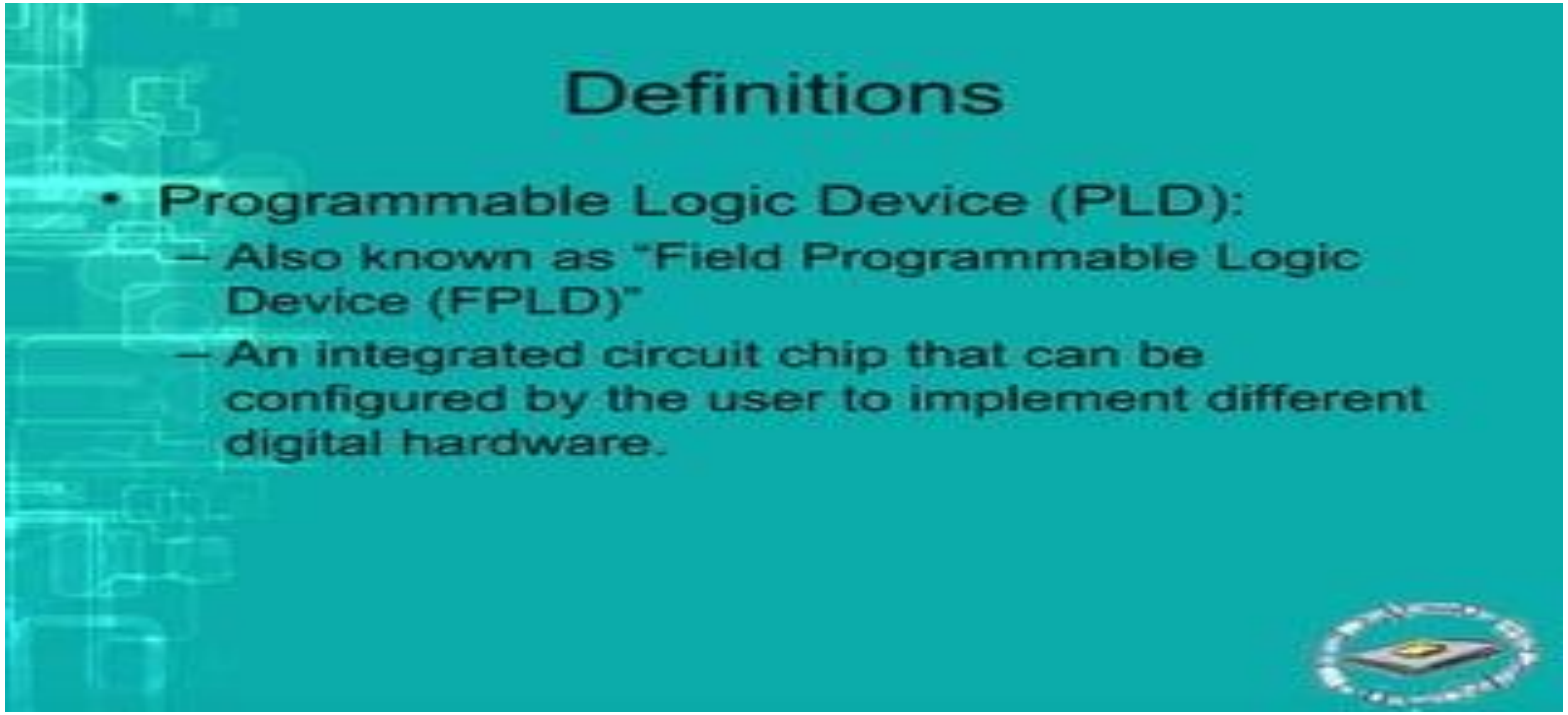
Programmable Logic Device

PLD

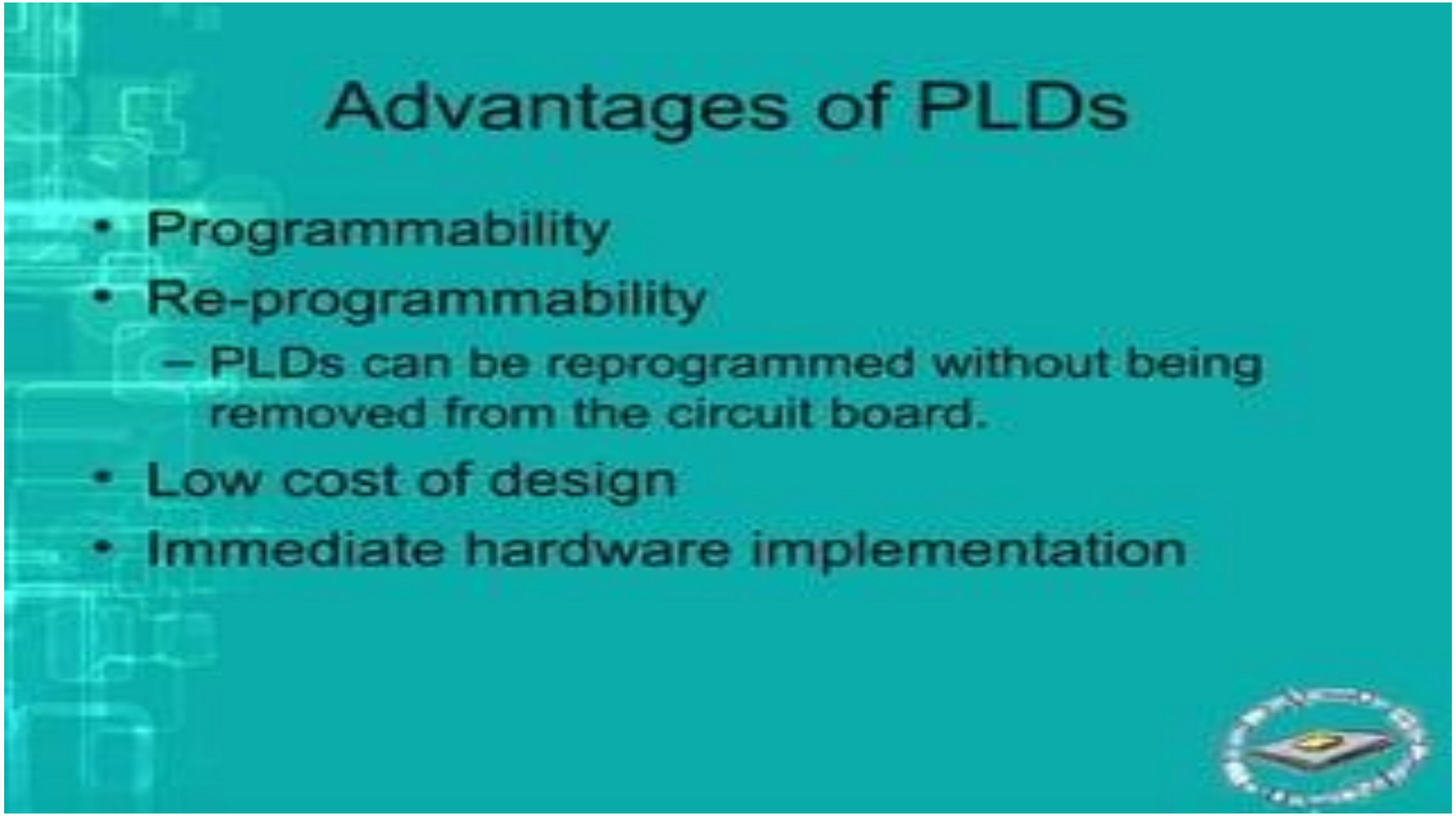
PLD

- An IC that contains large numbers of gates, flip-flops, etc. that can be configured by the user **to perform different functions** is called a **Programmable Logic Device** (PLD). It permits **elaborate** digital logic designs to be implemented by the user on a single device. The **internal logic gates and/or connections of PLDs** can be **changed/configured** by **a programming process**. On the other hand, Programmable Logic Devices (PLDs) are the components which **do not have a specific function associated with them.**



Definitions

- Programmable Logic Device (PLD):
  - Also known as "Field Programmable Logic Device (FPLD)"
  - An integrated circuit chip that can be configured by the user to implement different digital hardware.



Advantages of PLDs

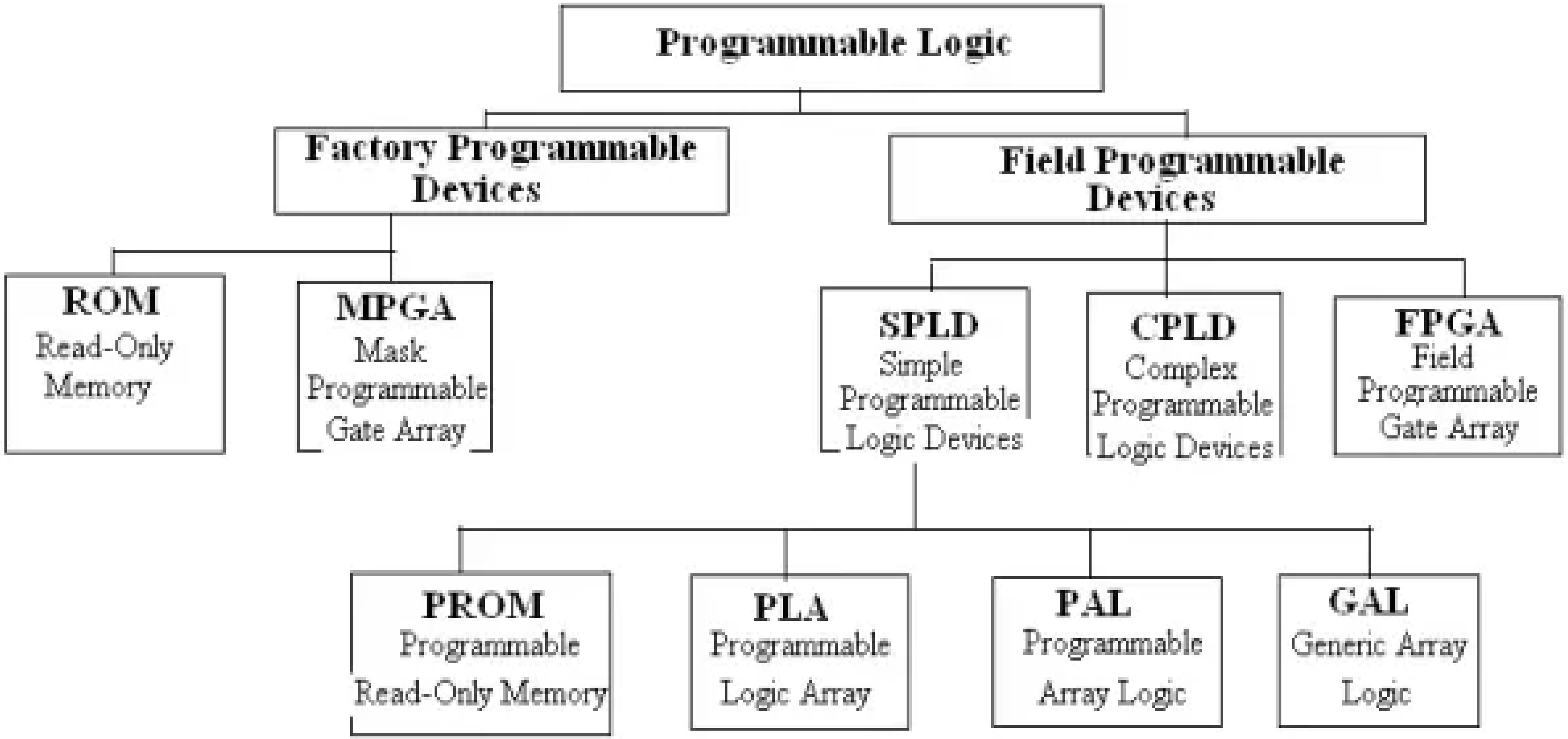
- Programmability
- Re-programmability
  - – PLDs can be reprogrammed without being removed from the circuit board.
- Low cost of design
- Immediate hardware implementation

PLD

- The purpose of a PLD device is to permit elaborate digital logic designs to be implemented by the user in a single device.

- Can be erased electrically and reprogrammed with a new design, making them very well suited for academic and prototyping

- ***Types of Programmable Logic Devices***
- SPLDs (Simple Programmable Logic Devices)
  - ROM (Read-Only Memory)
  - PLA (Programmable Logic Array)
  - PAL (Programmable Array Logic)
  - GAL (Generic Array Logic)
- CPLD (Complex Programmable Logic Device)
- FPGA (Field-Programmable Gate Array)



Programmable Logic

Factory Programmable Devices

Field Programmable Devices

ROM
Read-Only Memory

MPGA
Mask Programmable Gate Array

SPLD
Simple Programmable Logic Devices

CPLD
Complex Programmable Logic Devices

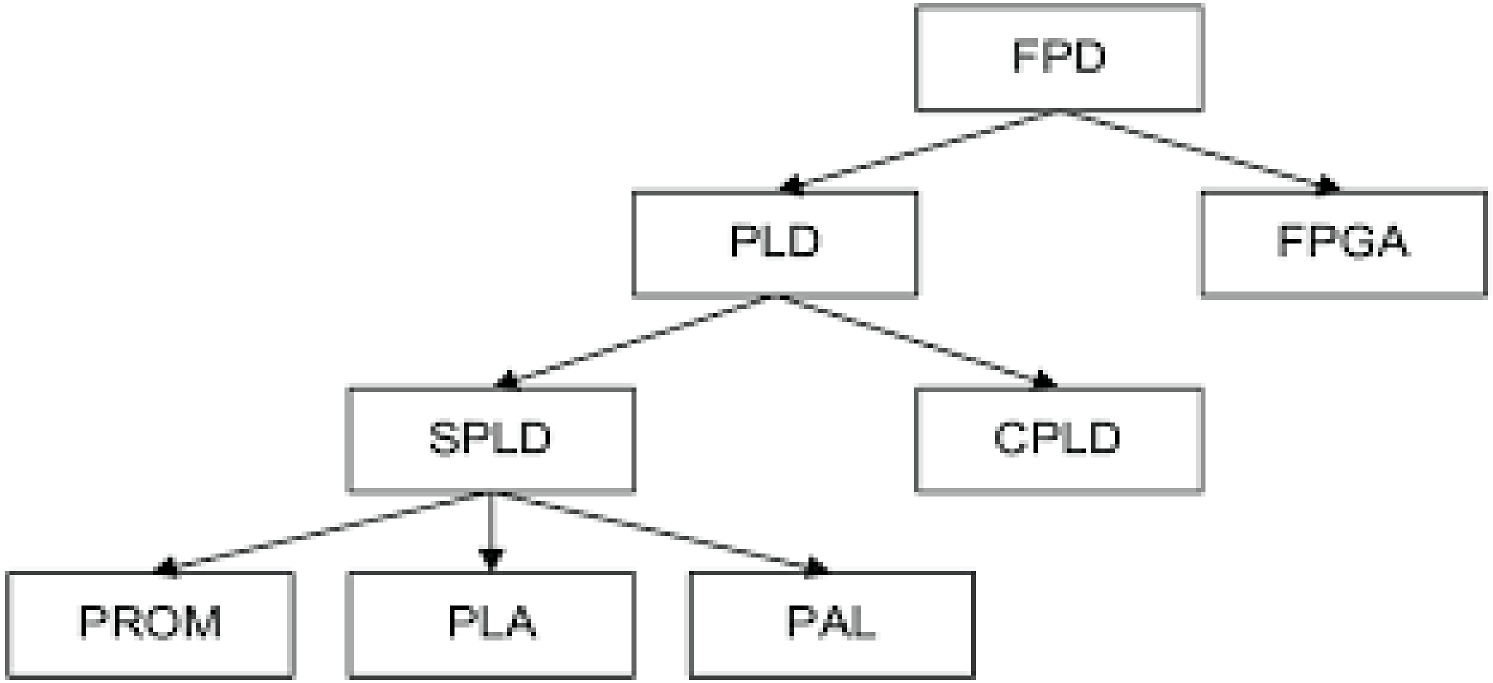
FPGA
Field Programmable Gate Array

PROM
Programmable Read-Only Memory

PLA
Programmable Logic Array

PAL
Programmable Array Logic

GAL
Generic Array Logic



FPD

PLD

FPGA

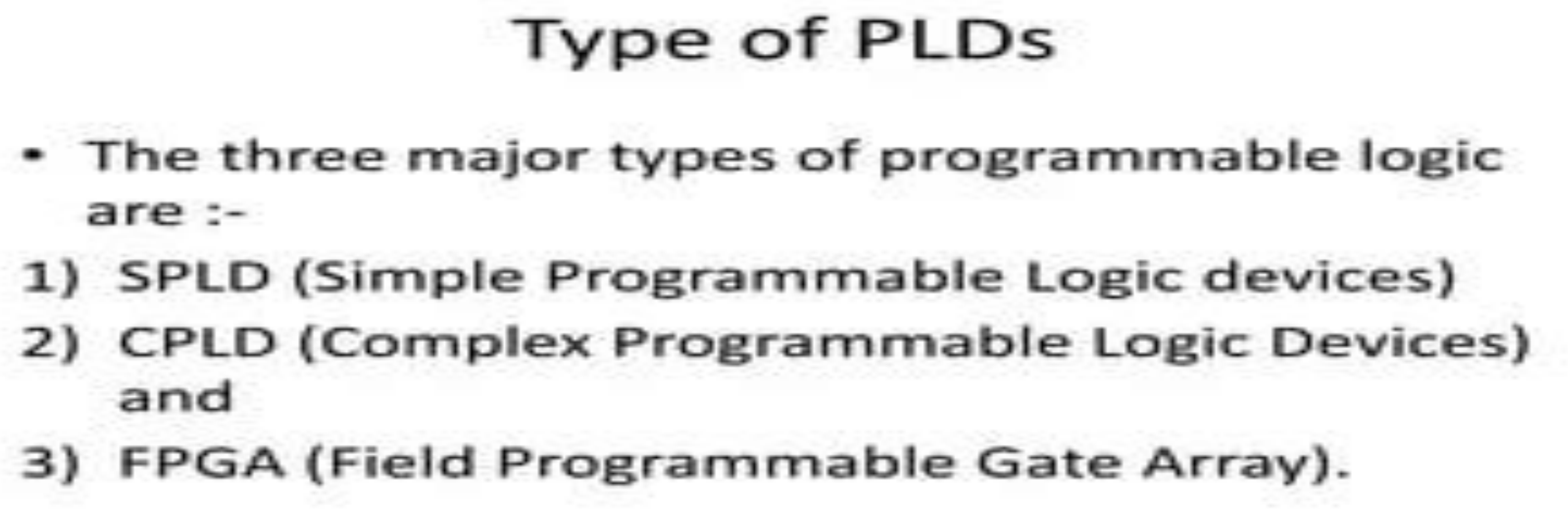
SPLD

CPLD

PROM

PLA

PAL



Type of PLDs

- The three major types of programmable logic are :-

1) SPLD (Simple Programmable Logic devices)
2) CPLD (Complex Programmable Logic Devices) and
3) FPGA (Field Programmable Gate Array).

PLDs

ROM : Programmable OR array

PLA : Programmable Logic Array .
Programmable OR – AND arrays.

PAL : Programmable Array Logic .
Programmable AND array, fixed OR

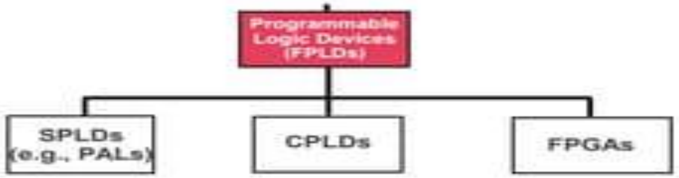
GAL : Generic Array Logic
Can be configured to emulate many earlier PLDs including those with internal Flip-Flops

CPLD : Complex PLD

FPGA : Field Programmable Gate Arrays

Three FPLD Types

- Simple Programmable Logic Device (SPLD)
  - LSI device
  - Less than 1000 logic gates
- Complex Programmable Logic Device (CPLD)
  - VLSI device
  - Higher logic capacity than SPLDs
- Field Programmable Gate Array (FPGA)
  - VLSI device
  - Higher logic capacity than CPLDs

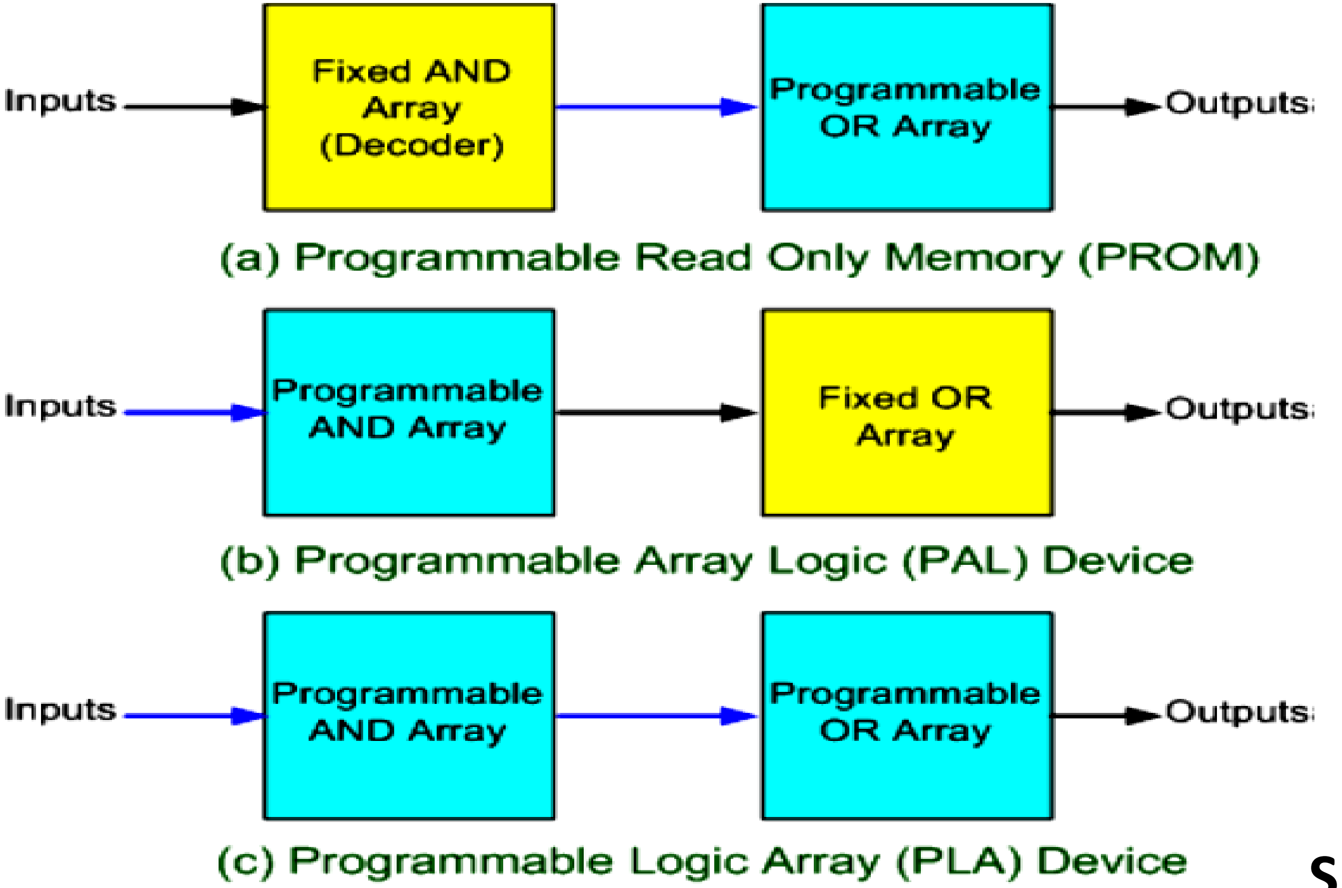


Programmable Logic Devices (FPLDs)

SPLDs (e.g., PALs)

CPLDs

FPGAs

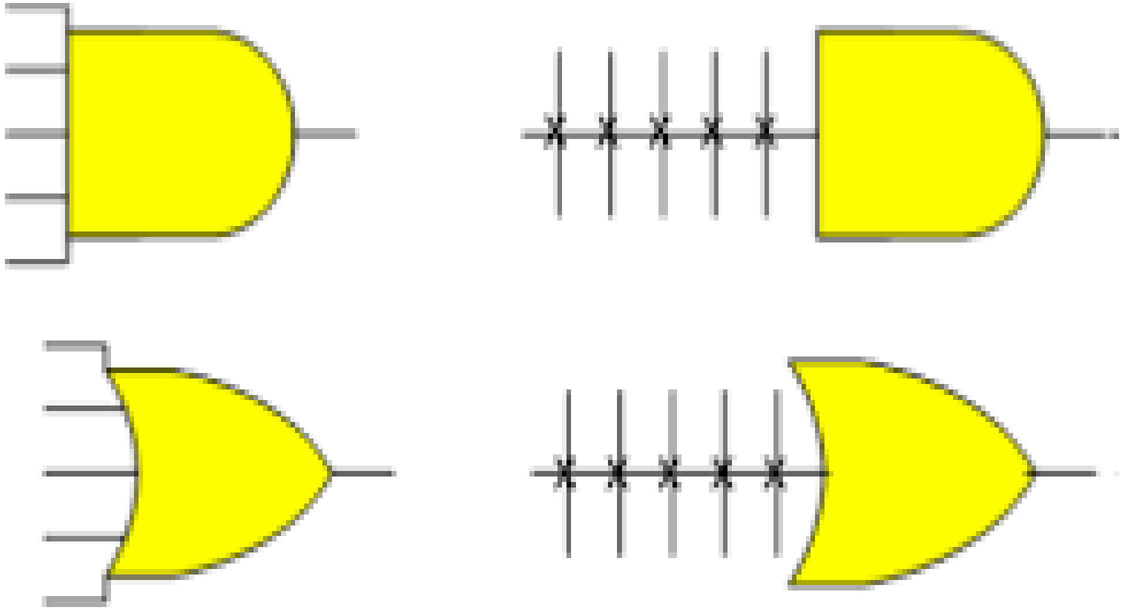


(a) Programmable Read Only Memory (PROM)

(b) Programmable Array Logic (PAL) Device

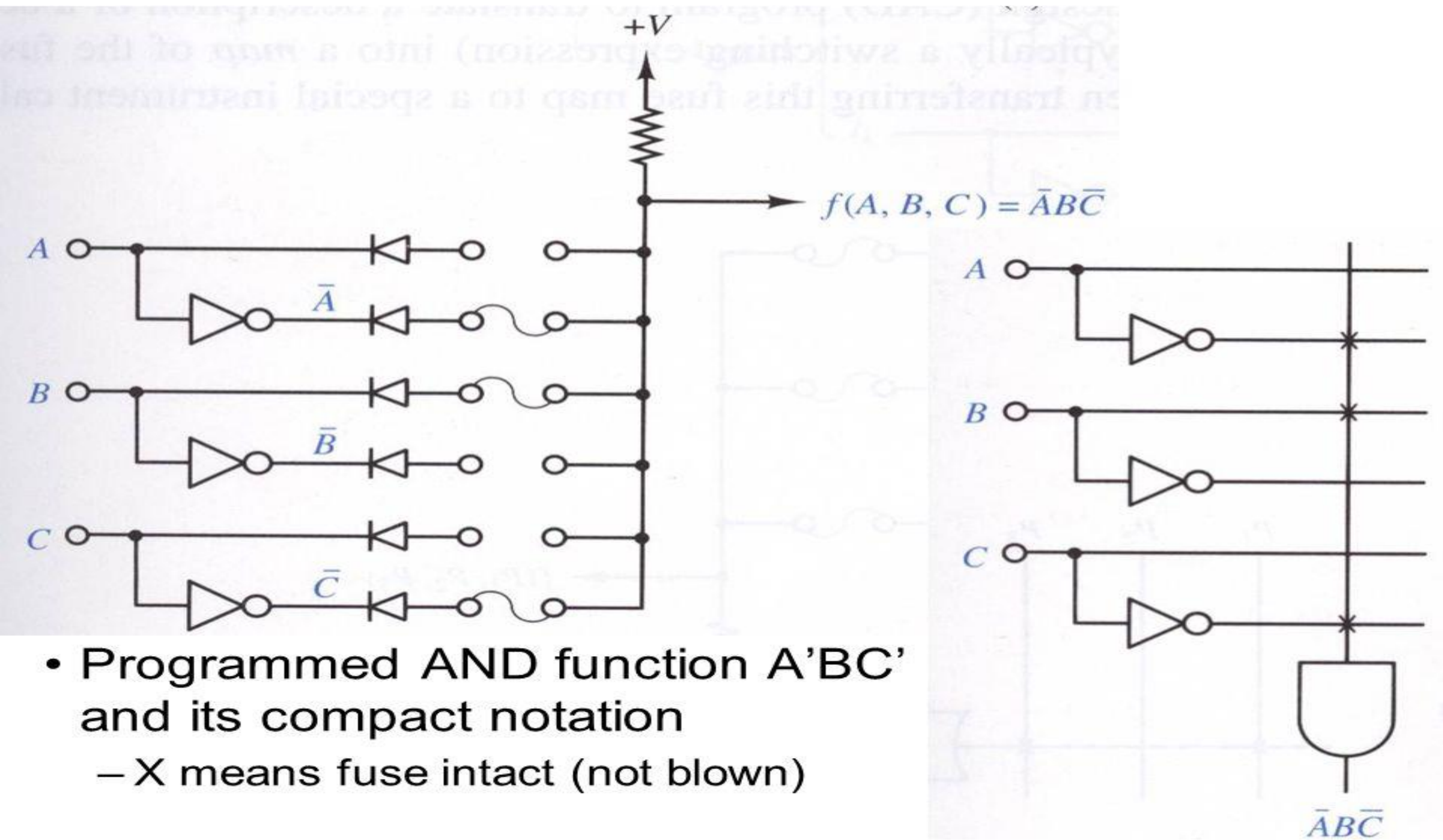
(c) Programmable Logic Array (PLA) Device

SPLD

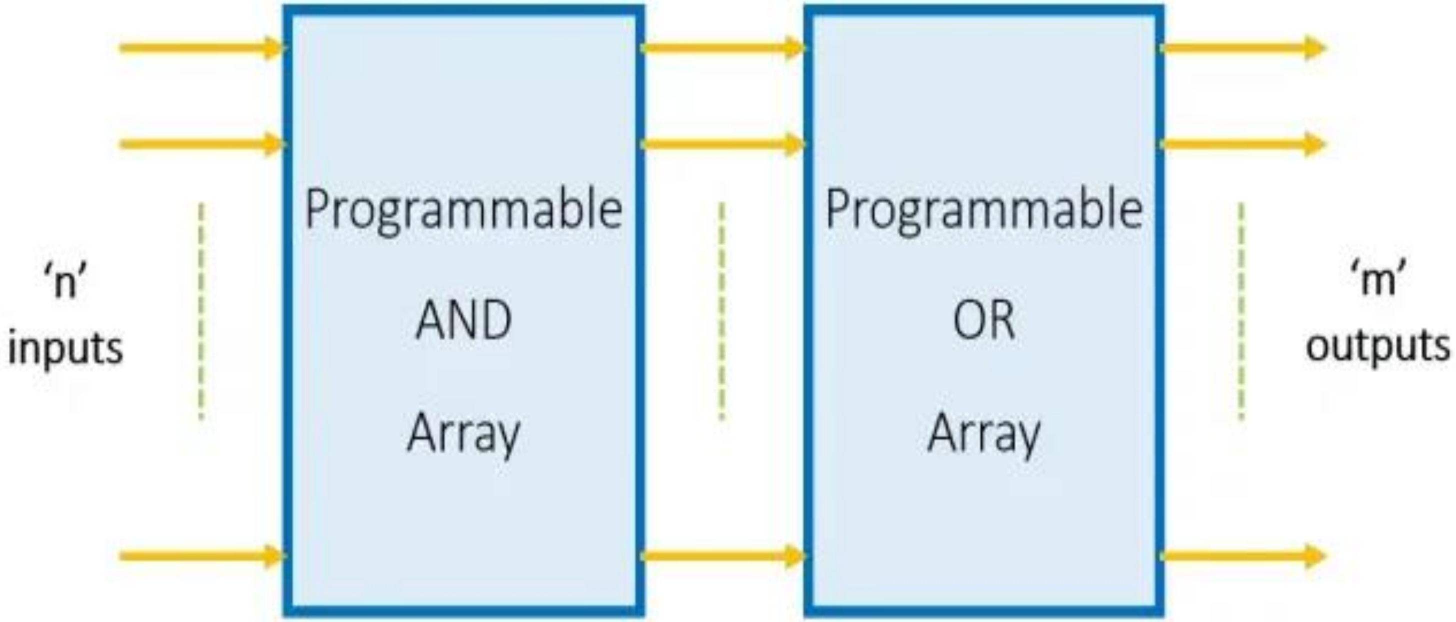


(a) Conventional Symbol

(b) Array Logic Symbol

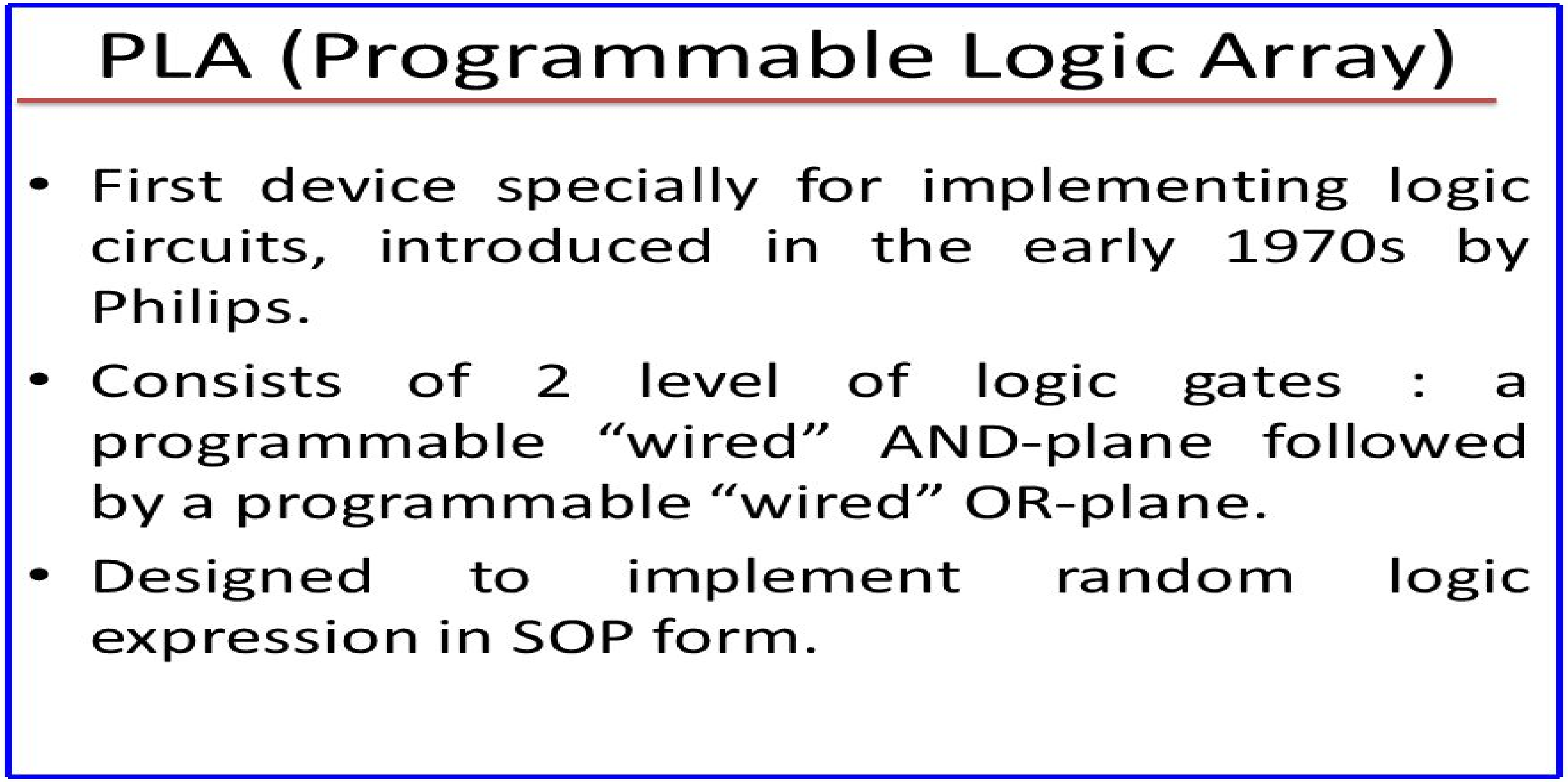


- Programmed AND function A’BC’ and its compact notation
  - X means fuse intact (not blown)



**Programmable
Logic Device**

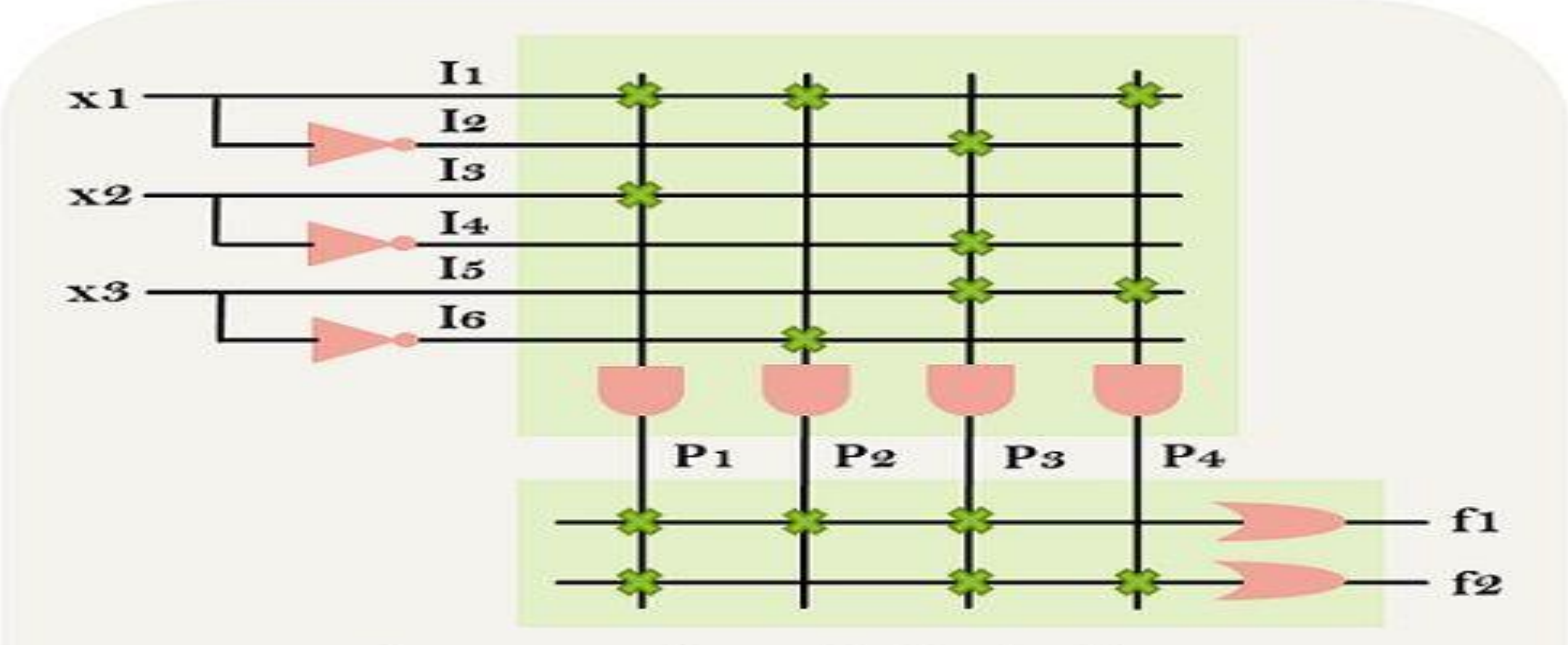
Programmable Logic Array



PLA (Programmable Logic Array)

- First device specially for implementing logic circuits, introduced in the early 1970s by Philips.
- Consists of 2 level of logic gates : a programmable “wired” AND-plane followed by a programmable “wired” OR-plane.
- Designed to implement random logic expression in SOP form.

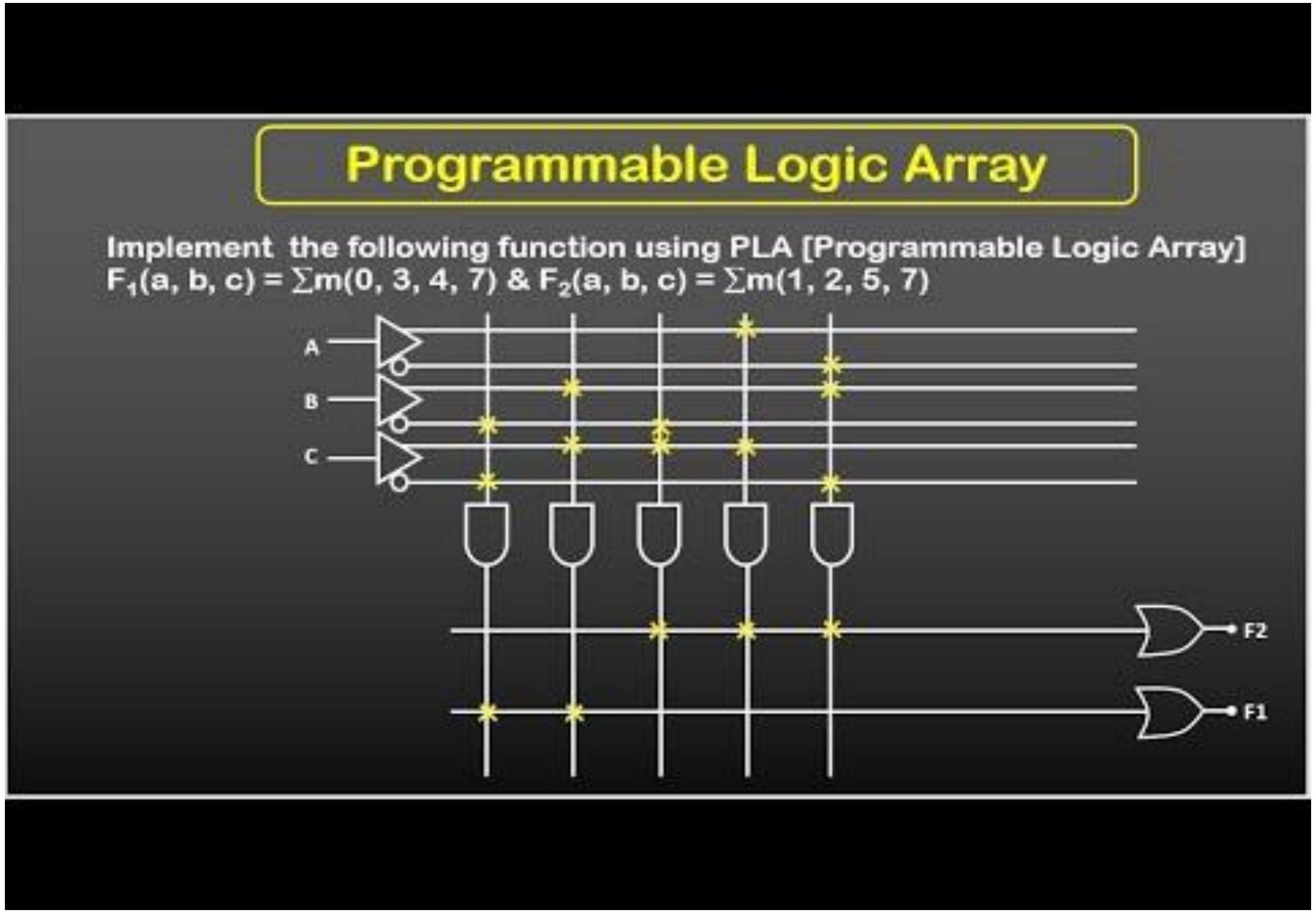
Programmable Logic Array



$$f_1 = x_1x_2 + x_1x'_3 + x'_1x'_2x_3$$
$$f_2 = x_1x_2 + x'_1x'_2x_3 + x_1x_3$$

Programmable Logic Array

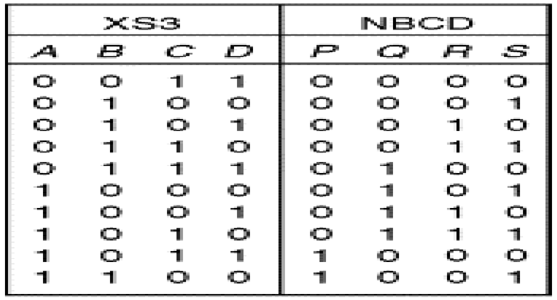
Programmable Logic Array



Programmable Logic Array

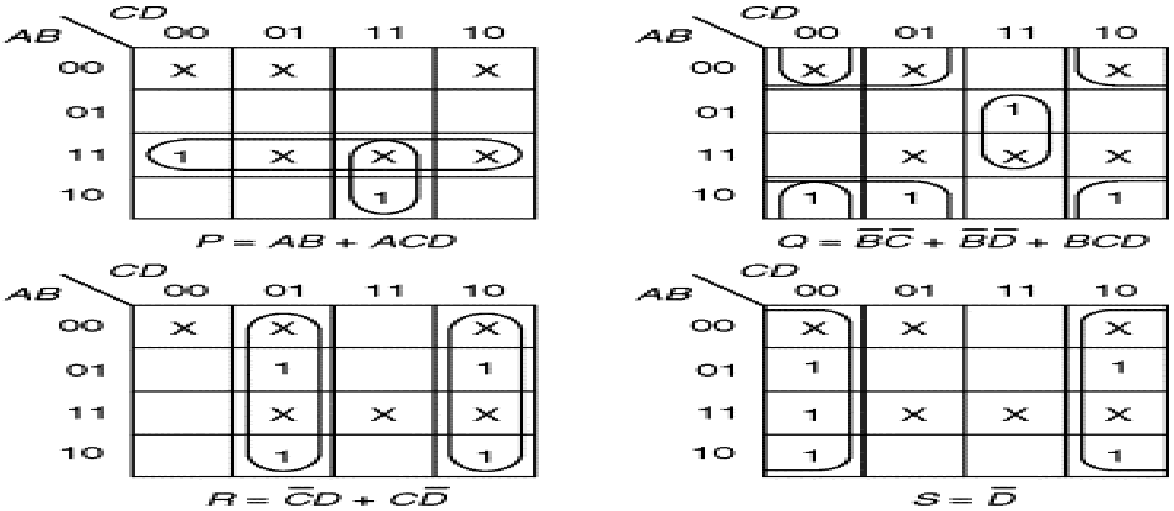
Implement the following function using PLA [Programmable Logic Array]
$F_1(a, b, c) = \sum m(0, 3, 4, 7)$ & $F_2(a, b, c) = \sum m(1, 2, 5, 7)$

Programmable Logic Array



XS3				NBCD			
A	*B*	*C*	*D*	*P*	*Q*	*R*	*S*
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	1
0	1	0	1	0	0	1	0
0	1	1	0	0	0	1	1
0	1	1	1	0	1	0	0
1	0	0	0	0	1	0	1
1	0	0	1	0	1	1	0
1	0	1	0	0	1	1	1
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	1

(a)



AB \ *CD*	00	01	11	10
00	X	X		X
01				
11	1	X	X	X
10			1	

$P = AB + ACD$

AB \ *CD*	00	01	11	10
00	X	X		X
01			1	
11		X	X	X
10	1	1		1

$Q = \overline{B}\overline{C} + \overline{B}\overline{D} + BCD$

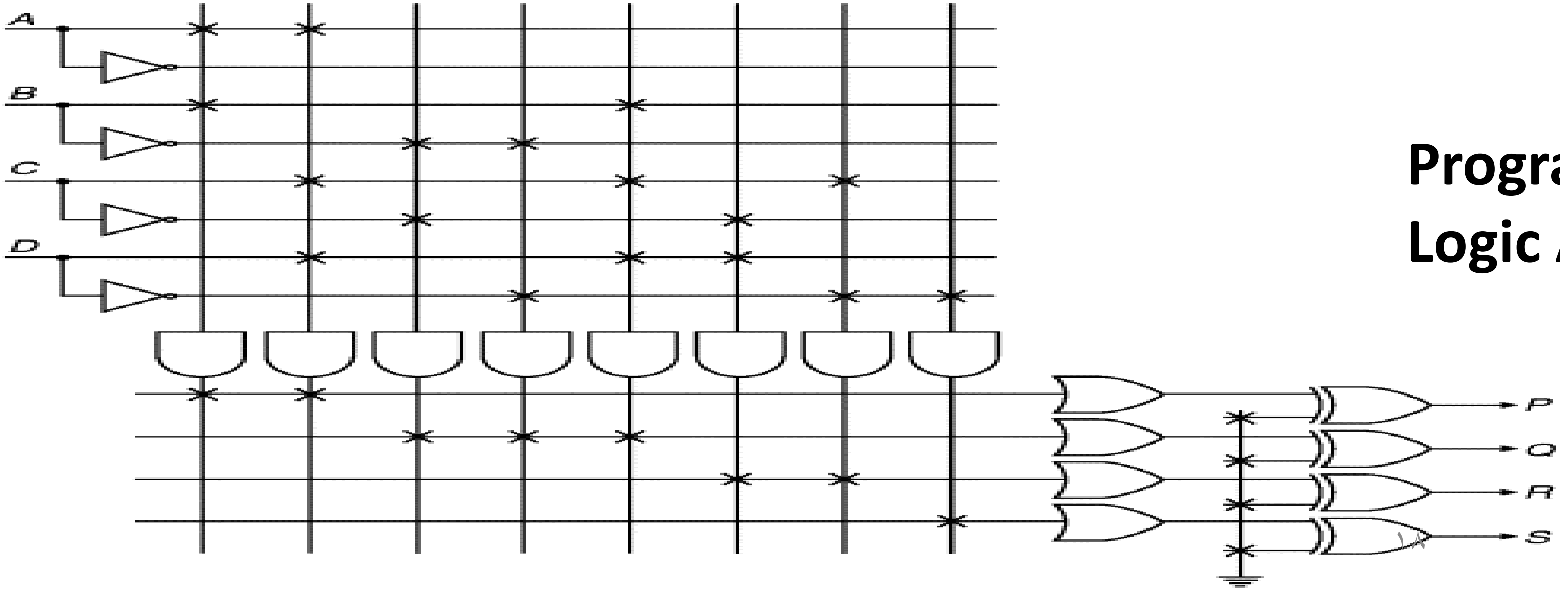
AB \ *CD*	00	01	11	10
00	X	X		X
01		1		1
11		X	X	X
10		1		1

$R = \overline{C}D + C\overline{D}$

AB \ *CD*	00	01	11	10
00	X	X		X
01	1			1
11	1	X	X	X
10	1			1

$S = \overline{D}$

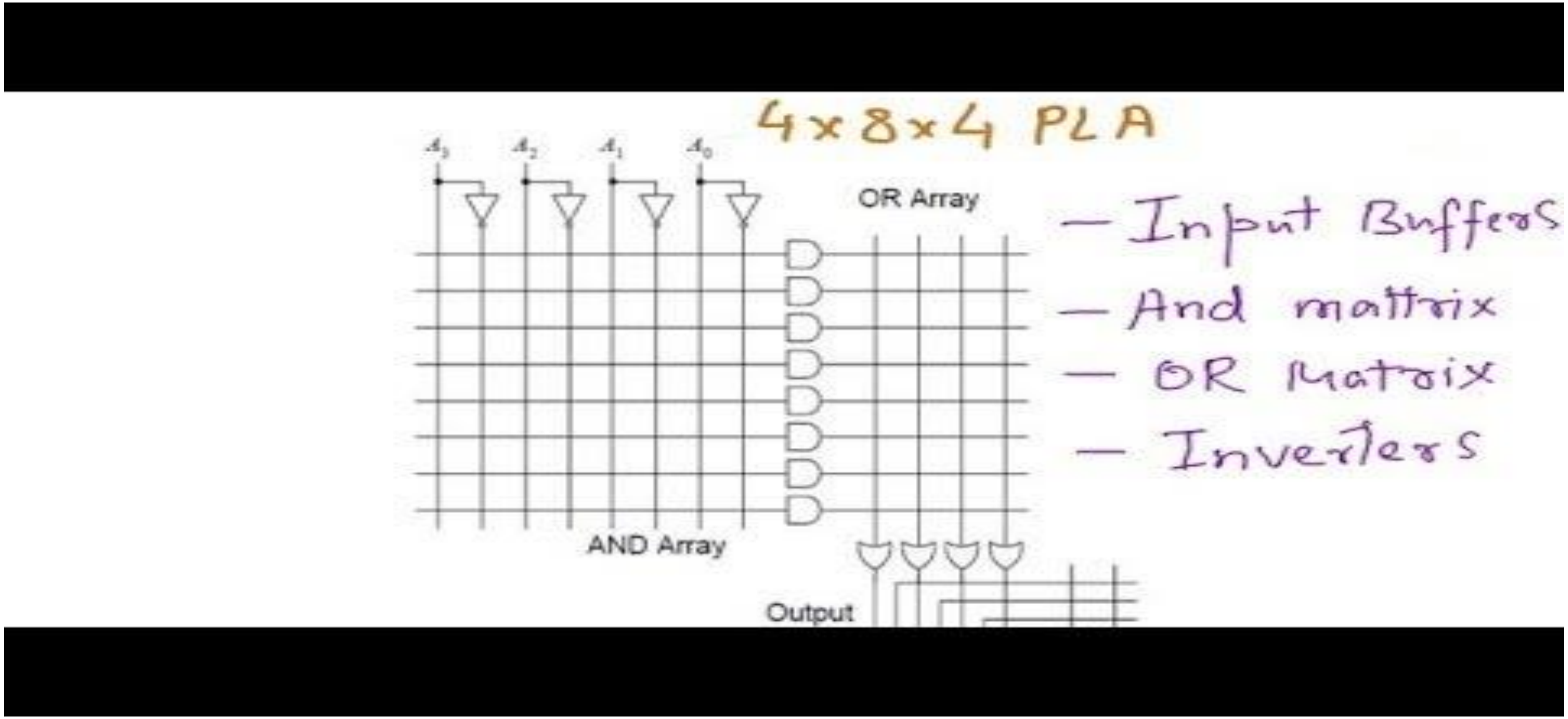
(b)



(c)

Programmable Logic Array

Programmable Logic Array



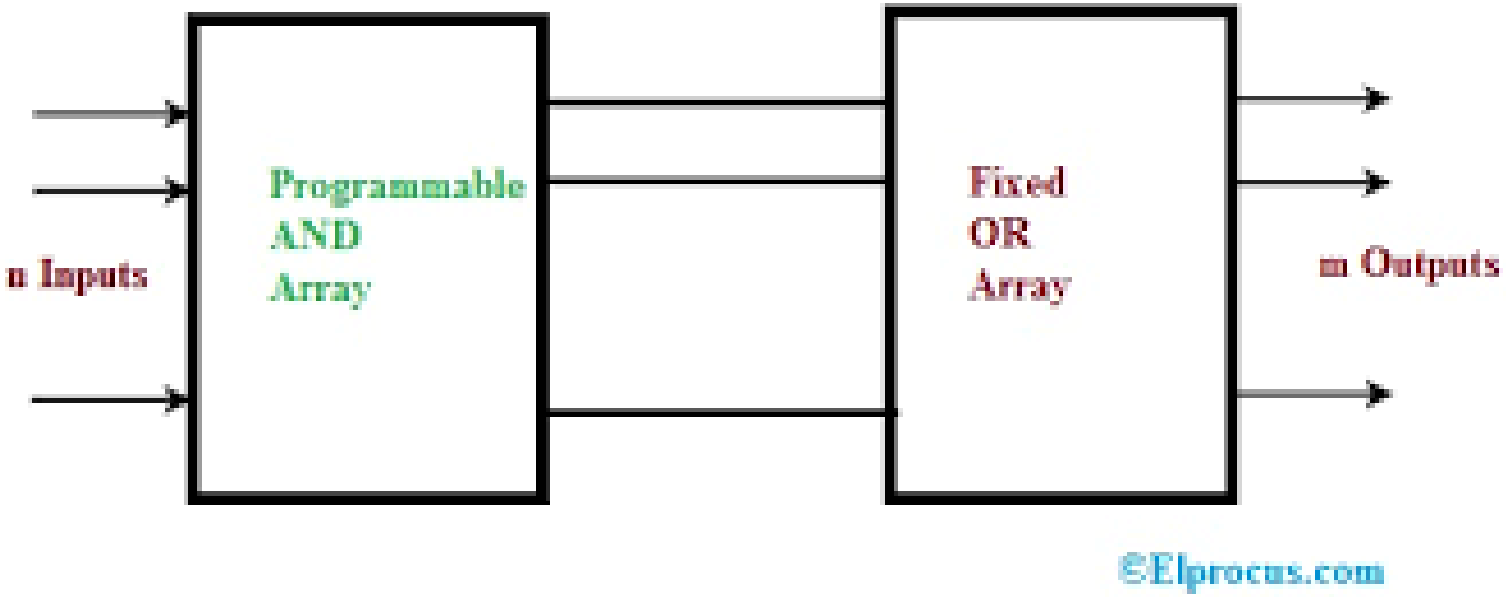
4x8x4 PLA

- Input Buffers
- And mattrix
- OR Matrix
- Inverters

Programmable Array Logic

PALs use an **OR gate array with fixed logic** while an **AND gate array which can be programmed** as per the requirement of the user. As a result, these devices express the output as a combination of inputs in sum-of-products form.

Programmable Array Logic



Programmable
Array Logic

Programmable Array Logic

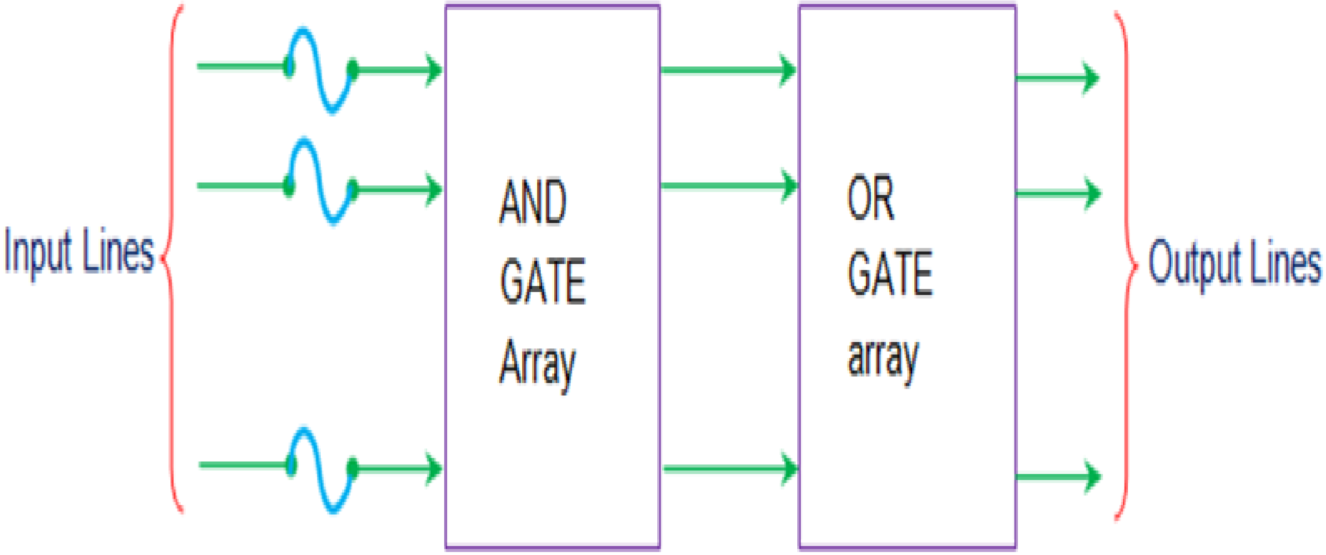
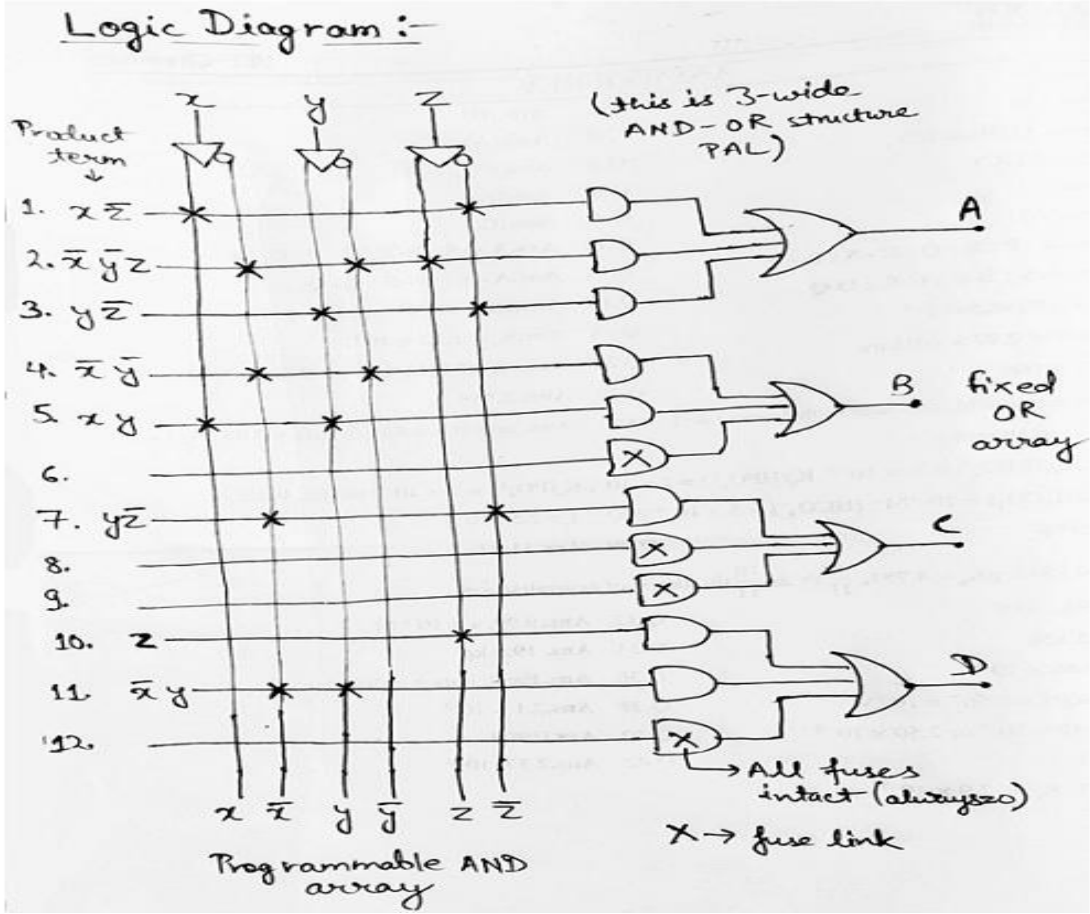


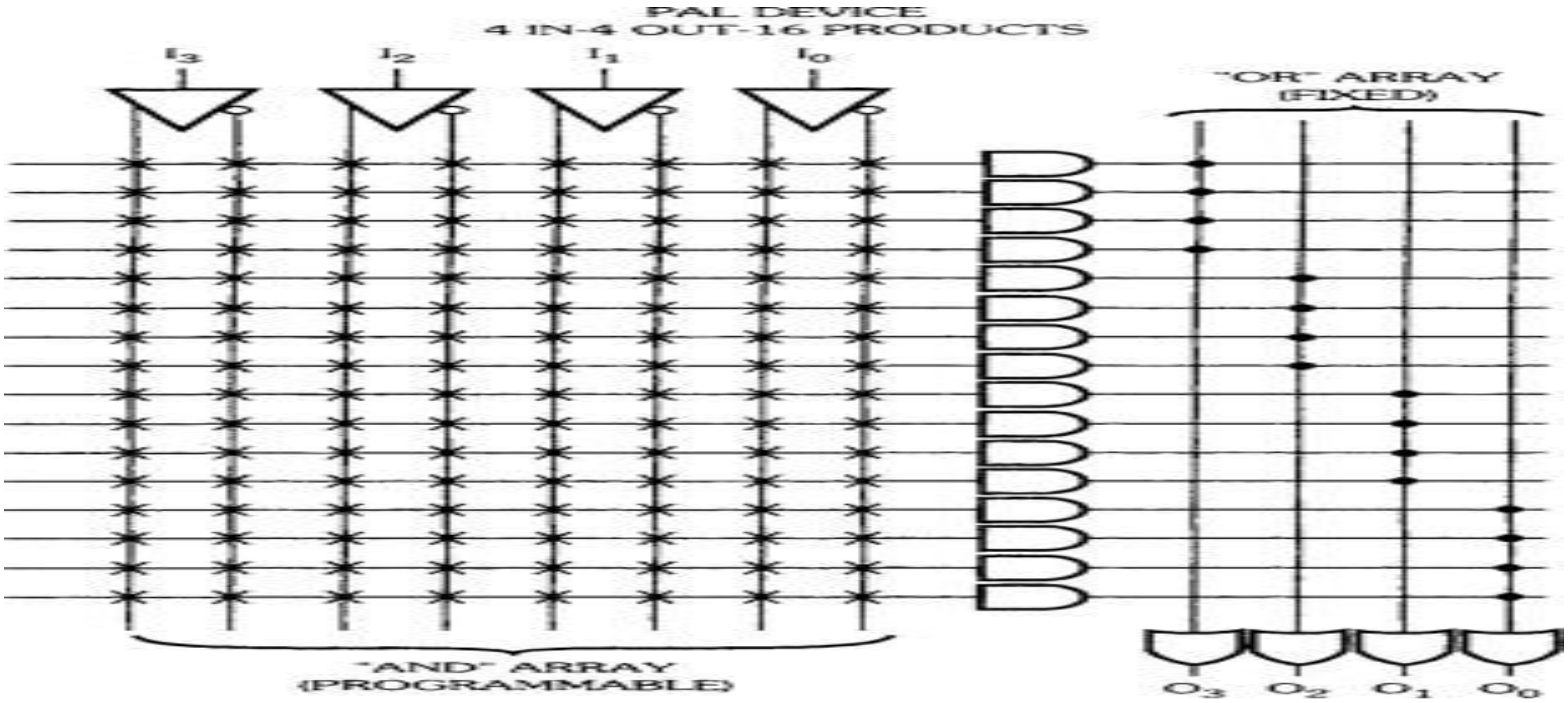
Figure 1 Programmable Array Logic (PAL)

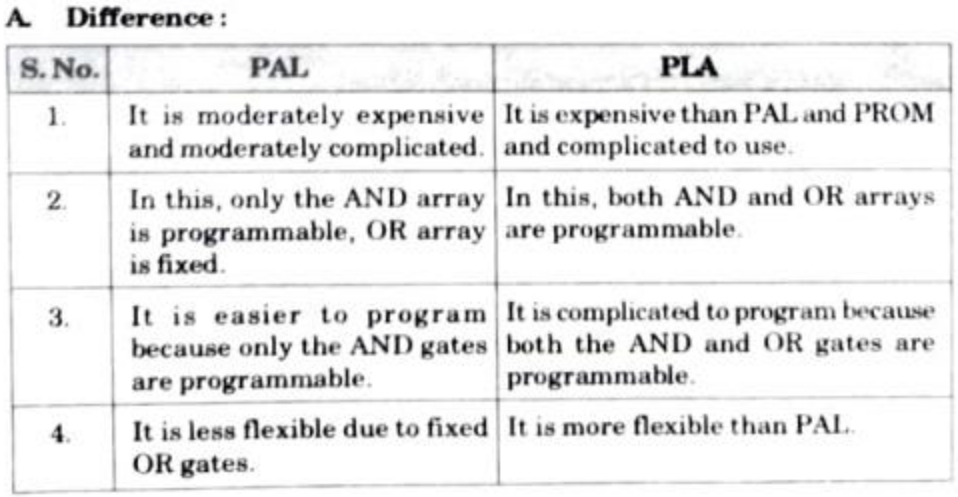
Programmable Logic Array



Programmable Logic Array

Programmable Logic Array





A. Difference :

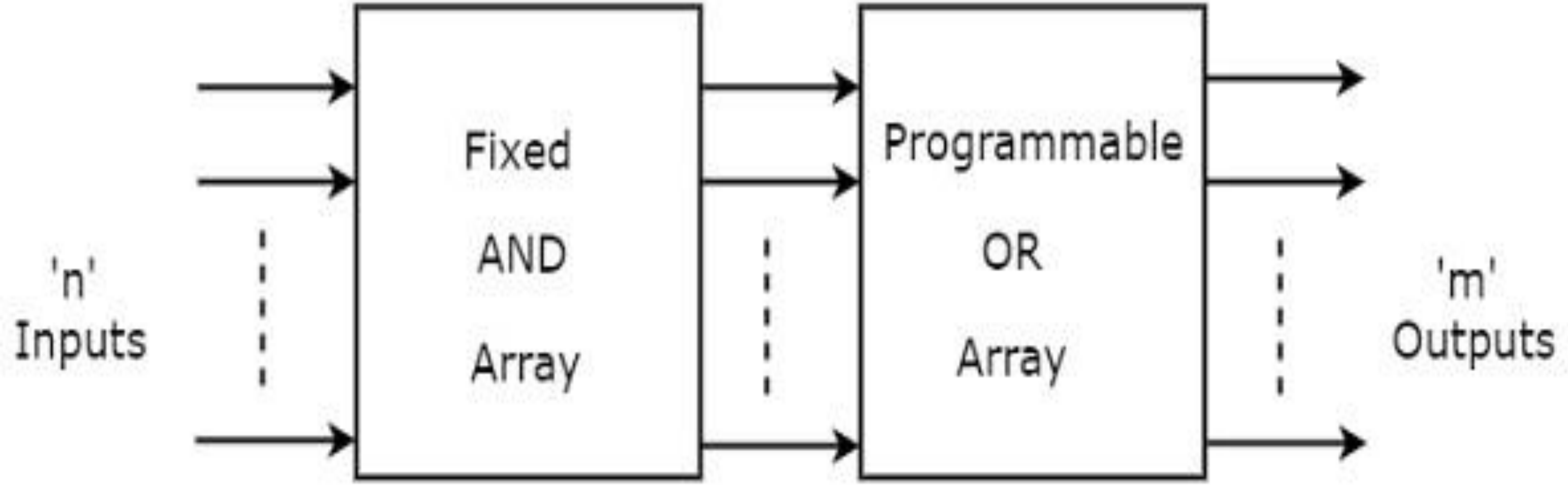
S. No.	PAL	PLA
1.	It is moderately expensive and moderately complicated.	It is expensive than PAL and PROM and complicated to use.
2.	In this, only the AND array is programmable, OR array is fixed.	In this, both AND and OR arrays are programmable.
3.	It is easier to program because only the AND gates are programmable.	It is complicated to program because both the AND and OR gates are programmable.
4.	It is less flexible due to fixed OR gates.	It is more flexible than PAL.

Generic Logic Array

Generic Logic Array (GLA)

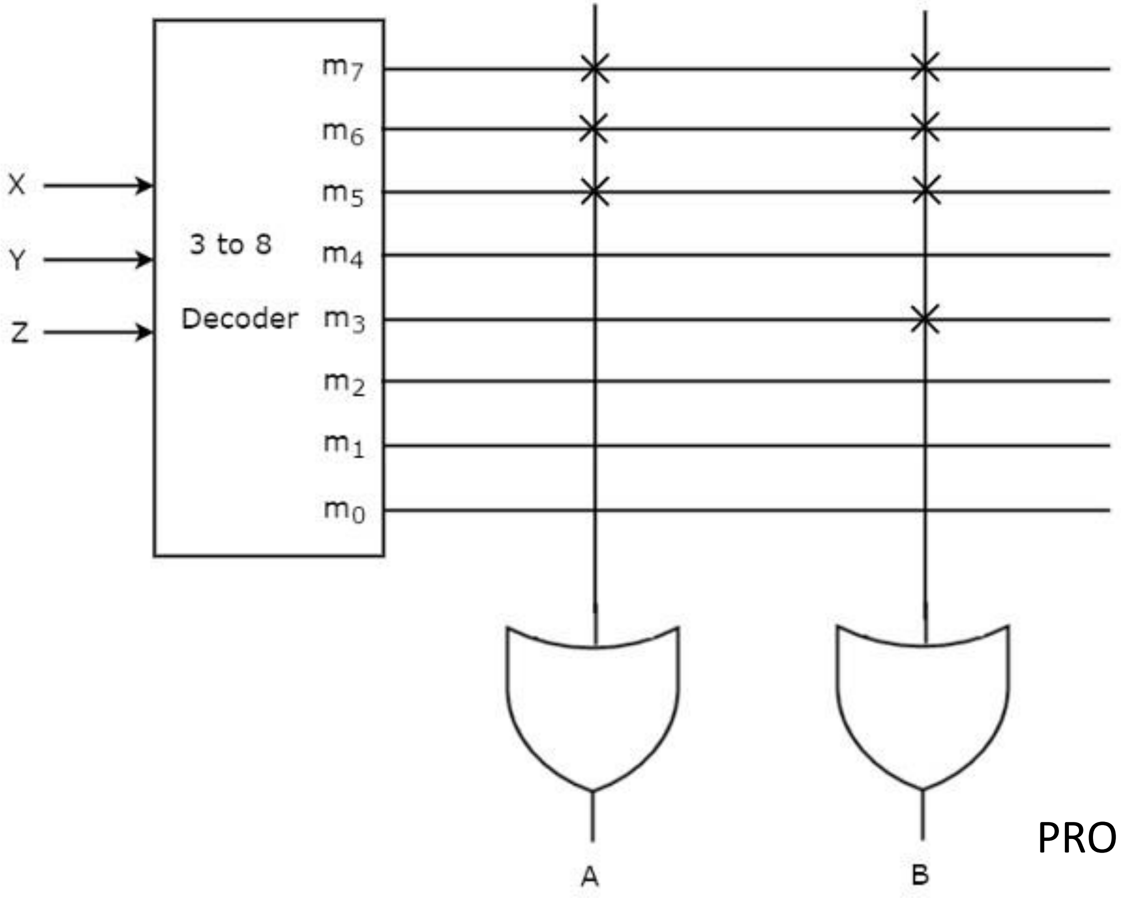
- These devices had their properties similar to those of PALs in addition to which they were electrically erasable and re-programmable. This important feature proved to be meritorious as it considerably eased the prototype design which in turn reduced the time to market.

Programmable Read-Only Memory



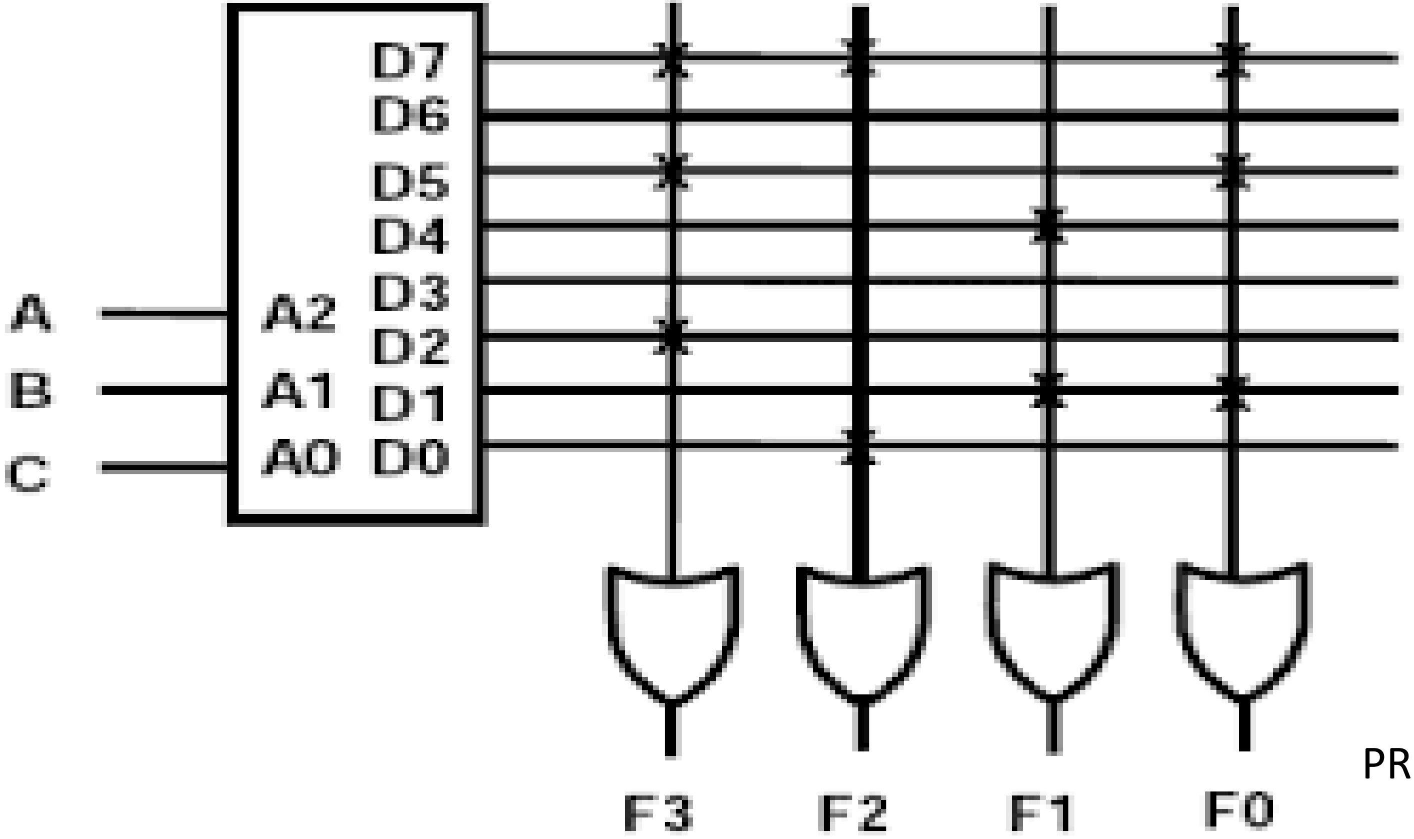
PROM

Programmable Read-Only Memory

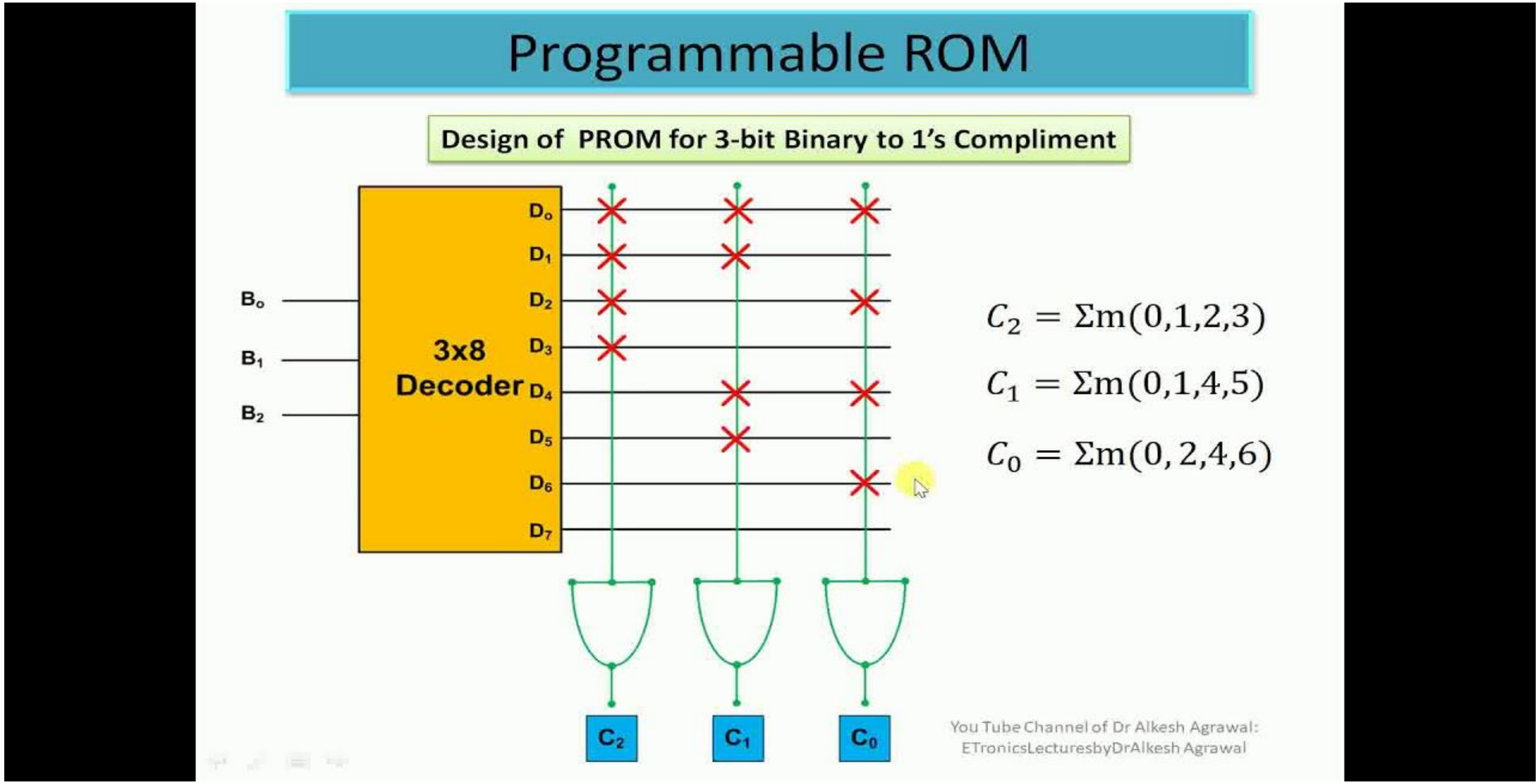


PROM

Programmable Read-Only Memory



PROM



Programmable ROM

Design of PROM for 3-bit Binary to 1's Compliment

$$C_2 = \Sigma m(0,1,2,3)$$

$$C_1 = \Sigma m(0,1,4,5)$$

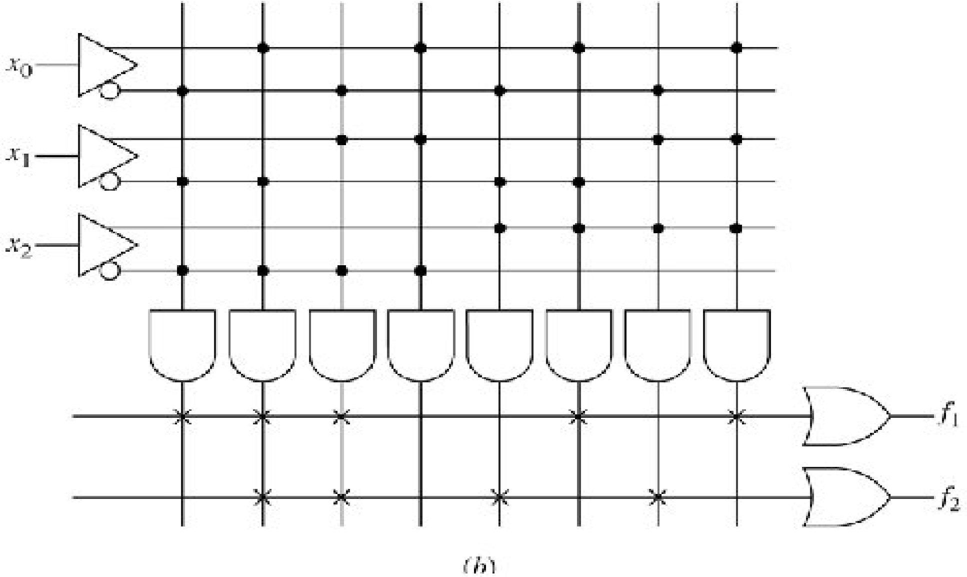
$$C_0 = \Sigma m(0, 2,4,6)$$

You Tube Channel of Dr Alkesh Agrawal:
ETronicsLecturesbyDrAlkesh Agrawal

Using a PROM for logic design

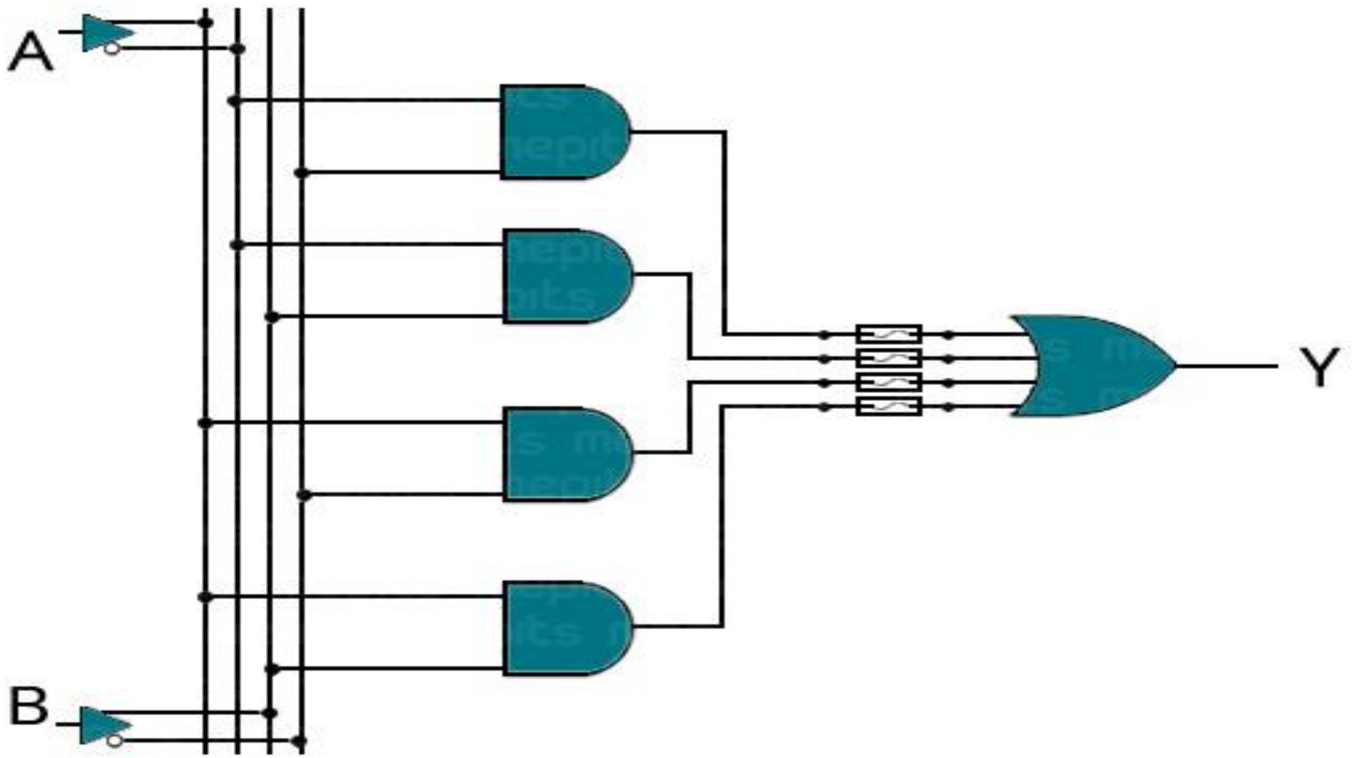
x_2	x_1	x_0	f_1	f_2
0	0	0	1	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	0
1	0	0	0	1
1	0	1	1	0
1	1	0	0	1
1	1	1	1	0

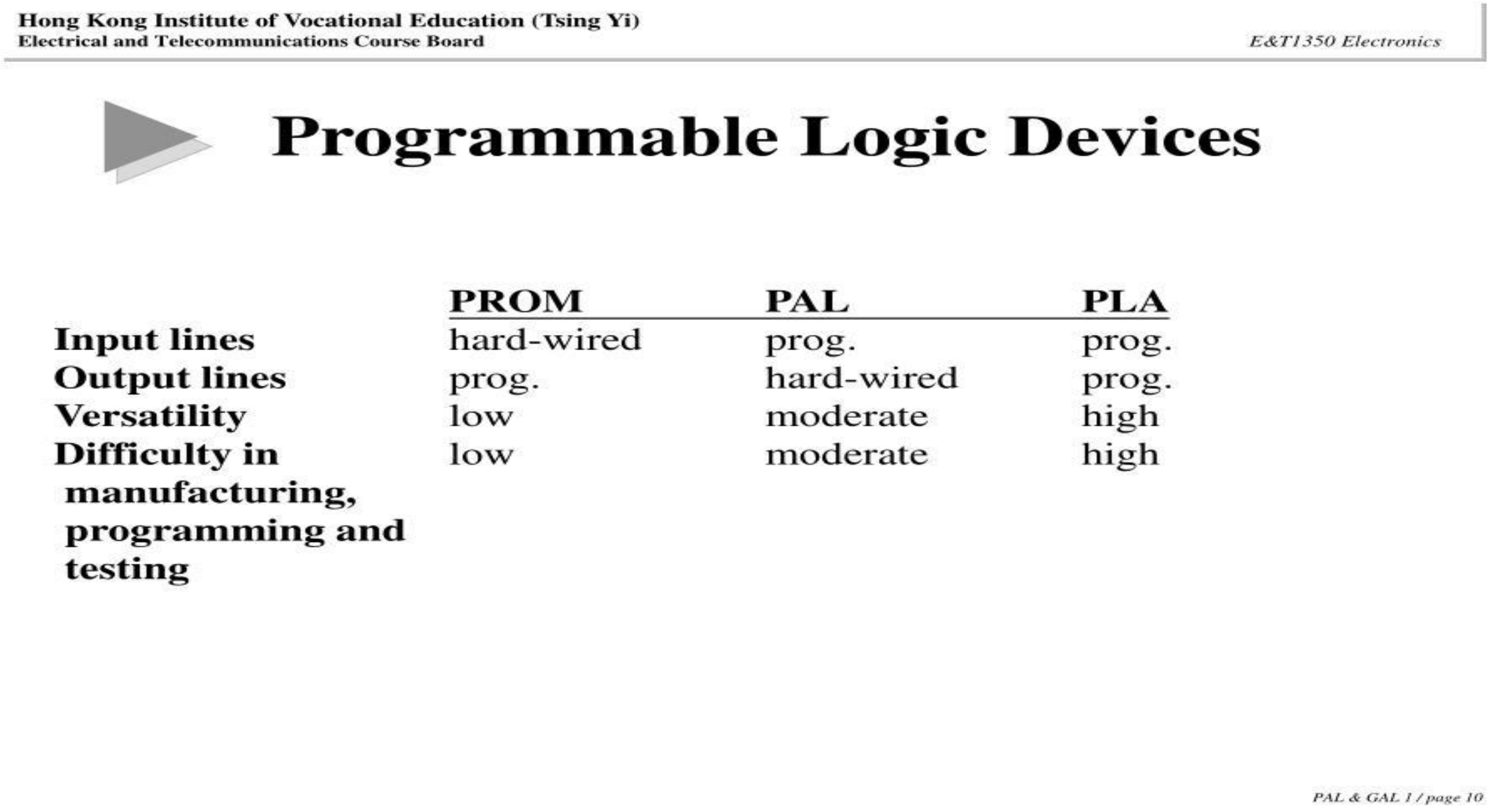
(a)



(a) Truth table. **(b) PROM realization.**

Programmable Read-Only Memory

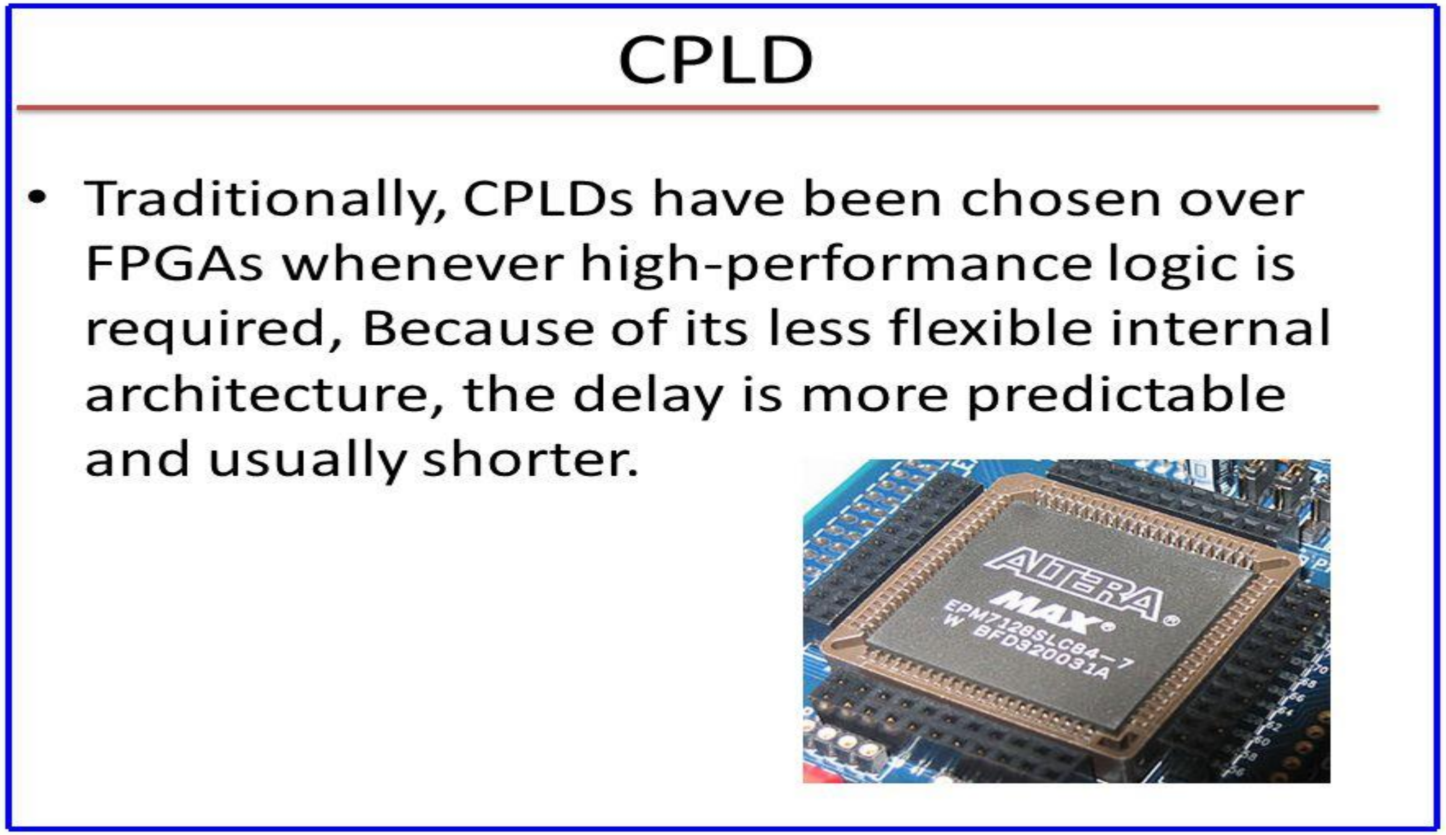




Programmable Logic Devices

	PROM	PAL	PLA
Input lines	hard-wired	prog.	prog.
Output lines	prog.	hard-wired	prog.
Versatility	low	moderate	high
Difficulty in manufacturing, programming and testing	low	moderate	high

Complex Programmable Logic Device



CPLD

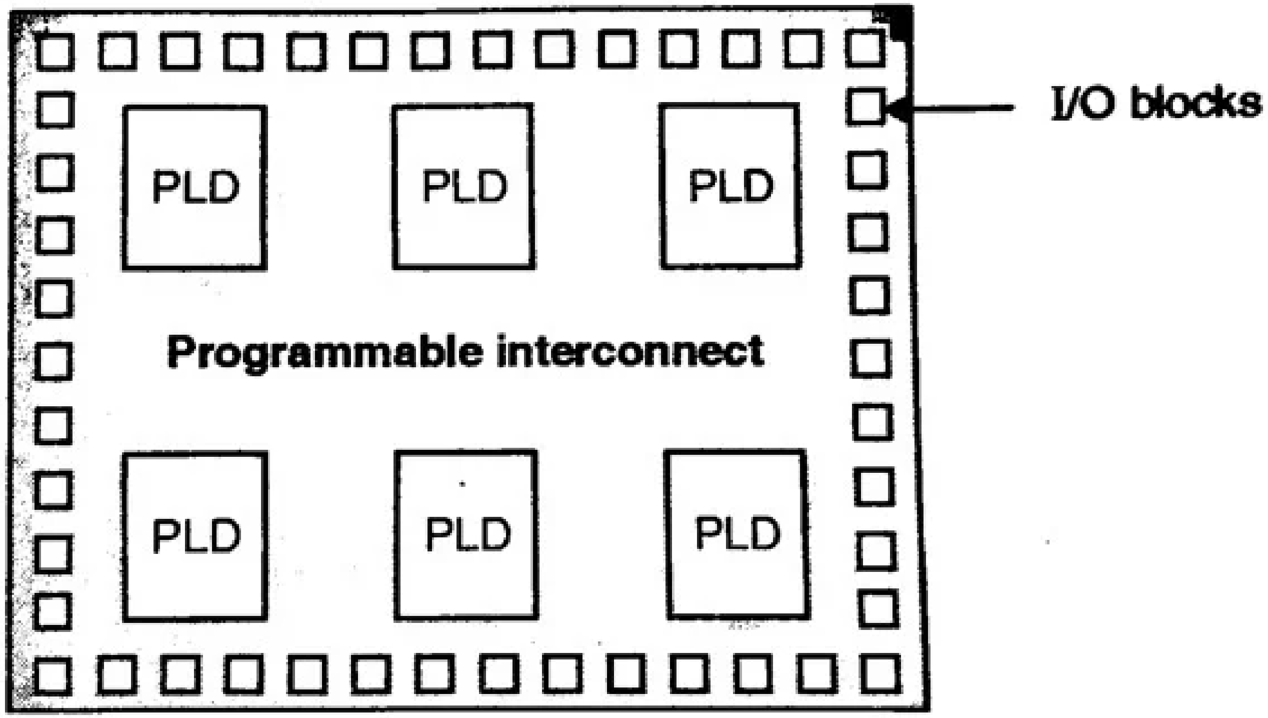
- Traditionally, CPLDs have been chosen over FPGAs whenever high-performance logic is required, Because of its less flexible internal architecture, the delay is more predictable and usually shorter.

ALTERA
MAX
EPM7128SLC84-7
W BFD320031A

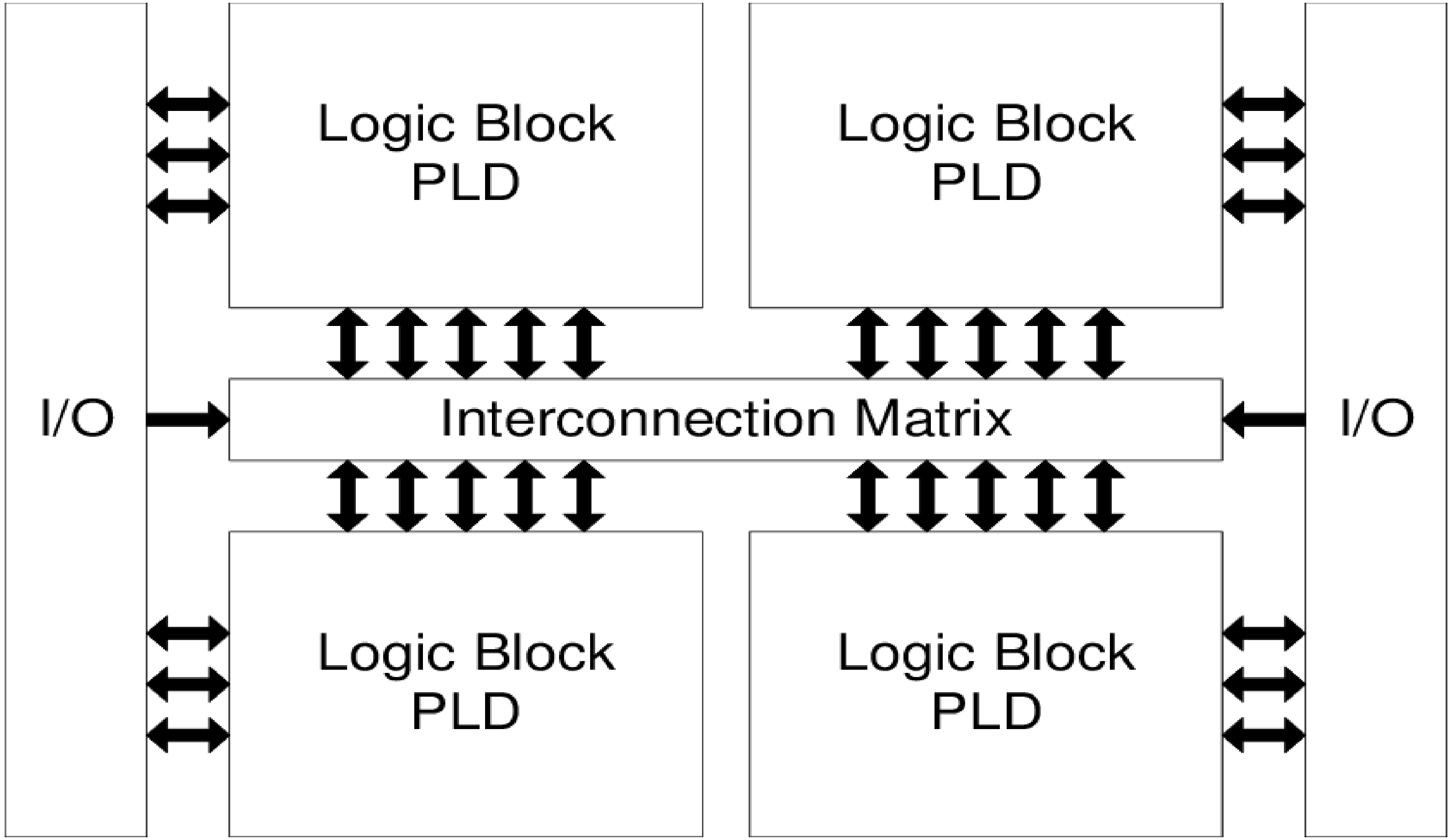
Complex Programmable Logic Device

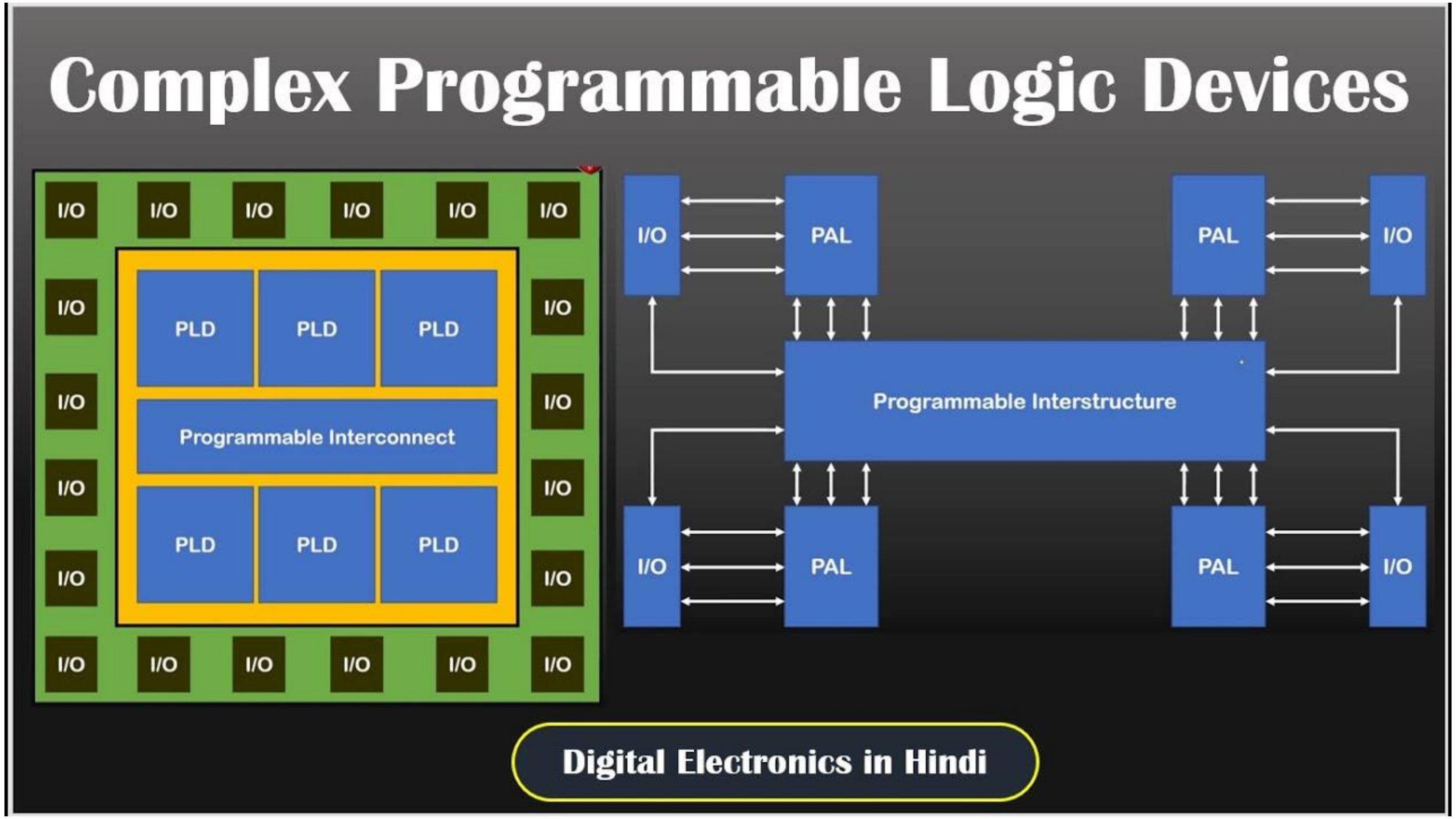
- Complex Programmable Logic Device (CPLD)
- CPLDs are denser than PALs and comprise of a large number of programmable logical elements. The interconnection between these macro cells is to be established by the user through the interconnecting network.
- Here sum-of-product establishing logical elements are combined together to form structures in order to reduce the number of input-output (IO) pins.
- This facilitates the implementation of more complex logic design with slightly worse propagation time when compared to that of PALs.
- These offer predictable timing characteristics making them most suitable for critical control applications with high performance.
- CPLDs are preferred to implement combinational logic based designs.

Complex Programmable Logic Device



Complex Programmable Logic Device

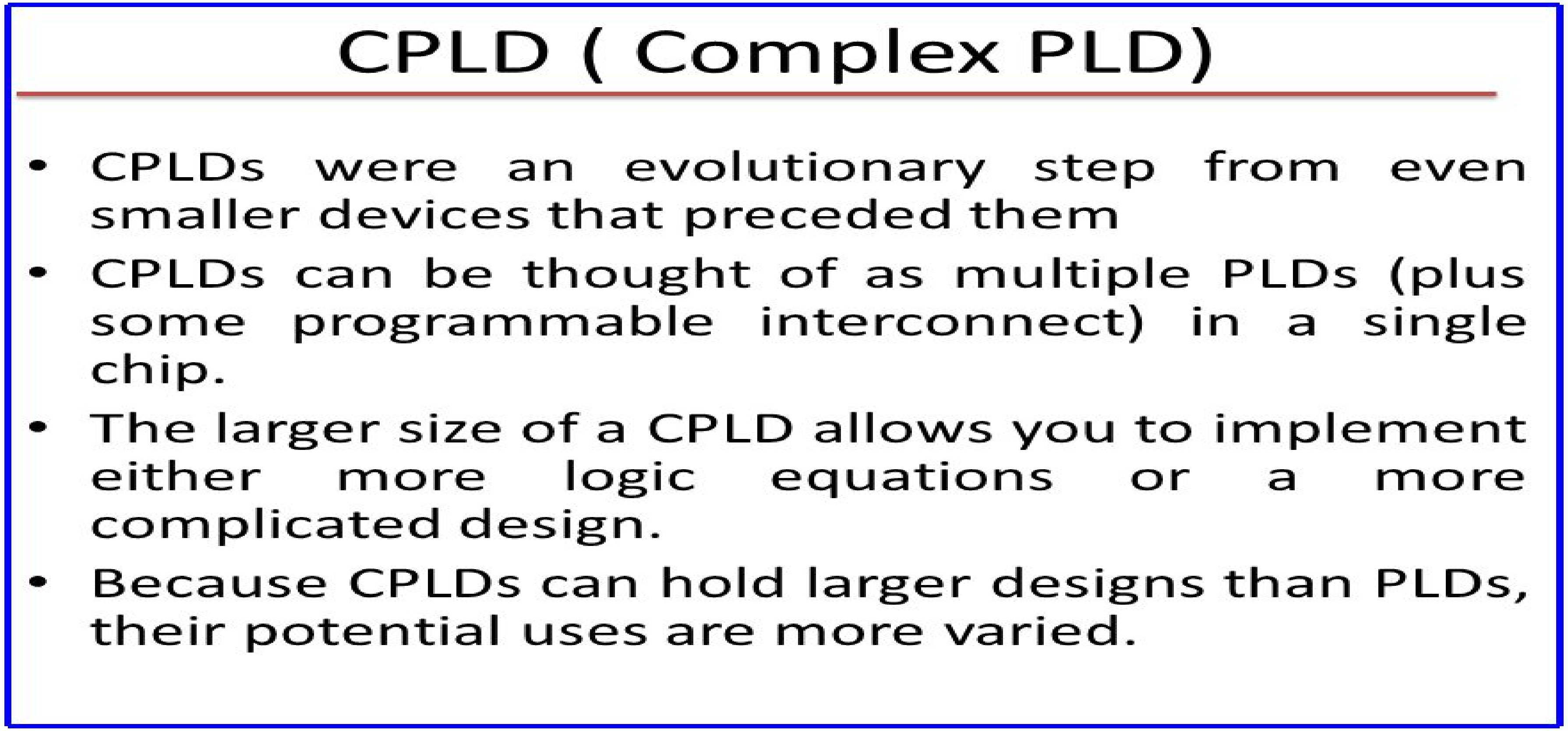




Complex Programmable Logic Devices

Digital Electronics in Hindi

Complex Programmable Logic Device

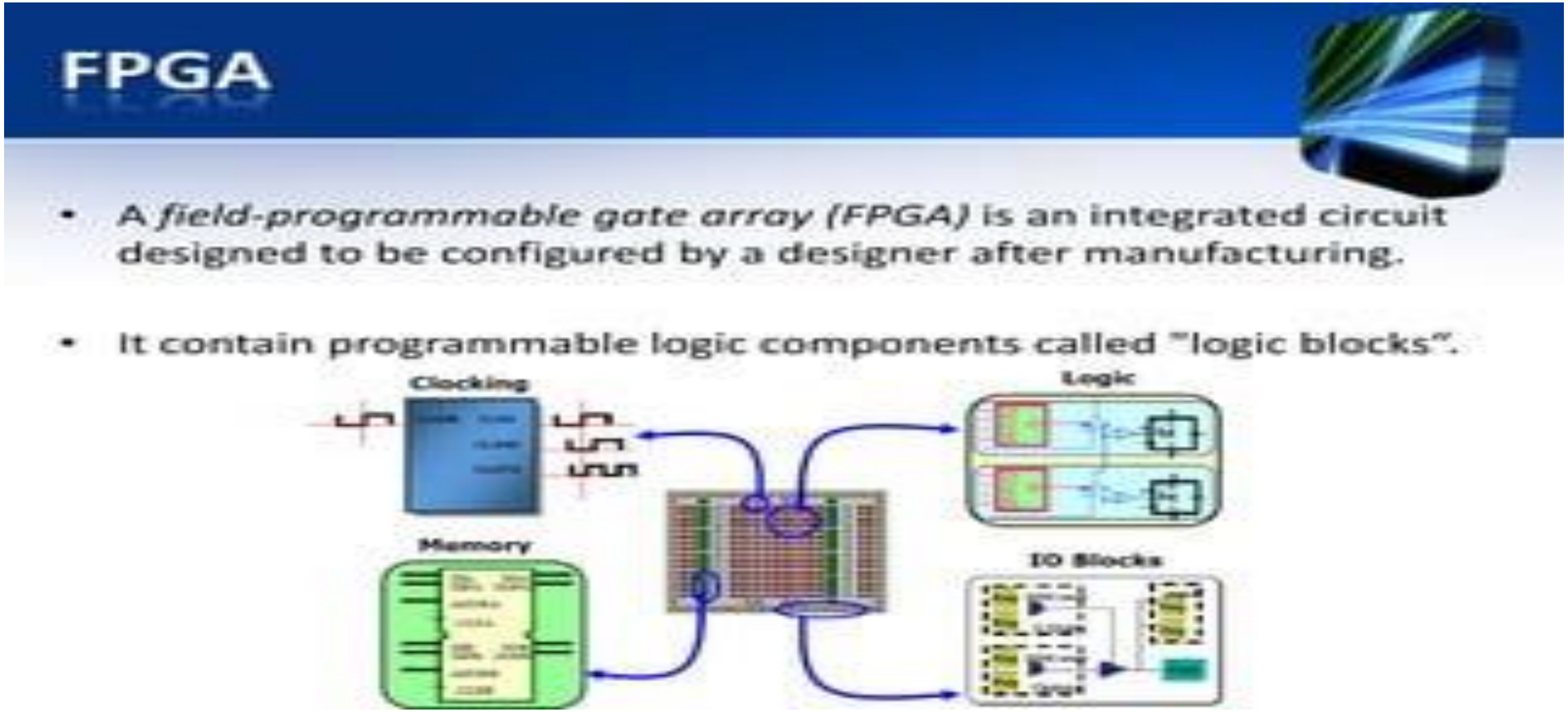


CPLD (Complex PLD)

- CPLDs were an evolutionary step from even smaller devices that preceded them
- CPLDs can be thought of as multiple PLDs (plus some programmable interconnect) in a single chip.
- The larger size of a CPLD allows you to implement either more logic equations or a more complicated design.
- Because CPLDs can hold larger designs than PLDs, their potential uses are more varied.

Field Programmable Gate Array

Field Programmable Gate Array

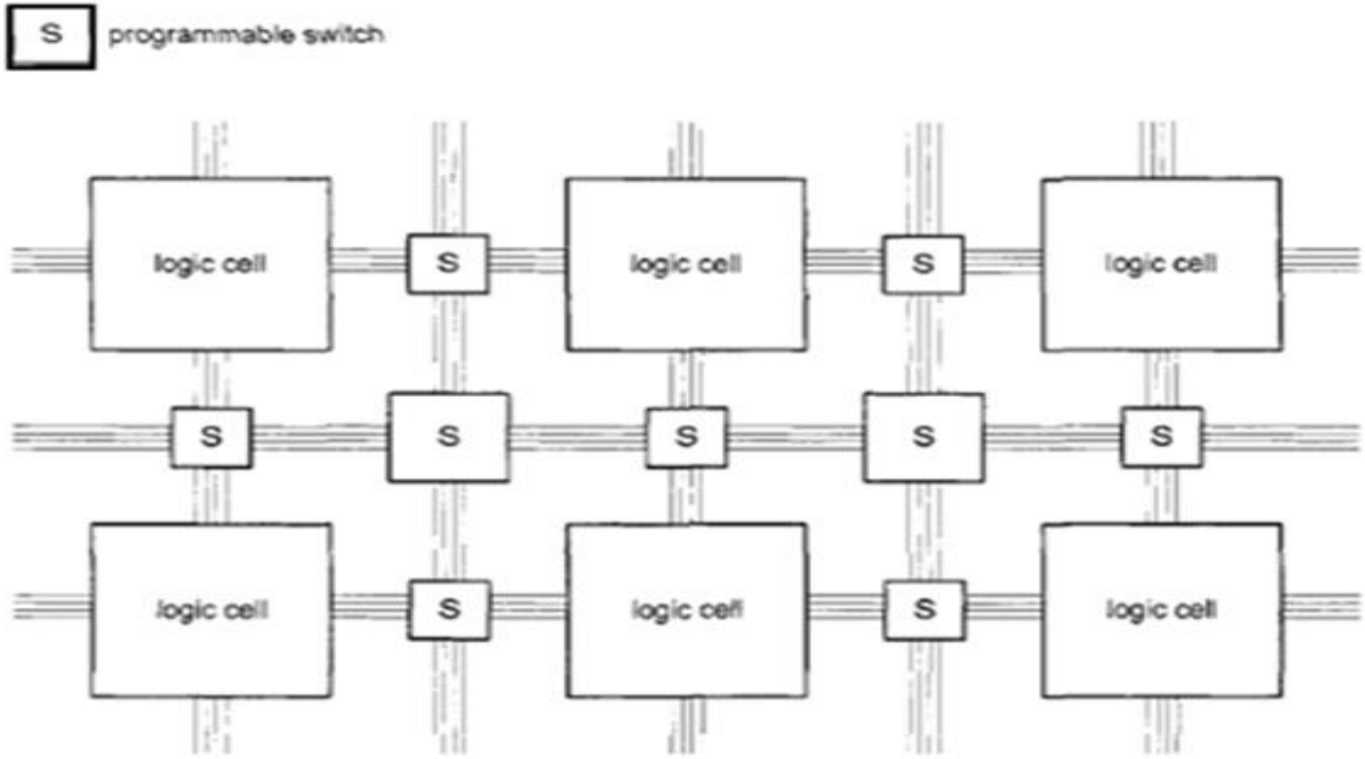


FPGA

- A *field-programmable gate array (FPGA)* is an integrated circuit designed to be configured by a designer after manufacturing.

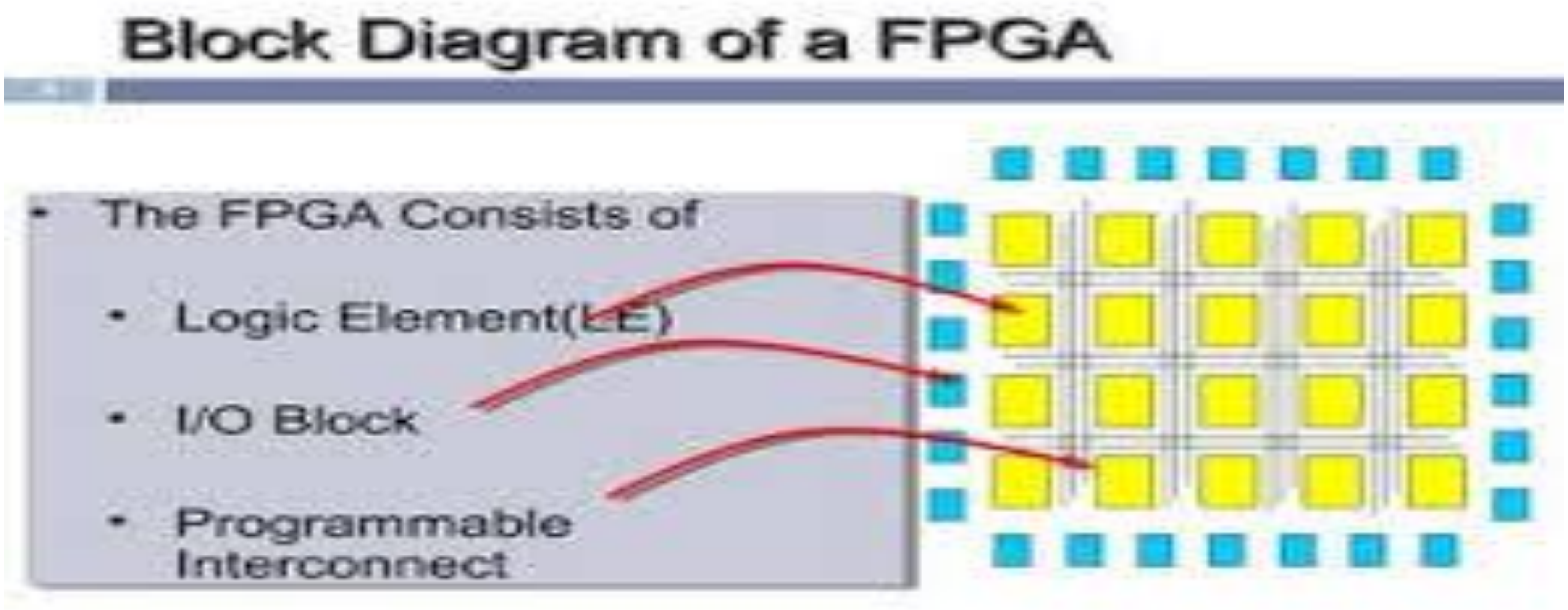
- It contain programmable logic components called "logic blocks".

Field Programmable Gate Array



Conceptual structure of an FPGA device.

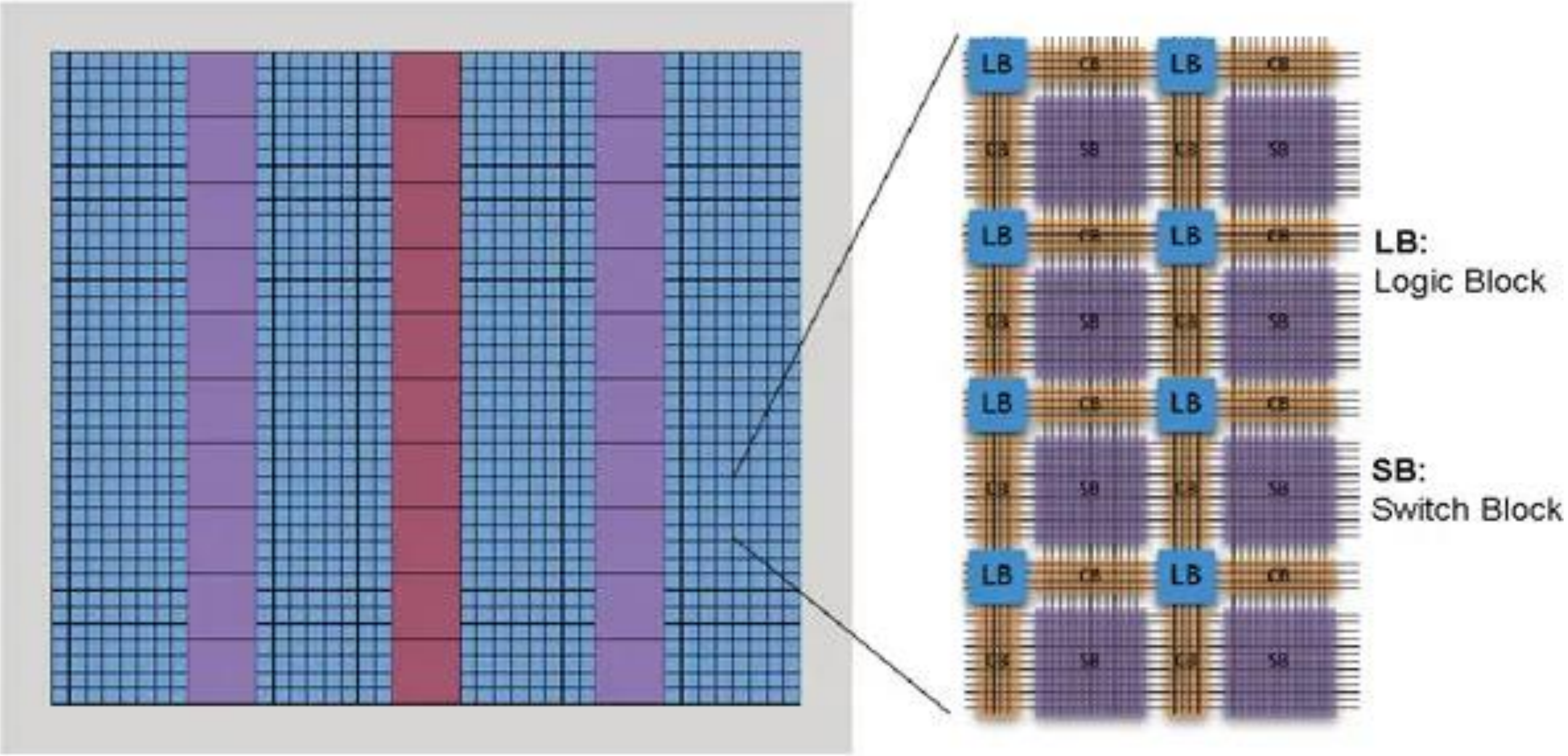
Field Programmable Gate Array



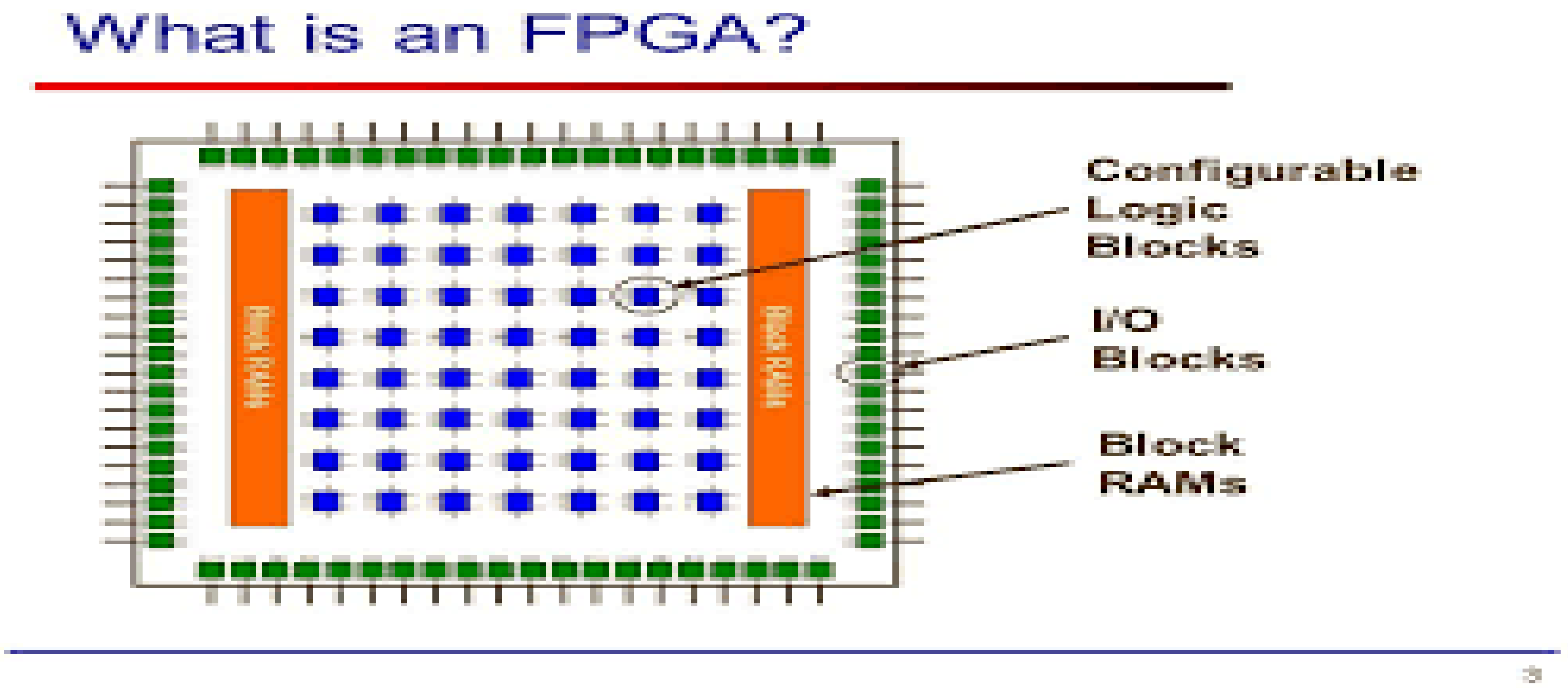
Block Diagram of a FPGA

- The FPGA Consists of
  - Logic Element(LE)
  - I/O Block
  - Programmable Interconnect

Field Programmable Gate Array

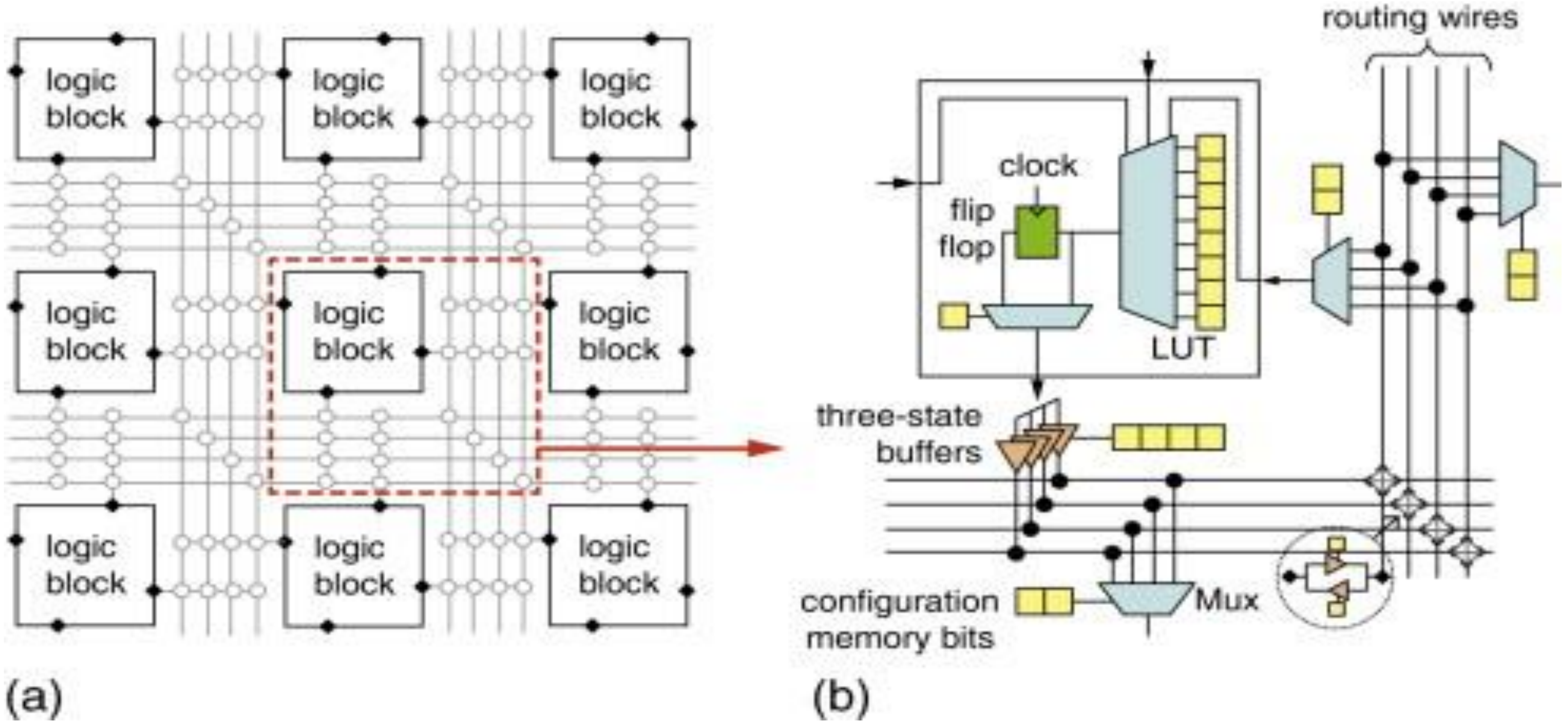


Field Programmable Gate Array

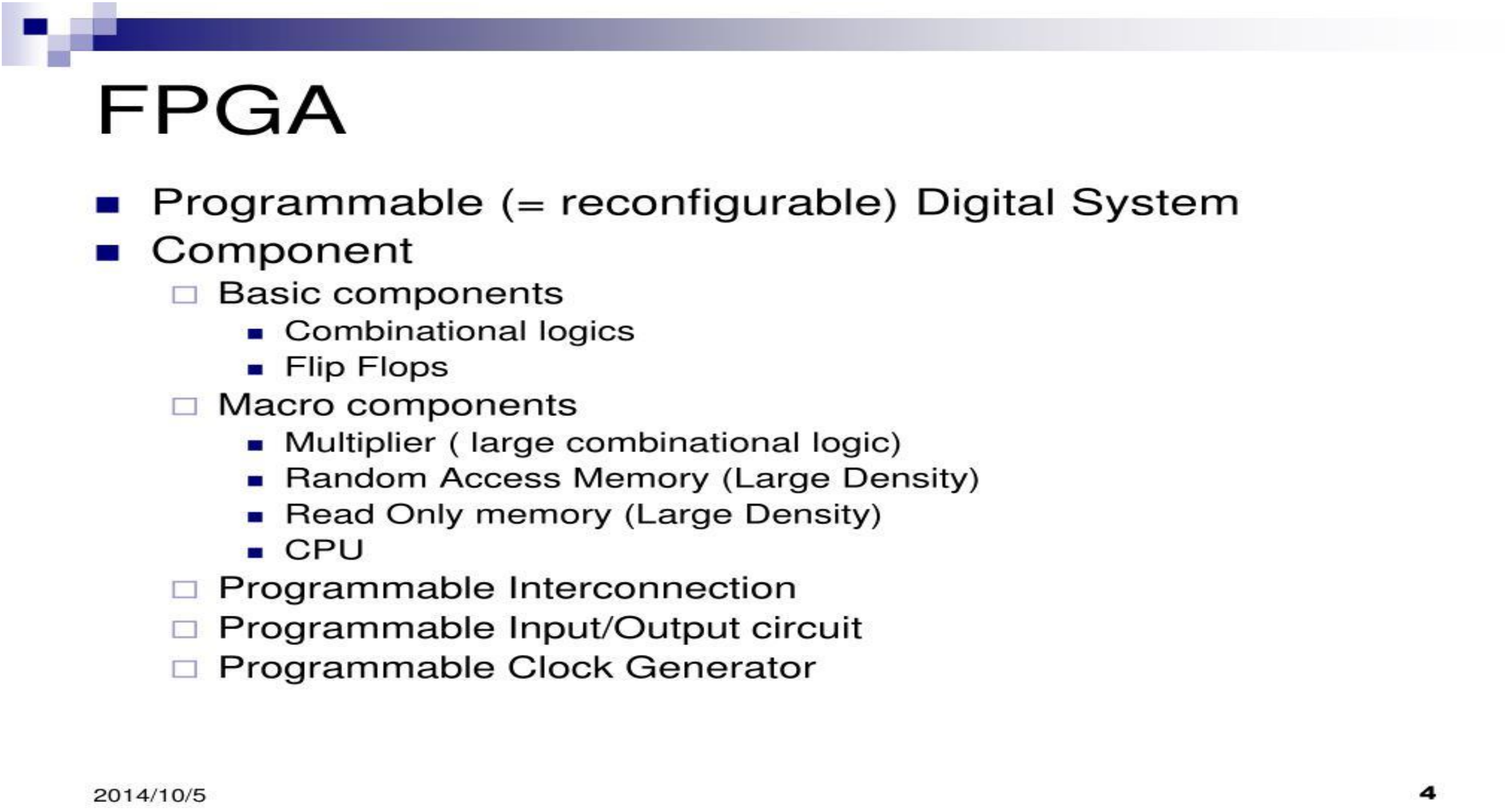


What is an FPGA?

Field Programmable Gate Array



Field Programmable Gate Array

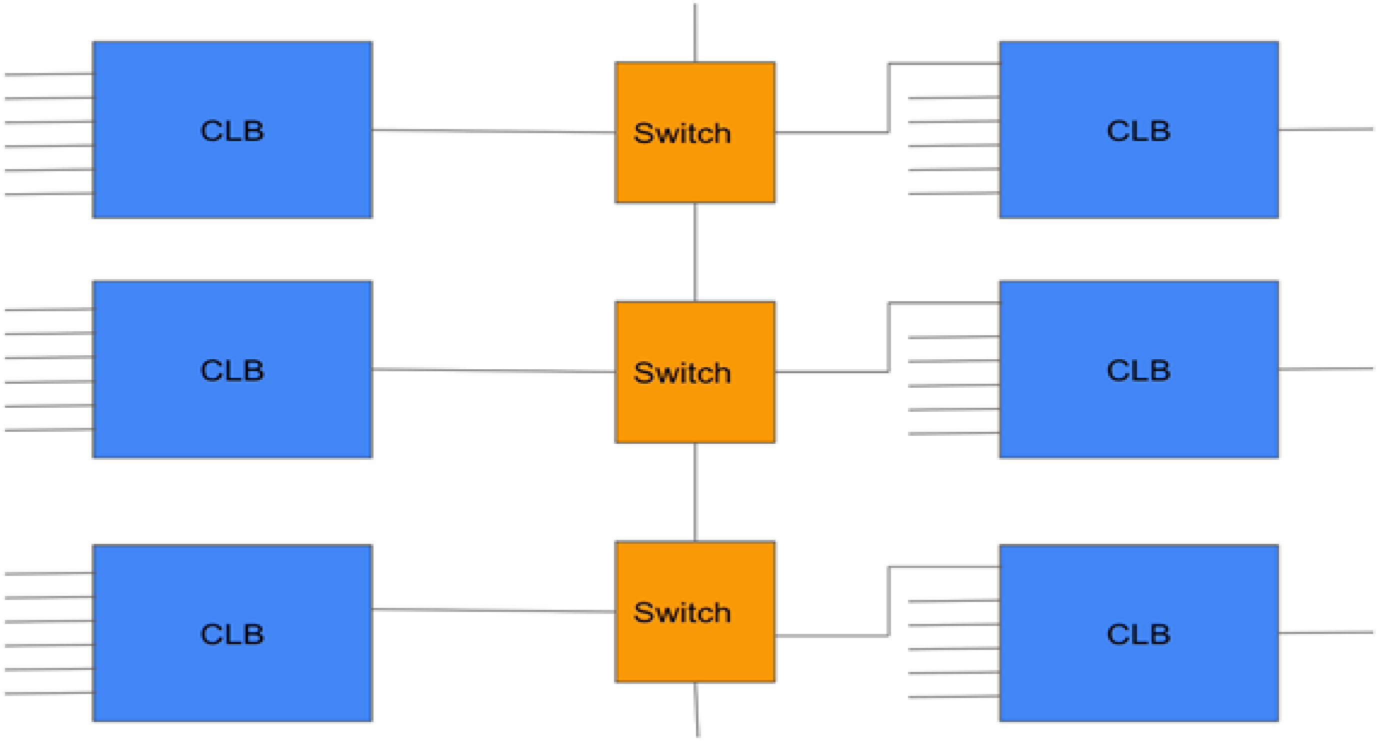


FPGA

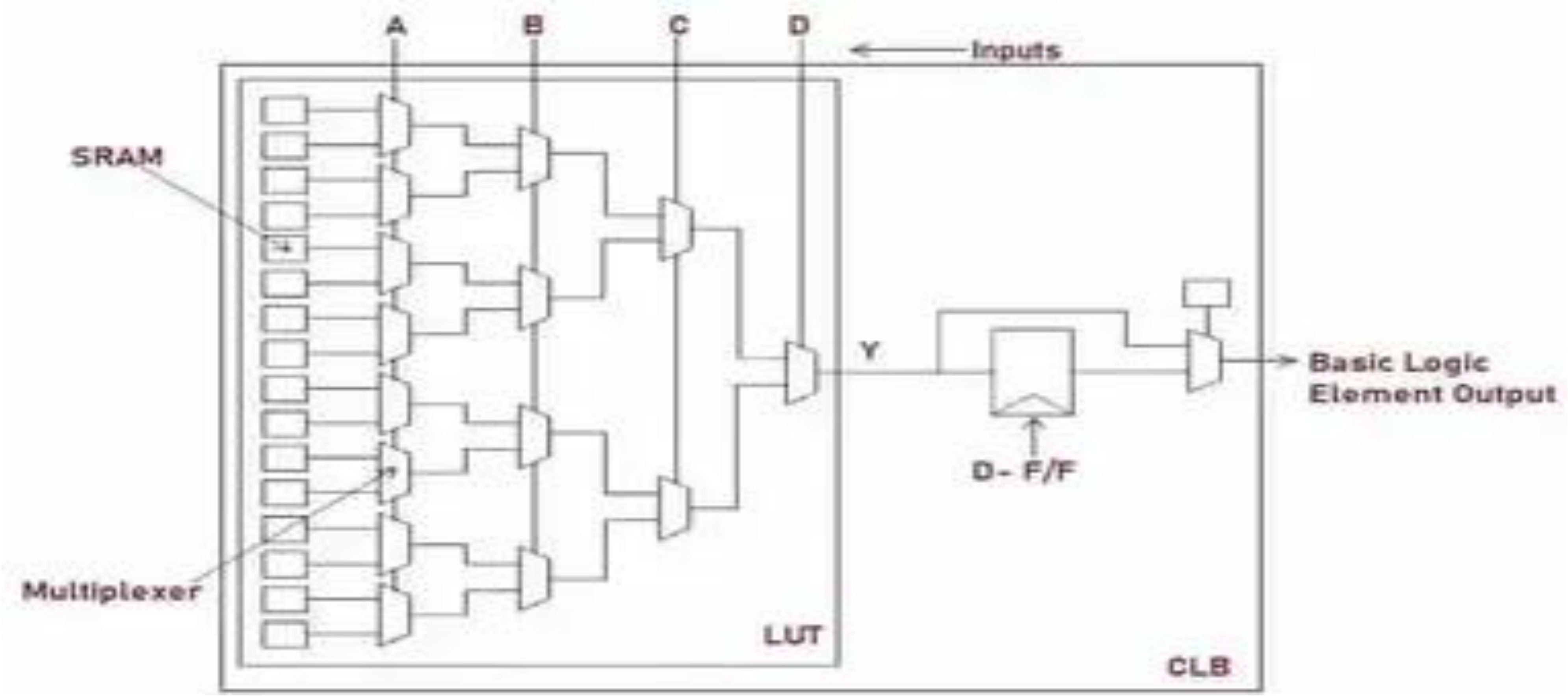
- Programmable (= reconfigurable) Digital System
- Component
  - □ Basic components
    - Combinational logics
    - Flip Flops
  - □ Macro components
    - Multiplier (large combinational logic)
    - Random Access Memory (Large Density)
    - Read Only memory (Large Density)
    - CPU
  - □ Programmable Interconnection
  - □ Programmable Input/Output circuit
  - □ Programmable Clock Generator

2014/10/5 4

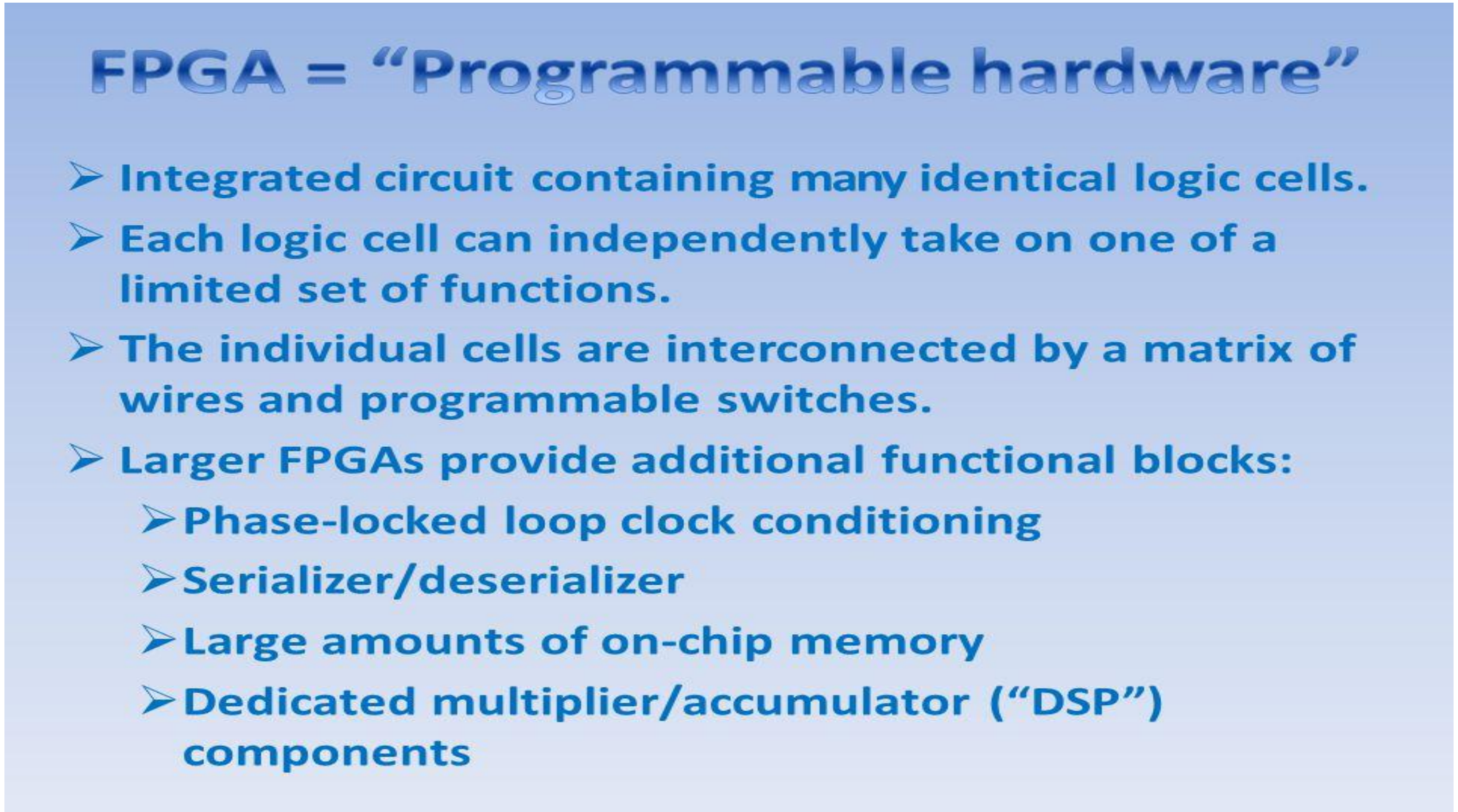
Field Programmable Gate Array



Field Programmable Gate Array



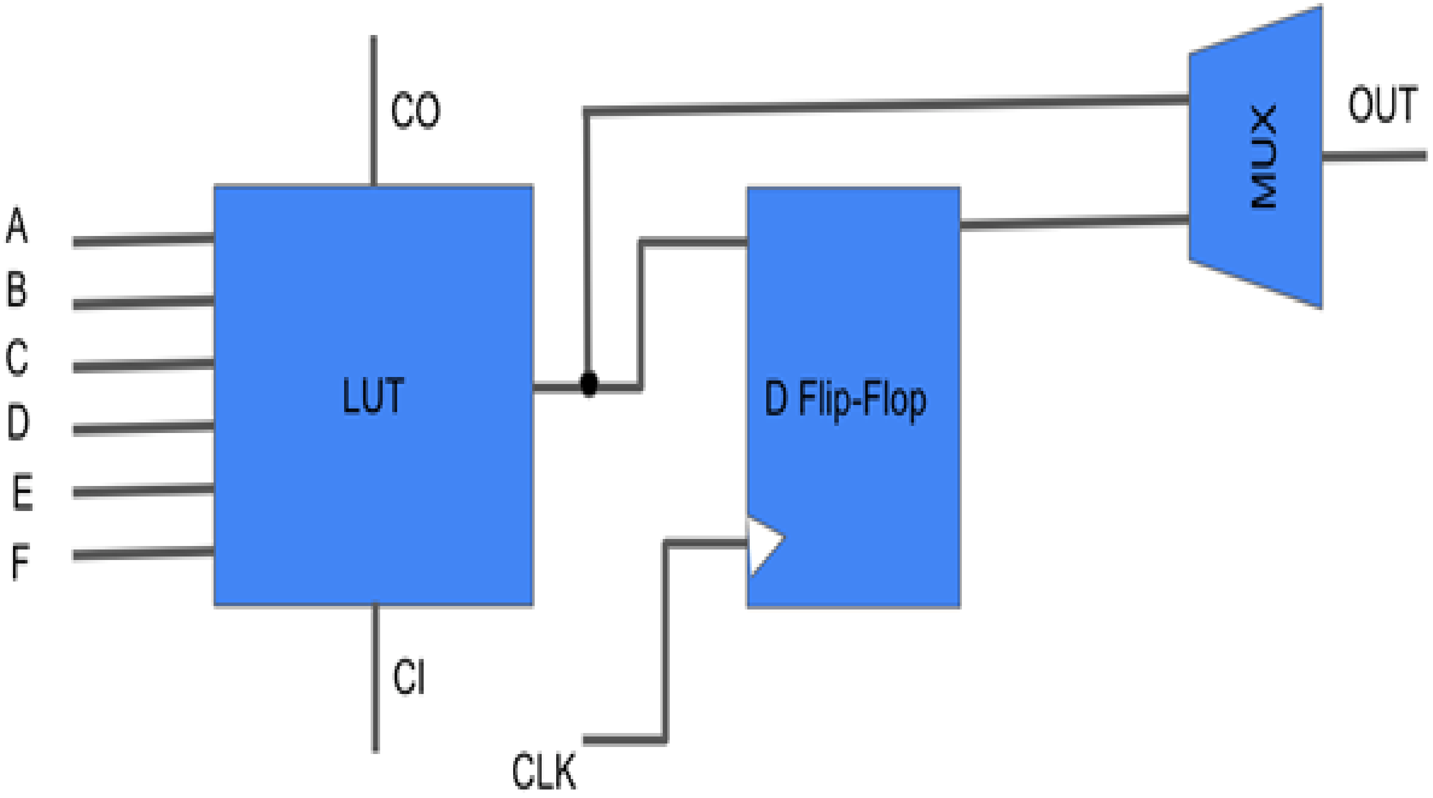
Field Programmable Gate Array



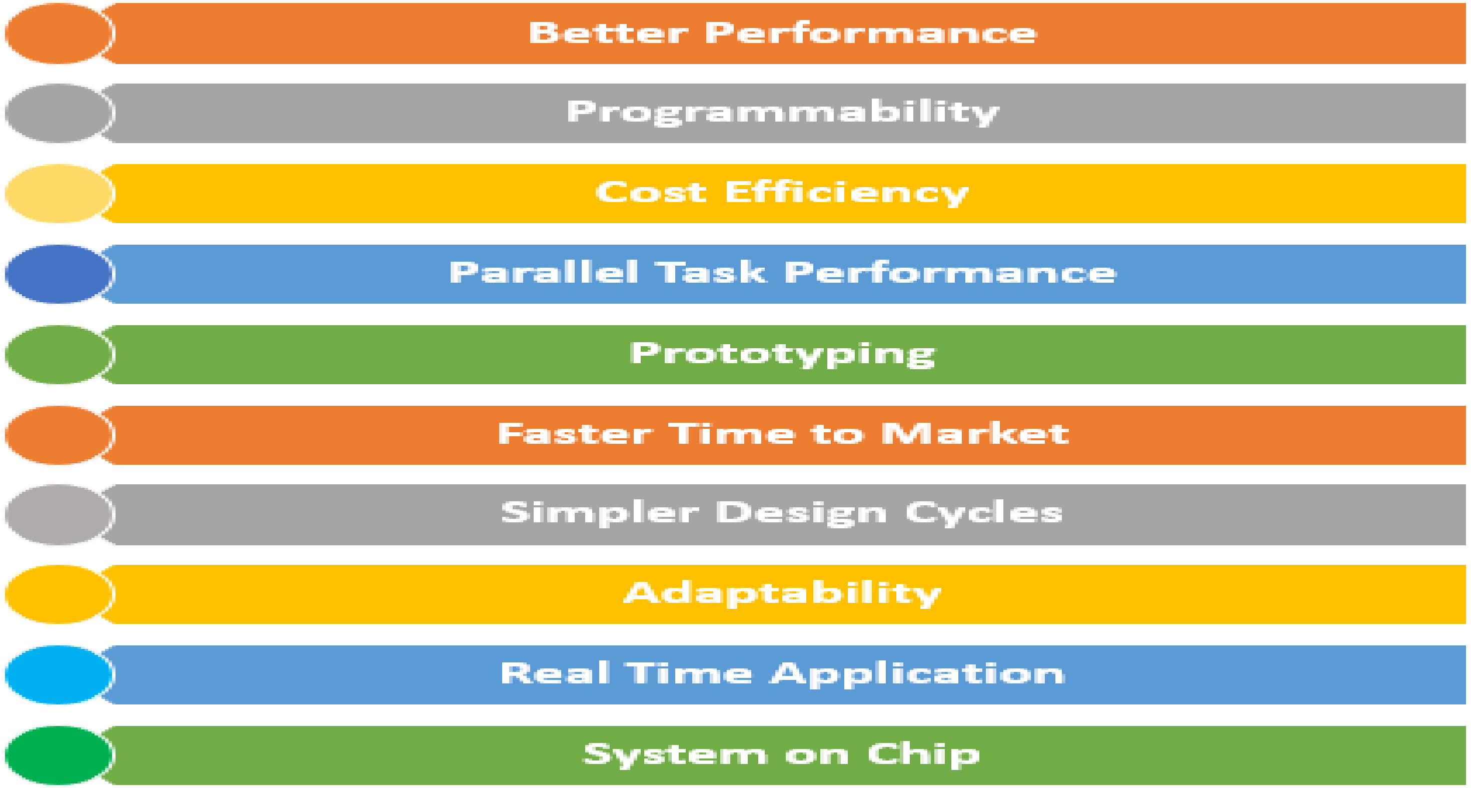
FPGA = “Programmable hardware”

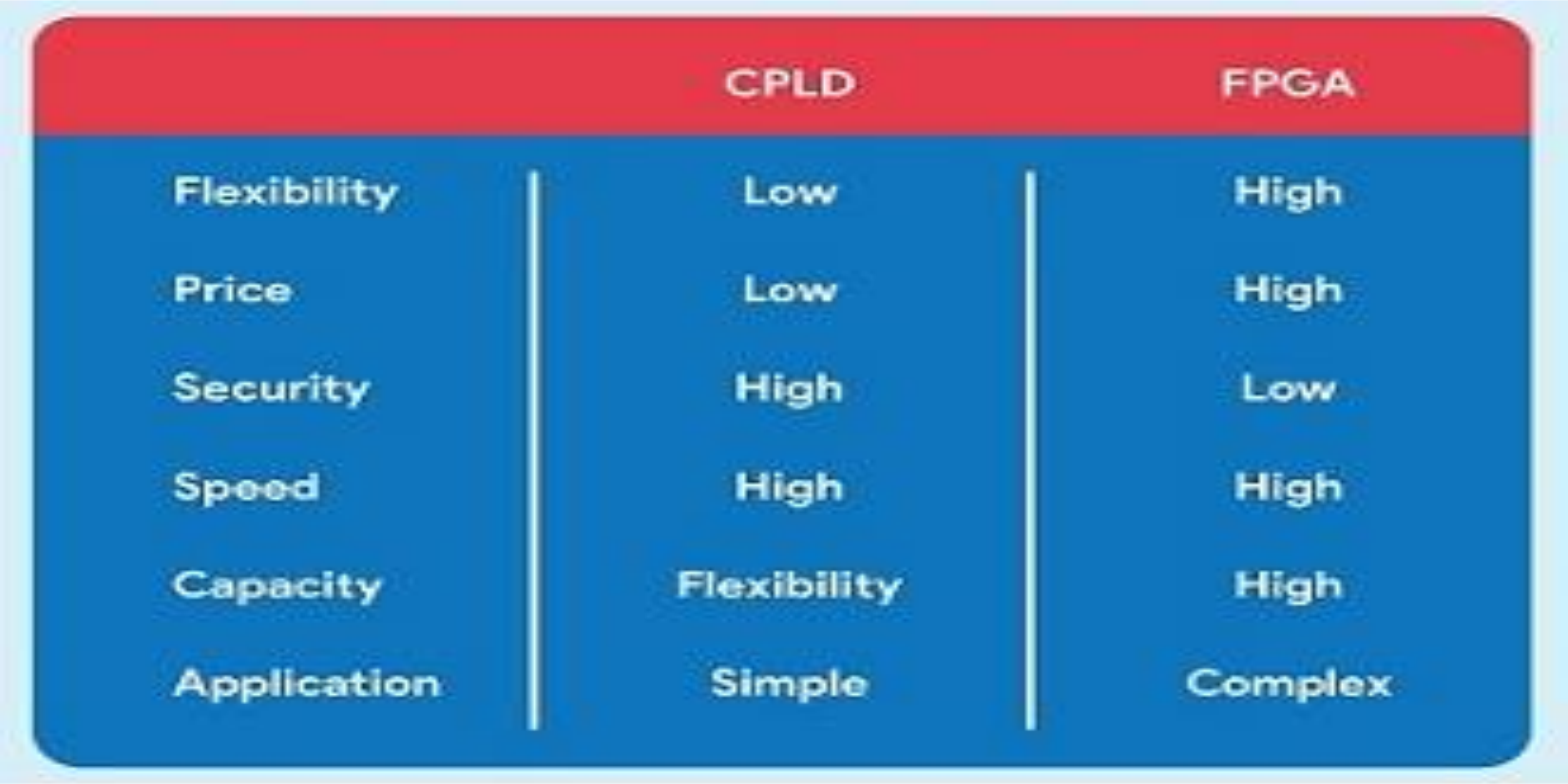
- ➢ Integrated circuit containing many identical logic cells.
- ➢ Each logic cell can independently take on one of a limited set of functions.
- ➢ The individual cells are interconnected by a matrix of wires and programmable switches.
- ➢ Larger FPGAs provide additional functional blocks:
  - ➢ Phase-locked loop clock conditioning
  - ➢ Serializer/deserializer
  - ➢ Large amounts of on-chip memory
  - ➢ Dedicated multiplier/accumulator (“DSP”) components

Field Programmable Gate Array



Top 10 FPGA Advantages



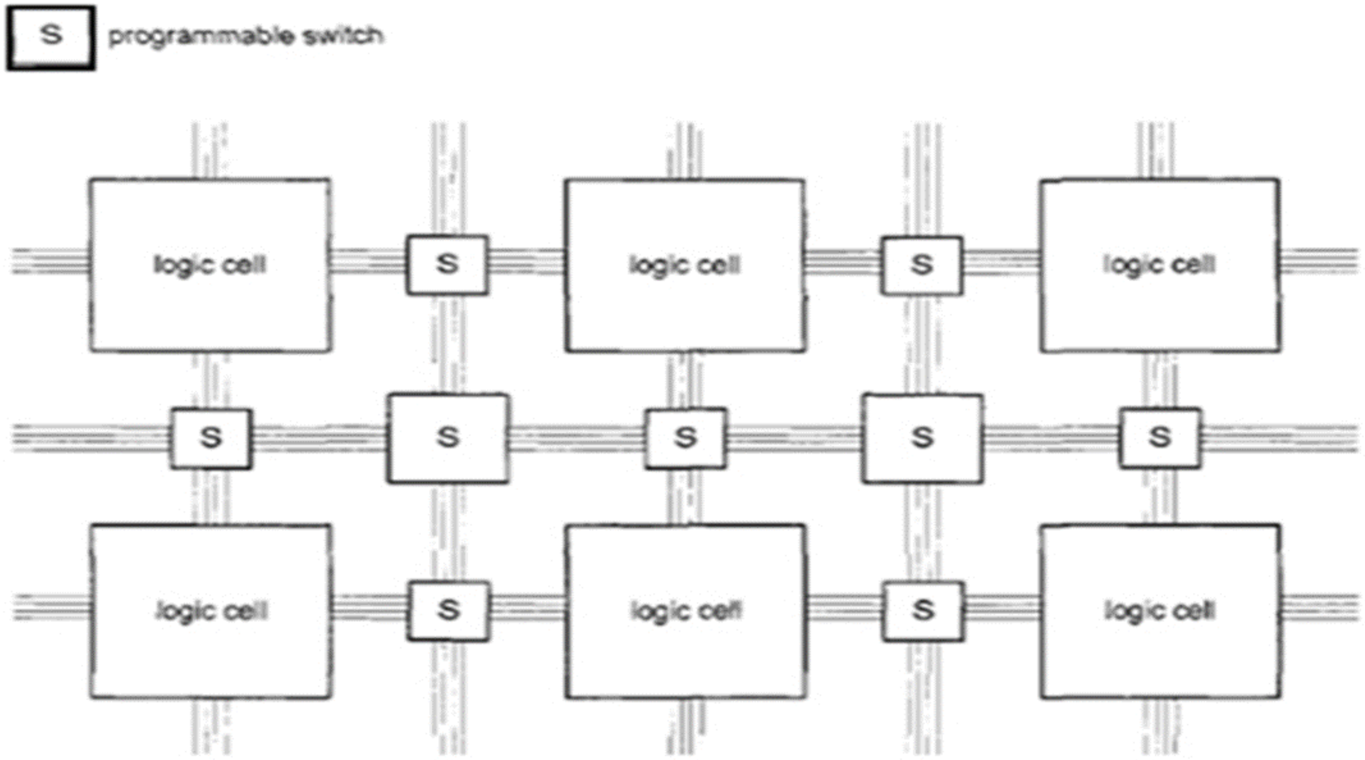


	CPLD	FPGA
Flexibility	Low	High
Price	Low	High
Security	High	Low
Speed	High	High
Capacity	Flexibility	High
Application	Simple	Complex

FPGA

An afield-programmable gate array (FPGA) is a logic device that contains a two-dimensional array of generic logic cells and programmable switches. The conceptual structure of an FPGA device is shown in Figure .1. *A* logic cell can be configured (i.e., *programmed)* to perform a simple function, and a programmable switch can be customized to provide interconnections among the logic cells.

A custom design can be implemented by specifying the function of each logic cell and selectively setting the connection of each programmable switch. Once the design and synthesis are completed, we can use a simple adaptor cable to download the desired logic cell and switch configuration to the FPGA device and obtain the custom circuit. Since this process can be done "in the field" rather than "in a fabrication facility (fab)," the device is known as field programmable.



Conceptual structure of an FPGA device.

Figure -1-

LUT based logic cell

A logic cell usually contains a small configurable combinational circuit with a D-type flip-flop (D FF). The most common method to implement a configurable combinational circuit is a look-up table (LUT). An n-input LUT can be considered as a small 2^n memory. By properly writing the memory content, we can use an LUT to implement any n-input combinational function. The conceptual diagram of a three input LUT-based logic cell is shown in Figure

2.(a). An example of a three-input LUT implementation of *a xor* b xor c is shown in Figure 2.(b). Note that the output of the LUT can be used directly or stored to the D FF. The latter can be used to implement sequential circuits.

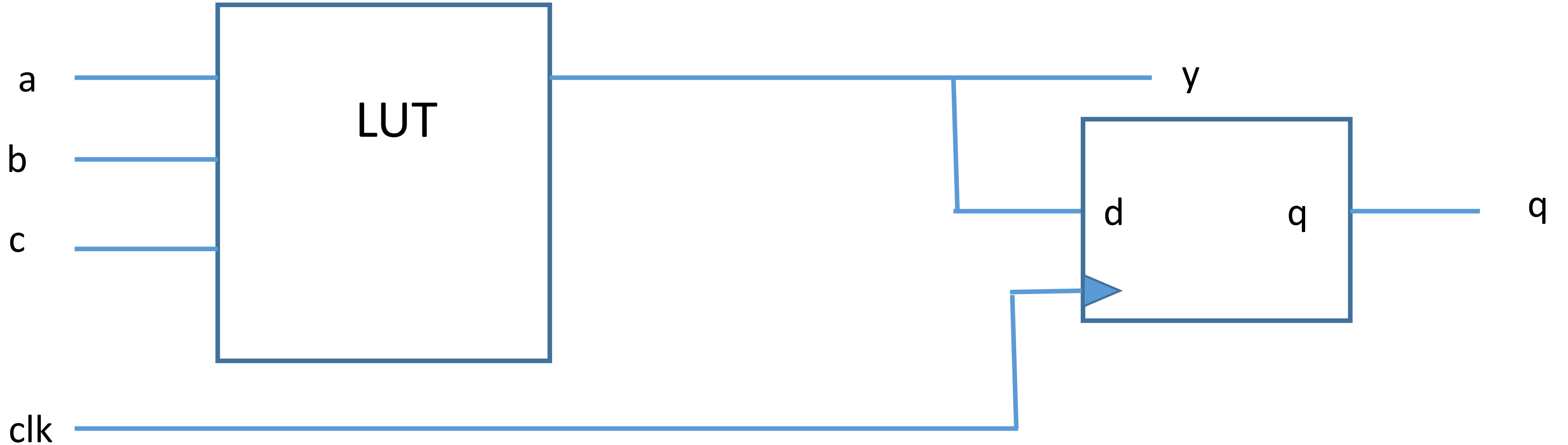
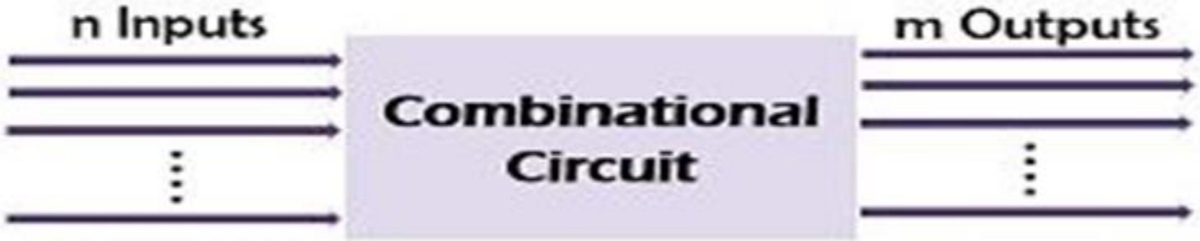


Figure 2-a Conceptual Diagram

A	b	c	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Figure 2-b Example Table

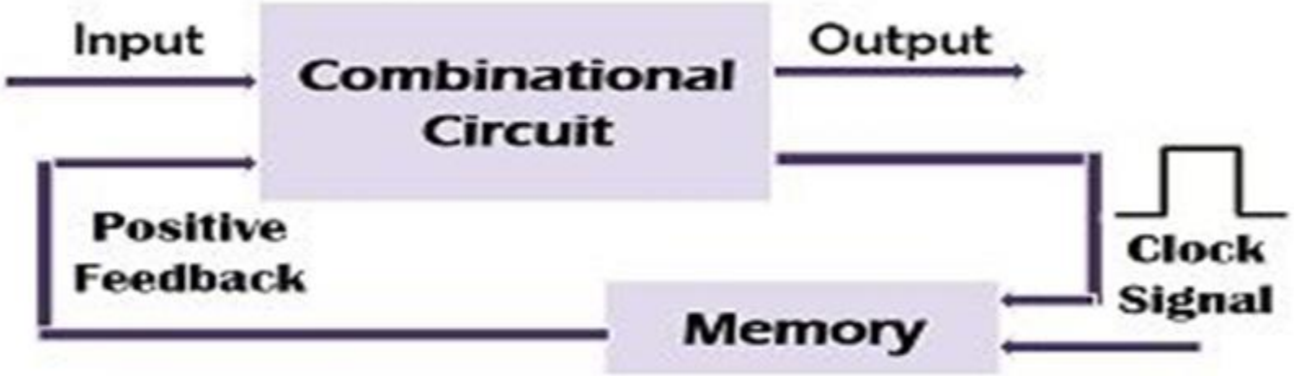
Figure 2- Three input LUT based logic circuit

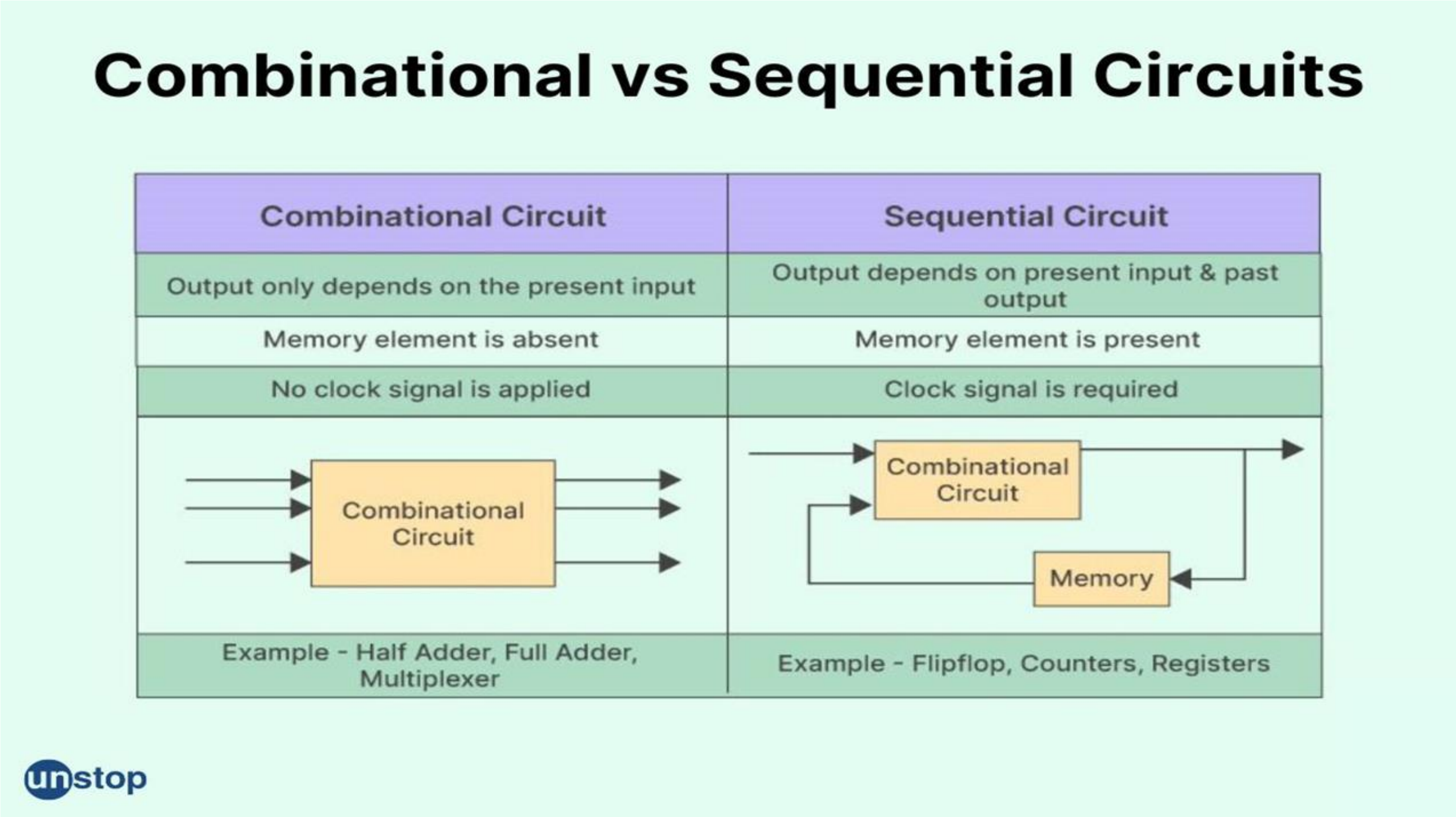


Combinational Circuit

Vs

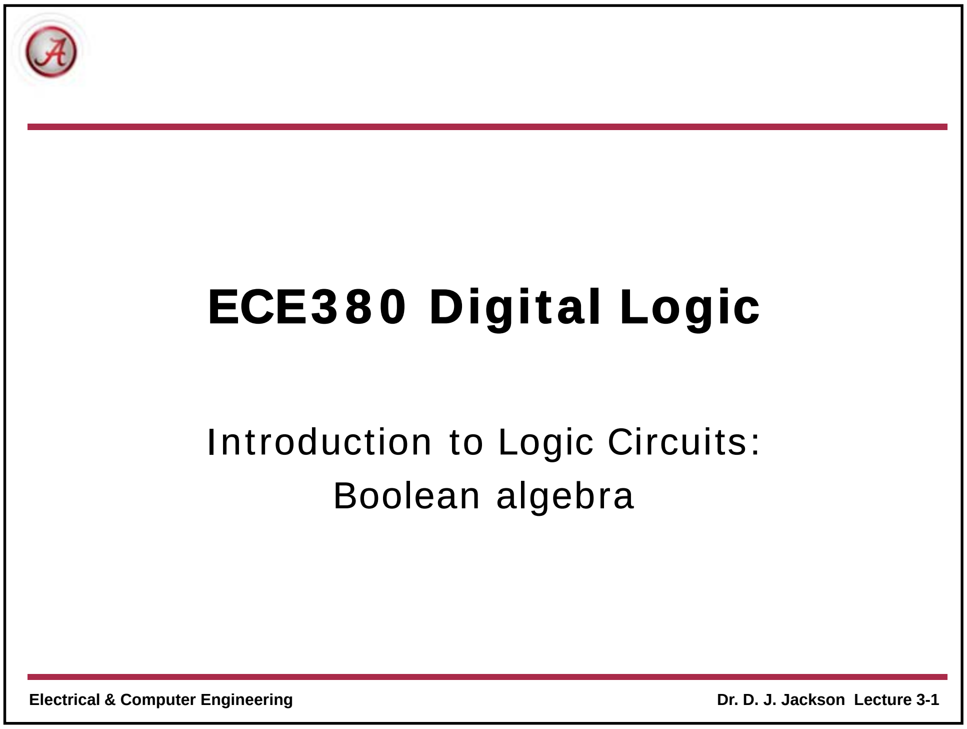
Sequential Circuit





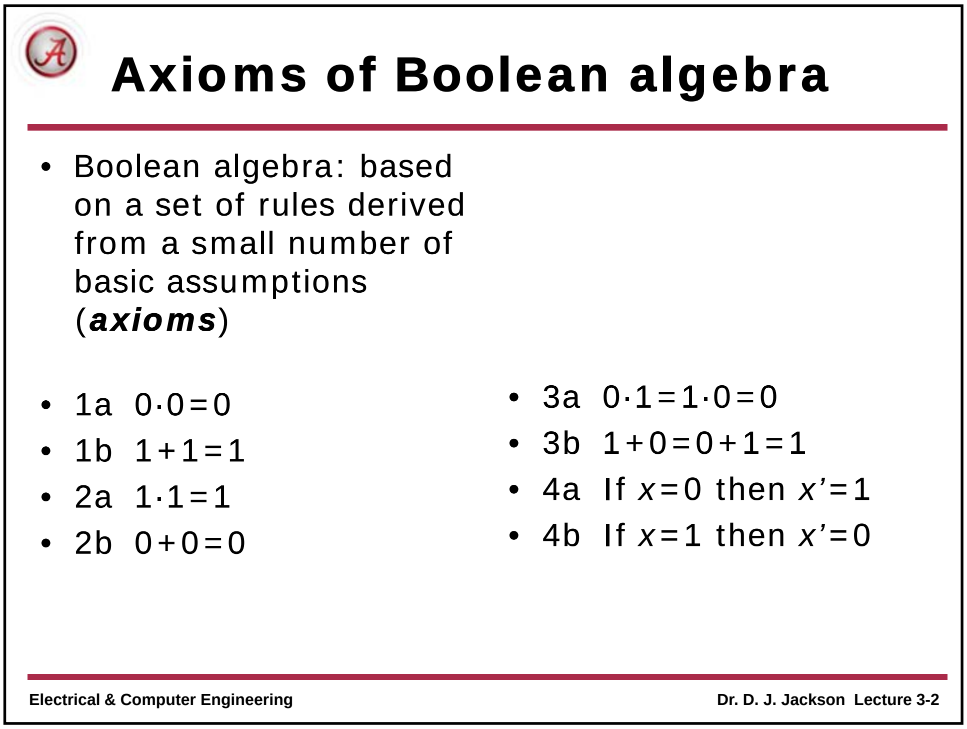
Combinational vs Sequential Circuits

Combinational Circuit	Sequential Circuit
Output only depends on the present input	Output depends on present input & past output
Memory element is absent	Memory element is present
No clock signal is applied	Clock signal is required
Combinational Circuit	Combinational Circuit Memory
Example - Half Adder, Full Adder, Multiplexer	Example - Flipflop, Counters, Registers



ECE380 Digital Logic

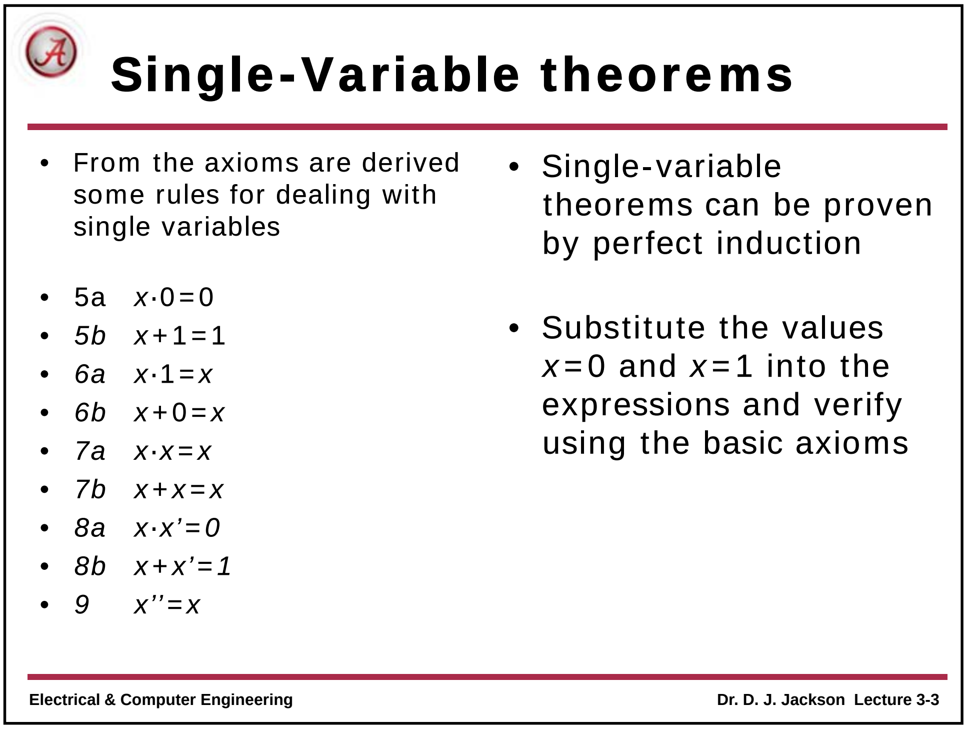
Introduction to Logic Circuits: Boolean algebra



Axioms of Boolean algebra

- Boolean algebra: based on a set of rules derived from a small number of basic assumptions (***axioms***)

- 1a $0 \cdot 0 = 0$
- 1b $1 + 1 = 1$
- 2a $1 \cdot 1 = 1$
- 2b $0 + 0 = 0$
- 3a $0 \cdot 1 = 1 \cdot 0 = 0$
- 3b $1 + 0 = 0 + 1 = 1$
- 4a If $x = 0$ then $x' = 1$
- 4b If $x = 1$ then $x' = 0$



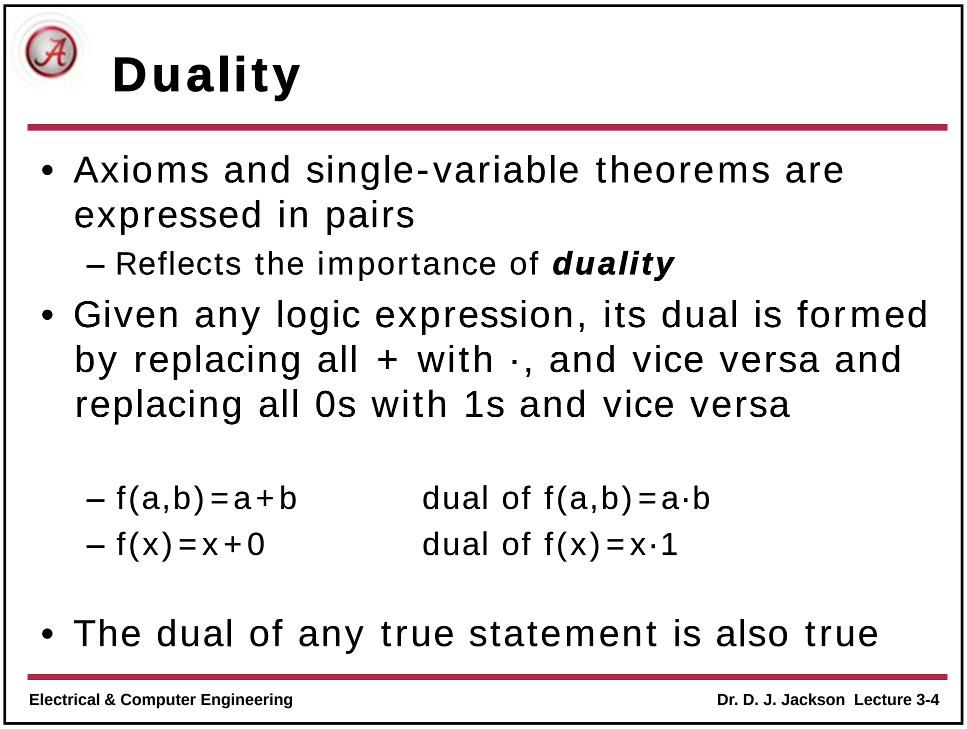
Single-Variable theorems

- From the axioms are derived some rules for dealing with single variables

- 5a $x \cdot 0 = 0$
- *5b* $x + 1 = 1$
- *6a* $x \cdot 1 = x$
- *6b* $x + 0 = x$
- *7a* $x \cdot x = x$
- *7b* $x + x = x$
- *8a* $x \cdot x' = 0$
- *8b* $x + x' = 1$
- *9* $x'' = x$

- Single-variable theorems can be proven by perfect induction

- Substitute the values $x=0$ and $x=1$ into the expressions and verify using the basic axioms

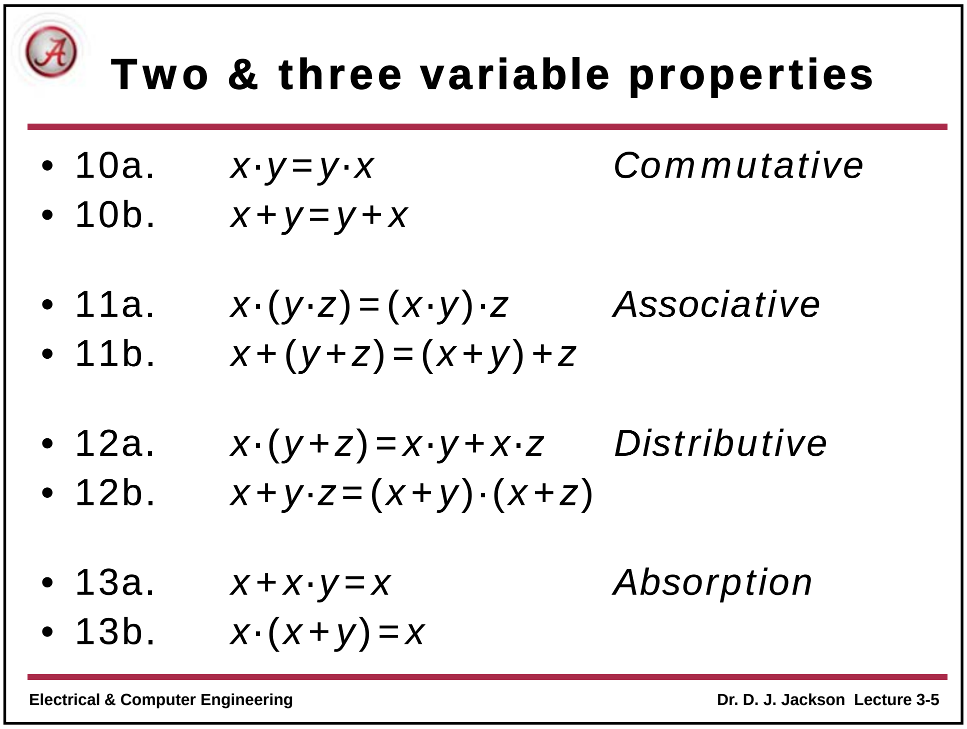


Duality

- Axioms and single-variable theorems are expressed in pairs
  - Reflects the importance of ***duality***
- Given any logic expression, its dual is formed by replacing all + with ·, and vice versa and replacing all 0s with 1s and vice versa

  - f(a,b)=a+b dual of f(a,b)=a·b
  - f(x)=x+0 dual of f(x)=x·1

- The dual of any true statement is also true



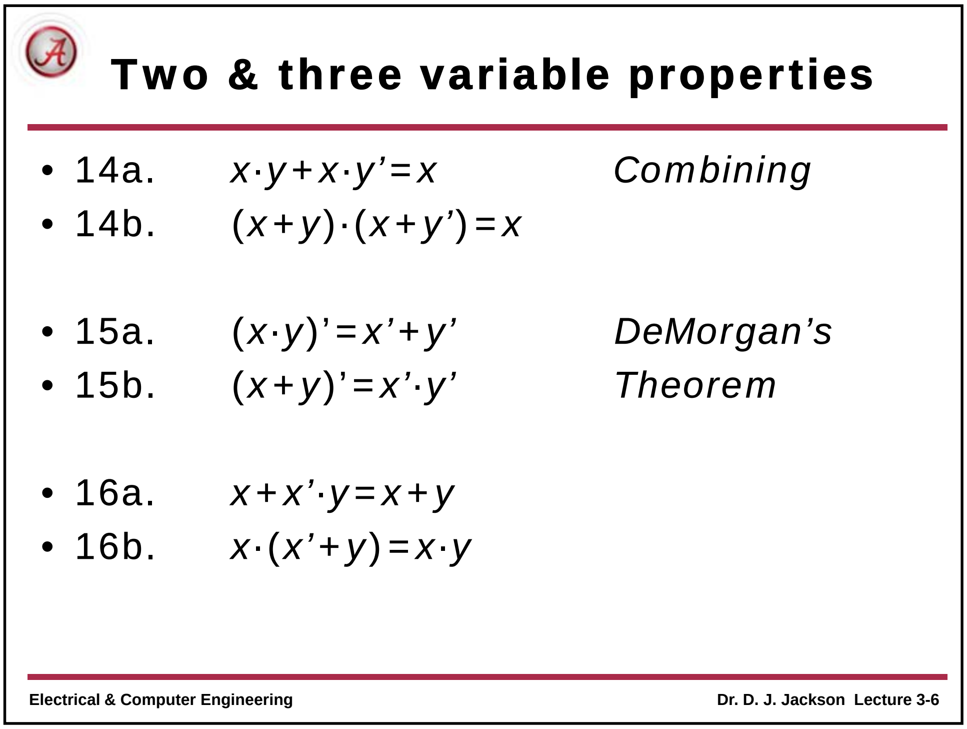
Two & three variable properties

- 10a. $x \cdot y = y \cdot x$ *Commutative*
- 10b. $x + y = y + x$

- 11a. $x \cdot (y \cdot z) = (x \cdot y) \cdot z$ *Associative*
- 11b. $x + (y + z) = (x + y) + z$

- 12a. $x \cdot (y + z) = x \cdot y + x \cdot z$ *Distributive*
- 12b. $x + y \cdot z = (x + y) \cdot (x + z)$

- 13a. $x + x \cdot y = x$ *Absorption*
- 13b. $x \cdot (x + y) = x$

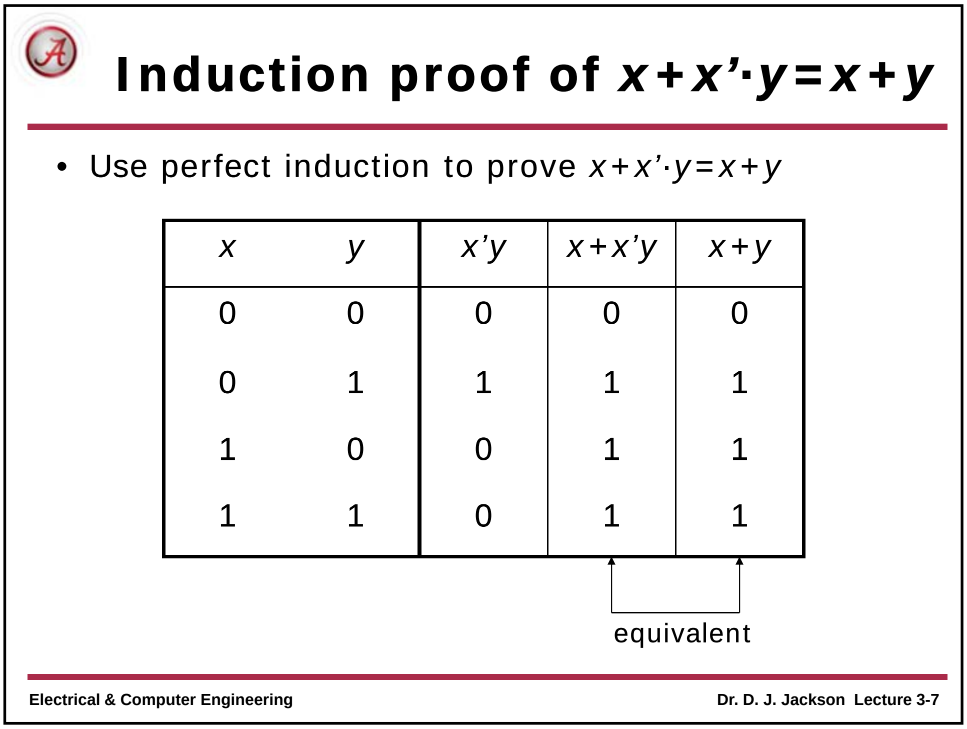


Two & three variable properties

- 14a. $x \cdot y + x \cdot y' = x$ *Combining*
- 14b. $(x + y) \cdot (x + y') = x$

- 15a. $(x \cdot y)' = x' + y'$ *DeMorgan's*
- 15b. $(x + y)' = x' \cdot y'$ *Theorem*

- 16a. $x + x' \cdot y = x + y$
- 16b. $x \cdot (x' + y) = x \cdot y$

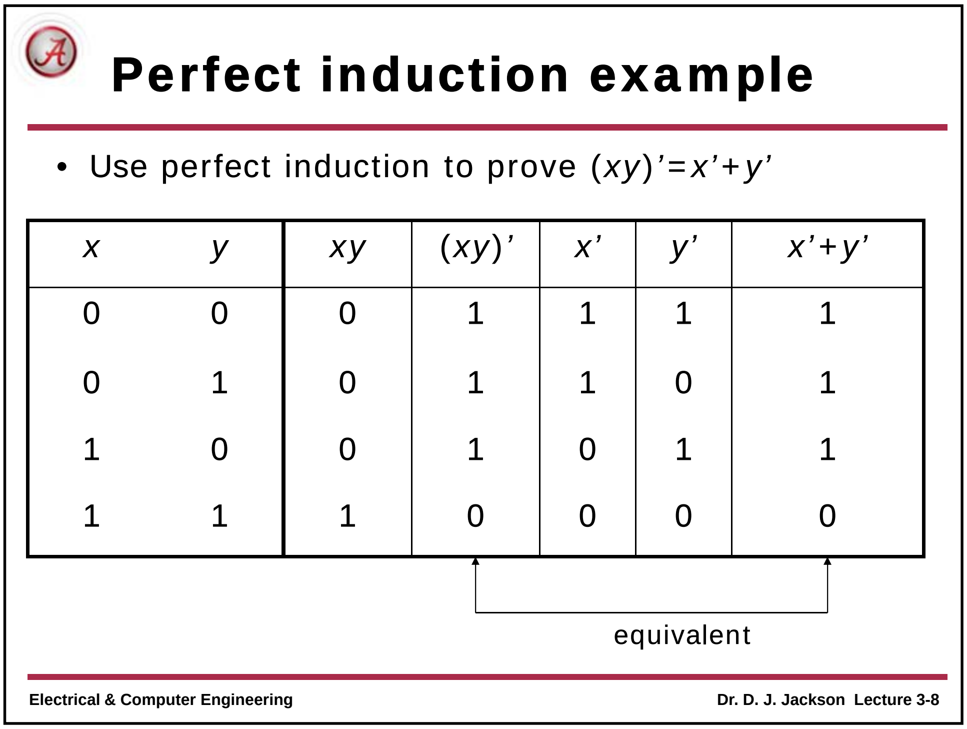


Induction proof of $x+x'\cdot y=x+y$

- Use perfect induction to prove $x+x'\cdot y=x+y$

x	y	$x'y$	$x+x'y$	$x+y$
0	0	0	0	0
0	1	1	1	1
1	0	0	1	1
1	1	0	1	1

equivalent

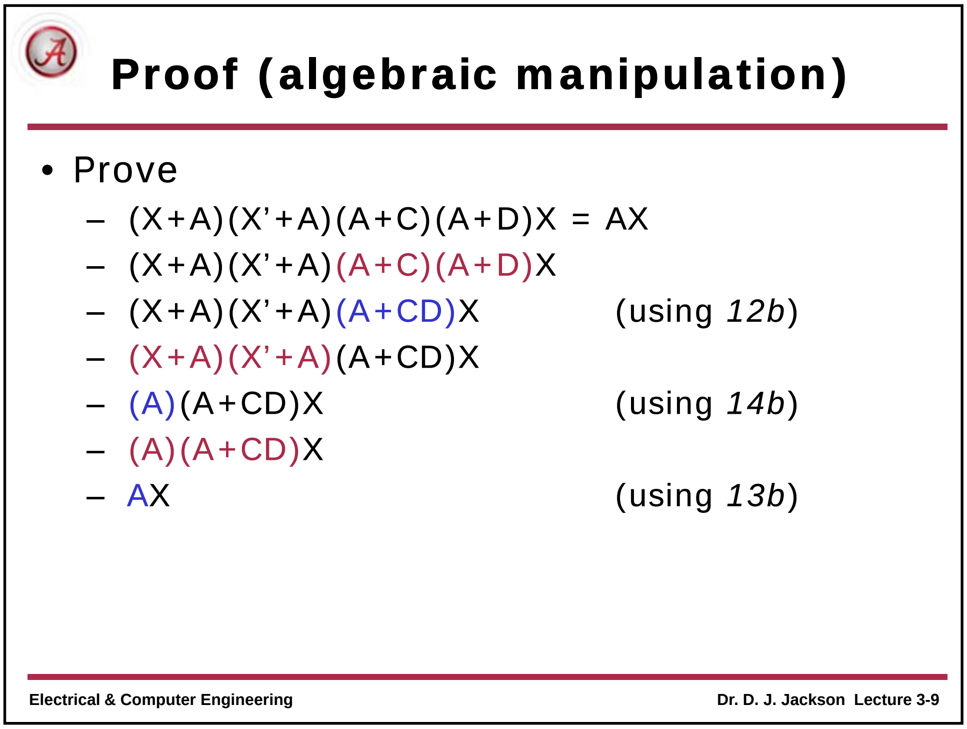


Perfect induction example

- Use perfect induction to prove $(xy)'=x'+y'$

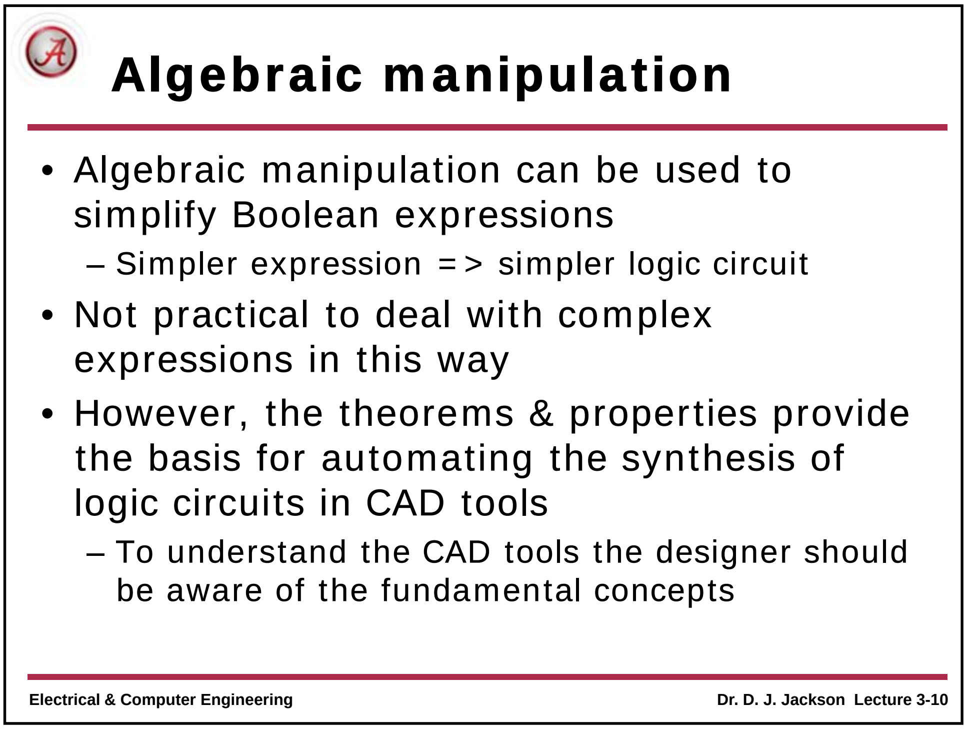
x	y	xy	$(xy)'$	x'	y'	$x'+y'$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

equivalent



Proof (algebraic manipulation)

- Prove
  - (X+A)(X'+A)(A+C)(A+D)X = AX
  - (X+A)(X'+A)(A+C)(A+D)X
  - (X+A)(X'+A)(A+CD)X (using *12b*)
  - (X+A)(X'+A)(A+CD)X
  - (A)(A+CD)X (using *14b*)
  - (A)(A+CD)X
  - AX (using *13b*)



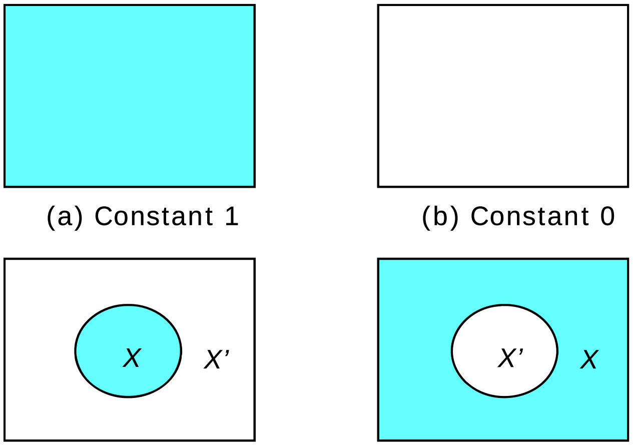
Algebraic manipulation

- Algebraic manipulation can be used to simplify Boolean expressions
  - Simpler expression => simpler logic circuit
- Not practical to deal with complex expressions in this way
- However, the theorems & properties provide the basis for automating the synthesis of logic circuits in CAD tools
  - To understand the CAD tools the designer should be aware of the fundamental concepts

Venn diagrams

- Venn diagram: graphical illustration of various operations and relations in an algebra of sets
- A set *s* is a collection of elements that are members of *s* (for us this would be a collection of Boolean variables and/or constants)
- Elements of the set are represented by the area enclosed by a contour (usually a circle)

Venn diagrams

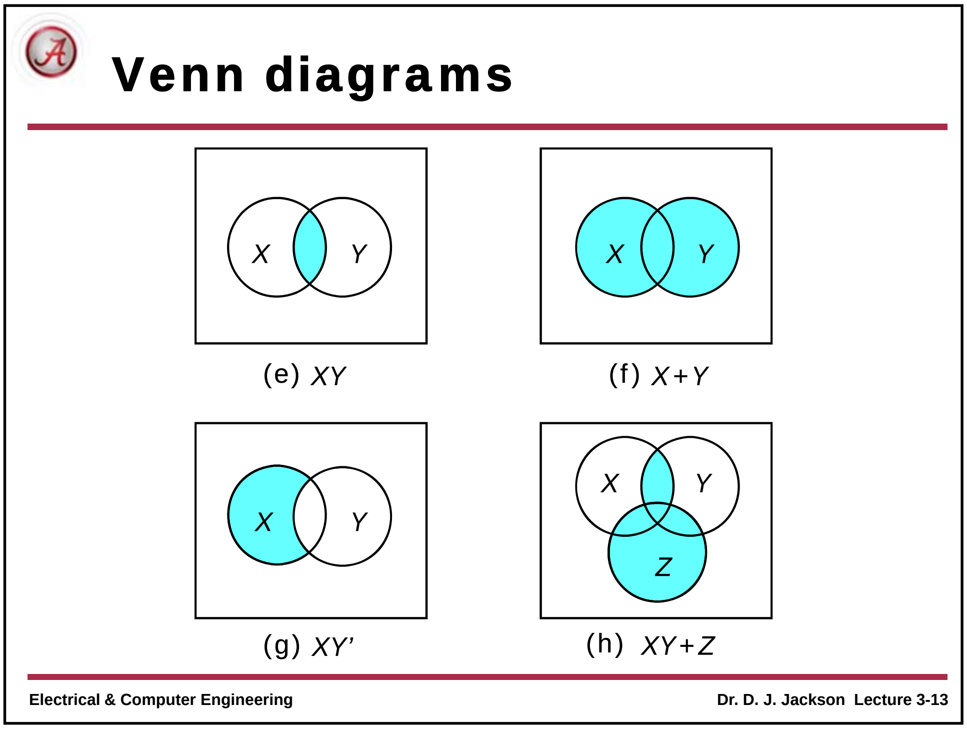


(a) Constant 1

(b) Constant 0

(c) Variable *X*

(d) *X'*



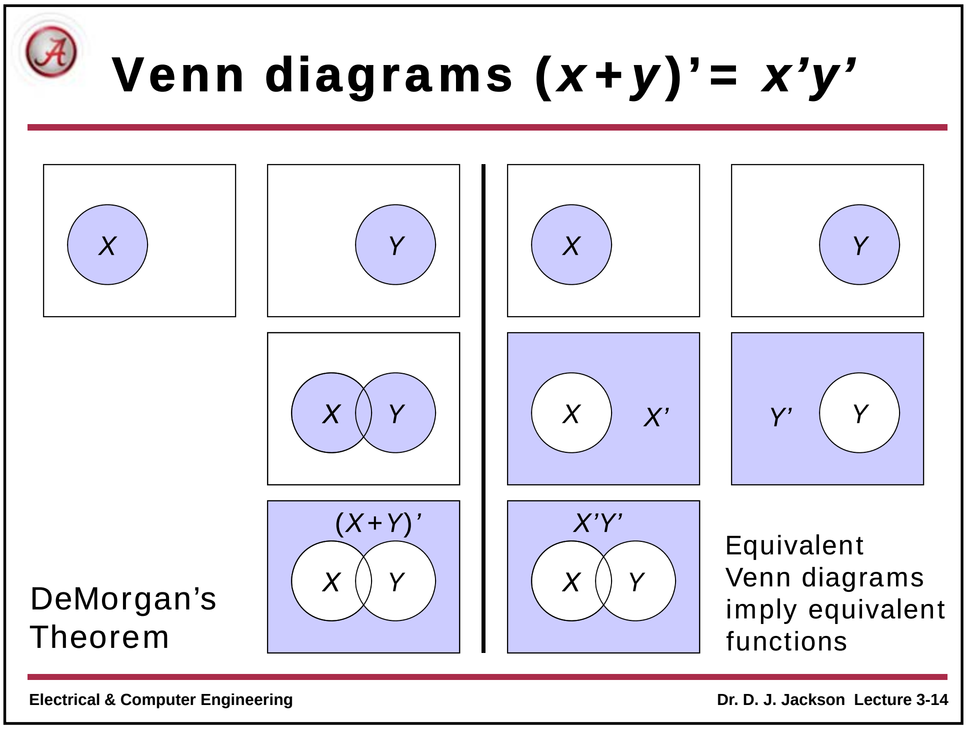
Venn diagrams

(e) XY

(f) $X+Y$

(g) XY'

(h) $XY+Z$



Venn diagrams $(x+y)' = x'y'$

$(X+Y)'$

$X'Y'$

DeMorgan's Theorem

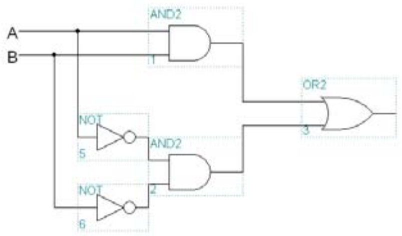
Equivalent Venn diagrams imply equivalent functions

Notation and terminology

- Because of the similarity with arithmetic addition and multiplication operations, the **OR** and **AND** operations are often called the ***logical sum*** and ***product*** operations
- The expression
  - ABC+A’BD+ACE’
  - Is a sum of three product terms
- The expression
  - (A+B+C)(A’+B+D)(A+C+E’)
  - Is a product of three sum terms

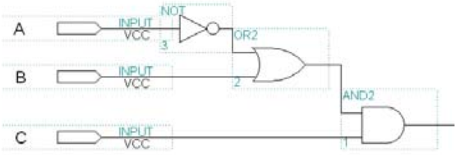
Precedence of operations

- In the absence of parentheses, operations in a logical expression are performed in the order
  - NOT, AND, OR
- Thus in the expression AB+A’B’, the variables in the second term are complemented before being ANDed together. That term is then ORed with the ANDed combination of A and B (the AB term)

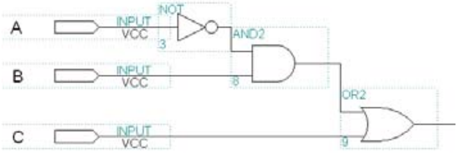


Precedence of operations

- Draw the circuit diagrams for the following
  - f(a,b,c)=(a’+b)c



  - f(a,b,c)=a’b+c



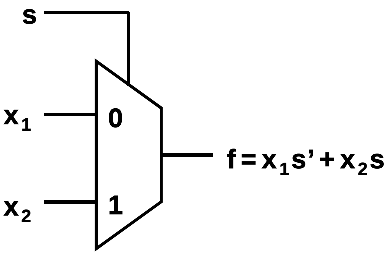
ECE380 Digital Logic

Combinatorial Circuit Building Blocks:

Multiplexers

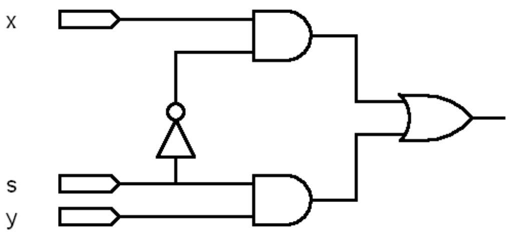
Multiplexers

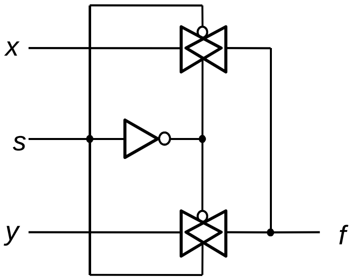
- A multiplexer (MUX) circuit has
  - A number of data inputs
  - One (or more) select inputs
  - One output
- It passes the signal value on one of its data inputs to its output based on the value(s) of the select signal(s)



s	$f(s,x_1,x_2)$
0	x_1
1	x_2

Multiplexer implementations

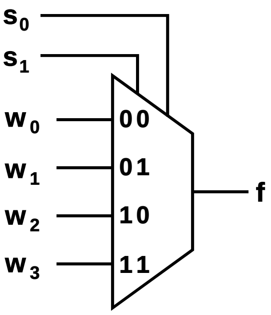




The preferred implementation

4-input multiplexer

- A 4-input multiplexer 'selects' one of four data inputs to be output based on the values of 2 select lines

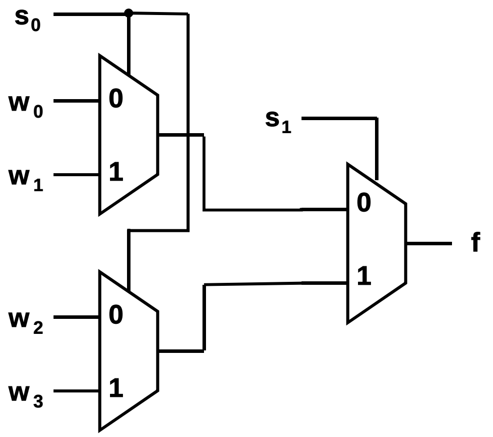


s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

$$f = s_1's_0'w_0 + s_1's_0w_1 + s_1s_0'w_2 + s_1s_0w_3$$

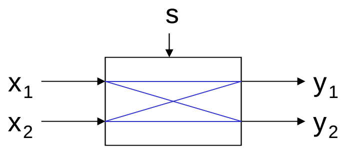
Building a 4-input MUX

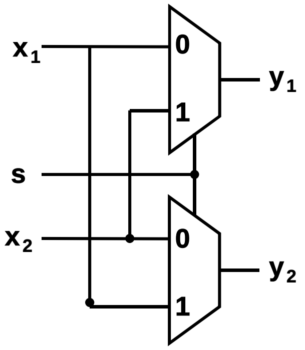
- A 4-input multiplexer can be constructed using 2-input multiplexers



MUX application (a 2x2 crossbar)

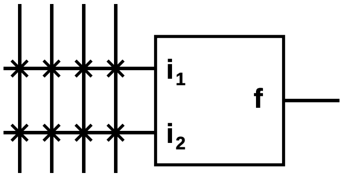
- A circuit with *n* inputs and *k* outputs whose function is to provide a capability to connect any input to any output is called a ***nxk crossbar switch***
  - With 2 inputs and 2 outputs, it is called a 2x2 crossbar
  - Useful in applications where it is necessary to connect one set of wires to another set of wires, where the connection pattern changes from time to time
  - Telephone switching networks are an example



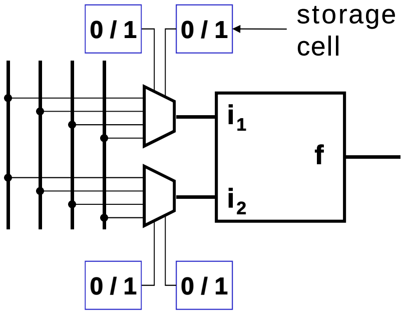


MUX application (prog. switch)

- In programmable devices (PLDs, CPLDs and FPGAs) programmable switches connect wires inside the device
  - These can be implemented with multiplexers



An FPGA logic block with programmable inputs

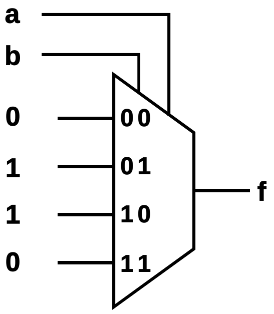


MUX implementation

Logic functions using MUXs

- MUXs can be used to synthesize logic functions
  - The LUT implementations use MUXs to select a (constant) value from a look-up table
- Consider the XOR function

a	*b*	*f*
0	0	0
0	1	1
1	0	1
1	1	0



Logic functions using MUXs

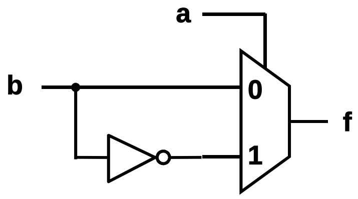
- The previous XOR solution is not particularly efficient

a	*b*	*f*
0	0	0
0	1	1
1	0	1
1	1	0

when a=0, f=b

when a=1, f=b’

a	*f*
0	b
1	b’



Logic functions using MUXs

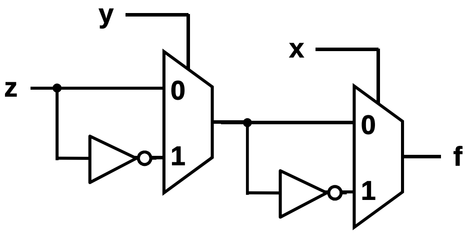
- Implement the following with a 2-input MUX and any additional logic gates

a	*b*	*f*
0	0	1
0	1	1
1	0	0
1	1	1

Logic functions using MUXs

- A 3-input XOR can be implemented with two 2-input MUXs

x	*y*	*z*	*f*	
0	0	0	0	y⊕z
0	0	1	1	
0	1	0	1	
0	1	1	0	
1	0	0	1	(y⊕z)’
1	0	1	0	
1	1	0	0	
1	1	1	1	



Logic functions using MUXs

- Implement the following with 2-input MUXs and any additional logic gates

x	*y*	*z*	*f*
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

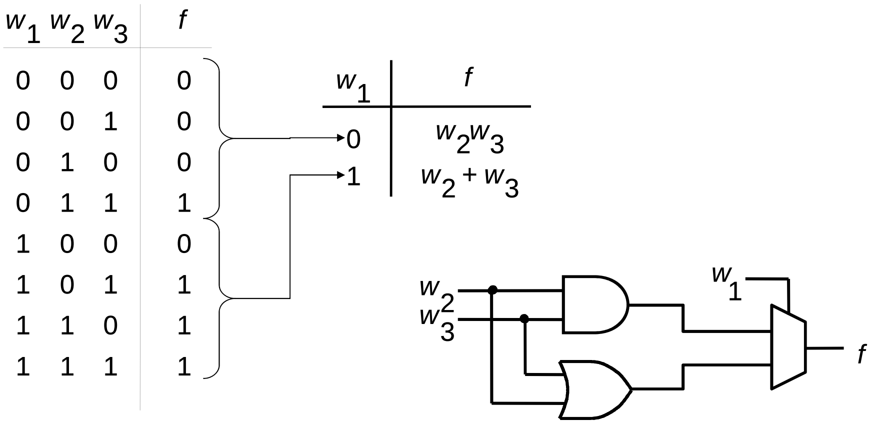
Shannon's expansion theorem

- Any Boolean function $f(w_1,\ldots,w_n)$ can be written in the form

  $f(w_1,\ldots,w_n) = (w_1)' \cdot f(0,w_2,\ldots,w_n) + (w_1) \cdot f(1,w_2,\ldots,w_n)$
- The expansion can be done using any of the n variables
- If $f(w_1,w_2,w_3) = w_1w_2 + w_1w_3 + w_2w_3$
  - Expanding this in terms of w_1 gives

$$f(w_1,w_2,w_3) = \underbrace{w_1(w_2+w_3)}_{\text{f when } w_1=1} + \underbrace{(w_1)'(w_2w_3)}_{\text{f when } w_1=0}$$

Shannon's expansion example



w_1	w_2	w_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

w_1	f
0	w_2w_3
1	$w_2 + w_3$

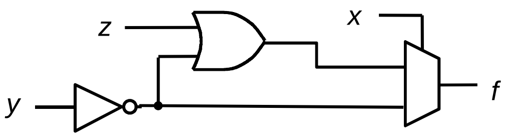
Shannon's expansion example

x	*y*	*z*	*f*
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

$f=x'y'z'+x'y'z+x'yz+xy'z'+xy'z$

choose x as the expansion variable

$f=x'(y'z'+y'z+yz)+x(y'z'+y'z)$
$f=x'(y'+z)+x(y')$



Shannon's expansion example

x	*y*	*z*	*f*
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

$f=x'y'z'+x'y'z+x'yz+xy'z'+xy'z$

choose z as the expansion variable

Lecture Three

Shannon's Expansion Theorem

T11a:

$$F(X1,\ldots,Xn) = F(0, X2,\ldots,Xn) \bullet \overline{X1} + F(1, X2,\ldots,Xn) \bullet X1$$

T11b:

$$F(X1,\ldots,Xn) = [F(1, X2,\ldots,Xn) + \overline{X1}] \bullet [F(0, X2,\ldots,Xn) + X1]$$

Shannon's Expansion Theorem

Let X1=0 in T11a

$$F(X1,\ldots,Xn) = F(0, X2,\ldots,Xn)\bullet\bar{0} + F(1, X2,\ldots,Xn)\bullet 0$$

$$F(X1,\ldots,Xn) = F(0, X2,\ldots,Xn)\bullet 1 + F(1, X2,\ldots,Xn)\bullet 0$$

$$F(X1,\ldots,Xn) = F(0, X2,\ldots,Xn)$$

Let X1=1 in T11a

$$F(X1,\ldots,Xn) = F(0, X2,\ldots,Xn)\bullet\bar{1} + F(1, X2,\ldots,Xn)\bullet 1$$

$$F(X1,\ldots,Xn) = F(0, X2,\ldots,Xn)\bullet 0 + F(1, X2,\ldots,Xn)\bullet 1$$

$$F(X1,\ldots,Xn) = F(1, X2,\ldots,Xn)$$

Shannon's Expansion Theorem

$$F(X1,\ldots,Xn) = F(0,0,X3,\ldots,Xn)\bullet \overline{X1}\bullet \overline{X2} + F(0,1,X3,\ldots,Xn)\bullet \overline{X1}\bullet X2$$

$$+F(1,0,X3,\ldots,Xn)\bullet X1\bullet \overline{X2} + F(1,1,X3,\ldots,Xn)\bullet X1\bullet X2$$

$$= I0\bullet \overline{X1}\bullet \overline{X2} + I1\bullet \overline{X1}\bullet X2 + I2\bullet X1\bullet \overline{X2} + I3\bullet X1\bullet X2$$

$$= I0\bullet m0 + I1\bullet m1 + I2\bullet m2 + I3\bullet m3$$

$$= \sum_{k=0}^{2^n-1} Ki\bullet m_i \qquad \text{Where} \quad m_i = m_i(X1, X2)$$

Example

Design a circuit using MUX to implement the following function by applying Shannon's Expansion Theorem T11a with respect to A and B

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

Solution

$$F(A,B,C) = F(0,0,C).\overline{A}.\overline{B} + F(0,1,C).\overline{A}.B$$
$$+ F(1,0,C).A.\overline{B} + F(1,1,C).A.B$$

$$F(0,0,C) = 0 + 0.\overline{C} = 0$$

$$F(0,1,C) = 0 + 1.\overline{C} = \overline{C}$$

$$F(1,0,C) = 1 + 0.\overline{C} = 1$$

$$F(1,1,C) = 1 + 1.\overline{C} = 1$$

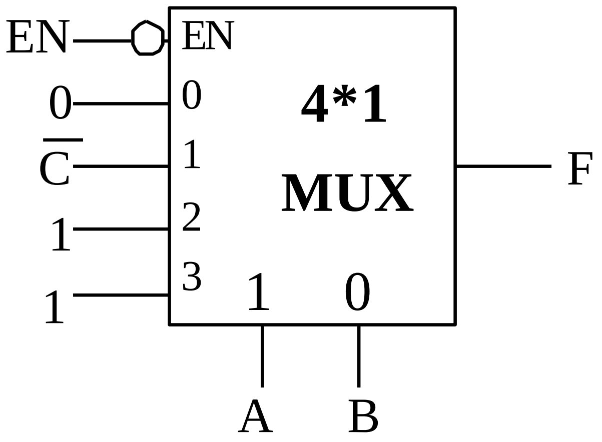


Figure (1)

Analyzing a Multiplexer Design

$$F(A,B,C) = \sum_{k=0}^{2^n-1} Ki \bullet m_i \qquad \text{Where} \quad m_i = m_i(A,B)$$

$$= I0 \bullet m0 + I1 \bullet m1 + I2 \bullet m2 + I3 \bullet m3$$

$$= 0 \bullet m0 + \overline{C} \bullet m1 + 1 \bullet m2 + C \bullet m3$$

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

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

$$= m_2 + m_4 + m_5 + m_7$$

$$= \sum m(2,4,5,7)$$

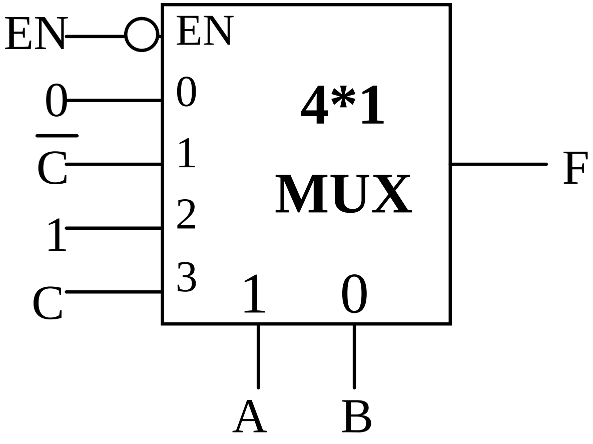


Figure (2)

Field Programmable Gate Array (FPGA)

- FPGAs are based on gate array technology, unlike the PROM technology of early PLDs. These devices comprise configurable logic blocks (**CLBs**) along with an **interconnection matrix** running in between.
- FPGAs work based on the **look-up tables (LUTs**) and the **flip-flops** that form a part of **CLB**. The user has to program the CLBs to perform a certain logical function and then use the **interconnection matrix** to connect one or more logic blocks together. Further, they comprise **input-output (I/O) ports** facilitating the design both from the point of programming as well as debugging.

- These devices are capable of implementing **state-machine-based sequential** designs along with designs based on **combinational logic**.
- FPGAs are used to realize more complex designs when compared to CPLDs due to their high density. Moreover, FPGAs offer the customer the **flexibility to design/re-design** the logic even after being deployed in the work field which gives them the name field-programmable. However, FPGAs have **larger propagation delays** when compared to CPLDs.
- All of these PLDs are programmable using device programs that **transfer the Boolean logic pattern onto the programmable device**

Programming Technologies

- There are a number of programming technologies that have been used for **reconfigurable architectures**. Each of these technologies has different characteristics which in turn have a significant effect on the programmable architecture. Some of the well-known technologies **include static memory, flash, and anti-fuse**.

SRAM-Based Programming Technology

Static memory cells are the basic cells used for SRAM-based FPGAs. Most commercial vendors **use static memory (SRAM)** based programming technology in their devices. These devices use static memory cells which are divided throughout the FPGA to provide configurability. An example of such a memory cell is shown in Fig.3 . In an SRAM-based FPGA, SRAM cells are mainly used for the following purposes:

1. To program **the routing interconnect** of FPGAs which are generally steered by small multiplexors.
2. To program **Configurable Logic Blocks** (CLBs) that are used to implement logic functions.

- SRAM-based programming technology has become the dominant approach for FPGAs because of **its re-programmability** and the use of standard **CMOS** process technology therefore leading to increased integration, higher speed and lower dynamic power consumption of new process with smaller geometry. There are however a number of drawbacks associated with SRAM-based programming technology. For example an SRAM cell requires **6 transistors** which makes the use of this technology costly in terms of area compared to other programming technologies. Further SRAM cells are **volatile** in nature and **external devices are required to permanently store the configuration data**. These external devices add to **the cost and area** overhead of SRAM-based FPGAs.

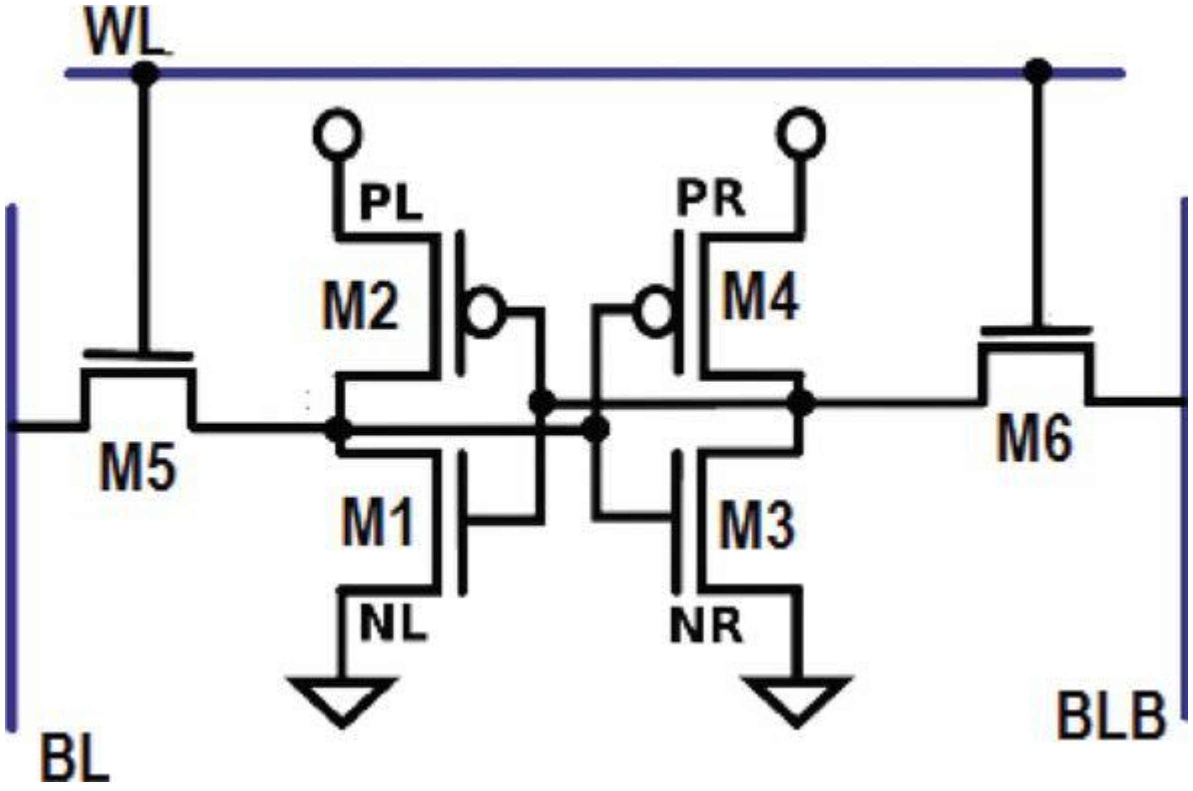


Figure -3-

Flash Programming Technology

One alternative to the SRAM-based programming technology is the use of flash or EEPROM-based programming technology. Flash-based programming technology

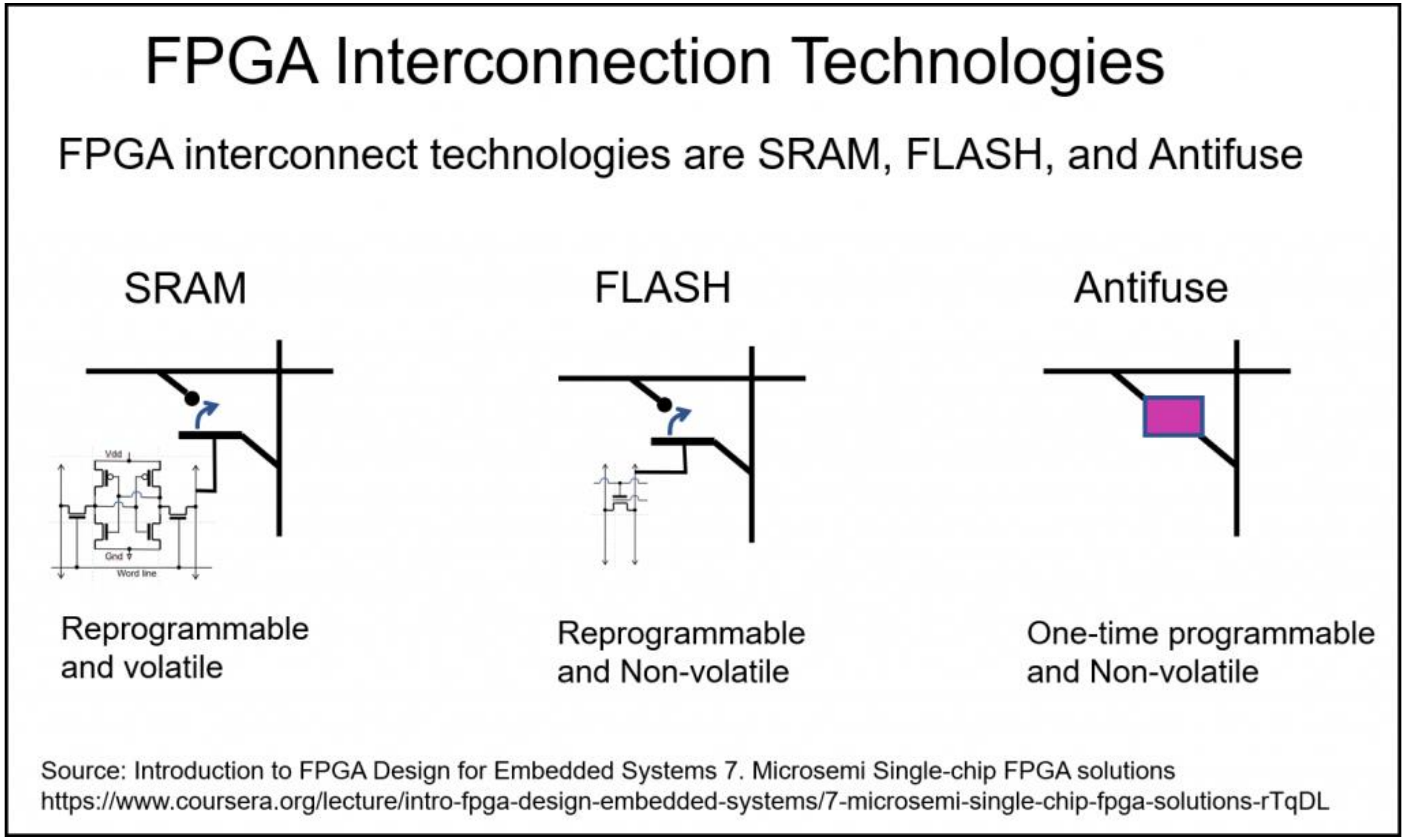
offers several advantages. For example, this programming technology is ***nonvolatile*** in nature. Flash-based programming technology is ***also more area-efficient*** than SRAM-based programming technology. Flash-based programming technology has its own disadvantages, flash-based technology uses **non-standard CMOS processes**.

Anti-fuse Programming Technology

An alternative to SRAM and flash-based technologies is anti-fuse programming technology. The primary advantage of anti-fuse programming technology is its **low area**. Also, this technology has lower **resistance and parasitic capacitance than the other two.**

Programming Technologies

- programming technologies. Further, this technology is **non-volatile** in nature. There are however significant disadvantages associated with this programming technology.
- For example, this technology does not make use of the standard CMOS process. Also, anti-fuse programming technology-based devices can **not be reprogrammed**.



FPGA Interconnection Technologies

FPGA interconnect technologies are SRAM, FLASH, and Antifuse

SRAM

Reprogrammable and volatile

FLASH

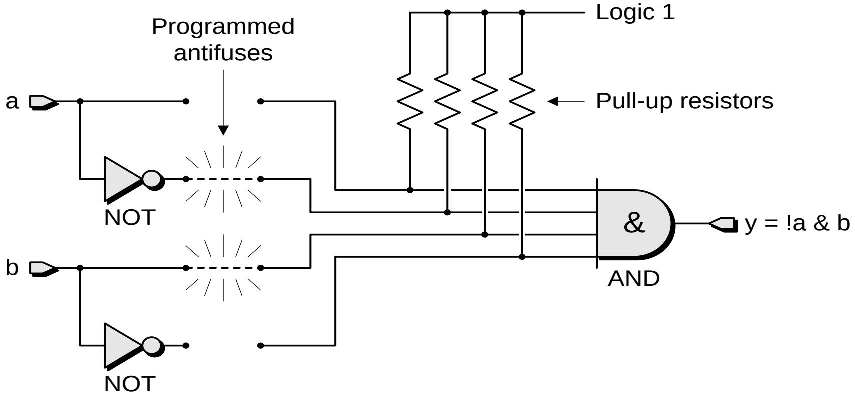
Reprogrammable and Non-volatile

Antifuse

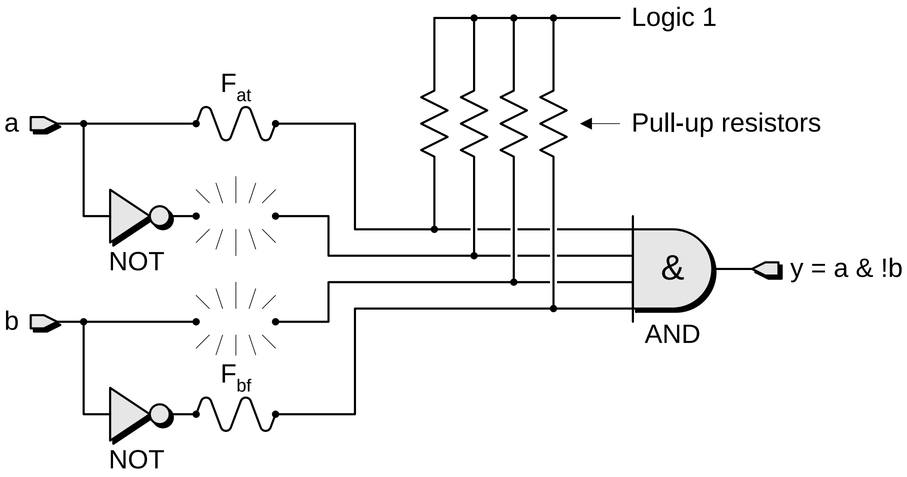
One-time programmable and Non-volatile

Source: Introduction to FPGA Design for Embedded Systems 7. Microsemi Single-chip FPGA solutions https://www.coursera.org/lecture/intro-fpga-design-embedded-systems/7-microsemi-single-chip-fpga-solutions-rTqDL

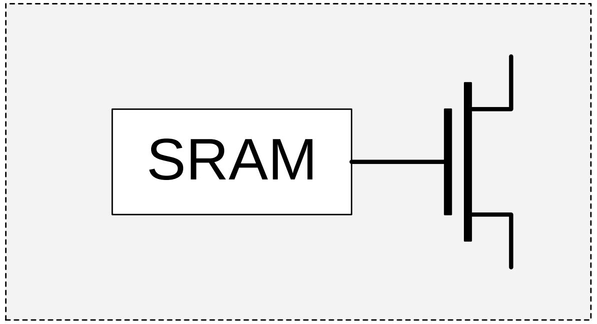
Figure -4-



Anti-fuse links

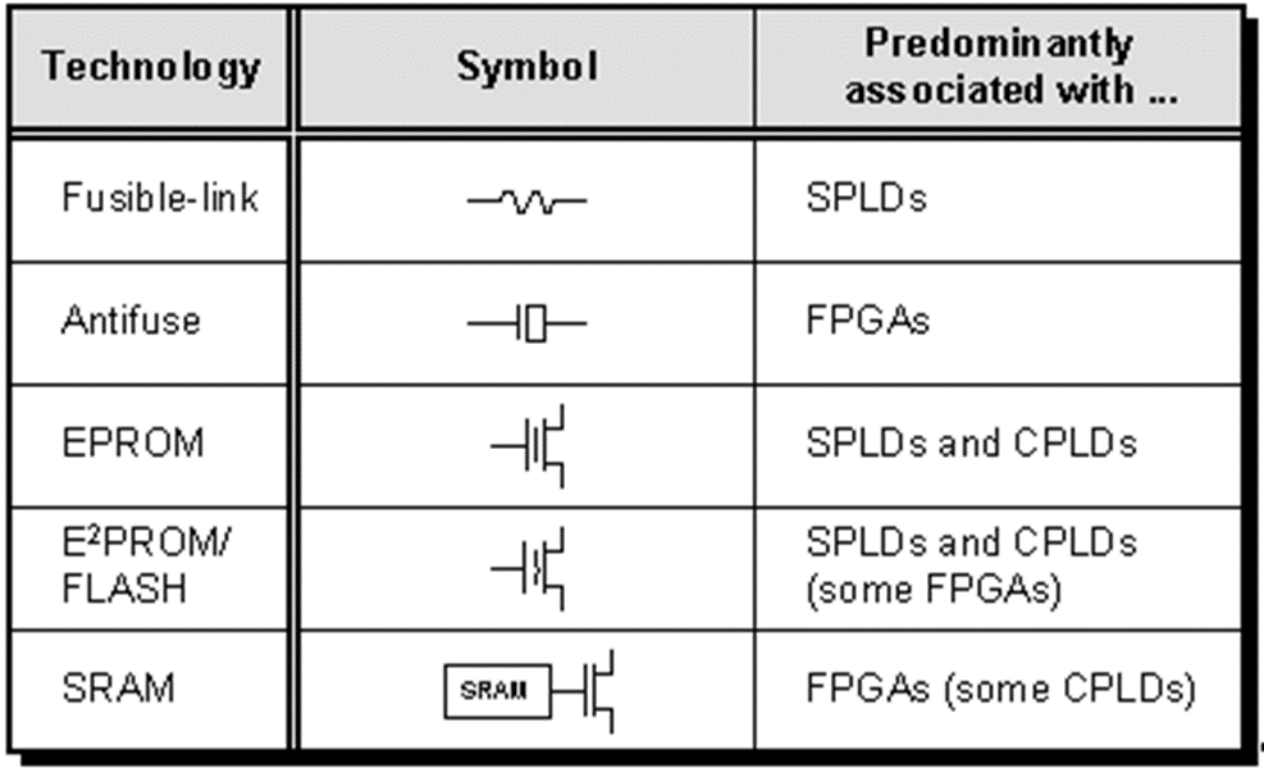


Fuse links



Volatile memory –
SRAM programmable

Figure -5-



Technology	Symbol	Predominantly associated with ...
Fusible-link		SPLDs
Antifuse		FPGAs
EPROM		SPLDs and CPLDs
E^2PROM/ FLASH		SPLDs and CPLDs (some FPGAs)
SRAM	SRAM	FPGAs (some CPLDs)

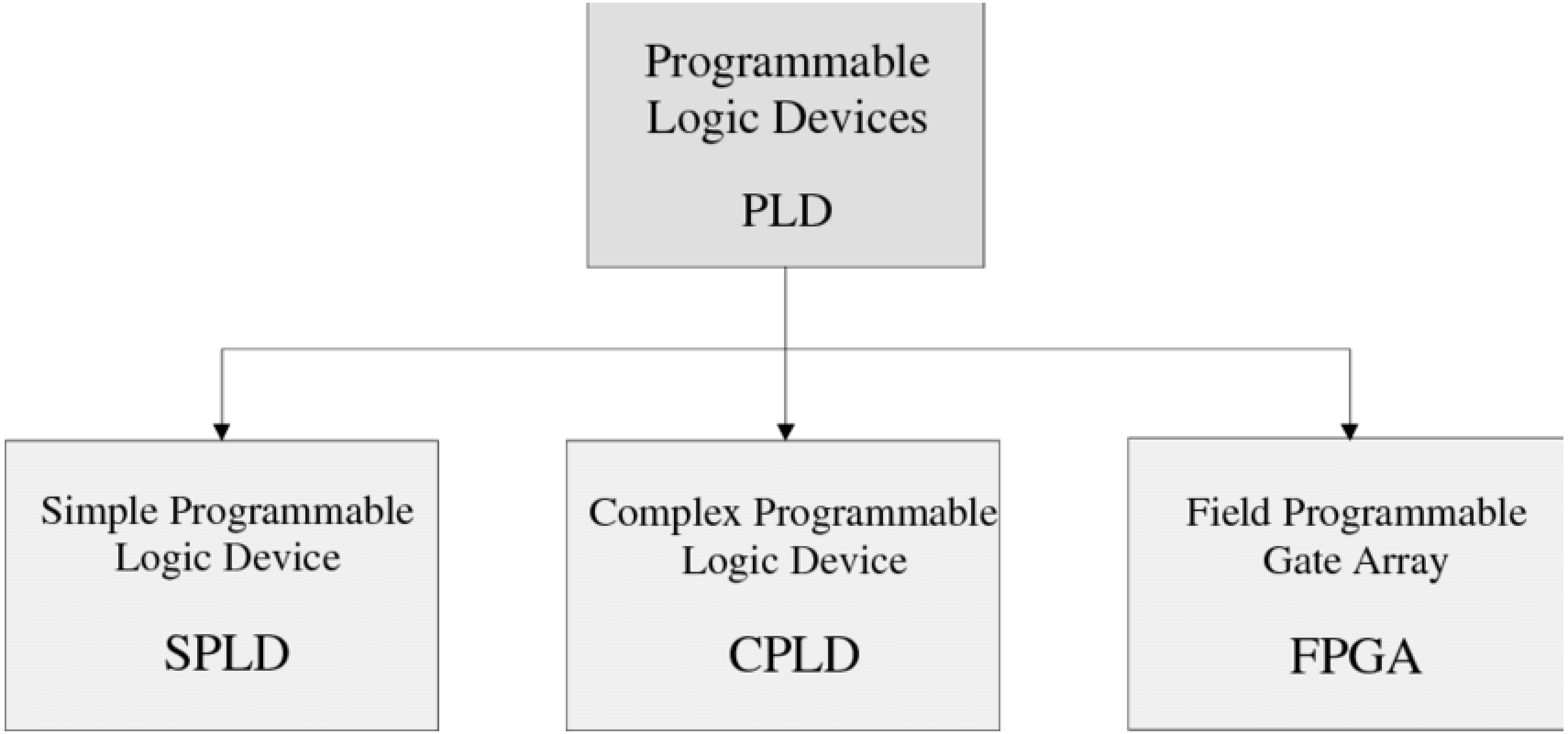
Classifying Devices

Device can be classed based on their level of programmability

One-Time Programmable: devices can be programmed only once; it's contents can not be changed. While typically these devices are fuse or anti-fuse-based, they can also be low-cost EPROM devices.

Re-programmable: These devices can have their configuration loaded more than once. SRAM-based and Flash-based devices may be reloaded without restriction.

Lecture Four


Programmable
Logic Devices
PLD
Simple Programmable
Logic Device
SPLD
Complex Programmable
Logic Device
CPLD
Field Programmable
Gate Array
FPGA

Complex Programmable Logic Devices (CPLDs)

*Complex Programmable Logic Devices (**CPLDs**)* integrate multiple **SPLDs** into a single chip. A typical **CPLD** provides a logic capacity equivalent to about **50** single **SPLDs**. Altera, the world's second and largest manufacturer − presently, the largest manufacturer is **Xilinx** − of programmable semiconductors, introduced the **MAX 5000, MAX 7000, and MAX 9000** series of CPLDs. MAX Wired – AND gates OR gate Flip Flop

5000 is based on older technology, the MAX **7000** family is a high−performance PLD and it is fabricated with **advanced CMOS** technology designed with state-of-the-art logic capacity and speed performance, and **MAX 9000** is very similar to the **MAX 7000** except that it has higher **logic capacity.**

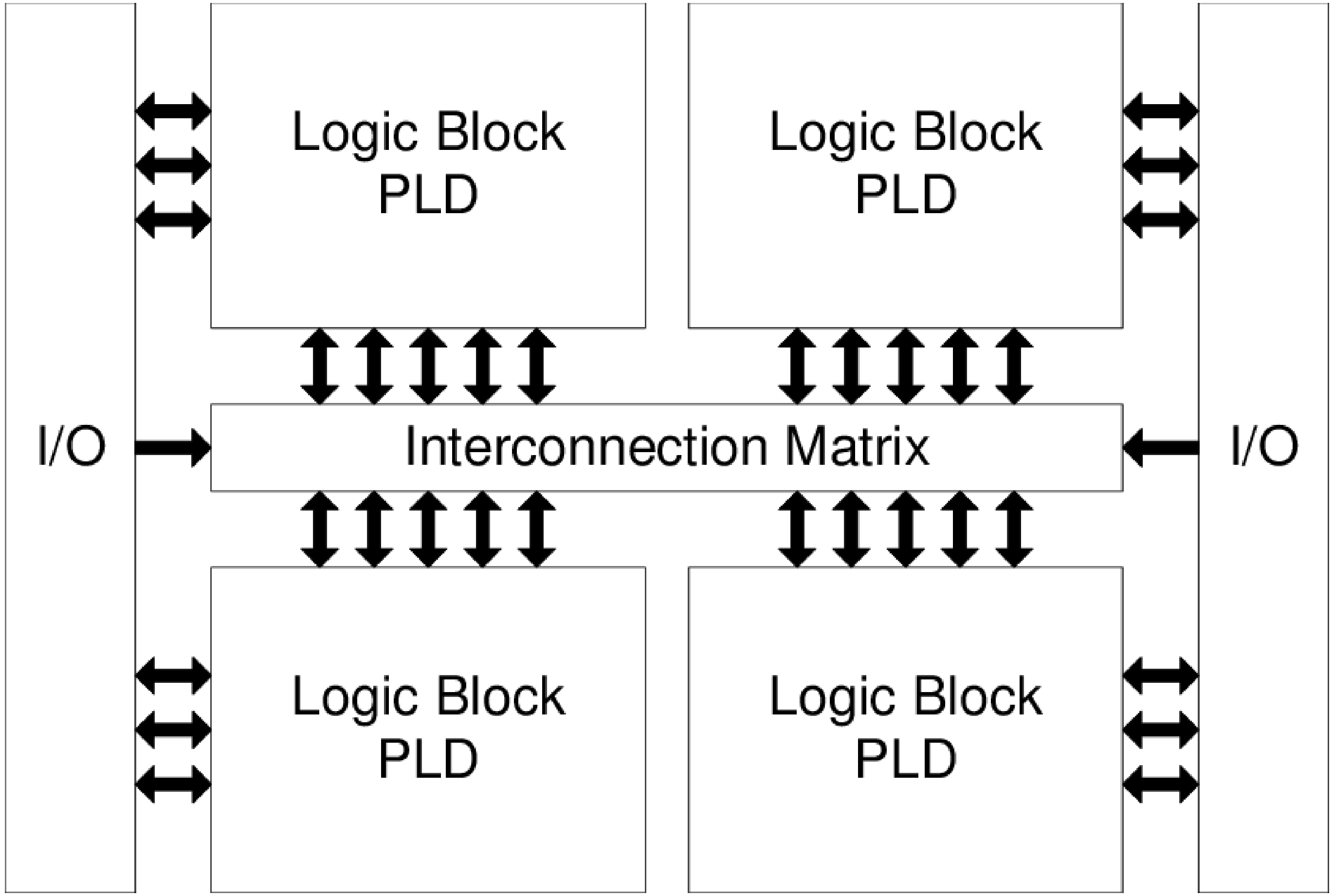
CPLD
(Complex *programmable logic devices*)

- **Commercially Available CPLDs**
  - **Altera CPLDs**
  - **Advanced Micro Devices (AMD) CPLDs**
  - **Lattice CPLDs**
  - **Cypress CPLDs**
  - **Xilinx CPLDs**
  - **Altera FLASHlogic CPLDs**

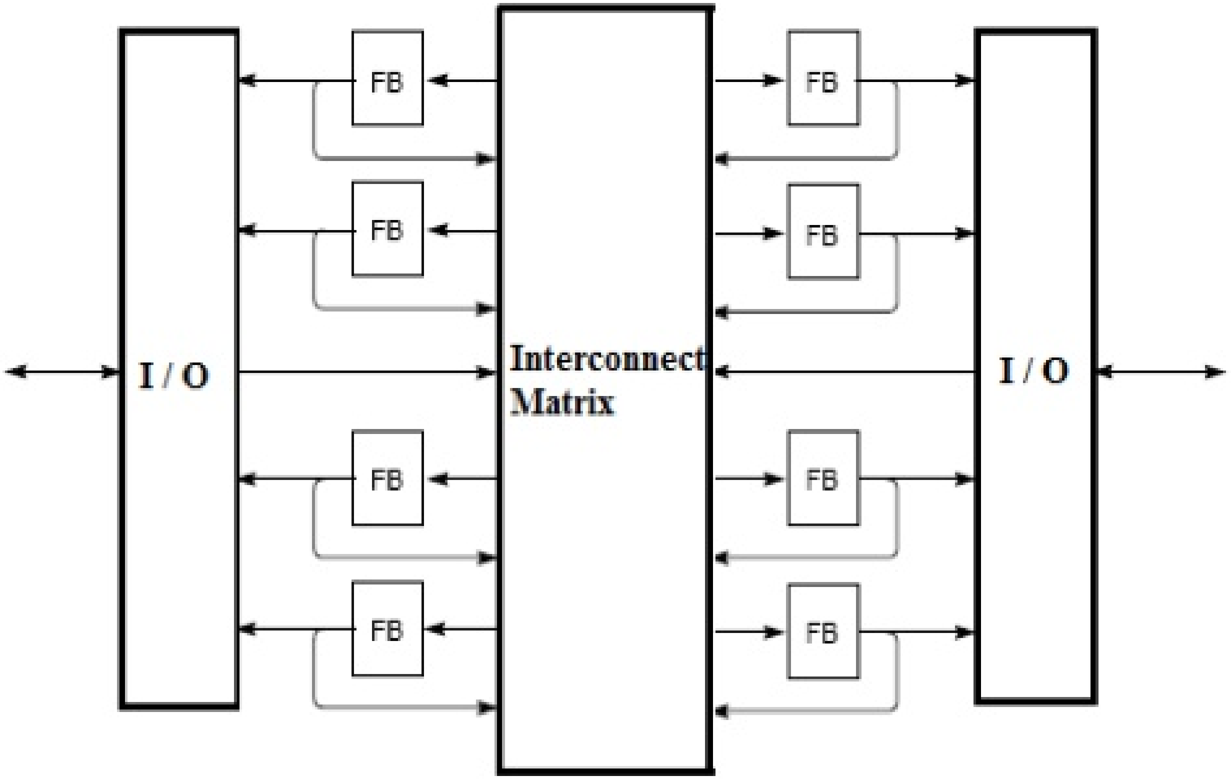
Altera CPLD

- Altera has developed three families of chips that fit within the CPLD category:
  - MAX 5000
  - MAX 7000, and
  - MAX 9000

- MAX 5000 represents an older technology that offers a cost effective solution,
- MAX 7000 series is widely used and offers state-of-the-art logic capacity and speed-performance.


Logic Block
PLD
Logic Block
PLD
I/O
Interconnection Matrix
I/O
Logic Block
PLD
Logic Block
PLD

CPLD

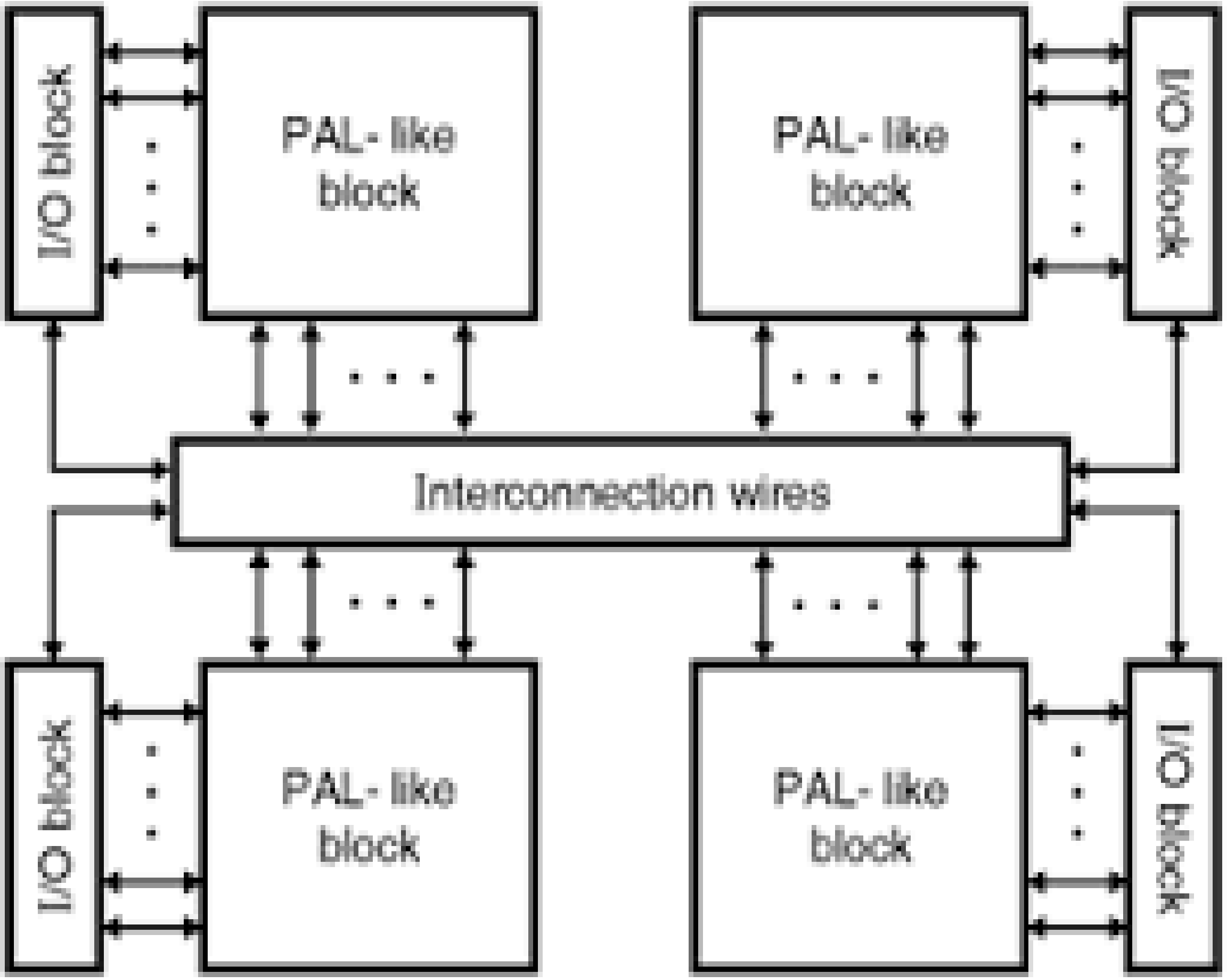


Detail of CPLD

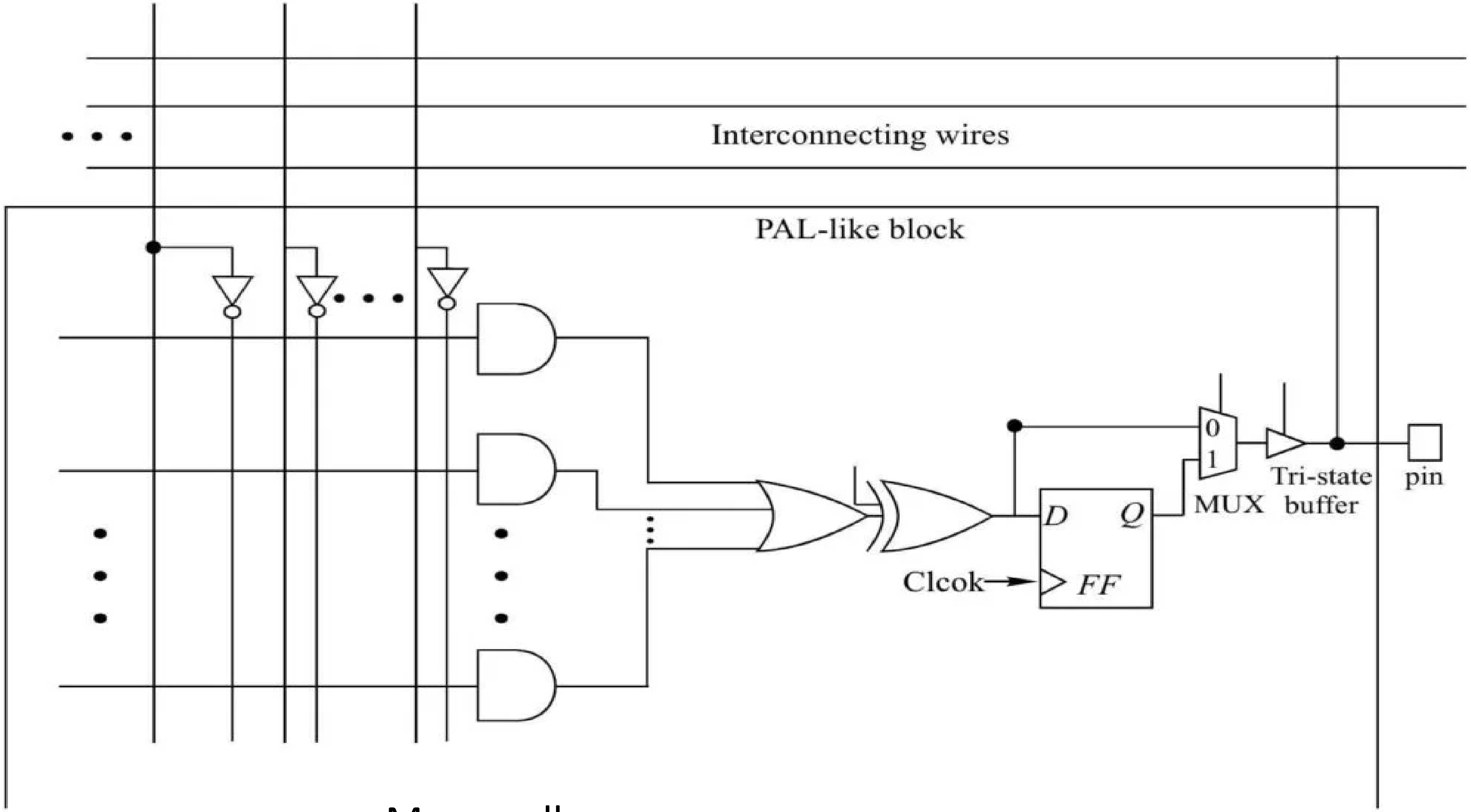
A Complex Programmable Logic Device (CPLD) is a **programmable logic** device and can **be programmed** by using **VHDL**. CPLDs are based on **EPROM or EEPROM technology**. CPLDs have **extended density than the SPLDs**. The concept of CPLDs is to have a few **macrocells** on **a single chip with simple logic paths**. CPLDs are classified depending on the architecture which gives rise to **high speed**, detailed timing, and simple software flow. The basic CPLD consists of the **configurable logic block** (**CLB**) which consists of AND gate arrays and interconnects. The logic blocks are **programmable AND, fixed OR** devices. **PALs and GALs are available only in small sizes, equivalent to a few hundred logic gates**. CPLD is an arrangement of multiple SPLD-like blocks on a single chip. CPLD consists of multiple circuit blocks in the chip. The circuit block in CPLD are the same as that of **PLA or PAL blocks**.

The figure below shows an example of a CPLD. This CPLD has four PAL blocks which are connected interconnection wires. The PAL block is also connected to a sub-circuit known as an **I/O block**. The I/O block is connected to a number of input and output pins. The PAL block consists of macrocells. The **macrocell** consists of **flip-flop,** a **multiplexer**, and a **tri-state buffer**. The flip-flop is used to store the output value produced by the OR gate. **The tri-state** buffer acts as a switch. In the function block, **the AND array gets inputs** from the I/O blocks and other function blocks. The product terms are given to fixed OR gates. The outputs of the multiplex or are then sent through a clocked flip-flop. The function blocks are designed similarly to PAL architectures.

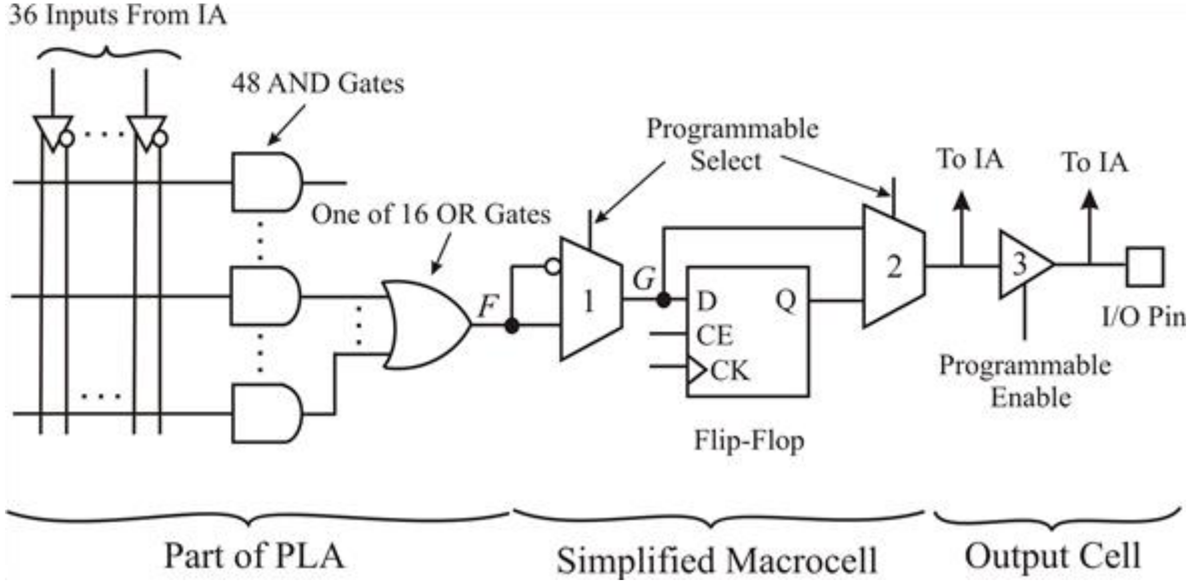
The I/O block is used to drive signals to the pins of the CPLD device. The CPLD interconnect is a programmable switch matrix.



Altera-Flash-Logic-CPLD


Interconnecting wires
PAL-like block
0
1
MUX
Tri-state
buffer
pin
D
Q
Clcok
FF

Macrocells


36 Inputs From IA
48 AND Gates
One of 16 OR Gates
F
Programmable
Select
1
G
D
Q
CE
CK
Flip-Flop
2
To IA
To IA
3
I/O Pin
Programmable
Enable
Part of PLA
Simplified Macrocell
Output Cell

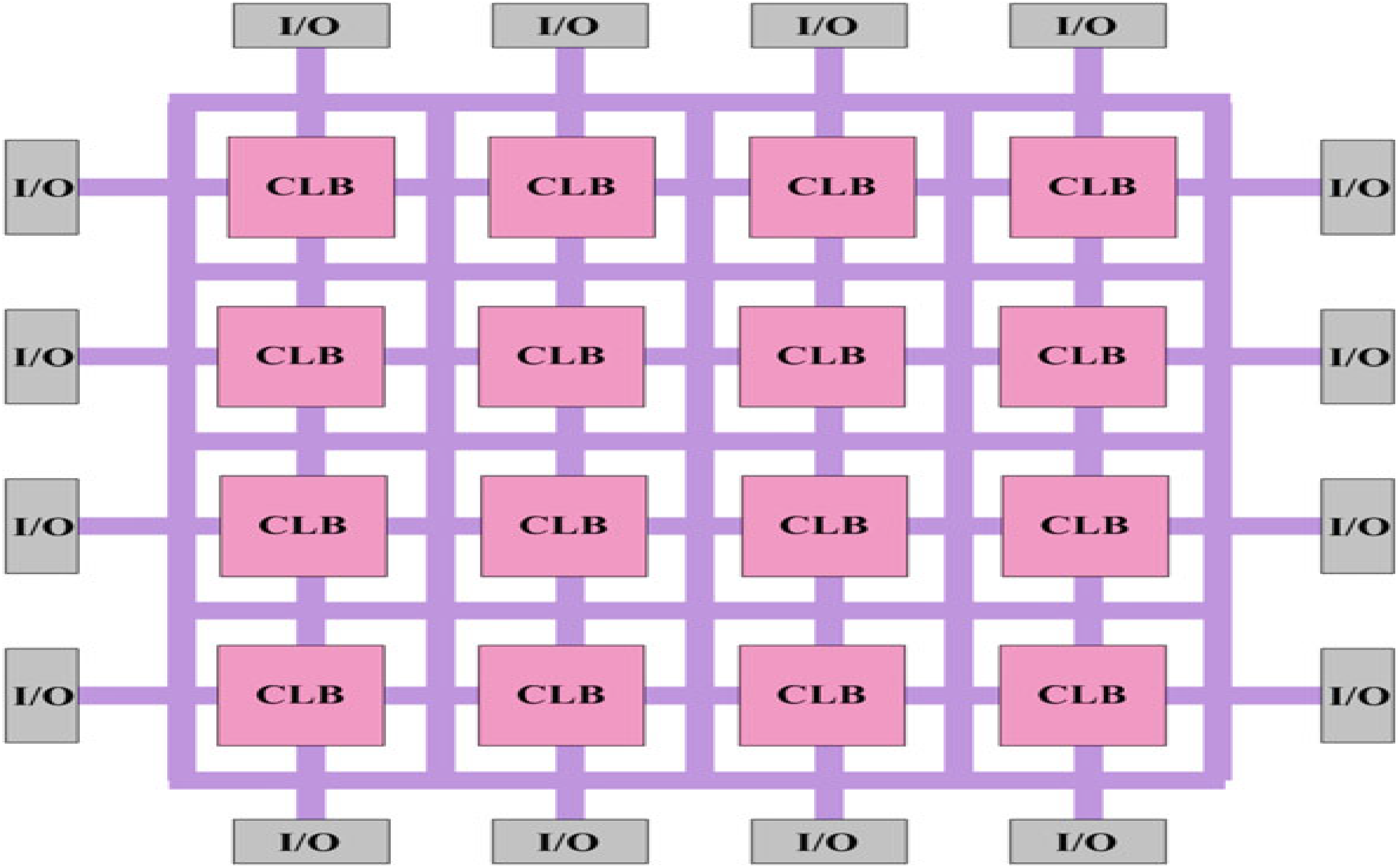
Macrocells

FPGA

Normally FPGAs comprise of:

- Programmable logic blocks which implement logic functions.
- Programmable routing that connects these logic functions.
- I/O blocks that are connected to logic blocks through routing interconnect and that make off-chip connections.

A generalized example of an FPGA is shown in the Figure below where configurable logic blocks (**CLBs**) are arranged in **a two dimensional grid** and are **interconnected** by **programmable routing** resources. I/O blocks are arranged at the **periphery** of the grid and they are also connected to the programmable routing interconnect. The "**programmable/reconfigurable**" term in FPGAs indicates their ability to implement **a new function** on the chip after its fabrication is complete. The configurability programmability of an FPGA is based on an **underlying programming** technology, which can cause a change in the behavior of a pre-fabricated chip after its fabrication.



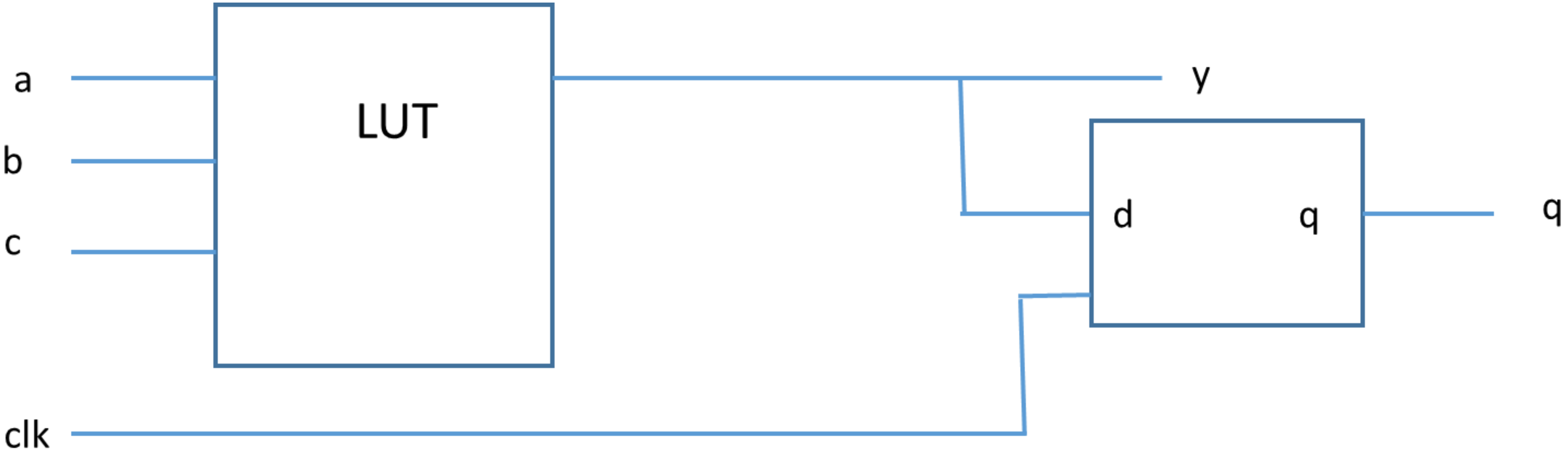
Overview of FPGA Architecture

Macro cell Most FPGA devices also embed certain *macro cells* or *macro blocks.* These are designed and fabricated at the transistor level, and their functionalities complement the general logic cells. Commonly used macro cells include **memory blocks**, **combinational multipliers, clock management circuits**, **and I/O interface circuits**. Advanced FPGA devices may even contain one or more prefabricated processor cores.

Overview of the Xilinx Spartan3 devices

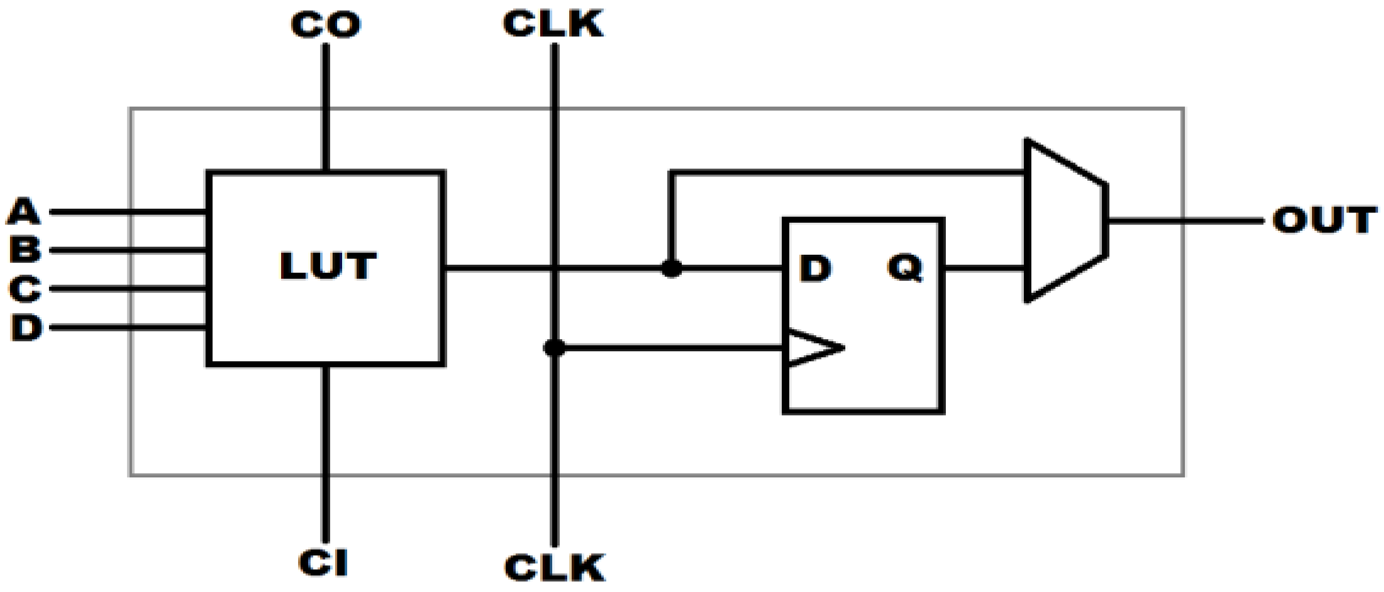
Xilinx Spartan-3 family FPGA devices. Based on the ratio between the number of logic cells and the I/O counts, the family is further divided into several subfamilies. Our discussion applies to all the subfamilies.

Logic cell, slice, and CLB The most basic element of the Spartan-3 device is a *logic cell* (LC), which contains a **four-input LUT** and a **D FF**, similar to that in Figure below. In addition, a logic cell contains **a carry circuit**, which is used to **implement arithmetic** functions, and a **multiplexing circuit**, which is used **to implement wide multiplexers**. The **LUT** can also be configured as a **16-by- 1 static random access memory (SRAM**) or a 16-bit shift register.

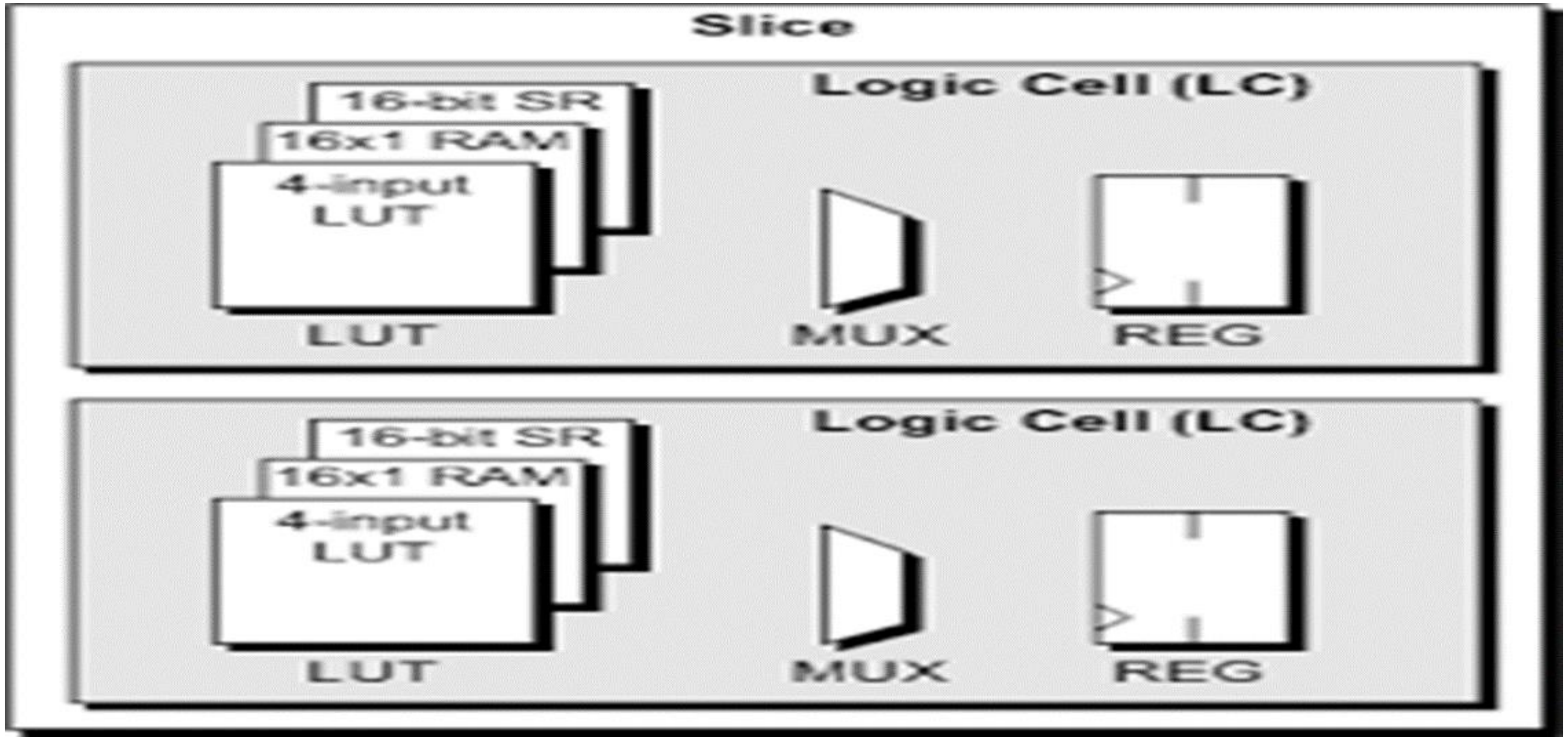


Conceptual Diagram

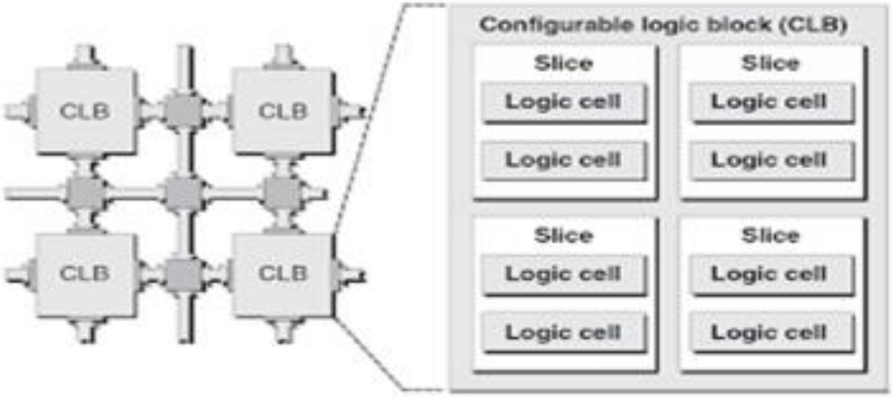
To increase flexibility and improve performance, eight logic cells are combined with a special internal **routing structure**. In Xilinx terms, two logic cells are grouped to form **a *slice,*** and **four slices** are grouped to form **a *configurable logic block*** (CLB).



Logic cell



A Slice containing two logic cells



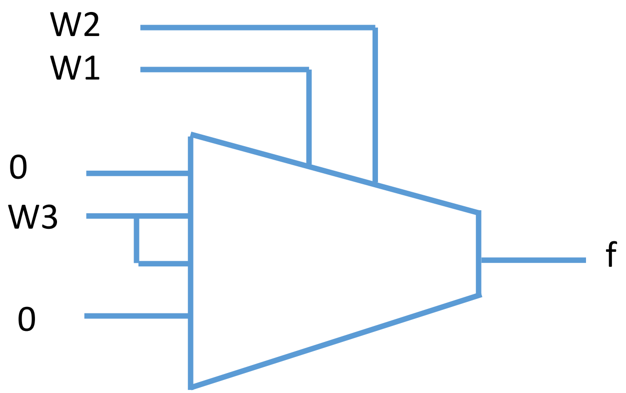
A CLB containing four slices (the number of slices depend on the FPGA family)

Example (1):-

Implement the following using 4X1 multiplexer and any any additional logic gates, using w1w2 as a selector?

W1	W2	W3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

W1	W2	f
0	0	0
0	1	W3
1	0	W3
1	1	0



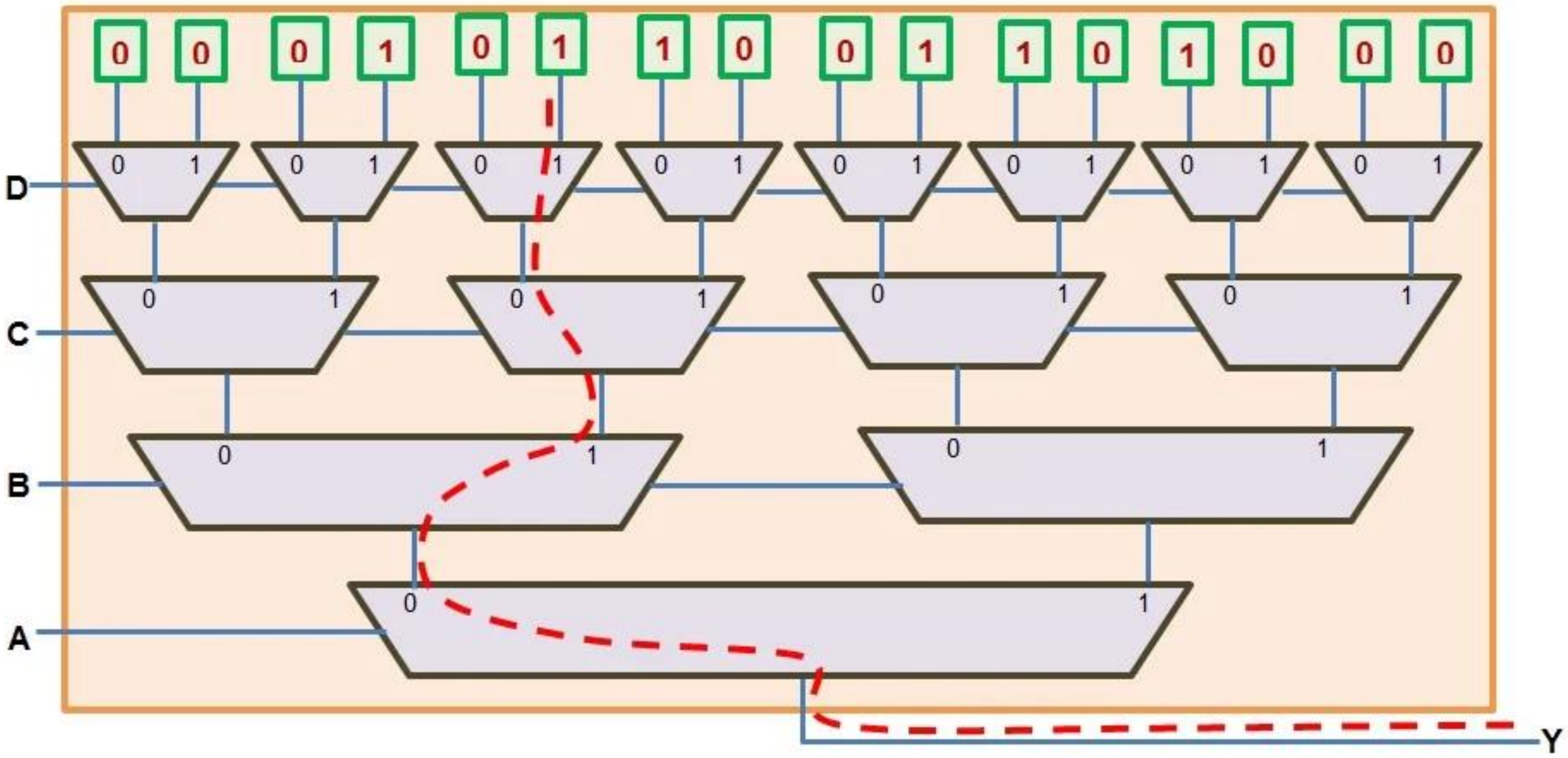
Implementation of solution of example -1-

Example (2):-

Implement the following using 2X1 multiplexer?

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

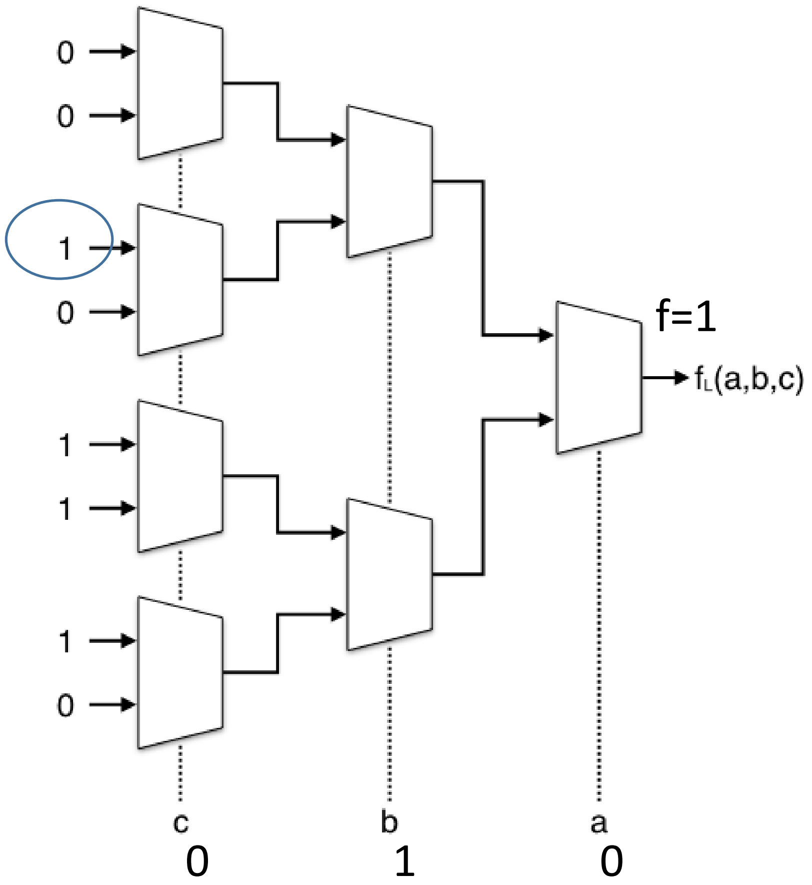
LooK UP-table

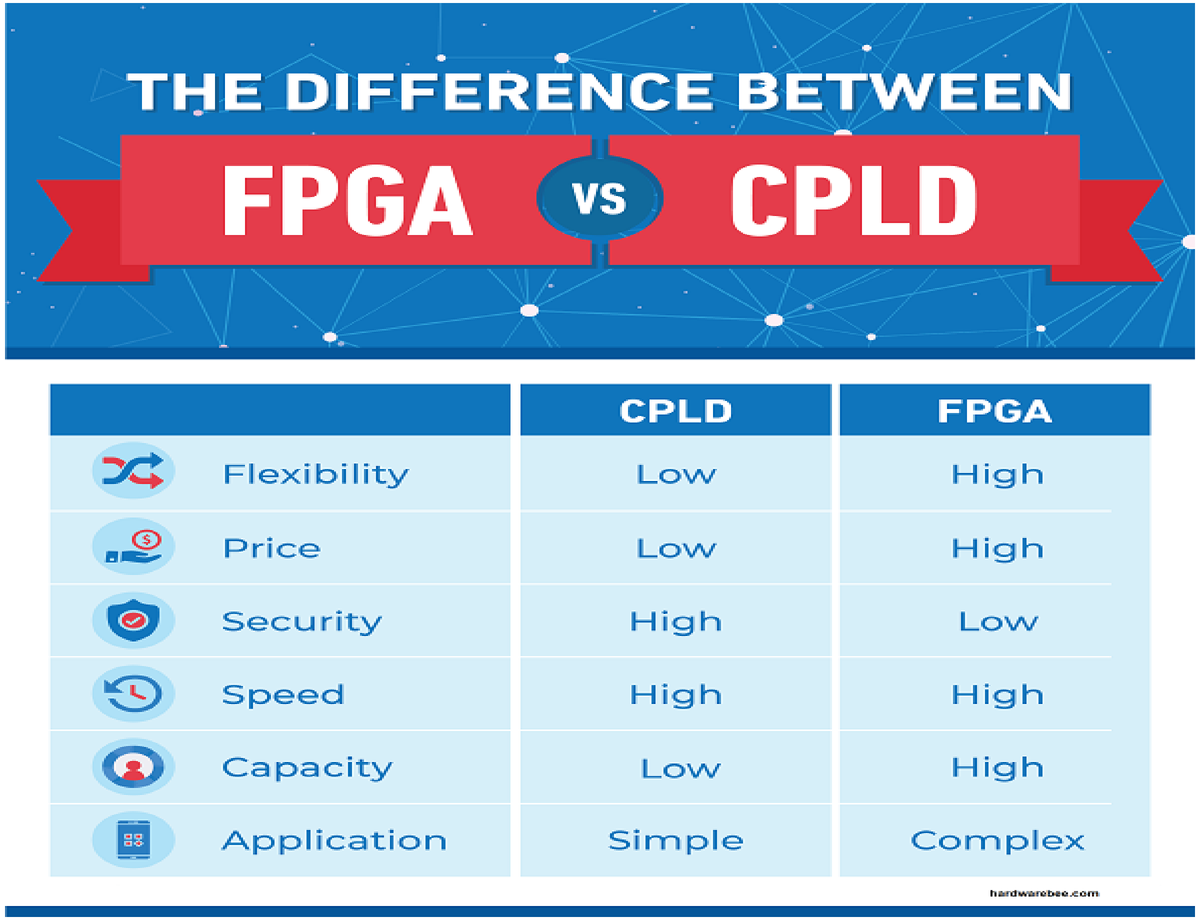


Example (3):-

Implement the following using 2X1 multiplexer?

a	b	c	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0





THE DIFFERENCE BETWEEN

FPGA VS CPLD

	CPLD	FPGA
Flexibility	Low	High
Price	Low	High
Security	High	Low
Speed	High	High
Capacity	Low	High
Application	Simple	Complex

hardwarebee.com

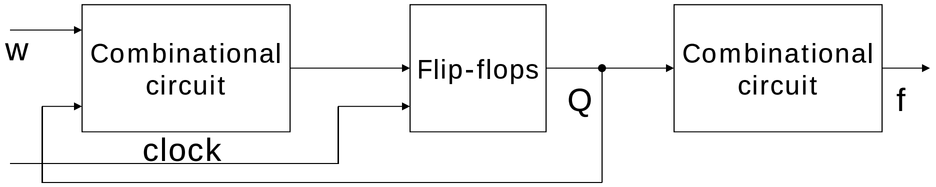
ECE380 Digital Logic

Synchronous Sequential Circuits:
State Diagrams, State Tables

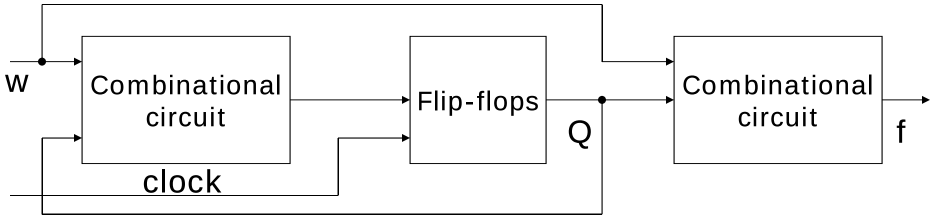
Synchronous sequential circuits

- Circuits where a clock signal is used to control operation are called ***synchronous sequential circuits***
  - The term ***active clock edge*** refers to the clock edge that causes a change in state (positive or negative)
- Realized using combinational logic and one or more flip-flops
- Two models for synchronous sequential circuits
  - **Moore model**: circuit outputs depend only on the present state of the circuit
  - **Mealy model**: circuit outputs depend on the present state of the circuit and the primary inputs
- Sequential circuits are also called ***finite state machines*** (**FSM**)

Moore versus Mealy machines



Moore state machine



Mealy state machine

Basic design steps

- We will introduce techniques for sequential circuit design via a simple example
- Design a circuit that meets the following specifications:
  - The circuit has one input, ***w***, and one output, ***z***
  - All changes in the circuit occur on the positive edge of the clock signal
  - Output ***z***=1 if the input ***w*** was 1 during the two immediately preceding clock cycles
- From this specification it is obvious that ***z*** cannot depend solely of the value of ***w***

Sequences of signals

- The example input and output sequence below aides in the description of the circuit

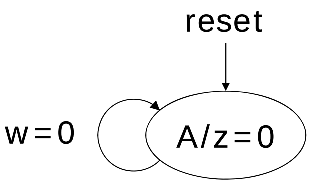
Clock cycle	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
w	0	1	0	1	1	0	1	1	1	0	1
z	0	0	0	0	0	1	0	0	1	1	0

State diagram

- The first step in designing an FSM is determining how many states are needed and which transitions are possible from one state to another
  - No preset procedure for this
  - The designer must think about what the circuit is to accomplish
- A good beginning is to define a **reset** state that the circuit should enter when power is applied or when a reset signal is received

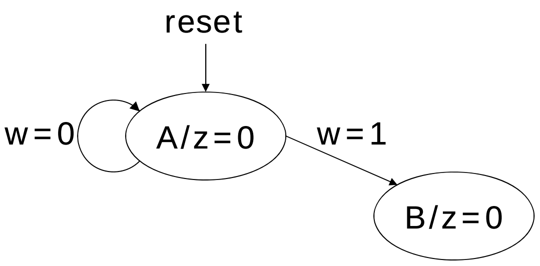
State diagram

- For our example, assume the starting state is called A
- As long as w=0, the circuit should do nothing and z=0



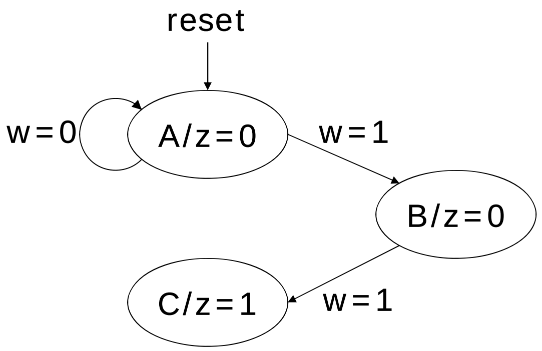
State diagram

- When w=1, the circuit should ‘remember’ this by transitioning to a new state (**B**)
- This transition should occur at the next positive edge of the clock signal

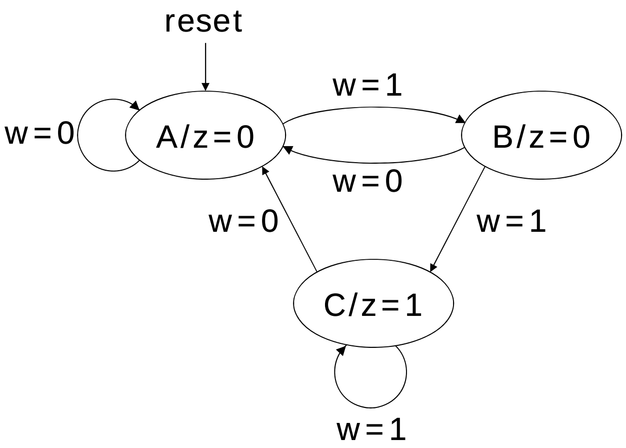


State diagram

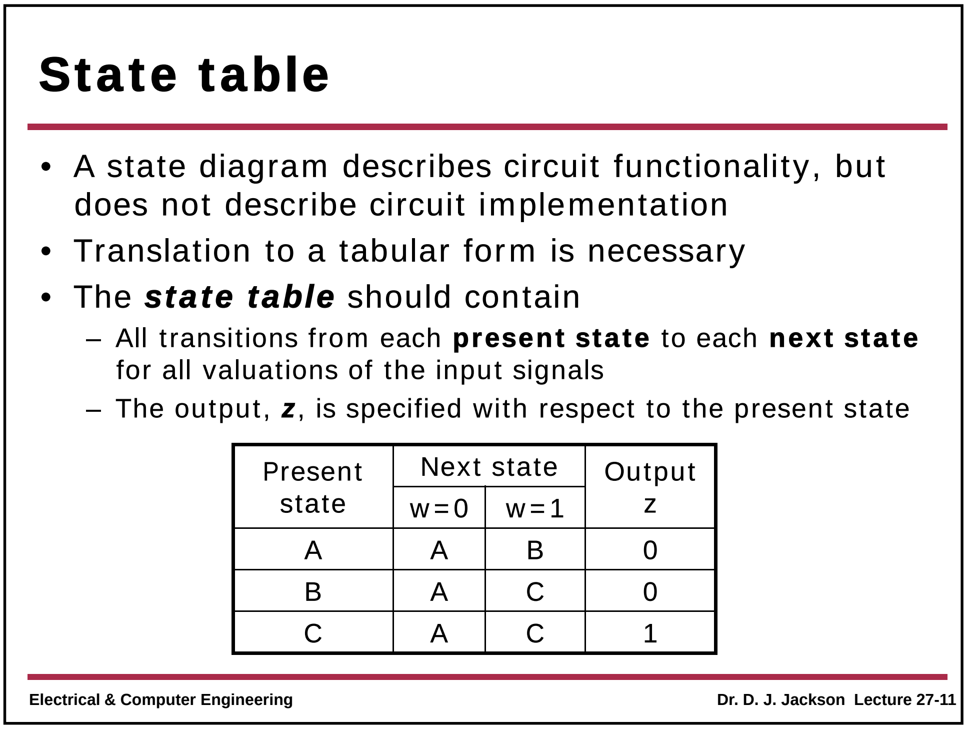
- When in state **B** and w=1, the circuit should 'remember' this by transitioning to a new state (**C**)



Complete state diagram



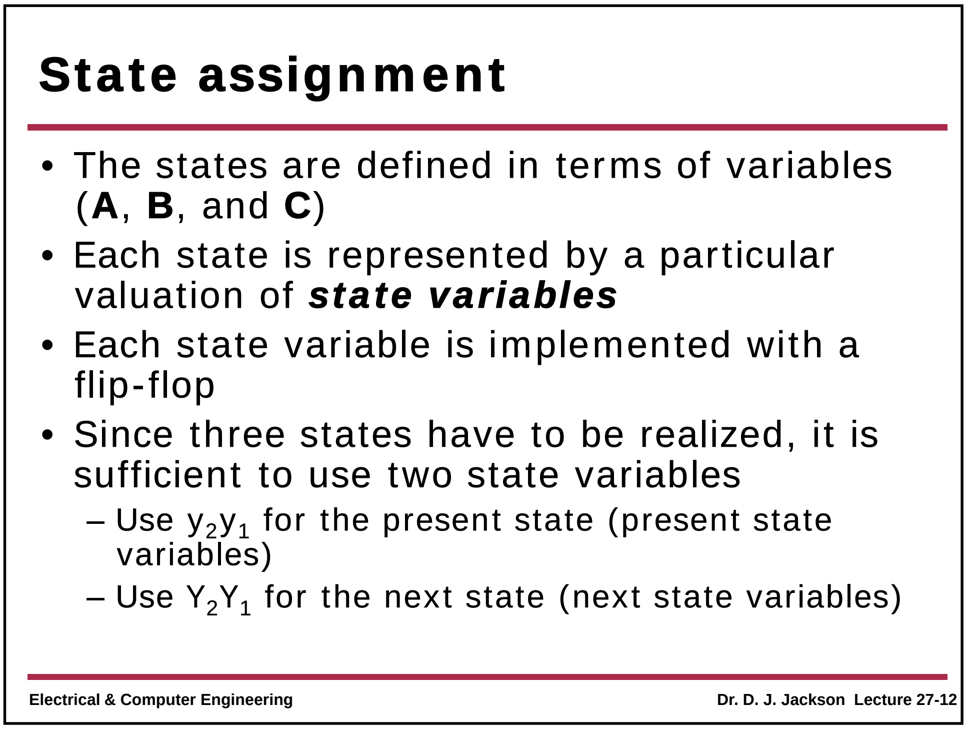
Moore model state diagram



State table

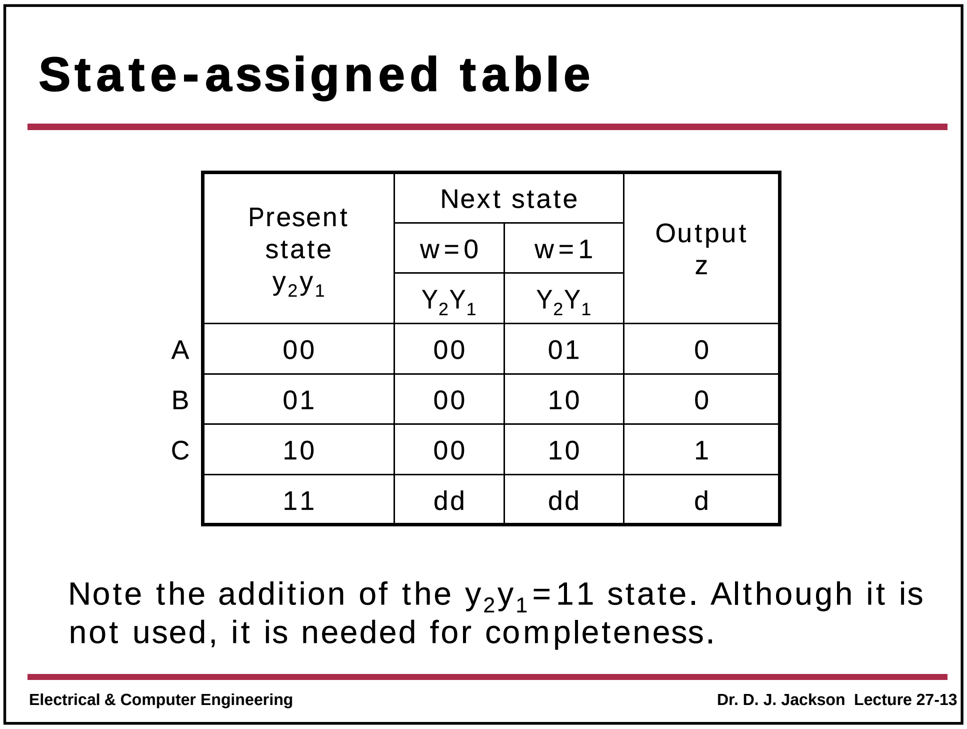
- A state diagram describes circuit functionality, but does not describe circuit implementation
- Translation to a tabular form is necessary
- The ***state table*** should contain
  - All transitions from each **present state** to each **next state** for all valuations of the input signals
  - The output, ***z***, is specified with respect to the present state

Present state	Next state		Output z
	w=0	w=1	
A	A	B	0
B	A	C	0
C	A	C	1



State assignment

- The states are defined in terms of variables (**A**, **B**, and **C**)
- Each state is represented by a particular valuation of ***state variables***
- Each state variable is implemented with a flip-flop
- Since three states have to be realized, it is sufficient to use two state variables
  - Use y_2y_1 for the present state (present state variables)
  - Use Y_2Y_1 for the next state (next state variables)



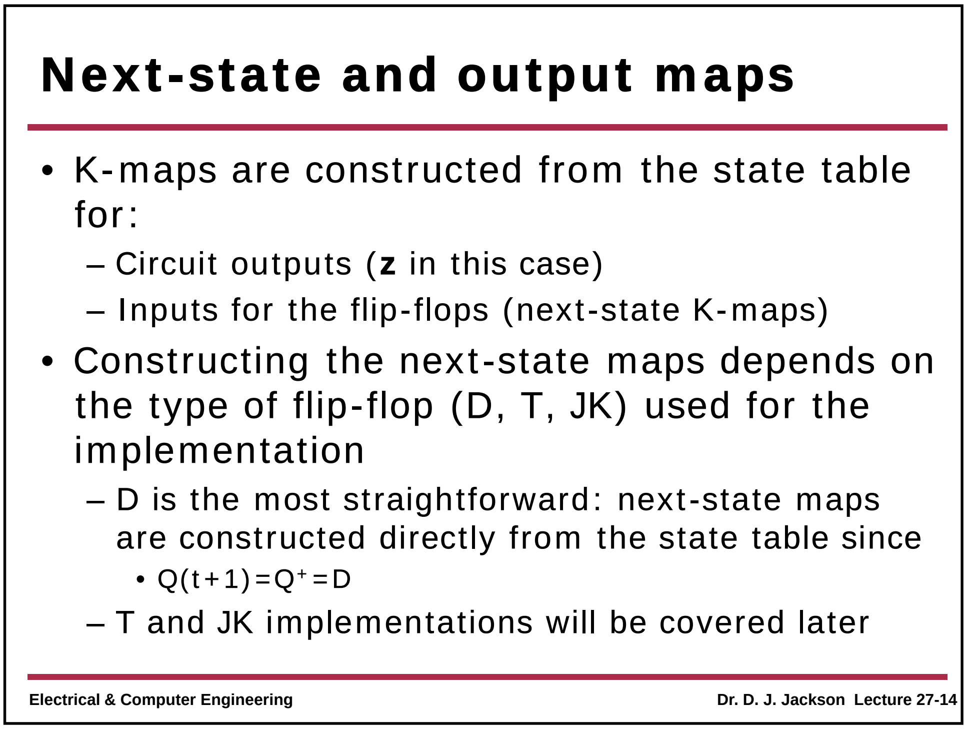
State-assigned table

	Present state y_2y_1	Next state		Output z
		$w=0$	$w=1$	
		Y_2Y_1	Y_2Y_1	
A	00	00	01	0
B	01	00	10	0
C	10	00	10	1
	11	dd	dd	d

Note the addition of the $y_2y_1=11$ state. Although it is not used, it is needed for completeness.

Electrical & Computer Engineering

Dr. D. J. Jackson Lecture 27-13

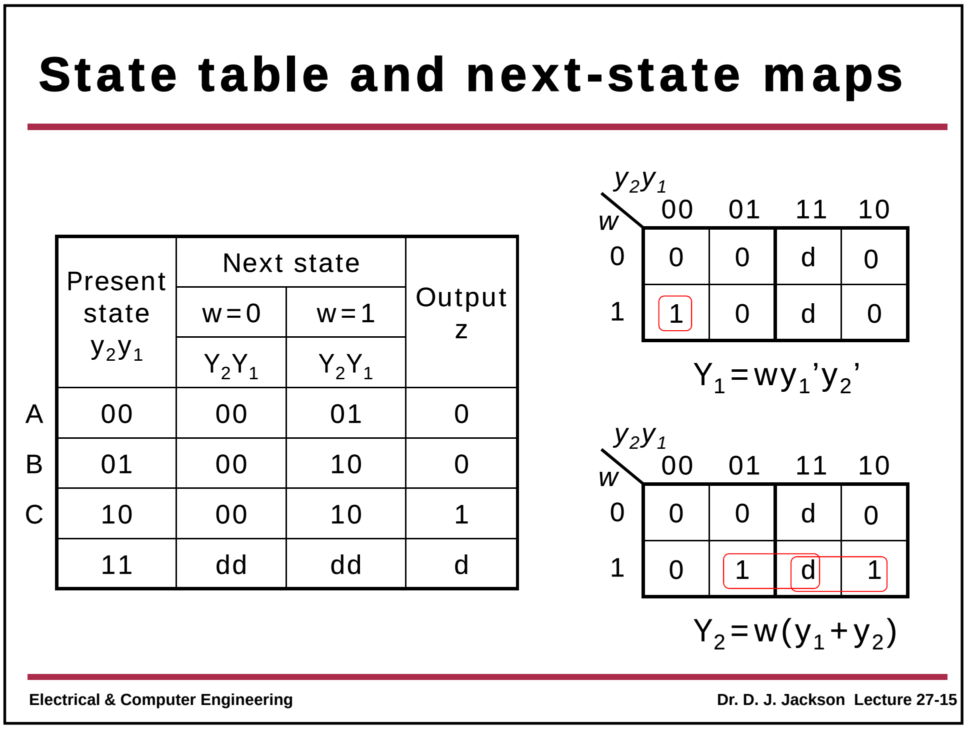


Next-state and output maps

- K-maps are constructed from the state table for:
  - – Circuit outputs (**z** in this case)
  - – Inputs for the flip-flops (next-state K-maps)
- Constructing the next-state maps depends on the type of flip-flop (D, T, JK) used for the implementation
  - – D is the most straightforward: next-state maps are constructed directly from the state table since
    - $Q(t+1)=Q^{+}=D$
  - – T and JK implementations will be covered later

Electrical & Computer Engineering

Dr. D. J. Jackson Lecture 27-14



State table and next-state maps

	Present state y_2y_1	Next state w=0	Next state w=1	Output z
		Y_2Y_1	Y_2Y_1	
A	00	00	01	0
B	01	00	10	0
C	10	00	10	1
	11	dd	dd	d

$w \backslash y_2y_1$	00	01	11	10
0	0	0	d	0
1	1	0	d	0

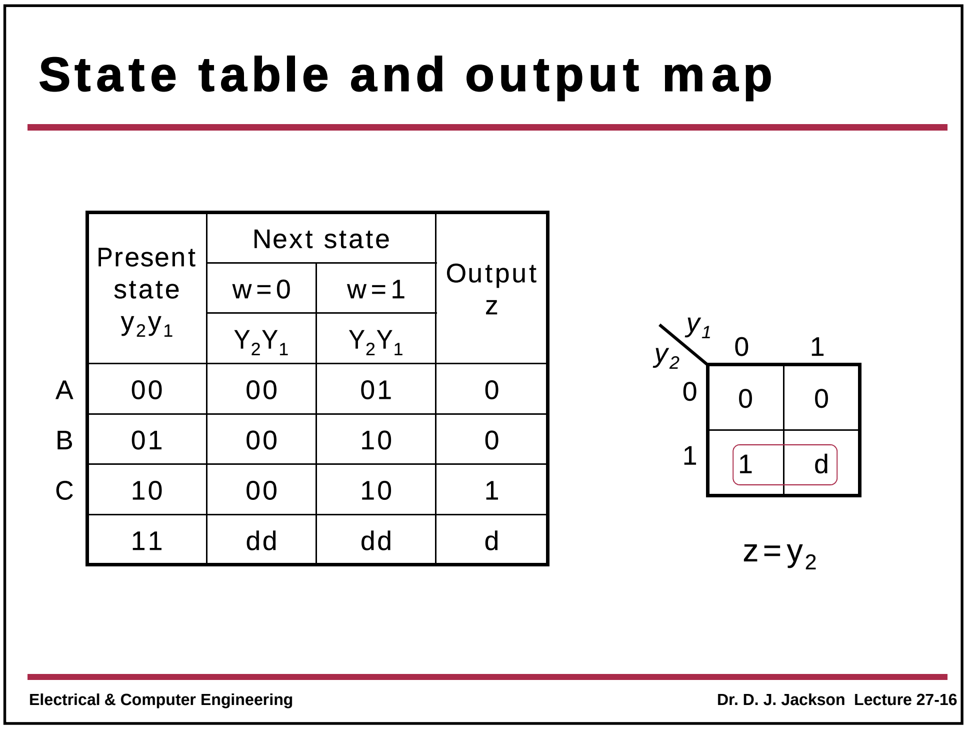
$Y_1 = wy_1'y_2'$

$w \backslash y_2y_1$	00	01	11	10
0	0	0	d	0
1	0	1	d	1

$Y_2 = w(y_1 + y_2)$

Electrical & Computer Engineering

Dr. D. J. Jackson Lecture 27-15



State table and output map

	Present state y_2y_1	Next state w=0	Next state w=1	Output z
		Y_2Y_1	Y_2Y_1	
A	00	00	01	0
B	01	00	10	0
C	10	00	10	1
	11	dd	dd	d

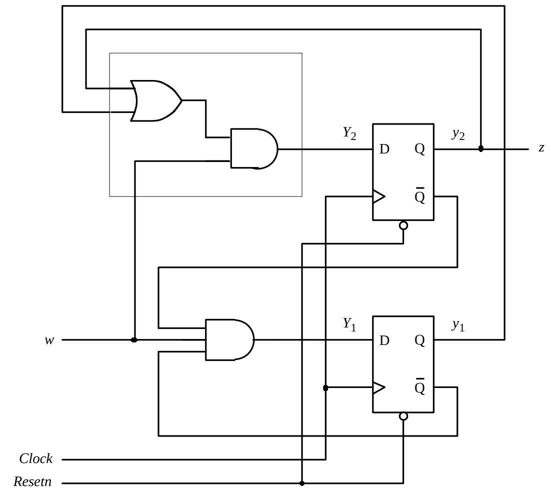
$y_2 \backslash y_1$	0	1
0	0	0
1	1	d

$z = y_2$

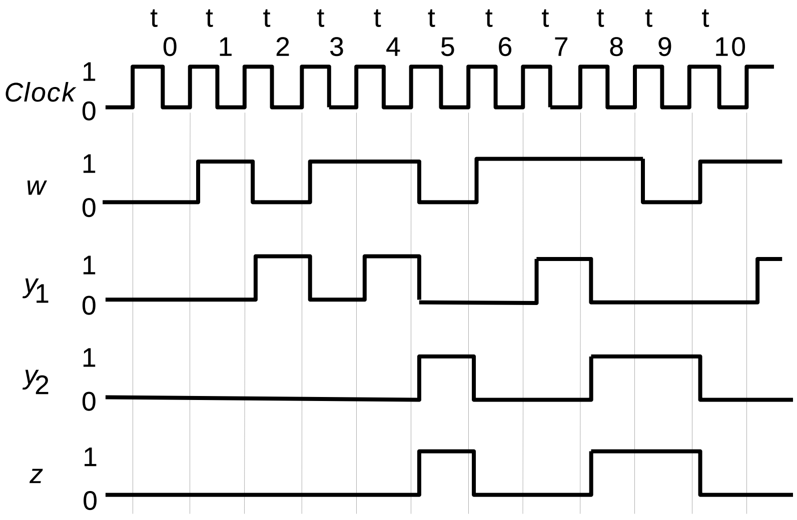
Electrical & Computer Engineering

Dr. D. J. Jackson Lecture 27-16

Circuit diagram



Timing diagram



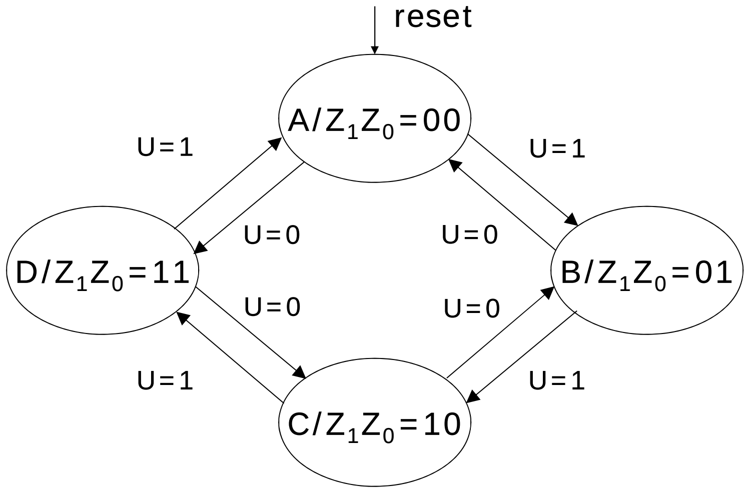
ECE380 Digital Logic

Synchronous Sequential Circuits:
Implementations using D-type,
T-type and JK-type Flip-Flops

Counter design example

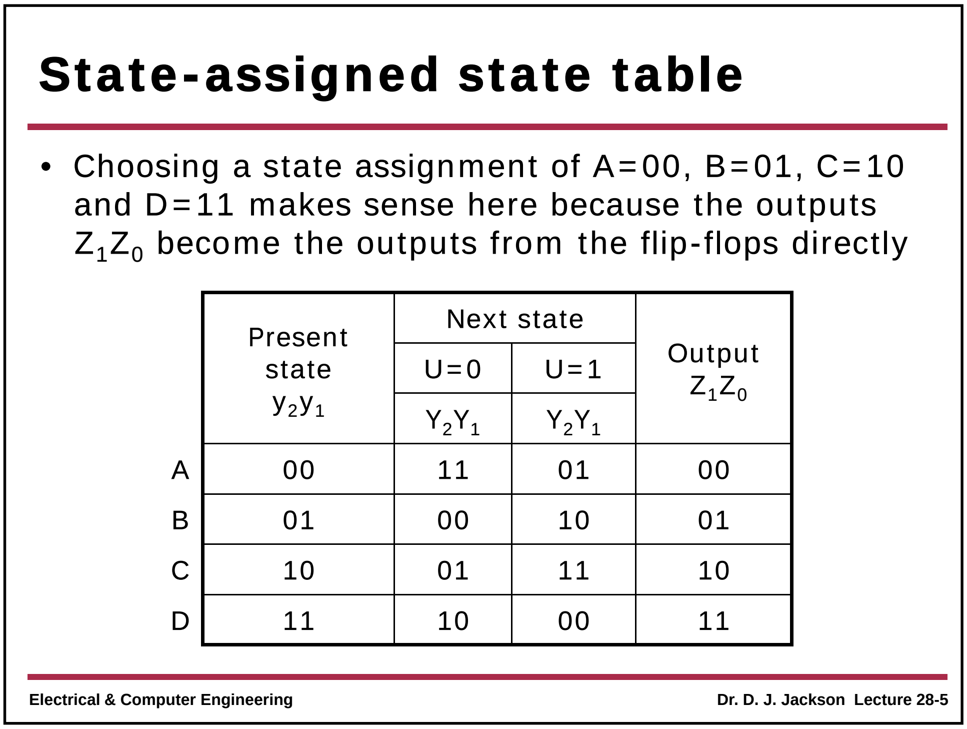
- Design a 2-bit counter that counts
  - in the sequence 0,1,2,3,0,… if a given control signal U=1, or
  - in the sequence 0,3,2,1,0,… if a given control signal U=0
- This represents a 2-bit binary up/down counter
  - An input U to control to count direction
  - A RESET input to reset the counter to the value zero
  - Two outputs (Z_1Z_0) representing the output (0-3)
  - Counter counts on positive edge transitions of a common clock signal
- Design this counter as a synchronous sequential machine using
  - D-type, T-type, JK-type flip-flops

Counter state diagram



Counter state table

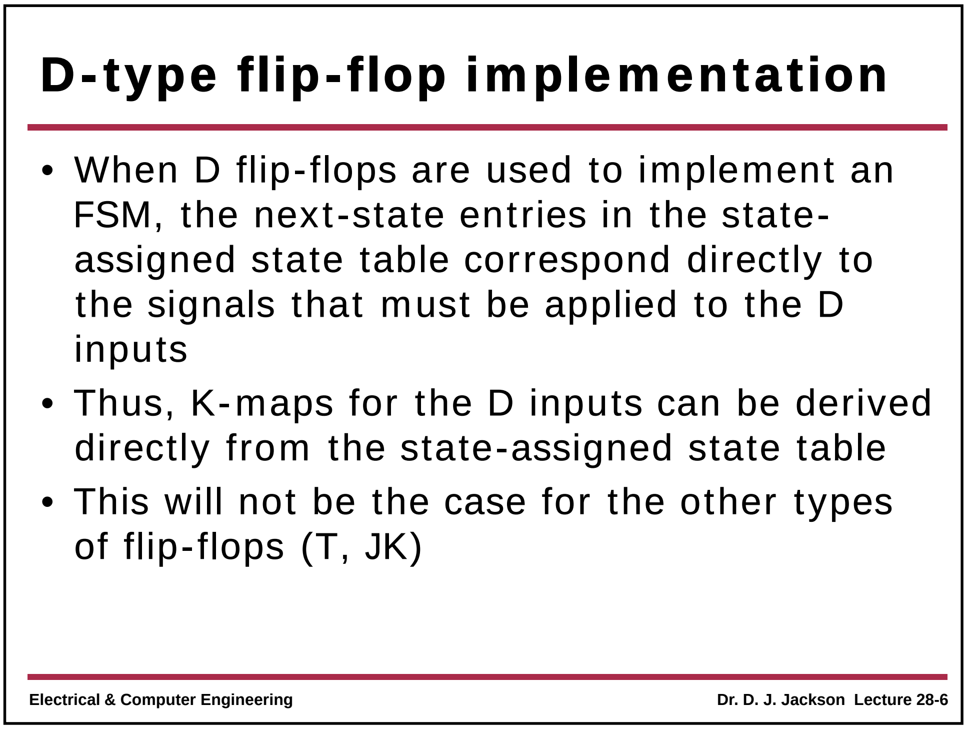
Present state	Next state		Output Z_1Z_0
	U=0	U=1	
A	D	B	00
B	A	C	01
C	B	D	10
D	C	A	11



State-assigned state table

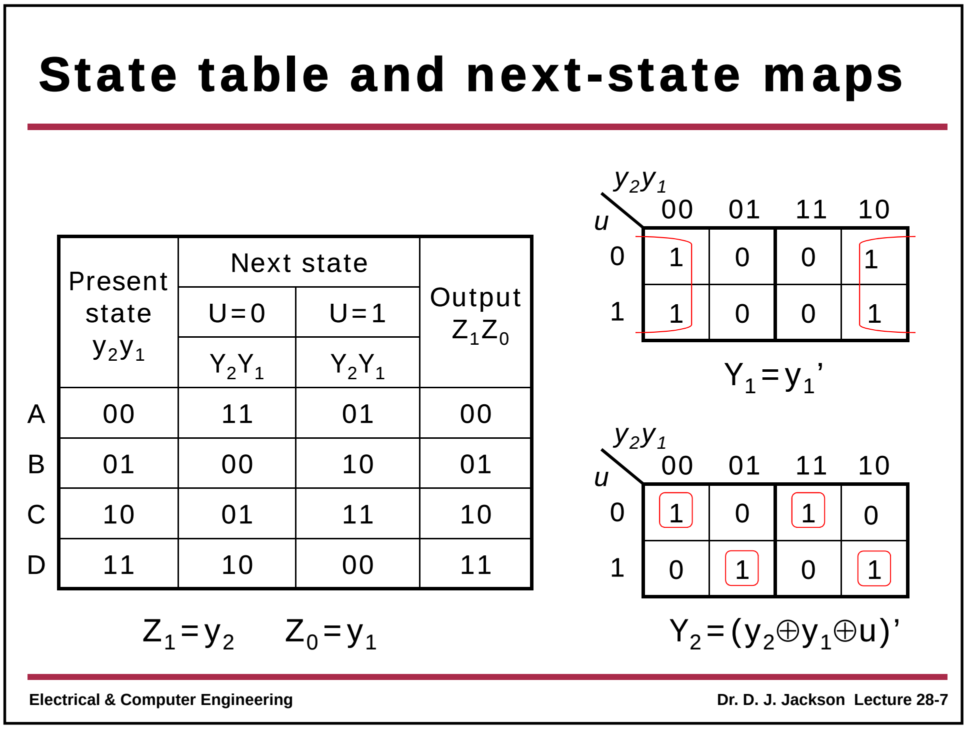
- Choosing a state assignment of A=00, B=01, C=10 and D=11 makes sense here because the outputs Z_1Z_0 become the outputs from the flip-flops directly

	Present state y_2y_1	Next state		Output Z_1Z_0
		U=0	U=1	
		Y_2Y_1	Y_2Y_1	
A	00	11	01	00
B	01	00	10	01
C	10	01	11	10
D	11	10	00	11



D-type flip-flop implementation

- When D flip-flops are used to implement an FSM, the next-state entries in the state-assigned state table correspond directly to the signals that must be applied to the D inputs
- Thus, K-maps for the D inputs can be derived directly from the state-assigned state table
- This will not be the case for the other types of flip-flops (T, JK)



State table and next-state maps

	Present state y_2y_1	Next state U=0 Y_2Y_1	Next state U=1 Y_2Y_1	Output Z_1Z_0
A	00	11	01	00
B	01	00	10	01
C	10	01	11	10
D	11	10	00	11

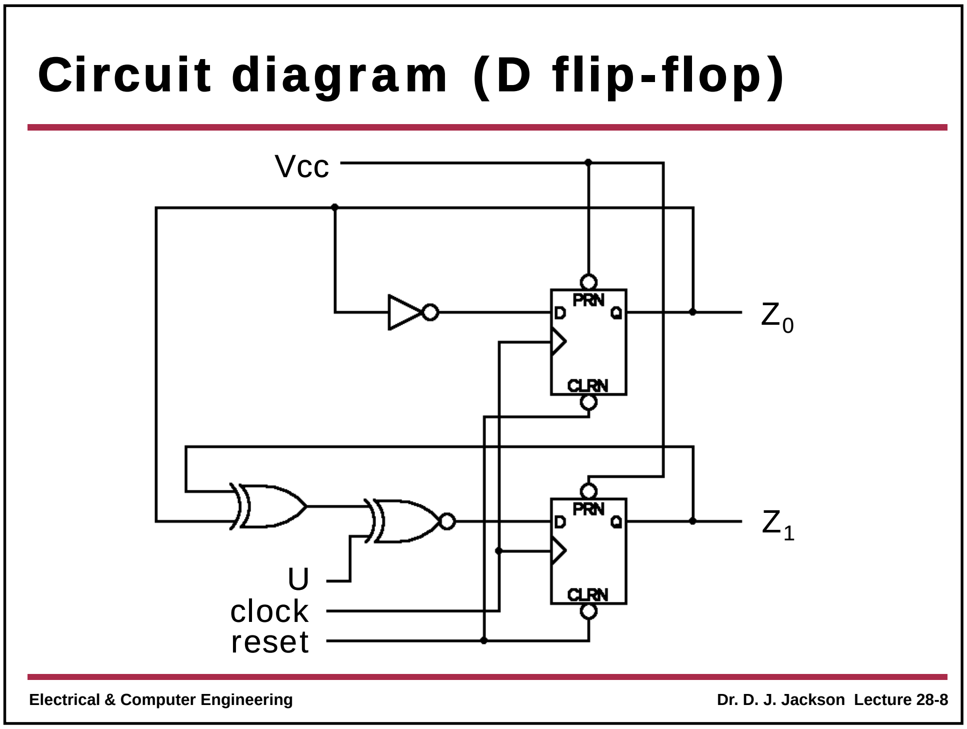
$Z_1=y_2$ $Z_0=y_1$

$u \backslash y_2y_1$	00	01	11	10
0	1	0	0	1
1	1	0	0	1

$Y_1=y_1'$

$u \backslash y_2y_1$	00	01	11	10
0	1	0	1	0
1	0	1	0	1

$Y_2=(y_2\oplus y_1\oplus u)'$



Circuit diagram (D flip-flop)

Design using other flip-flop types

- For the T- or JK-type flip-flops, we must derive the desired inputs to the flip-flops
- Begin by constructing a **transition table** for the flip-flop type you wish to use
  - This table simply lists required inputs for a given change of state
- The transition table is used with the state-assigned state table to construct an **excitation table**
  - The excitation table lists the required flip-flop inputs that must be 'excited' to cause a transition to the next state

Transition tables

J	K	Q	Q^+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Q	Q^+	J	K
0	0	0	D
0	1	1	D
1	0	D	1
1	1	D	0

JK transition table

T	Q	Q^+
0	0	0
0	1	1
1	0	1
1	1	0

Q	Q^+	T
0	0	0
0	1	1
1	0	1
1	1	0

T transition table

The transition table lists required flip-flop inputs to affect a specific change

T-type flip-flop implementation

Use entries from the transition table to derive the flip-flop inputs based on the state-assigned state table.

Q	Q_+	T
0	0	0
0	1	1
1	0	1
1	1	0

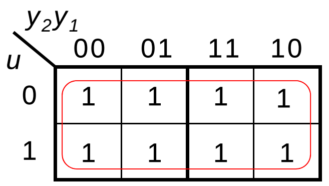
excitation table

Present state y_2y_1	Flip-flop inputs				Output Z_1Z_0
	U=0		U=1		
	Y_2Y_1	T_2T_1	Y_2Y_1	T_2T_1	
00	11	11	01	01	00
01	00	01	10	11	01
10	01	11	11	01	10
11	10	01	00	11	11

Excitation table and K-maps

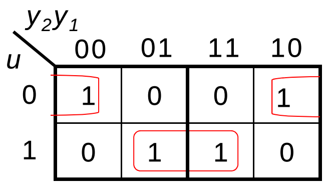
Present state y_2y_1	Flip-flop inputs		Output Z_1Z_0
	U=0	U=1	
	T_2T_1	T_2T_1	
00	11	01	00
01	01	11	01
10	11	01	10
11	01	11	11

$Z_1=y_2$ $Z_0=y_1$



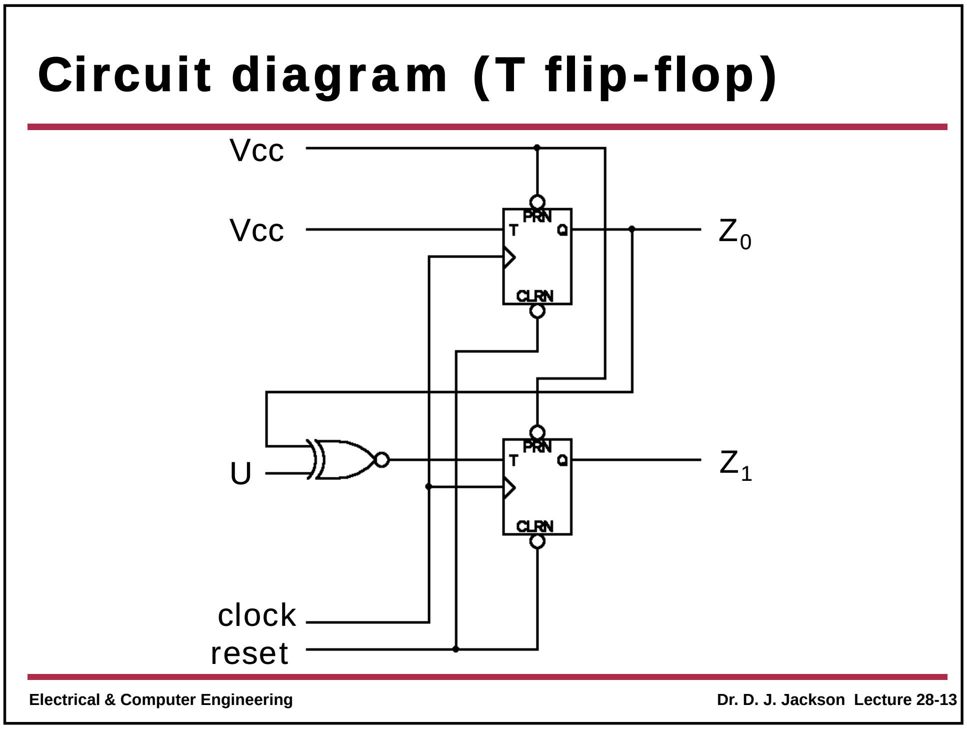
u \ y_2y_1	00	01	11	10
0	1	1	1	1
1	1	1	1	1

$T_1=1$

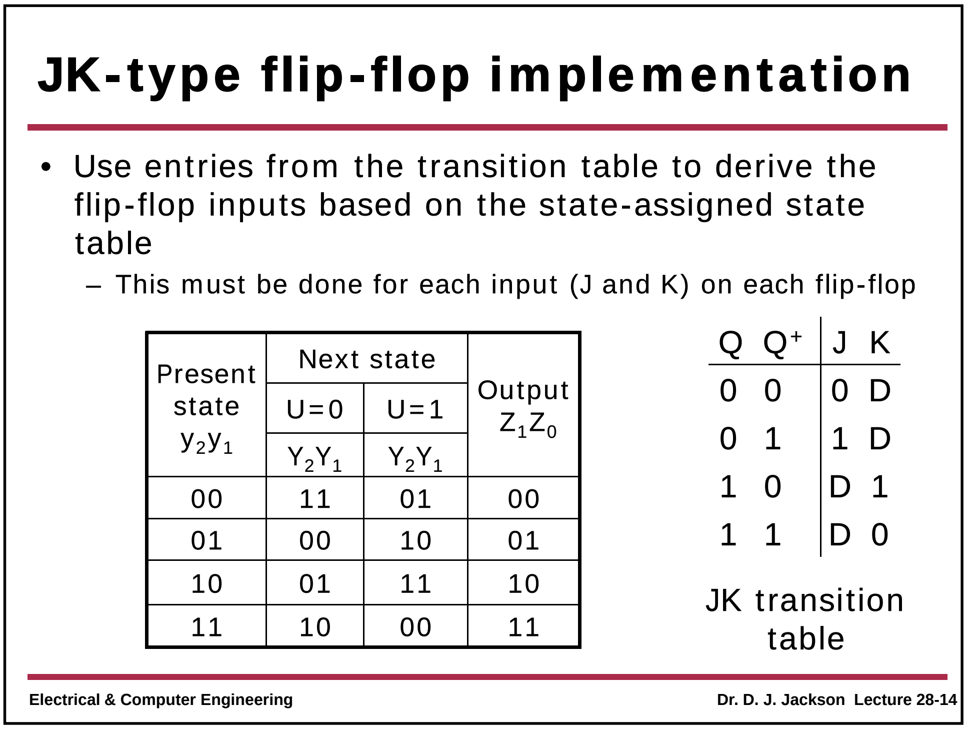


u \ y_2y_1	00	01	11	10
0	1	0	0	1
1	0	1	1	0

$T_2=y_1u+y_1'u'=(y_1\oplus u)'$



Circuit diagram (T flip-flop)



JK-type flip-flop implementation

- Use entries from the transition table to derive the flip-flop inputs based on the state-assigned state table
  - This must be done for each input (J and K) on each flip-flop

Present state y_2y_1	Next state		Output Z_1Z_0
	U=0	U=1	
	Y_2Y_1	Y_2Y_1	
00	11	01	00
01	00	10	01
10	01	11	10
11	10	00	11

Q	Q^+	J	K
0	0	0	D
0	1	1	D
1	0	D	1
1	1	D	0

JK transition table

JK-type flip-flop implementation

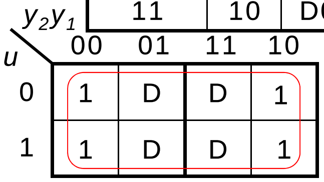
Q	Q^+	J	K
0	0	0	D
0	1	1	D
1	0	D	1
1	1	D	0

JK transition table

Present state y_2y_1	Flip-flop inputs						Output Z_1Z_0
	U=0			U=1			
	Y_2Y_1	J_2K_2	J_1K_1	Y_2Y_1	J_2K_2	J_1K_1	
00	11	1D	1D	01	0D	1D	00
01	00	0D	D1	10	1D	D1	01
10	01	D1	1D	11	D0	1D	10
11	10	D0	D1	00	D1	D1	11

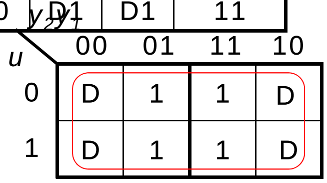
Excitation table and K-maps

Present state y_2y_1	Flip-flop inputs						Output Z_1Z_0
	U=0			U=1			
	Y_2Y_1	J_2K_2	J_1K_1	Y_2Y_1	J_2K_2	J_1K_1	
00	11	1D	1D	01	0D	1D	00
01	00	0D	D1	10	1D	D1	01
10	01	D1	1D	11	D0	1D	10
11	10	D0	D1	00	D1	D1	11



u \ y_2y_1	00	01	11	10
0	1	D	D	1
1	1	D	D	1

$J_1=1$



u \ y_2y_1	00	01	11	10
0	D	1	1	D
1	D	1	1	D

$K_1=1$

Excitation table and K-maps

Present state y_2y_1	Flip-flop inputs						Output Z_1Z_0
	U=0			U=1			
	Y_2Y_1	J_2K_2	J_1K_1	Y_2Y_1	J_2K_2	J_1K_1	
00	11	1D	1D	01	0D	1D	00
01	00	0D	D1	10	1D	D1	01
10	01	D1	1D	11	D0	1D	10
11	10	D0	D1	00	D1	D1	11

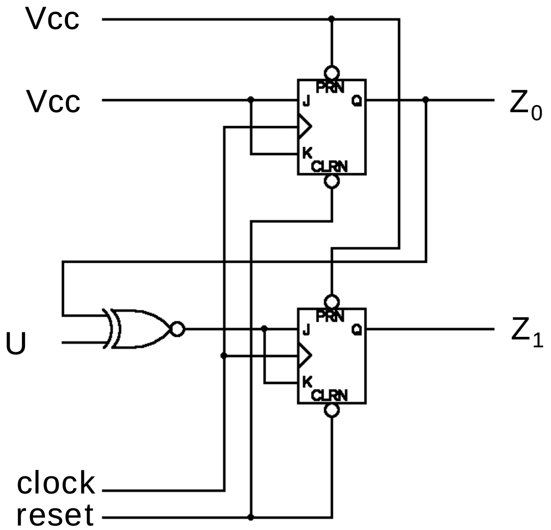
u \ y_2y_1	00	01	11	10
0	1	0	D	D
1	0	1	D	D

$J_2 = (y_1 \oplus u)'$

u \ y_2y_1	00	01	11	10
0	D	D	0	1
1	D	D	1	0

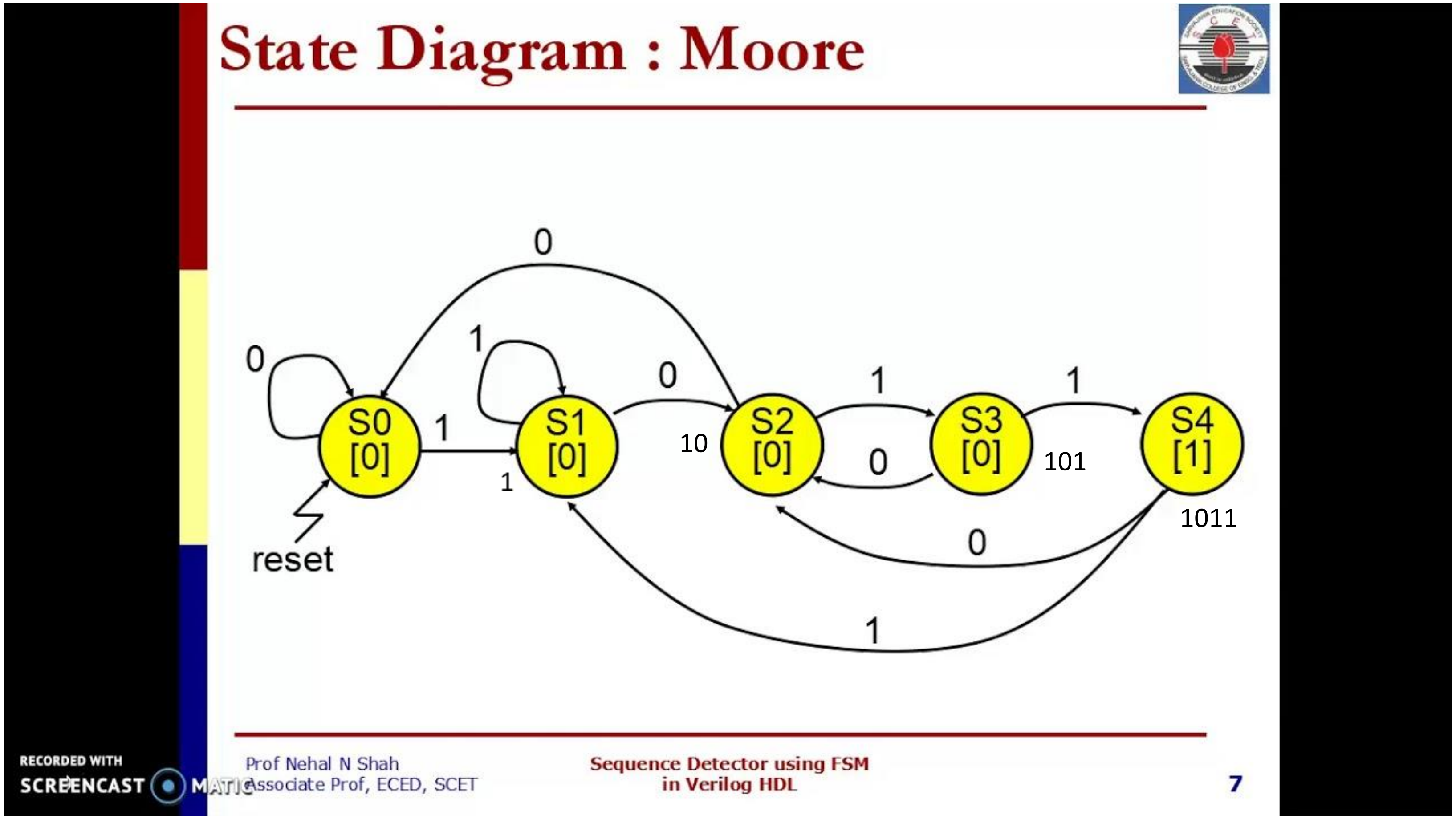
$K_2 = (y_1 \oplus u)'$

Circuit diagram (JK flip-flop)



Lecture Six

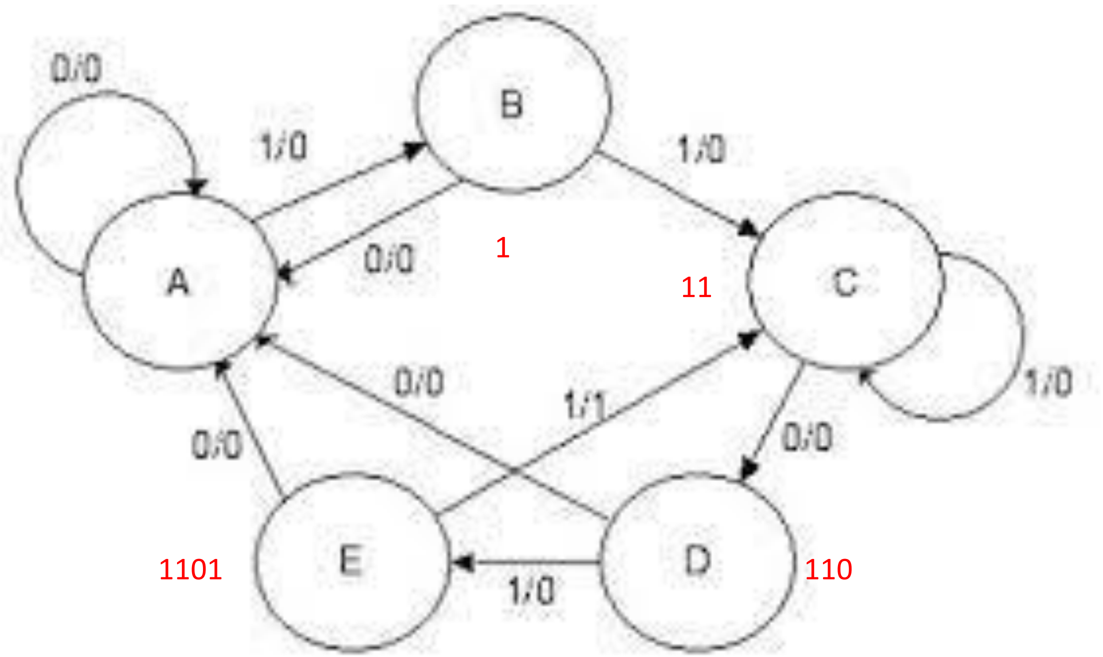
1011
Moore



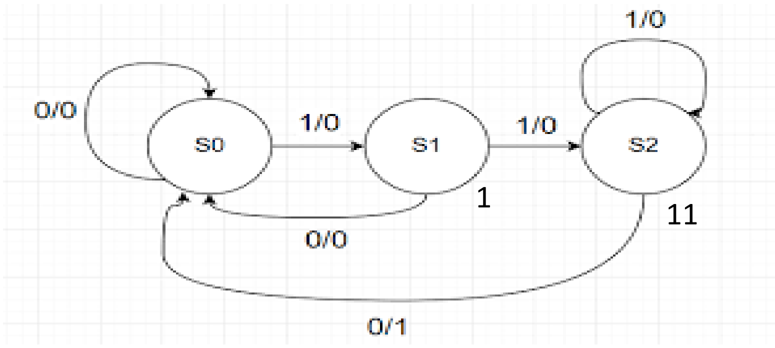
State Diagram : Moore

Prof Nehal N Shah
Associate Prof, ECED, SCET

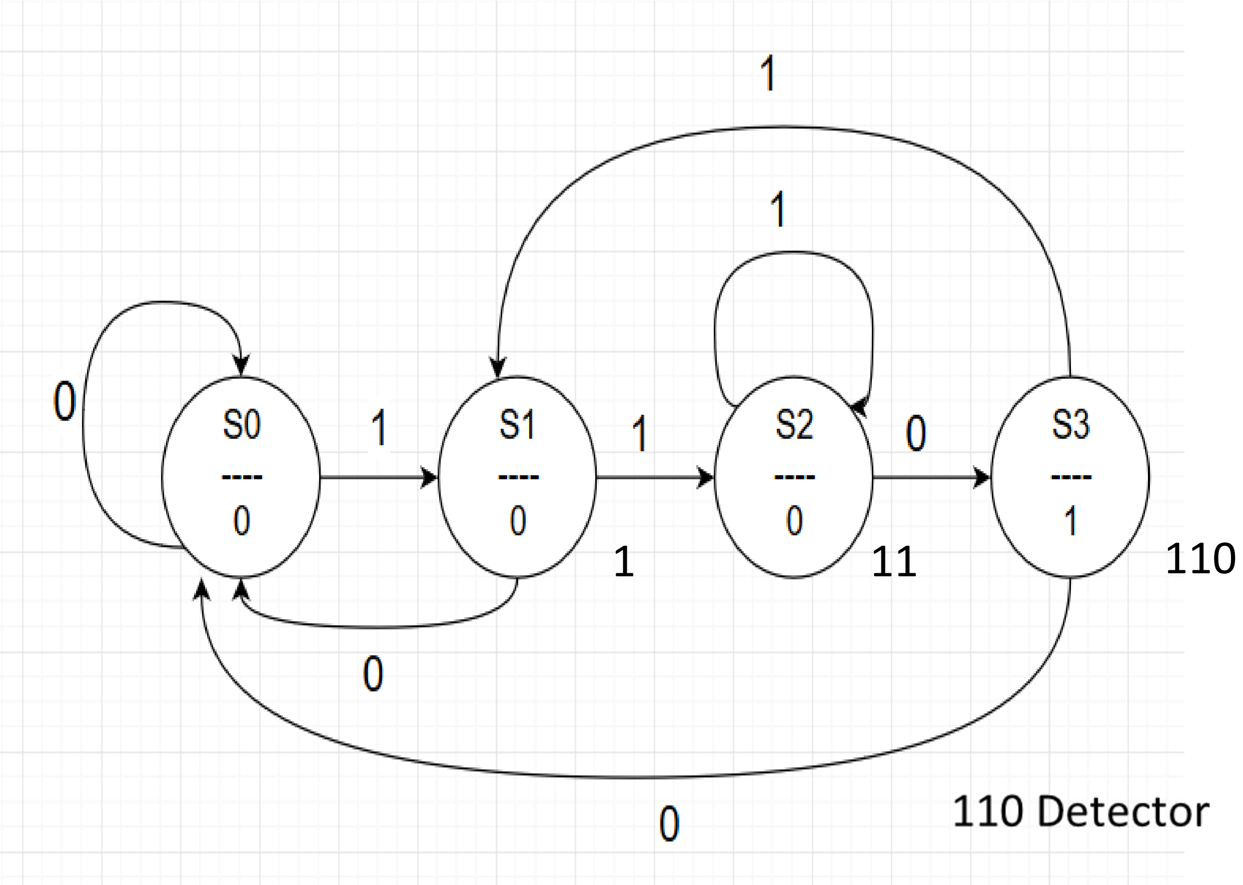
Sequence Detector using FSM
in Verilog HDL



11011 Mealy

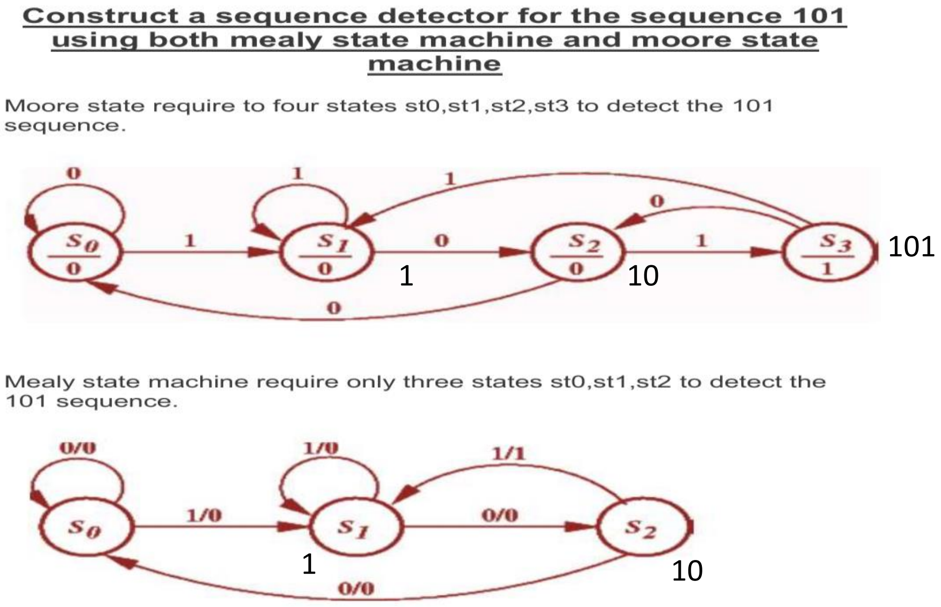


110 Detector
Mealy



110 Detector

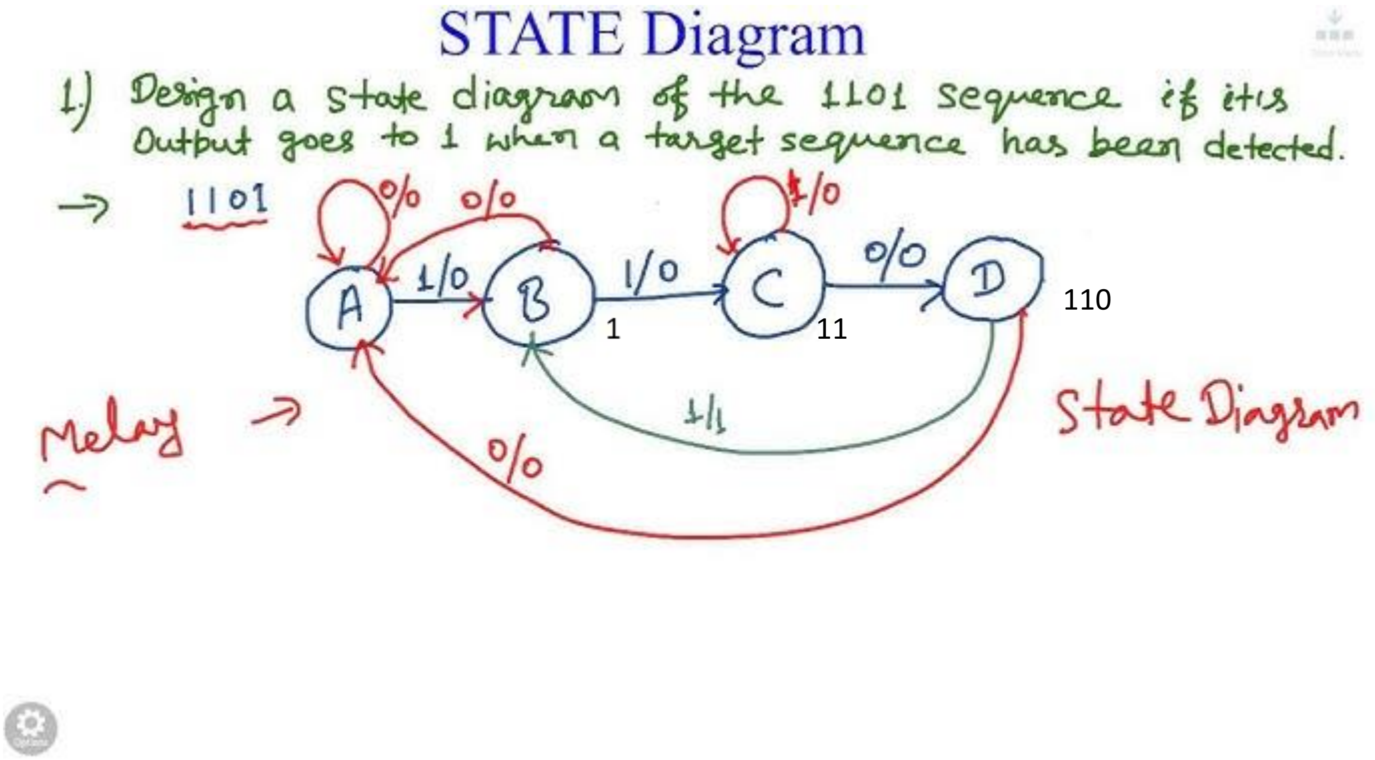
Moore



Construct a sequence detector for the sequence 101 using both mealy state machine and moore state machine

Moore state require to four states st0,st1,st2,st3 to detect the 101 sequence.

Mealy state machine require only three states st0,st1,st2 to detect the 101 sequence.

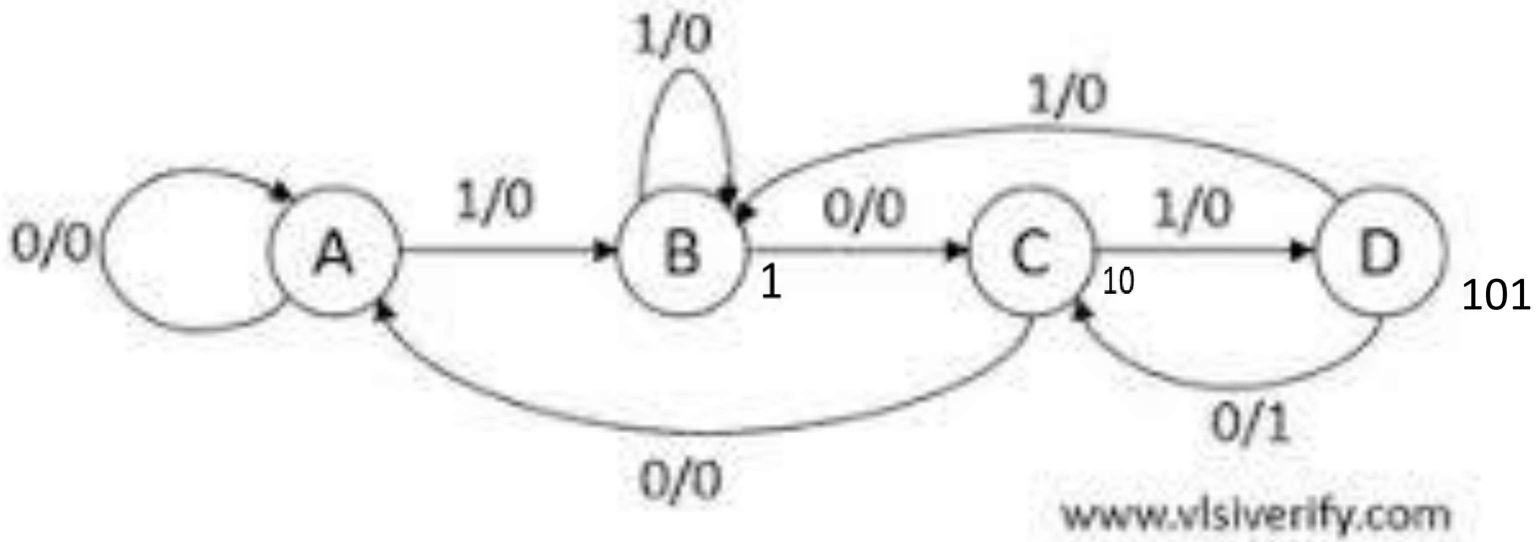


STATE Diagram

1.) Design a state diagram of the 1101 sequence if itis Output goes to 1 when a target sequence has bean detected.

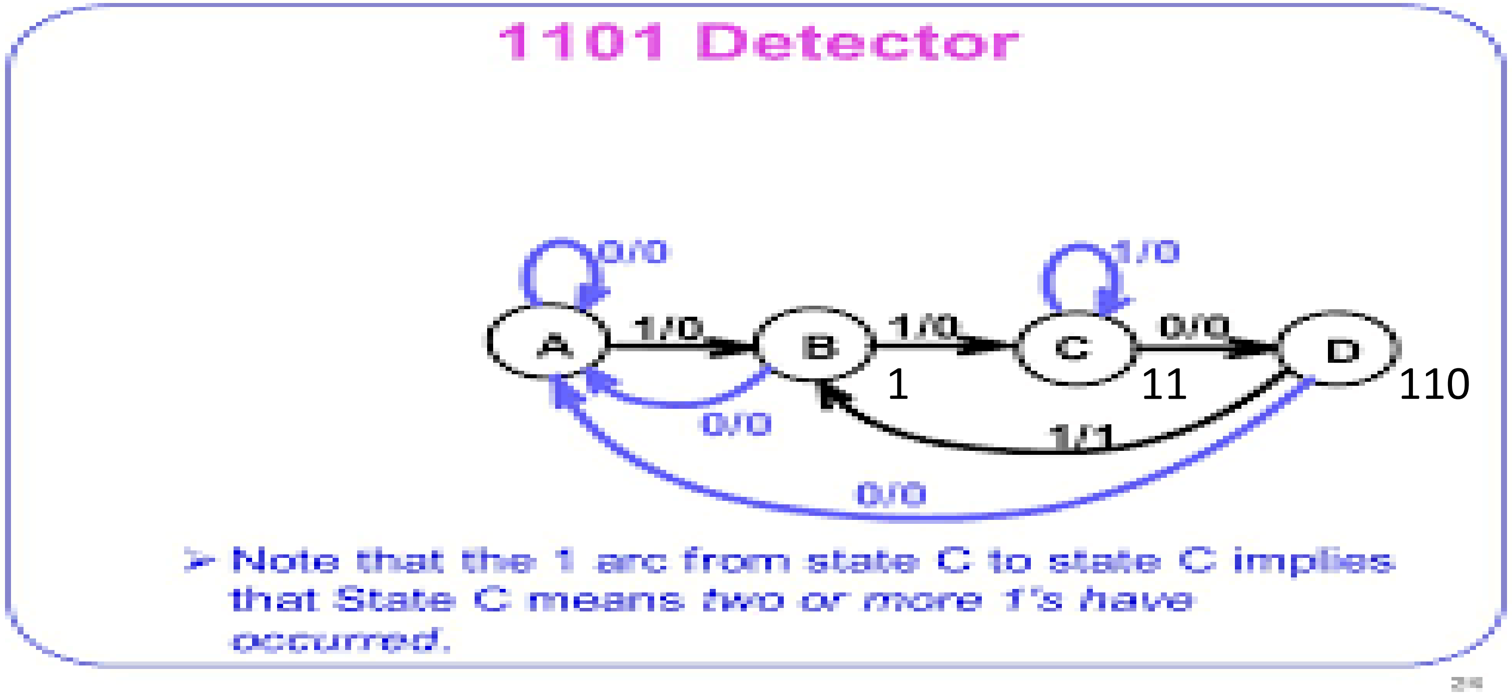
→ 1101

Melay →



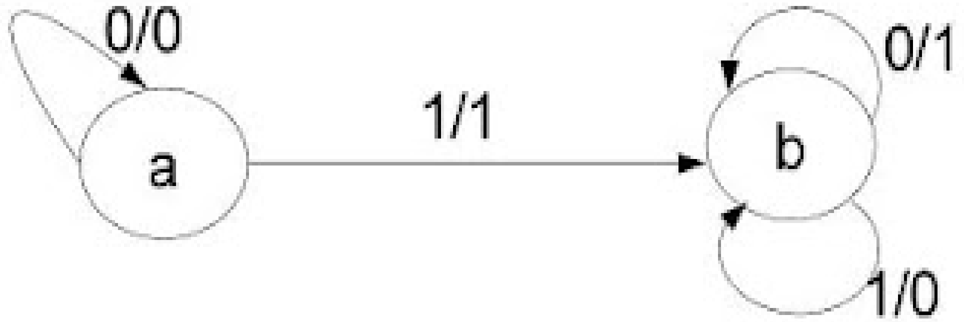
www.vlsiverify.com

1010 Overlapping Mealy Sequence Detector



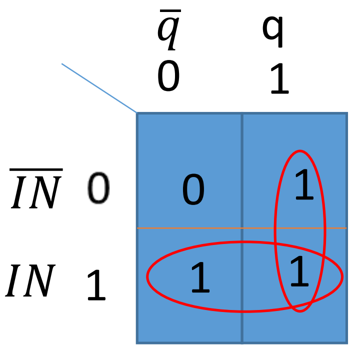
1101 Detector

- Note that the 1 arc from state C to state C implies that State C means *two or more 1's have occurred.*



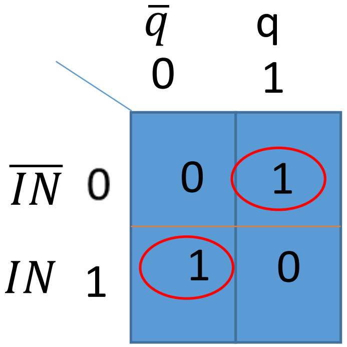
Present State	Next State		Output	
	IN=0	IN=1	IN=0	IN=1
q	Q	Q	Q	Q
a	a	b	0	1
b	B	b	1	0

	Present State	Next State		Output	
		IN=0	IN=1	IN=0	IN=1
	q	Q	Q	Q	Q
a	0	0	1	0	1
b	1	1	1	1	0



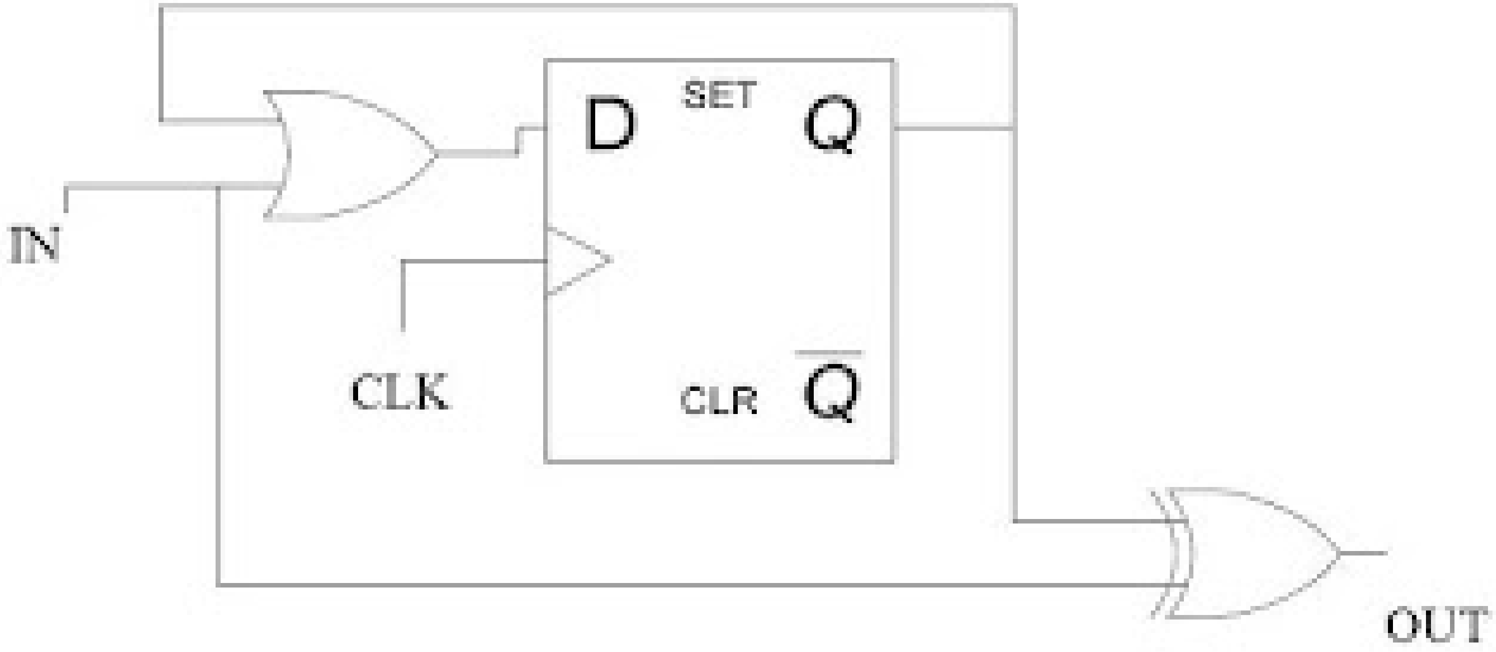
	$\bar{q}$ 0	q 1
$\overline{IN}$ 0	0	1
IN 1	1	1

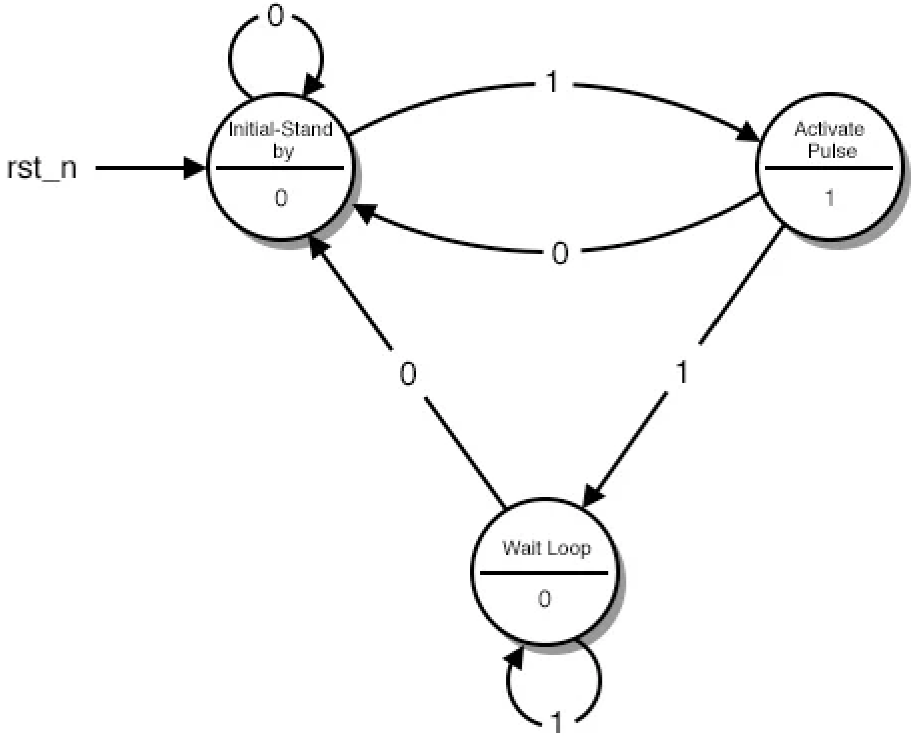
$$Q = IN + q$$



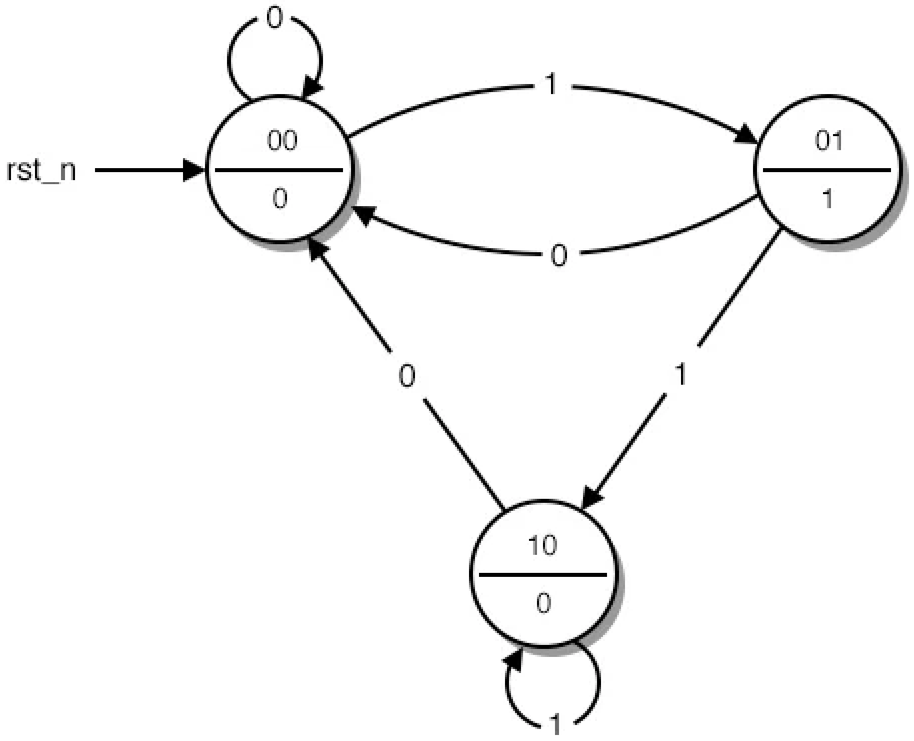
	$\bar{q}$ 0	q 1
$\overline{IN}$ 0	0	1
IN 1	1	0

$$Z = IN \quad XOR \quad q$$

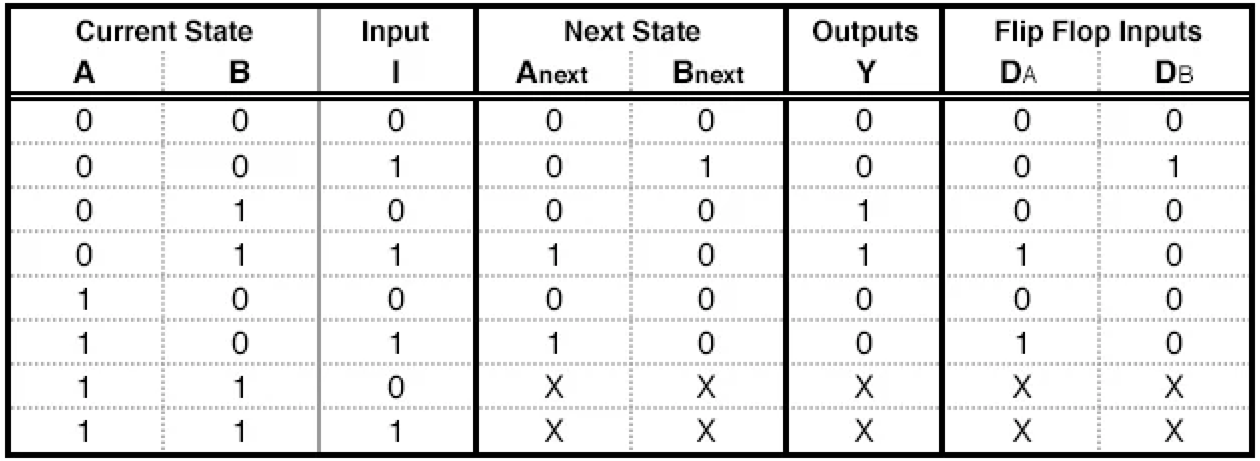

IN
CLK
D
SET
Q
CLR
Q̅
OUT



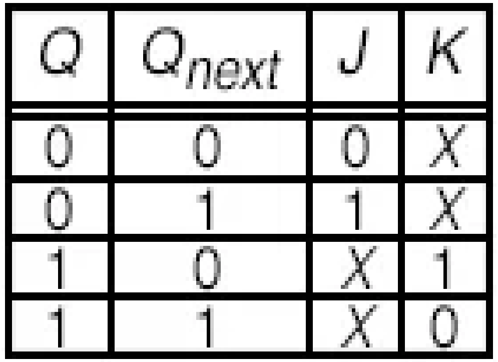
0
1
rst_n
Initial-Stand by
0
Activate Pulse
1
0
0
1
Wait Loop
0
1


0
1
rst_n
00
0
01
1
0
0
1
10
0
1

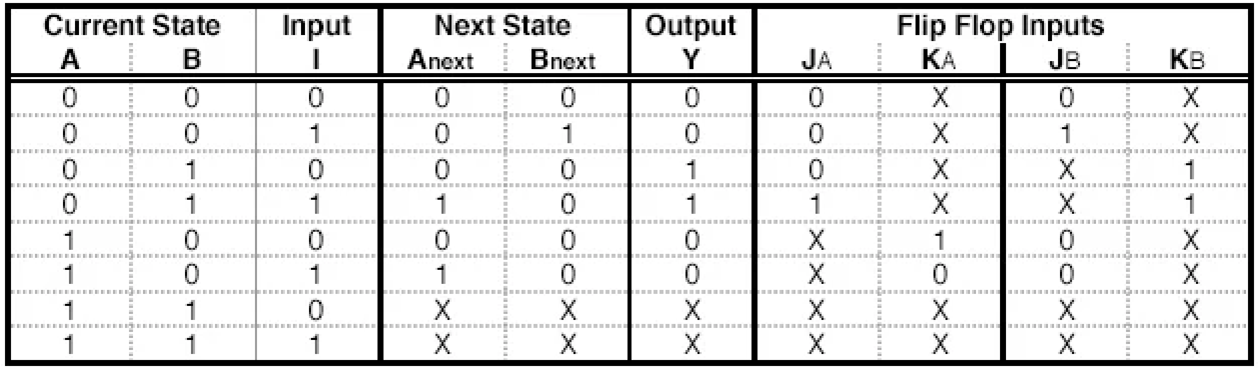
Current State		Input	Next State		Outputs
A	B	I	A_{next}	B_{next}	Y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	0	1
1	0	0	0	0	0
1	0	1	1	0	0
1	1	0	X	X	X
1	1	1	X	X	X



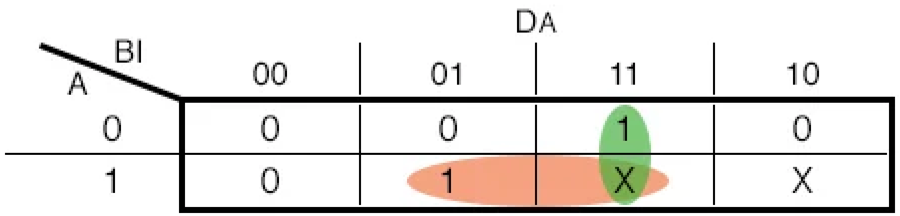
Current State		Input	Next State		Outputs	Flip Flop Inputs	
A	**B**	**I**	**A_{next}**	**B_{next}**	**Y**	**D_A**	**D_B**
0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	1
0	1	0	0	0	1	0	0
0	1	1	1	0	1	1	0
1	0	0	0	0	0	0	0
1	0	1	1	0	0	1	0
1	1	0	X	X	X	X	X
1	1	1	X	X	X	X	X



Q	Q_{next}	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0



Current State		Input	Next State		Output	Flip Flop Inputs			
A	**B**	**I**	**A**next	**B**next	**Y**	**J**A	**K**A	**J**B	**K**B
0	0	0	0	0	0	0	X	0	X
0	0	1	0	1	0	0	X	1	X
0	1	0	0	0	1	0	X	X	1
0	1	1	1	0	1	1	X	X	1
1	0	0	0	0	0	X	1	0	X
1	0	1	1	0	0	X	0	0	X
1	1	0	X	X	X	X	X	X	X
1	1	1	X	X	X	X	X	X	X



DA

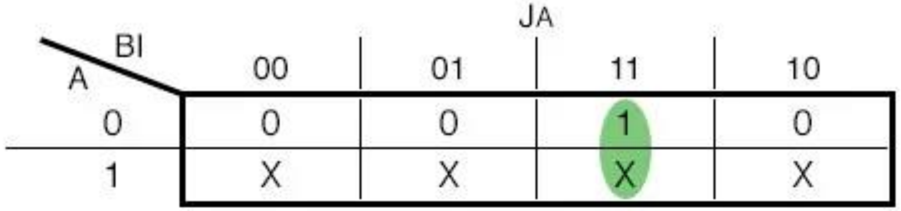
A \ BI	00	01	11	10
0	0	0	1	0
1	0	1	X	X

DB

A \ BI	00	01	11	10
0	0	1	0	0
1	0	0	X	X

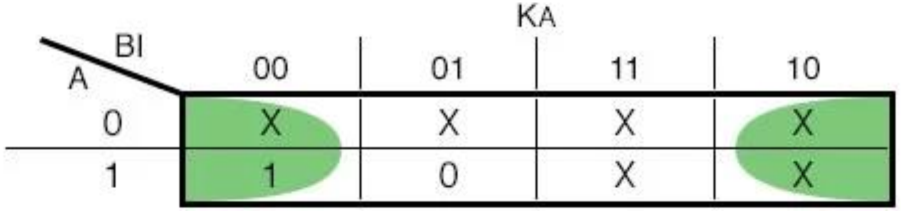
$$D_A = A \cdot I + B \cdot I = (A + B) \cdot I$$

$$D_B = \overline{A} \cdot \overline{B} \cdot I$$



J_A

A \ BI	00	01	11	10
0	0	0	1	0
1	X	X	X	X



K_A

A \ BI	00	01	11	10
0	X	X	X	X
1	1	0	X	X

J_B

A \ BI	00	01	11	10
0	0	1	X	X
1	0	0	X	X

K_B

A \ BI	00	01	11	10
0	X	X	1	1
1	X	X	X	X

$$J_A = B \cdot I$$

$$K_A = \overline{I}$$

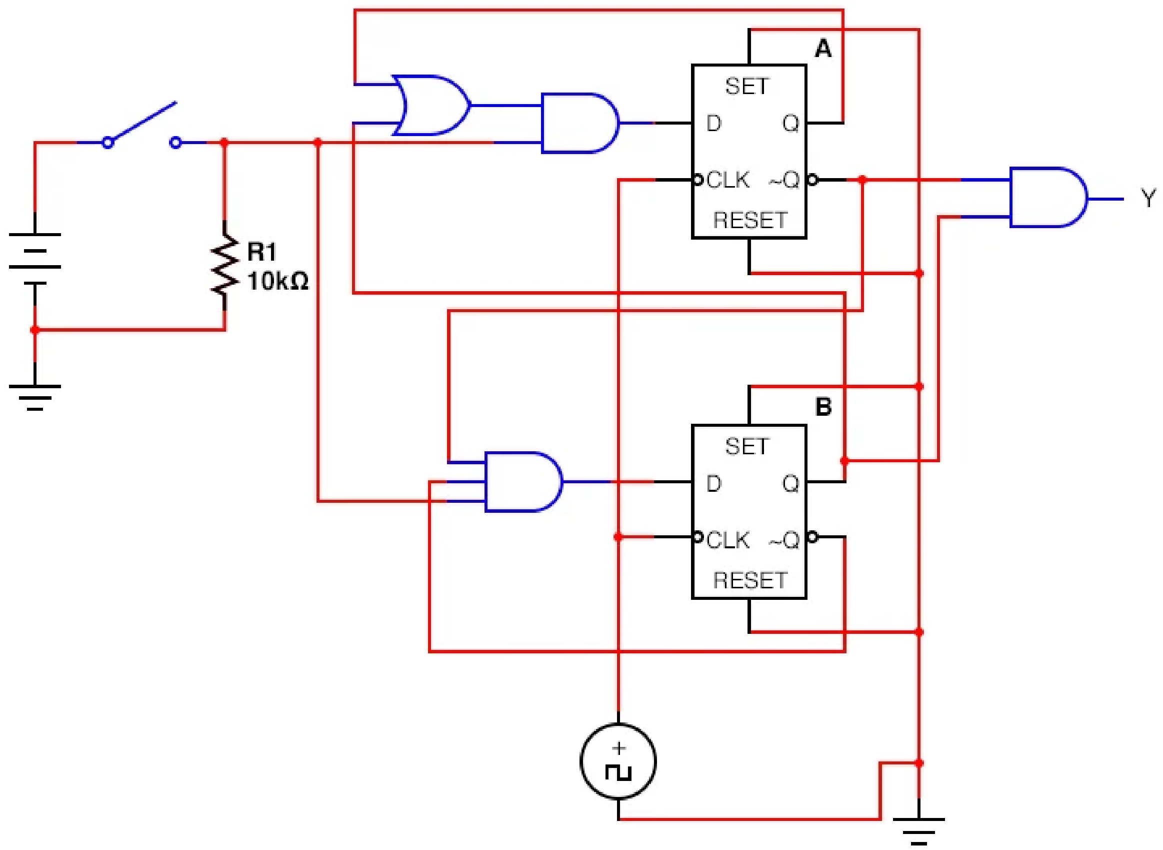
$$J_B = \overline{A} \cdot I$$

$$K_B = 1$$

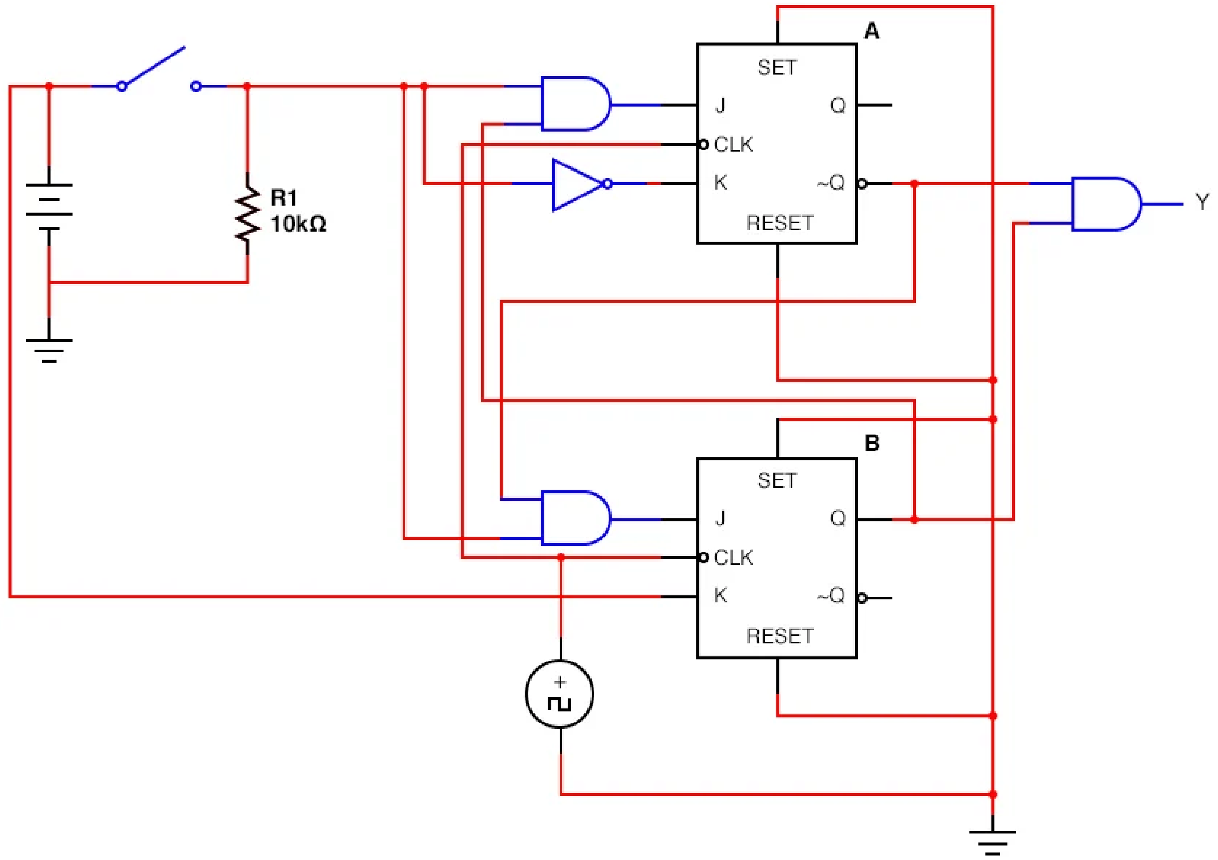
Y

A\B	0	1	
0	0	1	
1	0	1	X

$$Y = \overline{A} \cdot B$$



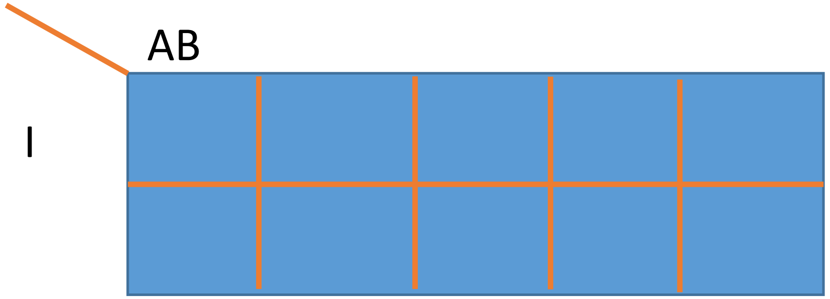
R1
10kΩ
A
SET
D
Q
CLK
~Q
RESET
B
SET
D
Q
CLK
~Q
RESET
Y
+


A
SET
J
Q
CLK
K
~Q
RESET
R1
10kΩ
Y
B
SET
J
Q
CLK
K
~Q
RESET
+

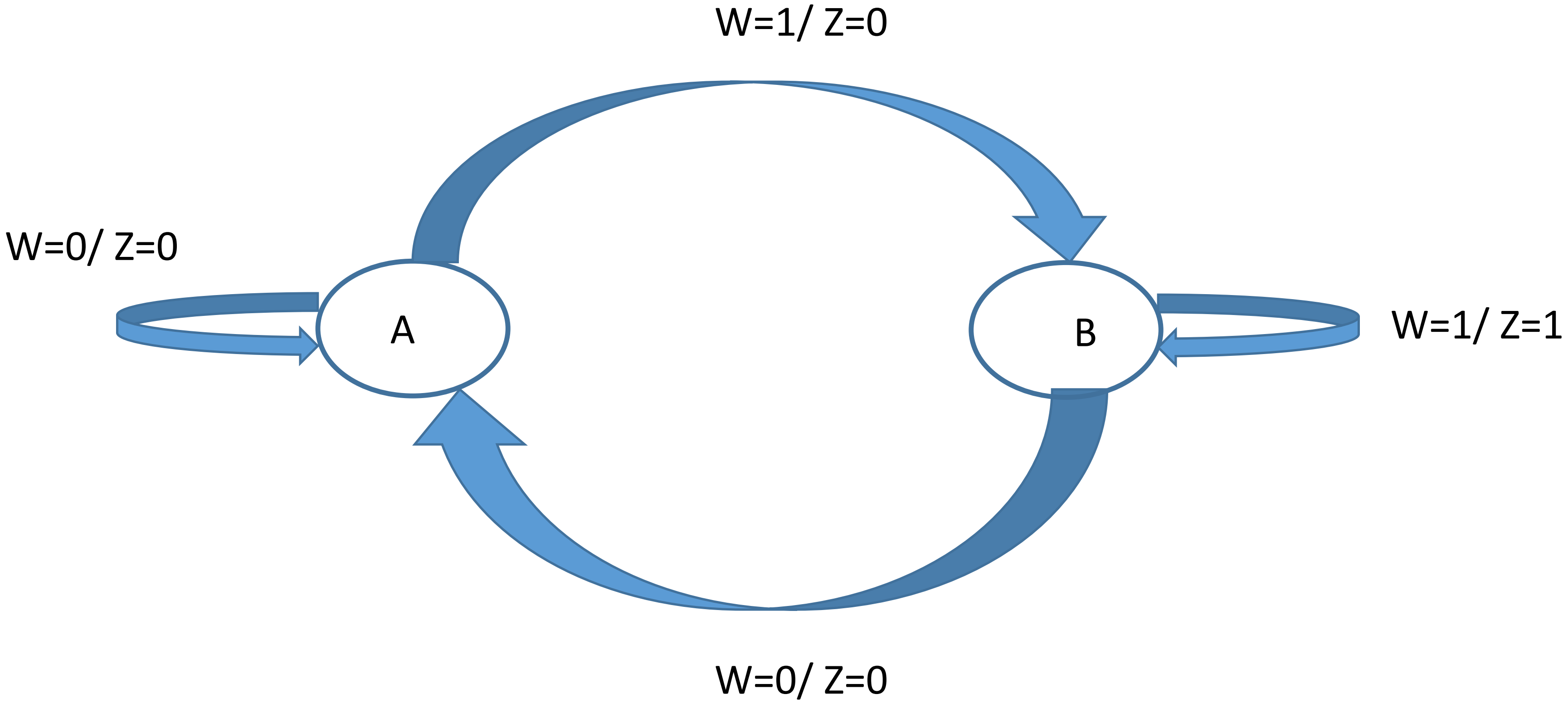
Home Work

Solve previous question in other form

Present State		Next State				Output
		I=0		I=1		
A	B	A	B	A	B	Y
0	0					

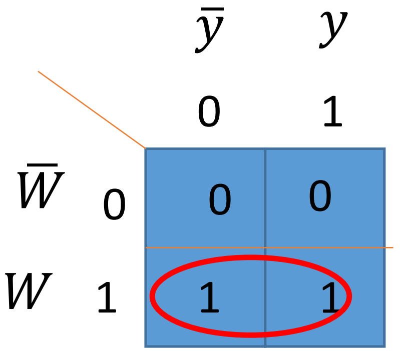

AB
I

Clock Cycle	t0	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10
W	0	1	0	1	1	0	1	1	1	0	1
Z	0	0	0	0	1	0	0	1	1	0	0


W=1/ Z=0
W=0/ Z=0
A
B
W=1/ Z=1
W=0/ Z=0

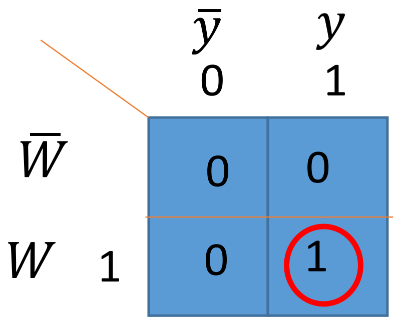
Present State	Next State		Output	
	W=0	W=1	W=0	W=1
A	A	B	0	0
B	A	B	0	1

	Present State	Next State		Output	
		W=0	W=1	W=0	W=1
	y	Y	Y	Z	Z
A	0	0	1	0	0
B	1	0	1	0	1



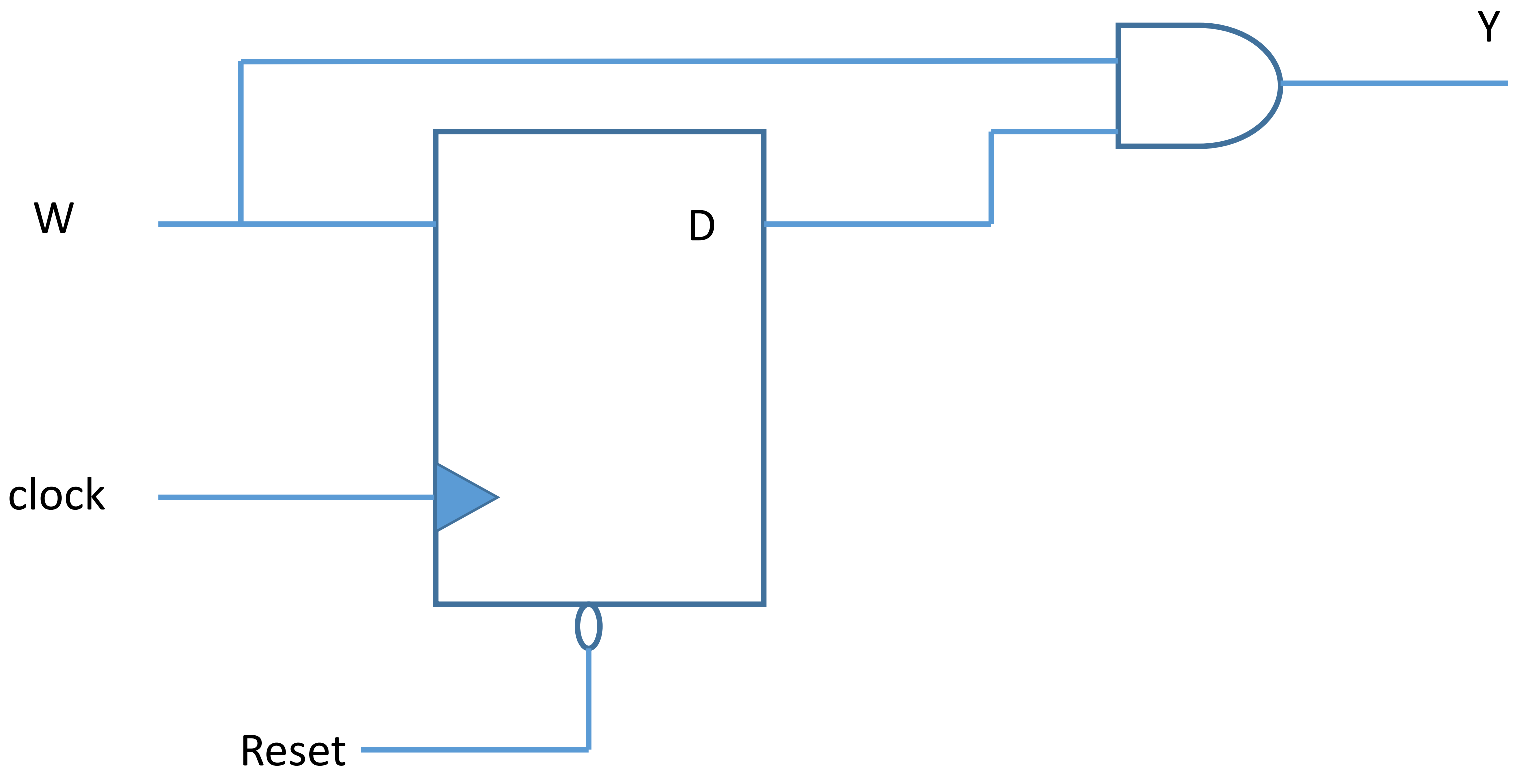
		$\bar{y}$	y
		0	1
$\bar{W}$	0	0	0
W	1	1	1

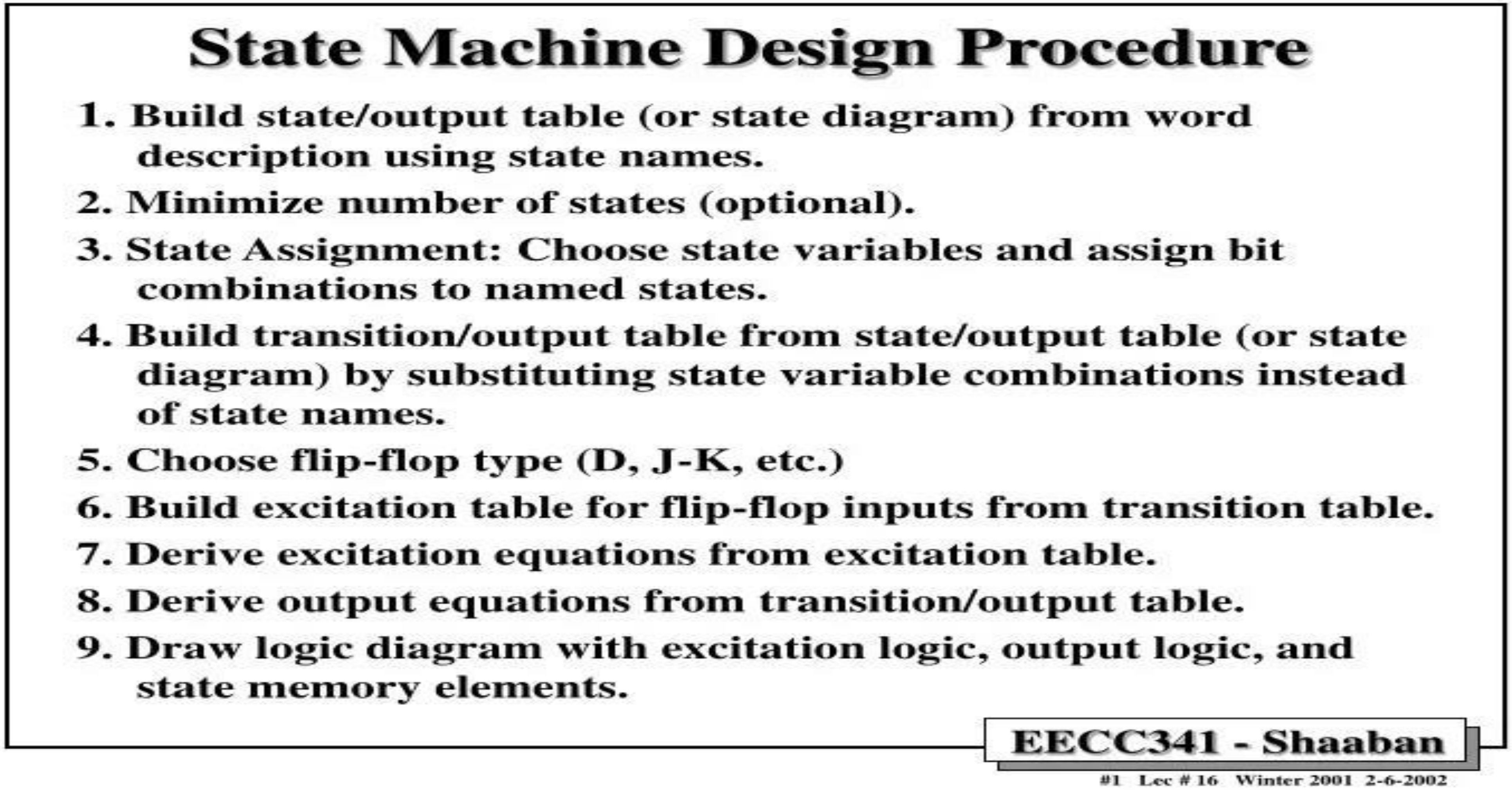
$Y = W$



		$\bar{y}$	y
		0	1
$\bar{W}$		0	0
W	1	0	1

Z$= Wy$

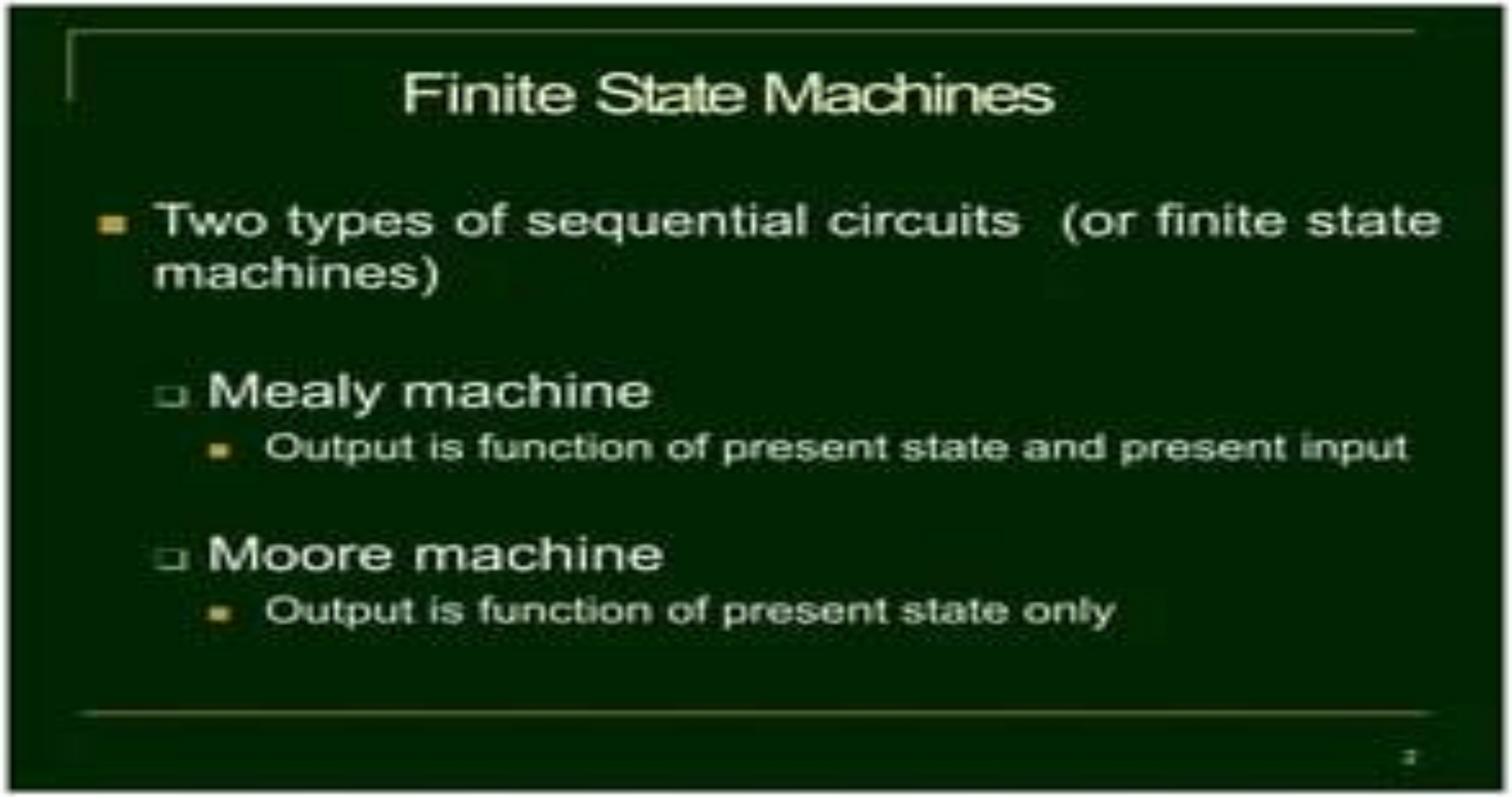

Y
W
D
clock
Reset



State Machine Design Procedure

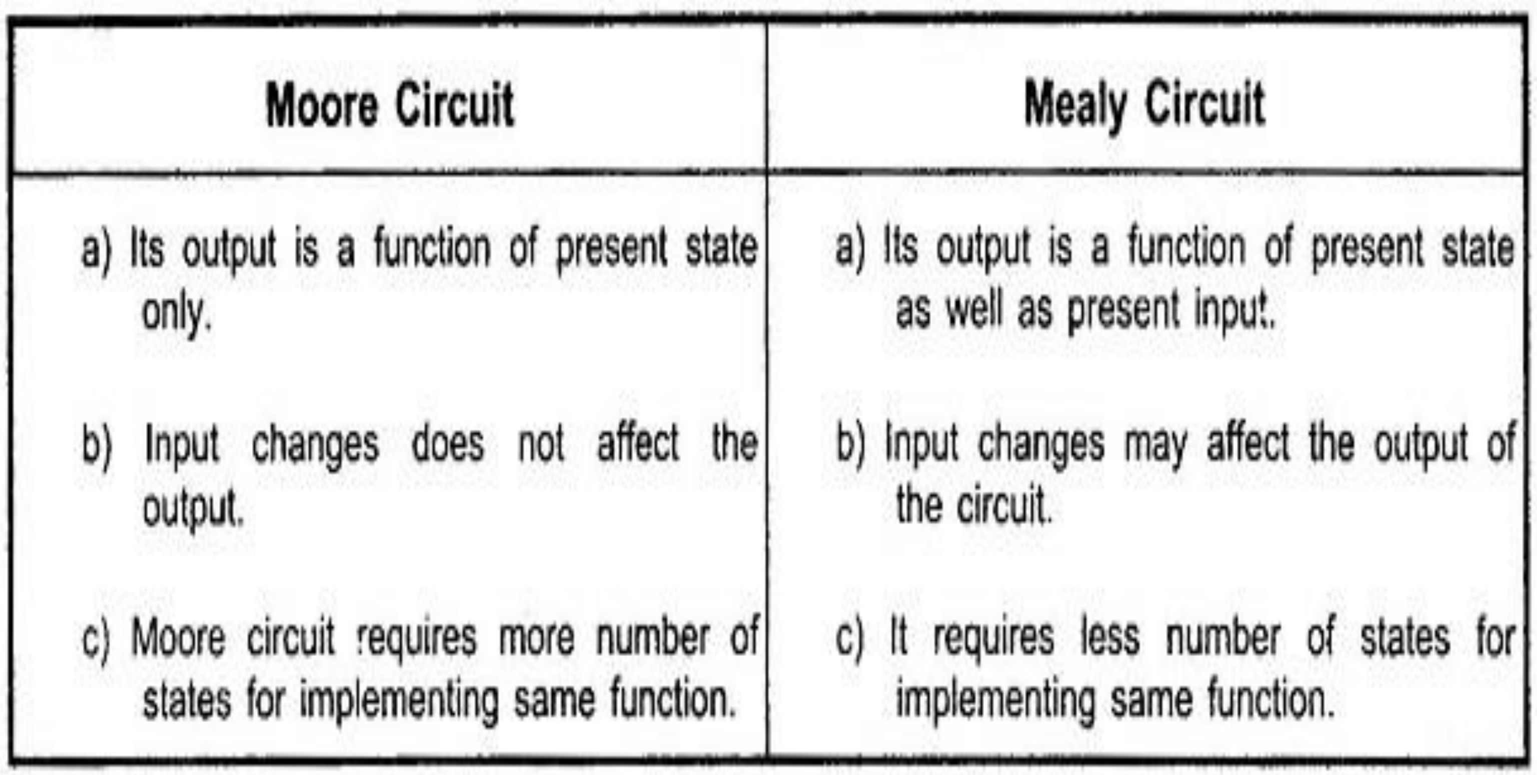
1. Build state/output table (or state diagram) from word description using state names.
2. Minimize number of states (optional).
3. State Assignment: Choose state variables and assign bit combinations to named states.
4. Build transition/output table from state/output table (or state diagram) by substituting state variable combinations instead of state names.
5. Choose flip-flop type (D, J-K, etc.)
6. Build excitation table for flip-flop inputs from transition table.
7. Derive excitation equations from excitation table.
8. Derive output equations from transition/output table.
9. Draw logic diagram with excitation logic, output logic, and state memory elements.

Lecture Seven



Finite State Machines

- Two types of sequential circuits (or finite state machines)
  - Mealy machine
    - Output is function of present state and present input
  - Moore machine
    - Output is function of present state only

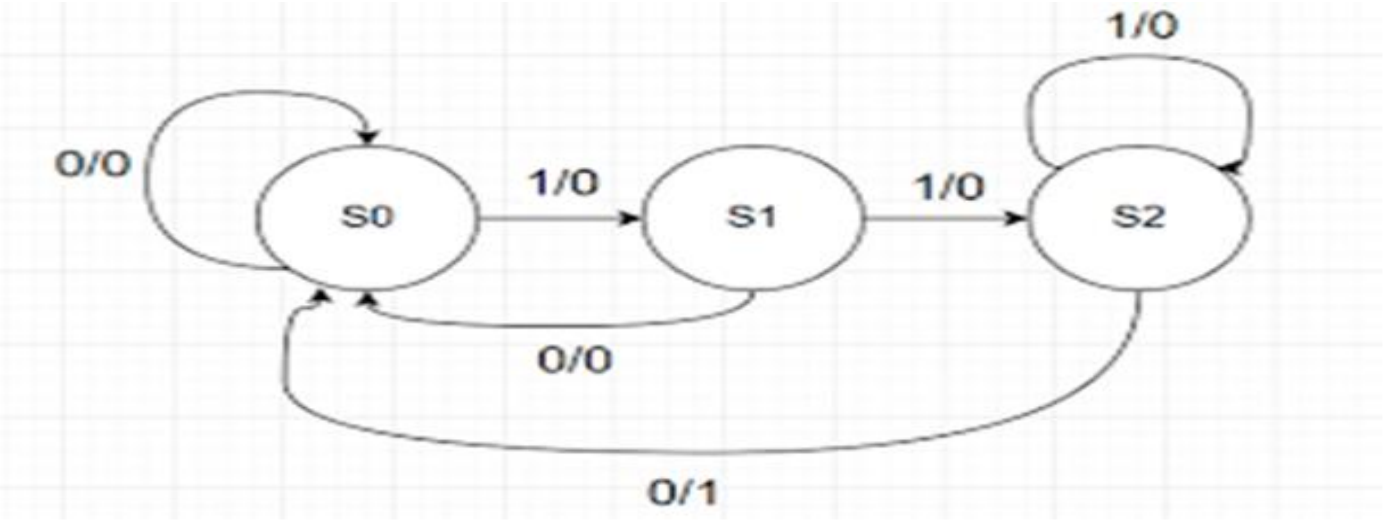


Moore Circuit	Mealy Circuit
a) Its output is a function of present state only.	a) Its output is a function of present state as well as present input.
b) Input changes does not affect the output.	b) Input changes may affect the output of the circuit.
c) Moore circuit requires more number of states for implementing same function.	c) It requires less number of states for implementing same function.

Table 3.13

Example:-

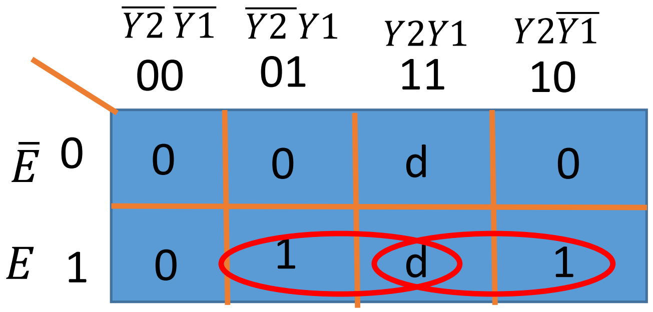
State machine design 110 Mealy?



110 Detector Mealy

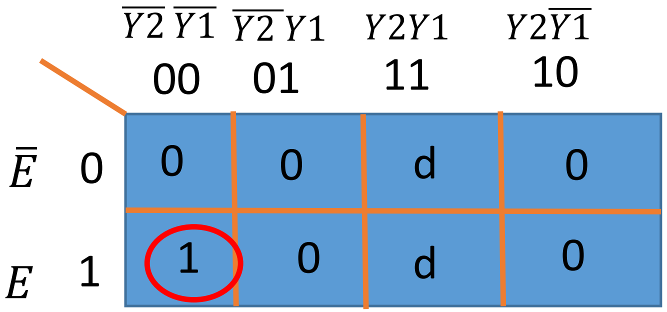
Present State	Next State		Output	
	E=0	E=1	E=0	E=1
S0	S0	S1	0	0
S1	S0	S2	0	0
S2	S0	S2	1	0

Present State		Next State				Output	
		E=0		E=1		E=0	E=1
Y2	Y1	Y2	Y1	Y2	Y1	Z	Z
0	0	0	0	0	1	0	0
0	1	0	0	1	0	0	0
1	0	0	0	1	0	1	0
1	1	d	d	d	d	d	d



	$\overline{Y2}\,\overline{Y1}$ 00	$\overline{Y2}\,Y1$ 01	$Y2Y1$ 11	$Y2\overline{Y1}$ 10
$\bar{E}$ 0	0	0	d	0
E 1	0	1	d	1

$$Y1 = EY1 + EY2$$



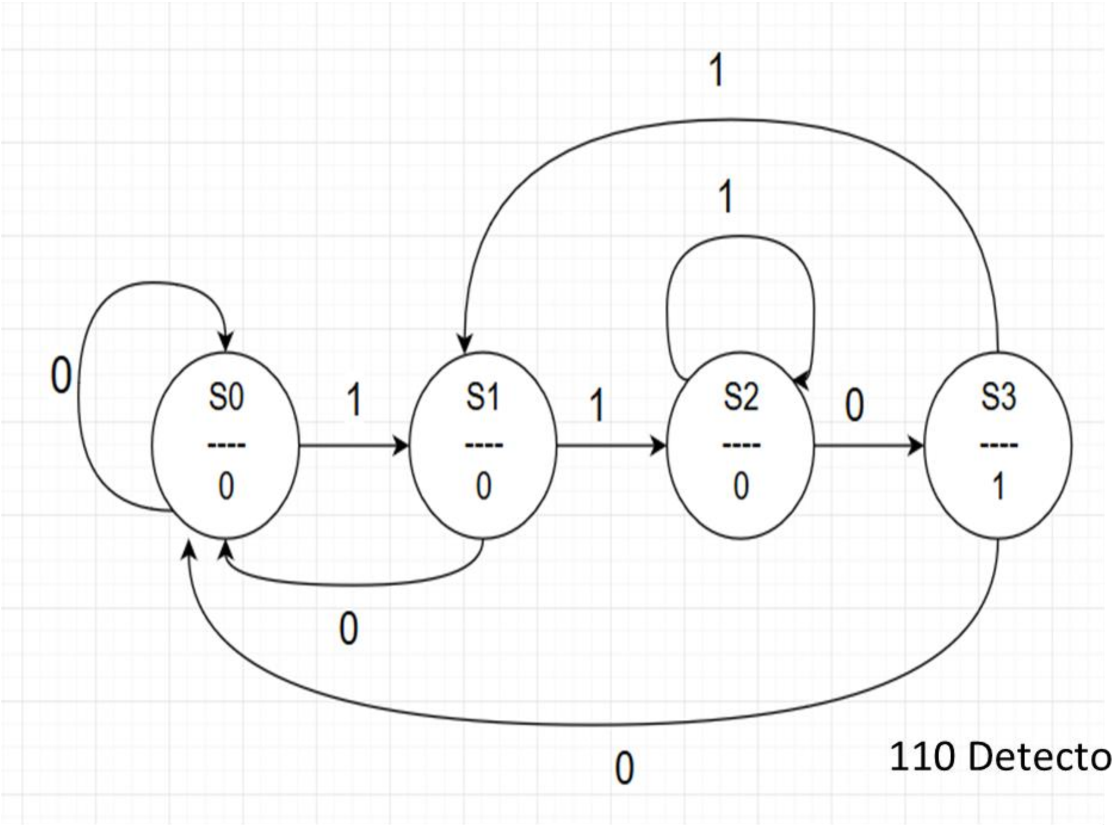
	$\overline{Y2}\,\overline{Y1}$ 00	$\overline{Y2}\,Y1$ 01	$Y2Y1$ 11	$Y2\overline{Y1}$ 10
$\bar{E}$ 0	0	0	d	0
E 1	1	0	d	0

$$Y2 = E\overline{Y2}\,\overline{Y1}$$

		$\overline{Y2}\,\overline{Y1}$	$\overline{Y2}\,Y1$	$Y2Y1$	$Y2\overline{Y1}$
		00	01	11	10
$\bar{E}$	0	0	0	d	1
E	1	0	0	d	0

$Z = \bar{E}$ Y2

Home Work
Draw circuit Diagram



110 Detector

Moore

Present State	Next State		Output
	E=0	E=1	Z
S0	S0	S1	0
S1	S0	S2	0
S2	S3	S2	0
S3	S0	S1	1

Present	State	Next State				Output
Y2	Y1	E=0		E=1		
		Y2	Y1	Y2	Y1	Z
0	0	0	0	0	1	0
0	1	0	0	1	0	0
1	0	1	1	1	0	0
1	1	0	0	0	1	1

`

	$\overline{Y2}\,\overline{Y1}$ 00	$\overline{Y2}\,Y1$ 01	$Y2Y1$ 11	$Y2\overline{Y1}$ 10
$\bar{E}$ 0	0	0	0	1
E 1	1	0	1	0

$$Y1 = EY2Y1 + E\overline{Y2}\ \overline{Y1} + \ \bar{E}\ Y2\overline{Y1}$$

	$\overline{Y2}\,\overline{Y1}$ 00	$\overline{Y2}\,Y1$ 01	$Y2Y1$ 11	$Y2\overline{Y1}$ 10
$\bar{E}$ 0	0	0	0	1
E 1	0	1	0	1

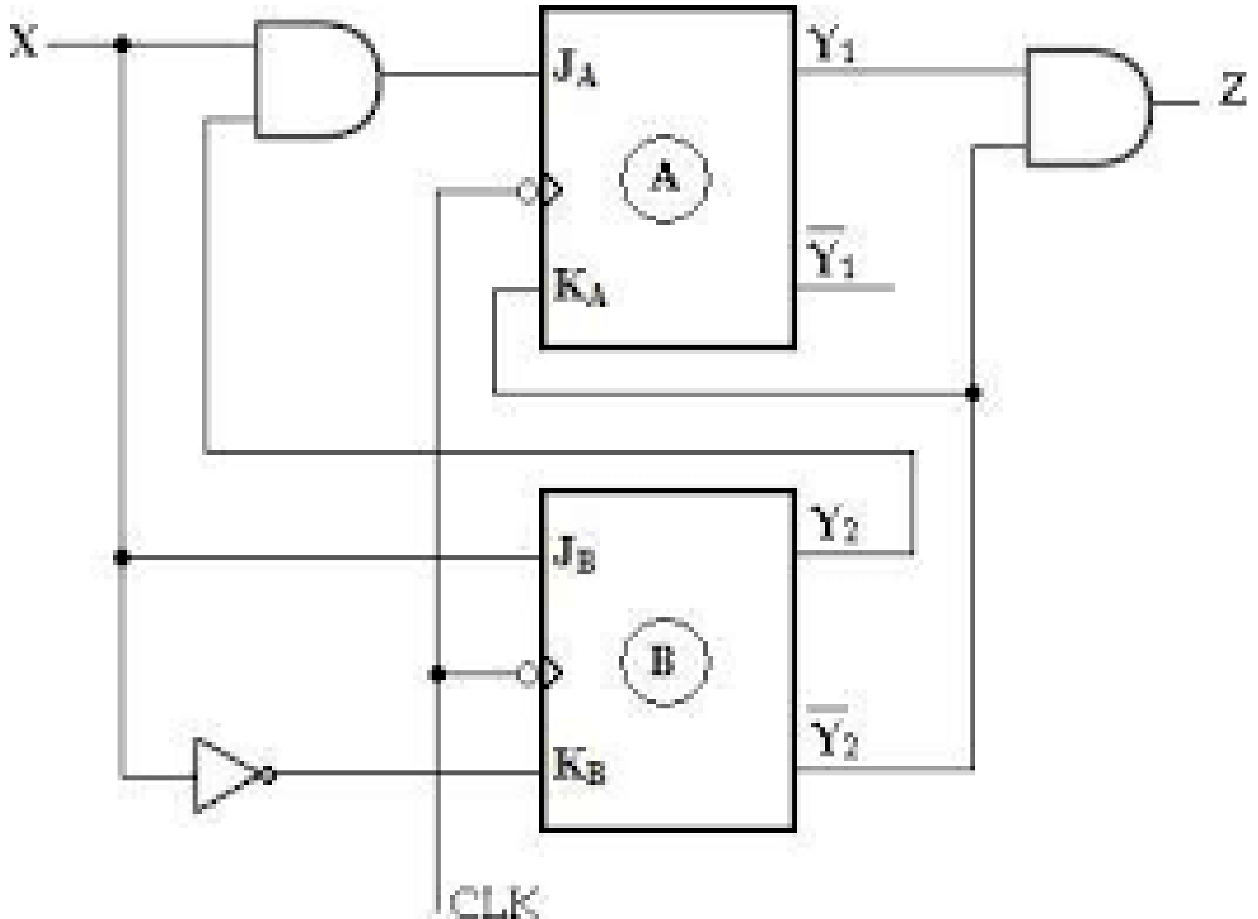
$$Y2 = E\overline{Y2}Y1 + \ Y2\overline{Y1}$$

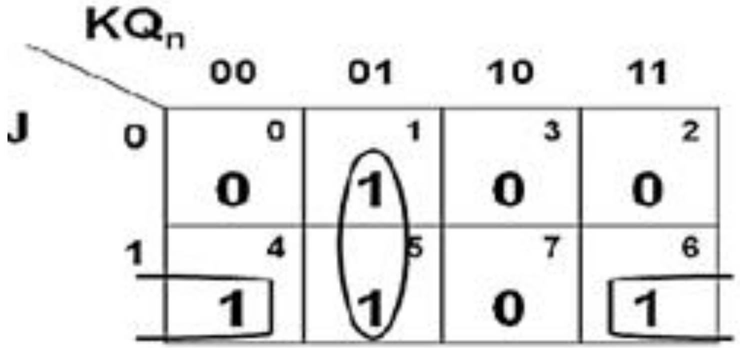
	$\overline{Y1}$ 0	$Y1$ 1
$\overline{Y2}$ 0	0	0
$Y2$ 1	0	1

$$Z = Y2Y1$$

Home Work
Draw circuit Diagram

Analyze the circuit below to obtain its state diagram?





J \ KQ_n	00	01	10	11
0	0	1	0	0
1	1	1	0	1

$$Q_{n+1} = JQ_n' + K'Q_n$$

J	K	Q_n	Q_{n+1}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Fig3. Characteristic equation of JK flip flop

Solution:-

$Y = Q = q.\bar{k} + \bar{q}.\mathrm{J}$

$J_A = y_2.X$

$K_A = \overline{y_2}$

$Y_1 = \overline{y_1}\,.J_A + y_1\overline{k_A}$

$Y_1 = \overline{y_1}\,.y_2.X + y_1y_2$

$J_B = X$

$K_B = \bar{X}$

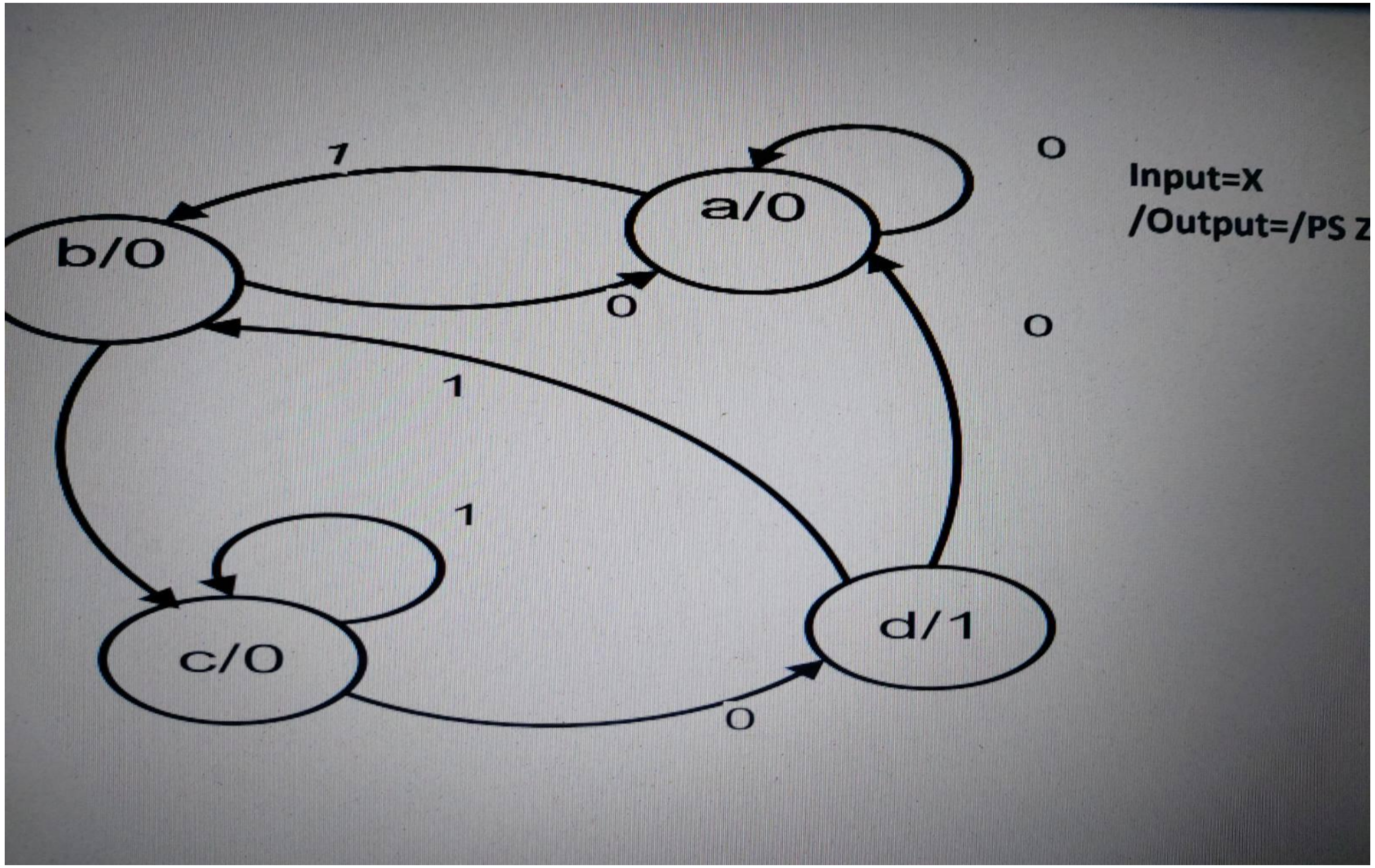
$Y_2 = \overline{y_2}\,.J_B + y_2\overline{k_B}$

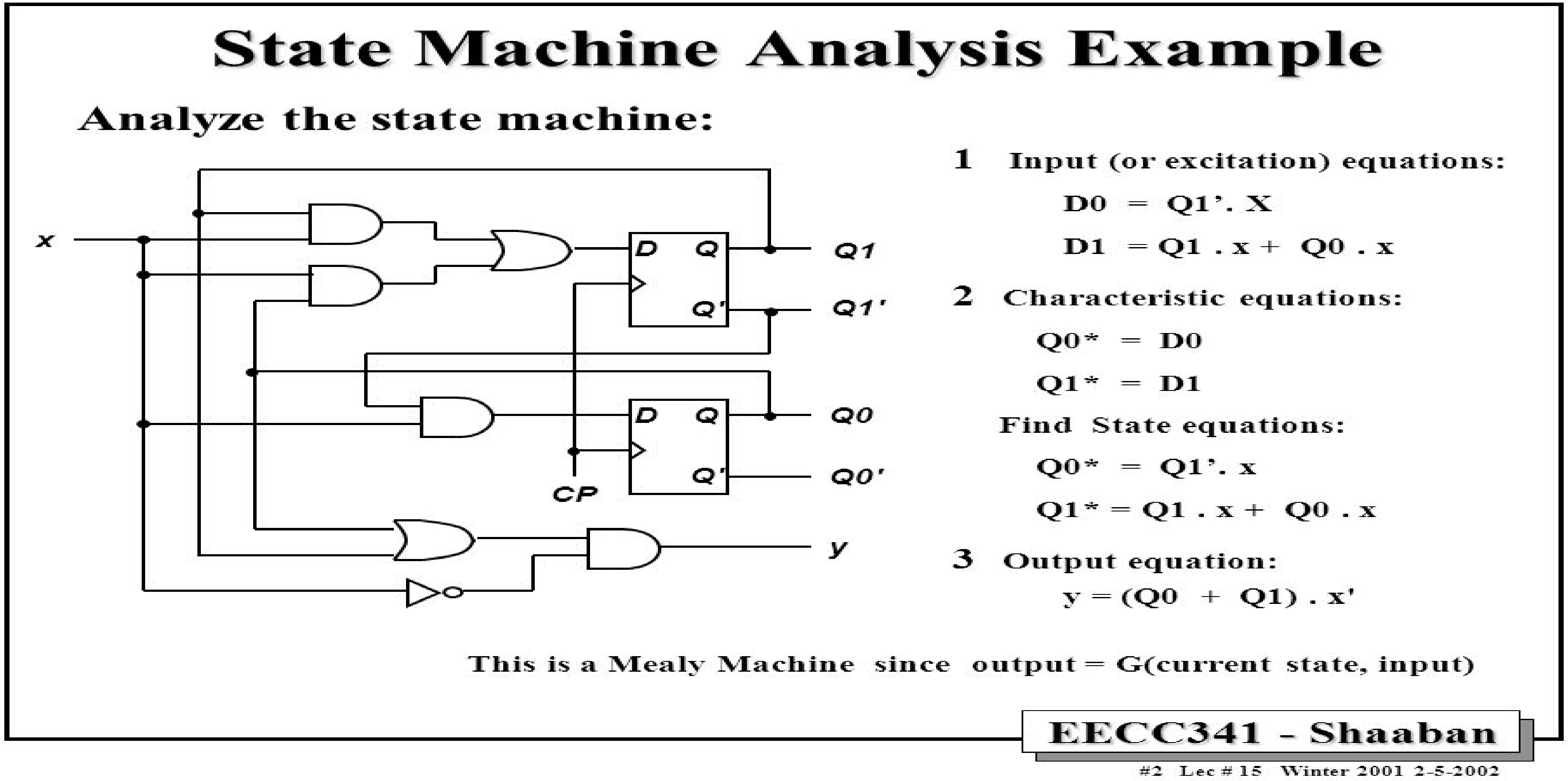
$Y_2 = \overline{y_2}\,.X + y_2X$

$Z = y_1.\overline{y_2}$

Present State	Next State		
	X=0	X=1	
y_1y_2	Y_1Y_2	Y_1Y_2	Z
00	00	01	0
01	00	11	0
11	10	11	0
10	00	01	1

Present State	Next State		
	X=0	X=1	
y_1y_2	Y_1Y_2	Y_1Y_2	Z
00	a	b	0
01	a	c	0
11	d	c	0
10	a	b	1


1
0
a/0
Input=X
/Output=/PS Z
b/0
0
0
1
1
c/0
d/1
0



State Machine Analysis Example

Analyze the state machine:

1 Input (or excitation) equations:

$D0 = Q1'. X$

$D1 = Q1 . x + Q0 . x$

2 Characteristic equations:

$Q0^* = D0$

$Q1^* = D1$

Find State equations:

$Q0^* = Q1'. x$

$Q1^* = Q1 . x + Q0 . x$

3 Output equation:

$y = (Q0 + Q1) . x'$

This is a Mealy Machine since output = G(current state, input)

State Machine Analysis

From the state equation and output equation, construct the state transition – output table.

State Equation

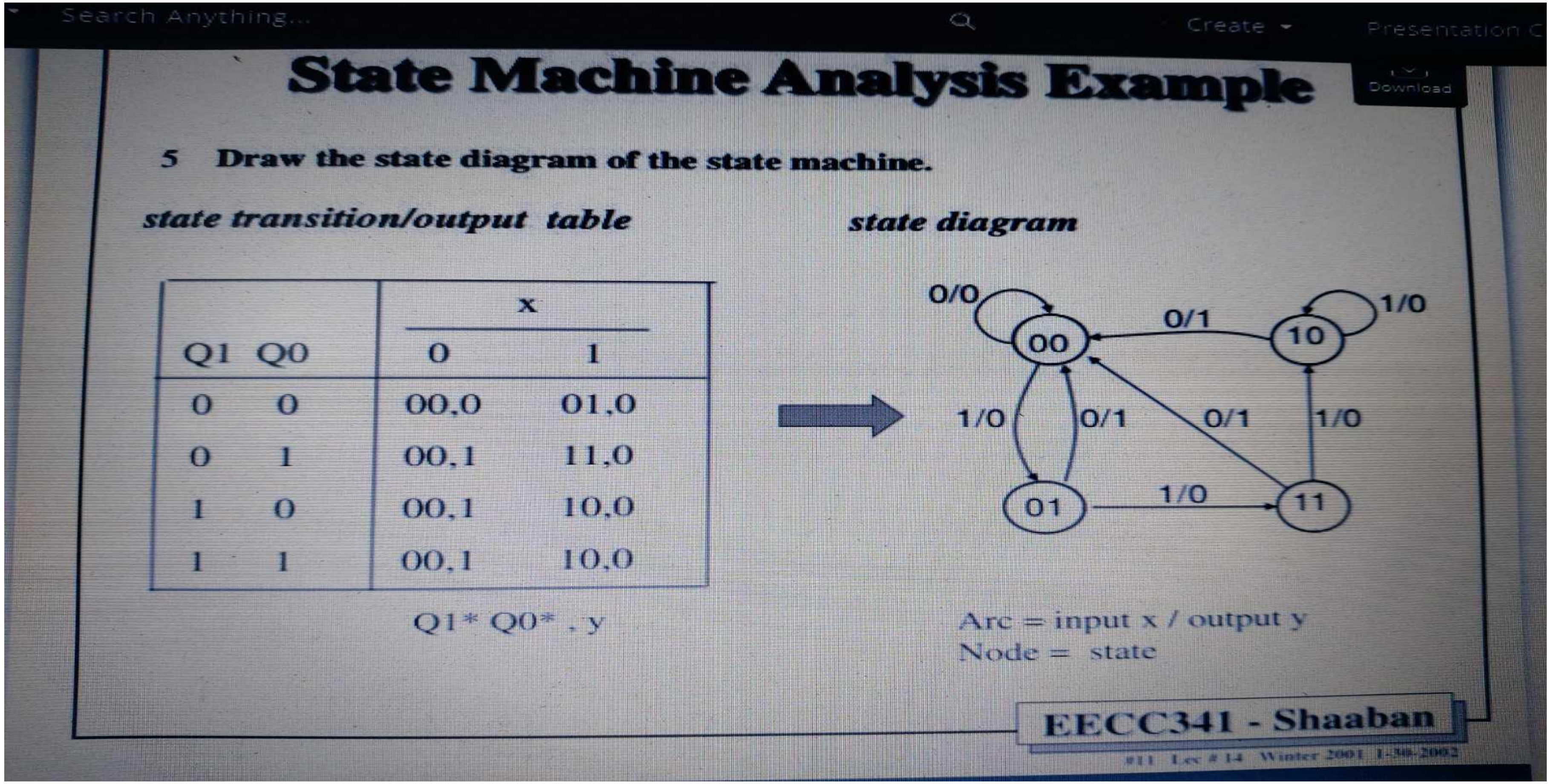
$$Q_0^* = \overline{Q_1 \quad} \quad .\mathrm{x}$$

$$Q_1^* = Q_1 \ .\mathrm{x}+Q_0 \ .\mathrm{x}$$

Output Equation

$$y = (Q_0 \ + Q_1 \ ).\bar{x}$$

Q_1	Q_0	**x=0**			**x=1**		
		Q_1^*	Q_0^*	y	Q_1^*	Q_0^*	y
0	0	0	0	0	0	1	0
0	1	0	0	1	1	1	0
1	0	0	0	1	1	0	0
1	1	0	0	1	1	0	0



State Machine Analysis Example

5 **Draw the state diagram of the state machine.**

state transition/output table

	x	
Q1 Q0	0	1
0 0	00,0	01,0
0 1	00,1	11,0
1 0	00,1	10,0
1 1	00,1	10,0

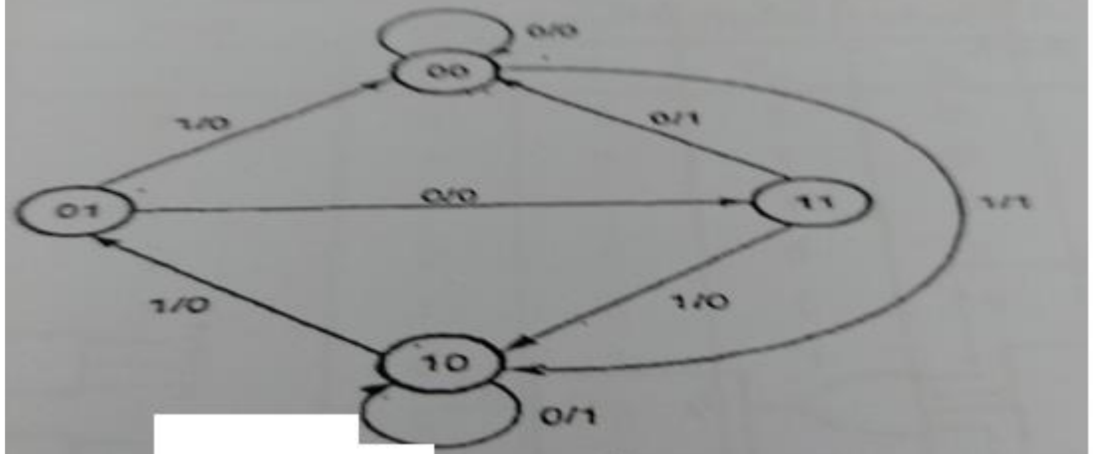
Q1* Q0* , y

state diagram

Arc = input x / output y
Node = state

EECC341 - Shaaban

Lecture eight



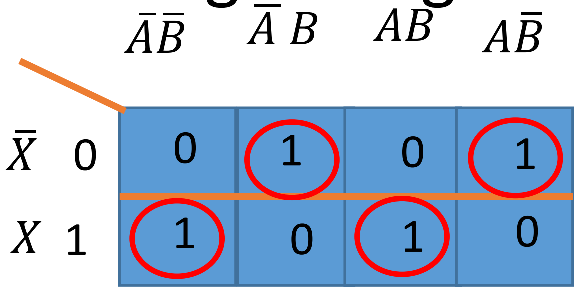
State diagram

D Flip-Flop

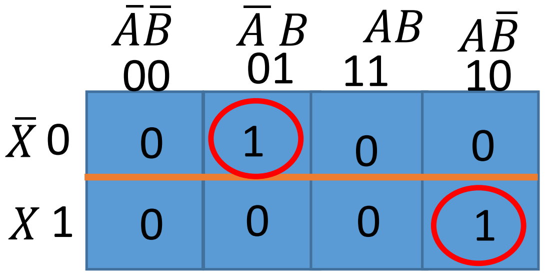
Present State		Next State		Output	
		X=0	X=1	X=0	X=1
A	B	AB	AB	Y	Y
0	0	00	10	0	1
0	1	11	00	0	0
1	0	10	01	1	0
1	1	00	10	1	0

D Flip-Flop

Design using D flip-flop



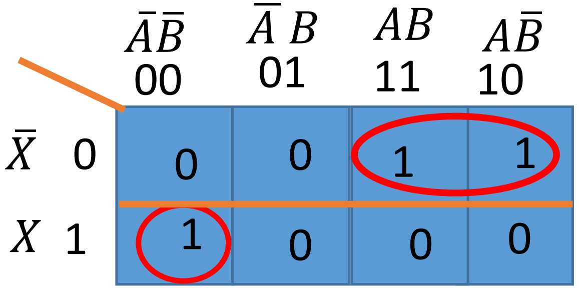
	$\bar{A}\bar{B}$	$\bar{A}\,B$	AB	$A\bar{B}$
$\bar{X}$ 0	0	1	0	1
X 1	1	0	1	0



	$\bar{A}\bar{B}$ 00	$\bar{A}\,B$ 01	AB 11	$A\bar{B}$ 10
$\bar{X}$ 0	0	1	0	0
X 1	0	0	0	1

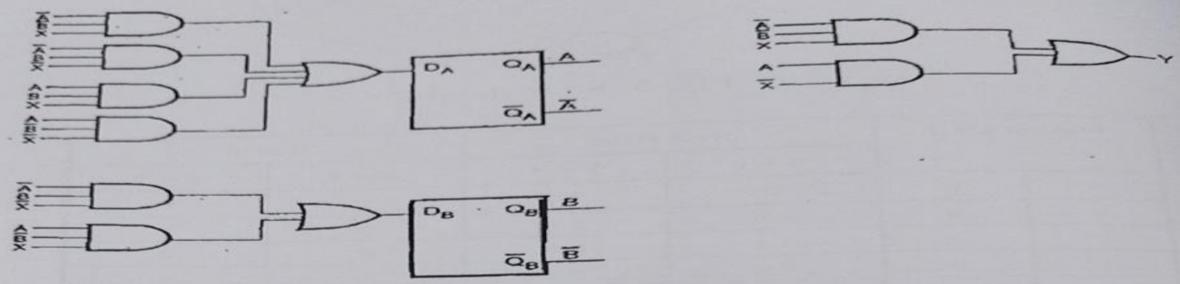
$DA = \bar{X}\;\bar{A}\,\mathrm{B} + \bar{X}A\;\bar{B} + X\;\bar{A}\bar{B} + XAB$

$DB = \bar{X}\;\bar{A}\,\mathrm{B} + XA\bar{B}$



	$\bar{A}\bar{B}$ 00	$\bar{A}\,B$ 01	AB 11	$A\bar{B}$ 10
$\bar{X}$ 0	0	0	1	1
X 1	1	0	0	0

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



"Logic diagram of given sequential circuit using D flip-flop"

ii) Design using T flip- flops :

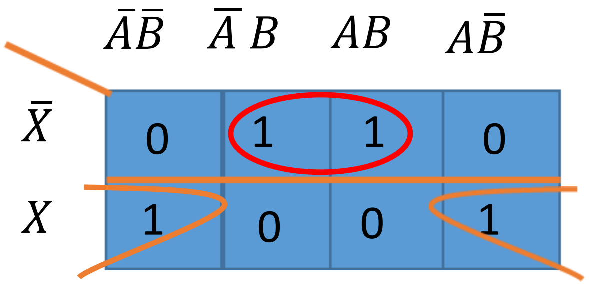
Q_n	Q_{n+1}	T
0	0	0
0	1	1
1	0	1
1	1	0

"Excitation table for T flip-flop"

T Flip-Flop

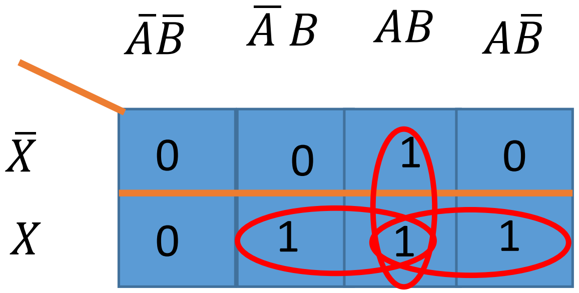
Present state		Input	Next State		Flip-flop Inputs		Output
A	B	X	A	B	TA	TB	Y
0	0	0	0	0	0	0	0
0	0	1	1	0	1	0	1
0	1	0	1	1	1	0	0
0	1	1	0	0	0	1	0
1	0	0	1	0	0	0	1
1	0	1	0	1	1	1	0
1	1	0	0	0	1	1	1
1	1	1	1	0	0	1	0

T Flip-Flop



	$\bar{A}\bar{B}$	$\bar{A}\,B$	AB	$A\bar{B}$
$\bar{X}$	0	1	1	0
X	1	0	0	1

$$TA = \bar{X}\,B + X\bar{B}$$

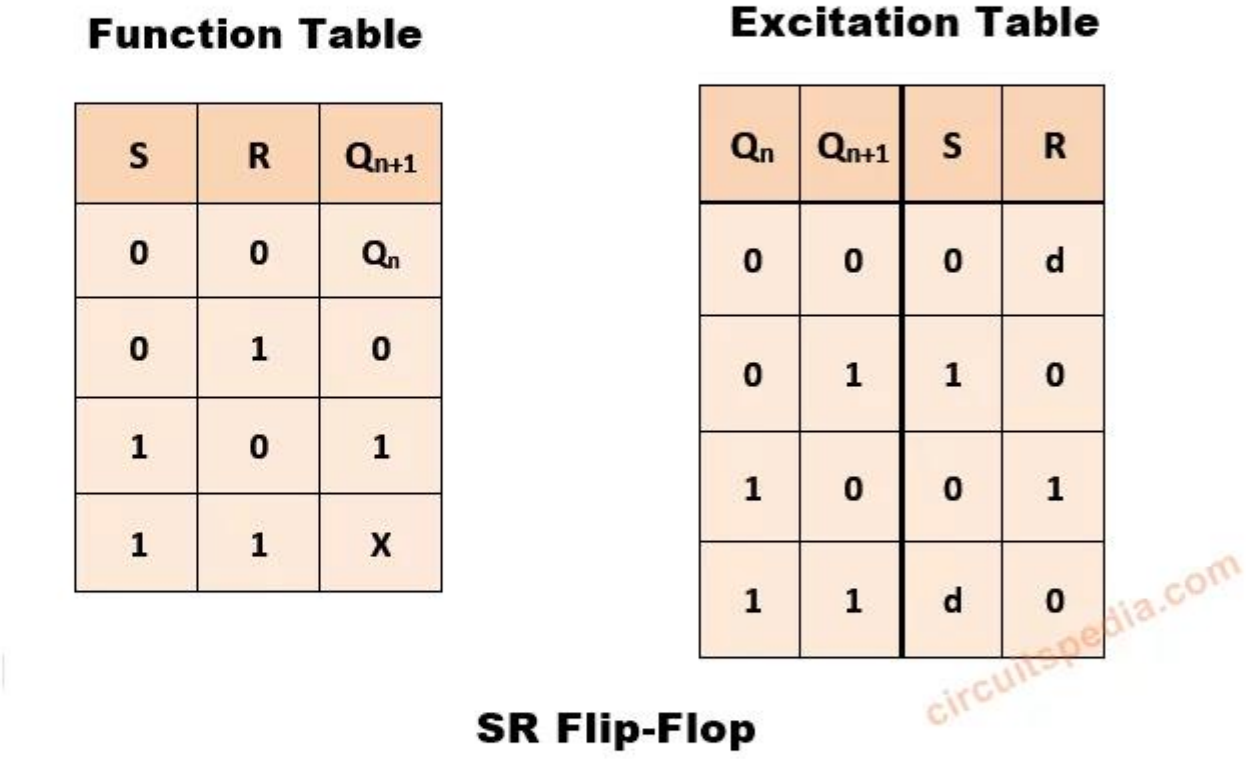


	$\bar{A}\bar{B}$	$\bar{A}\,B$	AB	$A\bar{B}$
$\bar{X}$	0	0	1	0
X	0	1	1	1

$$TB = XA + XB + AB$$

Y as previous case
in D flip-flop

RS Flip-Flop



Function Table

S	R	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	X

Excitation Table

Q_n	Q_{n+1}	S	R
0	0	0	d
0	1	1	0
1	0	0	1
1	1	d	0

SR Flip-Flop

RS Flip-Flop

Present State		Input	Next State		Flip flop Inputs				Output1
A	B	X	A	B	R_A	S_A	R_B	S_B	Y
0	0	0	0	0	x	0	x	0	0
0	0	1	1	0	0	1	x	0	1
0	1	0	1	1	0	1	0	x	0
0	1	1	0	0	x	0	1	0	0
1	0	0	1	0	0	x	x	0	1
1	0	1	0	1	1	0	0	1	0
1	1	0	0	0	1	0	1	0	1
1	1	1	1	0	0	x	1	0	0

Circuit Excitation Table

RS Flip-Flop

	$\bar{A}\bar{B}$	$\bar{A}B$	AB	$A\bar{B}$
$\bar{X}$	x	0	1	0
X	0	x	0	1

$$R_A = \bar{X}AB + XA\bar{B}$$

	$\bar{A}\bar{B}$	$\bar{A}B$	AB	$A\bar{B}$
$\bar{X}$	0	1	0	x
X	1	0	x	0

$$S_A = \bar{X}\,\bar{A}B + X\bar{A}\bar{B}$$

	$\bar{A}\bar{B}$	$\bar{A}B$	AB	$A\bar{B}$
$\bar{X}$	0	0	1	x
X	x	1	1	0

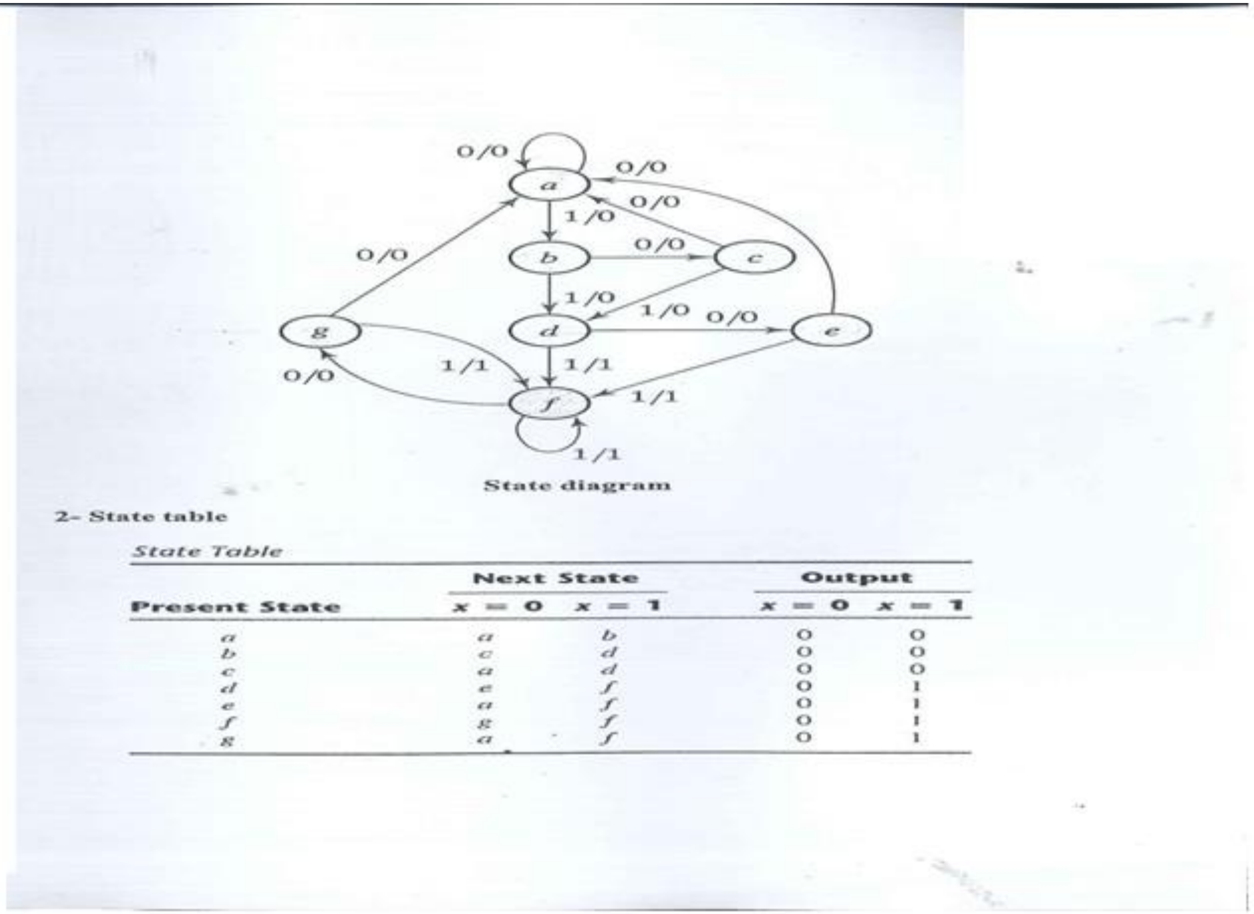
$$R_B = AB + XB$$

	$\bar{A}\bar{B}$	$\bar{A}B$	AB	$A\bar{B}$
$\bar{X}$	0	x	0	0
X	0	0	0	1

$$S_B = XA\bar{B}$$

Home Work

Design using JK flip-flop?

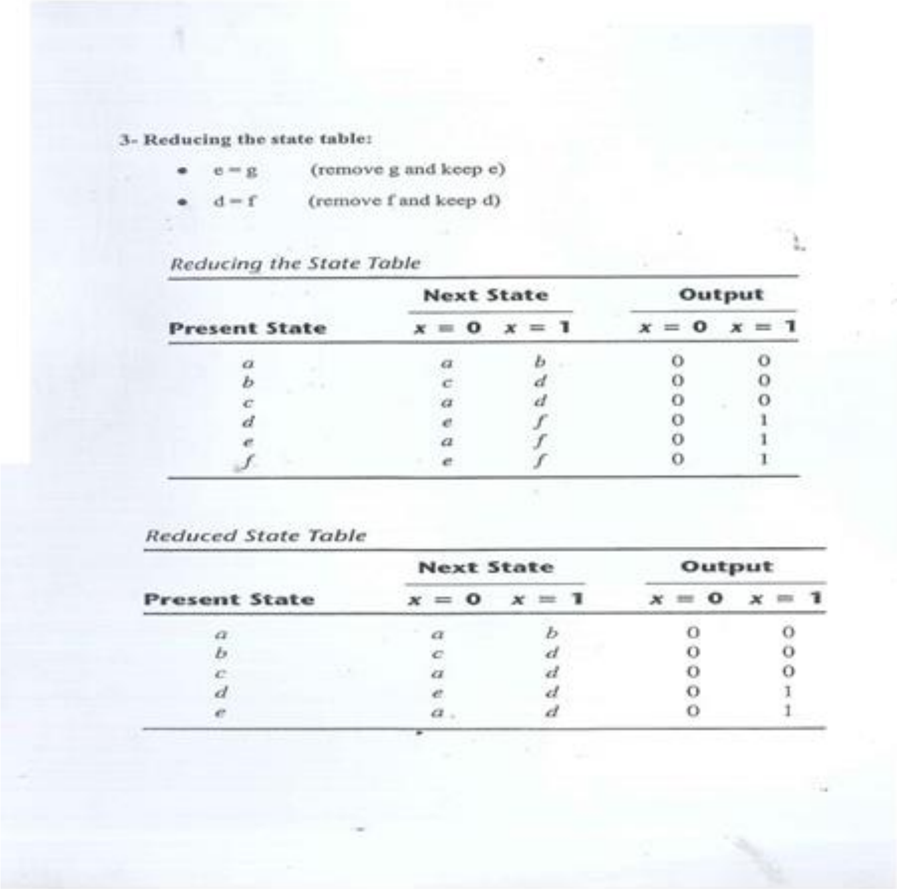


State diagram

2- State table

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



3- Reducing the state table:

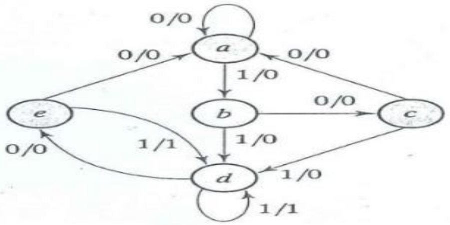
- e = g (remove g and keep e)
- d = f (remove f and keep d)

Reducing the State Table

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

Reduced State Table

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



"State diagram after reduction'

Lecture Nine

State Reduction

- Inspection
- Implication Table

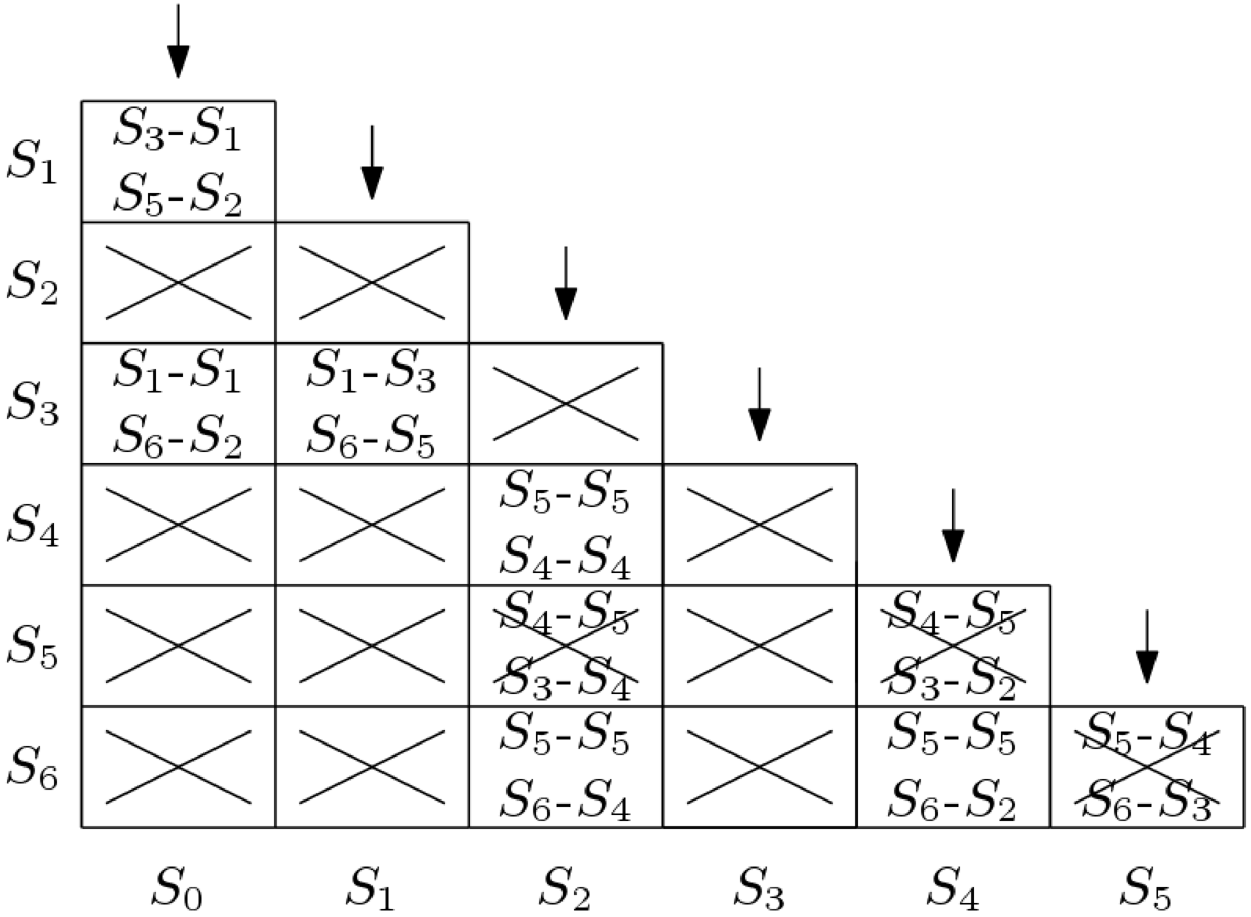
Example (1)

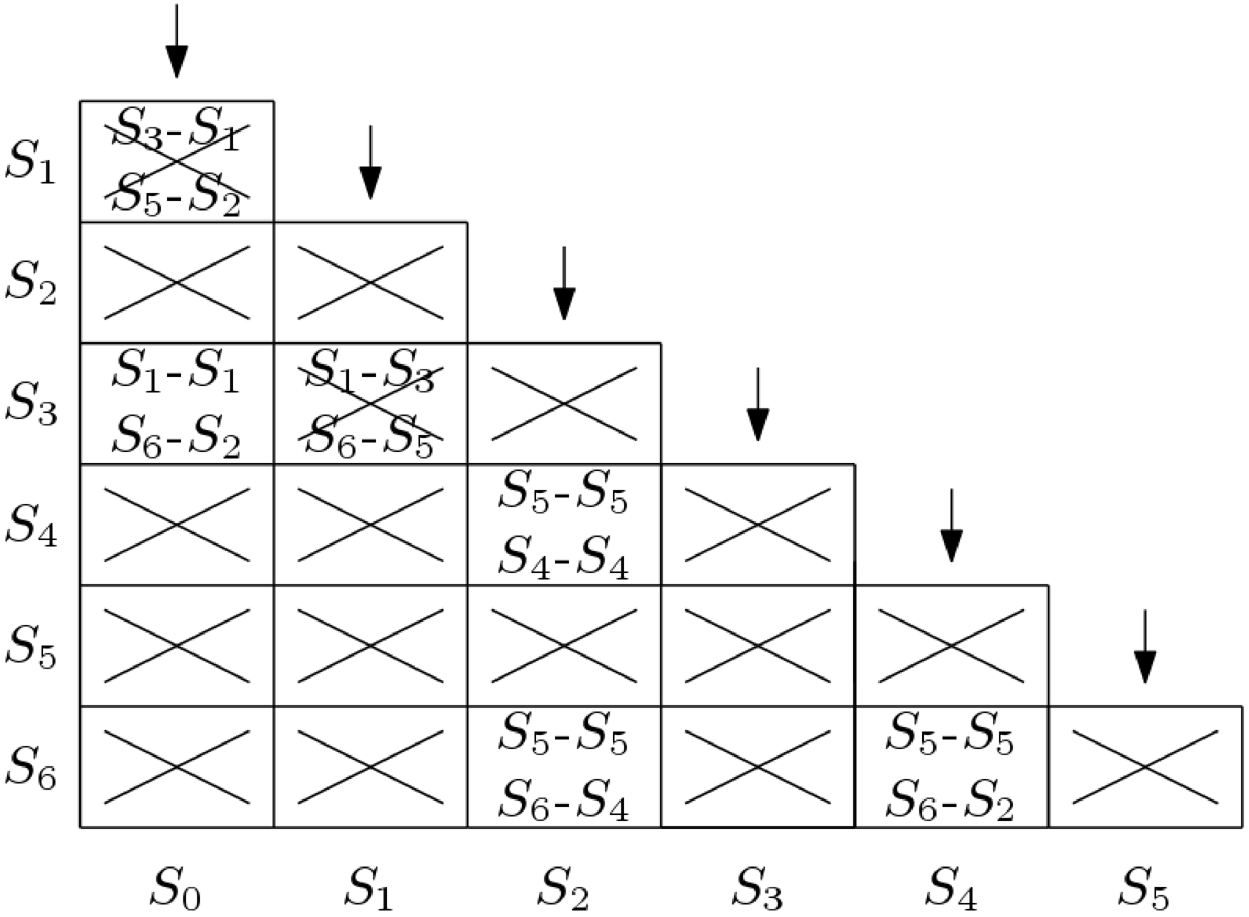
Reduce the number of states for table shown below using Implication table method?

Present State	Next State		Output
	$x = 0$	$x = 1$	
S_0	S_1	S_2	1
S_1	S_3	S_5	1
S_2	S_5	S_4	0
S_3	S_1	S_6	1
S_4	S_5	S_2	0
S_5	S_4	S_3	0
S_6	S_5	S_6	0

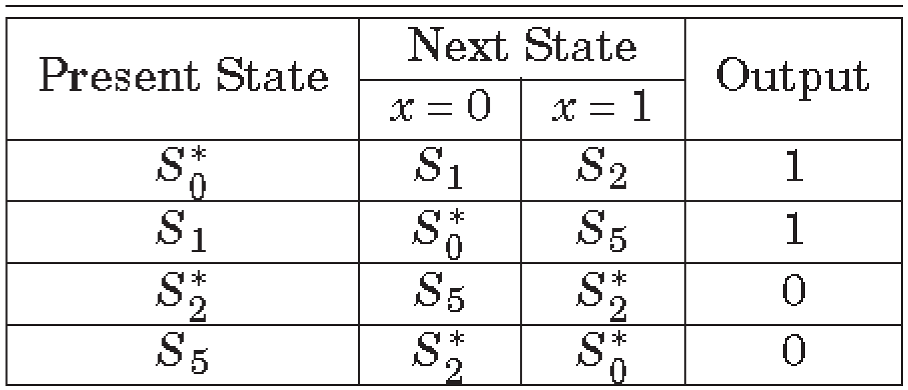
Present State	Next State		Output
	$x = 0$	$x = 1$	
S_0	S_1	S_2	1
S_1	S_3	S_5	1
S_2	S_5	S_4	0
S_3	S_1	S_6	1
S_4	S_5	S_2	0
S_5	S_4	S_3	0
S_6	S_5	S_6	0

S_1	S_3-S_1 S_5-S_2					
S_2	☒	☒				
S_3	S_1-S_1 S_6-S_2	S_1-S_3 S_6-S_5	☒			
S_4	☒	☒	S_5-S_5 S_4-S_4	☒		
S_5	☒	☒	S_4-S_5 S_3-S_4	☒	S_4-S_5 S_3-S_2	
S_6	☒	☒	S_5-S_5 S_6-S_4	☒	S_5-S_5 S_6-S_2	S_5-S_4 S_6-S_3
	S_0	S_1	S_2	S_3	S_4	S_5





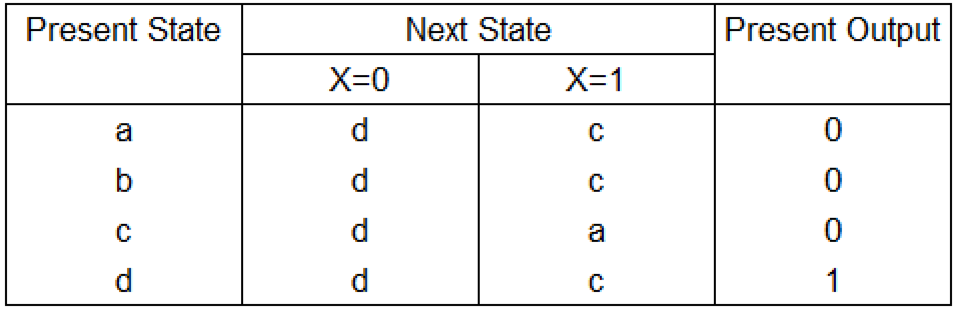
S_1	~~S_3-S_1~~ ~~S_5-S_2~~					
S_2	☒	☒				
S_3	S_1-S_1 S_6-S_2	~~S_1-S_3~~ ~~S_6-S_5~~	☒			
S_4	☒	☒	S_5-S_5 S_4-S_4	☒		
S_5	☒	☒	☒	☒	☒	
S_6	☒	☒	S_5-S_5 S_6-S_4	☒	S_5-S_5 S_6-S_2	☒
	S_0	S_1	S_2	S_3	S_4	S_5



Present State	Next State		Output
	$x = 0$	$x = 1$	
S_0^*	S_1	S_2	1
S_1	S_0^*	S_5	1
S_2^*	S_5	S_2^*	0
S_5	S_2^*	S_0^*	0

Example (2)

Inspection Method



Present State	Next State		Present Output
	X=0	X=1	
a	d	c	0
b	d	c	0
c	d	a	0
d	d	c	1

Present state	Next Sstate		Output
	X=0	X=1	
a	d	c	0
c	d	a	0
d	d	c	1

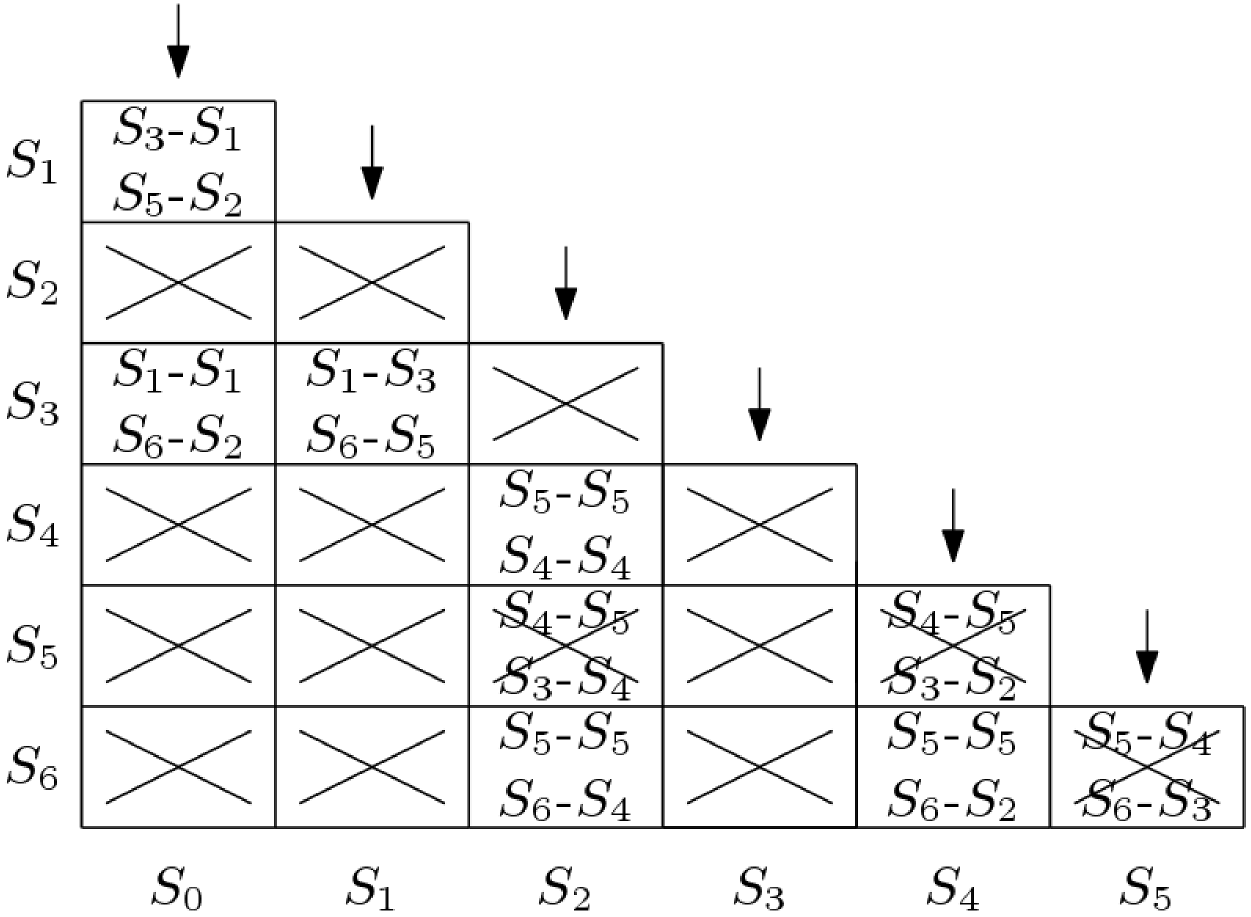
Present State	Next State		Output
	X=0	X=1	
a	d	a	0
d	d	a	1

Example (3) Inspection Method

Present State	Next State		Output	
	X=0	X=1	X=0	X=1
A	B	C	0	1
B	B	A	0	1
C	D	B	1	0
D	D	A	0	1

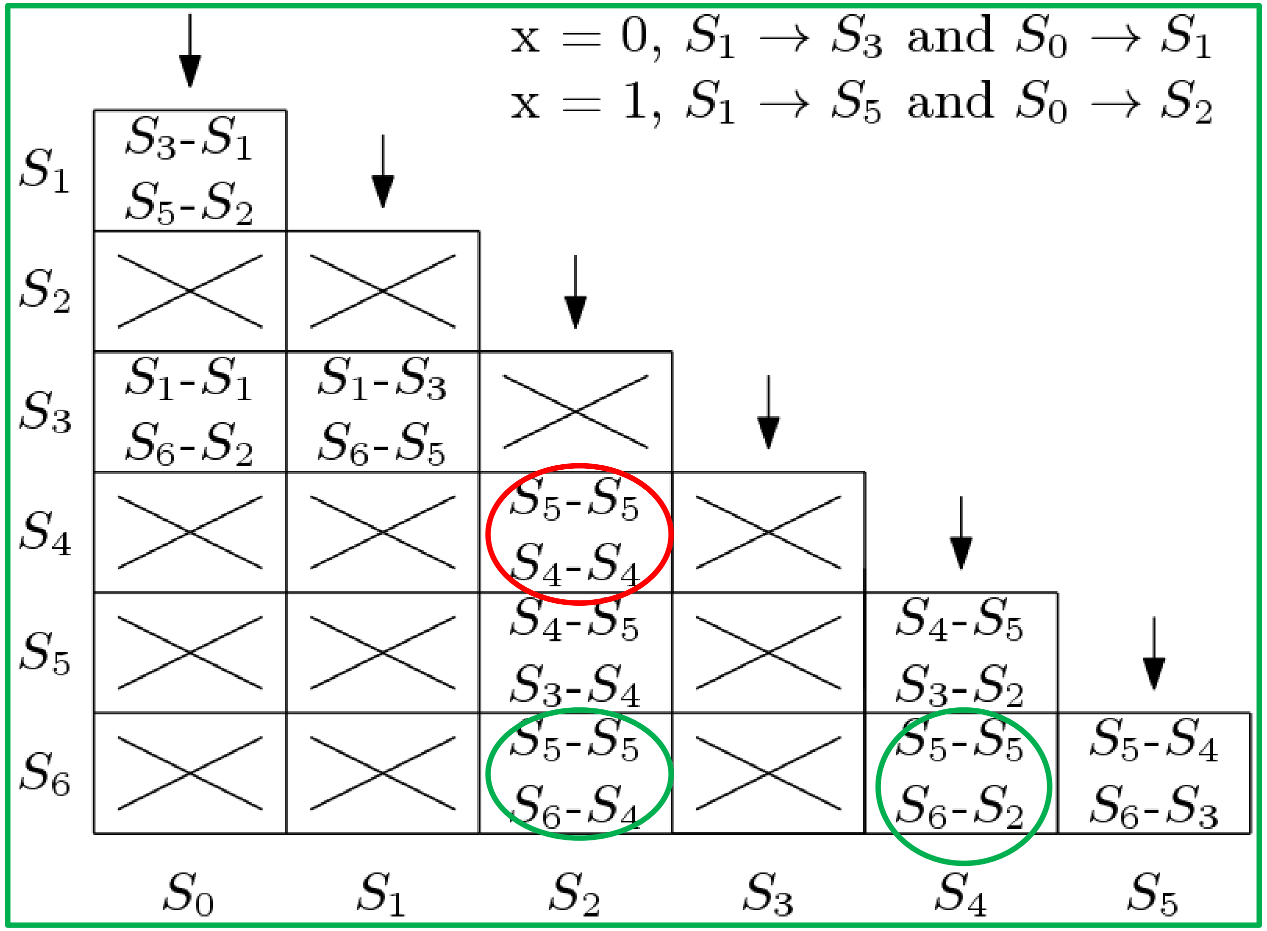
Present State	Next State		Output	
	X=0	X=1	X=0	X=1
A	B	C	0	1
B	B	A	0	1
C	B	B	1	0

Present State	Next State		Output
	$x = 0$	$x = 1$	
S_0	S_1	S_2	1
S_1	S_3	S_5	1
S_2	S_5	S_4	0
S_3	S_1	S_6	1
S_4	S_5	S_2	0
S_5	S_4	S_3	0
S_6	S_5	S_6	0



S_1	S_3-S_1 S_5-S_2					
S_2	✕	✕				
S_3	S_1-S_1 S_6-S_2	S_1-S_3 S_6-S_5	✕			
S_4	✕	✕	S_5-S_5 S_4-S_4	✕		
S_5	✕	✕	~~S_4-S_5~~ ~~S_3-S_4~~	✕	~~S_4-S_5~~ ~~S_3-S_2~~	
S_6	✕	✕	S_5-S_5 S_6-S_4	✕	S_5-S_5 S_6-S_2	~~S_5-S_4~~ ~~S_6-S_3~~
	S_0	S_1	S_2	S_3	S_4	S_5

	↓	↓	↓	↓	↓	↓
S_1	~~S_3-S_1~~ ~~S_5-S_2~~					
S_2	✗	✗				
S_3	S_1-S_1 S_6-S_2	~~S_1-S_3~~ ~~S_6-S_5~~	✗			
S_4	✗	✗	S_5-S_5 S_4-S_4	✗		
S_5	✗	✗	✗	✗	✗	
S_6	✗	✗	S_5-S_5 S_6-S_4	✗	S_5-S_5 S_6-S_2	✗
	S_0	S_1	S_2	S_3	S_4	S_5



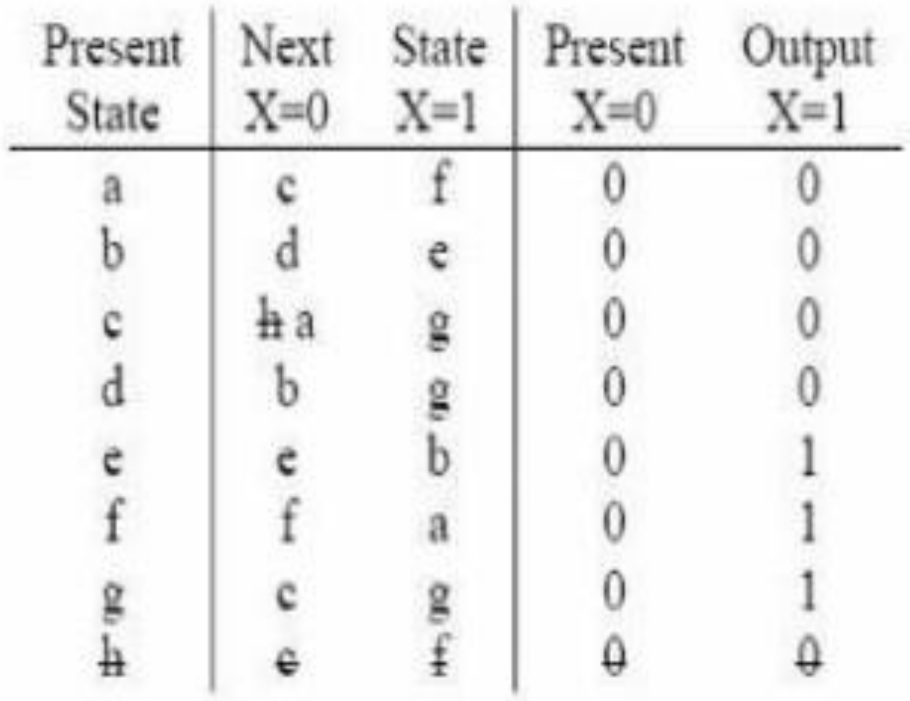
x = 0, $S_1 \to S_3$ and $S_0 \to S_1$

x = 1, $S_1 \to S_5$ and $S_0 \to S_2$

S_1	S_3-S_1 S_5-S_2					
S_2	✕	✕				
S_3	S_1-S_1 S_6-S_2	S_1-S_3 S_6-S_5	✕			
S_4	✕	✕	S_5-S_5 S_4-S_4	✕		
S_5	✕	✕	S_4-S_5 S_3-S_4	✕	S_4-S_5 S_3-S_2	
S_6	✕	✕	S_5-S_5 S_6-S_4	✕	S_5-S_5 S_6-S_2	S_5-S_4 S_6-S_3
	S_0	S_1	S_2	S_3	S_4	S_5

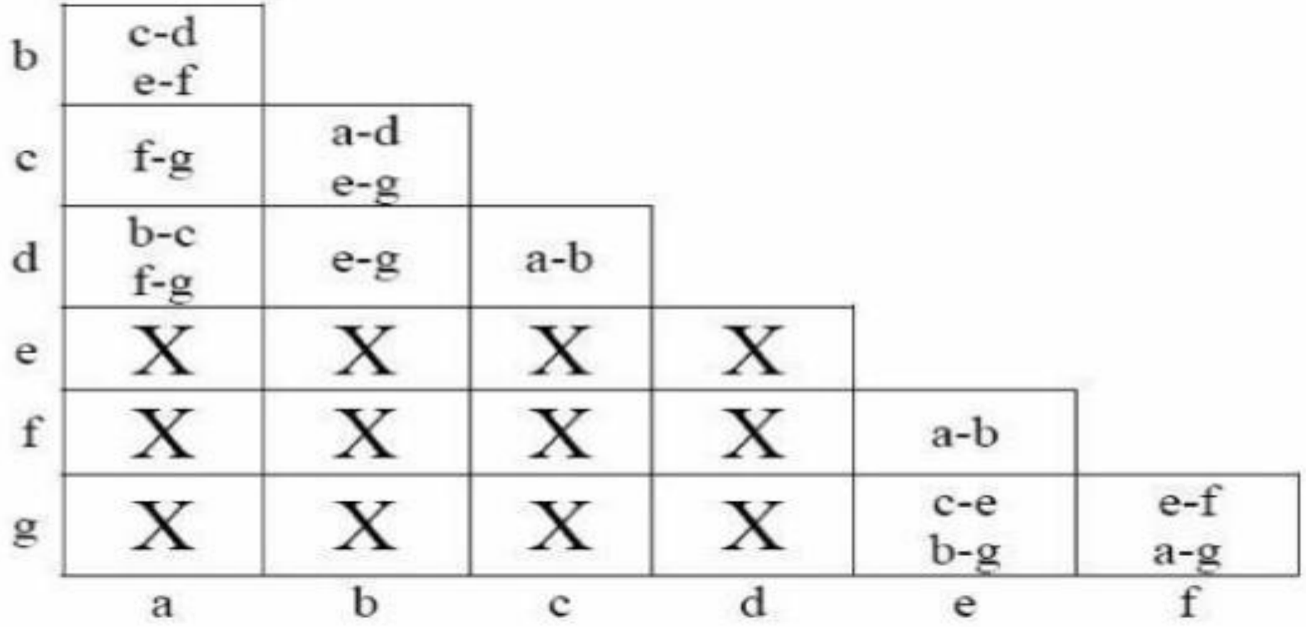
S2=S4

Present State	Next State		Output
	$x = 0$	$x = 1$	
S_0^*	S_1	S_2	1
S_1	S_0^*	S_5	1
S_2^*	S_5	S_2^*	0
S_5	S_2^*	S_0^*	0



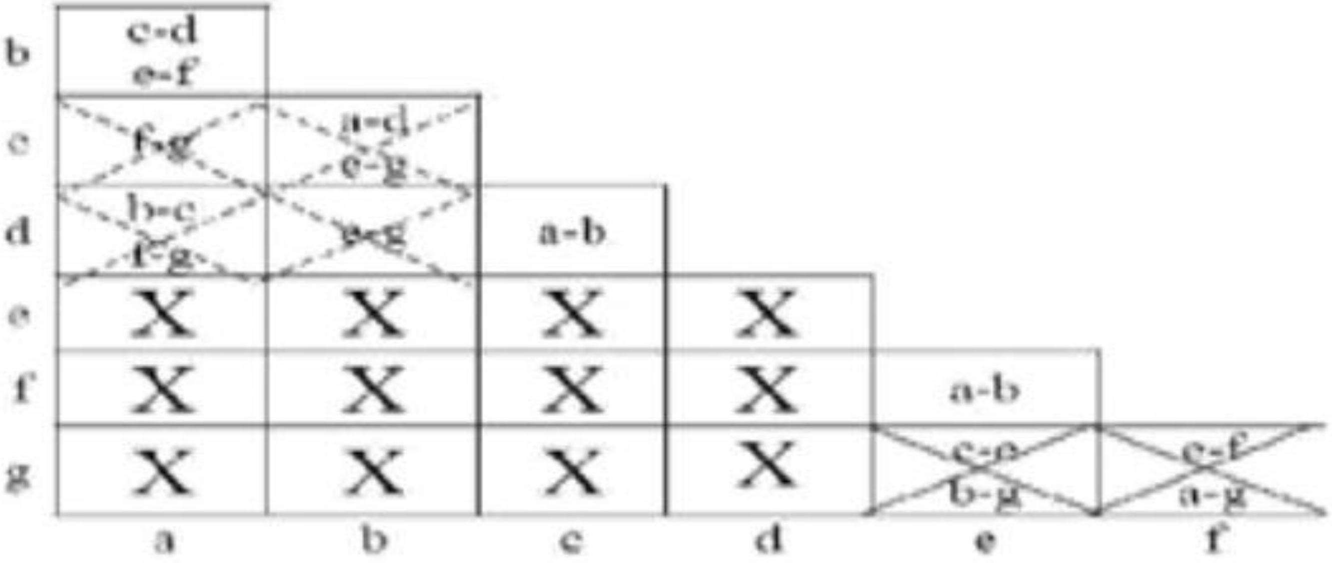
Present	Next	State	Present	Output
State	X=0	X=1	X=0	X=1
a	c	f	0	0
b	d	e	0	0
c	~~h~~ a	g	0	0
d	b	g	0	0
e	e	b	0	1
f	f	a	0	1
g	c	g	0	1
~~h~~	~~e~~	~~f~~	~~0~~	~~0~~

Figure 2 State Table Reduction by Row Matching



	a	b	c	d	e	f
b	c-d e-f					
c	f-g	a-d e-g				
d	b-c f-g	e-g	a-b			
e	X	X	X	X		
f	X	X	X	X	a-b	
g	X	X	X	X	c-e b-g	e-f a-g

Figure 3 Implication Chart (First Pass)



b	c-d e-f					
c	~~f-g~~	~~a-d~~ ~~e-g~~				
d	~~b-c~~ ~~f-g~~	~~e-g~~	a-b			
e	X	X	X	X		
f	X	X	X	X	a-b	
g	X	X	X	X	~~c-e~~ ~~b-g~~	~~e-f~~ ~~a-g~~
	a	b	c	d	e	f

Figure 4After Second and Third Pass

Present	Next	State	Present	Output
State	X=0	X=1	X=0	X=1
a	c	E	0	0
c	A	g	0	0
e	e	A	0	1
g	e	g	0	1

Figure 5 Final Reduced Table

Present State	Next State X = 0	1	Present Output
a	*d*	*c*	0
b	*f*	*h*	0
c	*e*	*d*	1
d	*a*	*e*	0
e	*c*	*a*	1
f	*f*	*b*	1
g	*b*	*h*	0
h	*c*	*g*	1

Original State Table

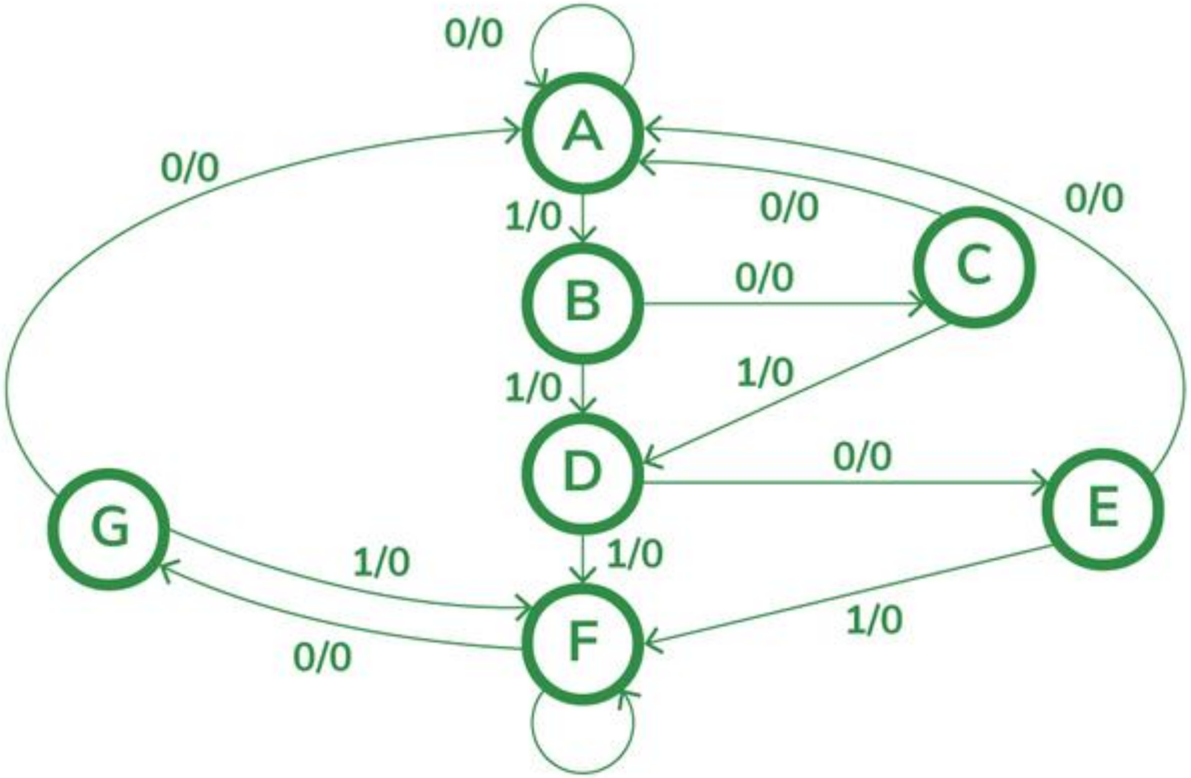
b	~~*d–f*~~ ~~*c–h*~~						
c	☒	☒					
d	*c–e*	~~*a–f*~~ ~~*e–h*~~	☒				
e	☒	☒	*a–d*	☒			
f	☒	☒	~~*e–f*~~ ~~*b–d*~~	☒	~~*c–f*~~ ~~*a–b*~~		
g	~~*b–d*~~ ~~*c–h*~~	~~*b–f*~~	☒	~~*a–b*~~ ~~*e–h*~~	☒	☒	
h	☒	☒	~~*c–e*~~ ~~*d–g*~~	☒	~~*a–g*~~	~~*c–f*~~ ~~*b–g*~~	☒
	a	*b*	*c*	*d*	*e*	*f*	*g*

Second Pass

Present State	Next State X = 0	1	Output
a	*a*	*c*	0
b	*f*	*h*	0
c	*c*	*a*	1
f	*f*	*b*	1
g	*b*	*h*	0
h	*c*	*g*	1

Reduced State Table

-rows d, e eliminated


0/0
A
0/0
1/0
0/0
0/0
B
0/0
C
1/0
1/0
D
0/0
E
G
1/0
1/0
F
0/0
1/0

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

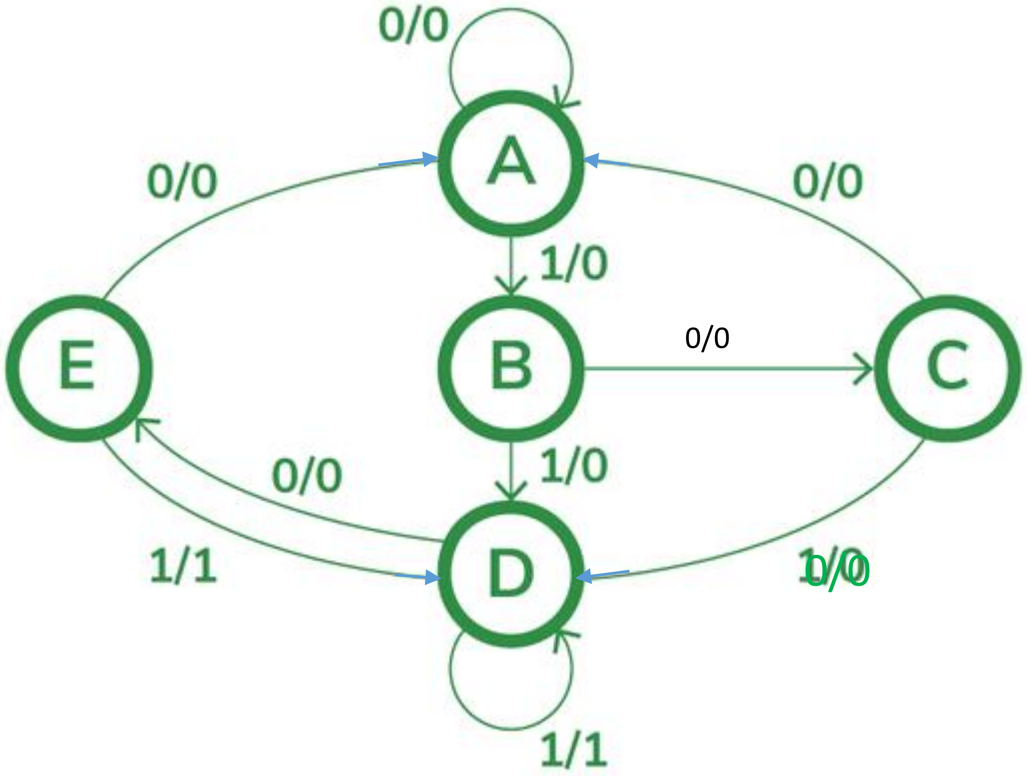
E=G , delet G

Substitute each G by E

Then D=F, delet F

Substitute each F by D

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


0/0
A
0/0
0/0
1/0
E
B
0/0
C
1/0
0/0
1/1
D
1/0
0/0
1/1