DIGITAL SYSTEM DESIGN (15A05402)

LECTURE NOTES

B.TECH

(III-YEAR& I-SEM)

Prepared by:

Dr. A. Pulla Reddy, Associate Professor

Department of Electronics and Communication Engineering

VEMU INSTITUTE OF TECHNOLOGY

(Approved By AICTE, New Delhi and Affiliated to JNTUA, Ananthapuramu)
Accredited By NAAC & ISO: 9001-2015 Certified Institution
Near Pakala, P. Kothakota, Chittoor- Tirupathi Highway
Chittoor, Andhra Pradesh - 517 112
Web Site: www.vemu.org

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY ANANTAPUR

B. Tech III-ISem. (ECE) **L T P C**

3 1 0 3

15A04504 DIGITAL SYSTEM DESIGN

Course Outcomes:

C315_1: Explain the operation CMOS, TTL logic families ECL logic families and interfacing between them.
C315_2: Demonstrate the concept in HDL and Model digital logic circuits using hardware description languages like VHDL
C315_3: Design and implement various Combinational circuits using basic IC structures
C315_4: Design and implement various sequential circuits using VHDL
C315_5: Develop VHDL programs for various complex combinational and Sequential circuits using VHDL

UNIT-I: CMOS LOGIC: Introduction to logic families, CMOS logic, CMOS logic families;BIPOLAR LOGIC AND INTERFACING**:** Bipolar logic, Transistor logic, TTL families, CMOS/TTL interfacing, low voltage CMOS logic and interfacing, Emitter coupled logic, Comparison of logic families, Familiarity with standard 74-series and CMOS 40- series-ICs – Specifications.

UNIT-II: HARDWARE DESCRIPTION LANGUAGES: HDL Based Digital Design, The VHDL Hardware Description Language–Program Structure, Types, Constants and Arrays, Functions and procedures, Libraries and Packages, Structural design elements, Dataflow design elements, Behavioral design elements, The Time Dimension, Simulation, Test Benches, VHDL Features for Sequential Logic Design, Synthesis

UNIT-III: COMBINATIONAL LOGIC DESIGN PRACTICES: Description of basic structures like Decoders, Encoders, Comparators, Multiplexers (74 –series MSI); Design of complex Combinational circuits using the basic structures; Designing Using combinational PLDs like PLAs, PALs ,PROMs CMOS PLDs; Adders & sub tractors, ALUs, Combinational multipliers; VHDL models for the above standard building block ICs.

UNIT-IV : SEQUENTIAL MACHINE DESIGN PRACTICES: Review of design of State machines**;** Standard building block ICs for Shift registers, parallel / serial conversion , shift register counters, Ring counters; Johnson counters, LFSR counter ; VHDL models for
the above standard building block ICs.Synchronous Design example using standard ICs

UNIT –V: Design Examples (using VHDL): Barrel shifter, comparators, floating-point encoder, and dual parity encoder.
Sequential logic Design: Latches & flip flops, PLDs, counters, shift register and their VHDL models.

Text Books:

1. John F.Wakerly ,"Digital Design Principles and Practices" 4th edition, Pearson Education., 2009
2. Charles H.Roth,Jr., "Fundamentals of Logic Design" 5th edition , CENGAGE Learning 2012.

References:

1. M.Morris Mano and Michael D. Cilleti., "Digital Logic Design" 4th edition Pearson Education., 2013
2. Stephen Brown and ZvonkoVranesic, "Fundamentals of digital logic with VHDL design" 2nd edition McGraw Hill Higher Education.
3. J. Bhasker, "A VHDL PRIMER" 3rd edition Eastern Economy Edition, PHI Learning, 2010.

UNIT- I

DIGITAL LOGIC FAMILIES

❖ History of Digital ICs and their Classifications

Integrated circuit: A collection of one or more gates fabricated on a single silicon chip is called an *Integrated Circuit(IC)* or it can be defined as an Integrated Circuit(IC) is a silicon wafer or a Die that contains two or more number of active such as diodes, transistors and some of passive components suchasresistors,capacitors.

NOTE: The passive element Inductor can't be fabricated using ICs, Because they have magnetic flux

History of IC:

- ✓ There are many ways are available to design electronic logic circuit. First electrically controlled logic circuits were developed in 1930s at Bell laboratories based on relays.
- ✓ The first electronic digital computer named as ENIAC was developed in mid 1940s based on vacuum tubes. It has 100 feet long, 10 feet height, 3 feet deep and consumed 140kw of power.
- ✓ By the invention of semiconductor diode and transistor after 1947 smaller, faster and more capable computers were designed.
- ✓ Better computers are designed by the invention of ICs which allowed multiples diodes, transistors and other components to be fabricated on a single chip of silicon in 1960s.
- ✓ The first IC family was introduced in 1960.

Classification of Digital ICs: Based on the size or number of logic components/gates fabricated per chip the ICs are classified into different Integrations as

- ✓ *Small Scale Integration (SSI):* It have less than 100 components (about 10 gates).
- ✓ *Medium Scale Integration (MSI):* It contains between 100-1000 components or have more than 10 but less than 100 gates.
- ✓ *Large Scale Integration (LSI):* Here number of components is between 1000 and 10000 or have gates between 100-1000.
- ✓ *Very Large Scale Integration (VLSI):* It contains components between 10000-100000 per chip or gates between 1000-10000 per chip.
- ✓ *Ultra Large Scale Integration (ULSI):* It contains more than 100000 components per chip.
- ✓ *Giant Scale Integration (GSI):* It contains much more than 2000000 components per chip.

❖ Logic Families and their classifications

Logic Families:

It is a collection of different IC chips that have similar input, output and internal circuit characteristics i.e. group of compatible ICs with same logic levels and supply voltages but perform different logic functions.

NOTE: 1) Chips from same family can be interconnected.

2) Chips from different family may not be compatible, means they may use different power supply voltages and input, output conditions.

Classification of Logic Families:

Logic families are mainly classified as two types as

Bipolar Logic Families: It mainly uses bipolar devices like diodes, transistors in addition to passive elements like resistors and capacitors. These are sub classified as saturated bipolar logic family and unsaturated bipolar logic family.

i) *Saturated Bipolar Logic Family:* In this family the transistors used in ICs are driven into saturation.

Examples: *a) Transistor-Transistor Logic (TTL)*
b) Resistor-Transistor Logic (RTL)
c) Direct Coupled Transistor Logic (DCTL)
d) Diode Transistor Logic (DTL)
e)High Threshold Logic(HTL)
f) Integrated Injection Logic (IIL or I^2L)

ii) *Unsaturated bipolar logic family:* In this family the transistors used in ICs are not driven into saturation.

Examples: *a) Schottky TTL*
b) Emitter Coupled Logic(ECL)

B) Unipolar Logic Families: It mainly uses Unipolar devices like MOSFETs in addition to passive elements like resistors and capacitors. These logic families have the advantages of high speed and lower power consumption than Bipolar families. These are classified as

i) PMOS or P-Channel MOS Logic Family
ii) NMOS or N-Channel MOS Logic Family
iii) CMOS Logic Family

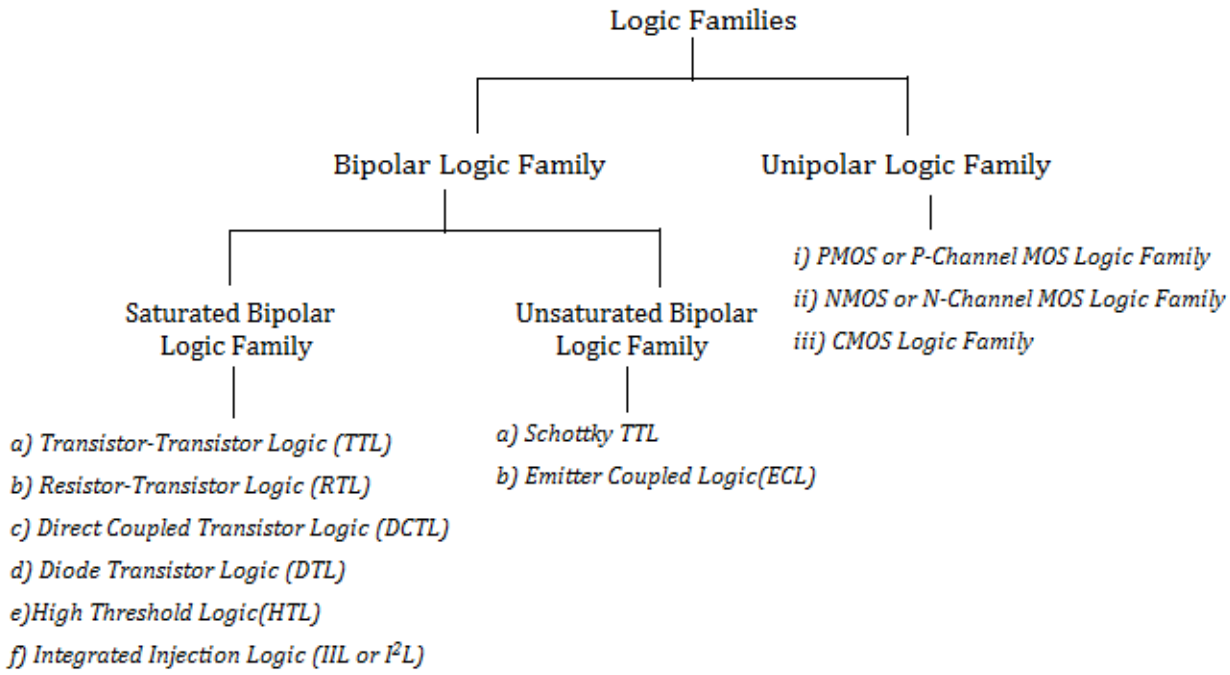


Figure 4.1: Classification of Logic Families

❖ **What is CMOS Logic? Give brief introduction about MOS transistors.**

CMOS Logic: The basic building blocks in CMOS logic circuits are MOS transistors. All circuits that are implemented by CMOS logic have Basic CMOS circuit which will form by the Complementary connection of NMOS and PMOS transistors. So that this logic is named as *Complementary Metal Oxide Semiconductor Logic.*

Metal Oxide Semiconductor transistor:

- ✓ A MOS transistor contains 4 terminals named as Gate(G), Source(S), Drain(D) and Substrate(Sb).
- ✓ Among 4 terminals Gate is an insulating terminal. So no conduction will takes place between remaining two(S and D) terminals. Hence it has highest resistance between Source and Drain.
- ✓ The voltage applied at the Gate terminal may create electric field that enhances or retards the flow of current between Source and Drain. Due to this field effect MOS transistors are called *Field Effect Transistors (FET).*
- ✓ Here the Resistance between Source and Drain also controlled by voltage applied at Gate terminal hence MOSFETs also called as *Voltage Controlled Resistors.*
- ✓ MOS transistors are classified into two types based on the use of the channel type as N-Channel or NMOS transistor and P-Channel or PMOS transistor.
- ✓ Both N-Channel or NMOS transistor and P-Channel or PMOS transistor are again sub divided into two

types based on their mode of operation as Enhancement mode and Deletion mode transistors

- ✓ In Enhancement mode of operation a channel is developed between two terminals Source and Drain of respective MOS transistors.ie. If it is NMOS transistor then N- Channel and it is PMOS then P-Channel transistor by applying required voltage at Gate terminal.
- ✓ In Depletion mode of operation the already existed channel will be removed between two terminals Source and Drain of respective MOS transistors.ie. If it is NMOS transistor then N-Channel and it is PMOS then P-Channel transistor by applying required voltage at Gate terminal.

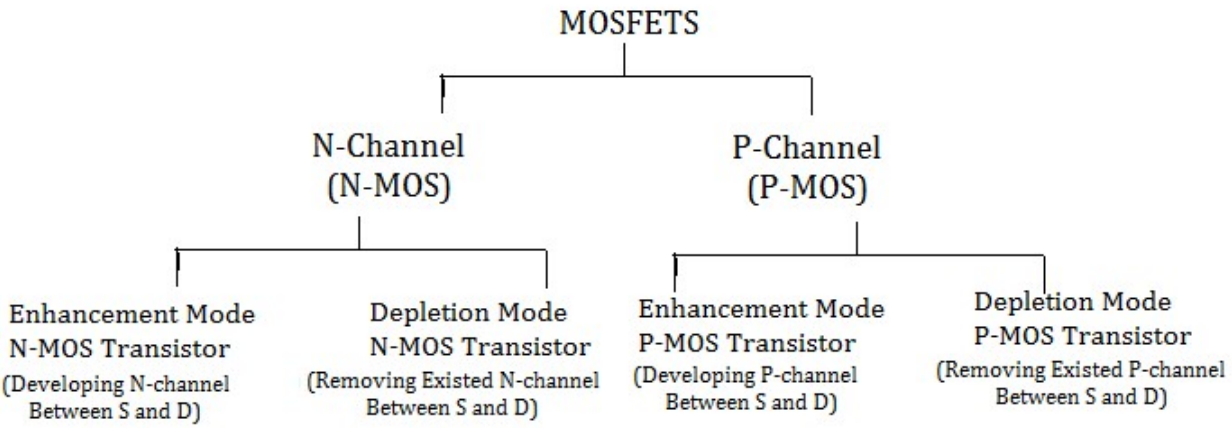


Figure 4.2 Classification of MOSFETs

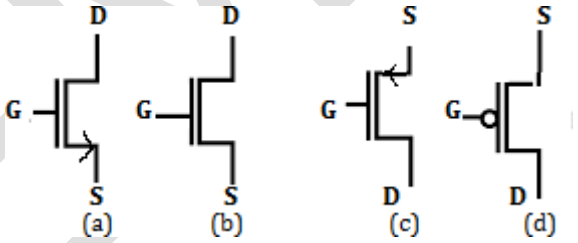


Figure 4.3 Symbols of NMOS(a,b) and PMOS(c,d) Transistors

- ✓ In N-MOS Transistor if input voltage V_{gs} is Zero then resistance between Source and Drain R_{ds} is very high in terms of MΩ and if V_{gs} is more positive voltage then R_{ds} is very low in terms of (0-10) Ω.
- ✓ In P-MOS Transistor if input voltage V_{gs} is Zero then resistance between Source and Drain R_{ds} is very high in terms of MΩ and if V_{gs} is more negative voltage then R_{ds} is very low in terms of (0-10) Ω.

Note: A small current will flows through high resistance between Gate and Source or Drain junctions.

❖ **Explain the Operation of basic CMOS circuit?**

Basic CMOS logic circuits:

- ✓ Basic NMOS and PMOS transistors that connected in a complementary connection to form CMOS logic circuit.
- ✓ It contains pull-up and pull-down networks.
- ✓ Pull-up network contains PMOS transistor and pull-down network consists of NMOS Transistor.
- ✓ When input applied as logic '0'(L) the PMOS transistor is in ON condition and that translates output to logic '1'(H), i.e., applied voltage is pulled up to 5V(H) from 0V(L) by PMOS transistor. Hence it is called as *Pull-Up transistor*.
- ✓ When input applied as logic '1'(H) the NMOS transistor is in ON condition and that translates output to logic '0'(L), i.e., applied voltage is pulled down to 0V(L) from 5V(H) by NMOS transistor. Hence it is called as *Pull-Down transistor*.

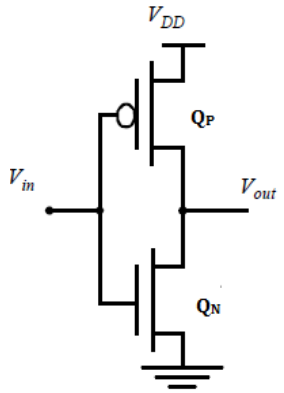


Figure 4.4: Basic CMOS Inverter circuit

V_{in}	PMOS Transistor	NMOS Transistor	V_{OUT}
0 V(L)	ON	OFF	5 V(H)
5 V(H)	OFF	ON	0 V(L)

Table 4.1 Function of CMOS circuit

Operation:

Case (i): When V_{in} = 0 V, then NMOS transistor is OFF state, since its V_{GS} = 0 V. PMOS is in ON state Since its V_{GS} is large negative (-5V). So PMOS presents only a small resistance between V_{DD} and output. Hence output is 5 V.

Case (ii): When V_{in}=5V, then PMOS transistor is in OFF state, since its V_{GS}=0V. NMOS is in ON state Since its V_{GS} is large positive (+5 V). So NMOS presents only a small resistance between output and ground. Hence output is 0 V.

Note: From case (i) & (ii) we can conclude that the operation of a basic CMOS circuit gives INVERTING operation.

❖ **2 Input NAND gate using CMOS logic**

Implementation of 2 Input NAND gate using CMOS logic:

- ✓ NAND gate is one of the basic logic gates to perform the digital operation on the input signals.
- ✓ It is the combination of AND Gate followed by NOT gate i.e. it is the opposite operation of AND gate where the Logic NAND gate is complementary of AND gate.
- ✓ The logic output of NAND gate is low (FALSE) only when the inputs are high (TRUE).
- ✓ To implementation 2 Input NAND gate using CMOS logic we require 2 pull-up PMOS and 2 pull-down NMOS transistors.

Operation:

Case (i): When A=B=0 V, then both NMOS transistors are in OFF state, since its $V_{GSA} = V_{GSB} = 0$ V. Both PMOS transistors (P_A, P_B) are in ON state. Since its V_{GSA} and V_{GSB} voltage is large negative (-5V). So PMOS transistors presents only a small resistance between V_{DD} and output. Hence output is 5 V.

Case (ii): When A=B=5 V, then both PMOS transistors are in OFF state, since its input voltages $V_{GSA} = V_{GSB} = 0$ V. Both NMOS transistors (N_A, N_B) are in ON state Since its V_{GSA} and V_{GSB} is large positive (+5V). So NMOS transistors presents only a small resistance between Output and ground. Hence output is 0 V.

Case (iii): When A=0 V and B=5 V, then NMOS transistor (N_A) is in OFF state, since its input voltage V_{GSA}= 0 V and NMOS transistor (N_B) is in ON state since its V_{GSB}=5V. PMOS transistor (P_A) is in ON state, since its V_{GSA}=-5V and PMOS transistor (P_B) is in OFF state since its V_{GSB}=0V. So PMOS transistor P_A presents only a small resistance between V_{DD} and output. Hence output is 5 V.

Case (iv): When A=5 V and B=0 V, then NMOS transistor (N_A) is in ON state, since its input voltage V_{GSA}= +5V and NMOS transistor (N_B) is in OFF state since its V_{GSB}=0V. PMOS transistor (P_A) is in OFF state, since its V_{GSA}=0V and PMOS transistor (P_B) is in ON state since its V_{GSB}=-5V. So PMOS transistor P_B presents only a small resistance between V_{DD} and output. Hence output is 5 V.

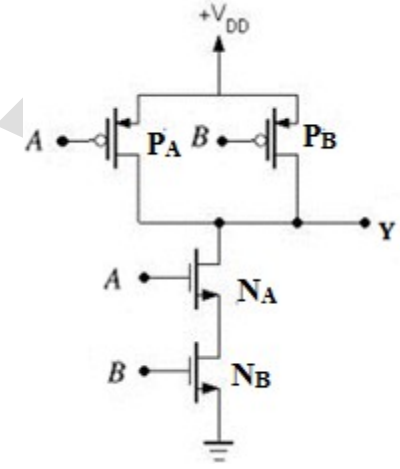


Figure 4.5: Two Input NAND gate using CMOS logic

Inputs		Transistors				Output
A	B	N_A	N_B	P_A	P_B	Y
0V(L)	0V(L)	OFF	OFF	ON	ON	5V(H)
0V(L)	5V(H)	OFF	ON	ON	OFF	5V(H)
5V(H)	0V(L)	ON	OFF	OFF	ON	5V(H)
5V(H)	5V(H)	ON	ON	OFF	OFF	0V(L)

Table 4.2 Function of CMOS 2 Input NAND circuit

❖ **Implementation of 2 Input NOR gate using CMOS logic:**

- ✓ NOR gate is one of the basic logic gates to perform the digital operation on the input signals.
- ✓ It is the combination of OR Gate followed by NOT gate i.e. it is the opposite operation of OR gate where the Logic NOR gate is complementary of OR gate.

- ✓ The logic output of NOR gate is HIGH (True) only when the inputs are LOW (False).
- ✓ To implementation 2 Input NOR gate using CMOS logic we require 2 pull-up PMOS and 2 pull-down NMOS transistors.

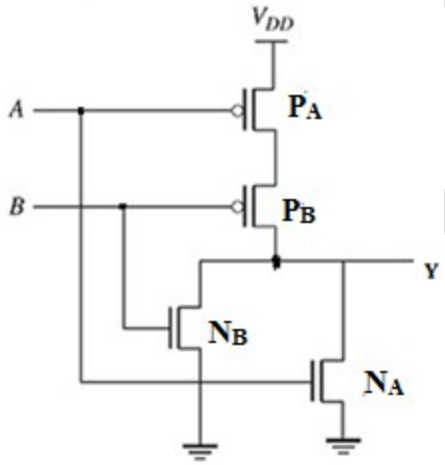
Operation:

Case (i): When A=B=0 V, then both NMOS transistors are in OFF state, since its $V_{GSA} = V_{GSB} = 0$ V. Both PMOS transistors (P_A, P_B) are in ON state. Since its V_{GSA} and V_{GSB} voltage is large negative (-5V). So PMOS transistors presents only a small resistance between V_{DD} and output. Hence output is 5 V.

Case (ii): When A=B=5 V, then both PMOS transistors are in OFF state, since its input voltages $V_{GSA} = V_{GSB} = 0$ V. Both NMOS transistors (N_A, N_B) are in ON state Since its V_{GSA} and V_{GSB} is large positive (+5V). So NMOS transistors presents only a small resistance between Output and ground. Hence output is 0 V.

Case (iii): When A=0 V and B=5 V, then NMOS transistor (N_A) is in OFF state, since its input voltage V_{GSA}= 0 V and NMOS transistor (N_B) is in ON state since its V_{GSB}=5V. PMOS transistor (P_A) is in ON state, since its V_{GSA}= -5V and PMOS transistor (P_B) is in OFF state since its V_{GSB}=0V. So no supply has the connection with output. Hence output is 0V.

Case (iv): When A=5 V and B=0 V, then NMOS transistor (N_A) is in ON state, since its input voltage V_{GSA}= +5V and NMOS transistor (N_B) is in OFF state since its V_{GSB}=0V. PMOS transistor (P_A) is in OFF state, since its V_{GSA}= 0V and PMOS transistor (P_B) is in ON state since its V_{GSB}=-5V. So no supply has the connection with output. Hence output is 0V.



Inputs		Transistors				Output
A	B	N_A	N_B	P_A	P_B	Y
0V(L)	0V(L)	OFF	OFF	ON	ON	5V(H)
0V(L)	5V(H)	OFF	ON	ON	OFF	0V(L)
5V(H)	0V(L)	ON	OFF	OFF	ON	0V(L)
5V(H)	5V(H)	ON	ON	OFF	OFF	0V(L)

Table 4.3 Function of CMOS 2 Input NOR

Figure 4.6: Two Input NOR gate circuit using CMOS logic

NOTE: *For a given Si Area N-Channel Transistor has lower ON resistance than P-Channel Transistor. So, the circuit having series N-Channel connection is more faster than Parallel P-Channel, Hence NAND circuit is faster than NOR.*

6. Explain the study of Steady state Electrical behavior of CMOS Circuits?

Ans) Steady state electrical behavior of CMOS Circuits: It is the behavior of the circuit when inputs and outputs are changing. Under this study we will observe

a. logic voltage levels and DC noise margin.
b. Fan-out.
c. Unused inputs.
d. Circuit behavior with respect to load.
e. Loading effects.
f. Electro Static Discharge(ESD).

a) logic voltage levels logic voltage levels and DC noise margin :

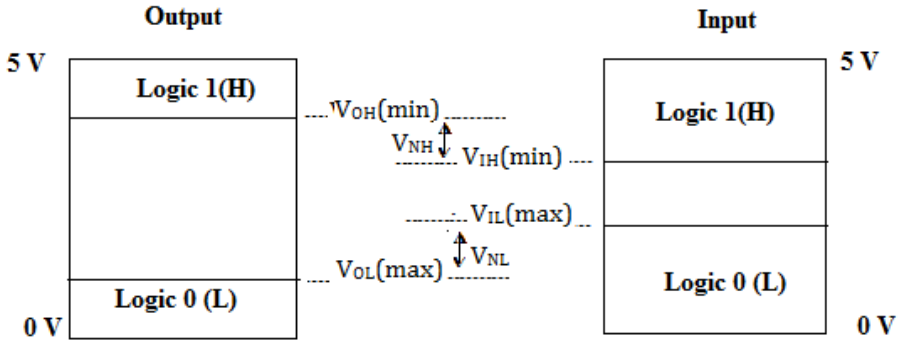


Figure 4.7: Logic voltage levels and DC noise margin

- ✓ **High Level Input Voltage (V_{IH}(min)):** It is the minimum voltage required for a logic '1' at an input. Any voltage below this level will not be accepted as HIGH by logic Circuit.
- ✓ **High Level Output Voltage (V_{OH}(min)):** It is the minimum voltage required for a logic '1' at an Output under defined load conditions.
- ✓ **Low Level Input Voltage (V_{IL}(max)):** It is the maximum voltage required for a logic '0' at an input. Any voltage above this level will not be accepted as LOW by logic Circuit.
- ✓ **Low Level Output Voltage (V_{OL}(max)):** It is the maximum voltage required for a logic '0' at an Output under defined load conditions.

Noise Margin:

- ✓ In practice sometimes unwanted signals called noise may occur by droping input voltage less than V_{IH}(min) and rise above the V_{IL}(max).
- ✓ To avoid this we kept V_{OH}(min), V_{OL}(max) to certain fraction of voltages below and above respectively. These limits are called DC Noise margins.
- ✓ DC noise margin allows digital circuit to function properly.
- ✓ The noise margin in HIGH state is $V_{NH} = V_{OH}(min) - V_{IH}(min)$
- ✓ The noise margin in LOW state is $V_{NL} = V_{IL}(max) - V_{OL}(max)$ Similarly

current parameters can also specified as

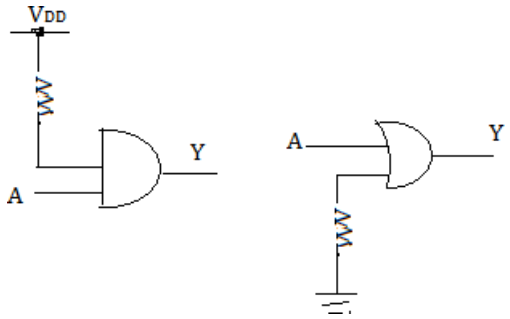
- ✓ **High Level Input Current (I_{IH}):** It is the current that flows into input when a specified high level voltage is applied to input.
- ✓ **High Level Output Current (I_{OH}):** It is the current that flows from an Output under specified load conditions.
- ✓ **Low Level Input Current (I_{IL}):** It is the current that flows into input when a specified low evel voltage is applied to input.
- ✓ **Low Level Output Current (I_{OL}):** It is the current that flows from an Output under specified load conditions.

b) Fan-Out:

- ✓ The fan-out in logic gate is number of inputs that the gate can drive without exceeding its worst case loading specifications.
- ✓ Fan-Out depends not only on the output characteristics but also on inputs that it is driving
- ✓ The LOW state and HIGH state fan-outs of a gate may not be equal.

c) Un-Used inputs:

- ✓ CMOS inputs should never be left unconnected. That means all the CMOS inputs have to be tied either to a fixed voltage level (0V or V_{DD}) or to another input.
- ✓ An unused CMOS input is sensitive to Noise and static charges that could easily bias both th P and N channel MOSFETs in conductive state, Which may result increased power dissipation and possible over heating.
- ✓ So An unused inputs of AND/NAND input should be tied to logic '1', and unused inputs of OR/NOR input should be tied logic '0' or ground.



(a) (b)

Figure 4.8: Unused inputs connected to V_{DD} and ground for a) AND gate b) Or gate

d) Circuit behavior with respect to load:

- ✓ The input impedance of CMOS circuit is very high so it consumes less current from the circuit that drive them.
- ✓ If other devices such as TTL Gate, LED, Discrete resistors which are requires large amount of current to operate are also connected to CMOS output.
- ✓ They are called as Resistive load or DC load. Let a CMOS circuit connected with a resistive load, then output is not ideal.

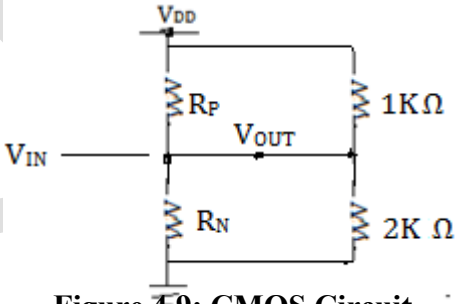


Figure 4.9: CMOS Circuit with resistive load.

- ✓ In such conditions output voltage may be greater than 0.1 V in LOW state and less than 4.4 V in HIGH state.
- ✓ In any one terminal circuit having only resistor and voltage source may be modeled as Thevinen's equivalent circuit.

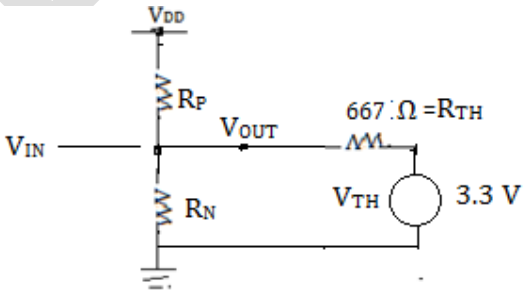


Figure 4.9: Equivalent Thevinen's Circuit

Thevinen's voltage is open circuit voltage of original, V_{TH}= $\frac{2K\Omega X5}{2K\Omega+1K\Omega}$= 3.33 V

Thevinen's Resistance obtained by Thevinen's voltage divided by short circuit current,

$R_{TH}=\frac{V_{TH}}{I(short)} = \frac{V_{TH}}{\frac{VDD}{1K\Omega}} = \frac{3.33\ V}{5\ mA} \cong 667\Omega$

Case-1: Let input V_{in}= 5 V, then NMOS transistor is in ON condition, so R_N value very low (≅ 100Ω). And PMOS transistor is in OFF condition, So R_P is more (≅ 1MΩ). Then

$V_{OUT} = \frac{V_{TH} X R_N}{R_N+R_{TH}} = \frac{3.3\ V\ X\ 100\Omega}{100\Omega+667\Omega} = 0.43\ V.$

Case-2: Let input V_{in}=0V, then NMOS transistor is in OFF condition, so R_N value very High (≅ 1MΩ). And PMOS transistor is in ON condition, So R_P is very low (≅ 200Ω). (Since always R_P is more R_N) Then $V_{OUT} = \frac{(V_{DD}-V_{TH}) X R_{TH}}{R_P+R_{TH}} + V_{TH}$

$= \frac{(5\ V-3.3\ V)\ X\ 667\Omega}{200\Omega+667\Omega} + 3.3\ V = 4.61\ V$

Conclusion: In Practical application, the output voltage is greater than 0.1 V (i.e 0.43 V) in LOW state and less than 4.4 V (i.e 4.43 V) in HIGH state has obtained due resistive load connections. So at tha time of designing CMOS circuit we must check High and Low state output voltages due to resistive loads to confirm they are in specified limit of $V_{OL(Max)}$ and $V_{OH(Min)}$.

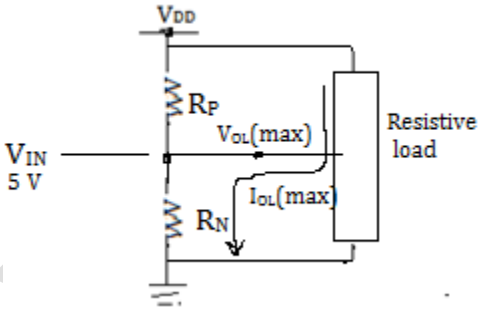


Figure 4.10(a) Sinking Current

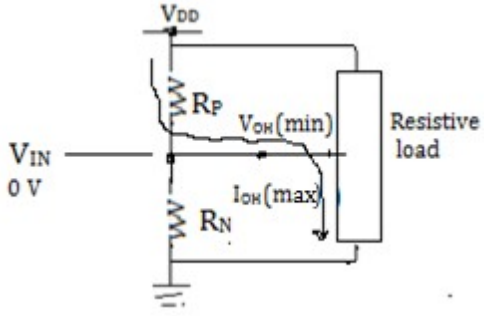


Figure 4.10(b) Sourcing Current

Sinking Current($I_{OL(max)}$): The maximum current that the output can sink in the low state while still maintaining an output voltage smaller than $V_{OL(Max)}$.

Sourcing Current($I_{OH(max)}$): The maximum current that the output can source/ supply in the high state while still maintaining an output voltage greater than $V_{OH(Min)}$.

e) Effects of loading: The basic loading effects on output are

- ✓ In the low state, the output voltage(V_{OL}) may increase beyond $V_{OL(max)}$.
- ✓ In the high state, the output voltage(V_{OH}) may fall beyond $V_{OH(min)}$.
- ✓ Propagation delay may increase beyond specifications.
- ✓ Output rise and fall times may increase beyond specifications.
- ✓ The Operating temperature may increase there by reducing reliability, It may also Cause device failure.

f) Electro Static Discharge(ESD):

- ✓ CMOS device can be easily destroyed by ESD.
- ✓ The primary source to charge is static electricity produced by rough handling and motion of various kinds of plastics and textiles.
- ✓ This Electro Static Discharge can be eliminated by just smooth handling the device.

❖ **Explain the study of Dynamic Electrical behavior of CMOS Circuits?**

Dynamic Electrical behavior of CMOS Circuits:

- ✓ It is the study of electrical behavior of CMOS circuits that the change occurring in output by the change in input.
- ✓ The speed and power consumption of a CMOS circuit mostly depends on DC or Dynamic characteristics of device and its load.
- ✓ Under this study we will observe Transition time, Propagation delay and Power consumption.

Transition Time:

- The amount of time that the output of a logic circuit takes to change from one state to another state is called *Transition Time.*
- In practice output can't change its state instantaneously, it takes some time to charge the stray capacitance and other components that they drive.
- Stray capacitance also called capacitive load or DC load.
- The transition time is measured in two ways by state change as RISE TIME and FALL TIME. They are depends on two factors i) On transistor resistance, ii) Load capacitance.

RISE TIME: It is the time taken by the output to change from LOW State to HIGH State.

FALL TIME: It is the time taken by the output to change from HIGH State to LOW State.

➢ **Analysis of RISE Time:** Let a CMOS circuit represented in-terms of Load capacitance and Transistor ON resistance while output changes from LOW State to HIGH State.

Figure 4.11: Output changes from LOW state to HIGH state.

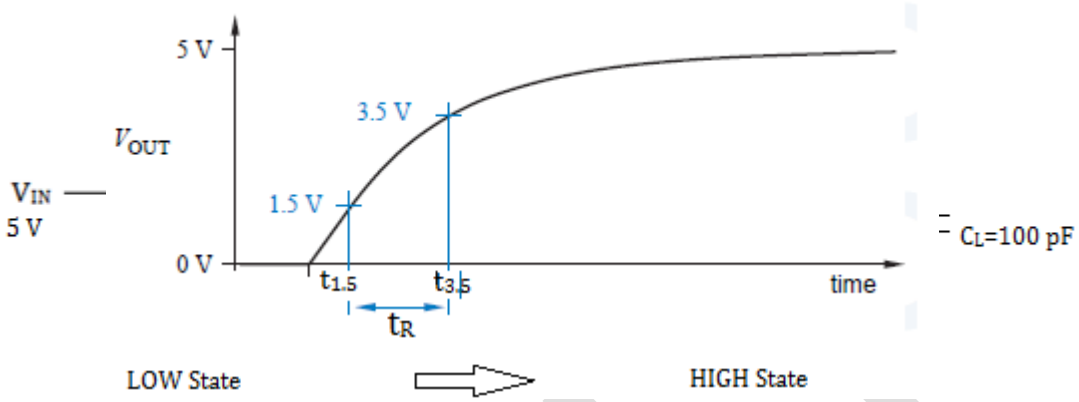


Figure 4.12: Output changes from LOW state to HIGH state.

- When input is 5V then R_N value is less because NMOS transistor is in ON State assume that as 100 Ω and R_P value is more because PMOS transistor is in OFF state assume that as 1M Ω.
- When input is 0V then R_N value is more because NMOS transistor is in OFF State assume that as 1M Ω and R_P value is less because PMOS transistor is in ON state assume that as 200 Ω.
- At time t=0 sec then $V_{OUT} \cong 0$ V and at time t=∞ sec then $V_{OUT} \cong 5$ V. Now we have to measure the time at lower limit of HIGH state($t_{3.5}$) and Upper limit of LOW state($t_{1.5}$)
- The output voltage is given by $V_{OUT}=V_{DD}-V_{DD}.e^{-t/Rp.CL}$ ----------(1)

  From Equation-(1) we may get $t=R_P.C_L.\ln\left(\frac{VDD-VOUT}{VDD}\right)$ ----------(2)

  By substituting parameters Values V_{DD}=5V, R_P=200Ω, C_L=100pF and V_{OUT}=3.5V

  then time $t_{3.5}= -20 \times 10^{-9}.\ln\left(\frac{5V-3.5V}{5V}\right)$=24.08nsec ----------(3)

  By substituting parameters Values V_{DD}=5V, R_P=200Ω, C_L=100pF and V_{OUT}=1.5V

  then time $t_{1.5}= -20 \times 10^{-9}.\ln\left(\frac{5V-1.5V}{5V}\right)$=7.13nsec ----------(4)

From Equations (3) & (4) RISE Time (t_R) = $t_{3.5} - t_{1.5}$ = 24.08nsec – 7.13nsec = 16.95nsec

∴ RISE Time (t_R) = 16.95 nsec.

- **Analysis of FALL Time:** Let a CMOS circuit represented in-terms of Load capacitance and Transistor ON resistance while output changes from LOW State to HIGH State.

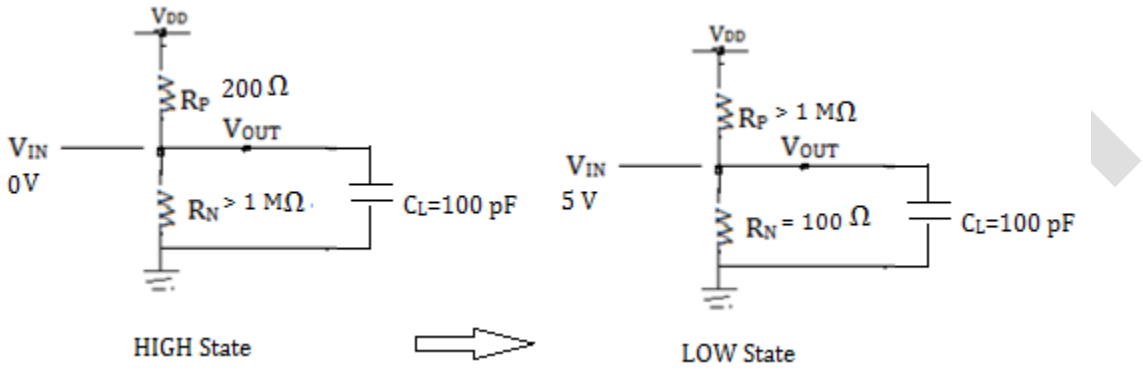


Figure 4.13: Output changes from HIGH state to LOW state

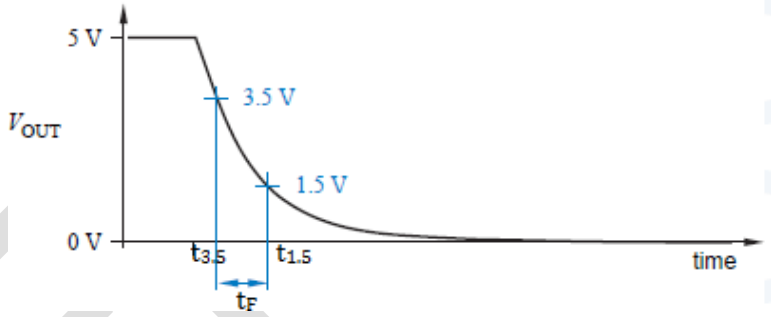


Figure 4.14: Output changes from HIGH state to LOW state

- When input is 0V then R_N value is more because NMOS transistor is in OFF State assume that as 1MΩ and R_P value is less because PMOS transistor is in ON state assume that as 200 Ω.
- When input is 5V then R_N value is less because NMOS transistor is in ON State assume that as 100 Ω and R_P value is more because PMOS transistor is in OFF state assume that as 1M Ω.
- At time t=0 sec then $V_{OUT} \cong 5$ V and at time t=∞ sec then $V_{OUT} \cong 0$ V. Now we have to measure the time at lower limit of HIGH state($t_{3.5}$) and Upper limit of LOW state($t_{1.5}$)
- The output voltage is given by $V_{OUT} = V_{DD}.e^{-t/RN.CL}$ ----------(1)

From Equation-(1) we may get $t = R_N.C_L.\ln\left(\frac{V_{OUT}}{V_{DD}}\right)$

) ----------(2)

By substituting parameters Values V_{DD}=5V, R_N=100Ω, C_L=100pF and V_{OUT}=3.5V

then time $t_{3.5}$= –10 X 10^{-9}.ln ($\frac{3.5V}{5V}$)=3.57nsec ----------(3)

By substituting parameters Values V_{DD}=5V, R_N=100Ω, C_L=100pF and V_{OUT}=1.5V

then time $t_{1.5}$= –10 X 10^{-9}.ln ($\frac{1.5V}{5V}$)=12.04nsec ----------(4)

From Equations (3) & (4) FALL Time (t_F) = $t_{1.5} - t_{3.5}$ = 12.04nsec − 3.57nsec = 8.47nsec

∴ FALL Time (t_F) = 8.47 nsec.

NOTE: In the above examples we have taken ON resistance of PMOS as double of NMOS, So we got $t_R = 2xt_F$.

But in high speed CMOS circuits we will take ON resistances equal, Then $t_R \cong t_F$

Propagation Delay:

- It is the amount of time that it takes for a change in input signal to produce a change in output signal.
- That is propagation delay of a gate is basically time interval between application of input and occurrence of output signal.
- Propagation delay is an important characteristics because it limits the speed at which it operates.
- Propagation delay is inversely proportional to speed. Propagation delay determines using 2 intervals as t_{PLH} and t_{PHL}.
- t_{PLH} is delay time measured when output is changing from logic '0' to logic '1' state.
- t_{PHL} is delay time measured when output is changing from logic '1' state to logic '0' state.
- The Propagation delay is given by $t_P = \max(t_{PLH}, t_{PHL})$, but some times it will be measured as average of t_{PLH} and t_{PHL}, as $\frac{tPLH + tPHL}{2}$

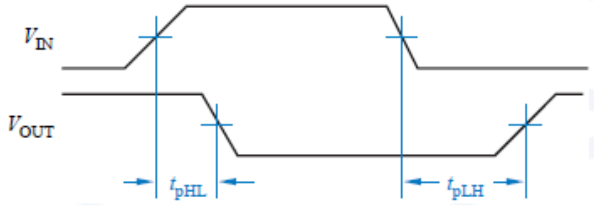


Figure 4.15: Propagation delay.

Power Consumption:

- When a CMOS output changes from Low to High, a transient charging current is supplied to load capacitance which may cause increase in frequency that leads to increase in power consumption by increased average current drawn from V_{DD}.
- The power consumption of a CMOS circuit whose output is not changing is called *static power dissipation* or *quiescent power dissipation.*

- A CMOS circuit consumes significant power only during transitions is called *dynamic power dissipation.*

- Power consumption during transitions is given by $P_T = C_{PD} \times V_{DD}^2 \times f$----------(5) where P_T --- Internal power dissipation during transitions

  V_{DD}--- Power supply voltage.

  C_{PD} --- Power dissipation capacitance

- Power consumption due to capacitive load is given by $P_L = C_L \times (V_{DD}^2/2) \times 2f$. (since 2 transitions/sec). Simplified equation is $P_L = C_L \times V_{DD}^2 \times f$----------(6)

- Hence the total dynamic power in CMOS is given by $P_D = P_T + P_L$. $P_D = (C_{PD} \times V_{DD}^2 \times f) + (C_L \times V_{DD}^2 \times f)$

  $= (C_{PD} + C_L) \times V_{DD}^2 \times f$

  $= C \times V_{DD}^2 \times f$.

- **Write short notes on CMOS logic families?**

The following are commercially available CMOS families

i) 4000 series CMOS logic family: The first commercially successful CMOS family was 4000- series CMOS. these circuits offered the benefit of low power dissipation, they were fairly slow and were not easy to interface with bipolar TTL family

ii) 74 series HC and HCT families: The prefix "74" is simply a number that was used by an early, popular supplier of TTL devices. The first two 74-series CMOS families are HC (High- speed CMOS) and HCT (High-speed CMOS, TTL compatible). Compared with the original 4000 family, HC and HCT both have higher speed and better current sinking and sourcing capability. The HCT family uses a power supply voltage *V*CC of 5 V and can be intermixed with TTL devices, which also use a 5-V supply. The HC family is optimized for use in systems that use CMOS logic exclusively, and can use any power supply voltage between 2 and 6 V.

iii) VHC and VHCT families: CMOS families were introduced in the 1980s and the 1990s. Two of the most recent and probably the most versatile are VHC (Very High-Speed CMOS) and VHCT (Very High-Speed CMOS, TTL compatible). VHC and VHCT differs only in their input levels and their output characteristics are same. Also like HC/HCT, VHC/VHCT outputs have symmetric output drive.

iv) FCT and FCT-T families: In the early 1990s, another CMOS family was launched. The key benefit of the *FCT (Fast CMOS, TTL compatible*) family was its ability to meet or exceed the speed and the output drive capability of the best TTL families while reducing power consumption and maintaining full compatibility with TTL. The original FCT family had the drawback of producing a full 5-V CMOS *V*OH, creating

enormous power dissipationand circuit noise as its outputs swing from 0 V to almost 5 V in high-speed (25 MHz+) applications.AkeyapplicationofFCT-Tisdrivingbusesandotherheavyloads.

❖ **What is Diode logic?Explain the operation of basic AND gate using diode logic?**

DIODE LOGIC: The basic building blocks in Bipolar logic families are diodes and transistors. If a logic circuit design involves only diodes and resistors it is called as 'Diode Logic'.

Design of AND gate Using Diode Logic:

Let two diodes D1 and D2 are connected in parallel whose cathodes are connected to inputs X, Y and Anodes are connected to Supply voltage to form a 2 input AND gate circuit. Z is the Output.

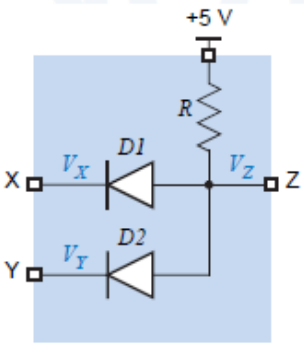


Figure 4.16(a) 2 input AND gate using Diode logic

Operation:

Case-1: Let inputs X = 0V (LOW), Y=0V (LOW);

- Cathodes of both diodes D1 and D2 are grounded
- So both diodes D1 and D2 are in Forward bias.
- Hence output is ideally zero. In practical 0.7 V for Si.
- Hence output Z is LOW.

Case-2: Let inputs X = 0V (LOW), Y=5V (HIGH);

- Cathode of diodes D1 is grounded and 5V is applied to diode D2.
- Diode D1 is in Forward bias and D2 is in reverse bias.
- It pulls output down to Low voltage, Ideally it is zero. In practical 0.7 V for Si.
- Hence output Z is LOW.

Case-3: Let inputs X = 5V (HIGH), Y=0V (LOW);

- Cathode of diodes D2 is grounded and 5V is applied to diode D1.
- Diode D2 is in Forward bias and D1 is in reverse bias.
- It pulls output down to Low voltage, Ideally it is zero. In practical 0.7 V for Si.
- Hence output Z is LOW.

Case-4: Let inputs X = 5V (HIGH), Y=5V (HIGH);

- Cathodes of both diodes D1 and D2 are connected to 5V
- Both diodes D1 and D2 are in reverse bias.

- No current will flows through diodes and resistor.
- It pulls output to supply voltage V_{CC}, ideally 5 V. In practical 4.3 V for Si.
- Hence output Z is HIGH.

Inputs		Diodes		Output
X	Y	D1	D2	Z
0V(LOW)	0V(LOW)	FB	FB	0V(LOW)
0V(LOW)	5V(HIGH)	FB	RB	0V(LOW)
5V(HIGH)	0V(LOW)	RB	FB	0V(LOW)
5V(HIGH)	5V(HIGH)	RB	RB	5V(HIGH)

Table 4.4 Functional table of 2 input AND gate using Diode logic

❖ **What is Diode logic? Explain the operation of basic OR gate using diode logic?**

DIODE LOGIC: The basic building blocks in Bipolar logic families are diodes and transistors. If a logic circuit design involves only diodes and resistors it is called as 'Diode Logic'

Design of OR gate Using Diode Logic:

Let two diodes D1 and D2 are connected in parallel whose Anodes are connected to inputs X, Y and cathodes are grounded to form a 2 input OR gate circuit. Z is the Output.

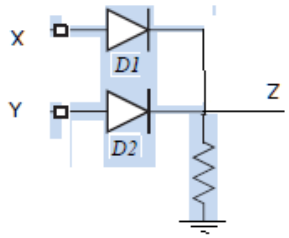


Figure 4.16(b) 2 input OR gate using Diode logic

Operation:

Case-1: Let inputs X = 0V (LOW), Y=0V (LOW);

- Cathodes of both diodes D1 and D2 are grounded through resistor and anodes are connected to input values.
- So both diodes D1 and D2 are in reverse bias (RB).
- Hence output is ideally zero. In practical 0.7 V for Si.
- Hence output Z is LOW.

Case-2: Let inputs X = 0V (LOW), Y=5V (HIGH);

- Diode D2 is in Forward bias (FB) and D1 is in reverse bias (RB).
- Its output is Ideally 5V. In practical 4.3 V for Si.
- Hence output Z is HIGH.

Case-3: Let inputs X = 5V (HIGH), Y=0V (LOW);

- Diode D1 is in Forward bias and D2 is in reverse bias.
- Its output is Ideally 5V. In practical 4.3 V for Si.
- Hence output Z is HIGH.

Case-4: Let inputs X = 5V (HIGH), Y=5V (HIGH);

- Cathodes of both diodes D1 and D2 are connected to 5V
- Both diodes D1 and D2 are in Forward bias.
- Large current will flows through diodes and resistor.
- Its output is ideally 5 V. In practical 4.3 V for Si.
- Hence output Z is HIGH.

Inputs		Diodes		Output
X	Y	D1	D2	Z
0V(LOW)	0V(LOW)	RB	RB	0V(LOW)
0V(LOW)	5V(HIGH)	RB	FB	5V(HIGH)
5V(HIGH)	0V(LOW)	FB	RB	5V(HIGH)
5V(HIGH)	5V(HIGH)	FB	FB	5V(HIGH)

Table 4.5 Functional table of 2 input OR gate using Diode logic

❖ **What is Diode logic? Explain how to cascade diode gates?**

DIODE LOGIC: The basic building blocks in Bipolar logic families are diodes and transistors. If a logic circuit design involves only diodes and resistors it is called as 'Diode Logic'

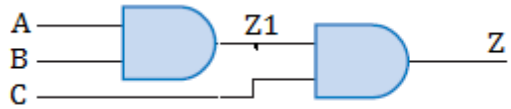


Figure 4.17(a) Cascaded connection of two AND gates

- The voltage levels of logic signal move toward the undefined region.
- In practice diode AND gate normally must be followed by a transistor amplifier to restore logic levels.

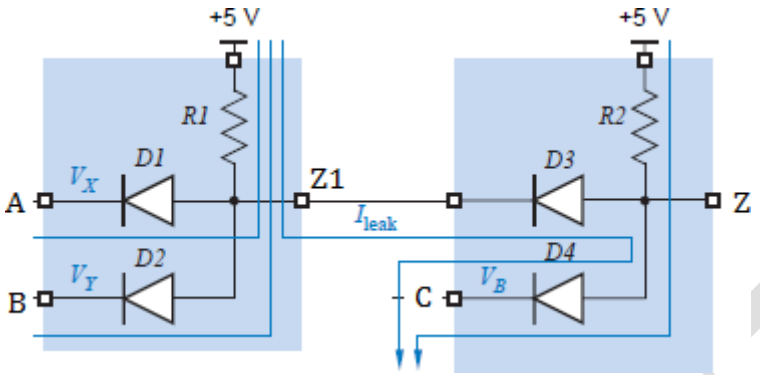


Figure 4.17(b) Diode logic Circuit for Cascaded connection of two AND gates

Operation:

Case-1: Let inputs A=0V(LOW), B=0V(LOW);

- Cathodes of both diodes D1 and D2 are grounded
- So both diodes D1 and D2 are in Forward bias.
- So output is ideally zero. In practical 0.7 V for Si.
- Hence intermediate output Z1 is LOW.
- If C=0V(LOW) then both diodes D3 and D4 are in Forward bias, which gives output Z is LOW.
- If C=5V(HIGH) then Diode D3 is in Forward bias and D4 is in reverse bias, which gives output Z is LOW

Case-2: Let inputs X=0V(LOW), Y=5V(HIGH);

- Cathode of diodes D1 is grounded and 5V is applied to diode D2.
- Diode D1 is in Forward bias and D2 is in reverse bias.
- It pulls output down to Low voltage, Ideally it is zero. In practical 0.7 V for Si.
- Hence intermediate output Z1 is LOW.
- If C=0V(LOW) then both diodes D3 and D4 are in Forward bias, which gives output Z is LOW.
- If C=5V(HIGH) then Diode D3 is in Forward bias and D4 is in reverse bias, which gives output Z is LOW

Case-3: Let inputs X=5V(HIGH), Y=0V(LOW);

- Cathode of diodes D2 is grounded and 5V is applied to diode D1.
- Diode D2 is in Forward bias and D1 is in reverse bias.
- It pulls output down to Low voltage, Ideally it is zero. In practical 0.7 V for Si.

- Hence intermediate output Z1 is LOW.
- If C = 0V (LOW) then both diodes D3 and D4 are in Forward bias, which gives output Z is LOW.
- If C = 5V (HIGH) then Diode D3 is in Forward bias and D4 is in reverse bias, which gives output Z is LOW

Case-4: Let inputs X=5V (HIGH), Y=5V (HIGH);

- Cathodes of both diodes D1 and D2 are connected to 5V
- Both diodes D1 and D2 are in reverse bias.
- No current will flows through diodes and resistor.
- It pulls output to supply voltage V_{CC}, ideally 5 V. In practical 4.3 V for Si.
- Hence intermediate output Z1 is HIGH.
- If C = 0V (LOW) then both diodes D3 reverse bias and D4 is in forward bias, which gives output Z is LOW.
- If C = 5V (HIGH) then both Diodes D3 and D4 is in reverse bias, which gives output Z is HIGH

Inputs		Diodes		Intermediate Output	Input	Diodes		Final Output
A	B	D1	D2	Z1	C	D3	D4	Z
0V(LOW)	0V(LOW)	FB	FB	0V(LOW)	0V(LOW)	FB	FB	0V(LOW)
					5V(HIGH)	FB	RB	0V(LOW)
0V(LOW)	5V(HIGH)	FB	RB	0V(LOW)	0V(LOW)	FB	FB	0V(LOW)
					5V(HIGH)	FB	RB	0V(LOW)
5V(HIGH)	0V(LOW)	RB	FB	0V(LOW)	0V(LOW)	FB	FB	0V(LOW)
					5V(HIGH)	FB	RB	0V(LOW)
5V(HIGH)	5V(HIGH)	RB	RB	5V(HIGH)	0V(LOW)	RB	FB	0V(LOW)
					5V(HIGH)	RB	RB	5V(HIGH)

Table 4.6 Functional table of Cascaded connection of two AND gates

❖ What is Bipolar logic? Explain Inverter operation using Bipolar logic?

- The basic building blocks in Bipolar logic families are diodes and transistors. If a logic circuit design involves diodes, transistors and resistors then it is called as 'Bipolar Logic'.

- Bipolar junction transistor is a three terminal device with regions or terminals as Emitter, Base and Collector formed by back to back connection of PN-junction diode as N-P-N Transistor and P-N-P transistor.
- In bipolar junction transistor among three terminals Base region is lightly doped, Emitter region is heavily doped compared to Base and lightly doped compared to Collector and Collector is heavily doped region.
- Bipolar junction transistor is used as a *current controlled switch.*
- The Base current decides the operating region of the transistor.
- Bipolar junction transistor can be operated in three regions or modes. They are

i) Cut-off Region: Both Base-Collector and Base-Emitter junctions are in reverse bias, only reverse current which is negligible will flows in the Transistor.

ii) Active Region: The base–emitter junction is forward biased and the base–collector junction is reverse biased. The collector–emitter current is approximately proportional to the base current.

iii) Saturation Region: Both Base-Collector and Base-Emitter junctions are in forward bias and voltage drops between Collector-Emitter and Base-Emitter are small(V_{CESat}, V_{BESat}). A large amount of current if flows which is controlled by R_C.

Transistor Logic Inverter:

Let an N-P-N transistor connected in a CE configuration.

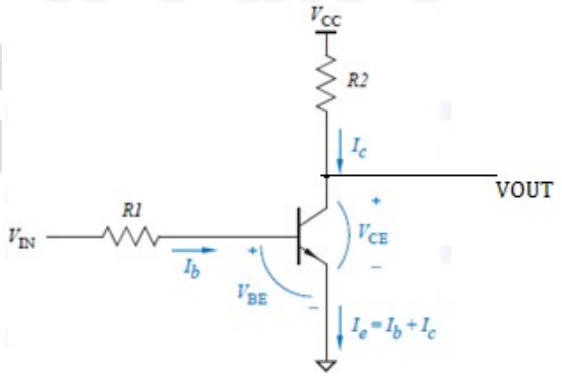


Figure 4.18(a): Common-emitter configuration of an N-P-N transistor.

Case-1: Let Vin = 0 V, then Base-Emitter junction is not in forward bias, so no Base current (I_B) will flows, Transistor is in OFF condition hence no current will flows through collector (I_C). Then the output voltage $V_{OUT} = V_{CC} = +5$ V.

As per Kirchoff's Voltage Law (KVL)

$V_{CE} = V_{CC} - Ic \times R2$

$V_{CE} = V_{CC} - (0) \times R2$ (SinceIc=0)

$V_{CE} = V_{CC} = +5\,V$

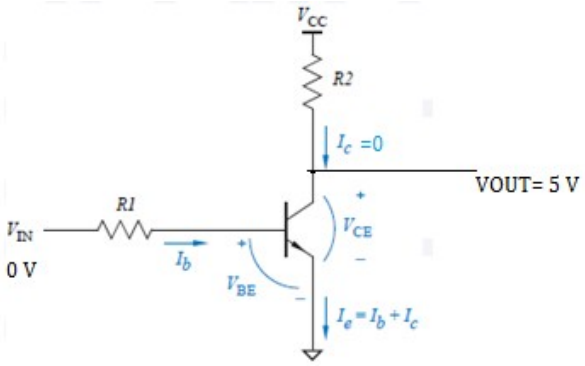


Figure 4.18(b): N-P-N transistor operation when Vin = 0 V.

Case-2: Let Vin = 5 V, then Base-Emitter junction is in forward bias, so Base current (I_B) will flows(select R1 and R2 values as to allow sufficient amount of I_B), Transistor is in ON condition hence a large current(I_C) will flows through collector. Then the output voltage $V_{OUT} = V_{CESat}$ = between 0.2 to 0.3 V.

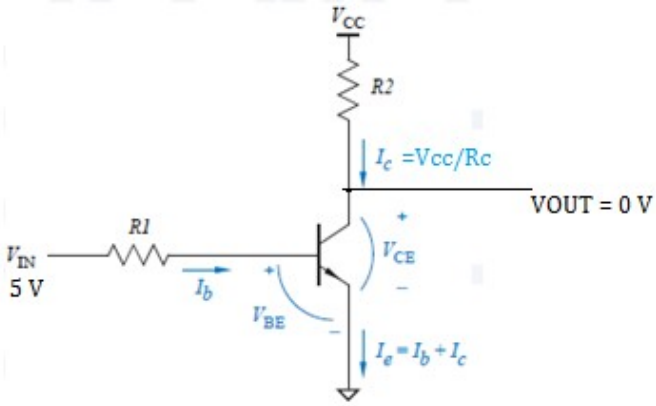


Figure 4.18(c): N-P-N transistor operation when Vin = 5 V.

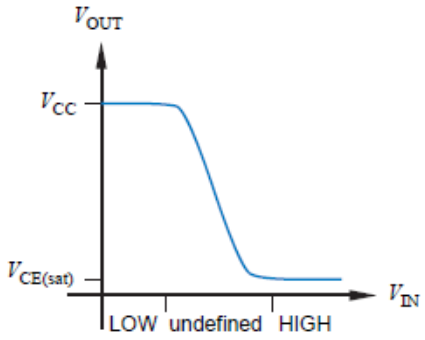


Figure 4.18(d): Inverting operation of N-P-N transistor.

❖ **With neat diagram explain the operation of 2 input NAND gate using TTL logic?**

- The NAND function is obtained by connecting AND gate with an inverting circuit.
- The circuit's operation is understood by dividing it into the three parts as
  - Diode AND gate and input protection.
  - Phase splitter.
  - Output stage.

Diode AND gate and input protection: Diodes D1, D2 and resistor R1 form a diode AND gate, Clamp diodes D5 and D6 do nothing in normal operation, but limit undesirable large negative voltages may occur on HIGH-to-LOW input transitions as a result of transmission- line effects. They are called as protective diodes.

Phase splitter: Transistor Q2 and the surrounding R2, R3 and R4 resistors form a phase splitter that controls the output stage. Depending on whether the diode AND gate produces a "low" or a "high" voltage at VA, Q2 is either cut off or turned on. Diode AND gate and input protection and Phase splitter form NAND gate.

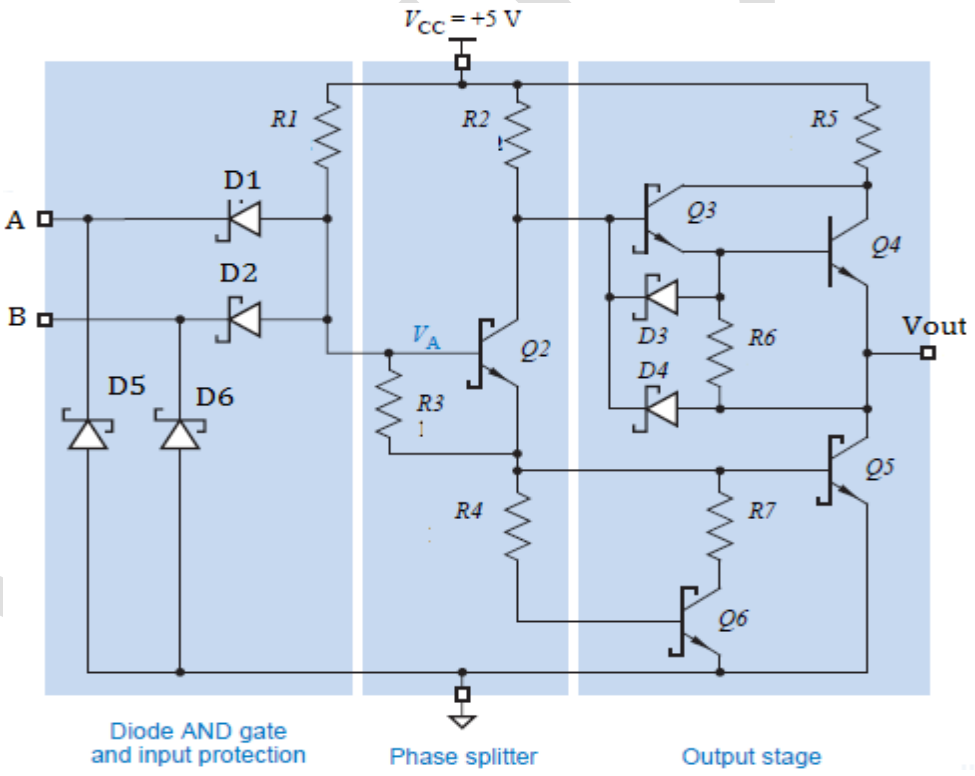


Figure 4.19: TTL-2 input NAND gate

Output stage: The output stage has two transistors, Q4 and Q5, only one of which is ON at any time. The TTL output stage is sometimes called a totem-pole or push-pull output which is similar to the p-channel and n-channel transistors in CMOS. Q4 and Q5 transistors provide active pull-up and pull-down to the HIGH and LOW states, respectively. Except Q4 remaining all transistors are Schottky clamped transistors because it can't saturate. Q3 and Q4 connected in Darlington pair which may provide shorter output rise time when switching from OFF to ON. Q6 regulates current flow into base of Q5 and helps to it in turning to OFF.

Inputs		Transistors					Output
A	B	Q2	Q3	Q4	Q5	Q6	V_{OUT}
0V (LOW)	0V(LOW)	OFF	ON	ON	OFF	OFF	5V(HIGH)
0V (LOW)	5V (HIGH)	OFF	ON	ON	OFF	OFF	5V(HIGH)
5V (HIGH)	0V(LOW)	OFF	ON	ON	OFF	OFF	5V(HIGH)
5V (HIGH)	5V (HIGH)	ON	OFF	OFF	ON	ON	0V (LOW)

Table 4.7 Functional table of TTL-2 input NAND gate.

❖ **With neat diagram explain the operation of 2 input NOR gate using TTL logic?**

- The NOR function is obtained by connecting OR gate with an inverting circuit.
- The circuit's operation is understood by dividing it into the three parts as
  - Diode inputs and input protection.
  - OR function and Phase splitter.
  - Output stage.
- If either input X or Y is HIGH, the corresponding phase-splitter transistor *Q2X* or *Q2Y* is turned on, which turns off *Q3* and *Q4* while turning on *Q5* and *Q6*, and the output is LOW.
- If both inputs are LOW, then both phase-splitter transistors are off, and the output is forced HIGH.
- The speed, input, and output characteristics of a TTL NOR gate are comparable to those of a TTL NAND.
- The TTL NOR gate's input circuits, phase splitter, and output stage are almost identical to those of an TTL NAND gate. The difference is that TTL NAND gate uses

diodes to perform the AND function, while TTL NOR gate uses parallel transistors in the phase splitter to perform the OR function.

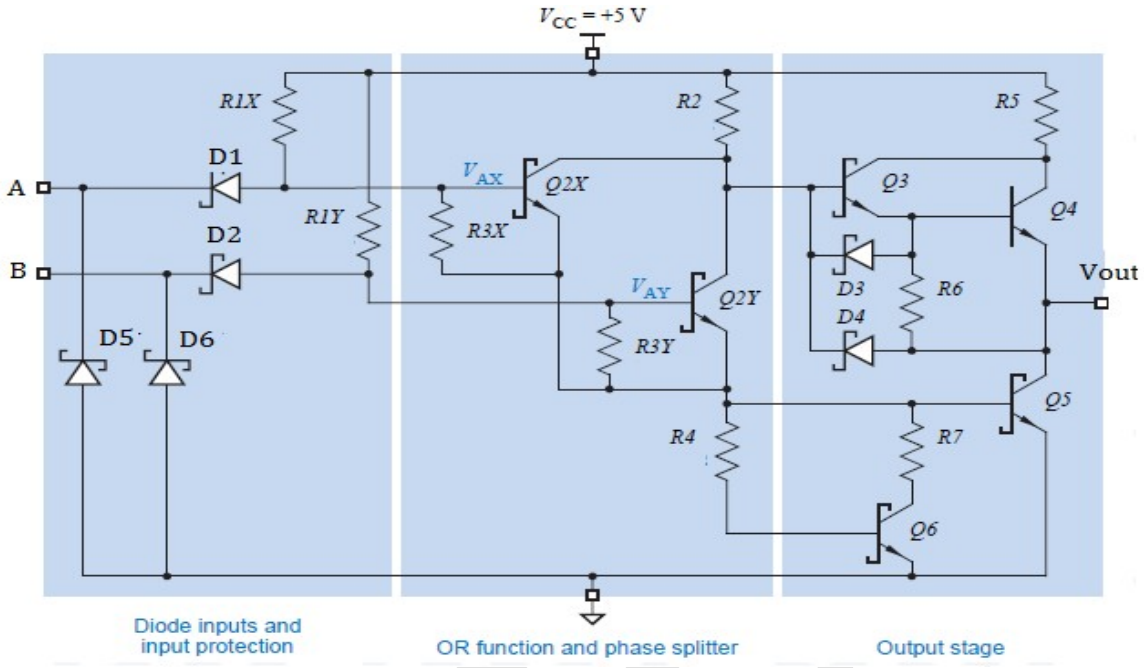


Figure 4.20: TTL-2 Input NOR gate

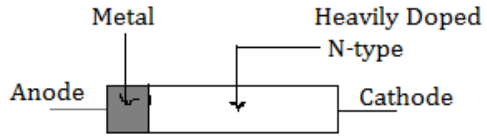
- An *n*-input NOR gate uses more transistors and resistors and is thus more expensive in silicon area than an *n*-input NAND. Internal leakage current limits the number of *Q2* transistors that can be placed in parallel, so NOR gates have poor fan-in. As a result, NOR gates are less commonly used than NAND gates in TTL designs.
- The most "natural" TTL gates are inverting gates like NAND and NOR. Non-inverting TTL gates include an extra inverting stage, typically between the input stage and the phase splitter. As a result, non-inverting TTL gates are typically larger and slower than the inverting gates.

Inputs		Transistors						Output
A	B	Q2X	Q2Y	Q3	Q4	Q5	Q6	V_{OUT}
0V(LOW)	0V(LOW)	OFF	OFF	ON	ON	OFF	OFF	5V (HIGH)
0V(LOW)	5V (HIGH)	OFF	ON	OFF	OFF	ON	ON	0V(LOW)
5V (HIGH)	0V(LOW)	ON	OFF	OFF	OFF	ON	ON	0V(LOW)
5V (HIGH)	5V (HIGH)	ON	ON	OFF	OFF	ON	ON	0V(LOW)

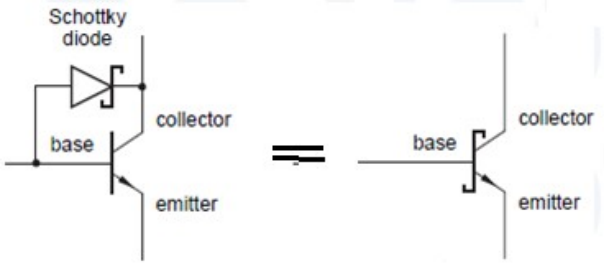
Table 4.8 Functional table of TTL-2 input NOR gate.

❖ **What is Schottky transistor? Explain its need in TTL family?**

- When the input of a saturated transistor is changed, the output does not change immediately; it takes extra time, called *storage time*, to come out of saturation.
- The storage time presents because the excess charge carriers in the base region provides Forward biasing which may provide deep saturation to the transistor.
- In fact, storage time accounts for a significant portion of the propagation delay in the original TTL logic family.
- The storage time can be reduced by removing excess charge carriers in the base region to prevent deep saturation to the transistor before transistor is switched from ON to OFF state.
- This is achieved by placing a Schottky diode between the base and collector of each transistor.
- Schottky diodes have little capacitance, fast recovery time, Hence it switched without storage delay time.
- There is Ge diode is available with low cut-in voltage (0.3 V) but it can't be used in this circuit which is designed by Si material. So Schottky diode is developed which has 0.3 V as cut-in voltage.
- Transistors, which do not saturate, are called Schottky clamped transistor or Schottky transistor.



(a)



(b)

Figure 4.21 (a) Construction of Schottky Diode (b)Schottky Transistor

❖ **Design a buffer circuit using CMOS logic and write its functional table?**

- Buffer Circuit gives output as a signal, Exactly similar to the signal which is applied at the input without having any distortion or loss of information.
- We know the basic CMOS circuit gives Inverting operation.
- So by using CMOS logic the buffer circuit is obtained by connecting two Basic CMOS circuits in Cascading connection.
- Cascading connection means first circuit output is connected as input to the second circuit.

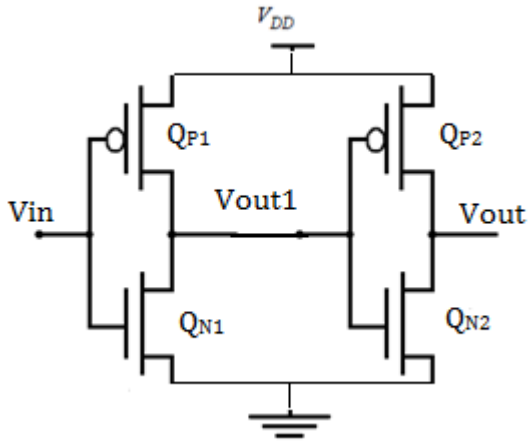


Figure 4.22 Circuit diagram of Buffer circuit using CMOS logic

Operation:

Case (1): When V_{in}= 0V (LOW), then NMOS transistor (Q_{N1}) is OFF state, since its V_{GS} = 0 V. PMOS Transistor (Q_{P1}) is in ON state, Since its V_{GS} is large negative (-5V). So PMOS presents only a small resistance between V_{DD} and output. Hence intermediate output V_{OUT1} is 5 V(HIGH). Then it is applied as input to the second CMOS inverter circuit. Then NMOS transistor (Q_{N2}) is in ON state and PMOS Transistor (Q_{P2}) is in OFF state, then final output V_{OUT} is 0V(LOW).

Case (2): When V_{in}=5V(HIGH), then PMOS transistor (Q_{P1}) is in OFF state, since its V_{GS} =0 V. NMOS transistor (Q_{N1}) is in ON state Since its V_{GS} is large positive (+5 V). So NMOS presents only a small resistance between output and ground. Hence intermediate output V_{OUT1} is 0 V. Then it is applied as input to the second CMOS inverter circuit. Then NMOS transistor (Q_{N2}) is in OFF state and PMOS Transistor (Q_{P2}) is in ON state, then final output V_{OUT} is 5V (HIGH).

Note: From case (i) & (ii) we can conclude that the Buffer circuit operation is obtained by cascading of two basic CMOS circuits.

Input	Transistors				Intermediate Output	Final Output
V_{IN}	Q_{N1}	Q_{N2}	Q_{P1}	Q_{P2}	V_{OUT1}	V_{OUT}
0 V(LOW)	OFF	ON	ON	OFF	5 V(HIGH)	0 V(LOW)
5V(HIGH)	ON	OFF	OFF	ON	0 V(LOW)	5V(HIGH)

Table 4.9 Functional table of Buffer Circuit using CMOS Logic.

❖ **Draw and Explain the operation of 2 input AND gate using CMOS logic?**

Implementation of 2 Input AND gate using CMOS logic:

- ✓ AND gate is one of the basic logic gates to perform the digital operation on the input signals.
- ✓ It is the combination of NAND Gate followed by NOT gate i.e. it is the opposite operation of NAND gate where the Logic AND gate is complementary of NAND gate.
- ✓ The logic output of AND gate is high (TRUE) only when the inputs are high (TRUE).

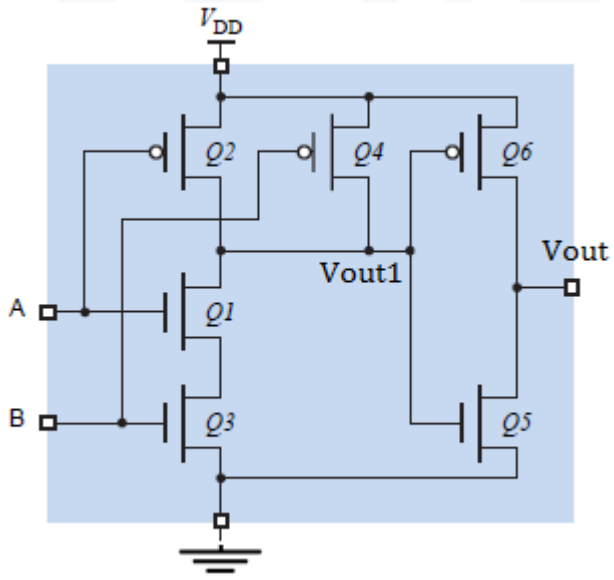


Figure 4.23: A 2-Input AND gate using CMOS logic

Operation:

Case (i): When A=B=0V(LOW), then both NMOS transistors (Q1, Q3) are in OFF state, since its $V_{GSA} = V_{GSB} =$ 0 V. Both PMOS transistors (Q2, Q4) are in ON state. Since its V_{GSA} and V_{GSB} voltage is large negative (-5V). So PMOS transistors presents only a small resistance between V_{DD} and output. Hence intermediate output V_{OUT1} is 5V(HIGH). Then it is applied as

input to CMOS inverter circuit, so NMOS transistors(Q5) is in ON state and PMOS transistors(Q6) is in OFF state, Hence the output V_{OUT} is obtained as 0V(LOW).

Case (ii): When A=B=5V(HIGH), then both PMOS transistors (Q2, Q4) are in OFF state, since its input voltages $V_{GSA} = V_{GSB} = 0$ V. Both NMOS transistors (Q1, Q3) are in ON state Since its V_{GSA} and V_{GSB} is large positive (+5V). So NMOS transistors presents only a small resistance between Output and ground. Hence intermediate output V_{OUT1} is 0V(LOW). Then it is applied as input to CMOS inverter circuit, so NMOS transistors (Q5) is in OFF state and PMOS transistors(Q6) is in ON state, Hence the output V_{OUT} is obtained as 5V(HIGH).

Case (iii): When A= 0V(LOW) and B=5V(HIGH), then NMOS transistor (Q1) is in OFF state, since its input voltage V_{GSA}= 0 V and NMOS transistor (Q3) is in ON state since its V_{GSB}=5V. PMOS transistor (Q2) is in ON state, since its V_{GSA}= -5V and PMOS transistor (Q4) is in OFF state since its V_{GSB}=0V. So PMOS transistor Q2 presents only a small resistance between V_{DD} and output. Hence intermediate output V_{OUT1} is 5V(HIGH). Then it is applied as input to CMOS inverter circuit, so NMOS transistors(Q5) is in ON state and PMOS transistors (Q6) is in OFF state, Hence the output V_{OUT} is obtained as 0V(LOW).

Case (iv): When A=5V(HIGH) and B=0V(LOW), then NMOS transistor (Q1) is in ON state, since its input voltage V_{GSA}=+5V and NMOS transistor (Q3) is in OFF state since its V_{GSB}=0V. PMOS transistor (Q2) is in OFF state, since its V_{GSA}= 0V and PMOS transistor (Q4) is in ON state since its V_{GSB}=-5V. So PMOS transistor Q4 presents only a small resistance between V_{DD} and output. Hence output is 5V(HIGH). Then it is applied as input to CMOS inverter circuit, so NMOS transistors(Q5) is in ON state and PMOS transistors (Q6) is in OFF state, Hence the output V_{OUT} is obtained as 0V(LOW).

Inputs		Transistors						Intermediate Output	Final Output
A	B	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	V_{OUT1}	V_{OUT}
0V(LOW)	0V(LOW)	OFF	ON	OFF	ON	ON	OFF	5 V(HIGH)	0 V(LOW)
0V(LOW)	5V(HIGH)	OFF	ON	ON	OFF	ON	OFF	5 V(HIGH)	0 V(LOW)
5V(HIGH)	0V(LOW)	ON	OFF	OFF	ON	ON	OFF	5 V(HIGH)	0 V(LOW)
5V(HIGH)	5V(HIGH)	ON	OFF	ON	OFF	OFF	ON	0 V(LOW)	5V(HIGH)

Table 4.10 Functional table of 2-input AND gate using CMOS Logic.

❖ **Draw and Explain the operation of 2 input OR gate using CMOS logic?**

Implementation of 2 Input NOR gate using CMOS logic:

- ✓ OR gate is one of the basic logic gates to perform the digital operation on the input signals.
- ✓ It is the combination of NOR Gate followed by NOT gate i.e. it is the opposite operation of NOR gate where the Logic OR gate is complementary of NOR gate.
- ✓ The logic output of OR gate is LOW (False) only when the inputs are LOW (False).

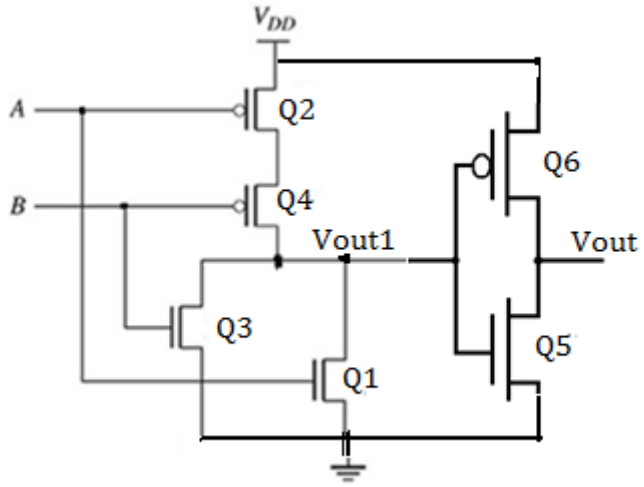


Figure 4.24: A 2-Input OR gate using CMOS logic

Operation:

Case (i): When A=B=0V(LOW), then both NMOS transistors (Q1, Q3) are in OFF state, since its $V_{GSA} = V_{GSB} =$ 0 V. Both PMOS transistors (Q2, Q4) are in ON state. Since its V_{GSA} and V_{GSB} voltage is large negative (-5V). So PMOS transistors presents only a small resistance between V_{DD} and output. Hence intermediate output V_{OUT1} is 5V(HIGH). Then it is applied as input to CMOS inverter circuit, so NMOS transistors(Q5) is in ON state and PMOS transistors(Q6) is in OFF state, Hence the output V_{OUT} is obtained as 0V(LOW).

Case (ii): When A=B=5V(HIGH), then both PMOS transistors (Q2, Q4) are in OFF state, since its input voltages $V_{GSA} = V_{GSB} = 0$ V. Both NMOS transistors (Q1, Q3) are in ON state Since its V_{GSA} and V_{GSB} is large positive (+5V). So NMOS transistors presents only a small resistance between Output and ground. Hence intermediate output V_{OUT1} is 0V(LOW). Then it is applied as input to CMOS inverter circuit, so NMOS transistors (Q5) is in OFF state and PMOS transistors (Q6) is in ON state, Hence the output V_{OUT} is obtained as 5V(HIGH).

Case (iii): When A= 0V(LOW) and B=5V(HIGH), then NMOS transistor (Q1) is in OFF state, since its input voltage V_{GSA}= 0 V and NMOS transistor (Q3) is in ON state since its V_{GSB}=5V. PMOS transistor (Q2) is in ON state, since its V_{GSA}= -5V and PMOS transistor (Q4) is in OFF state since its V_{GSB}=0V. So PMOS transistor Q2 presents only a small resistance between V_{DD} and output. Hence intermediate output V_{OUT1} is 0V(LOW). Then it is applied as input to CMOS inverter circuit, so NMOS transistors (Q5) is in OFF state and PMOS transistors (Q6) is in ON state, Hence the output V_{OUT} is obtained as 5V(HIGH).

Case (iv): When A=5V(HIGH) and B=0V(LOW), then NMOS transistor (Q1) is in ON state, since its input voltage V_{GSA}=+5V and NMOS transistor (Q3) is in OFF state since its V_{GSB}=0V. PMOS transistor (Q2) is in OFF state, since its V_{GSA}= 0V and PMOS transistor (Q4) is in ON state since its V_{GSB}=-5V. So PMOS transistor Q4 presents only a small resistance between V_{DD} and output. Hence output is 0V(LOW). Then it is applied as input to CMOS inverter circuit, so NMOS transistors (Q5) is in OFF state and PMOS transistors (Q6) is in ON state, Hence the output V_{OUT} is obtained as 5V(HIGH).

Inputs		Transistors						Intermediate Output	Final Output
A	B	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	V_{OUT1}	V_{OUT}
0V(LOW)	0V(LOW)	OFF	ON	OFF	ON	ON	OFF	5V(HIGH)	0V(LOW)
0V(LOW)	5V(HIGH)	OFF	ON	ON	OFF	OFF	ON	0V(LOW)	5V(HIGH)
5V(HIGH)	0V(LOW)	ON	OFF	OFF	ON	OFF	ON	0V(LOW)	5V(HIGH)
5V(HIGH)	5V(HIGH)	ON	OFF	ON	OFF	OFF	ON	0V(LOW)	5V(HIGH)

Table 4.11 Functional table of 2-input OR gate using CMOS Logic

❖ **Draw and Explain the implementation of CMOS AND-OR-INVERT Logic?**

- CMOS circuits can perform two levels of logic with just a single "level" of transistors.
- CMOS AND-OR-INVERT (AOI) gate is implemented by taking a logic function $Y=\overline{AB+CD}$
- In the above function AB forms a 2 input AND gate and CD also forms another 2 input AND gate. Both are combined with OR logic and entire function contains Invert or complement operation.
- Finally they form **AND-OR-INVERT Logic.**

 The Expression Y=A should be represented using De-Morgan's principle by assuming AB as P and CD as Q then Y= = . = .

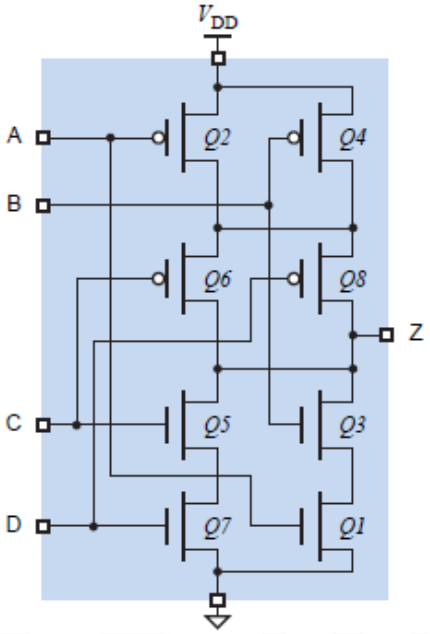
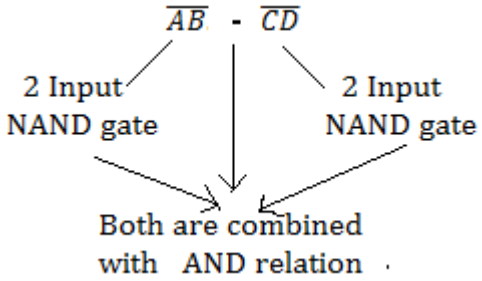


Figure 4.25: CMOS AND-OR-INVERT logic



Transistor	NAND gate/ OR relation	NOR gate/ AND relation
NMOS	Series	Parallel
PMOS	Parallel	Series

Table 4.12 Rules to design any Boolean function using CMOS logic

A	B	C	D	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Z
L	L	L	L	off	on	off	on	off	on	off	on	H
L	L	L	H	off	on	off	on	off	on	on	off	H
L	L	H	L	off	on	off	on	on	off	off	on	H
L	L	H	H	off	on	off	on	on	off	on	off	L
L	H	L	L	off	on	on	off	off	on	off	on	H
L	H	L	H	off	on	on	off	off	on	on	off	H
L	H	H	L	off	on	on	off	on	off	off	on	H
L	H	H	H	off	on	on	off	on	off	on	off	L
H	L	L	L	on	off	off	on	off	on	off	on	H
H	L	L	H	on	off	off	on	off	on	on	off	H
H	L	H	L	on	off	off	on	on	off	off	on	H
H	L	H	H	on	off	off	on	on	off	on	off	L
H	H	L	L	on	off	on	off	off	on	off	on	L
H	H	L	H	on	off	on	off	off	on	on	off	L
H	H	H	L	on	off	on	off	on	off	off	on	L
H	H	H	H	on	off	on	off	on	off	on	off	L

Table 4.13 Functional Table for : CMOS AND-OR-INVERT logic

 Any given Boolean expression should be expressed in terms of Universal gate (NAND/NOR) format by using De-Morgan's principle.

 All NMOS transistors in NAND gate implementation should be connected in SERIES and PMOS transistors are in PARALLEL connection.

- All NMOS transistors in NOR gate implementation should be connected in PARALLEL and PMOS transistors are in SERIES connection.
- If those implementations are involving AND relation then NMOS transistors of both implementations should be connected in PARALLEL and PMOS transistors of both implementations should be connected in SERIES connection as shown in Figure 4.25
- If those implementations are involving OR relation then NMOS transistors of both implementations should be connected in SERIES and PMOS transistors of both implementations should be connected in PARALLEL connection.

❖ **Draw and Explain the implementation of CMOS AND-OR-INVERT Logic?**

- CMOS circuits can perform two levels of logic with just a single "level" of transistors.
- CMOS OR-AND-INVERT(OAI) gate is implemented by taking a logic function Y=( )C
- In the above function A+B forms a 2 input OR gate and C+D also forms another 2 input OR gate. Both are combined with AND logic and entire function contains Invert or complement operation.
- Finally they form **OR- AND-INVERT Logic.**

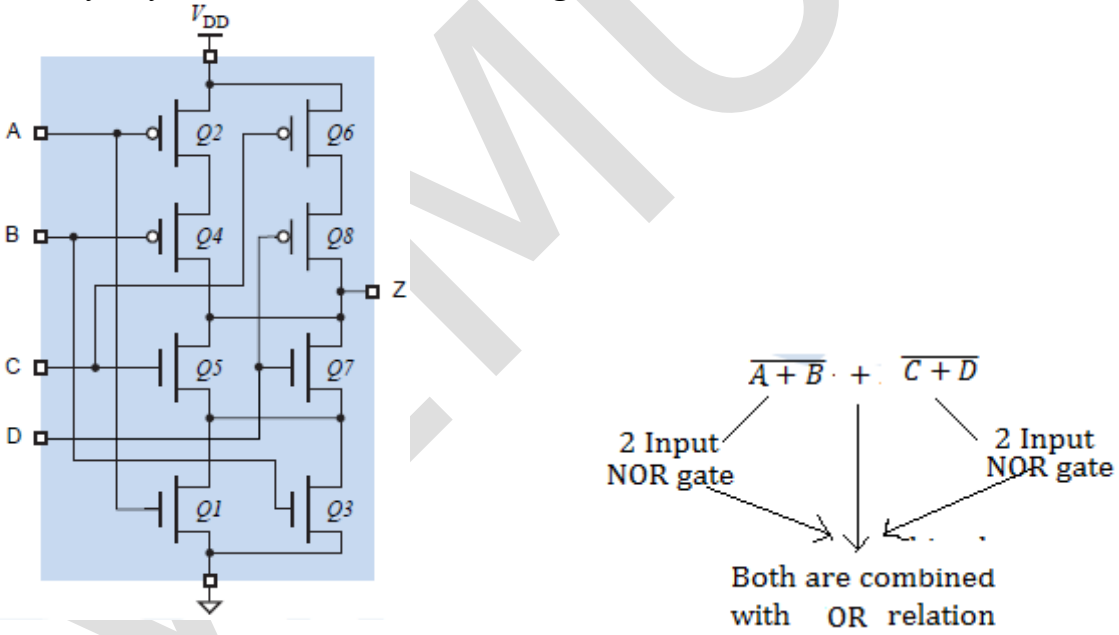


Figure 4.26: CMOS OR-AND-INVERT logic

- Any given Boolean expression should be expressed in terms of Universal gate (NAND/NOR) format by using De-Morgan's principle.
- All NMOS transistors in NAND gate implementation should be connected in SERIES and PMOS transistors are in PARALLEL connection.
- All NMOS transistors in NOR gate implementation should be connected in PARALLEL and PMOS transistors are in SERIES connection.
- If those implementations are involving AND relation then NMOS transistors of both implementations should be connected in PARALLEL and PMOS transistors of both implementations should be connectedinSERIESconnectionasshowninFigure4.25
- If those implementations are involving OR relation then NMOS transistors of both implementations should be connected in SERIES and PMOS transistors of both implementations should be connected in PARALLEL connection.

A	B	C	D	*Q1*	*Q2*	*Q3*	*Q4*	*Q5*	*Q6*	*Q7*	*Q8*	Z
L	L	L	L	off	on	off	on	off	on	off	on	H
L	L	L	H	off	on	off	on	off	on	on	off	H
L	L	H	L	off	on	off	on	on	off	off	on	H
L	L	H	H	off	on	off	on	on	off	on	off	H
L	H	L	L	off	on	on	off	off	on	off	on	H
L	H	L	H	off	on	on	off	off	on	on	off	L
L	H	H	L	off	on	on	off	on	off	off	on	L
L	H	H	H	off	on	on	off	on	off	on	off	L
H	L	L	L	on	off	off	on	off	on	off	on	H
H	L	L	H	on	off	off	on	off	on	on	off	L
H	L	H	L	on	off	off	on	on	off	off	on	L
H	L	H	H	on	off	off	on	on	off	on	off	L
H	H	L	L	on	off	on	off	off	on	off	on	H
H	H	L	H	on	off	on	off	off	on	on	off	L
H	H	H	L	on	off	on	off	on	off	off	on	L
H	H	H	H	on	off	on	off	on	off	on	off	L

Table 4.14 Functional Table for : CMOS OR-AND-INVERT logic

❖ **Explain about sinking and sourcing currents in TTL family?**

 When output of TTL NAND gate circuit is connected to input of another TTL NAND gate circuit then

Case (i): Sinking Current

- ✓ For input LOW driven by output of TTL circuit then the transistor Q5 is in ON state and it provides the path to ground flows current out from Diode B.
- ✓ When current flows into a TTL output in LOW State output is called SINKING CURRENT.

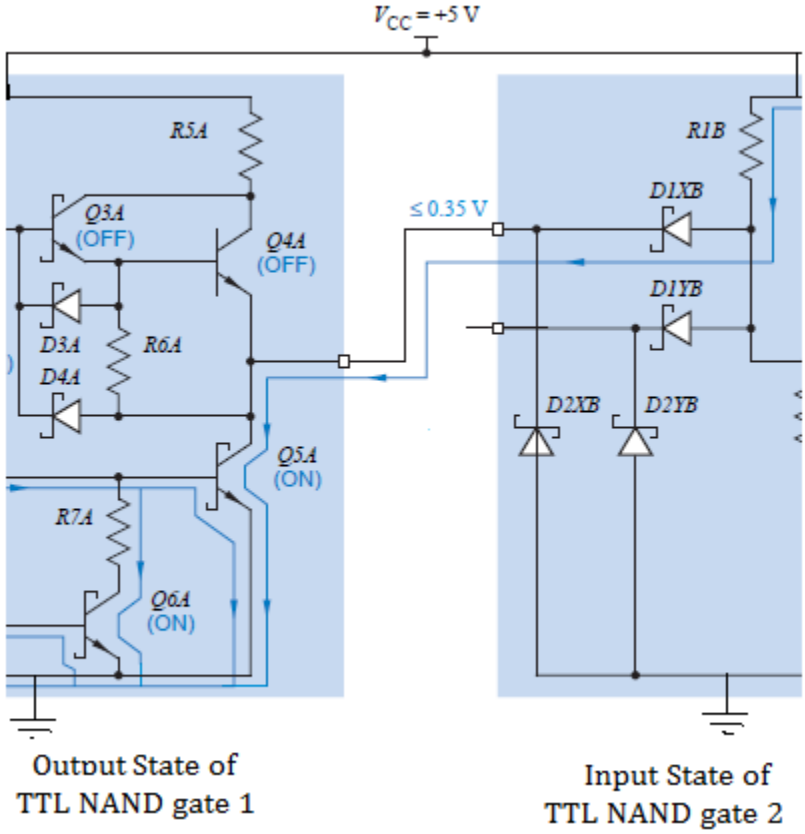


Figure 4.27: TTL output Driving a TTL input LOW (Sinking Current)

Case (ii): Sourcing Current

- ✓ For input HIGH driven by output of TTL circuit then the transistor Q4 is in ON state then a small amount of leakage current flows through diode A.
- ✓ When current flows out of a TTL output in HIGH State output is called SOURCING CURRENT.

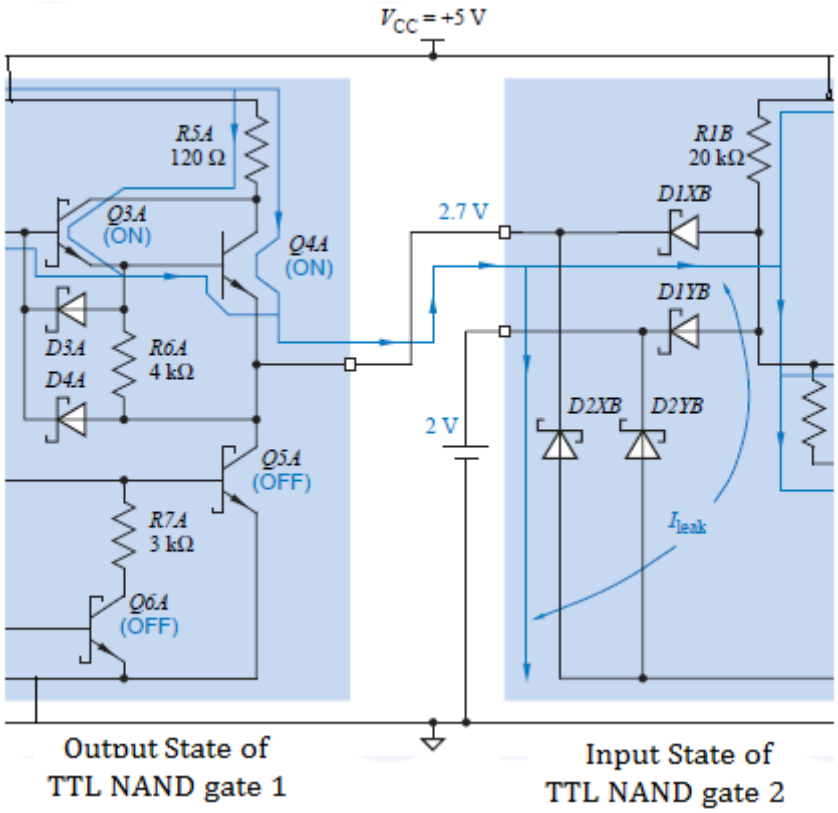


Figure 4.27: TTL output Driving a TTL input HIGH (Sourcing Current)

❖ **Explain the following regarding TTL Logic families?**

(i) Voltage levels & Noise margin **(ii) Supply Voltage & Temperature range**

(iii) Fan-Out **(iv) Un-used inputs**

(i) Voltage levels:

- ✓ **High Level Input Voltage (V_{IH}(min)):** It is the minimum voltage required for a logic '1' at an input. Any voltage below this level will not be accepted as HIGH by logic Circuit.
- ✓ **High Level Output Voltage (V_{OH}(min)):** It is the minimum voltage required for a logic '1' at an Output under defined load conditions.
- ✓ **Low Level Input Voltage (V_{IL}(max)):** It is the maximum voltage required for a logic '0' at an input. Any voltage above this level will not be accepted as LOW by logic Circuit.
- ✓ **Low Level Output Voltage (V_{OL}(max)):** It is the maximum voltage required for a logic '0' at an Output under defined load conditions.

Voltage Parameters	74	74S	74LS	74AS	74ALS	74F
V_{OH}(min) (V)	2.4	2.7	2.7	2.5	2.5	2.5
V_{OL}(max) (V)	0.4	0.5	0.5	0.5	0.5	0.5
V_{IH}(min) (V)	2.0	2.0	2.0	2.0	2.0	2.0
V_{IL}(max) (V)	0.8	0.8	0.8	0.8	0.8	0.8

Table 4.16 Voltage levels of TTL logic Families.

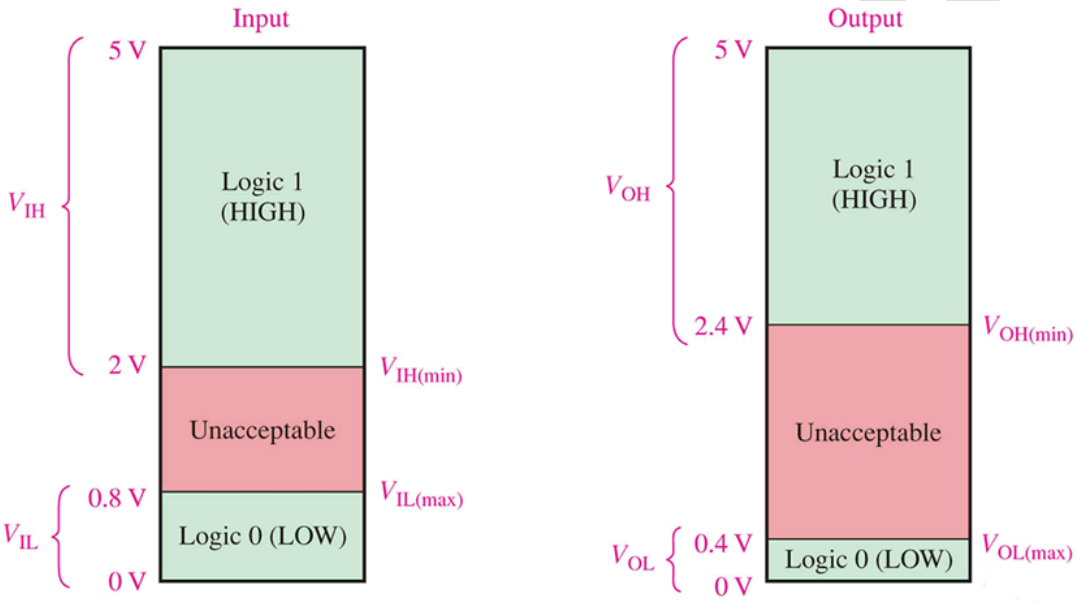


Figure 4.28: Voltage levels TTL family

Noise Margin:

- ✓ In practice sometimes unwanted signals called noise may occur by droping input voltage less than V_{IH}(min) and rise above the V_{IL}(max).
- ✓ To avoid this we kept V_{OH}(min), V_{OL}(max) to certain fraction of voltages below and above respectively. These limits are called DC Noise margins.
- ✓ DC noise margin allows digital circuit to function properly.
- ✓ The noise margin in HIGH state is $V_{NH}=V_{OH}(min)-V_{IH}(min)=2.4V-2.0V=0.4V$.
- ✓ The noise margin in LOW state is $V_{NL}=V_{IL}(max)-V_{OL}(max)=0.8V-0.4V=0.4V$

(ii) Supply Voltage & Temperature range:

- ✓ 54 Series have greater tolerance of voltage & temperature than 74 series.
- ✓ These 54 Series Devices are used when Reliable operation is necessary such as military and space applications.
- ✓ Only disadvantage is they are Expensive.

PARAMETER	74 SERIES	54 SERIES
Supply Voltage	5 V	5 V
Reliable Range	4.75 V – 5.25 V	4.5 V – 5.5 V
Temperature	0 – 70 ^{0} C	-55 – +125 ^{0} C

Table 4.15: Supply Voltage & Temperature range in TTL families.

(iii) Fan-Out:

- ✓ Fan-out is the measure of number of load inputs that are driven by a single output.
- ✓ DC Fan-out of CMOS Output driving CMOS inputs is virtually unlimited because CMOS inputs requires no current either in HIGH state or LOW state.
- ✓ Very Definite limits on Fan-out of TTL or CMOS outputs driving TTL inputs.
- ✓ In TTL if HIGH state and LOW state fan-outs are same then Fan-out is that value.
- ✓ In TTL if HIGH state and LOW state fan-outs are not same, then overall Fan-out is lesser than two fan-outs.

Current Values

- ✓ I_{IH} is the current flowing through an input when it's HIGH and input sink up to 20 □A
- ✓ I_{IL} is the current flowing through an input when it's LOW and sources up to 0.4 mA.
- ✓ I_{OH} is the current flowing through an output when it's HIGH and source up to 400 □A.
- ✓ I_{OL} is the current flowing through an output pin when it's LOW and sink up to 8 mA.

(iv) Un-used inputs:

- ✓ Like CMOS, Unused inputs of TTL may be pulled 'H' or 'L' as per its appropriate logic function.
- ✓ R_{pd}, R_{pu} are more critical for TTL than CMOS because their inputs drive more current especially in LOW state.

Selection of R_{pd}, R_{pu} values :

- ✓ If a Resistor (pull-down) drives 'n' TTL inputs then we must have $n.I_{IL(max)}.R_{pd} < V_{OL(max)}$
- ✓ If a Resistor (pull-up) drives 'n' TTL inputs then we must have $n.I_{IH(max)}.R_{pu} < V_{OH(min)}$

❖ **What is the need of Open collector output? Compare open collector output with Totem-pole or push-pull output.**

- ✓ TTL provide another type of output called as Open-Collector output.
- ✓ The output of two or more different gates with open collector output can be tied together
- ✓ A 2 input NAND gate eliminates pull-up transistor Q4 in Open collector output to take the output from Open collector of transistor Q5 .
- ✓ Collector of Q5 is Open so it will not work properly until we connect an External pull–up Resistor.
- ✓ Open Collector outputs of two or more gates can be Connected together using a connection called "Wired AND Connection"

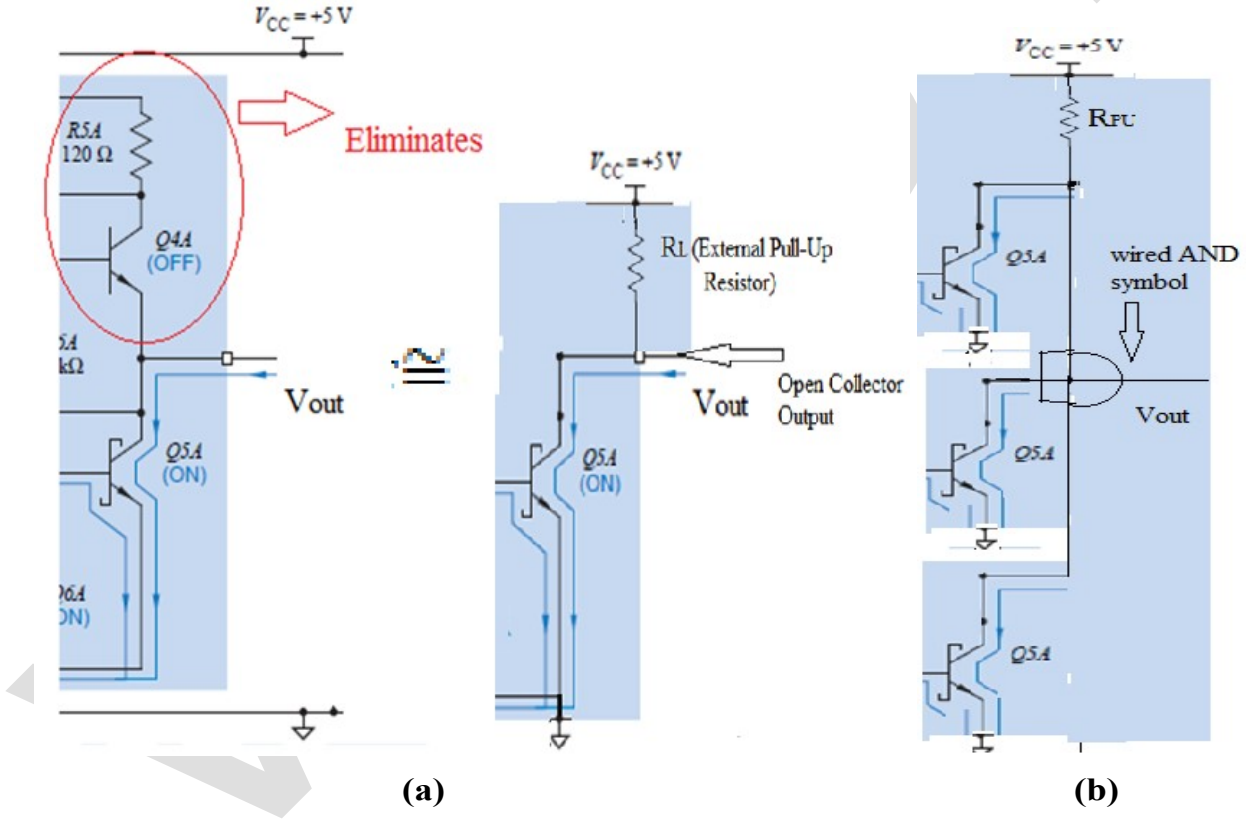


Figure 4.29: (a) Open collector output in TTL family (b) Wired AND Connection

Comparison of open collector output with Push-pull output:

S. No	TOTEM-POLE	Open Collector
1	Output Stage consists of pull-up, pull-down Transistor & Diode resistor	Output Stage consists of only pull-down Transistor.
2	External Pull-up Resistor not Required	External Pull-up Resistor Required for proper operation
3	Outputs of Two gates can't be tied together	Outputs of Two or more gates can tied together using Wired AND Technique
4	Operating Speed is High	Operating Speed is Low

Table 4.16: Comparison between Open Collector output with Push-Pull output

❖ **Write short notes on TTL logic families?**

➢ Transistor-Transistor Logic Families are classified as following:

(a) Early TTL families

- 74L Low power
- 74H High speed

(b) Schottky TTL families

- 74S Schottky
- 74LS Low powerSchottky
- 74AS Advanced Schottky
- 74ALS Advanced Low power Schottky
- 74F Fast

(a) Early TTL families:

✓ The original TTL family of logic gates was introduced by Sylvania in 1963. It was popularized by Texas Instruments, whose "7400-series" part numbers for gates and other TTL components quickly became an industry standard.

✓ As in 7400-series CMOS, devices in a given TTL family have part numbers of the form 74FAM*nn*, where "FAM" is an alphabetic family mnemonic and *nn* is a numeric function designator. Devices in different families with the same value of *nn* perform the same function.

- ✓ In the original TTL family, "FAM" is null and the family is called 74-series TTL.
- ✓ The 74H (High speed TTL) family used lower resistor values to reduce propagation delay at the expense
- ✓ of increased power consumption.
- ✓ The 74L (Low-power TTL) family used higher resistor values to reduce power consumption at the expense of propagation delay.

(b) Schottky TTL families:

- ✓ The first family to make use of Schottky transistors was 74S (Schottky TTL). With Schottky transistors and low resistor values, this family has much higher speed, but higher power consumption, than the original 74-series TTL.
- ✓ 74LS (Low-power Schottky TTL), introduced shortly after 74S. By combining Schottky transistors with higher resistor values, 74LS TTL matches the speed of 74- series TTL but has about one-fifth of its power consumption.
- ✓ Thus, 74LS is a preferred logic family for new TTL designs. Subsequent IC processing and circuit innovations gave rise to two more Schottky logic families as 74AS (Advanced Schottky TTL) and 74ALS (Advanced Low-power Schottky TTL).
- ✓ The 74AS (Advanced Schottky TTL) family offers speeds approximately twice as fast as 74S with approximately the same power consumption.
- ✓ The 74ALS (Advanced Low-power Schottky TTL) family offers both lower power and higher speeds than 74LS, and rivals 74LS in popularity for general-purpose requirements in new TTL designs.
- ✓ The 74F (Fast TTL) family is positioned between 74AS and 74ALS in the speed/power tradeoff, and is probably the most popular choice for high-speed requirements in new TTL designs.

Characteristics of TTL logic families:

- ✓ The important characteristics of TTL families are summarized in Table 4.17.
- ✓ The first two rows of the table list the propagation delay (in nanoseconds) and the power consumption (in milliWatts) of a typical 2-input NAND gate in each family.
- ✓ One figure of merit of a logic family is its *speed-power product* listed in the third row of the table.
- ✓ The remaining rows have values of voltage levels for all TTL families.

	74	74S	74LS	74AS	74ALS	74F
Performance ratings						
Propagation delay (ns)	9	3	9.5	1.7	4	3
Power dissipation (mW)	10	20	2	8	1.2	6
Max. clock rate (MHz)	35	125	45	200	70	100
Fan-out (same series)	10	20	20	40	20	33
Speed-Power product(pj)	90	60	19	13.6	4.8	18
Voltage parameters						
V_{OH}(min) (V)	2.4	2.7	2.7	2.5	2.5	2.5
V_{OL}(max) (V)	0.4	0.5	0.5	0.5	0.5	0.5
V_{IH}(min) (V)	2.0	2.0	2.0	2.0	2.0	2.0
V_{II} (max) (V)	0.8	0.8	0.8	0.8	0.8	0.8

Table 4.17 Characteristics of TTL logic families

❖ **What is interfacing? Explain how to interface CMOS/TTL logic families?**

- ✓ **INTERFACING:** Connecting outputs of one circuit to inputs of another circuit that may have different electrical characteristics.
- ✓ If two circuits that are going to interface have different Electrical characteristics, then direct contact can't be made.
- ✓ In such cases Driver and Load circuits are connected through INTERFACE. Interface circuitry shifts levels of voltage & current for compatibility.
- ✓ Driver output signal must satisfy the requirements of load circuit.
- ✓ If both driver and load require different power supplies, then outputs of both circuits must swing between its specified voltage ranges
- ✓ The interfacing may done in between two different logic families or within the same logic families.
- ✓ Interfacing in between CMOS and TTL logic families is achieved in ways as
  (a) TTL Driving CMOS circuits.
  (b) CMOS driving TTL circuits.

(a) TTL Driving CMOS circuits:

- In this TTL circuit is acts as a driver and CMOS circuit acts as Load circuit.
- These two different family circuits must meet the voltage and current requirements.
- Driver output signal must satisfy the requirements of load circuit.

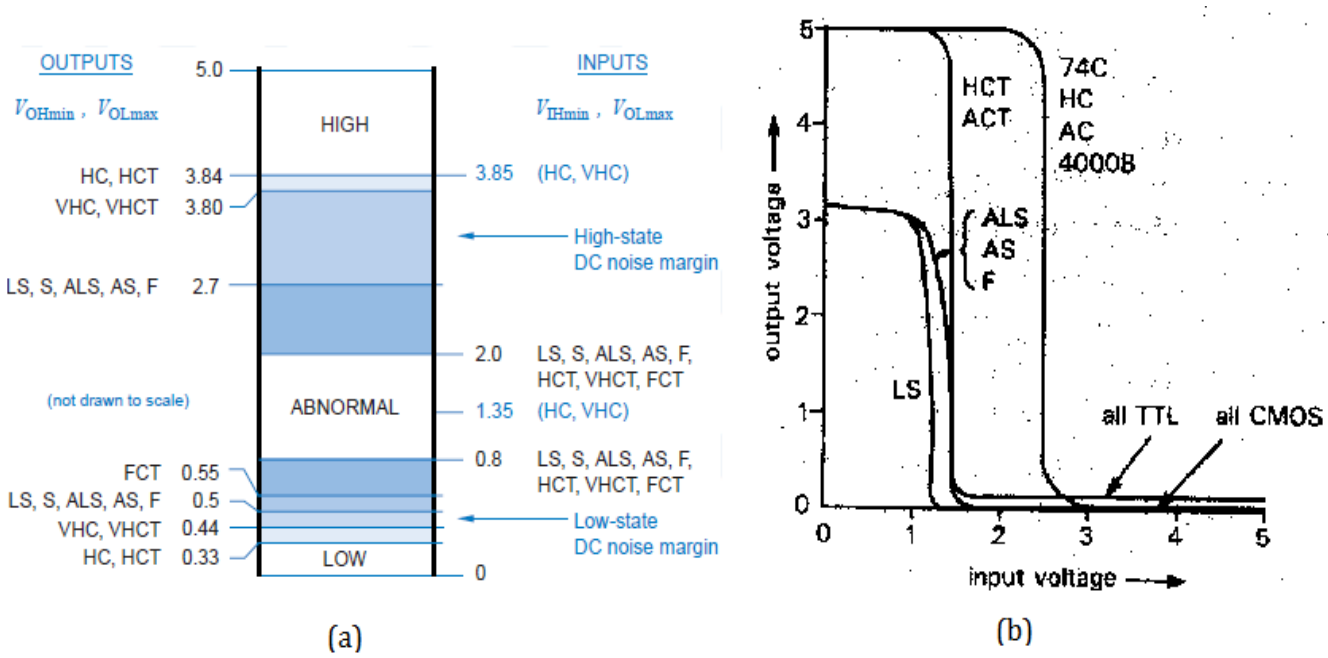


Figure 4.30: (a) Output and input levels for interfacing TTL and CMOS families. (b) TTL/CMOS Transfer Characteristics.

Current Values	CMOS		TTL		
	4000B	74 HC/HCT	74	74LS	74AS
$I_{IH(max)}$	1 µA	1 µA	40µA	20µA	200µA
$I_{IL(max)}$	1 µA	1 µA	1.6mA	0.4mA	2mA
$I_{OH(max)}$	0.4mA	4mA	0.4mA	0.4mA	2mA
$I_{OL(max)}$	0.4mA	4mA	16mA	8mA	20mA

Table 4.18(a) Current comparisons of CMOS and TTL families

Voltage Values	CMOS	TTL		
	74 HC/HCT	74	74LS	74AS
$V_{IH(min)}$	3.85V	2.0	2.0	2.0
$V_{IL(max)}$	1.35V	0.8	0.8	0.8
$V_{OH(min)}$	3.84V	2.4	2.7	2.5
$V_{OL(max)}$	0.33V	0.4	0.5	0.5

Table 4.18(b) Voltage levels comparisons of CMOS and TTL families

- From the above tables $I_{IH(max)}$, $I_{IL(max)}$ values for CMOS are extremely low as compared with $I_{OH(max)}$, $I_{OL(max)}$ values of TTL, So here have the Compatibility.
- But $V_{OH(min)}$ for TTL less than to $V_{IH(min)}$ for CMOS, here voltage requirements are not satisfied, in such cases compatibility can be done by connecting R_{PU}
- R_{PU} raises TTL Output to Approximately 5V in 'HIGH' state.

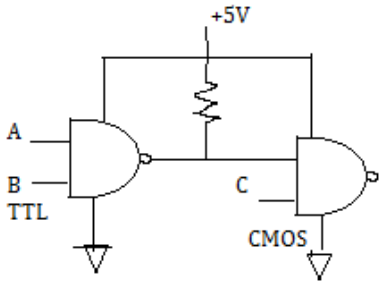


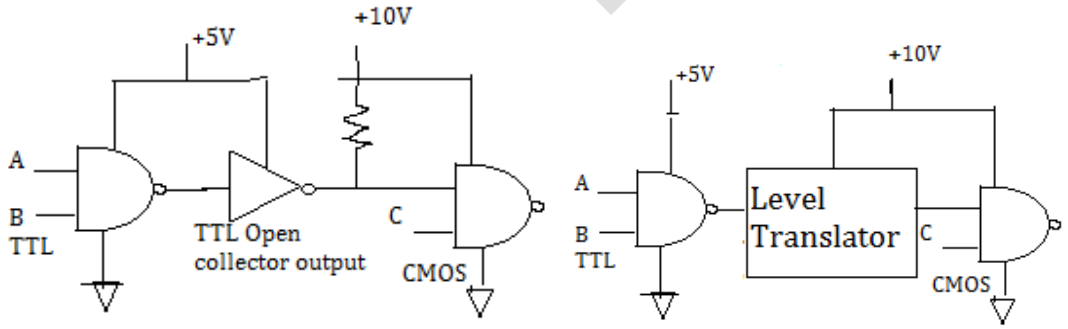
Figure 4.31: Interfacing of TTL NAND gate with CMOS NAND gate

TTL Driving HIGH Voltage CMOS:

- Sometimes we face a difficult situation that CMOS output operated with $V_{DD} > 5v$. Outputs of many TTL families can't operate more than 5v. So we have two alternatives

  (1) Use of Open collector buffer as Interface:

  (2) Level Translator: It shifts the low voltage output from TTL into High voltage input for CMOS



(a) **(b)**

Figure 4.32: (a) Use of Open collector buffer as Interface, (b) Level Translator

(b) CMOS Driving TTL circuits:

- CMOS Driving TTL in HIGH State: To Drive TTL in High State from tables 4.18(a) & 4.18(b) $V_{OH(min)}$ of CMOS, $V_{IH(min)}$ of TTL and $I_{OH(max)}$ of CMOS, $I_{IH(max)}$ of TTL are Satisfied. So no Interface is required.

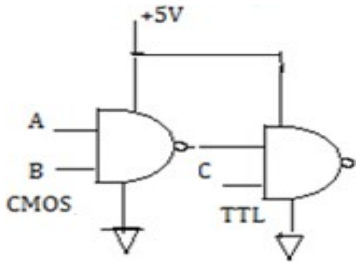


Figure 4.33: (a) CMOS Driving TTL in HIGH State

- CMOS Driving TTL in LOW State: To Drive TTL in Low State from table 4.18(a) $I_{OL(max)}$ of CMOS, $I_{IL(max)}$ of TTL are Satisfied. From table 4.18(b) the $V_{OL(min)}$ of CMOS, $V_{IL(min)}$ of TTL are not satisfied, So interface is needed.

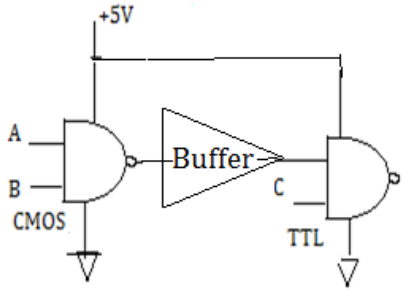


Figure 4.33: (b) CMOS Driving TTL in LOW State

High Voltage CMOS driving TTL: Some Manufacturers provide 74LS TTLS operating at 15v, but many of TTLs can't tolerate >7V. For such cases Voltage level Translators are used.

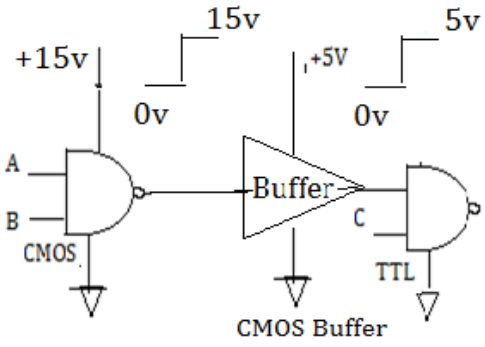


Figure 4.33: (c) High voltage CMOS Driving TTL

❖ **What is interfacing? Explain how to interface Low voltage CMOS interface?**

✓ **INTERFACING:** Connecting outputs of one circuit to inputs of another circuit that may have different electrical characteristics.

- ✓ Two important factors have led the IC industry to move towards lower power supply voltages in CMOS devices.
- ✓ In most applications, CMOS output voltages swing from rail to rail, so the V in the CV^2f equation is the power-supply voltage. Cutting power-supply voltage reduces dynamic power dissipation more than proportionally.
- ✓ As the industry moves towards ever-smaller transistor geometries, the oxide insulation between a CMOS transistor's gate and its source and drain is getting ever thinner, and thus incapable of insulating voltage potentials as "high" as 5V.
- ✓ As a result, JEDEC (Joint Electronic Device Engineering Council) is an IC industry standards group, selected 3.3V ± 0.3V, 2.5V ± 0.2V, and 1.8V± 0.15V as the next "standard" logic power-supply voltages.
- ✓ JEDEC standards specify the input and output logic voltage levels for devices operating with these power-supply voltages.
- ✓ For discrete logic families, the trend has been to produce parts that operate and produce outputs at the lower voltage, but that can also tolerate inputs at the higher voltage.

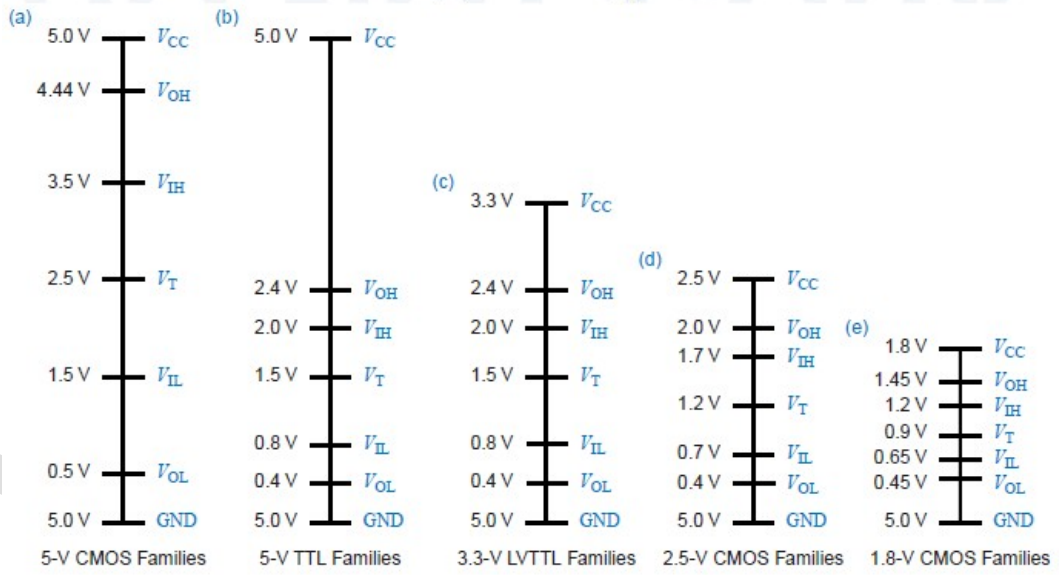


Figure 4.34: Comparisons of logic levels (a) 5V CMOS Families; (b) 5V TTL Families; (c) 3.3V LVTTL Families; (d) 2.5V CMOS Families; (e) 1.8V CMOS Families

- ✓ This approach has allowed 3.3-V CMOS families to operate with 5-V CMOS and TTL families, as we'll see in the next section. Many ASICs and microprocessors have followed a similar approach.
- ✓ A low voltage, such as 2.5 V, is supplied to operate the chip's internal gates, or *core logic*.
- ✓ A higher voltage, such as 3.3 V, is supplied to operate the external input and output circuits, or *pad ring*, for compatibility with older-generation devices in the system.

❖ **What is ECL? Explain the operation of basic ECL circuit?**

- ✓ TTL Family operates the Transistor in deep saturation mode, results the limitation of Switching speed by Storage delay time.
- ✓ To overcome this limitation, another circuit structure is used called Current Mode Logic. This logic family also called as Emitter Coupled Logic.

ECL Characteristics:

- It is fastest logic family because Propagation delay is short.
- For 10k, 100k families it is 1ns,for latest ECL it is 500ps.
- Preventing Transistor from deep saturation by keeping logic levels close to each other.
- Eliminates the storage delays & increases the Switching Speed.
- Noise margin is reduced, it is difficult to achieve good noise immunity.
- Power consumption is more because transistors are not in complete saturation.
- Switching Transients are less, because power supply current is more stable than TTL, CMOS.

Basic ECL Circuit:

- The basic idea of current-mode logic is illustrated by the inverter/buffer circuit in Figure 4.35. This circuit has both an inverting output (OUT1) and a non-inverting output (OUT2).
- This basic ECL circuit contains two output terminals, so it may provide Inverting and Non-Inverting operations in the same circuit.
- So Basic ECL circuit gives Inverter and Buffer operations in the same circuit.

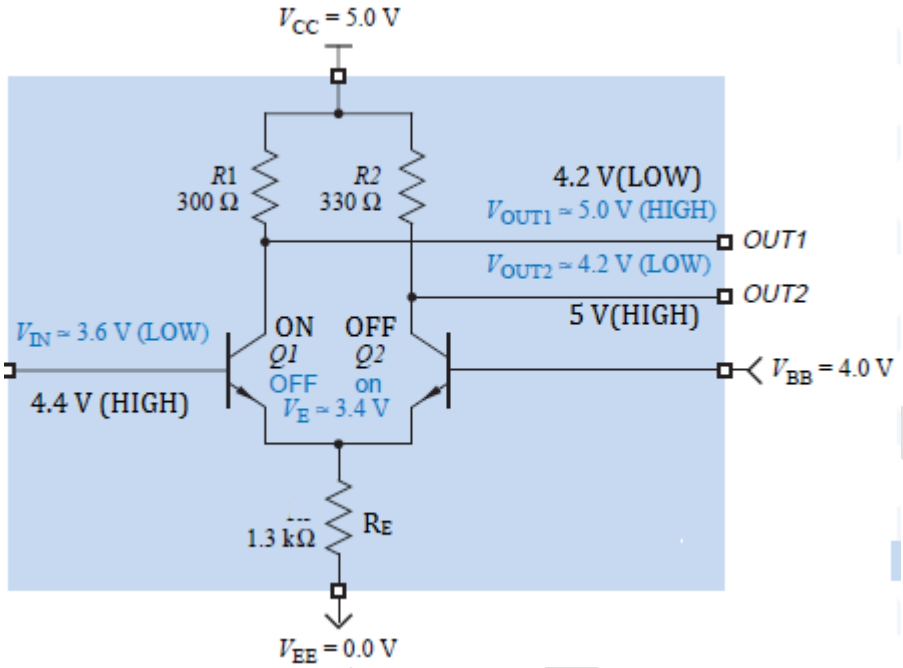


Figure 4.35: Basic ECL circuit.

- Two transistors are connected as a differential amplifier with a common emitter resistor. The supply voltages for this example are V_{CC} = 5V, V_{BB} = 4V, and V_{EE} = 0V, and the input LOW and HIGH levels are defined to be 3.6 and 4.4 V. This circuit actually produces output LOW and HIGH levels that are 0.6 V higher (4.2 and 5.0 V), but this is corrected in real ECL circuits.
- When **V_{IN} is HIGH**, as shown in the figure, transistor Q1 is ON, but not saturated, and transistor Q2 is OFF. Thus, VOUT2 is pulled to 5V (HIGH) through R2, and it can be shown that the voltage drop across R1 is about 0.8 V so that VOUT1 is about 4.2 V (LOW).
- When **V_{IN} is LOW**, as shown in Figure, transistor Q2 is ON, but not saturated, and transistor Q1 is OFF. Thus, VOUT1 is pulled to 5V through R1, and it can be shown that VOUT2 is about 4.2 V.
- The outputs of this inverter are called differential outputs because they are always complementary, and it is possible to determine the output state by looking at the difference between the output voltages (V_{OUT1} - V_{OUT2}) rather than their absolute values.

- That is, the output is 1 if ($V_{OUT1} - V_{OUT2}$) > 0, and it is 0 if ($V_{OUT2} - V_{OUT1}$) > 0. It is possible to build input circuits with two wires per logical input that define the logical signal value in this way; these are called differential inputs

Input	Transistor		Outputs	
V_{IN}	Q1	Q2	V_{OUT1}	V_{OUT2}
LOW	OFF	ON	HIGH	LOW
HIGH	ON	OFF	LOW	HIGH

Table 4.19 Functional table of Basic ECL circuit.

❖ **What is ECL? Explain the operation of ECL OR/NOR circuit?**

ECL/CML LOGIC:

- ✓ TTL Family operates the Transistor in deep saturation mode, results the limitation of switching speed by Storage delay time.
- ✓ To overcome this limitation, another circuit structure is used called Current Mode Logic. This logic family also called as Emitter Coupled Logic.

ECL OR/NOR circuit:

- ✓ To perform logic with the basic ECL circuit is connected with an additional transistors in parallel with *Q1*, similar to the approach in a TTL NOR gate.
- ✓ If any input is HIGH, the corresponding input transistor is active, and V_{OUT1} is LOW (NOR output). At the same time, *Q3* is OFF, and V_{OUT2} is HIGH (OR output).
- ✓ A HIGH output must supply base current to the inputs that it drives, and this current creates an additional voltage drop across R1 or R2, reducing the output voltage (and we don't have much margin to work with).
- ✓ These problems are solved in commercial ECL families, such as the 10K family and 100k family.

INPUTS		TRANSISTORS			OUTPUTS	
X	Y	Q1	Q2	Q3	OUT1	OUT2
0	0	OFF	OFF	ON	1	0
0	1	OFF	ON	OFF	0	1
1	0	ON	OFF	OFF	0	1
1	1	ON	ON	OFF	0	1

Table 4.20: Functional table of ECL 2-input OR/NOR circuit.

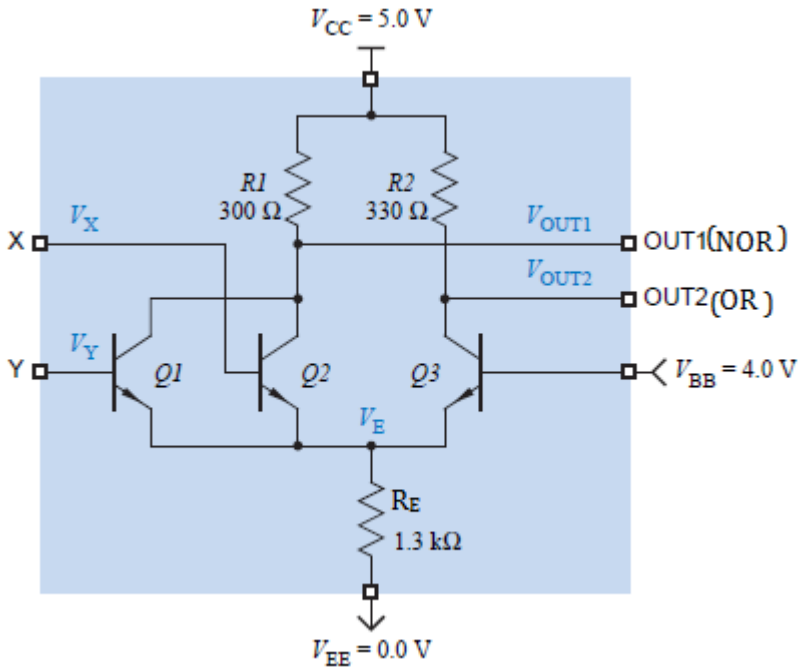


Figure 4.36: ECL 2-input OR/NOR gate circuit.

❖ **What is CML? Explain of ECL 10k and 100k families?**

Ans)

ECL/CML LOGIC:

- ✓ TTL Family operates the Transistor in deep saturation mode, results the limitation of switching speed by Storage delay time.
- ✓ To overcome this limitation, another circuit structure is used called Current Mode Logic. This logic family also called as Emitter Coupled Logic.

ECL 10k Family:

- AnEmitterFolloweroutputstageisaddedtoshifttheO/P,I/Plevels&toprovide very high current driving capacity up to 50mA
- AninternalbiasnetworkisaddedtoprovideV$_{BB}$ Withoutexternalpowersupply
- To improve Noise immunity it is designed with V$_{CC}$ =0v, V$_{EE}$ =-5.2v

Example: 10k ECL 2-INPUT OR/NOR GATE

X	Y	Q1	Q2	Q3	OUT1	OUT2
L	L	OFF	OFF	on	H	L
L	H	OFF	on	OFF	L	H
H	L	on	OFF	OFF	L	H
H	H	on	on	OFF	L	H

Table 4.21: Functional table of 10k ECL 2-INPUT OR/NOR GATE

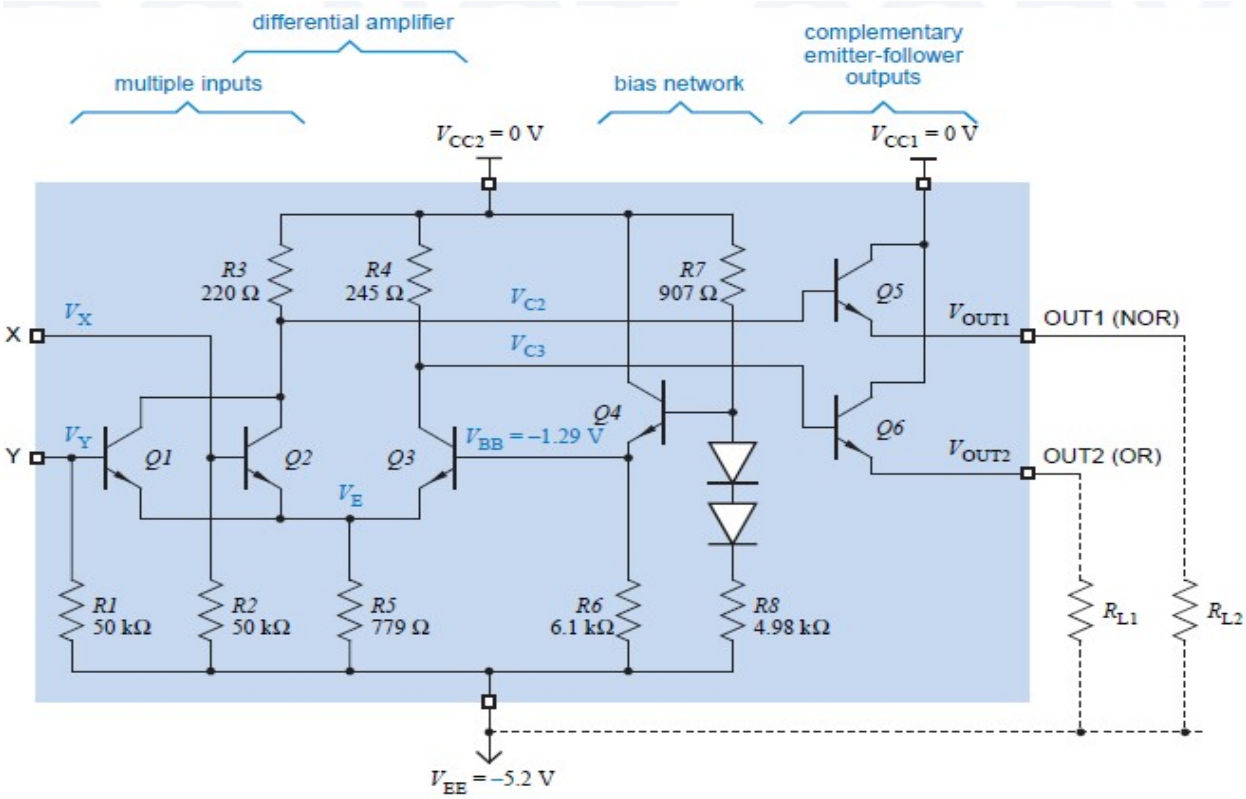


Figure 4.37: 10k ECL family 2-INPUT OR/NOR GATE

ECL 100k Family:

- This family has 100XXX(100101,100107,100170) part numbers.
- This family differs with 10k family in many aspects which are mentioned in table 4.22.

Positive ECL:

- Good noise immunity(resistive) available with V_{EE} =-5.2V,-4.5V.
- But CMOS, TTL Families are operate with +5v supply, to interface them with ECL we need ECL with Positive supply(+5v).
- PECL is used in High Speed and Clock distribution circuits.

PARAMETERS	ECL 10K	ECL 100K
Power Supply Voltage	- 5.2 v	- 4.5v
Propagation delay	2 ns	0.75 ns
Transition times	1.75 ns	0.70 ns
Power Consumption	20 mW	40 mW

Table 4.22 Comparison of ECL 10k family with 100K family

Compare voltage levels of all TTL logic families?

S.No	PARAMETER	CMOS	TTL	ECL
1	Device Used	nMOS, pMOS	BJT	BJT
2	$V_{IH(min)}$	3.5 V	2 V	-1.2 V
	$V_{IL(max)}$	1.5 V	0.8 V	-1.4 V
	$V_{OH(min)}$	4.95 V	2.4 V	-0.9 V
	$V_{OL(max)}$	0.005 V	0.4 V	-1.7 V
3	HIGH level Noise margin	1.45 V	0.4 V	0.3 V
4	LOW level Noise margin	1.45 V	0.4 V	0.3 V
5	Noise immunity	> TTL	< CMOS	More vulnerable to Noise
6	Propagation Delay	70 ns	10 ns	500 ps
7	Switching Speed	< TTL	>> CMOS	Fastest
8	Power Dissipation/Gate	0.1 mW	10 mW	25 mW
9	Speed-Power Product	0.7 pJ	100 pJ	0.5 pJ
10	Fan-out	50	10	25
11	Power Supply Voltage	3-15 V	Fixed 5V	-4.5 V to -5.2 V
12	Power Dissipation	Increase with f	Increase with f	Constant with f
13	Application	Portable Instruments (Battery Supply)	Laboratory Instruments	High Speed Instruments

UNIT-II

The VHDL Hardware Description Language

VHDL stands for **V**HSIC (Very High Speed Integrated Circuits) **H**ardware **D**escription **L**anguage. The VHDL Hardware Design Language In the mid-1980s, the U.S. Department of Defense (DoD) and the IEEE sponsored the development of a highly capable hardware-description language called *VHDL*.

It has become now one of industry's standard languages used to describe digital systems. The other widely used hardware description language is Verilog . Both are powerful languages that allow you to describe and simulate complex digital systems. A third HDL language is ABEL (Advanced Boolean Equation Language) which was specifically designed for Programmable Logic Devices (PLD). ABEL is less powerful than the other two languages and is less popular in industry. This tutorial deals with VHDL, as described by the IEEE standard.

VHDL has the following features:

- Designs may be decomposed hierarchically.
- Each design element has both a well-defined interface (for connecting it to other elements) and a precise behavioral specification (for simulating it).
- Behavioral specifications can use either an algorithm or an actual hardware structure to define an element's operation
- Concurrency, timing, and clocking can all be modeled. VHDL handles asynchronous as well as synchronous sequential- circuit structures.

The logical operation and timing behavior of a design can be simulated.

Design Flow:

There are several steps in a VHDL-based design process, often called the design flow.

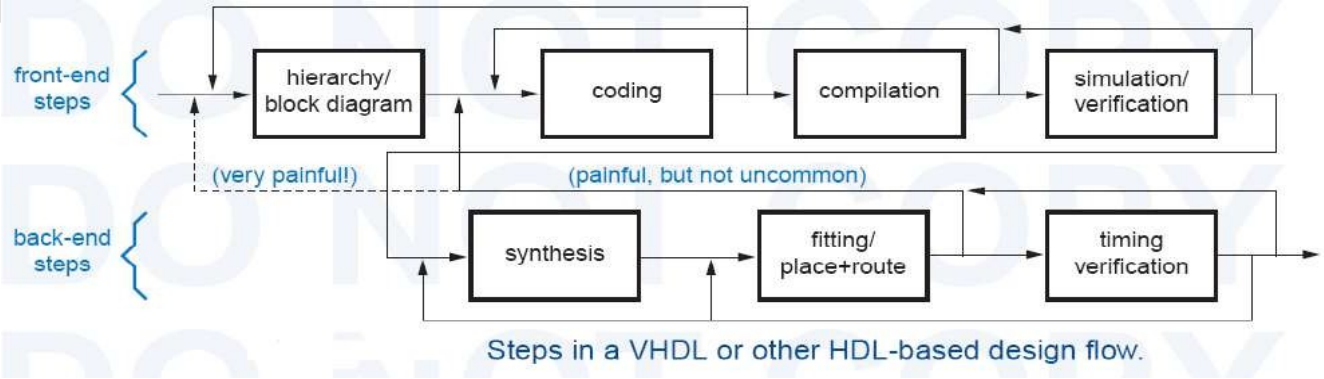


Fig.2.1 The "front end" begins with figuring out the basic approach and building blocks at the block diagram level.

Large logic designs, like software programs, are usually hierarchical, and VHDL gives us a good framework for defining modules and their interfaces, and filling in the details.

Coding:

The actual writing of VHDL code for modules, their interfaces, and their internal details. VHDL is a text-based language, in principle you can use any text editor for this part.

A specialized VHDL text editor include features like automatic highlighting of VHDL keywords, automatic indenting, built-in templates for frequently used program structures, and built-in syntax checking and one-click access to the compiler.

Compiler: A VHDL compiler analyzes your code for syntax errors and also checks your code for compatibility with other modules on which it relies.

It also creates the internal information that is needed for a simulator to process your design later. As in other programming endeavors, you probably shouldn't wait until the very end of coding to compile all of your code. Doing a piece at a time can prevent you from proliferating syntax errors, inconsistent names, and so on,

Simulator:

A VHDL simulator allows you to define and apply inputs to your design, and to observe its outputs, without ever having to build the physical circuit. In small projects we would probably generate inputs and observe outputs manually But for larger projects, VHDL gives you the ability to create "test benches" that automatically apply inputs and compare them with expected outputs.

There are two types of verification

1. Functional Verification
2. Timing Verification

In functional verification, we study the circuit's logical operation independent of timing considerations; gate delays and other timing parameters are considered to be zero.

In timing verification, we study the circuit's operation including estimated delays, and we verify that the setup, hold, and other timing requirements for sequential devices like flip-flops are met. Functional verification before starting the back-end steps.

In back-end there are three basic steps

1) Synthesis 2)Fitting/Place + Route 3)Timing Verification Synthesis:

The synthesis, converting the VHDL description into a set of primitives or components that can be assembled in the target technology.

For example, with PLDs or CPLDs, the synthesis tool may generate two-level sum-of-products equations.

In the fitting step, a fitting tool or fitter maps the synthesized primitives or components onto available device resources.

Place & Route:

For a PLD or CPLD, this may mean assigning equations to available AND-OR elements. For an ASIC, it may mean laying down individual gates in a pattern and finding ways to connect them within the physical constraints of the ASIC die.

The "final" step is timing verification of the fitted circuit. actual circuit delays due to wire lengths, electrical loading, and other factors can be calculated with reasonable precision.

VHDL Program Structure

A VHDL *entity* is simply a declaration of a module's inputs and outputs, while a VHDL *architecture* is a detailed description of the module's internal structure or behavior.

In the text file of a VHDL program, the entity declaration and architecture definition are separated shown in fig.2.2

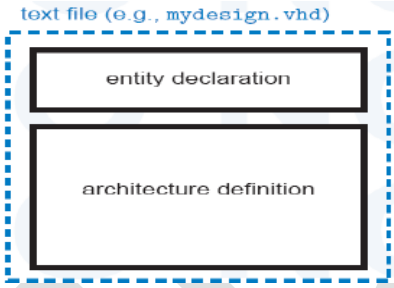


Figure 2.2 VHDL program file structure.

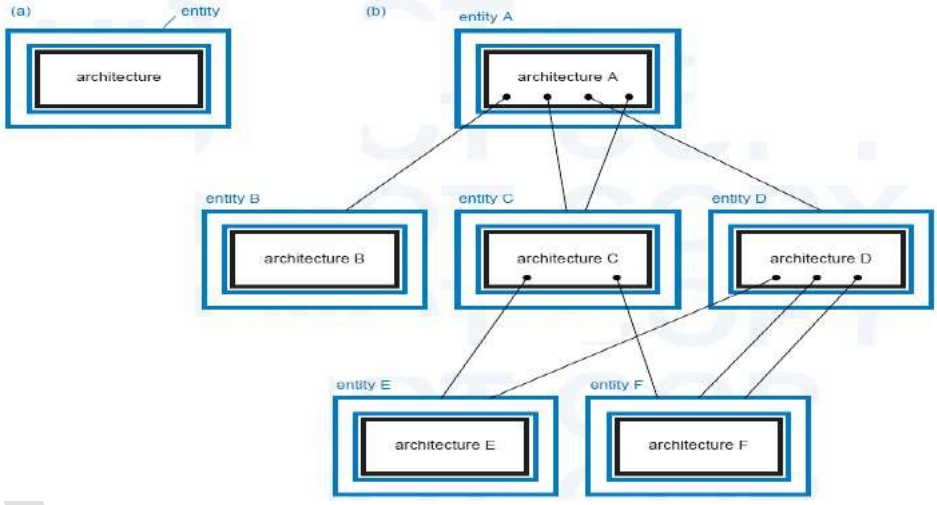


Figure 2.3 (a) illustrates the concept. Many designers like to think of a VHDL entity declaration as a "wrapper" for the

architecture, hiding the details of what's inside while providing the "hooks" for other modules to use it.

This forms the basis for hierarchical system design—the architecture of a top-level entity may use (or "instantiate") other entities, while hiding the architectural details of lower-level entities from the higher-level ones. As shown in (b), a higher- level architecture may use a lower-level entity multiple times, and multiple top-level architectures may use the same lower-level one. In the figure, architectures B, E and F stand alone; they do not use any other entities.

Like other high-level programming languages, VHDL generally ignores spaces and line breaks, and these may be provided as desired for readability.

Comments begin with two hyphens (--) and end at the end of a line.

Syntax of a VHDL entity declaration.

entity entity-name is

port (signal-names : mode signal-type; signal-names : mode signal-type;

...

signal-names : mode signal-type); end entity-name;

An entity declaration has shown in the general syntax. The entity declaration is to define its external interface signals or *ports* in its *port declaration* part. In addition to the keywords ***entity, is, port,*** and ***end***,

An entity declaration has the following elements:

entity-name A user-selected identifier to name the entity.

signal-names A comma-separated list of one or more user-selected identifiers to name external-interface signals.

mode One of four reserved words, specifying the signal direction:

in The signal is an input to the entity.

out The signal is an output of the entity. Note that the value of such a signal cannot be "read" inside the entity's architecture, only by other entities that use it.

buffer The signal is an output of the entity, and its value can also be read inside the entity's architecture.

inout The signal can be used as an input or an output of the entity. This mode is typically used for three-state input/output pins on PLDs.

signal-type A built-in or user-defined signal type.

Note that there is no semicolon after the final ***signal-type***; swapping the closing parenthesis with the semicolon after it is a common syntax error for beginning VHDL programmers.

- *type*: a built-in or user-defined signal type. Examples of types are bit, bit_vector, Boolean, character, std_logic, and std_ulogic.
  - *bit* – can have the value 0 and 1
  - *bit_vector* – is a vector of bit values (e.g. bit_vector (0 to 7)
  - *std_logic, std_ulogic, std_logic_vector, std_ulogic_vector*: can have 9 values to indicate the value and strength of a signal. Std_ulogic and std_logic are preferred over the bit or bit_vector types.
  - *boolean* – can have the value TRUE and FALSE.
  - *integer* – can have a range of integer values.
  - *real* – can have a range of real values.
  - *character* – any printing character.
  - *time* – to indicate time.

Entity Examples:

An example of the entity declaration of a Four-to-one multiplexer of which each input is an 8-bit word.

entity mux4_to_1 **is**

port (I0,I1,I2,I3: **in** std_logic_vector(7 **downto** 0); SEL: **in** std_logic_vector (1 **downto** 0);

OUT1: **out** std_logic_vector(7 **downto** 0));

end mux4_to_1;

An example of the entity declaration of a D flip-flop with set and reset inputs is

entity dff_sr **is**

port (D,CLK,S,R: **in** std_logic; Q,Qnot: **out** std_logic);

end dff_sr;

Architecture body

The architecture body specifies how the circuit operates and how it is implemented. An entity or circuit can be specified in a variety of ways, such as behavioral, structural (interconnected components), or a combination of the above. The *entity-name* in this definition must be the same as the one given previously in the entity declaration.

The *architecture-name* is a user-selected identifier, usually related to the entity name; it can be the same as the entity name if desired.

The architecture body looks as follows,

architecture architecture_name **of** NAME_OF_ENTITY **is**

-- Declarations

-- components declarations

-- signal declarations

-- constant declarations

-- function declarations

-- procedure declarations

-- type declarations

begin

-- Statements

end architecture_name; Behavioral model

The architecture body for the example of Figure 2, described at the behavioral level, is given below,

architecture behavioral **of** BUZZER **is begin**

WARNING <= (**not** DOOR **and** IGNITION) **or** (**not** SBELT **and** IGNITION);

end behavioral;

The header line of the architecture body defines the architecture name, e.g. behavioral, and associates it with the

entity, BUZZER. The architecture name can be any legal identifier. The main body of the architecture starts with the keyword **begin** and gives the Boolean expression of the function. We will see later that a behavioral model can be described in several other ways. The "<= " symbol represents an assignment operator and assigns the value of the expression on the right to the signal on the left. The architecture body ends with an **end** keyword followed by the architecture name.

A few other examples follow. The behavioral description of a two-input AND gate is shown below.

Architecture *architecture-name* of *entity-name* is

type declarations signal declarations constant declarations function definitions procedure definitions

component declarations

begin

concurrent-statement

...

concurrent-statement

End *architecture-name*;

The *signal declaration* gives the same information about a signal as in a port declaration, except that no mode is specified:

signal signal-names : signal-type;

VHDL *variables* are similar to signals, except that they usually don't have physical significance in a circuit.

The syntax of a *variable declaration* is just like that of a signal declaration, except that the variable keyword is

Variable *variable-names* : *variable-type*;

Types and Constants

All signals, variables, and constants in a VHDL program must have an associated "type". The *type* specifies the set or range of values that the object can take on,

VHDL predefined types

bit	character	severity_level
bit_vector	integer	string
Boolean	real	time

Predefined operators for VHDL's integer and boolean types.

integer Operators		boolean Operators
+	addition	and AND

-	subtraction	or OR
*	multiplication	Nand NAND
/	division	nor NOR
Mod	modulo division	xor Exclusive OR
rem	modulo remainder	xnor Exclusive NOR
abs **	absolute value exponentiation	not complementation

Type integer is defined as the range of integers including at least the range –2,147,483,647 through +2,147,483,647 (–231+1 through +231–1).

Type boolean has two values, true and false. The character type contains all of the characters in the ISO 8-bit character set; the first 128 characters are the ASCII characters.

The most commonly used types in typical VHDL programs are *userdefined types*, and the most common of these are *enumerated types*, which are defined by listing their values.

value-list is a comma-separated list (enumeration) of all possible values of the type. The values may be user-defined identifiers or characters.

type *type-name* is (*value-list*);

subtype *subtype-name* is *type-name start* to *end*;

subtype *subtype-name* is *type-name start* downto *end*;

constant *constant-name*: *type-name* := *value*;

type traffic_light_state is (reset, stop, wait, go); A standard user-defined logic type std_logic,

Definition of VHDL std_logic type

type STD_ULOGIC is ('U', -- Uninitialized
'X', -- Forcing Unknown '0', -- Forcing 0
'1', -- Forcing 1
'Z', -- High Impedance 'W', -- Weak Unknown 'L', -- Weak 0
'H', -- Weak 1
'-' -- Don't care
);

subtype STD_LOGIC is resolved STD_ULOGIC; type traffic_light_state is (reset, stop, wait, go);

VHDL also allows users to create *subtypes* of a type. The values in the subtype must be a contiguous range of values of the base type, from *start* to *end*.

subtype twoval_logic is std_logic range '0' to '1'; subtype fourval_logic is std_logic range 'X' to 'Z'; subtype negint is integer range -2147483647 to 1; subtype bitnum is integer range 31 downto 0;

Syntax of VHDL array declarations

type *type-name* is array (*start* to *end*) of *element-type*;

type *type-name* is array (*start* downto *end*) of *element-type*;

type *type-name* is array (*range-type*) of *element-type*;

type *type-name* is array (*range-type* range *start* to *end*) of *element-type*;

type *type-name* is array (*range-type* range *start* downto *end*) of *element-type*;

constant BUS_SIZE: integer := 32; -- width of component constant MSB: integer := BUS_SIZE-1; -- bit number of MSB constant Z: character := 'Z'; -- synonym for Hi-Z value

VHDL defines an *array* as an ordered set of elements of the same type, where each element is selected by an *array index*. Examples of VHDL array declarations.

type monthly_count is array (1 to 12) of integer; type byte is array (7 downto 0) of STD_LOGIC;

constant WORD_LEN: integer := 32;

type word is array (WORD_LEN-1 downto 0) of STD_LOGIC;

constant NUM_REGS: integer := 8;

type reg_file is array (1 to NUM_REGS) of word;

type statecount is array (traffic_light_state) of integer;

A VHDL *string* is a sequence of ISO characters enclosed in double quotes, such as "Hi there". A string is just an array of characters.

B := "11111111";

W := "11111110111111101111111011111110";

Libraries and Packages:

A VHDL *library* is a place where the VHDL compiler stores information about a particular design project, including intermediate files that are used in the analysis, simulation, and synthesis of the design.

The location of the library within a host computer's file system is implementation dependent. For a given VHDL design, the compiler automatically creates and uses a library named "work".

A complete VHDL design usually has multiple files, each containing different design units including entities and architectures.

When the VHDL compiler analyzes the each file in the design, it places the results in the "work" library, and it also searches this library for needed definitions, such as other entities.

Not all of the information needed in a design may be in the "work" library.

Each project has its own "work" library (typically a subdirectory within that project's overall directory), but must

also refer to a common library containing the shared definitions.

Even small projects may use a standard library such as the one containing IEEE standard definitions.

The designer can specify the name of such a library using a library *clause* at the beginning of the design file. For example, we can specify the IEEE library:

library ieee;

The clause "library work;" is included implicitly at the beginning of every VHDL design file.

Specifying a library name in a design gives it access to any previously analyzed entities and architectures stored in the library, but it does not give access to type definitions and the like. This is the function of "packages" and "use clauses," described next.

Packages:

A VHDL *package* is a file containing definitions of objects that can be used in other programs. The kind of objects that can be put into a package include signal, type, constant, function, procedure, and component declarations.

Signals that are defined in a package are "global" signals, available to any VHDL entity that uses the package.

Types and constants defined in a package are known in any file that uses the package. Likewise, functions and procedures defined in a package can be called in files that use the package, and components (described in the next subsection) can be "instantiated" in architectures that use the package.

A design can "use" a package by including a use *clause* at the beginning of the design file. For example, to use all of the definitions in the IEEE standard

1164 package, we would write

use ieee.std_logic_1164.all;

Here, "ieee" is the name of a library which has been previously given in a library clause. Within this library, the file named "std_logic_1164" contains the desired definitions. The suffix "all" tells the compiler to use all of the definitions in this file. Instead of "all", you can write the name of a particular object to use just its definition, for example,

use ieee.std_logic_1164.std_ulogic

This clause would make available just the definition of the std_ulogic type, without all of the related types and functions. However, multiple "use" clauses can be written to use additional definitions.

Syntax of a VHDL package definition:

package *package-name* is

type declarations signal declarations constant declarations

component declarations function declarations procedure declarations

end *package-name*;

package body *package-name* is

type declarations constant declarations function definitions procedure definitions

end *package-name*;

Structural Design Elements

In VHDL, each *concurrent statement* executes simultaneously with the other concurrent statements in the Same architecture body.

The most basic of VHDL's concurrent statements is the component *statement*, whose basic syntax is shown in Table 3.1. Here, *component-name* is the name of a previously defined entity that is to be used, or *instantiated*, within the current architecture body.

Syntax of a VHDL component statement

***label*: *component-name* port map(*signal1*, *signal2*, ..., *signaln*);**

***label*: *component-name* port map(*port1*=>*signal1*, *port2*=>*signal2*, ..., *portn*=>*signaln*);**

The port map keywords introduce a list that associates ports of the named entity with signals in the current architecture. The list may be written in either of two different styles. The first is a positional style; as in conventional programming languages, the signals in the list are associated with the entity's ports in the same order that they appear in the entity's definition.

The second is an explicit style; each of the entity's ports is connected to a signal using the "=>" operator, and these associations may be listed in any order.

Syntax of a VHDL component declaration.

component *component-name*
port (*signal-names* : *mode signal-type*;
***signal-names* : *mode signal-type*;**
...
***signal-names* : *mode signal-type*);**
end component;

A component must be declared in a *component declaration* in the architecture's definition. In above syntax, a component declaration is essentially the same as the port declaration part of the corresponding entity declaration—it lists the name, mode, and type of each of its ports.

The components used in an architecture may be ones that were previously defined as part of a design, or they may be part of a library.

A VHDL architecture that uses components is often called a *structural description* or *structural design*, because it defines the precise interconnection structure of signals and entities that realize the entity.

Example Structural VHDL program for a prime-number detector.

```
library IEEE;
use IEEE.std_logic_1164.all; entity prime is
port ( N: in STD_LOGIC_VECTOR (3 downto 0); F: out STD_LOGIC );
end prime;
```

```
architecture prime1_arch of prime is
signal N3_L, N2_L, N1_L: STD_LOGIC;
signal N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0: STD_LOGIC;
component INV port (I: in STD_LOGIC; O: out STD_LOGIC); end component;
component AND2 port (I0,I1: in STD_LOGIC; O: out STD_LOGIC); end component;
component AND3 port (I0,I1,I2: in STD_LOGIC; O: out STD_LOGIC); end component;
component OR4 port (I0,I1,I2,I3: in STD_LOGIC; O: out STD_LOGIC); end component;
begin
U1: INV port map (N(3), N3_L);
U2: INV port map (N(2), N2_L);
U3: INV port map (N(1), N1_L);
U4: AND2 port map (N3_L, N(0), N3L_N0);
U5: AND3 port map (N3_L, N2_L, N(1), N3L_N2L_N1);
U6: AND3 port map (N2_L, N(1), N(0), N2L_N1_N0);
U7: AND3 port map (N(2), N1_L, N(0), N2_N1L_N0);
U8: OR4 port map (N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0, F);
end                                                          prime1_arch;
```

UNIT- III
COMBINATIONAL LOGIC

❖ **Define combinational logic circuit and explain its analysis and design procedure?**

Ans) Combinational Logic Circuit: Combinational logic circuits are circuits which has connection of logic gates together to produce an output at any time depends upon the combination of input signals present at that instant only, and does not depend on any past conditions. So no Memory element is required in Combinational Logic Circuits.

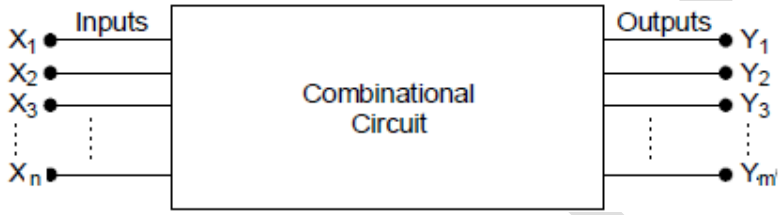


Figure 5.1 Block diagram of Combinational Logic

Circuit Combinational Logic Circuit Analysis Procedure:

- Analysis of a Combinational logic circuit is the procedure by which we can determine the function that the circuit can implement.
- In this procedure we have to obtain possible boolean expressions and truth tables of output functions from the give given circuit.

STEPS to follow in Analysis Procedure:

1. First check whether the given circuit is combinational or sequential circuit.
2. Assign labels to all gate outputs with arbitrary symbols and determine boolean functions for each gate output until to get last output.
3. Substitute all output functions to get final output.
4. Find the truth table for the above mentioned values by taking binary numbers from 0 to 2^n-1 for 'n' number of values.

Example:

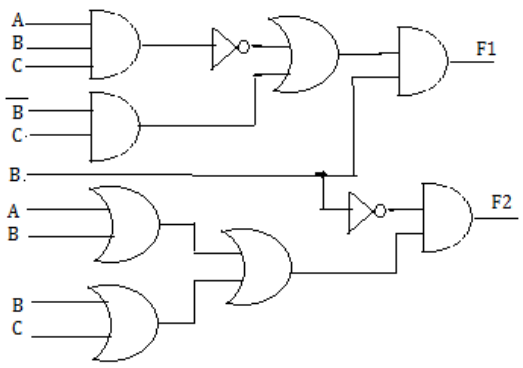


Figure 5.2 A combinational logic circuit example

Analysis of Figure 5.1:

Step-1: The given circuit has output dependency only on present inputs. So given circuit is combinational logic circuit.

Step-2: Assigning labels to all gate outputs in each stage.

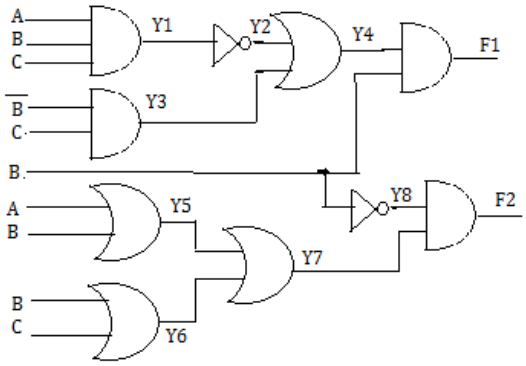


Figure 5.2 Assigning labels to all gate outputs

Step-3: Obtaining Boolean expressions for all output stages.

$Y=\overline{AB+CD}$ Y1= A.B.C; $Y2=\overline{Y1}=\overline{A.B.C}$; $Y3=\bar{B}.C$

$Y4=Y2+Y3=\overline{A.B.C}+\bar{B}.C=\bar{A}+\bar{B}+\bar{C}+\bar{B}.C=\bar{A}+\bar{B}(1+\bar{C})+\bar{C}==\bar{A}+\bar{B}+\bar{C}=\overline{A.B.C}$

Y5=A+B; Y6=B+C; Y7=Y5+Y6=A+B+C; $Y8=\bar{B}$;

The final outputs, $F1=B.Y4=B.(\overline{A.B.C})=B(\bar{A}+\bar{B}+\bar{C})=B.A+B.\bar{C}$

$F2=Y7.Y8=(A+B+C)\,\bar{B}=A\bar{B}+\bar{B}C$

Inputs			Each stage outputs								Output	
A	B	C	Y1	Y2	Y3	Y4	Y5	Y6	Y7	Y8	F1	F2
0	0	0	0	0	0	0	0	0	1	1	0	0
0	0	1	0	1	0	1	1	0	1	1	0	1
0	1	0	0	0	1	1	1	1	1	0	1	0
0	1	1	0	0	1	1	1	1	1	0	1	0
1	0	0	0	0	1	0	1	1	1	1	0	1
1	0	1	0	1	1	1	1	1	1	1	0	1
1	1	0	0	0	1	1	1	1	1	0	1	0
1	1	1	1	0	1	1	0	1	0	0	0	0

❖ **Combinational Logic Circuit Design Procedure:**

STEPS to follow in Design Procedure:

1. Problem definition and Determining number of input and output variables
2. Assign Letter symbols to input and output variables.
3. Obtain truth table indicating relation between inputs and outputs.
4. Obtain simplified boolean expressions each output using k-map.
5. By using above obtained boolean expressions draw logic diagram.

Example: Design a combinational logic circuit with 3 input variables that will produce output as logic '1' if and only if any two inputs have logic '1'.

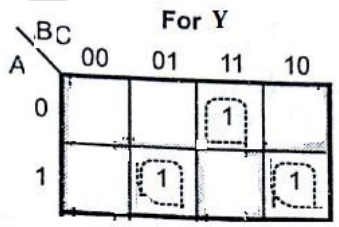
Step-1 & 2: From the given information let inputs are A, B, C. and Output as Y.

Step-3: Obtaining truth table

Inputs			Output
A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Table 5.2: Truth table

Step-4: Obtaining simplified boolean expression for output Y using k-map.



Step-5: Draw Logic Diagram

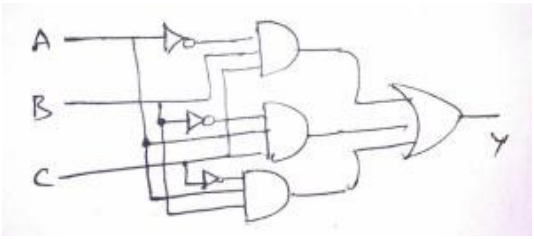


Figure 5.2 Assigning labels to all gate outputs

❖ **Define Decoder? Explain the operation of a binary decoder using example?**

Decoder: A Decoder is a multiple input and multiple output logic circuit which converts coded inputs into coded outputs. Each input codeword produces a different output codeword. There is one to one mapping from input codeword into output codeword.

Binary Decoder: A decoder is a combinational logic circuit which has n-bit binary input code and one activated output out of 2^n output codes called as Binary decoder.

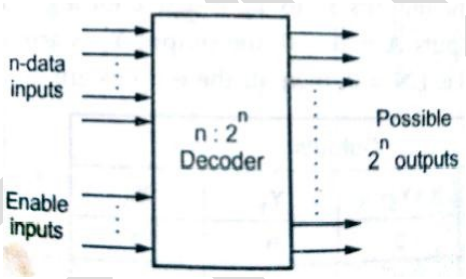


Figure 5.3 Binary

Decoder Example: Designing of 2 to 4 binary decoder

A 2:4 Decoder contains two inputs and produces four outputs, to activate an enable input is applied to this decoder.

Enable Input	Inputs		Outputs			
En	A	B	Y0	Y1	Y2	Y3
0	X	X	0	0	0	0
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

Table 5.3 Truth table for 2:4 Decoder

From the above truth table the outputs are obtained as

$Y0 = \bar{A}.\bar{B}$, $Y1 = \bar{A}.B$, $Y2 = A.\bar{B}$, $Y3 = A.B$

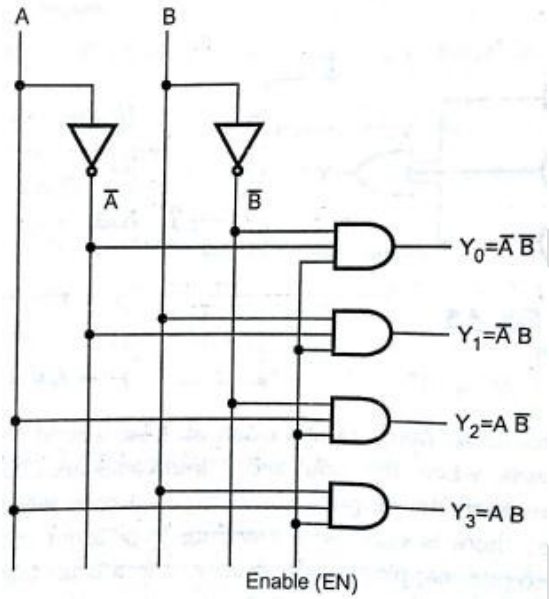


Figure 5.4: Logic diagram of 2:4

Decoder Example: Designing of 3 to 8 binary decoder

A 3:8 Decoder contains two inputs and produces four outputs, to activate an enable input is applied to this decoder.

Enable Input	Inputs			Outputs							
En	A	B	C	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
0	X	X	X	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	0	0	0
1	0	0	1	0	1	0	0	0	0	0	0
1	0	1	0	0	0	1	0	0	0	0	0
1	0	1	1	0	0	0	1	0	0	0	0
1	1	0	0	0	0	0	0	1	0	0	0
1	1	0	1	0	0	0	0	0	1	0	0
1	1	1	0	0	0	0	0	0	0	1	0
1	1	1	1	0	0	0	0	0	0	0	1

Table 5.4 Truth table for 3:8 Decoder

From the above truth table the outputs are obtained as

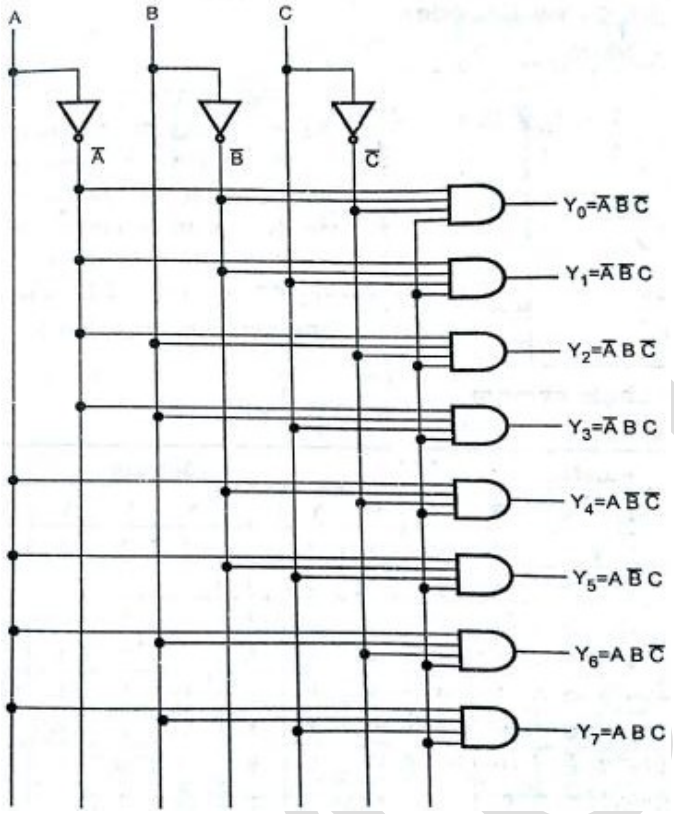


Figure 5.5: Logic diagram of 3:8 Decoder

❖ **Explain the operation of a binary decoder using IC 74X138?**

Decoder: A Decoder is a multiple input and multiple output logic circuit which converts coded inputs into coded outputs. Each input codeword produces a different output codeword. There is one to one mapping from input codeword into output codeword.

Binary Decoder: A decoder is a combinational logic circuit which has n-bit binary input code and one activated output out of 2^n output codes called as Binary decoder.

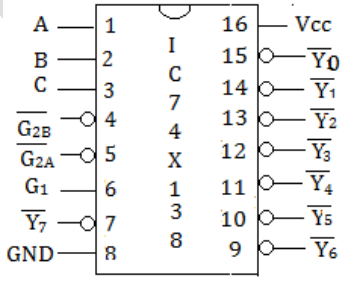


Figure 5.6: Pin diagram of IC 74X138

- ✓ The IC 74X138 is used to produce 3: 8 Decoder operation.
- ✓ The IC 74X138 is 16 pin Dual In Packaged(DIP) IC. It has three inputs A, B and C.

- ✓ The IC 74X138 has three enable inputs as G_1, G_{2A} and G_{2B}. Among these three enable inputs, G_1 is active high and G_{2A}, G_{2B} are active low.
- ✓ It produces Eight active low outputs as Y_0, Y_1, Y_2, Y_3, Y_4, Y_5, Y_6 and Y_7.

Enable Inputs			Inputs			Outputs(Active low)							
G_1	$\overline{G_{2A}}$	$\overline{G_{2B}}$	A	B	C	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
0	X	X	X	X	X	1	1	1	1	1	1	1	1
X	1	X	X	X	X	1	1	1	1	1	1	1	1
X	X	1	X	X	X	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	1	1	1	1	1	1	1
1	0	0	0	0	1	1	0	1	1	1	1	1	1
1	0	0	0	1	0	1	1	0	1	1	1	1	1
1	0	0	0	1	1	1	1	1	0	1	1	1	1
1	0	0	1	0	0	1	1	1	1	0	1	1	1
1	0	0	1	0	1	1	1	1	1	1	0	1	1
1	0	0	1	1	0	1	1	1	1	1	1	0	1
1	0	0	1	1	1	1	1	1	1	1	1	1	0

Table 5.5: Truth table for IC 74X138

❖ **Explain the operation of a binary decoder using IC 74X139?**

Decoder: A Decoder is a multiple input and multiple output logic circuit which converts coded inputs into coded outputs. Each input codeword produces a different output codeword. There is one to one mapping from input codeword into output codeword. **Binary Decoder:** A decoder is a combinational logic circuit which has n-bit binary input code and one activated output out of 2^n output codes called as Binary decoder.

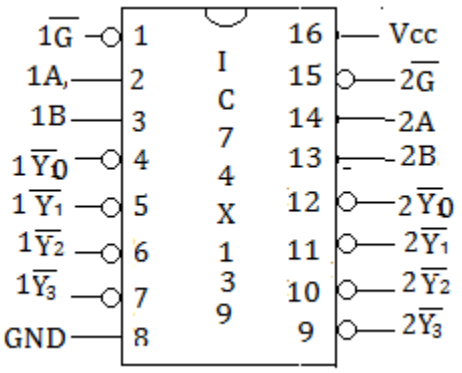


Figure 5.6: Pin diagram of IC 74X139

- ✓ The IC 74X139 is a Dual 2 to 4 Decoder.
- ✓ The IC 74X139 is 16 pin Dual In Packaged(DIP) IC. It has two sets of inputs 1A,1B and 2A, 2B.

- ✓ The IC 74X139 has two enable inputs as $1\overline{G}$, $2\overline{G}$ both are active low for two decoders separately.
- ✓ It produces four active low outputs as $1Y_0$, $1Y_1$, $1Y_2$, $1Y_3$, $1Y_4$ for one half of the dual 2 to 4 decoder and four active low outputs as $2Y_0$, $2Y_1$, $2Y_2$, $2Y_3$, $2Y_4$ for another half of the dual 2 to 4 decoder.
- ✓ For active low enable input only the IC 74X139 may produce valid outputs.

Enable Inputs		Inputs				Outputs(Active low)							
$1\overline{G}$	$2\overline{G}$	1A	1B	2A	2B	$1Y_0$	$1Y_1$	$1Y_2$	$1Y_3$	$2Y_0$	$2Y_1$	$2Y_2$	$2Y_3$
1	1	X	X	X	X	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	1	1	1	0	1	1	1
		0	1	0	1	1	0	1	1	1	0	1	1
		1	0	1	0	1	1	0	1	1	1	0	1
		1	1	1	1	1	1	1	0	1	1	1	0
0	1	0	0	0	0	0	1	1	1	1	1	1	1
		0	1	0	1	1	0	1	1	1	1	1	1
		1	0	1	0	1	1	0	1	1	1	1	1
		1	1	1	1	1	1	1	0	1	1	1	1
1	0	0	0	0	0	1	1	1	1	0	1	1	1
		0	1	0	1	1	1	1	1	1	0	1	1
		1	0	1	0	1	1	1	1	1	1	0	1
		1	1	1	1	1	1	1	1	1	1	1	0
1	1	0	0	0	0	1	1	1	1	1	1	1	1
		0	1	0	1	1	1	1	1	1	1	1	1
		1	0	1	0	1	1	1	1	1	1	1	1
		1	1	1	1	1	1	1	1	1	1	1	1

Table 5.6(a): Truth table for IC 74X139

Enable Inputs	Inputs		Outputs(Active low)			
$1\overline{G}$	1A	1B	$1Y_0$	$1Y_1$	$1Y_2$	$1Y_3$
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

Table 5.6(b): Truth table for IC 74X139(one half only)

❖ **What is the need for cascading in decoders? Implement the 4:16 binary decoder using required number of IC 74X138?**

The implementation of ICs for higher order decoder requires a large number of AND

gates which may leads to the increase in Area occupancy, power consumption and design complexity. So no separate IC is implemented for such kind of higher order decoders. These higher order decoders are designed by using cascading of lower order decoders. Hence cascading of decoders is used in the implementation of higher order decoders.

Implement the 4:16 binary decoder using IC 74X138

The IC 74X138 is used to produce 3: 8 Decoder operation. This implementation requires two IC 74X138 to get specified 4 inputs and 16 outputs. Each IC 74X138 has three inputs and 3 enable inputs as G_1, G_{2A} and G_{2B}. Among these three enable inputs, G_1 is active high and G_{2A}, G_{2B} are active low. It produces Eight active low outputs as Y_0, Y_1, Y_2, Y_3, Y_4, Y_5, Y_6 and Y_7. To get required specifications all three inputs of both IC 74X138 are shorted together with their respective input terminals. The 4^{th} required input, let as 'D' is taken from Enable inputs as shown in the figure 5.7. When D=logic '0' then first IC gets enabled or activated and gives 8 outputs of first IC74LS138(1) and When D=logic '1' then second IC gets enabled or activated and gives 8 outputs of first IC74LS138(2)

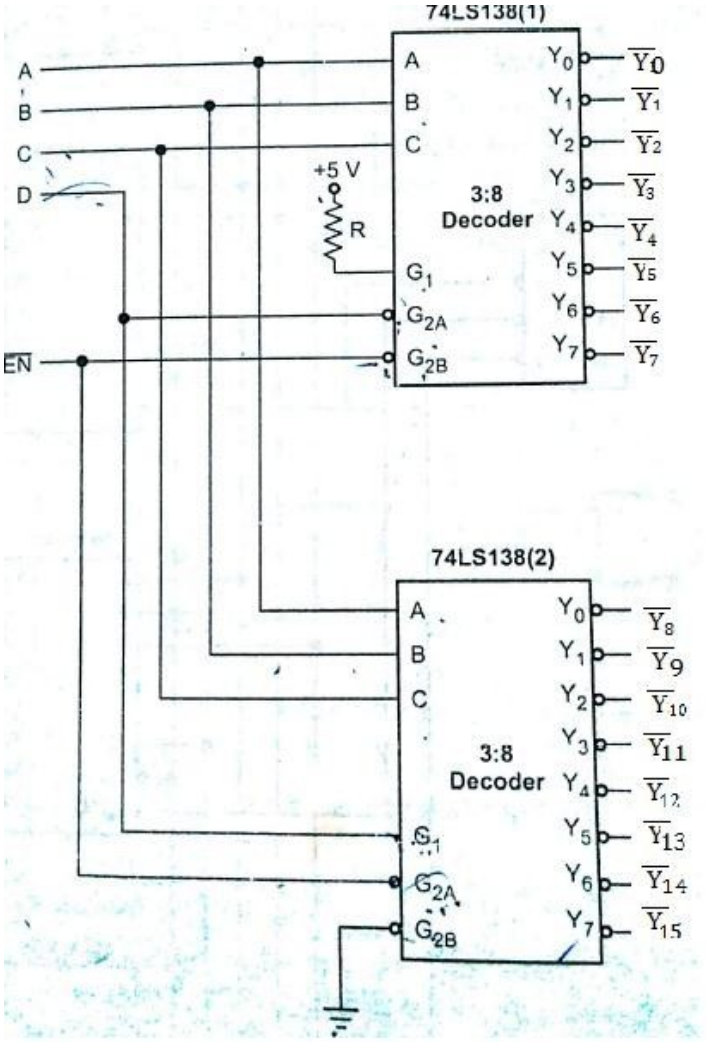
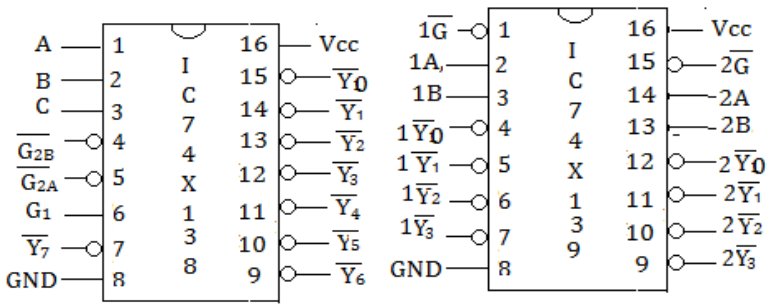


Figure 5.7: Implementation of 4:16 Decoder using IC 74X138

❖ **Implement the 5:32 binary decoder using required number of IC 74X138 and IC 74X139?**

The IC 74X138 is used to produce 3: 8 Decoder operation. This implementation requires two IC 74X138 to get specified 4 inputs and 16 outputs. Each IC 74X138 has three inputs and 3 enable inputs as G_1, G_{2A} and G_{2B}. Among these three enable inputs, G_1 is active high and G_{2A}, G_{2B} are active low. It produces seven active low outputs as Y_0, Y_1, Y_2, Y_3, Y_4, Y_5, Y_6 and Y_7. To get required specifications all three inputs of four number of IC 74X138 are shorted together with their respective input terminals.



The remaining 2 required inputs are taken from IC 74X139. The IC 74X139 is a Dual 2 to 4 Decoder. The IC 74X139 is 16 pin Dual In Packaged (DIP) IC. It has two sets of inputs 1A,1B and 2A, 2B. The IC 74X139 has two enable inputs as 1G, 2G both are active low for two decoders separately. It produces four active low outputs as $1Y_0$, $1Y_1$, $1Y_2$, $1Y_3$, $1Y_4$ for one half of the dual 2 to 4 decoder and $2Y_0$, $2Y_1$, $2Y_2$, $2Y_3$, $2Y_4$ for another half of the dual 2 to 4 decoder. For active low enable input only the IC 74X139 may produce valid outputs.

When the inputs of IC 74X139 are applied then corresponding resultant output is applied to active low enable input G_{2B} of IC74LS138. Based on that active low output, the corresponding IC74LS138 is activated and produces respective outputs. For each input combination of IC 74X139, only one IC74LS138 will be in active at a time. Truth table for one half of the IC 74X139 is given by

Enable Inputs	**Inputs**		**Outputs(Active low)**			
$1\overline{G}$	1A	1B	$1Y_0$	$1Y_1$	$1Y_2$	$1Y_3$
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

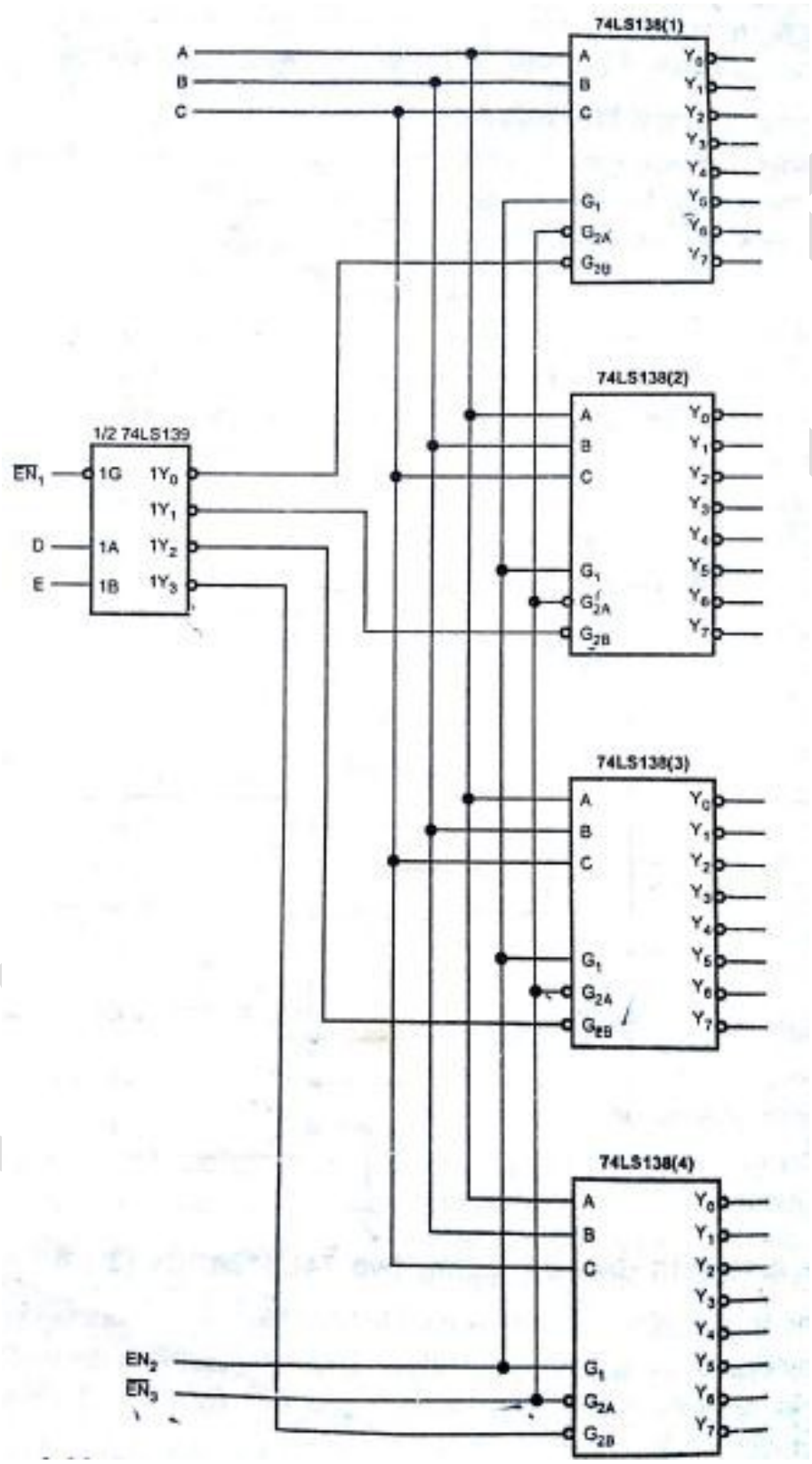

74LS138(1)
74LS138(2)
74LS138(3)
74LS138(4)
1/2 74LS139
A
B
C
D
E
EN1
EN2
EN3
1G
1A
1B
1Y0
1Y1
1Y2
1Y3
G1
G2A
G2B
Y0
Y1
Y2
Y3
Y4
Y5
Y6
Y7

Figure 5.8: Implementation of 5:32 Decoder using IC 74X138 and IC 74X139

❖ **(i)Realize the following boolean functions using required decoder.**
$F_1(A,B,C)= \Sigma m(1,2,3,7)$ and $F_2(A,B,C)=\pi M(1,3,5,7)$

(ii) Realize the following boolean functions using decoder IC 74X138.
$F_1(A,B,C)= \Sigma m(1,4,5,7)$ and $F_2(A,B,C)=\pi M(2,3,6,7)$

- ✓ The boolean functions can be represented as Sum Of Products (SOP) and Product Of Sum (POS).
- ✓ When decoder outputs are active high it provides outputs as minterms (standard product terms), Then the implemented boolean expression is called as Standard Sum Of Products.
- ✓ Standard SOPs can be implemented by decoder with help of OR gate when outputs are Active high. Standard POSs can also be implemented by decoder when outputs are Active high with help of NOR gate instead of OR gate by complementing SOP operation.
- ✓ When decoder outputs are active low it provides outputs as maxterms (standard sum terms), Then the implemented boolean expression is called as Standard Products Of Sum.
- ✓ Standard SOPs can be implemented by decoder with help of NAND gate when outputs are Active low. Standard POSs can also be implemented by decoder when outputs are Active low with help of AND gate instead of NAND gate by complementing SOP operation.

(i) $F_1(A,B,C)= \Sigma m(1,2,3,7)$ and $F_2(A,B,C)=\pi M(1,3,5,7)$

Solution: Given functions F_1 and F_2 have highest minterm as 7 and 3 inputs. So to implement both functions, the required decoder is 3:8 binary decoder. Let decoder has active high outputs. Hence we know, the standard SOPs can be implemented by decoder with help of OR gate when outputs are Active high.

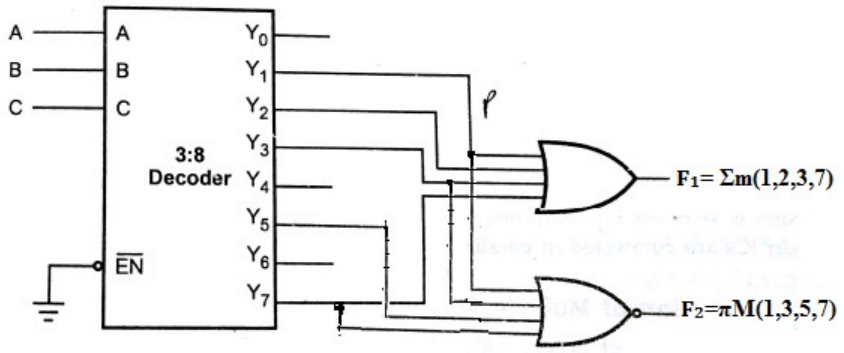


Figure 5.9: Implementation of F_1 and F_2 using Decoder

(ii) Realize the following boolean functions using decoder IC 74X138.

$F_1(A,B,C)= \Sigma m(1,4,5,7)$ and $F_2(A,B,C)=\pi M(2,3,6,7)$

IC 74X138 has three inputs and 3 enable inputs as G_1, G_{2A} and G_{2B}. Among these three enable inputs, G_1 is active high and G_{2A}, G_{2B} are active low. It produces Eight active low outputs as Y_0, Y_1, Y_2, Y_3, Y_4, Y_5, Y_6 and Y_7. Standard SOPs can be implemented by decoder with help of NAND gate when outputs are Active low. Standard POSs can also be implemented by decoder when outputs are Active low with help of AND gate instead of NAND gate by complementing SOP operation.

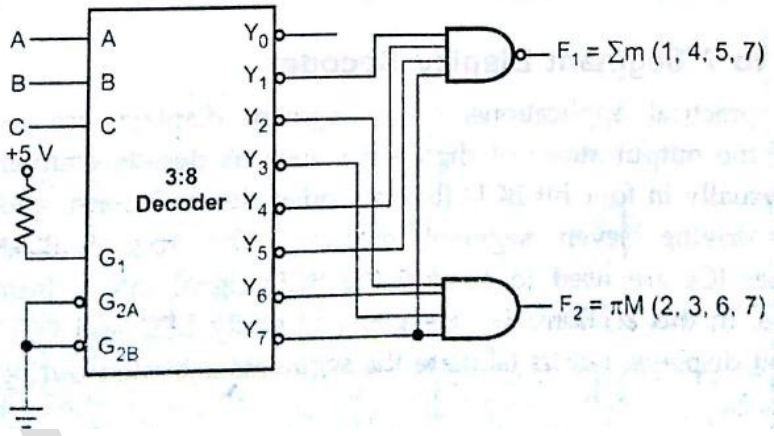


Figure 5.10: Implementation of F_1 and F_2 using Decoder IC74X138

❖ **Explain the operation of BCD to decimal decoder and configure the IC 7442?**

BCD to Decimal decoders have four inputs and ten outputs. The Binary Coded Decimal number are valid only from 0 to 9. So to get these valid BCD, we need 4 input variable. The 4-bit BCD input is decoded to activate one of the ten outputs.

Enable Inputs	Inputs				Outputs(Active low)									
En	A	B	C	D	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7	Y_8	Y_9
0	x	x	x	x	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	1	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	1	0	0	0	0	0	0	0
1	0	0	1	1	0	0	0	1	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	1	0	0	0	0	0
1	0	1	0	1	0	0	0	0	0	1	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0	1	0	0	0
1	0	1	1	1	0	0	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	0
1	1	0	0	1	0	0	0	0	0	0	0	0	0	1

Table 5.7(a): Truth table for BCD to Decimal Decoder

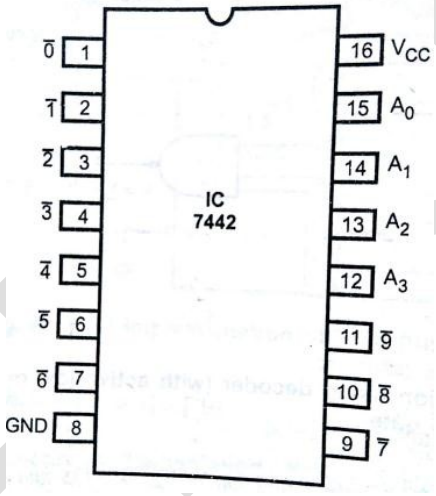


Figure 5.11: Pin diagram of BCD to Decimal decoder IC 7442

Pin Description: IC 7442 is a BCD to Decimal decoder. It accepts four active high BCD inputs and provides ten active low outputs. The Binary Coded Decimal number are valid only from 0 to 9. So to get these valid BCD. For the value above the 9, all outputs are active high.

Inputs				Outputs(Active low)									
A	B	C	D		1		3		5		7		
0	0	0	0	0	1	1	1	1	1	1	1	1	1
0	0	0	1	X	0	1	1	1	1	1	1	1	1
0	0	1	0	X	X	0	1	1	1	1	1	1	1
0	0	1	1	X	X	X	0	1	1	1	1	1	1
0	1	0	0	X	X	X	X	0	1	1	1	1	1

0	1	0	1	X	X	X	X	X	0	1	1	1	1
0	1	1	0	X	X	X	X	X	X	0	1	1	1
0	1	1	1	X	X	X	X	X	X	X	0	1	1
1	0	0	0	X	X	X	X	X	X	X	X	0	1
1	0	0	1	X	X	X	X	X	X	X	X	X	0

Table 5.7(b): Truth table for BCD to Decimal Decoder IC 7442

- **Design a BCD to Seven segment display decoder using common cathode connection and configure relevant Digital IC?**

A 7-segment display is used in watches, calculators, and devices to show decimal data. A digit is displayed by illuminating a subset of the 7 line segments. A 7-segment decoder has a 4-bit BCD as its input and the 7-segment code as its output. This decoder has two type of connections to produce outputs as common cathode and common anode connections. In common cathode connection the disabled segment indicated with active low value and enabled segments are indicated with active high value, But in common anode connection all connections are reversed compared to common cathode connection.

Figure 5.12(a): Seven segment display

Inputs				Outputs						
D	C	B	A	a	b	c	d	e	f	g
x	x	x	x	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

Table 5.8: Truth table for BCD to Seven segment display Decoder

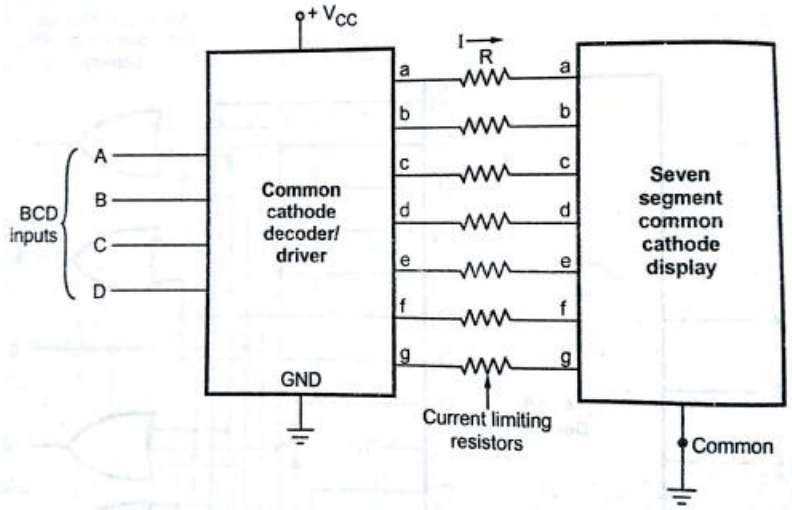


Figure 5.12(b) Basic connections for driving common cathode display

K-Map simplification:

For a

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

$a = A + C + BD + \overline{B}\,\overline{D}$

For b

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

$b = \overline{B} + \overline{C}\,\overline{D} + CD$

For c

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

$c = B + \overline{C} + D$

For d

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

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

For e

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

$e = \overline{B}\,\overline{D} + C\overline{D}$

For f

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

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

For g

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

$g = A + B\overline{C} + \overline{B}C + C\overline{D}$

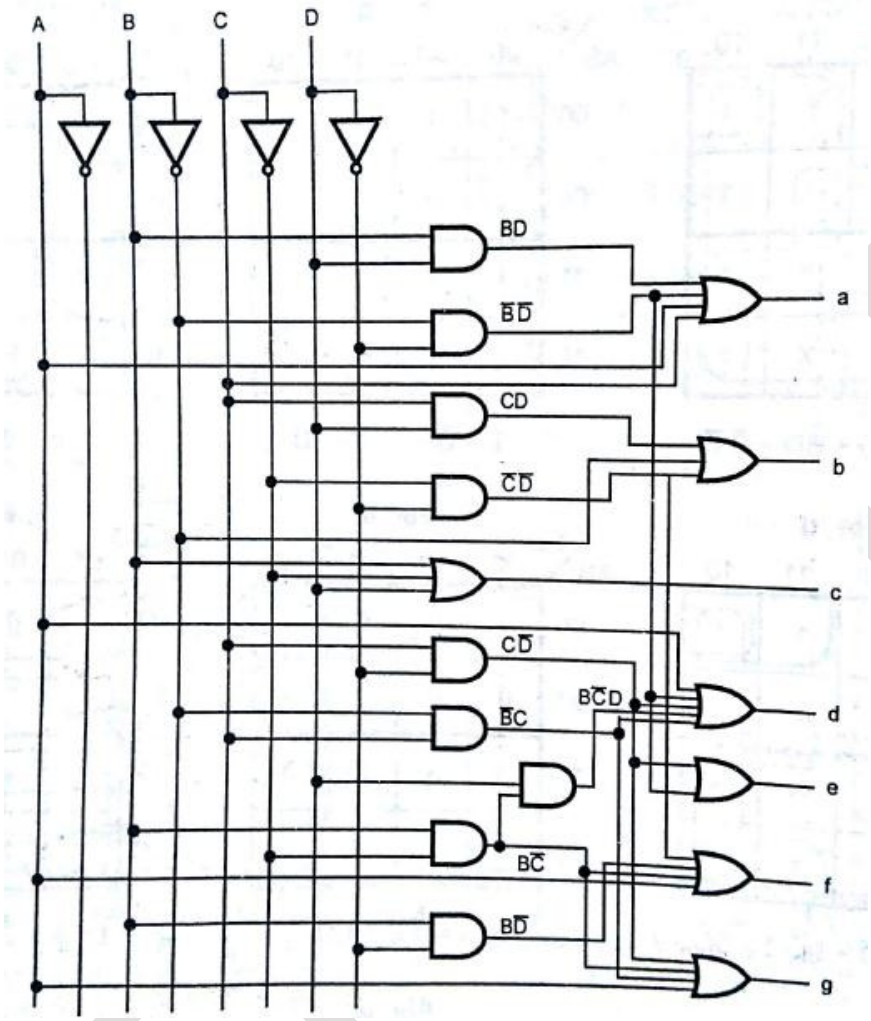


Figure 5.13: Logic diagram of BCD to seven segment decoder

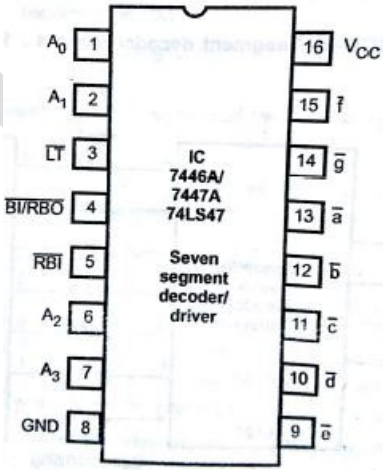


Figure 5.14: Pin diagram of BCD to Seven segment decoder IC 7447/7446A

In the IC7447 test lamp pin () is provided to test whether all segments are working properly or not. When pin is held low with pin open or at logic high, IC drives all display terminals ON (active low). When pin is pulled low all outputs are blanked; this pin also functions as a Ripple Blanking Output terminal. along with / can be used to provide ripple blanking feature.

To display Multiple digits we have to connect multiple seven segment decoders in multiplexed connection.

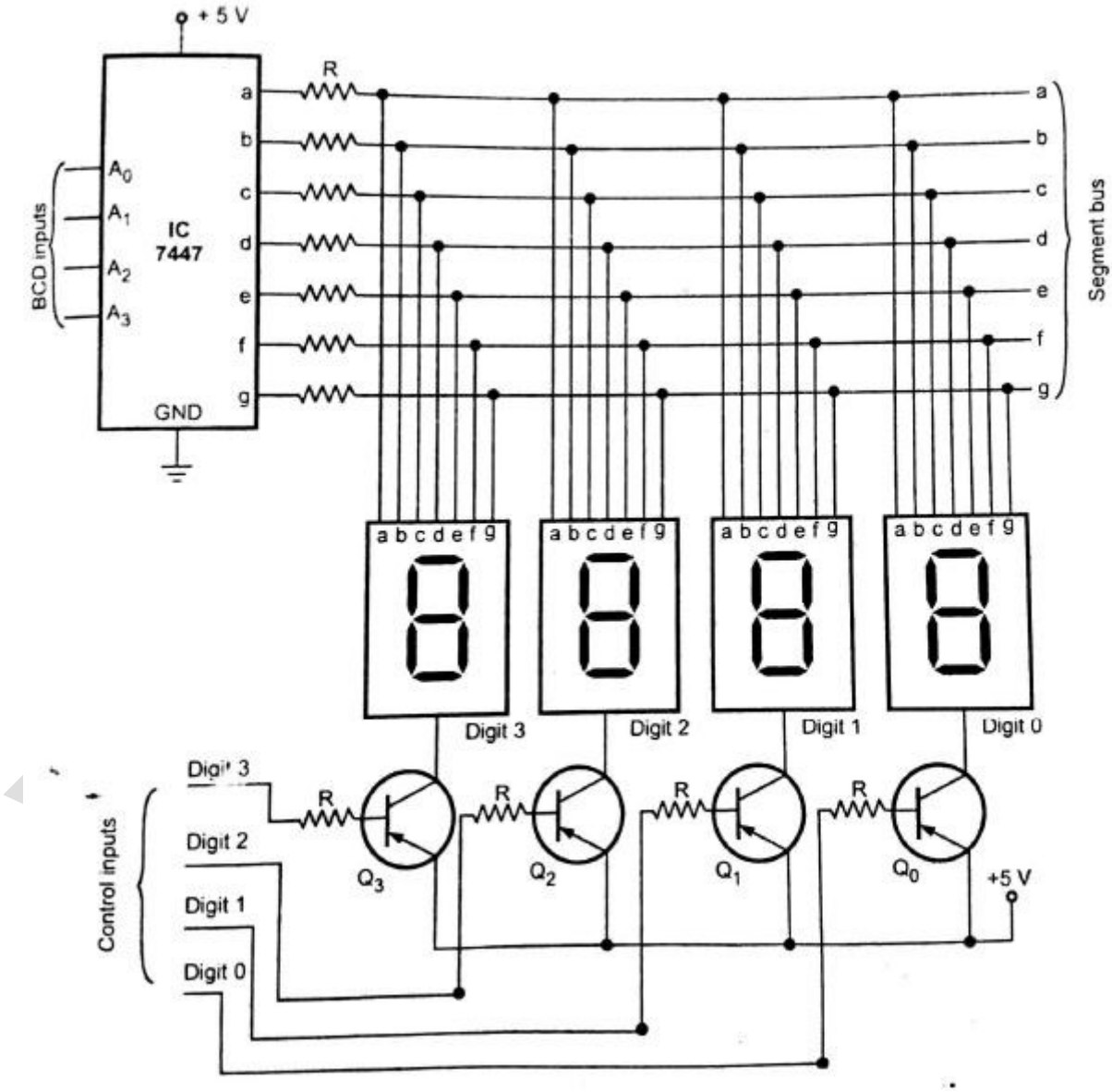


Figure 5.15: Seven segment display in multiplexed connection

❖ **Define binary encoder? Design octal to binary encoder?**

Encoder is a combinational logic circuit that allows 2^n data inputs and produces 'n' data outputs. Encoder gives a reverse operation of decoder circuit.

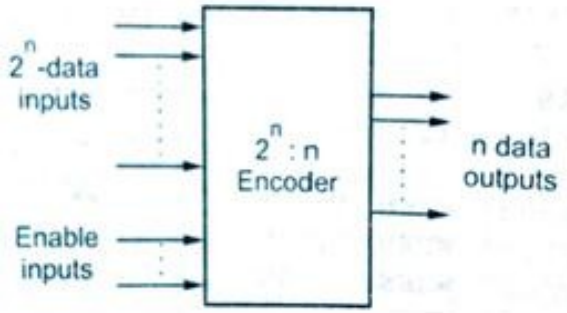


Figure 5.16: Block diagram of Encoder

Octal to binary encoder: Octal to binary encoder contains eight inputs and three outputs. Let eight inputs as $D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7$ and three outputs as A, B, C. Among all inputs only one input will be in active position to produce its corresponding value as output. Then the relation between input combinations and outputs is observed in truth table as follows.

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	A	B	C
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

Table 5.9: Octal to Binary Encoder

Then from truth table the boolean expressions for outputs A, B and C are obtained as

$$A = D_4 + D_5 + D_6 + D_7$$

$$B = D_2 + D_3 + D_6 + D_7$$

$$C = D_1 + D_3 + D_5 + D_7$$

From the above boolean expressions the logic diagram is obtained as

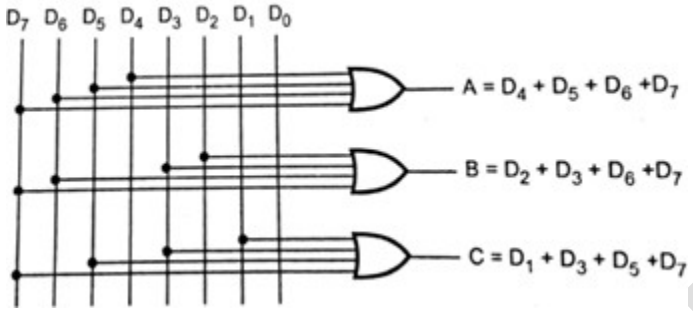


Figure 5.17: Logic diagram for Octal to Binary Encoder

- **Define binary encoder and explain the pin configuration of Decimal to BCD encoder?**

Encoder is a combinational logic circuit that allows 2^n data inputs and produces 'n' data outputs. Encoder gives a reverse operation of decoder circuit.

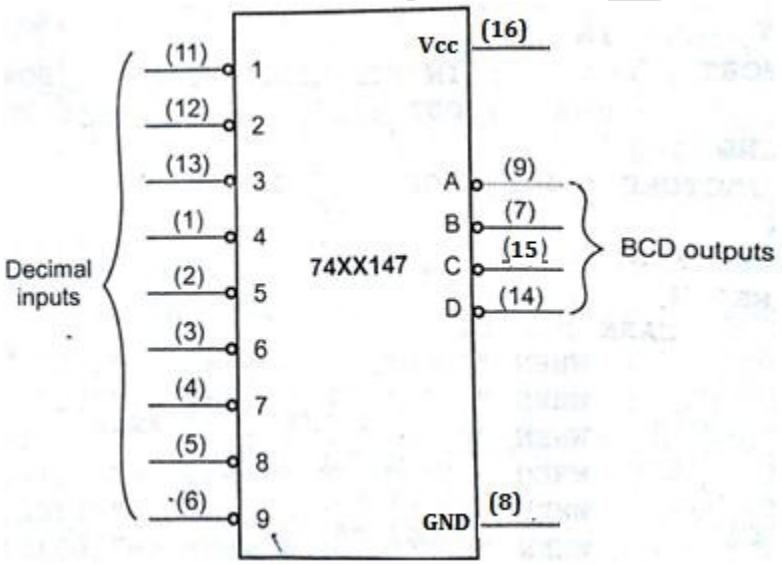


Figure 5.18: Pin diagram of Decimal to BCD Encoder

Decimal to BCD Encoder: The IC 74xx147 is used for Decimal to BCD encoder has nine active low inputs and four active low BCD outputs. The 10th pin is actually has no connection. If we consider it as 10th active low enable input there is no change is present in the IC operation. The IC 74xx147 has the operation as shown in the table 5.10. In the given table all the active low inputs are applied as logic '1' so the 0th (initial) input which is not in defined in the IC gives the corresponding Combinational output as 1111 because all BCD outputs are active low.

Inputs(Active low)									Outputs			
1		3		5		7			A	B	C	D
1	1	1	1	1	1	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1	1	1	0
X	0	1	1	1	1	1	1	1	1	1	0	1
X	X	0	1	1	1	1	1	1	1	1	0	0
X	X	X	0	1	1	1	1	1	1	0	1	1
X	X	X	X	0	1	1	1	1	1	0	1	0
X	X	X	X	X	0	1	1	1	1	0	0	1
X	X	X	X	X	X	0	1	1	1	0	0	0
X	X	X	X	X	X	X	0	1	0	1	1	1
X	X	X	X	X	X	X	X	0	0	1	1	0

Table 5.10 Truth table for Decimal to BCD Encoder

❖ **What is Priority encoder? Design a 4-bit Priority encoder?**

According to the definition of an encoder circuit, to get a valid output only one input should be in active among all available inputs. If more than one input is active at a time then to get a valid output, we have to consider the highest priority of all inputs. Let a four input priority encoder with four inputs as D_0, D_1, D_2, D_3, and two outputs as Y_0, Y_1, V, where 'V' is a valid output indicator. In this D_3 input with highest priority and D_0 with lowest priority. When D_3 input is high, regardless of other inputs output is '11'. D_2 has the next highest priority. Thus, when D_3=0 and D_2=1, regardless of other two lower priority inputs, output is '10', and so on. The valid output indicator 'V' indicates the obtaining of valid output for application of valid inputs. If all inputs in a priority encoder are '0' then V=0, otherwise V=1.

Inputs				Outputs		
D_0	D_1	D_2	D_3	Y_1	Y_0	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

Table 5.11 Truth table for 4-bit Priority Encoder

K-Map simplification:

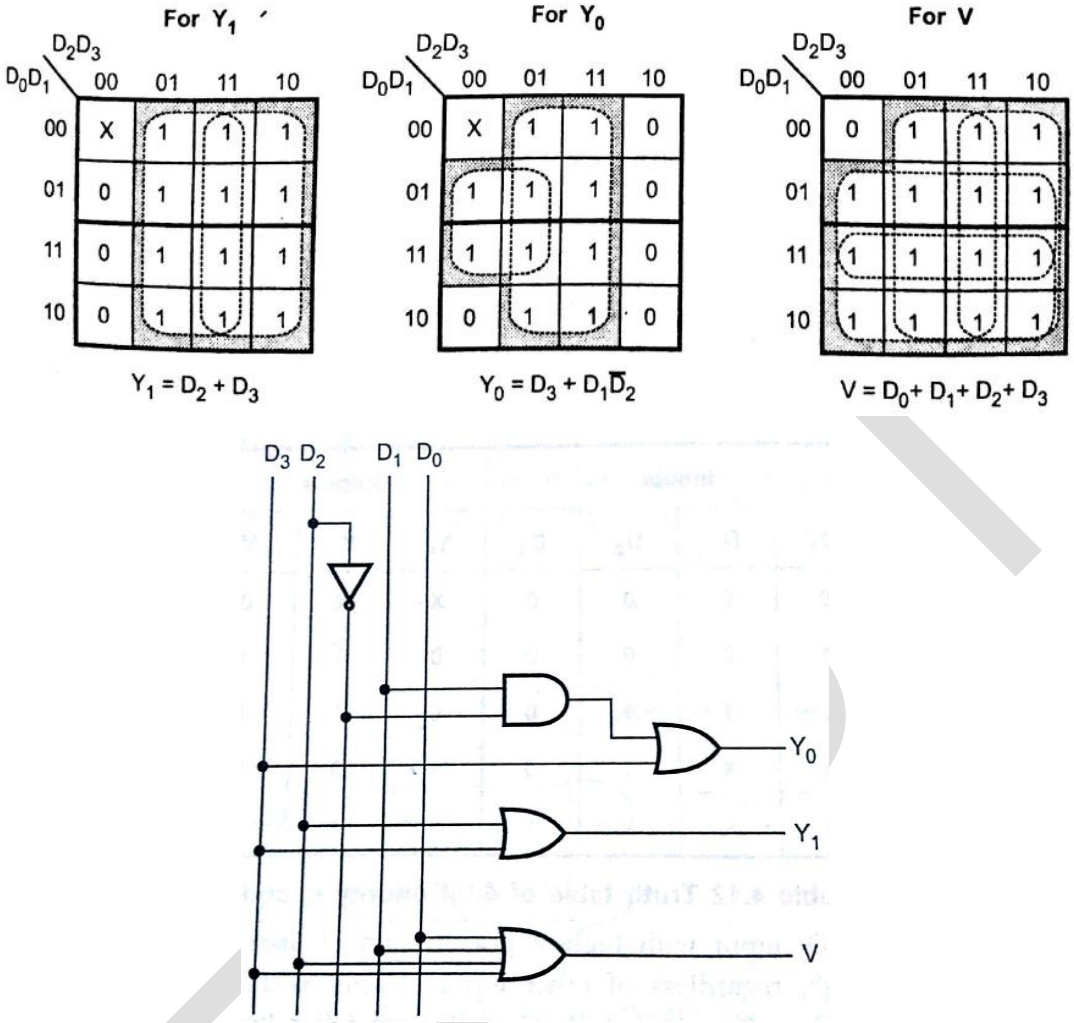


Figure 5.19: Logic diagram for 4-bit Priority Encoder

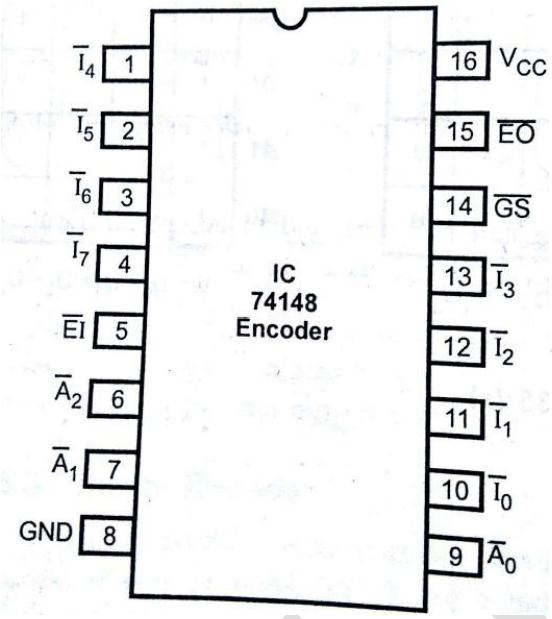


Figure 5.20: Pin diagram of 8-input Priority Encoder

Inputs									Outputs				
EI	0	1	2	3	4	5	6	7	A_2	A_1	A_0	GS	EO
1	x	x	x	x	x	x	x	x	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1	1	1	1	0
0	0	1	1	1	1	1	1	1	1	1	1	0	1
0	x	0	1	1	1	1	1	1	1	1	0	0	1
0	x	x	0	1	1	1	1	1	1	0	1	0	1
0	x	x	x	0	1	1	1	1	1	0	0	0	1
0	x	x	x	x	0	1	1	1	0	1	1	0	1
0	x	x	x	x	x	0	1	1	0	1	0	0	1
0	x	x	x	x	x	x	0	1	0	0	1	0	1
0	x	x	x	x	x	x	x	0	0	0	0	0	1

Table 5.12 Truth table for 8-bit Priority Encoder IC 74x148

Design a 32:5 priority encoder: To design a 32:5 priority encoder we require five, 8-bit Priority Encoder IC 74x148s and respective logic gates. The design arrangement is represented as shown in the figure 5.20

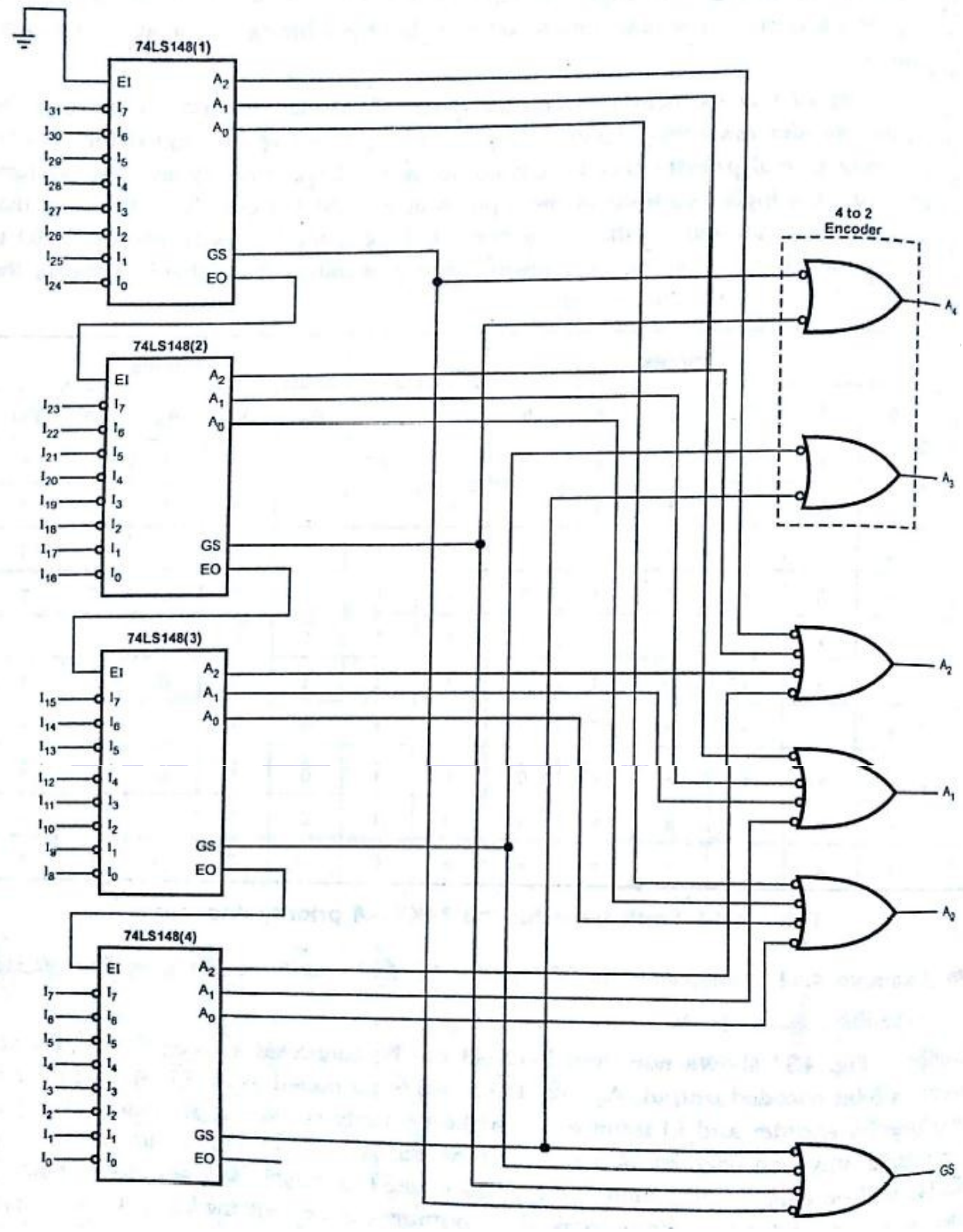


Figure 5.21: Design of 32-input 5-output priority encoder using IC 74x148 and gates

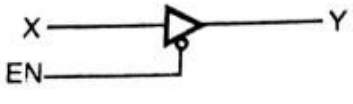
❖ **Tri-State devices and available digital ICs**

❖ A three state(tri-state) device is a digital device that exhibits three states as

- Logic 1 state (H).
- Logic 0 state (L).
- High impedance state (Z).

The high impedance state is the state of output that appears to be disconnected like an open circuit. In this state circuit has no logic significance. These tri-state device has a control input that can place the gate into a high impedance state. High impedance state is denoted with 'Z'. There are four types of tri-state gates are available. They are

- **Bufif0:** It is a tri-state device represents a non-inverting buffer that has three terminals as active low control input ,nInput X and output Y. If =0 then only it provides input X value as output. =1 then output exhibits as High impedance state.



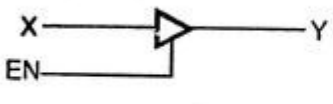
(a) Logic symbol

INPUTS		OUTPUT
EN	X	Y
0	0	0
0	1	1
1	x	Z

(b) Truth table

Figure 5.22: Bufif0 (a) Logic symbol; (b) Truth table

- **Bufif1:** It is a tri-state device represents a non-inverting buffer that has three terminals as active high control input En, Input X and output Y. If En=1 then only it provides input X value as output. En=0 then output exhibits as High impedance state.



(a) Logic symbol

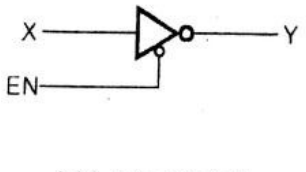
INPUTS		OUTPUT
EN	X	Y
1	0	0
1	1	1
0	x	Z

(b) Truth table

Figure 5.23: Bufif1 (a) Logic symbol; (b) Truth table

- **Notif0:** It is a tri-state device represents an inverting buffer that has three terminals as active low control input n Input X and output Y. If =0 then only it provides output as complement value of input X. =1 then output exhibits as High impedance state.

Figure 5.24: Notif0 (a) Logic symbol; (b) Truth table

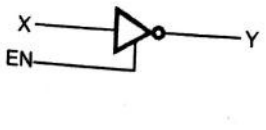


(a) Logic symbol

INPUTS		OUTPUT
EN	X	Y
0	0	1
0	1	0
1	x	Z

(b) Truth table

- **Notif1:** It is a tri-state device represents an inverting buffer that has three terminals as active High control input En, Input X and output Y. If En=1 then only it provides output as complement value of input X. En = 0 then output exhibits as High impedance state.



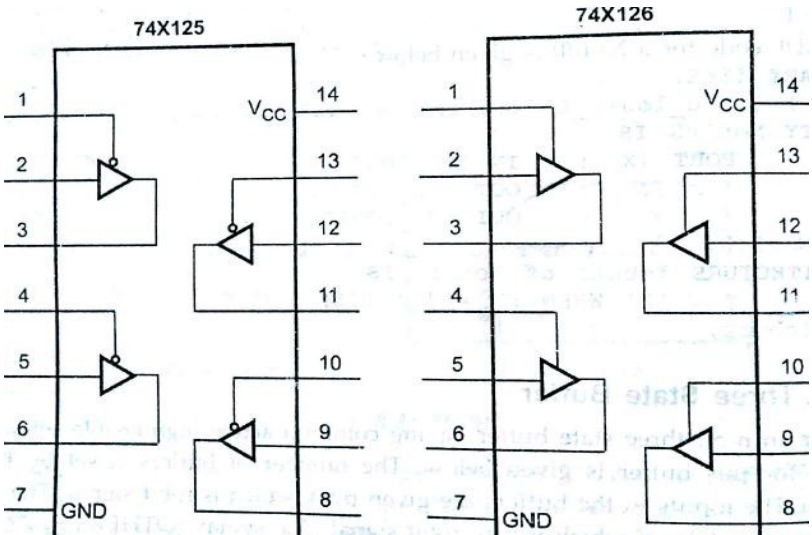
(a) Logic symbol

INPUTS		OUTPUT
EN	X	Y
1	0	1
1	1	0
0	x	Z

(b) Truth table

Figure 5.25: Notif1 (a) Logic symbol; (b) Truth table

Two ICs are available for tri-state devices, they are IC 74X125 and IC 74X126. Both the ICs are 14 pin Dual In Packaged ICs. The IC 74X125 has tri-state



buffer with active low enable and IC 74X126 has tri-state buffer with active high enable.

Figure 5.26: Pin diagrams of IC 74X125 and IC 74X126 tri-state buffers

❖ **What is a multiplexer? Design 4X1 and 8X1 multiplexer circuits?**

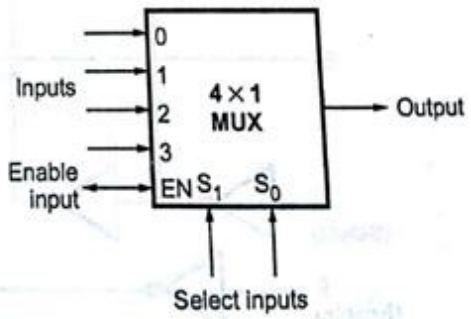
Multiplexer is a combinational digital logic circuit, it allows data information bits from several sources and selects only one data bit as output from all that are connected to the circuit based on the combination of selection input lines. That is selection of particular data input bit as output is controlled by a set of selection input lines. Hence multiplexer is also called as '*Data Selector'*. For 2^n input lines 'n' selection lines are required. So the mapping in multiplexer is *Many to One*.

Example: Design of 4X1 multiplexer.

A 4X1 multiplexer has 4 input lines, 2 selection input lines and one output line. Let D_0, D_1, D_2, D_3 are four data bits, S_0, S_1 are two selection input lines and 'Y' is the output. From four combinations of two selection inputs each one selects only one data bit from all four inputs (D_0, D_1, D_2, D_3) as output bit.

S_1	S_0	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

(a) Function table



(b) Logic symbol

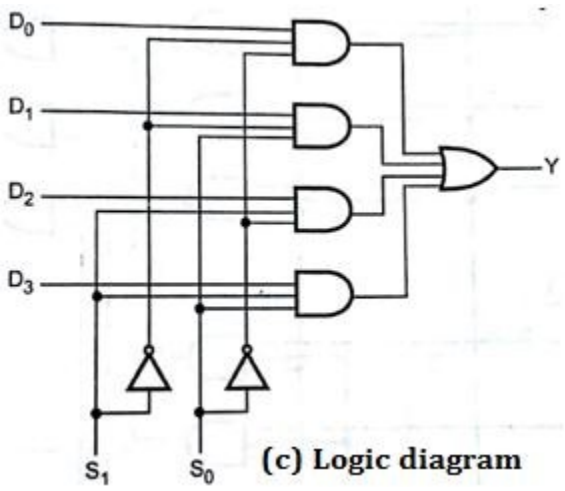


Figure 5.27: 4X1 multiplexer.

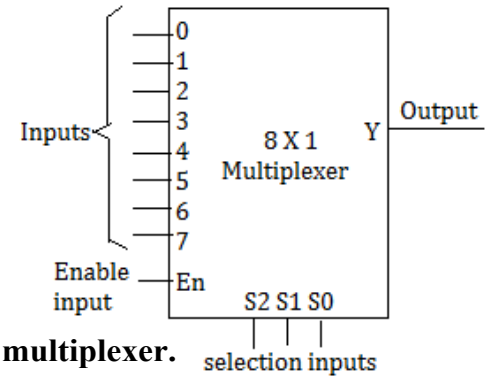
From the functional table, when S_0=0, S_1=0, then output selects D_0 ; when S_0=1, S_1=0, it selects D_1; when S_0=0, S_1=1, it selects D_2; when S_0=1, S_1=1, it selects D_3.

Example: Design of 8X1 multiplexer.

An 8X1 multiplexer has 8 input lines, 3 selection input lines and one output line. Let D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7 are eight data bits, S_0, S_1, S_2 are three selection input lines and 'Y' is the output. From four combinations of two selection inputs each one selects only one data bit from all eight inputs (D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7) as output bit.

Enable input	Selection inputs			Output
En	S_2	S_1	S_0	Y
0	X	X	X	0
1	0	0	0	D_0
1	0	0	1	D_1
1	0	1	0	D_2
1	0	1	1	D_3
1	1	0	0	D_4
1	1	0	1	D_5
1	1	1	0	D_6
1	1	1	1	D_7

(a) Truth table



(b) Logic Symbol

Figure 5.28: 8X1 multiplexer.

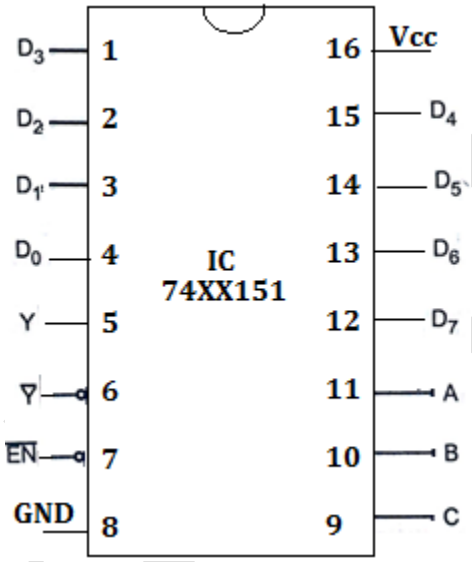
❖ **Explain the pin configuration and Design structure of the following ICs**

(i) 74XX151 (ii) 74XX157 (iii) 74XX153 (iv) 74XX150

(i) IC 74XX151:

The 74XX151 is a 8-to-1 multiplexer. It has eight inputs. It provides two outputs, one is active high, the other is active low. there are three select inputs C, B and A which select one of the eight inputs. The 74XX151 is provided with active low enable input.

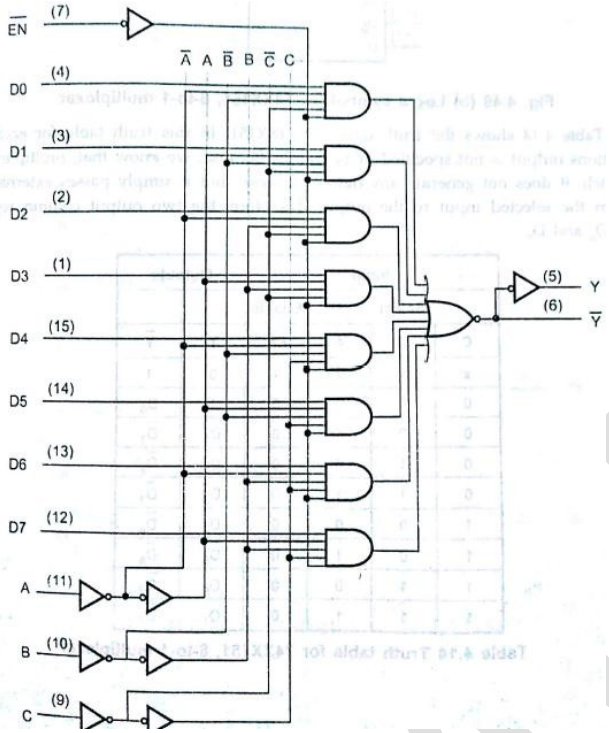
The IC 74xx151 has two output terminals, in those one output is active low and another output is active high.



(a) Pin Diagram of IC 74XX151 (8X1 Multiplexer)

Input				Outputs	
Select			Enable		
C	B	A	$\overline{EN}$	Y	$\overline{Y}$
x	x	x	1	0	1
0	0	0	0	D_0	$\overline{D_0}$
0	0	1	0	D_1	$\overline{D_1}$
0	1	0	0	D_2	$\overline{D_2}$
0	1	1	0	D_3	$\overline{D_3}$
1	0	0	0	D_4	$\overline{D_4}$
1	0	1	0	D_5	$\overline{D_5}$
1	1	0	0	D_6	$\overline{D_6}$
1	1	1	0	D_7	$\overline{D_7}$

(b) Truth table of IC 74XX151 (8X1 Multiplexer)



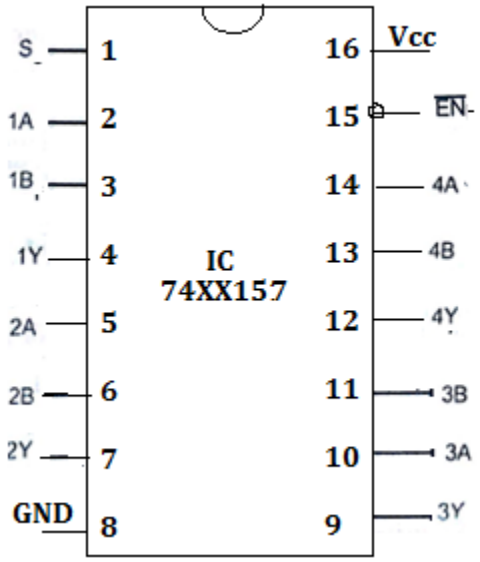
(c) Logic Diagram of IC 74XX151 (8X1 Multiplexer)

Figure 5.29: Implementation of IC 74XX151 (8X1 Multiplexer)

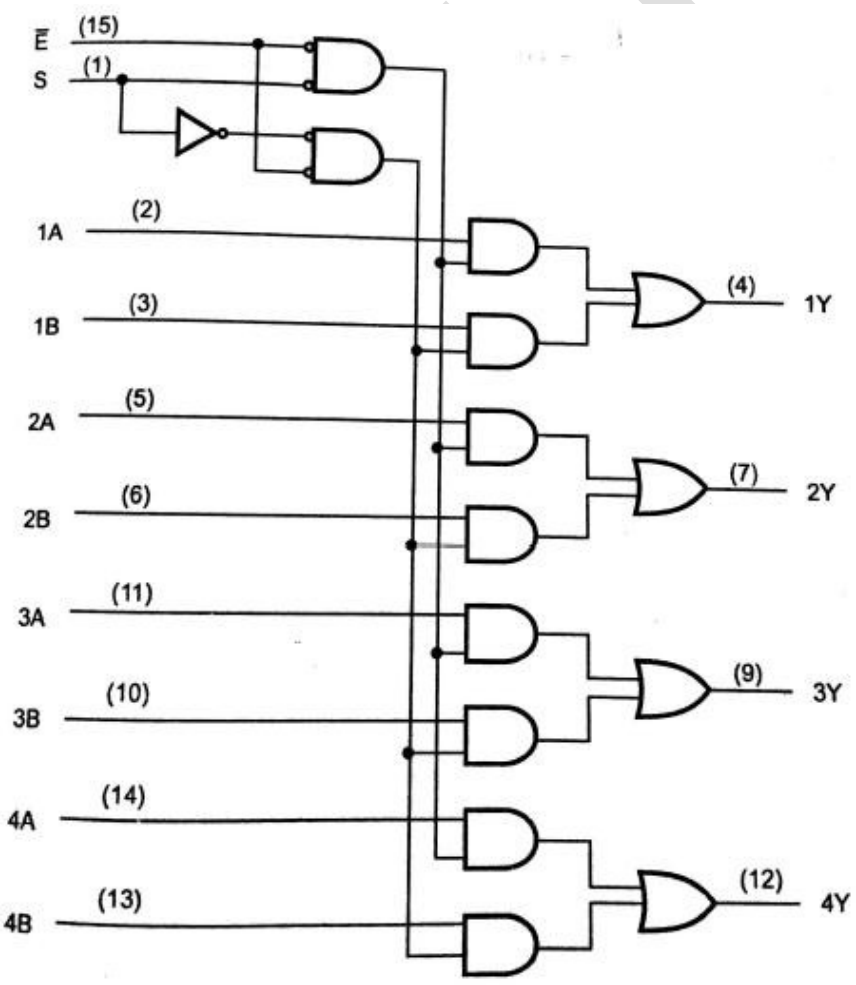
(ii) IC 74XX157: The IC 74XX157 is a Quad 2X1 multiplexer with 16 pins DIP. It has four 2X1 multiplexers that are fabricated on a single chip. The IC 74XX157 has one active low enable input and one selection input common for all 2X1 multiplexers.

Inputs		Outputs			
$\overline{E}$	S	1Y	2Y	3Y	4Y
1	x	0	0	0	0
0	0	1A	2A	3A	4A
0	1	1B	2B	3B	4B

(a) Truth table of IC 74XX157



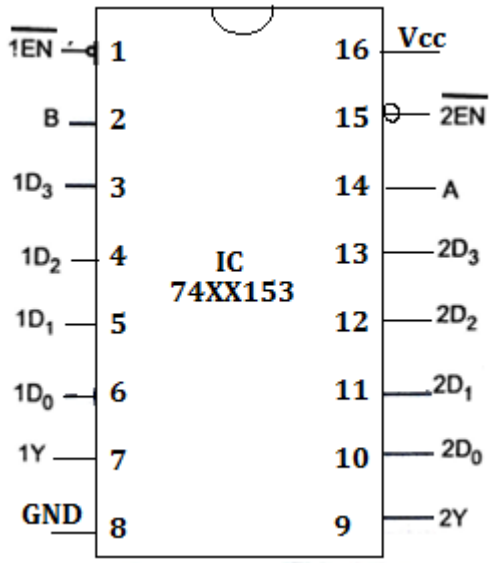
(b) Pin diagram of IC 74XX157



(c) Logic diagram of IC 74XX157

Figure 5.30: Implementation of IC 74XX157 (Quad 2X1 Multiplexer)

(iii) IC 74XX153: The IC 74XX153 is a Dual 4X1 multiplexer with 16 pins DIP. It has two identical and independent 4X1 multiplexers that are fabricated on a single chip. The IC 74XX153 has two active low enable inputs and two common selection inputs for both 4X1 multiplexers. When the corresponding enable input is active low then only its respective data input bit is obtained as output. If '1En' is active high no data bit will be appeared at output 1Y. Similarly if '2En' is active high no data bit will be appeared at output 2Y.



(a) Pin diagram of IC 74XX153(Dual 4X1 Multiplexer)

Inputs				Outputs	
1EN	2EN	B	A	1Y	2Y
0	0	0	0	$1D_0$	$2D_0$
0	0	0	1	$1D_1$	$2D_1$
0	0	1	0	$1D_2$	$2D_2$
0	0	1	1	$1D_3$	$2D_3$
0	1	0	0	$1D_0$	0
0	1	0	1	$1D_1$	0
0	1	1	0	$1D_2$	0
0	1	1	1	$1D_3$	0
1	0	0	0	0	$2D_0$
1	0	0	1	0	$2D_1$
1	0	1	0	0	$2D_2$
1	0	1	1	0	$2D_3$
1	1	x	x	0	0

(b) Truth table of IC 74XX153

Figure 5.31: Implementation of IC 74XX153 (Dual 4X1 Multiplexer)

(iv) IC 74XX150: The IC 74XX150 is a Dual 16X1 multiplexer with 24 pins DIP. The IC 74XX150 has one active low enable input four selection input lines. If the enable input is active low then only the respective data input bit is obtained as output. Enable input is active high no data bit will be appeared at output Y.

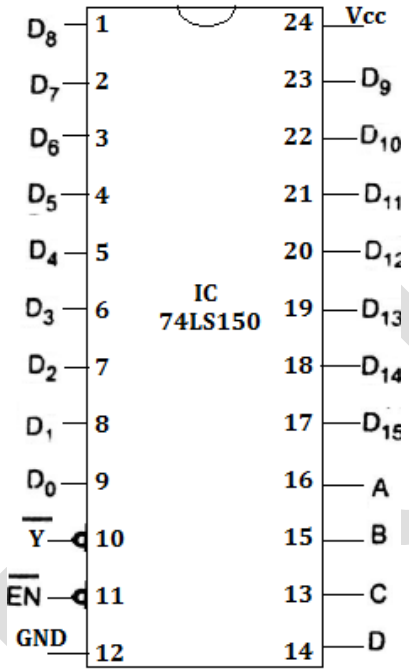


Figure 5.32(a): Pin diagram of IC 74XX150 (16X1 Multiplexer)

Figure 5.32(b): Truth table of IC 74XX150 (16X1 Multiplexer)

❖ **Design a 32X1 multiplexer by using required number of 8X1 multiplexers and 2:4**

Enable input	Selection inputs				Output
$\overline{En}$	D	C	B	A	$\overline{Y}$
1	X	X	X	X	0
0	0	0	0	0	$\overline{D_0}$
0	0	0	0	1	$\overline{D_1}$
0	0	0	1	0	$\overline{D_2}$
0	0	0	1	1	$\overline{D_3}$
0	0	1	0	0	$\overline{D_4}$
0	0	1	0	1	$\overline{D_5}$
0	0	1	1	0	$\overline{D_6}$
0	0	1	1	1	$\overline{D_7}$
0	1	0	0	0	$\overline{D_8}$
0	1	0	0	1	$\overline{D_9}$
0	1	0	1	0	$\overline{D_{10}}$
0	1	0	1	1	$\overline{D_{11}}$
0	1	1	0	0	$\overline{D_{12}}$
0	1	1	0	1	$\overline{D_{13}}$
0	1	1	1	0	$\overline{D_{14}}$
0	1	1	1	1	$\overline{D_{15}}$

Decoder?

To design a 32X1 multiplexer we require four 8X1 multiplexers and one 2:4 decoder. The three selection inputs of four 8X1 multiplexers are shorted together and remaining two selection inputs are taken from inputs of 2:4 decoder.

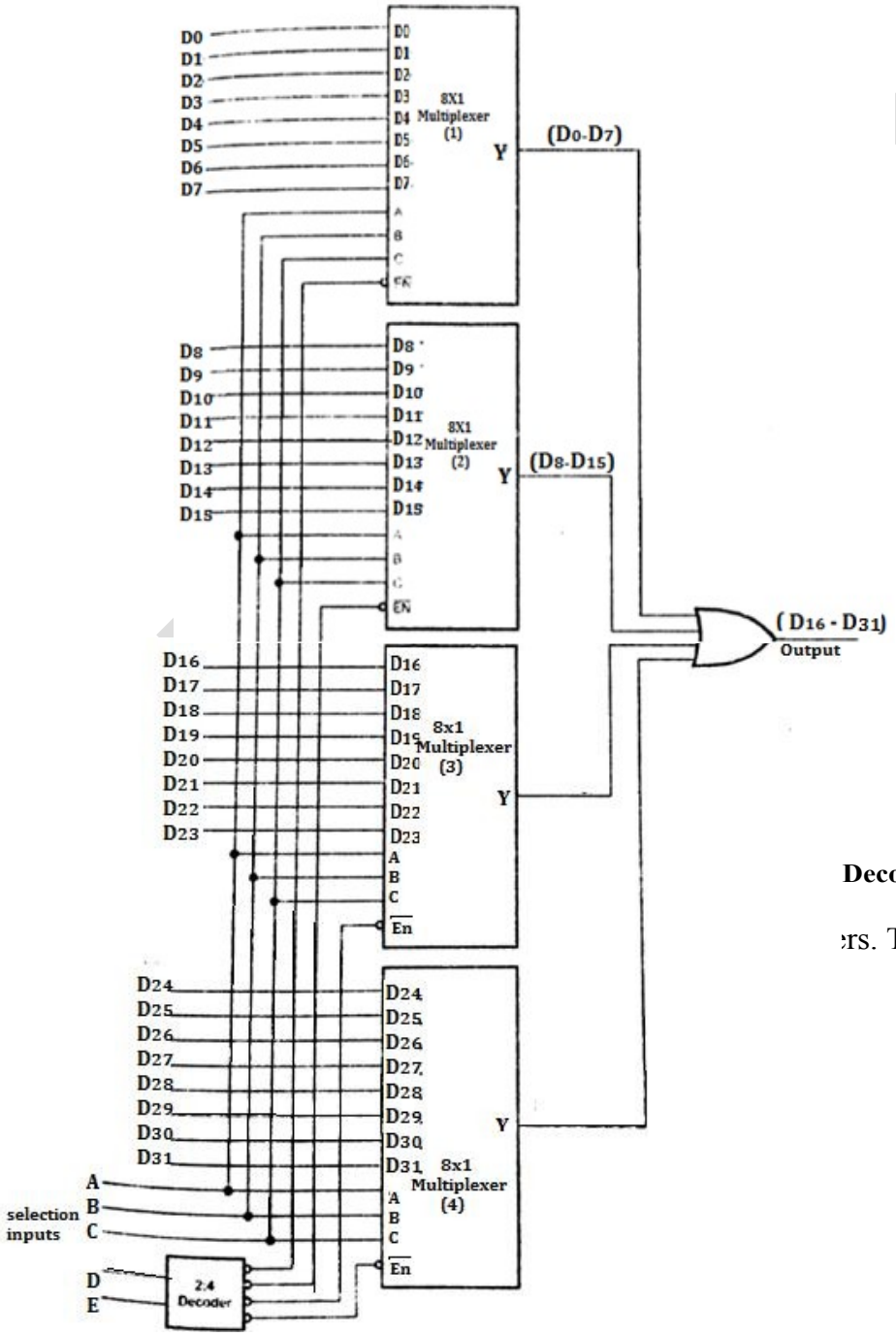


Figure 5.34 **Decoder**

The decoder ou ers. The operation 2:4 de

Table 5.6(b): Truth table for 2:4 Decoder

At a time only one 8X1 multiplexer is activated by depending on the operation of 2:4

decoder operation.

❖ **Design a 32X1 multiplexer by using required number of IC 74xx151 & IC 74x139?**

- The IC 74XX151 is an 8X1 multiplexer with 16 pins DIP. To design a 32X1 multiplexer we require four IC 74XX151.
- The three selection inputs of both ICs are shorted together, and remaining selection inputs are taken from inputs of a Decoder IC
- At a time only one IC74xx151 is activated by depending on the operation of Dual 2:4 decoder operation.

Enable Inputs	Inputs		Outputs(Active low)			
$1\overline{G}$	1A	1B	$1Y_0$	$1Y_1$	$1Y_2$	$1Y_3$
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

Table 5.6(b): Truth table for IC 74X139(one half only)

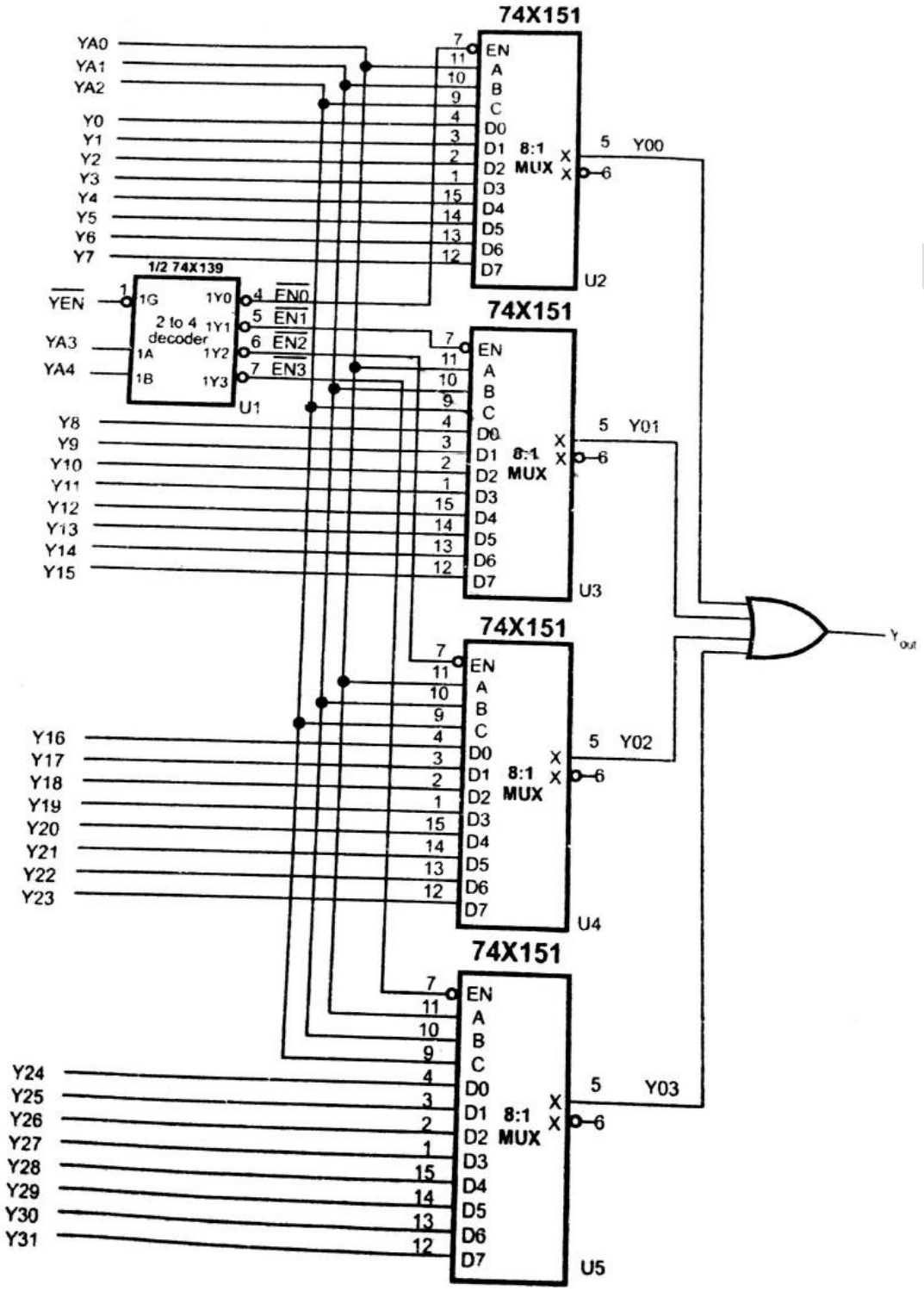


Figure 5.35: Implementation of 32X1 multiplexer using IC74XX151 & IC74xx139

❖ **Implement the boolean function F(A,B,C)= Σm (1, 3, 5, 6) using 8x1 multiplexer?**

In the given boolean function F(A,B,C)= Σm (1, 3, 5, 6), the number of inputs are three. So to implement the function F, an 8x1 multiplexer is required. In this implementation the minterms given in the boolean function are connected to logic '1', and not included inputs are connected to logic '0'.

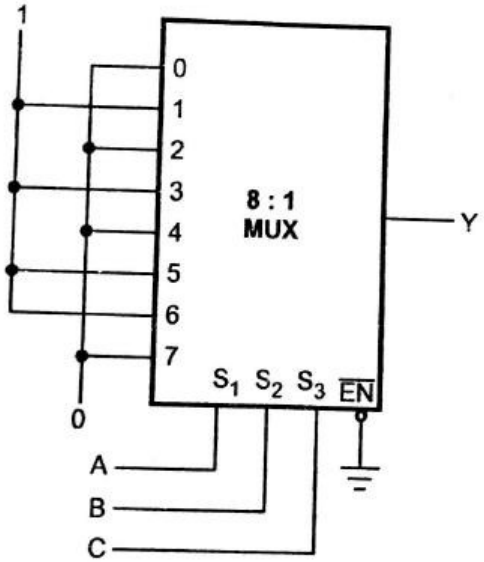


Figure 5.36: Implementation of F(A,B,C)= Σm (1, 3, 5, 6) using 8x1 multiplexer

❖ **Implement the boolean function F(A,B,C)= Σm (1, 3, 5, 6) using 4x1 multiplexer?**

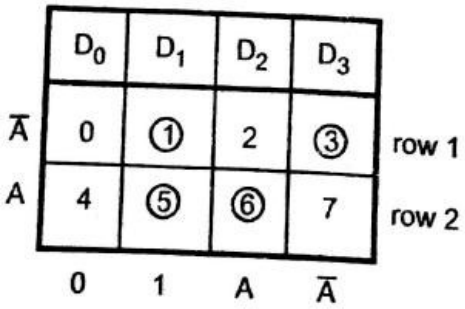
In the given boolean function F(A,B,C)= Σm (1, 3, 5, 6), the number of inputs are three. So to implement the function F, an 8x1 multiplexer is required. In this implementation the minterms given in the boolean function are connected to logic '1', and not included inputs are connected to logic '0'. But the implementation has to be design by using 4x1 Multiplexer. So to design a boolean function with lower order multiplexer we have two methods. They are row-wise implementation and column-wise implementation.

Row-Wise Implementation: In the row-wise implementation, the two input variable B, C are connected to two selection inputs of 4x1 multiplexer and input variable A is labeled for rows of implementation table. When 'A' variable is logic '0' then corresponding combinational row is named as *A* and when 'A' variable is logic '1' then corresponding

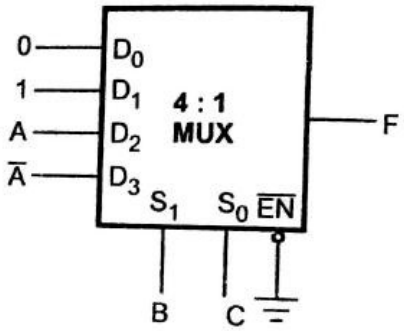
combinational row is named as A. Then the values for four inputs of multiplexer are obtained by comparing each column minterms by circle the included minterms in the given function. If both minterms in a column are not circled then the corresponding input of multiplexer is represented by logic '0', if both are circled then the corresponding input of multiplexer is represented by logic '1'. If only one minterm in a column is circled then the name of corresponding row is assigned to input of 4x1 multiplexer.

Minterm	A	B	C	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

(a) Truth table for function F(A,B,C)= Σm (1, 3, 5, 6)



(b) Implementation table



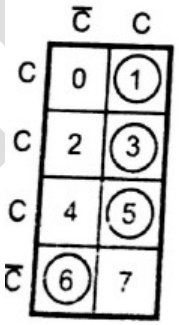
(c) Multiplexer implementation

Figure 5.37: Implementation of F(A,B,C) using Row-Wise Implementation

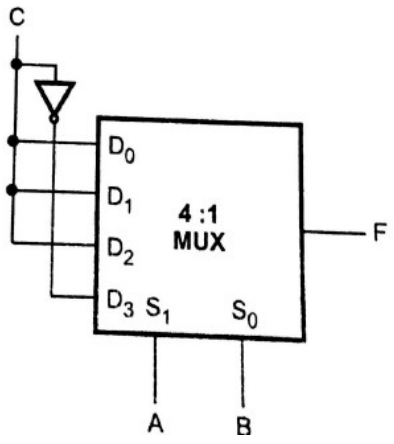
Column-Wise Implementation: In the Column-wise implementation, the two input variable A, B are connected to two selection inputs of 4x1 multiplexer and input variable 'C' is labeled for Columns of implementation table. When 'C' variable is logic '0' then corresponding combinational Column is named as $\bar{C}$ and when 'C' variable is logic '1' then corresponding combinational Column is named as C. Then the values for four inputs of multiplexer are obtained by comparing each row minterms by circle the included minterms in the given function. If both minterms in a row are not circled then the corresponding input of multiplexer is represented by logic '0', if both are circled then the corresponding input of multiplexer is represented by logic '1'. If only one minterm in a row is circled then the name of corresponding column is assigned to input of 4x1 multiplexer.

Minterm	A	B	C	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

(a) Truth table for function F(A,B,C)= Σm (1, 3, 5, 6)



(b) Implementation table



(c) Multiplexer implementation

Figure 5.38: Implementation of F(A,B,C) using Column-Wise Implementation

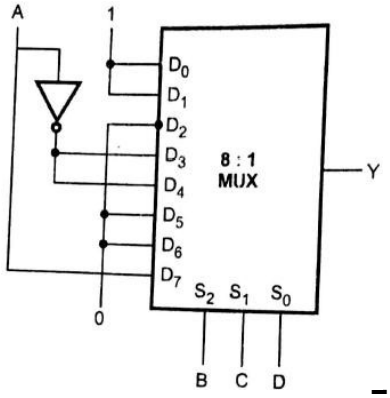
❖ **Implement the boolean function F(A, B, C, D)= Σm (0, 1, 3, 4, 8, 9, 15) using 8x1 multiplexer?**

In the given boolean function F(A, B, C, D)= Σm (0, 1, 3, 4, 8, 9, 15), the number of inputs are four. So to implement the function F, a 16x1 multiplexer is required. But the implementation has to be design by using 8x1 Multiplexer. So to design a boolean function with lower order multiplexer we have two methods. They are row-wise implementation and column-wise implementation.

In the row-wise implementation, the three input variable B, C, D are connected to three selection inputs of 8x1 multiplexer and input variable A is labeled for rows of implementation table. When 'A' variable is logic '0' then corresponding combinational row is named as $\bar{A}$ and when 'A' variable is logic '1' then corresponding combinational row is named as A. Then the values for eight inputs of multiplexer are obtained by comparing each column minterms by circle the included minterms in the given function. If both minterms in a column are not circled then the corresponding input of multiplexer is represented by logic '0', if both are circled then the corresponding input of multiplexer is represented by logic '1'. If only one minterm in a column is circled then the name of corresponding row is assigned to input of 8x1 multiplexer.

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

(d) Implementation table



(e) Multiplexer implementation

Figure 5.39: Implementation of F(A,B,C,D) using Row-Wise Implementation

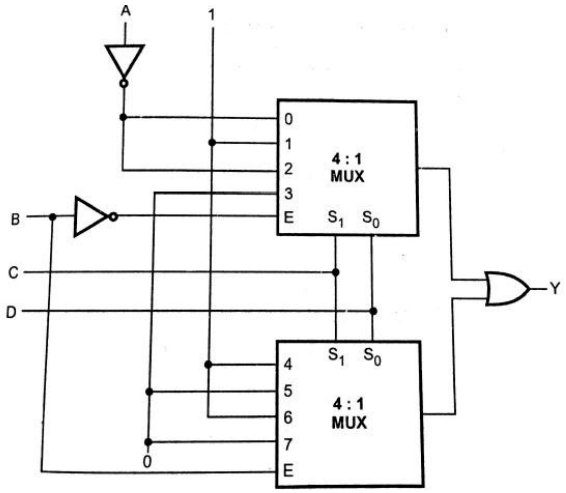
❖ **Implement the boolean function F(A, B, C, D)= Σm (0, 1, 2, 4, 6, 9, 12, 14) using 4x1 multiplexer?**

In the given boolean function F(A, B, C, D)= Σm (0, 1, 2, 4, 6, 9, 12, 14), the number of input variables are four. So to implement the function F, a 16x1 multiplexer is required. But the implementation has to be design by using 4x1 Multiplexer. So to design given function with 4x1 multiplexer, two 4x1 multiplexers are cascaded together. Hence to design a boolean function with lower order multiplexer we have two methods. They are row-wise implementation and column-wise implementation.

Row-Wise Implementation: In the row-wise implementation, the two input variable C, D are connected to two selection inputs of 4x1 multiplexer and input variable A is labeled for rows of implementation table. When 'A' variable is logic '0' then corresponding combinational row is named as *A* and when 'A' variable is logic '1' then corresponding combinational row is named as A. Then the values for four inputs of multiplexer are obtained by comparing each column minterms by circle the included minterms in the given function. If both minterms in a column are not circled then the corresponding input of multiplexer is represented by logic '0', if both are circled then the corresponding input of multiplexer is represented by logic '1'. If only one minterm in a column is circled then the name of corresponding row is assigned to input of 4x1 multiplexer. The input variable 'B' is connected to enable inputs of both 4x1 multiplexers in a complementary connection.

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

(a) Implementation table



(b) Multiplexer implementation

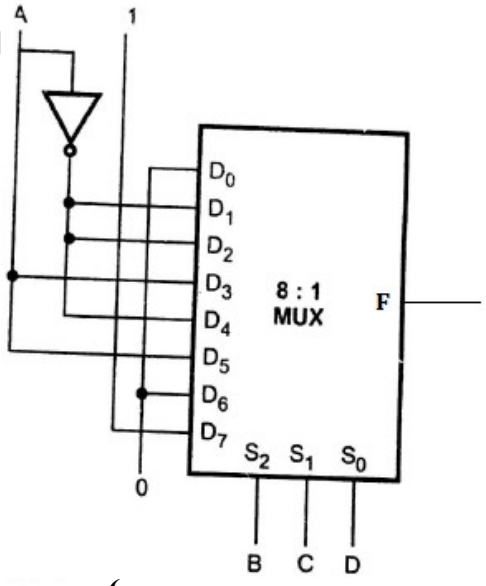
Figure 5.40: Implementation of F(A,B,C,D) using cascading of multiplexers

- **Implement the boolean function F(A, B, C, D)= πM (0, 3, 5, 8, 9, 10, 12, 14) using 8x1 multiplexer?**

In the given boolen function instead of minterms maxterms are given, so in the part of implementation we have to circle the maxterms which are not included in the given function.

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

(c) Implementation table



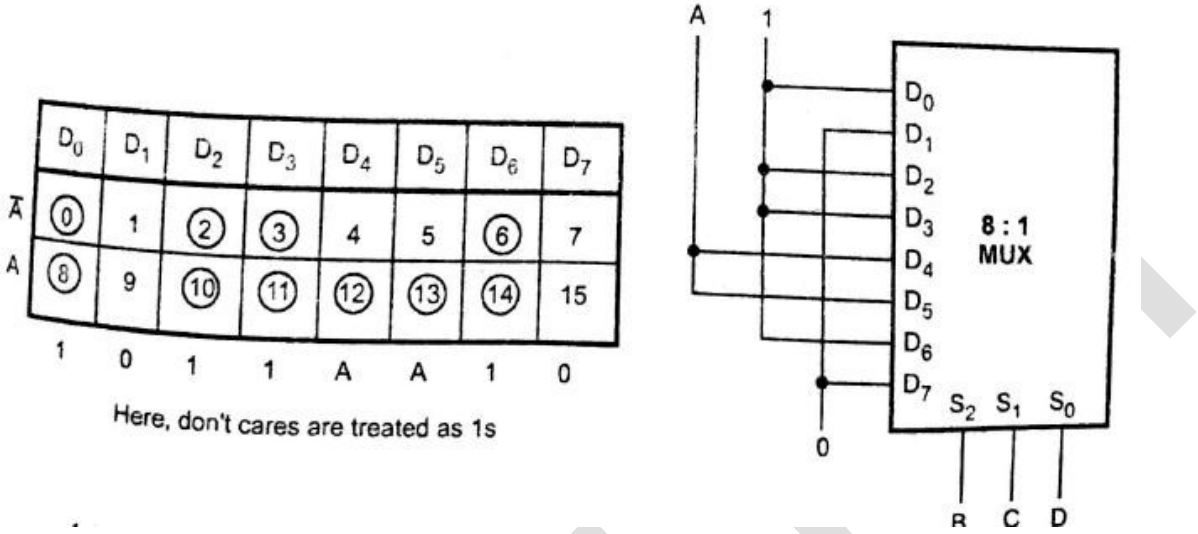
(

(d) Multiplexerimplementation

Figure 5.41: Implementation of F(A,B,C,D) using cascading of multiplexers

- **Implement the following boolean function using 8x1 multiplexer? F(A, B, C, D)= Σm (0, 2, 6, 10, 11, 12, 13)+d(3, 8, 14)**

In the given boolen function in addition to the minterms don't care values are given, so in the part of implementation we have to include and circle the don't care values also.



(e) Implementation table **(b) Multiplexer implementation**

Figure 5.42: Implementation of F(A,B,C,D) using 8x1 multiplexer

❖ **Define Demultiplexer. Explain its operation with a suitable example?**

Demultiplexer is a combinational digital logic circuit, it allows only one data bit from single source and distributed it to multiple outputs based on the combination of selection input lines. That is selection of particular output line is controlled by a set of selection input lines. Hence multiplexer is also called as '*Data Distributor'*. For 2^n output lines 'n' selection lines are required. So the mapping in Demultiplexer is One to *Many*.

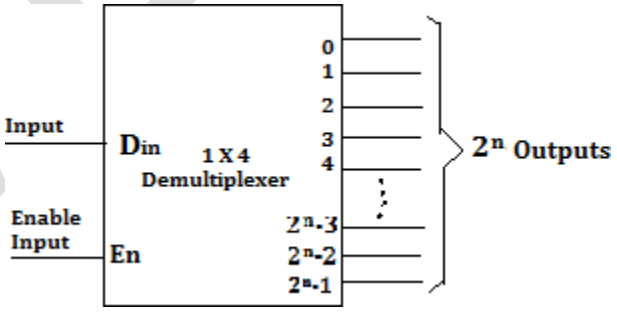


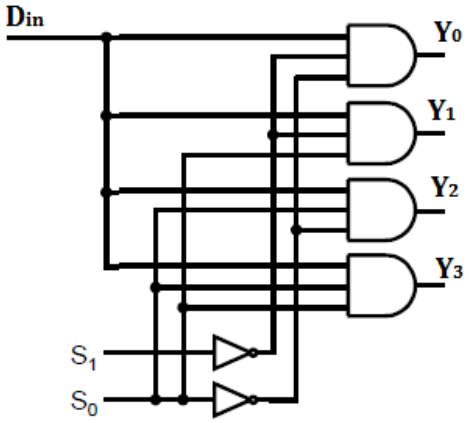
Figure 5.43: Block diagram of Demultiplexer

Example: Design of 1X4 Demultiplexer:

A 1X4 demultiplexer has 1 input line, 2 selection input lines and four output lines. Let Din is data bit, S_0, S_1 are two selection input lines and Y_0, Y_1, Y_2, Y_3 are the outputs. From four combinations of two selection inputs, Din is distributed to any one of Y_0, Y_1, Y_2, Y_3 data output lines.

Enable Input	Inputs		Outputs			
En	S_1	S_0	Y0	Y1	Y2	Y3
0	X	X	0	0	0	0
1	0	0	Din	0	0	0
1	0	1	0	Din	0	0
1	1	0	0	0	Din	0
1	1	1	0	0	0	Din

(f) Truth table for 1X4 Demultiplexer



(g) Logic diagram of 1X4 Demultiplexer

Figure 5.44: Design of 1X4 Demultiplexer

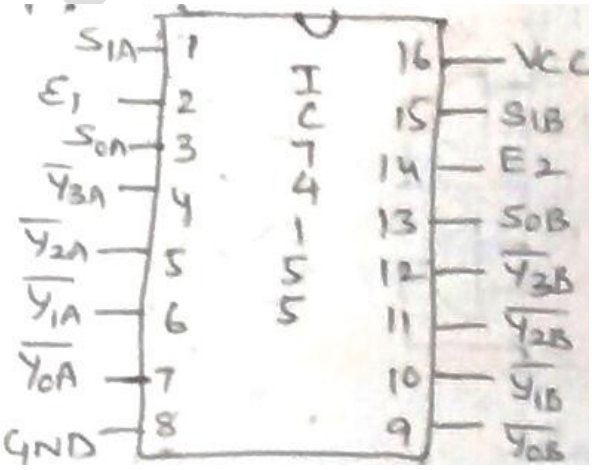


Figure 5.45: Pin Diagram of IC74xx155(dual 1X4 Demultiplexer)

❖ **Design a 4-bit binary to BCD code convertor circuit?**

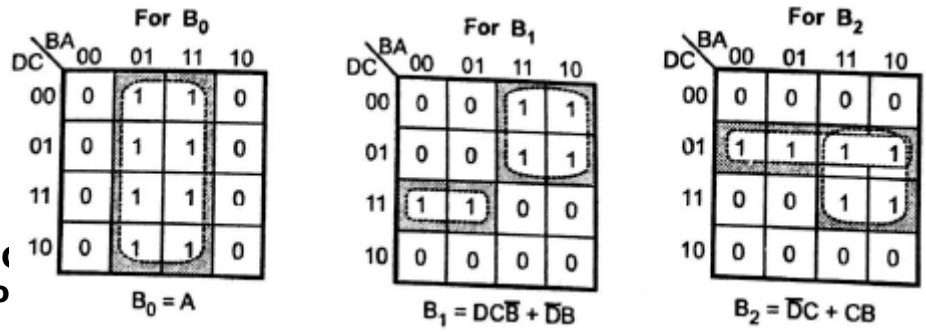
There are several types of binary codes are used in digital systems. Some of these codes are BCD, Excess-3, Gray and so on. Sometimes it may requires to change one code into another code format. The logic circuit used to convert one code into another code is called as 'Code Converter'.

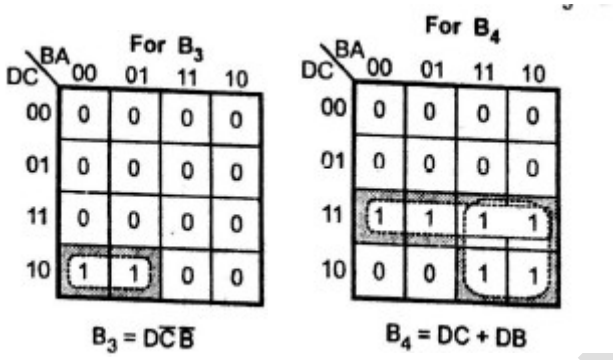
Designing of 4-bit binary to BCD code convertor:

Given 4-bit binary code has to convert into BCD code. We know BCD has valid numbers only from 0 to 9. To make the values above the 9 as valid BCD numbers, a value '6' is added. Here we have 4-binary bits at input so there is a possibility of 16 combinations. For 0 to 9 only the valid BCD code we can generate, for combinations from 10 to 15 a number 6 is added to get valid BCD numbers.

Binary code				BCD code				
D	C	B	A	B_4	B_3	B_2	B_1	B_0
0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	1
0	0	1	0	0	0	0	1	0
0	0	1	1	0	0	0	1	1
0	1	0	0	0	0	1	0	0
0	1	0	1	0	0	1	0	1
0	1	1	0	0	0	1	1	0
0	1	1	1	0	0	1	1	1
1	0	0	0	0	1	0	0	0
1	0	0	1	0	1	0	0	1
1	0	1	0	1	0	0	0	0
1	0	1	1	1	0	0	0	1
1	1	0	0	1	0	0	1	0
1	1	0	1	1	0	0	1	1
1	1	1	0	1	0	1	0	0
1	1	1	1	1	0	1	0	1

Table 5.13 Truth table for 4-bit binary to BCD code convertor K-Map simplification:





Logic Diagram design:

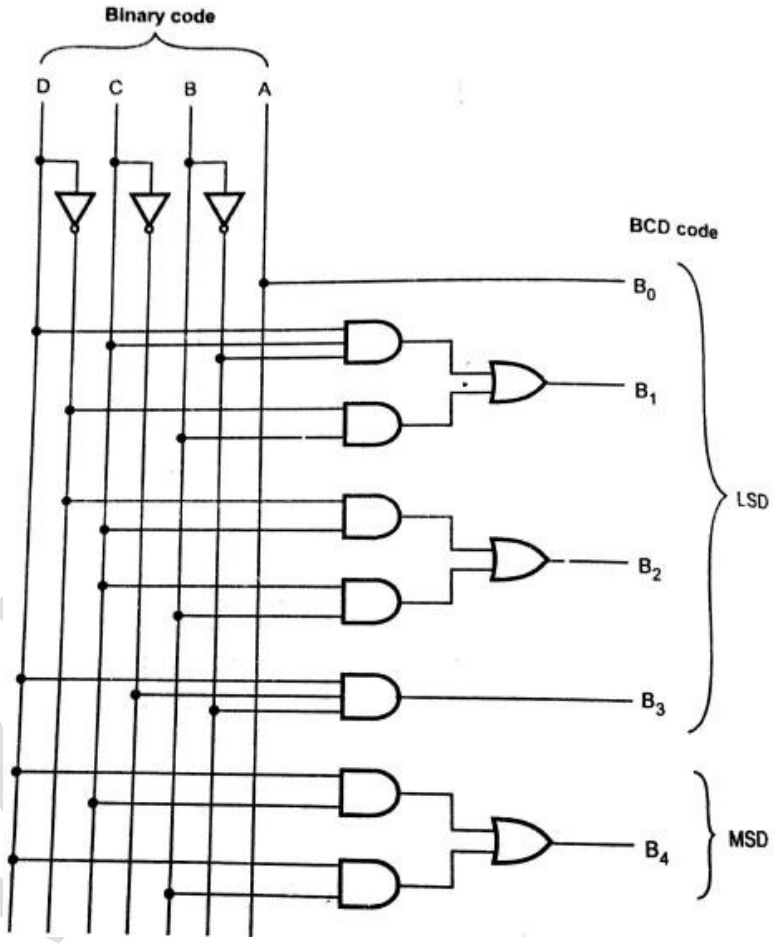


Figure 5.46: Logic Diagram of 4-bit binary to BCD code convertor

❖ **Design a BCD to Binary code convertor circuit?**

There are several types of binary codes are used in digital systems. Some of these codes are BCD, Excess-3, Gray and so on. Sometimes it may requires to change one code into another code format. The logic circuit used to convert one code into another code is

called as ‘Code Converter’.

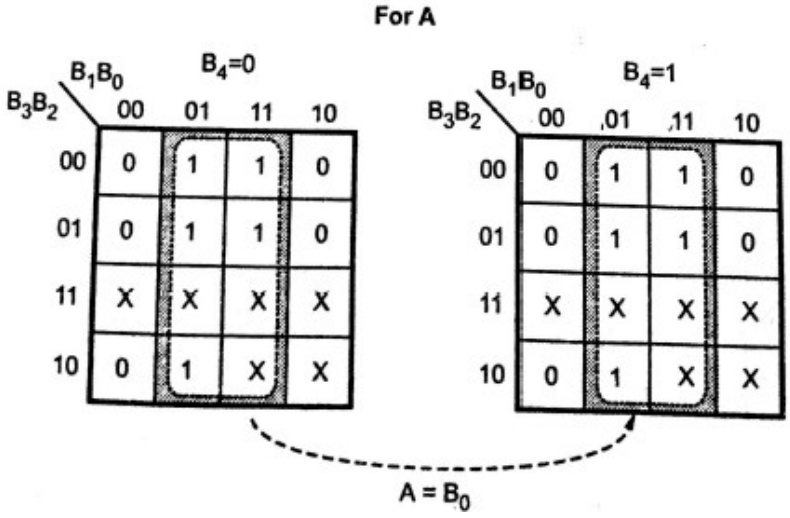
Design of BCD to Binary code convertor:

We know BCD has valid numbers only from 0 to 9. To make the values above the 9 as valid BCD numbers, a value '6' is added.

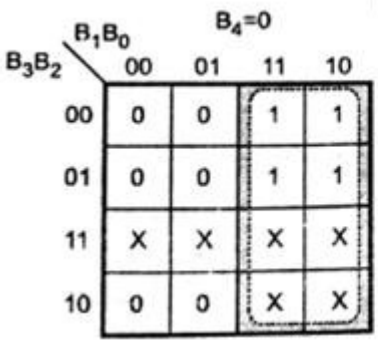
BCD code					Binary code				
B_4	B_3	B_2	B_1	B_0	E	D	C	B	A
0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	0	0	1	0
0	0	0	1	1	0	0	0	1	1
0	0	1	0	0	0	0	1	0	0
0	0	1	0	1	0	0	1	0	1
0	0	1	1	0	0	0	1	1	0
0	0	1	1	1	0	0	1	1	1
0	1	0	0	0	0	1	0	0	0
0	1	0	0	1	0	1	0	0	1
1	0	0	0	0	0	1	0	1	0
1	0	0	0	1	0	1	0	1	1
1	0	0	1	0	0	1	1	0	0
1	0	0	1	1	0	1	1	0	1
1	0	1	0	0	0	1	1	1	0
1	0	1	0	1	0	1	1	1	1
1	0	1	1	0	1	0	0	0	0
1	0	1	1	1	1	0	0	0	1
1	1	0	0	0	1	0	0	1	0
1	1	0	0	1	1	0	0	1	1

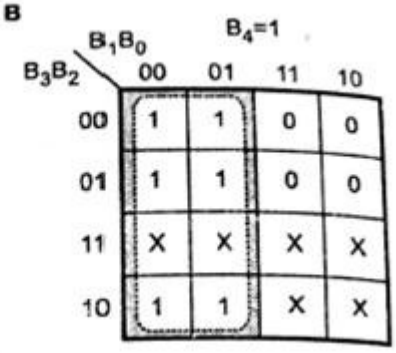
Table 5.14 Truth table for BCD to Binary code convertor

K-Map simplification:



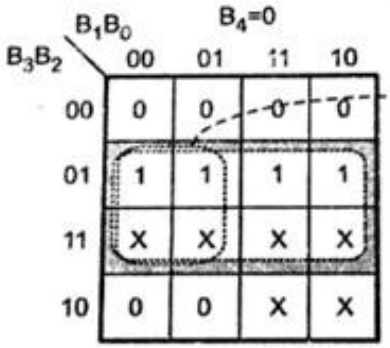
For B

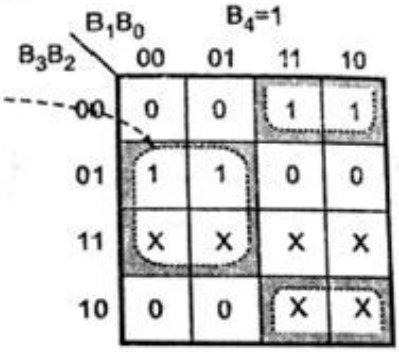




$B = B_1\overline{B}_4 + \overline{B}_1B_4$
$= B_1 \oplus B_4$

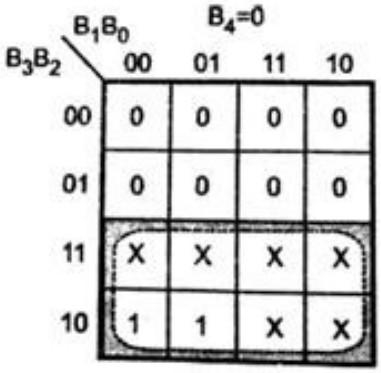
For C

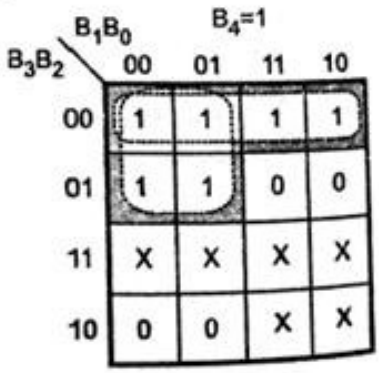




$C = \overline{B}_4B_2 + B_2\overline{B}_1 + B_4\overline{B}_2B_1$

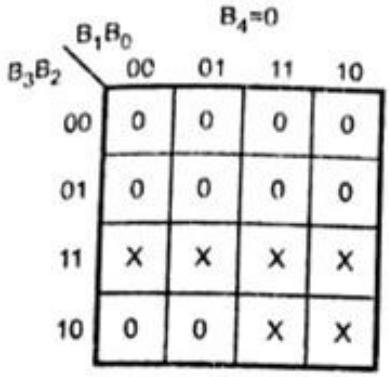
For D

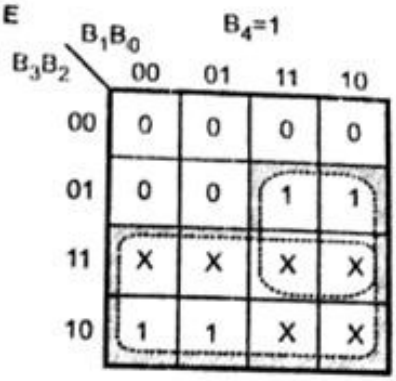




$D = \overline{B}_4B_3 + B_4\overline{B}_3\overline{B}_2 + B_4\overline{B}_3\overline{B}_1$

For E





$E = B_4B_3 + B_4B_2B_1$

Logic Diagram design:

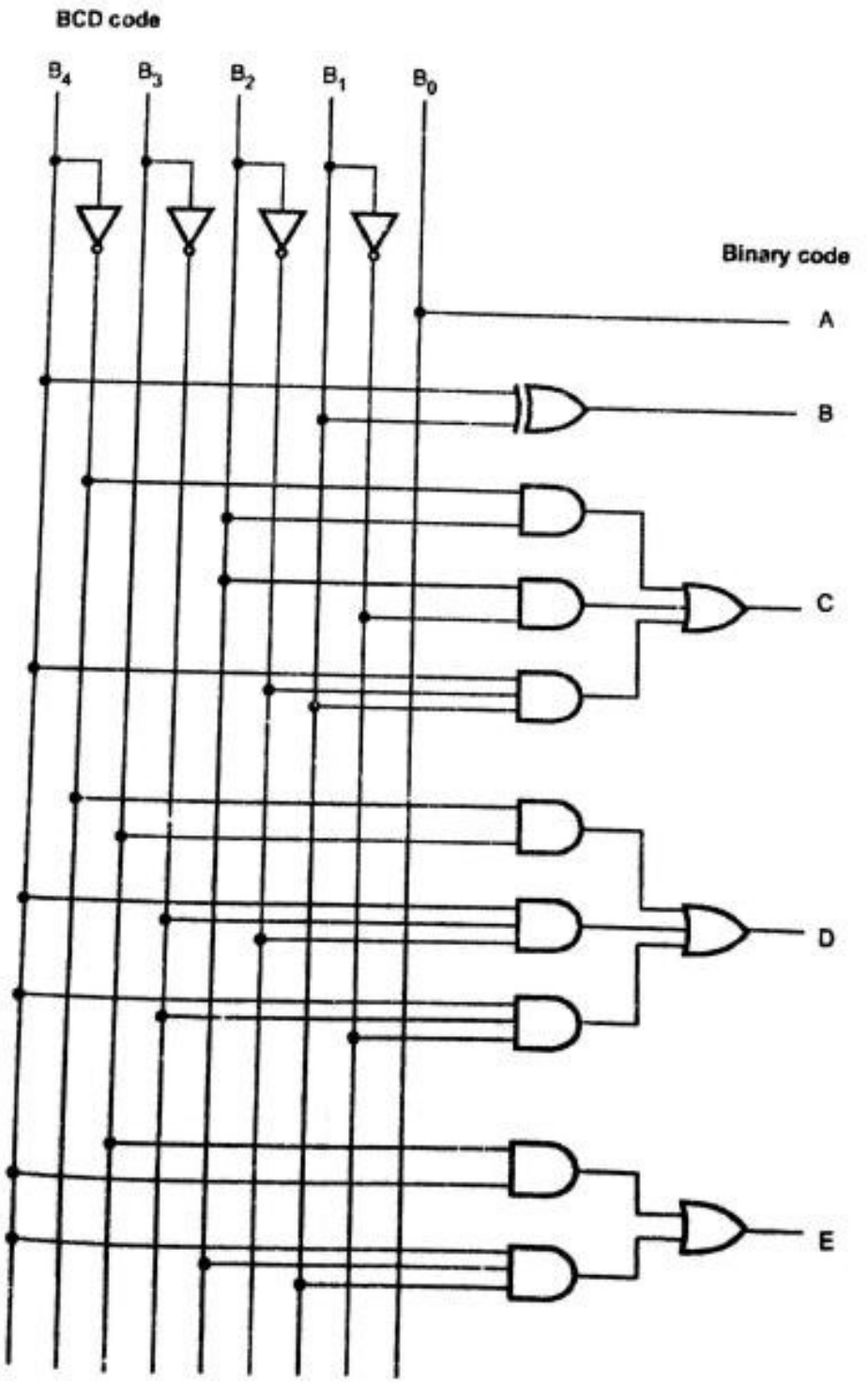


Figure 5.47: Logic Diagram of BCD to Binary code convertor

❖ **Design a BCD to Excess-3 code convertor circuit?**

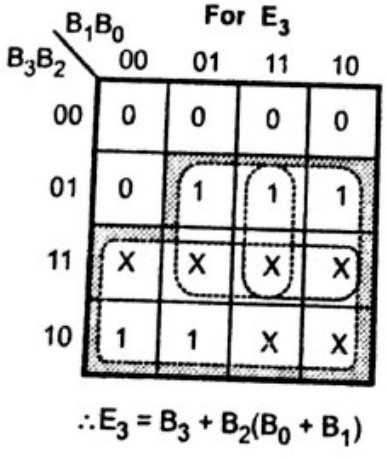
There are several types of binary codes are used in digital systems. Some of these codes are BCD, Excess-3, Gray and so on. Sometimes it may requires to change one code into another code format. The logic circuit used to convert one code into another code is called as 'Code Converter'.

Design of BCD to Excess-3 code convertor:

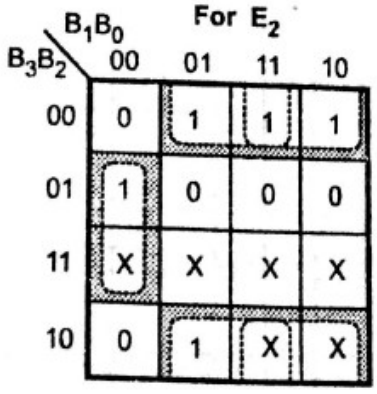
We know BCD has valid numbers only from 0 to 9. For each BCD number if we add a digit 3 then it will be converted into Excess-3 code. To get all valid BCD numbers we require four input variables, which may provide 16 combinations. In that 16 combinations only 0 to 9 are treated as valid BCD numbers. The combinations from 10 to 15 are represented as don't care values.

BCD code				Excess-3 code			
B_3	B_2	B_1	B_0	E_3	E_2	E_1	E_0
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

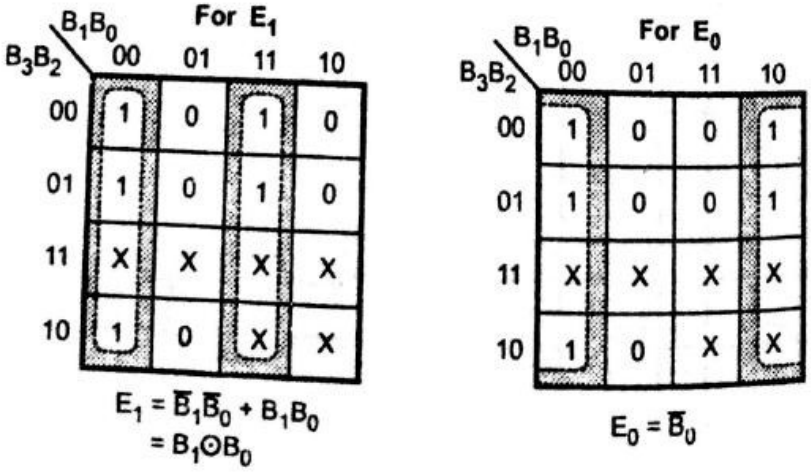
Table 5.15: Truth table for BCD to Excess-3 code convertor K-Map simplification:



$\therefore E_3 = B_3 + B_2(B_0 + B_1)$



$\therefore E_2 = B_2\bar{B}_1\bar{B}_0 + \bar{B}_2(B_0 + B_1)$



Logic Diagram design:

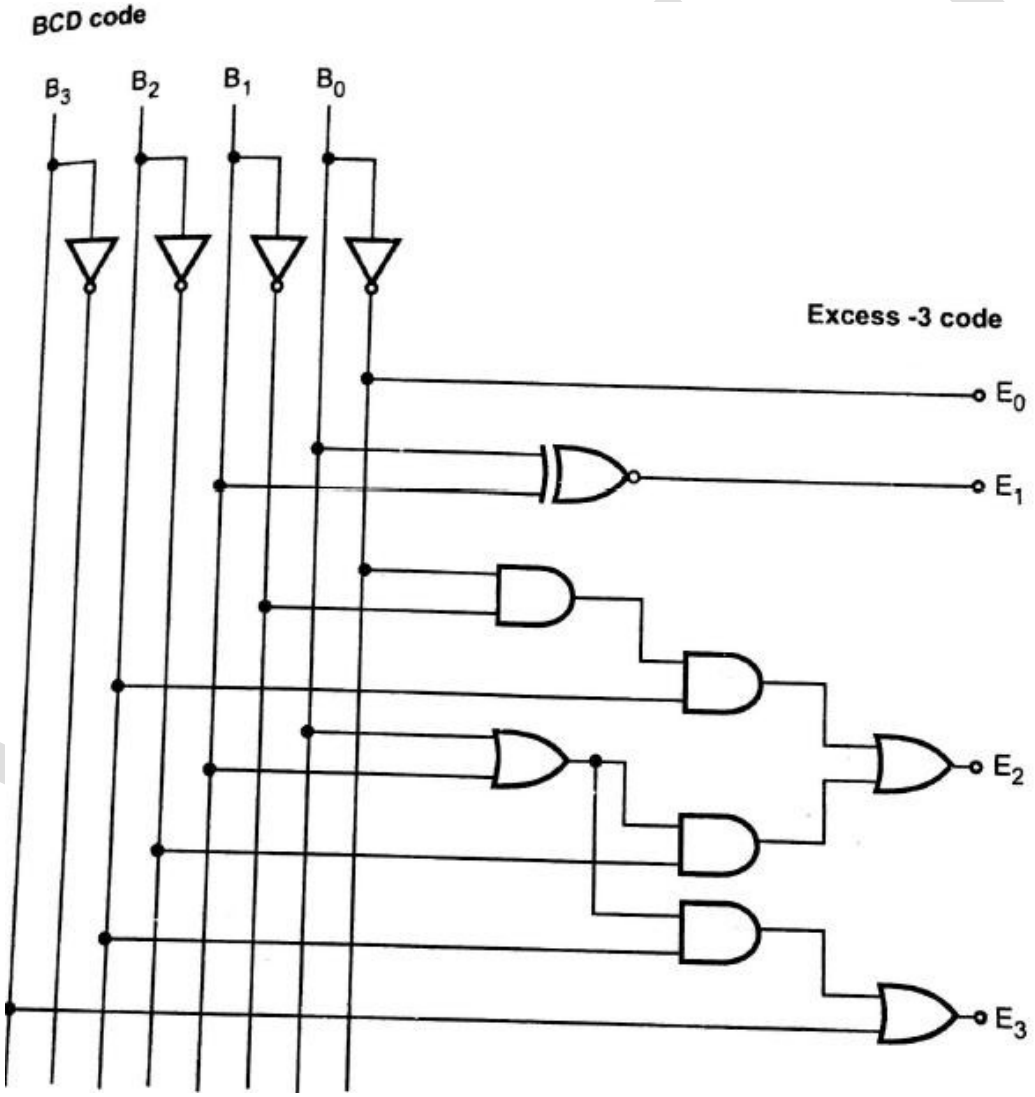


Figure 5.48: Logic Diagram of BCD to Excess-3 code convertor

❖ **Design a Excess-3 to BCD code convertor circuit?**

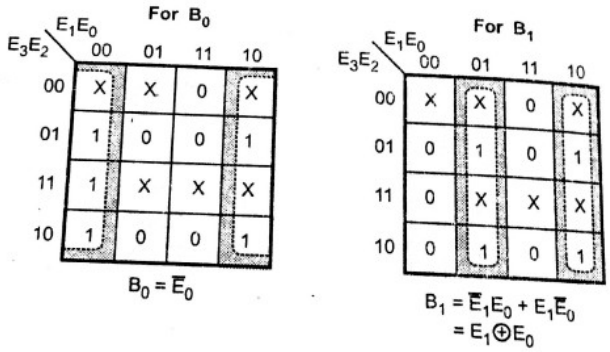
There are several types of binary codes are used in digital systems. Some of these codes are BCD, Excess-3, Gray and so on. Sometimes it may requires to change one code into another code format. The logic circuit used to convert one code into another code is called as 'Code Converter'.

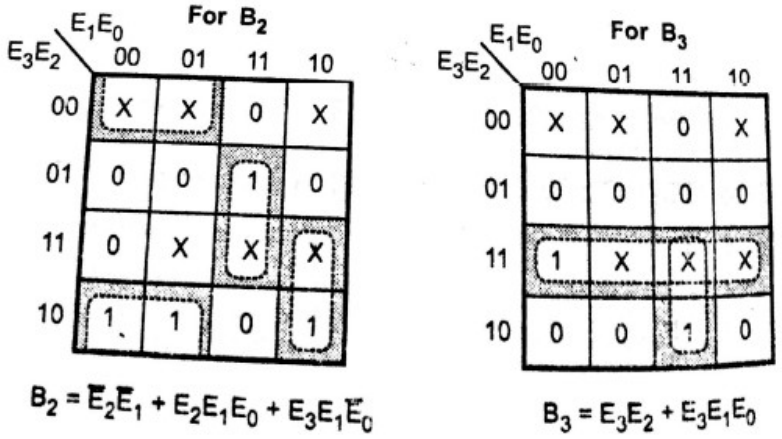
Design of Excess-3 to BCD code convertor:

We know BCD has valid numbers only from 0 to 9. For each BCD number if we add a digit 3 then it will be converted into Excess-3 code. To obtain valid BCD output we require four output variables. Excess-3 code starts from 3 and ends with 12 for BCD numbers. To get these values we require four input variables. The remaining combinations of four input variables such as 0, 1, 2, 13, 14 and 15 values will be treated as don't care conditions.

Excess-3 code				BCD code			
E_3	E_2	E_1	E_0	B_3	B_2	B_1	B_0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	1
0	1	0	1	0	0	1	0
0	1	1	0	0	0	1	1
0	1	1	1	0	1	0	0
1	0	0	0	0	1	0	1
1	0	0	1	0	1	1	0
1	0	1	0	0	1	1	1
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	1

Table 5.16: Truth table for Excess-3 to BCD code convertor K-Map simplification:





Logic Diagram design:

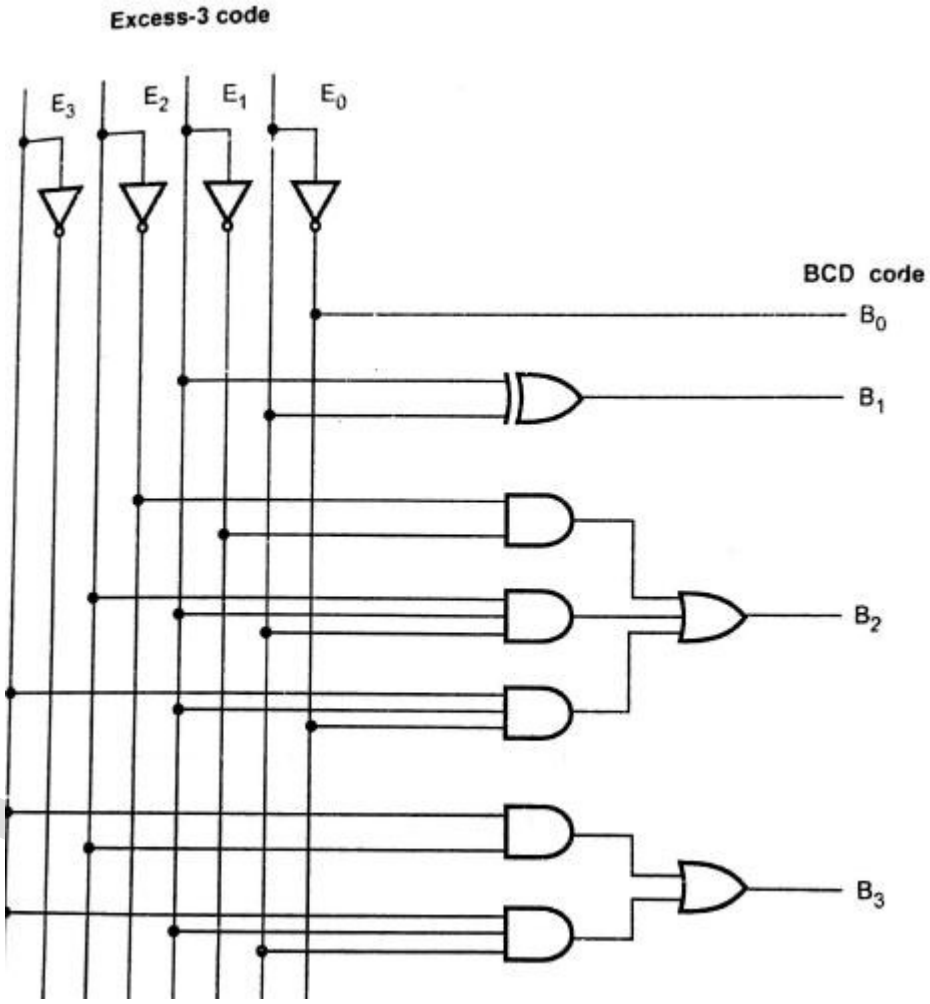


Figure 5.49: Logic Diagram of Excess-3 to BCD code convertor

❖ **Design a Binary to Gray code convertor circuit?**

The logic circuit used to convert one code into another code is called as 'Code Converter'.

Design of Binary to Gray code convertor:

Gray Code is used in digital circuits because it has an advantage that only one bit is changed between two successive numbers. In Gray code the first or MSB digit is obtained by taking MSB bit of corresponding binary numbers as it is and next digit in gray code is obtained by exclusive-or operation between two successive binary digits.

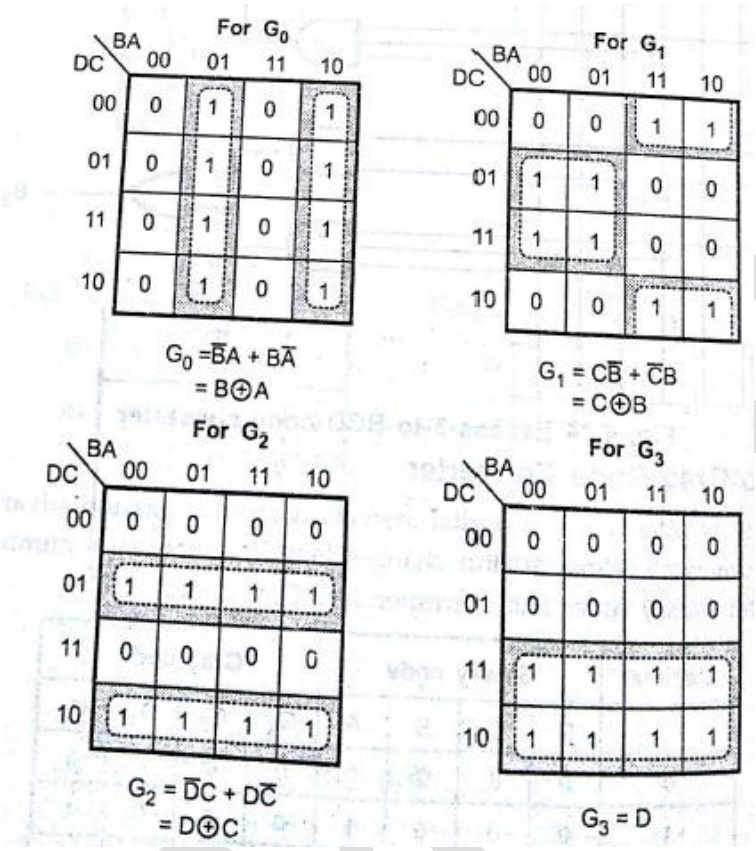
$$G_3 = D;$$
$$G_2 = D \oplus C;$$
$$G_1 = C \oplus B;$$
$$G_0 = B \oplus A;$$

Gray codes also generated by mirror method, in G_0 column take two successive digits as 0, 1 then put one mirror, it will be visible in reverse as 1, 0. And in G_1 column take four successive digits as 0, 0, 1, 1 then put one mirror, it will be visible in reverse as 1, 1, 0, 0. And in G_2 column take eight successive digits as 0, 0, 0, 0, 1, 1, 1, 1 then put one mirror, it will be visible in reverse as 1, 1, 1, 1, 0, 0, 0, 0. Then take MSB digit as 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1.

Binary code				Gray code			
D	C	B	A	G_3	G_2	G_1	G_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

Table 5.16: Truth table for Binary to Gray code convertor

K-Map simplification:



Logic Diagram design:

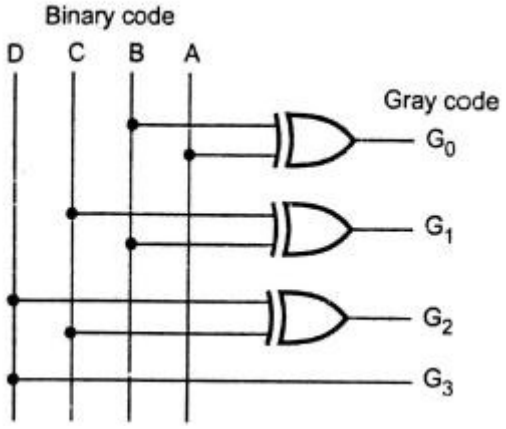


Figure 5.50: Logic Diagram of Binary to Gray code convertor

❖ **Design of Gray to Binary code convertor:**

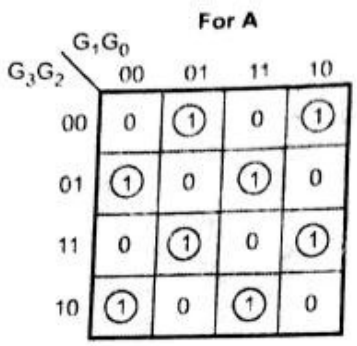
Gray Code is used in digital circuits because it has an advantage that only one bit is changed between two successive numbers. In Gray to Binary conversion the first or MSB digit in binary number is obtained by taking MSB bit of corresponding Gray code as it is and next

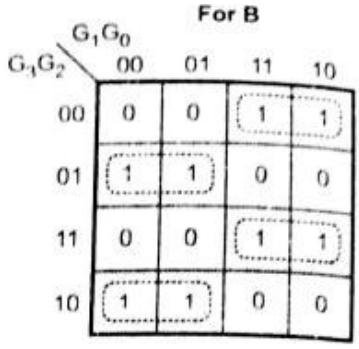
digit in binary code is obtained by exclusive-or operation between resultant and next binary digit.

$$D = G_3;$$
$$C = G_3 \oplus G_2;$$
$$B = G_2 \oplus G_1;$$
$$A = G_1 \oplus G_0;$$

Gray code				Binary code			
G_3	G_2	G_1	G_0	D	C	B	A
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	1	0	0	1	0
0	0	1	0	0	0	1	1
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	1
0	1	0	1	0	1	1	0
0	1	0	0	0	1	1	1
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	1
1	1	1	1	1	0	1	0
1	1	1	0	1	0	1	1
1	0	1	0	1	1	0	0
1	0	1	1	1	1	0	1
1	0	0	1	1	1	1	0
1	0	0	0	1	1	1	1

Table 5.17: Truth table for Gray to Binary code convertor K-Map simplification:





$$A = (\overline{G}_3G_2 + G_3\overline{G}_2)\,\overline{G}_1\overline{G}_0 + (\overline{G}_3\,\overline{G}_2 + G_3G_2)\,\overline{G}_1G_0$$
$$+ (\overline{G}_3G_2 + G_3\,\overline{G}_2)\,G_1G_0 + (\overline{G}_3\overline{G}_2 + G_3G_2)\,G_1\,\overline{G}_0$$
$$= (G_3 \oplus G_2)\,\overline{G}_1\overline{G}_0 + (G_3 \odot G_2)\overline{G}_1G_0$$
$$+ (G_3 \oplus G_2)\,G_1G_0 + (G_3 \odot G_2)\,G_1\overline{G}_0$$
$$= (G_3 \oplus G_2)\,(\overline{G}_1\overline{G}_0 + G_1G_0) + (G_3 \odot G_2)\,(\overline{G}_1G_0 + G_1\overline{G}_0)$$
$$= (G_3 \oplus G_2)\,(G_1 \odot G_0) + (G_3 \odot G_2)\,(G_1 \oplus G_0)$$
$$= (G_3 \oplus G_2)\,\overline{(G_1 \oplus G_0)} + \overline{(G_3 \oplus G_2)}\,(G_1 \oplus G_0)$$
$$= (G_3 \oplus G_2) \oplus (G_1 \oplus G_0)$$

$$B = (\overline{G}_3\overline{G}_2 + G_3G_2)G_1 + (\overline{G}_3G_2 + G_3\overline{G}_2)\overline{G}_1$$
$$= (G_3 \odot G_2)G_1 + (G_3 \oplus G_2)\overline{G}_1$$
$$= \overline{(G_3 \oplus G_2)}\,G_1 + (G_3 \oplus G_2)\,\overline{G}_1$$
$$= G_3 \oplus G_2 \oplus G_1$$

For C

G_3G_2 \ G_1G_0	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	0	0	0	0
10	1	1	1	1

For D

G_3G_2 \ G_1G_0	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	1	1	1
10	1	1	1	1

$$C = \overline{G}_3G_2 + G_3\overline{G}_2 = G_3 \oplus G_2 \qquad D = G_3$$

Logic Diagram design:

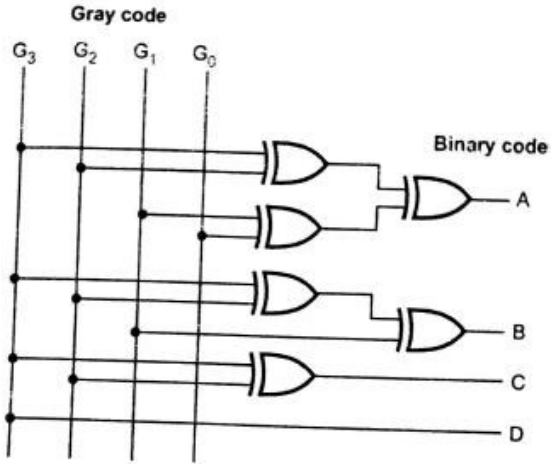


Figure 5.51: Logic Diagram of Gray to Binary code convertor

❖ **Design a BCD to Gray code convertor circuit?**

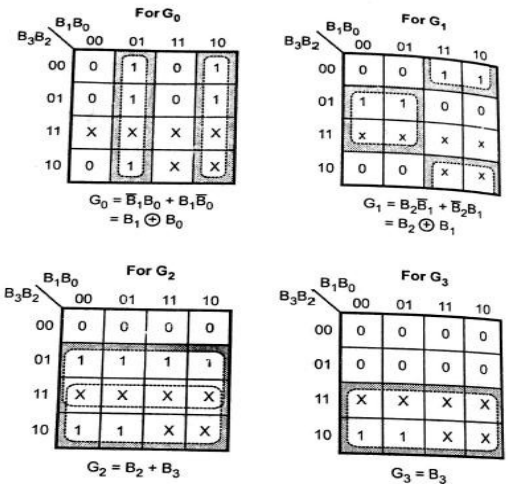
There are several types of binary codes are used in digital systems. Some of these codes are BCD, Excess-3, Gray and so on. Sometimes it may requires to change one code into another code format. The logic circuit used to convert one code into another code is called as 'Code Converter'.

Designing of BCD to Gray code convertor:

BCD code				Gray code			
B_3	B_2	B_1	B_0	G_3	G_2	G_1	G_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1

Table 5.18: Truth table for BCD to Gray code convertor

K-Map simplification:



Logic Diagram design:

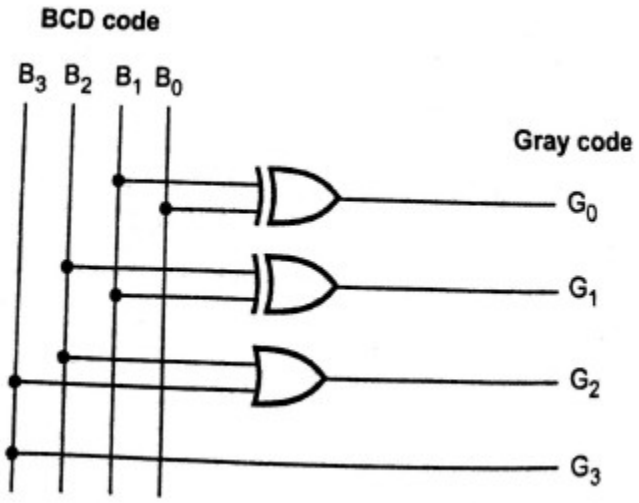


Figure 5.52: Logic Diagram of BCD to Gray code convertor

❖ **What is the need of parity in digital transmission? Design the parity generator circuit?**

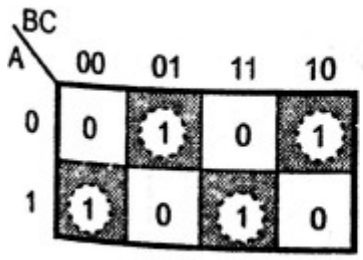
A parity bit is an extra added bit to the binary information bits used to detect the errors during transmission. The parity bit does not carry any information. It included in binary information at the time of transmission to make the number of 1s either even or odd. The message included with parity bit is transmitted and checked at the receiver end for errors. An error will be detected if the received message is not corresponds to the transmitted one. The circuit that generates the parity bit at the transmitter is called Parity Generator and the circuit that checks the parity in the receiver is called Parity Checker.

In even parity the added parity bit makes total number of 1's as even amount. In odd parity the added parity bit makes total number of 1's as odd amount.

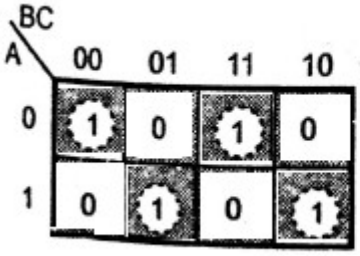
3-bit Message			Odd Parity Bit	Even Parity Bit
A	B	C	Po	Pe
0	0	0	1	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	0	1

Table 5.19: Truth table for 3-bit parity generator

K-Map simplification:



$$P_e = \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C} + ABC$$
$$= \bar{A}(\bar{B}C + B\bar{C}) + A(\bar{B}\bar{C} + BC)$$
$$= \bar{A}(B \oplus C) + A(B \odot C)$$
$$= \bar{A}(B \oplus C) + A\overline{(B \oplus C)}$$
$$= A \oplus (B \oplus C)$$



$$P_o = \bar{A}\bar{B}\bar{C} + \bar{A}BC + A\bar{B}C + AB\bar{C}$$
$$= \bar{A}(\bar{B}\bar{C} + BC) + A(\bar{B}C + B\bar{C})$$
$$= \bar{A}(B \odot C) + A(B \oplus C)$$
$$= \bar{A}\overline{(B \oplus C)} + A(B \oplus C)$$
$$= A \odot (B \odot C)$$

Logic Diagram design:

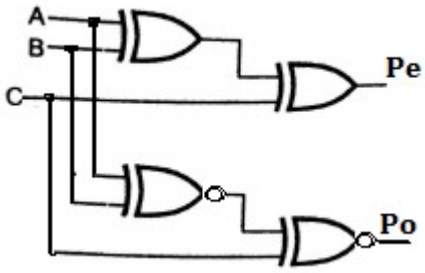


Figure 5.53: Logic Diagram of even and odd parity generator

Figure shows the pin diagram of a 7486, a TTL quad 2-input EX-OR gate. This digital integrated circuit contains four 2-input EX-OR gates inside a 14 pin dual-in-line package.

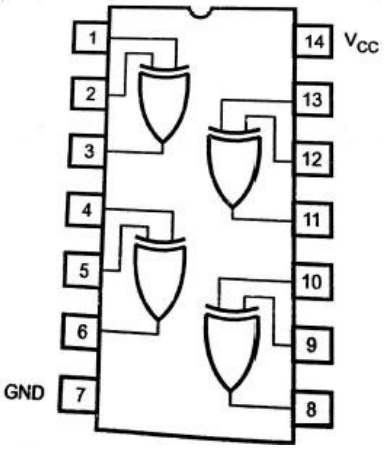


Figure 5.54: Pin Diagram of Ex-OR gate

❖ **What is the need of parity in digital transmission? Design the parity error checker circuit?**

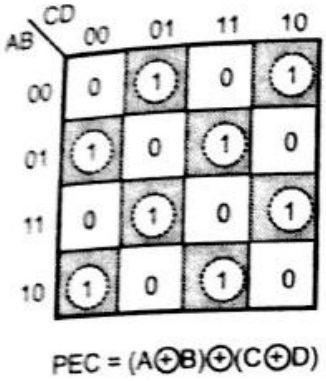
A parity bit is an extra added bit to the binary information bits used to detect the errors during transmission. The parity bit does not carry any information. It included in binary information at the time of transmission to make the number of 1s either even or odd. The message included with parity bit is transmitted and checked at the receiver end for errors. An error will be detected if the received message is not corresponds to the transmitted one. The circuit that generates the parity bit at the transmitter is called Parity Generator and the circuit that checks the parity in the receiver is called Parity Checker.

In even parity the added parity bit makes total number of 1's as even amount. In odd parity the added parity bit makes total number of 1's as odd amount. Parity Error Checker circuit verifies is there any error bit in received data or not. If received data stream has number of 1's as even number then there is no error. If it has odd number of 1's then Parity Error checker(PEC) gives a value 1, which means there is an error.

Four Bits Received				Parity Error Check
A	B	C	D	PEC
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Table 5.20: Truth table for 4-bit Parity Error Checker

K-Map simplification:



$$PEC = \bar{A}\bar{B}(\bar{C}D + C\bar{D}) + \bar{A}B(\bar{C}\bar{D} + CD) + AB(\bar{C}D + C\bar{D}) + A\bar{B}(\bar{C}\bar{D} + CD)$$

$$= \bar{A}\bar{B}(C \oplus D) + \bar{A}B(\overline{C \oplus D}) + AB(C \oplus D) + A\bar{B}(\overline{C \oplus D})$$

$$= (\bar{A}\bar{B} + AB)(C \oplus D) + (\bar{A}B + A\bar{B})(\overline{C \oplus D})$$

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

$$= (A \oplus B) \oplus (C \oplus D)$$

Logic Diagram design:

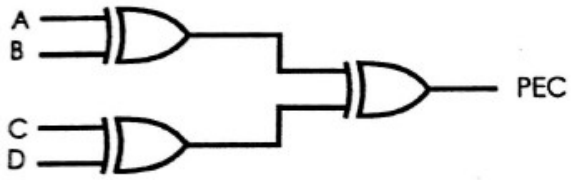


Figure 5.55: Logic Diagram of Parity Error Checker

❖ **Explain the pin configuration and internal structure of IC 74XX280?**

The 74XX280 is a 9-bit parity generator or checker commonly used to detect errors in high speed data transmission or data retrieval systems. Both **even** and **odd** parity outputs are available for generating or checking even or odd parity on up to 9-bits.

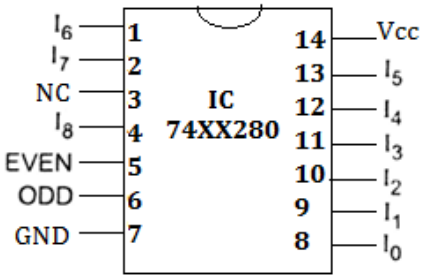


Figure 5.56: Pin Diagram of 9-bit Parity Generator or Checker

These universal 9-bit parity generators/checkers utilize advanced Schottky high-performance circuitry and feature odd (Σ ODD) and even (Σ EVEN) outputs to facilitate operation of either odd- or even-parity applications. The word-length capability is easily expanded by cascading. These devices can be used to upgrade the performance of most

systems utilizing the SN74ALS180 and SN74AS180 parity generators/checkers. Although the SN74ALS280 and

SN74AS280 are implemented without expander inputs, the corresponding function is provided by the availability of an input (I) at terminal 4 and the absence of any internal connection at terminal 3. This permits the SN74ALS280 and SN74AS280 to be substituted for the SN74ALS180 and SN74AS180 in existing designs to produce an identical function even if the devices are mixed with existing SN74ALS180 and SN74AS180 devices. The SN74ALS280 and SN74AS280 are characterized for operation from 0°C to 70°C.

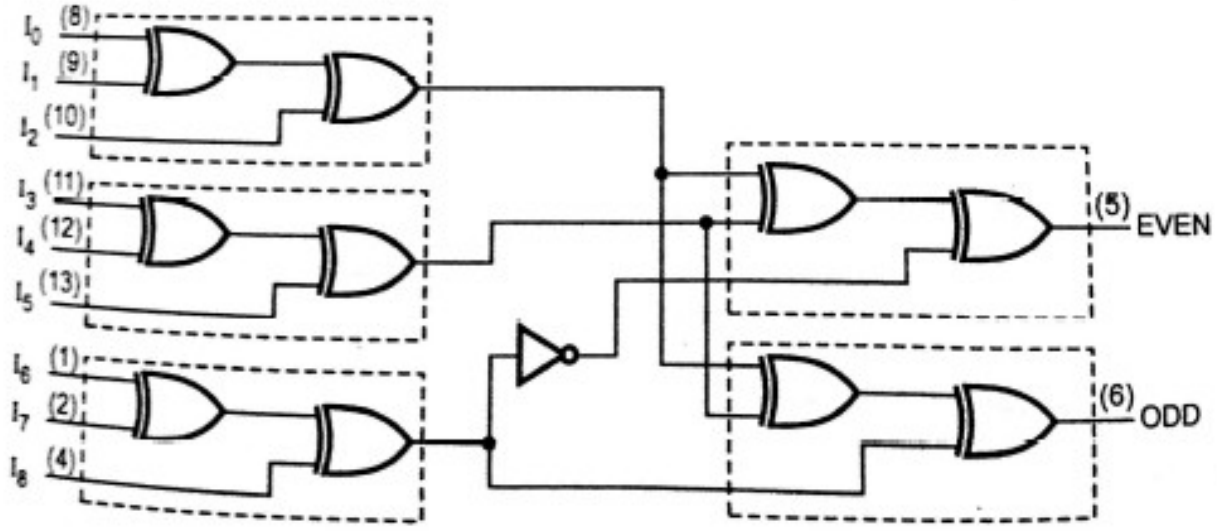


Figure 5.57: Logic Diagram for IC74xx280 (9-bit Parity Generator or Checker)

❖ **Design an error correcting circuit for 7 bit hamming code using IC 74xx280?**

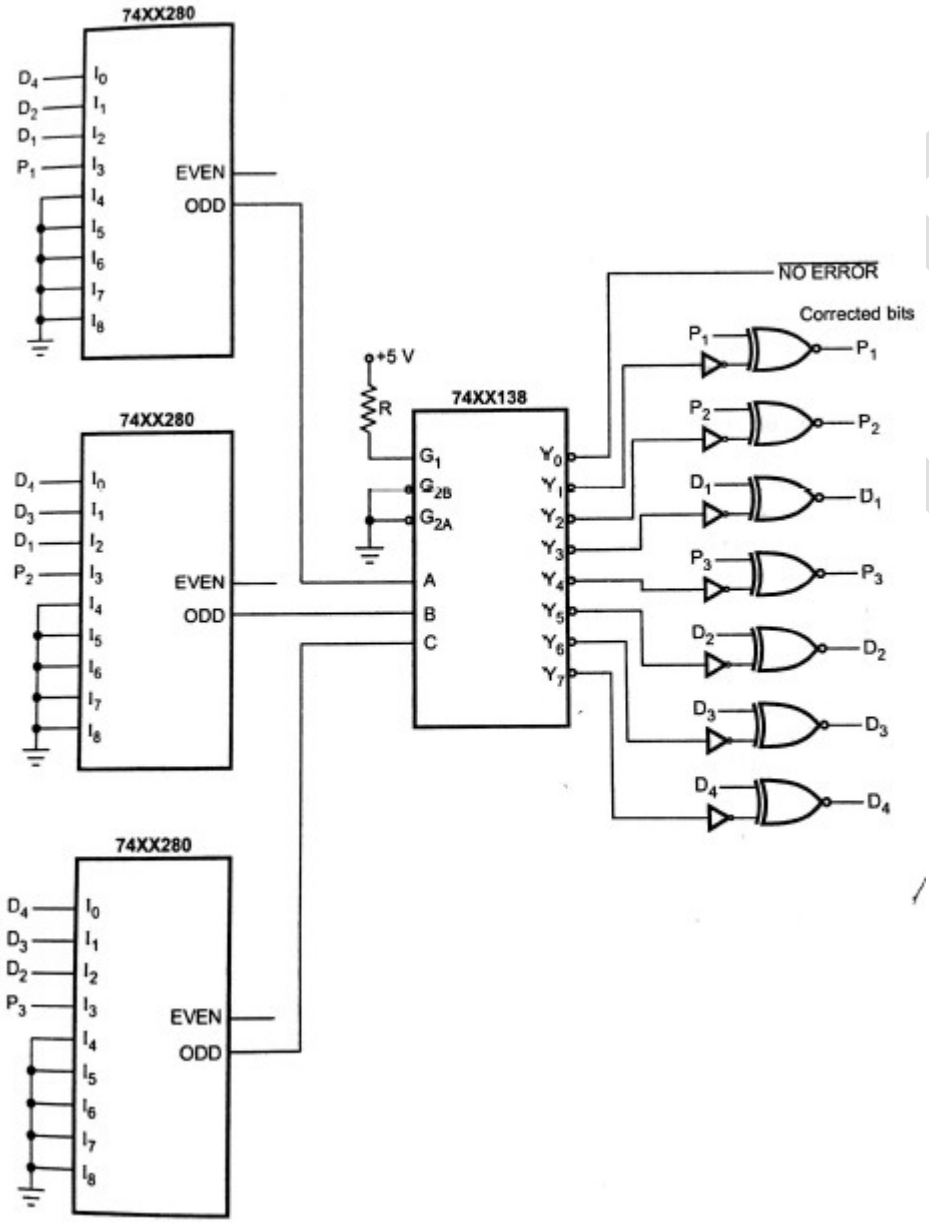


Figure 5.58: Logic Diagram for 7 bit hamming code using IC 74xx280

❖ **What is comparator? Design a 2-bit comparator using gates?**

A comparator is a special combinational circuit designed to compare the relative magnitude of two binary numbers. The following figure shows an n-bit, it receives two n-bit numbers A and B as inputs and gives outputs as A>B, A=B, A<B. Depending upon the relative magnitudes of two numbers one of the output is high.

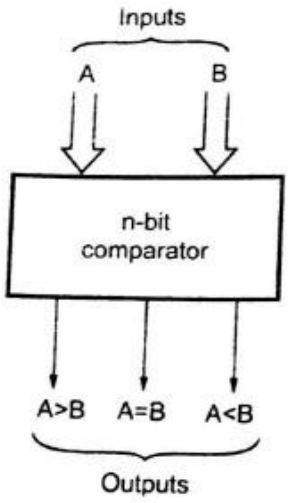
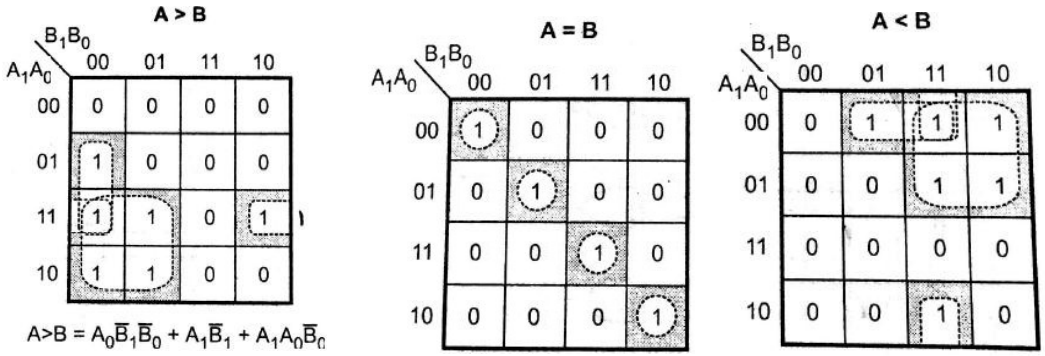


Figure 5.59: Block Diagram of n-bit comparator Designing of 2-bit comparator using gates:

Inputs				Outputs		
A_1	A_0	B_1	B_0	A > B	A = B	A< B
0	0	0	0	0	1	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	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	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	0	1	1	0	0	1
1	1	0	0	1	0	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	0	1	0

Table 5.21: Truth table for 2-bit Comparator

K-Map Simlification:



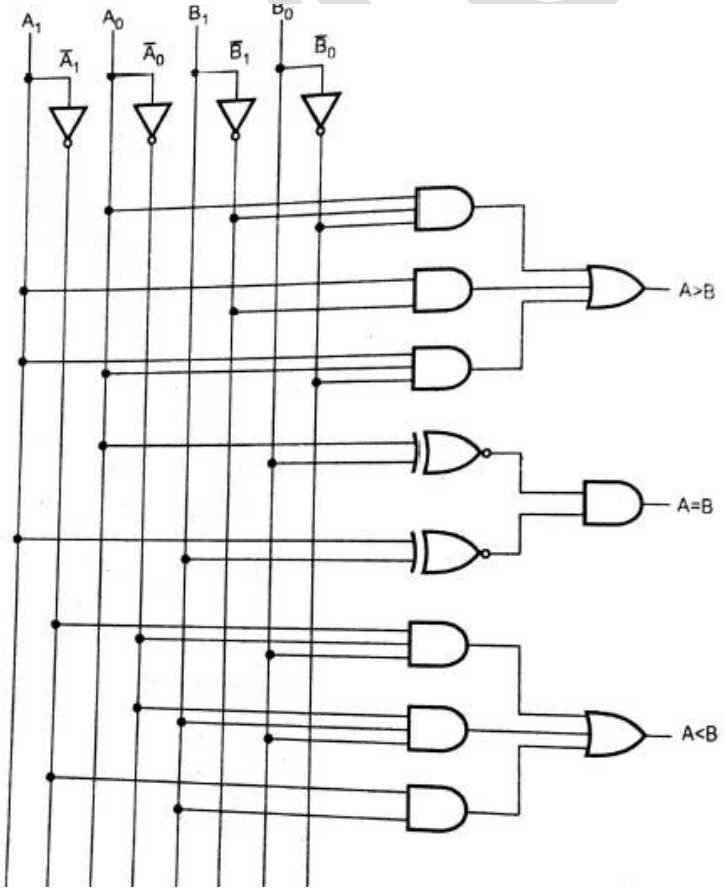
$$(A = B) = \overline{A}_1 \overline{A}_0 \overline{B}_1\overline{B}_0 + \overline{A}_1 A_0 \overline{B}_1 B_0 + A_1 A_0 B_1 B_0 + A_1 \overline{A}_0 B_1 \overline{B}_0$$

$$= \overline{A}_1\overline{B}_1 (\overline{A}_0\overline{B}_0 + A_0B_0) + A_1B_1 (A_0B_0 + \overline{A}_0\overline{B}_0)$$

$$= (A_0 \odot B_0)\,(A_1 \odot B_1)$$

$$(A < B) = \overline{A}_1 A_0 \overline{B}_0 + A_0 \overline{B}_1\overline{B}_0 + \overline{A}_1 B_1$$

Figure 5.60: Logic diagram of 2-bit comparator



❖ **What is comparator? Explain pin configuration and operation of IC7485?**

Comparator applications are common enough that several MSI comparators have been developed commercially. The *74x85* is a 4-bit comparator. It provides a greater-than output (AGTBOUT) and a less-than output (ALTBOUT) as well as an equal output (AEQBOUT). The *74x85* also has *cascading inputs* (AGTBIN, ALTBIN, AEQBIN) for combining multiple '85s to create comparators for more than four bits. Both the cascading inputs and the outputs are arranged in a 1-out-of-3 code, since in normal operation exactly one input and one output should be asserted. The cascading inputs are defined so the outputs of an '85 that compares less-significant bits are connected to the inputs of an '85 that compares more- significant bits.

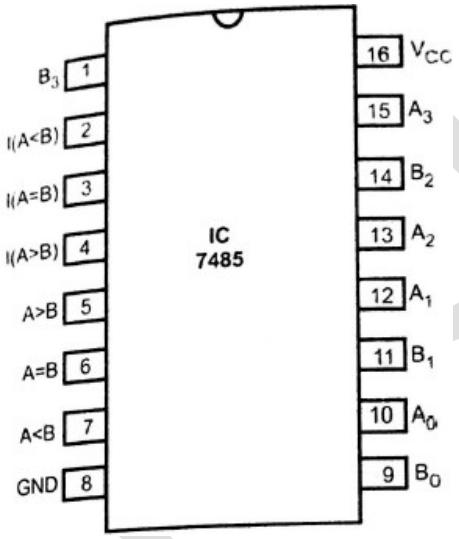


Figure 5.61: Pin diagram of IC 74x85 (4-bit comparator)

Comparing Inputs	Cascading Inputs			Outputs		
A B	I(A > B)	I(A = B)	I(A < B)	A > B	A = B	A < B
A > B	X	X	X	1	0	0
A = B	1	0	0	1	0	0
	X	1	X	0	1	0
	0	0	1	0	0	1
	0	0	0	1	0	1
	1	0	1	0	0	0
A < B	X	X	X	0	0	1

Table 5.22: Truth table for IC 74x85 (4-bit comparator)

❖ **Cascade a comparator**

Comparator applications are common enough that several MSI comparators have been developed commercially. The *74x85* is a 4-bit comparator. It provides a greater-than output (AGTBOUT) and a less-than output (ALTBOUT) as well as an equal output (AEQBOUT). The *74x85* also has *cascading inputs* (AGTBIN, ALTBIN, AEQBIN) for combining multiple '85s to create comparators for more than four bits. Both the cascading inputs and the outputs are arranged in a 1-out-of-3 code, since in normal operation exactly one input and one output should be asserted. The cascading inputs are defined so the outputs of an '85 that compares less-significant bits are connected to the inputs of an '85 that compares more- significant bits.

cascading outputs roughly according to the following pseudo-logic equations:

$$AGTBOUT = (A > B) + (A = B) \cdot AGTBIN$$

$$AEQBOUT = (A = B) \cdot AEQBIN$$

$$ALTBOUT = (A < B) + (A = B) \cdot ALTBIN$$

Example1:

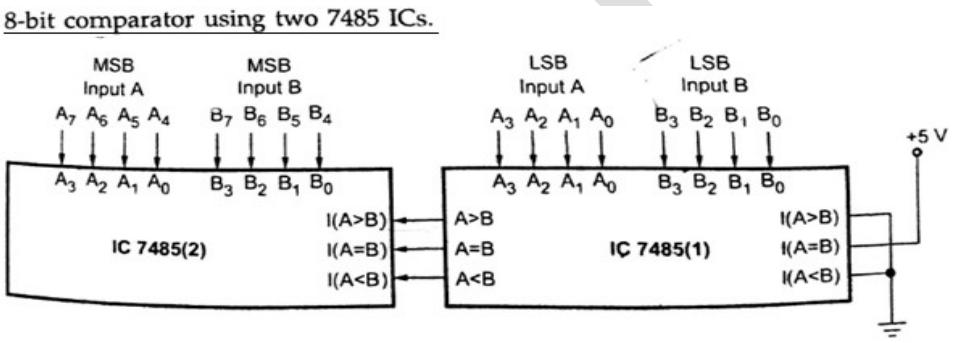


Figure 5.62: Design of 8-bit comparator using two 74X85

Example2:

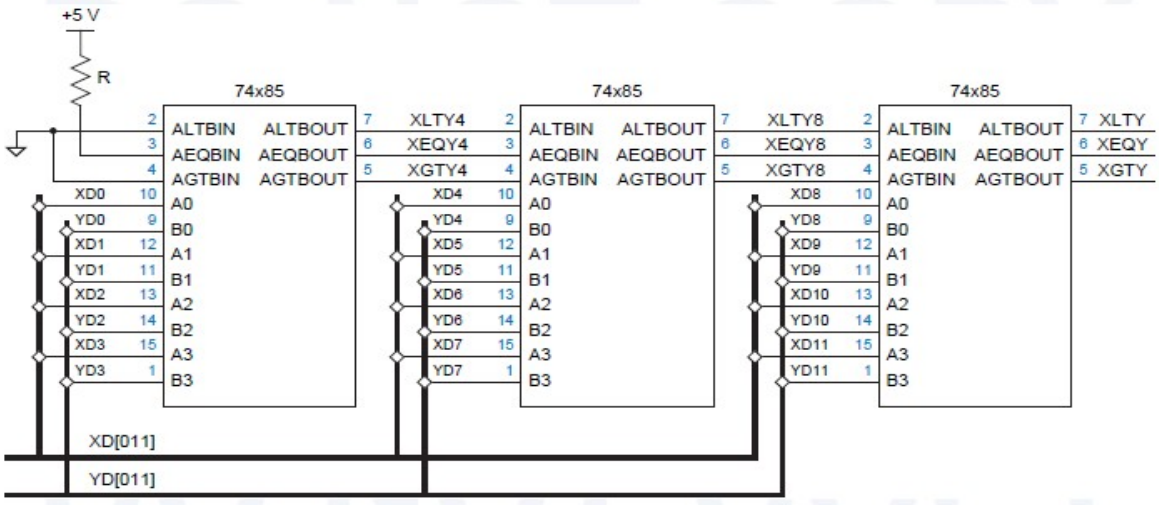


Figure 5.63: Design of 12-bit comparator using three 74X85

The cascading using IC74x85 is very easy because it has cascading inputs. We have one more comparator IC 74x682 is an 8-bit comparator which is not having any cascaded input. But still it can be used to cascade with another 74x682 to form larger comparator.

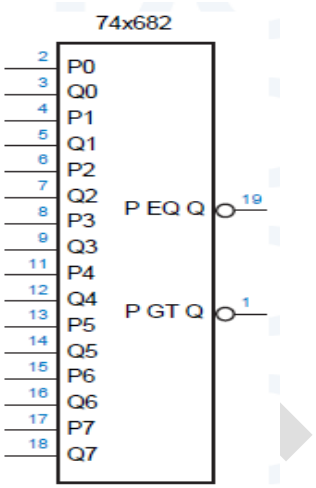


Figure 5.64Pin diagram of IC74X682 (8-bit comparator)

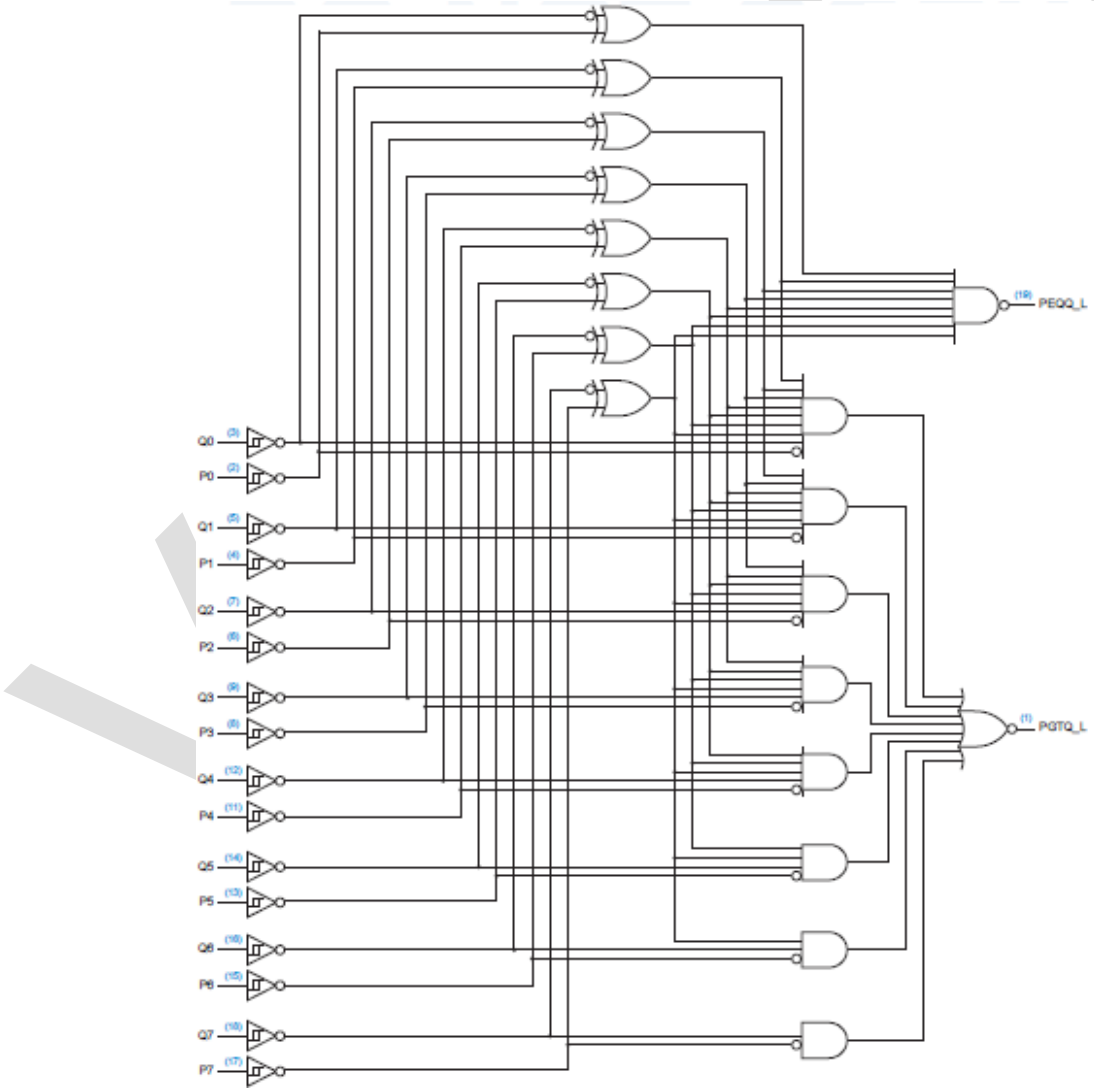


Figure 5.65: Logic diagram of IC 74x682

Example3: Design of 24-bit comparator using three 74X682

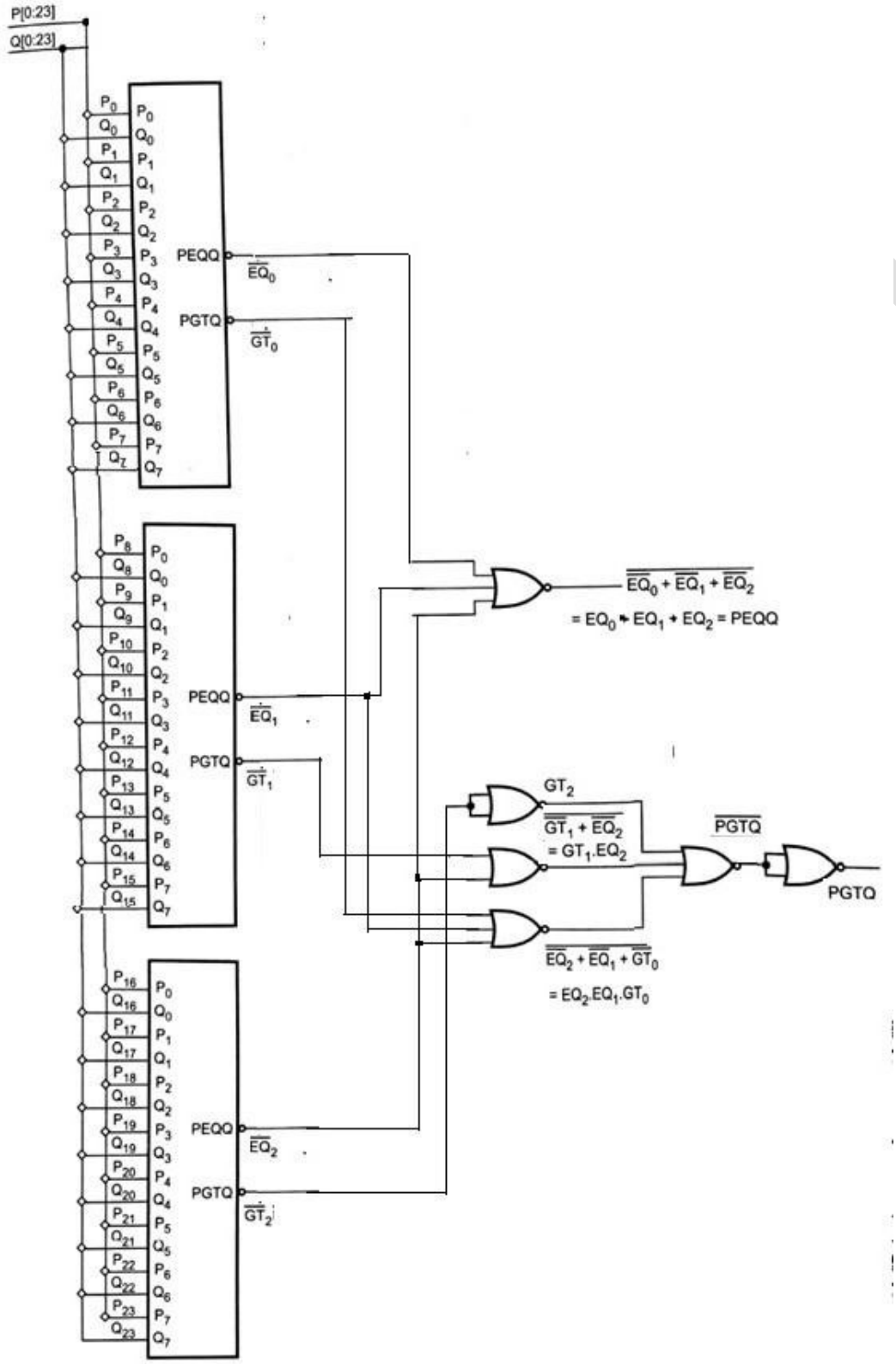


Figure 5.66: Design of 24-bit comparator using three 74X682

❖ **Design and explain the following logic circuits**

(i) Half Adder (ii)Full Adder (iii)Half Subtractor (iv) Full Subtractor

Digital circuits performs various basic arithmetic operations such as Additions, Subtractions, Multiplications and Division. The simple Addition operation has four possible elementary operations.

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 10_2$$

The first three operations produce a sum whose length is one digit, but when the last operation is performed sum is two digits. The higher significant bit of this result is called a **carry**, and lower significant bit is called **sum**. The logic circuit which performs this operation is called a **half-adder**. The circuit which performs addition of three bits (two significant bits and a previous carry) is a **full-adder**. Let us see the logic circuits to perform half-adder and full-adder operations.

(i) Half Adder:

The half-adder operation needs two binary inputs : augend and addend bits; and two binary outputs : sum and carry. The truth table gives the relation between input and output variables for half-adder operation.

Inputs		Outputs	
A	B	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Table 5.23: Truth table for Half Adder

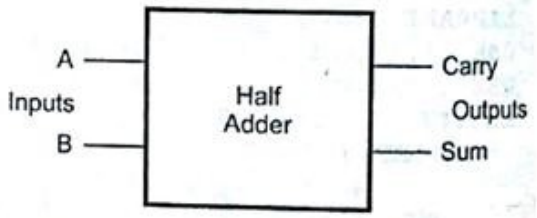


Figure 5.67: Block diagram of Half Adder

K-Map Simplification:

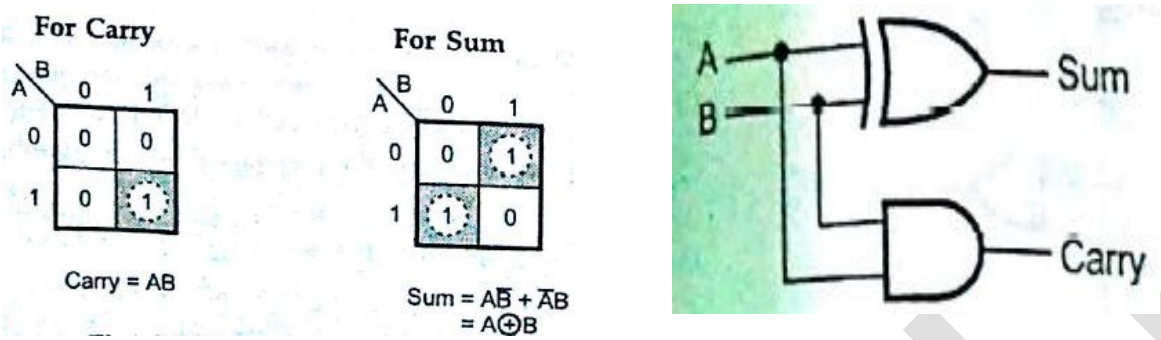


Figure 5.68: Logic diagram for Half Adder

Limitations of Half Adder: In multidigit addition we have to add teo bits alongwith the carry of previous digit addition, Such additions requires addition of three bits. This is not possible with Half Adder. Hence in practical Half Adders are not used.

(ii)Full Adder:

A full-adder is a combinational circuit that forms the arithmetic sum of three input bits. It consists of three inputs and two outputs. Two of the input variables, denoted by A and B, represent the two significant bits to be added. The third input C_{in}, represents the carry from the previous lower significant position.

Inputs			Outputs	
A	B	C_{in}	Carry	Sum
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

Table 5.24: Truth table for Full Adder

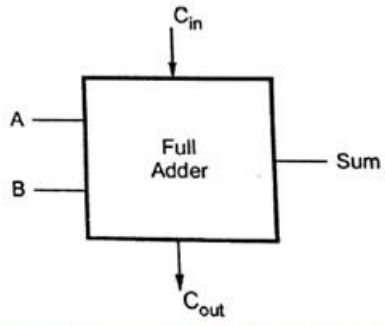
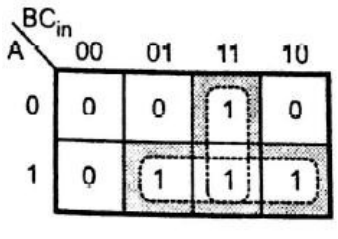
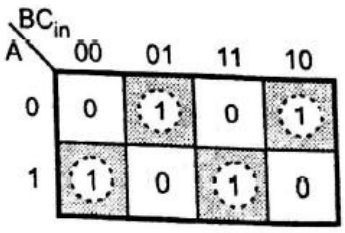


Figure 5.69: Block diagram of Full Adder



$C_{out} = AB + A\,C_{in} + B\,C_{in}$



$Sum = \bar{A}\,\bar{B}C_{in} + \bar{A}B\bar{C}_{in} + A\bar{B}\,\bar{C}_{in} + ABC_{in}$

Sum expression can be simplified as

$$\begin{aligned}\text{Sum} &= \bar{A}\,\bar{B}\,C_{in} + \bar{A}\,B\,\bar{C}_{in} + A\,\bar{B}\,\bar{C}_{in} + A\,B\,C_{in} \\ &= C_{in}\,(\bar{A}\,\bar{B} + AB) + \bar{C}_{in}\,(\bar{A}\,B + A\,\bar{B}) \\ &= C_{in}\,(A \odot B) + \bar{C}_{in}\,(A \oplus B) = C_{in}\,\overline{(A \oplus B)} + \bar{C}_{in}\,(A \oplus B) \\ &= C_{in} \oplus (A \oplus B)\end{aligned}$$

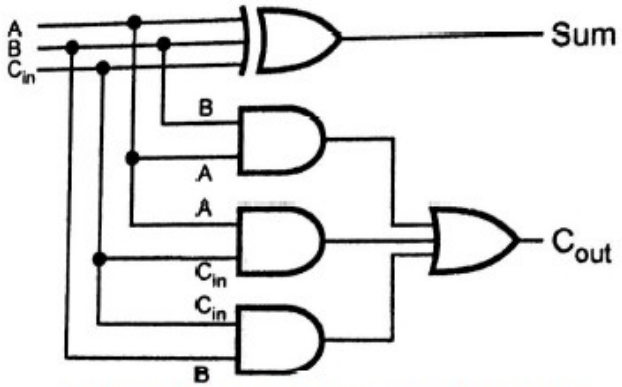


Figure 5.70: Logic diagram for Full Adder

Full Adder can also be implemented using two half adders.

$$\begin{aligned}\text{Sum} &= C_{in} \oplus (A \oplus B) \\ &= C_{in} \oplus (A\,\bar{B} + \bar{A}\,B) \\ &= C_{in}\,\overline{(A\,\bar{B} + \bar{A}\,B)} + \bar{C}_{in}\,(A\,\bar{B} + \bar{A}\,B) \\ &= C_{in}\,(\overline{A\,\bar{B}} \cdot \overline{\bar{A}\,B}) + \bar{C}_{in}\,(A\,\bar{B} + \bar{A}\,B) \\ &= C_{in}\,[(\bar{A} + B) \cdot (A + \bar{B})] + \bar{C}_{in}\,(A\,\bar{B} + \bar{A}\,B) \\ &= C_{in}\,(\bar{A}\,\bar{B} + A\,B) + \bar{C}_{in}\,(A\,\bar{B} + \bar{A}\,B) \\ &= \bar{A}\,\bar{B}\,C_{in} + A\,B\,C_{in} + A\,\bar{B}\,\bar{C}_{in} + \bar{A}\,B\,\bar{C}_{in} \\ &= \bar{A}\,\bar{B}\,C_{in} + \bar{A}\,B\,\bar{C}_{in} + A\,\bar{B}\,\bar{C}_{in} + A\,B\,C_{in}\end{aligned}$$

$$\begin{aligned}C_{out} &= AB + A\,C_{in} + BC_{in} \\ &= AB + A\,C_{in} + BC_{in}\,(A + \bar{A}) \\ &= ABC_{in} + AB + A\,C_{in} + \bar{A}\,BC_{in} \\ &= AB\,(C_{in} + 1) + A\,C_{in} + \bar{A}\,BC_{in} \quad (\because C_{in} + 1 = 1) \\ &= AB + AC_{in} + \bar{A}\,BC_{in} \\ &= AB + A\,C_{in}\,(B + \bar{B}) + \bar{A}\,B\,C_{in} \\ &= AB\,C_{in} + AB + A\,\bar{B}\,C_{in} + \bar{A}\,BC_{in} \\ &= AB\,(C_{in} + 1) + A\,\bar{B}\,C_{in} + \bar{A}\,BC_{in} \quad (\because C_{in} + 1 = 1) \\ &= AB + A\,\bar{B}\,C_{in} + \bar{A}\,BC_{in} \\ &= AB + C_{in}\,(A\,\bar{B} + \bar{A}\,B)\end{aligned}$$

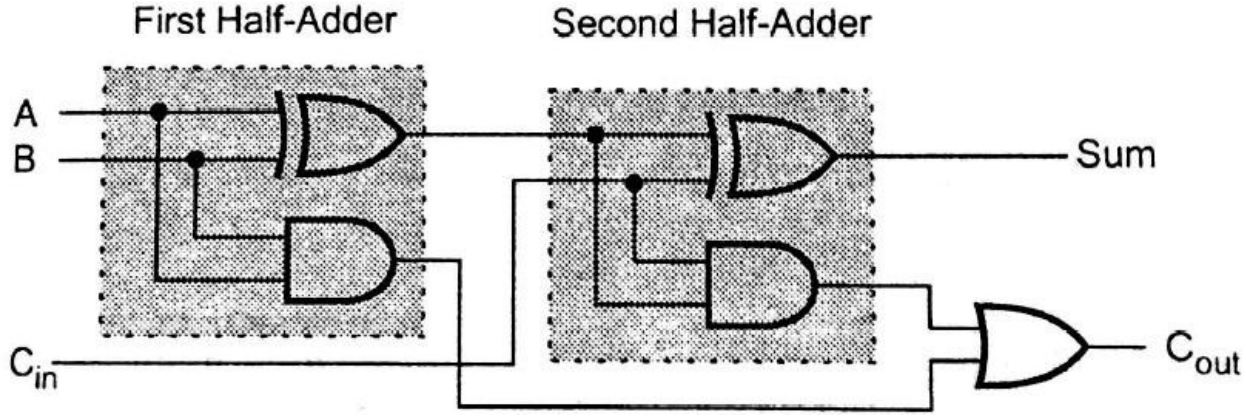


Figure 5.71: Implementation of Full Adder using two Half Adders

(iii) Half Subtractor:

The subtraction consists of four possible elementary operations, namely, $0 - 0 = 0$

$0 - 1 = 1$ with 1 borrow

$1 - 0 = 1$

$1 - 1 = 0$

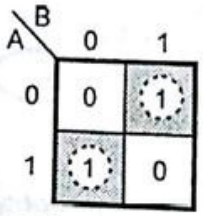
A half-subtractor is a combinational circuit that does the subtraction between two bits and produces their difference. It also has an output to specify if a 1 has been borrowed Let us designate minuend bit as A and the subtrahend bit as B. The Result of opeartion A – B for all possible values of A and B is tabulated

Inputs		Outputs	
A	B	Difference	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Table 5.25: Truth table for half-subtractor

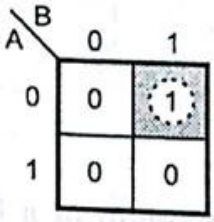
K-map simplification for half-subtractor

For Difference



Difference = $A\overline{B} + \overline{A}B$
= $A \oplus B$

For Borrow



Borrow = $\overline{A}B$

Logic Diagram

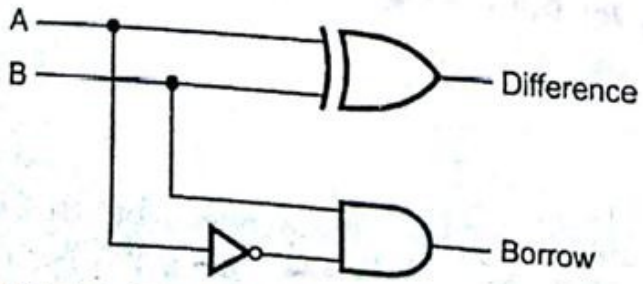


Figure 5.72: Implementation of half-subtractor

(iv) Full Subtractor:

A full subtractor is a combinational circuit that performs a subtraction between two bits taking into account borrow of the lower significant stage. This circuit has three inputs and two outputs. The three inputs are A, B and B_{in}, denote the minuend, subtrahend, and previous borrow respectively. The two outputs, D and B_{out}, represent the difference and output borrow respectively.

Inputs			Outputs	
A	B	B_{in}	D	B_{out}
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

Table 5.26 Truth table for full-subtractor

K-map simplification of D and B_{out}

For D

A \ BB_{in}	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$D = \bar{A}\bar{B}B_{in} + \bar{A}B\bar{B}_{in} + A\bar{B}\bar{B}_{in} + ABB_{in}$

For B_{out}

A \ BB_{in}	00	01	11	10
0	0	1	1	1
1	0	0	1	0

$B_{out} = \bar{A}B_{in} + \bar{A}B + BB_{in}$

Expression D can be simplified as

$$
\begin{aligned}
D &= \bar{A}\,\bar{B}\,B_{in} + \bar{A}\,B\,\bar{B}_{in} + A\,\bar{B}\,\bar{B}_{in} + A\,B\,B_{in} \\
&= B_{in}(\bar{A}\,\bar{B} + AB) + \bar{B}_{in}(\bar{A}\,B + A\,\bar{B}) \\
&= B_{in}(A \odot B) + \bar{B}_{in}(A \oplus B) \\
&= B_{in}\,\overline{(A \oplus B)} + \bar{B}_{in}(A \oplus B) \\
&= B_{in} \oplus (A \oplus B)
\end{aligned}
$$

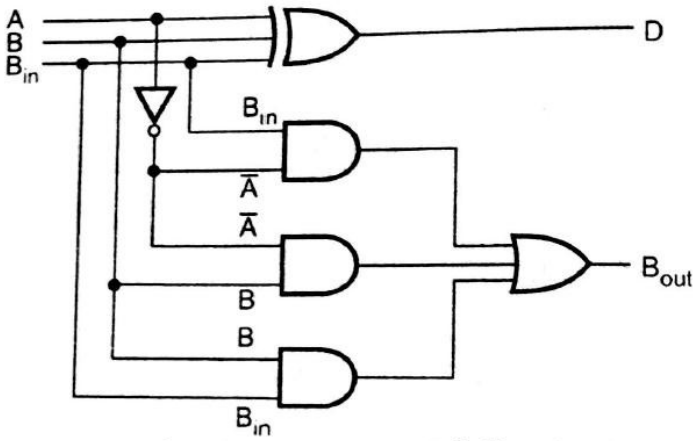


Figure 5.73: Implementation of full-subtractor

A full Subtractor can also be implemented using two half Subtractors

$$
\begin{aligned}
B_{out} &= \bar{A}B + \overline{(A\bar{B} + \bar{A}B)}\,B_{in} \\
&= \bar{A}B + (AB + \bar{A}\bar{B})\,B_{in} \\
&= \bar{A}B + ABB_{in} + \bar{A}\bar{B}B_{in} \\
&= \bar{A}B(1+B_{in}) + ABB_{in} + \bar{A}\bar{B}B_{in} \qquad \because (1+B_{in}) = B_{in} \\
&= \bar{A}B + \bar{A}BB_{in} + ABB_{in} + \bar{A}\bar{B}B_{in} \\
&= \bar{A}B + BB_{in}(\bar{A}+A) + \bar{A}\bar{B}B_{in} \\
&= \bar{A}B + BB_{in} + \bar{A}\bar{B}B_{in} \\
&= \bar{A}B(1+B_{in}) + BB_{in} + \bar{A}\bar{B}B_{in} \qquad \because (1+B_{in}) = B_{in} \\
&= \bar{A}B + \bar{A}BB_{in} + BB_{in} + \bar{A}\bar{B}B_{in} \\
&= \bar{A}B + \bar{A}B_{in}(B+\bar{B}) + BB_{in} \\
&= \bar{A}B + \bar{A}B_{in} + BB_{in}
\end{aligned}
$$

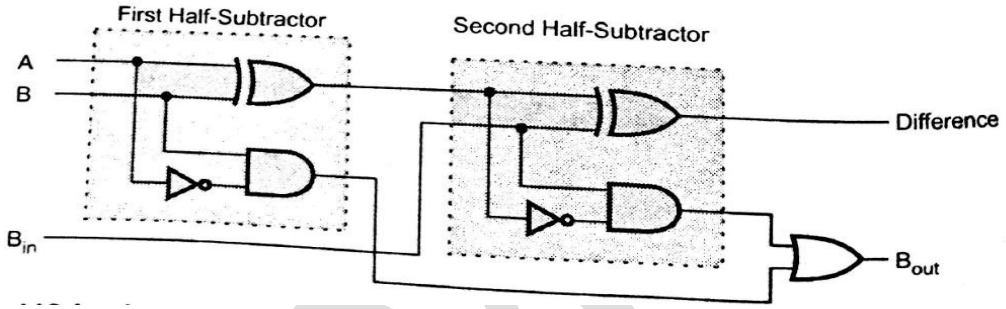


Figure 5.74: Implementation of Full Subtractor using two half subtractors

❖ **Ripple adder (n-bit parallel adder)**

A single full-adder is capable of adding two one-bit numbers and Output carry. In order to add binary numbers with more than one bit, additional full-adder must be employed. A n-bit, parallel adder can be constructed using number of full-adder circuits connected in parallel. The carry output of one full adder is connected to the carry input of the next higher-orde der full-adder

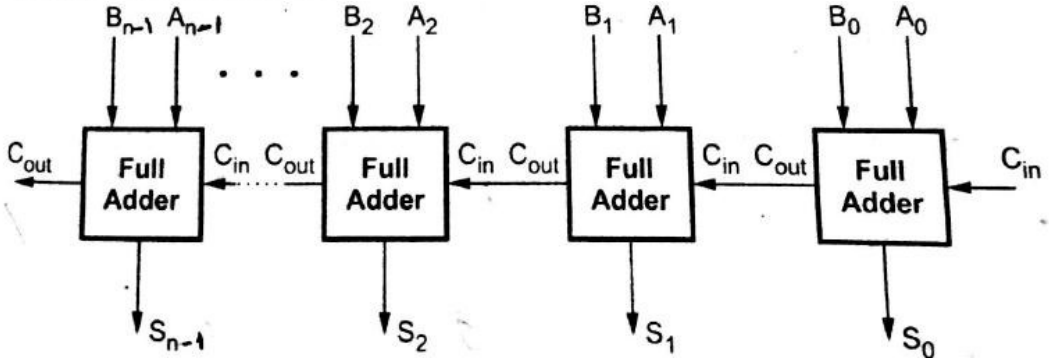


Figure 5.75: Block diagram of n-bit parallel adder

Designing of 4-bit parallel adder using full adders:

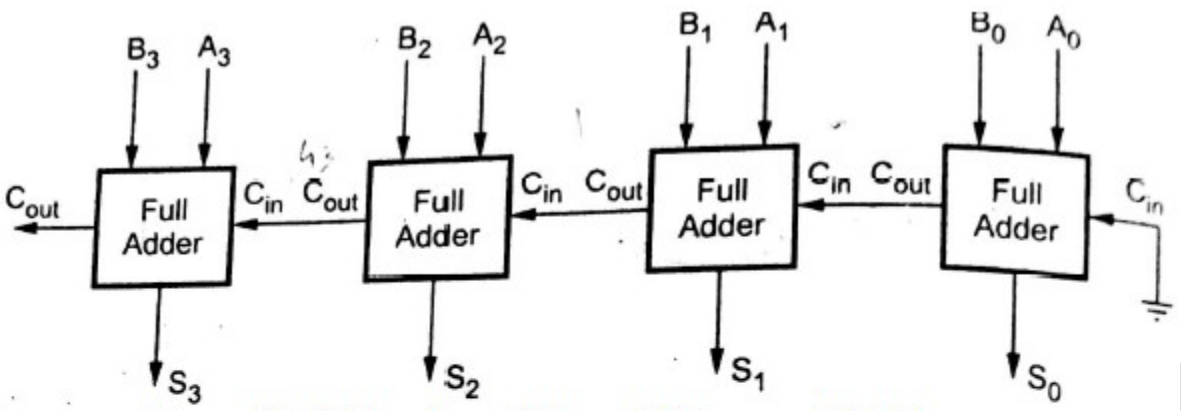


Figure 5.76(a):Implementation of 4-bit parrallel Adder

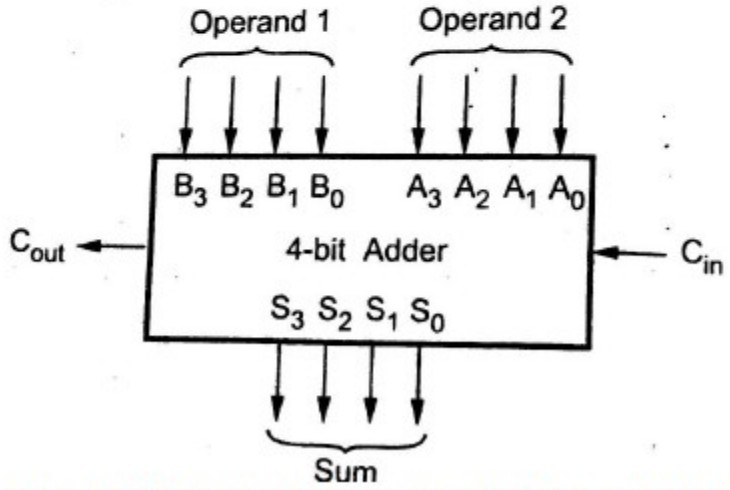


Figure 5.76(b): Block diagram of 4-bit parrallel Adder

❖ **How to eliminate carry propagation delay in adders? Explain the appropriate method? (or) Explain the operation of carry look ahead generator?**

In n-bit parallel adder the carry output of each stage full adder circuit is connected to its next stage full adder circuit. So the sum and carry output of each stage is waiting for its carry input, this leads a timing delay in addition process which is called as carry propagation delay.

Example: Addition of 0101 and 0111

```
  0101
+ 0111
------
  1100
```

In this addition LSB bits of two numbers are added then a carry is generated, the second digit is in waiting position to add until to get a carry. This process is continued till the completion of MSB bit addition. It takes high carry propagation delay to complete the addition process.

One method we have to increase adder speed by eliminating the inter stage carry delay is

called Look ahead-carry addition. It uses two functions carry generate and carry propagate.

carry propagate $P_i = A_i \oplus B_i$ and carry generate $G_i = A_i B_i$

In a full adder circuit the output sum and carry can be represented as $S_i = P_i \oplus C_i$

$C_{i+1} = G_i + P_i C_i$

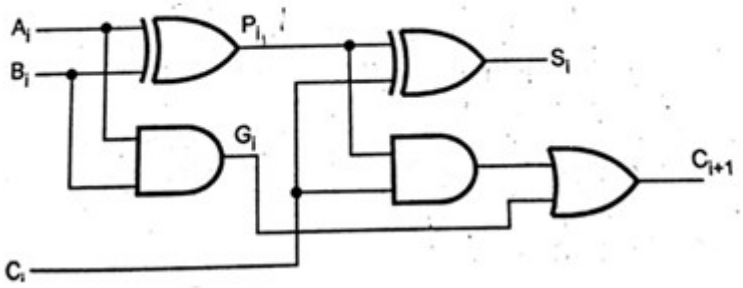


Figure 5.77: Full adder circuit with Carry generate and Carry propagate.

G_i is called carry carry generator, it generates a carry when both input A_i and B_i are logic 1.

P_i is called carry propagate, because it can propagate a carry from C_i to C_{i+1}.

Then the boolean expressions for carry outputs at each stage can be written as follows

$C_1 = G_0 + P_0 C_0$

$C_2 = G_1 + P_1 C_1$

$C_3 = G_2 + P_2 C_2 = G_2 + P_2 (G_1 + P_1 C_1) = G_2 + P_2 G_1 + P_2 P_1 C_1$

$C_4 = G_3 + P_3 C_3 = G_3 + P_3 (G_2 + P_2 G_1 + P_2 P_1 C_1) = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 C_1$

From the above expressions we can conclude that the carry C_1 Depends for C_0 and carry C_2 depends on C_1 and carries C_3 and C_4 also propagating in the same time that the carry C_2 can propagate. IN this way by using carry look-ahead adder we can reduce increased carry propagation dealy.

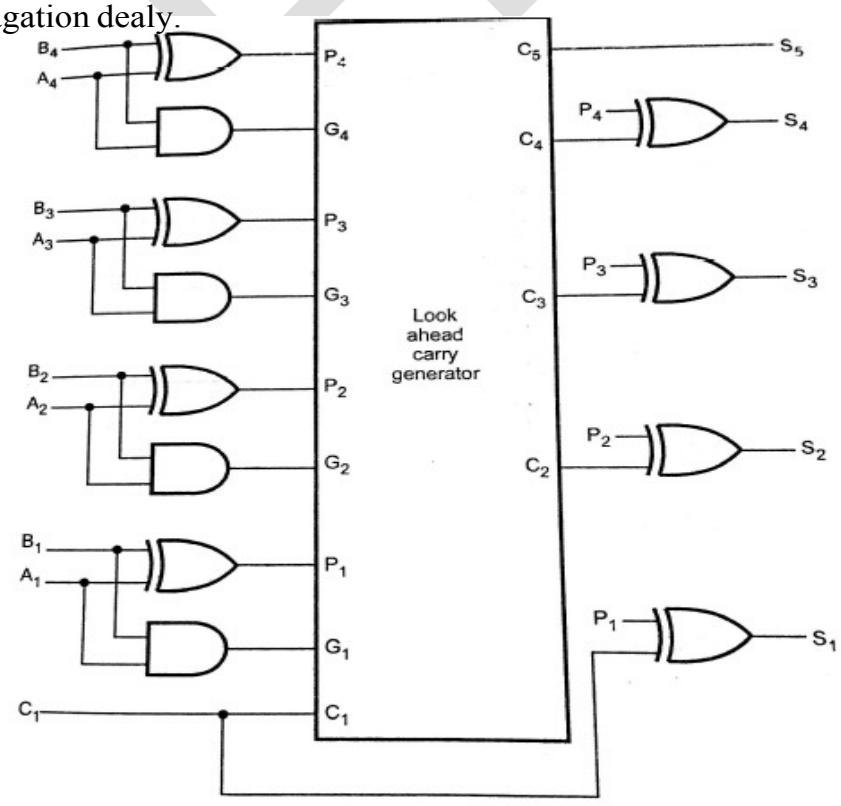


Figure 5.79: 4-bit parallel adder with look ahead carry generator

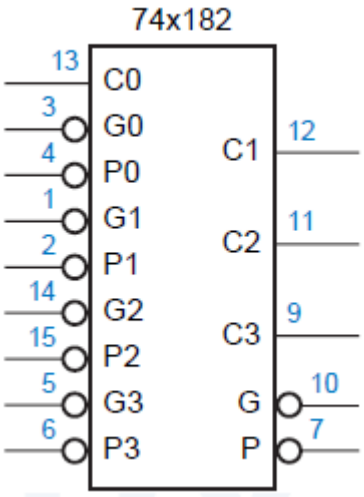


Figure 5.78:Pin diagram of IC74x182 Look-ahead carry circuit

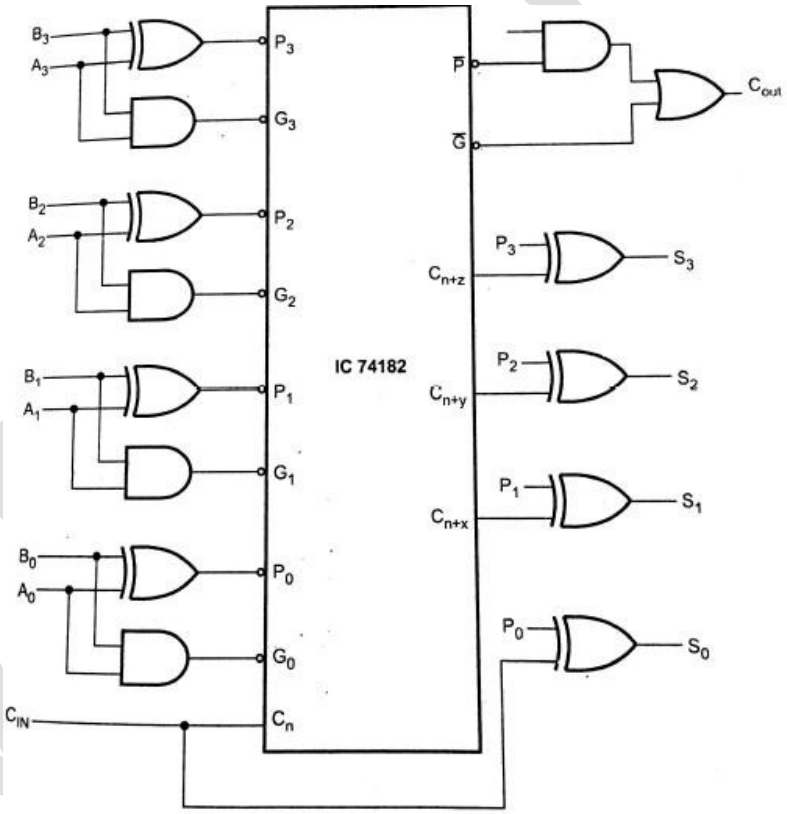


Figure 5.80:4-bit parallel adder using IC 74x182

❖ **Design a 4-bit binary parallel adder using IC 74LS283?**

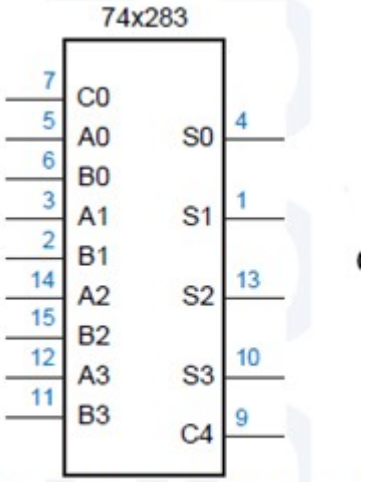


Figure 5.81(a): Pin diagram of IC74x283

The *74x283* is a 4-bit binary adder that forms its sum and carry outputs with just a few levels of logic, using the carry lookahead technique. Figure 5-81 is a logic symbol for the 74x283. The older *74x83* is identical except for its pinout, which has nonstandard locations for power and ground.

The logic diagram for the IC 74x283 has just a few differences from the general carry-lookahead design,it produces active-low versions of the carry-generate (g_i') and carry-propagate (p_i') signals, since inverting gates are generally faster than noninverting ones. it takes advantage of the fact that we can algebraically manipulate the half-sum equation

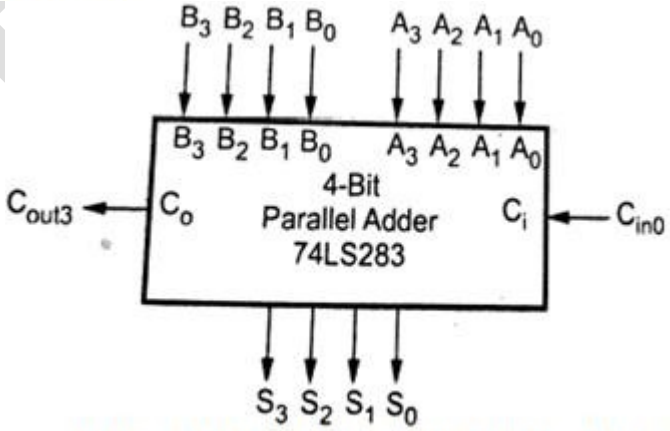


Figure 5.81(b): Logic Symbol for IC74x283

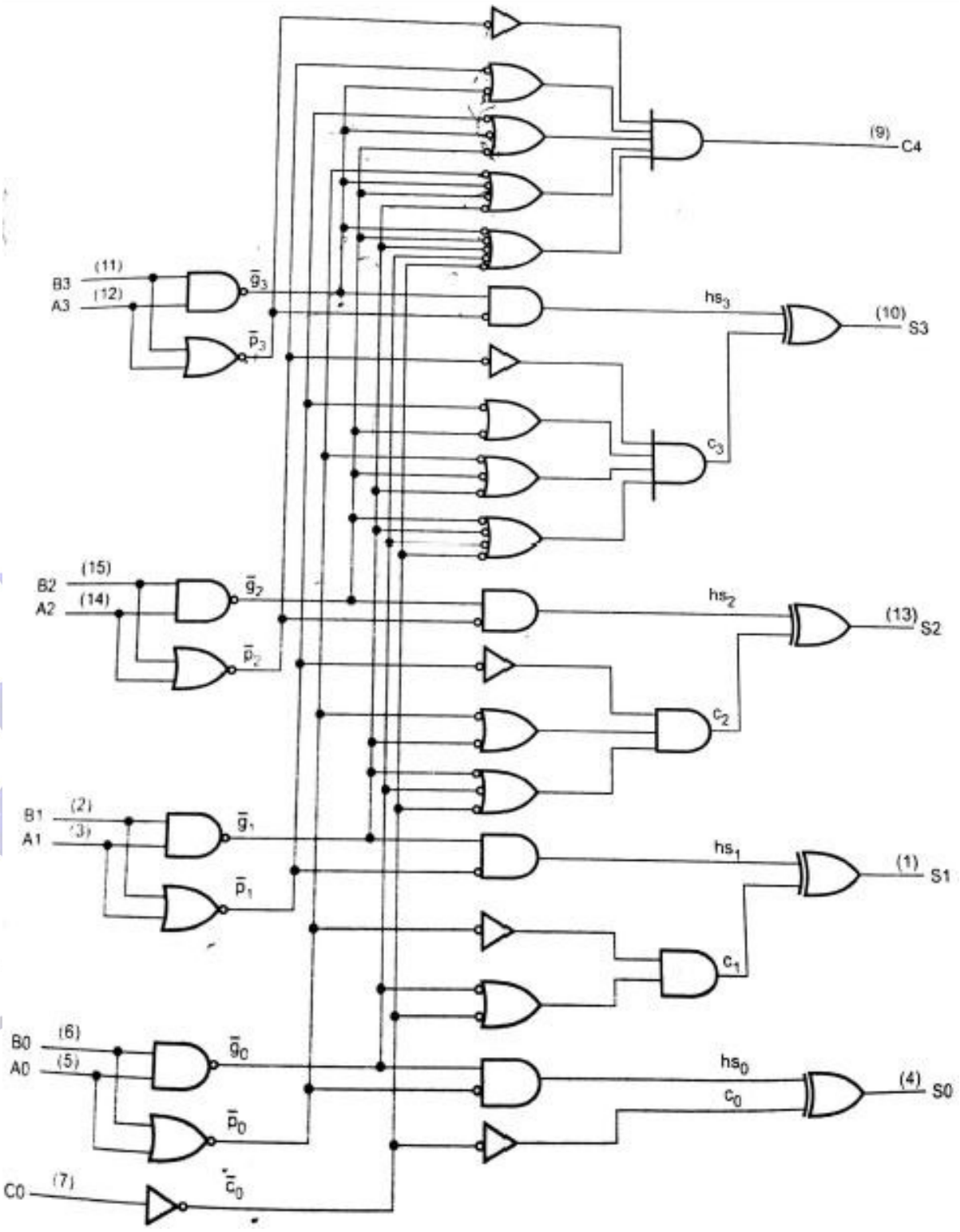


Figure 5.82:Logic diagram for IC74x283(4-bit binary parallel adder)

❖ **Write a short note on n-bit parallel Subtractor?**

n-Bit Parallel Subtractor

The subtraction of binary numbers can be done most conveniently by means of 2's complements. Remember that the subtraction A-B can be done by taking the 2's complement of B and adding it to A. The 2's complement can be obtained by taking the 1's complement and adding one to the least significant pair of bits ,the1's complement can be implemented with inverters and a one can be added to the sum through the input carry,

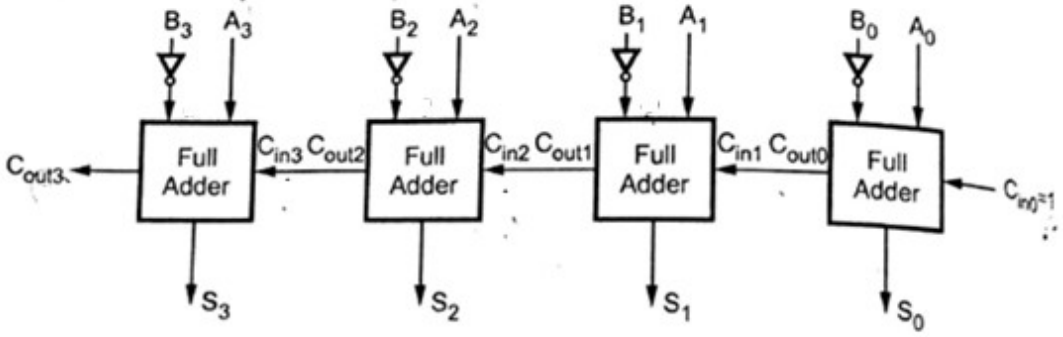


Figure 5.83: 4-bit parallel subtractor

❖ **Design a 4-bit Adder-Subtractor circuit and explain its operation?**

Binary Adder-Subtractor

The addition and subtraction operations can be combined into one circuit with one common binary adder. This is done by including an exclusive OR gate with each full adder, as shown in Fig. 4.126. The mode input M controls the operation of the circuit When M = 0, the circuit is an adder, and when M = 1, the circuit becomes a subtractor Each exclusive OR gate receives input M and one of the inputs of B. When M=0 we have $B \oplus 0 = B$. The full-adders receive the value of B, the input carry is 0, and the circuit performs A plus B. When M = 1, we have $B \oplus 1 = \overline{B}$ and $C_{in\,0} = 1$. The B inputs are all complemented and a 1 is added through the input carry. The circuit performs the operation A plus the 2's complement of B, i.e. A – B.

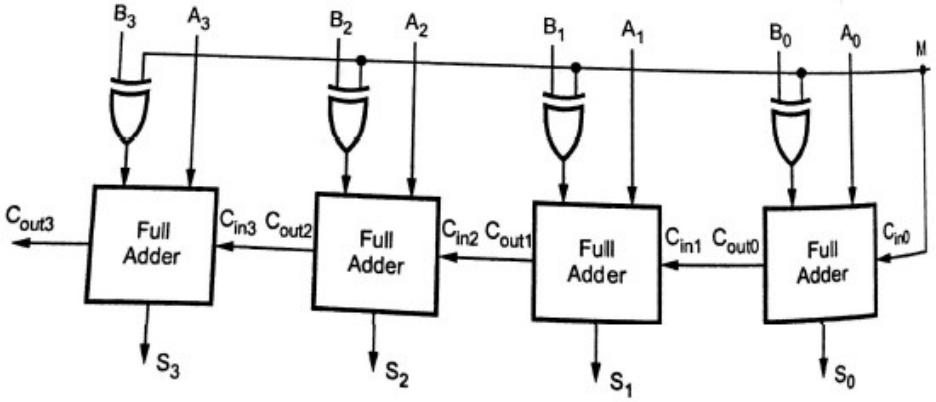


Figure 5.84: 4-bit adder-subtractor

❖ **Design and explain the operation of ALU?**

Arithmetic and Logic Unit(ALU) performs necessary arithmetic and logical operations. It is basically a multi functional combinational logic circuit. It provides select input to select perticular output.

IC 74LS181 is a 4-bit ALU. Its features are given by

- Provides 16 arithmetic operations : add, subtract, compare, double, plus twelve other arithmetic operations.
- Provides all 16 logic operations of two variables : exclusive-OR, compare, AND, NAND, OR, NOR, plus ten other logic operations.
- Full look ahead for high speed arithmetic operation on long words.

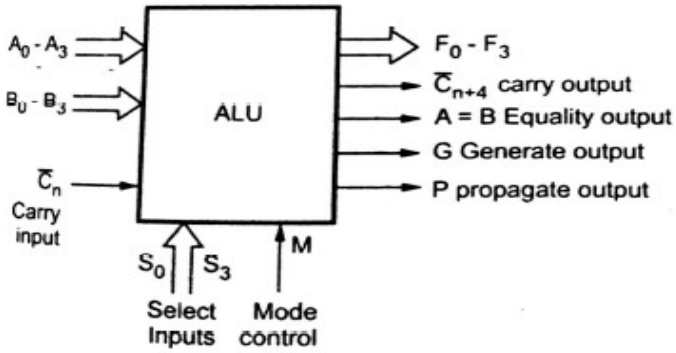


Figure 5.85(a) Block Diagram

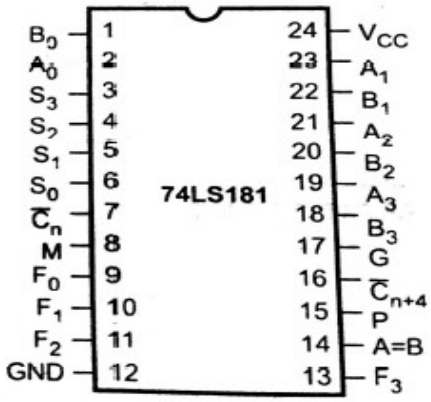


Figure 5.85(b) Pin Diagram

ALU has 4-bit operands $(A_0 - A_3$ and $B_0 - B_3)$. and a mode control input 'M'. It selects one of two modes of Arithmetic or Logical and it has four functional select inputs to select a particular function from the selected mode.

Pin names	Description
$A_0 - A_3$	Operand Inputs
$B_0 - B_3$	Operand Inputs
$S_0 - S_3$	Function Select Inputs
M	Mode Control Input
$\overline{C}_n$	Carry Input (Active LOW)
$F_0 - F_3$	Function Outputs
A = B	Comparator Output
G	Carry Generate Output
P	Carry Propagate Output
$\overline{C}_{n+4}$	Carry Output (Active LOW)

Table 5.27: Pin description of IC74LS181

Mode Select Inputs				Active HIGH Operands and F_n Outputs	
				Logic	Arithmetic (Note 2)
S_3	S_2	S_1	S_0	(M = 1)	(M = 0) ($\overline{C}_n$ = 1)
0	0	0	0	$F = \overline{A}$	F = A
0	0	0	1	$F = \overline{A} + \overline{B}$	F = A + B
0	0	1	0	$F = \overline{A}\,B$	$F = A + \overline{B}$
0	0	1	1	F = Logic 0	F = minus 1
0	1	0	0	$F = \overline{AB}$	F = A plus $A\overline{B}$
0	1	0	1	$F = \overline{B}$	F = (A + B) plus $A\overline{B}$
0	1	1	0	$F = A \oplus B$	F = A minus B minus 1
0	1	1	1	$F = A\overline{B}$	F = AB minus 1
1	0	0	0	$F = \overline{A} + B$	F = A plus AB
1	0	0	1	$F = \overline{A \oplus B}$	F = A plus B
1	0	1	0	F = B	$F = (A + \overline{B})$ plus AB
1	0	1	1	F = AB	F = AB minus 1
1	1	0	0	F = Logic 1	F = A plus A (Note 1)
1	1	0	1	$F = A + \overline{B}$	F = (A + B) plus A
1	1	1	0	F = A + B	$F = (A + \overline{B})$ plus A
1	1	1	1	F = A	F = A minus 1

Mode Select Inputs				Active LOW Operands and F_n Outputs	
				Logic	Arithmetic (Note 2)
S_3	S_2	S_1	S_0	(M = 1)	(M = 0) ($\overline{C}_n$ = 0)
0	0	0	0	$F = \overline{A}$	F = A minus 1
0	0	0	1	$F = \overline{AB}$	F = AB minus 1
0	0	1	0	$F = \overline{A} + \overline{B}$	$F = A\overline{B}$ minus 1
0	0	1	1	F = Logic 1	F = minus 1
0	1	0	0	$F = \overline{A} + \overline{B}$	F = A plus $(A + \overline{B})$
0	1	0	1	$F = \overline{B}$	$F = A\overline{B}$ plus $(A + \overline{B})$
0	1	1	0	$F = \overline{A \oplus B}$	F = A minus B minus 1
0	1	1	1	$F = A + \overline{B}$	$F = A + \overline{B}$
1	0	0	0	$F = \overline{A}\,B$	F = A plus (A + B)
1	0	0	1	$F = A \oplus B$	F = A plus B
1	0	1	0	F = B	$F = A\overline{B}$ plus (A + B)
1	0	1	1	F = A + B	F = A + B
1	1	0	0	F = Logic 0	F = A plus A (Note 1)
1	1	0	1	$F = A\overline{B}$	F = AB plus A
1	1	1	0	F = AB	$F = A\overline{B}$ minus A
1	1	1	1	F = A	F = A

Table 5.28: Functional table of IC74LS181

Functional Description

The 74LS181 is a 4-bit high speed parallel Arithmetic Logic Unit (ALU), controlled by the four Function Select inputs (S0-S3) and the Mode Control input (M), it can perform all the 16 possible logic operations or 16 different arithmetic operations on active HIGH or active LOW operands. The Function Table lists these operations.

When the Mode Control input (M) is HIGH, all internal carries are inhibited and the device performs logic operations on the individual bits as listed. When the Mode Control input is LOW, the carries are enabled and the device performs arithmetic operations on the two 4-bit words. The device incorporates full internal carry look ahead and provides for either ripple carry between devices using the $\overline{C}_{n+4}$ output, or for carry look ahead between packages using the signals P (Carry Propagate) and G (Carry Generate). In the ADD mode, P indicates that F is 15 or more, while G indicates that F is 16 or more. In the SUBTRACT mode, P indicates that F is zero or less, while G indicates that F is less than zero. P and G are not affected by carry in. When speed requirements are not stringent, it can be used in a simple ripple carry mode by connecting the Carry output ($\overline{C}_{n+4}$) signal to the Carry input (C_n) of the next unit.

The A = B output from the device goes HIGH when all four F outputs are HIGH and can be used to indicate logic equivalence over four bits when the unit is in the subtract mode. The A = B output is open-collector and can be wired - AND with other A = B out puts to give a comparison for more than four bits. The A = B signal can also be used with the $\overline{C}_{n+4}$ signal to indicate A > B and A < B.

The Function Table lists the arithmetic operations that are performed without a carry in. An incoming carry adds an one to each operation. Thus, select code 0110 generates A minus B minus 1 (2's complement notation) without a carry in and generates A minus B when a carry is applied. Because subtraction is actually performed by complementary addition (1's complement), a carry out means borrow; thus a carry is generated when there is no underflow and no carry is generated when there is underflow. As indicated, this device can be used with either active LOW inputs producing active LOW outputs or with active HIGH inputs producing active HIGH outputs. For either case the table lists the operations that are performed to the operands labeled inside the logic symbol.

❖ **Design 8-bit ALU circuit using IC74LS181?**

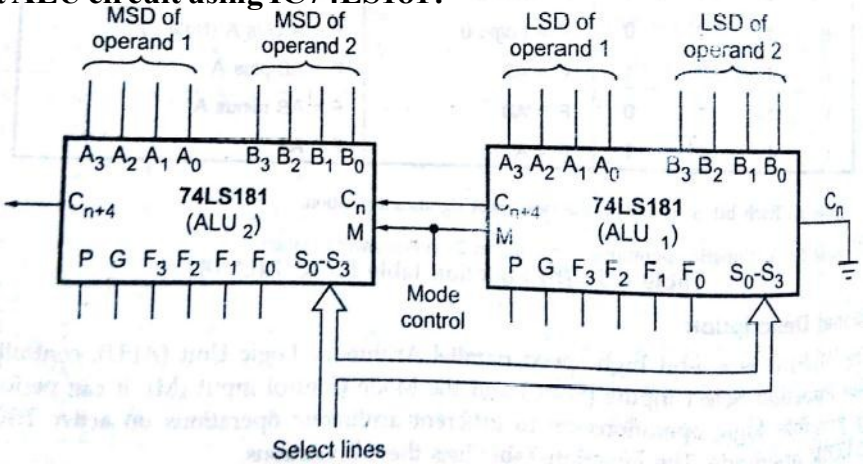


Figure 5.86: Design of 8-bit ALU circuit using IC74LS181

❖ **Design and explain the operation of ALU using IC74x381 and IC74x382?**

IC74x381 and IC74X382 are two other MSI ICs for ALU. The only difference between IC74x381 and IC74X382 is that IC74x381 provides group carry look-ahead outputs, while IC74X382 provides ripple carry and overflow outputs.

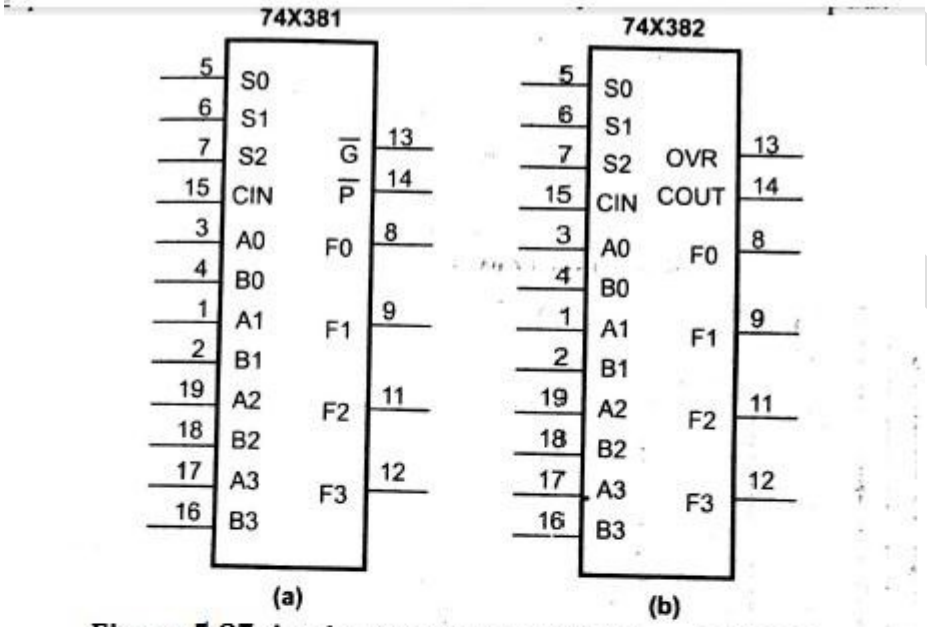


Figure 5.87: Logic symbols for 74X381 and 74X382

Inputs			Function
S2	S1	S0	
0	0	0	F = 0000
0	0	1	F = B minus A minus 1 plus CIN
0	1	0	F = A minus B minus 1 plus CIN
0	1	1	F = A plus B plus CIN
1	0	0	F = A ⊕ B
1	0	1	F = A + B
1	1	0	F = A · B
1	1	1	F = 1111

Table 5.29: Function table for IC 74X381 and IC 74X382

❖ **Explain the operation of combinational multiplier with an example?**

The cobinational multiplier is a circuit that uses n shifts and adds to multiply n-bit binary numbers. Although the shift-and-add algorithm emulates the way that we do paper-and-pencil multiplication of decimal numbers, there is nothing inherently "sequential" or "time dependent" about multiplication. That is, given two n-bit input words A and B it is possible to write a truth table that expresses the $2n$-bit product $P = A.B$ as a *combinational* function of A and B.A *combinational multiplier* is a logic circuit with such a truth table.

Example: Multiplication of 011_2 *by* 110_2

```
    0 1 1
  × 1 1 0
  -------
    0 0 0
+   0 1 1 0   ← shift left
+ 0 1 1 0 0   ← shift left
  -----------
  1 0 0 1 0
```

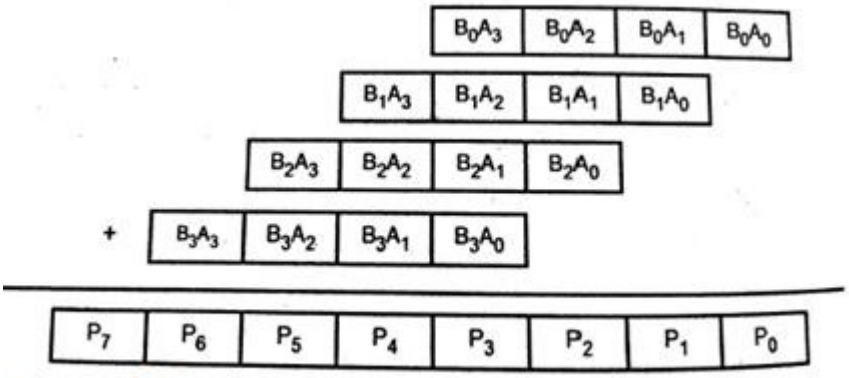


Figure 5.88: Multiplication process of 4 × 4 multiplier

Combinationa multiplier involves full adder circuits to add each product term obtained.

In each partial product row of full adders are connected to make an 4-bit ripple adder. Thus the first 4-bit ripple adder adds the first two rows of product components to produce the first partial product. The carry output generated is propagated to the most significant product component used to produce the next partial product. The subsequent adders adds each partial product with the next product component.

In the above multiplier, the least significant adder is full-adder 1 and the adder which gives the MSB of the product (P_7) is full-adder 12. If we assume for simplicity that the delays from any input to any output of a full adder are equal, say t_{pd}, then the worst-case path goes through 8 full-adders (1, 2, 3, 4, 7, 8, 11 and 12) and the its delay is 8 t_{pd}.

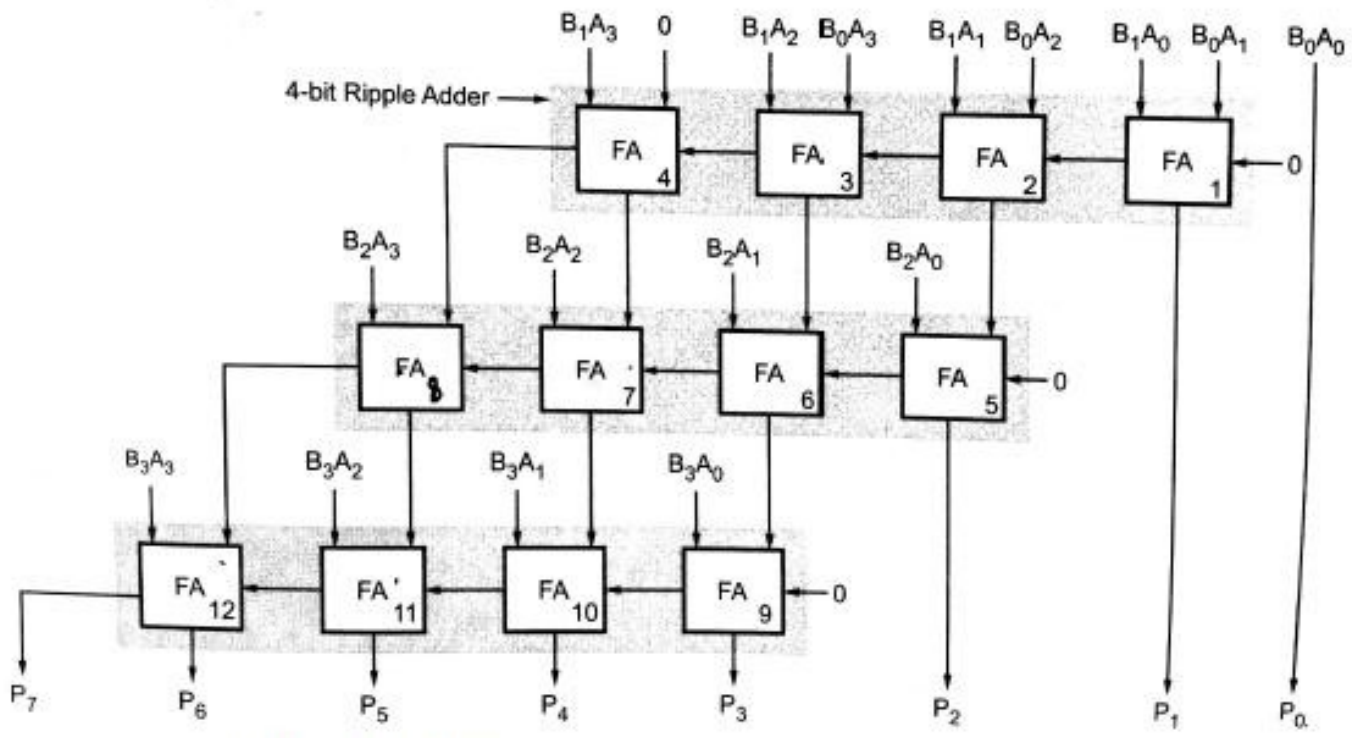


Figure 5.88(b): 4 × 4 combinational multiplier

❖ **Explain the operation of combinational multiplier using carry save method?**

In the above multiplier, the least significant adder is full-adder 1 and the adder which gives the MSB of the product (P_7) is full-adder 12. If we assume for simplicity that the delays from any input to any output of a full adder are equal, say t_{pd}, then the worst-case path goes through 8 full-adders (1, 2, 3, 4, 7, 8, 11 and 12) and the its delay is 8 t_{pd}.

To increase the speed of the addition process many times technique called **carry-save addition** is used. In this technique, the carry output from bit i during step j is applied to carry input for bit i + 1 during the next step, j + 1. After addition of product components in the last row, one more step is required in which the carries are allowed to ripple from the least to the most significant bit. The Fig. 5.89 shows 4 × 4 combinational multiplier using carry-save addition technique. In this the carry out of each full adder in the first seven row of full adders are connected to an input of an adder below it. Carries in the fourth row of full adders are connected to create a conventional ripple adder. This technique does not save any hardware but it reduces the propagation delay

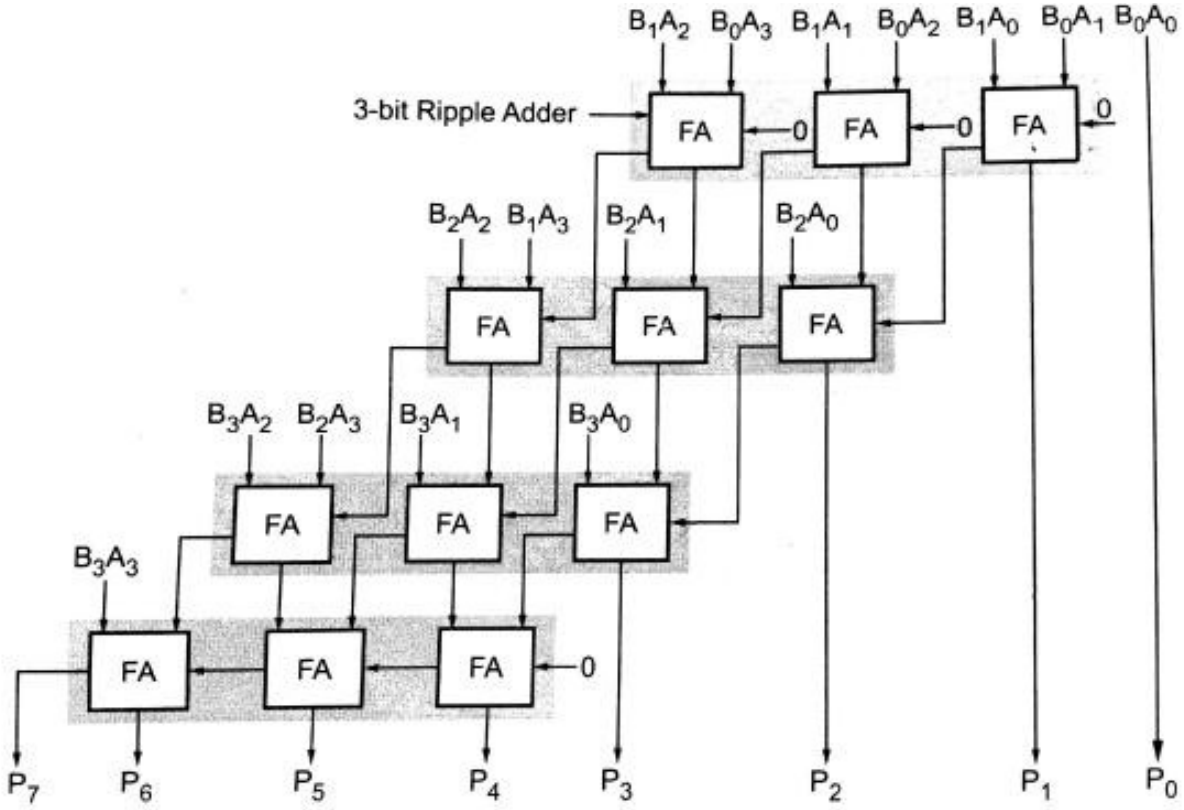


Figure 5.89: 4 × 4 combinational multiplier with carry save addition

❖ **Design a 16-bit ALU using look-ahead group carry circuit?**

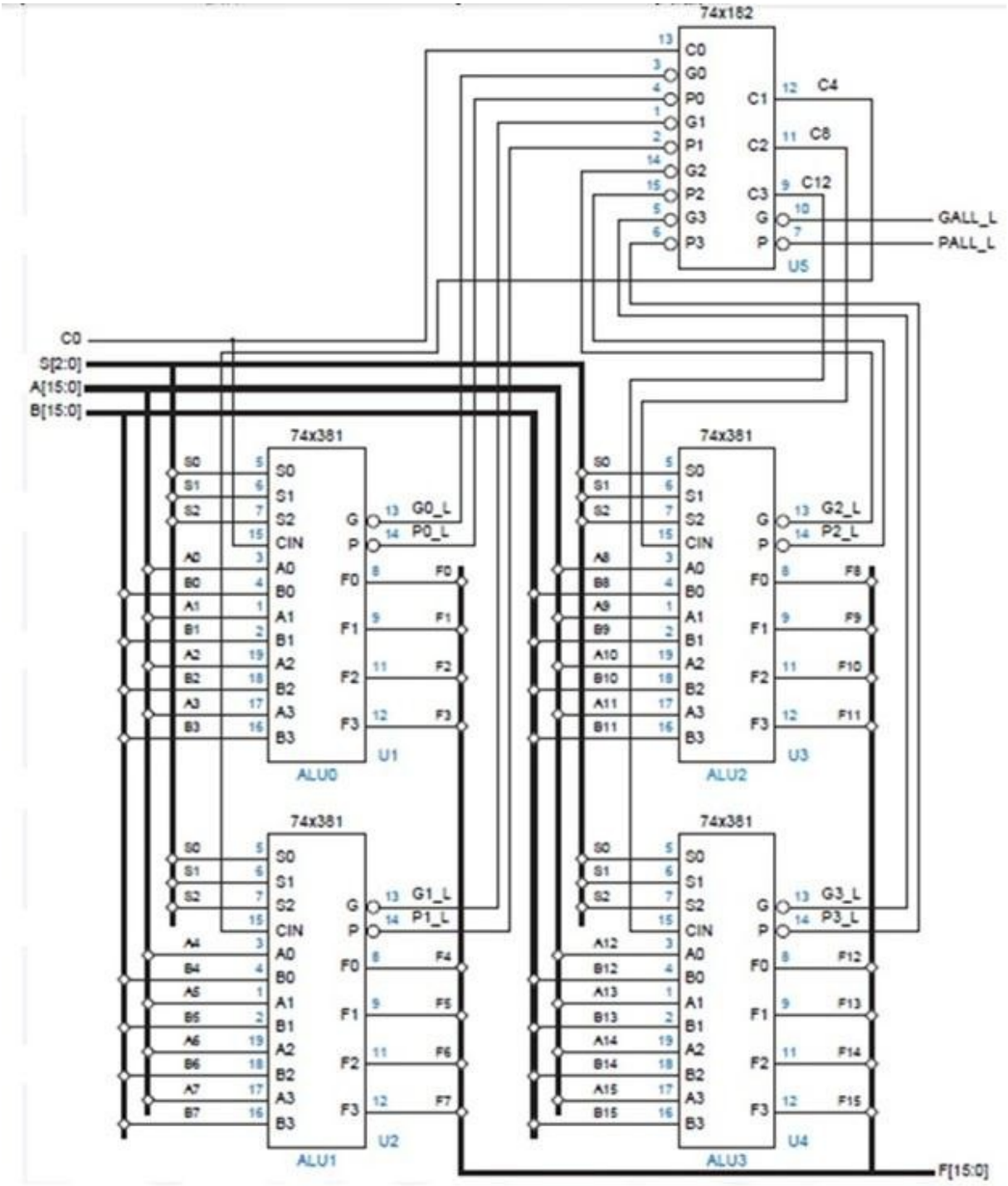


Figure 5.90; A 16-bit ALU using Group carry look-ahead circuit

❖ **Design a priority encoder for 16 inputs using two IC74X148s?**

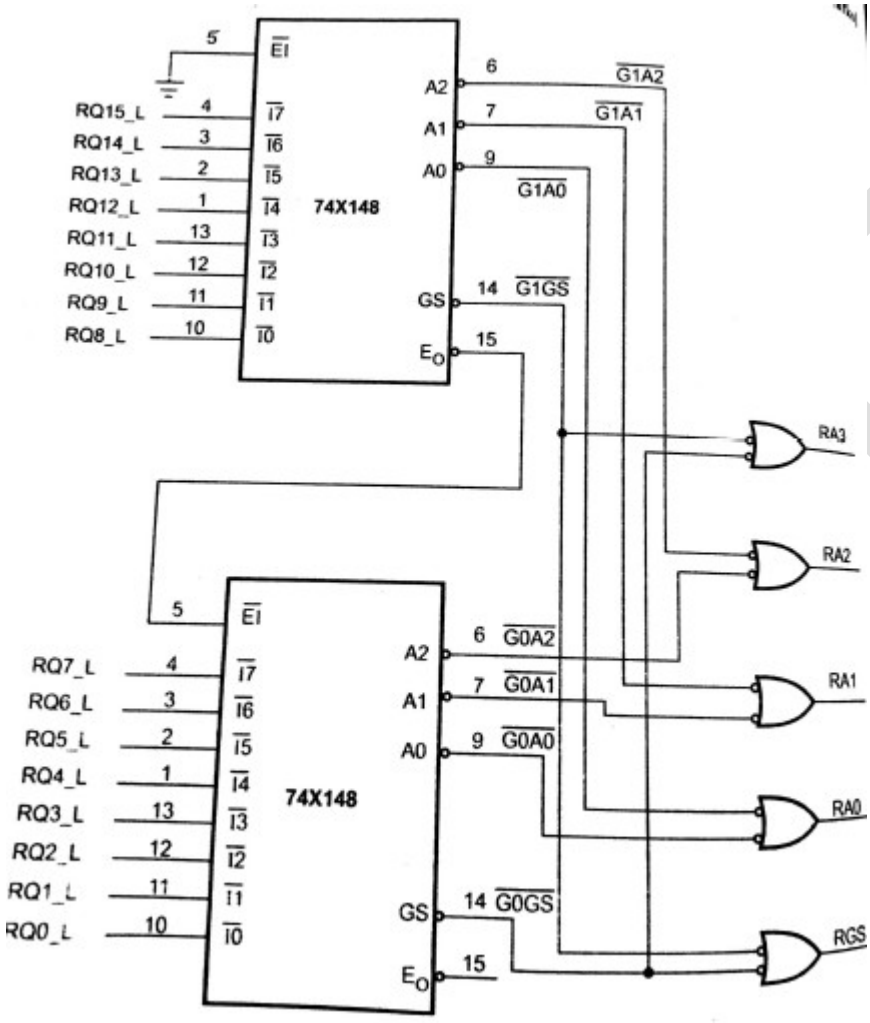
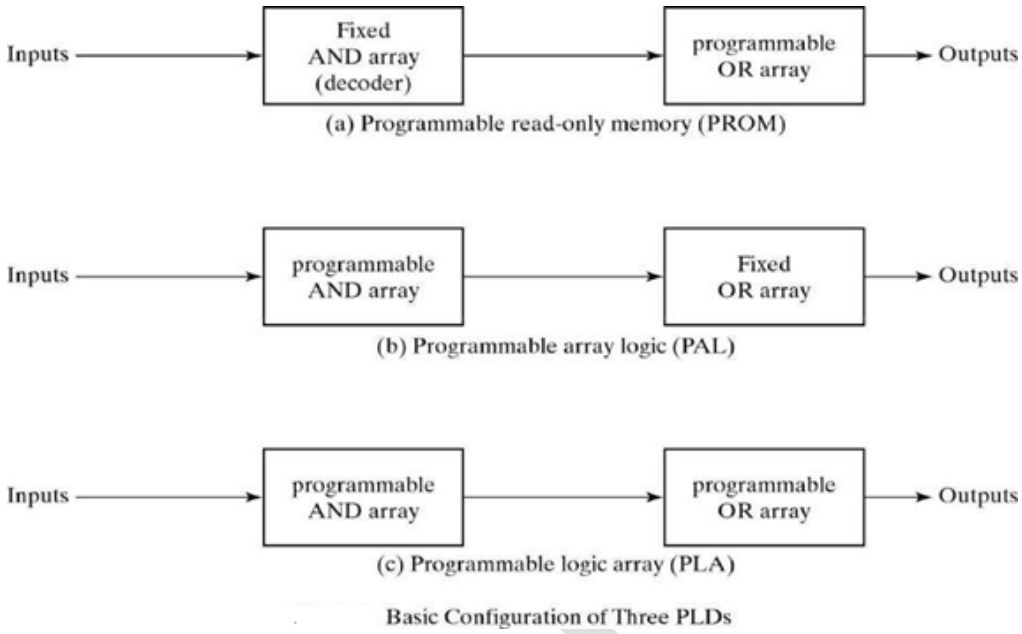


Figure 5.95: Priority Encoder for 16 inputs using two IC74X148s.

Combinational PLDs

- A combinational PLD is an integrated circuit with programmable gates divided into an AND array and an OR array to provide an AND-OR sum of product implementation.
- PROM: fixed AND array constructed as a decoder and programmable OR array.
- PAL: programmable AND array and fixed OR array.

PLA: both the AND and OR arrays can be programmed.



Basic Configuration of Three PLDs

Programmable Logic Array

In the above Fig. the decoder in PROM is replaced by an array of AND gates that can be programmed to generate any product term of the input variables. The product terms are then connected to OR gates to provide the sum of products for the required Boolean functions. The output is inverted when the XOR input is connected to 1 (since $x \oplus 1 = x'$). The output doesn't change and connect to 0 (since $x \oplus 0 = x$).

F1 = AB'+AC+A'BC'

F2 = (AC+BC)'

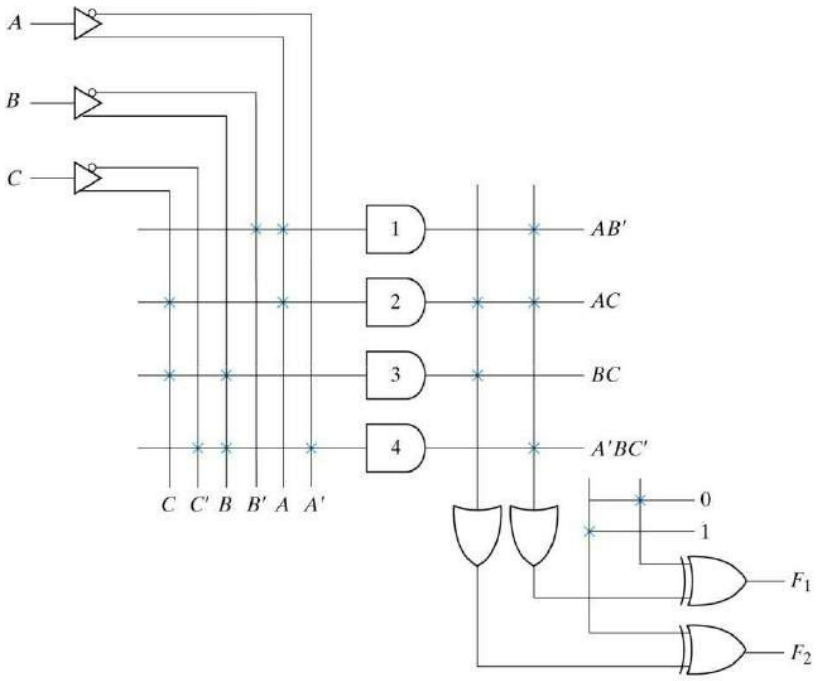


Fig. 7-14 PLA with 3 Inputs, 4 Product Terms, and 2 Outputs

Table 7-5
PLA Programming Table

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

Programming Table

1. First: lists the product terms numerically
2. Second: specifies the required paths between inputs and AND gates
3. Third: specifies the paths between the AND and OR gates
4. For each output variable, we may have a T(ture) or C(complement) for programming the XOR gate

Simplification of PLA

Careful investigation must be undertaken in order to reduce the number of distinct product terms, PLA has a finite number of AND gates. Both the true and complement of each function should be simplified to see which one can be expressed with fewer product terms and which one provides product terms that are common to other functions.

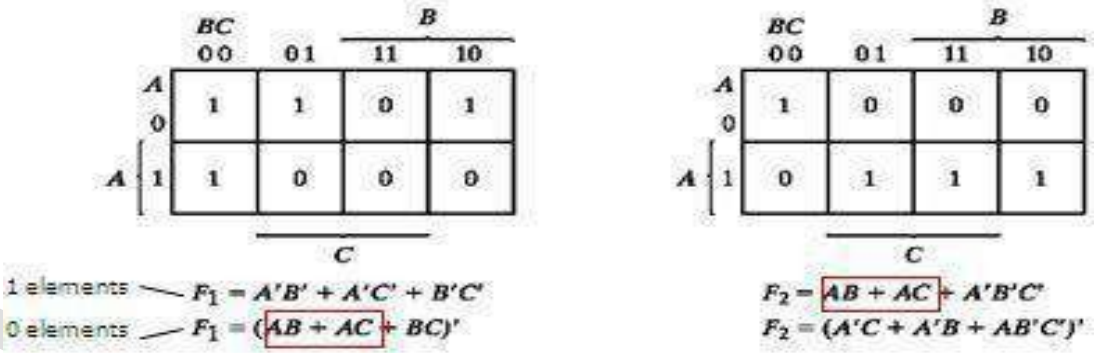
Example

Implement the following two Boolean functions with a PLA:

$F_1(A, B, C) = \sum(0, 1, 2, 4)$

$F_2(A, B, C) = \sum(0, 5, 6, 7)$

The two functions are simplified in the maps of Fig.7-15



PLA table by simplifying the function

Both the true and complement of the functions are simplified in sum of products. We can find the same terms from the group terms of the functions of F_1, F_1',F_2 and F_2' which will make the minimum terms.

F1 = (AB + AC + BC)'

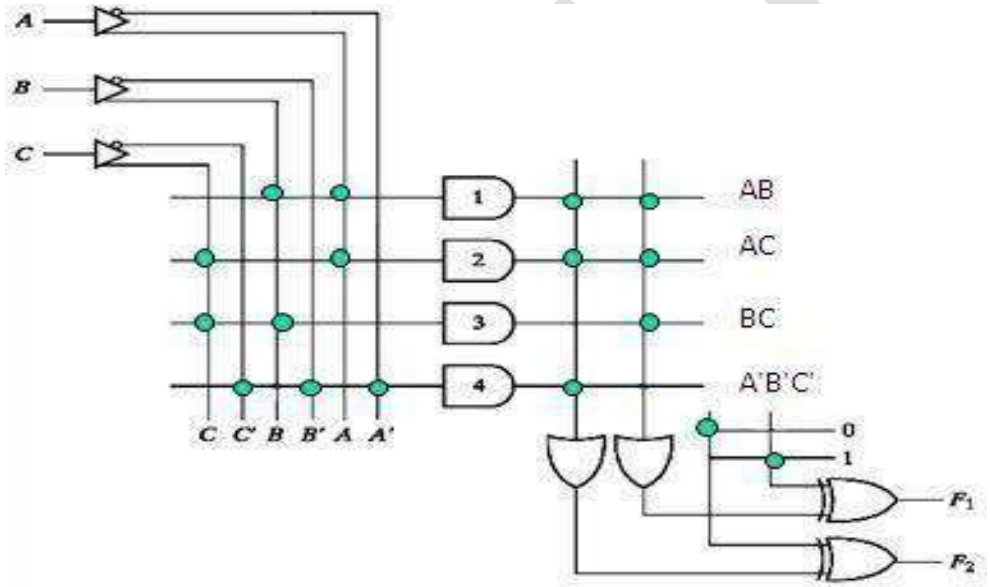
F2 = AB + AC + A'B'C'

PLA programming table

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

Fig. 7-15 Solution to Example 7-2

PLA implementation



Programmable Array Logic

The PAL is a programmable logic device with a fixed OR array and a programmable AND array.

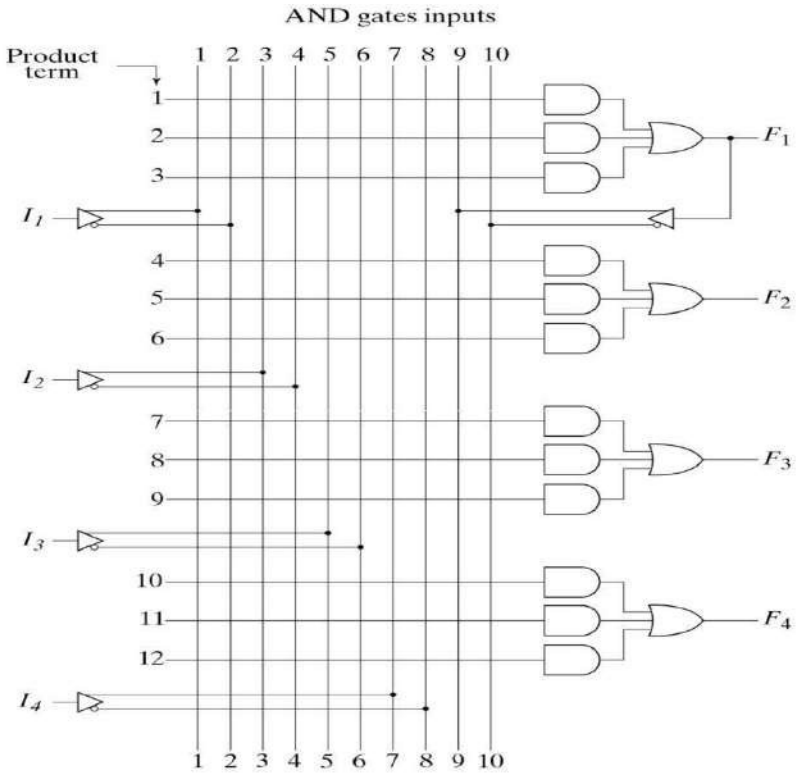


Fig. 7-16 PAL with Four Inputs, Four Outputs, and Three-Wide AND-OR Structure

When designing with a PAL, the Boolean functions must be simplified to fit into each section. Unlike the PLA, a product term cannot be shared among two or more OR gates. Therefore, each function can be simplified by itself without regard to common product terms. The output terminals are sometimes driven by three-state buffers or inverters.

Example

$w(A, B, C, D) = \sum(2, 12, 13)$
$x(A, B, C, D) = \sum(7, 8, 9, 10, 11, 12, 13, 14, 15)$
$y(A, B, C, D) = \sum(0, 2, 3, 4, 5, 6, 7, 8, 10, 11, 15)$
$z(A, B, C, D) = \sum(1, 2, 8, 12, 13)$

Simplifying the four functions as following Boolean functions:

$w = ABC' + A'B'CD'$
$x = A + BCD$
$w = A'B + CD + B'D'$
$w = ABC' + A'B'CD' + AC'D' + A'B'C'D = w + AC'D' + A'B'C'D$

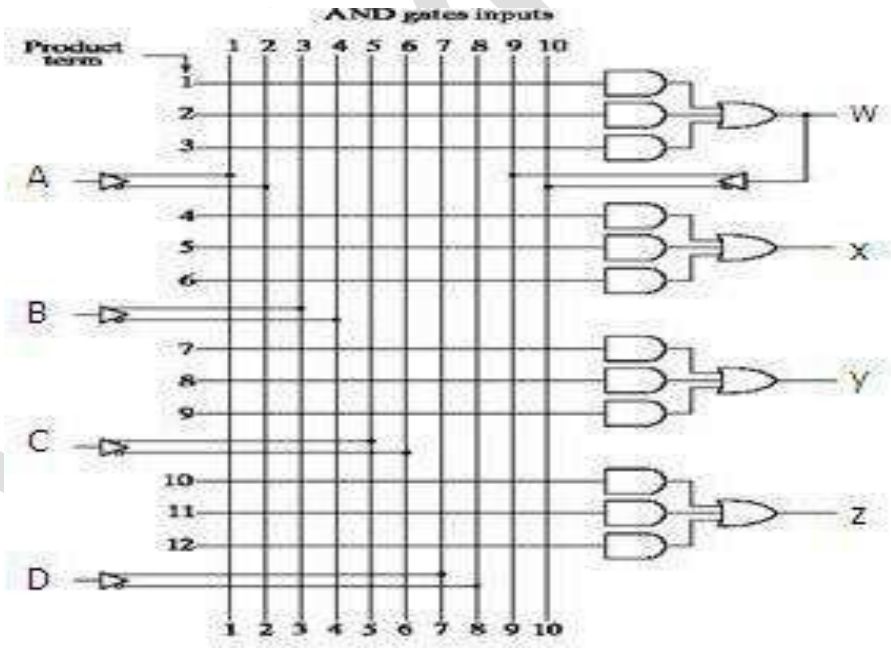
PAL Table

z has four product terms, and we can replace by w with two product terms, this will reduce the number of terms for z from four to three.

Table 7-6
PAL Programming Table

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

PAL implementation



Fuse map for example

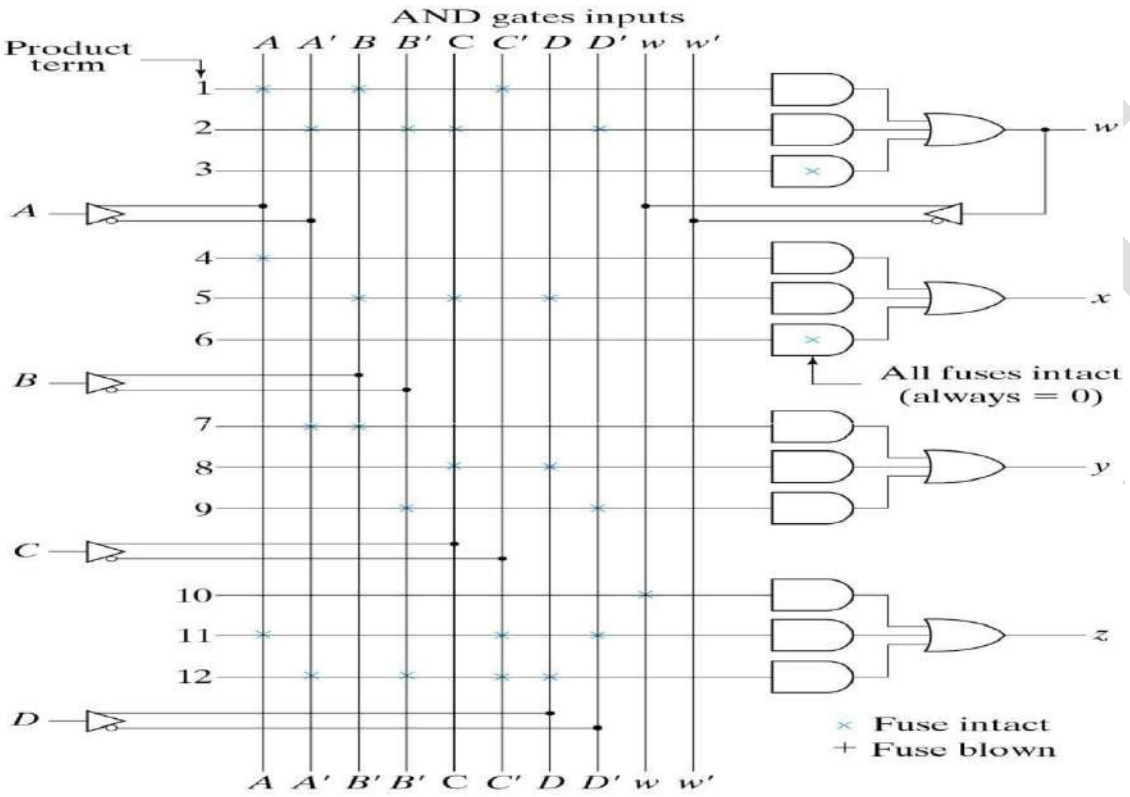


Fig. 7-17 Fuse Map for PAL as Specified in Table 7-6

❖ **Write VHDL code for ALU with IC74x381?**

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all;
ENTITY ALU IS
     PORT ( S : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
       A,B    : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
       F      : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
       (IN    : OUT STD_LOGIC);
END ALU;
ARCHITECTURE Behavior OF ALU IS
BEGIN
  PROCESS (S, A, B)
    BEGIN
        CASE S IS
          WHEN "000"=> F<="0000";
          WHEN "001"=> F<=B-A-1+CIN;
          WHEN "010"=> F<=A-B-1+CIN;
          WHEN "011"=> F<=A+B+CIN;
          WHEN "100"=> F<=A XOR B;
         WHEN "101"=> F<=A OR B;
         WHEN "110"=> F<=A AND B;
         WHEN OTHERS => F<=1111;
                END CASE;
              END PROCESS;
            END Behavior;
```

❖ **Write VHDL codes for carry look-ahead adder circuit?**

```
LIBRARY ieee;
 USE ieee.std_logic_1164.ALL;
 ENTITY c_l_addr IS
   PORT
     ( x_in     : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
       y_in     : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
       carry_in : IN STD_LOGIC;
       sum      : OUT STD_LOGIC_VECTOR(7 DOWNTO  0);
       carry_out : OUT STD_LOGIC      );
 END c_l_addr;
 ARCHITECTURE behavioral OF c_l_addr IS
 SIGNAL   h_sum           :   STD_LOGIC_VECTOR(7 DOWNTO 0);
 SIGNAL   carry_generate     :   STD_LOGIC_VECTOR(7 DOWNTO 0);
 SIGNAL   carry_propagate     :   STD_LOGIC_VECTOR(7 DOWNTO 0);
 SIGNAL   carry_in_internal   :   STD_LOGIC_VECTOR(7 DOWNTO 1);
 BEGIN
   h_sum <= x_in XOR y_in;
   carry_generate <= x_in AND y_in;
   carry_propagate <= x_in OR y_in;
   PROCESS (carry_generate,carry_propagate,carry_in_internal)
   BEGIN
   carry_in_internal(1) <= carry_generate(0) OR (carry_propagate(0) AND carry_in);
     inst: FOR i IN 1 TO 6 LOOP
         carry_in_internal(i+1)   <=   carry_generate(i)   OR   (carry_propagate(i)   AND
 carry_in_internal(i));
         END LOOP;
   carry_out <= carry_generate(7) OR (carry_propagate(7) AND carry_in_internal(7));
   END PROCESS;
   sum(0) <= h_sum(0) XOR carry_in;
   sum(7 DOWNTO 1) <= h_sum(7 DOWNTO 1) XOR carry_in_internal(7 DOWNTO 1);
```

END behavioral;

- **Write VHDL program for a full adder using two half adders implementation?**

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
ENTITY fulladd IS
     PORT (Cin, A, B,   : IN std_logic;
           COUT, sum    : OUT std_logic);
     end fulladd;
Architecture structure of full add IS
SIGNAL S1, C1           :   STD_LOGIC;
COMPONENT Halfadd
     PORT      (x, y,    : IN STD_LOGIC;
                S, C,    : OUT STD_LOGIC);
END COMPONENT;
BEGIN
     Stage0 : Halfadd Port map (X0, Y0, S0, C1);
     Stage1 : Halfadd Port map (Cin, S0, S1, C2);
     COUT <= C1 OR C2;
     sum <= S1;
END structure;
```

- **Write a VHDL code Design a 2X2 combinational multiplier? Ans)**

```
library ieee;
use ieee.std_logic_1164 all;
entity Arr_Mul is
port (A, B : in std_logic_vector (1 downto 0);
      P : out std_logic_vector (3 downto 0));
end Arr_Mul;

architecture MULT of Arr_Mul is
begin
      -- For simplicity propagation delay times are not considered in this example.
   P(0) <= B(0) and A(0);
   P(1) <= (B(0) and A(1)) xor (B(1) and A(0));
   P(2) <= (B(1) and A(1)) xor ((B(0) and A(1)) and (B(1) and A(0)));
   P(3) <= (B(1) and A(1)) and ((B(0) and A(1)) and (B(1) and A(0)));
end MULT;
```

❖ **Write a VHDL code Design a priority encoder for 16 inputs using two IC74X148s?**

```
Library IEEE;
use IEEE.STD_LOGIC_1164.all;
Entity Encoder is
        Port (
            EI_L      : in STD_LOGIC;
             I_L      : in STD_LOGIC_VECTOR(15 downto 0);
             A_L      : out STD_LOGIC_VECTOR(2 downto 0);
      EO_L, GS_L      : out STD_LOGIC);
         end Encoder;
         Architecture PriorityEncoder of Encoder is
         Signal EI   : STD_LOGIC;
         Signal  I   : STD_LOGIC_VECTORS(15 downto 0);
         Signal EO   : GS:STD_LOGIC;
         Signal  A   : STD_LOGIC_VECTOR(2 downto 0)
        begin
        Process (EI_L, I_L, EI, EO, GS, I, A)
        Variable j    : INTEGER range 15 downto 0;
        begin
        EI <= not EI_L;
         I <= not I_L;
        EO <= '1';
        GS <= 0;
         A <= 0000",
    if (EI) = '0' then EO <= "0";
    else for j in 15 downto 0 loop
    If I(j) = '1' then
       GS <= '1';
       EO <= '0';
        A <= CONV_STD_LOGIC_VECTOR (j,4);
       exit;
    end if
    end loop;
    end if;
       EO_L <= not EO;
       GS_L <= not GS;
        A_L <= not A;
    end process;
    end PriorityEncoder;
```

❖ **Write a VHDL code Design a 4-bit ripple carry adder?**

```
library ieee;
use ieee.std_logic_1164.all;
entity adder is
port (A, B : in std_logic_vector (3 downto 0);
    cin : in std_logic;
    sum : out std_logic_vector (3 downto 0);
    cout : out std_logic);
end adder;
architecture RCarry_adder of adder is
                                        --Assume 7.0-ns propagation delay for all gates.
    signal c1, c2, c3 : std_logic;
    constant delay_gt : time := 7 ns;
begin
    sum(0) <= (B(0) xor A(0)) xor cin after 2*delay_gt;
    sum(1) <= (B(1) xor A(1)) xor c1 after 2*delay_gt;
    sum(2) <= (B(2) xor A(2)) xor c2 after 2*delay_gt;
    sum(3) <= (B(3) xor A(3)) xor c3 after 2*delay_gt;

   c1 <= (A(0) and B(0)) or (A(0) and cin) or (B(0) and cin) after 2*delay_gt;
   c2 <= (A(1) and B(1)) or (A(1) and c1) or (B(1) and c1)  after 2*delay_gt;
   c3 <= (A(2) and B(2)) or (A(2) and c2) or (B(2) and c2)  after 2*delay_gt;
   cout <= (A(3) and B(3)) or (A(3) and c3) or (B(3) and c3) after 2*delay_gt;
end RCarry_adder;
```

❖ **Write a VHDL code Design a Dual priority encoder?**

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity Vprior2 is
    port (
        R: in STD_LOGIC_VECTOR (0 to 7);
        A, B: out STD_LOGIC_VECTOR (2 downto 0);
        AVALID, BVALID: buffer STD_LOGIC
    );
end Vprior2;

architecture Vprior2_arch of Vprior2 is
begin
```

```
process(R, AVALID, BVALID)
begin
  AVALID <= '0'; BVALID <= '0'; A <= "000"; B <= "000";
  for i in 0 to 7 loop
    if R(i) = '1' and AVALID = '0' then
      A <= CONV_STD_LOGIC_VECTOR(i,3); AVALID <= '1';
    elsif R(i) = '1' and BVALID = '0' then
      B <= CONV_STD_LOGIC_VECTOR(i,3); BVALID <= '1';
    end if;
  end loop;
end process;
end Vprior2_arch;
```

❖ **Write a VHDL code Design a 16-bit barrel shifter circuit?**

```
library ieee;
Use  ieee.std_logic_1164.all;
Use  ieee.std_logic_arith.all;
Entity barrel is
Port( din: in std_logic_vector (15 downto 0);
          S: in unsigned(3 downto 0);
        Dout: out std_logic_vector (15 downto 0));
End barrel;
Architecture behavioural of barrel is
Subtype dataword is std_logic_vector (15 downto 0);
Variable N: integer;
Variable tempd: dataword;
Begin
Procee(din, s)
Begin
N:=conv_integer(s);
 For I in 1 to N loop
tempd := tempd(14 downto 0) & tempd(15);
end loop;
Dout<=tempd;
End Process;
End behavioural;
```

❖ **Write a VHDL code Design a 1x4 demultiplexer?**

```
   library       ieee;
              use ieee.std_logic_1164.all;
entity Demux is
        port(en, d : in std_logic;
                s: in std_logic_vector(1 downto 0);
                              z: out std_logic_vector(3 downto 0));
        end Demux;
        architecture behavioural of Demux is
        signal z1 : std_logic_vector(3 downto 0);
```

```
process(en,d,s,z1)
begin
if (en=0) then z1<='0';
else
case s is
        when "00"=> z1(3)<=d;
        when "01"=> z1(2)<=d;
        when "10"=> z1(1)<=d;
        when "11"=> z1(0)<=d;
        when others=> nul;
end case;
end if;
end process;
z<= z1;
end behavioural;
```

- Write a VHDL code for 3:8 binary decoder using IC74X138

```
library      ieee;
        use ieee.std_logic_1164.all;
        entity dec38 is
        port(e1,e2,e3: in std_logic;
                        a: in std_logic_vector(2 downto 0);
                cout: out std_logic_vector(7 downto 0));
        end dec38;
        architecture  behavioural of dec38 is
        signal cout1:std_logic_vector(7 downto 0);
        begin
        process(a,e1,e2,e3)
        begin
        if( not e1 and not e2 and e3) ='0'
        then
        cout1<="11111111";
        else
        case a is
        when "000"=> cout1<="01111111";
        when "001"=> cout1<="10111111";
        when "010"=> cout1<="11011111";
        when "011"=> cout1<="11101111";
        when "100"=> cout1<="11110111";
        when "101"=> cout1<="11111011";
        when "110"=> cout1<="11111101";
        when "111"=> cout1<="11111110";
        when others=>cout1<="11111111";
        end case;
        end if;
        end process;
```

```
        cout<=cout1;
        end behavioural;
```

❖ **Write a VHDL code for 8x1 multiplexer IC 74x151?**

```
library ieee;
    use ieee.std_logic_1164.all;
    entity mux8 is
    port(a,b,c,d,e,f,g,h : in std_logic;
        strobe : in std_logic;
        ds: in std_logic_vector(2 downto 0);
                z: out std_logic);
    end mux8;
    architecture behavioural of mux8 is
    signal z1 : std_logic;
    begin
        process(strobe,a,b,c,d,e,f,g,h,ds)
    begin
        if strobe='1' then z1<='1';
            elsif strobe='0' then
                case ds is
                    when "000"=>z1<=not a;
                    when "001"=>z1<=not b;
                    when "010"=> z1<=not c;
                    when "011"=> z1<=not d;
                    when "100"=> z1<=not e;
                    when "101"=> z1<=not f;
                    when "110"=> z1<=not g;
                    when "111"=> z1<=not h;
                    when others=>z1<='1';
                end case;
            end if;
        end process;
        z<=z1;
    end behavioural;
```

UNIT-IV
SEQUENTIAL MACHINE DESIGN PRACTICES

❖ **2-bit asynchronous up counter:**

The 2-bit Asynchronous counter requires two flip-flops. Both flip-flop inputs are connected to logic '1', and initially both flip-flops are in reset position. The clock signal is connected to only flip-flop-A, the clock input for the flip-flop-B is applied from the output of flip-flop-A. Based on negative edge triggering of clock pulse, the output Q_A is obtained by toggle operation of Flip- flop-A because $J_A=K_A=1$ then it acts as toggle flip-flop. Now the Q_B output is obtained by depending on negative edges of Q_A. This operation is continued and it counts 4 clock pulses count such as "00", "01", "10", "11" . After completion of 4 clock pulses the fifth clock pulse gives initial count "00" again as shown in the figure 6. 28.

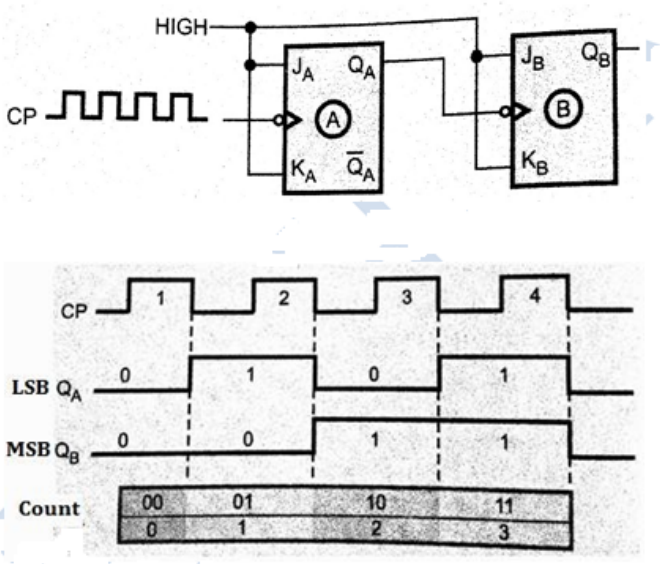


Figure 6.28(a): A 2-bit Asynchronous up counter

❖ **3-bit asynchronous up counter:**

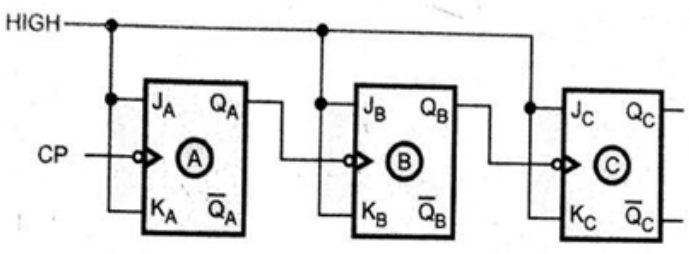


Figure 6.29(a): A 3-bit Asynchronous up counter

The 3-bit Asynchronous counter requires three flip-flops. All flip-flop inputs are connected to logic '1', and and initially all flip-flops are in reset position which gives $Q_{A,}= Q_{B,}= Q_C=0$. The clock signal is connected to only flip-flop-A, the clock input for the flip-flop-B is applied from the output of flip-flop-A and the clock input for the flip-flop-C is applied from the output of flip-flop-B. Based on negative edge triggering of clock pulse, the output Q_A is obtained by toggle operation of Flip-flop-A because $J_A=K_A=1$ then it acts as toggle flip-flop. Now the Q_B output is obtained by depending on negative edges of $Q_{A.}$ and the Q_C output is obtained by depending on negative edges of $Q_{B.}$ This operation is continued and it counts 8 clock pulse counts such as "000", "001", "010", "011", "100", "101", "110", "111". After completion of 8 clock pulses the 9^{TH} clock pulse gives initial count "000" again as shown in the figure 6. 29(b).

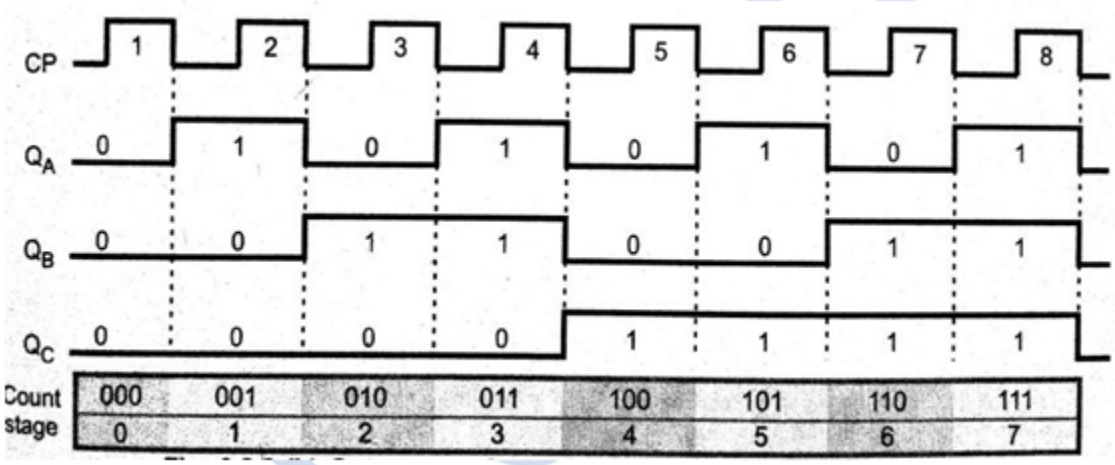


Figure 6.29(b): Timing waveforms of 3-bit Asynchronous up counter

❖ **4-bit Asynchronous Down counter**

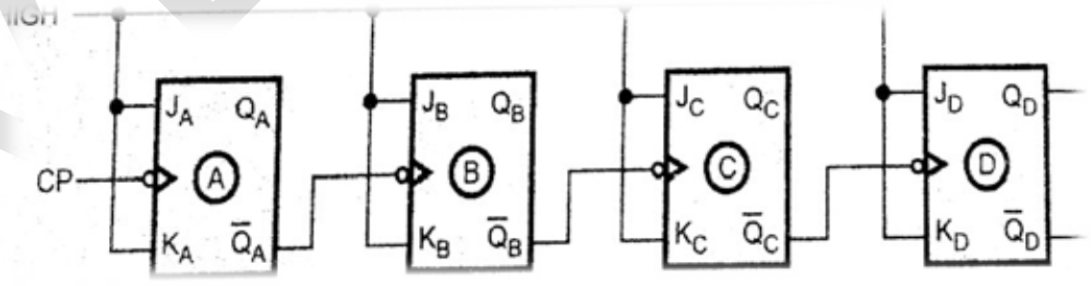


Figure 6.30(a): A 4-bit Asynchronous Down counter

The 4-bit Asynchronous counter down requires four flip-flops. All flip-flop inputs are connected to logic '1', and and initially all flip-flops are in reset position which gives $Q_{A,}= Q_{B,}= Q_C=0$. The clock

signal is connected to only flip-flop-A, the clock input for the flip-flop-B is applied from the output of flip-flop-A, the clock input for the flip-flop-C is applied from the output of flip-flop-B, the clock input for the flip-flop-D is applied from the output of flip-flop-C. Now to get down counting operation the complementary outputs are connected as clock input to its next higher order flip-flop, based on negative edge triggering of clock pulse, the output

is obtained by toggle operation of Flip-flop-A because $J_A=K_A=1$ then it acts as toggle flip-flop. So the output is obtained by depending on negative edges and the output is obtained by depending on negative edges of , and the output is obtained by dependingon negative edges of . This operation is continued and it counts 16 clock pulse counts such as "1111", "1110", "1101", "1100", "1011", "1010", "1001", "1000", "0111", "0110", "0101", "0100", "0011", "0010", "0001", "0000". After completion of 8 clock pulses the 9^{TH} clock pulse gives initial count "1111" again as shown in the figure 6. 30(b).

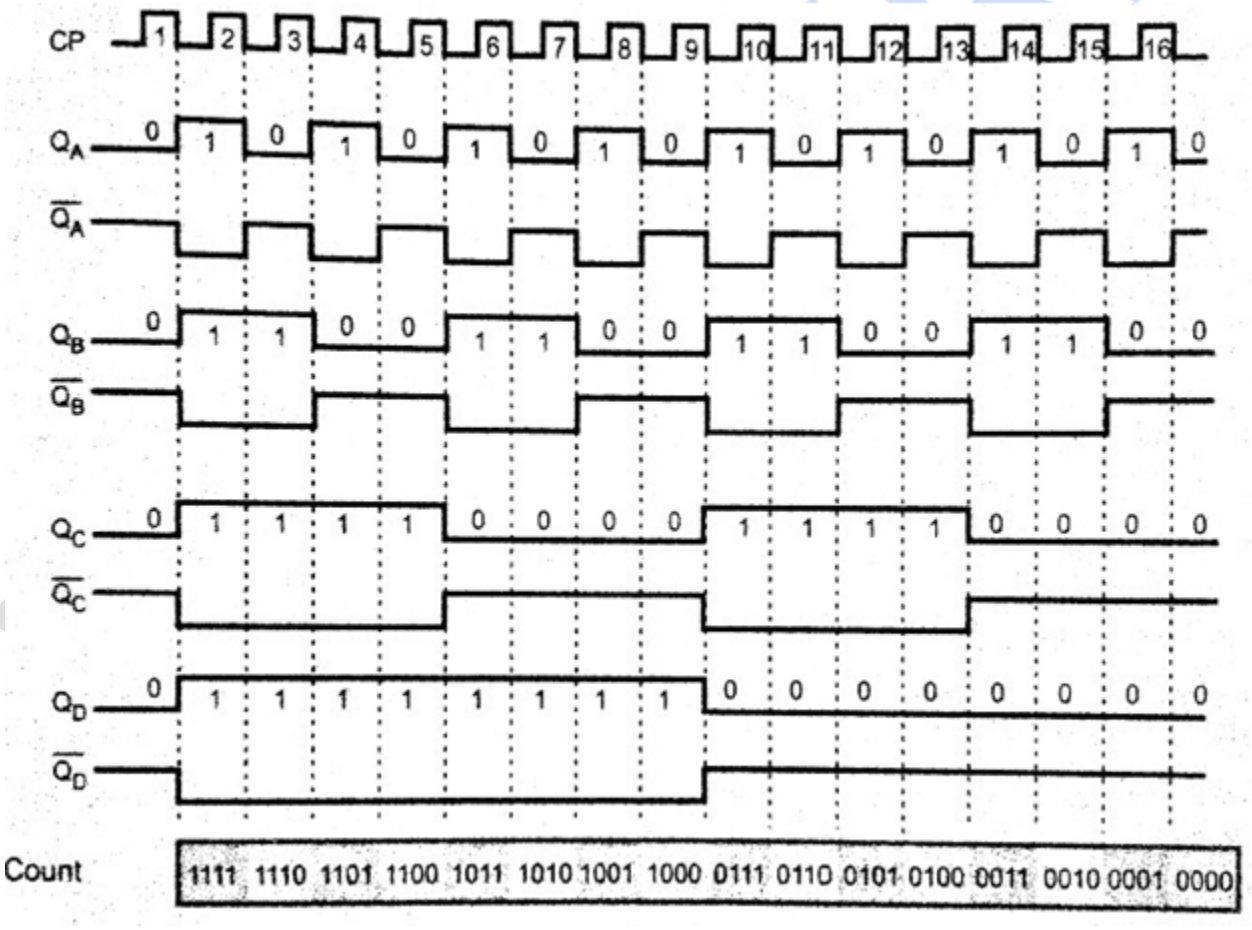


Figure 6.30(b): Timing waveforms of 4-bit Asynchronous up counter

16. Explain the operation of Asynchronous UP/DOWN counter with relevant example? Ans).

Using same circuitry we can develop both Up counting and Down counting operations. So in a single circuit to get two different operations we have to use a Mode control input which may decide whether the counter has to deliver either up counting or down counting operation. So to achieve that operation the following arrangement is constructed in between the Flip-flop output and the clock input of next stage flip-flop. When the control input M=0 it performs down counting operation and for M=1 it performs up counting operation.

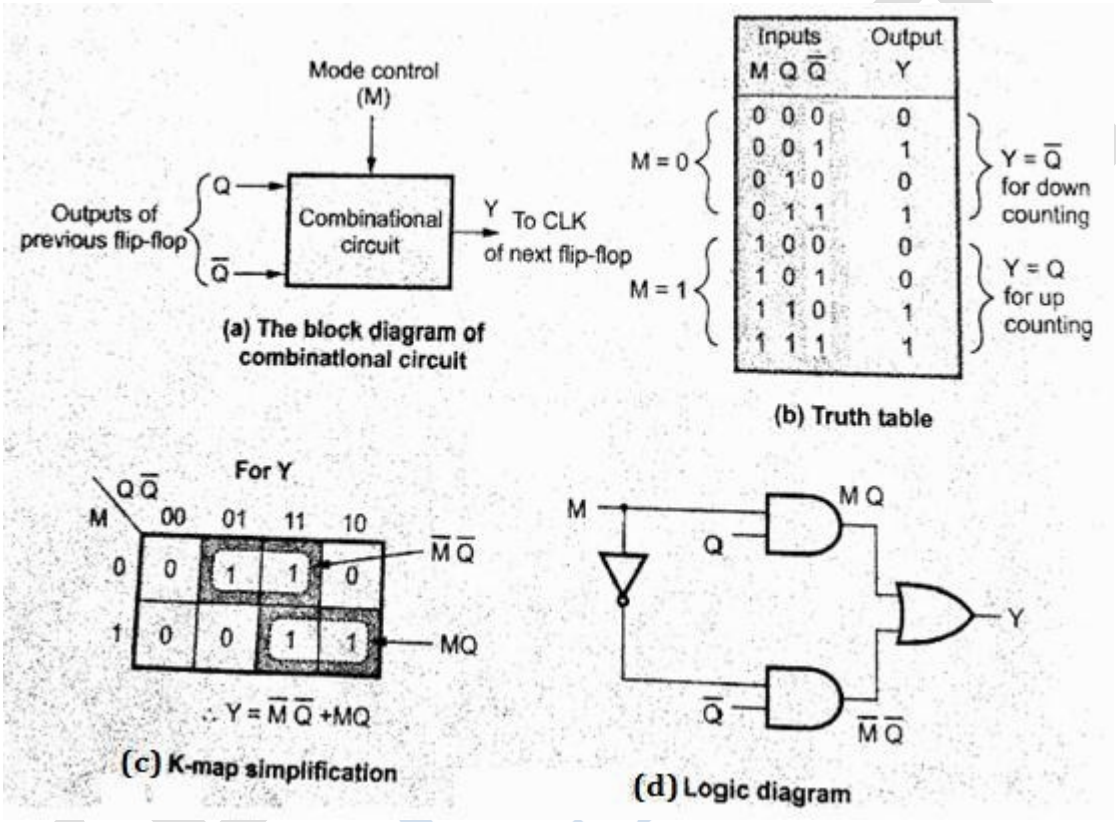


Figure 6.31: Logic implementation for Asynchronous

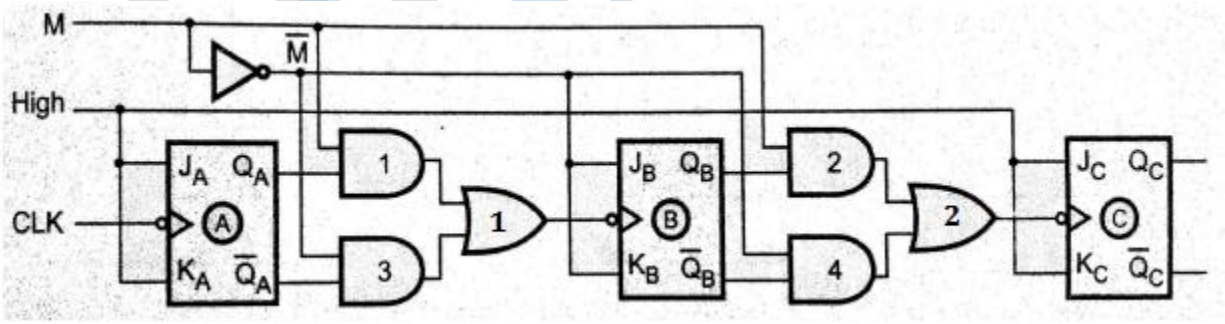


Figure 6.32: 3-bit Asynchronous

Operation:

Case(1): When M=0, then AND gate-1, AND agte-2 both provides output as '0' and AND gate-3

provides output as , AND gate-4 provides output as . Then OR gate-1 provides as clock input to the flip-flop-B and OR gate-2 provides as clock input to the Flip-flop-C. Hence we will get Down counting operation.

Case(2): When M=1, then AND gate-3, AND agte-4 both provides output as '0' and AND gate-1 provides output as Q_A, AND gate-2 provides output as Q_B .Then OR gate-1 provides Q_A as clock input to the flip-flop-B and OR gate-2 provides Q_B as clock input to the Flip-flop-C. Hence we will get Up counting operation.

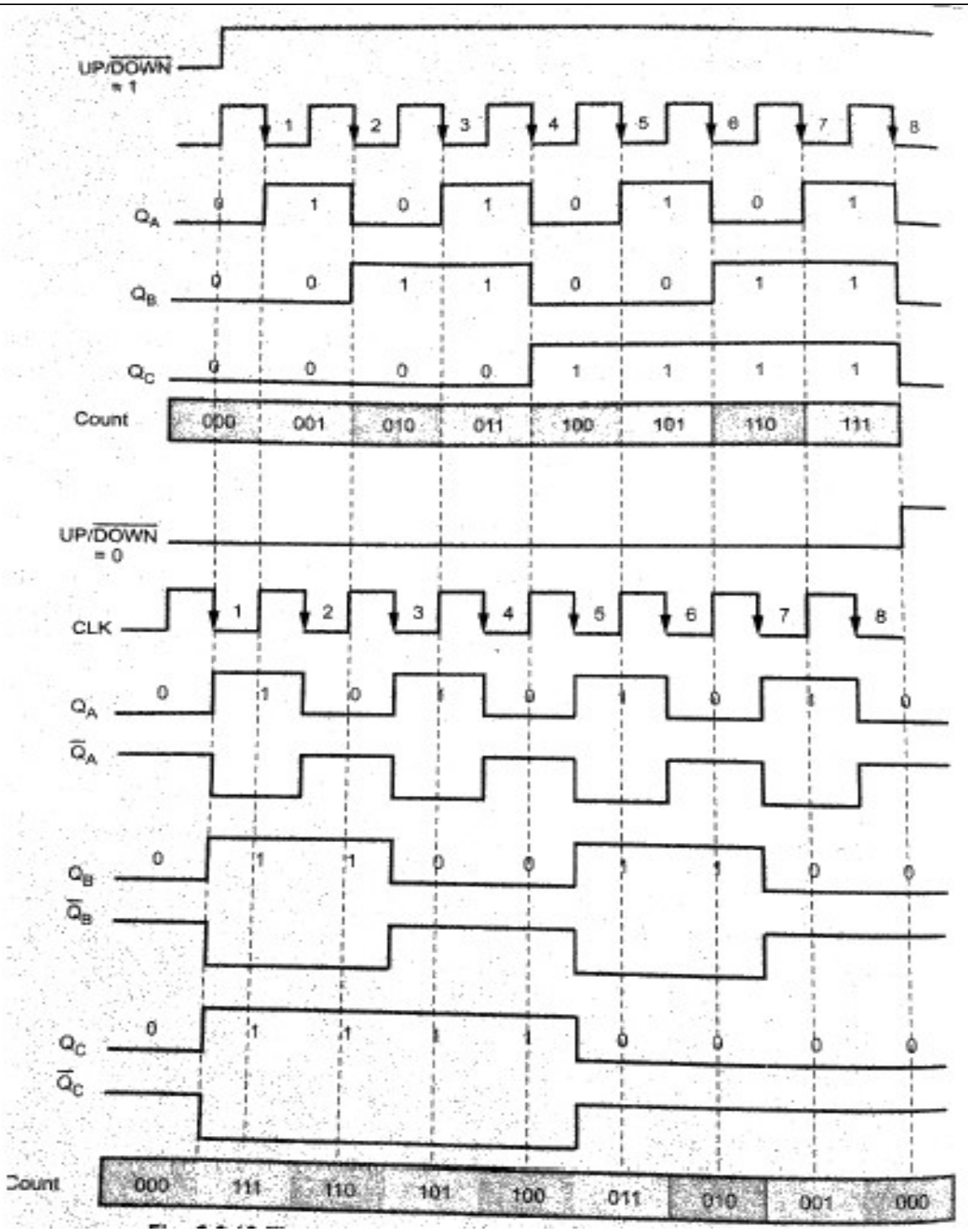
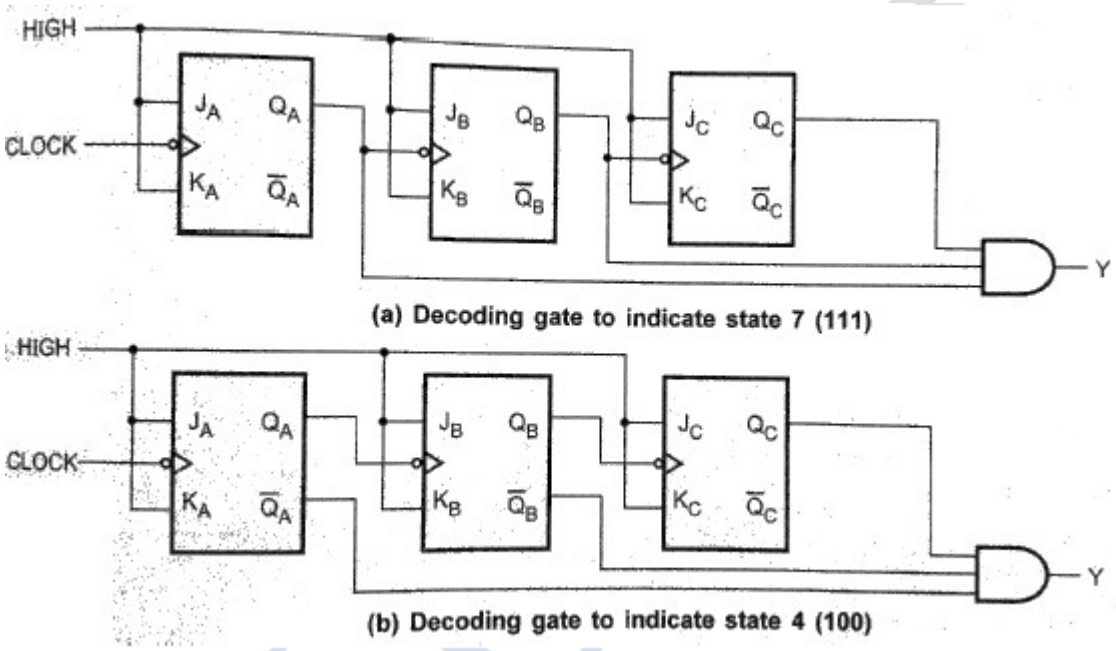


Figure 6.33: Timing waveforms of 3-bit Asynchronous Up/Down counter

❖ **Write short notes on the following.**

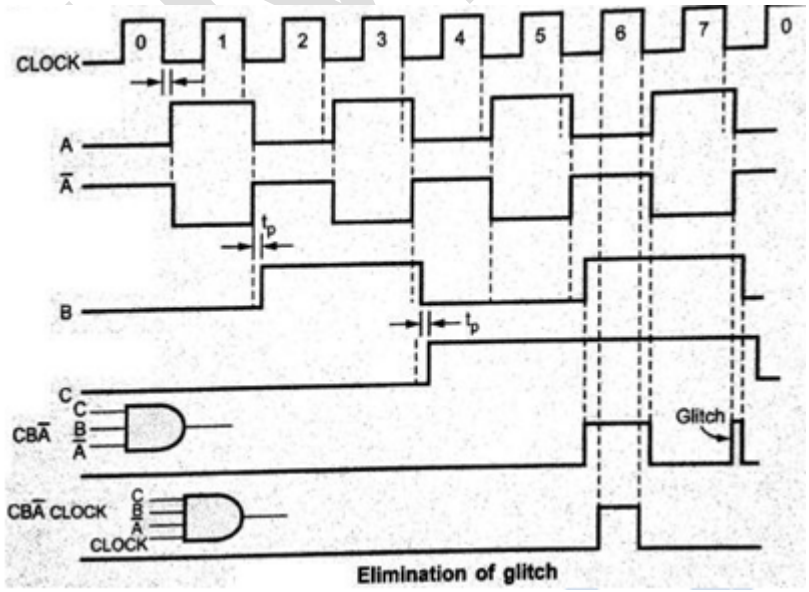
(i) Decoding gates in counters (ii) Glitch problem Ans)

(i) Decoding gates in counters: Decoding getes are used to indicate whether counter has reached to particular state or not. For this the outputs of counter are connected to the AND gate as inputs then the AND gate output gives high value for particular state. In figure 6.34,



(a) Decoding gate to indicate state 7 (111)

(b) Decoding gate to indicate state 4 (100)



Elimination of glitch

Figure 6.35: Glitch problem elimination in Asynchronous counters

❖ **What is synchronous counter? Explain its operation with timing waveforms?**

In a synchronous counter a synchronized clock pulse is applied to all flip-flops. So there is no

delay is present in the output production of final stage flip-flop. Initially both flip-flops are in reset position.

Example: The operation of 2-bit counter is as follows.

S. No	Condition	Operation
1	Initially let both the Flip-flops be in the reset state	QBQA = 00 initially
2	After 1st negative clock edge	As soon as the first negative clock edge is applied, Flip-flop-A will toggle and QA will change from 0 to 1. But at the instant of application of negative clock edge, QA , JB = KB = 0. Hence Flip-flop-B will not change its state. So QB will remain 0. QBQA = 01 after the first clock pulse
3	After 2nd negative clock edge	On the arrival of second negative clock edge, Flip-flop-A toggles again and QA changes from 1 to 0. But at this instant QA was 1. So JB = KB= 1 and Flip-flop-B will toggle. Hence QB changes from 0 to 1. QBQA = 10 after the second clock pulse.
4	After 3rd negative clock edge	On application of the third falling clock edge, Flip-flop-A will toggle from 0 to 1 but there is no change of state for Flip- flop-B.QBQA = 11 after the third clockpulse.
5	After 4th negative clock edge	On application of the next clock pulse, QA will change from 1 to 0 as QB will also change from 1 to 0. QBQA = 00 after the fourth clock pulse.

Table 6.25: Counting table of 2 bit synchronous up counter

A 2-bit Synchronous up counter

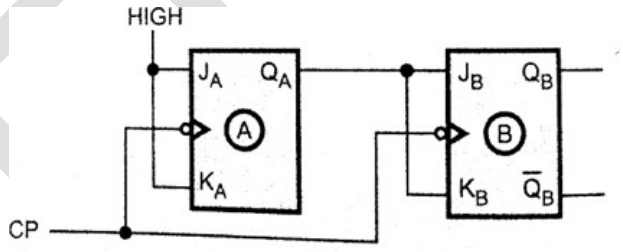


Figure 6.36(a): A 2-bit Synchronous up counter

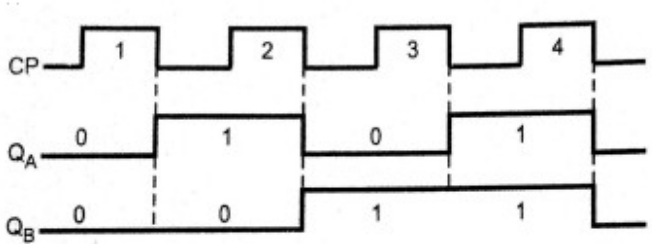


Figure 6.37(b): Timing waveforms of 2-bit Synchronous up counter

Example-3: Similar to the operation of 2-bit, 3-bit up counter the 4-bit synchronous up counter is also designed as follows.

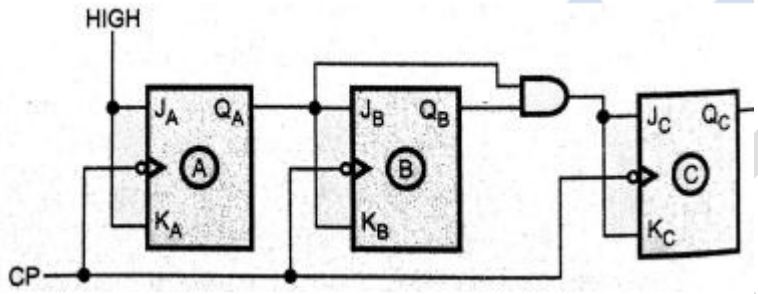


Figure 6.36(a): A 3-bit Synchronous up counter

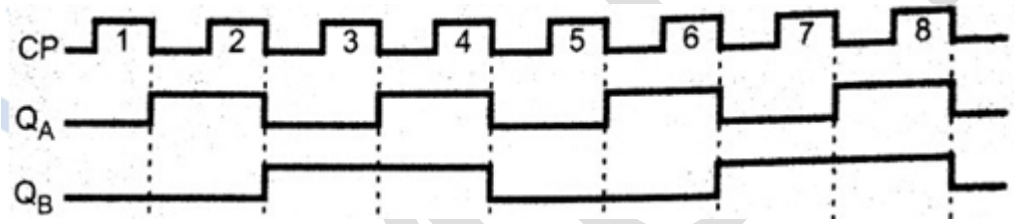


Figure 6.37(b): Timing waveforms of 3-bit Synchronous up counter

A 4-bit Synchronous up counter

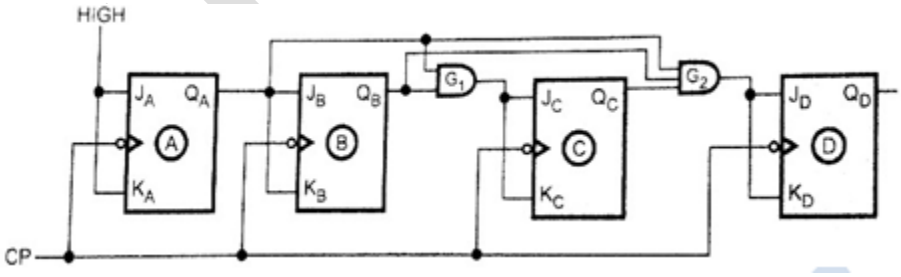


Figure 6.38(a): A 4-bit Synchronous up counter

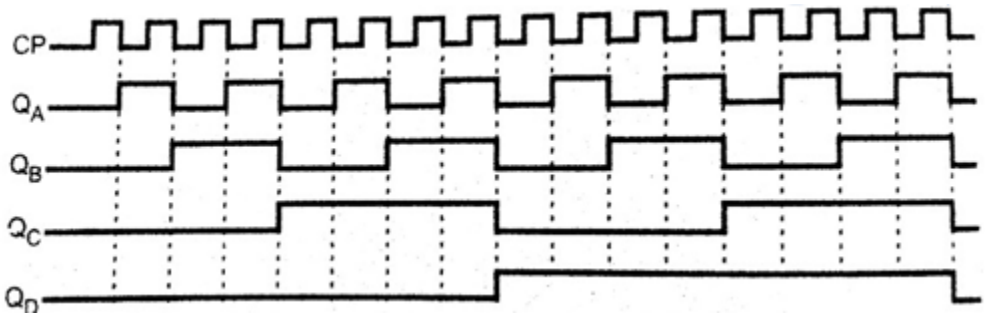


Figure 6.38(b): Timing waveform of 4-bit Synchronous up counter

❖ **Design and explain the operation of synchronous up/down counter?**

Within the same circuit there is a possibility of both up counting and down counting operations that is achieved by a mode control input 'M'. If M=0 it gives up counting and M=1 it gives down counting operations. To design a 3-bit synchronous up/down counter there is a requirement of 3 flip-flops. Their present state outputs are Q_A, Q_B, Q_C and Q_A^+, Q_B^+, Q_C^+ are the next state values respectively. Then the operation is observed from the truth table 6.26

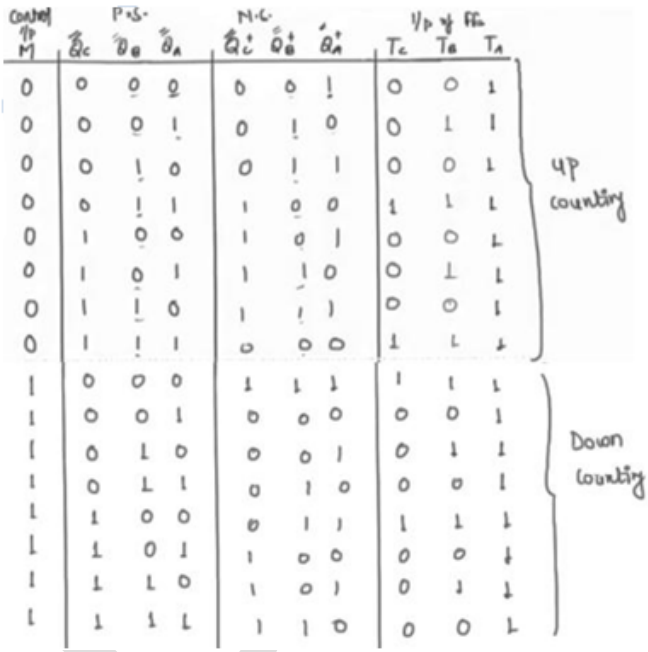
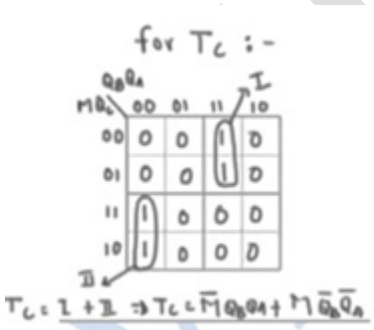
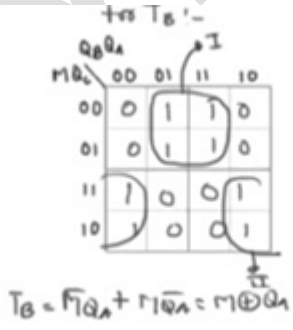
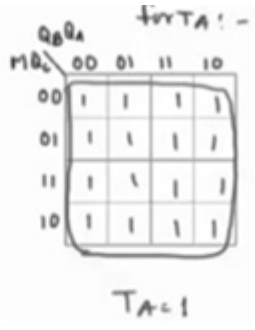


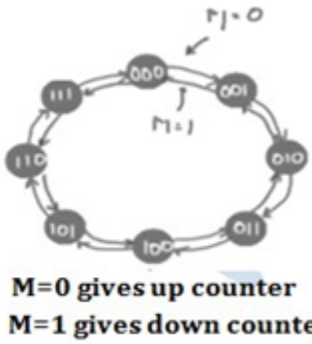
Table 6.26: 3 bit Synchronous Up/ Down counter

K-Map implementation:









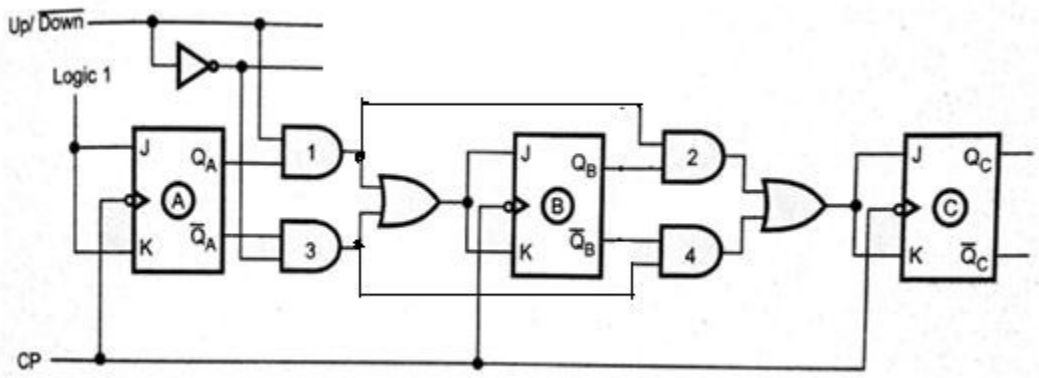


Figure 6.39: 3-bit Synchronous Up/Down counter

❖ **Define Moulo-N counter and design a MOD-5 counter using JK-flip-flop?**

The 2-bit ripple counter is called as MOD4 counter and 3-bit ripple counter is called as MOD8 counter. So in general, an n-bit ripple counter is called as modulo-N counter. Where,

MOD number = 2^n. MOD-5 counter can count values from "000" to "100". When the count reaches to "101" again it goes back to initial count "000".

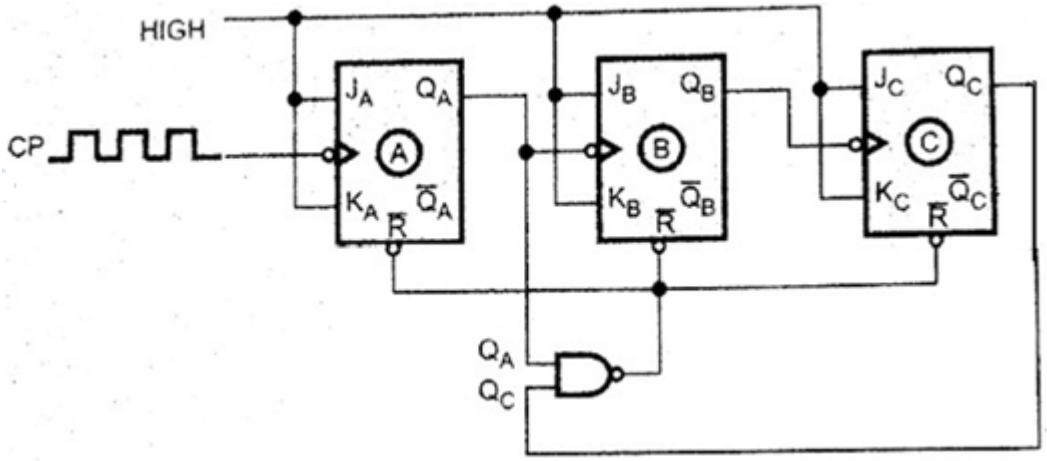


Figure 6.40(a): MOD-5 counter using RESET input

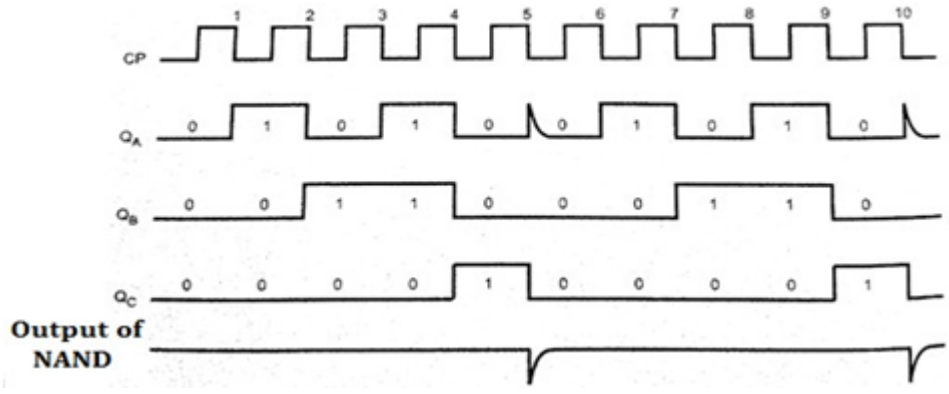


Figure 6.40(b): Timing waveforms of MOD-5 counter

The below figure gives the NAND gate input connections for MOD-N counters

NAND Gate Inputs	Counter
Q_A	MOD-1 Counter
Q_B	MOD-2 Counter
Q_A Q_B	MOD-3 Counter
Q_C	MOD-4 Counter
Q_A Q_C	MOD-5 Counter
Q_B Q_C	MOD-6 Counter
Q_A Q_B Q_C	MOD-7 Counter

Figure 6.40(c): NAND gate inputs for MOD-N counter

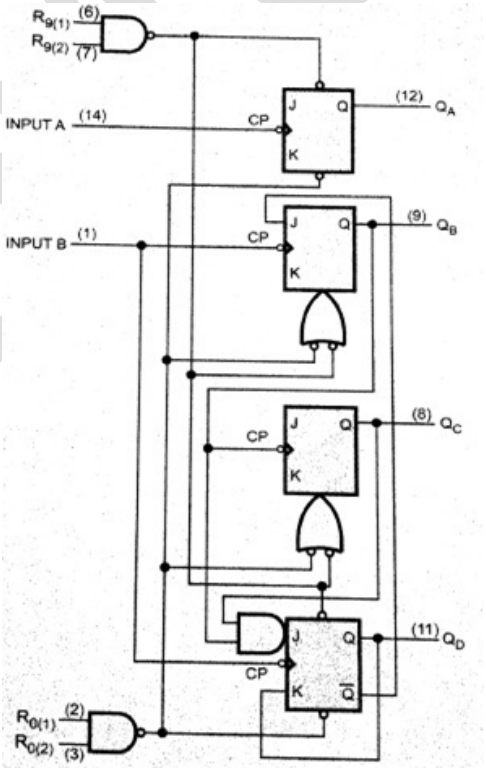


Figure 6.43: Logic diagram of IC7490

❖ **Explain the operation of 4-bit ripple counter using IC 7492/93? Modes of 7492:**

1.Modulo 12, Divide-by twelve Counter: The B input must be externally connected to the Qa output. The A input receives the incoming count and Qd produces a symmetrical divide-by-twelve square wave output.

2. Divide-By-Two and Divide-By-Six Counter : No external inter-connections are required. The first flip-flop is used as a binary element for the divide-by-two function. The B input is used to obtain the divide-by-three operation at the Q_B and Q_C outputs and divide by six operation at the Q_D output.

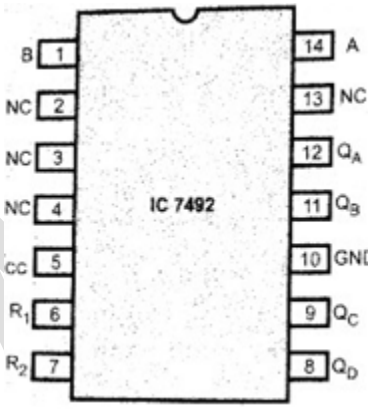
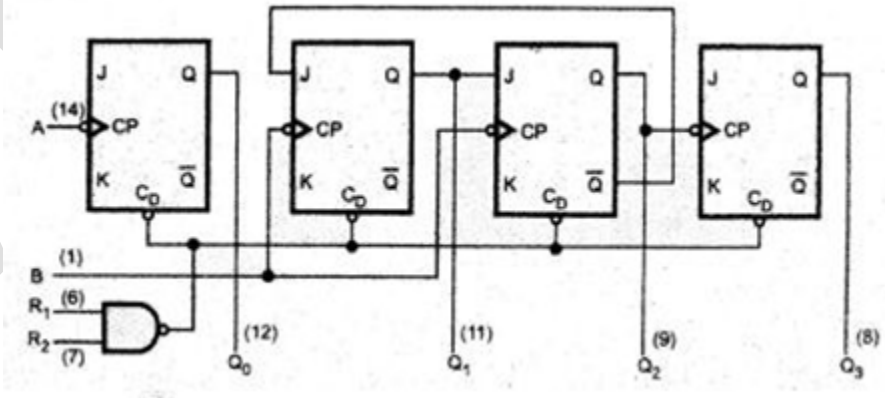


Figure 6.44(a): Pin diagram of IC 7492 (4-bit ripple counter) (b): Logic diagram of IC 7492 (4-bit ripple counter)



Output Q_A is connected to input Table 6.28: Truth table for IC 7492

Count	Output			
	Q_A	Q_B	Q_C	Q_D
0	L	L	L	L
1	H	L	L	L
2	L	H	L	L
3	H	H	L	L
4	L	L	H	L
5	H	L	H	L
6	L	L	L	H
7	H	L	L	H
8	L	H	L	H
9	H	H	L	H
10	L	L	H	H
11	H	L	H	H

IC7493

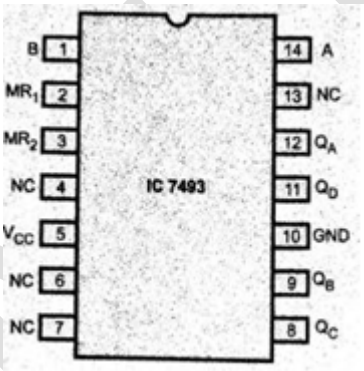
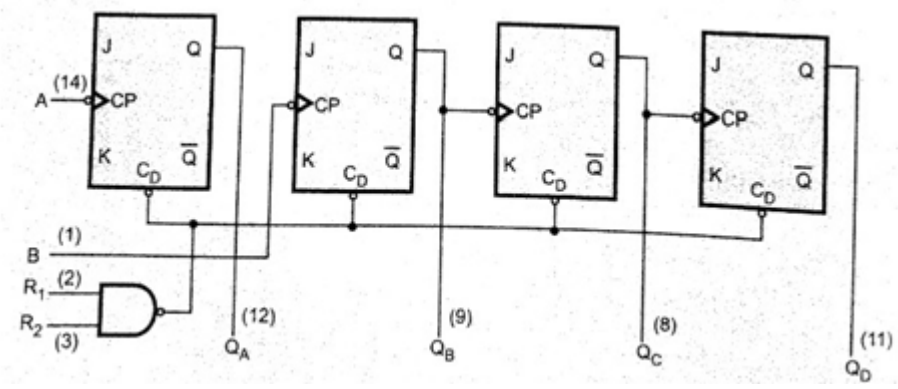


Figure 6.45(a): Pin diagram for IC7493



(b): Logic diagram for IC7493

Output Q_A is connected to input B

	Q_A	Q_B	Q_C	Q_D
0	L	L	L	L
1	H	L	L	L
2	L	H	L	L
3	H	H	L	L
4	L	L	H	L
5	H	L	H	L
6	L	H	H	L
7	H	H	H	L
8	L	L	L	H
9	H	L	L	H
10	L	H	L	H
11	H	H	L	H
12	L	L	H	H
13	H	L	H	H
14	L	H	H	H
15	H	H	H	H

Table 6.29: Truth table for IC 7492

❖ **Explain the operation of 4-bit Synchronous binary counter using IC74x163.**

Ans)

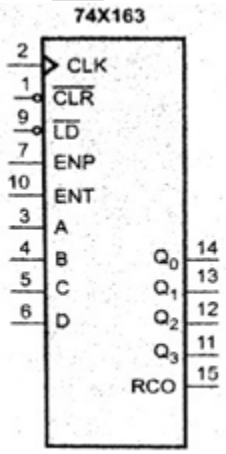


Figure 6.47: Pin diagram of IC74x163 (4-bit Synchronous binary counter)

i. IC 7493 is a 4-bit Synchronous binary counter designed with active low load and clear inputs

ii. It uses D flip-flops to perform load and clear functions. Each D input is driven by a 2- input multiplexer form by the combination of an OR gate and two AND gates.

iii. The multiplexer input is '0' if the clr input is applied as active low. If LD is applied as

active low signal then top AND gates passes 4 inputs A,B,C and D to the output.

iv. One input of XNOR gate corresponds of one count bit either Q_A, Q_B, Q_C and Q_D.

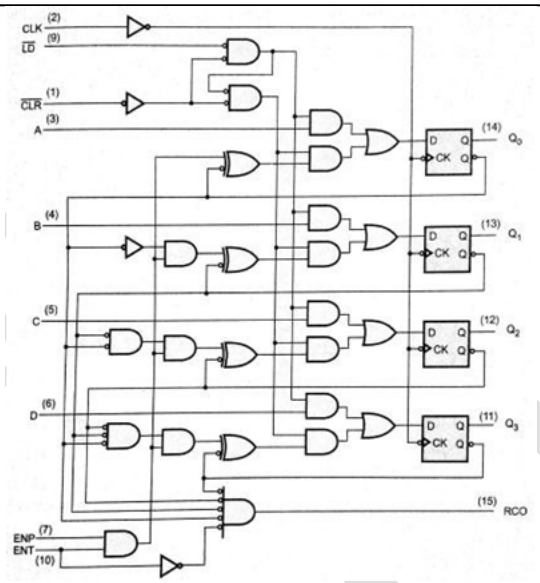
v. The XNOR gate gives complement output if and only if both enable ENP and ENT are maintained.

vi. The RCO signal indicates a carry from the most significant bit position.

Operating in free running mode: In this mode enable inputs are enabled continuously. A free running mode 74163 IC can be used for divide by 2, divide by 4, divide by 8 or divide by 16 counter.

Inputs				Current State				Next State			
$\overline{CLR}$	$\overline{LD}$	ENT	ENP	Q_3	Q_2	Q_1	Q_0	Q_{3^+}	Q_{2^+}	Q_{1^+}	Q_{0^+}
0	x	x	x	x	x	x	x	0	0	0	0
1	0	x	x	x	x	x	x	D	C	B	A
1	1	0	x	x	x	x	x	Q_3	Q_2	Q_1	Q_0
1	1	x	0	x	x	x	x	Q_3	Q_2	Q_1	Q_0
1	1	1	1	0	0	0	0	0	0	0	1
1	1	1	1	0	0	0	1	0	0	1	0
1	1	1	1	0	0	1	0	0	0	1	1
1	1	1	1	0	0	1	1	0	1	0	0
1	1	1	1	0	1	0	0	0	1	0	1
1	1	1	1	0	1	0	1	0	1	1	0
1	1	1	1	0	1	1	0	0	1	1	1
1	1	1	1	0	1	1	1	1	0	0	0
1	1	1	1	1	0	0	0	1	0	0	1
1	1	1	1	1	0	0	1	1	0	1	0
1	1	1	1	1	0	1	0	1	0	1	1
1	1	1	1	1	0	1	1	1	1	0	0
1	1	1	1	1	1	0	0	1	1	0	1
1	1	1	1	1	1	0	1	1	1	1	0
1	1	1	1	1	1	1	0	1	1	1	1
1	1	1	1	1	1	1	1	0	0	0	0

Table 6.30: Functional table for IC 74163



Functional logic diagram for IC 74163

❖ **Write and discuss the counter applications?**

Counters are mainly used in many areas some of them are

- Frequency counters
- Digital clock
- Time measurement
- A to D converter
- Frequency divider circuits
- Digital triangular wave generator

Digital clock:

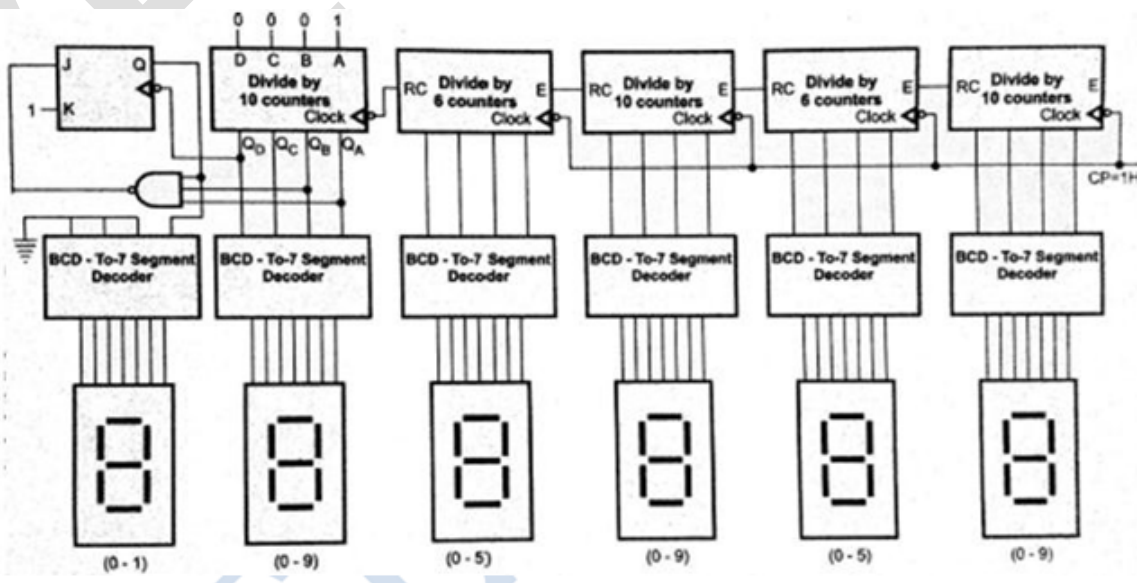


Figure 6.52: Circuit diagram of Digital clock

- A digital clock displays the seconds, minutes and hours.
- To design a digital clock we require three divide by 10 counters, two divide by 6 counters and a JK-Flip-flop.
- The combination of divide by 10 and divide by 6 counter forms divide by 60 counter.
- This counter counts seconds from 0-59 and one more divide by 60 counter counts 0-59 minutes.
- And third divide by 10 counter and a JK-flip-flop gives hours count from 0-12 as 12- format time.
- The outputs of counter are connected to BCD to seven segment drivers, which generates required signals to display input BCD counts on the seven segment displays.

Frequency Counter:

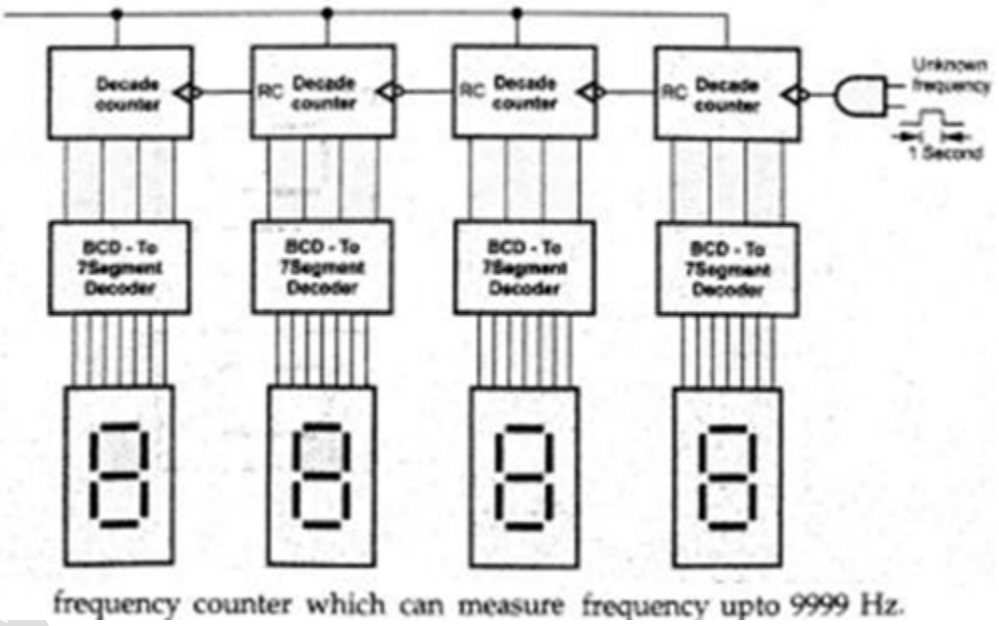


Figure 6.53: Frequency Counter

- A frequency counter is a circuit that can measure and display the frequency of a signal.
- The basic frequency counter circuit consists of counter circuit with decoder/ display circuitry and an AND gate.
- The AND gate inputs include the pulses with unknown frequency and a sample pulse with a known duration which controls how long the pulses with unknown frequency are allowed to

pass through the AND gate into the counter.

❖ **What is register? Explain different MSI buffer registers?**

A flip-flop is a memory cell used to store 1-bit of data. To store multiple data bits we require multiple flip-flops. The group of flip-flops is called as a REGISTER. A simplest register that designed by a group of D-Flip-flops is also called as Buffer Register. In the following example four D-Flip-flops are connected with common clock signal that all are stored multiple data bits at a time.

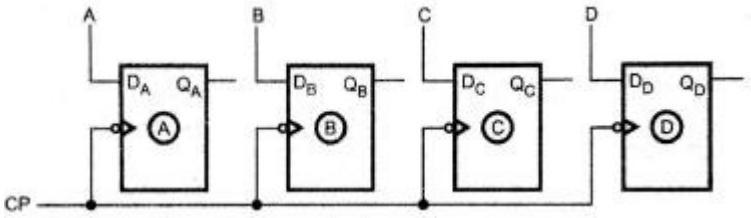


Figure: 6.56: 4-bit Register with negative Edge triggered D-Flip-flop

MSI Registers:

a) IC 74X175: It contains a four negative edge triggered D-Flip-flops with common clock and asynchronous active low clear inputs. It provides both active high and low outputs.

b) IC 74X174: This is a 6-bit register which contains six D-Flip-flops with comman active high clock input and Asynchronous activelo clear input. It has only active high outputs but no active low outputs.

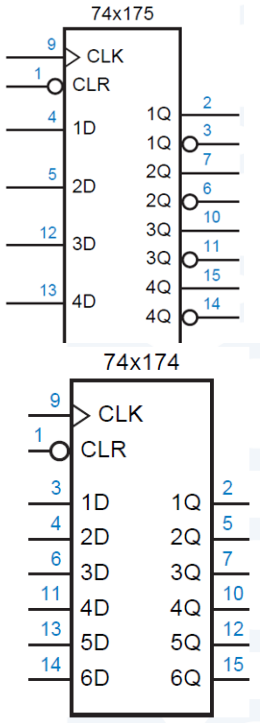


Figure 6.57: Six bit register

c) IC 74X374: This is a 8-bit register which contains eight D-Flip-flops with comman active high clock input and active low output enable. It has only active high outputs but no active low outputs. The outputs are collected through tri-state buffers. They are enabled by active low output enable.

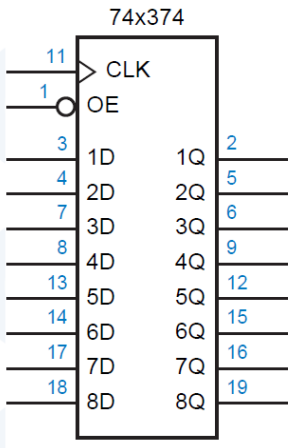


Figure 6.58: Eight bit register.

d) IC74X373: This is similar to IC74X374 except that it uses D- Latches instead of edge-triggered flip-flops. Therefore its output follow the corresponding inputs whenever C is declared and they latch the last input values when C is neglected.

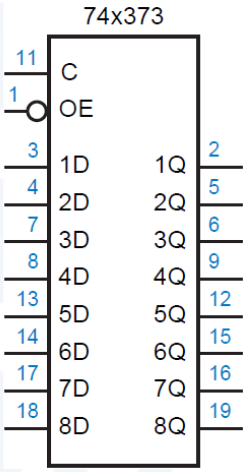
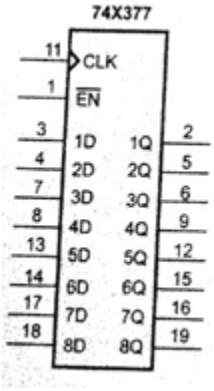


Figure 6.59: Eight bit register with D-Latches

e) IC74X273: This is a 8-bit register with a non tri-state outputs and non-active low Output enable input instead it provides an asynchronous active low clear input

f) IC 74X377: This is an edge-triggered register like 74X374, but it does not contain tri-state outputs instead it provides active low clock enable input.



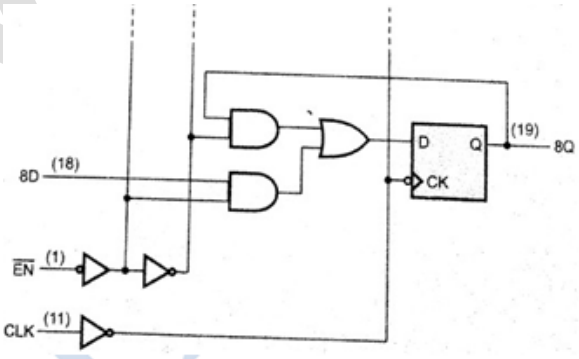


Figure 6.61: Eight bit register with D-Latches and without tri-state buffers

❖ **Define shift register and explain shift register modes?**

In a register the binary information can be moved from stage to stage with the register or out of the register upon application of clock pulses. These group of registers are called Shift Registers. Shift registers are operated in four modes. They are

- Serial-In Serial-Out shift mode (SISO).
- Serial-In Parallel-Out shift mode (SIPO).
- Parallel-In Parallel-Out shift mode (PIPO).
- Parallel-In Serial-Out shift mode (PISO).

Serial-In Serial-Out shift mode: The shifting operation is achieved in two ways either shifting data from left to right or from right to left. Figure 6.62(a) shows left to right shift and

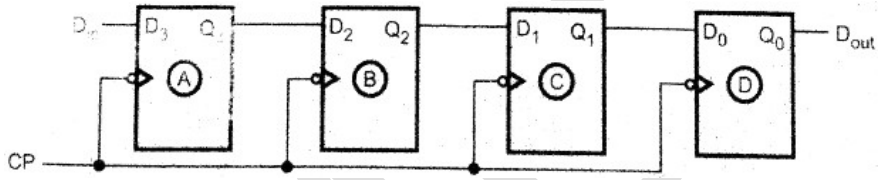


Figure 6.62(a): Serial in Serial out shift register- left to right shift

Figure 6.62(b) shows the right to left shift operation. let input as 4-bits data that are applied at input of D-Flip-flop. Only one data bit is applied to each flip-flop, here only D-flip-flop is elected because input to the D-flip-flop is equal to its next state value. In SISO mode initially each flip-flop output is '0'. By applying clock cycle bit by bit stored by shifting each output of flip-flop to its next stage flip-flop. For example the input data is '1111' then it requires four clock cycles to store. All the stored data bits are collected at the output of last stage flip-flop.

input data	Qa Qb Qc Qd	output collection
1 1 1 1	0 0 0 0	
0 1 1 1	→ 1 0 0 0	
0 0 1 1	→ 1 1 0 0	
0 0 0 1	→ 1 1 1 0	
0 0 0 0	→ 1 1 1 1	
	0 1 1 1 →	1 0 0 0
	0 0 1 1 →	1 1 0 0
	0 0 0 1 →	1 1 1 0
	0 0 0 0 →	1 1 1 1

Table 6.31: operation of shift register (left to right shift)

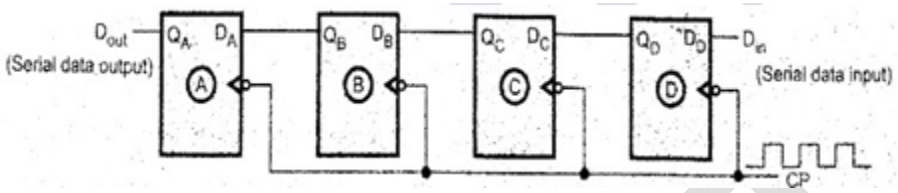


Figure 6.62(b): Serial in Serial Out shift register-right to left shift

Serial In-Parallel Out shift mode: let input as 4-bits data that are applied at input of D-Flip- flop. Only one data bit is applied to each flip-flop, here only D-flip-flop is elected because input to the D-flip-flop is equal to its next state value. In SIPO mode initially each flip-flop output is '0'. By applying clock cycle bit by bit stored by shifting each output of flip-flop to its next stage flip-flop. For example the input data is '1111' then it requires four clock cycles to store. Each flip-flop output is collected in parallel.

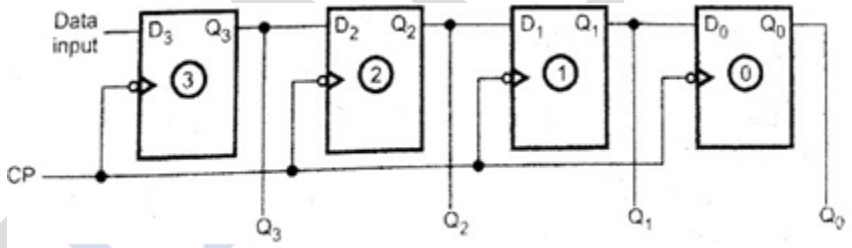


Figure 6.63: Serial in Parallel Out shift register-left to right shift

Parallel In-Parallel Out shift mode: In this mode the data is applied to the register in parallel and collected in parallel. That is each data bit is applied to a D-Flip-flop and collected from same flip-flop at applied clock pulse. This operation of Parallel In- Parallel Out mode of operation is observed in the following arrangement. In this mode there is no shifting is involved, here only data is loaded in to the flip-flops.

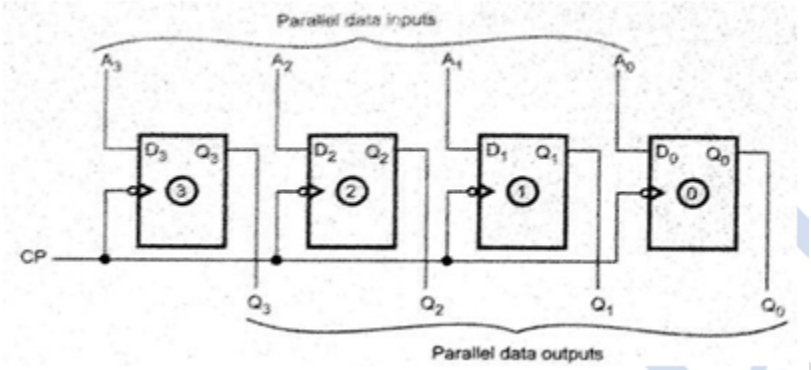


Figure 6.64: Parallel in Parallel Out shift register

Parallel in Serial Out shift Mode: In this mode of operation data is applied in parallel and entire output data is collected at output of last stage flip-flop i.e., in serial**.** So this design involves both parallel loading of data into flip-flops and shifting of data in serial. Hence by using same circuit we can get two operations such as parallel loading and shifting data in serial. This is achieved by a mode control input Shift/ . If Shift/ = 1, then active high shift line applies to AND gates G_4, G_5 and G_6 and another

input line for these AND gates are getting values from output of flip-flop. At that time the AND gates G_1, G_2 and G_3 gives output as '0', then OR gate gives output as Q_3, Q_2, Q_1, Q_0, So finally we will get shift operation. If Shift/ = 0, then active low load line applies to AND gates G_1, G_2 and G_3 and another input line for these AND gates are getting values from parallel applied input lines. At that time the AND gates G_4, G_5 and G_6 gives output as '0', then OR gate gives outputs as parallel loaded data. So finally we will get parallel load operation.

17. What is Bi- directional shift register and explain its operation?

Ans) If a shift register shifts data either from left to right or from right to left then it is called Uni-directional shift register. If shift register shifts the data in both directions then it is called Bi-directional shift register. To get two shift operations a control input is required named as Right/ . It has two serial inputs as *Serial data input for Right-shift* and *Serial data input for Left shift.*

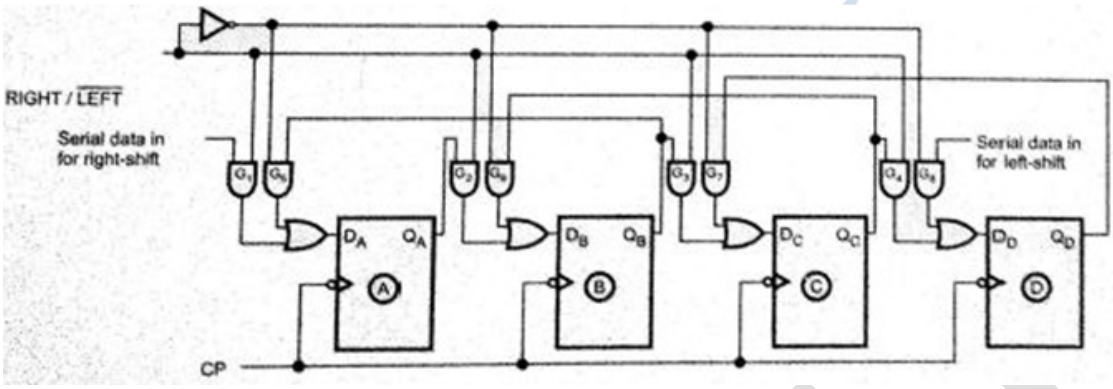


Figure 6.66: Bi-directional shift register

If Right/ =1, then active high right line is connecting logic 1 to AND gates G_1, G_2, G_3, and G_4 which gives left to right shifting operation. Whatever the *Serial data input for Right-shift* is applied through OR gate to D-flip-flop that will be shifted to right side depending on control input Right/ .

If Right/ =0, then active low left line is connecting logic 0 to AND gates G_5, G_6, G_7, and G_8, which gives right to left shifting operation. Whatever the *Serial data input for Left-shift* is applied through OR gate to D-flip-flop that will be shifted to Left side depending on control input Right/ .

❖ **Explain the operation of Universal shift register?**

A shift register is called as Universal shift register if has both bi-directional shift register with parallel load condition and satisfies four mode of operations such as SISO, SIPO, PISO, PIPO. The implementation of Universal shift register consists of multiplexers and flip-flops with asynchronous active low clear inputs. The number of multiplexers and flip-flops are depends on the number of data inputs that are applying. We know the multiplexer is operated with selection inputs, so here for 4-bits of data processing four 4X1 multiplexers are connected with common selection inputs S_0, S_1.

Case-1: When $S_0 = S_1 = 1$, then multiplexer input 3 is activated, to that input lines the input data I_0, I_1, I_2 and I_3 are applied in parallel. So in this case the operation is Parallel loading.

Case-2: When $S_0 = 1$, $S_1 = 0$, then multiplexer input 2 is activated, to that input serial input for shift left is applied and each stage flip-flop output is applied to input 2 of multiplexer. So the finalized operation in this case is right to left shift operation.

Case-3: When $S_0 = 0$, $S_1 = 1$, then multiplexer input 1 is activated, to that input serial input for

Selection Inputs		Register Operation
S_0	S_1	
1	1	Parallel Load
1	0	Shift Left
0	1	Shift Right
1	1	No change

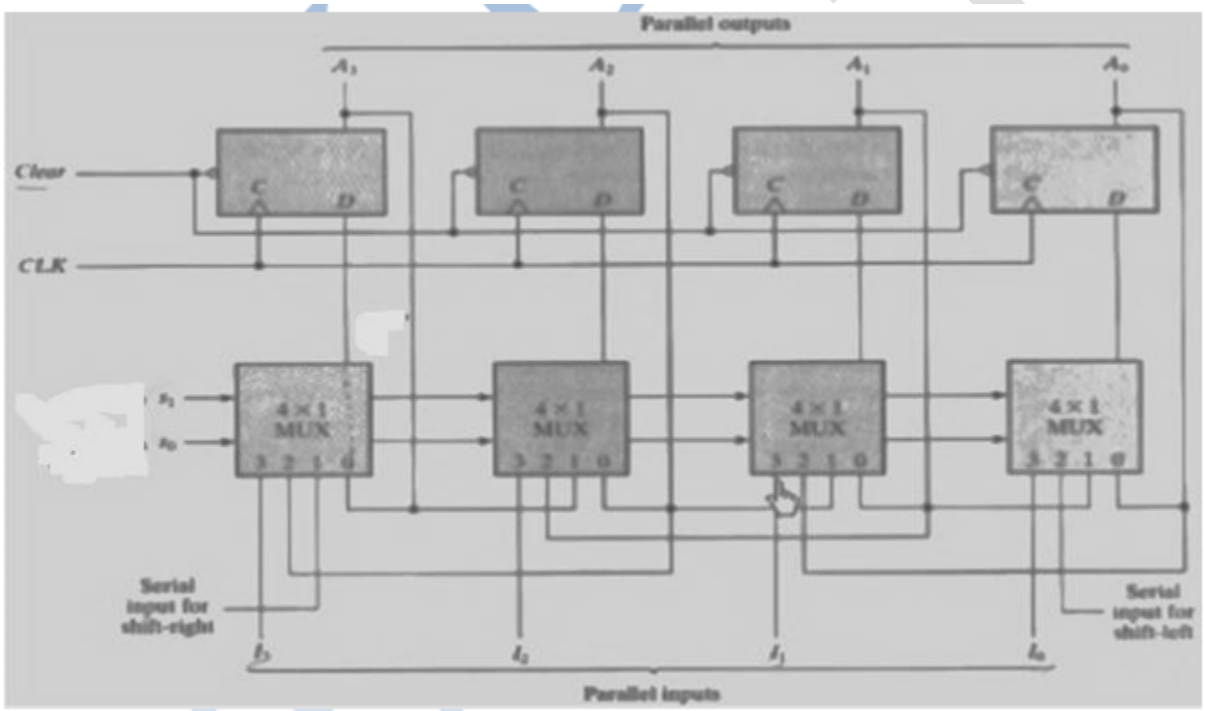


Figure 6.67: Universal shift register

❖ **What are the applications of shift registers? Explain.**

The first application of shift register is temporary data storage and bit manipulations.

The following are the some more applications of shift registers. They are

i. Delay line.

ii. Serial to Parallel Converter.

iii. Parallel to Serial Converter.

❖ **Explain the operation of parallel access shift register (IC 7495)?**

Parallel loading is accomplished by applying the four bits of data and taking the mode control input high. When the mode control input is held high, the AND gate on the right input to each NOR gate is enabled while the left AND gate is disabled. The data is loaded into associated ff and

appears at the output after negative transition of clock-2 input. During parallel loading, the entry of serial data is inhibited.

For left-shift operation, serial data input must be connected to the data input pin D, as shown in Fig. 6.71. It is also necessary to connect Q_D to C, Q_C to B, and Q_B to A. Now when the mode control input is held high, a data bit will be entered into flip-flop Q_D and each stored data bit will be shifted one flip-flop to the left on each negative clock 2 transition.

There are two separated clocks for left shift and right shift operation, When separate clocks are not required, the clock input may be applied simultaneously to clock 1 and clock 2.

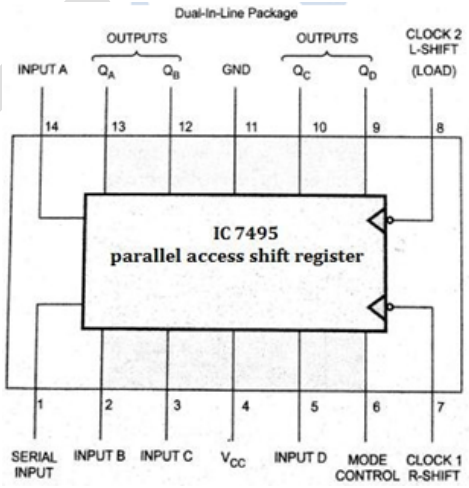


Figure 6.70: Pin diagram of IC 74X95

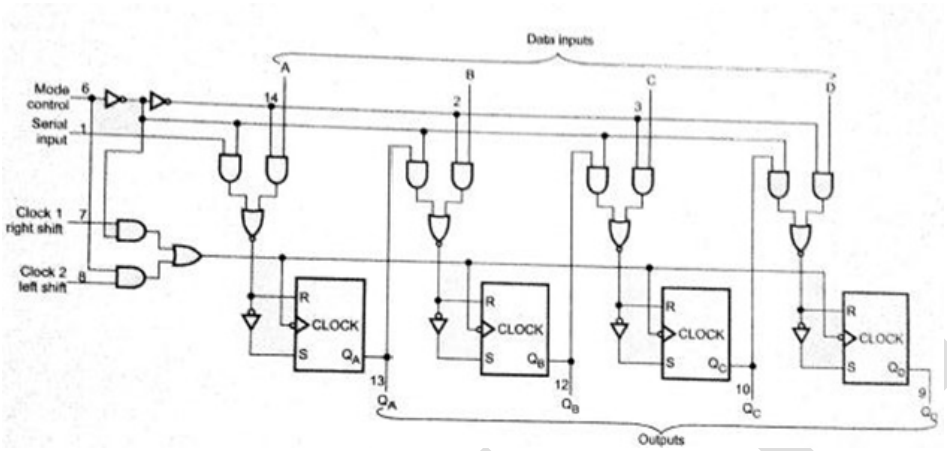


Figure 6.71: Circuit diagram of IC 74X95

❖ **Explain the implementation and operation of USR(Universal Shift Register) using IC 74194?**

A shift register is called as Universal shift register if has both bi-directional shift register with parallel load condition and satisfies four mode of operations such as SISO, SIPO, PISO, PIPO. The IC 74x194 has 4-bit universal shift register. It has 4 parallel data inputs (D_0-D_3) and S_0, S_1 are the control inputs.

Case-1: When $S_0 = S_1 = 1$, then multiplexer input 3 is activated, to that input lines the input data I_0, I_1, I_2 and I_3 are applied in parallel. So in this case the operation is Parallel loading.

Case-2: When $S_0 = 1$, $S_1 = 0$, then multiplexer input 2 is activated, to that input serial input for shift left (D_{SL}) is applied and each stage flip-flop output is applied to input 2 of multiplexer. So the finalized operation in this case is right to left shift operation.

Case-3: When $S_0 = 0$, $S_1 = 1$, then multiplexer input 1 is activated, to that input serial input for shift right (D_{SR}) is applied and each stage flip-flop output is applied to input 1 of multiplexer. So the finalized operation in this case is left to right shift operation.

Case-4: When $S_0 = S_1 = 0$, then multiplexer input 0 is activated whatever the data present in the previous stage that is obtained at the output. So in this case the operation has no change of data.

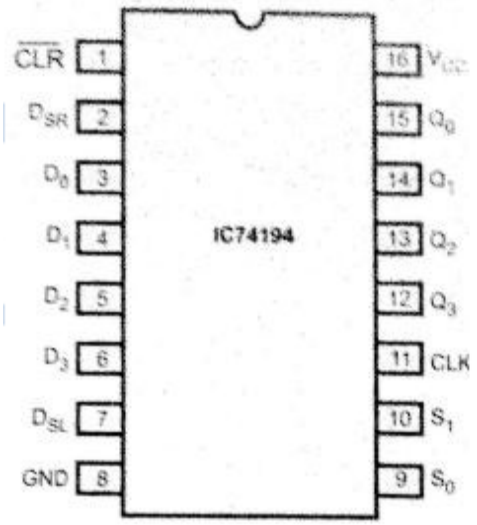


Figure 6.75: Pin diagram of IC 74x194

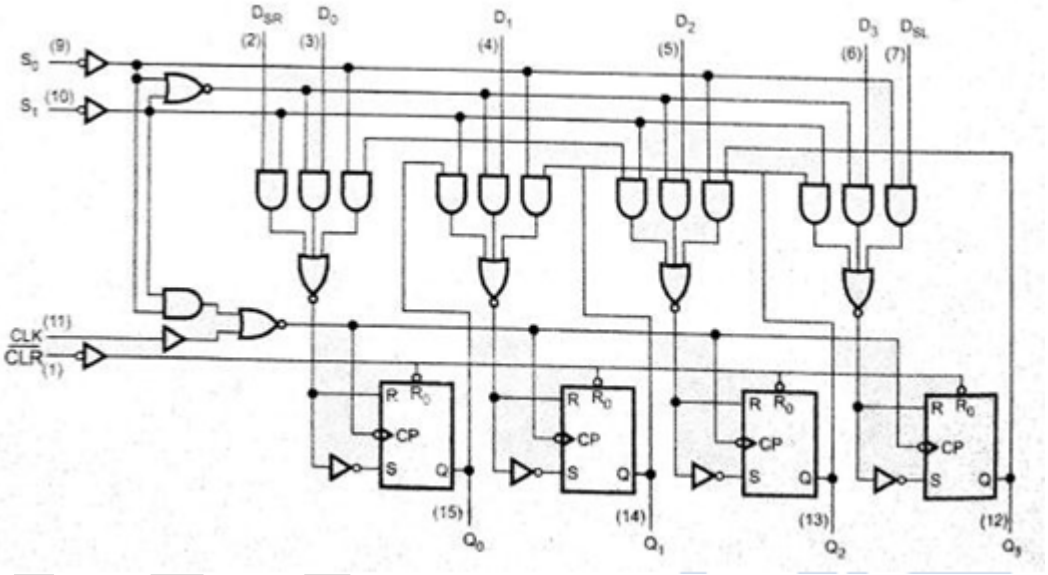
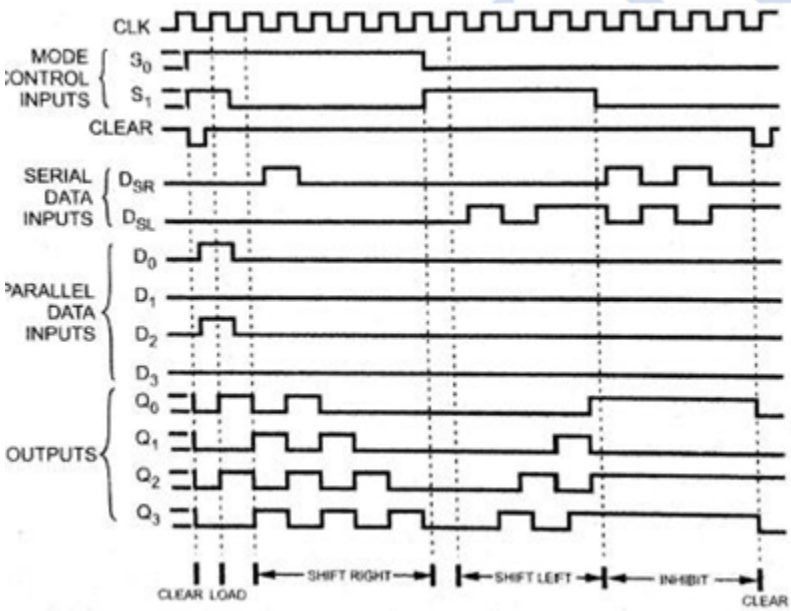


Figure 6.76: Logic diagram of IC 74x194

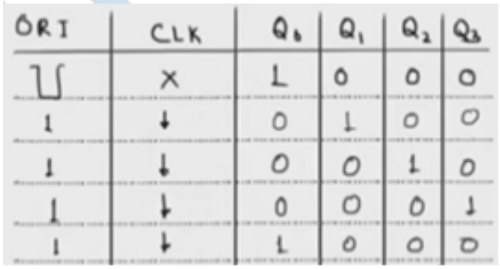


Operation mode	Inputs							Outputs			
	CLK	$\overline{CLR}$	S_1	S_0	D_{SR}	D_{SL}	D_n	Q_0	Q_1	Q_2	Q_3
Reset (clear)	X	0	X	X	X	X	X	0	0	0	0
Shift-Left	↑	1	1	0	X	0	X	Q_1	Q_2	Q_3	0
	↑	1	1	0	X	1	X	Q_1	Q_2	Q_3	1
Shift-Right	↑	1	0	1	0	X	X	0	Q_0	Q_1	Q_2
	↑	1	0	1	1	X	X	1	Q_0	Q_1	Q_2
Parallel Load	↑	1	1	1	X	X	D_n	D_0	D_1	D_2	D_3
Hold (Do nothing)	X	1	0	0	X	X	X	Q_0	Q_1	Q_2	Q_3

Table 6.36: Functional table of IC 74194

❖ **What is ring counter? Explain its operation and concept of self-correcting counters?**

Ring counter is counter named like that is because of the counting in a repetitive cycle like as ring. This operation is achieved by connecting a feedback connection from last stage flip-flop active high output to the input of first stage flip-flop. Initially all flip-flops are at rest or reset state and have the outputs Q_0, Q_1, Q_2 and Q_3 as '0'. For application of any clock cycle the same operation is obtained. So to get a valid count we have to make at least one flip-flop output as active high.



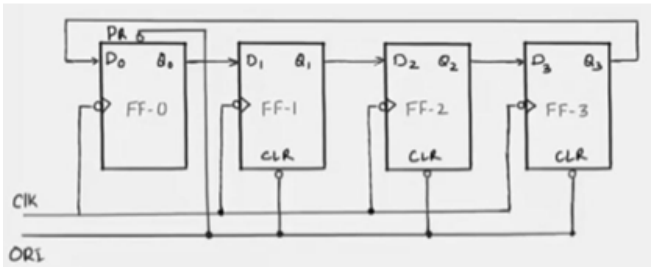


Figure 6.78: Ring counter circuit

Table 6.37: Functional table

In ring counter only the above mentioned counts are valid and remaining all possible counts with 4-bit data are valid. To make all the in valid counts into valid counts we have to apply left shift operation. such type of counters are called as self-correcting counters.

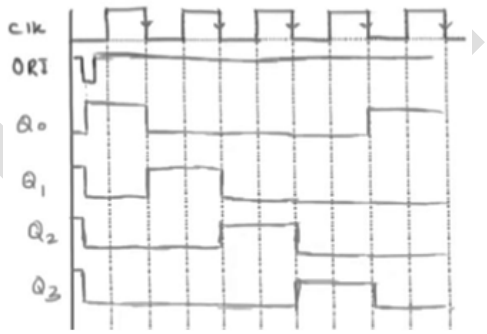


Figure 6.79: Timing waveforms of ring counter

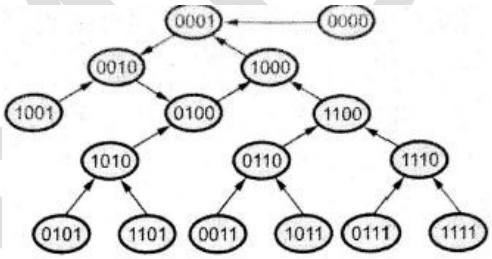
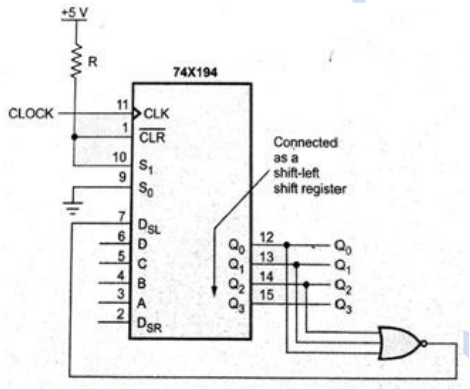


Figure 6.80: Self-correcting counters



Note: In ring counters the count vale = number of stages.

❖Explain the operation of Johnsons counter (Twisted ring counter)?

It has simple modification as compared to ring counter that it doesn't need any input to make count value as valid because the feedback is connected from complementary output

of last stage flip-flop. By this connection we will get eight valid counts, which are more counts compared to ring counter. In ring counter only four counts are valid counts. Since the feedback connection is applied from complementary output of a flip-flop, it seems to be a twist present in a ring. Hence by its shape this

Johnsons counter is also called as Twisted ring counter.

CLR	CLK	Q_0	Q_1	Q_2	Q_3
⊓	×	0	0	0	0
1	↓	1	0	0	0
1	↓	1	1	0	0
1	↓	1	1	1	0
1	↓	1	1	1	1
1	↓	0	1	1	1
1	↓	0	0	1	1
1	↓	0	0	0	1
1	↓	0	0	0	0

Table 6.39: Functional table

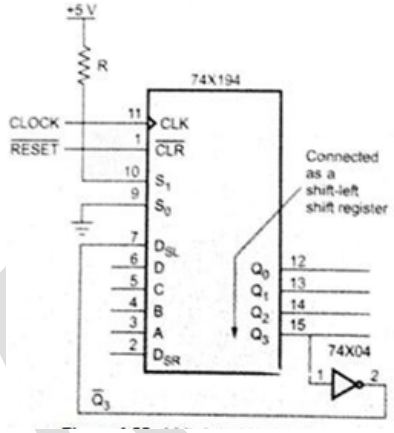
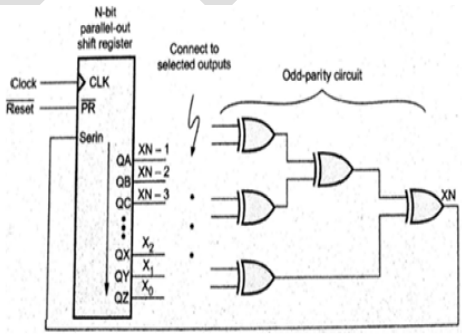
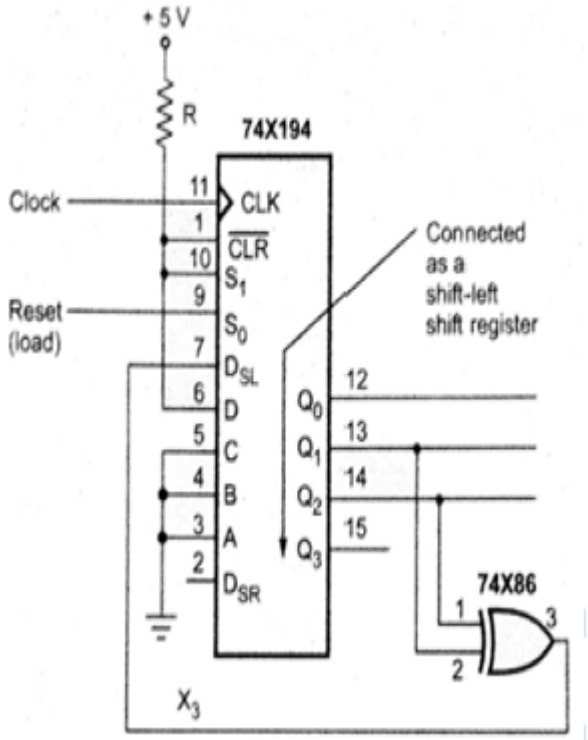


Figure 6.82(a): Logic diagram of a Johnsons counter;

❖ **What are the Linear Feedback Shift Registers (LFSR) counters?**



Structure of LFSR counter



3 Bit LFSR Counter

Q_2	Q_1	Q_0
0	0	1
0	1	0
1	0	1
0	1	1
1	1	1
1	1	0
1	0	0
0	0	1

3 bit LFSR Counter Operation

UNIT V
DESIGN EXAMPLES USING VHDL

Barrel Shifter

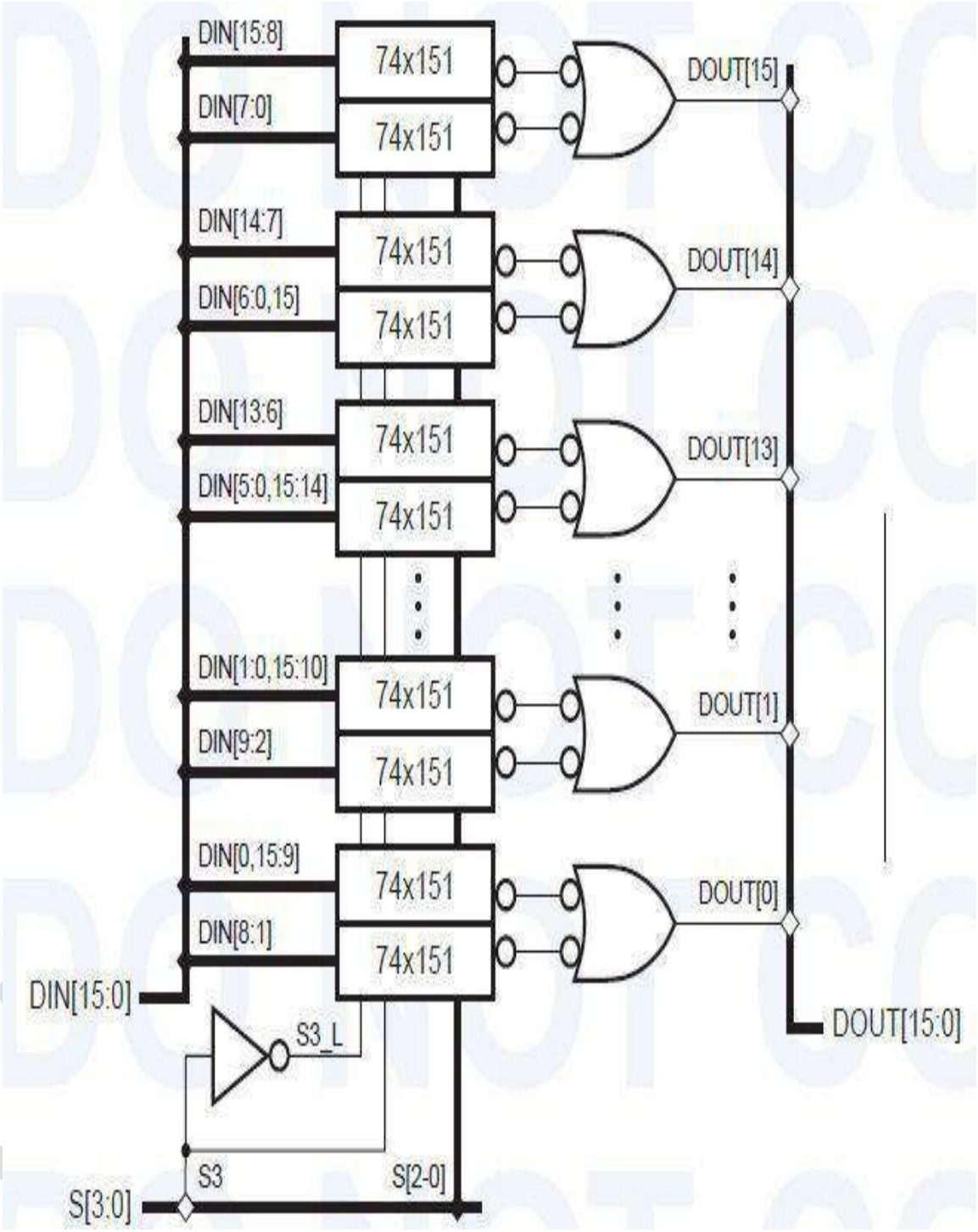
A *barrel shifter* is a combinational logic circuit with *n* data inputs, *n* data outputs, and a set of control inputs that specify how to shift the data between input and output. A barrel shifter that is part of a microprocessor CPU can typically specify the direction of shift (left or right), the type of shift (circular, arithmetic, or logical), and the amount of shift (typically 0 to *n*–1 bits, but sometimes 1 to *n* bits).

A simple 16-bit barrel shifter that does left circular shifts only, using a 4-bit control input S[3:0] to specify the amount of shift. For example, if the input word is ABCDEFGHGIHKLMNOP (where each letter represents one bit), and the control input is 0101 (5), then the output word is FGHGIHKLMNOPABCDE. From one point of view, this problem is deceptively simple.

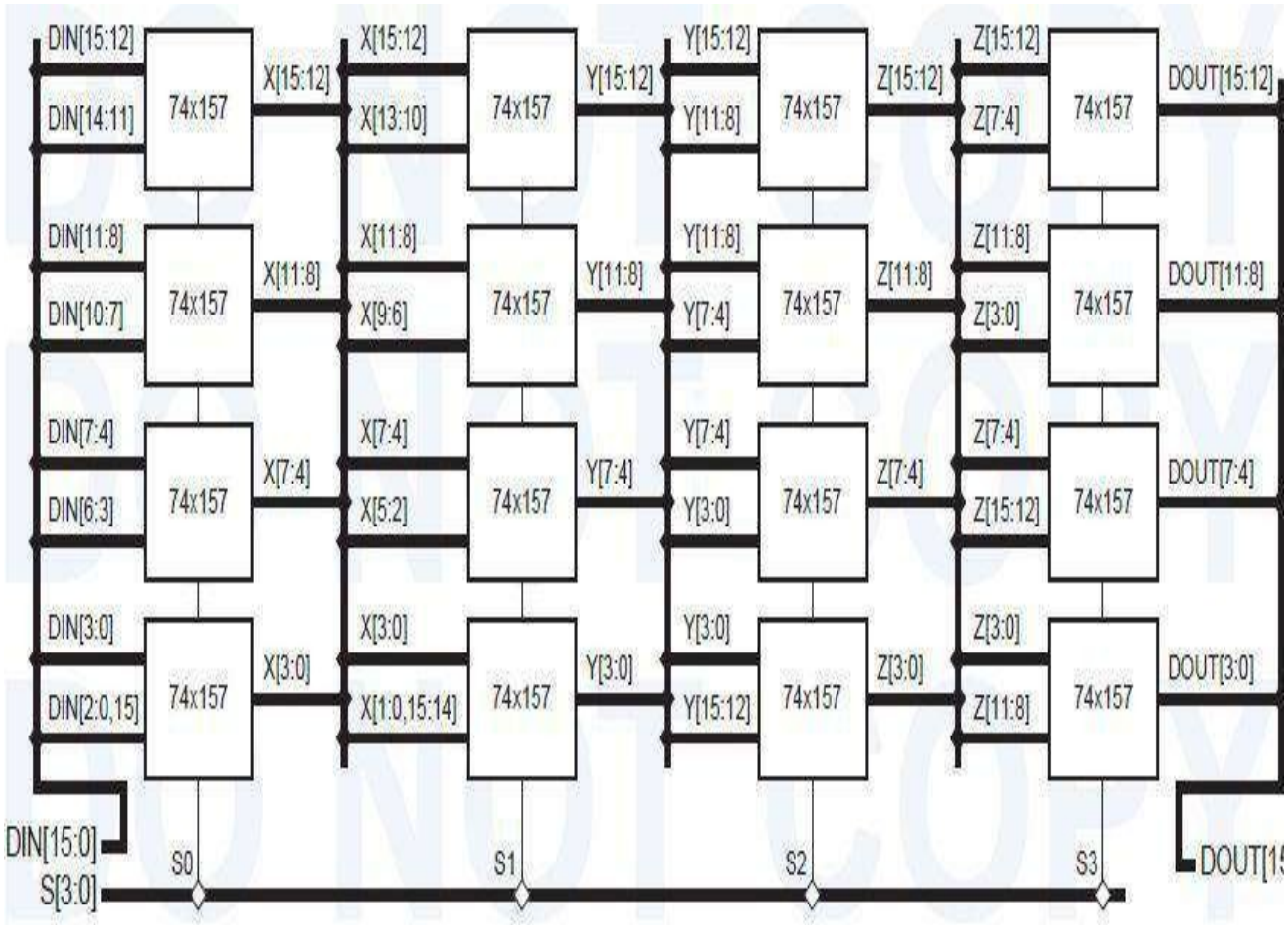
Each output bit can be obtained from a 16-input multiplexer controlled by the shift-control inputs, which each multiplexer data input connected to the appropriate On the other hand, when you look at the details of the design, you'll see that there are trade-offs in the speed and size of the circuit. Let us first consider a design that uses off-the-shelf MSI multiplexers.

A 16-input, one-bit multiplexer can be built using two 74x151s, by applying S3 and its complement to the EN_L inputs and combining the Y_L data outputs with a NAND gate, as we showed in Figure 5-66 for a 32-input multiplexer. The loworder shift-control inputs, S2–S0. connect to the like-named select inputs of the '151s.

We complete the design by replicating this 16-input multiplexer 16 times and hooking up the data inputs appropriately, as shown in Figure 6-1. The top '151 of each pair is enabled by S3_L, and the bottom one by S3; the remaining select bits are connected to all 32 '151s. Data inputs D0–D7 of each '151 are connected to the DIN inputs in the listed order from left to right.



The ’157-based approach requires only half as many MSI packages and has far less loading on the control and data inputs. On the other hand, it has the longest data-path delay, since each data bit must pass through four 74x157s. Halfway between the two approaches, we can use eight 74x153 4-input, 2-bit multiplexers two build a 4-input, 16-bit multiplexer. Cascading two sets of these, we can use S[3:2] to shift selectively by 0, 4, 8, or 12 bits, and S[1:0] to shift by 0–3 bits.



Simple Floating-Point Encoder

The previous example used multiple copies of a single building block, a multiplexer, and it was pretty obvious from the problem statement that a multiplexer was the appropriate building block. The next example shows that you sometimes have to look a little harder to see the solution in terms of known building blocks. Now let's look at a design problem whose MSI solution is not quite so obvious, a –fixed-point to floating-point encoder. An unsigned binary integer *B* in the range 0 B < 211 can be represented by 11 bits in –fixed-point format, *B b*10*b*9 *b*1*b*0. We can represent numbers in the same range with less precision using only 7 bits in a floating-point notation, *F M* 2*E*, where *M* is a 4-bit mantissa *m*3*m*2*m*1*m*0 and *E* is a 3-bit exponent *e*2*e*1*e*0. The smallest integer in this format is 0 20 and the largest is (24 1) 27.

Given an 11-bit fixed-point integer B, we can convert it to our 7-bit floating- point notation by –picking off four high-order bits beginning with the most significant 1, for example, The last term in each equation is a truncation error that results from the loss of precision in the conversion. Corresponding to this conversion operation, we can write the specification for a fixed-point to floating-point encoder circuit:

• A combinational circuit is to convert an 11-bit unsigned binary integer B into a 7-bit floating-point number *M,E*, where *M* and *E* have 4 and 3 bits, respectively. The numbers have the relationship $B = M\,2E\,T$, where *T* is the truncation error, $0\ T < 2E$.

Starting with a problem statement like the one above, it takes some creativity to come up with an efficient circuit design—the specification gives no clue. However, we can get some ideas by looking at how we converted numbers by hand earlier. We basically scanned each input number from left to right to find the first position containing a 1, stopping at the *b*3 position if no 1 was found. We picked off four bits starting at that position to use as the mantissa, and the starting position number determined the exponent. These operations are beginning to sound like MSI building blocks.

–Scanning for the first ‖ is what a generic priority encoder does. The output of the priority encoder is a number that tells us the position of the first 1. The position number determines the exponent; first-1 positions of *b*10 *b*3 imply exponents of 7–0, and positions of *b*2 *b*0 or no-1-found imply an exponent of 0. Therefore, we can scan for the first 1 with an 8-input priority encoder with inputs

11010110100101	7 0110100
00100101111001	5 01111
00000111110111	2 10
00000001011011	0 0
00000000010010	0 0

I7 (highest priority) through I0 connected to *b*10 *b*3. We can use the priority encoder's A2 – A0 outputs directly as the exponent, as long as the no-1-found case produces A2 – A0 000.
–Picking off four bits‖ sounds like a –selecting‖ or multiplexing operation.

The 3-bit exponent determines which four bits of *B* we pick off, so we can use the exponent bits to control an 8-input, 4-bit multiplexer that selects the appropriate four bits of *B* to form *M*.

• Since the available MSI priority encoder, the 74x148, has active-low inputs, the input number *B* is assumed to be available on an active-low bus B_L[10:0]. If only an active-high version of *B* is available, then eight inverters can be used to obtain the active-low version.
• If you think about the conversion operation a while, you'll realize that the most significant bit of the mantissa, *m*3, is always 1, except in the no-1-found case. The '148 has a GS_L output that indicates this case, allowing us to eliminate the multiplexer for *m*3.
• The '148 has active-low outputs, so the exponent bits (E0_L–E2_L) are produced in active-low form. Naturally, three inverters could be used to produce an active-high version.
• Since everything else is active-low, active-low mantissa bits are used too. Active-high bits are also readily available on the '148 EO_L and the '151 Y_L outputs.

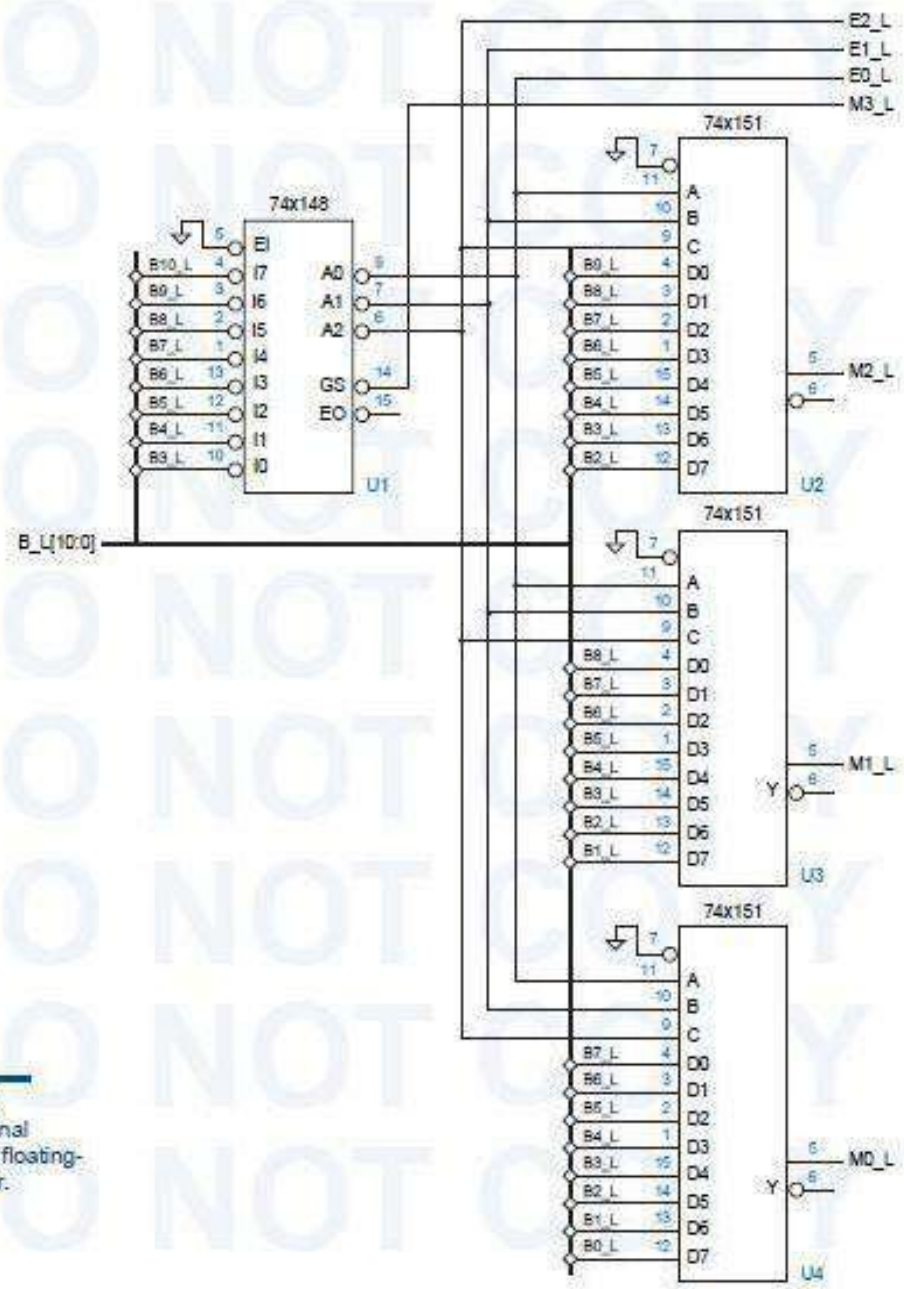


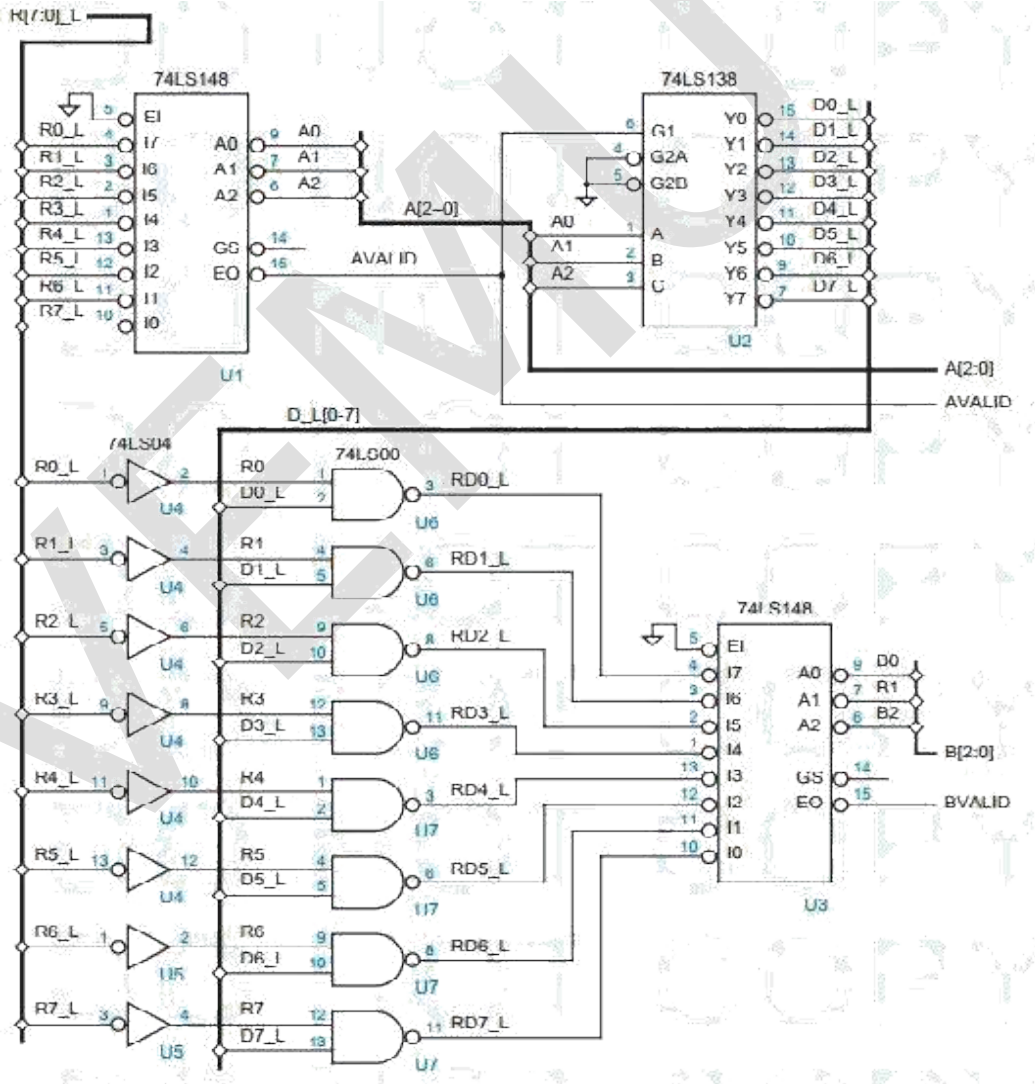
Figure 6-3
A combinational fixed-point to floating-point encoder.

Dual-Priority Encoder

Quite often MSI building blocks need a little help from their friends—ordinarygates— to get the job done. In this example, we'd like to build a priority encoder that identifies not only the highest but also the second-highest priority asserted signal among a set of eight request inputs. We'll assume for this example that the request inputs are active low and are named R0_L–R7_L, where R0_L has the highest priority. We'll use A2–A0 and AVALID to identify the highest-priority request, where AVALID is asserted only if at least one request

input is asserted. We'll use B2–B0 and BVALID to identify the second-highest-priority request, where BVALID is asserted only if at least two request inputs are asserted.

Finding the highest-priority request is easy enough, we can just use a 74x148. To find the second highest-priority request, we can use another '148, but only if we first –knock out‖ the highest-priority request before applying the request inputs. This can be done using a decoder to select a signal to knock out, based on A2–A0 and AVALID from the first '148. These ideas are combined in the solution shown in Figure . A 74x138 decoder asserts at most one of its eight outputs, corresponding to the highest-priority request input. The outputs are fed to a rank of NAND gates to –turn off‖ the highest-priority request. A trick is used in this solution is to get active-high outputs from the '148s. We can rename the address outputs A2_L–A0_L to be active high if we also change the name of the request input that is associated with each output combination. In particular, we complement the bits of the request number. In the redrawn symbol, request input I0 has the highest priority.



Cascading Comparators

Since the 74x85 uses a serial cascading scheme, it can be used to build arbitrarily large comparators. The 74x682 8-bit comparator, on the other hand, doesn't have any cascading inputs and outputs at all. Thus, at first glance, you might think that it can't be used to build larger comparators. But that's not true. If you think about the nature of a large comparison, it is clear that two wide inputs, say 32 bits (four bytes) each, are equal only if their corresponding bytes are equal. If we're trying to do a greater-than or less-than comparison, then the corresponding most-significant that are not equal determine the result of the comparison.

Using these ideas, Figure uses three 74x682 8-bit comparators to do equality and greater- than comparison on two 24-bit operands. The 24-bit results are derived from the individual 8-bit results using combinational logic for the following equations:

PEQQ EQ2 EQ1 EQ0
PGTQ GT2 EQ2 GT1 EQ2 EQ1 GT0

This –parallel‖ expansion approach is actually faster than the 74x85's serial cascading scheme, because it does not suffer the delay of propagating the cascading signals through a cascade of comparators. The parallel approach can be used to build very wide comparators using two-level AND-OR logic to combine the 8-bit results, limited only by the fan-in constraints of the AND-OR logic. Arbitrary large comparators can be made if you use additional levels of logic to do the combining.

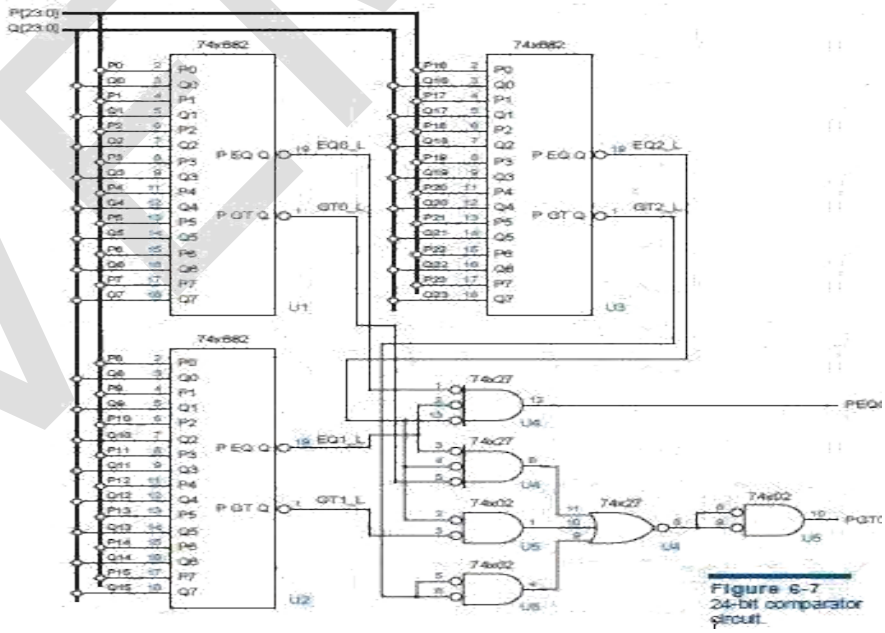


Figure 6-7
24-bit comparator circuit