DESIGN - I

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

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.

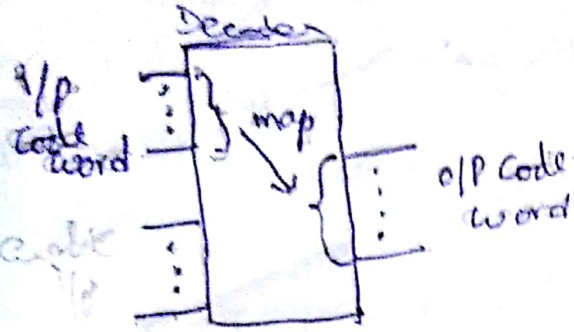
• DECODER

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.

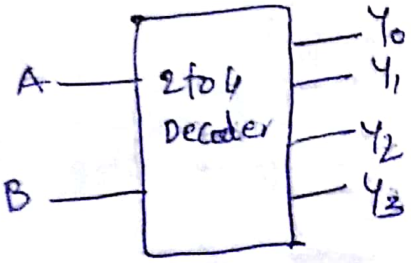
General structure of decoder.



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

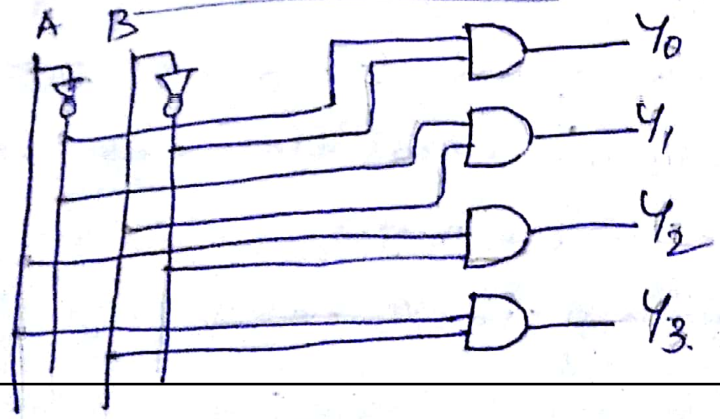


logic - symbol

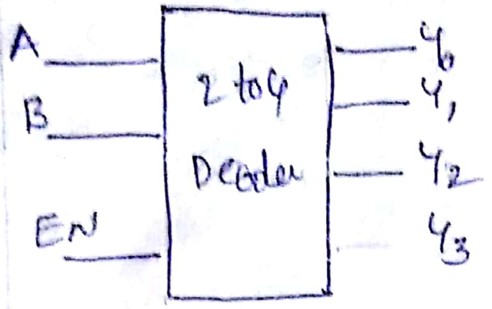
A	B	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Truth table

Internal Structure



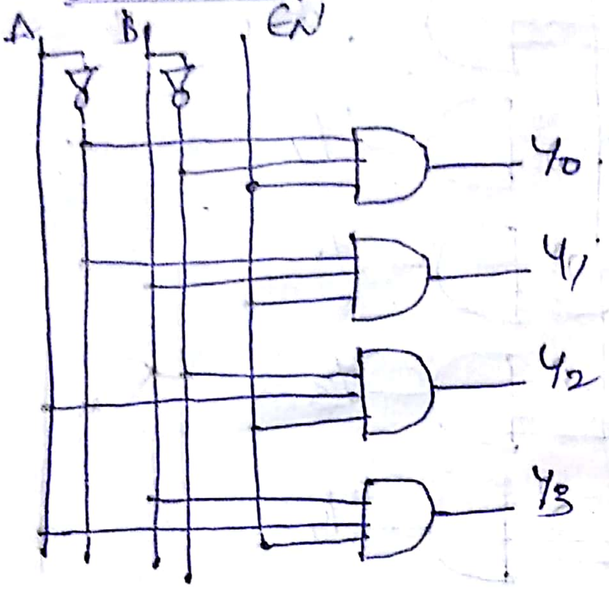
2 to 4 decoder with Enable i/p



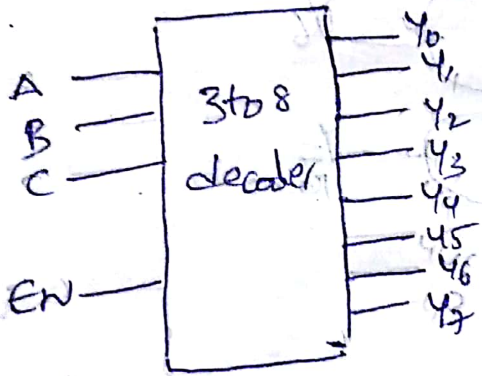
Truth table

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

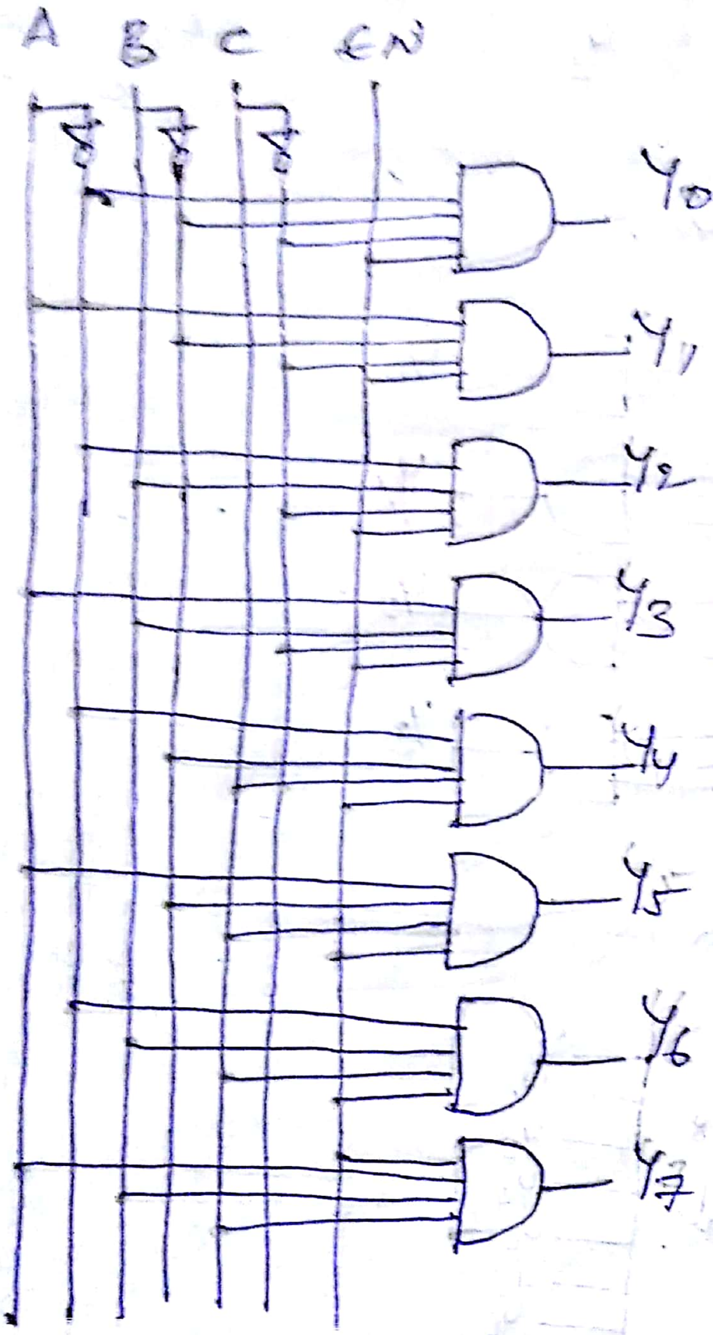


3 to 8 decoder with Enable i/p

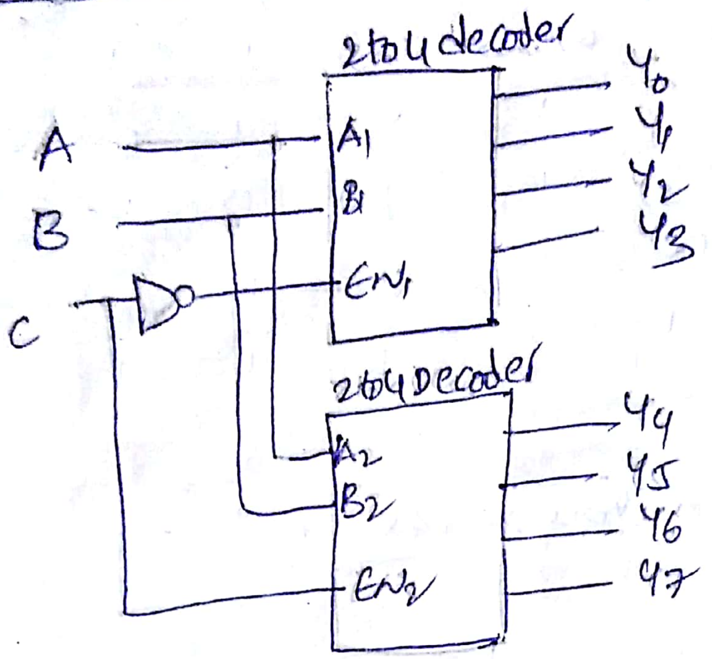


EN	A	B	C	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	x	x	x	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0

1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0



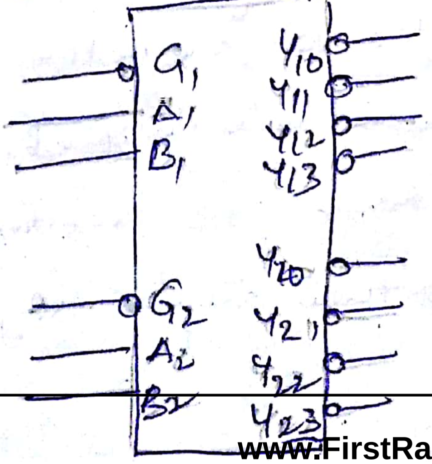
Design 3 to 8 decoder using 2 to 4 decoder



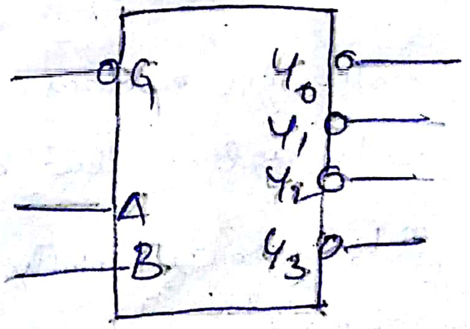
C	B	A	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

MSI DECODER

74x139 Dual 2 to 4 Decoder

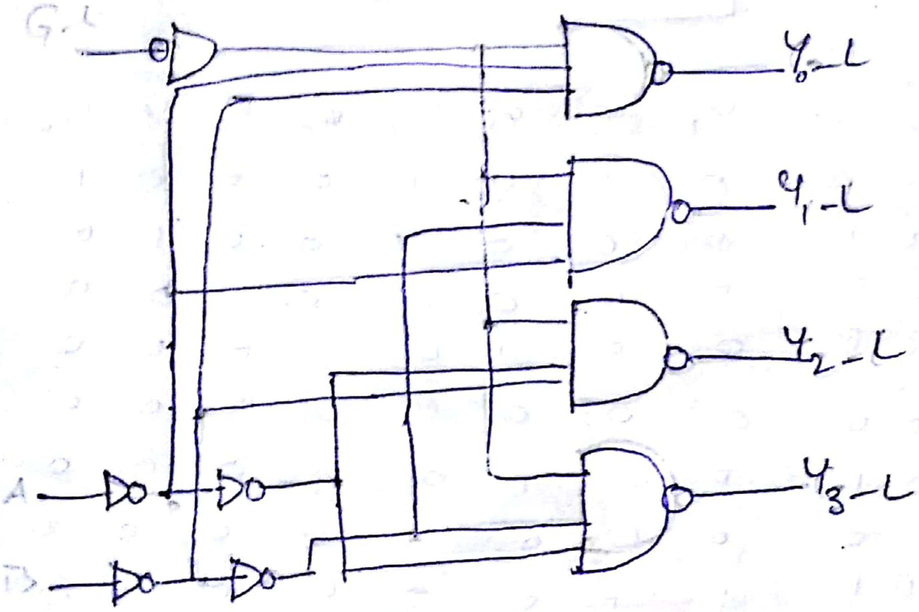


1/2 74x139



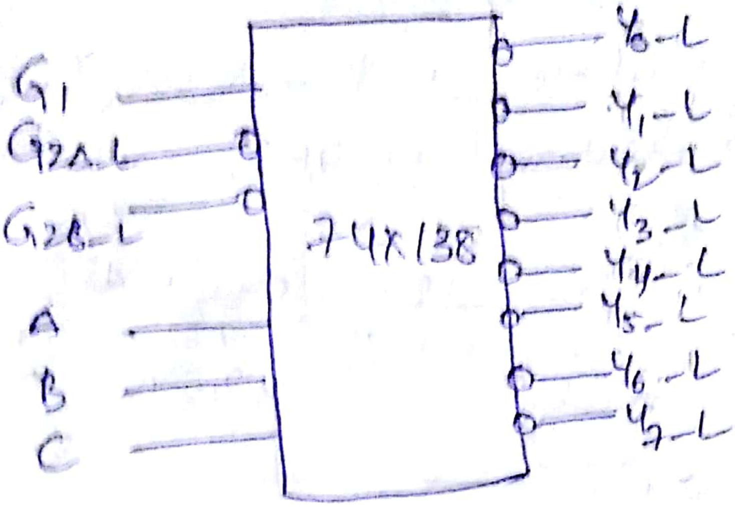
G-L	A	B	Y_3-L	Y_2-L	Y_1-L	Y_0-L
1	x	x	1	1	1	1
0	0	0	1	1	1	0
0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	1

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 decoders were originally designed with active-low o/ps. Since TTL inverting gates are generally faster than noninverting ones.



G1	G2A-L	G2B-L	A	B	C	Y7-L	Y6-L	Y5-L	Y4-L	Y3-L	Y2-L	Y1-L	Y0-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 (G1, G2A-L, G2B-L). all of which must be asserted for the selected o/p to be asserted.

The logic function of 138's is straight forward - an o/p is asserted if & only if [illegible] is enabled & the o/p is selected

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

$$Y_5 = \underbrace{G_1 \cdot G2A \cdot G2B}_{\text{enable.}} \cdot \underbrace{C \cdot \overline{B} \cdot A}_{\text{select}}$$

$$G2A = \overline{G2A\_L}$$

$$G2B = \overline{G2B\_L}$$

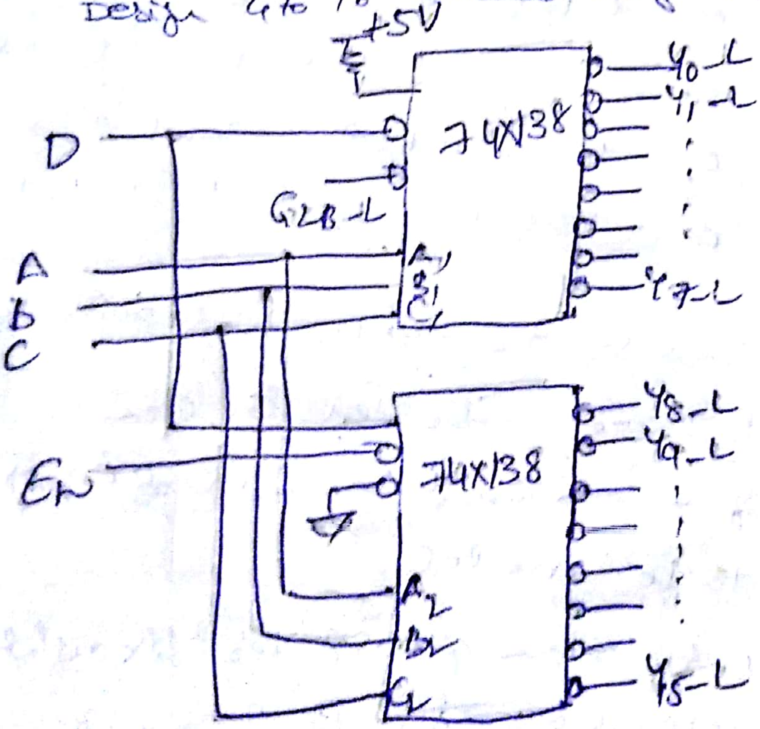
$$Y_5 = \overline{Y_{5}\_L}$$

$$\overline{Y_5\_L} = \overline{Y_5} = \overline{(G_1 \cdot \overline{G2A\_L} \cdot \overline{G2B\_L} \cdot C \cdot \overline{B} \cdot A)}$$

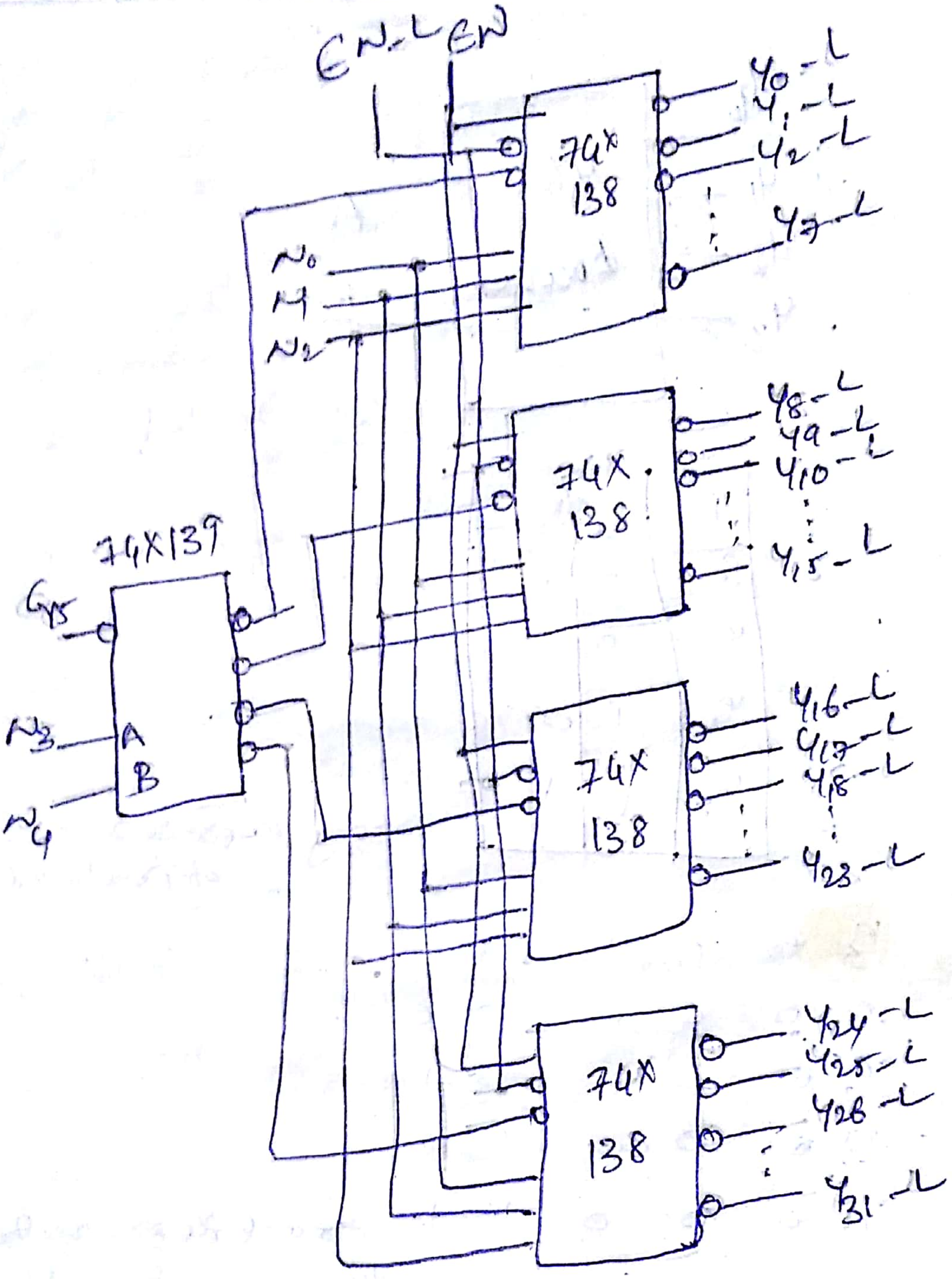
$$= \overline{G_1} + G2A\_L + G2B\_L + \overline{C} + B + \overline{A}$$

· CASCADING BINARY DECODERS

Design 4 to 16 Decoder using 74x138 decoder



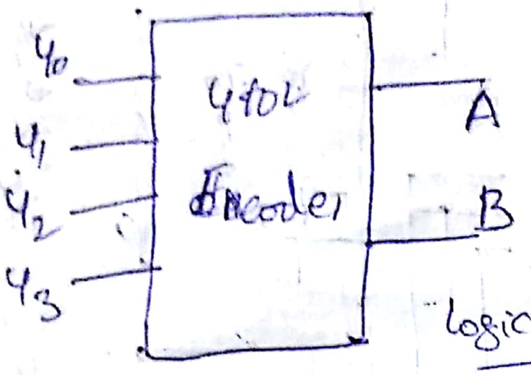
Design 5 to 32 decoder using 74x139 & 74x138



ENCODERS

It is a combinational logic circuit. It has 2^n i/ps and n o/ps. It performs Exact opposite function of decoder. It performs Some Coding operations.

Design 4 to 2 Encoder



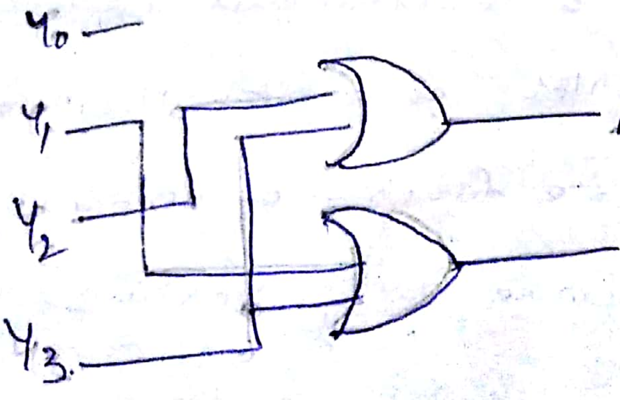
logic Symbol

I/Ps	O/Ps	
	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

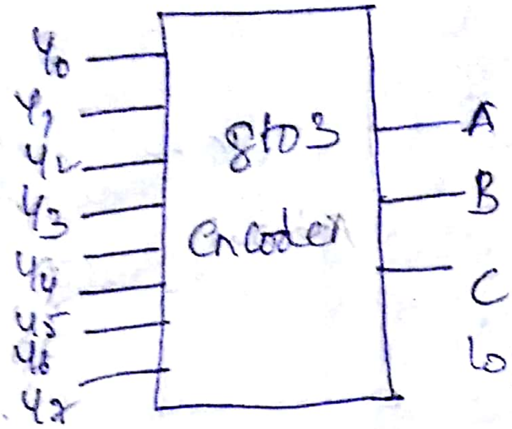
Exact Representation of Truth table.



$A = Y_2 + Y_3$

$B = Y_1 + Y_3$

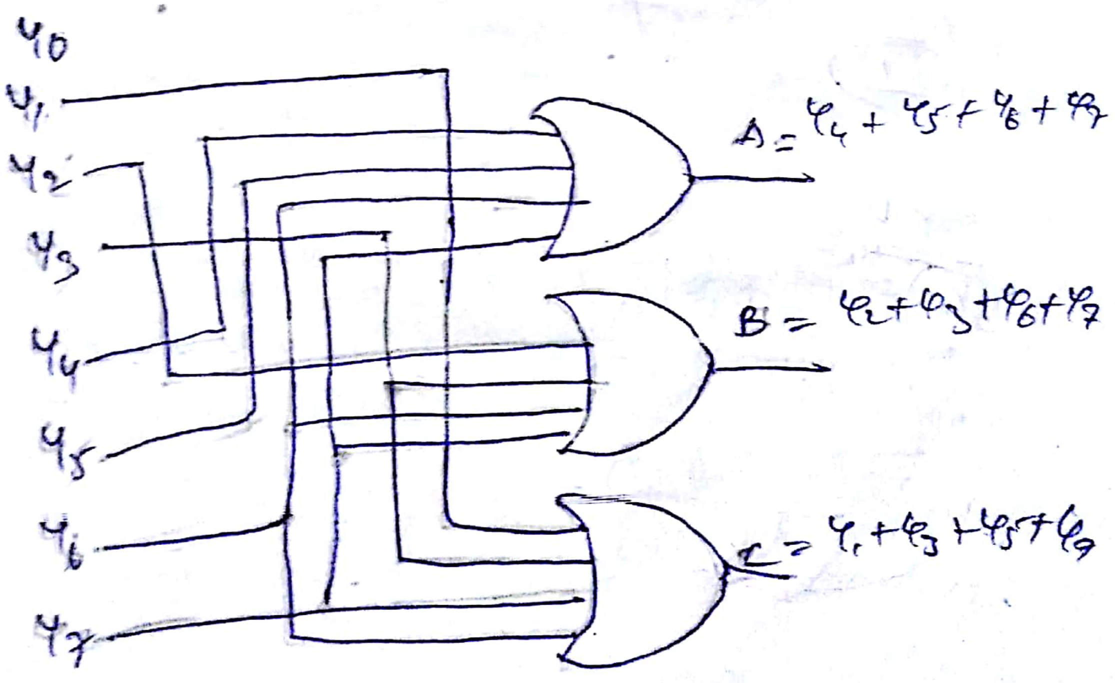
encoder



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

Y_3	Y_2	Y_1	Y_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

Y_0 -> highest priority

V is valid i/p

For A

Y_3Y_2 \ Y_1Y_0	00	01	11	10
00				
01	1			
11	1			
10	1			

$$A = \bar{Y_1}\,\bar{Y_0}\,Y_3 + \bar{Y_1}\,\bar{Y_0}\,\bar{Y_2}$$

For B

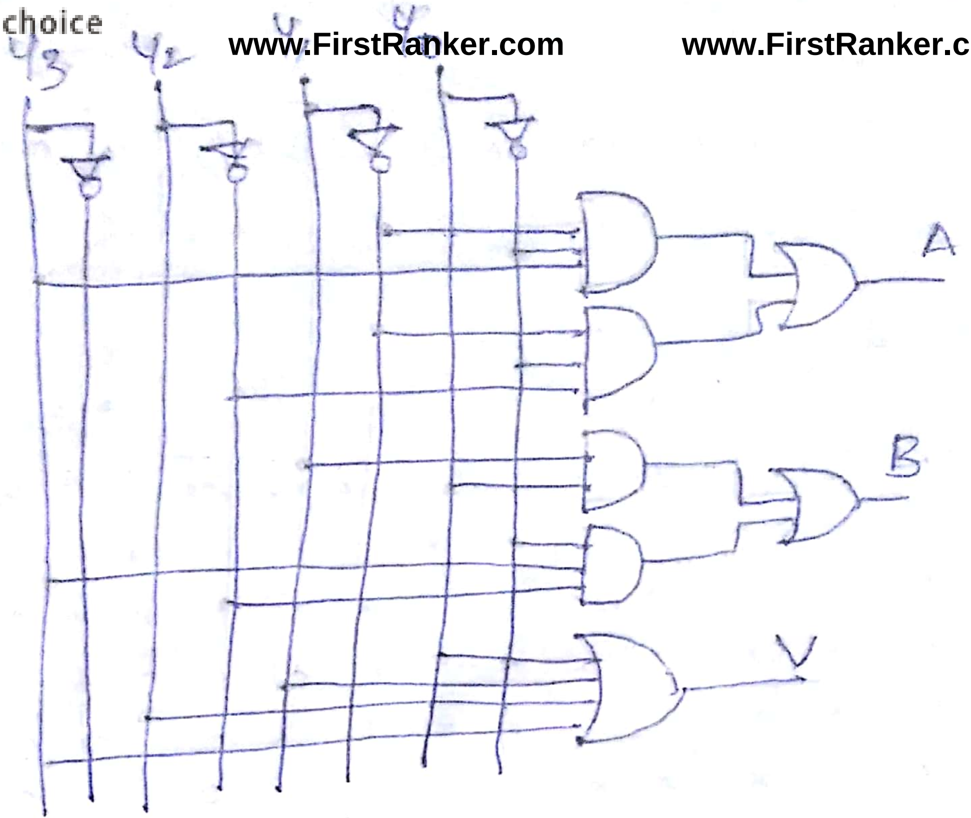
Y_3Y_2 \ Y_1Y_0	00	01	11	10
00				1
01				1
11				1
10	1			

$$B = Y_1 Y_0 + \bar{Y_0}\,Y_3\,\bar{Y_2}$$

For V

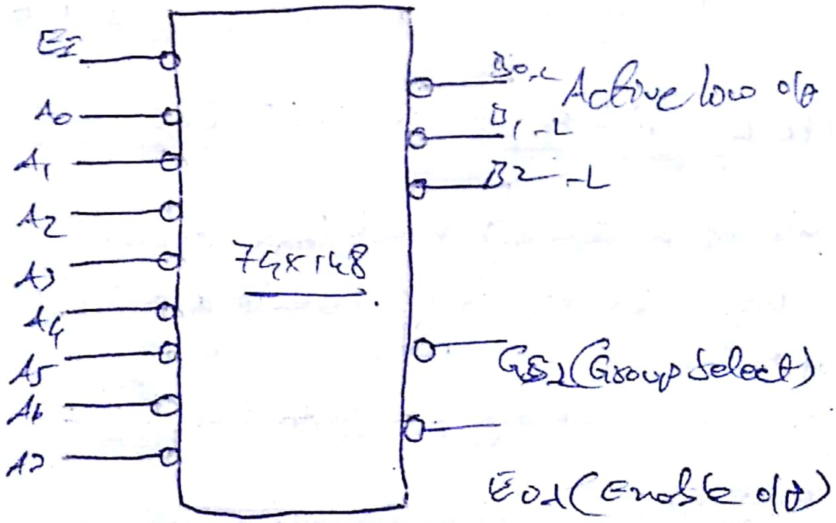
Y_3Y_2 \ Y_1Y_0	00	01	11	10
00		1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	1	1	1

$$V = Y_0 + Y_1 + Y_2 + Y_3$$



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 & 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 if EIL is asserted but no request i/p is asserted. Thus a lower priority 148 may be enabled.

Truth Table

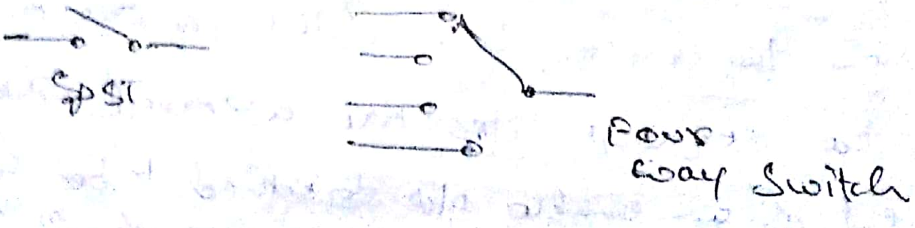
i/ps									o/ps				
EI_L	I_{0_L}	I_{1_L}	I_2	I_3	I_4	I_5	I_6	I_7	A_2	A_1	A_0	GS	E_0
1	x	x	x	x	x	x	x	x	1	1	1	1	1
0	x	x	x	x	x	x	x	0	0	0	0	0	1
0	x	x	x	x	x	x	0	1	0	0	1	0	1
0	x	x	x	x	x	0	1	1	0	1	0	0	1
0	x	x	x	x	0	1	1	1	0	1	1	0	1
0	x	x	x	0	1	1	1	1	1	0	0	0	1
0	x	x	0	1	1	1	1	1	1	0	1	0	1
0	x	0	1	1	1	1	1	1	1	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.

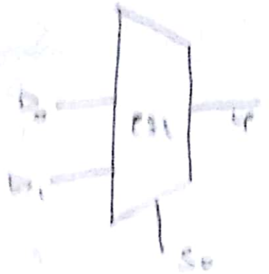


Design 2:1 Mux

2 → data i/ps

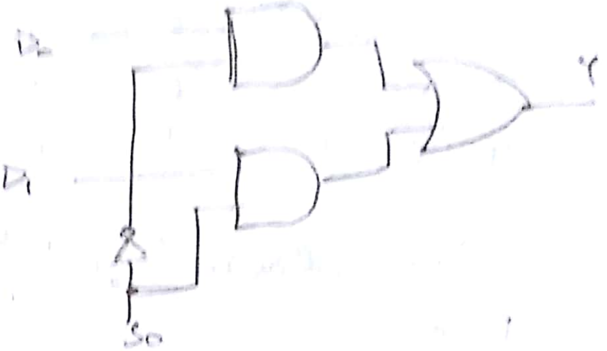
1 → selection i/p

1 → o/p.



Truth Table

S0	Y
0	D0
1	D1

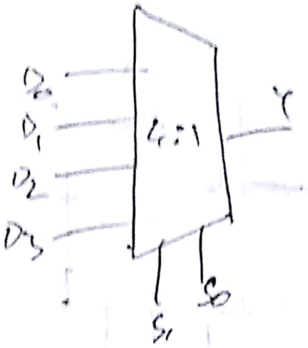


Design 4:1 Mux

4-data i/ps

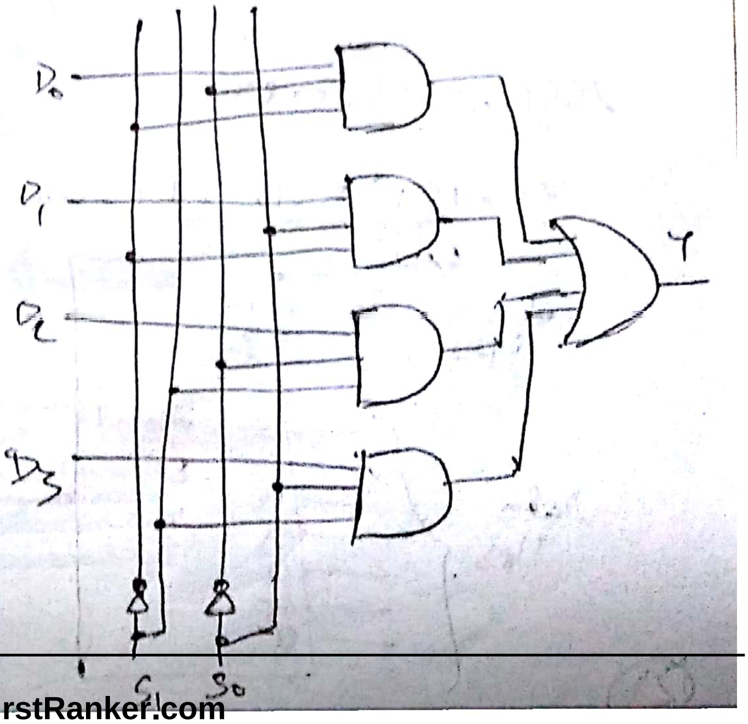
2-selection i/ps

01- o/p.



Truth Table

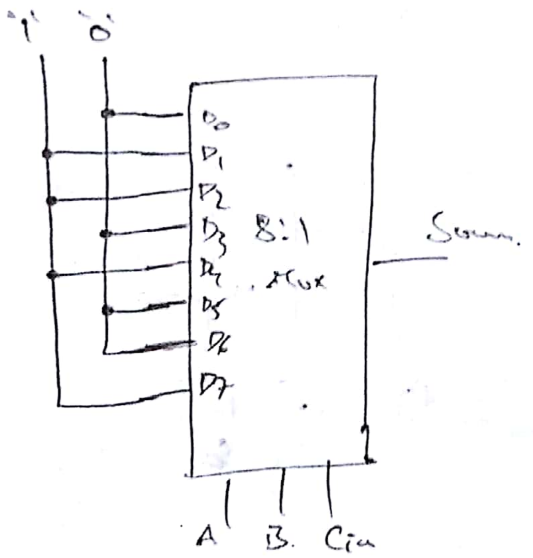
S1	S0	Y
0	0	D0
0	1	D1
1	0	D2
1	1	D3



Realize Fulladder [illegible]

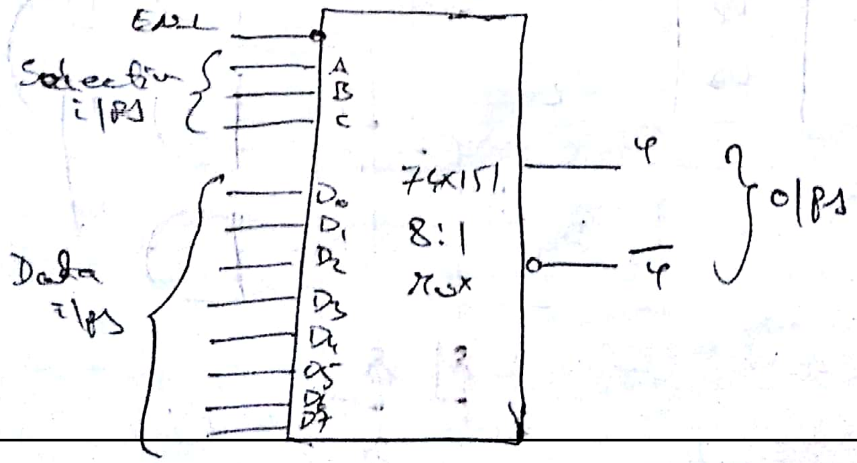
A	B	Cin	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

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



MSI Multiplexers

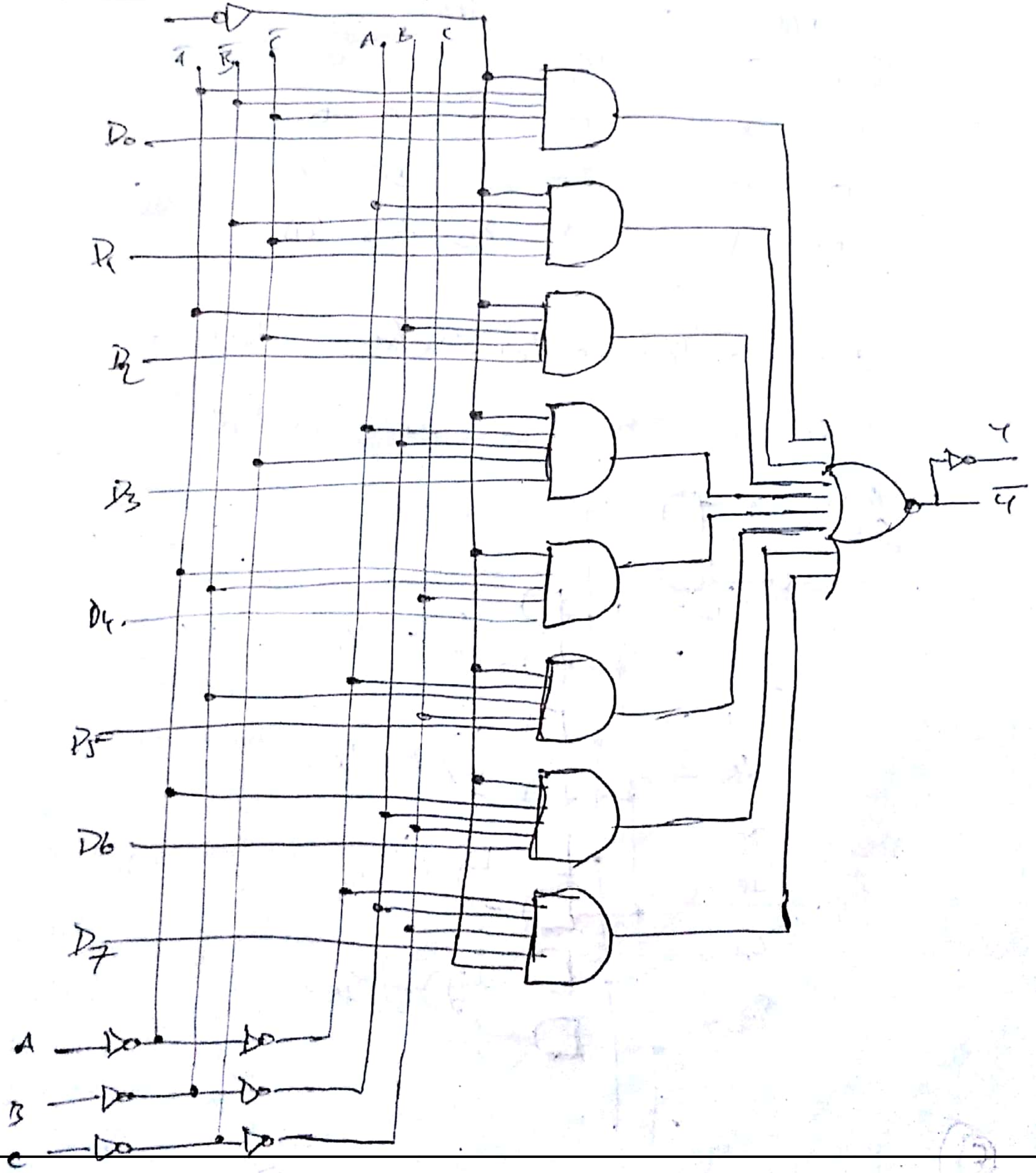
74x151 8:1P 1 bit Mux



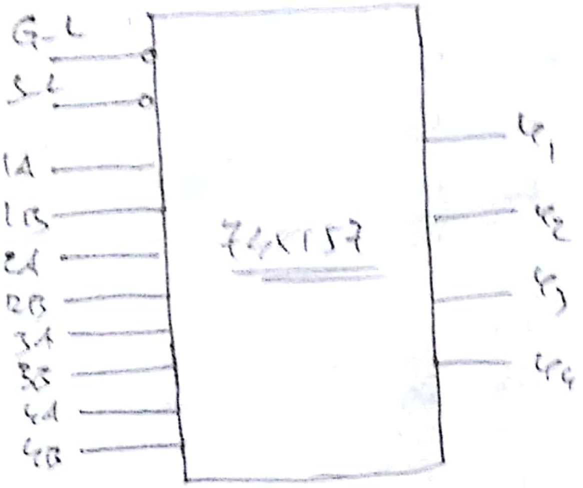
Truth table

i/p				o/ps	
ENL	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



74x157 → 2 I/p 4 bit mux



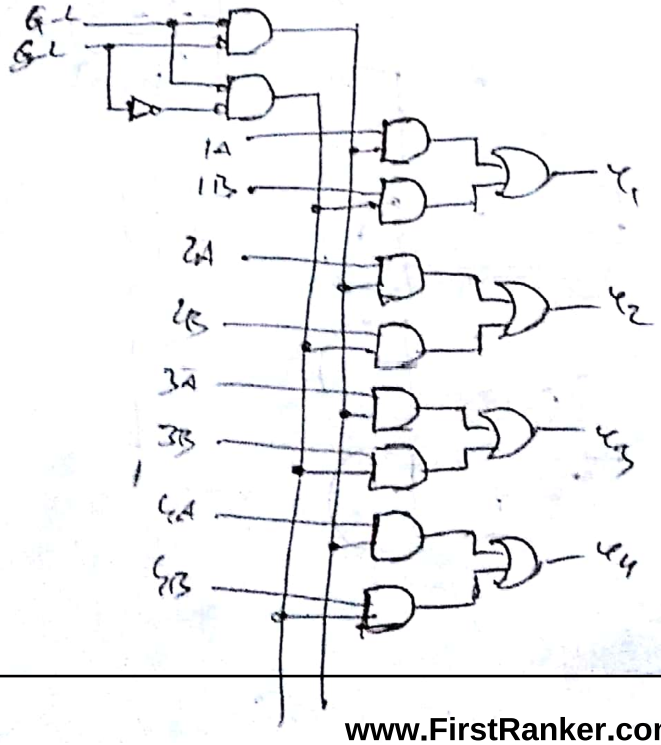
logic symbol

Truth Table

I/P		O/Ps			
G-L	S-L	Y1	Y2	Y3	Y4
1	x	0	0	0	0
0	0	1A	2A	3A	4A
0	1	1B	2B	3B	4B

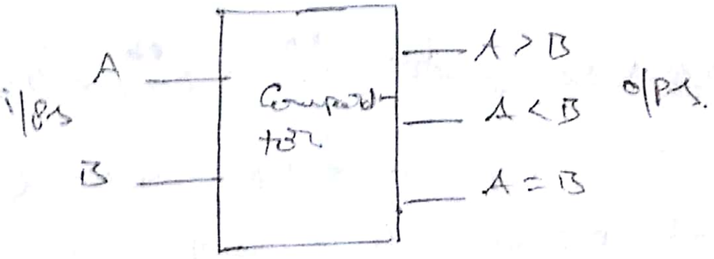
S-L → Active low select i/p line.

G-L → Active low gate pulse



Comparators

Comparator is a Combinational logic CKT which Compares two Binary signal of any length & gives the appropriate result i.e. greater than, less than & Equal to its o/p terminals.



If A & B are represent by 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.

$$2^{2\times n} = 2^{2\times 1} = 4 \text{ rows.}$$

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

Both i/ps must be equal length.

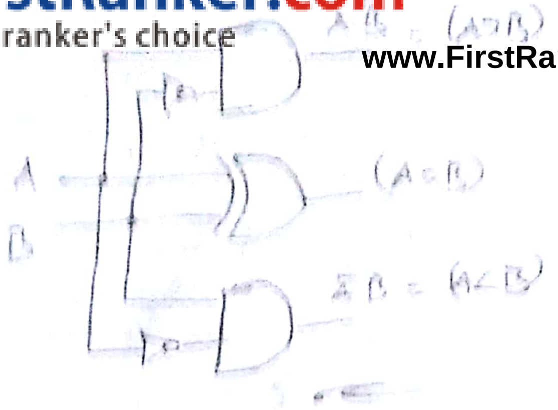
From the Truth table

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

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

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

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



Four Bit Comparator

For four Bit comparator $2^{2\times4} = 2^{8} = 256$ rows

This is very complicated to Design by using the above procedure (Truth table / model)

Hence to simplify the design compare two 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 or else we should not compare A_2 & B_2. in 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 7

$A>B \Rightarrow A_3>B_2 + x_3(A_2>B_2) +$

$x_3 x_2 (A_1>B_1) + x_3 x_2 x_1 (A_0>B_0)$

$\Rightarrow A_3\bar{B}_3 + x_3 A_2\bar{B}_2 + x_3 x_2 A_1\bar{B}_1 + x_3 x_2 x_1 A_0\bar{B}_0$

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

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

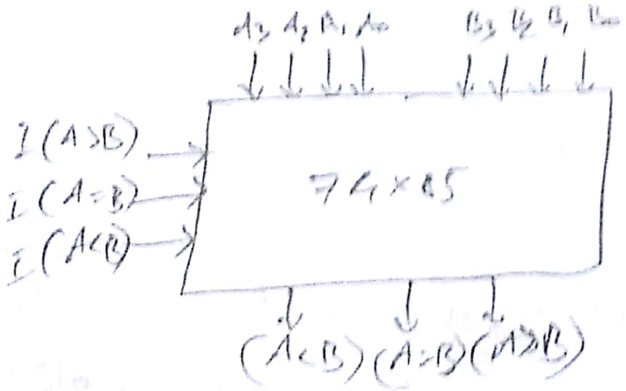
where $x_3 = A_3 \odot B_3$

$x_2 = A_2 \odot B_2$

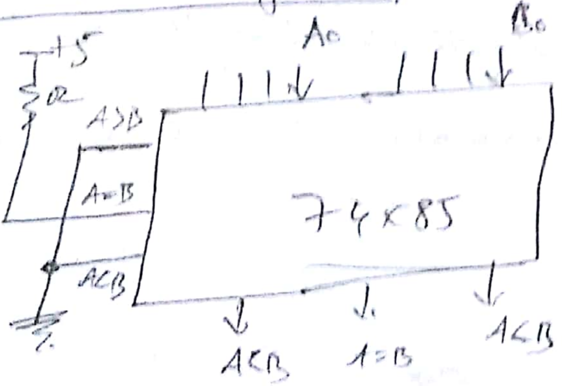
$x_1 = A_1 \odot B_1$

$x_0 = A_0 \odot B_0$

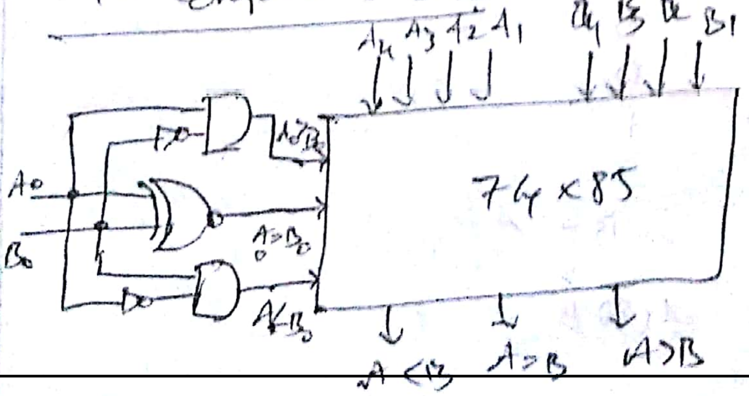
MSI Comparator



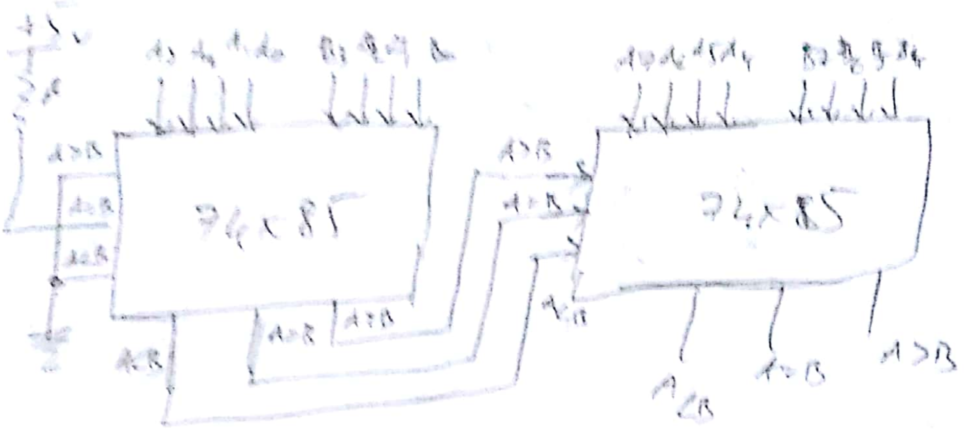
To Compare Single Bit



To Compare 5 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~~ a Combinational circuit which is used to add ~~two~~ three single bits & it generates Sum & carry o/ps.

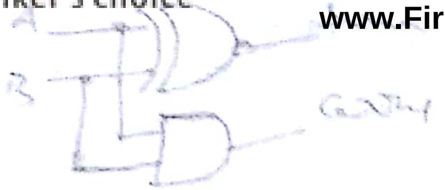
Half adder

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 = AB



Full Adder

A	B	Cin	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

$$\text{Sum} = A\bar{B}\,\overline{C_{in}} + \bar{A}\bar{B}C_{in} + ABC_{in} + \bar{A}B\overline{C_{in}}$$

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

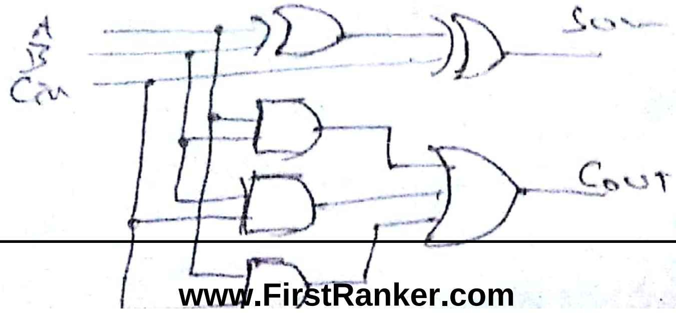
$$= A(\overline{B\oplus C_{in}}) + \bar{A}\,(B\oplus C_{in})$$

$$\text{Sum} = A\oplus B\oplus C$$

Carry

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

$$C_{out} = BC_{in} + AB + AC_{in}$$



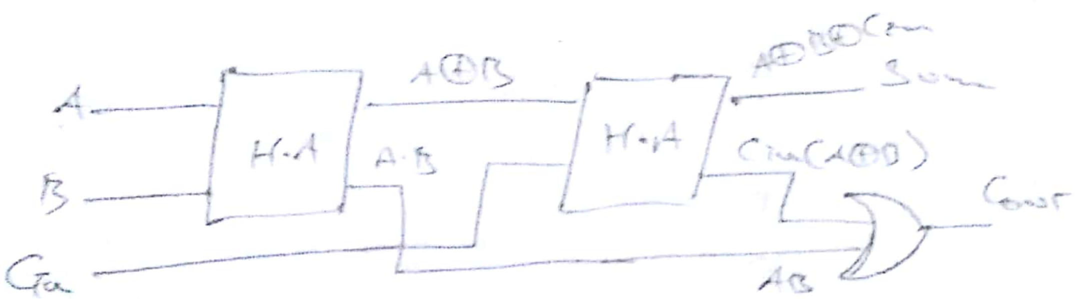
Full adder using half adder

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

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

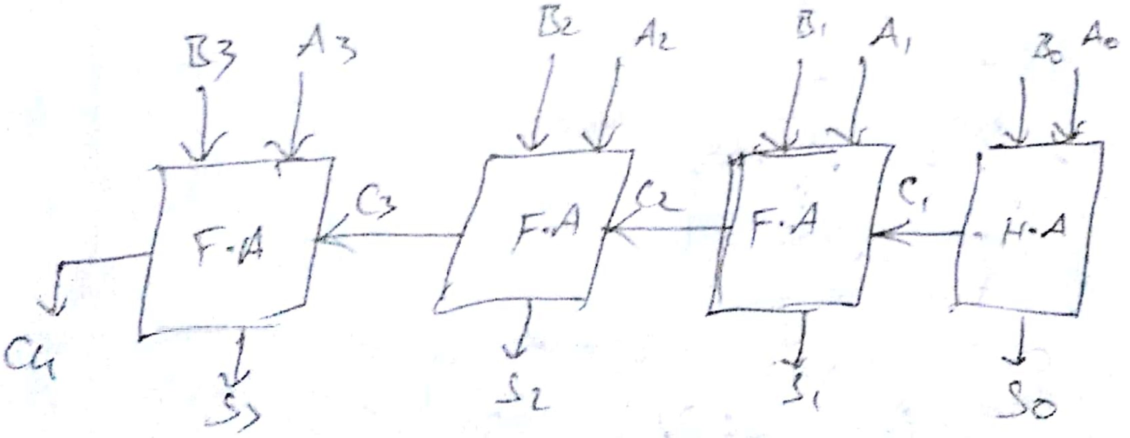
$$= C_{in}(\bar{A}B + A\bar{B}) + AB(\overline{C_{in}} + C_{in})$$

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



4-bit parallel adder (Ripple carry Adder)

$$\begin{array}{lccccc}
 & & C_3 & C_2 & C_1 & \\
A \Rightarrow & & A_3 & A_2 & A_1 & A_0 \\
B \Rightarrow & & B_3 & B_2 & B_1 & B_0 \\
\hline
 & C_4 & S_3 & S_2 & S_1 & S_0
\end{array}$$



BASIC BISTABLE ELEMENTS

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

Analysis. The circuit of fig I is often called a bistable, since a strictly digital Analysis shows that it ~~was~~ 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.

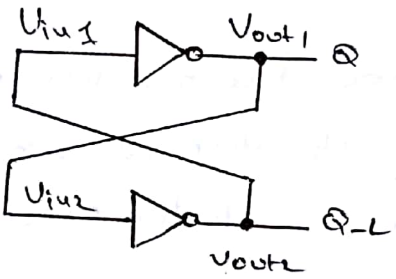


Fig 1: A pair of inverters forming a bistable element.

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,~~ well, and we'll will actually show a couple of appli.

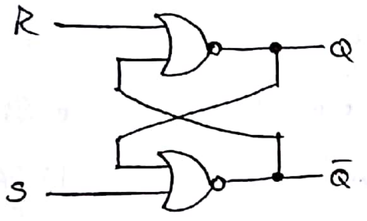
LATCHE & FLIP FLOPS

Latches & FlipFlops are the basic building blocks of most Sequential circuits. Typical digital systems use latches & flip Flops 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.

Most 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.

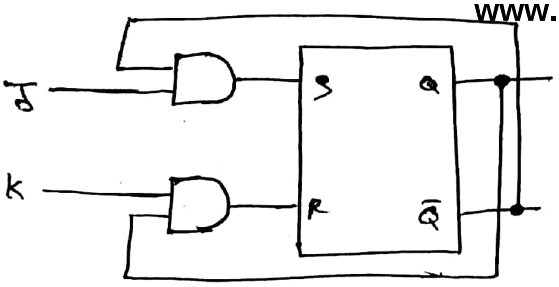
1. SET = 0, RESET = 0: This is the normal resting state of the NOR latch and it has no effect on the o/p state. Q and Q̄ will remain in whatever state they were prior to the occurance of this input condition.

2. SET = ~~0~~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 RESE 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 simultaniously, the resulting o/p state is ~~a~~ erratic & unpredictable. This o/p condition should not be used. It is forbidden (Invalid)

JK latch

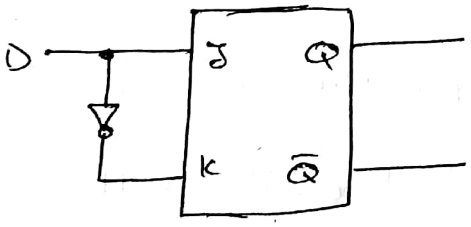


logic Diagram

J	K	S	R	Q_n	Q_{n+1}	State
0	0	0	0	0	0	nochange
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

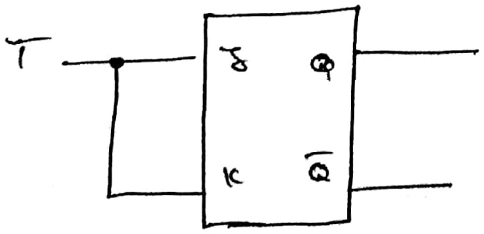


logic diagram

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

Truth table

T- latch



logic diagram

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

Truth table

Excitation tables of Flip Flops

SR Flip Flop

Qn	Qn+1	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

JK Flip Flop

Qn	Qn+1	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

D Flip Flop

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

T - Flip Flop

Qn	Qn+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 ~~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.

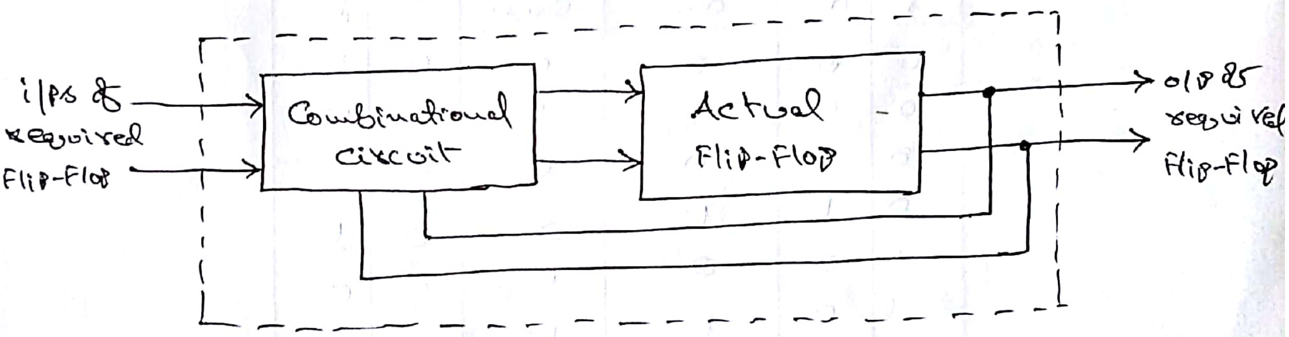


Fig 1 - Block diagram for conversion of Flip-Flop.

S-R Flip Flop to J-K Flip Flop :-

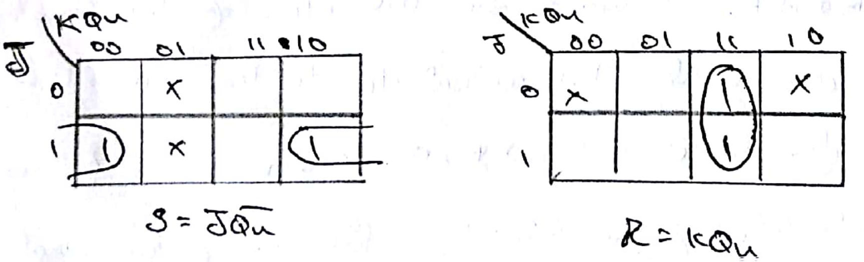
Here the External i/ps to the already available S-R Flip Flop will be J and K. S and R are the o/ps of the combinational circuit, which are also the actual i/ps to the S-R flip-flop. We write a truth table with J, K, Q_n, Q_{n+1}, S, and R, where Q_n is the present state of the flip flop & Q_{n+1} is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

[illegible] have eight combinations. For each combination of J, K & Qn, find the corresponding Q_{n+1}, i.e determine to which next state (Q_{n+1}) the JK Flip flop will go from the present state Q_n if the present ips. J & K are applied. Now Complete the table by writing the values of S & R required to get each Q_{n+1} from the corresponding Q_n, i.e. write what values of S & R are required to change the state of the flip-flop ~~⊕~~ from Q_n to Q_{n+1}.

The conversion table, the K-maps for S & R in terms of J, K & Q_n 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	Q_n	Q_{n+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



$S = J\overline{Q_n}$

$R = KQ_n$

K-maps for S & R

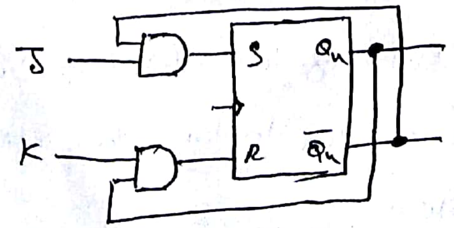


Fig-2 logic diagram

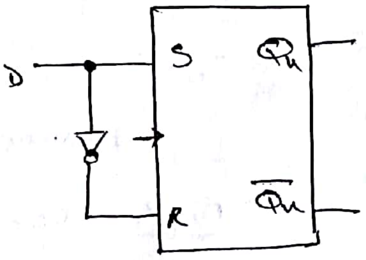
S-R to D flip flop:

Here S-R flip flop is available & we want the operation of the D flip flop 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/ps of the existing flip-flop S & R in terms of the External input D & the present state Qn.

The Conversion table, the K-maps for S & R interms of D & Qn, 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

D \ Q_n	0	1
0	0	0
1	1	X

$S = D$

D \ Q_n	0	1
0	X	1
1	0	0

$R = \overline{D}$

K-maps for S & R fig-3

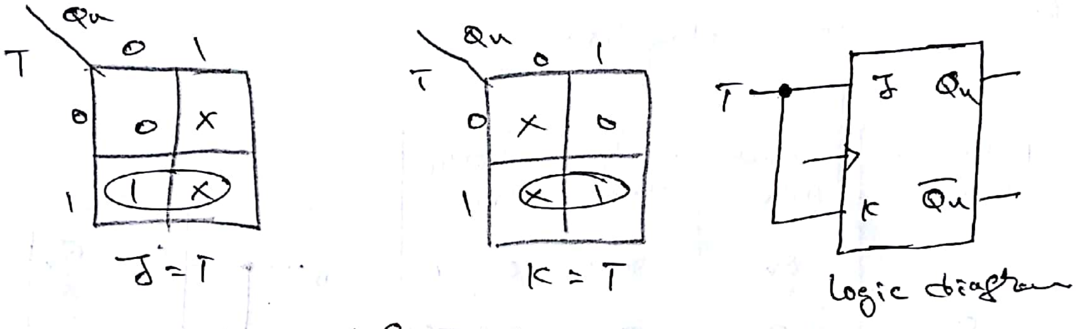
J-K flip flop to T flip-flop:-

Here J-K flip flop 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 & Qn make four combinations. Express J & K interms of T and Qn.

The Conversion table, the K-maps for J & K interms of T & Qn, & 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	Qn	Qn+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 JK flip flop to T flip flop

SSI Latches & FlipFlops:

Several types of discrete latches & FlipFlops are available as SSI parts. SSI latches and FlipFlops 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 a common C control line for each pair.

Among the devices in Fig 5 the most important is the 74x74, which contains two independent positive-edge triggered D-flip flops with preset & clear inputs.

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

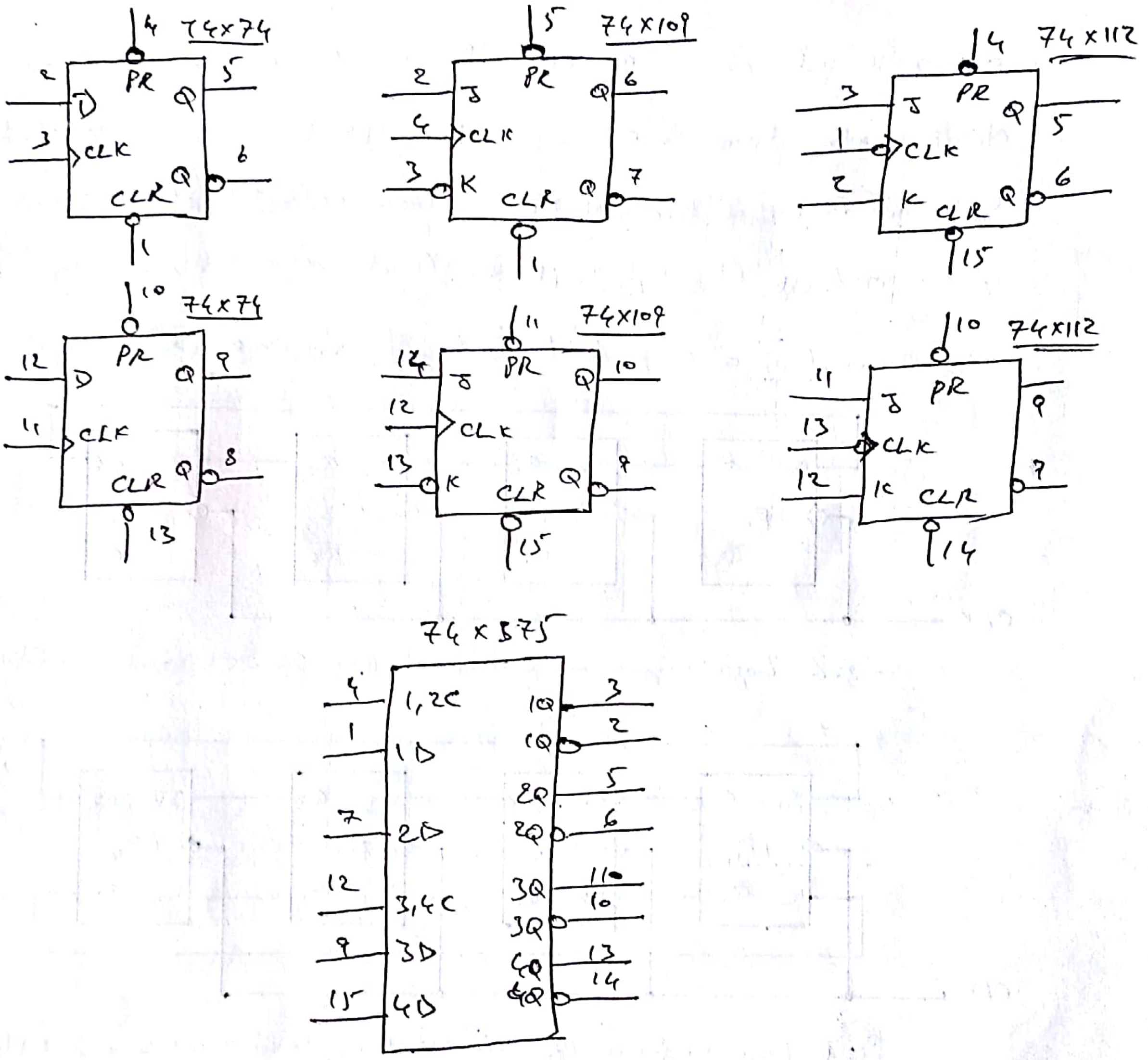


Fig5: pinouts for SSI latches & flip flops

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 FFs 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 ~~TO~~ first FlipFlop such ~~tha~~ that the array of FlipFlops is arranged in a ring & therefore, the name "ring Counter"

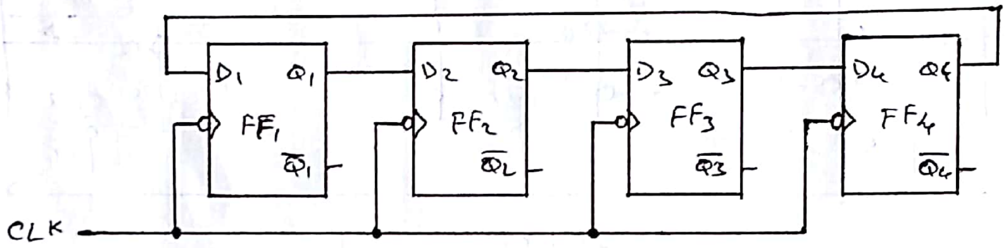


Fig: 6 Logic diagram of a 4-bit Ring Counter using D-flipflops

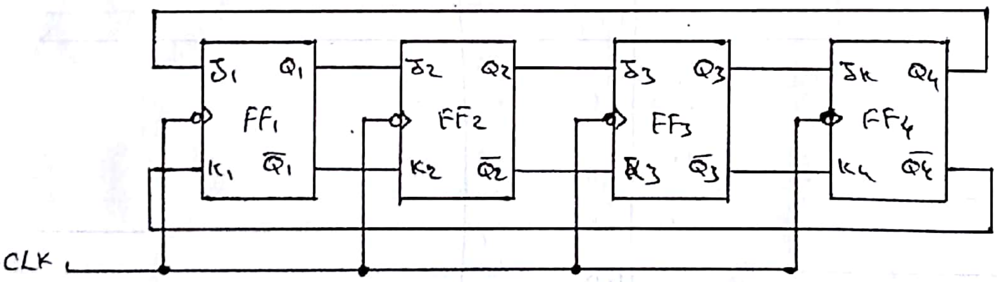
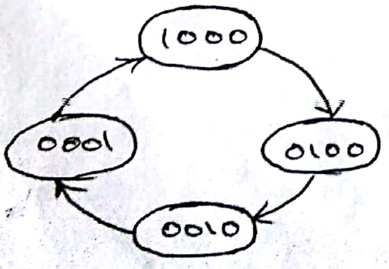


Fig 7: Logic diagram of a 4-bit Ring Counter using J-K flipFlops



State diagram

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

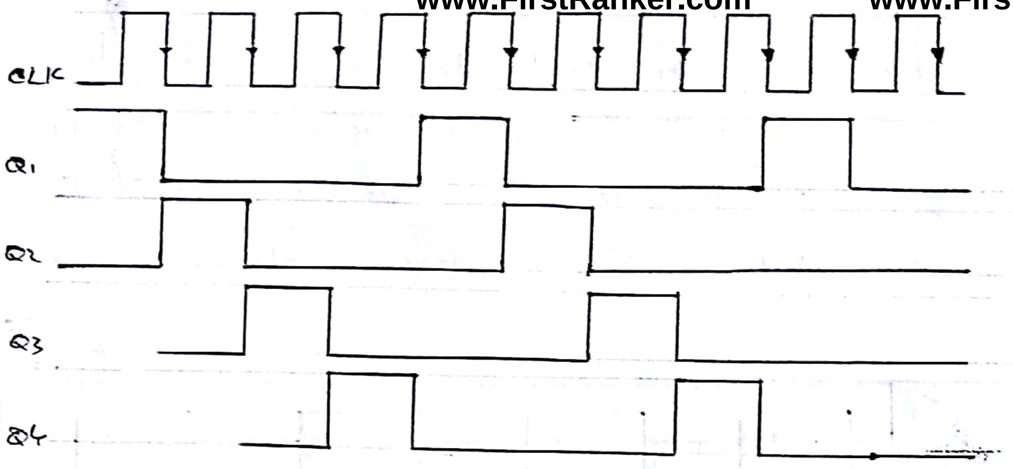


Fig 8: Timing diagram of a 4-bit ring counter.

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 un economical 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 & 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

...ring counter. This feed back arrangement Produces 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.

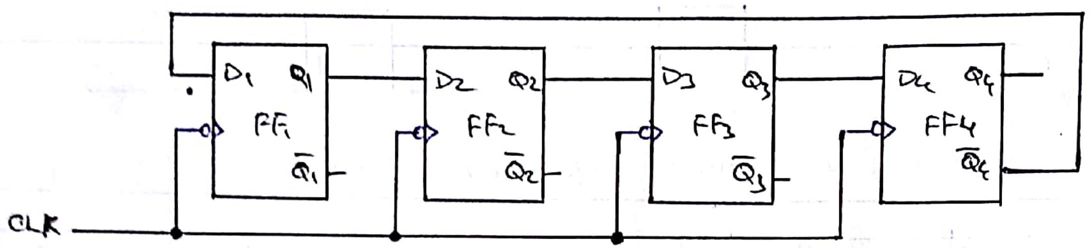


Fig10: logic diagram of a 4-bit twisted ring counter using D FFs.

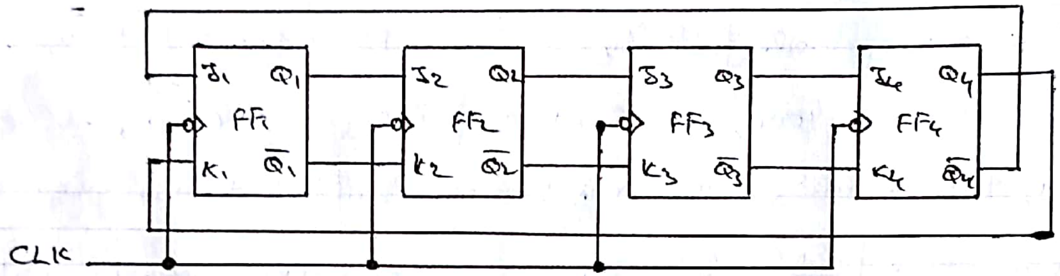
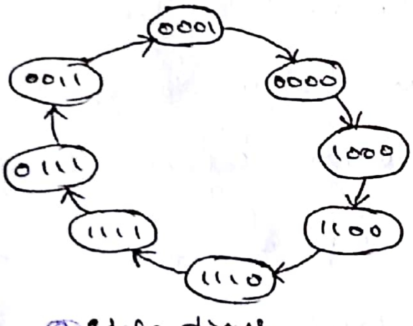


Fig11: logic diagram of a 4-bit twisted ring counter using JK FFs



(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

Fig12: State diagram & sequence table of a twister ring Counter

Let initially all the FFs be reset, i.e; 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.

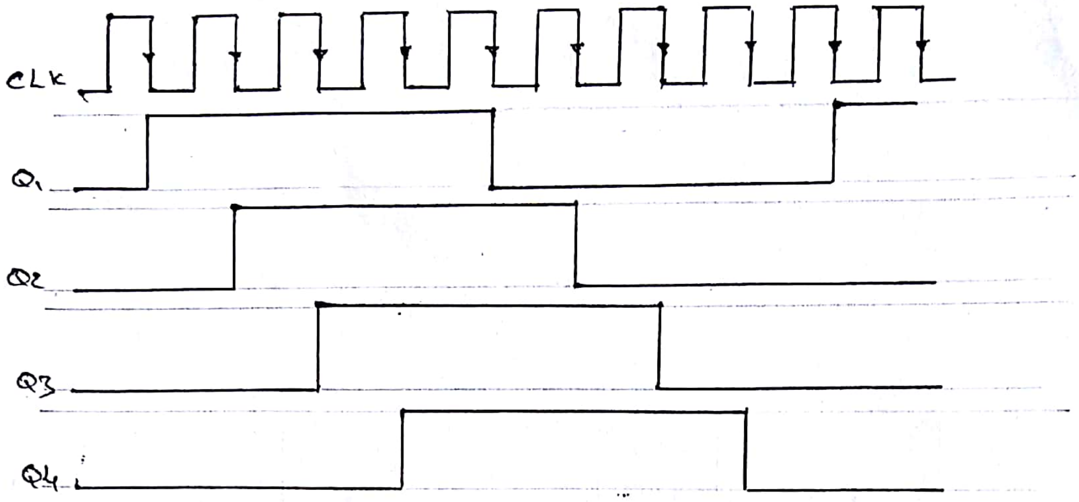


Fig 13: Timing diagram of a 4-bit twisted ring counter

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 requires 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)

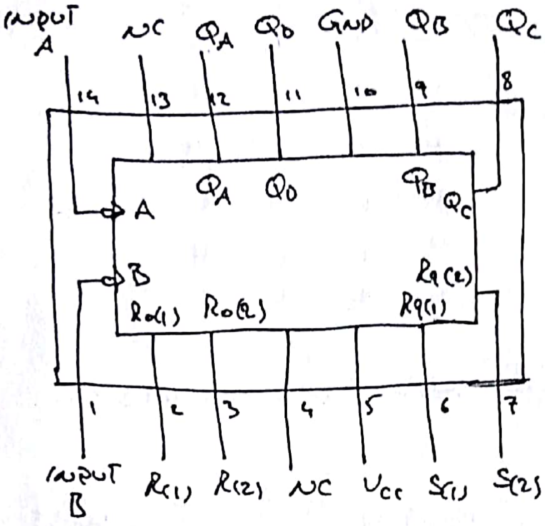


Fig: Connection diagram for 7490

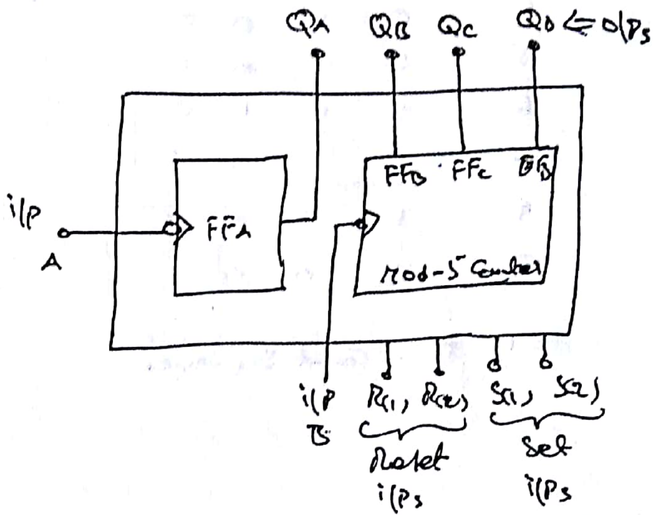


Fig: Basic internal structure of 7490

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 B i/p must be externally connected to the QA o/p & A i/p receives the incoming count.

2. Symmetrical Bi-quinary Divide-by-Ten Counter: The QD o/p must be externally connected to the A i/p. The i/p count is then applied to the B i/p & a ~~1/10~~ ÷10 square wave is obtained at o/p QA.

3. ÷2 & ÷5 Counter: No external interconnections 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 QD o/p.

	OUTPUTS			
Count	QD	QC	QB	QA
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

	OUTPUTS			
Count	QA	QD	QC	QB
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)

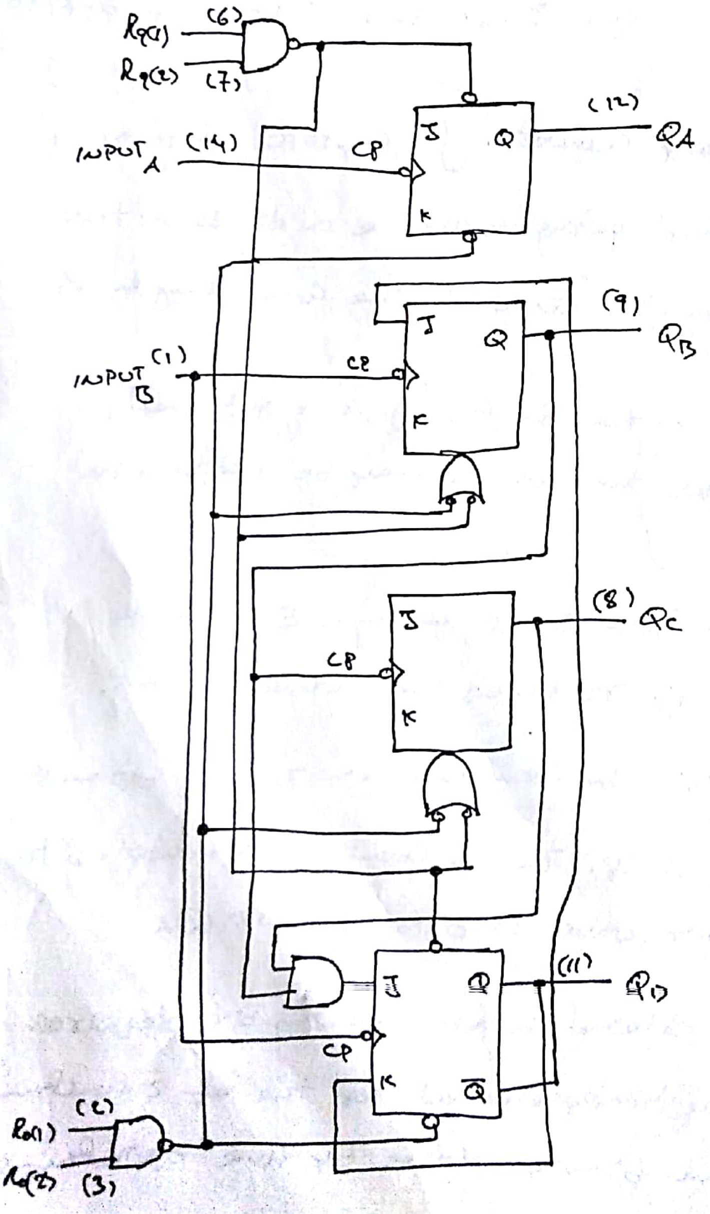
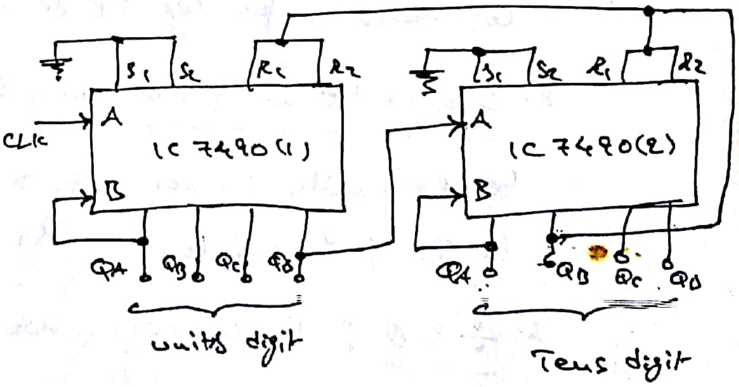


Fig: Logic Diagram for 7490

Reset i/ps				outputs			
Ro(1)	Ro(2)	Rq(1)	Rq(2)	QD	QC	QB	QA
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

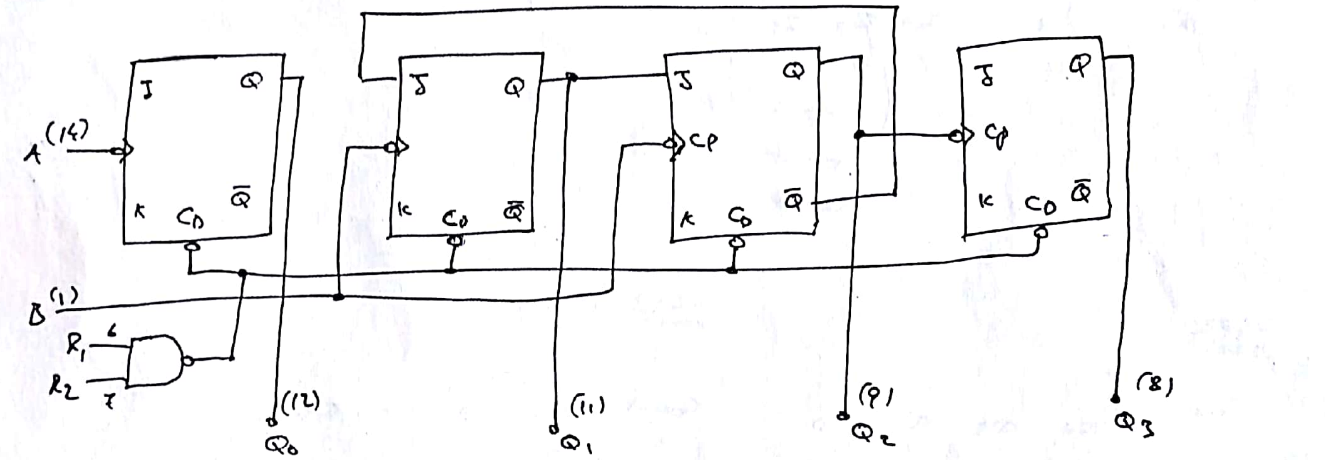
IC 7492/93 (4-bit Ripple Counters)

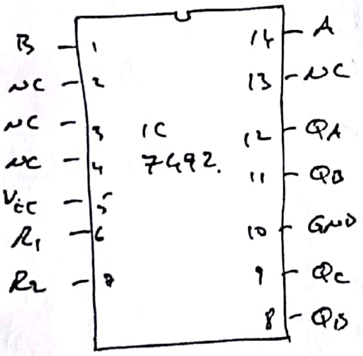
The 7492 & 7493 are high speed 4-bit ripple type Counters Partitioned into two sections. Each Counter has a divide-by-two section & either a divide-by-six (7492) or divide-by-eight (7493) section which are triggered by a High to low transition on the clock i/ps. Each section can be used separately or 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:



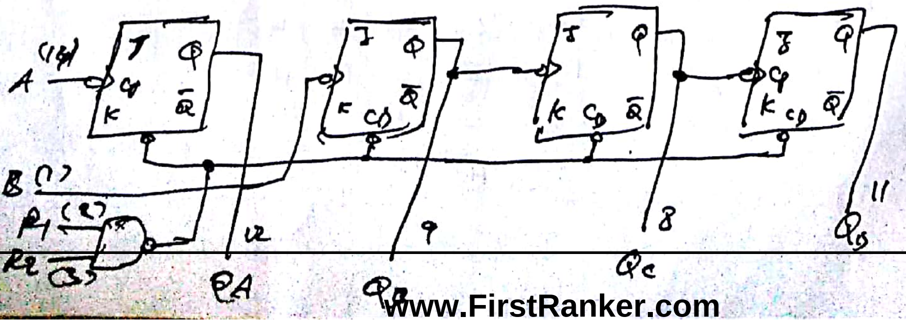


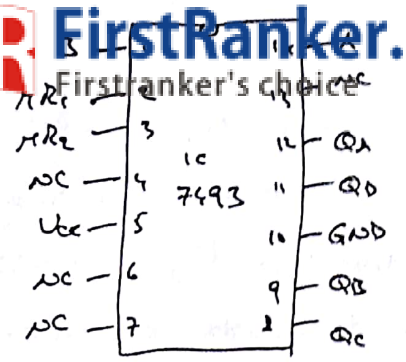
Modes of 7492

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.

2. ÷2 & ÷6 : no External interconnections are required. The first ff is used as a binary element for the ÷2 fun. The B ~~o/p~~ i/p is used to obtain the ÷3 operation ~~that~~ at the Q_B & Q_C o/ps. & divide by six operation at the Q_D o/ps.

7493





MODE OF 7493:-

1. 4-bit ripple counter: The o/p QA must be externally connected to i/p B. The i/p count pulses are applied to the i/p A. Simultaniously divisions of 2, 4, 8 & 16 are performed at the QA, QB, QC & QD o/p as shown in the truth table.

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

7492 & 7493

Reset i/p		O/PS			
R₁	R₂	QA	QB	QC	QD
H	H	L	L	L	L
L	H	count			
H	L	count			
L	L	count			

7492

Count	QA	QB	QC	QD	
0	L	L	L	L	0000
1	H	L	L	L	1000
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	QA	QB	QC	QD
0	L	L	L	L
1	H	L	L	L
2	L	H	L	L
3	H	H	L	L
4	~~H~~L	L	H	L
5	H	~~H~~L	H	L
6	~~H~~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:	$t0$	$t1$	$t2$	$t3$	$t4$	$t5$	$t6$	$t7$	$t8$	$t9$	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

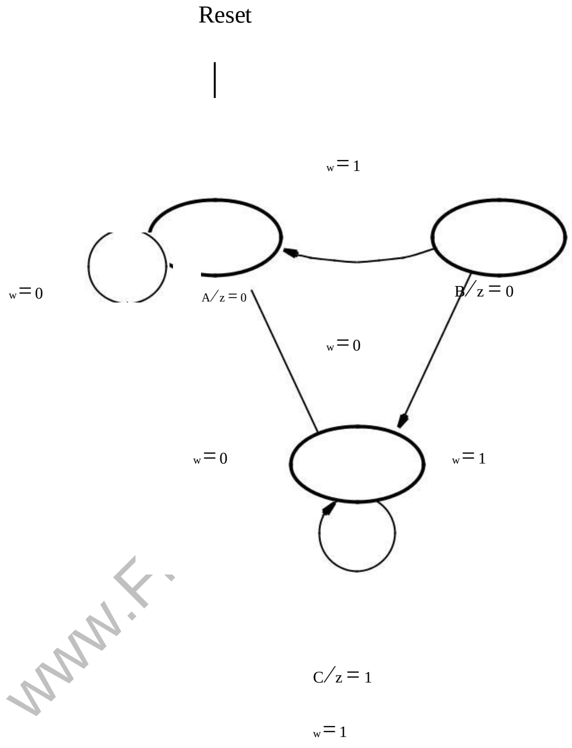


Figure 6.3 State diagram of a simple sequential circuit

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.

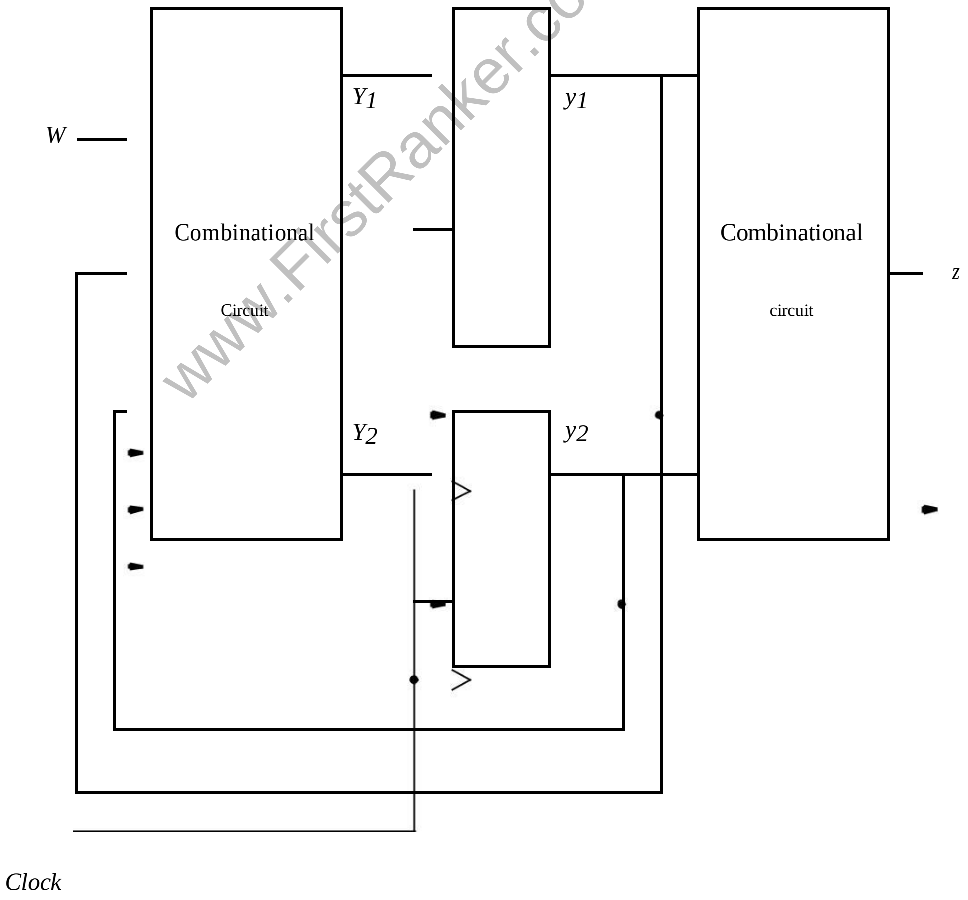
6.4 STATE ASSIGNMENT

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 Z
	w = 0	*w* = 1	
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.

6.5CHOICE 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

$$Y_1 = w y_{\underline{1}} y_{\underline{2}}$$

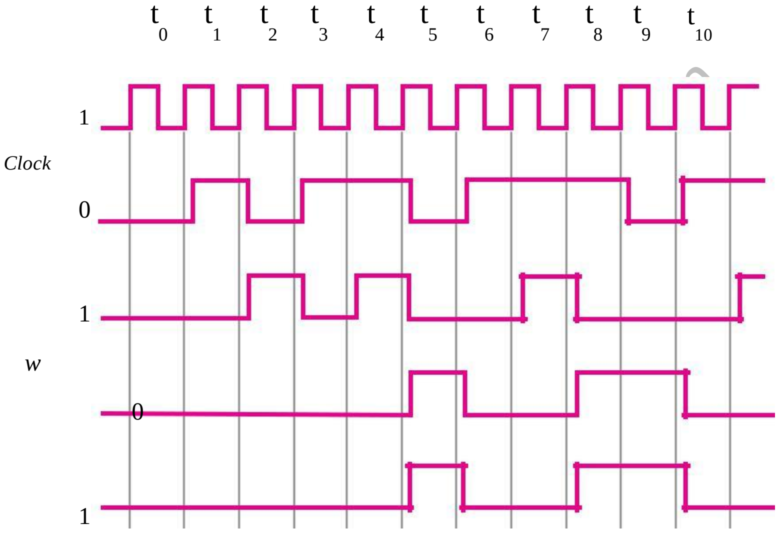
$$Y_2 = w(y_1 + y_2)$$

$$z = y_2$$

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 ;
```

```
               ELSE
                  y <  B ;
               END IF ;
            WHEN B   >
               IF w   ’0’ THEN
                  y <  A ;
               ELSE
                  y <  C ;
               END IF ;
            WHEN C   >
               IF w   ’0’ THEN
                  y <  A ;
               ELSE
                  y <  C ;
               END IF ;
         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 ;
```