GOVERNMENT OF TAMILNADU
DIRECTORATE OF TECHNICAL EDUCATION
CHENNAI – 600 025

STATE PROJECT COORDINATION UNIT

Diploma in Electronics and Communication Engineering

Course Code: 1040

M – Scheme

e-TEXTBOOK
for
VERY LARGE SCALE INTEGRATION
for
V Semester DECE

Convener for ECE Discipline:
Dr.M.JeganMohan M.E., MBA., Ph.D.,(Management), Ph.D.,(Engg)., M.I.S.T.E.,
Principal,
138, Government Polytechnic College,
Uthappanaickanoor,
Usilampatti, Madurai – 625 537.

Team Members for Very Large Scale Integration:

Er. M. K. Srinivasan, **M.E., M.I.S.T.E.,**
HOD / ECE,
Pattukkottai Polytechnic College,
Pattukkottai - 614 601.

Mrs. S. Kalaivani, M.E.,
Lecturer / ECE,
Government Polytechnic College,
Kottur, Theni.

Mrs. A. Amalorpava Selvi, **M.E.,**
Lecturer / ECE,
Pattukkottai Polytechnic College,
Pattukkottai - 614 601.

Validated By
Dr. S. Rajaram, M.E., Ph.D.,
Assistant Professor / ECE,
Thiagarajar College of Engineering,
Madurai – 625 015.

VERY LARGE SCALE INTEGRATION

DETAILED SYLLABUS

UNIT I

1.1 COMBINATIONAL CIRCUIT DESIGN: NMOS and CMOS logic implementation of Switch, NOT, AND, OR, NAND, and NOR Gates CMOS Transmission Gate. Digital logic variable, functions, inversion, gate/circuits, Boolean algebra and circuit synthesis using gates (Up to 4 variables).

1.2 COMBINATIONAL CIRCUIT BUILDING BLOCKS:

Circuit synthesis using Multiplexer, Demultiplexer, Encoders and Decoders, Arithmetic adder, Sub tractor and Comparator circuits. Hazards and races.

UNIT II

2.1 VHDL FOR COMBINATIONAL CIRCUIT: Introduction to VLSI and its design process. Introduction to CAD tool and VHDL: Design Entry, Synthesis, and Simulation. Introduction to HDL and different level of abstractions. HDL Statements and Assignments

2.2 VHDL CODE: AND, OR, NAND, NOR gates, Implementation of Mux, Demux, Encoder, decoder. Four bit Arithmetic adder, sub tractor and comparator in VHDL

UNIT III

3.1 SEQUENTIAL CIRCUIT DESIGN: Introduction/Refreshing to Flip- flops and its excitation table, counters and Shift registers

3.2 DESIGN STEPS: State diagram, State table, state assignment. Example for moore and mealy machines. Design of modulo counter (upto 3 bit) with only D flip-flops through state diagram

UNIT IV

4.1 VHDL FOR SEQUENTIAL CIRCUIT: VHDL constructs for storage elements. VHDL code for D Latch / D, JK and T Flip-flops withorwithout reset input.

4.2 VHDL EXAMPLES: Counters :Synchronous counters-2 bit &3 bit up counter. 3 bit up/down counter Decade counter, Johnson Counter

UNIT V

PLDS AND FPGA: Introduction to PROM, PLA and PAL. Implementation of combinational circuits with PROM, PAL and PLA (up to 4 variables). Comparison between PROM, PAL and PLA. Introduction to Complex Programmable Logic device, Field Programmable Gate Array. Introduction to ASIC. Types Of ASIC

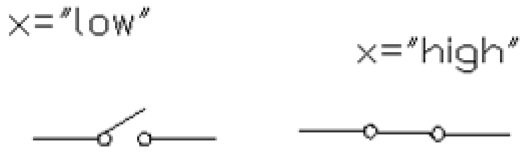
CONTENT

UNIT – 1

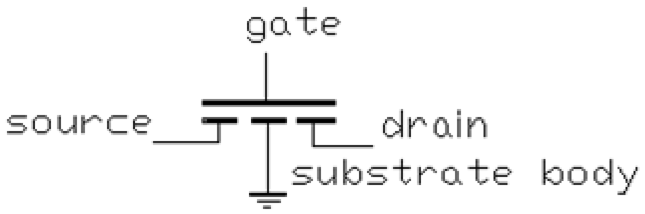
COMBINATIONAL CIRCUIT DESIGN:

NMOS IMPLEMENTATION OF SWITCH

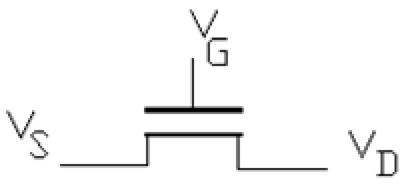
For the purpose of understanding how logic circuits are built, we can assume that a transistor operates as a simple switch. figure 1.1a shows a switch controlled by a logic signal , x when x is low, the switch is open, and when x is high, the switch is closed. The most popular type of transistor field-effect transistor (MOSFET) There are two different types of MOSFERs, Known as n-channel, abbreviated NMOS, and p-channel, denoted PMOS.



(a) A simple switch controlled by the input x



(b) NMOS transistor



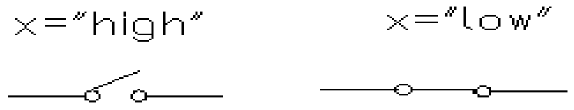
(c) Simplified symbol for an NMOS transistor

Figure 1.1 NMOS transistor as a switch

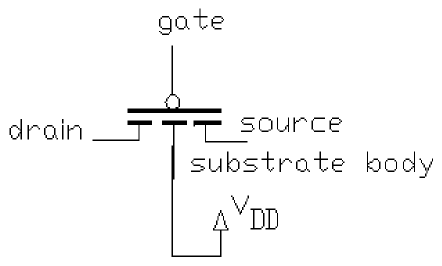
Figure 1.1b gives a graphical symbol for an NMOS transistor. it has four electrical terminals, called the source, drain, gate, and substrate. in logic circuits the substrate (also called body) terminal is connected to Gns. we will use the simplified graphical symbol in figure 1.1c, which omits the source and train terminals. They are distinguished in practice by the voltage levels applied to the transistor. The terminal with the lower voltage level is assumed as source.

If Vg is low, Then there is no connection between the source and drain, the transistor is turnetoff.If Vg is high, then the transistor is turned on and acts as a closed switch that connects the source and train terminals.

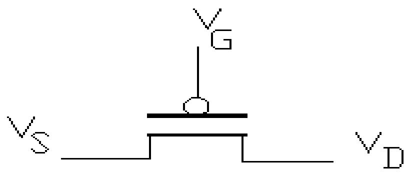
PMOS transistors have the opposite behavior of NMOS transistors. the type of switch is open when the control input x is high and closed when x is low . A symbol is shown in figure 1.2b.



(a) A Switch with the opposite behavior of Figure 1.2a



1.2 (b) PMOS transistor



1.2 (c) Simplified symbol for an PMOS transistor

In logic circuits the substrate of the PMOS transistor is always connected to V_{DD} leading to the simplified symbol in figure 1.2c. if Vg is high, then the PMOS transistor is turned on and acts as a closed switch that connect the source and drain. In the PMOS transistor the source is the node with the higher voltage.

Figure 1.3 summarizes the typical use of NMOS and PMOS transistor in logic circuits. An NMOS transistor is turned on when its gate terminal is high.

A PMOS transistor is turned on when the NMOS transistor is turned on, its drain is pulled down to Gns, and when the PMOS transistor is turned on its drain is pulled up to V_{DD}.

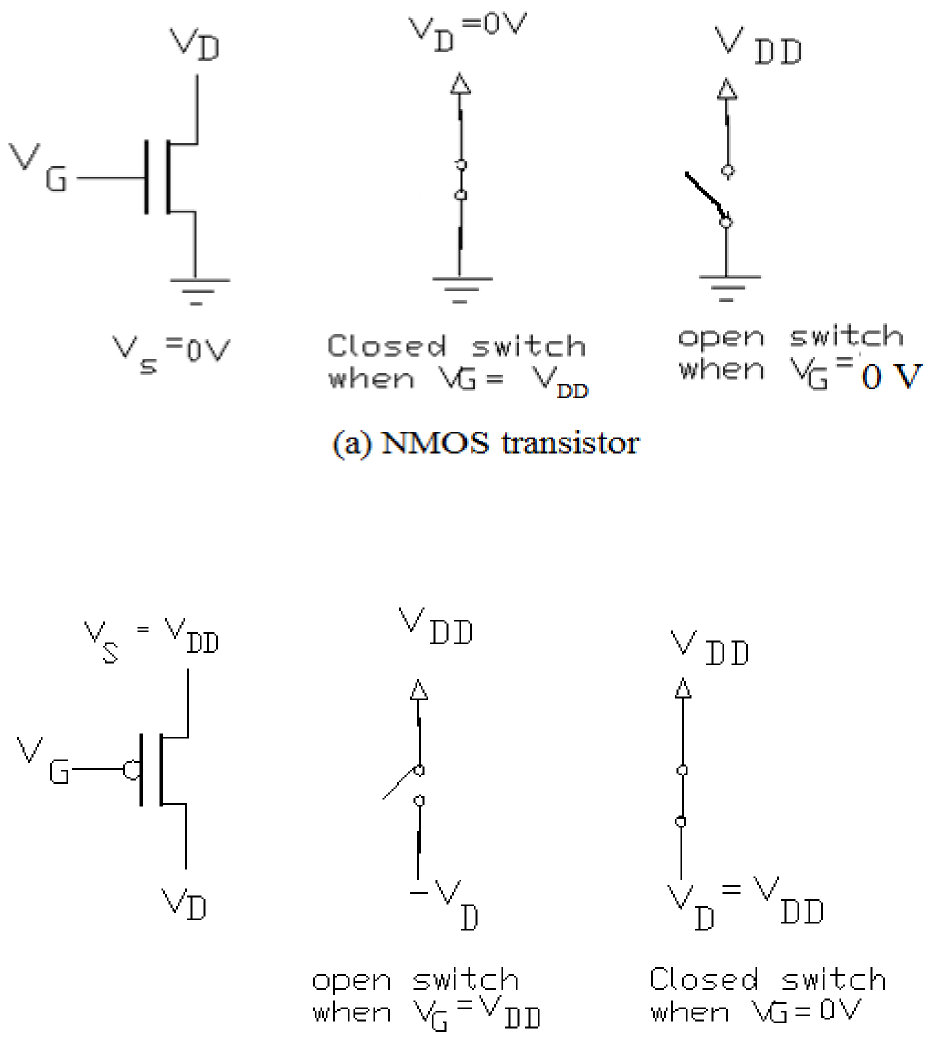

V_D
V_G
$V_S = 0V$
$V_D = 0V$
Closed switch when $V_G = V_{DD}$
V_{DD}
open switch when $V_G = 0 V$
(a) NMOS transistor
$V_S = V_{DD}$
V_G
V_D
V_{DD}
V_D
open switch when $V_G = V_{DD}$
V_{DD}
$V_D = V_{DD}$
Closed switch when $V_G = 0V$

Figure 1.3 NMOS and PMOS transistor in logic circuits.

NMOS Implementation of NOT Gate

NMOS Implementation of NOT Gate in the circuit in figure 1.4a, when Vx = 0v, the NMOS transistor is turned of f. No current flows through the resistor R, and Vf to a low voltage level

If Vf is viewed as a function of Vx then the the circuit is an NMOS implementation of a NOT gate. in logic terms this circuit implements the function f = x Figure 1.4b gives a simplified circuit diagram in which the connection to the positive terminal on the power supply is indicated by an arrow labeled Vdd and the connection to the negative power supply terminal is indicated by the Gnd symbol.

Figure 1.4c presents the graphical symbols for a NOT gate. The left symbol shows the input, output, power, and ground terminals, and the right the symbol shows only the input and output terminals. In practice only the simplified symbol is used. another name often used for the NOT gate is inverter.

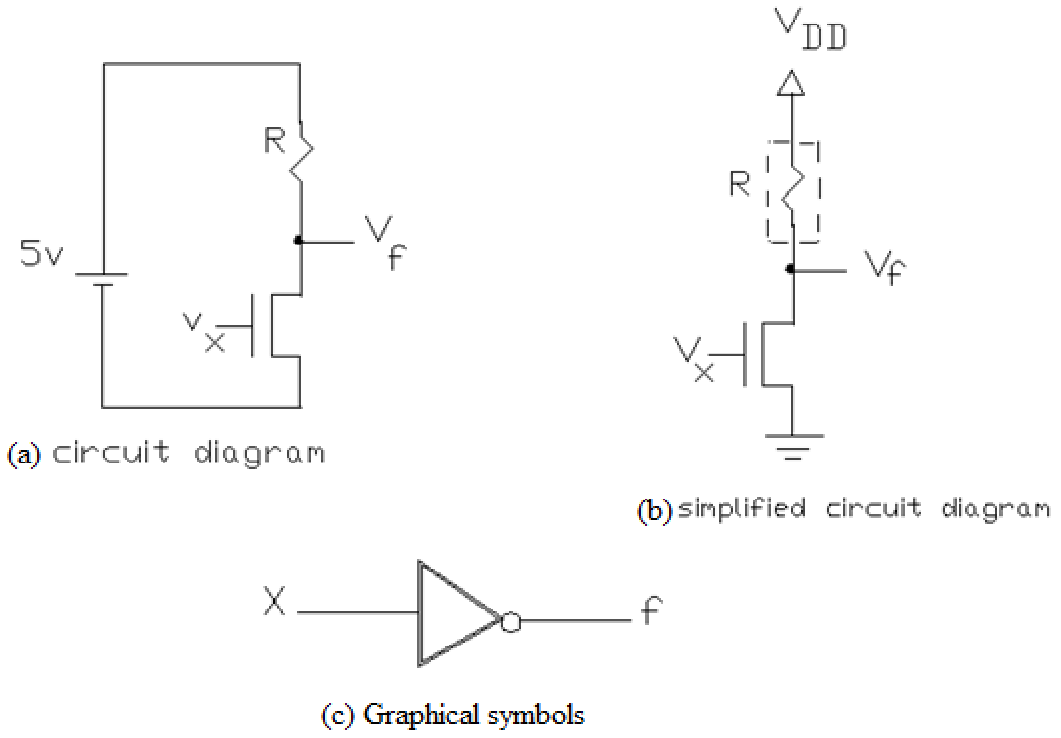
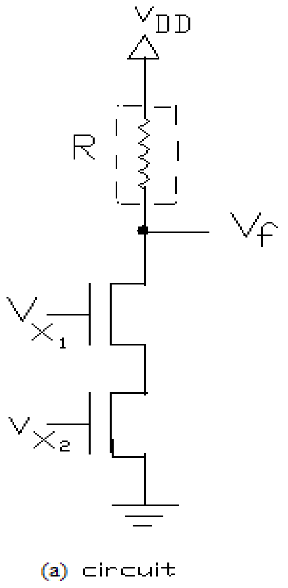


Figure 1.4 A NOT gate built using NMOS technology

NMOS Implementation of NAND Gate

Using NMOS transistor, we can implement the series connection as depicted in figure 1.5a. If $V_{x1} = V_{x2} = 5V$, both transistors will be on and V_f will be close to 0V. But if either V_{x1} or V_{x2} is 0, then no current will flow through the series – connected transistors and V_f will be pulled up to 5V. The resulting truth table for f, provided in terms of logic values, is given in figure 1.5b. Its graphical symbols are shown in figure 15c.



(b).Truth table

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

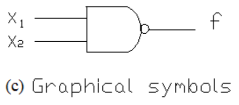
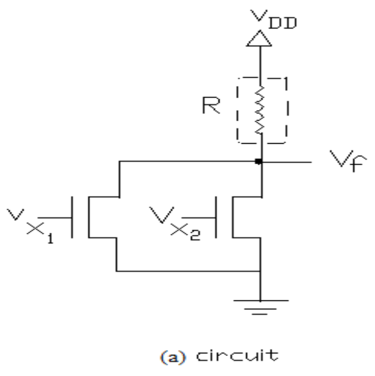


Figure 1.5 NMOS realization of a NAND Gate.

NMOS Implementation of NOR Gate :

The parallel connection of NMOS transistors is given in Figure 1.6a. Here, if either V_{x1} = 5 or V_{x2} =5 V, then V_{x2} will be close to 0 V. Only if both V_{x1} and V_{x2} are 0 will V_f be pulled up to 5V . A corresponding truth table is given in Figure 1.6b. The graphical symbols for the NOR gate appear in Figure 1.6c.



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

(b).Truth table

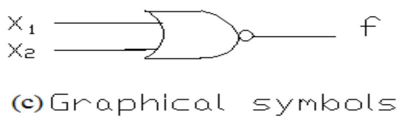
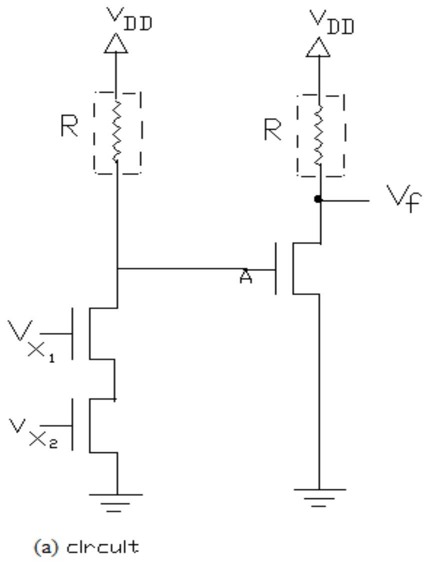


Figure 1.6 NMOS realization of a NOR gate.

NMOS implementation of AND Gate

Figure 1.7 indicates how an AND gate is built in NMOS technology by following a NAND gate with an inverter. Node A realizes the NAND of inputs x_1 and x_2 and f represents the AND function.



x_1	x_2	F
0	0	0
0	1	0
1	0	0
1	1	1

(b).Truth table

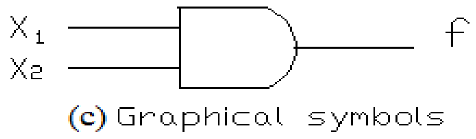
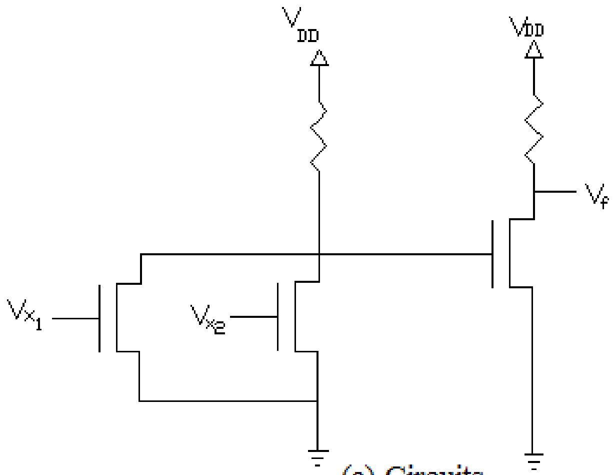


Figure 1.7 NMOS realization of an AND gate.

NMOS implementation of OR Gate

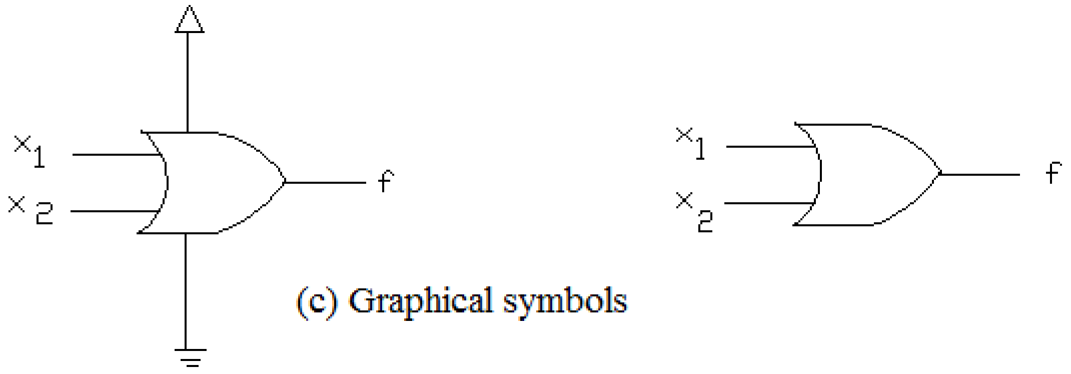
Figure 1.52 indicates how an or gate is built in NMOS technology by following NOR Gate with an inverter



(a) Circuits

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

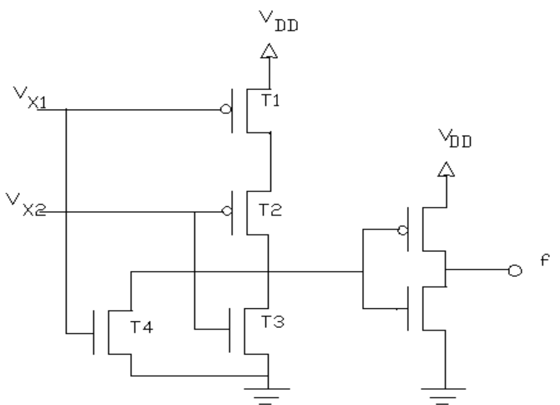
(b) Truth Table



(c) Graphical symbols

CMOS OR Gate

A CMOS OR gate is built with a NOR gate followed by a NOT gate.

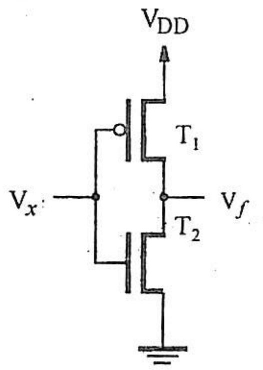


x_1	x_2	f
0	0	0
0	1	1
1	0	1
1	1	1

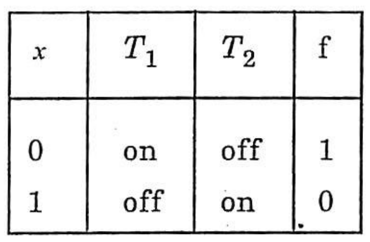
Figure 1.8a Circuit 1.8 b.Truth Table

CMOS NOT Gate:

The simplest example of a CMOS circuit, a NOT gate, is shown in figure 1.9. whenVx = 0 v, transistor T2 is off and transistor T1 is on This makes Vf = 5v, and since T2 is off and no current flows through the transistor. When Vx = 5V, T2 is on and T1 is off ThusVf = 0v, and no current flows because T1 is off



(a) Circuit



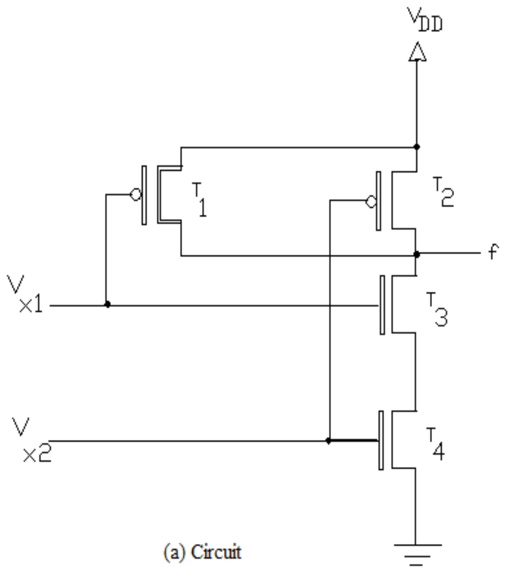
x	T_1	T_2	f
0	on	off	1
1	off	on	0

(b) Truth table and transistor states

Figure 1.9a Circuit 1.9b Truth table and transistor states

CMOS NAND Gate

Figure 1.10 shows a circuit diagram of CMOS NAND gate. The truth table in the figure specifies the state of each of the four transistors for each logic valuation of inputs X1 and X2 The circuit properly implements the NAND function Under static conditions no patch exists for current flow from Vdd to Gnd.



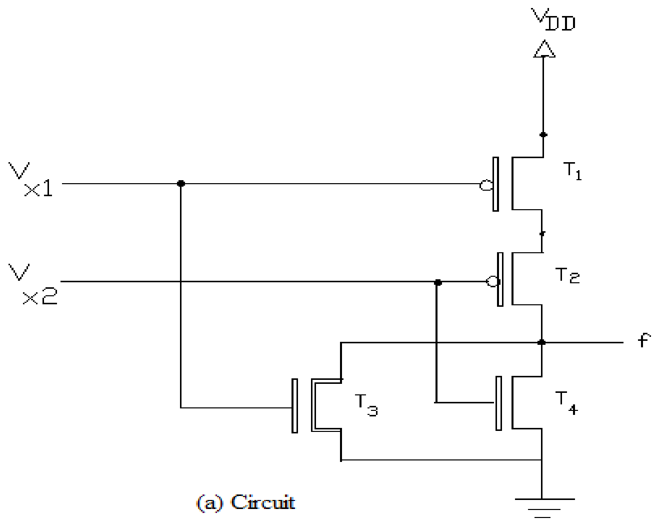
x_1	x_2	T_1	T_2	T_3	T_4	f
0	0	on	on	off	off	1
0	1	on	off	off	on	1
1	0	off	on	on	off	1
1	1	off	off	on	on	0

(b).Truth Table

Figure 1.10 CMOS realization of a NAND gate.

CMOS NOR Gate

The circuit for a CMOS NOR gate is shown in Fig. 1.11. This Circuit functions as per the truth table.



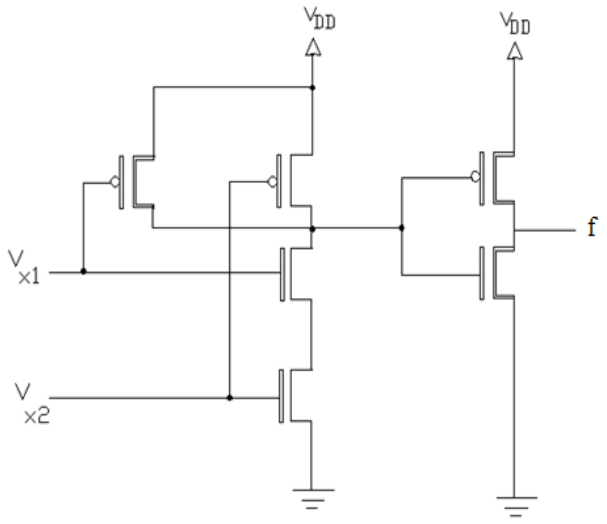
x_1	x_2	T_1	T_2	T_3	T_4	f
0	0	on	on	off	off	1
0	1	on	off	off	on	0
1	0	off	on	on	off	0
1	1	off	off	on	on	0

(b).Truth Table

Figure 1.11 CMOS realization of a NOR gate.

CMOS AND Gate

A CMOS AND gate is built by connecting a NAND gate to an inverter, as illustrated in figure 1.12. Similarly, an OR gate is constructed with a NOR gate followed by a NOT gate.



x_1	x_2	f
0	0	0
0	1	O
1	0	O
1	1	1

(b).1.12 b Truth Table

Figure 1.12a CMOS realization of a AND gate.

CMOS TRANSMISSION GATE

Basic Operation

A transmission gate, or analog switch, is defined as an electronic element that will selectively block or pass a signal level from the input to the output. This solid-state switch is comprised of a pMOS transistor and nMOS transistor. The control gates are biased in a complementary manner so that both transistors are either on or off.

When the voltage on node A is a Logic 1, the complementary Logic 0 is applied to node active-low A, allowing both transistors to conduct and pass the signal at IN to OUT. When the voltage on node active-low A is a Logic 0, the complementary Logic 1 is applied to node A, turning both transistors off and forcing a high-impedance condition on both the IN and OUT nodes. This high-impedance condition represents the third "state" (high, low, or high-Z) that the DS3690 channel may reflect downstream.

The schematic diagram (**Figure 1**) includes the arbitrary labels for IN and OUT, as the circuit will operate in an identical manner if those labels were reversed. This design provides true bidirectional connectivity without degradation of the input signal.

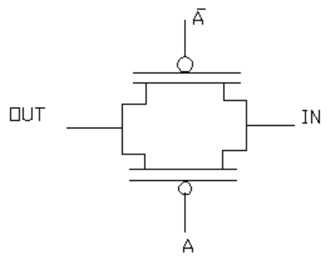


Figure 1. Schematic representation of a transmission gate.

The common circuit symbol for a transmission gate depicts the bidirectionalnature of the circuit's operation (**Figure 2**).

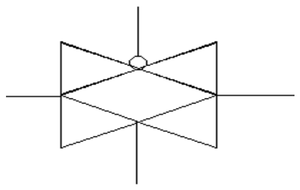


Figure 2. Circuit symbol.

Digital Logic Variables & Functions

In digital systems, binary circuits are used because the binary element is 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 = o and closed if x=1, as illustrated in figure 1.13a. The graphical symbol in figure 1.13b

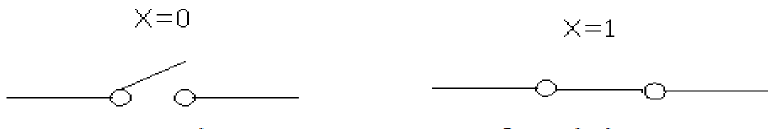
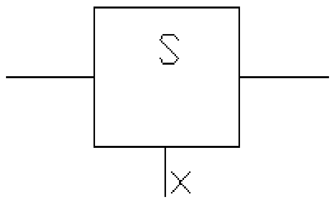


Figure 1.13a Two staes of a switch

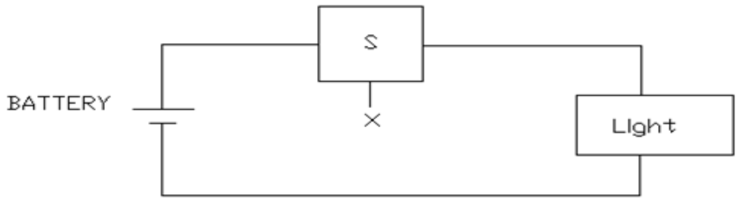
to represent such switches in the diagrams that follow Note that the control input x is shown explicitly in the symbol.



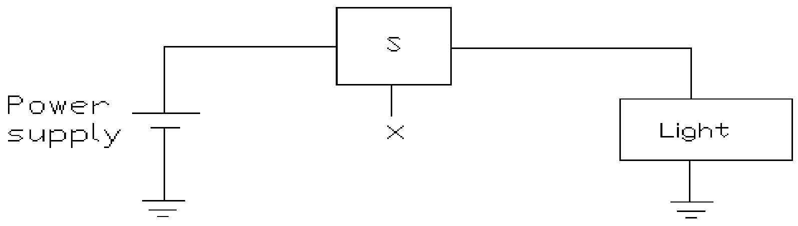
(b) Symbol for a switch

Figure 1.13b binary switch

Consider a simple application of a switch turns a small light bulb on or off this action is accomplished with the circuit in figure 1.14a. A battery provides the power source The current flows when the switch is closed, that is, when x = 1. In this example the input that causes changes in the behavior of the circuit is the switch control x.



(a) Simple connection to a battery



(b) Using a ground connection as the return path

Figure1.14 A light controlled by a switch

The output is defined as the state (or condition) the light, which we will denote by the letter L. if the light is on, we will say that L=1.if the light is off, L=0. using this convention, we can describe the state of the light as a function of the input variable x. since L=1 if x=1 and l=0 if x=0, we can say that

$$L(x) = x$$

We say that L(x) = x is a logic function and that x is an input variable.

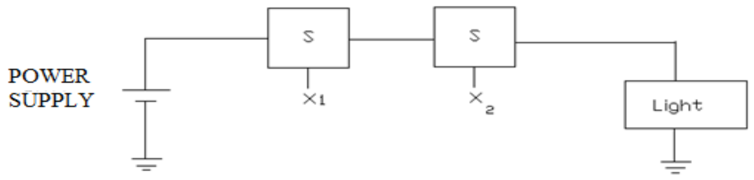
The circuit in figure 1.14a in an ordinary flashlight, where the switch is a simple mechanical device. in an electronic circuit the switch is implement as a transistor and the light may be a light-emitting diode (LED). An electronic circuit is powered by a power supply of a certain voltage, like 5 volts, One side of the power supply is connected to ground, as shown in figure 1.14b. the ground connection is used as the return path for current, to close the loop. This is achieved by connecting one side of the light to ground as indicated in the figure.

Consider now the possibility of the using two switches to control the state of the light let x1 and x2 be the control inputs for these switches the switches can be connected either in series or in parallel as shown in figure 1.15. using a series connection, the light will be turned on only if both switches are closed. if either switches are closed if either switch is open the light will be off. this behavior can be described by the expression

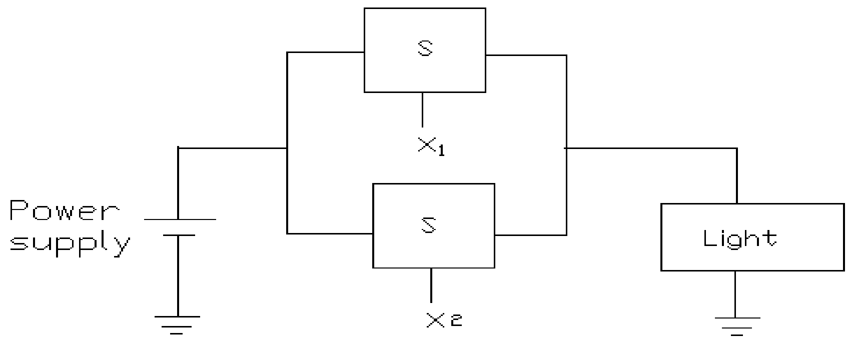
$$L(x_1, x_2) = x_1.x_2$$

Where $L = 1$ if $x_1 = 1$ and $x_2 = 1$,

$L = 0$ otherwise.



(a) The logical AND function (series connection)



(b) The logical OR function (parallel connection)

Figure 1.15 Two basic functions.

The "." symbol is called the AND operator, and the circuit in figure 1.15a is said to implement a logical AND function

The parallel connection of two switches is given in figure 1.15b. in this case the light will also be off only if both switches are open. This behavior can be on if either x1 or x2 switch is closed. The light will also be on if both switches are open . This behavior can be stated as

L(x1, x2)=x1+x2

Where L=1 if x1=1 or if x1=x2=1,

L=0 if x1=x2=0.

The + symbol is called the OR operator, and the circuit in Figure 1.15b is said to implement a logical of function

In the above expressions for AND and OR, the output L(x1,x2) is a logic function with input variables x1 and x2 the AND and OR functions are two of the most important logic functions. Together with some other simple function they can be used as building blocks for the implementation of all logic circuits. figure 1.16 illustrates how there switches can be used control the light in a more complex way. This series-parallel connection of switches realizes the logic function

L(x1, x2, x3)=(x1+x2.x3)

The light is on if x2, = 1 and, at the same time, at least one of the x1 or x2 inputs is equal to 1.

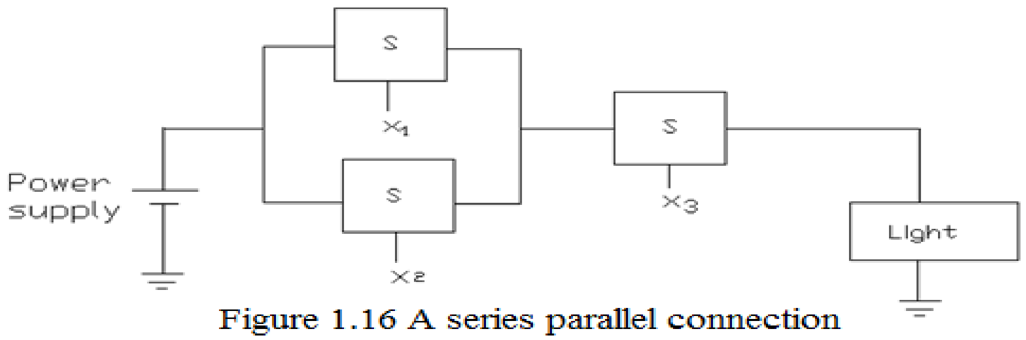


Figure 1.16 A series parallel connection

Inversion

A positive action takes place when a switch is opened. Suppose that we connect the light as shown in Figure 1.17. in this case the switch is connected in parallel with the light, rather than I series. Consequently, a closed switch will short-circuit the light and prevent the current from following through it. an extra resistor in this circuit dose not short-circuit the power supply. The light will be turned on when the switch is opened. formally, we express this functional behavior as

L(x)= x

Where L=1 if x=0,

L=0 if x = 1

The value of this function is the inverse of the value of the input variable instead of using the word inverse, it is more common to use the term complement. thus we say that L(x) is a complement of x in this example another frequently used term for the same operation is the NOT operation.

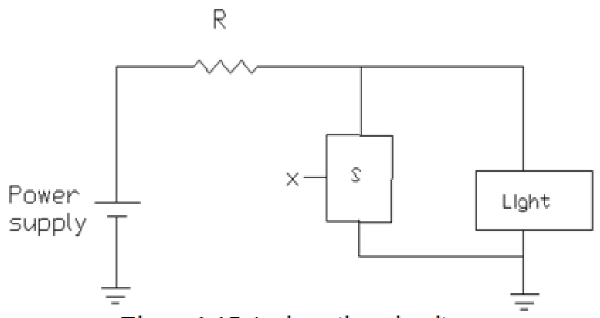
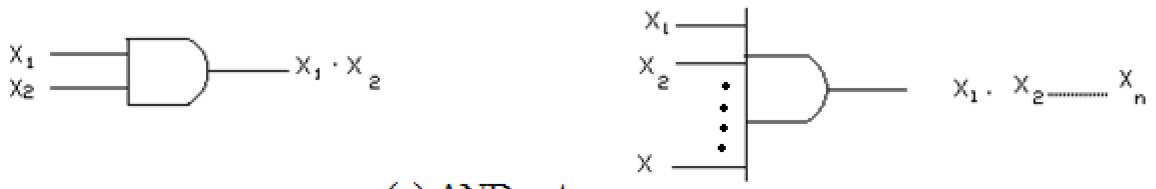


Figure 1.17 An inverting circuit

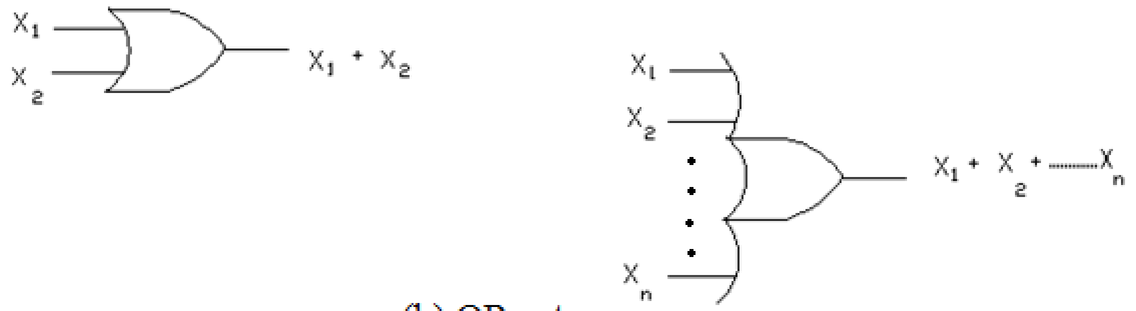
Logic Gates and Networks

Each logic operation can be implemented with transistors, resulting In a circuit element called logic gate a logic gate has one or more inputs and output that is a function of its inputs. A logic circuit diagram, consisting of graphical symbols representing the logic gates. the graphical symbols for the AND, OR, and NOT gates are shown in Figure 1.18. The figure indicates on the left side how the AND and OR gates are drawn when there are only a few inputs. On the right side it shows how the symbols are enlarged to accommodate a greater number of inputs.

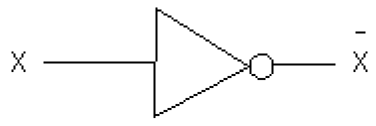
A larger circuit is implemented by a network of gates for example, the logic function from figure 1.19. A given logic function can be implemented with a number of different networks.



(a) AND gate



(b) OR gate



(c) NOT gate

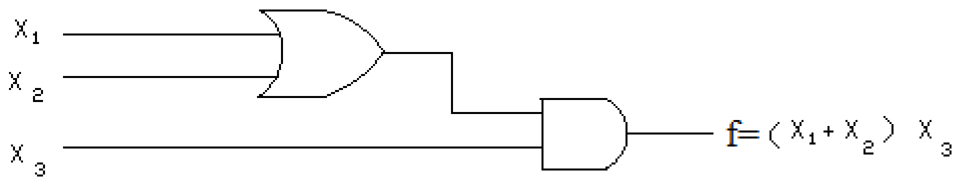


Figure 1.19 The function from Figure 1.18

BOOLEAN ALGEBRA

In 1849 George Boole published a scheme for the algebraic description of processes. it involved in logical thought and reasoning this scheme and its further refinements became known as Boolean algebra provides It was almost 100 years later that this algebra found application in the engineering sense. in the late 1930s claude Shannon showed that Boolean algebra provides an effective means of describing circuits built with switches. The algebra can, therefore, be used to describe logic circuits This algebra is a powerful tool that can be used for designing and analyzing logic circuits

Axioms of Boolean Algebra

Like any algebra, Boolean algebra is based on a set rules that are derived from a small number of basic assumptions are called axiom let us assume that Boolean algebra values, 0 and 1. Assume that the following axioms are true

1a	$0.0=0$
1b	$1+1=2$
2a	$1.1=1$
2b	$0+0=0$
3a	$0.1=1.0=0$
3b	$1+0=0+1=1$
4a	If $x = 0$, then $x = 1$
4b	If $x= 1$, then $x=0$

Single-Variable Theorems

From the axiom we can define some rules for single variables. these rules are often called theorems if x is variables in B, the n the following thermos hold:

5a. x.o=0

5b. x+1=1

6a. x.1=x

6b. x+0=x

7a. x.x=x

7b. x+x=x

8a. x.x=0

8b. x+x=1

9. x=x

it is easy to prove the validity of these theorems by substituting the values x=o and x=1 into the expressions and using the axioms given above. for example, in theorem 5a, if x = 0, then the theorem states that that 0.0 =0, which is true according to axiom la similarly, if x = 1, then theorem 5a status that 1.0 = 0, which is also true according to axiam 3a.

Duality

Given a logic expression, its dual is obtained by replacing all+ operators, and vice versa, and by the replacing all 0s with 1s, and vice versa. The dual of any true statement (axiom or theorem) in Boolean algebra is also a true statement.

Two-and Three – variable Properties

If x,y, and z are the variables in B, then the following properties hold:

10a.	x.y = y.x	Commutative
10b.	x+y = y+x	
11a.	x.(y.z) = (x.y).z	Associative
11b.	x+(y+z) = (x+y) +z	
12a.	x.(y+z) =x.y + x.z	Distributive
12b.	x+y.z = (x+y). (x+z)	
13a.	x+x.y = x	Absorption
13b.	x. (x+y) = x	
14a.	x.y + x.y = x	Combining
14b.	(x+y). (x+y) = x	
15a.	x.y. = x+y	De Morgan's theorem
15b.	x+y = x .y	
16a.	x+x .y = x+y	
16b.	x. (x+y) = x.y	
17a.	x. y+y. z+x . z=x .y+x.z	Consensus
17b.	(x+y). (y+z). (x+z) = (x+y). (x+z)	

INPUT		LHS		RHS		
x	**y**	$\bar{\bar{x}}.y$	$\overline{x.y}$	$\bar{x}$	$\bar{y}$	$\bar{x} + \bar{y}$
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

Figure 1.20 Proof of Demorgan's theorem in 15a.

Again, we can prove the validity of these properties either by perfect induction or by performing algebraic manipulation. figure 1.20 illustrates how perfect induction of a truth table. The evaluation of left-hand and right-hand sides of the identity in 15 a gives the same result

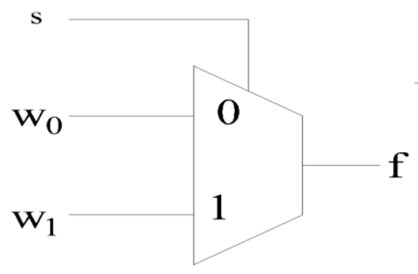
1.2. COMBINATIONAL CIRCUIT BUILDING BLOCKS

MULTIPLEXERS

A multiplexer circuit has a number of data inputs, one or more select inputs, and one output. it passes the signal value on one of the data inputs to the output. the data input is selected by the values of the select inputs figure1.21shows a 2-to1 multiplexer

part 1.21(a) gives the symbol commonly used the select input,s, chooses as the output of the multiplexer either input W0 or W1. the multiplexer's functionality can be described in the form of a truth table as shown in part 1.21b of the figure part 1.21(c) gives a sum-of-products implementation of the 2 to 1 multiplexer and part 1.21(d) illustrates haw can be constructed with transmission gates.

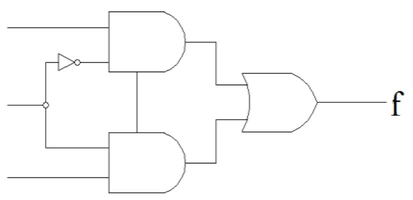
Figure 1.22a shows a – larger multiplexer with four data inputs , w_0, ….., w_3 and two select inputs, s_1 and s_0. As shown in the truth table in part (b) of the figure, the two-bit number represented by s_1s_0 selects one of the data inputs as the output of the multiplexer.



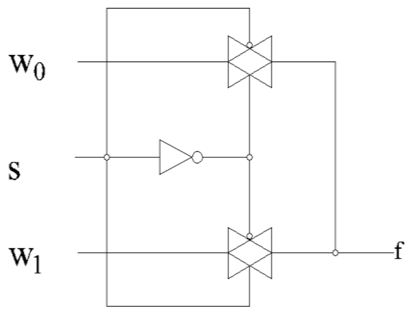
(a)graphical symbol

s	f
0	w_0
1	w_1

(b)truth table

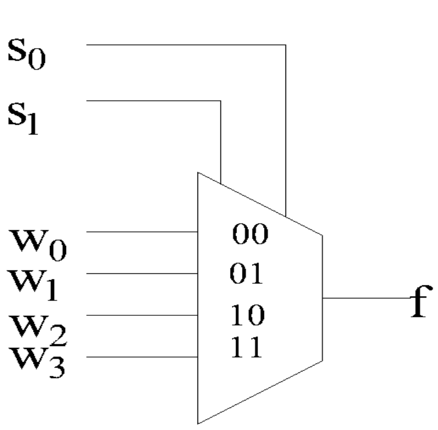


(c)sum -of-products circuit



(d)circuit with transmission gates

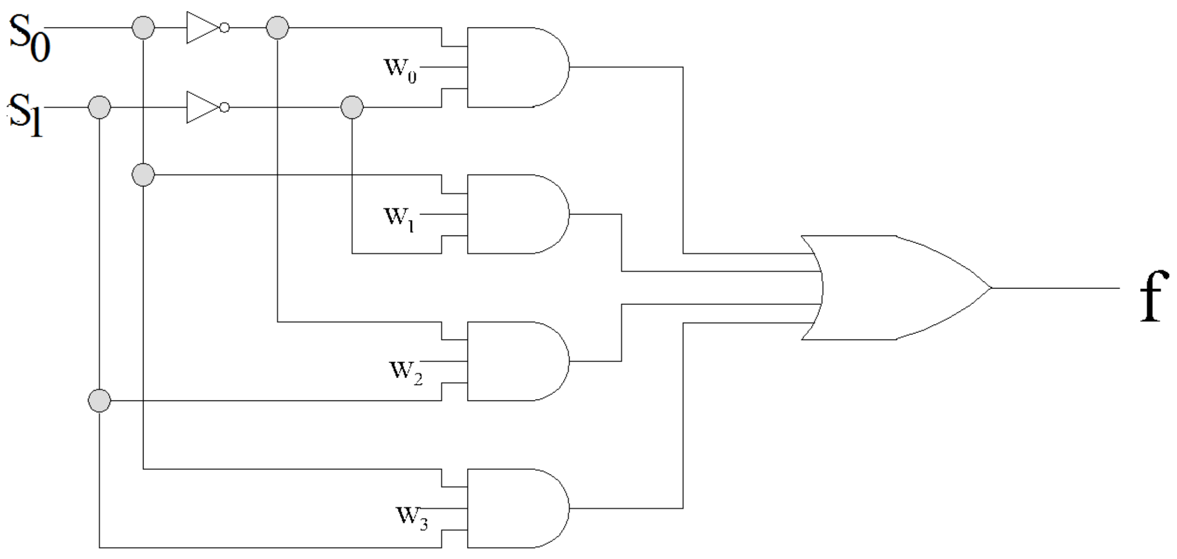
Figure 1.2 A 2-to-1 MULTIPLEXER



(a)graphical symbol

s_0 s_1	f
00	w_0
01	w_1
10	w_2
11	w_3

(b)truth table



(C) CIRCUIT

Figure 1.22 1 A 4-to-1 MULTIPLEXER

A sum – of – products implementation of the 4 to multiplexer appears in figure 1.22c. It realizes the multiplexer function.

$$F = s_1 s_0 w_0 + s_1 s_0 w_1 + s_1 s_0 w_2 + s_1 s_0 w_3$$

It is possible to build larger multiplexers using the same approach. Usually, the number of data inputs, n is a integer power of two. A multiplexer that has n data inputs w_0, ….., w_{n-1}, requires [$\log_2 n$] select inputs. Larger multiplexer can also be constructed from smaller multiplexers. For examples , the 4 to 1 multiplexer can be built using three 2 to -1 multiplexers as illustrated in figure 1.23.

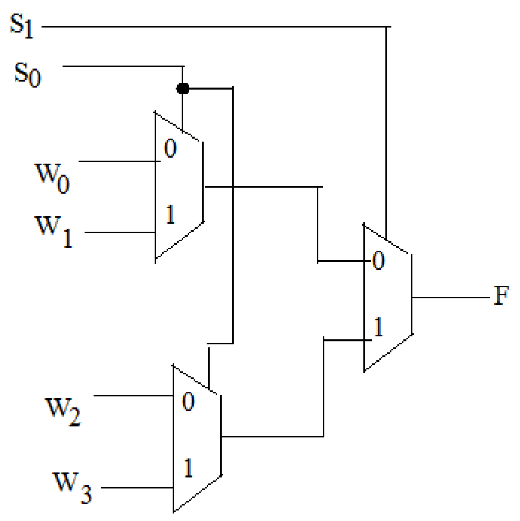


FIGURE 1.23 USING 2-to-1 MULTIPLEXER TO BUILD A 4-to-1 MULTIPLEXER

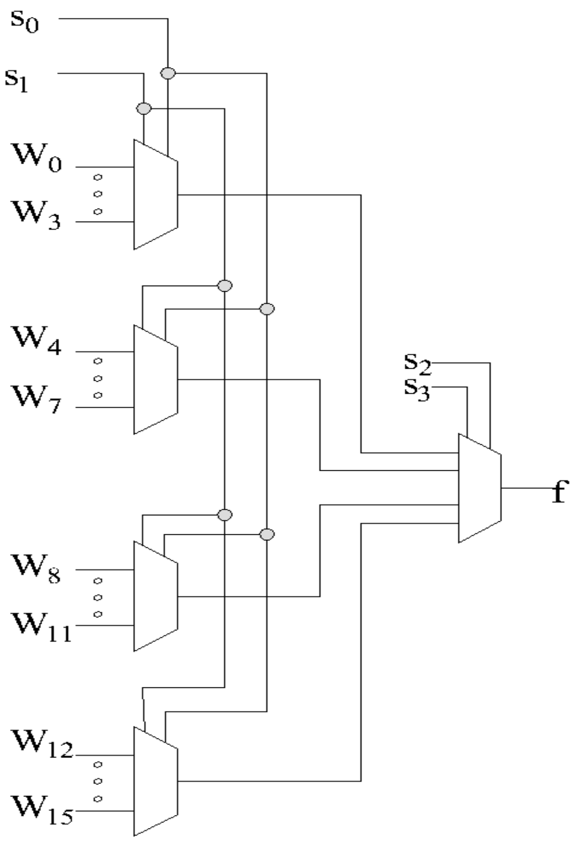


Figure 1.24 shows how a16 to 1 multiplexer is constructed with five 4 to 1 multiplexer

DEMULTIPLEXERS

The purpose of the multiplexer circuit is to multiplex then n data inputs onto the single data output under control of the select inputs

A circuit that performs the opposite function, namely, placing the value of a single data input onto multiple data outputs is called a **demultiplexer.** Thedemultiplexer can be implemented using a decoder circuit.

DECODERS

Decoders circuits are used to decode encoded information A binary decoder shown in the figure 1.25 is a logic circuit with n inputs and 2n outputs Only one outputs is asserted at a time, and each output corresponds to one valuation of the inputs .

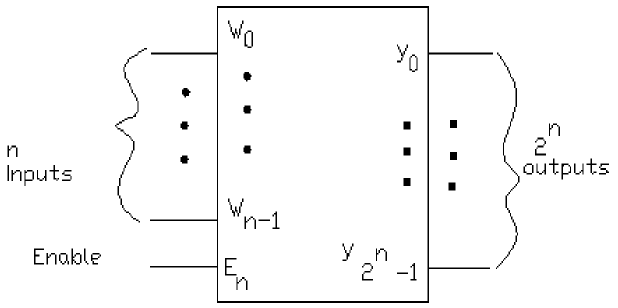
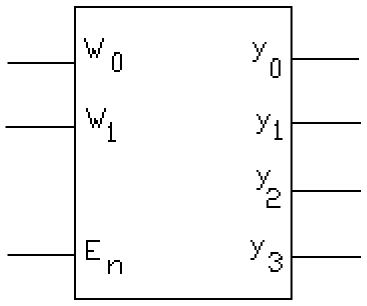


Figure 1.25 An n-to-2^n binary decoder

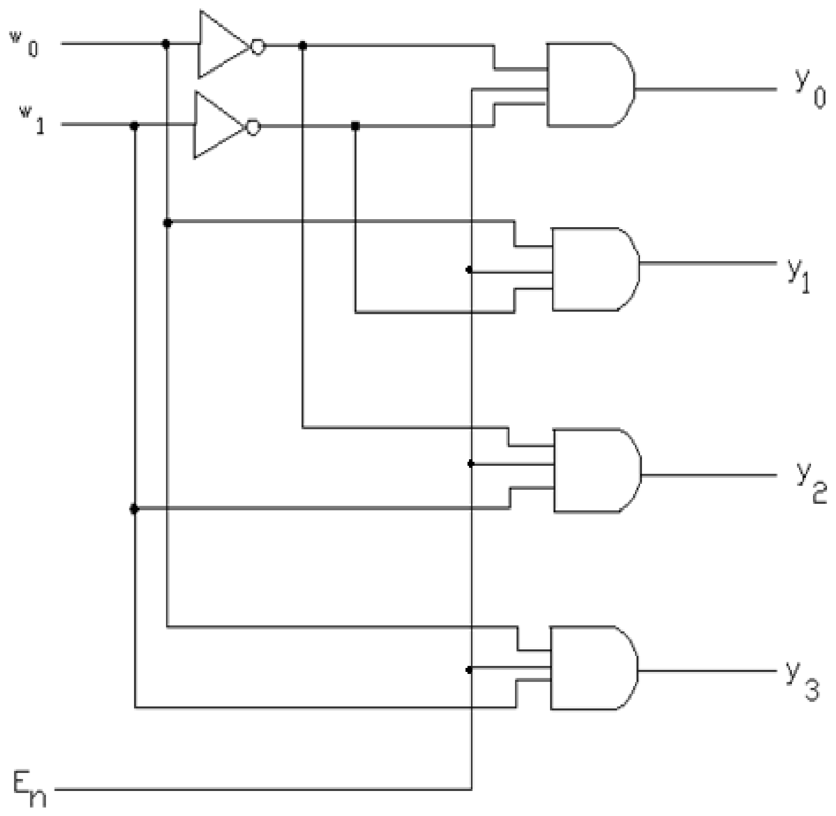
The decoder also has an enable input. En, that is used to disable the outputs; if En = 1, the valuation of wn-1….. w1w0 determines which of the outputs Is asserted

E_n	w_1	w_0	y_0	y_1	y_2	y_3
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



(b) Graphic symbol

(a) Truth Table



(c) Logic circuit

Figure 1.26 A-2-to-decoder

For example, the 2-to4 decoder in Figure 1.26 can be used as a 2-to4 Demultiplexer in this case the en input serves as the data input for the Demultiplexer, and they y0 to y3 outputs are the data input the valuation of w1 w0 determines which of the outputs is set to the value of En.

To see how the circuit works, consider the truth table in figure 1.26a. when En=0, all the outputs are set to 0, including the one selected by the valuation of w1w0 sets the appropriate to 1.

ENCODERS

An encoder performs the opposite function of a decoder in encodes given information into a more compact from.

BINARY ENCODERS

A binary encoder encodes information from 2n inputs into an n-bit code, as indicated in figure 1.27. exactly one of the input signals should have a value of 1, and the outputs present the binary number that identifies which input is equal to 1.

The truth table for a 4 to 2 encoder is provided in figure 1.27b. observe that the output y0 is 1 when either input y0 is 1 when either input w1 or w3 Is 1, and output y1 is 1 when input w2 or w3 is 1. Hence these outputs can be generate by the circuit Figure 1.27c

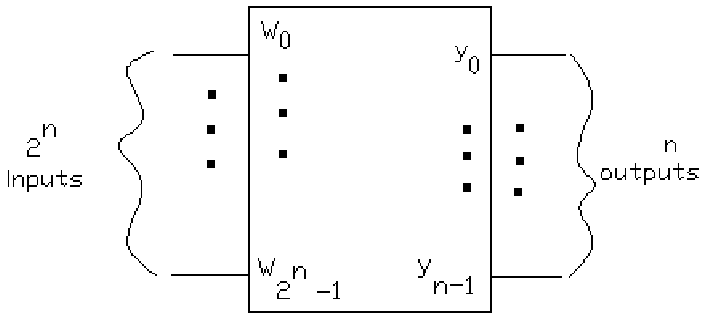


Figure 1.27 A 2^n-to-n binary encoder

Encoders are used to reduce the number of the bits needed to represent given information A practical use of encoders is for transmitting information in a digital system.

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

(b) Truth Table

BINNARY ADDER – SUBTRACTOR

A combinational circuit that performs the addition of three bits (two significant bits and a previous carry) is a full adder Two half adders can be employed to implement a full adder.

A binary adder subtractor is combinational circuit that performs the arithmetic operation of addition and subtraction with binary numbers the helf adder designs carried out first, from which we developthe full adder for two n bit numbers the subtraction circuit is included a complementing circuit

HALF ADDER

A half adder, needs two binary inputs and two binary outputs. the input variables designate the augends and addend bits; the output variables produce the sum and carry we assign symbols x and y to the two inputs and s (for sum) and C (for carry) to the outputs the block diagram of a half adder is shown in fig. 1.28. the truth table for the half adder is listed in Table 4.1. The c output represents the least significant bit of the sum.

The simplified Boolean functions for the two outputs can be obtained directly from the truth table the simplified sum of products expressions are the logic diagram of the half adder implemented in sum of products is shown in fig. 1.29(a) it can be also implemented with an exclusive OR and an AND gate as shown in fig 1.29(b). This from is used to show that two half adders can be used to construct a full adder.

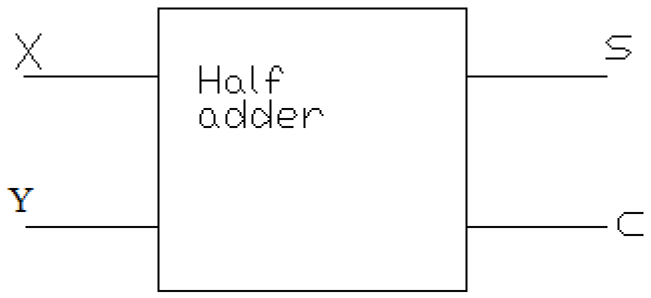


Figure1.28 Block diagram of half adder

Truth Table of Half Adder

X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

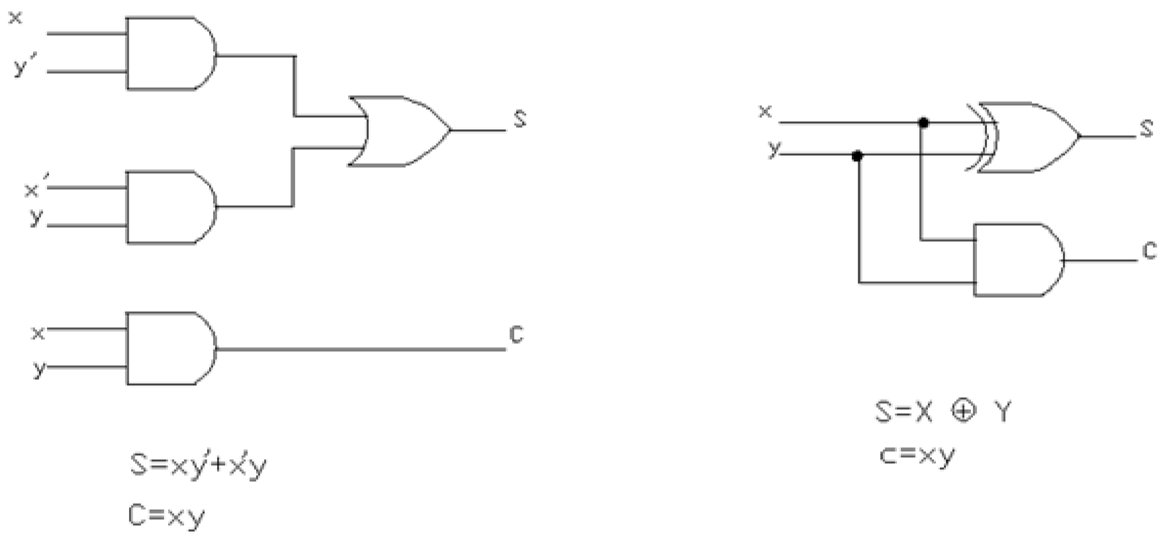


Figure 1.29 Implementation of half adder

FULL ADDER

A full adder is a combinational circuit that forms the arithmetic sum of three bits it consists of three inputs and two output. two of the input variables denoted by x and represents the two bits to be added the third input, z, represents the carry from the previous lower significant position

The block diagram of a full adder is shown in fig.1.30.

The truth table of the full adder is listed in table 1.2. the eight rows under the input variables designate all possible combinations of the three variables the output variables are determined from the arithmetic sum of the input bits.

When all input bits are 0, the output is 0. the S output is equal to 1 when only one input is equal to 1 or when all three inputs are equal to1. The c output has a carry of 1 if two or three inputs are equal to 1.

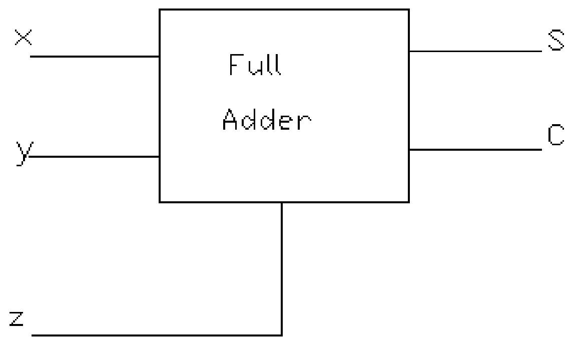


Figure 1.30 Block diagram of full adder

X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

The maps for the outputs of the full adder are shown in fig.1.31 the simplified expressions are

$$S = x'y'z + x'yz' + xy'z' + xyz$$

$$C = xy + xz + yz$$

The logic diagram for the full adder implemented in sum-of-products form is shown in fig 1.32. it can also be implemented with two half address and one OR gate, as shown in fig.1.33. the S output from the second half adder, giving

$$
\begin{aligned}
S &= z \oplus (x \oplus y) \\
&= z'(xy' + x'y) + z(xy' + xy)' \\
&= z'(xy' + x'y) + z(cy' + x'y') \\
&= xy'z' + x'yz' + xyz + x'y'z
\end{aligned}
$$

The carry output is

$$C = z(xy' + x'y) + xy = xy'z + x'yz + xy$$

yz x	00	01	11	10
0	m_0	m_1 1	m_3	m_2 1
1	m_4 1	m_5	m_7 1	m_6

yz x	00	01	11	10
0	m_0	m_1	m_3 1	m_2
1	m_4	m_5 1	m_7 1	m_6 1

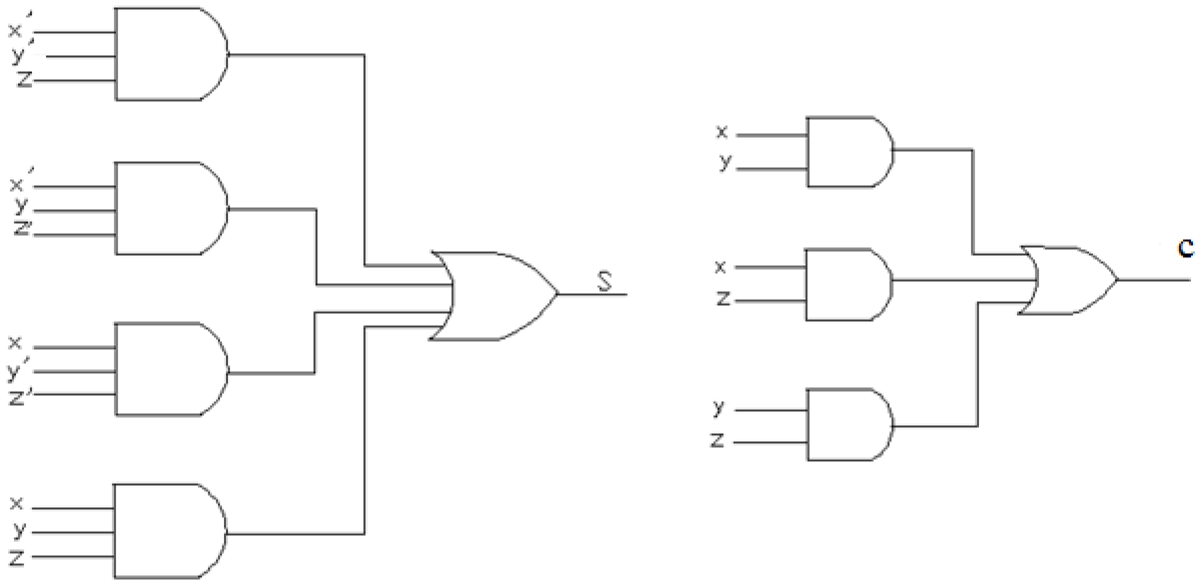


Figure 1.32 Implementation of full adder in sum-of- products

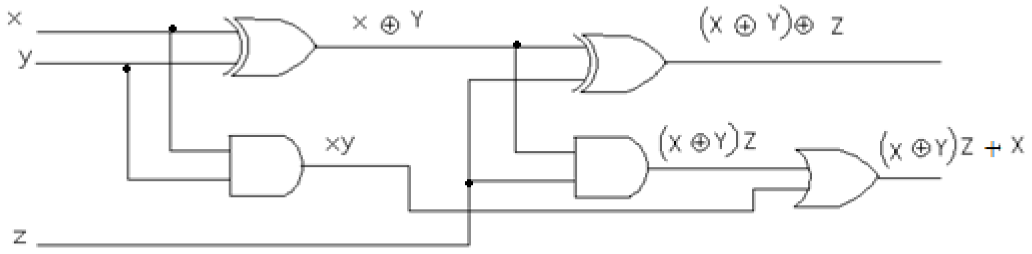


Figure 1.33 Implementation of full adder with two half adders an OR gate

Binary Adder

A binary adder is a digital circuit that produces the arithmetic sum of two binary numbers. it can be constructed with full adders connected in cascade, with the output carry from each full adders connected to the input carry of the next of four full adder (FA) circuits to provide a four-bit binary ripple carry adder.

The input carry to the adder is C0, and it ripples through the full adders to the full adders to the output carry c4. the S outputs generate the required sum bits.

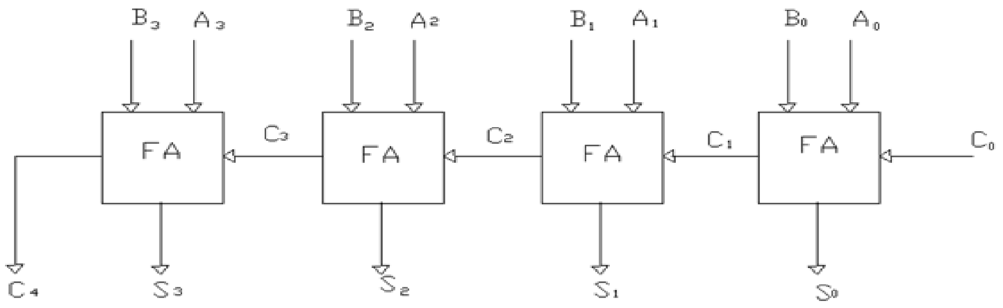


Figure 1.34 Four -bit adder

To demonstrate with a specific example, consider the two binary numbers A=1011and B=0011 Their sum S=1110 is formed with the four-bit adder as follows:

Subscript i:	**3**	**2**	**1**	**0**	
Input carry	0	1	1	0	C_i
Augends	1	0	1	1	A_i
Addend	0	0	1	1	B_i
Sum	1	1	1	0	S_i
Output carry	0	0	1	1	C_{i+1}

Half Subtractor

The block diagram shown in fig 1.35 is a half subtractor and it has two inputs and two outputs. The two inputs and y form the minuend and the subtrahend D is the difference output and B is the borrow output. the function table explains the working of the half subtract or (Table 1.3). The simplified sum of products expressions are

$$D = x'y + xy'$$

$$B = x'y$$

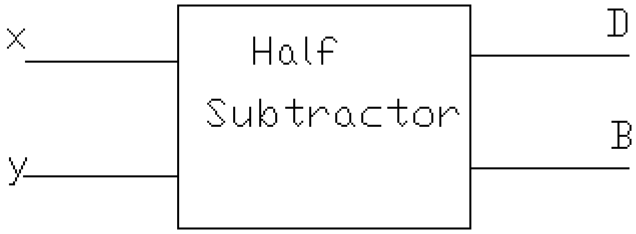


Figure 1.35 Block diagram of half subtractor

Table 1.3 Half Subtractor

x	**y**	**D**	**B**
0	0	0	1
0	1	1	1
1	0	1	0
1	1	0	0

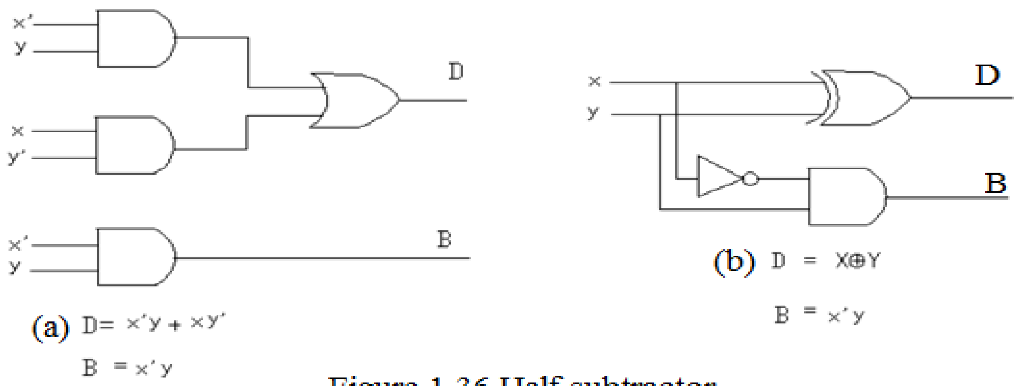


Figure 1.36 Half subtractor

The logic diagram implementation of these two expressions using basic gates is shown in fig 1.36(a) It can also be implemented using and EX-OR gate and an AND gate as indicated in Fig.1.36(b).

Full Subtractor

A full subtractor has three inputs and two outputs x,y and z are the inputs to be subtracted in which z represents borrow from the next stage. D and B are the outputs. The block diagrams of a full subtract or is shown in Fig. 1.37. Table 1.4 represents the truth table for a full subtractor and Fig. 1.38(a,b) shows the maps for outputs.

$$D = x'y'z + x'yz' + xy'z' + xyz$$

$$B = x'z + x'y + yz$$

The simplified expressions for D and B are implemented using basic gates are shown in fig 1.39.

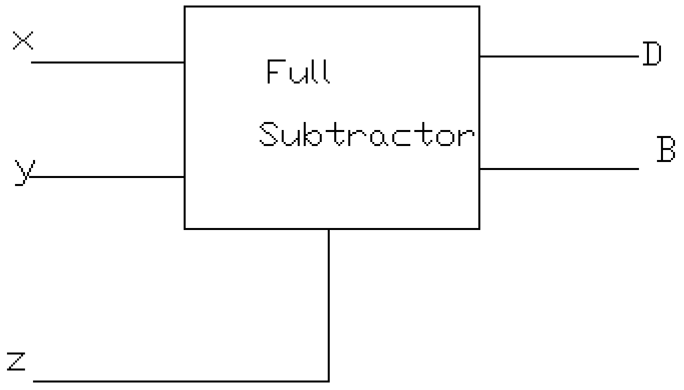


Figure 1.37 Block diagram of full subtractor

Table 1.4 Full Substractor

x	**Y**	**Z**	**D**	**B**
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

yz / x	00	01	11	10
0	m_0	m_1 1	m_3	m_2 1
1	m_4 1	m_5	m_7 1	m_6

Figure 1.38a Maps for full subtractor

(a) K map for D=x’y’z+x’yz’+xy’z’+xyz

yz / x	00	01	11	10
0	m_0	m_1	m_3 1	m_2
1	m_4	m_5 1	m_7 1	m_6 1

Figure 1.38b Maps for full substractor

(b) K map for B=x’z+x’y+yz

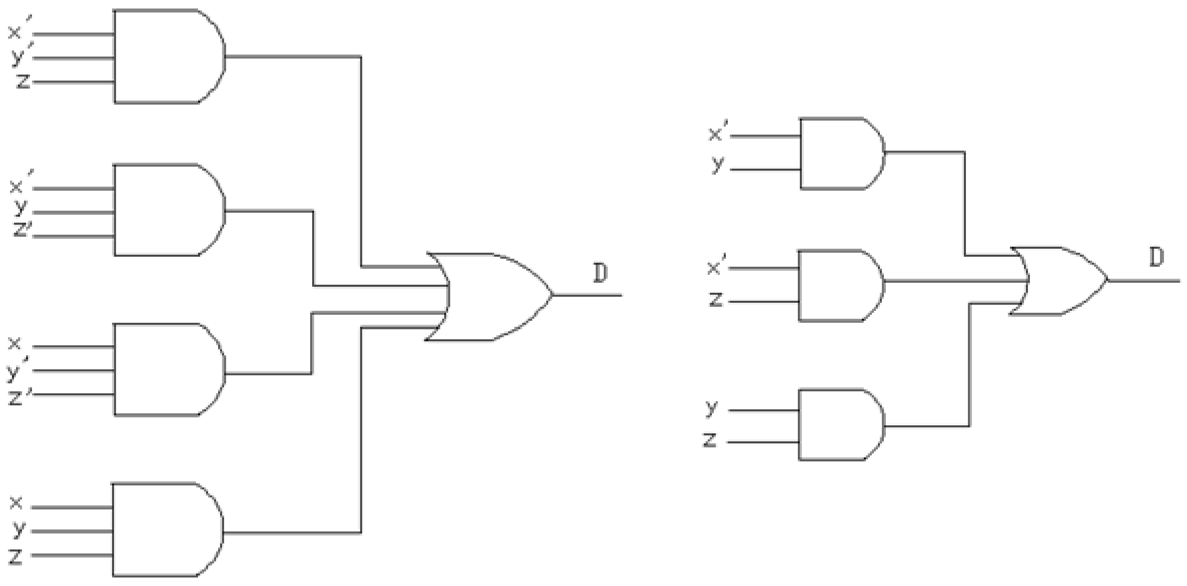


Figure 1.39 Implementation of full substractor

Binary subtractor

The subtractor of unsigned binary numbers is done by means of complements, Remember that the subtraction A-B is done by taking the 2's complement of B and adding it to A.

The circuit for subtracting A-B consists of an adder with inverters placed between each data input B and the corresponding input of the full adder. The input carry C0 must be equal to 1 when subtraction operations can be combined into one circuit with one common binary adder by including an exclusive – OR gate with each full adder Subtraction can be realized using an adder by controlling inputs to a parallel adder.

Fig.1.40(a) shows adder – subractor units using parallel adder

Considering the table, expressions for x and y can be obtained using K-map. The resulting expressions are

$x_1 = A_1$

$y_1 = B_1 \oplus M$ and

$C_1 = M$

These equations are implemented to obtain an adder. Subtractor logic diagram circuit and is shown in Fig. 1.40b.

M	x_i	y	C_i
0	A_i	B_i	0
1	A_i	B_i	1

M	A_i	B_i	C_i	D_i
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	1
1	0	0	0	0
1	0	1	0	1
1	1	0	1	0
1	1	1	1	1

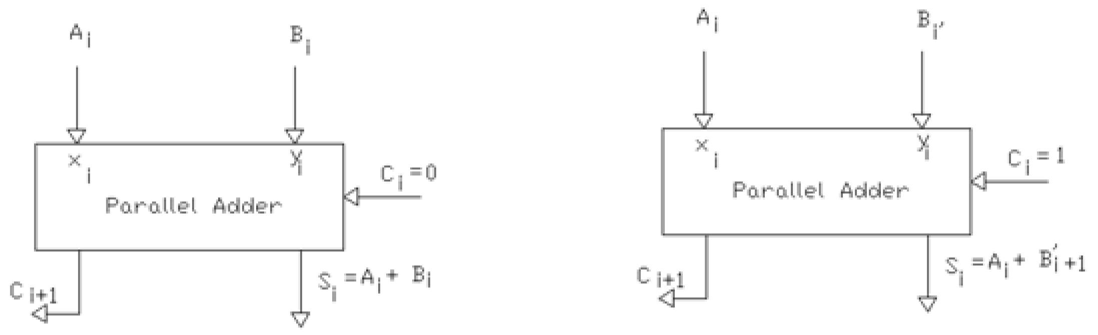


Figure 1.40 (a) Adder-Substractor units using Parallel adder

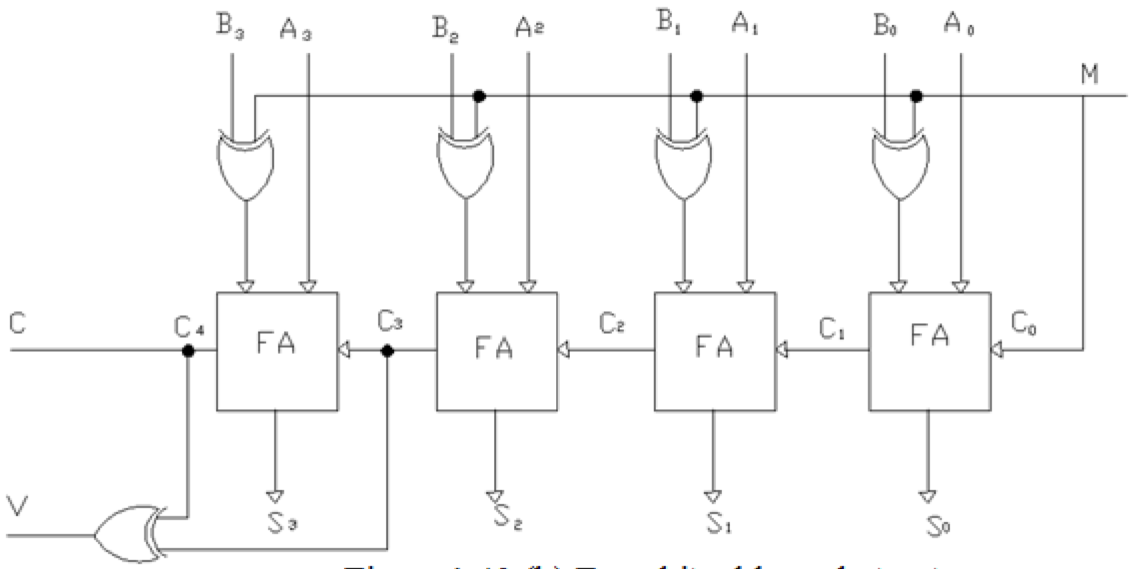


Figure 1.40 (b) Four-bit adder-substractor

MAGNITUDE COMPARATOR

The comparison of two numbers is an operation that determines whether one number is greater than, less than, or equalto the other number.

A magnitude comparator is a combinational circuit that compares two numbers A and B and determines their relative magnitudes The outcome of the comparison is specified by three binary variables that indicate whether A>B,A=B or a<B

Digital function designed by means of an algorithm-a procedure which specifies a finite set of steps that, if followed, give the solution to a problem we illustrate this method here by driving an algorithm for the design of a four –bit magnitude comparator.

The algorithm is a direct application of the procedure a person uses to compare the relative magnitudes of two numbers. consider two numbers, A and B, with four digits each. write the coefficients of the numbers in descending order of significance:

$$A = A_3A_2A_1A_0$$

$$B = B_3B_2B_1B_0$$

Each subscripted letter represents one of the digital the number are equal if all pairs of significant digits are equal: A3=B3, A2=B2, A1=B1, and A0=B0. When the numbers are binary, the digits are either 1 or 0, and the equality of each pair of bits can be expressed logically with an exclusive – NOR functions as

$$X_i = A_i B_i + A_i'B_i' \quad \text{for } I = 0,1,3$$

Where xi=1 only if the pair of bits in position I are equal (i.e., if both are 0)

The binary variable (A=B) is equal if all pairs of significant digits of the two numbers are equal.

To determine whether A is greatest or less than B, we inspect the relative magnitudes of pairs of significant digits starting from the most significant position if the two digit of a pair are equal, we compare the next lower significant pair of digits the comparison continues until a pair of unequal digits is reached. if the corresponding digit of A is 1 and that of B is 0, we conclude that corresponding digit of A is 0 and that B is 1, we have A<B. the sequential comparison can be expressed logically by the two Boolean function.

$$(A > B) = A_3 B_3' + x_3A_2B_2' + x_3A_2B_2' + x_3x_2A_1B_1' + x_3x_2x_1A_0B_0'$$

$$(A<B) = A_3'B_3 + x_3A_2'B_2 + x_2x_2A_1'B_1 + x_3x_2x_1A_0'B_0$$

The symbols (A>B) and (A<B) are binary output variables that are equal to 1 when A>B and A<B, respectively.

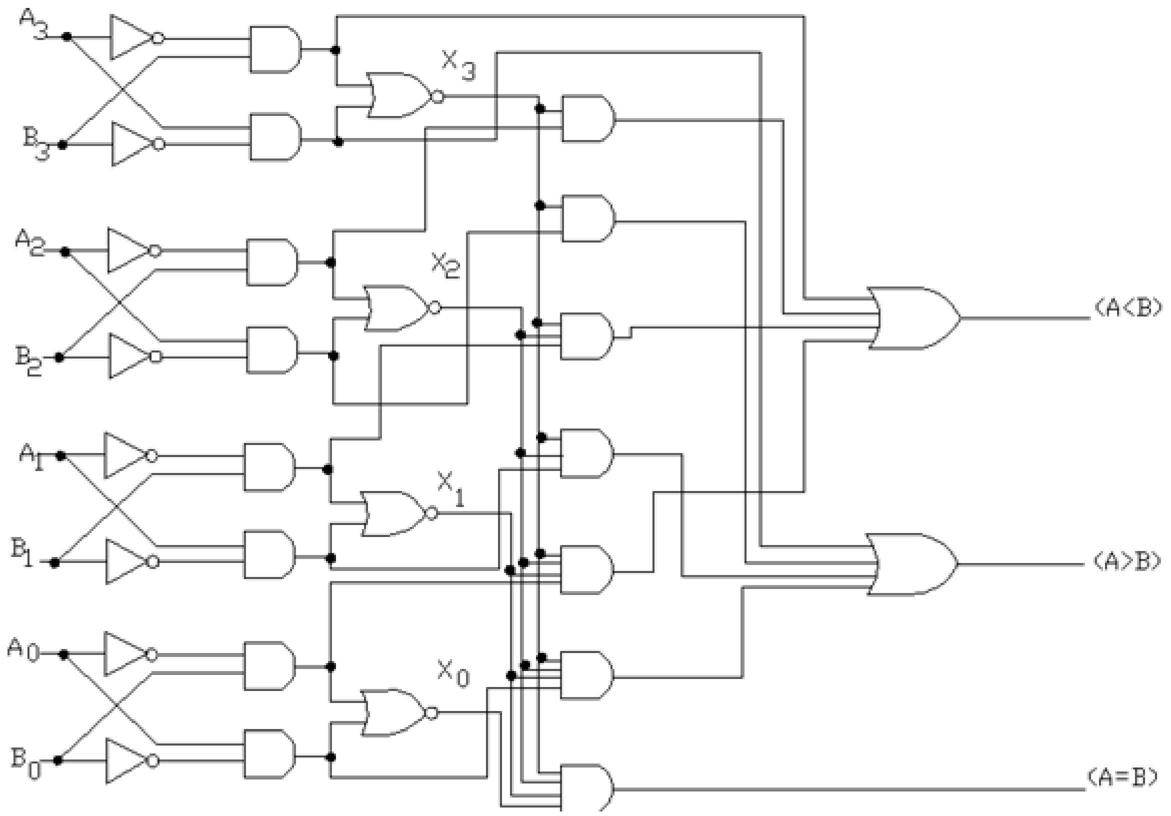
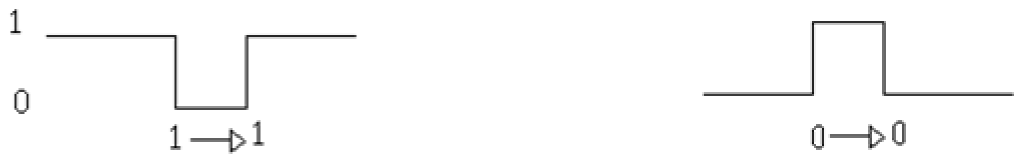


Figure 1.41 Four-bit magnitude comparator

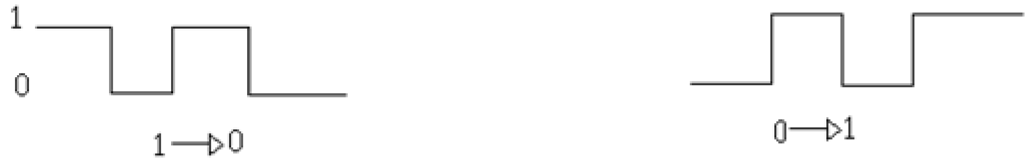
The logic diagram of the four-bit magnitude comparator is shown in fig. 1.41. The four x outputs are generated with exclusive NOR circuits and are applied to an AND gate to give the output binary variables (A=B).

HAZARDS

In asynchronous sequential circuit it is important that undesirable glitches on signals should not occur. the glitches on signals should not occur. The glitches caused by the structure of a given circuit and propagation delays in the circuit are referred to as hazards. Two types of hazards are illustrated in Figure 1.42



(a) Static hazard



(b) Dynamic of Hazard

Figure 1.42 Definition of hazards

A static hazard exists if a signal is supposed to remain at a particular logic value, As shown in figure 1.42 a, one type of static hazard is when the signal at level 1 is supposed to remain at 1 but dips to 0 for a short time Another type is when the signal is supposed to remain at level 0 but rises momentarily to1, thus producing a glitch

A different type of hazard may occur when a signal is supposed to change involves a short oscillation before the signal settles into its new level, as illustrated in figure 1.42b, then a dynamic hazard is said to exits.

Critical and non-critical race conditions:

A critical race occurs when the order in which internal variables are changed determines the eventual state that the state machine will end up in.

A non-critical race occurs when the order in which internal variables are changed does not alter the eventual state.

Static, dynamic, and essential race conditions:

Static race conditions

These are caused when a signal and its complement are combined together.

Dynamic race Conditions:

These result in multiple transitions when only one is intended. they are due to interaction between gates

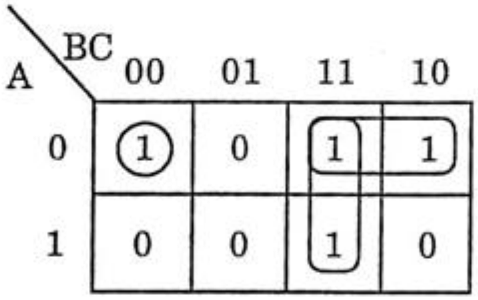
CIRCUIT SYNTHEIS USING GATES

Example : 1

Implement the function F = ∑ m {0,2,3,7} with minimal gates

SOULTION

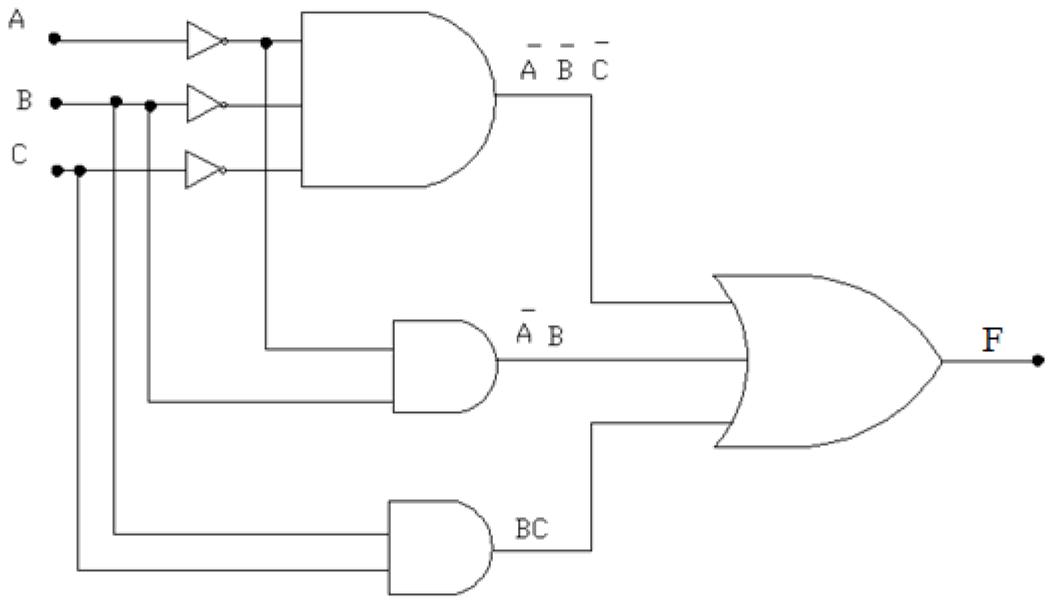
Step I



K map Simplification

$$\therefore F = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,B + B\,C$$

Step II



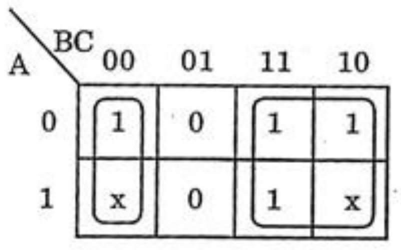
Implementation with minimal gates

Example : 2

Implement the function F = ∑ {0,2,3,7} with do not care 4 & 6 with minimal gates.

SOLUTION

Step I

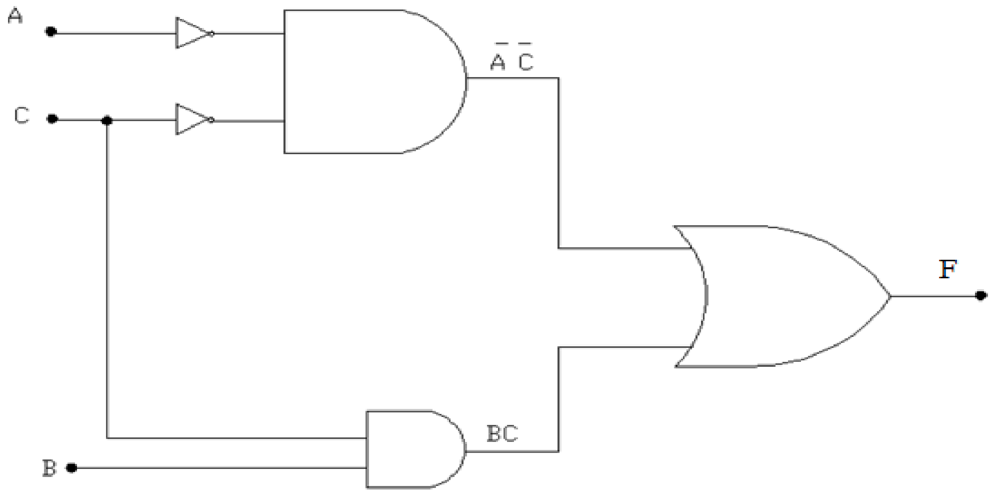


A \ BC	00	01	11	10
0	1	0	1	1
1	x	0	1	x

K map simplification

$$\therefore F = \overline{A}\,\overline{C} + B\,C$$

Step II



Implementation with mimimal gates

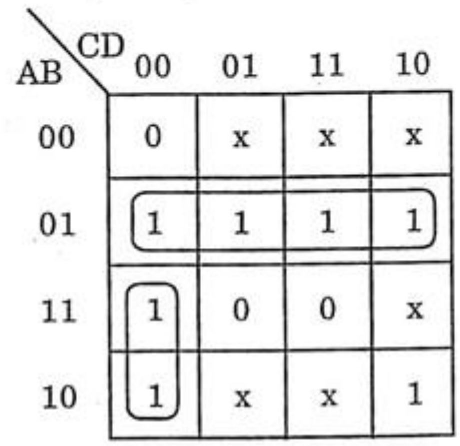
Example : 3 Implement the function

F(A,B,C,D) = ∑ m {4,5,6,7,8,12} + d{1,2,3,9,11,14} with only NAND gates

SOLUTION

Step I

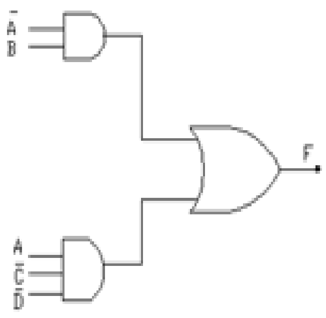
K map Simplification

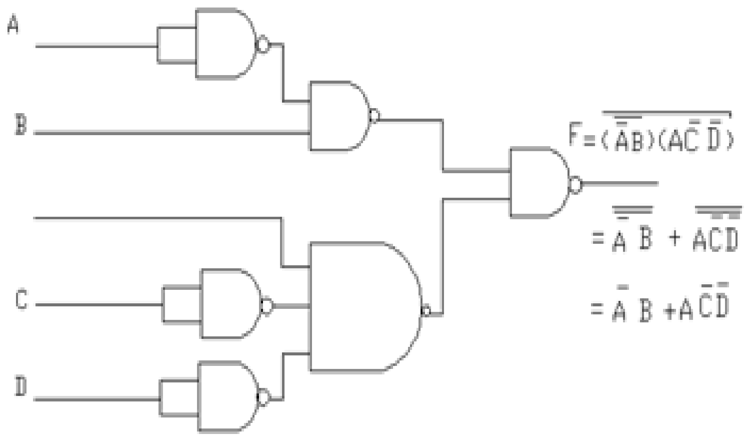


AB \ CD	00	01	11	10
00	0	x	x	x
01	1	1	1	1
11	1	0	0	x
10	1	x	x	1

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

Step II





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

Implementation

SYNTHESIS OF LOGIC FUNCTION USING MULTIPLEXER

Example : 1

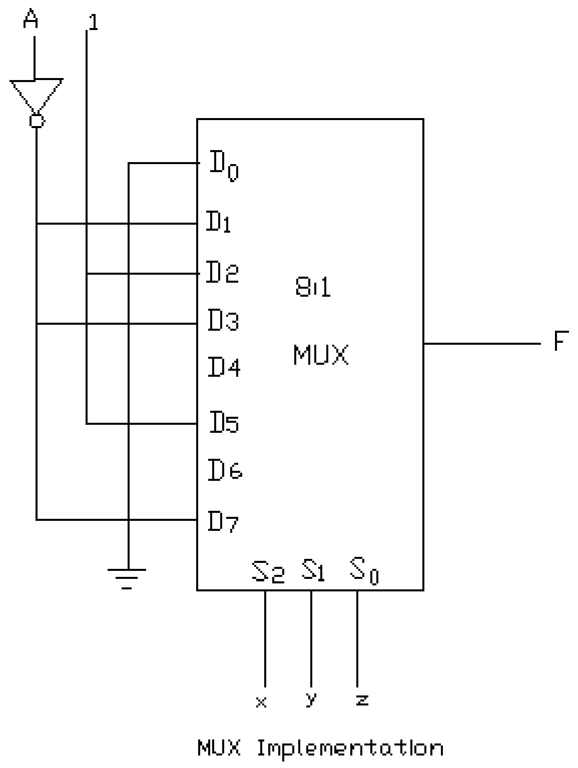
Implement the function f = ∑m{ 1,2,3,5,7,10,13} multiplexer

SOLUTION

Step I

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\overline{A}$	0	$\textcircled{1}$	$\textcircled{2}$	$\textcircled{3}$	4	$\textcircled{5}$	6	$\textcircled{7}$
A	8	9	$\textcircled{10}$	11	12	$\textcircled{13}$	14	15
	0	$\overline{A}$	1	$\overline{A}$	0	1	0	$\overline{A}$

Step II

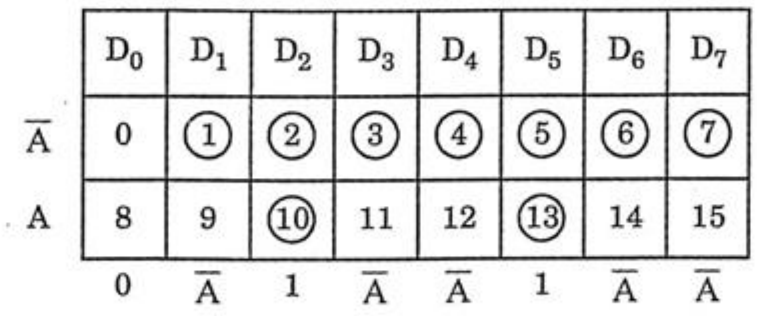


MUX Implementation

Example 2 : Implement the function F = ∑m{1,2,3,5,7,10,13} with don't care of 4 & 6 with multiplexer

SOLUTION Consider 4 & 6

Step I

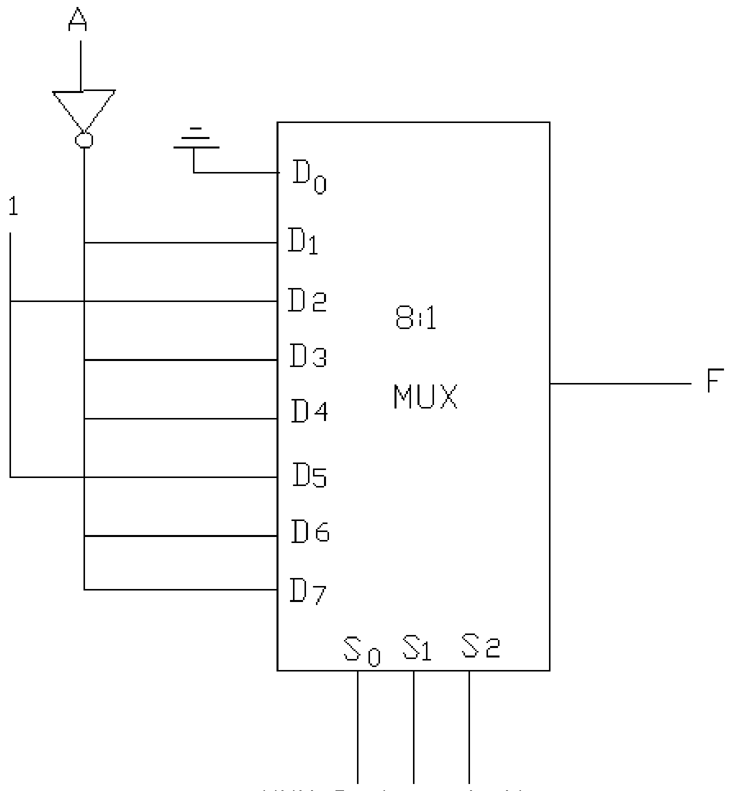


	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\overline{A}$	0	①	②	③	④	⑤	⑥	⑦
A	8	9	⑩	11	12	⑬	14	15
	0	$\overline{A}$	1	$\overline{A}$	$\overline{A}$	1	$\overline{A}$	$\overline{A}$

Here don't care as =1

Implementation Table

Step II

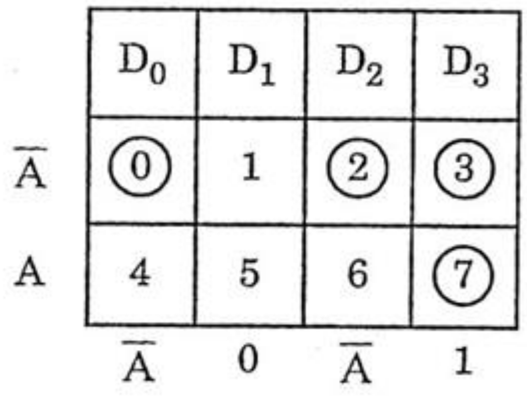


MUX Implementation

Example 3 : Implement the function F = ∑m {0,2,3,7} with mux

SOLUTION

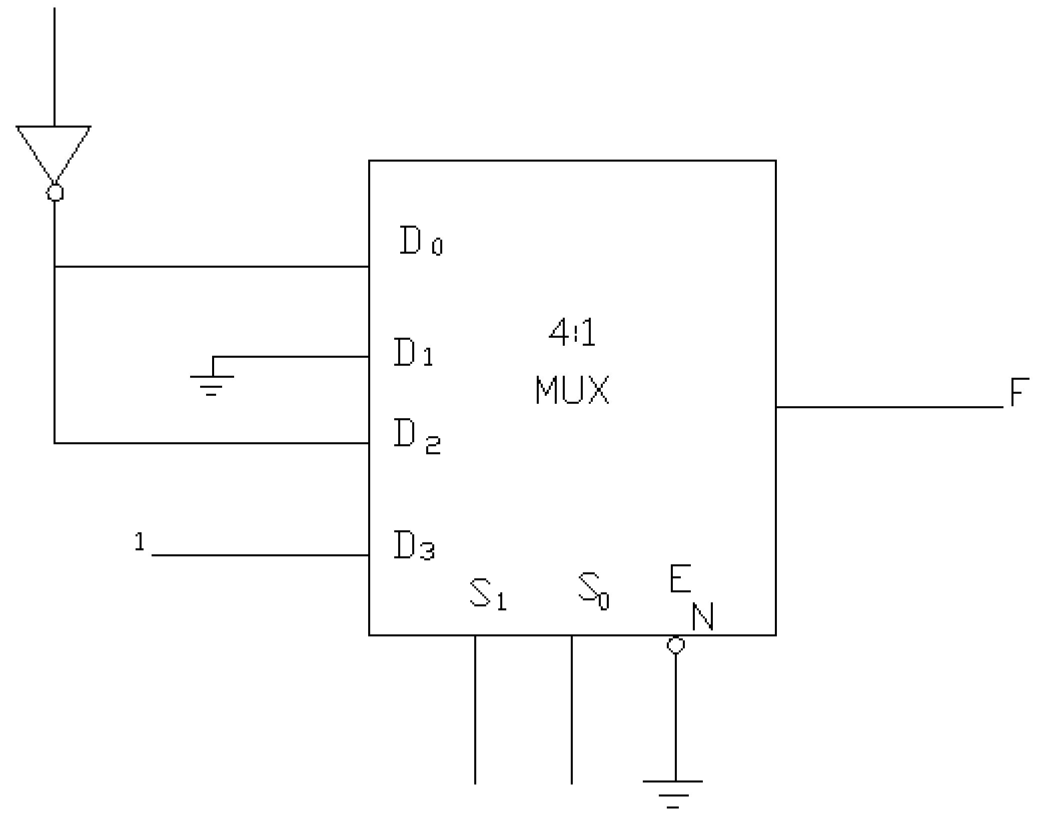
Step I



	D_0	D_1	D_2	D_3
$\bar{A}$	$\textcircled{0}$	1	$\textcircled{2}$	$\textcircled{3}$
A	4	5	6	$\textcircled{7}$
	$\bar{A}$	0	$\bar{A}$	1

Implementation Table

Step II



Example 4 : Implement the function F = ∑ m {0,2,3,7} with don't care 4 & 6 with mux

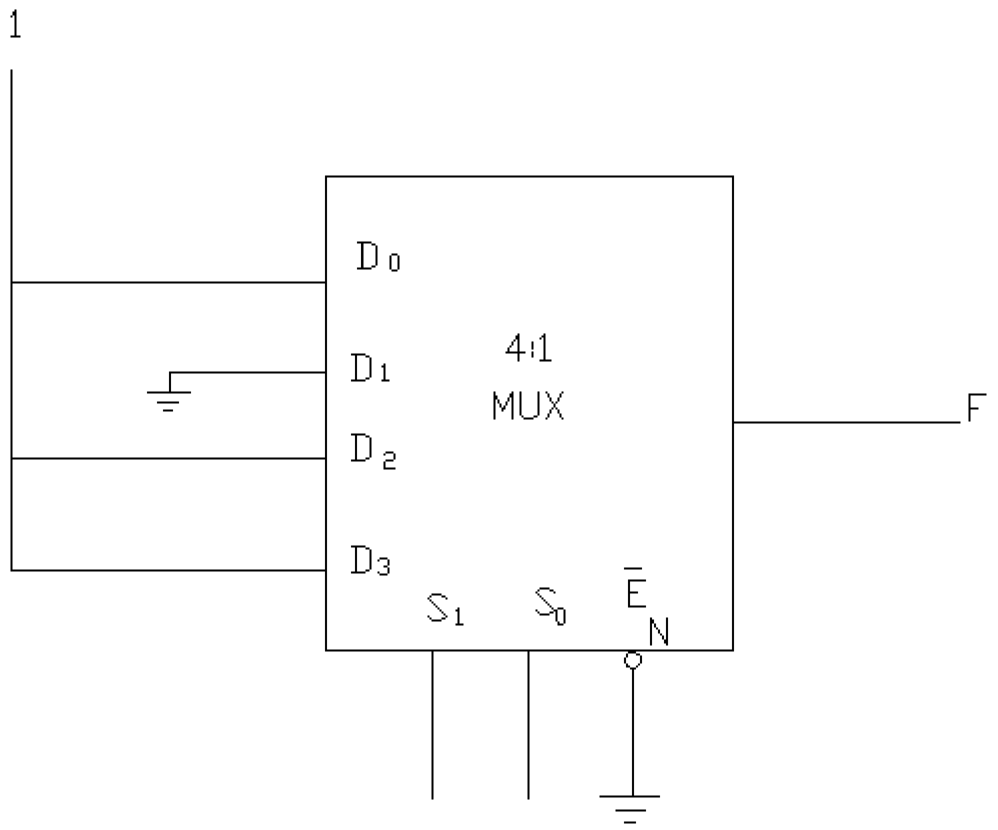
SOLUTION

Step I

	D_0	D_1	D_2	D_3
$\overline{A}$	⓪	1	②	③
A	④	5	⑥	⑦
	1	0	1	1

Here don't care as =1
Implementation Table

StepII



Example 5 :

Implement the following function using 4:1 mux F (A,B,C,D) = ∑ {0,1,2,4,6,9,12,14}

SOLUTION :

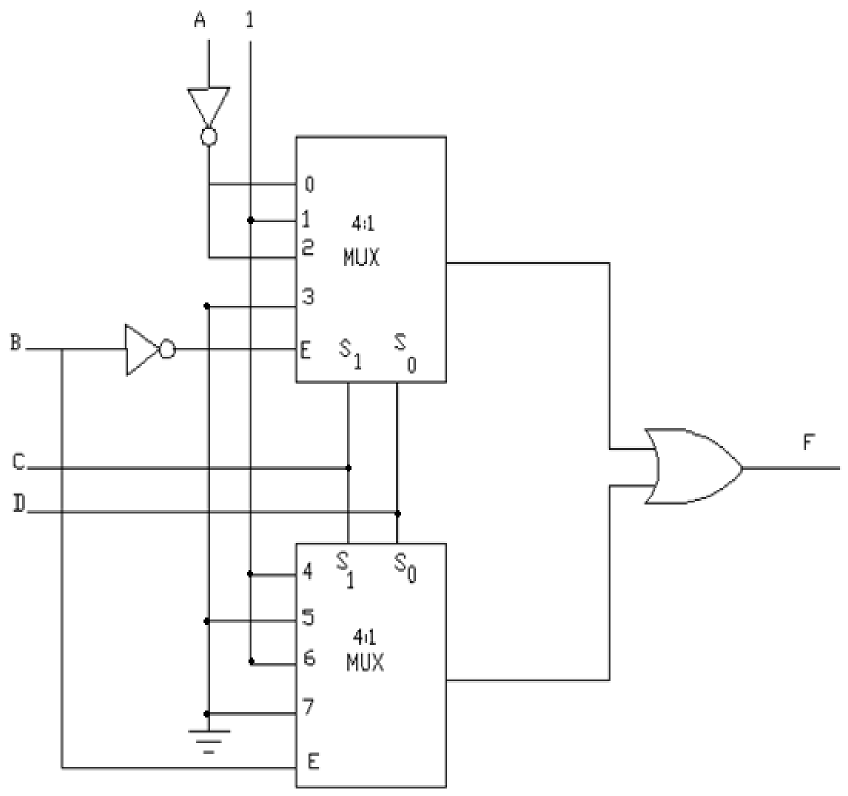
The function has four variables to implement this function, we require two 4:1 mux.

Step I

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\overline{A}$	⓪	①	②	3	④	5	⑥	7
A	8	⑨	10	11	⑫	13	⑭	15
	$\overline{A}$	1	$\overline{A}$	0	1	0	1	0

Implementation Table

Step II



UNIT II

2.1 VHDL FOR COMBINATIONAL CIRCUIT

Introduction to VLSI and its Design Process.

Introduction to CAD Tool and VHDL

logic circuits found in complex system's such as today's computers cannot be designed by manually They are designed using sophisticated CAD tools that automatically implement the synthesis techniques.

To design a logic circuit, a number of CAD tools are needed they are usually packaged together into a CAD system Cad system includes tools for the following tasks Design Entry Synthesis and Optimization, simulation and physical Design.

DESIGN ENTRY

The starting point in the process of designing a logic circuit is forming an idea of what the circuit is supposed to do and formulation of its general structure This is done manually by the designer the first step of this process involves entering into the CAD Systems. CAD is the description of the circuit being designed this stage is called design entry There are two design entry methods.

1. Using Schematic
2. Writing Source Code in a HDL

Schematic Capture

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. The word Schematic refers to a diagram of a circuit elements such as logic gates are depicted as graphical symbols and connection between circuit elements are drawn as lines.

A schematic capture tool uses the graphic symbols that represent gates of various types with different numbers of inputs from a library and the tool provides a graphical way of interconnecting the gates to create a logic network.

Writing Code in a HDL

A hardware Description Language (HDL) is similar to a computer programming language expect that an (HDL) is similar to describe hardware rather than a program to be executed on a computer

Two HDLs are IEEE Standards: VHDL (very High Speed integrated Circuit Hardware Description languages are mostly used in industry.

Design entry of a logic circuit is done by writing VHDL Code. Similar to the way in which large circuits are handled in schematic capture, VHDL code can be written in a modular way that facilitates hierarchical design VHDL design entry can be combined with other methods. for example, a schematic is described using VHDL

SYNTHESIS

Synthesis 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 HDL. Synthesis CAD tools generate efficient implementation of circuits from such specifications.

The process of translating or compiling, VHDL Code into a network of logic expressions that describe the logic functions needed to realize the circuit

The performance of a synthesized circuit can be assessed by physically constructing the circuit and testing it But, its behavior can also be evaluated by means of simulation.

FUNCTIONAL SIMULATION

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. it uses the logic expressions generated during synthesis and assumes that these expressions will be implemented with perfect gates through which signals propagate instantaneously the results of simulation are usually provided in the form of a timing diagram. The users can examine to Verify that the circuit operates as required

PHYSICAL DESIGN

After logic synthesis he the next step in the design flow is to determine exactly how to implement the circuits on a given chip. This step is often called Physical design. toolmap a circuit specified in the form of logic expressions into a realization

TIMING SIMULATION

A timing simulator evaluates the expected delays. of a designed logic circuit Its results can be used to determine if the generated circuit meets the timing requirements of the specification for the design.

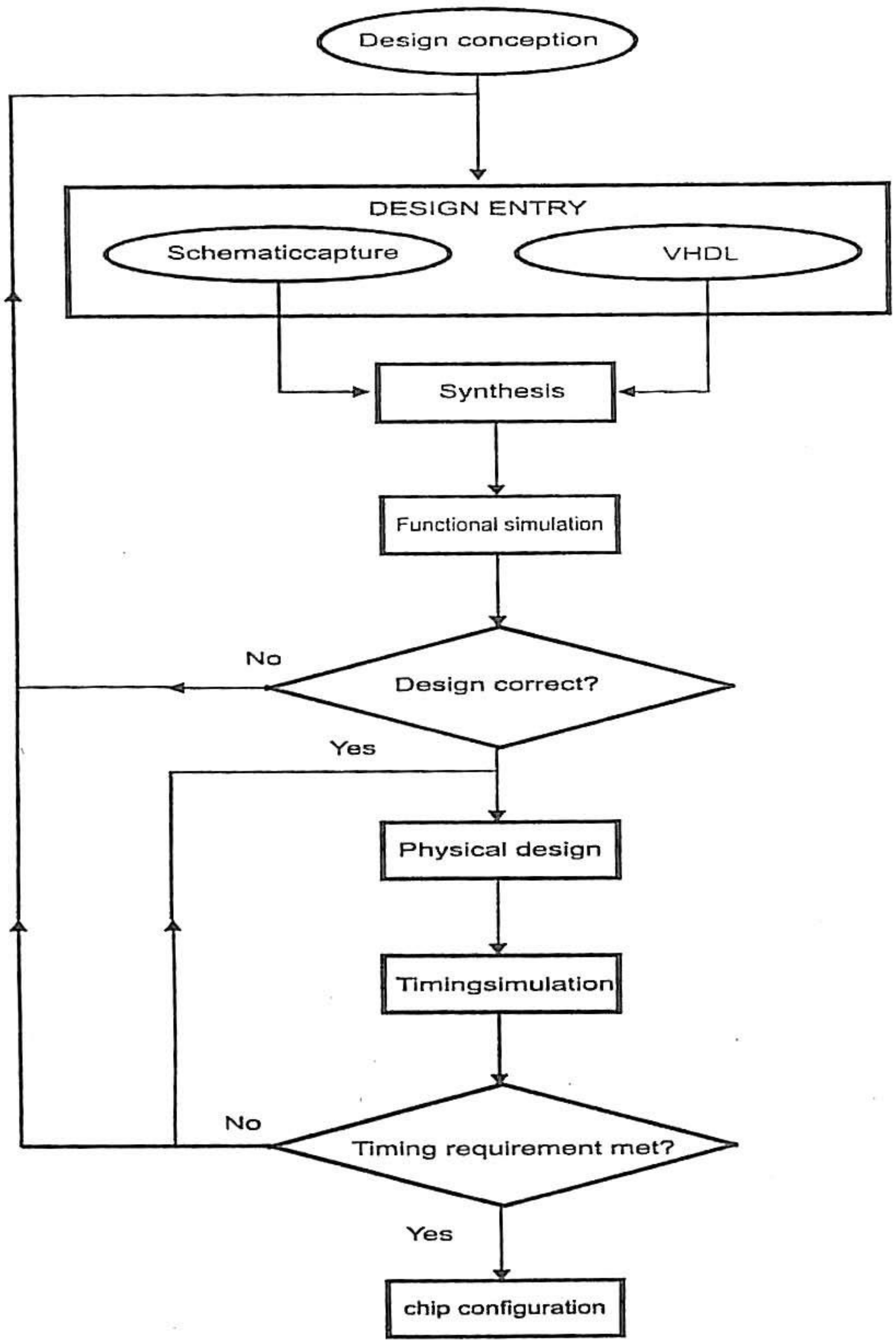


Figure 2.1 A typical CAD system

CHIP CONFIGURATION

When the designed circuit meets all requirements of the specification then the circuit is implemented on an actual chip this step is called chip configuration programming.

The CAD tool are the essential parts of a CAD system The complete design flow is shown in figure 2.1

INTRODUCTION TO VHDL

VHDL stands for very high-speed integrated circuit hardware description language used to model a digital system by dataflow, behavioral and structural style of modeling This language was first introduced in 1981 for the department of defense (DOD) under the VHSIC program In 1983 IBM, Texas instruments and Inter metrics started to develop this IEEE standardized the language

Describing a design

In VHDL an entity Is used to describe a hardware module

An entity can be described using,

1. Entity declaration
2. Architecture
3. Configuration
4. Package declaration
5. Package body

Let's see what are these?

1. Entity declaration

It defines the names, input signals and modes of a hardware module.

2. Architecture

It describes the internal description of design. each entity has least one architecture and an entity has at least one architecture and an entity can have many architecture can be described using structural, dataflow, behavioral or mixed style Architecture can be used to described a design at different levels of abstraction like gate level, register transfer level (RTL) or behavior level.

3. Configuration

If an entity contains many architectures and any one of the possible architecture binding with its entity is done using configuration it is used to bind the architecture body to its entity and a component with an entity

4. Package declaration

Package declaration is used to declare components, types, constants function and so on.

5. Package body:

Package body is used to declare the definitions and procedures that are procedures that are declared in corresponding package values can be assigned to constants declared in package body.

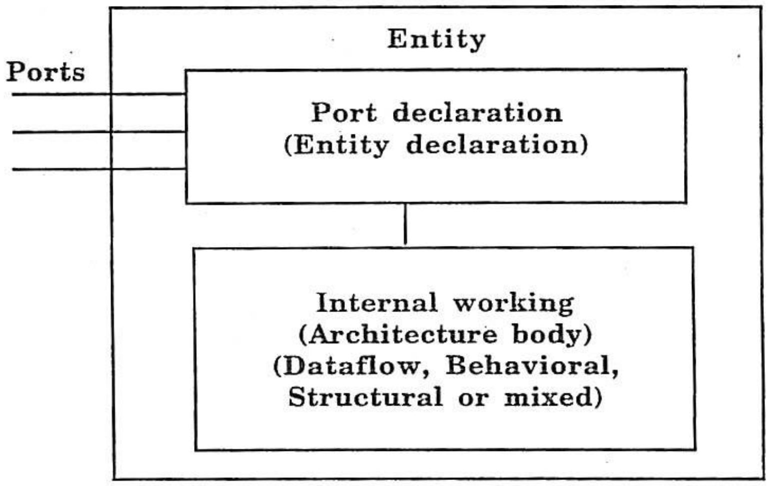
DIFFERENT LEVELS OF ABSTRACTIONS

The internal working of an entity can be defined using different modeling styles inside architecture body. They are

1. Dataflow modeling

2. Behavioral modeling (RTL Modeling)
3. Structural modeling

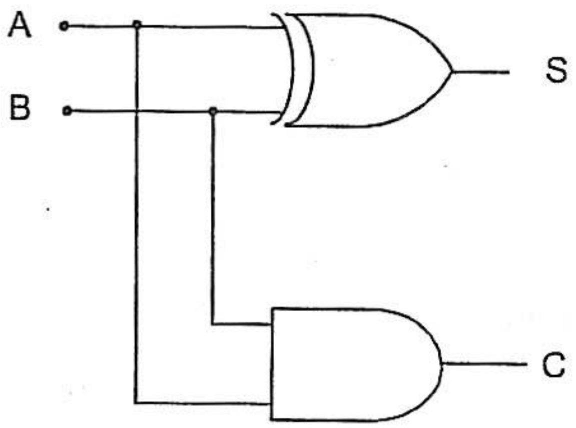
Structure of an entity



DATA FLOW MODELING

In this style of modeling, the internal working of an entity is implement using concurrent signal assignment

Let’s take half adder example which is having one XOR gate and a AND gate.



```
Library IEEE ;

use  IEEE. STD_LOGIC _1164. All ;

entity ha_en is

        port (A, B : in bit ; S, C : out bit) ;

end ha _en ;

architecture ha_ar of ha_en is

begin

                S<=A xor B ;

                C<=A and B;

end ha_ar
```

Here STD_LOGIC_ is IEEE standard. This defines a nine-value logic type, called STD_ULOGIC use is a keyword, which imports all the declarations from this package. the architecture body consist of concurrent signal assignments, which describes the functionality of the design whenever there is change is RHS, the expressions is evaluated and the value is assigned to LHS.

BEHAVIORAL MODELING

In this style of modeling, the internal working of an entity can be implemented using set of statements

It contains:

- ❖ Process statements
- ❖ Sequential statements
- ❖ Signal assignment statements
- ❖ Wait statements

Process statement is the primary mechanism used to model the behavior of an entity. it contains sequential statement, variable assignment (:=) statements or signal assignment (<=) statements etc. it may or may not contain sensitivity list If there is an event occurs on any of the signals in the sensitivitylist, the statements within the process is executed.

Inside the process the execution of statements will be sequential and if one entity is having two processes will be concurrent. At the end it waits for another event to occur.

```
        Library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity ha_beha_en is
            Port (
                       A: in BIT;
                       B: in BIT;
                       S: out BIT;
                        C: out BIT
                        );
end ha_beha__en;
architecture ha_beha_ar of ha_beha_en is begain
process_beh:process(A,B)
begain
       S<=A xor B;
        C<=A and B:
    end process process_beh:
end ha_beha_ar;
```

Here whenever there is a change in the value of A or B the process statements are executed.

Structural modeling:

The implementation of an entity is done through set of interconnected components.

If contains:

- ❖ Signal declaration
- ❖ Component instance
- ❖ Port maps
- ❖ Wait statements

Component declaration

Syntax

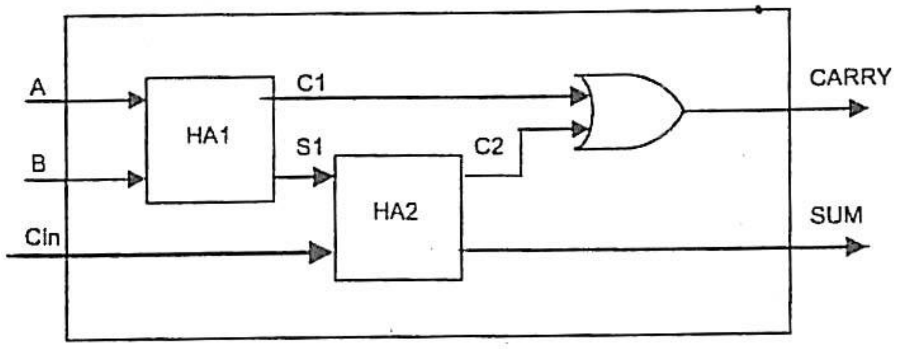
Component component_name [is]

List_of_interface ports;

end componets_name;

Before starting the component it should be declared using component declaration as shown above. component

Let's try to understand this by taking the example of full adder using 2 half adder and 1 OR gate.



```
Library IEEE;

USE IEEE.STD_LOGIC_1164. all;

entityfa_en is

port (A,B, Cin:inbit;SUM, CARRY: out bit);

endfa_en;

architecturefa_ar of fa_en is

componentha_en

port (A,B: in bit S,C: out bit);

end component

signal C1,C2,S1: bit;

begain

HA1: ha_en port map (A, B, SI, C1);

HA2: ha_en port map (S1, Cin, SUM, C2);

CARRY<=C1 or C2;

endfa_ar;
```

The program we have written for half adder in dataflow modeling is instantiated as shown above.ha_en is the name of the entity in data flow modeling. C1, C2, S1 are the signals used for internal connections of the component which are the declared using the keyword signal. Port map is used to connect different components as well as connect components to ports of the entity.

Component instantiation is done as follows.

```
component _label: component port map (signals_list)
```

Signal_list is the architecture signals which we are connecting to component ports. this can be done in different ways. What declared above is positional binding. One more type is the named binding The above can be written as,

```
HA1: ha_en port map (A=> A,B=>B,S=>S1, C=>C1);

HA2: ha_en port map (A=S1, B=> SUM, C=>C2);
```

VHDL STATEMENTS & ASSIGNMENTS

ASSIGNMENTS SATEMENTS

Assignments statements, which are called selected signal assignments, conditionals signal assignments, generate statements, if-then –else statements, and case statements.

SELECTED SIGNAL ASSIGNMENT

A selected signal assignments allows a signal to be assigned one of several values, based on a selection criterion Figure 2.2 shows how it can be used to describe a 2-to1 multiplexer. the entity named mux2to1, has the input w0,

W1, and s and the output f. the selected signal assignments begins with the key word WITH, which specifies that's is to be used for the selection criterion. the two WHEN clause

State that if assigned the value of w1. WHEN clause that selects w1 uses the word OTHERS, instead of the value 1. This is required because the VHDL syntax specifies that a WHEN clause must be included for every possible value of the selection signals s.

```
LIBRARY ieee;

USE iee.std_logic_1164.all;

ENITY 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;
```

Figure 2.2 VHDL code for a 2-to 1 multiplexer

Since it has the STD_LOGIC type, s can take the values 0,1,z, and other the keyword Others provides a convenient way of accounting for all logic values that are not explicitly listed in a WHEN clause.

CONDITIONAL SIGNAL ASSIGNMENT

Similar to the selected signal assignment, a conditional signal assignment allows a signal to be a set to one of several values Figure 2.3 shows the 2-to 1 multiplexer entity. it uses a conditional signal assignment to specify that f is assigned the value of wo when s=0, or else f is assigned the value of w1.

```
        LIBRARY ieee;
        USE ieee.std_logic_1164 all;
        ENITY mux2to1 IS
        PORT (w0, w1,s          :       IN STD_LOGIC;
        ff                      :       OUT STD_LOGIC);
        END mux 2 to 1;

        ARCHITECTURE Behavior OF mux2 to 1 IS
        BEGIN

                f<= WO EHEN S='0' ELSE W1;

        END Behavior
```

Figure 2.3 Specification of a 2-to-1 multiplexer using a conditional signal assignment

In this small example the conditional signal assignment has only one WHEN clause

GENERATE STATEMENTS

VHDL Provides a feature called the FOR GENERATE statements. the generate statement must have a label, so we have used the label G1 in the code. the loop instantiates four copies of mux4to1 component, using the loop index I in the range from 0 to 3 the variable I is not explicitly declared in the code; it is automatically defined as a loop variable whose scope is limited to the FOR GENERATE statement.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work. mux4to1_package.all;
ENITITY mux16to1,
PORT        (w: IN STD_LOGIC_VECTOR (0 TO15);
             s  : IN STD_LOGIC_VECTOR (3DOWN TO 0);
             f  :  (OUT_STD_LOGIC);
END mux16to1,
ARCHITECTURE Structure OF mux 16 to 1 IS
SIGNAL m: STD_LOGIC_VECTOR(0TO3);
BEGAIN
G1:    FOR in 0 TO 3 GENERATE
Muxes: mux 4 to 1 PORT MAP(
w(4*1), w(4*i+3), s(1 DOWNTO 0), m(i));
```

```
END GENERATE;
mux5; mux4 to 1 PORT MAP (m(0), m(1), m(2), m(3), s(3 DOWNTO 2);
END structure;
```

Figure 2.3 code for a 16-to1 multiplexer using a generate statement

In addition to the FOR GENERATE statement, VHDL provides another type of generate statement called IF GENARATE Figure 2.4 illustrates the use of both types of generate statements the decoder inputs are the four-bit signals w, the enable is En, and the outputs are the 16-bit signal y.

Following the component declaration for the dec2to4 sub circuit, the architecture defines the signal m, which represents the outputs of the 2-to4 decoder component are instantiated by the FOR GENERATE statement in each iteration of the loop, the statement labeled Dec_ri instantiates a dec2to4 component that corresponds to one of the dec2to4 component with data inputs w1 and w0, enable input m0, and outputs y0,y1,y3,. the other loop iterations also use data inputs w1w0, but use different bits of m and y.

The IF GENERATE statement, labeled G2, instantiates a dec2to4 component in the last loop iteration, for which the condition i=3 is true. this component represents the 2-to4 decoder where it has the two-bit data inputs w3 and w2, the enable En, and*the

```
        LIBBRARY ieee;
        USE ieee.std_logic_1164 all;
        ENTITY dec4to16 IS
        PORT (w : IN STD_LOGIC_VECTOR(3DOWNTO 0);
        En : IN STD_LOGIC;
        y   : OUT STD_LOGIC_VECTOR (0 TO 15);
END dec4to16 IS
ARCHITECTURE Structure OFdec4to16 IS
```

```
        COMPONENT dec2to4
        PORT (w : IN STD_LOGIC_VECTOR(1DOWNTO 0);
        En : IN STD_LOGIC;
        y   : OUT STD_LOGIC_VECTOR (0 TO 3));
        END COMPONENT;
        SIGNAL m: STD_LOGIC_VECTOR (0 TO3);
        BEGIN
        G1:FOR 1 IN 0 TO 3 GENERATE
        Dec_ri:dec2to4 PORT MAP (W(1 DOWNTO 0, M(i),y(4*Ito4*i+3);
        G2: IF I=3 GENERATE
        Dec_left; dec2to4 PORT MAP w (idowntoi-1), En, m);
                END GENERATE;
        END GENERATE;
END Structure;
```

Figure 2.4 Hierarchical code for 0 4-to-16 binary decoder

*The outputs m0,m1,m2, and m3

The generate statements in figures 2.9 and 2.10 are used to instantiate components. Another use of generate statements is to generate a set of logic equations.

CONCURRENT AND SEQUENTIAL ASSIGNMENT STATEMENTS

We have introduced several types of assignment statement; logic or arithmetic expressions, selected assignment statements, and conditional assignment statements. All of these statements share the property, that the order in which they appear in VHDL code does not affect the meaning of the code Because of this property, these statements are called the concurrent assignment statements.

VHDL also provides a second category of statements, called sequential assignment statements, for which the ordering of the statements, may affect the meaning of the code we will discuss two types of sequential assignment statements, called if-then—else statements and case statements VHDL requires that the sequential assignments statements placed inside another type of statement, called a process statement.

PROCESS STATEMENT

Figures 2.2 & Fig.2.3 show two ways of describing a 2-to-1 multiplexer, using the selected and conditional signal assignments the same circuit can also be described using an if-then-else statement, but this statement must be placed inside a process statement figure 2.5 shows the code using process statement the process statement, or simply process, begins with the PROCESS keyword, followed by a sensitivity list. for a combinational circuit like the multiplexer, the sensitivity list includes all input signals that are used inside the process the process statement is translated by the VHDL compiler into logic equations in the figure the process consists of the single if-then-else statement that describes the multiplexer function. thus the sensitivity list comprises the data inputs, w0, and w1, and the select input s.

In general, there is a number of statement inside a process Using VHDL, when there is a change in the value of any signal in the value of any signal in the process's sensitivity list, then the process becomes active.

Once active, the statements inside the process are evaluated in sequential order. Any assignments made to signals inside the process evaluated if there are multiple assignment to the same signal, only the last one has any visible effect.

```
LIBRARY ieee;

USE ieee.std_logic-1164 all;

ENITITY mux2to 1 IS

        PORT (w0, w1, s     :IN STD_LOGIC

                        f         :OUT STD_LOGIC);

END mux2to1;
```

```
ARCHITECTURE Behavior OF mux2to1 IS
        BEGIN
                IF='0' THEN
                f<w0;
                ELSE
                F<w1;
                END IF;
        END PROCESS;
```

Figure 2.5 A2to1 multiplexer specified using the if-then-else statement

CASE STATEMENT

A case statement is similar to a selected signal assignment the case statement has a selection signal and includes WHEN clauses for various valuations of this selection signal. figure 2.6 shows how the case statement can be used for describing the 2-to1 multiplexer circuit the CASE keyword, which specifies that s to be used as the selection signal. the first WHEN clause specifies, following the=> symbol, the statements that should be evaluated when s=0. in this example only statement evaluated when s=0 is f<=w0 The case statement must include a WHEN clause for all possible valuation of the selection signal. hence the second WHEN clause, which contains f<=w1, uses the OTHERS keyword.

```
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
        PROCESS (w0, w1, s)
        BEGIN
                CASE s IS
                        WHEN ‘0’=>
                                f<=w1;
                END CASE;
        END PROCESS;
END Behavior;
```

Figure 2.6 A case statement that represents a 2-to-1multiplexer.

VHDL OPERATERS

In this section we discuss the VHDL operators, that are useful for synthesizing logic circuits. Table lists these operators in groups that reflect the type of operations performed.

Operator category	Operator symbol	Operation performs
Logical	And OR NAND NOR XOR XNOR NOT	AND OR Not AND Not OR XOR Not XOR NOT
Relational	= /= > < >= <=	Equality Inequality Greater than Less than Greater than or equal to Less than or equal to
Arithmetic	+ - * /	Additional Subtraction Multiplication Division
Shift and Rotate	SLL SRL SLA ROL ROR	Shift left logical Shift right logical Shift left arithmetic Rotate left Rotate Right

To illustrate the results produced by the various operators, we will use three-bit vectors A(2DOWNTO 0),B(2DOWNTO 0), and C(2DOWNTO 0).

LOGICAL OPERATORS

The logical operators can be used with bit and Boolean types of operands. the operands can be either signal-bit scalars or inultibit vectors for example, the statement

C<=NOT A;

Produces the result c2=a2. c1=a1 and c0 = a0where a and c are bits of the vectors A and C.

The statement

C< A AND B;

Generates c2=a2. c2, c1=a1.b1, and c0=a0.b0. the other operators lead to similar evaluations.

RELATIONAL OPERATORS

The relational operators are used to compare expressions. The results of the comparison is TRUE or FALSE. The expressions that are compared must be of the same type For example, if A=010 the A>B evaluates to TRUE, AND B/=”010 evaluates to FALES.

ARITHMETIC OPERATORS

Arithmetic Operators perform standard arithmetic operations Thus

c<A+B;

Puts the three-bit sum of A plus B into C. The operation

CONCATENATE OPERATOR

This operator concatenates two or more vectors to create a large vector. For example,

D<= A & B ;

Defines the six – bit vector D = a_2a_1 $a_0b_2b_1b_{0.}$ Similarly , the concatenation

E < = “ 111” & A &”00”

Produces the eight – bit vector E = $111a_2a_1a_000$.

SHIFT AND ROTATE OPERATORS

A vector operand can be shifted to the right or left by a number of bits specified as a constant. When bits are shifted ; the vacant bit positions are filled with 0s. For example,

B< = A SLL 1 ;

Results in $b_2 = a_1$, $b_1 = a_0$ and $b_0 = 0$. Similarly,

B < = A SRL 2 ;

Yields $b_2 = b_1 = 0$ and $b_0 = a_2$

The arithmetic shift left, SLA, has the same effect as SLL. But, the arithmetic shift right, SRA, performs the sign extension by replicating the sign bit into the positions left vacant after shifting . Hence.

B < = A SRA 1 ;

Gives $b_2 = a_2$, $b_1 = a_2$, and $b_0 = a_1$.

An operand can also be rotated, in which case the bits shifted out from one end are placed into the vacated positions at the other end. For example.

B<=A ROR 2 ;

Produces $b_2 = a_1$, $b_1 = a_0$ and $b_0 = a_2$.

OPERATOR PRECEDENCE :

Operators in different categories have different precedence. Operators in the same category have the same precedence, and are evaluated from left to right in a given expression. It is a good practice to use parentheses to indicate the desired order of operations in the expression. To illustrate this point, consider the statement.

S<=A+B+C+D;

Which defines the addition of four vector operands. The VHDL complier will synthesize a circuit as if the expression was written in the form (A+B)+C)+D, which gives a cascade of three adders so that the final sum will be available after a propagation delay through three adders. By writing the statement as.

S<=(A+B)+(C+D) ;

They synthesized circuit will still have three adders, but since the sums A+B and C+D are generated in parallel, the final sum will be available after a propagation delay through only two adders.

Table groups the operators according to their functionality. It shows only those operators that are used to synthesize logic circuits. The VHDL Standard specifies additional operators, which are useful for simulation and documentation purposes. All operators are grouped into different classes.

2.2 VHDL CODE

VHDL CODE FOR AND GATE :

```
Library ieee ;

Use ieee. Std _logic_1164.all ;

Entity and Gate is

        Port (A,B ; in std_logic)

        F : out std_logic) ;

End and Gate ;
```

```
Architecture func of and Gate is begin
        F<= A and B ;
End func ;
```

VHDL CODE FOR OR GATE :

```
Library ieee ;
Use ieee. Std _logic_1164 all ;
Entity or Gate is
        Port (A, B : in std_logic ;
                F : out std _logic);
end or Gate ;
architecture func of orGate is
begin
        F< = A or B ;
end func ;
```

VHDL CODE FOR NAND GATE :

```
Library ieee ;
Use ieee.std_logic_1164 all ;

Entity nandGate is
        Port (A, B : in std_logic ;
                F : out std_logic) ;
end nand Gate ;
```

```
architecture func of nandGate is

begin

        F < = A nand B ;

end func ;
```

VHDL CODE FOR NOR GATE :

```
Library ieee ;

use ieee. Std_logic _11164. All ;

entity  norGate is

        port (A, B : in std_logic ;

                F : out std_logic) ;

end nor Gate :

architecture func of norGate is

begin

        F < = A nor B ;

End Func;
```

<u>VHDL CODE FOR 8:1 MUX</u>

```
library  IEEE ;

use IEEE. STD_LOGIC _1164. ALL ;

use IEEE. STD_ LOGIC _ARITH. ALL ;

use IEEE. STD_LOGIC _UNSIGNED. ALL ;

Entity mux IS

Port( s: in _ std_logic _ vector (2 downto 0);
     inp: in _ std_logic _ vector (7 downto 0);

op:out std_ logic );

END Entity mux;

Architecture mux OF  mux IS

BEGIN

PROCESS ( s,inp)

BEGIN

CASE s IS

WHEN “000”=>OP<=INP(0);

WHEN “001”=>OP<=INP(1);

WHEN “010”=>OP<=INP(2);

WHEN “011”=>OP<=INP(3);

WHEN “100”=>OP<=INP(4);

WHEN “101”=>OP<=INP(5);

WHEN “110”=>OP<=INP(6);

WHEN others =>op<=inp(7);

END case;

END PROCESS;

END ARCHITECTURE mux;
```

VHDL CODE FOR 4:1 MUX

```
library  IEEE ;

use IEEE. STD_LOGIC _1164. ALL ;

use IEEE. STD_ LOGIC _ARITH. ALL ;

use IEEE. STD_LOGIC _UNSIGNED. ALL ;

Entity mux1 IS

Port ( I;in std_Logic _ vector 3 downto 0 );

        Y: out std_ logic);

END mux1;

ARCHITECTURE Behavioral OF mux1 IS

BEGIN

PROCESS (I,S0,S1)

BEGIN

IF (S1=”0” and S0 =’0’) THEN

Y<=I0;

ELSIF (S1=”0” and S0 =’1’) THEN

Y<=I1;

ELSIF (S1=”1” and S0 =’1’) THEN

Y<=I2;

ELSIF (S1=”1” and S0 =’1’) THEN

Y<=I3;

END If;
END PROCESS;

END Behavioral;
```

VHDL Code for Encoder (4 ; 2)

```
library  IEEE ;

use IEEE. STD_LOGIC _1164. ALL ;

use IEEE. STD_ LOGIC _ARITH. ALL ;

use IEEE. STD_LOGIC _UNSIGNED. ALL ;


entity encod is

Port (a : in STD_LOGIC _VECTOR (3 downto 0) ;

B : out STD_LOGIC_VECTOR (1 down to 0)) ;

end encod ;


architecture Behavioral of encod is

begin

process(a)

begin

if (a(0) = ‘1’) then

b<= “00”;

elsif (a(1) = ‘1’) then

b<=”01”;
```

```
elsif (a(2) = ‘1’) then

b<=”10” ;

elsif (a(3) = ‘1’) then

b<=”11” ;

end if ;

end process ;

end Behavioral ;
```

VHDL CODE for 1 to 4 DEMULTIPLEXER

```
library  IEEE ;

use IEEE. STD_LOGIC _1164. ALL ;

use IEEE. STD_ LOGIC _ARITH. ALL ;

use IEEE. STD_LOGIC _UNSIGNED. ALL ;

entity DeMUX is

port( X: in std_logic;

sel:in std_logic_vector (1 downto 0);

A: out std_logic;

B: out std_logic;

C: out std_logic;

D: out std_logic;

end DeMUX;
```

```
architecture behaviour of DeMUX is

begin

process(sel, X)

begin

case sel is

when “00”=>

A <=X;

B <=’0’;

C <=’0’;

D <=’0’;

When “01” =>

B <=X;

A <=0;

C <=’0’;

D <=’0’;

When “10”=>

C <=X;

A <=’0’;

B <=’0’;

D <=’0’;

When others=>

D <=X;
```

```
A <=’0’;

B <=’0’;

C <=’0’;

end case;

end process;

end behaviour;
```

VHDL code for 8:3 ENCODER Logic Program

VHDL program for “8:3 Encoder” behavioral design

```
library IEEE ;

use IEEE STD_LOGIC _1164. ALL ;

use IEEE.STD_LOGIC_ARITH.ALL ;

use IEEE> STD_LOGIC_USIGNED.ALL ;

entity ENC2 is

                    Port (S : in std_logic ;

                              T : in  std_logic ;

                              U : in std_logic ;

                              V : in std_logic ;

                              W : in std_logic ;

                              Y : in std_logic ;

                              Z : in std_logic ;

                              OUT0 : out std_logic ;

                              OUT1 : out std_logic ;
```

```
                                        OUT 2 : out std_logic) ;

end ENC2 ;

architecture Behavioral of ENC2 is begin

                process (S,T, U, V, W, X, Y, Z)

                begin

                OUT0<=T OR V OR X OR Z ;

                OUT1 < = U OR V OR Y OR Z ;

                OUT 2 < = W OR X OR Y OR Z ;

                end process ;

end Behavioral ;
```

VHDL CODE FOR DECODER (2:4)

```
library IEEE ;

use IEEE. STD_LOGIC_1164.ALL ;

use IEEE. STD_LOGIC_ARITH.ALL ;

use IEEE.STD_LOGIC_UNSIGNED. ALL ;


entity decod 1 is

Port  (10 : in STD_LOGIC)

I1: in STD_LOGIC

En : in STD_LOGIC ;

Y : out STD_LOGIC_VECTOR (3 downto 0)) ;

end decod 1 ;
```

```
architecture Behavioral of decod 1 is

begin

process (I0, I1, En)

begin

if( En = ‘1’)

then

Y(0) < = (not I0) and (not I1) ;

Y(1)<= (not I0) and  I1 ;

Y(2)<= I0 and (not I1) ;

Y(3) <=I0 and I 1 ;

else

Y < = “0000” ;

end  if ‘

end  process ;

end Behavioral ;
```

VHDL CODE 3:8 DECORDER :

```
Entity decorder 3 x 8 is

Port (ctrl : in std_logic_vector (2 downto 0) ;

z : out std_logic_vector (7 downto 0) ;

end decoder 3x8 ;
```

```
architecture dec3x8_Dflow of decoder 3x8 is

begin

z<= “0000001”  when ctrl = “000” else

“00000010” when ctrl = “010” else

“000000100” when ctrl = “010” else

“00001000” when ctrl = “011” else

“00010000” when ctrl = “100” else

“00100000” when ctrl = “101” else

“01000000” when ctrl = “110” else

“10000000” ;

end dec 3x8 Dflow ;
```

VHDL CODE FOUR BIT ADDER / SUBTRACTOR --

```
-- This is the XOR gate

library ieee ;

use ieee. Std_logic_1164. All ;

--

entity xorGate is

        port (A, B ; in std_logic) ;

                F : out std_logic) ;

end xorGate ;

--

Architecture func of xorGate is
```

```
Begin

          F <=A xor B ;

end func ;

--

-- Now we build the four bit Adder Subtractor

Library ieee ;

use ieee. Std_logic_1164.all ;

entity adder Subtract or us

          port (mode                    :         in std_logic ;

          A3, A2, A1, A0                :         in std_logic ;

          B3, B2, B1, B0                :         in std_logic ;

          S3, S2, S1, S0                :         out std_logic ;

                    Court , V           :         out std_logic ) ;

end  adder Subtractor ;

-- Structural architecture

architecture struct of adder Subtractor is

          component xorGate is                    -XOR component

                    port (A, B : in std_logic ;

                              F : out std_logic ) ;

          end  component ;

          component Full_Adder is                 - FULL ADDER
```

```
component
          port (X, Y, Cin : in std_logic ;
                    sum, Cout : out std_logic) ;
          end component ;
--        interconnecting wires
          signal C1, C2, C3, C4 ; std_logic – intermediate carries
          signal xor 0, xor1, xor2, xor3, : std_logic ; - xor outputs
begin
          GX0 : xorGate port map (mode, B0, xor 0) ;
          GX1 : xorGate port map(mode, B1, xor 1) ;
          GX2 : xorGate port map(mode, B2, xor 2) ;
          GX3 : xorGate port map (mode, B3 , xor 3) ;


          FA0 : Full_Adder port map (A0, xor 0, mode, S0, C1) ; - S0
          FA1: Full_Adder port map (A1, xor 1, C1, S1, C2) ; - S1
          FA2: Full_Adder port map (A2, xor 2, C2, S2, C3) ; - S2
          FA3: Full_Adder port map (A13, xor 3, C3, S3, C4) ; - S3
          Vouut : xorGate port map (C3, C4, V) ;   -         V
          Cout <=C4                                -         Cout
End struct :
```

VHDL CODE FOR FOUR BIT ADDER

```
LIBRARY ieee ;
USE ieee. Std_logic _1164. all ;
```

Output of adder 4
Network is shown here

```
ENTITY adder 4 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 adder 4 ;
```

Intermediate Signals
Shown Here-these are
signal used in the
logic circuit

```
ARCHITECTURE Structure OF adder 4 IS
          SIGNAL c1, c2, c3 : STD_LOGIC ;
          COMPONENT fulladd

                                        PORT (Cin, x,y : IN STD_LOGIC ;
                                                  S, Cout : OUT STD _LOGIC) ;
```

Intermediate
outputs
are
Specified
in the

```
END COMPONENT ;
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 (C3, x3, y3, s3, c4) ;
Cin = > c3, Cout => Cout , x = >x3, y=>y3, s=>s3) ;
END Structure ;
```

Figure 2.9. VHDL Code for a four – bit adder

VHDL CODE FOR COMPARATOR

```
LIBRARY ieee ;
USE ieee. Std_logic_1164.all ;
USE work. Fulladd_package.all ;

ENTITY comparator IS
        PORT (X, Y : IN STD _LOGIC _VECTOR (3DOWNTO 0) ;
        V, N, Z     : OUT STD_LOGIC

END comparator ;
ARCHITECTURE Structure OF comparator IS

        SIGNAL S : STD_LOGIC _VECTOR (3 DOWNTO 0) ;
        SIGNAL C : STD_LOGIC_VECTOR (1 TO 4) ;
BEGIN
        Stage 0 : fulladd PORT MAP ('1', X(0), NOT Y(0) , S(0), C(1) ;
        Stage 1 : fulladd PORT MAP (C(1), X(1), NOT Y(1), S(1), C(2)) ;
        Stage 2 : fulladd PORT MAP (C(2), X(2), NOT Y(2),S(2), C(3)) ;
        Stage 3 : fulladd PORT MAP (C(3), X(3), NOT Y(3), S(3), C(4)) ;
        V<=C(4) XORC (3) ;
        N<=S(3) ;
        Z<='1' WHEN S (3 DOWN TO0) = "0000" ELSE '0'

END Structure ;
```

Figure 2.7 Structure VHDL code for the comparator circuit.

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

ENTITY comparator IS
        PORT (X, Y          :       IN STD_LOGIC _VECTOR (3 DO
        V, N, Z       :       OUT STD_LOGIC ) ;
END comparator ;
ARCHITECTURE Behavior of comparator IS
        SIGNAL S : STD _LOGIC_VECTOR (4DOWNTO 0) ;
BEGIN
    S<=('0' & X) – Y ;
    V<=S(4) XOR X(3) XOR Y(3) XOR S(3) ;
    N<=S(3) ;
    Z<='1' WHEN S(3 DOWNTO 0) = 0ELSE '0' ;
END Behavior ;
```

Figure 2.8 Behavioral VHDL code for the comparator circuit.

VHDL Code for 3 bit subs tractor

(This VHDL code use for 3 bit comparator by using full subtractor)

```
1.  library ieee ;
    Use ieee. Std_logic_1164. All ;

    entity comp_3 bit is
    port (a:in std_logic_vector (2 downto 0) ;
    b:in std_logic_vector (2 downto 0) ;
    agb, aeb, alb : inout std_logic) ;
    end comp_3bit ;

    architecture dataflow of comp_3 bit is
    component fullsub is
    port (a,b,c : in std_logic) ;
    end component ;
    signal s, c:std_logic_vector (2 down to 0) ;
    begin
    a1 : fullsub port map (a(0), b(0), '0', s(0), c(0));
    a2 : fullsub port map (a(1), b(1), c(0), s(1), c(1));
    a3 : fullsub port map (a(2), b(2), c(1), s(2), c(2));
    agb<=aeb nor alb ;
    aeb<=not (s(0) or s(1) or s(2))  ;
    alb < = c (2) ;
    end dataflow ;
```

Subtractor Code

```
    library ieee  ;
    use ieee. Std_logic _1164.all ;

    entity fullsub is
    port (a,b, c:instd_logic ;
    s, cout : out std_logic) ;
    end fullsub ;

    architecture dataflow of fullsub is
    begin
    s<=a xorb xor c ;
    cout < = (not a and (b or c) or (b and c) ;
    end dataflow ;
```

4X4 – Bit Multiplier VHDL Code

```
Library IEEE ;
use IEEE. STD_LOGIC_1164.ALL ;
use IEEE.MUMERIC_STD.ALL ;

entity Multiplier_VHEL is
    port
    (
        Nibblel, Nibble2 : in std_logic _vector (3 downto 0);
        Result : out std_logic_vector (7 downto 0)
        );
end entity Multiplier_VHDL ;
architecture Behavioral of Multiplier _VHEL is
begin

        Result   <=std_logic_vector (unsigned (Nibblel)* unsigned (Nibble 2)* unsigned
(Nibble2));
end architecture Behavioral ;
```

UNIT – III

3.1. INTRODUCTION / REPRESHING TO FLIP – FLOPS & ITS EXCITATION TABLE

T flip- flop

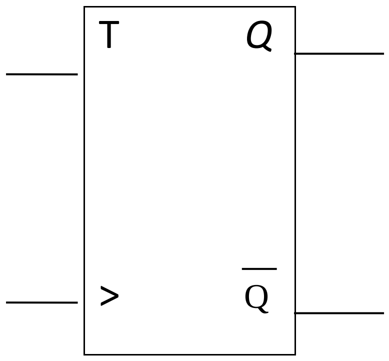


Fig. 3.1 A circuit symbol for a T-type flip-flop

If the T input is high, the T flip – flop changes state ("toggles") whenever the clock input is strobed. If the input is low, the flip-flop holds the previous value. This behavior is described by the characteristic equation :

$Q_{next} = T \oplus Q = T\overline{Q} + \overline{T}Q$ (expanding the XOR operator) and can be described in a truth table :

T flip – flop operation

Characteristic table

T	Q	Qnext	Comment
0	0	0	hold state (no clk)
0	1	1	hold state (no clk)
1	0	1	toggle
1	1	0	toggle

Excitation table

Q	Qnext	T	Comment
0	0	0	No change
1	1	0	No change
0	1	1	Complement
1	0	1	Complement

JK flip – flop

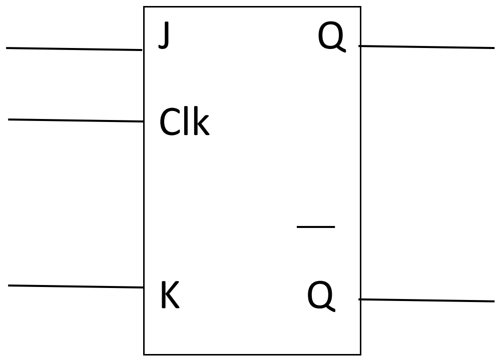


Fig. 3.2. A circuit symbol for a positive – edge – triggered JK flip – flop

JK flip – flop

The combination J = 1, K = 0 is a command to set the flip – flop; the combination J = 0, K=1 is a command to reset the flip – flop ; and the combination J = K = 1 is a command to toggle the flip-flop , i.e. , change its output to the logical complement of its current value. Setting J = K = 0 does NOT result.

The characteristic equation of the JK flip – flop is :

$Q_{next} = J\overline{Q} + \overline{K}Q$

and the corresponding truth table is :

JK flip – flop operation

Characteristic table

J	**K**	**Qnext**	**Comment**
0	0	Q	hold state
0	1	0	reset
1	0	1	set
1	1	Q	toggle

Excitation Table

Q	**Qnext**	**J**	**K**	**Comment**
0	0	0	X	No change
0	1	1	X	Set
1	0	X	1	Reset
1	1	X	0	No change

SR NOR latch

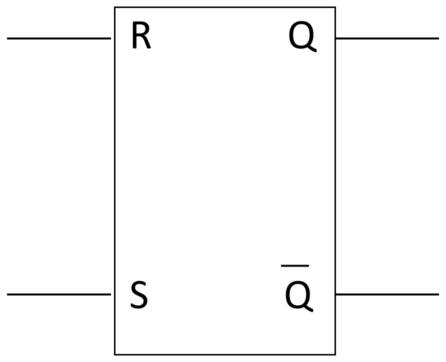


Fig : 3.3. Circuit symbol for SR latch

An SR latch, constructed from a pair of cross-coupled NOR gates

While the S and R inputs are both low, feedback maintains the Q and Q outputs in a constant state. If S (Set) is pulsed high while R (Reset) is held low, then the Q output is forced high, and stays high when S returns to low ; similarly, if R is pulsed high while S is held low, then the Q output is forced low, and stays low when R returns to low.

Characteristic table

S	R	Qnext	Action
0	0	Q	hold state
0	1	0	reset
1	0	1	set
1	1	X	not allowed

Excitation table

Q	Qnext	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

Characteristic equation : Q + R’Q + R’S or Q + = R’Q + S.

D – Flip – Flop

D- flip - flop is a very useful storage element. Its present state-next state table demonstrates the behavior of a D-flip-flop.

It has the following characteristics :

Characteristic equation : Q(t+1) = D

Characteristic Table

D	Q(t+1)	Operation
0	0	Reset
1	1	Set

Excitation Table

Q(t)	Q(t+1)	D
0	0	0
0	1	1
1	0	0
1	1	1

Counters

A counter is a circuit used to count a repeated set of values, like clock pulses. In this case, the counter is used to count the number of clock cycles. Since the clock pulses occur at known intervals, the counter can be used as an instrument for measuring time (and therefore period or frequency).

Counters can be classified into two types.

They are

i) Asynchronous (or) ripple (or) serial counter.

ii) Synchronious counter (or) parallel counter

In a serial counter each flip flop is triggered by the previous FF and thus the counter has a cumulative settling time. In synchronous counters the FFs are triggered by a single clock pulse simultaneously.

Comparison of Asynchronous counter and Synchronous counter

Asynchronous counter	Synchronous counter
1. Each FF clocked by previous FF	All FFs clocked simultaneously
2. Propagation delay of counter = Propagation delay of each FF x no. of FFs. Hence slow speed of operation.	Propagation delay of counter = Propagation delay of one FF and the combinational hardware. Hence high speed of operation.
3. Simple Hardware	More complex hardware

4-Bit Binary Up – Ripple counter (Asynchronous counter)

Pulse counters are formed by cascading the flip-flops. A4 bit Binary counter using four JK MS flip-flops is shown fig. 3.4.

The pulses to be counted are applied to the clock input of FF_1. For all stages J and K are tied to the supply voltage, so that J=K=1, and makes JK MS Flip Flop as a Toggle Flip Flop. Now the Q_1 output toggle in each falling or negative edge of the clock pulse.

Since Q_1 is the clock input for Flip flop FF_2, Q_2 toggles with each negative edge of Q_1. Similarly Q_3 toggles with each negative edge Q_2 and Q_4 toggles with each negative edge of Q_3.

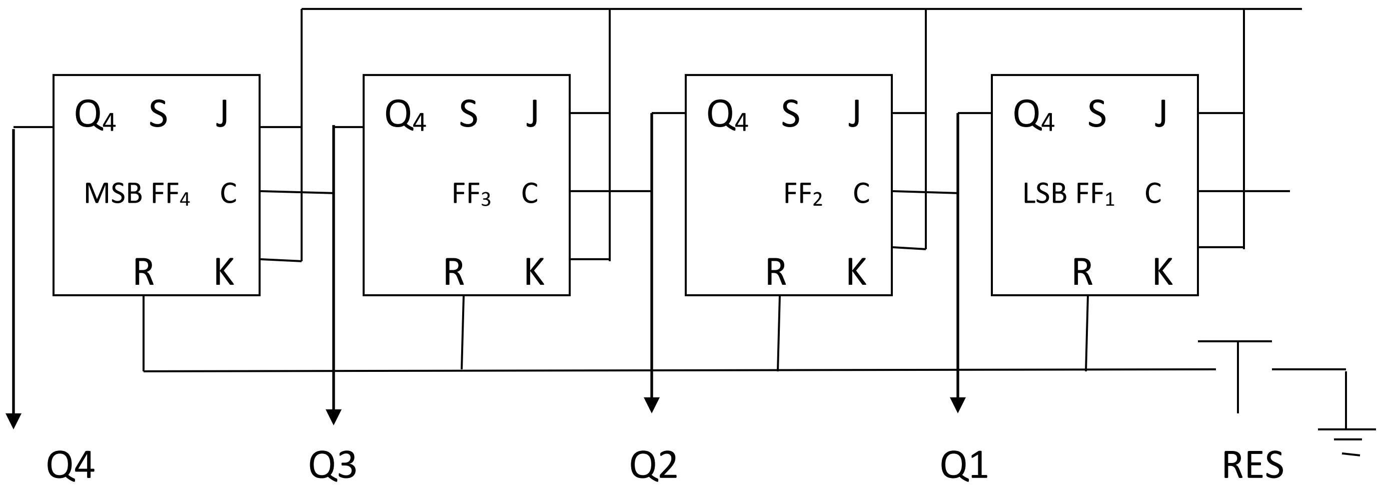


Fig 3.4

Pulse	Q_4	Q_3	Q_2	Q_1
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
16	0	0	0	0
17	0	0	0	1

4- Bit ripple down counter

A simple 4 bit Down Counter can be constructed using four JKMS flip flop in the toggle mode as shown in fig. 3.5. The clock pulse input is given to C_1. Q_1 is connected to C_2, Q_2 to C_3, Q_3 to C_4. The outputs are taken as usual from Q_4 Q_3 Q_2 Q_1. When all the flip flops are Reset, Q_4, Q_3, Q_2, Q_1 =0000. In the first negative edge of the clock, Q_1 toggles from 0 to 1 . This means Q_1 changes from 1 to 0 since this is a negative transition immediately Q_2 also toggles from 0 to 1 and Q_2 from 1 to 0 .

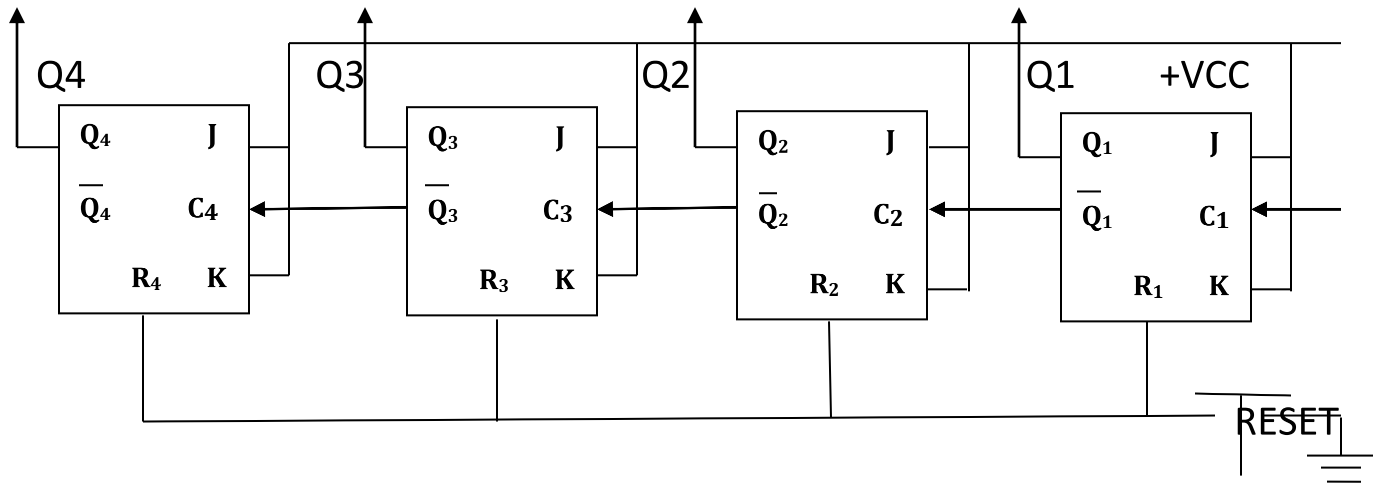


Fig3.5

R	Q_4	Q_4	Q_3	Q_3	Q_2	Q_2	Q_1	Q_1	Pulse
0 ←	1	0	1	0	1	0	1	0	0
15 ←	0	1	0	1	0	1	0	1	1
14 ←	0	1	0	1	0	1	1	0	2
13 ←	0	1	0	1	1	0	0	1	3
12 ←	0	1	0	1	1	0	1	0	4

The change in Q2 being a negative edge, Q_3 also toggles from 0 to 1 and Q_3 from 1 to 0. So a single clock pulse causes change from 0000 to 1111 by chain reaction.

In the next negative edge of the clock, Q_1 toggles from 1 to 0 and Q_1 from 0 to 1. The change Q_1 in now is positive and so no further toggling takes place in the other flip flops. So the output now is Q_4 Q_3 Q_2Q_1 = 1110. For the next negative edge of the clock pulse Q_2 toggles from 1 to 0 Q_2 from 0 to 1 . Now the output is Q_4 Q_3 Q_2 Q_1 = 1100. In this way the process is going on and finally $Q_4Q_3Q_2Q_1$ = 0000.

4- Bit binary UP / Down counter

An Up- Down counter can be constructed by using exclusive OR gates alone with the JK MS flip flops, as shown in Fig 3.6 (J & K inputs are connected to V_{CC})

If the control line is at 0 the output of the gates is Q and so we get up counting.

On the other hand, if the control line is held at 1, the output of the gates is Q_1 and then we get down counting.

In the up / down counters, the Flip Flop outputs can be set with the set terminal.

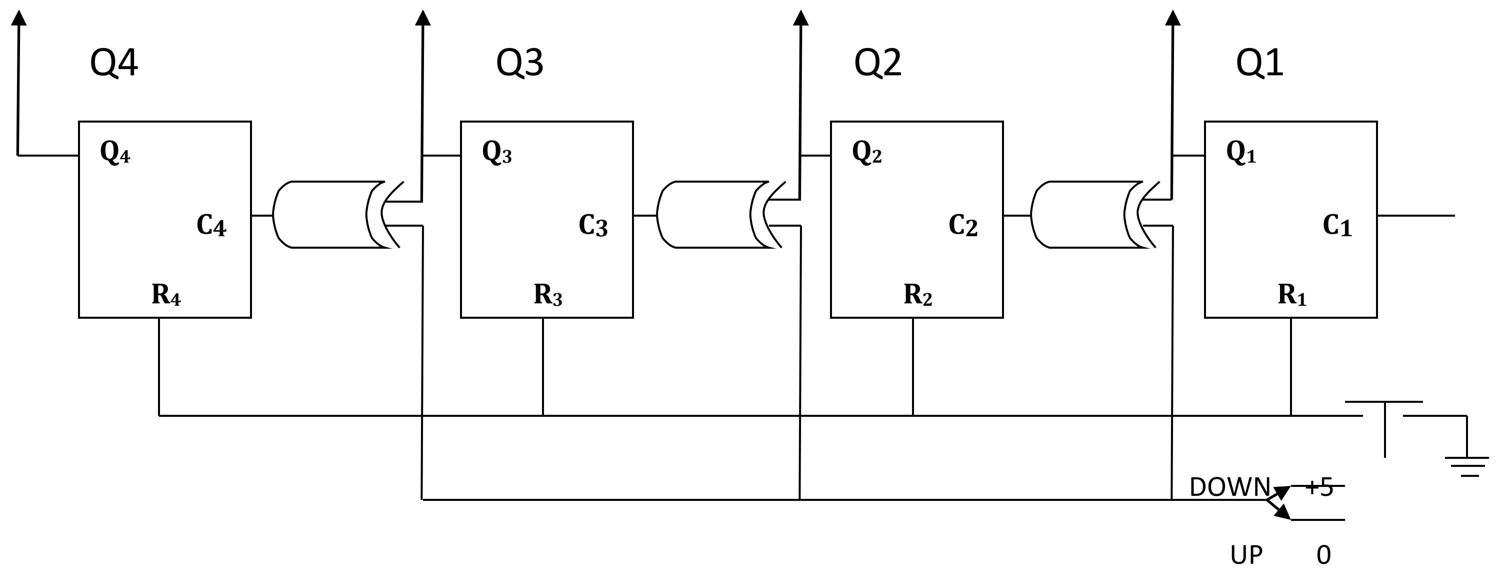


Fig 3.6

Ripple Counters

In a ripple counter the clock pulses are applied to clock input of first flip flop. The clock inputs of other flip flop are obtained from the previous Q outputs.

Let us consider this ripple counter in the state Q_4, Q_3, Q_2, Q_1 = 1111 (15).

When the next clock pulse arrives, all the flip – flops are reset giving Q_4, Q_3, Q_2, Q_1 = 0000

But this does not happen instantaneously because in the trailing edge of the clock pulse, Q_1 changes from 1 to 0, this trailing edge of Q_1 causes Q_2 to change from 1 to 0, this trailing edge of Q_2 causes Q3 to change from 1 to 0 and similarly for Q4. Though the final result is 0000, the output passes through intermediate states such as 1110, 1100, 1000.

Thus the output ripples through 4 flips flops and hence it is called as ripple counter. The ripple counter has a certain propagation delay just a fraction of micro second.

Also the intermediate state can cause trouble. To void this difficulty synchronous counters are developed. In synchronous counters output of all flip flop resets simultaneously.

Mod n counter

A counter, which is reset at the nth clock pulse is called mod 'n' counter or divide by 'n' counter or divide by 'n' counter. An ordinary 3 bit binary up counter is automatically reset at the 8th clock pulse. Hence it is called "mod 8 counter" or divide by 8 counter".

Similarly an ordinary 4 bit binary up counter will reset at the 16th clock pulse, hence it is called "mod 16 counter" or "divide by 16 counter".

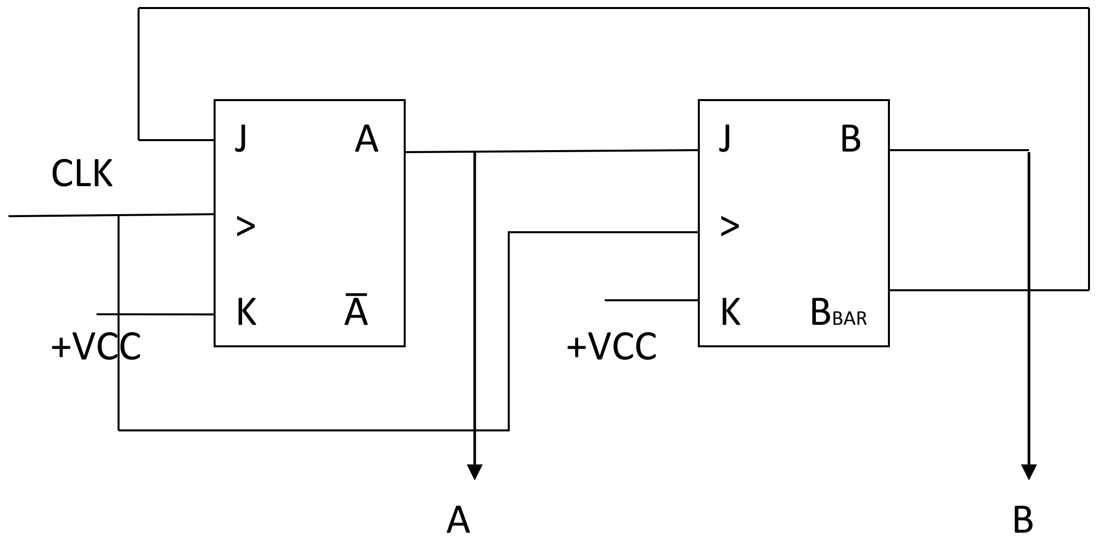
A mod -2 counter consists of only the flip-flop, a mod – 4 counter requires two flip-flops and it cunts through four discrete states. Three flip-flops form a mod -8 counter, while four flip – flop form a mod -16 counter. Hence the ordinary counters have a natural count of 2, 4, 8, 16, 32, 64 and so on by using proper number of flip – flops.

It we desire to construct the counters having the mod of other than 2,4,8,16 and so on, the following points to be remembered.

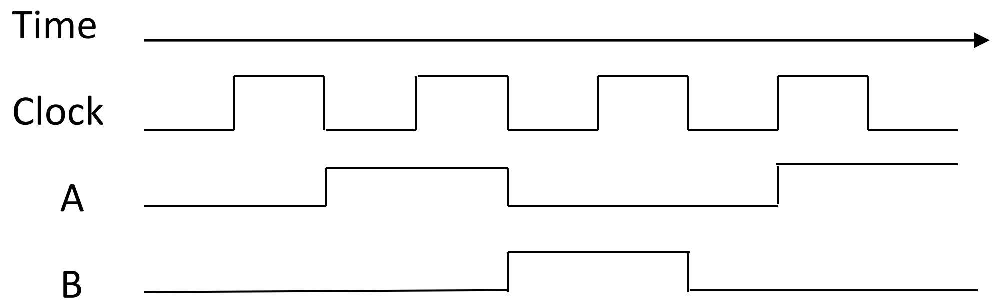
1. To determine the number of flip – flops required, it is determined by choosing the lowest natural count that is greater than the desired modified count. For example a mod-7 counter require three flip – flops.
2. Add an extra logic circuit , to reset the flip – flop in a required level.

A Mod – 3 Counter

The two flip – flops in fig. 3.7 is connected to provide a mod-3 counter. Since two flip-flops have a natural count of 4, this counter skips one state. The truth table in Fig.3.7(c) Show that this counter progresses through the count sequence 00, 01, 10 and then back to 00. It clearly skips count 1.



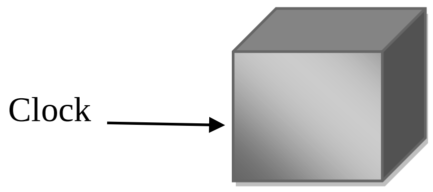
(a) Logic Diagram



(b) Wave forms

B	A	Count
0	0	0
0	1	1
1	0	2
0	0	0

© Truth Table



(d) Logic block

The two-flip-flop mod-3 counter is considered as a logic building block as shown in fig. 3.7(d). This counter divides the clock frequency by 3.

Mod-10 counter (Decade counter)

BCD numbers are ranging from 0000 to 1001, which have the decimal values of 0 to 9. BCD counter means, a counter which will count the values from 0000 to 1001, and also reset the next (10th) clock pulse.

Hence a mod 10 or divide by 10 counter is called BCD counter. A BCD needs four flip-flops and a two input NAND gate. The NAND gate is used to reset all the flip-flop at the 10th clock clock.

Synchronous counter :

A 4-bit synchronous counter is shown in Fig. 3.8. Here the clock pulses are fed to each flip flop simultaneously. So after the 15th clock pulse the state of the Flip Flop Q_4 Q_3 Q_2 Q_1 = 1111.

Q_1 toggles if $J_1 = K_1 = 1$

Q_2 Toggles if $Q_1 = q$

Q_3 Toggles if $Q_1 Q_2 = 1$ (i.e. $A_1 = 1$)

Q_4 Toggles if $A_2 = 1$ (i.e. $Q_3A_1 = 1$i.e. $Q_1Q_2Q_3 = 1$)

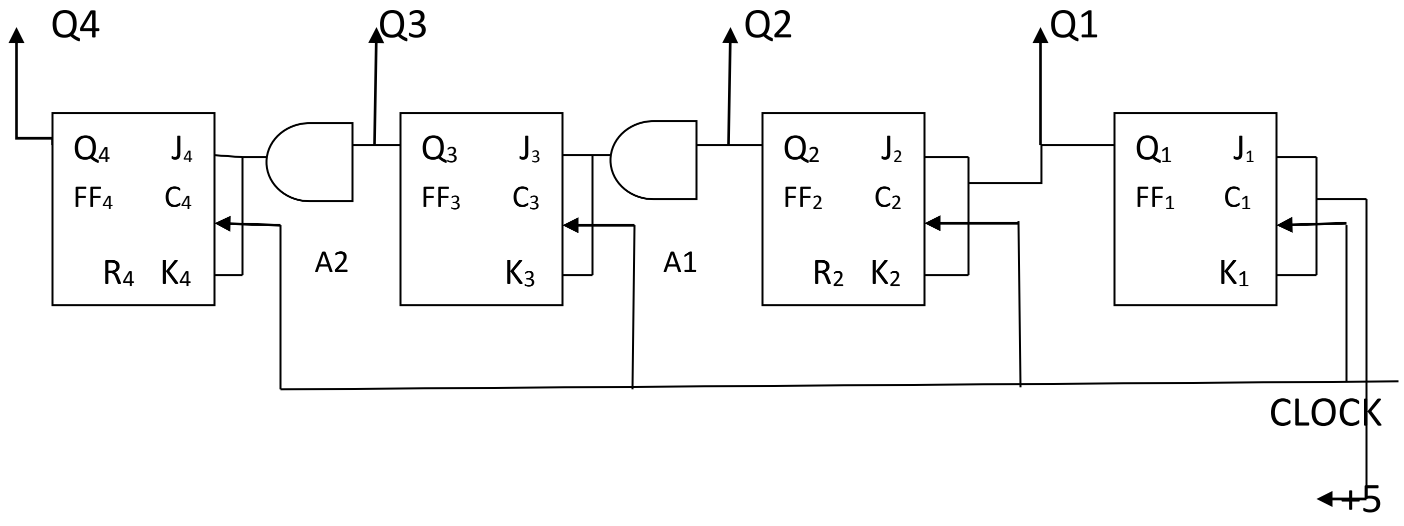


Fig 3.8 Synchronous Counter

At the arrival of 16th clock pulse Q_1 will change to 0. Immediately the states of all the Flip Flops will b 0000. Thus all the flip flops toggle simultaneously in a single step.

The maximum frequency of operation for synchronous counter is 34MHz while 16MHz for the Ripple counters in TTL.

Shift Registers

A register is simply a group of flip flops that can be used to store binary numbers. Each flip- flop can be store on bit of binary data.

A register used to store an 8 bit binary number must have eight flip – flops.

Naturally, the flip flops must be connected in cascaded manner, such that the binary numbers can be entered (shifted) into the register and possibly shifted out.

A group of flip flops connected to provide for entering and shifting the binary data is called shift registers.

The bits in a binary number can be moved from one place to another's in to ways, namely serial shifting and parallel shifting.

In serial shifting, the data bits are shifted in serial fashion beginning with either the MSB side or LSB side. In parallel shifting, all data bit are shifted simultaneously.

There are two ways to shift data into a register, (serial or parallel) and also two ways to shift the data out of the register.

This leads to the construction of four basic register types as shown the Fig. 3.9 they are :

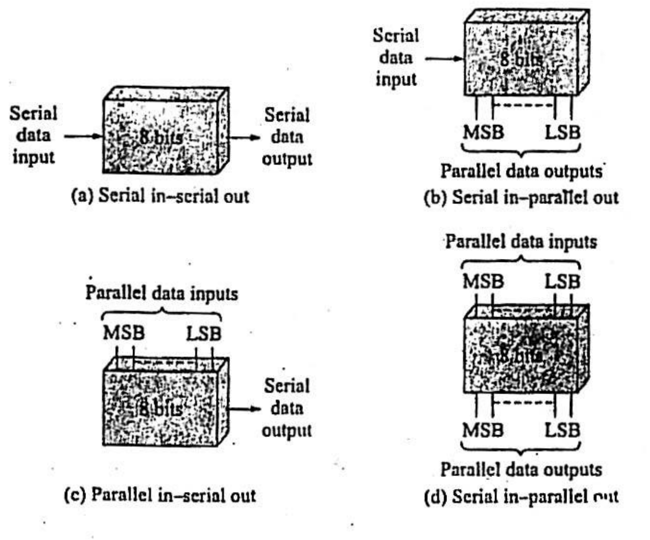


Fig. 3.9 Shift register types

1. Serial in – Serial out (SISO)
2. Serial in – Parallel Out (SISO)
3. Parallel in – Serial Out (PISO)
4. Parallel in – Parallel Out (PIPO)

We now need to consider the methods for shifting data in either a serial or parallel fashion. Data shifting techniques and methods for constructing the four different types of registers are discussed in the following sections.

1. Serial in & Serial out

Fig . 3.10 shows serial in serial out shift left register.

We will illustrate the entry of the four bit binary number 1111 into the register, beginning with the left-most bit.

Initially, register is cleared , So Q_A Q_B Q_CQ_D = 0000

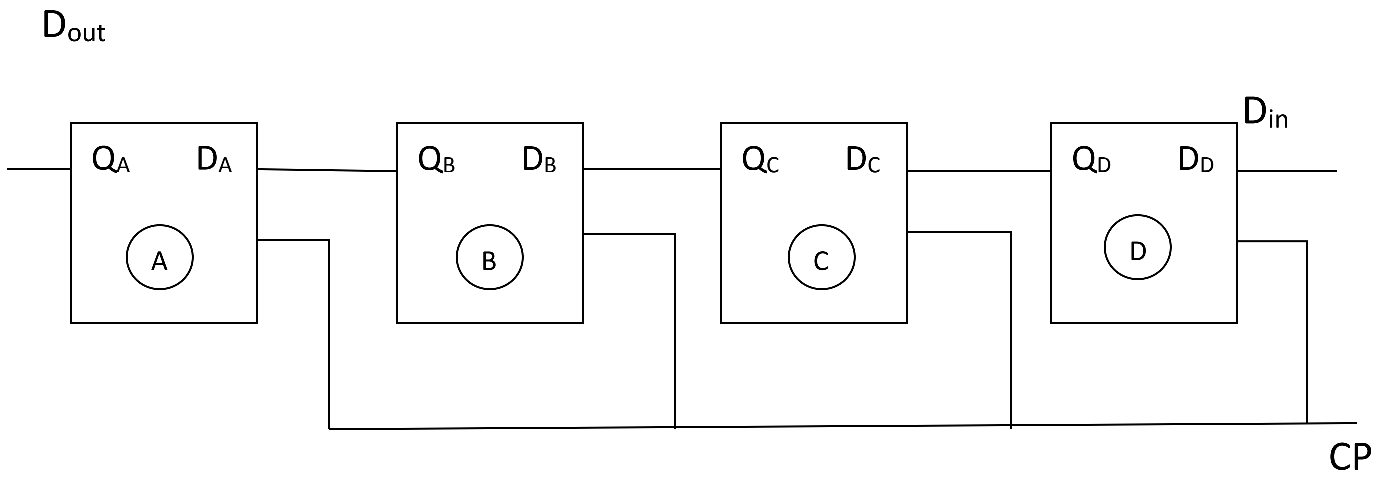


Fig 3.10

(a) When data 1111 is applied serial, i.e.

Left – most 1 is applied as D_{in}

$D_{in} = 1$, $Q_A\ Q_B\ Q_C\ Q_D = 0000$

The arrival of the first falling clock edge sets the right - most flip – flop, and the stored word becomes,

$Q_A\ Q_B\ Q_C\ Q_D = 0001$

(b) When the next negative clock edge hits, the Q_C flip – flop sets and the register contents become.

(c) The third negative clock edge results in ,

$Q_A\ Q_B\ Q_C\ Q_D = 0111$

(d) The fourth falling clock edge results in ,

$Q_A\ Q_B\ Q_C\ Q_D = 1111$

Fig. 3.11 shows serial in serial out shift right register

We will illustrate the entry of the four bit binary number 1111 into the register, beginning with the left-most bit.

Initially, register is cleared. So $Q_A\ Q_B\ Q_C\ Q_D = 0000$

(a) When data 1111 is applied serially, i.e. left – most 1 is applied as Din.

$D_{in} = 1$ $Q_A\ Q_BQ_CQ_D = 0011$

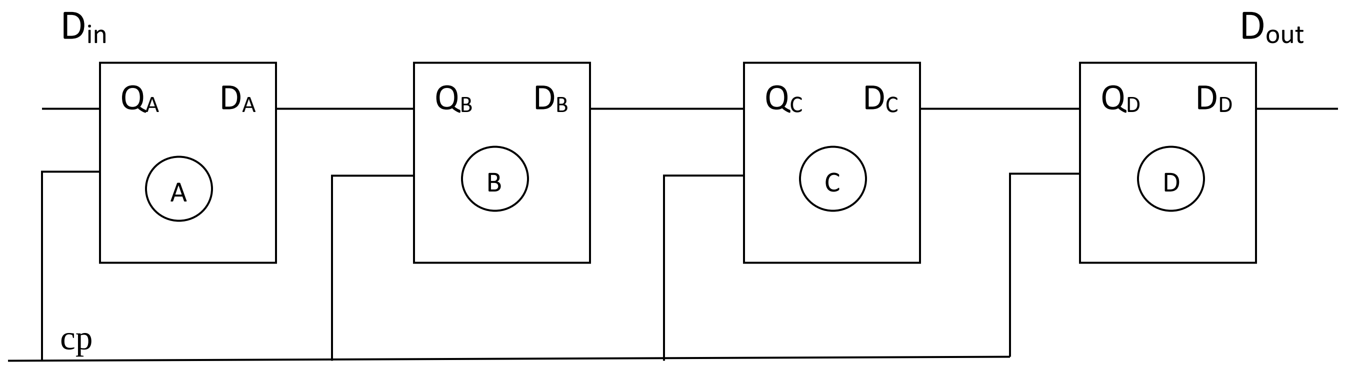


Fig3.11

The arrival of the first falling clock edge sets the left – most flip-flop, and the stored word becomes,

$Q_A\ Q_B\ Q_C\ Q_D = 1000$

(b) When the next falling clock edge hits, the Q_B flip – flop sets and the register contents become,

$Q_A\ Q_B\ Q_C\ Q_D=1100$

(c) The third falling clock edge results in,

$Q_A\ Q_B\ Q_C\ Q_D=1110$

(d) The fourth falling clock edge gives,

$Q_A\ Q_B\ Q_C\ Q_D=1111$

Serial in parallel out shift register

In this case, the data bits are entered into the register in the same manner as discussed in the last section, i.e. serially.

But the output is taken in parallel. Once the data are stored, each bit appears on its respective output line and all bits are available simultaneously, instead of a bit-bybit basis as with the serial output.

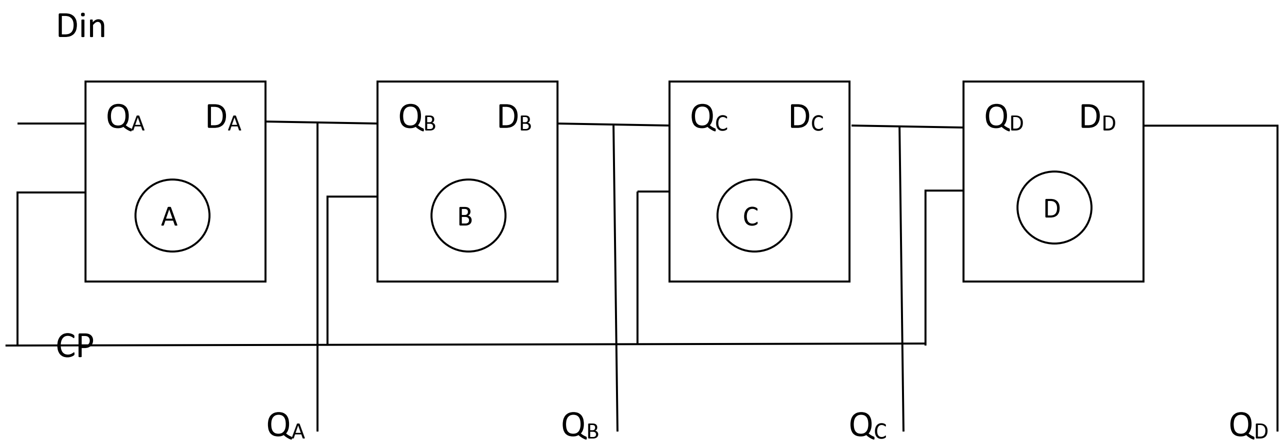


Fig3.12

Parallel in serial out shift register

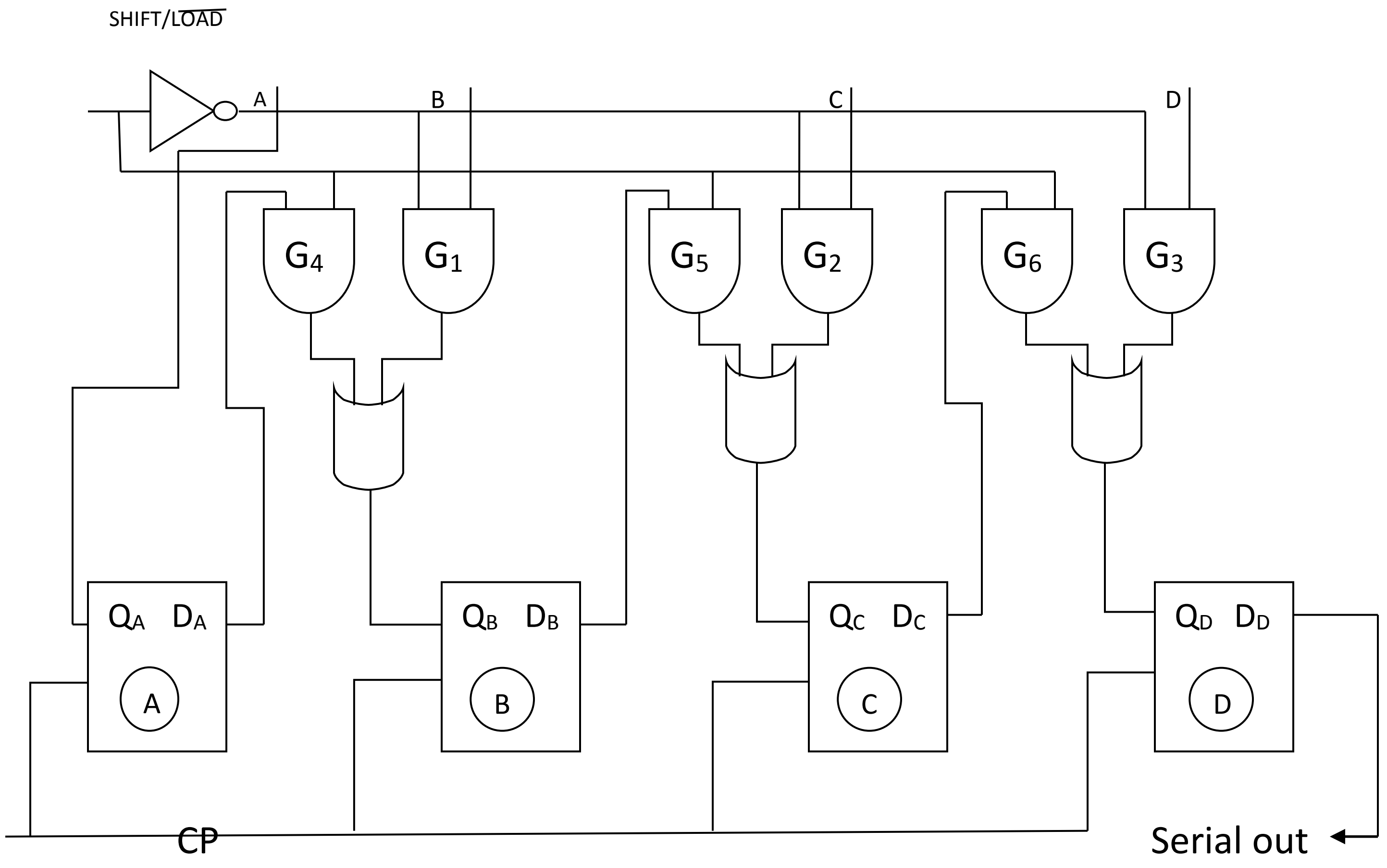


Fig3.13

In this type , the bits are entered in parallel i.e. simultaneously into their respective stages on parallel lines.

Fig. 3.13 illustrates a four - bit parallel in serial out register. There are four input lines X_A, X_B, X_C , X_D for entering data in parallel into the register.

SHIFT / LOAD is the control input which allows shift or loading data operation of the register.

When SHIFT / LOAD is low, gates G_1, G_2, G_3 are enabled, allowing each input data bit to be applied to D input of its respective flip-flop.

When a clock pulse is applied, the flip-flip with D=1 will SET and those with D=0 will RESET. Thus all four bits are stored simultaneously.

When SHIFT / LOAD is high gates G_1, G_2, G_3 are disabled and gates G_4 G_5, G_6 are enabled. This allows the data bits to shift left from one stage to the next.

Parallel in parallel out register

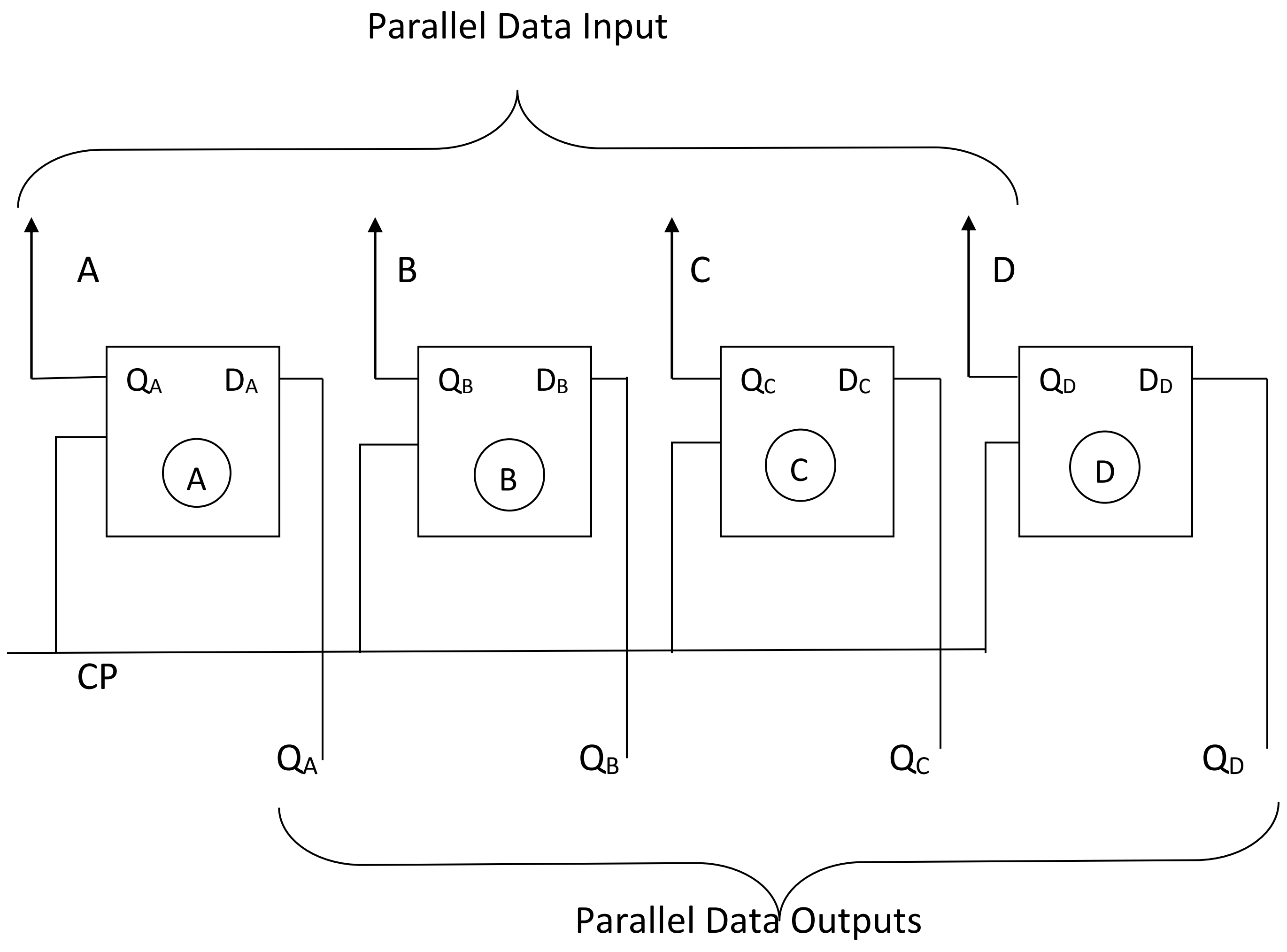


Fig3.14

From the third and second types of registers, it is cleared that how to enter the data in parallel i.e. all bits simultaneously into the register and how to take data out in parallel from the register.

In parallel in parallel out register, there is simultaneous entry of all data bits and the bits appear on parallel outputs simultaneously. Fig. 3.14 shows this type of register.

SEQUENTIAL CIRCUIT DESIGN

3.2. DESIGN STEPS

Sequential circuits are also called finite state machines (FSMs), The name derives from the fact that the functional behavior of these circuits can be represented using a finite, number of states. We will often use the term finite state machine, or simply machine, when referring to sequential circuits.

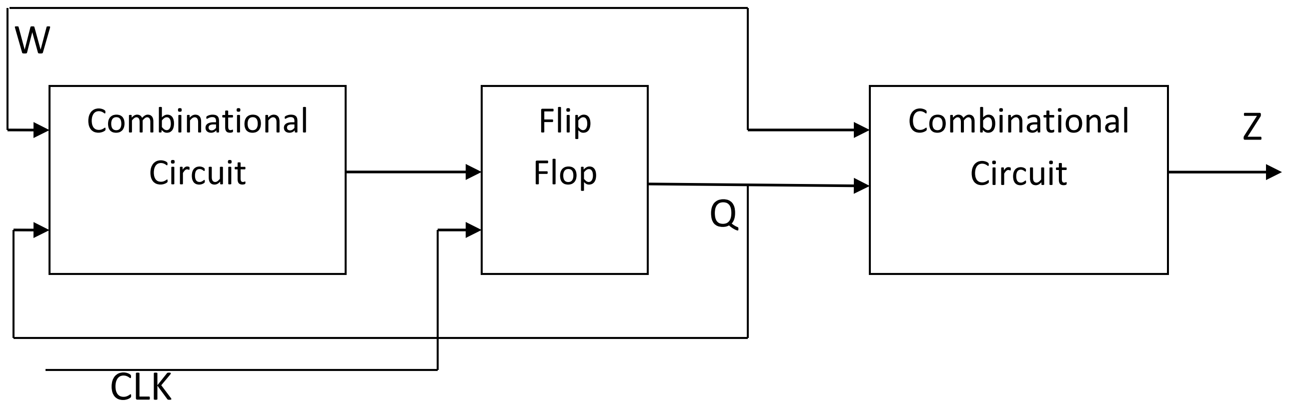


Fig3.15 The General Form of a Sequential Circuit

STATE DIAGRAM

The first step in designing a finite state machine is to determine how many state are needed and which transitions are possible from one state to another. A good way to begin is to select one particular state as a starting state; this is the state that the circuit should enter when power is first turned on or when a reset signal is applied.

The starting state is called state A. As long as the input w is 0, the circuit need not do anything, and so each active clock edge should result in the circuit remaining in state A. When w becomes equal to 1, the machine should recognize this, and move tot a different state, which we will call state B. This transition takes places on the next active clock edge after w has become equal to 1.

In state B, as in state A, the circuit should keep the value of output z at 0, because it has not yet seen w= 1 for two consecutive clock cycles. When in state G, if w is 0 at the next active clock edge, the circuit should move back to state A. However, if w = 1 when in state B, the circuit should change to a third state, called C, and it should then generate on output z=1. The circuit should remain in state C as long as w= 1 and should continue to maintain z = 1.

When w becomes 0, the machine should move back to state A. Since the preceding description handles all possible values of input w that the , machine can encounter in its various state. Hence we conclude that three states are needed to implement the desired machine.

Behavior of a sequential circuit is described in several different ways. The simplest method is to use a pictorial representation in the form of a state diagram. The state diagram is a graph that depicts states of the circuit as nodes (circles) and transitions between states as directed arcs. The state diagram in Figure 3.16 defines the behavior that corresponds to our specification. States A, B and C appear as nodes in the diagram.

Node A presents the starting state, and it is also the state that the circuit will reach after an input w=0 is applied. In this state the output z should be 0, which is indicated as A/z=0 in the node.

The circuit should remain in state A as long as w = 0, which is indicated by an arc with a label w = 0 that originates and terminates at this mode. The first occurrence of w=1 (following the condition w= 0) is recorded by moving from state A to state B. This transition is indicated on the graph by an arc originating at A and terminating at B.

The label w =1 on this arc denotes the input value that causes the transition. In state B the output remains at 0, which is indicated as B/z =0 in the node.

When the circuit is in state B, it will change to state C if w is still equal to 1 at the next active clock edge. In state C the output z becomes equal to 1. If w stays at 1 during subsequent clock cycles, the circuit will remain in state C maintaining z =1. However, if w becomes 0 when the circuit is either in state B or in state C, the next active clock edge will cause a transition to state A to take place.

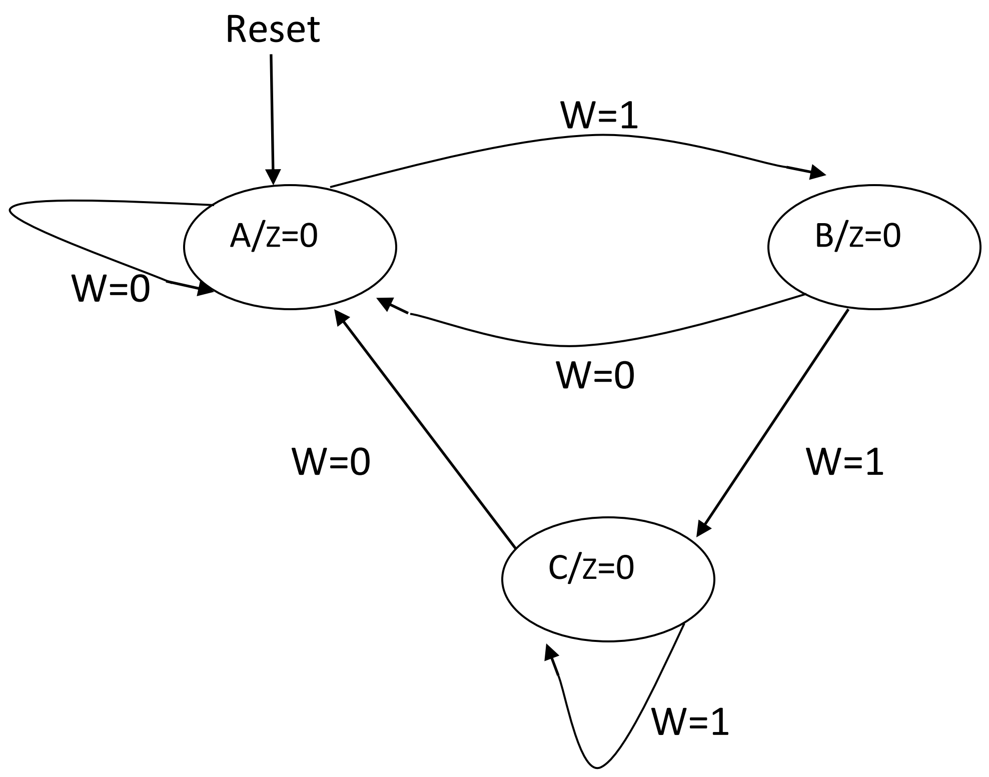


Figure 3.16 State diagram of a simple sequential circuit.

STATE TABLE :

Although the state diagram provides a description of the behavior of a sequential circuit that is easy to understand, to proceed with the implementation of the circuit, it is convenient to translate the information contained in the state diagram into a tabular form. Figure 3.17 shows the state table for our sequential circuit. The table indicates all transitions from each present state to the next state for different values of the input signal. Note that the output z is specified with respect to the present state.

Present state	Next State		Output z
	ω = 0	ω=1	
A	A	B	0
B	A	C	0
C	A	C	1

Figure 3.17 State table for the sequential circuit in figure 3.3

STATE ASSIGNMENT

The state table in Figure 3.4 defines the three states in terms of letters A, B and C. When implemented in a logic circuit, each state is represented by a particular valuation (combination of values) of state variables. Each state variable may be implemented in the form of a flip-flop. Since three states have to be realized, it is sufficient to use two state variables. Let these variables by y_1 and y_2.

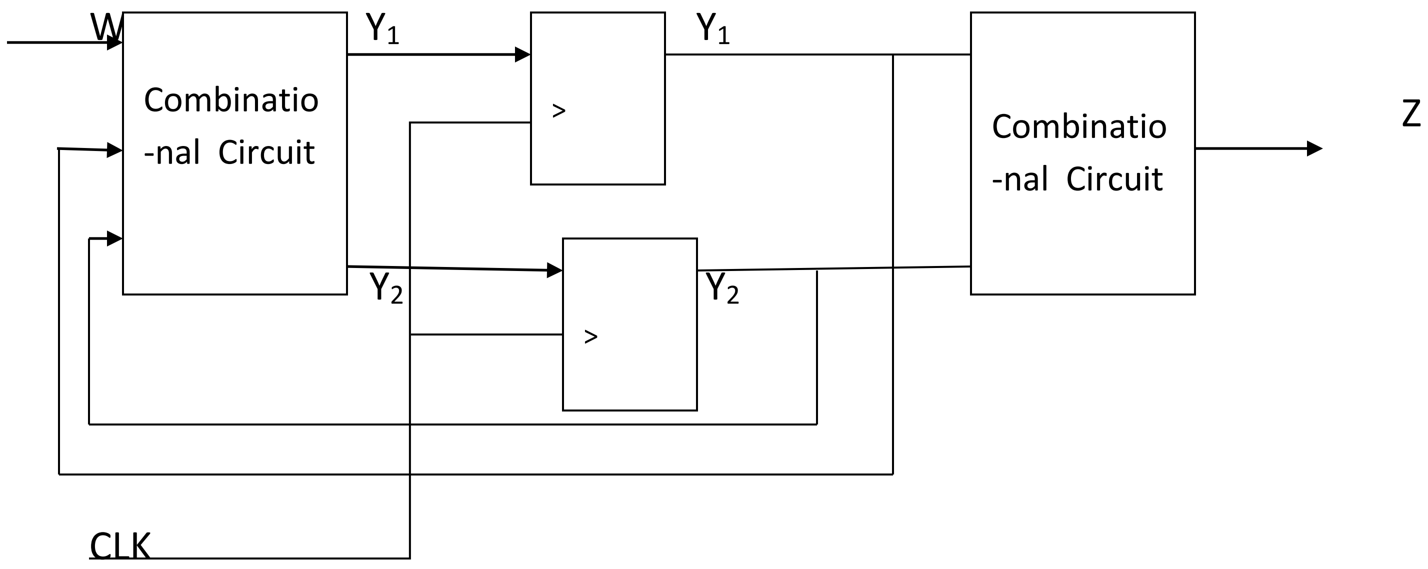


Figure 3.18 A general sequential circuit with input w, output z, and two state flip-flops

Figure 3.18 shows to indicate the structure of the circuit that implements the required finite state machine. Two flip-flops represent the state variables.

From the specification in Figures 3.16 and 3.17, the output z is dertermined only by the present state of the circuit.

Thus the block diagram in Figure 3.18 shows that z is a function of only y_1 and y_2; our design is of Moore type.

SUMMARY OF DESIGN STEPS

We can summarize the steps involved in designing a synchronous sequential circuit as follows :

1. Obtain the specification of the desired circuit.
2. Derive the states for the machine by first selecting starting state.

   Then, given the specification of the circuit, consider all valuations of the inputs to the circuit and create new states as needed for the machine to respond to these inputs.

   To keep track of the states as they are visited create a state diagram. When completed, the state diagram shows all states in the machine and gives the conditions under which the circuit moves from one state to another.
3. Create a state table from the state diagram.
4. In our sequential circuit example, there were only three states ; hence it was a simple matter to create the state table. However, in practice it is common to deal with circuits that have a large number of states.

   In such cases it is unlikely that the first attempt at deriving a state table will produce optimal results. Almost we will have more states than is really necessary. This can be corrected by a procedure that minimizes the number of states.
5. Decide on the number of state variables needed to represent all states and perform the state assignment.

   There are many different state assignments possible for a given sequential circuit. Some assignments may be better than others.
6. Choose the type of flip-flops to be used in the circuit Derive the next – stage logic expressions to control the inputs to all flip-flops and then derive logic expressions for the outputs of the circuit.
7. Implement the circuit as indicated by the logic expression.

EXAMPLES FOR Mealy and Moore Type Finite State Machines

Objectives

- **There are two basic ways to design clocked sequential circuits. These are using :**
  1. Mealy Machine,
  2. Moore Machine.

Mealy Machine

- In a Mealy machine, the outputs are a function of the present state and the value of the inputs as shown in Figure 3.19

Accordingly, the outputs may change asynchronously in response to any change in the inputs.

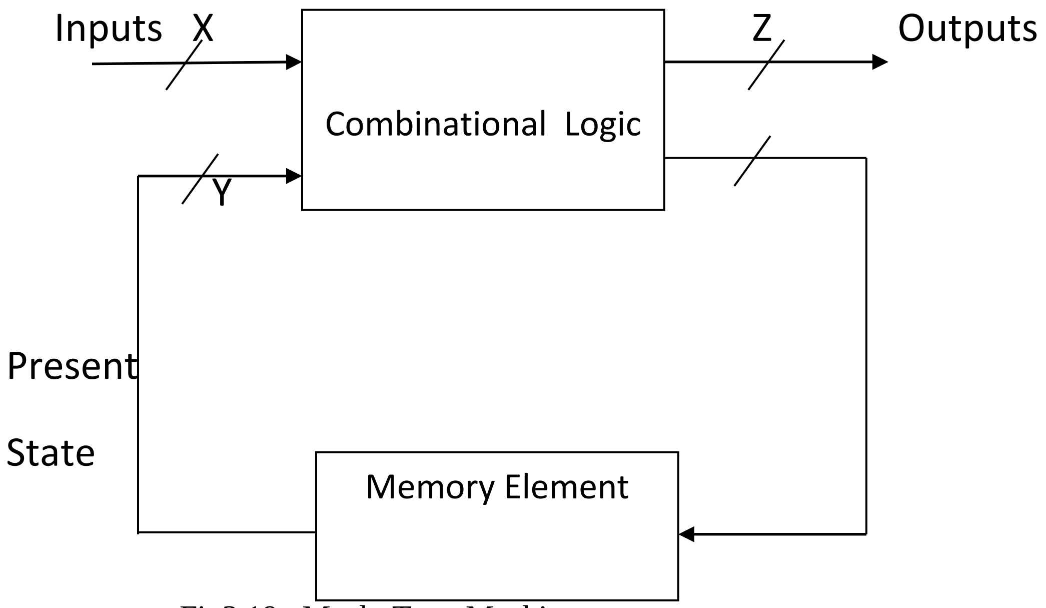


Fig3.19 Mealy Type Machine

Moore Machine

- ❖ In a Moore machine the outputs depend only on the present state as shown in Figure 3.20
- ❖ The outputs change synchronously with the state transition triggered by the active clock edge.

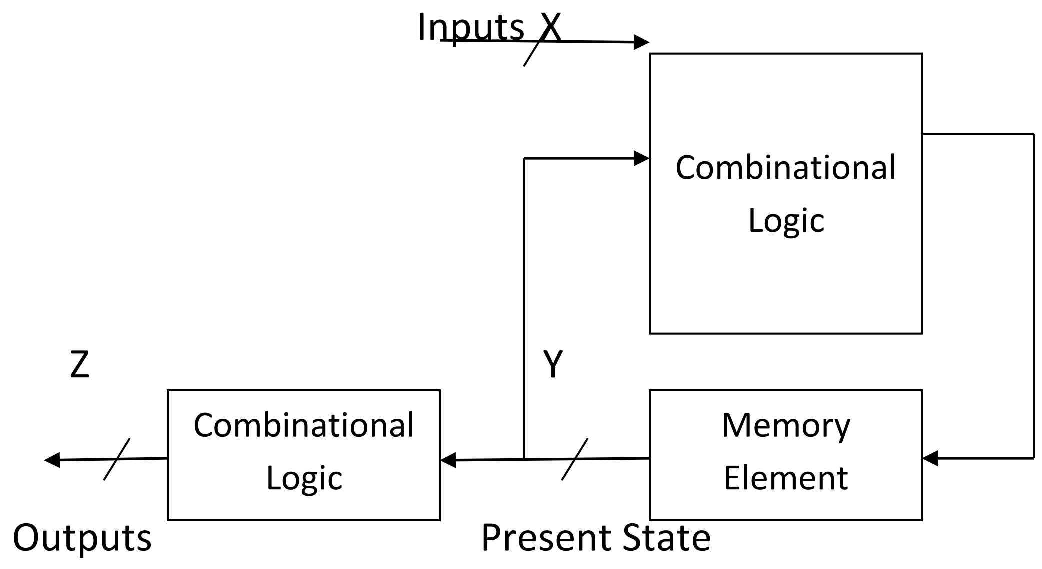


Fig3.20 Moore type machine

Mealy State Machine

- ❖ The Mealy machine state diagram is shown in Figure 3.21.
- ❖ Note that there is no reset condition in the state machine that employs two flip-flops. This means that the state machine can enter its unused state ‘11’ on start up.

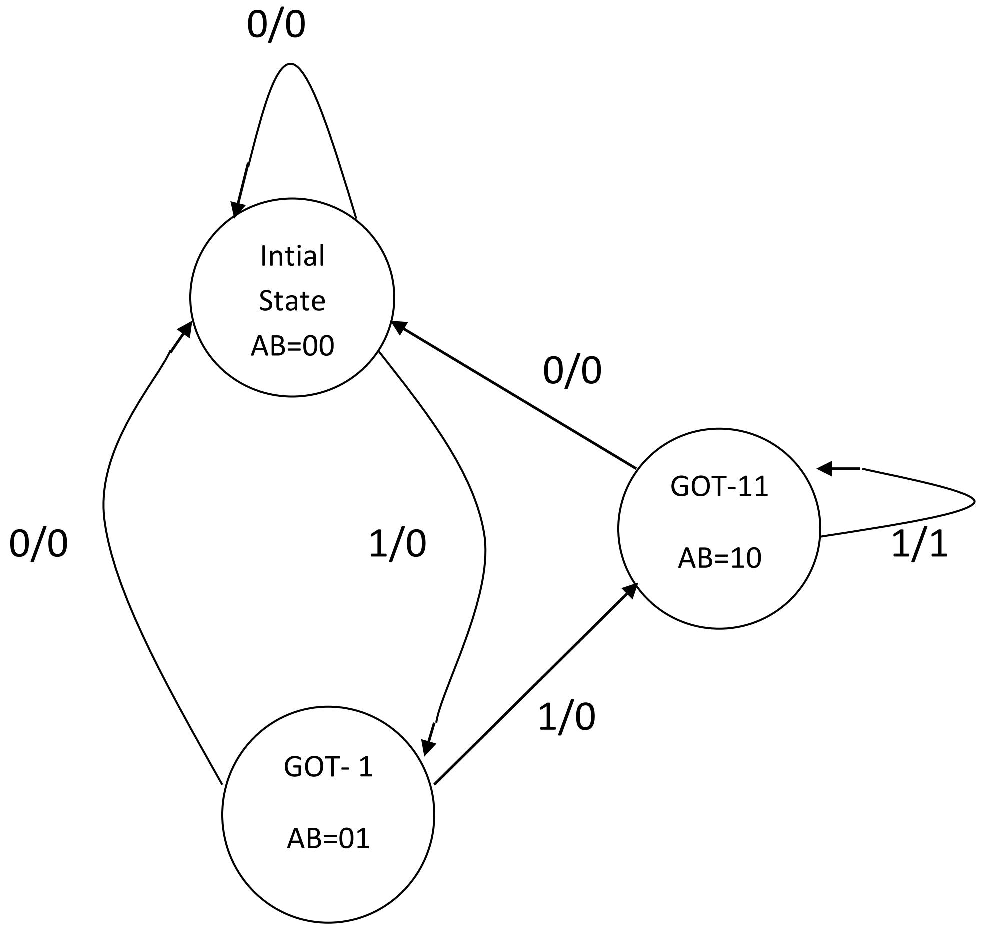


Figure 3.21 Mealy State Machine for ‘111’ Sequence Detector

- ❖ To make sure that machine gets resetted to a valid state, we use a ‘Reset’ signal.
- ❖ The logic diagram for this state machine is shown in Figure 3.22. Note that negative triggered flip-flops are used.

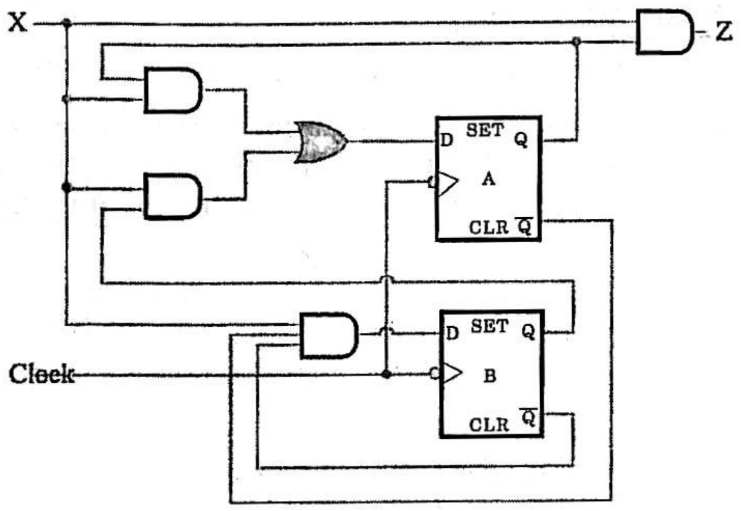


Figure 3.22 : Mealy State Machine Circuit Implementation

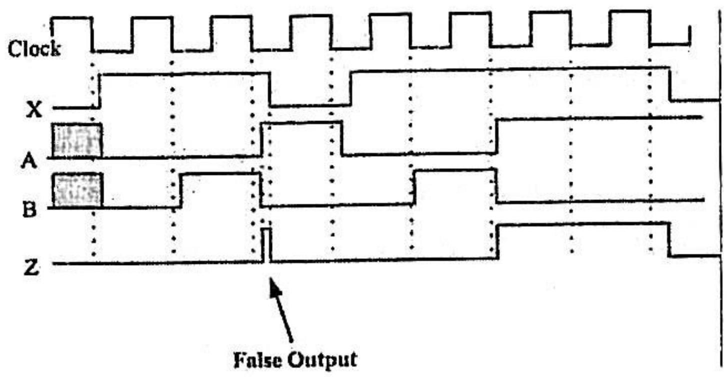


Figure 3.23: Timing Diagram for Mealy Model Sequence Detector

Timing Diagram for the circuit is shown in Figure 3.23.

❖ Since the output in Mealy model is a combination of present state and input values, an unsynchronized input with triggering clock may result in invalid output, as in the present case.

❖ Consider the present case where input ‘x’ remains high for sometime after state ‘AB=10’ is reached. This results in ‘False Output’, also known as ‘Output Glitch’.

Moore State Machine

- ❖ The Moore machine state diagram for '111' sequence detector is shown in Figure 3.24
- ❖ The state diagram is converted into its equivalent state table (See Table 1).
- ❖ The states are next encoded with binary values and we achieve a state transition table (See Table 2).

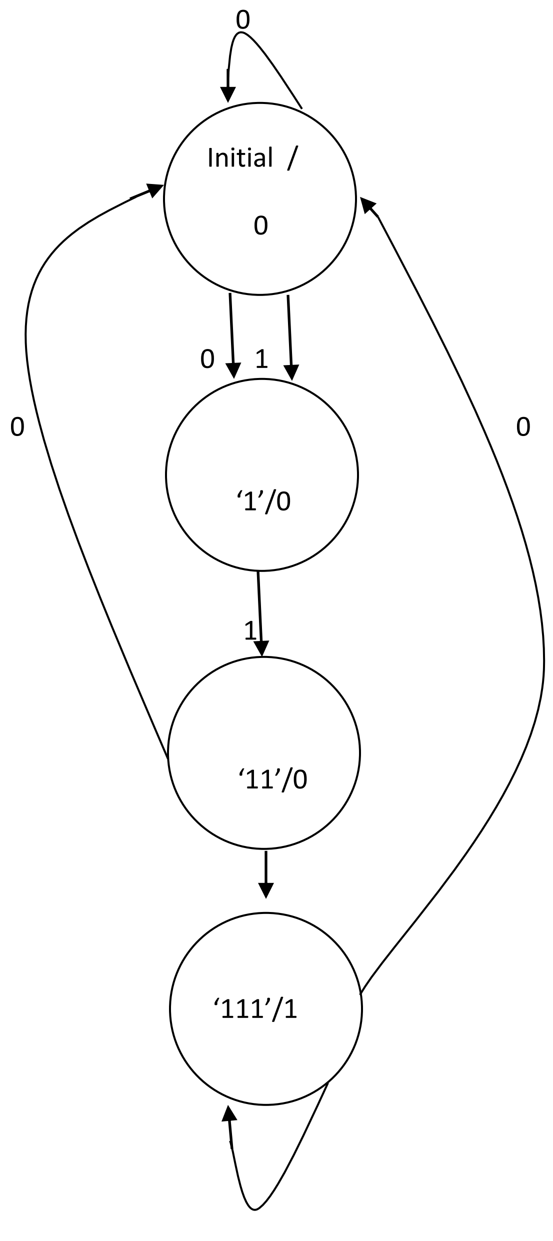


Figure 3.24: Moore Machine State Diagram

Table 1 : State Table

Present	**Next State**		**Output**
Present State	**Next State**		**Output**
	x =0	**x =1**	**Z**
Initial	Initial	Got – 1	0
Got – 1	Initial	Got – 11	0
Got – 11	Initial	Got – 111	0
Got – 111	Initial	Got – 111	1

Table 2 : State Transition Table and Output Table

Present	**Next State**		**Output**
Present State	**Next State**		**Output**
	x =0	**x =1**	**Z**
Initial	Initial	Got – 1	0
Got – 1	Initial	Got – 11	0
Got – 11	Initial	Got – 111	0
Got – 111	Initial	Got – 111	1

❖ We will use JK and D flip – flops for the Moore circuit implementation. The excitation tables for JK and D flip-flops (Table 3 & 4) are referenced to tabulated excitation table (See Table 5)

Table 3 : Excitation Table for JK flip-flop

Q (t)	**Q(t+1)**	**J**	**K**
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

Table 4: Excitation Table for D flip – flop

Q (t)	Q(t+1)	J
0	0	0
0	1	1
1	0	0
1	1	1

Table 5 : Excitation Table for the Moore Implementation

Inputs of Comb. Circuits			Next State		Outputs of Comb. Circuit			Output
Present State		Input			Flip-flop Inputs			
A	B	X	A	B	J_A	K_A	D_B	Z
0	0	0	0	0	0	X	0	0
0	0	1	0	1	0	X	1	0
0	1	0	0	0	0	X	0	0
0	1	1	1	0	1	X	0	0
1	0	0	0	0	X	1	0	0
1	0	1	1	1	X	0	1	0
1	1	0	0	0	X	1	0	1
1	1	1	1	1	X	0	1	1
1	1	1	1	1	X	0	1	1

❖ Simplifying Table 5 using maps, we get the following equations :

- $J_A = X.B$
- $K_A = X'$
- $D_B = X (A+B)$
- $Z = A.B$

❖ Note that the output is a function of present state values only.

❖ The circuit diagram for Moore machine circuit implementation is shown in Figure 3.25

- The timing diagram for Moore machine model is also shown in Figure 3.26
- There is no false output in a Moore model, since the output depends only on the state of the flop flops, which are synchronized with clock. The outputs remain valid throughout the logic state in Moore model.

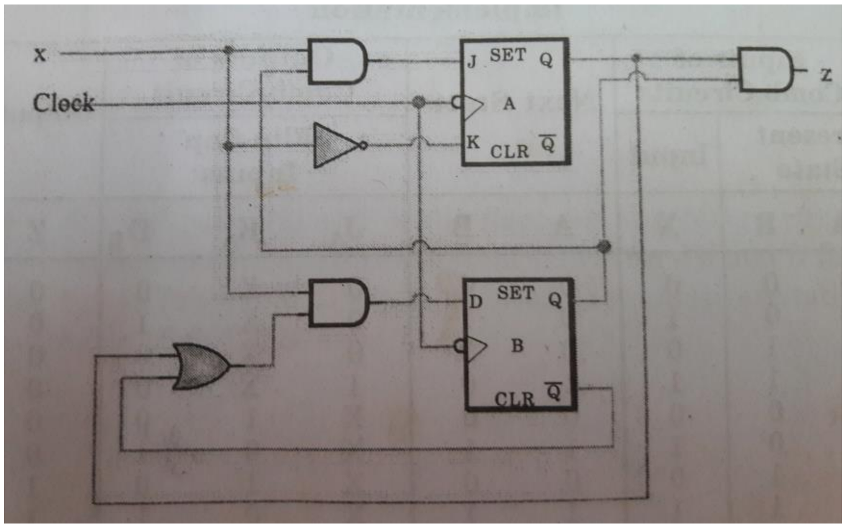


Fig3.25 Moore Machine Implementation for Sequence detector

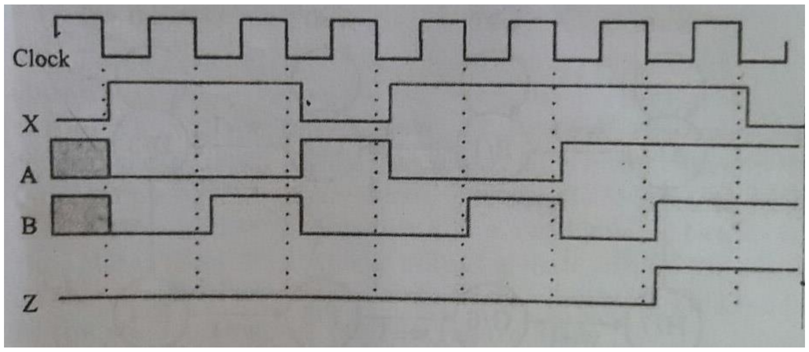


Figure 3.26 : Timing Diagram for Moore Model Sequence Detector.

DESIGN OF MODULO COUNTER (UPTO 3 BIT) WITH ONLY D FLIP FLOPS THROUGH STATE DIAGRAM

The counting sequence is 0, 1, 2,……., 6, 7,0,1,……

There exists an input signal w. The value of this signal is considered during each clock cycle. If w =0, the present count remains the same ; if w =1, the count is incremented.

STATE DIAGRAM AND STATE TABLE FOR A MODULO – 8 COUNTER

Figure 3.27 gives a state diagram for the desired counter. There is a state associated with each count. In the diagram state A corresponds to count 0, state B to count 1, and so on. We show the transitions between the states needed to implement the counting sequence. Note that the output signals are specified as depending only on the State of the counter at a given time, which is the Moore model of sequential circuits.

The state diagram may be represented in the state-table form as shown in figure 3.28

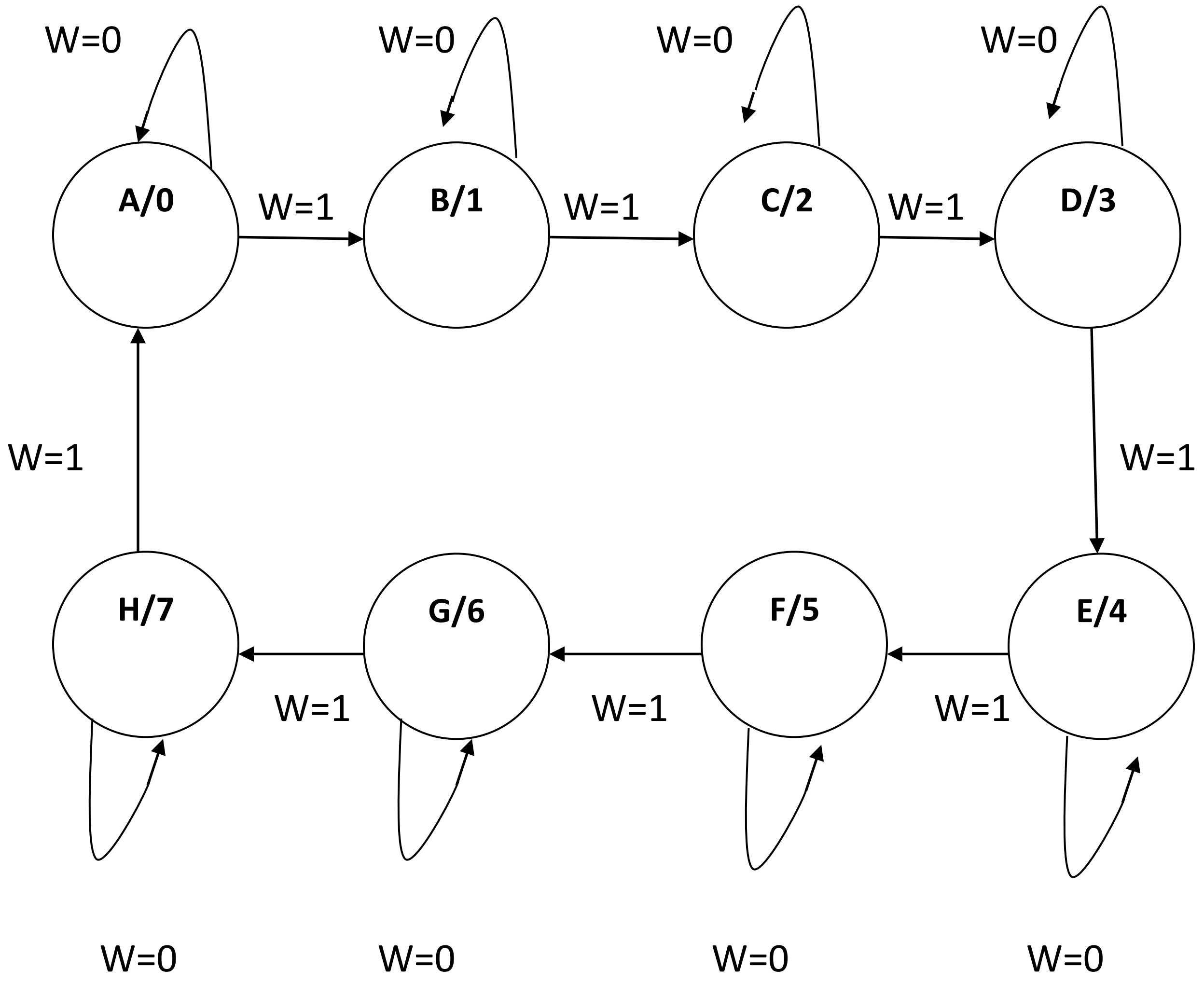


Fig3.27 State Diagram

Present State	Next State		Output
	ω =0	ω =1	
A	A	B	0
B	B	C	1
C	C	D	2
D	D	E	3
E	E	F	4
F	F	G	5
G	G	H	6
H	H	A	7

Figure . 3.28 State table for the counter

STATE ASSIGNMENT

Three state variable s are needed to represent the eight states. Let these variables, denoting the present state, be called y_2, y_1, and y_0. Let Y_2, Y_1, and Y_0 denote the corresponding next – state functions. The most convenient (and simplest) state assignment is to encode each state with the binary number that the counter should give as output in the state. Then the required output signals will be the same as the signals that represent the state variables. This leads to the state – assigned table in Figure 3.29

The final step in the design is to choose the type of flip-flops and derive the expressions that control the flip – flip-flop inputs. The most straight forward choice is to use d-type flip-flops.

	Present State $y_2y_1y_0$	**Next State**		**Output $z_2z_1z_0$**
		ω =0 $Y_2Y_1Y_0$	**ω =1 $Y_2Y_1Y_0$**	
A	000	000	001	000
B	001	001	010	001
C	010	010	011	010
D	011	011	100	011
E	100	100	101	100
F	101	101	110	101
G	110	110	111	110
H	111	111	000	111

Figure 3.29 State – assigned table for the counter

IMPLEMENTATION USING D - TYPE FLIP – FLOPS

When using d-type flip-flops to realize the finite state machine, each next – state function, Y_i, is connected to the D input of the flip-flop that implements the state variable y_i.

The next-state functions are derived from the information in Figure 3.12 Using Karnaugh maps in figure 3.13, we obtain the following implementation.

$D_o = Y_0 + wy_0 + wy_0$

$D_1 = Y_1 = wy_1 + y_1y_0 + wy_0y_1$

$D_2 = Y_2 = wy_2 + y_0y_2+y_1y_2 + wy_0y_1y_2$

The resulting circuit is given in Figure 3.14. It is not obvious how to extend this circuit to implement a larger counter, because no clear pattern is not found in the expressions for D_0, D_1 and D_2. However, we can rewrite these expressions as follows.

$$D_0 = wy_0 + wy_0$$
$$= w \oplus y_0$$

$D_1 = wy_1 + y_1y_0 + wy_0y_1$

$= (w+y_0)\ y_1 + wy_0y_1$

$= wy_0y_1 + wy_0y_1$

$= wy_0 \oplus y_1$

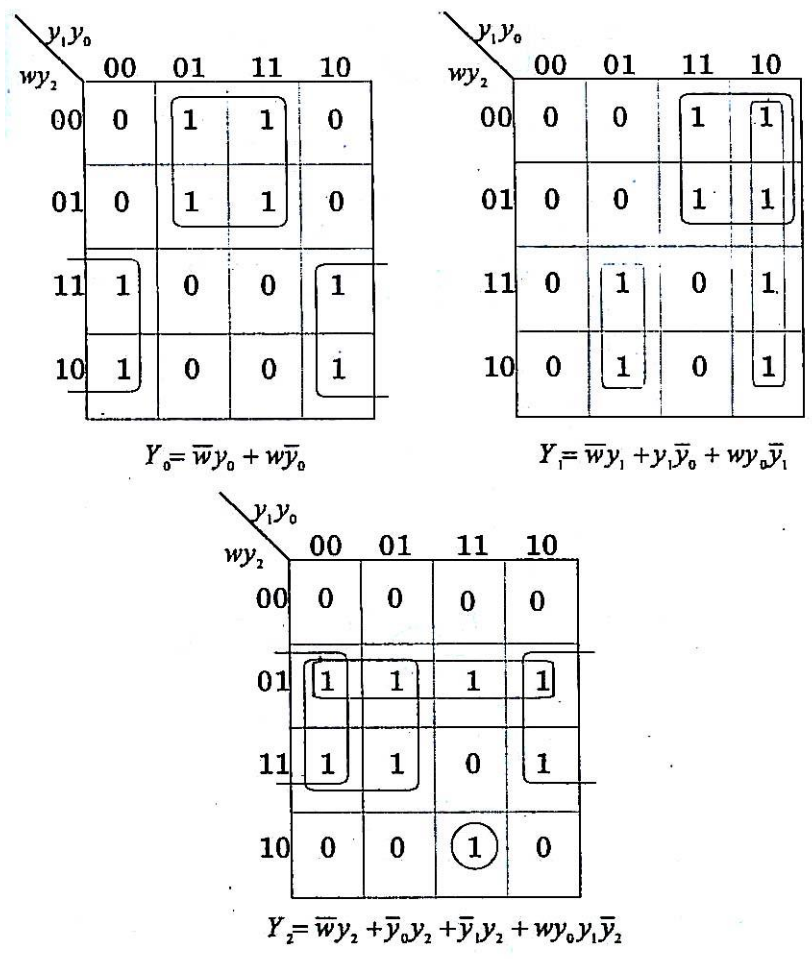

$Y_0 = \overline{w}y_0 + w\overline{y}_0$
$Y_1 = \overline{w}y_1 + y_1\overline{y}_0 + wy_0\overline{y}_1$
$Y_2 = \overline{w}y_2 + \overline{y}_0y_2 + \overline{y}_1y_2 + wy_0y_1\overline{y}_2$

Figure 3.13 Karnaugh maps for D Flip – flops for the counter

$D_2 \quad = \quad wy_2 + y_0y_2 + y_1y_2 + wy_0y_1y_2$

$= \quad (w+y_0+y_1)y_2 + wy_0y_1y_2$

$= \quad wy_0y_1y_2 + wy_0y_1y_2$

$= \quad wy_0y_1 \oplus y_2$

$Y_2 = \overline{w}.\ y_2 + \overline{y_0}.\ y_1 + \overline{y_1}.y_2 + w.\ y_0\ .\ \overline{y_2}$

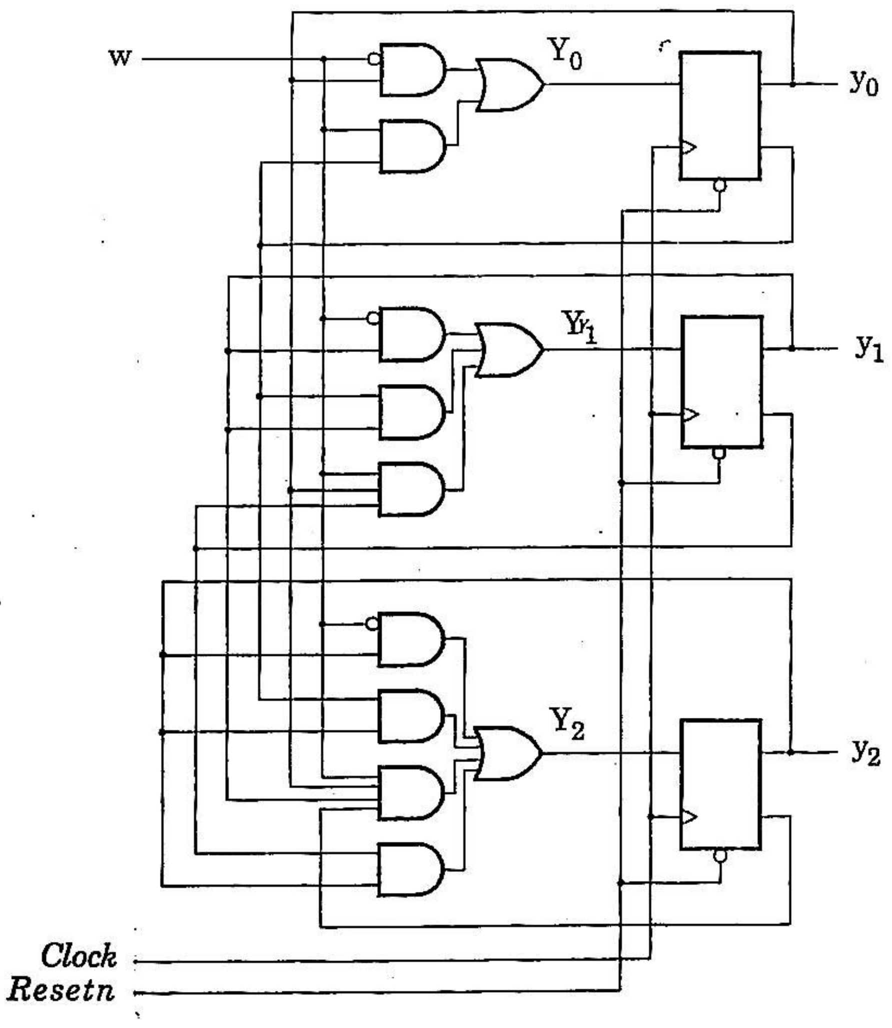
Figure 3.13 Karnaugh maps for D Flip – flops for the counter

$D_2 \quad = \quad \overline{w}y_2 + \overline{y_0}y_2 + \overline{y_1}y_2 + wy_0y_1\overline{y_2}$

$= \quad (\overline{w}+\overline{y_0}+\overline{y_1})y_2 + wy_0y_1\overline{y_2}$

$= \overline{wy_0y_1}y_2 + wy_0y_1\overline{y_2}$

$= wy_0y_1 \oplus y_2$



EXAMPLES :

Design a modulo 6 counter using D Flip Flops use proper excitation table & State diagram

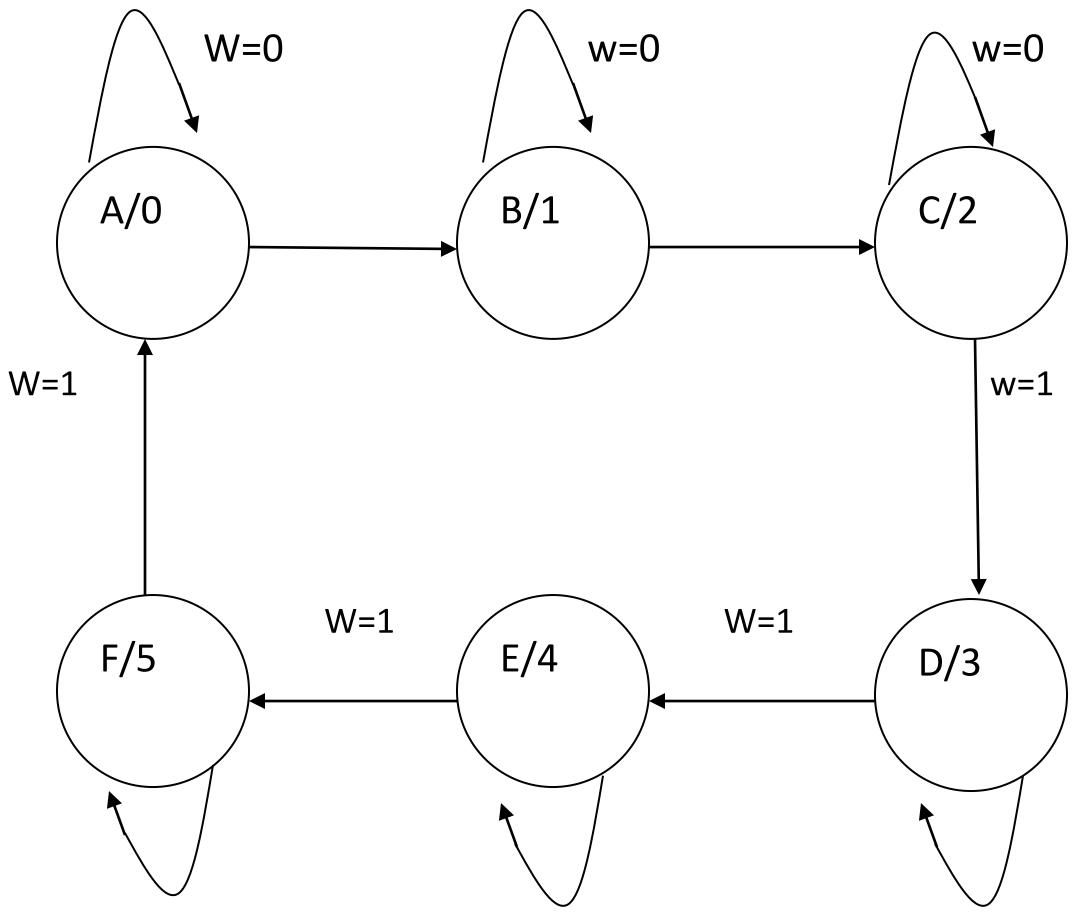
SOLUTION :

Step : 1

Count Sequence : $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$

Step 2 :

State Diagram



Step 3 :

State Table

Present State	**Next State**		**Output**
	ω =0	**ω =1**	
A	A	B	0
B	B	C	1
C	C	D	2
D	D	E	3
E	E	F	4
F	F	A	5

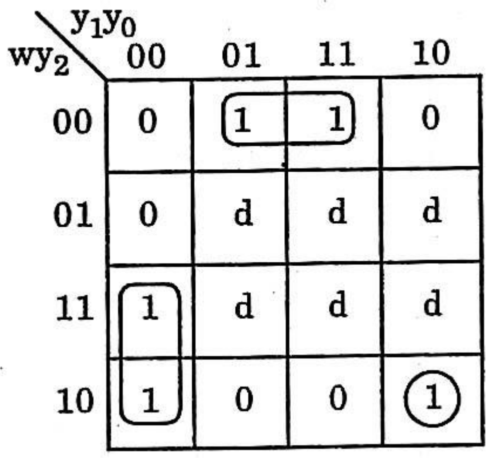
Step : 4

State Assigned Table

	Present State $y_2y_1y_0$	**Next State**		**Output** $z_2z_1z_0$
		ω =0 $Y_2Y_1Y_0$	**ω =1** $Y_2Y_1Y_0$	
A	000	000	001	000
B	001	001	010	001
C	010	010	011	010
D	011	011	100	011
E	100	100	101	100
F	101	101	000	101

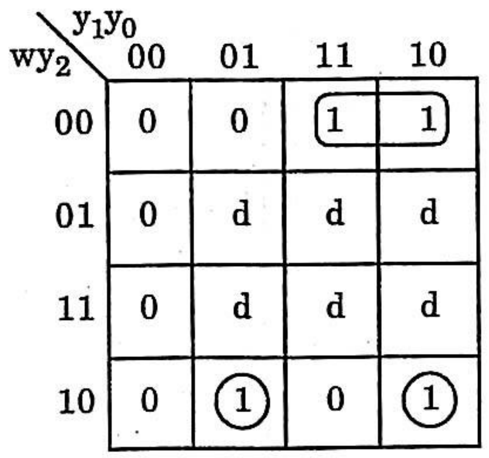
Step 5 :

K map implication



$wy_2 \backslash y_1y_0$	00	01	11	10
00	0	1	1	0
01	0	d	d	d
11	1	d	d	d
10	1	0	0	1

K map for y_0



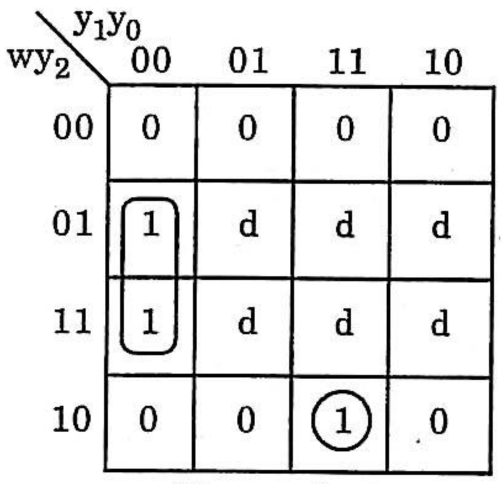
$wy_2 \backslash y_1y_0$	00	01	11	10
00	0	0	1	1
01	0	d	d	d
11	0	d	d	d
10	0	1	0	1

K map for y_1

$y_0=\overline{w}\,\overline{y}_2y_0+w\overline{y},y_0+w\overline{y}_2\,y_1\,y_0$

$y_1=\overline{w}\,\overline{y}_2y_1,+w\overline{y}_2$

$\therefore D=y_0=\overline{w}\,\overline{y}_2y_0+w\overline{y}_1\;y_0+w\overline{y}_2y_1y_0$ $\therefore D_1=y_1=\overline{w}\,\overline{y}_2y_1+w\overline{y}_2$



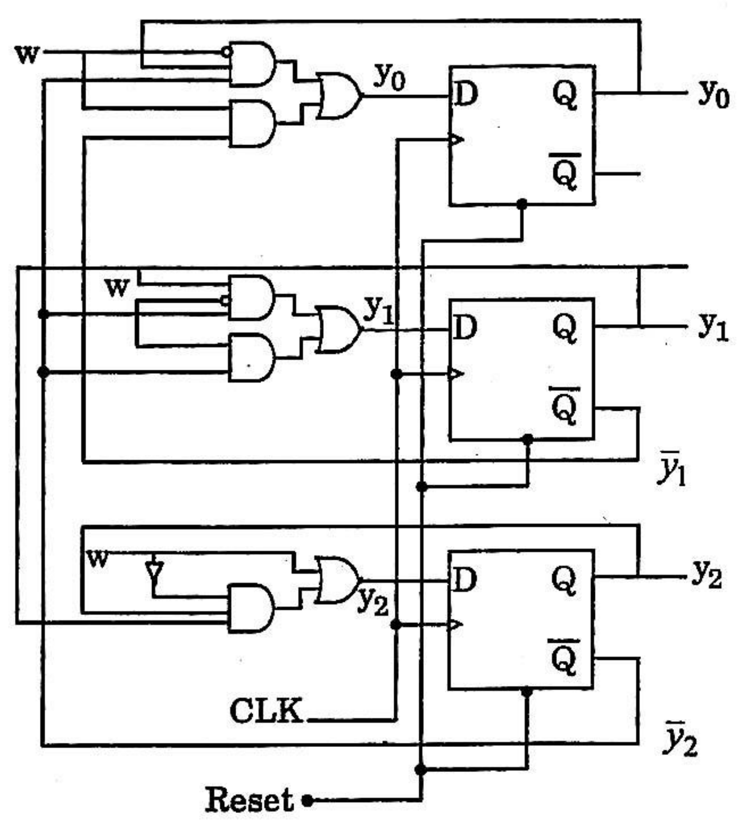
$wy_2 \backslash y_1y_0$	00	01	11	10
00	0	0	0	0
01	1	d	d	d
11	1	d	d	d
10	0	0	1	0

K map for y_2

$y_2=y_2\overline{y}_1\,y_0+wy_2y_1y_0$

$\therefore D_2=y_2=y_2\,\overline{y}_1\,y_0+w\overline{y}_2\,y_1\,y_0$

Step 6 : Logic diagram



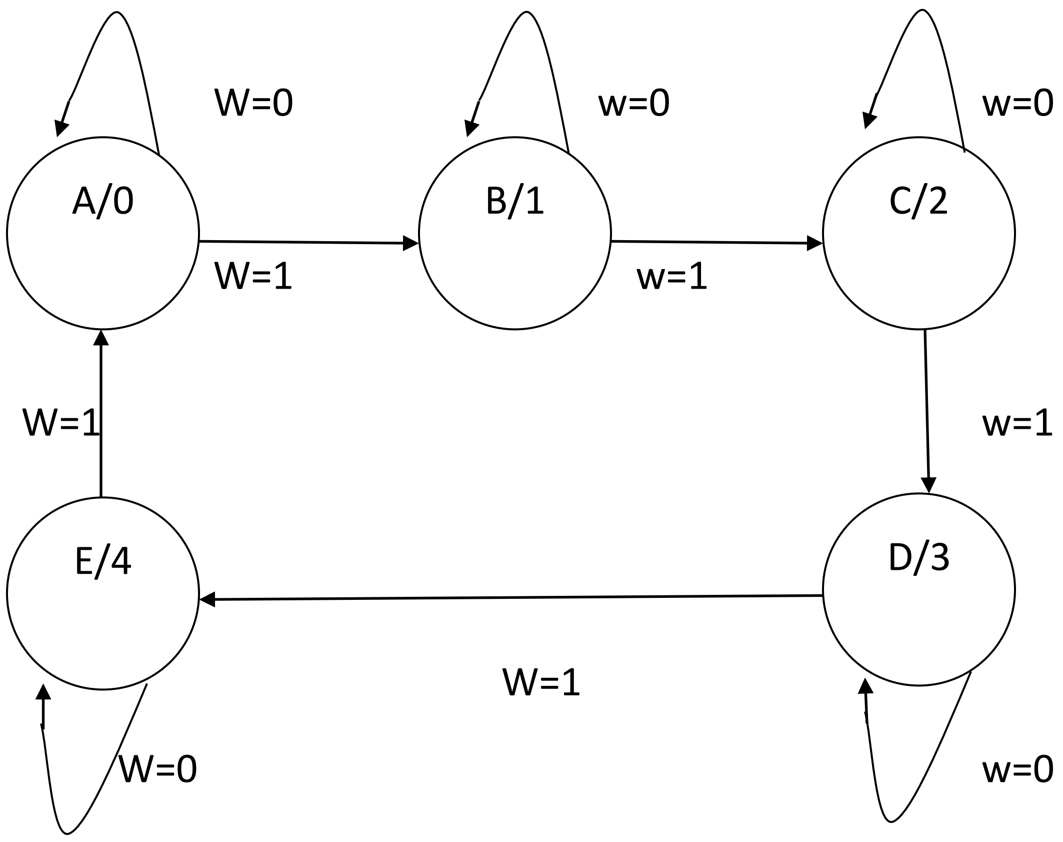
EXAMPLE:

Design a modulo 5 counter using D Flip Flops use proper excitation table & State diagram

SOLUTION

Step 1 : Count Sequence : $0 \longrightarrow 1 \longrightarrow 2 \longrightarrow 3 \longrightarrow 4$

Step 2 : State Diagram



Step 3 :

State Table

Present State	Next State		Output
	ω =0	ω =1	
A	A	B	0
B	B	C	1
C	C	D	2
D	D	E	3
E	E	F	4

Step 4 :

State Assigned Table

	Present State y_1y_0	Next State $\omega=0$ Y_1Y_0	Next State $\omega=1$ Y_1Y_0	Output z_1z_0
A	000	000	001	000
B	001	001	010	001
C	010	010	011	010
D	011	011	100	011
E	100	100	000	100

Step 5

K map simplification

	00	01	11	10
00	0	1	1	0
01	0	d	d	d
11	0	d	d	d
10	1	0	0	1

K map for y_0

	00	01	11	10
00	0	0	1	1
01	0	d	d	d
11	0	D	d	d
10	0	1	0	1

K map for y_1

$Y_0 = \overline{wy}_2y_0 + w\overline{y_2y_0}$

$Y_1 = \overline{wy}_2y_1 + w\overline{y_2}$

$D_0 = Y_0 = \overline{wy}_2y_0 + w\overline{y_2y_0}$

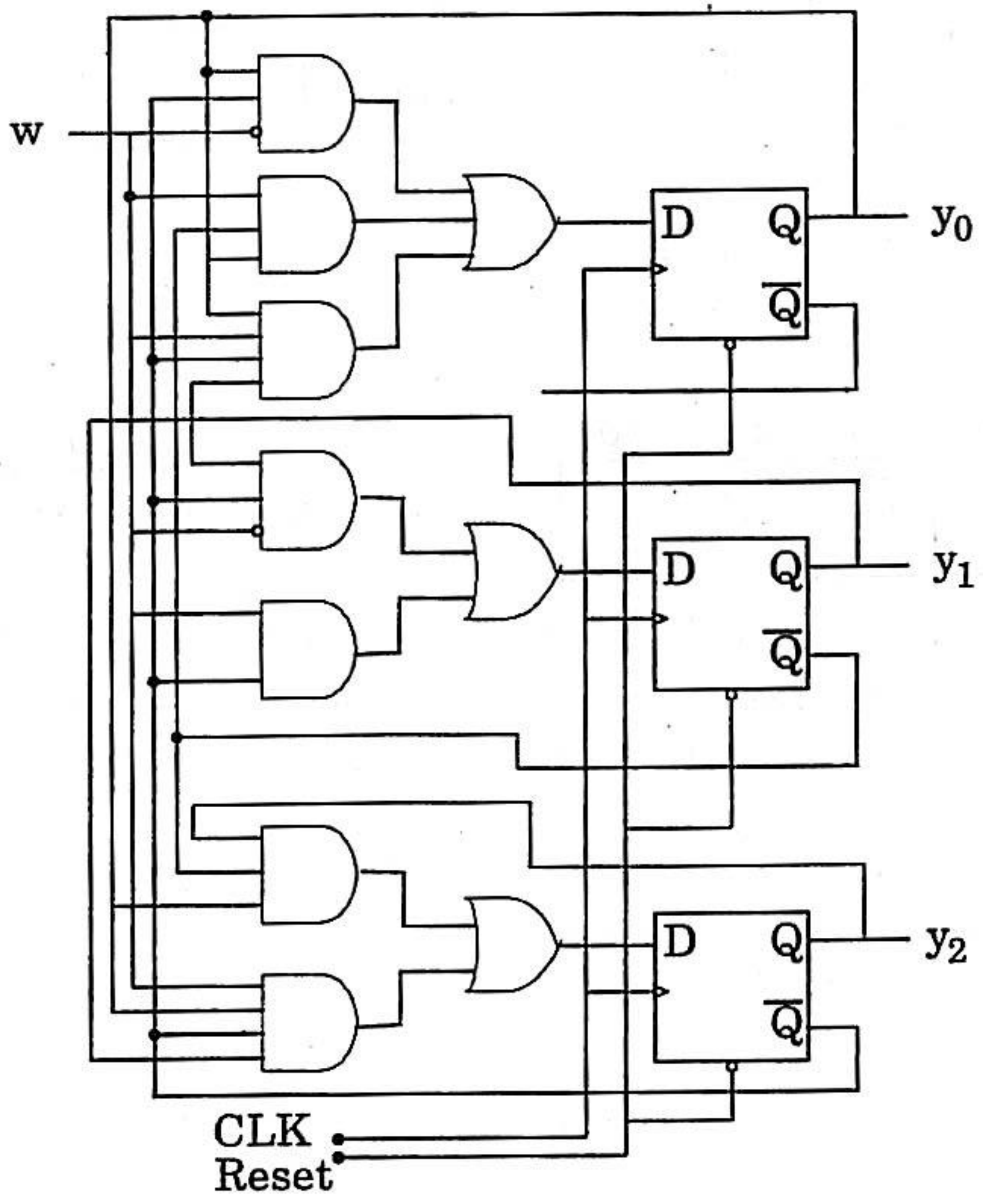
$D_1 = Y_1 = \overline{wy}_2y_1 + w\overline{y_2}$

	00	01	11	10
00	0	0	0	0
01	1	D	d	d
11	0	d	d	d
10	0	0	1	0

K map for y_2

$Y_2 = \overline{w} y_2 \overline{y_1} y_0 + w \overline{y_2} y_1 y_0$

$D_2 = Y_2 = \overline{w} y_2 \overline{y_1} y_0 + w \overline{y_2} y_1 y_0$


w
D
Q
$\overline{Q}$
y_0
D
Q
$\overline{Q}$
y_1
D
Q
$\overline{Q}$
y_2
CLK
Reset

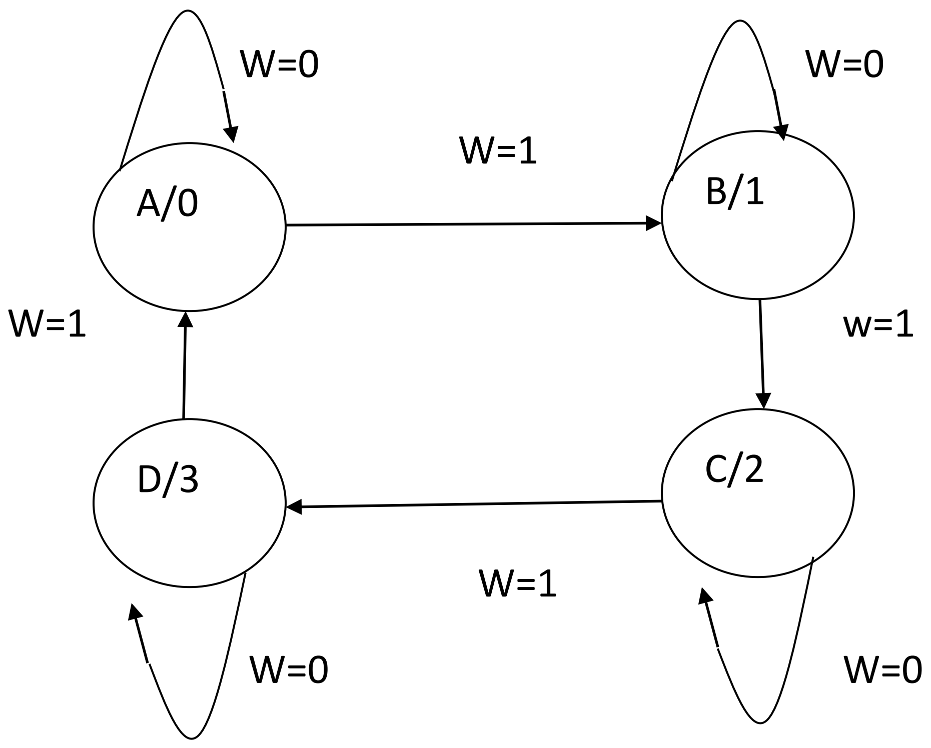
EXAMPLE:

Design a modulo 4 counter using D Flip Flops use proper excitation table & State diagram

Solution :

Step : 1 : Count Sequence $0 \longrightarrow 1 \longrightarrow 2 \longrightarrow 3$

Step : 2 : State Diagram



Step 3 :

State Table

Present State	Next State		Output
	ω =0	ω =1	
A	A	B	0
B	B	C	1
C	C	D	2
D	D	E	3

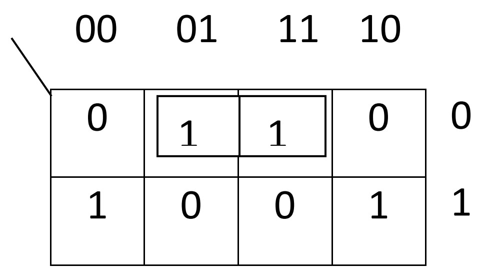
Step 4 :

State Assigned Table

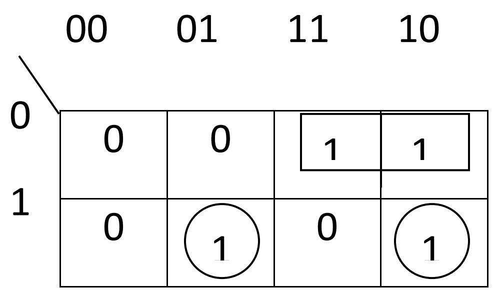
	Present State y_1y_0	**Next State**		**Output** z_1z_0
		ω =0 Y_1Y_0	**ω =1** Y_1Y_0	
A	00	00	01	00
B	01	01	10	01
C	10	10	11	10
D	11	11	00	11

Step 5

K map simplification



K Map For Y_0



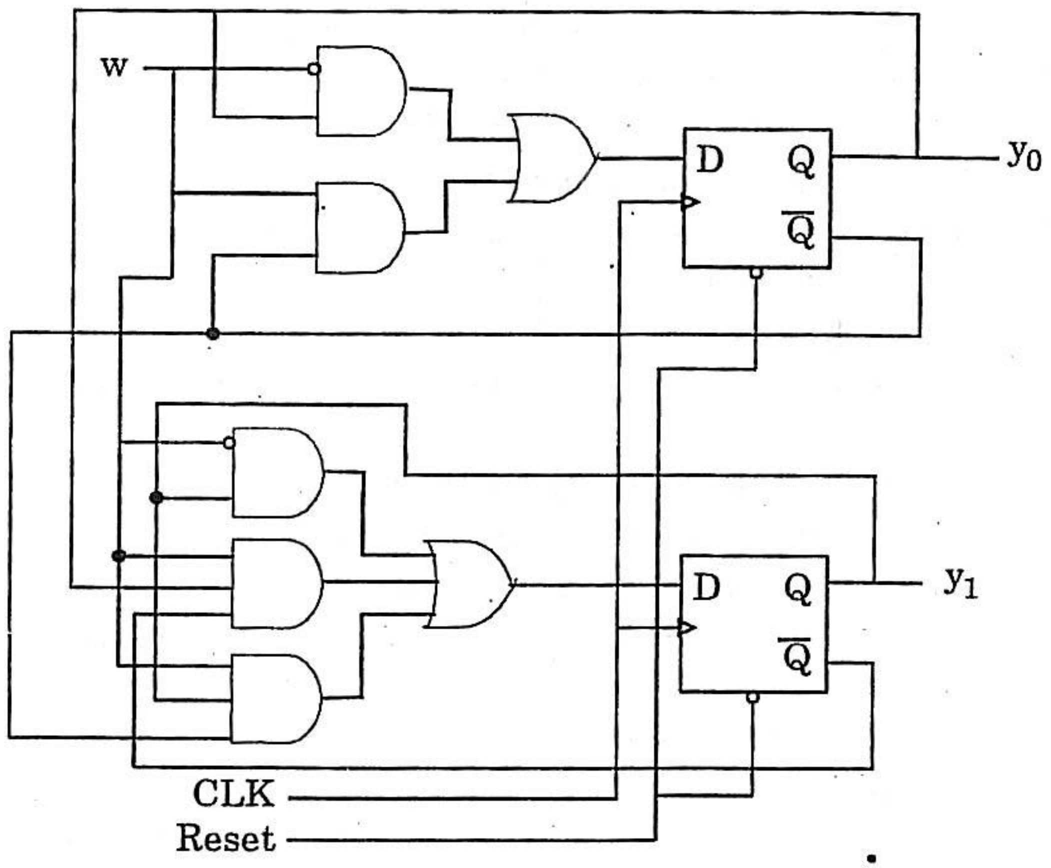
K Map For Y_1

$Y_0 = w \quad y_0$

$D0 = Y_0 = w \quad Y_0$

$Y_1 = \bar{w}y_1 + w\bar{y}_1y_0 + wy_1\bar{y}_0$

$D1 = wy_1 + w\bar{y}_1y_0 + wy_1\bar{y}_0$


w
D
Q
$\overline{Q}$
y_0
D
Q
$\overline{Q}$
y_1
CLK
Reset

UNIT – IV

4.1 VHDL CODE FOR SEQUENTIAL CIRCUIT

VHDL Constructs for storage elements with reset input

VHDL code for a Gated D Latch

```
LIBRARY ieee;
USE ieee.std_logic-1164.all;
          ENTITY latch is PORT (D, clk :      IN STD – LOGIC;
                              Q     :      OUT STD – LOGIC;
END latch;

ARCHITECTURE Behavior OF latch IS BEGIN
          PROCESS (D, clk)
          BEGIN
               IF clk        =      '1' THEN
                    Q<=D;
               END PROCESS;
          END Behavior;
```

VHDL code for D flip with Reset input

```
library IEEE;
use IEEE.std_logic_1664.all;

entity d_ff_srss is
port (
d,clk,reset,set : in STD_LOGIC;
q : out STD_LOGIC);
end d_ff_
ARCHITECTURE Behavior OF srss of  d_ff_
```

```
begin
process(clk)
begin
if clk’event and clk=’1’ then
if reset=’1’ then
q <= ‘0’;
elsif set =’1’ then
q <= ‘0’;
elsif set =’1’ then
q <= ‘1’;
else
q <=d;
end if;
end if;
end process;
end d_ff_srss;
q <=d;
end if;
end if;
end process;
end d_ff_srss;
```

<u>VHDL Code for D.. Flip Flop Without reset Input</u>

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity DFF1 is
Port (D : in std_logi;
        CLK : in std_logic;
        Q : out std_logic;
        QN : out std_logic;
end DFF1;

architecture Behavioral of DFF1 is
begin
        process (CLK)
        begin
                        if CLK = ‘1’ then
                        Q <= D;
                        QN <=NOT D;
                        end if;
        end process;
end Behavioral;
```

<u>VHDL Code for JK Flip flop with reset Input</u>

```
Library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity JKFF3 IS
        Port (CLOCK : in std_logic;
        J : in std_logic;
        K : in std_logic;
        REST : in std_logic;
        Q : out std_logic;
        QBAR : out std_logic);
end JKFF3;

architecture Behavioral of JKFF3 is
signal state : std_logic;
signal input : std_logic_vector (1 downto 0);
begin
        input < = J & K;
        p:procees (CLOCK, RESET) is
        begin
                        if RESET =’1’ then
                        state<=’0’
                        elsif (rising_edge (CLOCK) then
                        case(input) is
                        when”11”=>
                        state<=not state;
                        when “10” =>
                        state<=’1’;
                        when “01”=>
                        state<=’0’;
                        when other=>null;
                        end case;
```

```
                    end if;
                end process;
            end Behavioral;
```

VHDL Code for JK Flip flop without reset Input

```
Library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity JKFLIPELOP1 is
Port (J : in std_logic;
    K : in std_logic;
    CLK : in std_logic;
    Q : inout std_logic;
    QN : inout std_logic);
end JKFLIPFLOP1;

architecture Behaviroral of JKFLIPFLOP1 is
begin
    process (CLK,J,K)
    begin
        if (CLK=’1’ and CLK’ event) then
        if (J=’0’ and K=’0’) then
            Q <=Q
            QN <+QN;
        elsif(J=’0’ and K=’1’) then
            Q<= ‘1’;
            QN <= ‘0’;
```

```
                elsif(J=’1’ and K=’0’) then
                        Q <= ‘0’;
                        QN <= ‘1’;
                elsif (J=’1’ and K=’0’) then
                        Q <=’0’;
                        QN <= ‘1’;
                elsif(J=’1’ and K=’1’) then
                        Q <= NOT Q;
                        QN <= NOT QN’
                end if;
                end if’
        end process;
end Behaviroal;
```

VHDL code for T FF with reset input

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity tff is
Port (clk : in STD_LOGIC;
reset : in STD_LOGIC;
t : in STD_LOGIC;
q : out STD_LOGIC;
end tff;
architecture Behavioral of tff is
signal q_reg: std_logic;
signal q_next: std_logic;
begin
```

```
process(clk)
begin
if (reset = ‘1’) then
q_reg <= ‘0’; elsif (clk’ event and clk = ‘1’) then
 q_reg <= q_next;
end if;
end process;
q_next <= q_reg
when t = ‘0’ else not (q_reg);
q <= q_reg;
end Behaviroal;
```

VHDL Program for T Flip-Flop without Reset input

```
Library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSINED.ALL;
entity tflip is
port (T, CLK:in bit;
Q : inout bit;
QN : out bit);
end tflip;
architecture Behaviroal of tflip is
begin
process (CLK)
begin
if CLK =’0’ and CLK’ event then
Q <= (T and not Q) or (not T and Q) after 10 ns;
end if;
QN<=NOT Q;
```

```
END PROCESS;
END BEHAVIORAL;
```

4.2 VHDL EXAMPLES

Counters (up to 3 bits)

(i) Three – bit up – Counter with synchronous reset

```
        LIBRARY ieee ;

        USE ieee.std_logic_1164.all;

        USE ieee.std_logic_unsigend.all;



        ENTITY upcounter IS

                PORT (clock         : IN STD_LOGIC ;

                        clear         : IN STD_LOGIC ;

                        q             : OUT STD_LOGIC_VECTOR

                                        (2 DOWNTO 0)) ;

        END upcounter ;



        ARCHITECTURE behavior OF upcounter IS

        SIGNAL count  :STD_LOGIC_VECTOR (2 DOWN TO 0);

        BEGIN

                IF clock'EVENT AND clock = '1' THEN

                        IF clear = '1' THEN
                        count< = “000” ;
                    ELSE
                        count<= count + ‘1’ ;
                    END IF ;
                END IF ;
            END PROCESS ;
                q <= count ;
            END behavior ;
```

3- bit up counter with reset

Truth Table

Clock	**Count (2)**	**Count (1)**	**Count (0)**
Clear	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

NOTE :

If we want to design 2 bit up counter the only change is the width of q is changed as 2 bit as shown

[q : OUT STD_LOGIC_VECTOR (1 DOWNTO 0)) ;]

(ii) Three – bit down – counter with synchronous reset

```
LIBRARY ieee ;

USE ieee.std_logic_1164.all;

USE ieee.std_logic_unsigned.all;

ENTITY downcounter IS

    PORT (clock     : IN STD_LOGIC ;

          clear     : IN STD_LOGIC ;

          q         : OUT STD_LOGIC_VECTOR

                        (2 DOWNTO 0));

END downcounter ;


ARCHITECTURE  behavior  OF  downcounter  IS

SIGNAL count :STD_LOGIC_VECTOR (2 DOWNTO 0);

BEGIN
```

```
downcounter : PROCESS (clock)
BEGIN
    IF clock’EVENT AND clock = ‘1’ THEN
        IF clear = ‘1’ THEN
            count< = “000” ;
        ELSE
            count< = count – ‘1’ ;
        END IF ;
    END IF ;
END PROCESS ;
q < = count ;
END behavior ;
```

<u>3-bit down counter with reset :</u>Truth Table

Clock	**Count (2)**	**Count (1)**	**Count (0)**
Clear	0	0	0
1	1	1	1
2	1	1	0
3	1	0	1
4	1	0	0
5	0	1	1
6	0	1	0
7	0	0	1

Note :

If we want to design 2 bit downcounter the only change is the width of q is changed as 2 bit as shown.

[q : OUT STD_LOGIC_VECTOR (1 DOWNTO 0));]

VHDL Program for decade counter

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

    ENTITY modcounter IS
    PORT (clock : IN STD_LOGIC ;
          clear  : IN STD_LOGIC ;
    q    : OUT STD_LOGIC_VECTOR (3 DOWNTO 0)) ;
    END modcounter ;

    ARCHITECTURE behavior OF mod counter IS
    SIGNAL count: STD_LOGIC_VECTOR (3 DOWNTO 0);
    BEGIN
    modcounter: PROCESS (clock)
          BEGIN
               IF clock ‘EVENT AND clock = ‘1’ THEN
                    IF (clear = ‘1’ OR count = “1001”) THEN
                         count < = “0000”;
                    ELSE
                        count <= count + ‘1’;
                  END IF;
              END IF;
       END PROCESS
              q < = count;
       END behavior;
```

Modulo 5 up counter

Truth Table

Clock	Count (3)	Count (2)	Count (1)	Count (0)
Clear	0	0	0	0
1	0	0	0	0
2	0	0	0	1
3	0	0	1	0
4	0	0	1	1
5	0	1	0	0
6	0	1	0	1
7	0	1	1	0
8	0	1	1	1
9	1	0	0	0
10	1	0	0	1
11	0	0	0	0
12	0	0	0	1

Note:

If we want to design modulo 6 counter it means that it has to count up to 5 [101] and for the next clock the value is 000. So the only change in the program is line number '15' the change is mentioned below.

IF (clear = '1' OR count = "101") THEN

(iv) 3 bit up / down counter with synchronous reset

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

ENTITY modcounter IS
PORT (clock: IN STD_LOGIC;
        clear: IN STD_LOGIC;
        select: IN STD_LOGIC;
q: OUT STD_LOGIC_VECTOR (2DOWNTO 0)) ;
END updowncounter;
ARCHITECTURE behavior OF updowncounter IS
SIGNAL count: STD_LOGIC_VECTOR (2DOWNTO 0);
BEGIN
updowncounter: PROCESS (clock)
      BEGIN
          IF clock ‘EVENT AND clock = ‘1’ THEN
              IF clear = ‘1’ THEN
                  count < = “000” ;
ELSIF select = ‘1’
                  count <= count + ‘1’;
  ELSE
                  count <= count – ‘1’;
          END IF;
      END IF
   END PROCESS
          q < = count;
   END behavior;
```

3-bit up / down counter

Clock	**Count (2)**	**Count (1)**	**Count (0)**
Clear	1	0	0
1	1	0	1
2	1	0	0
3	1	0	1
4	1	1	0
5	1	1	1
6	1	1	0
7	1	1	1
8	0	1	0
9	0	1	1
10	0	1	0
11	0	1	1
12	0	0	0
13	0	0	1

Explanation:

- The code defines an entity named up-down counter.
- It has the inputs clock, select, clear (synchronous reset input) and the output q.
- The process sensitivity list includes clock. Because the value of q depends on the changes in the value of this signal.
- During positive clock edge (if clock ‘EVENT AND clock = ‘1’) only the output q is changed. If clear input is equal to 1, the output count = 000. Else it will check the condition select input, if select = 1 count = count + 1. Else the output count = count-1, and it is assigned to q.

Johnson Counter

In a Johnson counter, inverted output of the last stage flip-flop is fed back to the input of the first stage flip flop.

Clock	**L**	**Q_0**	**Q_0**	**Q_0**	**Q_0**
0	1	0	0	0	0
1	0	1	0	0	0
2	0	1	1	0	0
3	0	1	1	1	0
4	0	1	1	1	1
5	0	0	1	1	1
6	0	0	0	1	1
7	0	0	0	0	1
8	0	0	0	0	0
9	0	1	0	0	0

Table Counting sequence of 4 bit Johnson Counter

Program

```
LIBRARY ieee;
USE ieee.std_logic_1164.911
USE ieee.std_logic_unsigend.all;
ENTITY Johnson IS
     PORT (clock     : IN STD_LOGIC;
          C          : IN STD_LOGIC;
          Q          : BUFFERSTD_LOGIC_VECTOR
                       (3 DOWNTO 0));
END Johnson;
```

```
ARCHITECTURE Behavior of Johnson IS
BEGIN
        PROCESS (clock, C)
        BEGIN
        WAIT UNTIL clock ‘EVENT AND clock = ‘1’;
        IF C = ‘1’ THEN
           Q < = “0000”
        ELSE
        Q (3) < = Q (2);
        Q (2) < = Q (1);
            Q (1) < = Q (0);
            Q (0) <= Q (3);
        END IF;
END PROCESS;
        q <= count;
END behavior;
```

INTRODUCTION TO PROM, PLA & PAL

Programmable Read Only Memory (PROM)

A programmable read only memory (PROM) is a device that includes both the decoder and the OR gates within a single IC package. The Fig. no 5.1 shows the block diagram of PROM. It consists of n input lines and m output lines. Each bit combination of the input variables is called as an address. Each bit combination that comes out of the output lines is called as a word. The number of bits per word is equal to the number of output lines, m. The address specified in binary number denotes one of the minterms of n variables. The number of distinct addresses possible with n input variables is 2^n distinct addresses in PROM, there are 2^n distinct words in the PROM. The words available on the output lines at any given time depends on the address value applied to the input lines.

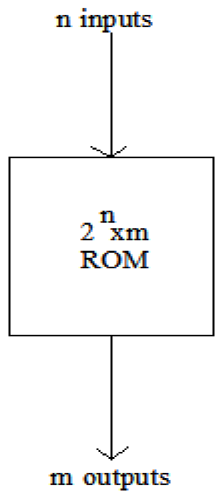


Fig.N0:5.1 Block diagram of PROM

Let us consider 64 x 4 PROM. The PROM consists of 64 words which consists of 4-bits each. This indicates that there are four output lines and particular word from 64 words presently available on the output lines is determined from the six input lines. There are only six inputs in a 64x4 PROM because $2^6 = 64$ and with six variables, it can specify 64 addresses or minterms. For each address input, there is a unique selected word. Thus, if the input address is 000000, word number 0 is selected and applied to the output lines. If the input address is 111111, word number 63 is selected and applied to the output lines.

The Fig.no 5.2 shows the internal logic construction of a 64 x 4 PROM. The six input variables are decoded in 64 lines by means of 64 AND gates and 6 inverters. Each output of the decoder represents one of the minterms function of six variables. The 64 outputs of the decoder are connected through fuses to each OR gate. Only four of these fuses are shown in the diagram, but actually each OR gate has 64 inputs and each input goes through a fuse that can be shown as desired.

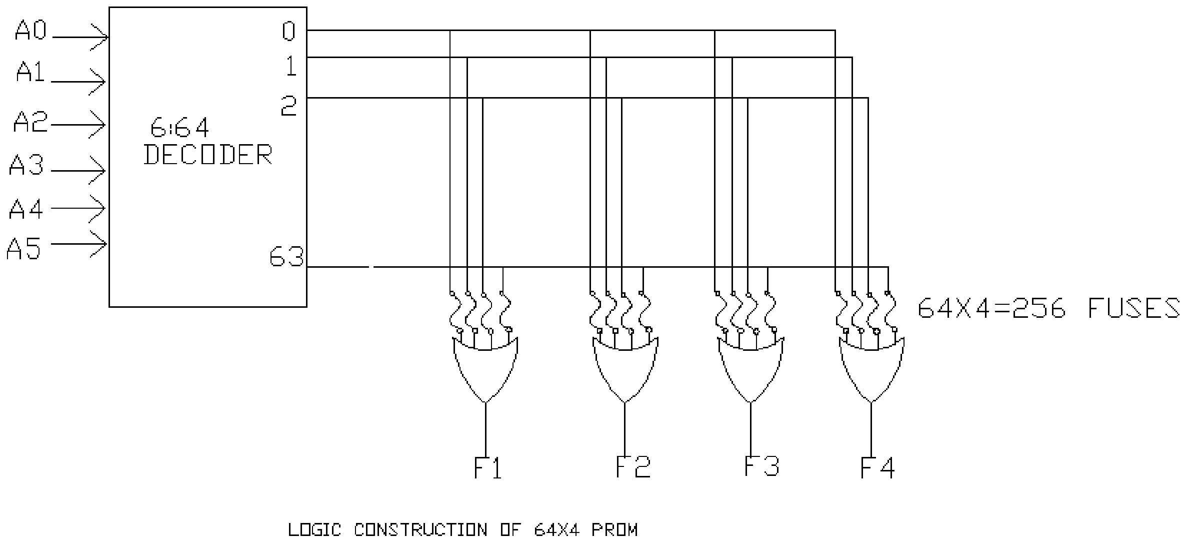


Fig.N0:5.2 Logic construction of 64 x4 PROM

Combinational Logic Implementation using PROM

By Looking, at the logic diagram of the PROM, each output provides the sum of all the minterms of n input variables. (i.e., any Boolean function can be expressed in sum of minterms). By breaking the links of those minterms not included in the function, each PROM output can be made to represent the Boolean function of one of the output variables in the combinational circuit. For an n-input, m-output combinational circuit, it needs a 2^nx m PROM.

Example:

Using PROM realize the following expressions.

F_1 (a,b,c) = Σ m (0,1,3,5,7)

F_2 (a,b,c) = Σ m (1,2,5,6)

Solution:

The given functions have three inputs. They generate $2^3 = 8$ minterms and since there are two functions, there are two outputs. The functions can be realized as shown in Fig no 5.3

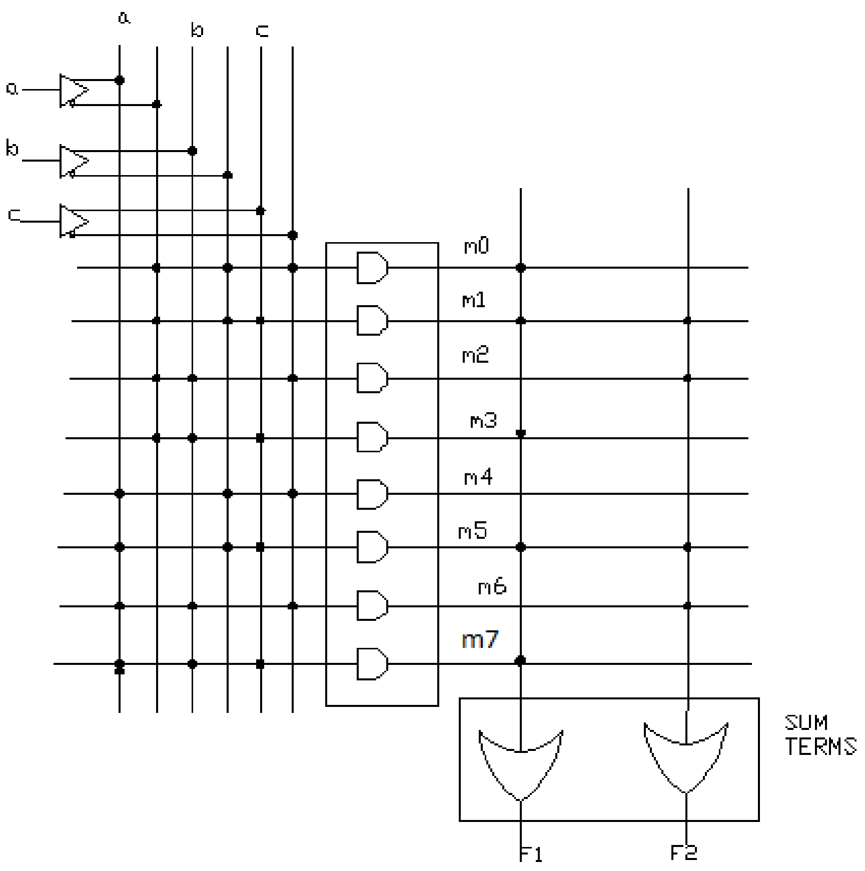


Fig.No:5.3 2^nx m PROM

Example:

Design a combination circuit using a PROM. The circuit accepts 3-bit binary number and generates its equivalent Excess – 3 codes.

Solution:

Let us derive the truth table for the given combinational circuit. Table shows the truth table.

Table No: 5.1 : Truth table for 3-bit binary to Excess -3 converter

Inputs			Outputs			
A_2	A_1	A_0	B_3	B_2	B_1	B_0
0	0	0	0	0	1	1
0	0	1	0	1	0	0
0	1	0	0	1	0	1
0	1	1	0	1	1	0
1	0	0	0	1	1	1
1	0	1	1	0	0	0
1	1	0	1	0	0	1
1	1	1	1	0	1	0

In practice while the process of designing combinational circuits with PROM, it is not necessary to show the internal gate connections of fuses inside the unit, as shown in the Fig no.5.3. This was shown for demonstration purpose only. The designer has to specify only the PROM (inputs and outputs) and its truth table, as shown in the fig.

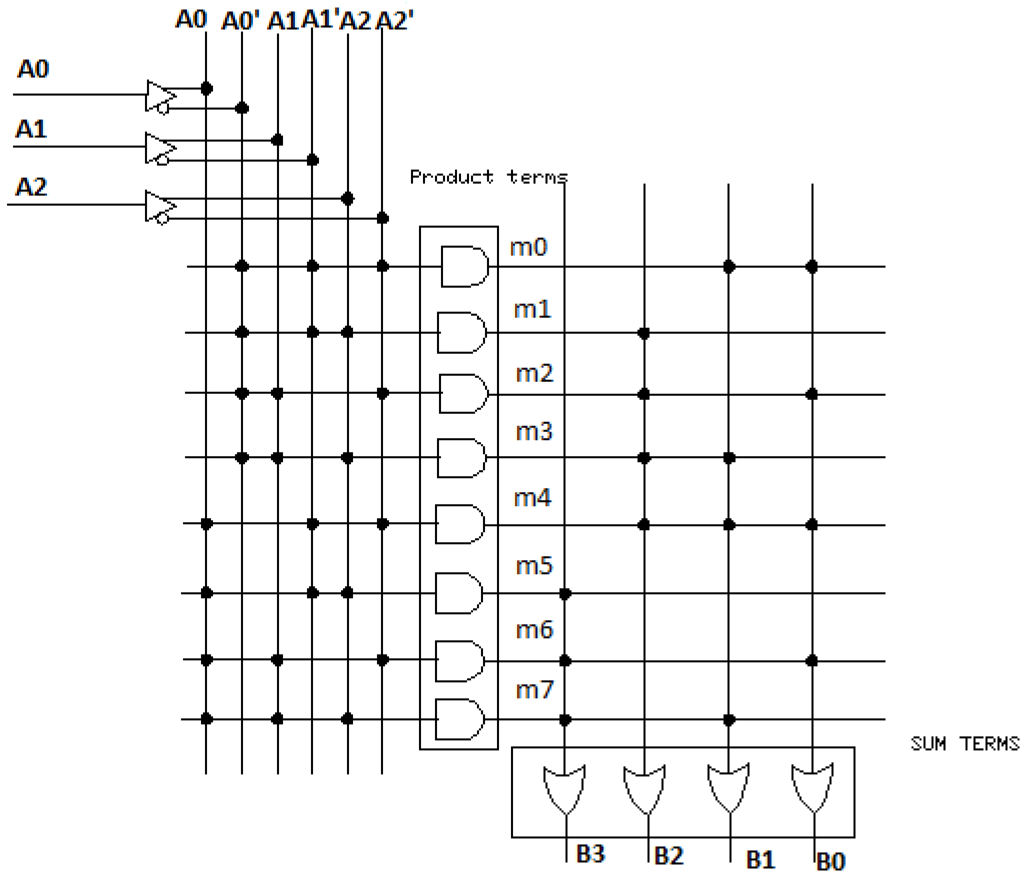


Fig.N0:5.4 combinational circuits with PROM

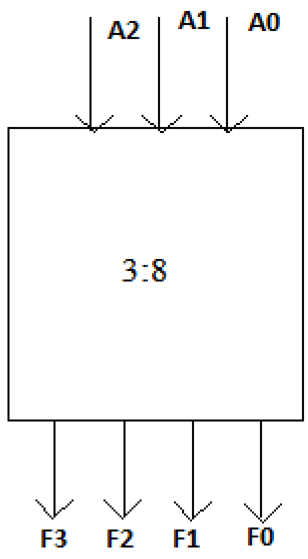


Fig.No:5.5 BLOCK DIAGRAM OF PROM

Table No: 5.2 PROM truth table

A_2	A_1	A_0	F_3	F_2	F_1	F_0
0	0	0	0	0	1	1
0	0	1	0	1	0	0
0	1	0	0	1	0	1
0	1	1	0	1	1	0
1	0	0	0	1	1	1
1	0	1	1	0	0	0
1	1	0	1	0	0	1
1	1	1	1	0	1	0

PROGRAMMABLE LOGIC ARRAY (PLA)

Several types of PLDs are commercially available. The first developed was the programmable logic array (PLA). The general structure of a PLA is shown in Figure no 5.6. Based on the idea that logic functions can be realized in sum-of-products form, a PLA consists of a collection of AND gates that feeds a set of OR gates. As shown in the figure, the PLA's inputs x_1,................ x_n pass through a set of buffers (which provide both the true value and the complement of each input) into a circuit block called an, AND plane, or AND array.

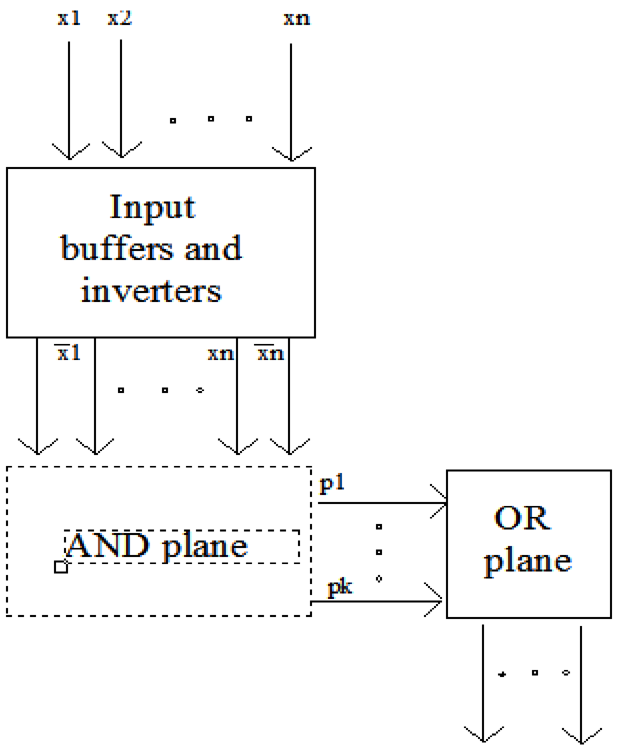


Fig.No:5.6 BLOCK DIAGRAM OF PLA

The AND plane produces a set of product terms P_1.........P_k. Each output can be configured to realize any sum of P_1,..........,P_k and hence any sum-of-products function as the PLA inputs.

A more detailed diagram of a small PLA is given in Figure no 5.7, which shows a PLA with three inputs, four product terms, and two outputs. Each AND gate in the AND plane has six inputs, corresponding to the true and complementing versions of the three input signals.

Each connection to an AND gate is programmable; a signal that is connected to an AND gate is indicated with a wavy line, and a signal that is not connected to the gate is shown with a broken line. The circuitry is designed in such a way that any unconnected AND – gate inputs do not affect the output of the AND gate.

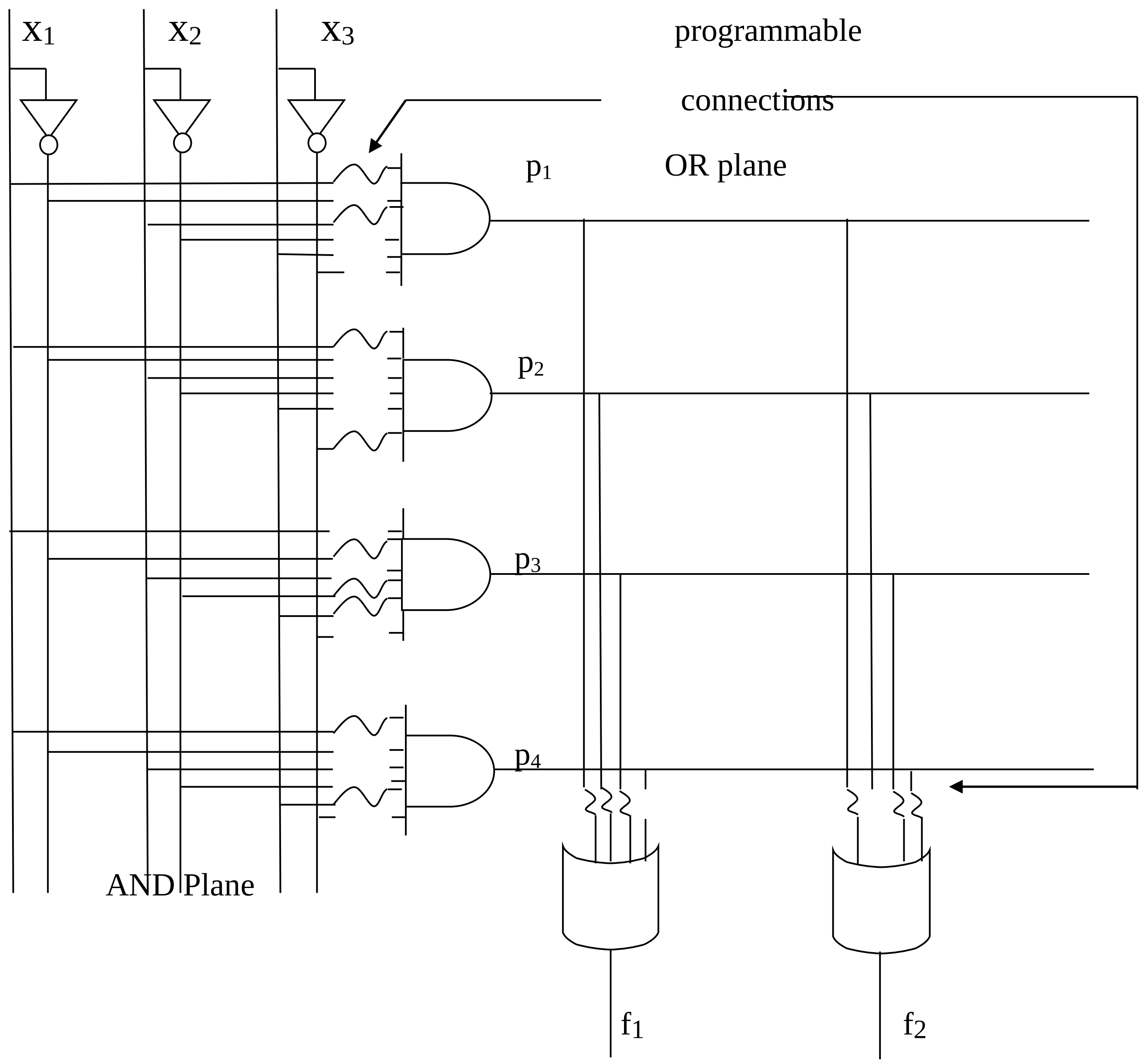


Fig 5.7 Gate Level Diagram of a PLA

In Figure no.5.8 the AND gate that produces P_1 is shown connected to the inputs x_1 and x_2. Hence $P_1 = x_1x_2$ Similarly, $P_2 = x_1x_3$, $P_3 = x_1x_2x_3$ and $P_4 = x_1\overline{x_3}$. Programmable connections also exist for the OR plane. Output f_1 is connected to product terms P_1, P_2, and P_3. It is therefore realizes the function $f_1 = x_1x_2+x_1x_3+x_1x_2x_3$ Similarly, output $f_2 = x_1x_2+ x_1x_2x_3+ \overline{x_1x_3}$ Although shows the PLA programmed to implement the functions described above, by programming the AND and OR planes differently, in which each of the output f_1 and f_2 could implement various functions of x_1, x_2, and x_3. The only constraint on the function is that can be implemented is the size of the AND plane because it products only four product terms.

Although Figure no.5.7 illustrates clearly the functional structure of a PLA, this style of drawing is not suitable for larger chips. Instead it has become customary in technical literature to use the style shown in Figure no 5.8. Each AND gate is depicted as a single horizontal line attached to an AND –gate symbol. The possible inputs to the AND gate are drawn as vertical lines that cross the horizontal line. At any crossing of a vertical and horizontal line, a programmable connection indicated by an x. Figure 5.8 shows the programmable connections needed to implement the product terms in Figure no 5.7. Each OR gate is drawn in a similar manner, with a vertical line attached to an OR-gate symbol. The AND gate outputs cross these lines, and corresponding programmable connections can be formed. The figure illustrates the programmable connections that produce the functions f_1 and f_2 from figure 5.7.

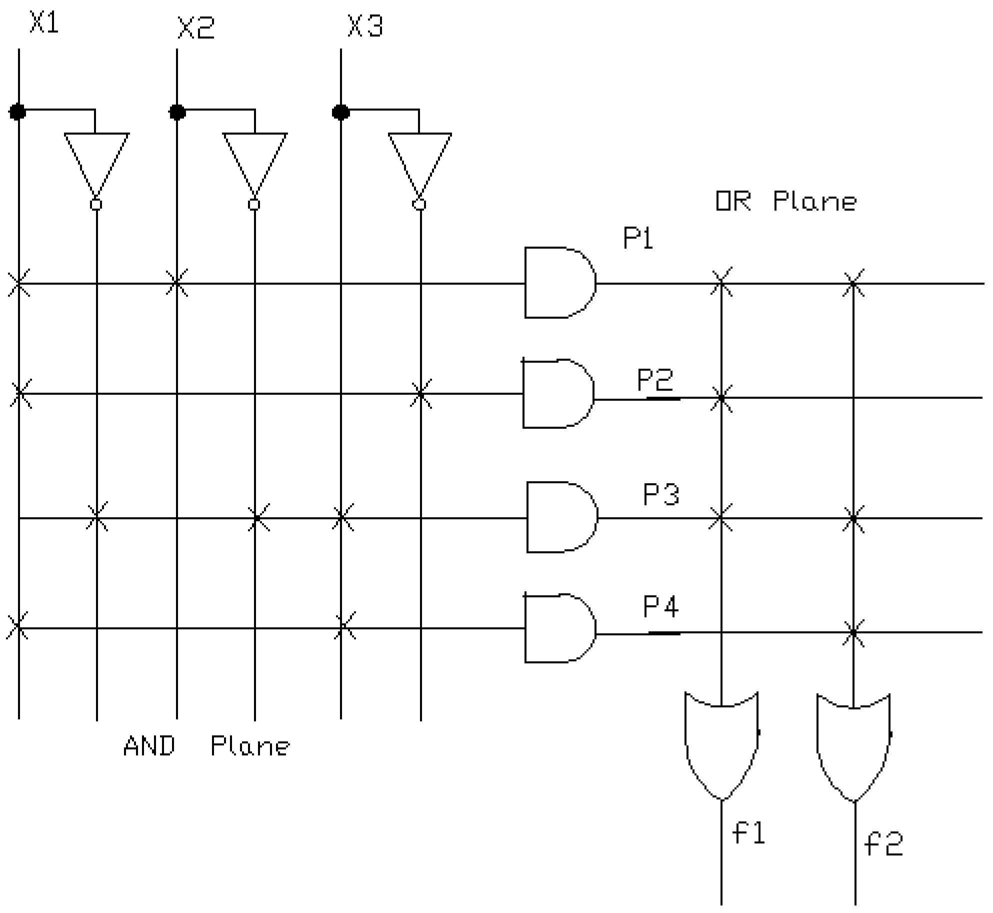


Fig.No:5.8 Customary Schematic for the PLA in figure 5.7

PROGRAMMABLE ARRAY LOGIC (PAL)

In a PLA both the AND and OR planes are programmable. The programmable switches presented two difficulties for manufacturers of these devices;

(i) They were hard to fabricate correctly and (ii) they reduced the speed performance of circuits implemented in PLAs. These drawbacks led to the development of a similar device in which the AND plane is programmable, but the OR plane is fixed. Such a chip is known as a Programmable Array Logic (PAL) device.

Because they are simpler to manufacture, and thus less expensive than PLAs, and offer better performance. PALs have become popular in practical applications.

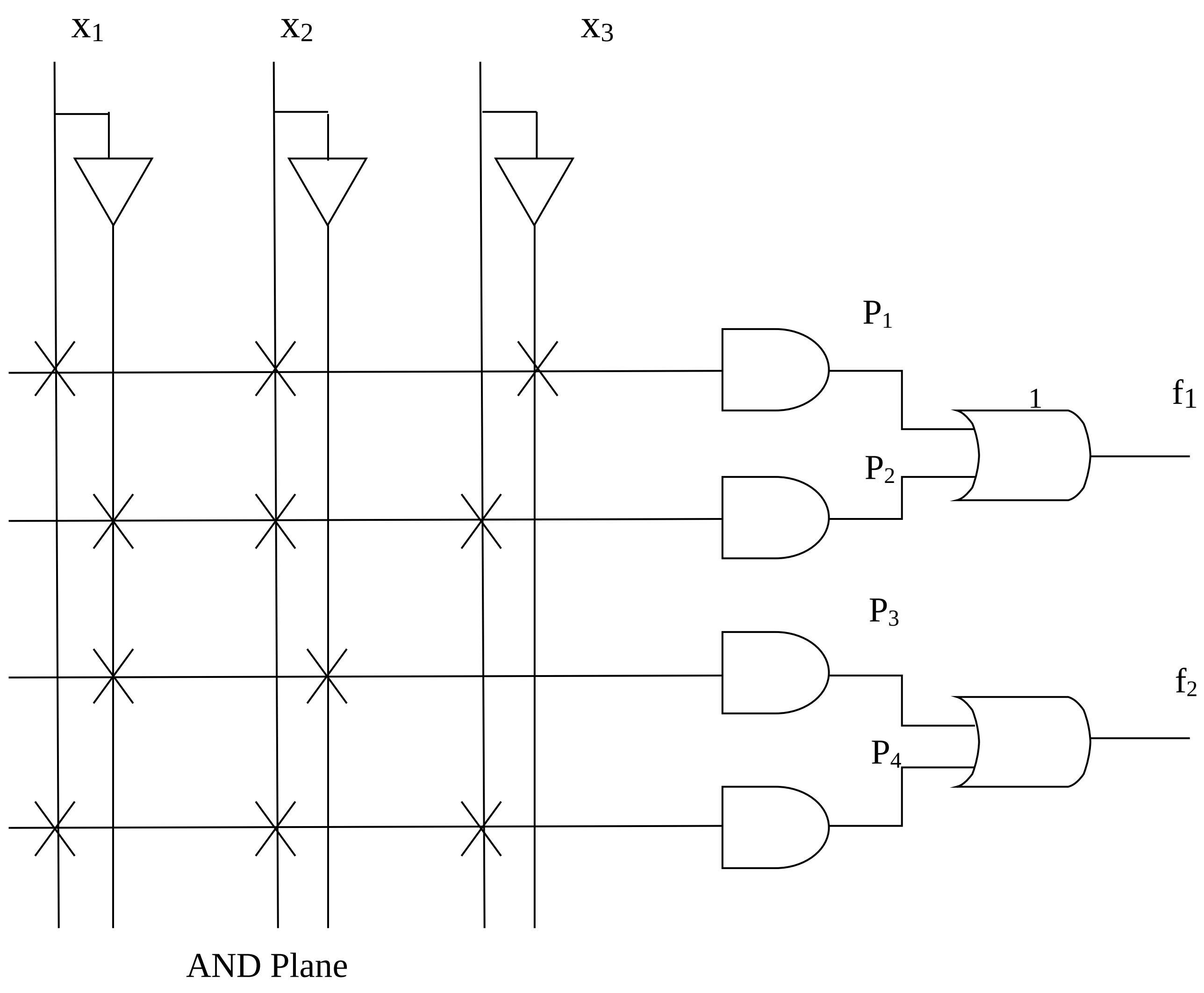


Fig 5.9 An Example of a PAL

An example of a PAL with three inputs, four product terms and two outputs is given in figure 5.9. The product terms P_1 and P_2 are hardwired to one OR gate, and P_3 and P_4 are hardwired to the other OR gate. The PAL is shown programmed to realize the two logic functions

$f_1 = x_1\overline{x_2}x_3 + \overline{x}_1x_2x_3$ and $f_2 = \overline{x_1x_2} + x_1x_2x_3$. In comparison to the PLA in figure 5.3, the PAL offers less flexibility

The PLA allows up to four product terms per OR gate, whereas the OR gates in the PAL have only two inputs. To compensate for the reduced flexibility, PALs are manufactured in a range of sizes, with various numbers of inputs and outputs, and different numbers of inputs to the OR gates.

So far we have assumed that the OR gates in a PAL, as in a PLA, connect directly to the output pins of the chip. In many PALs extra circuitry is added at the output of each OR gate to provide additional flexibility. It is customary to use the term macro cell to refer to the OR gate combined with the extra circuitry.

An example for the flexibility that may be provided in a macro cell is given in Figure 5.10. The symbol labeled flip-flop represents a memory element. It stores the value produced by the OR gate output at a particular point in time and can hold that value as indefinite. The flip-flop is controlled by the signal called clock. When clock makes a transition from logic value 0 to 1, flip-flop stores the value at its D input at that time and this value appears at the flip-flop's Q output. Flip-flops are used for implementing many types of logic circuits.

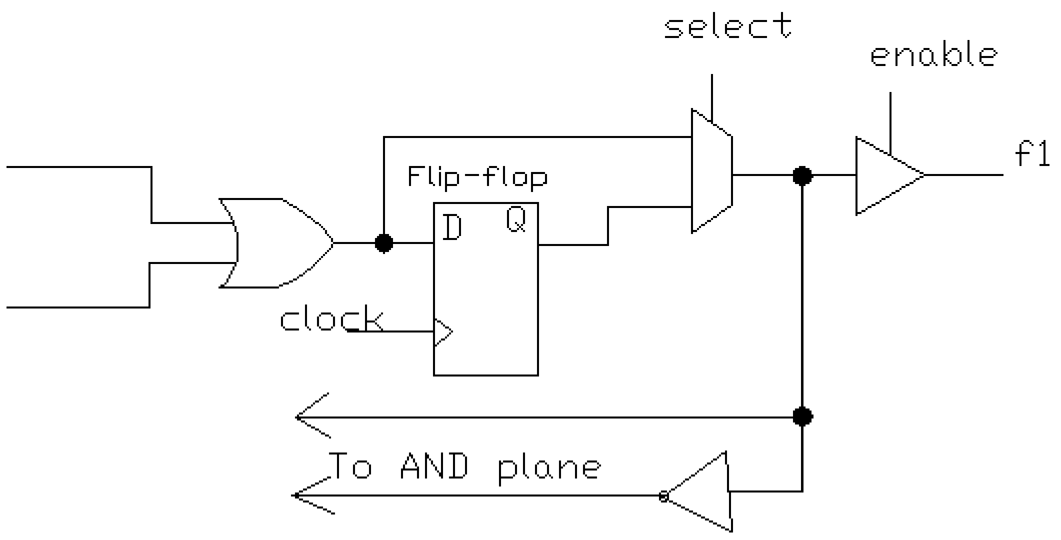


Fig. No: 5.10 2-to-1 multiplexer

In Figure no. 5.10, a 2-to-1 multiplexer selects as an output from the PAL either the OR-gate output or the flip-flop output. The multiplexer's select line can be programmed to be either 0 or 1. Figure no. 5.8 shows another logic gate, called a tri-state buffer, connected between the multiplexer and the PAL output.

Finally, the multiplexer's output is "feed back" to the AND plane in the PAL. This feedback connection allows the logic function produced by the multiplexer to be used internally in the PAL. This allows the implementation of circuits that have multiple stages or levels, of logic gates.

Comparison between PROM, PLA and PAL

Sr. No	PROM	PLA	PAL
1	AND array is fixed and OR array is programmable	Both AND and OR arrays are programmable	OR array is fixed and AND array is programmable
2	Cheaper and simple to use.	Costliest and complex than PAL and PROMs.	Cheaper and simpler
3	All minterms are decoded	AND array can be programmed to get desired minterms	AND array can be programmed to get desired minterms.
4	Only Boolean functions in standard SOP form can be implemented using PROM	Any Boolean functions in SOP form can be implemented using PLA	Any Boolean functions in SOP form can be implemented using PAL.

COMPLEX PROGRAMMABLE LOGIC DEVICES (CPLDs)

PLAs and PALs are useful for implementing a wide variety of small digital circuits. Each device can be used to implement circuits that do not require more than number of inputs, product terms, and outputs that are provided in the particular chip.

These chips are limited to a number of inputs plus outputs of not more than 32. For implementation of circuits that require more inputs and outputs, either multiple PLAs or PALs can be employed or else a more sophisticated type of chip, called a complex programmable logic device (CPLD), can be used.

A CPLD comprises of multiple circuit blocks on a single chip, with internal wiring resources to connect the circuit blocks. Each circuit block is similar to a PLA or a PAL; which refers to the circuit blocks as PAL-like blocks. An example of a CPLD is given in Figure. No 5.11. It includes four PAL-like blocks that are connected to a set of interconnected wires Each PAL-like block is also connected to a sub circuit labeled I/O block, which is attached to a number of the chip's input and output pins.

Figure No.5.12 shows an example of the wiring structure and the connections to PAL-like block in a CPLD. The PAL-like block includes 3 macro cells, each consisting of a four – input OR gate.

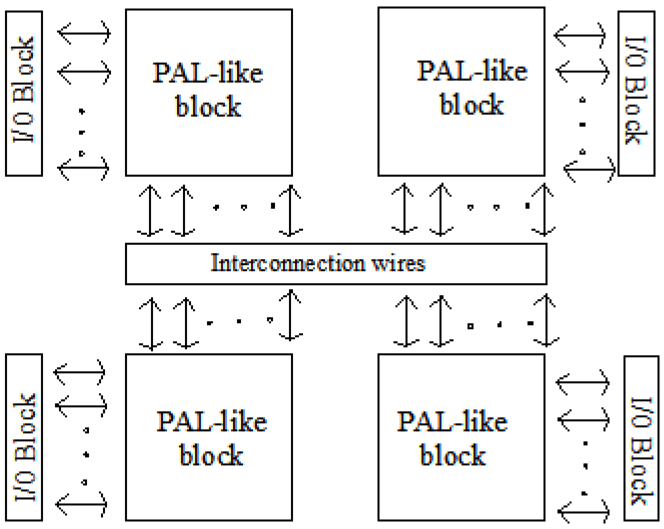


Fig.No:5.11 BLOCK DIAGRAM OF PAL

The OR-gate output is connected to another type of logic gate. It is called an exclusive – OR (XOR) gate. The behavior of an XOR gate is same as for an OR gate except that, if both of the inputs are 1, XOR gate produces a 0. One input to the XOR gate in Figure 5.10 can be programmable connected to 1 or 0: if 1, then the XOR gate complements the OR-gate output, and if 0 then XOR gate has no effect.

The macro cell also includes a flip-flop, a multiplexer, and a tri-state buffer. The flip-flop is used to store the output value produced by the OR gate. Each tri-state buffer is connected to a pin on the CPLD package. The tri-state buffer acts as a switch that allows each pin to be used either as an output from the CPLD or as an input.

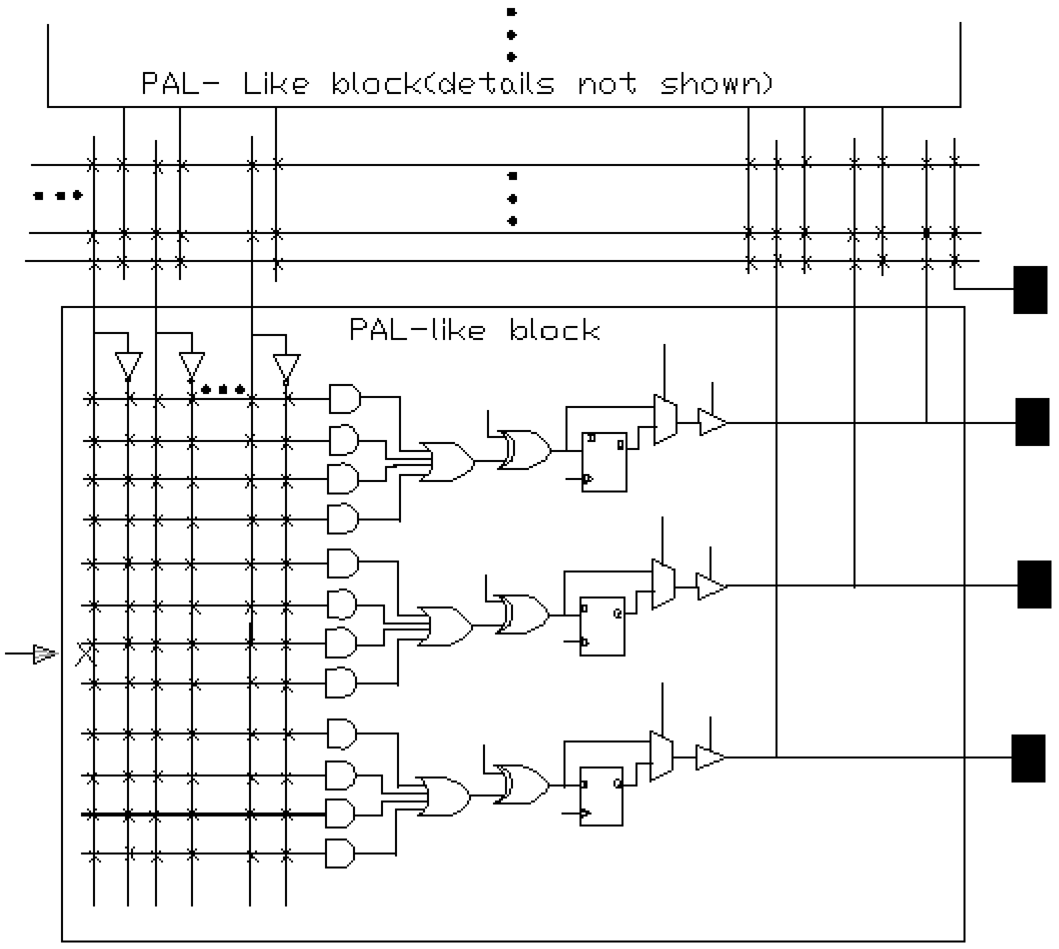


Fig.No:5.12 PAL BLOCK DETAILS

FIELD: PROGRAMMABLE GATE ARRAYS

A filed – programmable gate array (FPGA) is a programmable logic device. It supports implementation of large logic circuits. FPGAs are quite different from SPLDs and CPLDs because FPGAs do not contain AND or OR planes.

Instead, FPGAs provide logic blocks for implementation of the required functions. The general structure of an FPGA is illustrated in Figure no 5.13. It contains three main types of resources: logic blocks, I/O blocks for connecting to the pins of the package, and interconnected wires and switches.

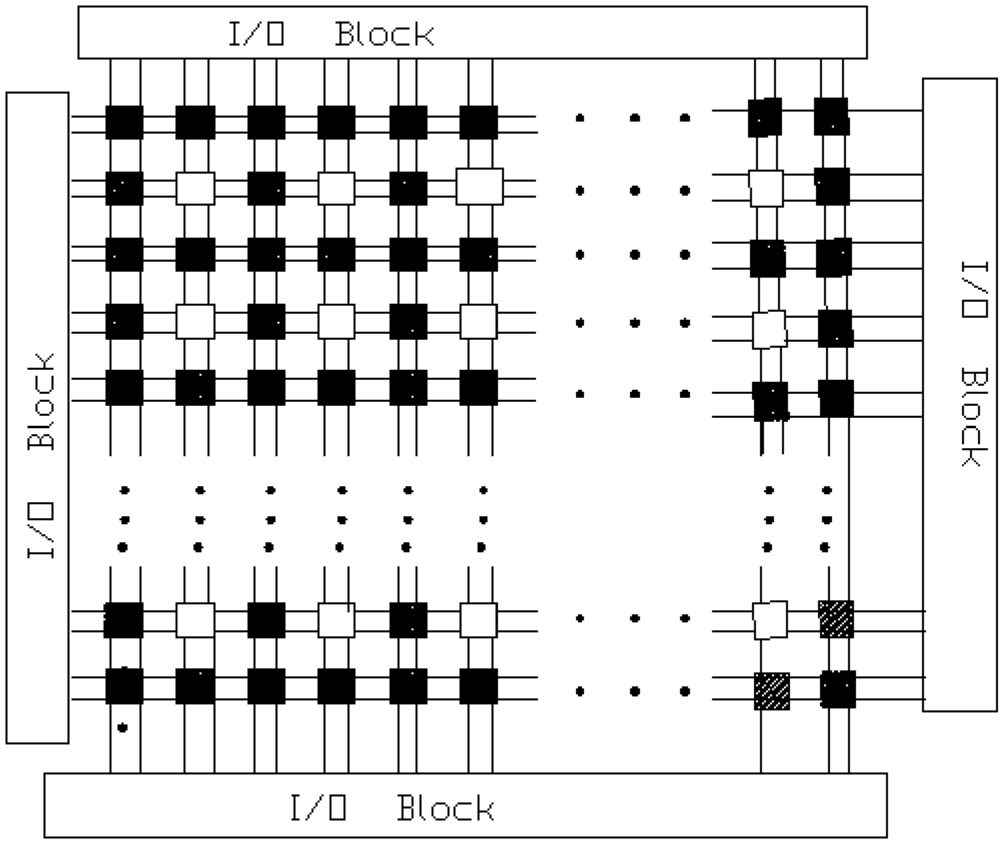


Fig.No:5.13 LOGIC BLOCK DIAGRAM OF FPGA

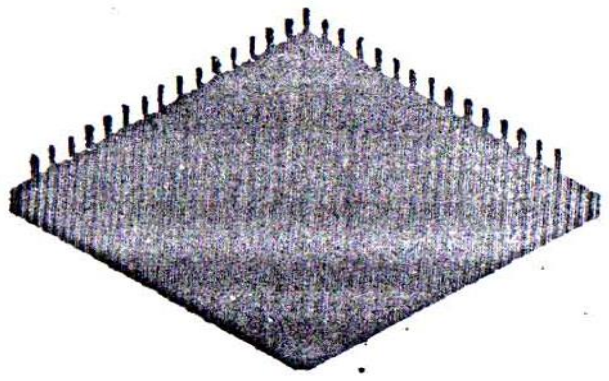


Fig.No:5.14 PIN DIAGRAM OF FPGA

The logic blocks are arranged in a two-dimensional array, and the interconnected wires are organized as horizontal and vertical routing channels between rows and columns of logic blocks. The routing channels consist of wires and programmable switches that allow the logic blocks to be interconnected in many ways.

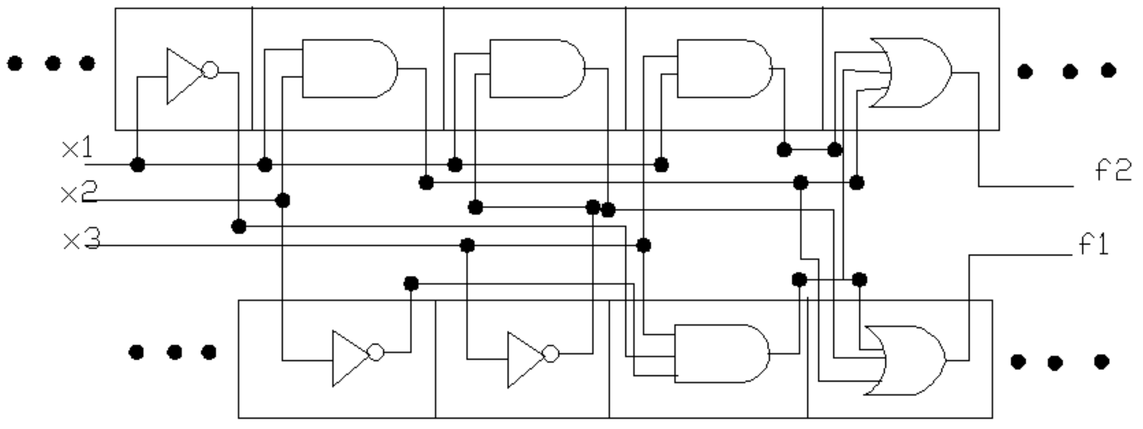
Figure no .5.12 shows two locations for programmable switches; the dark boxes adjacent to logic blocks hold switches that connect the logic block input and output terminals to the interconnected wires, and the dark boxes that are diagonally between logic blocks connect one interconnected wire to another (such as a vertical wire to a horizontal wire). Programmable connections also exist between the I/O blocks and the interconnected wires. The actual number of programmable switches and wires in an FPGA varies in commercial chips.

FPGAs can be used to implement logic circuits of more than a million equivalent gates in size. FPGA chips are available in a variety of packages, including the PLCC and QFP package described earlier. Figure no 5.12 depicts another type of package, called a pin gird array (PGA), APGA package may contain up to a hundreds of pins in total, which extend straight outward from the bottom of the package, in gird pattern. Another packaging technology that has emerged is known as the Ball gird array (BGA). The BGA is similar to the PGA except that the pins are small round balls.

INTRODUCTION TO ASIC

When the chip designer does not need complete flexibility for the layout of each individual transistor in a custom chip, some of the design effort can be avoided by using a technology known as standard cells. Chips are made by using this technology are often called application – specific integrated circuits (ASICs). This technology is illustrated in Figure no. 5.15, which allow a small portion of a chip. The rows of logic gates may be connected by wires that are created in the routing channels between the rows of gates.

In general, many types of logic gates may be used in such type of a chip. The available gates are prebuilt and are stored in a library that can be accessed by the designer. In Figure no 5.13, the wires are drawn in two fashions. This scheme is used because metal wires can be created on integrated circuits in multiple layers, which makes it possible for two wires to cross one another without creating a short circuit.



A section of two rows in a standard-cell chip

Fig.No:5.15 TWO ROW STANDARD CELL CHIP

The thin black wires represent one layer of metal wires, and the thick black wires are a different layer. Each dark square represent a hard-wired connection (called a via) between a wire on one layer and a wire on the other layer. In current technology it is possible to have eight or more layers of metal wiring. Some of the metal layers can be placed on top of the transistors in the logic gates, resulting in a more efficient chip layout.

Like a custom chip, a standard-cell chip is created from scratch according to a user's specifications.

<u>TYPES OF ASIC</u>

1. Full custom ASICs
2. Semi custom ASICs
   - i. Standard cell based ASICs
   - ii. Gate Array based ASICs
     - a. Channeled Gate array
     - b. Channel-less Gate Array
     - c. Embedded Gate array

Full Custom ASICs

In full custom ASICs, an Engineer designs all the logic cells, circuits and layout even interconnects are customized. These ICs are expensive to manufacture and design

Semi Custom ASICs

In semi custom ASICs, some of the logic cells are predesigned and some of the interconnects are customized.

Standard cell based ASICs

It uses predesigned AND gates, OR gates, Multiplexers, Flip Flops known as standard cells. These standard cells are placed in standard cell area. When implementing the design, the standard cells are combined with the fixed blocks that are placed below the standard cell area. So, designer only defines the placement of standard cells.

Gate array based ASICs

In gate array based ASICs, predesigned and pre-characterized logic cells are arranged in a gate array library. So, the designer can choose the gate array to implement the design.

Channeled Gate Array

Rows of logic cells are separated by channels that are used for making interconnection between the rows of logic cells. The space allotted for interconnect is fixed.

Structured Gate Array

Certain areas in the chip are dedicated to implement specific function.

Channel-Less Gate Array

No space is provided for interconnection instead the interconnection is done over the top of gate array devices

Advantages

1. Reducing system cost
2. Low power consumption
3. Improve speed
4. Space saving
5. Full custom Capability

Applications

1. Low noise audio circuit
2. DC-Dc converters
3. Linear regulators
4. Interface circuit for bar code readers
5. Timer Electronics

Implementation of Combinational circuits with PAL & PAL (up to 4 variables)

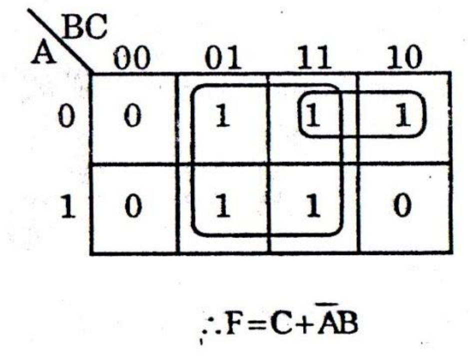
PLA Implementation

Example 1

Implement the function *f* = ∑ {1,2,3,5,7} in PLA

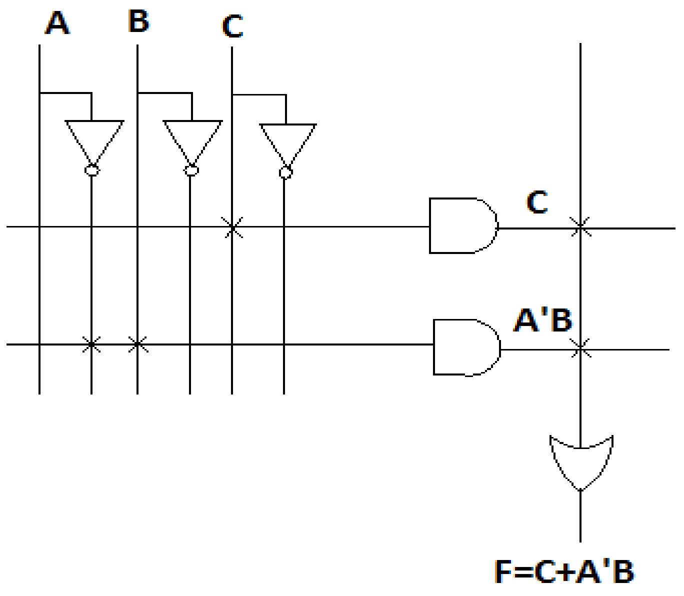
SOLUTION

Step 1



Step 2

PLA Implementation



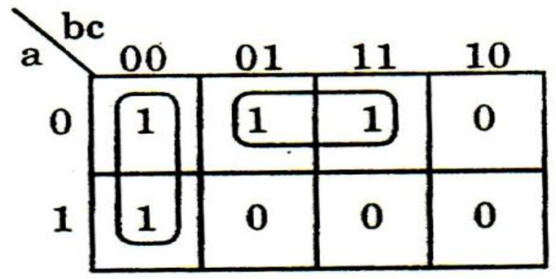
Example 2: Illustrate how a PLA will be used for Combinational Logic for the functions:

f1 (a,b,c) = ∑*m* (0,1,3,4)

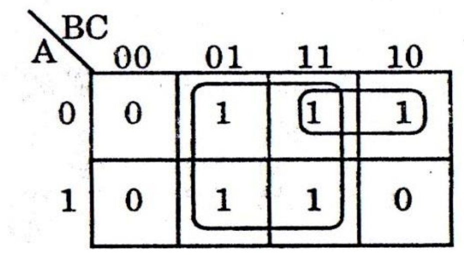
f2 (a,b,c) = ∑*m* (1,2,3,5,7)

SOLUTION

Step 1 K map Simplification



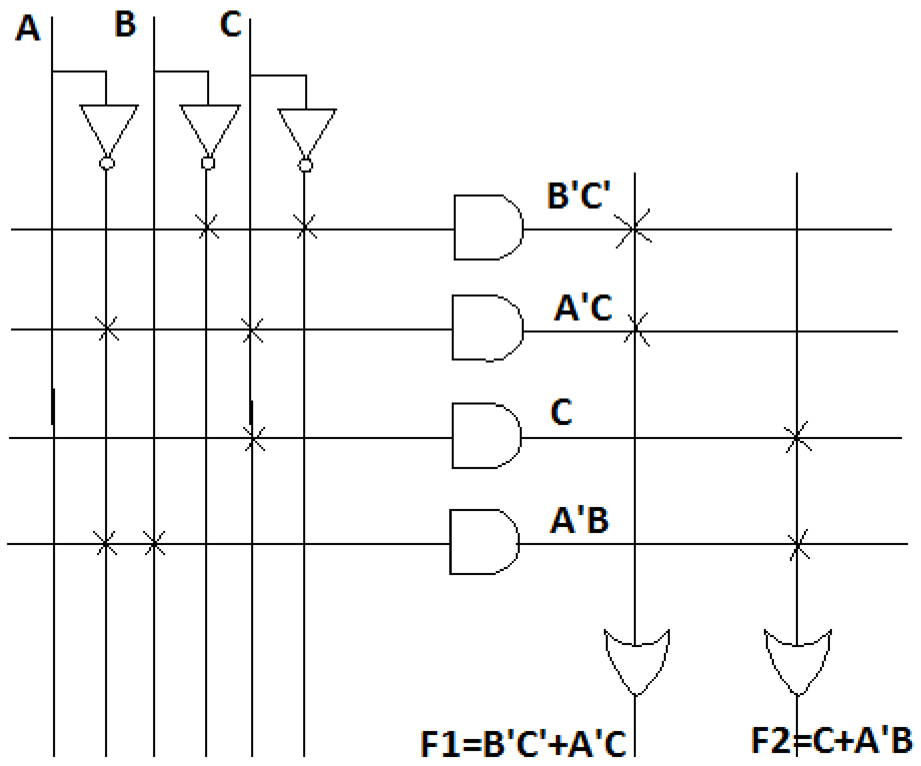
$\therefore f_1 = \bar{b}\,\bar{c} + \bar{a}\,e$



$\therefore F = C + \bar{A}B$

Step 2

PLA Implementation



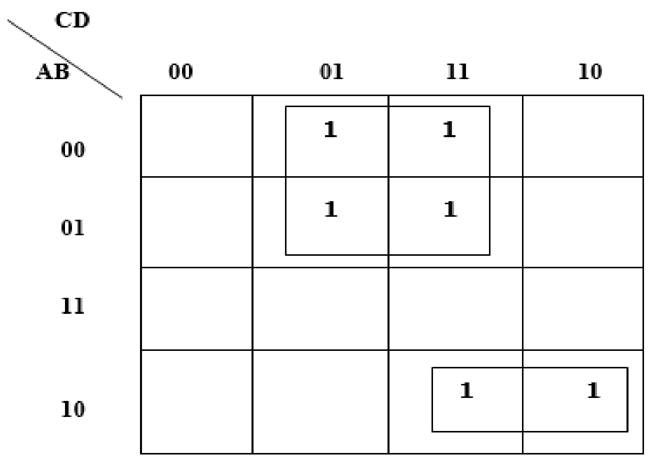
Example: 3

A combinational circuit is defined by the function F = ∑ {1,3,5,7,10,11} Implement the function in PLA

SOLUTION

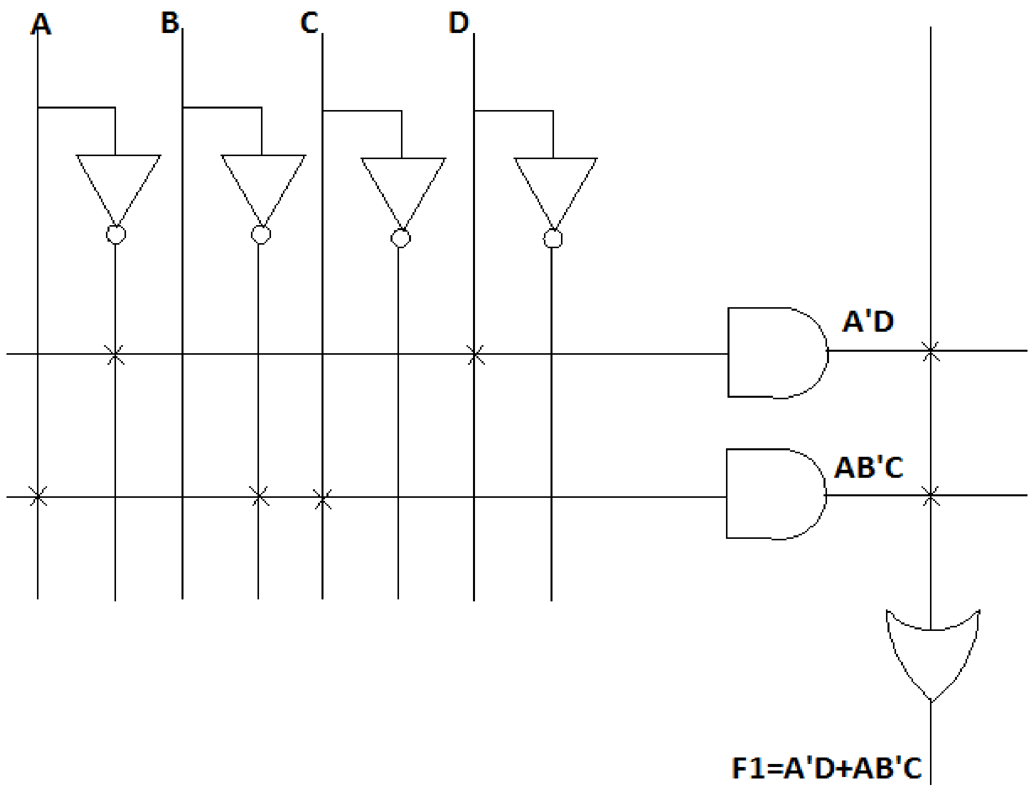
Step 1

K Map Simplification



Step 2

PLA Implementation



PAL IMPLEMENTATION

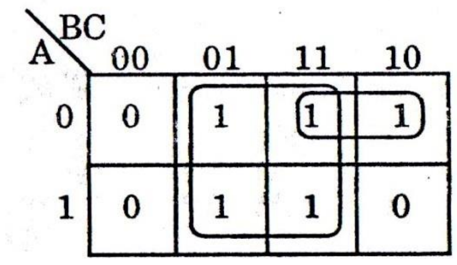
Example: 1

Implement the function F =∑{1,2,3,5,7} in PAL.

SOLUTION

Step: 1

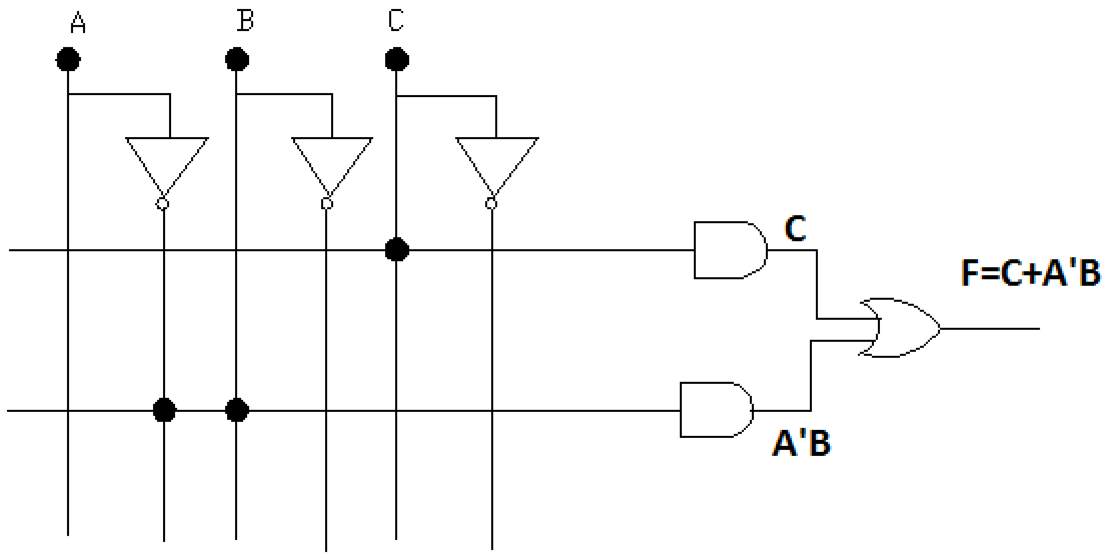
K map Simplification



$\therefore F = C + \bar{A}B$

Step 2

PLA Implementation



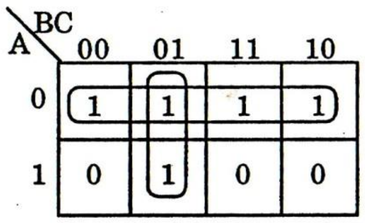
Example: 2

Implement the function F = ∑{0,1,2,3,5} in PAL

SOLUTION

Step 1

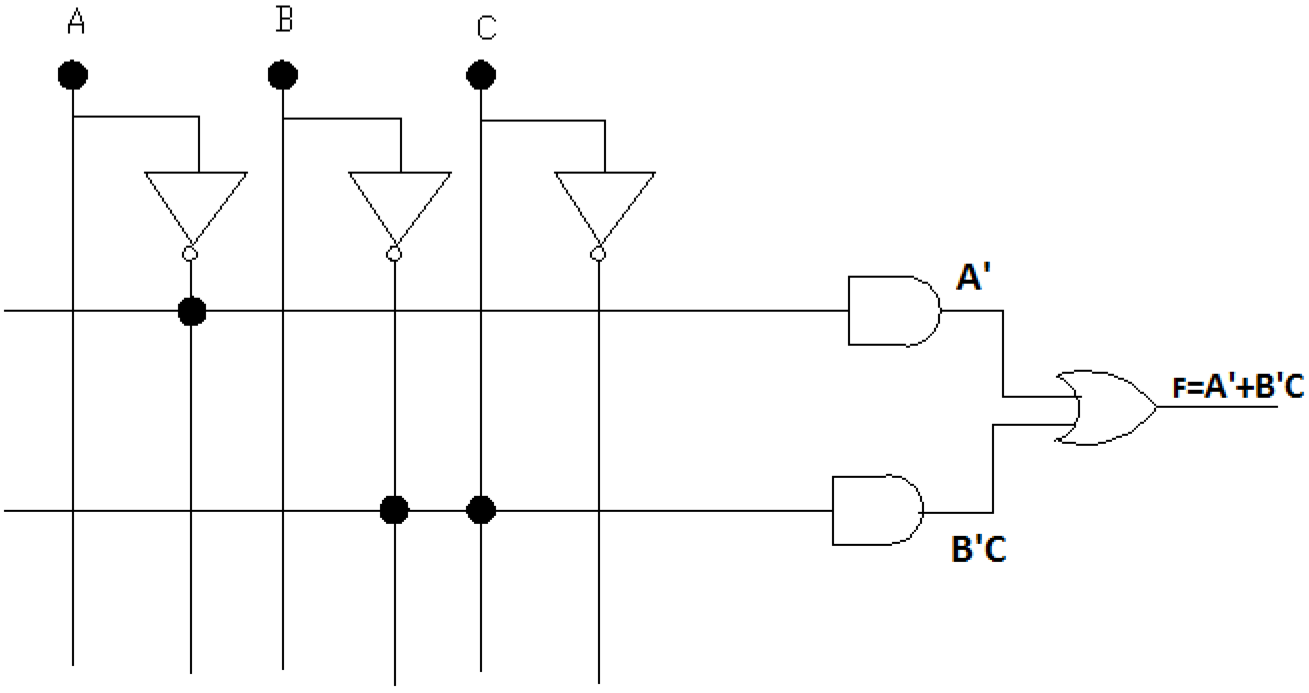
K map Simplification



$$\therefore F = \overline{A} + \overline{B}C$$

Step 2

PAL Implementation



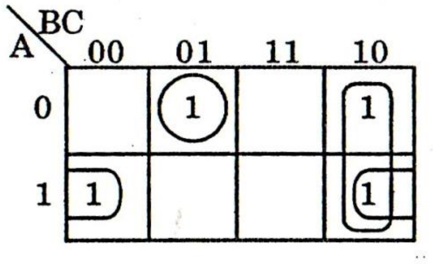
Example: 3

Implement the function in F = ∑ {1,3,4,6} PAL

SOULTION

Step 1

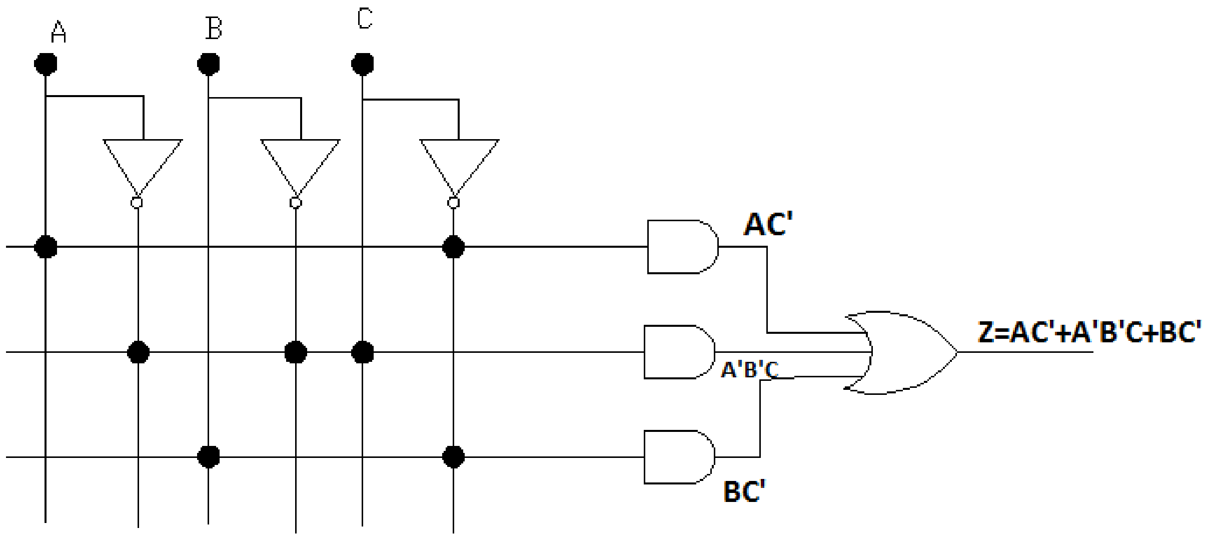
K map Simplification



$$\therefore F = A\overline{C} + \overline{A}\,\overline{B}C + B\overline{C}$$

Step 2

PAL Implementation



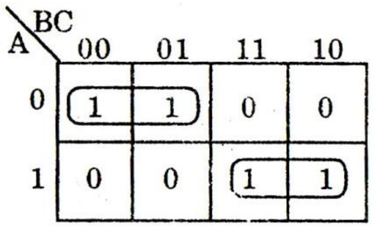
Example: 4

Implement the function F = ∑ m {0,1,6,7} in PAL

SOLUTION

Step: 1

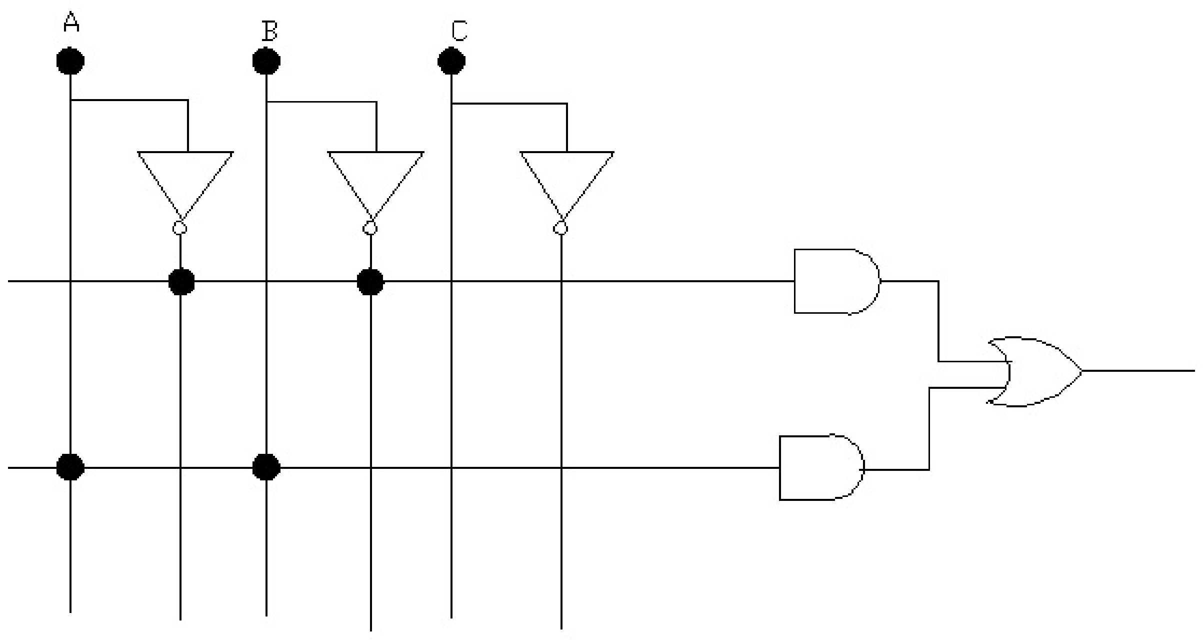
K map Simplification



$$\therefore F = \overline{A}\,\overline{B} + A\,B$$

Step: 2

PAL Implementation



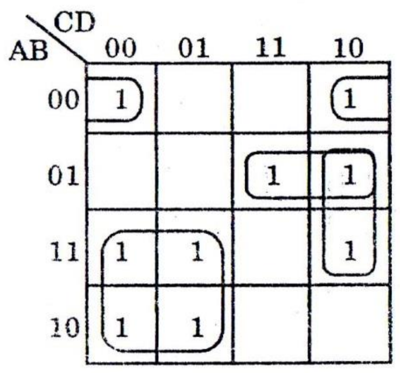
Example: 5

A combinational circuit is defined by the function Implement the function F = ∑ m {0,2,6,7,8,9,12,13,14}in PAL

SOLUTION

Step 1

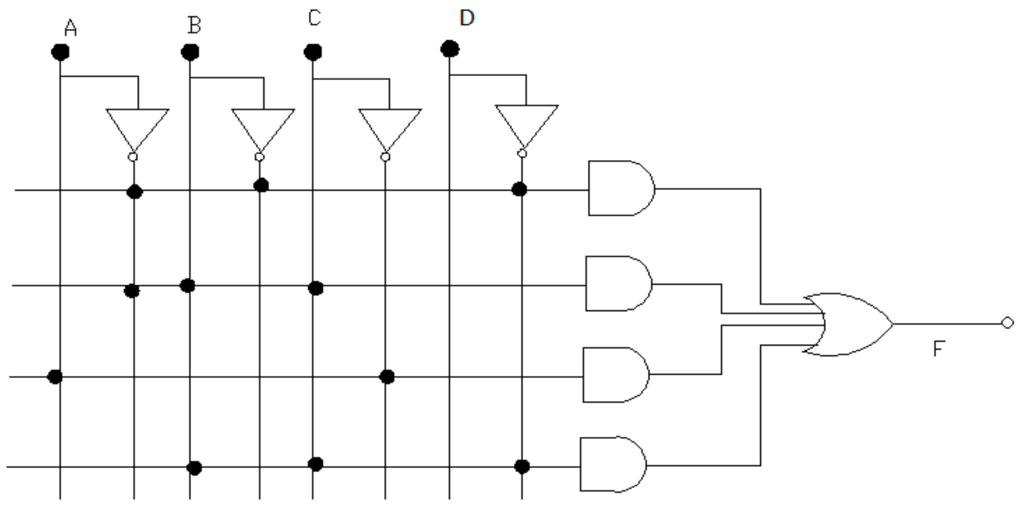
K Map Simplification



$$\therefore F = \overline{A}\,\overline{B}\,\overline{D} + \overline{A}\,B\,C + A\,\overline{C} + B\,C\,\overline{D}$$

Step 2

PAL Implementation



REVIEW QUESTIONS

UNIT-I

Part-A

1. Draw the Transistor level implementation of NAND Gate using CMOS logic
2. Draw the Transistor level implementation of NOR Gate using CMOS logic
3. What are the universal Gates?
4. Why NAND & NOR Gates are said to be universal gates?
5. Distinguish between combinational & sequential circuits
6. List few combinational circuits

Part-B

1. Draw the circuit of Half Adder.
2. Draw the circuit of full adder.
3. What do you meant by Hazards? List the types of Hazards.
4. What are the types of races? Define Critical Race.

Part-C

1. Implement the function with $f=\Sigma 0,2,3,7$ minimal gates
2. Implement the above function $f=\Sigma 0,2,3,7$ with 4:1 mux
3. Implement the function with do not care conditions of 4&6 with minimal gates
4. Implement the function with minimal gates with 4:1mux
5. Implement the function $f=\Sigma\{1,2,3,5,7,10,13\}$ with minimal gates
6. Implement the above function with 4:1mux
7. Implement the function $f=\Sigma\{1,2,3,5,7,10,13\}$
8. Implement the above function with 4:1 mux
9. Draw the circuit of NMOs, NAND, NOR, AND, OR,
10. Draw the circuit of CMOS NAND, NOR, AND, OR

UNIT-II

Part-A

1. Define Synthesis
2. Define Timing Simulation
3. Expand VHDL
4. What are the different levels of abstractions?
5. Define selected signal assignment.

Part-B

1. What are assignment statements?
2. Define for Generate statement?
3. Distinguish between concurrent assignment & sequential assignment & sequential assignment statement.
4. List any Two VHDL operators & Explain
5. Define data flow modeling.
6. Write the VHDL code for OR Gate
7. Write the VHDL code for AND Gate
8. Write the VHDL code for NOT Gate

Part C

1. Explain in detail about different levels of obstructions.
2. Explain in detail about assignment statements.
3. Write a VHDL Code four bit adder.
4. Write a VHDL Code for four bit Comparator
5. Write a VHDL Code for four bit Multiplier
6. Write a VHDL Code for AND, OR , NOR Gates
7. Write a VHDL Code for 4:1 mux
8. Write a VHDL Code for four bit multiplier
9. Write a VHDL Code for Demux
10. Write a VHDL Code for mux.

UNIT-III

Part – A

1. What is the main element in the sequential circuit?
2. Write the excitation Table for T Flip Flop.
3. What do you mean by SISO & PISO?
4. Are Latch & Flip Flop Same?
5. Distinguish between combinational circuit & Sequential circuit.
6. Write the excitation table for D Flip Flop
7. List the various shift Registers present in digital circuit.

Part – B

1 Distinguish between Latch & Flip Flop .What are the types of Flip-Flops

2. Write down the count sequence for Modulo 8 Counter. & Draw the state diagram
3. Write down the count sequence for Modulo 6 Counter. & Draw the state diagram
4. Distinguish between synchronous & Asynchronous Counter.
5. Define state Table.
6. Define state diagram.
7. Write down the excitation Table for JK Flip Flop.

Part – C

1. Design a modulo 8 bit counter using D Flip Flop. Use proper excitation table & State diagram.
2. Design a modulo 6 bit counter using D Flip Flop. Use proper excitation table & State diagram.
3. Design a modulo 5 bit counter using D Flip Flop. Use proper excitation table & State diagram.
4. Design a modulo 4 bit counter using D Flip Flop. Use proper excitation Table & State diagram.
5. Write down the summary of Design Steps
6. Give out the examples for Moore & Mealy Machine
7. Define Mealy & Moore Machines.

UNIT-IV

Part – A

1. What do you meant by storage elements?
2. Write the importance of D FF
3. Write the importance of T FF
4. Write the importance of JK FF

Part – B

1. Write the VHDL code for T FF
2. Write the VHDL code for D FF
3. Write the VHDL code for JK FF

PART C

1. Write a VHDL code for 2 bit up counter

2 Write a VHDL code for 3 bit up/down counter.

3. Write a VHDL code for Decade counter.
4. Write a VHDL code for Johnson Counter.

UNIT-V

Part – A

1. Define PLA
2. Draw the simple circuit of PLA structure
3. Define PAL
4. Expand PLA & PAL

Part – B

1. Draw the General Structure of CPLD
2. Draw the General Structure of FPGA
3. Define ASIC. Write the types of ASICs
4. . Draw the simple circuit of PAL structure
5. Bring out comparison between PROM, PLA & PAL

Part – C

1. (a) Write short notes on PLA

   (b) Implement the following functions in PLA

   f1(a,b,c) = ∑ m{0,1,3,4}

   f2(a,b,c) = ∑{0,2,6,7,8,9,12,13,14}

2. Implement the following function in PAL

   f=∑m{0,2,6,7,8,9,12,13,14}

3. (a) Write short notes on PAL

   (b) Implement the function F = ∑ m(0,1,2,3,5) in PAL

4. (a) Explain about GPLD in detail

   (b) Explain about FPGA in detail

5. Design a combination circuit using a PROM. The circuit accepts 3-bit binary number and generates its equivalent Excess – 3 codes.