

BIT 203

DIGITAL ELECTRONICS & LOGIC DESIGN

Instruction	4 Periods Per week
Duration of Examination	3 Hours
Univ. Exam	75 Marks
Sessionals	25 Marks

Course Objectives:

1. To learn the principles of digital hardware and support given by it to the software.
2. To explain the operation and design of combinational and arithmetic logic circuits.
3. To design hardware for real world problems.

UNIT – I

Design Concepts – Digital Hardware, Design process, Design of digital hardware Introduction to logic circuits – Variables and functions, Logic gates and networks. Boolean algebra, Synthesis using AND, OR, and NOT Gates, Design examples. Optimized implementation of logic functions – Karnaugh Map, Strategies for minimization, minimizing Product-of-Sum Forms, Incompletely Specified functions, multiple output circuits. NAND and NOR logic networks, Introduction to CAD tools and Very High Speed Integrated Circuit Hardware Description Language (VHDL).

UNIT – II

Programmable logic devices: general structure of a Programmable Logic Array (PLA), gate level diagram, schematic diagram, Programmable Array Logic (PAL) Structure of CPLDs and FPGAs, 2-input and 3-input lookup tables (LUT). Design of Arithmetic-circuits, VHDL for Arithmetic-circuits Combinational circuit building blocks – Multiplexers, Decoders, Encoders, Code converters, Arithmetic comparison circuits. VHDL for Combinational circuits.

UNIT – III

Basic Latch Gated SR Latch, Gated D Latch, Master-Slave and Edge- Triggered D Flip-Flops, T Flip-flop, JK Flip-flop, Excitation tables. Registers-Shift Register, Counters-Asynchronous and synchronous counters, Ring counter, Johnson counter, VHDL code for D Flip-flop and Up-counter

UNIT – IV

Synchronous Sequential Circuits – Basic design steps. Moore and Mealy state model, State minimization, Design of a Counter using the Sequential Circuit Approach. Algorithmic State Machine (ASM) charts

UNIT – V

Asynchronous Sequential Circuits – Behaviour, Analysis, Synthesis, State reduction, State Assignment, examples. Hazards: static and dynamic hazards. Significance of Hazards. Clock skew, set up and hold time of a flip-flop

Suggested Reading:

1. Stephen Brown, Zvonko Vranesic, “Fundamentals of Digital Logic with VHDL Design”, 2nd Edition, McGraw Hill, 2009.
2. Jain R.P., “Modern Digital Electronics,” 3rd Edition, TMH, 2003.
3. John F. Wakerly, “Digital Design Principles & Practices”, 3rd Edition, Prentice Hall, 2001
4. M. Morris Mano, Charles R. Kime, “Logic and Computer Design Fundamentals”, 2nd Edition, Pearson Education Asia, 2001.
5. ZVI Kohavi, Switching and Finite Automata Theory, 2nd Edition, Tata McGraw Hill, 1995.
6. William I Fletcher, “An Engineering Approach to Digital Design”, Eastern Economy Edition, PHI
7. H.T. Nagle, “Introduction to Computer Logic”, Prentice Hall, 1975.

UNIT – I

Design Concepts – Digital Hardware, Design process, Design of digital hardware Introduction to logic circuits – Variables and functions, Logic gates and networks. Boolean algebra, Synthesis using gates, Design examples. Optimized implementation of logic functions – Karnaugh Map, Strategies for minimization minimizing product-of-sum functions. Multiple output circuits. NAND and NOR logic networks Introduction to CAD tools and VHDL.

DESIGN CONCEPTS

Design is the blue print of the system.

HERE WE ARE GOING TO LEARN ABOUT THE TOPICS LIKE

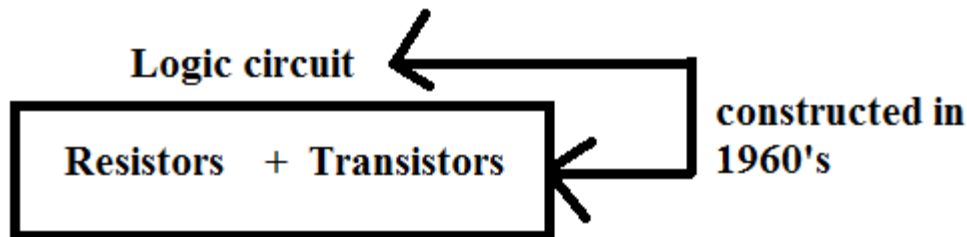
1. Digital Hardware.
2. Design process.
3. Design of digital hardware

NOTE

Computer consists of logic circuits and logic circuits are designed using CAD tools.

DIGITAL HARDWARE

1. In 1960s logic circuits were constructed with bulky components, such as transistors and resistors that came as individual parts.



2. Integrated circuit chips are manufactured on a silicon wafer and wafer is cut to produce the individual chips, which are then placed inside a special type of chip package.

3. GORDON MOORE LAW OR MOORE'S LAW

It states that **the number of transistors will be doubled that could be placed on a chip every 1.5 to 2 years.**

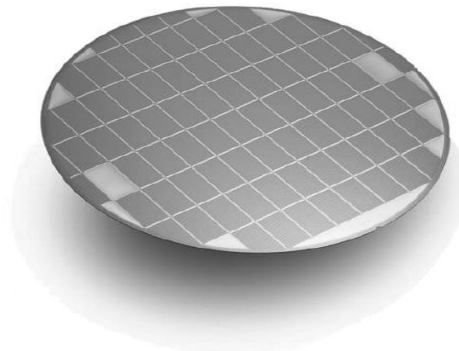


Figure 1.1 A silicon wafer (courtesy of Altera Corp.).

4. The designer place the logic circuits on a single chip, designing circuits involve number of chips placed on PCB (printed circuit board).

5. Logic circuit ----→transistors + resistors-----→chip-----→wafer

Table 1.1 A sample of the International Technology Roadmap for Semiconductors.

	Year					
	2006	2007	2008	2009	2010	2012
Technology feature size	78 nm	68 nm	59 nm	52 nm	45 nm	36 nm
Transistors per cm ²	283 M	357 M	449 M	566 M	714 M	1,133 M
Transistors per chip	2,430 M	3,061 M	3,857 M	4,859 M	6,122 M	9,718 M

TYPES OF CHIPS

1. Standard chips.
2. Programmable logic devices
3. custom chips.

STANDARD CHIPS

It chip contains a small amount of circuitry (<100 transistors) and performs a **simple function and fixed functionality**.

DRAWBACK

In standard chips is that the functionality of each chip is fixed and cannot be changed.

PROGRAMMABLE LOGIC DEVICES (PLD)

1. The PLD's can be configured by the user to implement a wide range of different logic circuits.
2. The chip is a collection of programmable switches that allow the internal circuitry in the chip to be configured in many different ways.
3. The designer can implement whatever functions are needed for a particular application by choosing an appropriate configuration of the switches. The switches are programmed by the end user, rather than when the chip is manufactured. Such chips are known as programmable logic devices (PLDs).
4. PLDs can be programmed multiple times. This capability is advantageous because a designer who is developing a prototype of a product can program a PLD to perform some function.
5. PLD example is field-programmable gate array (FPGA).

CUSTOM CHIPS OR SEMI-CUSTOM DESIGN OR APPLICATION-SPECIFIC INTEGRATED CIRCUITS (ASICS)

Chips are intended for use in specific applications and are sometimes called application-specific integrated circuits (ASICs).

The main advantage of a custom chip is that its design can be optimized for a specific task; hence it usually leads to better performance. It is possible to include a larger amount of logic circuitry in a custom chip than would be possible in other types of chips. The cost of producing such chips is high, but if they are used in a product that is sold in large quantities, then the cost per chip will be reduced

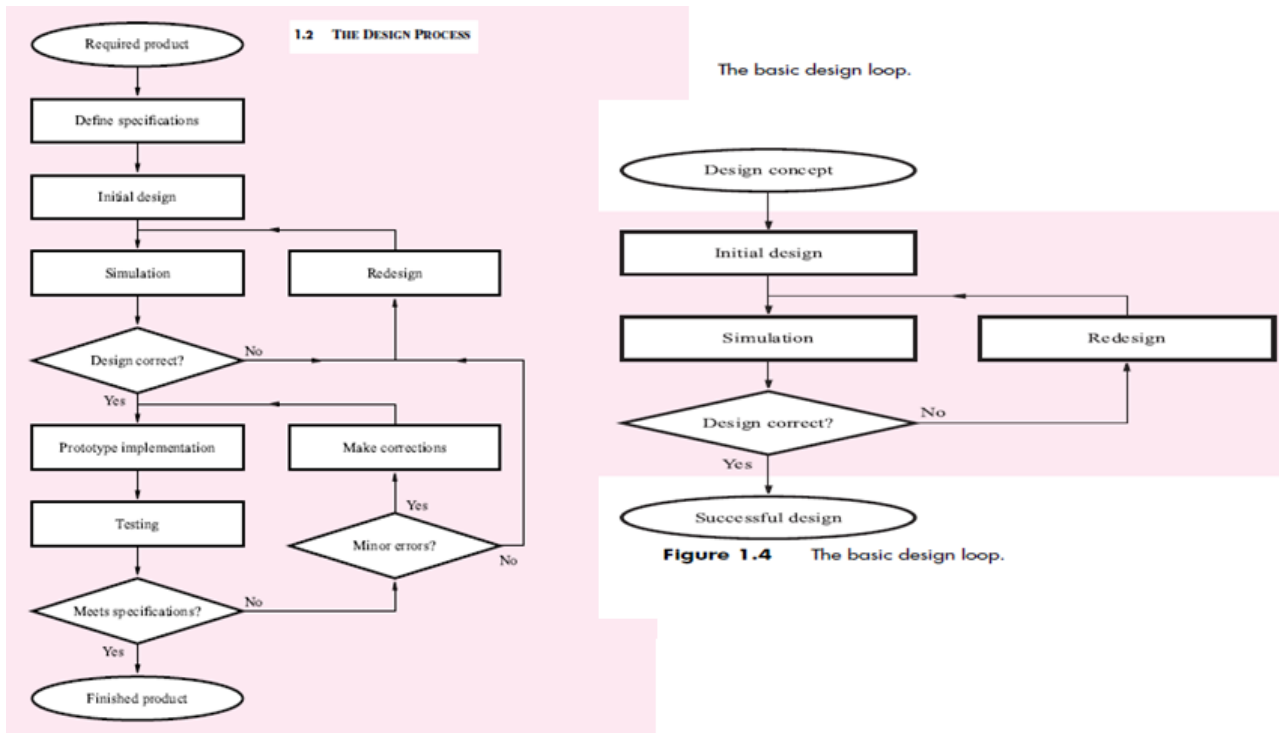
A disadvantage of the custom-design approach is that manufacturing a custom chip often takes a considerable amount of time, on the order of months.

1. The Design Process

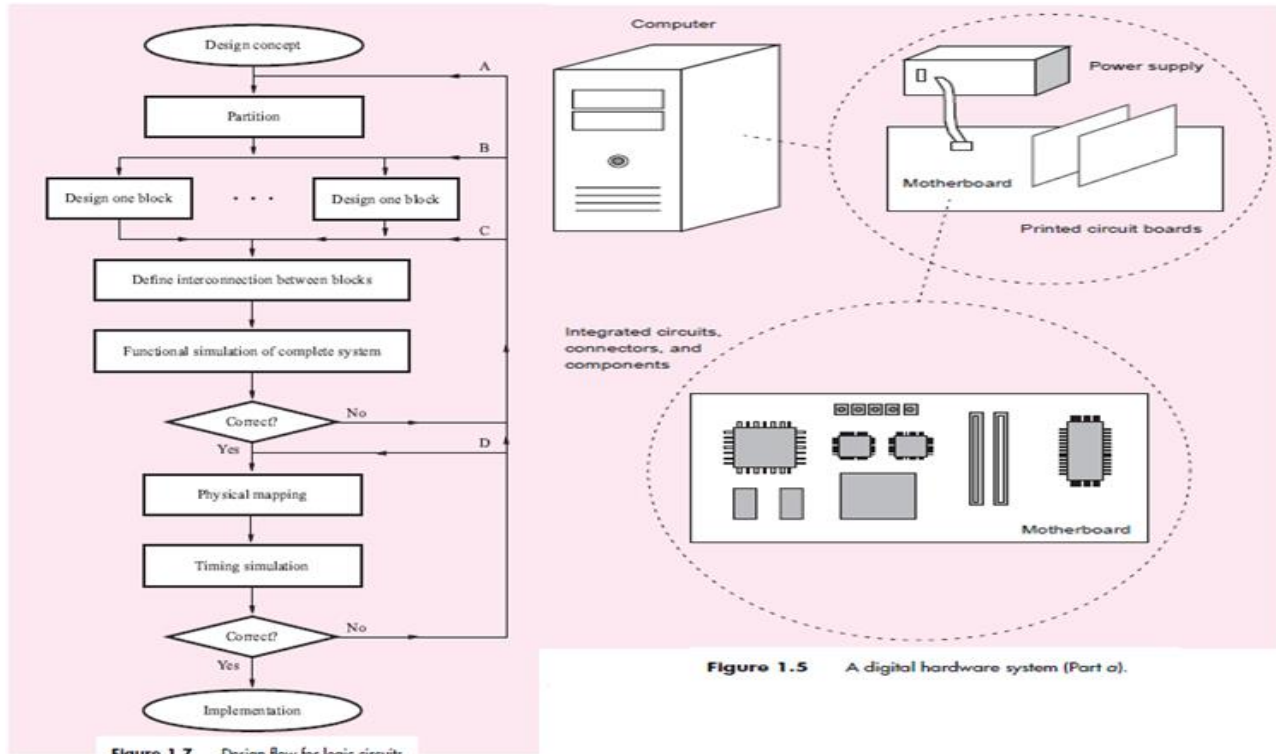
Design of Digital Hardware

1. Basic Design Loop.
2. Structure of a Computer.
3. Design of a Digital Hardware Unit.

THE DESIGN PROCESS, BASIC DESIGN LOOP



STRUCTURE OF A COMPUTER, DESIGN OF A DIGITAL HARDWARE UNIT

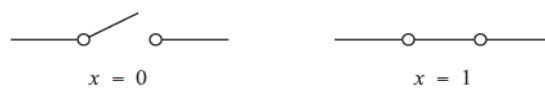


VARIABLE

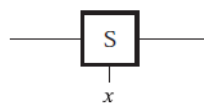
1. It is a data named used to store a data value.
2. In digital electronics we are going to deal with electronic signals which are represented by voltages 0.5 and 3.5 volts.

VOLTAGE	SIGNAL
0.5	0
3.5	1

3. The variables which we use in digital electronics that deals with binary values (0/1) are called as binary variables.
4. Electronic signals (voltages) can be in one of the states.
I.e. each voltage (signal) represented by a binary value.
5. Binary element is a switch that has two states. If a given switch is controlled by an input variable x , then we will say that the switch is open if $x = 0$ and closed if $x = 1$



(a) Two states of a switch

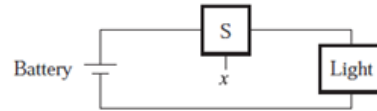


(b) Symbol for a switch

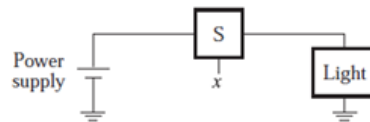
Figure 2.1 A binary switch.

FUNCTION

1. The output of a binary expression is assigned to a function. Or
2. It gives the relationship between input variables and output variables.
3. Boolean function consists of Boolean variables.

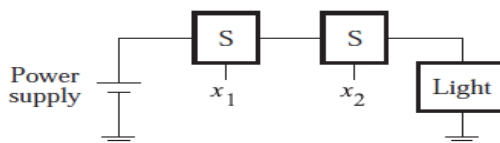


(a) Simple connection to a battery

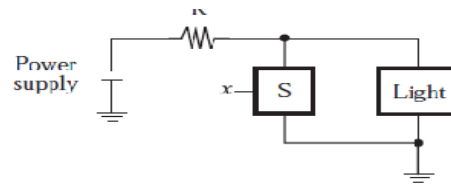


(b) Using a ground connection as the return path

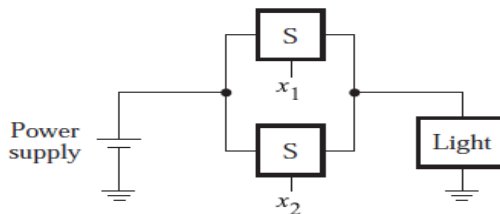
A light controlled by a switch.



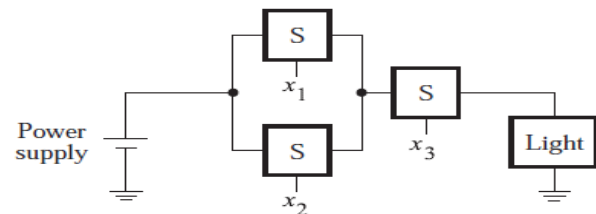
(a) The logical AND function (series connection)



An inverting circuit.



(b) The logical OR function (parallel connection)











A series-parallel connection.

LOGIC GATES AND NETWORKS

1. Each logic operation can be implemented electronically with transistors, resulting in a circuit element called a logic gate.
2. A logic gate has one or more inputs and one output that is a function of its inputs.
3. A logic gates are represented by graphical symbols. The graphical symbols for the AND, OR, and NOT gates are given below

Gate	Graphic Symbol	Truth Table
------	----------------	-------------

Name																				
Buffer		<table border="1"> <thead> <tr> <th colspan="2">NOT gate</th> </tr> <tr> <th>A</th> <th>\bar{A}</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> </tr> </tbody> </table>	NOT gate		A	\bar{A}	0	0	1	1										
NOT gate																				
A	\bar{A}																			
0	0																			
1	1																			
NOT		<table border="1"> <thead> <tr> <th colspan="2">NOT gate</th> </tr> <tr> <th>A</th> <th>\bar{A}</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> </tr> </tbody> </table>	NOT gate		A	\bar{A}	0	1	1	0										
NOT gate																				
A	\bar{A}																			
0	1																			
1	0																			
AND		<table border="1"> <thead> <tr> <th colspan="3">2 Input AND gate</th> </tr> <tr> <th>A</th> <th>B</th> <th>$A \cdot B$</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </tbody> </table>	2 Input AND gate			A	B	$A \cdot B$	0	0	0	0	1	0	1	0	0	1	1	1
2 Input AND gate																				
A	B	$A \cdot B$																		
0	0	0																		
0	1	0																		
1	0	0																		
1	1	1																		
NAND		<table border="1"> <thead> <tr> <th colspan="3">2 Input NAND gate</th> </tr> <tr> <th>A</th> <th>B</th> <th>$\overline{A \cdot B}$</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> </tr> </tbody> </table>	2 Input NAND gate			A	B	$\overline{A \cdot B}$	0	0	1	0	1	1	1	0	1	1	1	0
2 Input NAND gate																				
A	B	$\overline{A \cdot B}$																		
0	0	1																		
0	1	1																		
1	0	1																		
1	1	0																		
OR		<table border="1"> <thead> <tr> <th colspan="3">2 Input OR gate</th> </tr> <tr> <th>A</th> <th>B</th> <th>$A + B$</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </tbody> </table>	2 Input OR gate			A	B	$A + B$	0	0	0	0	1	1	1	0	1	1	1	1
2 Input OR gate																				
A	B	$A + B$																		
0	0	0																		
0	1	1																		
1	0	1																		
1	1	1																		
NOR		<table border="1"> <thead> <tr> <th colspan="3">2 Input NOR gate</th> </tr> <tr> <th>A</th> <th>B</th> <th>$\overline{A + B}$</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> </tr> </tbody> </table>	2 Input NOR gate			A	B	$\overline{A + B}$	0	0	1	0	1	0	1	0	0	1	1	0
2 Input NOR gate																				
A	B	$\overline{A + B}$																		
0	0	1																		
0	1	0																		
1	0	0																		
1	1	0																		
XOR		<table border="1"> <thead> <tr> <th colspan="3">2 Input EXOR gate</th> </tr> <tr> <th>A</th> <th>B</th> <th>$A \oplus B$</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> </tr> </tbody> </table>	2 Input EXOR gate			A	B	$A \oplus B$	0	0	0	0	1	1	1	0	1	1	1	0
2 Input EXOR gate																				
A	B	$A \oplus B$																		
0	0	0																		
0	1	1																		
1	0	1																		
1	1	0																		
XNOR		<table border="1"> <thead> <tr> <th colspan="3">2 Input EXNOR gate</th> </tr> <tr> <th>A</th> <th>B</th> <th>$\overline{A \oplus B}$</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </tbody> </table>	2 Input EXNOR gate			A	B	$\overline{A \oplus B}$	0	0	1	0	1	0	1	0	0	1	1	1
2 Input EXNOR gate																				
A	B	$\overline{A \oplus B}$																		
0	0	1																		
0	1	0																		
1	0	0																		
1	1	1																		

NOTE: NAND and NOR gates are called as universal gates because by using these gates you can implement any gate.

NETWORK

1. It is non thing but a collection of gates and there interconnection.
2. Conversion of network diagram to function is called as **Analysis Process**.

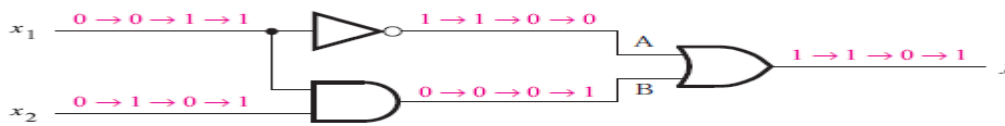
Analysis Process

Network diagram----->**Function.**

3. Conversion of function to network diagram is called as **Synthesis Process**.

Synthesis Process

Function -----> **Network diagram.**



(a) Network that implements $f = \bar{x}_1 + x_1 \cdot x_2$

x_1	x_2	$f(x_1, x_2)$	A	B
0	0	1	1	0
0	1	1	1	0
1	0	0	0	0
1	1	1	0	1

(b) Truth table

BOOLEAN ALGEBRA

1. Boolean algebra is used for simplification of a Boolean expression.
2. Boolean expression consists of Boolean variables.
3. Boolean algebra uses Boolean identities or Boolean rules for simplification of Boolean expressions.
4. Boolean algebra is based on a set of rules that are derived from a small number of basic assumptions Called as axioms.
5. Boolean rules are called as theorems.

BOOLEAN IDENTITIES OR BOOLEAN RULES

AXIOMS OF BOOLEAN ALGEBRA	SINGLE-VARIABLE THEOREMS

1a. $0 \cdot 0 = 0$	5a. $x \cdot 0 = 0$
1b. $1 + 1 = 1$	5b. $x + 1 = 1$
2a. $1 \cdot 1 = 1$	6a. $x \cdot 1 = x$
2b. $0 + 0 = 0$	6b. $x + 0 = x$
3a. $0 \cdot 1 = 1 \cdot 0 = 0$	7a. $x \cdot x = x$
3b. $1 + 0 = 0 + 1 = 1$	7b. $x + x = x$
4a. If $x = 0$, then $\bar{x} = 1$	8a. $x \cdot \bar{x} = 0$
4b. If $x = 1$, then $\bar{x} = 0$	8b. $x + \bar{x} = 1$
	9. $\bar{\bar{x}} = x$

TWO- AND THREE-VARIABLE PROPERTIES

10a. $x \cdot y = y \cdot x$ <i>Commutative</i>	13b. $x \cdot (x + y) = x$
10b. $x + y = y + x$	14a. $x \cdot y + x \cdot \bar{y} = x$ <i>Combining</i>
11a. $x \cdot (y \cdot z) = (x \cdot y) \cdot z$ <i>Associative</i>	14b. $(x + y) \cdot (x + \bar{y}) = x$
11b. $x + (y + z) = (x + y) + z$	15a. $\overline{x \cdot y} = \bar{x} + \bar{y}$ <i>DeMorgan's theorem</i>
12a. $x \cdot (y + z) = x \cdot y + x \cdot z$ <i>Distributive</i>	15b. $\overline{x + y} = \bar{x} \cdot \bar{y}$
12b. $x + y \cdot z = (x + y) \cdot (x + z)$	16a. $x + \bar{x} \cdot y = x + y$
13a. $x + x \cdot y = x$ <i>Absorption</i>	16b. $x \cdot (\bar{x} + y) = x \cdot y$
	17a. $x \cdot y + y \cdot z + \bar{x} \cdot z = x \cdot y + \bar{x} \cdot z$ <i>Consensus</i>
	17b. $(x + y) \cdot (y + z) \cdot (\bar{x} + z) = (x + y) \cdot (\bar{x} + z)$

x	y	$x \cdot y$	$\overline{x \cdot y}$	\bar{x}	\bar{y}	$\bar{x} + \bar{y}$
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

LHS
RHS
 Proof of DeMorgan's theorem

SYNTHESIS USING GATES

Conversion of function to network diagram is called as **Analysis Process**.

Synthesis Process

Function -----> **Network diagram**.

MIN-TERMS

In a function of n variables, a product term in which each of the n variables appears once is called a min-term.

MAX-TERMS

The principle of duality suggests that if it is possible to synthesize a function f by considering the rows in the truth table for which $f = 1$, then it should also be possible to synthesize f by considering the rows for which $f = 0$. This alternative approach uses the complements of min-terms, which are called max-terms.

A logic expression consisting of sum (OR) terms that are the factors of a logical product (AND) is said to be of the product-of-sums (POS) form. If each sum term is a max-term, then the expression is called a canonical product-of-sums for the given function. Any function f can be synthesized by finding its **canonical product-of-sums**.

A logic expression consisting of product (AND) terms that are summed (ORed) is said to be of the sum-of-products (SOP) form. If each product term is a min-term, then the expression is called a **canonical sum-of-products** for the function f .

CANONICAL FORM (ALL LETTERS SHOULD BE PRESENT IN THE PRODUCT TERMS)

$$F = abc + ab1c + a1b1c1$$

NON-CANONICAL FORM (ONLY FEW LETTERS WILL BE PRESENT IN THE PRODUCT TERMS)

$$F = ac + ab1c + a1b1$$

DESIGN EXAMPLES

1. Three-Way Light Control.
2. Multiplexer Circuit.

THREE-WAY LIGHT CONTROL

A large room has three doors and that a switch near each door controls a light in the room. It has to be possible to turn the light on or off by changing the state of any one of the switches.

Let x_1 , x_2 , and x_3 be the input variables that denote the state of each switch. Assume that the light is off if all switches are open. Closing any one of the switches will turn the light on. Then turning on a second switch will have to turn off the light. Thus the light will be on if exactly one switch is closed, and it will be off if two (or no) switches are closed. If the light is off when two switches are closed, then it must be possible to turn it on by closing the third switch. If $f(x_1, x_2, x_3)$ represents the state of the light, then the required functional behavior can be specified as shown in the truth table in Figure

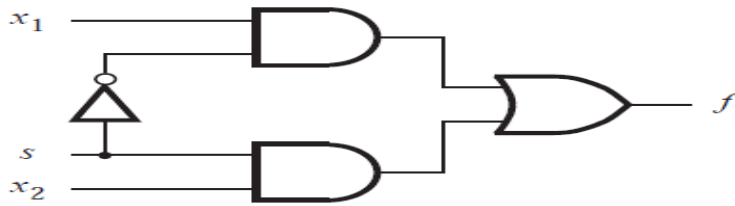
x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Truth table for the three-way light control.

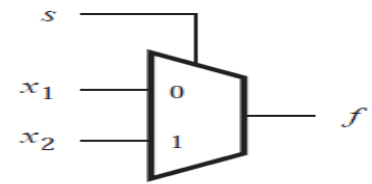
$$\begin{aligned}
 f &= m_1 + m_2 + m_4 + m_7 \\
 &= \bar{x}_1\bar{x}_2x_3 + \bar{x}_1x_2\bar{x}_3 + x_1\bar{x}_2\bar{x}_3 + x_1x_2x_3
 \end{aligned}$$

MULTIPLEXER CIRCUIT OR MUX OR DATA SELECTOR

It is used to send one of the inputs as outputs for a circuit. It consists of select pins for its operations.



(b) Circuit



(c) Graphical symbol

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

(d) More compact truth-table representation

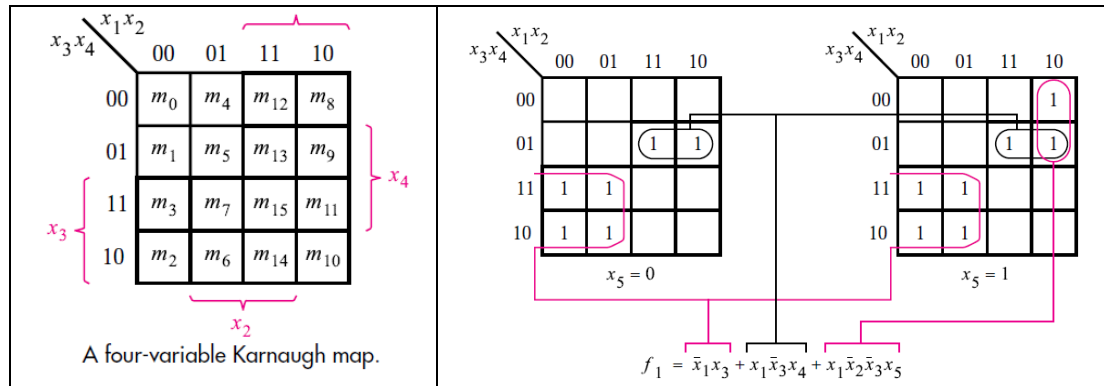
OPTIMIZED IMPLEMENTATION OF LOGIC FUNCTIONS

1. Karnaugh Map.
2. Strategies for minimization.
3. Minimizing product-of-sum functions

KARNAUGH MAP

It is used for the simplification of the Boolean expressions. The Karnaugh map approach provides a systematic way of performing this optimization.

TWO-VARIABLE MAP	THREE-VARIABLE MAP
FOUR-VARIABLE MAP	FIVE-VARIABLE MAP



STRATEGY FOR MINIMIZATION

Here we are going to minimize the circuit (number of gates present in the circuit) and find the cost for it.

TECHNIQUES USED FOR THIS MINIMIZATION ARE

1. KMAPS.
2. Product of sums form along with KMAP.
3. Using don't care condition.

KMAP TERMINOLOGY

In this we are going to minimize the circuit and find the cost for it.

LITERAL

Variables in a Boolean expression are called as literals. For example, the product term x^1yz has three literals, and the term ab^1c^1d has four literals.

IMPLICANT

A product term that indicates the input valuation(s) for which a given function is equal to 1 is called an implicant of the function.

PRIME IMPLICANT

An implicant is called a prime implicant if it cannot be combined into another implicant that has fewer literals.

COVER

A collection of implicants that account for all valuations for which a given function is equal to 1 is called a cover of that function. A number of different covers exist for most functions.

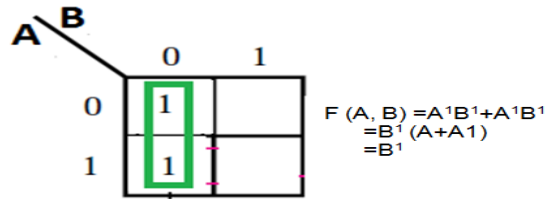
COST

The cost of a logic circuit is the number of gates plus the total number of inputs to all gates in the circuit.

Cost (logic circuit) = number of gates + Total number of inputs to all gates in the circuit.

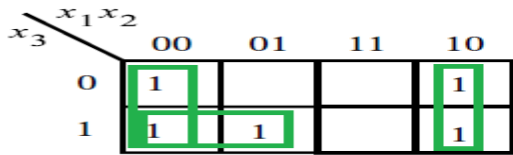
1. Simplify the Boolean function of 2 variables using KMAP method.

$F(A, B) = \Sigma(0, 2) = A^1B^1 + Ab^1$



2. Simplify the Boolean function of 3 variables using KMAP method.

$F(A, B, C) = \Sigma(0, 2, 4, 5, 6)$



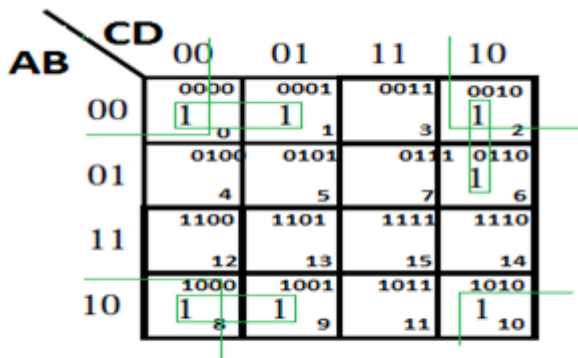
$F = A^1B^1C^1 + AB^1C^1 + AB^1C^1 + AB^1C^1 + A^1BC^1 + ABC^1$
 $F = B^1C^1(A + A^1) + AB^1(C + C^1) + BC^1(A + A^1)$
 $F = B^1C^1 + AB^1 + BC^1$
 $F = C^1(B + B^1) + AB^1$
 $F = C^1 + AB^1$

NOTE

A, B, C = 3 VARIABLES = n
 Number of cells in KMAP = 2^n

2. Simplify the Boolean function of 4 variables using KMAP method.

$F(A, B, C, D) = \Sigma(0, 1, 2, 6, 8, 9, 10)$



A four-variable Karnaugh map.

0, 2, 8, 10

0, 2, 8, 10 → $A^1B^1C^1D^1 + A^1B^1CD^1 = A^1B^1D^1$

8, 10 → $AB^1C^1D^1 + AB^1CD^1 = AB^1D^1$

0, 2, 8, 10 = B^1D^1

0, 1-----→ $A^1B^1C^1D^1+A^1B^1C^1D= A^1B^1C^1$
 2, 6-----→ $A^1B^1CD^1+A^1BCD^1= A^1CD^1$
 8, 9-----→ $AB^1C^1D^1+AB^1C^1D= AB^1C^1$

B^1D^1 $A^1B^1C^1$
 A^1CD^1 AB^1C^1
F (A, B, C, D) = $B^1D^1+A^1CD^1+B^1C^1$

DON'T CARE CONDITIONS

1. The 1's and 0's in map represent the min-terms that make the function equal to 1 or 0.
2. Some occasions when it don't matter if the function produces 0 or 1 for the given min-term.
3. since the function may be either 0 or 1. We say that we don't care what the function o/p is to be for this min-term.
4. Min-terms that may produce either 0 or 1 for the function are said to be don't care conditions and are marked with an "X" in a map.
5. These don't care conditions can be used to provide further simplification of the algebraic expression.
6. Some times for group we take 0's also to get optimal Boolean expression. The 0's are called don't 'cares.

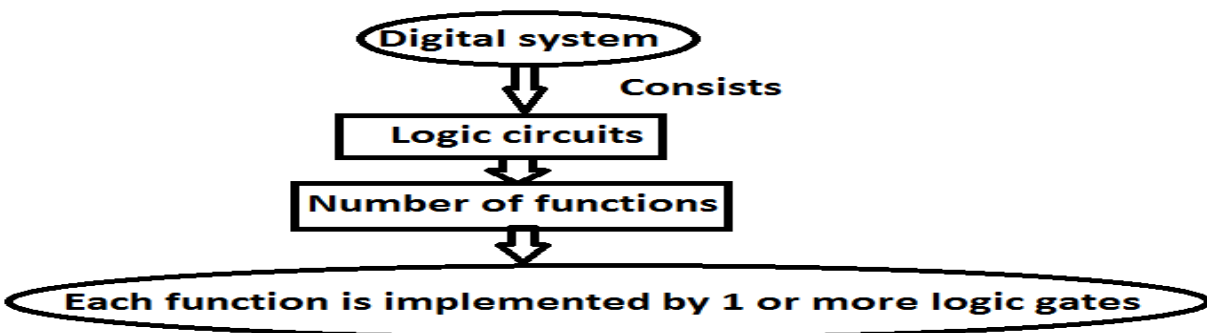
PRODUCT OF SUMS SIMPLIFICATION

Instead of grouping 1's in KMAP we group 0's in the KMAP then it is called as product of sums simplification.

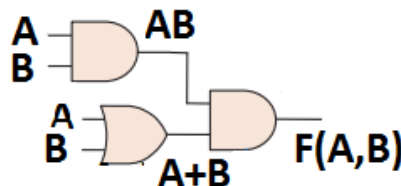
KMAP	Product of sums simplification
F----→1	F ¹ ----→0
Sum(products)	products (Sum)

MULTIPLE OUTPUT CIRCUITS

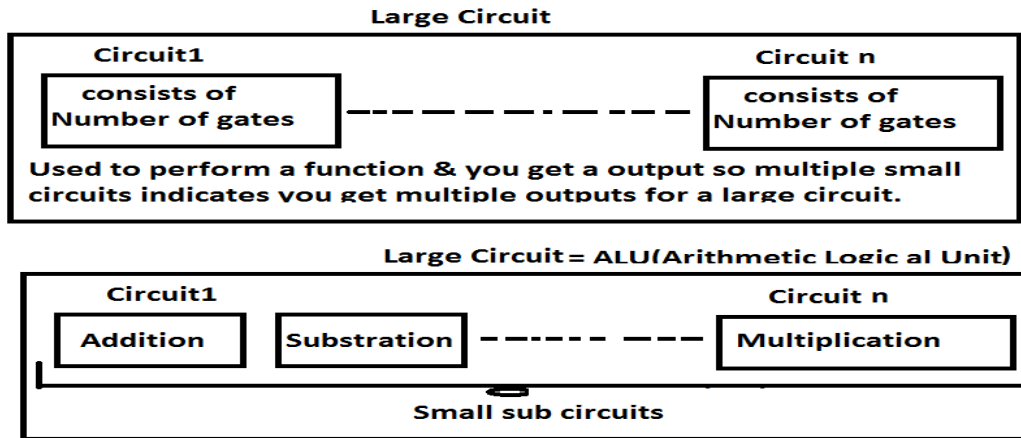
1. Any basic gate takes 1 or more inputs and produces a single output.
I.e. single output function.
2. In practice, these functions may be a part of the large circuit that has many such functions.
3. In digital system numbers of functions are as a part, each of some functional logic circuit.



I.e. circuits that implement these functions may be combined into a less costly single circuit with multiple outputs by sharing some gates needed in the implementation of the single functions.



$$F(A, B) = (AB (A+B))^{-1}$$



NAND AND NOR LOGIC NETWORKS

1. Logic network consists of only gates.
2. Logic network designed or constructed using only NOR gates called as NOR logic network.
3. Logic network designed or constructed using only NAND gates called as NAND logic network.
4. Logic network designed or constructed using both NAND AND NOR gates called as NAND AND NOR logic network.
5. NAND and NOR gates are called as universal gates because using these gates you can construct any gate.

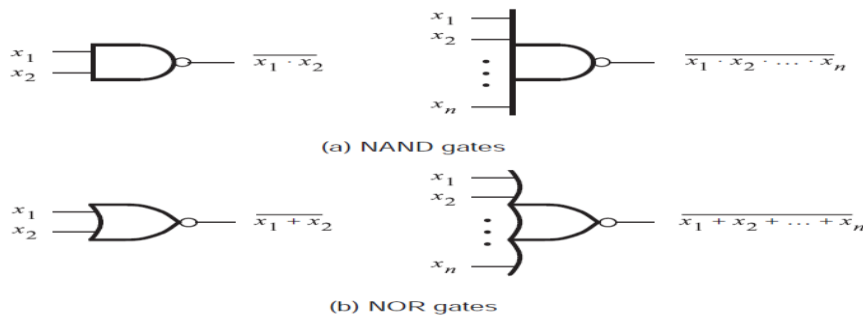


Figure 2.20 NAND and NOR gates.

INTRODUCTION TO CAD TOOLS AND VHDL.

1. A designer could use this approach manually for small circuits. However, logic circuits found in complex systems, such as today's computers, cannot be designed manually—they are designed using sophisticated CAD tools that automatically
2. CAD tools are present in CAD system.
3. CAD tools are used for 4 tasks
 - A. Design entry.
 - B. Synthesis and optimization.
 - C. Simulation.
 - D. Physical design.

DESIGN ENTRY

1. Designer enters circuit description into the CAD system for generating the circuit in automated mode called as Design entry.

2. DESIGN ENTRY METHODS (A, B)

- A. Truth table
- B. Using schematic capture.
- C. Writing source code in a hardware description language.

TRUTH TABLE

1. CAD system allows design entry using truth tables where the table is specified as a plain text file.
2. It is also possible to specify a truth table as a set of wave forms in the timing diagram.

DRAW BACK

Truth tables are practical only for functions with small number of variables.

SCHEMATIC CAPTURE

1. A logic circuit can be defined by drawing logic gates and interconnecting them with wires. A CAD tool for entering a designed circuit in this way is called a schematic capture tool. Schematic refers to a diagram of a circuit in which circuit elements, such as logic gates, are depicted as graphical symbols and connections between circuit elements are drawn as lines .
2. The collection of symbols is called a library.
3. The gates in the library can be imported into the user's schematic, and the tool provides a graphical way of interconnecting the gates to create a logic network.
4. In a CAD system user can create a circuit that includes within it other smaller circuits. This methodology is known as hierarchical design and provides a good way of dealing with the complexities of large circuits.
5. Hierarchical design created with the schematic capture tool provided with CAD system called as Graphic Editor.

DISADVANTAGE

1. Schematic is a commercial tool and has unique user interface and functionality and training required, circuit being designed should be large.
2. A useful method for dealing with large circuits is to write source code using a hardware description language to represent the circuit.

HARDWARE DESCRIPTION LANGUAGES

1. It is used to describe hardware rather than a program to be executed on a computer.
2. According to Institute of Electrical and Electronics Engineers (IEEE) there are two HDLs are IEEE standards:
 - A. VHDL (VERY HIGH SPEED INTEGRATED CIRCUIT HARDWARE DESCRIPTION LANGUAGE).
 - B. VERILOG HDL.

VHDL (VERY HIGH SPEED INTEGRATED CIRCUIT HARDWARE DESCRIPTION LANGUAGE)

1. Popular than verilog.
2. More advantages than verilog.
3. A circuit specified in VHDL can be implemented in different types of chips and with CAD tools provided by different companies, without having to change the VHDL specification.
4. **Design portability** is an important advantage because digital circuit technology changes rapidly.
5. VHDL encourages sharing and reuse of VHDL-described circuits.
6. Faster development of new products in cases where existing VHDL code can be adapted for use in the design of new circuits.

SYNTHESIS

1. It is the process of generating a logic circuit from an initial specification that may be given in the form of a schematic diagram or code written in a hardware description language.
2. The process of translating, or compiling, VHDL code into a network of logic gates is part of synthesis. The output is a set of logic expressions that describe the logic functions needed to realize the circuit.
3. CAD tools perform this process automatically.
4. Synthesis tools manipulate the users design to automatically practice an equivalent but better circuit.
5. Logic function represented by an expression using resources available in the technology. It involves 2 steps called a technology mapping.
 - A. Layout synthesis.
 - B. Physical design.

FUNCTIONAL SIMULATION SIMULATION

1. A model set of problems.
2. Events that can be used **to teach someone how to do something**.
3. The process of making such a model.

SIMULATOR

It is equivalent i.e designed to represent a model in real time conditions.

Types of simulation

- A. Behavioral or Functional simulation.
- B. Timing Simulation.

FUNCTIONAL SIMULATION

1. A circuit represented in the form of logic expressions can be simulated to verify that it will function as expected. The tool that performs this task is called a functional simulator.
2. **User's initial design** is represented by the logic equation generated during synthesis, and assumes that these expressions will be implemented with perfect gates through which signals propagate instantaneously. For each valuation, the simulator evaluates the outputs produced by the expressions.
3. **User specifies valuations** of the circuit's inputs that should be applied during simulation.

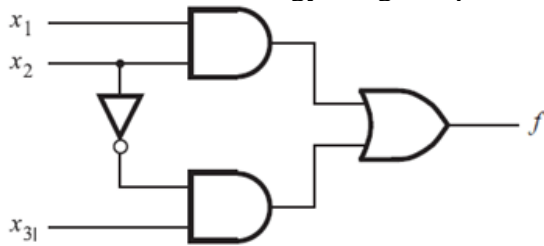
4. Output of simulation is provided in the form of a timing diagram or truth table which the user can examine to verify that the circuit operates as required.

PHYSICAL DESIGN

How to implement the circuit on a given chip is called as physical design. There are several different technologies that may be used to implement logic circuits. The physical design tools map a circuit specified in the form of logic expressions into a realization that makes use of the resources available on the target chip.

INTRODUCTION TO VHDL AND CAD TOOLS

1. It is a standard language (IEEE) for describing digital circuits.
IEEE adopted in 1987 called
2. IEEE-----→IEEE1076
IEEE adopted in 1993 called
3. IEEE-----→IEEE1164
4. It describes the structure of complex digital circuits.
5. Each logic signal in the circuit is represented in VHDL code as a data object.
6. VHDL code represents a logic circuit.
7. Inputs and outputs are declared in a construct called as entity.
8. The input and output signals for the entity are called its ports, and they are identified by the keyword PORT.
9. Signal names can include any letter or number, as well as the underscore character ‘_’.
10. Signal name must begin with a letter, and a signal name cannot be a VHDL keyword.
11. The circuit’s functionality must be specified with a VHDL construct called architecture.
12. VHDL supports Boolean operators: AND, OR, NOT, NAND, NOR, XOR, and XNOR.
13. In VHDL terminology a logic expression is called a simple assignment statement.



A simple logic function.

```
ENTITY example1 IS
    PORT ( x1, x2, x3 : IN BIT ;
          f           : OUT BIT ) ;
END example1 ;
```

VHDL entity declaration for the circuit in Figure

```
ENTITY example2 IS
    PORT ( x1, x2, x3, x4 : IN BIT ;
          f, g           : OUT BIT ) ;
END example2 ;
```

```
ARCHITECTURE LogicFunc OF example2 IS
BEGIN
    f <= (x1 AND x3) OR (x2 AND x4) ;
    g <= (x1 OR NOT x3) AND (NOT x2 OR x4) ;
END LogicFunc ;
```

VHDL code for a four-input function.

Unit - 2

Programmable logic devices (PLD)

1. It is an IC that contains large no. of gates, flipflops e.t.c that can be configured by the user to perform different functions.

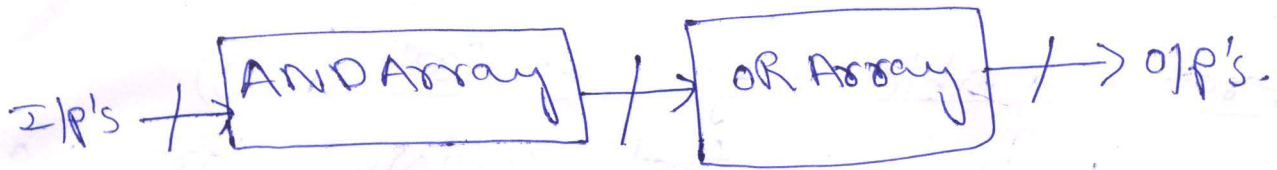
PLD

gates, flipflops
- can be configured
by user to perform a fun

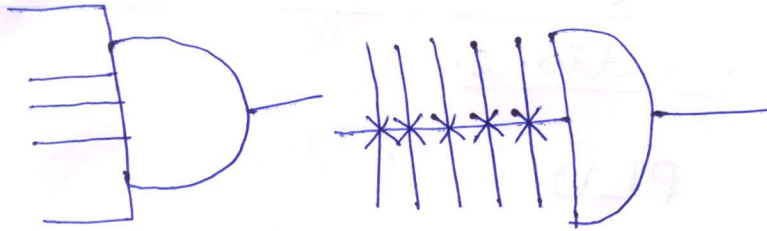
2. It is a combination of array of AND gates (AND-array) & an array of OR gates (OR array).

PLD = AND array + OR array

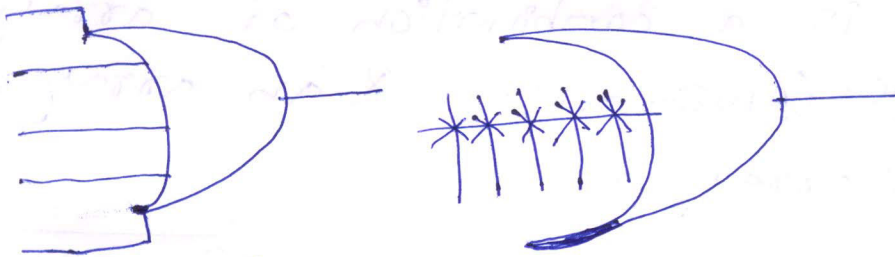
PLD's	standard IC's.
<ol style="list-style-type: none">1. less board space2. more reliable3. less power4. less cost.5. Faster.6. Easy to assemble	<ol style="list-style-type: none">1. differ. cost (in setting, soldering & testing high)



PLD

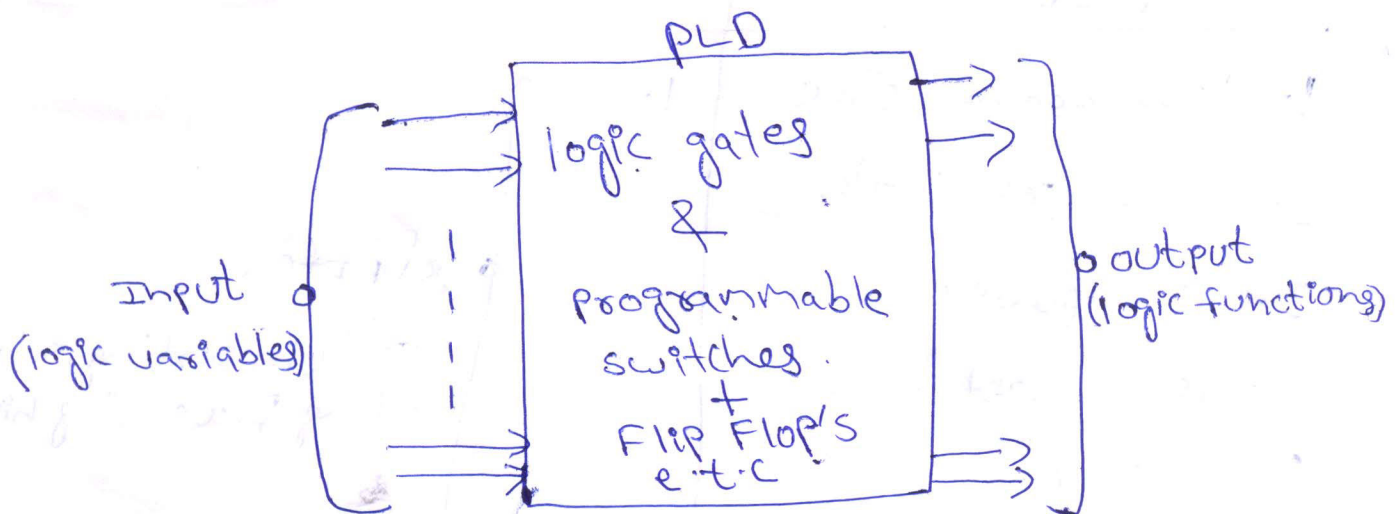


Array logic symbols for multiple I/P AND & OR gate.

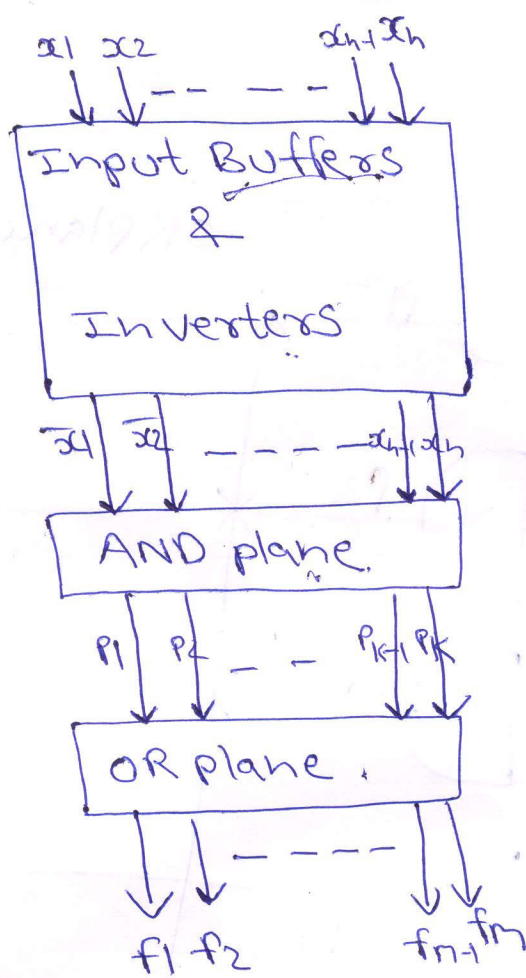


Note

PLD have hundred's to millions of gates.



PLD

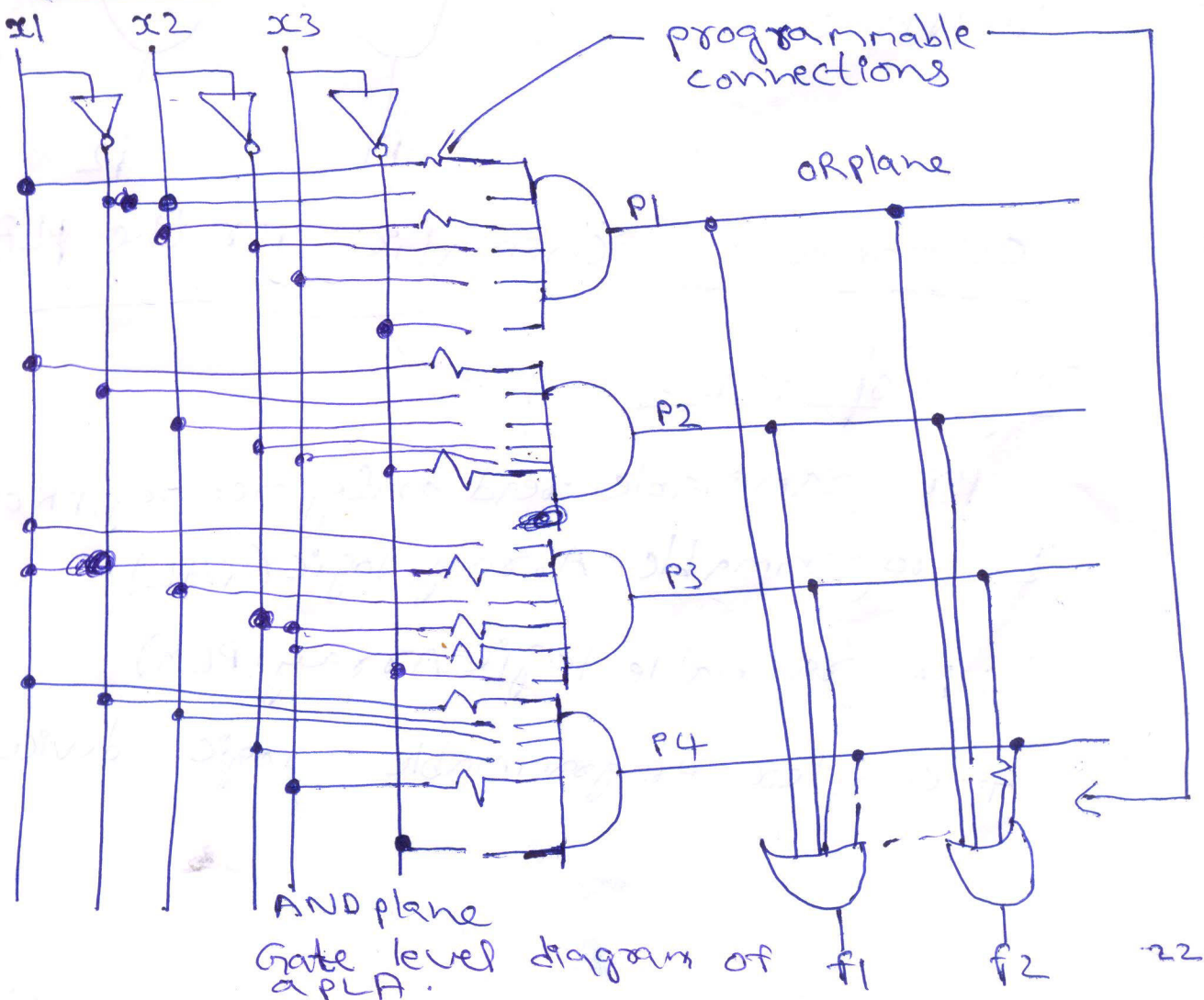


$x_1 \dots x_n = \text{Inputs}$

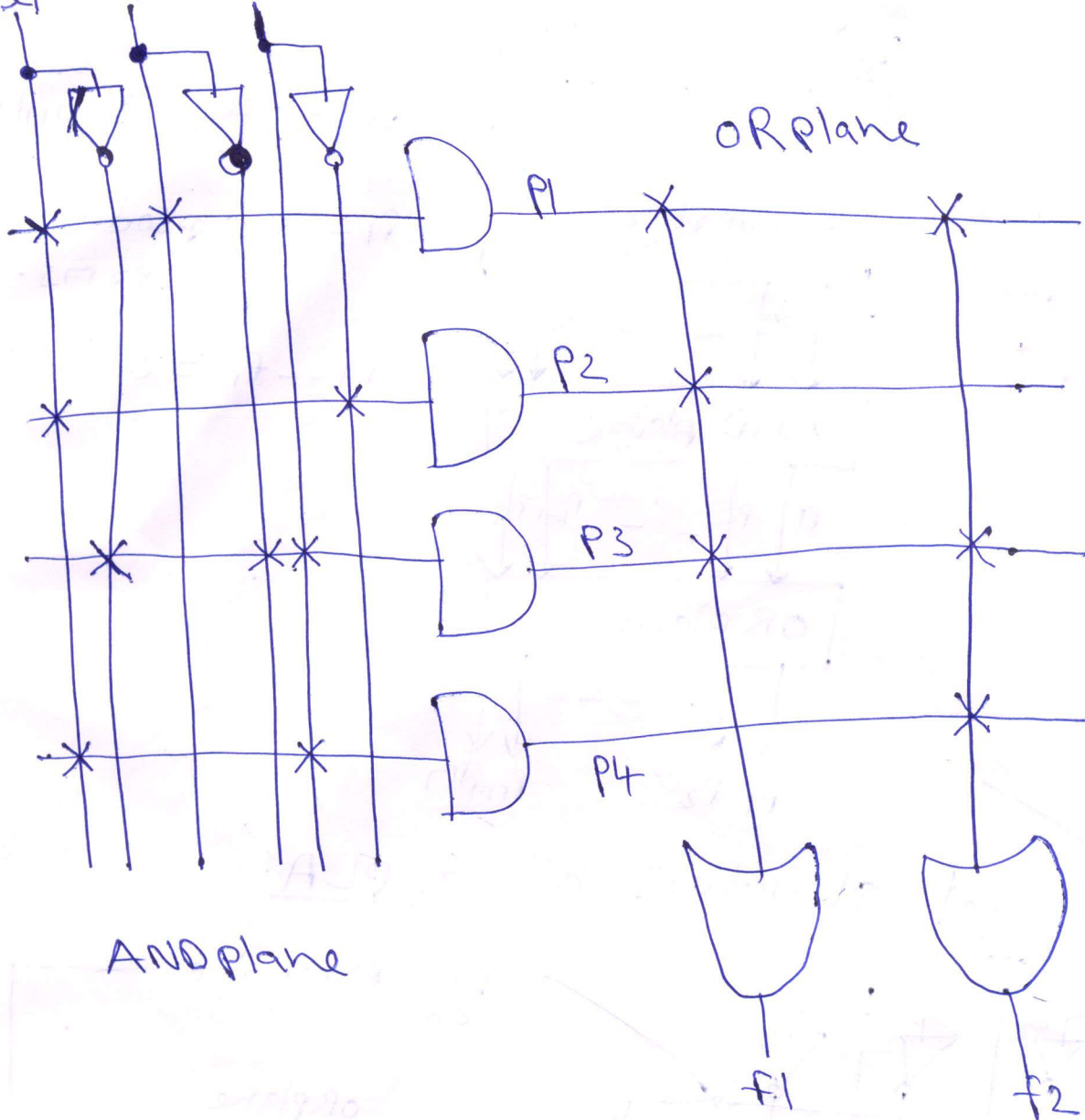
$P_1 \dots P_k = \text{product terms}$

$f_1 \dots f_m = \text{outputs}$

General structure of a PLA:



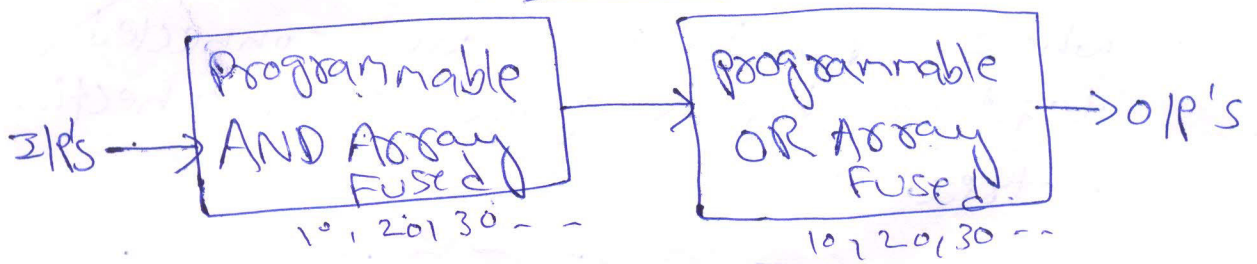
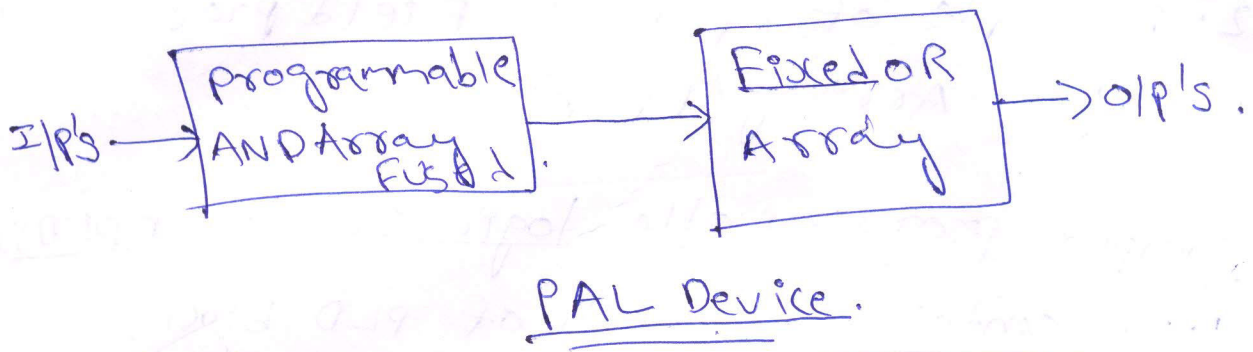
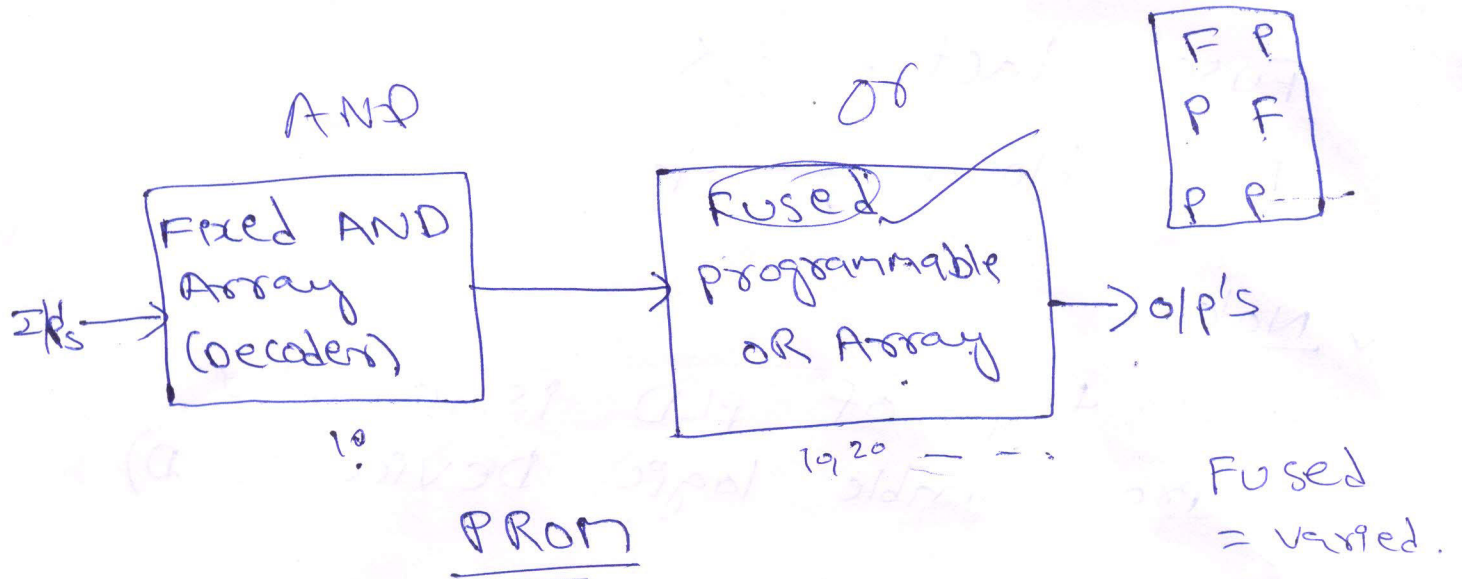
Gate level diagram of a PLA.



customary schematic for the PLA.

Types of PLD's

1. Programmable read only memory (PROM)
2. Programmable Array logic (PAL)
3. Programmable logic Array (PLA)
4. complex programmable logic device (CPLD)

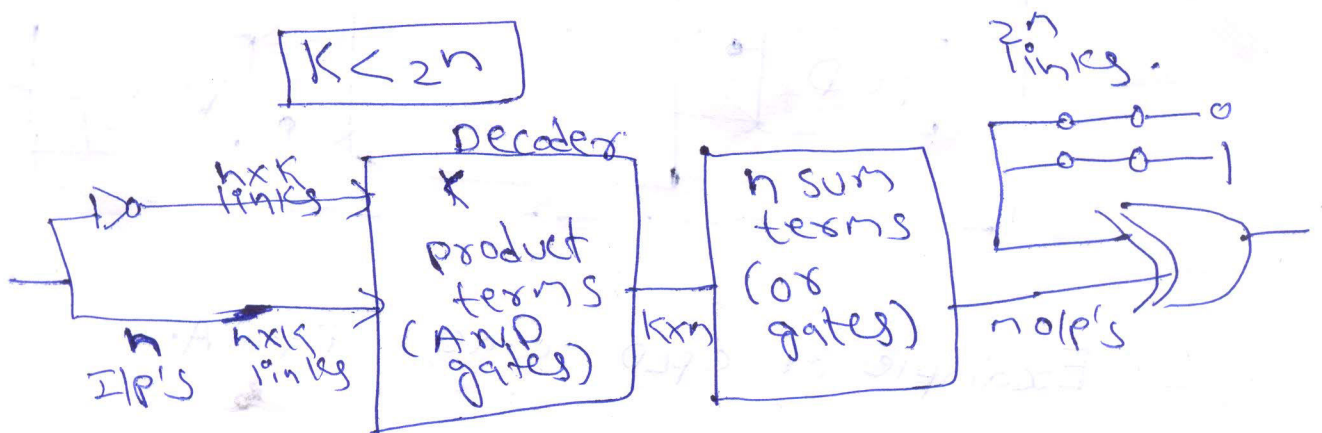


PLA

No of AND gates = K .

Number of I/P = n

$$K < 2^n$$



Fuse intact — X

Fuse blown — +

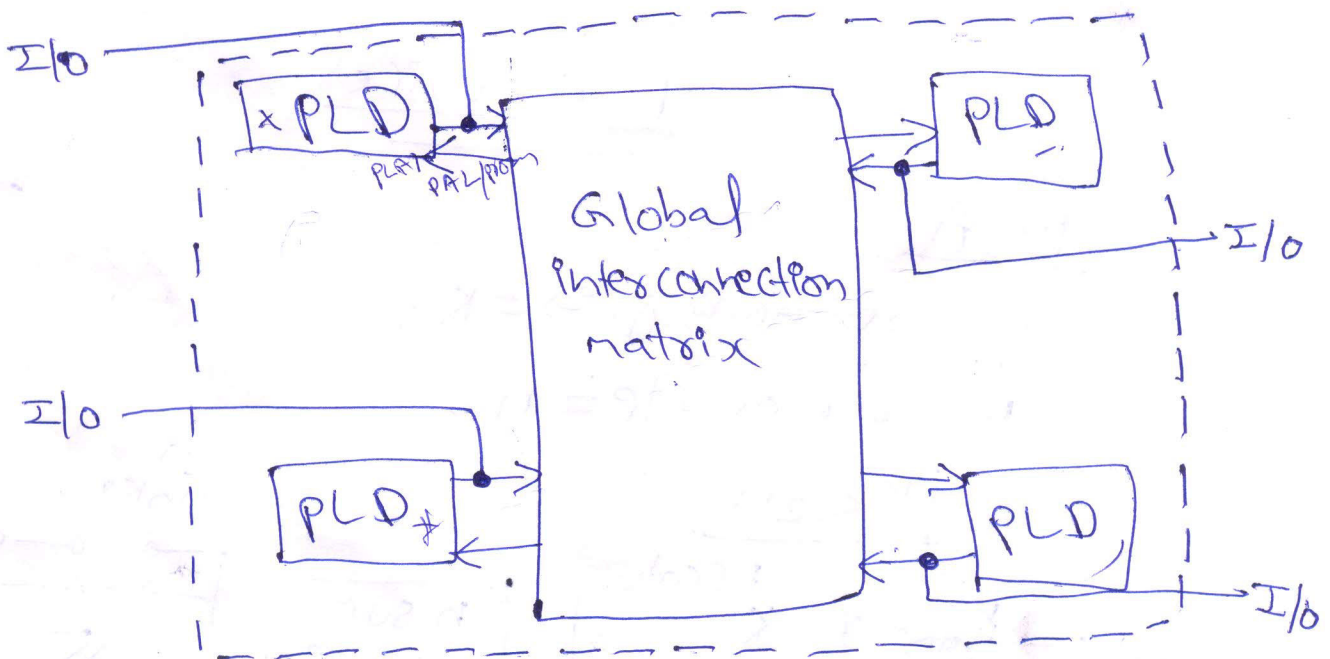
✓ Note

1. 4th type of PLD is complex Programmable logic Device (CPLD)

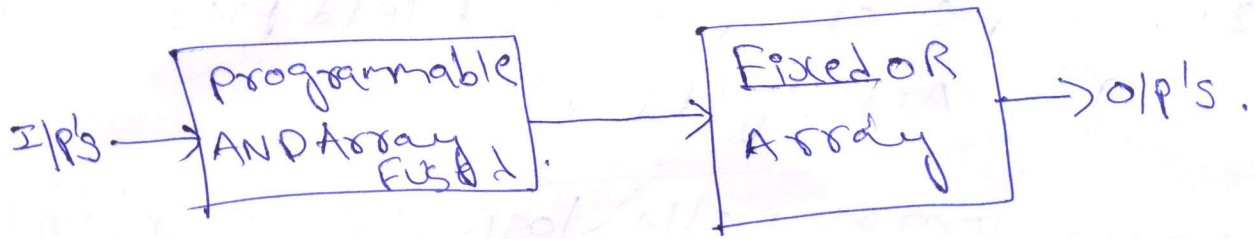
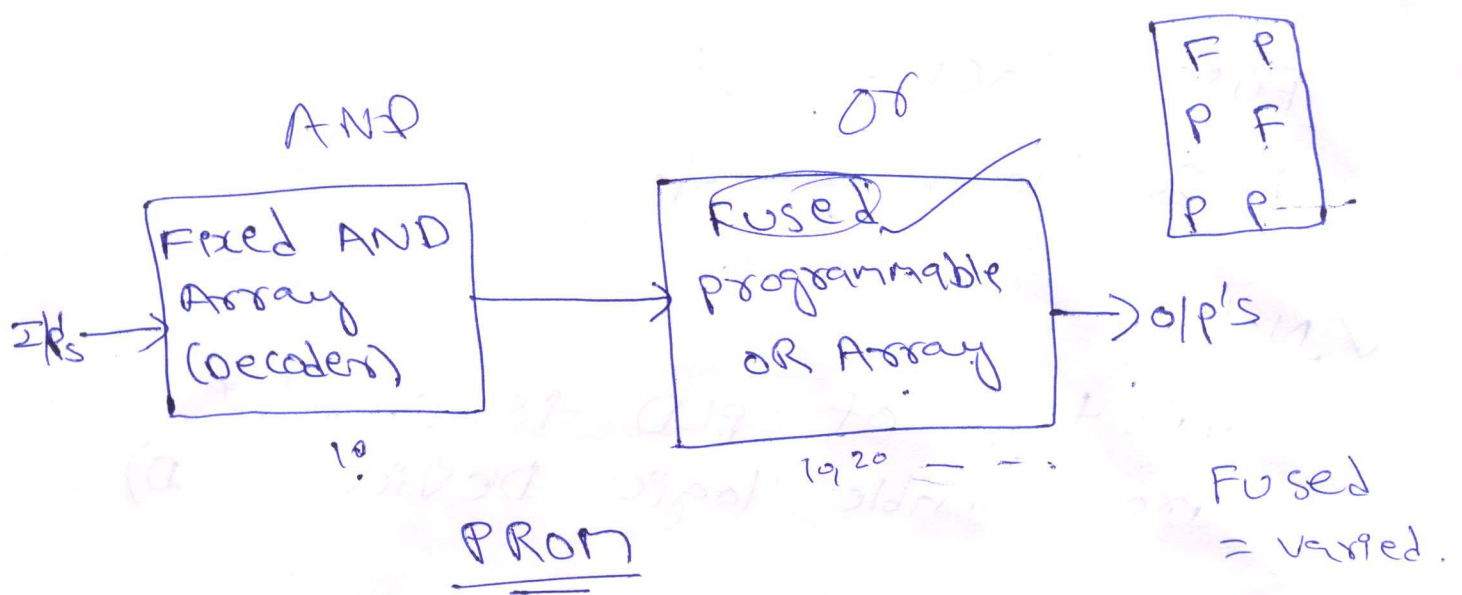
2. Example of CPLD is Field programmable gate Array (FPGA).

Complex programmable logic Devices (CPLD)

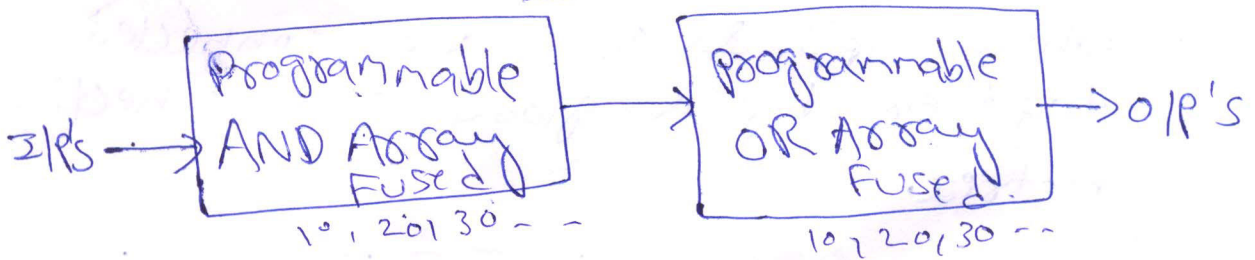
1. It contains a bunch of PLD blocks whose I/O's & O/P's are connected together by a global interconnection matrix.



2. Example of CPLD was FPGA.



PAL Device.



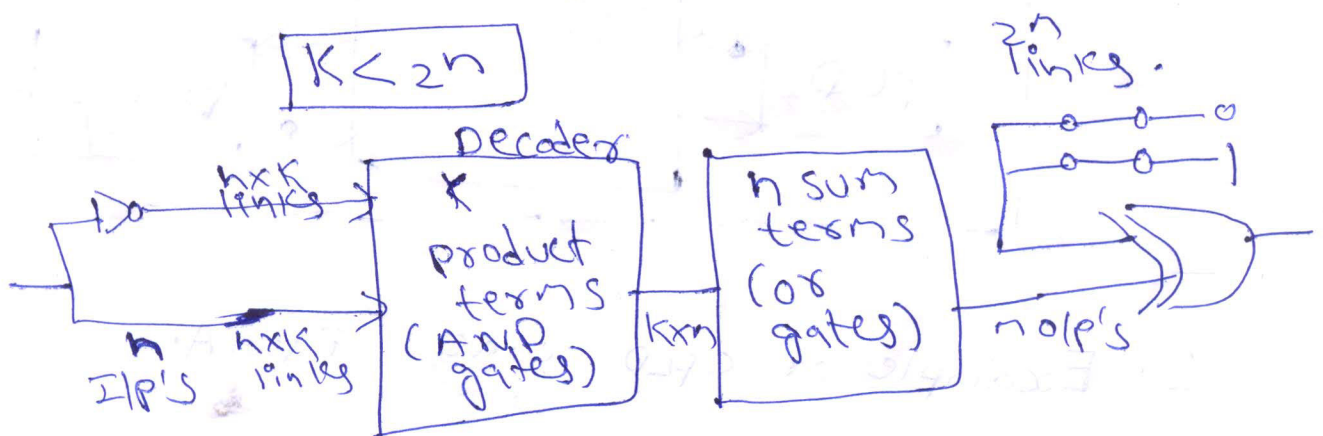
PLA Device

PLA

No of AND gates = K .

Number of I/P = n

$$K < 2^n$$



Field Programmable Gate Array (FPGAs)

It consists of 3 main structures.

1. Programmable logic structure.
2. Programmable routing structure.
3. Programmable Input/output (I/O).

Programmable logic structure

1. It consists of a 2Dimensional array of configurable logic blocks (CLB's).
2. CLB (programmed) used to implement any boolean function. of its I/P variables (4-6 I/P variables).
3. Functions of large no. of variables & implemented using more than one CLB.
4. CLB = 1/2 FF'S allow implementation of sequential logic.

Programmable routing structures

1. used to CLB's.
2. It contain 3 routing resources.
 - a. vertical & horizontal routing channels.
 - i. It consist of differ length wires that can be connected together if needed.
 - ii. These channel run vertically & horizontally b/w columns & rows of CLB as shown in figure.

b. connection boxes

It is a set of programmable links that can connect I/P & O/P pins of the CLB's to wires of the vertical (or) the horizontal routing channels.

c. switch boxes

located at the interconnection of the vertical & horizontal channels.

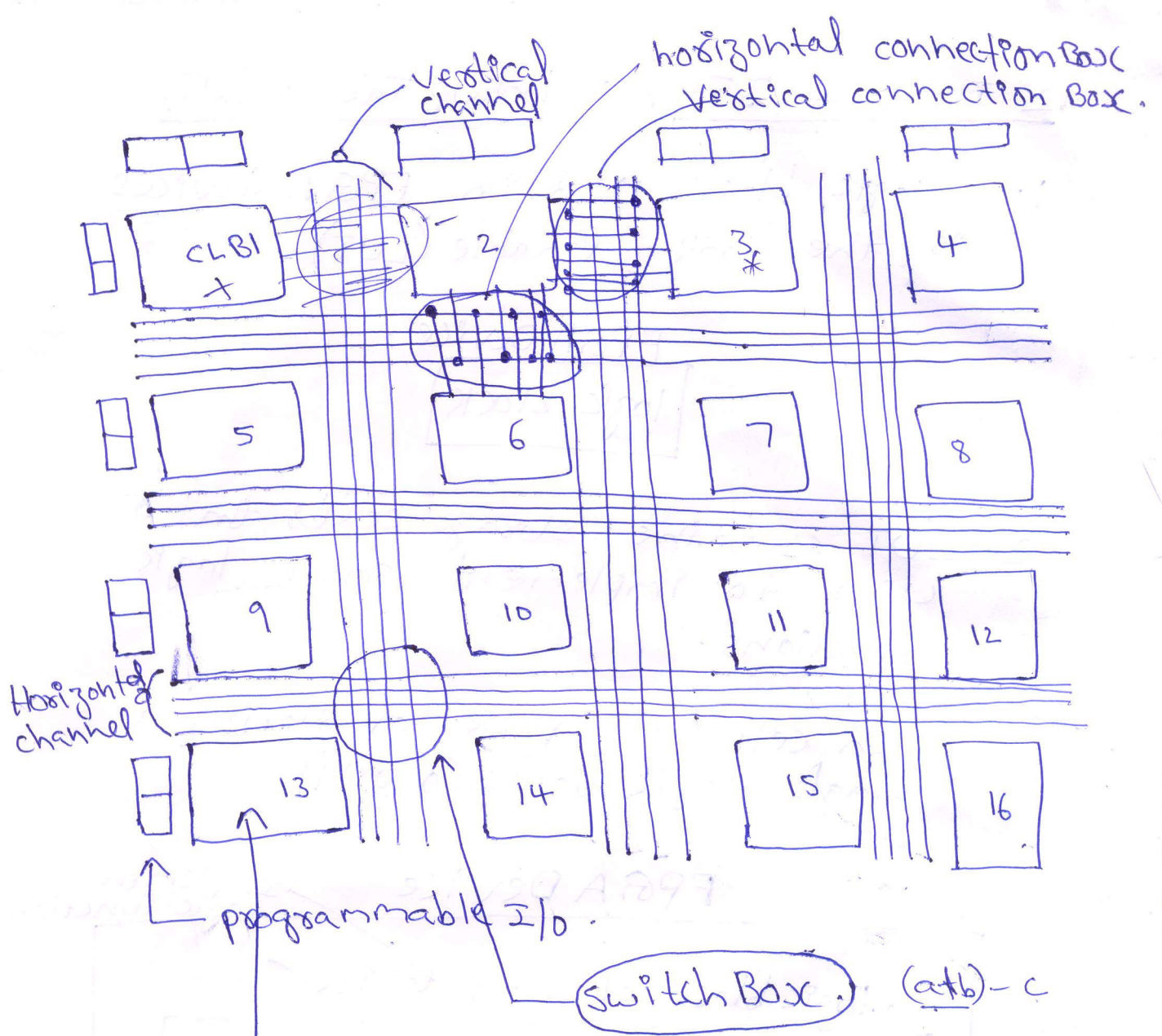
These are a set of programmable links that can connect wire segments in the horizontal & vertical channels.

Programmable I/O

1. These are mainly buffers that can be configured either as I/P buffers, O/P buffers (or) I/P/O/P buffers.

2. They allow the pins of the FPGA chip to function either as I/P pins, O/P pins (or) I/P/O/P pins.

Routing = horizontal, vertical connection box
+
horizontal, vertical channels.
+
switch boxes



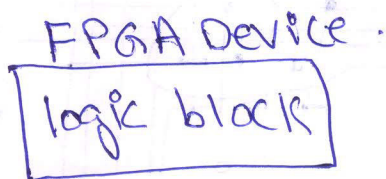
CLB (Implement any boolean function of 4 to 6 I/O variables).

Note CLB = PLA = PLD = PROM.

1. PLD's are IC's that are manufactured using SSI, MSI, LSI, VLSI.
2. MSI (10 - 1000 gates)
3. combinational ckt & MSI form
E.g. adders, subtractors, comparators, decoders, encoders, multiplexers -- e.t.c.

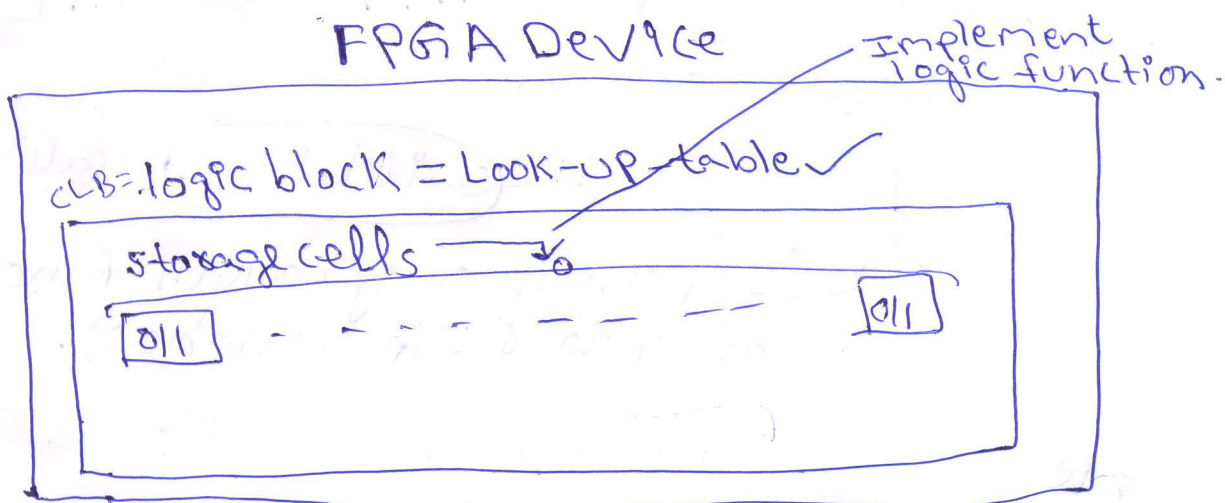
2 Input & 3 Input Look-up Tables (LUT)

1. A logic block used in FPGA devices is the look-up-table (LUT)



2. LUT contains storage cells that are used to implement small logic functions.

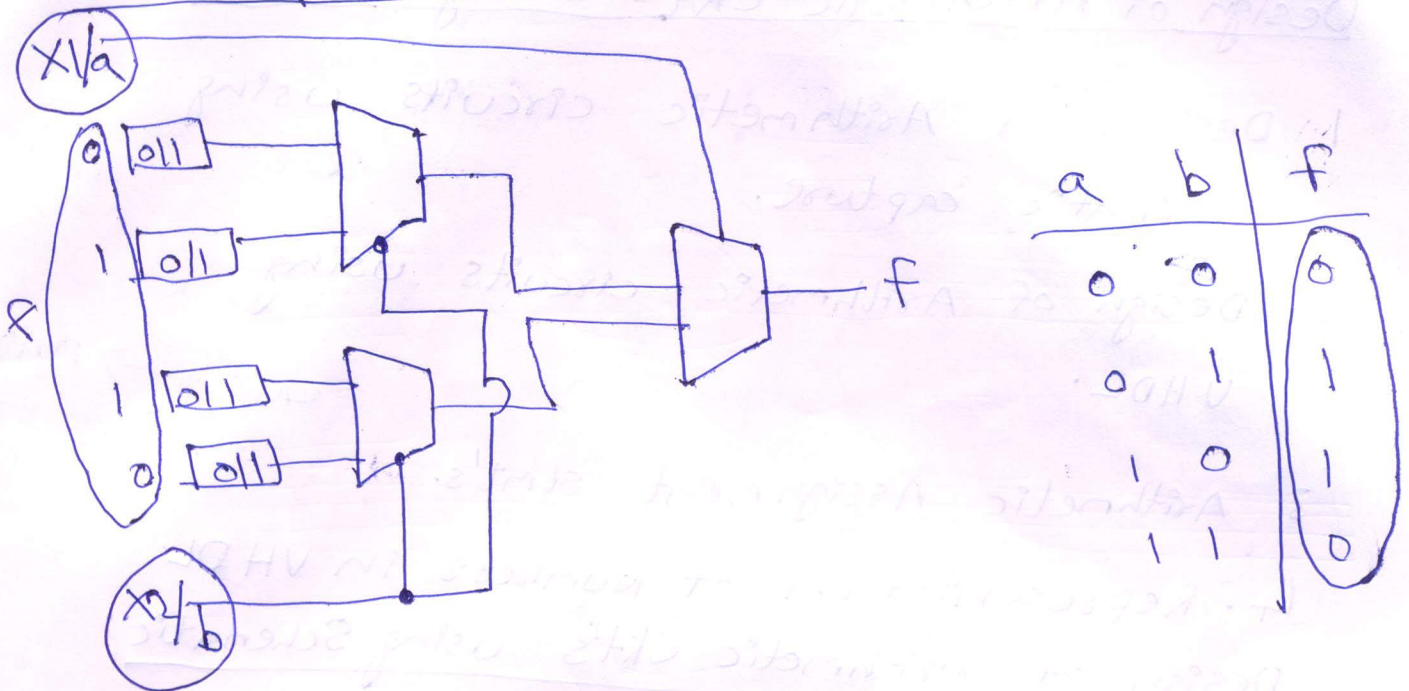
3. Each cell is capable of storing a single logic value (0 or 1)



4. multiplexers are used to select one of the storage cells for output.

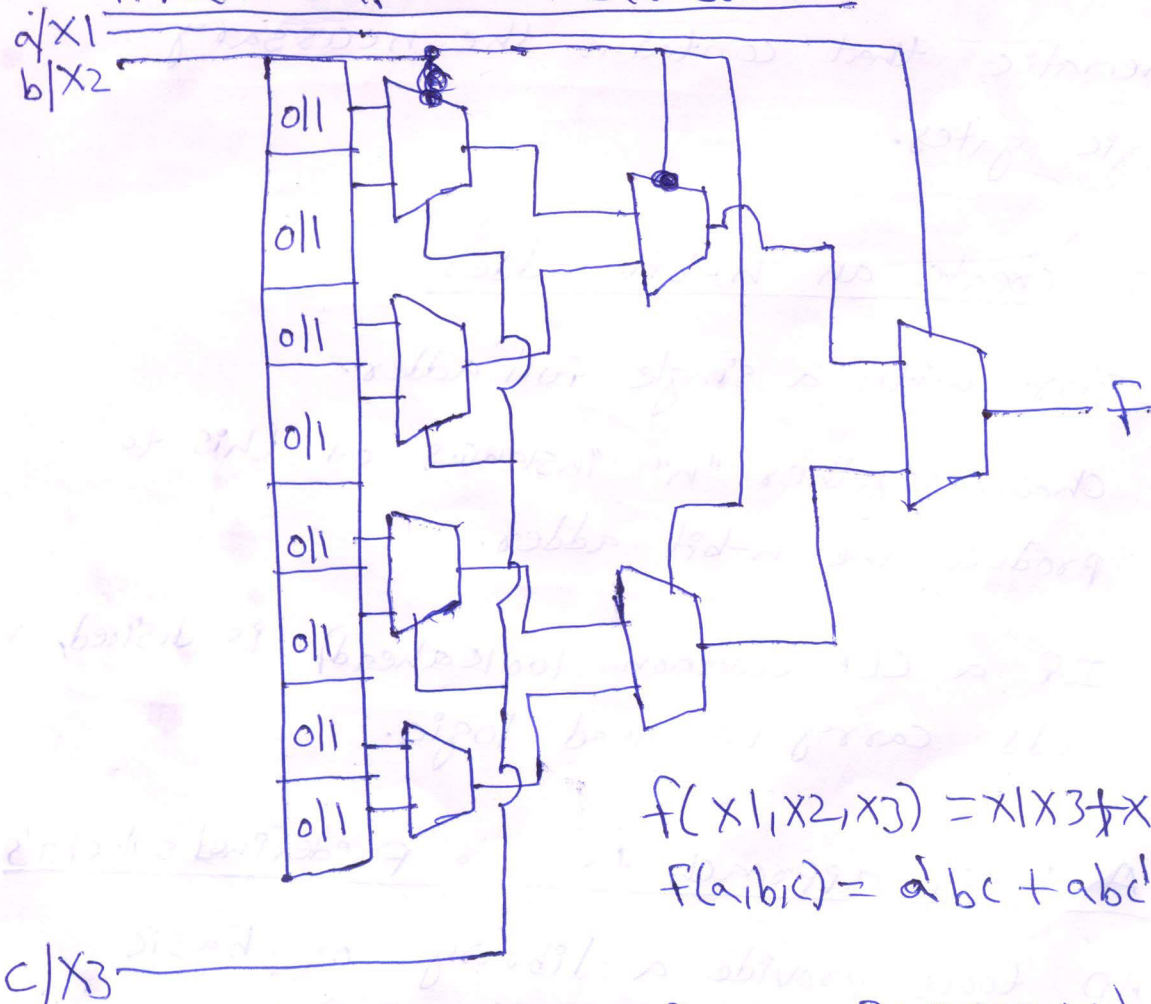
5. Essentially, the cells store the truth table for a function & the multiplexers select a particular cell for output based on a set of select (ctrl) inputs.

Two Input LUT structure



programmed LUT ($F = ab + ab'$)

Three-input LUT structure



show the diagrams for a programmed LUT that implements the function.

Design of Arithmetic ckt's using CAD Tools

1. Design of Arithmetic circuits using schematic capture.

2. Design of Arithmetic circuits using VHDL.

CAD Tool \rightarrow Arith
+1 - - -

3. Arithmetic Assignment stmt's.

4. Representation of numbers in VHDL.

Design of Arithmetic ckt's using Schematic capture

1. Schematic capture is used to draw a schematic that contains the necessary logic gates.

2. To create an n-bit adder

a. Start with a single full adder.

b. Chain together "n" instances of this to produce the n-bit adder.

c. If a CLA (Carry Lookahead) is desired, add carry lookahead logic.

3. A better approach to use predefined subckt's

a. CAD tools provide a library of basic gates.

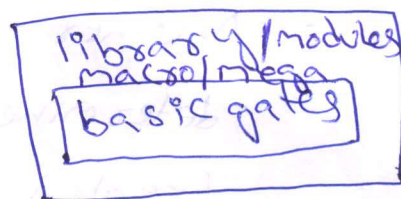
b. most CAD Tools also provide a library of commonly used CKT's, such as address.

c. Each subckt is provided as a module that can be imported into a schematic & used as a part of a larger ckt. In some CAD systems the modules/library functions are called as macro/mega functions.

4. macro Function Types

1. Technology dependent.
2. Technology independent.

CAD Tools



Technology dependent

It is designed to suit a specific type of chip (Ex:-FPGA).

LPM = macro/mega()

↑
TP T2

Technology independent

It is designed to implement in any type of chip, with different CKT's for different types of chips.

5. a. A good example of a library of macrofunctions is the part of the MAX+PLUSII CAD system which is the Library of parameterized modules (LPM).

b. Each module is technology dependent.

c. Each module is parameterized:-

It can be used in a variety of ways.

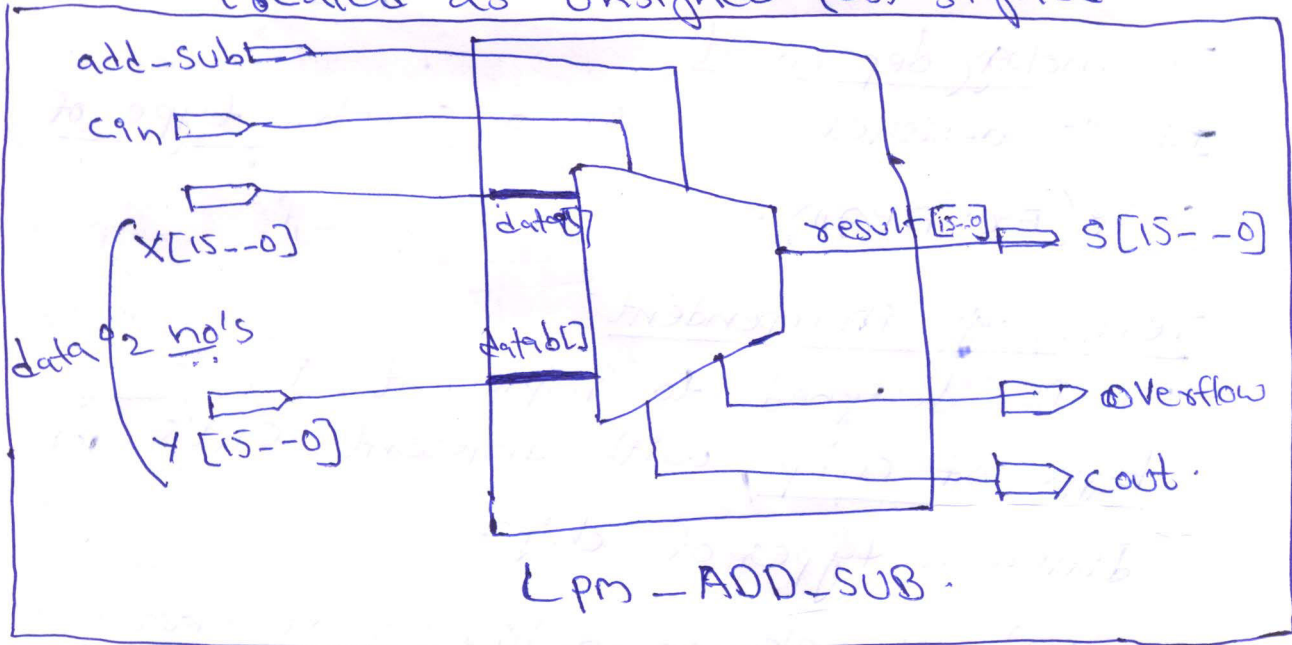
LPM_ADD_SUB

1. The LPM library includes a n-bit adder named LPM_ADD_SUB.

a. implements a basic add/subtract ckt.

b. The number of bit's, n, is set by a parameter LPM_WIDTH.

c. Another parameter LPM_Representation determines whether the numbers are treated as unsigned (0s) signed.



$$X[15--0] = \text{data a}[]$$

$$Y[15--0] = \text{data b}[]$$

$$\text{result}[E] = S[15--0]$$

15--0 = 16 bits It can add.

Design using VHDL

1. we can use a hierarchical approach in designing VHDL code.

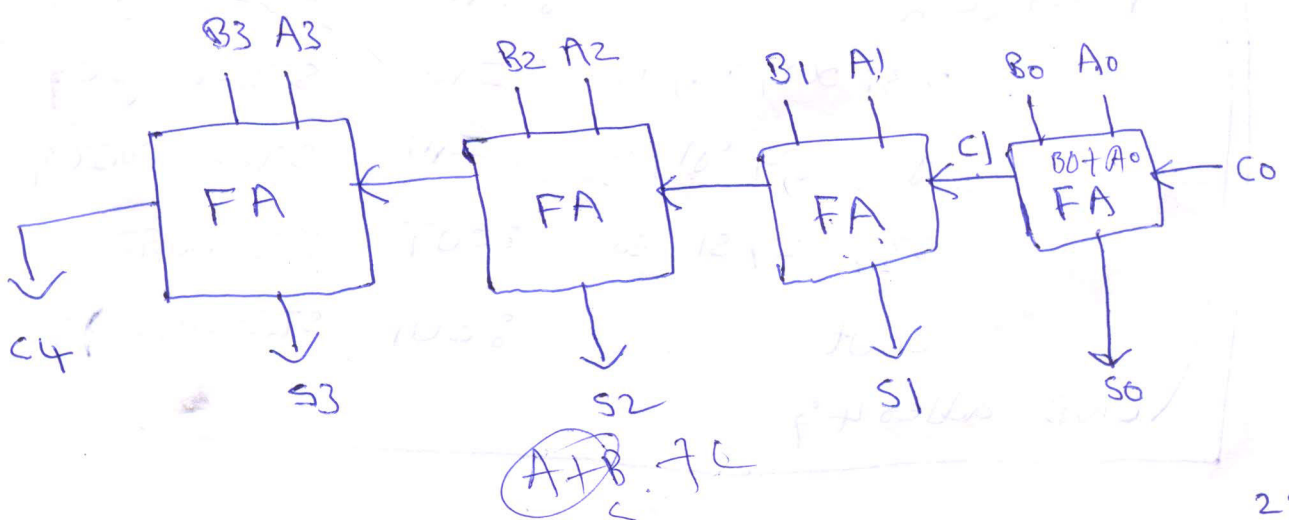
- First construct a VHDL entity for a full adder.
- use multiple instances to create a multi-bit adder.

2. In VHDL, a logic signal is represented as a data object.

- we used a BIT data type before that could only take on the values 0 and 1.
- Another data type, STD-LOGIC is actually preferable because it can assume several different values [0, 1, Z (high impedance), don't care].

3. we must declare the library where the data type exist's, & declare that we will use the data type.

```
LIBRARY ieee;  
USE ieee.std-logic-1164.all;
```



VHDL Full adder

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY fulladd IS
PORT (cin, x1, y1 : IN STD_LOGIC;
      s, cout : OUT STD_LOGIC);
END fulladd.

ARCHITECTURE logicfunc OF fulladd IS
BEGIN
  s <= XOR OR cin;
  cout <= (x AND y) OR (cin AND x) OR (cin AND y);
END logicfunc;
```

VHDL 4-bit ripple carry adder

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY adder4 IS
PORT (cin : IN STD_LOGIC;
      x3, x2, x1, x0 : IN STD_LOGIC;
      y3, y2, y1, y0 : IN STD_LOGIC;
      s3, s2, s1, s0 : OUT STD_LOGIC;
      cout : OUT STD_LOGIC);
END adder4;
```

ARCHITECTURE structure of adder4 IS

```
SIGNAL c1, c2, c3 : STD_LOGIC;
```

A	1010
B	0010
<hr/>	
C1	00

```
COMPONENT fulladd
```

```
  PORT (cin, x1, y1 : IN STD_LOGIC;
```

```
        s, cout    : OUT STD_LOGIC);
```

```
END COMPONENT
```

BE BEGIN

```
stage 0 : fulladd PORT MAP (cin, x0, y0, s0, c1);
```

```
stage 1 : fulladd PORT MAP (c1, x1, y1, s1, c2);
```

```
stage 2 : fulladd PORT MAP (c2, x2, y2, s2, c3);
```

```
stage 3 : fulladd PORT MAP
```

```
(cin => c3, cout => cout, x => x3, y => y3, s => s3);
```

```
END structure
```

Note

1. signal defines that it will be used internal to the design.
2. COMPONENT fulladd defines the PORT for a subckt (component) i.e. defined in another file (fulladd.vhd in this example).
3. The VHDL file (fulladd.vhd) should normally be in the same directory as the file adder4.vhd.

Stage 0 : fulladd PORT MAP (cin, x0, y0, s0, c1) ;

1. Defines an instance of the component fulladd named stage0.
2. Uses positional association because the I/O's & o/p's listed in the PORT MAP appear in the exact same order as in the component statement.

Stage 3 : fulladd PORT MAP (
cin => c3, cout => cout, x => x3, y => y3, s => s3) ;

1. Defines an instance of the component fulladd named stage3.
2. Uses named association because each I/O & o/p listed in the PORT MAP is associated with a specific named signal in the component stmt.

VHDL Package

It can be created for a component (subckt) such that the component stmt is not explicitly required when creating instances of the component in another file.

VHDL packages

```

LIBRARY ieee;
use ieee.std-logic-1164.all;

PACKAGE fulladd-package IS
  COMPONENT fulladd
    PORT (cin, x1, y1 : IN STD-LOGIC;
          s, cout : OUT STD-LOGIC);
  END COMPONENT;
END fulladd-package;
  
```

Note

usually compiled as a separate file in the same directory as fulladd.vhd but can be located in the source file after the ARCHITECTURE construct.

```

LIBRARY ieee;
use ieee.std-logic-1164.all;
use work.fulladd-package.all;

ENTITY adder4 IS
  PORT (cin : IN STD-LOGIC;
        x3, x2, x1, x0 : IN STD-LOGIC;
        y3, y2, y1, y0 : IN STD-LOGIC;
        s3, s2, s1, s0 : OUT STD-LOGIC;
        cout : OUT STD-LOGIC);
END adder4;

ARCHITECTURE
  SIGNAL
    stage 0 : fulladd PORT MAP (cin, x0, y0, s0, c1);
    1 : fulladd PORT MAP (c1, x1, y1, s1, c2);
    2 : fulladd PORT MAP (c2, x2, y2, s2, c3);
    3 (cin => c3, cout => cout, x => x3, y => y3, s => s3);
  END STRUCTURE
  
```


Number's in VHDL

1. A no in VHDL is a multibit signal data object.

```
SIGNAL C: STD_LOGIC_VECTOR (D TO 3);
```

2. C is a 3-bit STD-logic signal.

C - a 3-bit quantity.

c(1) - a 1-bit quantity (the MSB);

c(2) - a 1-bit quantity;

c(3) - a 1-bit quantity (the LSB);

C = 4
0123
0100
3 210

3. The ordering of the bits can be reversed.

```
SIGNAL X: STD_LOGIC_VECTOR (3 DOWN TO 0);
```

4. X is a 4-bit STD-logical signal.

5. → x(3) is the MSB.

→ x(0) is the least significant bit

```
LIBRARY ieee
```

```
USE ieee.std_logic_1164.all;
```

```
USE work.fulladd_package.all;
```

```
ENTITY adder4 IS
```

```
PORT (cin : IN STD_LOGIC;
```

```
      x, y : IN STD_LOGIC_VECTOR(3 DOWN TO 0);
```

```
      s : IN STD_LOGIC_VECTOR(3 DOWN TO 0);
```

```
      cout : OUT STD_LOGIC);
```

```
END adder4;
```

ARCHITECTURE structure of adder4 IS

```
SIGNAL C: STD_LOGIC_VECTOR (1 TO 3);
```

Combinational circuit building blocks

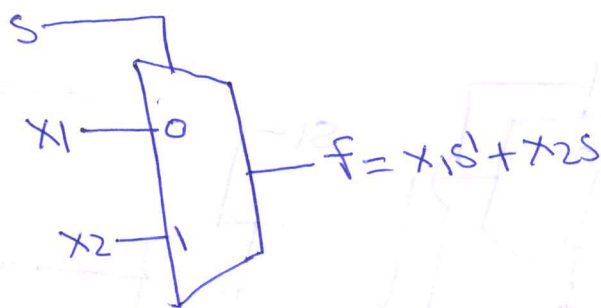
1. Multiplexers.
2. Decoders.
3. Encoders.
4. Code converters.
5. Arithmetic comparison ckt's.

Combinational ckt



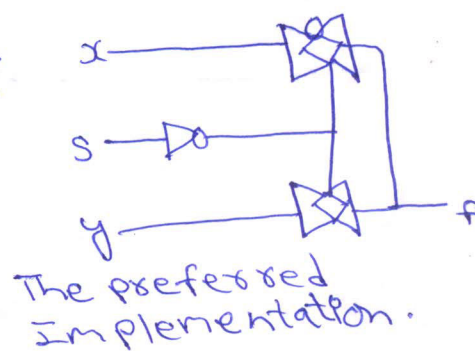
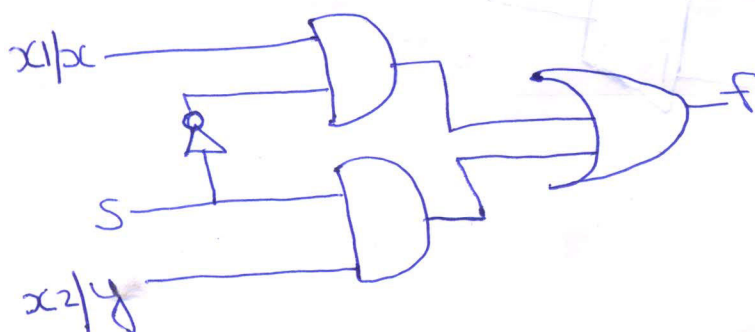
Multiplexers / mux / Data selector

1. It is a ckt consist's of
 - a. no of data Input's
 - b. 1 more select Input's (select lines)
 - c. one output.
2. It passes the signal value on one of its data Input's to its output based on the value(s) of the select signal(s).



S	f(s, x1, x2)
0	x1
1	x2

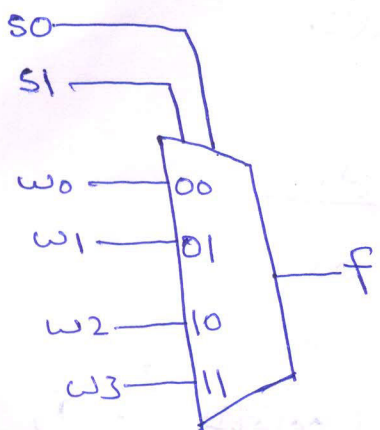
2x1 Multiplexer Implementations



The preferred Implementation.

4X1 Input Multiplexer

1. It selects one of the 4 data inputs to be output based on the values of 2 select lines.

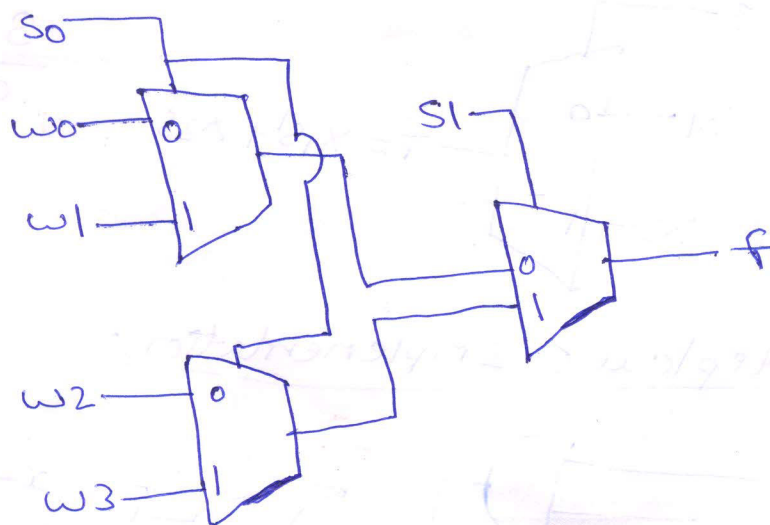


S1	S0	F
0	0	W0
0	1	W1
1	0	W2
1	1	W3

$$F = S_1' S_0' W_0 + S_1' S_0 W_1 + S_1 S_0' W_2 + S_1 S_0 W_3$$

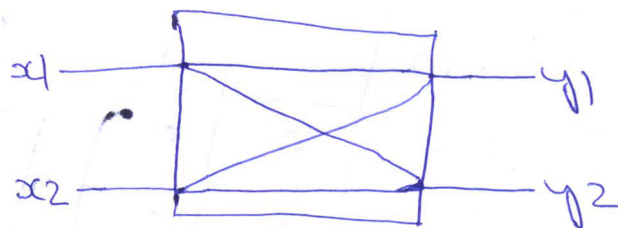
Building a 4 Input mux

1. It can be constructed using 2 Input multiplexer's.

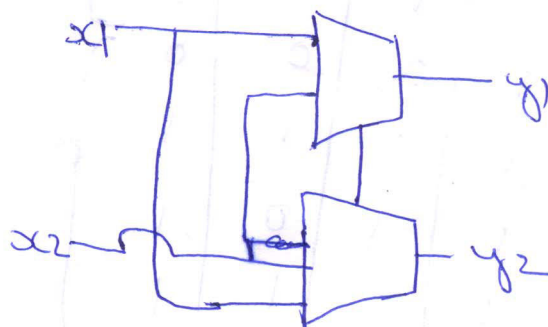


mux application (a 2x2 crossbar)

1. A ckt with "n" inputs & k outputs whose function is to provide a capability to connect any input to any output is called a n x k cross bar switch.
2. with 2 inputs & 2 outputs it is called 2 x 2 crossbar.
3. Useful in applications where it is necessary to connect one set of wires to another set of wires, to another where the connection pattern changes from time to time.
4. Telephone switching NWs are an example.



2x2 crossbar switch.



Shannon's expansion Theorem

1. Any boolean function $f(w_1, \dots, w_n)$ can be written in the form

$$f(w_1, \dots, w_n) = w_1' \cdot f(0, w_2, \dots, w_n) + w_1 \cdot f(1, w_2, \dots, w_n)$$

2. The expansion can be done using any of the "n" variables.

3. If $f(w_1, w_2, w_3) = w_1 w_2 + w_1 w_3 + w_2 w_3$

Expanding this in terms of w_1 gives

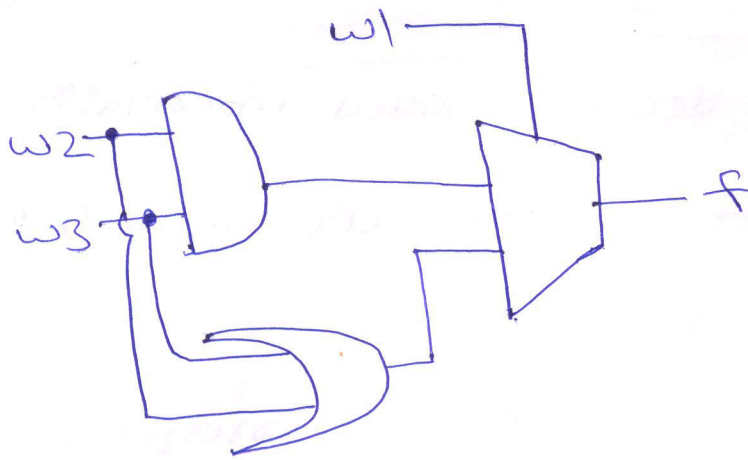
$$f(w_1, w_2, w_3) = \underbrace{w_1(w_2 + w_3)}_{f \text{ when } w_1=1} + \underbrace{w_1'(w_2 w_3)}_{f \text{ when } w_1=0}$$

f when $w_1=1$ f when $w_1=0$

Shannon's expansion example

w_1	w_2	w_3	f	w_1	f
0	0	0	0	0	$w_2 w_3$
0	0	1	0	0	$w_2 w_3$
0	1	0	0	0	$w_2 w_3$
0	1	1	1	0	$w_2 w_3$
1	0	0	0	1	$w_2 + w_3$
1	0	1	1	1	$w_2 + w_3$
1	1	0	1	1	$w_2 + w_3$
1	1	1	1	1	$w_2 + w_3$

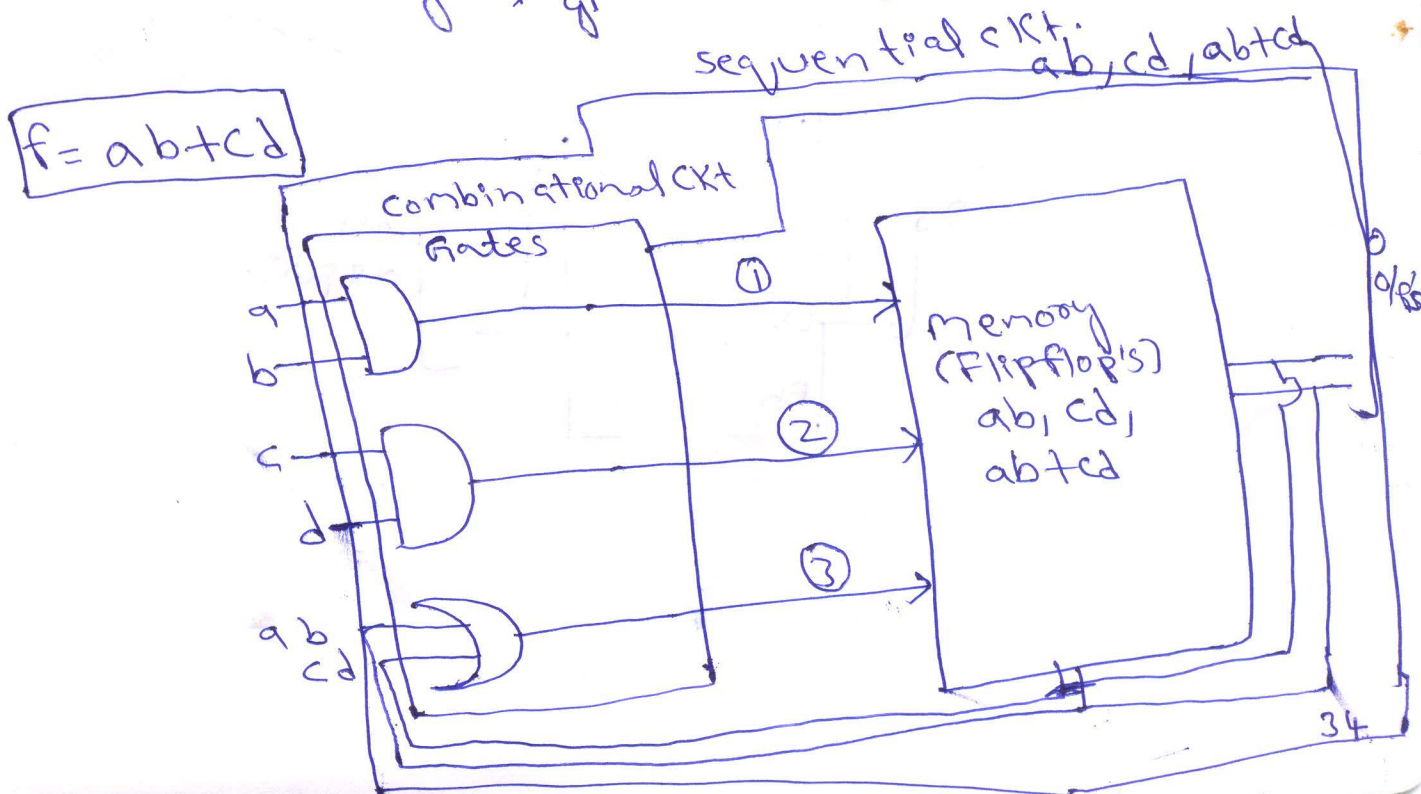
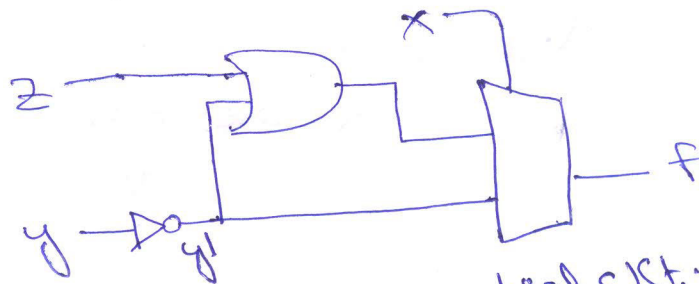
The diagram shows a circuit implementation of the expansion. A vertical line represents the variable w_1 . Two gates are connected to this line: an AND gate with inputs w_1 and $w_2 w_3$, and an OR gate with inputs w_1 and $w_2 + w_3$. The outputs of these two gates are connected to a single OR gate, which produces the final output f . The truth table columns are aligned with the circuit components.



2. $f = x'y'z' + x'y'z + x'yz + xy'z' + xy'z$
 choose "x" as expansion variable.

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

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



Decoders

1. It is used to decode encoded information.

2. A binary decoder is logic ckt with " n " inputs and 2^n outputs.

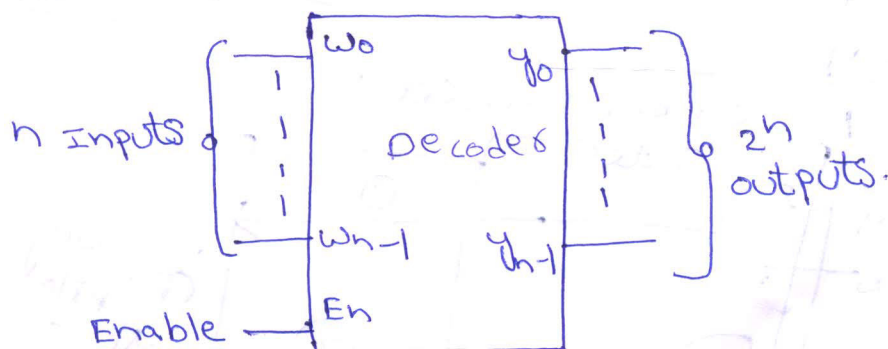
3. Each time some outputs come corresponds to one valuation of the inputs.

i.e. only one output is asserted at any time (one-hot-encoded) & each output corresponds to one valuation of the inputs.

4. An enable input (E_n) is used to disable the o/p's.

a. If $E_n = 0$, none of the decoder outputs is asserted.

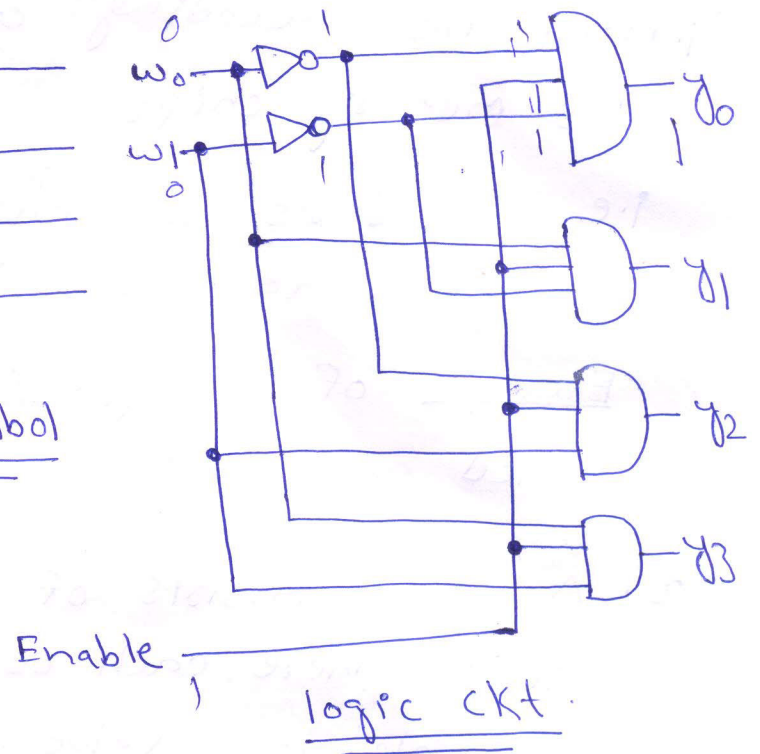
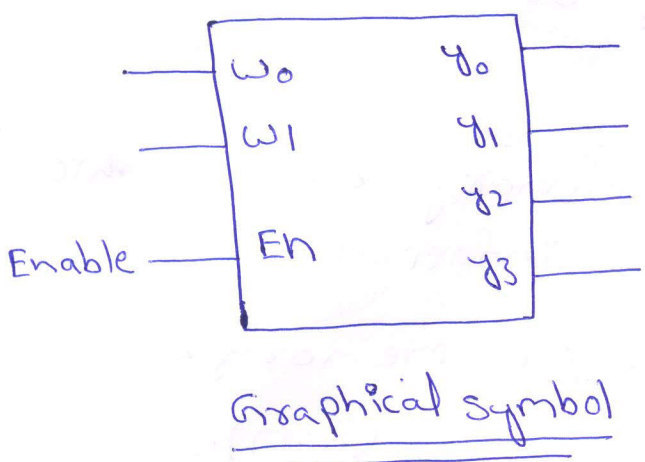
b. If $E_n = 1$, one of the output is asserted according according to the valuation of the input's.



$$E_n = 1$$

$$E_n = 0$$

2 to 4 decoder circuit

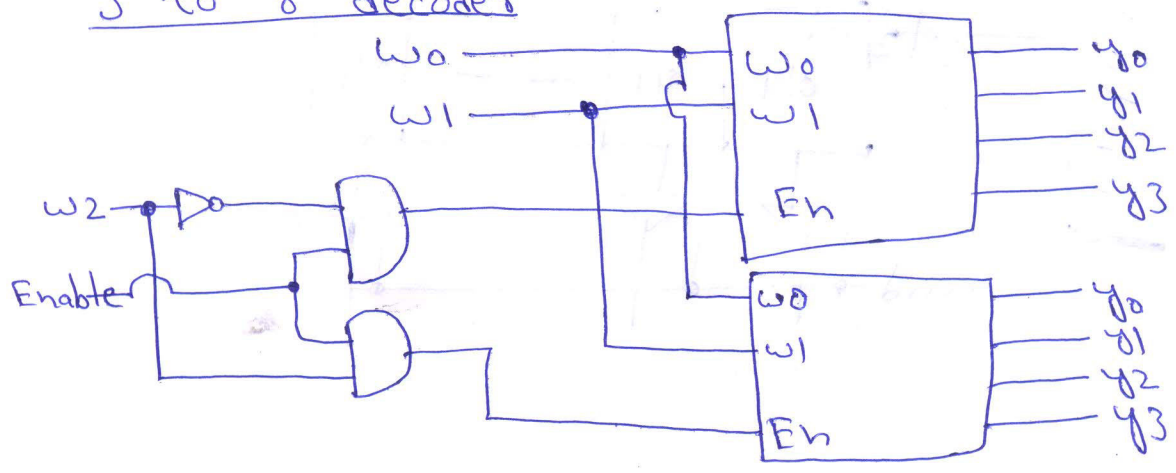


Truth Table

En	w1	w0	y0	y1	y2	y3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	X	X	0	0	0	0

$w_0 = 0$
 $w_1 = 0$
 $En = 1$
 $y_0 = 1$

3 to 8 decoder



Decoder application

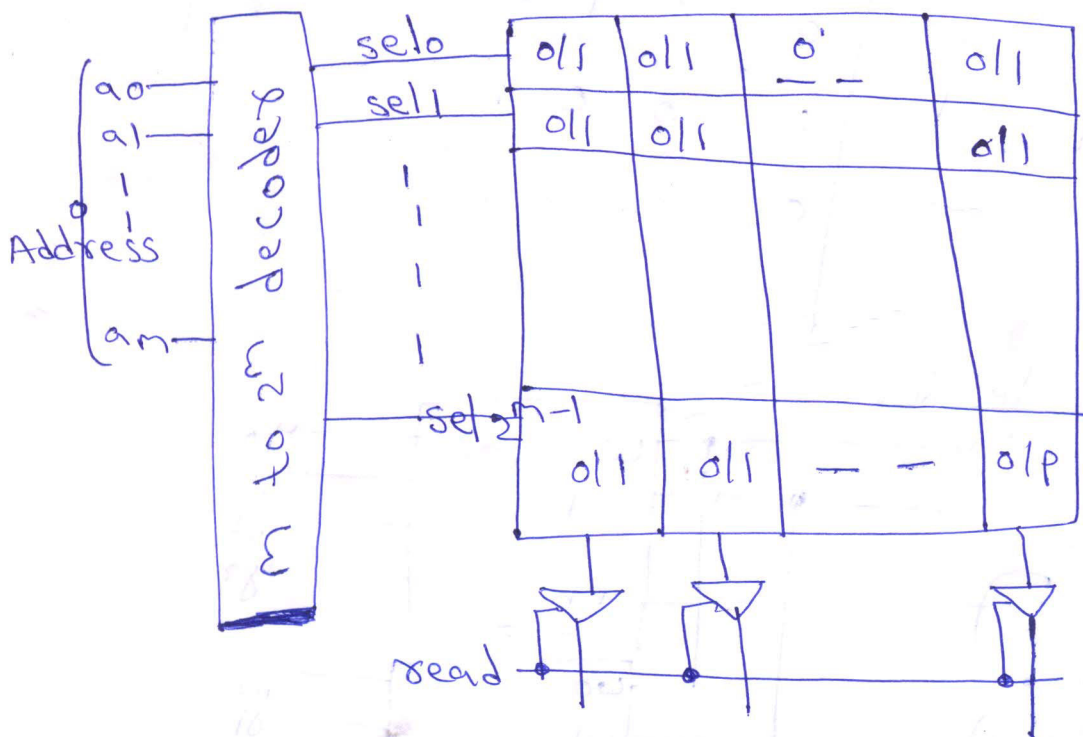
1. Used for decoding of address lines for memory chips.

i.e. Decoder's in memory blocks, which are used to store information.

Example of a type of memory block is called a read-only memory (ROM).

2. A ROM consists of a collection of storage cells, where each cell permanently stores a single logic value, either 0 (or) 1.

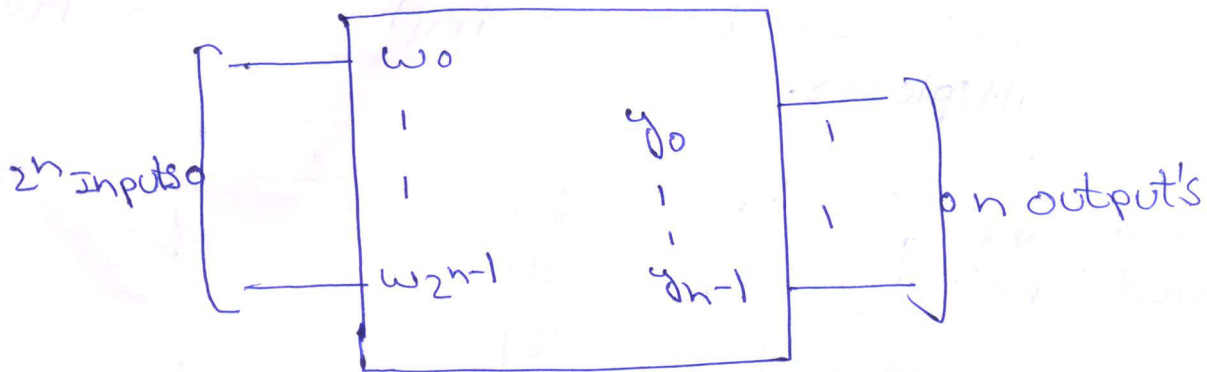
A $2^m \times n$ read only memory block



4. Encoders reduce the number of bits needed to represent given information.

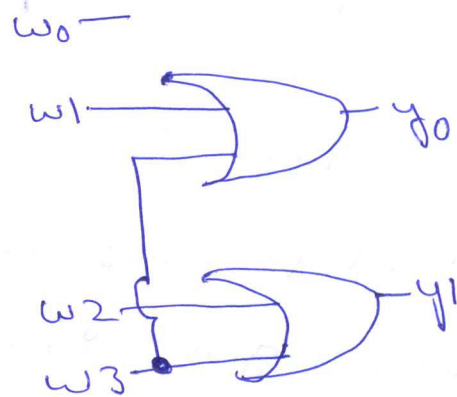
5. practical use: Transmitting info in a digital system.

2^n to n binary encoder



w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Truth Table



CKT

Priority Encoders

1. Another useful class of encoders is based on the priority of the input signals.
2. In a priority encoder, each input has a priority level associated with it.
3. The encoder output's indicate the active input that has the highest priority.
 - a. when an input with a high priority is asserted, the other lower priority input's are ignored.
 - b. Assume that w_0 has the lowest priority & w_3 has the highest.
 - c. The output z indicates when none of the input's are 1.

4. letting

$$i_0 = w_3 w_2 w_1 w_0$$

$$i_1 = w_3 w_2 w_1$$

$$i_2 = w_3 w_2$$

$$i_3 = w_3$$

$w_3 \ w_2 \ w_1 \ w_0$				z 1's don't leave.		
w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

$$y_0 = i_1 + i_3 \quad y_1 = i_2 + i_3$$

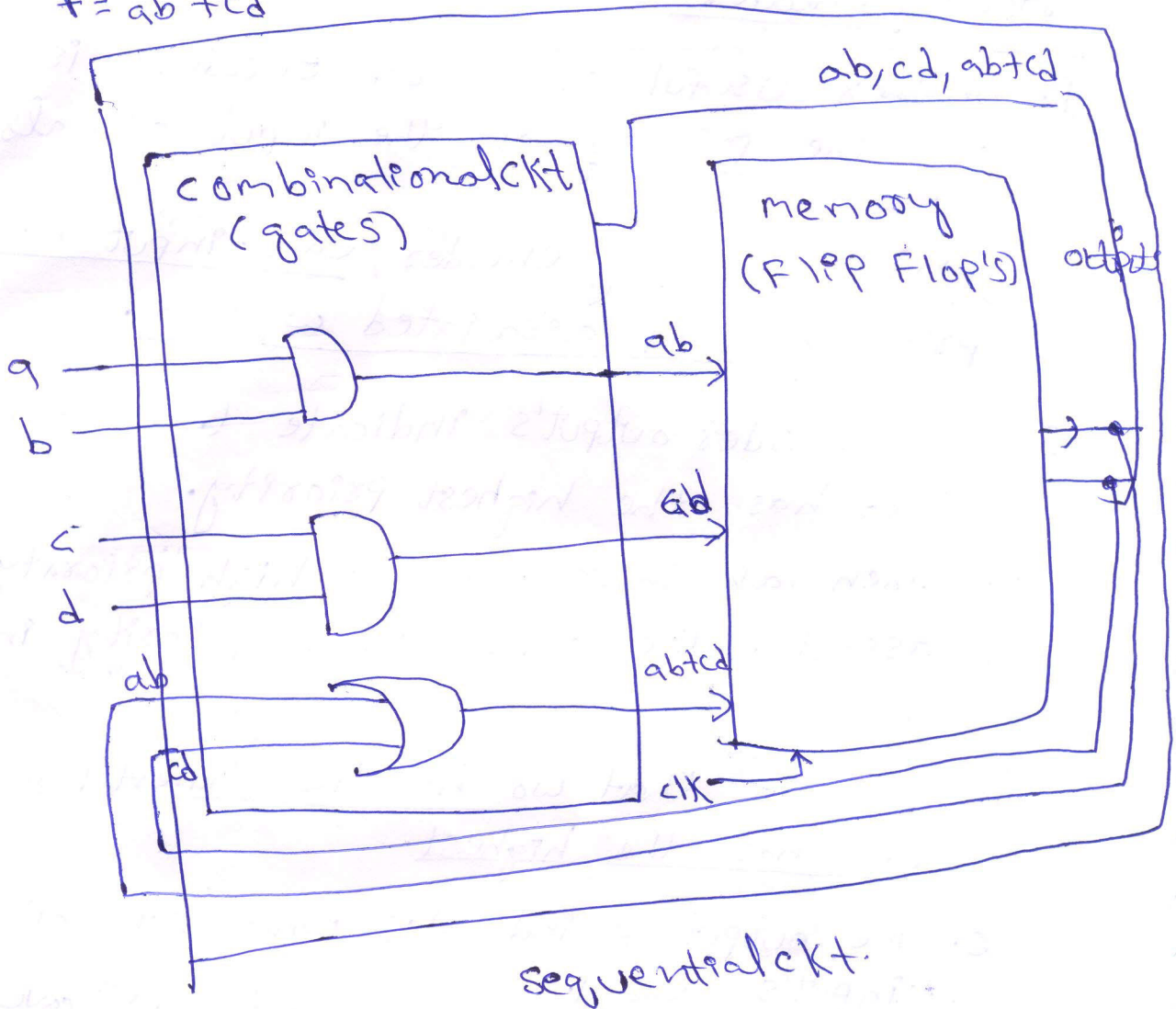
$$z = i_1 + i_2 + i_3 + i_4$$

4 to 2 priority encoder Truth table

$$2^4 = 16$$

sequential ckt

$$f = ab + cd$$



sequential ckt(sc)

synchronous sc
(clk)

Asynchronous sc
(no clk).

code converters

1. It is used to convert from one type of input encoding to another type of output encoding.

Example

a. A 3-to-8 decoder converts from a binary number to a one-hot encoding at the output.

b. A 8 to 3 encoder performs the opposite.

2. many different types of code converter ckt's can be constructed.

a. one common example a BCD-to-7 segment decoder.

3. BCD-to-7 segment decoder

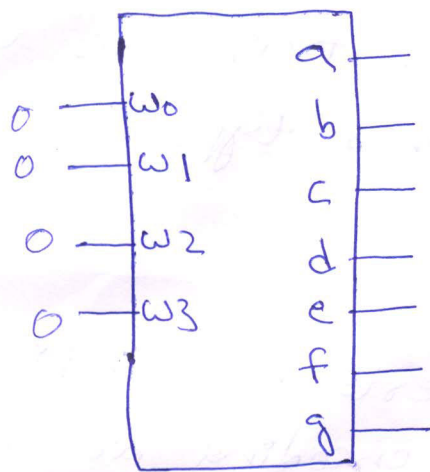
a. converts one binary-coded decimal (BCD) digit into info suitable for driving a digit-oriented display.

i. A vending machine display is an example.

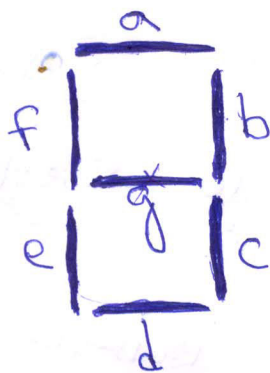
b. The circuit convert's a BCD digit's into 7
-signals that are used to drive (activate)
the segments in the display.

i. Each segment is a small LED, which glows when driven by an electrical signal.

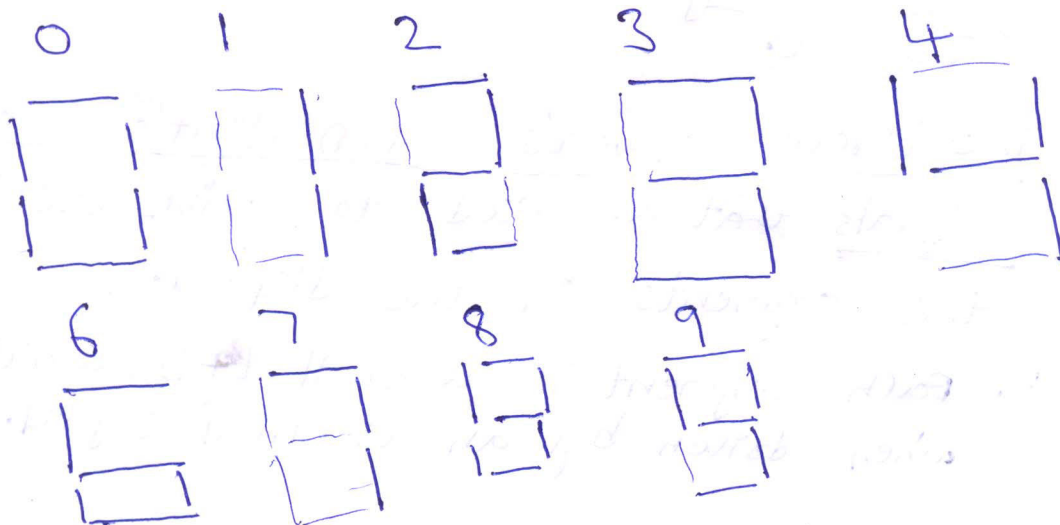
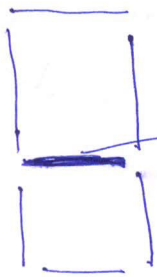
BCD-to-7-segment decoder



w3	w2	w1	w0	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	0	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1



Here 1's indicate LED glows \nearrow
 abcdefg = 111110



Arithmetic comparison ckt's / comparators

1. It compares the relative sizes of 2 binary numbers.
2. It is also called as comparator.

Design of a comparator

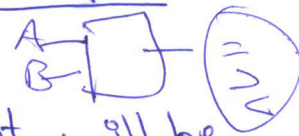
1. let A & B which represent unsigned binary numbers of each of "n" bit inputs.

2. comparator produce 3 types of output's.

a. $A \text{ eq } B \longrightarrow A = B.$

b. $A \text{ gt } B \longrightarrow A > B.$

c. $A \text{ lt } B \longrightarrow A < B.$



output will be set to 1.

3. It can be designed by creating a truth table that specifies the 3 output's as functions of A & B. However, even for moderate values of "n", the truth table is large.

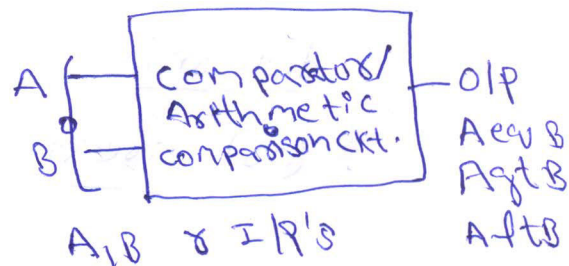
4. A better approach is to derive the comparator ckt by considering the bit's of A & B in pair's.

Example

1. let $n = 4$

$$A = a_3 a_2 a_1 a_0$$

$$B = b_3 b_2 b_1 b_0$$



2. Define intermediate signals called i_3, i_2, i_1 & i_0 .

3. Each signal i_k is "1" if bits of A & B with the same index are equal.

i.e. $i_k = a_k \oplus b_k$

If $i_3 = 1$

~~If~~ $i_2 = 1$

If $i_1 = 1$

If $i_0 = 1$

} o/p of comparator
is $A \text{ eq } B = i_3 i_2 i_1 i_0$

i.e. $A = a_3 a_2 a_1 a_0 = \begin{matrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{matrix}$
 $B = b_3 b_2 b_1 b_0 = \begin{matrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{matrix}$
 $i_3 i_2 i_1 i_0 = \underline{1111}$

$A \text{ eq } B = i_3 i_2 i_1 i_0$

4. $A > B$ can be derived by considering the bits of A & B in the MSB to LSB. The 1st bit position k at which a_k & b_k differ determines whether A is less than (or) greater than B.

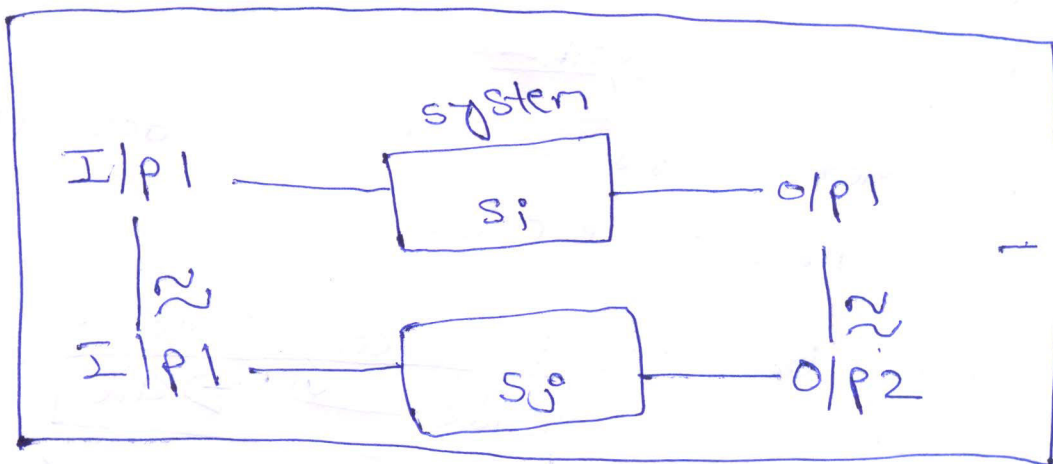
If $(a_k=0 \ \& \ b_k=1) \longrightarrow A < B$

If $(a_k=1 \ \& \ b_k=0) \longrightarrow A > B$.

$$\text{Agt B output} = a_3 \bar{b}_3 + i_3 a_2 \bar{b}_2 + i_3 i_2 a_1 \bar{b}_1 + i_3 i_2 i_1 a_0 \bar{b}_0$$

5. The i_k signals ensure that only the first digit's considered from the left to the right, of A and B that differ determine the value of AgtB.

$$A \neq B = \overline{A \text{ eq } B} + A \neq B$$



Literal

Implicant.

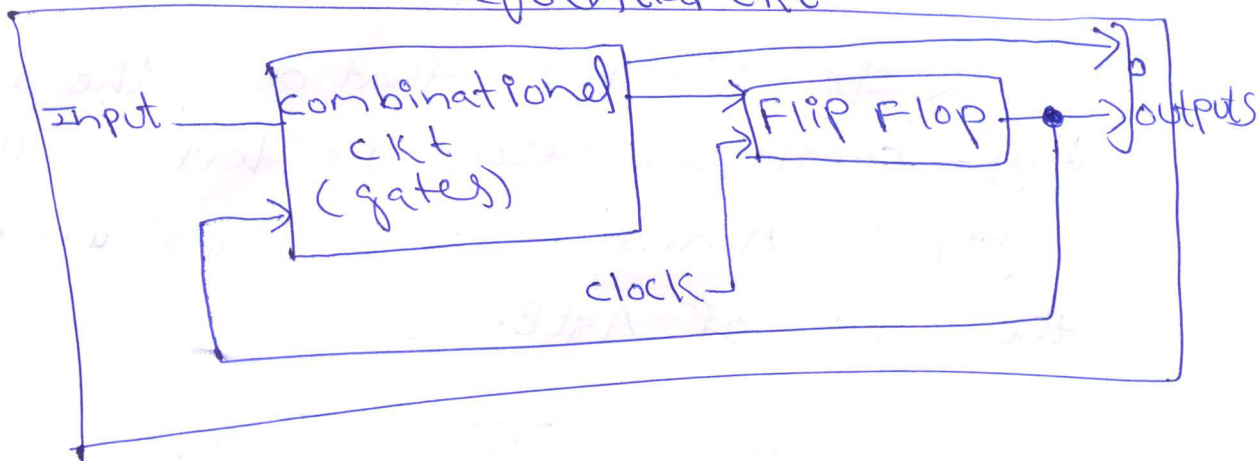
Prime

essent.

VHDL for combinational ckt's

1. The gates by themselves constitute a combinational ckt, but when included with the flip-flops, the overall ckt is classified as a sequential ckt.

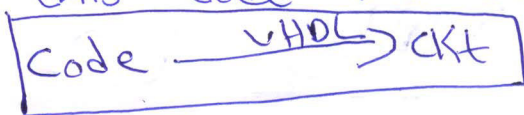
Sequential ckt.



Introduction to VHDL

1. Designer writes a logic ckt description in VHDL source code.

2. VHDL translates this code into a logic ckt.



3. Representation of digital signals in VHDL

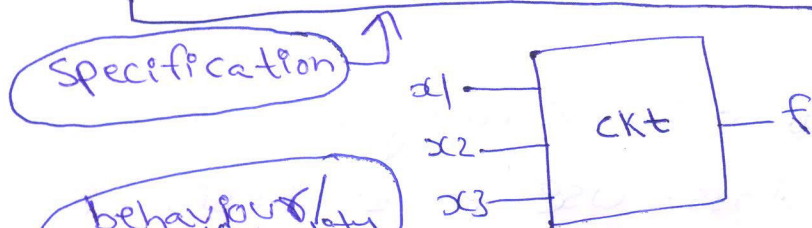
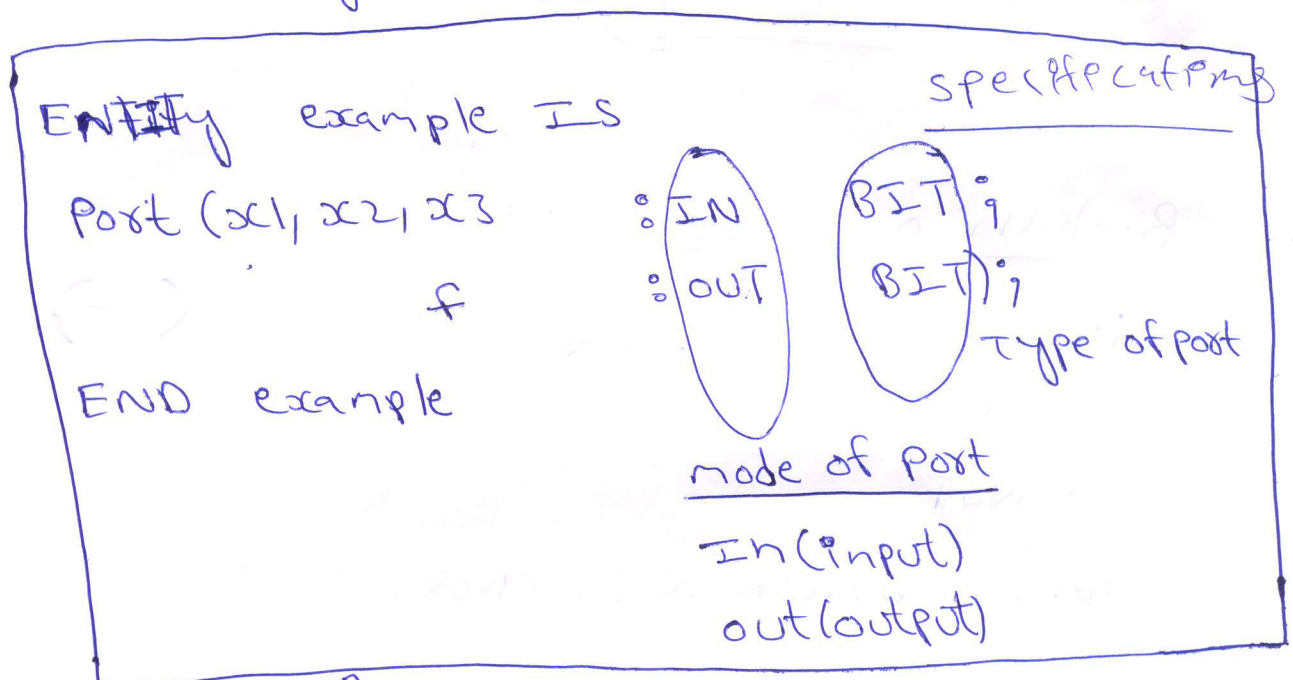
- a. logic signals in VHDL are represented as a data object.

- b. VHDL includes a data type called bit.

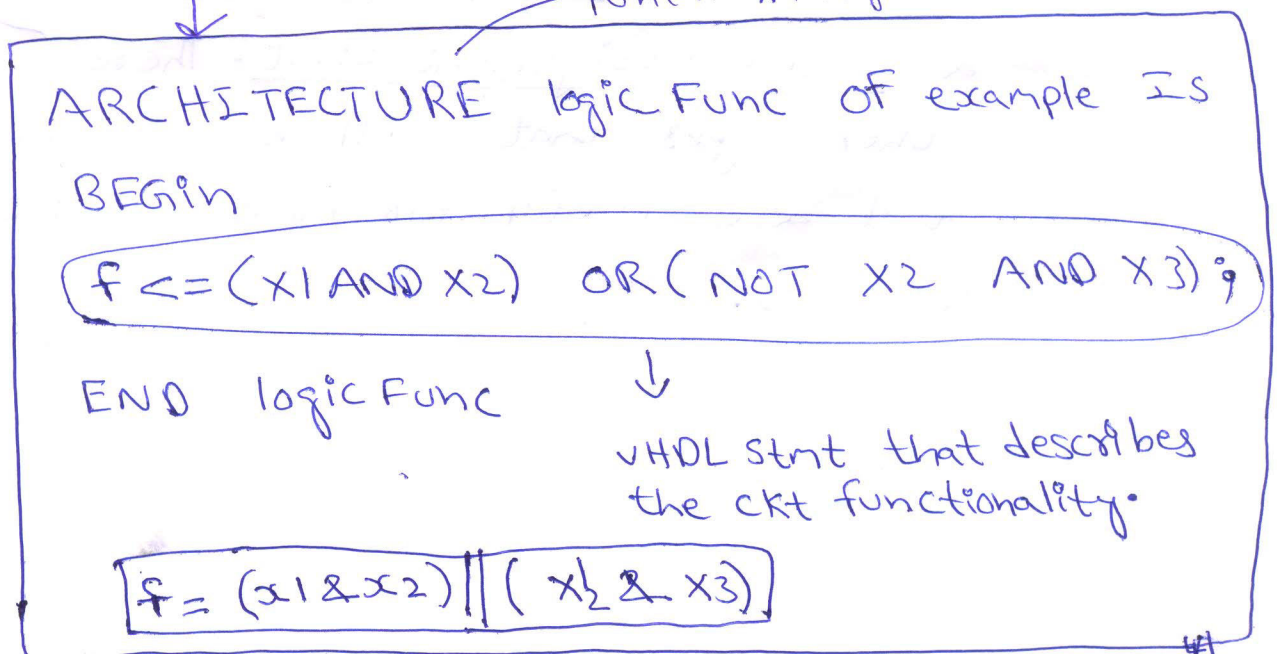
- c. BIT object's can assume only 2 values: 0 and 1.

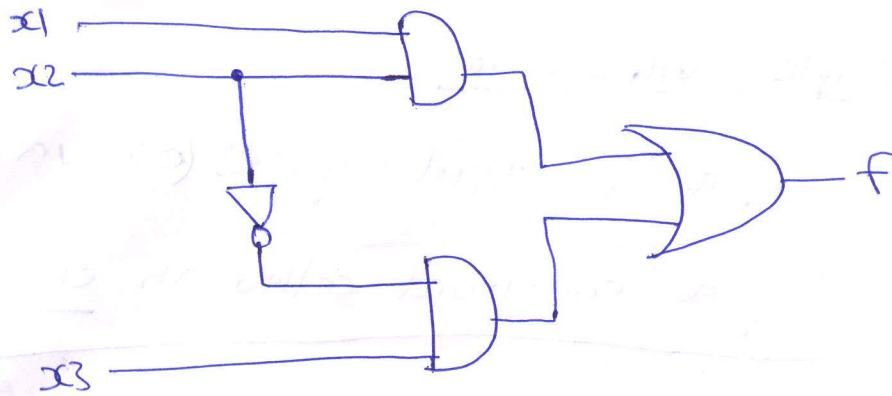
writing simple VHDL code

1. Declare input & output signals (or) variables.
2. Done using a construct called an entity.



3. ckt functionality is specified using a VHDL construct called an architecture.





Boolean operators in VHDL

1. AND
2. OR
3. NOT
4. NAND, NOR, XOR, XNOR.
5. Assignment operator (\leftarrow)

Note

1. VHDL don't assume any precedence of logic operators. Use parenthesis in expressions to determine precedence.
2. In VHDL, a logic expression is called a simple assignment stmt. There are other types that will be introduced that are useful for more complex circuits.

Assignment statements Types

VHDL provides several types of stmt's that can be used to assign logic values to signals.

1. simple assignments stmt's.

- used previously, for logic (or) arithmetic expressions.

2. selected signal assignments.

3. conditional signal assignments.

are similar.

4. Generate stmt's.

5. If-then-else stmt's.

6. case stmt's.

selected signal assignment \approx switch (with, when)

1. It allows a signal to be assigned one of several values, based on a selection criterion.

2. Keyword "WITH" specifies that "s" is used for the selection criterion.

3. Two when clauses state that $f = w_0$ when $s = 0$ & $f = w_1$ otherwise

4. Keyword "OTHERS" must be used.

$s=0 \longrightarrow f=w_0$

$\longrightarrow f=w_1$ otherwise.

ARCHITECTURE behavior OF mux2to1 IS

BEGIN

WITH S SELECT

$f \leq w_0$ WHEN '0';

w_1 WHEN OTHERS;

END behavior;

Design using VHDL

1. In VHDL, a logic signal is represented as a data object:

a. we used a BIT data type before that could only take on the values 0 & 1.

b. Another data type, STD-LOGIC, is actually preferable because it can

(i) [0, 1, z (high impedance),
- (don't care) e.t.c]

(ii) The STD-LOGIC-VECTOR data type can be used for multi

-bit values.

2. we must declare the library where the data type exists, and declare that we will use the data type.

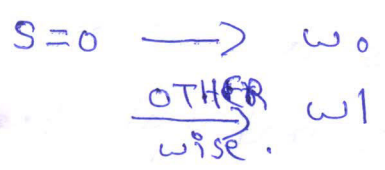
```
LIBRARY ieee;  
USE ieee.std_logic_1164.all;
```

2-to-1 multiplexer VHDL code

```
LIBRARY ieee;  
USE ieee.std_logic_1164.all;
```

```
ENTITY mux2to1 IS  
    PORT (w0, w1, s : IN STD_LOGIC;  
          f : OUT STD_LOGIC);  
END mux2to1;
```

```
ARCHITECTURE behavior OF mux2to1 IS  
BEGIN  
    with s SELECT  
        f <= w0 WHEN '0',  
           w1  WHEN OTHERS;  
END behavior;
```



4-to-1 multiplexer VHDL code

```
LIBRARY ieee;
```

```
USE ieee.std_logic_1164.all;
```

```
ENTITY mux4to1 IS
```

```
PORT (w : IN STD_LOGIC_VECTOR (3 DOWNTO 0)  
      w(0), w(1), w(2), w(3)
```

```
      s : IN STD_LOGIC_VECTOR (1 DOWNTO 0)  
      s(0), s(1)
```

```
      f : OUT STD_LOGIC);
```

```
END mux4to1
```

```
ARCHITECTURE behavior OF mux4to1 IS
```

```
BEGIN
```

```
WITH s SELECT
```

```
f <= w(0) WHEN "00" ;
```

```
      w(1) WHEN "01" ;
```

```
      w(2) WHEN "10" ;
```

```
      w(3) WHEN others ;
```

```
END behavior;
```

conditional signal assignment ~ switch

1. similar to the selected signal

assignment, a conditional signal

assignment allows a signal to be

set to one of several values.

2. uses WHEN & ELSE keyword to define the condition & actions.

```
ENTITY mux2to1 IS
PORT (w0, w1, s : IN STD-LOGIC;
      f : OUT STD-LOGIC);
END mux2to1;
```

```
ARCHITECTURE behavior OF mux2to1 IS
BEGIN
  f <= w0 WHEN s = '0' ELSE w1;
END behavior
```

Behavioral versus structural VHDL

1. The previous VHDL code examples are termed behavioral VHDL because they describe the behavior of a ckt without describing exactly how it is implemented in hardware.
2. Another VHDL coding style is structural.
 - a. For structural VHDL, the user typically describes the structure of a design by interconnecting several simpler design's from a library.
 - b. The library component's may be user-defined (or) provided by the CAD Tool vendor.

Generate statements

1. whenever we write structural VHDL code, we often create instances of a particular component.

a. A multi-stage ripple carry adder made from a number of single-bit full adder's might be an example.

2. If we need to create a large number of instances of a component, a more compact form is desired.

3. VHDL provides a feature called the FOR GENERATE stmt.

a. This stmt provides a loop structure for describing regularly structured hierarchical code.

4-bit ripple carry adder

```
LIBRARY ieee;  
USE ieee.std_logic_1164.all;  
USE WORK.fulladd-package.all;
```

ENTITY adder4 IS

```
PORT (cin      : IN    STD_LOGIC;  
      x, y     : IN    STD_LOGIC_VECTOR (3 DOWNTO 0);  
      s        : OUT   STD_LOGIC_VECTOR (3 DOWNTO 0);  
      cout     : OUT   STD_LOGIC);
```

END adder4

ARCHITECTURE structure OF adder4 IS

BEGIN

```
SIGNAL c : STD_LOGIC_VECTOR (0 TO 4);
```

BEGIN

```
c(0) <= cin;
```

```
cout <= c(4);
```

```
G1: FOR i IN 0 TO 3 GENERATE  
  stages : fulladd PORT MAP (c(i), x(i),  
                             y(i), s(i), c(i+1));
```

```
END GENERATE;
```

```
END structure.
```


Process statement

1. several types of assignment stmt's all have the property that the order in which they appear in VHDL code don't affect the meaning of the code.

2. Because of this property, these stmt's are called concurrent assignment stmt's.

3. VHDL provides a 2nd category of stmt's, sequential assignment stmt's, for which the ordering of the stmt's may affect the meaning of the code.

a. If-then-else and case stmt's are sequential.

4. VHDL requires that sequential assignment stmt's be placed inside another stmt, the process stmt.

5. The process stmt/ simply process, begins with the PROCESS keyword, followed by a parenthesized list of signals called the sensitivity list.

4-bit ripple carry adder

```
LIBRARY IEEE;  
USE IEEE.STD_LOGIC_1164.ALL;  
USE WORK.fulladd-package.all;
```

ENTITY adder4 IS

```
PORT (cin      : IN  STD_LOGIC;  
      x, y     : IN  STD_LOGIC_VECTOR (3 DOWNTO 0);  
      s        : OUT STD_LOGIC_VECTOR (3 DOWNTO 0);  
      cout     : OUT STD_LOGIC);
```

END adder4

ARCHITECTURE structure OF adder4 IS

BEGIN

```
SIGNAL c : STD_LOGIC_VECTOR (0 TO 4);
```

BEGIN

```
c(0) <= cin;
```

```
cout <= c(4);
```

```
G1: FOR i IN 0 TO 3 GENERATE  
  stages : fulladd PORT MAP (c(i), x(i),  
                             y(i), s(i), c(i+1));
```

```
END GENERATE;
```

```
END structure.
```

- a. This list includes, at most, all the signals used inside the process.
 - b. Generally the list includes all signals that can be used to "activate" the process.
6. Stmt's inside the process are evaluated in sequential order.
 7. Assignment made inside the process are not visible outside the process until all stmt's in the process have been evaluated.
 - a. If there are multiple assignment to the same signal inside a process, only the last one has any visible effect.

2-to-1 mux as a process

ARCHITECTURE behavior OF mux2to1 IS
BEGIN

```

PROCESS (w0, w1, s)
  BEGIN
    ① IF s = '0' THEN
      f <= w0; ①
    ELSE
      f <= w1;
    END IF;
  END PROCESS;
END Behavior;
  
```

sensitivity list, whenever a list entry changes, the process is re-evaluated (activated).

Priority Encoder (IF-THEN-ELSE)

ARCHITECTURE ~ behaviors OF priority IS

```
BEGIN
```

```
PROCESS (w)
```

```
BEGIN
```

```
z <= "100";
```

```
IF w(1) = '1' THEN z <= "01" - ENDIF;
```

```
" " 2 " " " 10 " ;
```

```
" " 3 " " " 11 " ;
```

```
z <= '1';
```

```
IF w = '0000' THEN z <= '0' ENDIF;
```

```
END PROCESS;
```

```
END behavior;
```

Case statement

1. It is similar to a selected assignment stmt in that the case stmt has a selection signal & includes WHEN clauses for various valuations of the selection signal.
2. Begins with a CASE keyword.
3. Each WHEN clause specifies

the stmt's that should be evaluated when the selection signal has a specified value.

4. The case stmt must include a when clause for all valuations of the selection signal.

a. use the OTHERS keyword.

2-to-1 mux with CASE

ARCHITECTURE behavior of mux2to1 is

BEGIN

PROCESS (w0, w1, s)

BEGIN

CASE s IS

WHEN '0' => f <= w0;

WHEN OTHERS => f <= w1;

END CASE

END PROCESS

END behavior

Example

process (a)

variable count: integer := 1;

begin

count := count + 1;

end process

count contain total no of event's that occurred on signal a.

Process Stmt

variables can be declared & used inside the process stmt only. But they retain their value throughout the entire simulation.

2-to-4 binary decoder with CASE

ARCHITECTURE behavior OF dec2to4 IS

BEGIN

PROCESS(W, En)

BEGIN

IF EN = '1' THEN

CASE W IS

WHEN "00" => Y = "1000";

WHEN "01" => Y = "0100";

WHEN "10" => Y = "0010";

WHEN OTHERS => Y = "0001";

ELSE

Y <= "0000";

END IF

END PROCESS

END behavior.

BEGIN

stage 0 : full add - PORT MAP(cin, X(0), Y(0), S(0), c(1));

stage 1 : full add - PORT MAP(c(1), X(1), Y(1), S(1), c(2));

stage 2 : full add - PORT MAP(c(2), X(2), Y(2), S(2), c(3));

stage 3 : full add - PORT MAP(c(3), X(3), Y(3), S(3), out);

END structure.

Behavioral VHDL descriptions

```
LIBRARY ieee;  
use ieee.std_logic_1164.all;  
use ieee.numeric_std.all  
-- provides arithmetic functions for vectors.
```

```
ENTITY adder16 IS  
PORT (X, Y : IN UNSIGNED (IS DOWNTO 0);  
      S : OUT UNSIGNED (IS DOWNTO 0));  
END adder16
```

```
ARCHITECTURE Behaviour of adder16 IS  
BEGIN  
    S <= X + Y;  
END Behaviour;
```

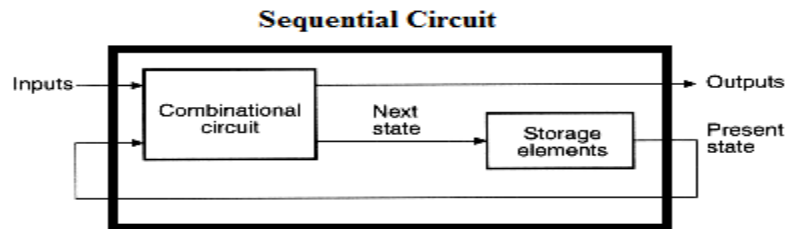
we are really describing the behavior of the ckt.

UNIT – III

Basic Latch Gated SR Latch, Gated D Latch, Master-Slave and Edge- Triggered D Flip-Flops, T Flip-flop, JK Flip-flop, Excitation tables. Registers-Shift Register, Counters-Asynchronous and synchronous counters, Ring counter, Johnson counter, VHDL code for D Flip-flop and Up-counter

SEQUENTIAL CIRCUITS

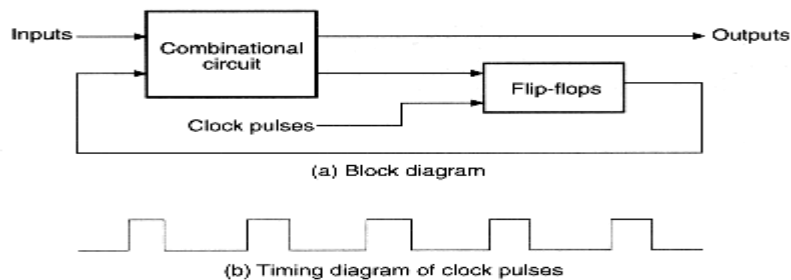
1. It consists of a combinational circuit (used to perform operations like decoding, encoding, addition and subtraction e.t.c. And interconnected with storage elements.) And a memory element (flip-flops). Flip-flops will be given a clock.
2. In combinational circuit, the output is only a function of all inputs and given any combination of inputs, it is always possible to predict the output. In sequential circuit, the output is not only a function inputs but history of the input changes.
3. The gates in the combinational circuit determine the binary value to be stored in the flip-flop after each clock transition.
4. The output of flip-flops in turn are applied to the combinational circuit as inputs and determine the circuits behavior.
5. External outputs of a sequential circuit are functions of both external inputs and the present state of the flip-flops.



SYNCHRONOUS CIRCUITS

In asynchronous sequential circuits, input changes at any time may result in the any of of the outputs or internally stored information (called state) to change. Such circuits are difficult to design because of dependence on propagation delays and their interaction with timing of input changes.

A synchronous sequential makes use of clock signals so that the storage elements (and outputs) only change at discrete instants of time in relation to the clock signal. Clocked synchronous circuits provide some degree of independence on timing variations related to gate propagation delays.

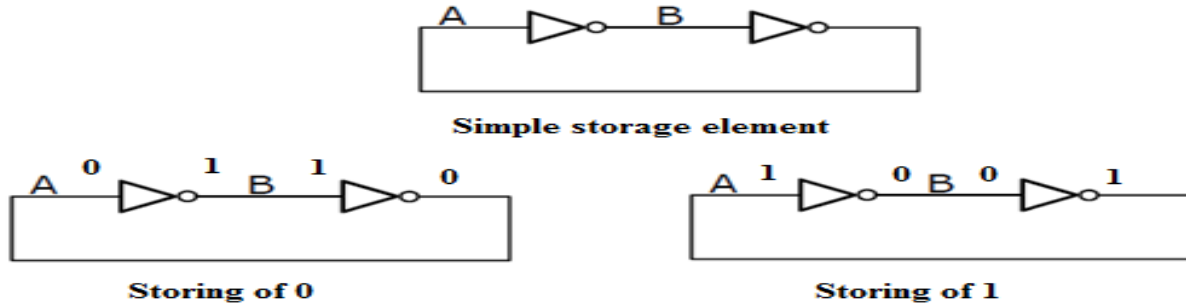


UNIT – III

Types of Triggering or Activations	
<p>1. Edge Triggering.</p> <p>Edge Triggering is of two types</p> <ol style="list-style-type: none"> 1. Negative Edge Triggering. 2. Positive Edge Triggering. 	<p>2. Level or pulse Triggered.</p> <p>Level Triggered</p> <p>In this type of circuits, the circuit may get activated Either at the High or Low Level.</p>
<p>Negative Edge Triggering (On = from 1 to 0, off = other time). It is a type of Triggering circuit, which will get activated at the time of negative going input signal [Trigger signal]. I.e. When the Trigger signal Falls from High (logic 1) to Low (Logic 0).</p> <p>Positive Edge Triggering (On = from 0 to 1, off = other time). It is another type of Triggering Circuit, which will get activated at the time of Positive going Input signal [Increasing]. i.e. When the trigger signal rises from Low (logic 0) to High (Logic 1)</p>	

SIMPLE MEMORY ELEMENT

It uses feedback path for the basis of remembering data.



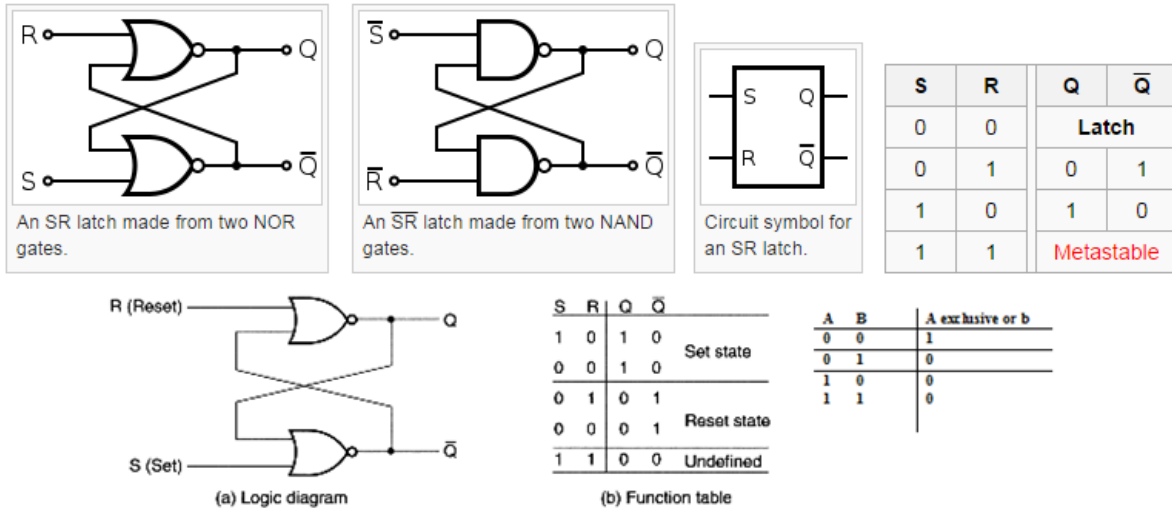
SR LATCH (SET/RESET)

It is an asynchronous device and works independently of control signals and relies only on the state of the S and R inputs. SR latch created with two NOR gates that have a cross-feedback loop. SR latches can also be made from NAND gates, but the inputs are swapped and negated. In this case, it is sometimes called an **SR latch**.

When a high is applied to the *Set* line of an SR latch, the *Q* output goes high (and *Q* low). The feedback mechanism, however, means that the *Q* output will remain high, even when the *S* input goes low again. This is how the latch serves as a memory device. Conversely, a high input on the *Reset* line will drive the *Q* output low (and *Q* high), effectively resetting the latch's "memory". When both inputs are low, the latch "latches" – it remains in its previously set or reset state.

UNIT – III

When both inputs are high at once, however, there is a problem: it is being told to simultaneously produce a high Q and a low Q . This produces a "race condition" within the circuit - whichever flip-flop succeeds in changing first will feedback to the other and assert itself. Ideally, both gates are identical and this is "meta-stable", and the device will be in an undefined state for an indefinite period. In real life, due to manufacturing methods, one gate will always win, but it's impossible to tell which it will be for a particular device from an assembly line. The state of $S = R = 1$ is therefore "illegal" and should never be entered.



When the device is powered up, a similar condition occurs, because both outputs, Q and \overline{Q} , are low. Again, the device will quickly exit the meta-stable state due to differences between the two gates, but it's impossible to predict which of Q and \overline{Q} will end up high. To avoid spurious actions, you should always set SR flip-flops to a known initial state before using them - you must not assume that they will initialize to a low state.

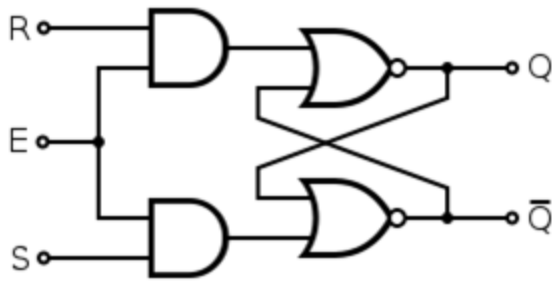
GATED SR LATCH

In some situations it may be desirable to dictate when the latch can and cannot latch. The **gated SR latch** is a simple extension of the SR latch which provides an *Enable* line which must be driven high before data can be latched. Even though a control line is now required, the SR latch is not synchronous, because the inputs can change the output even in the middle of an enable pulse.

When the *Enable* input is low, then the outputs from the AND gates must also be low, thus the Q and \overline{Q} outputs remain latched to the previous data. Only when the enable input is high can the state of the latch change, as shown in the truth table. When the enable line is asserted, a gated SR latch is identical in operation to an SR latch.

The *Enable* line is sometimes a clock signal, but is usually a read or write strobe.

UNIT – III

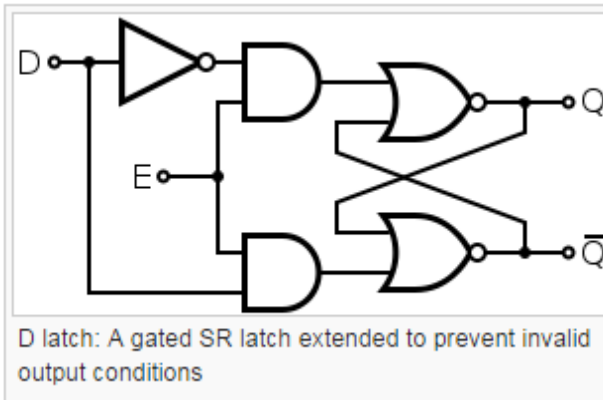


Enable	S	R	Q	\bar{Q}
0	0	0	Latch	
0	0	1	Latch	
0	1	0	Latch	
0	1	1	Latch	
1	0	0	Latch	
1	0	1	0	1
1	1	0	1	0
1	1	1	Metastable	

D LATCH (D FOR "DATA") OR TRANSPARENT LATCH

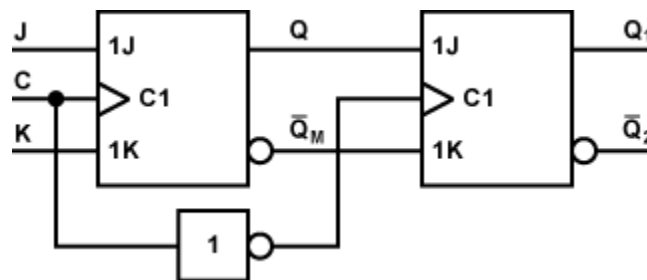
It is a simple extension of the gated SR latch that removes the possibility of invalid input states. Since the gated SR latch allows us to latch the output without using the *S* or *R* inputs, we can remove one of the inputs by driving both the *Set* and *Reset* inputs with a complementary driver: we remove one input and automatically make it the inverse of the remaining input.

The D latch outputs the *D* input whenever the *Enable* line is high, otherwise the output is whatever the *D* input was when the *Enable* input was last high. This is why it is also known as a transparent latch - when *Enable* is asserted, the latch is said to be "transparent" - it signals propagate directly through it as if it isn't there.



Enable	D	Q	\bar{Q}
0	0	Latch	
0	1	Latch	
1	0	0	1
1	1	1	0

MASTER-SLAVE FLIP FLOP



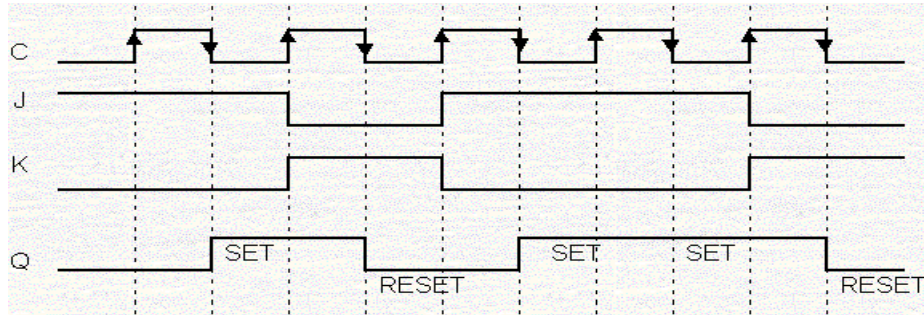
Master Slave JK Flip Flop

UNIT – III

A master slave flip flop contains two clocked flip flops (master (+ve edge triggered flip-flop) and the second slave (-ve edge triggered flip-flop)). It responds to the +ve level of the clock (clock is high). i.e. master is active. The output of the master is set or reset according to the state of the input. As the slave is inactive during this period its output remains in the previous state. When clock becomes low the output of the slave flip flop changes because it becomes active during low clock period. The final output of master slave flip flop is the output of the slave flip flop. So the output of master slave flip flop is available at the end of a clock pulse.

WORKING

When $Clk=1$, the master J-K flip flop gets disabled. The Clk input of the master input will be the opposite of the slave input. So the master flip flop output will be recognized by the slave flip flop only when the Clk value becomes 0. Thus, when the clock pulse makes a transition from 1 to 0, the locked outputs of the master flip flop are fed through to the inputs of the slave flip-flop making this flip flop edge or pulse-triggered. To understand better take a look at the timing diagram illustrated below.



Thus, the circuit accepts the value in the input when the clock is HIGH, and passes the data to the output on the falling-edge of the clock signal. This makes the Master-Slave J-K flip flop a Synchronous device as it only passes data with the timing of the clock signal.

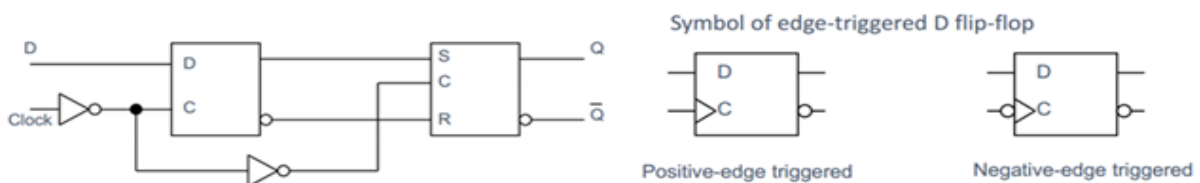
EDGE-TRIGGERED FLIP-FLOP PULSE-TRIGGERED

Read input while clock is 1, change output when the clock goes to 0. What happens during the entire HIGH part of clock can affect eventual output.

EDGE-TRIGGERED

Read input only on edge of clock cycle (positive or negative)

Example: Positive Edge-Triggered D Flip-Flop On the positive edge (while the clock is going from 0 to 1), the input D is read, and almost immediately propagated to the output Q. Only the value of D at the positive edge matters. **Or An edge-triggered flip-flop** changes states either at the positive edge (rising edge) or at the negative edge (falling edge) of the clock pulse on the control input. The three basic types are introduced here: S-R, J-K and D.



EXCITATION OR CHARACTERISTIC TABLE

1. It gives the relationship between input and outputs.

Relation given by excitation table

Inputs ← ----- → **outputs**

1. The characteristic tables of the flip-flops specify the next state when the inputs and present state are known.

Specify

FF (present or Input's known) ----- → **Next state.**

2. During design of sequential circuits we know the required transitions and wish to find the flip-flops inputs conditions that will cause the required transition.

To know

Q (T), Q (T+1) ----- → SR/D/JK/T.

3. For this reason we need a table that list the required input combinations for a given change of state such a table is called as Flip-flop excitation table.

(a) JK Flip-Flop				(b) SR Flip-Flop			
J	K	Q (t + 1)	Operation	S	R	Q (t + 1)	Operation
0	0	Q(t)	No change	0	0	Q(t)	No change
0	1	0	Reset	0	1	0	Reset
1	0	1	Set	1	0	1	Set
1	1	$\overline{Q}(t)$	Complement	1	1	?	Undefined

(c) D Flip-Flop			(d) T Flip-Flop		
D	Q (t + 1)	Operation	T	Q (t + 1)	Operation
0	0	Reset	0	Q(t)	No change
1	1	Set	1	$\overline{Q}(t)$	Complement

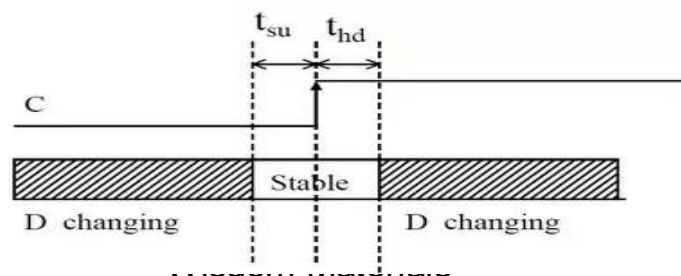
SETUP TIME

It is the minimum amount of time the data signal should be held steady before the clock event so that the data are reliably sampled by the clock. This applies to synchronous circuits such as the flip-flop. Or the amount of time the Synchronous input (D) must be stable before the active edge of the Clock. The Time when input data is available and stable before the clock pulse is applied is called Setup time.

HOLD TIME

It is the minimum amount of time the data signal should be held steady after the clock event so that the data are reliably sampled. This applies to synchronous circuits such as the flip-flop. Or The amount of time the synchronous input (D) must be stable after the active edge of clock. The Time after clock pulse where data input is held stable is called hold time.

Setup, Hold Time

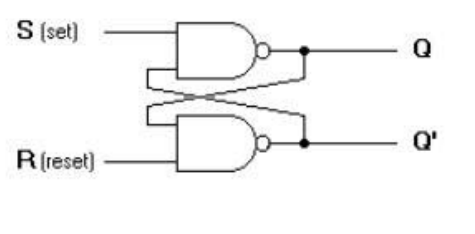
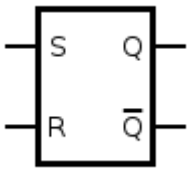
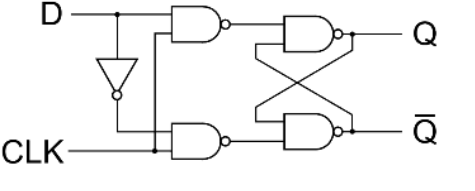
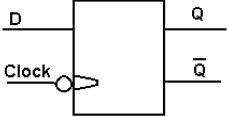
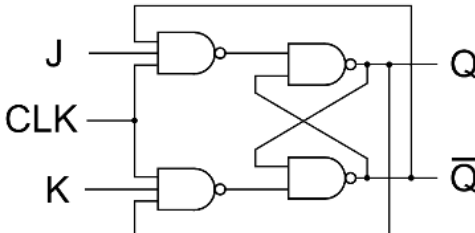
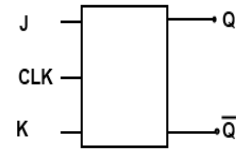
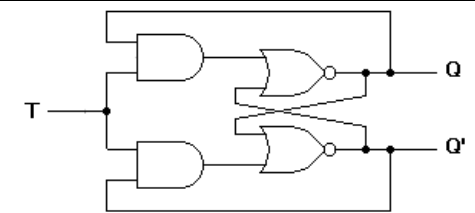
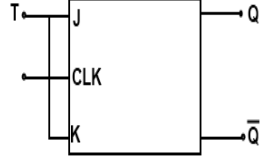


UNIT – III

FLIP FLOP

A digital computer needs devices which can store information. A flip flop is a binary storage device. It can store binary bit either 0 or 1. It has two stable states HIGH and LOW i.e. 1 and 0. It has the property to remain in one state indefinitely until it is directed by an input signal to switch over to the other state. It is also called bi-stable multi-vibrator.

They can be used to keep a record or what value of variable (input, output or intermediate). Flip flop are also used to exercise control over the functionality of a digital circuit i.e. change the operation of a circuit depending on the state of one or more flip flops. These devices are mainly used in situations which require one or more of these three Operations, storage and sequencing.

Type of Flip Flop	Circuit	Truth Or Excitation table	Graphic Symbol																				
SR		<p style="text-align: center;">(b) SR Flip-Flop</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>S</th> <th>R</th> <th>$Q(t + 1)$</th> <th>Operation</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>$Q(t)$</td> <td>No change</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> <td>Reset</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> <td>Set</td> </tr> <tr> <td>1</td> <td>1</td> <td>?</td> <td>Undefined</td> </tr> </tbody> </table>	S	R	$Q(t + 1)$	Operation	0	0	$Q(t)$	No change	0	1	0	Reset	1	0	1	Set	1	1	?	Undefined	
S	R	$Q(t + 1)$	Operation																				
0	0	$Q(t)$	No change																				
0	1	0	Reset																				
1	0	1	Set																				
1	1	?	Undefined																				
D		<p style="text-align: center;">(c) D Flip-Flop</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>D</th> <th>$Q(t + 1)$</th> <th>Operation</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>Reset</td> </tr> <tr> <td>1</td> <td>1</td> <td>Set</td> </tr> </tbody> </table>	D	$Q(t + 1)$	Operation	0	0	Reset	1	1	Set												
D	$Q(t + 1)$	Operation																					
0	0	Reset																					
1	1	Set																					
JK		<p style="text-align: center;">(a) JK Flip-Flop</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>J</th> <th>K</th> <th>$Q(t + 1)$</th> <th>Operation</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>$Q(t)$</td> <td>No change</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> <td>Reset</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> <td>Set</td> </tr> <tr> <td>1</td> <td>1</td> <td>$\bar{Q}(t)$</td> <td>Complement</td> </tr> </tbody> </table>	J	K	$Q(t + 1)$	Operation	0	0	$Q(t)$	No change	0	1	0	Reset	1	0	1	Set	1	1	$\bar{Q}(t)$	Complement	
J	K	$Q(t + 1)$	Operation																				
0	0	$Q(t)$	No change																				
0	1	0	Reset																				
1	0	1	Set																				
1	1	$\bar{Q}(t)$	Complement																				
T		<p style="text-align: center;">T Flip-Flop</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>T</th> <th>$Q(t + 1)$</th> <th>Operation</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>$Q(t)$</td> <td>No change</td> </tr> <tr> <td>1</td> <td>$\bar{Q}(t)$</td> <td>Complement</td> </tr> </tbody> </table>	T	$Q(t + 1)$	Operation	0	$Q(t)$	No change	1	$\bar{Q}(t)$	Complement												
T	$Q(t + 1)$	Operation																					
0	$Q(t)$	No change																					
1	$\bar{Q}(t)$	Complement																					

UNIT – III

REGISTERS

1. A flip-flop stores one bit of information.
2. Group of n flip-flops is called as a register and is used to store n bits of information.
3. Data can be moved from one register or flip-flop to another is known as loading.
4. Data can be moved from left shift, right and parallel from one register or flip-flop to another.
5. Normally “D flip-flop” are used for building or constructing a register.
6. A register that provides the ability to shift its contents is called a shift register.

SHIFT REGISTER

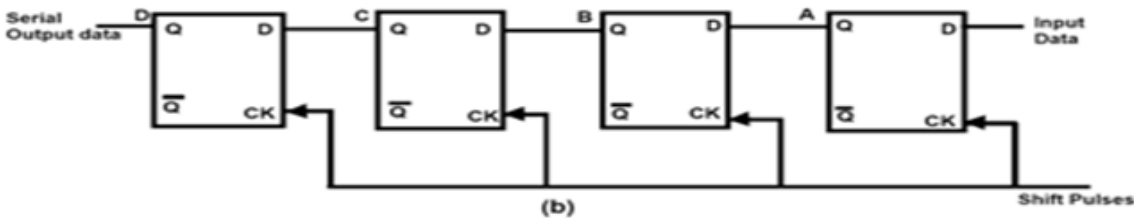
It is a storage device that used to store binary data. When a number of flip flop are connected in series it is called a register. A single flip flop is supposed to stay in one of the two stable states 1 or 0 or in other words the flip flop contains a number 1 or 0 depending upon the state in which it is. A register will thus contain a series of bits which can be termed as a word or a byte.

If in these registers the connection is done in such a way that the output of one of the flip flop forms in input to other, it is known as a shift register. The data in a shift register is moved serially (one bit at a time). The shift register can be built using RS, JK or D flip-flops various types of shift registers are available some of them are given as under.

1. Shift Left Register.
2. Shift Right Register.
3. Shift around Register.
4. Bi-directional Shift Register.

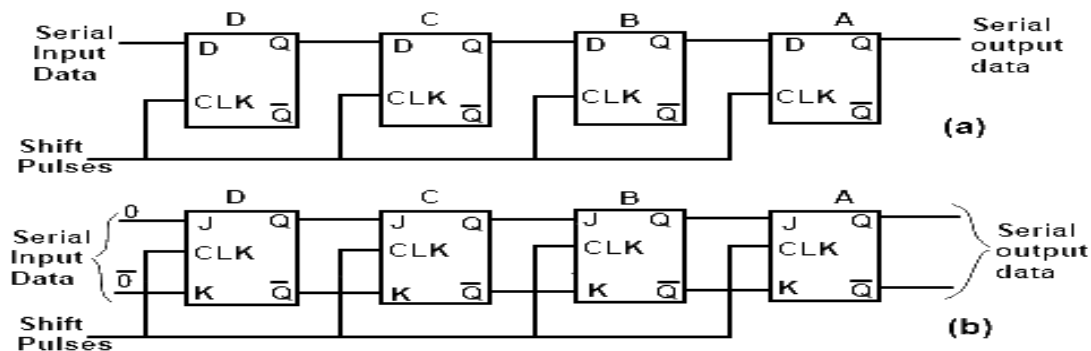
SHIFT LEFT REGISTER

It is used to move or shift data (bits) from right to left with in a register.



SHIFT RIGHT REGISTER

It is used to move or shift data (bits) from right to left with in a register.

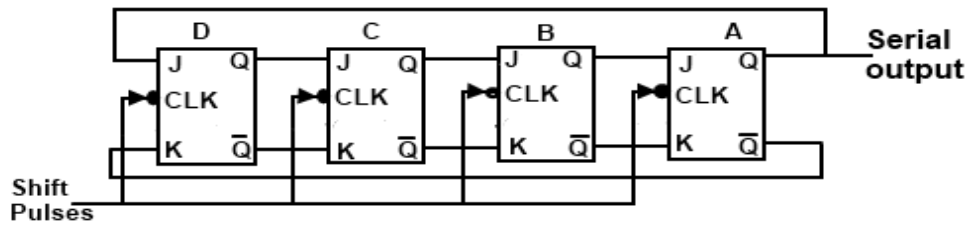


Shift Right Register (a) D-type (b) JK

UNIT - III

SHIFT AROUND REGISTER

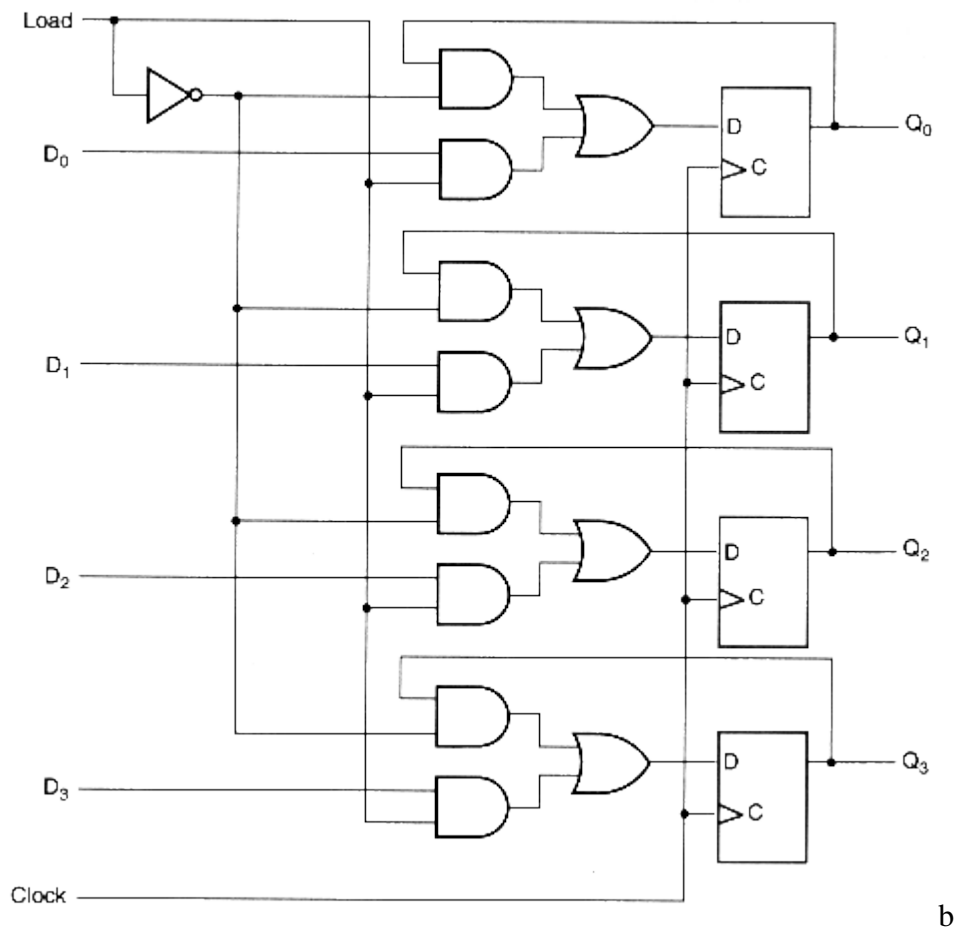
It is used to shift the data in a circular manner.



Bidirectional Shift Register

A bidirectional shift register is one which can do both the shift left and shift right operations.

PARALLEL LOAD



4 BIT SHIFT REGISTER WITH PARALLEL LOAD

UNIT – III

THE SHIFT REGISTER

It is another type of sequential logic circuit that can be used for the storage or the transfer of data in the form of binary numbers. This sequential device loads the data present on its inputs and then moves or “shifts” it to its output once every clock cycle, hence the name “shift register”.

A shift register basically consists of several single bit “D-Type Data Latches”, one for each data bit, either a logic “0” or a “1”, connected together in a serial type daisy-chain arrangement so that the output from one data latch becomes the input of the next latch and so on.

Data bits may be fed in or out of a shift register serially, that is one after the other from either the left or the right direction, or all together at the same time in a parallel configuration.

The number of individual data latches required to make up a single Shift Register device is usually determined by the number of bits to be stored with the most common being 8-bits (one byte) wide constructed from eight individual data latches.

Shift Registers are used for data storage or for the movement of data and are therefore commonly used inside calculators or computers to store data such as two binary numbers before they are added together, or to convert the data from either a serial to parallel or parallel to serial format. The individual data latches that make up a single shift register are all driven by a common clock (Clk) signal making them synchronous devices.

Shift register IC’s are generally provided with a clear or reset connection so that they can be “SET” or “RESET” as required. Generally, shift registers operate in one of four different modes with the basic movement of data through a shift register being:

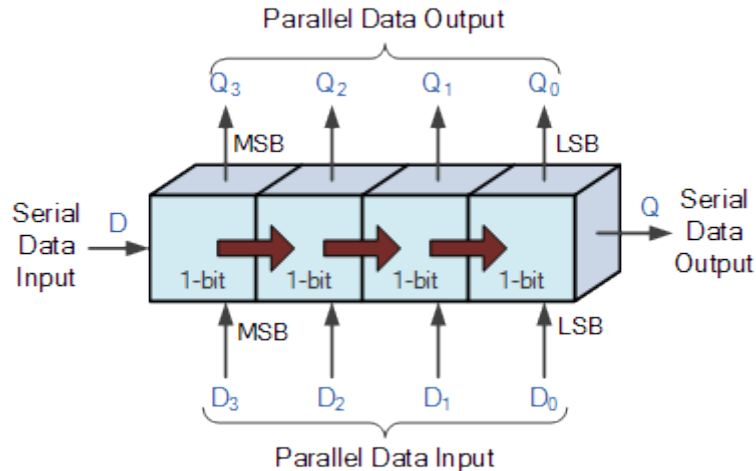
SERIAL-IN TO PARALLEL-OUT (SIPO) - the register is loaded with serial data, one bit at a time, with the stored data being available at the output in parallel form.

SERIAL-IN TO SERIAL-OUT (SISO) - the data is shifted serially “IN” and “OUT” of the register, one bit at a time in either a left or right direction under clock control.

PARALLEL-IN TO SERIAL-OUT (PISO) - the parallel data is loaded into the register simultaneously and is shifted out of the register serially one bit at a time under clock control.

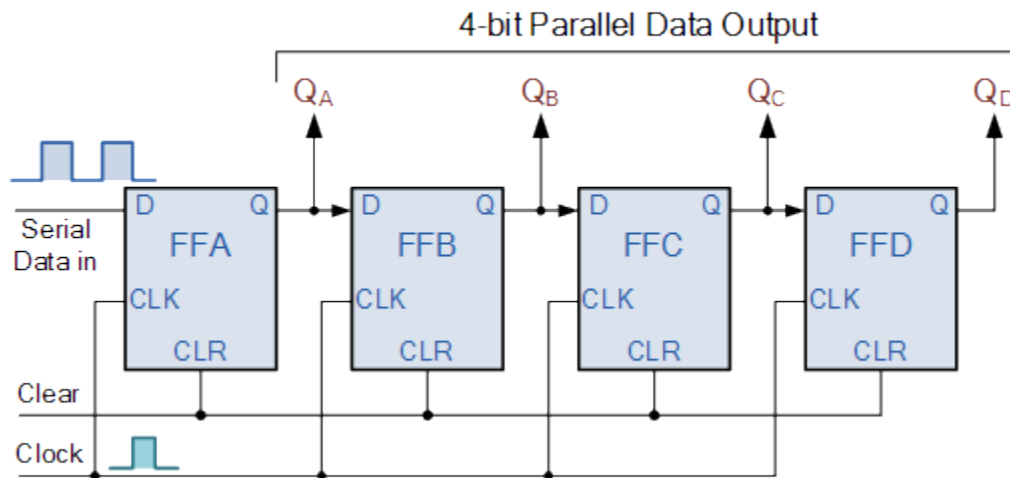
PARALLEL-IN TO PARALLEL-OUT (PIPO) - the parallel data is loaded simultaneously into the register, and transferred together to their respective outputs by the same clock pulse. The effect of data movement from left to right through a shift register can be presented graphically as:

UNIT – III



Also, the directional movement of the data through a shift register can be either to the left, (left shifting) to the right, (right shifting) left-in but right-out, (rotation) or both left and right shifting within the same register thereby making it bidirectional. In this tutorial it is assumed that all the data shifts to the right, (right shifting).

SERIAL-IN TO PARALLEL-OUT (SIPO) SHIFT REGISTER 4-BIT SERIAL-IN TO PARALLEL-OUT SHIFT REGISTER



The operation is as follows. Lets assume that all the flip-flops (FFA to FFD) have just been RESET (CLEAR input) and that all the outputs Q_A to Q_D are at logic level “0” ie, no parallel data output.

If a logic “1” is connected to the DATA input pin of FFA then on the first clock pulse the output of FFA and therefore the resulting Q_A will be set HIGH to logic “1” with all the other outputs still remaining LOW at logic “0”. Assume now that the DATA input pin of FFA has returned LOW again to logic “0” giving us one data pulse or 0-1-0.

The second clock pulse will change the output of FFA to logic “0” and the output of FFB and Q_B HIGH to logic “1” as its input D has the logic “1” level on it from Q_A . The

UNIT – III

logic “1” has now moved or been “shifted” one place along the register to the right as it is now at QA.

When the third clock pulse arrives this logic “1” value moves to the output of FFC (QC) and so on until the arrival of the fifth clock pulse which sets all the outputs QA to QD back again to logic level “0” because the input to FFA has remained constant at logic level “0”.

The effect of each clock pulse is to shift the data contents of each stage one place to the right, and this is shown in the following table until the complete data value of 0-0-0-1 is stored in the register. This data value can now be read directly from the outputs of QA to QD.

Then the data has been converted from a serial data input signal to a parallel data output. The truth table and following waveforms show the propagation of the logic “1” through the register from left to right as follows.

BASIC DATA MOVEMENT THROUGH A SHIFT REGISTER

Clock Pulse No	QA	QB	QC	QD
0	0	0	0	0
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1
5	0	0	0	0



Note that after the fourth clock pulse has ended the 4-bits of data (0-0-0-1) are stored in the register and will remain there provided clocking of the register has stopped. In practice the input data to the register may consist of various combinations of logic “1” and “0”. Commonly available SIPO IC’s include the standard 8-bit 74LS164 or the 74LS594.

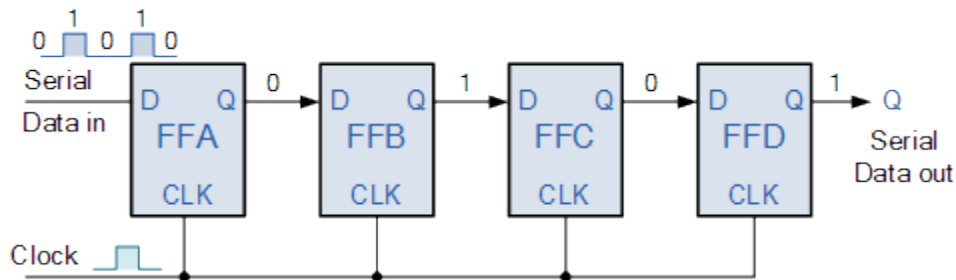
UNIT – III

SERIAL-IN TO SERIAL-OUT (SISO) SHIFT REGISTER

This shift register is very similar to the SIPO above, except were before the data was read directly in a parallel form from the outputs QA to QD, this time the data is allowed to flow straight through the register and out of the other end. Since there is only one output, the DATA leaves the shift register one bit at a time in a serial pattern, hence the name Serial-in to **Serial-Out Shift Register or SISO**.

The SISO shift register is one of the simplest of the four configurations as it has only three connections, the serial input (SI) which determines what enters the left hand flip-flop, the serial output (SO) which is taken from the output of the right hand flip-flop and the sequencing clock signal (Clk). The logic circuit diagram below shows a generalized serial-in serial-out shift register.

4-BIT SERIAL-IN TO SERIAL-OUT SHIFT REGISTER



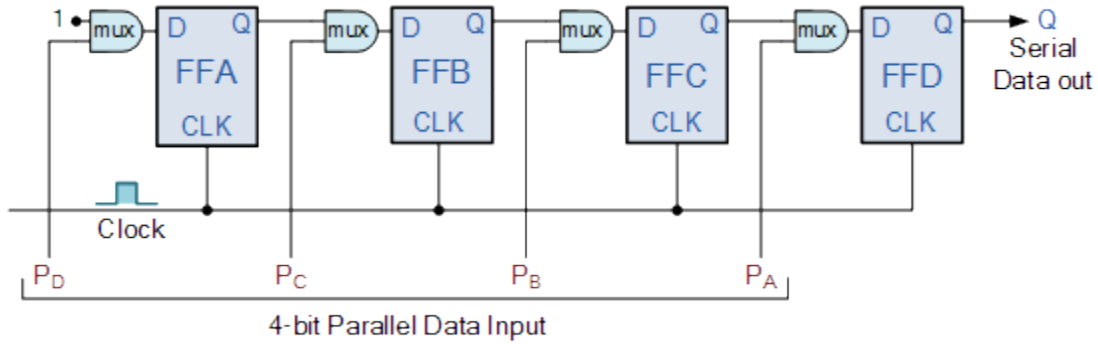
You may think what's the point of a SISO shift register if the output data is exactly the same as the input data. Well this type of Shift Register also acts as a temporary storage device or as a time delay device for the data, with the amount of time delay being controlled by the number of stages in the register, 4, 8, 16 etc or by varying the application of the clock pulses. Commonly available IC's include the 74HC595 8-bit Serial-in to Serial-out Shift Register all with 3-state outputs.

PARALLEL-IN TO SERIAL-OUT (PISO) SHIFT REGISTER

The Parallel-in to Serial-out shift register acts in the opposite way to the serial-in to parallel-out one above. The data is loaded into the register in a parallel format in which all the data bits enter their inputs simultaneously, to the parallel input pins PA to PD of the register. The data is then read out sequentially in the normal shift-right mode from the register at Q representing the data present at PA to PD.

This data is outputted one bit at a time on each clock cycle in a serial format. It is important to note that with this type of data register a clock pulse is not required to parallel load the register as it is already present, but four clock pulses are required to unload the data.

4-BIT PARALLEL-IN TO SERIAL-OUT SHIFT REGISTER

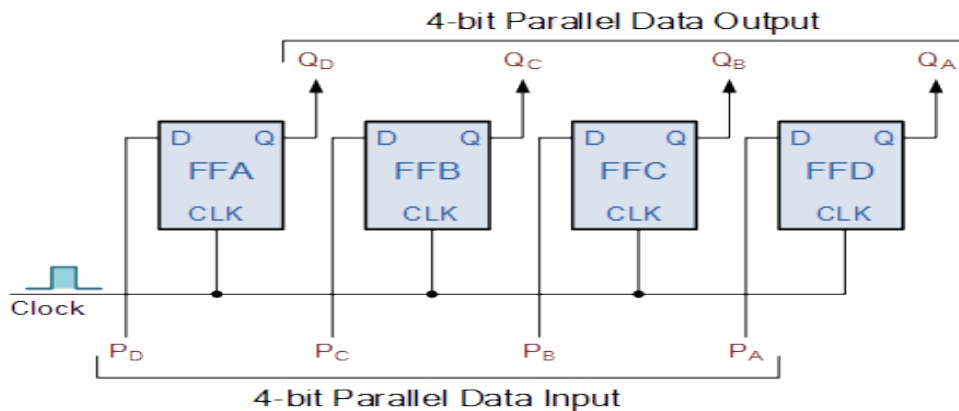


As this type of shift register converts parallel data, such as an 8-bit data word into serial format, it can be used to multiplex many different input lines into a single serial DATA stream which can be sent directly to a computer or transmitted over a communications line. Commonly available IC's include the 74HC166 8-bit Parallel-in/Serial-out Shift Registers.

PARALLEL-IN TO PARALLEL-OUT (PIPO) SHIFT REGISTER

The final mode of operation is the Parallel-in to Parallel-out Shift Register. This type of shift register also acts as a temporary storage device or as a time delay device similar to the SISO configuration above. The data is presented in a parallel format to the parallel input pins PA to PD and then transferred together directly to their respective output pins QA to QA by the same clock pulse. Then one clock pulse loads and unloads the register. This arrangement for parallel loading and unloading is shown below.

4-BIT PARALLEL-IN TO PARALLEL-OUT SHIFT REGISTER



The PIPO shift register is the simplest of the four configurations as it has only three connections, the parallel input (PI) which determines what enters the flip-flop, the parallel output (PO) and the sequencing clock signal (Clk).

Similar to the Serial-in to Serial-out shift register, this type of register also acts as a temporary storage device or as a time delay device, with the amount of time delay being varied by the frequency of the clock pulses. Also, in this type of register there are no interconnections between the individual flip-flops since no serial shifting of the data is required.

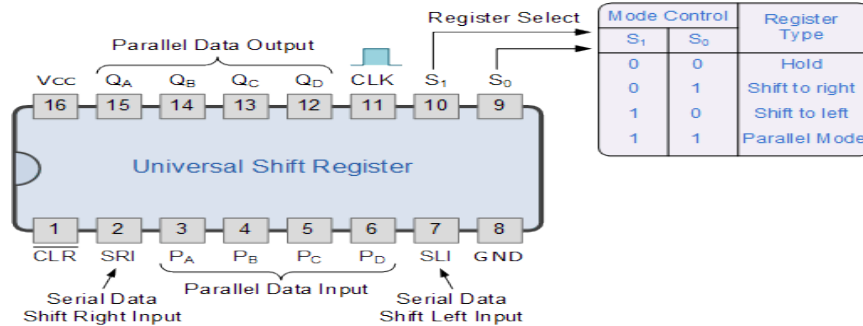
UNIT – III

UNIVERSAL SHIFT REGISTER

Today, there are many high speed bi-directional “universal” type Shift Registers available such as the TTL 74LS194, 74LS195 or the CMOS 4035 which are available as 4-bit multi-function devices that can be used in either serial-to-serial, left shifting, right shifting, serial-to-parallel, parallel-to-serial or parallel-to-parallel multifunction data register, hence the name “Universal”.

These universal shift registers can perform any combination of parallel and serial input to output operations but require additional inputs to specify desired function and to pre-load and reset the device. A commonly used universal shift register is the TTL 74LS194 as shown below.

4-BIT UNIVERSAL SHIFT REGISTER 74LS194



They can be configured to respond to operations that require some form of temporary memory storage or for the delay of information such as the SISO or PIPO configuration modes or transfer data from one point to another in either a serial or parallel format. Universal shift registers are frequently used in arithmetic operations to shift data to the left or right for multiplication or division.

SHIFT REGISTER TUTORIAL SUMMARY

1. A simple Shift Register can be made using only D-type flip-Flops, one flip-Flop for each data bit.
2. The output from each flip-Flop is connected to the D input of the flip-flop at its right.
3. Shift registers hold the data in their memory which is moved or “shifted” to their required positions on each clock pulse.
4. Each clock pulse shifts the contents of the register one bit position to either the left or the right.
5. The data bits can be loaded one bit at a time in a series input (SI) configuration or be loaded simultaneously in a parallel configuration (PI).
6. Data may be removed from the register one bit at a time for a series output (SO) or removed all at the same time from a parallel output (PO).
7. One application of shift registers is in the conversion of data between serial and parallel, or parallel to serial.
8. Shift registers are identified individually as SIPO, SISO, PISO, PIPO, or as a Universal Shift Register with all the functions combined within a single device.

In the next tutorial about Sequential Logic Circuits, we will look at what happens when the output of the last flip-flop in a shift register is connected directly back to the input of the first flip-flop producing a closed loop circuit that constantly re-circulates the data around the loop. This then produces another type of sequential logic circuit called a Ring Counter that are used as decade counters and dividers.

UNIT – III

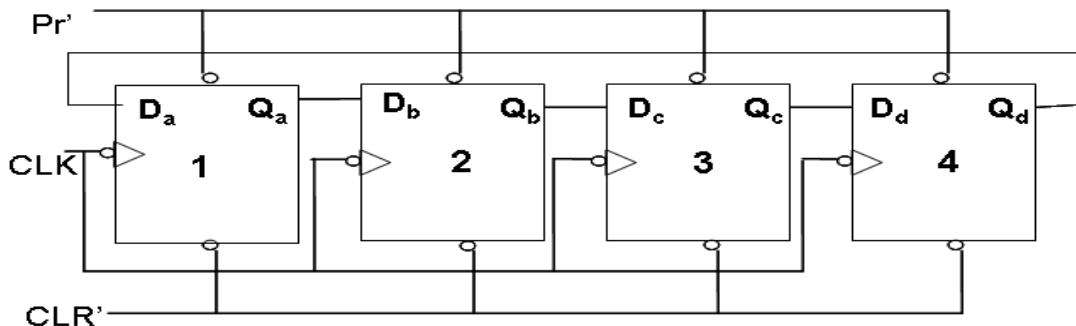
COUNTER

It is a sequential circuit that counts the input pulses given to a circuit. It is a register that goes through a predetermined sequence of states upon the application of input pulses.

TYPES OF COUNTERS	
SYNCHRONOUS COUNTER	ASYNCHRONOUS COUNTER
<p>Here input pulses are applied to all clock pulse inputs of all flip flops simultaneously (directly). Synchronous counter is also known as parallel sequential circuit.</p>	<p>Here the flip flop output transition serves as a source for triggering other flip flops. In other words, the clock pulse inputs of all flip flops, except the first, are triggered not by the incoming pulses, but rather by the transition that occurs in previous flip flop's output. Asynchronous counter is also known as serial sequential Circuit.</p>
<p>SYNCHRONOUS COUNTER TYPES</p> <ol style="list-style-type: none"> 1. Ring Counter. 2. Johnson Counter (Switch Tail or Twisted Ring Counter). 	<p>ASYNCHRONOUS COUNTER TYPES</p> <ol style="list-style-type: none"> 1. Binary Ripple Counter. 2. Up Down Counter.

SYNCHRONOUS COUNTER RING COUNTER

It is a synchronous counter since all the flip flops are clocked simultaneously. A ring counter is a circular shift register with only one flip flop being set at any particular time, all others are cleared. The single bit is shifted from one flip flop to the other to produce the sequence of timing signals.



UNIT – III

It is a 4-bit shift register connected as a ring counter. In this counter, Serial In D_a is connected to Serial Out Q_d . First of all, CLR' is set to 0 to clear all flip flops and then it is set to 1 for the circuit operation. After clearing all the flip flops, Pr' of 4th flip flop is set to 0 while for all other three flip flops, it set to 1. This is done so that the initial value of register becomes 0001 [$Pr'=0$, sets the 4th flip flop to 1]. Single bit is shifted right with every clock pulse. Each flip flop is in 1 state once every four clock pulses.

CLK	Q_a	Q_b	Q_c	Q_d
0	0	0	0	0
0	0	0	0	1
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

DISADVANTAGES OF RING COUNTER

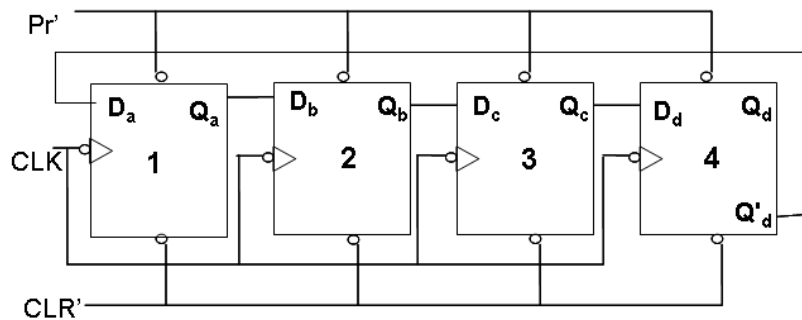
1. Using ring counter, one can count only four distinct states which is totally wastage of flip flops.
2. Ring counter doesn't count in a binary sequence so it is not preferred.

Mod of the Ring Counter is "n" where n is the number of flip flops. In a Ring Counter, the frequency of output is divided by n, therefore, it is known as **divide by N counter**.

JOHNSON COUNTER OR TWISTED RING COUNTER

An n-bit ring counter circulates a single bit among the flip flops to provide n distinct states. The number of states can be doubled if the shift register is connected as a switch tail ring counter.

A switch tail ring counter is a circular shift register with the complement output of the last flip flop connected to the input of the first flip flop. The circular connection is made from the complement output of the rightmost flip flop to the input of the leftmost flip flop. The register shifts its contents once to the right with every clock pulse and at the same time, the complement value of flip flop 4 are transferred to flip flop 1.



UNIT – III

Starting from cleared states, the 4-bit switch tail ring counter goes through a sequence of 8 states. In general, a k-bit switch tail ring counter will go through a sequence of **2k states**.

Starting from all 0's, each shift operation inserts 1's from the left until the register is filled with all 1's. In the following sequence 0's are inserted from the left until the register is again filled with all 0's.

CLK	Qa	Qb	Qc	Qd	Qd'
0	0	0	0	0	1
1	1	0	0	0	
2	1	1	0	0	
3	1	1	1	0	
4	1	1	1	1	0
5	0	1	1	1	
6	0	0	1	1	
7	0	0	0	1	
8	0	0	0	0	1

Mod of Johnson Counter is $2n$, therefore it is known as Divide by $2N$ Counter. Frequency of Output = frequency of Clock Pulse / mod

$$f_{\text{johnson}} = f / 2n$$

DISADVANTAGE OF JOHNSON COUNTER

It is that it doesn't count in a binary sequence. It can be constructed for any number of timing sequences. The number of flip flops needed is one half the numbers of timing signals.

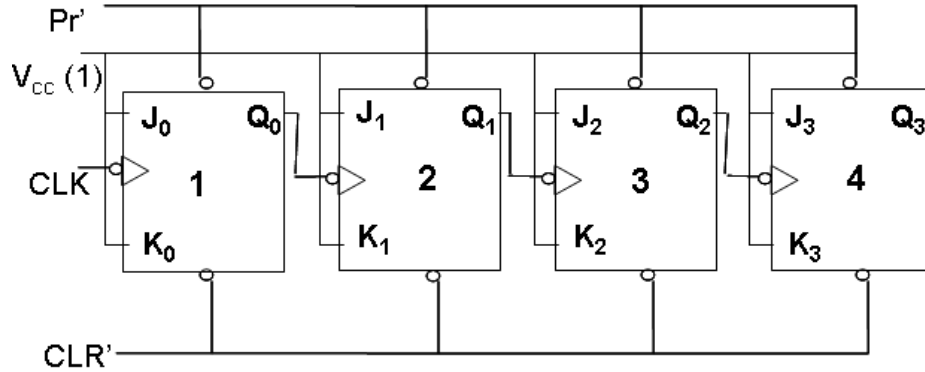
ASYNCHRONOUS COUNTER

BINARY RIPPLE COUNTER

A counter that follows the **binary sequence** is called a binary counter. A binary ripple counter consists of a series of complementing flip flops (T or JK FF) with the output of each flip flop connected to the clock pulse input of the next higher order flip flop. The flip flop holding the least significant bit receives the incoming count pulses.

It is known as **ripple counter** because the flip flops change one at a time in rapid succession and the signal propagates through the counter in a ripple fashion. CLK is coming for subsequent flip flops from previous flip flops and change state only when transition of previous flip flop's output is from high to low i.e. from 1 to 0.

UNIT – III



The lowest order bit Q_0 gets complimented with each count pulse. Every time Q_0 goes from 1 to 0, it complements Q_1 . Every time Q_1 goes from 1 to 0, it complements Q_2 and so on.

A **complimentary flip flop** can be obtained in 3 ways as described below:

1. Using T Flip Flop
2. Using JK Flip Flop with J & K inputs tied together
3. Using D Flip Flop with the complement output connected to the D input. In this way the D input is always the complement of the present state and next clock pulse will cause the flip flop to complement.

CLK	Q3	Q2	Q1	Q0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

UNIT – III

Mod of Binary Ripple Counter = 2^n , where n is the number of flip flops. The counter counts in a binary sequence from 0 to 2^{n-1}

DISADVANTAGE OF BINARY RIPPLE COUNTER

This counter is slow as delay of each flip flop has to be taken into account as they are not clocked simultaneously.

UP DOWN COUNTER

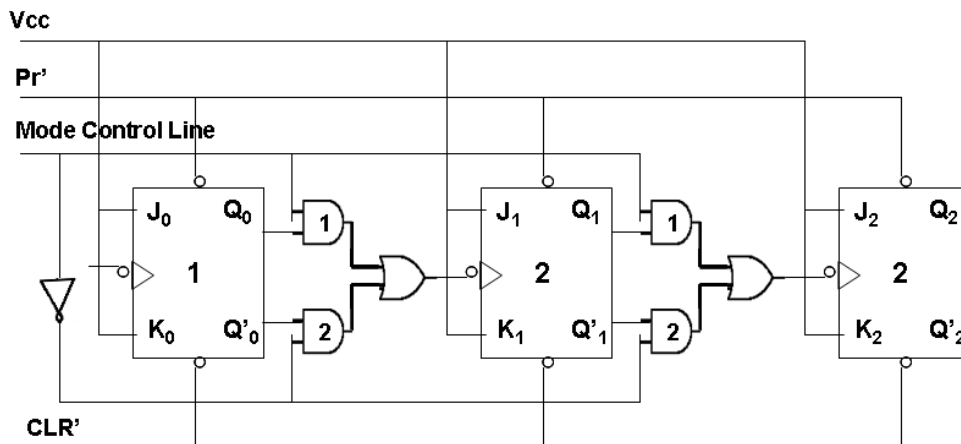
A counter which can be made to count in either the forward or reverse direction is called an up-down, a reversible or forward-backward counter.

DOWN COUNTER

A binary counter with a reverse count is called a binary down counter. In a down counter, the binary counter is decremented by 1 with every input count pulse. The count of a 4-bit down counter starts from binary 15 and continues to binary counts 14, 13, 12... 0 and then back to 15. In a binary down counter, outputs are taken from the complement terminals Q' of all flip flops. For a down counter, when Q goes from 0 to 1, Q' will go from 1 to 0 and complement the next flip flop.

UP COUNTER

A binary counter with a normal count is called a binary up counter. In a up counter, the binary counter is incremented by 1 with every input clock pulse. Outputs are taken from the normal output terminal Q of all flip flops. For a up counter when Q goes from 1 to 0, it complements the next flip flop.



In above diagram, **mode control line is also called up down counter line.**

When mode control line is 1, all gates labeled as 1 will be enabled and all gates labeled as 2 will be disabled. The counter works like a **Up Counter**.

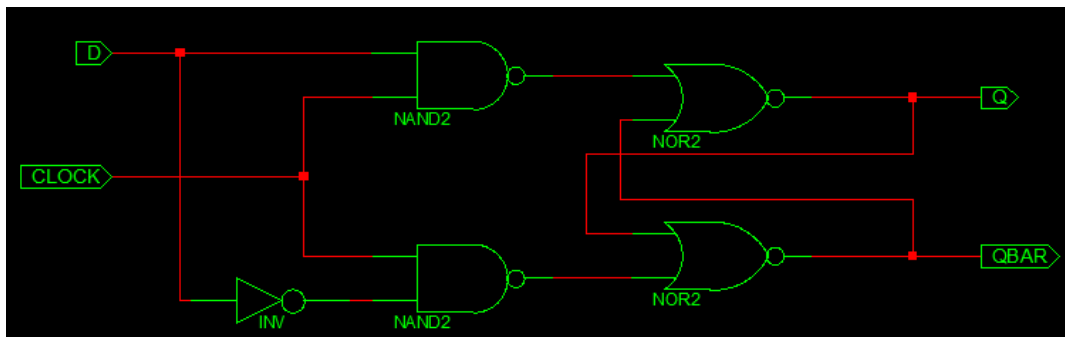
When mode control line is 0, all gates labeled as 1 will be disabled and all gates labels as 1 will be enabled. The counter works like a **Down Counter**.

UNIT – III

CLK	Q2	Q1	Q0	Q0'	Q1'	Q2'
0	0	0	0	1	1	1
1	1	1	1	0	0	0
2	1	1	0	1	0	0
3	1	0	1	0	1	0
4	1	0	0	1	1	0
5	0	1	1	0	0	1
6	0	1	0	1	0	1
7	0	0	1	0	1	1
8	0	0	0	1	1	1

VHDL CODE FOR D FLIP FLOP

The D input goes directly into the S input and the complement of the D input goes to the R input. The D input is sampled during the occurrence of a clock pulse. If it is 1, the flip-flop is switched to the set state (unless it was already set). If it is 0, the flip-flop switches to the clear state.



D FLIPFLOP TRUTH TABLE

Q	D	Q(T+1)
0	0	0
0	1	1
1	0	0
1	1	1

VHDL Code for D FlipFlop

```

library ieee;
use ieee. std_logic_1164.all;
use ieee. std_logic_arith.all;
use ieee. std_logic_unsigned.all;

entity D_FF is
PORT( D,CLOCK: in std_logic;
Q: out std_logic);
end D_FF;

architecture behavioral of D_FF is
begin
process(CLOCK)
begin
if(CLOCK='1' and CLOCK'EVENT) then
Q <= D;
end if;
end process;
end behavioral;
    
```

VHDL CODE FOR UP-COUNTER

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Counter_VHDL is
port( Number: in std_logic_vector(0 to 3);
Clock: in std_logic;
Load: in std_logic;
Reset: in std_logic;
Direction: in std_logic;
Output: out std_logic_vector(0 to 3) );
end Counter_VHDL;

architecture Behavioral of Counter_VHDL is
signal temp: std_logic_vector(0 to 3);
begin
process(Clock,Reset)
begin
if Reset='1' then
temp <= "0000";
elsif ( Clock'event and Clock='1') then
if Load='1' then
temp <= Number;
elsif (Load='0' and Direction='0') then
temp <= temp + 1;
elsif (Load='0' and Direction='1') then
temp <= temp - 1;
end if;
end if;
end process;
Output <= temp;
end Behavioral;

```


UNIT-4

SYNCHRONOUS SEQUENTIAL CIRCUITS: – Basic design steps. Moore and Mealy state model, State minimization, Design of a Counter using the Sequential Circuit Approach. Algorithmic State Machine (ASM) charts

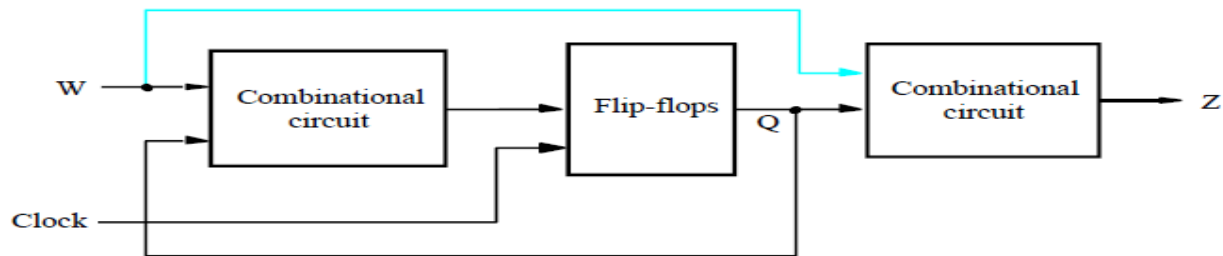
SYNCHRONOUS SEQUENTIAL CIRCUITS

1. It consists of a **combinational circuit (present behavior)** to which **memory elements (past behavior)** are connected via a feedback path.

Combinational circuit= Collection of gates which perform different operations.

Memory element = Collection of Flip Flops used to store data.

2. It extends the capabilities of our systems by including the past behavior of the circuit with its present behavior.



The general form of a sequential circuit.

3. It has a set of inputs, W ; a state, Q ; and a set of outputs, Z .

4. Sequential circuits are of two types

a. SYNCHRONOUS SEQUENTIAL CIRCUIT

A sequential Circuit whose operation is controlled by a clock signal is called a **synchronous sequential circuit**.

b. ASYNCHRONOUS SEQUENTIAL CIRCUITS

A sequential Circuit whose operation is not controlled by a clock signal is called **asynchronous sequential circuits**.

MOORE TYPE MODEL

A sequential circuit where the outputs depend only on the state of the circuit, Q is called as Moore type machine or model

MEALY TYPE MODEL

A sequential circuit where the outputs depends on both the state Q , and the primary inputs, W is called as Mealy type machine or model

5. It is also called a **finite state machine (FSMs)** which is a more formal name that is often found in technical literature.

UNIT-4

BASIC DESIGN STEPS

Design is non thing but the blue print of the system. I.e. here we are going to develop the circuit.

BASIC DESIGN STEPS ARE

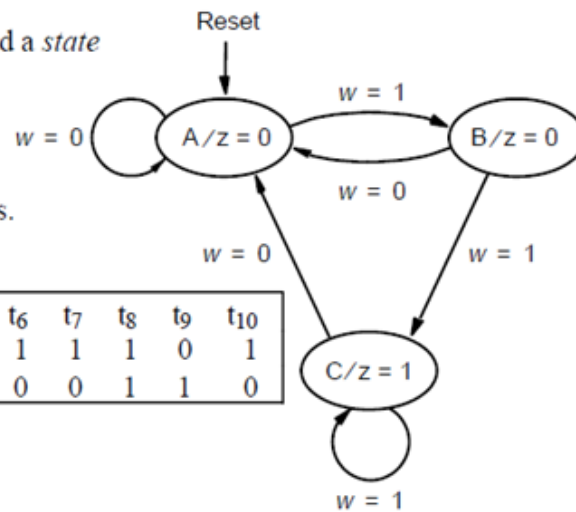
1. Identify the states of the circuit.
2. Construct the state diagram using the states.
3. Construct the state table using the states Assignment table.
4. Choice of flip flops.
5. using of KMAP for generating the equations.
6. Plotting the generated equations in the newly constructed circuit.

States---→State Diagram---→state table---→state Assignment table----→Choice of flip-flop---→Use of KMAP for getting the equations---→ plotting in the circuit.

Suppose that we wish to design a circuit that meets the Synchronous Sequential Circuits following specification:

1. The circuit has one input w , and one output z .
2. All changes occur on the positive edge of the clock.
3. The output z is equal to 1 if during the two immediately preceding clock cycles the input w was equal to 1. Otherwise, z is equal to 0.

- The circuit should remain in state C, as long as $w=1$ and continue to maintain $z=1$.
- When $w=0$, the machine should move back to state A.
- Thus three states are needed to implement the desired machine.
- The simplest representation of this information is in a pictorial form called a *state diagram*.
- A *state diagram* depicts states of the circuit as nodes (circles) and transitions between states as directed arcs.

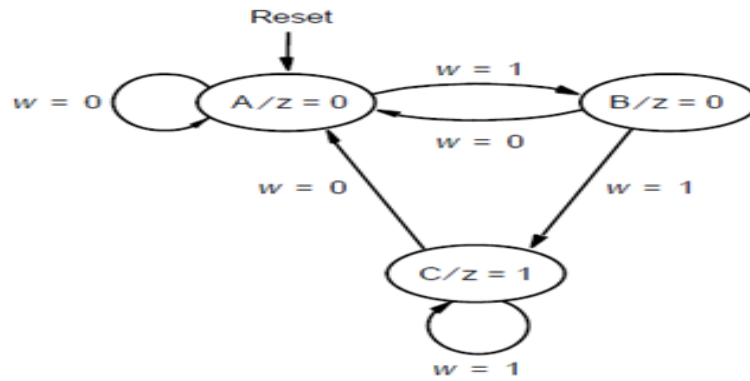


Clockcycle:	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
w :	0	1	0	1	1	0	1	1	1	0	1
z :	0	0	0	0	0	1	0	0	1	1	0

STATE DIAGRAM

Designing a finite state machine is to determine how many states are needed, and what transitions are possible from one state to another.

- One can arbitrarily select one particular state as a starting state; generally, this is the state that the circuit should enter when power is first turned on or when a reset signal is applied.



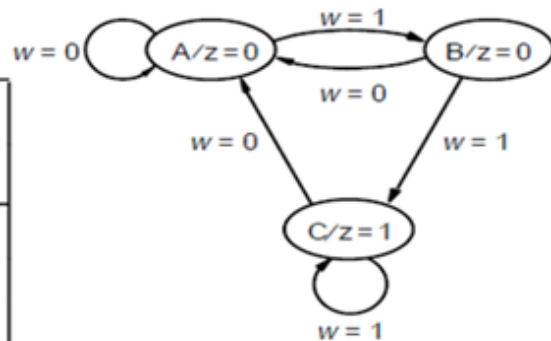
State Diagram

STATE TABLE

State diagram can also be represented in the form of table called as state diagram. State table contains all the information of the state diagram, input signal, and output signals, but places them in a form from which it is easier to simplify and implement a circuit.

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

State table



STATE ASSIGNMENT TABLE

1. Each state in a sequential circuit is represented by a particular valuation (combination of values) of state variables.
2. Each state variable is implemented in the form of a flip flop.
3. Three States can be represented using two state variables.
4. The mapping of state table information produces an assigned state table which, like a truth table, used for the design of a combinational circuit which will generate the correct output signal Z and drives the inputs for the flip-flops.
5. State assigned table is used to generate a combinational logic using techniques such as KMAPS, etc.

Number of states=3 (A, B and C)

$2^1 = 2 < 3$ (states (A, B and C))

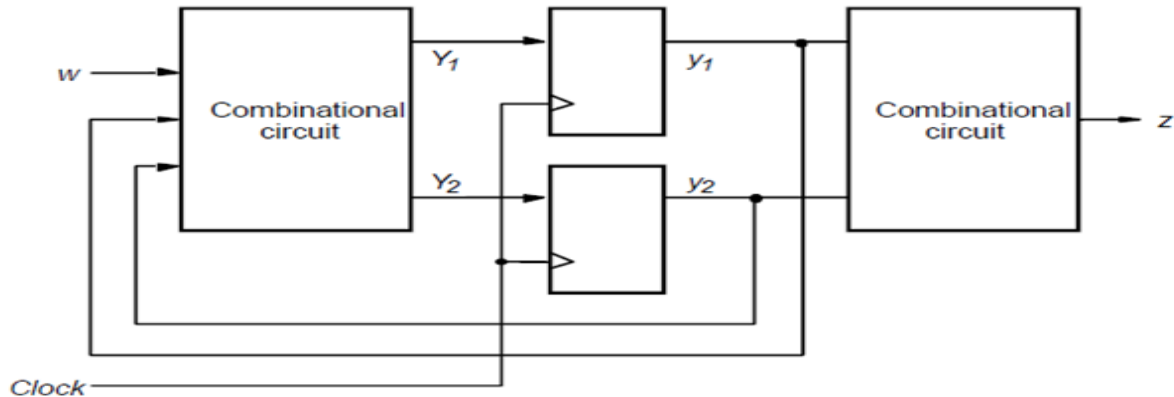
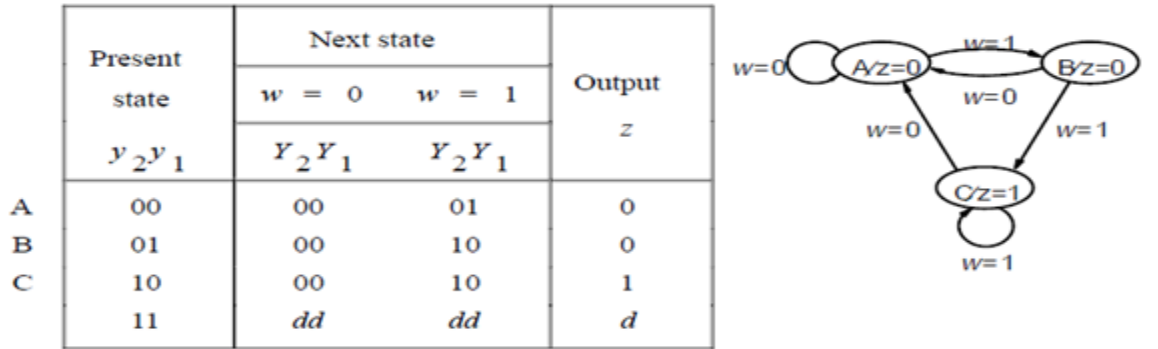
Case failed (1 Flip flop).

$2^2 = 4 > 3$ (states (A, B and C))

Case passed (2 Flip flop).

So take 2 flop flops for constructing a combinational circuit.

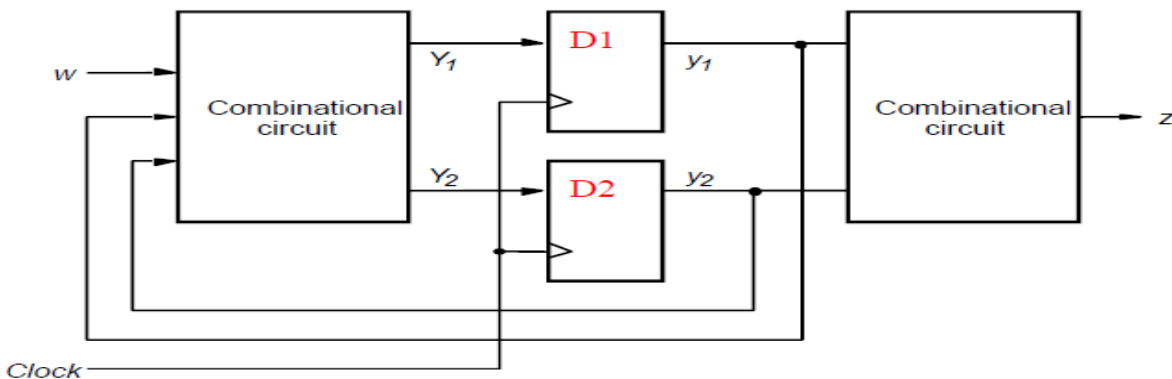
UNIT-4



A general sequential circuit with input w , output z , and two state flip-flops.

CHOICE OF FLIP-FLOPS

1. Flip Flops are memory elements which maintain "state" and therefore necessary to decide which type will be used in any implementation.
2. the most straightforward choice is to use D-type flip-flops, since the values for the next state are simply clocked into the flip-flops to become the new current state values.
3. Regardless of the type of flip-flop selected to be used, the next state columns of the assigned state table represent what needs to be stored in the flip-flops and thus each individual column can be thought of as an individual function. If the inputs to the two flip-flops are called D1 and D2, then these signals are the same as Y1 and Y2.



UNIT-4

Y1 Karnaugh Map:

$y_2 y_1$	00	01	11	10
w	0	0	d	0
1	1	0	d	0

Ignoring don't cares: $Y_1 = w\bar{y}_1\bar{y}_2$
 Using don't cares: $Y_1 = w\bar{y}_1\bar{y}_2$

Present state	Next state		Output z
	w = 0	w = 1	
$y_2 y_1$	$Y_2 Y_1$	$Y_2 Y_1$	
A 00	00	01	0
B 01	00	10	0
C 10	00	10	1
11	d d	d d	d

Y2 Karnaugh Map:

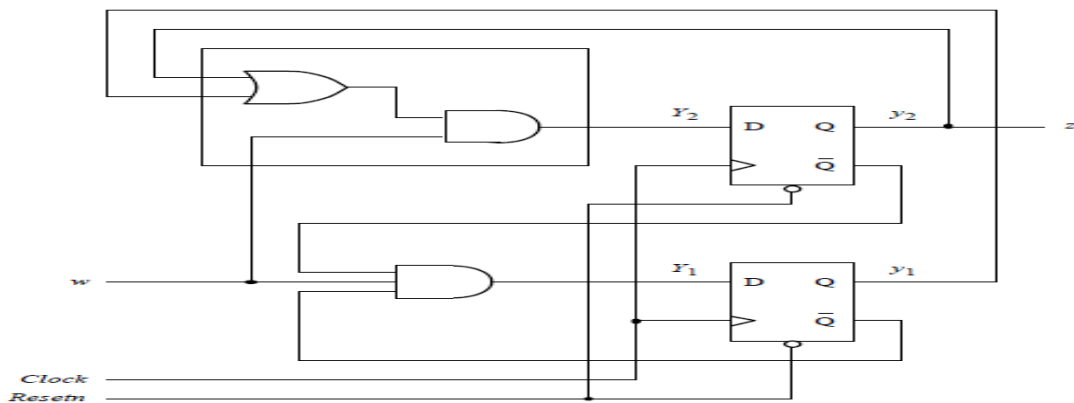
$y_2 y_1$	00	01	11	10
w	0	0	d	0
1	0	1	d	1

Ignoring don't cares: $Y_2 = w y_1 \bar{y}_2 + w \bar{y}_1 y_2$
 Using don't cares: $Y_2 = w y_1 + w y_2 = w(y_1 + y_2)$

z Karnaugh Map:

y_2	0	1
y_1	0	0
1	1	d

Ignoring don't cares: $z = \bar{y}_1 y_2$
 Using don't cares: $z = y_2$



Final implementation of the sequential circuit

STATE MINIMIZATION

1. FSM requires number of states which will be needed to implement the desired machine.
2. Minimizing the number states of requires a fewer flip-flops so the complexity of the combinational circuit will be reduced.
3. In the FSM **equivalent states or redundant states or duplicate states** will be there. we remove the equivalent states by using **Partitioning Minimization Procedure or partitioning method**.
4. A partition consists of one or more blocks, where each block comprises a subset may be equivalent but the states in a given block are Synchronous Sequential Circuits of states that equivalent, definitely not equivalent to the states in the other blocks.
5. The partitioning method initially assumes that all states are equivalent and then proceeds to determine those states which are not equivalent by analyzing each states k-successors.
6. **Equivalent States or State equivalence:** Two states S_i and S_j are said to be equivalent if and only if for every possible input sequence, the same output sequence will be produced regardless of whether S_i or S_j is the initial state.

UNIT-4

Partition minimization example: Consider the following state table

Present State	Next State		Output
	W=0	W=1	
A	B	C	1
B	D	F	1
C	F	E	0
D	B	G	1
E	F	C	0
F	E	D	0
G	F	G	0

PARTITION METHOD

P1=(ABCDEFG)	Take all states to partition one (p1).	
P2=(ABD)(CEFG)	Create groups depends on output. Z=0(ABD) z=1(CEFG)	
P3=(ABD)(CEG)(F)	<p>W=0 for (ABD)</p> <p>A-----→B B-----→D D-----→B</p> <p>All are in the same block so no further partitions are possible.</p> <p>W=1 for (ABD)</p> <p>A-----→C B-----→F D-----→G</p> <p>All are in the same block so no further partitions are possible.</p>	<p>W=0 for (CEFG)</p> <p>C-----→E E-----→F F-----→F G-----→F</p> <p>All are in the same block so no further partitions are possible.</p> <p>W=1 for (CEFG)</p> <p>C-----→E E-----→C F-----→D G-----→G</p> <p>All are NOT in the same block so further partitions are possible. I.E (CEG)(F)</p>
P4=(ABD)(CEG)(F)	<p>W=0 for (ABD)</p> <p>A-----→B B-----→D D-----→B</p> <p>All are in the same block so no further partitions are possible.</p> <p>W=1 for (ABD)</p> <p>A-----→C B-----→F D-----→G</p> <p>All are NOT in the same block so further partitions are possible. I.E (AD)(B)</p>	<p>W=0 for (CEG)</p> <p>C-----→F E-----→F G-----→F</p> <p>All are in the same block so no further partitions are possible.</p> <p>W=1 for (CEG)</p> <p>C-----→E E-----→C G-----→G</p> <p>All are NOT in the same block so further partitions are possible. I.E (CEG)(F)</p>
P5=(AD)(B)(CEG)(F)	<p>W=0 for (AD)</p> <p>A-----→B D-----→B</p>	<p>W=0 for (CEG)</p> <p>C-----→F E-----→F</p>
P4=P5		

UNIT-4

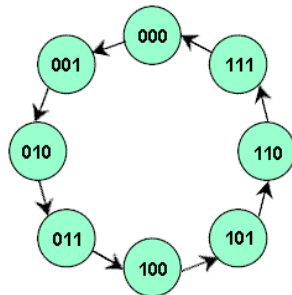
So A=D B C=E=G F	All are in the same block so no further partitions are possible. W=1 for (AD) A-----→C D-----→G All are NOT in the same block so no further partitions are possible.	G-----→F All are in the same block so no further partitions are possible. W=1 for (CEG) C-----→E E-----→C G-----→G All are in the same block so NO further partitions are possible.
---------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

THE RESULTING STATE TABLE IS AS FOLLOWS

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

DESIGN OF A COUNTER USING THE SEQUENTIAL CIRCUIT APPROACH

Design a counter specified by the state diagram using T flip-flops.



State diagram of a 3-bit binary counter (Represent 0-7 Numbers in binary form). Now derive the excitation table from the state diagram, which is shown in Table

EXCITATION TABLE

Output Transitions								Inputs			
	Present State				Next State				T2	T1	T0
	Q2	Q1	Q0		Q2	Q1	Q0		Q2	Q1	Q0
0	0	0	0	1	0	0	1	1	0	0	1
1	0	0	1	2	0	1	0	3	0	1	1
2	0	1	0	3	0	1	1	1	0	0	1
3	0	1	1	4	1	0	0	7	1	1	1
4	1	0	0	5	1	0	1	1	0	0	1
5	1	0	1	6	1	1	0	3	0	1	1
6	1	1	0	7	1	1	1	1	0	0	1
7	1	1	1	0	0	0	0	7	1	1	1

UNIT-4

Next step is to transfer the flip-flop input functions to Karnaugh maps to derive a simplified Boolean expressions, which is shown in Figure

	Q0	0	1
Q2Q1	00	0	0
01	0	1	
11	0	1	
10	0	0	

	Q0	0	1
Q2Q1	00	0	1
01	0	1	
11	0	1	
10	0	1	

	Q0	0	1
Q2Q1	00	1	1
01	1	1	
11	1	1	
10	1	1	

Karnaugh maps

T2 map **T1 map** **T0 map**

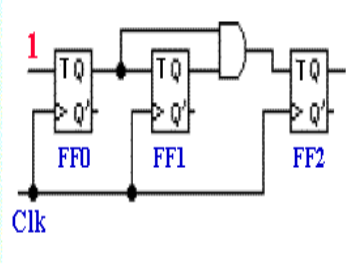
T2 values have to be taken in the cells
T1 values have to be taken in the cells
T0 values have to be taken in the cells

The following expressions are obtained:

$T0 = 1;$ $T1 = Q0;$ $T2 = Q1 * Q0$

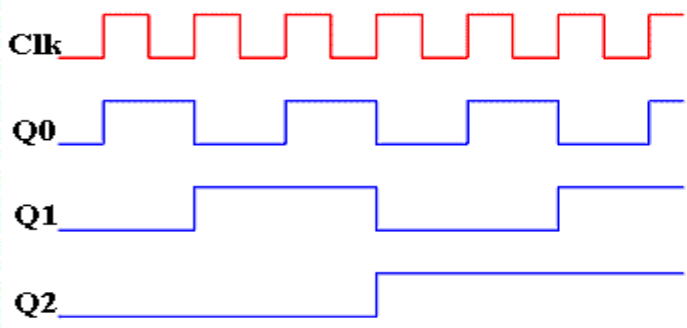
You have to plot in the table
Present State <-----> Inputs

Finally, draw the logic diagram of the circuit from the expressions obtained. The complete logic diagram of the counter is shown in Figure

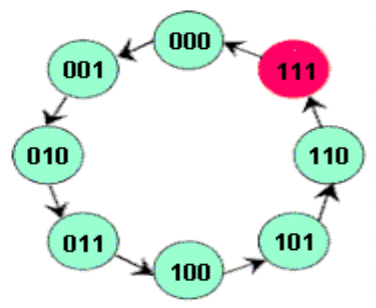


Logic diagram of 3-bit binary counter.

Timing behaviour and state transition of a 3-bit binary counter.



Timing Diagram



State Transition

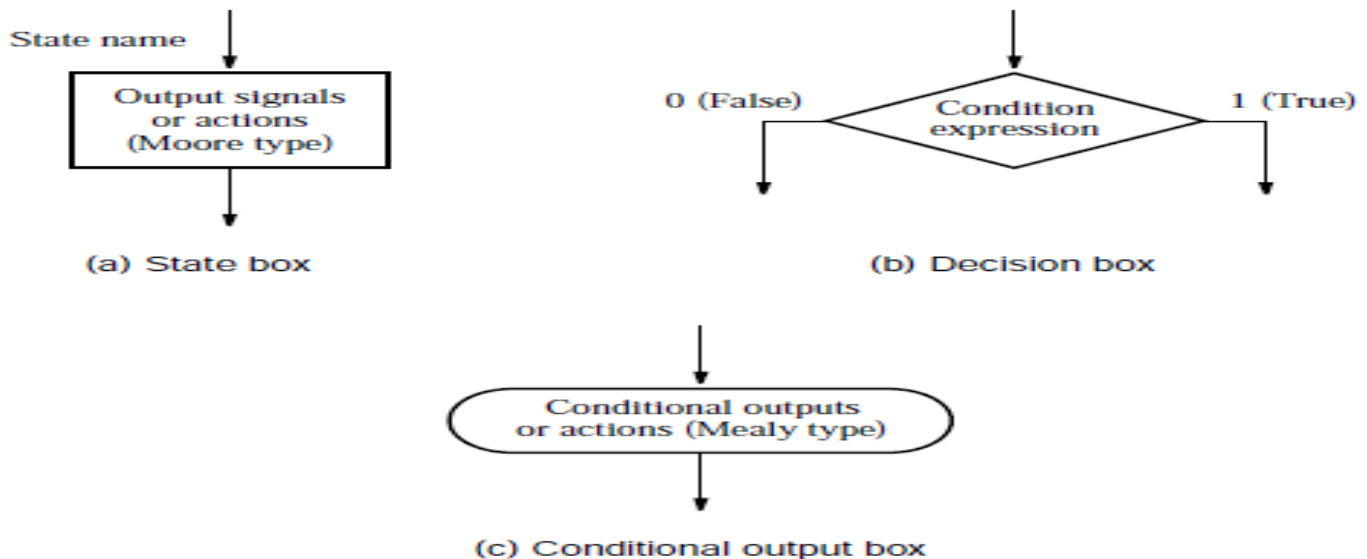
Notice the state transition which corresponds to the clock pulse.

UNIT-4

ALGORITHMIC STATE MACHINE (ASM)

1. The state diagrams & State tables describe FSM's behavior with only a few inputs & outputs.
2. For larger machines designers often use a different form of representation called ASM chart.
3. An ASM chart (type of flowchart) used to represent the state transitions & outputs for an FSM.
4. ASM charts Elements: a. **State box.** b. **Decision box.** C. **Conditional output box.**

SYNCHRONOUS SEQUENTIAL CIRCUITS



Elements used in ASM charts.

STATE BOX

1. A rectangle represents a state of the FSM.
2. It is equivalent to a node in the state diagram or a row in the state table.
3. The name of the state is indicated outside the box in the top-left corner.
4. The Moore-type outputs are listed inside the box.
5. These are the outputs that depend only on the values of the state variables that define the state; we will refer to them simply as Moore outputs. It is customary to write only the name of the signal that has to be asserted. Thus it is sufficient to write z , rather than $z = 1$, to indicate that the output z must have the value 1. Also, it may be useful to indicate an action that must be taken; for example, $\text{Count} \leftarrow \text{Count} + 1$ specifies that the contents of a counter have to be incremented by 1. Of course, this is just a simple way of saying that the control signal that causes the counter to be incremented must be asserted.

DECISION BOX

1. A diamond indicates that the stated condition expression is to be tested and the exit path is to be chosen accordingly.
2. The condition expression consists of one or more inputs to the FSM.

UNIT-4

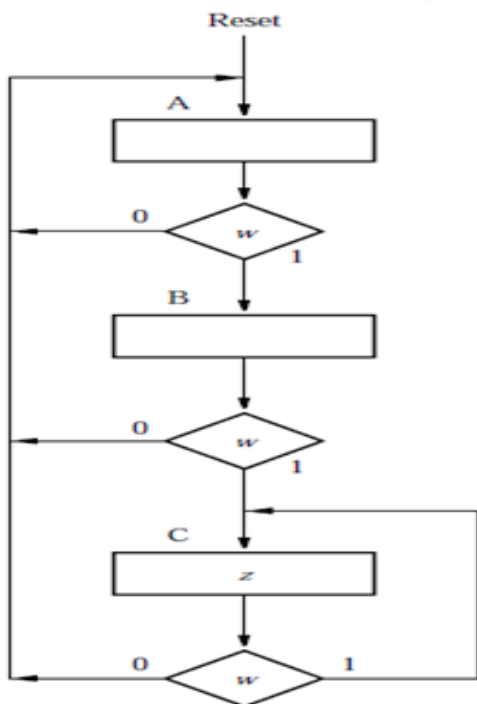
3. For example, w indicates that the decision is based on the value of the input w , whereas $w_1 \cdot w_2$ indicates that the true path is taken if $w_1 = w_2 = 1$ and the false path is taken otherwise.

CONDITIONAL OUTPUT BOX

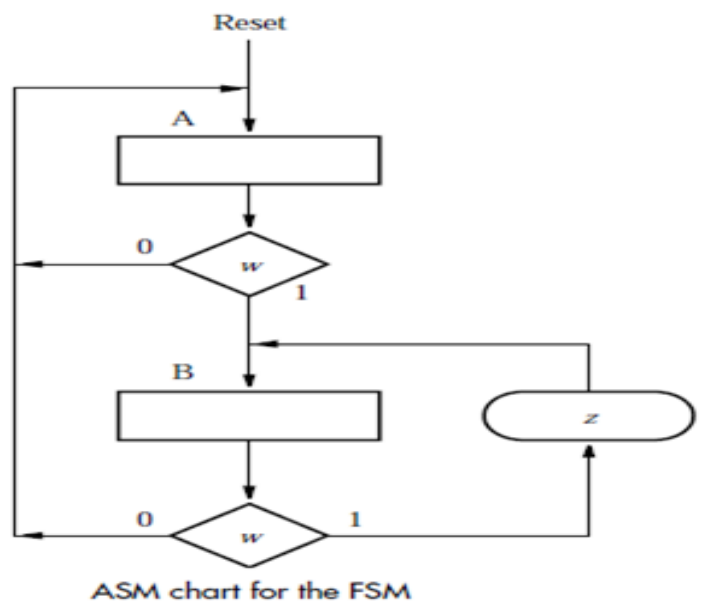
1. An oval denotes the output signals that are of Mealy type.
2. These outputs depend on the values of the state variables and the inputs of the FSM. We will refer to these outputs simply as Mealy outputs.
3. The condition that determines whether such outputs are generated is specified in a decision box.

Example:

ALGORITHMIC STATE MACHINE (ASM) CHARTS



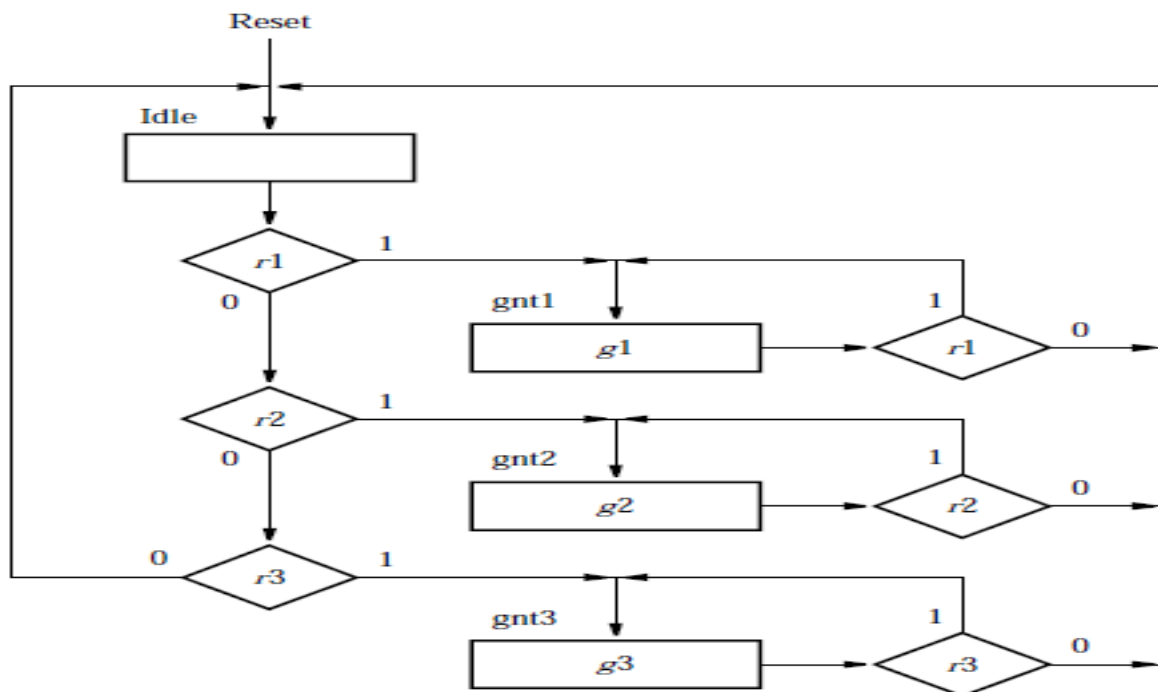
SYNCHRONOUS SEQUENTIAL CIRCUITS



1. The above figure (ASM chart) provides an example with Mealy outputs that represents the FSM. The transitions between state boxes depend on the decisions made by testing the value of the input variable w .
2. In each case if $w = 0$, the exit path from a decision box leads to state A .
3. If $w = 1$, then a transition from A to B or from B to C takes place.
4. If $w = 1$ in state C , then the FSM stays in that state.
5. The chart specifies a Moore output z , which is asserted only in state C , as indicated in the state box. In states A and B , the value of z is 0 (not asserted), which is implied by leaving the corresponding state boxes blank.
6. The output, z , is equal to 1 when the machine is in state B and $w = 1$. This is indicated using the conditional output box. In all other cases the value of z is 0, which is implied by not specifying z as an output of state B for $w = 0$ and state A for w equal to 0 or 1.

UNIT-4

1. The above figure (ASM chart) provides an example for the arbiter FSM.
2. The decision box drawn below the state box for *Idle* specifies that
If $r1 = 1$, then the FSM changes to state *gnt1*.
3. In this state the FSM asserts the output signal $g1$.
4. The decision box to the right of the state box for *gnt1* specifies that as long as $r1 = 1$, the machine stays in state *gnt1*, and when $r1 = 0$, it changes to state *Idle*.
5. The decision box labeled $r2$ that is drawn below the state box for *Idle* specifies that if $r2 = 1$, then the FSM changes to state *gnt2*. This decision box can be reached only after first checking the value of $r1$ and following the arrow that corresponds to $r1 = 0$.
6. Similarly, the decision box labeled $r3$ can be reached only if both $r1$ and $r2$ have the value 0. Hence the ASM chart describes the required priority scheme for the arbiter.



ASM chart for the arbiter FSM

UNIT-5

Asynchronous Sequential Circuits: – Behavior, Analysis, Synthesis, State reduction, State Assignment, examples. Hazards: static and dynamic hazards. Significance of Hazards. Clock skew, set up and hold time of a flip-flop

ASYNCHRONOUS SEQUENTIAL CIRCUITS

Synchronous sequential circuits have

1. State variables: F/Fs.
2. Controlled by a clock.
3. Operate in pulse mode.

ASYNCHRONOUS SEQUENTIAL CIRCUITS

1. Do not operate in synchronous with clock signal.
2. Do not use F/Fs to represent state variables.
3. Changes in state are dependent on whether each of inputs to the circuit has the logic level 0 or 1 at any given time.
4. Hazards that cause incorrect behavior of a circuit.

TO ACHIEVE RELIABLE OPERATION

1. The inputs to the circuit must change one at a time.
2. There must be sufficient time between the changes in input signals to allow the circuit to reach a stable state.
3. A circuit that adheres to these constraints is said to operate in the fundamental mode.

ADVANTAGES OF ASYNCHRONOUS SEQUENTIAL CIRCUITS

1. No clock skew (clock signal arrives at different time).
2. Lower power (Synchronous: clock signal must be present every time and everywhere).
3. Average-case performance VS worst case performance.
4. Easing of global timing issues.
5. Partial optimization.
6. Better external input handling.

DRAW BACKS

1. More difficult to design.
2. Concerns for hazards and glitches.
3. Unsure about faster performance.

WHY ASYNCHRONOUS SEQUENTIAL CIRCUITS?

1. Used when speed of operation is important
2. Response quickly without waiting for a clock pulse
3. Used in small independent systems
4. Only a few components are required
5. Used when the input signals may change independently of internal clock
5. Asynchronous in nature
6. Used in the communication between two units that has their own independent clocks.
7. Must be done in an asynchronous fashion.

ASYNCHRONOUS SEQUENTIAL CIRCUITS OPERATION MODES

1. Steady-state condition.
2. Fundamental mode.
3. Pulse Mode.

UNIT-5

STEADY-STATE CONDITION

- Current states and next states are the same
- Difference between Y and y will cause a transition

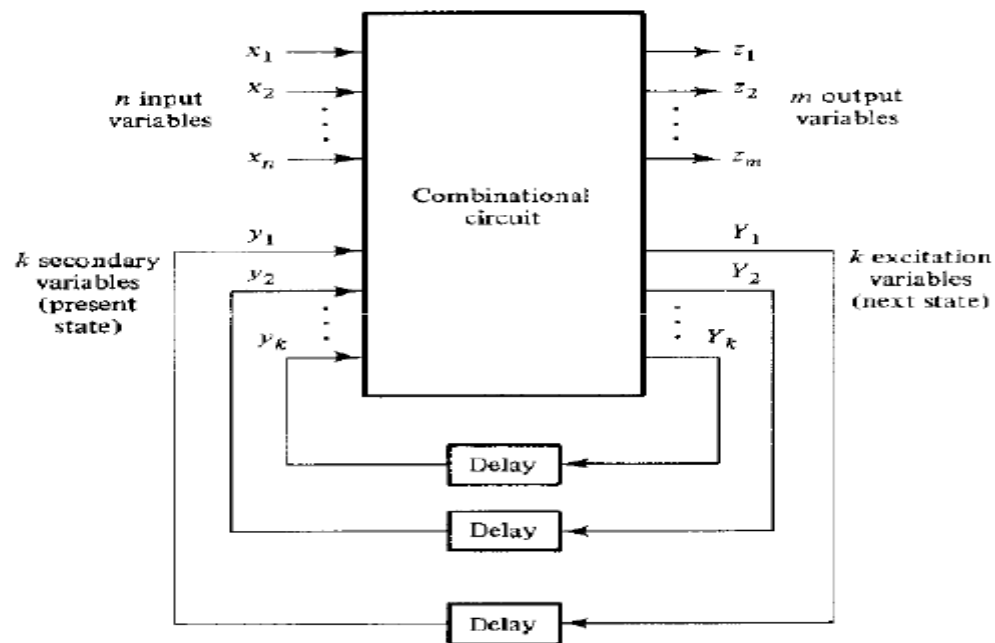
FUNDAMENTAL MODE

- No simultaneous changes of two or more variables.
 - The time between two input changes must be longer than the time it takes the circuit to a stable state
 - The input signals change one at a time and only when the circuit is in a stable condition
- Fundamental Mode

PULSE MODE

- The inputs and outputs are represented by pulses.
- Only one input is allowed to have pulse present at any time.
- Similar to synchronous sequential circuits except without a clock signal.

GENERAL BLOCK DIAGRAM



BEHAVIOR

The action or reaction of something (as a machine or substance) under specified circumstances.

ANALYSIS

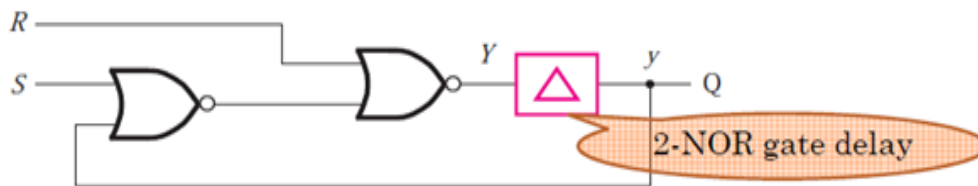
- Detailed examination of the elements or structure of something, typically as a basis for discussion or interpretation.
- The process of separating something into its constituent elements. **Or**
 - From the logic diagram, determine input, output, and state variable.
 - Derive Boolean functions.
 - Obtain excitation table, flow table, state diagram.

UNIT-5

STEPS IN THE IN ANALYSIS PROCESS

1. Each feedback path is cut.
 - a. A delay element is inserted at the point where the cut is made.
 - b. A cut can be made anywhere in a particular loop formed by feedback connection, as long as there is only one cut per (state variable) loop.
2. Next-state and output expressions are derived from the circuit.
3. The excitation table is derived.
4. A flow table is obtained.
5. A corresponding state diagram is derived from the flow table if desired.
- 6. In analysis we will identify stable states.**
- 7. A stable state is one where present state is equal to next state. I.e. $y=Y$.**

ASYNCHRONOUS BEHAVIOR OF SR-LATCH



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

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

Stable state: given inputs, if a circuit reaches a state and remains in that state, then the state is said to be “stable”.

Present state y	Next state			
	$SR = 00$	01	10	11
	Y	Y	Y	Y
0	0	0	1	0
1	1	0	1	0

stable state

(b) State-assigned table

Circles denote “stable” states, i.e., state “unchanged”.

SYNTHESIS OF ASYNCHRONOUS CIRCUITS (DESIGN)

1. Determine state diagram and primitive flow table.
2. Reduce states if possible.
3. Assign states and obtain excitation table.
4. Derive Boolean functions and design circuit. **Or**

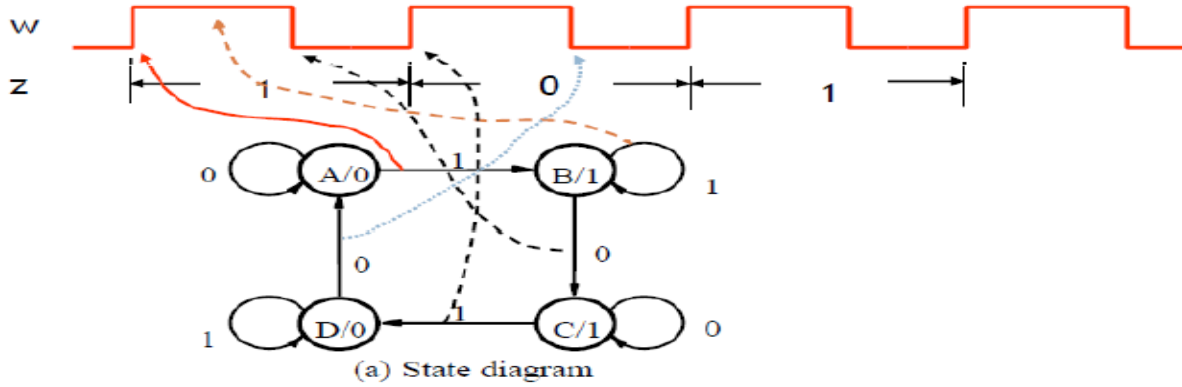
THE SAME BASIC STEPS USED TO SYNTHESIZE THE SYNCHRONOUS CIRCUITS

1. Devise a state diagram for an FSM.
2. Derive the flow table and reduce the number of states if Possible.
3. Perform the state assignment and derive the excitation table.
4. Obtain the next-state and output expressions.
5. Construct a circuit that implements these expressions Reverse of Analysis.

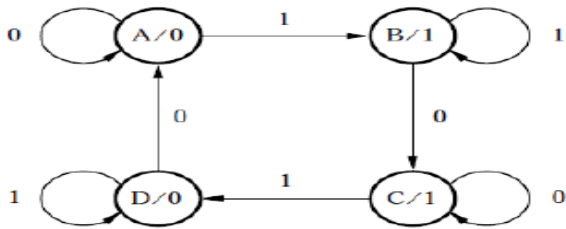
UNIT-5

EXAMPLE: SERIAL PARITY GENERATOR

1. Input w : pulses are applied to w .
2. Output z .
3. $Z=1$ if the number of previously applied pulses is odd.



EXAMPLE: SERIAL PARITY GENERATOR (2)



Present state $y_2 y_1$	Next state		Output z
	$w = 0$	$w = 1$	
00	00	01	0
01	10	01	1
10	10	11	1
11	00	11	0

(a) Poor state assignment

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

(b) Flow table

Present state $y_2 y_1$	Next state		Output z
	$w = 0$	$w = 1$	
00	00	01	0
01	11	01	1
11	11	10	1
10	00	10	0

(b) Good state assignment

STATE ASSIGNMENT

Present state $y_2 y_1$	Next state		Output z
	$w = 0$	$w = 1$	
00	00	01	0
01	10	01	1
10	10	11	1
11	00	11	0

(a) Poor state assignment

Present state $y_2 y_1$	Next state		Output z
	$w = 0$	$w = 1$	
00	00	01	0
01	11	01	1
11	11	10	1
10	00	10	0

(b) Good state assignment

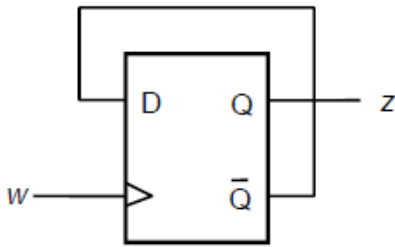
- ❖ State assignment (a) has a major flaw
- ❖ state D = 11 : $w=0 \rightarrow$ state A
- ❖ $y_2 y_1 = 11 \rightarrow y_2 y_1 = 00$
- ❖ the values of the next-state variables determined by the networks of logic gates with varying delays
 - suppose y_1 changes first
 - ❖ $y_2 y_1 = 10 \rightarrow$ state C(10)
 - ❖ state C is stable when $w=0$
 - suppose y_2 changes first
 - ❖ $y_2 y_1 = 01 \rightarrow$ state B (01)
 - ❖ try to change to $y_2 y_1 = 10$ when $w=0$
 - ❖ if y_1 changes first, $y_2 y_1 = 00$
- *race condition occurs*

Circuit that implements the FSM

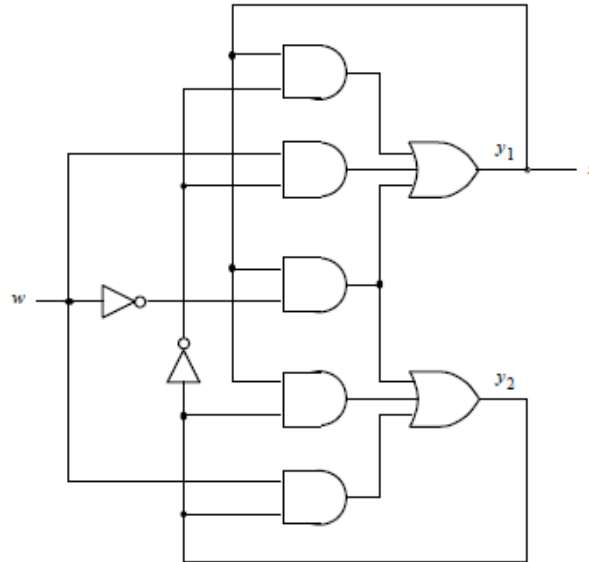
$$Y_1 = w\bar{y}_2 + \bar{w}y_1 + y_1\bar{y}_2$$

$$Y_2 = wy_2 + \bar{w}y_1 + y_1y_2$$

$$z = y_1$$



Synchronous solution



Asynchronous solution

STATE REDUCTION (We use Partitioning method for state reduction.)

UNIT 4	UNIT 5
State Minimization For synchronous circuits with clock	State Reduction For A synchronous circuits without clock
Partition Method	Partition Method
Equivalent States	Compatible States

1. FSM requires number of states which will be needed to implement the desired machine.
2. Minimizing the number states of requires a fewer flip-flops so the complexity of the combinational circuit will be reduced.
3. In the FSM **Compatible states or redundant states or duplicate states** will be there. we remove the equivalent states by using **Partitioning Minimization Procedure or partitioning method**.
4. A partition consists of one or more blocks, where each block comprises a subset may be equivalent but the states in a given block are Synchronous Sequential Circuits of states that equivalent, definitely not equivalent to the states in the other blocks.
5. The partitioning method initially assumes that all states are equivalent and then proceeds to determine those states which are not equivalent by analyzing each states k-successors.
6. **COMPATIBLE STATES:** Two states S_i and S_j are said to be equivalent if and only if for every possible input sequence, the same output sequence will be produced regardless of whether S_i or S_j is the initial state.

UNIT-5

PARTITION MINIMIZATION EXAMPLE: CONSIDER THE FOLLOWING STATE TABLE

Present State	Next State		Output
	W=0	W=1	
A	B	C	1
B	D	F	1
C	F	E	0
D	B	G	1
E	F	C	0
F	E	D	0
G	F	G	0

PARTITION METHOD

P1=(ABCDEFG)	Take all states to partition one (p1).	
P2=(ABD)(CEFG)	Create groups depends on output. Z=0(ABD) z=1(CEFG)	
P3=(ABD)(CEG)(F)	<p>W=0 for (ABD)</p> <p>A-----→B B-----→D D-----→B</p> <p>All are in the same block so no further partitions are possible.</p> <p>W=1 for (ABD)</p> <p>A-----→C B-----→F D-----→G</p> <p>All are in the same block so no further partitions are possible.</p>	<p>W=0 for (CEFG)</p> <p>C-----→E E-----→F F-----→F G-----→F</p> <p>All are in the same block so no further partitions are possible.</p> <p>W=1 for (CEFG)</p> <p>C-----→E E-----→C F-----→D G-----→G</p> <p>All are NOT in the same block so further partitions are possible. I.E (CEG)(F)</p>
P4=(ABD)(CEG)(F)	<p>W=0 for (ABD)</p> <p>A-----→B B-----→D</p>	<p>W=0 for (CEG)</p> <p>C-----→F E-----→F</p>

UNIT-5

	<p>D-----→B All are in the same block so no further partitions are possible.</p> <p>W=1for (ABD)</p> <p>A-----→C B-----→F D-----→G</p> <p>All are NOT in the same block so further partitions are possible. I.E (AD)(B)</p>	<p>G-----→F All are in the same block so no further partitions are possible.</p> <p>W=1 for (CEG)</p> <p>C-----→E E-----→C G-----→G</p> <p>All are NOT in the same block so further partitions are possible. I.E (CEG)(F)</p>
<p>P5=(AD)(B)(CEG)(F)</p> <p>P4=P5 So A=D B C=E=G F</p>	<p>W=0 for (AD)</p> <p>A-----→B D-----→B</p> <p>All are in the same block so no further partitions are possible.</p> <p>W=1for (AD)</p> <p>A-----→C D-----→G</p> <p>All are NOT in the same block so no further partitions are possible.</p>	<p>W=0 for (CEG)</p> <p>C-----→F E-----→F G-----→F</p> <p>All are in the same block so no further partitions are possible.</p> <p>W=1 for (CEG)</p> <p>C-----→E E-----→C G-----→G</p> <p>All are in the same block so NO further partitions are possible.</p>

THE RESULTING STATE TABLE IS AS FOLLOWS

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

STATE ASSIGNMENT

It is a process where for each entry in the state table we will give a unique binary value.

UNIT-5

Present State	Next state		Output z
	$w = 0$	$w = 1$	
A	(A)	B	0
B	C	(B)	1
C	(C)	D	1
D	A	(D)	0

In STATE TABLE Replace A with 00
 B with 01
 C with 11
 D with 10

Then the new table is called as STATE ASSIGNMENT TABLE.

State Table

Present state $y_2 y_1$	Next state		Output z
	$w = 0$	$w = 1$	
00	(00)	01	0
01	11	(01)	1
11	(11)	10	1
10	00	(10)	0

State Assignment Table

Using STATE TABLE, STATE ASSIGNMENT TABLE is generated.

HAZARDS

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

3. TWO TYPES OF HAZARDS

- a. Static.
- b. Dynamic.
- c. Function hazards.

STATIC

A static hazard is the situation where, when one input variable changes, the output changes momentarily before stabilizing to the correct value.

TYPES OF STATIC HAZARDS

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

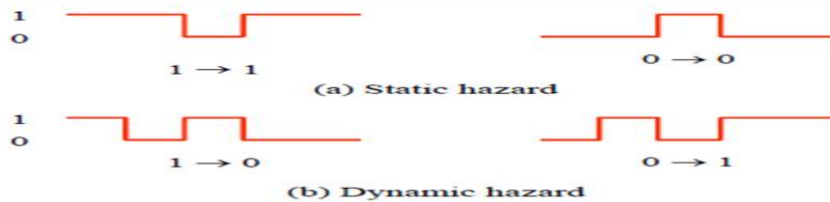
DYNAMIC

UNIT-5

It is the possibility of an output changing more than once as a result of a single input change. Dynamic hazards often occur in larger logic circuits where there are different routes to the output (from the input). If each route has a different delay, then it quickly becomes clear that there is the potential for changing output values that differ from the required / expected output. e.g. A logic circuit is meant to change output state from 1 to 0, but instead changes from 1 to 0 then 1 and finally rests at the correct value 0. This is a dynamic hazard.

As a rule, dynamic hazards are more complex to resolve, but note that if all static hazards have been eliminated from a circuit, then dynamic hazards cannot occur.

Definition of hazards



FUNCTION HAZARDS

These are a non-solvable hazard which occurs when more than one input variable changes at the same time. Hazards such as function hazards cannot be logically eliminated as the problem lies with actual specification of the circuit. The only real way to avoid such problems is to restrict the changing of input variables so that only one input should change at any given time.

HAZARDS AND GLITCHES USUAL SOLUTIONS

1. Wait until signals are stable by using a clock.
 - a. preferable
 - b. easiest to design when there is a clock
 - c. Synchronous circuits.
2. Design hazard-free circuits
 - a. sometimes necessary
 - b. asynchronous design

HAZARDS SIGNIFICANCE

WHY DO WE CARE ABOUT HAZARDS?

COMBINATIONAL NETWORKS

- a. Don't care – the network will function correctly

SYNCHRONOUS SEQUENTIAL NETWORKS

- a. don't care - the input signals must be stable within setup and hold time of flip-flops .
- b. Period between clock edges allows hazards to settle.

ASYNCHRONOUS SEQUENTIAL NETWORKS

- a. Hazards can cause the network to enter an incorrect state

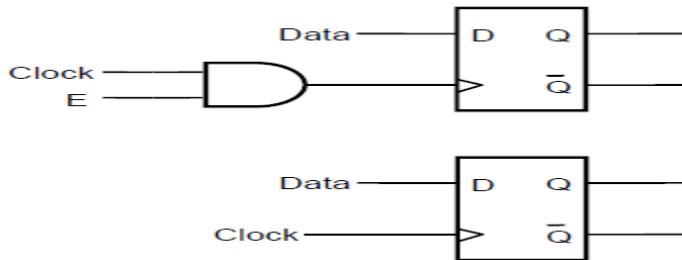
UNIT-5

- b. Circuitry that generates the next-state variables must be hazard-free Power consumption is proportional to the number of transitions.

CLOCK SKEW

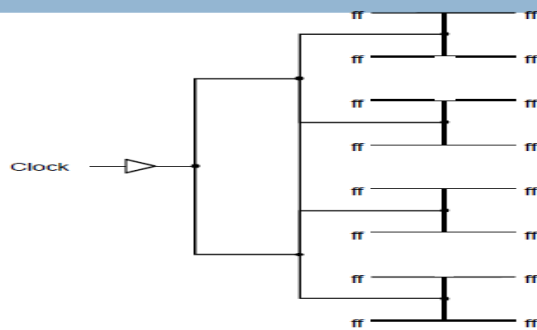
The clock signal arrives at different times at different F/F is called as clock skew.

- a. With or without clock enable circuits.
b. wires whose lengths vary appreciably.



For to remove clock skew we use H-tree clock distribution network. This is called as **CLOCK SYNCHRONIZATION Network**

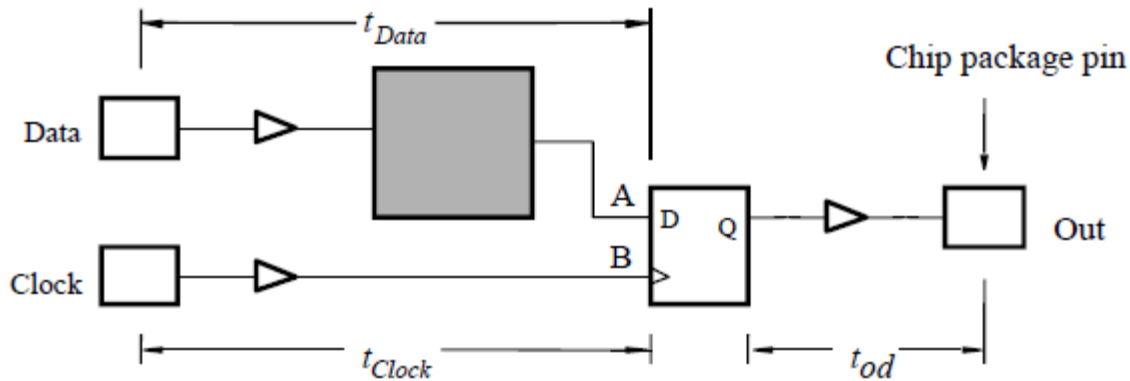
An H-tree clock distribution network



CLOCK SKEW DEPENDS ON F/F TIMING PARAMETERS

1. Setup time t_{su} .
 2. Hold time t_{th} .
 3. Register delay or propagation delay t_{rd} .
 4. Output delay time t_{od} .
- a. required for the change in Q to propagate to an output pin on the chip.

UNIT-5



A flip-flop in an integrated circuit

SETUP TIME

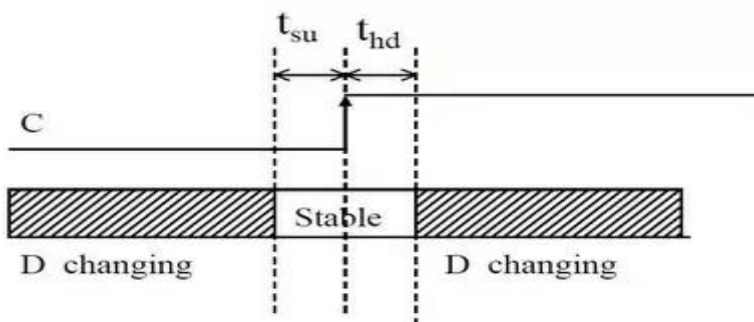
It is the minimum amount of time the data signal should be held steady before the clock event so that the data are reliably sampled by the clock. This applies to synchronous circuits such as the flip-flop. **Or**

The amount of time the Synchronous input (D) must be stable before the active edge of the Clock is called as setup time. The Time when input data is available and stable before the clock pulse is applied is called Setup time.

HOLD TIME

It is the minimum amount of time the data signal should be held steady after the clock event so that the data are reliably sampled. This applies to synchronous circuits such as the flip-flop. **Or** It is the amount of time the synchronous input (D) must be stable after the active edge of clock. The Time after clock pulse where data input is held stable is called hold time.

Setup, Hold Time



SETUP AND HOLD VIOLATION

If Setup time is T_s for a flip-flop and if data is not stable before T_s time from active edge of the clock, there is a Setup violation at that flip flop. So if data is changing in the non-shaded area (in the above figure) before active clock edge, then it's a Setup violation.

And If hold time is T_h for a flip flop and if data is not stable after T_h time from active edge of the clock, there is a hold violation at that flip flop. So if data is changing in the non-shaded area (in the above figure) after active clock edge, then it's a Hold violation.