DIGITAL ELECTRONICS

MODULE 1: PRINCIPLES OF COMBINATION LOGIC

- Definition of combinational logic
- Canonical forms
- Generation of switching equations from Truthtables.
- Karnaugh Maps (3, 4 & 5 Variables)
- Incompletely specified functions
- Simplifying Maxterm equations
- Quine Mccluskey minimization technique.
- Quine Mccluskey using dont care terms.
- Reduced Prime implicants Tables.

III ECE

DEFINITION OF COMBINATIONAL LOGIC:

- Combinational logic deals with the techniques of combining the basic gates into circuits that perform some desired function.

  Ex: Adders, Subtractors, Decoders, Encoders, Multipliers, Dividers, Display drivers and Keyboard Encoders.

- Logic circuit without feedback from output to the input, constructed from a functionally complete gate set, are said to be combinational.

- Logic circuits that contain no memory are combinational.

COMBINATIONAL LOGIC MODEL.

- Combinational logic can be modeled as shown in fig 1.1.

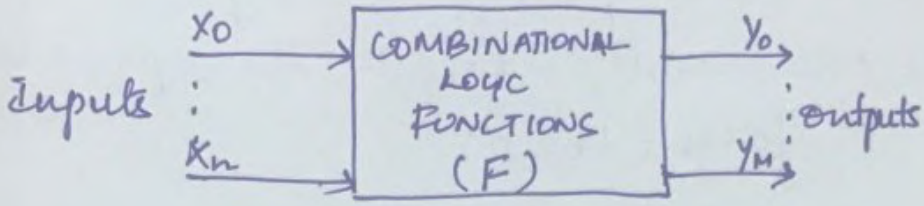


Fig 1.1 - COMBINATIONAL LOGIC MODEL

- Let $\underline{X}$ be the set of all input variables $\{x_0, x_1, \ldots x_n\}$ and $\underline{Y}$ be the set of all output variables $\{Y_0, Y_1, \ldots Y_m\}$.
- The combinational function $\underline{F}$, operates on the input variable set $\underline{X}$, to produce the output variable set $\underline{Y}$.
- Output Variables Y_0 through Y_m are not fed back to the input. The output is related to input as

$$Y = F(x)$$

- The relationship between the input and output variables can be expressed in equations, logic diagrams or truth tables.

- A truth table specifies the input conditions under which the ops are true or false (1 or 0).

- Switching equations are then derived from the truth tables and realized using gates.

PROBLEM STATEMENTS TO TRUTH TABLE

- Before any combinational logic system can be designed it must be defined.
- Proper statement of a problem is the most important part of any digital design task.
- Once correctly and clearly stated, any problem can be converted to the necessary logic for implementation.

GENERAL LOGIC DESIGN SEQUENCE

Fig 1.2 illustrates the sequence of design tasks in a general way.

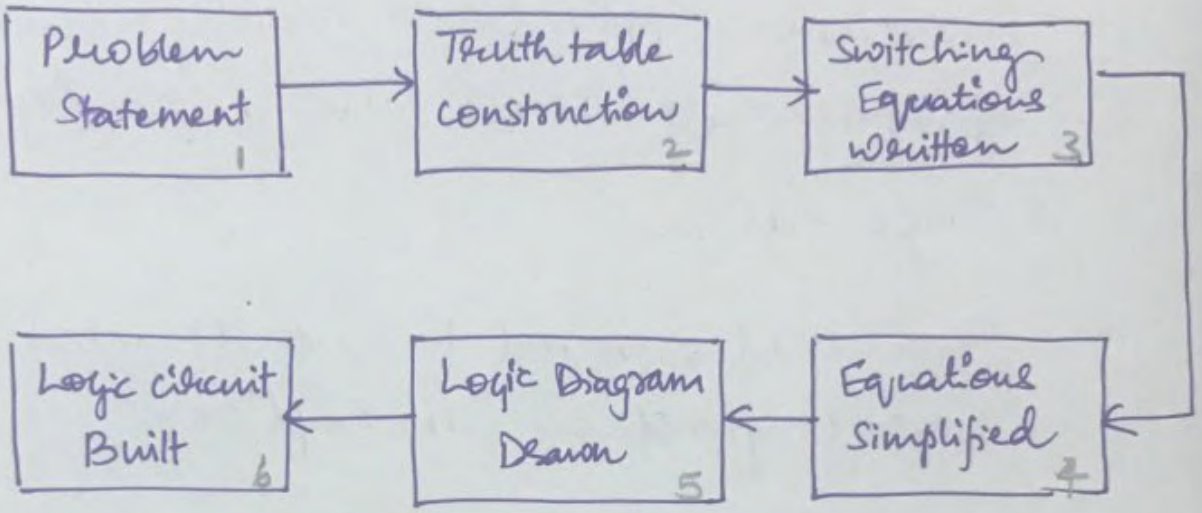


Fig 1.2 · GENERAL LOGIC DESIGN SEQUENCE

1. First task is to define the Problem to be solved.
2. The problem is then rewritten in the form of a truth table.
3, 4, 5. From the truth table, the switching equations can be written and simplified and the logic diagram drawn.
6. The logic diagram can be realized using any of the three main digital integrated circuit families.

TTL - Transistor - Transistor logic,
ECL - Emitter coupled logic or
CMOS - complementary Metal-oxide Silicon.

- Practical applications rarely come in a prepackaged truth table, ready for logic design.
- Truth tables must be constructed from verbal problem descriptions.

Ex Pg 6, 7, 8, 9.

DERIVING SWITCHING EQUATIONS.

- Boolean Equations can be directly derived from a truth table or from the logic Diagram.
- Likewise a truth table or logic diagram can be constructed from the Boolean equations.
- Each input variable group that produces a logical 1 in a truth table output column can form a term in a Boolean Switching equation.
- For Example, in the following truth table, the output variable M_1 is a 1 in four cases.

S_3	S_2	S_1	M_4	M_3	M_2	M_1
0	0	0	0	0	0	0
0	0	1	1	0	0	1
0	1	0	1	0	1	0
0	1	1	1	0	0	1
1	0	0	1	1	0	0
1	0	1	1	0	0	1
1	1	0	1	1	1	0
1	1	1	1	0	0	1

$\{S_3', S_2', S_1\}$, $\{S_3', S_2, S_1\}$ $\{S_3, S_2', S_1\}$ and $\{S_3, S_2, S_1\}$.

- The remaining input variable combinations cause output M_1 to be a logical 0.

- A boolean equation can be written that defines all the conditions under which output M_1 is a logical 1.

- Each term in the equation is formed by ANDing the input variables. Each AND term is then ORed with the other AND terms to complete the output Boolean equation.

- For M_1, the boolean equation would be written as

$$M_1 = S_3' S_2' S_1 + S_3' S_2 S_1 + S_3 S_2' S_1 + S_3 S_2 S_1$$

- Each AND term (Product term) identifies one input condition where the output is a 1.

DEFINITIONS

1. **LITERAL** : A Literal is a Boolean Variable or its complement. For instance, let X be a binary variable, then both X and X' would be literals.

2. **PRODUCT TERM**: A Product term is a literal or the logical product (AND) of multiple literals.

   For instance, let X, Y and Z be the binary variables. Then a representative product term could be X, XY, XYZ or X'YZ etc....

3. **SUM TERM**: A Sum is a literal or the logical OR of multiple literals. Let X, Y and Z be the binary variables.
   Then a representative Sum term could be X, X'+Y, or X+Y'+Z' etc..

4. **SUM OF PRODUCTS**: A Sum of Products (SOP) is the logical OR of multiple product terms. Each product term is the AND of binary literals.
   For ex: XY' + X'+YZ +XY'Z' is a SOP Expression.

5. **PRODUCT OF SUMS:** A Product of SUMS (POS) is the logical AND of multiple OR terms. Each sum term is the OR of binary literals.

For ex: $(X+Y')(X+Y+Z')(Y'+Z')$ is a POS expression.

6. **MINTERM:** A Minterm is a special case product (AND) term. A minterm is a product term that contains all of the input variables (each literal no more than once) that make up a boolean expression.

7. **MAXTERM:** A maxterm is a special case SUM (OR) term. A maxterm is a Sum term that contains all of the input variables (each literal no more than once) that make up a boolean expression.

8. **CANONICAL SUM OF PRODUCTS:** A canonical sum of products is a complete set of minterms that defines when an output variable is a logical 1.

- Each minterm corresponds to the row in the truthtable where the output function is 1 i.e, the SOP for the output M in the following table is

$$M = a'bms + ab'ms + abms.$$

a	b	m	s	M	
0	0	0	0	0	
0	0	0	1	0	
0	0	1	0	0	
0	0	1	1	0	
0	1	0	0	0	
0	1	0	1	0	
0	1	1	0	0	
0	1	1	1	1	← a'bms
1	0	0	0	0	
1	0	0	1	0	
1	0	1	0	0	
1	0	1	1	1	← ab'ms
1	1	0	0	0	
1	1	0	1	0	
1	1	1	0	0	
1	1	1	1	1	← abms

9. CANONICAL PRODUCT OF SUMS: A canonical product of sums is a complete set of maxterms that defines when an output is a logical 0.

Each maxterm corresponds to a row in the truthtable where the output is a 0

i.e, the POS for the output O_1 in the following table is

$$O_1' = (C_1+C_2+C_3)(C_1'+C_2+C_3)(C_1'+C_2+C_3')(C_1'+C_2'+C_3) * (C_1'+C_2'+C_3')$$

C_1	C_2	C_3	O_1	
0	0	0	0	$(C_1+C_2+C_3)$
0	0	1	1	$(C_1+C_2+C_3)$
0	1	0	1	$(C_1+C_2'+C_3)$
0	1	1	1	$(C_1+C_2'+C_3')$
1	0	0	0	$(C_1'+C_2+C_3)$
1	0	1	0	$(C_1'+C_2+C_3')$
1	1	0	0	$(C_1'+C_2'+C_3)$
1	1	1	0	$(C_1'+C_2'+C_3')$

The table 1.1 shows the complete nature of minterms and maxterms.

- An input variable is complemented when it has value 0 if we are writing minterms.
- The input variables are complemented when they have a value 1, if we are writing maxterms.

- Minterms represent output variable 1s and maxterms represent output variables 0s.

- Lower case m is used to denote a minterm and Upper case M is used to denote a maxterm.

- The number subscript indicates the decimal value of the term.

abc	MINTERM		MAXTERM	
	TERM	DESIGNATION	TERM	DESIGNATION
000	$a'b'c'$	m_0	$(a+b+c)$	M_0
001	$a'b'c$	m_1	$(a+b+c')$	M_1
010	$a'bc$	m_2	$(a+b'+c)$	M_2
011	$a'bc$	m_3	$(a+b'+c')$	M_3
100	$ab'c'$	m_4	$(a'+b+c)$	M_4
101	$ab'c$	m_5	$(a'+b+c')$	M_5
110	abc'	m_6	$(a'+b'+c)$	M_6
111	abc	m_7	$(a'+b'+c')$	M_7

Table 1.1 - Minterm & Maxterm Designations

- output equations can be written directly from the truth table using either minterms or maxterms.

- When an output equation is written in minterms or maxterms, it is a canonical expression.

CANONICAL FORMS

- Canonical is a word used to describe a condition of a switching equation.
- In normal use the word means "conforming to a general rule". The rule for switching logic, is that each term used in a switching equation must contain all of the available input variables.
- Two formats generally exist for expressing switching equations in a canonical form.
  1. Sum of Minterms (Products)
  2. Product of Maxterms. (sums)
- Canonical expressions are not simplified

why canonical forms?!?!?!

1. Situations that occur when logic designers must manipulate Boolean equations for purposes other than simplification.
2. Conversion from one form to another form (TTL Nand Gates to ECL NOR Gates)
3. Entry into the Karnaugh Map.

1. SOP Equation to Canonical form.

To place a SOP equation to canonical form using Boolean algebra, we do the following.

(i) Identify the missing variable (s) in each AND term. (xy)

(ii) AND the missing term and it's complement with the original AND term.

$$xy(z+z')$$

Because $(z+z') = 1$, the original AND term value is not changed.

(iii) Expand the term by application of the property of distribution

$$xy(z+z') = xyz + xyz'$$

2. Pos Equation to canonical form

To place a Pos equation to canonical form using Boolean Algebra, we do this

(i) Identify the missing variables in each OR term. $(x+y')$

(ii) OR the missing terms and its complement with the original OR term.

Original : $x+y'$

Modified : $(x+y'+zz')$ ← using missing variable + its complement

→ Because $zz'=0$, the original term value is not changed.

(iii) Expand the term by application of distributive property.

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

GENERATION OF SWITCHING EQUATIONS FROM TRUTH TABLES

- Switching equations can be written more conveniently by using the minterm or maxterm numerical designation as shown in Table 1.1 (pg. No 1.10), instead of writing the variable names or their complements.

- m and M designation can be dropped and the decimal equivalent value for

the term can be written directly.

1. CANONICAL SOP EQUATION

For ex: consider the canonical SOP equation

$$P = ab'c + ab'c' + abc' + abc + a'bc$$

- For decode each of the minterms based on binary weighting of each variable and produce a list of decimal decoded minterms, the result would be

$$P = \Sigma(5, 4, 6, 7, 3).$$

- To keep the input variable notation from being lost in a minterm list the relationship $P = f(a, b, c)$ is used.
- This means that the output variable P is a function of the set of input variables $\{a, b, c\}$ with input variable a being the most significant bit (MSB).
- The sign Σ indicates summation and stands for the sum of products canonical form.
- when numbers in a decimal decoded set are preceded by a Σ sign, a

SOP expression is indicated. Each number represents a minterm.

$$P = f(a,b,c) = (ab'c + ab'c' + abc' + abc + a'bc)$$
$$= \sum (5,4,6,7,3)$$

2. CANONICAL POS EQUATION.

- Canonical Pos expressions can be written in similar fashion as SOP expressions.
- π (Pi) sign is used to indicate POS canonical form.
- The decimal number listed in a Pos set represents maxterms.
- The input variable names are indicated in the same manner as in SOP equations.

$$X = f(A,B,C)$$

- Maxterms are complement of minterms.

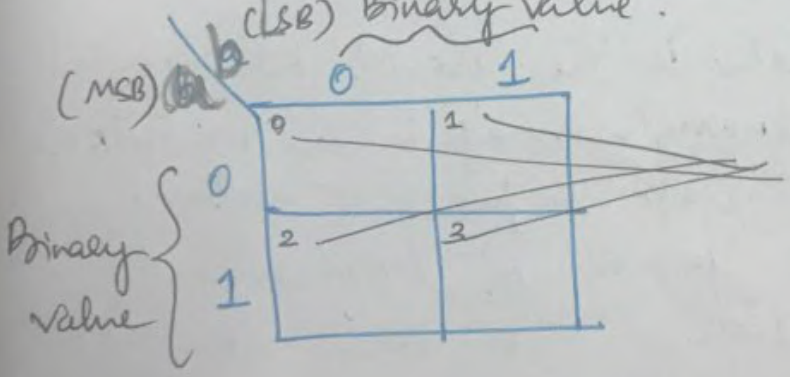
KARNAUGH MAPS.

- Simplification of Switching equations reduces the amount of hardware needed to realize a given function.
- Reduction of gates and gate inputs may result in fewer integrated circuits, which in turn decreases cost and improves reliability.
- Boolean Algebra can be used to simplify equations but the process is lengthy and error prone.
- It is required a systematic method for finding and eliminating any redundancies in an equation.
- A better approach is the use of Karnaugh map. The Karnaugh map is a matrix of squares. Each square represents minterm or maxterm from a boolean expression/equation. The arrangement of the matrix square permits identification of input variables redundancies, which helps reduce the output equation.

- The Karnaugh map identifies all of the cases for a given set of input variables where groups of minterms may contain redundant variables of the form $x + x' = 1$. When these groups are identified, the redunt variables can be eliminated resulting in a simplified output function.

- If a given switching equation contains a minterm, then a 1 is entered into the square that represents that term, A maxterm is represented by a 0.

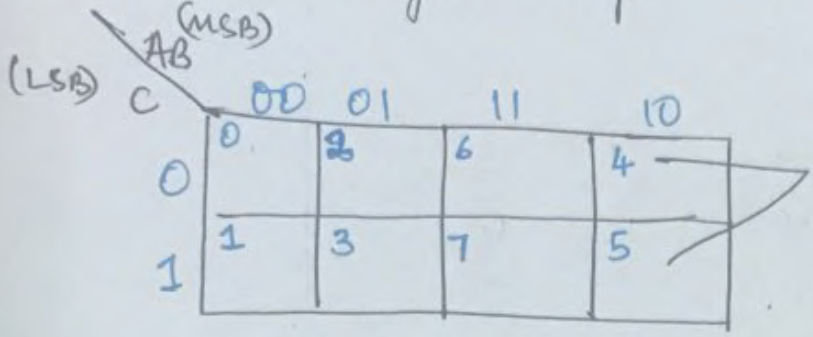
Two Variable K-Map.



- All four possible combinations of input variables are represented.

THREE AND FOUR VARIABLE KARNAUGH MAPS.

- The fig represents three-variable Karnaugh map.



- Each Square in the map represents a possible minterm or maxterm of a 3-variable function.
- The upper-left square represents binary 000, minterm ($A'B'C'$) or maxterm ($A+B+C$).
- As three binary variables can represent eight (8) unique combinations, we find 8 squares are needed.
- If minterm $A'B'C'$ (000) occured in a switching equation, then a 1 would be inserted into the upper left square.
- Assignment of a square to each of the 8 minterms resulting from 3-input variables results in given Table.

Inputs	Minterm	Maxterm
A B C	m	M
0 0 0	0	0
0 0 1	1	1
0 1 0	2	2
0 1 1	3	3
1 0 0	4	4
1 0 1	5	5
1 1 0	6	6
1 1 1	7	7

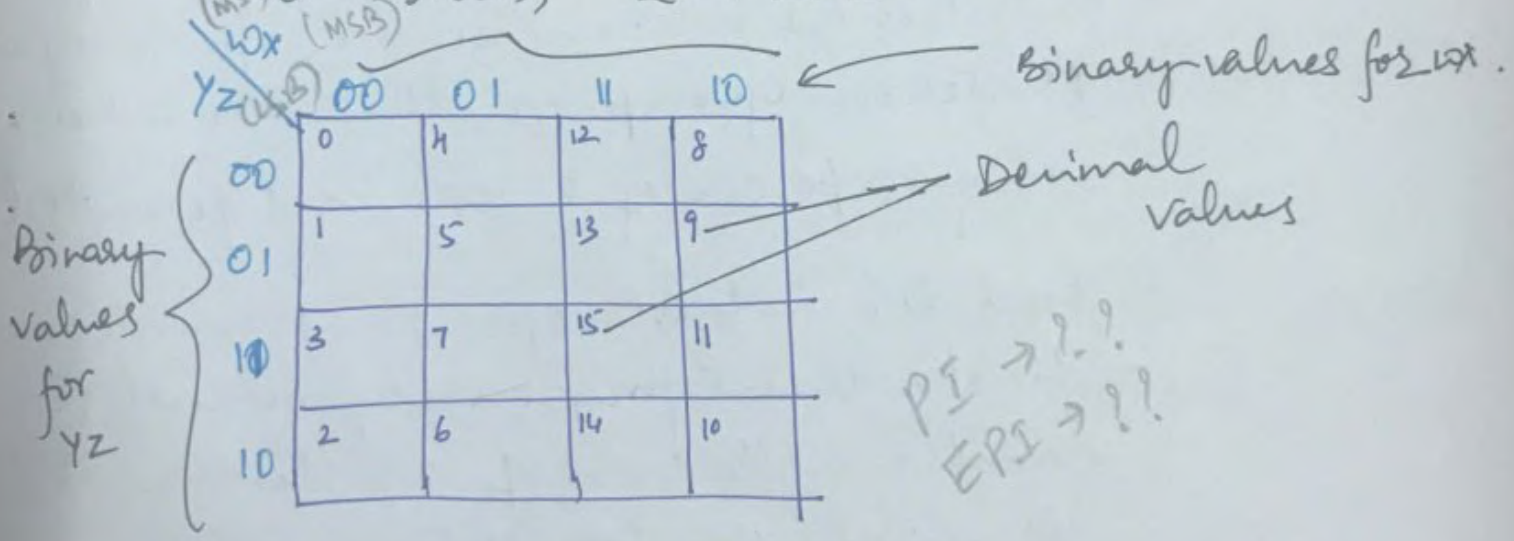
- We may label the decimal value for each square in a Karnaugh map by decoding the binary numbers as shown in fig( ).
- Across the top and down side of the 3 variable K-map only one bit changes occur between adjacent squares for each column and row
- Each adjacent row or column differs by only one variable.

FOUR VARIABLE K-MAP.

- is the same length both horizontally and vertically.
- Variable representation is the same in both directions, two variables across (00, 01, 10, 11) and two down.
- The input variable binary weighting is

  $W = 2^3 = 8$, $X = 2^2 = 4$, $Y = 2^1 = 2$, $Z = 2^0 = 1$

  W → MSB, Z → LSB.



fg( ): Four variable K-Map.

- Loading K-map involves writing 1 in the squares corresponding to a minterm or a 0 in the square corresponding to a maxterm.

STEPS INVOLVED IN LOADING & DETERMINING THE ESSENTIAL PRIME IMPLICANTS.

Step 1: load the minterms into the K-map. by placing a 1 in the appropriate Square.

Step 2: Look for Groups of minterms (PI)

(a) Group Size must be a power of 2 (∵ of Binary number System)

(b) PIs are formed from Groups of minterms whose size is a power of 2. Minterm Group Size that are not an integer power of 2 are not permitted.

- Find the largest groups of minterms first, then progressively evaluate smaller collections of minterms until all groups are found.

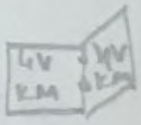
Step 3: Once all of the possible PIs have been identified, we determine whether any have minterms that are unique. If so, that PI is an Essential Prime Implicant [EPI]

Step 4: Select all essential prime implicants and a minimal set of remaining Prime Implicants that cover all remaining 1s in the K-Map.

Step 5: More than one equally simplified result is possible when more than one set of remaining PIs contain the same no. of minterms and maxterms.

FIVE VARIBLE K-MAP.

— Five variable K-maps can be constructed so that 3-variables are laid out horizontally and 2 vertically as shown.



DE \ ABC	000	001	011	010	110	111	101	100
00	0	4	12	8	24	28	20	16
01	1	5	13	9	25	29	21	17
11	3	7	15	11	27	31	23	19
10	2	6	14	10	26	30	22	18

fig ( ) : 5-variable K-Map.

- A, B, C, D, E are the input variables.
- 2^5 possible combinations exist, ranging from $(00000)_2$ to $(11111)_2$

A	← MSB	2^4	= 16
B		2^3	= 08
C		2^2	= 04
D		2^1	= 02
E	← LSB	2^0	= 01

- The decimal value associated with each square in the map is found by adding the column and row values.
- Each half of the five-variable map corresponds to a single 4-variable K-map.

STACKED VERSION OF 5-Variable K-MAP.

DE \ ABC	000	001	011	010	110	111	101	100
00	0	4	12	8	24	28	20	16
01	1	5	13	9	25	29	21	17
11	3	7	15	11	27	31	23	19
10	2	6	14	10	26	30	22	18

4-Variable K-Map | 4-Variable K-Map.

16, 20, 24 4V KM

08 4V KM

INCOMPLETELY SPECIFIED FUNCTIONS.
(DON'T CARE TERMS)

- When an output value is known for every possible combination of input variables, the function is said to be completely specified.
- When an op value is not known for every combination of input variables, usually because all combinations cannot occur, the function is said to be incompletely specified.
- The minterms or maxterms that are not used as a part of the output function are called don't care terms.
- For ex: Binary to Ex-3 BCD as shown in Table.
- BCD codes including Ex-3 can represent only a single digit decimal character (0 to 9)
- The decimal number 9 takes 4-bits in Ex-3 BCD code.
- Input combinations (10, 11, 12, 13, 14, 15) in decimal are not used to specify any output variable

value. outputs A, B, C, D are incompletely specified for those input code values that are don't care terms.

Truth table for Binary to Ex-3 BCD code conversion.

Binary	Ex-3 BCD
WXYZ	ABCD
0000	0011
0001	0100
0010	0101
0011	0110
0100	0111
0101	1000
0110	1001
0111	1010
1000	1011
1001	1100
1010	Don't care
1011	— " —
1100	— " —
1101	— " —
1110	— " —
1111	— " —

- Writing the equations for output variables A, B, C, D, including the don't care terms, we get.

$$A = f(w,x,y,z) = \sum(5,6,7,8,9) + \sum d(10,11,12,13,14,15)$$

$$B = f(w,x,y,z) = \sum(1,2,3,4,9) + \sum d(10,11,12,13,14,15)$$

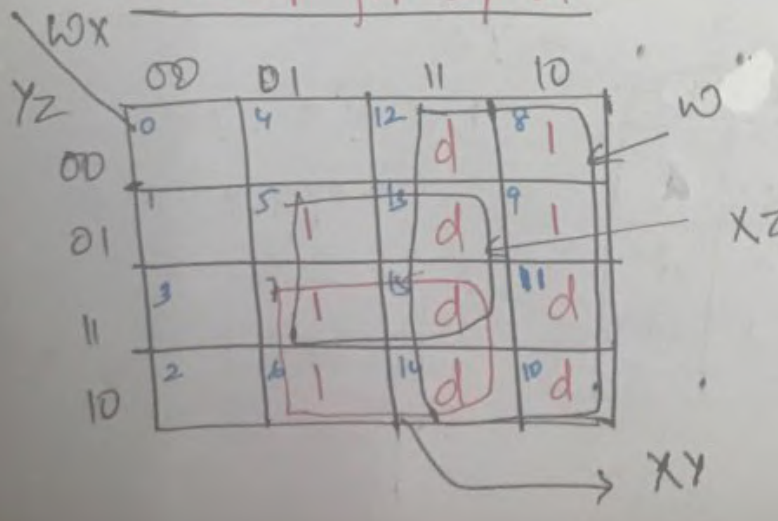
$$C = f(w,x,y,z) = \sum(0,3,4,7,8) + \sum d(10,11,12,13,14,15)$$

$$D = f(w,x,y,z) = \sum(0,2,4,6,8) + \sum d(10,11,12,13,14,15)$$

- Don't care terms are distinguished from regular minterms in that it doesn't matter whether we assign them a value of 0 or 1, because these input combinations never occur.

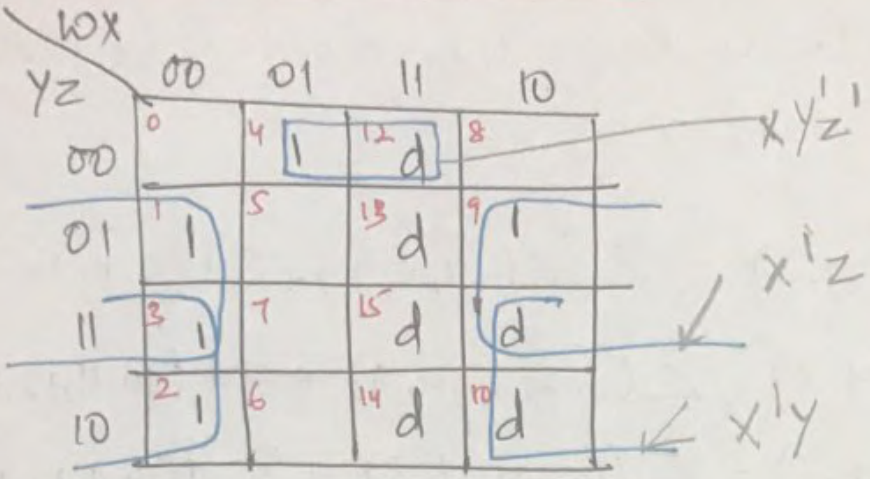
- Don't care terms are indicated by a d in each of the K-maps.

K-Map for op A



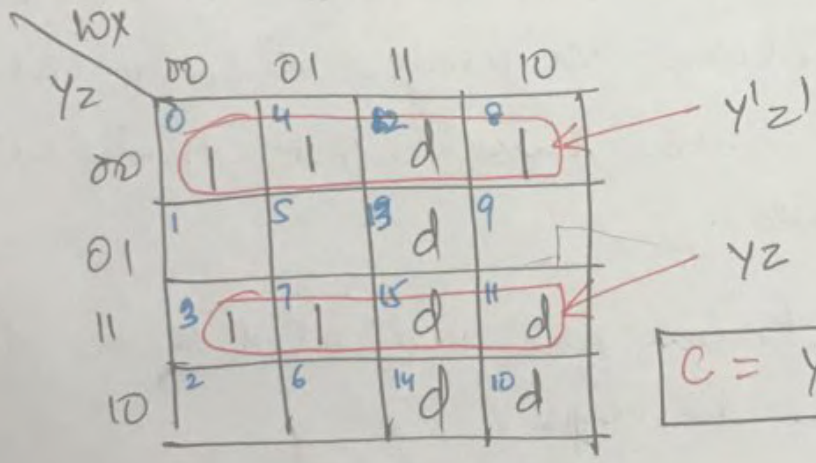
$$\therefore A = w + xz + xy$$

K-Map for op B.



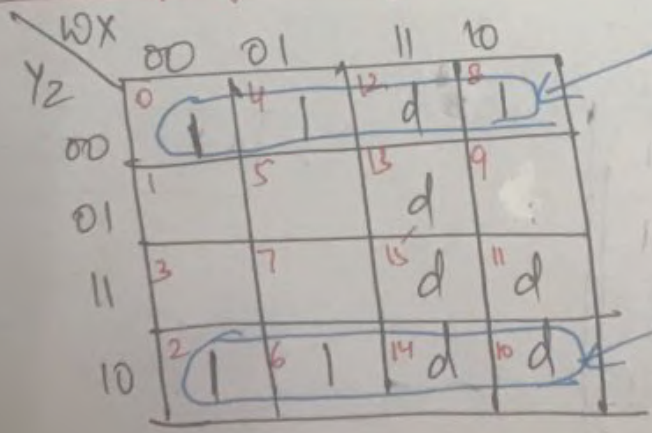
$$B = xy'z' + x'z + x'y.$$

K-Map for op C.



$$C = yz + y'z' = y \odot z.$$

K-Map for op D.



$D = y'z' + yz'$

$= (y + y')z' = z'.$

$$D = z'$$

Procedure for the determination and use of don't care terms.

① → Develop the truth table that describes the input/output relationship.

② → Determine if all of the input combinations are used to generate the ops.

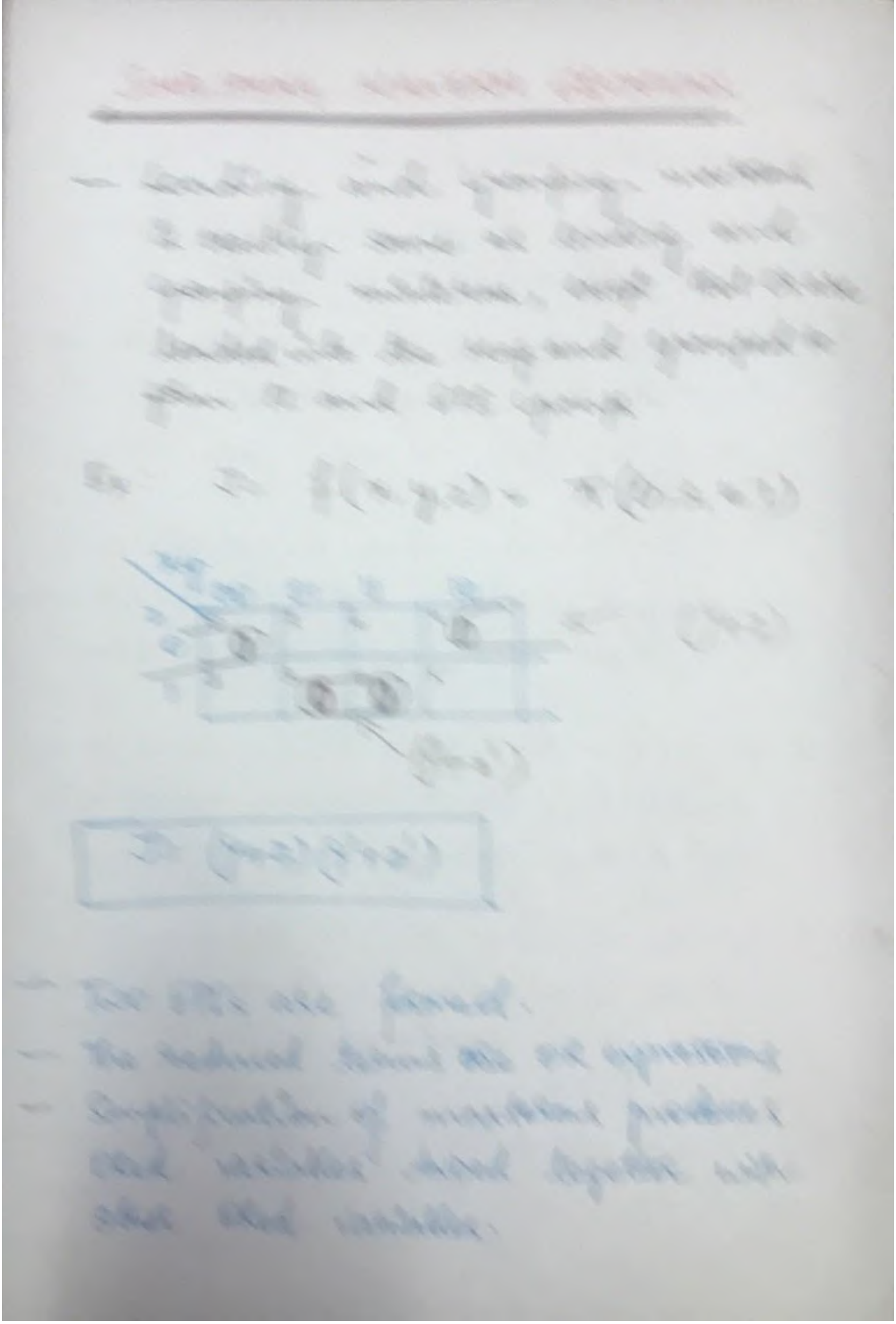
(a) If so, then no don't care terms exist.

(b) If not, then those combinations of input variables not used to determine output values are don't care terms.

③ → Once the don't care terms have been identified, use a separate symbol, in the K-Map squares.

④ → Create as large an EPI grouping as possible, including don't care terms that have been combined with normal minterms.

⑤ → Do not group don't care terms by themselves. A don't care PI is meaningless, because don't care terms are not used to generate the output function.



QUINE - Mc CLUSKEY MINIMIZATION TECHNIQUE.

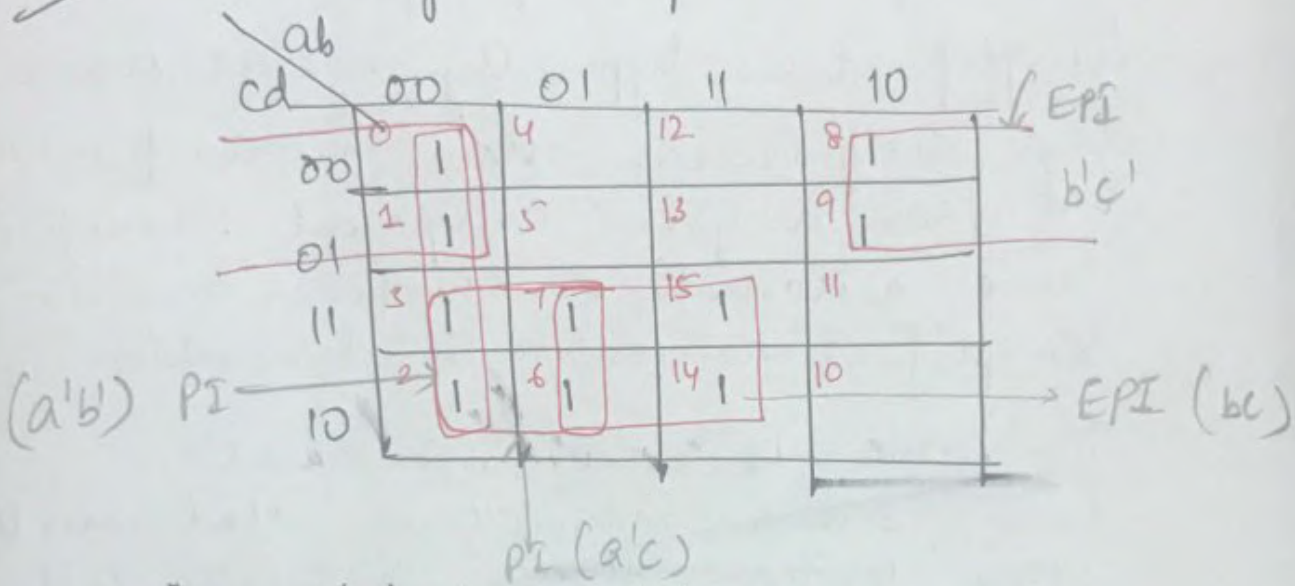
- For many applications, the no. of variables in a problem is too large to simplify manually using K-Maps.
- Simplification typically means that a logic designer can obtain more functional use from a given component. Therefore some automatic or computer driven simplification routine is desirable.
- The QUINE - McCLUSKEY minimization technique is an algorithm that uses the same boolean Algebra postulates that were used with Karnaugh Maps but in a form suitable for a computer solution.
  - Large K-Maps require recognition of group of terms that may form essential prime implicants. The larger the map, the more difficult this pattern recognition becomes.
  - The QM approach eliminates the need for such pattern recognition.

Ex: Simplify the following using the QM Minimization technique.

$$D = f(a,b,c,d) = \Sigma(0,1,2,3,6,7,8,9,14,15).$$

Soln Case 1

Using K-Map.



$$D = b'c' + bc + a'b'$$

or

$$D = b'c' + bc + a'c.$$

Case 2 : QM Technique.

Step 1: Arrange all of the minterms, in a list of increasing order, so that group of terms contain the same no. of 1s.

- Each minterm in the original expression is arranged in increasing order according to no. of 1's contained each group.

- Group 0 contains no. 1s., group 1 contains only those minterms that have single 1 (1,2,8), group 2 contains minterms with two 1s (3,6,9), group 3 contains minterms with three 1s (7,14) and group 4 contains minterms with four 1's (15).

group 1

group 0 → 0 → 0000, 1 → 0001, 2 → 0010, 8 → 1000

3 → 0011, 6 → 0110, 9 → 1001 ← group 2

7 → 0111, 14 → 1110 ← group 3

15 → 1111 ← group 4.

Grouping minterms according to no. of 1s.

Group	Minterm	a b c d
0	0	0 0 0 0
1	1	0 0 0 1
1	2	0 0 1 0
1	8	1 0 0 0
2	3	0 0 1 1
	6	0 1 1 0
	9	1 0 0 1
3	7	0 1 1 1
	14	1 1 1 0
4	15	1 1 1 1

Table 1.

Step 2 : Create a new table showing the minterms in group n (ex: 0) that matched with those from group n+1 (1) such that they differ in only one position.

- Compare minterm {0} in group 0 of Table 1 with each of minterms in group 1. When all the minterms in group 0 have been compared with those in group 1.
- Compare the minterms in group 1 with those in group 2.
- This process is repeated until all of the minterms in each group have been compared to those of in the next higher group.
- When a minterm in a group combined with a minterm in adjacent group, a dash (-) is used to indicate an eliminated variable.
- The combined minterms are Grouped together in Table 2. - Each combination results in a new minterm group.

Table 2: Creation of minterm groups of two.

Group	Minterm	a b c d
0	0,1	0 0 0 - ✓
0	0,2	0 0 - 0 ✓
0	0,8	- 0 0 0 ✓
1	1,3	0 0 - 1 ✓
	1,9	- 0 0 1 ✓
	2,3	0 0 1 - ✓
	2,6	0 - 1 0 ✓
	8,9	1 0 0 - ✓
2	3,7	0 - 1 1 ✓
	6,7	0 1 1 - ✓
	6,14	- 1 1 0 ✓
3	7,15	- 1 1 1 ✓
	14,15	1 1 1 - ✓

- As each minterm, combines with a minterm in the next higher group is checked (✓), indicating that it is now part of a larger group.

Step 3 : All of the adjacent minterm groups contained in Table ② are compared to see if groups of 4 can be made.

- The criteria for forming groups of 4 are as follows.
  (a) The dashes in the groups of two must be in the same bit position and
  (b) only one variable change is allowed.
- A comparison is made of each minterm in group n with each minterm in group n+1. Those that meet the criteria are combined in a larger group as shown in Table ③.

Group	Minterm	a b c d
0	0, 1, 2, 3	0 0 – –
0	0, 1, 8, 9	– 0 0 –
1	2, 6, 3, 7	0 – 1 –
2	6, 7, 14, 15	– 1 1 –

Table 3 : Creation of Minterms of group of 4.

Step 4: Minterms {0,1} in group 0 is compared with {2,3} in group 1, to form a group {0,1,2,3}

- If the dashes (-) are in the same position and only one variable changes then a newer group is created.
- This process is repeated until no further combination is possible. That is each minterm in group n is compared with each minterm group in n+1 in Table ②.

Step 5 → All nonchecked minterm groups are now considered as PIs.

Step 6 → All of the PIs are formed into a PI Table as shown in Table ④.

PI terms	Decimal	0	1	2	3	6	7	8	9	14	15
$a'b'$	0,1,2,3	x	x	x	x						
$b'c'$	0,1,8,9	x	x					(x)	(x)		
$a'c$	2,6,3,7			x	x	x	x				
bc	6,7,14,15					x	x			(x)	(x)

Table 4: Prime Implicant Table.

- The prime implicant table lists each of the minterms contained in the original switching equation across the top of the table.

- Each PI is listed vertically in 2 forms. PI terms and the decimal list of minterms that make up the PI.

Step 7 → Evaluate the prime implicants by circling those minterms that are contained in only one PI (only one x in a column).

- The minterms (8, 9, 14, 15) meet this condition → Essential PI.

- (0, 1, 8, 9) & {6, 7, 14, 15} are EPI.

- minterms {2, 3} are contained in 2 PIs {0, 1, 2 3} & {2, 3, 6, 7}. we need one or the other of these PIs to cover minterms in the equation but not both.

$$\therefore \boxed{\begin{array}{c} D = b'c' + bc + a'b' \\ \text{or} \\ D = b'c' + bc + a'c \end{array}}$$

QUINE-Mc. CLUSKY USING DON'T CARE TERMS

- The same rules that applied to using don't care terms with the K-Map are appropriate for QM technique.

Ex: $S = f(w, x, y, z) = \sum(1, 3, 13, 15) + \sum d(8, 9, 10, 11)$

Step 1: Construct a list of minterms and don't care terms classified according to the number of 1s. Indicate the don't care terms by using a * symbol. Don't care terms are never included as prime implicants by themselves.

group 1	group 2	group 3	group 4
1 → 0001	3 → 0011	13 → 1101	15 → 1111
8* → 1000	9* → 1001	11* → 1011	
	10* → 1100		

Group	Minterm	w x y z
1	1	0 0 0 1
	8*	1 0 0 0
2	3	0 0 1 1
	9*	1 0 0 1
	10*	1 1 0 0
3	11*	1 0 1 1
	13	1 1 0 1
4	15	1 1 1 1

Table 1: Grouping of Minterms.

Step 2 : compare terms in group n, including don't care terms in group. n+1, for a single variable change.

- Treat don't care terms as a 1 in finding PIs.

group	Minterms	w x y z	
1	(1, 3)	0 0 - 1	✓
1	(1, 9*)	- 0 0 1	✓
1	(8, 9*)	1 0 0 -	✓
1	(8, 10*)	1 0 - 0	✓
2	(3, 11*)	- 0 1 1	✓
2	(9* 11*)	1 0 - 1	✓
2	(9* 13)	1 - 0 1	✓
2	(10*, 11*)	1 0 1 -	✓
3	(11*, 15)	1 - 1 1	✓
3	(13, 15)	1 1 - 1	✓

Table 2 : Minterm groups of two.

Step 3: Repeat Step 2 to create a table of group of 4 minterm and don't care terms. Repeat Step 3 until no further grouping can occur.

Group	Minterm	w x y z
1	1, 3, 9*, 11*	- 0 - 1
1	8*, 9*, 10*, 11*	1 0 - -
2	9*, 13, 11*, 15	1 - - 1

Table 3: Minterms groups of 4.

- Table 3 represents group of 4 Minterms and don't care terms. Each term is EPI. But (8*, 9*, 10*, 11*) contains only don't care terms ∴ not a PI.

Step 4: Creation of PI Table

PI terms	Decimal	1	3	13	15	← Minterms
x'z	1 3 9* 11*	⊗	⊗			→ EPI
wz	9* 11* 13 15			⊗	⊗	→ EPI

∴ $S = wz + x'z$

Problems:

-1. Place the following equation into proper canonical form.

a) $P = f(a,b,c) = ab' + ac' + bc$ (SOP)

Sol^{n} $P = ab'(c+c') + ac'(b+b') + bc(a+a')$

$= ab'c + ab'c' + abc' + ab'c' + abc + a'bc$

$\therefore P = ab'c + ab'c' + abc' + abc + a'bc.$

b) $G = f(w,x,y,z) = w'x + yz'$ (SOP)

$G = w'x(y+y')(z+z') + yz'(x+x')(w+w')$

$= xyw'z + xyw'z' + xy'w'z' + xy'wz'$

$+ xywz' + xyw'z' + x'yz' + x'yw'z'$

$= xyw'z + xyw'z' + xy'w'z' + xyw'z' + xywz'$

$+ x'ywz' + x'yw'z'$

(c) $T = f(a,b,c) = (a+b')(b'+c)$ (Pos)

Soln $T = (a+b')(b'+c)$

$= (a+b'+cc')(aa'+b'+c)$

$= (a+b'+c)(a+b'+c')(a+b'+c)(a'+b'+c)$

$= (a+b'+c)(a+b'+c')(a'+b'+c)$

(d) $J = f(A,B,C,D) = (A+B'+C)(A'+D)$ Pos.

$J = (A+B'+C)(A'+D)$

$= (A+B'+C+DD')(A'+BB'+CC'+D)$

$= (A+B'+C+D)(A+B'+C+D')$
$(A'+B+C+D)(A'+B+C'+D)$ ~~$(A'+B'+C+D)$~~
$(A'+B'+C+D)(A'+B'+C'+D)$ ~~$(A'+B'+C+D)$~~

2. Write the canonical minterm and maxterm expressions for the following table

(a)

ip				op
a	b	m	s	M
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Minterm Expression

$M = f(a, b, m, s)$

$= \sum(7, 11, 15)$

$= a'bms + ab'ms + abm$

Maxterm Expn

$M = f(a, b, m, s)$

$= \pi(0, 1, 2, 3, 4, 5, 6, 8, 9, 10$

(b)

	Inputs ←			outputs →				
	c_1	c_2	c_3	o_1	o_2	o_3	w_1	w_2
0	0	0	0	0	1	1	0	0
1	0	0	1	1	1	0	0	0
2	0	1	0	1	0	1	0	0
3	0	1	1	1	0	0	1	0
4	1	0	0	0	1	1	0	0
5	1	0	1	0	1	0	1	0
6	1	1	0	0	0	1	1	0
7	1	1	1	0	0	0	0	1

Minterm Expressions

$$O_1 = f(c_1, c_2, c_3) = c_1'c_2'c_3 + c_1'c_2c_3' + c_1'c_2c_3$$
$$= \Sigma(1, 2, 3)$$

$$O_2 = f(c_1, c_2, c_3) = c_1'c_2'c_3' + c_1'c_2'c_3 + c_1c_2'c_3' + c_1c_2'c_3$$
$$\Sigma(0, 1, 4, 5)$$

$$O_3 = f(c_1, c_2, c_3) = c_1'c_2'c_3' + c_1'c_2c_3' + c_1c_2'c_3' + c_1c_2c_3'$$
$$\Sigma(0, 2, 4, 6)$$

$$w_1 = f(c_1, c_2, c_3) = c_1'c_2c_3 + c_1c_2'c_3 + c_1c_2c_3'$$
$$\Sigma(3, 5, 6)$$

$$w_2 = f(c_1, c_2, c_3) = c_1c_2c_3$$
$$\Sigma 7$$

Maxterm Expressions

$O_1 = f(C_1 C_2 C_3)_4 = \pi(0,4,5,6,7)$

$O_2 = f(C_1, C_2 C_3) = \pi(2,3,6,7)$

$O_3 = f(C_1, C_2, C_3) = \pi(1,3,5,7)$

$W_1 = f(C_1, C_2, C_3) = \pi(0,1,2,4,7)$

$W_2 = f(C_1, C_2, C_3) = \pi(0,1,2,3,4,5,6)$

3. Express the following SOP equations in a minterm list.

a. $H = f(A,B,C) = \underset{011}{A'BC} + \underset{001}{AB'C} + \underset{111}{ABC}$

$= \Sigma(3,1,7)$

b. $G = f(w,x,y,z) = \underset{1110}{wxyz'} + \underset{1010}{wx'yz'} + \underset{0110}{w'xyz'}$

$+ \underset{0010}{w'x'yz'}$

$= \Sigma(2,6,10,14)$

4. Express the following Pos equation in a maxterm list.

a. $T = f(a,b,c) = \underset{010}{(a+b'+c)}\ \underset{011}{(a+b'+c')}\ \underset{110}{(a'+b'+c)}$

$= \pi(2,3,6)$

b. $J = f(A\,B\,C\,D) = (A+B'+C+D)\,(A'+B+C+D)$
$(A+B'+C+D')\,(A'+B'+C+D)$
$(A'+B+C'+D)\,(A'+B'+C'+D)$

$= \pi(4, 5, 8, 10, 12, 14)$

5. Write the POS & SOP equations for A & A' for the following Truth table.

	X YZ	A	A'
0	000	0	1
1	001	0	1
2	010	1	0
3	011	0	1
4	100	1	0
5	101	1	0
6	110	0	1
7	111	0	1

Sop for A

$A = \Sigma(2, 4, 5)$

Sop fo A'

$A' = \Sigma(0, 1, 3, 6, 7)$

Pos for A

$A = \pi(0, 1, 3, 6, 7)$

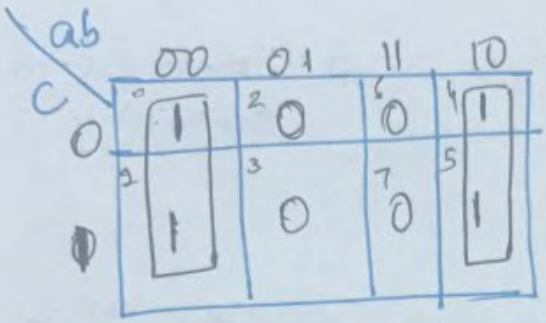
Sop for A' $\rightarrow$ $A' = \pi(2, 4, 5)$

Problems on K-Map.

① Simplify the Boolean function using K-map.

$$Y = f(a,b,c) = \sum 0,1,4,5$$

Soln



$Y = a'bc' + a'b'c + ab'c' + ab'c$

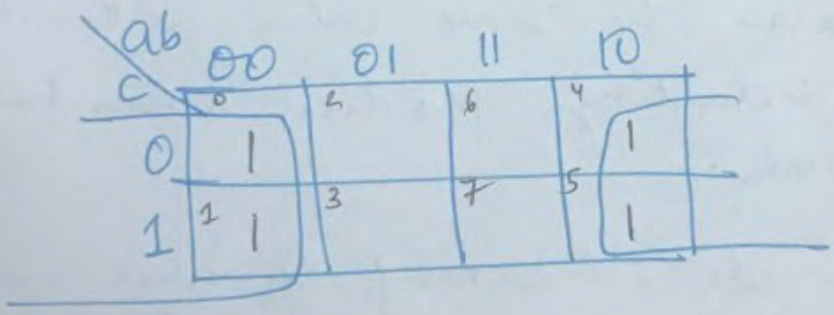
$= a'b'(c+c') + ab'(c+c')$

$= a'b' + ab'$

- Two groups of minterms

$$Y = a'b' + ab'$$

or



one group of 4 minterms.

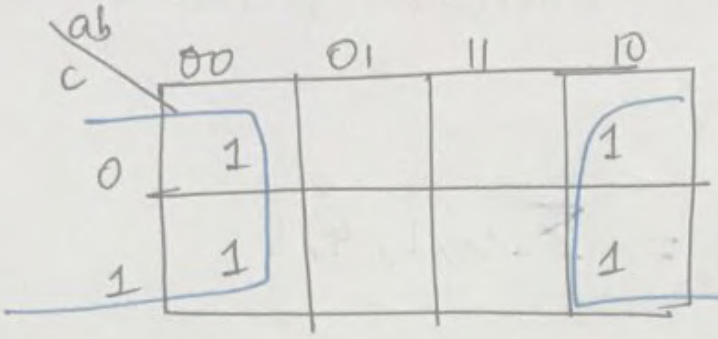
$$Y = b'$$

$Y = a'b'c' + a'b'c + ab'c' + ab'c$

$= a'b'(c+c') + ab'(c+c')$

$= a'b' + ab' = b'(a+a') = b'$

3- variable K-Map showing Prime Implicant PI [0,1,4,5].



2^n K-Map
$n=3$, $m=2$
(0,1,4,5) forms a prime implicant

#literals [$n-m = 3-2 = 1$] remaining

$Y = b'$

- A set of 2^n K-Map squares are combined to form a Prime Implicant, If n-variables of the equation being simplified have a 2^n permutations within the set and the remaining m-n variables have the same value with in the set. The resulting product term has m-n literals.

- In otherwards 2^n K-map squares may be combined to form a group of minterms containing m-n literals, where m represents the no. of variables in the original equation.

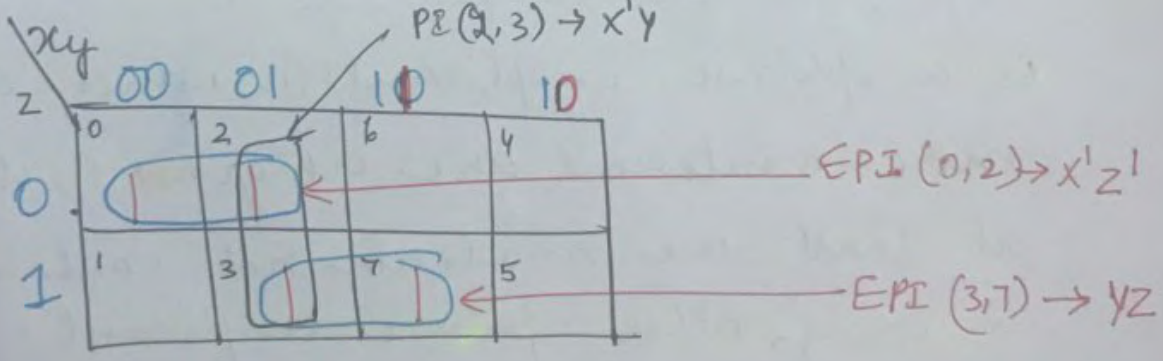
- Any single minterm or permitted group of minterms is called an implicant of an output function.

- A prime implicant is a group of minterms that cannot be combined with any other minterms or groups.

- Minterms (0,1,4,5) are each implicants of y. The prime implicants consist of the 4 minterms grouped together.

(2) $G = f(x,y,z) = \sum(0,2,3,7)$

Soln



- Expression G is loaded into the K-Map.
- 3 groups of minterms are found (0,2), (2,3) & (3,7)

- Each of the 3 groups form a Prime implicant.

  Prime Implicant (0,2) reduces to $x'z'$
  Illy (2,3) reduced to $x'y$
  & (3,7) reduces to yz.

- Groups (0,2) and (3,7) both contain a minterm that is not in any other group.

  Minterm (0) is unique to PI (0,2)
  Minterm (7) is unique to PI (3,7).

- An Essential Prime Implicant [EPI] is a prime implicant in which one or more minterms are unique i.e, it contains at least one minterm not contained in any other prime implicant.

- The simplified expression for G includes only the essential prime implicants. The non essential prime implicant (2,3) term is redundant because all of its minterms are covered by the

two essential prime implicants.

The simplified expression is

$$G = x'z' + yz$$

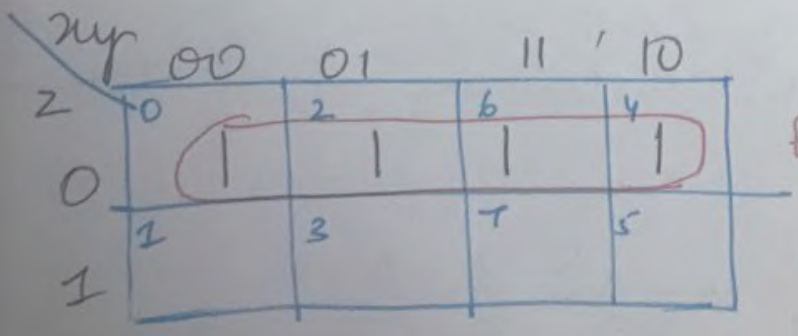
- Procedure for finding essential prime implicants is this

(a) find the prime implicants by finding all permitted maximum sized groups of minterms.

(b) find essential prime implicants by identifying those prime implicants that contain at least one minterm not found in any other prime implicant.

(3)

$$D = f(x, y, z) = \sum 0, 2, 4, 6.$$

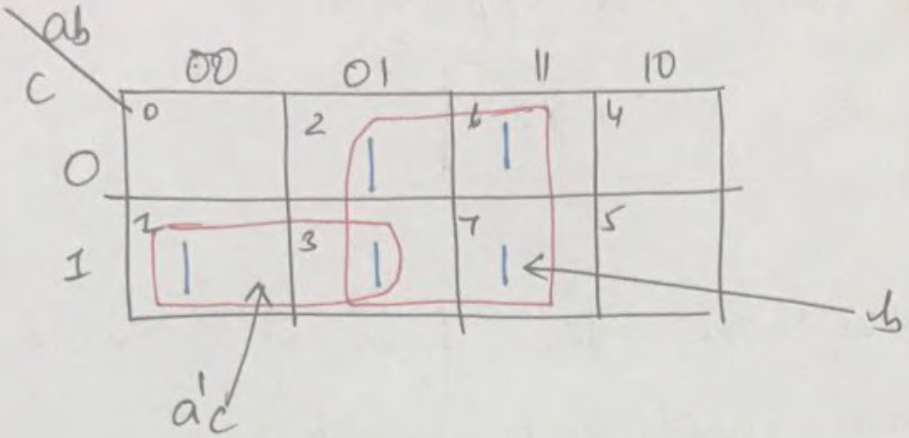


EPI (0,2,4,6) → z'

$$D = z'$$

(4) $Q = f(a,b,c) = \sum 1, 2, 3, 6, 7$

Soln

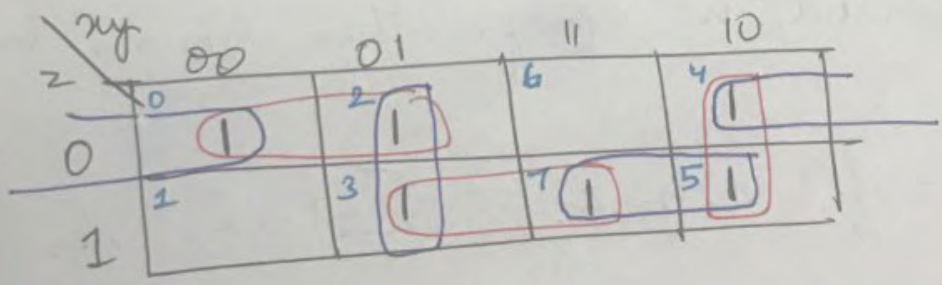


$Q = b + a'c$

– 2 prime Implicants exist (1,3) & (2,3,6,7). Both are essential because each has a unique minterm.

(5) $J = f(x,y,z) = \sum 0, 2, 3, 4, 5, 7$

Soln



– 6 minterms containing 2 minterms exist.

(0 2) → $x'z'$ (3,7) → yz

(0,4) → $y'z'$ (7,5) → xz

(2,3) → $x'y$ (4,5) → xy'

- The final simplified equation is not unique ie, more than one equally simplified result is possible.
- The product terms in the simplified result must covers all of the minterms.
- For ex, we must pick a prime implicant (0,2) or (0,4) to cover the minterm(0).
- TWO equally simplified equations are

$$J = x'z' + xy' + yz$$ which covers (0,2,3,4,5,7)

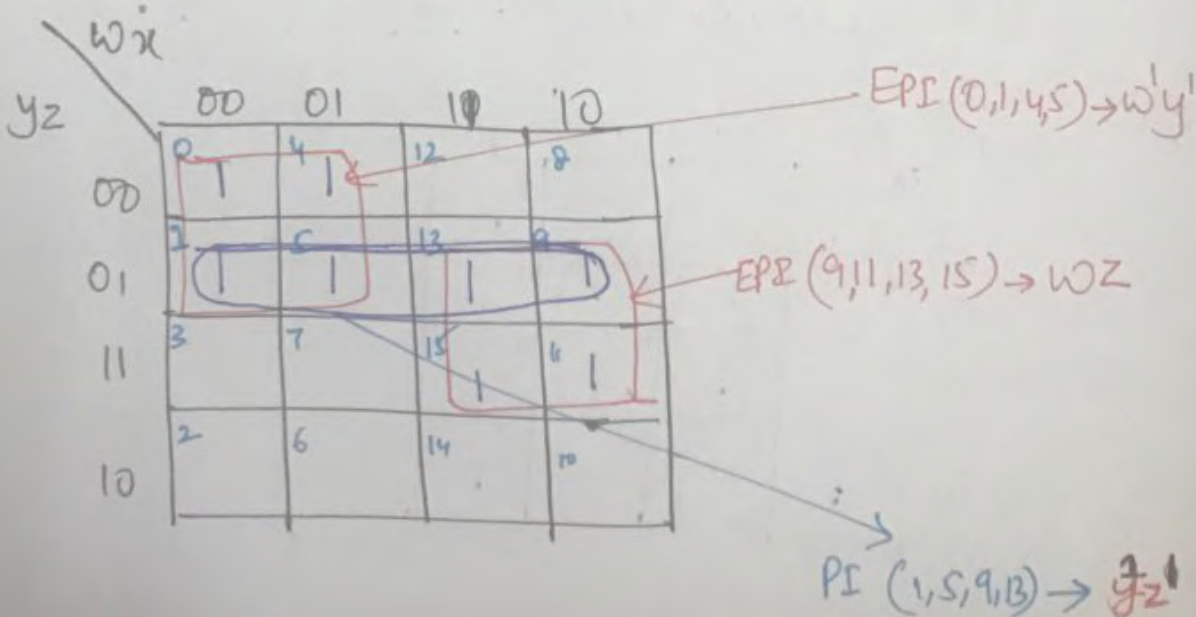
or

$$J = y'z' + x'y + xz$$ which covers (0,2,3,7,4,5)

(6) Simplify the following 4-variable equations.

$$K = f(w,x,y,z) = \sum 0,1,4,5,9,11,13,15$$

Soln



- 3 four variable Prime Implicants are present {0, 1, 4, 5} {9, 11, 13, 15} & {1, 5, 9, 13}

- 2 EPI are present {0, 1, 4, 5}
  {9, 11, 13, 15}

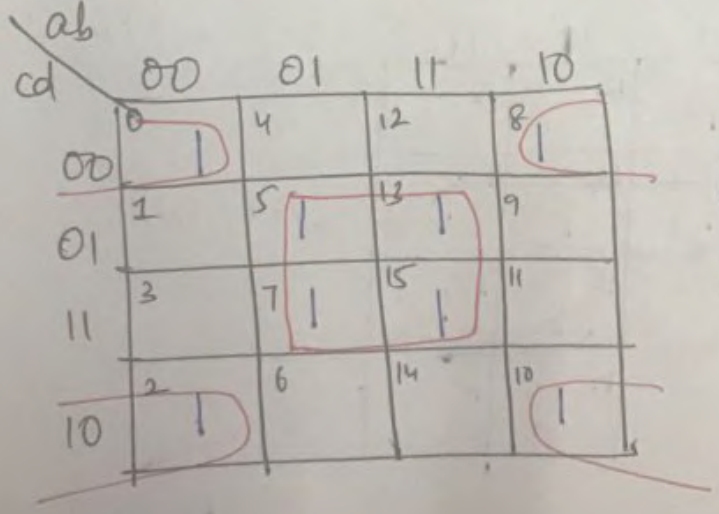
- Prime Implicant {1, 5, 9, 13} is not essential because all of its minterms are contained in other 2 prime Implicants.

- The final simplified eqn is

$$K = w'y' + wz$$

(7) $L = f(a, b, c, d) = \sum 0, 2, 5, 7, 8, 10, 13, 15$

Soln



$L = bd + b'd'$

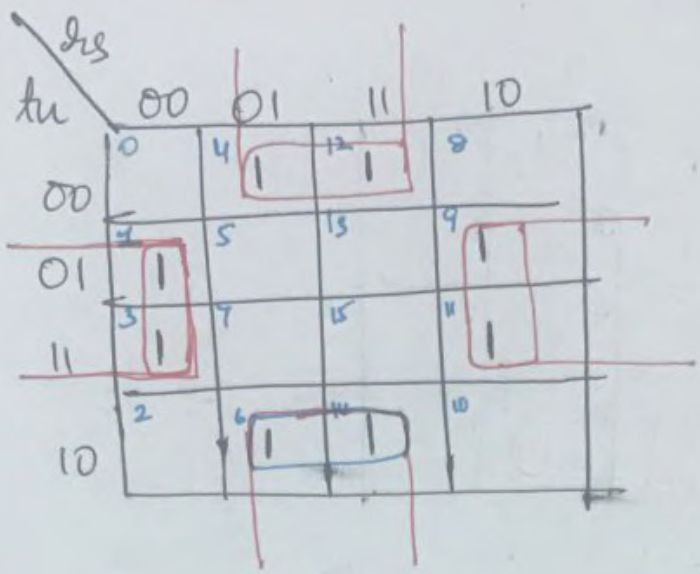
$L = (b \oplus d)'$

$= b \odot d$

(8) (P8)

$P = f(r,s,t,u) = \sum (1,3,4,6,9,11,12,14)$

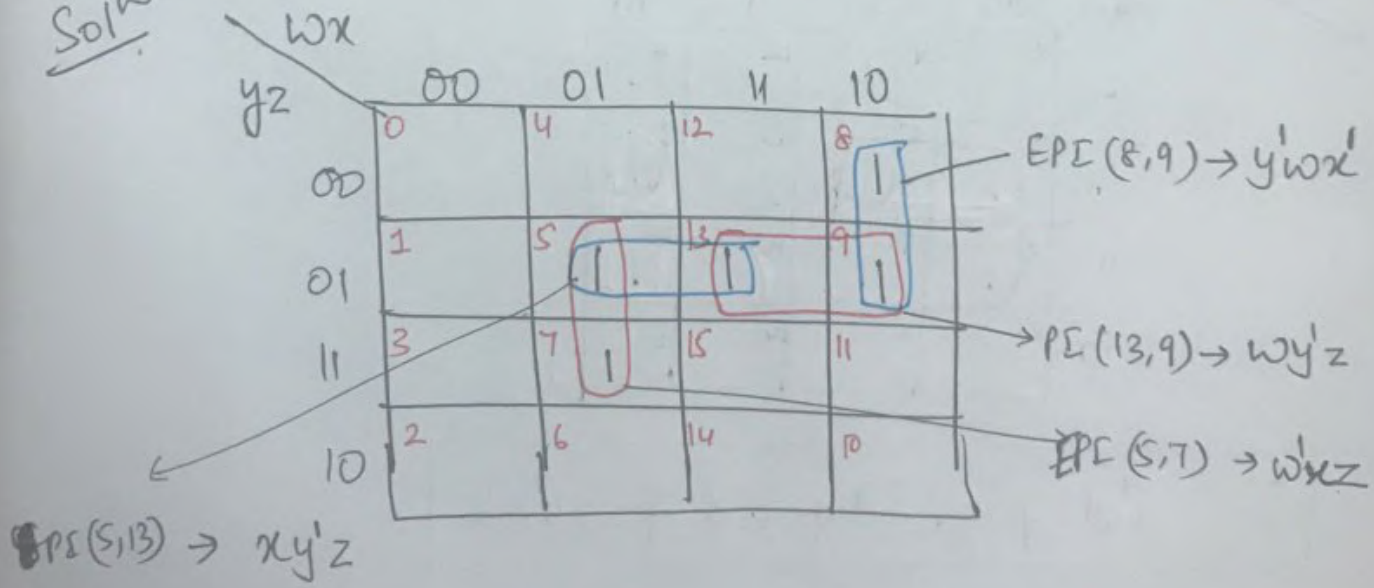
Soln



$$P = su' + \dot{s}u = s \oplus u$$

(9) $D = f(w,x,yz) = \sum (5,7,8,9,13)$

Soln

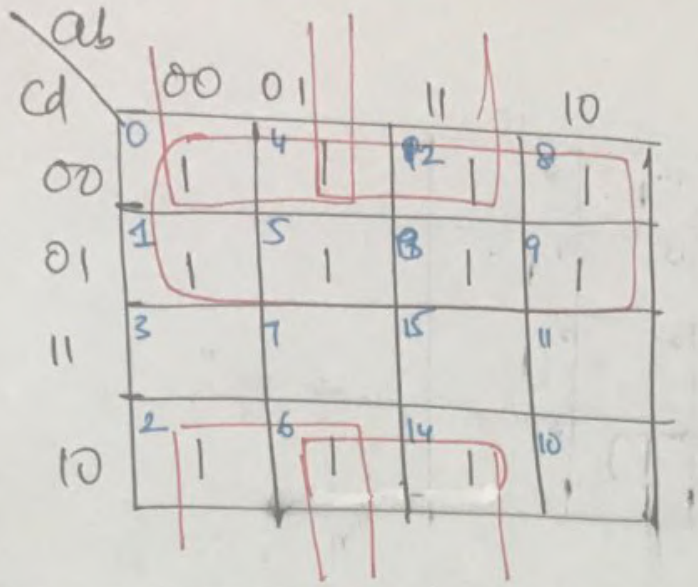


$D = w'xz + wx'y' + xy'z$

$D = w'xz + wx'y' + wy'z$

(10) $P = f(a,b,c,d) = \sum(0,1,2,4,5,6,8,9,12,13,14)$

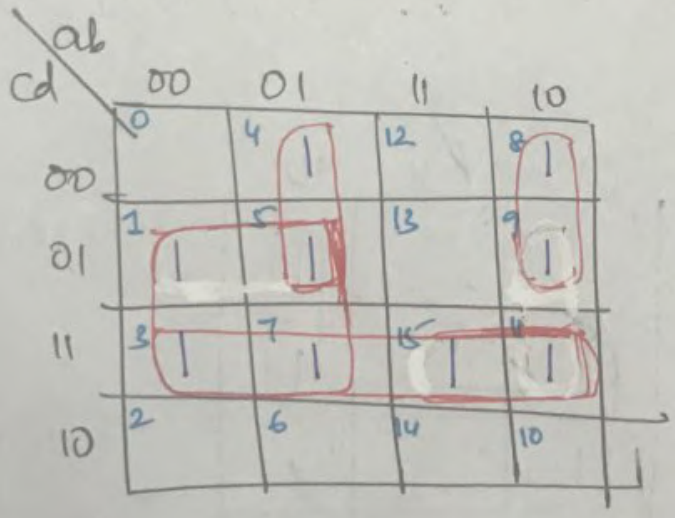
Soln



$$P = c' + a'd' + bd'$$

(11) $S = f(a,b,c,d) = \sum(1,3,4,5,7,8,9,11,15)$

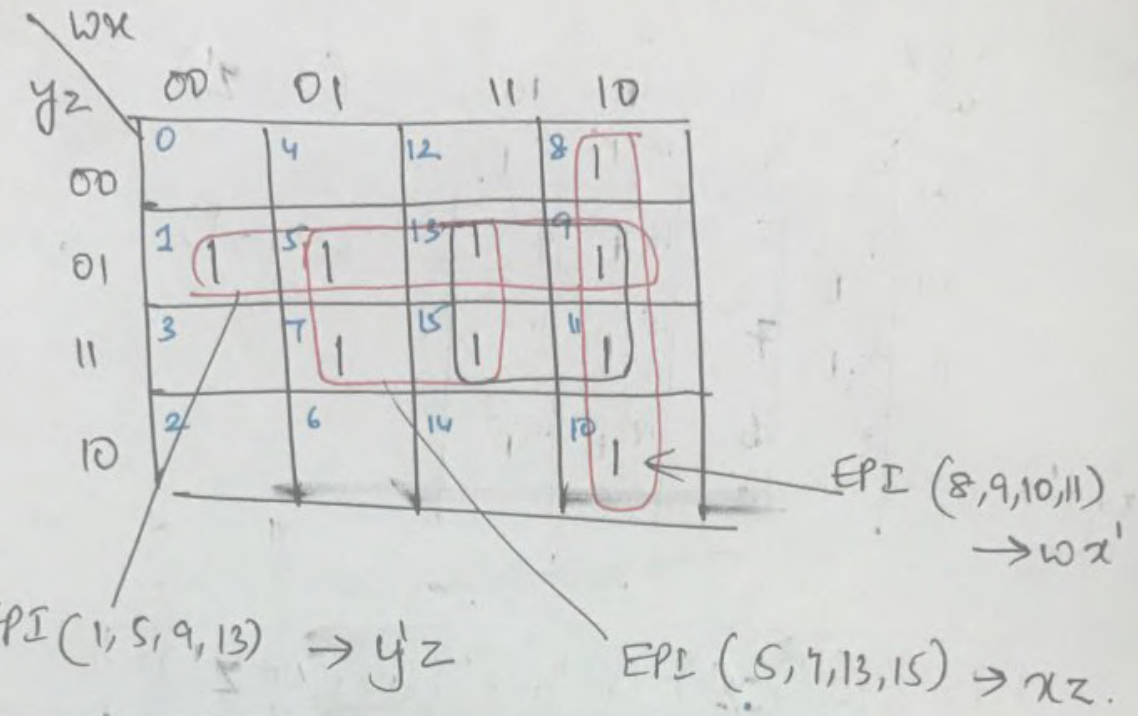
Soln



$$S = a'd + a'bc' + cd + ab'c'$$

(12) $f(w,x,y,z) = U = \sum(1,5,7,8,9,10,11,13,15)$

Soln

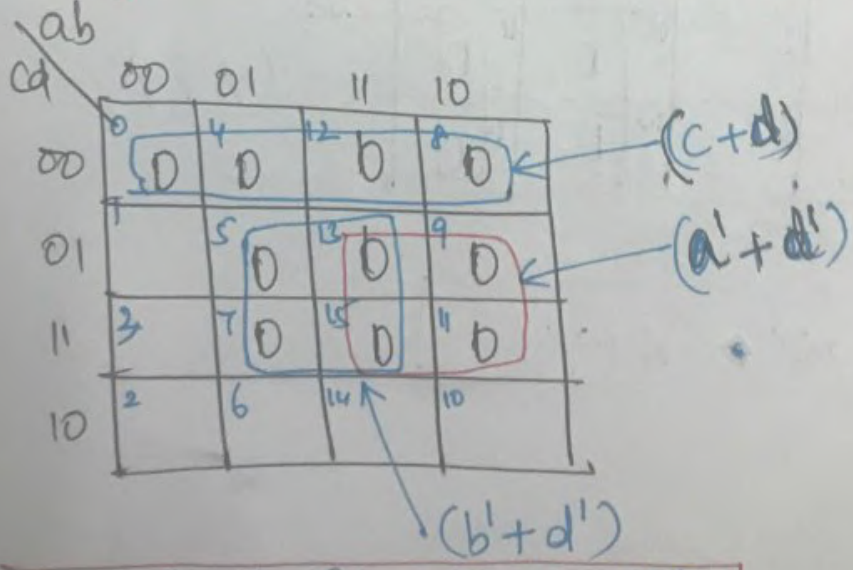


$$U = y'z + wx' + xz.$$

(13) Simplify.

$$G = f(a,b,c,d) = \pi(0,4,5,7,8,9,11,12,13,15)$$

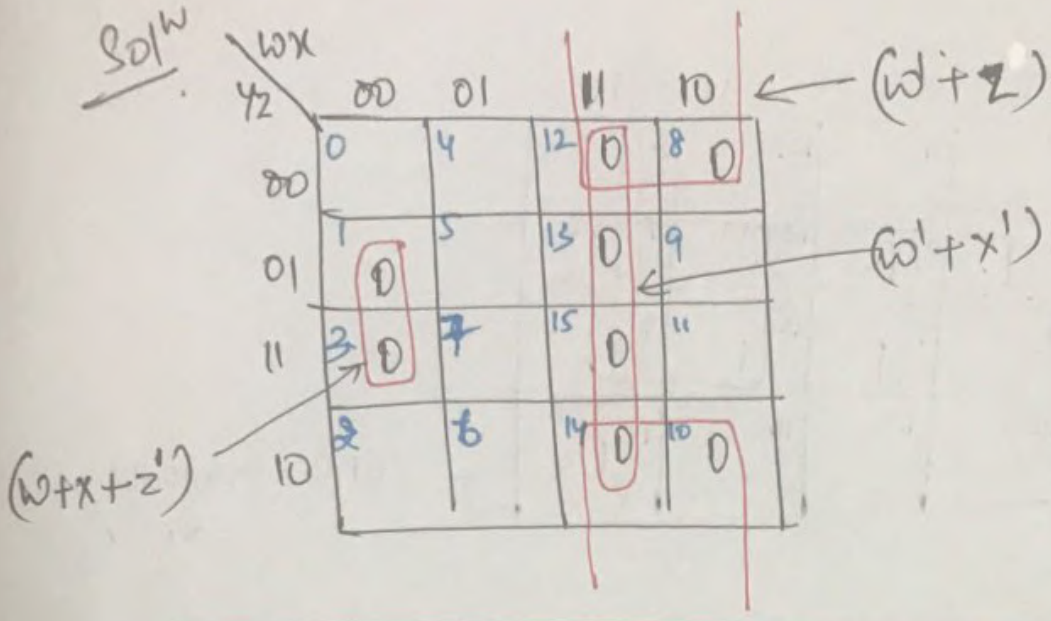
Soln



$$G = (c+d)(b'+d')(a'+d')$$

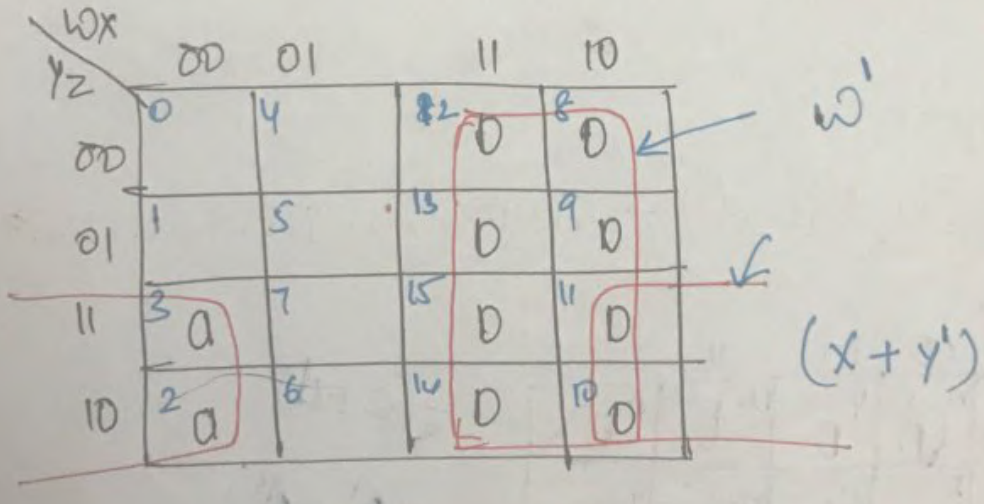
(14) : T= $f(w,x,y,z) = \pi(1,3,8,10,12,13,14,15)$

Sol^n



$T = (w'+z)(w'+x')(w+x+z')$

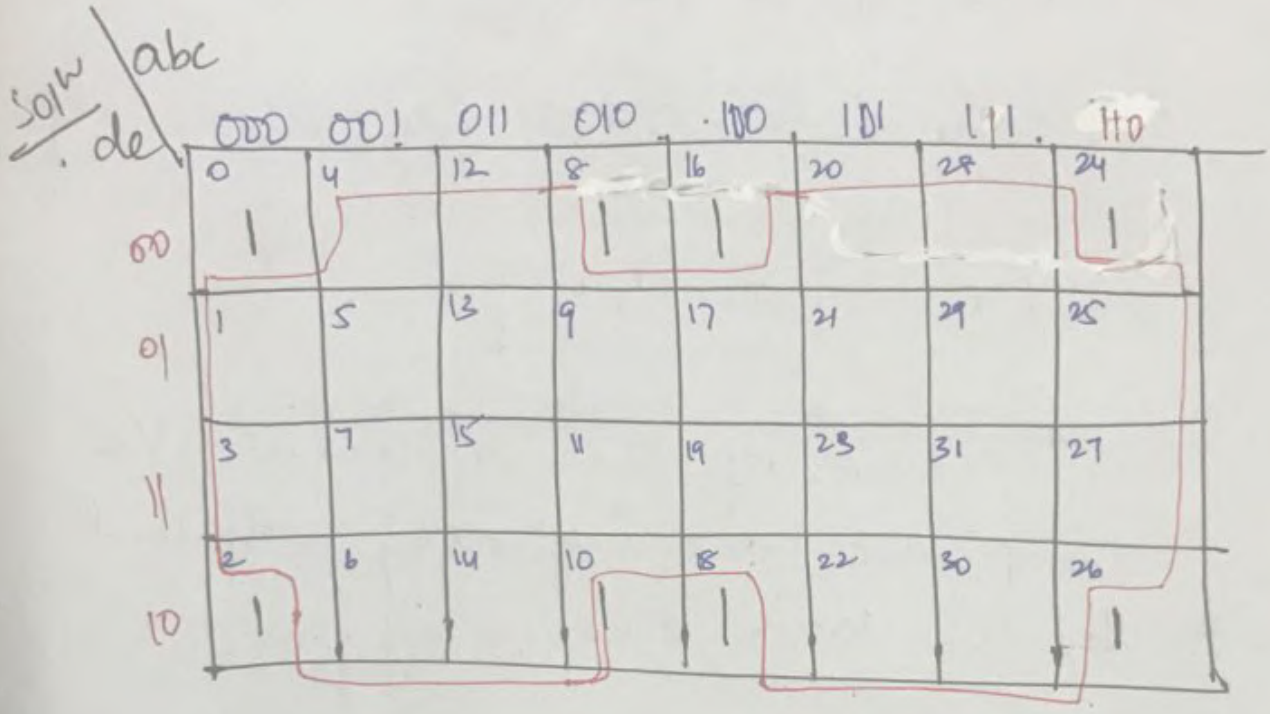
(15) $A = f(w,x,y,z) = \pi(2,3,8,9,10,11,12,13,14,15)$



$A = w' \cdot (x+y')$

(16) Simplify the following 5-variable expression using a K-map.

$T = f(a,b,c,d,e) = \sum(0,2,8,10,16,18,24,26)$



- corners in a 4-variable map are logically adjacent.
- cut the 5-variable map in the center and slide the right half over on top of the left, we can see that 4-variable maps aligned vertically.
- The square of one 4-variable map is logically adjacent to the square in the same relative position on the four variable map.

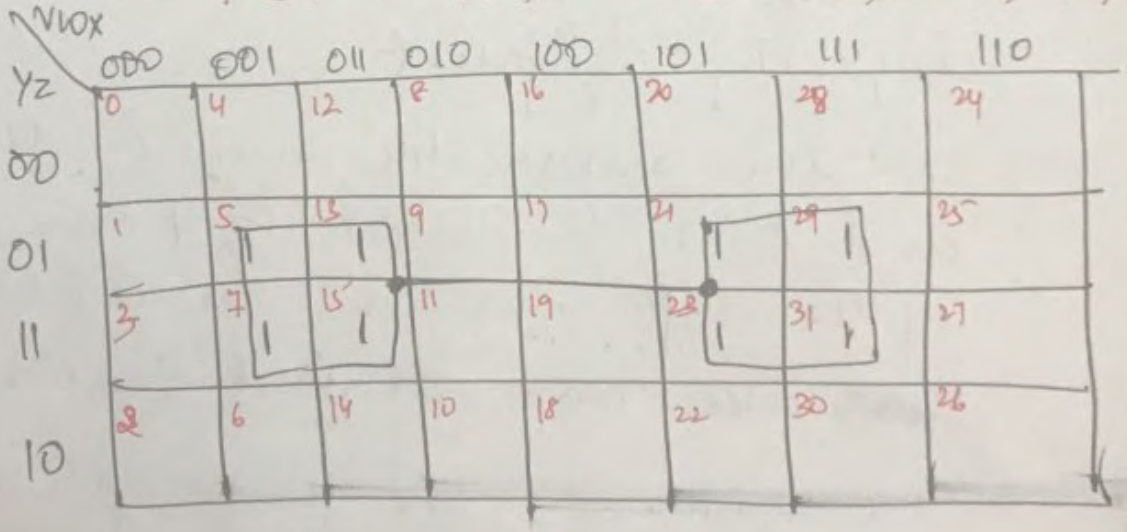
- Minterm {0} is vertically aligned with minterm {16}. Like minterms {2,18}, {8,24} and {10,26} are vertically aligned.

- Minterms located in all corners of each four variable map produces an EPI $T = c'e'$.

XX- The centres of the 2, 4-variable maps are also logically adjacent by the same reasoning.

(17) $R = f(v,w,x,y,z) = \Sigma(5,7,13,15,21,23,29,31)$

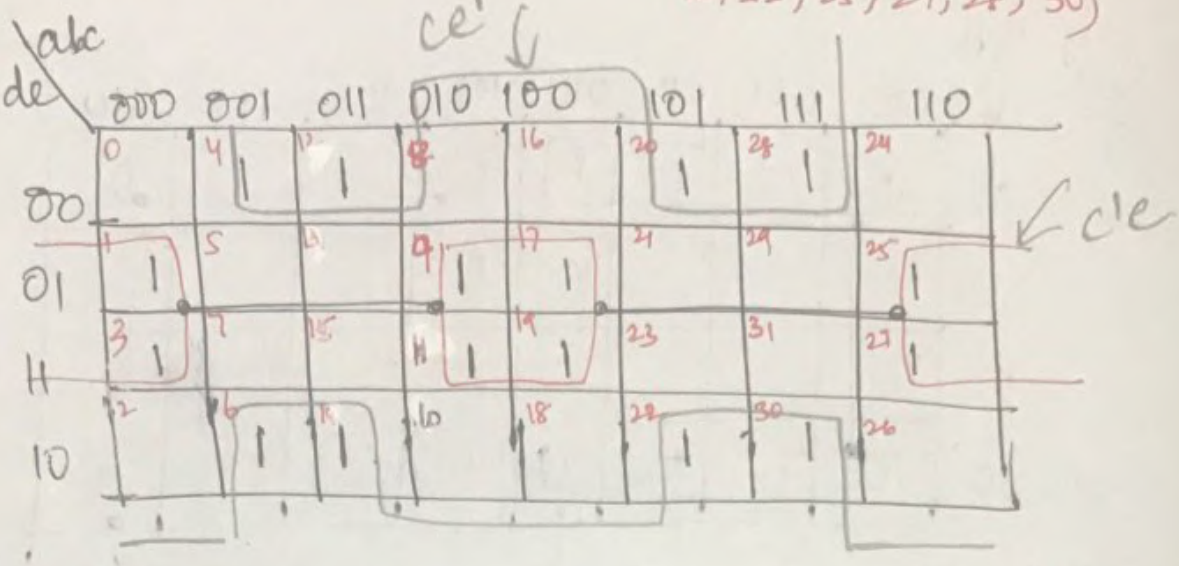
Soln



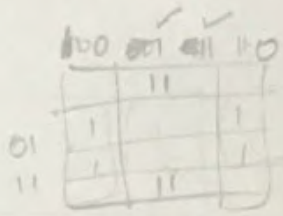
$R = XZ$

(18) $w = f(a,b,c,d,e) = \sum(1,3,4,6,9,11,12,14,17,19,20,22,25,27,28,30)$

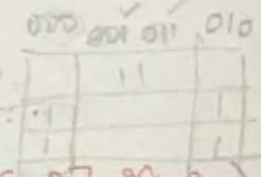
Sol'n



$$w = c'e + ce'$$

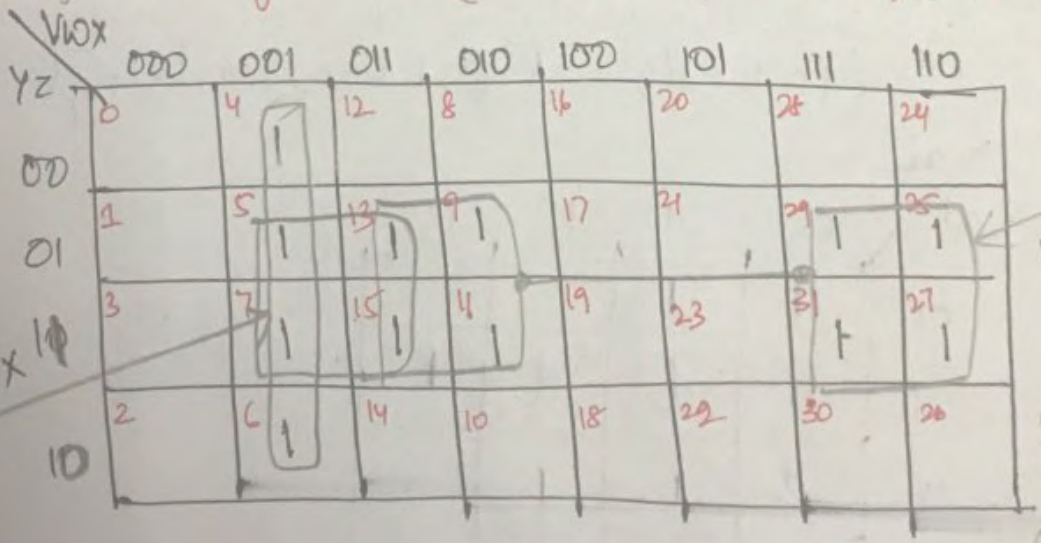


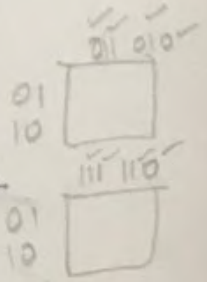
(19)



$$J = f(v,w,x,y,z) = \sum(4,5,6,7,9,11,13,15,25,27,29,31)$$

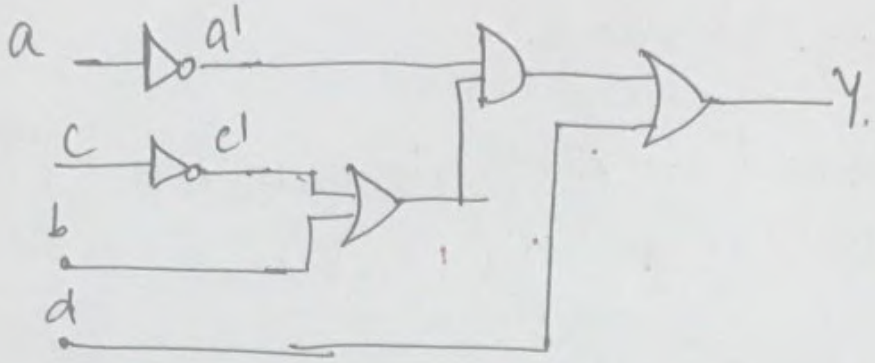
Sol'n





$$J = wz + v'w'x$$

$Y = a'(c'+b) + d$



(Q2) Write the truthtable and design a circuit to generate op using k-map for the problem statement given.

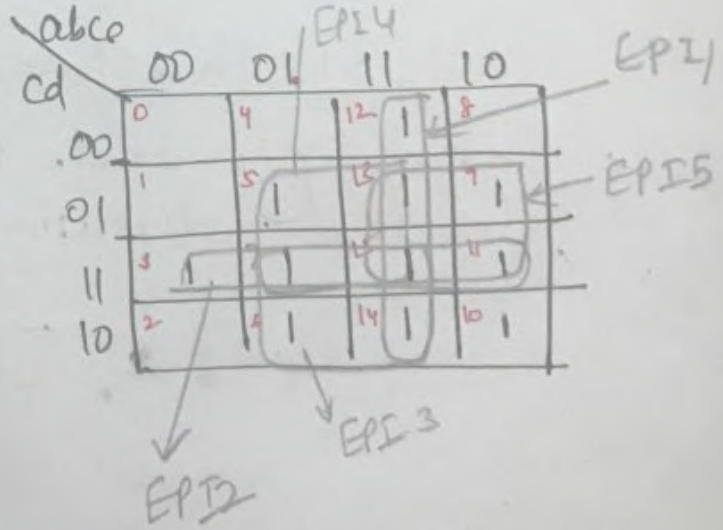
Jan 2017 6M

→ op of a combinational circuit having 4 inputs and an op becomes logical 1 when two or more inputs goto logical 1.

Soln.

	a	b	c	d	Y
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	1
4	0	1	0	0	0
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	1
8	1	0	0	0	0
9	1	0	0	1	1
10	1	0	1	0	1
11	1	0	1	1	1
12	1	1	0	0	1
13	1	1	0	1	1
14	1	1	1	0	1
15	1	1	1	1	1

$Y = \sum(3,5,6,7,9,10,11,12,13,14,15)$



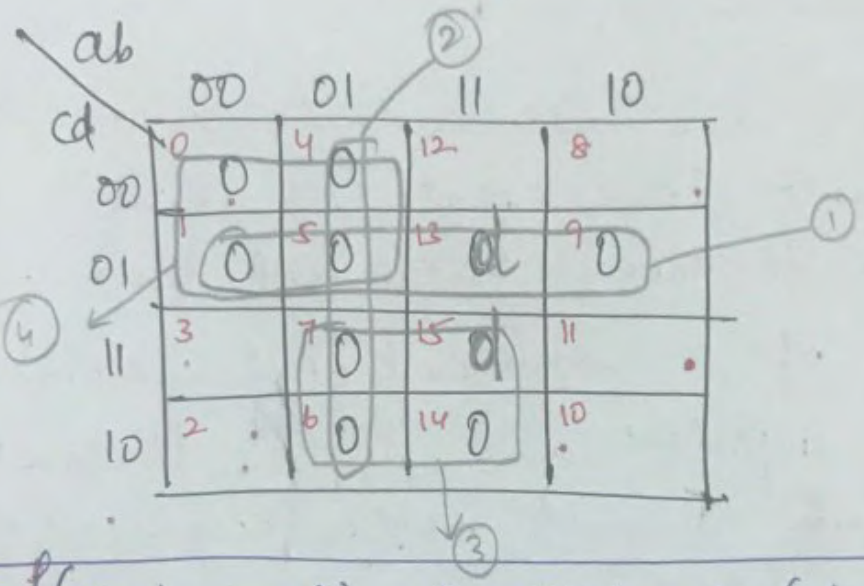
$Y = ab + cd + bc + bd + ad$.

(23) Jan 2017 4 Marks

Obtain a minimal logical expression for the given maxterm expression using K-Map.

$$f(a,b,c,d) = \pi(0,1,4,5,6,7,9,14) + \pi_d(13,15)$$

Soln



$$f(a,b,c,d) = (c+d')(a+b')(b'+c')(a+c)$$

(24) Jan 2017 8M

Use QM method of minimization to obtain PIs and Minimal expression for the following minterm expression.

$$f(a,b,c,d) = \sum(0,1,4,5,7,8,13,15) + \sum d(2)$$

0 → 0000
Group 0

1 → 0001
4 → 0100
2 → 0010
8 → 1000

5 → 0101
Group 2

7 → 0111
13 → 1101
Group 3

15 → 1111
Group 4

Group	Minterm	a b c d
0	0 ✓	0 0 0 0
1	1 ✓	0 0 0 1
	2 ✓	0 0 1 0
	4 ✓	0 1 0 0
	8 ✓	1 0 0 0
2	5 ✓	0 1 0 1
	10 ✓	1 0 1 0
3	7 ✓	0 1 1 1
	13 ✓	1 1 0 1
4	15 ✓	1 1 1 1

Table 1: Minterm Group of 1.

Group	Minterm	a b c d
0	(0,1) ✓	0 0 0 -
	(0,2) ✓	0 0 - 0
	(0,4) ✓	0 - 0 0
	(0,8) ✓	- 0 0 0
1	(1,5) ✓	0 - 0 1
	(1,9) ✓	- 0 0 1
	(2,6) ✓	0 1 0 -
	(2,10) ✓	- 1 0 -
2	(5,7) ✓	0 1 - 1
	(5,13) ✓	- 1 0 1
3	(7,15) ✓	- 1 1 1
	(13,15) ✓	1 1 - 1

Table 2: Minterm Groups of 2

Table(3): Minterm groups of ~~b~~

(25) convert the boolean function to ..

(a) $Y = f(a,b,c) = (a+b)(a+c)$ minterm canonical form.

$$Y = f(a,b,c) = (a+b)(a+c)$$
$$= aa + ac + ba + bc$$
$$= a + ac + ba + bc.$$

$$Y = f(a,b,c) = a(b+\bar{b})(c+\bar{c}) + ac(b+\bar{b}) + ba(c+\bar{c}) + bc(a+\bar{a})$$
$$= abc + ab\bar{c} + a\bar{b}c + a\bar{b}\bar{c} + abc + \bar{a}bc + abc + ab\bar{c} + abc + \bar{a}bc$$

$$\boxed{Y = abc + ab\bar{c} + a\bar{b}c + a\bar{b}\bar{c} + \bar{a}bc.}$$

(b) $P = f(a,b,c,d) = (a+b)(b+c)(\bar{c}+a)$ maxterm canonical form

$$P = (a+b+c\bar{c})(b+c+a\bar{a}) + (a+b\bar{b}+\bar{c})$$
$$= (a+b+c)(a+b+\bar{c})(a+b+c)(\bar{a}+b+c)$$
$$(a+b+\bar{c})(a+\bar{b}+\bar{c})$$

$$\boxed{P = (a+b+c)(a+b+\bar{c})(\bar{a}+b+c)(a+\bar{b}+\bar{c})}$$

Table 3: Minterm groups of 4.

Group	Minterm	a b c d	
0	(0, 1, 4, 5)	0 - 0 -	$a'c'$
	(0, 4, 1, 5)	0 - 0 -	
2	(5, 7, 13, 15)	- 1 - 1	bd
	(5, 13, 7, 15)	- 1 - 1	

Table 3: Minterm groups of (4)

Prime Implicant Table

PIs		0	1	4	5	7	8	13	15
$a'c'$	(0, 1, 4, 5)	(X)	(X)	(X)	X				
bd	(5, 7, 13, 15)				X	(X)		(X)	(X)

$$f(a, b, c, d) = a'c' + bd + b'c'd'$$

(26) Find all the prime Implicants of the function

$$f(a, b, c, d) = \Pi\,(0, 2, 3, 4, 5, 12, 13) + \Pi d(8, 10)$$

Soln

0 → 0000 — Group 0

2 → 0010
4 → 0100
8* → 1000 — Group 1

3 → 0011
5 → 0101
12 → 1100
10* → 1010 — Group 2

13 → 1101 — Group 3.

Step 1: Arrange the maxterms in order of increasing no. of 1's

Groups	maxterm	a b c d
0	0	0 0 0 0 ✓
1	2	0 0 1 0 ✓
	4	0 1 0 0 ✓
	8*	1 0 0 0 ✓
2	3	0 0 1 1 ✓
	5	0 1 0 1 ✓
	10*	1 0 1 0 ✓
	12	1 1 0 0 ✓
3	13	1 1 0 1 ✓

Step 2: Grouping of maxterms of order 2

Group	Maxterm	a b c d
0	(0 2) ✓	0 0 – 0
	(0, 4) ✓	0 – 0 0
	(0, 8*) ✓	– 0 0 0
1	(2, 3) X	0 0 1 –
	(2, 10*) ✓	– 0 1 0
	(4, 5) ✓	0 1 0 –
	(4, 12) ✓	– 1 0 0
	(8*, 10*) ✓	1 0 – 0
	(8*, 12) ✓	1 – 0 0
2	(12, 13) ✓	1 1 0 –
	(5, 13)	– 1 0 1

Step 3 : Maxterm grouping of order 4.

Group	Maxterm	abcd	
0	(0, 2, 8*, 10*)	-0-0	→ b+d
	(0, 4, 8*, 12)	--00	→ c+d
	(0, 8*, 2, 10*)	-0-0	→ b+d
	(0, 8*, 4, 12)	--00	→ c+d
1	(4, 5, 12, 13)	-10-	→ $\bar{b}$+c

Step 4:

Prime Implicant Table.

	Group	Maxterm								
		0	2	3	4	5	12	13	8*	10*
b+d	0	x	(x)		:				x	(x)
c+d	0	x			(x)		x		x	
a+b+$\bar{c}$	1		x	(x)						
$\bar{b}$+c	1				x	(x)	x	(x)		

Prime Implicants are

(b+d), (c+d), (a+b+$\bar{c}$) & ($\bar{b}$+c)

Simplified Boolean expression

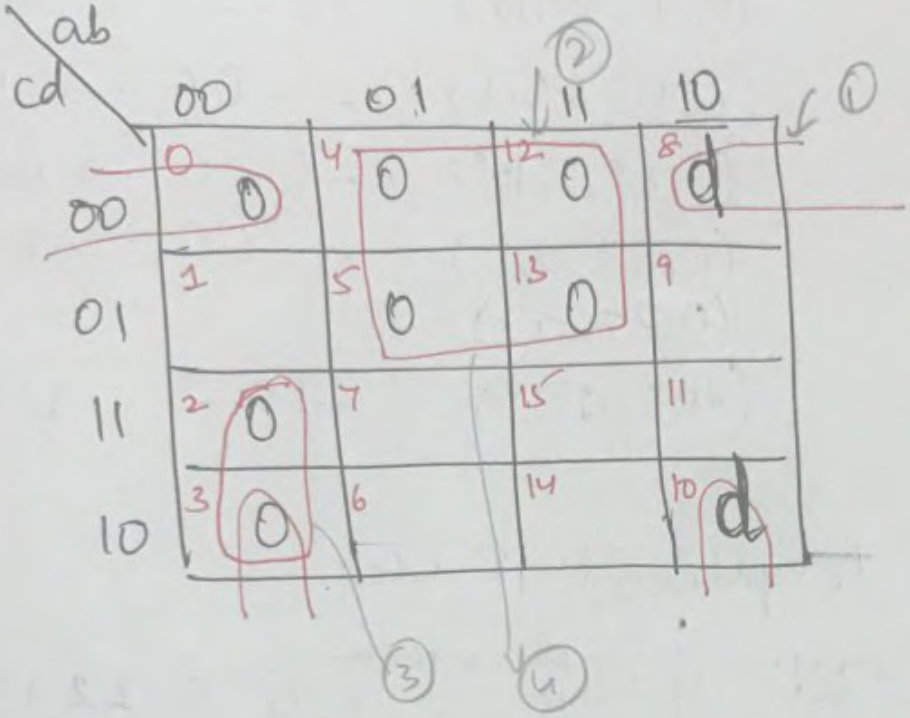
M = (b+d) (a+b+$\bar{c}$) ($\bar{b}$+c)

or

(c+d), (a+b+$\bar{c}$) ($\bar{b}$+c)

Verification using K-Map.

$f(a,b,c,d) = \Pi(0,2,3,4,5,12,13) + \Pi d(8,10)$



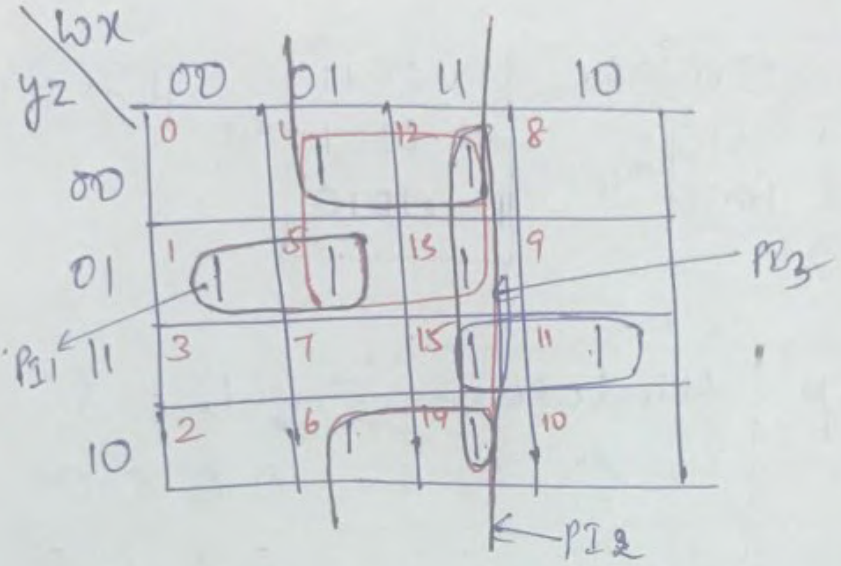
$M = (b+d)(\bar{b}+c)(a+b+\bar{c})$

(21) Using k-map determine the minimal Sum of product expression and realize the simplified expression using NAND gates

Jun 2017 (8M)

$$M = f(w, x, y, z) = \Sigma(1, 4, 5, 6, 11, 12, 13, 14, 15)$$

Soln



$$M = w'y'z + x'z' + wx + wyz$$

$$M = z(wy + w'y') + x(z' + w)$$

(28) Simplify the given boolean function using Quine - Mc cluskey Method. verify the result using K-map

June 2017 (12M)

$$Y = f(a,b,c,d) = \Sigma(0, 2, 3, 5, 8, 10, 11)$$

Sol^{n}

0 → 0000 → 40

2 → 0010
8 → 1000 → 41

3 → 0011
5 → 0101
10 → 1010 → 42

11 → 1011 ← 43.

Group	Minterm	a b c d
0	0	0 0 0 0
1	2	0 0 1 0
	8	1 0 0 0
2	3	0 0 1 1
	5	0 1 0 1
	10	1 0 1 0
3	11	1 0 1 1

Table 1: Minterm groups of 1.

- Arranging minterms in increasing order of no. of 1's.

Group 0 → 0 1's Group 2 → 2 1's

Group 1 → 1 1's Group 3 → 3 1's

Table 2: Minterm groups of two.

group	Minterm	abcd
0	(0,2) ✓	00-0 ✓
	(0,8) ✓	-000 ✓
1	(2,3) ✓	001- ✓
	(2,10) ✓	-010 ✓
	(8,10) ✓	10-0 ✓
2	(3,11) ✓	-011 ✓
	(10,11) ✓	101- ✓

Table 3: group of minterms of order 4.

Group	Minterm	abcd.	
0	(0,2,8,10)	-0-0	b'd'
	(0,8,2,10)	-0-0	
1	(2,3,10,11)	-01-	b'c
	(2,10,3,11)	-01-	

Table 4: creation of PI table

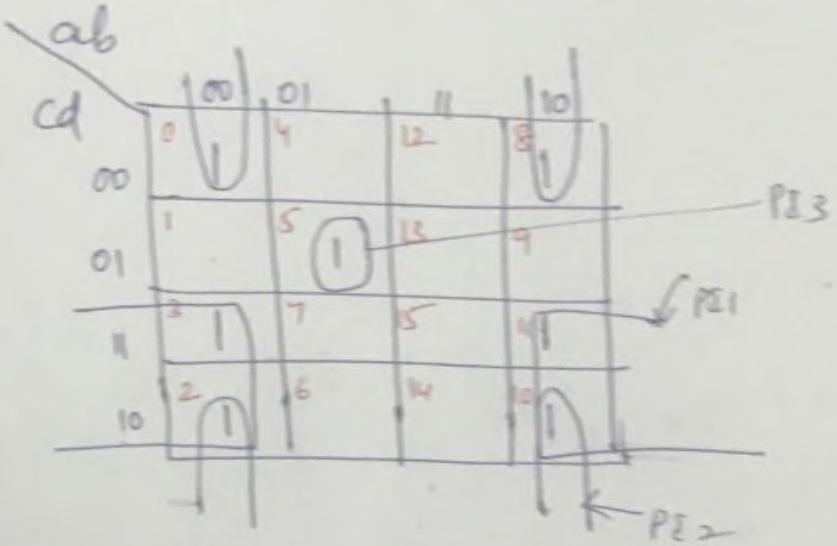
PI	Minterm	0	2	3	5	8	10	11
b'd'	(0,2,8,10)	×	×			(×)	×	
b'c	(2,3,10,11)		×	(×)			×	(×)

a'b'c'd

As 5 is not mapped to any PIs it is added to the final expression.

$$\therefore \boxed{Y = b'd' + b'c + a'bc'd}$$

Verification using K-Map.



$$\boxed{Y = b'c + b'd' + a'bc'd}$$

(29) Find the minimal sum for the incomplete Boolean function using QM method & Prime Implicant table reduction.

$$f(a,b,c,d) = \sum (3,4,5,7,10,12,14,15) + \sum d(2).$$

P(8)

Soln

3 → 0011	4 → 1000	7 → 0111	15 → 1111
5 → 0101	2* → 0010	14 → 1110	
12 → 1100			
10 → 1010			
Group 2	Group 1	Group 3	Group 4

Step 1: Arrange the minterms in increasing order of no. of 1's.

Group	Minterm	a b c d
1	2*	0 0 1 0
	4	0 1 0 0
2	3	0 0 1 1
	5	0 1 0 1
	12	1 1 0 0
	10	1 0 1 0
3	7	0 1 1 1
	14	1 1 1 0
4	15	1 1 1 1

Step 2: Minterms of order 2.

Group	Minterm	abcd	
1	(2*, 3)	001 - x	Q
	(4, 5)	010 _ x	R
	(4, 12)	- 100 x	S
	(2*, 10)	- 010 x	T
2	(3, 7)	0 - 11 x	U
	(5, 7)	01 - 1 x	V
	(12, 14)	11 - 0 x	W
	(10, 14')	1 - 10 x	X
3	(7, 15)	- 11 1 x	Y
	(14, 15)	111 - x	Z

Step 3 : Grouping of minterms of order 4.

Group	Minterm	abcd
1		

NOT POSSIBLE

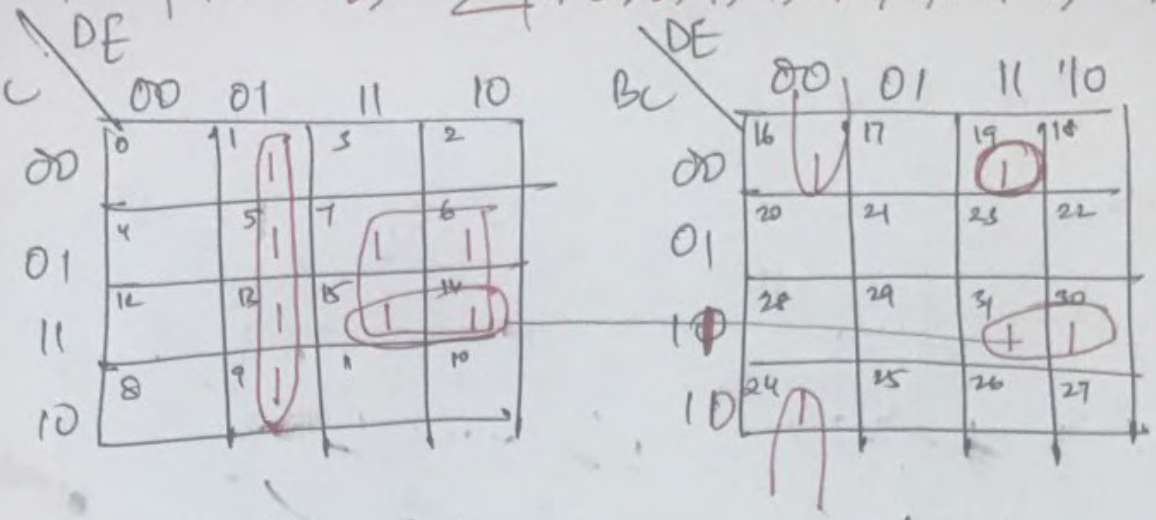
Steps : Prime Implicant Table.

→ Don't care

Rows	PI	2^4	3	4	5	7	10	12	14	15
Q	$\bar{a}\bar{b}c$	X	X							
R	$\bar{a}b\bar{c}$			X	X					
S	$b\bar{c}\bar{d}$			X				X		
T	$\bar{b}c\bar{d}$	X					X			
U	$\bar{a}\,\blacksquare\,d$		X			X				
V	$\bar{a}bd$				X	X				
W	$ab\bar{c}$							X	X	
X	$a\,\blacksquare\,\bar{d}$						X		X	
Y	bcd					X				X
Z	abc								X	X

$Y = f(ABCDE) = \sum(1, 5, 6, 7, 9, 13, 14, 15, 16, 19, 24, 30, 31)$

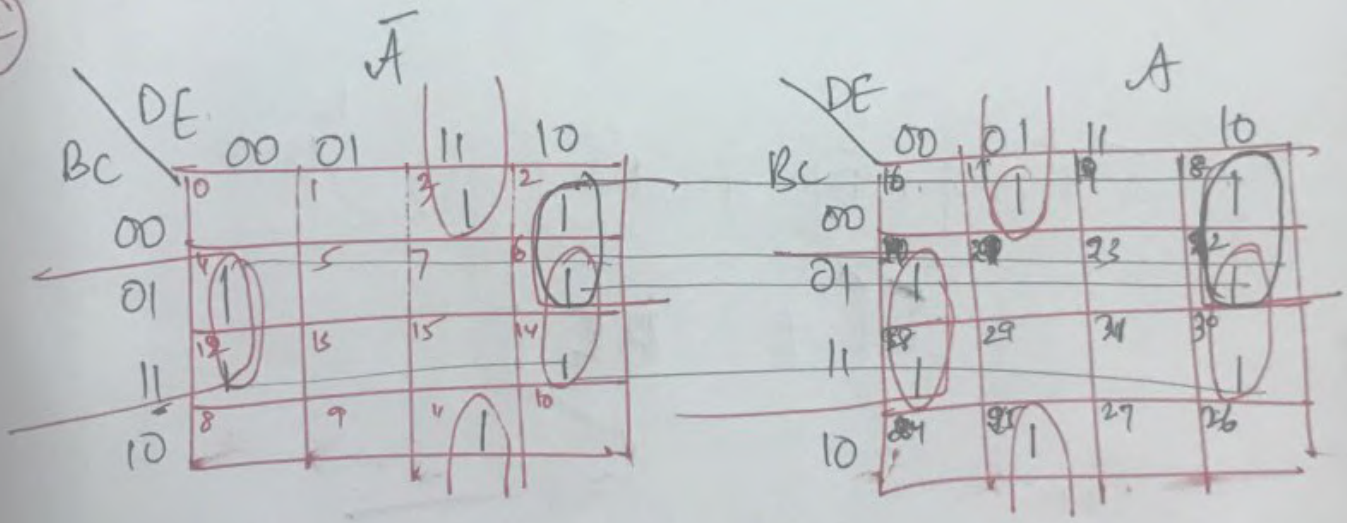
(1)



$A = 0 (\bar{A})$ $A = 1$

$Y = \bar{A}\bar{D}E + A\bar{B}\bar{C}DE + \bar{A}\bar{C}\bar{D}\bar{E} + BCD + \bar{A}CD$

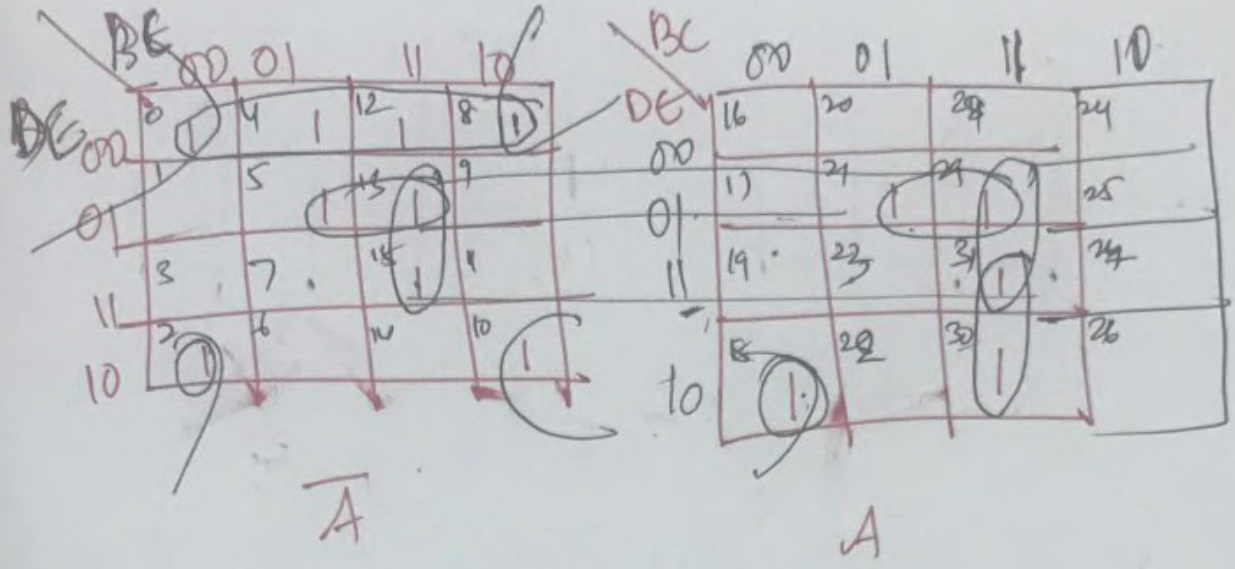
(2)



$f(ABCDE) = \sum m(2, 3, 4, 6, 11, 12, 14, 17, 18, 20, 22, 25, 28, 30)$

$Y = \bar{C}\bar{E} + \bar{A}\bar{C}DE + A\bar{C}DE + \bar{B}D\bar{E} +$

(3) $f(ABCDE)$

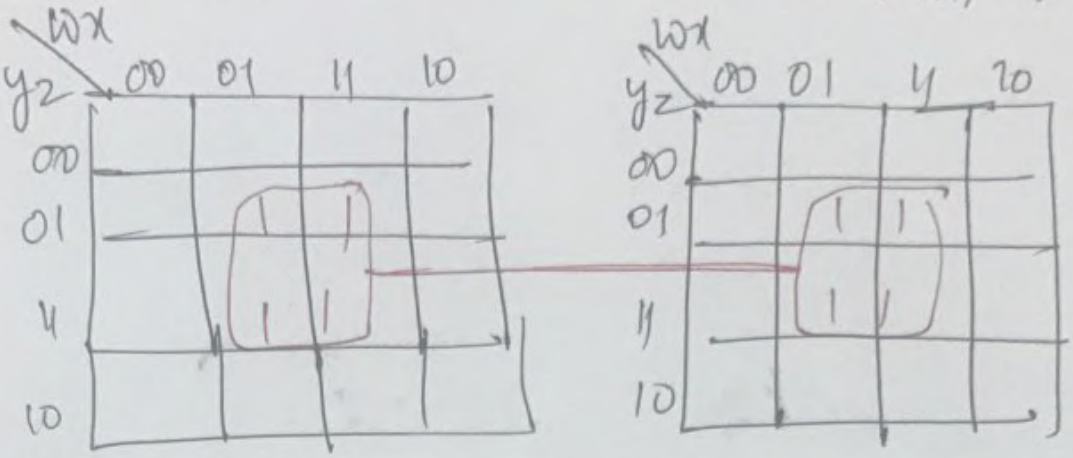


~~$Y = \bar{A}\bar{C}\bar{E} + \bar{A}B\bar{C} + B\bar{C}$~~

$Y = \bar{A}\bar{E}\bar{C} + A\bar{D}\bar{E} + \bar{D}\bar{E}\bar{B}\bar{C} + \overset{BC}{AD} +$

$+ CDE +$ ~~BEE~~

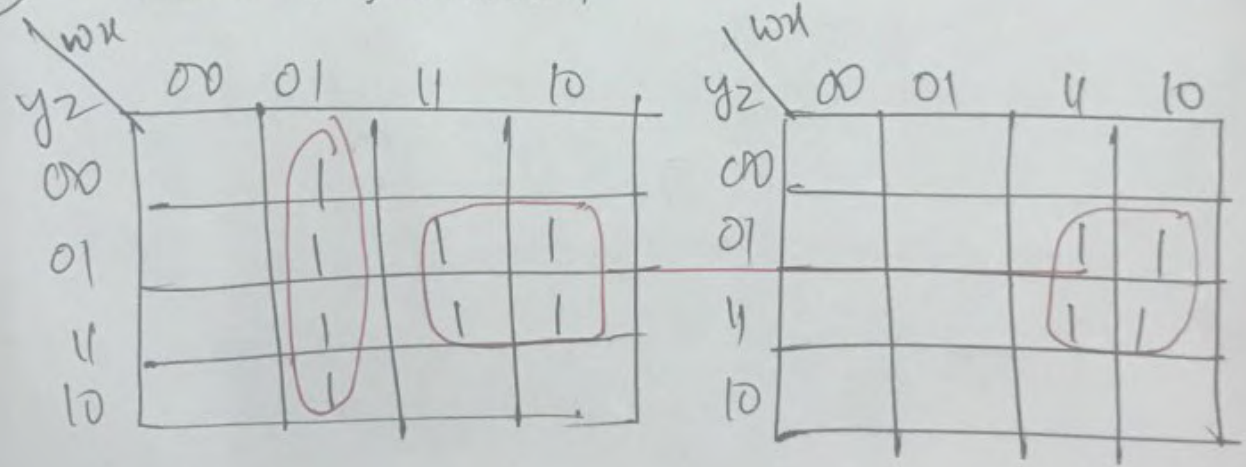
(4) $R = f(vwxyz) = \sum 5, 7, 13, 15, 21, 23, 29, 31$



$\bar{V}$ V

$R = zx$

(19) Second Method



$\bar{V}$ V

$J = wz + \bar{v}\bar{w}x$

(1) Obtain all the prime implicants of the given Boolean functions using Quine Mcclusky method.

(25)

(i) $f(abc) = \Sigma m(0,2,3,4)$

a \ bc	00	01	11	10
0	1 0	1	1 3	1 2
1	1 4	5	7	6

$Y = \bar{b}\bar{c} + \bar{a}b$

PI (0,4) $\rightarrow \bar{b}\bar{c}$ $\rightarrow$ EPI

PI (2,3) $\rightarrow \bar{a}b$ $\rightarrow$ EPI

PI (0,2) $\rightarrow \bar{a}\bar{c}$ $\rightarrow$ PI

	abc	Y
0	000	1
1	001	0
2	010	1
3	011	1
4	100	1
5	101	0
6	110	0
7	111	0

Group of 0's $\rightarrow$ 000 $\rightarrow$ 0 ✓

Group of 1-1's $\rightarrow$ 001 $\rightarrow$ 1

010 $\rightarrow$ 2 ✓

100 $\rightarrow$ 4 ✓

Group of 2-1's 011 $\rightarrow$ 3 ✓

101 $\rightarrow$ 5

110 $\rightarrow$ 6

Group of 3-1's 111 $\rightarrow$ 7

(1) Grouping minterms according to no. of 1's

Group	Minterm	abc
0	0	000
1	2	010
	4	100
2	3	011

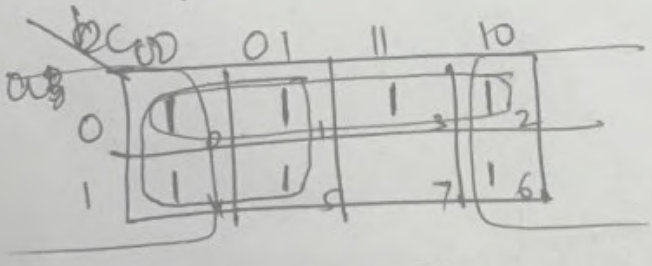
Table 2 : creation of minterms groups of 2

group	Minterm	abc
0	0, 2	0 – 0
	0, 4	– 0 0
1	2, 3	0 1 –

PI terms ~~Minterm~~	Decimal	0	2	3	4
$\bar{a}\bar{c}$	0, 2	×	×		
$\bar{b}\bar{c}$	0, 4 ←	×			(×)
$\bar{a}b$	2, 3 ←		×	(×)	

$$Y = \bar{a}b + \bar{b}\bar{c}$$

(2) $Y = f(abc) = \Sigma m(0,1,2,3,4,5,6)$



PI (0,2,4,6) → $\bar{c}$
PI (0,1,4,5) → $\bar{b}$
PI (0,1,2,3) → $\bar{a}$

$$Y = \bar{a} + \bar{b} + \bar{c}$$

group of 0 – 1's → 000 → 0 –
group of 1 – 1's → 001 → 1 –
010 → 2 –
100 → 4 –
group of 2 – 1's → 011 → 3 –
110 → 6 –
101 → 5 –

Creation of minterms of 2.

Group	Minterm	abc
0	0	000
1	1	001
	2	010
	4	100
2	3	011
	5	101
	6	110

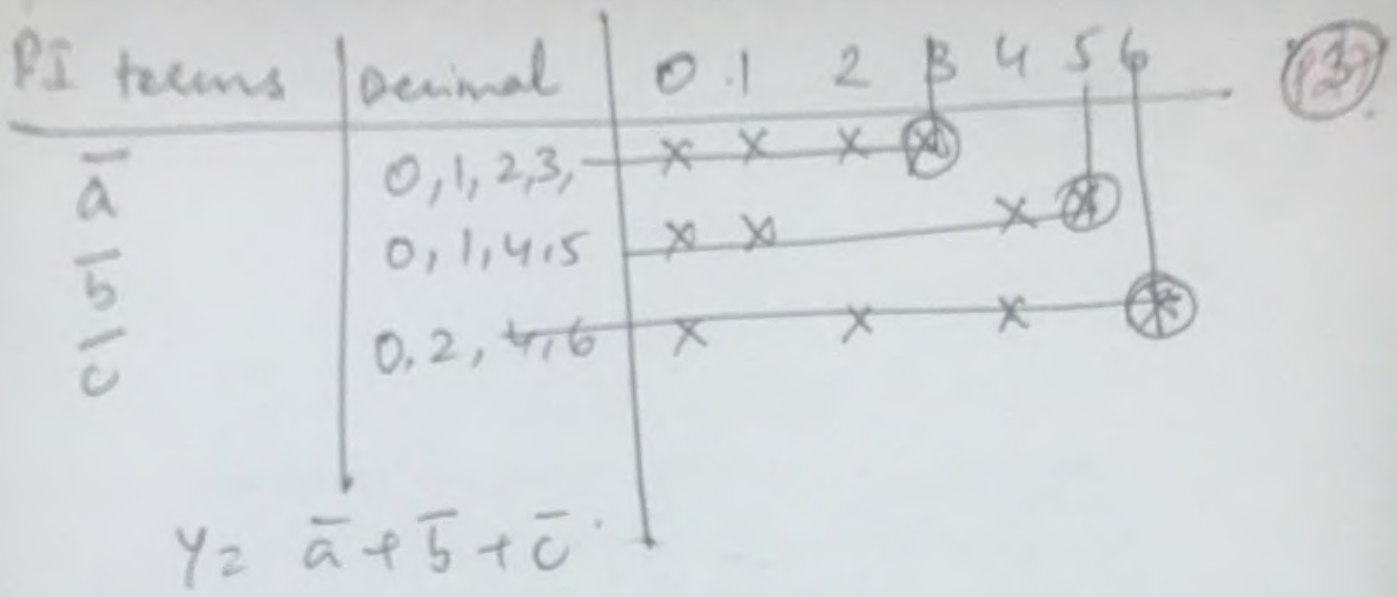
Grouping minterms according to no. of 1's.

creation of minterms of 2

Group	Minterm	abc
0	0,1	00- ✓
	0,2	0-0
	0,4	-00
1	1,3	0-1
	1,5	-01
	2,3	01- ✓
	2,6	-10
	4,5	10- ✓
	4,6	1-0

Creation of minterms of 4

Group	Minterm	abc	
0	0,1,2,3	0--	$\bar{a}$
	0,1,4,5,	-0-	$\bar{b}$
	0,2,4,6	--0	$\bar{c}$


PI terms
Decimal
0 1 2 3 4 5 6
$\bar{a}$
$\bar{b}$
$\bar{c}$
0, 1, 2, 3,
0, 1, 4, 5
0, 2, 4, 6
$Y_2 = \bar{a} + \bar{b} + \bar{c}$

Solving POS Using QM technique.

(1) $Y = f(abcd) = \Pi M(0,2,3,4,5,12,13) + \Pi d(8,10)$

Soln $Y = f(abcd) = \Pi M(0,2,3,4,5,8^*,10^*,12,13)$
$M_0\ M_2\ M_3\ M_4\ M_5\ M_8\ M_{10}\ M_{12}\ M_{13}$

$$Y = M_0 \cdot M_2\ M_3\ M_5\ M_8\ M_{10}\ M_{12}\ M_{13}$$

$$\overline{Y} = \overline{M_0\ M_2\ M_3\ M_5\ M_8\ M_{10}\ M_{12}\ M_{13}}$$

$$\overline{Y} = m_0 + m_2 + m_3 + m_5 + m_8 + m_{10} + m_{12} + m_{13}$$

Let us apply QM technique to find the Prime Implicants of $\overline{Y}$.

Step 1: Arranging the minterms in order of no. of 1's.

Group 0 → 0000 → 0
Group 1 → 0010 → 2
→ 1000 → 8*
Group 2 → 0011 → 3
0101 → 5
1010 → 10*
1100 → 12
Group 3 → 1101 → 13

Step 2: Group of minterms in Ascending order

Group	minterm	abcd
0	0	0000 ✓
1	2	0010 ✓
	8*	1000 ✓
2	3	0011 ✓
	5	0101 ✓
	10*	1010 ✓
	12	1100 ✓
3	13	1101 ✓

Step 3: Group of minterms of 2

Group	Minterm	abcd
0	(0,2)	00-0 ✓
	(0,8*)	-000 ✓
1	(2,3)	001- X
	(2,10*)	-010
	(8*,10*)	10-0 ✓
	(8*,12)	1-00 X
2	(5,13)	-101 X
	(12,13)	110- X

Step 4: Group of minterms of 4

Group	Minterm	abcd
0	(0,2),8*,10*)	-0-0
	(0,8*) (2,10*)	-0-0
1		

		abcd		Minterm	Maxterm
PI	(0 2 8* 10*)	-0-0	→	b'd'	b+d
	(2,3)	001-	→	a'b'c	a+b+c'
	(8*,12)	1-00	→	ac'd'	$\bar{a}$+c+d
	(5,13)	-101	→	bc'd*	$\bar{b}$+c+$\bar{d}$
	(12,13)	110-	→	abc'	$\bar{a}$+$\bar{b}$+c

$Y = (b+d)(a+b+c')(\bar{a}+c+d)(\bar{b}+c+\bar{d})(\bar{a}+\bar{b}+c$

(2) $Y = f(abcd) = \pi(0, 2, 3, 4, 5, 12, 13) + dc(8, 10)$

$$\boxed{Y = (a+b+\bar{c})(b+d)(c+d)(\bar{b}+c)}$$

MODULE 2: ANALYSIS AND DESIGN OF COMBINATIONAL LOGIC

- General approach to combinational logic design.
- Decoders
  (a) BCD Decoders
- Encoders
- Digital Multiplexers
  (a) Using Multiplexers as Boolean function generators.
- Adders and Subtractors
  (a) Cascading full adders
  (b) Look ahead carry
- Binary comparators.

III ECE

General approach to combinational Logic Design.

- The Synthesis of combinational logic starts with a problem statement and progresses through a series of steps, terminating in the final circuit design.

- The combinational logic design steps are as follows.

Step 1: Develop a statement describing the problem to be solved.

Step 2: Based on the problem statement, Construct a truth table that clearly establishes the relation b/w the input and output variables.

Step 3: Use K-Maps or QM Techniques to simplify the functions in deriving the output equations. This may require that the output equations be expressed as either SOP or POS. The best solution will require the fewest gates and gate inputs.

Step 4: Arrange the simplified equations to suit the logic primitive type to be used in realizing the circuit. (Using NANDS, NORS or AND-OR logic required)

Step 5: Draw the final logic diagram.

Step 6: Document the design by identifying variables names that indicate assertion levels, if possible provide Truth table.

Example: Design a combinational circuit that will multiply 2, 2-bit binary values.

Sol^n ① Let the inputs be $A(A_1, A_0)$ & $B(B_1, B_0)$. and the output be $P(P_3, P_2, P_1, P_0)$.

- The four output variables are necessary because the maximum product of two, 2-bit values requires 4 bits. Let each variable be active high.

② Construct the Truth table (Next page)

③ The individual simplified equations are:

$$P_3 = A_1 A_0 B_1 B_0.$$

$$P_2 = A_1 A_0' B_0 + A_1 B_1 B_0'$$

$$P_1 = A_1' A_0 B_1 + A_0 B_1 B_0' + A_1 B_1' B_0 + A_1 A_0' B_0$$

$$P_0 = A_0 B_0.$$

$A_1\ A_0$
$B_1\ B_0$

A_1B_0 A_0B_0
B_1A_1' B_1A_0 X

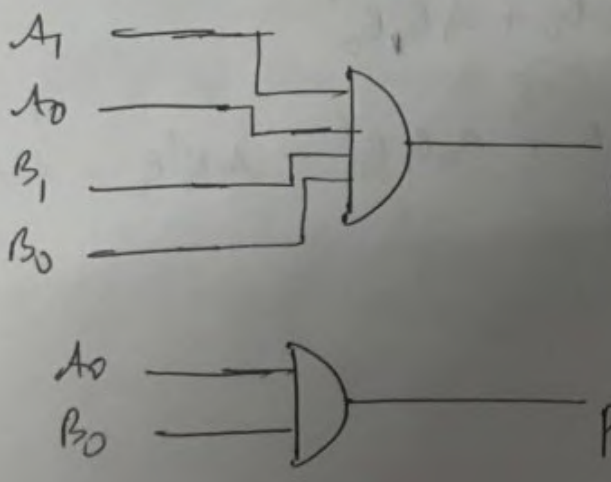
B_1A_1 $A_1B_0+B_1A_0$ A_0B_0

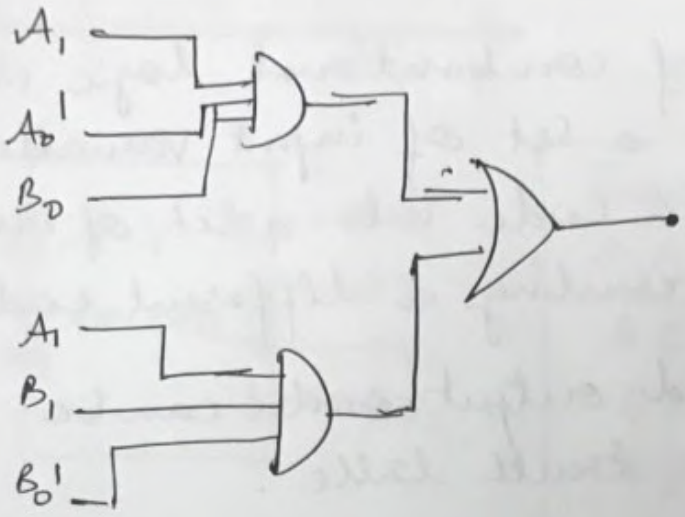
Truth table

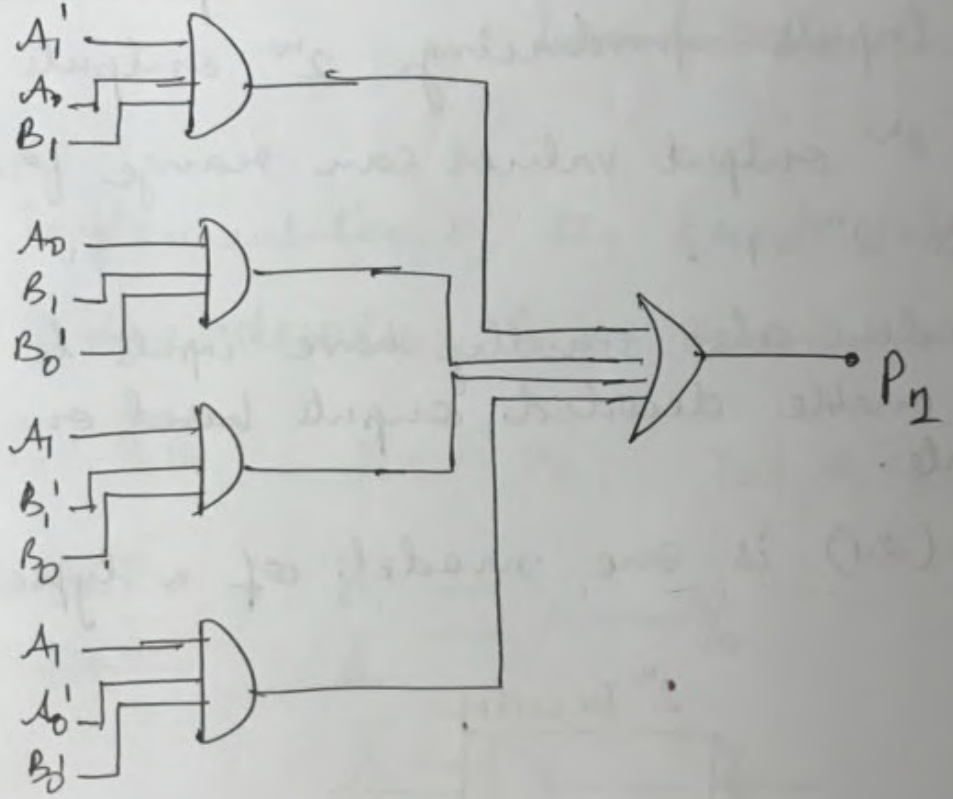
A_1	A_0	B_1	B_0	P_3	P_2	P_1	P_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1

11
11
111
11x
1001
11
00
10
1100

Logic Diagram for 2x2-bit Multiplier




A_1
A_0'
B_0
A_1
B_1
B_0'
P_2


A_1'
A_0
B_1
A_0
B_1
B_0'
A_1
B_1'
B_0
A_1
A_0'
B_0
P_1

DECODERS

- are a class of combinational logic circuits that convert a set of input variables representing a code into a set of output variables representing a different code.
- The input and output codes can be expressed in a truth table.
- Encoded information is represented as n inputs producing 2^n outputs.
- The 2^n output values can range from 0 to $2^n - 1$.
- Decoders also usually have inputs to activate or enable decoded outputs based on data inputs.
- Fig (2.1) is one model of a typic decoder.

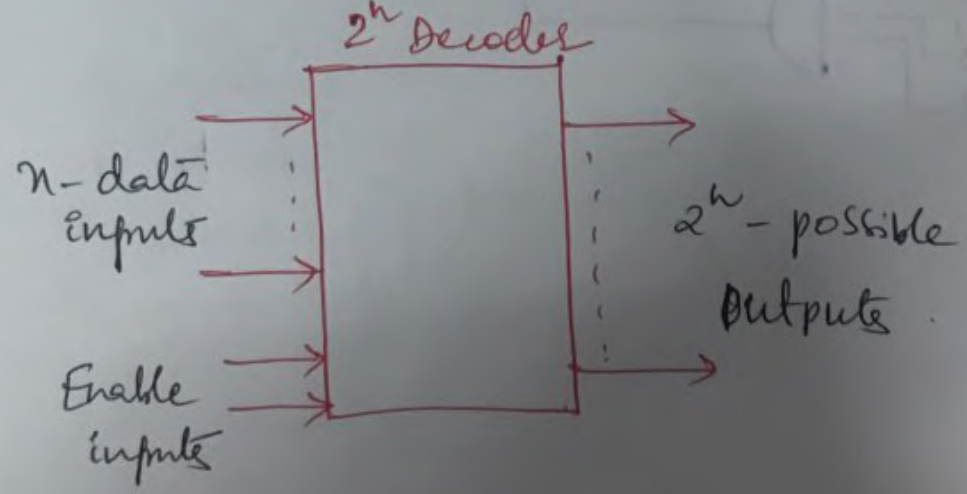
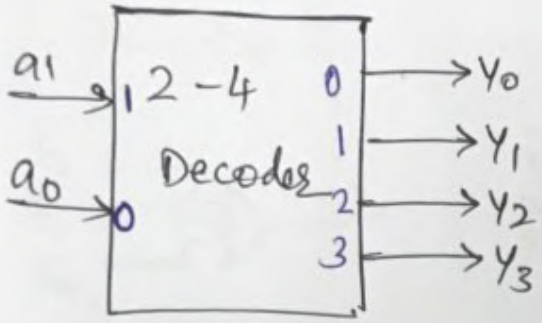


fig 2.1 : Typical Decoder.

2-4 line DECODER



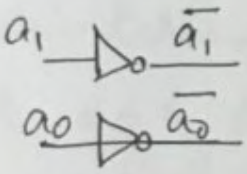
Symbol

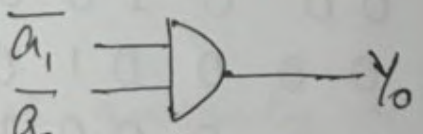
Truth Table

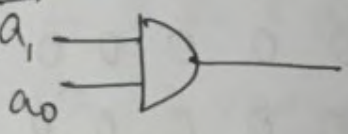
Inputs $a_1 a_0$	outputs $Y_0 Y_1 Y_2 Y_3$
0 0	1 0 0 0
0 1	0 1 0 0
1 0	0 0 1 0
1 1	0 0 0 1

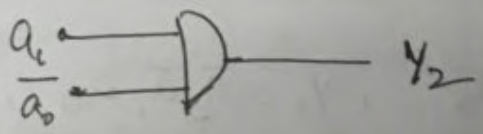
- The implementation of the truth table of the 2 to 4 line decoder is as shown.

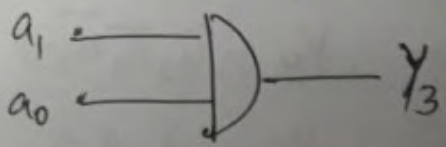
$Y_0 = \bar{a_1}\bar{a_0}$, $Y_1 = \bar{a_1} a_0$, $Y_2 = a_1 \bar{a_0}$, $Y_3 = a_1 a_0$.





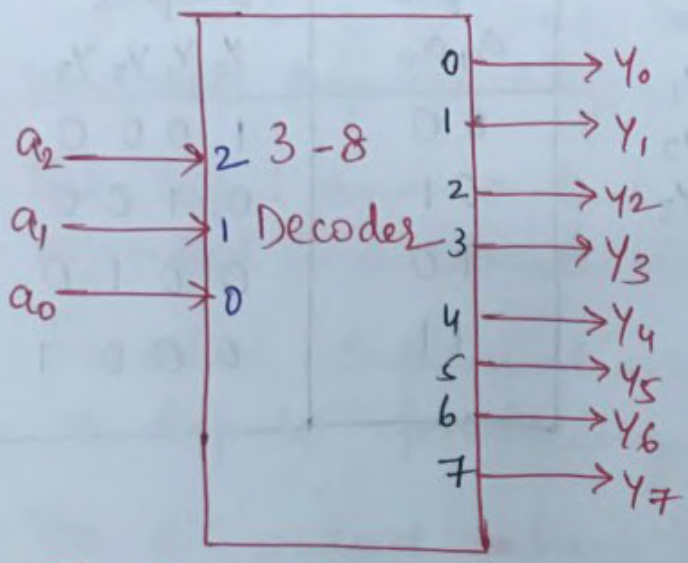






3-8 Line DECODER.

Symbol.



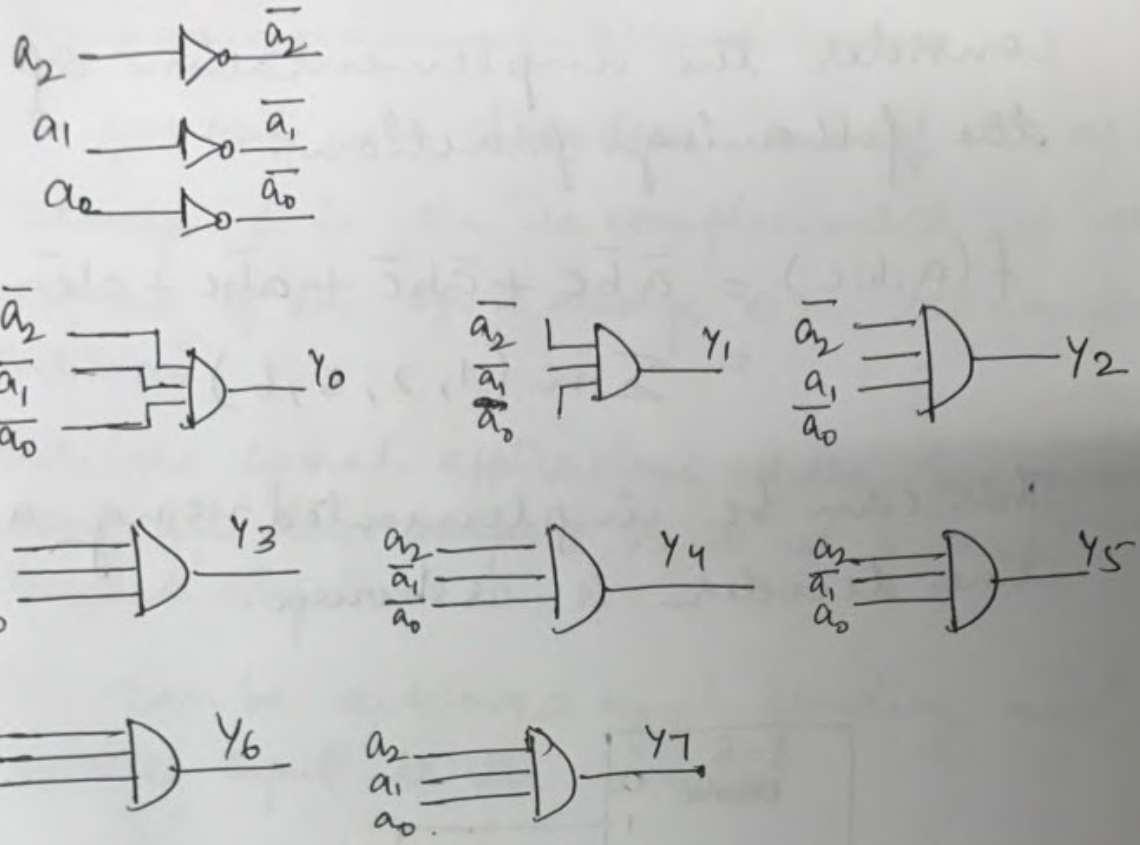
Truth Table.

a_2	a_1	a_0	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

$Y_0 = \overline{a_2}\,\overline{a_1}\,\overline{a_0}$, $Y_1 = \overline{a_2}\,\overline{a_1}\,a_0$, $Y_2 = \overline{a_2}\,a_1\,\overline{a_0}$

$Y_3 = \overline{a_2}\,a_1\,a_0$, $Y_4 = a_2\,\overline{a_1}\,\overline{a_0}$, $Y_5 = a_2\,\overline{a_1}\,a_0$

$Y_6 = a_2\,a_1\,\overline{a_0}$, $Y_7 = a_2\,a_1\,a_0$.



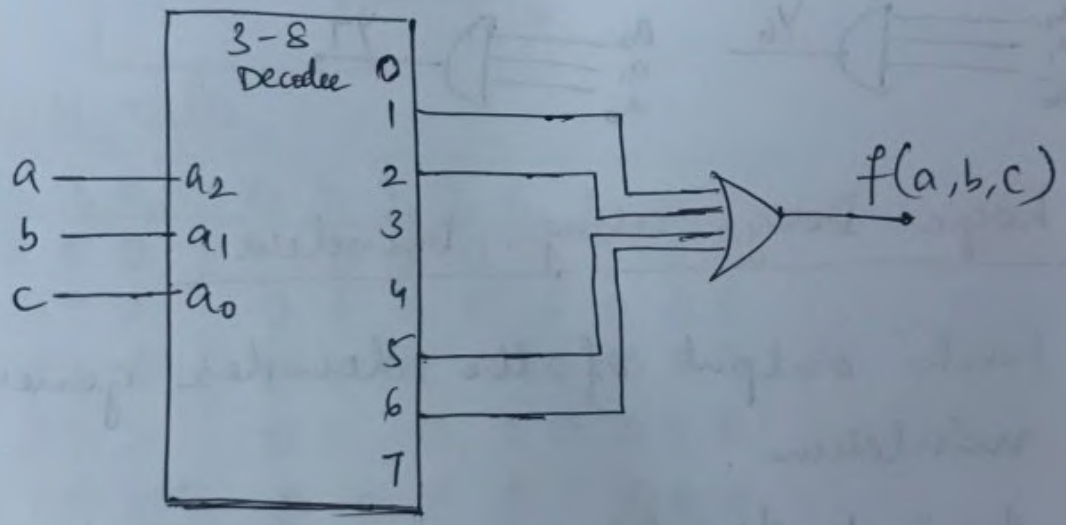
Logic Design using Decoders.

- Each output of the decoder generates a minterm.
- A 2-4 decoder generates 4 minterm using 2 variables and a 3 to 8 decoder generates 8 minterms with 3 variables.
- Thus a n to 2^n line decoder is very convenient to generate n variable minterms and realize a SOP expression.

consider the implementation of the following function.

$$f(a,b,c) = \bar{a}\bar{b}c + \bar{a}b\bar{c} + a\bar{b}c + ab\bar{c}$$
$$= \sum m(1,2,5,6)$$

This can be implemented using a 3 to 8 line decoder is as shown.

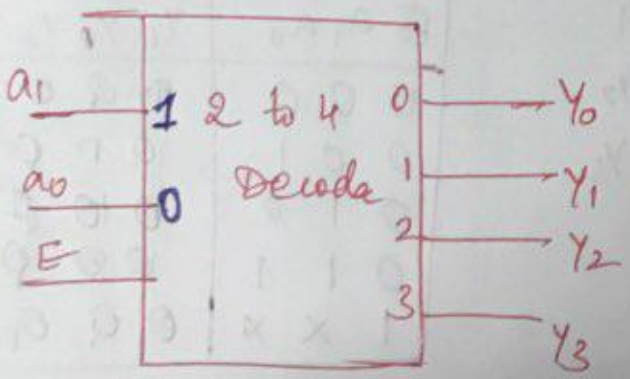


DECODERS WITH ENABLE INPUT:

- In general, in decoders one of the op is always 1 in the uncomplemented op version or one of the ops is always 0 in the complemented version.
- There are several applications where we would want all the outputs to be at 0 or 1 respectively.
- This can be achieved by including an enable input to the decoder.
- The truth table, symbol and schematic of 2 to 4 line decoder with an enable input and uncomplemented op is as shown,

2 to 4 line Decoder with Enable input. (ACTIVE HIGH)

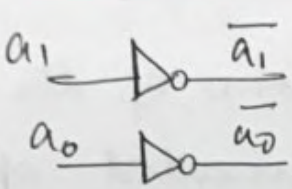
Symbol

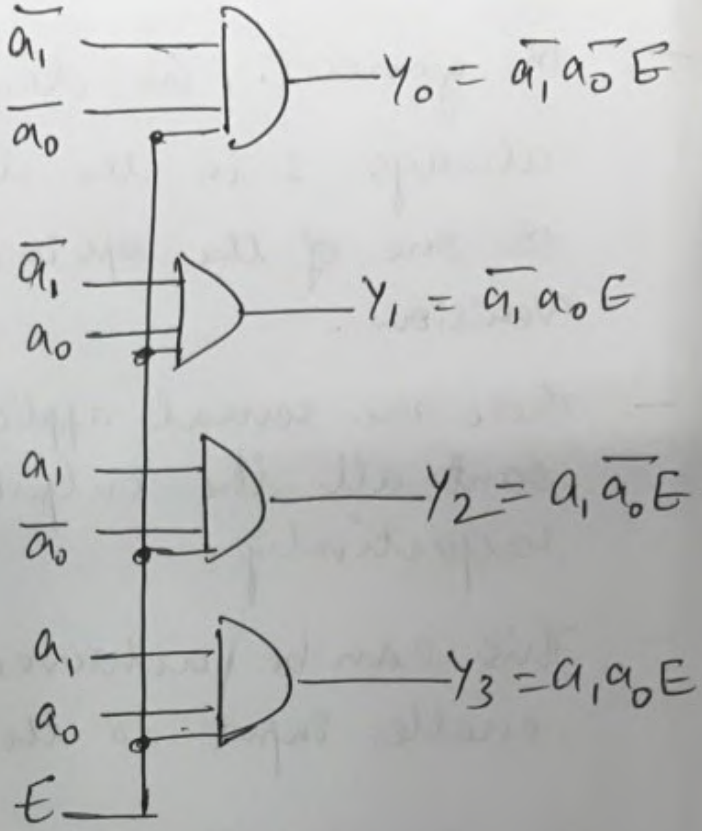


Truth Table

Inputs			Outputs			
E	a_1	a_0	Y_0	Y_1	Y_2	Y_3
0	X	X	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

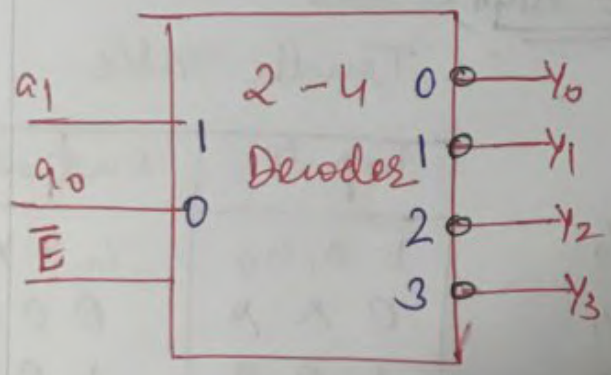
Schematic





2-to-4 line Decoder with Active low Enable input.

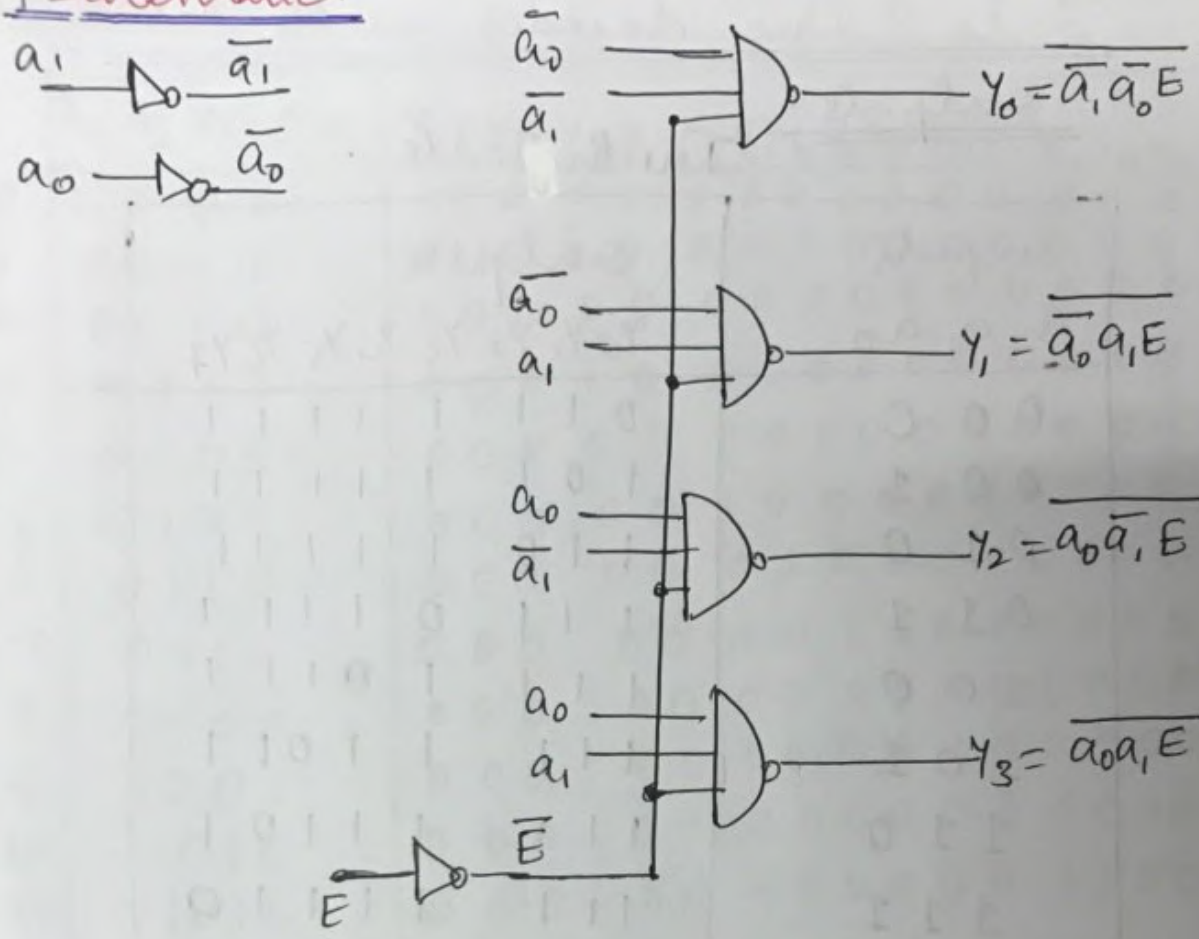
Symbol



Truth Table.

Inputs			outputs			
E	a_1	a_0	Y_0	Y_1	Y_2	Y_3
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0
1	X	X	1	1	1	1

Schematic



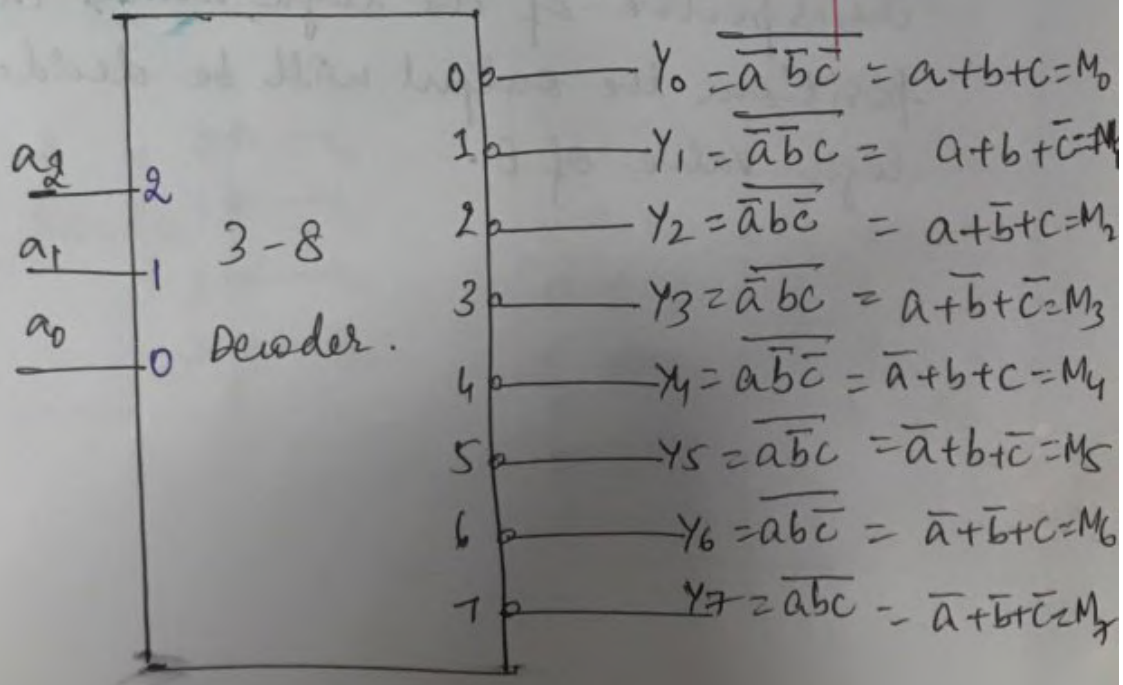
** The X in the truth table indicate a don't care condition. This means that irrespective of the logic values in the 'X' positions the output will be decided by the logic value of E.

3 to 8 Line decoder with Active low outputs.

Truth Table.

Inputs			outputs.							
a_2	a_1	a_0	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
0	0	0	0	1	1	1	1	1	1	1
0	0	1	1	0	1	1	1	1	1	1
0	1	0	1	1	0	1	1	1	1	1
0	1	1	1	1	1	0	1	1	1	1
1	0	0	1	1	1	1	0	1	1	1
1	0	1	1	1	1	1	1	0	1	1
1	1	0	1	1	1	1	1	1	0	1
1	1	1	1	1	1	1	1	1	1	0

Symbol.



2.7a

Design 4:16 decoder using 2 to 4 decoder.

	$a_3\ a_2\ a_1\ a_0$	$Y_0\ Y_1\ Y_2\ Y_3\ Y_4\ Y_5\ Y_6\ Y_7\ Y_8\ Y_9\ Y_{10}\ Y_{11}\ Y_{12}\ Y_{13}\ Y_{14}\ Y_{15}$
0	0 0 0 0	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1	0 0 0 1	0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2	0 0 1 0	0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
3	0 0 1 1	0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
4	0 1 0 0	0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
5	0 1 0 1	0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
6	0 1 1 0	0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
7	0 1 1 1	0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
8	1 0 0 0	0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
9	1 0 0 1	0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
10	1 0 1 0	0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
11	1 0 1 1	0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
12	1 1 0 0	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
13	1 1 0 1	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
14	1 1 1 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
15	1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

$Y_0 = \bar{a_3}\bar{a_2}\bar{a_1}\bar{a_0}$ $Y_1 = \bar{a_3}\bar{a_2}\bar{a_1}a_0$ $Y_2 = \bar{a_3}\bar{a_2}a_1\bar{a_0}$

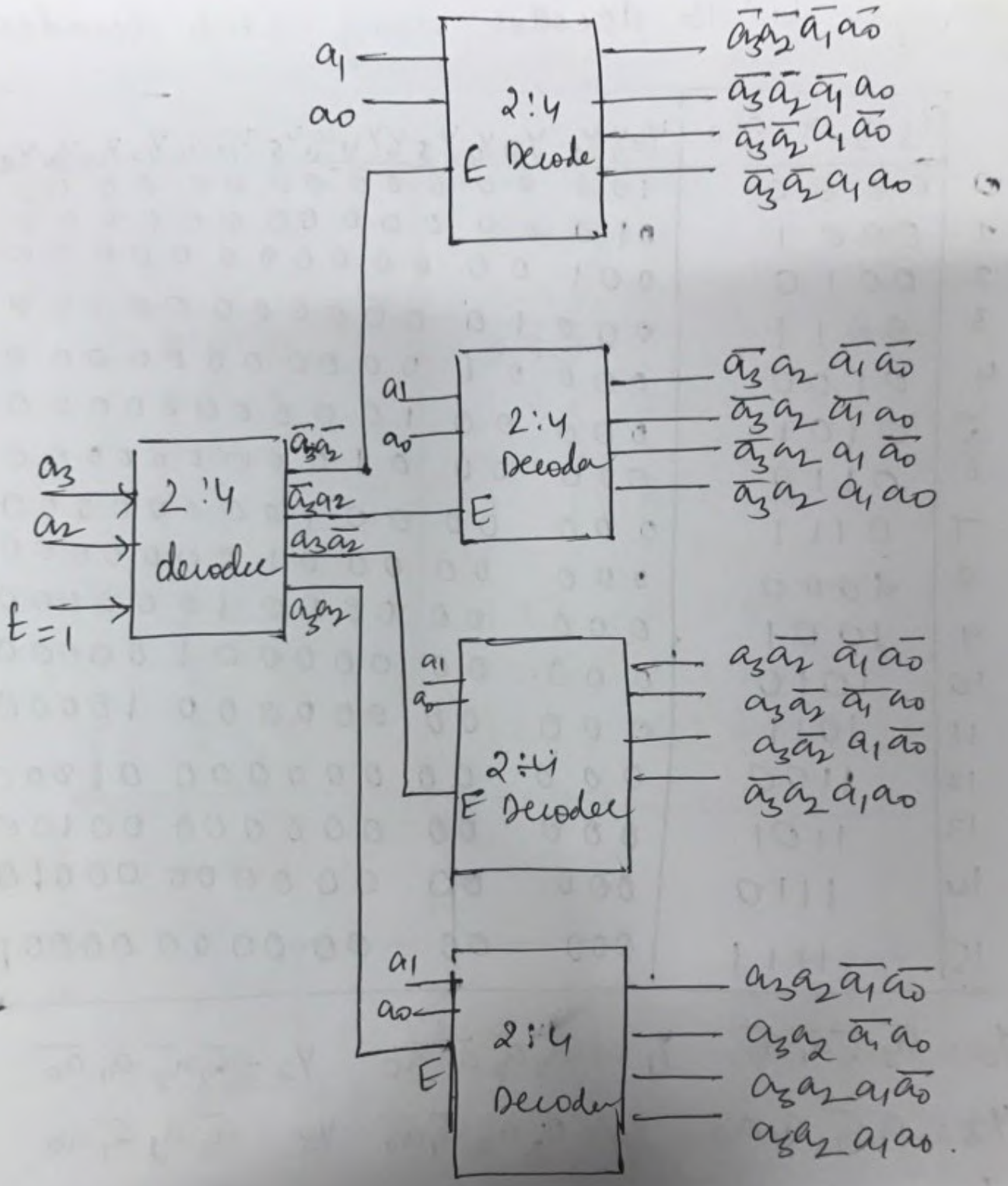
$Y_3 = \bar{a_3}\bar{a_2}a_1a_0$ $Y_4 = \bar{a_3}a_2\bar{a_1}\bar{a_0}$ $Y_5 = \bar{a_3}a_2\bar{a_1}a_0$

$Y_6 = \bar{a_3}a_2a_1\bar{a_0}$ $Y_7 = \bar{a_3}a_2a_1a_0$ $Y_8 = a_3\bar{a_2}\bar{a_1}\bar{a_0}$

$Y_9 = a_3\bar{a_2}\bar{a_1}a_0$ $Y_{10} = a_3\bar{a_2}a_1\bar{a_0}$ $Y_{11} = a_3\bar{a_2}a_1a_0$

$Y_{12} = a_3a_2\bar{a_1}\bar{a_0}$ $Y_{13} = a_3a_2\bar{a_1}a_0$ $Y_{14} = a_3a_2a_1\bar{a_0}$

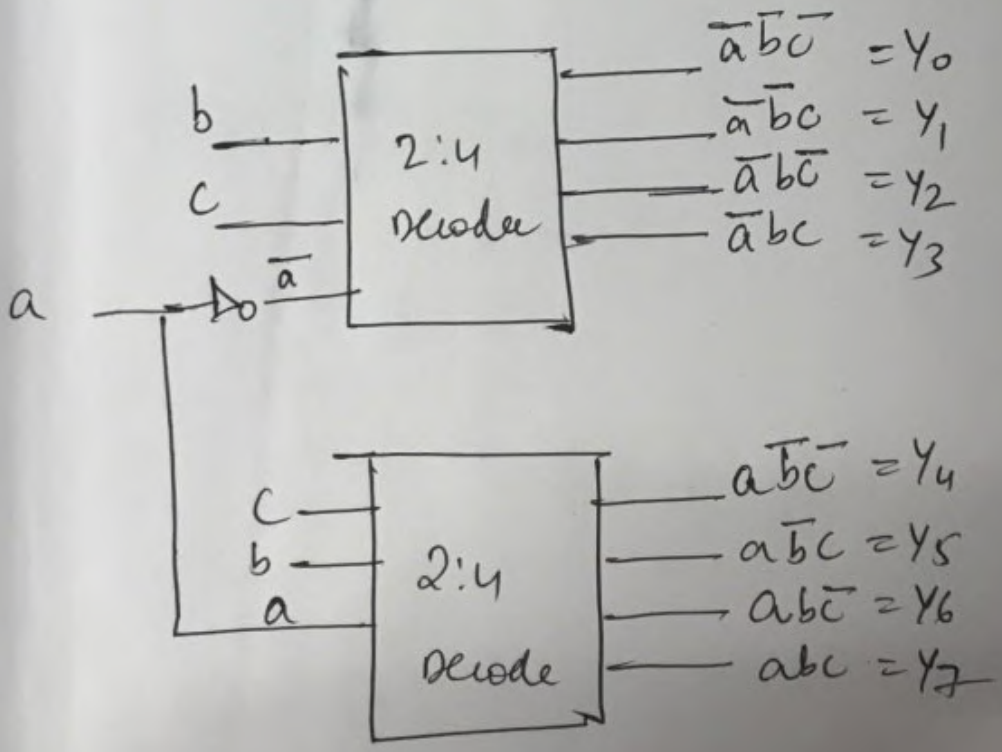
$Y_{15} = a_3a_2a_1a_0$.



2.76

Construct 3:8 Decoder using 2 to 4 line Decoder.

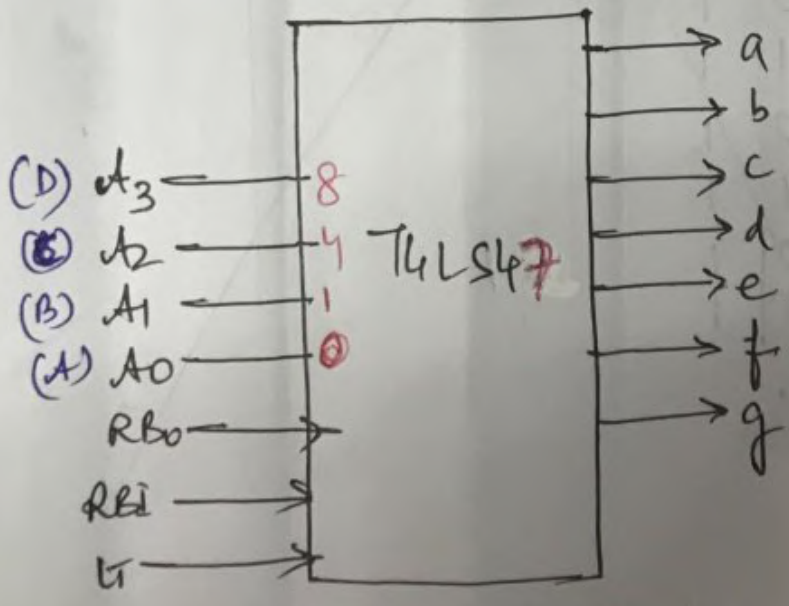
a b c	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
0 0 0	1	0	0	0	0	0	0	0
0 0 1	0	1	0	0	0	0	0	0
0 1 0	0	0	1	0	0	0	0	0
0 1 1	0	0	0	1	0	0	0	0
1 0 0	0	0	0	0	1	0	0	0
1 0 1	0	0	0	0	0	1	0	0
1 1 0	0	0	0	0	0	0	1	0
1 1 1	0	0	0	0	0	0	0	1

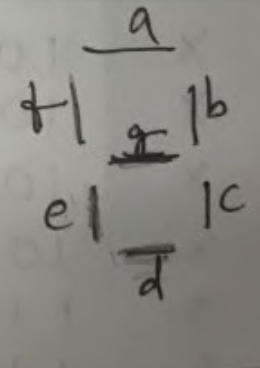


BCD TO SEVEN SEGMENT DECODER

- 74LS47 is a BCD to 7-segment decoder
- It accepts a binary coded decimal as input and converts it into a pattern to drive a seven-segment for displaying digits 0 to 9.
- 74LS47 accepts 4 lines of BCD(8421) input data and 7 op lines.

Pin Diagram:





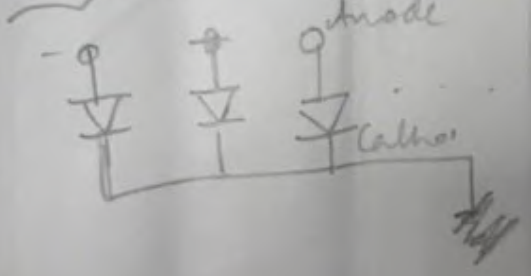
where LT → Lamp/Display Test (Active Low)

RB_O – Ripple Blank output
RB_I – Ripple Blank input.

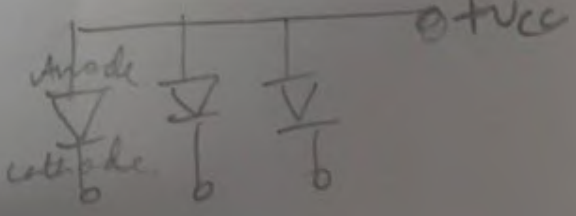
Truth Table (common cathode)

	$\overline{LT}$	$\overline{RBI}$	$\overline{RBO}$	DCBA	a b c d e f g
0	1	1	1	0000	1 1 1 1 1 1 0
1	1	X	1	0001	0 1 1 0 0 0 0
2	1	X	1	0010	1 1 0 1 1 0 1
3	1	X	1	0011	1 1 1 1 0 0 1
4	1	X	1	0100	0 1 1 0 0 1 1
5	1	X	1	0101	1 0 1 1 0 1 1
6	1	X	1	0110	1 0 1 1 1 1 1
7	1	X	1	0111	1 1 1 0 0 0 0
8	1	X	1	1000	1 1 1 1 1 1 1
9	1	X	1	1001	1 1 1 1 0 1 1
BI	X	X	0	XXXX	0 0 0 0 0 0 0
LT	0	X	1	XXXX	
RBI	1	0	0	0000	0 0 0 0 0 0 0

Common cathode

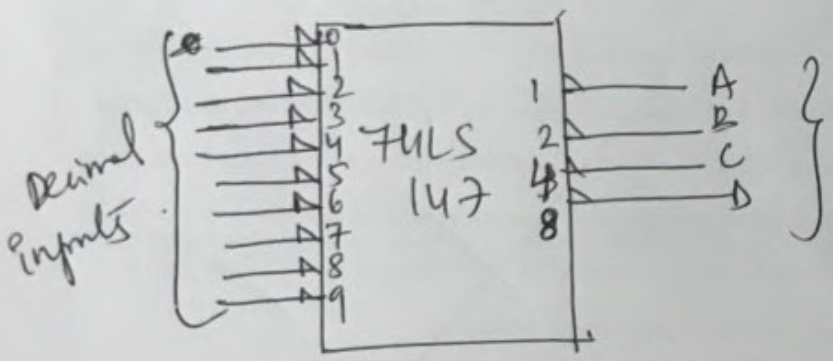


Common Anode



10 line to BCD encoder

- Encoding decimal keypad or switch panel output data into BCD prior to transmission to a digital system is a typical application of 74XX147.
- Consider a 10-key keypad used to input set point values into a digital control system.
- When the key is pressed it forces its corresponding digit line to a logic 0. That value is presented to the encoder producing a BCD output that is in turn sent to the digital control system for processing.

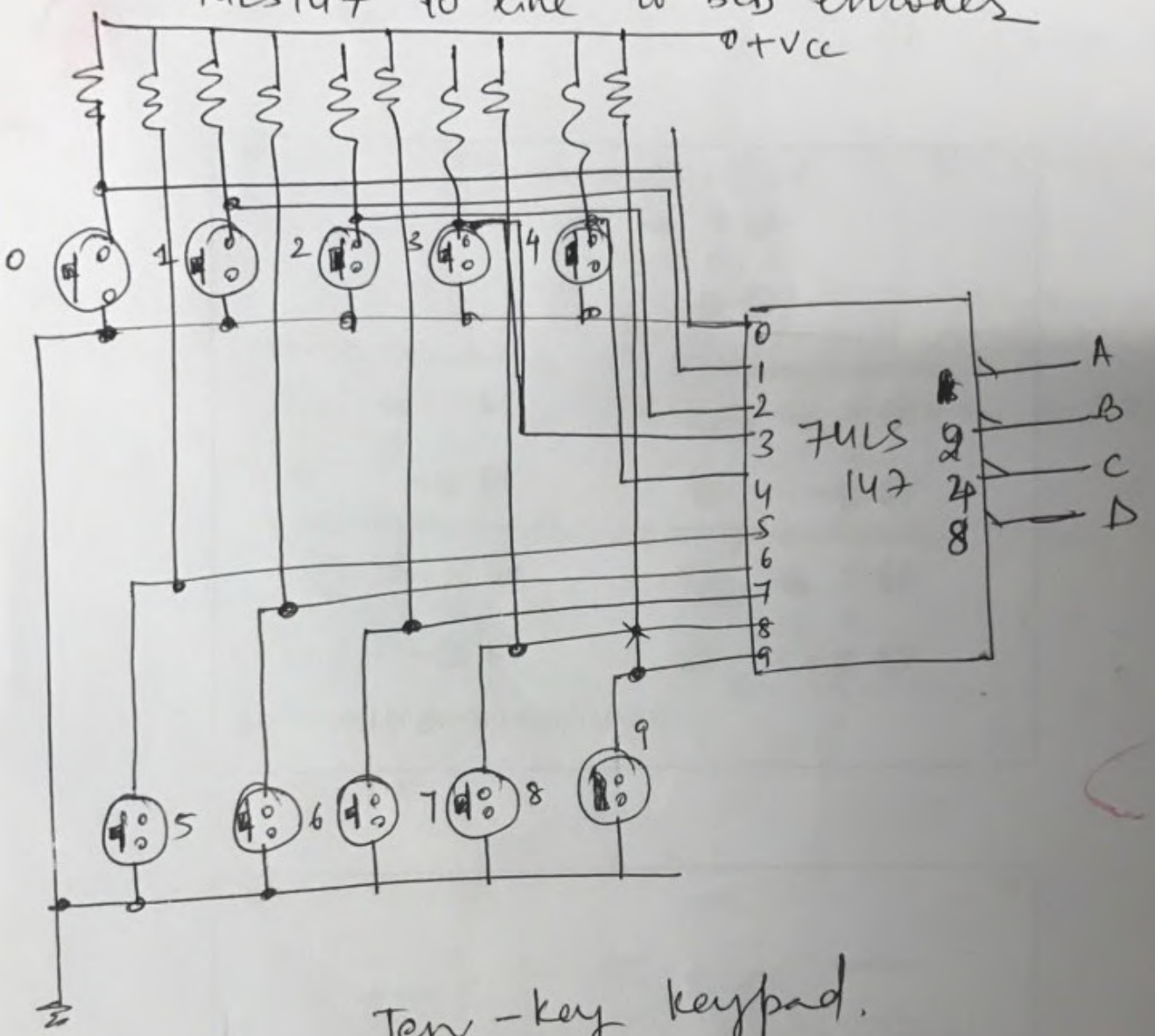


Truth table for a 10-line to BCD encoder

0 1 2 3 4 5 6 7 8 9	D C B A
0 0 0 0 0 0 0 0 0 0	1 1 1 1
1 0 0 0 0 0 0 0 0 0	0 0 0 0
X 1 0 0 0 0 0 0 0 0	0 0 0 1
X X 1 0 0 0 0 0 0 0	0 0 1 0
X X X 1 0 0 0 0 0 0	0 0 1 1
X X X X 1 0 0 0 0 0	0 1 0 0
X X X X X 1 0 0 0 0	0 1 0 1
X X X X X X 1 0 0 0	0 1 1 0
X X X X X X X 1 0 0	0 1 1 1
X X X X X X X X 1 0	1 0 0 0
X X X X X X X X X 1	1 0 0 1

Q.8C

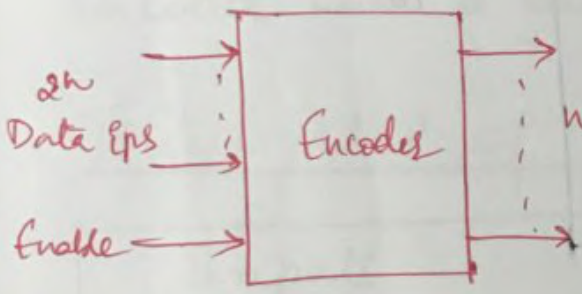
Keypad interface to a digital System using 74LS147 10 line to BCD encoder



Ten-key keypad.

ENCODERS.

- perform a function that is the inverse of decoders.
- Encoders have multiple inputs and outputs. Encoders have more input than output variables.
- An encoder produces n outputs from 2^n inputs.
- A typical encoder is as shown.



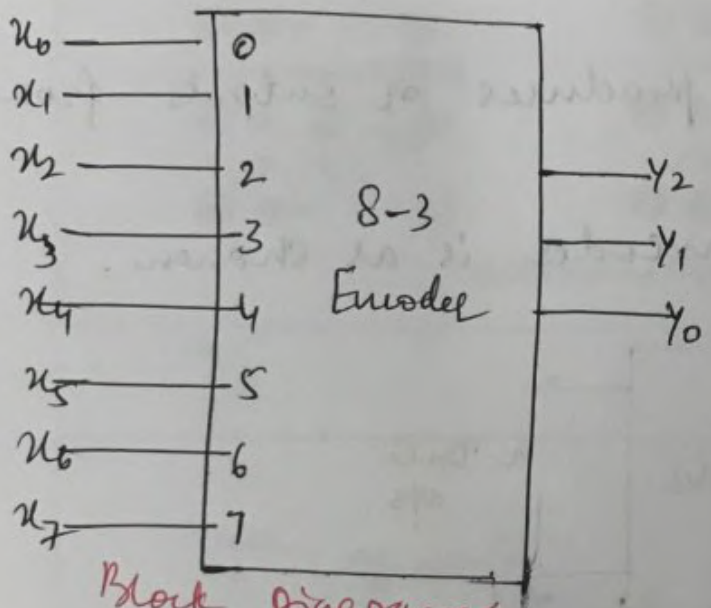
- Some of the examples of encoders are 4 to 2 line and 8 to 3 line encoders.

Truth table of 8 to 3 line encoder.

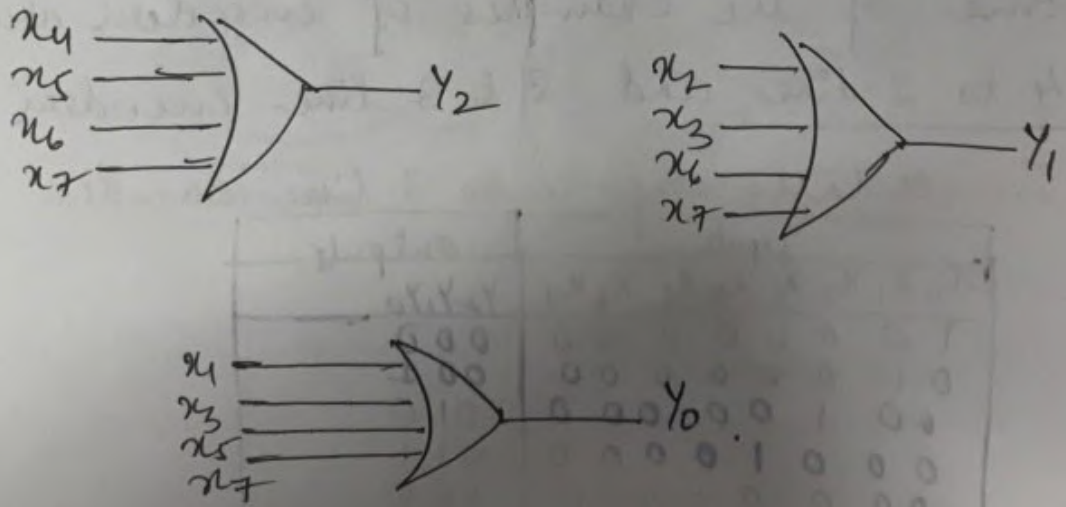
Inputs								outputs		
X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7	Y_2	Y_1	Y_0
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

Boolean expression for the ops can be written as

$$Y_2 = x_4 + x_5 + x_6 + x_7$$
$$Y_1 = x_2 + x_3 + x_6 + x_7$$
$$Y_0 = x_1 + x_3 + x_5 + x_7$$



Block diagram



Implementation of 8-3 line Encoder.

Priority Encoder

- There are some problems associated with previous implementation of Encoder.
- It is possible that when more than one inputs are high at logic 1, there may be an error in the output code.

  For ex: if $x_3 = x_4 = 1$, then $Y_2Y_1Y_0 = 111$, whereas this op has resulted for $x_7 = 1$.

- problem can be overcome by the priority encoder with a validity indicator.

Truth Table of 8-3 line Priority Encoder

Inputs								outputs			
x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	Y_2	Y_1	Y_0	valid.
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
x	1	0	0	0	0	0	0	0	0	1	1
x	x	1	0	0	0	0	0	0	1	0	1
x	x	x	1	0	0	0	0	0	1	1	1
x	x	x	x	1	0	0	0	1	0	0	1
x	x	x	x	x	1	0	0	1	0	1	1
x	x	x	x	x	x	1	0	1	1	0	1
x	x	x	x	x	x	x	1	1	1	1	1

Write the condensed truth table for a 4 to 2 line priority encoder with a valid op where the highest priority is given to the highest bit position of input with highest index and obtain the minimal expressions for the ops.

Soln

	x_0	x_1	x_2	x_3	Y_1	Y_0	valid
0	0	0	0	0	0	0	0
8	1	0	0	0	0	0	1 ✓
4,12	X	1	0	0	0	1	1 ✓
2,6,10,14	X	X	1	0	1	0	1 ✓
1,3,5,7,9 11,13,15	X	X	X	1	1	1	1 ✓

$x_3x_2x_1x_0$	Y_1Y_0
1000	00
100	01
0010	10
0001	11 ✓

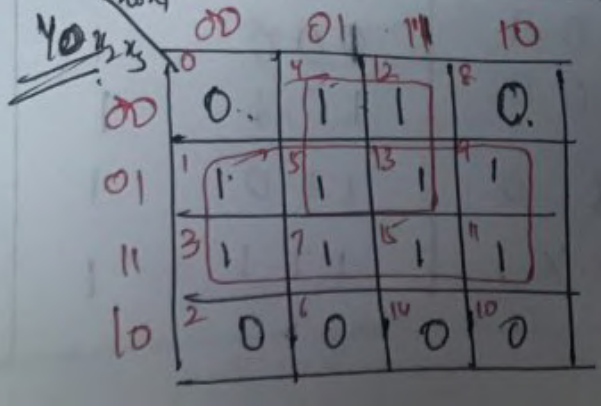
$x_3x_2x_1x_0$	Y_1Y_0
000 1	00
001 0	01
0100	10
1000	11

0001
001X

Y_1

$Y_1 = x_2 + x_3$

Y_0



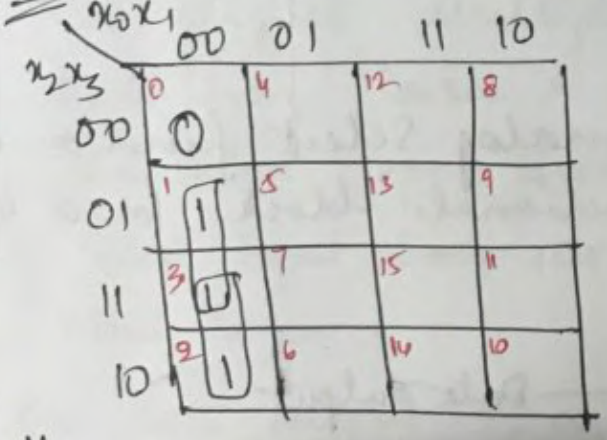
$Y_0 = x_3 + \bar{x}_2 x_1$

Repeat the problem (previous) assigning a highest priority to the least significant input & input with lowest index.

Soln

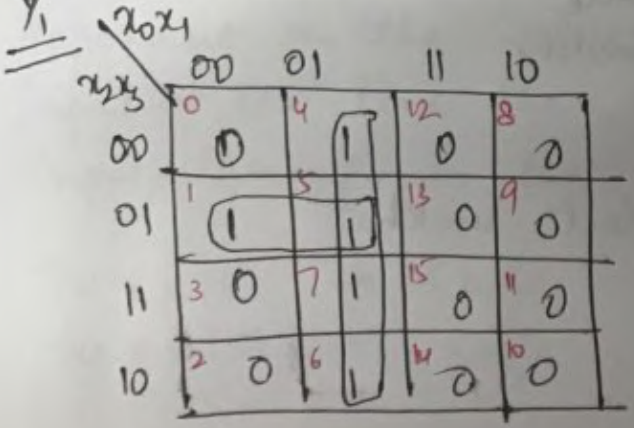
cell	$x_0\ x_1\ x_2\ x_3$	$Y_1\ Y_0$	valid
0	0 0 0 0	0 0	0
8, 9, 10, 11, 12 13, 14, 15	1 X X X	0 0	1
4, 5, 6, 7	0 1 X X	0 1	1
2, 3	0 0 1 X	1 0	1
1	0 0 0 1	1 1	1

Y_0



$$Y_0 = \bar{x_0}\bar{x_1}x_3 + \bar{x_0}\bar{x_1}x_2$$
$$= \bar{x_0}\bar{x_1}(x_2 + x_3)$$

Y_1



$$Y_1 = \bar{x_0}x_1 + \bar{x_0}\bar{x_2}x_3$$
$$= \bar{x_0}(x_1 + \bar{x_2}x_3)$$

DIGITAL MULTIPLEXERS.

- Digital Multiplexers provide the digital equivalent of an analog selector switch.
- A digital Multiplexer connects one of n inputs to a single output line, so that the logical value of the input is transferred to the output.
- The one of n input selection is determined by m select inputs, where $n = 2^m$.
- A 4 to 1 multiplexer requires 2 select inputs and an 8 to 1 multiplexers requires 3 select inputs.
- Fig shows the analog select function and the digital functional block for a 4:1 MUX.

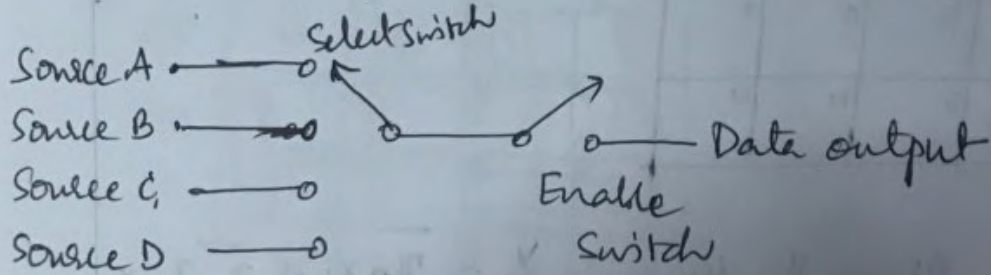


fig : Analog Select Switch

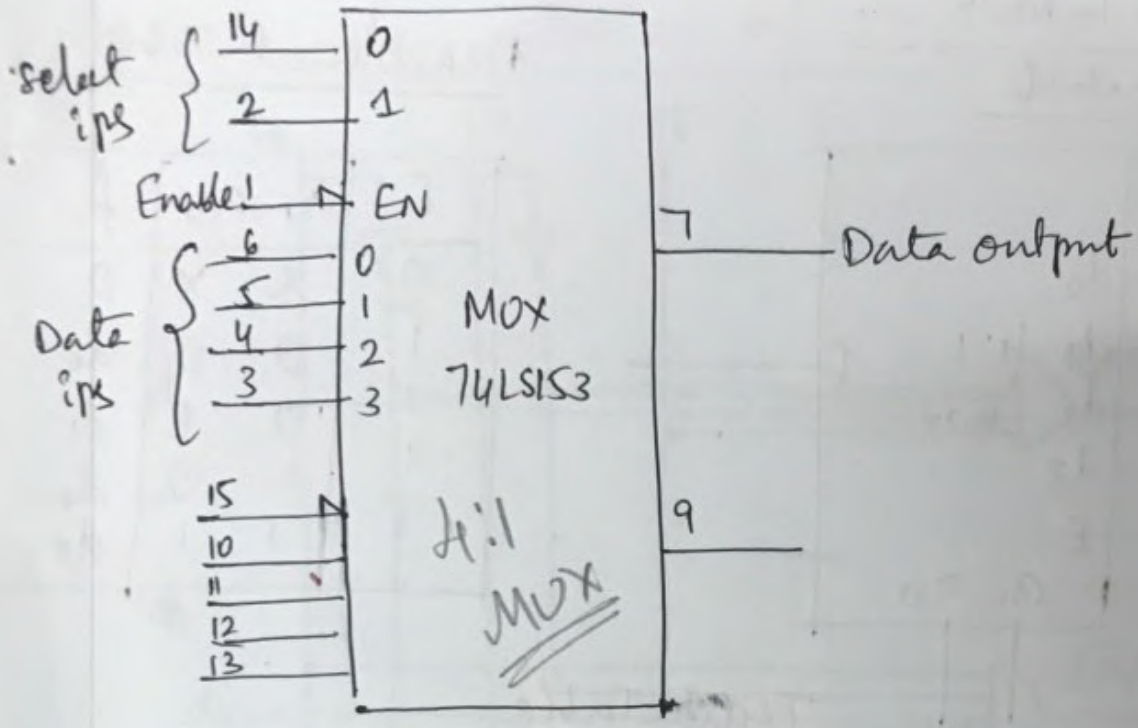


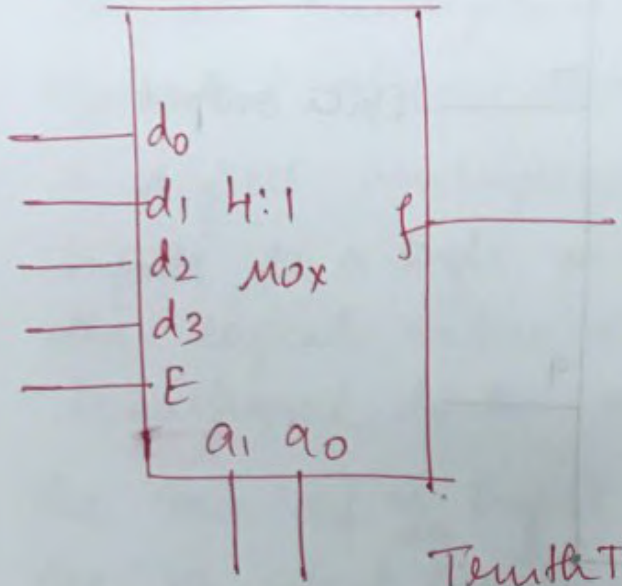
fig: IEEE Symbol for a dual 4:1 MUX.

- The digital Multiplexer routes digital data from Source A to the output when the select input lines are 00.
- If select input lines are 01, Source B is selected and so on.
- The data output of a digital Multiplexer is dependent on the value of the data at a given input at the time the select lines switch the input through to the output.

74XX151 → Single 8:1 Mux
74XX150 → Single 16:1 Mux

4 : 1 MUX.

Symbol



Functional Table

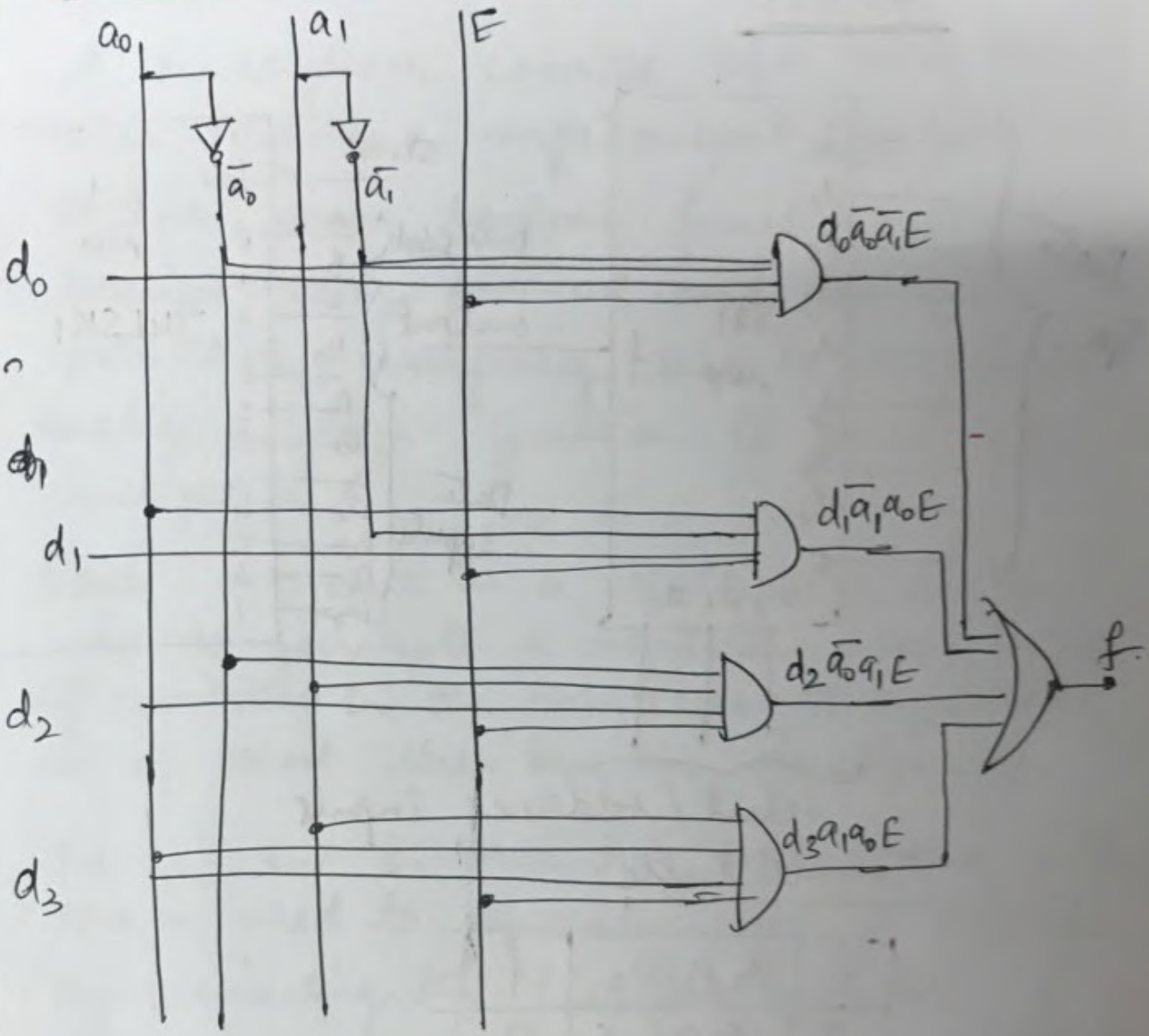
E	a_1	a_0	f
0	X	X	0
1	0	0	d_0
1	0	1	d_1
1	1	0	d_2
1	1	1	d_3

Truth Table.

E	a_1 a_0	d_0 d_1 d_2 d_3	f.
0	X X	X X X X	0
1	0 0	0 X X X	0 (d_0)
1	0 0	1 X X X	1 (d_0)
1	0 1	X 0 X X	0 (d_1)
1	0 1	X 1 X X	1 (d_1)
1	1 0	X X 0 X	0 (d_2)
1	1 0	X X 1 X	1 (d_2)
1	1 1	X X X 0	0 (d_3)
1	1 1	X X X 1	1 (d_3)

$$f = d_0 \bar{a_1} \bar{a_0} E + d_1 \bar{a_1} a_0 E + d_2 a_1 \bar{a_0} E + d_3 a_0 a_1 E$$

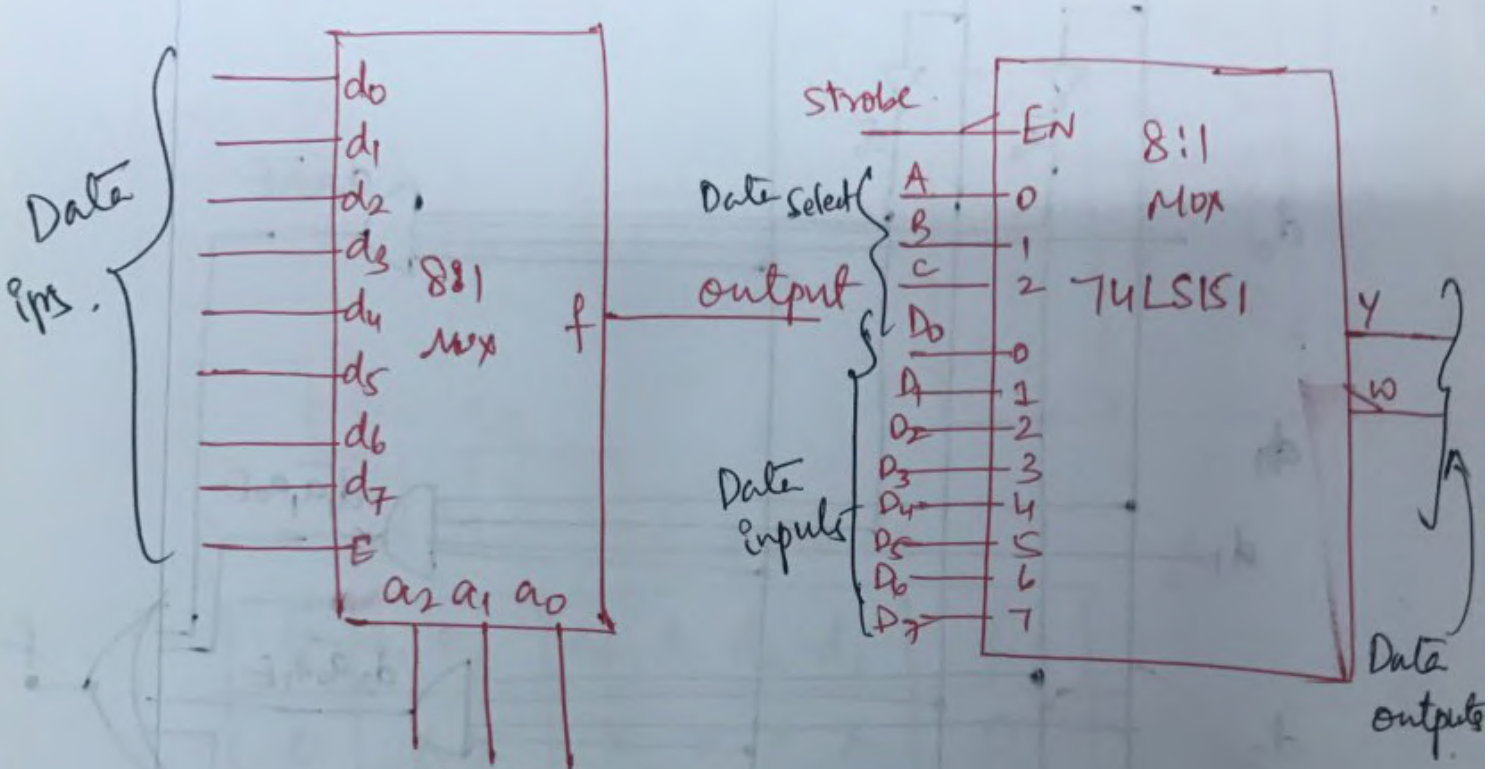
Implementation of 4:1 MUX



- when E=0, irrespective of the other inputs f=0.
- when E=1, the data at the addressed or selected input appears at the output

8:1 Mux.

Symbol



Truth table.

E	a_2 a_1 a_0	f
0	x x x	0
1	0 0 0	d_0
1	0 0 1	d_1
1	0 1 0	d_2
1	0 1 1	d_3
1	1 0 0	d_4
1	1 0 1	d_5
1	1 1 0	d_6
1	1 1 1	d_7

Using Multiplexers as Boolean Function Generators.

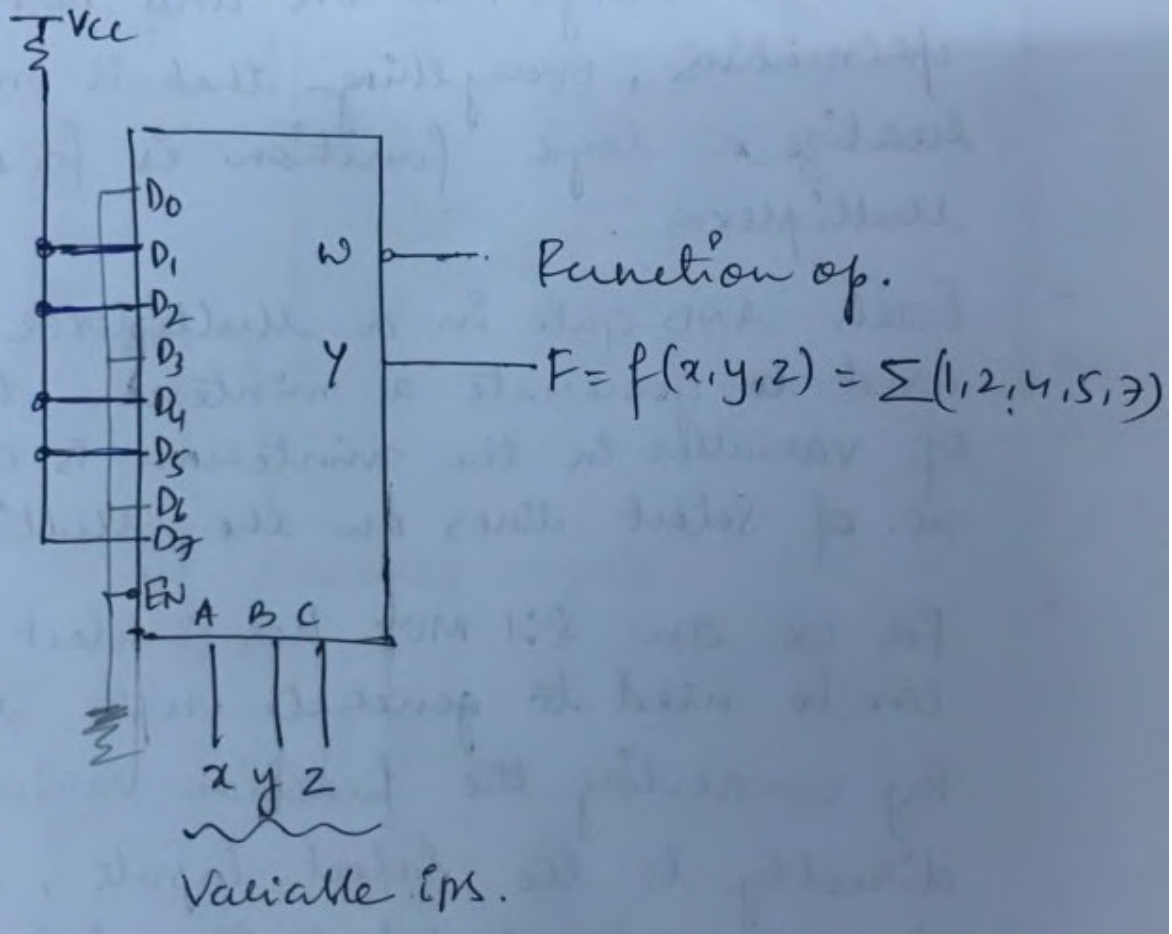
- A Multiplexer consists of a set of AND gates feeding a single output OR gate. Because any boolean function can be realized using AND-OR and NOT primitives, everything that is needed to realize a logic function is found in a Multiplexer.
- Each AND gate in a Multiplexer can be used to generate a minterm when the no. of variables in the minterm is equal to the no. of select lines on the Multiplexer.
- Fd ex: an 8:1 MUX has 3 select lines, so it can be used to generate upto 2^3 minterms. By connecting the function variables directly to the select inputs, a Multiplexer can be made to select the AND gate that corresponds to the minterm in the function.
- If a minterm exists in a function, we connect the AND gate data input to 1. If does not exist we connect it to a 0.

- Then, when a minterm occurs, the select lines switch the AND input 1 to the output.

Example 1: Realize the expression

$F = f(x,y,z) = \sum(1,2,4,5,7)$ using 8:1 MUX.

Soln:



Example 2: We can increase the no. of variables that a given Multiplexer can realize by connecting the multiplexer data inputs to 0, 1 variable or complemented variable.

For ex': a 4-variable Boolean function

2.15

$T = f(w,x,y,z) = \sum(0,1,2,4,5,7,8,9,12,13)$ can be realized using an 8:1 Mux.

- Three variables are used as select inputs and the fourth is connected as needed to the Mux data inputs.

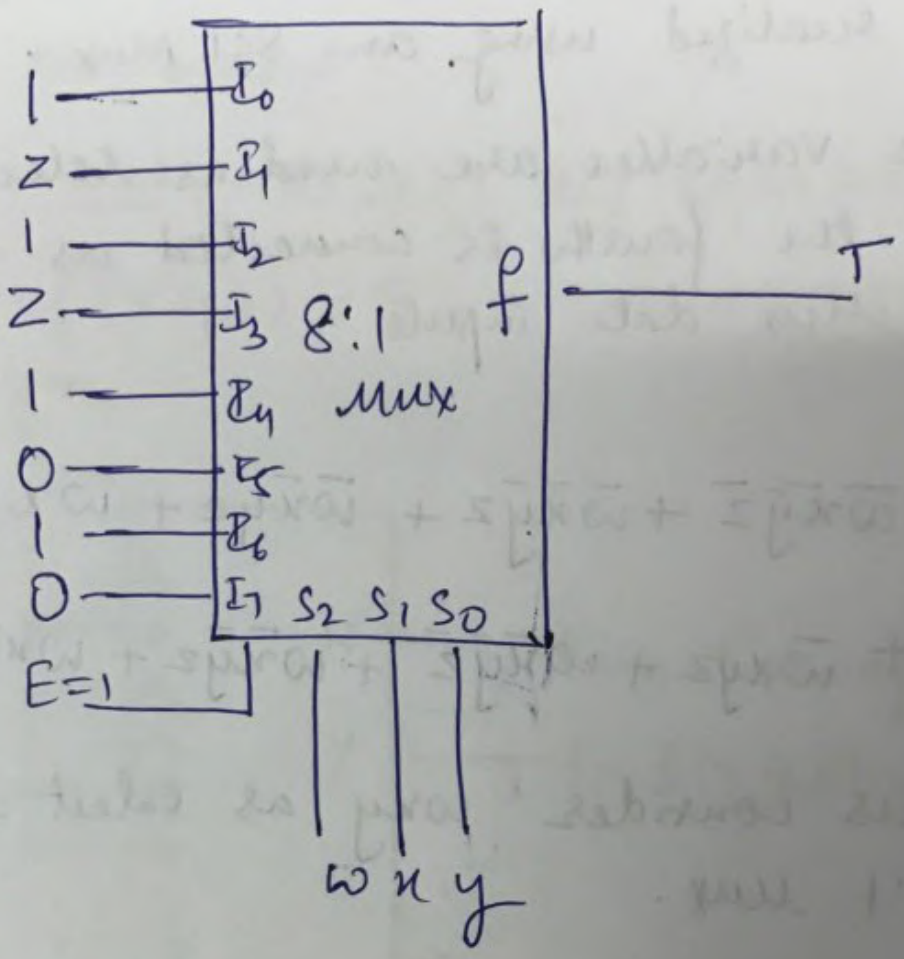
$$T = \bar{w}\bar{x}\bar{y}\bar{z} + \bar{w}\bar{x}\bar{y}z + \bar{w}\bar{x}yz + \bar{w}x\bar{y}\bar{z} + \bar{w}x\bar{y}z + \bar{w}xyz + w\bar{x}\bar{y}\bar{z} + w\bar{x}\bar{y}z + wx\bar{y}\bar{z} + wx\bar{y}z$$

- Let us consider wxy as select lines of 8:1 Mux.

$$T = \underset{000}{\bar{w}\bar{x}\bar{y}}(\bar{z}+z) + \underset{001}{\bar{w}\bar{x}y}(z) + \underset{010}{\bar{w}x\bar{y}}(z+\bar{z}) + \underset{011}{\bar{w}xy}(z) + \underset{100}{w\bar{x}\bar{y}}(z+\bar{z}) + \underset{110}{wx\bar{y}}(z+\bar{z}) + \underset{101}{w\bar{x}y}(0) + \underset{111}{wxy}(0)$$

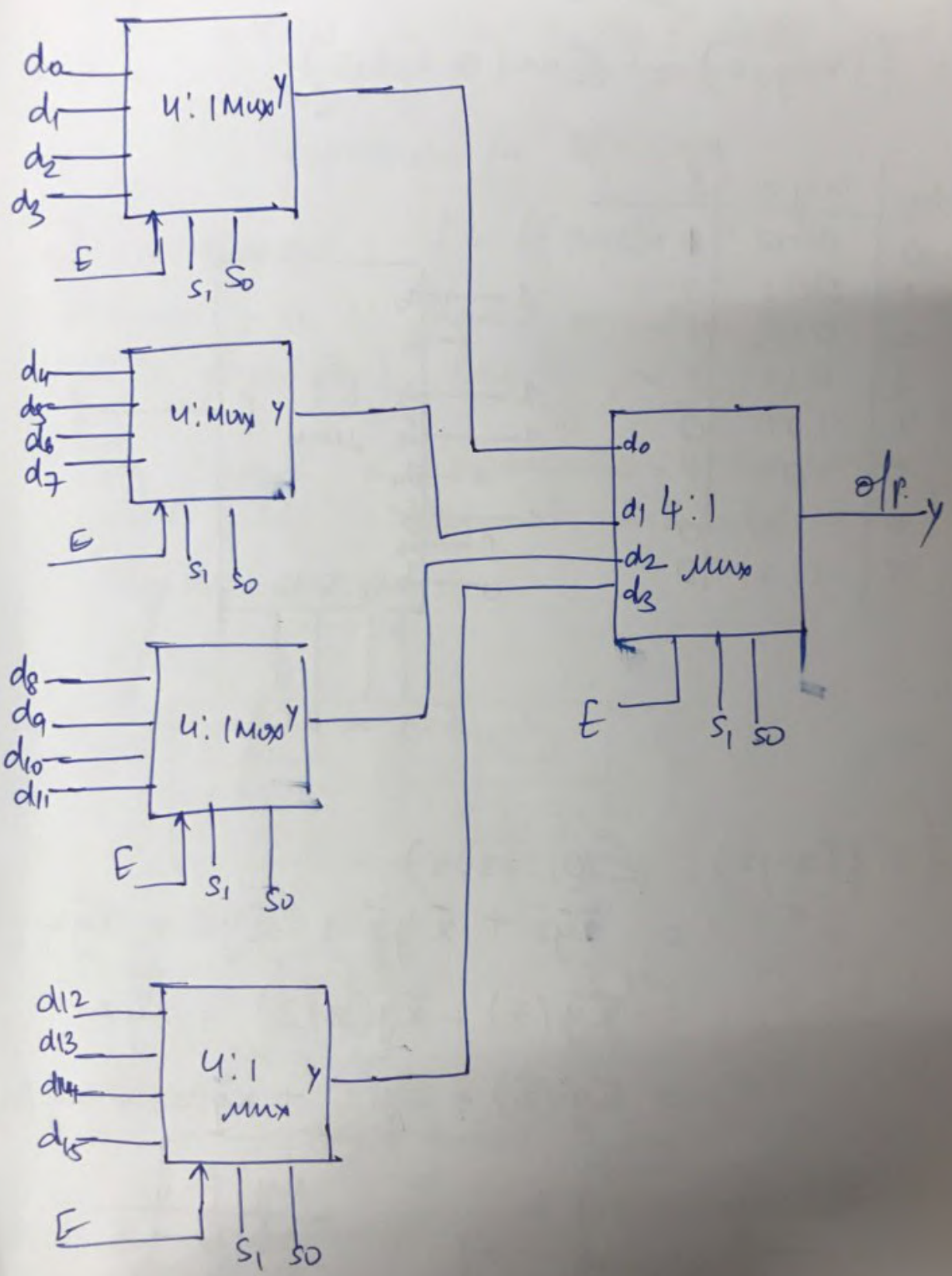
w x y $S_2 S_1 S_0$	T
000	$(z+\bar{z}) = 1$
001	z
010	$z+\bar{z} = 1$
011	z
100	$(z+\bar{z}) = 1$
101	0
110	$z+\bar{z} = 1$
111	0

P.T.O.



16:1 mux using 4:1 mux

2.15a

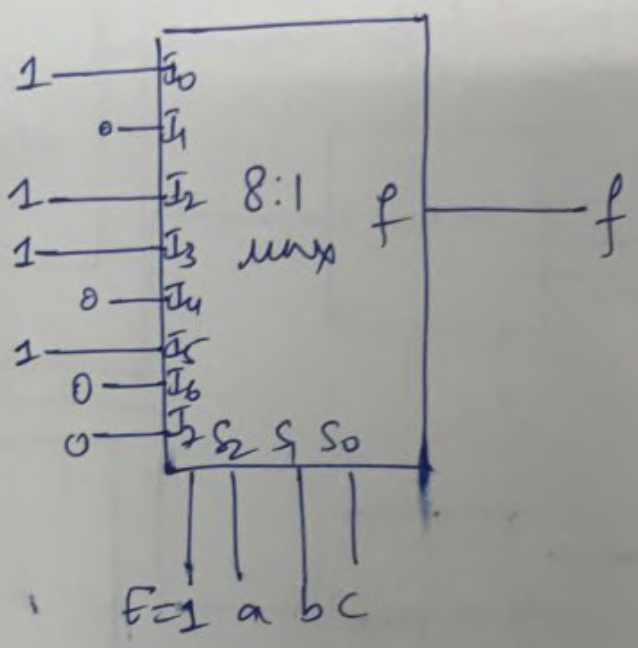


Realize the minterm canonical formula using 8:1 mux and 4:1 mux.

$$f = f(x,y,z) = \Sigma m(0,2,3,5)$$

Soln

dec	xyz	f
0	000	1 ✓
1	001	0
2	010	1 ✓
3	011	1 ✓
4	100	0
5	101	1 ✓
6	110	0
7	111	0

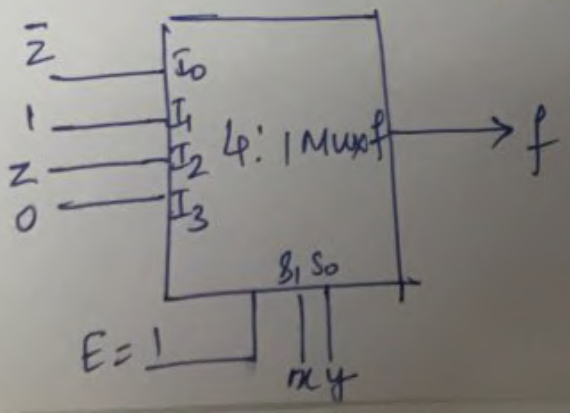


$$f = f(xyz) = \Sigma(0,2,3,5)$$

$$= \bar{x}\bar{y}\bar{z} + \bar{x}y\bar{z} + \bar{x}yz + x\bar{y}z$$

$$= \bar{x}\bar{y}(\bar{z}) + \bar{x}y(z+\bar{z}) + x\bar{y}z$$

$$= \bar{x}\bar{y}(\bar{z}) + \bar{x}y(1) + x\bar{y}(z) + xy(0)$$



xy	f
00	$I_0 \rightarrow \bar{z}$
01	$I_1 \rightarrow 1$
10	$I_2 \rightarrow z$
11	$I_3 \rightarrow 0$

2.15b

Using K-Map:

$$f = I_0\bar{x}\bar{y} + I_1\bar{x}y + I_2x\bar{y} + I_3xy$$ and $E=1$.

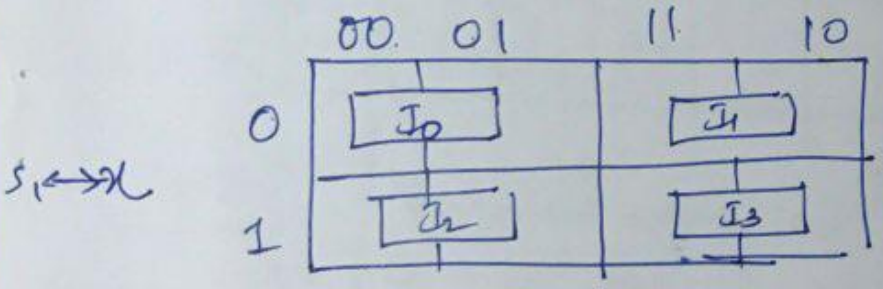
↓

Expression for 4:1 Mux.

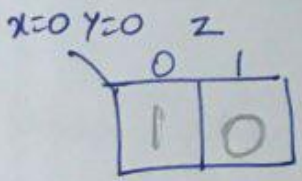
To implement $f = f(xyz) = \Sigma m(0,2,3,5)$

- Assume x is placed on the S_1 line and Y is placed on S_0 line
- Fig shows a 3-variable k-map along with this assignment indicated by double arrows.

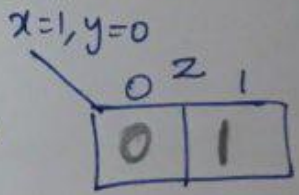
$S_0 \leftrightarrow yz$



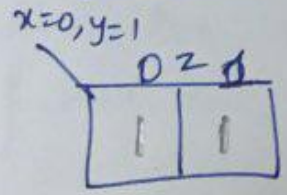
I_0 Map



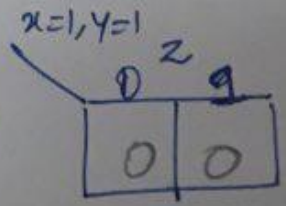
I_2 Map



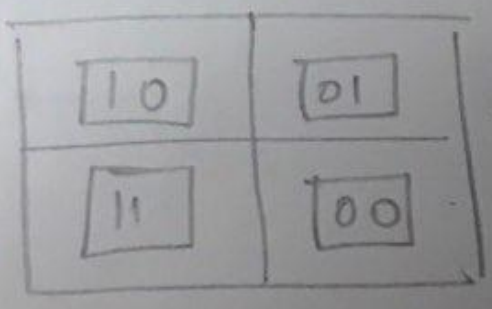
I_1 Map



I_3 Map



- $I_0 = \bar{z}$
- $I_1 = 1\,(z+\bar{z})$
- $I_2 = z$
- $I_3 = 0$



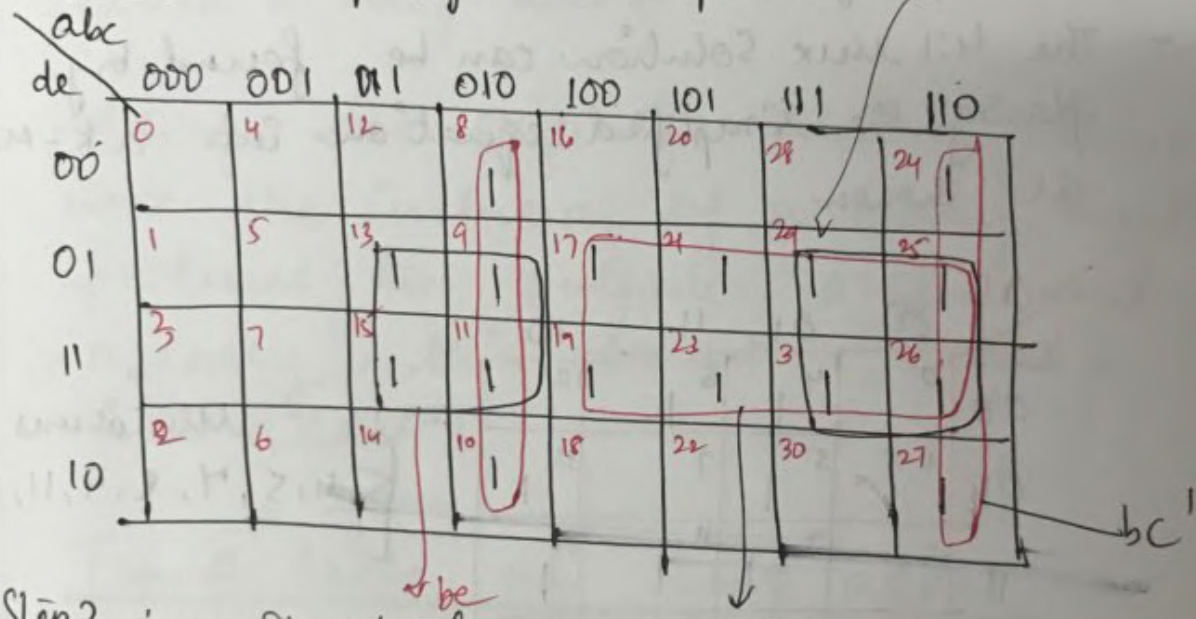
2.16

Realize the following Boolean function using least no. of ICs.

$S = f(a,b,c,d,e) = \sum(8,9,10,11,13,15,17,19,21,23,24,25,26,27,29,31)$

Soln

Step 1: Simplify the expression.

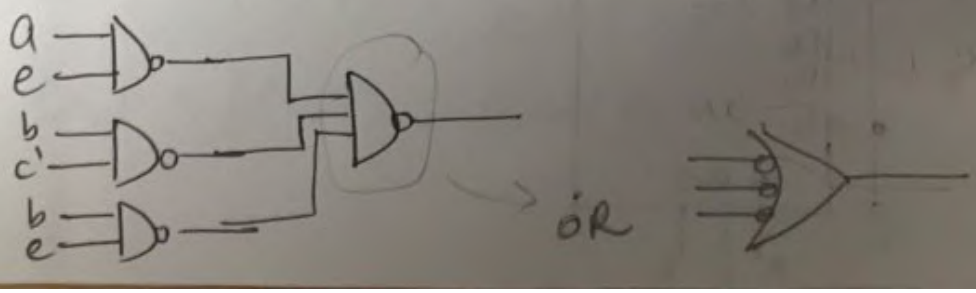


Step 2: Simplified expression ae

$S = be + bc' + ae$

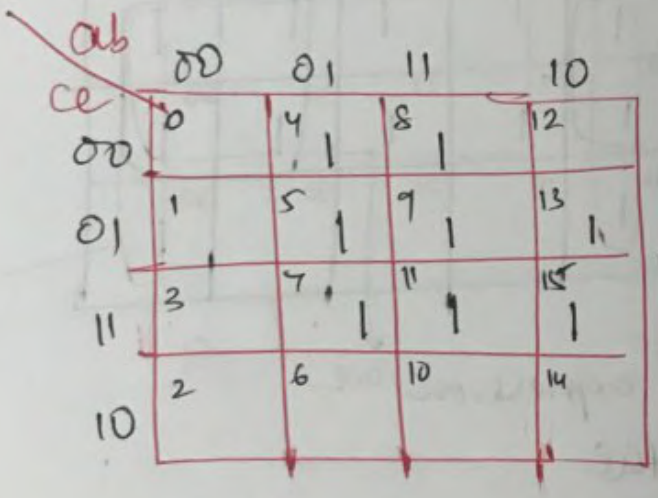
Step 3: Draw the logic diagram and determine the no. of ics needed to realize the function.

$\overline{S} = \overline{ae + bc' + be} = \overline{ae} \cdot \overline{bc'} \cdot \overline{be}$



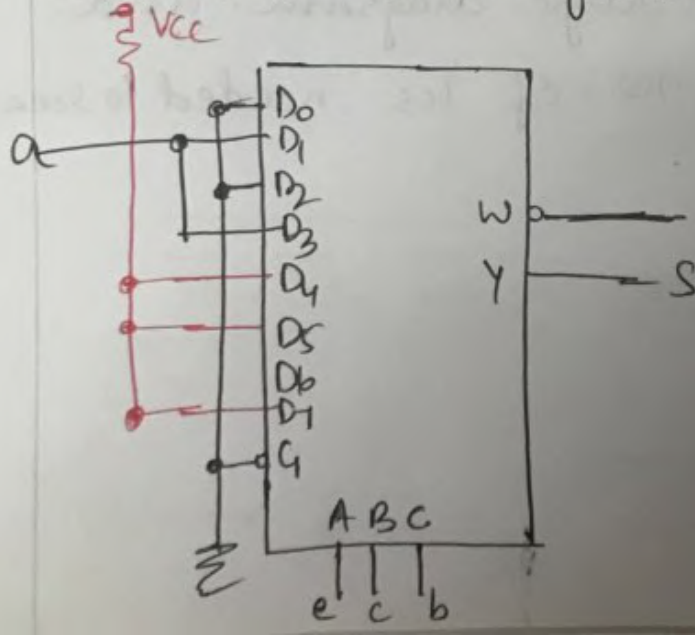
Step 4: Determine the Multiplexer size based on the no. of remaining Variables.

- Four Variables remain after simplification, permitting a single 8:1 or 4:1 mux Solution.
- The 4:1 mux Solution can be found by placing the simplified equation into a K-Map as shown.



Minterms $\sum 4, 5, 7, 8, 9, 11, 13, 15$

- The 8:1 Solution for the eqn is as shown.



$S = ae + bc' + be$

ADDERS & SUBTRACTORS.

- Adding 2 single-bit binary values produces a sum and a carry output. This operation is called a half-add and the circuit that realizes the function is called a half-adder.

- Adding two single-bit binary values with the inclusion of a carry input produces two outputs, a sum and a carry, this circuit is called a full adder.

Truth table of Half adder

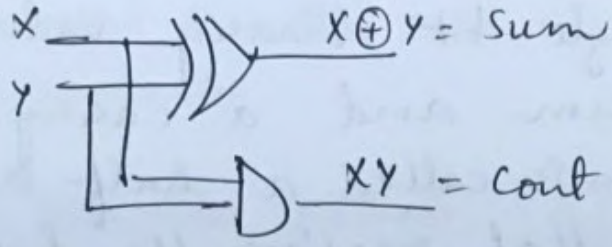
	X	Y	S	Cout
0	0	0	0	0
1	0	1	1	0
2	1	0	1	0
3	1	1	0	1

Boolean Equations.

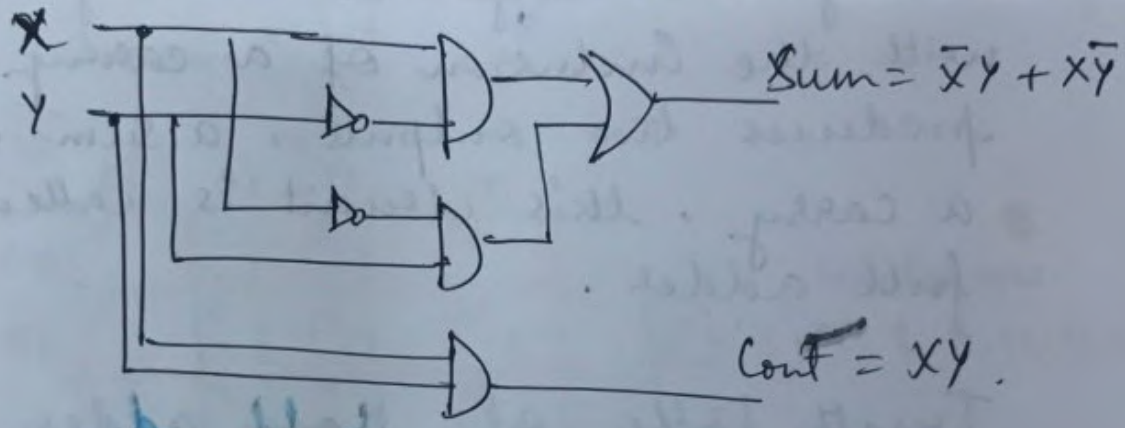
$$S = f(X,Y) = \sum m(1,2) = X'Y + XY' = X \oplus Y$$

$$Cout = f(X,Y) = \sum m(3) = XY$$

Half Adder Logic Diagram.



(or)



Using NOR

Using NAND

Truth Table of Full adder.

	Inputs			outputs	
	X	Y	Cin	S	Cout
0	0	0	0	0	0
1	0	0	1	1	0
2	0	1	0	1	0
3	0	1	1	0	1
4	1	0	0	1	0
5	1	0	1	0	1
6	1	1	0	0	1
7	1	1	1	1	1

Boolean Equations

$$S = f(X, Y, C_{in}) = \sum m(1, 2, 4, 7)$$

$$C_{out} = f(X, Y, C_{in}) = \sum m(3, 5, 6, 7)$$

Full Adder K-Maps.

S

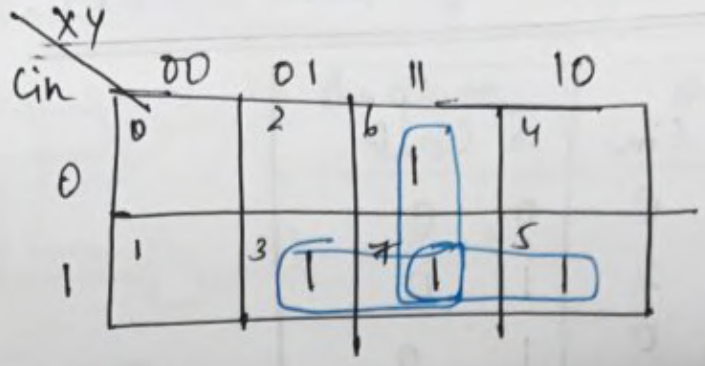
C_{in} \ XY	00	01	11	10
0	0	2 1	6	4 1
1	1 1	3	7 1	5

$$S = \bar{X}Y\bar{C_{in}} + \bar{X}\bar{Y}C_{in} + XYC_{in} + X\bar{Y}\bar{C_{in}}$$

$$= C_{in}(\bar{X}Y + X\bar{Y}) + C_{in}(\bar{X}\bar{Y} + XY)$$

$$\boxed{S = X \oplus Y \oplus C_{in}}$$

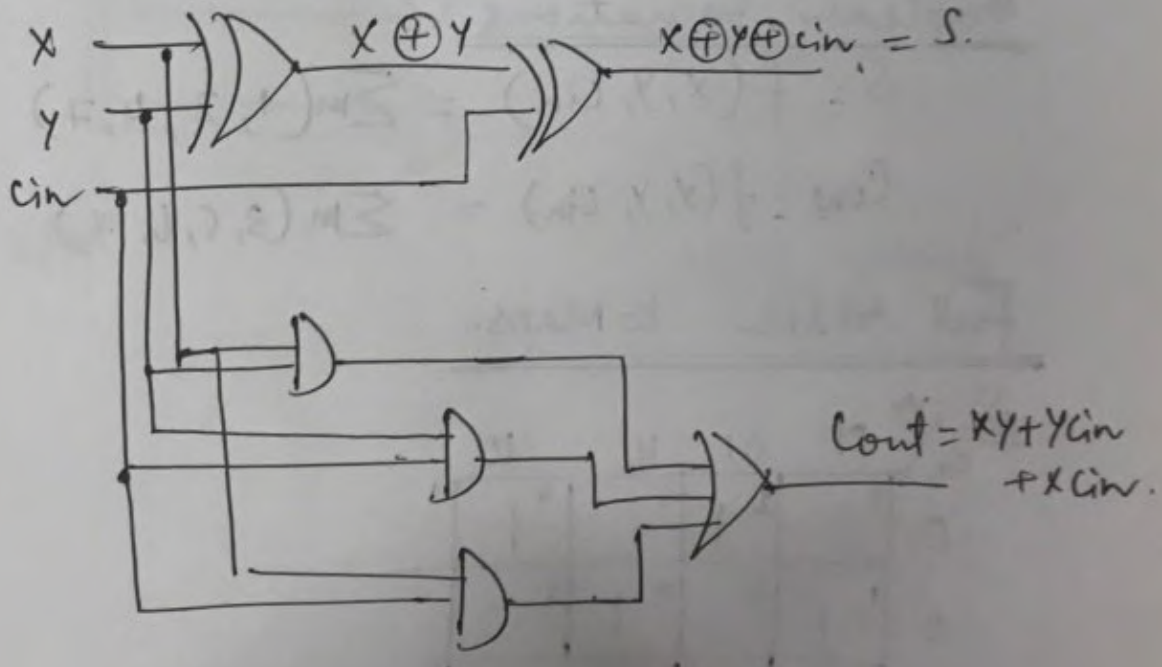
Cout



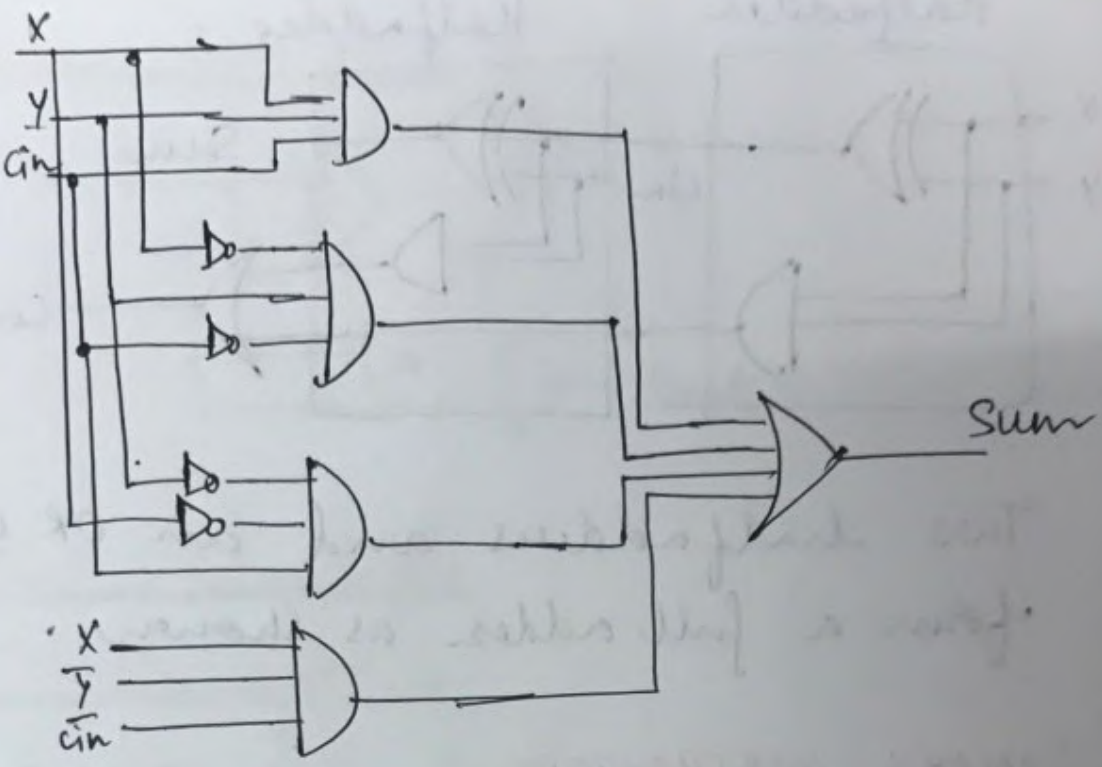
$$Cout = XY + YCin + XCin.$$

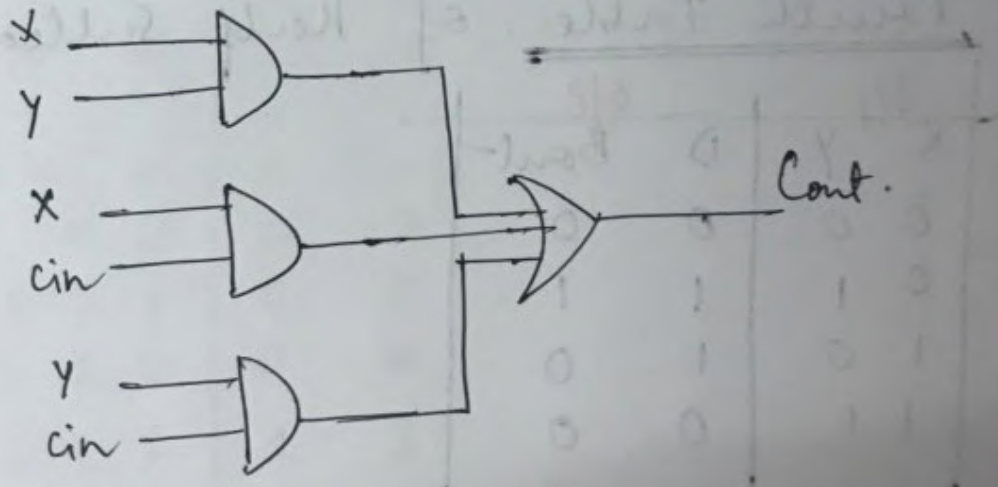
~~Fig~~

Full Adder Logic Diagram.



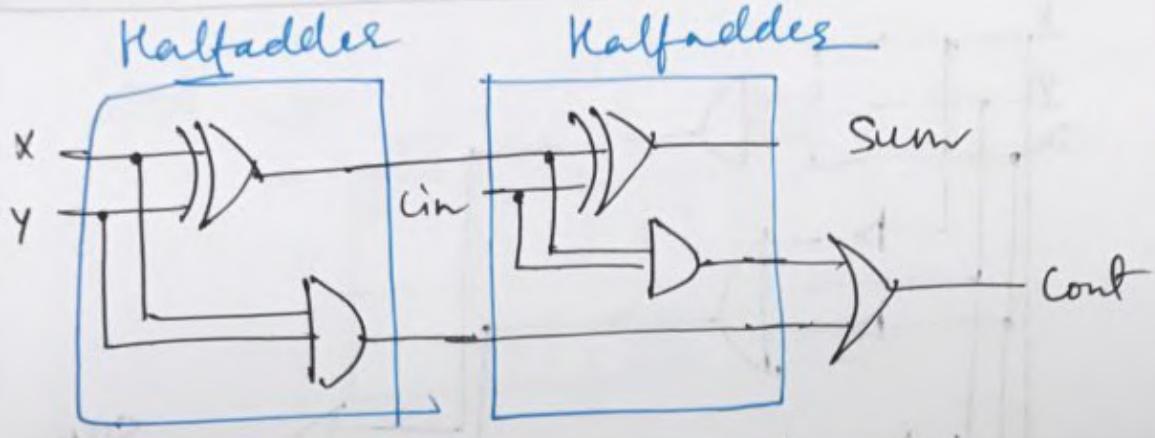
OR





- Two half adders circuits can be combined, along with an additional gate, to form a full adder.

P.T.O.



- Two halfadders and an OR gate form a full adder as shown.

BINARY SUBTRACTORS.

Truth Table of Half Subtractor

	I/P		O/P	
	X	Y	D	Bout
0	0	0	0	0
1	0	1	1	1
2	1	0	1	0
3	1	1	0	0

Boolean Equation.

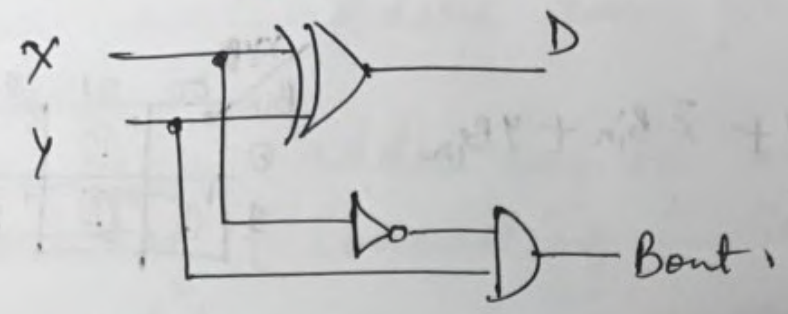
$$D = f(X,Y) = \Sigma m(1,2)$$

$$= \bar{X}Y + X\bar{Y} = X \oplus Y$$

$$Bout = f(X,Y) = \Sigma m(1)$$

$$= \bar{X}Y.$$

Logic Diagram.



Full Subtractor.

Truth Table.

	X	Y	Bin	D	Bout
0	0	0	0	0	0
1	0	0	1	1	1
2	0	1	0	1	1
3	0	1	1	0	1
4	1	0	0	1	0
5	1	0	1	0	0
6	1	1	0	0	0
7	1	1	1	1	1

Boolean Equation.

$$D = f(X, Y, B_{in}) = \sum m(1, 2, 4, 7)$$

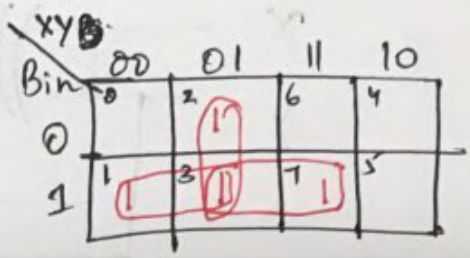
$$= \bar{X}\bar{Y}B_{in} + \bar{X}Y\bar{B_{in}} + X\bar{Y}\bar{B_{in}} + XYB_{in}$$

$$= B_{in}(\bar{X}\bar{Y} + XY) + \bar{B_{in}}(\bar{X}Y + X\bar{Y})$$

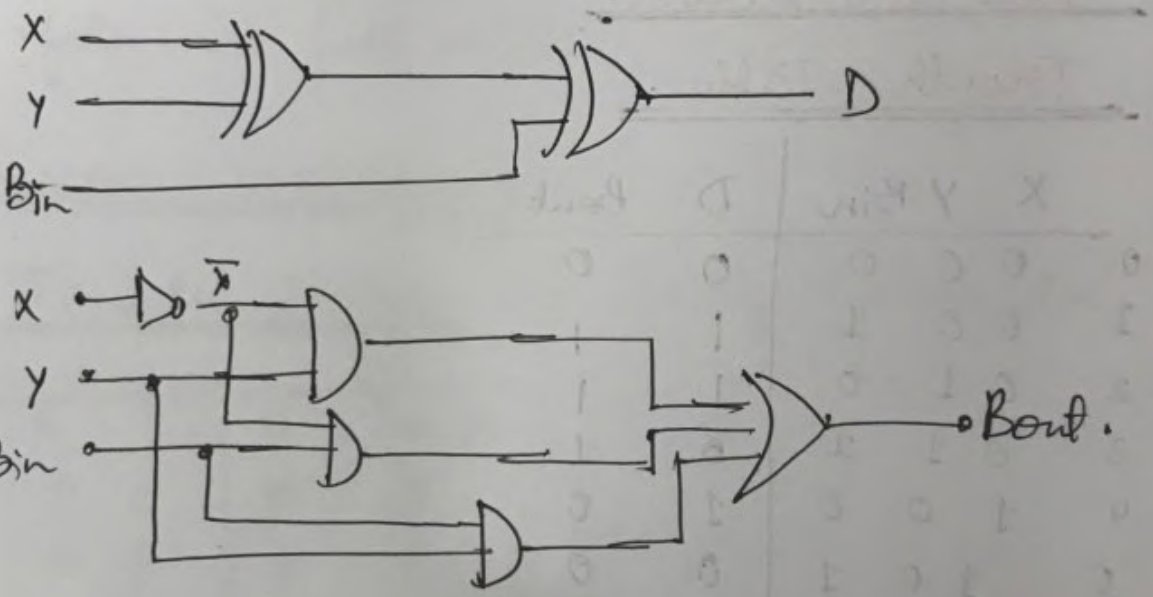
$$\boxed{D = X \oplus Y \oplus B_{in}}$$

$$B_{out} = f(x, y, B_{in}) = \sum m(1,2,3,7)$$

$$= \bar{x}y + \bar{x}B_{in} + yB_{in}$$

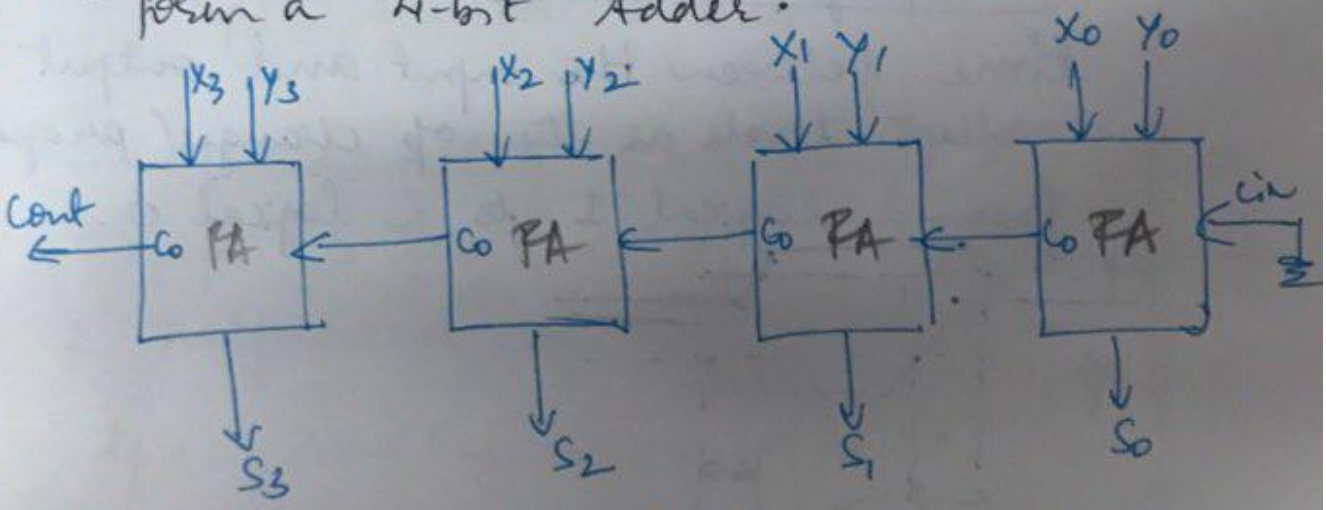


Logic Diagram.



Cascading Full adders.

- Two halfadders can form a full adder.
- An n-bit adder can be built by cascading (connecting in series) n full adders.
- Each full adder represents a bit position. The least significant bit position can be a half adder. because a carry from a less significant position does not exist.
- Each carry out from a fulladder becomes the carry-in to the next higher order adder.
- Fig shows cascading four full adders to form a 4-bit Adder.

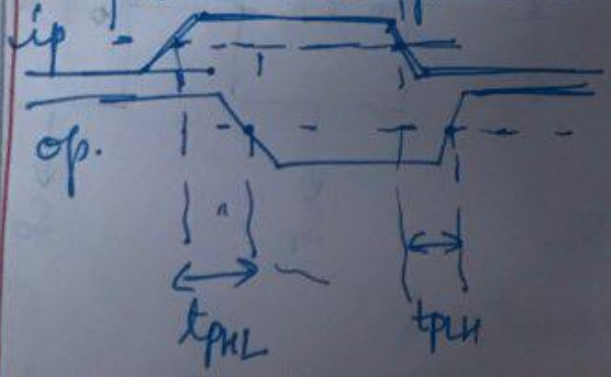


$S_3S_2S_1S_0$ → Sum outputs

$X_0X_1X_2X_3$ & $Y_0Y_1Y_2Y_3$ → Data operands.

- The carry-in input of the least significant full adder is grounded.
- 2 Binary inputs $X(X_3X_2X_1X_0)$ and $Y(Y_3Y_2Y_1Y_0)$ are presented to the adder from external source.
- The add time required is determined by calculating the propagation delays ** through the full adder.
- The carry propagate from one-full adder to the next higher-order full adder. This type of addition is called ripple Carry propagation (addition).

** Propagation delay → is the delay time between the input and output voltage levels as the op changes (propagates) from a logical 1 to a logical 0.



- Total add delay time is the product of the sum of the no. of stages in the adder and the carry-in to carry-out propagation delay time.

- The carry-in to carry-out propagation delay is used instead of the input to sum output delay because it is the signal that ripples from one full adder to another.

Let us Assume propagation delay for Xor gate be 8ns., OR gate – 5ns, And Gate – 6ns

So The propagation time from any data input (X, Y, Cin) to the sum o/p is

$$t_{pd} = 8ns + 8ns = 16ns.$$

- Propagation delay from any data-input (X, Y, Cin) to the C-out of a full adder using 2 half adders is

$$t_{pd} = 8ns + 6ns + 5ns = 19ns.$$

- The total propagation delay for the 4-bit Adder (using 2 half adders per adder) is

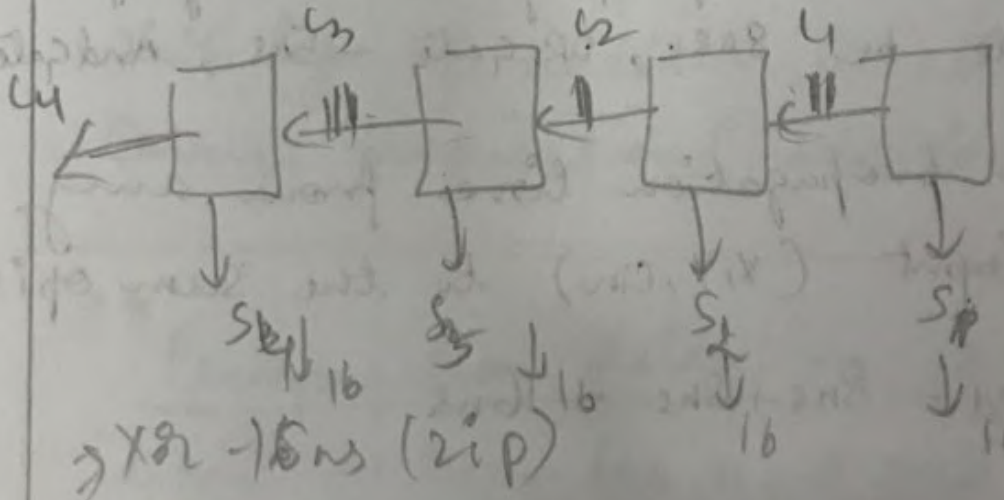
t_{pd} = 19ns + 3 (11ns)

= 19ns + 3 (11ns)

= 19ns + 33ns = 52ns.

↓ ↳

LSB Adder 11ns each for the other 3 Adders.



⇒ XOR - 16ns (2ip)

⇒ Carry C_4 = 44 ns.

S_4 = 33 + 16. = 49 ns

LOOK AHEAD CARRY ADDER

- The look ahead carry, or fast carry, technique reduces propagation times through the adder.
- This is accomplished by generating all the individual carry terms needed by each full adder in as few levels of logic as possible.
- The sum of any fulladder stage in a multiple stage adder is written

$$S_j = X_j \oplus Y_j \oplus C_{inj}$$

- The C_{inj} term must be generated based on all values of X and Y that precede the j^{th} full adder as shown in fig.

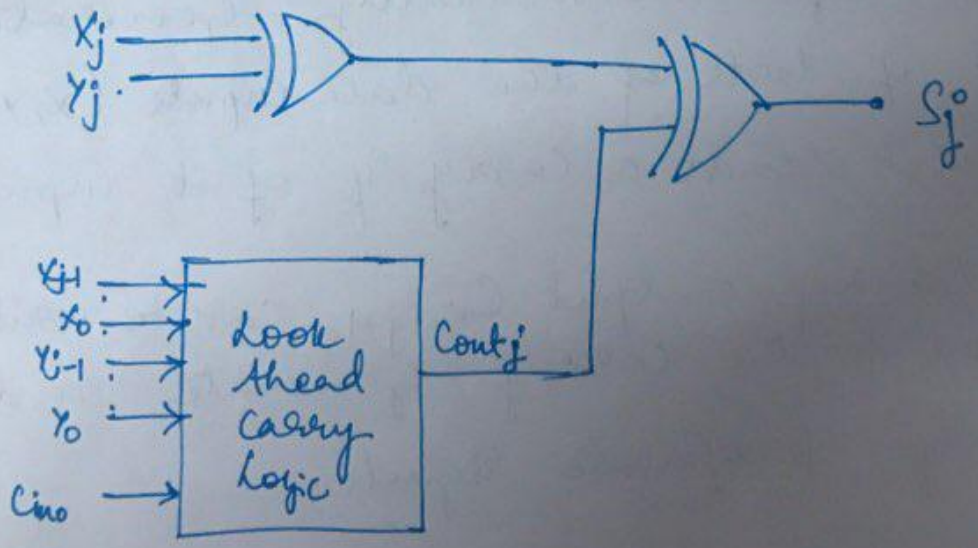


fig: generation of C_{inj} for a look ahead carry

by the exclusive-or-gates prior to entering the parallel binary adder and the necessary initial carry-in of 0 is provided.

Since the operands in subtraction can be either signed or unsigned, the output of a binary subtracter must be interpreted appropriately. For example, for unsigned operands, the output from the binary subtracter of Fig. 5.4 is the true difference if the minuend is greater than or equal to the subtrahend. However, the output from the subtracter is the 2's-complement representation of the difference if the minuend is less than the subtrahend. Again the reader is referred back to Chapter 2 for the details of binary arithmetic with signed and unsigned numbers.

5.1.2 Carry Lookahead Adder

In view of the fact that subtraction is readily achievable through the addition of complements, further discussion of the addition/subtraction process is restricted only to the realization of binary adders.

Although the operands are applied in parallel, all the networks illustrated thus far in this section are subject to a ripple effect. The ripple effect dictates the overall speed at which the network operates. To see this, consider the ripple binary adder of Fig. 5.3. It is possible that a carry is generated in the least-significant-bit-position stage, and, owing to the operands, this carry must propagate through all the remaining stages to the highest-order-bit-position stage. For example, such a situation occurs when the two n-bit operands are $01 \cdots 11$ and $00 \cdots 01$ so that the n-bit sum $10 \cdots 00$ is produced. Assuming the binary full adder of Fig. 5.2, two levels of logic are needed to generate the carry at the least-significant-bit-position stage, two levels of logic are needed to propagate the carry through each of the next n–2 higher-order stages, and two levels of logic are needed to form the sum or carry at the highest-order-bit-position stage. If each gate is assumed to introduce a unit time of propagation delay, then the maximum propagation delay for the ripple adder becomes $2n$ units of time. Of course, this is a worst-case condition. However, since normally all signals must complete their propagations through a network before new inputs are applied, this worst-case condition becomes a limiting factor in the network's overall speed of operation. To decrease the time required to perform addition, an effort must be made to speed up the propagation of the carries. One approach for doing this is to reduce the number of logic levels in the path of the propagated carries. Adders designed with this consideration in mind are called *high-speed adders*.

Equations (5.1) and (5.2) are the sum and carry equations for the outputs at the ith stage of a binary adder. As seen by these equations, the sum and carry outputs at a given stage are a function of the output carry from the previous stage, which, in turn, is a function of the output carry from still another previous stage, etc. This corresponds to the undesirable ripple effect. If the input carry at a given stage is expressed in terms of the operand variables themselves, i.e., $x_0, x_1, \ldots, x_{n-1}$ and $y_0, y_1, \ldots, y_{n-1}$, then the ripple effect is eliminated and the overall speed of the adder increased.

To see how this is done, again consider Eq. (5.1) for the output carry at the ith stage, i.e.,

$$\begin{aligned} c_{i+1} &= x_i y_i + x_i c_i + y_i c_i \\ &= x_i y_i + (x_i + y_i) c_i \end{aligned}$$

The first term in the last equation, $x_i y_i$, is called the *carry-generate function* since it corresponds to the formation of a carry at the ith stage. The second term, $(x_i + y_i)c_i$, corresponds to a previously generated carry c_i that must propagate past the ith stage to the next stage. The $x_i + y_i$ part of this term is called the *carry-propagate function.* Letting the carry-generate function be denoted by the Boolean variable g_i and the carry-propagate function by p_i, i.e.,

$$g_i = x_i y_i \tag{5.3}$$

$$p_i = x_i + y_i \tag{5.4}$$

the output carry equation for the ith stage is given by

$$c_{i+1} = g_i + p_i c_i$$

Using this general result, the output carry at each of the stages can be written in terms of just the carry-generate functions, the carry-propagate functions, and the initial input carry c_0 as follows:

$$c_1 = g_0 + p_0 c_0 \tag{5.5}$$

$$\begin{aligned} c_2 &= g_1 + p_1 c_1 \\ &= g_1 + p_1(g_0 + p_0 c_0) \\ &= g_1 + p_1 g_0 + p_1 p_0 c_0 \end{aligned} \tag{5.6}$$

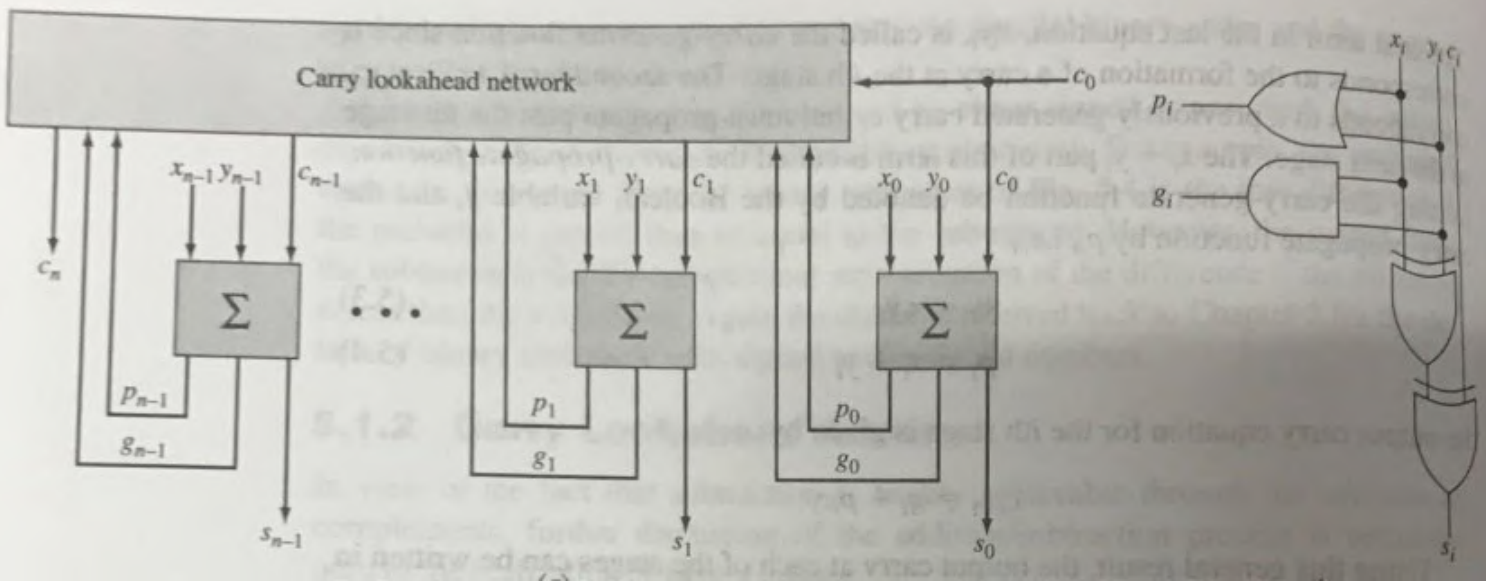
$$\begin{aligned} c_3 &= g_2 + p_2 c_2 \\ &= g_2 + p_2(g_1 + p_1 g_0 + p_1 p_0 c_0) \\ &= g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0 \end{aligned} \tag{5.7}$$

$$\begin{aligned} c_4 &= g_3 + p_3 c_3 \\ &= g_3 + p_3(g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0) \\ &= g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0 \end{aligned} \tag{5.8}$$

$$\vdots$$

$$c_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \cdots + p_i p_{i-1} \cdots p_1 g_0 + p_i p_{i-1} \cdots p_0 c_0 \tag{5.9}$$

Since each carry-generate function and carry-propagate function is itself only a function of the operand variables as indicated by Eqs. (5.3) and (5.4), the output carry and, correspondingly, the input carry, at each stage can be expressed as a function of the operand variables and the initial input carry c_0. In addition, since the output sum bit at any stage is also a function of the previous stage output carry as indicated by Eq. (5.2), it also can be expressed in terms of just the operand variables and c_0 by the substitution of an appropriate carry equation having the form of Eq. (5.9). Parallel adders whose realizations are based on the above equations are called *carry lookahead adders.* The general organization of a carry lookahead adder is shown in Fig. 5.7*a* where the carry lookahead network corresponds to a logic network based on Eqs. (5.5) to (5.9). The sigma blocks correspond to the logic needed to form the sum bit, the carry-generate function, and the



BINARY COMPARATORS.

- A binary magnitude comparator is a logic circuit that provides output information indicating the relative magnitude of 2 input operands.
- 3 output conditions exist as a result of comparison of 2 operands.
  - i) A is equal to B.
  - ii) A is greater than B
  - iii) A is less than B.

1-Bit comparator.

- Consider the case where both operands A and B are single bit binary numbers.

Truth Table.

ips		ops.		
A	B	A=B	A>B	A<B
0	0	1	0	0
0	1	0	0	1
1	0	0	1	0
1	1	1	0	0

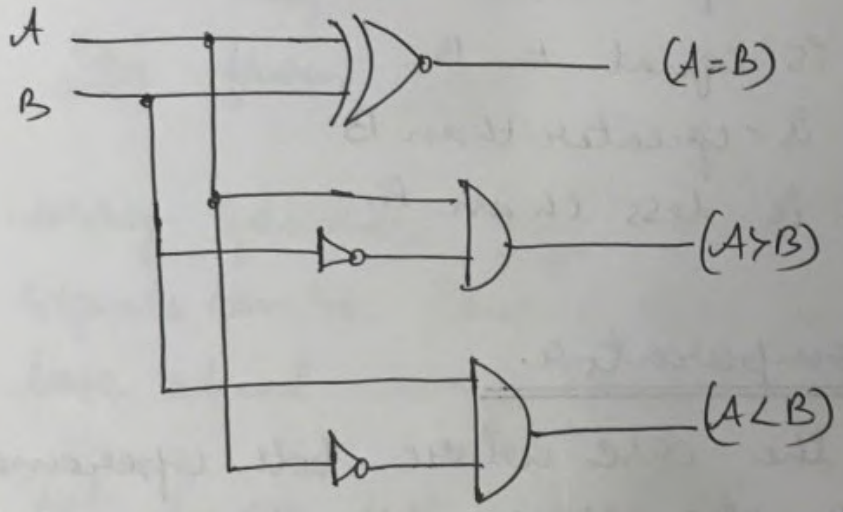
- The equations for each output are

$(A=B) = A'B' + AB = (A \oplus B)'$

$(A>B) = AB'$

$(A<B) = A'B.$

Logic Diagram.



2-Bit comparator.

- Increasing the size of the operand to two bits results in

$(A=B) = f(a_1, a_0, b_1, b_0) = \sum(0, 5, 10, 15)$

$(A>B) = f(a_1, a_0, b_1, b_0) = \sum(4, 8, 9, 12, 13, 14)$

$(A<B) = f(a_1, a_0, b_1, b_0) = \sum(1, 2, 3, 6, 7, 11)$

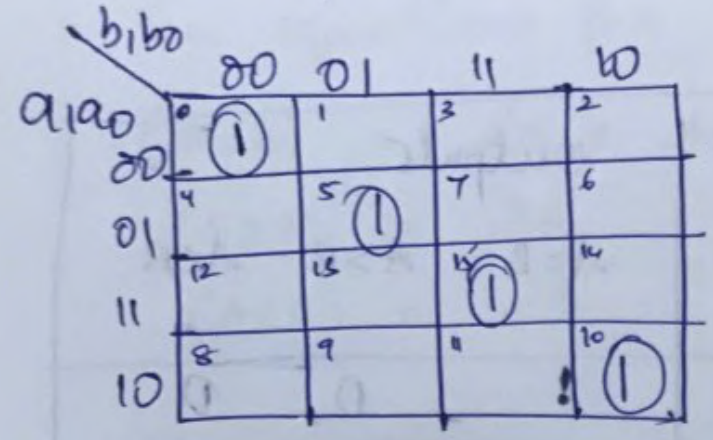
2.26.

TRUTH TABLE.

Inputs A B $a_1 a_0$ $b_1 b_0$	outputs A=B	A>B	A<B
0 0 0 0	1	0	0
0 0 0 1	0	0	1
0 0 1 0	0	0	1
0 0 1 1	0	0	1
0 1 0 0	0	0	1
0 1 0 1	1	0	0
0 1 1 0	0	0	1
0 1 1 1	0	0	1
1 0 0 0	0	1	0
1 0 0 1	0	1	0
1 0 1 0	1	0	0
1 0 1 1	0	0	1
1 1 0 0	0	1	0
1 1 0 1	0	1	0
1 1 1 0	0	1	0
1 1 1 1	1	0	0

P.T.O

K map for $A=B = \Sigma(0,5,10,15)$



$(A=B) = \bar{a}_1\bar{a}_0\bar{b}_1\bar{b}_0 + \bar{a}_1a_0\bar{b}_1b_0 + a_1a_0b_1b_0 + a_1\bar{a}_0b_1\bar{b}_0$

K map for $(A>B) = \Sigma(4,8,9,12,13,14)$

$(A>B) = a_0\bar{b}_1\bar{b}_0 + a_1\bar{b}_1 + a_1a_0\bar{b}_0$

K map for $(A<B) = \Sigma(1,2,3,6,7,11)$

$(A<B) = \bar{a}_1\bar{a}_0b_0 + \bar{a}_1b_1 + \bar{a}_0b_1b_0$

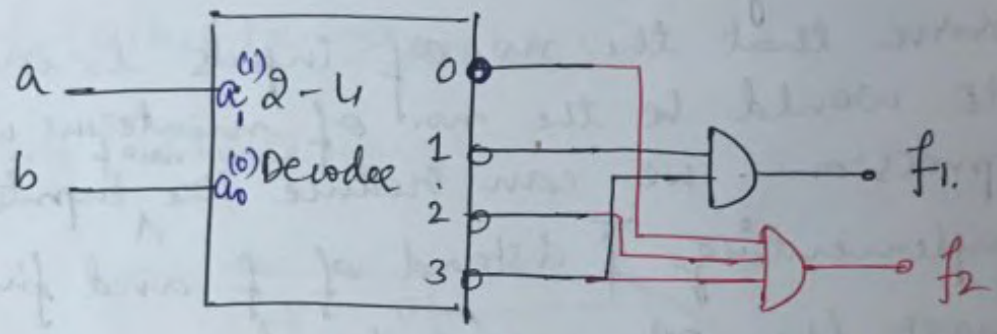
Problems – Module 2.

1. Implement the following functions in maxterm canonical form using 2 to 4 line decoder with active low outputs.

$f_1(a,b) = \Pi M(1,3)$

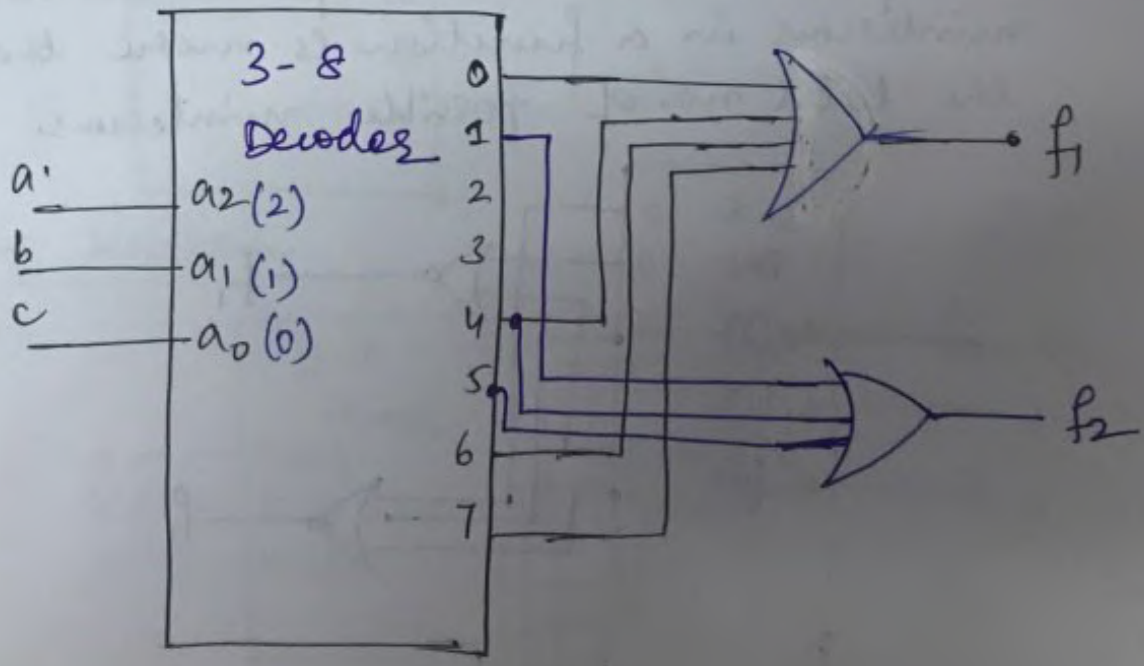
$f_2(a,b) = \Pi(0,2,3)$

Soln



2. Implement the following functions using 3-8 Decoder.

$f_1(a,b,c) = \sum m(0,4,6,7)$ $f_2(a,b,c) = \sum m(1,4,5)$



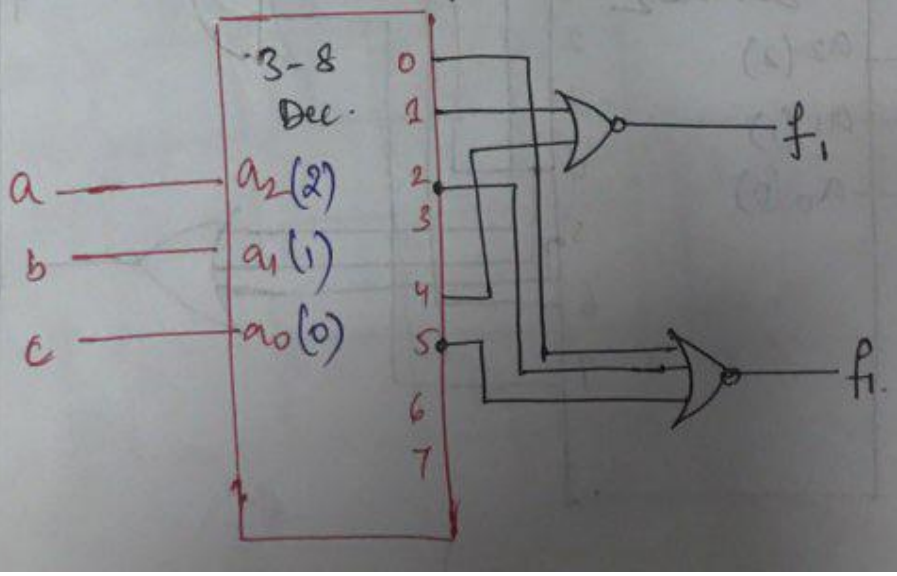
③ Implement the following functions using a decoder minimizing the no. of inputs to be summed.

$f_1(a,b,c) = \sum m(0,2,3,5,6,7) = \Pi M(1,4)$

$f_2(a,b,c) = \sum m(1,3,4,6,7) = \Pi M(0,2,5)$

Soln

- Observe that the no. of inputs to each OR gate would be the no. of minterms in the expression. We can reduce the no. of inputs by implementing $\bar{f}$ instead of f and finally invert the OR output by f.
- This can simply be done by replacing the OR gate with a NOR gate.
- This procedure can be adopted when the no. of minterms in a function is more than half the total no. of possible minterms.



M2 (12)

(A) Implement the following functions expressed in maxterm canonical form in two possible ways using 3 to 8 Decoder.

$f_1(a,b,c) = \Pi M(2,3,4,5,7)$

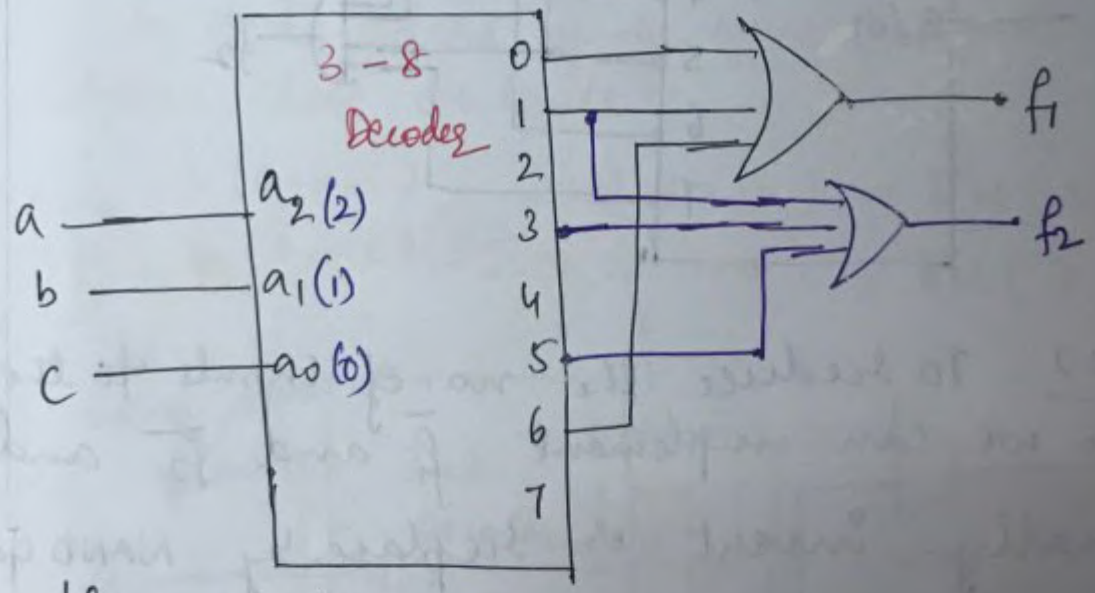
$f_2(a,b,c) = \Pi M(0,2,4,6,7)$

Soln

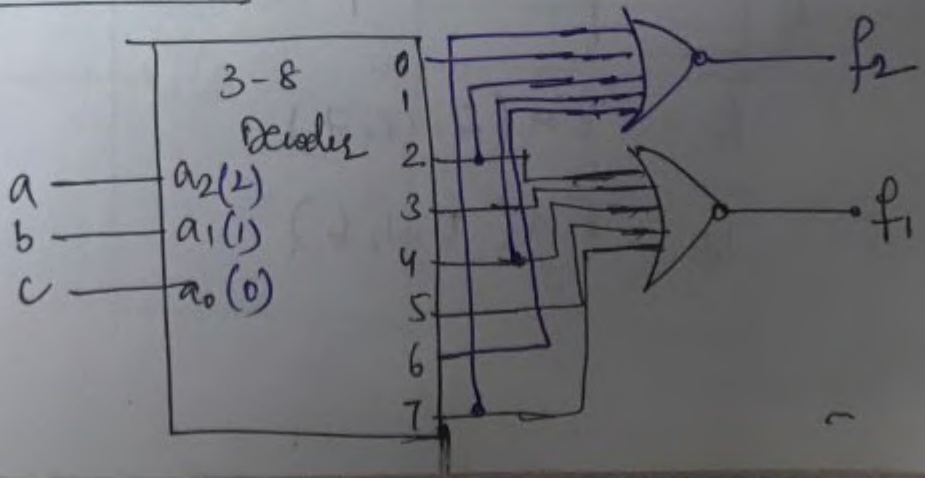
$f_1(a,b,c) = \Sigma m(0,1,6)$

$f_2(a,b,c) = \Sigma m(1,3,5)$

Using Minterm



Using Maxterm



(5) Implement the following functions using 3 to 8 Decoder with Active low outputs.

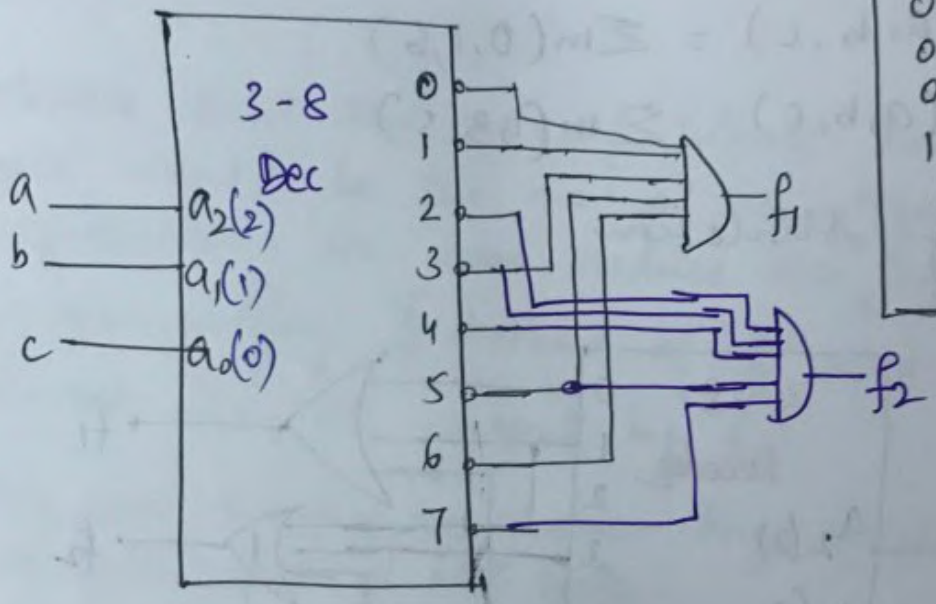
$f_1(a,b,c) = \overline{\Pi M}(0,1,3,5,6)$

$f_2(a,b,c) = \overline{\Pi M}(2,3,4,5,7)$

Soln

Case 1:

abc	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
000	0	1	1	1	1	1	1	1
001	1	0	1	1	1	1	1	1
010	1	1	0	1	1	1	1	1
011	1	1	1	0	1	1	1	1
100	1	1	1	1	0	1	1	1
101	1	1	1	1	1	0	1	1
110	1	1	1	1	1	1	0	1
111	1	1	1	1	1	1	1	0



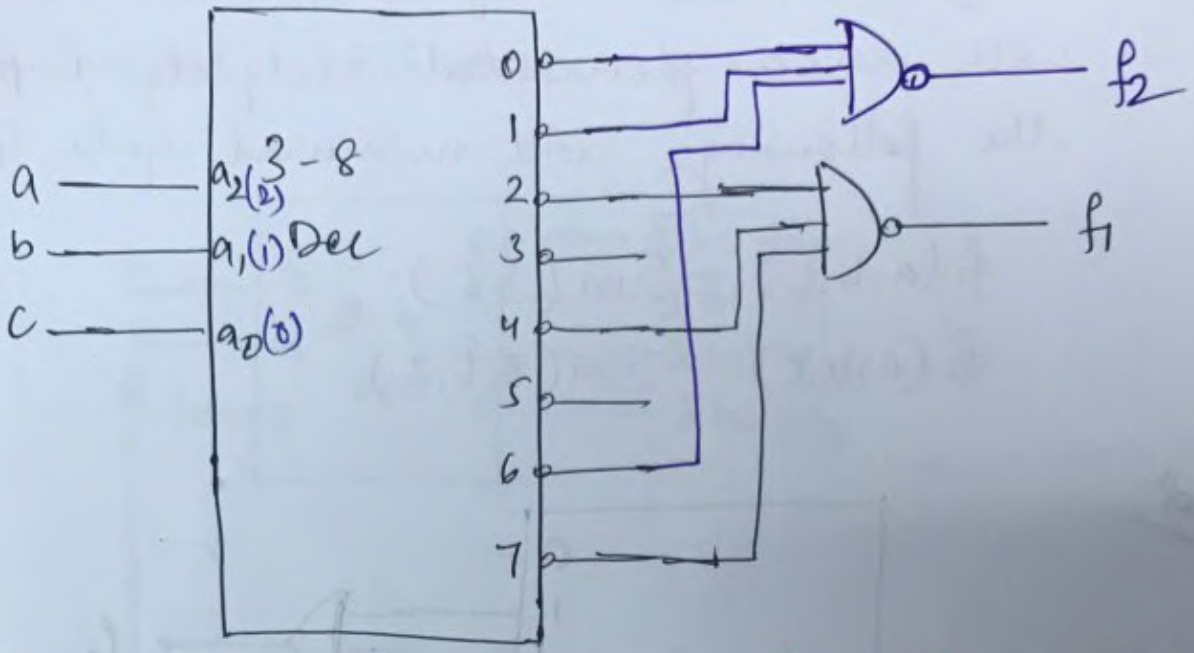
Case 2: To reduce the no. of inputs to the AND gate we can implement $\overline{f_1}$ and $\overline{f_2}$ and finally invert or replace by NAND gate in place of AND gate

$\overline{f_1} = \Pi M(2,4,7)$

$\overline{f_2} = \Pi M(0,1,6)$

M2

P3

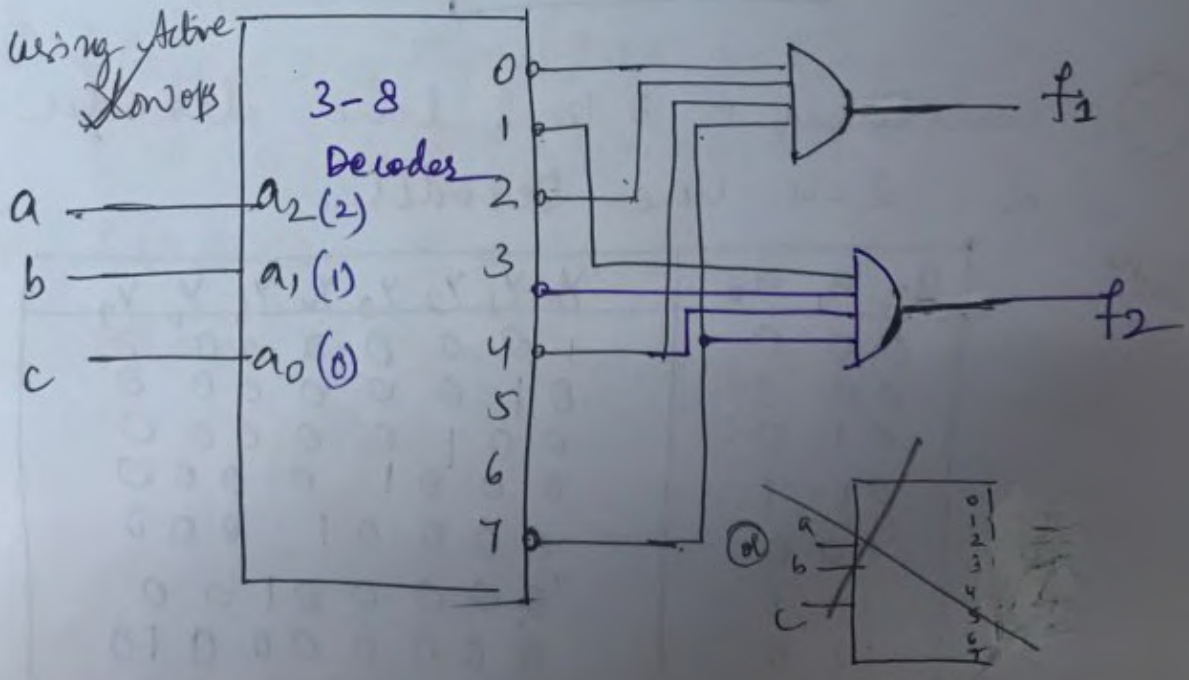


(6) Implement the following functions using 3 to 8 decoders with NAND outputs or Active low outputs.

$f_1(a,b,c) = \Sigma m(1,3,5,6) = \Pi M(0,2,4,7)$

$f_2(a,b,c) = \Sigma m(0,2,5,6) = \Pi M(1,3,4,7)$

Soln

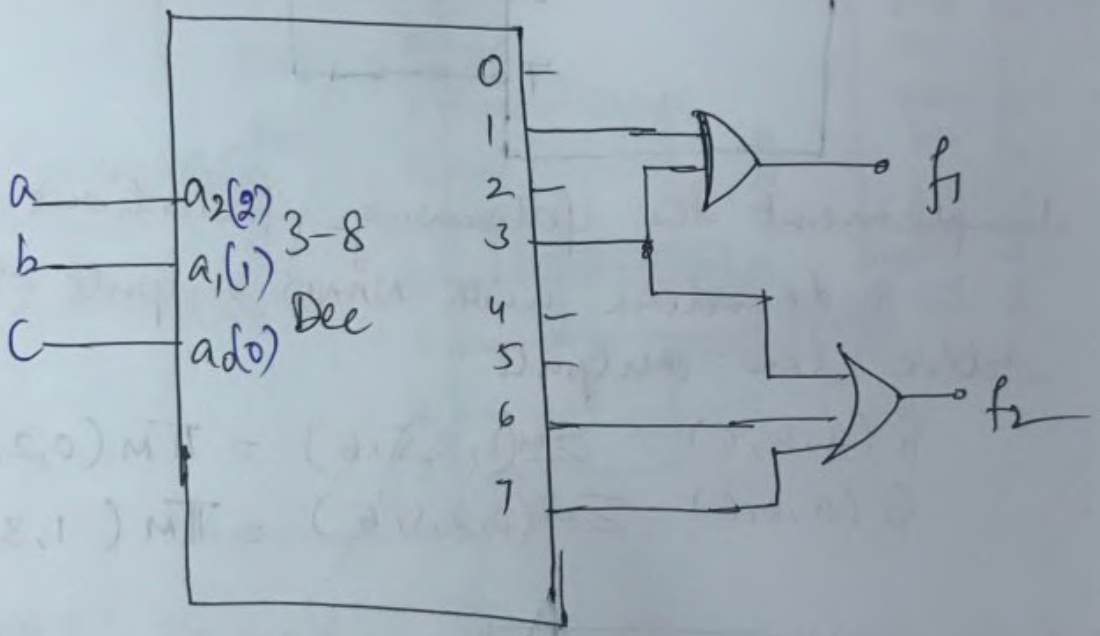


(6) Using OR gates along with a decoder with uncomplemented outputs, implement the following with minimal gate ips.

$f_1(a,b,c) = \Sigma m(1,3)$

$f_2(a,b,c) = \Sigma m(3,6,7)$

Soln



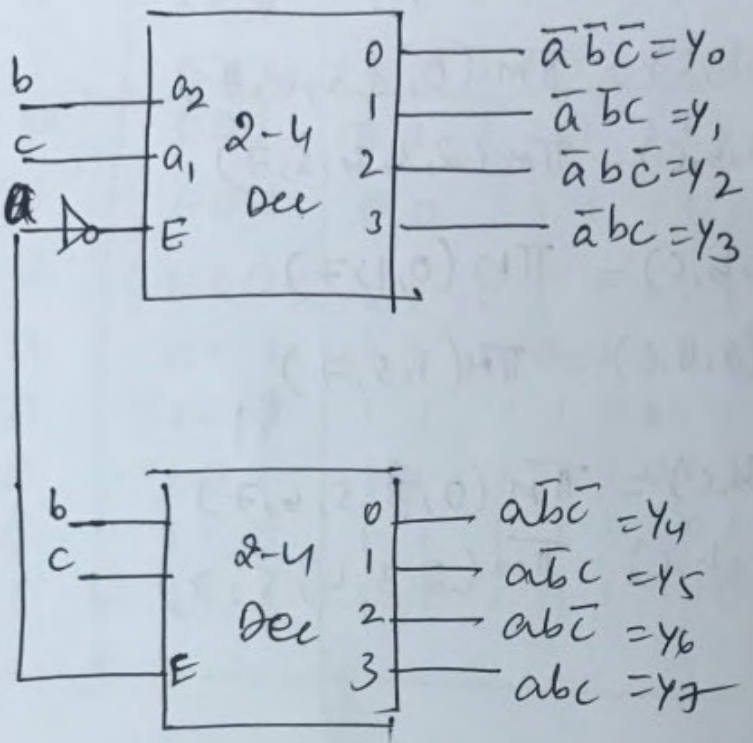
(7) Construct a 3 to 8 line decoder using 2, 2-4 line Decoder.

Soln.

a_2	a_1	a_0	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

M2 (P4)

- The MSB a is connected to enable input as shown.



(8) Implement the following with uncomplemented outputs.

a) $f_1(a,b,c) = \Sigma m(0,1,5,6,7)$

$f_2(a,b,c) = \Sigma m(1,2,3,6,7)$

b) $f_1(a,b,c) = \Sigma m(0,2,4)$

$f_2(a,b,c) = \Sigma m(1,2,4,5,7)$

(9) Using decoders with complemented ops, implement the following function pairs with minimum gate ips.

a) $f_1(a,b,c) = \overline{\Pi M}(0,3,5,6,7)$

$f_2(a,b,c) = \overline{\Pi M}(2,3,4,5,7)$

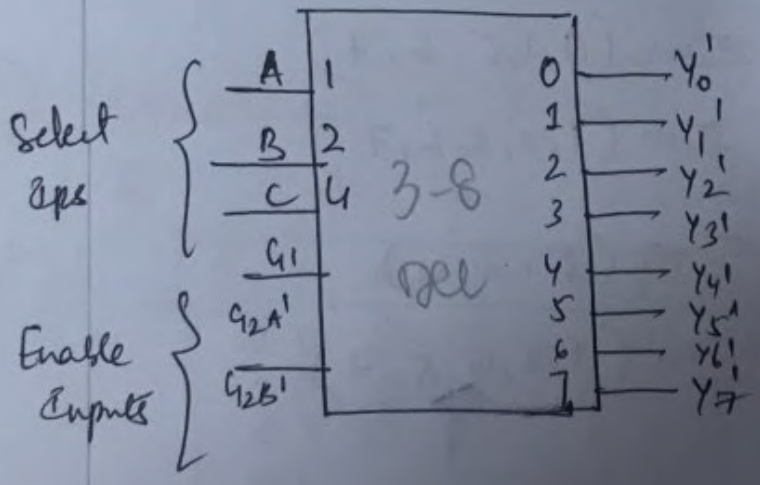
b) $f_1(a,b,c) = \overline{\Pi M}(0,1,7)$

$f_2(a,b,c) = \overline{\Pi M}(1,5,7)$

c) $f_1(a,b,c) = \overline{\Pi M}(0,3,5,6,7)$

$f_2(a,b,c) = \overline{\Pi M}(2,3,4,5,7)$

(10) 74LS138 Decoder.



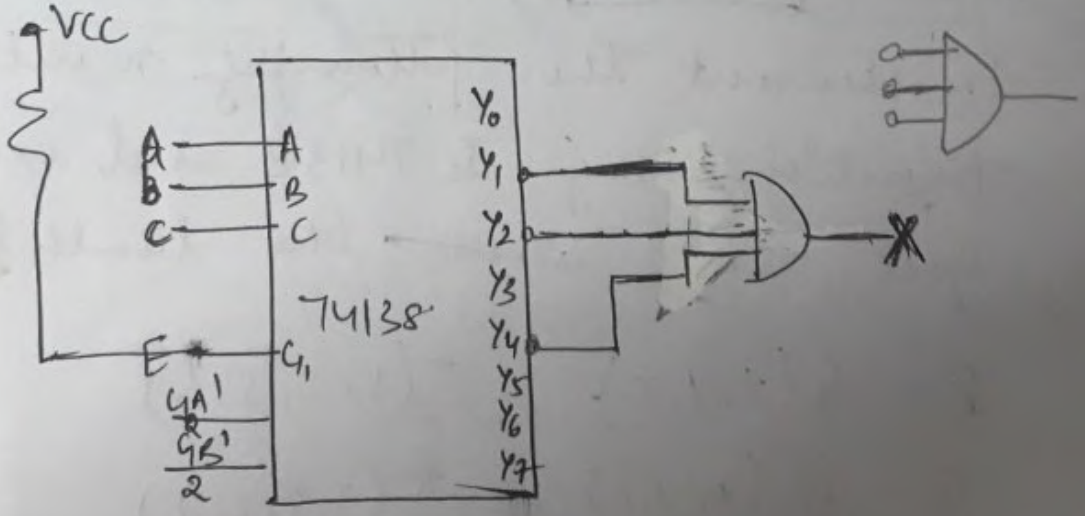
M2
P3

Truth table

G_1	G_{2A}'	G_{2B}'	C	B	A	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
0	X	X	X	X	X	1	1	1	1	1	1	1	1
X	1	X	X	X	X	1	1	1	1	1	1	1	1
X	X	1	X	X	X	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	1	1	1	1	1	1	1
1	0	0	0	0	1	1	0	1	1	1	1	1	1
1	0	0	0	1	0	1	1	0	1	1	1	1	1
1	0	0	0	1	1	1	1	1	0	1	1	1	1
1	0	0	1	0	0	1	1	1	1	0	1	1	1
1	0	0	1	0	1	1	1	1	1	1	0	1	1
1	0	0	1	1	0	1	1	1	1	1	1	0	1
1	0	0	1	1	1	1	1	1	1	1	1	1	0

Ex: $f(abc) = \sum m\,0,3,5,6,7$ $\quad \bar{f}(a,b,c) = \overline{\sum m\,(1,2,4)}$

~~$\prod M\,(1,2,4)$~~ $= \prod M(1,2,4)$



(1) Implement the following function using 74138 decoder -

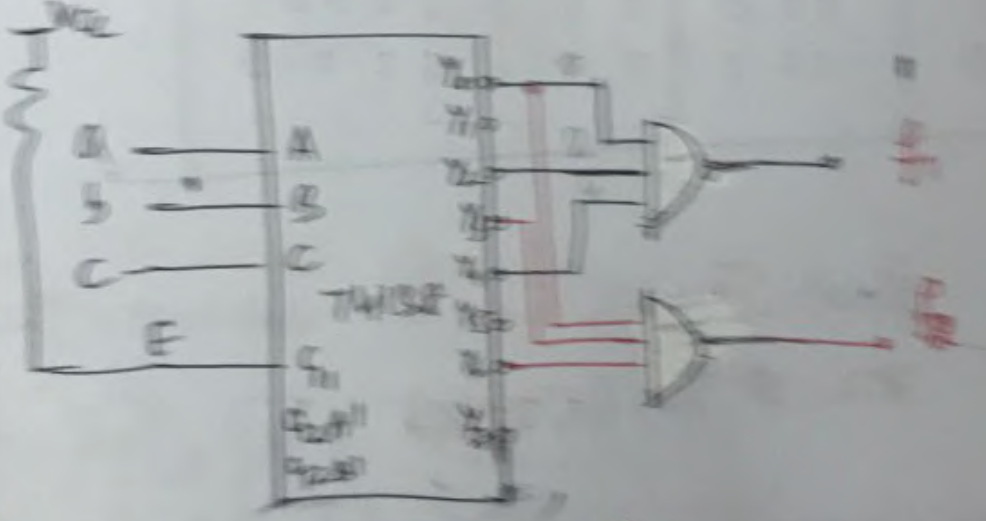
$F_1(x,y,z) = \sum(0,2,4)$

$F_2(x,y,z) = \sum(1,2,4,5,7)$

Case 1 :- Direct -

Case 2 :- $F_1 \rightarrow$ direct

$F_2 \rightarrow \overline{F_2} = \prod(0,3,6)$



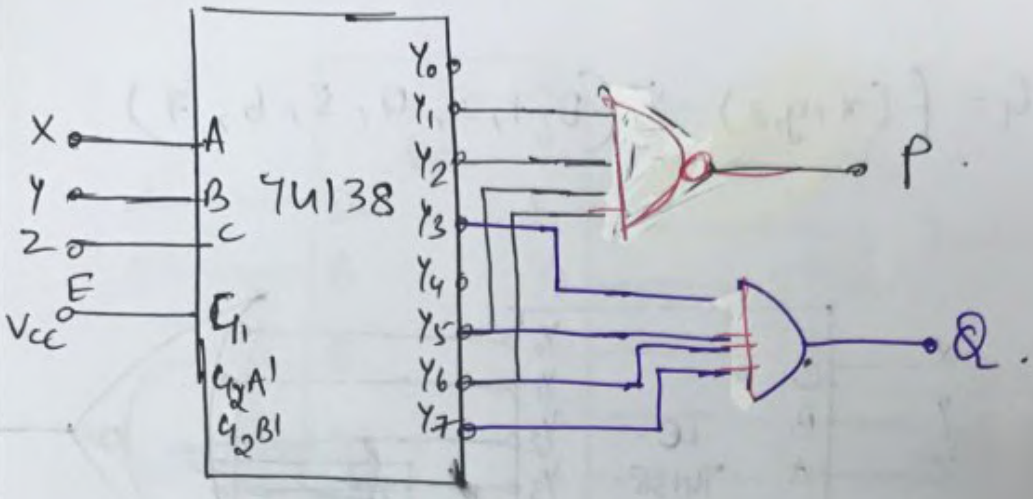
(2) Implement the following multiple o/p function using 74138 and external gate. Also write the truth table -

$P = F_1(x,y,z) = \Sigma(1,2,5,6)$

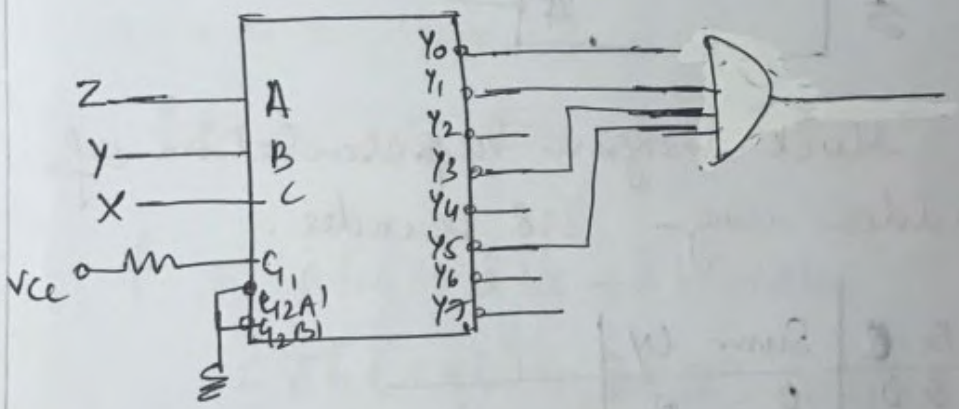
$Q = F_2(x,y,z) = \pi(3,5,6,7)$

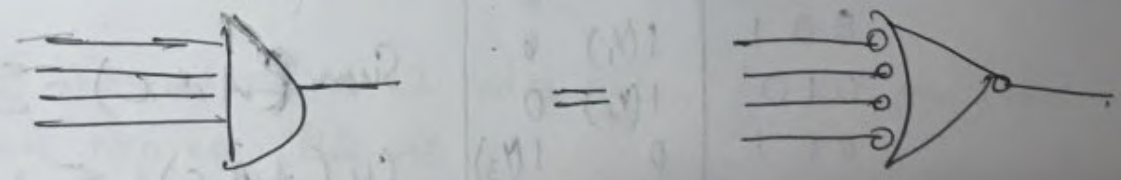
M2

(P6)



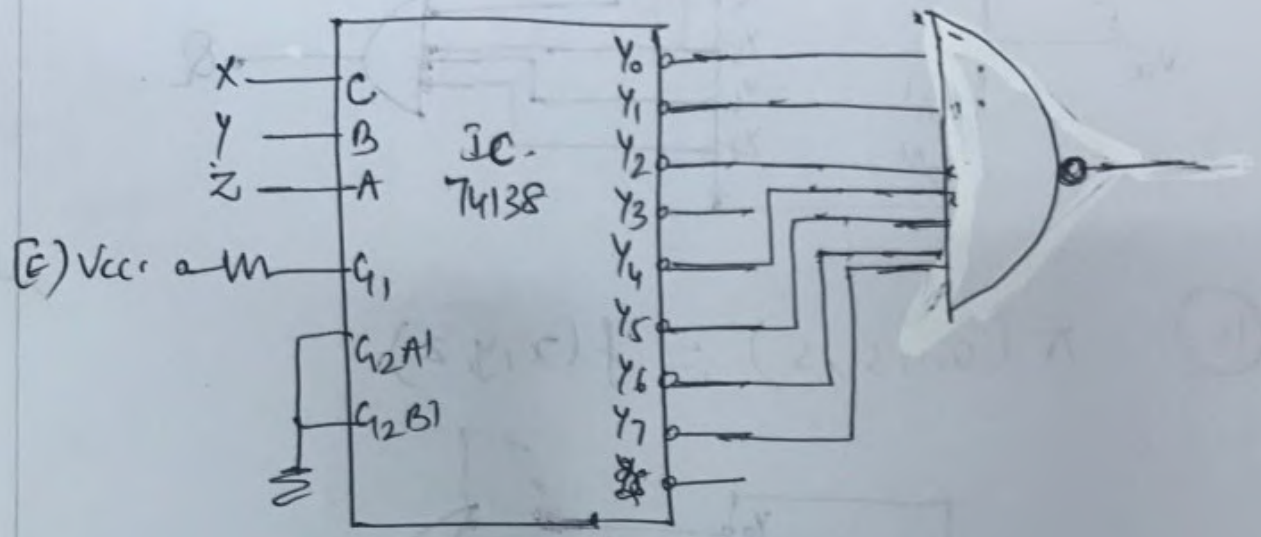
(13) $\pi(0,1,3,5) = f(x,y,z)$





(14) $G = f(x,y,z) = \pi(3)$

(15) $G = f(x,y,z) = \sum(0,1,2,4,5,6,7)$



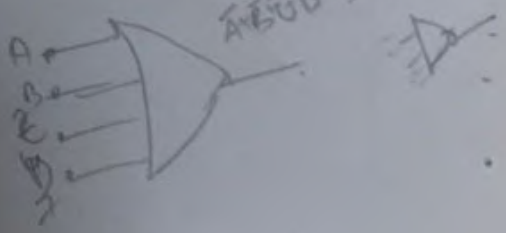
(16) write block diagram representation of fulladder using 3:8 Decoder.

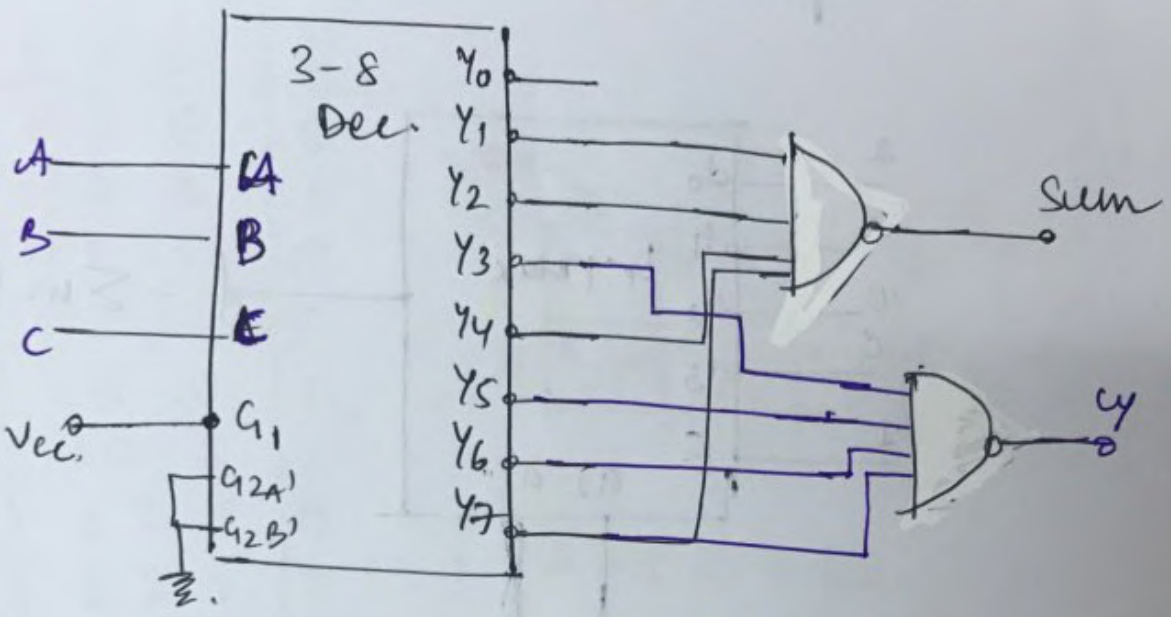
Soln

A	B	C	Sum	Cy
0	0	0	0	0
0	0	1	1 (Y_1)	0
0	1	0	1 (Y_2)	0
0	1	1	0	1 (Y_3)
1	0	0	1 (Y_4)	0
1	0	1	0	1 (Y_5)
1	1	0	0	1 (Y_6)
1	1	1	1 (Y_7)	1 (Y_7)

$Sum(A,B,C) = \sum 1,2,4,7$

$Cy(A,B,C) = \sum 3,5,6,7$





(17) Implement the following function using a 4 to 1 mux.

$$f(a,b,c) = \sum m\,(0,1,2,7)$$

abc
0 - 000
1 - 001
2 - 010
7 - 111

Soln

$$f = \bar{a}\bar{b}\bar{c} + \bar{a}\bar{b}c + \bar{a}b\bar{c} + abc.$$

$$= \bar{a}\bar{b}(c+\bar{c}) + \bar{a}b(\bar{c}) + ab(c)$$

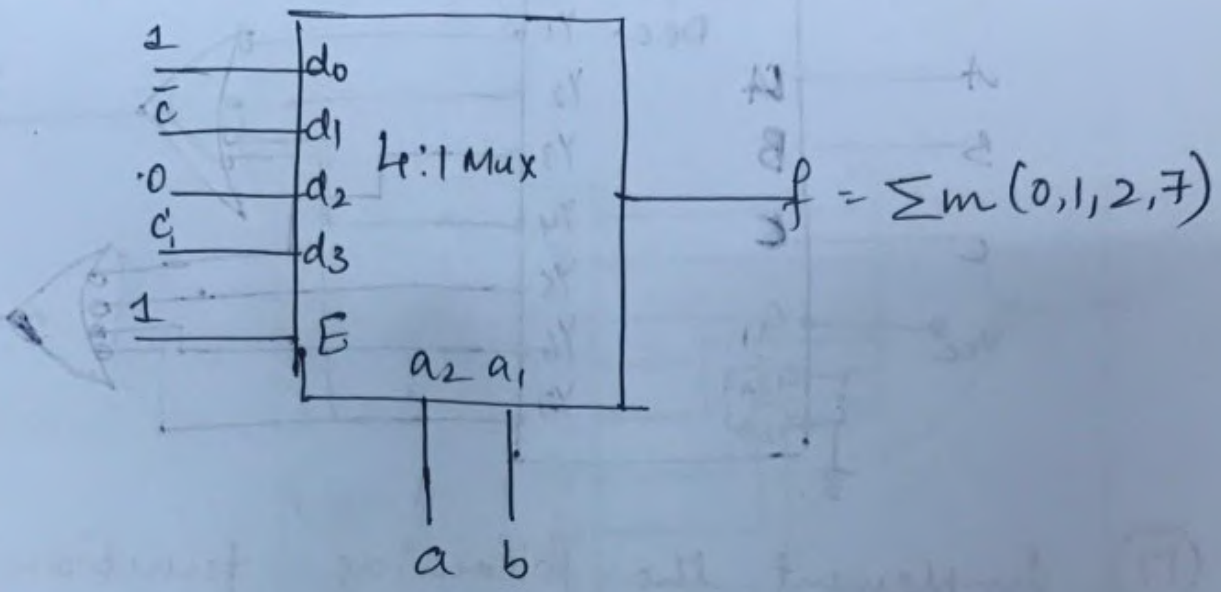
- In terms of a 2 variable function, we can express this as

$$f = \bar{a}\bar{b}(1) + \bar{a}b(\bar{c}) + a\bar{b}(0) + ab(c)$$

$$= \bar{a}\bar{b}(d_0) + \bar{a}b(d_1) + a\bar{b}(d_2) + ab(d_3)$$

- Assuming enable is at logic 1, ab as address lines / select lines.

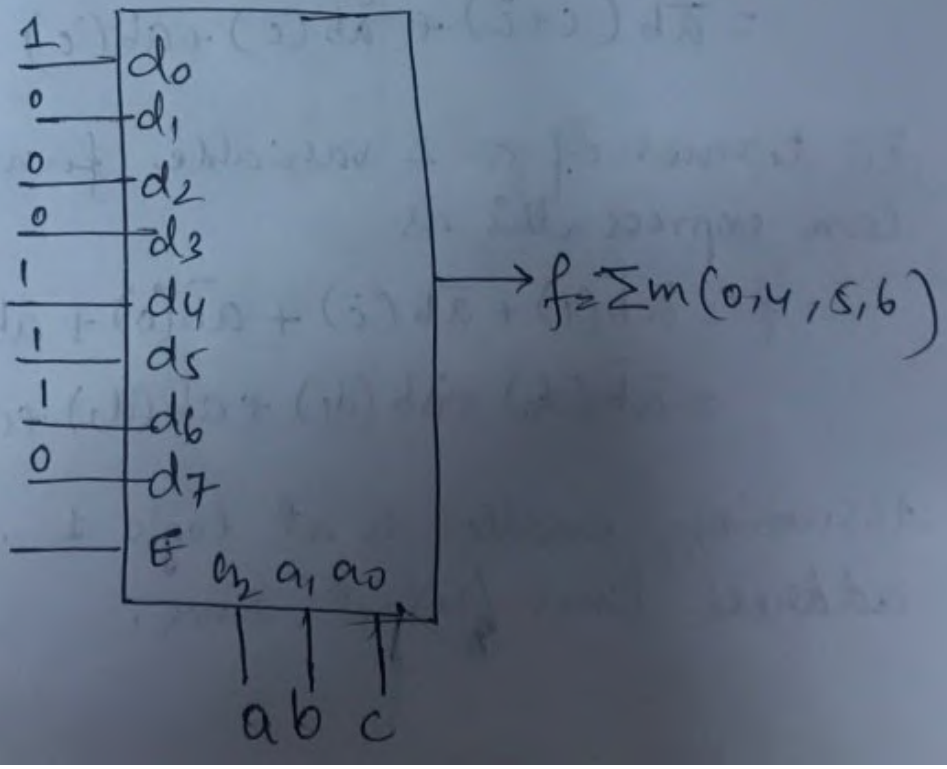
The implementation is shown below



(18) Implement the following function using 8:1 Mux.

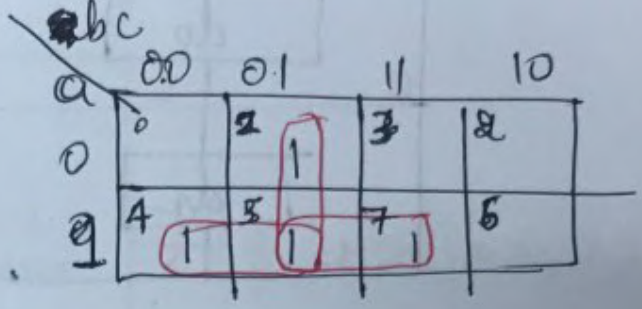
$$f(a,b,c) = \Sigma m(0,4,5,6)$$

Soln



(19) $f = \sum m(1,4,5,7)$

	a	b	c	f
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1



$f = \bar{a}b + \bar{a}c + bc$

4:1 Mux

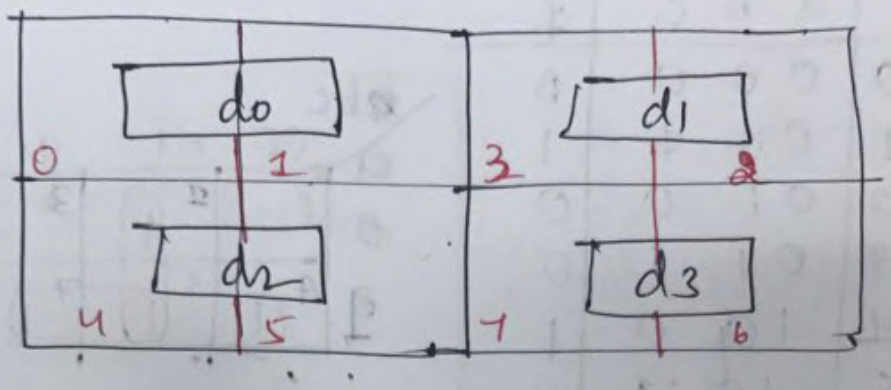
$$f = \bar{a}\bar{b}(d_0) + \bar{a}b(d_1) + a\bar{b}(d_2) + ab(d_3)$$

Let a be assigned to Select line a_1
b to Select line a_0 in 4:1 Mux.

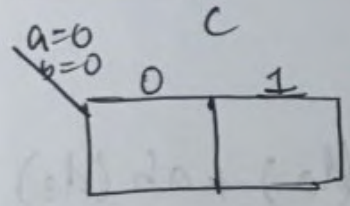
Hence d_0 relates to cells with $a=0, b=0$
d_1 $a=0\ b=1$
d_2 $a=1\ b=0$
d_3 $a=1\ b=1$

- Cells 0 to 1 corresponds to $a=0\ b=0$.
  Cell 0 corresponds to $c=0$
  Cell 1 corresponds to $c=1$.
- On these lines, the K-Maps for the Sub-function

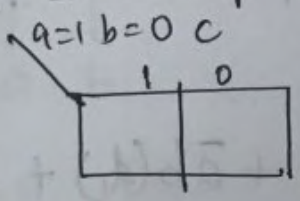
d_0, d_1, d_2, d_3 is as shown.



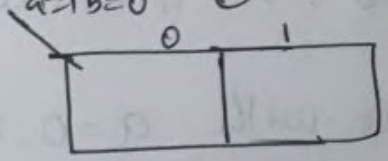
d_0 map



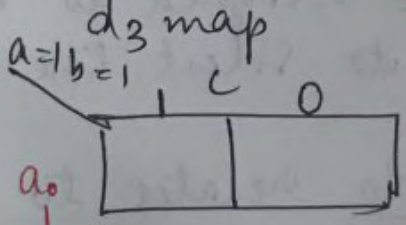
d_1 map

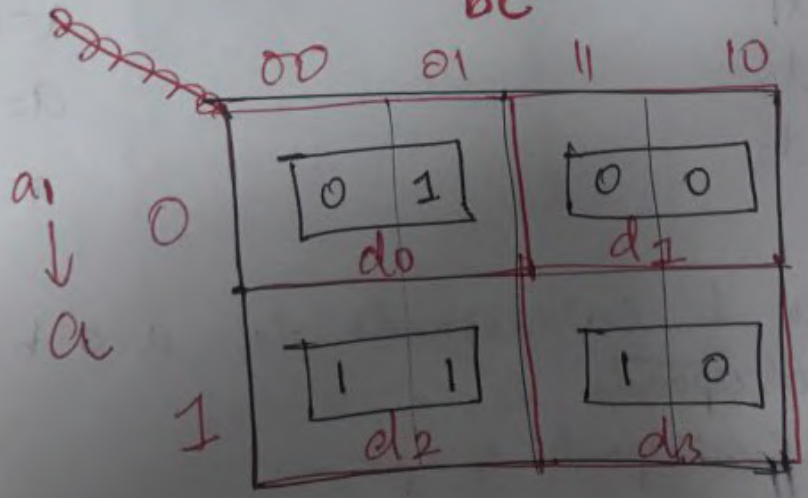


d_2 map



d_3 map

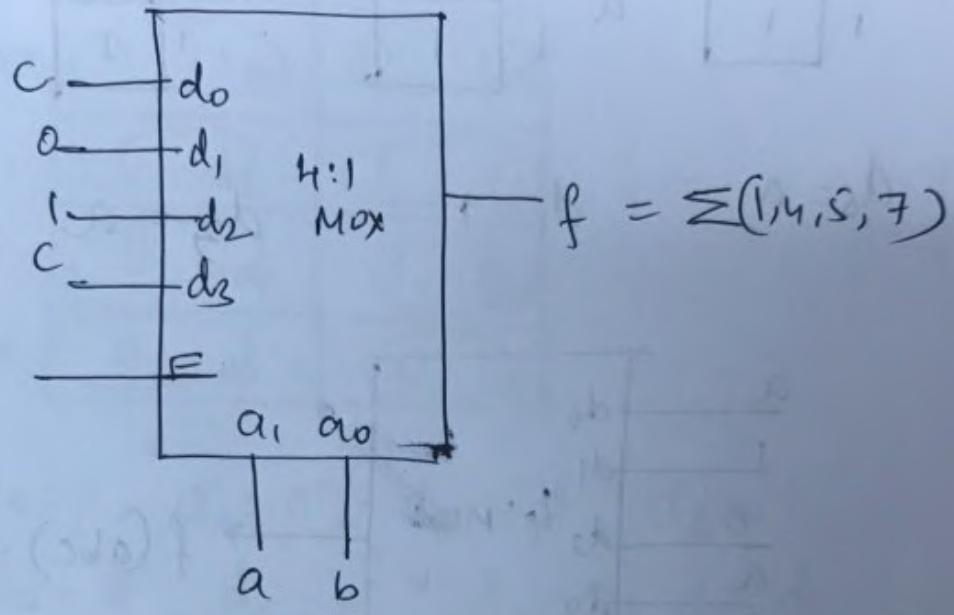




M2

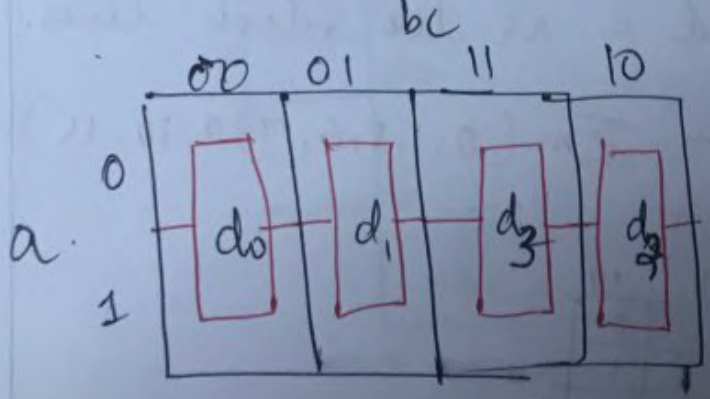
$d_0 = c$, $d_1 = 0$, $d_2 = 1$, $d_3 = c$.

Implementation of 4:1 Mux.



(2a) $f(a,b,c) = \Sigma m(1,4,5,7)$ using a 4:1 Mux using b and c as address lines.

a \ bc	00	01	11	10
0	0	1	0	0
1	1	1	1	0



d_0-map

b=1 c=0

a	0	1
0	0	
1	1	

d_1 map

b=0 c=1

a	0	1
0	1	
1	1	

d_2 map

b=1 c=0

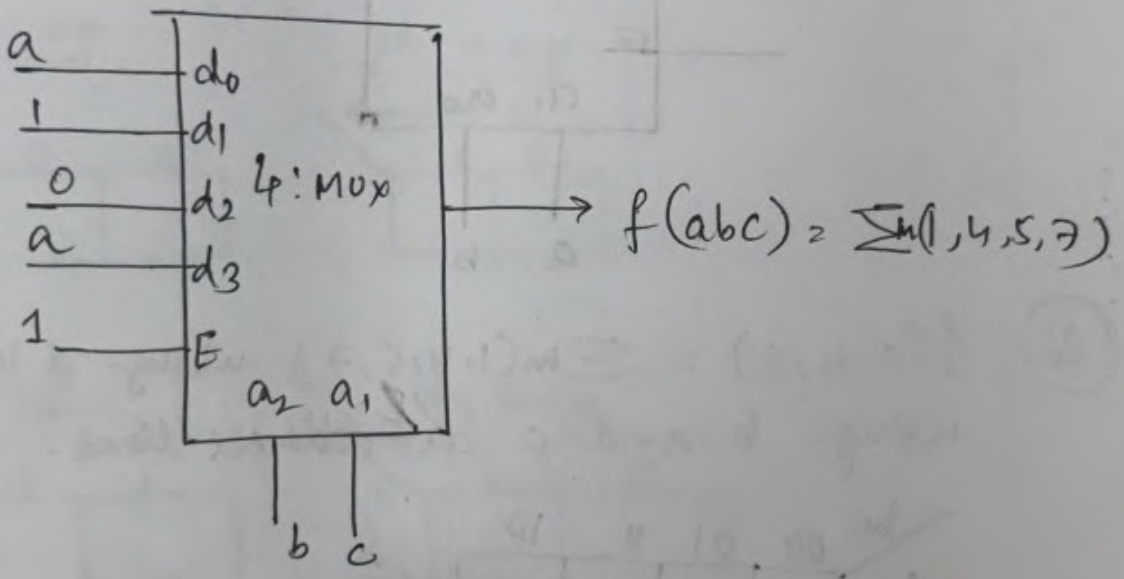
a	0	1
0	0	
1	1	

d_3 map

b=1 c=0

a	0	1
0	0	
1	0	

$d_0 = a$. $d_1 = 1$ $d_2 = a$ $d_3 = 0$.



(21) Implement the following function using 8:1 Mux.

Treat a, b and c as the select lines.

$f(a,b,c,d) = \sum m(0,1,5,6,7,9,10,15)$

Soln

ab \ cd	00	01	11	10
00	1	1	0	0
01	0	1	0	1
11	0	0	1	0
10	0	1	0	1

M2-
P10

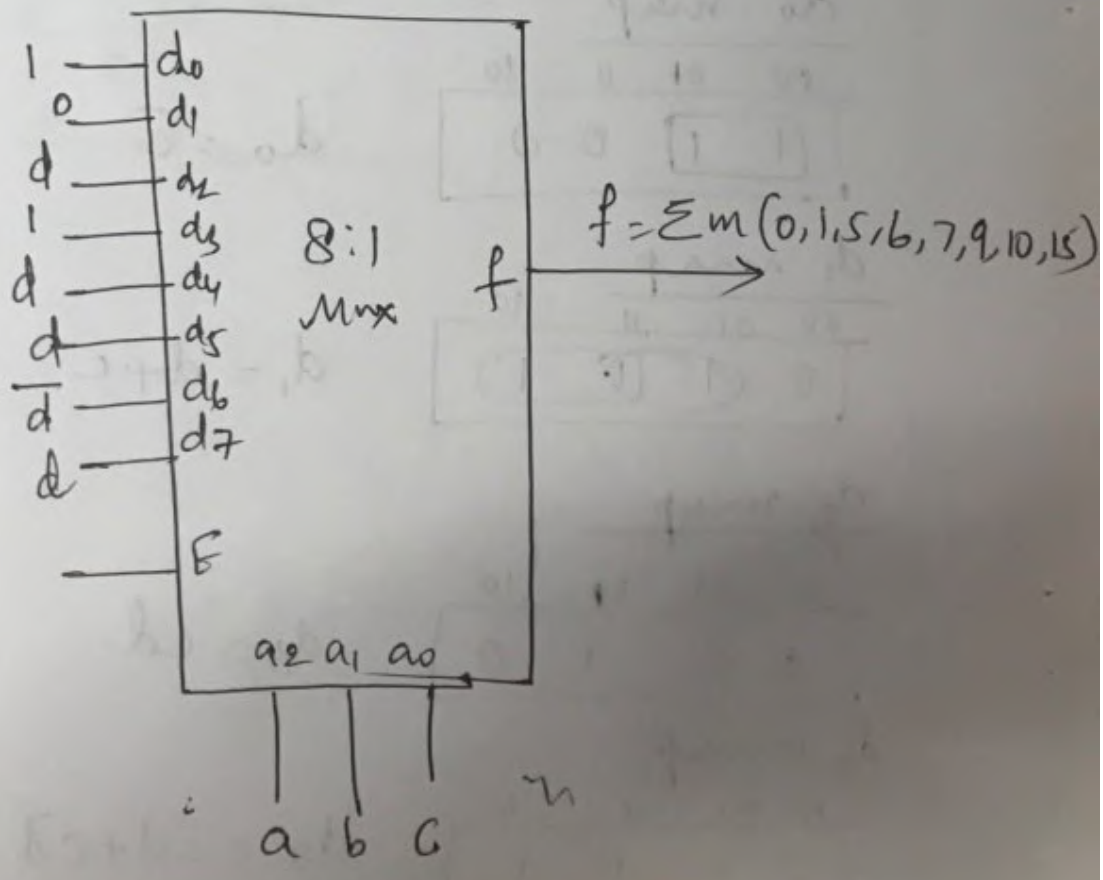
Sol^n

The 4-variable map with sub-function is as shown (a_0 ↓ c d)

a_2a_1 ↓↓ a b	00	01	11	10
00	1 (d_0)	1	0 (d_4)	0
01	0 (d_2)	1	1 (d_3)	1
11	0 (d_6)	0	1 (d_7)	0
10	0 (d_4)	1	0 (d_5)	1

$d_0 = 1,\ d_1 = 0,\ d_2 = d,\ d_3 = 1$

$d_4 = d,\ d_5 = \bar{d},\ d_6 = 0,\ d_7 = d$



(22) Implement the previous problem using 4:1 Mux with a & b as select lines.

Soln

ab \ cd	00	01	11	10	
00	1	1	0	0	d_0
01	0	1	1	1	d_1
11	0	0	1	0	d_3
10	0	1	0	1	d_2

($a_2 a_0$ → a b)

Subfunction maps are as follows.

d_0 map

00	01	11	10
1	1	0	0

$d_0 = \bar{c}$

d_1 map

00	01	11	10
0	1	1	1

$d_1 = d + c$

d_3 map

00	01	11	10
0	0	1	0

$d_3 = cd$

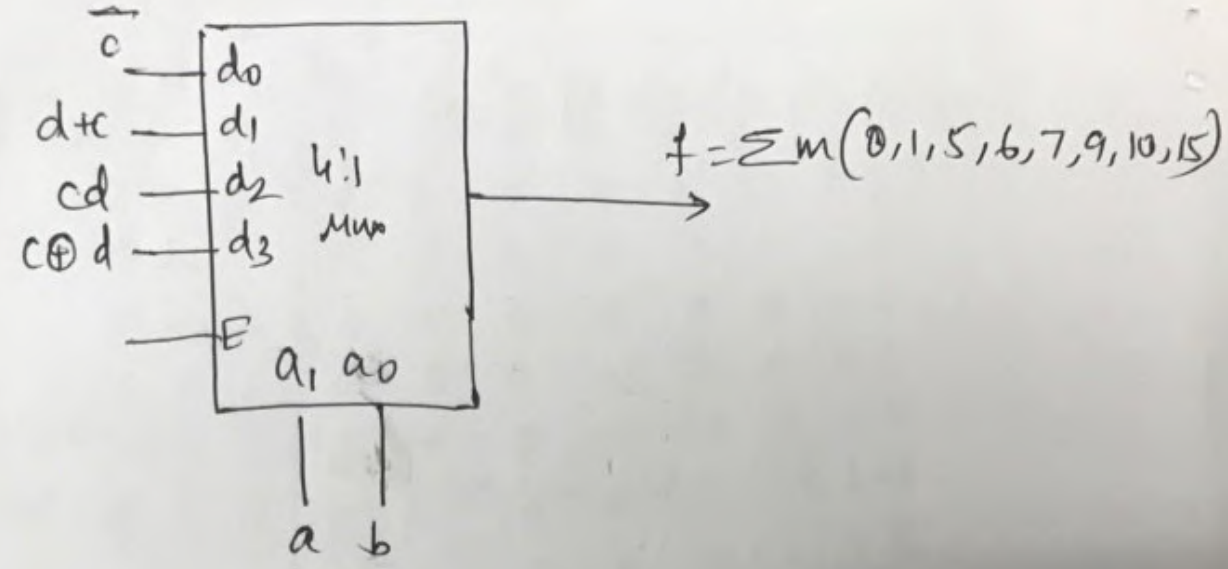
d_2 - map

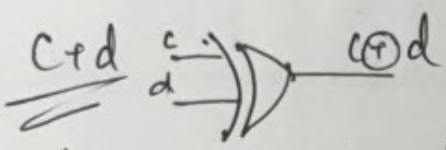
00	01	11	10
0	1	0	1

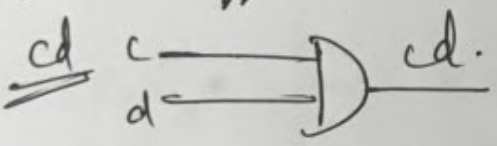
$d_2 = \bar{c}d + c\bar{d} = c \oplus d$

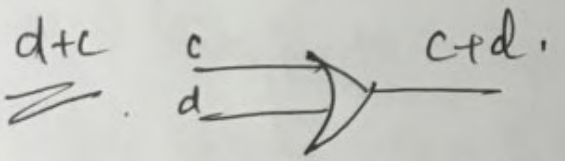
M2

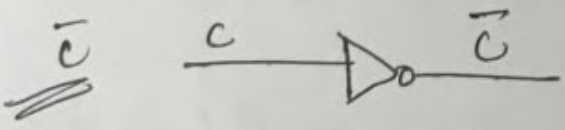
P11











23

3 to 8 Decoder

x_2 x_1 x_0	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7	
0 0 0	1	0	0	0	0	0	0	0	$\bar{x}_2\bar{x}_1\bar{x}_0$
0 0 1	0	1	0	0	0	0	0	0	$\bar{x}_2\bar{x}_1 x_0$
0 1 0	0	0	1	0	0	0	0	0	$\bar{x}_2 x_1 \bar{x}_0$
0 1 1	0	0	0	1	0	0	0	0	$\bar{x}_2 x_1 x_0$
1 0 0	0	0	0	0	1	0	0	0	$x_2\bar{x}_1\bar{x}_0$
1 0 1	0	0	0	0	0	1	0	0	$x_2\bar{x}_1 x_0$
1 1 0	0	0	0	0	0	0	1	0	$x_2 x_1 \bar{x}_0$
1 1 1	0	0	0	0	0	0	0	1	$x_2 x_1 x_0$

$Y_0 = \bar{x}_2\bar{x}_1\bar{x}_0 = m_0$ $Y_1 = \bar{x}_2\bar{x}_1 x_0$ $Y_2 = \bar{x}_2 x_1 \bar{x}_0$

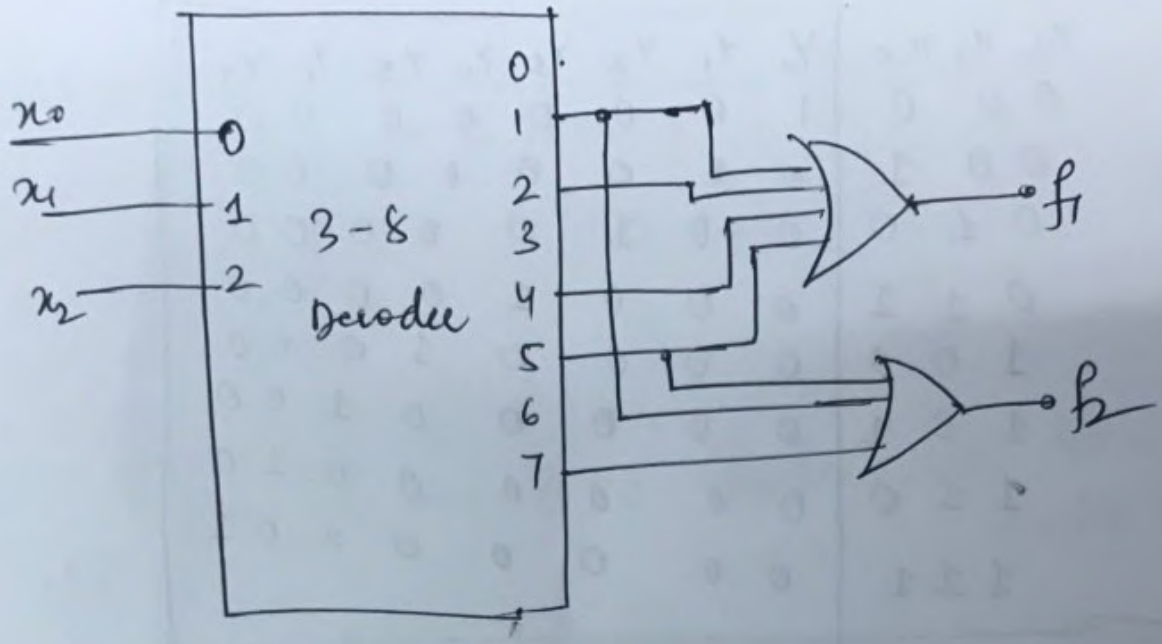
$Y_3 = \bar{x}_2 x_1 x_0$, $Y_4 = x_2\bar{x}_1\bar{x}_0$ $Y_5 = x_2\bar{x}_1 x_0$

$Y_6 = x_2 x_1 \bar{x}_0$ $Y_7 = x_2 x_1 x_0$.

- generally n to 2^n decoder is a minterm generator.
- By using OR gates in conjunction with an n to 2^n decoder, realizations of Boolean functions are possible.

Ex: $f_1(x_2 x_1 x_0) = \sum m(1,2,4,5)$

$f_2(x_2, x_1 x_0) = \sum m(1,5,7)$



— complement of a minterm canonical formula is the sum of those minterms not appearing in the original formula.

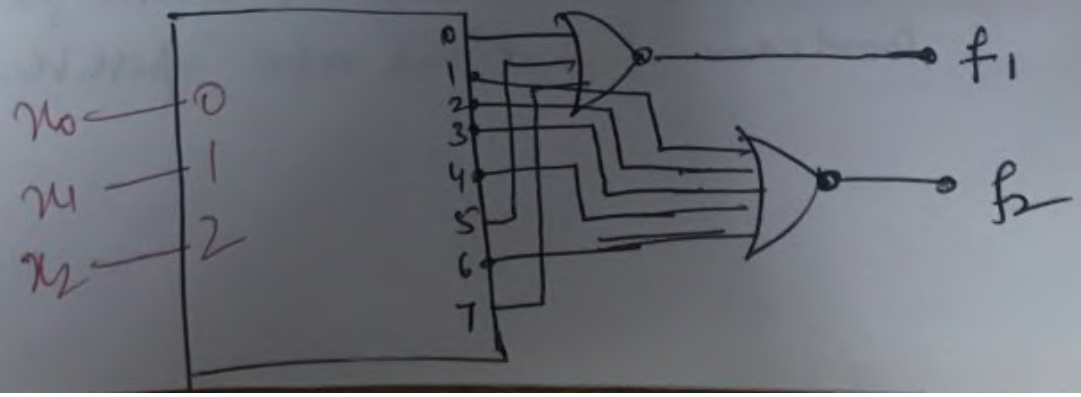
ex: if ① $f_1(x_2 x_1 x_0) = \sum 0,1,3,4,5,6) = \overline{\sum(2,7)}$

then ~~$f_1(x_2 x_1 x_0) = 0,3,6,7$~~

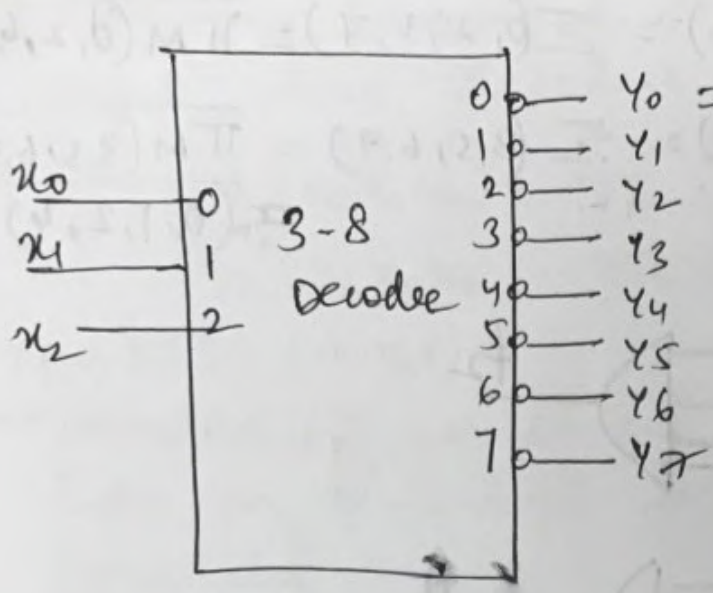
$\overline{f_1}(x_2 x_1 x_0) = \sum 2,7$

② $f_2(x_2 x_1 x_0) = \sum m(1,2,3,4,6) = \overline{\sum m(0,5,7)}$

$\overline{f_2}(x_2 x_1 x_0) = \sum m(0,5,7)$



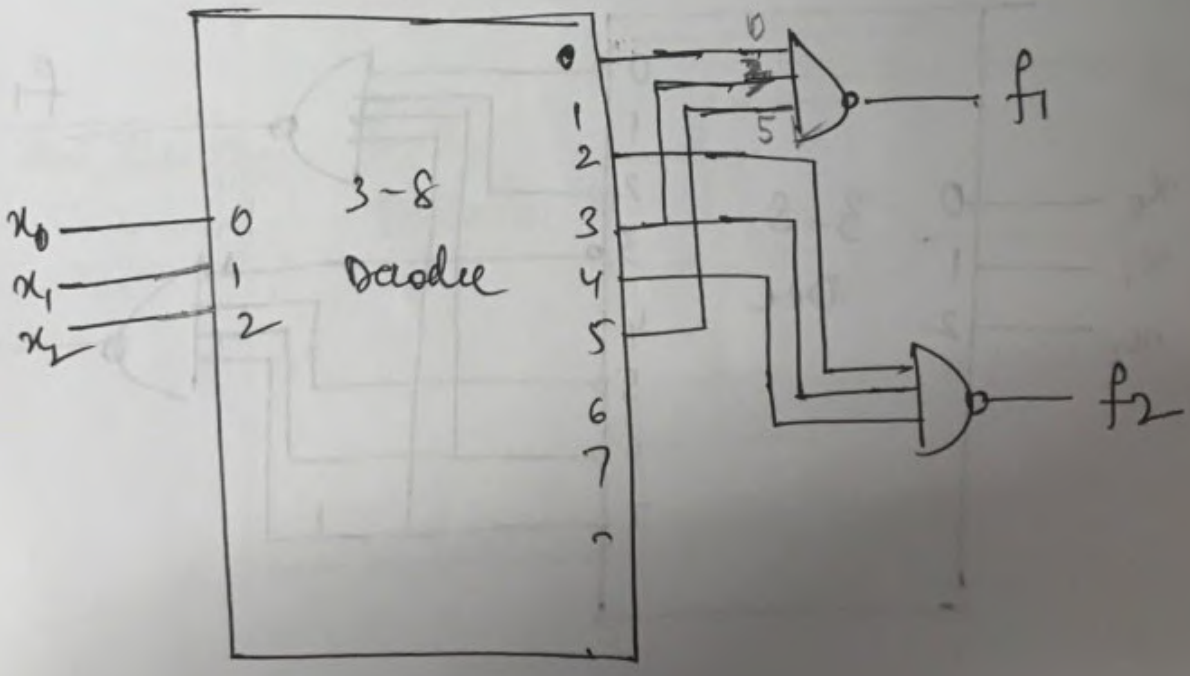
P13



Ex:

$f_1(x_2 x_1 x_0) = \Pi M(0,3,5) =$

$f_2(x_2 x_1 x_0) = \Pi M(2,3,4)$



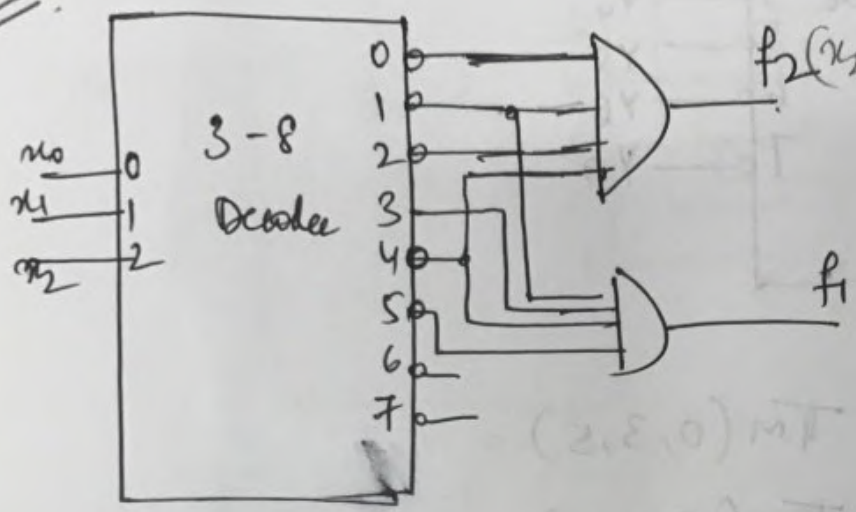
$= \overline{\Sigma m(1,3,4,5)}$

Ex2 : $f_1(x_2 x_1 x_0) = \Sigma(0,2,6,7) = \overline{\Pi M(0,2,6,7)}$

$f_2(x_2 x_1 x_0) = \Sigma(3,5,6,7) = \overline{\Pi M(3,5,6,7)}$

$\overline{\Sigma m(0,1,2,4)}$

AND



$f_2(x_2 x_1 x_0) = \Sigma(0,2,6,7)$

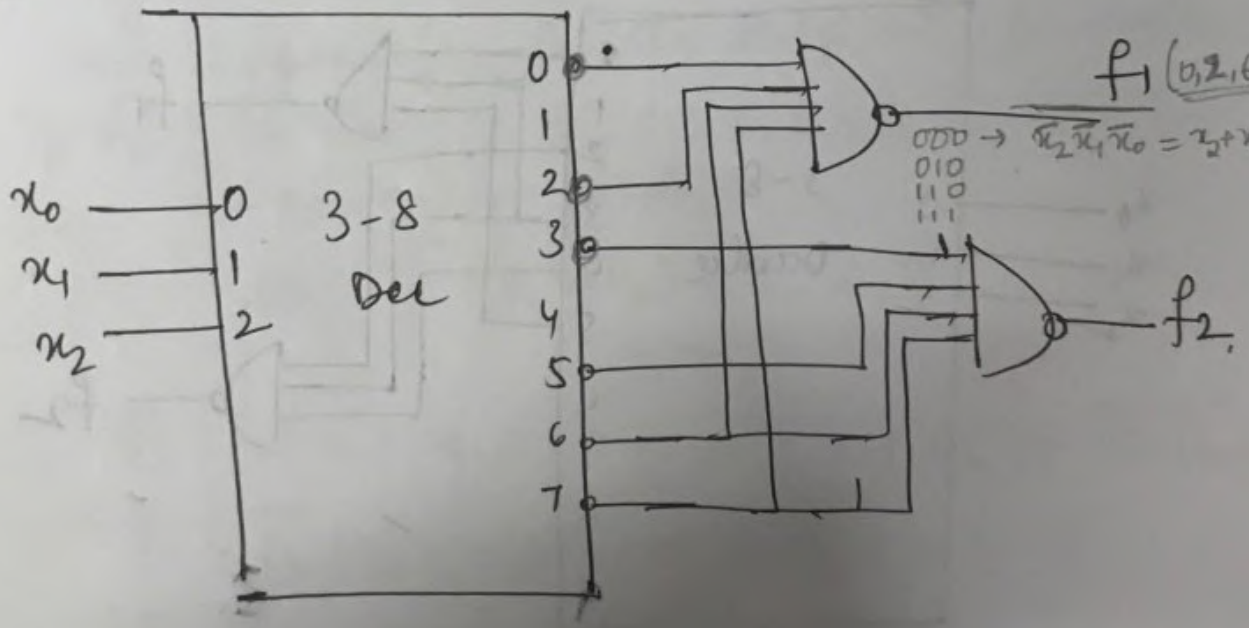
000 → $\bar{x}_2 \bar{x}_1 \bar{x}_0 = \overline{x_2 + x_1 + x_0}$

010

110

111

NAND



$\overline{A}$

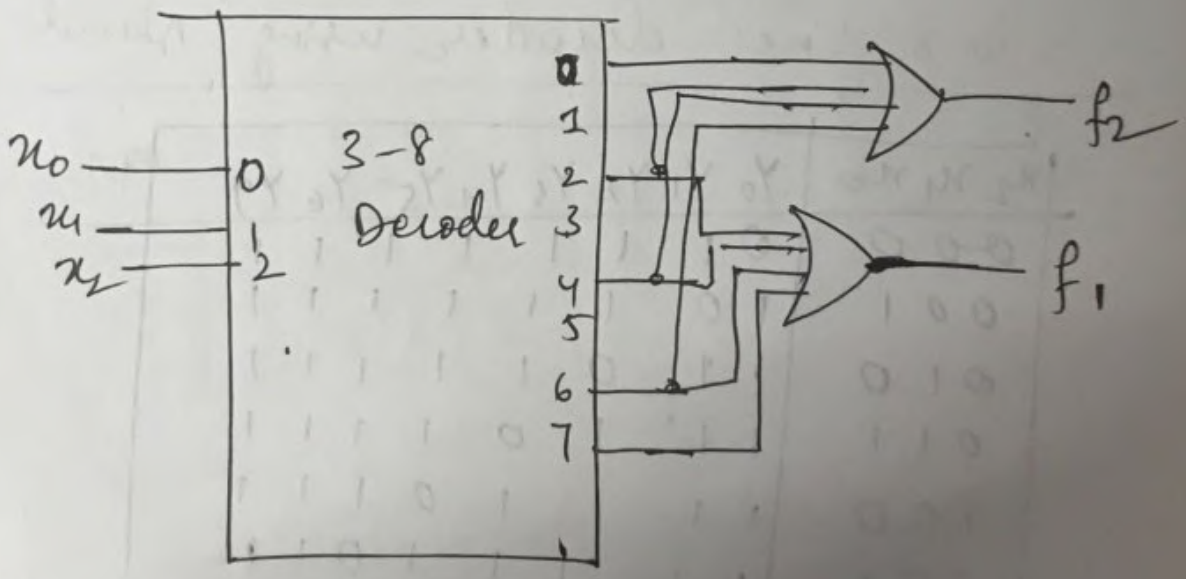
Maxterm canonical formulas using n to 2^n line decoders.

$$f_1(x_2x_1x_0) = \overline{\Pi M}\,(0,1,3,5)$$
$$f_2(x_2x_1x_0) = \overline{\Pi M}\,(1,3,6,7)$$

— Maxterm canonical can be converted into an equivalent Minterm canonical formula.

For ex $f_1(x_2x_1x_0) = \overline{\Pi M}(0,1,3,5) = \Sigma(2,4,6,7)$

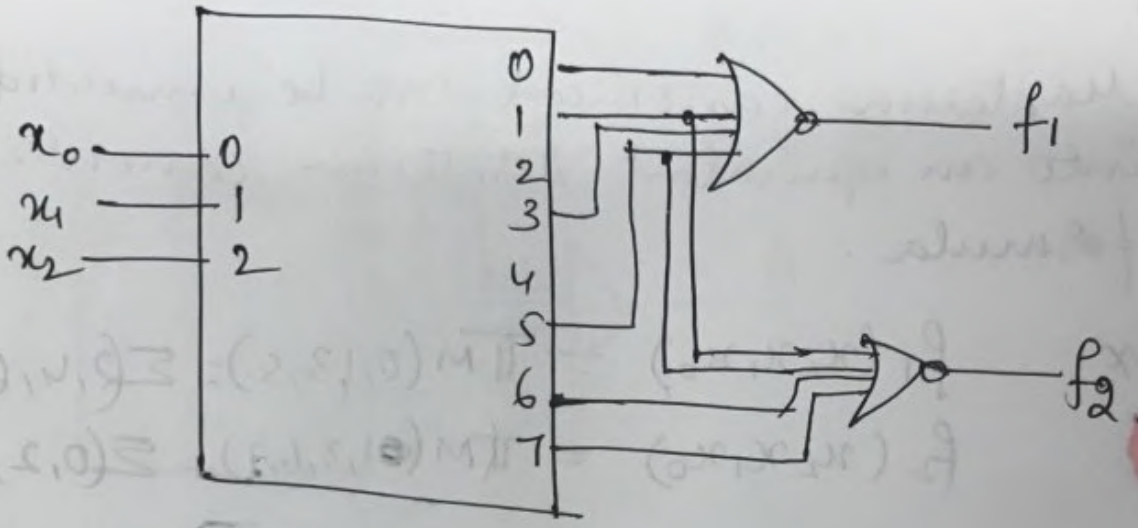
$f_2(x_2x_1x_0) = \Pi M(1,3,6,7) = \Sigma(0,2,4,5)$



- The expression can also be written as

$f_1(x_2x_1x_0) = \Pi M(0,1,3,5) = \overline{\Sigma m(0,1,3,5)}$

$f_2(x_2x_1x_0) = \Pi M(1,3,6,7) = \overline{\Sigma m(1,3,6,7)}$



3 to 8 line decoder using Nand gates.

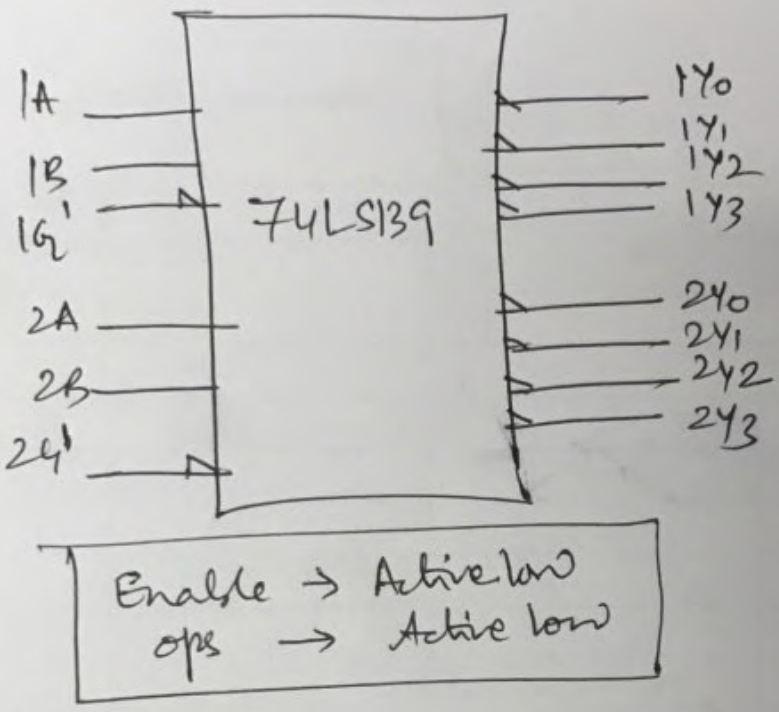
x_2 x_1 x_0	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7	Maxterm
0 0 0	0	1	1	1	1	1	1	1	$x_2+x_1+x_0$
0 0 1	1	0	1	1	1	1	1	1	$x_2+x_1+\bar{x}_0$
0 1 0	1	1	0	1	1	1	1	1	$x_2+\bar{x}_1+x_0$
0 1 1	1	1	1	0	1	1	1	1	$x_2+\bar{x}_1+\bar{x}_0$
1 0 0	1	1	1	1	0	1	1	1	$\bar{x}_2+x_1+x_0$
1 0 1	1	1	1	1	1	0	1	1	$\bar{x}_2+x_1+\bar{x}_0$
1 1 0	1	1	1	1	1	1	0	1	$\bar{x}_2+\bar{x}_1+x_0$
1 1 1	1	1	1	1	1	1	1	0	$\bar{x}_2+\bar{x}_1+\bar{x}_0$

$\overline{\bar{x}_2+x_1+x_0} = \overline{x_2 \cdot \bar{x}_1 \bar{x}_0} = x_2\bar{x}_1\bar{x}_0$

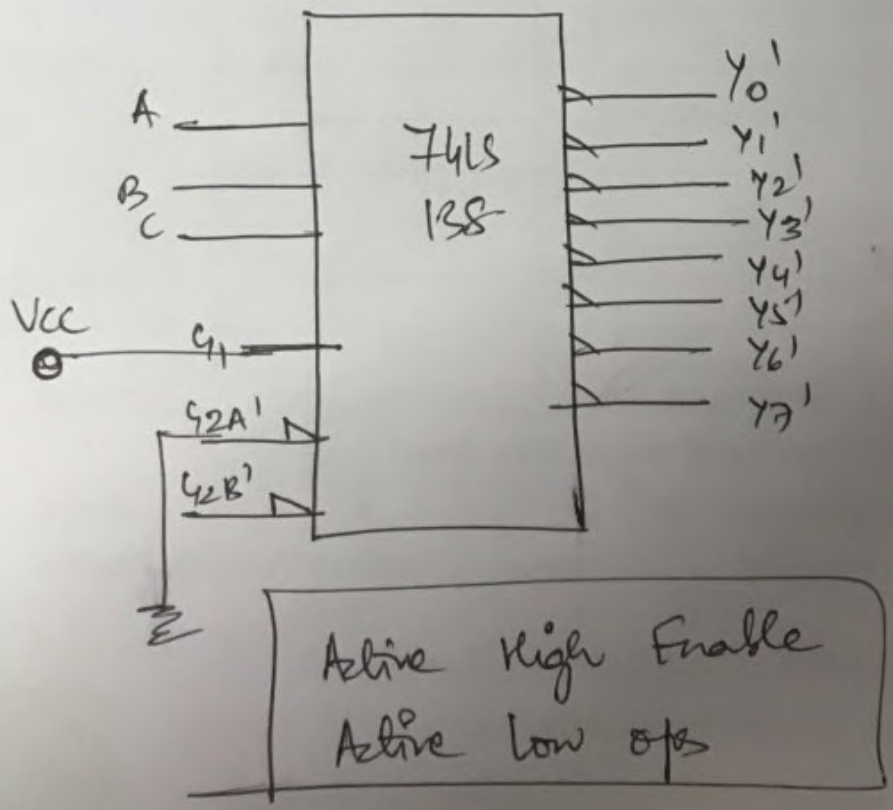
$\overline{0} \qquad 0$

Decoders

① 2:4 decoder — 74LS139

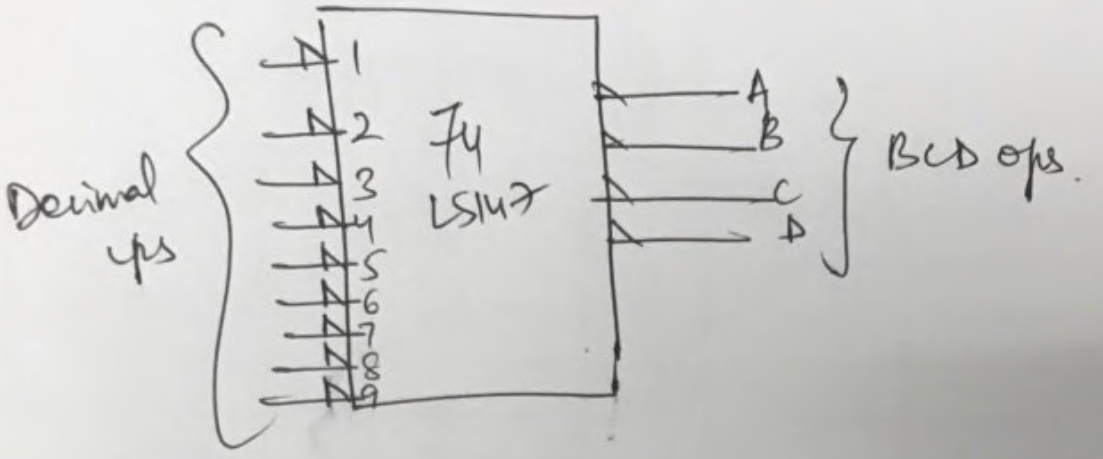


② 3 to 8 decoder → 74LS138



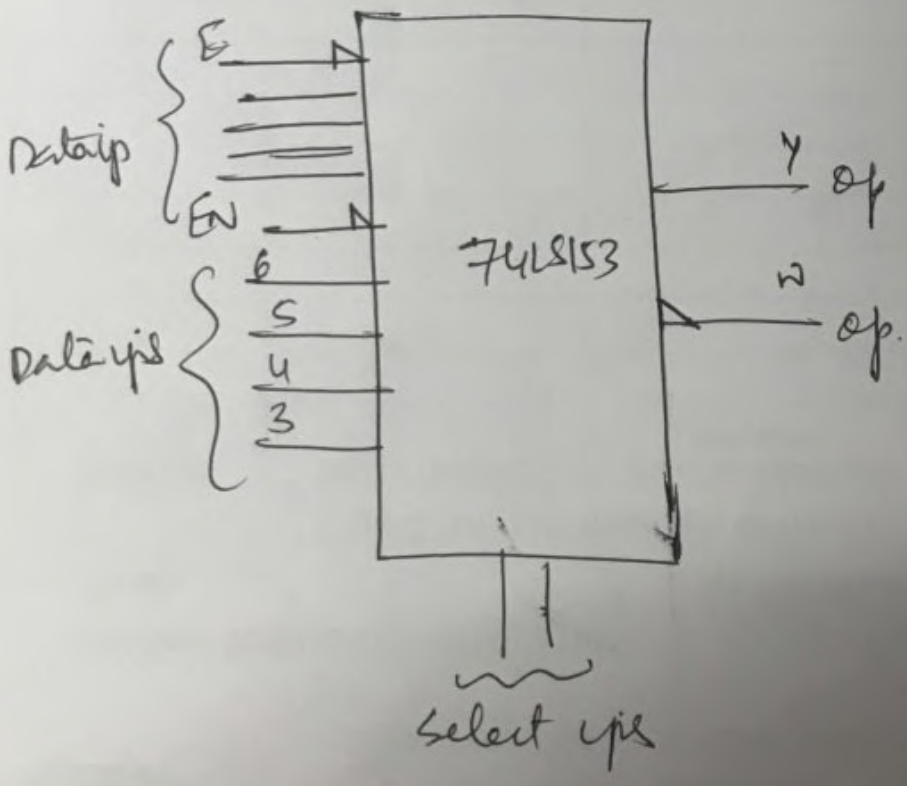
Encoders

(1) · 9 to 4 BCD Encodee (74LS147



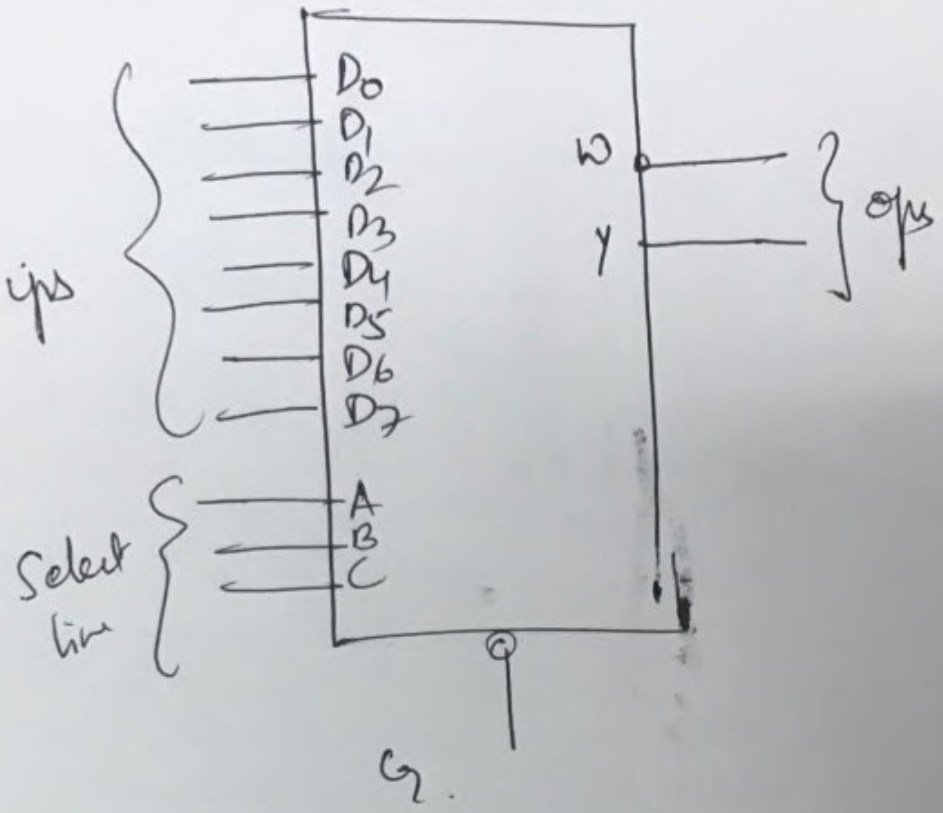
Multiplexers

(1) 4:1 (Dual) Multiplexer 74LS153



Active low Enable.

② 8:1 Multiplexer – 74LS151



Active low Enable.

PROGRAMMABLE LOGIC DEVICES. (PLDS)

- A programmable logic device is a general name for a digital integrated circuit capable of being programmed to provide a variety of different logic functions.
- Simple Combinational PLDs are capable of realizing from 2 to 10 functions of 4 to 16 variables with a single integrated circuit.
- More complex PLDs may contain thousands of gates and flipflops.
- Thus a single PLD can replace a large no. of integrated circuits and thus leads to lower cost design.
- When a digital system is designed using a PLD changes in the design can easily be made by changing the programming of the PLD without having to change the wiring in the system.

Programmable Logic Arrays (PLAs)

- A programmable logic array performs the same functions as ROM.
- A PLA with n inputs and m outputs can realize m functions of n variables.
- The internal organization of the PLA consists of (1) an AND array which realizes selected product terms of the input variables. (2) The OR array ORs together the product terms needed to perform the output functions.
- So PLA implements a Sum of products expression.

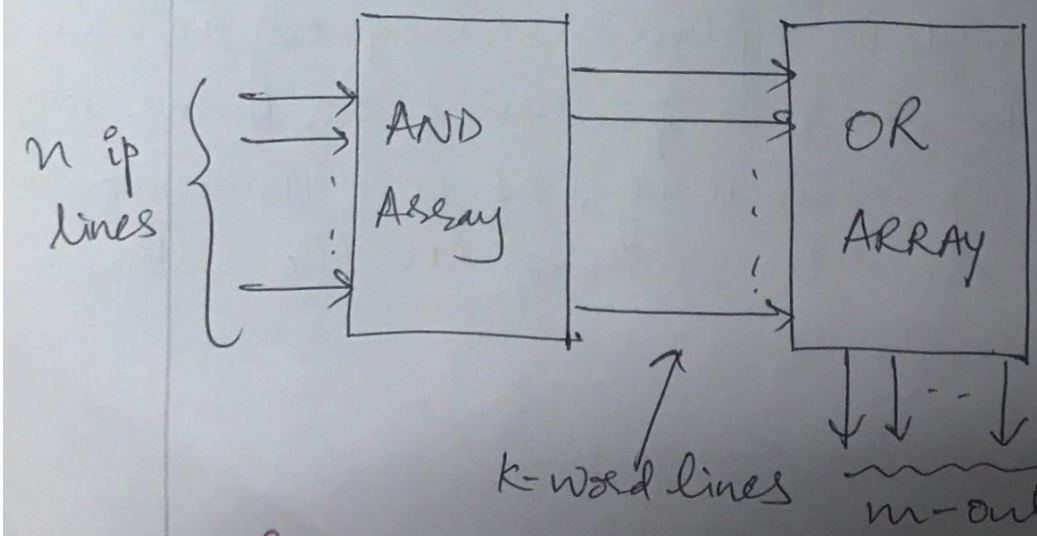


fig (1): Programmable Logic Array Structure

Fig (2) Shows a PLA which realizes the SOP expression. Product terms are formed in the AND array by connecting Switching elements at appropriate points in the array.

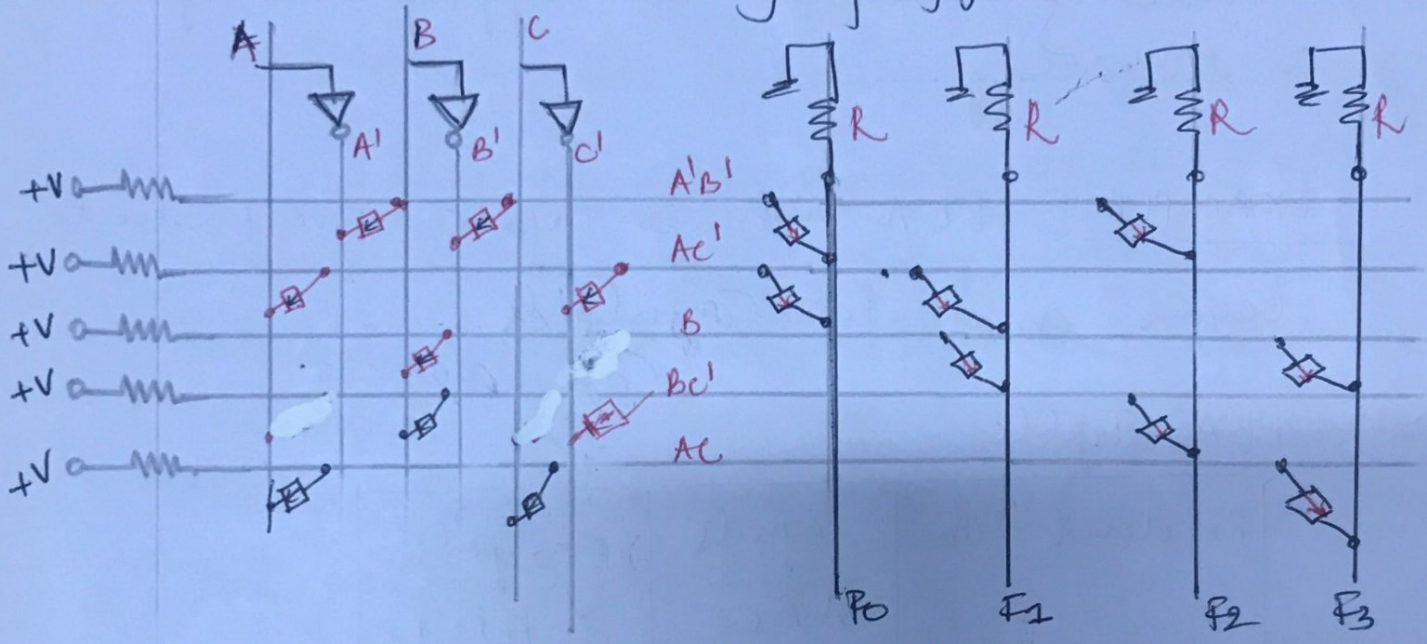
EXAMPLE: PLA with 3 inputs five product terms and four outputs.

PLA Table

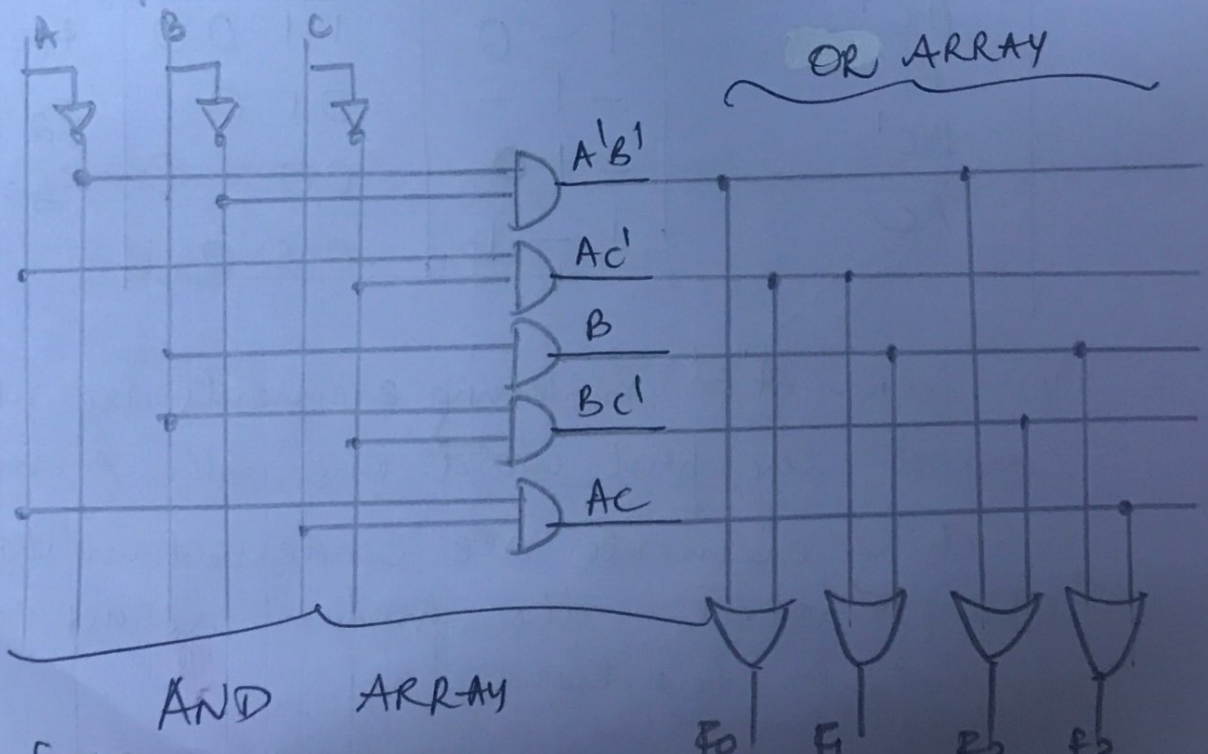
Product term	Inputs ABC	outputs F_0 F_1 F_2 F_3	
$A'B'$	0 0 –	1 0 1 0	$F_0 = A'B' + AC'$
AC'	1 – 0	1 1 0 0	$F_1 = AC' + B$
B	– 1 –	0 1 0 1	$F_2 = A'B' + BC'$
BC'	– 1 0	0 0 1 0	$F_3 = B + AC$.
AC	1 – 1	0 0 0 1	

- To form $A'B'$ Switching elements are used to connect the first word line with A' and B' lines. Switching elements are connected in the OR array to select the product terms needed for the output functions.

For example $F_0 = A'B' + AC'$, switching elements are used to connect the $A'B'$ and AC' lines to the F_0 line. The connections in the AND and OR arrays of this PLA make it equivalent to the AND-OR array of fig(3).



fig(2): PLA with 3 inputs, 5 product terms and 4 outputs.



fig(3): AND-OR Array Equivalent of fig(2).

PLA Table - the contents of the PLA is specified in this table.

- The input side of the table specifies the product terms. The symbols 0, 1 and - indicate whether a variable is complemented, not complemented or not present in the corresponding product term.
- The output side of the table specifies which product terms appear in each output function.
- A 1 or 0 indicates whether a given product term is present or not present in the corresponding output functions.

Thus the first row in the table indicates that the term $A'B'$ is present in output functions F_0 & F_2.

Example 2

Realizing eqn using PLA.

$f_1 = \Sigma m(2,3,5,7,8,9,10,11,13,15)$

$f_2 = \Sigma m(2,3,5,6,7,10,11,14,15)$

$f_3 = \Sigma m(6,7,8,9,13,14,15)$

$f_1 = bd + b'c + ab'$

$f_2 = c + a'bd$

$f_3 = bc + ab'c' + abd$

Sol^n PLA Table (Table 2)

	abcd	f_1 f_2 f_3
$a'bd$	01–1	110
abd	11–1	101
$ab'c'$	100–	101
$b'c$	–01–	100
c	––1–	010
bc	–11–	001

- fig (4) shows the corresponding PLA Structure which has 4 inputs, 6 product terms, and 3 outputs.
- A dot at the intersection of a word line and an input or output line indicates the presence of a switching element in the array.

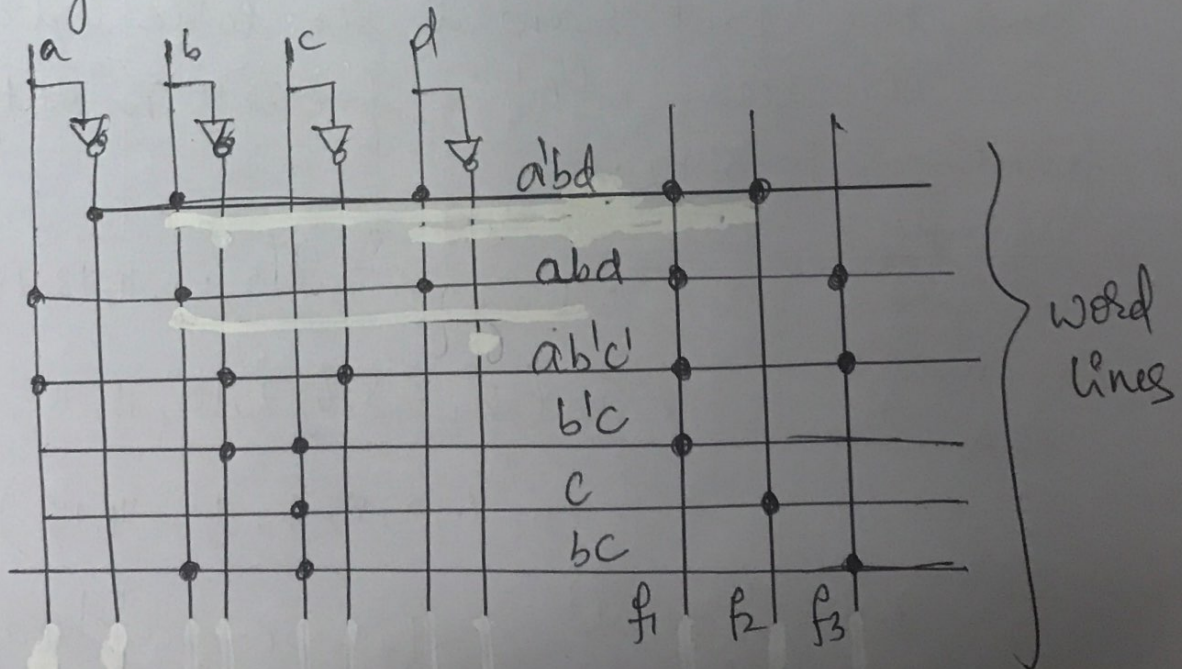


fig (4) : PLA Structure

Each row in the PLA table represents a general product term.

- Therefore, zero, one or more rows may be selected by each combination of input values.
- TO determine the value of f_i for a given input information/combination the values of f_i in the selected rows of the PLA table must be ored together.
- FOR the table 2 (1) If abcd = 0001, no rows are selected and all f's are 0.

(2) If abcd = 1001, only the third row is selected and $f_1 f_2 f_3 = 101$.

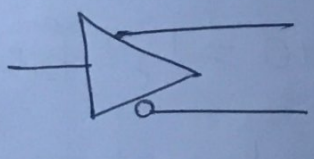
(3) If abcd = 0111, the first, fifth and sixth rows are selected.

Therefore $f_1 = 1+0+0 = 1$, $f_2 = 1+1+0 = 1$ and $f_3 = 0+0+1 = 1$.

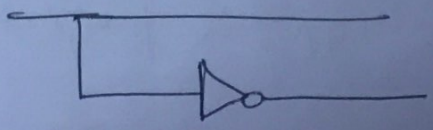
- PLAs can be mask-programmable and field programmable.
- Mask programmable type is programmed at time of manufacture.
- Field programmable PLA has programmable interconnection points that use electronic charges to store a pattern in the AND and OR arrays.

PROGRAMMABLE ARRAY LOGIC (PAL)

- The PAL is a special case of programmable logic array in which the AND array is programmable and the OR array is fixed.
- The basic structure is same as PLA.
- Since only AND array is programmable, the PAL is less expensive than the more general PLA & PAL is easier to program.
- Logic designers frequently use PALs to replace individual logic gates when several logic functions must be realized.
- Fig (5a) represents a segment of an unprogrammed PAL.

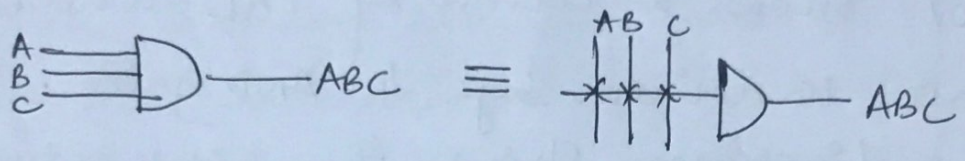


The symbol represents an input buffer which is logically equivalent to



- A buffer is used because each PAL input must drive many AND gate inputs.

- When the PAL is programmed, some of the interconnection points are programmed to make the desired connections to the AND gate inputs.
- Connections to the AND gate inputs in a PAL are represented by X's as shown.



EXAMPLE: REALIZATION OF FUNCTION $I_1 I_2' + I_1' I_2$ using PAL.

- The X's in fig 5(b) indicates that I_1 & I_2' lines are connected to the first AND gate, and the I_1' & I_2 lines are connected to the other gate.

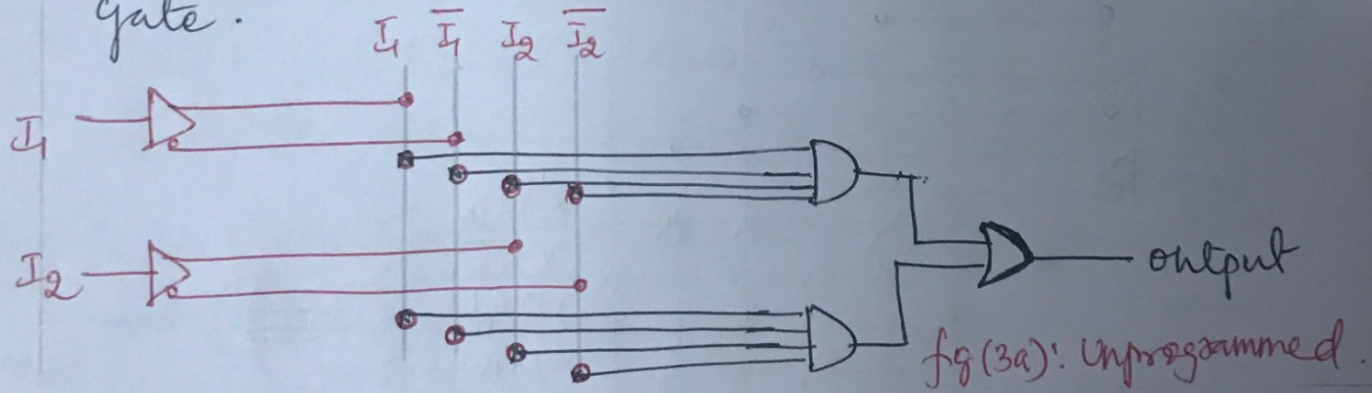


fig (3a): Unprogrammed.

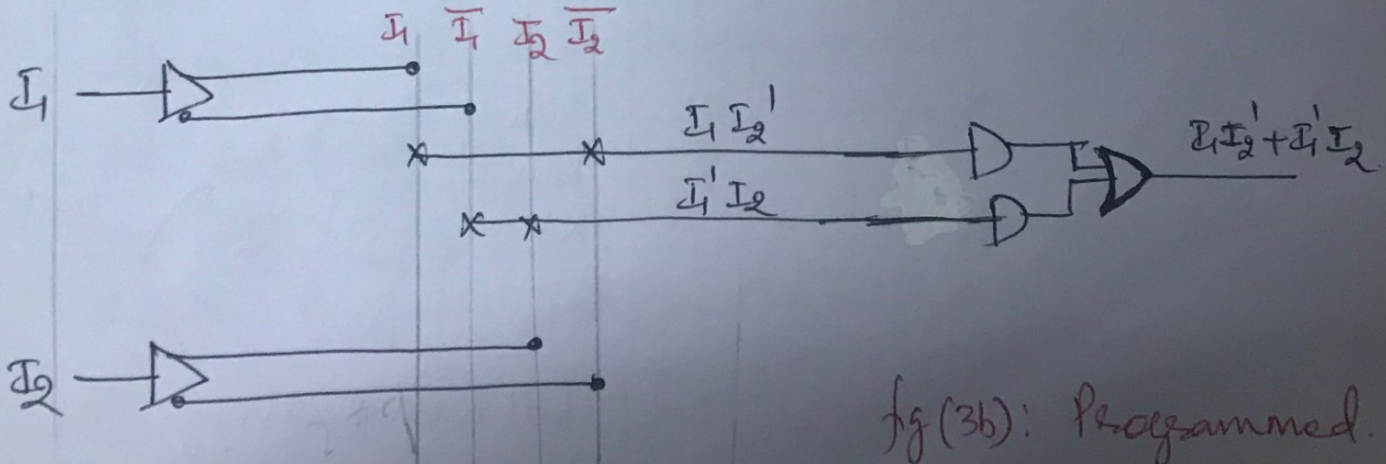


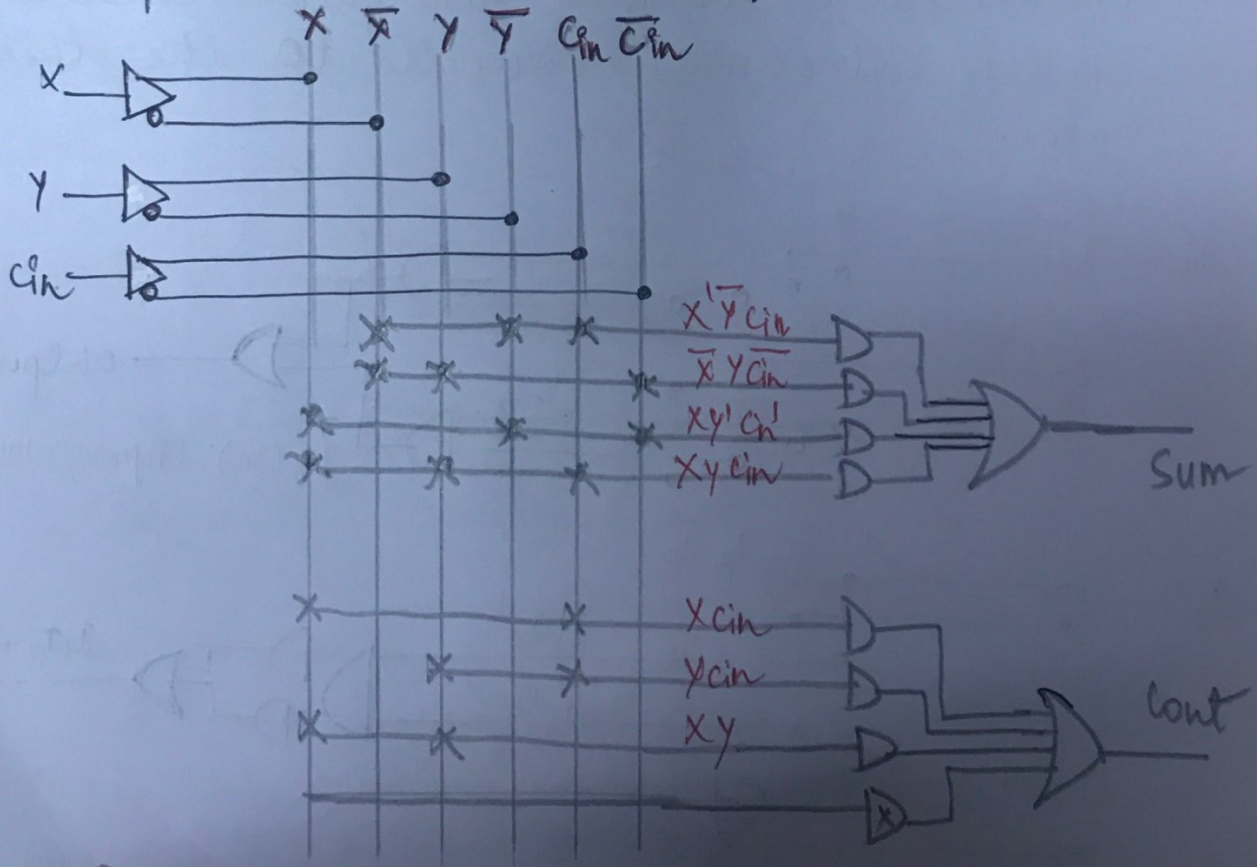
fig (3b): Programmed.

Example 2: Implementation of full adder using PAL.

$$Sum = X'Y'C_{in} + X'YC_{in}' + XY'C_{in}' + XYC_{in}$$

$$Cout = XC_{in} + YC_{in} + XY.$$

- Fig (6) shows a section of PAL where each OR gate is driven by 4 AND gates. The X's on the diagram show the connections that are programmed into the PAL to implement the full adder equations. For ex: , the first row of X's implements the product term $X'Y'C_{in}$.



fig(b): Implementation of full adder using PAL.

MODULE 3: FLIP FLOPS — PART A 3.1

> Basic Bistable elements ✓
> Latches ✓
> Timing considerations ✗
> Master-slave FlipFlops (Pulse Triggered F/F)
> SR FlipFlops ✓
> JK FlipFlops ✓
> Edge Triggered FlipFlops ✗
> characteristic Equations. ✓

III EC

DSD

Introduction to sequential circuits.

- A sequential circuit is defined as a circuit in which the output at any instant are dependent not only upon the inputs present at that instant but also upon the past history of inputs.
- The past history of inputs must be preserved by the network.
- Therefore sequential networks are said to have memory.

- There are 2 basic types of sequential ckts / networks. They are distinguished by the timing of the signals with in the network.

(1) A Synchronous sequential network is one in which its behavior is determined by the values of the signals at only discrete instants of time.
- These circuits have a master clock generator which produces a sequence of clock pulses.

(2) Asynchronous circuit is one in which its behavior is immediately affected by the input signal changes

- The basic logic element that provides memory in many sequential circuit is the Flip-Flop.
- The Flip-Flop is a simple sequential circuit. It is known that sequential circuits require the existence of feedback. Feedback is present in Flip-Flop circuits.
- A Flip-Flop has 2 stable conditions (Bistable). To each of these stable conditions is associated a state or the storage of a binary symbol.

THE BASIC BISTABLE ELEMENT.

- Flip-Flop circuit is the basic bistable element as shown.

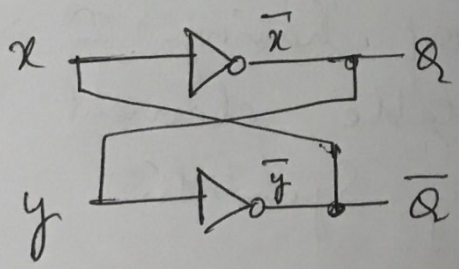


fig (1): Basic Bistable element.

- This circuit has 2 outputs. It consists of 2 cross coupled not gates i.e, the output of the first not gate serving as the input to the second and the output of the second not gate

serving as the input to the first.

- clearly this structure involves the feedback.

- Basic bistable element is a circuit having 2 stable conditions (states).

Case 1: Assume $x = 0$ $\therefore Q = \bar{x} = 1$

$\bar{x} = y = 1$ $\bar{y} = 0$

$\therefore \bar{Q} = \bar{y} = x = 0$.

Thus the circuit is stable with

$\bar{Q} = x = \bar{y} = 0$ & $Q = \bar{x} = y = 1$.

Case 2: Assume $x = 1$ $\bar{x} = 0$ $\therefore Q = \bar{x} = 0$

$y = \bar{x} = 0$, $\bar{y} = 1$ $\therefore \bar{Q} = \bar{y} = 1$.

- As a result of having 2 stable conditions, the basic bistable element is used to store binary symbols.

- The binary symbol that is stored in the basic bistable element is referred to as the content or state of the element. The state of the basic bistable element is given by the signal value at the Q-output terminal.

Hence the Q output terminal is called the normal output. & $\bar{Q}$ is referred to as the complementary output.

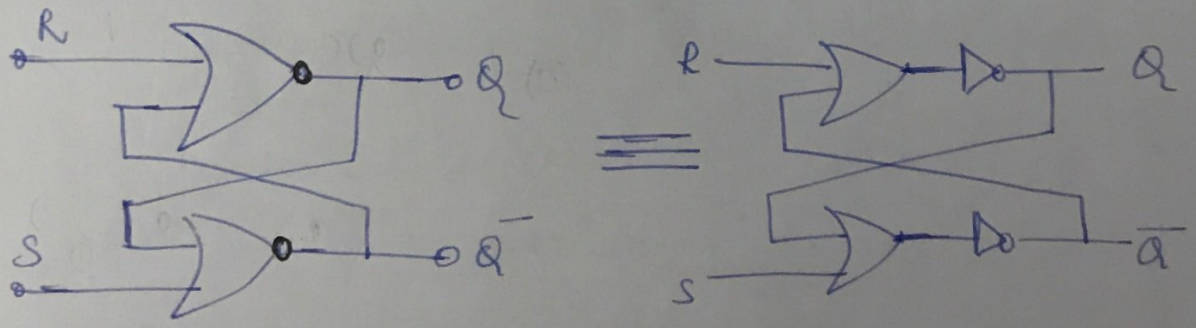
- When the device is storing a 1, it is said to be in its 1-state or set.
- When the device is storing a 0, it is said to be in its 0-state or reset.
- The basic bistable element has no inputs. When power is applied, it becomes stable in one of its 2 stable states. It remains in this state until power is removed.
- A Flipflop is a bistable device, with inputs, that remains in a given state as long as power is applied and until input signals are applied to cause its op to change.
- The process of storing a 1 into a flipflop is called setting or presetting the flipflop and storing a 0 is called resetting or clearing the flipflop.

LATCHES

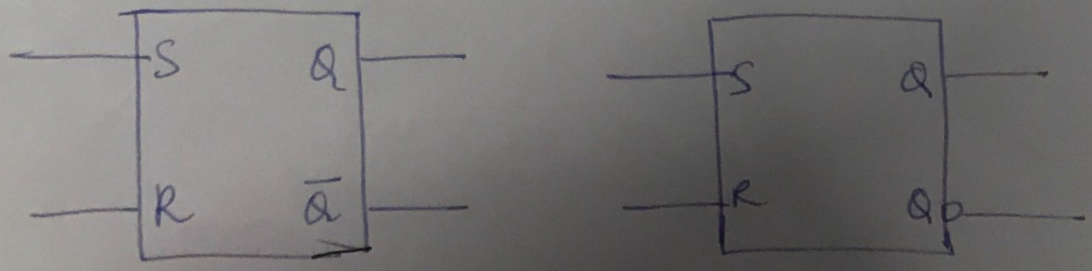
- Storage devices called latches form one class of flip flops.
- The output essentially responds immediately to changes on the input lines, although a special control signal, called the enable or clock, might also need to present.

SR LATCH

- Fig(2) shows the SR (Set - Reset) latch that consists of 2 cross - coupled nor gates. It has 2 inputs, S and R and 2 ops Q & $\bar{Q}$.



fig(2): Logic Diagram



fig(3): Logic Symbol.

Truth Table

Inputs S R	outputs Q^+ $\bar{Q}^+$
0 0	Q $\bar{Q}$
0 1	0 1
1 0	1 0
1 1	0* 0*

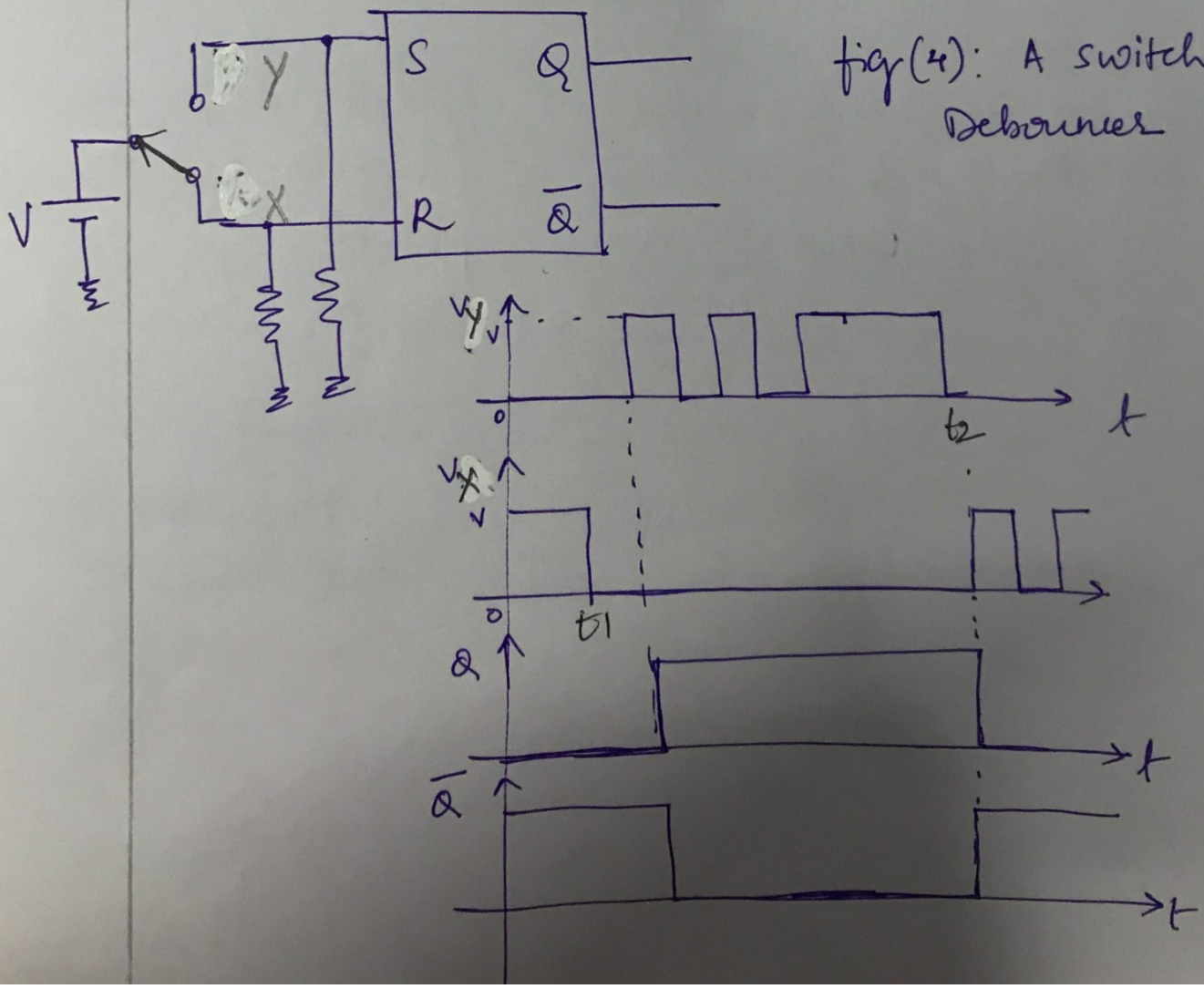
* unpredictable behavior will result if Inputs return to 0 simultaneously.

- Latch is in one of its two stable states when these inputs are applied.
- In the table, Q denotes the present state of the latch i.e, Q is the state of the device at the time the input signals are applied.
- Q^+ is called the next state of the latch.
- For S=0, R=0, the entries Q and $\bar{Q}$ in the Q^+ and $\bar{Q}^+$ columns, are interpreted as the next state of the device is the same as its present state.
- For S=0, R=1, Q=0 $\bar{Q}$=1
- For S=1, R=0, Q=1, $\bar{Q}$=0.

- For the above 3 situations the outputs Q and $\bar{Q}$ are complementary.
- When S=1, R=1., outputs of both nor gates = 0. consequently they are not complementary outputs.

S=R=1 input is frequently regarded as a forbidden input condition.

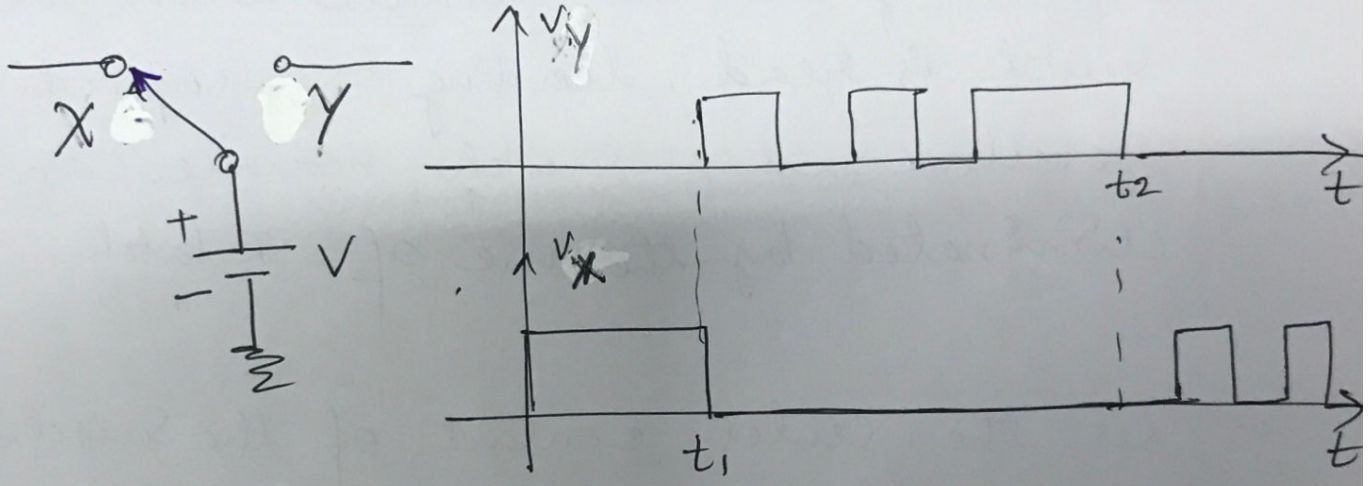
An application of the SR Latch : A SWITCH DEBOUNCER.



fig(4): A switch Debouncer

Switch / contact Bounce.

- When a mechanical switches such as toggle switches or push buttons are switched from one position to the other, several make and break operations occur at the second position called switch bounce or contact bounce.



- When the center contact is moved from X to Y at time t_1, the voltage at X gets to zero as soon as the contact leaves X.
- However when it touches Y, several make and break operations take place due to the spring like nature of the contacts. A similar occurence can be seen at X, when the center contact is moved from Y to X at time t_2 resulting in o/p as shown above.

This effect is called the switch bounce.

- This is undesirable when used with digital circuits because b/w t_1 and t_2 for example, the logic level read by a gate connected to Y could be a 0 or 1. depending on the instance when the switch is read, leading to unpredictable results, Such a switch bounce can be eliminated by the use of SR latch.

- Let the center contact of the switch be initially at position X. Therefore $S=0$, $R=1$ resulting $Q=0$ and $\bar{Q}=1$. as seen for time $t < t_1$.
- At $t=t_1$, the switch is moved from X to Y. After the center contact leaves X and till it reaches Y, $S=R=0$. Thus Q and $\bar{Q}$ remains in same state (i.e, $Q=0$, $\bar{Q}=1$)
- As soon as the center contact touches Y, the inputs to the latch $S=1$, $R=0$

and the latch sets to one with $Q=1$ and $\bar{Q}=0$.

- ~~when the centre contact leaves from B due to contact~~
- Thus the SR latch can be useful in removing contact bounce.

$\bar{S}\bar{R}$ LATCH

The $\bar{S}\bar{R}$ latch is constructed by cross coupling two Nand gates as shown in fig(5).

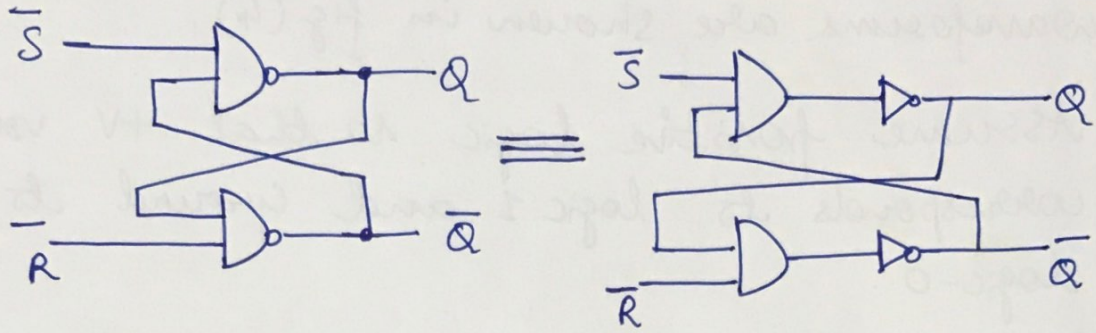


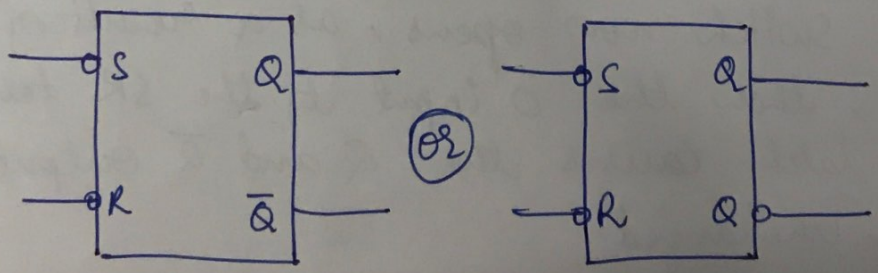
fig (5) : Logic Diagram

Truth Table

$\bar{S}\bar{R}$	Q^+	$\bar{Q}^+$
0 0	1*	1*
0 1	1	0
1 0	0	1
1 1	Q	$\bar{Q}$

* unpredictable behaviour will result if inputs return to 1 simultaneously.

Logic Symbol



- From the truth table, it is seen that when $\bar{S} = \bar{R} = 1$ the logic diagram reverts to the basic bistable element i.e, cross coupling of 2 not gates.
- Thus the device has 2 stable states.
- If one of the inputs to $\bar{S}\bar{R}$ latch is made 0 and other is 1, then the output of the nand-gate having the o/ps becomes 1. This in turn, is applied as an input to the second nand gate that also has a 1 as its other ip. consequently the output of the nand gate becomes 0.

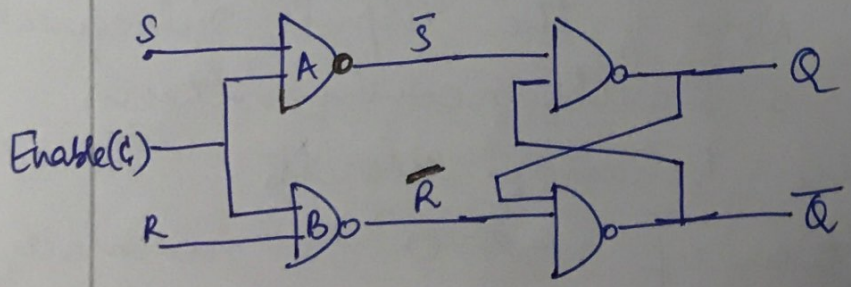
  Thus $\bar{R} = 0$ & $\bar{S} = 1$ the latch resets.

  $\bar{R} = 1$ & $\bar{S} = 0$ the latch sets.

If $\bar{R} = 0$ & $\bar{S} = 0$. then both the ops becomes 1. Now if the inputs subsequently return to 1 simultaneously, then unpredictable behavior results. Thus the application of $\bar{S} = \bar{R} = 0$ is normally not recommended.

THE GATED SR LATCH

- The inputs for both SR and $\bar{S}\bar{R}$ latch are asynchronous. That is, a change in value of these inputs causes an immediate change of the outputs.
- It is frequently desirable to prevent input activation signals from affecting the state of the latch immediately, but rather to have the effect occur at some desirable time or alternatively, to allow the input changes to be effective only during a prescribed period of time.
- For these situations, a gated SR latch is used. The gated SR latch is also called an SR Latch with enable.
- A gated SR Latch is as shown in fig( )

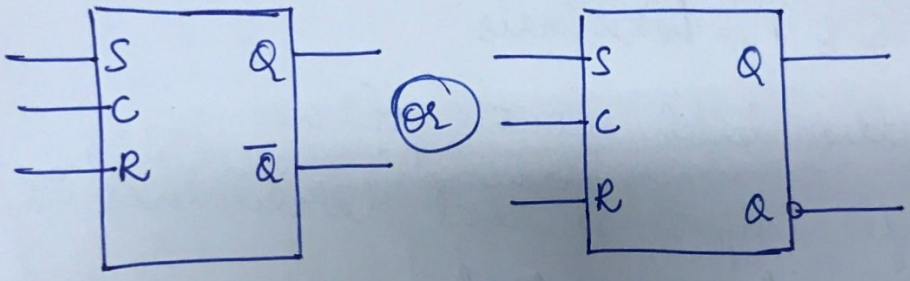


fig(6): Logic Diagram.

Truth Table

C	S	R	Q^+	$\bar{Q}_+$
1	0	0	Q	$\bar{Q}$
1	0	1	0	1
1	1	0	1	0
1	1	1	1*	1*
0	X	X	Q	$\bar{Q}$

Logic Diagram



- A gated SR latch consists of $\bar{S}\bar{R}$ latch along with 2 additional Nandgates and a control input C, referred to as enable, gate or clock input.
- The enable input C, determines when the S and R inputs become effective.
- As long as the enable input is 0, the ops of nandgates A and B are 1.
- In this case, any changes on the S & R lines are blocked and the output is said to be latched in its present state (disabled).

- When the enable signal is 1, the gated latch is said to be enabled. Here, the latch behaves as a regular SR latch.

THE GATED D LATCH.

- The Gated D latch is a gated SR latch in which a NOT Gate is connected between the S & R terminals.
- Thus the latch consists of a
  1. Single Input D that determines its next state
  2. a control or enable input C that determines when the D input is effective.

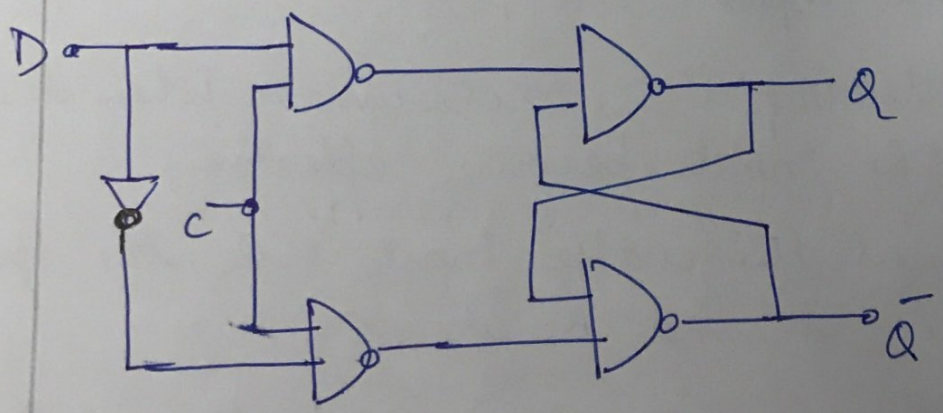
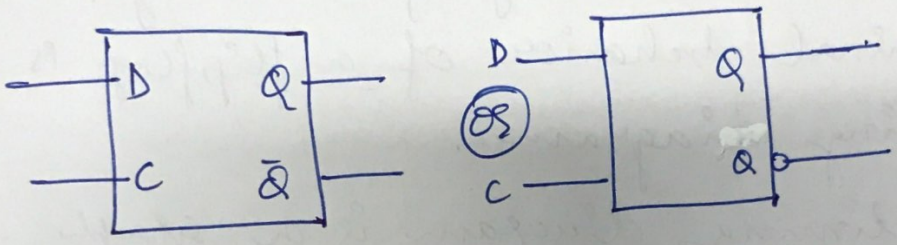


fig (7): Logic Diagram

Truth Table

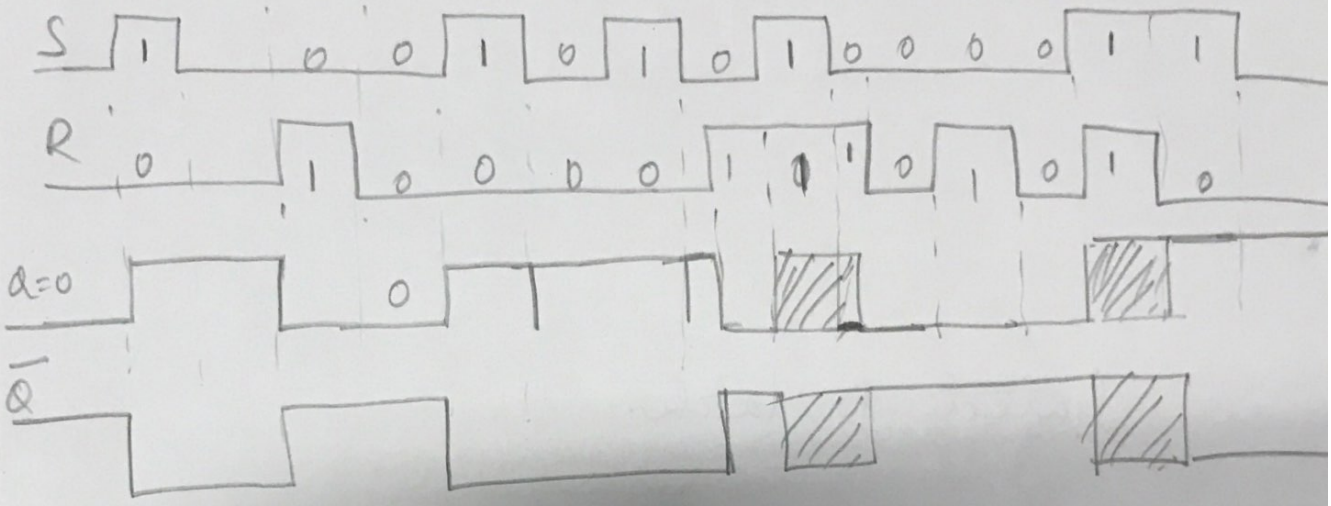
D	C	Q^+	$\overline{Q^+}$
0	1	0	1
1	1	1	0
X	0	Q	$\overline{Q}$

Logic Symbol

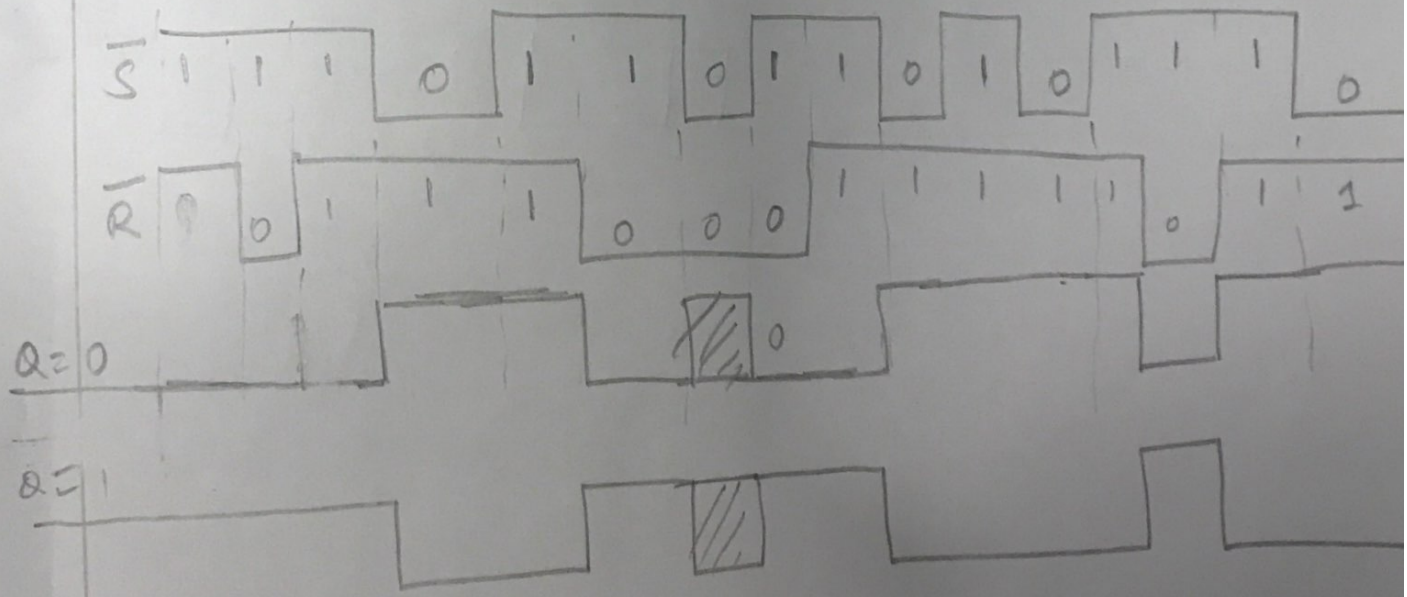


- When C=1, the output of the latch follows the values applied to the D input terminal.
  If D=0 then Q=0 & if D=1 then Q=1.
- When latch is disabled i.e, C=0, the latch remains in the state prior to the enable signal going to 0.
- The gated D latch avoids the S=R=1 condition.

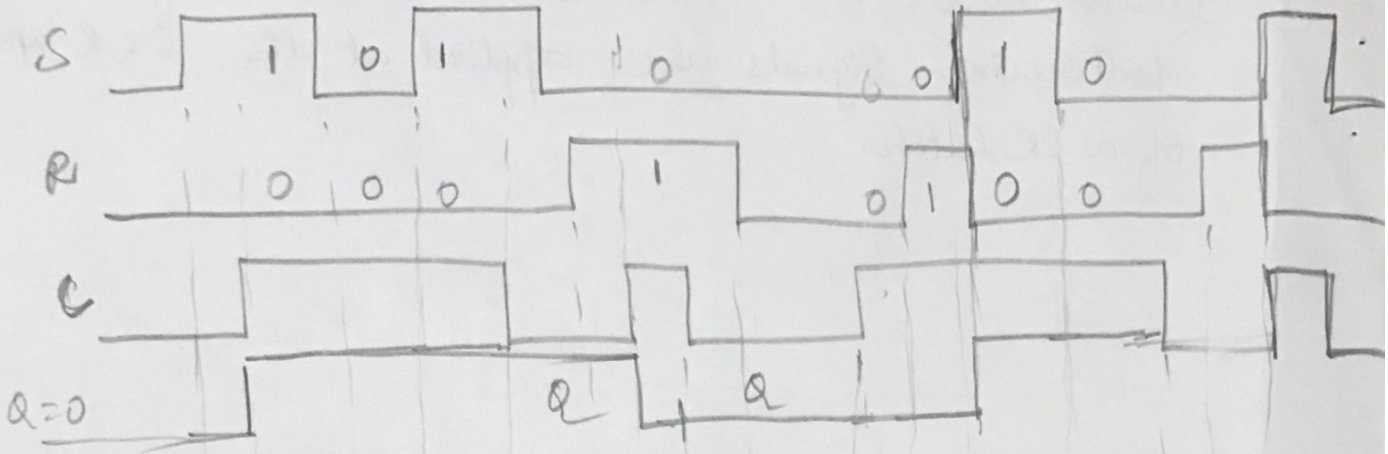
(1) Draw the waveforms at Q and $\bar{Q}$ if the following signals are applied at the S & R inputs of a SR Latch.



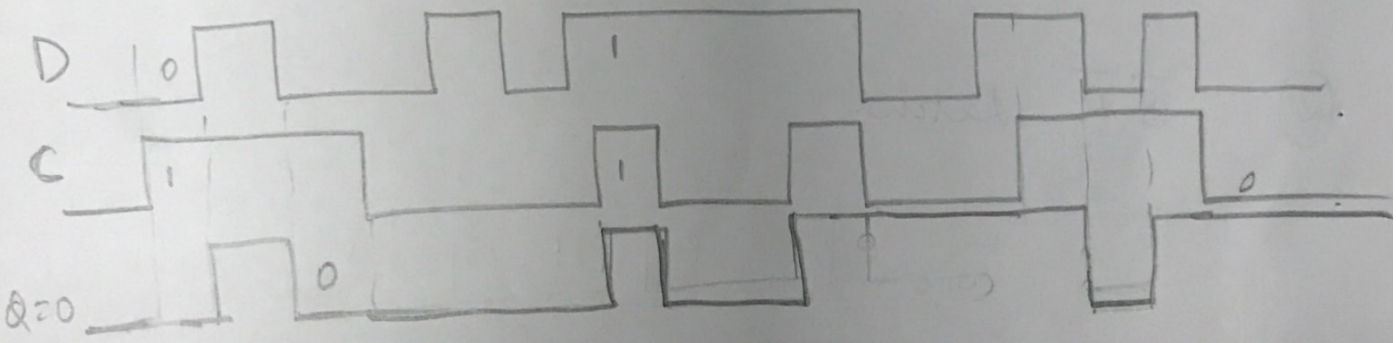
(2) $\bar{S}\,\bar{R}$ Latch.



Gated SR latch



Gated D latch.



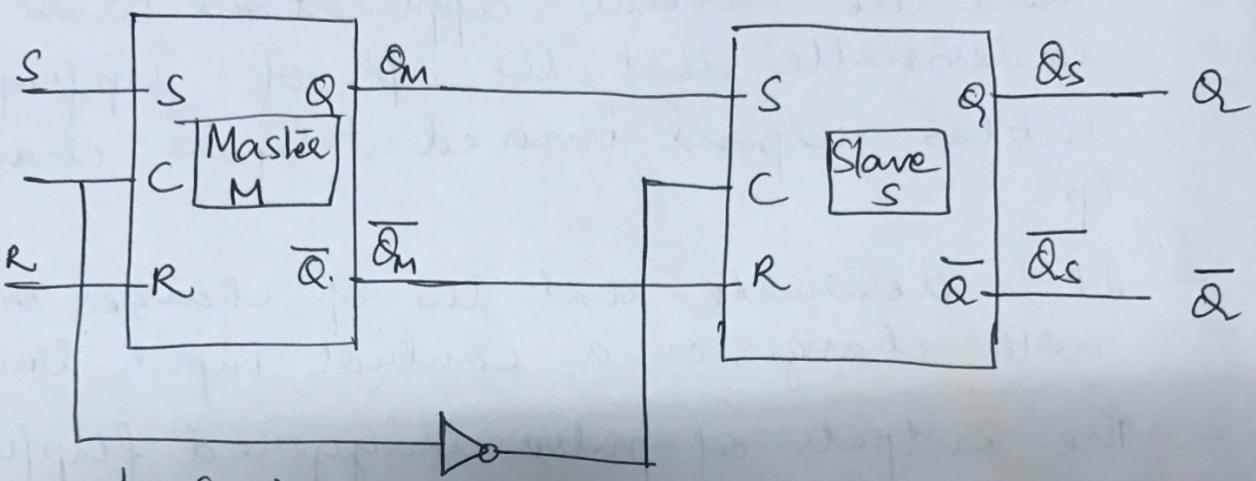
PULSE TRIGGERED FLIPFLOPS

- Pulse triggered flipflops are also referred to as Master slave flip flops.
- There are several applications where it is undesirable that the ops of flip flops and latches respond immediately to changes in ip.
- It is desirable that the op changes only with changes on a control input line.
- The outputs of pulse triggered flipflops and edge triggered flipflops respond only when changes take place on their control ip lines.

1. Pulse triggered Master slave SR Flipflop.

- The master slave flipflop has 2 sections
  a) Master
  b) Slave
  } cascaded together.
- The master register the data on one level (say logic 1) of the input control signal, which is transferred to the slave on the other level (logic 0) of the input control signal.

Fig(9a) shows a Master slave SR flipflop configured using two gated SR Latches.



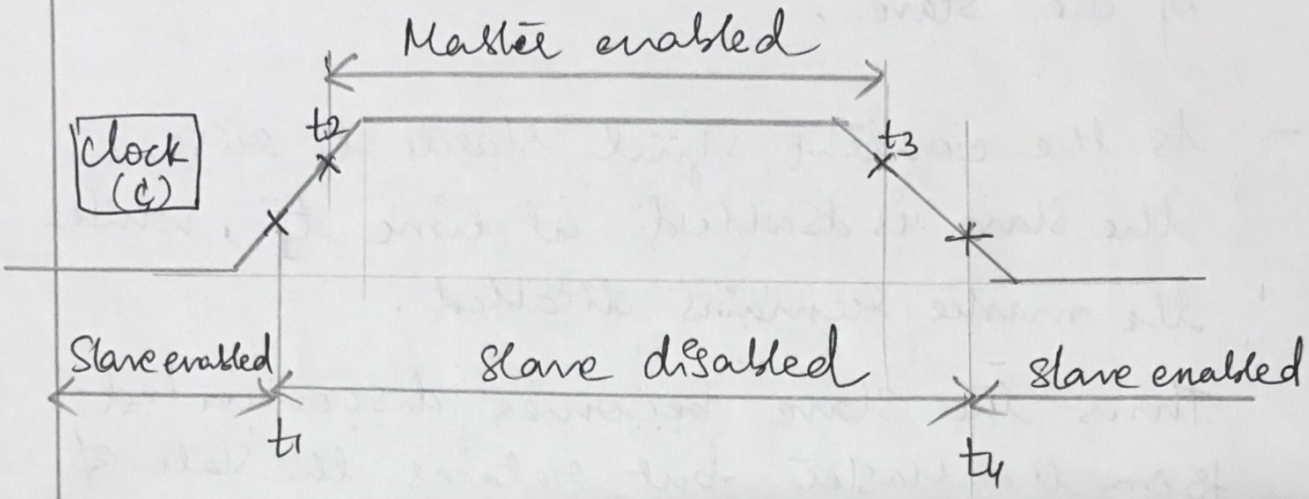
fig(9a): Master Slave SR Flipflop

Truth Table.

S	R	C	Q^+	$\bar{Q}^+$
0	0	⊓	Q	$\bar{Q}$
0	1	⊓	0	1
1	0	⊓	1	0
1	1	⊓	NOT DEFINED	
X	X	0	Q	$\bar{Q}$

- The information input lines S and R are used to set and reset the flipflop.
- A clock signal C is applied to the control ip line.

- The timing behavior of the Master slave flip flop is referenced to the control signal as shown in fig( ).



- The transition of the control signal from its low to high value (0 to 1) in positive logic is called rising/leading/positive edge of the control signal.
- Similarly from high to low (1 to 0) in positive logic, is called falling/trailing/negative edge of the control signal.

Case 1 : When C=0, the Master being a gated SR latch is disabled and any changes on the S and R input lines are ignored.

- At the same time, the slave is enabled due to the presence of the inverter. Hence the slave is in the same state as that

of the master. Since the Q_M and $\overline{Q_M}$ outputs of the master are connected to the S and R inputs respectively of the slave.

- As the control signal starts to rise, the slave is disabled at time t_1, while the master remains disabled.

Thus the slave becomes disconnected from the master but retains the state of the master.

- The control signal continues to rise, and it is at time t_2 that the master is enabled.

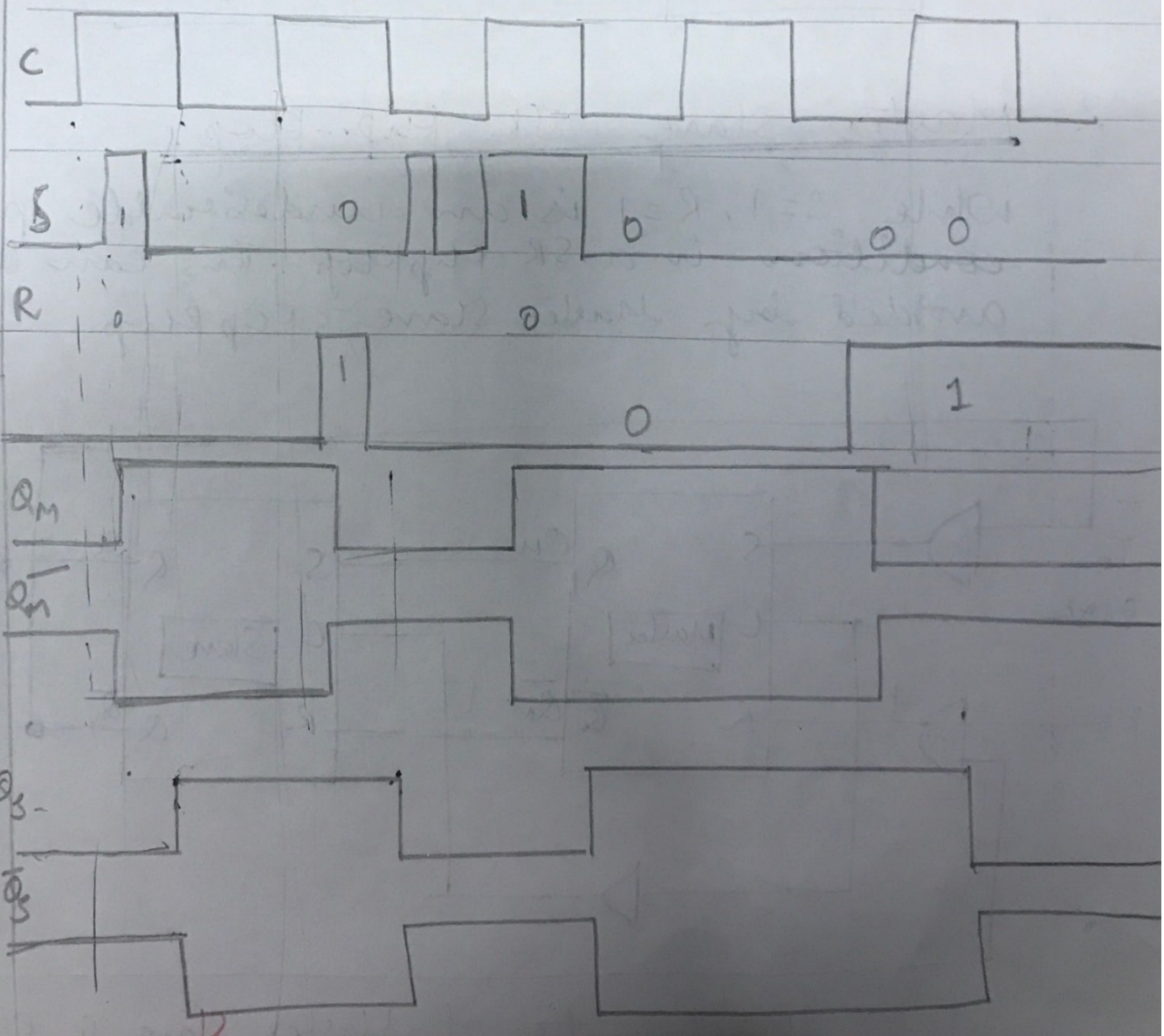
Case 2: When C=1, the master being a gated SR latch, responds to the inputs on the S and R lines.

- Since the slave is disabled due to the presence of the inverter, any changes to the state of the master are not reflected to the slave.

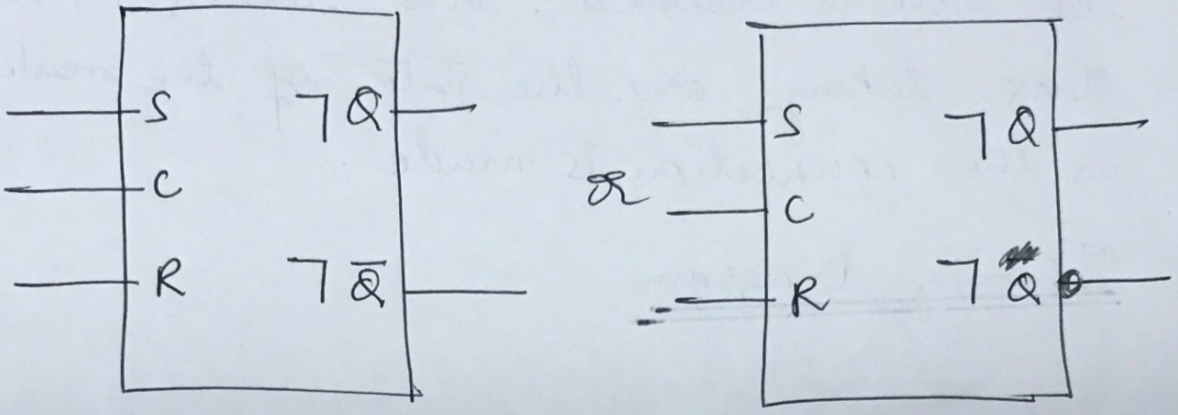
- The control signal is subsequently returned to its low level at time t_3.

- At this time, the master is disabled, causing it to latch on to its new state. However, it is not until time t_4, that the slave is enabled. This results in the slave taking on the state of the master as the connection is made.

Timing Diagram

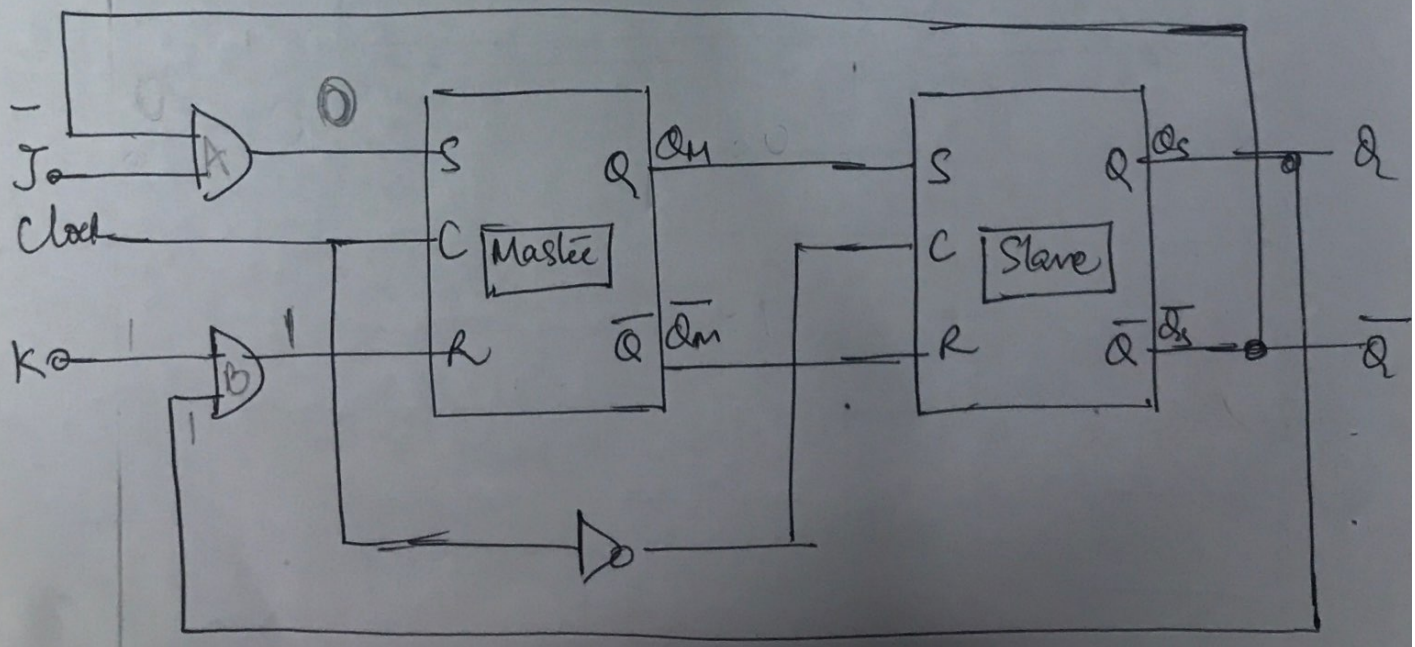


Logic Symbols of Master Slave SR Flip Flop.



2. Master Slave JK Flip-Flop:

While S=1, R=1 is an undesirable ip condition in a SR FlipFlop. This can be avoided by Master Slave JK Flip Flop.



fig(10) : Schematic of Master Slave JK F/F.

Truth / Functional table.

J	K	C	Q^+	$\bar{Q}^+$
0	0	⊓	Q	$\bar{Q}$
0	1	⊓	0	1
1	0	⊓	1	0
1	1	⊓	$\bar{Q}$	Q
x	x	0	Q	$\bar{Q}$

Case 1a: When J=K=1. Let Q=1, $\bar{Q}$=0.

O/p of AND gate A is 0 and o/p of AND gate B is 1. This means S=0 and R=1 for the master flip flop. When the clock goes high Q_M=0, $\bar{Q}_M$=1.

On the next falling edge of the clock, the state of the master is transferred to to the slave. Thus Q_S=0 & $\bar{Q}_S$=1

or Q=0, & $\bar{Q}$=1.

Thus when J=1=K, Input has caused the output to change.

Case 1b: when J=K=1, let Q=0, $\bar{Q}$=1

O/p of Andgate 1 = 1 = S

O/p of Andgate 2 = 0 = R

During the rising edge of the clock Master sets the op $Q_M = 1$ and $\overline{Q_M} = 0$.

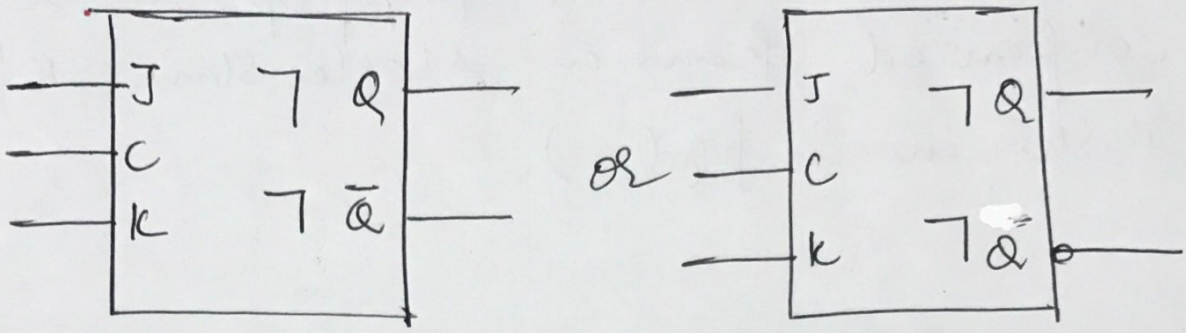
During the falling edge of the clock, the state of the Master is transferred to the slave making $Q = 1$ & $\overline{Q} = 0$.

Case 2: Let $J = K = 0$., corresponds to $S = 0, R = 0$. thus the next state is same as previous state.

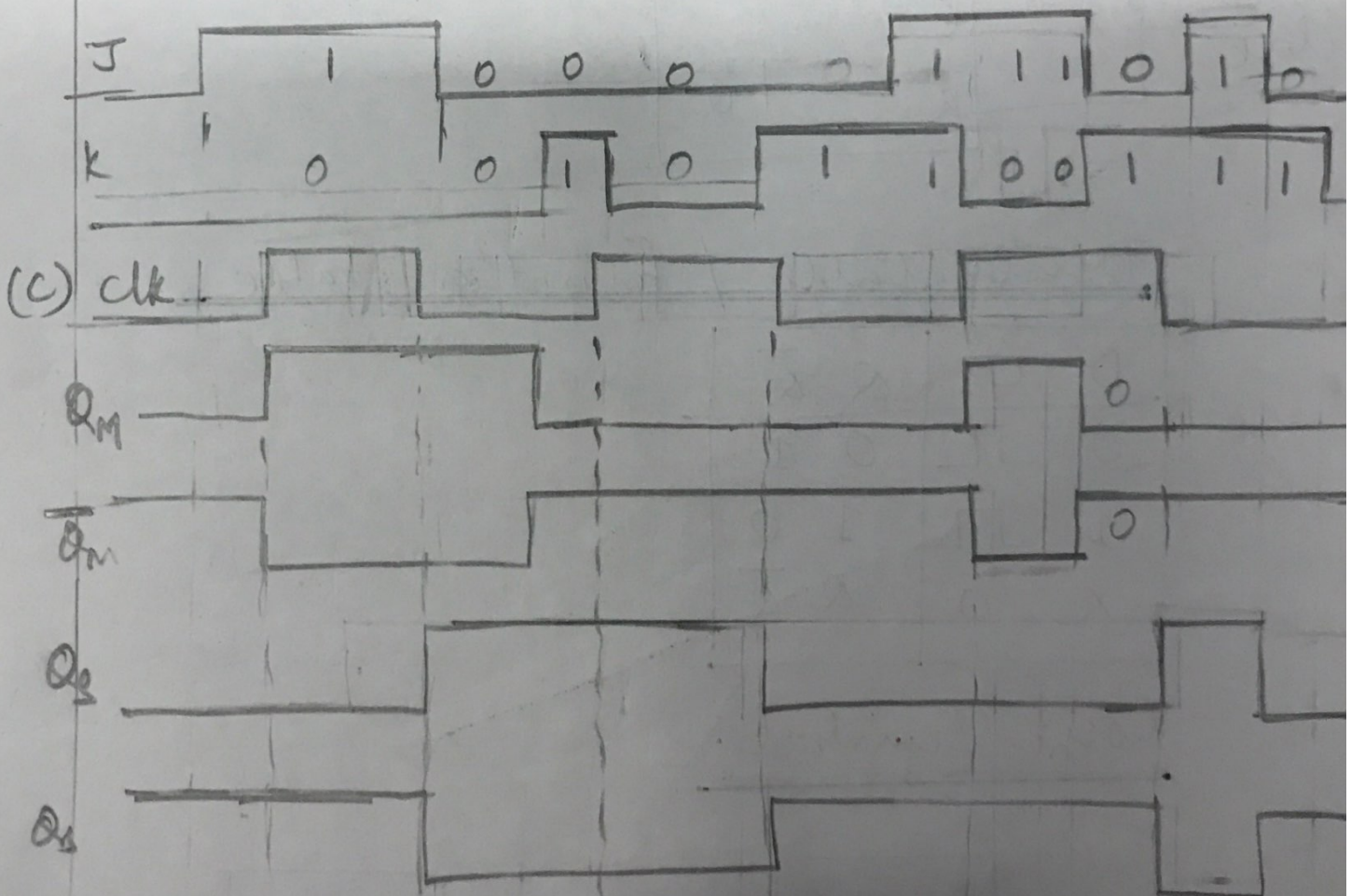
Case 3: Let $J = 1$; $K = 0$ with $Q = 0$ & $\overline{Q} = 1$. corresponds to $S = 1, R = 0$ Master sets $Q_M = 1$ & $\overline{Q_M} = 0$ in the rising edge of the clock & Slave sets $Q_S = 1$ & $\overline{Q_S} = 0$ in the falling edge of the clock.

Case 4: Let $J = 0, K = 1$ with $Q = 1$ & $\overline{Q} = 0$ corresponds to $S = 0$ & $R = 1$ Master resets Q_M to 0 and $\overline{Q_M} = 1$ in the rising edge of clock and slave transfers the state of Q_M and $\overline{Q_M}$ to Q and $\overline{Q}$ in the falling edge of the clock.

Symbols of Master Slave JK Flip Flop

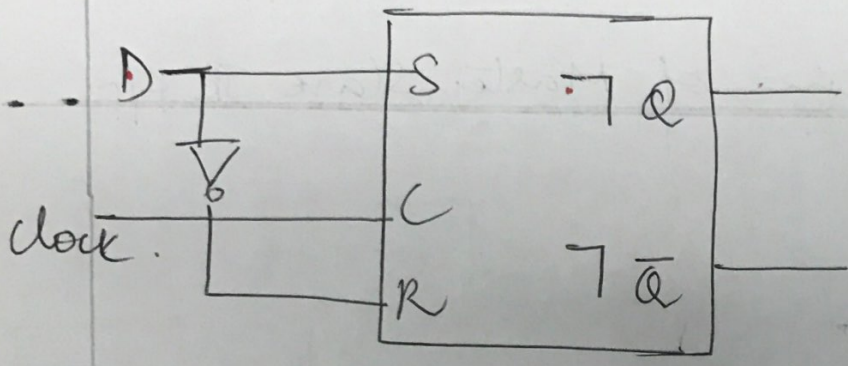


Timing Diagram of Master Slave JK F/F



Master slave D flip flop.

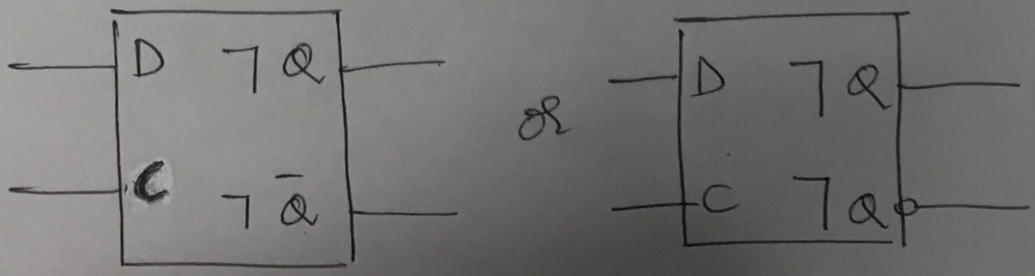
- A master slave D flipflop can be configured from a master slave SR flipflop as shown in fig( ).



Truth table / function table.

D	C	Q	$\bar{Q}$
0	⎍	0	1
1	⎍	1	0
X	0	Q	$\bar{Q}$

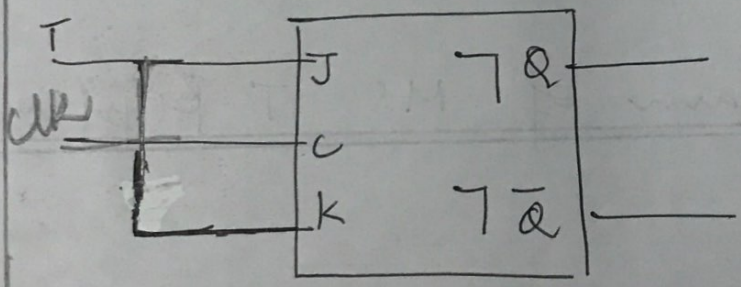
Logic Symbol.



The master registers D Input during the period when clock is high and the state of the Master is transferred to slave at the falling edge of the clock.

Master Slave T Flip Flop.

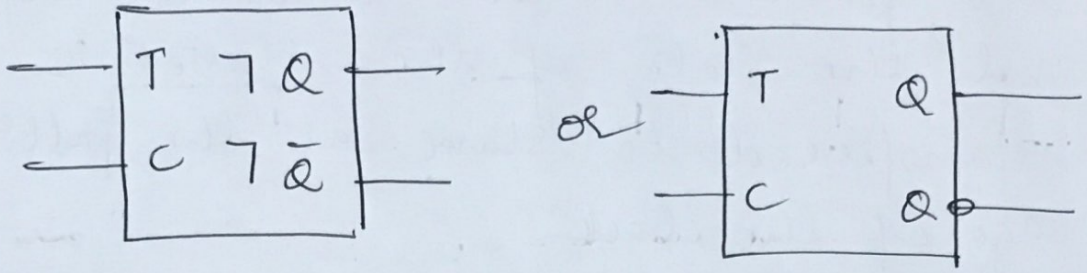
A master slave T Flip Flop can be configured from a master slave JK flip flop as shown.



Function / Truth Table

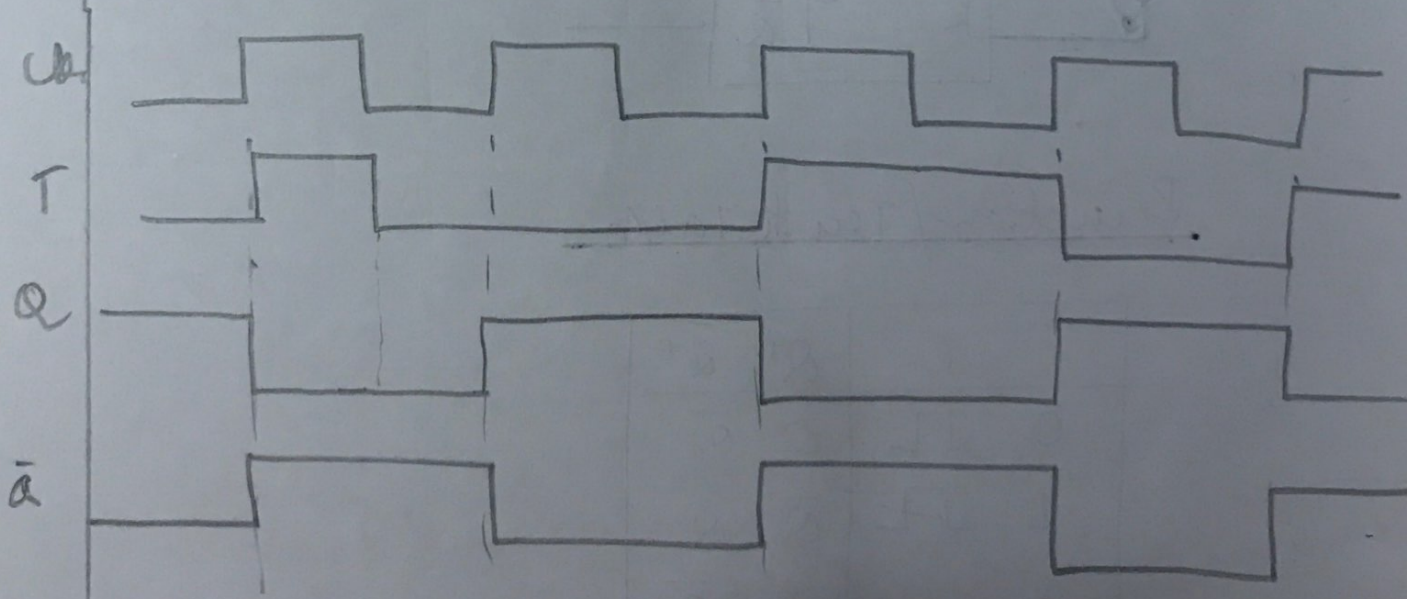
T	C	Q^+	$\bar{Q}^+$
0	⊓	Q	$\bar{Q}$
1	⊓	$\bar{Q}$	Q
X	0	Q	$\bar{Q}$

Logic Symbol

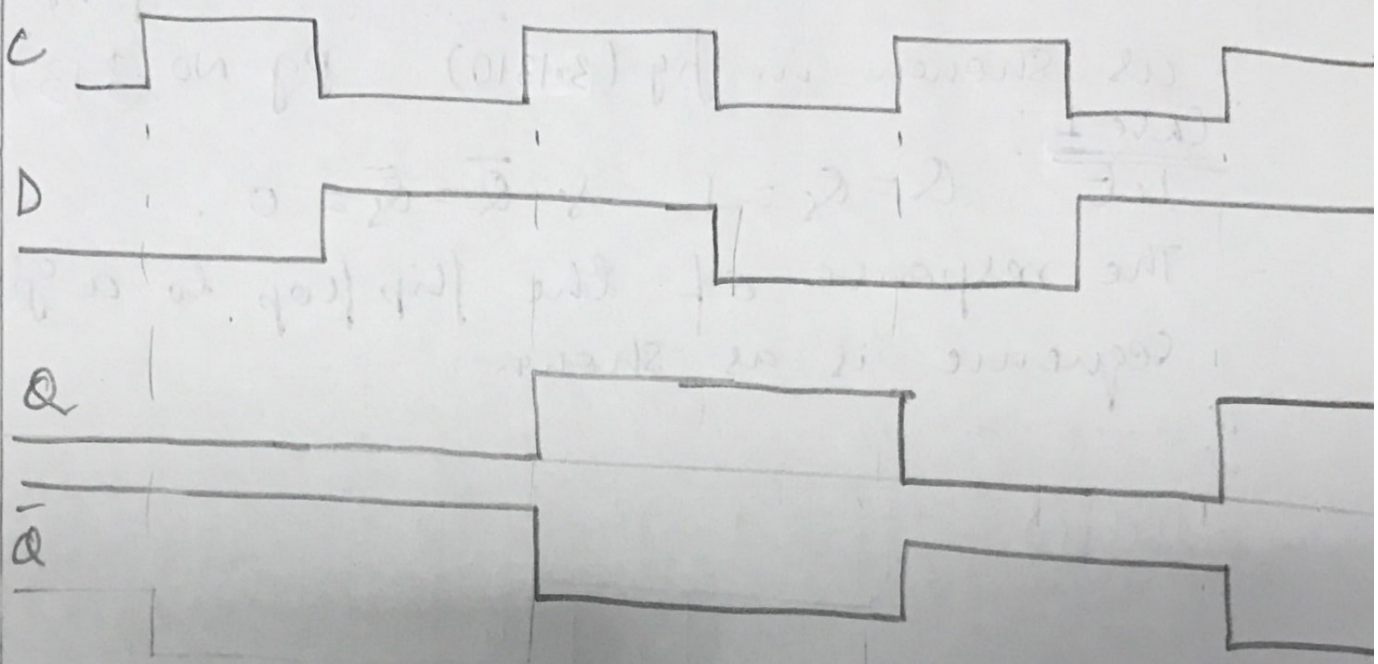


When T=0 (i.e, J=0, K=0); the next state of the T-flipflop equals to its present state and when T=1 (J=K=1) the next state is the complement of the present state.

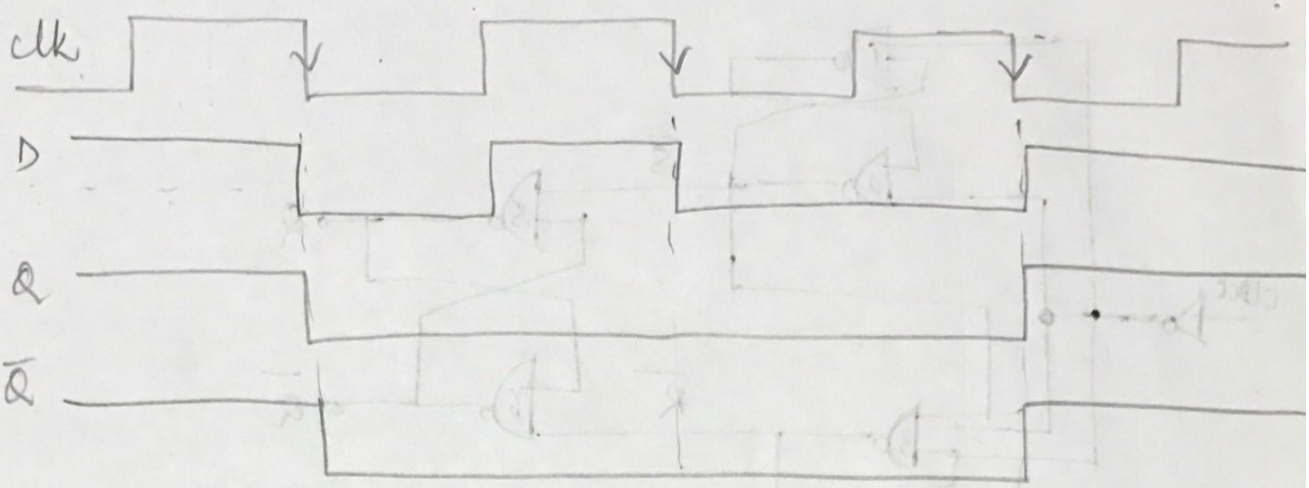
Timing Diagram of MS T Flipflop.



Timing Diagram of MS D Flipflop.



Timing waveforms



CHARATERISTIC EQUATIONS

- A variation of the simplified function tables, called the next state tables is given in following Table.

S	R	Q^+
0	0	Q
0	1	0
1	0	1
1	1	–

J	K	$\bar{Q}^+$
0	0	Q
0	1	0
1	0	1
1	1	$\bar{Q}$

D	Q^+
0	0
1	1

T	$\bar{Q}^+$
0	Q
1	$\bar{Q}$

- The next state table show the value of the next state of the flipflops for each combination of values to the present state of the flipflops and their information Input lines.

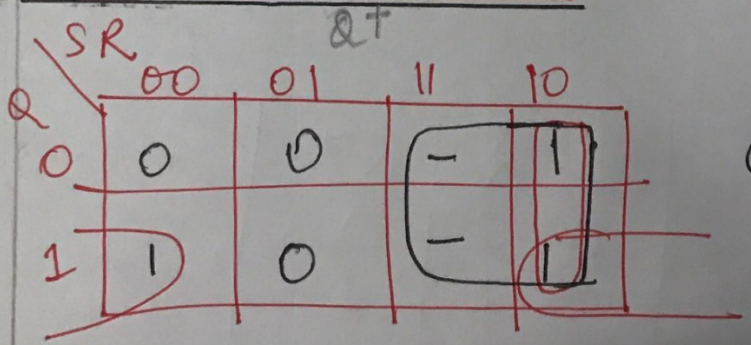
Since Q can have two values, each row of the simplified function table becomes 2 rows in the next state table.

- For each table, the appropriate interpretation is that for a given present state Q and inputs, the application of a control signal causes the flip flop to change to the next state Q^+.

- The Algebraic description of of the next state table of a flip flop is called the characteristic equation of the flip flop.

- This description is easily obtained by constructing the K-Map for Q^+ in terms of the present state and information input variables.

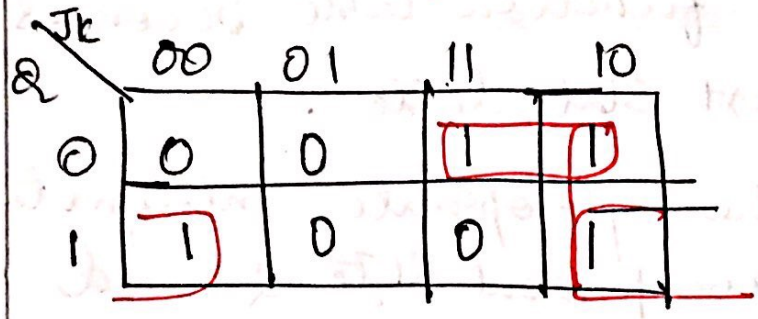
SR Flip flop.



$Q^+ = \bar{R}Q + S$

S	R	Q	Q^+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	–	–
1	1	–	–

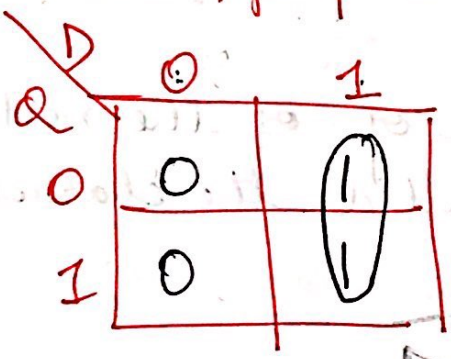
JK Flip flop.



$$Q^+ = J\bar{Q} + \bar{K}Q$$

J K Q	Q^+
0 0 0	0
0 0 1	1
0 1 0	0
0 1 1	0
1 0 0	1
1 0 1	1
1 1 0	1
1 1 1	0

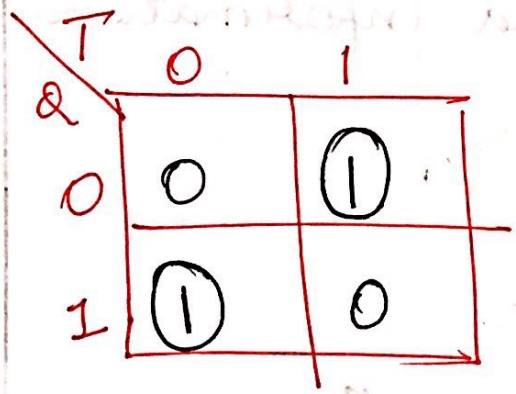
D Flipflop



$$Q^+ = D$$

D Q	Q^+
0 0	0
0 1	0
1 0	1
1 1	1

T Flipflop



$$Q^+ = T\bar{Q} + \bar{T}Q$$

$$= T \oplus Q$$

T Q	Q^+
0 0	0
0 1	1
1 0	1
1 1	0

Characteristics Equations.

D F/F.

Present State(Q)	Next State Q+	Input D
0	0	0
0	1	1
1	0	0
1	1	1

T F/F

PS Q	NS Q+	I/P T
0	0	0
0	1	1
1	0	1
1	1	0

SR F/F

PS Q	NS Q+	I/P SR
0	0	0X
0	1	10
1	0	01
1	1	X0

JK F/F

PS Q	NS Q+	I/P JK
0	0	0X
0	1	1X
1	0	X1
1	1	X0

DESIGN OF SYNCHRONOUS COUNTERS.

- Synchronous counter is characterised by the count pulses being applied directly to the control ips, C, of the clocked flipflops that comprise the counter.
- As a result, all the flipflops change simultaneously and the new state of the counter is observable in a minimum amount of time.
- Any counter is a pattern generator in which the counting sequence serves as the order in which a series of various patterns is produced. These patterns can then be used to enable / disable various portions of a logic network so as to control its behavior.
- In any event, non binary counting sequences are often desirable.
- In this context, general procedure for designing synchronous counters having prescribed op pattern is developed using 4 types of clocked flip flops.

DESIGN OF SYNCHRONOUS MOD-6 COUNTER USING CLOCKED JK FLIPFLOP

- Consider a general structure assuming the use of JK flip flop as shown in fig (4.

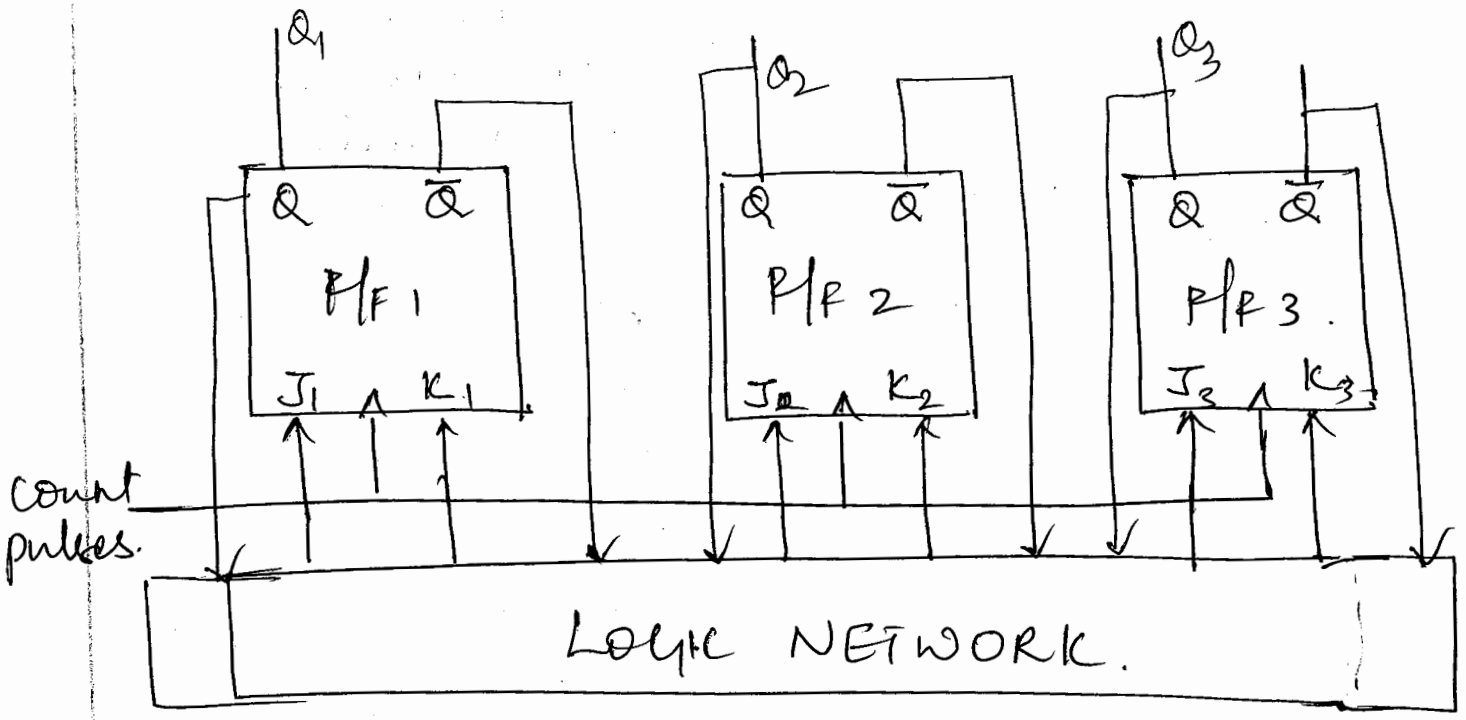


Fig (4.12) : General structure of Mod6 counter.

Counting Sequence → (0, 2, 3, 6, 5, 1)

Q_1	Q_2	Q_3
0	0	0
0	1	0
0	1	1
1	1	0
1	0	1
0	0	1

- The 3 clocked flip flops have the count pulses applied directly to their control ips.
- The current state of the counter is applied to a logic network. The function of the logic network is to generate the appropriate signals for the J and k terminals of the clocked flip-flops so that the specified next state in the counting sequence results upon the occurence of the triggering edge of a clock pulse.
- In this case, the logic structure of the network can be described by 6 Boolean expressions, one for each of the six inputs to the three flip-flops, in terms of the Boolean variables Q_1, Q_2 and Q_3 that correspond to the present state of the counter.
- TO obtain these expressions, a truth table for the logic network — Excitation table is first developed and then the simplified Boolean expressions are obtained.

Table 1. Shows the excitation table for the Synchronous Mod-6 counter.

- It is divided into 3 Sections
  a) Present State
  b) Next State
  c) Flip flop inputs.

- The counting sequence is listed in the present state Section and the desired next state for each present state is entered in the next Section.

- Third table is filled using the terminal behavior of the JK flip flop — Application table as shown in Table 2.

	Present State	Next State	Flip flop Inputs		
	$Q_1Q_2Q_3$	$Q_1Q_2Q_3$	J_1K_1	J_2K_2	J_3K_3
0	000	010	0X	1X	0X
2	010	011	0X	X0	1X
3	011	110	1X	X0	X1
6	100	101	X0	X1	1X
5	101	001	X1	0X	X0
1	001	000	0X	0X	X1

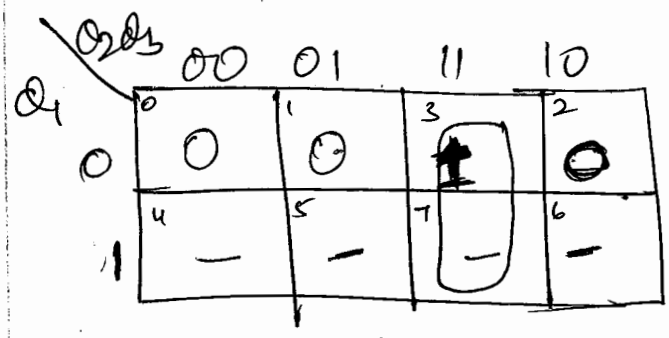
Table 1; Excitation table for

Q	Q^+	J	K
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

Table 2: Application table for clocked JK flip flop.

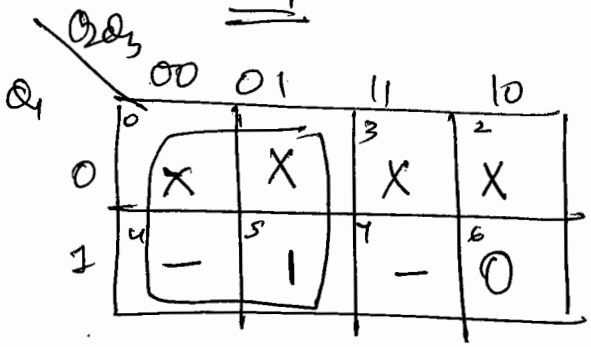
Determination of Minimal Sum expressions using K-Map.

J_1



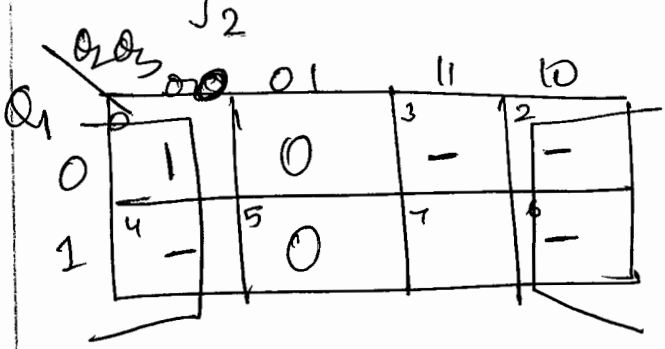
$J_1 = Q_2Q_3$

K_1



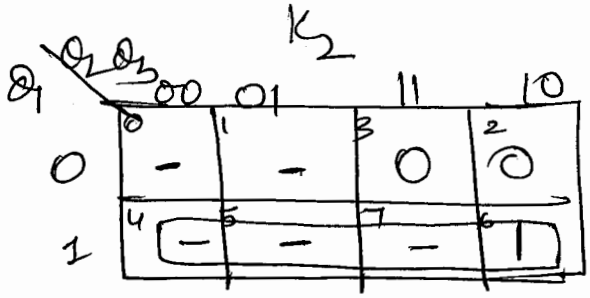
$K_1 = \overline{Q_2}$

J_2



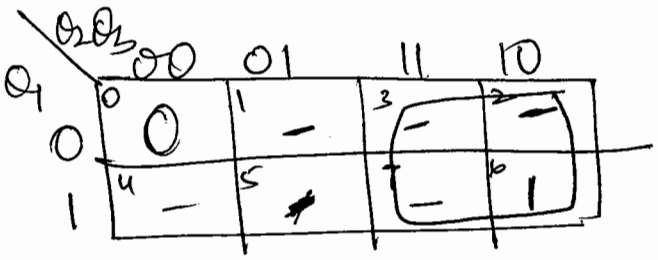
$J_2 = \overline{Q_3}$

K_2

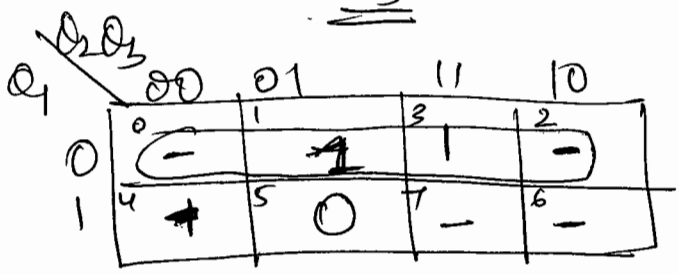


$K_2 = Q_1$

J_3



K_3



$J_3 = Q_2$

$K_3 = \overline{Q_1}$

— These expressions lead to the logic diagram for the MOD - Synchronous counter as shown in fig (4.13).

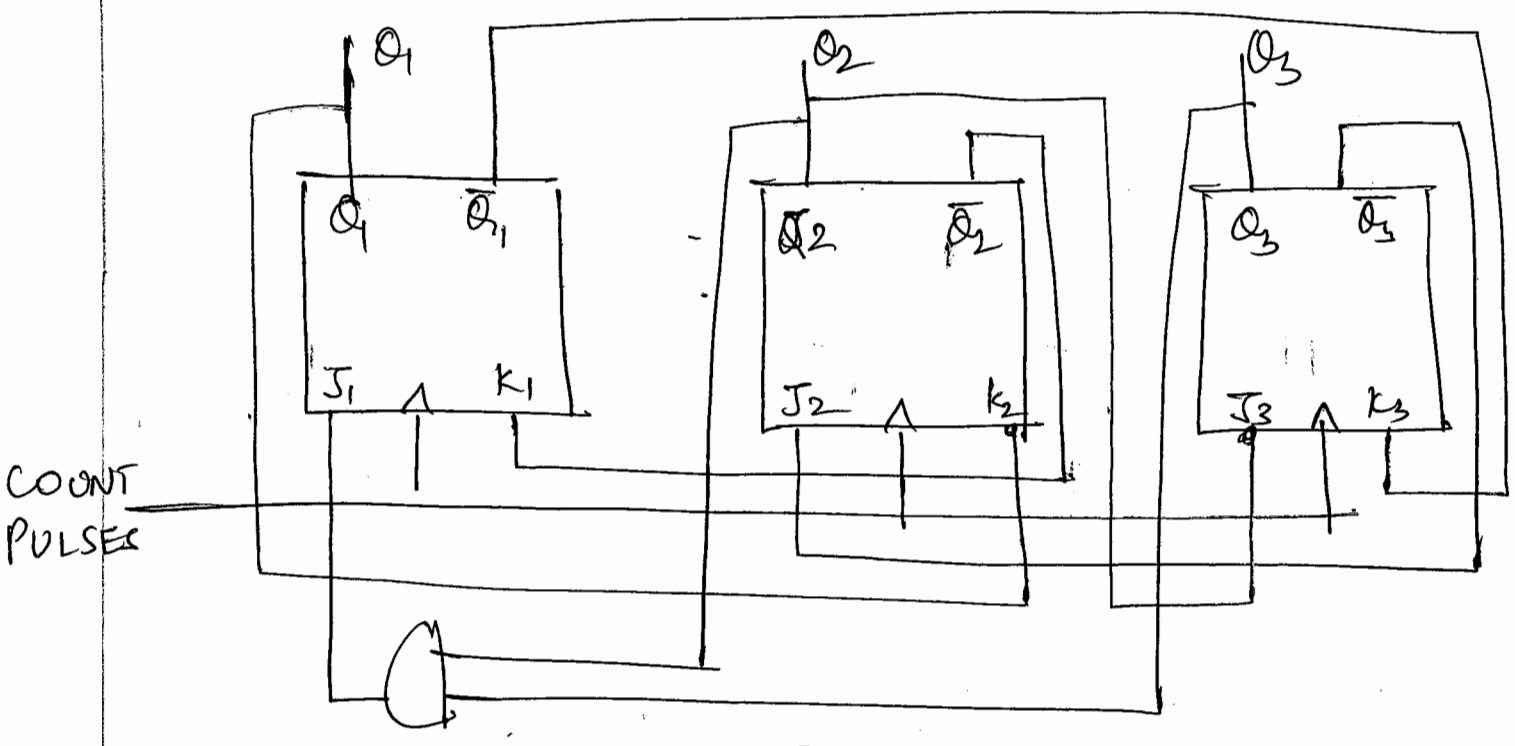
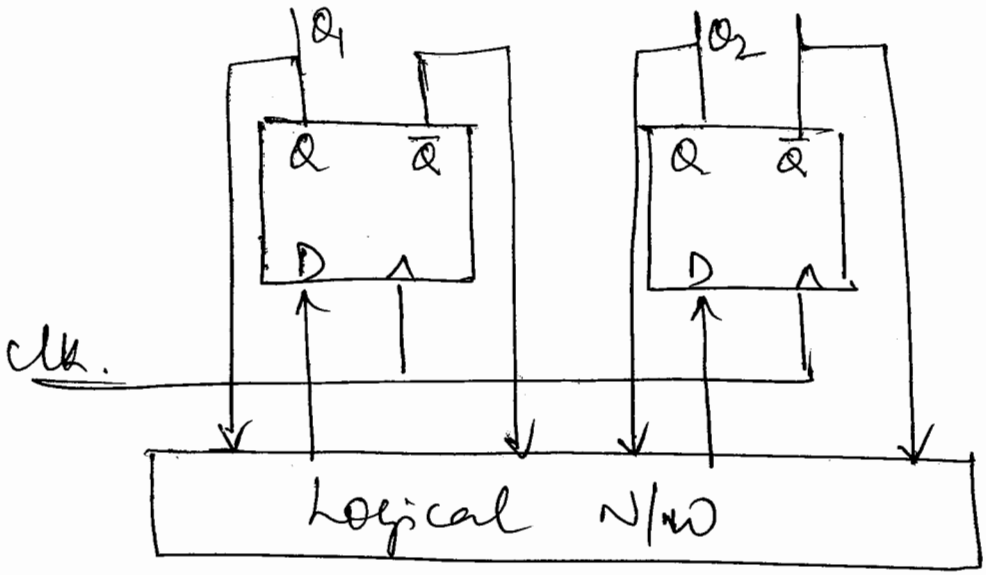


fig 4.13

Explain Mod-4 using D-Flipflop.



Counting Sequence [0, 1, 2, 3]

Q_1	Q_2
0	0
0	1
1	0
1	1

Application Table

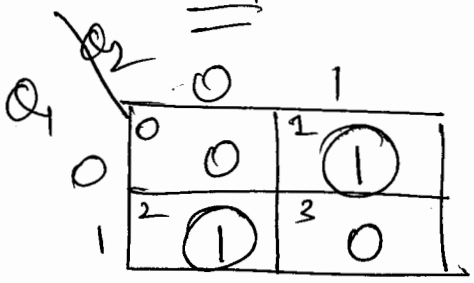
Q	Q^+	D
0	0	0
0	1	1
1	0	0
1	1	1

Excitation table of Mod-4 Synchronous counter using D Flip-Flop.

Present State	Next State	Flip flop inputs
Q_1 Q_2	Q_1 Q_2	D_1 D_2
0 0	0 1	0 1
0 1	1 0	1 0
1 0	1 1	1 1
1 1	0 0	0 0

Boolean Expressions for Logic N/w. of Mod4 counter.

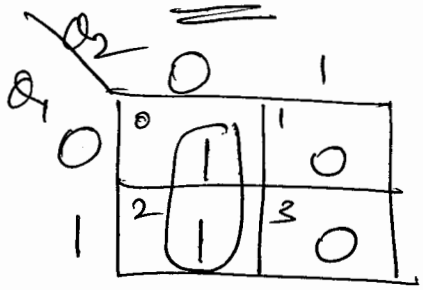
D_1



$D_1 = \bar{Q}_1 Q_2 + Q_1 \bar{Q}_2$

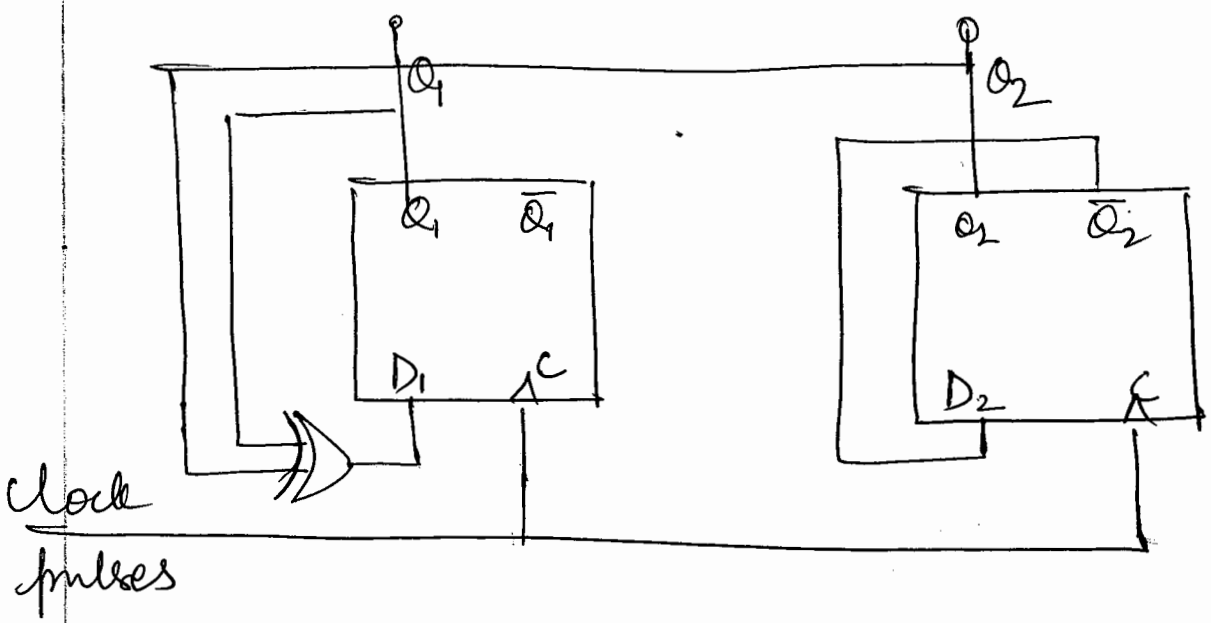
$= Q_1 \oplus Q_2$

D_2



$D_2 = \bar{Q}_2$.

Logic diagram of Mod4 Synchronous counter.



(3) Design a Synchronous Mod-6 counter using T- Flipflop.

(4) Design Mod-6 counter using SR flip flop. counting sequence 000, 001, 011, 100, 101, 111, 000. (0, 1, 3, 4, 5, 7, 0--)

Excitation table of SR flip flop

Q	Q^+	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

Excitation table of Mod-6 counter using SR flip flop

Present state $Q_1Q_2Q_3$	Next state $Q_1Q_2Q_3$	Flip flop ips S_1R_1	S_2R_2	S_3R_3
0 → 000	001	0 X	0 X	1 0
1 → 001	011	0 X	1 0	X 0
3 → 011	100	1 0	0 1	0 1
4 → 100	101	X 0	0 X	1 0
5 → 101	111	X 0	1 0	X 0
7 → 111	000	0 1	0 1	0 1

K-Map for S1, R1, S2, R2 & S3, R3

S_1

Q_1 \ Q_2Q_3	00	01	11	10
0	0	0	1	X
1	X	X	0	X

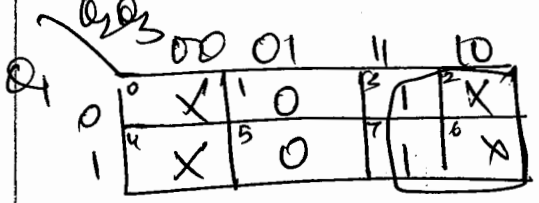
$S_1 = \bar{Q}_1 Q_2$

R_1

Q_1 \ Q_2Q_3	00	01	11	10
0	X	X	0	X
1	0	0	1	X

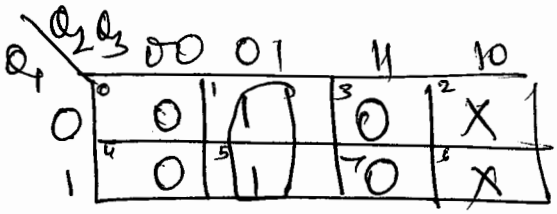
$R_1 = Q_1 Q_2$

R_2



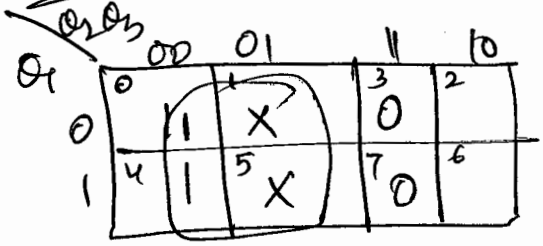
$R_2 = Q_2$

S_2



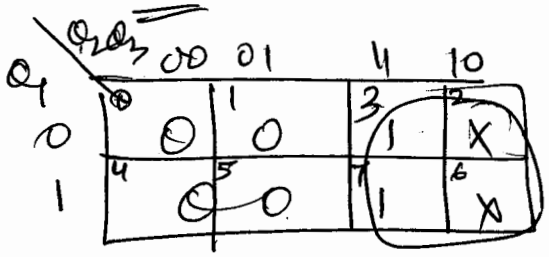
$S_2 = \overline{Q_2}Q_3$

S_3



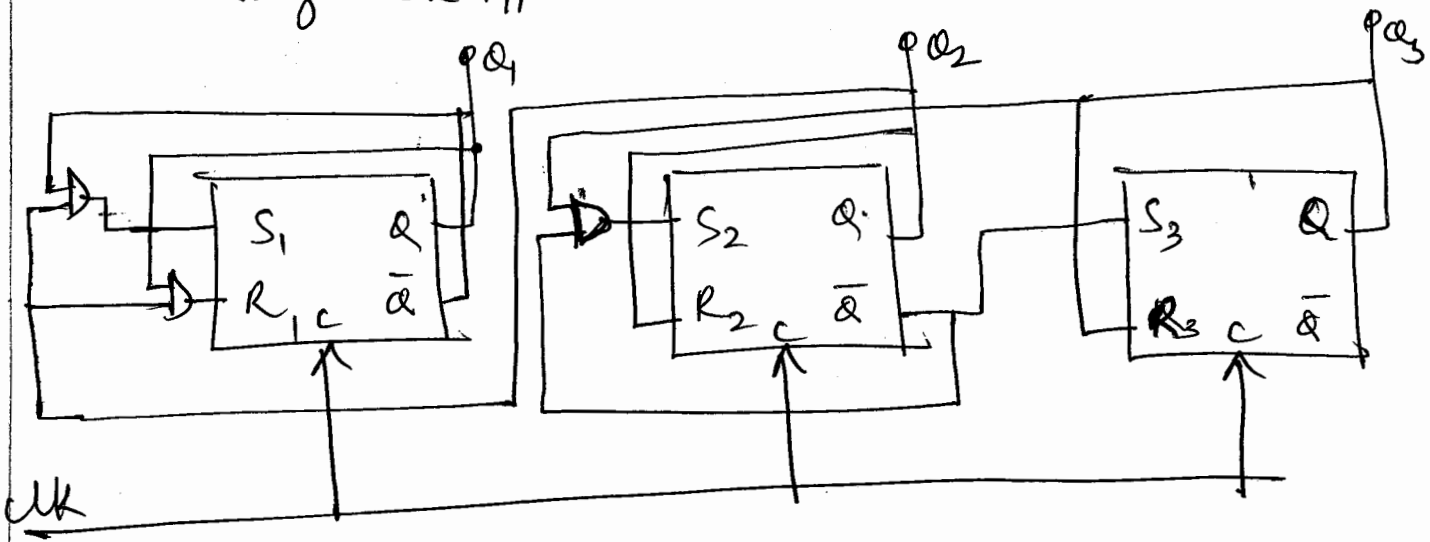
$S_3 = \overline{Q_2}$

R_3

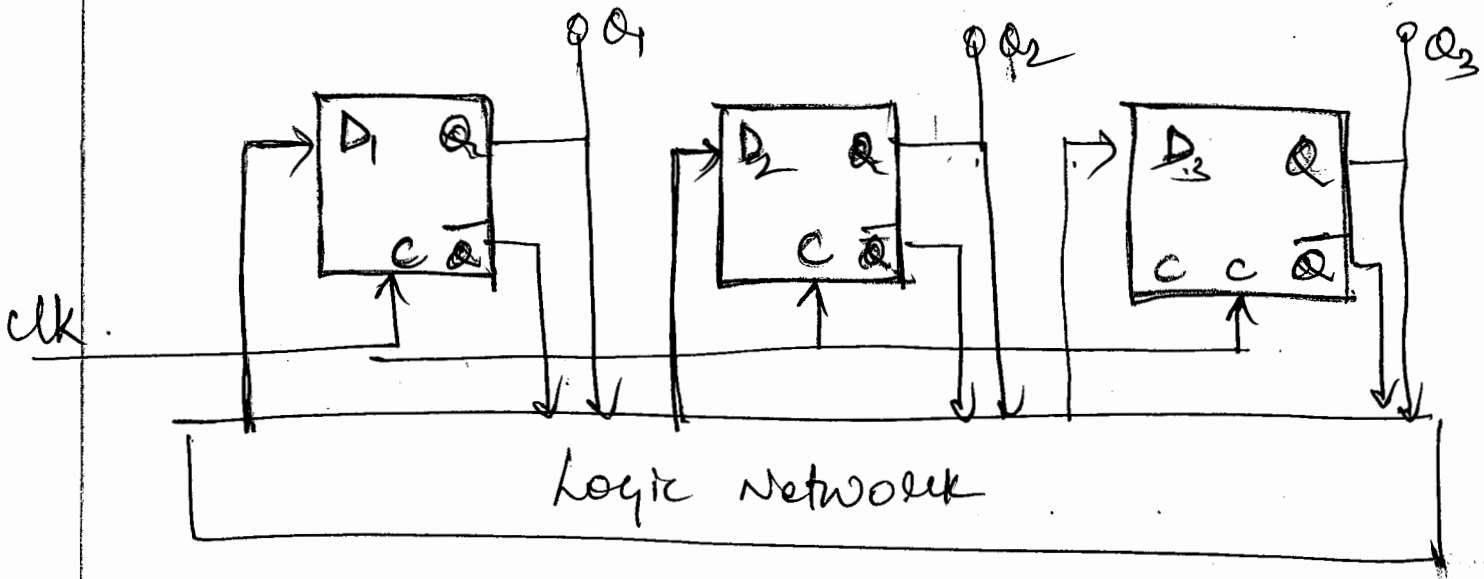


$R_3 = Q_3$.

Logic diagram for synchronous counter using SR F/F.



(5) Mod-6 counter using D flip flop to generate the sequence (0, 2, 3, 6, 5, 1, 0 1 0)



Excitation table of D F/F

Q	Q+	D
0	0	0
0	1	1
1	0	0
1	1	1

Excitation table of Mod6 counter

	Present State	Next State	F/F Inputs		
	$Q_1Q_2Q_3$	$Q_1Q_2Q_3$	D_1	D_2	D_3
0	0 0 0	0 1 0	0	1	0
2	0 1 0	0 1 1	0	1	1
3	0 1 1	1 1 0	1	1	0
6	1 1 0	1 0 1	1	0	1
5	1 0 1	0 0 1	0	0	1
1	0 0 1	0 0 0	0	0	0
Repeat 0					

K-Map for D_1, D_2, D_3

D_1

Q_1 \ Q_2Q_3	00	01	11	10
0	0	0	0	0
1	X	0	X	1

$D_1 = Q_2Q_3$

D_2

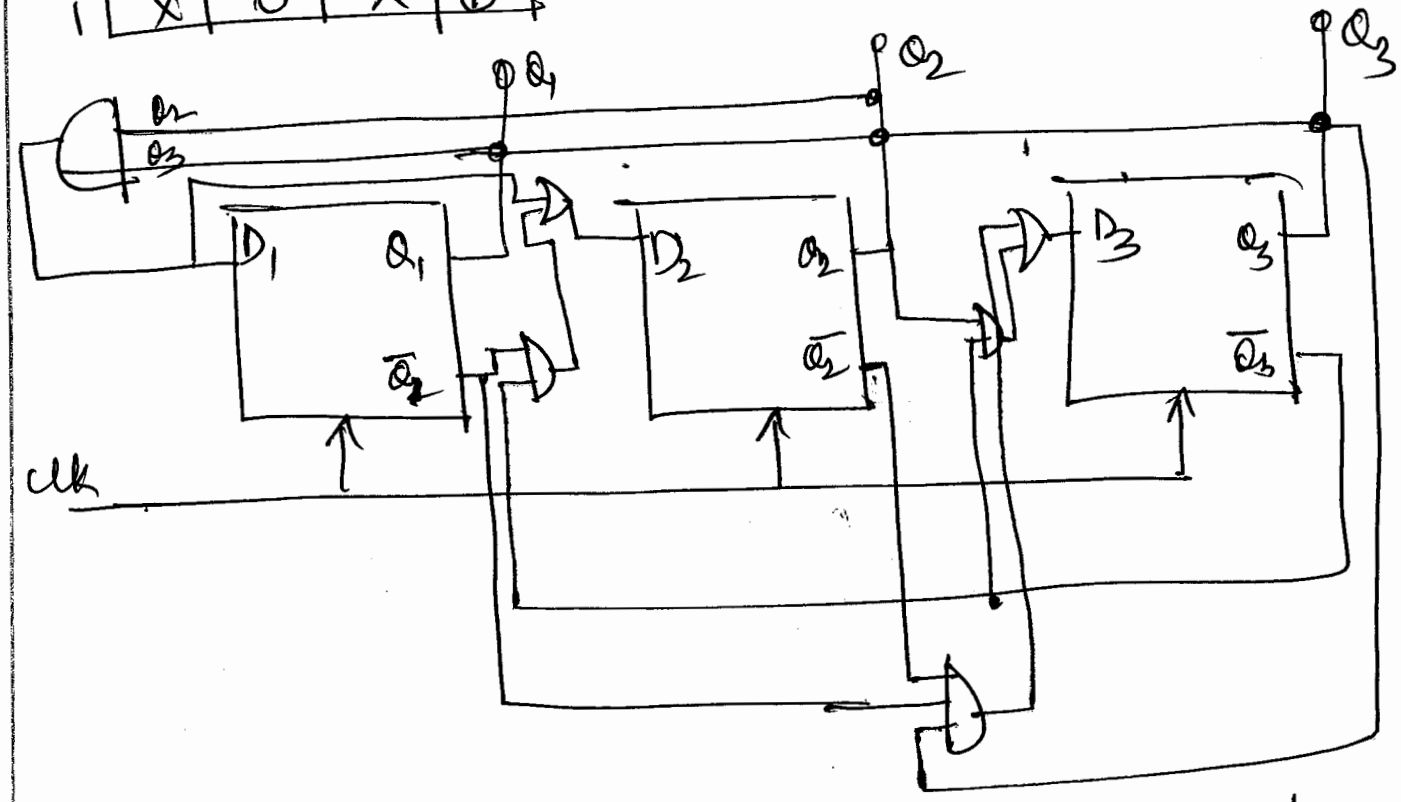
Q_1 \ Q_2Q_3	00	01	11	10
0	0	0	1	1
1	X	0	X	0

$D_2 = Q_2Q_3 + \bar{Q}_1\bar{Q}_3$

D_3

Q_1 \ Q_2Q_3	00	01	11	10
0	0	1	0	1
1	X	0	X	1

$D_3 = \bar{Q}_1\bar{Q}_2Q_3 + \bar{Q}_3Q_2$



Logic diagram for Mod-6 Synchronous Counter using D F/F.

MODULE – 5
SEQUENTIAL CIRCUIT DESIGN

- Mealy & Moore Models
- State Machine Notation
- Synchronous Sequential circuit Analysis
- Construction of State diagrams
- Counter Design.

DE

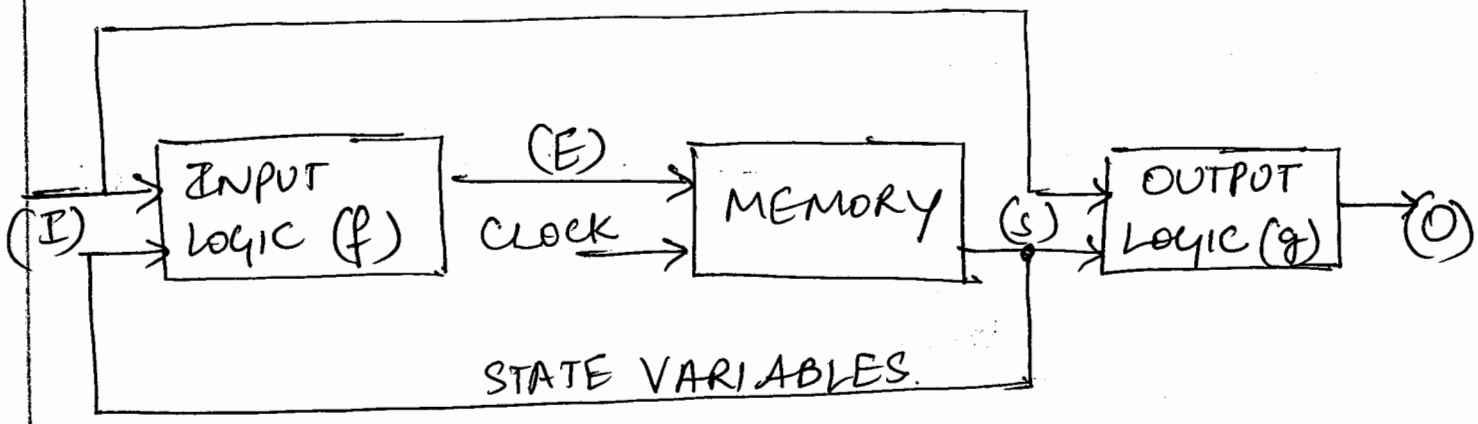
III EC

MODULE 5 - SEQUENTIAL CIRCUIT DESIGN

- Synchronous sequential circuit memory, usually consisting of flip flops, updates circuit states information.
- The two most sequential circuit models are Mealy & Moore models - include combinational and memory submits.
- The logic functions shown in the model diagrams fig(1) & fig(2) translate Inpnt and flip-flop output information.
- The Inpnt logic produces the excitation ips to the flip flops, and the output logic converts Inpnt and flip flop data to satisfy the output variable requirements.
- A Mealy sequential circuit differs from a Moore circuit in the variables used to generate output function.
- In a mealy machine the external ip variables [I] and the state variables (S) are applied to the op logic function(g) to generate output (O).

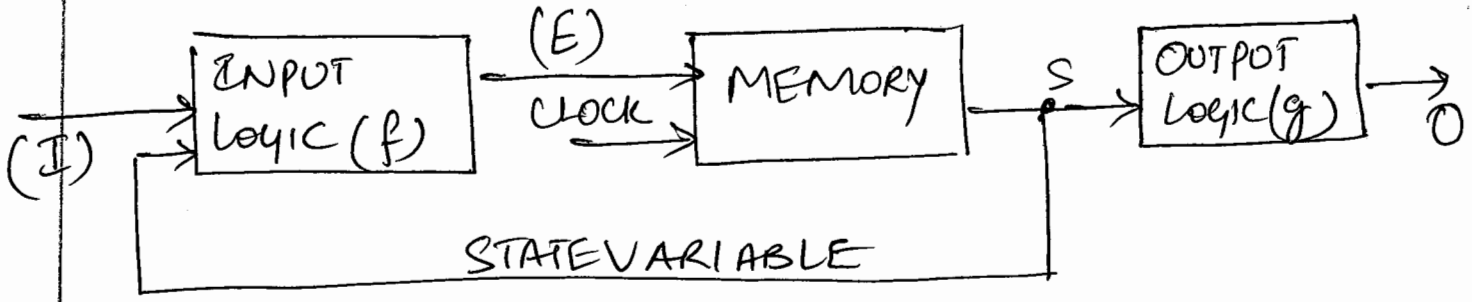
A Moore circuit op is dependent only on the state variables (S).

MEALY SEQUENTIAL CIRCUIT MODEL



(I) → Input variables
(E) → Excitation
(S) → State variables
(g) → output logic
(f) → Input logic
(O) → output variables $O = f(PS, P.Ip)$

MOORE SEQUENTIAL CIRCUIT MODEL



STATE MACHINE NOTATION.

- Several different names are given to Boolean variables in state machine.

1. **INPUT VARIABLE** : All variables that originate outside the sequential Machine are said to be input variables.

2. **OUTPUT VARIABLE** : All variables that exist the sequential Machine are said to be output variables.

3. **STATE VARIABLE**: The output of memory (flip-flops) defines the state of a sequential machine. Decoded state variables produce the output variables.

4. **EXCITATION VARIABLE** Excitation variables are the inputs to memory (flip-flops). The name Excitation is used because the variable excites the memory to change.

5. **STATE**: The state of a sequential machine is defined by the content of the memory. When memory is realized with flip flops, the machine state is defined by the Q outputs.

State variables and states is (related) defined by the expression $2^x = y$

where x = no. of state variables (flipflops)

6. **PRESENT STATE**: The states of all state variables, at some time t, before the next clock edge, represents a condition called present state.

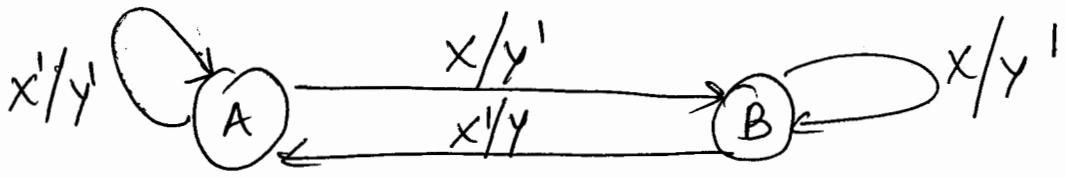
7. **NEXT STATE**: The states of all state variables, at some time, t+1 represents a condition called Next State. The next state of a Sequential Machine is represented by the memory states after a particular clock t.

STATE DIAGRAM.

- A state diagram is a graphical representation of a sequential circuit, where individual states are represented by a circle with identifying symbol located inside.
- Changes from state to state are indicated by directed arcs.
- Input conditions that cause state changes to occur and the resulting op signals are written adjacent to the directed arc.

MEALY STATE NOTATION.

Fig(1) represents the symbolic notation for a one input, one output Mealy sequential circuit.



X ⇒ input variable
Y ⇒ output variable
A & B ⇒ Symbols representing different States.

X/Y ⇒ Input/output.

Case 1: PS = A
If input variable x=0, then the machine remains in state A & Y=0.

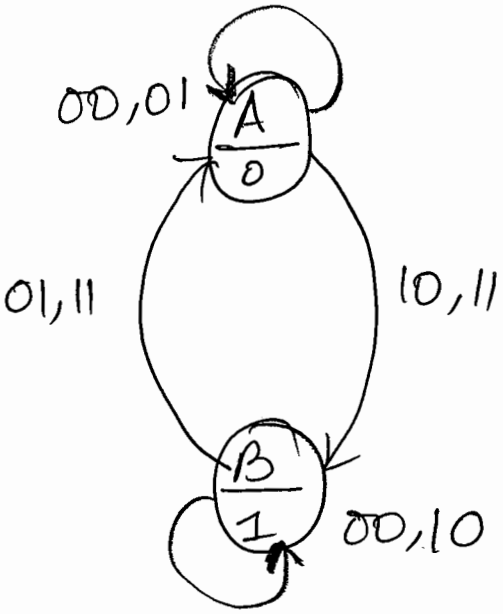
Case 2: If x=1, then a transition to State B is made Y=0.

Case 3: PS = B
If x=0, then the machine remains in State B with y=0.

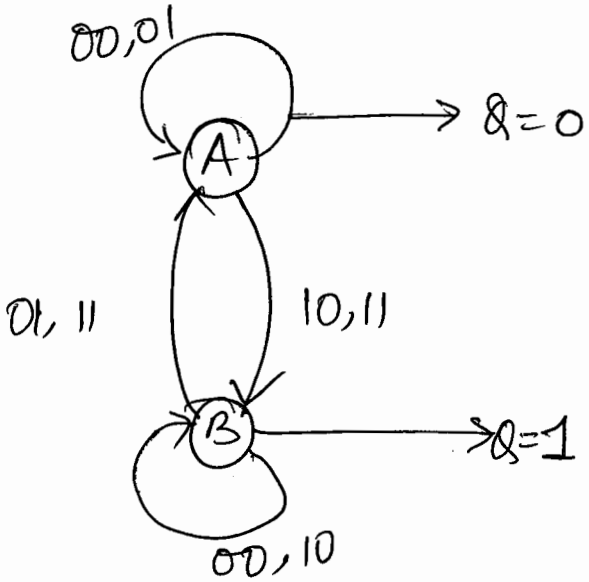
Case 4: PS = B
If x=0, then there is a change of

MOORE CIRCUIT STATE NOTATION - JK F/F.

A JK flip flop can be modeled by a state diagram as shown in fig (2) & fig (3)



fig(2): output variables written under state variable names



fig(3): output variables indicated by arcs.

- Each state in the state diagram represents a state of the flip flop, either set or reset.
- State A represents the case where Q=0 & State B represents the case where Q=1.

J K	Q $\bar{Q}$
0 0	NC
0 1	1 0
1 0	1 0
1 1	$\bar{Q}$ Q

case1: If JK=00 when the machine is in State A, then it remains in State A and Q=0,

Case 2: If PS = A and JK = 10, a transition to State B is made and the op Z changes to 1.

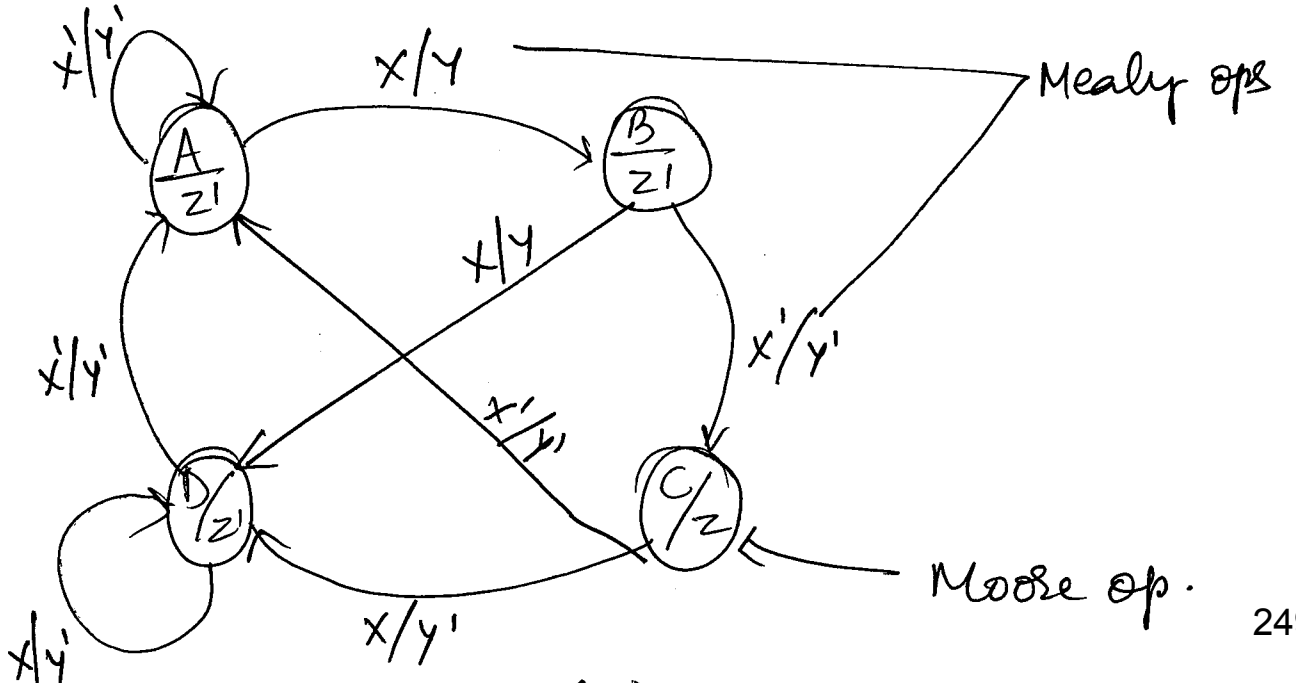
Case 3: If JK = 10 or 11, PS = A, a transition from State A to B is made.

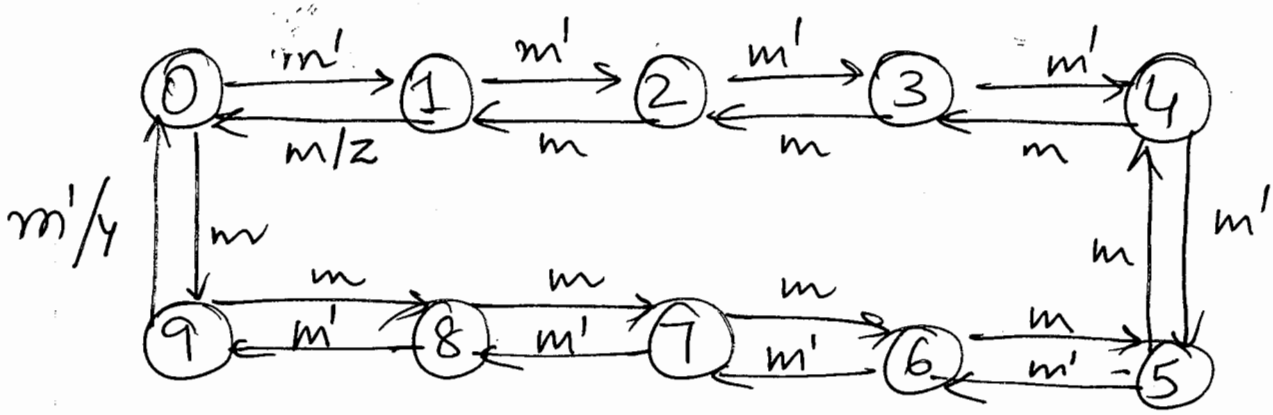
Case 4: If PS = B, the input conditions JK = 01 or JK = 11 cause a transition to State A.

Case 5: If PS = B & JK = 00 or JK = 10, the machine remains in State B.

MIXED MEALY & MOORE STATE NOTATION.

Sequential State machines can be represented by a mix of both Mealy and Moore notation as shown in fig(4).





- The State diagram is a mealy machine because the 2 op variables are dependent on the Mode input M and the present state.

2. SEQUENCE DETECTOR.

Construct a Mealy Machine that will detect a Serial ip sequence of 10110. The detection of the required bit pattern can occur in a longer data string and the correct pattern can overlap with another pattern.

- when the ip pattern has been detected, cause an op Z to be asserted high.

Soln For example, let the input string be

A B C D E C D E C D E

X = 10110110110

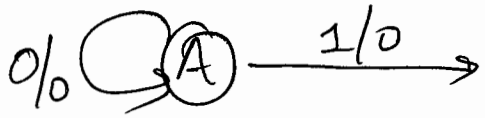
Z = 00001001001

1) Develop the state diagram one state at a time. (A) The LSB to detected is a 1.

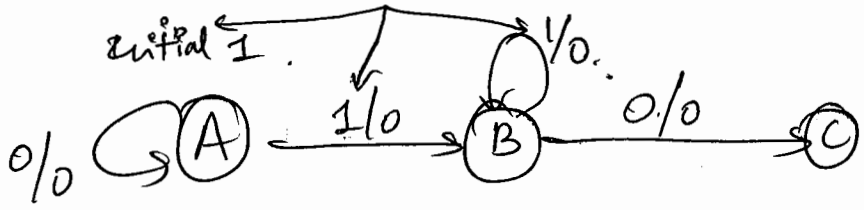
State A is the initial state.

It checks for a 1. If a 1 occurs, the machine changes from State A to B.

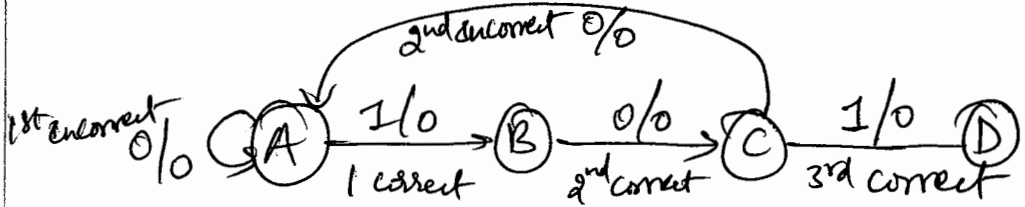
If a 0 occurs, the machine remains in State A - waiting for the first 1.



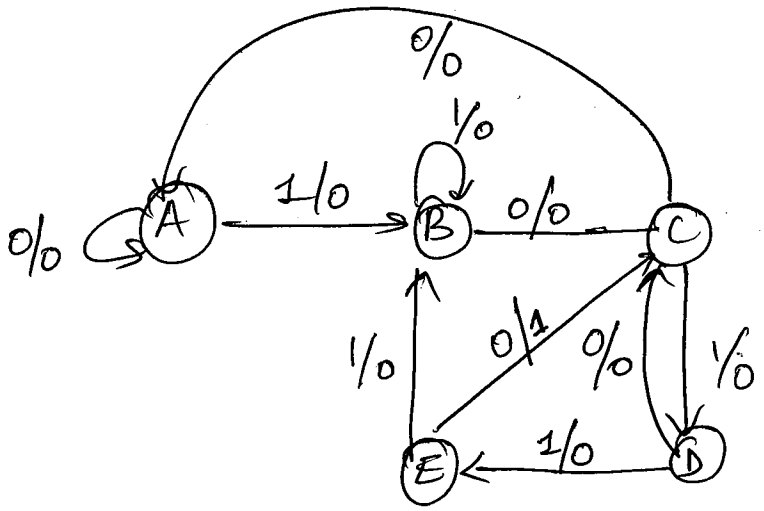
(2) The second bit is 0. If it is detected the machine goes to C otherwise it remains in B.



(3) The third bit is 1. If it detects 1, transition from C to D is made otherwise the machine goes back to State A as 100 string cannot be a part of the String



(4)



If it detects 1 transition from State D to E is made. otherwise it goes back to State C.

(5) The 5th correct bit is 0. If the machine detects 0, transition from State E to D is made.

If 1 is detected, the machine changes from State E to B. indicating that the first 1 of a new string is detected.

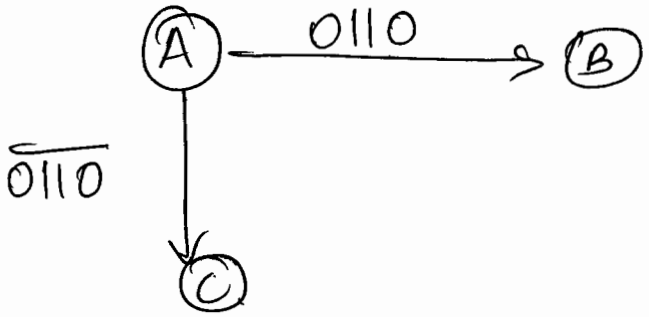
3. SEQUENCE DETECTOR – SERIAL STRING OF 4 BITS.

– Let the Input combination be

0110, 1010, 0100, 1000. (input the left most code first).

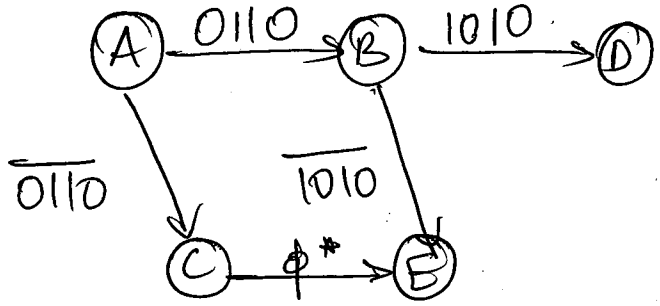
(1) Start in some initial state (A). The first pattern is either correct or incorrect. If it is correct, then state A changes to B.

- If the code is incorrect, change to a new state C.



(2) State B indicates the first correct input. State C indicates an incorrect first code input. The second four input code is entered next. A correct code input causes a transition from B to D. The state changes from B to E after one correct and one incorrect code inputs have been entered.

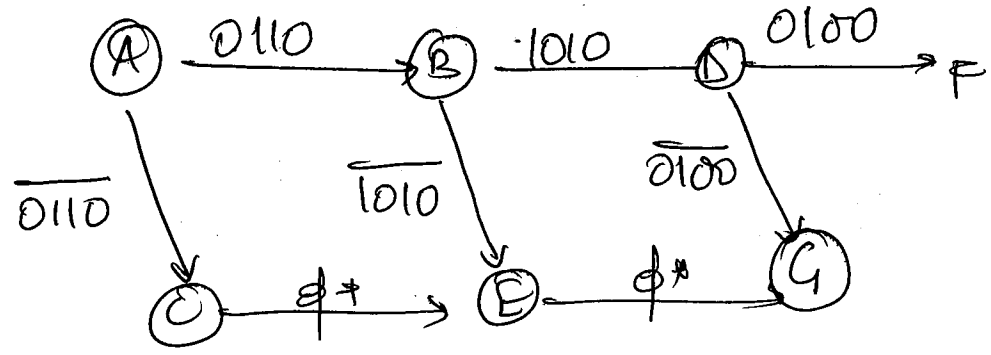
- Two incorrect code inputs leaves the system in state E.



(3) State D indicates that the second correct input code has been entered.

State E indicates a second entry attempt.

- The third correct ip causes a state transition from D to F. If on the third attempt, any input is incorrect the machine changes to State G.

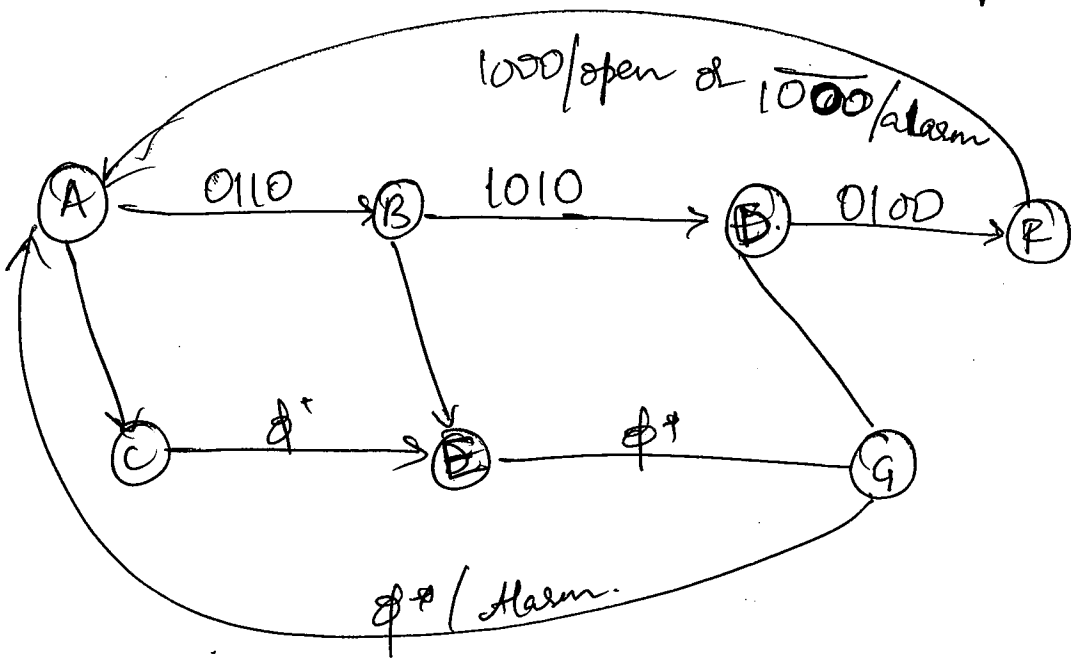


ϕ^* → all input combinations cause a transition.

(4) The fourth correct input causes a state transition from F to A while generating

- An incorrect input attempt after 3 correct entries also causes an F to A State transition with alarm asserted.

- The fourth input from G causes a G to A transition and an alarm op.



4. SERIAL Ex-3 to BCD Code converter.

Design a Mealy Machine that converts Serial Ex-3 code (x) into Serial Binary coded decimal (BCD) data.

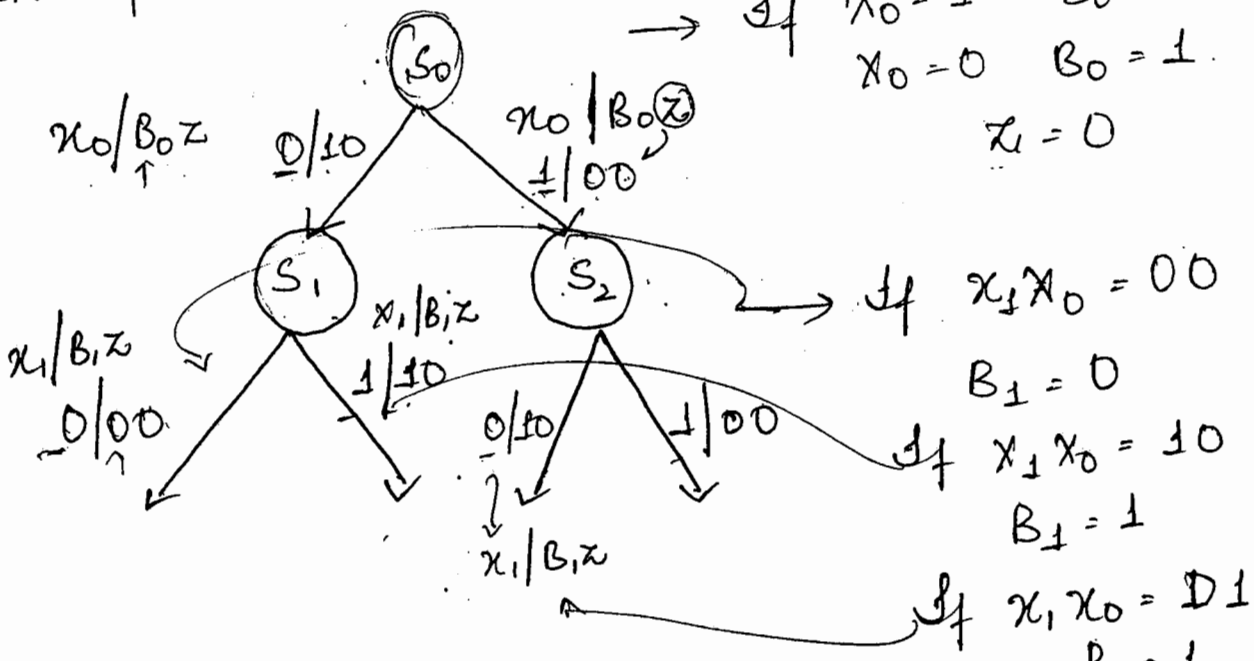
- Both Ex-3 & BCD are 4-bit codes the machine is to return to the begining after 4 inputs.

EXCESS 3- BCD CODE CONVERTER.

X_3	X_2	X_1	X_0	B_3	B_2	B_1	B_0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	1
0	1	0	1	0	0	1	0
0	1	1	0	0	0	1	1
0	1	1	1	0	1	0	0
1	0	0	0	0	1	0	1
1	0	0	1	0	1	1	0
1	0	1	0	0	1	1	1
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	1

Note: Let z be an output, that = 1 if any non X-3 input sequence occurs.

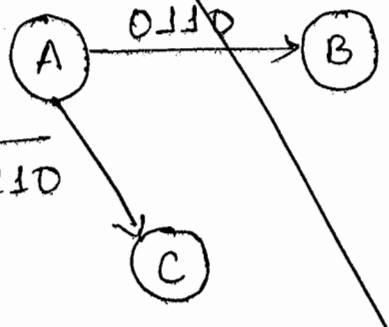
→ Start from an initial state S_0



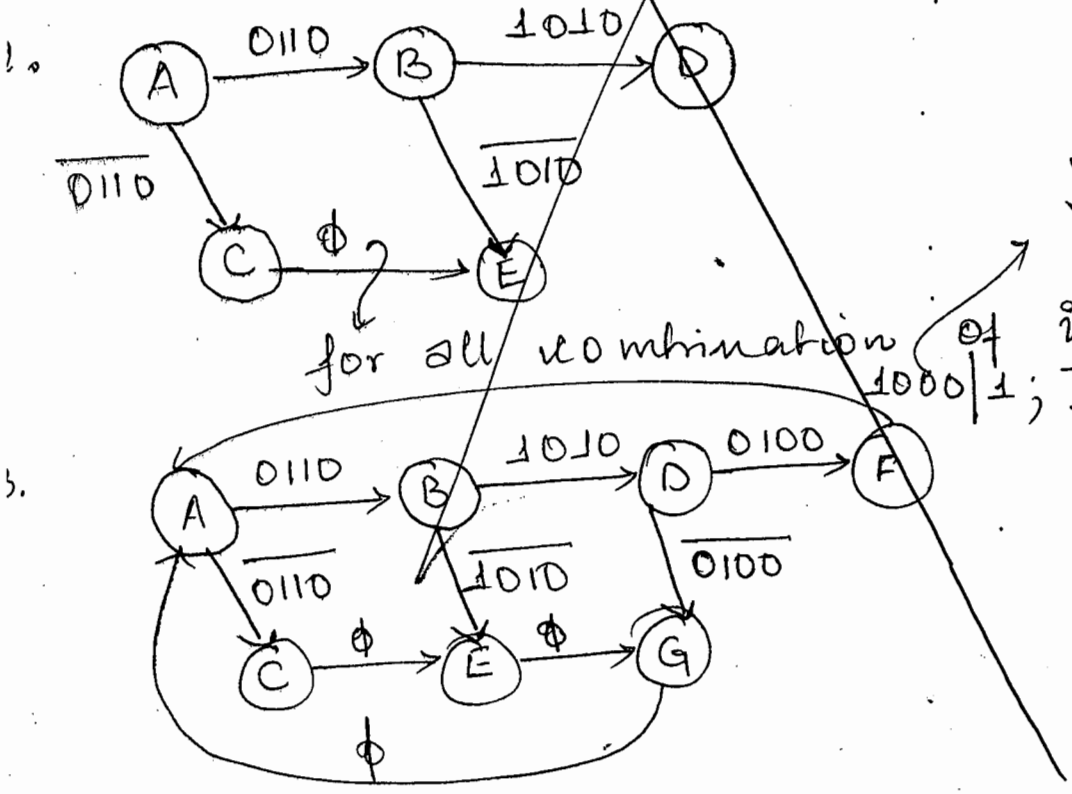
→ Sequence detector to detect a serial string of four inputs.

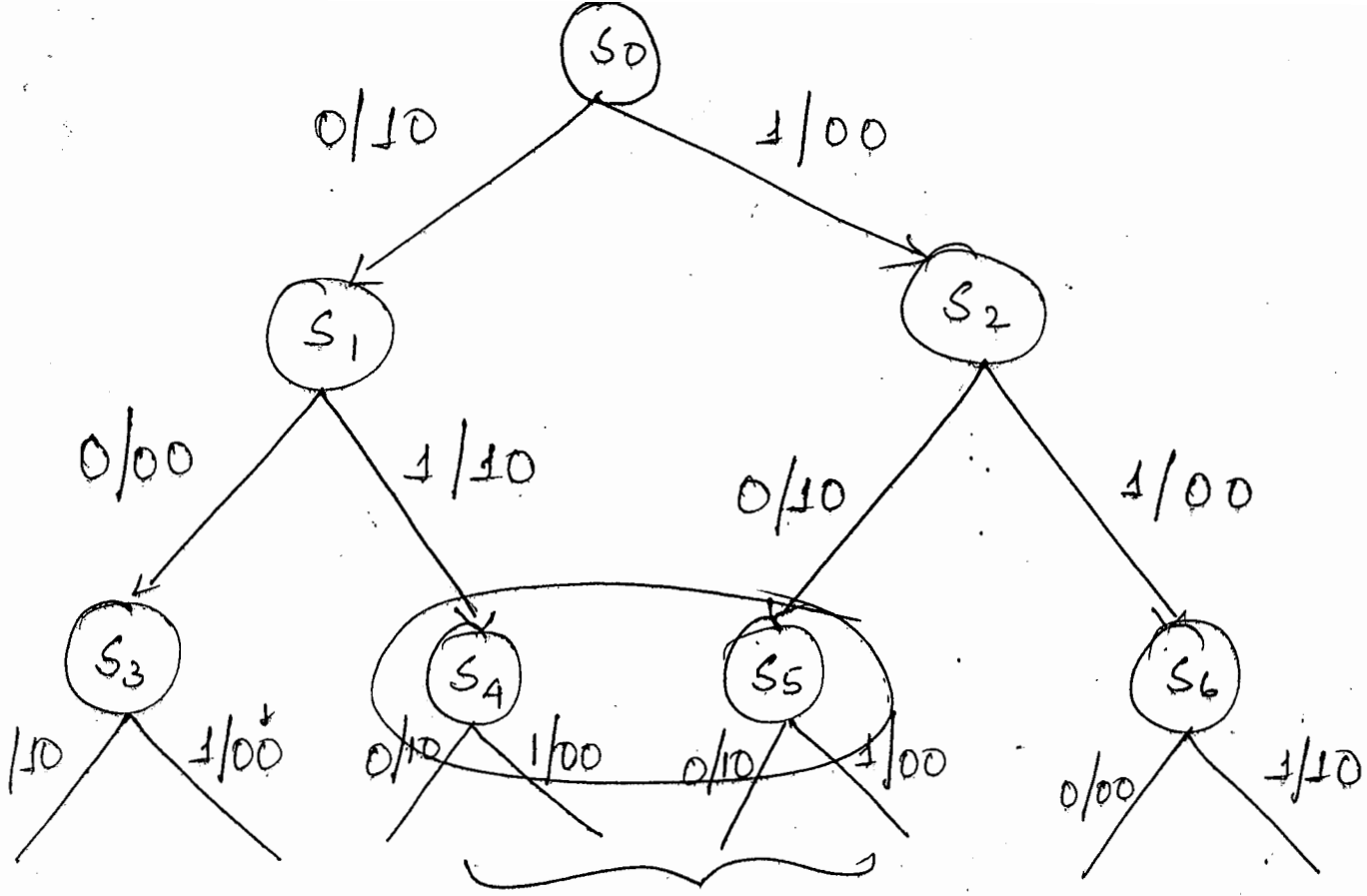
Let the input combination be 0110, 1010, 0100, 1000

1. Start from some initial state A.

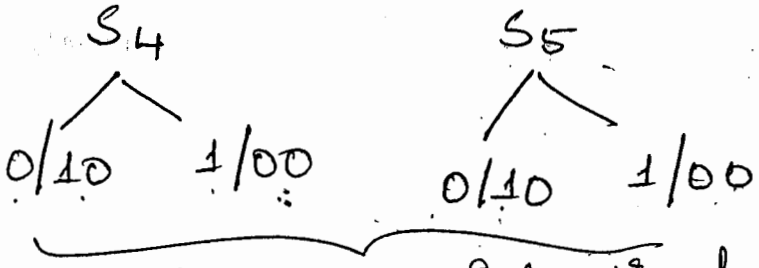


for all other combinations, state A changes to state C. (is).

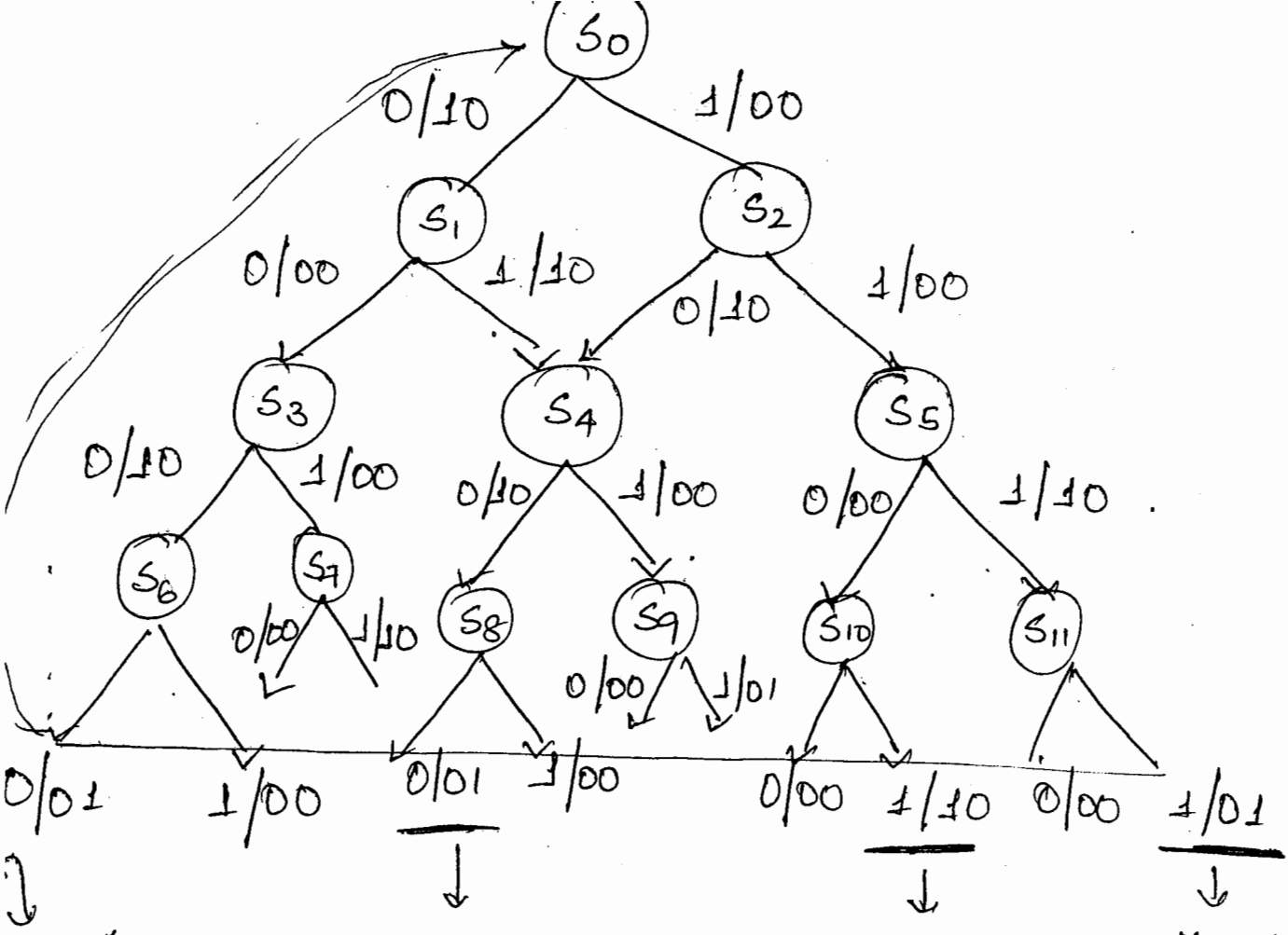




S_4 & S_5 have two o/p transitions



Both are identical, ∴ remove either S_4 or S_5



B_3 Z

$X_3X_2X_1X_0$
$= 0000$
invalid
X_3 code
$\therefore Z = 1$.

$X_3X_2X_1X_0$
$\rightarrow 0001$
invalid

$X_3X_2X_1X_0$
$= 1111$
invalid.

MOD 8 SYNCHRONOUS COUNTER DESIGN

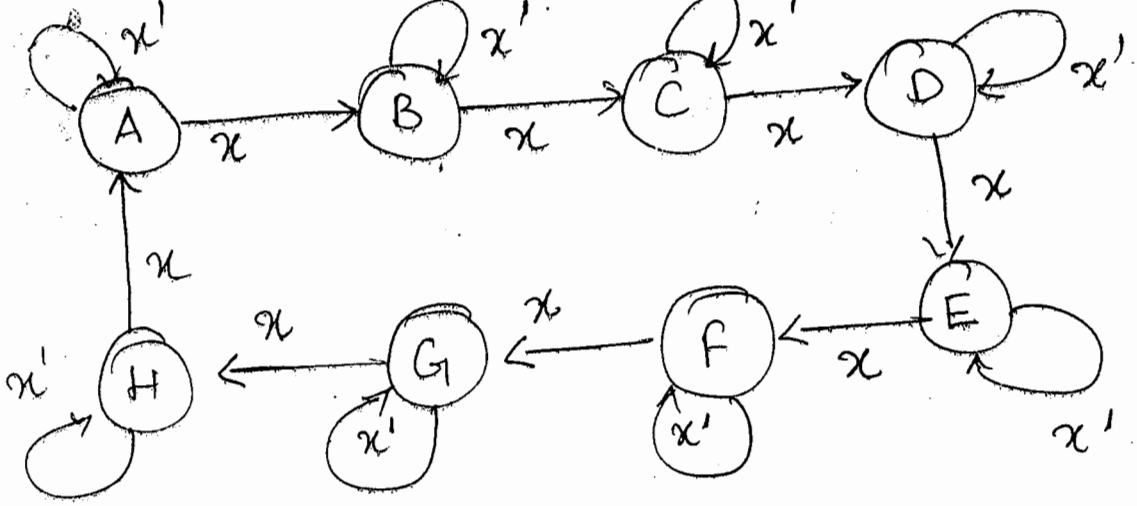
→ UP COUNTER USING JK-FF.

→ Let x be clock enable.

If $x = 1$ → up count

$x = 0$ → remain in same state.

5.23



PS	NS	
	$x=0$	$x=1$
A	A	B
B	B	C
C	C	D
D	D	E
E	E	F
F	F	G
G	G	H
H	H	A

→ State table

PS			NS					
			$x=0$			$x=1$		
f_2	f_1	f_0	f_2	f_1	f_0	f_2	f_1	f_0
0	0	0	0	0	0	0	0	1
0	0	1	0	0	1	0	1	0
0	1	0	0	1	0	0	1	1
0	1	1	0	1	1	1	0	0
1	0	0	1	0	0	1	0	1
1	0	1	1	0	1	1	1	0
1	1	0	1	1	0	1	1	1
1	1	1	1	1	1	0	0	0

PS			N.S $x=0$						N.S $x=1$					
F_2	F_1	F_0	J_2	K_2	J_1	K_1	J_0	K_0	J_2	K_2	J_1	K_1	J_0	K_0
0	0	0	0	x	0	x	0	x	0	x	0	x	1	x
0	0	1	0	x	0	x	x	0	0	x	1	x	x	1
0	1	0	0	x	0	x	0	x	0	x	x	0	1	x
0	1	1	0	x	x	0	x	0	1	x	x	1	x	1
1	0	0	x	0	0	x	0	x	x	0	0	x	1	x
1	0	1	x	0	0	x	x	0	x	0	1	x	x	1
1	1	0	x	0	x	0	0	x	x	0	x	0	1	x
1	1	1	x	0	x	0	x	0	x	1	x	1	x	1

Solving for J & K using K-maps.

$J_2 = F_2 F_1 x$

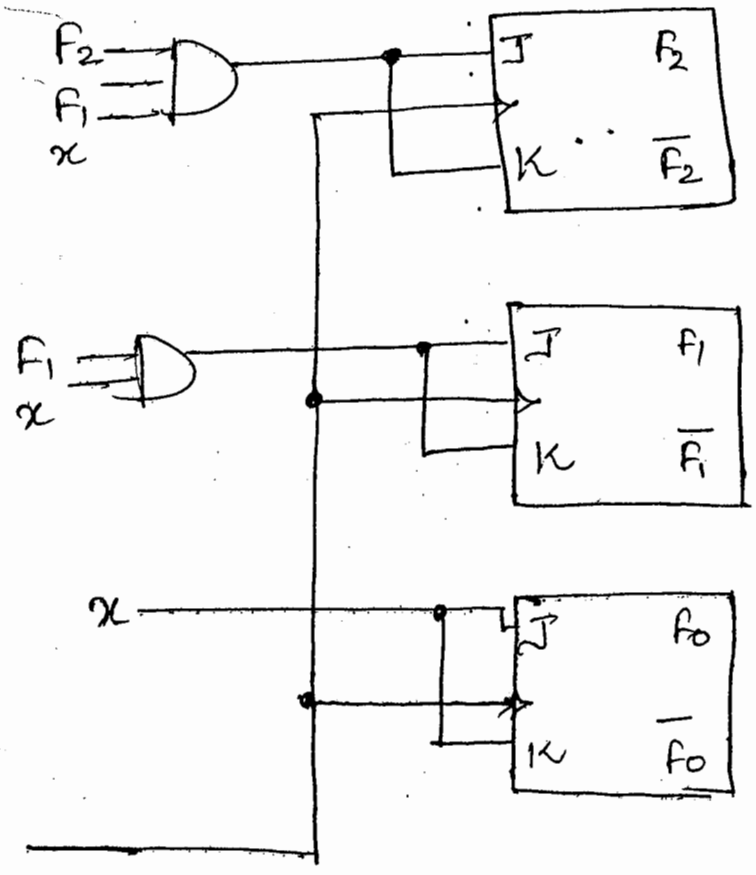
$K_2 = F_2 F_1 x$

$J_1 = F_1 x$

$K_1 = F_1 x$

$J_0 = x$

$K_0 = x$.



14.1 Design of a Sequence Detector

To illustrate the design of a clocked Mealy sequential circuit, we will design a sequence detector. The circuit has the form shown in Figure 14-1.

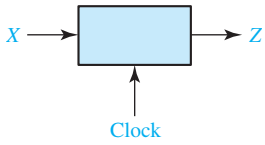


FIGURE 14-1
Sequence Detector to be Designed

The circuit will examine a string of 0's and 1's applied to the X input and generate an output $Z = 1$ only when a prescribed input sequence occurs. It will be assumed that the input X can only change between clock pulses. Specifically, we will design the circuit so that any input sequence ending in 101 will produce an output $Z = 1$ coincident with the last 1. The circuit does not reset when a 1 output occurs. A typical input sequence and the corresponding output sequence are

$$\begin{array}{lcccccccccccccccc} X = & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ Z = & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ (\text{time:} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15) \end{array} \tag{14-1}$$

Initially, we do not know how many flip-flops will be required, so we will designate the circuit states as S_0, S_1, etc., and later assign flip-flop states to correspond to the circuit states. We will construct a state graph to show the sequence of states and outputs which occur in response to different inputs. Initially, we will start the circuit in a reset state designated S_0. If a 0 input is received, the circuit can stay in S_0 because the input sequence we are looking for does not start with 0. However, if a 1 is received, the circuit must go to a new state (S_1) to "remember" that the first input in the desired sequence has been received (Figure 14-2). The labels on the graph are of the form X/Z, where the symbol before the slash is the input and the symbol after the slash is the corresponding output.

When in state S_1, if we receive a 0, the circuit must change to a new state (S_2) to remember that the first two inputs of the desired sequence (10) have been received. If a 1 is received in state S_2, the desired input sequence (101) is complete and the output should be l. The question arises whether the circuit should then go to a new state or back to S_0 or S_1. Because the circuit is not supposed to reset when an output occurs, we cannot go back to S_0. However, because the last 1 in a sequence can also be the first 1 in a new sequence, we can return to S_1, as indicated in Figure 14-3.

FIGURE 14-2

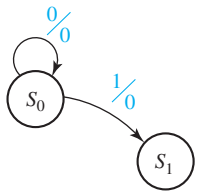
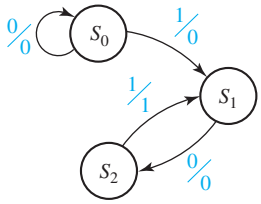
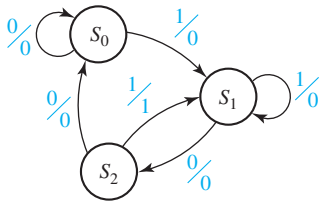


FIGURE 14-3



The graph of Figure 14-3 is still incomplete. If a 1 input occurs when in state S_1, we can stay in S_1 because the sequence is simply restarted. If a 0 input occurs in state S_2, we have received two 0's in a row and must reset the circuit to state S_0 because 00 is not part of the desired input sequence, and going to one of the other states could lead to an incorrect output. The final state graph is given in Figure 14-4. Note that for a single input variable each state must have two exit lines (one for each value of the input variable) but may have any number of entry lines, depending on the circuit specifications.

FIGURE 14-4
Mealy State Graph for Sequence Detector



State S_0 is the starting state, state S_1 indicates that a sequence ending in 1 has been received, and state S_2 indicates that a sequence ending in 10 has been received. An alternative way to start the solution would be to first define states in this manner and then construct the state graph. Converting the state graph to a state table yields Table 14-1. For example, the arc from S_2 to S_1 is labeled 1/1. This means that when the present state is S_2 and $X = 1$, the present output is 1. This 1 output is present as soon as X becomes 1, that is, *before* the state change occurs. Therefore, the 1 is placed in the S_2 row of the table.

TABLE 14-1

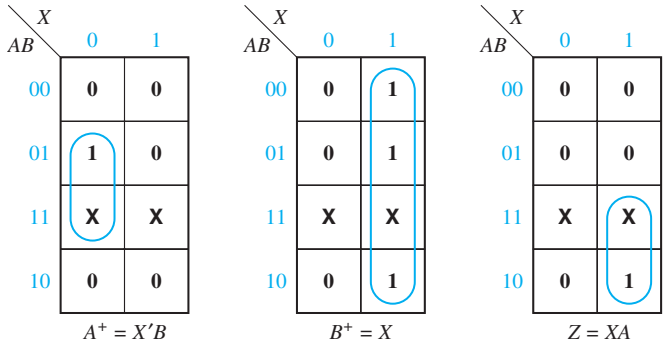
Present State	Next State $X = 0$	Next State $X = 1$	Present Output $X = 0$	Present Output $X = 1$
S_0	S_0	S_1	0	0
S_1	S_2	S_1	0	0
S_2	S_0	S_1	0	1

At this point, we are ready to design a circuit which has the behavior described by the state table. Because one flip-flop can have only two states, two flip-flops are needed to represent the three states. Designate the two flip-flops as A and B. Let flip-flop states $A = 0$ and $B = 0$ correspond to circuit state S_0; A = 0 and $B = 1$ correspond to S_1; and $A = 1$ and $B = 0$ correspond to circuit state S_2. Each circuit state is then represented by a unique combination of flip-flop states. Substituting the flip-flop states for S_0, S_1 and S_2 in the state table yields the transition table (Table 14-2).

TABLE 14-2

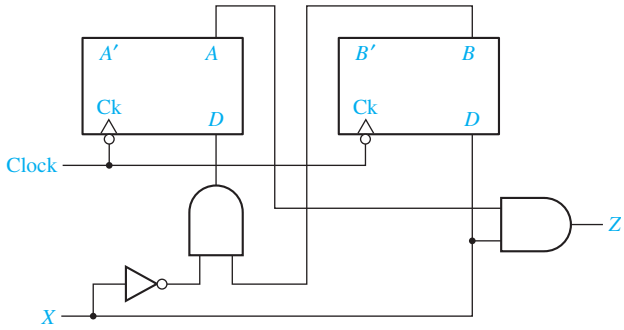
	A^+B^+		Z	
AB	$X = 0$	$X = 1$	$X = 0$	$X = 1$
00	00	01	0	0
01	10	01	0	0
10	00	01	0	1

From this table, we can plot the next-state maps for the flip-flops and the map for the output function Z:



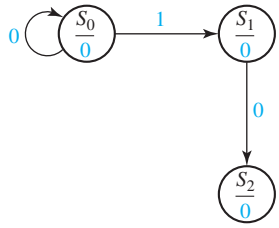
The flip-flop inputs are then derived from the next-state maps using the same method that was used for counters (Section 12.4). If D flip-flops are used, $D_A = A^+ = X'B$ and $D_B = B^+ = X$, which leads to the circuit shown in Figure 14-5. Initially, we will reset both flip-flops to the 0 state. By tracing signals through the circuit, you can verify that an output $Z = 1$ will occur when an input sequence ending in 101 occurs. To avoid reading false outputs, always read the value of Z after the input has changed and before the active clock edge.

FIGURE 14-5

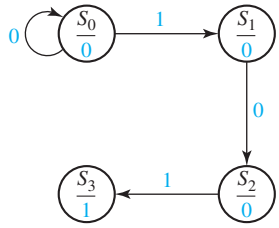


The procedure for finding the state graph for a Moore machine is similar to that used for a Mealy machine, except that the output is written with the state instead of with the transition between states. We will rework the previous example as a Moore machine to illustrate this procedure. The circuit should produce an output of 1 only if an input sequence ending in 101 has occurred. The design is similar to that for the

Mealy machine up until the input sequence 10 has occurred, except that 0 output is associated with states S_0, S_1, and S_2:

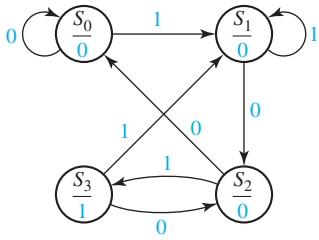


Now, when a 1 input occurs to complete the 101 sequence, the output must become 1; therefore, we cannot go back to state S_1 and must create a new state S_3 with a 1 output:



We now complete the graph, as shown in Figure 14-6. Note the sequence 100 resets the circuit to S_0. A sequence 1010 takes the circuit back to S_2 because another 1 input should cause Z to become 1 again.

FIGURE 14-6
Moore State Graph for Sequence Detector



The state table corresponding to the circuit is given by Table 14-3. Note that there is a single column for the output because the output is determined by the present state and does not depend on X. Note that in this example the Moore machine requires one more state than the Mealy machine which detects the same input sequence.

TABLE 14-3

Present State	Next State $X = 0$	Next State $X = 1$	Present Output(Z)
S_0	S_0	S_1	0
S_1	S_2	S_1	0
S_2	S_0	S_3	0
S_3	S_2	S_1	1

Because there are four states, two flip-flops are required to realize the circuit. Using the state assignment $AB = 00$ for S_0, $AB = 01$ for S_1, $AB = 11$ for S_2, and $AB = 10$ for S_3, the following transition table for the flip-flops results (Table 14-4):

TABLE 14-4

	A^+B^+		
AB	$X = 0$	$X = 1$	Z
00	00	01	0
01	11	01	0
11	00	10	0
10	11	01	1

The output function is $Z = AB'$. Note that Z depends only on the flip-flop states and is independent of X, while for the corresponding Mealy machine, Z was a function of X. The derivation of the flip-flop input equations is straightforward and will not be given here.

14.2 More Complex Design Problems

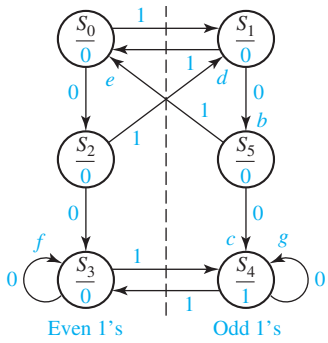
In this section we will derive a state graph for a sequential circuit of somewhat greater complexity than the previous examples. The circuit to be designed again has the form shown in Figure 14-1. The output Z should be 1 if the input sequence ends in either 010 or 1001, and Z should be 0 otherwise. Before attempting to draw the state graph, we will work out some typical input-output sequences to make sure that we have a clear understanding of the problem statement. We will determine the desired output sequence for the following input sequence:

$$X = 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ 1\ 1\ 0$$
$$\qquad\quad\ \uparrow\ \ \uparrow\ \ \uparrow\uparrow\ \qquad\ \uparrow\ \ \uparrow$$
$$\qquad\quad\ a\ \ b\ \ c\ d\ \qquad\ e\ \ f$$
$$Z = 0\ 0\ 0\ 1\ 0\ 1\ 0\ 1\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0$$

At point a, the input sequence ends in 010, one of the sequences for which we are looking, so the output is $Z = 1$. At point b, the input again ends in 010, so $Z = 1$. Note that overlapping sequences are allowed because the problem statement does not say anything about resetting the circuit when a 1 output occurs. At point c, the input sequence ends in 1001, so Z is again 1. Why do we have a 1 output at points d, e, and f? This is just one of many input sequences. A state machine that gives the correct output for this sequence will not necessarily give the correct output for all other sequences.

We will start construction of the state graph by working with the two sequences which lead to a 1 output. Then, we will later add arrows and states as required to make sure that the output is correct for other cases. We start off with a reset state S_0 which corresponds to having received no inputs. Whenever an input is received that corresponds to part of one of the sequences for which we are looking, the circuit should go to a new state to "remember" having received this input. Figure 14-7 shows a partial state graph which gives a 1 output for the sequence 010. In this graph S_1 corresponds

FIGURE 14-12



State	Input Sequences
S_0	Reset or even 1's
S_1	Odd 1's
S_2	Even 1's and ends in 0
S_3	Even 1's and 00 has occurred
S_4	Odd 1's and 00 has occurred
S_5	Odd 1's and ends in 0

0's can be ignored. Therefore, we can stay in S_3 (arrow f). Similarly, extra 0 inputs can be ignored in S_4 (arrow g). This completes the Moore state diagram, and we should go back and verify that the correct output sequence is obtained for various input sequences.

14.3 Guidelines for Construction of State Graphs

Although there is no one specific procedure which can be used to derive state graphs or tables for every problem, the following guidelines should prove helpful:

1. First, construct some sample input and output sequences to make sure that you understand the problem statement.
2. Determine under what conditions, if any, the circuit should reset to its initial state.
3. If only one or two sequences lead to a nonzero output, a good way to start is to construct a partial state graph for those sequences.
4. Another way to get started is to determine what sequences or groups of sequences must be remembered by the circuit and set up states accordingly.
5. Each time you add an arrow to the state graph, determine whether it can go to one of the previously defined states or whether a new state must be added.
6. Check your graph to make sure there is one and only one path leaving each state for each combination of values of the input variables.
7. When your graph is complete, test it by applying the input sequences formulated in part 1 and making sure the output sequences are correct.

Several examples of deriving state graphs or tables follow.

Example 1

A sequential circuit has one input (X) and one output (Z). The circuit examines groups of four consecutive inputs and produces an output $Z = 1$ if the input sequence 0101 or 1001 occurs. The circuit resets after every four inputs. Find the Mealy state graph.

Solution

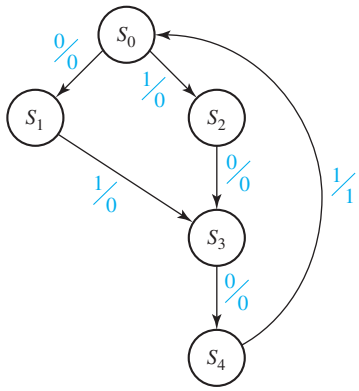
A typical sequence of inputs and outputs is

$X = 0101 \mid 0010 \mid 1001 \mid 0100$
$Z = 0001 \mid 0000 \mid 0001 \mid 0000$

The vertical bars indicate the points at which the circuit resets to the initial state. Note that an input sequence of either 01 or 10 followed by 01 will produce an output of $Z = 1$. Therefore, the circuit can go to the same state if either 01 or 10 is received. The partial state graph for the two sequences leading to a 1 output is shown in Figure 14-13.

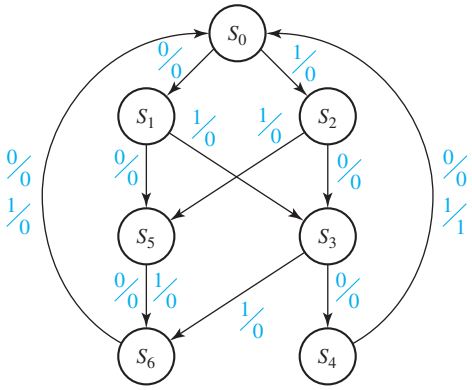
Note that the circuit resets to S_0 when the fourth input is received. Next, we add arrows and labels to the graph to take care of sequences which do not give a 1 output, as shown in Figure 14-14.

FIGURE 14-13
Partial State Graph for Example 1



State	Sequence Received
S_0	Reset
S_1	0
S_2	1
S_3	01 or 10
S_4	010 or 100

FIGURE 14-14
Complete State Graph for Example 1



State	Sequence Received
S_0	Reset
S_1	0
S_2	1
S_3	01 or 10
S_4	010 or 100
S_5	Two inputs received, no 1 output is possible
S_6	Three inputs received, no 1 output is possible

The addition of states S_5 and S_6 was necessary so that the circuit would not reset to S_0 before four inputs were received. Note that once a 00 or 11 input sequence has been received (state S_5), no output of 1 is possible until the circuit is reset.

Example 2

A sequential circuit has one input (X) and two outputs (Z_1 and Z_2). An output $Z_1 = 1$ occurs every time the input sequence 100 is completed, provided that the sequence 010 has never occurred. An output $Z_2 = 1$ occurs every time the input sequence 010 is completed. Note that once a $Z_2 = 1$ output has occurred, $Z_1 = 1$ can never occur but not vice versa. Find a Mealy state graph and table.

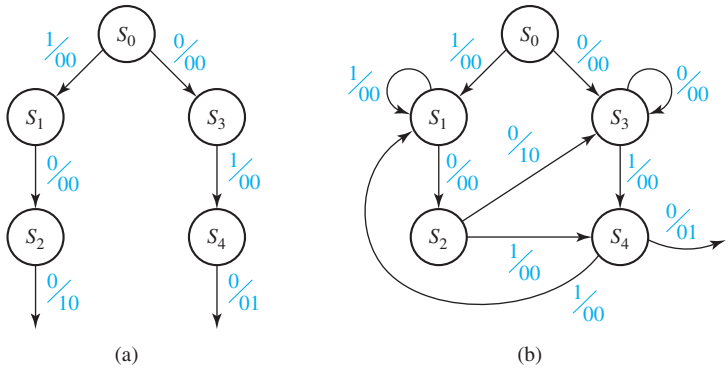
Solution A typical sequence of inputs and outputs is:

$$\begin{array}{lcccccccccc|ccccccccccc} X = & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ Z_1 = & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ Z_2 = & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}$$

Note that the sequence 100 occurs twice before 010 occurs, and $Z_1 = 1$ each time. However, once 010 occurs and $Z_2 = 1$, $Z_1 = 0$ even when 100 occurs again. $Z_2 = 1$ all five times that 010 occurs. Because we were not told to reset the circuit, 01010 means that 010 occurred twice.

We can begin to solve this problem by constructing the part of the state graph which will give the correct outputs for the sequences 100 and 010. Figure 14-15(a) shows this portion of the state graph.

FIGURE 14-15
Partial Graphs for Example 2



An important question to ask at this point is, what does this circuit need to remember to give the correct outputs? The circuit will need to remember how much progress has been made on the sequence 010, so it will know when to output $Z_2 = 1$. The circuit will also need to remember how much progress has been made on the sequence 100 and whether 010 has ever occurred, so it will know when to output $Z_1 = 1$.

Keeping track of what is remembered by each state will help us make the correct state graph. Table 14-5 will help us to do this. State S_0 is the initial state of the circuit, so there is no progress on either sequence, and 010 has never occurred. State S_1 is the state we go to when a 1 is received from S_0, so in state S_1, we have made progress on the sequence 100 by getting a 1. In state S_2, we have made progress on the sequence 100 by getting 10. Similarly, states S_3 and S_4 represent progress of 0 and 01 toward 010. In S_1,

TABLE 14-5
State Descriptions for Example 2

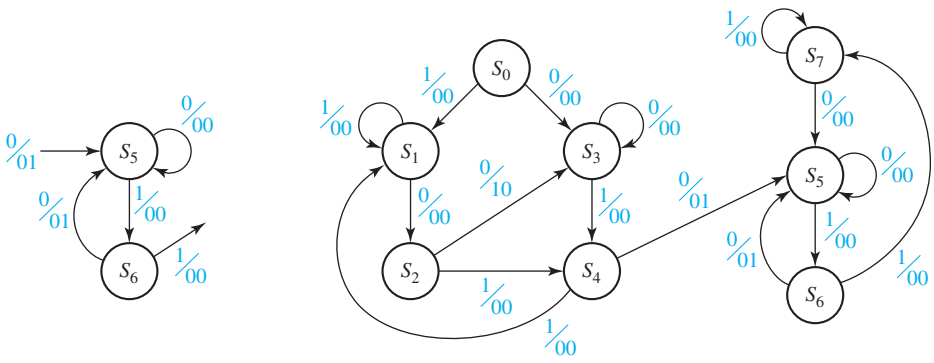
State	Description		
S_0	No progress on 100	No progress on 010	
S_1	Progress of 1 on 100	No progress on 010	
S_2	Progress of 10 on 100	Progress of 0 on 010	010 has never occurred
S_3	No progress on 100	Progress of 0 on 010	
S_4	Progress of 1 on 100	Progress of 01 on 010	
S_5		Progress of 0 on 010	
S_6		Progress of 01 on 010	010 has occurred
S_7		No progress on 010	

there is no progress toward the sequence 010, and in S_3, there is no progress toward the sequence 100. However, in S_2, we have received 10, so if the next two inputs are 1 and 0, the sequence 010 will be completed. Therefore, in S_2, we have not only made progress of 10 toward 100, but we have also made progress of 0 toward 010. Similarly, in S_4, we have made progress of 1 toward 100, as well as progress of 01 toward 010.

Using this information, we can fill in more of the state graph to get Figure 14-15(b). If the circuit is in state S_1 and a 1 is received, then the last two inputs are 11. The previous 1 is of no use toward the sequence 100. However, the circuit will need to remember the new 1, and there is a progress of 1 toward the sequence 100. There is no progress on the sequence 010, and 010 has never occurred, but this is the same situation as state S_1. Therefore, the circuit should return to state S_1. Similarly, if a 0 is received in state S_3, the last two inputs are 00. There is a progress only of 0 toward the sequence 010, there is no progress toward 100, and 010 has never occurred, so the circuit should return to state S_3. In state S_2, if a 0 is received, the sequence 100 is complete and the circuit should output $Z_1 = 1$. Then, there is no progress on another sequence of 100, and 010 has still not occurred. However, the last input is 0, so there is progress of 0 toward the sequence 010. We can see from Table 14-5 that this is the same situation as S_3, so the circuit should go to state S_3. If, in state S_2, a 1 is received, we have made progress of 01 toward 010 and progress of 1 toward 100, and 010 has still not occurred. We can see from Table 14-5 that the circuit should go to state S_4.

If a 0 is received in state S_4, the sequence 010 is complete, and we should output $Z_2 = 1$. At this point we must go to a new state (S_5) to remember that 010 has been received so that $Z_1 = 1$ can never occur again. When S_5 is reached, we stop looking for 100 and only look for 010. Figure 14-16(a) shows a partial state graph that outputs $Z_2 = 1$ when the input sequence ends in 010. In S_5 we have progress of 0 toward 010 and additional 0's can be ignored by looping back to S_5. In S_6 we have progress of 01 toward 010. If a 0 is received, the sequence is completed, $Z_2 = 1$ and we can go back to S_5 because this 0 starts the 010 sequence again.

FIGURE 14-16
State Graphs for Example 2



(a) Partial graph for 010

(b) Complete state graph

If we receive a 1 in state S_6, the 010 sequence is broken and we must add a new state (S_7) to start looking for 010 again. In state S_7 we ignore additional 1's, and when a 0 is received, we go back to S_5 because this 0 starts the 010 sequence over again. Figure 14-16(b) shows the complete state graph, and the corresponding table is Table 14-6.

TABLE 14-6

Present State	Next State $X = 0$	Next State $X = 1$	Output (Z_1Z_2) $X = 0$	Output (Z_1Z_2) $X = 1$
S_0	S_3	S_1	00	00
S_1	S_2	S_1	00	00
S_2	S_3	S_4	10	00
S_3	S_3	S_4	00	00
S_4	S_5	S_1	01	00
S_5	S_5	S_6	00	00
S_6	S_5	S_7	01	00
S_7	S_5	S_7	00	00

Example 3

A sequential circuit has two inputs (X_1, X_2) and one output (Z). The output remains a constant value unless one of the following input sequences occurs:

(a) The input sequence $X_1\ X_2 = 01, 11$ causes the output to become 0.
(b) The input sequence $X_1\ X_2 = 10, 11$ causes the output to become 1.
(c) The input sequence $X_1\ X_2 = 10, 01$ causes the output to change value.

(The notation $X_1X_2 = 01, 11$ means $X_1 = 0, X_2 = 1$ followed by $X_1 = 1, X_2 = 1$.) Derive a Moore state graph for the circuit.

Solution

The only sequences of input pairs which affect the output are of length two. Therefore, the previous and present inputs will determine the output, and the circuit must remember only the previous input pair. At first, it appears that three states are required, corresponding to the last input received being $X_1X_2 = 01, 10$ and (00 or 11). Note that it is unnecessary to use a separate state for 00 and 11 because neither input starts a sequence which leads to an output change. However, for each of these states the output could be either 0 or 1, so we will initially define six states as follows:

Previous Input (X_1X_2)	Output (Z)	State Designation
00 or 11	0	S_0
00 or 11	1	S_1
01	0	S_2
01	1	S_3
10	0	S_4
10	1	S_5

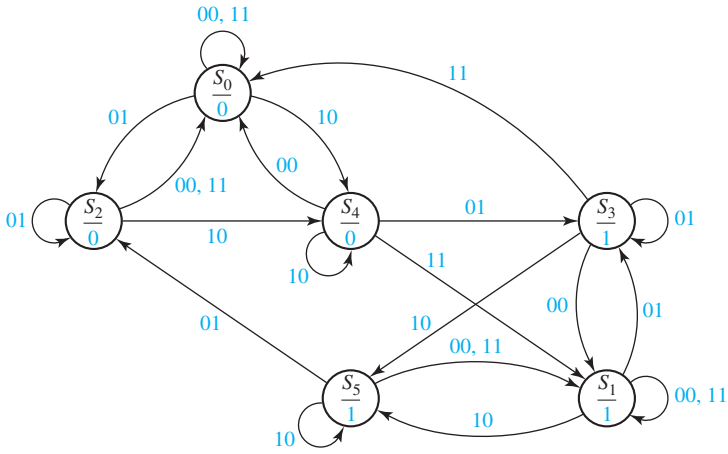
Using this state designation, we can then set up a state table (Table 14-7). The six-row table given here can be reduced to five rows, using the methods given in Unit 15.

TABLE 14-7

Present State	Z	Next State $X_1X_2 = 00$	01	11	10
S_0	0	S_0	S_2	S_0	S_4
S_1	1	S_1	S_3	S_1	S_5
S_2	0	S_0	S_2	S_0	S_4
S_3	1	S_1	S_3	S_0	S_5
S_4	0	S_0	S_3	S_1	S_4
S_5	1	S_1	S_2	S_1	S_5

The S_4 row of this table was derived as follows. If 00 is received, the input sequence has been 10, 00, so the output does not change, and we go to S_0 to remember that the last input received was 00. If 01 is received, the input sequence has been 10, 01, so the output must change to 1, and we go to S_3 to remember that the last input received was 01. If 11 is received, the input sequence has been 10, 11, so the output should become 1, and we go to S_1. If 10 is received, the input sequence has been 10, 10, so the output does not change, and we remain in S_4. Verify for yourself that the other rows in the table are correct. The state graph is shown in Figure 14-17.

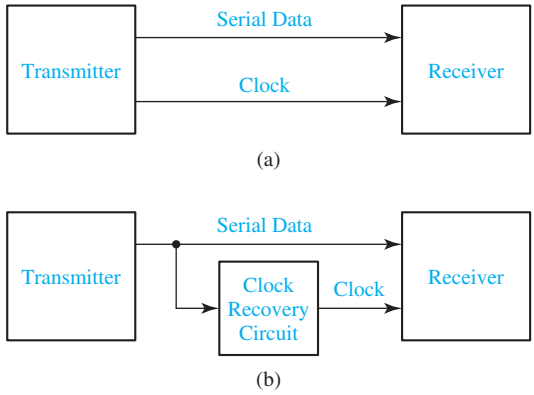
FIGURE 14-17
State Graph for Example 3



14.4 Serial Data Code Conversion

As a final example of state graph construction, we will design a converter for serial data. Binary data is frequently transmitted between computers as a serial stream of bits. As shown in Figure 14-18(a), a clock signal is often transmitted along with the data,

FIGURE 14-18
Serial Data Transmission



3. If the reduced table has m states ($2^{n-1} < m \leq 2^n$), n flip-flops are required. Assign a unique combination of flip-flop states to correspond to each state in the reduced table. The guidelines given in Section 15.8 may prove helpful in finding an assignment which leads to an economical circuit.
4. Form the transition table by substituting the assigned flip-flop states for each state in the reduced state table. The resulting transition table specifies the next states of the flip-flops, and the output in terms of the present states of the flip-flops and the input.
5. Plot next-state maps and input maps for each flip-flop and derive the flip-flop input equations. (Depending on the type of gates to be used, either determine the sum-of-products form from the 1's on the map or the product-of-sums form from the 0's on the map.) Derive the output functions.
6. Realize the flip-flop input equations and the output equations using the available logic gates.
7. Check your design by signal tracing, computer simulation, or laboratory testing.

16.2 Design Example–Code Converter

We will design a sequential circuit to convert BCD to excess-3 code. This circuit adds three to a binary-coded-decimal digit in the range 0 to 9. The input and output will be serial with the least significant bit first. A list of allowed input and output sequences is shown in Table 16-1.

Table 16-1 lists the desired inputs and outputs at times t_0, t_1, t_2, and t_3. After receiving four inputs, the circuit should reset to the initial state, ready to receive another group of four inputs. It is not clear at this point whether a sequential circuit can actually be realized to produce the output sequences as specified in Table 16-1 without delaying the output.

TABLE 16-1

X Input (BCD)				Z Output (excess-3)			
t_3	t_2	t_1	t_0	t_3	t_2	t_1	t_0
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

For example, if at t_0 some sequences required an output $Z = 0$ for $X = 0$ and other sequences required $Z = 1$ for $X = 0$, it would be impossible to design the circuit without delaying the output. For Table 16-1 we see that at t_0 if the input is 0 the output is always 1, and if the input is 1 the output is always 0; therefore, there is no conflict at t_0. At time t_1 the circuit will have available only the inputs received at t_1 and t_0. There will be no conflict at t_1 if the output at t_1 can be determined only from the inputs received at t_1 and t_0. If 00 has been received at t_1 and t_0, the output should be 1 at t_1 in all three cases where 00 occurs in the table. If 01 has been received, the output should be 0 at t_1 in all three cases where 01 occurs. For sequences 10 and 11 the outputs at t_1 should be 0 and 1, respectively. Therefore, there is no output conflict at t_1. In a similar manner we can check to see that there is no conflict at t_2, and at t_3 all four inputs are available, so there is no problem.

We will now proceed to set up the state table (Table 16-2), using the same procedure as in Section 15.1. The arrangement of next states in the table is different from that in Table 15-1 because in this example the input sequences are received with least significant bit first, while for Table 15-1 the first input bit received is listed first in the sequence. Dashes (don't-cares) appear in this table because only 10 of the 16 possible 4-bit sequences can occur as inputs to the code converter. The output part of the table is filled in, using the reasoning discussed in the preceding paragraph. For example, if the circuit is in state B at t_1 and a 1 is received, this means that the sequence 10 has been received and the output should be 0.

Next, we will reduce the table using row matching. When matching rows which contain dashes (don't-cares), a dash will match with any state or with any output value. By matching rows in this manner, we have $H \equiv I \equiv J \equiv K \equiv L$ and $M \equiv N \equiv P$. After eliminating I, J, K, L, N, and P, we find $E \equiv F \equiv G$ and the table reduces to seven rows (Table 16-3).

TABLE 16-2 State Table for Code Converter

Time	Input Sequence Received (Least Significant Bit First)	Present State	Next State $X = 0$	Next State 1	Present Output (Z) $X = 0$	Present Output (Z) 1
t_0	reset	A	B	C	1	0
t_1	0	B	D	F	1	0
	1	C	E	G	0	1
t_2	00	D	H	L	0	1
	01	E	I	M	1	0
	10	F	J	N	1	0
	11	G	K	P	1	0
t_3	000	H	A	A	0	1
	001	I	A	A	0	1
	010	J	A	–	0	–
	011	K	A	–	0	–
	100	L	A	–	0	–
	101	M	A	–	1	–
	110	N	A	–	1	–
	111	P	A	–	1	–

TABLE 16-3 Reduced State Table for Code Converter

Time	Present State	Next State $X=0$	Next State $X=1$	Present Output (Z) $X=0$	Present Output (Z) $X=1$
t_0	A	B	C	1	0
t_1	B	D	E	1	0
	C	E	E	0	1
t_2	D	H	H	0	1
	E	H	M	1	0
t_3	H	A	A	0	1
	M	A	–	1	–

An alternate approach to deriving Table 16-2 is to start with a state graph. The state graph (Figure 16-1) has the form of a tree. Each path starting at the reset state represents one of the ten possible input sequences. After the paths for the input sequences have been constructed, the outputs can be filled in by working backwards along each path. For example, starting at t_3, the path 0 0 0 0 has outputs 0 0 1 1 and the path 1 0 0 0 has outputs 1 0 1 1. Verify that Table 16-2 corresponds to this state graph.

Three flip-flops are required to realize the reduced table because there are seven states. Each of the states must be assigned a unique combination of flip-flop states. Some assignments will lead to economical circuits with only a few gates, while other assignments will require many more gates. Using the guidelines given in Section 15.8, states B and C, D and E, and H and M should be given adjacent assignments in order to simplify the next-state functions. To simplify the output function, states (A, B, E, and M) and (C, D, and H) should be given adjacent assignments. A good assignment for this example is given on the map and table in Figure 16-2. After the state assignment has been made, the transition table is filled in according to the assignment, and the next-state maps are plotted as shown in Figure 16-3. The D input equations are then read off the Q^+ maps as indicated. Figure 16-4 shows the resulting sequential circuit.

FIGURE 16-1 State Graph for Code Converter

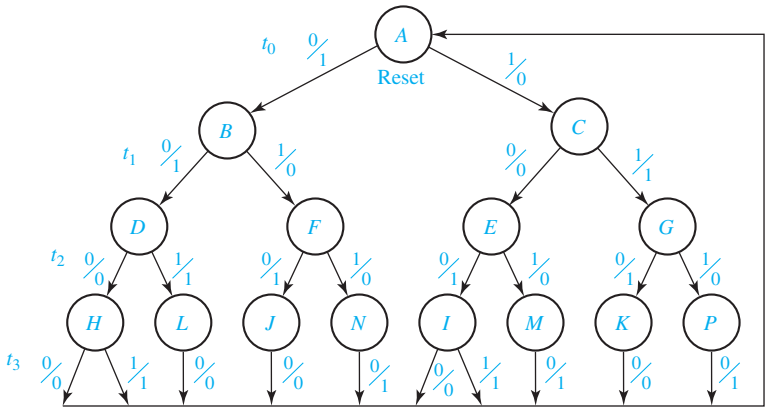


FIGURE 16-2
Assignment Map and Transition Table for Flip-Flops

Q_2Q_3 \ Q_1	0	1
00	A	B
01		C
11	H	D
10	M	E

(a) Assignment map

		$Q_1^+Q_2^+Q_3^+$		Z	
	$Q_1Q_2Q_3$	$X = 0$	$X = 1$	$X = 0$	$X = 1$
A	0 0 0	1 0 0	1 0 1	1	0
B	1 0 0	1 1 1	1 1 0	1	0
C	1 0 1	1 1 0	1 1 0	0	1
D	1 1 1	0 1 1	0 1 1	0	1
E	1 1 0	0 1 1	0 1 0	1	0
H	0 1 1	0 0 0	0 0 0	0	1
M	0 1 0	0 0 0	x x x	1	x
–	0 0 1	x x x	x x x	x	x

(b) Transition table

FIGURE 16-3
Karnaugh Maps for Code Converter Design

Q_2Q_3 \ XQ_1	00	01	11	10
00	1	1	1	1
01	X	1	1	X
11	0	0	0	0
10	0	0	0	X

$D_1 = Q_1^+ = Q_2'$

Q_2Q_3 \ XQ_1	00	01	11	10
00	0	1	1	0
01	X	1	1	X
11	0	1	1	0
10	0	1	1	X

$D_2 = Q_2^+ = Q_1$

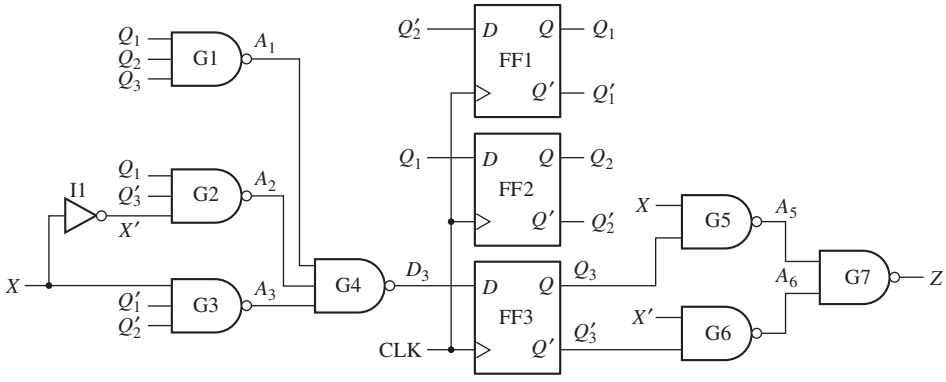
Q_2Q_3 \ XQ_1	00	01	11	10
00	0	1	0	1
01	X	0	0	X
11	0	1	1	0
10	0	1	0	X

$D_3 = Q_3^+ = Q_1Q_2Q_3 + X'Q_1Q_3' + XQ_1'Q_2'$

Q_2Q_3 \ XQ_1	00	01	11	10
00	1	1	0	0
01	X	0	1	X
11	0	0	1	1
10	1	1	0	X

$Z = X'Q_3' + XQ_3$

FIGURE 16-4
Code Converter Circuit



16.3 Design of Iterative Circuits

Many of the design procedures used for sequential circuits can be applied to the design of iterative circuits. An iterative circuit consists of a number of identical cells interconnected in a regular manner. Some operations, such as binary addition, naturally lend themselves to realization with an iterative circuit because the same operation is performed on each pair of input bits. The regular structure of an iterative circuit makes it easier to fabricate in integrated circuit form than circuits with less regular structures.

The simplest form of an iterative circuit consists of a linear array of combinational cells with signals between cells traveling in only one direction (Figure 16-5). Each cell is a combinational circuit with one or more primary inputs (x_i) and possibly one or more primary outputs (z_i). In addition, each cell has one or more secondary inputs (a_i) and one or more secondary outputs (a_{i+1}). The a_i signals carry information about the "state" of one cell to the next cell.

The primary inputs to the cells ($x_1, x_2, \ldots, x_n$) are applied in parallel; that is, they are all applied at the same time. The a_i signals then propagate down the line of cells. Because the circuit is combinational, the time required for the circuit to reach a steady-state condition is determined only by the delay times of the gates in the cells. As soon as steady state is reached, the outputs may be read. Thus, the iterative circuit can function as a parallel-input, parallel-output device, in contrast with the sequential circuit in which the input and output are serial. One can think of the iterative circuit as receiving its inputs as a sequence in space in contrast with the sequential circuit which receives its inputs as a sequence in time. The parallel adder of Figure 4-3 is an example of an iterative circuit that has four identical cells. The serial adder of Figure 13-12 uses the same full adder cell as the parallel adder, but it receives its inputs serially and stores the carry in a flip-flop instead of propagating it from cell to cell.

Design of a Comparator

As an example, we will design a circuit which compares two n-bit binary numbers and determines if they are equal or which one is larger if they are not equal. Direct design as a $2n$-input combinational circuit is not practical for n larger than 4 or 5, so we will try the iterative approach. Designate the two binary numbers to be compared as

$$X = x_1x_2 \ldots x_n \qquad \text{and} \qquad Y = y_1y_2 \ldots y_n$$

We have numbered the bits from left to right, starting with x_1 as the most significant bit because we plan to do the comparison from left to right.

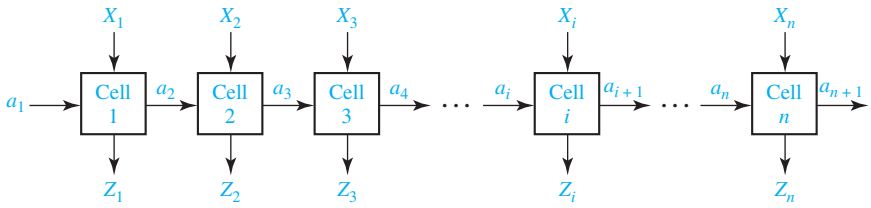


FIGURE 16-5 Unilateral Iterative Circuit

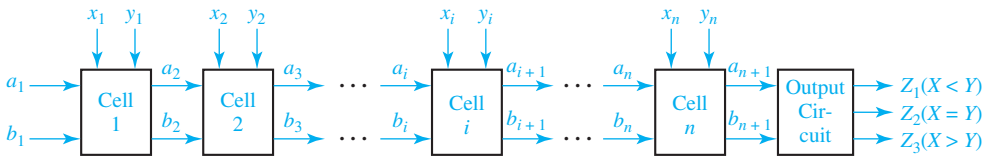


FIGURE 16-6
Form of Iterative Circuit for Comparing Binary Numbers

Figure 16-6 shows the form of the iterative circuit, although the number of leads between each pair of cells is not yet known. Comparison proceeds from left to right. The first cell compares x_1 and y_1 and passes on the result of the comparison to the next cell, the second cell compares x_2 and y_2, etc. Finally, x_n and y_n are compared by the last cell, and the output circuit produces signals to indicate if $X = Y$, $X > Y$, or $X < Y$.

We will now design a typical cell for the comparator. To the left of cell i, three conditions are possible: $X = Y$ so far ($x_1\, x_2 \ldots x_{i-1} = y_1 y_2 \ldots y_{i-1}$), $X > Y$ so far, and $X < Y$ so far. We designate these three input conditions as states S_0, S_1, and S_2, respectively. Table 16-4 shows the output state at the right of the cell (S_{i+1}) in terms of the $x_i y_i$ inputs and the input state at the left of the cell (S_i). If the numbers are equal to the left of cell i and $x_i = y_i$, the numbers are still equal including cell i, so $S_{i+1} = S_0$. However, if $S_i = S_0$ and $x_i y_i = 10$, then $x_1 x_2 \ldots x_i > y_1 y_2 \ldots y_i$ and $S_{i+1} = S_1$. If $X > Y$ to the left of cell i, then regardless of the values of x_i and y_i, $x_1 x_2 \ldots x_i > y_1 y_2 \ldots y_i$ and $S_{i+1} = S_1$. Similarly, if $X < Y$ to the left of cell i, then $X < Y$ including the inputs to cell i, and $S_{i+1} = S_2$.

TABLE 16-4
State Table for Comparator

		S_{i+1}				
	S_i	$x_i y_i = 00$	01	11	10	$Z_1 Z_2 Z_3$
$X = Y$	S_0	S_0	S_2	S_0	S_1	0 1 0
$X > Y$	S_1	S_1	S_1	S_1	S_1	0 0 1
$X < Y$	S_2	S_2	S_2	S_2	S_2	1 0 0

The logic for a typical cell is easily derived from the state table. Because there are three states, two intercell signals are required. Using the guidelines from Section 15.8 leads to the state assignment $a_i b_i = 00$ for S_0, 01 for S_1, and 10 for S_2. Substituting this assignment into the state table yields Table 16-5. Figure 16-7 shows the Karnaugh maps, next-state equations, and the realization of a typical cell using NAND gates. Inverters must be included in the cell because only a_i and b_i and not their complements are transmitted between cells.

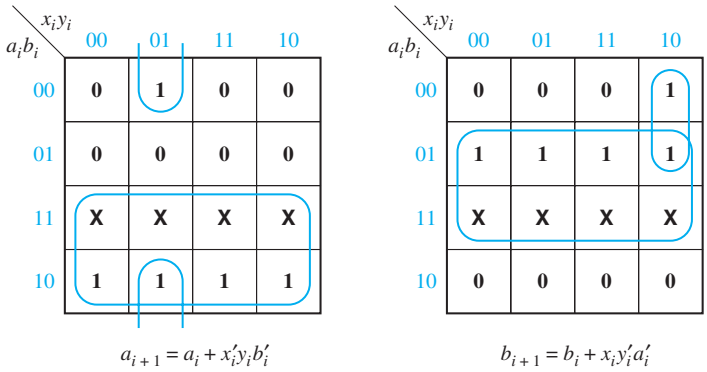
The $a_1 b_1$ inputs to the left end cell must be 00 because we must assume that the numbers are equal (all 0) to the left of the most significant bit. The equations for the first cell can then be simplified if desired:

$$a_2 = a_1 + x_1' y_1 b_1' = x_1' y_1$$
$$b_2 = b_1 + x_1 y_1' a_1' = x_1 y_1'$$

TABLE 16-5
Transition Table for Comparator

	$a_{i+1} b_{i+1}$				
$a_i b_i$	$x_i y_i = 00$	01	11	10	$Z_1 Z_2 Z_3$
0 0	00	10	00	01	0 1 0
0 1	01	01	01	01	0 0 1
1 0	10	10	10	10	1 0 0

FIGURE 16-7
Typical Cell for Comparator



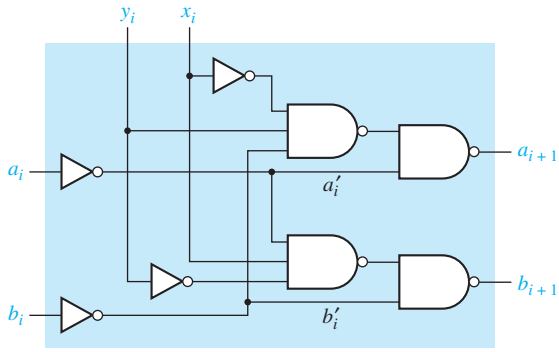
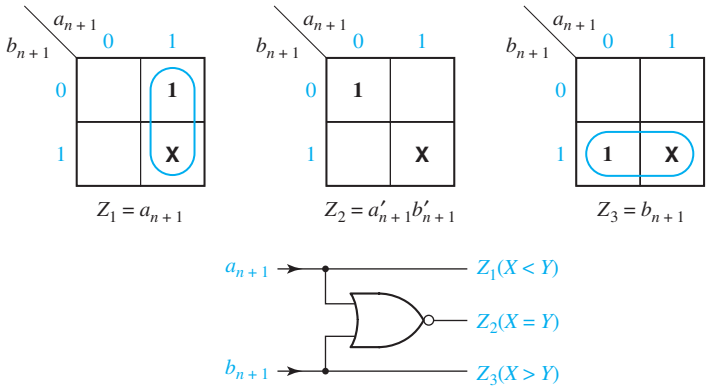


FIGURE 16-8
Output Circuit for Comparator



For the output circuit, let $Z_1 = 1$ if $X < Y$, $Z_2 = 1$ if $X = Y$, $Z_3 = 1$ if $X > Y$. Figure 16-8 shows the output maps, equations, and circuit.

Conversion to a sequential circuit is straightforward. If x_i and y_i inputs are received serially instead of in parallel, Table 16-4 is interpreted as a state table for a sequential circuit, and the next-state equations are the same as in Figure 16-7. If D flip-flops are used, the typical cell of Figure 16-7 can be used as the combinational part of the sequential circuit, and Figure 16-9 shows the resulting circuit. After all of the inputs have been read in, the output is determined from the state of the two flip-flops.

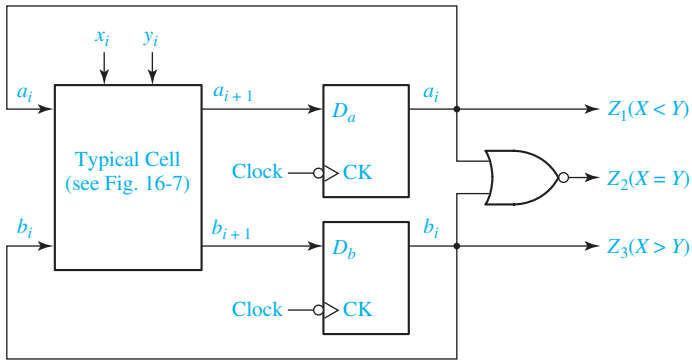


FIGURE 16-9
Sequential Comparator for Binary Numbers

This example indicates that the design of a unilateral iterative circuit is very similar to the design of a sequential circuit. The principal difference is that for the iterative circuit the inputs are received in parallel as a sequence in space, while for the sequential circuit the inputs are received serially as a sequence in time. For the iterative circuit, the state table specifies the output state of a typical cell in terms of its input state and primary inputs, while for the corresponding sequential circuit, the same table specifies the next state (in time) in terms of the present state and inputs. If D flip-flops are used, the typical cell for the iterative circuit can serve as the combinational logic for the corresponding sequential circuit. If other flip-flop types are used, the input equations can be derived in the usual manner.

16.4 Design of Sequential Circuits Using ROMs and PLAs

A sequential circuit can easily be designed using a ROM (read-only memory) and flip-flops. Referring to the general model of a Mealy sequential circuit given in Figure 13-17, the combinational part of the sequential circuit can be realized using a ROM. The ROM can be used to realize the output functions ($Z_1, Z_2, \ldots, Z_n$) and the next-state functions ($Q_1^+, Q_2^+, \ldots, Q_k^+$). The state of the circuit can then be stored in a register of D flip-flops and fed back to the input of the ROM. Thus, a Mealy sequential circuit with m inputs, n outputs, and k state variables can be realized using k D flip-flops and a ROM with $m + k$ inputs (2^{m+k} words) and $n + k$ outputs. The Moore sequential circuit of Figure 13-19 can be realized in a similar manner. The next-state and output combinational subcircuits of the Moore circuit can be realized using two ROMs. Alternatively, a single ROM can be used to realize both the next-state and output functions.

Use of D flip-flops is preferable to J-K flip-flops because use of two-input flip-flops would require increasing the number of outputs from the ROM. The fact that the D flip-flop input equations would generally require more gates than the J-K equations is of no consequence because the size of the ROM depends only on the number of inputs and outputs and not on the complexity of the equations being

realized. For this reason, the state assignment which is used is also of little importance, and, generally, a state assignment in straight binary order is as good as any.

In Section 16.2, we realized a code converter using gates and D flip-flops. We will now realize this converter using a ROM and D flip-flops. The state table for the converter is reproduced in Table 16-6(a). Because there are seven states, three D flip-flops are required. Thus, a ROM with four inputs (2^4 words) and four outputs is required, as shown in Figure 16-10. Using a straight binary state assignment, we can construct the transition table, seen in Table 16-6(b), which gives the next state of the flip-flops as a function of the present state and input. Because we are using D flip-flops, $D_1 = Q_1^+$, $D_2 = Q_2^+$, and $D_3 = Q_3^+$. The truth table for the ROM, shown in Table 16-6(c), is easily constructed from the transition table. This table gives the ROM outputs (Z, D_1, D_2, and D_3) as functions of the ROM inputs (X, Q_1, Q_2, and Q_3).

Sequential circuits can also be realized using PLAs (programmable logic arrays) and flip-flops in a manner similar to using ROMs and flip-flops. However, in the case of PLAs, the state assignment may be important because the use of a

TABLE 16-6

(a) State table

Present State	Next State $X=0$	Next State 1	Present Output (Z) $X=0$	Present Output (Z) 1
A	B	C	1	0
B	D	E	1	0
C	E	E	0	1
D	H	H	0	1
E	H	M	1	0
H	A	A	0	1
M	A	–	1	–

(b) Transition table

	$Q_1Q_2Q_3$	$Q_1^+Q_2^+Q_3^+$ $X=0$	$Q_1^+Q_2^+Q_3^+$ $X=1$	Z $X=0$	Z $X=1$
A	0 0 0	001	010	1	0
B	0 0 1	011	100	1	0
C	0 1 0	100	100	0	1
D	0 1 1	101	101	0	1
E	1 0 0	101	110	1	0
H	1 0 1	000	000	0	1
M	1 1 0	000	–	1	–

(c) Truth table

X	Q_1	Q_2	Q_3	Z	D_1	D_2	D_3
0	0	0	0	1	0	0	1
0	0	0	1	1	0	1	1
0	0	1	0	0	1	0	0
0	0	1	1	0	1	0	1
0	1	0	0	1	1	0	1
0	1	0	1	0	0	0	0
0	1	1	0	1	0	0	0
0	1	1	1	x	x	x	x
1	0	0	0	0	0	1	0
1	0	0	1	0	1	0	0
1	0	1	0	1	1	0	0
1	0	1	1	1	1	0	1
1	1	0	0	0	1	1	0
1	1	0	1	1	0	0	0
1	1	1	0	x	x	x	x
1	1	1	1	x	x	x	x

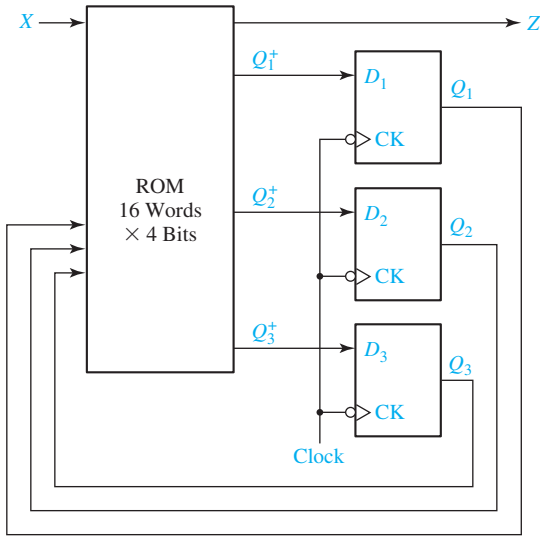


FIGURE 16-10
Realization of Table 16.6(a) Using a ROM

good state assignment can reduce the required number of product terms and, hence, reduce the required size of the PLA.

As an example, we will consider realizing the state table of Table 16-6(a) using a PLA and three D flip-flops. The circuit configuration is the same as Figure 16-10, except that the ROM is replaced with a PLA of appropriate size. Using a straight binary assignment leads to the truth table given in Table 16-6(c). This table could be stored in a PLA with four inputs, 13 product terms, and four outputs, but this would offer little reduction in size compared with the 16-word ROM solution discussed earlier.

If the state assignment of Figure 16-2 is used, the resulting output equation and D flip-flop input equations, derived from the maps in Figure 16-3, are

$$\begin{aligned} D_1 &= Q_1^+ = Q_2' \\ D_2 &= Q_2^+ = Q_1 \\ D_3 &= Q_3^+ = Q_1Q_2Q_3 + X'Q_1Q_3' + XQ_1'Q_2' \\ Z &= X'Q_3' + XQ_3 \end{aligned} \tag{16-1}$$

The PLA table which corresponds to these equations is in Table 16-7. Realization of this table requires a PLA with four inputs, seven product terms, and four outputs.

TABLE 16-7

X	Q_1	Q_2	Q_3	Z	D_1	D_2	D_3
–	–	0	–	0	1	0	0
–	1	–	–	0	0	1	0
–	1	1	1	0	0	0	1
0	1	–	0	0	0	0	1
1	0	0	–	0	0	0	1
0	–	–	0	1	0	0	0
1	–	–	1	1	0	0	0

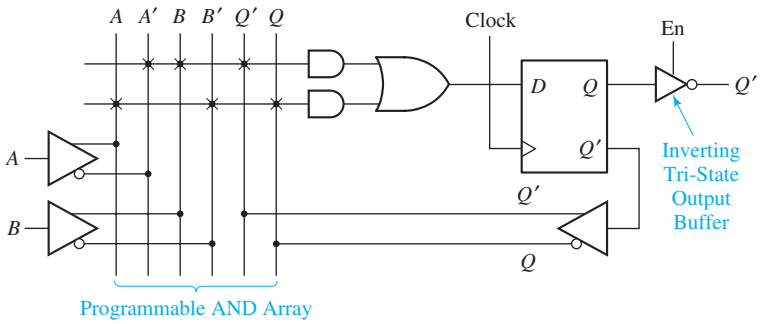
Next, we will verify the operation of the circuit of Figure 16-4 using a PLA which corresponds to Table 16-7. Initially, assume that $X = 0$ and $Q_1Q_2Q_3 = 000$. This selects rows --0- and 0--0 in the table, so $Z = 1$ and $D_1D_2D_3 = 100$. After the active clock edge, $Q_1Q_2Q_3 = 100$. If the next input is $X = 1$, then rows --0- and -1-- are selected, so $Z = 0$ and $D_1D_2D_3 = 110$. After the active clock edge, $Q_1Q_2Q_3 = 110$. Continuing in this manner, we can verify the transition table of Figure 16-2.

PALs also provide a convenient way of realizing sequential circuits. PALs are available which contain D flip-flops that have their inputs driven from programmable array logic. Figure 16-11 shows a segment of a sequential PAL. The D flip-flop is driven from an OR gate which is fed by two AND gates. The flip-flop output is fed back to the programmable AND array through a buffer. Thus, the AND gate inputs can be connected to A, A', B, B', Q, or Q'. The X's on the diagram show the connections required to realize the next-state equation

$$Q^+ = D = A'BQ' + AB'Q$$

The flip-flop output is connected to an inverting tri-state buffer, which is enabled when En = 1.

FIGURE 16-11
Segment of a Sequential PAL



16.5 Sequential Circuit Design Using CPLDs

As discussed in Section 9.7, a typical CPLD contains a number of macrocells that are grouped into function blocks. Connections between the function blocks are made through an interconnection array. Each macrocell contains a flip-flop and an OR gate, which has its inputs connected to an AND gate array. Some CPLDs are based on PALs, in which case each OR gate has a fixed set of AND gates associated with it. Other CPLDs are based on PLAs, in which case any AND gate output within a function block can be connected to any OR gate input in that block.

Figure 16-12 shows the structure of a Xilinx CoolRunner II CPLD, which uses a PLA in each function block. This CPLD family is available in sizes from two to 32 function blocks (32 to 512 macrocells). Each function block has 16 inputs from the AIM (advanced interconnection matrix) and up to 40 outputs to the AIM. Each function block PLA contains the equivalent of 56 AND gates.

4. Optional simulation exercises:
   (a) Simulate the serial adder of Figure 13-12 and test it.
   (b) Connect two 4-bit shift registers to the inputs of the adder that you simulated in (a) to form a serial adder with accumulator (as in Figure 18-1). Supply the shift signal and clock signal from switches so that a control circuit is unnecessary. Test your adder using the following pairs of binary numbers:

       0101 + 0110, 1011 + 1101

   (c) Input the control circuit from the equations of Figure 18-4, connect it to the circuit which you built in (b), and test it.
5. When you are satisfied that you can meet all of the objectives, take the readiness test.

Circuits for Arithmetic Operations

This unit introduces the concept of using a sequential circuit to control a sequence of operations in a digital system. Such a control circuit outputs a sequence of control signals that cause operations such as addition or shifting to take place at the appropriate times. We will illustrate the use of control circuits by designing a serial adder, a multiplier, and a divider.

18.1 Serial Adder with Accumulator

In this section we will design a control circuit for a serial adder with an accumulator. Figure 18-1 shows a block diagram for the adder. Two shift registers are used to hold the 4-bit numbers to be added, X and Y. The X register serves as an

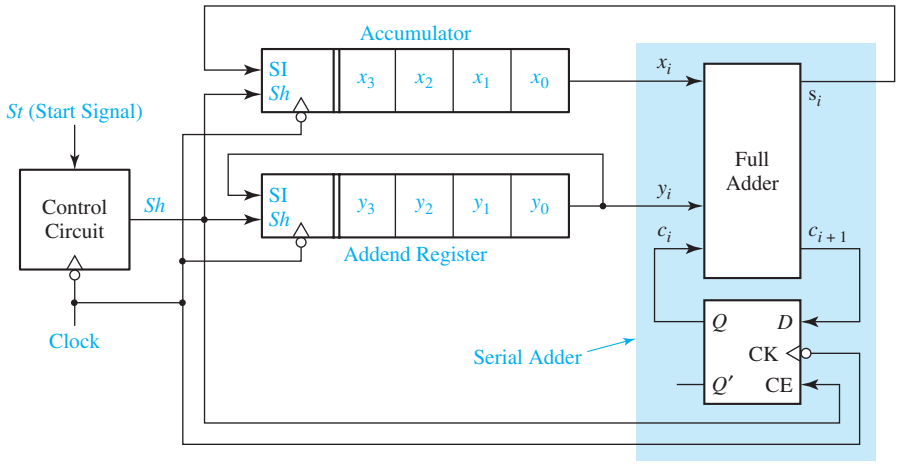


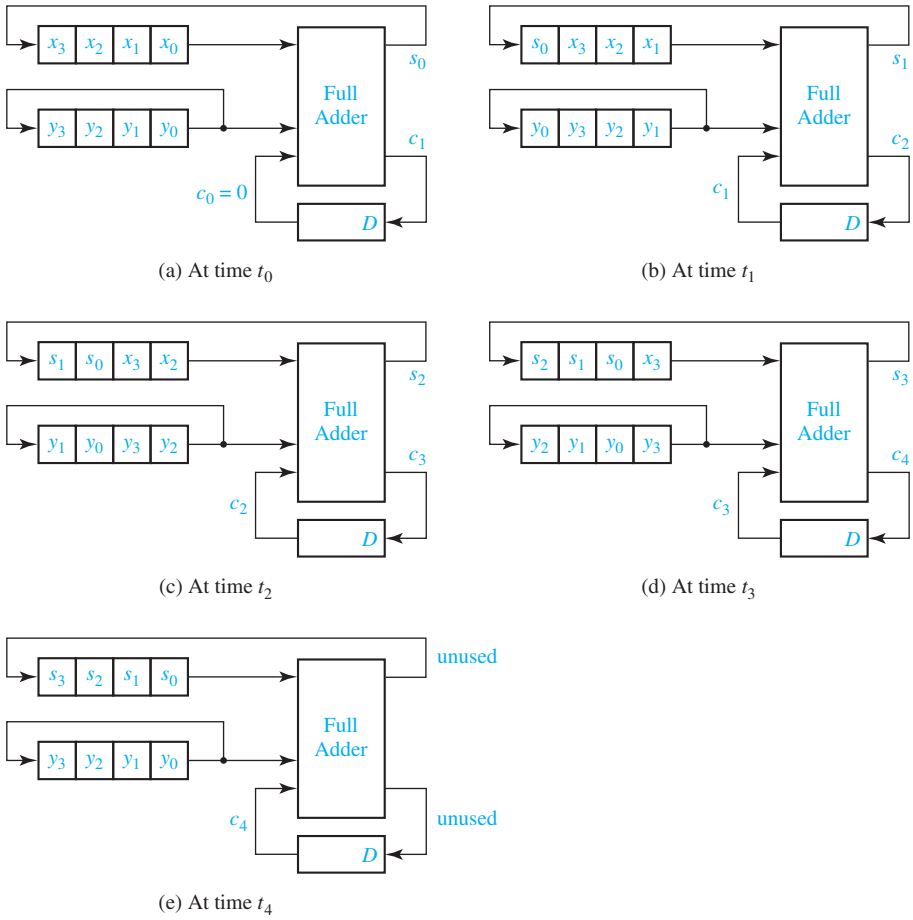
FIGURE 18-1
Block Diagram for Serial Adder with Accumulator

accumulator and the Y register serves as an addend register. When the addition is completed, the contents of the X register are replaced with the sum of X and Y. The addend register is connected as a cyclic shift register so that after shifting four times it is back in its original state, and the number Y is not lost. The box at the left end of each shift register shows the inputs: Sh (shift signal), SI (serial input), and *Clock*. When $Sh = 1$ and an active clock edge occurs, SI is entered into x_3 (or y_3) at the same time as the contents of the register are shifted one place to the right. The additional connections required for initially loading the X and Y registers and clearing the carry flip-flop are not shown in the block diagram.

The serial adder, highlighted in blue in the diagram, is the same as the one in Figure 13-12, except the D flip-flop has been replaced with a D flip-flop with clock enable. At each clock time, one pair of bits is added. Because the full adder is a combinational circuit, the sum and carry appear at the full adder output after the propagation delay. When $Sh = 1$, the falling clock edge shifts the sum bit into the accumulator, stores the carry bit in the carry flip-flop, and rotates the addend register one place to the right. Because Sh is connected to CE on the flip-flop, the carry is only updated when shifting occurs.

Figure 18-2 illustrates the operation of the adder. Shifting occurs on the falling clock edge when $Sh = 1$. In this figure, t_0 is the time before the first shift, t_1 is the time after the first shift, t_2 is the time after the second shift, etc. Initially, at time t_0, the accumulator contains X and the addend register contains Y. Because the full adder is a combinational circuit, x_0, y_0, and c_0 are added independently of the clock to form the sum s_0 and carry c_1. When the first falling clock edge occurs, s_0 is shifted into the accumulator and the remaining accumulator digits are shifted one position to the right. The same clock edge stores c_1 in the carry flip-flop and rotates the addend register right. The next pair of bits, x_1 and y_1, are now at the full adder input, and the adder generates the sum and carry, s_1

FIGURE 18-2
Operation of Serial Adder



and c_2, as seen in Figure 18-2(b). The second falling edge shifts s_1 into the accumulator, stores c_2 in the carry flip-flop, and cycles the addend register right. Bits x_2 and y_2 are now at the adder input, as seen in Figure 18-2(c), and the process continues until all bit pairs have been added, as shown in Figure 18-2(e).

Table 18-1 shows a numerical example of the serial adder operation. Initially, the accumulator contains 0101 and the addend register contains 0111. At t_0, the full adder computes $1 + 1 + 0 = 10$, so $s_i = 0$ and $c_i^+ = 1$. After the first falling clock

TABLE 18-1
Operation of Serial Adder

	X	Y	C_i	S_i	C_i^+
t_0	0101	0111	0	0	1
t_1	0010	1011	1	0	1
t_2	0001	1101	1	1	1
t_3	1000	1110	1	1	0
t_4	1100	0111	0	(1)	(0)

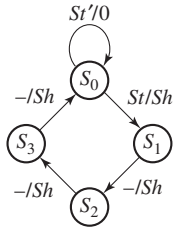
edge (time t_1) the first sum bit has been entered into the accumulator, the carry has been stored in the carry flip-flop, and the addend has been cycled right. After four falling clock edges (time t_4), the sum of X and Y is in the accumulator, and the addend register is back to its original state.

The control circuit for the adder must now be designed so that after receiving a start signal, the control circuit will put out four shift signals and then stop. Figure 18-3 shows the state graph and table for the control circuit. The circuit remains in S_0 until a start signal is received, at which time the circuit outputs $Sh = 1$ and goes to S_1. Then, at successive clock times, three more shift signals are put out. It will be assumed that the start signal is terminated before the circuit returns to state S_0 so that no further output occurs until another start signal is received. Dashes appear on the graph because once S_1 is reached, the circuit operation continues regardless of the value of St. Starting with the state table of Figure 18-3 and using a straight binary state assignment, the control circuit equations are derived in Figure 18-4.

A serial processing unit, such as a serial adder with an accumulator, processes data one bit at a time. A typical serial processing unit (Figure 18-5) has two shift registers. The output bits from the shift register are inputs to a combinational circuit. The combinational circuit generates at least one output bit. This output bit is fed into the input of a shift register. When the active clock edge occurs, this bit is stored in the first bit of the shift register at the same time the register bits are shifted to the right.

The control for the serial processing unit generates a series of shift signals. When the start signal (St) is 1, the first shift signal (Sh) is generated. If the shift registers

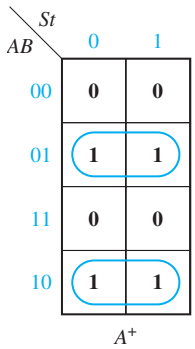
FIGURE 18-3
State Graph for Serial Adder Control



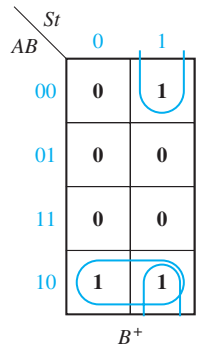
	Next State		Sh	
	$St = 0$	1	0	1
S_0	S_0	S_1	0	1
S_1	S_2	S_2	1	1
S_2	S_3	S_3	1	1
S_3	S_0	S_0	1	1

FIGURE 18-4
Derivation of Control Circuit Equations

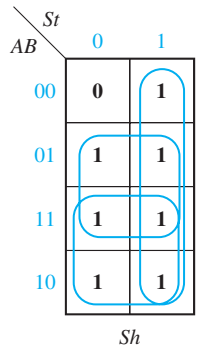
	AB	A^+B^+	
		0	1
S_0	00	00	01
S_1	01	10	10
S_2	10	11	11
S_3	11	00	00



$D_A = A'B + AB'$
$= A \oplus B$



$D_B = St\,B' + AB'$



$Sh = St + A + B$

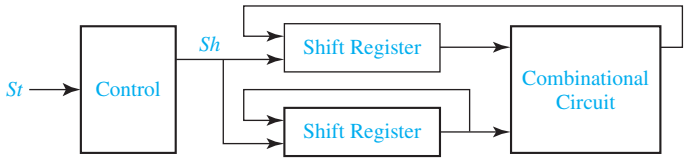


FIGURE 18-5
Typical Serial Processing Unit

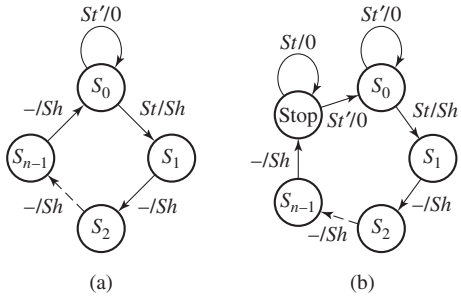


FIGURE 18-6
State Graphs for Serial Processing Unit

have n bits, then a total of n shift signals must be generated. If St is 1 for only one clock time, then the control state graph [Figure 18-6(a)] stops when it returns to state S_0. However, if St can remain 1 until after the shifting is completed, then a separate stop state is required, as shown in Figure 18-6(b). The control remains in the stop state until St returns to 0.

18.2 Design of a Parallel Multiplier

Next, we will design a parallel multiplier for positive binary numbers. As illustrated in the example in Section 1.3, binary multiplication requires only shifting and adding. The following example shows how each partial product is added in as soon as it is formed. This eliminates the need for adding more than two binary numbers at a time.

Multiplicand ⟶	1101	(13)
Multiplier ⟶	1011	(11)
	1101	
	1101	
Partial Products	100111	
	0000	
	100111	
	1101	
Product ⟶	10001111	(143)

The multiplication of two 4-bit numbers requires a 4-bit multiplicand register, a 4-bit multiplier register, and an 8-bit register for the product. The product

register serves as an accumulator to accumulate the sum of the partial products. Instead of shifting the multiplicand left each time before it is added, as was done in the previous example, it is more convenient to shift the product register to the right each time. Figure 18-7 shows a block diagram for such a parallel multiplier. As indicated by the arrows on the diagram, 4 bits from the accumulator and 4 bits from the multiplicand register are connected to the adder inputs; the 4 sum bits and the carry output from the adder are connected back to the accumulator. (The actual connections are similar to the parallel adder with accumulator shown in Figure 12-5.) The adder calculates the sum of its inputs, and when an add signal (Ad) occurs, the adder outputs are stored in the accumulator by the next rising clock edge, thus causing the multiplicand to be added to the accumulator. An extra bit at the left end of the product register temporarily stores any carry (C_4) which is generated when the multiplicand is added to the accumulator.

Because the lower four bits of the product register are initially unused, we will store the multiplier in this location instead of in a separate register. As each multiplier bit is used, it is shifted out the right end of the register to make room for additional product bits.

The Load signal loads the multiplier into the lower four bits of ACC and at the same time clears the upper 5 bits. The shift signal (Sh) causes the contents of the product register (including the multiplier) to be shifted one place to the right when the next rising clock edge occurs. The control circuit puts out the proper sequence of add and shift signals after a start signal ($St = 1$) has been received. If the current multiplier bit (M) is 1, the multiplicand is added to the accumulator followed by a right shift; if the multiplier bit is 0, the addition is skipped and only the right shift occurs. The multiplication example at the beginning of this section (13×11) is reworked below showing the location of the bits in the registers at each clock time.

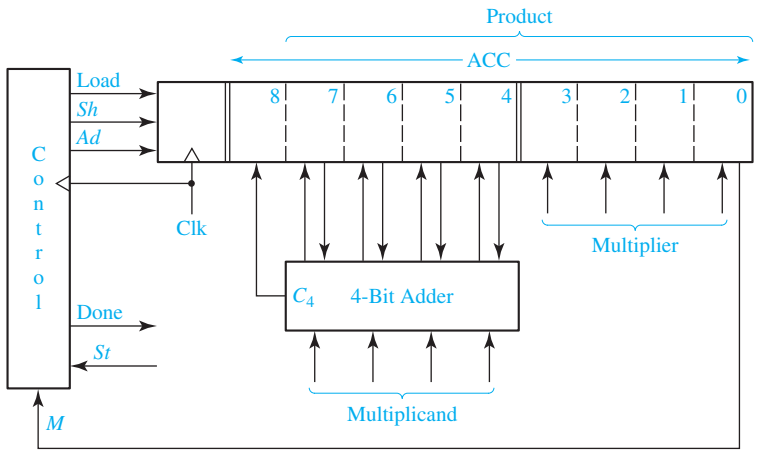
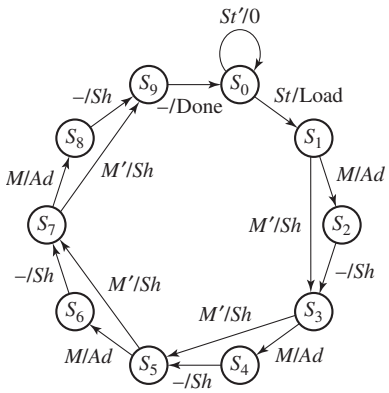


FIGURE 18-7
Block Diagram for Parallel Binary Multiplier

initial contents of product register	0 0 0 0 0 1 0 1 1 ←M	(11)
(add multiplicand because $M = 1$)	1 1 0 1	(13)
after addition	0 1 1 0 1 1 0 1 1	
after shift	0 0 1 1 0 1 1 0 1 ←M	
(add multiplicand because $M = 1$)	1 1 0 1	
after addition	1 0 0 1 1 1 1 0 1	
after shift	0 1 0 0 1 1 1 1 0 ←M	
(skip addition because $M = 0$)		
after shift	0 0 1 0 0 1 1 1 1 ←M	
(add multiplicand because $M = 1$)	1 1 0 1	
after addition	1 0 0 0 1 1 1 1 1	
after shift (final answer)	0 1 0 0 0 1 1 1 1	(143)
dividing line between product and multiplier		

The control circuit must be designed to output the proper sequence of add and shift signals. Figure 18-8 shows a state graph for the control circuit. The notation used on this graph is defined in Section 14.5. M/Ad means if $M = 1$, then the output Ad is 1 (and the other outputs are 0). M'/Sh means if $M' = 1$ ($M = 0$), then the output Sh is 1 (and the other outputs are 0). In Figure 18-8, S_0 is the reset state, and the circuit stays in S_0 until a start signal ($St = 1$) is received. This generates a Load signal, which causes the multiplier to be loaded into the lower 4 bits of the accumulator (ACC) and the upper 5 bits of ACC to be cleared on the next rising clock edge. In state S_1, the low order bit of the multiplier (M) is tested. If $M = 1$, an add signal is generated and, then, a shift signal is generated in S_2. If $M = 0$ in S_1, a shift signal is generated because adding 0 can be omitted. Similarly, in states S_3, S_5, and S_7, M is tested to determine whether to generate an add signal followed by shift or just a shift signal. A shift signal is always generated at the next clock time following an add signal (states S_2, S_4, S_6, and S_8). After four shifts have been generated, all four multiplier bits have been processed, and the control circuit goes to a Done state and terminates the multiplication process.

FIGURE 18-8
State Graph for Multiplier Control



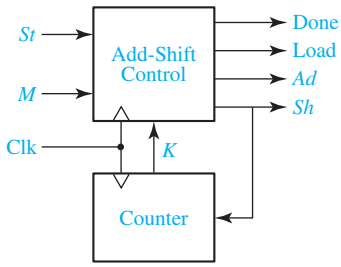
As the state graph indicates, the control performs two functions—generating add or shift signals as needed and counting the number of shifts. If the number of bits is large, it is convenient to divide the control circuit into a counter and an add-shift control, as shown in Figure 18-9(a). First, we will derive a state graph for the add-shift control which tests M and St and outputs the proper sequence of add and shift signals (Figure 18-9(b)). Then, we will add a completion signal (K) from the counter which stops the multiplier after the proper number of shifts have been completed. Starting in S_0 in Figure 18-9(b), when a start signal ($St = 1$) is received, a Load signal is generated. In state S_1, if $M = 0$, a shift signal is generated and the circuit stays in S_1. If $M = 1$, an add signal is generated and the circuit goes to state S_2. In S_2 a shift signal is generated because a shift always follows an add. Back in S_1, the next multiplier bit (M) is tested to determine whether to shift, or add and then shift. The graph of Figure 18-9(b) will generate the proper sequence of add and shift signals, but it has no provision for stopping the multiplier.

In order to determine when the multiplication is completed, the counter is incremented on the active clock edge each time a shift signal is generated. If the multiplier is n bits, a total of n shifts are required. We will design the counter so that a completion signal (K) is generated after $n - 1$ shifts have occurred. When $K = 1$, the circuit should perform one more addition if necessary and then do the final shift. The control operation in Figure 18-9(c) is the same as Figure 18-9(b) as long as $K = 0$. In state S_1, if $K = 1$, we test M as usual. If $M = 0$, we output the final shift signal and stop; however, if $M = 1$, we add before shifting and go to state S_2. In state S_2, if $K = 1$, we output one more shift signal and then go to S_3. The last shift signal will reset the counter to 0 at the same time the add-shift control goes to the Done state.

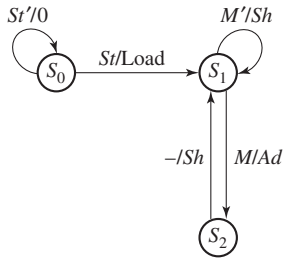
As an example, consider the multiplier of Figure 18-7, but replace the control circuit with Figure 18-9(a). Because $n = 4$, a 2-bit counter is needed, and $K = 1$ when the counter is in state 3 (11_2). Table 18-2 shows the operation of the multiplier when 1101 is multiplied by 1011. S_0, S_1, and S_2 represent states of the control circuit [Figure 18-9(c)]. The contents of the product register at each step is the same as given on p. 600.

At time t_0 the control is reset and waiting for a start signal. At time t_1, the start signal $St = 1$, and a Load signal is generated. At time t_2, $M = 1$, so an Ad signal is generated. When the next clock occurs, the output of the adder is loaded into the accumulator and the control goes to S_2. At t_3, an Sh signal is generated, so, shifting occurs and the counter is incremented at the next clock. At t_4, $M = 1$, so $Ad = 1$, and the

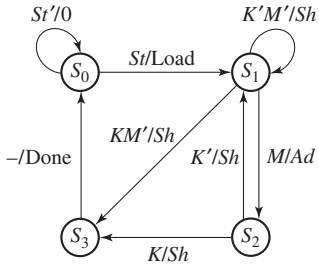
FIGURE 18-9



(a) Multiplier control



(b) State graph for add-shift control



(c) Final state graph for add-shift control

TABLE 18-2 Operation of a Multiplier Using a Counter

Time	State	Counter	Product Register	*St*	*M*	*K*	Load	*Ad*	*Sh*	Done
t_0	S_0	00	000000000	0	0	0	0	0	0	0
t_1	S_0	00	000000000	1	0	0	1	0	0	0
t_2	S_1	00	000001011	0	1	0	0	1	0	0
t_3	S_2	00	011011011	0	1	0	0	0	1	0
t_4	S_1	01	001101101	0	1	0	0	1	0	0
t_5	S_2	01	100111101	0	1	0	0	0	1	0
t_6	S_1	10	010011110	0	0	0	0	0	1	0
t_7	S_1	11	001001111	0	1	1	0	1	0	0
t_8	S_2	11	100011111	0	1	1	0	0	1	0
t_9	S_3	00	010001111	0	1	0	0	0	0	1

adder output is loaded into the accumulator at the next clock. At t_5 and t_6, shifting and counting occurs. At t_7, three shifts have occurred and the counter state is 11, so $K = 1$. Because $M = 1$, addition occurs, and the control goes to S_2. At t_8, $Sh = K = 1$, so at the next clock the final shift occurs, and the counter is incremented back to state 00. At t_9, a Done signal is generated.

The multiplier design given here can easily be expanded to 8, 16, or more bits simply by increasing the register size and the number of bits in the counter. The add-shift control would remain unchanged.

18.3 Design of a Binary Divider

We will consider the design of a parallel divider for positive binary numbers. As an example, we will design a circuit to divide an 8-bit dividend by a 4-bit divisor to obtain a 4-bit quotient. The following example illustrates the division process:

```
                        1010   quotient
divisor   1101  ⟌  10000111   dividend
                    1101
                     0111
                     0000
                      1111
(135 ÷ 13 = 10 with   1101
   a remainder of 5)   0101
                       0000
                       0101   remainder
```

Just as binary multiplication can be carried out as a series of add and shift operations, division can be carried out by a series of subtraction and shift operations. To construct the divider, we will use a 9-bit dividend register and a 4-bit divisor register, as shown in Figure 18-10. During the division process, instead of

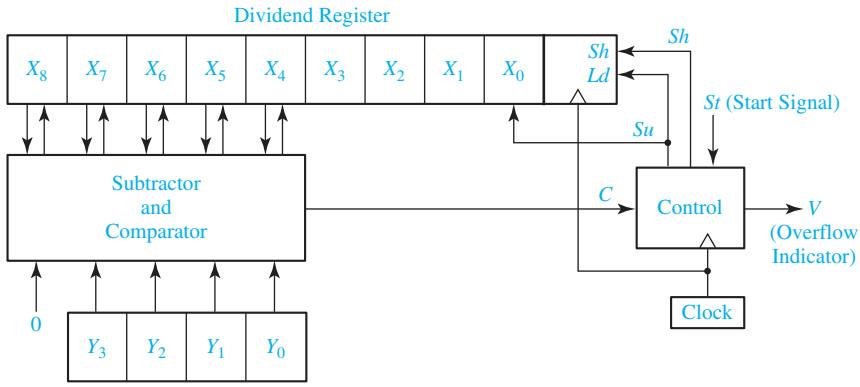


FIGURE 18-10
Block Diagram for Parallel Binary Divider

shifting the divisor to the right before each subtraction as shown in the preceding example, we will shift the dividend to the left. Note that an extra bit is required on the left end of the dividend register so that a bit is not lost when the dividend is shifted left. Instead of using a separate register to store the quotient, we will enter the quotient bit-by-bit into the right end of the dividend register as the dividend is shifted left. Circuits for initially loading the dividend into the register will be added later.

The preceding division example (135 divided by 13) is now reworked, showing the location of the bits in the registers at each clock time. Initially, the dividend and divisor are entered as follows:

0	1	0	0	0	0	1	1	1

1	1	0	1

Subtraction cannot be carried out without a negative result, so we will shift before we subtract. Instead of shifting the divisor one place to the right, we will shift the dividend one place to the left:

1 0 0 0 0 1 1 1 | 0 ← Dividing line between dividend and quotient
1 1 0 1 ← Note that after the shift, the rightmost position in the dividend register is "empty".

Subtraction is now carried out, and the first quotient digit of 1 is stored in the unused position of the dividend register:

0 0 0 1 1 1 1 1 | 1 ← first quotient digit

Next, we shift the dividend one place to the left:

0 0 1 1 1 1 1 | 1 0
1 1 0 1

Because subtraction would yield a negative result, we shift the dividend to the left again, and the second quotient bit remains 0:

$$\begin{array}{l} 0\ 1\ 1\ 1\ 1\ 1 \mid 1\ 0\ 0 \\ \quad 1\ 1\ 0\ 1 \end{array}$$

Subtraction is now carried out, and the third quotient digit of 1 is stored in the unused position of the dividend register:

$$0\ 0\ 0\ 1\ 0\ 1 \mid 1\ 0\ 1 \longleftarrow \text{third quotient digit}$$

A final shift is carried out and the fourth quotient bit is set to 0:

$$\underbrace{0\ 0\ 1\ 0\ 1}_{\text{remainder}} \mid \underbrace{1\ 0\ 1\ 0}_{\text{quotient}}$$

The final result agrees with that obtained in the first example. Note that in the first step the leftmost 1 in the dividend is shifted left into the leftmost position (X_8) in the X register. If we did not have a place for this bit, the division operation would have failed at this step because $0000 < 1101$. However, by keeping the leftmost bit in X_8, $10000 \geq 1101$, and subtraction can occur.

If as a result of a division operation, the quotient would contain more bits than are available for storing the quotient, we say that an overflow has occurred. For the divider of Figure 18-10 an overflow would occur if the quotient is greater than 15, because only 4 bits are provided to store the quotient. It is not actually necessary to carry out the division to determine if an overflow condition exists, because an initial comparison of the dividend and divisor will tell if the quotient will be too large. For example, if we attempt to divide 135 by 7, the initial contents of the registers would be:

$$\begin{array}{l} 0\ 1\ 0\ 0\ 0\ 0\ 1\ 1\ 1 \\ \quad 0\ 1\ 1\ 1 \end{array}$$

Because subtraction can be carried out with a nonnegative result, we should subtract the divisor from the dividend and enter a quotient bit of 1 in the rightmost place in the dividend register. However, we cannot do this because the rightmost place contains the least significant bit of the dividend, and entering a quotient bit here would destroy that dividend bit. Therefore, the quotient would be too large to store in the 4 bits we have allocated for it, and we have detected an overflow condition. In general, for Figure 18-10, if initially $X_8X_7X_6X_5X_4 \geq Y_3Y_2Y_1Y_0$ (i.e., if the left five bits of the dividend register exceed or equal the divisor), the quotient will be greater than 15 and an overflow occurs. Note that if $X_8X_7X_6X_5X_4 \geq Y_3Y_2Y_1Y_0$, the quotient is

$$\frac{X_8\,X_7\,X_6\,X_5\,X_4\,X_3\,X_2\,X_1\,X_0}{Y_3\,Y_2\,Y_1\,Y_0} \geq \frac{X_8\,X_7\,X_6\,X_5\,X_4\,0000}{Y_3\,Y_2\,Y_1\,Y_0} = \frac{X_8\,X_7\,X_6\,X_5\,X_4 \times 16}{Y_3\,Y_2\,Y_1\,Y_0} \geq 16$$

The operation of the divider can be explained in terms of the block diagram of Figure 18-10. A shift signal (*Sh*) will shift the dividend one place to the left on the next rising clock edge. Because the subtracter is a combinational circuit, it computes

$X_8X_7X_6X_5X_4 - Y_3Y_2Y_1Y_0$, and this difference appears at the subtracter output after a propagation delay. A subtract signal (Su) will load the subtracter output into $X_8X_7X_6X_5X_4$ and set the quotient bit (the rightmost bit in the dividend register) to 1 on the next rising clock edge. To accomplish this, Su is connected to both the Ld input on the shift register and the data input on flip-flop X_0. If the divisor is greater than the five leftmost dividend bits, the comparator output is $C = 0$; otherwise, $C = 1$. The control circuit generates the required sequence of shift and subtract signals. Whenever $C = 0$, subtraction cannot occur without a negative result, so a shift signal is generated. Whenever $C = 1$, a subtract signal is generated, and the quotient bit is set to one.

Figure 18-11 shows the state diagram for the control circuit. When a start signal (St) occurs, the 8-bit dividend and 4-bit divisor are loaded into the appropriate registers. If C is 1, the quotient would require five or more bits. Because space is only provided for a 4-bit quotient, this condition constitutes an overflow, so the divider is stopped, and the overflow indicator is set by the V output. Normally, the initial value of C is 0, so a shift will occur first, and the control circuit will go to state S_2. Then, if $C = 1$, subtraction occurs. After the subtraction is completed, C will always be 0, so the next active clock edge will produce a shift. This process continues until four shifts have occurred, and the control is in state S_5. Then, a final subtraction occurs if $C = 1$, and no subtraction occurs if $C = 0$. No further shifting is required, and the control goes to the stop state. For this example, we will assume that when the start signal (St) occurs, it will be 1 for one clock time, and, then, it will remain 0 until the control circuit is back in state S_0. Therefore, St will always be 0 in states S_1 through S_5.

We will now design the control circuit using a one-hot assignment (see Section 15.9) to implement the state graph. One flip-flop is used for each state with $Q_0 = 1$ in S_0, $Q_1 = 1$ in S_1, $Q_2 = 1$ in S_2, etc. By inspection, the next-state and output equations are

$$
\begin{aligned}
Q_0^+ &= St'Q_0 + CQ_1 + Q_5 & Q_1^+ &= StQ_0 \\
Q_2^+ &= C'Q_1 + CQ_2 & Q_3^+ &= C'Q_2 + CQ_3 \\
Q_4^+ &= C'Q_3 + CQ_4 & Q_5^+ &= C'Q_4 \\
Load &= St\,Q_0 & V &= CQ_1 \\
Sh &= C'(Q_1 + Q_2 + Q_3 + Q_4) = C'(Q_0 + Q_5)' \\
Su &= C(Q_2 + Q_3 + Q_4 + Q_5) = C(Q_0 + Q_1)'
\end{aligned}
\tag{18-1}
$$

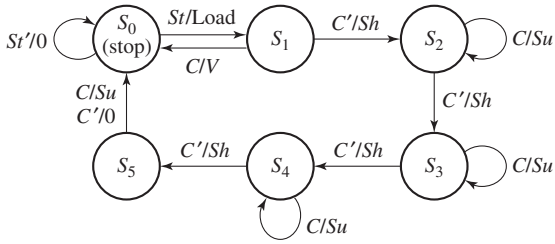
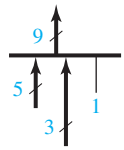


FIGURE 18-11
State Graph for Divider Control Circuit

Because there are three arrows leading into S_0, Q_0^+ has three terms. The equation for *Sh* has been simplified by noting that if the circuit is in state S_1 or S_2 or S_3 or S_4, it is not in state S_0 or S_5.

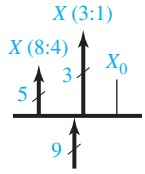
The subtracter in Figure 18-10 can be constructed using five full subtracters, as shown in Figure 18-12. Because the subtracter is a combinational circuit, whenever the numbers in the divisor and dividend registers change, these changes will propagate to the subtracter outputs. The borrow signal will propagate through the full subtracters before the subtracter output is transferred to the dividend register. If the last borrow signal (b_9) is 1, this means that the result is negative. Hence, if b_9 is 1, the divisor ($Y_3Y_2Y_1Y_0$) is greater than $X_8X_7X_6X_5X_4$, and $C = 0$. Therefore, $C = b_9'$, and a separate comparator circuit is unnecessary. Under normal operating conditions (no overflow) for this divider, we can also show that $C = d_8'$. At any subtraction step, because the divisor is only four bits, $d_8 = 1$ would allow a second subtraction without shifting. However, this can never occur because the quotient digit cannot be greater than 1. Therefore, if subtraction is possible, d_8 will always be 0 after the subtraction, so $d_8 = 0$ implies $X_8X_7X_6X_5X_4$ is greater than $Y_3Y_2Y_1Y_0$ and $C = d_8'$.

The block diagram of Figure 18-10 does not show how the dividend is initially loaded into the *X* register. This can be accomplished by adding a MUX at the *X* register inputs, as shown in Figure 18-13. This diagram uses bus notation to avoid drawing multiple wires. When several busses are merged together to form a single bus, a *bus merger* is used. For example, the symbol



means that the 5-bit subtracter output is merged with bits $X_3X_2X_1$ and a logic 1 to form a 9-bit bus. Thus, the MUX output will be $d_8d_7d_6d_5d_4X_3X_2X_11$ when Load = 0.

Similarly, the symbol



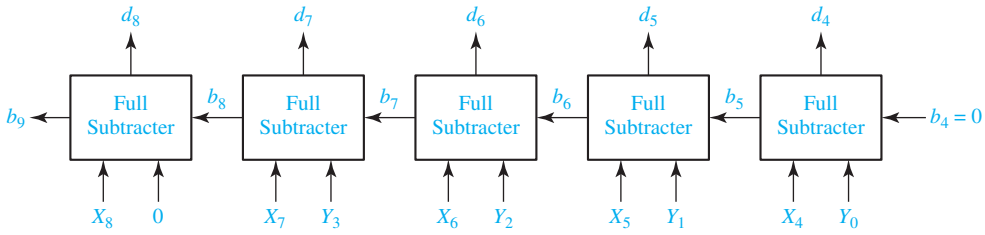


FIGURE 18-12
Logic Diagram for 5-Bit Subtracter

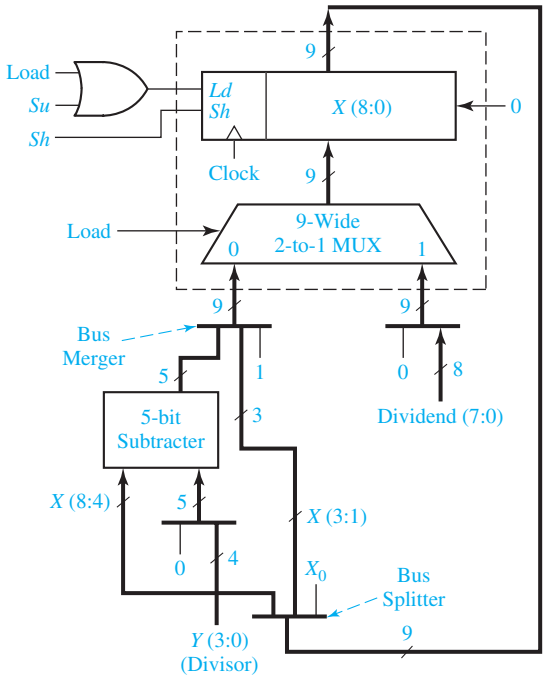


FIGURE 18-13
Block Diagram for Divider Using Bus Notation

represents a *bus splitter* that splits the 9 bits from the X register into $X_8X_7X_6X_5X_4$ and $X_3X_2X_1$; X_0 is not used. Bus mergers and splitters do not require any actual hardware; they are just a symbolic way of showing bus connections.

The X register is a left-shift register with parallel load capability, similar to the register in Figure 12-10. On the rising clock edge, it is loaded when Ld = 1 and shifted left when $Sh = 1$. Because the register must be loaded with the dividend when Load = 1 and with the subtracter output when $Su = 1$, Load and Su are ORed together and connected to the Ld input. The MUX selects the dividend (preceded by a 0) when Load = 1. When Load = 0, it selects the bus merger output which consists of the subtracter output, $X_3X_2X_1$, and a logic 1. When $Su = 1$ and the clock rises, this MUX output is loaded into X. The net result is that $X_8X_7X_6X_5X_4$ gets the subtracter output, $X_3X_2X_1$ is unchanged, and X_0 is set to 1.

Programmed Exercise 18.1

Cover the lower part of each page with a sheet of paper and slide it down as you check your answers. Write your answer in the space provided before looking at the correct answers.

Module 5.

Design of a Sequence Detector

A sequence detector is a sequential state machine whi takes an input string of bits and generates an outp '1' whenever the target sequence has been detected

Sequence detectors are of 2 types.

i) Overlapping

ii) Non-Overlapping.

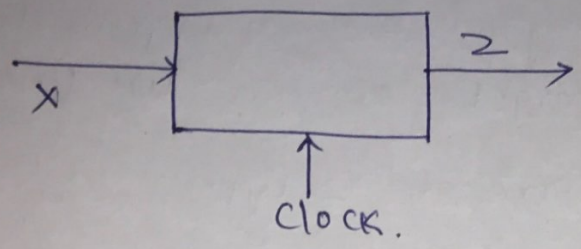
In overlapping sequence detector, the last bit of one sequence become the first bit of second sequence.

In Non-overlapping sequence detector, the last bit of previous sequence is not considered.

Steps in designing sequence detector

1. Develop the state diagram.
2. Code assignment
3. Make present state / Next state table
4. Draw the K-maps for Flip-Flop inputs.
5. Implement the circuit.

Let us consider the design of a clocked Mealy sequential circuit. The circuit has the following form.



a) The circuit will examine incoming string of 0's & 1's applied to the 'x' input. & generates an output $z=1$ only when a prescribed input sequence occurs.

b) It is assumed that 'x' can change only between clock pulses.

c) The circuit will be designed such that any input sequence ending in 101 will produce an output $z=1$.

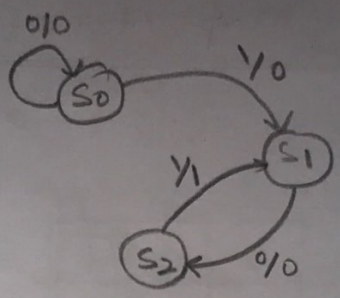
101

A typical input sequences & the corresponding output sequences are -

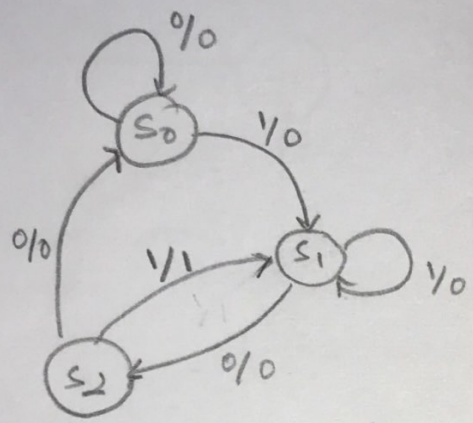
X =	0	0	1	1	0	1	1	0	0	1	0	1	0	0
op	0	0	0	0	0	1	0	0	0	0	0	1	0	0

State Graph

Step 1



Step. 2



State Transition Table

State	Present State A	B	Input	Next state		output	D_A	D_B
S_0	0	0	0	S_0	0 0	0	0	0
	0	0	1	S_1	0 1	0	0	1
S_1	0	1	0	S_2	1 0	0	1	0
	0	1	1	S_1	0 1	0	0	1
S_2	1	0	0	S_0	0 0	0	0	0
	1	0	1	S_1	0 1	1	0	1
S_3	1	1	0	x	x x	x	x	x
	1	1	1	x	x x	x	x	x

K- map for finding the Flip Flop Inputs

D_A

A \ BX	00	01	11	10
0	0	0	0	1
1	0	0	0	X

$D_A = B\bar{X}$

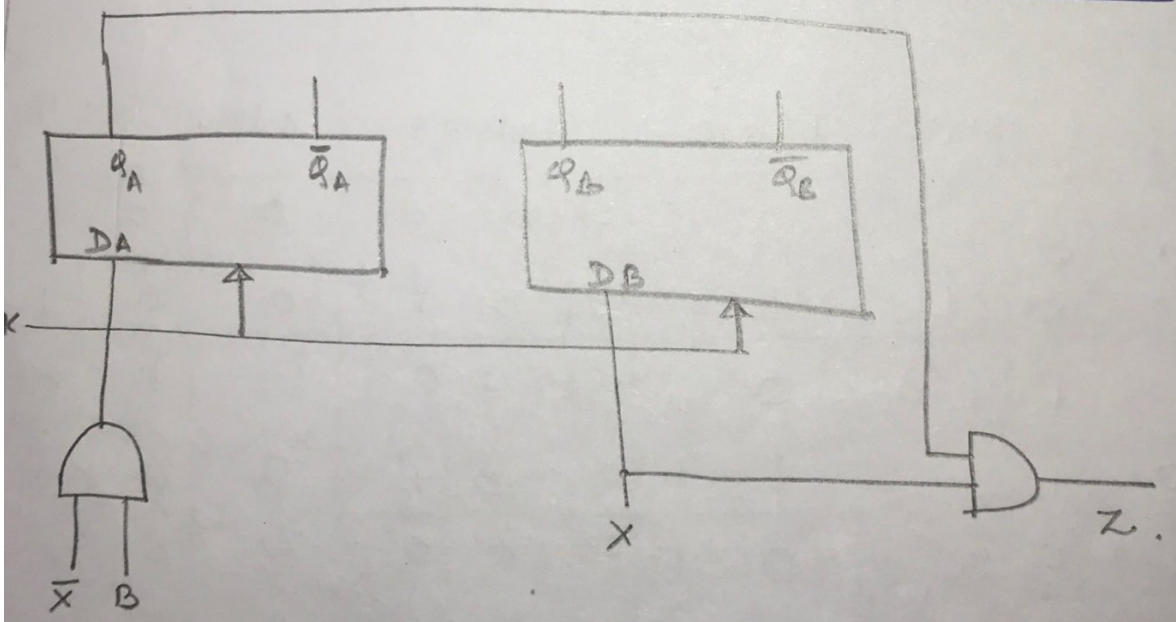
A \ BX	00	01	11	10
0	0	1	1	0
1	0	1	X	0

$D_B = X$.

A \ BX	00	01	11	10
0	0	0	0	0
1	0	1	X	X

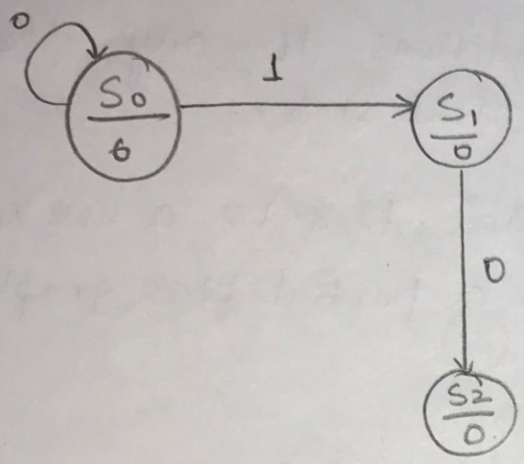
$Z = AX$.

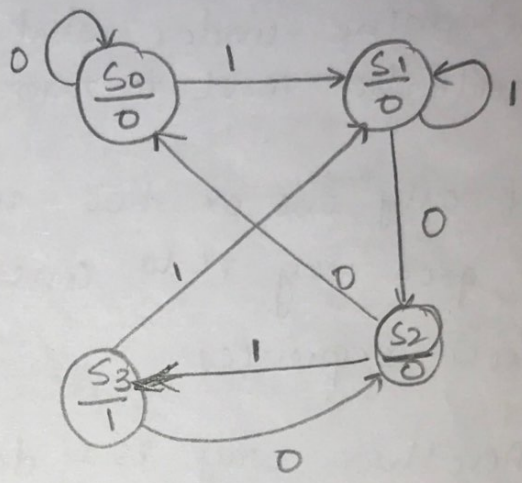
Implementation using Flip Flops & Gates.



The Stategraphs using Moore machine

The procedure for finding state graph using Moore machine is similar to Mealy machine except that the outputs are written inside the states instead of transitions.





3 Guidelines for constructing State Graphs.

Although there are no one specific procedure to derive state graphs. the following guidelines should prove helpful.

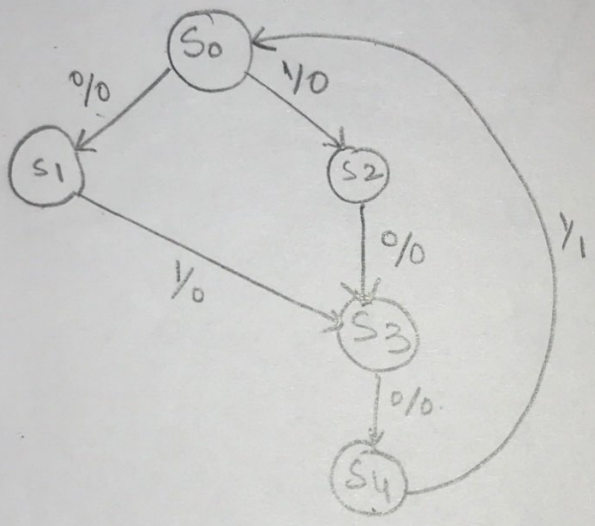
1. First construct some sample input & output sequences to understand the problem statement
2. Determine under what conditions, if any, the circuit should be reset to zero. (Initial state).
3. If only one or two sequences lead to a non zero output a good way is to construct a partial state graphs for those sequences.
4. Another way is to determine what sequences must be remembered by the ckt & set up states accordingly.
5. Each time an arrow to the graph is added, find it can go to previous states or new states.
6. check the graph to make sure there is one & only one path leaving each state for each combination of values of the input variables.
7. When the graph is constructed. check it by applying sequences of inputs.

Ex:

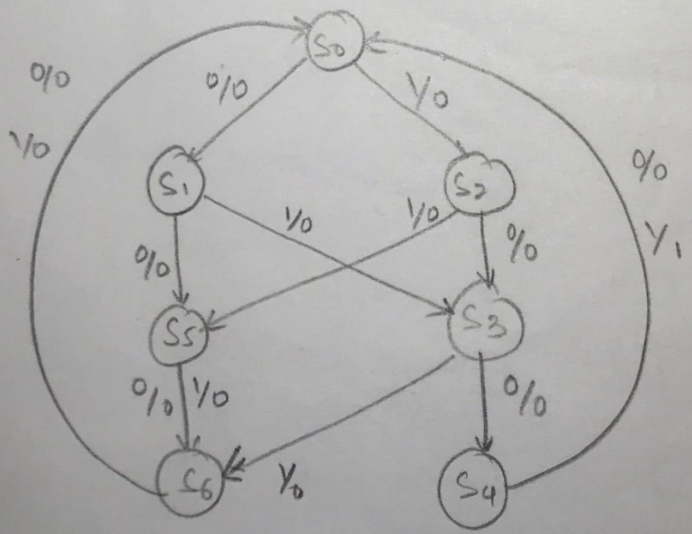
$X = \begin{array}{|cccc|cccc|cccc|cccc|} \hline 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ \hline \end{array}$

A sequential ckt

1001
0101



State	Reset
S0	
S1	0
S2	1
S3	01 or 10
S4	010 or 100
S5	
S6	



Design of Sequential circuit

1. Code converter. – BCD to Excess-3 code.

– This circuit adds three to a binary coded decimal digit in the range 0 to 9. The ip and op will be serial with the least significant bit first.

X Input BCD $t_3\ t_2\ t_1\ t_0$	Z output Excess-3 $t_3\ t_2\ t_1\ t_0$
0 0 0 0	0 0 1 1
0 0 0 1	0 1 0 0
0 0 1 0	0 1 0 1
0 0 1 1	0 1 1 0
0 1 0 0	0 1 1 1
0 1 0 1	1 0 0 0
0 1 1 0	1 0 0 1
0 1 1 1	1 0 1 0
1 0 0 0	1 0 1 1
1 0 0 1	1 1 0 0

– Table lists the desired inputs and outputs @ time t_0, t_1, t_2 & t_3.

– After receiving four ips, the circuit should reset to the initial state, ready to receive another group of 4 ips.

– we see that @ t_0 if the input is 0 the op is always 1. & if input is 1 the op is always 0. ∴ there is no conflict @ time t_0.

- At time t_1, 00 → 1 @ t_1
  01 → 0 @ t_1
  10 & 11 → 01 @ t_1 respectively.

- Therefore there is no conflict @ t_1.
- In a similar manner we can check to see there is no conflict @ t_2 and its all ips are available.

State table

- Input sequences are received with least significant bit first.
- Don't care (dashes) appear in this table because only 10 of the 16 possible 4-bit sequences can occur as inputs to the code converter.
- If the circuit is in state B @ t_1 and a 1 is received, this means that the sequence 10 has been received and the output should be 0.

Time	Input Sequence Received LSB	Present State	Next State x=0	Next State x=1	Present output z x=0	Present output z x=1
t_0	reset	A	B	C	1	0
t_1	0	B	D	E	1	0
	1	C	F	G	0	1
t_2	00	D	H	L	0	1
	01	E	I	M	1	0
	10	F	J	N	1	0
	11	G	K	P	1	0
t_3	000	H	A	A	0	1
	001	I	A	A	0	1
	010	J	A	–	0	–
	011	K	A	–	0	–
	100	L	A	–	0	–
	101	M	A	–	0	–
	110	N	A	–	0	–
	111	P	A	–	1	–

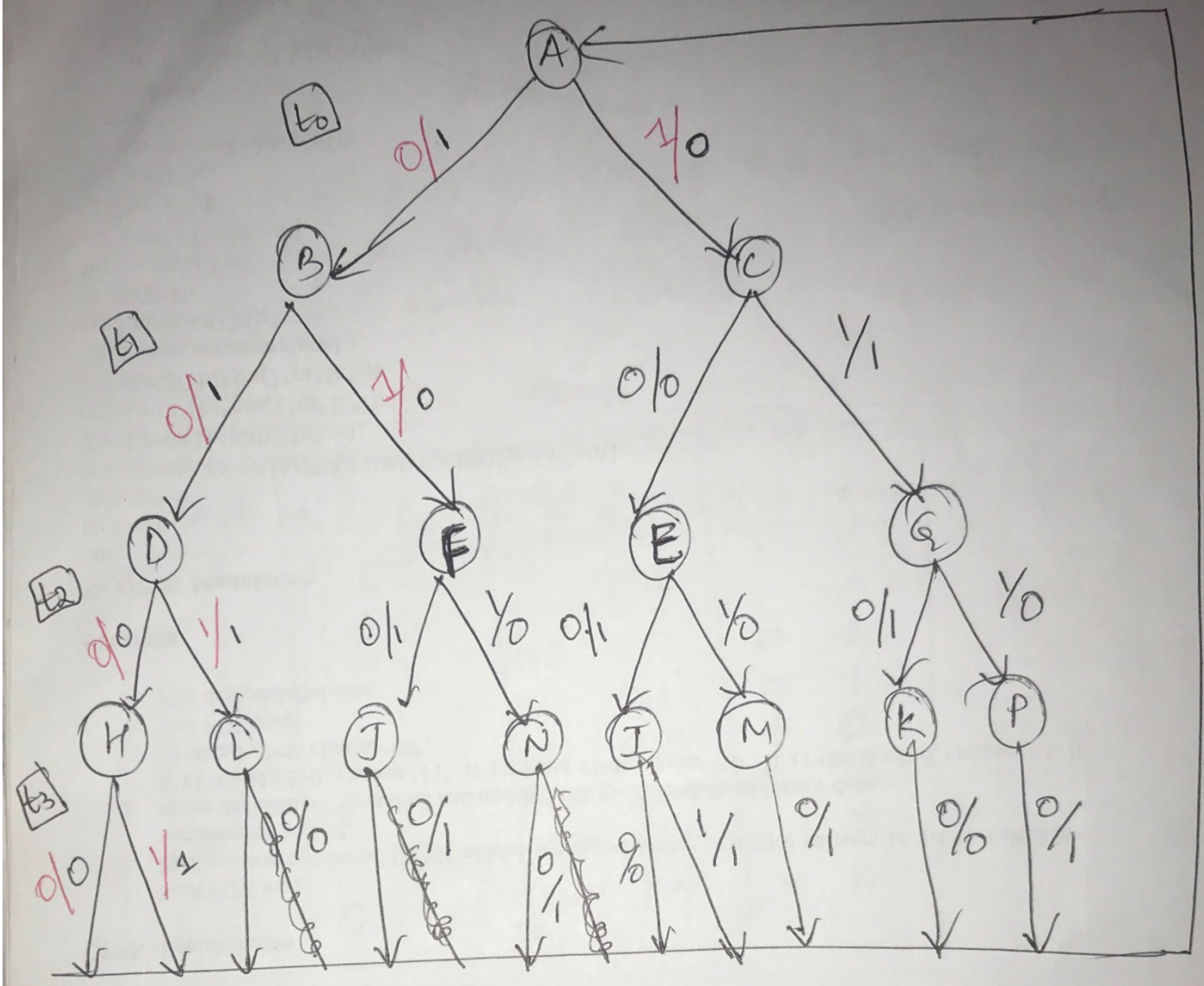
- Next we will reduce the table using row matching. When matching rows which contains dashes (don't cares), a dash will match with any state or with any output value.

- By matching rows in this manner, we have $H = J = K = L$ and $M = N = P$.
- After eliminating I, J, K, L, M, N and P, we find $E = F = G$ and the table reduces to 7 rows.

Reduced state table for code converter

Time	Present State	Next State X=0	Next State X=1	present output(Z) X=0	present output(Z) X=1
t_0	A	B	C	1	0
t_1	B	D	E	1	0
	C	E	E	0	1
t_2	D	H	H	0	1
	E	H	M	1	0
t_3	H	A	A	0	1
	M	A	–	1	–

State graph for the code converter



- The state graph has the form of a tree.
- Each path starting at the reset state represents one of the two possible input sequences.
- After the paths for the input sequences have been constructed, the outputs can be filled by working backwards along

each path.

- For example starting @ t_3, the path 0000 has outputs 0011 and 1000 has outputs 1001.

- 3 flipflops are required to realize the reduced table because there are seven states.

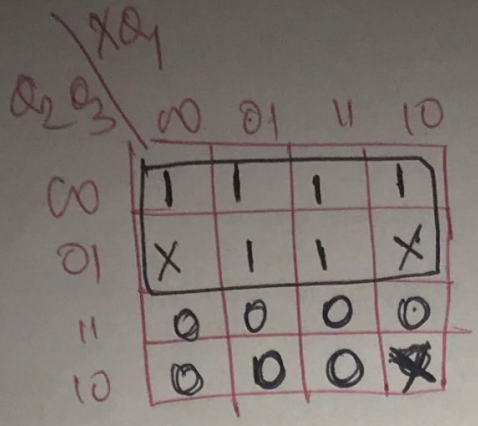
Transition table.

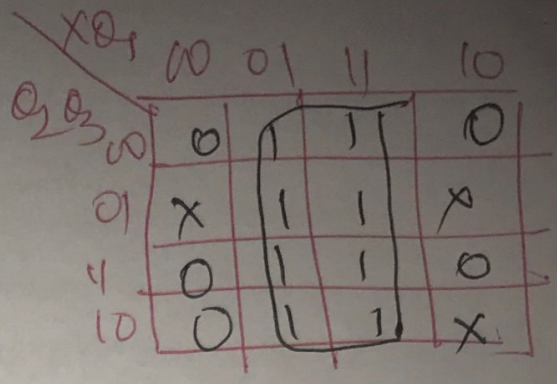
	state	PS $Q_1 Q_2 Q_3$	NS $Q_1^+ Q_2^+ Q_3^+$ x=0	NS x=1	Z x=0	Z x=1
0	A	000	100	101	1	0
4	B	100	111	110	1	0
5	C	101	110	110	0	1
7	D	111	011	011	0	1
6	E	110	011	010	0	0
3	H	011	000	000	0	0
2	M	010	000	xxx	1	x
1	—	001	xxx	xxx	x	x

$Q_1^+ Q_2^+ Q_3^+$ → using D flipflop → $D_1 D_2 D_3$

Assignment Map.

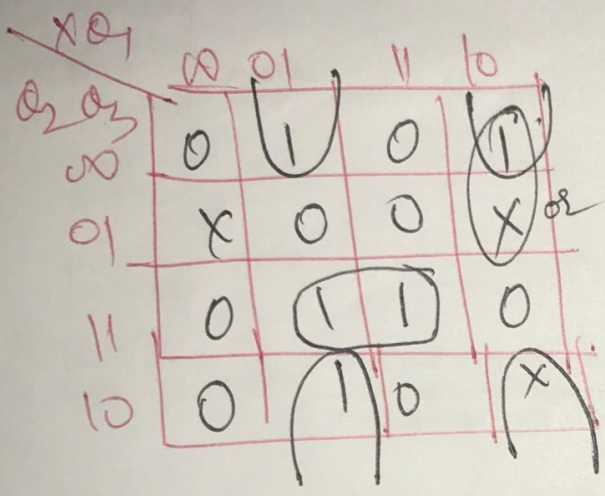
$Q_2 Q_3$ \ Q_1		
	A 0	B 4
	— 1	C 5
	H 3	D 7
	M 2	E 6

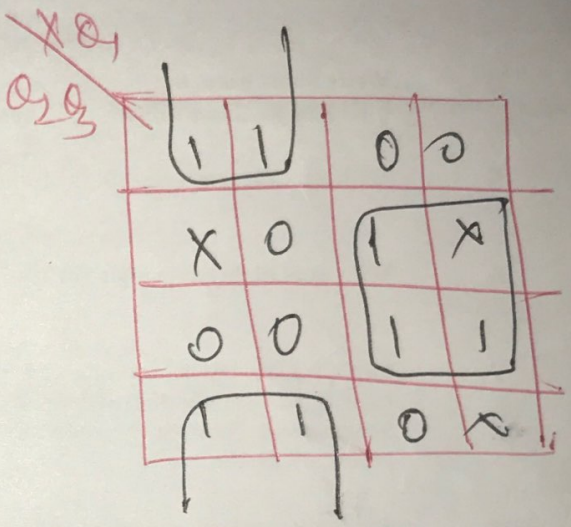




$Q_1^+ = D_1 = \overline{Q_2}$

$Q_2^+ = D_2 = Q_1$





$Q_3^+ = D_3 = Q_1Q_2Q_3 + X'Q_1Q_3' + XQ_1'Q_2'$

$Z = X'Q_3' + XQ_3$

Code converter circuit.

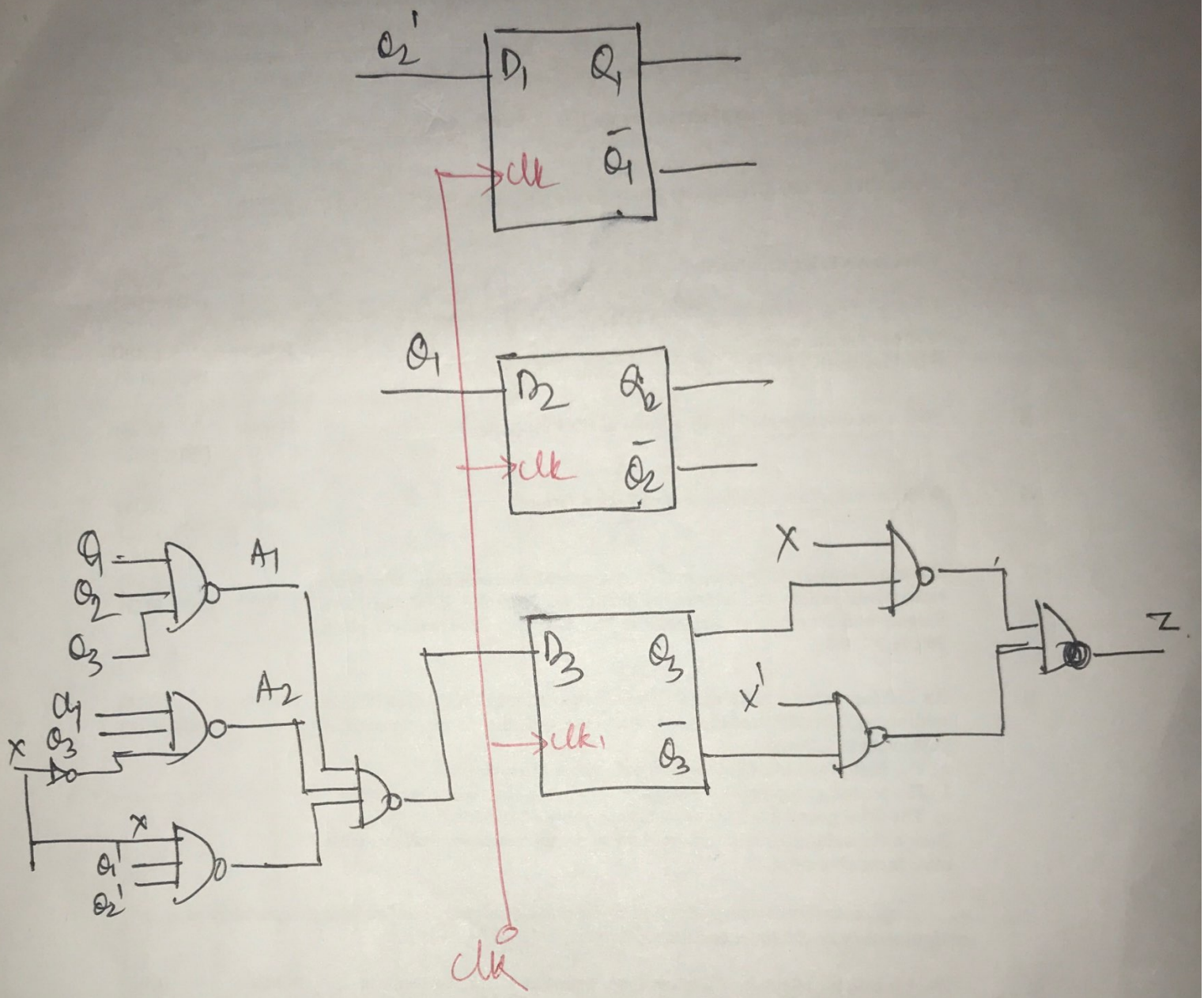


fig (1)

Design of sequential circuits using ROM's & PLA's.

① Code converter using ROM & D flipflops.

State table

Present state	Next state X=0	Next state X=1	Present state of (Z) X=0	Present state of (Z) X=1
A	B	C	1	0
B	D	E	1	0
C	E	E	0	1
D	H	H	0	1
E	H	M	1	0
H	A	A	0	1
M	A	—	1	—

Table ①

Transition table

State	$Q_1Q_2Q_3$	$Q_1^+Q_2^+Q_3^+$ X=0	$Q_1^+Q_2^+Q_3^+$ X=1	Z X=0	Z X=1
A	000	001	010	1	0
B	001	011	100	1	0
C	010	100	100	0	1
D	011	101	101	0	1
E	100	101	110	1	0
H	101	000	000	0	1
M	110	000	—	1	—

Table ②

Truth table

$X Q_1 Q_2 Q_3$	$Z \; D_1 D_2 D_3$
0 0 0 0	1 0 0 1
0 0 0 1	1 0 1 1
0 0 1 0	0 1 0 0
0 0 1 1	0 1 0 1
0 1 0 0	1 1 0 1
0 1 0 1	0 0 0 0
0 1 1 0	0 0 0 0
0 1 1 1	X X X
1 0 0 0	0 1 0 0
1 0 0 1	0 1 0 0
1 0 1 0	1 0 0 1
1 0 1 1	1 0 1 1
1 1 0 0	0 1 1 0
1 1 0 1	0 0 0 0
1 1 1 0	X X X
1 1 1 1	X X X

Table (3)

Realization of code converter using ROM

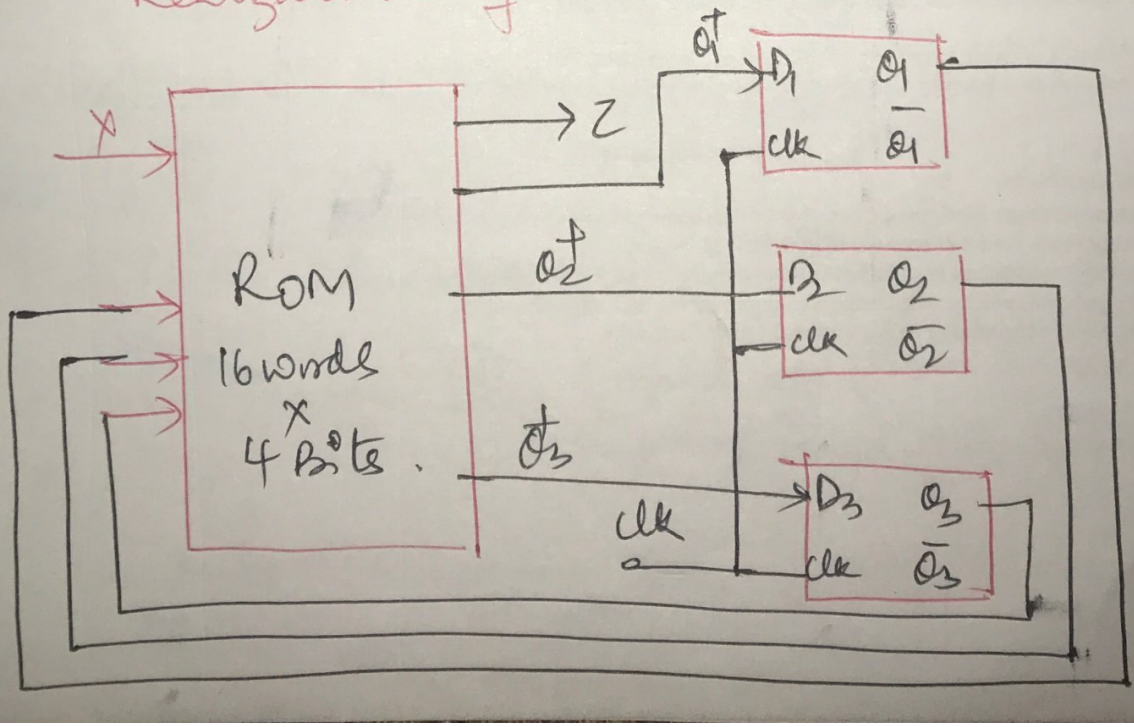


fig (2)

- As there are seven states, 3 D flipflops are required. Thus a ROM with 4-inputs (2^4 words) and four outputs is required.

- As we are using D flipflops $D_1 = Q_1^+$, $D_2 = Q_2^+$ and $D_3 = Q_3^+$.

- The truth table for ROM is as shown in Table (3).

- This table gives the ROM outputs $(Z, D_1, D_2 D_3)$ as functions of the ROM inputs (x, Q_1, Q_2, Q_3)

(2) Sequential circuit (code converter) using PLAs

The circuit configuration is same as fig (1) except that ROM is replaced with a PLA of appropriate size.

- The table (3) could be stored in a PLA with 4 inputs, 13 product terms and 4 outputs.

WKT

$$D_1 = Q_1^+ = Q_2'$$
$$D_2 = Q_2^+ = Q_1$$
$$D_3 = Q_3^+ = Q_1Q_2Q_3 + X'Q_1Q_3' + XQ_1'Q_2'$$
$$Z = X'Q_3' + XQ_3$$

- The PLA table which corresponds to these equations is in Table (4).

Realization of this table requires a PLA with 4 inputs, seven product terms and 4 outputs.

X Q_1 Q_2 Q_3	Z D_1 D_2 D_3
– – 0 –	0 1 0 0
– 1 – –	0 0 1 0
– 1 1 1	0 0 0 1
0 1 – 0	0 0 0 1
1 0 0 –	0 0 0 1
0 – – 0	1 0 0 0
1 – – 1	1 0 0 0

Table (4)

- we will verify the operation of the circuit in fig (1) pg.no ___ using a PLA which corresponds to table (4).

- Initially Assume that $X=0$ and $Q_1Q_2Q_3 = 000$
  This selects row $-0-$ & $Z=1$ &
  $Q_1Q_2Q_3 = 100$.
- After the active clock edge, $Q_1Q_2Q_3 = 100$,
  if $X=1$ then rows $--0-$ and
  $-1--$ are selected So $Z=0$ & $Q_1Q_2Q_3 = 110$.
- After the active clock edge, $Q_1Q_2Q_3 = 110$.
- Continuing this manner, we can verify the transition table. ⇒ Table (2)
- PALS are available which contain D flipflops that have their inputs driven from programmable array logic.
- Fig (3) shows a segment of a sequential PAL. The D flipflop is driven from an OR gate which is fed by 2 AND gates.
- The flipflop output is fedback to the programmable AND array through a buffer.
- Thus the AND gate inputs can be connected to A, A', B, B' or Q'.

- The X's (~~don't care~~) on the diagram shows the connections required to realize the next state equation.

$$Q^+ = D = A'B\bar{Q} + AB'Q.$$

- The flip flop op is connected to an inverting tri-state buffer, which is enabled when En = 1.

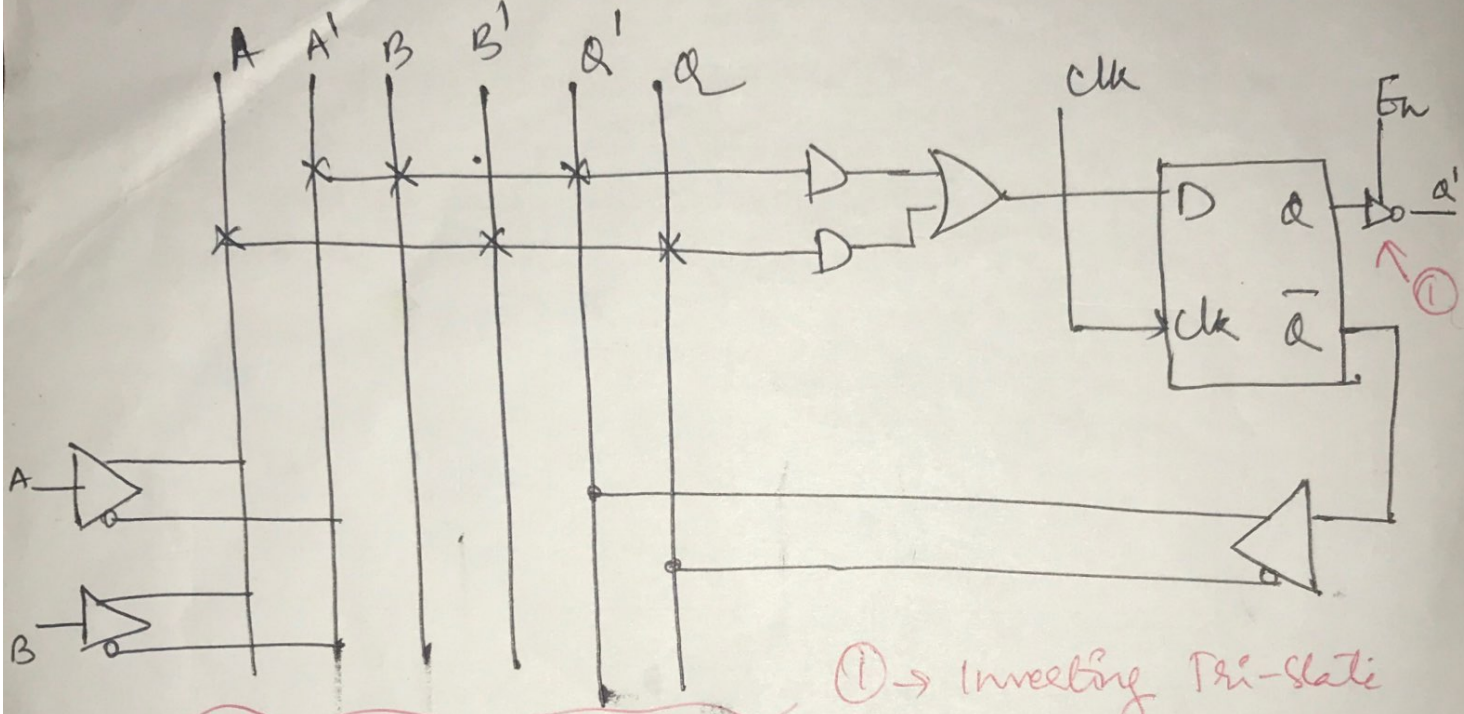


fig ③: segment of a sequential PAL.

Design of Binary Multiplier.

Design a multiplier for positive binary numbers. Binary multiplication requires only shifting and adding.

Consider the following example:

Multiplicand – 1 1 0 1

Multiplier – 1 0 1 1

1101 x 1 at 0th → 1 1 0 1

1101 x 1 at 1st → 1 1 0 1

1101 x 0 at 2nd → 0 0 0 0

1101 x 1 at 3rd → 1 1 0 1

1 0 0 0 1 1 1 1

answer

here we are adding upto 5 bits this may be eliminated by below method.

Multiplicand 1 1 0 1

Multiplier 1 0 1 1

1101 x 1 at 0th position → 1 1 0 1

1101 x 1 at 1st position → 1 1 0 1

partial product → 1 0 0 1 1 1

1101 x 0 at 2nd position 0 0 0 0

partial product → 1 0 0 1 1 1

1101 x 1 at 3rd pos → 1 1 0 1

1 0 0 0 1 1 1 1 ← answer

performed 2bit addition

Multiplication of two 4 bit numbers requires

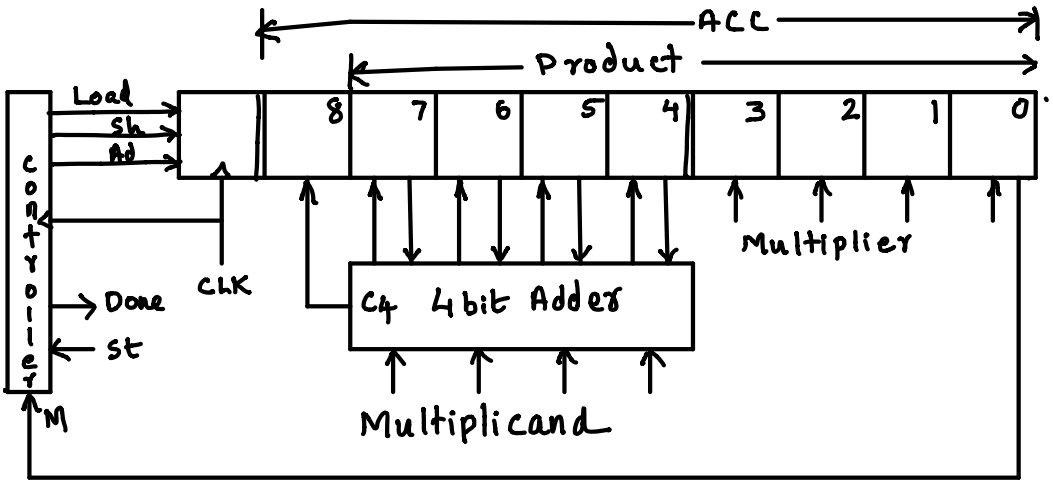
one 4 bit multiplicand register.

one 4 bit multiplier register

one 8 bit register for product.

The product register serves as an accumulator to accumulate the sum of the partial products Instead of shifting multiplicand left each time, it is convenient if we shift the product register to the right each time.

Block diagram of a parallel binary multiplier



4 bits from the accumulator and 4 bits from the multiplicand register are connected to the adder. the adder o/p (sum and carry) are connected back to accumulator.

When 'Ad' signal is given adder adds, and its o/p is stored in accumulator during next clock.

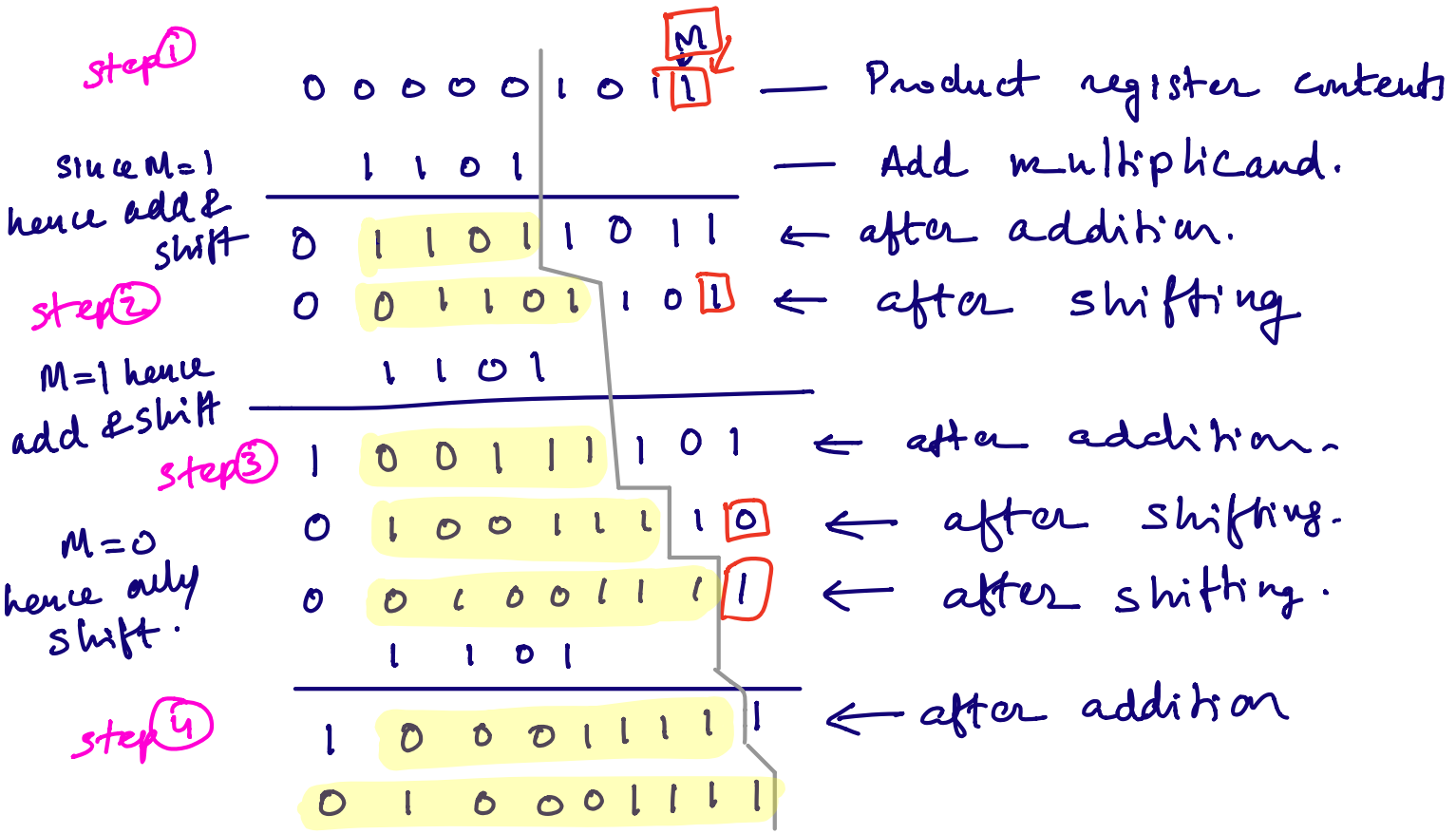
Load signal loads the multiplier into the lower four bits of ACC and clears the upper bits.

Shift signal causes the content of the product register to be shifted one place to the right.

Control Circuit put the proper sequence of add and shift signals after St=1.
If current multiplier bit is 1 (M=1) multiplicand is added to the accumulator followed by right shift. if M=0, addition is skipped and only right shift occurs.

Consider following example.

11 (1011) x 13 (1101) = 143 (10001111)



In step 1: Last four bits of the product register is loaded with the multiplier values (eg: 1011)

00000 1011 ← M
1101

Since 'M' (ie, last bit) is 1, the multiplicand values (eg: 1101) are added to the 4 bits in the MSB of the product register. (initially these bits are zero).

Step 2: The product register now contains the results of addition in upper nibble, lower nibble is the multiplier values. Hence after addition, the product register values are (in this example) 1101 1011.

Step 3. In this step, the product register is shifted to the right by one bit. In our example, after shift, the product register is 0110 1101 (1) ← Mbit

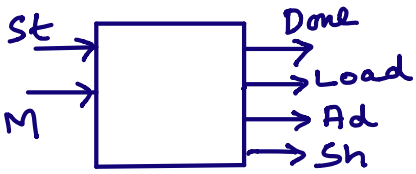
Step 4. After shifting the 'M' bit is checked. Since this bit is '1' in our example, once again the multiplicand is added to upper four bits. i.e, 0110 1101 + 1101 = 1001 1101. Hence after addition the product register is 0011 1101.

Step 5: The product register shifted right again and the value is 1001 1110 [0] and M bit is 0. Hence addition process is skipped and product register will be shifted again and new value is 0100 1111 [1]

Step 6: Since M bit is '1' the multiplicand is added and product register will be shifted. Hence after 4 shifting operation the product register containg 1000 1111 which is the result.

Design of control circuit for the binary multiplier.

The control circuit is designed to output the proper sequence of add (Ad) and shift (Sh) signals. The state graph of control circuit is shown below.



The notation used is

input / o/p ⇒ St M / Ad Sh

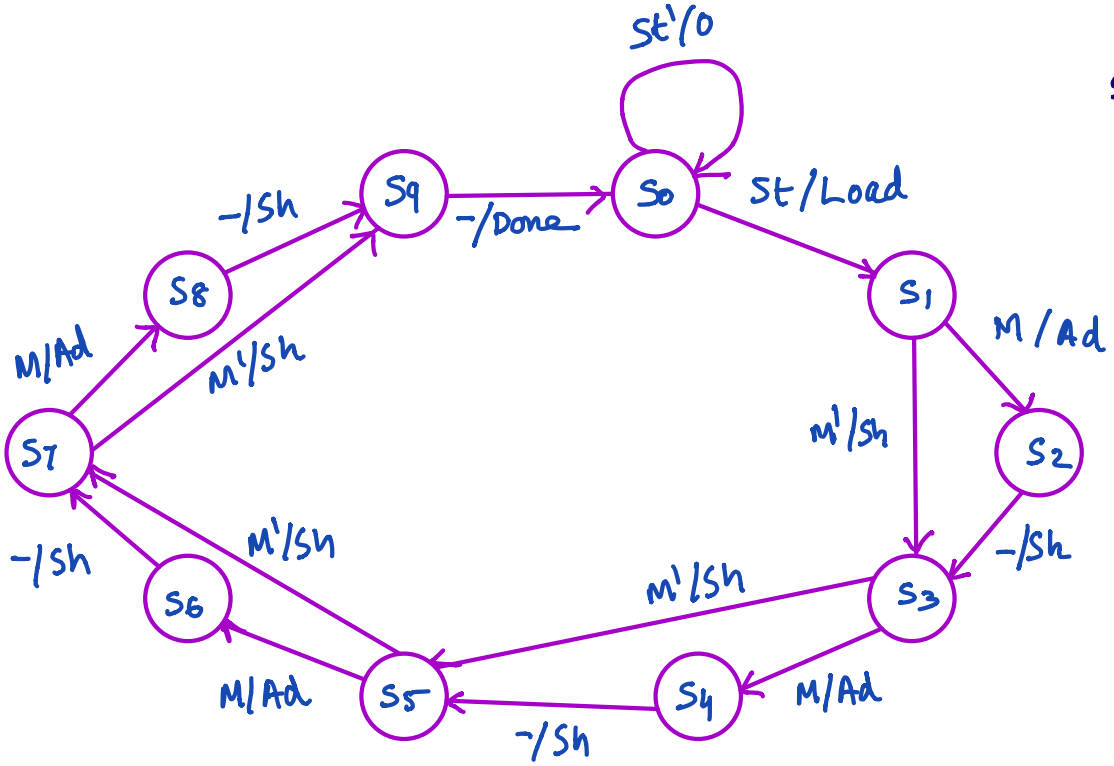
means St=1, M=1
Ad=1, Sh=1

St / Sh means St=1, Sh=1

all other values are zero.

M' / Sh – means M=0, Sh=1 and all other values are zero.

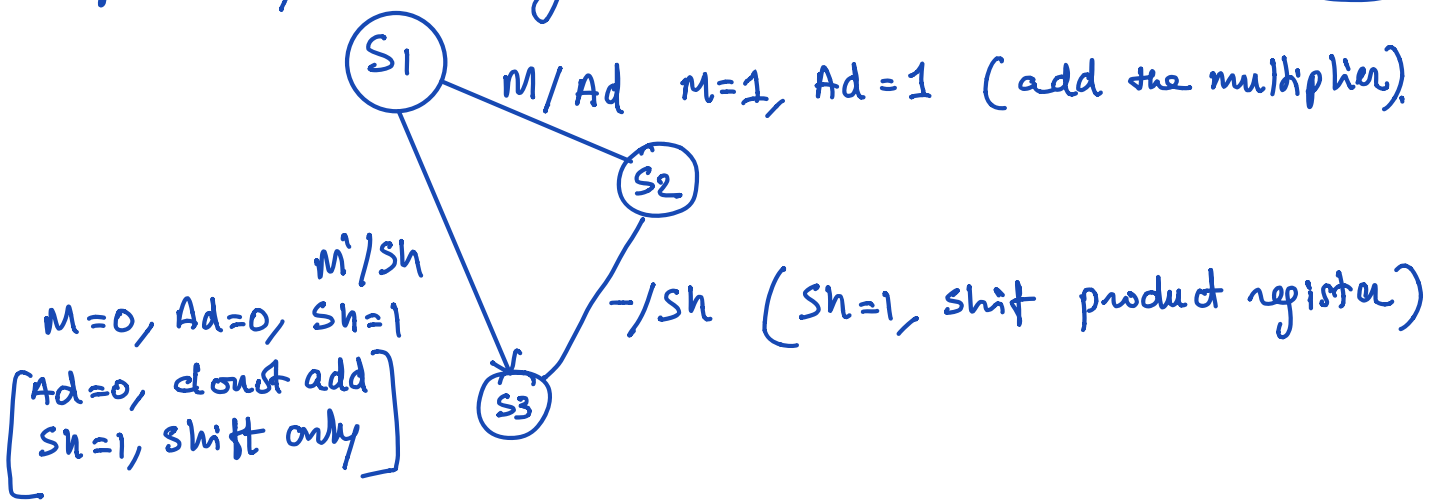
S_0 – reset state
S_1 – M is tested



S_0 at next state, if St=0, it will remain in same state till St=1.

Once $St=1$, product register will be loaded with values and multiplicand is available at adder inputs. Hence Load = 1. (St/Load).

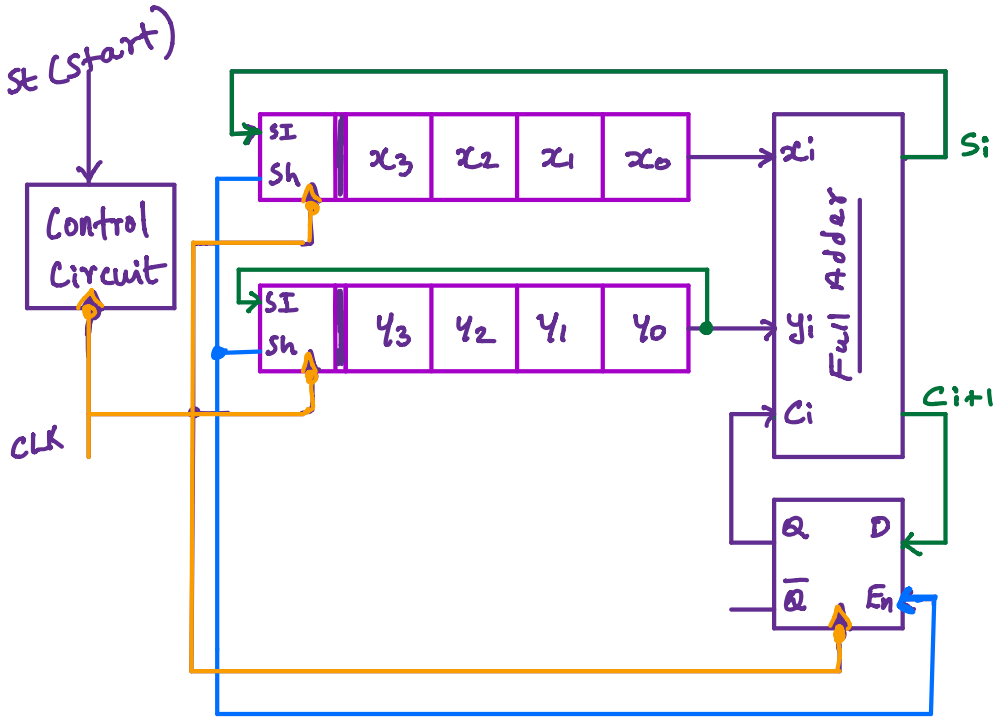
S_1 at this state, circuit will check for M bit. if $M=1$, the addition will hapen ($Ad=1$) and will move to next state S_2. later will shift the product register by making $Sh=1$ and move to S_3. If $M=0$, the addition will be skipped at will only shift the product register and will move to S_3



The same sequence will be continued till S_9 at S_9 the control circuit goes to Done state and terminates the multiplication process.

Serial Adder with Accumulator.

The figure below shows the block diagram of serial adder with accumulator.



Two shift registers are used to hold the 4-bit numbers to be added, 'X' and 'Y'. X register is accumulator and Y register is addend register.

When addition is completed, content of 'X' register is replaced with the sum of X and Y.

Y register is a cyclic shift register such that at the end of four clock pulses, the value in addend register is back to original state. 'SI' is serial input, 'Sh' is shift control signal

Sh = 1 allows the value at 'SI' to enter into the shift register and values at x_3, x_2, x_1 will get shifted right and x_0 value will be available at the input of full adder. A similar operation happens to the Y register also. The result of addition of x_0 and y_0 will be Sum = S_0 and

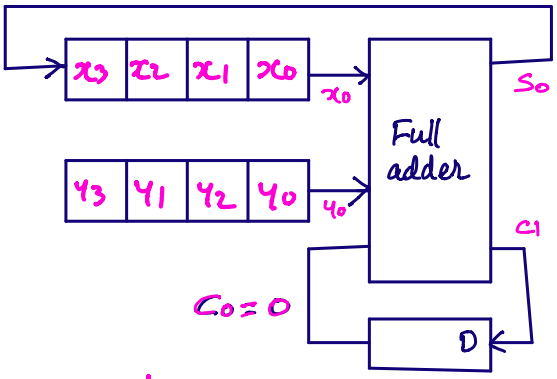
Carry = C_1 (This carry is fed as carry-in while adding x_1 and y_1.)

Since C_1 need to be added with x_1 and y_1, C_1 will be stored in 'D' register and will arrive at the input of full adder during next clock pulse only.

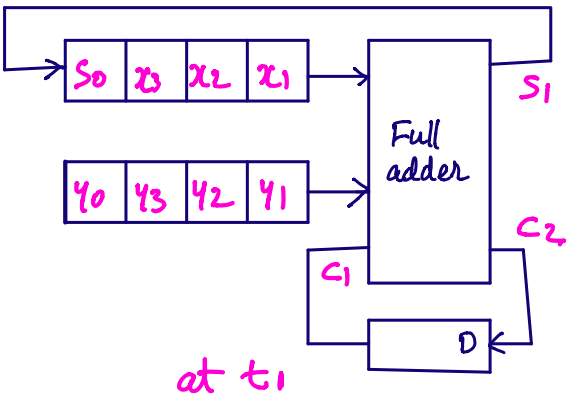
At every clock pulse, the next lsb of X and Y will arrive at the input of full adder and will be added along with carry generated by previous bit's addition.

The figure below shows a numerical example for the same.

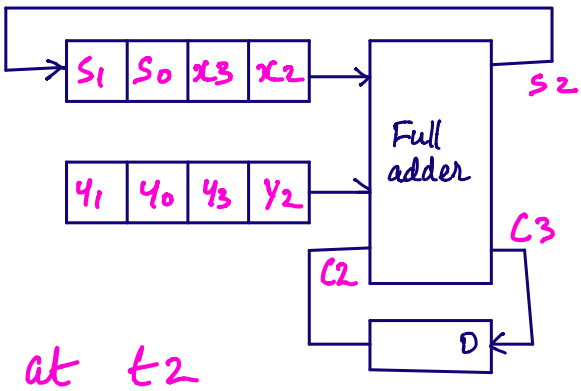
	X	Y	C_i	S_i	C_i^+
t_0	0101	0111	0	0	1
t_1	0010	1011	1	0	1
t_2	0001	1101	1	1	1
t_3	1000	1110	1	1	0
t_4	1100	0111	0	(1)	(0)



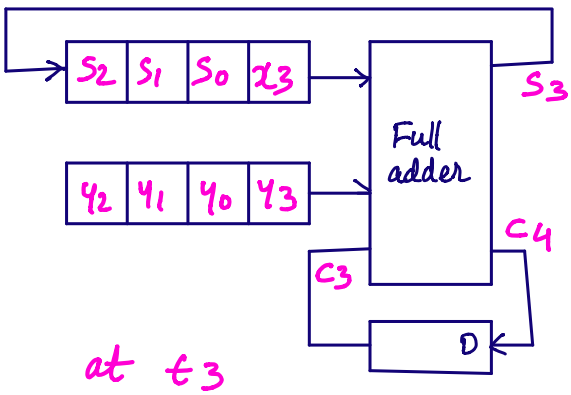
at t_0



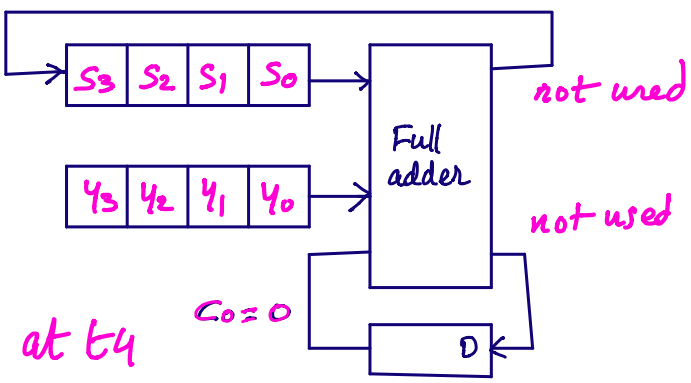
at t_1



at t_2



at t_3



at t_4

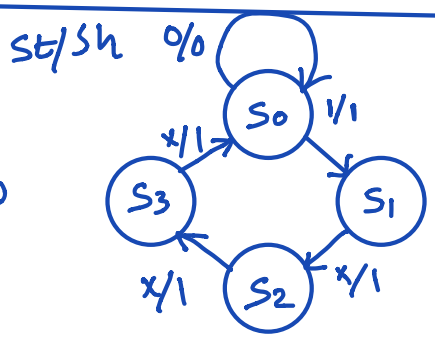
at t_4, accumulator is filled with 'Sum' o/p.

Design of control circuit for serial adder:

After receiving a start signal (st=1) control circuit will put out four shift signals (sh=1) and stops (st=0)

The state diagram is shown here:

st- state input
sh- shift signal o/p



Let S_0 = A B = 0 0

S_1 = 0 1

S_2 = 1 0

S_3 = 1 1

St	A	B	A^+	B^+	Sh
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	0	0	1

D_A (A^+)

St \ AB	00	01	11	10
0	0	1	0	1
1	0	1	0	1

$D_A = \bar{A}B + A\bar{B} = A \oplus B$

D_B (B^+)

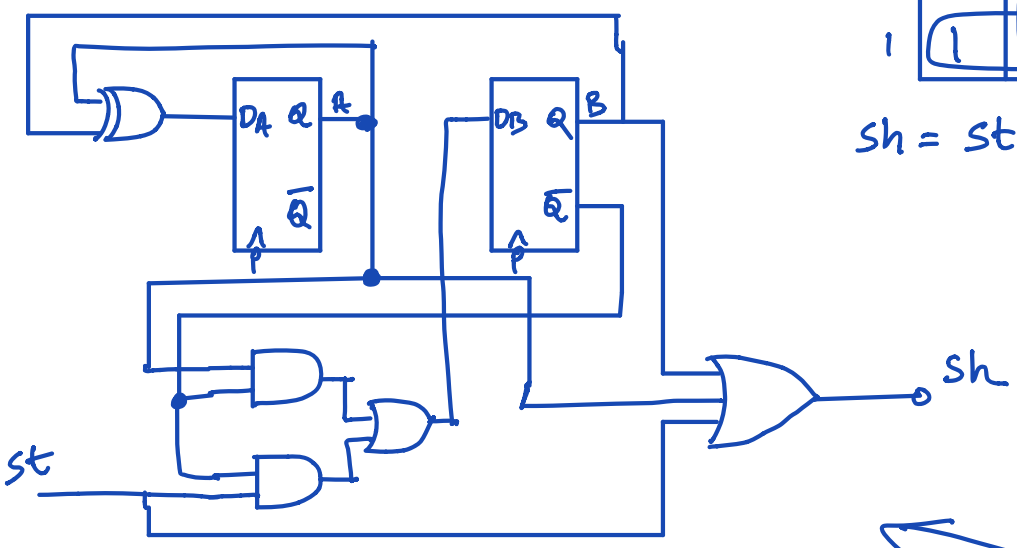
St \ AB	00	01	11	10
0	0	0	0	1
1	1	0	0	1

$D_B = St\,\bar{B} + A\bar{B}$

Sh

St \ AB	00	01	11	10
0	0	1	1	1
1	1	1	1	1

$Sh = St + A + B$



Realization of Controller Circuit