A logic family is a collection of different IC chips that have similar input, output and internal circuit characteristics, but that perform different logic functions.

Chips from the same family can be interconnected to perform any desired logic function. On the other hand, chips from differing families may not be compatible; they may use different power-supply voltages or may use different input & output conditions to represent logic values.

The most successful bipolar logic family is transistor-transistor logic (TTL). First introduced in the 1960s, TTL now is actually a family of logic families that are compatible with each other but differ in speed, power consumption, and cost.

Metal-Oxide Semiconductor field-effect transistor (MOSFET), or simply MOS transistor, were difficult to fabricate in the early days, and it wasn't until the 1960s that a wave of developments made MOS based logic & memory circuits practical. Even then, MOS circuits lagged bipolar circuits considerably in speed and were attractive only in selected applications because of their lower power consumption and higher levels of integration.

Beginning in the mid-1970s, advances in the design of MOS circuits, in particular Complementary MOS (CMOS) circuits, vastly increased their performance & popularity. By far the majority of new large-scale integrated circuits, such as microprocessors & memories, use CMOS. CMOS circuits now account for the vast majority of the world wide IC market.

CMOS logic is both the most capable & the easiest to understand commercial digital logic technology.

As a consequence of the industry's transition from TTL to CMOS over long period of time, many CMOS families were designed to be somewhat compatible with TTL.

CMOS LOGIC

The basic building blocks in CMOS logic circuits are MOS transistors. In any logic circuit, there is a range of voltages that is interpreted as a logic 0, and another, nonoverlapping range that is interpreted as a logic 1.

A typical CMOS logic circuit operates from a 5-volt power supply. Such a circuit may interpret any voltage in the range 0-1.5 V as a logic 0, and in the range 3.5 - 5.0V as a logic 1. Thus, the definitions of LOW & HIGH for 5-volt CMOS logic are as shown in figure 1.

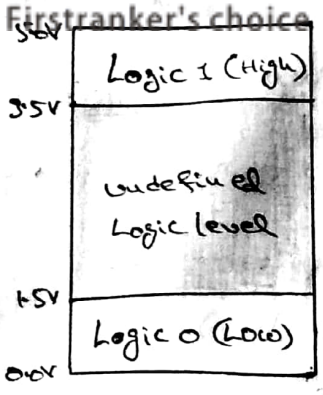


fig 1: Logic levels for typical CMOS logic circuits

Voltages in the intermediate range (1.5 - 3.5V) are not expected to occur except during signal transitions, and yield undefined logic values (i.e., a circuit may interpret them as either 0 or 1).

MOS Transistors:

A MOS transistor can be modeled as a 3-terminal device that acts like a voltage controlled resistance. As suggested by figure 2, an i/p voltage applied to one terminal controls the resistance between the remaining two terminals. In digital logic applications, a MOS transistor is operated so its resistance is always either very high (and the transistor is OFF) or very low (and the transistor is ON).

There are two types of MOS transistors, n-channel & p-channel; the names refer to the type of semiconductor material used for the resistance-controlled terminals.

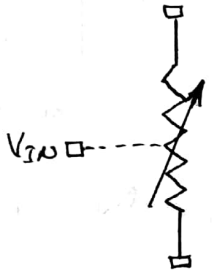
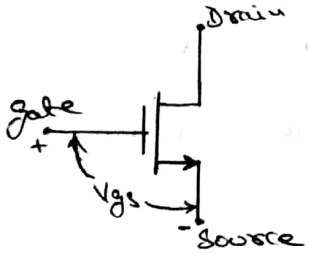


fig 2 The MOS transistor as a voltage controlled resistance.

The circuit symbol for an n-channel MOS (NMOS) transistor is shown in figure 3:



Voltag-controlled resistance
increase V_{gs} => decrease R_{ds}
i.e $V_{gs} \propto \frac{1}{R_{ds}}$

Note: normally $V_{gs} \geq 0$

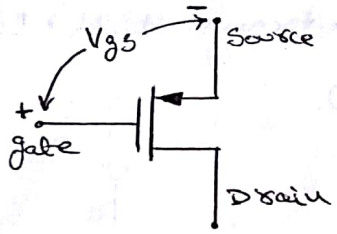
fig 3 Circuit symbol for an n-channel MOS (NMOS) transistor

The terminals are called gate, source, & drain.

As you might guess from the orientation of the circuit symbol, the drain is normally at a higher voltage than the source.

The voltage from gate to source (V_{gs}) in an NMOS transistor is normally zero or positive. If $V_{gs} = 0$, then the resistance from drain to source (R_{ds}) is very high, at least a megaohm ($10^6 \Omega$) or more. As we increase V_{gs} (i.e., increase the voltage on the gate), R_{ds} decreases to a very low value, 100 or so less in some devices.

The circuit symbol for a p-channel MOS (PMOS) transistor is shown in figure 4.9.



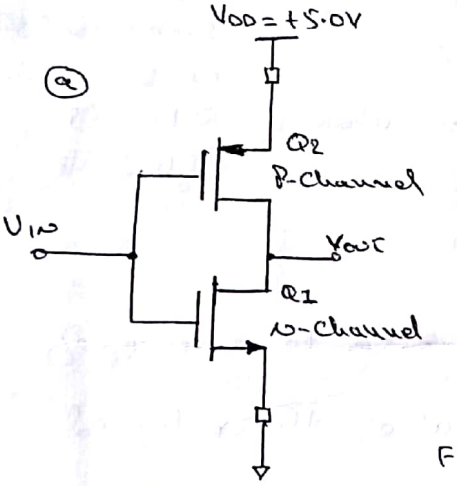
Voltage - Controlled resistance
decrease V_{gs} ⇒ decrease R_{ds}
i.e; $V_{gs} \propto R_{ds}$
Note: normally, $V_{gs} \leq 0$

Fig 4 circuit symbol for a p-Channel MOS (PMOS) transistor

Operation is analogous to that of an NMOS transistor, except that the source is normally at a higher voltage than the drain, and V_{gs} is normally zero or negative. If V_{gs} is zero, then the resistance from source to drain (R_{ds}) is very high. As we algebraically decrease V_{gs} (i.e., decrease the voltage on the gate), R_{ds} decreases to a very low value.

NMOS and PMOS transistors are used together in a complementary way to form CMOS logic.

The simplest CMOS circuit, a logic inverter, requires only one of each type of transistor, connected as shown in figure 5(a). The power-supply voltage, V_{DD}, typically may be in the range 2-6 V and is most often set at 5.0V for compatibility with TTL circuits.



(b)

V_{IN}	Q_1	Q_2	V_{OUT}
0.0(L)	OFF	ON	5.0(H)
5.0(H)	ON	OFF	0.0(L)

(c)

Fig:5 CMOS inverter (a) circuit diagram (b) functional behaviour (c) Logic Symbol

Case-I

V_{IN} is 0.0V.

In this case, the bottom, n-Channel transistor Q1 is OFF, since its V_{gs} is 0. but the top, P-channel transistor Q_2 is ON, since its V_{gs} is a large -ve (-5.0V). Therefore, Q_2 presents only a small resistance between the power-supply terminal (V_{DD}, +5.0V) and the o/p terminal (V_{OUT}), and the o/p voltage is 5.0V.

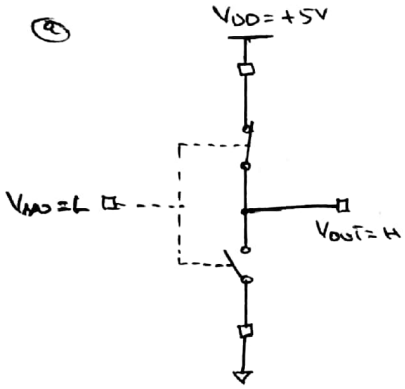
Case-II

V_{IN} is 5.0V.

Here, Q_1 is ON, since its V_{gs} is a large +ve value (+5.0V) but Q_2 is OFF, since its V_{gs} is 0. Thus, Q_1 presents a small resistance between the o/p terminal & ground, so the o/p voltage is 0V.

(6)

Another way to visualize CMOS operation uses switches. As shown in figure 6, the n-channel (bottom) transistor is modeled by a normally-open switch, and the p-channel (top) transistor by a normally-closed switch. Applying a HIGH voltage changes each switch to the opposite of its normal state, as shown in (b).



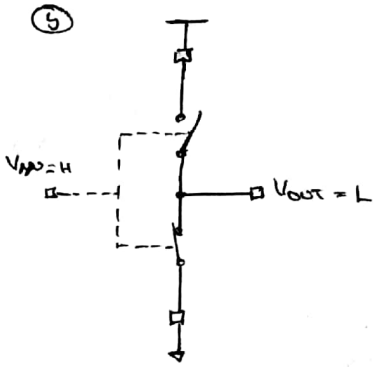


Fig 6 Switch model for CMOS inverter
(a) Low i/p
(b) High i/p

The switch model gives rise to a way of drawing CMOS circuits that makes their logical behaviour more readily apparent. As shown in fig 7, different symbols are used for the p- & n-channel transistors to reflect their logical behaviour. The n-channel transistor (Q_1) is switched 'on', & current flows between source & drain, when a HIGH voltage is applied to its gate; this seems natural enough. The p-channel transistor (Q_2) has the opposite behavior. It is "on" when a LOW voltage is applied; the inversion bubble on its gate indicates this inverting behaviour.

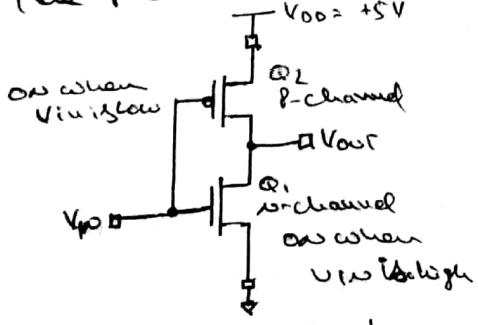
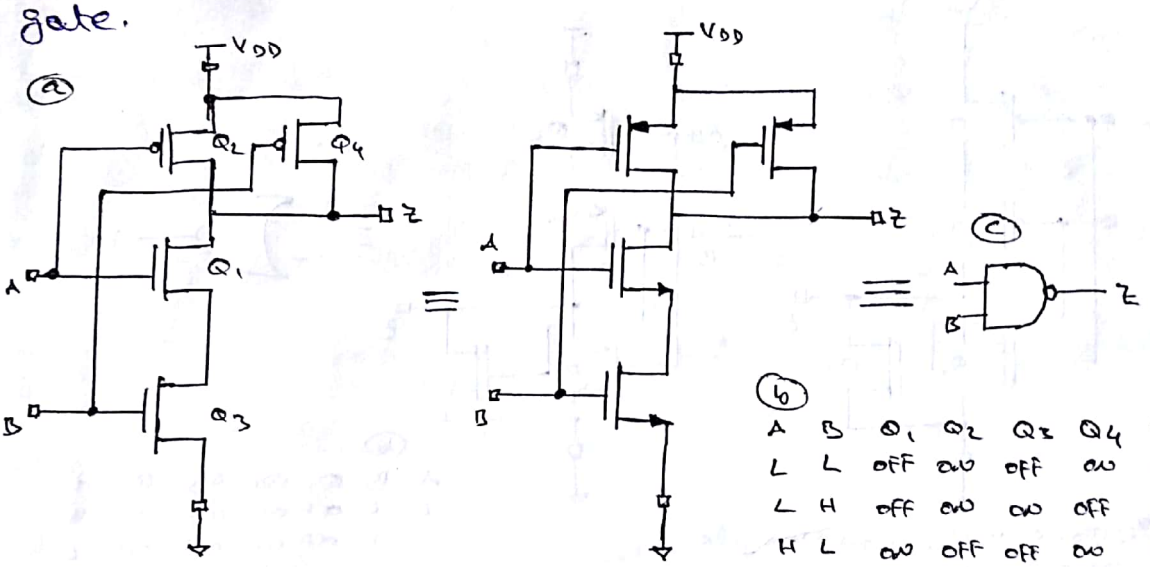


Fig 7: CMOS inverter logical operation

(6)

CMOS NAND gate

NAND gate can be constructed using CMOS. A k-i/p gate uses k p-channel & k-n-channel transistors. Figure 5.3 shows a 2-i/p CMOS NAND gate.

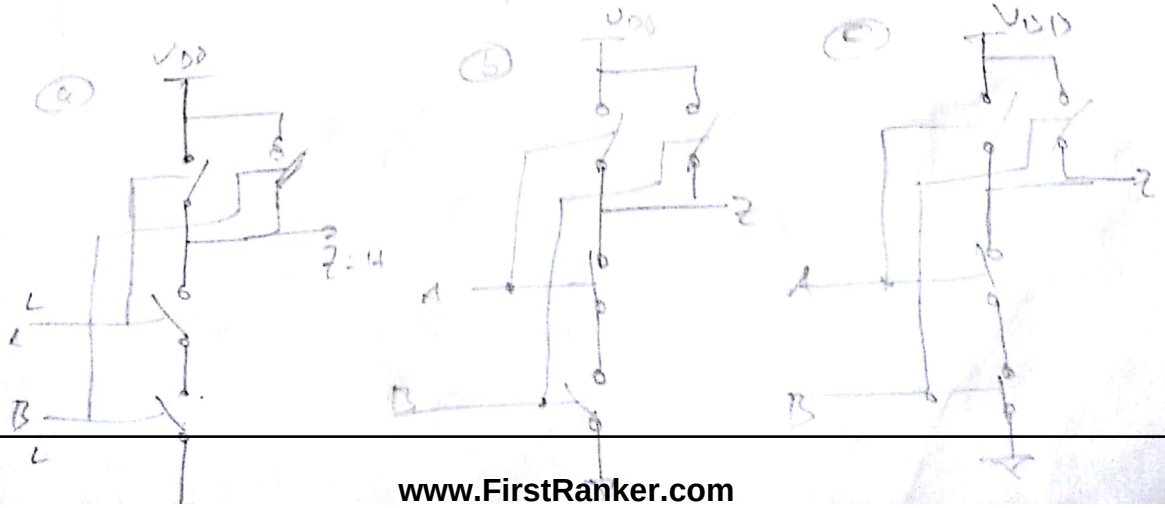


b

A	B	Q_1	Q_2	Q_3	Q_4	Z
L	L	OFF	ON	OFF	ON	H
L	H	OFF	ON	ON	OFF	H
H	L	ON	OFF	OFF	ON	H
H	L	ON	OFF	ON	OFF	L

Fig 5: CMOS 2-input NAND gate (a) circuit (b) functiontable (c) logic symbol

If either i/p is LOW, the o/p Z has a low-impedance connection to V_{DD} through the corresponding ON P-channel transistor, and the path to ground is blocked by the corresponding "OFF" n-channel transistor. If both i/ps are HIGH, the path to V_{DD} is blocked, and Z has a low-impedance connection to ground. Fig 5.4 shows the switch model for the NAND gate's operation.



CMOS NOR gate

NOR gate can be constructed using CMOS.

A K-input gate uses K n-channel & K p-channel transistors. Figure 10 shows a CMOS NOR gate.

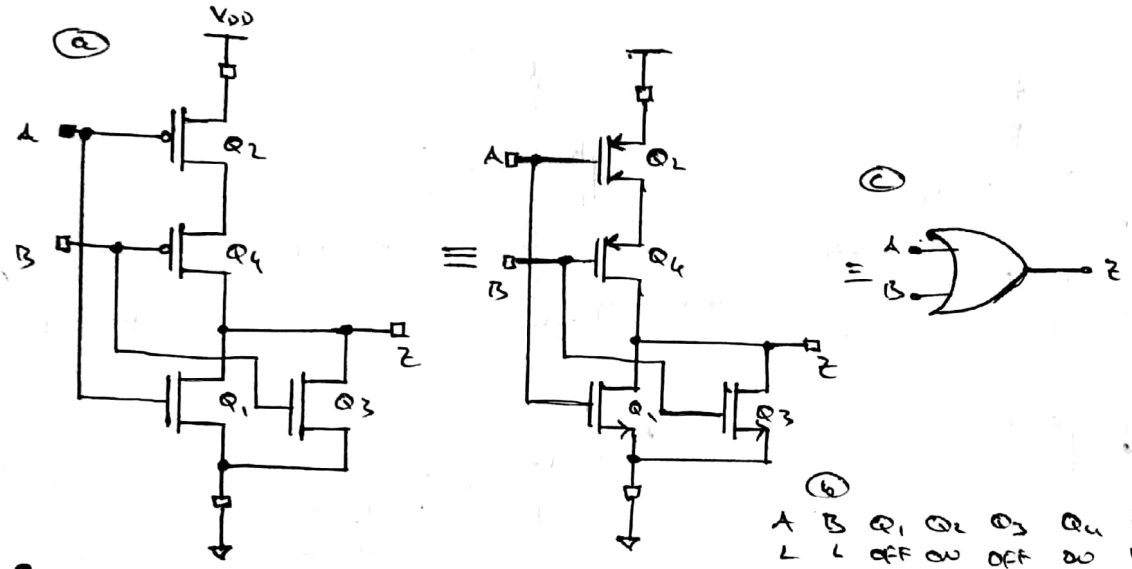


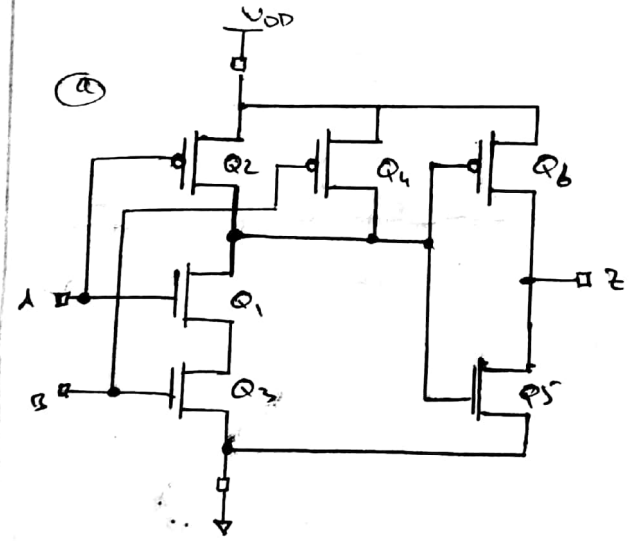
Fig ⑩: CMOS 2-input NOR gate
- ⓐ circuit diagram
- ⓑ function table
- ⓒ logic symbol

If both i/ps are Low, then the o/p Z has a Low impedance connection to V_{DD} through the "ON" p-channel transistors, and the path to ground is blocked by the "OFF" n-channel transistors. If either input is High, the path to V_{DD} is blocked, and Z has a Low-impedance connections to ground. Figure shows the switch model for the NOR gate's operation.

Non-Inverting gates

In CMOS, and in most other logic families the simplest gates are inverters, and the next simplest are NAND gates & NOR gates.

CMOS non inverting buffers and AND and OR gates are obtained by connecting an inverter to the o/p of the corresponding inverting gate. Thus, fig shows a non inverting buffer and figure shows an AND gate. Combining NOR with an inverter yields an OR gate as shown in fig



(a)

(b)

A	B	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Z
L	L	OFF	ON	OFF	ON	ON	OFF	L
L	H	OFF	ON	ON	OFF	ON	OFF	L
H	L	ON	OFF	OFF	ON	ON	OFF	L
H	H	ON	OFF	ON	OFF	OFF	ON	H

(c)

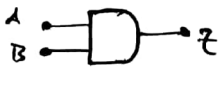


Fig : CMOS 2-input AND gate (a) circuit diagram (b) Function table (c) logic symbol

Electrical Behaviour of CMOS Circuits

CMOS Steady State Electrical Behaviour

The steady-state behaviour is the behaviour of the CKT when i/p & o/p's are not changing.

Logic levels & Noise margin

The CMOS device manufacturers specify the four voltage parameters in the data sheet. They are:

$V_{IH(min)}$ - High level input voltage.

It is a minimum voltage level required for a logical 1 at an i/p. Any voltage below this level will not be accepted as a High by the logic CKT.

$V_{IL(max)}$ - low level input voltage

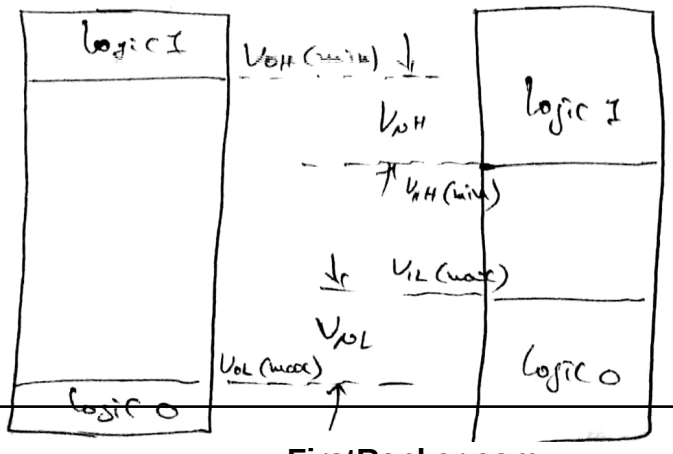
It is a maximum voltage level required for a logic 0 at an i/p. Any voltage above this level will not be accepted as a low by the logic CKT.

$V_{OH(min)}$ - High-level o/p voltage

It is a minimum voltage level at a logic CKT o/p for the logic 1 state under defined load conditions.

$V_{OL(max)}$ - low-level o/p voltage

It is a maximum voltage level at a logic CKT o/p for the logical 0 state under defined load conditions.



$V_{NL} = V_{IL(max)} - V_{OL(max)}$

Voltage levels & noise margins

The power-supply voltage V_{CC} & ground are often called the power supply rails.

CMOS levels are typically a fraction of the power supply rails.

$V_{OH(min)}$: $V_{CC} - 0.1V$

$V_{IH(min)}$: 70% of V_{CC}

$V_{IL(max)}$: 30% of V_{CC}

$V_{OL(max)}$: Ground + 0.1V

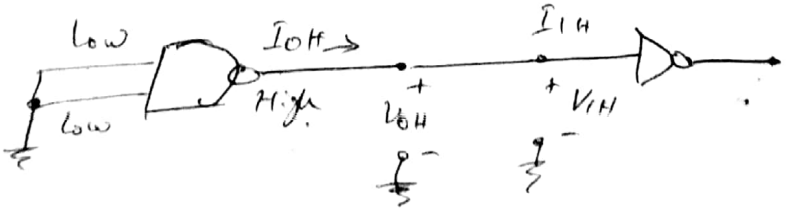
Like voltage levels, device manufacturers also specify current parameters. There are,

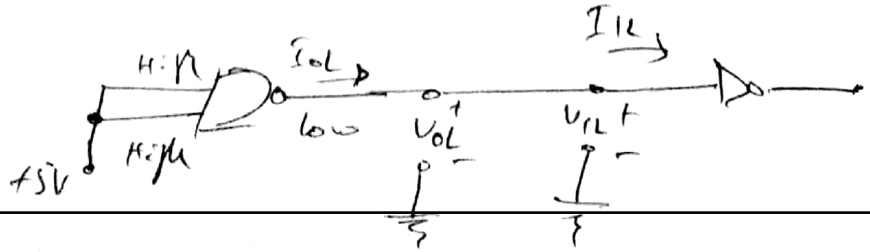
I_{IH} - High level input current : It is a current that flows into an i/p when a specified high-level voltage is applied to that i/p.

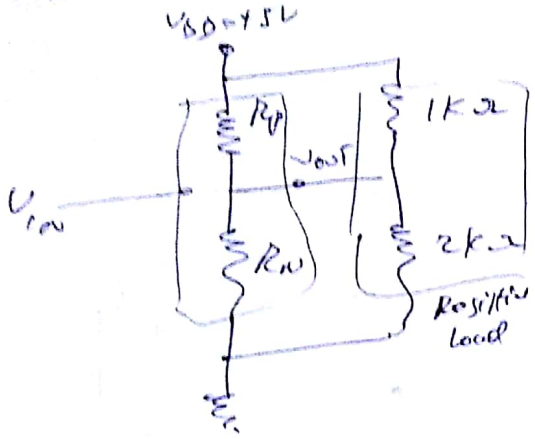
I_{IL} - low level input current : It is a current that flows into an i/p when specified low-level voltage is applied to that i/p

I_{OH} - High level o/p current : It is a current that flows from an o/p in the logical 1 state under specified load conditions.

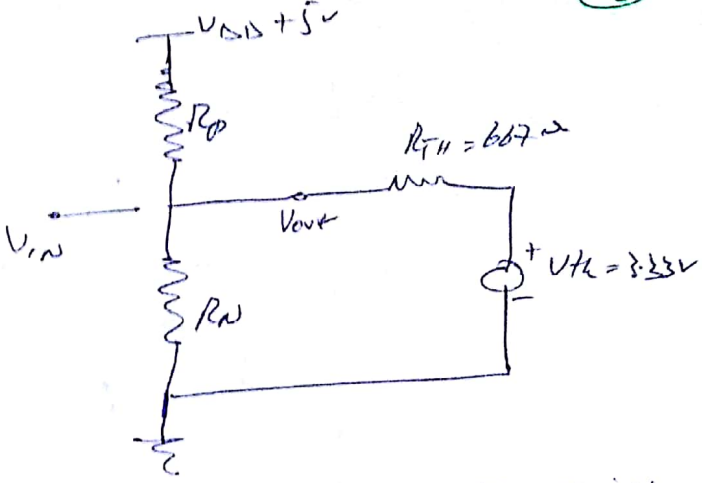
I_{OL} - Low level o/p current : It is the current that flows from an o/p in the logical 0 state under specified load conditions.







Resistive load of CMOS Inverter

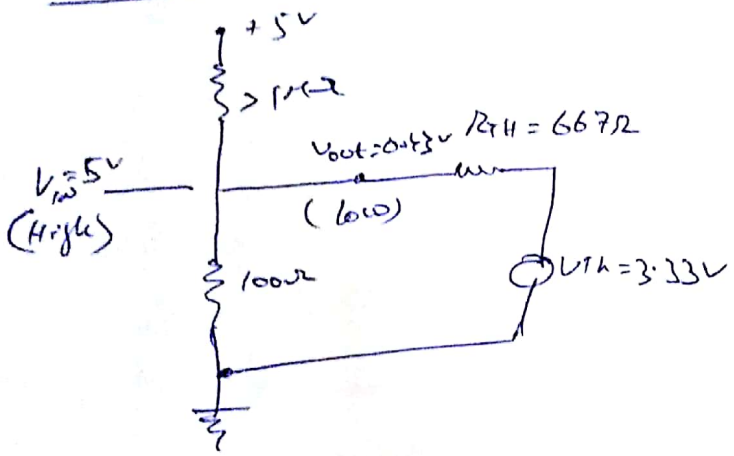


Resistive load modeled by a Thevenin equivalent CKT

$$V_{Th} = \frac{2K}{2K+1K} \times 5V = 3.33V$$

$$R_{Th} = 1K \| 2K = 667\Omega$$

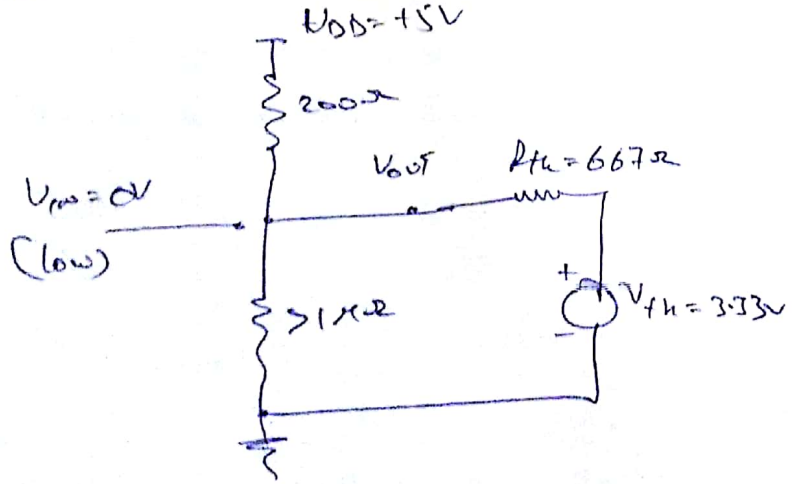
Condition 1: Input High



$$V_{out} = \frac{V_{TH} \times 100}{(100+667)} = \frac{3.33V \times 100}{100+667}$$

$$= 0.43V$$

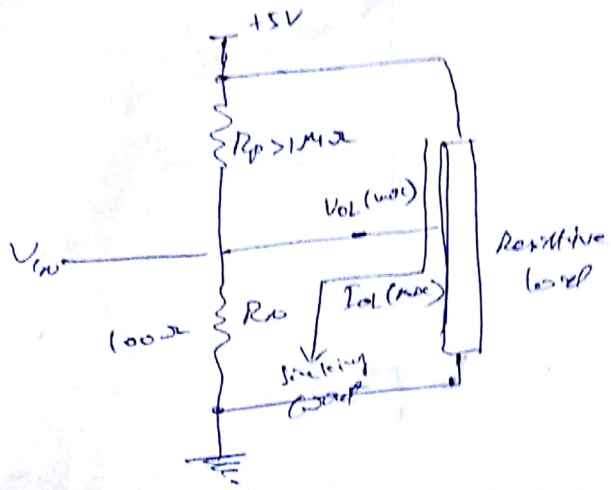
Condition 2: Input Low



$$V_{out} = \frac{(V_{DD}-V_{Th})R_{TH}}{R_{TH}+200} + V_{Th}$$

$$= \frac{(5-3.33) \times 667}{667+200} + 3.33$$

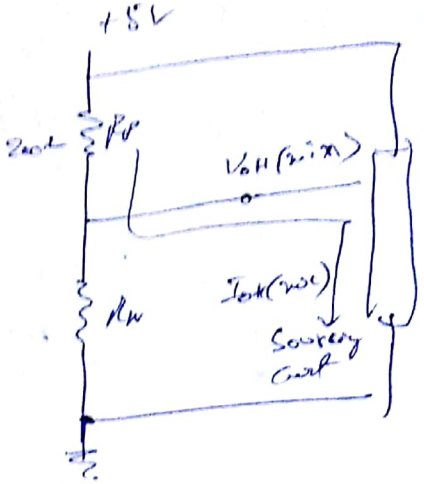
$$= 4.61V$$



$$\text{Sinking current} = \frac{V_{OL(max)}}{R_N}$$

$$= \frac{0.43V}{100}$$

$$= 4.3mA$$

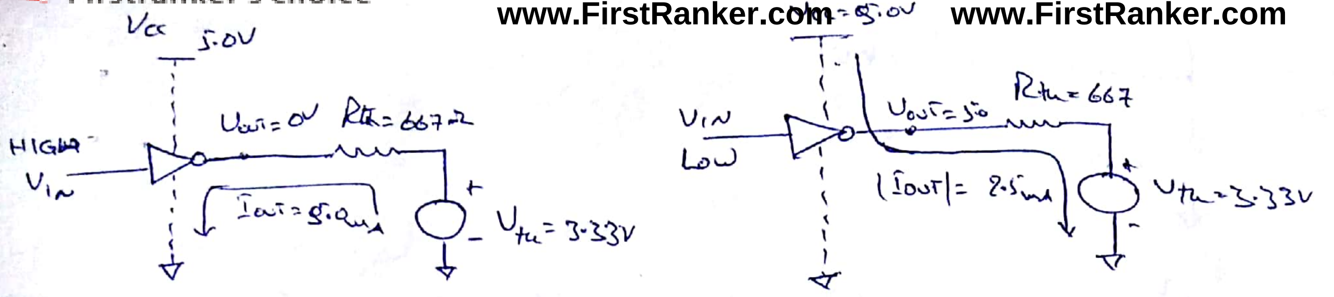


$$\text{Sourcing current} = \frac{V_{DD} - V_{OH(min)}}{R_p}$$

$$= \frac{5 - 4.61}{200}$$

$$= 1.95mA$$

Simulating sink and source current



(a) output low (b) output HIGH

Assume ON resistance of transistor equal to zero, i.e no voltage drop across it in it, or very good worst case estimates of o/p current.

For low output

$$I_{out} = \frac{V_{th}}{R_{th}} = \frac{3.33}{667} = 5\,mA$$

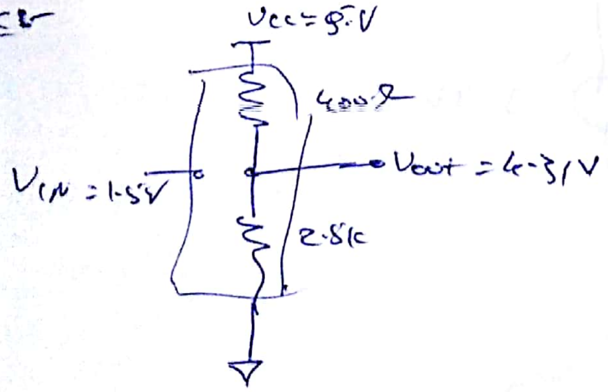
For High output

$$I_{out} = \frac{5 - 3.33}{667} = 2.5\,mA$$

Circuit Behaviour with non ideal i/p's

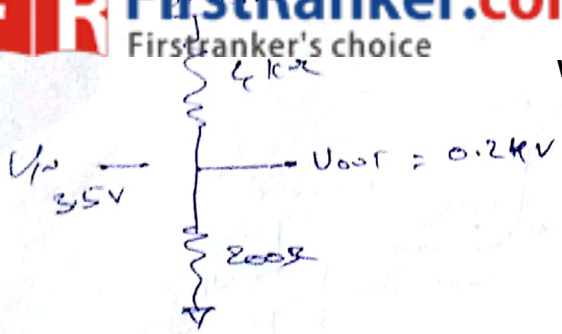
If the i/p voltage is not close to the power supply rail, then the 'ON' transistor may not be fully 'ON' & its resistance may increase. Like wise, the 'OFF' transistor may not be fully 'OFF' & its resistance may be quite a bit less than one MΩ. These two effects combine to move the output voltage away from the power supply rail.

Ex:-



$$V_{out} = \frac{(2.5k)5}{2.5k + 400} = 4.31\,V$$

$V_{out} = \frac{200\,(5V)}{200+4k} = 0.24V$



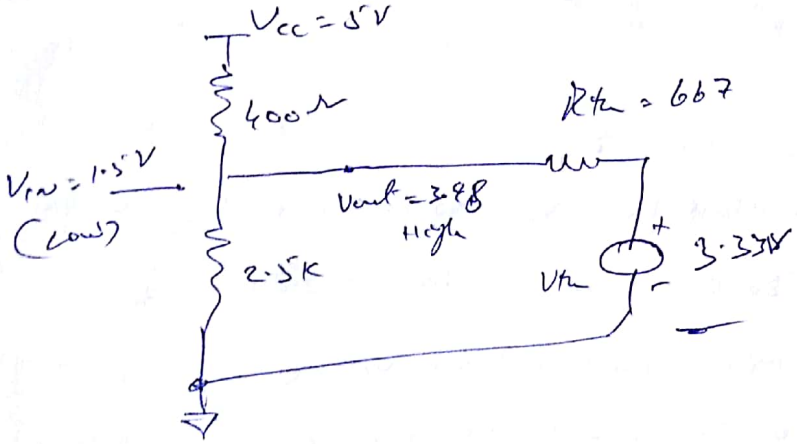
Whats worse is that the o/p structure is now consuming non trivial amount of power. The current flow with the 1.5V i/p is

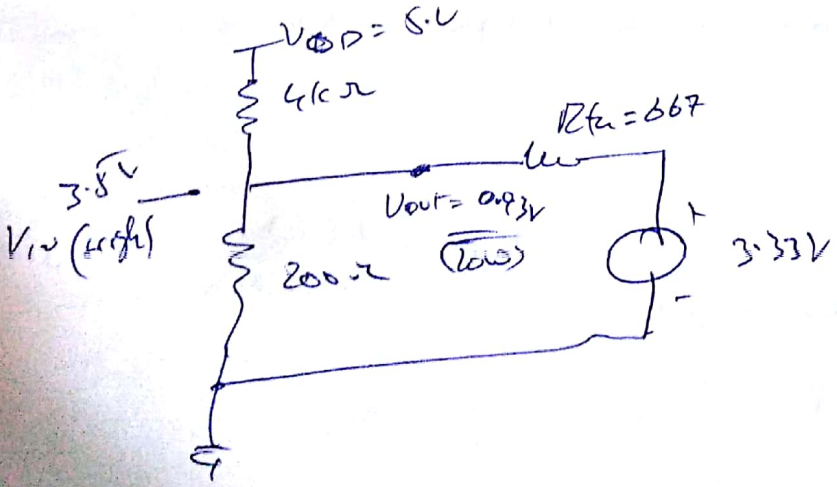
$$I_{switched} = \frac{5}{400\Omega + 2.5k} = 1.72\,mA$$

& power consumption is

$$P_{switched} = 5.0V \times I_{switched} = 5 \times 1.72\,mA = 8.62\,mW$$

The o/p voltage of CMOS inverter deteriorates further with a resistive load.





... resistances are not known, however they can be calculated by following eqns.

$$R_P(ON) = \frac{V_{DD} - V_{OH}(min)}{|I_{OH}(max)|}$$

$$R_N(ON) = \frac{V_{OL}(max)}{I_{OL}(max)}$$

Let us consider the parameters of CMOS HC-series listed in the table

Parameter	CMOS load	Parameter	TTL load
$I_{OL}(max)_C$	0.02 mA	$I_{OL}(max)_T$	4.0 mA
$V_{OL}(max)_C$	0.1 V	$V_{OL}(max)_T$	0.33 V
$I_{OH}(max)_C$	−0.02 mA	$I_{OH}(max)_T$	−4.0 mA
$V_{OH}(min)_C$	4.4 V	$V_{OH}(min)_T$	3.84 V

From the table

$$R_P(ON) = \frac{5 - 3.84}{|-4.0 \times 10^{-3}|} = 290\Omega \quad (175\Omega)$$

$$R_N(ON) = \frac{0.33}{4 \times 10^{-3}} = 82.5\Omega$$

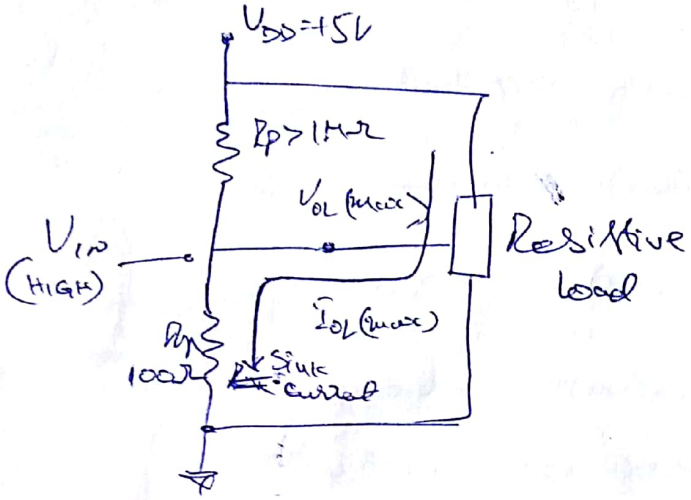
} For TTL load

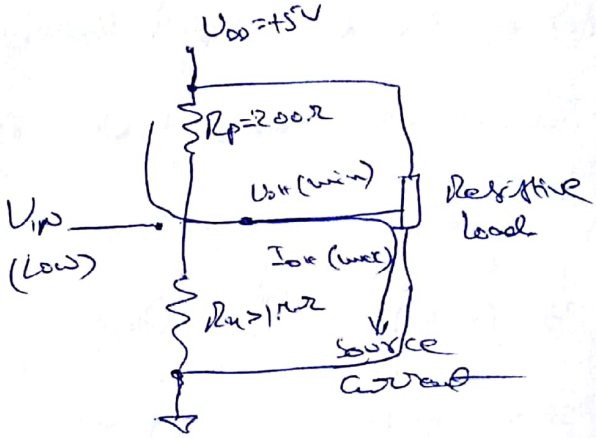
$$R_P(ON) = \frac{5 - 4.4}{0.02 \times 10^{-3}} = 0.03$$

$$R_N(ON) = \frac{0.1}{0.02 \times 10^{-3}} = 0.005$$

$I_{OL(max)}$: The maximum current that the o/p can sink in the Low state while still maintaining an o/p voltage smaller than $V_{OL(max)}$

$I_{OH(max)}$: The maximum current that the o/p can source/supply in the HIGH state while still maintaining an o/p voltage greater than $V_{OH(min)}$





Sink current: A device o/p is said to sink current when current flows from the power supply, through the load, and through the device o/p to ground.

$$\text{Sink current} = I_{Sink} = I_{OL(max)} = \frac{V_{OL(max)}}{R_N} \quad (\text{or}) \quad \frac{V_{out}}{R_N}$$

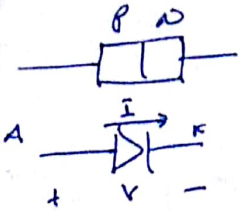
$$= \frac{0.43V}{100} = \underline{4.3\ mA}$$

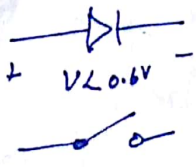
Source current: The o/p is said to source current when current flows from the power supply, out of the device o/p, & through the load to ground.

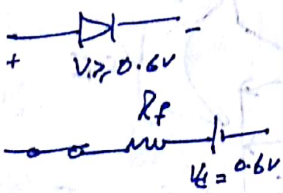
$$\text{Source current} = I_{source} = I_{OH(max)} = \frac{V_{out}}{R_p} = \frac{V_{OH(min)}}{R_p}$$

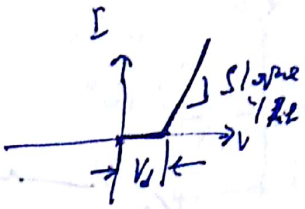
$$= \frac{V_{DD} - V_{out}}{R_p}$$

$$= \frac{5 - 4.61}{200} = \frac{0.39}{200} = \underline{1.95\ mA}$$





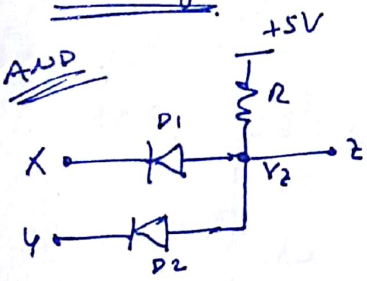




0-2 −0
2-3 −1
3-5 −1

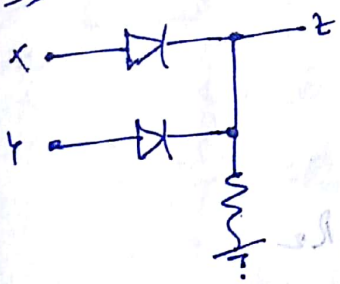
Diode logic

AND



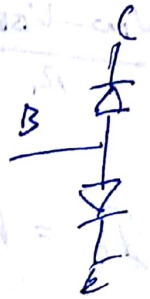
X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

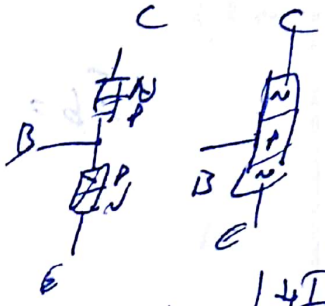
OR

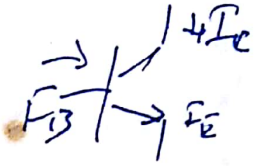


X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

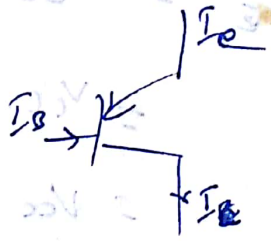
Transistor logic



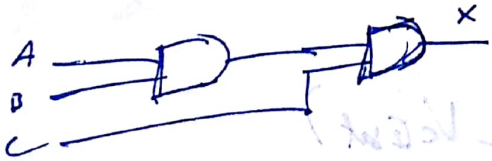


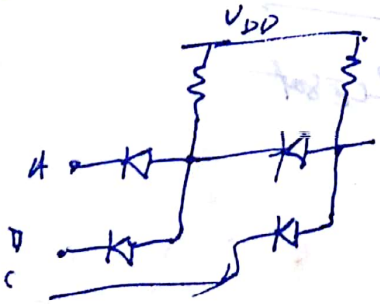


$I_E = I_B + I_C$



$I_C = I_B + I_E$



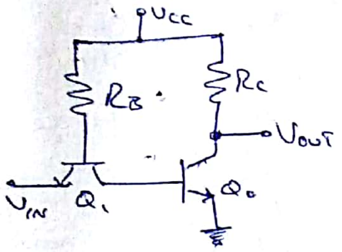


DCTL

RTL

DTL

Simple TTL inverter



⇒ V_{IN} is low, Q1 B-E junction forward biased. Firstranker's choice of R_B → Q_1 saturates for $V_{IN} = V_{Lo}$

V_{BE_0} i.e. voltage across Emitter to Base transistor of Q_0 is $V_{BE_0} = V_{IN} + V_{CE1}(sat)$

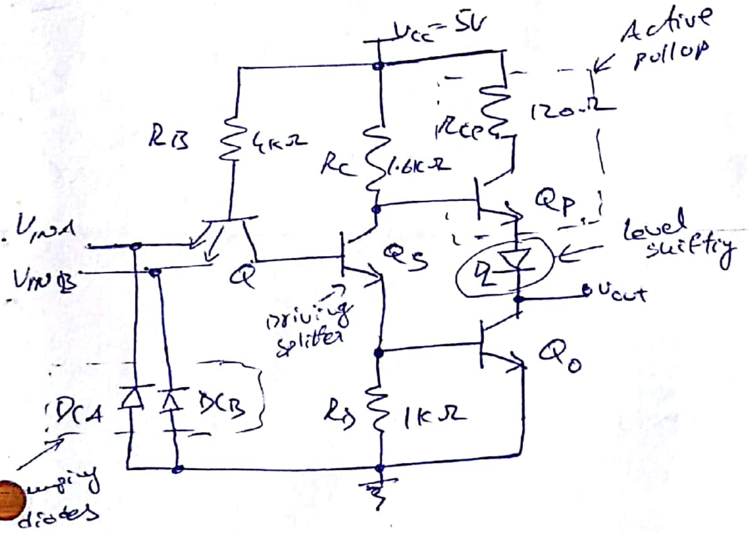
$\doteq V_{Lo} + 0.2V$

$= 0 + 0.2 = 0.2V$ this is in sufficient to drive transistor Q_0 hence Q_0 is in cut off

$\therefore V_{OH} = V_{OUT} = V_{CC}$

⇒ Increasing V_{IN} → V_{BEO} will reach 0.7V (ON) → Q_0 will forward bias

Standard TTL NAND gate (7400)



→ clamping diodes: DC : To protect i/p from negative spikes

→ Discharge resistor R_D : Provides a discharge path for the base of Q_0 when switching from sat. to cutoff (OL to OH) Faster speed.

→ Active pull-up (Q_P & R_{CP}): Provides larger current to load gates w/o for faster charging of load bases and increased fan-out.

→ level shifting Diode (DL) : Provides voltage isolation (0.7V) b/w Q_0 & Q_P to ensure no simultaneous conduction of both.

LEVEL SHIFTING DIODE

For OL, to ensure Q_P & Q_0 will not conduct simultaneously.

Q_S = sat, Q_P = cutoff?? & Q_0 = sat. $V_{out} = OL$

$V_{BEP} = V_{BP} - V_{EP}$ →(1)

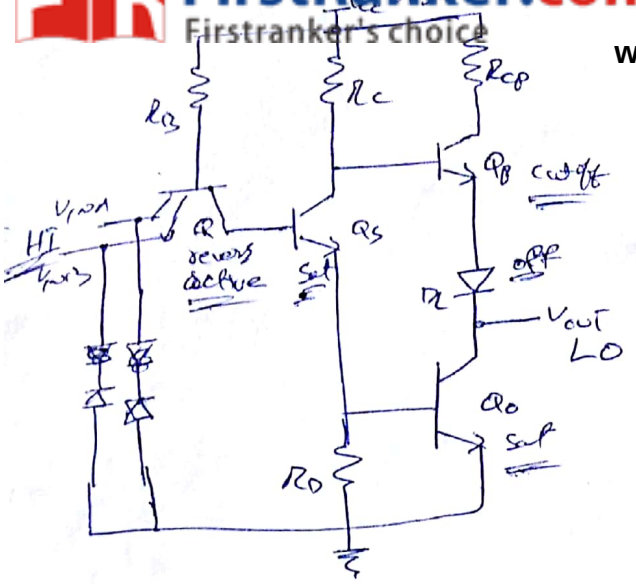
$V_{BP} = V_{BEo}(sat) + V_{CES}(sat)$ →(2)

$V_{EP} = V_{CEo}(sat) + V_{DL}(on)$ →(3)

$\therefore V_{BEP} = V_{BEo}(sat) + V_{CES}(sat) - V_{CEo}(sat) + V_{DL}(on)$

$= V_{BEo}(sat) + V_{DL}(on) < V_{BE}(FA)$

→ Q_P cannot be conducting → cutoff because of level shifting DL.



If either input of V_{INB} is low (i.e=0) then Q is ON & the base voltage is at its lowest value $V - I_1 R$. Here again R is used generally each element having distinct values in practice. The collector of Q & the base of Q_S are low so that Q_S is off. In this situation the base of Q_P is high turning this transistor ON while the base of Q_O is 0 & this transistor is off. In this situation the o/p is connected to the high voltage line 'V' & goes high.

if V_{INA} & V_{INB} Both are at High level Q is off, turning Q_S ON. This reduces the voltage on the base of Q_P & below its emitter voltage so that it becomes non-conducting. The base of Q_O on the other hand is driven up by an amount $I_2 R$ & becomes Conducting. ~~The base of Q_O on the other hand is driven~~ In this case the o/p is connected to ground through Q_O & the o/p is low.

V_{INA}	V_{INB}	Q	Q_S	Q_P	Q_O	O/P
0	0	ON	OFF	ON	OFF	1
0	1	ON	OFF	ON	OFF	1
1	0	ON	OFF	ON	OFF	1
1	1	OFF	ON	OFF	ON	0

Behavioural Modeling

Entity Declaration:-

1. An entity declaration describes the external interface of the entity, that is it gives the black-box view.
2. It specifies the name of the entity, the names of interface ports, their mode (i.e direction) and the type of ports.

The syntax is

```
entitity entity_name is
    generic (list-of-generics-and-their types);
    Port (list-of-interface-port-names-and-their
    types);
    Entity-item-declarations]

[begin
    entity-statements]

end entity-name;
```

Architecture Body:-

1. It describes the internal view of an entity.
2. It describes the functionality or the structure of the entity
3. The syntax is

```
architecture architecture-name of entity-name
    [architecture_item-declarations]
begin
    Concurrent-statements;
```

Process - statement
block - statement
Cocurrent - Procedure - Call
Concurrent - assertion - statement
Concurrent - signal - assignment - statement
Component - instation - statement
Generate - statement

End architecture_name;

3) Process Statement:-

1. A Process statement contains sequential state-ments that describe the functionality of a portion of an entity in sequential terms.

Syntax is

[Process_label:] Process [(sensitivity_list)]
[Process - item - declarations]
begin
Sequential - statements;
Variable - assignment - statement
Signal - assignment - statement
wait - statement
if - statement
case - statement
loop - statement
null - statement
exit - statement
next - statement
assertion - statement
Procedure - call - statement
Return - statement
end process [Proces - label];

3.2 Variable Assignment statement:-

1. Variables can be declared and used inside a process statement.
2. A variable is assigned a value using the variable assignment statement that typically has the form

Variable-object = expression

Consider the following process statement

```
Process (A)
    Variable events_on_A: Integer:=0;
begin
    events_on_A := events_on_A+1;
end process;
```

3.3. Signal Assignment statement:-

1. Signals are assigned values using a signal assignment statement the simplest form of signal assignment statement is

Signal-object <= expression [after delay-value];

3.4 Wait Statement:-

1. A process may be suspended by means of a sensitivity list.
2. That is, when a process has a sensitivity list.
3. It always suspends after executing the last sequential statement in the process.
4. The wait statement provides an alternative way to suspend the execution of a process.

www.FirstRanker.com

There are three basic forms of the wait statement

wait on Sensitivity_list;
wait until boolean_expression;
wait for time_expression;

5. They may also be combined in a single wait statement.

3.5 If statement:

1. An 'if' statement selects a sequence of statements for execution based on the value of a condition.
2. The condition can be any expression that evaluates to a boolean value.

The general form of an if statement is

```
if boolean_expression
        sequential_statements
elsif boolean_expression then
        sequential_statements
else
        sequential_statements
end if;
```

3.6 Case statement:

The format of a case statement is

```
case expression is
    when choices => seq_statements
    when choice => seq_statements
    --- can have any no. of statements
    when others => seq_statements
end case;
```

3.7 Null Statement :-

1. The statement
   null; is a sequential statement that does not cause any action to take place and execution continues with the next statement.

3.8 Loop statement :-

1. A loop statement is used to iterate through a set of sequential statements.
2. The syntax of a loop statement is
   [loop-label :] iteration-scheme that has the form for identifier. in range.

3.9 Exit statement :-

1. The exit statement is a sequential statement that can be used only inside a loop.
2. It causes execution to jump out of the innermost loop or the loop whose label is specified.
   The syntax for an exit statement is
   exit [loop-label] [when condition];

3.10 Next statement :-

1. The next statement is also a sequential statement that can be used only inside a loop.
2. The syntax is the same as that for the exit statement except that the keyword next replaces the keyword exit.

The Syntax for Next statement

next [loop-label] [when Condition];

3.11 Assertion Statement:-

1. Assertion statements are useful in modeling constrains of an entity.

The Syntax is

assert boolean-expression
[report string-expression]
[severity expression];

3.12 More on Signal Assignment Statement:-

1. Signal assignment statement used inside a process that illustrated the delta delay model.
2. Aside from the delta delay model, there are two other types of delay models that can be used with signal assignments, inertial and transport.

3.12.1 Inertial Delay Model:-

1. In this delay model the delays often found in switching circuits.
2. It represents the time for which an input value must be stable before the value is allowed to propagate to the output. Z <= A after 10ns.

delay = 10ns

3.12.2 Transport Delay Model :-

1. Transport delay models the delays in hardware that do not exhibit any inertial delay.

2. This delay represents pure propagation delay, that is any changes on an input is transported to the output, no matter how small, after the specified delay.

Z ⇐ transport A after 10ns

delay = 10ns

3.12.3 Creating Signal waveforms :-

1. In all examples of signal assignment statements that we have seen so far, we have always assigned a single value to a signal; this need not be so.

Phase1 ⇐ '0', '1' after 8ns, '0' after 13ns, '1' after 50ns;

3.12.4 Signal drivers :-

1. The driver of a signal holds its current value and all its future values as a sequence of one or more transactions, where each transaction identifies the value to appear on the signal along with the time at which the value is to appear.

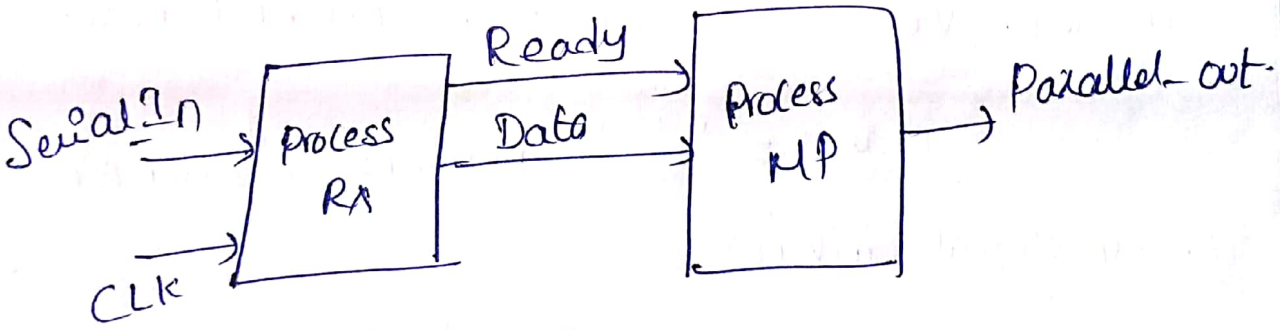
```
Proc1
begin
    Reset ⇐ 3 after 5ns, 21 after 10ns, 14 after 17ns;
end proc1;
```

3.13 Other Sequential Statements :-

1. There are two other forms of sequential statements.
   (i) Procedure Call Statement
   (ii) Return statement.

3.14 Multiple Processes :-

1. A process statement is a concurrent statement, it is possible to have more than one process in within an architecture body.
2. This makes it possible to capture the behaviour of interfacing processes.



A Combinational logic circuit is one whose o/ps depend only on its current i/ps.

A Combinational circuit may Contain an arbitrary no. of logic gates and inverters but no feedback loops.

In combinational circuit, ANALYSIS we start with a logic diagram & proceed to a formal description of the function performed by that circuit. Such as a truth table or a logic expression.

In SYNTHESIS we do the reverse, starting with a formal description & proceeding to a logic diagram.

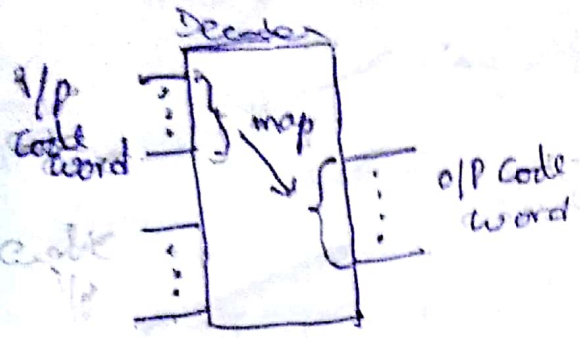
DECODER

A decoder is a multiple-i/p, multiple-o/p logic circuit that converts coded i/p into coded o/ps, where the i/p & o/p codes are different.

The i/p code generally has fewer bits than the o/p code, & there is a one-to-one mapping from i/p code words into o/p code words.

In a one-to-one mapping, each i/p code word produces a different o/p code word.

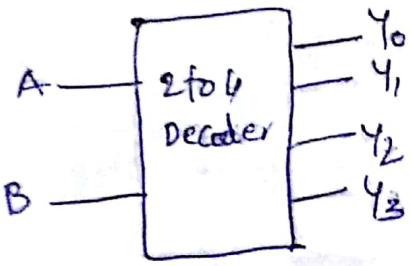
General Structure of decoder.



The most commonly used i/p code is an n-bit binary code, where an n-bit word represents one of 2^n different coded values, normally the integers from 0 to $2^n - 1$.

The most commonly used o/p code is a 1 out of m code which contains m bits, where one bit is asserted at any time.

2 to 4 Decoder

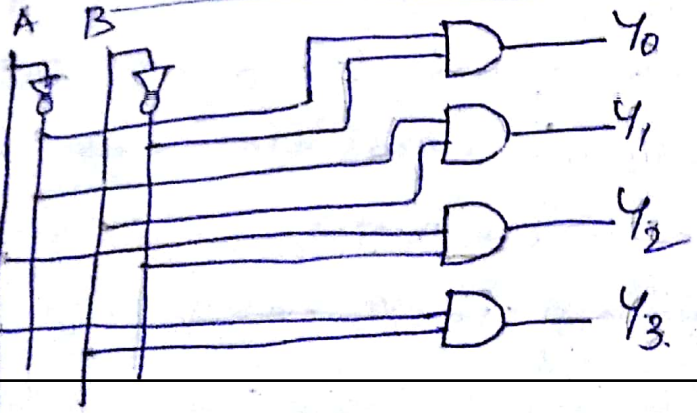


logic - Symbol

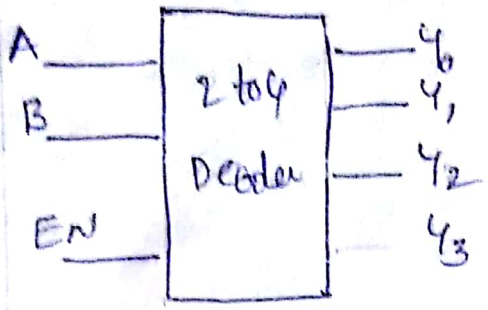
A	B	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Truth table

Internal Structure



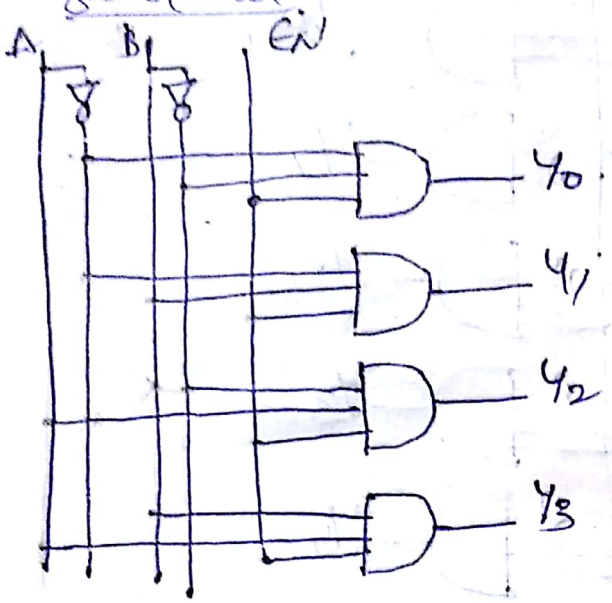
2 to 4 decoder with Enable i/p



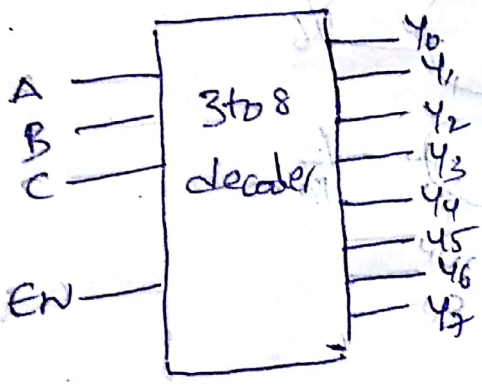
Truth table

EN	A	B	Y_3	Y_2	Y_1	Y_0
0	x	x	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

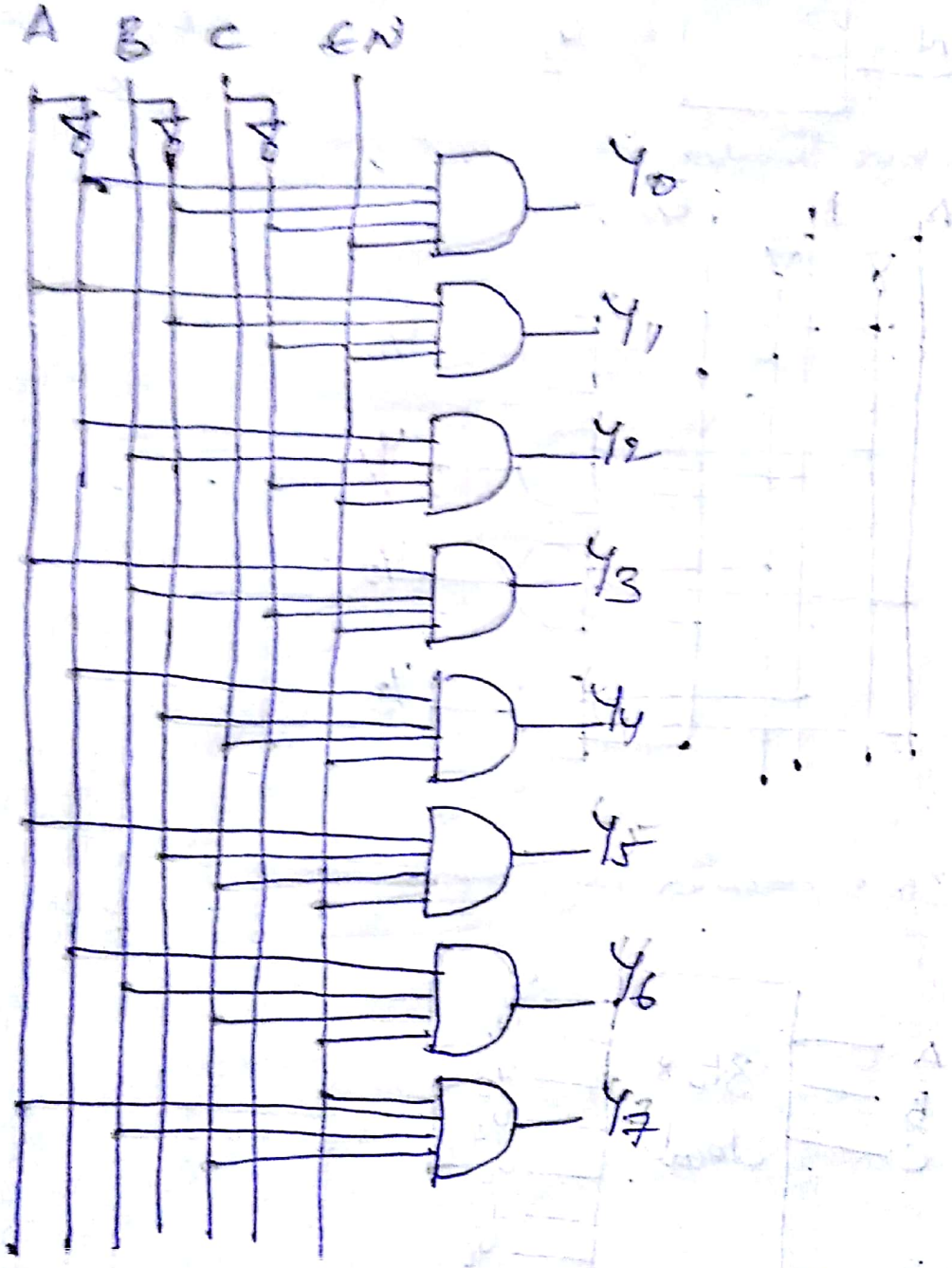
logic symbol



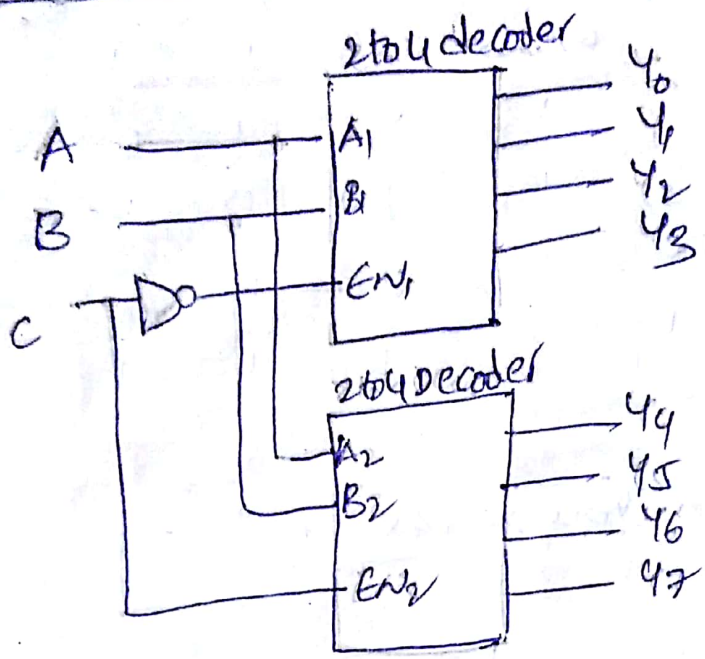
3 to 8 decoder with Enable i/p



EN	A	B	C	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	x	x	x	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0



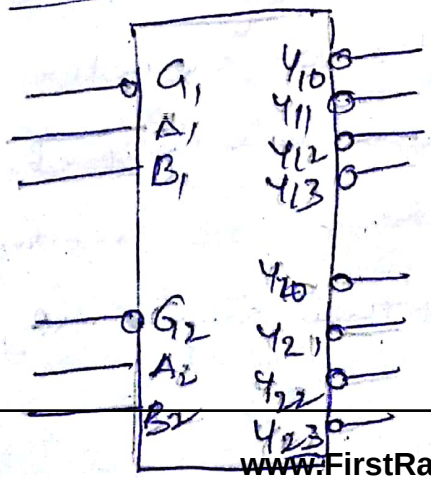
Design 3 to 8 decoder using 2 to 4 decoder



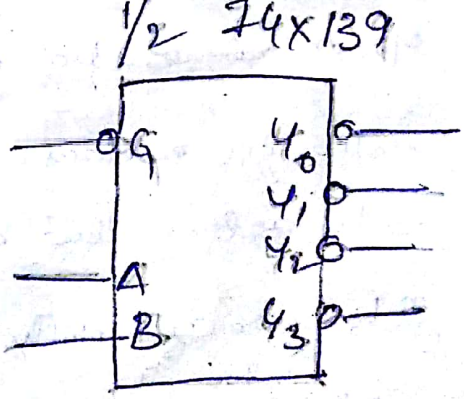
C	B	A	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

MSI DECODER

74x139 Dual 2 to 4 decoder

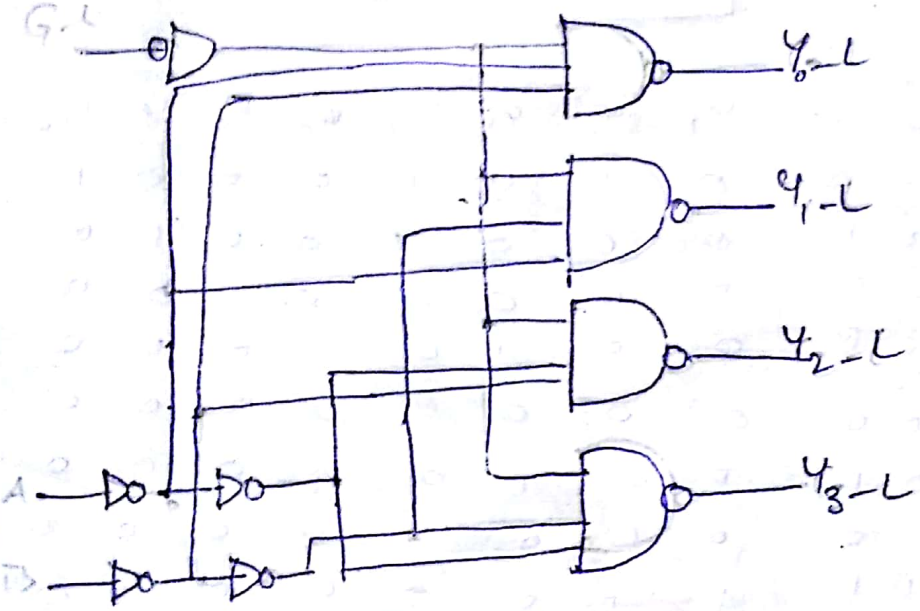


1/2 74x139



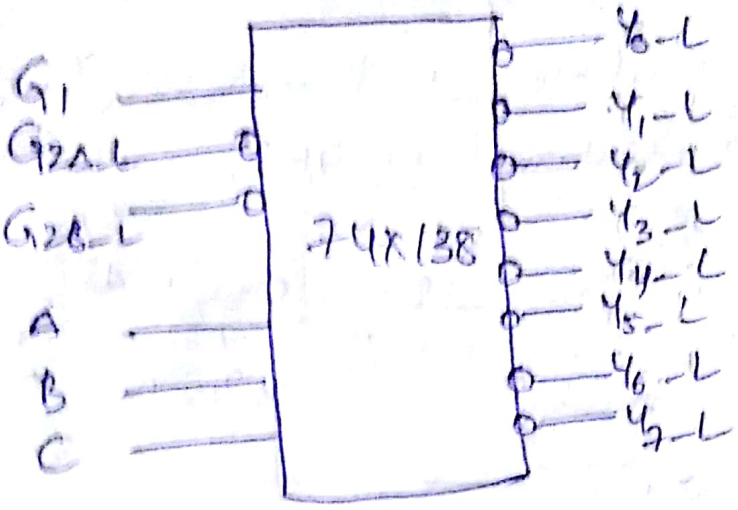
G-L	A	B	Y_3-L	Y_2-L	Y_1-L	Y_0-L
1	x	x	1	1	1	1
0	0	0	1	1	1	0
0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	1

Internal Structure of 74X139



Two identical & independent 2 to 4 decoders are contained in a single MSI part. the 74X139.

The internal structure shows that the o/p & the enable i/p of the 139 are active low. Most MSI circuits were originally designed with active-low o/ps. Since TTL inverting gates are generally faster than noninverting ones.



G_1	G_{2A}-L	G_{2B}-L	A	B	C	Y_{7}-L	Y_{6}-L	Y_{5}-L	Y_{4}-L	Y_{3}-L	Y_{2}-L	Y_{1}-L	Y_{0}-L
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	1	1	1	1	1	1	1	0
1	0	0	0	0	1	1	1	1	1	1	1	0	1
1	0	0	0	1	0	1	1	1	1	1	0	1	1
1	0	0	0	1	1	1	1	1	1	0	1	1	1
1	0	0	1	0	0	1	1	1	0	1	1	1	1
1	0	0	1	0	1	1	1	0	1	1	1	1	1
1	0	0	1	1	0	1	0	1	1	1	1	1	1
1	0	0	1	1	1	0	1	1	1	1	1	1	1

Like the 74x139 the 74x138 has active-low o/ps & it has three enable i/ps (G_1, G_{2A}-L, G_{2B}-L), all of which must be asserted for the selected o/p to be asserted.

The logic function of '138' is straightforward - an o/p is asserted if & only if

Thus, we can easily write logic statement for an internal o/p signal such as Y_5 in terms of the input signal.

$$Y_5 = \underbrace{G_1 \cdot G_{2A} \cdot G_{2B}}_{\text{enable}} \cdot \underbrace{C \cdot \bar{B} \cdot A}_{\text{select}}$$

$$G_{2A} = \overline{G_{2A\_L}}$$

$$G_{2B} = \overline{G_{2B\_L}}$$

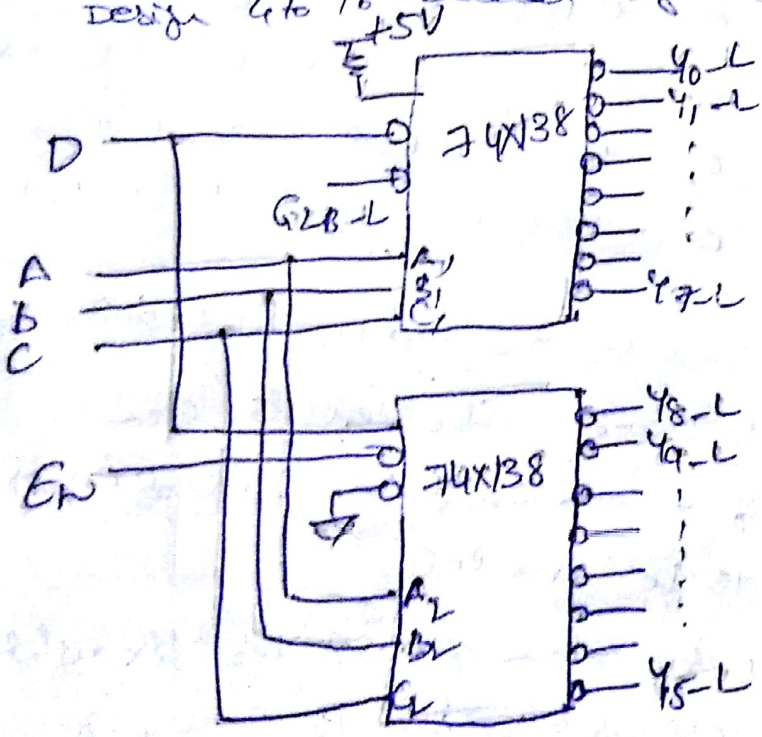
$$Y_5 = \overline{Y_{5\_L}}$$

$$\overline{Y_{5\_L}} = \overline{Y_5} = (G_1 \cdot \overline{G_{2A\_L}} \cdot \overline{G_{2B\_L}} \cdot C \cdot \bar{B} \cdot A)$$

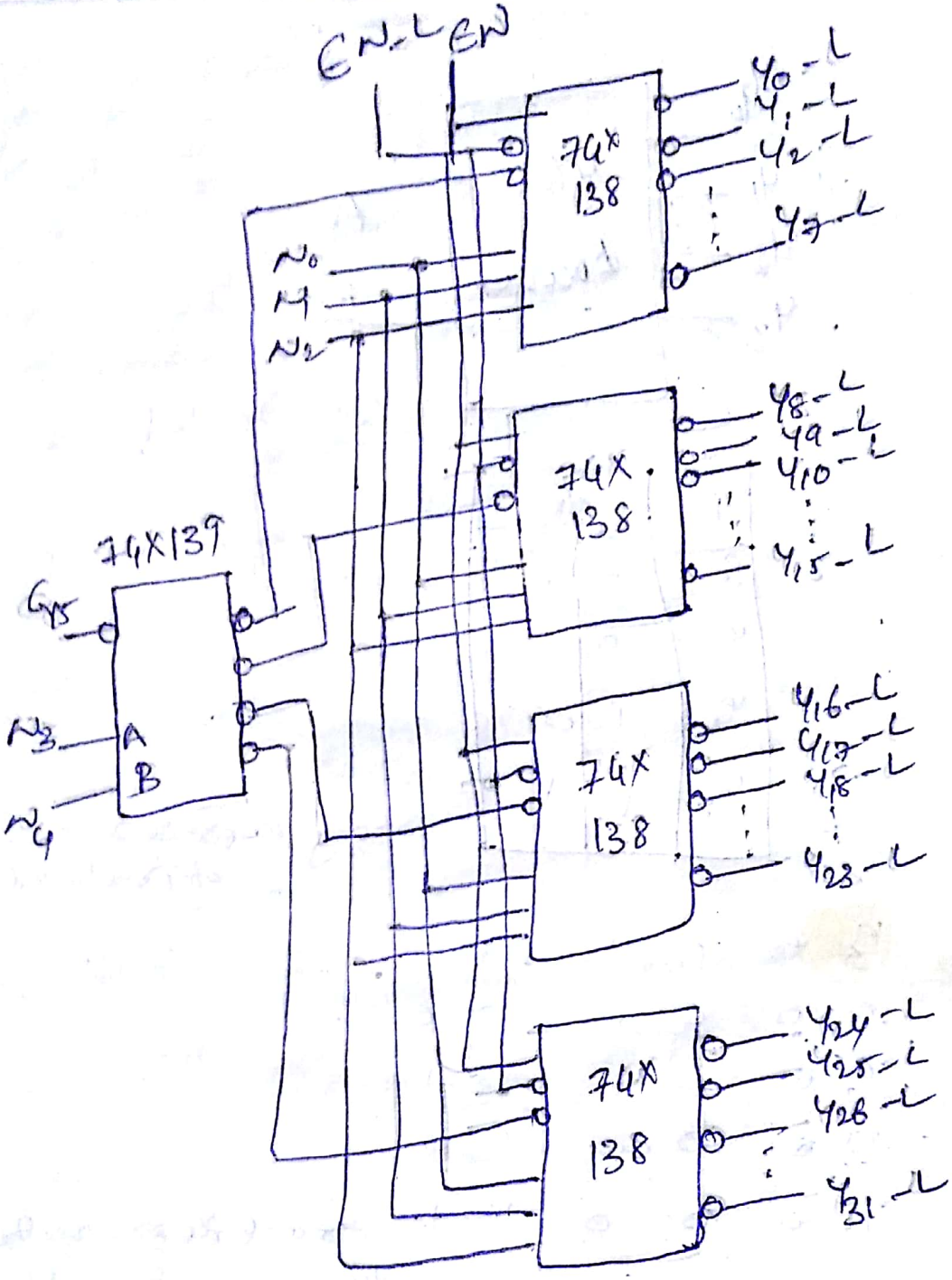
$$= \bar{G}_1 + G_{2A\_L} + G_{2B\_L} + \bar{C} + B + \bar{A}$$

• CASCADING BINARY DECODERS

Design 4 to 16 decoder using 74x138 decoder



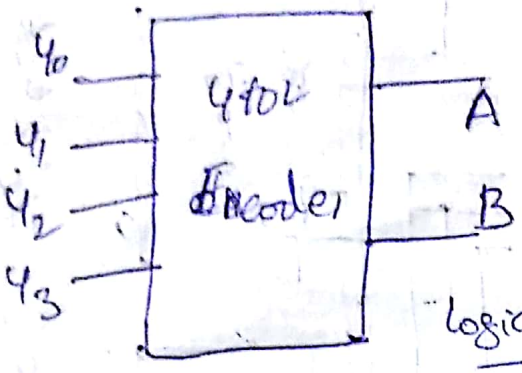
Design 5 to 32 decoder using 74x139 & 74x138



ENCODERS

It is a combinational logic circuit. It has 2^n i/ps & n o/ps. It performs exact opposite function of decoder. It performs some coding operations.

Design 4 to 2 Encoder



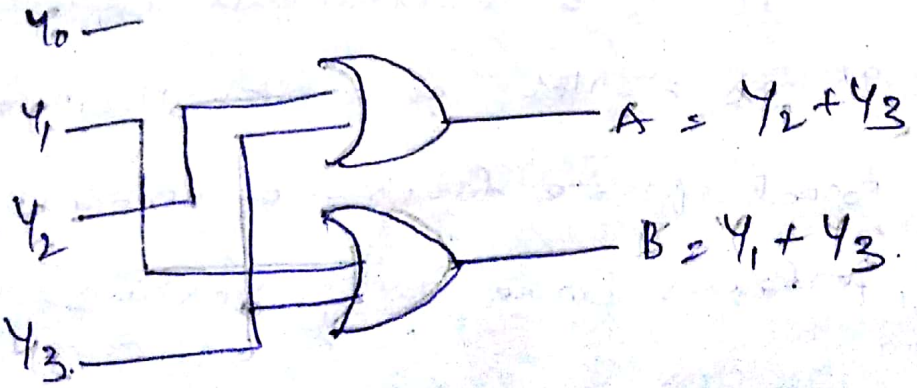
Logic Symbol

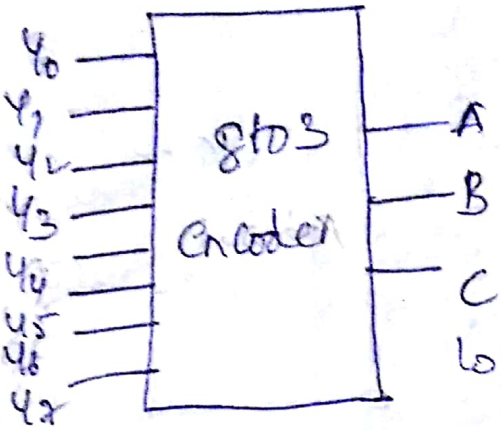
IPS	OPS	
	A	B
Y_0	0	0
Y_1	0	1
Y_2	1	0
Y_3	1	1

wrong Representation of Truth table

Y_3	Y_2	Y_1	Y_0	A	B
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Exact Representation of Truth table.

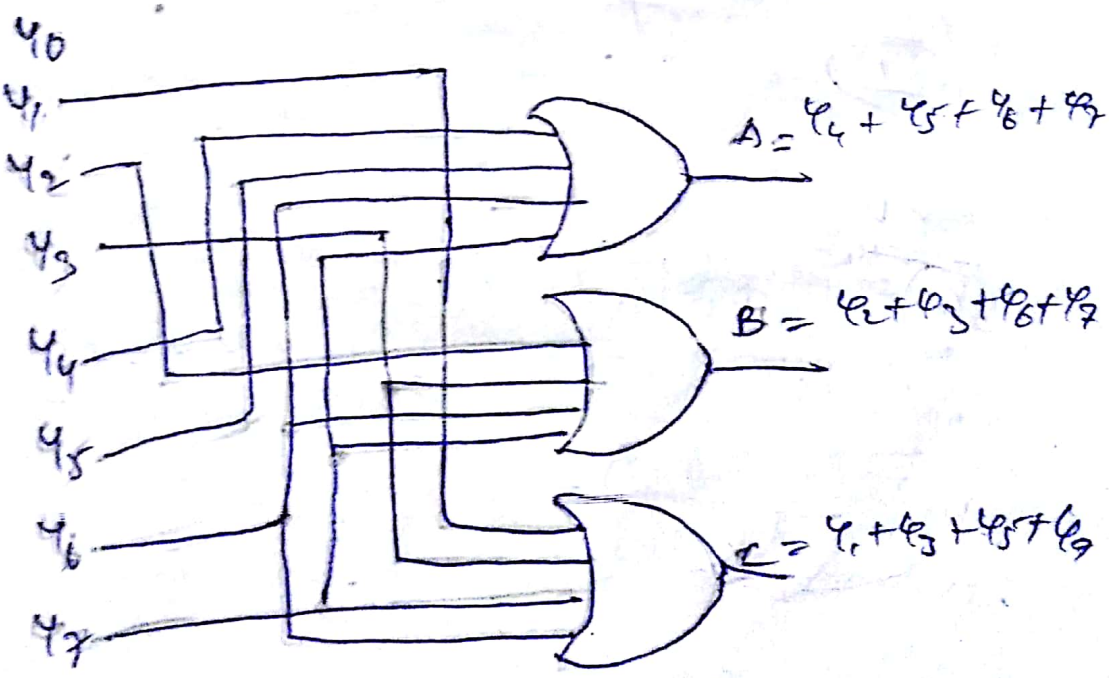




Logic Symbol

Truth Table:

Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	A	B	C
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1



Internal Structure

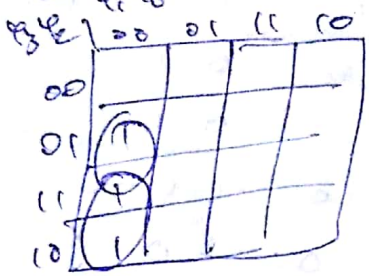
4-bit Priority Encoder

I_3	I_2	I_1	I_0	A	B	V
x	x	x	1	0	0	1
x	x	1	0	0	1	1
x	1	0	0	1	0	1
1	0	0	0	1	1	1
0	0	0	0	x	x	0

I_0 -> highest priority

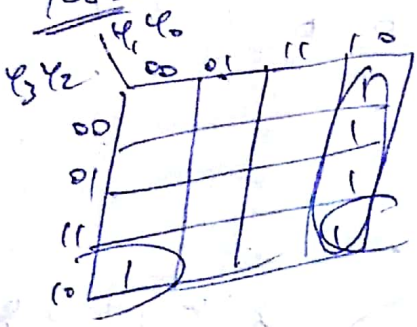
V is valid i/p

For A



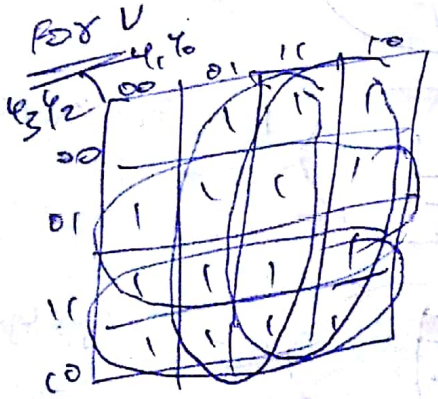
$$A = \overline{I_1}\,\overline{I_0}\,I_3 + \overline{I_1}\,\overline{I_0}\,\overline{I_2}$$

For B

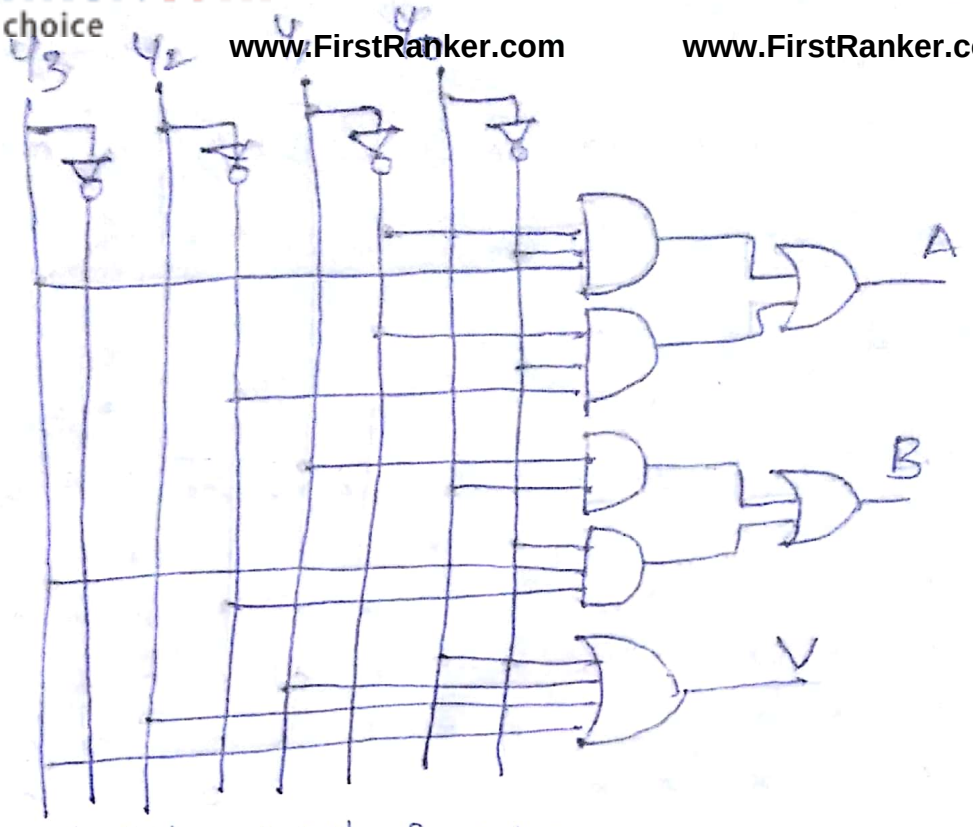


$$B = I_1 I_0 + \overline{I_0}\, I_3\, \overline{I_2}$$

For V

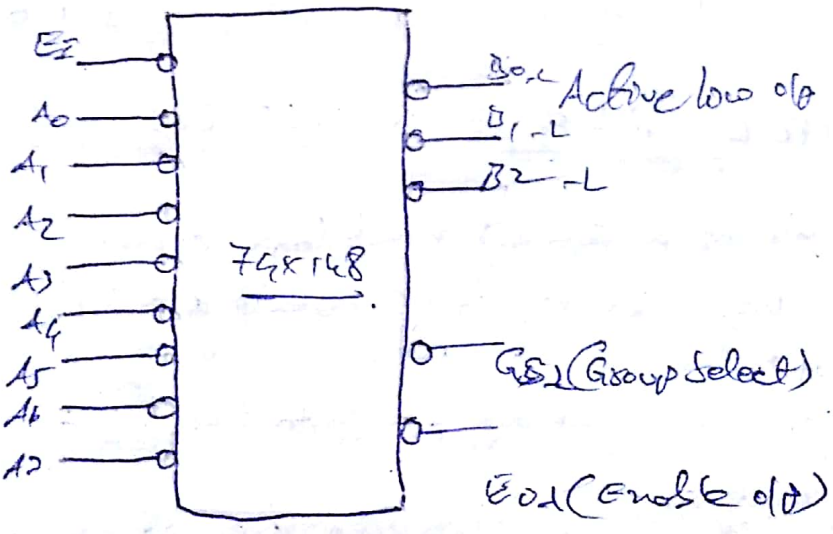


$$V = I_0 + I_1 + I_2 + I_3$$



4-bit Priority Encoder

74x148 is a 8bit priority Encoder



The 148 has a GS_L o/p that is asserted when the device is enabled & one or more of the request i/ps are asserted. The EO_L signal is an enable o/p designed to be connected to the EI_L i/p of another 148 that handles lower priority requests.

EOL is asserted, EIL is asserted but no request i/p is asserted; Thus a lower priority I/P may be enabled.

Truth Table

i/ps									o/ps				
EIL	I_{0L}	I_{1L}	I_2	I_3	I_4	I_5	I_6	I_7	A_2	A_1	A_0	GS	EO
1	x	x	x	x	x	x	x	x	1	1	1	1	1
0	x	x	x	x	x	x	x	0	0	0	0	0	1
0	x	x	x	x	x	x	0	1	0	0	1	0	1
0	x	x	x	x	x	0	1	1	0	1	0	0	1
0	x	x	x	x	0	1	1	1	0	1	1	0	1
0	x	x	x	0	1	1	1	1	1	0	0	0	1
0	x	x	0	1	1	1	1	1	1	0	1	0	1
0	x	0	1	1	1	1	1	1	1	1	0	0	1
0	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	0

MULTIPLEXERS & DEMULTIPLEXERS

Mux: It is a combinational logic circuit which has selection i/p & data i/p & only one output.

n selection i/ps & 2^n data inputs & only one o/p.

Mux is also called as data selector(or) digital switch.

Design 2:1 Mux

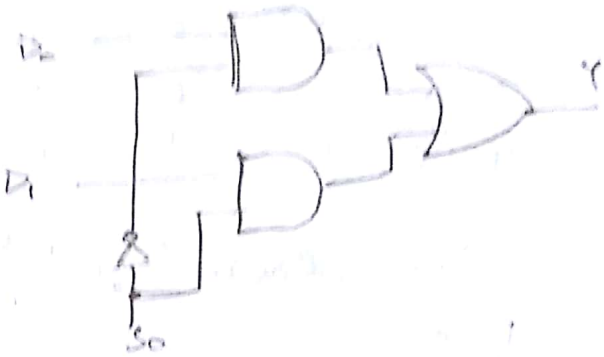
2 → data i/ps

1 → selection i/p

1 → o/p.

Truth Table

S_0	Y
0	D_0
1	D_1

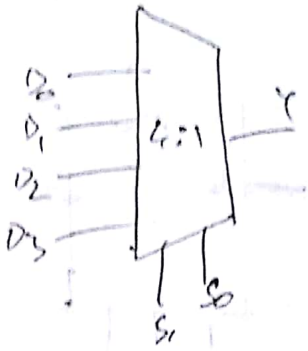


Design 4:1 Mux

4 - data i/ps

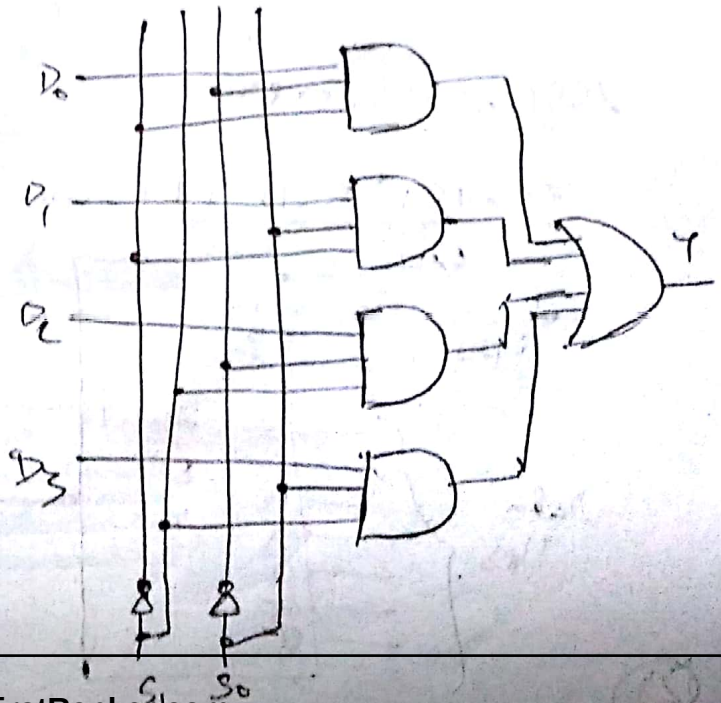
2 - Selection i/ps

01 - o/p.



Truth Table

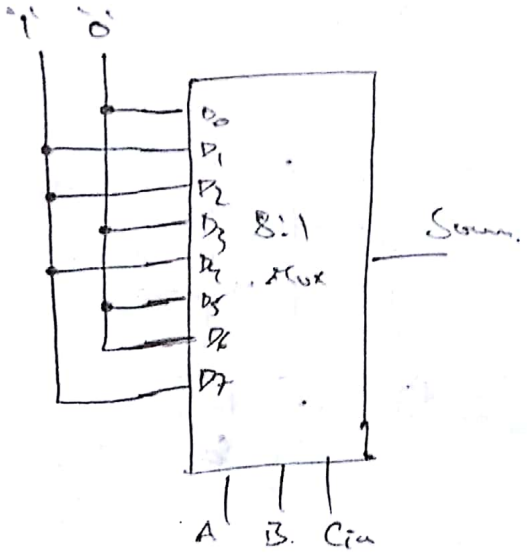
S_1	S_0	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3



Realize Full adder

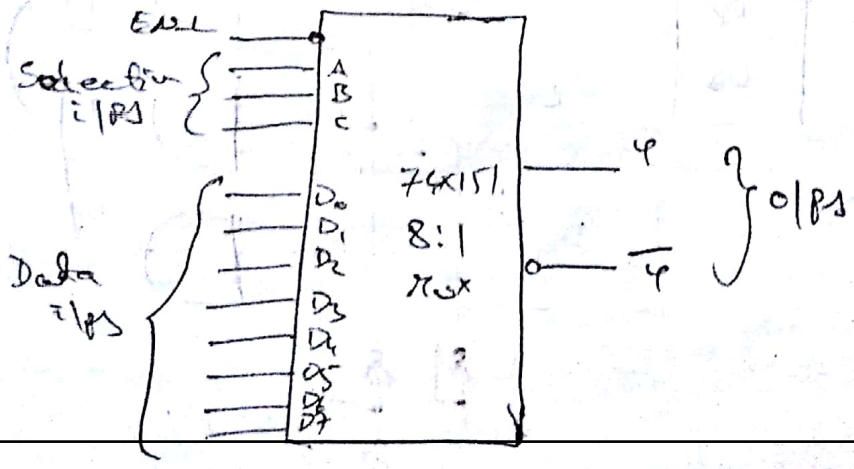
A	B	Cin	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$S(A, B, C_{in}) = \sum_m (1, 2, 4, 7)$$



MSI multiplexers

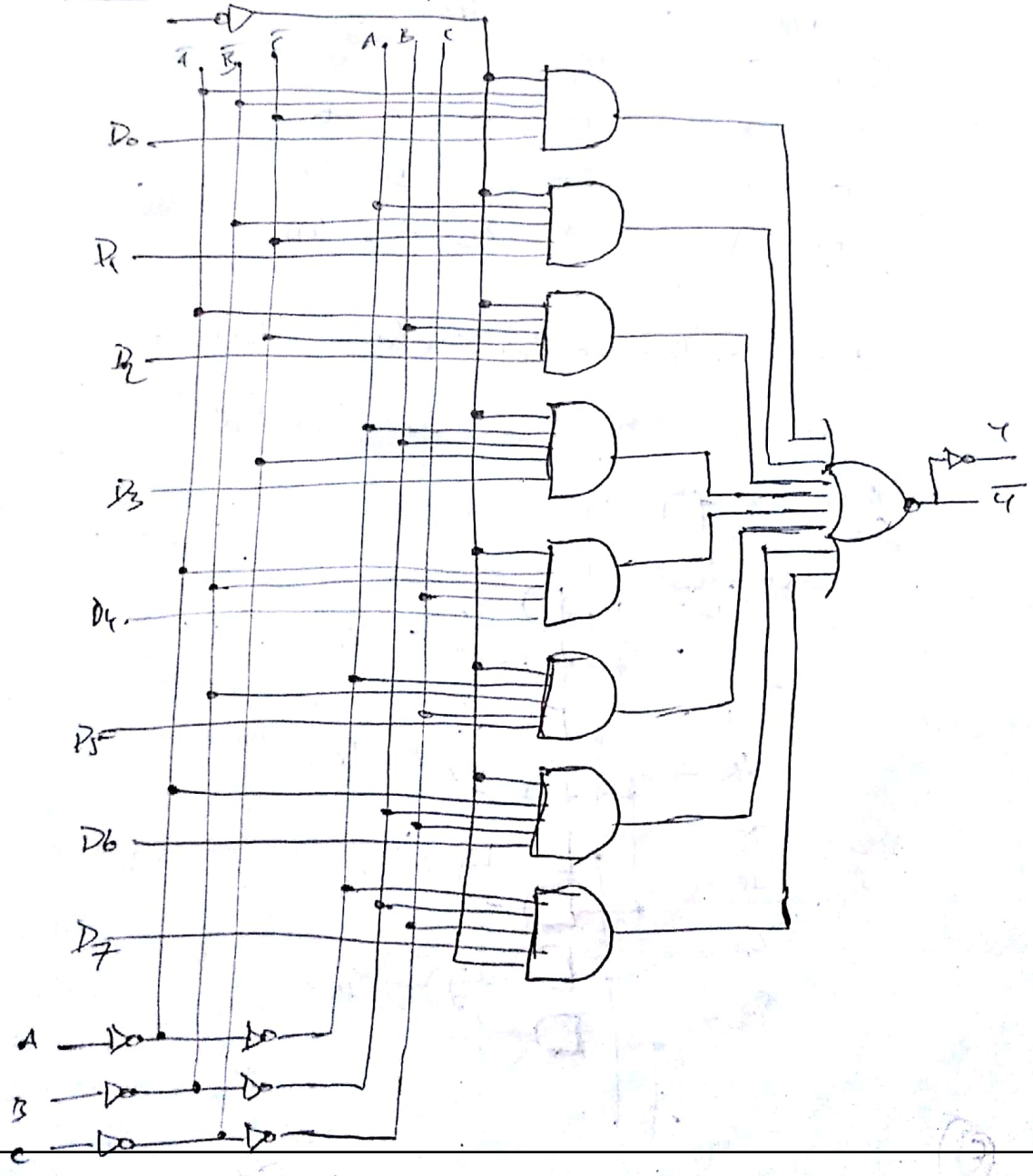
74x151 8:1 MUX 1 bit mux

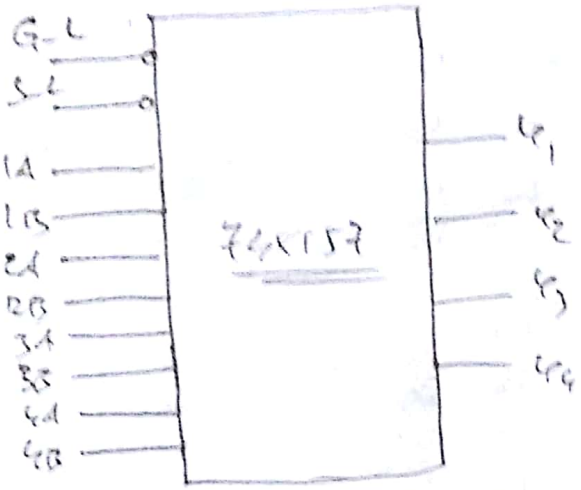


Truth table

I/P				O/P	
EN	C	B	A	Y	$\bar{Y}$
1	x	x	x	0	1
0	0	0	0	D_0	$\bar{D_0}$
0	0	0	1	D_1	$\bar{D_1}$
0	0	1	0	D_2	$\bar{D_2}$
0	0	1	1	D_3	$\bar{D_3}$
0	1	0	0	D_4	$\bar{D_4}$
0	1	0	1	D_5	$\bar{D_5}$
0	1	1	0	D_6	$\bar{D_6}$
0	1	1	1	D_7	$\bar{D_7}$

Internal Structure





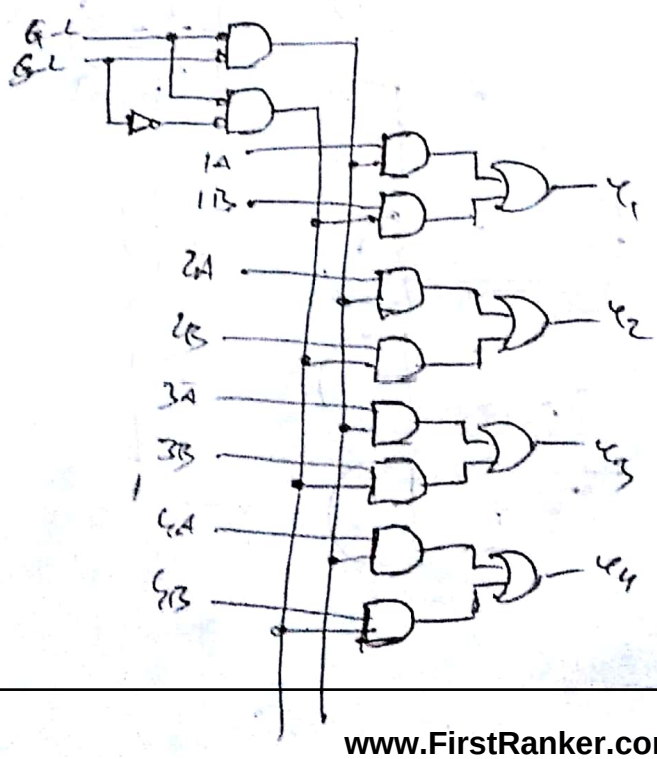
Logic symbol

Truth Table

I/P		O/Ps			
G-L	S-L	Y_1	Y_2	Y_3	Y_4
1	X	0	0	0	0
0	0	1A	2A	3A	4A
0	1	1B	2B	3B	4B

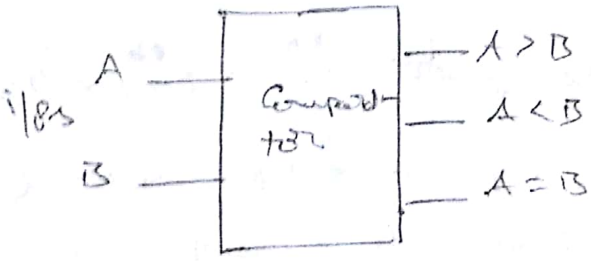
S-L → Active low selection I/P line.

G-L → Active low gate pulse



Comparators

Comparator is a combinational logic ckt which compares two binary (group of bits) of any length & gives the appropriate result i.e. greaterthan, lessthan & Equal to its o/p terminals.



If A & B are represent the only one bit length then

A	B	A>B	A<B	A=B
0	0	0	0	1
0	1	0	1	0
1	0	1	0	0
1	1	0	0	1

the above truth table has four rows.

$2^{exn} = 2^{2\times 2|} = 4$ rows.

where 'n' is no. of bits per i/p.

Both i/ps must be equal length.

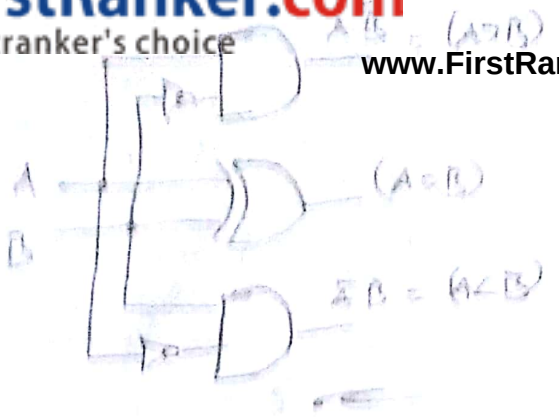
From the Truth table

$(A > B) = A\bar{B}$

$(A < B) = \bar{A}B$

$(A = B) = \bar{A}\bar{B} + AB$

$= \overline{A \oplus B} = A \odot B$



Four Bit Comparator

For four bit comparator $2^{4\times4} = 2^{8} = 256$ rows. It is very complicated to design by using the above procedure (truth table method)

Hence to simplify the design compare from equal weighted bits from MSB to LSB & determine the conditions greater, less & equals.

$A = A_3\ A_2\ A_1\ A_0$

$B = B_3\ B_2\ B_1\ B_0$

If A_3 is 1 & $B_3 = 0$ then $A_3 > B_3$, if $A_3 = B_3$ only then we can compare A_2 & B_2 & else we should not compare A_2 & B_2. If the same way only if $A_3 = B_3$, $A_2 = B_2$ $A_1 = B_1$ then only we have to compare A_0 & B_0 the result is if

$$A>B \Rightarrow A_3>B_2 + x_3(A_2>B_2) + x_3x_2'(A_1>B_1) + x_3x_2x_1(A_0>B_0)$$

$$\Rightarrow A_3\bar{B_3} + x_3A_2\bar{B_2} + x_3x_2A_1\bar{B_1} + x_3x_2x_1A_0\bar{B_0}$$

$(A<B) \Rightarrow$ $\bar{A}_3 B_3 + x_3 \bar{A}_2 B_2 + x_3 x_2 \bar{A}_1 B_1 + x_3 x_2 x_1 \bar{A}_0 B_0$

$(A=B) \Rightarrow$ $x_3 x_2 x_1 x_0$

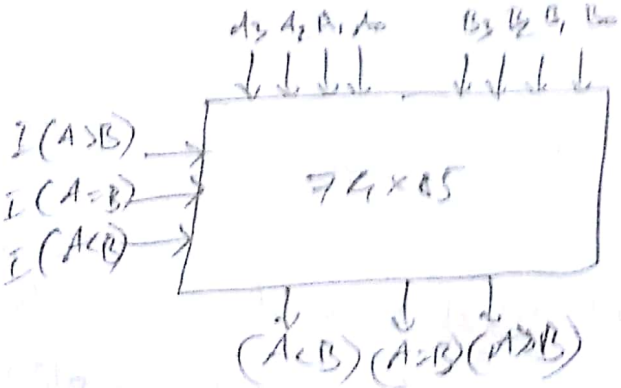
where $x_3 = A_3 \odot B_3$

$x_2 = A_2 \odot B_2$

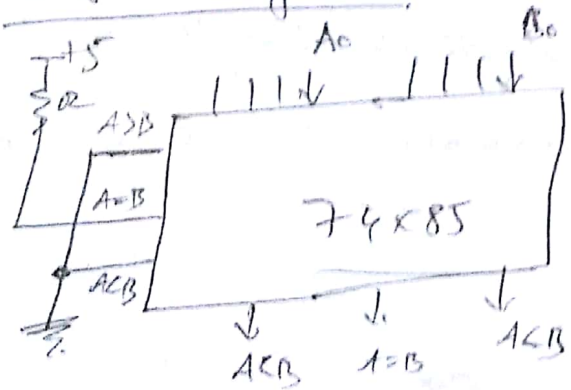
$x_1 = A_1 \odot B_1$

$x_0 = A_0 \odot B_0$

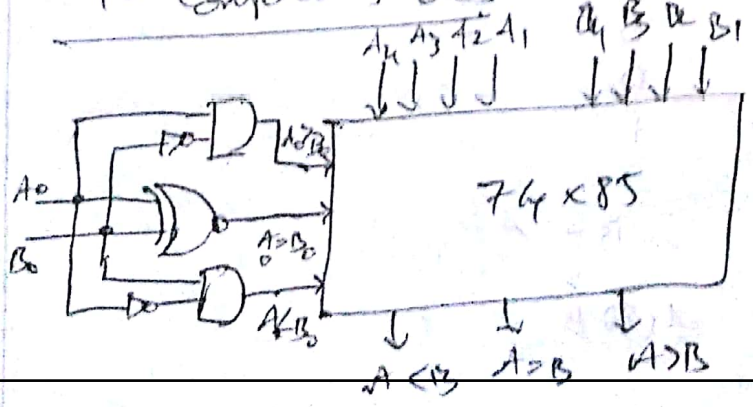
MSI Comparator



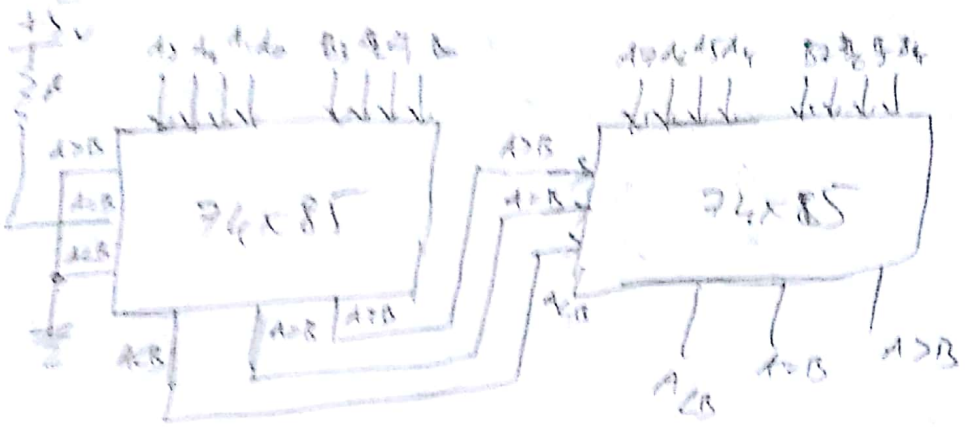
To Compare Single Bit



To Compare 4 bits



Design 8-bit comparator using 74x85



ADDERS

HALF Adders
FULL Adders.

Half adder is used to add two single bits & it generates Sum & Carry o/ps.

Full adder is a Combinational circuit which is used to add three single bits & it generates Sum & Carry o/ps.

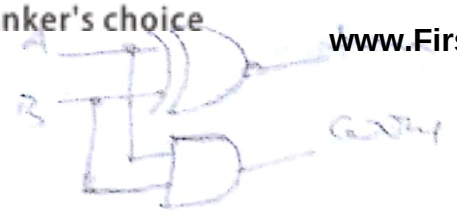
Half adder

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$$Sum = \bar{A}B + A\bar{B}$$

$$= A \oplus B$$

$$Carry = A\bar{B}$$



Full Adder

A	B	C_{in}	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

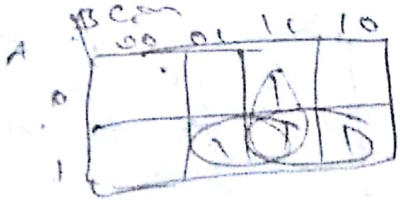
$$Sum = A\bar{B}\bar{C_{in}} + \bar{A}\bar{B}C_{in} + A B C_{in} + \bar{A} B \bar{C_{in}}$$

$$= A(\bar{B}\bar{C_{in}} + BC_{in}) + \bar{A}(\bar{B}C_{in} + B\bar{C_{in}})$$

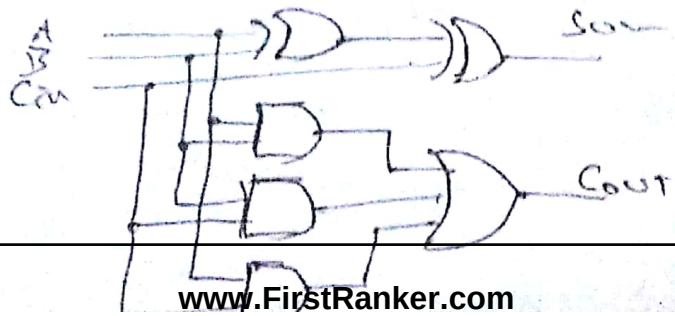
$$= A(\overline{B \oplus C_{in}}) + \bar{A}(B \oplus C_{in})$$

$$Sum = A \oplus B \oplus C$$

Cout



$$Cout = BC_{in} + AB + AC_{in}$$



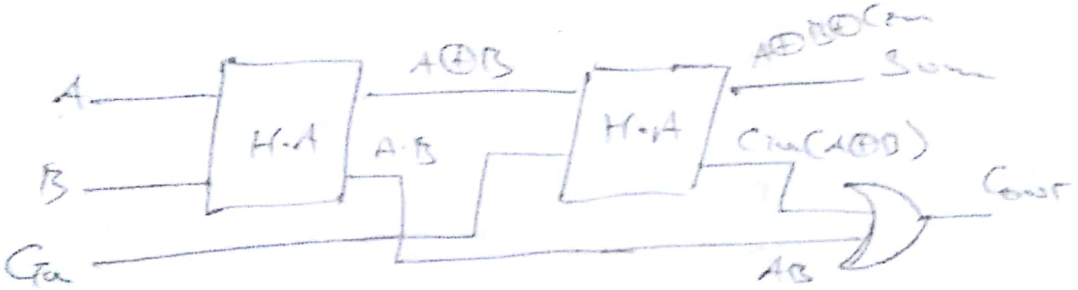
Full adder using half adder

$$Sum = A \oplus B \oplus C_{in}$$

$$C_{out} = \bar{A}BC_{in} + A\bar{B}C_{in} + AB\bar{C}_{in} + ABC_{in}$$

$$= C_{in}(\bar{A}B + A\bar{B}) + AB(\bar{C}_{in} + C_{in})$$

$$C_{out} = C_{in}(A \oplus B) + AB$$

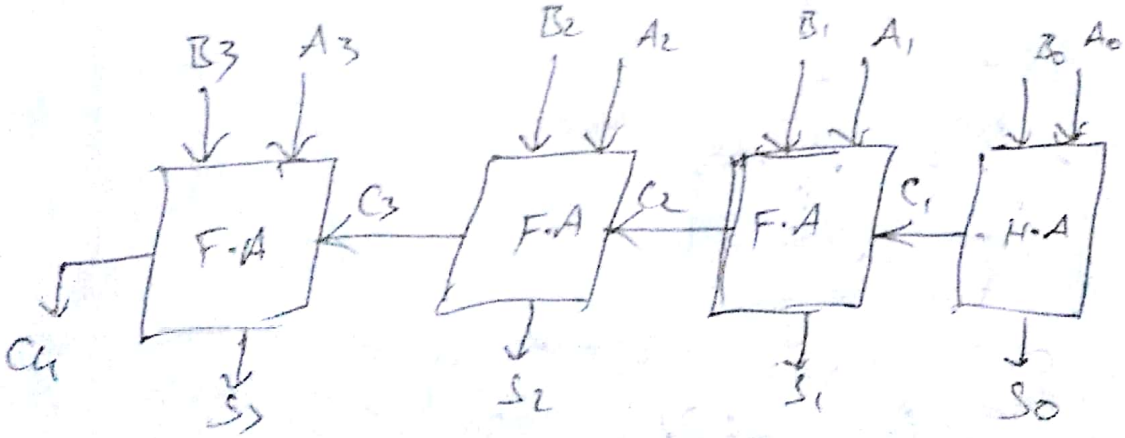


4-bit parallel adder (Ripple carry Adder)

$$A \Rightarrow A_3\ A_2\ A_1\ A_0 \quad (C_3\ C_2\ C_1)$$

$$B \Rightarrow B_3\ B_2\ B_1\ B_0$$

$$C_4\ S_3\ S_2\ S_1\ S_0$$



BASIC BISTABLE ELEMENTS

The simplest sequential circuit consists of a pair of inverters forming a feedback loop, as shown in fig 1. It has no inputs & two outputs, Q and Q_L.

Analysis. The circuit of fig 1 is often called a bistable, since a strictly digital analysis shows that it has two stable states. If Q is High, then the bottom inverter has a HIGH i/p & a LOW o/p, which forces the top inverters o/p HIGH as we assumed in the first place. But if Q is Low, then the bottom inverter has a LOW i/p & a HIGH o/p, which forces Q Low, another stable situation. We could use a single state variable, the state of signal Q, to describe the state of the circuit; there are two possible states, Q=0 & Q=1.

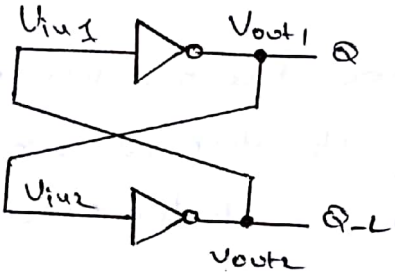


Fig 1: A pair of inverters forming a bistable element.

The bistable element is so simple that it has no inputs & therefore no way of controlling or changing its state. When power is first applied to the circuit, it randomly comes up in one state & the other and stays there forever. Still, it serves our illustrative purpose very well, and we will actually show a couple of applications.

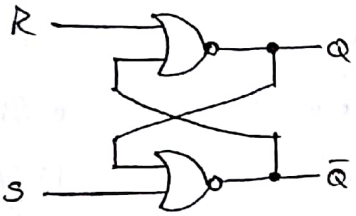
LATCHES & FLIP FLOPS

Latches & FlipFlops are the basic building blocks of most sequential circuits. Typical digital systems use latches & FlipFlops that are prepackaged functionally specified devices in a standard IC. In ASIC design environments, latches & FlipFlops are typically predefined cells specified by the ASIC vendor.

All digital designers use the name flip-flop for a sequential device that normally samples its i/ps & changes its o/p only at times determined by a clocking signal.

On the other hand, most digital designers use the name latch for a sequential device that watches all of its i/ps continuously and changes its o/ps at any time, independent of a clocking signal.

The NOR gate S-R latch



logic diagram

S	R	Q_n	Q_{n+1}	State
0	0	0	0	no change
0	0	1	1	
0	1	0	0	Reset (0)
0	1	1	0	
1	0	0	1	set (1)
1	0	1	1	
1	1	0	x	Invalid
1	1	1	x	

Truth table

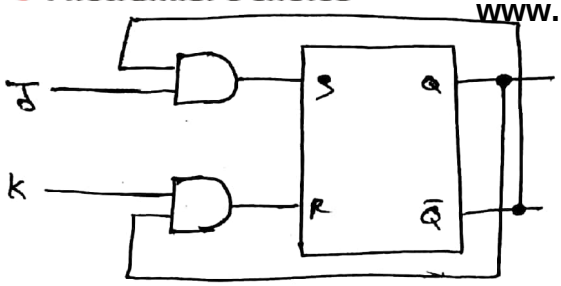
The analysis of the operation of the active HIGH NOR latch can be summarized as follows.

1. SET=0, RESET=0: This is the normal resting state of the NOR latch and it has no effect on the o/p state. Q and $\bar{Q}$ will remain in whatever state they were prior to the occurance of this input condition.

2. SET=1, RESET=0: This will always set Q=1, where it will remain even after SET returns to 0.

3. SET=0, RESET=1: This will always reset Q=0, where it will remain even after RESET returns to 0.

4. SET=1, RESET=1: This condition tries to SET & RESET the latch at the same, & it produces $Q=\bar{Q}=0$; If the i/p are returned to zero simultaneously, the resulting o/p state is erratic & unpredictable. This o/p condition ~~should not be used if it is forbidden (Invalid)~~

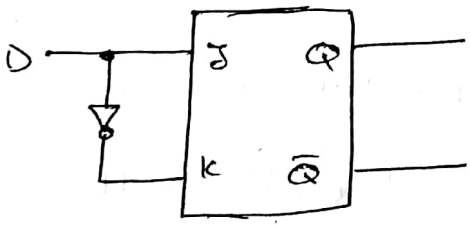


Logic Diagram

J	K	S	R	Qn	Qn+1	State
0	0	0	0	0	0	no change
0	0	0	0	1	1	
0	1	0	0	0	0	Reset (0)
0	1	0	1	1	0	
1	0	1	0	0	1	Set (1)
1	0	0	0	1	1	
1	1	1	0	0	1	Complement ($\bar{Q}_n$)
1	1	0	1	1	0	

Truth Table

D latch

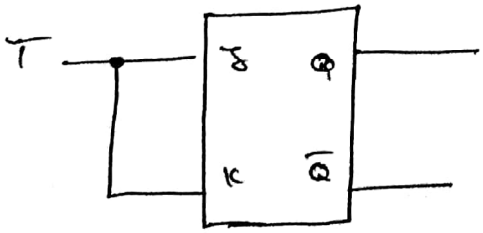


logic diagram

D	Qn	Qn+1
0	0	0
0	1	0
1	0	1
1	1	1

Truth table

T - latch



logic diagram

T	Qn	Qn+1
0	0	0
0	1	1
1	0	1
1	1	0

Truth table

Excitation tables of FlipFlops

SR FlipFlop

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

JK Flip Flop

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

D Flip Flop

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

T - Flip Flop

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

For the design of Sequential circuits we should know the Excitation tables of FlipFlops. The Excitation table of a Flip Flop can be obtained from its truth table. It indicates the inputs required to be applied to the flip-flop to take it from the present state to the next state. The truth tables & Excitation tables of various flip-flops are given above

Flip Flop Conversions

To convert one type of flip-flop into another type, a combinational circuit is designed such that if the inputs of the required flipflop (along with the o/ps of the actual Flip-Flop if required) are fed as i/ps to the combinational circuit and the output of the combinational circuit is connected to the inputs of the actual flip-flop, then the o/p of the actual flip flop is the o/p of the required flip-flop. In other words, it means that, to convert one type of flip-flop into another type, we have to obtain the expressions for the i/ps of the existing flip-flop in terms of the i/ps of the required flip-flop & the present state variable of the existing flip-flop & implement them. The arrangement is shown in fig 1.

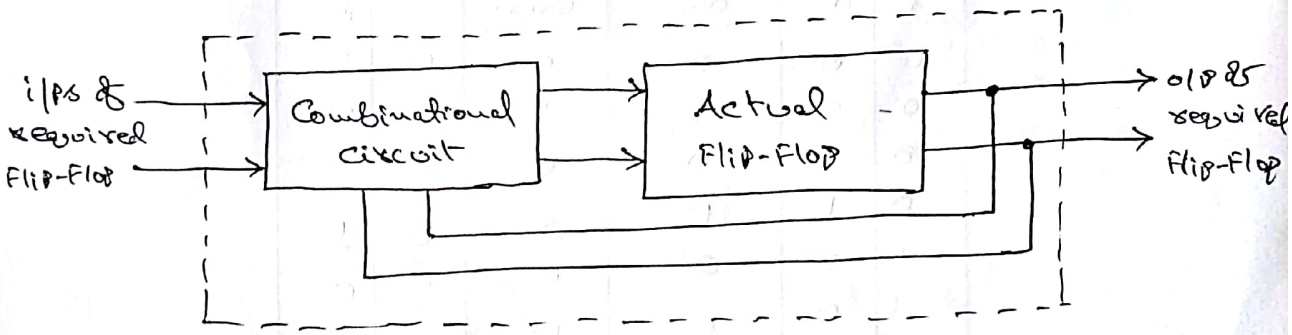


Fig 1 - Block diagram for conversion of Flip-Flop.

S-R FlipFlop to J-K FlipFlop :-

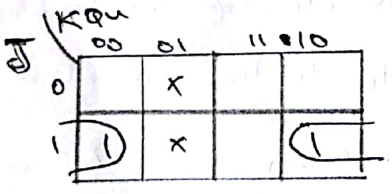
Here the external i/ps to the already available S-R Flip-Flop will be J & K. S and R are the o/ps of the combinational circuit, which are also the actual i/ps to the S-R flip-flop. We write a truth table with J, K, Q_n, Q_{n+1}, S, and R, where Q_n is the present state of the flipflop & Q_{n+1} is the next state obtained when the particular J & K i/p-s are applied, i.e, Q_n denotes the state of the flip-flop before the application of the i/ps & Q_{n+1} refers to the state obtained by the flip-flop after the application of i/ps.

... have eight combinations. For each combination of J, K & Qn, find the corresponding Qn+1, i.e determine to which next state (Qn+1) the JK Flip flop will go from the present state Qn if the present i/ps J & K are applied. Now Complete the table by writing the values of S & R required to get each Qn+1 from the correspondig Qn, ie. write what values of S & R are required to change the state of the flip-flop from Qn to Qn+1.

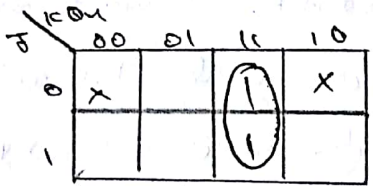
The conversion table, the K-maps for S & R in terms of J, K & Qn and the logic diagram showing the conversion from S-R to J-K are shown in Fig-2

External inputs		Present State	Next State	Flip-Flop inputs	
J	K	Qn	Qn+1	S	R
0	0	0	0	0	x
0	0	1	1	x	0
0	1	0	0	0	x
0	1	1	0	0	1
1	0	0	1	1	0
1	0	1	1	x	0
1	1	0	1	1	0
1	1	1	0	0	1

Conversion table



$S = J\bar{Q_n}$



$R = KQ_n$

K-maps for S & R

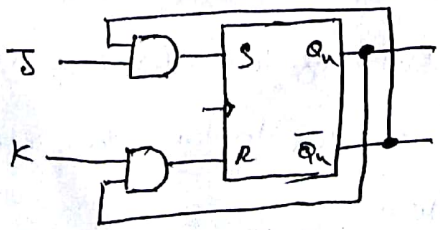


Fig-2 Logic diagram

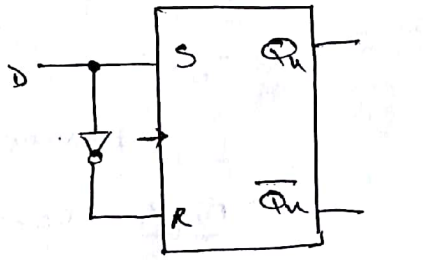
S-R to D flip flop:

Here S-R flip flop is available & we want the operation of the D flipflop from it. So D is the external i/p & the o/ps of the Combinational circuit are the i/ps to the available S-R flip-flop. Express the i/p.s of the existing flip-flop S & R in terms of the External input D & the present state Q_n.

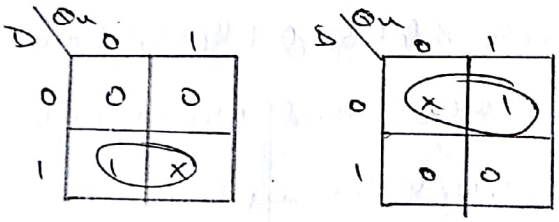
The Conversion table, the K-maps for S & R in terms of D & Q_n, and the logic diagram showing the conversion from S-R to D are shown below figs.

External inputs	Present state	next state	Flipflop inputs.	
D	Q_n	Q_{n+1}	S	R
0	0	0	0	X
0	1	0	0	1
1	0	1	1	0
1	1	1	X	0

Conversion table



logic diagram



K-maps for S & R fig.3

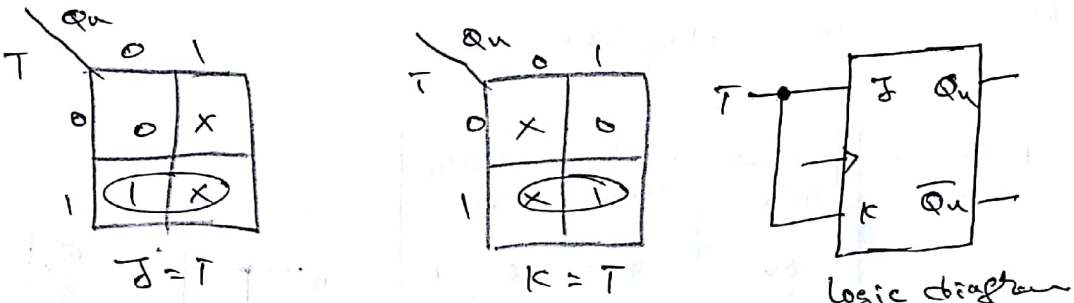
J-K flip flop to T flip-flop:-

Here J-K flipflop is available & we want T flip-flop operation from it. So T is the External i/p & J & K are the actual inputs to the existing Flip-Flop. T & Q_n make four combinations. Express J & K in terms of T and Q_n.

The Conversion table, the K-maps for J & K in terms of T & Q_n, & and the logic diagram showing the conversion from JK to T are shown in fig 4.

External Input	Present State	Next State	Flip-Flop inputs.	
T	Q_n	Q_{n+1}	J	K
0	0	0	0	X
0	1	1	X	0
1	0	1	1	X
1	1	0	X	1

Conversion Table



K-maps for J & K

Fig 4: Conversion of J K flip flop to T flip flop

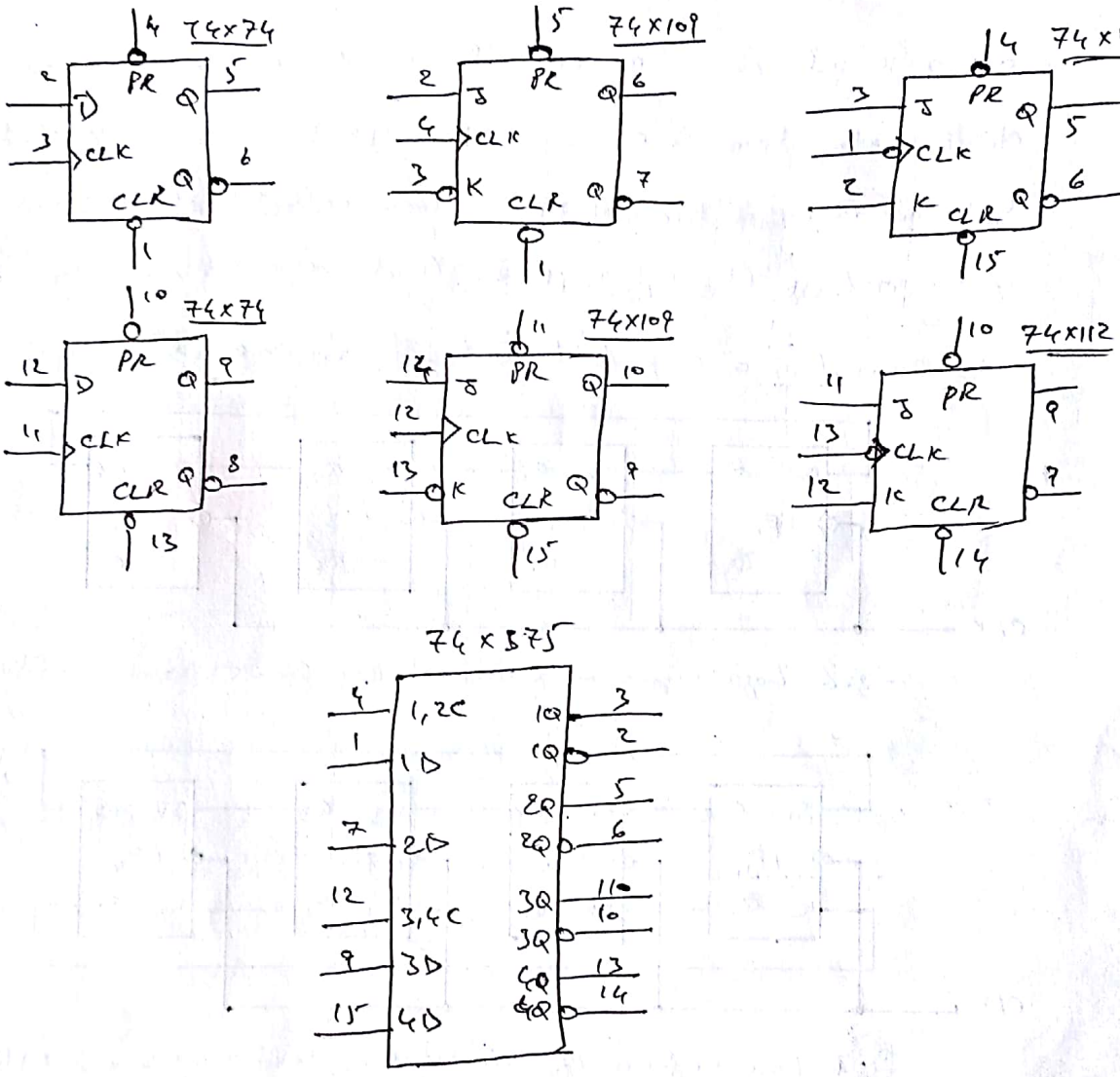
SSI latches & Flipflops:

Several types of discrete latches & Flip Flops are available as SSI parts. SSI latches and Flip Flops have been eliminated to a large extent in modern designs as their functions are embedded in PLDs & FPGAs.

Fig-5 shows the pinouts for several SSI sequential devices. The only latch in the figure is the 74x375, which contains four D latches, similar in function to the "generic" D latches. Because of pin limitations, the latches are arranged in pairs with a common C control line for each pair.

Among the devices in Fig 5 the most important is the 74x7x, which contains two independent positive-edge triggered D-flip flops with preset & clear inputs.

The 74x109 is a positive-edge-triggered J-K flip flop with an active low i/p (named $\overline{K}$ or K-L). Another JK Flip Flop is the 74x112, which has an active-low clock i/p.



Figs: pinouts for SSI latches & flip flops

Counters & Registers

Ring Counter: This is the simplest Shift Register Counter. The basic ring counter using D FFs is shown Fig 6. The realization of this counter using J-K FF's is shown in fig 7. Its state diagram & the sequence table shown in fig 8. Its timing diagram is shown in fig. 9. The flipflops are arranged as in a normal shift register, i.e, the Q o/p of each stage is connected to the D i/p of the next stage, but the Q o/p of the last FF is connected back to the D i/p of the first FlipFlop such that the array of FlipFlops is arranged in a ring & therefore, the name 'Ring Counter'

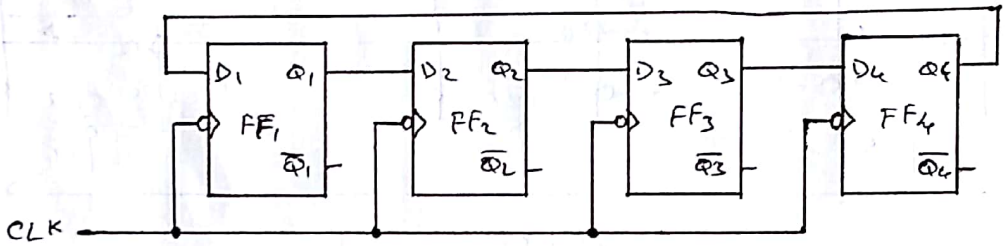


Fig: 6 Logic diagram of a 4-bit Ring Counter using D-flipflops

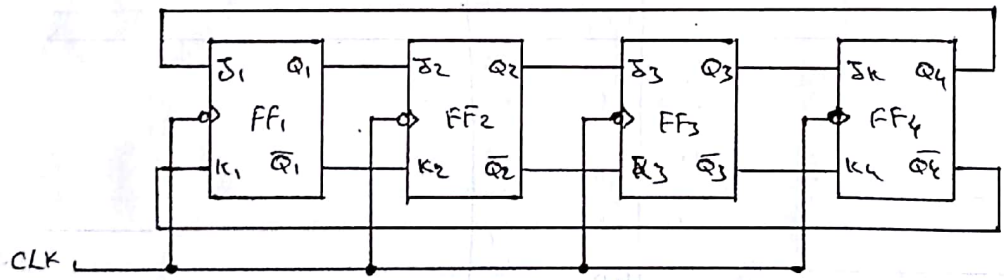
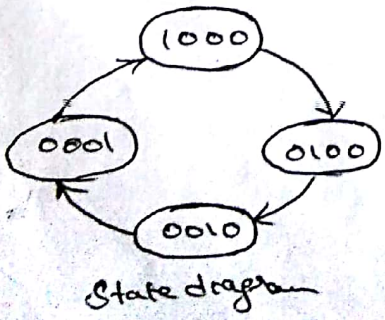


Fig 7: Logic diagram of a 4-bit Ring Counter using J-K flipFlops



Q1	Q2	Q3	Q4	After Clock
1	0	0	0	0
0	1	0	0	1
0	0	1	0	2
0	0	0	1	3
1	0	0	0	4
0	1	0	0	5
0	0	1	0	6
0	0	0	1	7

Sequence Table

Fig 8: State diagram & sequence table

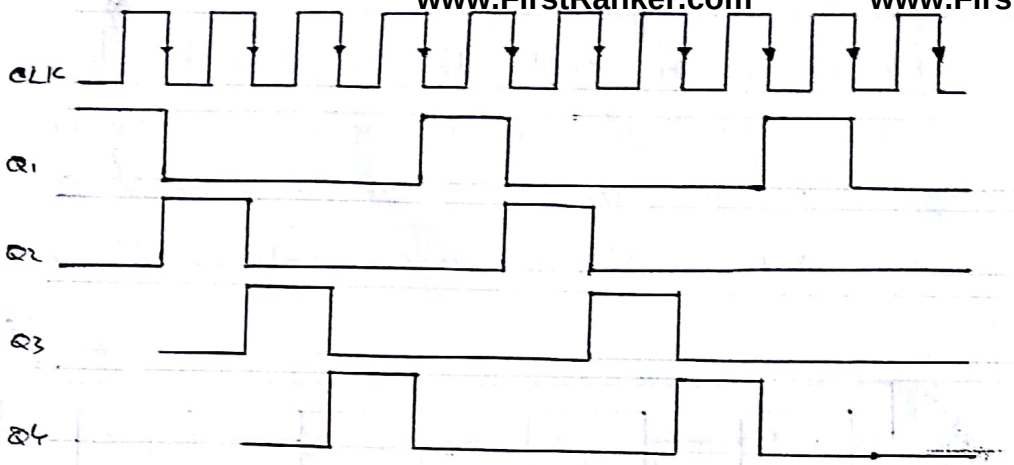


Fig 8: Timing diagram of a 4-bit ring Counter.

In most instances, only a single 1 is in the register and is made to circulate around the register as long as clock pulses are applied. Initially, the first FF is preset to a 1. So, the initial state is 1000, i.e, $Q_1=1$, $Q_2=0$, $Q_3=0$, & $Q_4=0$. After each clock pulse, the contents of the register are shifted to the right by one bit and Q_4 is shifted back to Q_1. The sequence repeats after four clock pulses. The no. of distinct states in the ring Counter, i.e, the MOD of the ring Counter is equal to the no. of FFs used in the Counter. An n-bit ring Counter can count only n bits, where as n-bit ripple Counter can count 2^n bits. So, the ring Counter ~~can count only n bits,~~ is uneconomical compared to a ripple Counter, but has the advantage of requiring no decoder. Since we can read the count by simply noting which FF is set. Since it is entirely a synchronous operation it requires no gates external to FFs, it has the ~~number~~ further advantage of being very fast.

Johnson Counter: (Twisted ring Counter)

This Counter is obtained from a serial-in, serial-out shift register by providing feedback from the inverted output of the last FF to the D input of the first FF. The Q o/p of each stage is connected to the D i/p of the next stage, but the $\bar{Q}$ o/p of the last stage is connected to the D i/p of first stage, therefore, the

twisted ring counter. This feed back arrangement produced a unique sequence of states.

The logic diagram of a 4-bit Johnson Counter using DFFs is shown in fig 10. The realization of the same using JK FFs is shown in fig 11. The state diagram & the sequence table are shown in fig 12. The timing diagram of a Johnson Counter is shown in fig 13.

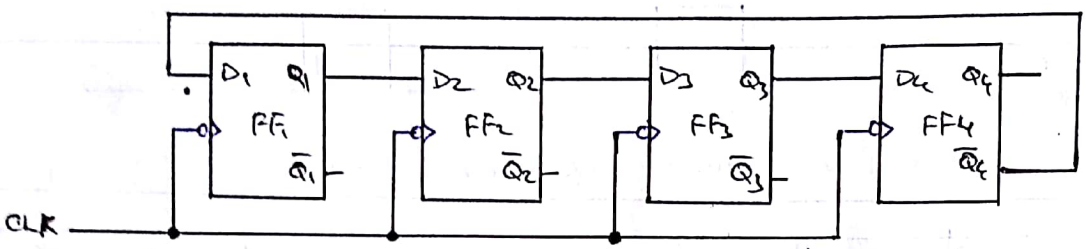


Fig 10: logic diagram of a 4-bit twisted ring counter using D FFs

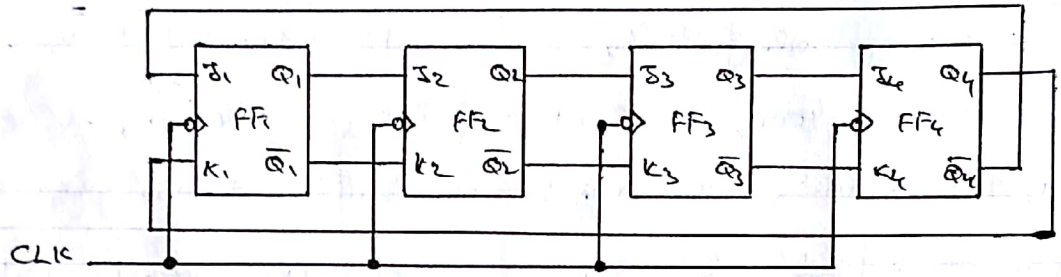
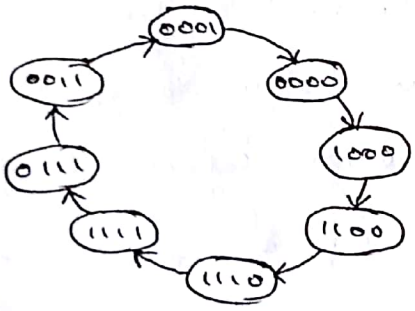


Fig 11: logic diagram of a 4-bit twisted ring counter using JK FFs



(a) State diagram

Q1	Q2	Q3	Q4	After Clock
0	0	0	0	0
1	0	0	0	1
1	1	0	0	2
1	1	1	0	3
1	1	1	1	4
0	1	1	1	5
0	0	1	1	6
0	0	0	1	7
0	0	0	0	8
1	0	0	0	9

(b) Sequence table

Fig 12: State diagram & sequence table of a twisted ring counter

Let initially all the FFs be reset, ie; the state of the counter be 0000. After each clock pulse, the level of Q_1 is shifted to Q_2, the level of Q_2 to Q_3, Q_3 to Q_4 and the level of $\bar{Q}_4$ to Q_1 and the sequence given in fig 12(b) is obtained. This sequence is repeated after every eight clock pulses.

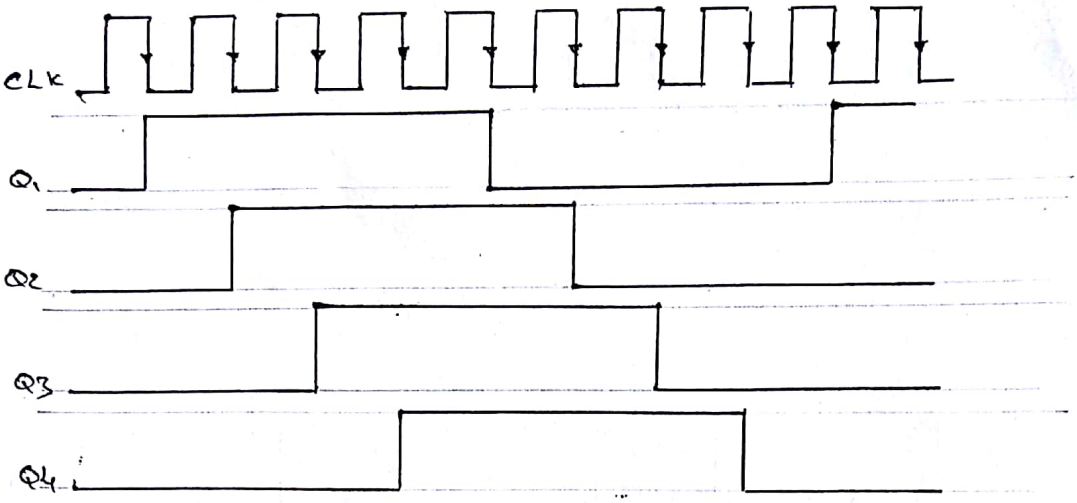


Fig 13: Timing diagram of a 4-bit twisted ring Counter

An n FF Johnson Counter can have 2n unique states & can count up to 2n pulses. So it is a mod-2n counter. It is more economical than the normal ring counter, but less economical than the ripple counter. It required two i/p gates for decoding regardless of the size of the counter. Thus, it requires more decoding circuitry than that by the normal ring counter, but less than that by the ripple counter. It represents a middle ground between the ring counter & the ripple counter.

Basic Sequential logic Design Steps

The procedure for designing Synchronous Sequential circuits can be summarized by a list of recommended steps.

1. From the word description & specifications of the desired operation, derive a state diagram for the circuit.
2. Reduce the number of states if necessary.
3. Assign binary values to the states.
4. Obtain the binary-coded state table.
5. Choose the type of flip-flops to be table.
6. Derive the simplified flip-flop input equations & o/p equations
7. Draw the logic diagram.

DESIGN OF COUNTERS USING DIGITAL ICs

→ IC 7490 (Decade Binary Counter)

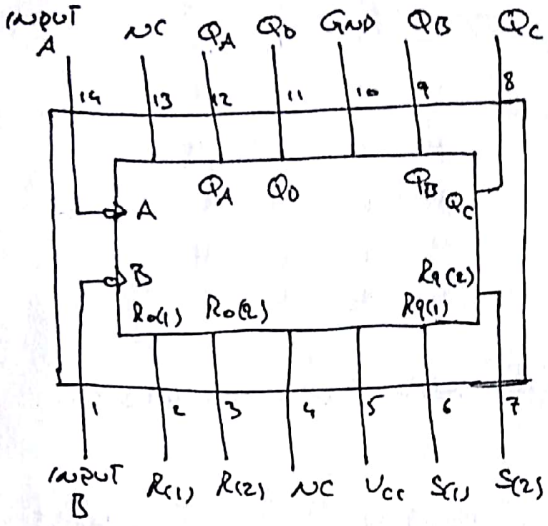


Fig: Connection diagram for 7490

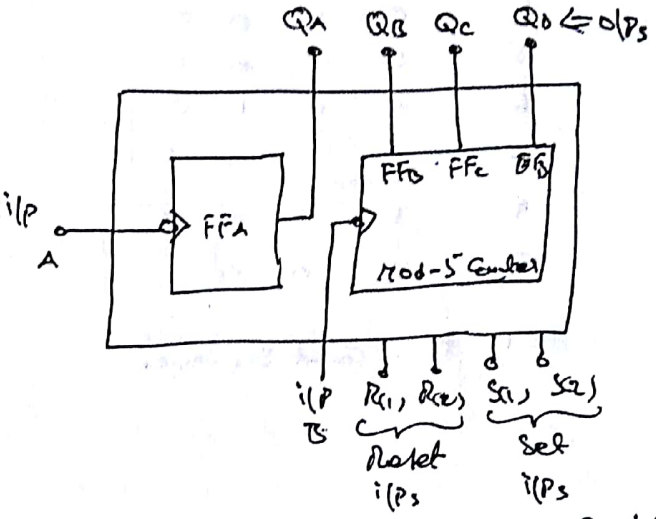


Fig: Basic internal structure of 7490

IC 7490 is a decade binary counter. It consists of four master-slave flip-flops & additional gating to provide a divide by two counter & a three stage binary counter for which the count length is divide-by-five.

Since the o/p from the divide by two section is not internally connected to the succeeding stages, the devices may be operated in various counting modes.

1. BCD Decade (8421) Counter: The 8 i/p must be externally connected to the Q_A o/p & A i/p receives the incoming count.

2. Symmetrical Bi-quinary Divide-by-Ten Counter: The Q_D o/p must be externally connected to the A i/p. The i/p count is then applied to the B i/p & a ~~$\frac{1}{5}$~~ ÷ 10 square wave is obtained at o/p Q_A.

3. ÷ 2 & ÷ 5 Counter: No external interconnecting are required. The first FF is used as a binary element for the ÷ 2 function. The B i/p is used to obtain binary divide by five operation at the Q_D o/p.

Count	OUTPUTS Q_D	Q_C	Q_B	Q_A
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1

Table 1: BCD count sequence

Count	OUTPUTS Q_A	Q_D	Q_C	Q_B
0	L	L	L	L
1	L	L	L	H
2	L	L	H	L
3	L	L	H	H
4	L	H	L	L
5	H	L	L	L
6	H	L	L	H
7	H	L	H	L
8	H	L	H	H
9	H	H	L	L

Table 2: BCD Bi-Quinary (5-2)

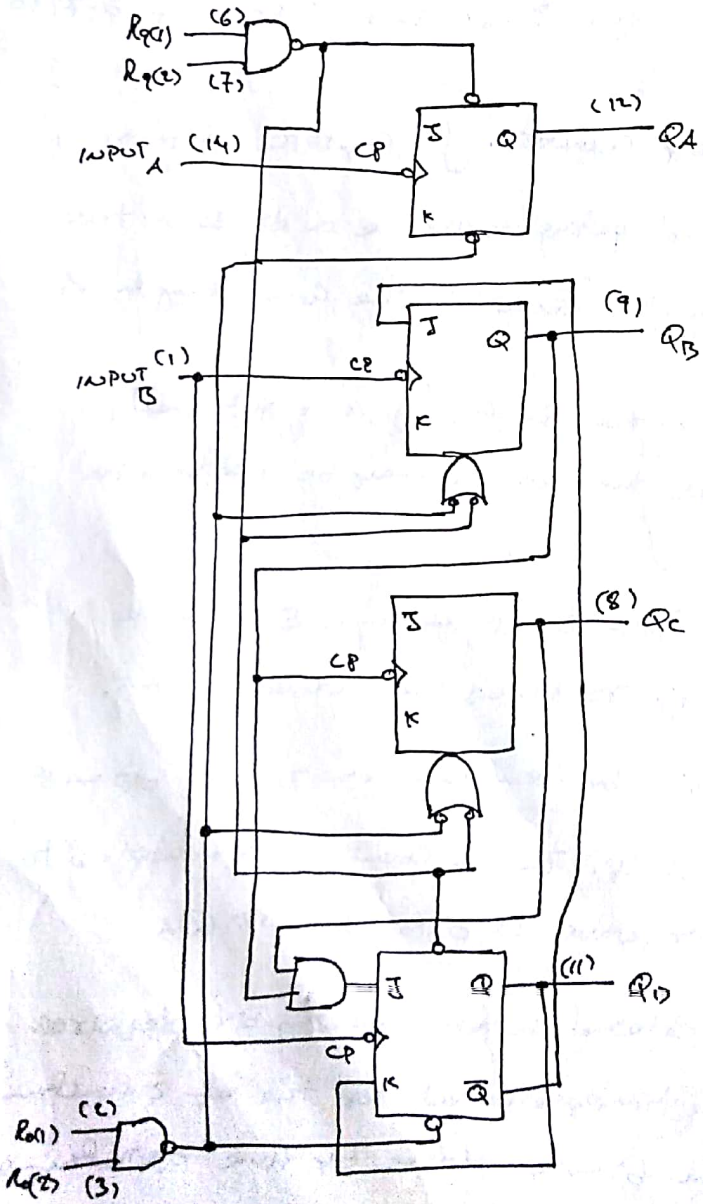
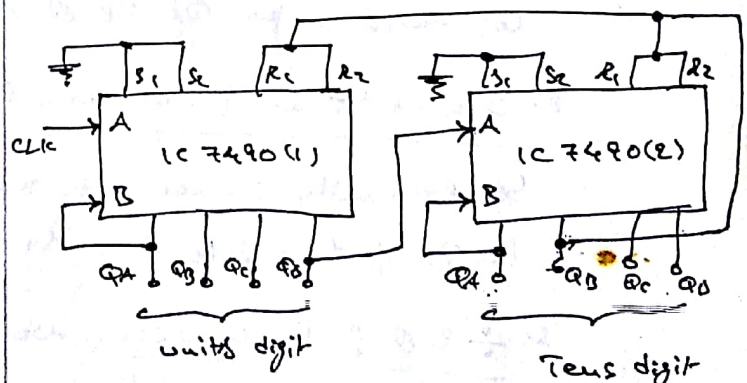


Fig: Logic Diagram for 7490

Reset i/ps $R_{0(1)}$	$R_{0(2)}$	$R_{9(1)}$	$R_{9(2)}$	outputs Q_D	Q_C	Q_B	Q_A
H	H	L	X	L	L	L	L
H	H	X	L	L	L	L	L
X	X	H	H	H	L	L	H
X	L	X	L	COUNT			
L	X	L	X	COUNT			
L	X	X	L	COUNT			
X	L	L	X	COUNT			

Table Reset/count function table

⇒ MOD-20 counter :-



Divide by 20 Counter using IC 7490

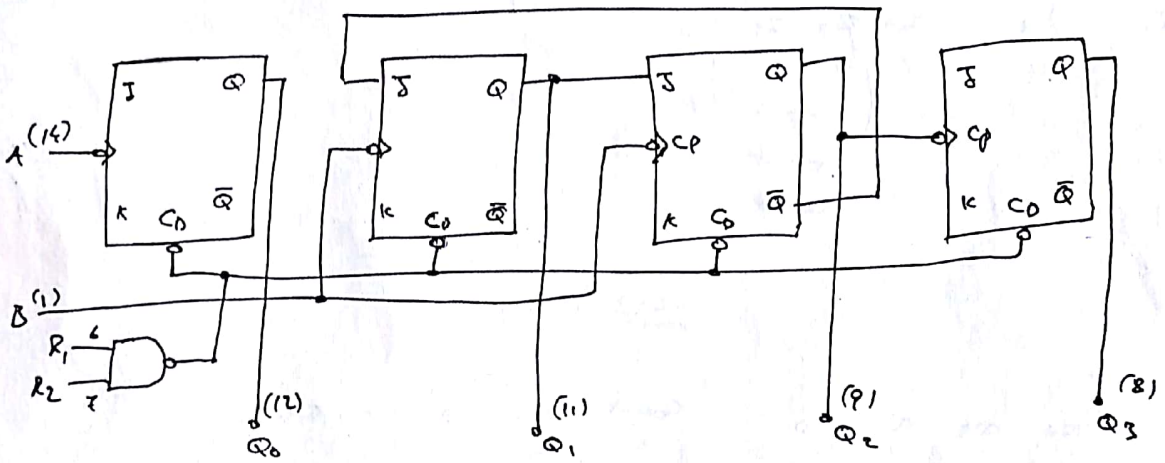
IC 7492/93 (4-bit Ripple Counters)

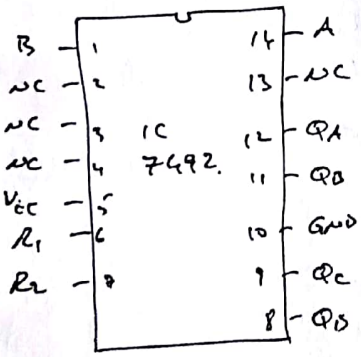
The 7492 & 7493 are high speed 4-bit ripple type counters Partitioned into two sections. Each counter has a divide-by-two section & either a divide-by-six (7492) or divide-by-eight (7493) section which are triggered by a High to low transition on the clock I/Ps. Each section can be used separately & tied together to form divide-by-twelve or divide-by-sixteen counters.

7492 → ÷ 12 counter
7493 → ÷ 16 counter

Each device consists of four master slave Flip flops which are internally connected to Provide a ÷ 2 section.

7492:

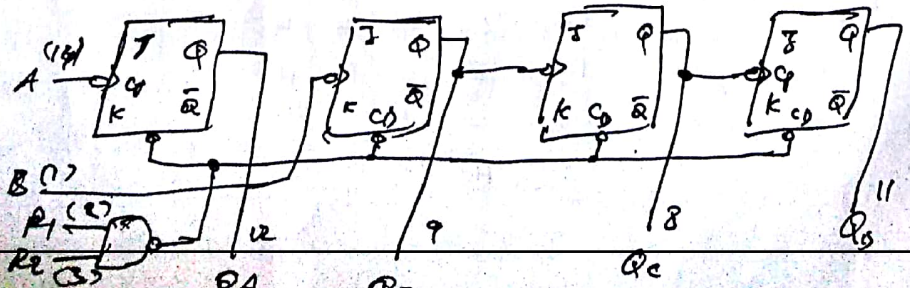




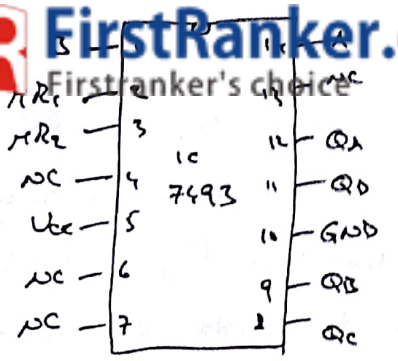
Modes of 7492

1. Mod-12: The B i/p must be Externally connected to the Q_A o/p. The A i/p receives the incoming count & Q_D produces a symmetrical ÷ 12 square wave o/p. i.e. ÷ 2 & ÷ 6. No External interconnections are required. The first FF is used as a binary element for the ÷ 2 Part. The B i/p is used to obtain the ÷ 3 operation that the Q_B & Q_C o/ps. & divide by six operation at the Q_D o/p.

7493



MODE of 7493:-



1. 4-bit ripple counter: The o/p Q_A must be externally connected to i/p B. The i/p count pulses are applied to the i/p A. Simultaneously divisions of 2, 4, 8 & 16 are performed at the Q_A, Q_B, Q_C & Q_D o/p as shown in the truth table.

2. 3-bit Ripple Counter: The i/p count pulses are applied to i/p A. Simultaneously frequency divisions of 2, 4, & 8 are available at the Q_B Q_C & Q_D with respect to the 3-bit ripple through counter.

7492 & 7493

Reset i/p		O/Ps			
R_1	R_2	Q_A	Q_B	Q_C	Q_D
H	H	L	L	L	L
L	H	Count			
H	L	Count			
L	L	Count			

7492

Count	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

7493

Count	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

UNIT-6

Synchronous and Asynchronous Sequential Circuits

6.1 BASIC DESIGN STEPS

The circuit has one input, *w*, and one output, *z*.

All changes in the circuit occur on the positive edge of a clock signal.

The output *z* is equal to 1 if during two immediately preceding clock cycles the input *w* was equal to 1. Otherwise, the value of *z* is equal to 0.

Thus, the circuit detects if two or more consecutive 1s occur on its input *w*. Circuits that detect the occurrence of a particular pattern on its input(s) are referred to as *sequence detectors*.

From this specification it is apparent that the output *z* cannot depend solely on the present value of *w*. To illustrate this, consider the sequence of values of the *w* and *z* signals during 11 clock cycles, as shown in Figure 8.2. The values of *w* are assumed arbitrarily; the values of *z* correspond to our specification. These sequences of input and output values indicate that for a given input value the output may be either 0 or 1. For example, $w = 0$ during clock cycles t_2 and t_5, but $z = 0$ during t_2 and $z = 1$ during t_5. Similarly, $w = 1$ during t_1 and t_8, but $z = 0$ during t_1 and $z = 1$ during t_8. This means that *z* is not determined only by the present value of *w*, so there must exist different states in the circuit that determine the value of *z*.

6.2 STATE DIAGRAM

The first step in designing a finite state machine is to determine how many states are needed and which transitions are possible from one state to another. There is no set procedure for this task. The designer must think carefully about what the machine has to accomplish. A good way to begin is to select one particular state as a *starting* state; this is the state that the circuit should enter when power is first turned on or when a *reset* signal is applied. For our example let us assume that the starting state is called state *A*. As long as the input *w* is 0, the circuit need not do anything, and so each active clock edge should result in the circuit remaining in state *A*. When *w* becomes equal to 1, the machine should recognize this, and move to a different state, which we will call state *B*. This transition takes place on the next active clock edge

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

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

Figure 6.2 Sequences of input and output signals.

state *C* as long as *w* = 1 and should continue to maintain *z* = 1. When *w* becomes 0, the machine should move back to state *A*. Since the preceding description handles all possible values of input *w* that the machine can encounter in its various states, we can conclude that three states are needed to implement the desired machine.

Now that we have determined in an informal way the possible transitions between states, we will describe a more formal procedure that can be used to design the corresponding sequential circuit. Behavior of a sequential circuit can be described in several different ways. The conceptually simplest method is to use a pictorial representation in the form of a *state diagram*, which is a graph that depicts states of the circuit as nodes (circles) and transitions between states as directed arcs. The state diagram in Figure 8.3 defines the behavior that corresponds to our specification. States *A*, *B*, and *C* appear as nodes in the diagram. Node *A* represents the starting state, and it is also the state that the circuit will reach *after* an input *w* = 0 is applied. In this state the output *z* should be 0, which is indicated as *A*/*z*=0 in the node. The circuit should remain in state *A* as long as *w* = 0, which is indicated by an arc with a label *w* = 0 that originates and terminates at this node. The first occurrence of *w* = 1 (following the condition *w* = 0) is recorded by moving from state *A* to state *B*. This transition is indicated on the graph by an arc originating at *A* and terminating at *B*. The label *w* = 1 on this arc denotes the input value that causes the transition. In state *B* the output remains at 0, which is indicated as *B*/*z*=0 in the node.

When the circuit is in state *B*, it will change to state *C* if *w* is still equal to 1 at the next active clock edge. In state *C* the output *z* becomes equal to 1. If *w* stays at 1 during subsequent clock

cycles, the circuit will remain in state *C* maintaining *z* = 1. However, if *w* becomes 0 when the circuit is either in state *B* or in state *C*, the next active clock edge will cause a transition to state *A* to take place.

In the diagram we indicated that the *Reset* input is used to force the circuit into state *A*, which is possible regardless of what state the circuit happens to be in. We could treat

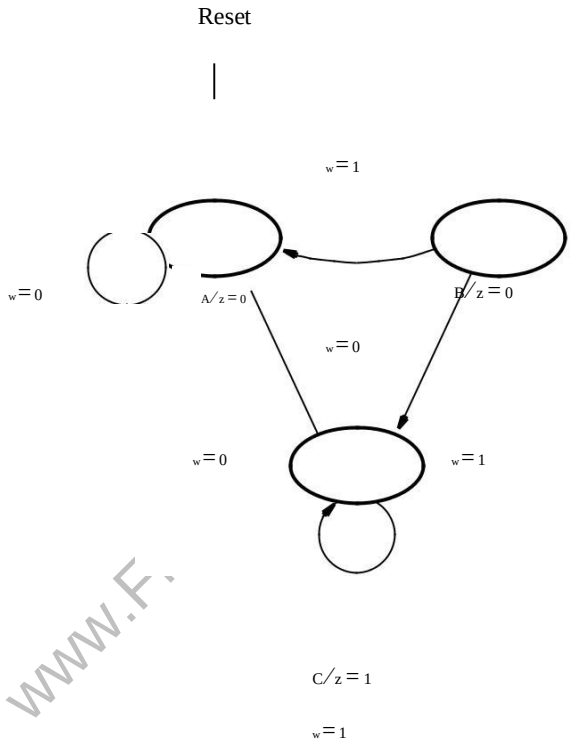


Figure 6.3 State diagram of a simple sequential circuit

Reset as just another input to the circuit, and show a transition from each state to the starting state *A* under control of the input *Reset*. This would complicate the diagram unnecessarily. States in a finite state machine are implemented using flip-flops.

6.3 STATE TABLE

Although the state diagram provides a description of the behavior of a sequential circuit that is easy to understand, to proceed with the implementation of the circuit, it is convenient to translate the information contained in the state diagram into a tabular form. Figure 8.4 shows the *state table* for our sequential circuit. The table indicates all transitions from each *present state* to the *next state* for different values of the input signal. Note that the output z is specified with respect to the present state, namely, the state that the circuit is in at present time. Note also that we did not include the *Reset* input; instead, we made an implicit assumption that the first state in the table is the starting state.

> We now show the design steps that will produce the final circuit. To explain the basic design concepts, we first go through a traditional process of manually performing each design step. This is followed by a discussion of automated design techniques that use modern computer aided design (CAD) tools.

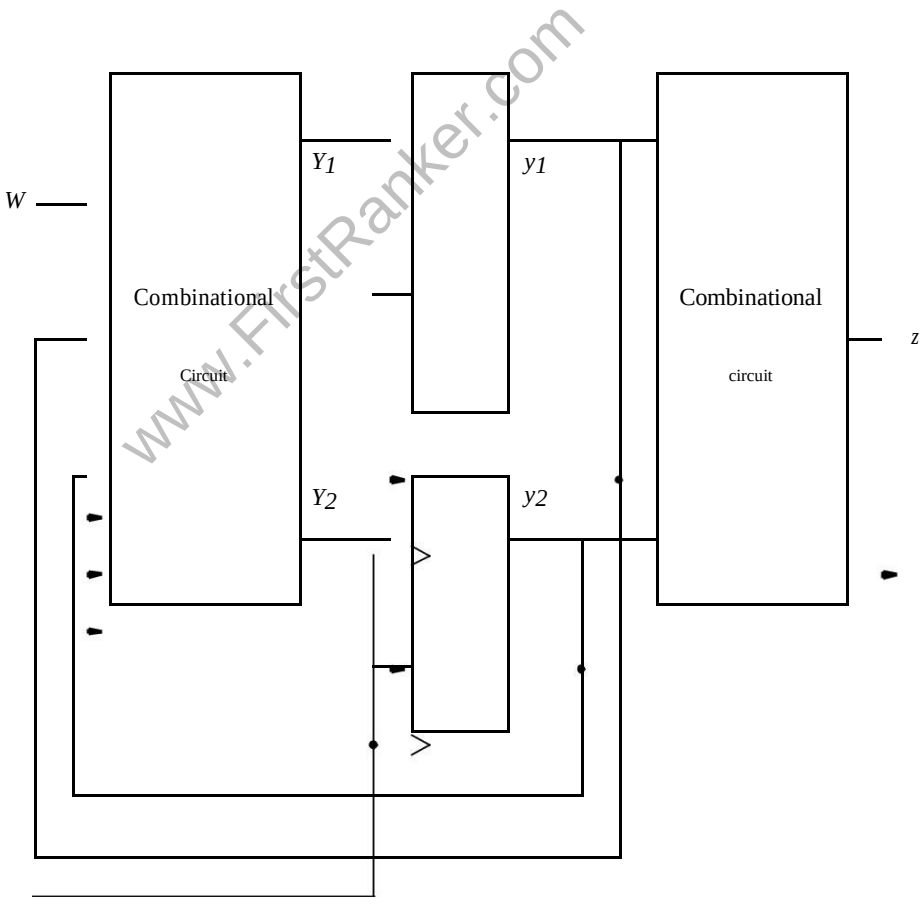
6.4 STATE ASSIGNMENT

The state table in Figure 8.4 defines the three states in terms of letters *A*, *B*, and *C*. When implemented in a logic circuit, each state is represented by a particular valuation (combi-nation of values) of *state variables*. Each state variable may be implemented in the form of a flip-flop. Since three states have to be realized, it is sufficient to use two state variables. Let these variables be y_1 and y_2.

Now we can adapt the general block diagram in Figure to our example as shown in Figure 6.5, to indicate the structure of the circuit that implements the required finite state machine. Two flip-flops represent the state variables. In the figure we have not specified the type of flip-flops to be used; this issue is addressed in the next subsection.

Present state	Next state		Output
	$w = 0$	$w = 1$	Z
A	A	B	0
B	A	C	0
C	A	C	1

Figure 6.4 State table for the sequential circuit



The signals y_1 and y_2 are also fed back to the combinational circuit that determines the next state of the FSM. This circuit also uses the primary input signal w. Its outputs are two signals, Y_1 and Y_2, which are used to set the state of the flip-flops. Each active edge of the clock will cause the flip-flops to change their state to the values of Y_1 and Y_2 at that time. Therefore, Y_1 and Y_2 are called the *next-state variables*, and y_1 and y_2 are called the *present-state variables*. We need to design a combinational circuit with inputs w, y_1, and y_2, such that for all valuations of these inputs the outputs Y_1 and Y_2 will cause the machine to move to the next state that satisfies our specification. The next step in the design process is to create a truth table that defines this circuit, as well as the circuit that generates z.

6.5 CHOICE OF FLIP-FLOPS AND DERIVATION OF NEXT-STATE AND OUTPUT EXPRESSIONS

From the state-assigned table in Figure 8.6, we can derive the logic expressions for the next-state and output functions. But first we have to decide on the type of flip-flops that will be used in the circuit. The most straightforward choice is to use D-type flip-flops, because in this case the values of Y_1 and Y_2 are simply clocked into the flip-flops to become the new values of y_1 and y_2. In other words, if the inputs to the flip-flops are called D_1 and D_2, then these signals are the same as Y_1 and Y_2. Note that the diagram in Figure 8.5 corresponds exactly to this use of D-type flip-flops. For other types of flip-flops, such as JK type, the relationship between the next-state variable and inputs to a flip-flop is not as straightforward; we will consider this situation in section 8.7.

The required logic expressions can be derived as shown in Figure 8.7. We use Karnaugh maps to make it easy for the reader to verify the validity of the expressions. Recall that in Figure 8.6 we needed only three of the four possible binary valuations to represent the states. The fourth valuation, $y_2y_1 = 11$, should never occur in the circuit because the circuit is constrained to move only within states A, B, and C; therefore, we may choose to treat this valuation as a don't-care condition. The resulting don't-care squares in the Karnaugh maps are denoted by d's. Using the don't cares to simplify the expressions, we obtain

$$Y_1 = wy_1y_2$$

$$Y_2 = w(y_1 + y_2)$$

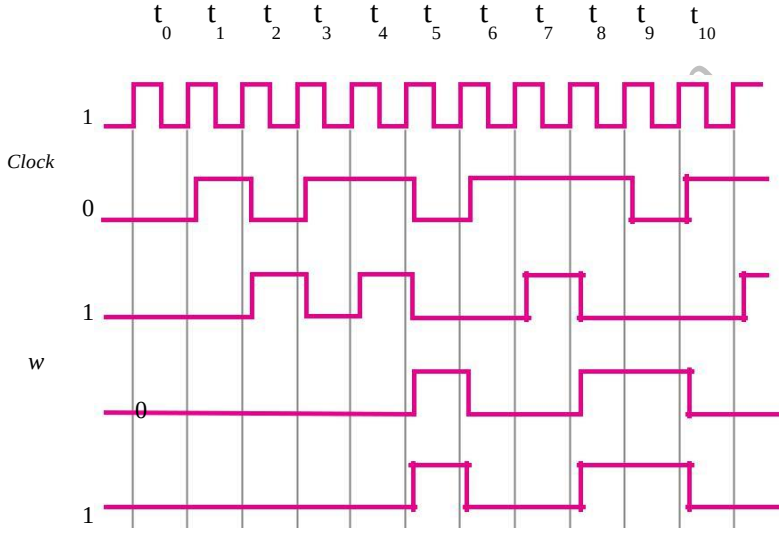
$$z = y_2$$

Since $D_1 = Y_1$ and $D_2 = Y_2$, the logic circuit that corresponds to the preceding expressions is implemented as shown in Figure 8.8. Observe that a clock signal is included, and the circuit is provided with an active-low reset capability. Connecting the clear input on the flip-flops to an external *Resetn* signal, as shown in the figure, provides a simple means

for forcing the circuit into a known state. If we apply the signal *Resetn* = 0 to the circuit, then both flip-flops will be cleared to 0, placing the FSM into the state y_2y_1 = 00.

6.6 TIMING DIAGRAM

we are using positive-edge-triggered flip-flops, all changes in the signals occur shortly after the positive edge of the clock. The amount of delay from the clock edge depends on the propagation delays through the flip-flops. Note that the input signal *w* is also shown to change slightly after the active edge of the clock. This is a good assumption because in a typical digital system an input such as *w* would be just an output of another circuit that is synchronized by the same clock.



8.2 STATE-ASSIGNMENT PROBLEM

The basic concepts involved in the design of sequential circuits, we should revisit some details where alternative choices are possible. In section 6.1 we suggested that some state assignments may be better than others. To illustrate this we can reconsider the example in Figure 8.4. We already know that the state assignment in Figure 6.6 leads to a simple-looking circuit in Figure 8.8. But can the FSM of Figure 6.4 be implemented with an even simpler circuit by using a different state assignment.

In general, circuits are much larger than our example, and different state assignments can have a substantial effect on the cost of the final implementation. While highly desirable, it is often impossible to find the best state assignment for a large circuit. The exhaustive approach of trying all possible state assignments is not practical because the number of available state assignments is huge. CAD tools usually perform the state assignment using heuristic techniques. These techniques are usually proprietary, and their details are seldom published.

6.8 ONE-HOT ENCODING

Another interesting possibility is to use as many state variables as there are states in a sequential circuit. In this method, for each state all but one of the state variables are equal to 0. The variable whose value is 1 is deemed to be "hot." The approach is known as the *one-hot encoding* method.

6.9 VHDL CODE FOR MOORE-TYPE FSMS

VHDL does not define a standard way of describing a finite state machine. Hence while adhering to the required VHDL syntax, there is more than one way to describe a given FSM. An example of VHDL code for the FSM of Figure 8.3 is given in Figure 8.29. For the convenience of discussion, the lines of code are numbered on the left side. Lines 1 to 6 declare an entity named *simple*, which has input ports *Clock*, *Resetn*, and *w*, and output port *z*. In line 7 we have used the name *Behavior* for the architecture body, but of course, any valid VHDL name could be used instead.

The TYPE keyword, which is a feature of VHDL that we have not used previously. The TYPE keyword allows us to create a user-defined signal type. The new signal type is named State_type, and the code specifies that a signal of this type can have three possible values: *A*, *B*, or *C*. Line 9 defines a signal named *y* that is of the State_type type. The *y* signal is used in the architecture body to represent the outputs of the flip-flops that implement the states in the FSM. The code does not specify the number of bits represented by *y*. Instead, it specifies that *y* can have the three symbolic values *A*, *B*, and *C*. This means that we have not specified the number of state flip-flops that should be used for the FSM. As we will see below, the VHDL compiler automatically chooses an appropriate number of state flip-flops when synthesizing a circuit to implement the machine. It also chooses the state assignment for states *A*, *B*, and *C*. Some CAD systems, such as Quartus II, assume that the first state listed in the TYPE statement (line 8) is the reset state for the machine. The state assignment that has all flip-flop outputs equal to 0 is used for this state. Later in this section, we will show

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

ENTITY simple IS
    PORT ( Clock, Resetn,
w                   : IN    STD_LOGIC ;
                            STD_LOGIC )
Z                   : OUT ;
END simple ;

ARCHITECTURE Behavior OF simple IS
    TYPE State_type IS (A, B, C) ;
    SIGNAL y : State_type ;
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            y <= A ;
        ELSIF (Clock'EVENT AND Clock = '1')
THEN
            CASE y IS
                WHEN A =>
                    IF w = '0' THEN
                        y <= A ;
```

```
20                      ELSE
21                          y <  B ;
22                      END IF ;
23                  WHEN B   >
24                      IF w   '0' THEN
25                          y <  A ;
26                      ELSE
27                          y <  C ;
28                      END IF ;
29                  WHEN C   >
30                      IF w   '0' THEN
31                          y <  A ;
32                      ELSE
33                          y <  C ;
34                      END IF ;
35              END CASE ;

            END IF ;
        END PROCESS ;
        z <  '1' WHEN y   C ELSE '0' ;
    END Behavior ;
```


6.9 SPECIFYING THE STATE ASSIGNMENT IN VHDL CODE

That the state assignment may have an impact on the complexity of the designed circuit. An obvious objective of the state-assignment process is to minimize the cost of implementation. The cost function that should be optimized may be simply the number of gates and flip-flops. But it could also be based on other considerations that may be representative of the structure of PLD chips used to implement the design. For example, the CAD software may try to find state

encodings that minimize the total number of AND terms needed in the resulting circuit when the target chip is a CPLD.

In VHDL code it is possible to specify the state assignment that should be used, but there is no standardized way of doing so. Hence while adhering to VHDL syntax, each CAD system permits a slightly different method of specifying the state assignment. The Quartus II system recommends that state assignment be done by using the attribute feature of VHDL. An *attribute* refers to some type of information about an object in VHDL code. All signals automatically have a number of associated *predefined* attributes. An example is the EVENT attribute that we use to specify a clock edge, as in Clock'EVENT.

In addition to the predefined attributes, it is possible to create a user-defined attribute. The *user-defined* attribute can be used to associate some desired type of information with an object in VHDL code. In Quartus II manual state assignment can be done by creating a user-defined attribute associated with the State_type type. This is illustrated in Figure 8.34, which shows the first few lines of the architecture from Figure 8.33 with the addition of a user-defined attribute. We first define the new attribute called ENUM_ENCODING, which has the type STRING. The next line associates ENUM_ENCODING with the State_type type and specifies that the attribute has the value "00 01 11". When translating the VHDL code, the Quartus II compiler uses the value of ENUM_ENCODING to make the state assignment $A = 00$, $B = 01$, and $C = 11$.

```
ARCHITECTURE Behavior OF simple IS

    TYPE State TYPE IS (A, B, C) ;

    ATTRIBUTE ENUM ENCODING          : STRING ;

    ATTRIBUTE ENUM ENCODING OF State type : TYPE IS "00 01 11" ;

    SIGNAL y present, y next         : State type ;

BEGIN

    .
    .
    .
```

Figure 8.34 A user-defined attribute for manual state assignment.

```
LIBRARY ieee ;

USE ieee.std logic 1164.all
;

ENTITY simple IS

    PORT ( Clock, Resetn,
    w                       : INSTD LOGIC ;
                            : OUT STD LOGIC )
           Z                ;

END simple ;



ARCHITECTURE Behavior OF simple IS

    SIGNAL y presentz, y next : STD LOGIC VECTOR(1 DOWNTO
    0);

    CONSTANT A : STD LOGIC VECTOR(1 DOWNTO 0) :     ”00” ;

    CONSTANT B : STD LOGIC VECTOR(1 DOWNTO 0) :     ”01” ;

    CONSTANT C : STD LOGIC VECTOR(1 DOWNTO 0) :     ”11” ;

BEGIN

    PROCESS ( w, y present )

    BEGIN

        CASE y present IS

            WHEN A  >

                IF w   ’0’ THEN y next <  A ;
```

```
                                        -
                ELSE y next <  B ;
                        -
                END IF ;
            WHEN B  >

                IF w    '0' THEN y next <  A ;
                                        -
                ELSE y next <  C ;
                        -
                END IF ;
            WHEN C  >

                IF w    '0' THEN y next <  A ;
                                        -
                ELSE y next <  C ;
                        -
                END IF ;
            WHEN OTHERS >

                    next <  A ;
        END CASE ;

    END PROCESS ;

    PROCESS ( Clock, Resetn )

    BEGIN

IF Resetn                               '0'
                                        THEN

    y present <                         A ;
                -

                ELSIF (Clock'EVENT AND Clock        '1') THEN
```

```
            y present <   y next ;
             -              -

        END IF ;

    END PROCESS ;

    z <    '1' WHEN y present  C ELSE '0' ;
                        -

END Behavior ;
```

6.10 SPECIFICATION OF MEALY FSMS USING VHDL

A Mealy-type FSM can be specified in a similar manner as a Moore-type FSM. Figure 8.36 gives complete VHDL code for the FSM in Figure 8.23. The state transitions are described in the same way as in our original VHDL example in Figure 8.29. The signal *y* represents the state flip-flops, and State_type specifies that *y* can have the values *A* and *B*. Compared to the code in Figure 8.29, the major difference in the case of a Mealy-type FSM is the way in which the code for the output is written. In Figure 8.36 the output *z* is defined using a CASE statement. It states that when the FSM is in state *A*, *z* should be 0, but when in state *B*, *z* should take the value of *w*. This CASE statement properly describes the logic needed for *z*, but it may not be obvious why we have used a second CASE statement in the code, rather than specify the value of *z* inside the CASE statement that defines the state transitions. The reason is that the CASE statement for the state transitions is nested inside the IF statement that waits for a clock edge to occur. Hence if we placed the code for *z* inside this CASE statement, then the value of *z* could change only as a result of a clock edge. This does not meet the requirements of the Mealy-type FSM, because the value of *z* must depend not only on the state of the machine but also on the input *w*.

Implementing the FSM specified in Figure 8.36 in a CPLD chip yields the same equa-tions as we derived manually in section 8.3. Simulation results for the synthesized circuit appear in Figure 8.37. The input waveform for *w* is the same as the one we used for the Moore-type machine in Figure 8.32. Our Mealy-type machine behaves correctly, with *z* becoming 1 just after the start of the second consecutive clock cycle in which *w* is 1.

In the simulation results we have given in this section, all changes in the input *w* occur immediately following a positive clock edge. This is based on the assumption stated in section 8.1.5 that in a real circuit *w* would be synchronized with respect to the clock that controls the FSM. In Figure 8.38 we illustrate a problem that may arise if *w* does not meet this specification. In this case we have assumed that the changes in *w* take place at the

```
LIBRARY ieee ;

        USE ieee.std logic 1164.all
        ;

        ENTITY mealy IS

            PORT ( Clock, Resetn,
            w                   : INSTD LOGIC ;
                                : OUT STD LOGIC )
                  Z             ;

        END mealy ;


        ARCHITECTURE Behavior OF mealy IS

            TYPE State type IS (A, B) ;

            SIGNAL y : State type ;

        BEGIN

            PROCESS ( Resetn, Clock )

            BEGIN

                IF Resetn   '0' THEN

                    y <   A ;

                ELSIF (Clock'EVENT AND Clock '1')
                    THEN CASE y IS


                        WHEN A  >

                            IF w   '0' THEN y <  A ;
```

```
                ELSE y <  B ;
                END IF ;
            WHEN B  >
                IF w   '0' THEN y <  A ;
                ELSE y <  B ;
                END IF ;
        END CASE ;
    END IF ;
END PROCESS ;

PROCESS ( y, w )
BEGIN
    CASE y IS
        WHEN A  >
            z <   '0' ;
WHEN B                  >
z <                     w ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```