

**MALLA REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY**  
**DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING**

**DIGITAL NOTES ON**  
**DIGITAL SYSTEM DESIGN**  
**FOR**  
**III/IV B.TECH I SEMESTER**

**PREPARED BY:**  
**PATIBANDLA ANITHA**  
**ASSOCIATE PROFESSOR,**  
**DEPT. OF ECE**

III Year B.Tech I Sem

L T/P/D C  
3 1/-/- 3**CORE ELECTIVE-I  
(R15A0411)DIGITAL SYSTEM DESIGN****OBJECTIVES:**

This course provides in depth knowledge digital system design of digital circuits, which is the basis for design of any digital circuit. The main objectives are:

- To design and analysis of sequential circuits.
- To impart to student the concepts of sequential circuits, enabling them to analyze sequential systems in terms of state machines.
- To understand about the SM charts and their realization
- To implement synchronous state machines using flip-flops.
- To detect the fault models in sequential circuits.

**UNIT -I: Minimization and Transformation of Sequential Machines:** The Finite State Model – Capabilities and limitations of FSM –State equivalence and machine minimization – Simplification of incompletely specified machines-Merger chart methods-Concept of Minimal Cover Table-Compatibility Graph.

**UNIT -II: Fundamental mode model** –Flow table –State reduction –Excitation and output Tables-Primitive Flow Table-Hazards-Design of Hazard free circuits.

**UNIT III: Digital Design:** Digital Design Using ROMs, PALs, BCD Adder, 32 –bit adder-PLA-PLA minimization-PLA Folding-Simple column folding-Problems.

**UNIT -IV: Faults in Digital Circuits:** Failures and Faults-Modelling of Faults-Single stuck at fault model –Multiple stuck at fault models –Stuck Open Faults-Bridging fault model. Fault diagnosis of combinational circuits by conventional methods –Path sensitization techniques, Boolean Difference method –Kohavi algorithm-examples.

**UNIT -V: SM Charts:** State machine charts, Derivation of SM Charts, Realization of SM Chart, Implementation of Dice Game, and Binary Multiplier.

**TEXT BOOKS:**

1. Fundamentals of Logic Design –Charles H. Roth, 5<sup>th</sup> Ed., Cengage Learning.
2. Switching Theory and Logic Design –A. Anand Kumar, PHI
3. Logic Design Theory –N. N. Biswas, PHI

**REFERENCE BOOKS:**

1. Switching and Finite Automata Theory –Z. Kohavi , 2 nd Ed., 2001, TMH
2. Digital Design –Morris Mano, M.D.Ciletti, 4th Edition, PHI.
3. Digital Circuits and Logic Design –Samuel C. Lee , PHI
4. Fault tolerant and fault testable hardware design Parag K. Lala

**OUTCOMES**

Upon completion of the course, the student will be able to:

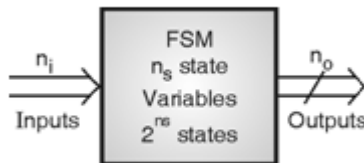
- Design and analysis of sequential circuits.
- Understand the concepts of sequential circuits, enabling them to analyze sequential systems in terms of state machines.
- Understand about the SM charts and their realization
- Implement synchronous state machines using flip-flops.
- Detect the fault models in sequential circuits.

## UNIT -I

### Minimization and Transformation of Sequential Machines

#### Finite State Machines

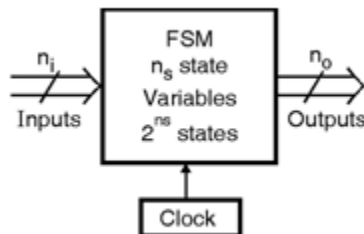
For a sequential logic system number of outputs (no) depend on the present and past, values of the inputs. Sequential logic systems are known as finite-state machines (FSMs). FSMs are considered to have a number of internal states, which are determined by some combination of values of the  $n_s$  state variables if— The FSM changes to a new state depending upon the present state and the inputs. The outputs depend on the present state and the inputs (Mealy machine) or just the present state (Moore machine).



There are two types of FSMs, synchronous FSM and asynchronous FSM.

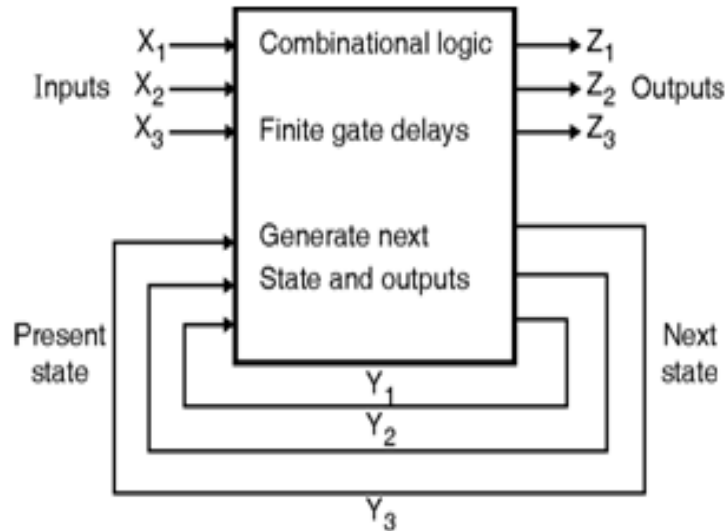
#### Synchronous FSM:

(The operation of a synchronous FSM is carried out by using a clock. At each clock event the state changes to a new state which is determined by the present state and inputs.



#### Asynchronous FSM:

Asynchronous sequential systems do not have clock and the internal states changes depending upon the change in inputs. Asynchronous FSMs are mainly used where a fast response to input changes. Asynchronous FSMs are also used where the introduction of extra frequency components related to the clock should be avoided.



FSM is a type of sequential circuit which is designed to sequence through the finite states in a predetermined sequential manner. An FSM consists of three parts:

- (1) Sequential current state register
- (2) Combinational next state logic
- (3) Combinational output logic

(1) Sequential current state register:

In this register set of  $n$ -bit flip-flops are used and are clocked by clock signal to hold the state vector of the FSM. For the state vector of  $n$ -bit  $2^n$  possible binary patterns are used for state encoding.

(2) Combinational next state logic:

As we know that, the FSM stays in a single state and at each active transition it changes from the current state to the next state. The next state is always a function of the inputs and its current state.

(3) Combinational output logic:

Outputs in FSM seem to be the function of the current state and primary inputs. Generally in a Moore FSM, the user wants to derive the outputs from the next state. We know that

synchronous sequential circuits change (affect) their states for every positive (or negative) transition of the clock signal based on the input. So, this behavior of synchronous sequential circuits can be represented in the graphical form and it is known as state diagram.

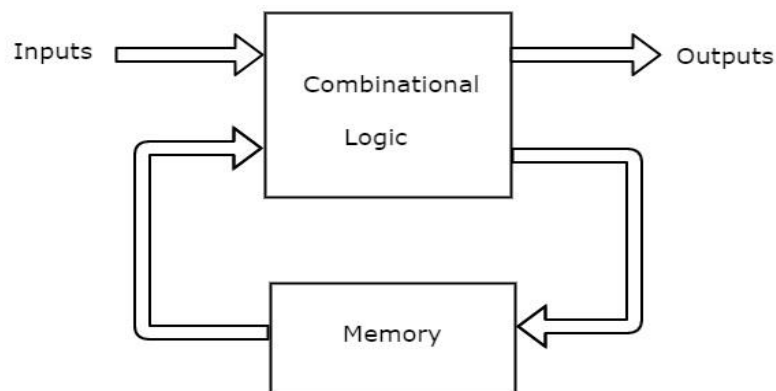
A synchronous sequential circuit is also called as **Finite State Machine (FSM)**, if it has finite number of states. There are two types of FSMs.

- Mealy State Machine
- Moore State Machine

Now, let us discuss about these two state machines one by one.

### Mealy State Machine

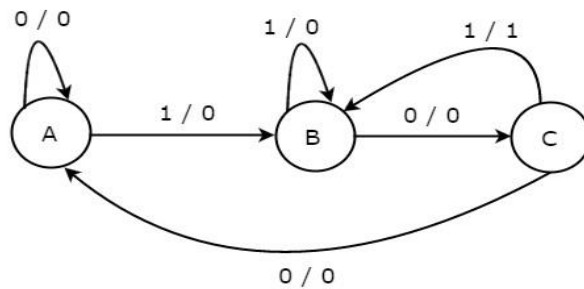
A Finite State Machine is said to be Mealy state machine, if outputs depend on both present inputs & present states. The **block diagram** of Mealy state machine is shown in the following figure.



As shown in figure, there are two parts present in Mealy state machine. Those are combinational logic and memory. Memory is useful to provide some or part of previous outputs (**present states**) as inputs of combinational logic.

So, based on the present inputs and present states, the Mealy state machine produces outputs. Therefore, the outputs will be valid only at positive (or negative) transition of the clock signal.

The **state diagram** of Mealy state machine is shown in the following figure.

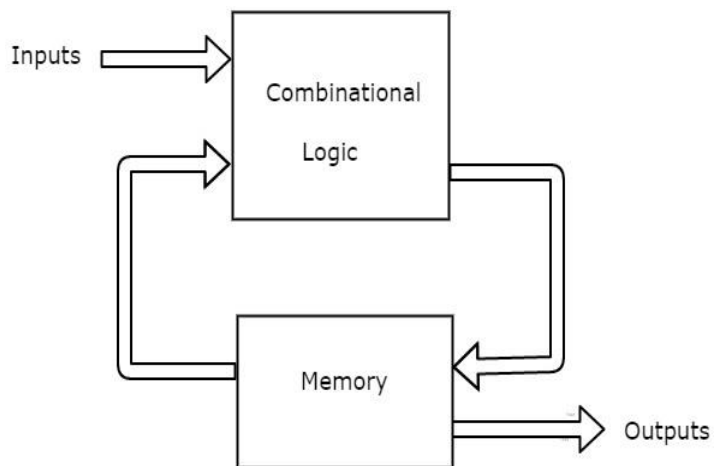


In the above figure, there are three states, namely A, B & C. These states are labelled inside the circles & each circle corresponds to one state. Transitions between these states are represented with directed lines. Here, 0 / 0, 1 / 0 & 1 / 1 denotes **input / output**. In the above figure, there are two transitions from each state based on the value of input, x.

In general, the number of states required in Mealy state machine is less than or equal to the number of states required in Moore state machine. There is an equivalent Moore state machine for each Mealy state machine.

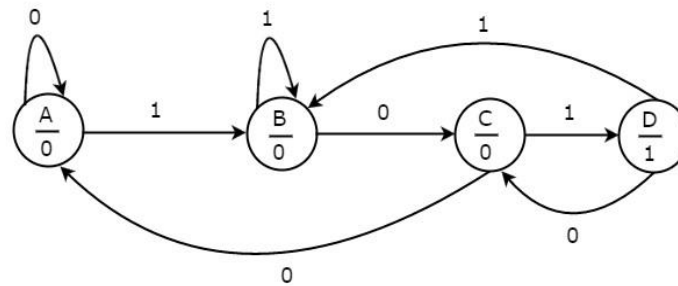
### Moore State Machine

A Finite State Machine is said to be Moore state machine, if outputs depend only on present states. The **block diagram** of Moore state machine is shown in the following figure.



As shown in figure, there are two parts present in Moore state machine. Those are combinational logic and memory. In this case, the present inputs and present states determine the next states. So, based on next states, Moore state machine produces the outputs. Therefore, the outputs will be valid only after transition of the state.

The **state diagram** of Moore state machine is shown in the following figure.



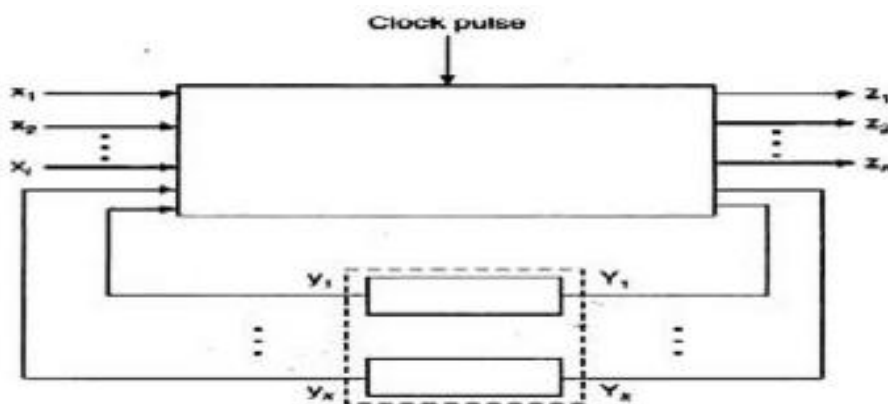
In the above figure, there are four states, namely A, B, C & D. These states and the respective outputs are labelled inside the circles. Here, only the input value is labeled on each transition. In the above figure, there are two transitions from each state based on the value of input, x.

In general, the number of states required in Moore state machine is more than or equal to the number of states required in Mealy state machine. There is an equivalent Mealy state machine for each Moore state machine. So, based on the requirement we can use one of them.

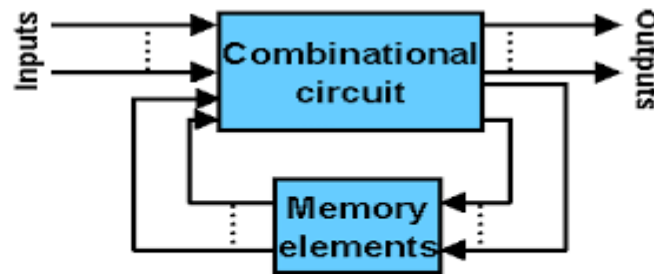
### Finite State Machine:

Finite state machine can be defined as a type of machine whose past histories can affect its future behavior in a finite number of ways. To clarify, consider for example of binary full adder. Its output depends on the present input and the carry generated from the previous input. It may have a large number of previous input histories but they can be divided into two types: (i) Input

The most general model of a sequential circuit has inputs, outputs and internal states. A sequential circuit is referred to as a finite state machine (FSM). A finite state machine is abstract model that describes the synchronous sequential machine. The fig. shows the block diagram of a finite state model.  $X_1, X_2, \dots, X_l$  are inputs.  $Z_1, Z_2, \dots, Z_m$  are outputs.  $Y_1, Y_2, \dots, Y_k$  are state variables, and  $Y_1, Y_2, \dots, Y_k$  represent the next state.







**Block Diagram of a Sequential Circuit**

### Capabilities and limitations of finite-state machine

Let a finite state machine have  $n$  states. Let a long sequence of input be given to the machine. The machine will progress starting from its beginning state to the next states according to the state transitions. However, after some time the input string may be longer than  $n$ , the number of states. As there are only  $n$  states in the machine, it must come to a state it was previously been in and from this phase if the input remains the same the machine will function in a periodically repeating fashion. From here a conclusion that for a  $n$  state machine the output will become periodic after a number of clock pulses less than equal to  $n$  can be drawn. States are memory elements. As for a finite state machine the number of states is finite, so finite number of memory elements are required to design a finite state machine.

#### Limitations:

1. Periodic sequence and limitations of finite states: with  $n$ -state machines, we can generate periodic sequences of  $n$  states are smaller than  $n$  states. For example, in a 6-state machine, we can have a maximum periodic sequence as 0,1,2,3,4,5,0,1....
2. No infinite sequence: consider an infinite sequence such that the output is 1 when and only when the number of inputs received so far is equal to  $P(P+1)/2$  for  $P=1,2,3,\dots$ , i.e., the desired input-output sequence has the following form:

Input: x  
 Output: 1 0 1 0 0 1 0 0 0 0 10 0 0 0 1 0 0 00 0 1

Such an infinite sequence cannot be produced by a finite state machine.

3. Limited memory: the finite state machine has a limited memory and due to limited memory it cannot produce certain outputs. Consider a binary multiplier circuit for multiplying two

arbitrarily large binary numbers. The memory is not sufficient to store arbitrarily large partial products resulted during multiplication.

Finite state machines are two types. They differ in the way the output is generated they are:

1. Mealy type model: in this model, the output is a function of the present state and the present input.
2. Moore type model: in this model, the output is a function of the present state only.

Mathematical representation of synchronous sequential machine:

The relation between the present state  $S(t)$ , present input  $X(t)$ , and next state  $s(t+1)$  can be given as

$$S(t+1) = f\{S(t), X(t)\}$$

The value of output  $Z(t)$  can be given as

$$Z(t) = g\{S(t), X(t)\} \quad \text{for mealy model}$$

for Moore

$$Z(t) = G\{S(t)\} \quad \text{model}$$

Because, in a mealy machine, the output depends on the present state and input, whereas in a Moore machine, the output depends only on the present state.

#### Comparison between the Moore machine and mealy machine:

Moore machine	mealy machine
1. its output is a function of present state only $Z(t) = g\{S(t)\}$	1. its output is a function of present state as well as present input $Z(t) = g\{S(t), X(t)\}$
2. input changes do not affect the output	2. input changes may affect the output of the circuit
3. it requires more number of states for implementing same function	3. it requires less number of states for implementing same function

#### Mealy model:

When the output of the sequential circuit depends on both the present state of the flip-flops and on the inputs, the sequential circuit is referred to as mealy circuit or mealy machine.

The fig. shows the logic diagram of the mealy model. Notice that the output depends up on the present state as well as the present inputs. We can easily realize that changes in the input during the clock pulse cannot affect the state of the flip-flop. They can affect the output of the circuit. If the input variations are not synchronized with a clock, the derived output will also not be synchronized with the clock and we get false output. The false outputs can be eliminated by allowing input to change only at the active transition of the clock.

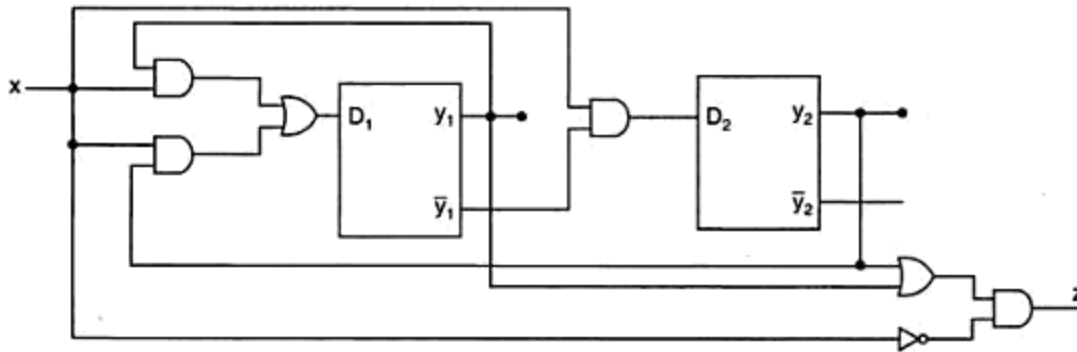


Fig: Logic diagram of a Mealy model

The behavior of a clocked sequential circuit can be described algebraically by means of state equations. A state equation specifies the next state as a function of the present state and inputs. The mealy model shown in fig. consists of two D flip-flops, an input x and an output z. since the D input of a flip-flop determines the value of the next state, the state equations for the model can be written as

$$Y_1(t+1) = y_1(t)x(t) + y_2(t)x(t)$$

$$Y_2(t+1) = y_1(t)x(t)$$

And the output equation is

$$Z(t) = \{y_1(t) + y_2(t)\} X'(t)$$

Where  $y(t+1)$  is the next state of the flip-flop one clock edge later,  $x(t)$  is the present input, and  $z(t)$  is the present output. If  $y_1(t+1)$  are represented by  $y_1(t)$  and  $y_2(t)$ , in more compact form, the equations are

$$Y_1(t+1) = y_1x + y_2x$$

$$Y_2(t+1) =$$

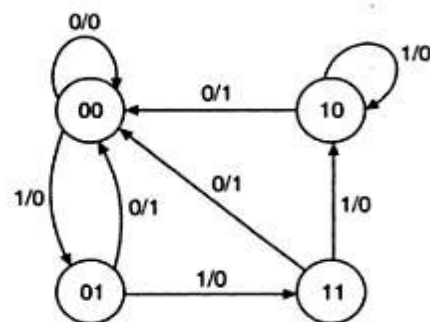
$$y_1x'$$

$$Z = (y_1 + y_2) x'$$

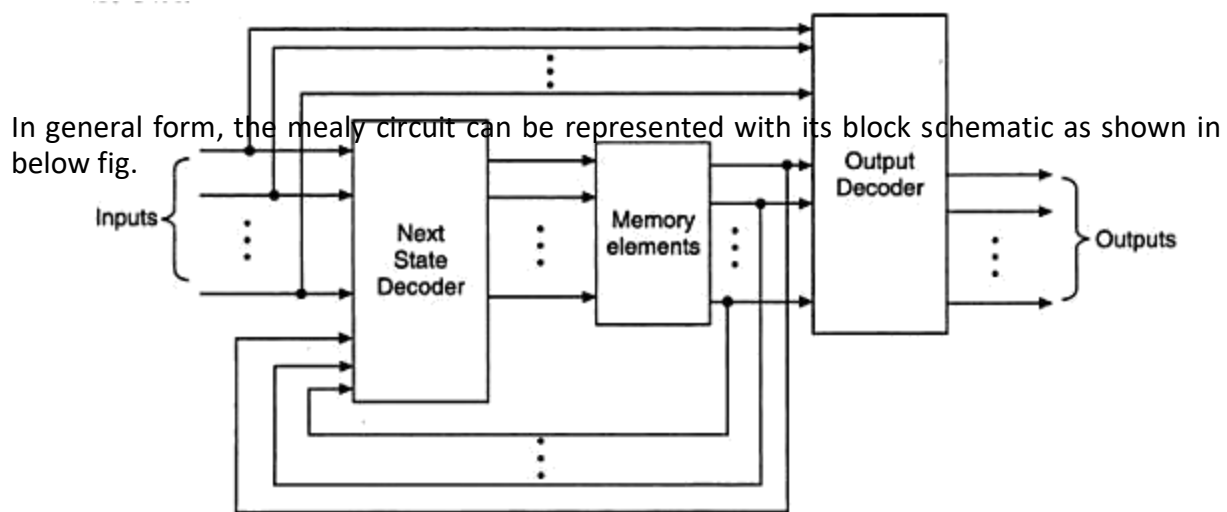
The stable table of the mealy model based on the above state equations and output equation is shown in fig. the state diagram based on the state table is shown in fig.

PS		NS				O/P	
		x = 0		x = 1		x = 0	x = 1
$y_1$	$y_2$	$Y_1$	$Y_2$	$Y_1$	$Y_2$	$z$	$z$
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0

(a) State table



(b) State diagram



### Moore model:

when the output of the sequential circuit depends up only on the present state of the flip-flop, the sequential circuit is referred as to as the Moore circuit or the Moore machine. Notice that the output depend only on the present state. It does not depend upon the input at all. The input is used only to determine the inputs of flip-flops. It is not used to determine the output. The circuit shown has two T flip-flops, one input x, and one output z. it can be described algebraically by two input equations an output equation.

$$T_1 =$$

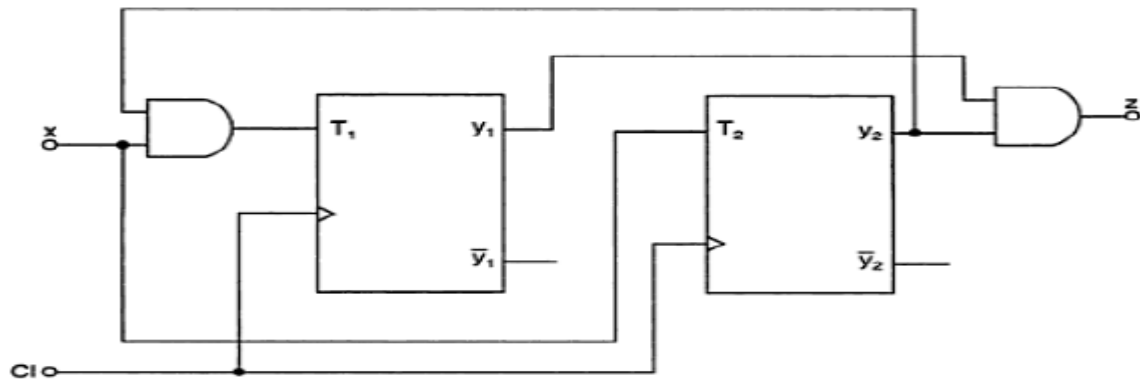
$$y_2 x$$

$$T_2 =$$

$$x$$

$$Z = y$$

$$1 y_2$$



The characteristic equation of a T-flip-flop is

$$Q(t+1) = TQ' + T'Q$$

The values for the next state can be derived from the state equations by substituting T1 and T2 in the characteristic equation yielding

$$\begin{aligned} Y_1(t+1) &= Y_1 = (y_2x) \oplus (y_2x)y_1 + (y_2x)y_1x \\ &= y_1 y_2 + y_1 x + y_1 y_2 x \\ &= y_2 (t+1) = x \oplus y_2 = xy_2 + x y_2 \end{aligned}$$

The state table of the Moore model based on the above state equations and output equation is shown in fig.

In general form , the Moore circuit can be represented with its block schematic as shown in below fig.

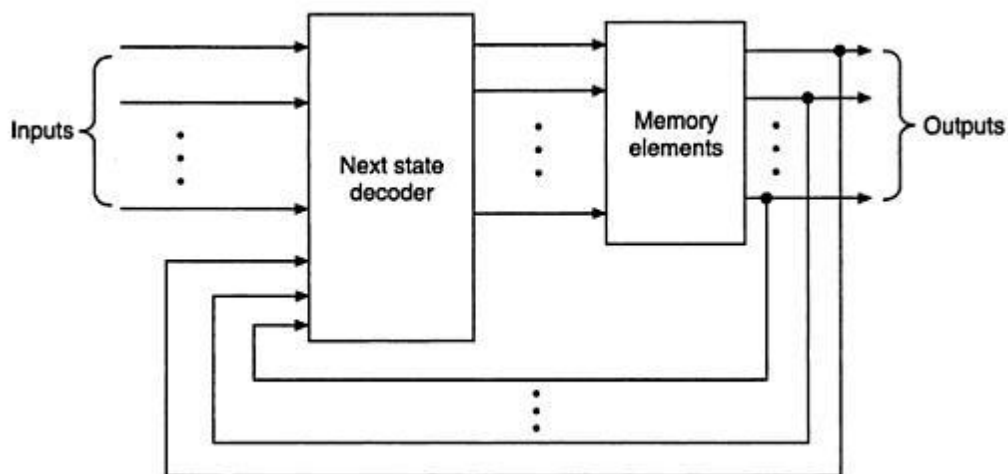


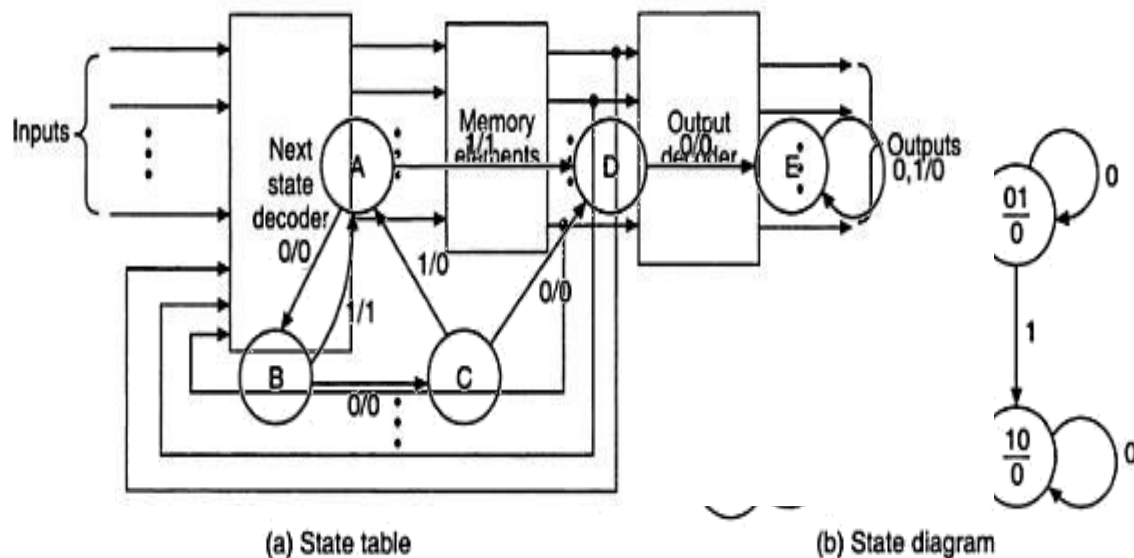
Figure: Moore circuit model:

Figure: Moore circuit model with an output decoder

Important definitions and theorems:

**A). Finite state machine-definitions:**

Consider the state diagram of a finite state machine shown in fig. it is five-state machine with one input variable and one output variable.



**Successor:** looking at the state diagram when present state is A and input is 1, the next state is D. this condition is specified as D is the successor of A. similarly we can say that A is the 1 successor of B, and C,D is the 11 successor of B and C, C is the 00 successor of A and D, D is the 000 successor of A, E, is the 10 successor of A or 0000 successor of A and so on.

**Terminal state:** looking at the state diagram, we observe that no such input sequence exists which can take the sequential machine out of state E and thus state E is said to be a terminal state.

**Strongly-connected machine:** in sequential machines many times certain subsets of states may not be reachable from other subsets of states. Even if the machine does not contain any terminal state. If for every pair of states  $s_i, s_j$ , of a sequential machine there exists an input sequence which takes the machine M from  $s_i$  to  $s_j$ , then the sequential machine is said to be strongly connected.

### B). state equivalence and machine minimization:

In realizing the logic diagram from a stat table or state diagram many times we come across redundant states. Redundant states are states whose functions can be accomplished by other states. The elimination of redundant states reduces the total number of states of the machines which in turn results in reduction of the number of flip-flops and logic gates, reducing the cost of the final circuit.

Two states are said to be equivalent. When two states are equivalent, one of them can be removed without altering the input output relationship.

**State equivalence theorem:** it states that two states  $s_1$ , and  $s_2$  are equivalent if for every possible input sequence applied. The machine goes to the same next state and generates the same output. That is

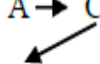
If  $S_1(t+1) = S_2(t+1)$  and  $z_1 = z_2$ , then  $s_1 = s_2$

### C). Distinguishable states and distinguishing sequences:

Two states  $s_a$ , and  $s_b$  of a sequential machine are distinguishable, if and only if there exists at least one finite input sequence which when applied to the sequential machine causes different outputs sequences depending on whether  $s_a$  or  $s_b$  is the initial state.

Consider states A and B in the state table, when input  $X=0$ , their outputs are 0 and 1 respectively and therefore, states A and B are called 1-distinguishable. Now consider states A and E. the output sequence is as follows.

$X=0$      $A \rightarrow C, 0$     and  $E \rightarrow D, 0$  ; outputs are the same



$C \rightarrow E, 0$  and  $D \rightarrow b, 1$  ; outputs are different

Here the outputs are different after 2-state transition and hence states A and E are 2-. Again consider states A and C . the output sequence is as follows:

$X=0$   $A \rightarrow C, 0$  and  $C \rightarrow E, 0$ ; outputs are the same  
 $C \rightarrow E, 0$  and  $E \rightarrow D, 0$  ; outputs are the same  
 $E \rightarrow D, 0$  and  $D \rightarrow B, 1$  ; outputs are different

Here the outputs are different after 3- transition and hence states A and B are 3-distinguishable. the concept of K- distinguishable leads directly to the definition of K-equivalence. States that are not K-distinguishable are said to be K-equivalent.

#### Truth table for Distinguishable states:

PS	NS,Z	
	X=0	X=1
A	C,0	F,0
B	D,1	F,0
C	E,0	B,0
D	B,1	E,0
E	D,0	B,0
F	D,1	B,0

#### State Reduction:

The reduction of the number of flip-flops in a sequential circuit is referred to as the *state reduction* problem. State-reduction algorithms are concerned with procedures for reducing the number of states in a state table, while keeping the external input-output requirements unchanged. Since ( $N$ ) flip-flops produce ( $2N$ ) states, a reduction in the number of states may (or may not) result in a reduction in the number of flip-flops. An  $n$  predictable effect in reducing the number of flip-flops is that sometimes the equivalent circuit (with fewer flip-flops) may require more combinational gates. We will illustrate the state reduction procedure with an example. We start with a sequential circuit whose specification is given in the state diagram shown in Fig. (1). In this example, only the input-output sequences are important; the internal



states are used merely to provide the required sequences. For this reason, the states marked inside the circles are denoted by letter symbols instead of their binary values. This is in constant to a binary counter, where the binary value sequence of the state themselves is taken as the outputs.

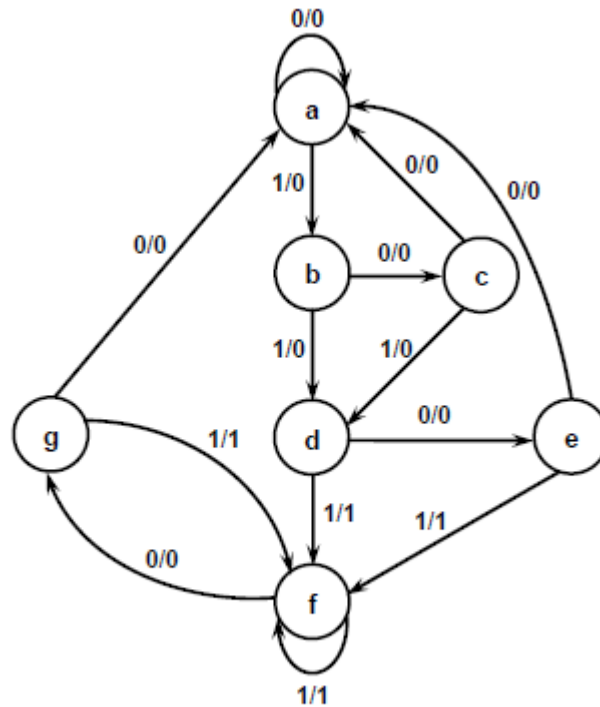


Fig. (1): State Diagram.

There are an infinite number of input sequences that may be applied to the circuit; each results in a unique output sequence. As an example, consider the input sequence  $[01010110100]$  starting from the initial state (a). Each input of 0 or 1 produces an output of 0 or 1 and causes the circuit to go to the next state. the output and state sequence for the given input sequence as follows: With the circuit in initial state (a), an input of 0 produces an output of 0 and the circuit remains in state (a). With present state (a) and input of 1, the output is 0 and the next state is (b). With present state (b) and input of 0, the output is 0 and next state is (c). Continuing this process, we find the complete sequence to be as

State	a	a	b	c	d	e	f	f	g	f	g	a
Input	0	1	0	1	0	1	1	0	1	0	0	
Output	0	0	0	0	0	1	1	0	1	0	0	

follows:

In each column, we have the present state, input value, and output value. The next state is written on top of the next column. It is important to realize that in this circuit, the states themselves are of secondary importance because we are interested only in output sequences

caused by input sequences. Now let us assume that we have found a sequential circuit whose state diagram has less than seven states and we wish to compare it with the circuit whose state diagram is given by Fig. (1). If identical input sequences are applied to the two circuits and identical outputs occur for all input sequences, then the two circuits are said to be equivalent (as far as the input-output is concerned) and one may be replaced by the other. The problem of state reduction is to find ways of reducing the number of states in a sequential circuit without altering the input-output relationships.

We now proceed to reduce the number of states for this example. First, we need the state table; it is more convenient to apply procedures for state reduction using a table rather than a diagram. The state table of the circuit is listed in Table (1) and is obtained directly from the state diagram.

Table (1)  
State Table.

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

An algorithm for the state reduction of a completely specified state table is given here without proof: **"Two states are said to be equivalent if, for each member of the set of inputs, they give exactly the same output and send the circuit either to the same state or to an equivalent state."** When two states are equivalent, one of them can be removed without altering the input-output relationships.

Now apply this algorithm to Table (1). Going through the state table, we look for two present states that go to the same next state and have the same output for both input combinations. States (g) and (e) are two such states: they both go to states (a & se) are equivalent and one of these states can be removed. The procedure of removing a state and replacing it by its equivalent is demonstrated in Table (2). The row with present state (g) is removed and state (g) is replaced by state (e) each time it occurs in the next-state columns

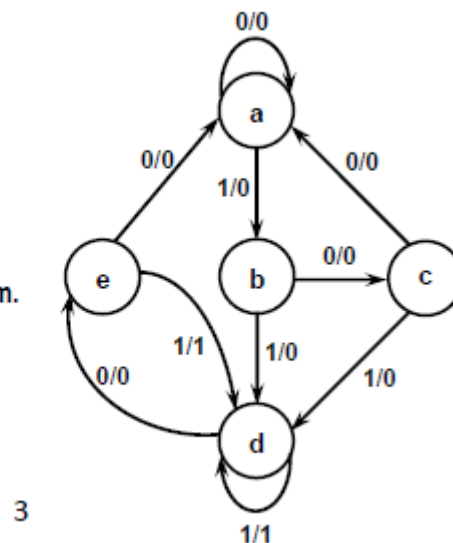
Table (2)  
Reducing the State Table.

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

Table (3)  
Reduced State Table.

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

Fig. (2):  
Reduced State Diagram.



3

Present state (f) now has next states (e and f) and outputs 0 and 1 for x=0 and x=1, respectively. The same next states and outputs appear in the row with present (d). Therefore, states (f and d) are equivalent and state (f) can be removed and replaced by (d). The final reduced table is shown in Table (3). The state diagram for the reduced table consists of only five states and is

shown in Fig. (2). This state diagram satisfies the original input-output specifications and will produce the required output sequence for any given input sequence.

The following list derived from the state diagram of Fig. (2) is for the input sequence used previously (note that the same output sequence results, although the state sequence is different):

State	a	a	b	c	d	e	d	d	e	d	e	a
Input	0	1	0	1	0	1	1	0	1	0	0	
Output	0	0	0	0	0	1	1	0	1	0	0	

In fact, this sequence is exactly the same as that obtained for Fig. (1), if we replace (g by e and f by d). Checking each pair of states for possible equivalency can be done systematically by means of a procedure that employs an implication table. The implication table consists of squares, one for every suspected pair of possible equivalent states. By judicious use of the table, it is possible to determine all pairs of equivalent states in a state table. The use of the implication table for reducing the number of states in a state table is demonstrated in the next section. The sequential circuit of this example was reduced from seven to five state. In general, reducing the number of states in a state table may result in a circuit with less equipment.

However, the fact that a state table has been reduced to fewer state doesn't guarantee a saving in the number of flip-flops or the number of gates.

#### Implication Table:

The state-reduction procedure for completely specified state tables is based on the algorithm that two states in a state table can be combined into one if 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 state. Consider for example, the state table shown in Table (4). 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). Similarly, from the last two rows of Table (4), we find that the pair of states (c, d) imply 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 as well as (c and d). As a consequence, the four rows of Table (4) can be reduced to two rows by combining (a and b) into one state and (c and d) into a second state. The checking of each pair of states for possible equivalence in a table with a large number of states can be done systematically by means of an implication table. The implication table is a chart that consists of squares, one for every possible pair of states, that provide spaces for listing any possible implied states. By judicious use of the table, it is possible to determine all pairs of equivalent states. The state table of Table (5) will be used to illustrate this procedure. The implication table is shown in Fig. (3). 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 states with a square placed in the intersection of a row and a column where the two states can be tested for equivalence.

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 (✓). Some of the squares have entries of implied states that must be further investigated to determine whether they are equivalent or not. The step-by-step procedure of filling in the squares is as follows. First, we place a cross in any square corresponding to a pair of states whose outputs are not equal for every input. In this case, state (c) has a different output than any other state, so a cross is placed in the two squares of row (c) and the four squares of column (c). There are nine other squares in this category in the implication table.

**Table (4)**  
**State Table to Demonstrate Equivalent States.**

Present State	Next State		Output	
	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

Next, we enter in the remaining squares the pairs of states that are implied by the pair of states representing the squares. We do that starting from the top square in the left column and going down and then proceeding with the next column to the right. From the state table, we see that pair (a,b) imply (d,e), so (d,e) is recorded in the square defined by column (a and row b). We proceed in this manner until the entire table is completed. Note that states (d,e) are equivalent because they go to the same next state and have the same output. Therefore, a check mark is recorded in the square defined by column (d and row e), indicating that the two states are equivalent and independent of any implied pair.

The next step is to make successive passes through the table to determine whether any additional squares should be marked with a cross. A square in the table is crossed out if it contains at least one implied pair that is not equivalent. For example, the square defined by (a) and (f) is marked with a cross next to (c,d) because the pair (c,d) defines a square that contains a cross. This procedure is repeated until no additional squares can be crossed out.

Finally, all the squares that have no crosses are recorded with check marks. These squares define pairs of equivalent states. In this example, the equivalent states are:

(a,b) (d,e) (d,g) (e,g)

Table (5)  
State Table to be Reduced.

Present State	Next State		Output	
	x=0	x=1	x=0	x=1
a	d	a	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

b	(d,e) ✓				
c	x	x			
d	x	x	x		
e	x	x	x	✓	
f	(c,d) x (a,b)	(c,e) x (a,b)	x	x	x
g	x	x	x	(d,e) ✓	(d,e) ✓
	a	b	c	d	e

Fig. (3): Implication table.

We now combine pairs of states into larger groups of equivalent states. The last three pairs can be combined into a set of three equivalent states (d,e,g) because each one of the states in the group is equivalent to the other two. The final partition of the states consists of the equivalent states found from the implication table, together with all the remaining states in the state table that are not equivalent to any other state. (a,b) (c) (d,e,g) (f) This means that Table (5) can be reduced from seven states to four states, one for each member of the above partition. The reduced table is obtained by replacing state (b by a and states e and g by d).

Table (6)  
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

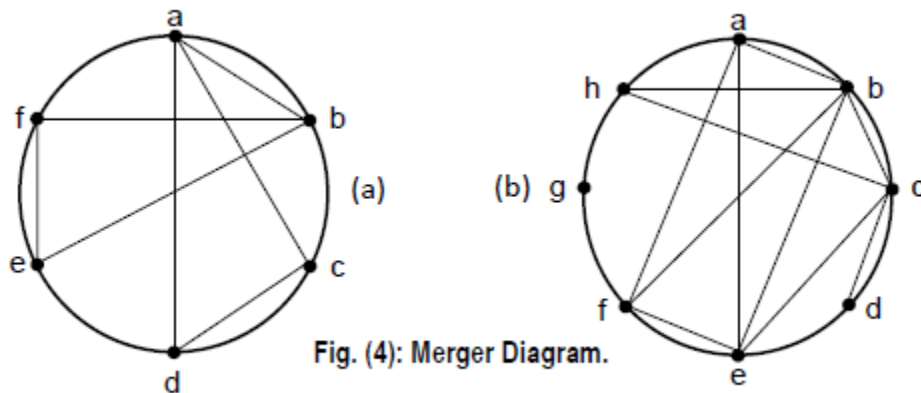
Merger Diagram:

Having found all the compatible pairs, the next step is to find larger sets of states that are compatible. 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, as shown in Fig. (4). 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 to any other state. A line represents a compatible pair. A triangle constitutes a compatible with three states. An n-state compatible is represented in the merger diagram by an n-state polygon with all its diagonals connected.

The merger diagram of Fig. (4-a) is obtained from the list of compatible pairs derived from the implication table. There are seven straight lines connecting the dots, one for each compatible pair. The lines from a geometrical pattern consisting of two triangles connecting (a,c, d) and (b, e, f) and a line (a, b). The maximal compatibles are:

(a,b) (a,c,d) (b,e,f)

Fig. (4-b) shows the merger diagram of an 8-state. The geometrical patterns are a rectangle with its two diagonals connected to form the 4-state compatible (a, b, e, f), a triangle (b, c, h), a line (c, d), and a single state (g) that is not compatible to any other state. The maximal compatibles are: (a,b,e,f) (b,c,h) (c,d) (g)



### Merger Chart Methods:

#### Merger graphs:

The merger graph is a state reducing tool used to reduce states in the incompletely specified machine. The merger graph is defined as follows.

1. Each state in the state table is represented by a vertex in the merger graph. So it contains the same number of vertices as the state table contains states.
2. Each compatible state pair is indicated by an unbroken line drawn between the two state

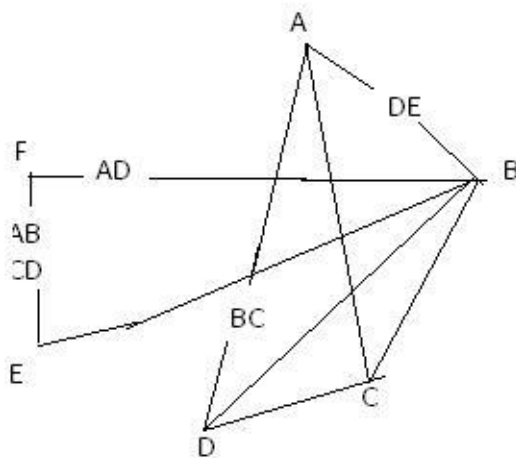
vertices

- Every potentially compatible state pair with non-conflicting outputs but with different next states is connected by a broken line. The implied states are written in the line break between the two potentially compatible states.
- If two states are incompatible no connecting line is drawn.

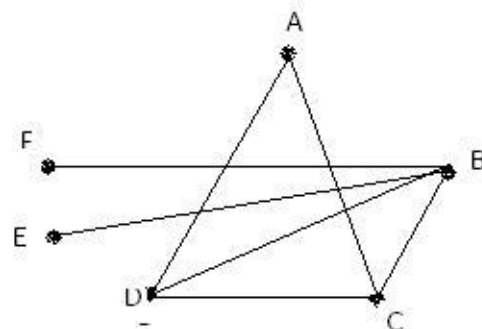
Consider a state table of an incompletely specified machine shown in fig. the corresponding merger graph shown in fig.

**State table:**

PS	NS,Z			
	I1	I2	I3	I4
A	...	E,1	B,1	...
B	...	D,1	...	F,1
C	F,1	...	...	...
D	...	...	C,1	...
E	C,0	...	A,0	F,1
F	D,0	A,1	B,0	...



a) Merger graph



b) simplified merger graph

States A and B have non-conflicting outputs, but the successor under input I2 are compatible only if implied states D and E are compatible. So, draw a broken line from A to B with DE written in between. states A and C are compatible because the next states and output entries of states A and C are not conflicting. Therefore, a line is drawn between nodes A and C. states A and D have non-conflicting outputs but the successor under input I3 are B and C. hence join A and D by a broken line with BC entered in between.



Two states are said to be incompatible if no line is drawn between them. If implied states are incompatible, they are crossed and the corresponding line is ignored. Like, implied states D and E are incompatible, so states A and B are also incompatible. Next, it is necessary to check whether the incompatibility of A and B does not invalidate any other broken line. Observe that states E and F also become incompatible because the implied pair AB is incompatible. The broken lines which remain in the graph after all the implied pairs have been verified to be compatible are regarded as complete lines.

After checking all possibilities of incompatibility, the merger graph gives the following seven compatible pairs.

(A, C) (A, D) (B, C) (B, D) (C, D) (B, E) (B, F)

These compatible pairs are further checked for further compatibility. For example, pairs (B,C)(B,D)(C,D) are compatible. So (B, C, D) is also compatible. Also pairs (A,c)(A,D)(C,D) are compatible. So (A,C,D) is also compatible. . In this way the entire set of compatibles of sequential machine can be generated from its compatible pairs.

To find the minimal set of compatibles for state reduction, it is useful to find what are called the maximal compatibles. A set of compatibles state pairs is said to be maximal, if it is not completely covered by any other set of compatible state pairs. The maximum compatible can be found by looking at the merger graph for polygons which are not contained within any higher order complete polygons. For example only triangles (A, C,D) and (B,C,D) are of higher order. The set of maximal compatibles for this sequential machine given as

(A, C, D) (B, C, D) (B, E) (B, F)

**Example:**

Draw the merger graph and obtain the set of maximal compatibles for the incompletely specified sequential machine whose state table is given in Table 7.24.

**Table 7.24** Example 7.9: State table

PS	NS, Z	
	I <sub>1</sub>	I <sub>2</sub>
A	E, 0	B, 0
B	F, 0	A, 0
C	E, –	C, 0
D	F, 1	D, 0
E	C, 1	C, 0
F	D, –	B, 0

mark × in the corresponding cell. For example, states B and C are incompatible because their outputs are conflicting and hence the cell corresponding to them contains a cross mark ×. Similarly states B, E; D, E; E, F are incompatible. Hence put a × mark in the corresponding cells. On the other hand, states A and B are compatible and hence the cell corresponding to them contains the check mark ✓. Similarly, cells corresponding to states A, D; A, E; A, G; B, G; C, F; D, F; D, G are also compatible. So a check mark is put in those cells also. The implied pairs or pairs corresponding to the state pair are written within the cell as shown in Table 7.26. For example, states A and C are compatible only when implied states E and F are compatible. Therefore, EF is written in the cell corresponding to states A and C. States C and E are compatible only when implied states A and B, and D and F are compatible. So AB and DF are written in the cell corresponding to states C and E. In a similar way, the entire merger table is written. Now it is necessary to check whether the implied pairs are compatible or not by checking the corresponding cells. The implied states are

PS	NS, Z			
	00	01	11	10
A	E, 0	-	-	-
B	-	F, 1	E, 1	A, 1
C	F, 0	-	A, 0	F, 1
D	-	-	A, 1	-
E	-	C, 0	B, 0	D, 1
F	C, 0	C, 1	-	-
G	E, 0	-	-	A, 1

Figure: state table

B	✓					
C	<del>DE</del>	x				
D	✓	AE	x			
E	✓	x	AB DF	x		
F	CE	CF	✓	✓	x	
G	✓	✓	<del>EF</del> <del>AG</del>	✓	AD	CE

**State Minimization:****Completely Specified Machines**

- ❓ Two states,  $s_i$  and  $s_j$  of machine  $M$  are *distinguishable* if and only if there exists a finite input sequence which when applied to  $M$  causes different output sequences depending on whether  $M$  started in  $s_i$  or  $s_j$ .
- ❓ Such a sequence is called a *distinguishing sequence* for  $(s_i, s_j)$ .
- ❓ If there exists a distinguishing sequence of length  $k$  for  $(s_i, s_j)$ , they are said to be *k-distinguishable*.

EXAMPLE:

PS	NS, z	
	x=0	x=1
A	E, 0	D, 1
B	F, 0	D, 0
C	E, 0	B, 1
D	F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0

- states A and B are 1-distinguishable, since a 1 input applied to A yields an output 1, versus an output 0 from B.
- states A and E are 3-distinguishable, since input sequence 111 applied to A yields output 100, versus an output 101 from E.
- States  $si$  and  $sj$  ( $si \sim sj$ ) are said to be equivalent iff no distinguishing sequence exists for  $(si, sj)$ .
- If  $si \sim sj$  and  $sj \sim sk$ , then  $si \sim sk$ . So state equivalence is an equivalence relation (i.e. it is a reflexive, symmetric and transitive relation).
- An equivalence relation partitions the elements of a set into equivalence classes.
- Property: If  $si \sim sj$ , their corresponding X-successors, for all inputs X, are also equivalent.
- Procedure: Group states of  $M$  so that two states are in the same group iff they are equivalent (forms a partition of the states).

### Completely Specified Machines

PS	NS, z	
	x=0	x=1
A	E, 0	D, 1
B	F, 0	D, 0
C	E, 0	B, 1
D	F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0

$P_i$  : partition using distinguishing sequences of length  $i$ .

Partition:

Distinguishing Sequence:

$P_0 = (A \ B \ C \ D \ E \ F)$

$P_1 = (A \ C \ E)(B \ D \ F)$

$x = 1$

$P_2 = (A \ C \ E)(B \ D)(F)$

$x = 1; x = 1$

$P_3 = (A \ C)(E)(B \ D)(F)$

$x = 1; x = 1; x = 1$

$P_4 = (A \ C)(E)(B \ D)(F)$

Algorithm terminates when  $P_k = P_{k+1}$

Outline of state minimization procedure:

- All states equivalent to each other form an equivalence class. These may be combined into one state in the reduced (quotient) machine.
- Start an initial partition of a single block. Iteratively refine this partition by separating the 1-distinguishable states, 2-distinguishable states and so on.
- To obtain  $P_{k+1}$ , for each block  $B_i$  of  $P_k$ , create one block of states that not 1-distinguishable within  $B_i$ , and create different blocks states that are 1-distinguishable

within  $B_i$ .

**Theorem:** The equivalence partition is unique.

**Theorem:** If two states,  $s_i$  and  $s_j$ , of machine  $M$  are distinguishable, then they are  $(n-1)$ -distinguishable, where  $n$  is the number of states in  $M$ .

**Definition:** Two machines,  $M_1$  and  $M_2$ , are *equivalent* ( $M_1 \sim M_2$ ) if, for every state in  $M_1$  there is a corresponding equivalent state in  $M_2$  and vice versa.

**Theorem.** For every machine  $M$  there is a minimum machine  $M_{red} \sim M$ .  $M_{red}$  is unique up to isomorphism.

## Completely Specified Machines

- Reduced machine obtained from previous example:

$$P_4 = (A\ C)(E)(B\ D)(F) \\ = \alpha\ \beta\ \gamma\ \delta$$

PS	NS, z	
	x=0	x=1
$\alpha$	$\beta, 0$	$\gamma, 1$
$\beta$	$\alpha, 0$	$\delta, 1$
$\gamma$	$\delta, 0$	$\gamma, 0$
$\delta$	$\gamma, 0$	$\alpha, 0$

PS	NS, z	
	x=0	x=1
A	E, 0	D, 1
B	F, 0	D, 0
C	E, 0	B, 1
D	F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0

### State Minimization: Incompletely Specified Machines

Statement of the problem: given an incompletely specified machine  $M$ , find a machine  $M'$  such that:

- on any input sequence,  $M'$  produces the same outputs as  $M$ , whenever  $M$  is specified.
- there does not exist a machine  $M''$  with fewer states than  $M'$  which has the same property

#### Machine $M$ :

PS	NS, z	
	x=0	x=1
s1	s3, 0	s2, 0
s2	s2, -	s3, 0
s3	s3, 1	s2, 0

Attempt to reduce this case to usual state minimization of completely specified machines.

- ❓ Brute Force Method: Force the don't cares to all their possible values and choose the smallest of the completely specified machines so obtained.
- ❓ In this example, it means to state minimize two completely specified machines obtained from  $M$ , by setting the don't care to either 0 and 1.

Suppose that the - is set to be a 0.

PS	NS, z	
	x=0	x=1
s1	s3, 0	s2, 0
s2	s2, 0	s3, 0
s3	s3, 1	s2, 0

- ❓ States s1 and s2 are equivalent if s3 and s2 are equivalent, but s3 and s2 assert different outputs under input 0, so s1 and s2 are not equivalent.
- ❓ States s1 and s3 are not equivalent either.
- ❓ So this completely specified machine cannot be reduced further (3 states is the minimum).

Suppose that the - is set to be a 1.

PS	NS, z	
	x=0	x=1
s1	s3, 0	s2, 0
s2	s2, 1	s3, 0
s3	s3, 1	s2, 0

$j_1 \neq j_2, \exists (Q_i, a) \exists Q_j1 \text{ ???}, \text{ and } \text{??}$

- ❓ States s1 is incompatible with both s2 and s3.
- ❓ States s3 and s2 are equivalent.
- ❓ So number of states is reduced from 3 to 2.

Machine  $M''_{red}$  :

PS	NS, z	
	x=0	x=1
A	A, 1	A, 0
B	B, 0	A, 0

Can this always be done?

Machine  $M$ :

PS	NS, z	
	x=0	x=1
s1	s3, 0	s2, 0
s2	s2, -	s1, 0
s3	s1, 1	s2, 0

Machine  $M_2$ :

PS	NS, z	
	x=0	x=1
s1	s3, 0	s2, 0
s2	s2, 0	s1, 0
s3	s1, 1	s2, 0

Machine  $M_3$ :

PS	NS, z	
	x=0	x=1
s1	s3, 0	s2, 0
s2	s2, 1	s1, 0
s3	s1, 1	s2, 0

Machine  $M_2$  and  $M_3$  are formed by filling in the unspecified entry in  $M$  with 0 and 1, respectively.

Both machines  $M_2$  and  $M_3$  cannot be reduced. Conclusion?:  $M$  cannot be minimized further! But is it a correct conclusion?

**Note:** that we want to 'merge' two states when, for any input sequence, they generate the same output sequence, but only where both outputs are specified.



**Definition:** A set of states is compatible if they agree on the outputs where they are all specified.

**Machine  $M''$ :**

PS	NS, z	
	x=0	x=1
s1	s3, 0	s2, 0
s2	s2, -	s1, 0
s3	s1, 1	s2, 0

In this case we have two compatible sets: A = (s1, s2) and B = (s3, s2). A reduced machine Mred can be built as follows.

Machine Mred

PS	NS, z	
	x=0	x=1
A	A, 1	A, 0
B	B, 0	A, 0

PS	NS, z			
	I1	I2	I3	I4
s1	s3, 0	s1, -	-	-
s2	s6, -	s2, 0	s1, -	-
s3	-, 1	-, -	s4, 0	-
s4	s1, 0	-, -	-	s5, 1
s5	-, -	s5, -	s2, 1	s1, 1
s6	-, -	s2, 1	s6, -	s4, 1

A set of compatibles that cover all states is: (s3s6), (s4s6), (s1s6), (s4s5), (s2s5).

But (s3s6) requires (s4s6),

(s4s6) requires (s4s5), (s4s5) requires (s1s5), (s1s6)  
 requires (s1s2), (s1s2) requires (s3s6), (s2s5)  
 requires (s1s2).

So, this selection of compatibles requires too many other compatibles...

PS	NS, z			
	I1	I2	I3	I4
s1	s3, 0	s1, -	-	-
s2	s6, -	s2, 0	s1, -	-
s3	-, 1	-, -	s4, 0	-
s4	s1, 0	-, -	-	s5, 1
s5	-, -	s5, -	s2, 1	s1, 1
s6	-, -	s2, 1	s6, -	s4, 1

☐ Another set of compatibles that covers all states is (s1s2s5), (s3s6), (s4s5).

☐ But (s1s2s5) requires (s3s6) (s3s6) requires (s4s6)

☐ (s4s6) requires (s4s5) (s4s5) requires (s1s5).

☐ So must select also (s4s6) and (s1s5).

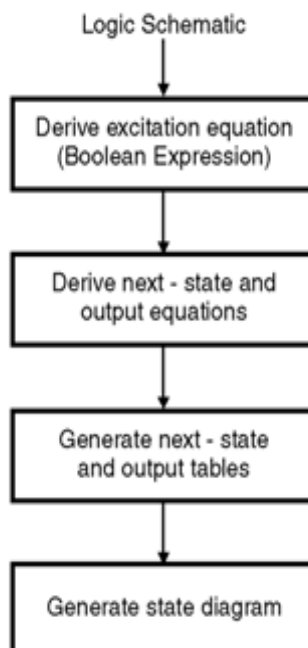
☐ Selection of minimum set is a binate covering problem

## UNIT -II

### Fundamental mode model

#### Analysis-of-Sequential-Circuits

Analysis of Sequential Circuits : The behaviour of a sequential circuit is determined from the inputs, the outputs and the states of its flip-flops. Both the output and the next state are a function of the inputs and the present state. The analysis task is much simpler than the synthesis task. To analyze a circuit, we simply reverse the steps of synthesis process. Figure below shows the analysis steps.



#### Analysis procedure of a sequential circuit:

1. We start with the logic schematic from which we can derive excitation equations for each flip-flop input.
2. Then, to obtain next-state equations, we insert the excitation equations into the characteristic equations.
3. The output equations can be derived from the schematic, and once we have our output and next-state equations, we can generate the next-state and output tables as well as state diagrams.



- When we reach this stage, we use either the table or the state diagram to develop a timing diagram which can be verified through simulation.

### Stability:

For a given set of inputs (i.e., values), the system is stable if the circuit eventually reaches steady state and the excitation variables and secondary variables are equal and unchanging (little  $y$  = capital  $y$ ), otherwise the circuit is unstable.

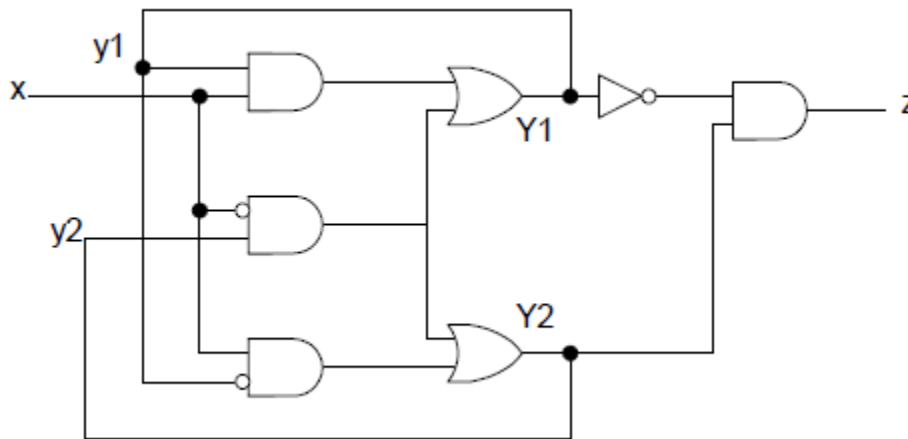
### Fundamental Mode:

A circuit is operating in fundamental mode if we assume/force the following restrictions on how the inputs can change: 1. only one input is allowed to change at a time 2. the input changes only after the circuit is stable.

Asynchronous circuits are identified by:

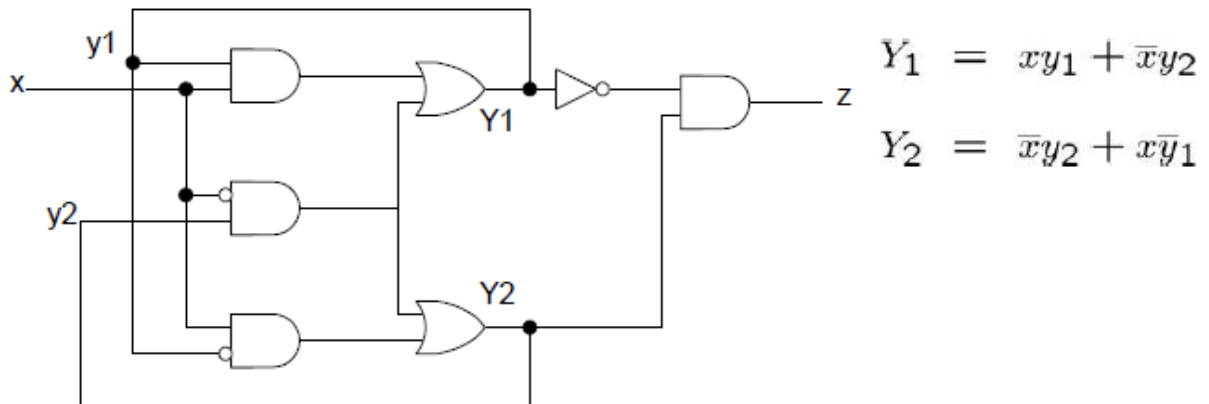
- ☐ The presence of combinatorial feedback paths, and/or
- ☐ The presence of un-clocked storage elements (i.e., latches).
- ☐ Analysis involves obtaining a table or diagram that describes the sequence of internal states and outputs as a function of changes in the circuit inputs.
- ☐ The tables we will try to obtain are **transition tables** and **Flow tables**

Consider the following circuit that has combinatorial feedback paths (and is therefore identified as asynchronous). No apparent latches in the circuit



Circuit has one input ( $x$ ), one output ( $z$ ), two secondary variables ( $y_1$ ,  $y_2$ ) and two excitation variables ( $Y_1$ ,  $Y_2$ ).

Write logic equations for the excitation variables in terms of the circuit inputs and secondary variables:



Write logic equations for circuit outputs in terms of the circuit inputs and secondary variables:

$$z = \bar{y}_1y_2$$

#### Transition Table:

Using these equations, we can write a **transition table** that shows excitation variables and outputs as a function of inputs and secondary variables:

$$Y_1 = xy_1 + \bar{x}y_2$$

$$Y_2 = \bar{x}y_2 + xy_1$$

$$z = \bar{y}_1y_2$$

curr state y2y1	next state		output	
	x=0 Y2Y1	x=1 Y2Y1	x=0 z	x=1 z
00	00	10	0	0
01	00	01	0	0
10	11	10	1	1
11	11	01	0	0

**Note that stable states (secondary variables equal to excitation variables) are circled.**

We can also create a **flow table**, which is just the transition table with binary numbers replaced with symbols (e.g., let **a = 00**, **b = 01**, **c = 10** and **d = 11**):

curr state y2y1	next state		output	
	x=0 Y2Y1	x=1 Y2Y1	x=0 z	x=1 z
00	00	10	0	0
01	00	01	0	0
10	11	10	1	1
11	11	01	0	0

curr state y2y1	next state		output	
	x=0 Y2Y1	x=1 Y2Y1	x=0 z	x=1 z
a	a	c	0	0
b	a	b	0	0
c	d	c	1	1
d	d	b	0	0

**Analysis Example –Flow Table Alternative**

Another way to draw a flow table:

	x=0	x=1
a	a, 0	c, 0
b	a, 0	b, 0
c	d, 1	c, 1
d	d, 0	b, 0

Left-most column shows current state (secondary variables), and the inputs are listed across the top. Entries in the matrix show the next state (excitation variables) and output values.

**Primitive Flow Tables**

Flow table with only one stable state per row is called a primitive flow table. E.g., a primitive flow table:

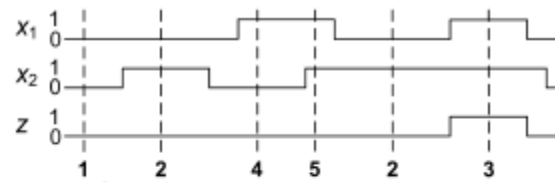
	x	
	0	1
a	a	b
b	c	b
c	c	d
d	a	d

E.g., a flow table that is not a primitive flow table:

	x1x2			
	00	01	11	10
a	a	a	a	b
b	a	a	b	b

**Flow table: analogous to the state table**

Example: Consider a sequential circuit with two inputs  $x_1$  and  $x_2$  and one output  $z$ . The initial input state is  $x_1 = x_2 = 0$ . The output value is to be 1 if and only if the input state is  $x_1 = x_2 = 1$  and the preceding input state is  $x_1 = 0, x_2 = 1$



Input-output sequences

State, output			
$x_1x_2$	00	01	11
1,0	→ 2		
		↓	
		2,0	→ 3
			↓
			3,1

Partial flow table

State, output				
$x_1x_2$	00	01	11	10
<b>1,0</b>	2	—		4
1	<b>2,0</b>	3	—	
—	2	<b>3,1</b>		4
1	—	5	<b>4,0</b>	
—	2	<b>5,0</b>		4

Primitive flow table

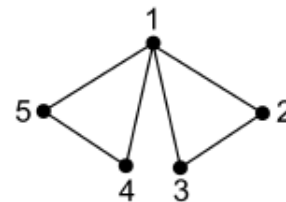
**Reduction Of Flow Tables:**

Reduction of primitive flow table has two functions:

- Elimination of redundant stable states
- Merging those stable states which are distinguishable by input states

Example: Rewrite primitive flow table like a state table

PS	State, output				
	$x_1x_2$	00	01	11	10
1	1,0	2	—	4	
2	1	2,0	3	—	
3	—	2	3,1	4	
4	1	—	5	4,0	
5	—	2	5,0	4	



Merger graph

Maximal compatibles: {(123), (145)}

Reduced flow tables

State, output			
$x_1x_2$			
00	01	11	10
1,0	2,0	3,1	4,0
1,0	2,0	5,0	4,0

State, output			
$x_1x_2$			
00	01	11	10
<b>1,0</b>	2,0	5,0	4,0
1,0	<b>2,0</b>	<b>3,1</b>	4,0

(a) Closed covering {(123),(45)}. (b) Closed covering {(145),(23)}.

## Specifying the Output Symbols

Assignment of output values to the unstable states in the reduced flow table • When the circuit is to go from one stable state to another stable state associated with the same output value: assign the same output value to the unstable state en route to avoid a momentary opposite value • When the state changes from one stable state with a given output value to another stable state with a different output value: the transition may be associated with either of these output values – When the relative timing of the output value change is of no importance: choose the output value so as to minimize logic.

State, output				
$x_1x_2$				
00	01	11	10	
1,0	2	3,0	4	
1	2,1	3	4,0	
5,1	6	7,1	8	
5	6,0	7	8,0	

(a) Reduced flow table.

State, output				
$x_1x_2$				
00	01	11	10	
1,0	2,1	3,0	4,0	
1,0	2,1	3,0	4,0	
5,1	6,0	7,1	8,0	
5,1	6,0	7,1	8,0	

(b) Reduced flow table with output values specified.

## Excitation and Output Tables

Example: Reduced flow table

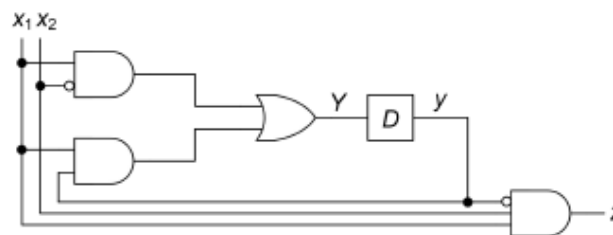
State, output				
$x_1x_2$				
00	01	11	10	
1,0	2,0	3,1	4,0	
1,0	2,0	5,0	4,0	

Excitation and output table

$y$	$Y, z$				
	$x_1x_2$	00	01	11	10
0		0,0	0,0	0,1	1,0
1		0,0	0,0	1,0	1,0

$$Y = x_1x_2' + x_1y$$

$$z = x_1x_2y'$$



Synthesis procedure for SIC fundamental-mode asynchronous circuits:

1. Construct a primitive flow table from the verbal description: specify only those output values that are associated with stable states
2. Obtain a minimum-row reduced flow table: use either the merger graph or merger table for this purpose
3. Assign secondary variables to the rows of the reduced flow table and construct excitation and output tables: specify output values associated with unstable states according to design requirements

## 4. Derive excitation and output functions, and the corresponding hazard-free Circuit

Example: Design an asynchronous sequential circuit with two inputs,  $x_1$  and  $x_2$ , and two outputs,  $G$  and  $R$ , as follows.

- Initially, both input values and both output values are 0
- Whenever  $G = 0$  and either  $x_1$  or  $x_2$  becomes 1,  $G$  becomes 1
- When the second input becomes 1,  $R$  becomes 1
- The first input value that changes from 1 to 0 turns  $G$  equal to 0
- $R$  becomes 0 when  $G$  is 0 and either input value changes from 1 to 0

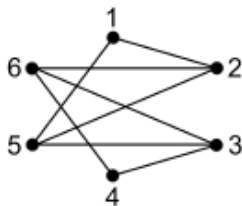
State, GR				
$x_1x_2$	00	01	11	10
1,00				
		2,10		
		3,01		
			4,11	
				5,10
				6,01

(a) Table containing only stable states.

State, GR				
$x_1x_2$	00	01	11	10
1,00	2	—	5	
1	2,10	4	—	
1	3,01	4	—	
—	3	4,11	6	
1	—	4	5,10	
1	—	4	6,01	

(b) Completed primitive flow table.

## Merger graph



## Reduced flow table

State, GR				
$x_1x_2$	00	01	11	10
1,00	2,10	4,11	5,10	
1,01	3,01	4,11	6,01	

## Excitation and output table

	$Y, GR$			
$y$	$x_1x_2$			
	00	01	11	10
0	0,00	0,10	1,11	0,10
1	0,01	1,01	1,11	1,01

$$Y = (x_1 + x_2)y + x_1x_2$$

$$G = (x_1 + x_2)y' + x_1x_2$$

$$R = y + x_1x_2$$

## Analysis Summary

Procedure to determine transition table and/or flow table from a circuit with combinatorial feedback paths:

- Determine feedback paths.
- Label  $Y$  (excitation variables) at output and  $y$  (secondary variables at input).
- Derive logic expressions for  $Y$  (excitation variables) in terms of circuit inputs and secondary

variables. Do the same for circuit outputs.

❑ Create a transition table and flow table.

❑ Circle stable states where Y (excitation variables) are equal to y (secondary variables).

### Latch Analysis

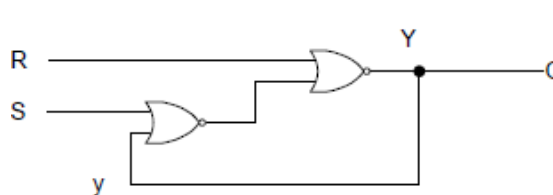
We can use the previous analysis technique to see how latches work...

❑ We will consider SR (built with NOR gates) and S'R' (built with NAND gates)

Latches.

### SR Latches

❑ We can analyze an SR latch using the previous technique:



S	R	Q	$\bar{Q}$
1	0	1	0
0	0	1	0
0	1	0	1
0	0	0	1
1	1	0	0

(after  $S = 1, R = 0$ )

(after  $S = 0, R = 1$ )

Y=Q

	SR			
y	00	01	11	10
0	0	0	0	1
1	1	0	0	1

❑ Equations derived for secondary variable (same equation for output):

$$Y' = R + (S + y)'$$

$$Y = \overline{R + (S + y)} = S\bar{R} + \bar{R}y \quad Y' = (R' \cdot (S + y))'$$

$$\Rightarrow Y = R'(S + y) = R'S + R'y$$

❑ Since we want to avoid the SR=11 situation, we can write:

$$Y = S + \bar{R}y \quad \text{if } SR = 0$$

## SR Latches

	SR				
	00	01	11	10	
$y_0$	0	0	0	1	$Y=Q$
1	1	0	0	1	

□ Can derive the transition table and the flow table:

$$Y = \overline{R + (S + y)} = S\overline{R} + \overline{R}y$$

$$Y = S + \overline{R}y \text{ if } SR = 0$$

Don't care

curr state	next state				output
	SR=00 01 11 10				
	Y	Y	Y	Y	
y					
0	0	0	0	1	0
1	1	0	0	1	1

curr state	next state				output
	SR=00 01 11 10				
	Y	Y	Y	Y	
y					
a	a	a	a	b	0
b	b	a	a	b	1

**Note:** We can see the undesirable case when  $SR=11$  and inputs change.

⌘ Depending on the various delays and assuming  $SR=11 \rightarrow SR=00$ ...

⌘ If  $SR=11 \rightarrow SR=10 \rightarrow SR=00$ , we get stable state with output of 1.

⌘ If  $SR=11 \rightarrow SR=01 \rightarrow SR=00$ , we get stable state with output of 0.

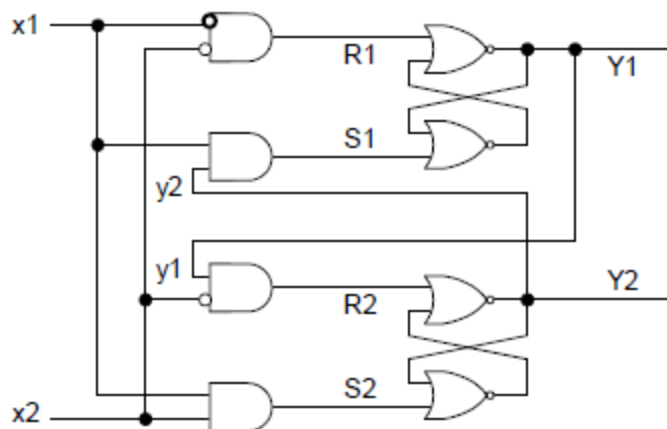
⌘ So the stable state is unpredictable.

curr state	next state				output
	SR=00 01 11 10				
	Y	Y	Y	Y	
y					
0	0	0	0	1	0
1	1	0	0	1	1

### Analysis With Latches

We might have asynchronous circuits with latches in them: We identify two inputs ( $x_1, x_2$ ), two excitation variables ( $Y_1, Y_2$ ), two secondary variables ( $y_1, y_2$ ) and two latches.





Since we see latches, we obtain logic equations for the latch inputs:

$$\begin{aligned} S_1 &= x_1 y_2 \\ R_1 &= \bar{x}_1 \bar{x}_2 \end{aligned}$$

$$\begin{aligned} S_2 &= x_1 x_2 \\ R_2 &= \bar{x}_2 y_1 \end{aligned}$$

Since we are working with latches, we should confirm that the latches do not ever enter the undesirable state ( $SR=11$  for NOR,  $SR=00$  for NAND).

In our circuit, we have NOR latches, so we find:

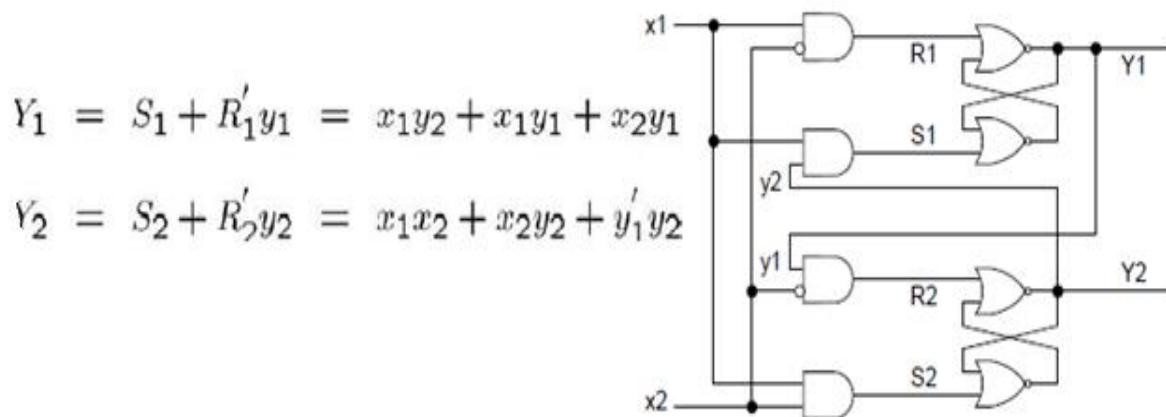
$$S_1 R_1 = x_1 y_2 \bar{x}_1 \bar{x}_2 = 0$$

$$S_2 R_2 = x_1 x_2 \bar{x}_2 y_1 = 0$$

#### Derive the transition table.

❑ We need to find the excitation equations in terms of secondary variables and the circuit inputs.

❑ To do this, we need to use the latch equations:



□ Finally, use equations to derive transition table (could also find the flow table):

$$Y_1 = S_1 + R_1' y_1 = x_1 y_2 + x_1 y_1 + x_2 y_1$$

$$Y_2 = S_2 + R_2' y_2 = x_1 x_2 + x_2 y_2 + y_1 y_2$$

		x1x2			
	y1y2	00	01	11	10
curr state	00	00	00	01	00
	01	01	01	11	11
	11	00	11	11	10
	10	00	10	11	10

Y1Y2

### Analysis Summary With Latches

Label each latch output with **Yj** and its feedback path with **yj**.

☐ Derive logic equations for latch inputs **Sj** and **Rj**.

☐ Check of  $SR=0$  for NOR Latches and  $S'R'=0$  for NAND Latches. If not satisfied, the circuit may not work correctly.

☐ Create logic equations for latch outputs **Yj** using the known behavior of a latch ( $Y=S+R'y$  for NOR Latches and  $Y=S'+Ry$  for NAND Latches).

☐ Construct a transition table using the logic equations for the latch outputs and circuit stable states.

☐ Obtain a flow table, if desired.

### Asynchronous Circuit Design

Given verbal problem description:

☐ Obtain a primitive flow table (one stable state per row) from problem description.

☐ Reduce the flow table to get a smaller flow table with less states.

☐ Perform state assignment (need to avoid race conditions) to obtain a transition

table.

□ Obtain next state and output equations (need to avoid hazards and glitches).

□ Draw circuit (with or without latches).

Design Example

Consider a circuit with two inputs, D and G and one output, Q. Output Q follows D with G=1, otherwise Q holds its value.

□ Assume fundamental mode operation – only one input changes at a time

state	Inputs D      G		Output Q	Behavior : trnsfer D to o/p if G is 1; retain D value if G →0
a	0	1	0	Transfer D to Q
b	1	1	1	Transfer D to Q
c	0	0	0	Keep previous Q=0; after a or d
d	1	0	0	Keep previous Q=0; after c
e	1	0	1	Keep previous Q=1; after b or f
f	0	0	1	Keep previous Q=1; after e

state	Inputs D      G		Output Q	Behavior : trnsfr D to o/p if G is 1; retain D value if G →0
a	0	1	0	Transfr D to Q
b	1	1	1	Transfr D to Q
c	0	0	0	Keep previous Q=0; after a or d
d	1	0	0	Keep previous Q=0; after c
e	1	0	1	Keep previous Q=1; after b or f
f	0	0	1	Keep previous Q=1; after e

□ Note: Outputs depend only on state (Moore-like):

curr state	next state				output Q
	DG=00	DG=01	DG=10	DG=11	
a	c	ⓐ	-	b	0
b	-	a	e	ⓑ	1
c	ⓒ	a	d	-	0
d	c	-	ⓓ	b	0
e	f	-	ⓔ	b	1
f	ⓕ	a	e	-	1

Design Example – Primitive Flow Table  
ONLY ONE STABLE STATE PER ROW

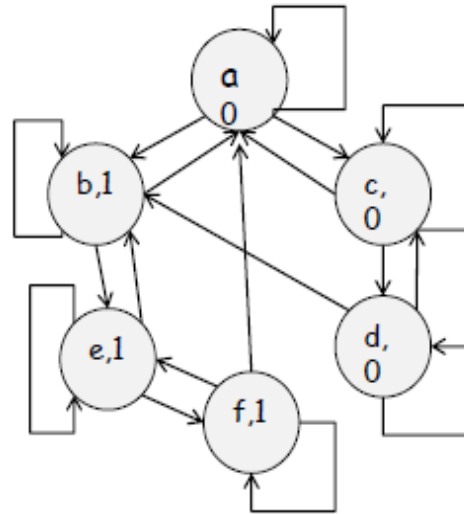
□ Note: Some unspecified entries due to the fundamental mode assumption (e.g., in state a, DG=01, so we never go from DG=01 → DG=10)...

### Design Example – Reduced Flow Table

For the moment, assume that the following flow table will also work for the verbal problem description – assume (a,c,d) and (b,e,f) can be merged.

Original flow table:

curr state	next state				output Q
	DG=00	DG=01	DG=10	DG=11	
a	c	a	-	b	0
b	-	a	e	b	1
c	c	a	d	-	0
d	c	-	d	b	0
e	f	-	e	b	1
f	f	a	e	-	1



Reduced flow table:

curr state	next state				output Q
	DG=00	DG=01	DG=10	DG=11	
a	a	a	a	b	0
b	b	a	b	b	1

**Design Example - State Assignment and Transition Table**

We only have two states, so we can let a=0, and b=1.

Our transition table becomes:

curr state (y)	next state (Y)				output Q
	DG=00	DG=01	DG=10	DG=11	
0	0	0	0	1	0
1	1	0	1	1	1

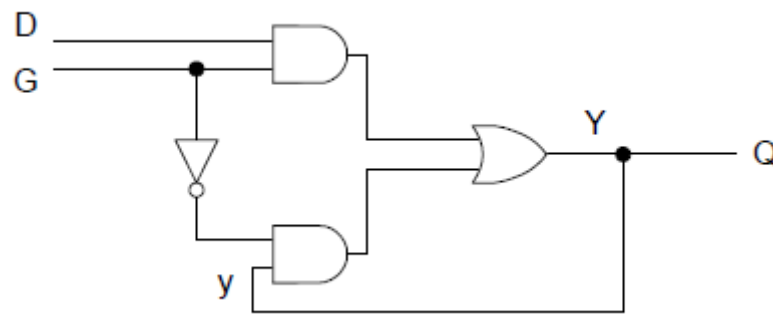
**Design Example - Logic Equations**

We can make K-Maps to determine excitation variables (Y) and output (Z) in terms of circuit inputs and secondary variables (y):

		DG				
		00	01	11	10	
y	0	0	0	1	0	$Y = DG + G'y$
	1	1	0	1	1	

Output equal to the secondary (state) variable.

Can finally draw the circuit:



### Implementation Using Latches

We can also implement asynchronous circuits using latches at the outputs.

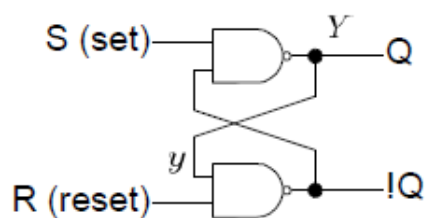
□ Given the map for each excitation variable Y, derive necessary equations for S and R of a latch to produce Y.

□ Derive Boolean equations for S and R.

□ Need to make sure the S and R never have equal (potential problem in Latch).

Implementation Using Latches – SR Latch Excitation Table

Recall an S'R' Latch (NAND) works:



S	R	Q	$\bar{Q}$
1	0	0	1
1	1	0	1
0	1	1	0
1	1	1	0
0	0	1	1

(after  $S = 1, R = 0$ )

(after  $S = 0, R = 1$ )

Assuming we never have the  $SR=00$  case. Can write **excitation table**:

S	R	y	Y
1	X	0	0
0	1	0	1
1	0	1	0
X	1	1	1

## Implementation

## Using

## Latches

Consider our example again, and assume we want to use a S'R' Latch:

DG					
y	00	01	11	10	
0	0	0	1	0	
1	1	0	1	1	

$$Y = DG + G'y$$

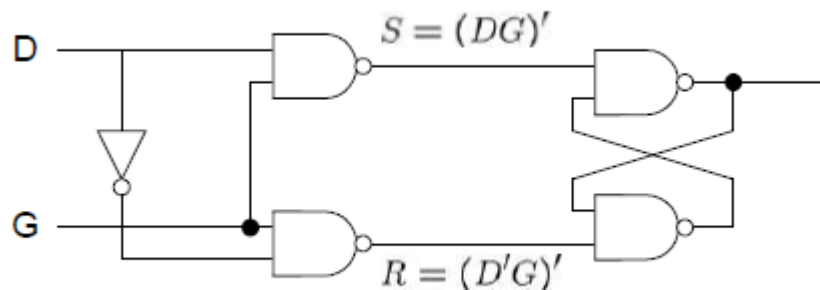
Need to figure out how to select S and R for the NAND Latch (while making sure never 0 at same time):

DG									
y	00	01	11	10					
0	1	1	0	1					
1	X	1	X	X					

$$S = (DG)'$$

S	R	y	Y
1	X	0	0
0	1	0	1
1	0	1	0
X	1	1	1

DG									
y	00	01	11	10					
0	X	X	1	X					
1	1	0	1	1					

$$R = (D'G)'$$


## Output Assignment

Flow and transition tables might have unspecified entries for circuit outputs.

⌚ This might be a result of the fundamental mode assumption.

⌚ This might be a result of unstable states.

⌚ Note: output values always assigned for stable states!

⌚ We should think about the correctness of these unspecified don't care output Values.

⌚ We might temporarily pass through these values while transitioning from one stable state to another stable state.

**Example:**

Consider the following flow table with don't cares at some outputs (circuit has one input and one output):

curr state	next state		output	
	x=0	x=1	x=0	x=1
a	(a)	b	0	-
b	c	(b)	-	0
c	(c)	d	1	-
d	a	(d)	-	1

We might consider using the un-specified output values as don't cares in order to minimize the logic function for the output. ❑ We need to be careful with output don't cares in asynchronous design.

❑ Consider start and stop STABLE STATES due to a change in input value.

❑ If both stable states produce a 0 output, make output 0 instead of a don't care.

❑ If both stable states produce a 1 output, make output 1 instead of a don't care.

❑ If stable states produce different outputs, the output can remain a don't care and be used to find a smaller output circuit.

❑ We do this to avoid GLITCHES in the output (e.g., if the output should go 0->0 (or 1->1), it should remain 0 (or 1) during the transition through an unstable state.

**Example:**

Recall the flow table... If we consider possible transitions, we see that some of the output don't cares should be changed to 0 or 1 to avoid GLITCHES.

curr state	next state		output	
	x=0	x=1	x=0	x=1
a	(a)	b	0	-
b	c	(b)	-	0
c	(c)	d	1	-
d	a	(d)	-	1

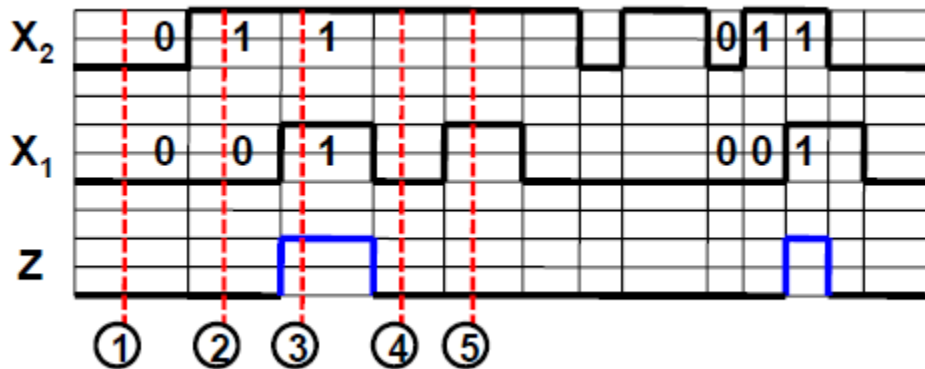
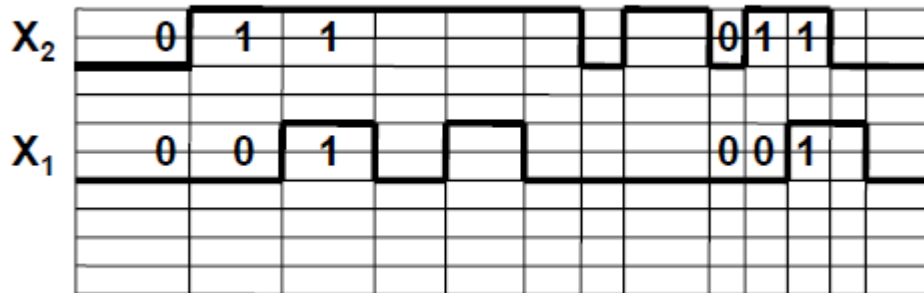
curr state	next state		output	
	x=0	x=1	x=0	x=1
a	(a) →	b	0	0
b	c	(b) ↓	-	0
c	(c) →	d	1	1
d	a	(d) ↓	-	1

The above changes will avoid temporary glitches at the outputs during transitions where the output should not change.

**Fundamental Mode Circuit Design**

Design a fundamental mode sequential circuit with two inputs  $X_2$  and  $X_1$ . The single output  $Z$  is to be 1 only when  $X_2X_1 = 11$ , provided that this is the third of a sequence of input combinations 00 10 11. Otherwise, the output is to be 0. The design must be sure of no spurious 1 outputs will occur during transitions between two states with 0 outputs. Both inputs will not change simultaneously.

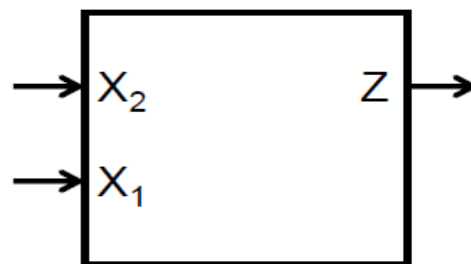
$X_2X_1 \Rightarrow 00\ 10\ 11$        $Z = 1$



### Fundamental Mode Circuit Design

	$X_2, X_1$			
	00	01	11	10
①	①, 0			
②				
③				
④				
⑤				
⑥				

00, 0    ①, 0





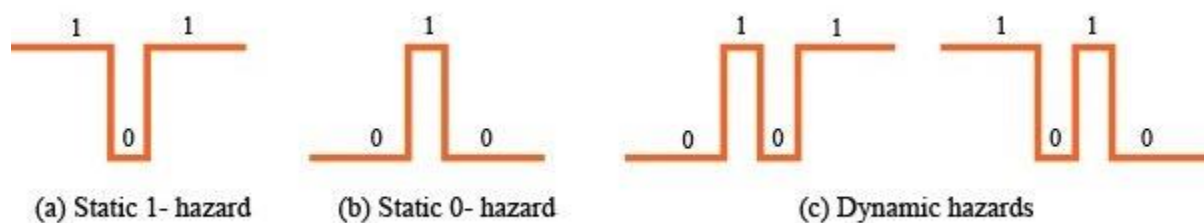
### What are Hazards ?

Hazards in any system are obviously an un-desirable effect caused by either a deficiency in the system or external influences. Logic hazards are manifestations of a problem in which changes in the input variables do not change the output correctly due to some form of delay caused by logic elements (NOT, AND, OR gates, etc.) This results in the logic not performing its function properly. The three different most common kinds of hazards are usually referred to as static, dynamic and function hazards. Hazards are a temporary problem, as the logic circuit will eventually settle to the desired function. Therefore, in synchronous designs, it is standard practice to register the output of a circuit before it is being used in a different clock domain or routed out of the system, so that hazards do not cause any problems. If that is not the case, however, it is imperative that hazards be eliminated as they can have an effect on other connected systems.

### Hazards in Combinational Logic

If the input of a combinational circuit changes, unwanted switching variations may appear in the output. These variations occur when different paths from the input to output have different delays. If, from response to a single input change and for some combination of propagation delay, an output momentarily goes to 0 when it should remain a constant value of 1, the circuit is said to have a static 1-hazard. Likewise, if the output momentarily goes to 1 when it should remain at a constant value of 0, the circuit is said to have a 0-hazard.

When an output is supposed to change values from 0 to 1, or 1 to 0, this output may change three or more times; if this situation were to occur, the circuit is said to have a dynamic hazard. Figure 1.1 shows the different outputs from a circuit with hazards. In each of the three cases, the steady-state output of the circuit is correct, however, a switching variation appears at the circuit output when the input is changed.

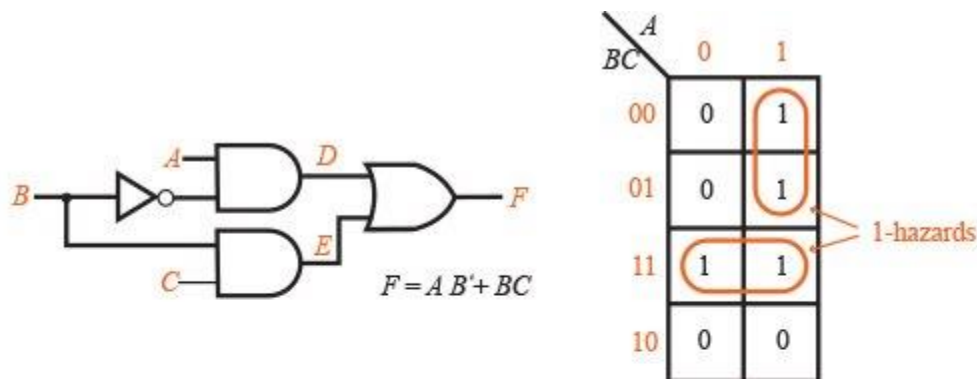


The first hazard in Figure 1.2, the static 1-hazard depicts that if  $A = C = 1$ , then  $F = B + B' = 1$ , thus the output F should remain at a constant 1 when B changed from 1 to 0. However in the next illustration, the static 0-hazard, if each gate has a propagation of 10 ns, E will go to 0 before D goes to 1, resulting in a momentary 0 appearing at output F. This is also known to be a glitch caused by the 1-hazard. One should note that right after B changes to 0, both the

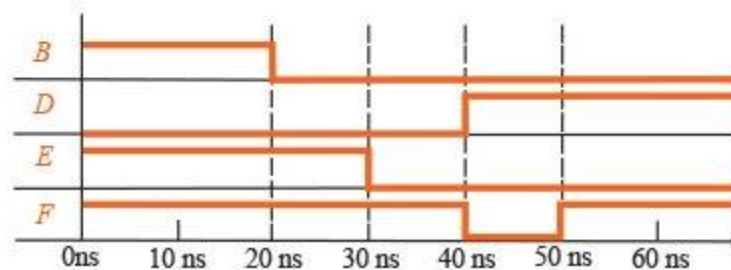
inverter input (B) and output (B') are 0 until the delay has elapsed. During this propagation period, both of these input terms in the equation for F have value of 0, so F also momentarily goes to a value of 0. These hazards, static and dynamic, are completely independent of the propagation delays that exist in the circuit. If a combinational circuit has no hazards, then it is said that for any combination of propagation delays and for any individual input change, that output will not have a variation in I/O value. On the contrary, if a circuit were to contain a hazard, then there will be some combination of delays as well as an input change for which the output in the circuit contains a transient.

This combination of delays that produce a glitch may or may not be likely to occur in the implementation of the circuit. In some instances it is very unlikely that such delays would occur. The transients (or glitches) that result from static and dynamic timing hazards very seldom cause problems in fully synchronous circuits, but they are a major issue in asynchronous circuits (which includes nominally synchronous circuits that involve either the use of asynchronous preset/reset inputs that use gated clocks).

The variation in input and output also depends on how each gate will respond to a change of input value. In some instances, if more than one input gate changes within a short amount of time, the gate may or may not respond to the individual input changes. One example in Figure 1.2, assuming that the inverter (B) has a propagation delay of 2ns instead of 10ns. Then input D and E changes reaching the output OR gate are 2ns from each other, thus the OR gate may or may not generate the 0 glitch.



(a) Circuit with a static 1-hazard



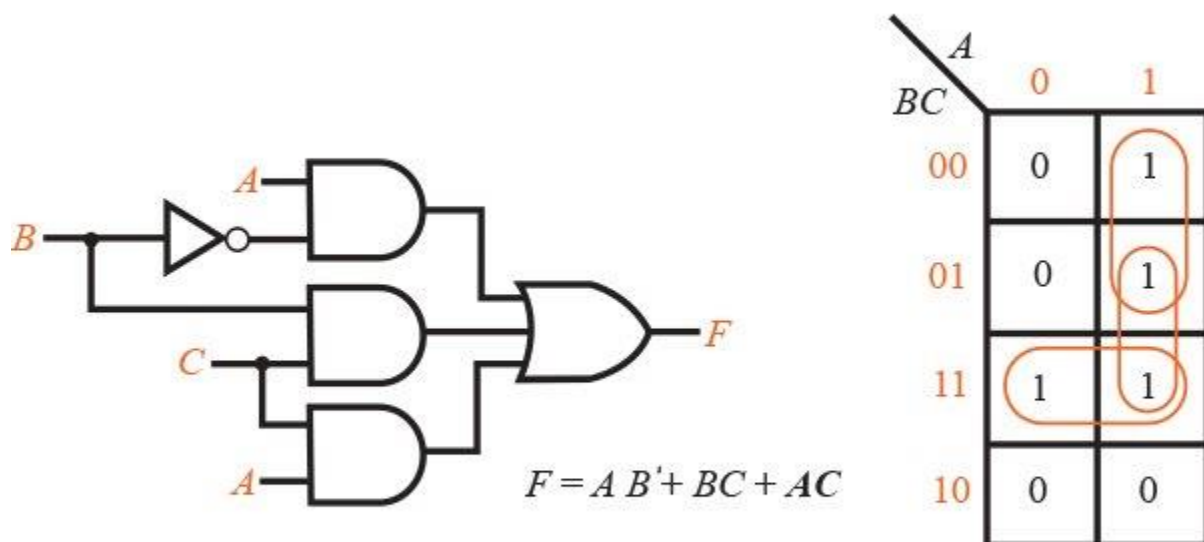
(b) Timing chart

A gate displaying this type of response is said to have what is known as an inertial delay. Rather often the inertial delay value is presumed to be the same as the propagation delay of the gate. When this occurs, the circuit above will respond with a 0 glitch only for inverter propagation delays that are larger than 10ns. However, if an input gate invariably responds to input change that has a propagation delay, is said to have an ideal or transport delay. If the OR gate shown above has this type of delay, then a 0 glitch would be generated for any nonzero value for the inverter propagation delay.

Hazards can always be discovered using a Karnaugh map. The map illustrated above in Figure 1.2, which not a single loop covers both minterms  $ABC$  and  $AB'C$ . Thus if  $A = C = 1$  and  $B$ 's value changes, both of these terms can go to 0 momentarily; from this momentary change, a 0 glitch is found in  $F$ . To detect hazards in a two-level AND-OR combinational circuit, the following procedure is completed:

A sum-of-products expression for the circuit needs to be written out. Each term should be plotted on the map and looped, if possible. If any two adjacent 1's are not covered by the same loop, then a 1-hazard exists for the transition between those two 1's. For any  $n$  variable map, this transition only occurs when one variable changes value and the other  $n - 1$  variables are held constant.

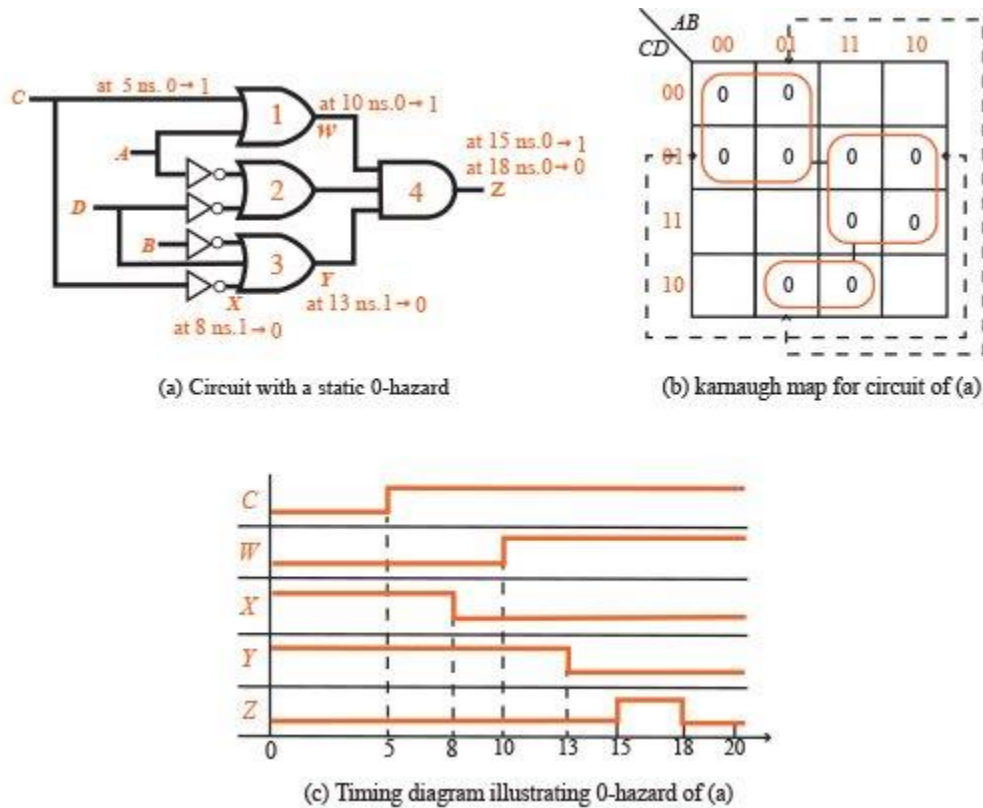
If another loop is added to the Karnaugh map in Fig. 1.2(a) and then add the corresponding gate to the circuit in Figure 1.3 below, the hazard can be eliminated. The term  $AC$  remains at a constant value of 1 while  $B$  is changing, thus a glitch cannot appear in the output. With this change,  $F$  is no longer a minimum SOP.



The above is a circuit with numerous 0-hazards. The function that represents the circuit's output is:

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

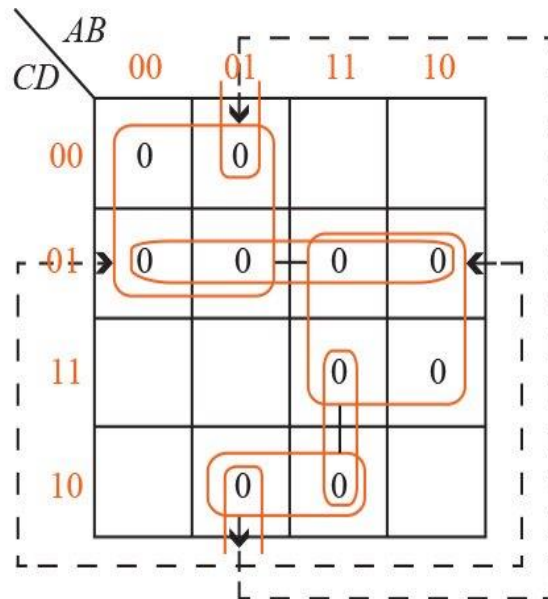
The Karnaugh map in Fig. 1.4(b) has four pairs of adjacent 0's that are not covered by a common loop. The arrows indicate where each 0 is not being looped, and they each correspond to a 0-hazard. If  $A = 0$ ,  $B = 1$ ,  $D = 0$ , and  $C$  changes from 0 to 1, there is a chance that a spike can appear at the output for any combination of gate delays. lastly, Fig. 1.4(c) depicts a timing diagram that, assumes a delay of 3ns for each individual inverter and a delay of 5ns for each AND gate and each OR gate.



The 0-hazards can be eliminated by looping extra prime implicants that cover the 0's adjacent to one another, as long as they are not already covered by a common loop. By eliminating algebraically redundant terms, or consensus terms, the circuit can be reduced to the following equation below. Using three additional loops will completely eliminate the 0-hazards, resulting the following equation:

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

This figure below illustrates the Karnaugh map after removing the 0-hazards.



In digital logic hazards are usually referred to in one of three ways:

- Static Hazards
- Dynamic Hazards
- Function Hazards
- 

### Static Hazards

A static hazard is the situation where, when one input variable changes, the output changes momentarily before stabilizing to the correct value. There are two types of static hazards:

- Static-1 Hazard: the output is currently 1 and after the inputs change, the output momentarily changes to 0,1 before settling on 1
- Static-0 Hazard: the output is currently 0 and after the inputs change, the output momentarily changes to 1,0 before settling on 0

In properly formed two-level AND-OR logic based on a Sum Of Products expression, there will be no static-0 hazards. Conversely, there will be no static-1 hazards in an OR-AND implementation of a Product Of Sums expression.

The most commonly used method to eliminate static hazards is to add redundant logic (consensus terms in the logic expression).

Let us consider an imperfect circuit that suffers from a delay in the physical logic elements i.e. AND gates etc. The simple circuit performs the function noting:

$$f = X_1 * X_2 + X_1' * X_3$$

If we first look at the starting diagram, it is clear that if no delays were to occur, then the circuit would function normally. However, no two gates are ever manufactured exactly the same. Due to this imperfection, the delay for the first AND gate will be slightly different than its counterpart. Thus an error occurs when the input changes from 111 to 011. i.e. when X1 changes state.

Now we know roughly how the hazard is occurring, for a clearer picture and the solution on how to solve this problem, we would look to the Karnaugh map. The two gates are shown by solid rings, and the hazard can be seen under the dashed ring. A theorem proved by Huffman<sup>[1]</sup> tells us that by adding a redundant loop 'X2X3' this will eliminate the hazard.

So our original function is now:  $f = X_1 * X_2 + X_1' * X_3 + X_2 * X_3$

Now we can see that even with imperfect logic elements, our example will not show signs of hazards when X1 changes state. This theory can be applied to any logic system. Computer programs deal with most of this work now, but for simple examples it is quicker to do the debugging by hand. When there are many input variables (say 6 or more) it will become quite difficult to 'see' the errors on a Karnaugh map.

*Definition:- "When one input variable changes, the output changes momentarily when it shouldn't"*

This particular type of hazard is usually due to a NOT gate within the logic. We can see the effects of the delay in the circuit from the following flash animation.

The hazard can be dealt with in two ways:

1. Insert another (additional) delay to the circuit. This then eliminates the static hazard.
2. Eliminate the hazard by inserting more logic to counteract the effects (Note this makes assumptions that the logic will fail)

The first case is the most used of the two options. This is because it does not make assumptions about the logic, instead the method adds redundancy to overcome the hazard.

To solve the hazard we shall use our previous example and apply a theory that 'Huffman' discovered. The insertion of a redundant loop can eliminate a static hazard.

In the next example, it will also be evident that there will not be a situation where a static '0' occurs. A static '0' hazard is one which briefly goes to '1' when it should remain at '0'. A static '1' hazard is the reverse of this situation, i.e. the output should remain at '1' yet under some condition it briefly changes state to '0' (something we shall see in the following example)..

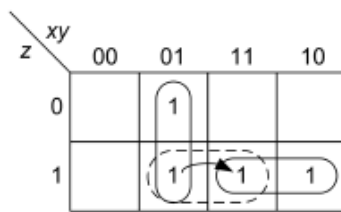
### **Example of Static Hazards**

The Static '1' Hazard.

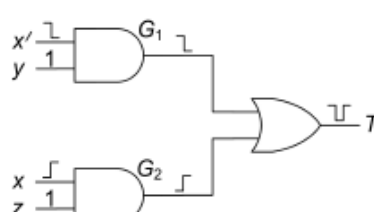
Let us consider an imperfect circuit that suffers from a delay in the physical logic elements i.e. AND gates etc.

**Example:**  $T(x,y,z) = \sum(2,3,5,7)$

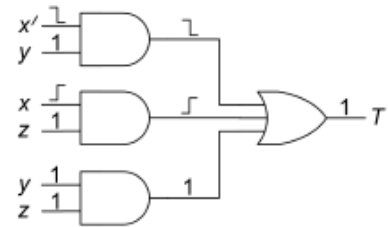
- Static-1 logic hazard (SIC)



(a) Map for  $T = x'y + xz$ .



(b) Gate network.



(c) SIC hazard-free network

**Adjacent combinations:** differ in the value of a single variable

- E.g.,  $x'yz$  and  $xyz$

**SIC static logic hazard:** transition between a pair of adjacent input combinations, which correspond to identical output values, that may generate a momentary spurious output value

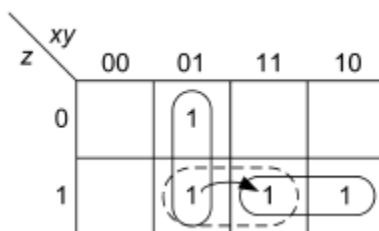
- Occurs when no cube in the K-map contains both combinations
  - **Solution:** cover both combinations with a cube

Transition cube  $[m1, m2]$ : set of all minterms that can be reached from minterm  $m1$  and ending at minterm  $m2$

Example: Transition cube  $[010, 100]$  contains: 000, 010, 100, 110 Required cube: transition cube that must be included in some product of

the sum-of-products realization in order to get rid of the static-1logic hazard

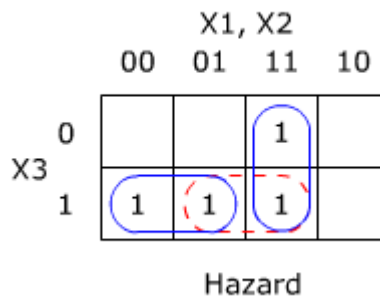
Example: Required cube is  $[011, 111]$



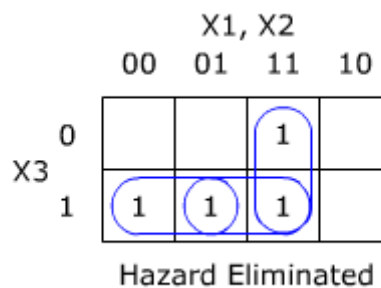
The simple circuit performs the function:

$f = X1.X2 + X1'.X3$  and the logic diagram can be shown as follows:

Now we know roughly how the hazard is occurring, for a clearer picture and the solution on how to solve this problem, we look to the Karnaugh Map:



This Karnaugh Map shows the circuit. The two gates are shown by solid rings, and the hazard can be seen under the dashed ring. The theory proved by Huffman tells us that by adding a redundant loop 'X2X3' this will eliminate the hazard. So the resulting logic is of the form shown in the next figure.

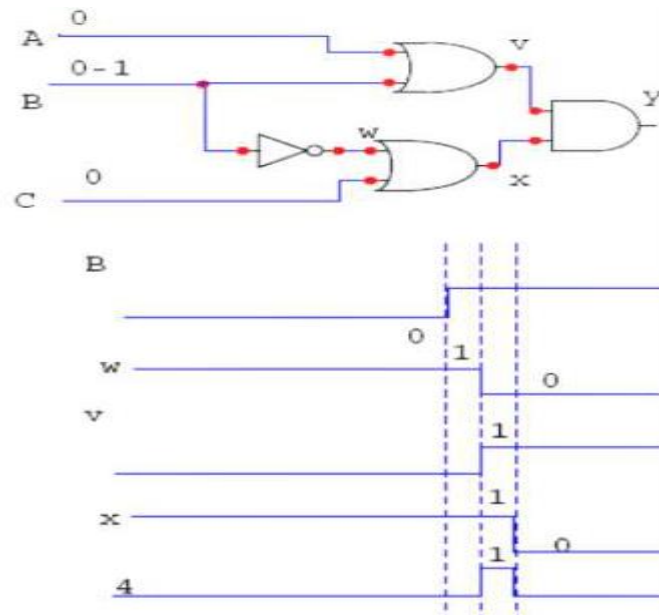


So our original function is now:  $f = X1.X2 + X1'.X3 + X2.X3$

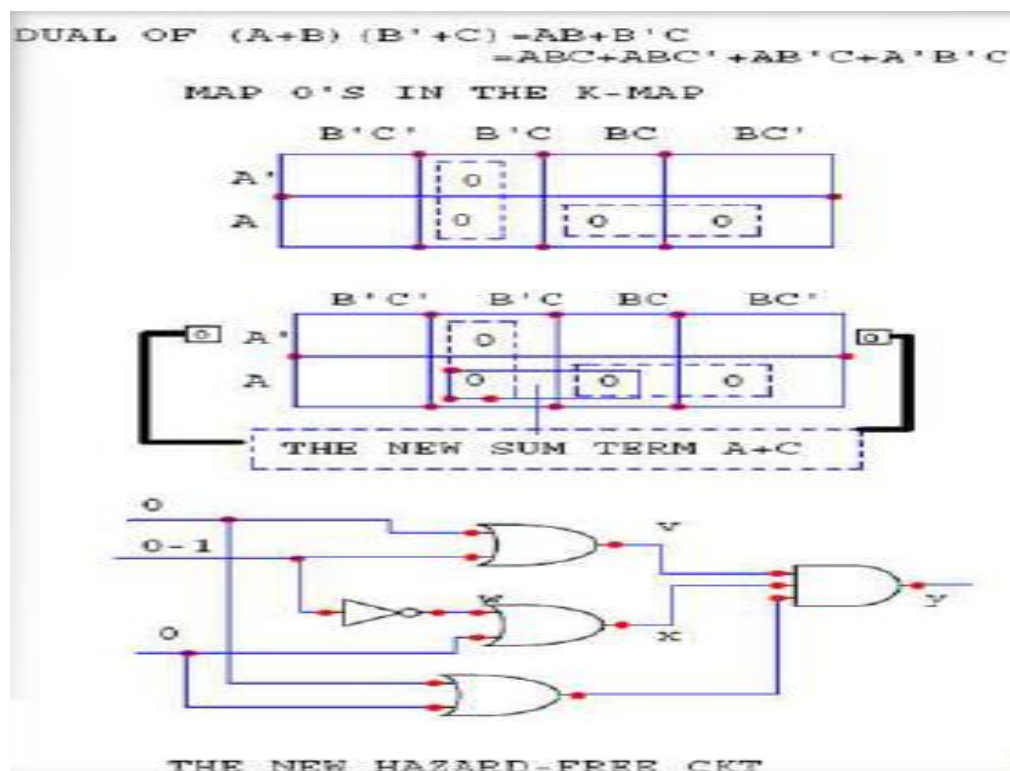
### static-0 hazard

The output should be 0 but goes momentary to 1 as a result of an input change. A static-0 hazard occurs in OR-AND circuits when an input variable and its complement are connected to two different OR gates.



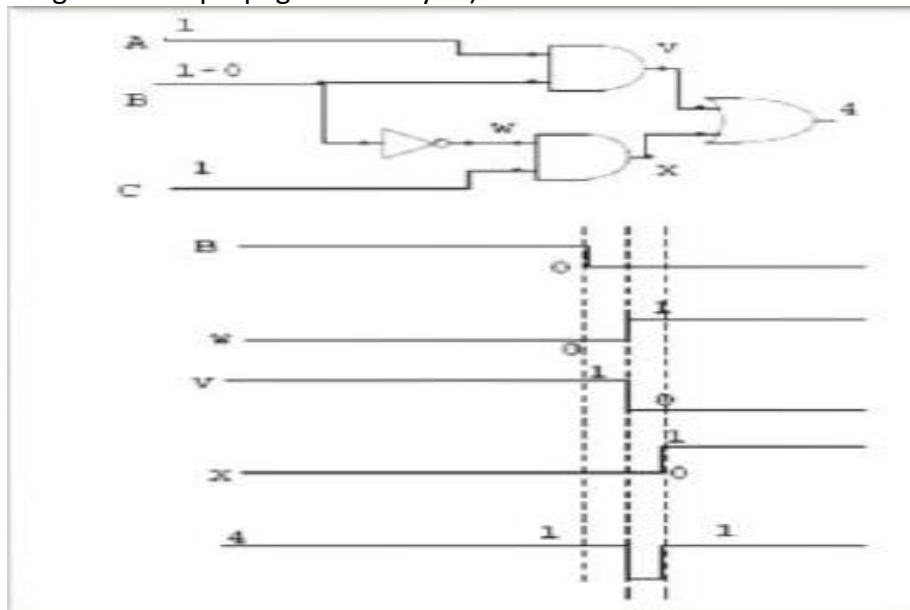


- The procedure to find and eliminate static-0 hazards using K-maps is done in a dual way to finding static-1 hazards.
- Static-0 hazards are found using kmaps by finding adjacent 0 cells that are covered by different sum terms.
- To eliminate static-0 hazards, additional sum terms (prime implicants) are needed to cover such cells thus covering the transition of the variable causing the hazard.



**static-1 hazard**

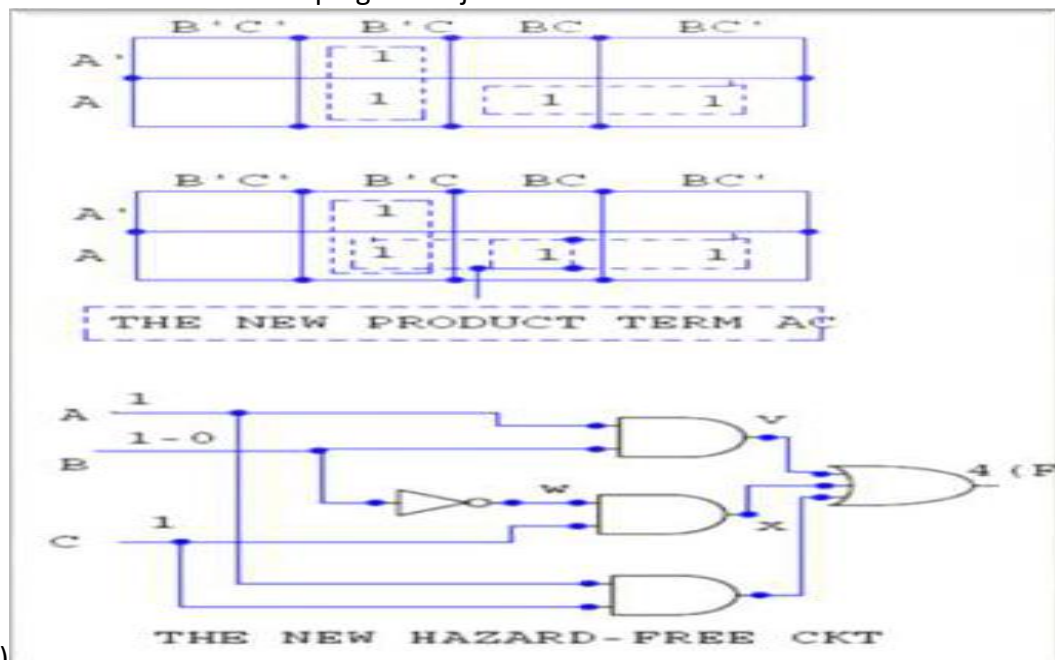
A static-1 hazard exists in the following AND-OR circuit when  $A = 1$ ,  $C = 1$  and  $B$  changes from 1 to 0 (assume all gates have propagation delay  $D$ ):



Static-1 hazards are found using k-maps by finding adjacent 1 cells that are covered by different product terms.

> To eliminate static-1 hazards, additional product terms (prime implicants) are needed to coversuch cells thus covering the transition of the variable causing the hazard.>For in the previous example the static-1 hazard is eliminated by including the additional product term  $AC$

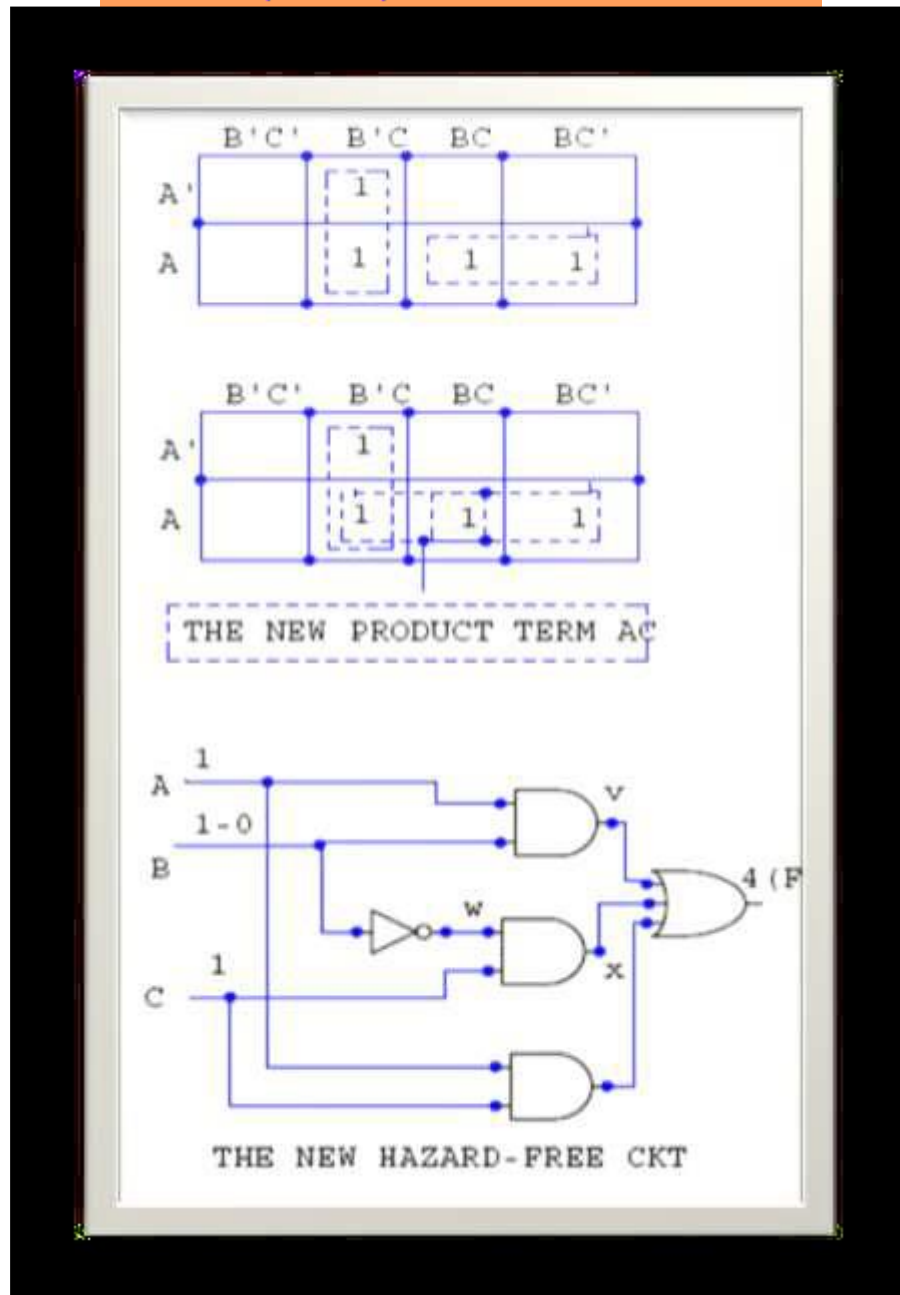
Grouping the adjacent 1's in the two



groups)

$$f = AB + B'C = ABC' + ABC + AB'C + A'B'C \text{ WITH HAZARD}$$

$$f = A B + B' C + A C: \text{Hazard eliminated}$$



Now we can see that even with imperfect logic elements, our example will not show signs of hazards when X1 changes state. This theory can be applied to any logic system. Computer programs deal with most of this work now, but for simple examples it is quicker to do the debugging by hand. When there are many input variables (say 6 or more) Dynamic Hazards

Definition:- "A dynamic hazard is the possibility of an output changing more than once as a result of a single input change"

Dynamic hazards often occur in larger logic circuits where there are different routes to the output (from the input). If each route has a different delay, then it quickly becomes clear that there is the potential for changing output values that differ from the required / expected output.

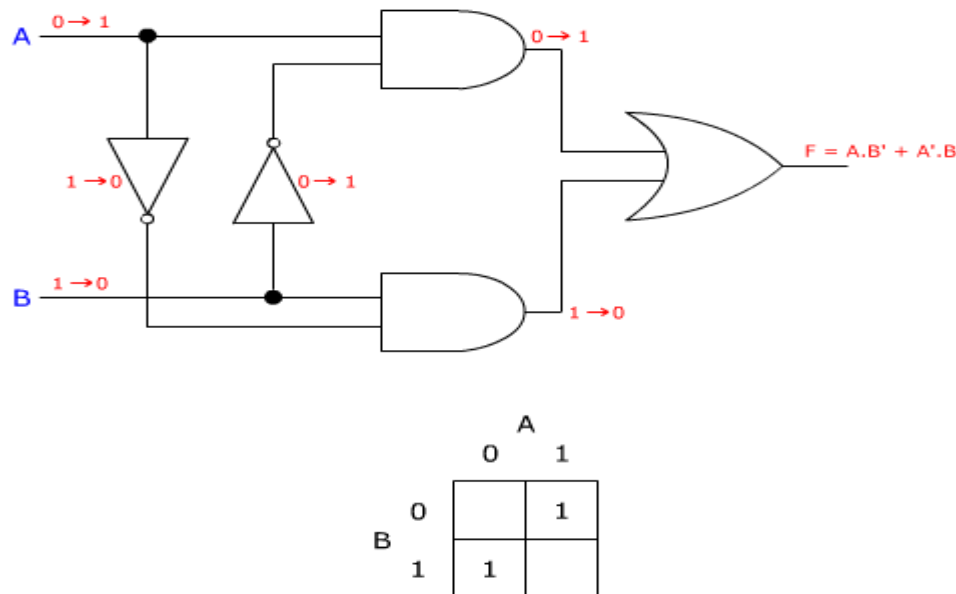
e.g. A logic circuit is meant to change output state from '1' to '0', but instead changes from '1' to '0' then '1' and finally rests at the correct value '0'. This is a dynamic hazard.

As we shall see, dynamic hazards take a more complex method to resolve (which we shall not cover). Let us explain this more with a slide show.

### Function Hazards

Function hazards are non-solvable hazards which occurs when more than one input variable changes at the same time. Hazards such as function hazards can not be logically eliminated as the problem lies with actual specification of the circuit. The only real way to avoid such problems is to restrict the changing of input variables so that only one input should change at any given time. Restrictions are not always possible, for instance let us imagine some logic circuit that has two inputs. One input is used for a clock signal, and the other is connected to a random noise source that we wish to measure. It should be clear that restrictions in this case would not be an effective solution.

The simplest example of this is the exclusive-or function.



In this scenerio it is quite difficult to see how a hazard could occur if the circuit is built up on the same couple of chips. However let us imagine that some circuit designer has split this function across different chips (i.e. one NOT gate on one chip and the other NOT gate is implemented on another chip across the PCB somewhere)

Let us setup the initial state of our circuit.  $A = 1$ ,  $B = 0$ . Now let's say there is a delay in the NOT gate marked (X). The inputs now change simultaneously so that  $A = 0$  and  $B = 1$  (remember in a equally delayed circuit or a perfect circuit, the circuit output would match the specification). If we observe what the circuit should do, and do not change the output of the NOT gate X (this simulates a delay in gate X), it should be clear that the output of the circuit changes. Now we change the output of NOT gate X and the circuit goes back to the proper state.

The most effective way to solve this hazard would be to carefully design the PCB so that delays are all equal, or at least match the delays on each path. i.e. Delay of A's path = Delay of B's path. Yet adding more gates to the circuit by the same methods as described in dynamic and static hazards will not work as Huffman's method cannot be applied.

## UNIT III

### Digital Design

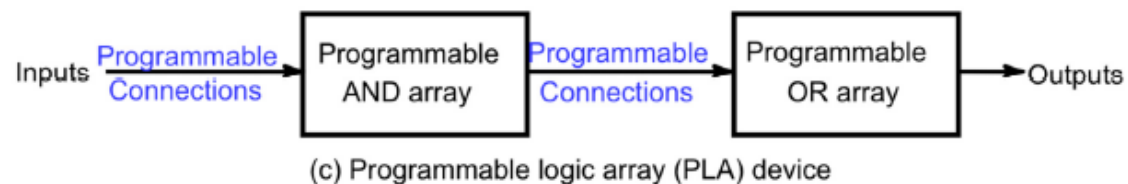
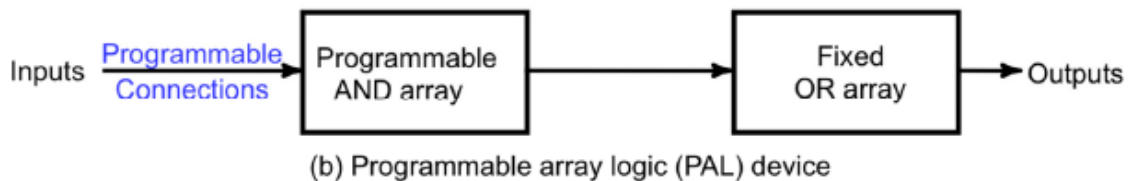
#### Programmable Logic Devices

Read Only Memory (ROM) - a fixed array of AND gates and a programmable array of OR gates

\_ Programmable Array Logic (PAL) - a programmable array of AND gates feeding a fixed array of OR gates.

\_ Programmable Logic Array (PLA) - a programmable array of AND gates feeding a programmable array of OR gates.

\_ Complex Programmable Logic Device (CPLD) /Field- Programmable Gate Array (FPGA) - complex enough to be called "architectures"



#### READ ONLY MEMORY

\_ Read Only Memories (ROM) or Programmable Read Only Memories (PROM) have:

- N input lines,
- M output lines, and
- $2^N$  decoded minterms.

\_ Fixed AND array with  $2^N$  outputs implementing all N-literal minterms.

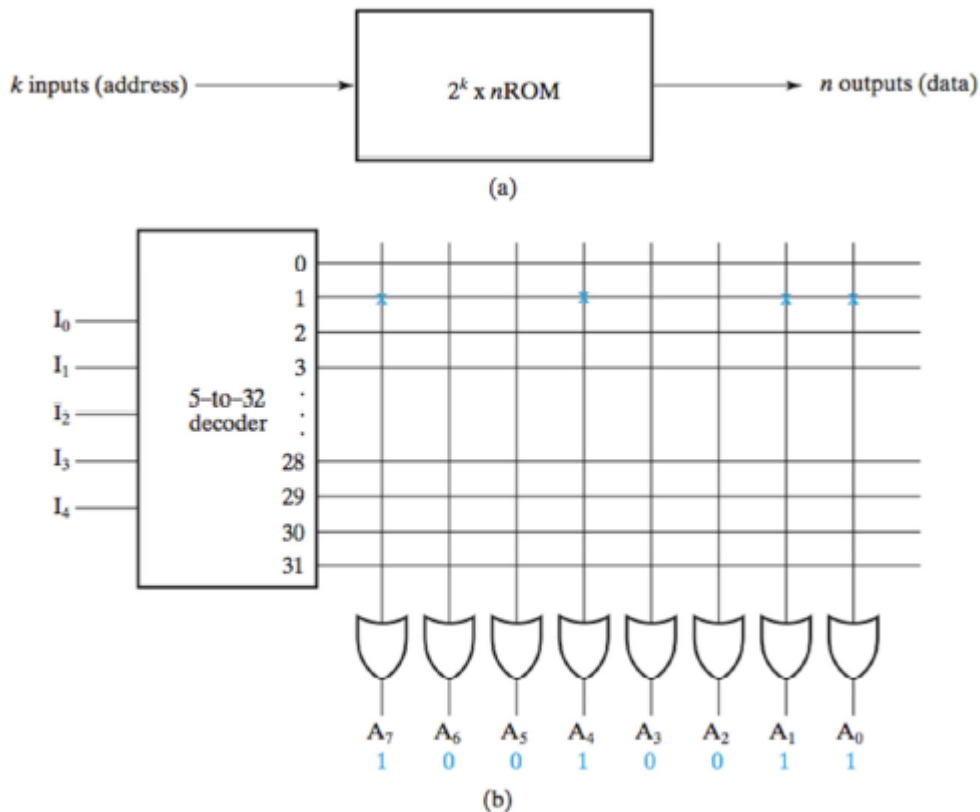
\_ Programmable OR Array with M outputs lines to form up to M sum of minterm expressions.

\_ A program for a ROM or PROM is simply a multiple-output truth table

- If a 1 entry, a connection is made to the corresponding minterm for the corresponding output

- If a 0, no connection is made

\_ Can be viewed as a memory with the inputs as addresses of data (output values), hence ROM or PROM names!



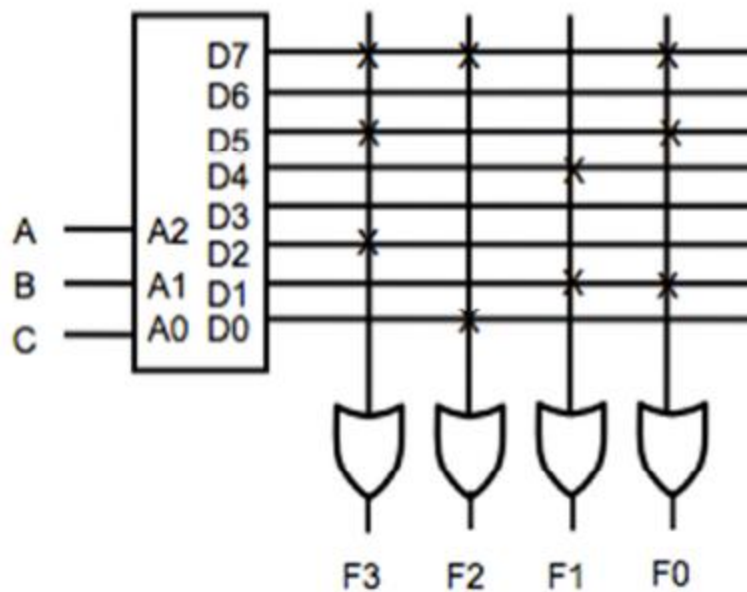
**Figure: Block diagram and Internal Logic of a ROM**

Depending on the programming technology and approaches, read-only memories have different names

1. ROM – mask programmed
2. PROM – fuse or antifuse programmed
3. EPROM – erasable floating gate programmed
4. EEPROM or E2PROM – electrically erasable floating gate programmed
5. FLASH memory: electrically erasable floating gate with multiple erasure and programming modes.

\_ Example: A 8 X 4 ROM (N = 3 input lines, M= 4 output lines)

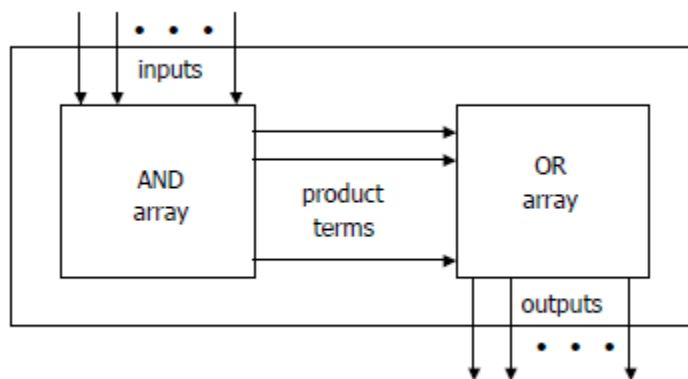
- The fixed "AND" array is a "decoder" with 3 inputs and 8 outputs implementing minterms.
- The programmable "OR" array uses a single line to represent all inputs to an OR gate. An "X" in the array corresponds to attaching the minterm to the OR
- Read Example: For input  $(A_2, A_1, A_0) = 011$ , output is  $(F_3, F_2, F_1, F_0) = 0011$ .
- What are functions  $F_3, F_2, F_1$  and  $F_0$  in terms of  $(A_2, A_1, A_0)$ ?



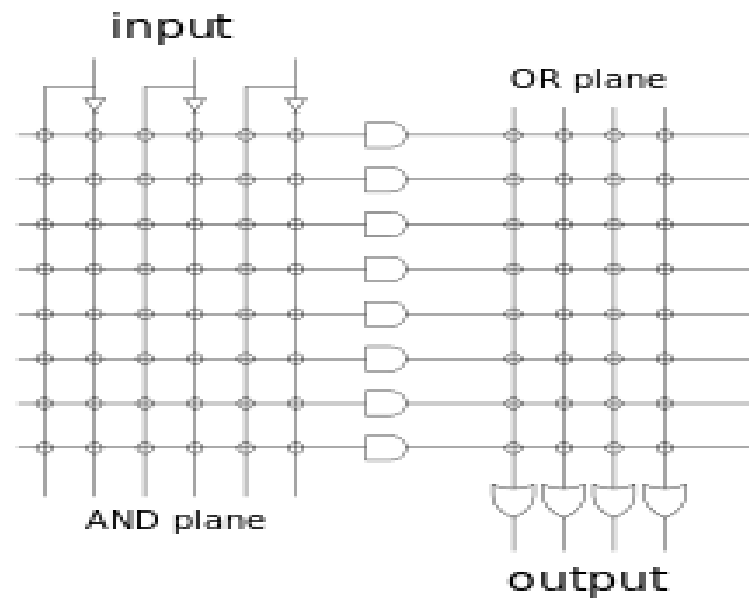
### PROGRAMMABLE LOGIC ARRAY (PLA)

A **programmable logic array (PLA)** is a kind of programmable logic device used to implement combinational logic circuits. The PLA has a set of programmable and gate planes, which link to a set of programmable or gateplanes, which can then be conditionally complemented to produce an output. It has  $2^n$  and gates for  $n$  input variables and for  $m$  outputs from PLA, there should be  $m$  or gates, each with programmable inputs from all of the and gates. This layout allows for a large number of logic functions to be synthesized in the sum of products canonical forms. PLAs differ from programmable array logic devices (pals and gals) in that both the and and or gate planes are programmable.

PLA schematic example







Compared to a ROM and a PAL, a PLA is the most flexible having a programmable set of ANDs combined with a programmable set of ORs.

### ■ Shared product terms among outputs

example:

$$\begin{aligned} F0 &= A + B' C' \\ F1 &= A C' + A B \\ F2 &= B' C' + A B \\ F3 &= B' C + A \end{aligned}$$

personality matrix

product term	inputs			outputs			
	A	B	C	F0	F1	F2	F3
AB	1	1	–	0	1	1	0
B'C	–	0	1	0	0	0	1
AC'	1	–	0	0	1	0	0
B'C'	–	0	0	1	0	1	0
A	1	–	–	1	0	0	1

input side:  
 1 = uncomplemented in term  
 0 = complemented in term  
 – = does not participate

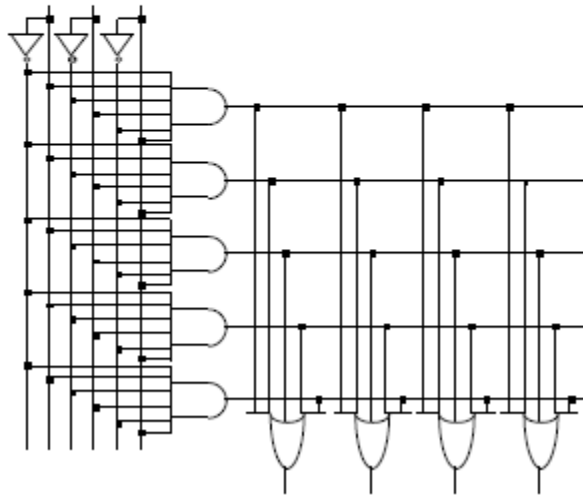
output side:  
 1 = term connected to output  
 0 = no connection to output

reuse of terms

## Before Programming

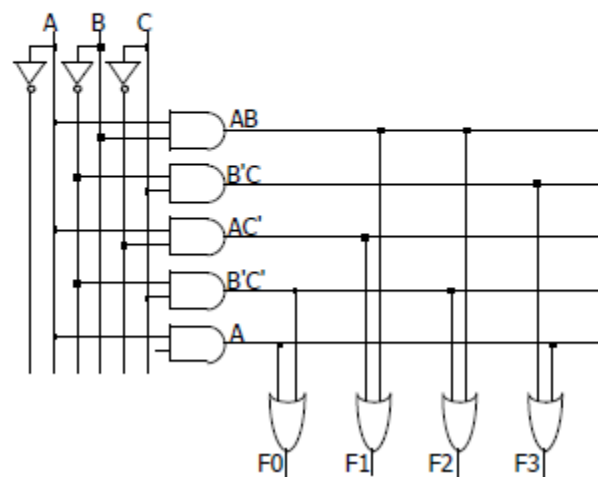
### ■ All possible connections available before "programming"

- In reality, all AND and OR gates are NANDs



### ■ Unwanted connections are "blown"

- Fuse (normally connected, break unwanted ones)
- Anti-fuse (normally disconnected, make wanted connections)

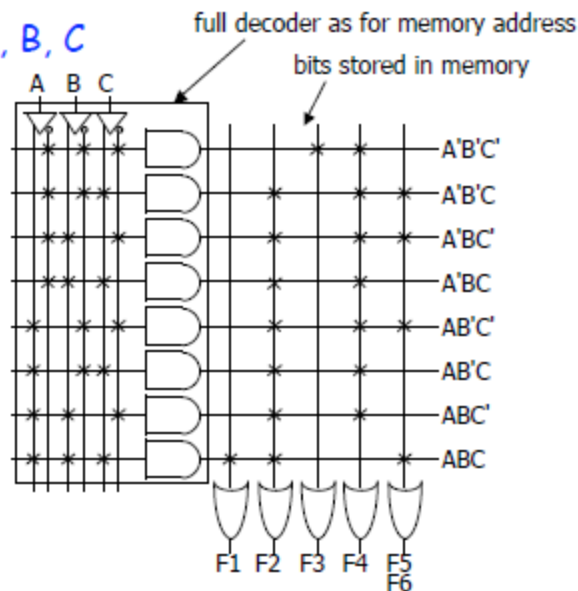


## Programmable Logic Array Example

### Multiple functions of A, B, C

- $F1 = A B C$
- $F2 = A + B + C$
- $F3 = A' B' C'$
- $F4 = A' + B' + C'$
- $F5 = A \text{ xor } B \text{ xor } C$
- $F6 = (A \text{ xnor } B \text{ xnor } C)$

A	B	C	F1	F2	F3	F4	F5	F6
0	0	0	0	0	1	1	0	0
0	0	1	0	1	0	1	1	1
0	1	0	0	1	0	1	1	1
0	1	1	0	1	0	1	0	0
1	0	0	0	1	0	1	1	1
1	0	1	0	1	0	1	0	0
1	1	0	0	1	0	1	0	0
1	1	1	1	1	0	0	1	1



## PLA Design Example

### BCD to Gray code converter

A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	1	1	1	0
0	1	1	0	1	0	1	0
0	1	1	1	1	0	1	1
1	0	0	0	1	0	0	1
1	0	0	1	1	0	0	0
1	0	1	-	-	-	-	-
1	1	-	-	-	-	-	-

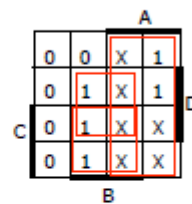
minimized functions:

$$W = A + B D + B C$$

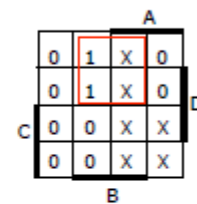
$$X = B C'$$

$$Y = B + C$$

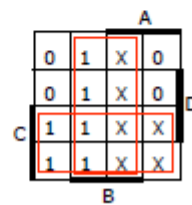
$$Z = A'B'C'D + B C D + A D' + B' C D'$$



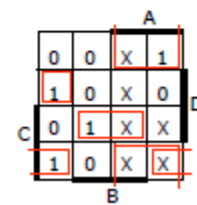
K-map for W



K-map for X



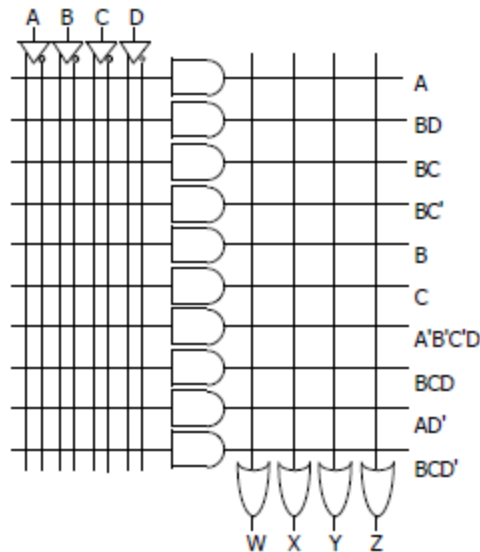
K-map for Y



K-map for Z

## PLA Design Example (cont'd)

### Code converter: programmed PLA



minimized functions:

$$W = A + B D + B C$$

$$X = B C'$$

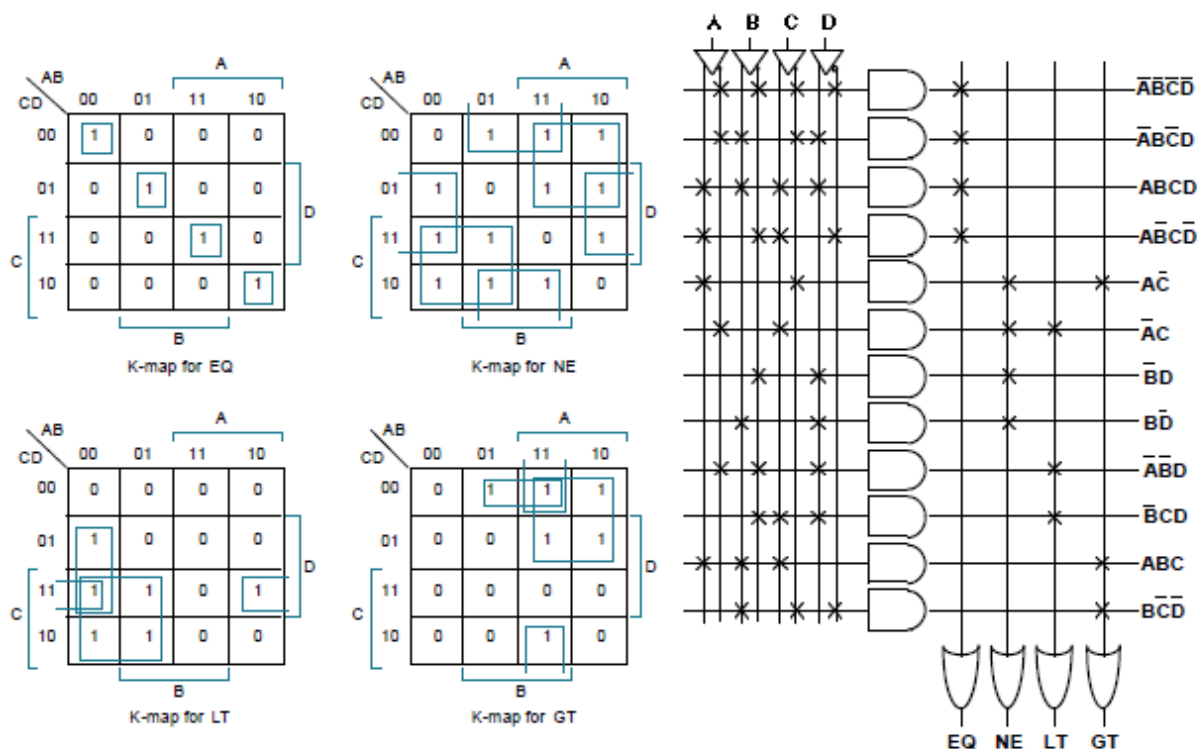
$$Y = B + C$$

$$Z = A'B'C'D + B C D + A D' + B' C D'$$

not a particularly good  
candidate for PLA  
implementation since no terms  
are shared among outputs

however, much more compact  
and regular implementation  
when compared with discrete  
AND and OR gates

### Another PLA Example: Magnitude Comparator



### Advantages

- A PLA can have large N and M permitting implementation of equations that are impractical for a ROM (because of the number of inputs, N, required)
- A PLA has all of its product terms connectable to all outputs, overcoming the problem of the limited inputs to the PAL Ors
- Some PLAs have outputs that can be complemented, adding POS functions

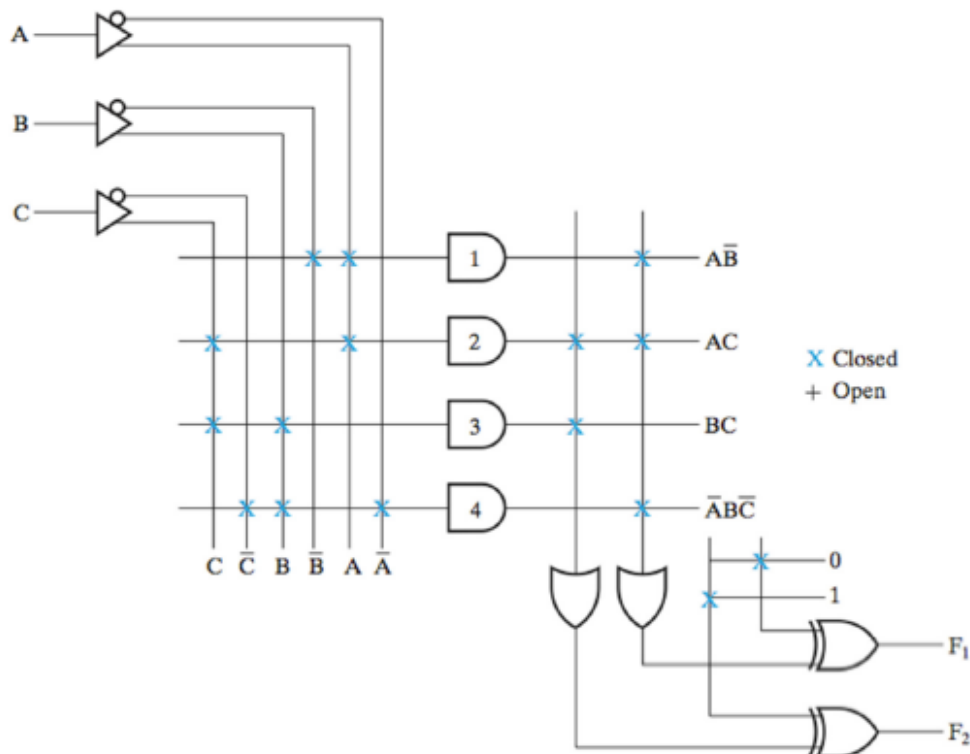
### Disadvantages

- Often, the product term count limits the application of a PLA.
- Two-level multiple-output optimization is required to reduce the number of product terms in an implementation, helping to fit it into a PLA.
- Multi-level circuit capability available in PAL not available in PLA. PLA requires external connections to do multi-level circuits.

### Programmable Logic Array Example

$$F_1 = AB' + AC + A'BC'$$

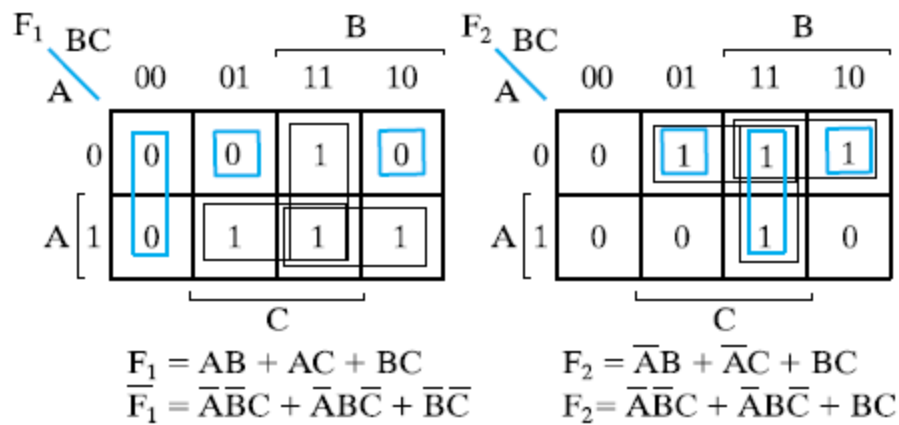
$$F_2 = (AC + BC)'$$



### Example: Implementing a Combinational Circuit Using a PLA

$$F_1(A,B,C) = \sum m(3,5,6,7)$$

$$F_2(A,B,C) = \sum m(1,2,3,7)$$



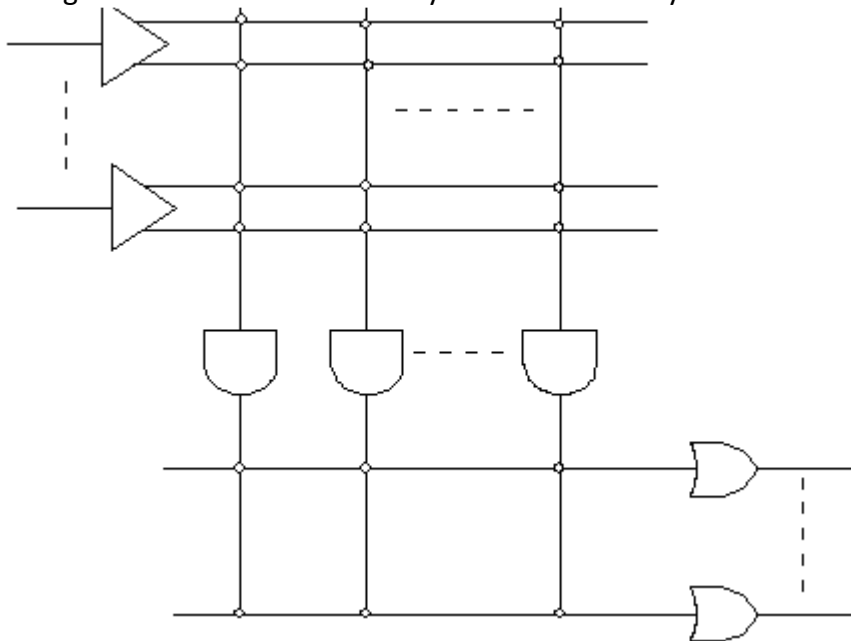
The solution is:

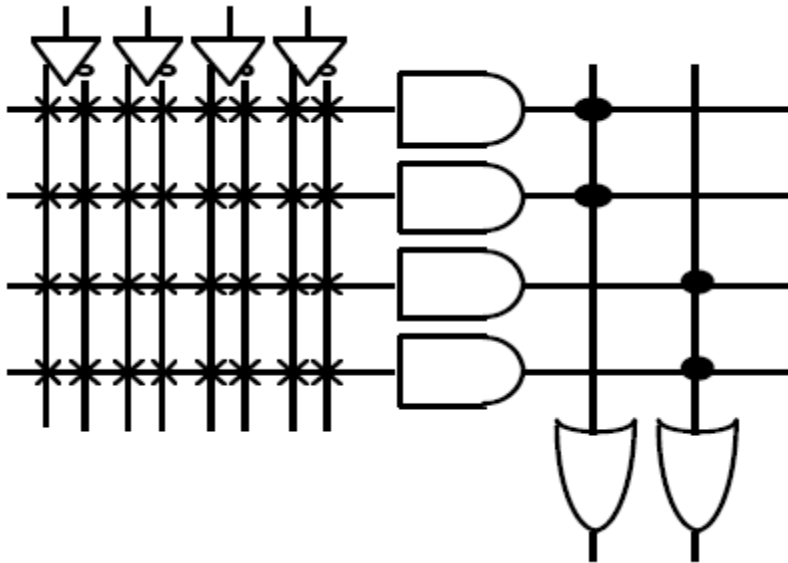
$$F_1 = \overline{\overline{A}\overline{B}C + \overline{A}B\overline{C} + \overline{B}\overline{C}}$$

$$F_2 = \overline{A}\overline{B}C + \overline{A}B\overline{C} + BC$$

#### PROGRAMMABLE ARRAY LOGIC (PAL)

The PAL is the opposite of the ROM, having a programmable set of ANDs combined with fixed ORs. A given column of the OR array has access to only a subset of the possible product terms





#### \_ Disadvantage

- ROM guaranteed to implement any M functions of N inputs. PAL may have too few inputs to the OR gates.

#### \_ Advantages

- For given internal complexity, a PAL can have larger N and M
- Some PALs have outputs that can be complemented, adding POS functions
- No multilevel circuit implementations in ROM (without external connections from output to input). PAL has outputs from OR terms as internal inputs to all AND terms, making implementation of multi-level circuits easier.

#### Programmable Array Logic Example

##### \_ 4-input, 3-output PAL with fixed, 3-input OR terms

\_ What are the equations for F1 through F4?  $W(A,B,C,D) = \sum m(2,12,13)$

$$X(A,B,C,D) = \sum m(7,8,9,10,11,12,13,14,15)$$

$$Y(A,B,C,D) = \sum m(0,2,3,4,5,6,7,8,10,11,15)$$

$$Z(A,B,C,D) = \sum m(1,2,8,12,13)$$

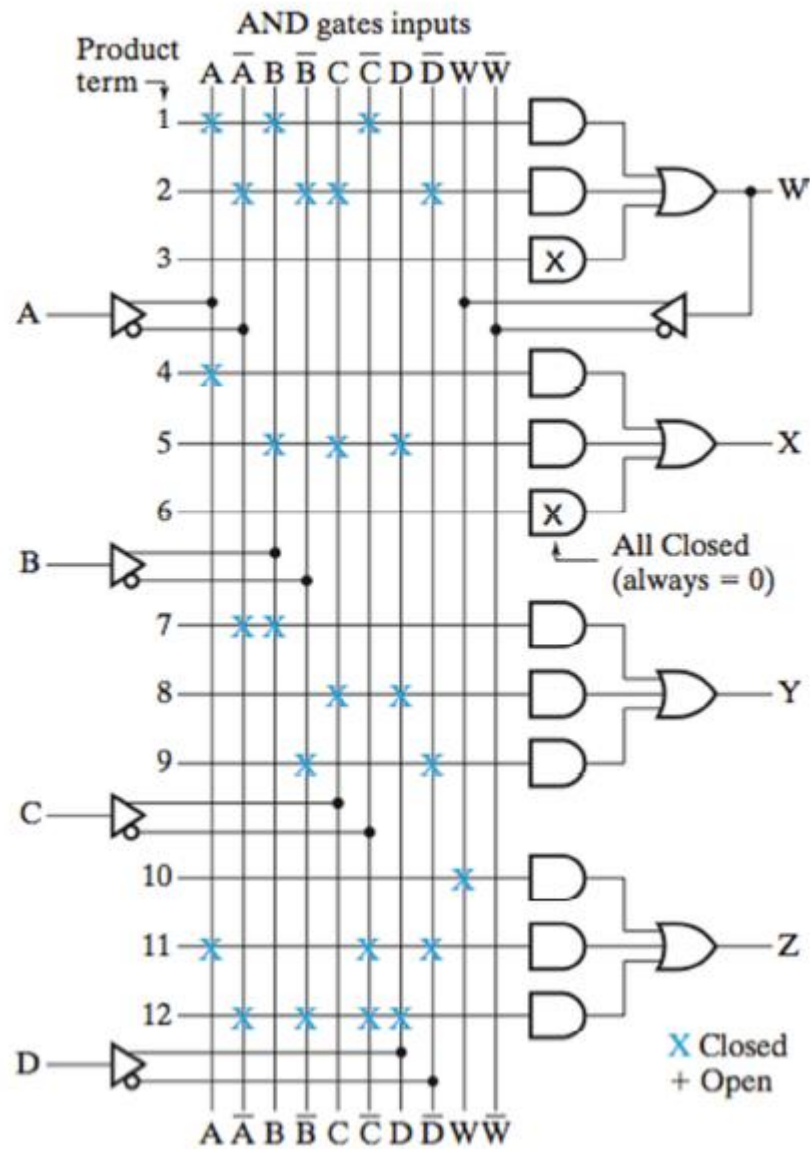
Simplifying the four function to a minimum number of terms results in the following Boolean functions

$$W = ABC' + A'B'CD'$$

$$X = A + BCD$$

$$Y = A'B + CD + B'D'$$

$$Z = ABC' + A'B'CD' + AC'D' + A'B'C'D = W + AC'D' + A'B'C'D$$





**PAL** Design Example: BCD to Gray Code Converter**TruthTable**

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

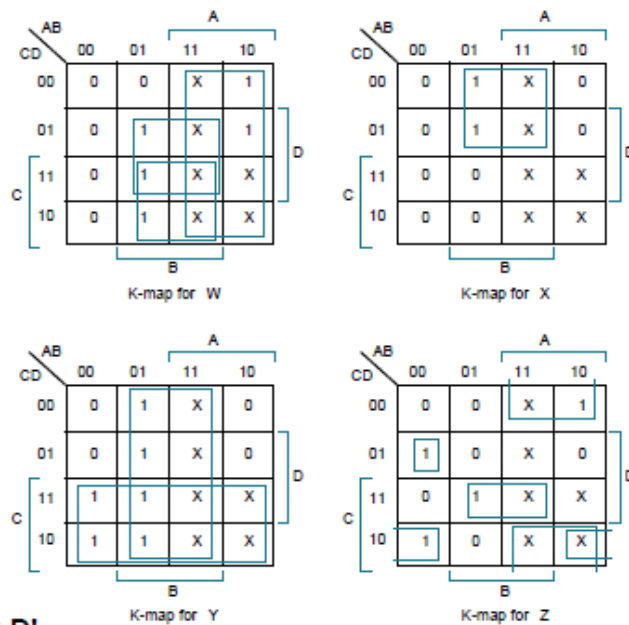
**Minimized  
Functions:**

$$W = A + B D + B C$$

$$X = B C'$$

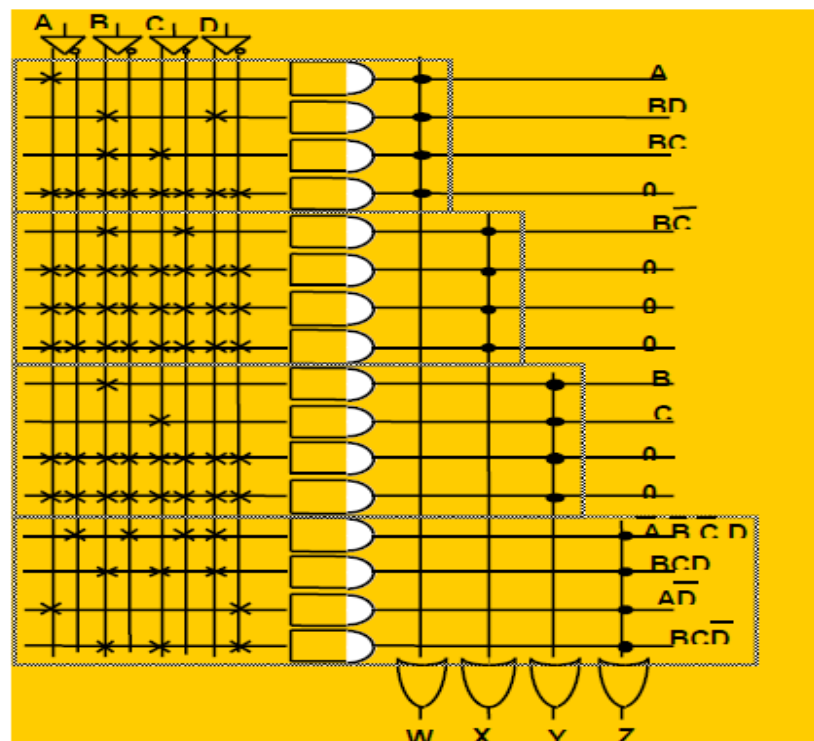
$$Y = B + C$$

$$Z = A'B'C'D + B C D + A D' + B' C D'$$

**K-maps**

**Programmed PAL:**

**4 product terms  
per each OR gate**



## PLA- OPTIMIZATION

- The major disadvantage of the PLA is that most practical logic problems leave much PLA area unused.
- A straight forward physical design results into a significant waste of silicon area, which is undesirable.
- Also, speed and power become critical parameters as the size of the PLA increases
- The gate capacitances of the input signals carried by long poly-silicon lines become the key factor in determining the timing (speed)performance.
- In moderate to large PLA's, the poly-silicon resistance becomes as important factor as the capacitance.
- The signal is degraded with the large resistance added to the line, no matter how large the drivers are. Further, if the PLA becomes large, the width of the power and the ground lines should also be increased to avoid possible metal migration.
- Due to the regular structure the PLA takes more area (space) when implemented in a VLSI chip than the Gate or Standard cell circuits.
- To accommodate all components and modules with in a small area (for area efficient design) two important operations are used.
- One is the Minimization and the other is the Folding of PLAs
- This minimization techniques are based on Boolean minimization algorithms , which remove redundant product terms and inturn decreases the PLA area.
- In the minimization process the functions being implemented are reduced without changing the input/output relationship, so that the original PLA can be transformed into a smaller one.

## Boolean Minimization Techniques

- One of the Boolean minimization techniques used is to remove redundant product terms in the personality matrix of a PLA and there by removing the rows that implement these terms.
- Another Boolean minimization technique is to determine if a term  $xf$  can be replaced by a term  $f$ , where  $f$  is the product of some variables other than  $x$  and covers more min-terms than does  $xf$ .
- This technique is called raising of terms.
- With the advancements in VLSI ,the problem of minimization has become more cumbersome, due to increase in the number of variables.
- Hence most of the classical methods have become almost outdated.
- The first cube based algorithm MINI was developed at IBM by Hong et al in 1974.
- Most extensively used PLA minimizer currently available is ESPRESSO II ,developed at the university of California-Berkeley by Brayton et al.
- The McBoole logic minimizer developed at McGill university by Dagenais et al is based on the Quine-McCluskey philosophy and generates all the prime cubes.
- Another better multiple-output minimization algorithm than ESPRESSO II and McBoole in many PLAs has been developed at IISc, Bangalore,India by Gurunath and Biswas in 1989.
- This algorithm is based on switching theoretic concepts and is a fast technique for the determination of essential prime cubes only.

## PLA Folding

- Folding is a technology-independent, topological minimization technique, developed for array structures, that attempts to place two or more input/output (product term) signals together so that they can share the same physical column (row).
- It does not change the implementation of the logic in any manner, but reduces the number of columns and rows, and consequently reduces the area of the PLA.

### Simple folding

- When a pair of inputs or outputs share the same column or row, respectively. It is assumed that the input lines and the output lines are either on the upper or lower sides of the columns thus, there are no intersections between folded lines.
- Most often, the input and output lines are folded in the AND and OR matrix, respectively, due to electrical and physical constraints.

### Multiple folding

- It is a more general technique where the input and output lines are folded as much as possible to minimize the number of columns, respectively rows, in AND and OR matrices.
- This method reduces the area. However, routing of the input and output lines is more complicated, and another metal or poly-silicon layer may be required.
- Therefore, multiple folding is efficient when the PLA is a component of a large system where several metal or poly-silicon layers are already required.

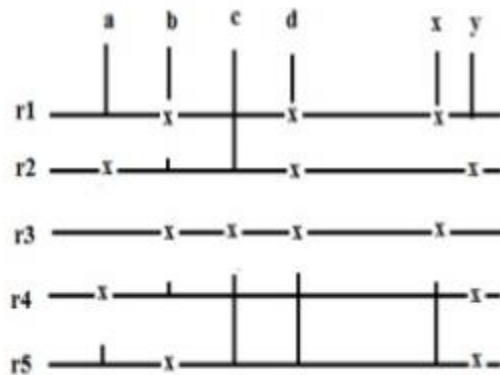


- **Bipartite folding** is a special example of simple folding where column breaks between two parts in the same column must occur at the same horizontal level in either the AND or OR-matrix.
  - **Constrained folding** is a restricted folding where some constraints such as the order and place of lines are given and accommodated with other folding
- 
- It has been shown that PLA folding problems are NP-complete and the number of possible solutions approximates  $c!$  or  $r!$ , where  $c$  and  $r$  are the number of columns and rows in the initial PLA, respectively.
  - As the number of inputs and outputs is very large in recent modern PLAs, it's not possible to implement these algorithms manually.
  - Hence the procedure of folding is automatized, and many computer based (CAD tools) algorithms have been proposed

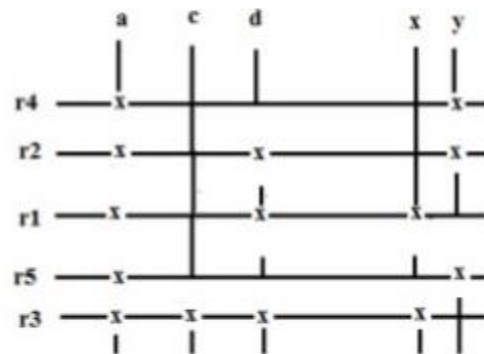
## Column Folding

- Column folding is the sharing of a single physical column by two or more columns (input/output signals) of a PLA.
- Column folding is said to be simple if utmost two signals (logical columns) share a single physical column.
- It is called multiple if more than two columns can share a single column.
- Column folding can be obtained by permuting the rows of the PLA as shown in the next slide.

## Column folding



(a)Original PLA



(b).Row Permuted PLA

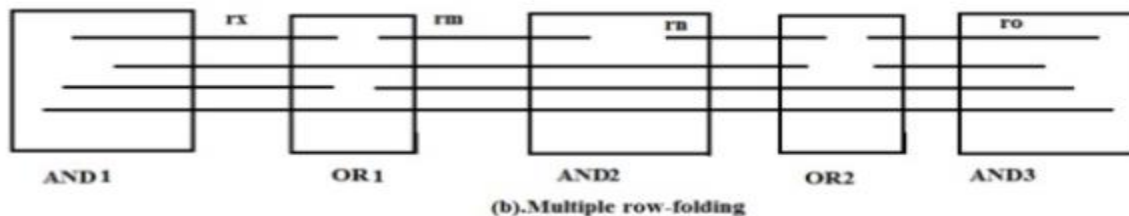
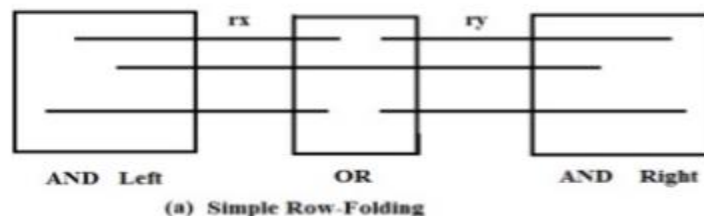
- Column folding can be obtained by permuting the rows of the PLA. For example, in slide (a), no column folding is evident.
- However, from the given representation of the PLA, if the rows are permuted so that r4, r2, r1, r5, and r3 are in order, then it is possible to fold and place the column a above the column b. This row-permuted PLA is shown in slide (b).
- From the previous slide it is clear that column folding introduces a restriction on the order of the rows.
- More specifically, if column x is folded and placed above column y, then all the rows that receive the signal x must be placed above those rows that use the signal y.
- For example, in slide (b), to fold column a above b, rows r4 and r2 should be placed above r1, r5 and r3.

- The restrictions imposed on the order of rows by one set of folded column signals might conflict with the row ordering desired by another set of folded columns.
- Such conflicts on row orderings along with some other conditions make column folding an NP-complete optimization problem.
- An ordering of rows is termed as optimal. if it needs to the maximal folding of columns

### Row Folding

- Row folding is defined as the sharing of a single physical row by two or more product terms (logical rows) of a PLA.
- Row folding is called simple if only two rows share a single row and is called multiple if more than two rows share a single row.
- Row folding is more complex than column folding, due to the fact that rows interact with both the input and output columns.

### Row-Folding-Example



- In order that two rows can be folded together, it is necessary that the input columns feeding one row be separated from those that feed the other by the output columns. Such separation segments the PLA.
- There are two ways to segment PLAs for simple row folding.
- One, that splits the original AND-plane resulting in the AND-OR AND structure.
- The other that yields the OR-AND-OR structure.
- The PLA in the AND-OR-AND form, consists of two AND-planes (the left and the right) and an OR-plane between these two planes.
- The OR-AND-OR PLA consists of two OR-planes (the left and the right) on either side of an AND-plane.
- In both cases, a row belonging solely to the left (AND- or OR-) plane can be folded only with a row belonging solely to the right (AND or OR-) plane.
- A multiple row-folded PLA consists of a sequence of alternating AND- and OR-planes.
- Row folding can be achieved by segmenting the PLA planes and then permuting the columns appropriately.

## Adder Design

A ripple-carry adder works in the same way as pencil-and-paper methods of addition. Starting at the rightmost (least significant) digit position, the two corresponding digits are added and a result obtained. It is also possible that there may be a carry out of this digit position (for example, in pencil-and-paper methods, "9+5=4, carry 1"). Accordingly, all digit positions other than the rightmost need to take into account the possibility of having to add an extra 1, from a carry that has come in from the next position to the right.

This means that no digit position can have an absolutely final value until it has been established whether or not a carry is coming in from the right. Moreover, if the sum without a carry is 9 (in pencil-and-paper methods) or 1 (in binary arithmetic), it is not even possible to tell whether or not a given digit position is going to pass on a carry to the position on its left. At worst, when a whole sequence of sums comes to ...99999999... (in decimal) or ...11111111... (in binary), nothing can be



deduced at all until the value of the carry coming in from the right is known, and that carry is then propagated to the left, one step at a time, as each digit position evaluated "9+1=0, carry 1" or "1+1=0, carry 1". It is the "rippling" of the carry from right to left that gives a ripple-carry adder its name, and its slowness. When adding 32-bit integers, for instance, allowance has to be made for the possibility that a carry could have to ripple through every one of the 32 one-bit adders.

Carry lookahead depends on two things:

1. Calculating, for each digit position, whether that position is going to propagate a carry if one comes in from the right.
2. Combining these calculated values to be able to deduce quickly whether, for each group of digits, that group is going to propagate a carry that comes in from the right.

Supposing that groups of four digits are chosen. Then the sequence of events goes something like this:

1. All 1-bit adders calculate their results. Simultaneously, the lookahead units perform their calculations.
2. Suppose that a carry arises in a particular group. Within at most five gate delays, that carry will emerge at the left-hand end of the group and start propagating through the group to its left.
3. If that carry is going to propagate all the way through the next group, the lookahead unit will already have deduced this. Accordingly, *before the carry emerges from the next group*, the lookahead unit is immediately (within one gate delay) able to tell the *next* group to the left that it is going to receive a carry – and, at the same time, to tell the next lookahead unit to the left that a carry is on its way.

The net effect is that the carries start by propagating slowly through each 4-bit group, just as in a ripple-carry system, but then move four times as fast, leaping from one lookahead carry unit to the next. Finally, within each group that receives a carry, the carry propagates slowly within the digits in that group.

The more bits in a group, the more complex the lookahead carry logic becomes, and the more time is spent on the "slow roads" in each group rather than on the "fast road" between the groups (provided by the lookahead carry logic). On the other hand, the fewer bits there are in a group, the more groups have to be traversed to get from one end of a number to the other, and the less acceleration is obtained as a result.

Deciding the group size to be governed by lookahead carry logic requires a detailed analysis of gate and propagation delays for the particular technology being used.

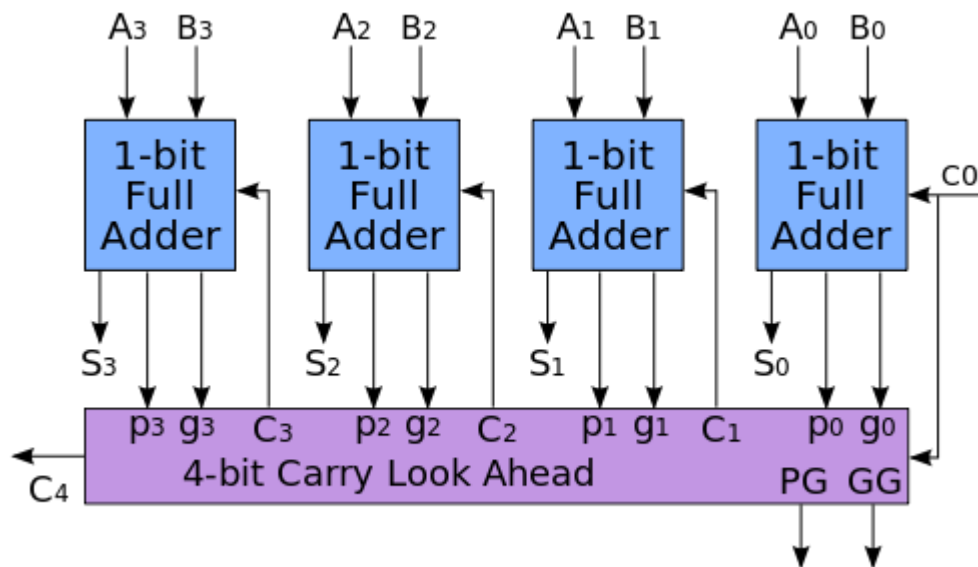
It is possible to have more than one level of lookahead carry logic, and this is in fact usually done. Each lookahead carry unit already produces a signal saying "if a carry comes in from the right, I will propagate it to the left", and those signals can be combined so that each group of (let us say) four lookahead carry units becomes part of a "supergroup" governing a total of 16 bits of the numbers being added. The "supergroup" lookahead carry logic will be able to say whether a carry entering the supergroup will be propagated all the way through it, and using this information, it is able to propagate carries from right to left 16 times as fast as a naive ripple carry. With this kind of two-level implementation, a carry may first propagate through the "slow road" of individual adders, then, on reaching the left-hand end of its group, propagate through the "fast road" of 4-bit lookahead carry logic, then, on reaching the left-hand end of its supergroup, propagate through the "superfast road" of 16-bit lookahead carry logic.

Again, the group sizes to be chosen depend on the exact details of how fast signals propagate within logic gates and from one logic gate to another.

For very large numbers (hundreds or even thousands of bits), lookahead carry logic does not become any more complex, because more layers of supergroups and supersupergroups can be added as necessary. The increase in the number of gates is also moderate: if all the group sizes are four, one would end up with one third as many lookahead carry units as there are adders. However, the "slow roads" on the way to the faster levels begin to impose a drag on the whole system (for instance, a 256-bit adder could have up to 24 gate delays in its carry processing), and the mere physical transmission of signals from one end of a long number to the other begins to be a problem. At these sizes, carry-save adders are preferable, since they spend no time on carry propagation at all.

A **carry-lookahead adder (CLA)** or **fast adder** is a type of adder used in digital logic. A carry-lookahead adder improves speed by reducing the amount of time required to determine carry bits. It can be contrasted with the simpler, but usually slower, ripple carry adder for which the carry bit is calculated alongside the sum bit, and each bit must wait until the previous carry has been calculated to begin calculating its own result and carry bits (see adder for detail on ripple carry adders). The carry-lookahead adder calculates one or more carry bits before the sum, which

reduces the wait time to calculate the result of the larger value bits. The Kogge-Stone adder and Brent-Kung adder are examples of this type of adder.



Carry lookahead logic uses the concepts of *generating* and *propagating* carries. Although in the context of a carry lookahead adder, it is most natural to think of generating and propagating in the context of binary addition, the concepts can be used more generally than this. In the descriptions below, the word *digit* can be replaced by *bit* when referring to binary addition of 2.

The addition of two 1-digit inputs  $A$  and  $B$  is said to *generate* if the addition will always carry, regardless of whether there is an input carry (equivalently, regardless of whether any less significant digits in the sum carry). For example, in the decimal addition  $52 + 67$ , the addition of the tens digits 5 and 6 *generates* because the result carries to the hundreds digit regardless of whether the ones digit carries (in the example, the ones digit does not carry ( $2+7=9$ )).

In the case of binary addition,  $A \cdot B$  generates if and only if both  $A$  and  $B$  are 1. If we write  $G(A, B)$  to represent the binary predicate that is true if and only if  $A \cdot B$  generates, we have:

$$G(A, B) = A \cdot B$$

The addition of two 1-digit inputs  $A$  and  $B$  is said to *propagate* if the addition will carry whenever there is an input carry (equivalently, when the next less significant digit in the sum carries). For example, in the decimal addition  $37 + 62$ , the addition of the tens digits 3 and 6 *propagate* because the result would carry to the hundreds digit *if* the ones were to carry (which in this example, it does not). Note that propagate and generate are defined with respect to a single digit of addition and do not depend on any other digits in the sum.

In the case of binary addition,  $A + B$  propagates if and only if at least one of  $A$  or  $B$  is 1. If we write  $P(A, B)$  to represent the binary predicate that is true if and only if  $A + B$  propagates, we have:

$$P(A, B) = A + B$$

Sometimes a slightly different definition of *propagate* is used. By this definition  $A + B$  is said to propagate if the addition will carry whenever there is an input carry, but will not carry if there is no input carry. Fortunately, due to the way generate and propagate bits are used by the carry lookahead logic, it doesn't matter which definition is used. In the case of binary addition, this definition is expressed by:

$$P'(A, B) = A \oplus B$$

For binary arithmetic, or is faster than xor and takes fewer transistors to implement. However, for a multiple-level carry lookahead adder, it is simpler to use  $P'(A, B)$ .

Given these concepts of generate and propagate, a digit of addition carries precisely when either the addition generates or the next less significant bit carries and the addition propagates. Written in boolean algebra, with  $C_i$  the carry bit of digit  $i$ , and  $P_i$  and  $G_i$  the propagate and generate bits of digit  $i$  respectively,

$$C_{i+1} = G_i + (P_i \cdot C_i)$$

## Implementation details [\[ edit \]](#)

For each bit in a binary sequence to be added, the Carry Look Ahead Logic will determine whether that bit pair will generate a carry or propagate a carry. This allows the circuit to "pre-process" the two numbers being added to determine the carry ahead of time. Then, when the actual addition is performed, there is no delay from waiting for the ripple carry effect (or time it takes for the carry from the first Full Adder to be passed down to the last Full Adder). Below is a simple 4-bit generalized Carry Look Ahead circuit that combines with the 4-bit Ripple Carry Adder we used above with some slight adjustments:

For the example provided, the logic for the generate (g) and propagate (p) values are given below. Note that the numeric value determines the signal from the circuit above, starting from 0 on the far left to 3 on the far right:

$$C_1 = G_0 + P_0 \cdot C_0$$

$$C_2 = G_1 + P_1 \cdot C_1$$

$$C_3 = G_2 + P_2 \cdot C_2$$

$$C_4 = G_3 + P_3 \cdot C_3$$

Substituting  $C_1$  into  $C_2$ , then  $C_2$  into  $C_3$ , then  $C_3$  into  $C_4$  yields the expanded equations:

$$C_1 = G_0 + P_0 \cdot C_0$$

$$C_2 = G_1 + P_1 \cdot C_1$$

$$C_3 = G_2 + P_2 \cdot C_2$$

$$C_4 = G_3 + P_3 \cdot C_3$$

Substituting  $C_1$  into  $C_2$ , then  $C_2$  into  $C_3$ , then  $C_3$  into  $C_4$  yields the expanded equations:

$$C_1 = G_0 + P_0 \cdot C_0$$

$$C_2 = G_1 + G_0 \cdot P_1 + C_0 \cdot P_0 \cdot P_1$$

$$C_3 = G_2 + G_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + C_0 \cdot P_0 \cdot P_1 \cdot P_2$$

$$C_4 = G_3 + G_2 \cdot P_3 + G_1 \cdot P_2 \cdot P_3 + G_0 \cdot P_1 \cdot P_2 \cdot P_3 + C_0 \cdot P_0 \cdot P_1 \cdot P_2 \cdot P_3$$

To determine whether a bit pair will generate a carry, the following logic works:

$$G_i = A_i \cdot B_i$$

To determine whether a bit pair will propagate a carry, either of the following logic statements work:

$$P_i = A_i \oplus B_i$$

$$P_i = A_i + B_i$$

The reason why this works is based on evaluation of  $C_1 = G_0 + P_0 \cdot C_0$ . The only difference in the truth tables between  $(A \oplus B)$  and  $(A + B)$  is when both  $A$  and  $B$  are 1. However, if both  $A$  and  $B$  are 1, then the  $G_0$  term is 1 (since its equation is  $A \cdot B$ ), and the  $P_0 \cdot C_0$  term becomes irrelevant. The XOR is used normally within a basic full adder circuit; the OR is an alternate option (for a carry lookahead only) which is far simpler in transistor-count terms.

The Carry Look Ahead 4-bit adder can also be used in a higher-level circuit by having each CLA Logic circuit produce a propagate and generate signal to a higher-level CLA Logic circuit. The group propagate ( $PG$ ) and group generate ( $GG$ ) for a 4-bit CLA are:

$$PG = P_3 \cdot P_2 \cdot P_1 \cdot P_0$$

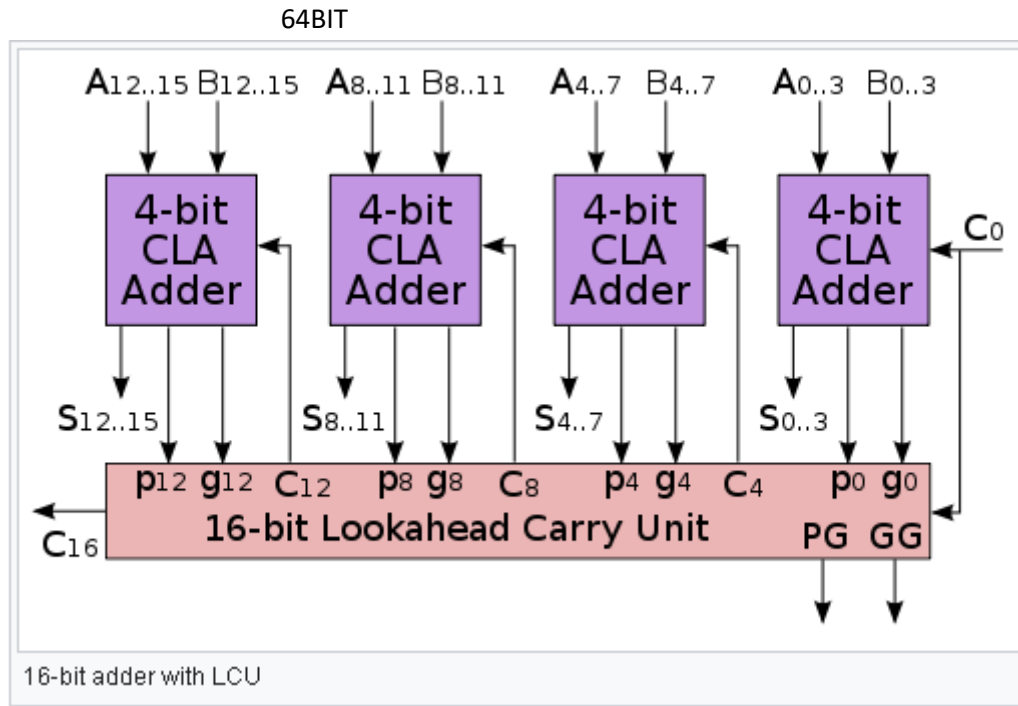
$$GG = G_3 + G_2 \cdot P_3 + G_1 \cdot P_3 \cdot P_2 + G_0 \cdot P_3 \cdot P_2 \cdot P_1$$

Putting 4 4-bit CLAs together yields four group propagates and four group generates. A [Lookahead Carry Unit](#) (LCU) takes these 8 values and uses identical logic to calculate  $C_i$  in the CLAs. The LCU then generates the carry input for each of the 4 CLAs and a fifth equal to  $C_{16}$ .

The calculation of the [gate delay](#) of a 16-bit adder (using 4 CLAs and 1 LCU) is not as straight forward as the ripple carry adder. Starting at time of zero:

- calculation of  $P_i$  and  $G_i$  is done at time 1
- calculation of  $C_i$  is done at time 3
- calculation of the  $PG$  is done at time 2
- calculation of the  $GG$  is done at time 3
- calculation of the inputs for the CLAs from the LCU are done at
  - time 0 for the first CLA
  - time 5 for the second, third & fourth CLA
- calculation of the  $S_i$  are done at
  - time 4 for the first CLA
  - time 8 for the second, third & fourth CLA
- calculation of the final carry bit ( $C_{16}$ ) is done at time 5

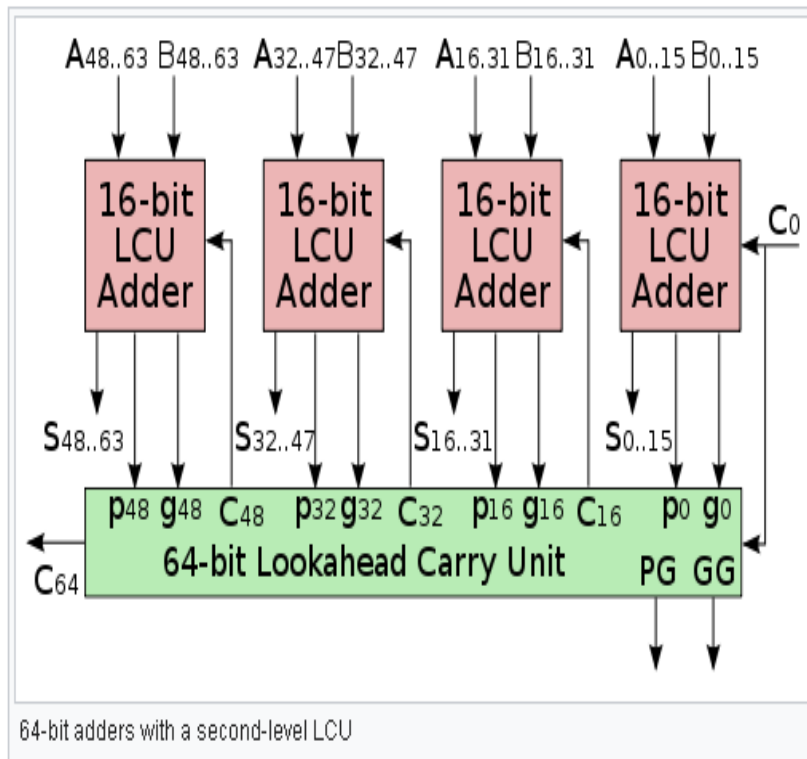
The maximum time is 8 gate delays (for  $S_{[8-15]}$ ). A standard 16-bit [ripple carry adder](#) would take  $16*3-1=47$  gate delays.



ADER



By combining 4 CLAs and an LCU together creates a 16-bit adder. Four of these units can be combined to form a 64-bit adder. An additional (second-level) LCU is needed that accepts the propagate ( $P_{LCU}$ ) and generate ( $G_{LCU}$ ) from each LCU and the four carry outputs generated by the second-level LCU are fed into the first-level LCUs.



## Unit-IV

### Faults in Digital Circuits

## Some Real Defects in Chips

- Processing Faults
  - missing contact windows
  - parasitic transistors
  - oxide breakdown
- Material Defects
  - bulk defects (cracks, crystal imperfections)
  - surface impurities (ion migration)
- Time-Dependent Failures
  - dielectric breakdown
  - electro migration
- Packaging Failures
  - contact degradation
  - seal leaks

## Faults, Errors and Failures

- Fault: A physical defect within a circuit or a system
  - May or may not cause a system failure
- Error: Manifestation of a fault that results in incorrect circuit (system) outputs or states
  - Caused by faults
- Failure: Deviation of a circuit or system from its specified behavior
  - Fails to do what it should do
  - Caused by an error
- Fault ---> Error ---> Failure

## Modeling of Faults

- Fault model identifies target faults
  - Model faults most likely to occur
- Fault model limits the scope of test generation
  - Create tests only for the modeled faults
- Fault model makes effectiveness measurable by experiments
  - Fault coverage can be computed for specific test patterns to reflect its effectiveness
- Fault model makes analysis possible
  - Associate specific defects with specific test patterns

## Fault models

In general the effect of a fault is represented by means of a model, which represents the change the fault produces in circuit signals. The fault models in use today are:

1. Stuck-at fault
2. Bridging fault
3. Stuck-open fault

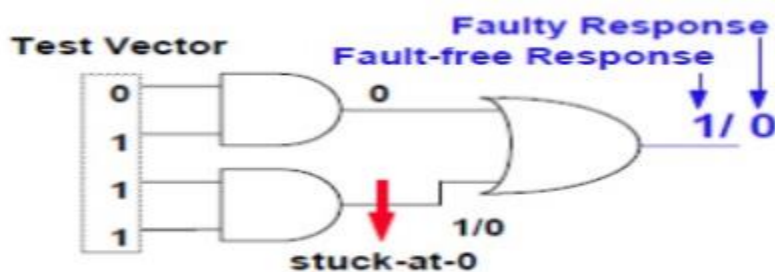
## Stuck-at Faults

The most common model used for logical faults is the “single stuck-at fault”. It assumes that a fault in a logic gate results in one of its inputs or the output being fixed to either a logic 0 (stuck-at-0) or a logic 1 (stuck-at-1). Stuck-at-0 and stuck-at-1 faults are often abbreviated to s-a-0 and s-a-1 respectively, and the abbreviations will be adopted here.

# Single Stuck-At Faults

Assumptions:

- Only one line is faulty.
- Faulty line permanently set to 0 or 1.
- Fault can be at an input or output of a gate.

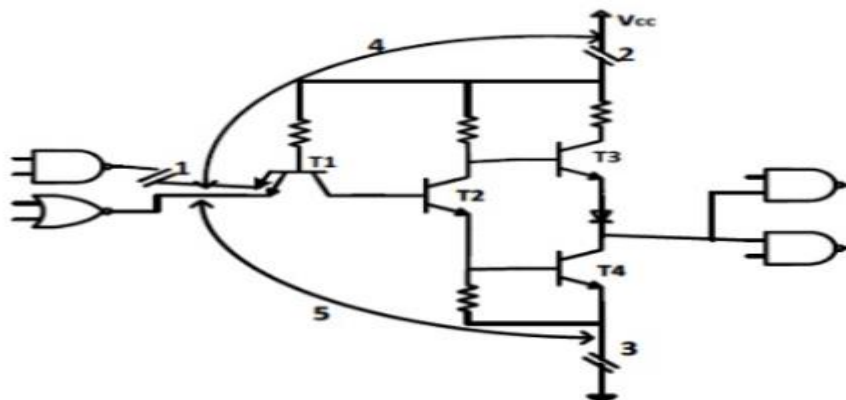


## Why Single Stuck-At Fault Model?

- Complexity is greatly reduced.  
Many different physical defects may be modeled by the same logical single stuck-at fault.
- Single stuck-at fault is technology independent.  
Can be applied to TTL, ECL, CMOS, etc.
- Single stuck-at fault is design style independent.  
Gate Arrays, Standard Cell, Custom VLSI
- Even when single stuck-at fault does not accurately model some physical defects, the tests derived for logic faults are still valid for most defects.
- Single stuck-at tests cover a large percentage of multiple stuck-at faults.

The stuck-at fault model, often referred to as the “classical” fault model, offers good representation for the most common types of failures, e.g. short-circuits (“shorts”) and open-circuits (“opens”) in many technologies. The Figure below illustrates the transistor-transistor (TTL) realization of a NAND gate, the numbers 1, 2, 3 indicating places where opens may principally occur, while 4 and 5 indicate the basic types of shorts.

## Schematic diagram of a NAND gate

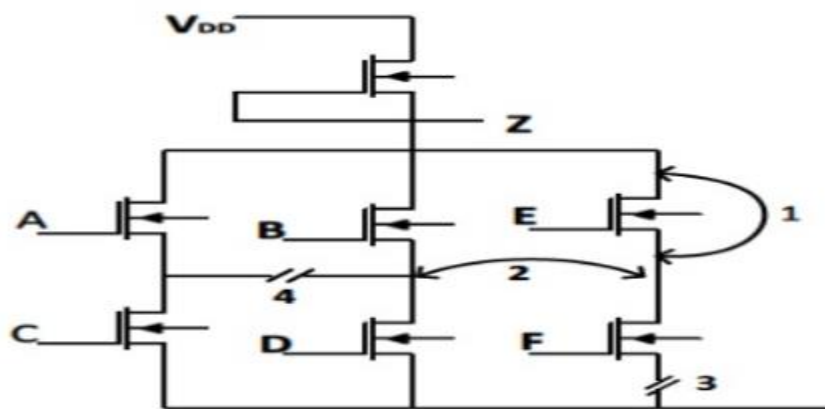


- **1. Signal line open (fault 1):** This fault prevents the sink current  $I_s$  from flowing through the emitter of the input transistor T1 into the output of the preceding gate. Thus, the input appears to be connected to a constant level 1, i.e. s-a-1.
- **2. Supply voltage open (fault 2):** In this case the gate is deprived of its supply voltage and thus neither the current  $I_s$ , which would switch the transistor T1 on, nor the current  $I_t$ , which may excite T3, can flow. Both output transistors are cut off and the output appears to be open. The fault can be interpreted as the gate output s-a-1.
- **3. Ground open (fault 3):** This fault prevents transistors T2 and T4 from conducting and thus the current  $I_t$  continually switches transistor T3 on. The output has the value of a normal logic 1, i.e. the fault may be interpreted as output s-a-1.
- **4. Signal line and Vcc short-circuited (fault 4):** This fault is of the s-a-1 type but the transistor T4 of the preceding gate is overloaded. Thus a secondary fault can be caused.
- **5. Signal line and ground short-circuited (fault 5):** A fault of this type may be interpreted as s-a-0.



The stuck-at model has gained wide acceptance in the past mainly because of its relative success with small scale integration. However, it is not very effective in accounting for all faults in present day VLSI/ULSI chips, which mainly use MOS technology. Faults in MOS circuits do not necessarily produce logical faults that can be described as stuck-at faults. This can be illustrated by the example in the fig. below

## A MOS network



- Two possible shorts numbered 1 and 2 and two possible opens numbered 3 and 4 are indicated in the diagram
- Short number 1 can be modeled by s-a-1 of input E
- open number 3 can be modeled by s-a-0 of input E or input F or both
- On the other hand, short number 2 and open number 4 cannot be modeled by any stuck-at-fault because they involve a modification of the network function

- For example, in the presence of short number 2 the network function will change to:  
$$Z = [(A+B+E)(C+D+F)]'$$
- and open number 4 will also cause a change in the function. Determine the new function that will result from open number 4.

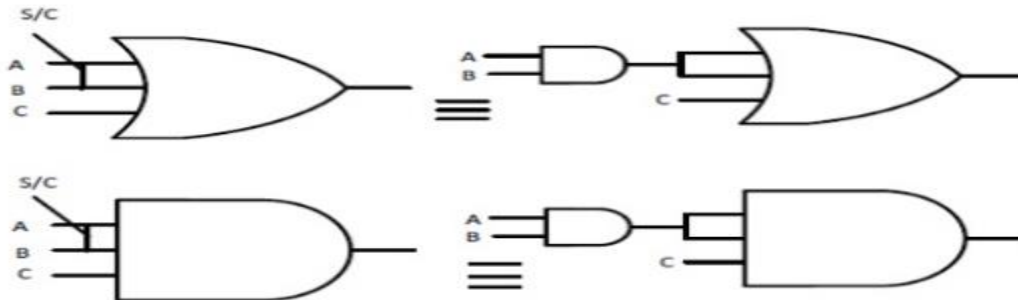
## Multiple Stuck-at Faults

- **A multiple stuck-at fault means that any set of lines is stuck-at some combination of (0,1) values.**
- **The total number of single and multiple stuck-at faults in a circuit with  $k$  single fault sites is  $3^k - 1$ .**
- **A single fault test can fail to detect the target fault if another fault is also present, however, such masking of one fault by another is rare.**
- **Statistically, single fault tests cover a very large number of multiple faults.**

## Bridging (Short-circuit) Faults

- Bridging faults form an important class of permanent faults which cannot be modeled as stuck-at-faults
- A bridging fault occurs when two leads in a logic network are connected accidentally and “wired logic” is performed at the connection
- Depending on whether positive or negative logic is being used the faults have the effect, respectively, of ANDing or ORing the signals involved as shown in the Fig. below.

## Example of bridging faults



## Comparing bridging and stuck-at faults

- With stuck-at faults, if there are  $n$  lines in the circuit, there are  $2n$  possible single stuck-at faults, and  $(3^n - 1)$  possible multiple stuck-at faults
- With bridging faults, if bridging between any  $s$  lines in a circuit are considered, the number of single bridging faults alone will be  $\binom{n}{s}$  and the number of multiple bridging faults will be very much larger

## Bridging faults inside an integrated circuit chip

Bridging faults inside an integrated circuit chip may arise if:

- the insulation between adjacent layers of metallization inside the chip breaks down
- two conductors in the same layer are shorted due to improper masking or etching



## printed circuit level bridging faults

At the printed circuit level bridging faults occur due to:

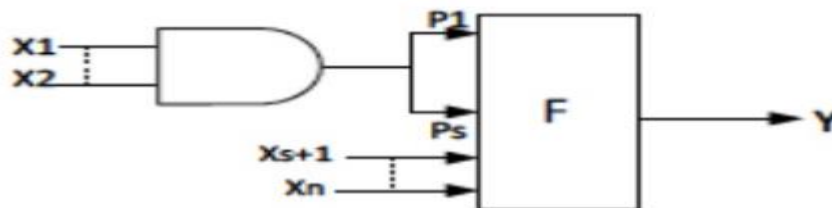
- defective printed circuit traces,
- Feed throughs,
- loose or excess bare wires,
- shorting of the pins of a chip

## Types of bridging faults

Bridging faults may be classified into two types:

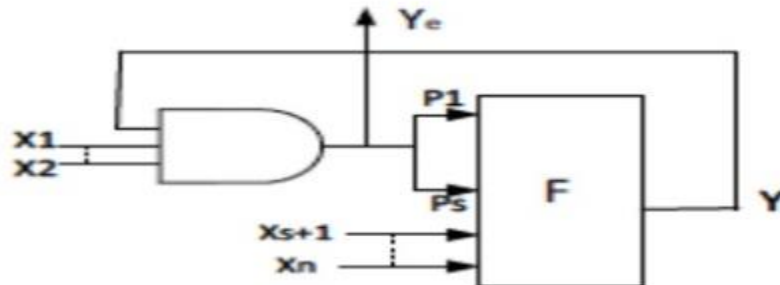
1. Input bridging
2. Feedback bridging

- Let us consider a combinational circuit implementing  $F(X_1, X_2, \dots, X_n)$ . If there is bridging among  $s$  input lines of the circuit, it has an input bridging fault of multiplicity  $s$ .
- Logical models of input bridging :



## feedback bridging

- A feedback bridging fault of multiplicity  $s$  results if there is bridging among the output of the circuit and  $s$  input lines
- Logical models of feedback bridging



- With the feedback bridging fault between the primary output  $Y_e$  and  $s$  input lines, the faulty primary output is equal to the AND function of the original output of the circuit and  $x_1, x_2, \dots, x_s$ .
- The presence of a feedback bridging fault can cause a circuit to oscillate or convert it to a sequential circuit
- Under feedback bridging ( $Yx_1, x_2, \dots, x_s$ ) any circuit  $N$  implementing  $F(x_1, x_2, \dots, x_n)$  oscillates if the input combination  $(x_1 \dots x_n)$  satisfies the following condition

$$x_1 x_2 \dots x_s F(0, 0, \dots, 0, x_{s+1}, \dots, x_n) \bar{F}(1, 1, \dots, 1, x_{s+1}, \dots, x_n) = 1$$

## Asynchronous

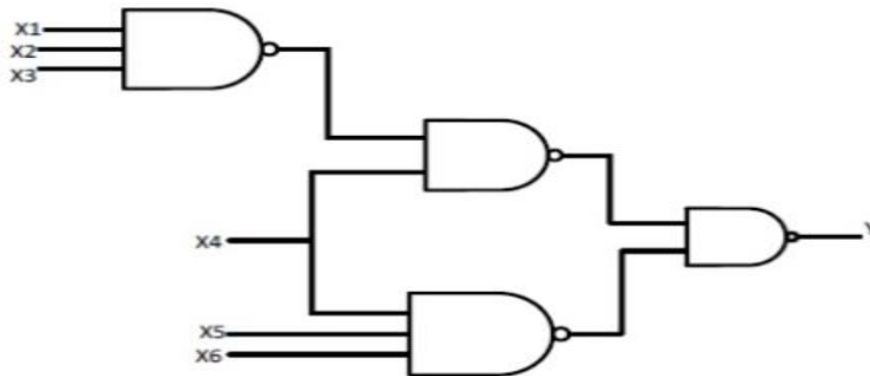
- The network will behave like an asynchronous sequential circuit if :

$$x_1 x_2 \dots x_s \bar{F}(0, 0, \dots, 0, x_{s+1}, \dots, x_n) F(1, 1, \dots, 1, x_{s+1}, \dots, x_n) = 1$$

### Example:

- if the network of Fig. below has the feedback bridging fault YX1X2, show that it will oscillate for the input combination  $(X1, X2, X3, X4, X5, X6) = (1, 1, 1, 1, 0, 0)$
- Prove the network will show asynchronous behavior if the input combination  $X4 = X5 = X6 = 1$  is applied in the presence of YX4X5X6 bridging.

### A Network with feedback bridging faults



## Solution:

- First find the expression for Y

- From the diagram,

$$Y = (X_1 X_2 X_3)' \cdot X_4 + (X_4 X_5 X_6)$$

- For oscillation,

$$x_1 x_2 \dots x_s F(0, 0, \dots, 0, x_{s+1}, \dots, x_n) \bar{F}(1, 1, \dots, 1, x_{s+1}, \dots, x_n) = 1$$

- There is bridging at YX1X2
- $X_1 \cdot X_2 = 1 \cdot 1 = 1$
- $F(0, 0, 1, 1, 0, 0) = 1$
- $F'(1, 1, 1, 1, 0, 0) = 1$
- Therefore:  $1 \cdot 1 \cdot 1 = 1$
- This shows that the circuit oscillates.

- For asynchronous behavior,

$$x_1 x_2 \dots x_s \bar{F}(0, 0, \dots, 0, x_{s+1}, \dots, x_n) F(1, 1, \dots, 1, x_{s+1}, \dots, x_n) = 1$$

- There is bridging at YX4X5X6 with input combination  $X_4 = X_5 = X_6 = 1$
- $X_4 \cdot X_5 \cdot X_6 = 1 \cdot 1 \cdot 1 = 1$
- $F'(1, 1, 1, 0, 0, 0) = 1$
- $F(1, 1, 1, 1, 1, 1) = 1$
- Therefore  $1 \cdot 1 \cdot 1 = 1$
- This shows that the circuit is asynchronous

### path sensitization method

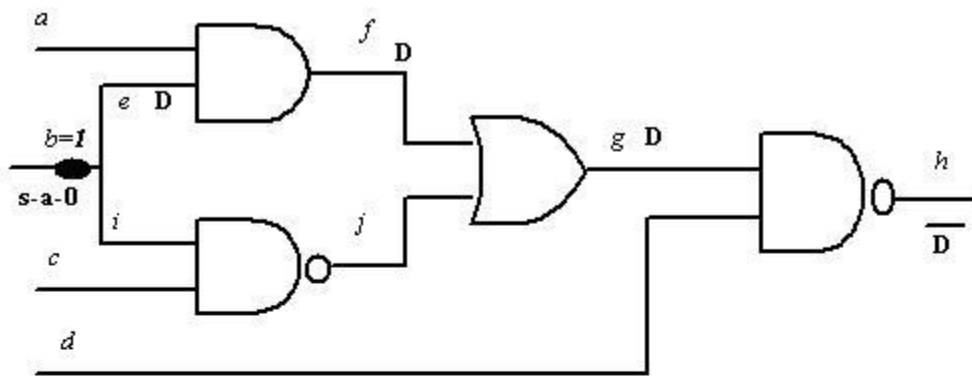
**Fault sensitization:** In this step a stuck-at fault is activated by setting the signal driving the faulty net to an opposite value from the fault value.

1. **Fault propagation:** In this step a path is selected from the fault site to some primary output, where the effect of the fault can be observed for its detection.
2. **Line justification:** In this step the signals in (internal) nets or some primary inputs, which were assigned for fault sensitization/propagation, are justified by setting (remaining) primary inputs of the circuit.

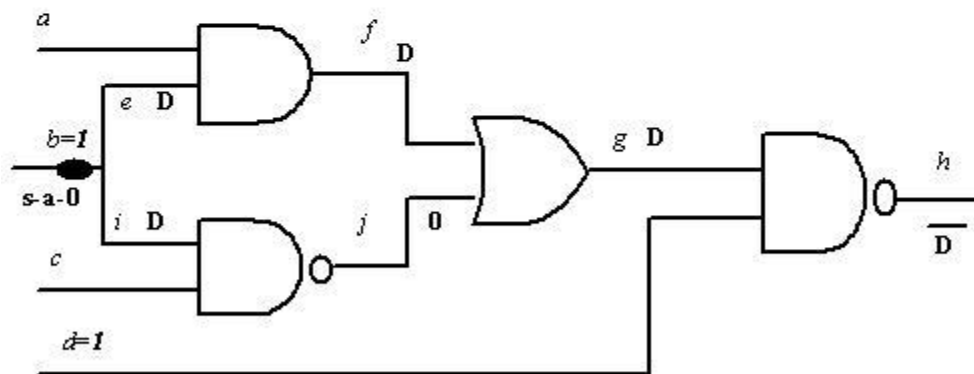
In the second and third steps, a conflict may occur, where a necessary signal assignment contradicts some previously-made assignment. When conflicts occur we need to take a new alternative path for fault propagation and see if all signals can be justified. We have seen some simple examples for ATPG using the path sensitization (sensitization-propagation-justification) approach in the last module. Now we will see a bit more complex example of ATPG using the path sensitization approach. However, instead of using Boolean algebra (as in last module), we will use Roth's

5 valued algebra. Following that, in the next lecture we will learn D-algorithm--the primary and formal algorithm for ATPG using path sensitization.

As shown in Figure 4 (a), there is a s-a-0 fault in input  $b$ . To sensitize the fault, simply  $b$  is to be made 1. Now let us take the path " $e$ - $f$ - $g$ - $h$ " for propagating the effect to the output  $h$ . The signals in the nets of the path, in terms of Roth's 5 valued algebra, are shown in Figure 4(a). It may be noted that we have successfully propagated  $\overline{D}$  to the output; it implies that the fault can be propagated to the output using the path selected. Now let us justify the signals, by setting the inputs of the gates in the path selected (for fault propagation), but not themselves being in the path, to non-controlling values. For example, net  $j$ , is a input to the OR gate that is in the path selected for fault propagation, but  $j$  is not itself in the path; so  $j$  is to be 0. Similarly,  $d$  is to be 1, and  $a$  is to be 1. However, it must be noted that  $j$  cannot be made 0; if  $c=1$  then  $j=D$  and if  $c=0$  then  $j=1$ . So we have reached a conflict at  $j$ ; Figure 4(b). Now we must backtrack and select a new path for propagation.



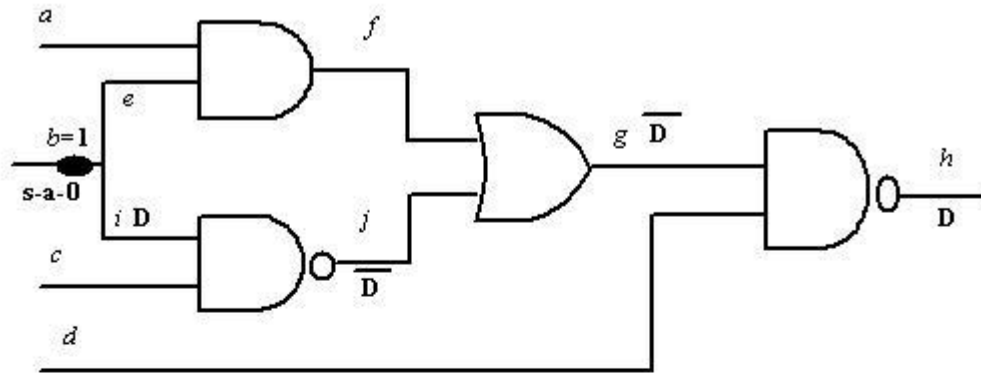
(a) Sensitization by  $b=1$   
Propagation by path  $e$ - $f$ - $g$ - $h$



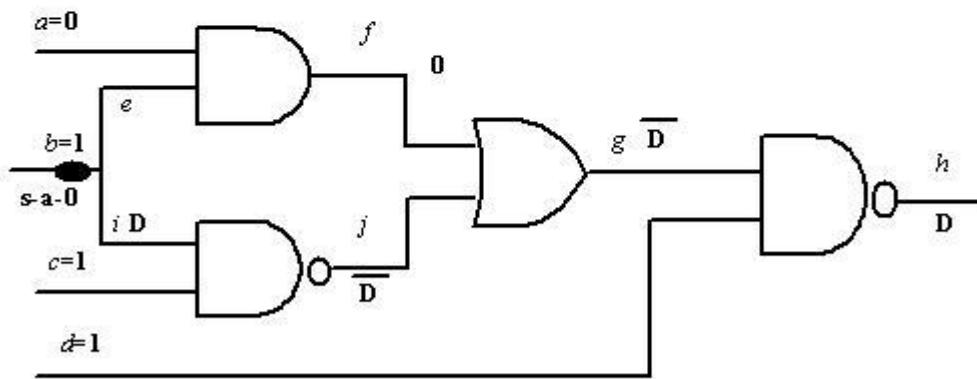
(b) Justification  $j=0$   
 $j=D$ , if  $c=1$  and  $j=1$  if  $c=0$   
(conflict at  $j$ )

Figure 4. Illustration of backtrack in path sensitization based ATPG

Figure 5 shows the ATPG if the new path is “*i-j-g-h*”. Figure 5 (a) shows the values in the nets required to propagate the fault to the output. In the new path we have again successfully propagated the fault to the output, but now a D is obtained instead of  $\overline{D}$  (as in the previous case). This implies that the fault can be propagated to the output using the new path, however, the reflection of the effect is reversed. To justify the signals, we require  $d=1, c=1, f=0, a=0$ .  $f=0$  is easily obtained by setting  $a=0$ . So we have successfully, justified the signals if the path is “*i-j-g-h*”; test pattern is  $a=0, b=1, c=1, d=1$  and effect at output is D.



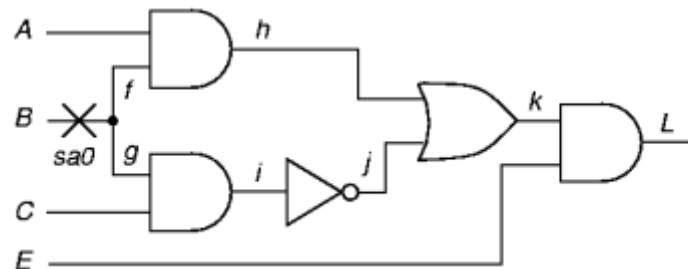
(a) Sensitization by  $b=1$   
Propagation by path *i-j-g-h*



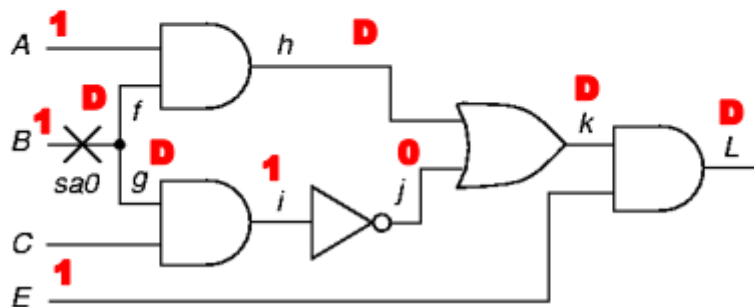
(b) Justification  $d=1, c=1, f=0, a=0$   
(Successful)

Figure 5. Successfully found test pattern in an alternative path (circuit of Figure 4)

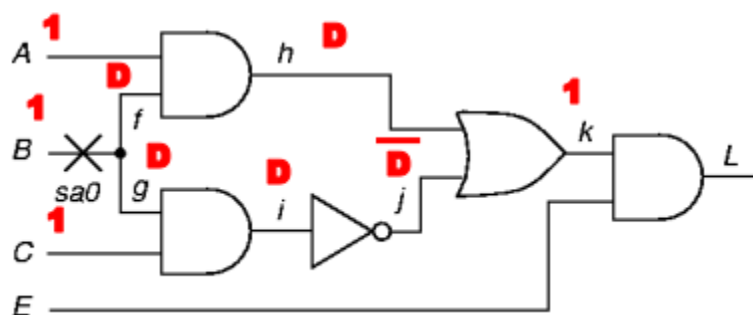
- Example (B stuck-at 0)
- Fault activation
  - Requires  $B = 1, f = D, g = D$
- Fault propagation
  - Three scenarios are possible
    - paths  $f - h - k - L, g - i - j - k - L$ , or both



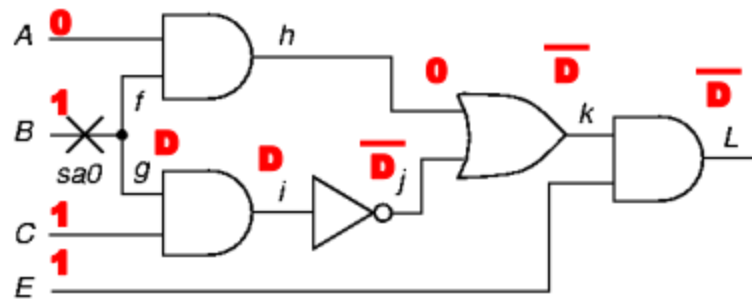
- Try path  $f - h - k - L$ 
  - Requires  $A = 1, j = 0, E = 1$
- Blocked at j
  - Since there is no way to justify 1 on i



- Try simultaneous
  - paths  $f - h - k - L$  and  $g - i - j - k - L$
- Blocked at k because
  - D-frontier (chain of D or  $\bar{D}$ ) disappears



- Final try: path  $g - i - j - k - L$
- test found!

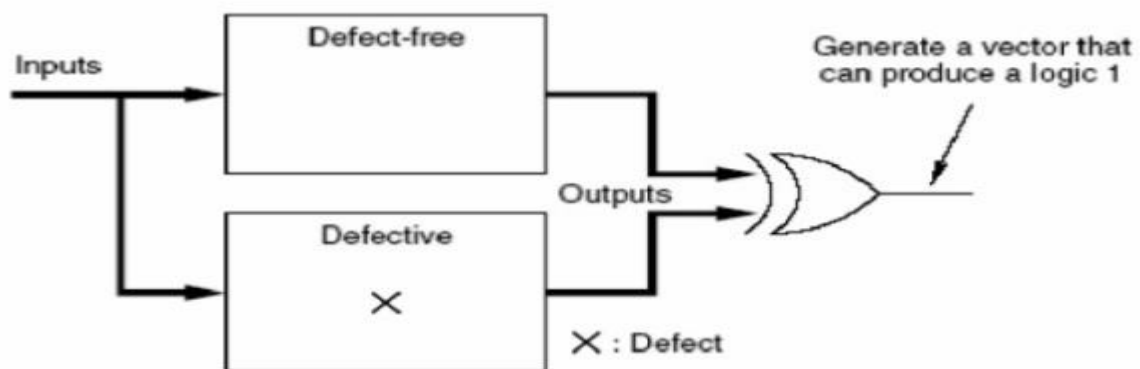


## Boolean Difference method

Boolean difference is a type of deterministic method for finding TV

BD gives all possible test vectors

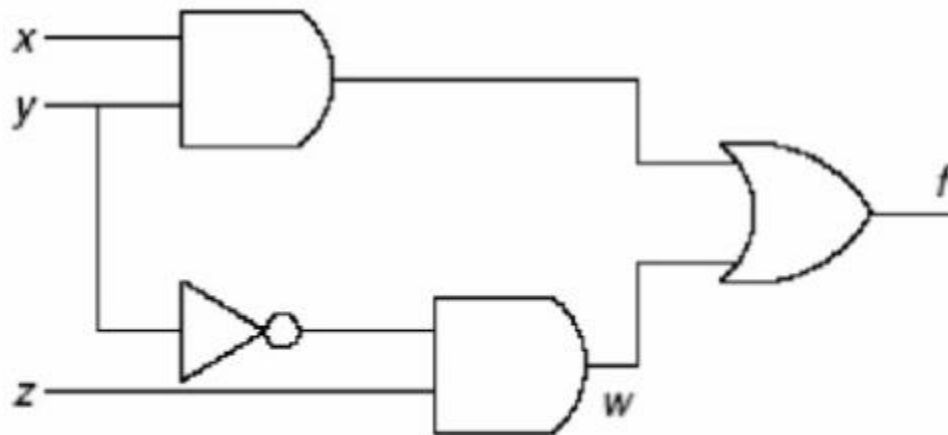
Conceptual View of ATPG



Generate an input test vector that can distinguish the defect free circuit from the hypothetically defective one



## Theoretical basis – Boolean difference



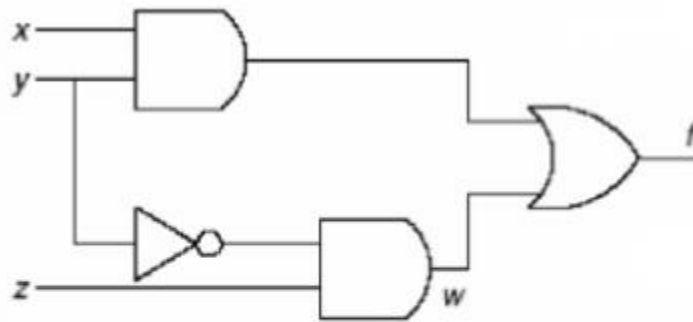
The output of the circuit is given as  $f = x \cdot y + \bar{y} \cdot z$

Let target fault be  $y$  s-a-0. Under this condition, the output of the faulty circuit is shown to be  $f_{\bar{y}} = f / y = 0$

Generate an input test vector such that  $f_y \oplus f_{\bar{y}} = 1$

- $f_y \oplus f_{\bar{y}} = 1$  if and only if  $f_y$  and  $f_{\bar{y}}$  result in opposing logic values
- Any TV that can set  $f_y \text{ XOR } f_{\bar{y}} = 1$  is able to produce opposing values at the outputs of the fault-free and faulty circuits respectively
- $\frac{df}{dy} = f_y \oplus f_{\bar{y}}$
- Now to test the fault say  $y$  at s-a-0, we need to initialize the node  $y$  to 1 (i.e.,  $y = 1$ ) and  $\frac{df}{dy} = 1$  i.e.,  $y \cdot \frac{df}{dy} = 1$
- Similarly, to test the fault say  $y$  at s-a-1 i.e.,  $\bar{y} \cdot \frac{df}{dy} = 1$

Find TV to test fault **s-a-0** at node **y** using Boolean difference method



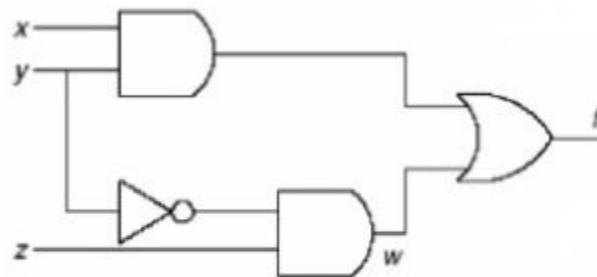
$$f = x \cdot y + \bar{y} \cdot z ; f_y = x ; f_{\bar{y}} = z$$

$$y \cdot \frac{df}{dy} = y \cdot (f_y \oplus f_{\bar{y}}) = y \cdot (x \oplus z) = \bar{x} \cdot y \cdot z + x \cdot y \cdot \bar{z}$$

$$y \cdot \frac{df}{dy} = 1 \text{ will give the required TV}$$

**TV will be  $x y z = \{011, 110\}$**

Find TV to test fault **s-a-0** at node **w** using BD method



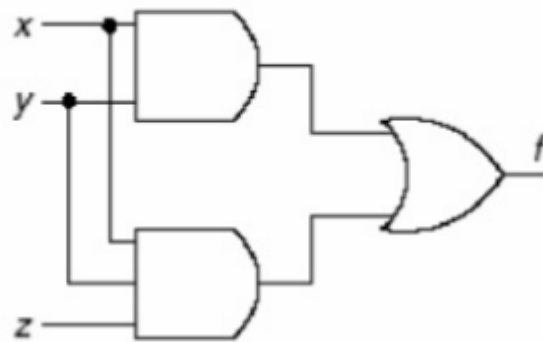
$$f = x \cdot y + w ; w = \bar{y} \cdot z ; f_w = 1 ; f_{\bar{w}} = x \cdot y$$

$$w \cdot \frac{df}{dw} = w \cdot (f_w \oplus f_{\bar{w}}) = \bar{y} \cdot z (1 \oplus x \cdot y) = \bar{y} \cdot z (\bar{x} + \bar{y}) = \bar{x} \bar{y} z + \bar{y} z = \bar{y} z$$

$$w \cdot \frac{df}{dw} = 1 \text{ will give the required TV}$$

**TV will be  $x y z = \{x01\}$**

Find TV to test fault **s-a-0** at node **Z** using BD method



$$f = x \cdot y + x \cdot y \cdot z ; f_z = x \cdot y ; f_{\bar{z}} = x \cdot y$$

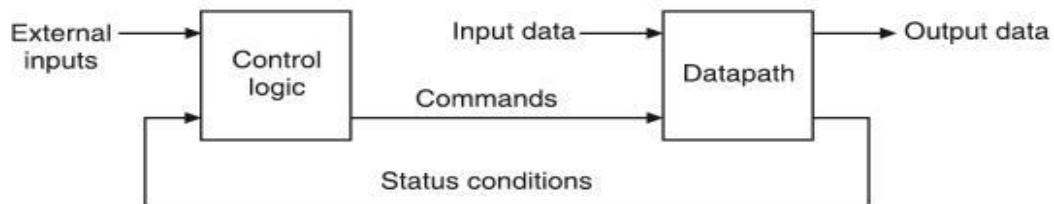
- $z \cdot \frac{df}{dz} = z \cdot (f_z \oplus f_{\bar{z}}) = 1$  will give the required TV
- But,  $f_z \oplus f_{\bar{z}} = 0$
- The condition for testability ( $f_z \oplus f_{\bar{z}} = 1$ ) is not satisfiable
- Hence, the fault is undetectable
- Redundancy in the circuit is the cause for undetectable faults

## Unit V

### SM Charts

#### Algorithmic State Machines:

- ? The binary information stored in the digital system can be classified as either data or control information.
- ? The data information is manipulated by performing arithmetic, logic, shift and other data processing tasks.
- ? The control information provides the command signals that controls the various operations on the data in order to accomplish the desired data processing task.
- ? Design a digital system we have to design two subsystems data path subsystem and control subsystem.



Interaction between control logic and datapath.

#### ASM CHART:

- ? A special flow chart that has been developed specifically to define digital hardware algorithms is called ASM chart.
- ? A hardware algorithm is a step by step procedure to implement the desire task.

#### Difference b/n conventional flow chart and ASM chart:

- ? conventional flow chart describes the sequence of procedural steps and decision paths for an algorithm without concern for their time relationship
- ? An ASM chart describes the sequence of events as well as the timing relationship b/n the states of sequential controller and the events that occur while going from one state to the next

#### Basic Components of ASM charts

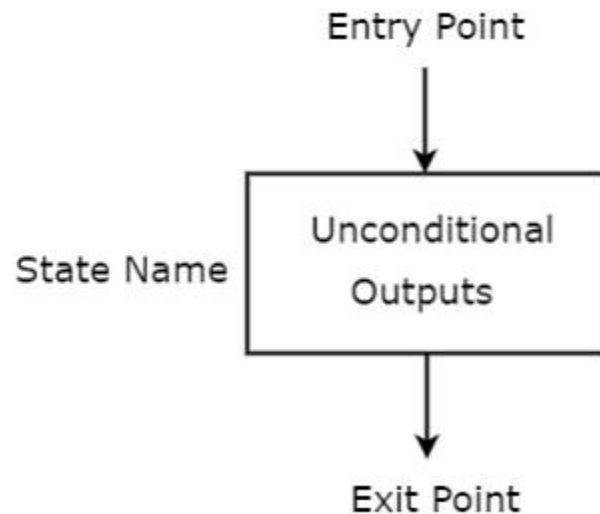
Following are the three basic components of ASM charts.

- State box
- Decision box

- Conditional output box

### State box

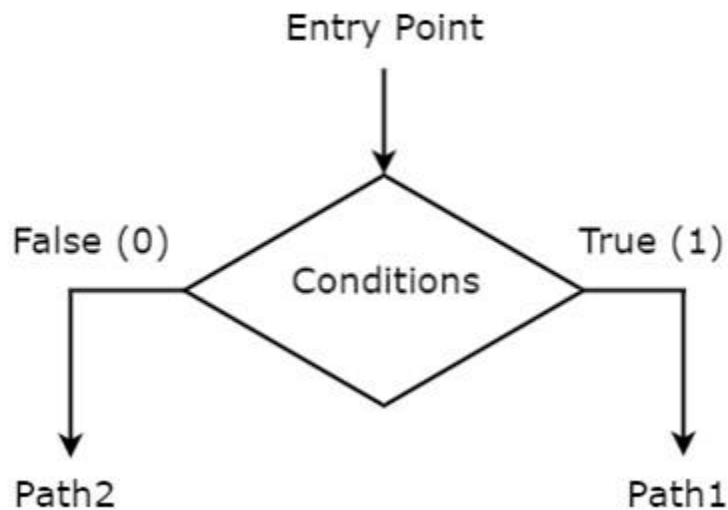
State box is represented in rectangular shape. Each state box represents one state of the sequential circuit. The **symbol** of state box is shown in the following figure.



It is having one entry point and one exit point. Name of the state is placed to the left of state box. The unconditional outputs corresponding to that state can be placed inside state box. **Moore** state machine outputs can also be placed inside state box.

### Decision box

Decision box is represented in diamond shape. The **symbol** of decision box is shown in the following figure.



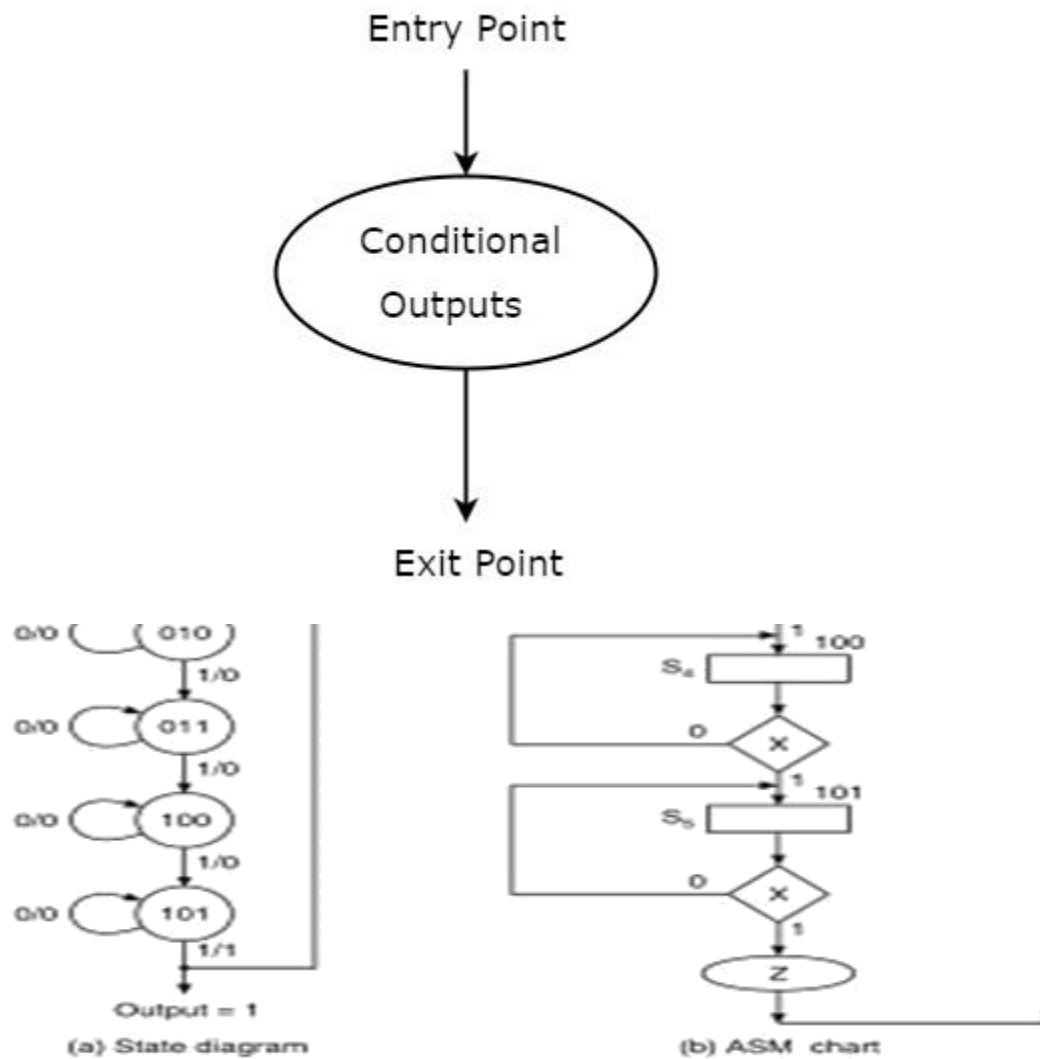
It is having one entry point and two exit paths. The inputs or Boolean expressions can be placed inside the decision box, which are to be checked whether they are true or false. If the condition is true, then it will prefer path1. Otherwise, it will prefer path2.

### Conditional output box

Conditional output box is represented in oval shape. The **symbol** of conditional output box is shown in the following figure.

It is also having one entry point and one exit point similar to state box. The conditional outputs can be placed inside state box. In general, **Mealy** state machine outputs are represented inside conditional output box. So, based on the requirement, we can use the above components

properly for drawing ASM charts.

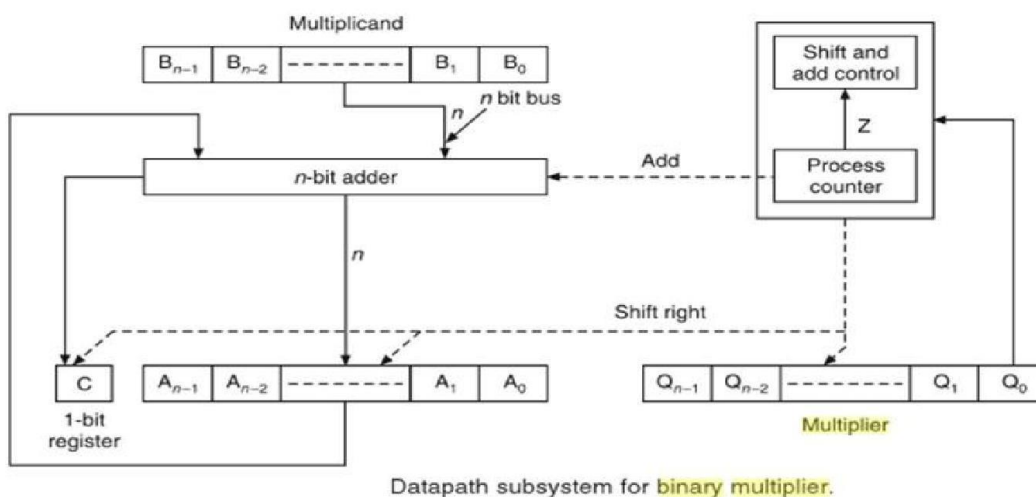


State diagram and ASM chart for mod-6 counter.

Binary Multiplier:

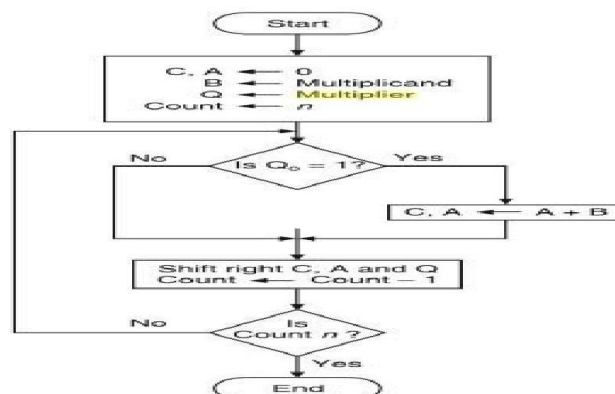
$$\begin{array}{r}
 1101 \\
 1010 \\
 \hline
 0000 \\
 1101 \\
 0000 \\
 1101 \\
 \hline
 10000010
 \end{array}
 \begin{array}{l}
 \leftarrow 13_{10} \dots \text{Multiplicand} \\
 \leftarrow 10_{10} \dots \text{Multiplier} \\
 \leftarrow \text{Partial product 1} \\
 \leftarrow \text{Partial product 2} \\
 \leftarrow \text{Partial product 3} \\
 \leftarrow \text{Partial product 4} \\
 \leftarrow 130_{10} \dots \text{Product}
 \end{array}$$

Data path subsystem for binary multiplier



#### Multiplication Operation Steps

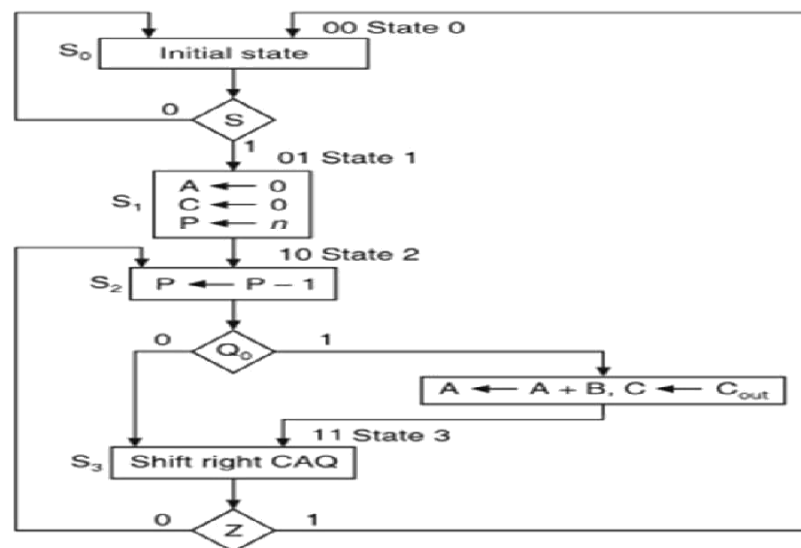
1. Bit 0 of multiplier operand ( $Q_0$  of Q register) is checked.
2. If bit 0 ( $Q_0$ ) is one then multiplicand and partial product are added and all bits of C, A and Q registers are shifted to the right one bit, so that the C bit goes into  $A_{n-1}$ ,  $A_0$  goes into  $Q_{n-1}$ , and  $Q_0$  is lost. If bit 0 ( $Q_0$ ) is 0, then no addition is performed, only shift operation is carried out.
3. Steps 1 and 2 are repeated n times to get the desired result in the A and Q registers.





B	C	A	Q	Components	Count P
1 1 0 1	0	0 0 0 0	1 0 1 0	$B \leftarrow \text{Multiplicand}$ $Q \leftarrow \text{Multiplier}$ $A \leftarrow 0, C \leftarrow 0, P \leftarrow n$	100 (4)
1 1 0 1	0	0 0 0 0	1 0 1 0	$P \leftarrow P - 1$ $Q_0 = 0$	011 (3)
	0	0 0 0 0	0 1 0 1	C A Q shifted right	
1 1 0 1	0	1 1 0 1	0 1 0 1	$P \leftarrow P - 1$ $Q_0 = 1, A \leftarrow A + B$	010 (2)
	0	0 1 1 0	1 0 1 0	C A Q shifted right	
1 1 0 1	0	0 1 1 0	1 0 1 0	$P \leftarrow P - 1$ $Q_0 = 0$	001 (1)
	0	0 0 1 1	0 1 0 1	C A Q shifted right	
1 1 0 1	1	0 0 0 0	0 1 0 1	$P \leftarrow P - 1$ $Q_0 = 1, A \leftarrow A + B$	000 (0)
	0	1 0 0 0	0 0 1 0	C A Q shifted right	

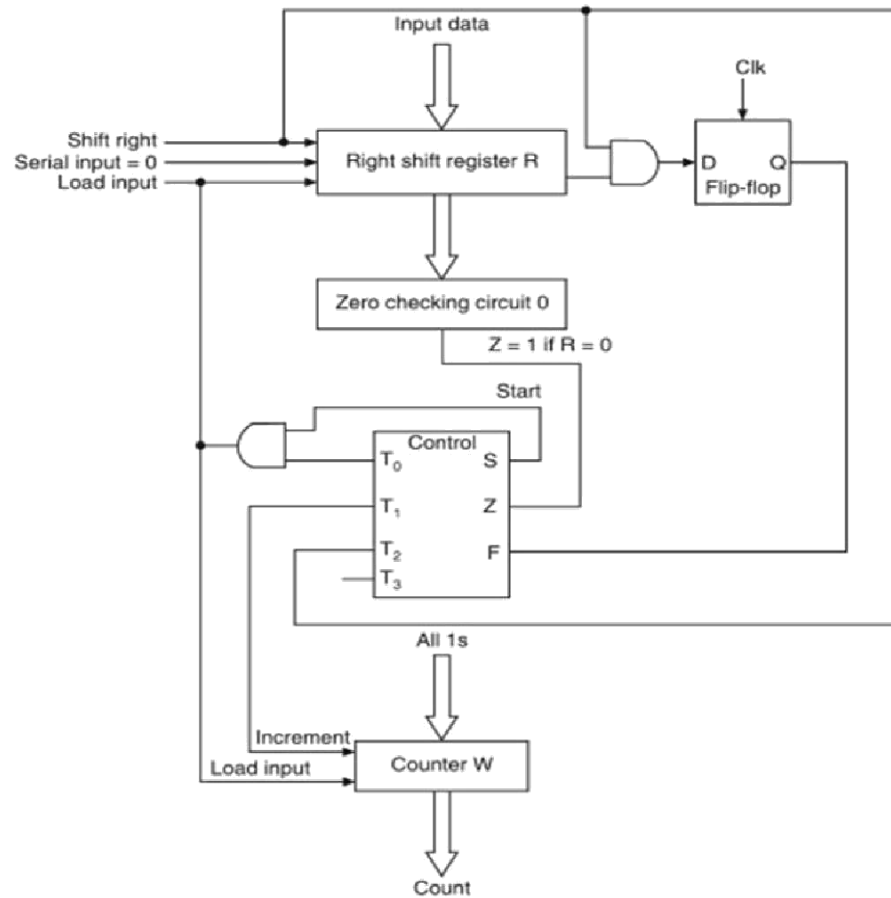
Flow chart for multiplication in a computer.



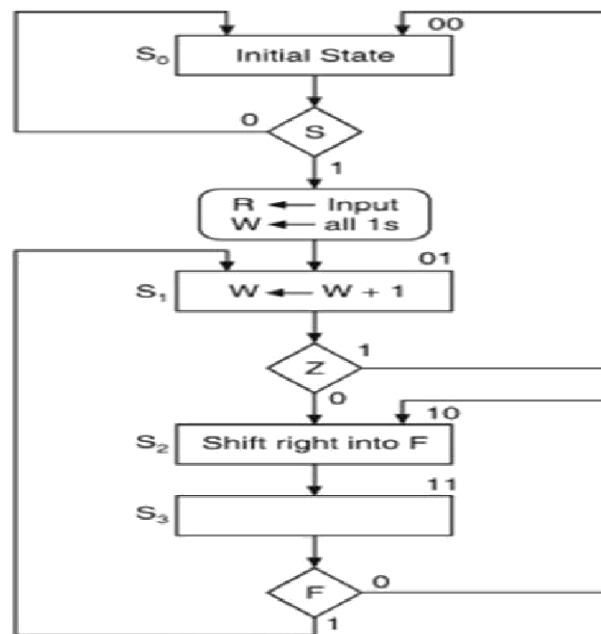
ASM chart for a binary multiplier.

## ASM FOR WEIGHING MACHINE

In the algorithm for tabular minimization of Boolean expressions, we have to arrange the minterms in the ascending order of their weights. This is only one of the many situations when we have to examine the 1s of a given binary word. The weight of a binary number is defined as the number of 1s present in its binary representation.



Datapath subsystem for weighing machine.



ASM chart for weighing machine.

*State  $S_0$ :* Initially the weighing machine is in state  $S_0$ . The weighing process starts when start (S) signal becomes 1. While in state  $S_0$ , if S is 1, the clock pulse causes three jobs to be done simultaneously:

1. Binary number is loaded into register R.
2. W register is set to all 1s.
3. The machine is transferred to state  $S_1$ .

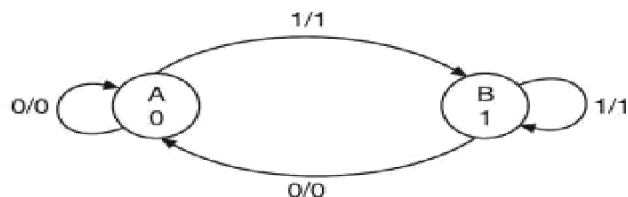
*State  $S_1$ :* While in state  $S_1$ , the clock pulse causes two jobs to be done simultaneously:

1. Counter W is incremented by 1 (in the first round, all 1s become all 0s).
2. If Z is 0, the machine goes to the state  $S_2$ ; if Z is 1, the machine goes to state  $S_0$ .

*State  $S_2$ :* In this state, register R is shifted right by 1 bit so that LSB goes into F and MSB is loaded with 0.

*State  $S_3$ :* In this state, the value of F is checked. If it is 0, the machine is transferred to the state  $S_2$ , otherwise the machine is transferred to state  $S_1$ . Thus, when  $F = 1$ , W is incremented.

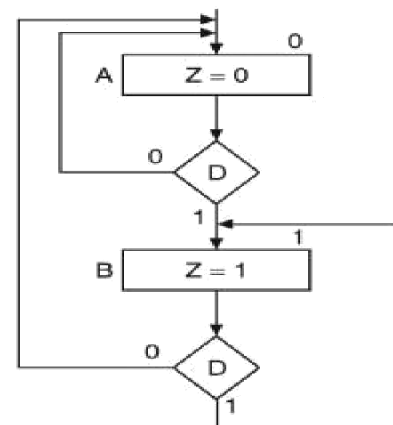
All the operations occur in coincidence with the clock pulse while in the corresponding state. Also notice that the register R should eventually contain all 0s when the last 1 is shifted into it.



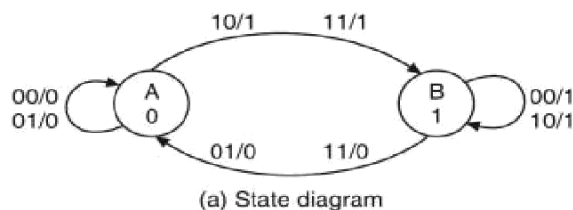
(a) State diagram

PS	NS, O/P Input D	
	D = 0	D = 1
A	A, 0	B, 1
B	A, 0	B, 1

(b) State table



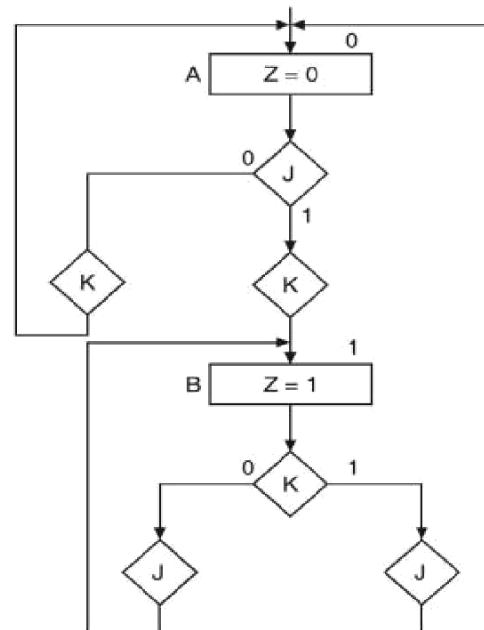
(c) ASM chart



(a) State diagram

PS	NS, O/P Input J-K			
	00	01	10	11
A	A, 0	A, 0	B, 1	B, 1
B	B, 1	A, 0	B, 1	A, 0

(b) State table



(c) ASM chart