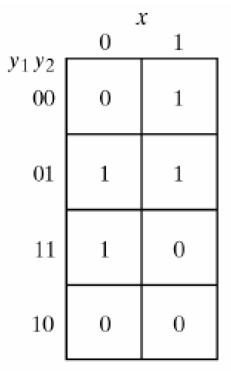
The Boolean expressions are:

$$Y_1 = xy_1 + x'y_2$$
$$Y_2 = xy_1' + x'y_2$$



y_1y_2 \ x	0	1
00	0	0
01	1	0
11	1	1
10	0	1

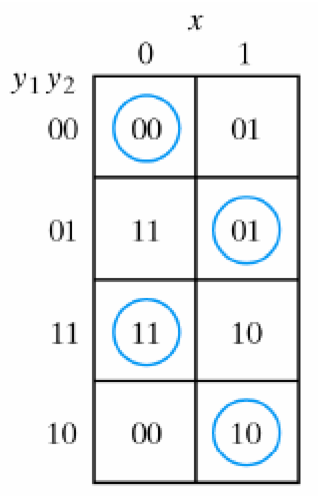
Map for
$Y_1 = xy_1 + x'y_2$



y_1y_2 \ x	0	1
00	0	1
01	1	1
11	1	0
10	0	0

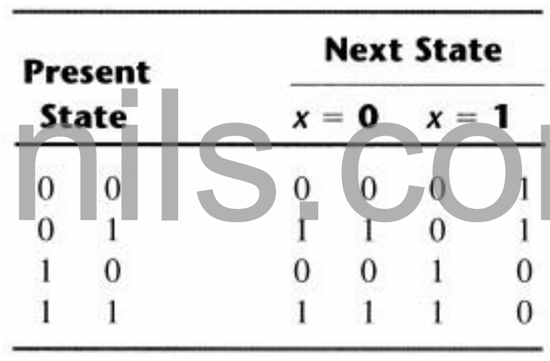
Map for
$Y_2 = xy'_1 + x'y_2$

Combining the binary values in corresponding squares the following *transition table* is obtained:



y_1y_2 \ x	0	1
00	00	01
01	11	01
11	11	10
10	00	10

The transition table shows the value of *Y* = *Y*1*Y*2 inside each square. Those entries where *Y* = *y* are circled to indicate a stable condition. The circuit has four stable *total states* – *y*1*y*2*x* = 000, 011, 110, and 101 – and four unstable total states – 001, 010, 111, and 100. The state table of the circuit is shown below:



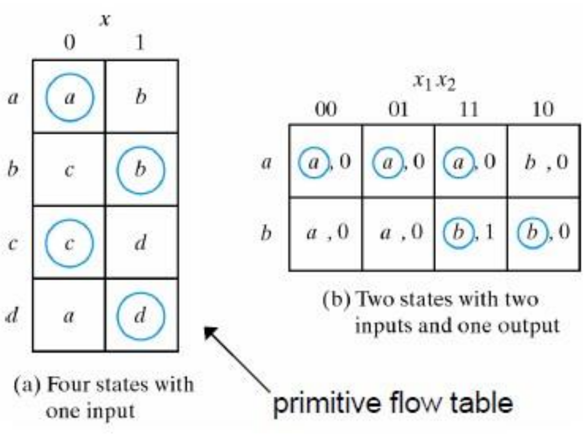
Present State		Next State $x = 0$		Next State $x = 1$	
0	0	0	0	0	1
0	1	1	1	0	1
1	0	0	0	1	0
1	1	1	1	1	0

This table provides the same information as the transition table.

Flow Table

In a *flow table* the states are named by letter symbols. Examples of flow tables are as follows:

In order to obtain the circuit described by a flow table, it is necessary to assign to each state a distinct value.



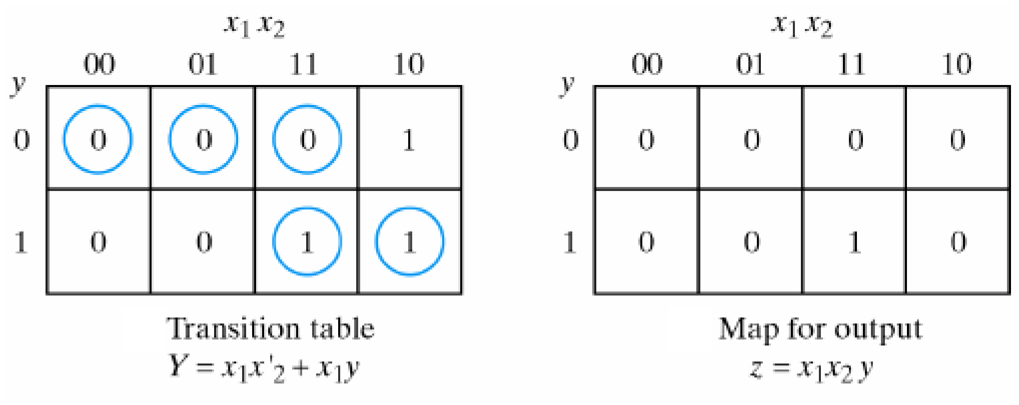
	0	1
a	a	b
b	c	b
c	c	d
d	a	d

(a) Four states with one input

	00	01	11	10
a	a, 0	a, 0	a, 0	b, 0
b	a, 0	a, 0	b, 1	b, 0

(b) Two states with two inputs and one output

primitive flow table



y \ x_1x_2	00	01	11	10
0	0	0	0	1
1	0	0	1	1

Transition table
$Y = x_1x'_2 + x_1y$

y \ x_1x_2	00	01	11	10
0	0	0	0	0
1	0	0	1	0

Map for output
$z = x_1x_2y$

The resulting logic diagram is shown below:

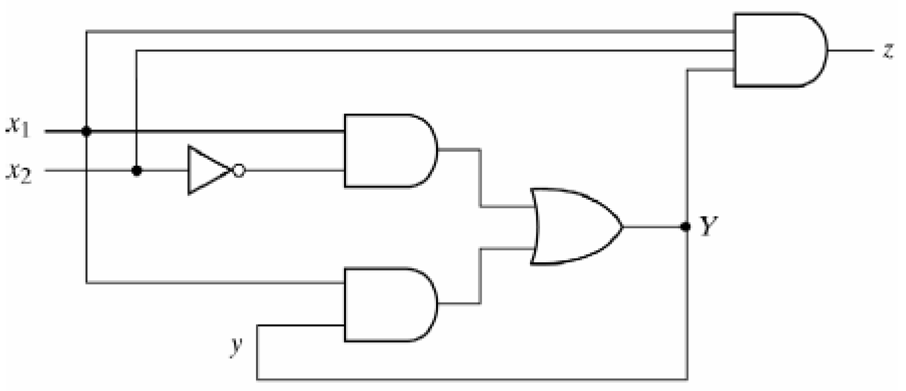
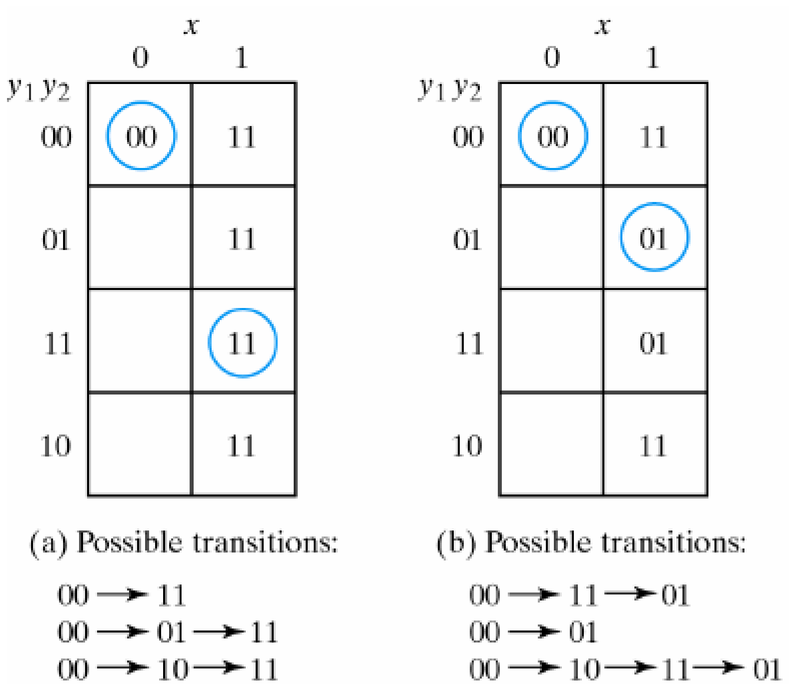


Fig 4.2.1 Logic diagram

Race Conditions

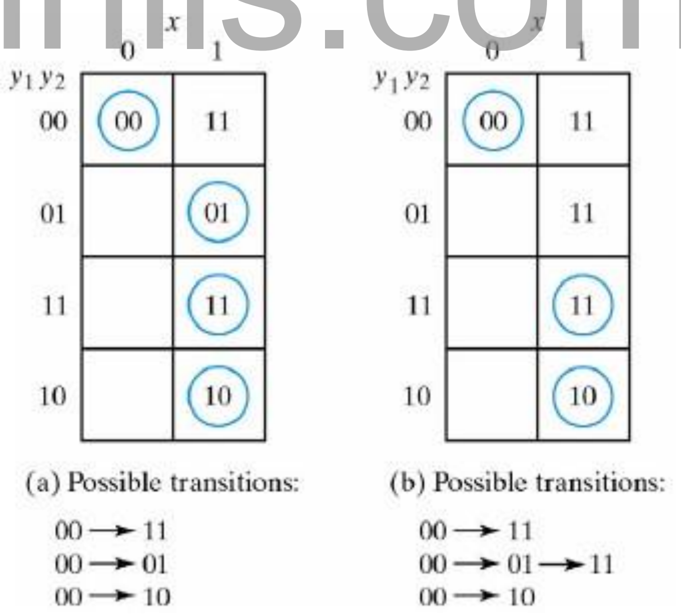
A *race* condition exists in an asynchronous circuit when two or more binary state variables change value in response to a change in an input variable. When unequal delays are encountered, a race condition may cause the state variable to change in an unpredictable manner. If the final stable state that the circuit reaches does not depend on the order in which the state variables change, the race is called a *noncritical race*. Examples of noncritical races are illustrated in the transition tables below:



(a) Possible transitions:
$00 \rightarrow 11$
$00 \rightarrow 01 \rightarrow 11$
$00 \rightarrow 10 \rightarrow 11$

(b) Possible transitions:
$00 \rightarrow 11 \rightarrow 01$
$00 \rightarrow 01$
$00 \rightarrow 10 \rightarrow 11 \rightarrow 01$

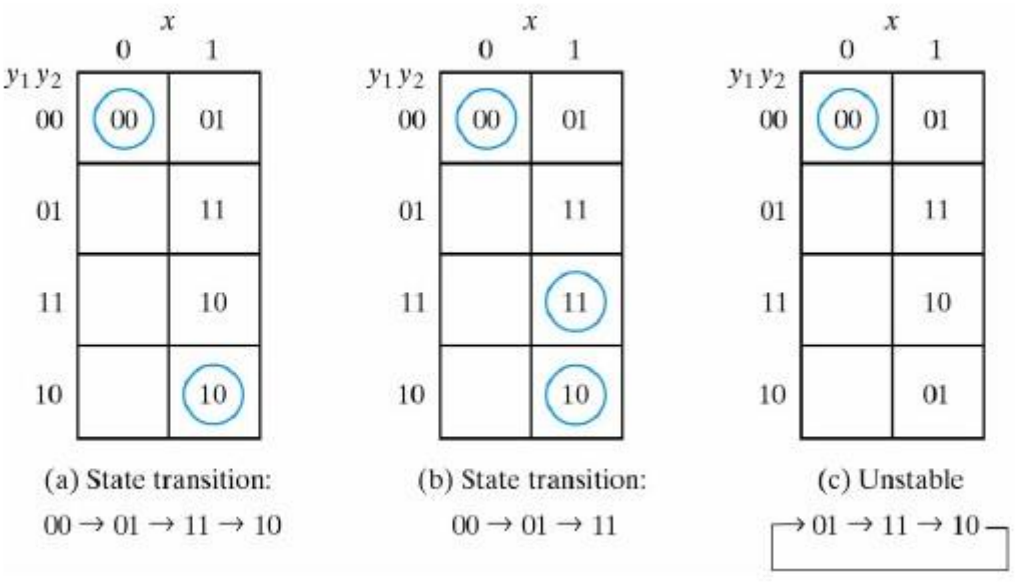
The transition tables below illustrate critical races:



(a) Possible transitions:
$00 \rightarrow 11$
$00 \rightarrow 01$
$00 \rightarrow 10$

(b) Possible transitions:
$00 \rightarrow 11$
$00 \rightarrow 01 \rightarrow 11$
$00 \rightarrow 10$

Races can be avoided by directing the circuit through a *unique* sequence of intermediate unstable states. When a circuit does that, it is said to have a *cycle*. Examples of cycles are:



y_1y_2 \ x	0	1
00	00	01
01		11
11		10
10		10

(a) State transition: 00 → 01 → 11 → 10

y_1y_2 \ x	0	1
00	00	01
01		11
11		11
10		10

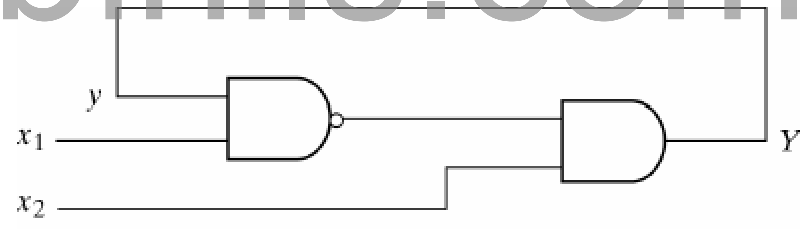
(b) State transition: 00 → 01 → 11

y_1y_2 \ x	0	1
00	00	01
01		11
11		10
10		01

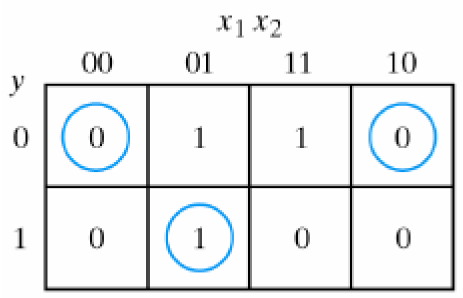
(c) Unstable
→ 01 → 11 → 10 ─┘

Stability Considerations

An asynchronous sequential circuit may become unstable and oscillate between unstable states because of the presence of feedback. The instability condition can be detected from the transition table. Consider the following circuit:



The excitation function is: Y = (x1y)′x2 = (x1′ + y′)x2 = x1′x2 + x2y′ and the transition table for the circuit is:



y \ x_1x_2	00	01	11	10
0	0	1	1	0
1	0	1	0	0

Those values of *Y* that are equal to *y* are circled and represent stable states. When the input *x1x2* is 11, the state variable alternates between 0 and 1 indefinitely.

4.3 HAZARDS

In designing asynchronous sequential circuits, care must be taken to conform to certain restrictions and precautions to ensure that the circuits operate properly. The circuit must be operated in fundamental mode with only one input changing at any time and must be free of critical races. In addition, there is one more phenomenon called a hazard that may cause the circuit to malfunction.

Hazards are unwanted switching transients that may appear at the output of a circuit because different paths exhibit different propagation delays. Hazards occur in combinational circuits, where they may cause a temporary false output value. When they occur in asynchronous sequential circuits hazards may result in a transition to a wrong stable state.

Hazards In Combinational Circuits

A hazard is a condition in which a change in a single variable produces a momentary change in output when no change in output should occur.

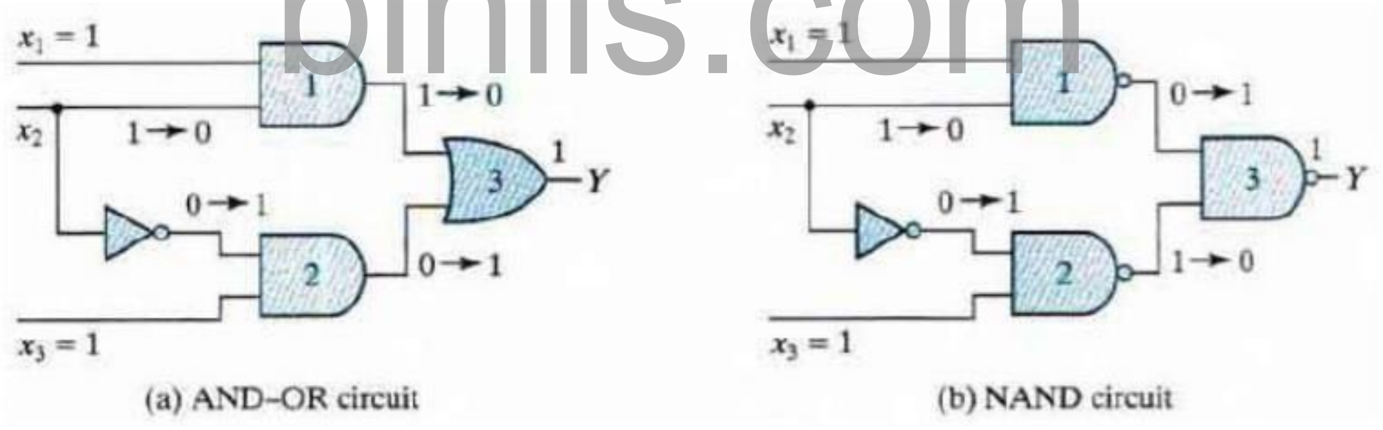


Fig 4.3.1 Circuits with Hazards

Assume that all three inputs are initially equal to 1. This causes the output of gate 1 10 be 1, that of gate 2 to be 0 and that of the circuit to be 1. Now consider a change in x 2 from 1 to 0. Then the output of gate 1 changes to 0 and that of gate 2 changes to 1, leaving the output at 1. However, the output may momentarily go to 0 if the propagation delay through the inverter is taken into consideration. The delay in the inverter may cause the output of gate 1 to change to 0 before the output of gale 2 changes to 1.

The two circuits shown in Fig implement the Boolean function in sum-of-products form:

$$Y = x_1x_2 + \overline{x_2}x_3 \qquad Y = (x_1 + \overline{x_2})(x_2 + x_3)$$

This type of implementation may cause the output to go to 0 when it should remain a 1. If however, the circuit is implemented instead in product-of-sums form namely, then the output may momentarily go to 1 when it should remain 0. The first case is referred to as **static 1-hazard** and the second case as **static 0-hazard**.

A third type of hazard, known as **dynamic hazard**, causes the output to change three or more times when it should change from 1 to 0 or from 0 to 1.

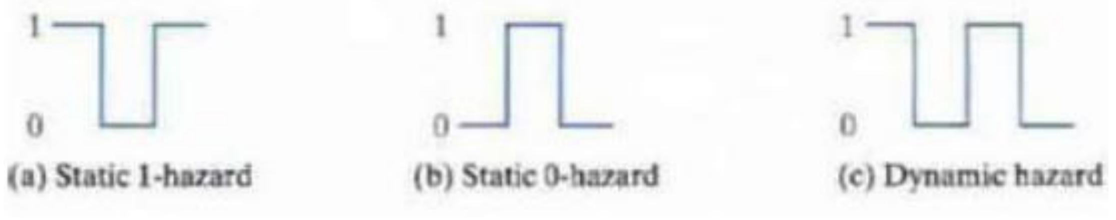


Fig 4.3.2 Types of hazards

The change in x2 from 1 to 0 moves the circuit from minterm 111 to minterm 101. The hazard exists because the change in input results in a different product term covering the two minterm.

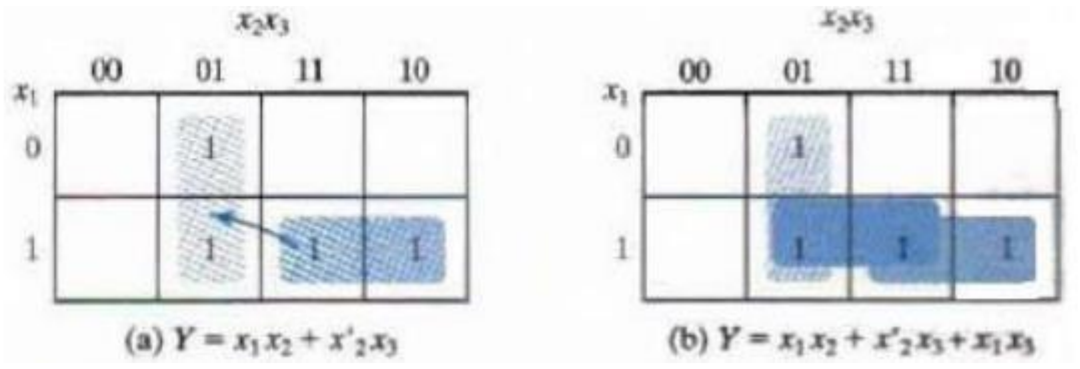

x_2x_3
x_1
00
01
11
10
0
1
1
1
1
1
(a) $Y = x_1x_2 + x'_2x_3$
x_2x_3
x_1
00
01
11
10
0
1
1
1
1
1
(b) $Y = x_1x_2 + x'_2x_3 + x_1x_3$

Fig 4.3.3 Illustrates hazard and its removal

Minterm 111 is covered by the product term implemented in gate 1 and minterm 101 is covered by the product term implemented in gate 2. The remedy for eliminating a hazard is to enclose the two minterms with another product term that overlaps both groupings. The hazard-free circuit obtained by such a configuration is shown in figure below.

The extra gate in the circuit generates the product term x_1x_3. In general, hazard s in combinational circuits can be removed by cove ring any two minterms that may produce a hazard with a product term common to both. The removal of hazards requires the addition of redundant gates to the circuit.

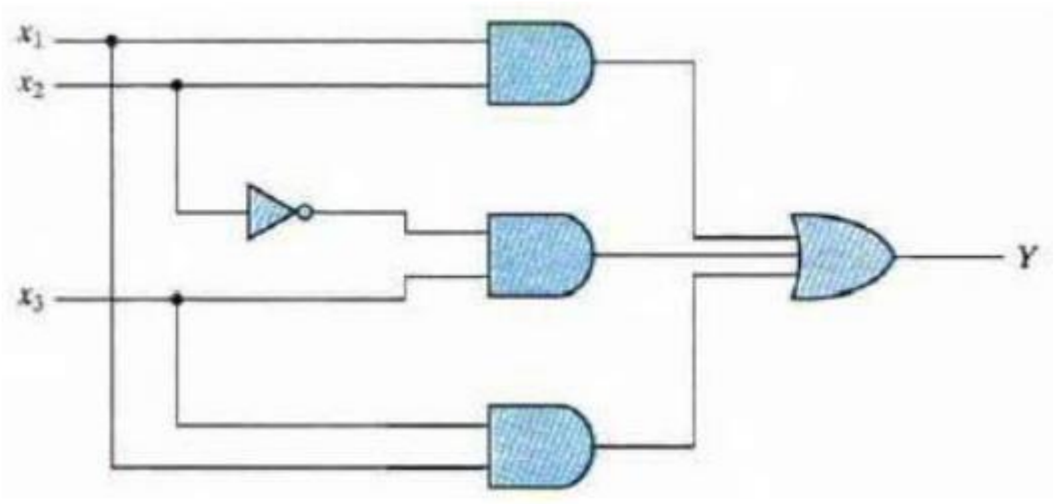


Fig 4.3.4 Hazard free circuit

Essential Hazards

Another type of hazard that may occur in asynchronous sequential circuits is called an **essential hazard**. This type of hazard is caused by unequal delays along two or more paths that originate from the same input. An excessive delay through an inverter circuit in comparison to the delay associated with the feedback path may cause such a hazard.

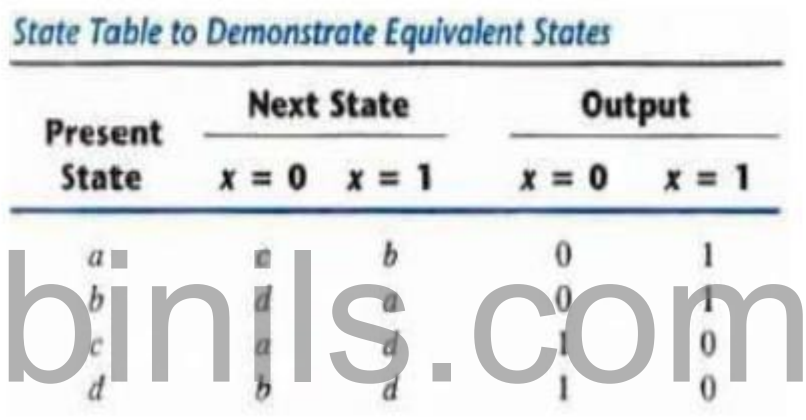
Essential hazards cannot be corrected by adding redundant gates as in static hazards. The problem that they impose can be corrected by adjusting the amount of delay in the affected path. To avoid essential hazards, each feedback loop must be handled with individual care to ensure that the delay in the feedback path is long enough compare d with delays of other signals that originate from the input terminals.

4.4 Reduction of State And Flow Tables

The procedure for reducing the number of internal states in an asynchronous sequential circuit resembles the procedure that is used for synchronous circuits.

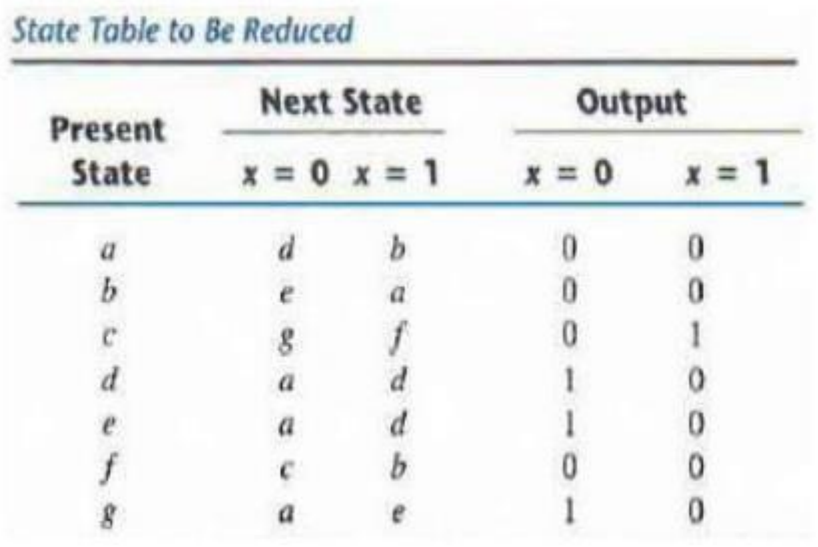
Implication Table and Implied State

The state-reduction procedure for completely specified state tables is based on an algorithm that combines two slates in a slate table into one as long as they can be shown to be equivalent. Two states are equivalent if, for each possible input, they give exactly the same output and go to the same next states or to equivalent next states.


State Table to Demonstrate Equivalent States

Present	Next State		Output	
State	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	*c*	*b*	0	1
b	*d*	*a*	0	1
c	*a*	*d*	1	0
d	*b*	*d*	1	0

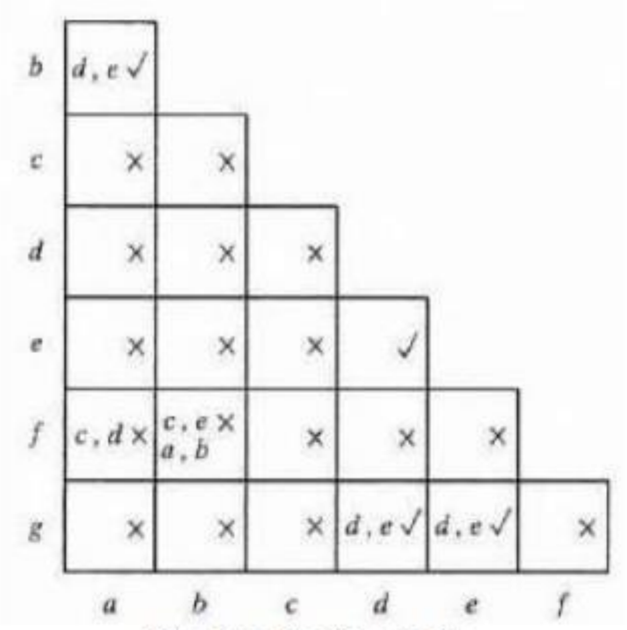
Consider for example the state table shown in above table. The present states a and b have the same output for the same input. Their next states are c and d for x = 0 and b and a for x = 1. If we can show that the pair of states (c, d) are equivalent, then the pair of states (a , b) will also be equivalent, because they will have the same or equivalent next states. When this relationship exists, we say that (a. b) imply (c, d) in the sense that if a and b are equivalent then r and d have to be equivalent. Similarly, from the last two rows of above table, we find that the pair of stales (c, d) implies the pair of states (a, b). The characteristic of equivalent states is that if (a, b) imply (c, d) and (c, d) imply (a, b), then both pairs of states are equivalent that is, a and b are equivalent, and so are c and d. As a consequence, the four rows of table can be reduced to two rows by combining a and b into one state and c and d into a second state.



State Table to Be Reduced

Present State	Next State		Output	
	x = 0	x = 1	x = 0	x = 1
a	d	b	0	0
b	e	a	0	0
c	g	f	0	1
d	a	d	1	0
e	a	d	1	0
f	c	b	0	0
g	a	e	1	0

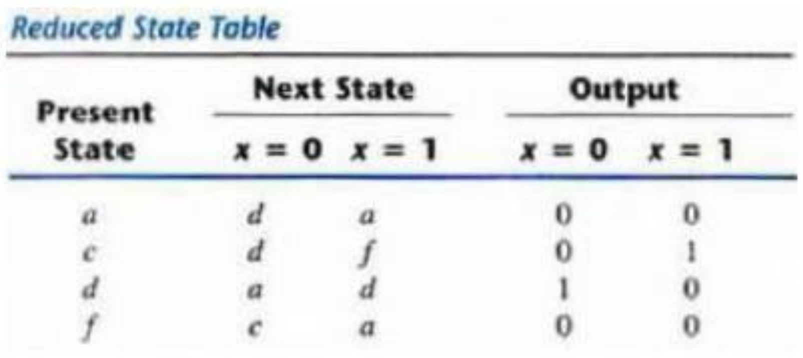
The implication table is shown in Fig. On the left side along the vertical are listed all the states defined in the state table except the first and across the bottom horizontally are listed all the states except the last. The result is a display of all possible combinations of two stares with a square placed in the intersection of a row and a column where the two states can be tested for equivalence. Two states having different outputs for the same input are not equivalent.



b	d, e ✓					
c	×	×				
d	×	×	×			
e	×	×	×	✓		
f	c, d ×	c, e × a, b	×	×	×	
g	×	×	×	d, e ✓	d, e ✓	×
	a	b	c	d	e	f

Fig: Implication table

Two states that are not equivalent are marked with a cross [X] in the corresponding square whereas their equivalence is recorded with a check mark('Dik'). Some of the squares have entries of implied states that must be investigated further to determine whether they are equivalent. Thus table can be reduced from seven states to four one for each member of the preceding partition. The reduced state table is obtained by replacing state b by a and states e and g by d and it is shown below,



Reduced State Table

Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	d	a	0	0
c	d	f	0	1
d	a	d	1	0
f	c	a	0	0

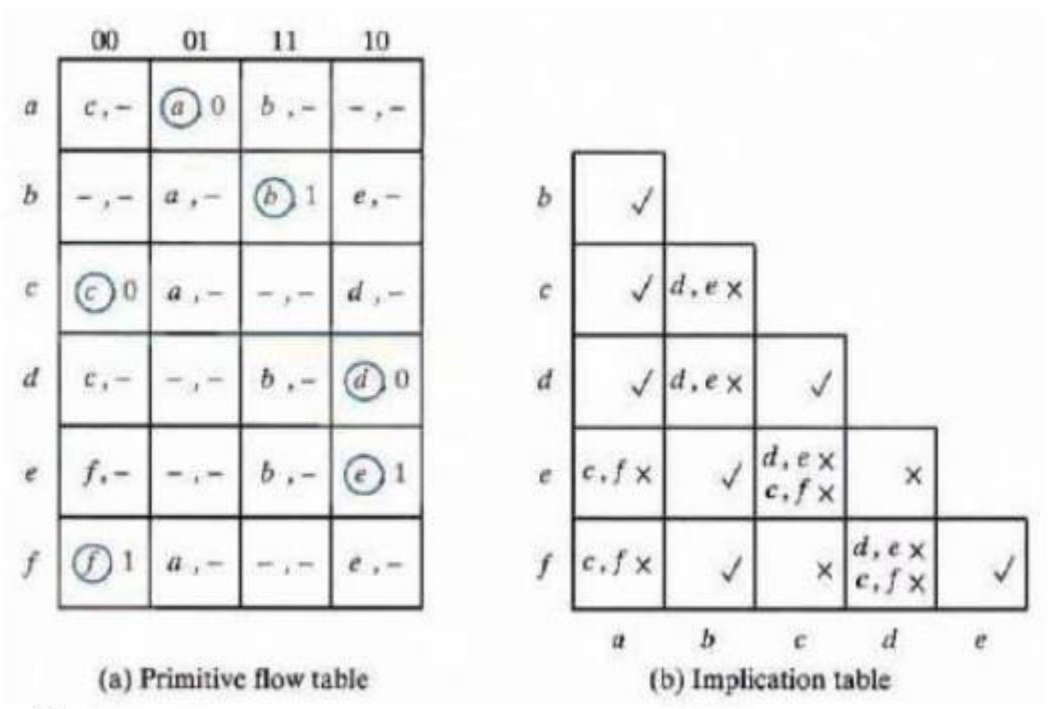
Merging of the Flow Table

Incompletely specified states can be combined to reduce the number of state in the flow table. Such stares cannot be called equivalent because the formal definition of equivalence requires that all outputs and next states be specified for all inputs. Instead, two incompletely specified states that can be combined are said to be *Compatible.* The process that must be applied in order to find a suitable group of compatibles for the purpose of merging a flow table can be divided into three steps:

1. Determine all compatible pairs by using the implication table.
2. Find the maximal compatibles with the use of a merge r diagram.
3. Find a minimal collection of compatibles that covers all the states and is closed.

Compatible Pairs

The entries in each square of primitive flow table represent the next state and output The dashes represent the unspecified states or outputs. The implication table is used to fmd compatible states just as it is used to find equivalent stales in the completely specified case. The only difference is that, when comparing rows, we are at liberty to adjust the dashes to fit any desired condition.



	00	01	11	10
a	$c,-$	$\textcircled{a}$ 0	$b,-$	$-,-$
b	$-,-$	$a,-$	$\textcircled{b}$ 1	$e,-$
c	$\textcircled{c}$ 0	$a,-$	$-,-$	$d,-$
d	$c,-$	$-,-$	$b,-$	$\textcircled{d}$ 0
e	$f,-$	$-,-$	$b,-$	$\textcircled{e}$ 1
f	$\textcircled{f}$ 1	$a,-$	$-,-$	$e,-$

(a) Primitive flow table

b	✓				
c	✓	d,e ×			
d	✓	d,e ×	✓		
e	c,f ×	✓	d,e × c,f ×	×	
f	c,f ×	✓	×	d,e × c,f ×	✓
	a	b	c	d	e

(b) Implication table

The compatible pairs are,
$(a,b)(a,c)(a,d)(b,e)(b,f)(c,d)(e,f)$

Maximal Compatibles

The maximal compatible is a group of compatibles that contains all the possible combinations of compatible states. The maximal compatible can be obtained from a merger diagram. The merger diagram is a graph in which each state is represented by a dot placed along the circumference of a circle. Lines are drawn between any two corresponding dots that form a compatible pair. All possible compatibles can be obtained from the merger diagram by observing the geometrical patterns in which states are connected to each other. An isolated dot represents a state that is not compatible with any other state. A line represents a compatible pair. A triangle constitutes a compatible with three states.

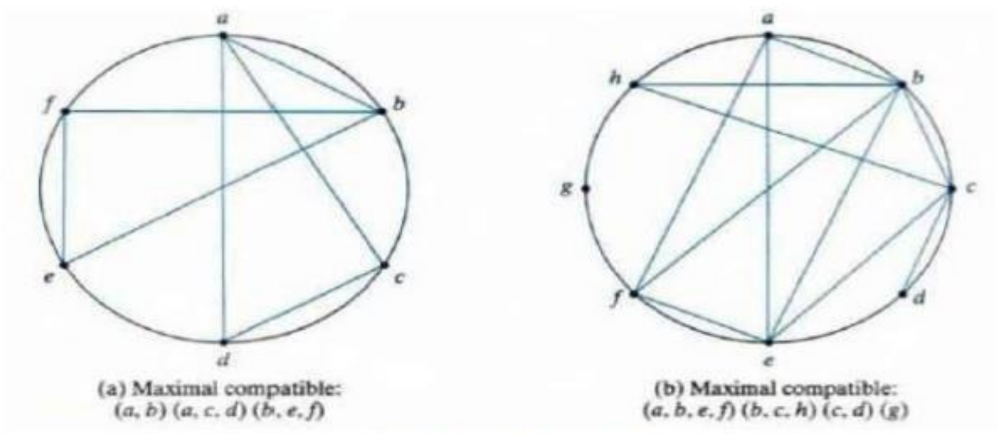
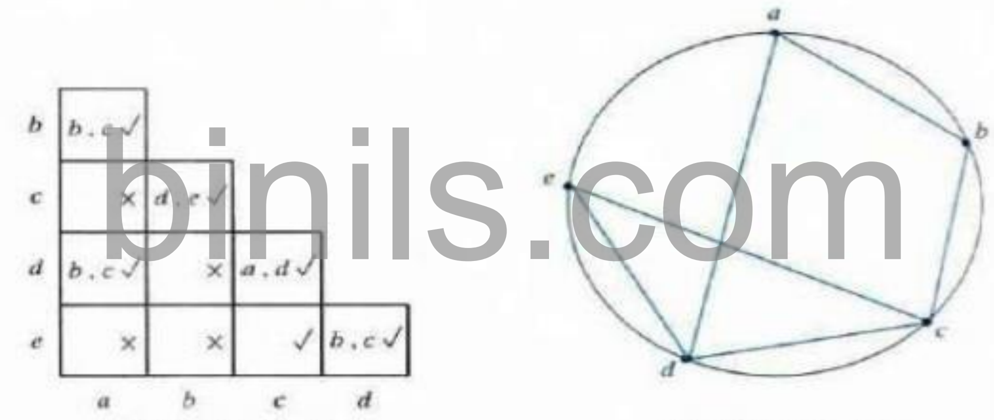


Fig: Merger diagrams

The maximal compatibles of fig (a) are$(a,b)(a,c,d)(b,e,f)$

The maximal compatibles of fig (b) are$(a,b,e,f)(b,c,h)(c,d)(g)$

Closed-Covering Condition



b	b , c ✓			
c	×	d , e ✓		
d	b , c ✓	×	a , d ✓	
e	×	×	✓	b , c ✓
	a	b	c	d

(a) Implication table

(b) Merger diagram

Compatibles	(a, b)	(a, d)	(b, c)	(c, d, e)
Implied states	(b, c)	(b, c)	(d, e)	(a, d) (b, c)

(c) Closure table

The condition that must be satisfied for merging rows is that the set of chosen compatibles must cover all the states and must be closed. The set will cover all the states if it includes all the states of the original state table. The closure condition is satisfied if there are no implied states or if the implied states are included within the set. A closed set of compatibles that covers all the states is called a closed covering.

4.5 Memory and Programmable Logic

A **memory unit** is a device to which binary information is transferred for storage and from which information is retrieved when needed for processing. When data processing takes place, information from memory is transferred to selected registers in the processing unit. A memory unit is a collection of cells capable of storing a large quantity of binary information.

Communication between memory and its environment is achieved through data input and output lines, address selection lines, and control lines that specify the direction of transfer.

Types of Memories

There are two types of memories that are used in digital systems: random-access memory (RAM) and read-only memory (ROM) The process of storing new information into memory is referred to as a memory “write” operation. The process of transferring the stored information out of memory is referred to as a memory “read” operation. RAM can perform both write and read operations. ROM can perform only the read operation. This means that suitable binary information is already stored inside memory and can be retrieved or read at any time. However, that information cannot be altered by writing.

ROM is one example of a PLD. Other such units are the programmable logic array (PLA) - Programmable array logic (PAL), and the field -programmable gate array (FPGA). A PLD is an integrated circuit with internal logic gates connected through electronic paths that behave similarly to fuses.

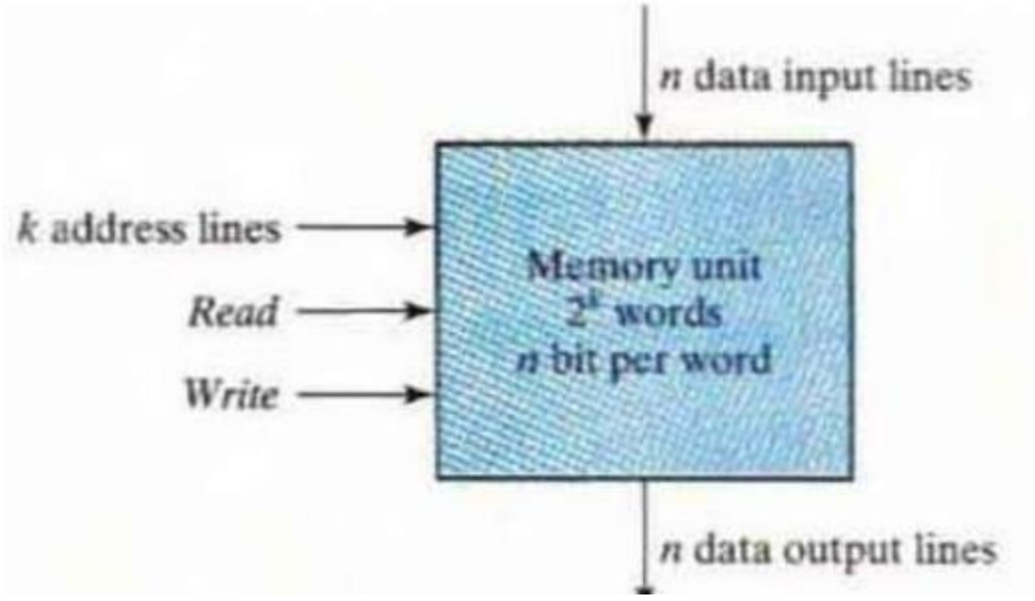


Fig 4.5.1 Block diagram of a memory unit.

The n data input lines provide the information to be stored in memory and the n data output lines supply the information coming out of memory. The k address lines specify the particular word chosen among the many available. The two control inputs specify the direction of transfer desired: The Write input causes binary data to be transferred into the memory and the Read input causes binary data to be transferred out of memory.

A typical PLD may have hundreds to millions of gates interconnected through hundreds to thousands of internal paths. Instead of having multiple input lines into the gate, we draw a single line entering the gate. The input lines are drawn perpendicular to this single line and are connected to the gate through internal fuses as shown in the figure. In a similar fashion, we can draw the array logic for an AND gate.

4.6 Read Only Memory (ROM)

A ROM is essentially a memory device in which permanent binary information is stored.

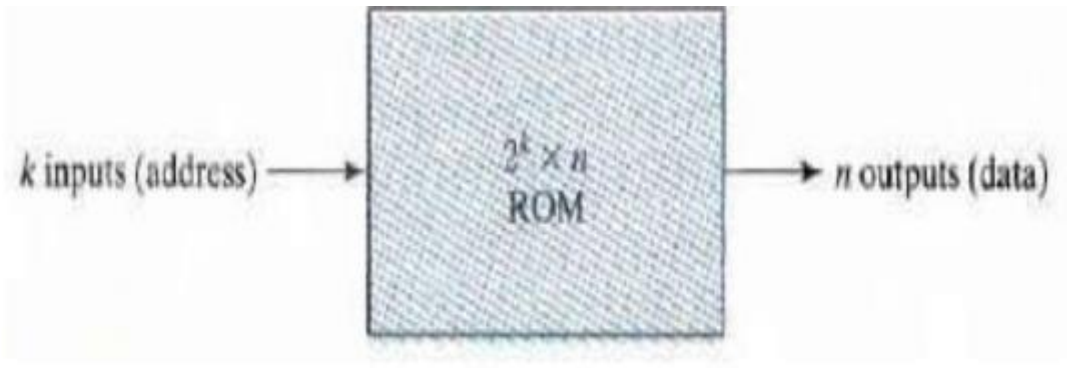


Fig 4.6.1 ROM

The inputs provide the address for memory and the outputs give the data bits of the stored word that is selected by the address. The number of words in a ROM is determined from the fact that k address input lines are needed to specify 2^k words.

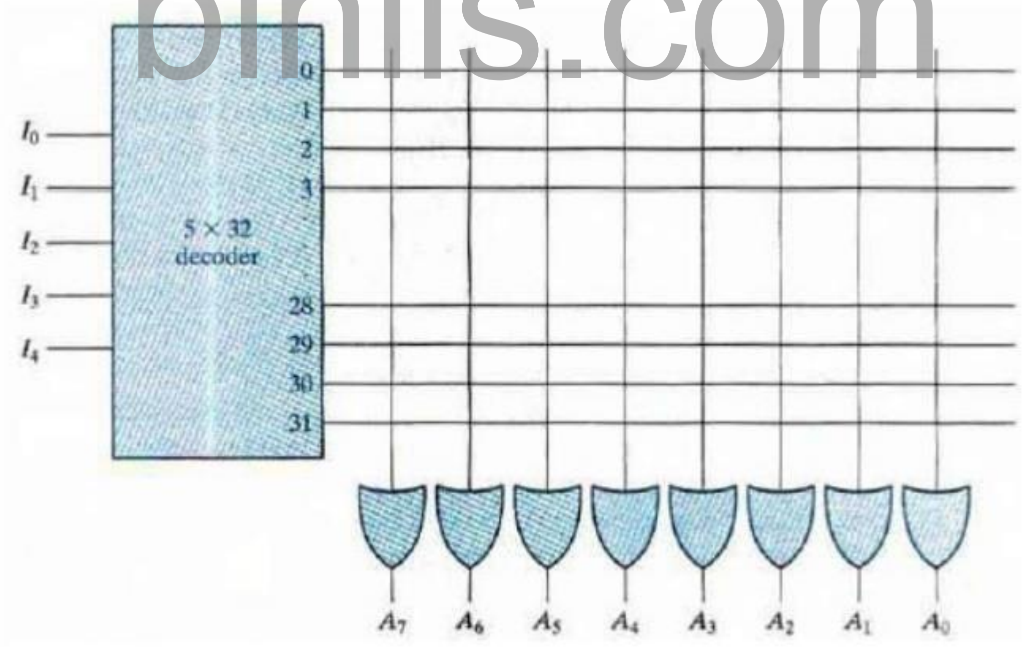
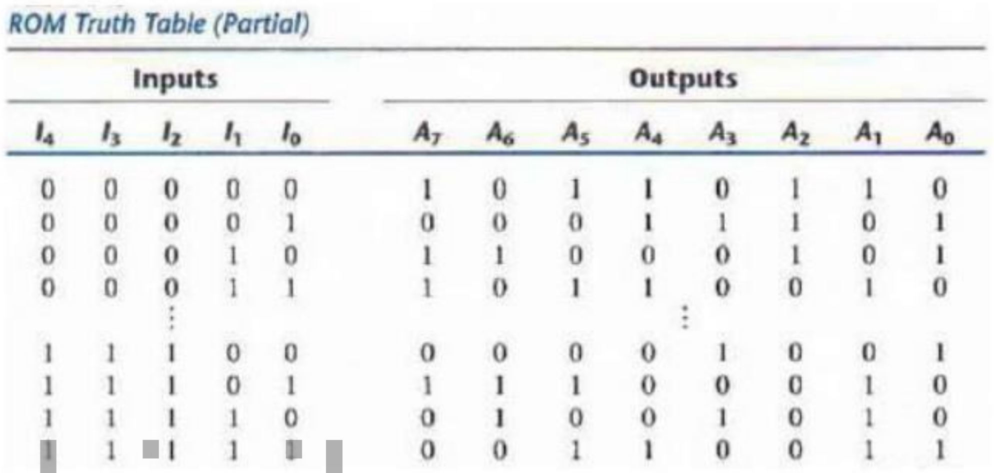


Fig 4.6.2 Internal logic of a 32× 8 ROM ×

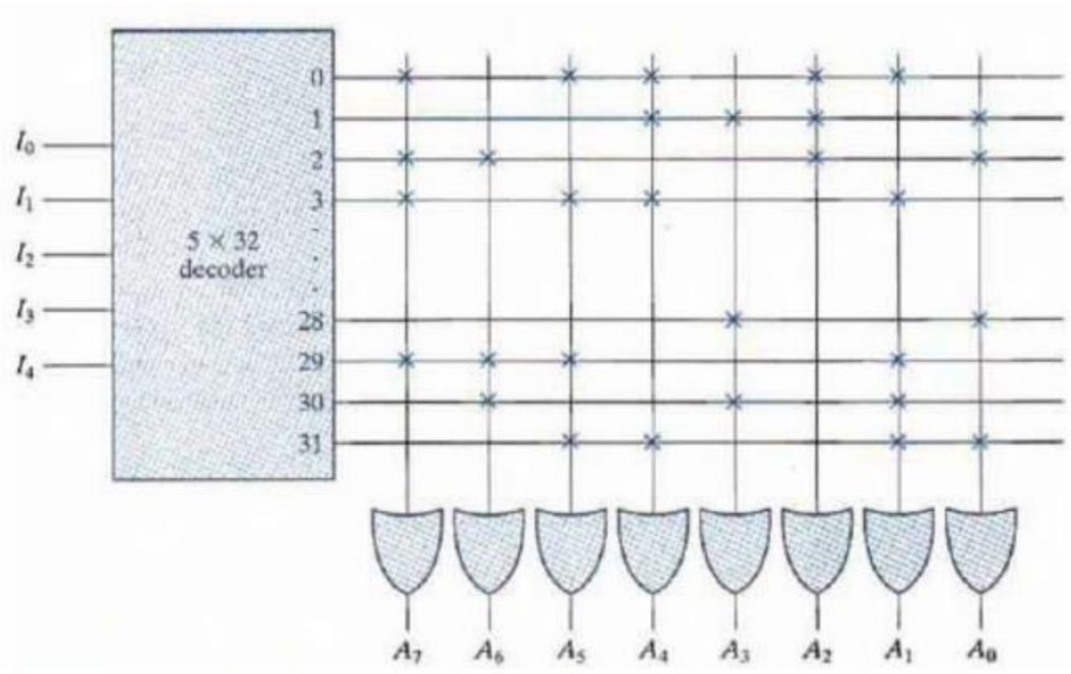
The five inputs are decoded into 32 distinct outputs by means of a 5 32 decoder. Each output of the decoder represents a memory address. The 32 outputs of the decoder are connected to each of the eight OR gates. Each OR gate must be considered as having 32 inputs. Each output of the decoder is connected to one of the inputs of each OR gate. Since each OR gate has 32 input connections and there are 8 OR gates, the ROM contains 32 x 8 = 256 internal connections.

A programmable connection between two lines is logically equivalent to a switch that can be altered to be either closed (meaning that the two lines are connected) or open (meaning that the two lines are disconnected). The programmable intersection between two lines is sometimes called a cross point.


ROM Truth Table (Partial)

Inputs					Outputs							
I_4	I_3	I_2	I_1	I_0	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
0	0	0	0	0	1	0	1	1	0	1	1	0
0	0	0	0	1	0	0	0	1	1	1	0	1
0	0	0	1	0	1	1	0	0	0	1	0	1
0	0	0	1	1	1	0	1	1	0	0	1	0
		⋮						⋮				
1	1	1	0	0	0	0	0	0	1	0	0	1
1	1	1	0	1	1	1	1	0	0	0	1	0
1	1	1	1	0	0	1	0	0	1	0	1	0
1	1	1	1	1	0	0	1	1	0	0	1	1

For example, programming the ROM according to the truth table given by table. Every 0 listed in the truth table specifies the absence of a connection and every 1 listed specifies a path that is obtained by a connection.



Types of ROMs

The required paths in a ROM may be programmed in four different ways.

The first is called mask programming and is done by the semiconductor company during the last fabrication process of the unit. This procedure is costly because the vendor charges the customer a special fee for custom masking the particular ROM.

For small quantities, it is more economical to use a second type of ROM called programmable read-only memory- PROM. The fuses in the PROM are blown by the application of a high-voltage pulse to the device through a special pin. A blown fuse defines a binary 0 state and an intact fuse gives a binary 1 state. The hardware procedure for programming ROMs or PROMs is irreversible and once programmed, the fixed pattern is permanent and cannot be altered.

A third type of R OM is the erasable PROM or EPROM, which can be restructured to the initial state even though it has been programmed previously. When the EPROM is placed under a special ultraviolet light for a given length of time. the shortwave radiation discharges the intern al floating gates that serve as the programmed connections. After erasure, the EPROM returns to its initial state and can be reprogrammed to a new set of values.

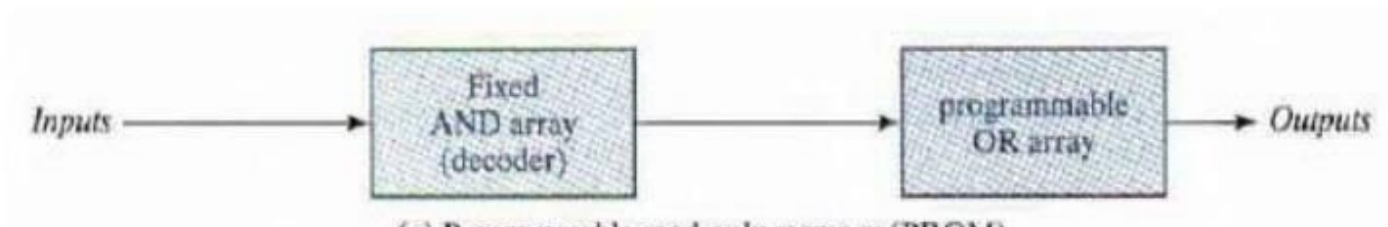
The fourth type of ROM is the electrically erasable PROM (EEPROM or E^{2}PROM). This device is like the EPROM except that the previously programmed connections can be erased with an electrical signal instead of ultraviolet light. The advantage is that the device can be erased without removing it from its socket.

Flash memory devices are similar to EEPROMs, but have additional built-in circuitry to selectively program and erase the device in-circuit, without the need for a special programmer. They have widespread application in modern technology in cell phones, digital cameras, set-top boxes, digital TV, telecommunications, non volatile data storage and microcontrollers. Their low consumption of power makes them an attractive storage medium for laptop and notebook computers.

Combinational PLDs

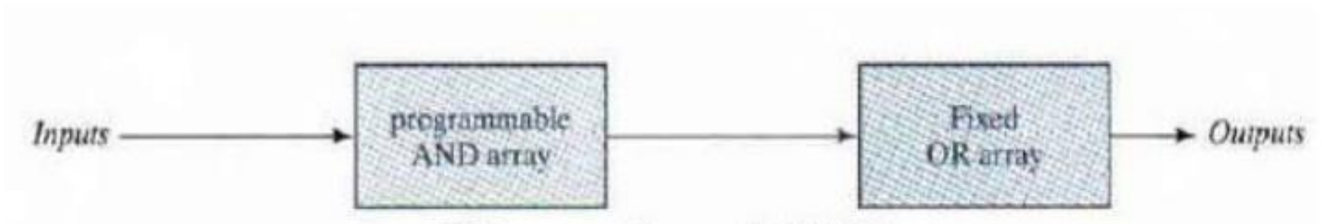
The PROM is a combinational programmable logic device (PLD)-an integrated circuit with programmable gates divided into an AND array and an OR array to provide an AND-OR sum of-product implementation.

There are three major types of combinational PLDs, differing in the placement of the programmable connections in the AND-OR array.



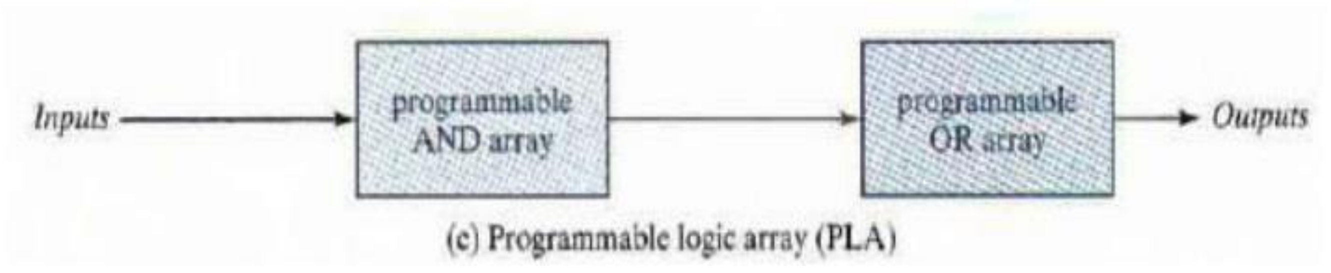
(a) Programmable read-only memory (PROM)

The PROM has a fixed AND array constructed as a decoder and a programmable OR array. The programmable OR gates implement the Boolean functions in sum-of-mintenns form.



(b) Programmable array logic (PAL)

The PAL has a programmable AND array and a fixed OR array. The AND gates are programmed to provide the product terms for the Boolean functions, which are logically summed in each OR gate.



(c) Programmable logic array (PLA)

The most flexible PLD is the PLA, in which both the AND and OR arrays can be programmed. The product terms in the AND array may be shared by any OR gate to provide the required sum-of-products implementation.

4.7 Programmable Logic Devices

There are a wide variety of ICs that can have their logic function "programmed" into them after they are manufactured. Most of these devices use technology that also allows the function to be reprogrammed, which means that if you find a bug in your design.

Programmable logic arrays (PLAs) were the first programmable logic devices. PLAs contained a two-level structure of AND and OR gates with user-programmable connections. Using this structure, a designer could accommodate any logic function up to a certain level of complexity using the well-known theory of logic synthesis and minimization

PLA structure was enhanced and PLA costs were reduced with the introduction of programmable array logic (PAL) devices. Today, such devices are generically called programmable logic devices (PLDs), and are the "MSI" of programmable logic industry.

The PLA is similar in concept to the PROM, except that the PLA does not provide full decoding of the variables and does not generate all the minterms. The decoder is replaced by an array of AND gates that can be programmed to generate any product term of the input variables. The product terms are then connected to OR gates to provide the sum of products for the required Boolean functions. The output is inverted when the XOR input is connected to

0 (since x $\oplus$ 0 = x).

$$F_1 = A\bar{B} + AC + \bar{A}B\bar{C}$$

$$F_2 = \overline{(AC + BC)}$$

The fuse map of a PLA can be specified in a tabular form. The first section lists the product terms numerically. The second section specifies the required paths between inputs and AND gates. The third section specifies the paths between the AND and OR gates. For each output variable, we may have a T'(for true) or C (for complement) for programming the XOR gate.

For each product term, the inputs are marked with 1, 0, or - (dash). If a variable in the product term appears in the form in which it is true, the corresponding input variable is marked with a 1. If it appears complemented, the corresponding input variable is marked with a 0. If the variable is absent from the product term, it is marked with a dash.

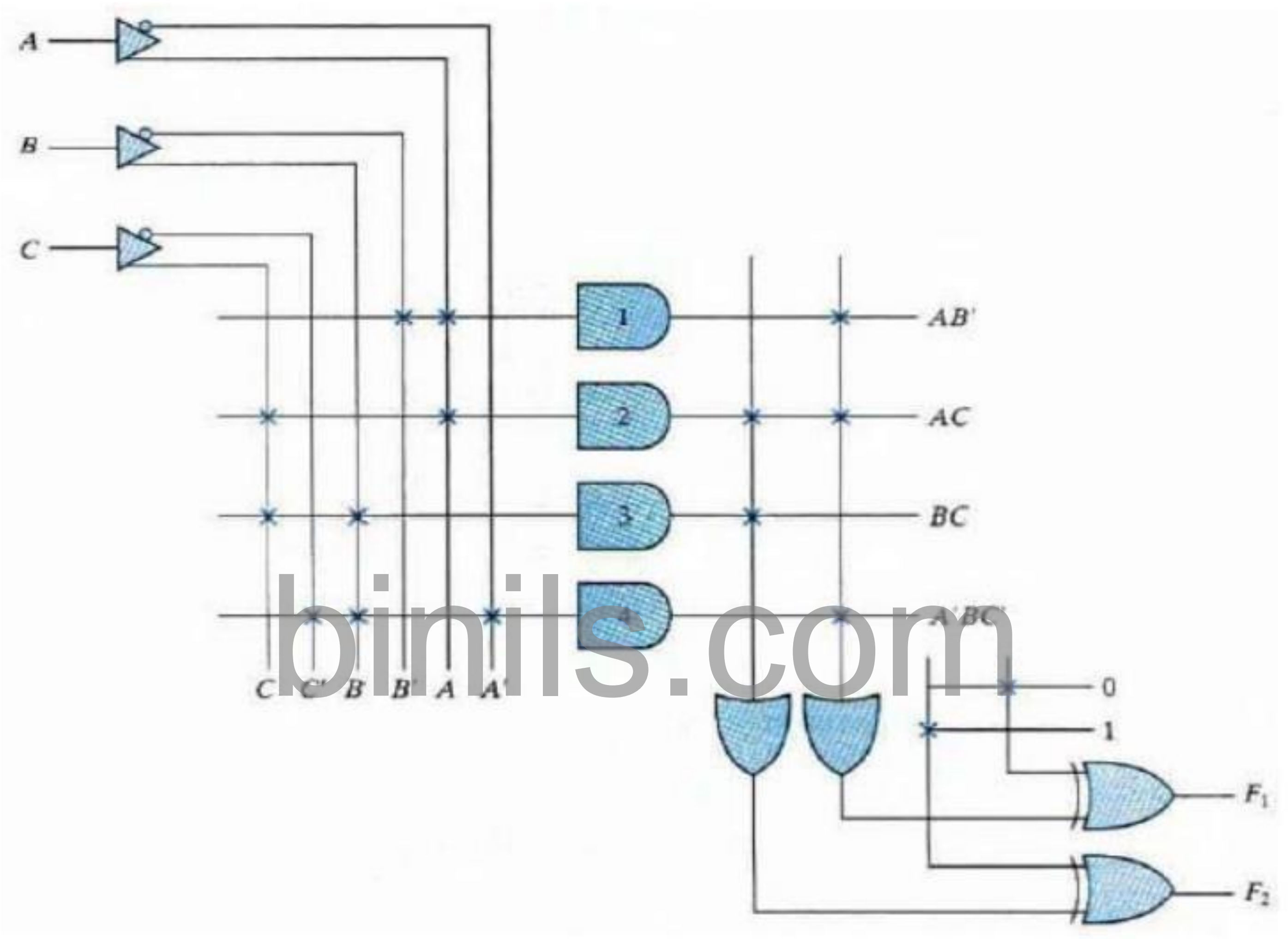
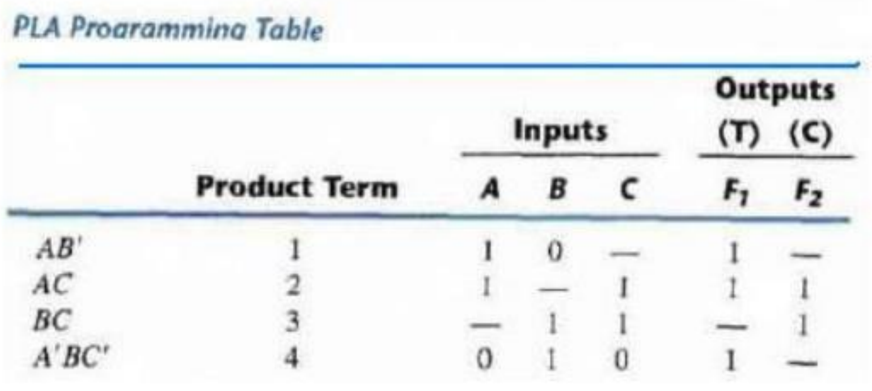


Fig 4.7.1 PLA



PLA Programming Table

		Inputs			Outputs	
					(T)	(C)
	Product Term	A	B	C	F_1	F_2
AB'	1	1	0	—	1	—
AC	2	1	—	1	1	1
BC	3	—	1	1	—	1
$A'BC'$	4	0	1	0	1	—

Programmable Array Logic

The PAL is a programmable logic device with a fixed OR array and a programmable AND array. Because only the AND gates are programmable, the PAL is easier to program than but is not as flexible as the PLA.

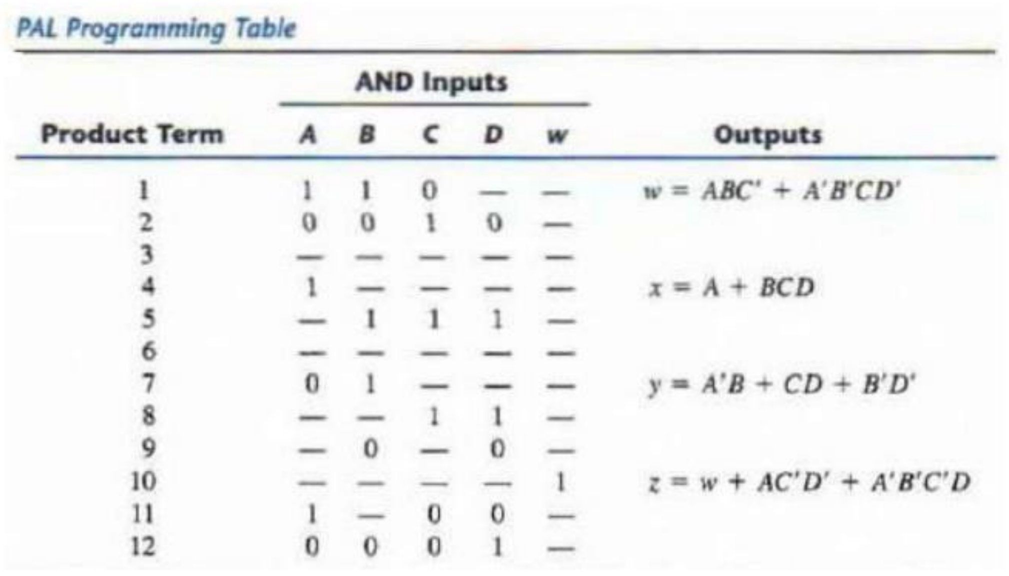
There are four sections in the unit each composed of an AND-OR array that is three wide, the term used to indicate that there are three programmable AND gates in each section and one fixed OR gate. Each AND gate has 10 programmable input connections, shown in the diagram by 10 vertical lines intersecting each horizontal line. The horizontal line symbolizes the multiple-input configuration of the AND gate. One of the outputs is connected to a buffer-inverter gate and then fed back into two inputs of the AND gates.

As an example of using a PAL in the design of a combinational circuit, consider me following Boolean functions, given in sum-of-minterms form:

$$\begin{aligned}
w(A, B, C, D) &= \Sigma(2, 12, 13)\\
x(A, B, C, D) &= \Sigma(7, 8, 9, 10, 11, 12, 13, 14, 15)\\
y(A, B, C, D) &= \Sigma(0, 2, 3, 4, 5, 6, 7, 8, 10, 11, 15)\\
z(A, B, C, D) &= \Sigma(1, 2, 8, 12, 13)
\end{aligned}$$

Simplifying the four functions to a minimum number of terms results in the following Boolean functions:

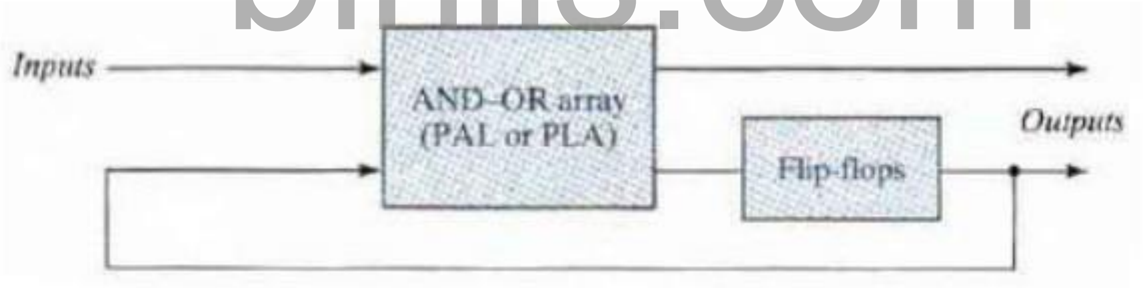
$$\begin{aligned}
w &= ABC' + A'B'CD'\\
x &= A + BCD\\
y &= A'B + CD + B'D'\\
z &= ABC' + A'B'CD' + AC'D' + A'B'C'D\\
&= w + AC'D' + A'B'C'D
\end{aligned}$$


PAL Programming Table

Product Term	A	B	C	D	w	Outputs
	AND Inputs					
1	1	1	0	—	—	$w = ABC' + A'B'CD'$
2	0	0	1	0	—	
3	—	—	—	—	—	
4	1	—	—	—	—	$x = A + BCD$
5	—	1	1	1	—	
6	—	—	—	—	—	
7	0	1	—	—	—	$y = A'B + CD + B'D'$
8	—	—	1	1	—	
9	—	0	—	0	—	
10	—	—	—	—	1	$z = w + AC'D' + A'B'C'D$
11	1	—	0	0	—	
12	0	0	0	1	—	

Sequential Programmable Devices

Digital systems are designed with flip-flops and gates. Since the combinational PLD consists of only gates, it is necessary to include external flip-flops when they are used in the design. Sequential programmable devices include both gates and flip-flops.



1. Sequential (or simple) programmable logic device (SPLD)
2. Complex programmable logic device (CPLD)
3. Field-programmable gate array (FPGA)

The ever-increasing capacity of integrated circuits created an opportunity for IC manufacturers to design larger PLDs for larger digital-design applications. Instead, IC manufacturers devised complex PLD (CPLD) architectures to achieve the required scale. A typical CPLD is merely a collection of multiple PLDs and an interconnection structure, all on the same chip. In addition to the individual PLDs, the on-chip interconnection structure is also programmable, providing a rich variety of design possibilities. CPLDs can be scaled to larger sizes by increasing the number of individual PLDs and the richness of the interconnection structure on the CPLD chip.

At about the same time that CPLDs were being invented, other IC manufacturers took a different approach to scaling the size of programmable logic chips. Compared to a CPLD, a field-programmable gate arrays (FPGA) contains a much larger number of smaller individual logic blocks, and provides a large, distributed interconnection structure that dominates the entire chip.

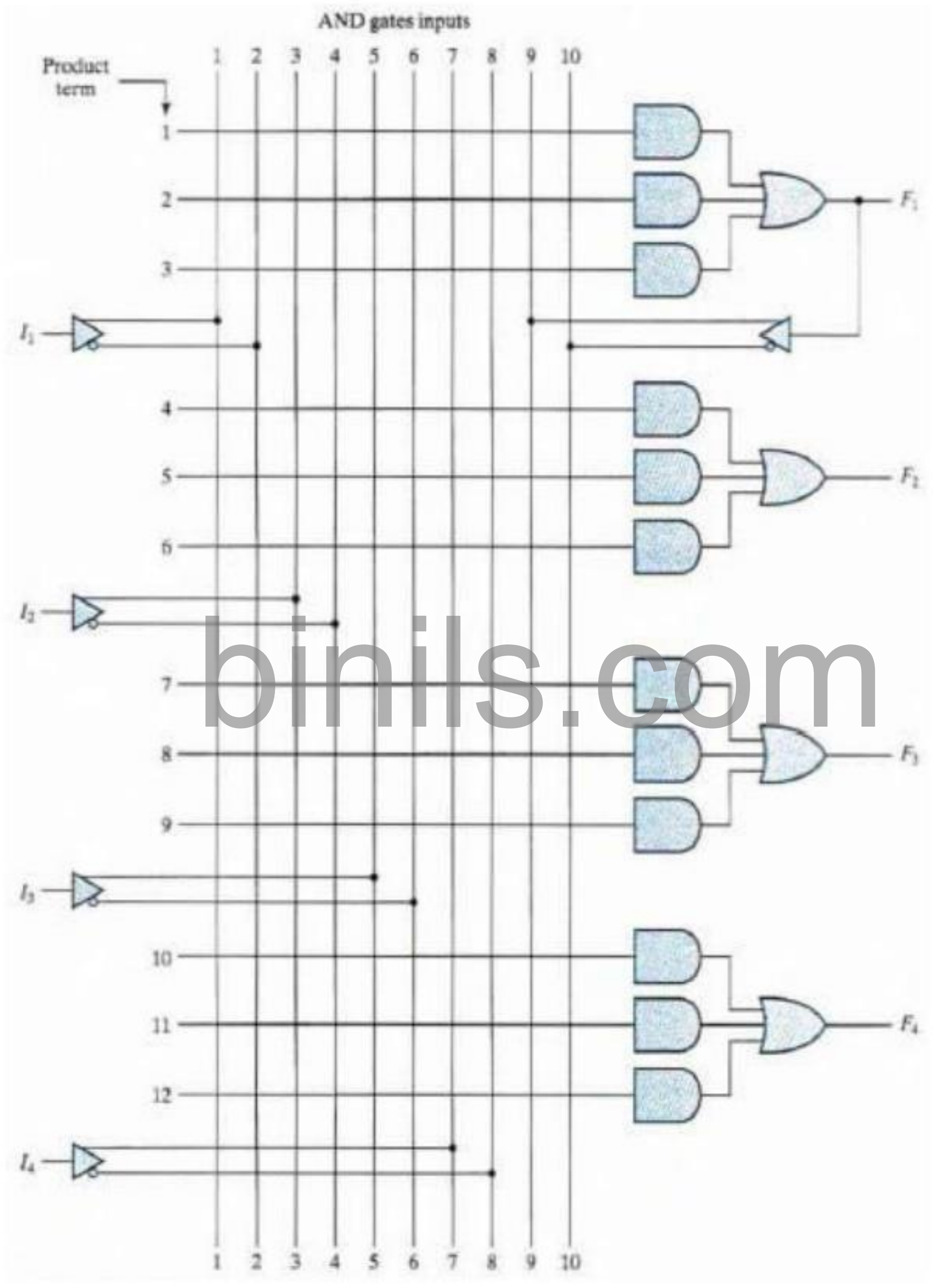


Fig: PAL with four inputs, four outputs and three AND-OR structure

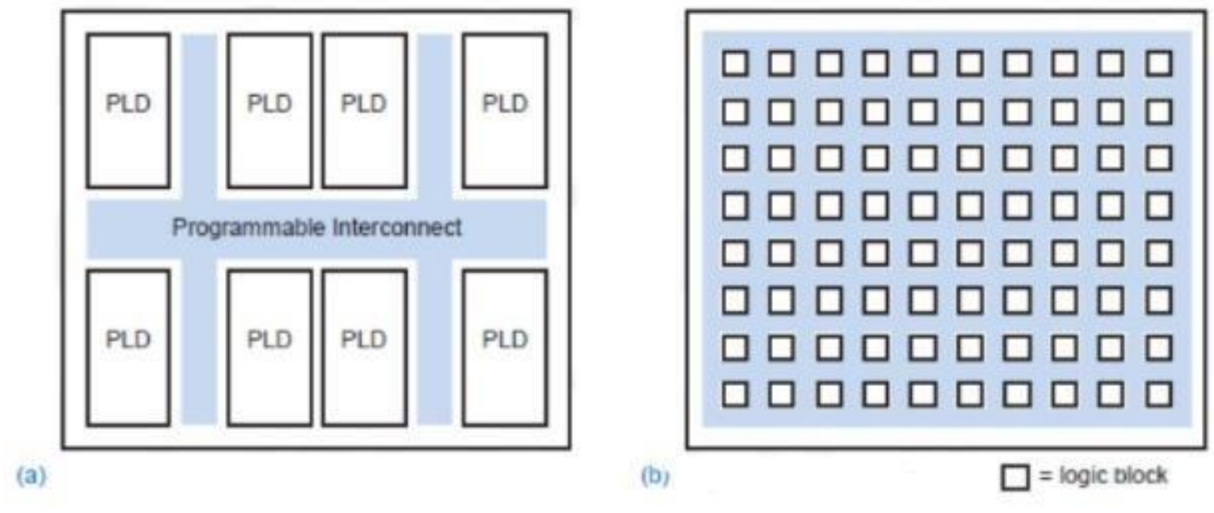


Fig 4.7.1 a) CPLD b) FPGA

Field-Programmable Gate Arrays

Even larger devices, often called *field-programmable gate arrays (FPGAs)*, use read/write memory cells to control the state of each connection. The read/write memory cells are volatile— they do not retain their state when power is removed. Therefore, when power is first applied to the FPGA, all of its read/write memory must be initialized to a state specified by a separate, external nonvolatile memory. This memory is typically either a programmable read-only memory (PROM) chip attached directly to the FPGA or it's part of a microprocessor subsystem that initializes the FPGA as part of overall system initialization.