DIGITAL SYSTEM DESIGN

INDEX

UNIT-I

FUNDAMENTALS OF DIGITAL TECHNIQUES

NUMBER SYSTEMS & CODES

- Philosophy of number systems
- Complement representation of negative numbers
- Binary arithmetic
- Binary codes
- Error detecting & error correcting codes:Hamming codes

HISTORY OF THE NUMERAL SYSTEMS:

A **numeral system** (or **system of numeration**) is a linguistic system and mathematical notation for representing numbers of a given set by symbols in a consistent manner. For example, It allows the numeral "11" to be interpreted as the binary numeral for *three*, the decimal numeral for *eleven*, or other numbers in different bases. Ideally, a numeral system will:

- Represent a useful set of numbers (e.g. all whole numbers, integers, or real numbers)
- Give every number represented a unique representation (or at least a standard representation)
- Reflect the algebraic and arithmetic structure of the numbers.

For example, the usual decimal representation of whole numbers gives every whole number a unique representation as a finite sequence of digits, with the operations of arithmetic (addition, subtraction, multiplication and division) being present as the standard algorithms of arithmetic. However, when decimal representation is used for the rational or real numbers, the representation is no longer unique: many rational numbers have two numerals, a standard one that terminates, such as 2.31, and another that recurs, such as 2.309999999 Numerals which terminate have no non-zero digits after a given position. For example,numerals like **2.31 and 2.310 are taken** to be the same, except in the experimental sciences, where greater precision is denoted by the trailing zero.

The most commonly used system of numerals is known as Hindu-Arabic numerals.Great Indian mathematicians Aryabhatta of Kusumapura (5th Century) developed the place value notation.

Brahmagupta (6th Century) introduced the symbol zero.

BINARY

The ancient Indian writer Pingala developed advanced mathematical concepts for describing prosody, and in doing so presented the first known description of a binary numeral system.A full set of 8 trigrams and 64 hexagrams, analogous to the 3-bit and 6-bit binary numerals, were known to the ancient Chinese in the classic text *I Ching*. An arrangement of the hexagrams of the *I Ching*, ordered according to the values of the corresponding binary numbers (from 0 to 63), and a method for generating thesame, was developed by the Chinese scholar and philosopher Shao Yong in the 11th century.

In 1854, British mathematician George Boole published a landmark paper detailing an algebraic system of logic that would become known as Boolean algebra. His logical calculus was to become instrumental in the design of digital electronic circuitry. In 1937, Claude Shannon produced his master's thesis at MIT that implemented Boolean algebra and binary arithmetic using electronic relays and switches for the first time in history. Entitled *A SymbolicAnalysis of Relay and Switching Circuits*, Shannon's thesis essentially founded practical digital circuit design.

Binary codes

Binary codes are codes which are represented in binary system with modification from the original ones.

- Weighted Binary codes
- Non Weighted Codes

Weighted binary codes are those which obey the positional weighting principles, each position of the number represents a specific weight. The binary counting sequence is an example.

Decimal	BCD 8421	Excess-3	84-2-1	2421	5211	Bi-Quinary 5043210			5	0	4	3	2	1	0
0	0000	0011	0000	0000	0000	0100001		0		X					X
1	0001	0100	0111	0001	0001	0100010		1		X				X	
2	0010	0101	0110	0010	0011	0100100		2		X			X		
3	0011	0110	0101	0011	0101	0101000		3		X		X			
4	0100	0111	0100	0100	0111	0110000		4		X	X				
5	0101	1000	1011	1011	1000	1000001		5	X						X
6	0110	1001	1010	1100	1010	1000010		6	X					X	
7	0111	1010	1001	1101	1100	1000100		7	X				X		
8	1000	1011	1000	1110	1110	1001000		8	X			X			
9	1001	1111	1111	1111	1111	1010000		9	X		X				

Reflective Code

A code is said to be reflective when code for 9 is complement for the code for 0, and so is for 8 and 1 codes, 7 and 2, 6 and 3, 5 and 4. Codes 2421, 5211, and excess-3 are reflective, whereas the 8421 code is not.

Sequential Codes

A code is said to be sequential when two subsequent codes, seen as numbers in binary representation, differ by one. This greatly aids mathematical manipulation of data. The 8421 and Excess-3 codes are sequential, whereas the 2421 and 5211 codes are not.

Non weighted codes

Non weighted codes are codes that are not positionally weighted. That is, each position within the binary number is not assigned a fixed value. Ex: Excess-3 code

Excess-3 Code

Excess-3 is a non weighted code used to express decimal numbers. The code derives its name from the fact that each binary code is the corresponding 8421 code plus 0011(3).

Gray Code

The gray code belongs to a class of codes called minimum change codes, in which only one bit in the code changes when moving from one code to the next. The Gray code is non-weighted code, as the position of bit does not contain any weight. The gray code is a reflective digital code which has the special property that any two subsequent numbers codes differ by only one bit. This is also called a unitdistance code. In digital Gray code has got a special place.

Decimal Number	Binary Code	Gray Code	Decimal Number	Binary Code	Gray Code
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

Binary to Gray Conversion

- Gray Code MSB is binary code MSB.
- Gray Code MSB-1 is the XOR of binary code MSB and MSB-1.
- MSB-2 bit of gray code is XOR of MSB-1 and MSB-2 bit of binary code.
- MSB-N bit of gray code is XOR of MSB-N-1 and MSB-N bit of binary code.

Error detection codes

1) Parity bits

A **parity bit** is a bit that is added to a group of source bits to ensure that the number of set bits (i.e., bits with value 1) in the outcome is even or odd. It is a very simple scheme that can be used to detect single or any other odd number (i.e., three, five, etc.) of errors in the output. An even number of flipped bits will make the parity bit appear correct even though the data is erroneous.

2) Checksums

A **checksum** of a message is a modular arithmetic sum of message code words of a fixed word length (e.g.,byte values). The sum may be negated by means of a one's-complement prior to transmission to detect errorsresulting in all-zero messages.Checksum schemes include parity bits, check digits, and longitudinal redundancy checks. Some checksum schemes, such as the Luhn algorithm and the Verhoeff algorithm, are specifically designed to detect errorscommonly introduced by humans in writing down or remembering identification numbers.

3) Cyclic redundancy checks (CRCs)

A **cyclic redundancy check (CRC)** is a single-burst-error-detecting cyclic code and non-secure hash function designed to detect accidental changes to digital data in computer networks. It is characterized by specification of a so-called *generator polynomial*, which is used as the divisor in a polynomial long division over a finite field, taking the input data as the dividend, and where the remainder becomes the result.Cyclic codes have favorable properties in that they are well suited for detecting burst errors. CRCs are particularly easy to implement in hardware, and are therefore commonly used in digital networks and storage devices such as hard disk drives.Even parity is a special case of a cyclic redundancy check, where the single-bit CRC is generated by the divisor $x+1$.

NUMBER BASE CONVERSIONS

Any number in one base system can be converted into another base system Types

1) decimal to any base
2) Any base to decimal

3) Any base to Any base

Decimal number: $123.45 = 1\,10^2 + 2\,10^1 + 3\,10^0 + 4\,10^{-1} + 5\,10^{-2}$

Base b number: $N = a_{q-1}b^{q-1} + \cdots + a_0 b^0 + \cdots + a_{-p}b^{-p}$

$b > 1, \quad 0 <= a_i <= b-1$

Integer part: $a_{q-1}a_{q-2} \cdots a_0$

Fractional part: $a_{-1}a_{-2} \cdots a_{-p}$

Most significant digit: a_{q-1}

Least significant digit: a_{-p}

Binary number (b=2): $1101.01 = 1\,2^3 + 1\,2^2 + 0\,2^1 + 1\,2^0 + 0\,2^{-1} + 1\,2^{-2}$

Representing number N in base b: $(N)_b$

Complement of digit a: $a' = (b-1)-a$

Decimal system: 9's complement of 3 = 9-3 = 6

Binary system: 1's complement of 1 = 1-1 = 0

Fractional number:

$$(N)_{b_1} = a_{-1}b_2^{-1} + a_{-2}b_2^{-2} + \cdots + a_{-p}b_2^{-p}$$

$$b_2 \cdot (N)_{b_1} = a_{-1} + a_{-2}b_2^{-1} + \cdots + a_{-p}b_2^{-p+1}$$

Example: Convert $(0.3125)_{10}$ to base 8

0.3125 × 8 = 2.5000 hence $a_{-1} = 2$

0.5000 × 8 = 4.0000 hence $a_{-2} = 4$

Thus, $(0.3125)_{10} = (0.24)_8$

Decimal to Binary

Example: Convert $(432.354)_{10}$ to binary

Q_i	r_i
216	$0 = a_0$
108	$0 = a_1$
54	$0 = a_2$
27	$0 = a_3$
13	$1 = a_4$
6	$1 = a_5$
3	$0 = a_6$
1	$1 = a_7$
	$1 = a_8$

$0.354 \times 2 = 0.708$ hence $a_{-1} = 0$
$0.708 \times 2 = 1.416$ hence $a_{-2} = 1$
$0.416 \times 2 = 0.832$ hence $a_{-3} = 0$
$0.832 \times 2 = 1.664$ hence $a_{-4} = 1$
$0.664 \times 2 = 1.328$ hence $a_{-5} = 1$
$0.328 \times 2 = 0.656$ hence $a_{-6} = 0$
$a_{-7} = 1$
etc.

Thus, $(432.354)_{10} = (110110000.0101101\ldots)_2$

Octal To Binary

Example: Convert $(123.4)_8$ to binary

$(123.4)_8 = (001\ 010\ 011.100)_2$

Example: Convert $(1010110.0101)_2$ to octal

$(1010110.0101)_2 = (001\ 010\ 110.010\ 100)_2 = (126.24)_8$

Error Detection and Correction Codes

- No communication channel or storage device is completely error-free
- As the number of bits per area or the transmission rate increases, more errors occur. • Impossible to detect or correct 100% of the errors **Hamming Codes**

1. One of the most effective codes for error-recovery
2. Used in situations where random errors are likely to occur
3. Error detection and correction increases in proportion to the number of parity bits (error-checking bits) added to the end of the information bits

code word = information bits + parity bits

Hamming distance: the number of bit positions in which two code words differ.

10001001

10110001

* * *

Minimum Hamming distance or D(min) : determines its error detecting and correcting capability.

4) Hamming codes can always detect D(min) – 1 errors, but can only correct half of those errors.

EX.

Data Bits	Parity Bit	Code Word
00	0	000
01	1	011
10	1	101
11	0	110

000*	100
001	101*
010	110*
011*	111

5) Single parity bit can only detect error, not correct it

6) Error-correcting codes require more than a single parity bit EX. 0 0 0 0 0

0 1 0 1 1

1 0 1 1 0

1 1 1 0 1

Minimum Hamming distance = 3

Can detect up to 2 errors and correct 1 error
Cyclic Redundancy Check

1. Let the information byte F = 1001011
2. The sender and receiver agree on an arbitrary binary pattern P. Let P = 1011.
3. Shift F to the left by 1 less than the number of bits in P. Now, F = 1001011000.

4. Let F be the dividend and P be the divisor. Perform “modulo 2 division”.
5. After performing the division, we ignore the quotient. We got 100 for the remainder, which becomes the actual CRC checksum.
6. Add the remainder to F, giving the message M: 1001011 + 100 =

M is decoded and checked by the message receiver using the reverse process.

```
      ____1010100
1011 | 1001011100
       1011
       001001
        1001
        0010
        001011
          1011
          0000          ← Remainder
```

1001011100 = M

UNIT-II

BOOLEAN ALGEBRA AND THEOREMS

- **Fundamental postulates of Boolean algebra**
- **Basic theorems and properties**
- **Switching functions**
- **Canonical and Standard forms**
- **Algebraic simplification digital logic gates, properties of XOR gates**
- **Universal gates**
- **Multilevel NAND/NOR realizations**

Boolean Algebra: Boolean algebra, like any other deductive mathematical system, may be defined with aset of elements, a set of operators, and a number of unproved axioms or postulates. A *set* of elements is anycollection of objects having a common property. If **S** is a set and ***x*** and ***y*** are certain

objects, then **x** Î **S**denotes that ***x*** is a member of the set **S**, and ***y*** Ï**S** denotes that ***y*** is not an element of **S**. A set with adenumerable number of elements is specified by braces: **A** = {1,2,3,4}, *i.e.* the elements of set **A** are thenumbers 1, 2, 3, and 4. A *binary operator* defined on a set S of elements is a rule that assigns to each pair ofelements from S a unique element from S._ Example: In *a*b=c*, we say that * is a binary operator if it specifies a rule for finding *c* from the pair (*a*,*b*)and also if *a*, *b*, *c* Î S.

CLOSURE: The Boolean system is *closed* with respect to a binary operator if for every pair of Boolean values,it produces a Boolean result. For example, logical AND is closed in the Boolean system because it accepts only Boolean operands and produces only Boolean results.

A set *S* is closed with respect to a binary operator if, for every pair of elements of *S*, the binary operator specifies a rule for obtaining a unique element of *S*.

For example, the set of natural numbers N = {1, 2, 3, 4, … 9} is closed with respect to the binary operator plus (+) by the rule of arithmetic addition, since for any *a*, *b* Î N we obtain a unique *c* Î N by the operation *a* + *b* = c.

ASSOCIATIVE LAW:

A binary operator * on a set *S* is said to be associative whenever (*x* * *y*) * *z* = *x* * (*y* * *z*) for all *x*, *y*, *z* Î S, forall Boolean values x, y and z.

COMMUTATIVE LAW:

A binary operator * on a set *S* is said to be commutative whenever x * *y* = *y* * *x* for all *x*, *y*, *z* ϵ S

IDENTITY ELEMENT:

A set ***S*** is said to have an identity element with respect to a binary operation * on S if there exists an element *e* ϵ S with the property *e* * *x* = *x* * *e* = *x* for every *x* ϵ S

BASIC IDENTITIES OF BOOLEAN ALGEBRA

- *Postulate 1(Definition)*: A Boolean algebra is a closed algebraic system containing a set *K* of two or more elements and the two operators · and + which refer to logical AND and logical OR •*x* + *0* = *x*

- $x \cdot 0 = 0$
- $x + 1 = 1$
- $x \cdot 1 = 1$
- $x + x = x$
- $x \cdot x = x$
- $x + x' = x$
- $x \cdot x' = 0$
- $x + y = y + x$
- $xy = yx$
- $x + (y + z) = (x + y) + z$
- $x\,(yz) = (xy)\,z$
- $x (y + z) = xy + xz$
- $x + yz = (x + y)(x + z)$
- $(x + y)' = x' y'$
- $(xy)' = x' + y'$
- $(x')' = x$

DeMorgan's Theorem

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

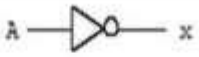
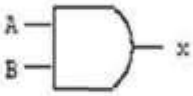
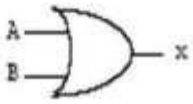
Generalized DeMorgan's Theorem (a) $(a + b + \ldots z)' = a'b' \ldots z'$

(b) $(a.b \ldots z)' = a' + b' + \ldots z$‘

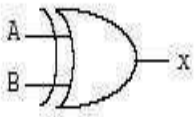
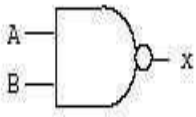
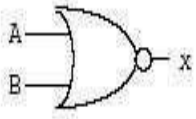
LOGIC GATES

Formal logic: In formal logic, a statement (proposition) is a declarative sentence that is either true(1) or false (0). It is easier to communicate with computers using formal logic.

- Boolean variable: Takes only two values – either true (1) or false (0). They are used as basic units of formal logic.
- Boolean algebra: Deals with binary variables and logic operations operating on those variables.
- Logic diagram: Composed of graphic symbols for logic gates. A simple circuit sketch that represents inputs and outputs of Boolean functions.

Name	Graphic symbol	Algebraic function	Truth table	
Inverter		x = A'	A | x 0 | 1 1 | 0	
AND		x = AB	A B | x 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1	True if both are true.
OR		x = A + B	A B | x 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1	True if either one is true.

- Other common gates include:

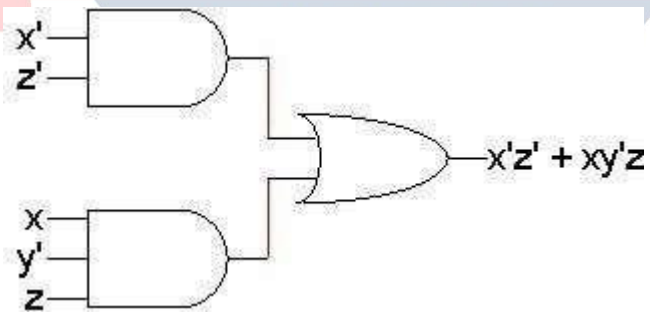
Name	Graphic symbol	Algebraic function	Truth table	
Exclusive-OR (XOR)		x = A ⊕ B = A'B + AB'	A B | x 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0	Parity check: True if only one is true.
NAND		x = (AB)'	A B | x 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0	Inversion of AND.
NOR		x = A + B	A B | x 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0	Inversion of OR.

Minimization of switching functions is to obtain logic circuits with least circuit complexity. This goal is very difficult since how a minimal function relates to the implementation technology is

important. For example, If we are building a logic circuit that uses discrete logic made of small scale Integration ICs(SSIs) like 7400 series, in which basic building block are constructed and are available for use. The goal of minimization would be to reduce the number of ICs and not the logic gates. For example, If we require two 6 and gates and 5 Or gates,we would require 2 AND ICs(each has 4 AND gates) and one OR IC. (4 gates). On the other hand if the same logic could be implemented with only 10 nand gates, we require only 3 ICs. Similarly when we design logic on Programmable device, we may implement the design with certain number of gates and remaining gates may not be used.

Whatever may be the criteria of minimization we would be guided by the following:

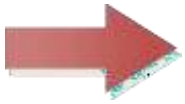
- Boolean algebra helps us simplify expressions and circuits
- Karnaugh Map: A graphical technique for simplifying a Boolean expression into either form:
  - o minimal sum of products (MSP) o
    minimal product of
    sums (MPS)
- Goal of the simplification.
  - o There are a minimal number of product/sum terms oEach term has a minimal number of literals
- Circuit-wise, this leads to a *minimal* two-levelimplementation



- A two-variable function has four possible minterms. We can re-arrange these minterms into a Karnaugh map

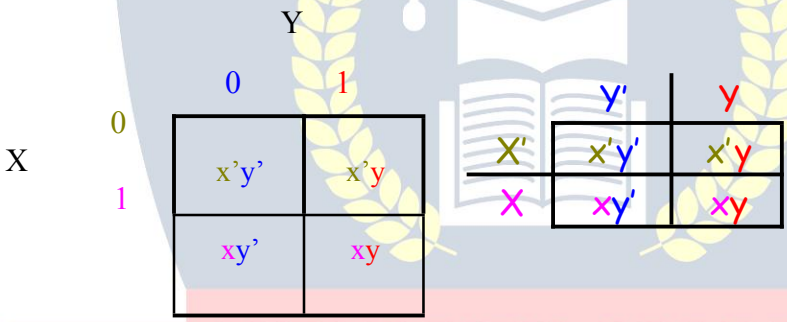
x	Y	Minterm
0	0	x'y'
0	1	x'y
1	0	xy'
1	1	Xy

0 X
1



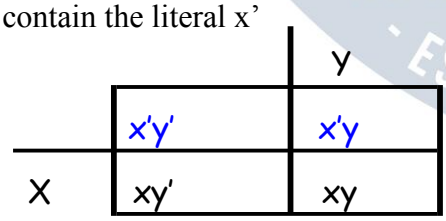
	0	1
0	x'y'	x'y
1	xy'	xy

- Now we can easily see which minterms contain common literals
  - Minterms on the left and right sides contain y' and y respectively
  - Minterms in the top and bottom rows contain x' and x respectively



K- map Simplification

- Imagine a two-variable sum of minterms x'y' + x'y
- Both of these minterms appear in the top row of a Karnaugh map, which means that they both contain the literal x'



- What happens if you simplify this expression using Boolean algebra?
- x'y' + x'y= x'(y' + y) [Distributive]

= x'+1 [y + y' = 1]

= x' [x + 1 = x]

A Three-Variable Karnaugh Map

- For a three-variable expression with inputs x, y, z, the arrangement of minterms is more tricky:

X \ YZ	00	01	11	10
0	x'y'z'	x'y'z	x'yz	x'yz'
1	xy'z'	xy'z	xyz	xyz'

X \ YZ	00	01	11	10
0	m_0	m_1	m_3	m_2
1	m_4	m_5	m_7	m_6

- Another way to label the K-map (use whichever you like):

X \ Y	Z			
	x'y'z'	x'y'z	x'yz	x'yz'
X	xy'z'	xy'z	xyz	xyz'

X \ Y	Z			
	m_0	m_1	m_3	m_2
X	m_4	m_5	m_7	m_6

- With this ordering, any group of 2, 4 or 8 adjacent squares on the map contains common literals that can be factored out

X \ Y	Z			
	x'y'z'	x'y'z	x'yz	x'yz'
X	xy'z'	xy'z	xyz	xyz'

$$
\begin{aligned}
& x'y'z + x'yz \\
= \; & x'z(y' + y) \\
= \; & x'z \bullet 1 \\
= \; & x'z
\end{aligned}
$$

- "Adjacency" includes wrapping around the left and right sides:

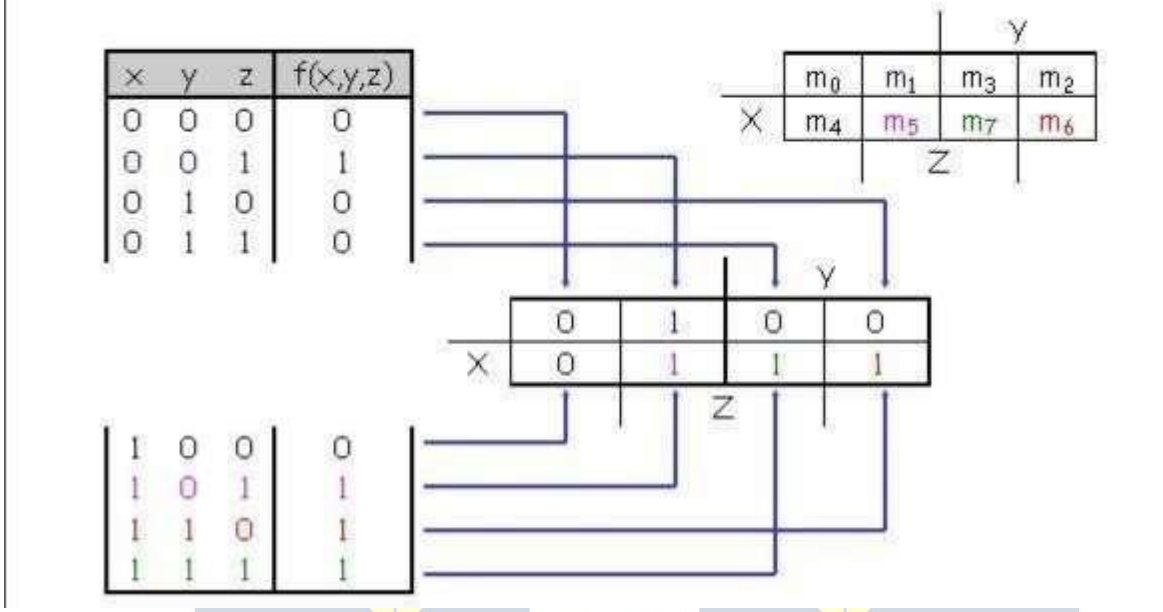
X \ Y	Z			
	x'y'z'	x'y'z	x'yz	x'yz'
X	xy'z'	xy'z	xyz	xyz'

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

- We'll use this property of adjacent squares to do our simplifications.

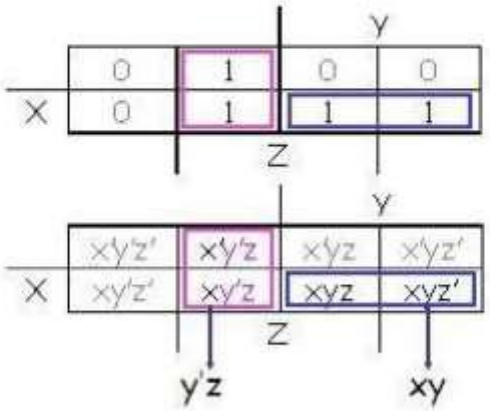
K-maps From Truth Tables

- We can fill in the K-map directly from a truth table
  - The output in row *i* of the table goes into square m_i of the K-map
  - Remember that the rightmost columns of the K-map are "switched"



Reading the MSP from the K-map

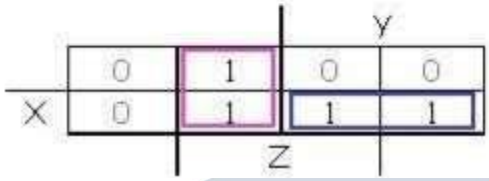
- You can find the minimal SoP expression
  - Each rectangle corresponds to one product term
  - The product is determined by finding the common literals in that rectangle



F(x,y,z)= y'z + xy

Grouping the Minterms Together

- The most difficult step is grouping together all the 1s in the K-map
  - Make rectangles around groups of one, two, four or eight 1s
  - All of the 1s in the map should be included in at least one rectangle
  - Do *not* include any of the 0s
  - Each group corresponds to one product term



K-map Simplification of SoP Expressions

- Let's consider simplifying f(x,y,z) = xy + y'z + xz
- You should convert the expression into a sum of minterms form,
  - The easiest way to do this is to make a truth table for the function, and then read off the minterms
  - You can either write out the literals or use the minterm shorthand
- Here is the truth table and sum of minterms for our example:

x	y	z	f(x,y,z)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$$f(x,y,z) = x'y'z + xy'z + xyz' + xyz$$
$$= m_1 + m_5 + m_6 + m_7$$

Unsimplifying Expressions

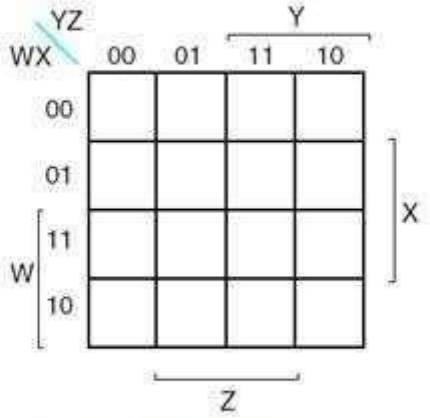
- You can also convert the expression to a sum of minterms with Boolean algebra
  - Apply the distributive law in reverse to add in missing variables.
  - Very few people actually do this, but it's occasionally useful.

$$\begin{aligned} xy + y'z + xz &= (xy \bullet 1) + (y'z \bullet 1) + (xz \bullet 1) \\ &= (xy \bullet (z' + z)) + (y'z \bullet (x' + x)) + (xz \bullet (y' + y)) \\ &= (xyz' + xyz) + (x'y'z + xy'z) + (xy'z + xyz) \\ &= xyz' + xyz + x'y'z + xy'z \\ &= m_1 + m_5 + m_6 + m_7 \end{aligned}$$

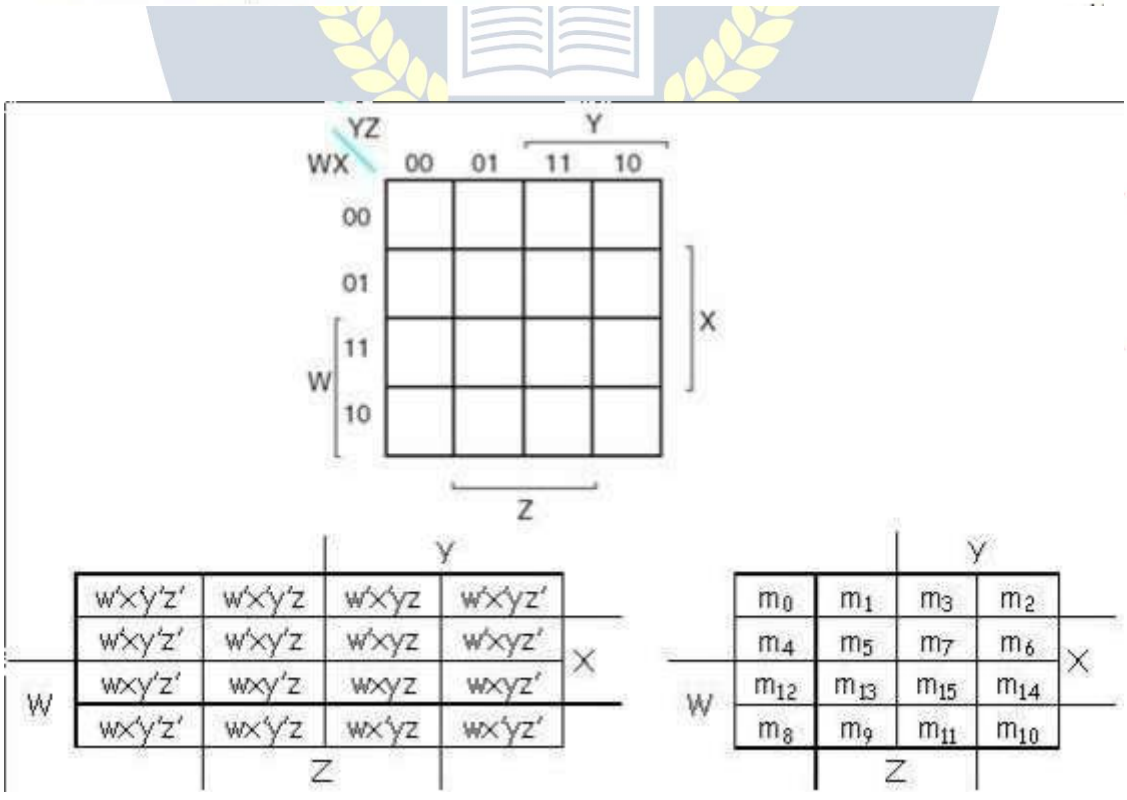
- In both cases, we're actually "unsimplifying" our example expression
  - The resulting expression is larger than the original one!
  - But having all the individual minterms makes it easy to combine them together with the K-map

Four-variable K-maps – f(W,X,Y,Z)

- We can do four-variable expressions too!
  - The minterms in the third and fourth columns, *and* in the third and fourth rows, are switched around.
  - Again, this ensures that adjacent squares have common literals

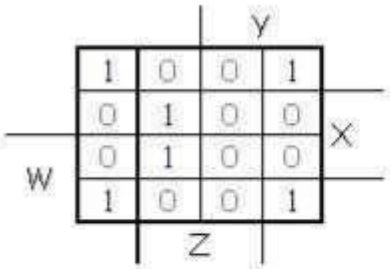


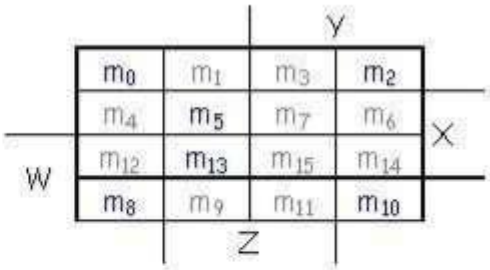
- Grouping minterms is similar to the three-variable case, but:
  - You can have rectangular groups of 1, 2, 4, 8 or 16 minterms
  - You can wrap around *all four* sides



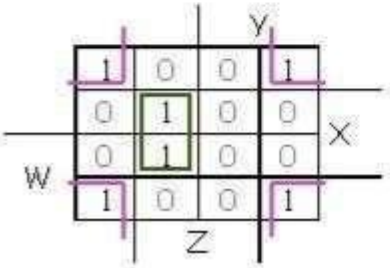
Simplify m0+m2+m5+m8+m10+m13

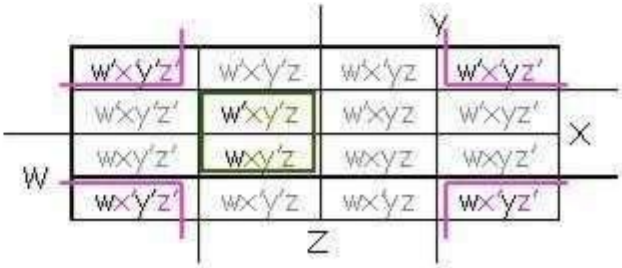
- The expression is already a sum of minterms, so here's the K-map:





- We can make the following groups, resulting in the MSP x'z' + xy'z



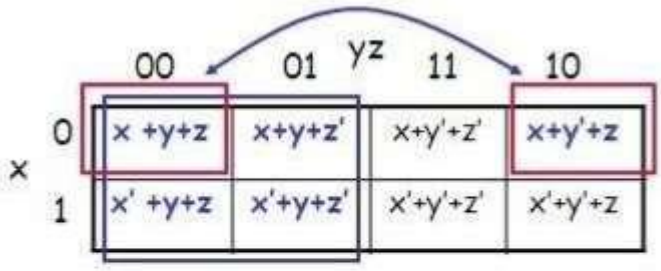


PoS Optimization

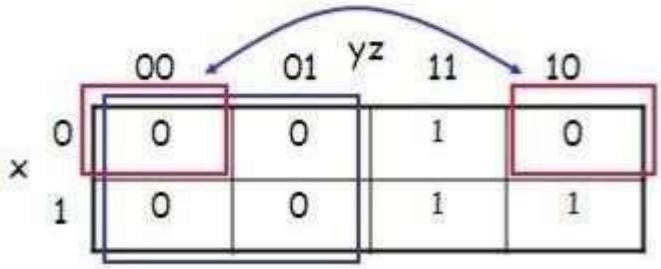
- Maxterms are grouped to find minimal PoS expression

x \ yz	00	01	11	10
0	x+y+z	x+y+z'	x+y'+z'	x+y'+z
1	x'+y+z	x'+y+z'	x'+y'+z'	x'+y'+z

• $F(W,X,Y,Z) = \prod M(0,1,2,4,5)$



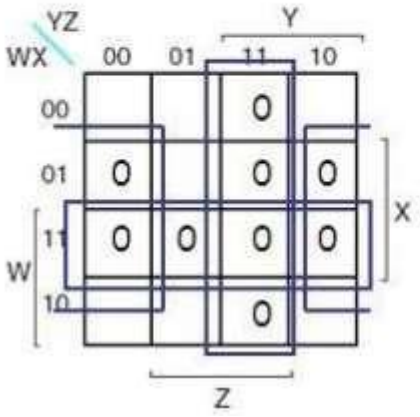
$F(W,X,Y,Z) = Y \cdot (X + Z)$



PoS Optimization from SoP

$F(W,X,Y,Z) = \Sigma m(0,1,2,5,8,9,10)$

$= \prod M(3,4,6,7,11,12,13,14,15)$



$F(W,X,Y,Z) = (W' + X')(Y' + Z')(X' + Z)$

Or,

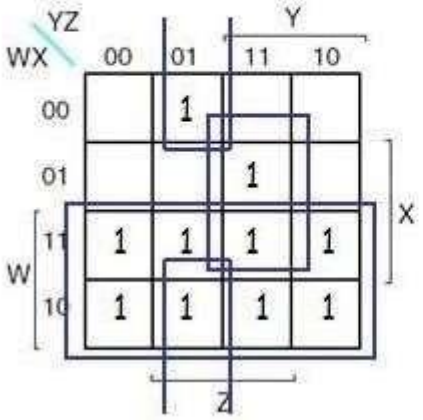
$F(W,X,Y,Z) = X'Y' + X'Z' + W'Y'Z$

Which one is the minimal one?

SoP Optimization from PoS

$F(W,X,Y,Z) = \Pi M(0,2,3,4,5,6)$

$= \Sigma m(1,7,8,9,10,11,12,13,14,15)$



$F(W,X,Y,Z) = W + XYZ + X'Y'Z$

Don't care

- You don't always need all 2^n input combinations in an n-variable function
  - If you can guarantee that certain input combinations never occur
  - If some outputs aren't used in the rest of the circuit
- We mark don't-care outputs in truth tables and K-maps with Xs.

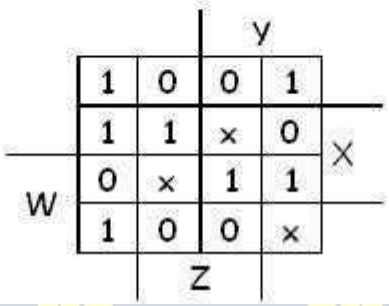
x	y	z	f(x,y,z)
0	0	0	0
0	0	1	1
0	1	0	X
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	X
1	1	1	1

- Within a K-map, each X can be considered as either 0 or 1. You should pick the interpretation that allows for the most simplification.

- Find a MSP for

$$f(w,x,y,z) = \Sigma m(0,2,4,5,8,14,15), d(w,x,y,z) = \Sigma m(7,10,13)$$

This notation means that input combinations wxyz = 0111, 1010 and 1101 (corresponding to minterms m_7, m_{10} and m_{13}) are unused.



K-map Summary

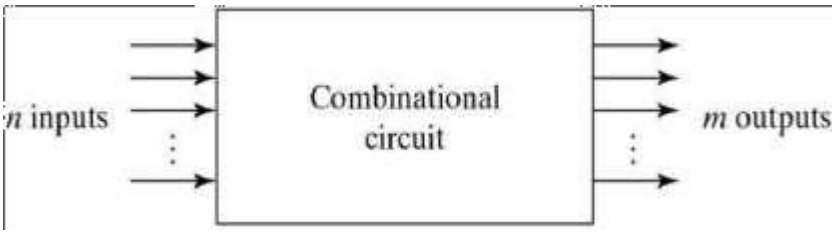
- K-maps are an alternative to algebra for simplifying expressions
  - The result is a MSP/MPS, which leads to a minimal two-level circuit
  - It's easy to handle don't-care conditions
  - K-maps are really only good for manual simplification of small expressions...
  - Things to keep in mind:
  - Remember the correct order of minterms/maxterms on the K-map
  - When grouping, you can wrap around all sides of the K-map, and your groups can overlap
  - Make as few rectangles as possible, but make each of them as large as possible. This leads to fewer, but simpler, product terms
  - There may be more than one valid solution

UNIT-III

DESIGN OF COMBINATIONAL CIRCUITS

Combinational Logic

- Logic circuits for digital systems may be combinational or sequential.
- A combinational circuit consists of input variables, logic gates, and output variables.



For n input variables,there are 2^n possible combinations of binary input variables .For each possible input Combination ,there is one and only one possible output combination.A combinational circuit can be described by m Boolean functions one for each output variables.Usually the input s comes from flip-flops and outputs goto flip-flops.

Design Procedure:

1. The problem is stated
2. The number of available input variables and required output variables is determined.
3. The input and output variables are assigned lettersymbols.
4. The truth table that defines the required relationship between inputs and outputs is derived.
5. The simplified Boolean function for each output is obtained.
6. The logic diagram is drawn.

Adders:

Digital computers perform variety of information processing tasks,the one is arithmetic operations.And the most basic arithmetic operation is the addition of two binary digits.i.e, 4 basic possible operations are:

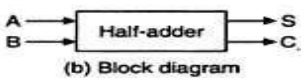
0+0=0,0+1=1,1+0=1,1+1=10

The first three operations produce a sum whose length is one digit, but when augends and addend bits are equal to 1,the binary sum consists of two digits.The higher significant bit of this result is called a carry.A combinational circuit that performs the addition of two bits is called a half- adder. One that performs the addition of 3 bits (two significant bits & previous carry) is called a full adder.& 2 half adder can employ as a full-adder.

The Half Adder: A Half Adder is a combinational circuit with two binary inputs (augends and addend bits and two binary outputs (sum and carry bits.) It adds the two inputs (A and B) and produces the sum (S) and the carry (C) bits. It is an arithmetic operation of addition of two single bit words.

Inputs		Outputs	
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

(a) Truth table



(b) Block diagram

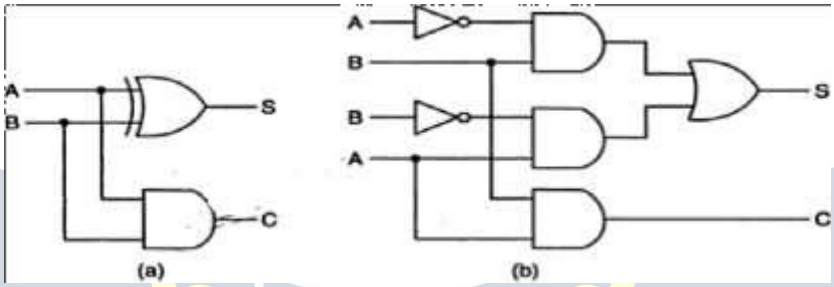
The Sum(S) bit and the carry (C) bit, according to the rules of binary addition, the sum (S) is the X-OR of A and B (It represents the LSB of the sum). Therefore,

$$S=A'B+AB'$$

The carry (C) is the AND of A and B (it is 0 unless both the inputs are 1).Therefore,

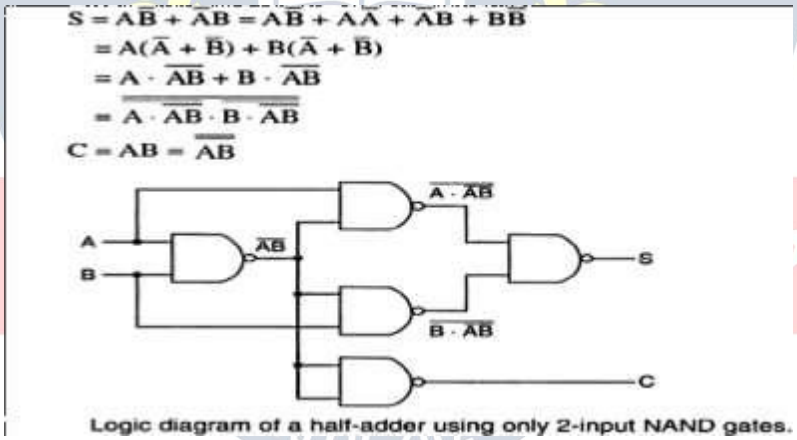
$$C=AB$$

A half-adder can be realized by using one X-OR gate and one AND gate a



Logic diagrams of half-adder

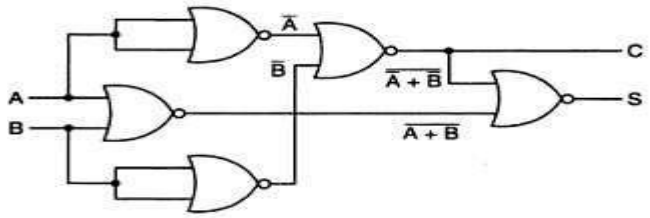
NAND LOGIC:



Logic diagram of a half-adder using only 2-input NAND gates.

NOR Logic:

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



Logic diagram of a half-adder using only 2-input NOR gates.

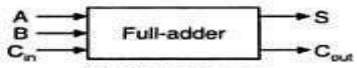
The Full Adder:

A Full-adder is a combinational circuit that adds two bits and a carry and outputs a sum bit and a carry bit. To add two binary numbers, each having two or more bits, the LSBs can be added by using a half-adder. The carry resulted from the addition of the LSBs is carried over to the next significant column and added to the two bits in that column. So, in the second and higher columns, the two data bits of that column and the carry bit generated from the addition in the previous column need to be added.

The full-adder adds the bits A and B and the carry from the previous column called the carry-in Cin and outputs the sum bit S and the carry bit called the carry-out Cout . The variable S gives the value of the least significant bit of the sum. The variable Cout gives the output carry.The eight rows under the input variables designate all possible combinations of 1s and 0s that these variables may have. The 1s and 0s for the output variables are determined from the arithmetic sum of the input bits. When all the bits are 0s , the output is 0. The S output is equal to 1 when only 1 input is equal to 1 or when all the inputs are equal to 1. The Cout has a carry of 1 if two or three inputs are equal to 1.

Inputs			Sum	Carry
A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(a) Truth table



(b) Block diagram

Full-adder.

From the truth table, a circuit that will produce the correct sum and carry bits in response to every possible combination of A,B and Cin is described by

$$S = ABCin + AB\overline{Cin} + \overline{A}BCin + ABCin$$

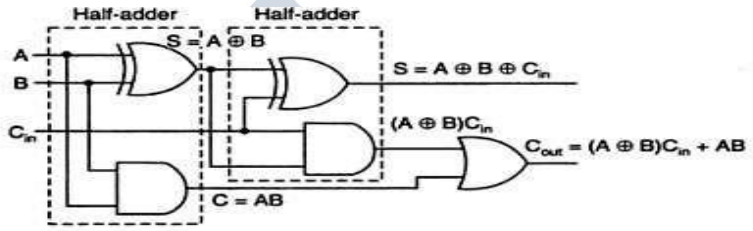
$$Cout = \underline{AB}Cin + AB\underline{Cin} + ABCin + \underline{A}BCin$$

and

$$S = A \oplus B \oplus Cin$$

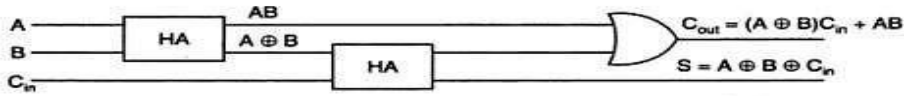
$$Cout = ACin + BCin + AB$$

The sum term of the full-adder is the X-OR of A,B, and Cin, i.e, the sum bit the modulo sum of the data bits in that column and the carry from the previous column. The logic diagram of the full-adder using two X-OR gates and two AND gates (i.e, Two half adders) and one OR gate is



Logic diagram of a full-adder using two half-adders.

The block diagram of a full-adder using two half-adders is



Block diagram of a full-adder using two half-adders.

Even though a full-adder can be constructed using two half-adders, the disadvantage is that the bits must propagate through several gates in accession, which makes the total propagation delay greater than that of the full-adder circuit using AOI logic.

The Full-adder neither can also be realized using universal logic, i.e., either only NAND gates or only NOR gates as

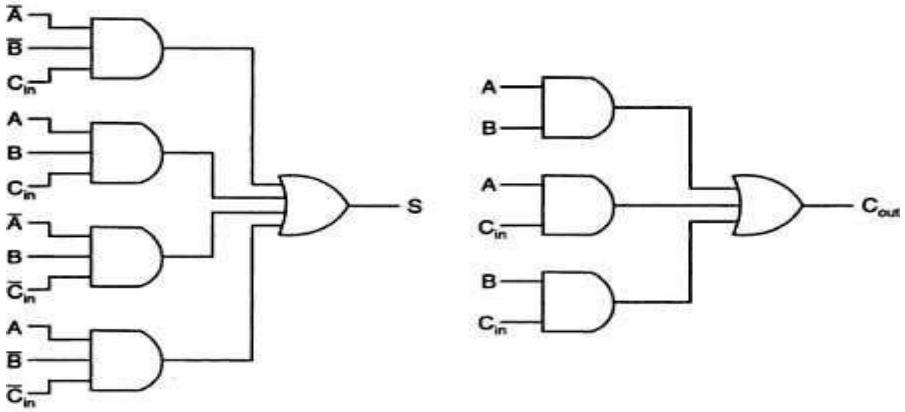
$$A \oplus B = \overline{A \cdot \overline{AB}} \cdot \overline{B \cdot \overline{AB}}$$

Then

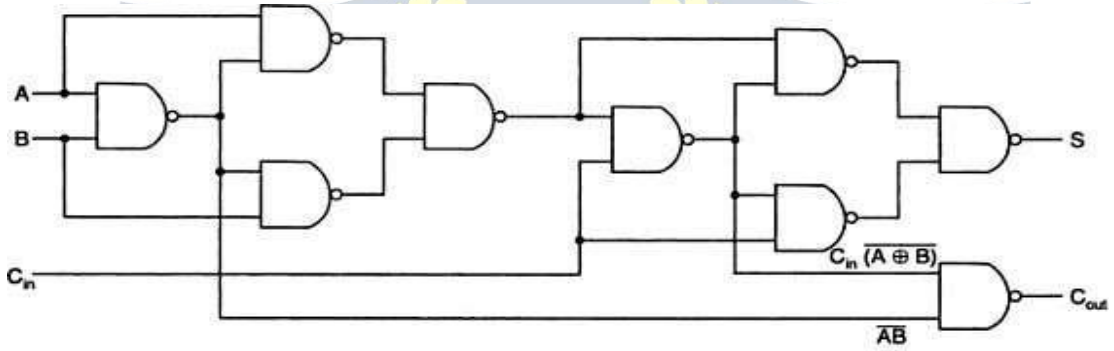
$$S = A \oplus B \oplus C_{in} = \overline{\overline{(A \oplus B) \cdot \overline{(A \oplus B)C_{in}}} \cdot \overline{C_{in} \cdot \overline{(A \oplus B)C_{in}}}}$$

NAND Logic:

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



Sum and carry bits of a full-adder using AOI logic.



Logic diagram of a full-adder using only 2-input NAND gates.

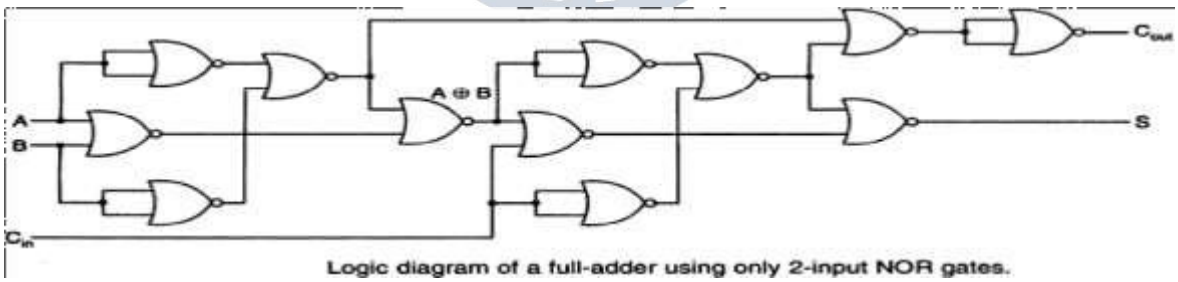
NOR Logic:

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

Then

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

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



Logic diagram of a full-adder using only 2-input NOR gates.

Subtractors:

The subtraction of two binary numbers may be accomplished by taking the complement of the subtrahend and adding it to the minuend. By this, the subtraction operation becomes an addition operation and instead of having a separate circuit for subtraction, the adder itself can be used to

perform subtraction. This results in reduction of hardware. In subtraction, each subtrahend bit of the number is subtracted from its corresponding significant minuend bit to form a difference bit. If the minuend bit is smaller than the subtrahend bit, a 1 is borrowed from the next significant position., that has been borrowed must be conveyed to the next higher pair of bits by means of a signal coming out (output) of a given stage and going into (input) the next higher stage.

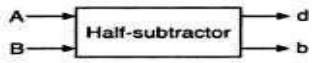
The Half-Subtractor:

A Half-subtractor is a combinational circuit that subtracts one bit from the other and produces the difference. It also has an output to specify if a 1 has been borrowed. . It is used to subtract the LSB of the subtrahend from the LSB of the minuend when one binary number is subtracted from the other.

A Half-subtractor is a combinational circuit with two inputs A and B and two outputs d and b. d indicates the difference and b is the output signal generated that informs the next stage that a 1 has been borrowed. When a bit B is subtracted from another bit A, a difference bit (d) and a borrow bit (b) result according to the rules given as

Inputs		Outputs	
A	B	d	b
0	0	0	0
1	0	1	0
1	1	0	0
0	1	1	1

(a) Truth table



(b) Block diagram

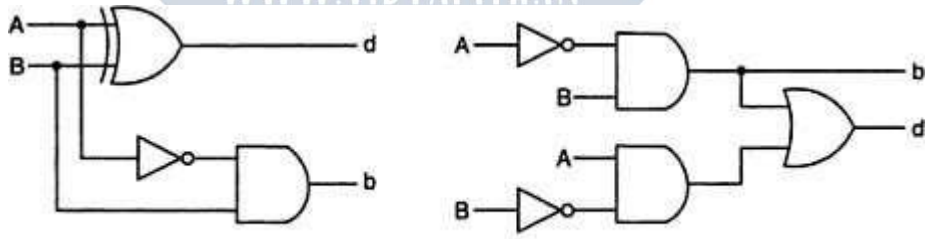
Half-subtractor.

The output borrow b is a 0 as long as A≥B. It is a 1 for A=0 and B=1. The d output is the result of the arithmetic operation 2b+A-B.

A circuit that produces the correct difference and borrow bits in response to every possible combination of the two 1-bit numbers is , therefore ,

$$d = A\beta + \quad A \oplus B \text{ and } b = \quad B$$

That is, the difference bit is obtained by X-OR ing the two inputs, and the borrow bit is obtained by ANDing the complement of the minuend with the subtrahend.Note that logic for this exactly the same as the logic for output S in the half-adder.



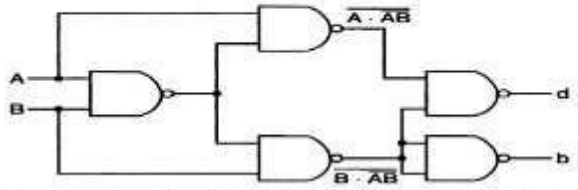
Logic diagrams of a half-subtractor.

A half-substractor can also be realized using universal logic either using only NAND gates or using NOR gates as:

NAND Logic:

$$d = A \oplus B = \overline{\overline{A \cdot \overline{AB}} \cdot \overline{B \cdot \overline{AB}}}$$

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



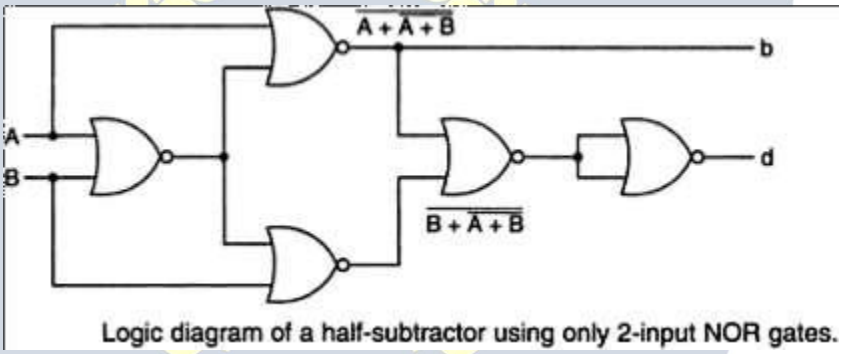
Logic diagram of a half-subtractor using only 2-input NAND gates.

NOR Logic:

$$d = A \oplus B = A\bar{B} + \bar{A}B = A\bar{B} + B\bar{B} + \bar{A}B + A\bar{A}$$

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

$$d = \bar{A}B = \bar{A}(A + B) = \overline{\bar{A}(A + B)} = \overline{A + (\overline{A + B})}$$



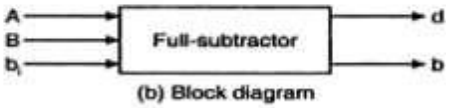
Logic diagram of a half-subtractor using only 2-input NOR gates.

The Full-Subtractor:

The half-subtractor can be only for LSB subtraction. IF there is a borrow during the subtraction of the LSBs, it affects the subtraction in the next higher column; the subtrahend bit is subtracted from the minuend bit, considering the borrow from that column used for the subtraction in the preceding column. Such a subtraction is performed by a full-subtractor. It subtracts one bit (B) from another bit (A) , when already there is a borrow bi from this column for the subtraction in the preceding column, and outputs the difference bit (d) and the borrow bit(b) required from the next d and b. The two outputs present the difference and output borrow. The 1s and 0s for the output variables are determined from the subtraction of A-B-bi.

Inputs			Difference	Borrow
A	B	b_i	d	b
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

(a) Truth table



(b) Block diagram

Full-subtractor.

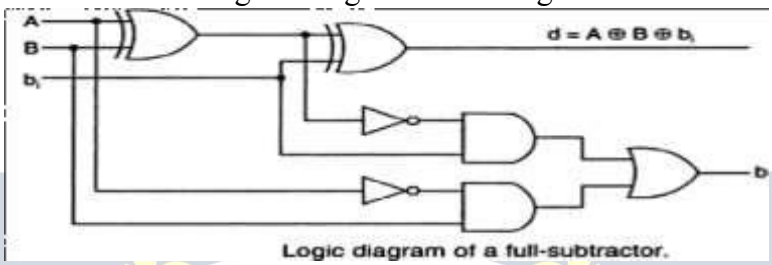
From the truth table, a circuit that will produce the correct difference and borrow bits in response to every possiblecombinations of A,B and bi is

$$d = \bar{A}\bar{B}b_i + \bar{A}B\bar{b}_i + A\bar{B}\bar{b}_i + ABb_i$$
$$= b_i(AB + \bar{A}\bar{B}) + \bar{b}_i(A\bar{B} + \bar{A}B)$$
$$= b_i(\overline{A \oplus B}) + \bar{b}_i(A \oplus B) = A \oplus B \oplus b_i$$

and

$$b = \bar{A}\bar{B}b_i + \bar{A}B\bar{b}_i + \bar{A}Bb_i + ABb_i = \bar{A}B(b_i + \bar{b}_i) + (AB + \bar{A}\bar{B})b_i$$
$$= \bar{A}B + (\overline{A \oplus B})b_i$$

A full-subtractor can be realized using X-OR gates and AOI gates as



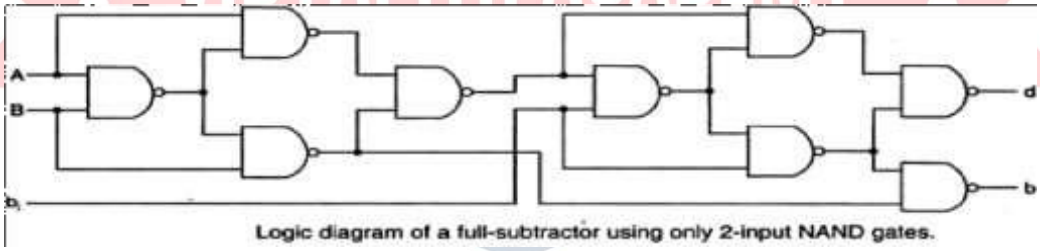
Logic diagram of a full-subtractor.

The full subtractor can also be realized using universal logic either using only NAND gates or using NOR gates as:

NAND Logic:

$$d = A \oplus B \oplus b_i = \overline{\overline{(A \oplus B) \oplus b_i}} = \overline{(A \oplus B)\overline{(A \oplus B)b_i} \cdot b_i\overline{(A \oplus B)b_i}}$$

$$b = \bar{A}B + b_i(\overline{A \oplus B}) = \overline{\bar{A}B + b_i(\overline{A \oplus B})}$$
$$= \overline{\bar{A}B \cdot b_i(\overline{A \oplus B})} = \overline{B(\bar{A} + \bar{B}) \cdot b_i(\bar{b}_i + (\overline{A \oplus B}))}$$
$$= \overline{B \cdot \overline{AB} \cdot b_i[b_i \cdot (\overline{A \oplus B})]}$$



Logic diagram of a full-subtractor using only 2-input NAND gates.

NOR Logic:

$$d = A \oplus B \oplus b_i = \overline{\overline{(A \oplus B) \oplus b_i}}$$
$$= \overline{\overline{(A \oplus B)b_i} + \overline{(A \oplus B)\bar{b}_i}}$$
$$= \overline{[(A \oplus B) + \overline{(A \oplus B)\bar{b}_i}][b_i + \overline{(A \oplus B)\bar{b}_i}]}$$

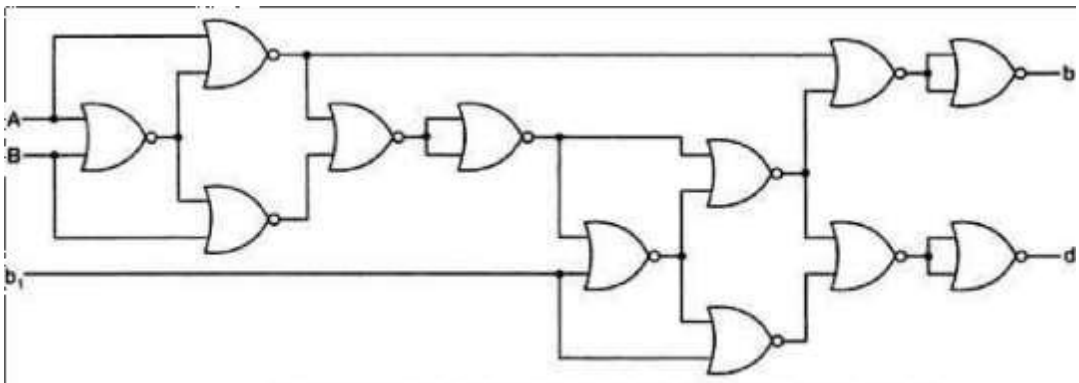
$$= (A \oplus B) + \overline{(A \oplus B) + b_i} + b_i + \overline{(A \oplus B) + b_i}$$

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

$$b = \bar{A}B + b_i(\overline{A \oplus B})$$

$$= \bar{A}(A + B) + (\overline{A \oplus B})[(A \oplus B) + b_i]$$

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

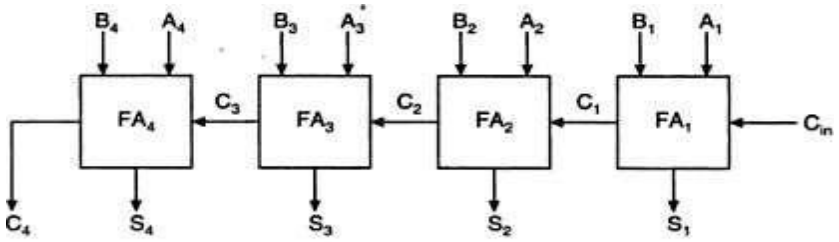


Logic diagram of a full subtractor using only 2-input NOR gates.

Binary Parallel Adder:

A binary parallel adder is a digital circuit that adds two binary numbers in parallel form and produces the arithmetic sum of those numbers in parallel form. It consists of full adders connected in a chain , with the output carry from each full-adder connected to the input carry of the next full-adder in the chain.

The interconnection of four full-adder (FA) circuits to provide a 4-bit parallel adder. The augends bits of A and addend bits of B are designated by subscript numbers from right to left, with subscript 1 denoting the lower –order bit. The carries are connected in a chain through the fulladders. The input carry to the adder is Cin and the output carry is C4. The S output generates the required sum bits. When the 4-bit full-adder circuit is enclosed within an IC package, it has four terminals for the augends bits, four terminals for the addend bits, four terminals for the sum bits, and two terminals for the input and output carries. AN n-bit parallel adder requires n-full adders. It can be constructed from 4-bit, 2-bit and 1-bit full adder ICs by cascading several packages. The output carry from one package must be connected to the input carry of the one with the next higher –order bits. The 4-bit full adder is a typical example of an MSI function.



Logic diagram of a 4-bit binary parallel adder.

Ripple carry adder:

In the parallel adder, the carry –out of each stage is connected to the carry-in of the

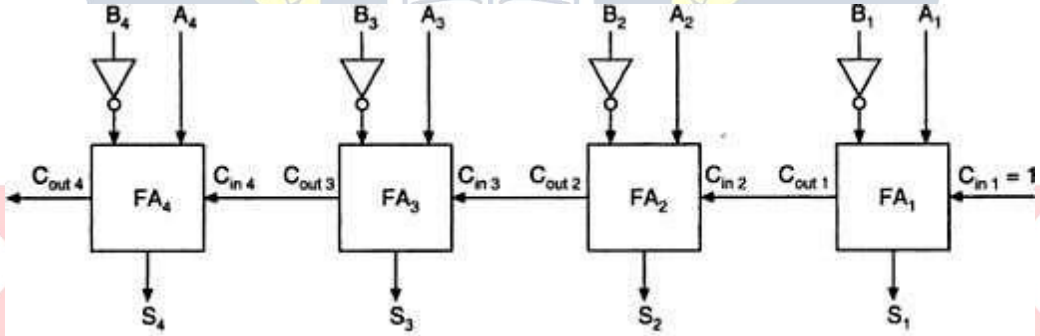
next stage. The sum and carry-out bits of any stage cannot be produced, until sometime after the carry-in of that stage occurs. This is due to the propagation delays in the logic circuitry,
which lead to a time delay in the addition process. The carry propagation delay for each full- adder is the time between the application of the carry-in and the occurrence of the carry-out.

The 4-bit parallel adder, the sum (S1) and carry-out (C1) bits given by FA1 are not valid, until after the propagation delay of FA1. Similarly, the sum S2 and carry-out (C2) bits given by FA2 are not valid until after the cumulative propagation delay of two full adders (FA1 and FA2) , and so on. At each stage ,the sum bit is not valid until after the carry bits in all the preceding stages are valid. Carry bits must propagate or ripple through all stages before the most significant sum bit is valid. Thus, the total sum (the parallel output) is not valid until after the cumulative delay of all the adders.

The parallel adder in which the carry-out of each full-adder is the carry-in to the next most significant adder is called a ripple carry adder.. The greater the number of bits that a ripple carry adder must add, the greater the time required for it to perform a valid addition. If two numbers are added such that no carries occur between stages, then the add time is simply the propagation time through a single full-adder.

4- Bit Parallel Subtractor:

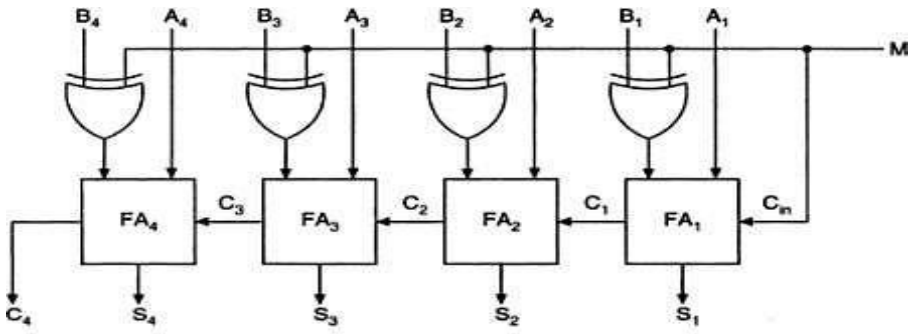
The subtraction of binary numbers can be carried out most conveniently by means of complements , the subtraction A-B can be done by taking the 2‘s complement of B and adding it to A . The 2‘s complement can be obtained by taking the 1‘s complement and adding 1 to the least significant pair of bits. The 1‘s complement can be implemented with inverters as



Logic diagram of a 4-bit parallel subtractor.

Binary-Adder Subtractor:

A 4-bit adder-subtractor, the addition and subtraction operations are combined into one circuit with one common binary adder. This is done by including an X-OR gate with each fulladder. The mode input M controls the operation. When M=0, the circuit is an adder, and when M=1, the circuit becomes a subtractor. Each X-OR gate receives input M and one of the inputs of B. When M=0, $B \oplus 0 = B$ .The full-adder receives the value of B , the input carry is 0 and the circuit performs A+B. when $B \oplus 1 = B'$ and C1=1. The B inputs are complemented and a 1 is through the input carry. The circuit performs the operation A plus the 2‘s complement of B.



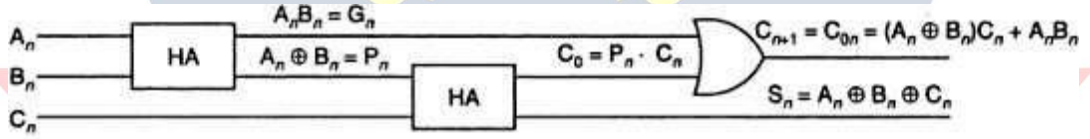
Logic diagram of a 4-bit binary adder-subtractor.

The Look-Ahead –Carry Adder:

In parallel-adder,the speed with which an addition can be performed is governed by the time required for the carries to propagate or ripple through all of the stages of the adder. The look-ahead carry adder speeds up the process by eliminating this ripple carry delay. It examines all the input bits simultaneously and also generates the carry-in bits for all the stages simultaneously.

The method of speeding up the addition process is based on the two additional functions of the full-adder, called the carry generate and carry propagate functions.

Consider one full adder stage; say the nth stage of a parallel adder as shown in fig. we know that is made by two half adders and that the half adder contains an X-OR gate to produce the sum and an AND gate to produce the carry. If both the bits An and Bn are 1s, a carry has to be generated in this stage regardless of whether the input carry Cin is a 0 or a 1. This is called generated carry, expressed as Gn= An.Bn which has to appear at the output through the OR gate as shown in fig.



A full adder (*n*th stage of a parallel adder).

Thereis another possibility of producing a carry out. X-OR gate inside the half-adder at the input produces an intermediary sum bit- call it Pn –which is expressed as $P_n = A_n \oplus B_n$. Next Pn and Cn are added using the X-OR gate inside the second half adder to produce the final sum bit and and output carry$C_0 = P_n.C_n = (A_n \oplus B_n)C_n$ which becomes carry for the (n+1) th stage.

$S_n = P_n \oplus C_n$ where $P_n = A_n \oplus B_n$

Consider the case of both Pn and Cn being 1. The input carry Cn has to be propagated to the output only if Pn is 1. If Pn is 0, even if Cn is 1, the and gate in the second half-adder will inhibit Cn . the carry out of the nth stage is 1 when either Gn=1 or Pn.Cn =1 or both Gn and Pn.Cn are equal to 1.

For the final sum and carry outputs of the nth stage, we get the following Boolean expressions.

$$S_n = P_n \oplus C_n \text{ where } P_n = A_n \oplus B_n$$
$$C_{on} = C_{n+1} = G_n + P_n C_n \text{ where } G_n = A_n \cdot B_n$$

Observe the recursive nature of the expression for the output carry at the nth stage which becomes the input carry for the (n+1)st stage .it is possible to express the output carry of a higher significant stage is the carry-out of the previous stage.

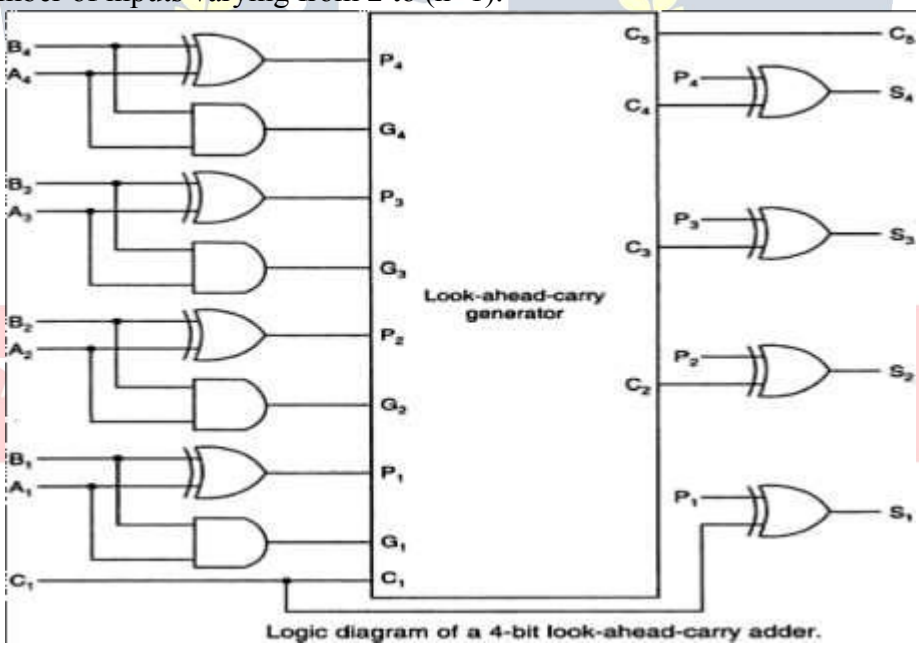
Based on these , the expression for the carry-outs of various full adders are as follows,

$$C_1 = G_0 + P_0 \cdot C_0$$
$$C_2 = G_1 + P_1 \cdot C_1 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0$$
$$C_3 = G_2 + P_2 \cdot C_2 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0$$
$$C_4 = G_3 + P_3 \cdot C_3 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_0$$

The general expression for n stages designated as 0 through $(n-1)$ would be

$$C_n = G_{n-1} + P_{n-1} \cdot C_{n-1} = G_{n-1} + P_{n-1} \cdot G_{n-2} + P_{n-1} \cdot P_{n-2} \cdot G_{n-3} + \ldots + P_{n-1} \cdot \ldots P_0 \cdot C_0$$

Observe that the final output carry is expressed as a function of the input variables in SOP form. Which is two level AND-OR or equivalent NAND-NAND form. Observe that the full look-ahead scheme requires the use of OR gate with (n+1) inputs and AND gates with number of inputs varying from 2 to (n+1).



Logic diagram of a 4-bit look-ahead-carry adder.

2's complement Addition and Subtraction using Parallel Adders:

Most modern computers use the 2's complement system to represent negative numbers and to perform subtraction operations of signed numbers can be performed using only the addition operation ,if we use the 2's complement form to represent negative numbers.

The circuit shown can perform both addition and subtraction in the 2's complement. This adder/subtractor circuit is controlled by the control signal ADD/SUB'. When the ADD/SUB' level is HIGH, the circuit performs the addition of the numbers stored in registers A and B. When the

ADD/Sub‘ level is LOW, the circuit subtract the number in register B from the number in register A. The operation is:

When ADD/SUB‘ is a 1:

1. AND gates 1,3,5 and 7 are enabled , allowing B0,B1,B2and B3 to pass to the OR gates 9,10,11,12 . AND gates 2,4,6 and 8 are disabled , blocking B0‘,B1‘,B2‘, and B3‘ from reaching the OR gates 9,10,11 and 12.

2. The two levels B0 to B3 pass through the OR gates to the 4-bit parallel adder, to be added to the bits A0 to A3. The sum appears at the output S0 to S3

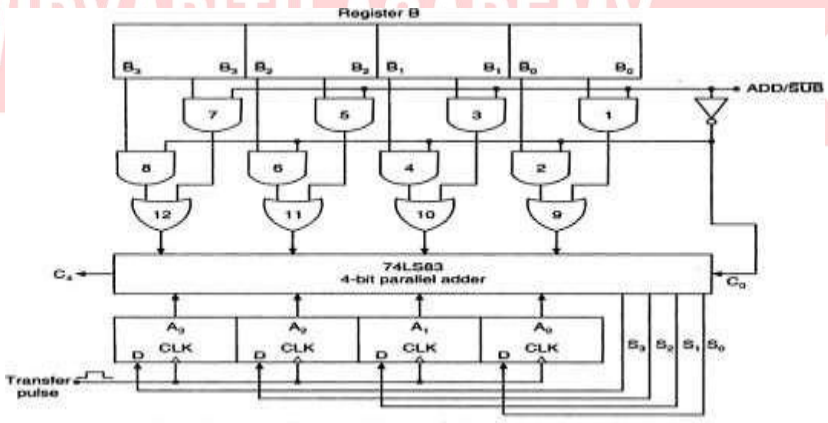
3. Add/SUB‘ =1 causes no carry into the adder.

When ADD/SUB‘ is a 0:

1. AND gates 1,3,5 and 7 are disabled , allowing B0,B1,B2and B3 from reaching the OR gates 9,10,11,12 . AND gates 2,4,6 and 8 are enabled , blocking B0‘,B1‘,B2‘, and B3‘ from reaching the OR gates.
2. The two levels B0‘ to B3‘ pass through the OR gates to the 4-bit parallel adder, to be added to the bits A0 to A3.The C0 is now 1.thus the number in register B is converted to its 2‘s complement form.

3. The difference appears at the output S0 to S3.

Adders/Subtractors used for adding and subtracting signed binary numbers. In computers , the output is transferred into the register A (accumulator) so that the result of the addition or subtraction always end up stored in the register A This is accomplished by applying a transfer pulse to the CLK inputs of register A.



Logic diagram of a parallel adder/subtractor using 2's complement system.

Serial Adder:

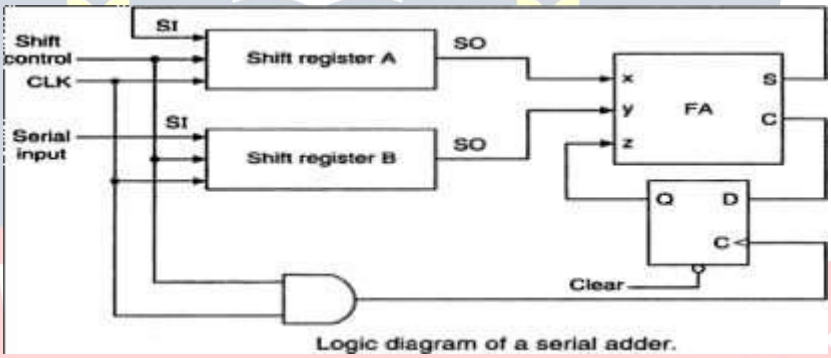
A serial adder is used to add binary numbers in serial form. The two binary numbers to be added serially are stored in two shift registers A and B. Bits are added one pair at a time through a single full adder (FA) circuit as shown. The carry out of the full-adder is transferred to a D flip-flop. The output of this flip-flop is then used as the carry input for the next pair of significant bits. The sum bit from the S output of the full-adder could be transferred to a third shift register. By

shifting the sum into A while the bits of A are shifted out, it is possible to use one register for storing both augend and the sum bits. The serial input register B can be used to transfer a new binary number while the addend bits are shifted out during the addition.

The operation of the serial adder is:

Initially register A holds the augend, register B holds the addend and the carry flip-flop is cleared to 0. The outputs (SO) of A and B provide a pair of significant bits for the full-adder at x and y. The shift control enables both registers and carry flip-flop , so, at the clock pulse both registers are shifted once to the right, the sum bit from S enters the left most flip-flop of A , and the output carry is transferred into flip-flop Q . The shift control enables the registers for a number of clock pulses equal to the number of bits of the registers. For each succeeding clock pulse a new sum bit is transferred to A, a new carry is transferred to Q, and both registers are shifted once to the right. This process continues until the shift control is disabled. Thus the addition is accomplished by passing each pair of bits together with the previous carry through a single full adder circuit and transferring the sum, one bit at a time, into register A.

Initially, register A and the carry flip-flop are cleared to 0 and then the first number is added from B. While B is shifted through the full adder, a second number is transferred to it through its serial input. The second number is then added to the content of register A while a third number is transferred serially into register B. This can be repeated to form the addition of two, three, or more numbers and accumulate their sum in register A.



Logic diagram of a serial adder.

Difference between Serial and Parallel Adders:

The parallel adder registers with parallel load, whereas the serial adder uses shift registers. The number of full adder circuits in the parallel adder is equal to the number of bits in the binary numbers, whereas the serial adder requires only one full adder circuit and a carry flip- flop. Excluding the registers, the parallel adder is a combinational circuit, whereas the serial adder is a sequential circuit. The sequential circuit in the serial adder consists of a full-adder and a flip-flop that stores the output carry.

BCD Adder:

The BCD addition process:

1. Add the 4-bit BCD code groups for each decimal digit position using ordinary binary addition.

2. For those positions where the sum is 9 or less, the sum is in proper BCD form and no correction is needed.

3. When the sum of two digits is greater than 9, a correction of 0110 should be added to that sum, to produce the proper BCD result. This will produce a carry to be added to the next decimalposition.

A BCD adder circuit must be able to operate in accordance with the above steps. In other words, the circuit must be able to do the following:

1. Add two 4-bit BCD code groups, using straight binaryaddition.
2. Determine, if the sum of this addition is greater than 1101 (decimal 9); if it is , add 0110 (decimal 6) to this sum and generate a carry to the next decimalposition.

The first requirement is easily met by using a 4- bit binary parallel adder such as the 74LS83 IC .For example , if the two BCD code groups A3A2A1A0and B3B2B1B0 are applied to a 4-bit parallel adder, the adder will output S4S3S2S1S0 , where S4 is actually C4 , the carry –out of the MSB bits.

The sum outputs S4S3S2S1S0 can range anywhere from 00000 to 100109when both the BCD code groups are 1001=9). The circuitry for a BCD adder must include the logic needed to detect whenever the sum is greater than 01001, so that the correction can be added in. Those cases , where the sum is greater than 1001 are listed as:

S_4	S_3	S_2	S_1	S_0	Decimal number
0	1	0	1	0	10
0	1	0	1	1	11
0	1	1	0	0	12
0	1	1	0	1	13
0	1	1	1	0	14
0	1	1	1	1	15
1	0	0	0	0	16
1	0	0	0	1	17
1	0	0	1	0	18

Let us define a logic output X that will go HIGH only when the sum is greater than 01001 (i.e, for the cases in table). If examine these cases ,see that X will be HIGH for either of the following conditions:

1. Whenever S4 =1(sum greater than 15)

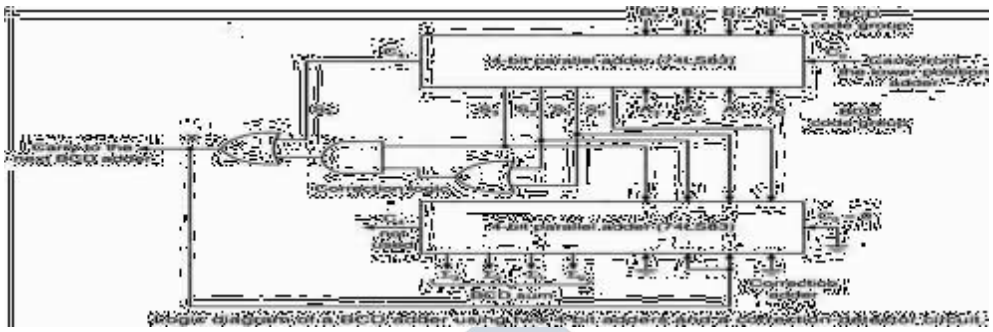
2. Whenever S3 =1 and either S2 or S1 or both are 1 (sum 10 to

15) This condition can be expressed as

X=S4+S3(S2+S1)

Whenever X=1, it is necessary to add the correction factor 0110 to the sum bits, and to generate a carry. The circuit consists of three basic parts. The two BCD code groups A3A2A1A0 and B3B2B1B0 are added together in the upper 4-bit adder, to produce the sum S4S3S2S1S0. The logic gates shown implement the expression for X. The lower 4-bit adder will add the correction 0110 to the sum bits, only when X=1, producing the final BCD sum output represented by ∑3∑2∑1∑0. The X is also the carry-out that is produced when the sum is greater than 01001. When X=0, there is no carry and no addition of 0110. In such cases, ∑3∑2∑1∑0= S3S2S1S0.

Two or more BCD adders can be connected in cascade when two or more digit decimal numbers are to be added. The carry-out of the first BCD adder is connected as the carry-in of the second BCD adder, the carry-out of the second BCD adder is connected as the carry-in of the third BCD adder and so on.



EXCESS-3 (XS-3) ADDER:

To perform Excess-3 additions,

1. Add two xs-3 code groups
2. If carry=1, add 0011(3) to the sum of those two codegroups
   If carry =0, subtract 0011(3) i.e., add 1101 (13 in decimal) to the sum of those two code groups. Ex: Add 9 and 5

```
          1100  9 in Xs-3
         +1000 ____        5 in xs-3
         _ _ __
      1  0100              there is a carry
+0011    0011               add 3 to each group
------   ----------
0100     0111              14 in xs-3
  (1)      (4)
```

EX:

```
(b)   0 1 1 1   4 in XS-3
    + 0 1 1 0   3 in XS-3
      1 1 0 1   no carry
    + 1 1 0 1   Subtract 3 (i.e. add 13)
Ignore carry 1 1 0 1 0   7 in XS-3
                 (7)
```

Implementation of xs-3 adder using 4-bit binary adders is shown. The augend (A3 A2A1A0) and addend (B3B2B1B0) in xs-3 are added using the 4-bit parallel adder. If the carry is a 1, then 0011(3) is added to the sum bits S3S2S1S0 of the upper adder in the lower 4-bit parallel adder. If the carry is a 0, then 1101(3) is added to the sum bits (This is equivalent to subtracting 0011(3) from the sum bits. The correct sum in xs-3 is obtained

Excess-3 (XS-3) Subtractor:

To perform Excess-3 subtraction,

1. Complement the subtrahend
2. Add the complemented subtrahend to theminuend.
3. If carry =1, result is positive. Add 3 and end around carry to the result . If carry=0, the result is negative. Subtract 3, i.e, and take the 1‘s complement of the result.

Ex: Perform 9-4

```
  1100    9 in xs-3
 +1000    Complement of 4 n Xs-3
 ------
(1) 0100  There is a carry
 +0011    Add 0011(3)
--------
  0111
     1    End around carry
--------
  1000    5 in xs-3
```

The minuend and the 1‘s complement of the subtrahend in xs-3 are added in the upper 4-bit parallel adder. If the carry-out from the upper adder is a 0, then 1101 is added to the sum bits of the upper adder in the lower adder and the sum bits of the lower adder are complemented to get the result. If the carry-out from the upper adder is a 1, then 3=0011 is added to the sum bits of the lower adder and the sum bits of the lower adder give the result.

Binary Multipliers:

In binary multiplication by the paper and pencil method, is modified somewhat in digital machines because a binary adder can add only two binary numbers at a time.

In a binary multiplier, instead of adding all the partial products at the end, they are added two at a time and their sum accumulated in a register (the accumulator register). In addition, when the multiplier bit is a 0,0s are not written down and added because it does not affect the final result. Instead, the multiplicand is shifted left by one bit.

The multiplication of 1110 by 1001 using this process is

```
Multiplicand 1110
Multiplier   1001
                  1110 The LSB of the multiplier is a 1; write down the multiplicand; shift the
                       multiplicand one position to the left (1 1 1 0 0 )
                  1110 The second multiplier bit is a 0; write down the previous result 1110;
                       shift the multiplicand to the left again (1 1 1 0 0 0)
                +1110000   The fourth multiplier bit is a 1 write down the new
                       multiplicand add it to the first partial product to obtain the final
                       product.
             1111110
```

This multiplication process can be performed by the serial multiplier circuit , which multiplies two 4-bit numbers to produce an 8-bit product. The circuit consists of following elements

X register: A 4-bit shift register that stores the multiplier --- it will shift right on the falling edge of the clock. Note that 0s are shifted in from the left.

B register: An 8-bit register that stores the multiplicand; it will shift left on the falling edge of the clock. Note that 0s are shifted in from the right.

A register: An 8-bit register, i.e, the accumulator that accumulates the partial products.

Adder: An 8-bit parallel adder that produces the sum of A and B registers. The adder outputs S7 through S0 are connected to the D inputs of the accumulator so that the sum can be transferred to the accumulator only when a clock pulse gets through the AND gate.
The circuit operation can be described by going through each step in the multiplication of 1110 by 1001. The complete process requires 4 clock cycles.

1. **Before the first clock pulse***:* Prior to the occurrence of the first clock pulse, the register A is loaded with 00000000, the register B with the multiplicand 00001110, and the register X with the multiplier 1001. Assume that each of these registers is loaded using its asynchronous inputs(i.e., PRESET and CLEAR). The output of the adder will be the sum of A and B,i.e., 00001110.
2. **First Clock pulse***:*Since the LSB of the multiplier (X0) is a 1, the first clock pulse gets through the AND gate and its positive going transition transfers the sum outputs into the accumulator. The subsequent negative going transition causes the X and B registers to shift right and left, respectively. This produces a new sum of A and B.
3. **Second Clock Pulse:** The second bit of the original multiplier is now in X0 . Since this bit is a 0, the second clock pulse is inhibited from reaching the accumulator. Thus, the sum outputs are not transferred into the accumulator and the number in the accumulator does not change. The negative going transition of the clock pulse will again shift the X and B registers. Again a new sum is produced.
4. **Third Clock Pulse**:The third bit of the original multiplier is now in X0;since this bit is a 0, the third clock pulse is inhibited from reaching the accumulator. Thus, the sum outputs are not transferred into the accumulator and the number in the accumulator does not change. The negative going transition of the clock pulse will again shift the X and B registers. Again a new sum is produced.
5. **Fourth Clock Pulse:** The last bit of the original multiplier is now in X0 , and since it is a 1, the positive going transition of the fourth pulse transfers the sum into the accumulator. The accumulator now holds the final product. The negative going transition of the clock pulse shifts X and B again. Note that, X is now 0000, since all the multiplier bits have been shifted out.

Code converters:

The availability of a large variety of codes for the same discrete elements of information results in the use of different codes by different digital systems. It is sometimes necessary to use the output of one system as the input to another. A conversion circuit must be inserted between the two systems if each uses different codes for the same information. Thus a code converter is a logic circuit whose inputs are bit patterns representing numbers (or character) in one cod and whose outputs are the corresponding representation in a different code. Code converters are usually multiple output circuits.

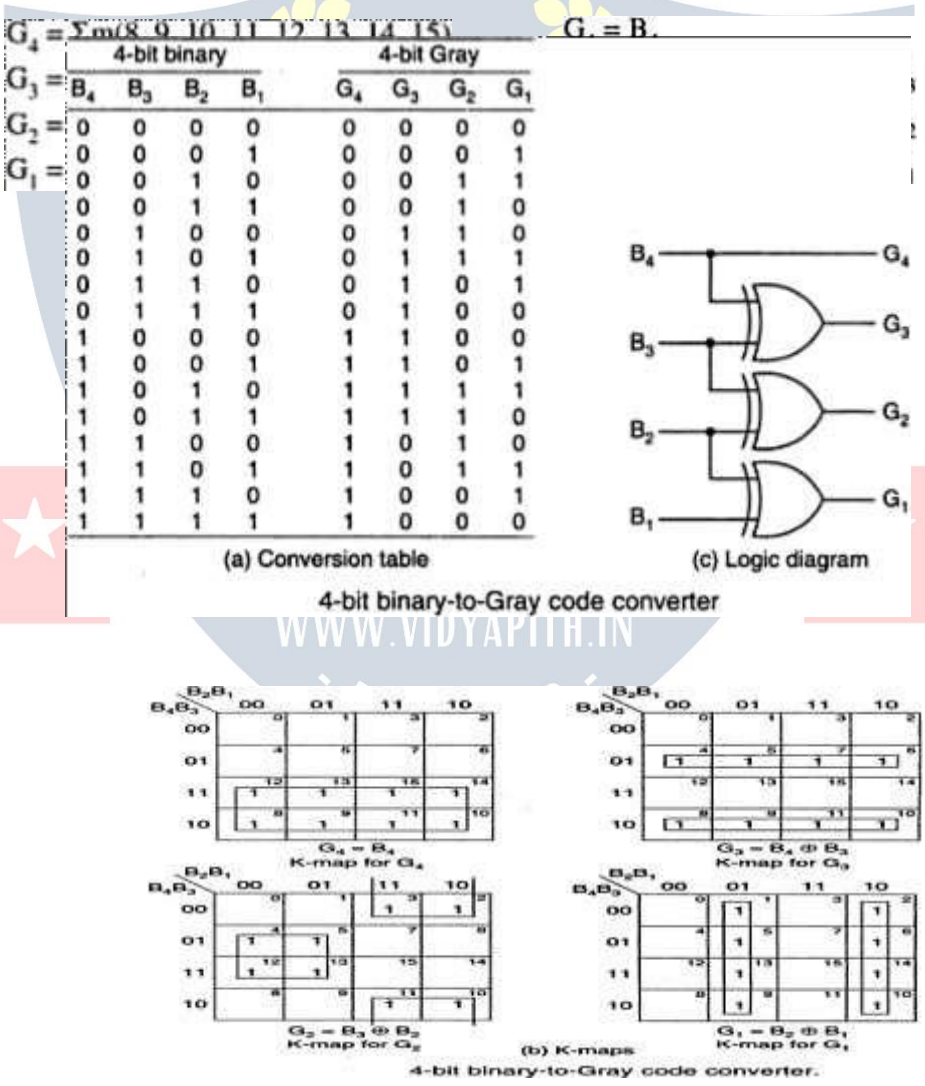
To convert from binary code A to binary code B, the input lines must supply the bit combination of elements as specified by code A and the output lines must generate the corresponding bit combination of code B. A combinational circuit performs this transformation by means of logic gates.

For example, a binary –to-gray code converter has four binary input lines B4, B3,B2,B1 and four gray code output lines G4,G3,G2,G1. When the input is 0010, for instance, the output should be 0011 and so forth. To design a code converter, we use a code table treating it as a truth table to express each output as a Boolean algebraic function of all the inputs.

In this example, of binary –to-gray code conversion, we can treat the binary to the gray code table as four truth tables to derive expressions for G4, G3, G2, and G1. Each of these four expressions would, in general, contain all the four input variables B4, B3,B2,and B1. Thus,this code converter is actually equivalent to four logic circuits, one for each of the truth tables.

The logic expression derived for the code converter can be simplified using the usual techniques, including _don‘t cares‘ if present. Even if the input is an unweighted code, the same cell numbering method which we used earlier can be used, but the cell numbers --must correspond to the input combinations as if they were an 8-4-2-1 weighted code.

Design of a 4-bit binary to gray code converter:



Design of a 4-bit gray to Binary code converter:

$B_4 = \Sigma\, m(12, 13, 15, 14, 10, 11, 9, 8) = \Sigma\, m(8, 9, 10, 11, 12, 13, 14, 15)$
$B_3 = \Sigma\, m(6, 7, 5, 4, 10, 11, 9, 8) = \Sigma\, m(4, 5, 6, 7, 8, 9, 10, 11)$
$B_2 = \Sigma\, m(3, 2, 5, 4, 15, 14, 9, 8) = \Sigma\, m(2, 3, 4, 5, 8, 9, 14, 15)$
$B_1 = \Sigma\, m(1, 2, 7, 4, 13, 14, 11, 8) = \Sigma\, m(1, 2, 4, 7, 8, 11, 13, 14)$

$B_4 = G_4$

$B_3 = \bar{G}_4 G_3 + G_4 \bar{G}_3 = G_4 \oplus G_3$

$B_2 = \bar{G}_4 G_3 \bar{G}_2 + \bar{G}_4 \bar{G}_3 G_2 + G_4 \bar{G}_3 \bar{G}_2 + G_4 G_3 G_2$

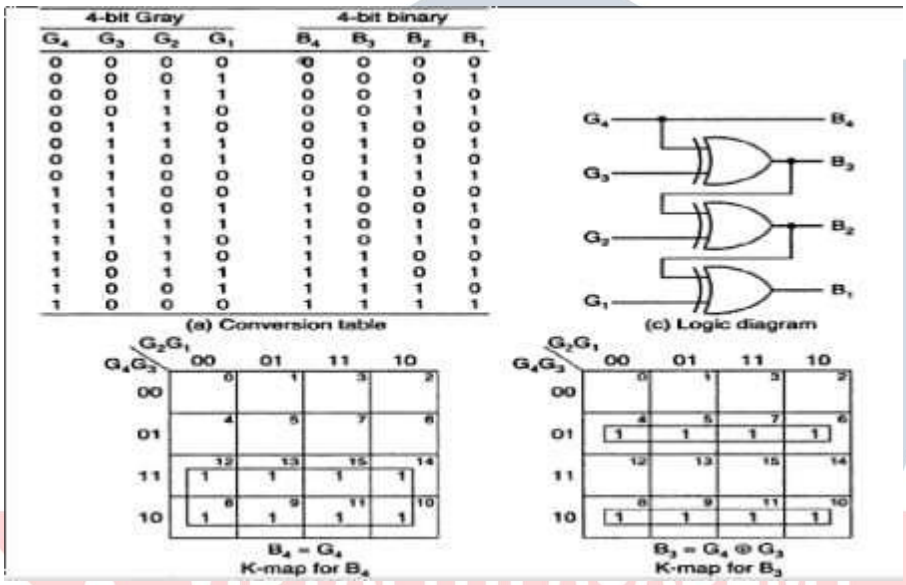
$= \bar{G}_4(G_3 \oplus G_2) + G_4(\overline{G_3 \oplus G_2}) = G_4 \oplus G_3 \oplus G_2 = B_3 \oplus G_2$

$B_1 = \bar{G}_4 \bar{G}_3 \bar{G}_2 G_1 + \bar{G}_4 \bar{G}_3 G_2 \bar{G}_1 + \bar{G}_4 G_3 G_2 G_1 + \bar{G}_4 G_3 \bar{G}_2 \bar{G}_1 + G_4 G_3 \bar{G}_2 G_1$
$+ G_4 G_3 G_2 \bar{G}_1 + G_4 \bar{G}_3 G_2 G_1 + G_4 \bar{G}_3 \bar{G}_2 \bar{G}_1$

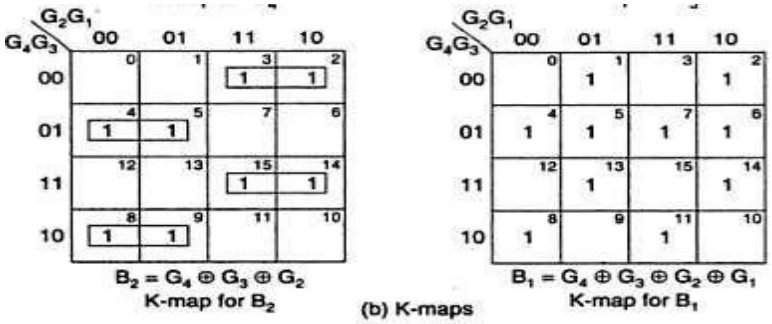
$= \bar{G}_4 \bar{G}_3(G_2 \oplus G_1) + G_4 G_3(G_2 \oplus G_1) + \bar{G}_4 G_3(\overline{G_2 \oplus G_1}) + G_4 \bar{G}_3(\overline{G_2 \oplus G_1})$

$= (G_2 \oplus G_1)(\overline{G_4 \oplus G_3}) + (\overline{G_2 \oplus G_1})(G_4 \oplus G_3)$

$= G_4 \oplus G_3 \oplus G_2 \oplus G_1$



4-bit Gray				4-bit binary			
G_4	G_3	G_2	G_1	B_4	B_3	B_2	B_1
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	1	0	0	1	0
0	0	1	0	0	0	1	1
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	1
0	1	0	1	0	1	1	0
0	1	0	0	0	1	1	1
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	1
1	1	1	1	1	0	1	0
1	1	1	0	1	0	1	1
1	0	1	0	1	1	0	0
1	0	1	1	1	1	0	1
1	0	0	1	1	1	1	0
1	0	0	0	1	1	1	1



(b) K-maps

4-bit Gray-to-binary code converter.

Design of a 4-bit BCD to XS-3 code converter:

8421 code				XS-3 code			
B_4	B_3	B_2	B_1	X_4	X_3	X_2	X_1
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

(a) Conversion table

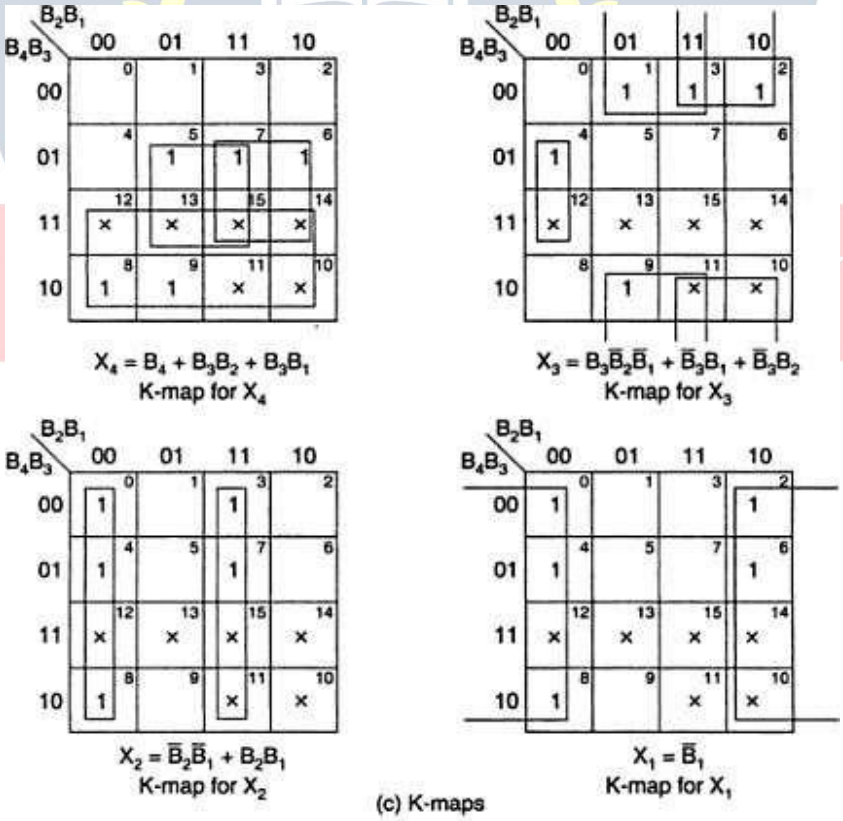
$X_4 = \Sigma\, m(5, 6, 7, 8, 9) + d(10, 11, 12, 13, 14, 15)$
$X_3 = \Sigma\, m(1, 2, 3, 4, 9) + d(10, 11, 12, 13, 14, 15)$
$X_2 = \Sigma\, m(0, 3, 4, 7, 8) + d(10, 11, 12, 13, 14, 15)$
$X_1 = \Sigma\, m(0, 2, 4, 6, 8) + d(10, 11, 12, 13, 14, 15)$

The minimal expressions are

$X_4 = B_4 + B_3B_2 + B_3B_1$
$X_3 = B_3\bar{B}_2\bar{B}_1 + \bar{B}_3B_1 + \bar{B}_3B_2$
$X_2 = \bar{B}_2\bar{B}_1 + B_2B_1$
$X_1 = \bar{B}_1$

(b) Minimal expressions

4-bit BCD-to-XS-3 code converter



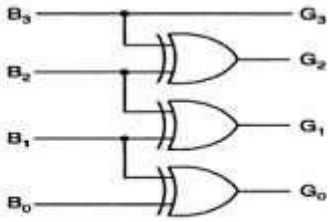
(c) K-maps

4-bit BCD-to-XS-3 code converter.

Design of a BCD to gray code converter:

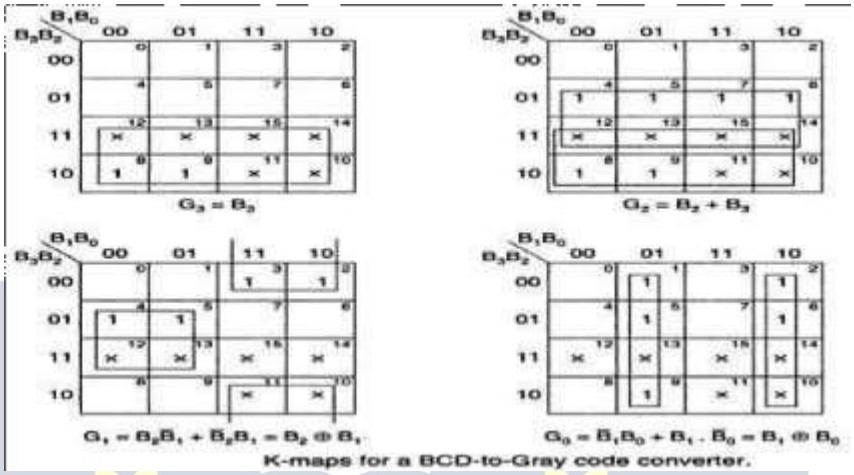
BCD code				Gray code			
B_3	B_2	B_1	B_0	G_3	G_2	G_1	G_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1

(a) BCD-to-Gray code conversion table



(b) Logic diagram

BCD-to-Gray code converter.

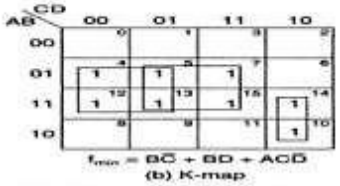


K-maps for a BCD-to-Gray code converter.

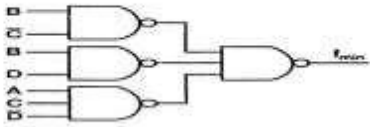
Design of a SOP circuit to Detect the Decimal numbers 5 through 12 in a 4-bit gray code Input:

Decimal number	4-bit Gray code A	B	C	D	Output f
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	1	0
3	0	0	1	0	0
4	0	1	1	0	0
5	0	1	1	1	1
6	0	1	0	1	1
7	0	1	0	0	1
8	1	1	0	0	1
9	1	1	0	1	1
10	1	1	1	1	1
11	1	1	1	0	1
12	1	0	1	0	1
13	1	0	1	1	0
14	1	0	0	1	0
15	1	0	0	0	0

(a) Truth table



(b) K-map



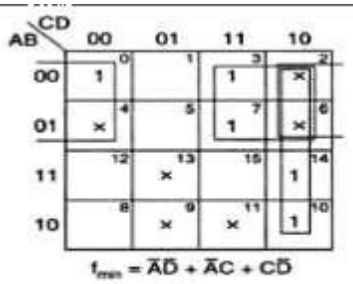
(c) NAND logic

Truth table, K-map and logic diagram for the SOP circuit.

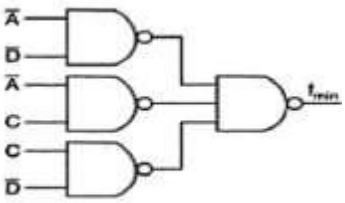
Design of a SOP circuit to detect the decimal numbers 0,2,4,6,8 in a 4-bit 5211 BCD code input:

Decimal number	5211 code A	B	C	D	Output f
0	0	0	0	0	1
1	0	0	0	1	0
2	0	0	1	1	1
3	0	1	0	1	0
4	0	1	1	1	1
5	1	0	0	0	0
6	1	0	1	0	1
7	1	1	0	0	0
8	1	1	1	0	1
9	1	1	1	1	0

(a) Truth table



(b) K-map



(c) Logic diagram

Truth table, K-map and logic diagram for the SOP circuit.

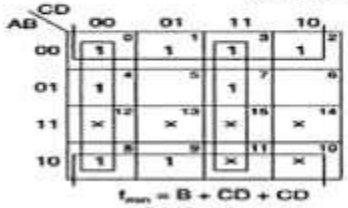
Design of a Combinational circuit to produce the 2's complement of a 4-bit binary number:

Input A	B	C	D	Output E	F	G	H
0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1
0	0	1	0	1	1	1	0
0	0	1	1	1	1	0	1
0	1	0	0	1	1	0	0
0	1	0	1	1	0	1	1
0	1	1	0	1	0	1	0
0	1	1	1	1	0	0	1
1	0	0	0	1	0	0	0
1	0	0	1	0	1	1	1
1	0	1	0	0	1	1	0
1	0	1	1	0	1	0	1
1	1	0	0	0	1	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	0	1	0
1	1	1	1	0	0	0	1

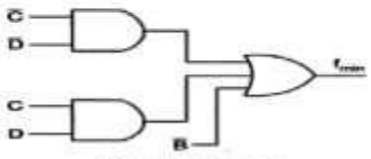
(a) Conversion table

Conversion table and K-maps for the circuit

(a) Seven-segment display



(b) K-map



(c) Logic diagram

Comparators:

$$\text{EQUALITY} = (A_3 \odot B_3)(A_2 \odot B_2)(A_1 \odot B_1)(A_0 \odot B_0)$$

Block diagram of a 1-bit comparator.

The logic for a 1-bit magnitude comparator: Let the 1-bit numbers be $A = A_0$ and $B = B_0$.
If $A_0 = 1$ and $B_0 = 0$, then $A > B$.
Therefore,

$$A > B: G = A_0\bar{B}_0$$

If $A_0 = 0$ and $B_0 = 1$, then $A < B$.
Therefore,

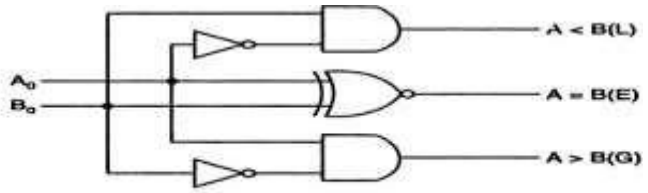
$$A < B: L = \bar{A}_0 B_0$$

If A_0 and B_0 coincide, i.e. $A_0 = B_0 = 0$ or if $A_0 = B_0 = 1$, then $A = B$.
Therefore,

$$A = B : E = A_0 \odot B_0$$

A_0	B_0	L	E	G
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

(a) Truth table



(b) Logic diagram

1-bit comparator.

1. Magnitude Comparator:

1- bit Magnitude Comparator:

The logic for a 2-bit magnitude comparator: Let the two 2-bit numbers be $A = A_1 A_0$ and $B = B_1 B_0$.

1. If $A_1 = 1$ and $B_1 = 0$, then $A > B$ or
2. If A_1 and B_1 coincide and $A_0 = 1$ and $B_0 = 0$, then $A > B$. So the logic expression for $A > B$ is

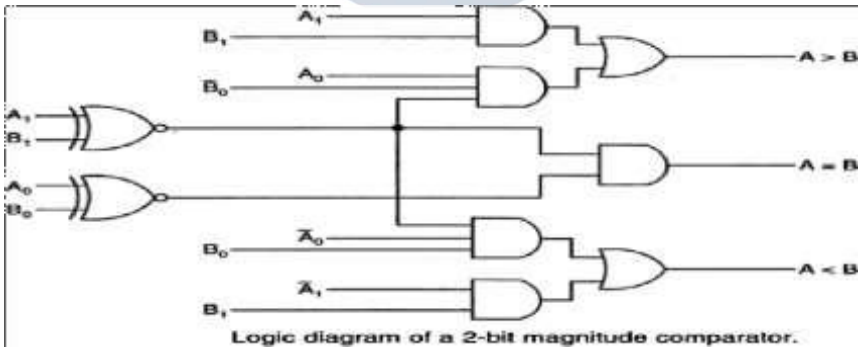
$$A > B : G = A_1\bar{B}_1 + (A_1 \odot B_1)A_0\bar{B}_0$$

1. If $A_1 = 0$ and $B_1 = 1$, then $A < B$ or
2. If A_1 and B_1 coincide and $A_0 = 0$ and $B_0 = 1$, then $A < B$. So the expression for $A < B$ is

$$A < B : L = \bar{A}_1 B_1 + (A_1 \odot B_1)\bar{A}_0 B_0$$

If A_1 and B_1 coincide and if A_0 and B_0 coincide then $A = B$. So the expression for $A = B$ is

$$A = B : E = (A_1 \odot B_1)(A_0 \odot B_0)$$



Logic diagram of a 2-bit magnitude comparator.

4- Bit Magnitude Comparator:

The logic for a 4-bit magnitude comparator: Let the two 4-bit numbers be $A = A_3A_2A_1A_0$ and $B = B_3B_2B_1B_0$.

1. If $A_3 = 1$ and $B_3 = 0$, then $A > B$. Or
2. If A_3 and B_3 coincide, and if $A_2 = 1$ and $B_2 = 0$, then $A > B$. Or
3. If A_3 and B_3 coincide, and if A_2 and B_2 coincide, and if $A_1 = 1$ and $B_1 = 0$, then $A > B$. Or
4. If A_3 and B_3 coincide, and if A_2 and B_2 coincide, and if A_1 and B_1 coincide, and if $A_0 = 1$ and $B_0 = 0$, then $A > B$.

From these statements, we see that the logic expression for $A > B$ can be written as

$$(A > B) = A_3\overline{B}_3 + (A_3 \odot B_3)A_2\overline{B}_2 + (A_3 \odot B_3)(A_2 \odot B_2)A_1\overline{B}_1 + (A_3 \odot B_3)(A_2 \odot B_2)(A_1 \odot B_1)A_0\overline{B}_0$$

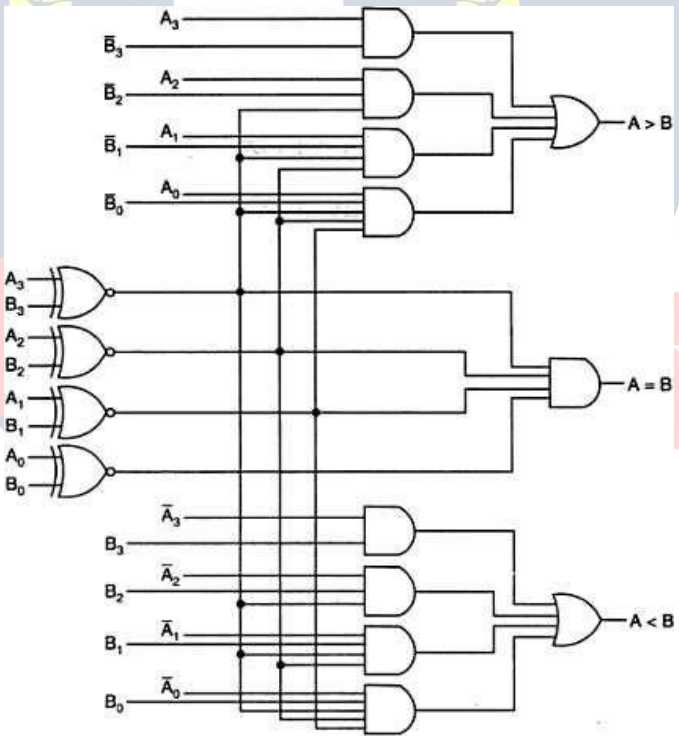
Similarly, the logic expression for $A < B$ can be written as

$$A < B = \overline{A}_3B_3 + (A_3 \odot B_3)\overline{A}_2B_2 + (A_3 \odot B_3)(A_2 \odot B_2)\overline{A}_1B_1 + (A_3 \odot B_3)(A_2 \odot B_2)(A_1 \odot B_1)\overline{A}_0B_0$$

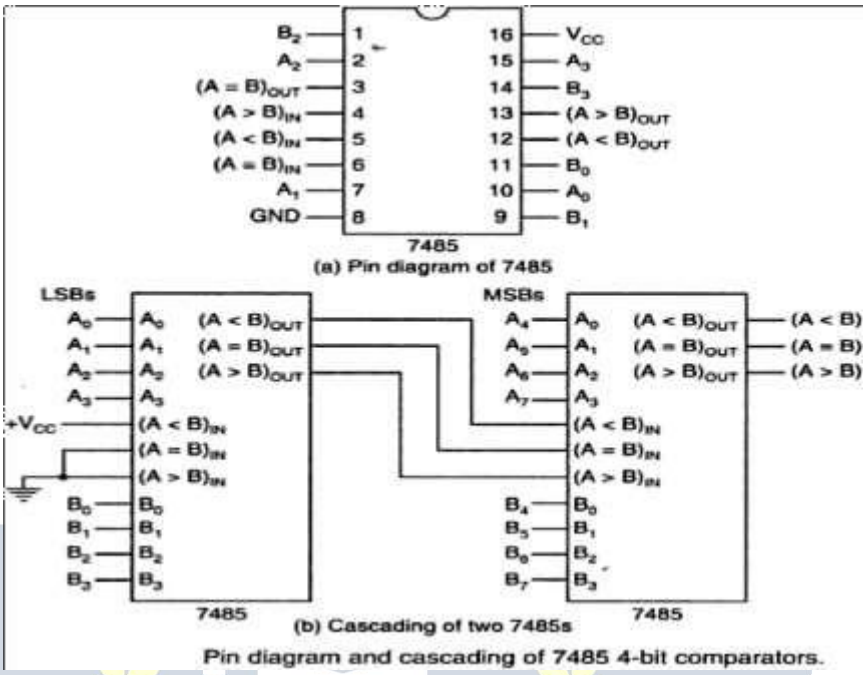
If A_3 and B_3 coincide and if A_2 and B_2 coincide and if A_1 and B_1 coincide and if A_0 and B_0 coincide, then $A = B$.

So the expression for $A = B$ can be written as

$$(A = B) = (A_3 \odot B_3)(A_2 \odot B_2)(A_1 \odot B_1)(A_0 \odot B_0)$$

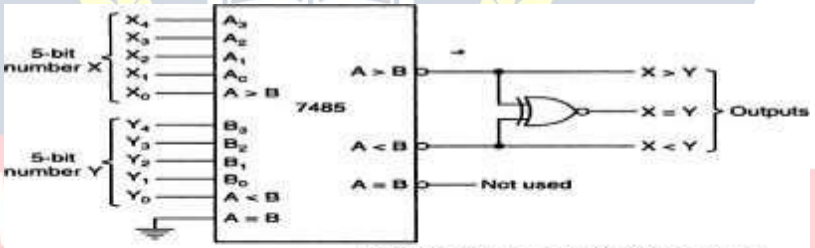


IC Comparator:

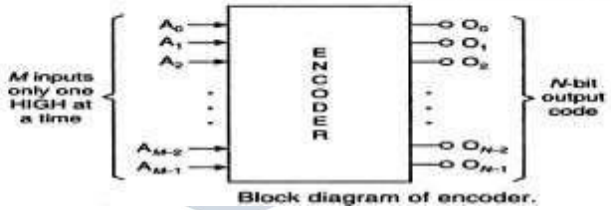


Pin diagram and cascading of 7485 4-bit comparators.

ENCODERS:

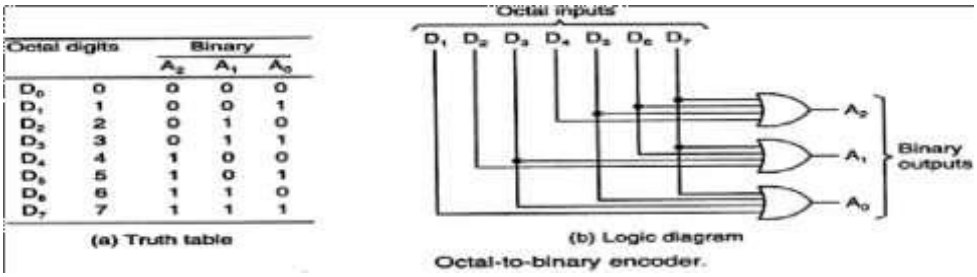


Use of 7485 as a 5-bit comparator.

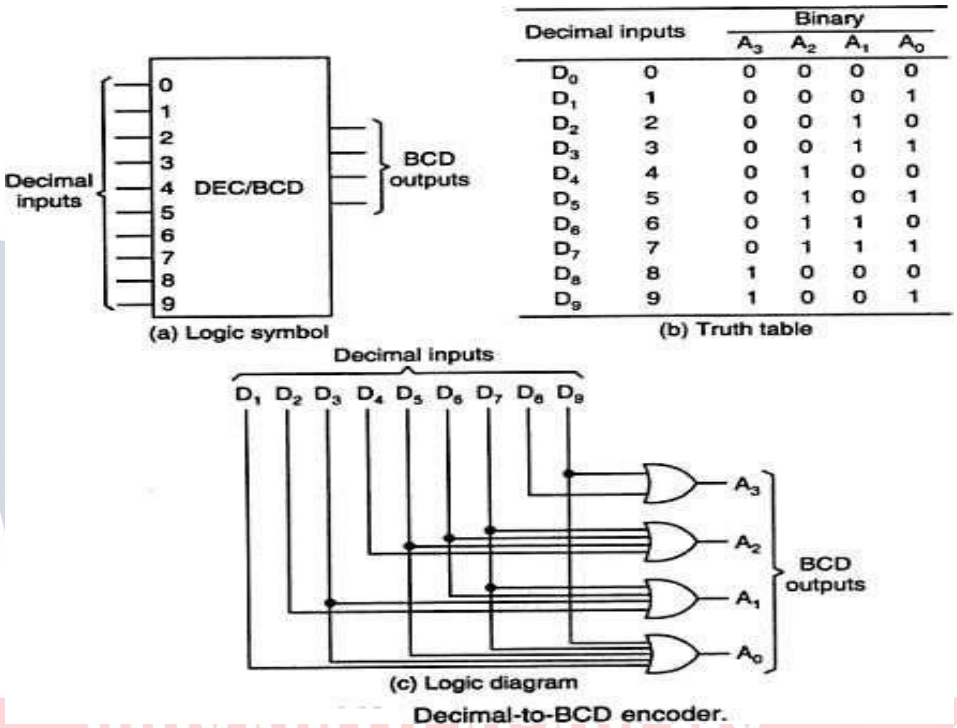


Block diagram of encoder.

Octal to Binary Encoder:



Decimal to BCD Encoder:



Tristate bus system:

In digital electronics **three-state**, **tri-state**, or **3-state**logic allows an output port to assume a high impedance state in addition to the 0 and 1 logic levels, effectively removing the output from the circuit. This allows multiple circuits to share the same output line or lines (such as a bus which cannot listen to more than one device at a time).

Three-state outputs are implemented in many registers, bus drivers, and flip-flops in the 7400 and 4000 series as well as in other types, but also internally in many integrated circuits. Other typical uses are internal and external buses in microprocessors, computer memory, and peripherals. Many devices are controlled by an active-low input called OE (Output Enable) which dictates whether the outputs should be held in a highimpedance state or drive their respective loads (to either 0- or 1-level).

A tristate buffer can be thought of as a switch. If *B* is on, the switch is closed. If B is off, the switch is open.

INPUT		OUTPUT
A	B	C
0	1	0
1		1
X	0	Z (high impedance)

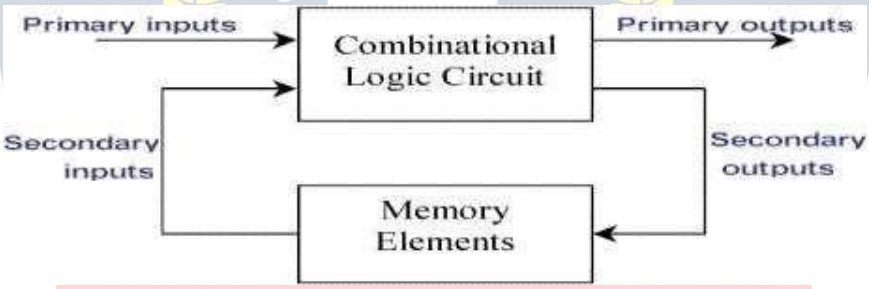
UNIT-IV

SEQUENTIAL CIRCUITS

Classification of sequential circuits: Sequential circuits may be classified as two types.

1. Synchronous sequentialcircuits
2. Asynchronous sequentialcircuits

Combinational logic refers to circuits whose output is strictly depended on the present value of the inputs. As soon as inputs are changed, the information about the previous inputs is lost, that is, combinational logics circuits have no memory. Although every digital system is likely to have combinational circuits, most systems encountered in practice also include memory elements, which require that the system be described in terms of sequential logic. Circuits whose output depends not only on the present input value but also the past input value are known as **sequential logic circuits**. The mathematical model of a sequential circuit is usually referred to as a **sequential machine**.



Comparison between combinational and sequential circuits

Combinational circuit	Sequential circuit
1. In combinational circuits, the output variables at any instant of time are dependent only on the present input Variables	1. in sequential circuits the output variables at any instant of time are dependent not only on the present input variables, but also on the present state
2.memory unit is not requires in combinational circuit	2.memory unit is required to store the past history of the input variables
3. these circuits are faster because the delay between the i/p and o/p	3. sequential circuits are slower than combinational circuits due to propagation delay of gates only

4. easy to design	4. comparatively hard to design

Level mode and pulse mode asynchronous sequential circuits:

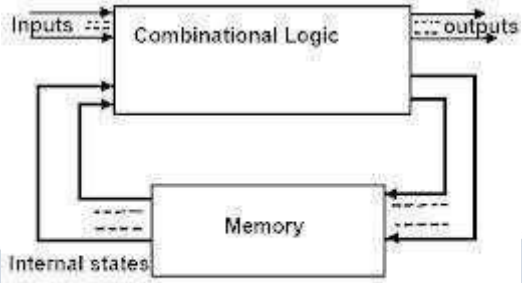


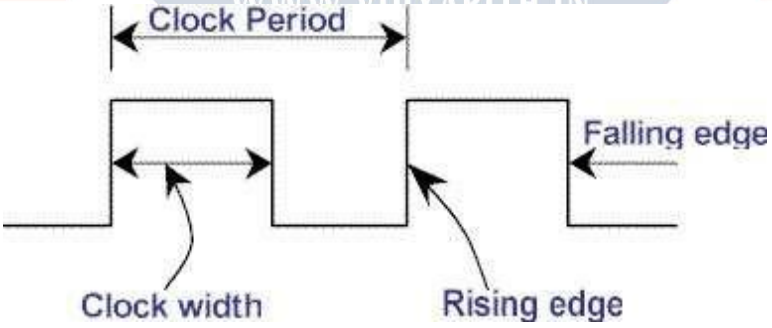
Figure 1: Asynchronous Sequential Circuit

Fig shows a block diagram of an asynchronous sequential circuit. It consists of a combinational circuit and delay elements connected to form the feedbackloops. The present state and next state variables in asynchronous sequential circuits called secondary variables and excitation variables respectively..

There are two types of asynchronous circuits: fundamental mode circuits and pulse mode circuits.

Synchronous and Asynchronous Operation:

Sequential circuits are divided into two main types: **synchronous** and **asynchronous**. Their classification depends on the timing of their signals.***Synchronous*** sequential circuits change their states and output values at discrete instants of time, which are specified by the rising and falling edge of a free-running **clock signal**. The clock signal is generally some form of square wave as shown in Figure below.



From the diagram you can see that the **clock period** is the time between successive transitions in the same direction, that is, between two rising or two falling edges. State transitions in synchronous sequential circuits are made to take place at times when the clock is making a transition from 0 to 1 (rising edge) or from 1 to 0 (falling edge). Between successive clock pulses there is no change in the information stored in memory.

The reciprocal of the clock period is referred to as the **clock frequency**. The **clock width** is defined as the time during which the value of the clock signal is equal to 1. The ratio of the clock width and clock period is referred to as the duty cycle. A clock signal is said to be **active high** if the state changes occur at the clock's rising edge or during the clock width. Otherwise, the clock is said to be **active low**. Synchronous sequential circuits are also known as **clocked sequential circuits**.

The memory elements used in synchronous sequential circuits are usually flip-flops. These circuits are binary cells capable of storing one bit of information. A flip-flop circuit has two outputs, one for the normal value and one for the complement value of the bit stored in it. Binary information can enter a flip-flop in a variety of ways, a fact which give rise to the different types of flip-flops. For information on the different types of basic flip-flop circuits and their logical properties, see the previous tutorial on flip-flops.

In ***asynchronous*** sequential circuits, the transition from one state to another is initiated by the change in the primary inputs; there is no external synchronization. The memory commonly used in asynchronous sequential circuits are time-delayed devices, usually implemented by feedback among logic gates. Thus, asynchronous sequential circuits may be regarded as combinational circuits with feedback. Because of the feedback among logic gates, asynchronous sequential circuits may, at times, become unstable due to transient conditions. The instability problem imposes many difficulties on the designer. Hence, they are not as commonly used as synchronous systems.

Fundamental Mode Circuits assumes that:

1. The input variables change only when the circuit is stable
2. Only one input variable can change at a giventime
3. Inputs are levels are not pulses

A pulse mode circuit assumes that:

1. The input variables are pulses instead of levels
2. The width of the pulses is long enough for the circuit to respond to the input
3. The pulse width must not be so long that is still present after the new state is reached.

Latches and flip-flops

Latches and flip-flops are the basic elements for storing information. One latch or flip- flop can store one bit of information. The main difference between latches and flip-flops is that for latches, their outputs are constantly affected by their inputs as long as the enable signal is asserted. In other words, when they are enabled, their content changes immediately when their inputs change. Flip-flops, on the other hand, have their content change only either at the rising or falling edge of the enable signal. This enable signal is usually the controlling clock signal. After the rising or falling edge of the clock, the flip-flop content remains constant even if the input changes.

There are basically four main types of latches and flip-flops: SR, D, JK, and T. The major differences in these flip-flop types are the number of inputs they have and how they change state. For each type, there are also different variations that enhance their operations. In this chapter, we will look at the operations of the various latches and flip-flops.the flip-flops has two outputs, labeled Q and Q‘. the Q output is the normal output of the flip flop and Q‘ is the inverted output.

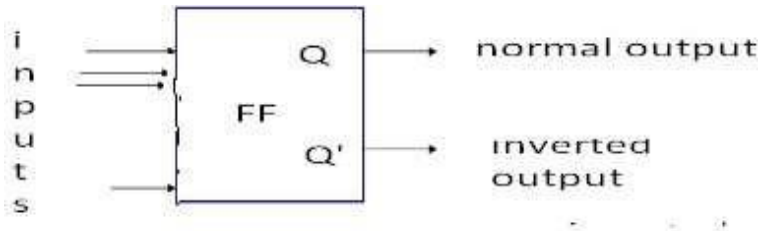


Figure: basic symbol of flipflop

A latch may be an active-high input latch or an active –LOW input latch.active –HIGH means that the SET and RESET inputs are normally resting in the low state and one of them will be pulsed high whenever we want to change latch outputs.

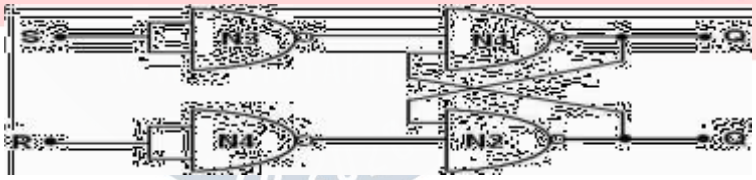
SR latch:

The latch has two outputs Q and Q‘. When the circuit is switched on the latch may enter into any state. If Q=1, then Q‘=0, which is called SET state. If Q=0, then Q‘=1, which is called RESET state. Whether the latch is in SET state or RESET state, it will continue to remain in the same state, as long as the power is not switched off. But the latch is not an useful circuit, since there is no way of entering the desired input. It is the fundamental building block in constructing flip-flops, as explained in the following sections

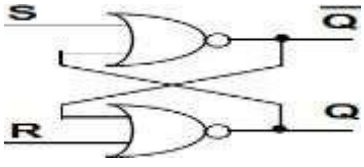
NAND latch

NAND latch is the fundamental building block in constructing a flip-flop. It has the property of holding on to any previous output, as long as it is not disturbed.

The opration of NAND latch is the reverse of the operation of NOR latch.if 0‘s are replaced by 1‘s and 1‘s are replaced by 0‘s we get the same truth table as that of the NOR latch shown



NOR latch



S	R	Q	$\overline{Q}$	Function
0	0	Q^+	$\overline{Q}^+$	Storage State
0	1	0	1	Reset
1	0	1	0	Set
1	1	0-?	0-?	Indeterminate State

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

1. SET=0, RESET=0: this is normal resting state of the NOR latch and it has no effect on the output state. Q and Q‘ will remain in whatever stste they were prior to the occurrence of this input condition.
2. SET=1, RESET=0: this will always set Q=1, where it will remain even after SET returns to 0
3. SET=0, RESET=1: this will always reset Q=0, where it will remain even after RESET returns to 0
4. SET=1,RESET=1; this condition tries to SET and RESET the latch at the same time, and it produces Q=Q‘=0. If the inputs are returned to zero simultaneously, the resulting output stste is erratic and unpredictable. This input condition should not be used.

The SET and RESET inputs are normally in the LOW state and one of them will be pulsed HIGH. Whenever we want to change the latch outputs..

RS Flip-flop:

The basic flip-flop is a one bit memory cell that gives the fundamental idea of memory device. It constructed using two NAND gates. The two NAND gates N1 andN2 are connected such that, output of N1 is connected to input of N2 and output of N2 to input of N1. These form the feedback path the inputs are S and R, and outputs are Q and Q‘. The logic diagram and the block diagram of R-S flip-flop with clocked input

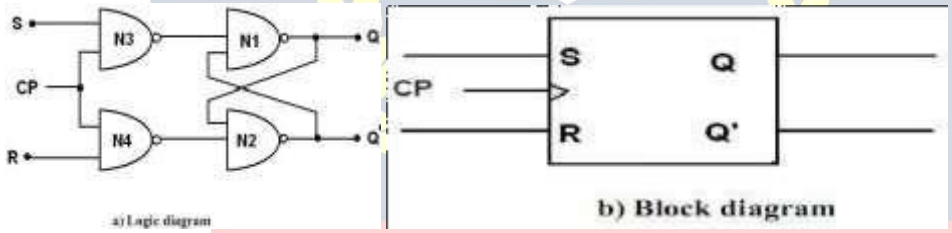


Figure: RS Flip-flop

The flip-flop can be made to respond only during the occurrence of clock pulse by adding two NAND gates to the input latch. So synchronization is achieved. i.e., flip-flops are allowed to change their states only at particular instant of time. The clock pulses are generated by a clock pulse generator. The flip-flops are affected only with the arrival of clock pulse**.**

Operation:

1. When CP=0 the output of N3 and N4 are 1 regardless of the value of S and R. This is given as input to N1 and N2. This makes the previous value of Q and Q‘unchanged.
2. When CP=1 the information at S and R inputs are allowed to reach the latch and change of state in flip-flop takes place.
3. CP=1, S=1, R=0 gives the SET state i.e., Q=1, Q‘=0.
4. CP=1, S=0, R=1 gives the RESET state i.e., Q=0, Q‘=1.
5. CP=1, S=0, R=0 does not affect the state offlip-flop.
6. CP=1, S=1, R=1 is not allowed, because it is not able to determine the next state. This condit ion is said to be a ―race condition.

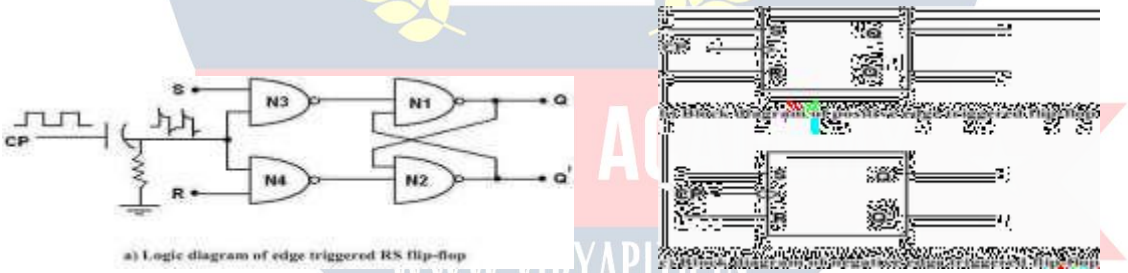
In the logic symbol CP input is marked with a triangle. It indicates the circuit responds to an input change from 0 to 1. The characteristic table gives the operation conditions of flip-flop. Q(t) is the present state maintained in the flip-flop at time _t'. Q(t+1) is the state after the occurrence of clock pulse.

Truth table

S	R	$Q_{(t+1)}$	Comments
0	0	Q_t	No change
0	1	0	Reset / clear
1	0	1	Set
1	1	*	Not allowed

Edge triggered RS flip-flop:

Some flip-flops have an RC circuit at the input next to the clock pulse. By the design of the circuit the R-C time constant is much smaller than the width of the clock pulse. So the output changes will occur only at specific level of clock pulse. The capacitor gets fully charged when clock pulse goes from low to high. This change produces a narrow positive spike. Later at the trailing edge it produces narrow negative spike. This operation is called edge triggering, as the flip-flop responds only at the changing state of clock pulse. If output transition occurs at rising edge of clock pulse (0 1),it is called positively edge triggering. If it occurs at trailing edge (1 0) it is called negative edge triggering. Figure shows the logic and block diagram.

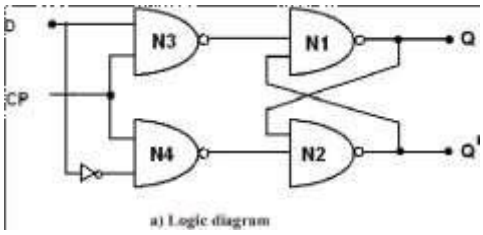


a) Logic diagram of edge triggered RS flip-flop

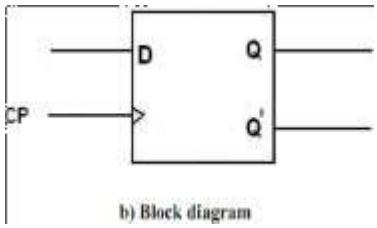
Figure: Edge triggered RS flip-flop D

flip-flop:

The D flip-flop is the modified form of R-S flip-flop. R-S flip-flop is converted to D flip-flop by adding an inverter between S and R and only one input D is taken instead of S and R. So one input is D and complement of D is given as another input. The logic diagram and the block diagram of D flip-flop with clocked input



a) Logic diagram



b) Block diagram

When the clock is low both the NAND gates (N1 and N2) are disabled and Q retains its last value. When clock is high both the gates are enabled and the input value at D is transferred to its output Q. D flip-flop is also called ―Data flip-flop‖.

Truth table

CP	D	Q
0	x	Previous state
1	0	0
1	1	1

Edge Triggered D Flip-flop:

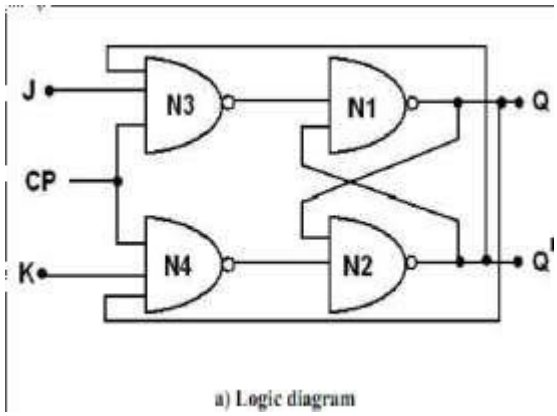
Truth table

PRESET	CLEAR	CP	D	Q
0	0	X	X	*(forbidden)
0	1	X	X	1
1	0	X	X	0
1	0	0	X	NC
1	1	1	X	NC
1	1	↓	X	NC
1	1	↑	0	0
1	1	↑	1	1

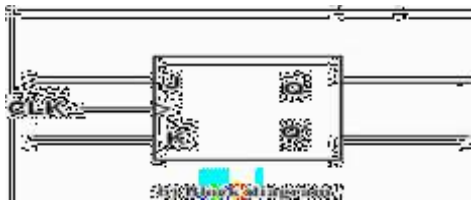
Figure: truth table, block diagram, logic diagram of edge triggered flip-flop

JK flip-flop (edge triggered JK flip-flop)

The race condition in RS flip-flop, when R=S=1 is eliminated in J-K flip-flop. There is a feedback from the output to the inputs. Figure 3.4 represents one way of building a JK flip-flop**.**



a) Logic diagram



Truth table

J	K	$Q_{(t+1)}$	Comments
0	0	Q_t	No change
0	1	0	Reset / clear
1	0	1	Set
1	1	Q'_t	Complement/ toggle.

Figure: JK flip-flop

The J and K are called control inputs, because they determine what the flip-flop does when a positive clock edge arrives.

Operation:

1. When J=0, K=0 then both N3 and N4 will produce high output and the previous value of Q and Q‘ retained as it is.
2. When J=0, K=1, N3 will get an output as 1 and output of N4 depends on the value of Q. The final output is Q=0, Q‘=1 i.e., reset state
3. When J=1, K=0 the output of N4 is 1 and N3 depends on the value of Q‘. The final output is Q=1 and Q‘=0 i.e., set state
4. When J=1, K=1 it is possible to set (or) reset the flip-flop depending on the current state of output. If Q=1, Q‘=0 then N4 passes ‘0‘to N2 which produces Q‘=1, Q=0 which is reset state. When J=1, K=1, Q changes to the complement of the last state. The flip-flop is said to be in the toggle state.

The characteristic equation of the JK flip-flop is:

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

JK flip-flop operation

Characteristic table				Excitation table				
J	**K**	**Q_{next}**	**Comment**	**Q**	**Q_{next}**	**J**	**K**	**Comment**
0	0	Q	hold state	0	0	0	X	No change
0	1	0	reset	0	1	1	X	Set
1	0	1	set	1	0	X	1	Reset
1	1	Q	toggle	1	1	X	0	No change

T flip-flop:

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

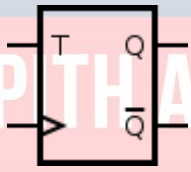


Figure : symbol for T flip flop

$$Q_{next} = T \oplus Q = T\overline{Q} + \overline{T}Q$$ (expanding the XOR operator)

When T is held high, the toggle flip-flop divides the clock frequency by two; that is, if clock frequency is 4 MHz, the output frequency obtained from the flip-flop will be 2 MHz This "divide by" feature has application in various types of digital counters. A T flip-flop can also be built using a JK flip-flop (J & K pins are connected together and act as T) or D flip-flop (T input and $P_{revious}$ is connected to the D input through an XOR gate).

T flip-flop operation[28]

Characteristic table				Excitation table			
T	Q	Q_{next}	**Comment**	Q	Q_{next}	T	**Comment**
0	0	0	hold state (no clk)	0	0	0	No change
0	1	1	hold state (no clk)	1	1	0	No change
1	0	1	Toggle	0	1	1	Complement
1	1	0	Toggle	1	0	1	Complement

Flip flop operating characteristics:

The operation characteristics specify the performance, operating requirements, and operating limitations of the circuits. The operation characteristics mentions here apply to all flip- flops regardless of the particular form of the circuit.

Propagation Delay Time: is the interval of time required after an input signal has been applied for the resulting output change to occur.

Set-up Time: is the minimum interval required for the logic levels to be maintained constantly on the inputs (J and K, or S and R, or D) prior to the triggering edge of the clock pulse in order for the levels to be reliably clocked into the flip-flop.

Hold Time: is the minimum interval required for the logic levels to remain on the inputs after the triggering edge of the clock pulse in order for the levels to be reliably clocked into the flip- flop.

Maximum Clock Frequency: is the highest rate that a flip-flop can be reliably triggered. **Power**

Dissipation: is the total power consumption of the device. It is equal to product of supply voltage (Vcc) and the current (Icc).

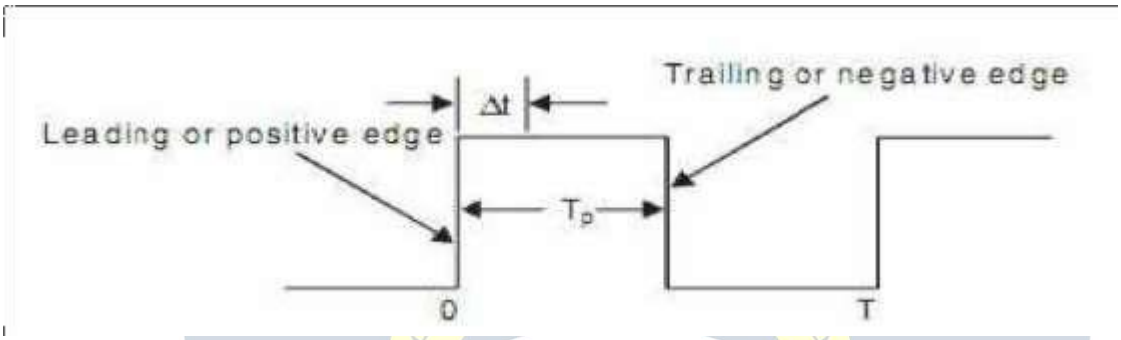
$$P=V_{cc}.I_{cc}$$

The power dissipation of a flip flop is usually in mW.

Pulse Widths: are the minimum pulse widths specified by the manufacturer for the Clock, SET and CLEAR inputs.

Clock transition times: for reliable triggering, the clock waveform transition times should be kept very short. If the clock signal takes too long to make the transitions from one level to other, the flip flop may either triggering erratically or not trigger at all. Race around Condition

The inherent difficulty of an S-R flip-flop (i.e., S = R = 1) is eliminated by using the feedback connections from the outputs to the inputs of gate 1 and gate 2 as shown in Figure. Truth tables in figure were formed with the assumption that the inputs do not change during the clock pulse (CLK = 1). But the consideration is not true because of the feedback connections.



- Consider, for example, that the inputs are J = K = 1 and Q = 1, and a pulse as shown in Figure is applied at the clock input.
- After a time interval t equal to the propagation delay through two NAND gates in series, the outputs will change to Q = 0. So now we have J = K = 1 and Q = 0.
- After another time interval of t the output will change back to Q = 1. Hence, we conclude that for the time duration of tP of the clock pulse, the output will oscillate between 0 and 1. Hence, at the end of the clock pulse, the value of the output is not certain. This situation is referred to as a race-around condition.
- Generally, the propagation delay of TTL gates is of the order of nanoseconds. So if the clock pulse is of the order of microseconds, then the output will change thousands of times within the clock pulse.
- This race-around condition can be avoided if tp< t < T. Due to the small propagation delay of the ICs it may be difficult to satisfy the above condition.
- A more practical way to avoid the problem is to use the master-slave (M-S) configuration as discussed below.

Applications of flip-flops:

Frequency Division: When a pulse waveform is applied to the clock input of a J-K flip-flop that is connected to toggle, the Q output is a square wave with half the frequency of the clock input. If more flip-flops are connected together as shown in the figure below, further division of the clock frequency can be achieved

. **Parallel data storage:** a group of flip-flops is called register. To store data of N bits, N flip-flops are required. Since the data is available in parallel form. When a clock pulse is applied to all flip-flops simultaneously, these bits will transfer will be transferred to the Q outputs of the flip flops.

Serial data storage: to store data of N bits available in serial form, N number of D-flip-flops is connected in cascade. The clock signal is connected to all the flip-flops. The serial data is applied to the D input terminal of the first flip-flop.

Transfer of data: data stored in flip-flops may be transferred out in a serial fashion, i.e., bit-by-bit from the output of one flip-flops or may be transferred out in parallel form.

Excitation Tables:

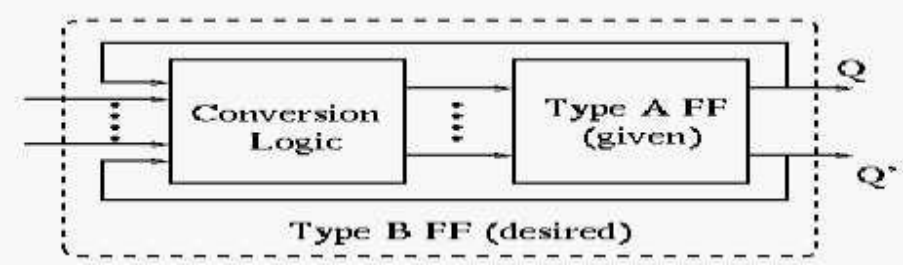
Previous State -> Present State	D
0 -> 0	0
0 -> 1	1
1 -> 0	0
1 -> 1	1

Previous State -> Present State	J	K
0 -> 0	0	X
0 -> 1	1	X
1 -> 0	X	1
1 -> 1	X	0

Previous State -> Present State	S	R
0 -> 0	0	X
0 -> 1	1	0
1 -> 0	0	1
1 -> 1	X	0

Previous State -> Present State	T
0 -> 0	0
0 -> 1	1
1 -> 0	1
1 -> 1	0

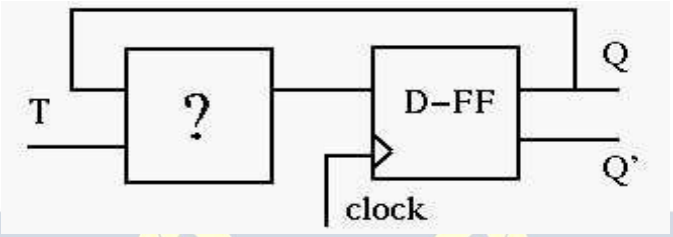
Conversions of flip-flops:



The key here is to use the excitation table, which shows the necessary triggering signal (S,R,J,K, D and T) for a desired flip-flop state transition :

Q_t	Q_{t+1}	S	R	J	K	D	T
0	0	0	x	0	x	0	0
0	1	1	0	1	x	1	1
1	0	0	1	x	1	0	1
1	1	x	0	x	0	1	0

Convert a D-FF to a T-FF:

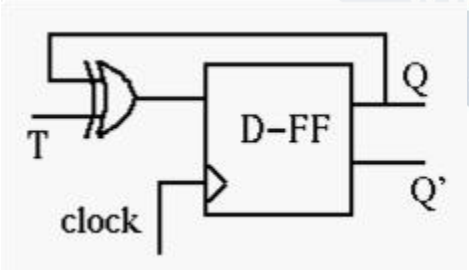


We need to design the circuit to generate the triggering signal D as a function of T and Q: .
Consider the excitation table:

$$D = f(T, Q).$$

Q_t	Q_{t+1}	T	D
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	1

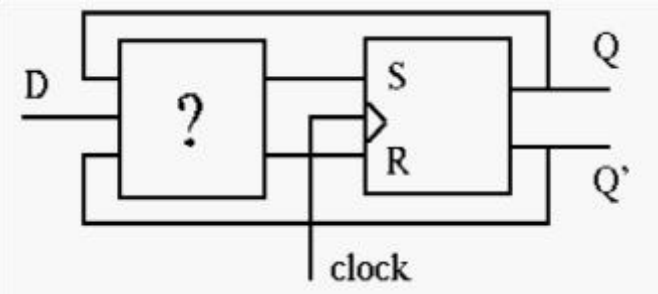
Treating as a function of and current FF state, we have



$$D = T'Q + TQ' = T \oplus Q$$

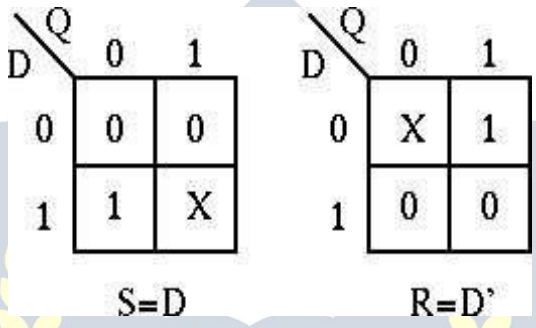
Convert a RS-FF to a D-FF:

We need to design the circuit to generate the triggering signals S and R as functions of and consider the excitation table:

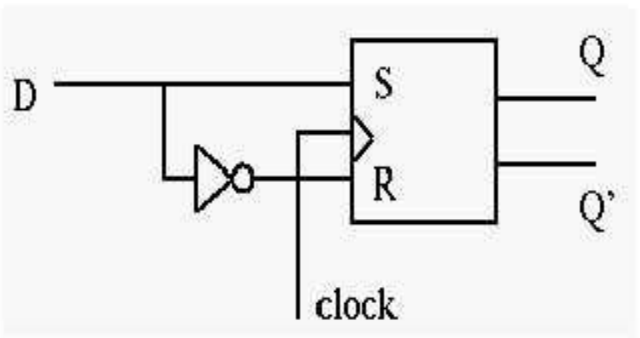


Q_t	Q_{t+1}	D	S	R
0	0	0	0	x
0	1	1	1	0
1	0	0	0	1
1	1	1	x	0

The desired signal and can be obtained as functions of and current FF state from the Karnaugh maps:



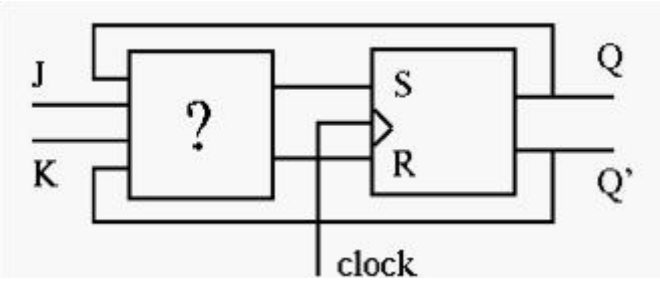
$S = D, \quad R = D'$



Convert a RS-FF to a JK-FF:

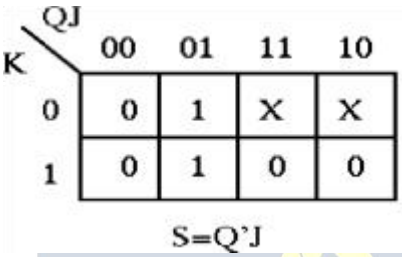
We need to design the circuit to generate the triggering signals S and R as functions of, J, K.

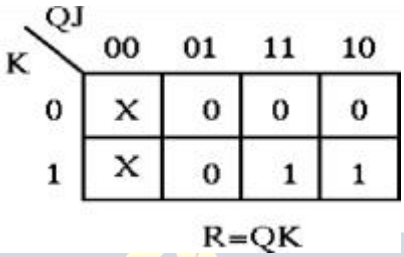
Consider the excitation table**:** The desired signal and as functions of, and current FF state can be obtained from the Karnaugh maps:



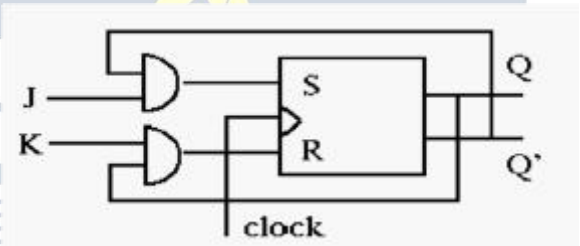
Q_t	Q_{t+1}	J	K	S	R
0	0	0	x	0	x
0	1	1	x	1	0
1	0	x	1	0	1
1	1	x	0	x	0

K-maps:





$S = Q'J, \quad R = QK$



The Master-Slave JK Flip-flop:

The Master-Slave Flip-Flop is basically two gated SR flip-flops connected together in a series configuration with the slave having an inverted clock pulse. The outputs from Q and Q from the "Slave" flip-flop are fed back to the inputs of the "Master" with the outputs of the "Master" flip-flop being connected to the two inputs of the "Slave" flip-flop. This feedback configuration from the slave's output to the master's input gives the characteristic toggle of the JK flip-flop as shown below.

The input signals J and K are connected to the gated "master" SR flip-flop which "locks" the input condition while the clock (Clk) input is "HIGH" at logic level "1". As the clock input of the "slave" flip-flop is the inverse (complement) of the "master" clock input, the "slave" SR flip-flop does not toggle. The outputs from the "master" flip-flop are only "seen" by the gated "slave" flip-flop when the clock input goes "LOW" to logic level "0". When the clock is "LOW", the outputs from the "master" flip-flop are latched and any additional changes to its inputs are ignored. The gated "slave" flip-flop now responds to the state of its inputs passed over by the "master" section. Then on the "Low-to-High" transition of the clock pulse the inputs of the "master" flipflop are fed through to the gated inputs of the "slave" flip-flop and on the "High-to- Low" transition the same inputs are reflected on the output of the "slave" making this type of flip-flop edge or pulse-triggered. Then, the circuit accepts input data when the clock signal is "HIGH", and passes the data to the output on the falling-edge of the clock signal. In other words, the Master-Slave JK Flip-flop is a "Synchronous" device as it only passes data with the timing of the clock signal.

Sequential Circuit Design

- Steps in the design process for sequentialcircuits
- State Diagrams and State Tables Examples

- Steps in Design of a Sequential Circuit

1. Specification – A description of the sequential circuit. Should include a detailing of the inputs, the outputs, and the operation. Possibly assumes that you have knowledge of digital system basics.
2. Formulation: Generate a state diagram and/or a state table from the statement of the problem.
3. State Assignment: From a state table assign binary codes to the states.
4. Flip-flop Input Equation Generation: Select the type of flip-flop for the circuit and generate the needed input for the required statetransitions
5. Output Equation Generation: Derive output logic equations for generation of the output from the inputs and current state.
6. Optimization: Optimize the input and output equations. Today, CAD systems are typically used for this in real systems.
7. Technology Mapping: Generate a logic diagram of the circuit using ANDs, ORs, Inverters, and F/Fs.
8. Verification: Use a HDL to verify the design.

Shift registers:

In digital circuits, a **shift register** is a cascade of flip-flops sharing the same clock, in which the output of each flip-flop is connected to the "data" input of the next flip-flop in the chain, resulting in a circuit that shifts by one position the "bit array" stored in it, *shifting in* the data present at its input and *shifting out* the last bit in the array, at each transition of the clock input. More generally, a **shift register** may be multidimensional, such that its "data in" and stage outputs are themselves bit arrays: this is implemented simply by running several shift registers of the same bitlength in parallel.

Shift registers can have both parallel and serial inputs and outputs. These are often configured as **serial-in, parallel-out** (SIPO) or as **parallel-in, serial-out** (PISO). There are also types that have both serial and parallel input and types with serial and parallel output. There are also **bi- directional** shift registers which allow shifting in both directions: L→R or R→L. The serial input and last output of a shift register can also be connected to create a **circular shift register**

Shift registers are a type of logic circuits closely related to counters. They are basically for the storage and transfer of digital data.

Buffer register:

The buffer register is the simple set of registers. It is simply stores the binary word. The buffer may be controlled buffer. Most of the buffer registers used D Flip-flops.

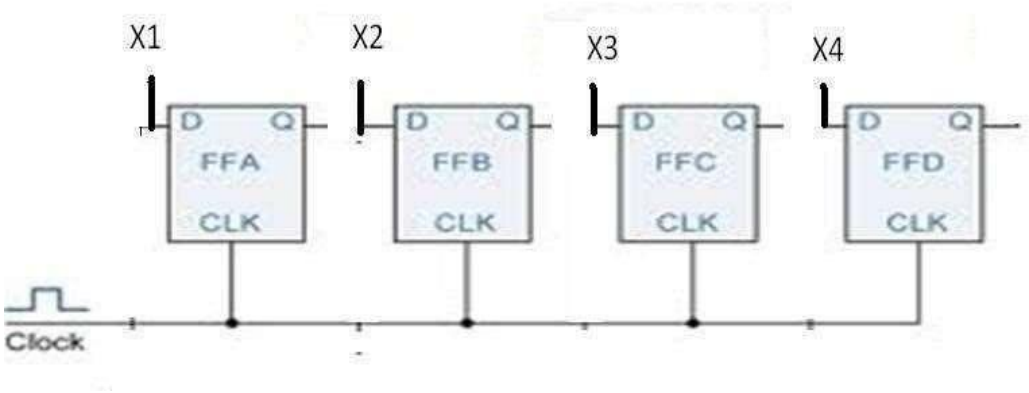


Figure: logic diagram of 4-bit buffer register

The figure shows a 4-bit buffer register. The binary word to be stored is applied to the data terminals. On the application of clock pulse, the output word becomes the same as the word applied at the terminals. i.e., the input word is loaded into the register by the application of clock pulse. When the positive clock edge arrives, the stored word becomes:

$$Q_4Q_3Q_2Q_1=X_4X_3X_2X_1$$

$$Q=X$$

Controlled buffer register:

If goes LOW, all the FFs are RESET and the output becomes, Q=0000.

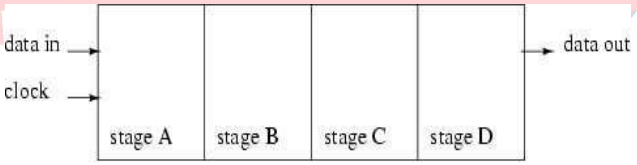
When CLR is HIGH, the register is ready for action. LOAD is the control input. When LOAD is HIGH, the data bits X can reach the D inputs of FF‘s.

$$Q4Q3Q2Q1=X4X3X2X1$$

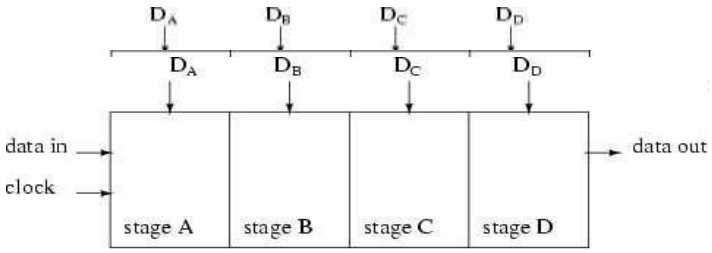
$$Q=X$$

When load is low, the X bits cannot reach the FF‘s.

Data transmission in shift registers:



Serial-in, serial-out shift register with 4-stages



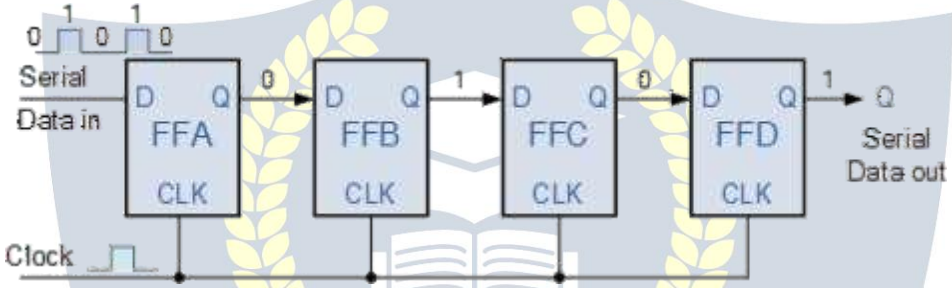
Parallel-in, serial-out shift register with 4-stages

A number of ff's connected together such that data may be shifted into and shifted out of them is called shift register. data may be shifted into or out of the register in serial form or in parallel form. There are four basic types of shift registers.

1. Serial in, serial out, shift right, shift registers
2. Serial in, serial out, shift left, shift registers
3. Parallel in, serial out shift registers
4. Parallel in, parallel out shift registers

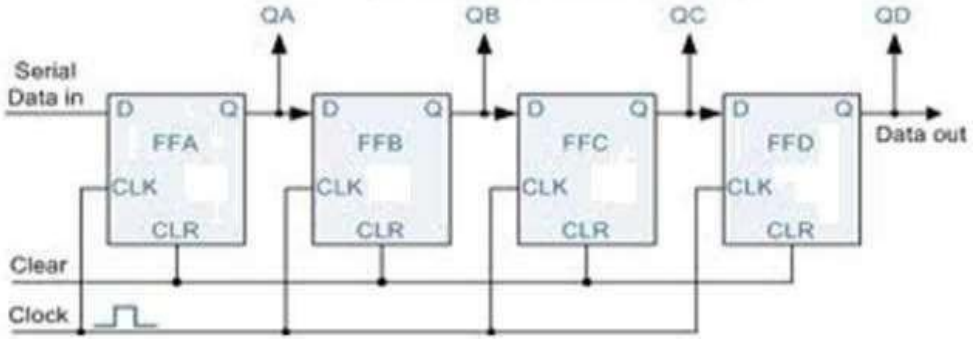
Serial IN, serial OUT, shift right, shift left register:

The logic diagram of 4-bit serial in serial out, right shift register with four stages. The register can store four bits of data. Serial data is applied at the input D of the first FF. the Q output of the first FF is connected to the D input of another FF. the data is outputted from the Q terminal of the last FF.



When serial data is transferred into a register, each new bit is clocked into the first FF at the positive going edge of each clock pulse. The bit that was previously stored by the first FF is transferred to the second FF. the bit that was stored by the Second FF is transferred to the third FF.

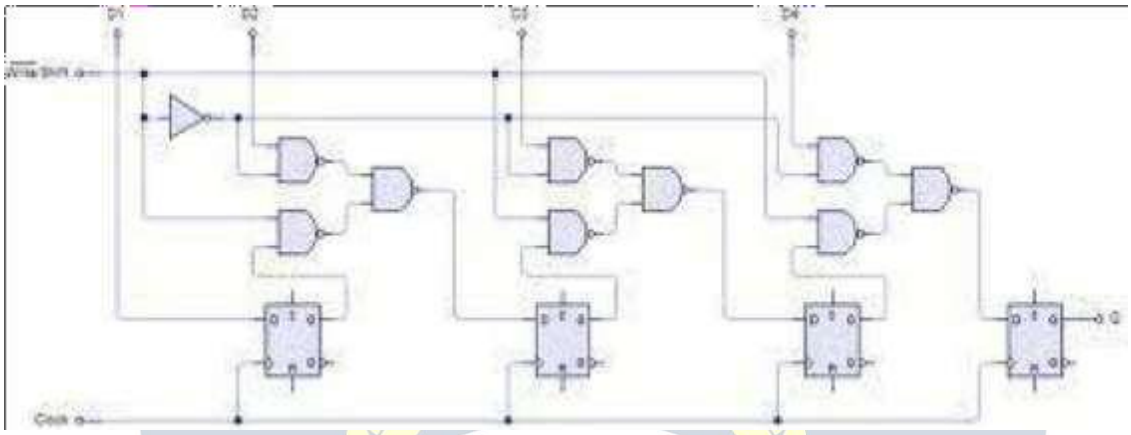
Serial-in, parallel-out, shift register:



In this type of register, the data bits are entered into the register serially, but the data stored in the register is shifted out in parallel form.

Once the data bits are stored, each bit appears on its respective output line and all bits are available simultaneously, rather than on a bit-by-bit basis with the serial output. The serial-in, parallel out, shift register can be used as serial-in, serial out, shift register if the output is taken from the Q terminal of the last FF.

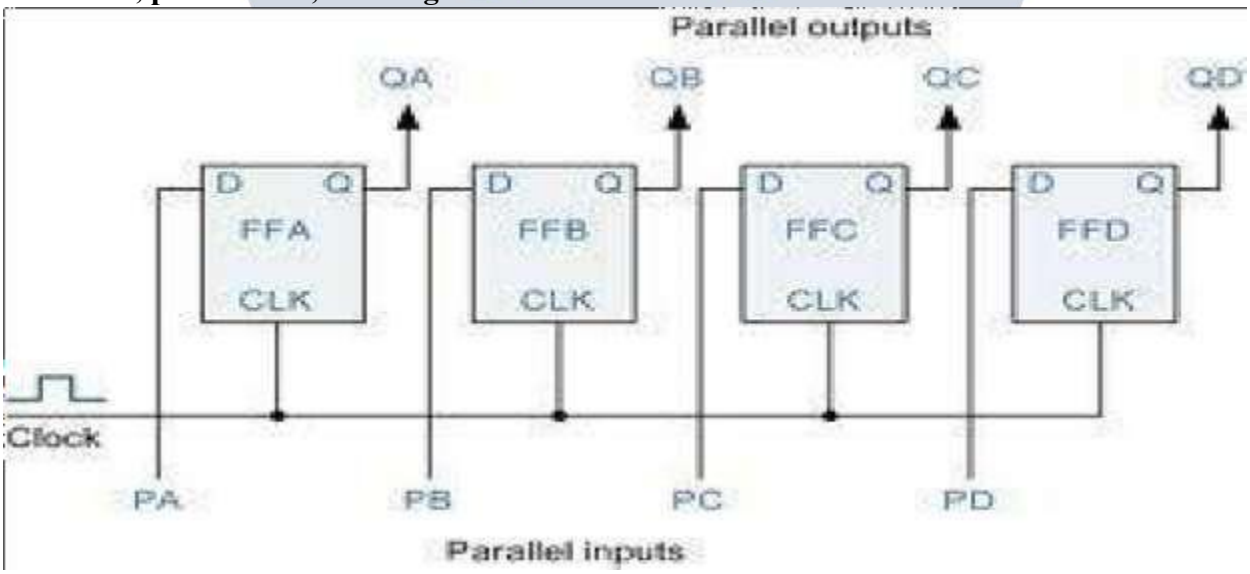
Parallel-in, serial-out, shift register:



For a parallel-in, serial out, shift register, the data bits are entered simultaneously into their respective stages on parallel lines, rather than on a bit-by-bit basis on one line as with serial data bits are transferred out of the register serially. On a bit-by-bit basis over a single line.

There are four data lines A,B,C,D through which the data is entered into the register in parallel form. The signal shift/ load allows the data to be entered in parallel form into the register and the data is shifted out serially from terminalQ4

Parallel-in, parallel-out, shift register



In a parallel-in, parallel-out shift register, the data is entered into the register in parallel form, and also the data is taken out of the register in parallel form. Data is applied to the D input terminals of the FF‘s. When a clock pulse is applied, at the positive going edge of the pulse, the D inputs are

shifted into the Q outputs of the FFs. The register now stores the data. The stored data is available instantaneously for shifting out in parallel form.

Bidirectional shift register:

A bidirectional shift register is one which the data bits can be shifted from left to right or from right to left. A fig shows the logic diagram of a 4-bit serial-in, serial out, bidirectional shift register. Right/left is the mode signal, when right /left is a 1, the logic circuit works as a shiftregister.the bidirectional operation is achieved by using the mode signal and two NAND gates and one OR gate for each stage.

A HIGH on the right/left control input enables the AND gates G1, G2, G3 and G4 and disables the AND gates G5,G6,G7 and G8, and the state of Q output of each FF is passed through the gate to the D input of the following FF. when a clock pulse occurs, the data bits are then effectively shifted one place to the right. A LOW on the right/left control inputs enables the AND gates G5, G6, G7 and G8 and disables the And gates G1, G2, G3 and G4 and the Q output of each FF is passed to the D input of the preceding FF. when a clock pulse occurs, the data bits are then effectively shifted one place to the left. Hence, the circuit works as a bidirectional shift register

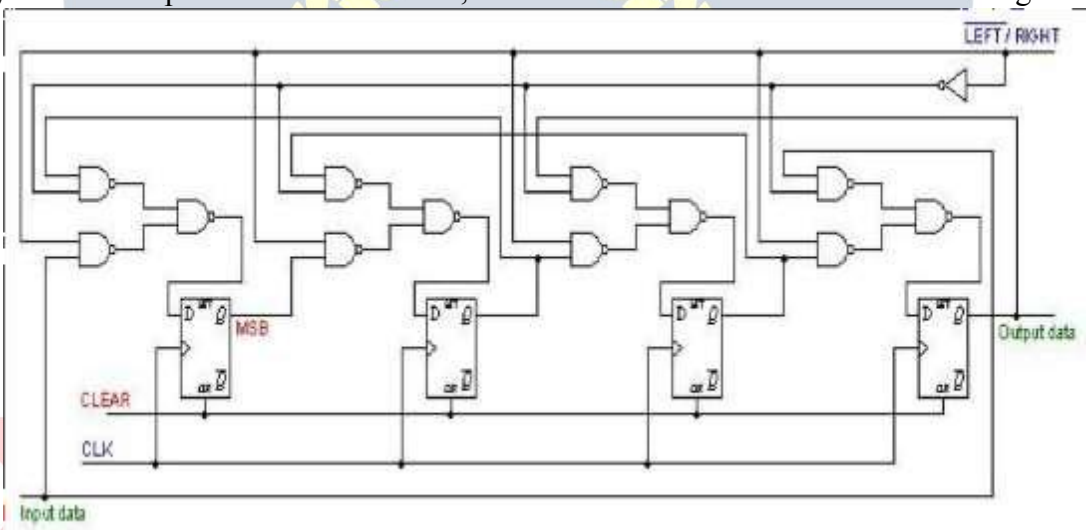


Figure: logic diagram of a 4-bit bidirectional shift register

Universal shift register:

A register is capable of shifting in one direction only is a unidirectional shift register. One that can shift both directions is a bidirectional shift register. If the register has both shifts and parallel load capabilities, it is referred to as a universal shift registers. Universal shift register is a bidirectional register, whose input can be either in serial form or in parallel form and whose output also can be in serial form or I parallel form.

The most general shift register has the following capabilities.

1. A clear control to clear the register to 0
2. A clock input to synchronize theoperations

3. A shift-right control to enable the shift-right operation and serial input and output lines associated with the shift-right. A shift-left control to enable the shift-left operation and serial input and output lines associated with theshift-left
4. A parallel loads control to enable a parallel transfer and the n input lines associated with the parallel transfer
5. N parallel output lines
6. A control state that leaves the information in the register unchanged in the presence of the clock.

A universal shift register can be realized using multiplexers. The below fig shows the logic diagram of a 4-bit universal shift register that has all capabilities. It consists of 4 D flip-flops and four multiplexers. The four multiplexers have two common selection inputs s1 and s0. Input 0 in each multiplexer is selected when S1S0=00, input 1 is selected when S1S0=01 and input 2 is selected when S1S0=10 and input 4 is selected when S1S0=11. The selection inputs control the mode of operation of the register according to the functions entries. When S1S0=0, the present value of the register is applied to the D inputs of flip-flops. The condition forms a path from the output of each flip-flop into the input of the same flip-flop. The next clock edge transfers into each flip-flop the binary value it held previously, and no change of state occurs. When S1S0=01, terminal 1 of the multiplexer inputs have a path to the D inputs of the flip-flop. This causes a shiftright operation, with serial input transferred into flip-flopA4. When S1S0=10, a shift left operation results with the other serial input going into flip-flop A1. Finally when S1S0=11, the binary information on the parallel input lines is transferred into the register simultaneously during the next clock cycle

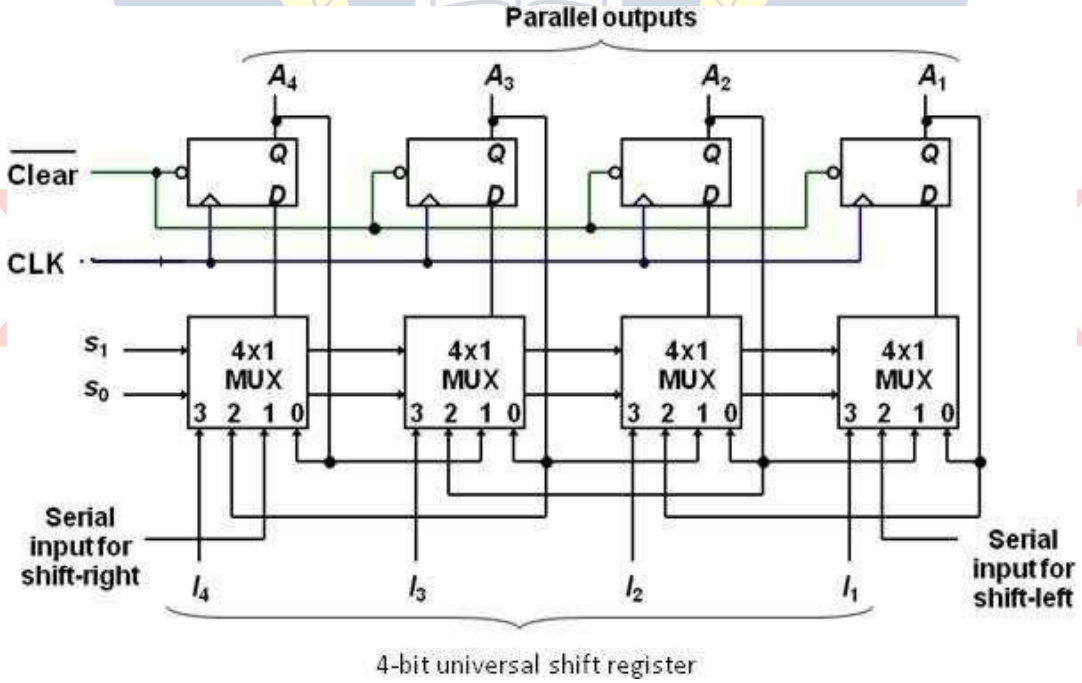


Figure: logic diagram 4-bit universal shift register

Function table for theregister

mode control

S0	S1	register operation
0	0	No change
0	1	Shift Right
1	0	Shift left
1	1	Parallel load

Counters:

Counter is a device which stores (and sometimes displays) the number of times particular event or process has occurred, often in relationship to a clock signal. A Digital counter is a set of flip flops whose state change in response to pulses applied at the input to the counter. Counters may be asynchronous counters or synchronous counters. Asynchronous counters are also called ripple counters

In electronics counters can be implemented quite easily using register-type circuits such as the flip-flops and a wide variety of classifications exist:

- Asynchronous (ripple) counter – changing state bits are used as clocks to subsequent state flip-flops
- Synchronous counter – all state bits change under control of a single clock
- Decade counter – counts through ten states per stage
- Up/down counter – counts both up and down, under command of a control input
- Ring counter – formed by a shift register with feedback connection in a ring
    - Johnson counter – a *twisted* ring counter Cascaded counter Modulus counter.

Each is useful for different applications. Usually, counter circuits are digital in nature, and count in natural binary Many types of counter circuits are available as digital building blocks, for example a number of chips in the 4000 series implement different counters.

Occasionally there are advantages to using a counting sequence other than the natural binary sequence such as the binary coded decimal counter, a linear feed-back shift register counter, or a gray-code counter.

Counters are useful for digital clocks and timers, and in oven timers, VCR clocks, etc.

Asynchronous counters:

An asynchronous (ripple) counter is a single JK-type flip-flop, with its J (data) input fed from its own inverted output. This circuit can store one bit, and hence can count from zero to one before it overflows (starts over from 0). This counter will increment once for every clock cycle and takes two clock cycles to overflow, so every cycle it will alternate between a transition from 0 to 1 and a transition from 1 to 0. Notice that this creates a new clock with a 50% duty cycle at exactly half the frequency of the input clock. If this output is then used as the clock signal for a similarly

arranged D flip-flop (remembering to invert the output to the input), one will get another 1 bit counter that counts half as fast. Putting them together yields a two-bit counter:

Two-bit ripple up-counter using negative edge triggered flip flop:

Two bit ripple counter used two flip-flops. There are four possible states from 2 – bit up- counting I.e. 00, 01, 10 and 11.

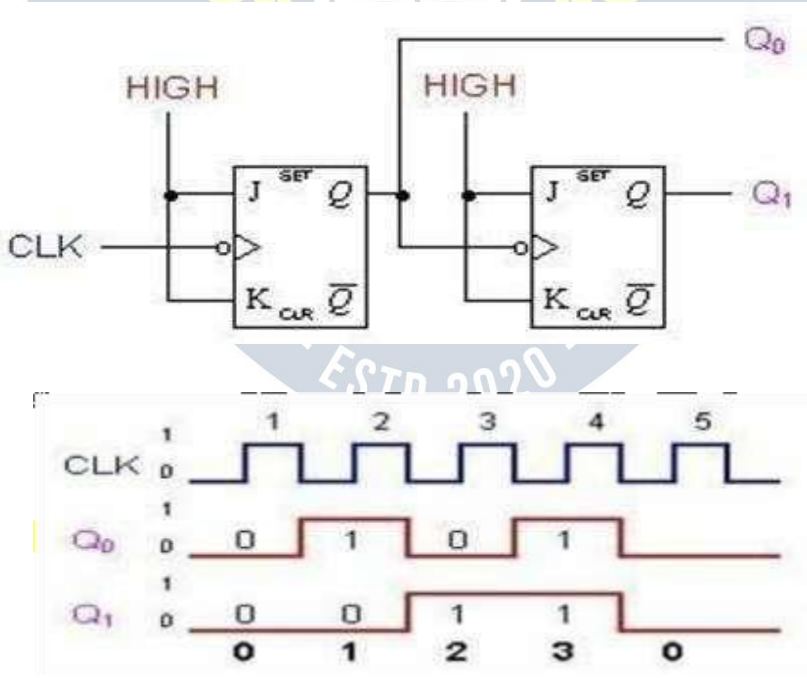
· The counter is initially assumed to be at a state 00 where the outputs of the tow flip-flops are noted as Q_1Q_0. Where Q_1 forms the MSB and Q_0 forms the LSB.

· For the negative edge of the first clock pulse, output of the first flip-flop FF_1 toggles its state. Thus Q_1 remains at 0 and Q_0 toggles to 1 and the counter state are now read as 01.

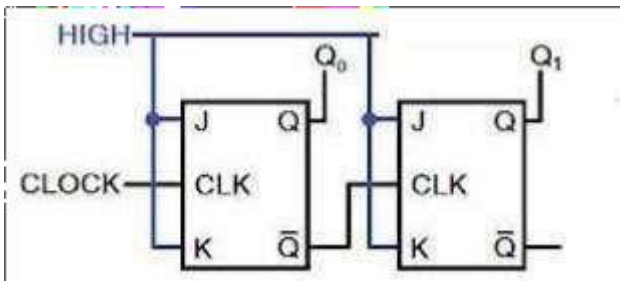
· During the next negative edge of the input clock pulse FF_1 toggles and $Q_0 = 0$. The output Q0 being a clock signal for the second flip-flop FF_2 and the present transition acts as a negative edge for FF_2 thus toggles its state $Q_1 = 1$. The counter state is now read as 10.

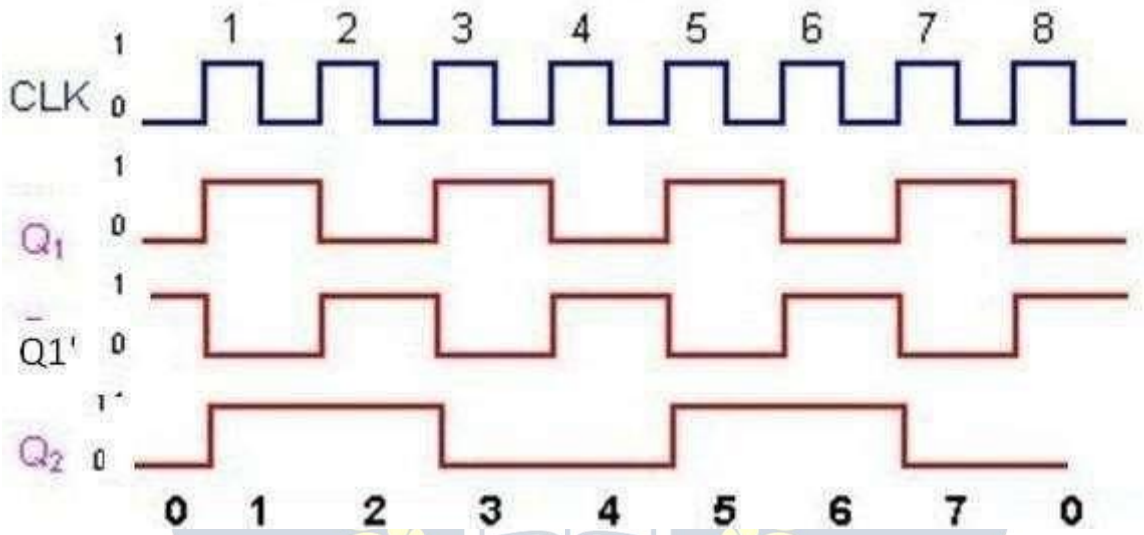
· For the next negative edge of the input clock to FF_1 output Q0 toggles to 1. But this transition from 0 to 1 being a positive edge for FF_2 output Q_1 remains at 1. The counter state is now read as 11.

· For the next negative edge of the input clock, Q_0 toggles to 0. This transition from 1 to 0 acts as a negative edge clock for FF_2 and its output Q_1 toggles to 0. Thus the starting state 00 is attained. Figure shown below



Two-bit ripple down-counter using negative edge triggered flip flop:





A 2-bit down-counter counts in the order 0,3,2,1,0,1…….,i.e, 00,11,10,01,00,11 …..,etc. the above fig. shows ripple down counter, using negative edge triggered J-K FFs and its timing diagram.

- For down counting, Q1‘ of FF1 is connected to the clock of Ff2. Let initially all the FF1 toggles, so, Q1 goes from a 0 to a 1 and Q1‘ goes from a 1 to a 0.
- The negative-going signal at Q1‘ is applied to the clock input of FF2, toggles Ff2 and, therefore, Q2 goes from a 0 to a 1.so, after one clock pulse Q2=1 and Q1=1, I.e., the state of the counter is 11.
- At the negative-going edge of the second clock pulse, Q1 changes from a 1 to a 0 and Q1‘from a 0 to a 1.
- This positive-going signal at Q1‘ does not affect FF2 and, therefore, Q2 remains at a 1. Hence , the state of the counter after second clock pulse is 10
- At the negative going edge of the third clock pulse, FF1 toggles. So Q1, goes from a 0 to a 1 and Q1‘ from 1 to 0. This negative going signal at Q1‘ toggles FF2 and, so, Q2 changes from 1 to 0, hence, the state of the counter after the third clock pulse is01.
- At the negative going edge of the fourth clock pulse, FF1 toggles. So Q1, goes from a 1 to a 0 and Q1‘ from 0 to 1. . This positive going signal at Q1‘ does not affect FF2 and, so, Q2 remains at 0, hence, the state of the counter after the fourth clock pulse is 00.

Two-bit ripple up-down counter using negative edge triggered flip flop:

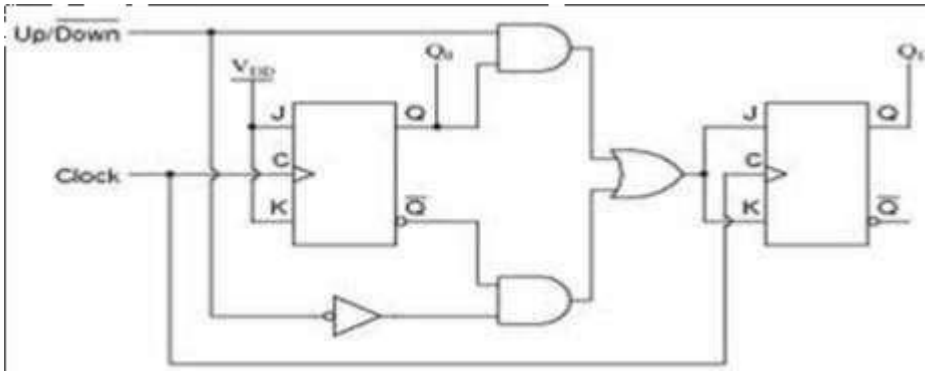


Figure: asynchronous 2-bit ripple up-down counter using negative edge triggered flip flop

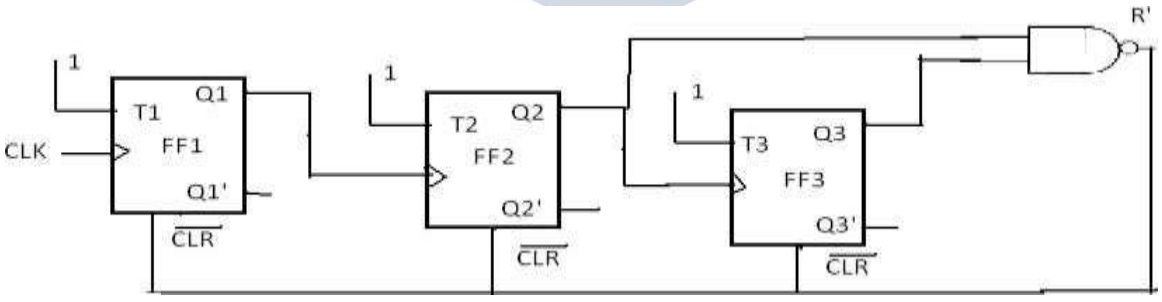
- As the name indicates an up-down counter is a counter which can count both in upward and downward directions. An up-down counter is also called a forward/backward counter or a bidirectional counter. So, a control signal or a mode signal M is required to choose the direction of count. When M=1 for up counting, Q1 is transmitted to clock of FF2 and when M=0 for down counting, Q1‘ is transmitted to clock of FF2. This is achieved by using two AND gates and one OR gates. The external clock signal is applied toFF1.
- Clock signal to FF2= (Q1.Up)+(Q1‘. Down)= Q1m+Q1‘M‘

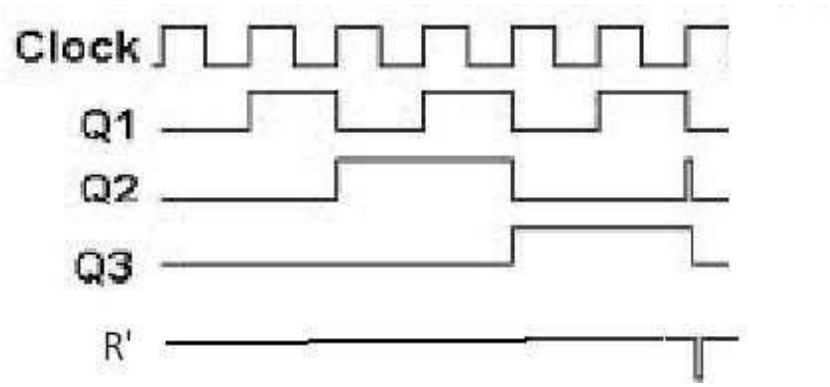
Design of Asynchronous counters:

To design a asynchronous counter, first we write the sequence , then tabulate the values of reset signal R for various states of the counter and obtain the minimal expression for R and R‘ using K-Map or any other method. Provide a feedback such that R and R‘ resets all the FF‘s after the desired count

Design of a Mod-6 asynchronous counter using T FFs:

A mod-6 counter has six stable states 000, 001, 010, 011, 100, and 101. When the sixth clock pulse is applied, the counter temporarily goes to 110 state, but immediately resets to 000 because of the feedback provided. it is ―divide by-6-counter‖, in the sense that it divides the input clock frequency by 6.it requires three FFs, because the smallest value of n satisfying the condition$N \leq 2^n$ is n=3; three FFs can have 8 possible states, out of which only six are utilized and the remaining two states 110and 111, are invalid. If initially the counter is in 000 state, then after the sixth clock pulse, it goes to 001, after the second clock pulse, it goes to 010, and so on.





After sixth clock pulse it goes to 000. For the design, write the truth table with present state outputs Q3, Q2 and Q1 as the variables, and reset R as the output and obtain an expression for R in terms of Q3, Q2, and Q1that decides the feedback into be provided. From the truth table, R=Q3Q2. For active-low Reset, R‘ is used. The reset pulse is of very short duration, of the order of nanoseconds and it is equal to the propagation delay time of the NAND gate used. The expression for R can also be determined as follows.

R=0 for 000 to 101, R=1 for 110, and R=X=for111
Therefore,

R=Q3Q2Q1‘+Q3Q2Q1=Q3Q2

The logic diagram and timing diagram of Mod-6 counter is shown in the above fig.

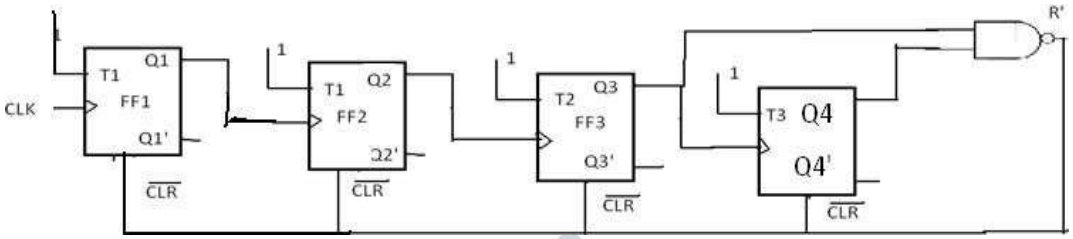
The truth table is as shown in below.

After pulses	States			
	Q3	Q2	Q1	R
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	0
4	1	0	0	0
5	1	0	1	0
6	1	1	0	1
	0	0	0	0
7	0	0	0	0

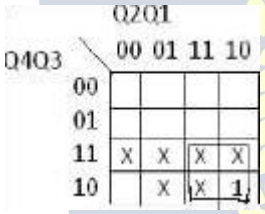
Design of a mod-10 asynchronous counter using T-flip-flops:

A mod-10 counter is a decade counter. It also called a BCD counter or a divide-by-10 counter. It requires four flip-flops (condition $10 \leq 2^n$ is n=4). So, there are 16 possible states, out of which ten are valid and remaining six are invalid. The counter has ten stable state, 0000 through

1001, i.e., it counts from 0 to 9. The initial state is 0000 and after nine clock pulses it goes to 1001. When the tenth clock pulse is applied, the counter goes to state 1010 temporarily, but because of the feedback provided, it resets to initial state 0000. So, there will be a glitch in the waveform of Q2. The state 1010 is a temporary state for which the reset signal R=1, R=0 for 0000 to 1001, and R=C for 1011 to 1111.



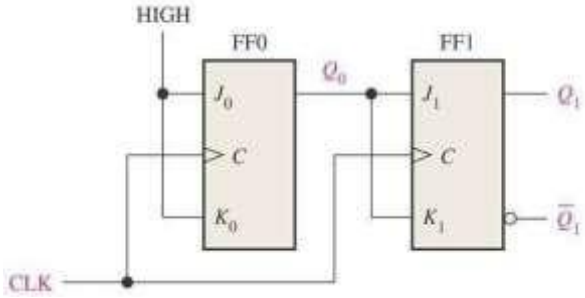
The count table and the K-Map for reset are shown in fig. from the K-Map R=Q4Q2. So, feedback is provided from second and fourth FFs. For active –HIGH reset, Q4Q2 is applied to the clear terminal. For active-LOW reset 4 2 is connected isof all Flip=flops.

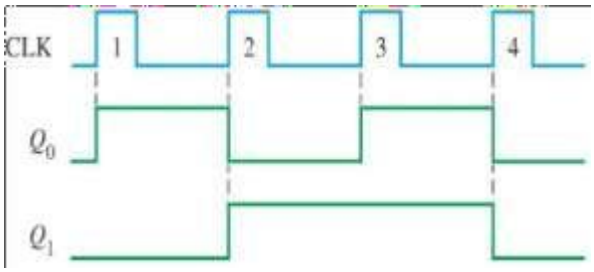


After pulses	Count Q4	Q3	Q2	Q1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	0	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	0	1	0	1
10	0	0	0	0

Synchronous counters:

Asynchronous counters are serial counters. They are slow because each FF can change state only if all the preceding FFs have changed their state. if the clock frequency is very high, the asynchronous counter may skip some of the states. This problem is overcome in synchronous counters or parallel counters. Synchronous counters are counters in which all the flip flops are triggered simultaneously by the clock pulses Synchronous counters have a common clock pul se applied simultaneously to all flip-flops. A 2-Bit Synchronous Binary Counter





Design of synchronous counters:

For a systematic design of synchronous counters. The following procedure is used.

Step 1: State Diagram: draw the state diagram showing all the possible states state diagram which also be called nth transition diagrams, is a graphical means of depicting the sequence of states through which the counter progresses.

Step2: number of flip-flops: based on the description of the problem, determine the required number n of the flip-flops- the smallest value of n is such that the number of states $N \leq 2^n$--- and the desired counting sequence.

Step3: choice of flip-flops excitation table: select the type of flip-flop to be used and write the excitation table. An excitation table is a table that lists the present state (ps) , the next state(ns) and required excitations.
Step4: minimal expressions for excitations: obtain the minimal expressions for the excitations of the FF using K-maps drawn for the excitation of the flip-flops in terms of the present states and inputs.

Step5: logic diagram: draw a logic diagram based on the minimal expressions

Design of a synchronous 3-bit up-down counter using JK flip-flops:

Step1: determine the number of flip-flops required. A 3-bit counter requires three FFs. It has 8 states (000,001,010,011,101,110,111) and all the states are valid. Hence no don‘t cares. For selecting up and down modes, a control or mode signal M is required. When the mode signal M=1 and counts down when M=0. The clock signal is applied to all the FFs simultaneously.

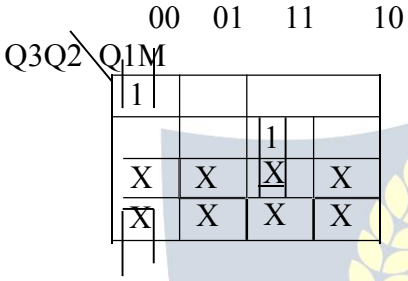
Step2: draw the state diagrams: the state diagram of the 3-bit up-down counter is drawn as

Step3: select the type of flip flop and draw the excitation table: JK flip-flops are selected and the excitation table of a 3-bit up-down counter using JK flip-flops is drawn as shown in fig.

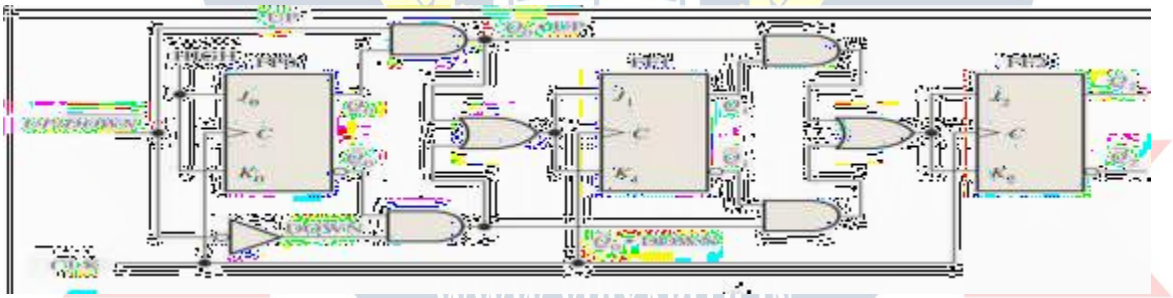
PS			mode	NS			required excitations					
Q3	Q2	Q1	M	Q3	Q2	Q1	J3	K3	J2	K2	J1	K1
0	0	0	0	1	1	1	1	x	1	x	1	x
0	0	0	1	0	0	1	0	x	0	x	1	x
0	0	1	0	0	0	0	0	x	0	x	x	1
0	0	1	1	0	1	0	0	x	1	x	x	1
0	1	0	0	0	0	1	0	x	x	1	1	x
0	1	0	1	0	1	1	0	x	x	0	1	x
0	1	1	0	0	1	0	0	x	x	0	x	1
0	1	1	1	1	0	0	1	x	x	1	x	1
1	0	0	0	0	1	1	x	1	1	x	1	x
1	0	0	1	1	0	1	x	0	0	x	1	x

1	0	1	0	1	0	0	x	0	0	x	x	1
1	0	1	1	1	1	0	x	0	1	x	x	1
1	1	0	0	1	0	1	x	0	x	1	1	x
1	1	0	1	1	1	1	x	0	x	0	1	x
1	1	1	0	1	1	0	x	0	x	0	x	1
1	1	1	1	0	0	0	x	1	x	1	x	1

Step4: obtain the minimal expressions: From the excitation table we can conclude that J1=1 and K1=1, because all the entries for J1and K1 are either X or 1. The K-maps for J3, K3,J2 and K2 based on the excitation table and the minimal expression obtained from them are shown in fig.



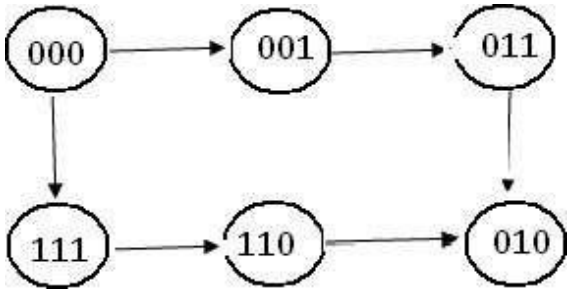
Step5: draw the logic diagram: a logic diagram using those minimal expressions can be drawn as shown in fig.



Design of a synchronous modulo-6 gray cod counter:

Step 1: the number of flip-flops: we know that the counting sequence for a modulo-6 gray code counter is 000, 001, 011, 010, 110, and 111. It requires n=3FFs ($N \leq 2^n$, i.e., $6 \leq 2^3$). 3 FFs can have 8 states. So the remaining two states 101 and 100 are invalid. The entries for excitation corresponding to invalid states are don‘t cares.

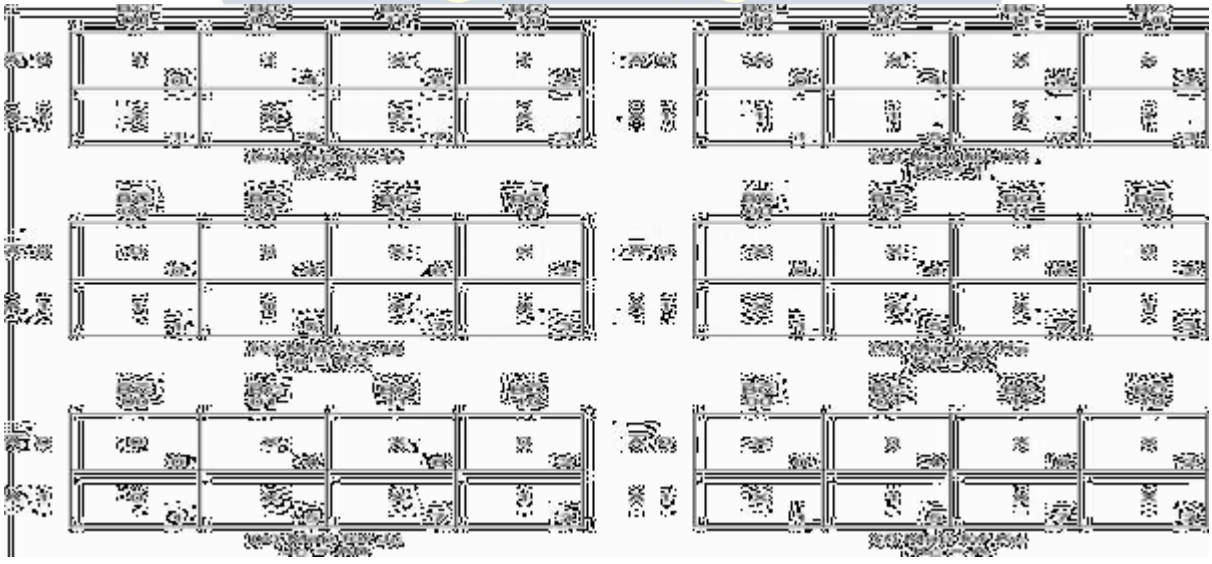
Step2: the state diagram: the state diagram of the mod-6 gray code converter is drawn as shown in fig.



Step3: type of flip-flop and the excitation table: T flip-flops are selected and the excitation table of the mod-6 gray code counter using T-flip-flops is written as shown in fig.

PS			**NS**			**required excitations**		
Q3	**Q2**	**Q1**	**Q3**	**Q2**	**Q1**	**T3**	**T2**	**T1**
0	0	0	0	0	1	0	0	1
0	0	1	0	1	1	0	1	0
0	1	1	0	1	0	0	0	1
0	1	0	1	1	0	1	0	0
1	1	0	1	1	1	0	0	1
1	1	1	0	0	0	1	1	1

Step4: The minimal expressions: the K-maps for excitations of FFs T3,T2,and T1 in terms of outputs of FFs Q3,Q2, and Q1, their minimization and the minimal expressions for excitations obtained from them are shown if fig



Step5: the logic diagram: the logic diagram based on those minimal expressions is drawn as shown in fig.

Design of a synchronous BCD Up-Down counter using FFs:

Step1: the number of flip-flops: a BCD counter is a mod-10 counter has 10 states (0000 through 1001) and so it requires n=4FFs($N \leq 2^n$,, i.e., $10 \leq 2^4$). 4 FFS can have 16 states. So out of 16 states, six states (1010 through 1111) are invalid. For selecting up and down mode, a control or mode signal M is required. , it counts up when M=1 and counts down when M=0. The clock signal is applied to all FFs.

Step2: the state diagram: The state diagram of the mod-10 up-down counter is drawn as shown in fig.

Step3: types of flip-flops and excitation table: T flip-flops are selected and the excitation table of the modulo-10 up down counter using T flip-flops is drawn as shown in fig.

The remaining minterms are don't cares(∑d(20,21,22,23,24,25,26,37,28,29,30,31)) from the excitation table we can see that T1=1 and the expression for T4,T3,T2 are as follows.
T4=∑m(0,15,16,19)+d(20,21,22,23,24,25,26,27,28,29,30,31)
T3=∑m(7,15,16,8)+d(20,21,22,23,24,25,26,27,28,29,30,31)
T2=∑m(3,4,7,8,11,12,15,16)+d(20,21,22,23,24,25,26,27,28,29,30,31)

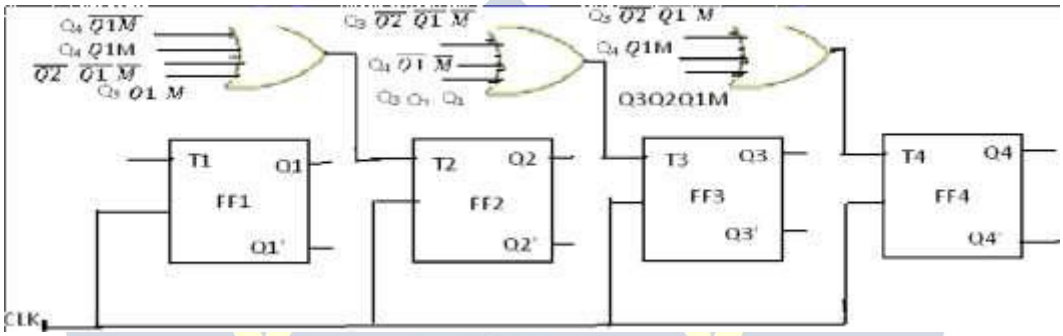
PS				**mode**	**NS**				**required excitations**			
Q4	**Q3**	**Q2**	**Q1**	**M**	**Q4**	**Q3**	**Q2**	**Q1**	**T4**	**T3**	**T2**	**T1**
0	0	0	0	0	1	0	0	1	1	0	0	1
0	0	0	0	1	0	0	0	1	0	0	0	1
0	0	0	1	0	0	0	0	0	0	0	0	1
0	0	0	1	1	0	0	1	0	0	0	1	1
0	0	1	0	0	0	0	0	1	0	0	1	1
0	0	1	0	1	0	0	1	1	0	0	0	1
0	0	1	1	0	0	0	1	0	0	0	0	1
0	0	1	1	1	0	1	0	0	0	1	1	1
0	1	0	0	0	0	0	1	1	0	1	1	1
0	1	0	0	1	0	1	0	1	0	0	0	1
0	1	0	1	0	0	1	0	0	0	0	0	1
0	1	0	1	1	0	1	1	0	0	0	1	1
0	1	1	0	0	0	1	0	1	0	0	1	1
0	1	1	0	1	0	1	1	1	0	0	0	1
0	1	1	1	0	0	1	1	0	0	0	0	1
0	1	1	1	1	1	0	0	0	1	1	1	1

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

Step4: The minimal expression: since there are 4 state variables and a mode signal, we require 5 variable kmaps. 20 conditions of Q4Q3Q2Q1M are valid and the remaining 12 combinations are invalid. So the entries for excitations corresponding to those invalid combinations are don‘t cares. Minimizing K-maps for T2 we get

T 2= Q4Q1‘M+Q4‘Q1M+Q2Q1‘M‘+Q3Q1‘M‘

Step5: the logic diagram: the logic diagram based on the above equation is shown in fig.



Shift register counters:

One of the applications of shift register is that they can be arranged to form several types of counters. The most widely used shift register counter is ring counter as well as the twisted ring counter.

Ring counter: this is the simplest shift register counter. The basic ring counter using D flip- flops is shown in fig. the realization of this counter using JK FFs. The Q output of each stage is connected to the D flip-flop connected back to the ring counter.

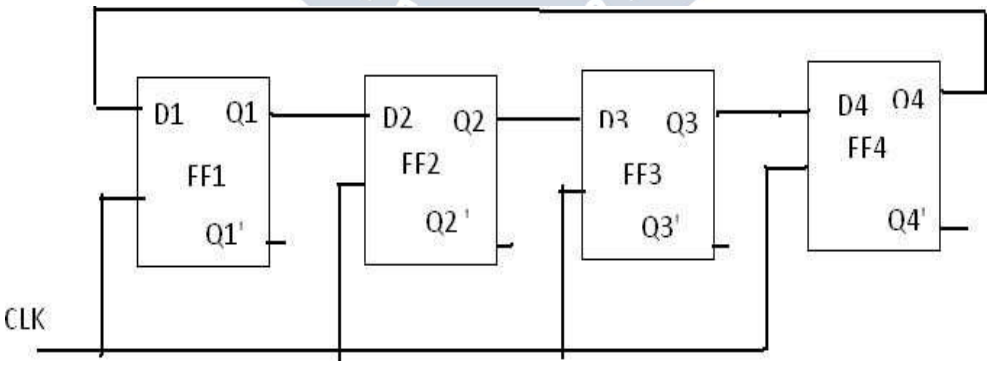
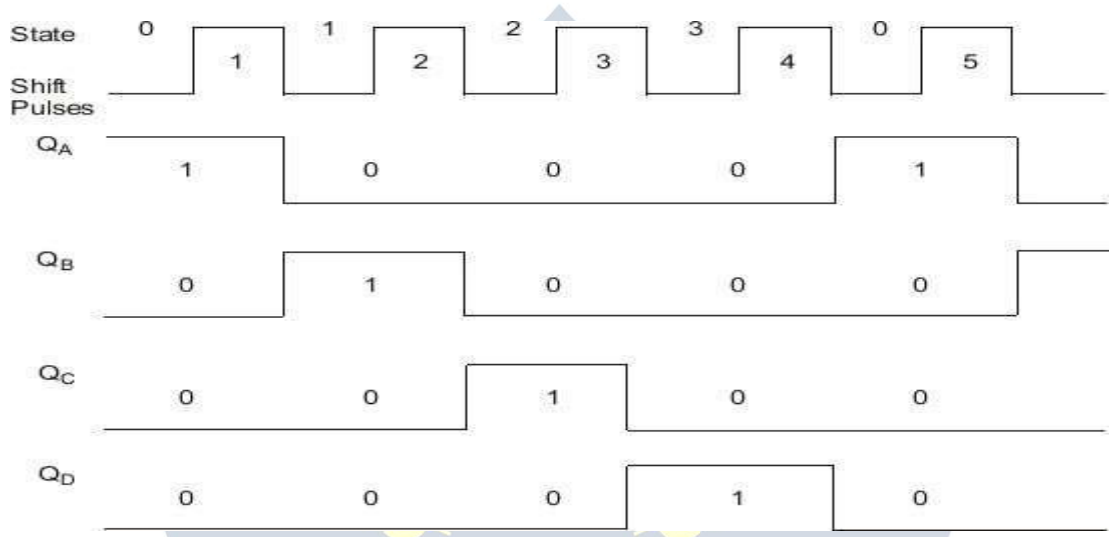


fig: logic diagram of 4-bit ring counter using D flip-flops

Only a single 1 is in the register and is made to circulate around the register as long as clock pulses are applied. Initially the first FF is present to a 1. So, the initial state is 1000, i.e., Q1=1, Q2=0,Q3=0,Q4=0. After each clock pulse, the contents of the register are shifted to the right by one bit and Q4 is shifted back to Q1. The sequence repeats after four clock pulses. The number of distinct states in the ring counter, i.e., the mod of the ring counter is equal to number of FFs used in the counter. An n-bit ring counter can count only n bits, where as n-bit ripple counter can count 2^n bits. So, the ring counter is uneconomical compared to a ripple counter but has advantage of requiring no decoder, since we can read the count by simply noting which FF is set. Since it is entirely a synchronous operation and requires no gates external FFs, it has the further advantage of being very fast. **Timing diagram:**



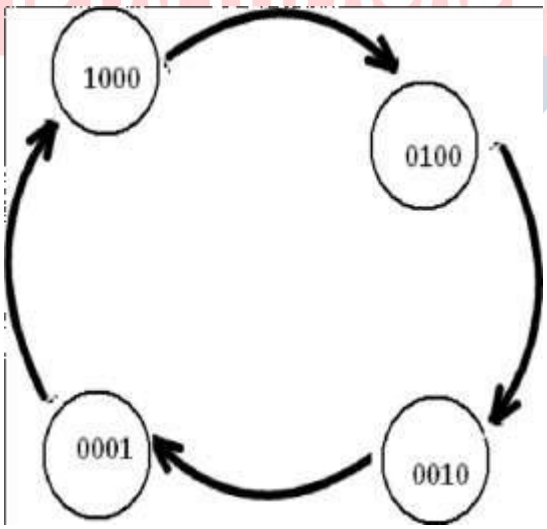


Figure: state diagram

Twisted Ring counter (Johnson counter):

This counter is obtained from a serial-in, serial-out shift register by providing feedback from the inverted output of the last FF to the D input of the first FF. the Q output of each is connected to the D input of the next stage, but the Q‘ output of the last stage is connected to the D input of the first stage, therefore, the name twisted ring counter. This feedback arrangement produces a unique sequence of states.

The logic diagram of a 4-bit Johnson counter using D FF is shown in fig. the realization of the same using J-K FFs is shown in fig.. The state diagram and the sequence table are shown in figure. The timing diagram of a Johnson counter is shown infigure.

Let initially all the FFs be reset, i.e., the state of the counter be 0000. After each clock pulse, the level of Q1 is shifted to Q2, the level of Q2to Q3, Q3 to Q4 and the level of Q4‘to Q1 and the sequences given in fig.

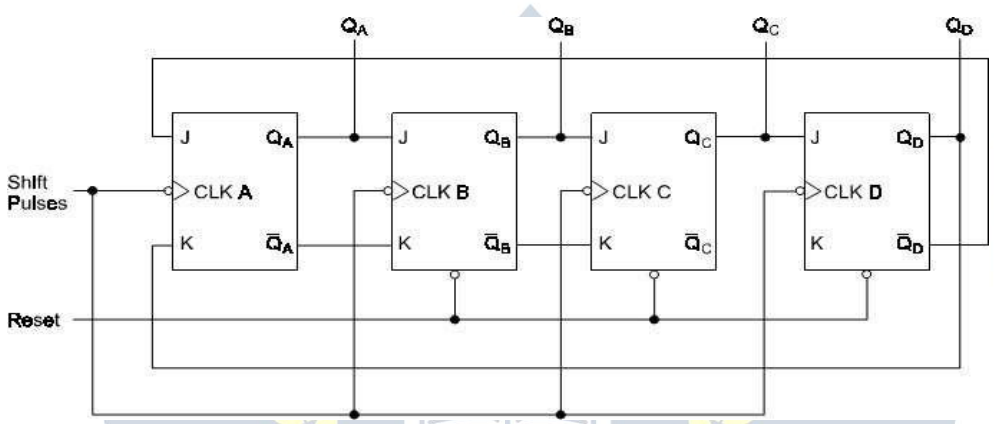


Figure: Johnson counter with JK flip-flops

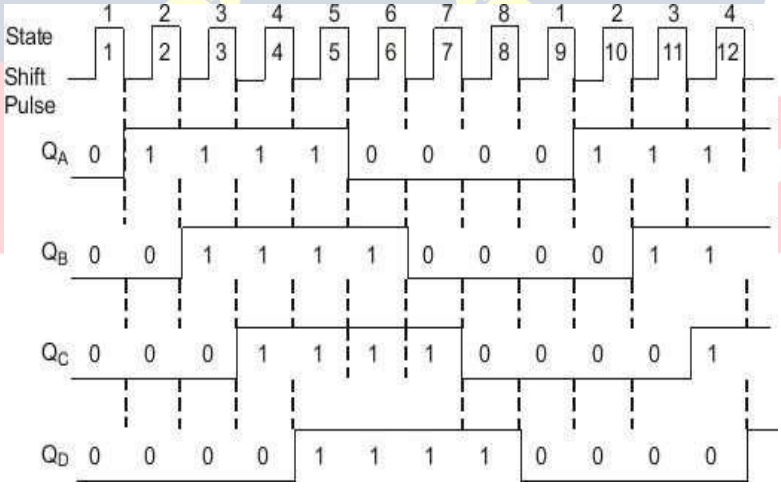
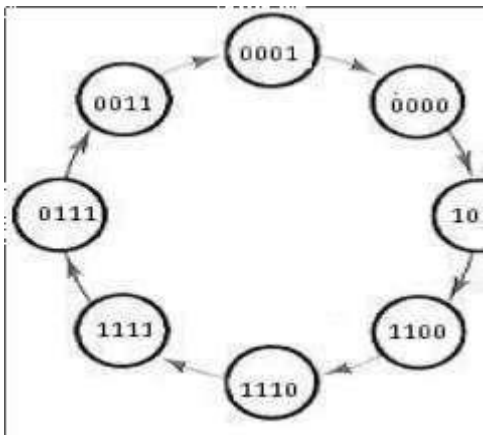


Figure: timing diagram

State diagram:

	Q1	Q2	Q3	Q4	after clock pulse
0	0	0	0	0	0



1	0	0	0	1
1	1	0	0	2
1	1	1	0	3
1	1	1	1	4
0	1	1	1	5
0	0	1	1	6
0	0	0	1	7
0	0	0	0	8
1	0	0	0	9

Excitation table

Synthesis of sequential circuits:

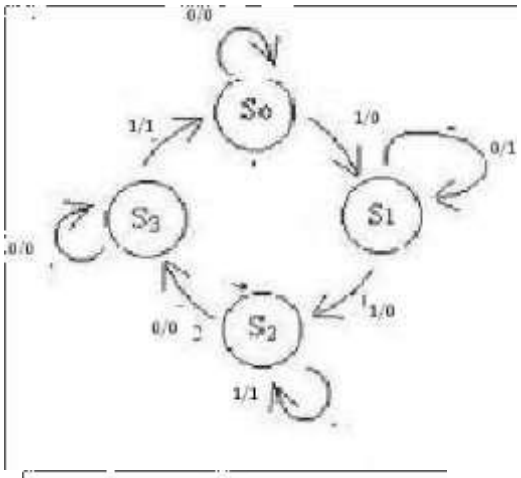
The synchronous or clocked sequential circuits are represented by two models.

1. Moore circuit: in this model, the output depends only on the present state of the flip-flops
2. Meelay circuit: in this model, the output depends on both present state of the flip- flop. And the inputs.

Sequential circuits are also called finite state machines (FSMs). This name is due to the fast that the functional behavior of these circuits can be represented using a finite number of states.

State diagram: the state diagram or state graph is a pictorial representation of the relationships between the present state, the input, the next state, and the output of a sequential circuit. The state diagram is a pictorial representation of the behavior of a sequential circuit.

The state represented by a circle also called the node or vertex and the transition between states is indicated by directed lines connecting circle. a directed line connecting a circle with itself indicates that the next state is the same as the present state. The binary number inside each circle identifies the state represented by the circle. The direct lines are labeled with two binary numbers separated by a symbol. The input value is applied during the present state is labeled after the symbol.



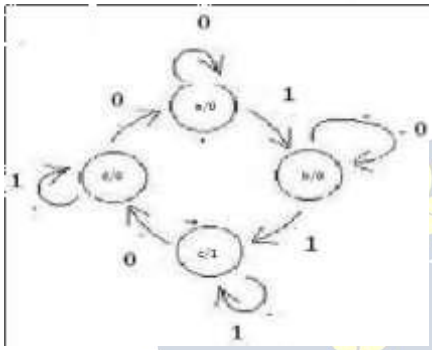
NS,O/P

INPUT X

PS X=0 X=1 a a,0 b,0 b b,1 c,0 c d,0 c,1 d d,0 a,1

fig :a) state diagram (meelay circuit) fig: b) state table

In case of moore circuit ,the directed lines are labeled with only one binary number representing the input that causes the state transition. The output is indicated with the circle below the present state, because the output depends only on the present state and not on the input.



in a

NS

INPUT X

PS X=0 X=1 O/P a a b 0 b b c 0 c d c 1 d d 0

fig: a) state diagram (moore circuit) fig:b) state table

Serial binary adder:

Step1: word statement of the problem: the block diagram of a serial binary adder is shown in fig. it is a synchronous circuit with two input terminals designated X1and X2 which carry the two binary numbers to be added and one output terminal Z which represents the sum. The inputs and outputs consist of fixed-length sequences 0s and 1s.the output of the serial Z_i at time $t_i$is a function of the inputs $X_1(t_i)$ and $X_2(t_i)$ at that time t_{i-1} and of carry which had been generated at t_{i-1}. The carry which represent the past history of the serial adder may be a 0 or 1. The circuit has two states. If one state indicates that carry from the previous addition is a 0, the other state indicates that the carry from the previous addition is a 1

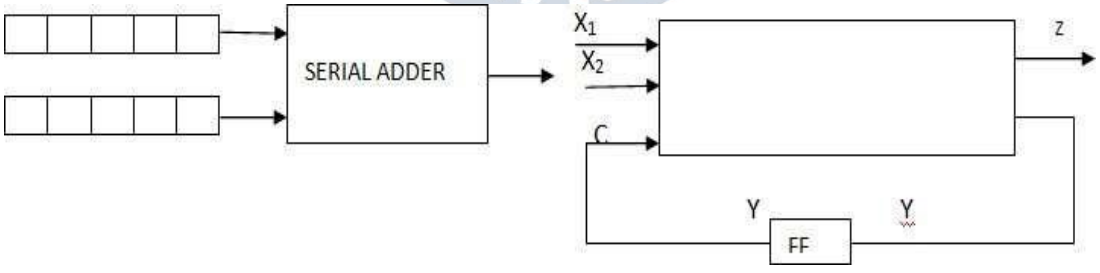
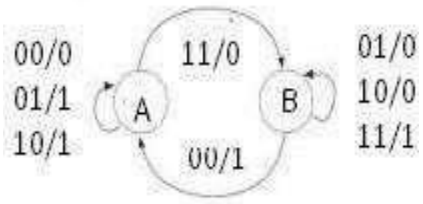


Figure: block diagram of serial binary adder

Step2 and 3: state diagram and state table: let a designate the state of the serial adder at t_i if a carry 0 was generated at t_{i-1}, and let b designate the state of the serial adder at t_i if carry 1 was generated at t_{i-1} .the state of the adder at that time when the present inputs are applied is referred to as the present state(PS) and the state to which the adder goes as a result of the new carry value is referred to as next state(NS).

The behavior of serial adder may be described by the state diagram and state table.



PS	X1 X2	NS ,O/P			
		0	1	1	
0		1	0	1	
0	A	A,0	B,0	B,1	B,0
	B	A,1	B,0	B,0	B,1

Figures: serial adder state diagram and state table

If the machine is in state B, i.e., carry from the previous addition is a 1, inputs X_1=0 and X_2=1 gives sum, 0 and carry 1. So the machine remains in state B and outputs a 0. Inputs X_1=1 and X_2=0 gives sum, 0 and carry 1. So the machine remains in state B and outputs a 0. Inputs X_1=1 and X_2=1 gives sum, 1 and carry 0. So the machine remains in state B and outputs a 1. Inputs X_1=0 and X_2=0 gives sum, 1 and carry 0. So the machine goes to state A and outputs a 1. The state table also gives the same information.

Setp4: reduced standard from state table: the machine is already in this form. So no need to do anything

Step5: state assignment and transition and output table:

The states, A=0 and B=1 have already been assigned. So, the transition and output table is as shown.

PS	**NS**			**O/P**			
0	**0**	**1**	**1**	**0**	**0**	**1**	**1**
	0	**1**			**0**	**1 0**	**1 0** **1**
0	0	0	0	1	0	1	1 1
1	0	1		1	1	1 0	0 1

STEP6: choose type of FF and excitation table: to write table, select the memory element the excitation table is as shown in fig.

PS	I/P	NS	I/P-FF	O/P	y	x1	x2	Y D	Z
0	0	0		0	0	0	0	0	0
	1	0		0		1			
0	1	0		0	0	0	1		
0		1	1	1	1	0			
1		0	0	0	0	1			

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

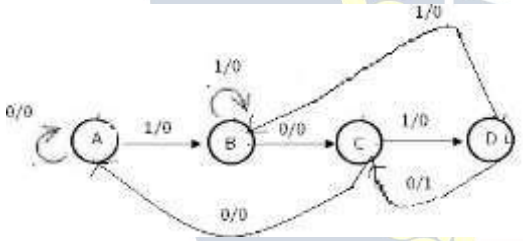
Sequence detector:

Step1: word statement of the problem: a sequence detector is a sequential machine which produces an output 1 every time the desired sequence is detected and an output 0 at all other times

Suppose we want to design a sequence detector to detect the sequence 1010 and say that overlapping is permitted i.e., for example, if the input sequence is 01101010 the corresponding output sequence is 00000101.

Step2 and 3: state diagram and state table: the state diagram and the state table of the sequence detector. At the time t_1, the machine is assumed to be in the initial state designed arbitrarily as A. while in this state, the machine can receive first bit input, either a 0 o r a 1. If the input bit is 0, the machine does not start the detection process because the first bit in the desired sequence is a

1. If the input bit is a 1 the detection process starts.



PS	NS,Z	
	X=0	X=1
A	A,0	B,0
B	C,0	B,0
C	A,0	D,0
D	C,1	B,0

Figure: state diagram and state table of sequence detector

So, the machine goes to state B and outputs a 0. While in state B, the machinery may receive 0 or 1 bit. If the bit is 0, the machine goes to the next state, say state c, because the previous two bits are 10 which are a part of the valid sequence, and outputs 0.. if the bit is a 1, the two bits become 11 and this not a part of the valid sequence

Step4: reduced standard form state table: the machine is already in this form. So no need to do anything.

Step5: state assignment and transition and output table: there are four states therefore two states variables are required. Two state variables can have a maximum of four states, so, all states are utilized and thus there are no invalid states. Hence, there are no don‘t cares. Let a=00, B=01, C=10 and D=11 be the state assignment.

	NS(Y1Y2)		O/P(z)			
PS(y1y2	X=0		X=1		X=0	X=1
A= 0 0	0	0	0	1	0	0
B=0 1	1	0	0	1	0	0
C=1 0	0	0	1	1	0	0
D=1 1	1	1	0	1	1	0

Step6: choose type of flip-flops and form the excitation table: select the D flip-flops as memory elements and draw the excitation table.

				INPUTS -				
PS	**I/P**	**NS**	**FFS**	**O/P**				
y1	Y2	X		Y1	Y2	D1	D2	Z
0	0	0	0	0	0	0	0	
0	0	1	0	1	0	1	0	
0	1	0	1	0	1	0	0	
0	1	1	0	1	0	1	0	
1	0	0	0	0	0	0	0	
1	0	1	1	1	1	1	0	
1	1	0	1	0	1	0	1	
1	1	1		0	1	0	1	0

Step7: K-maps and minimal functions: based on the contents of the excitation table , draw the k-map and simplify them to obtain the minimal expressions for D1 and D2 in terms of y1, y2 and x as shown in fig. The expression for z (z=y1,y2) can be obtained directly from table

Step8**:** implementation: the logic diagram based on these minimal expressions

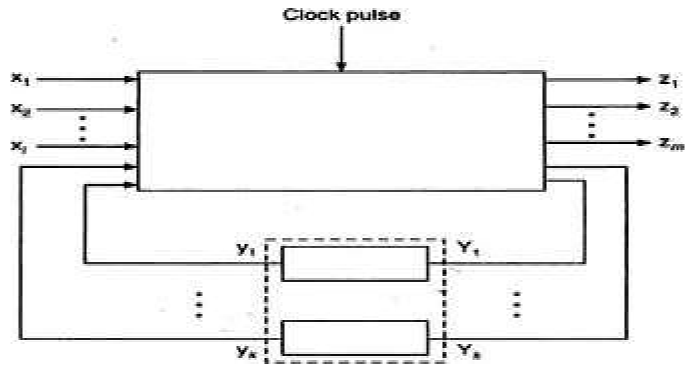
UNIT-V

CAPABILITIES AND MINIMIZATION OF SEQUENTIAL MACHINES

Finite State Machine:

Finite state machine can be defined as a type of machine whose past histories can affect its future behavior in a finite number of ways. To clarify, consider for example of binary full adder. Its output depends on the present input and the carry generated from the previous input. It may have a large number of previous input histories but they can be divided into two types: (i) Input

The most general model of a sequential circuit has inputs, outputs and internal states. A sequential circuit is referred to as a finite state machine (FSM). A finite state machine is abstract model that describes the synchronous sequential machine. The fig. shows the block diagram of a finite state model. X_1, X2,….., X_l, are inputs. Z_1, Z2,….,Z_m are outputs. Y_1,Y_2,….Y_k are state variables, and Y_1,Y_2,….Y_k represent the next state.



Capabilities and limitations of finite-state machine

Let a finite state machine have n states. Let a long sequence of input be given to the machine. The machine will progress starting from its beginning state to the next states according to the state transitions. However, after some time the input string may be longer than n, the number of states. As there are only n states in the machine, it must come to a state it was previously been in and from this phase if the input remains the same the machine will function in a periodically repeating fashion. From here a conclusion that _for a n state machine the output will become periodic after a number of clock pulses less than equal to n can be drawn. States are memory elements. As for a finite state machine the number of states is finite, so finite number of memory elements are required to design a finite state machine.

Limitations:

1. Periodic sequence and limitations of finite states: with n-state machines, we can generate periodic sequences of n states are smaller than n states. For example, in a 6-state machine, we can have a maximum periodic sequence as 0,1,2,3,4,5,0,1….
2. No infinite sequence: consider an infinite sequence such that the output is 1 when and only when the number of inputs received so far is equal to P(P+1)/2 for P=1,2,3….,i.e., the desired input-output sequence has the following form:

Input: x x x x x x x x x x x x x x x x x x x x x x
Output: 1 0 1 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1

Such an infinite sequence cannot be produced by a finite state machine.

3. Limited memory: the finite state machine has a limited memory and due to limited memory it cannot produce certain outputs. Consider a binary multiplier circuit for multiplying two arbitrarily large binary numbers. The memory is not sufficient to store arbitrarily large partial products resulted during multiplication.
   Finite state machines are two types. They differ in the way the output is generate they are:
   a) Mealy type model: in this model, the output is a function of the present state and the present input.
   b) Moore type model: in this model, the output is a function of the present state only.

Mathematical representation of synchronous sequential machine:
The relation between the present state S(t), present input X(t), and next state s(t+1) can be given as

$$S(t+1)= f\{S(t),X(t)\}$$

The value of output Z(t) can be given as Z(t)= g{S(t),X(t)} for mealy model Z(t)= G{S(t)} for Moore model

Because, in a mealy machine, the output depends on the present state and input, where as in a Moore machine, the output depends only on the present state.

Comparison between the Moore machine and mealy machine:

Moore machine	Mealy machine
1. its output is a function of present state only Z(t)= g{S(t)}	1. its output is a function of present state as well as present input Z(t)=g{S(t),X(t)}
2. input changes do not affect the output	2. input changes may affect the output of the circuit
3. it requires more number of states for implementing same function	3. it requires less number of states for implementing same function

Mealy model:

When the output of the sequential circuit depends on the both the present state of the flip-flops and on the inputs, the sequential circuit is referred to as mealy circuit or mealy machine. The fig. shows the logic diagram of the mealy model. Notice that the output depends up on the present state as well as the present inputs. We can easily realize that changes in the input during the clock pulse cannot affect the state of the flip-flop. They can affect the output of the circuit. If the input variations are not synchronized with a clock, he derived output will also not be synchronized with the clock and we get false output. The false outputs can be eliminated by allowing input to change only at the active transition of the clock.

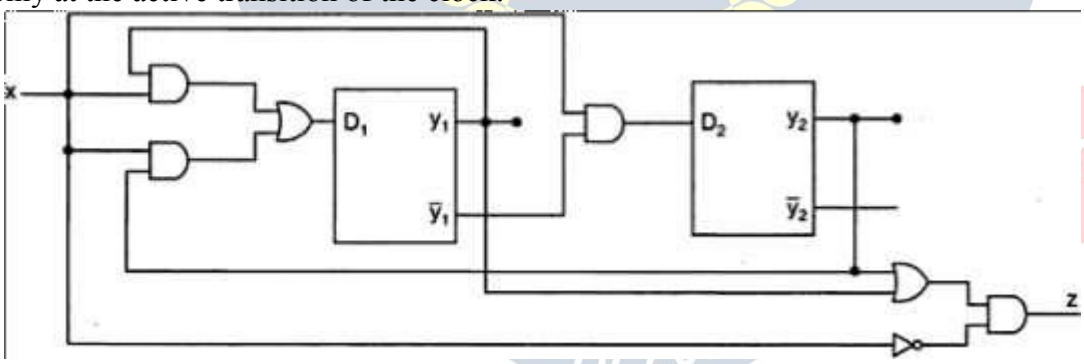


Fig: logic diagram of a mealy model

The behavior of a clocked sequential circuit can be described algebraically by means of state equations. A state equation specifies the next state as a function of the present state and inputs. The mealy model shown in fig. consists of two D flip-flops, an input x and an output z. since the D input of a flip-flop determines the value of the next state, the state equations for the model can be written as

$Y_1 (t+1)=y_1(t)x(t)+y_2(t)x(t)$

$Y_2(t+1)= 1(t)x(t)$

And the output equation is

$Z(t)=\{ y_1(t)+y_2(t)\} X'(t)$

Where y(t+1) is the next state of the flip-flop one clock edge later, x(t) is the present input, and z(t) is the present output. If y1(t+1) are represented by y1(t) and y2(t) , in more compact form, the equations are

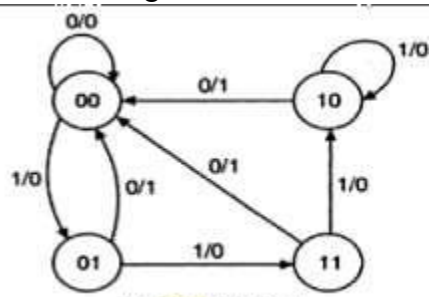
$Y1(t+1)=y1=y1x+y2x$

$Y2(t+1)=y2=y1'x$ $Z=(y1+y2)x'$

The stable table of the mealy model based on the above state equations and output equation is shown in fig. the state diagram based on the state table is shown in fig.

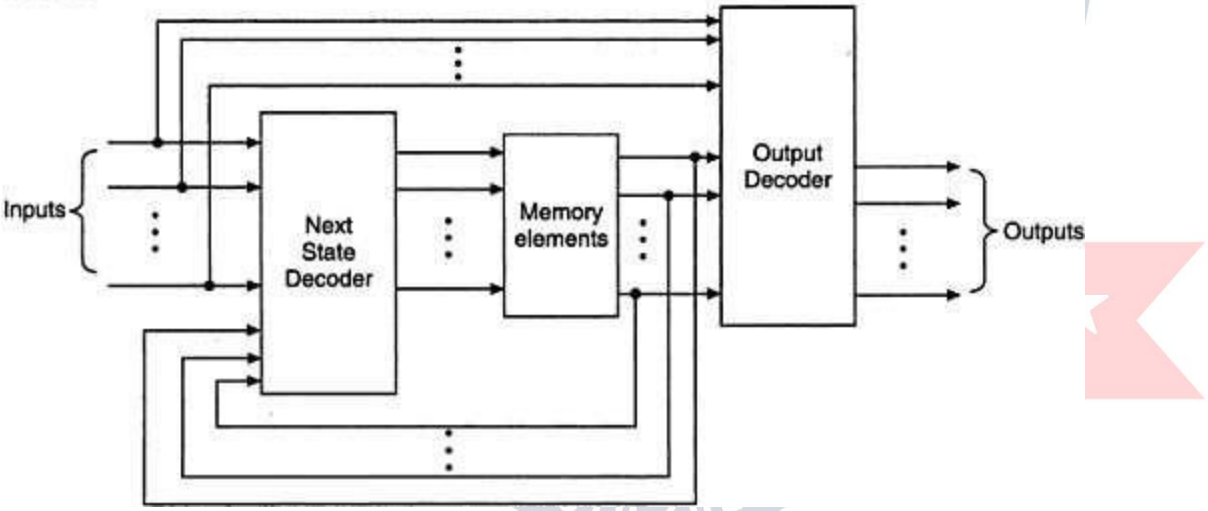
PS		NS				O/P	
		x = 0		x = 1		x = 0	x = 1
y_1	y_2	Y_1	Y_2	Y_1	Y_2	z	z
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0

(a) State table



(b) State diagram

In general form, the mealy circuit can be represented with its block schematic as shown in below fig.



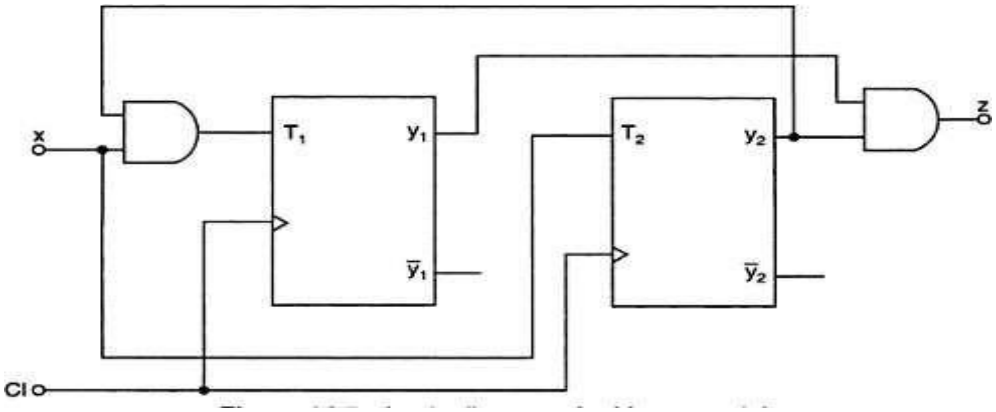
Moore model: when the output of the sequential circuit depends up only on the present state of the flip-flop, the sequential circuit is referred as to as the Moore circuit or the Moore machine.

Notice that the output depend only on the present state. It does not depend upon the input at all. The input is used only to determine the inputs of flip-flops. It is not used to determine the output. The circuit shown has two T flip-flops, one input x, and one output z. it can be described algebraically by two input equations an output equation.

$T_1=y_2x$

$T_2=x$

$Z=y_1y_2$



The characteristic equation of a T-flip-flop is

$$Q(t+1)=TQ'+T'Q$$

The values for the next state can be derived from the state equations by substituting T_1 and T_2 in the characteristic equation yielding

$Y_1(t+1)=Y_1=(y_2x) \oplus =(\ 2\ )y1+(y2x)\ 1$

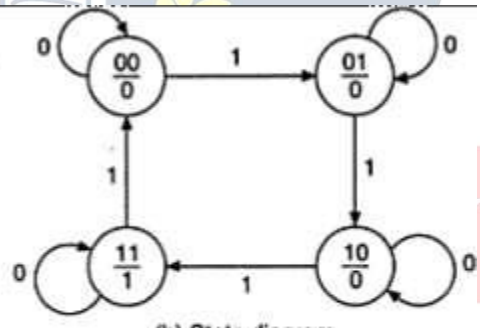
$= y1\ \ 2+ y1\ \ +\ 1y2x$

$= y2\ (t+1)= \oplus y2= x\ 2+\ \ y2$

The state table of the Moore model based on the above state equations and outpu t equation is shown in fig.

PS		NS				O/P
		x = 0		x = 1		
y_1	y_2	Y_1	Y_2	Y_1	Y_2	z
0	0	0	0	0	1	0
0	1	0	1	1	0	0
1	0	1	0	1	1	0
1	1	1	1	0	0	1

(a) State table



(b) State diagram

In general form , the Moore circuit can be represented with its block schematic as shown in below fig.

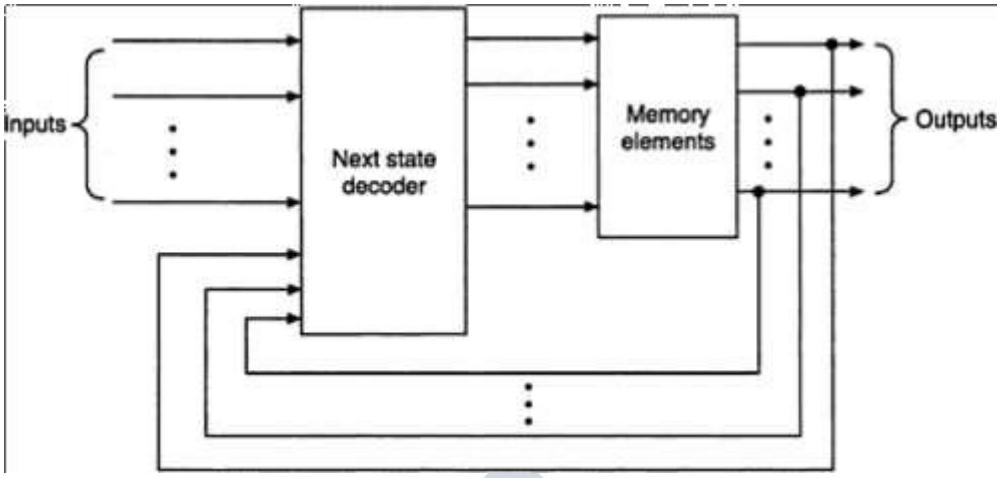


Figure: moore circuit model:

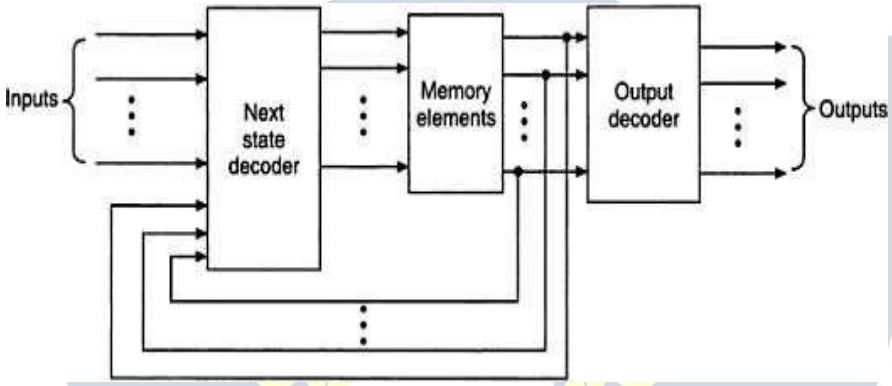


Figure: moore circuit model with an output decoder

Mealy and Moore

- Sequential machines are typically classified as either a Mealy machine or a Moore machine implementation.
- Moore machine: The outputs of the circuit depend only upon the current state ofthe circuit.
- Mealy machine: The outputs of the circuit depend upon both the current state of the circuit and the inputs.

An example to go through the steps

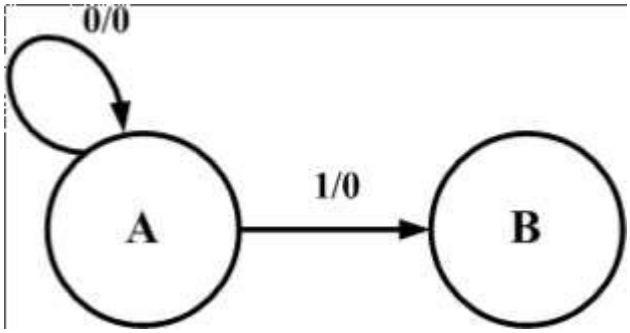
The specification: The circuit will have one input, X, and one output, Z. The output Z will be 0 except when the input sequence 1101 are the last 4 inputs received on X. In that case it will be a 1

Generation of a state diagram

- Create states and meaning for them.

State A – the last input was a 0 and previous inputs unknown. Can also be the reset state. State B – the last input was a 1 and the previous input was a 0. The start of a new sequence possibly.

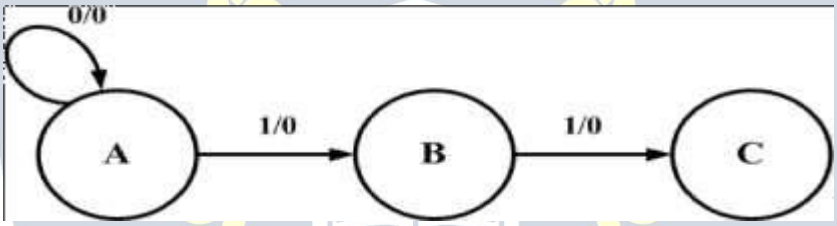
- Capture this in a state diagram



- Capture this in a state diagram

- Circles represent the states
- Lines and arcs represent the transition betweenstates.
- The notation Input/output on the line or arc specifies the input that causes this transition and the output for this change of state.

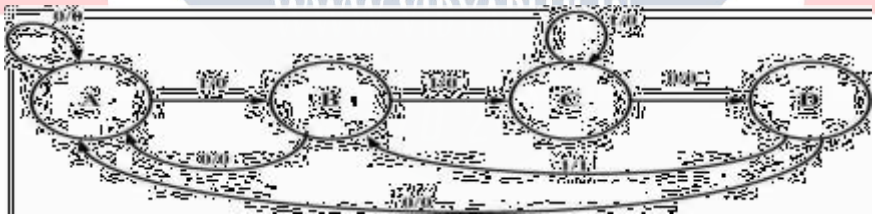
- Add a state C – Have detected the input sequence 11 which is the start of thesequence



Add a state D

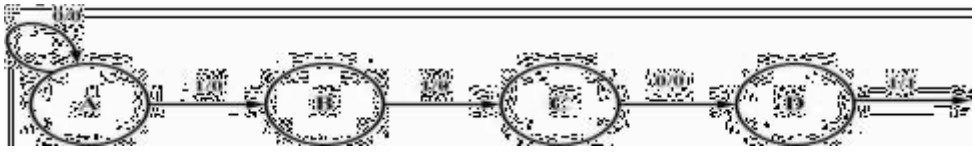
State D – have detected the 3rd input in the start of a sequence, a 0, now having 110. From State D, if the next input is a 1 the sequence has been detected and a 1 is output.

- The previous diagram was incomplete.



In each state the next input could be a 0 or a 1. This must be includedThe state table

- This can be done directly from the state diagram



Prresent State	Next State X =0	X=1	Output X=0	X=1
A	A	B	0	0
B	A	C	0	0
C	D	C	0	0
D	A	B	0	1

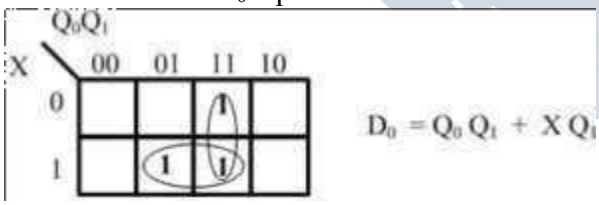
- Now need to do a state assignment **Select a state assignment**

- Will select a grayencoding
- For this state A will be encoded 00, state B 01, state C 11 and state D 10

Prresent State	Next State X =0	X=1	Output X=0	X=1
00	00	01	0	0
01	00	11	0	0
11	10	11	0	0
10	00	01	0	1

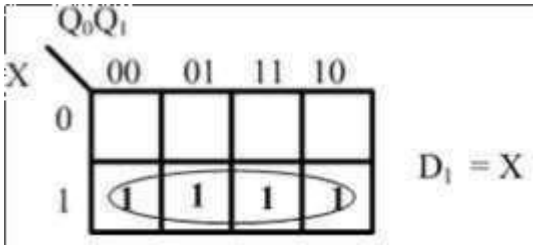
Flip-flop input equations

- Generate the equations for the flip-flop inputs
- Generate the D_0 equation



$D_0 = Q_0 Q_1 + X Q_1$

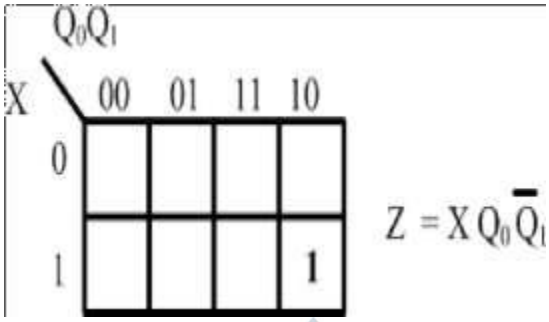
- Generate the D_1 equation



$D_1 = X$

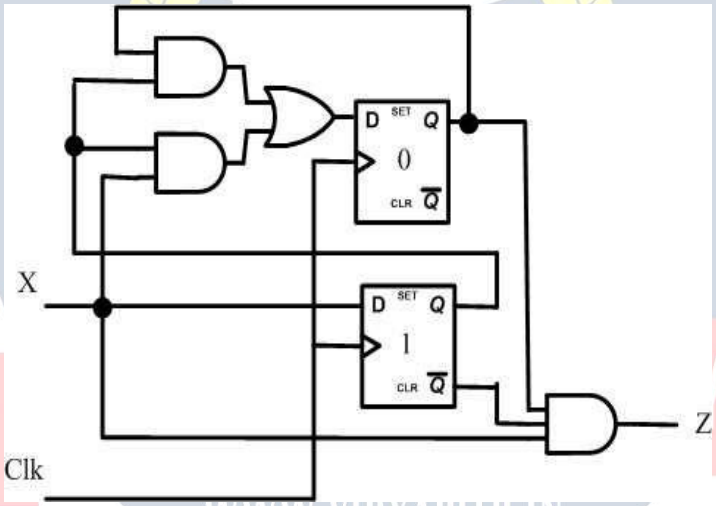
The output equation

- The next step is to generate the equation for the output Z and what is needed to generate it.
- Create a K-map from the truth table.



Now map to a circuit

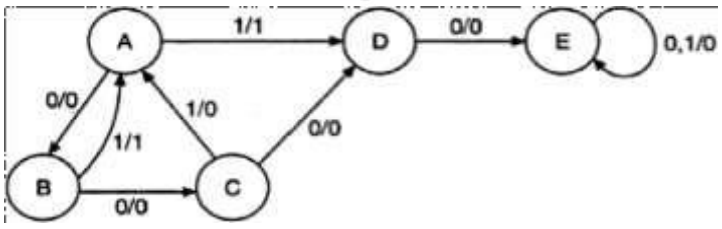
- The circuit has 2 D type F/Fs



Important definitions and theorems:

A). Finite state machine-definitions:

Consider the state diagram of a finite state machine shown in fig. it is five-state machine with one input variable and one output variable.



Successor: looking at the state diagram when present state is A and input is 1, the next state is D. this condition is specified as D is the successor of A. similarly we can say that A is the 1 successor

of B, and C,D is the 11 successor of B and C, C is the 00 successor of A and D, D is the 000 successor of A,E, is the 10 successor of A or 0000 successor of A and so on.

Terminal state: looking at the state diagram , we observe that no such input sequence exists which can take the sequential machine out of state E and thus state E is said to be a terminal state.

Strongly-connected machine: in sequential machines many times certain subsets of states may not be reachable from other subsets of states. Even if the machine does not contain any terminal state. If for every pair of states s_i, s_j, of a sequential machine there exists an input sequence which takes the machine M from s_i to sj, then the sequential machine is said to be strongly connected.

B). state equivalence and machineminimization:

In realizing the logic diagram from a stat table or state diagram many times we come across redundant states. Redundant states are states whose functions can be accomplished by other states. The elimination of redundant states reduces the total number of states of the machines which in turn results in reduction of the number of flip-flops and logic gates, reducing the cost of the final circuit.

Two states are said to be equivalent. When two states are equivalent, one of them can be removed without altering the input output relationship.

State equivalence theorem: it states that two states s_1, and s_2 are equivalent if for every possible input sequence applied. The machine goes to the same next state and generates the same output. That is

If $S_1(t+1)= s_2(t+1)$ and $z_1=z_2$, then $s_1=s_2$

C). distinguishable states and distinguishingsequences:

Two states s_a, and s_b of a sequential machine are distinguishable, if and only if there exists at least one finite input sequence which when applied to the sequential machine causes different outputs sequences depending on weather s_a or s_b is the initial state.

Consider states A and B in the state table, when input X=0, their outputs are 0 and 1 respectively and therefore, states A and B are called 1-distinguishabke. Now consider states A and E . the output sequence is as follows.

X=0 A C,0 and E D, 0 ; outputs are the same

C E,0 and D b,1 ; outputs aredifferent

Here the outputs are different after 2-state transition and hence states A and E are 2- distungishable. Again consider states A and C . the output sequence is as follows:

X=0 A C,0 and C E, 0; outputs are the same C E,0 and

E D,0 ; outputs are the same E D,0 and D B,1 ;

outputs are

different

Here the outputs are different after 3- transition and hence states A and B are 3-distuingshable. the concept of K- distuingshable leads directly to the definition of K-equivalence. States that are not K-distinguishable are said to be K-equivalent.

Truth table for Distunigshable states:

PS	NS,Z	
	X=0	X=1
A	C,0	F,0
B	D,1	F,0
C	E,0	B,0
D	B,1	E,0
E	D,0	B,0
F	D,1	B,0

Merger Chart Methods:

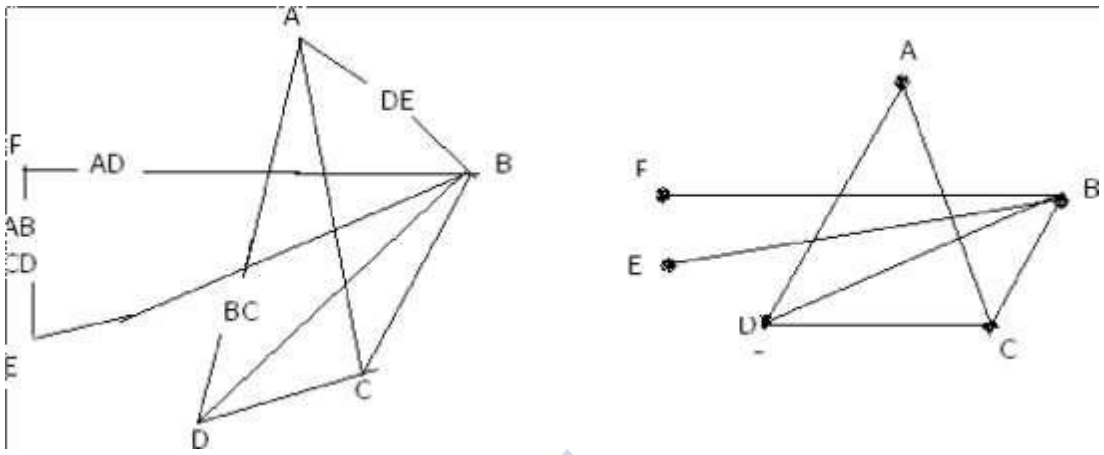
Merger graphs:

The merger graph is a state reducing tool used to reduce states in the incompletely specified machine. The merger graph is defined as follows.

1. Each state in the state table is represented by a vertex in the merger graph. So it contains the same number of vertices as the state table contains states.
2. Each compatible state pair is indicated by an unbroken line draw between the two state vertices
3. Every potentially compatible state pair with non-conflicting outputs but with different next states is connected by a broken line. The implied states are written in theline break between the two potentially compatible states.
4. If two states are incompatible no connecting line is drawn.

Consider a state table of an incompletely specified machine shown in fig. the corresponding merger graph shown in fig. **State table:**

PS	NS,Z			
	I1	I2	I3	I4
A	…	E,1	B,1	….
B	…	D,1	…	F,1
C	F,1	…	…	…
D	…	…	C,1	…
E	C,0	…	A,0	F,1
F	D,0	A,1	B,0	…



a) Merger graph b) simplified merger graph

States A and B have non-conflicting outputs, but the successor under input $I_2$are compatible only if implied states D and E are compatible. So, draw a broken line from A to B with DE written in between states A and C are compatible because the next states and output entries of states A and C are not conflicting. Therefore, a line is drawn between nodes A and C. states A and D have nonconflicting outputs but the successor under input I_3 are B and C. hence join A and D by a broken line with BC entered In between.

Two states are said to be incompatible if no line is drawn between them. If implied states are incompatible, they are crossed and the corresponding line is ignored. Like, implied states D and E are incompatible, so states A and B are also incompatible. Next, it is necessary to check whether the incompatibility of A and B does not invalidate any other broken line. Observe that states E and F also become incompatible because the implied pair AB is incompatible. The broken lines which remain in the graph after all the implied pairs have been verified to be compatible are regarded as complete lines.

After checking all possibilities of incompatibility, the merger graph gives the following seven compatible pairs.

(A, C) (A, D) (B, C) (B, D) (C, D) (B, E) (B, F)

These compatible pairs are further checked for further compatibility. For example, pairs (B,C)(B,D)(C,D) are compatible. So (B, C, D) is also compatible. Also pairs (A,c)(A,D)(C,D) are compatible. So (A,C,D) is also compatible. . In this way the entire set of compatibles of sequential machine can be generated from its compatible pairs.

To find the minimal set of compatibles for state reduction, it is useful to find what are called the maximal compatibles. A set of compatibles state pairs is said to be maximal, if it is not completely covered by any other set of compatible state pairs. The maximum compatible can be found by looking at the merger graph for polygons which are not contained within any higher order complete polygons. For example only triangles (A, C,D) and (B,C,D) are of higher order. The set of maximal compatibles for this sequential machine given as

(A, C, D) (B, C, D) (B, E) (B, F)

Example:

Draw the merger graph and obtain the set of maximal compatibles for the incompletely specified sequential machine whose state table is given in Table 7.24.

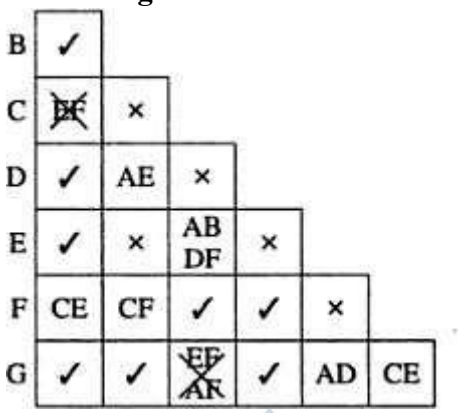
Table 7.24 Example 7.9: State table

PS	NS, Z	
	I_1	I_2
A	E, 0	B, 0
B	F, 0	A, 0
C	E, –	C, 0
D	F, 1	D, 0
E	C, 1	C, 0
F	D, –	B, 0

mark × in the corresponding cell. For example, states B and C are incompatible because their outputs are conflicting and hence the cell corresponding to them contains a cross mark ×. Similarly states B, E; D, E; E, F are incompatible. Hence put a × mark in the corresponding cells. On the other hand, states A and B are compatible and hence the cell corresponding to them contains the check mark ✓. Similarly, cells corresponding to states A, D; A ,E; A, G; B, G; C, F; D, F ; D, G are also compatible. So a check mark is put in those cells also. The implied pairs or pairs corresponding to the state pair are written within the cell as shown in Table 7.26. For example, states A and C are compatible only when implied states E and F are compatible. Therefore, EF is written in the cell corresponding to states A and C. States C and E are compatible only when implied states A and B, and D and F are compatible. So AB and DF are written in the cell corresponding to states C and E. In a similar way, the entire merger table is written. Now it is necessary to check whether the implied pairs are compatible or not by observing the merger table. The implied states are incompatible if the corresponding cell contains a ×. For example, implied pair E, F is incompatible because cell EF contains a ×. Similarly, implied pairs EF, AF are incompatible because EF contains a ×. It is indicated by a ×.

PS	NS, Z			
	00	01	11	10
A	E, 0	–	–	–
B	–	F, 1	E, 1	A, 1
C	F, 0	–	A, 0	F, 1
D	–	–	A, 1	–
E	–	C, 0	B, 0	D, 1
F	C, 0	C, 1	–	–
G	E, 0	–	–	A, 1

Figure: state table



State Minimization:
Completely Specified Machines

- Two states, s_i and s_j of machine M are *distinguishable* if and only if there exists a finite input sequence which when applied to M causes different output sequences dependingon whether M started in s_i or s_j.
- Such a sequence is called a *distinguishing sequence* for (s_i, s_j).
- If there exists a distinguishing sequence of length k for (s_i, s_j), they are said to be *k-distinguishable*.

EXAMPLE:

PS	NS, z	
	x=0	x=1
A	E, 0	D, 1
B	F, 0	D, 0
C	E, 0	B, 1
D	F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0

- states A and B are 1-distinguishable, since a 1 input applied to A yields an output 1, versus an output 0 from B.
- states A and E are 3-distinguishable, since input sequence 111 applied to A yieldsoutput 100, versus an output 101 from E.
- States s_i and s_j ($s_i \sim s_j$) are said to be equivalent iff no distinguishing sequence exists for (s_i, s_j).
- If $s_i \sim s_j$ and $s_j \sim s_k$, then $s_i \sim s_k$. So state equivalence is an equivalence relation (i.e. it is a reflexive, symmetric and transitiverelation).
- An equivalence relation partitions the elements of a set into equivalence classes.
- Property: If $s_i \sim s_j$, their corresponding X-successors, for all inputs X, are also equivalent.
- Procedure: Group states of M so that two states are in the same group iff they are equivalent (forms a partition of the states).

Completely Specified Machines

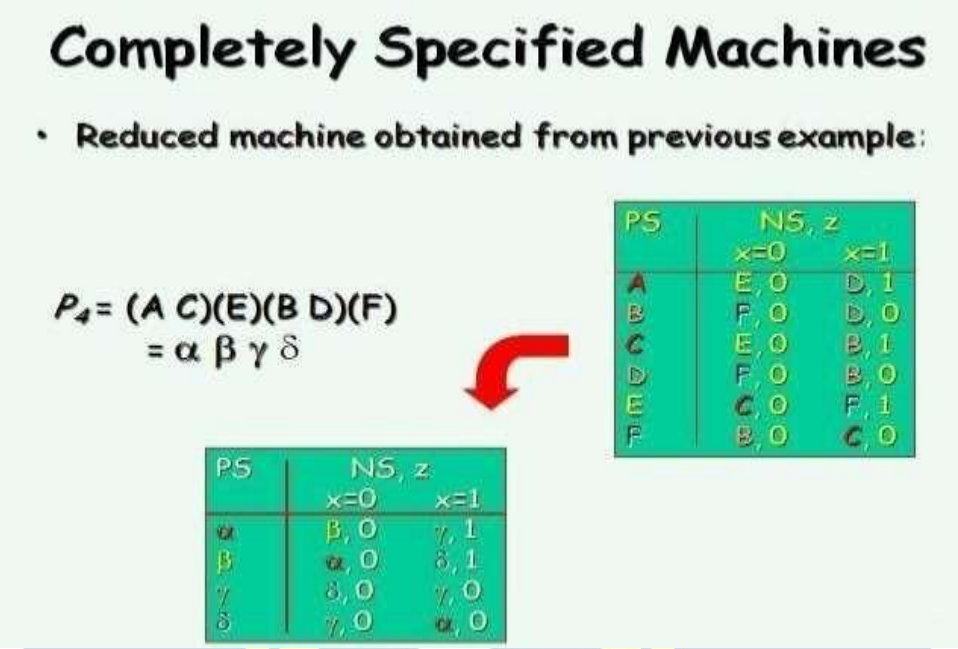
PS	NS, z	
	x=0	x=1
A	E, 0	D, 1
B	F, 0	D, 0
C	E, 0	B, 1
D	F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0

P_i : partition using distinguishing sequences of length i.
Partition: Distinguishing Sequence:
P_0 = (A B C D E F)
P_1 = (A C E)(B D F) x =1
P_2 = (A C E)(B D)(F) x =1; x =1
P_3 = (A C)(E)(B D)(F) x =1; x =1; x =1
P_4 = (A C)(E)(B D)(F)
Algorithm terminates when $P_k = P_{K+1}$ Outline of state minimization procedure:

- All states equivalent to each other form an equivalence class. These may becombined into one state in the reduced (quotient) machine.
- Start an initial partition of a single block. Iteratively refine this partition by separating the 1-distinguishable states, 2-distinguishable states and soon.
- To obtain P_{k+1}, for each block B_i of P_k, create one block of states that not 1- distinguishable within B_i , and create different blocks states that are 1-distinguishable within B_i .

Theorem: The equivalence partition is unique.
Theorem: If two states, s_i and s_j, of machine M are distinguishable, then they are(n-1)-distinguishable, where n is the number of states in M.
Definition: Two machines, M_1 and M_2, are *equivalent* ($M_1 \sim M_2$) if, for every state in M_1 there is a corresponding equivalent state in M_2 and vice versa.

Theorem. For every machine M there is a minimum machine $M_{red} \sim M$. M_{red} is unique up to isomorphism.



State Minimization of CSMs: Complexity
Algorithm DFA ~ DFA_{min}

Input: A finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ with no unreachable states.

Output: A minimum finite automaton $M' = (Q', \Sigma, \delta', q'_0, F')$. *Method*:

1. $t := 2$; $Q_0 := \{$ undefined $\}$; $Q_1 := F$; $Q_2 := Q \setminus F$.
2. while there is $0 < i \leq t$, $a \in \Sigma$ with $\delta(Q_i,a) \subseteq Q_j$, for all $j \leq t$ do (a) Choose such an i, a, and $j \leq t$ with $\delta(Q_i,a) \subseteq Q_j \subset \Sigma$.
   (b) $Q_{t+1} := \{q \in Q_i \mid \delta(q,a) \in Q_j\}$;
   $Q_i := Q_i \setminus Q_{t+1}$; $t := t+1$.

end.

3. (* Denote $[q]$ the equivalence class of state q, and $\{Q_i\}$ the set of all equivalence classes. *)

$Q' := \{Q_1, Q_2, ..., Q_t\}$. $q'_0 := [q_0]$.

$F' := \{ [q] \in Q' \mid q \in F \}$.

$\delta'([q], a) := [\delta(q,a)]$ for all $q \in Q$, $a \in \Sigma$.

Standard implementation: $O(kn^2)$, where $n = |Q|$ and $k = |\Sigma|$ Modification of the body of the while loop:

1. Choose such an i, $a \in \Sigma$, and choose $j_1, j_2 \leq t$ with $j_1 \neq j_2$, $\delta(Q_i,a) \subseteq Q_{j1} \subset \Sigma$, and $\delta(Q_i,a) \subseteq Q_{j2} \subset \Sigma$.

2. If $|\{q \in Q_i \mid \delta(q,a) \in Q_{j1}\}| \leq |\{q \in Q_i \mid \delta(q,a) \in Q_{j2}\}|$

then $Q_{t+1} := \{q \in Q_i \mid \delta(q,a) \in Q_{j1}\}$ else

$Q_{t+1} := \{q \in Q_i \mid \delta(q,a) \in Q_{j2}\}$ fI;

$Q_i := Q_i \setminus Q_{t+1}$; $t := t+1$.

(i.e. put smallest set in $t+1$)

Note: $|Q_{t+1}| \leq 1/2|Q_i|$. Therefore, for all $q \in Q$, the name of the class which contains a given state q changes at most $\log(n)$ times.

Goal: Develop an implementation such that all computations can be assigned to transitions containing a state for which the name of the corresponding class is changed.

Suitable data structures achieve an $O(kn \log n)$ implementation.

State Minimization:

Incompletely Specified Machines

Statement of the problem: given an incompletely specified machine M, find a machine M' such that:

- on any input sequence, M' produces the same outputs as M, whenever M is specified.
- there does not exist a machine M'' with fewer states than M' which has the same property

Machine *M*:

PS	NS, z	
	x=0	x=1
s1	s3, 0	s2, 0
s2	s2, -	s3, 0
s3	s3, 1	s2, 0

Attempt to reduce this case to usual state minimization of completely specified machines.

- Brute Force Method: Force the don't cares to all their possible values and choosethe smallest of the completely specified machines so obtained.
- In this example, it means to state minimize two completely specified machines obtained from M, by setting the don't care to either 0 and 1.

Suppose that the - is set to be a 0.

PS	NS, z	
	x=0	x=1
s1	s3, 0	s2, 0
s2	s2, 0	s3, 0
s3	s3, 1	s2, 0

- States s1 and s2 are equivalent if s3 and s2 are equivalent, but s3 and s2 assert different outputs under input 0, so s1 and s2 are not equivalent.
- States s1 and s3 are not equivalent either.

- So this completely specified machine cannot be reduced further (3 states is the minimum).

Suppose that the - is set to be a 1.

PS	NS, z	
	x=0	x=1
s1	s3, 0	s2, 0
s2	s2, 1	s3, 0
s3	s3, 1	s2, 0

- States s1 is incompatible with both s2 and s3.
- States s3 and s2 are equivalent.
- So number of states is reduced from 3 to 2.

Machine M''_{red}:

PS	NS, z	
	x=0	x=1
A	A, 1	A, 0
B	B, 0	A, 0

Can this always be done? Machine *M*:

PS	NS, z	
	x=0	x=1
s1	s3, 0	s2, 0
s2	s2, -	s1, 0
s3	s1, 1	s2, 0

Machine M_2:

PS	NS, z	
	x=0	x=1
s1	s3, 0	s2, 0
s2	s2, 0	s1, 0
s3	s1, 1	s2, 0

Machine M_3:

PS	NS, z	
	x=0	x=1
s1	s3, 0	s2, 0
s2	s2, 1	s1, 0
s3	s1, 1	s2, 0

Machine M_2 and M_3 are formed by filling in the unspecified entry in M with 0 and 1, respectively.

Both machines M_2 and M_3 cannot be reduced.
Conclusion?: M cannot be minimized further!
But is it a correct conclusion?
Note: that we want to _merge' two states when, for any input sequence, they generate the same output sequence, but only where both outputs are specified.
Definition: A set of states is compatible if they agree on the outputs where they are all specified.
Machine *M''* :

PS	NS, z	
	x=0	x=1
s1	s3, 0	s2, 0
s2	s2, -	s1, 0
s3	s1, 1	s2, 0

In this case we have two compatible sets: A = (s1, s2) and B = (s3, s2). A reduced machine M_{red} can be built as follows. Machine M_{red}

PS	NS, z	
	x=0	x=1
A	A, 1	A, 0
B	B, 0	A, 0

PS	NS, z			
	I1	I2	I3	I4
s1	s3,0	s1,-	-	-
s2	s6,-	s2,0	s1,-	-
s3	-,1	-,-	s4,0	-
s4	s1,0	-,-	-	s5,1
s5	-,-	s5,-	s2,1	s1,1
s6	-,-	s2,1	s6,-	s4,1

A set of compatibles that cover all states is: (s3s6), (s4s6), (s1s6), (s4s5), (s2s5).
But (s3s6) requires (s4s6),
(s4s6) requires(s4s5), (s4s5) requires (s1s5),
(s1s6) requires (s1s2), (s1s2) requires (s3s6),
(s2s5) requires (s1s2).
So, this selection of compatibles requires too many other compatibles...

PS	NS, z			
	I1	I2	I3	I4
s1	s3,0	s1,-	-	-
s2	s6,-	s2,0	s1,-	-
s3	-,1	-,-	s4,0	-
s4	s1,0	-,-	-	s5,1
s5	-,-	s5,-	s2,1	s1,1
s6	-,-	s2,1	s6,-	s4,1

- Another set of compatibles that covers all states is (s1s2s5), (s3s6), (s4s5). □ But (s1s2s5) requires (s3s6) (s3s6) requires (s4s6)
- (s4s6) requires (s4s5) (s4s5) requires (s1s5).
- So must select also (s4s6) and (s1s5).
- Selection of minimum set is a binate covering problem

When a next state is unspecified, the future behavior of the machine is unpredictable. This suggests the definition of admissible input sequence.

Definition. An input sequence is *admissible*, for a starting state of a machine if no unspecified next state is encountered, except possibly at the final step.

Definition. State s_i of machine M_1 is said to *cover*, or *contain*, state s_j of M_2 provided

1. every input sequence admissible to s_j is also admissible to s_i, and
2. its application to both M_1 and M_2 (initially is s_i and s_j, respectively) results in identical output sequences whenever the outputs of M_2 are specified.