

LECTURE NOTES
ON
DIGITAL ELECTRONICS

B.Tech III semester (R18)

V.BINDUSREE
Assistant Professor

J.SRAVANA
Assistant Professor

ELECTRICAL AND ELECTRONICS ENGINEERING
INSTITUTE OF AERONAUTICAL ENGINEERING
(Autonomous)
DUNDIGAL, HYDERABAD -500043

MODULE-I
NUMBER SYSTEMS

\

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 the same, 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 Symbolic Analysis 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				

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

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

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 errors resulting 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 errors commonly introduced by humans in writing down or remembering identification numbers.

Error detection codes

3) 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.

4) 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 errors resulting 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 errors commonly introduced by humans in writing down or remembering identification numbers.

5) 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: $123.45 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 + 4 \cdot 10^{-1} + 5 \cdot 10^{-2}$.

Base b number: $N = a_{q-1}b^{q-1} + \dots + a_1b^1 + a_0b^0 + \dots + a_{-p}b^{-p}$
 $b > 1, \quad 0 \leq a_i \leq b-1$
Integer part: $a_{q-1}a_{q-2} \dots a_0$
Fractional part: $a_{-1}a_{-2} \dots a_{-p}$.
Most significant digit: $a_{q-1} \dots$
Least significant digit: a_{-p}

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

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

Decimal to Binary

Fractional number:

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

	Q_i	r_i							
	216	$0 = a_0$	0.354	$2 = 0.708$	hence	$a_{-1} = 0$			
	108	$0 = a_1$	0.708	$2 = 1.416$	hence	$a_{-2} = 1$			
	54	$0 = a_2$	0.416	$2 = 0.832$	hence	$a_{-3} = 0$			
	27	$0 = a_3$	0.832	$2 = 1.664$	hence	$a_{-4} = 1$			
E	13	$1 = a_4$	0.664	$2 = 1.328$	hence	$a_{-5} = 1$			
	6	$1 = a_5$	0.328	$2 = 0.656$	hence	$a_{-6} = 0$			
	3	$0 = a_6$.			$a_{-7} = 1$			
	1	$1 = a_7$				etc.			
T		$1 = a_8$							

Thus, $(432.354)_{10} = (110110000.0101101...)_{2}$

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	Parity	Code
	<u>Bits</u>	<u>Bit</u>	<u>Word</u>
	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

paritybit 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 : $1001011 + 100 = 1001011100 = M$

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

$$\begin{array}{r} \overline{) 100101100} \\ \underline{1011} \\ 001001 \\ \underline{1001} \\ 0010 \\ \underline{0010} \\ 0000 \\ \underline{1011} \\ 0000 \end{array} \quad \leftarrow \text{Remainder}$$

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 asset of elements, a set of operators, and a number of unproved axioms or postulates. A *set* of elements is any collection of objects having a common property. If S is a set and x and y are certain objects, then $x \in S$ Denotes that x is a member of the set S , and $y \notin S$ denotes that y is not an

element of S . A set with a denumerable number of elements is specified by braces: $A = \{1, 2, 3, 4\}$, i.e. the elements of set A are the numbers 1, 2, 3, and 4. A *binary operator* defined on a set S of elements is a rule that assigns to each pair of elements 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 $a, b, c \in S$.

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 $N = \{1, 2, 3, 4, \dots, 9\}$ is closed with respect to the binary operator plus (+) by the rule of arithmetic addition, since for any $a, b \in N$ we obtain a unique $c \in N$ 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 $x, y, z \in S$, for all 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 $x, y, z \in S$

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 $e \in S$ with the property $e * x = x * e = x$ for every $x \in S$

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 \cdot and $+$ which refer to logical AND and logical OR
- $x + 0 = x$
- $x \cdot 0 = 0$
- $x + 1 = 1$
- $x \cdot 1 = x$
- $x + x = x$

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

DeMorgan's Theorem

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

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

Generalized DeMorgan's

Theorem (a) $(a + b + \dots$

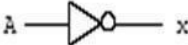
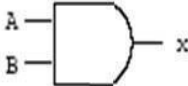
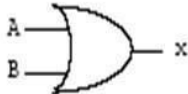
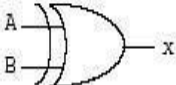
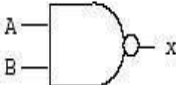
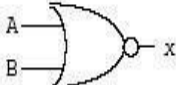
$z)' = a'b' \dots z'$

(b) $(a \cdot b \dots z)' = a' + b' + \dots z'$

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		$x = A'$	<table><tr><th>A</th><th>x</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	x	0	1	1	0									
A	x																	
0	1																	
1	0																	
AND		$x = AB$	<table><tr><th>A</th><th>B</th><th>x</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table> <p>True if both are true.</p>	A	B	x	0	0	0	0	1	0	1	0	0	1	1	1
A	B	x																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR		$x = A + B$	<table><tr><th>A</th><th>B</th><th>x</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table> <p>True if either one is true.</p>	A	B	x	0	0	0	0	1	1	1	0	1	1	1	1
A	B	x																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
• Other common gates include:																		
Name	Graphic symbol	Algebraic function	Truth table															
Exclusive-OR (XOR)		$x = A \oplus B$ $= A'B + AB'$	<table><tr><th>A</th><th>B</th><th>x</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table> <p>Parity check: True if only one is true.</p>	A	B	x	0	0	0	0	1	1	1	0	1	1	1	0
A	B	x																
0	0	0																
0	1	1																
1	0	1																
1	1	0																
NAND		$x = (AB)'$	<table><tr><th>A</th><th>B</th><th>x</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table> <p>Inversion of AND.</p>	A	B	x	0	0	1	0	1	1	1	0	1	1	1	0
A	B	x																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
NOR		$x = A + B$	<table><tr><th>A</th><th>B</th><th>x</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table> <p>Inversion of OR.</p>	A	B	x	0	0	1	0	1	0	1	0	0	1	1	0
A	B	x																
0	0	1																
0	1	0																
1	0	0																
1	1	0																

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 blocks 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.

MULTILEVEL IMPLEMENTATION USING NAND AND NOR GATE:

Two-level implementation means that any path from input to output contains maximum two gates hence the name two-level for the two levels of gates.

Implementing Two-Level logic using NOR gate requires the Boolean expression to be in Product of Sum (POS) form.

In Product of Sum form, 1st level of the gate is OR gate and 2nd level of the gate is AND gate.

To implement a Boolean function using NOR gate, there are basically three steps;

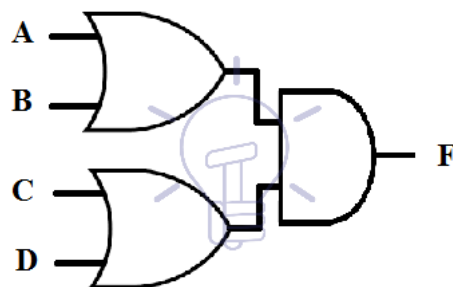
Product of Sum Form

First, you need to have a simplified Product of Sum expression for the function you need to implement.

Simplified Product of Sum expression can be made using Karnaugh Map (K-map) by combining the '0's and then inverting the output function.

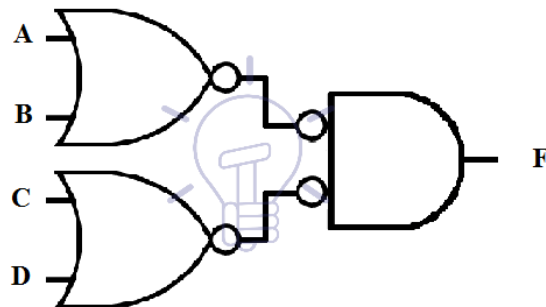
$$F = (A + B)(C + D)$$

Draw its schematic using AND-OR NOT gates as shown in the figure given below.



Mixed Notation

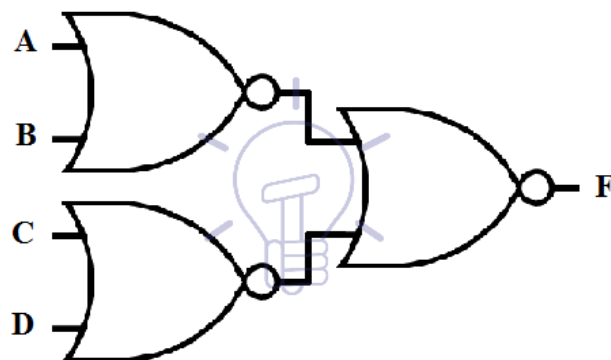
Next step is to draw the above-mentioned schematic using OR-Invert and Invert-AND gates. OR-Invert should replace OR gates and invert-AND replaces AND gates. This schematic is said to be in mixed notation and its schematic is given below.



A bubble means complement. Two bubbles along a line mean double complementation and they cancel each other. However, a single bubble along a line should be compensated by inserting an Inverter in that line or if it is an input line then you can also feed a complemented input if available.

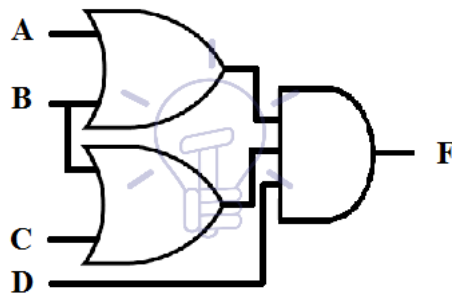
NOR Gate Conversion

The last step is to redraw the whole schematic replacing OR-Invert and Invert-AND gate symbol by NOR gate symbol because OR-Invert and Invert-AND are equivalent to NOR gate. The final schematic is shown in the figure given below.

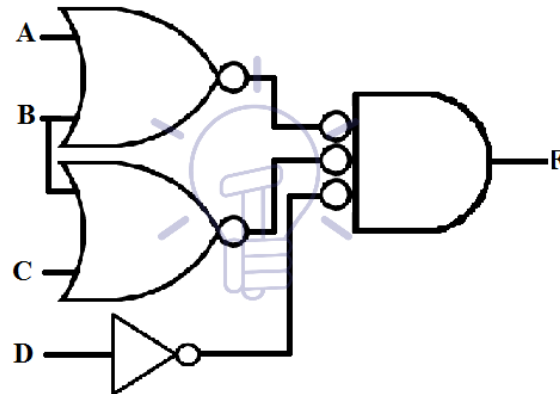


$$F = (A + B)(B + C)D$$

This function is in simplified Product of Sum form. First, we need to draw its OR-AND schematic.

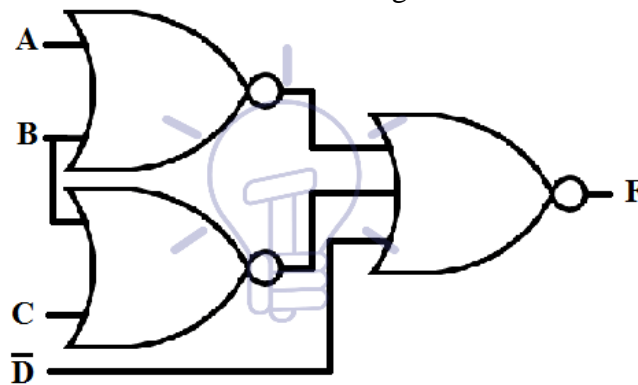


Now we convert the above-given schematic into mixed notation by converting OR gate into OR-INVERT and AND gate into INVERT-AND.



Input line D to the input of AND gate has a single bubble. To compensate this bubble we need to either insert an inverter in this line or complement the input D if available.

Now replace every OR-Invert and Invert-AND with NOR gate as shown in the figure given below.



Multi level implementation using NOR Gate

Schematic having more than two levels of gates is known as a multi-level schematic.

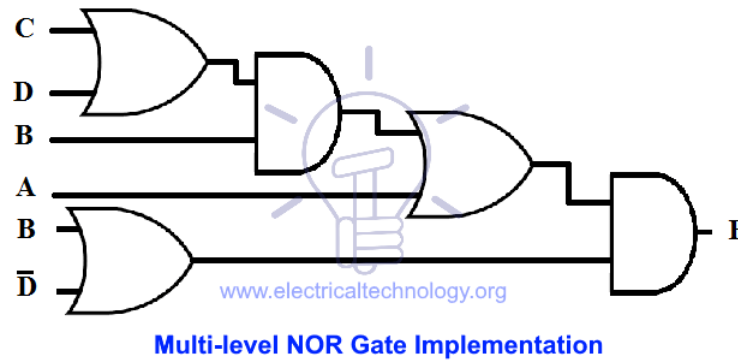
We can implement multi-level POS expression using NOR gate. The conversion of multi-level expression into NOR gate has the same method as two-level implementation.

The multi-level expression can be converted into two-level expression but for the sake of realization, we will implement a multi-level expression.

Suppose a 4-level function:

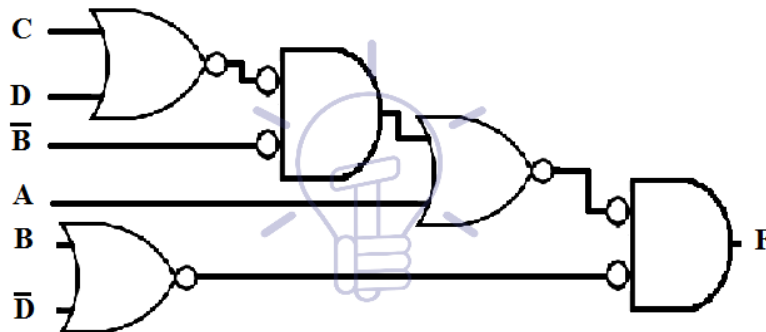
$$F = (A + B(C + D))(B + D')$$

First, we will draw its schematic using AND, OR, NOT gates.



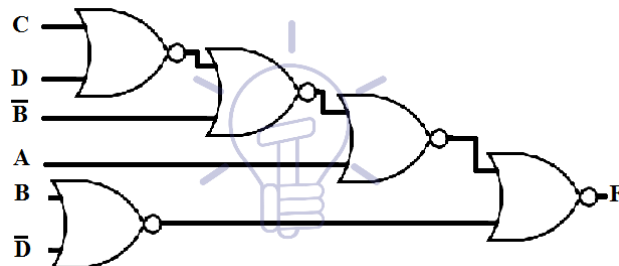
Notice the OR-AND pattern like two-level implementation. It can be easily converted since the bubble cancels each other.

Now we will convert it into mixed notation for NOR.



The two bubbles along a single line cancel each other. However, there is a single bubble at the 2nd level gate's input. so we will complement the input B to compensate the bubble.

Now redraw the whole schematic replacing OR-Invert and Invert-AND with NOR gate symbol as shown in the figure below.

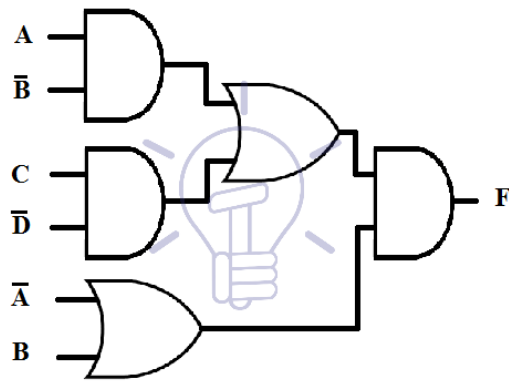


3-Level Implementation & Example using NOR Gate

A 3-level implementation using NOR gate's Example is given below;

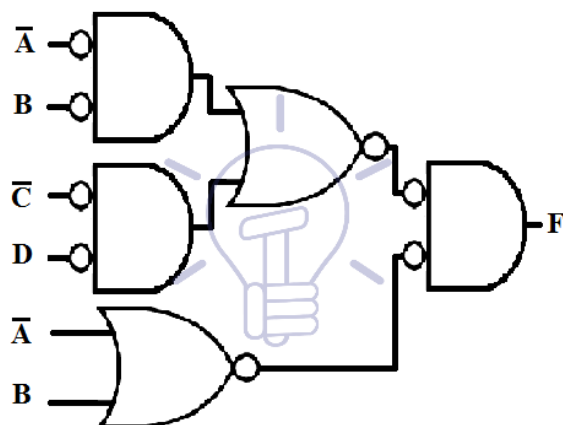
$$F = (AB' + CD')(A' + B)$$

First, we will draw its schematic using AND,OR,NOT gates as given in the figure below.



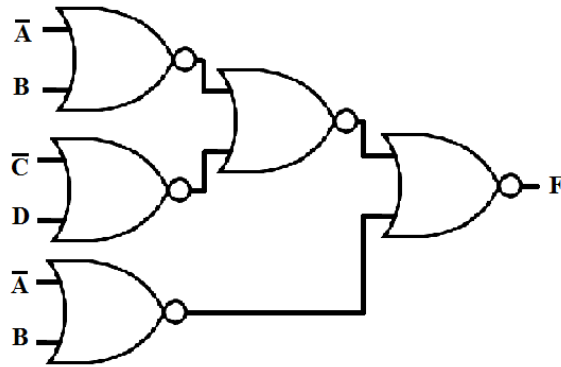
3-Level NOR Gate Implementation

Now we will convert it into mixed notation for NOR.



The single bubbles at the input line of all first level gates need an inverter or the inputs to be complimented. The two bubbles along the same line cancel each other.

Now that all the bubbles have been accounted for, we will redraw this schematic by replacing OR-Invert and Invert-AND with NOR gates as shown in the figure below.



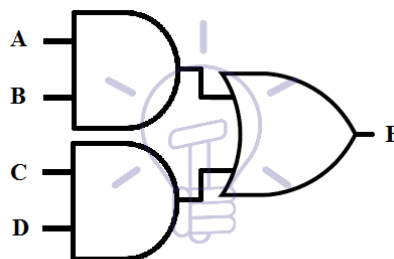
NAND GATE:

NAND Gate is a universal logic gate which means any Boolean logic can be implemented using NAND gate including individual logic gates. In other words, any kind of Boolean function can be implemented using only NAND gates.

NAND gate is commercially used because it allows the access to wired logic which is a logic function formed by connecting the outputs of NAND gates. Wired logic does not consist of a physical gate but the wires behave as a logic function. The other reason for commercial usage of NAND gate is that it can be easily fabricated and has a low fabrication cost. It also shrinks the schematic by decreasing the number of gates, which results in small size and as small delay, fast speed and Low power consumption.

As we know a typical Boolean function implementation consists of AND, OR and NOT gates. To implement a whole Boolean function using NAND gate first, we need to convert these gates into NAND gate.

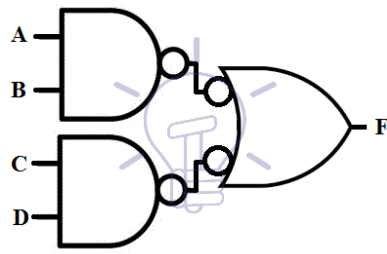
Related Articles:



Two-Level Implementation

Mixed Notation

2nd step is to convert the AND-OR schematic into mixed notation. In mixed notation for NAND gate, AND gate is converted into AND-invert and OR gate is converted into INVERT-OR. Mixed notation design for the above function is given below.



Mixed Notation for NAND Gate

Notice the bubble in a single line. A single bubble means a complement (inversion). Two bubbles on the same line mean double complementation which cancels each other.

MODULE-II

COMBINATIONAL LOGIC CIRCUITS

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:

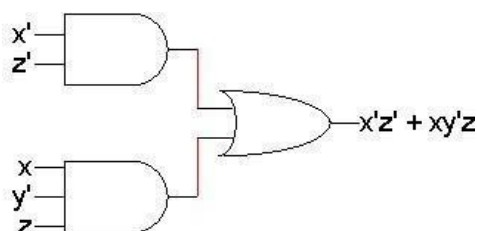
- Boolean algebra helps us simplify expressions and circuits
- Karnaugh Map: A graphical technique for simplifying a Boolean expression into either form:

minimal sum of products
(MSP) o minimal product of
sums (MPS)

- Goal of the simplification.

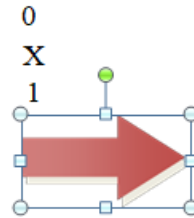
There are a minimal number of product/sum terms oEach term has a minimal number of literals

- 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	1
	$x'y'$	$x'y$
	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

		Y	
		0	1
X	0	$x'y'$	$x'y$
	1	xy'	xy

	y'	y
x'	$x'y'$	$x'y$
x	xy'	xy

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'

		Y
		y'
X		$x'y'$
		$x'y$

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

$$= x' + 1 \quad [y + y' = 1]$$

$$= x' \quad [x + 1 = x]$$

		Y	
		0	1
X	0	$x'y'$	$x'y$
	1	xy'	xy

		Y'	Y
X'		$x'y'$	$x'y$
X		xy'	xy

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'

		y
	<u>$x'y'$</u>	<u>$x'y$</u>
x	<u>xy'</u>	<u>xy</u>

- 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:

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

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

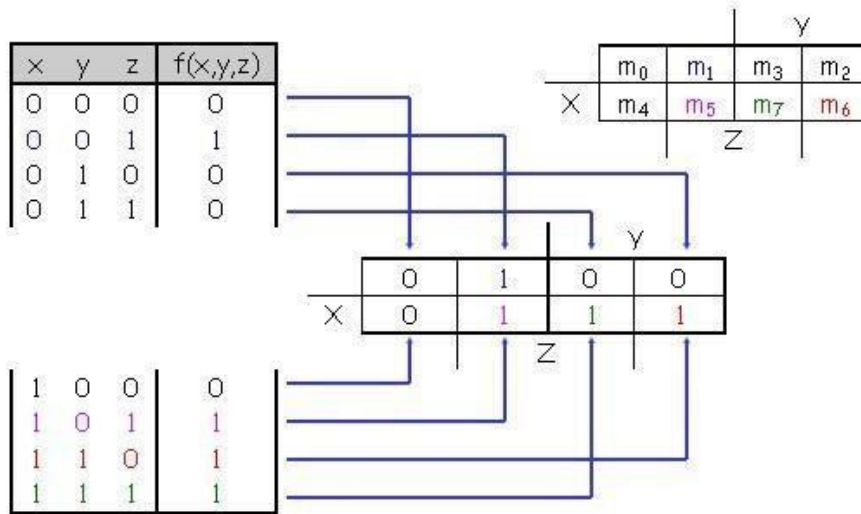
- Another way to label the K-map (use whichever you like):

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

		Y			
		m_0	m_1	m_3	m_2
X		m_4	m_5	m_7	m_6
		Z			

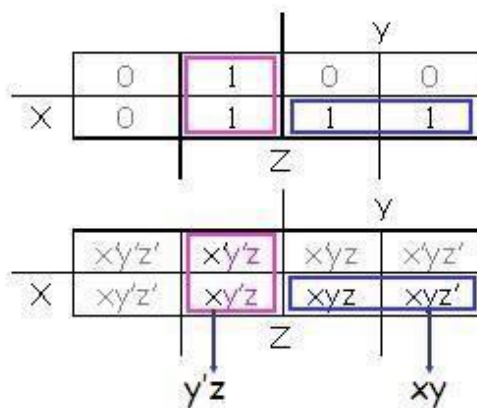
K-MAPS FROM TRUTH TABLES

- 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

			Y	
	0	1	0	0
X	0	1	1	1
		Z		

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

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

UNSIMPLIFYING EXPRESSIONS

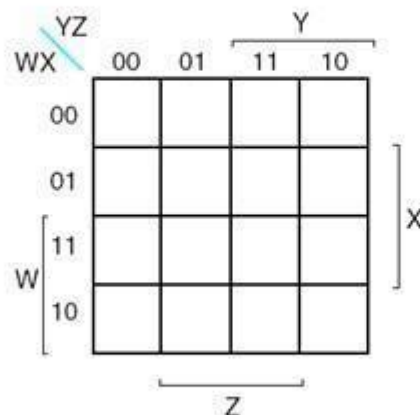
- You can also convert the expression to a sum of minterms with Boolean algebra
 - Apply the distributive law in reverse to add in missing variables.
 - Very few people actually do this, but it's occasionally useful.

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

- 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

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

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

		Y				
		m ₀	m ₁	m ₃	m ₂	
		m ₄	m ₅	m ₇	m ₆	X
W		m ₁₂	m ₁₃	m ₁₅	m ₁₄	
		m ₈	m ₉	m ₁₁	m ₁₀	
		Z				

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

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

		Y				
		w'x'y'z'	w'x'y'z	w'x'yz	w'x'yz'	
		w'xy'z'	w'xy'z	w'xyz	w'xyz'	X
W		wxy'z'	wxy'z	wxyz	wxyz'	
		wx'y'z'	wx'y'z	wx'yz	wx'yz'	
		Z				

Simplify $m_0+m_2+m_5+m_8+m_{10}+m_{13}$

PoS Optimization

- Maxterms are grouped to find minimal PoS expression

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

- $F(W,X,Y,Z) = \prod M(0,1,2,4,5)$

		00	01	YZ	11	10
x	0	$x+y+z$	$x+y+z'$		$x+y'+z'$	$x+y'+z$
	1	$x'+y+z$	$x'+y+z'$		$x'+y'+z'$	$x'+y'+z$

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

		00	01	YZ	11	10
x	0	0	0		1	0
	1	0	0		1	1

PoS Optimization from SoP

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

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

		YZ		Y	
		00	01	11	10
W	00			0	
	01	0		0	0
	11	0	0	0	0
	10			0	
		Z			X

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

Or,

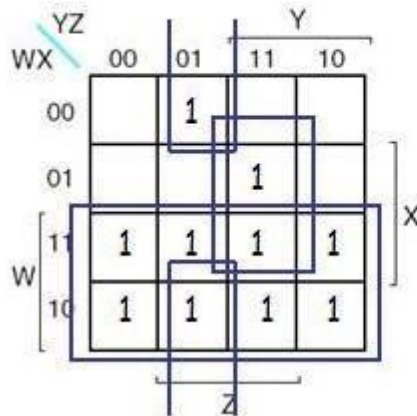
$$F(W,X,Y,Z) = X'Y' + X'Z' + W'Y'Z$$

Which one is the minimal one?

SoP Optimization from PoS

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

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



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

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

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

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

				y
	1	0	0	1
	1	1	x	0
	0	x	1	1
W	1	0	0	x
				Z

5-VARIABLE K-MAP

In above boolean table, from 0 to 15, A is 0 and from 16 to 31, A is 1. A 5-variable K-Map is drawn as below.

	A'					A			
	D'E'	D'E	DE	DE'		D'E'	D'E	DE	DE'
B'C'	0	1	3	2		16	17	19	18
B'C	4	5	7	6		20	21	23	22
BC	12	13	15	14		28	29	31	30
BC'	8	9	11	10		24	25	27	26

Again, as we did with 3-variable & 4-variable K-Map, carefully note the numbering of each cell. Now, we have two squares and we can loop octets, quads and pairs between these two squares. What we need to do is to visualize second square on first square and figure out adjacent cells. Let's understand how to simplify 5-variables K-Map by taking couple of examples.

Example 1 of 5-Variable K-Map

Given function, $F = \sum (1, 3, 4, 5, 11, 12, 14, 16, 20, 21, 30)$

Since, the biggest number is 30, we need to have 5 variables to define this function.

Let's draw K-Map for this function by writing 1 in cells that are present in function and 0 in rest of the cells.

Applying rules of simplifying K-Map, there is no octet. There is one quad that is obtained by visualizing second square on first, there are 4 adjacent cells – 4,5,20 and 21. The octet is highlighted by a blue connecting line. There are 5 pairs. Similar to quad, there is one pair between two squares which is highlighted by the blue connecting line.

(4, 5, 20, 21) – $B'CD'$ (Since A & E are the changing variables, it is eliminated)

(12, 14) – $A'BCE'$ (Since D is the changing variable, it is eliminated)

(14, 30) – $BCDE'$ (Since A is the changing variable, it is eliminated)

(3, 11) – $A'C'DE$ (Since B is the changing variable, it is eliminated)

(16, 20) – $AB'D'E'$ (Since C is the changing variable, it is eliminated)

(1, 3) – $A'B'C'E$ (Since D is the changing variable, it is eliminated)

Thus, $F = B'CD' + A'BCE' + BCDE' + A'C'DE + AB'D'E' + A'B'C'E$

Example 2 of 5-Variable K-Map

Given function, $F = \Sigma (0, 2, 3, 5, 7, 8, 11, 13, 17, 19, 23, 24, 29, 30)$

Since, the biggest number is 30, we need to have 5 variables to define this function.

Let's draw K-Map for this function by writing 1 in cells that are present in function and 0 in rest of the cells.

	A'				A			
	D'E'	D'E	DE	DE'	D'E'	D'E	DE	DE'
B'C'	0	1	3	2	16	17	19	18
	0	0	1	1	0	1	1	0
B'C	4	5	7	6	20	21	23	22
	0	1	1	0	0	0	1	0
BC	12	13	15	14	28	29	31	30
	0	1	0	0	0	1	0	1
BC'	8	9	11	10	24	25	27	26
	1	0	1	0	1	0	0	0

Applying rules of simplifying K-Map, there is no octet. First we need to look for quads within each of the squares. There are none but there is a quad between two squares that is obtained by visualizing second square on first, there are 4 adjacent cells – 3, 7, 19 and 23. This quad is highlighted by blue connecting line. There are 6 pairs, out of which two are between two squares, highlighted by blue connecting line.

(3, 7, 19, 23) - $B'DE$ (Since A & C are the changing variables, they are eliminated)

(3, 11) – $A'C'DE$ (Since B is the changing variables, it is eliminated)

(1, 2) – $A'B'C'E'$ (Since D is the changing variables, it is eliminated)

(5, 7) – $A'B'CE$ (Since D is the changing variables, it is eliminated)

(17, 19) – $AB'C'E$ (Since D is the changing variables, it is eliminated)

(13, 29) – $BCD'E$ (Since A is the changing variables, it is eliminated)

(8, 24) – $BC'D'E'$ (Since A is the changing variables, it is eliminated)

There is 1 in cell 30, which can not be looped with any adjacent cell, hence it can not be simplified further and left as it is.

30 – $ABCDE'$

Thus, $F = B'DE + A'C'DE + A'B'C'E' + A'B'CE + AB'C'E + BCD'E + BC'D'E' + ABCDE'$

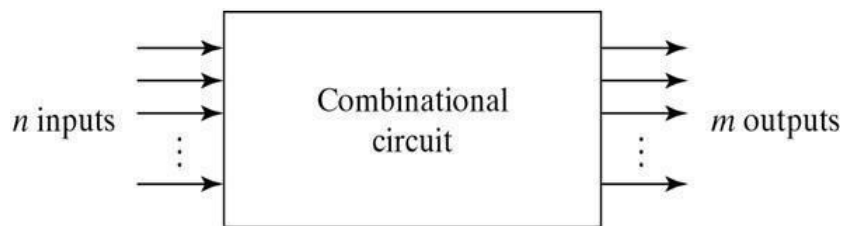
K-map Summary

- 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

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 variable. Usually the inputs come from flip-flops and outputs go to 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 letters/symbols.
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 a 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:

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

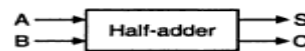
The first three operations produce a sum whose length is one digit, but when augends and addend bits are equal to 1, the binary sum consists of two digits. The higher significant bit of this result is

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

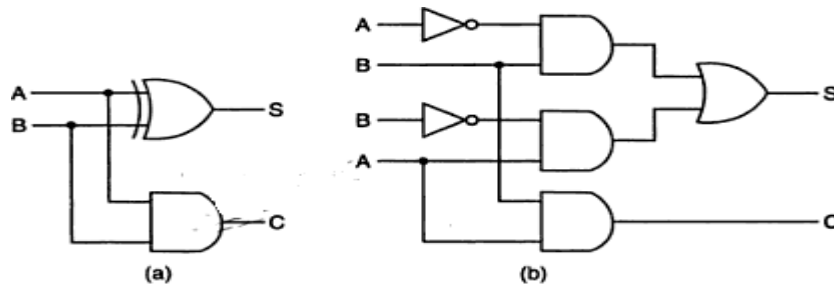
The Sum(S) bit and the carry (C) bit, according to the rules of binary addition, the sum (S) is the X-OR of A and B (It represents the LSB of the sum). Therefore,

$$S = A \oplus B$$

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

$$C = AB$$

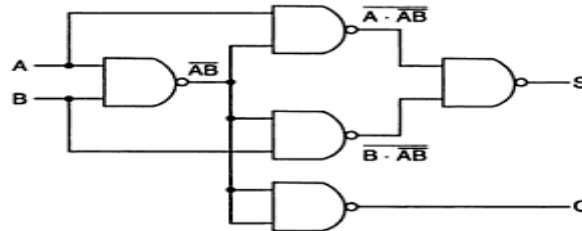
A half-adder can be realized by using one X-OR gate and one AND gate a



Logic diagrams of half-adder

NAND LOGIC:

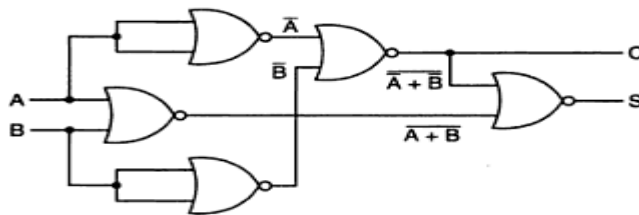
$$\begin{aligned}
 S &= A\bar{B} + \bar{A}B = A\bar{B} + A\bar{A} + \bar{A}B + B\bar{B} \\
 &= A(\bar{A} + B) + B(\bar{A} + \bar{B}) \\
 &= A \cdot \bar{A}B + B \cdot \bar{A}\bar{B} \\
 &= \overline{A \cdot \bar{A}B \cdot B \cdot \bar{A}\bar{B}} \\
 C &= AB = \overline{\bar{A}\bar{B}}
 \end{aligned}$$



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

NOR Logic:

$$\begin{aligned}
 S &= A\bar{B} + \bar{A}B = A\bar{B} + A\bar{A} + \bar{A}B + B\bar{B} \\
 &= A(\bar{A} + B) + B(\bar{A} + \bar{B}) \\
 &= (A + B)(\bar{A} + \bar{B}) \\
 &= \overline{\overline{A + B} \cdot \overline{\bar{A} + \bar{B}}} \\
 C &= AB = \overline{\bar{A}\bar{B}} = \overline{\bar{A} + \bar{B}}
 \end{aligned}$$



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

The Full Adder:

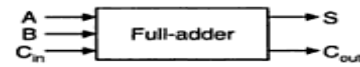
A Full-adder is a combinational circuit that adds two bits and a carry and outputs a sum bit and a carry bit. To add two binary numbers, each having two or more bits, the LSBs can be added by using a half-adder. The carry resulted from the addition of the LSBs is carried over to the next significant 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 C_{in} and outputs the sum bit S and the carry bit called the carry-out C_{out} . The variable S gives the value of the least significant bit of the sum. The variable C_{out} 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 C_{in} is described by

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

$$ABC_{in} \quad C_{out} = \overline{A}\overline{B}C_{in} + \overline{A}B\overline{C_{in}} +$$

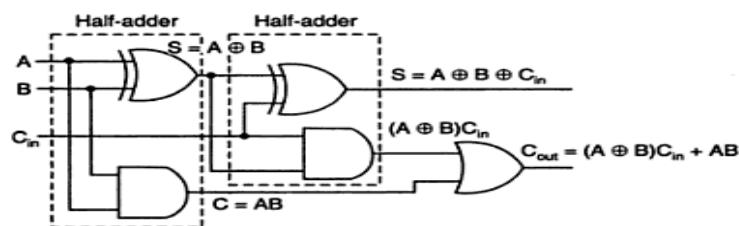
$$A\overline{B}\overline{C_{in}} + ABC_{in}$$

and

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

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

The sum term of the full-adder is the X-OR of A,B, and C_{in}, 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

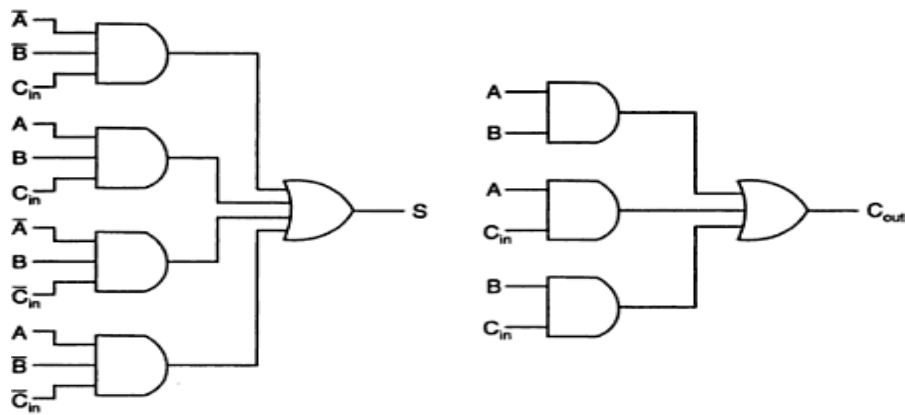
$$A \oplus B = \overline{\overline{A \cdot AB} \cdot \overline{B \cdot AB}}$$

Then

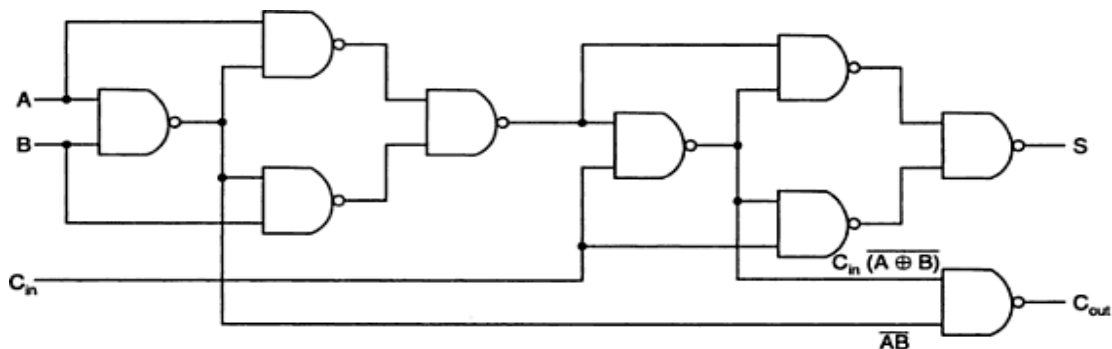
$$S = A \oplus B \oplus C_{in} = \overline{\overline{(A \oplus B) \cdot (A \oplus B)C_{in}} \cdot \overline{C_{in} \cdot (A \oplus B)C_{in}}}$$

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.

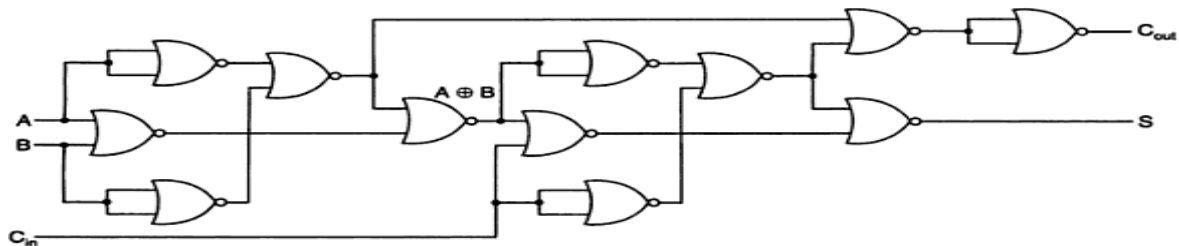
NOR Logic:

Then

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

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

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



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.

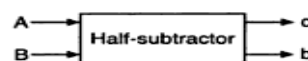
The Half-Subtractor:

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

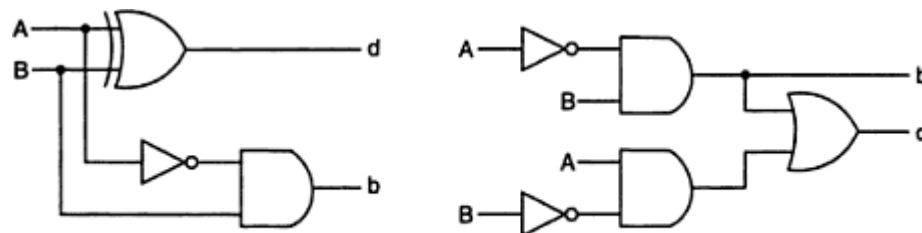
Half-subtractor.

The output borrow b is a 0 as long as $A \geq B$. 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,

$$d = A \oplus B \text{ and } b = \bar{A}B$$

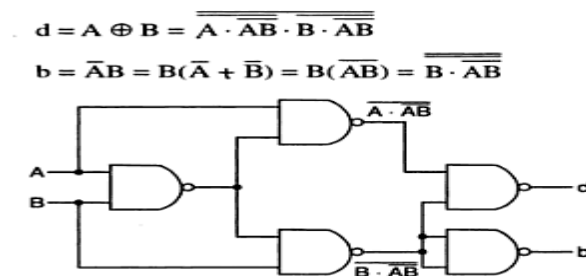
That is, the difference bit is obtained by X-OR ing the two inputs, and the borrow bit is obtained by ANDing the complement of the minuend with the subtrahend. Note that logic for this exactly the same as the logic for output S in the half-adder.



Logic diagrams of a half-subtractor.

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

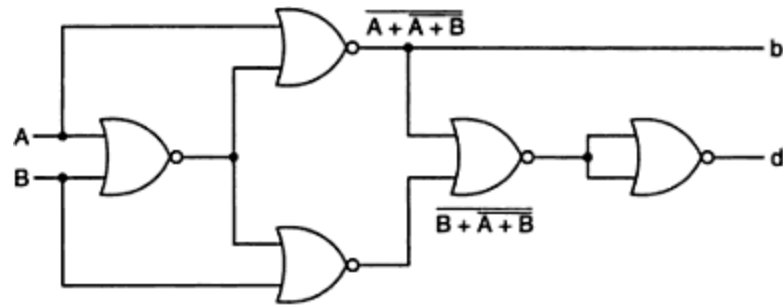
NAND Logic:



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

NOR Logic:

$$\begin{aligned} d &= A \oplus B = A\bar{B} + \bar{A}B = A\bar{B} + B\bar{B} + \bar{A}B + A\bar{A} \\ &= \bar{B}(A + B) + \bar{A}(A + B) = \overline{B + A + B + A + A + B} \\ d &= \bar{A}B = \bar{A}(A + B) = \overline{\overline{A}(A + B)} = A + (\overline{A + B}) \end{aligned}$$



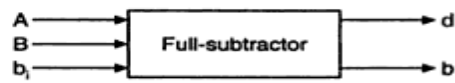
Logic diagram of a half-subtractor using only 2-input NOR gates.

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 b_i 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 $A - B - b_i$.

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.

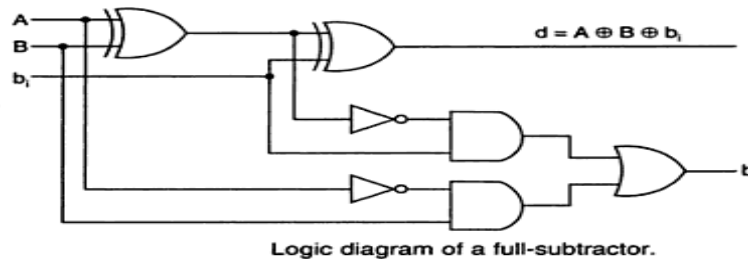
From the truth table, a circuit that will produce the correct difference and borrow bits in response to every possible combinations of A, B and b_i is

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

and

$$\begin{aligned}
 b &= \bar{A}\bar{B}b_i + \bar{A}B\bar{b}_i + \bar{A}Bb_i + ABb_i = \bar{A}B(b_i + \bar{b}_i) + (AB + \bar{A}\bar{B})b_i \\
 &= \bar{A}B + (A \oplus B)b_i
 \end{aligned}$$

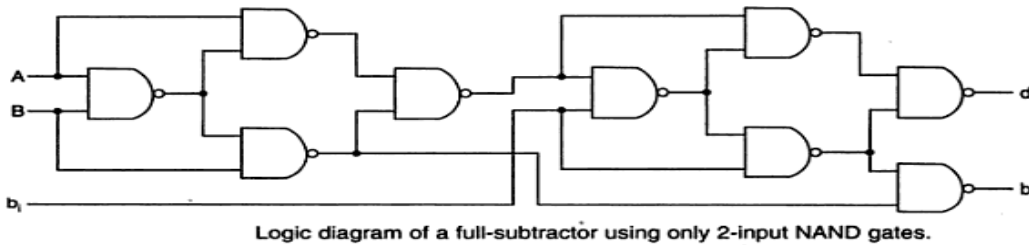
A full-subtractor can be realized using X-OR gates and AOI gates as



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

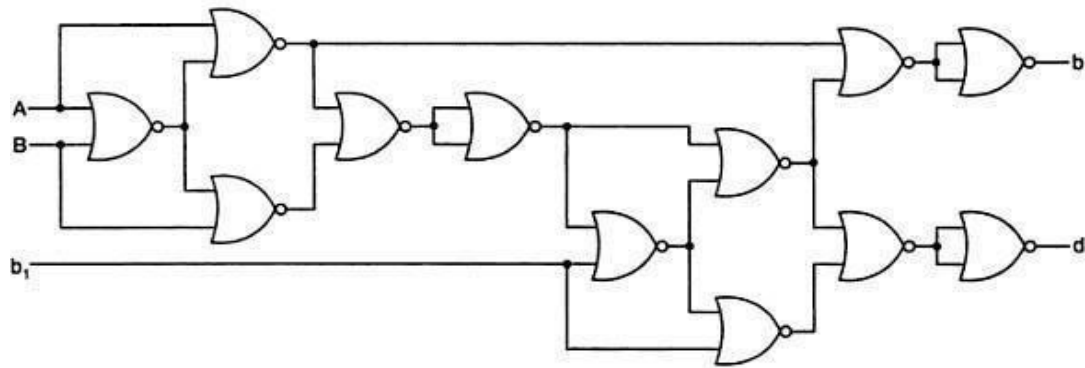
NAND Logic:

$$\begin{aligned}
 d &= A \oplus B \oplus b_i = \overline{\overline{(A \oplus B)} \oplus b_i} = \overline{(A \oplus B)(A \oplus B)b_i \cdot b_i(A \oplus B)b_i} \\
 b &= \overline{AB} + b_i(\overline{A \oplus B}) = \overline{AB} + b_i(\overline{A \oplus B}) \\
 &= \overline{AB} \cdot b_i(\overline{A \oplus B}) = \overline{B(\overline{A} + \overline{B}) \cdot b_i(\overline{b_i} + (A \oplus B))} \\
 &= \overline{B \cdot \overline{AB} \cdot b_i[b_i \cdot (A \oplus B)]}
 \end{aligned}$$



NOR Logic:

$$\begin{aligned}
 d &= A \oplus B \oplus b_i = \overline{\overline{(A \oplus B)} \oplus b_i} \\
 &= \overline{(A \oplus B)b_i + (A \oplus B)\overline{b_i}} \\
 &= \overline{[(A \oplus B) + (A \oplus B)\overline{b_i}][b_i + (A \oplus B)\overline{b_i}]} \\
 &= \overline{(A \oplus B) + (A \oplus B) + b_i + b_i + (A \oplus B) + b_i} \\
 &= \overline{(A \oplus B) + (A \oplus B) + b_i + b_i + (A \oplus B) + b_i} \\
 b &= \overline{AB} + b_i(\overline{A \oplus B}) \\
 &= \overline{A(A + B) + (A \oplus B)[(A \oplus B) + b_i]} \\
 &= \overline{A + (A + B) + (A \oplus B) + (A \oplus B) + b_i}
 \end{aligned}$$

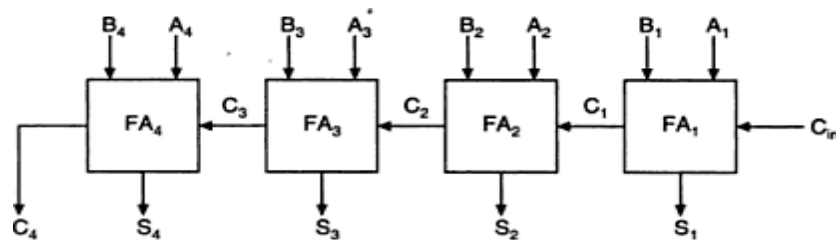


Logic diagram of a full subtractor using only 2-input NOR gates.

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 full-adders. The input carry to the adder is C_{in} and the output carry is C_4 . 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,

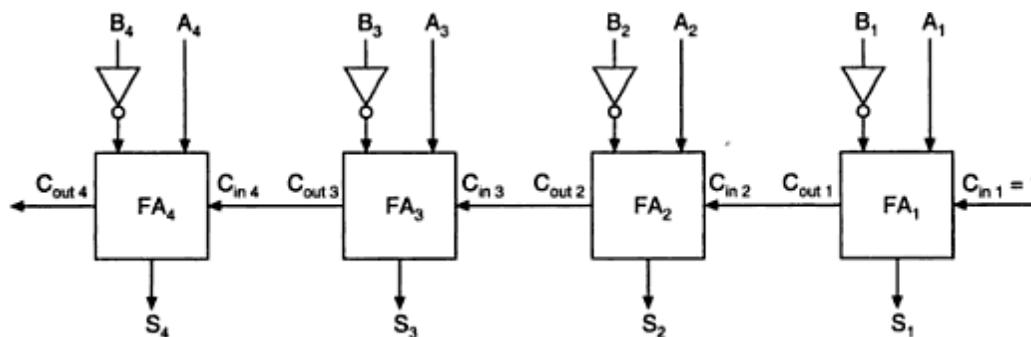
which lead to a time delay in the addition process. The carry propagation delay for each full-adder is the time between the application of the carry-in and the occurrence of the carry-out.

The 4-bit parallel adder, the sum (S_1) and carry-out (C_1) bits given by FA1 are not valid, until after the propagation delay of FA1. Similarly, the sum S_2 and carry-out (C_2) 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

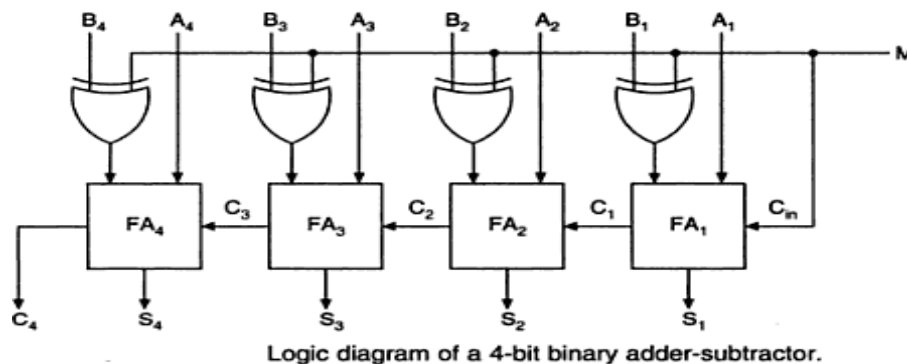


Logic diagram of a 4-bit parallel subtractor.

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 full-adder. 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 $C_1=1$. 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.

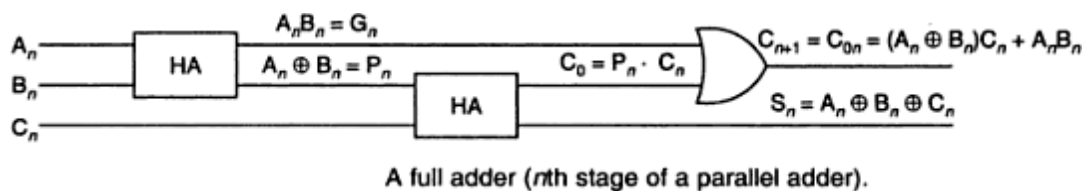


The Look-Ahead –Carry Adder:

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 n th 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 A_n and B_n are 1s, a carry has to be generated in this stage regardless of whether the input carry C_{in} is a 0 or a 1. This is called generated carry, expressed as $G_n = A_n \cdot B_n$ which has to appear at the output through the OR gate as shown in fig.



There is another possibility of producing a carry out. X-OR gate inside the half-adder at the input produces an intermediary sum bit- call it P_n –which is expressed as $P_n = A_n \oplus B_n$. Next P_n and C_n are added using the X-OR gate inside the second half adder to produce the final

sum bit and $S_n = P_n \oplus C_n$ where $P_n = A_n \oplus B_n$ and output carry $C_0 = P_n \cdot C_n = (A_n \oplus B_n) \cdot C_n$ which becomes carry for the (n+1) th stage.

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

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

$$S_n = P_n \oplus C_n \text{ where } P_n = A_n \oplus B_n$$

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

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,

$$C_1 = G_0 + P_0 \cdot C_0$$

$$C_2 = G_1 + P_1 \cdot C_1 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0$$

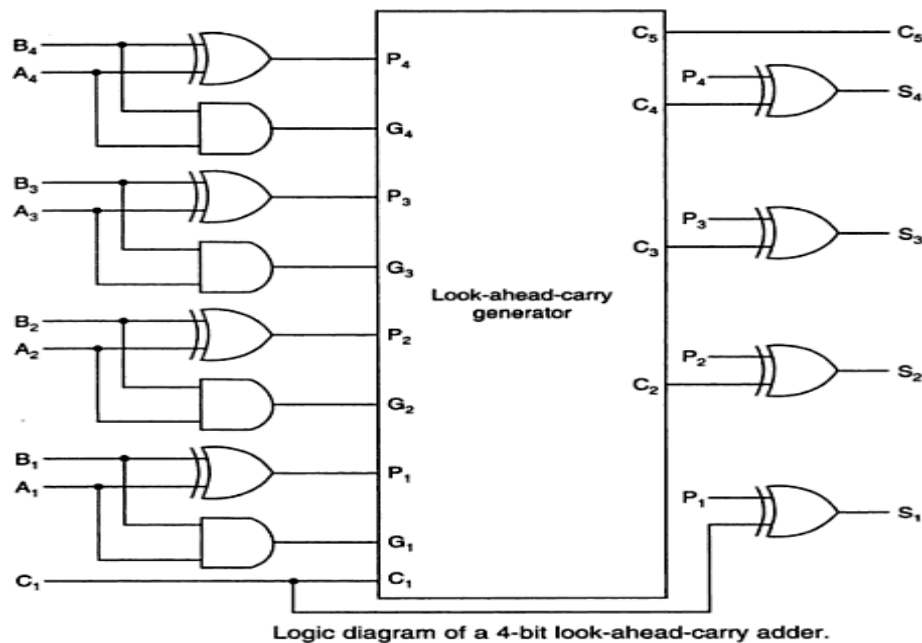
$$C_3 = G_2 + P_2 \cdot C_2 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0$$

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

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

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

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).



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 subtracts 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, B₂ and 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, B₂ and 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.

- 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 ends up stored in the register A. This is accomplished by applying a transfer pulse to the CLK inputs of register A.

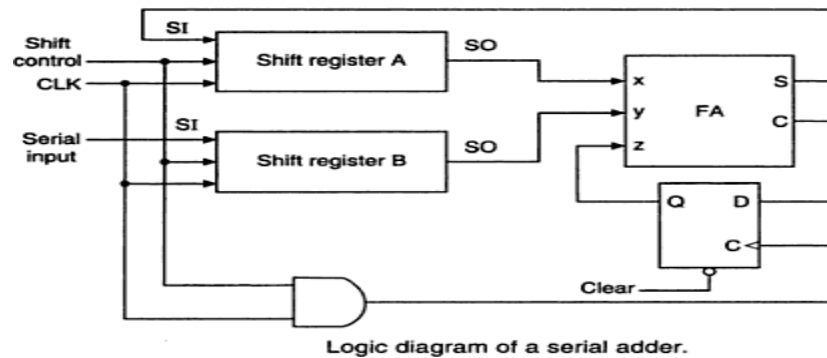
[illegible]

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.

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.



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 decimal position.

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 binary addition.
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 decimal position.

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 $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$ are applied to a 4-bit parallel adder, the adder will output $S_4S_3S_2S_1S_0$, where S_4 is actually C_4 , the carry-out of the MSB bits.

The sum outputs $S_4S_3S_2S_1S_0$ can range anywhere from 00000 to 100109 when 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

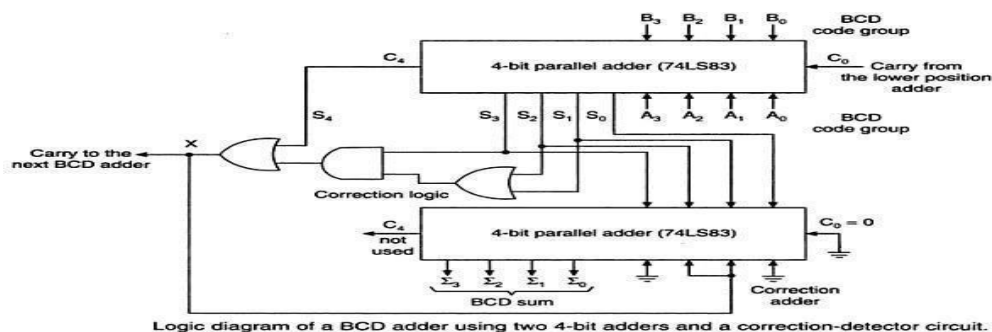
Let us define a logic output X that will go HIGH only when the sum is greater than 01001 (i.e., for the cases in table). If examine these cases, see that X will be HIGH for either of the following conditions:

1. Whenever $S_4 = 1$ (sum greater than 15)
 2. Whenever $S_3 = 1$ and either S_2 or S_1 or both are 1 (sum 10 to 15)
- This condition can be expressed as

$$X = S_4 + S_3(S_2 + S_1)$$

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 $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$ are added together in the upper 4-bit adder, to produce the sum $S_4S_3S_2S_1S_0$. 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 $\Sigma_3\Sigma_2\Sigma_1\Sigma_0$. 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, $\Sigma_3\Sigma_2\Sigma_1\Sigma_0 = S_3S_2S_1S_0$.

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 ($A_3A_2A_1A_0$) and addend ($B_3B_2B_1B_0$) 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 $S_3S_2S_1S_0$ 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 the minuend.
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 in 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 CODE CONVERTERS:

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 code 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.

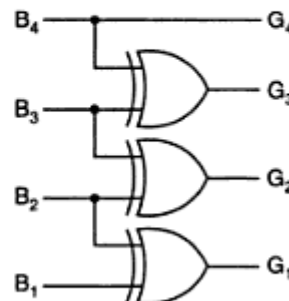
Design of a 4-bit binary to gray code converter:

$$G_4 = \sum m(8, 9, 10, 11, 12, 13, 14, 15)$$

4-bit binary				4-bit Gray			
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	0	0	0	1	0
0	0	1	1	0	0	1	1
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	1
0	1	1	0	0	1	1	0
0	1	1	1	0	1	1	1
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	1
1	0	1	0	1	0	1	0
1	0	1	1	1	0	1	1
1	1	0	0	1	1	0	0
1	1	0	1	1	1	0	1
1	1	1	0	1	1	1	0
1	1	1	1	1	1	1	1

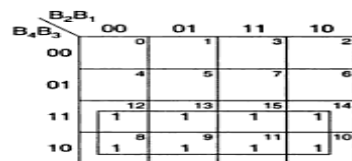
(a) Conversion table

$$G_i = B_i$$

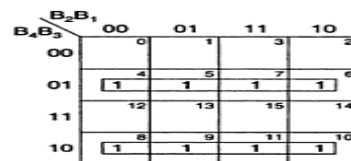


(c) Logic diagram

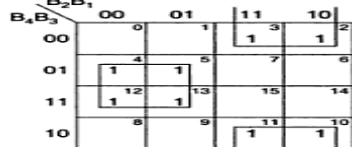
4-bit binary-to-Gray code converter



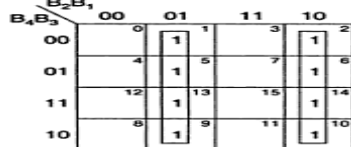
$G_4 = B_4$
K-map for G_4



$G_3 = B_4 \oplus B_3$
K-map for G_3



$G_2 = B_3 \oplus B_2$
K-map for G_2



$G_1 = B_2 \oplus B_1$
K-map for G_1

(b) K-maps

4-bit binary-to-Gray code converter.

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

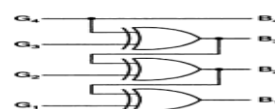
$$B_4 = \sum m(12, 13, 15, 14, 10, 11, 9, 8) = \sum m(8, 9, 10, 11, 12, 13, 14, 15)$$

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

(a) Conversion table



$B_4 = G_4$
K-map for B_4

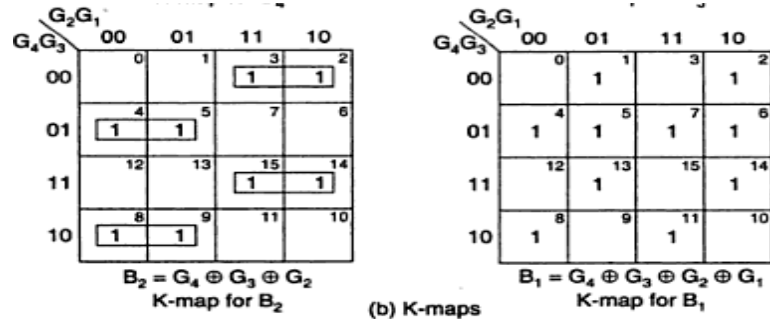


(c) Logic diagram



$$B_3 = G_4 \oplus G_3$$

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



4-bit Gray-to-binary code converter.

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

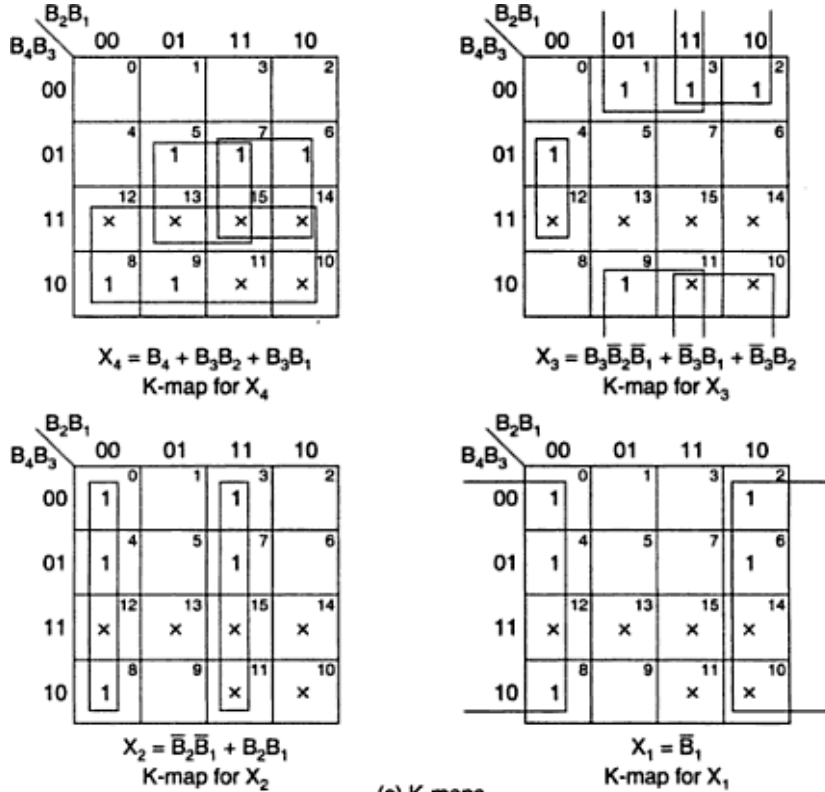
8421 code				XS-3 code			
B_4	B_3	B_2	B_1	X_4	X_3	X_2	X_1
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

(a) Conversion table

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

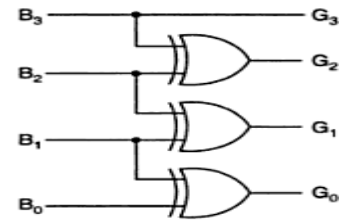


4-bit BCD-to-XS-3 code converter.

Design of a BCD to gray code converter:

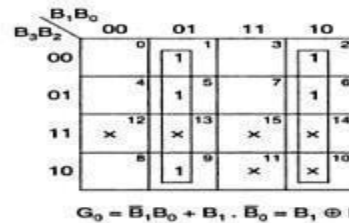
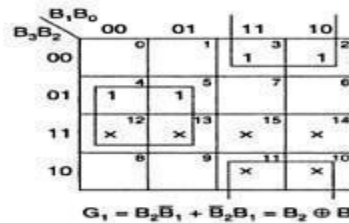
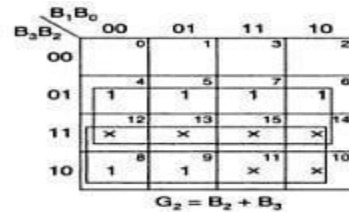
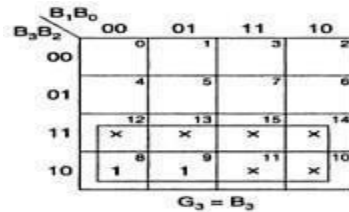
BCD code				Gray code			
B ₃	B ₂	B ₁	B ₀	G ₃	G ₂	G ₁	G ₀
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1

(a) BCD-to-Gray code conversion table



(b) Logic diagram

BCD-to-Gray code converter.



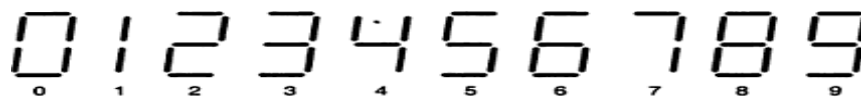
K-maps for a BCD-to-Gray code converter.

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

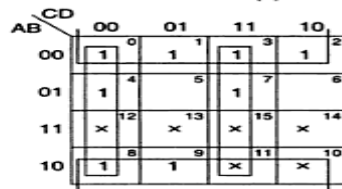
Input				Output			
A	B	C	D	E	F	G	H
0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1
0	0	1	0	1	1	1	0
0	0	1	1	1	1	0	1
0	1	0	0	1	1	0	0
0	1	0	1	1	0	1	1
0	1	1	0	1	0	1	0
0	1	1	1	1	0	0	1
1	0	0	0	1	0	0	0
1	0	0	1	0	1	1	1
1	0	1	0	0	1	1	0
1	0	1	1	0	1	0	1
1	1	0	0	0	1	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	0	1	0
1	1	1	1	0	0	0	1

(a) Conversion table

Conversion table and K-maps for the circuit

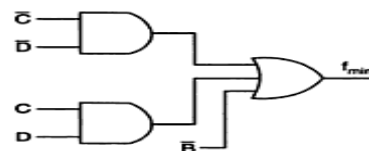


(a) Seven-segment display



$$f_{min} = B + \bar{C}\bar{D} + CD$$

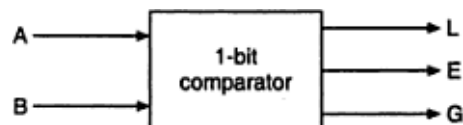
(b) K-map



(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,

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

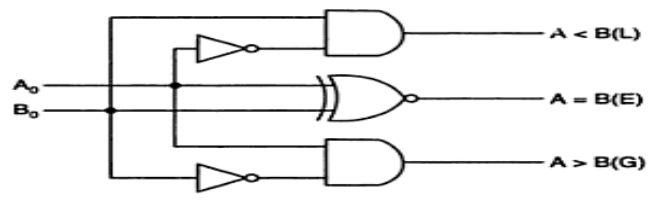
If A_0 and B_0 coincide, i.e. $A_0 = B_0 = 0$ or if $A_0 = B_0 = 1$, then $A = B$.

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

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

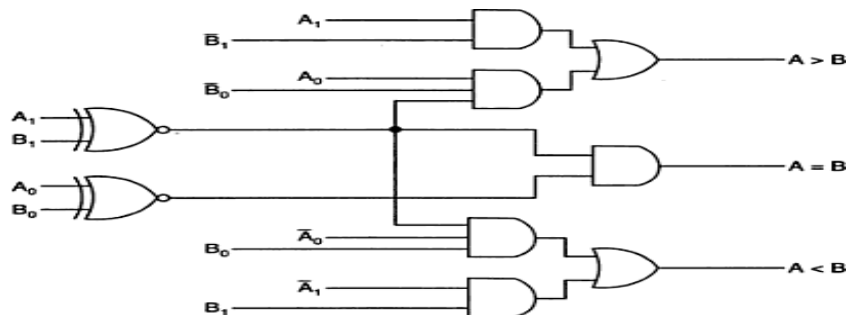
1. If $A_1 = 0$ and $B_1 = 1$, then $A < B$ or

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

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

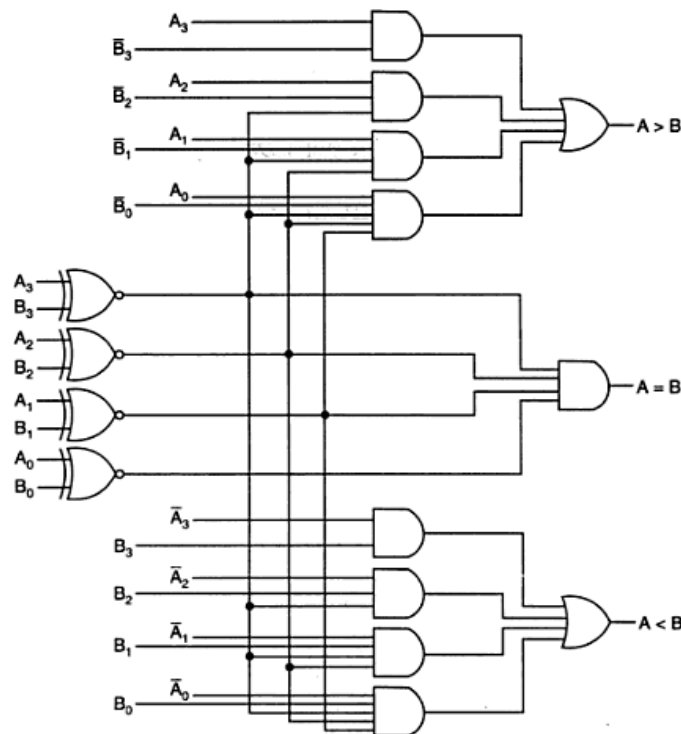
Similarly, the logic expression for $A < B$ can be written as

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

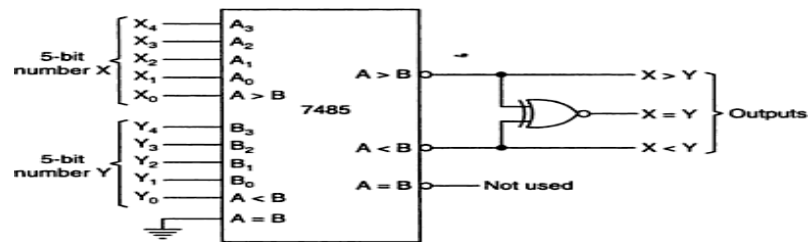
If A_3 and B_3 coincide and if A_2 and B_2 coincide and if A_1 and B_1 coincide and if A_0 and B_0 coincide, then $A = B$.

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

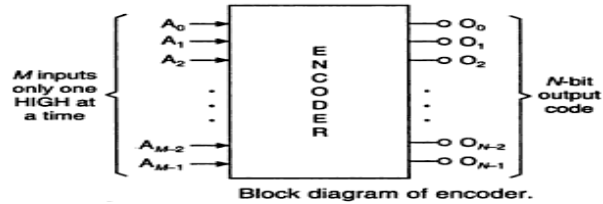
$$(A = B) = (A_3 \odot B_3)(A_2 \odot B_2)(A_1 \odot B_1)(A_0 \odot B_0)$$



ENCODERS:



Use of 7485 as a 5-bit comparator.

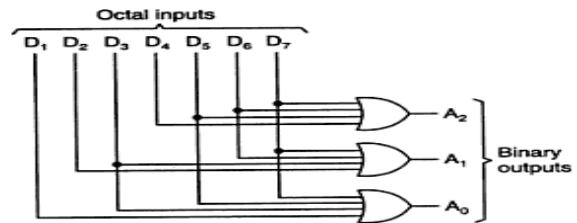


Block diagram of encoder.

Octal to Binary Encoder:

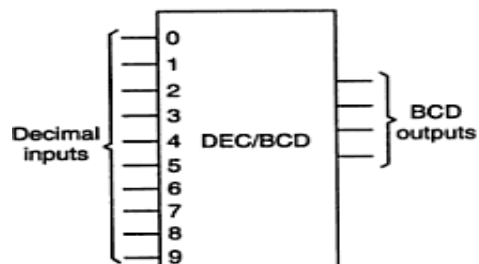
Octal digits		Binary		
		A ₂	A ₁	A ₀
D ₀	0	0	0	0
D ₁	1	0	0	1
D ₂	2	0	1	0
D ₃	3	0	1	1
D ₄	4	1	0	0
D ₅	5	1	0	1
D ₆	6	1	1	0
D ₇	7	1	1	1

(a) Truth table



(b) Logic diagram
Octal-to-binary encoder.

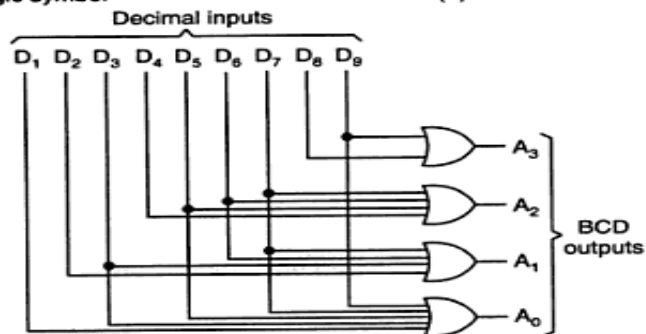
Decimal to BCD Encoder:



(a) Logic symbol

Decimal inputs		Binary			
		A ₃	A ₂	A ₁	A ₀
D ₀	0	0	0	0	0
D ₁	1	0	0	0	1
D ₂	2	0	0	1	0
D ₃	3	0	0	1	1
D ₄	4	0	1	0	0
D ₅	5	0	1	0	1
D ₆	6	0	1	1	0
D ₇	7	0	1	1	1
D ₈	8	1	0	0	0
D ₉	9	1	0	0	1

(b) Truth table



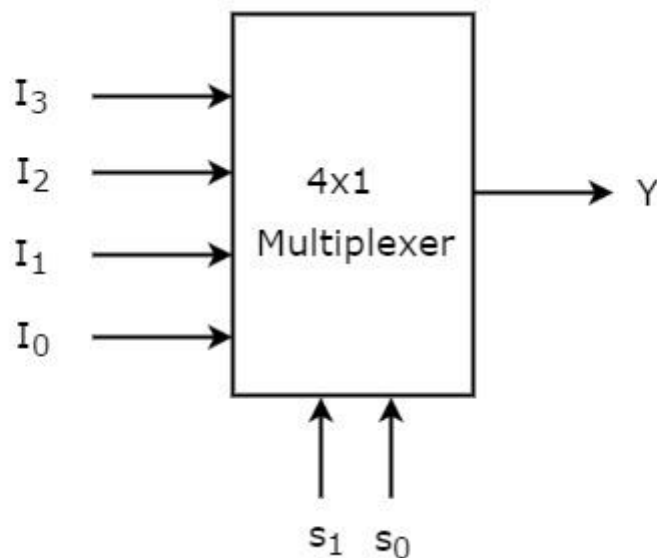
(c) Logic diagram
Decimal-to-BCD encoder.

Multiplexer is a combinational circuit that has maximum of 2^n data inputs, 'n' selection lines and single output line. One of these data inputs will be connected to the output based on the values of selection lines.

Since there are 'n' selection lines, there will be 2^n possible combinations of zeros and ones. So, each combination will select only one data input. Multiplexer is also called as **Mux**.

4x1 Multiplexer

4x1 Multiplexer has four data inputs I_3 , I_2 , I_1 & I_0 , two selection lines s_1 & s_0 and one output Y . The **block diagram** of 4x1 Multiplexer is shown in the following figure.



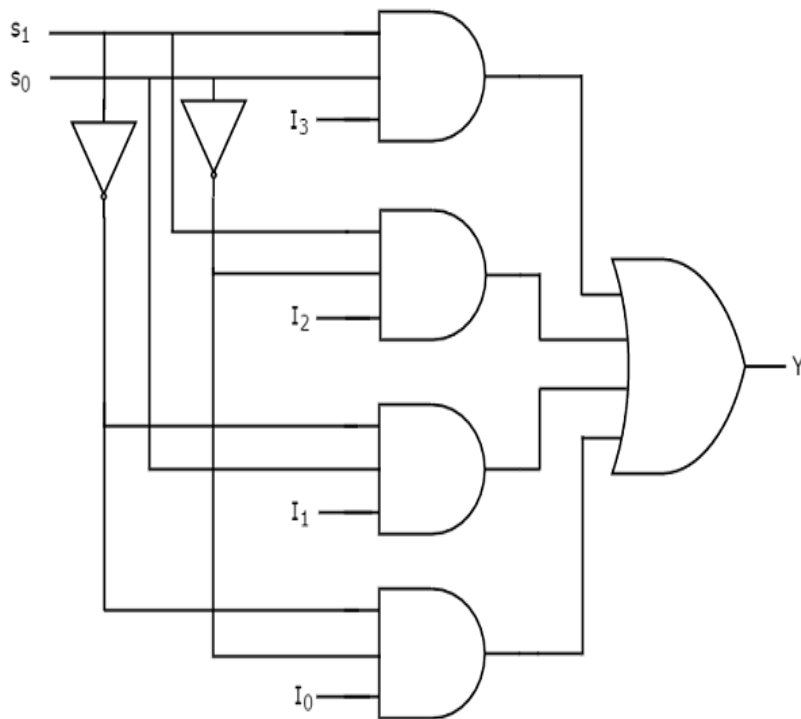
One of these 4 inputs will be connected to the output based on the combination of inputs present at these two selection lines. **Truth table** of 4x1 Multiplexer is shown below.

Selection Lines		Output
s_1	s_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

From Truth table, we can directly write the **Boolean function** for output, Y as

$$Y = S_1'S_0'I_0 + S_1'S_0I_1 + S_1S_0'I_2 + S_1S_0I_3$$

We can implement this Boolean function using Inverters, AND gates & OR gate. The **circuit diagram** of 4x1 multiplexer is shown in the following figure.



We can easily understand the operation of the above circuit. Similarly, you can implement 8x1 Multiplexer and 16x1 multiplexer by following the same procedure.

Implementation of Higher-order Multiplexers.

Now, let us implement the following two higher-order Multiplexers using lower-order Multiplexers.

- 8x1 Multiplexer
- 16x1 Multiplexer

8x1 Multiplexer:

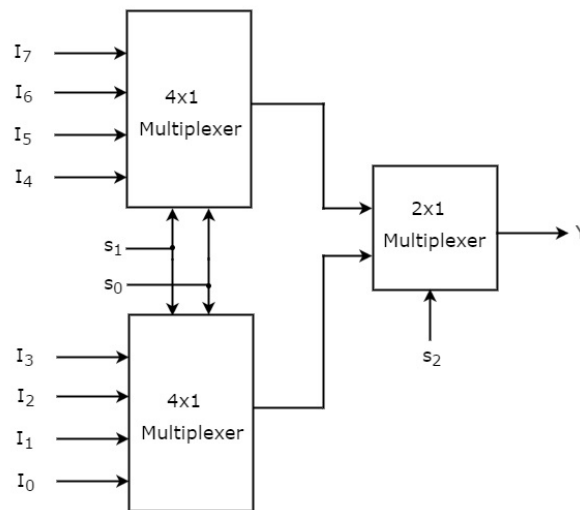
In this section, let us implement 8x1 Multiplexer using 4x1 Multiplexers and 2x1 Multiplexer. We know that 4x1 Multiplexer has 4 data inputs, 2 selection lines and one output. Whereas, 8x1 Multiplexer has 8 data inputs, 3 selection lines and one output.

So, we require two **4x1 Multiplexers** in first stage in order to get the 8 data inputs. Since, each 4x1 Multiplexer produces one output, we require a **2x1 Multiplexer** in second stage by considering the outputs of first stage as inputs and to produce the final output.

Let the 8x1 Multiplexer has eight data inputs I_7 to I_0 , three selection lines s_2 , s_1 & s_0 and one output Y. The **Truth table** of 8x1 Multiplexer is shown below.

Selection Inputs			Output
s_2	s_1	s_0	Y
0	0	0	I_0
0	0	1	I_1
0	1	0	I_2
0	1	1	I_3
1	0	0	I_4
1	0	1	I_5
1	1	0	I_6
1	1	1	I_7

We can implement 8x1 Multiplexer using lower order Multiplexers easily by considering the above Truth table. The **block diagram** of 8x1 Multiplexer is shown in the following figure.



The same **selection lines, s_1 & s_0** are applied to both 4x1 Multiplexers. The data inputs of upper 4x1 Multiplexer are I_7 to I_4 and the data inputs of lower 4x1 Multiplexer are I_3 to I_0 . Therefore, each 4x1 Multiplexer produces an output based on the values of selection lines, s_1 & s_0 .

The outputs of first stage 4x1 Multiplexers are applied as inputs of 2x1 Multiplexer that is present in second stage. The other **selection line, s_2** is applied to 2x1 Multiplexer.

- If s_2 is zero, then the output of 2x1 Multiplexer will be one of the 4 inputs I_3 to I_0 based on the values of selection lines s_1 & s_0 .
- If s_2 is one, then the output of 2x1 Multiplexer will be one of the 4 inputs I_7 to I_4 based on the values of selection lines s_1 & s_0 .

Therefore, the overall combination of two 4x1 Multiplexers and one 2x1 Multiplexer performs as one 8x1 Multiplexer.

16x1 Multiplexer:

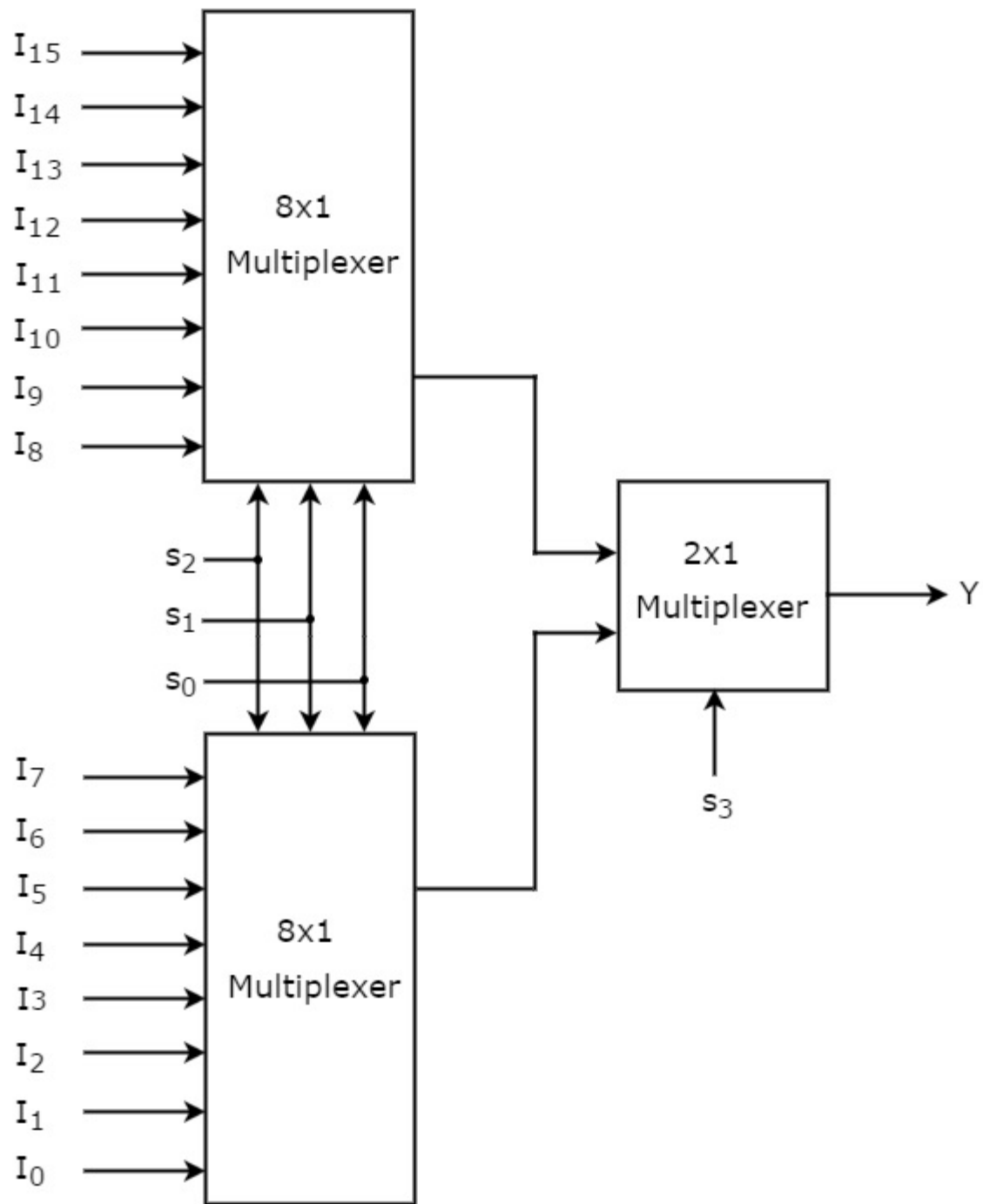
In this section, let us implement 16x1 Multiplexer using 8x1 Multiplexers and 2x1 Multiplexer. We know that 8x1 Multiplexer has 8 data inputs, 3 selection lines and one output. Whereas, 16x1 Multiplexer has 16 data inputs, 4 selection lines and one output.

So, we require two **8x1 Multiplexers** in first stage in order to get the 16 data inputs. Since, each 8x1 Multiplexer produces one output, we require a 2x1 Multiplexer in second stage by considering the outputs of first stage as inputs and to produce the final output.

Let the 16x1 Multiplexer has sixteen data inputs I_{15} to I_0 , four selection lines s_3 to s_0 and one output Y . The **Truth table** of 16x1 Multiplexer is shown below.

Selection Inputs				Output
s_3	s_2	s_1	s_0	Y
0	0	0	0	I_0
0	0	0	1	I_1
0	0	1	0	I_2
0	0	1	1	I_3
0	1	0	0	I_4
0	1	0	1	I_5
0	1	1	0	I_6
0	1	1	1	I_7
1	0	0	0	I_8
1	0	0	1	I_9
1	0	1	0	I_{10}
1	0	1	1	I_{11}
1	1	0	0	I_{12}
1	1	0	1	I_{13}
1	1	1	0	I_{14}
1	1	1	1	I_{15}

We can implement 16x1 Multiplexer using lower order Multiplexers easily by considering the above Truth table. The **block diagram** of 16x1 Multiplexer is shown in the following figure.



The **same selection lines, s_2 , s_1 & s_0** are applied to both 8x1 Multiplexers. The data inputs of upper 8x1 Multiplexer are I_{15} to I_8 and the data inputs of lower 8x1 Multiplexer are I_7 to I_0 . Therefore, each 8x1 Multiplexer produces an output based on the values of selection lines, s_2 , s_1 & s_0 .

The outputs of first stage 8x1 Multiplexers are applied as inputs of 2x1 Multiplexer that is present in second stage. The other **selection line, s_3** is applied to 2x1 Multiplexer.

- If s_3 is zero, then the output of 2x1 Multiplexer will be one of the 8 inputs I_7 to I_0 based on the values of selection lines s_2 , s_1 & s_0 .
- If s_3 is one, then the output of 2x1 Multiplexer will be one of the 8 inputs I_{15} to I_8 based on the values of selection lines s_2 , s_1 & s_0 .

Therefore, the overall combination of two 8x1 Multiplexers and one 2x1 Multiplexer performs as one 16x1 Multiplexer.

Applications of Multiplexer:

Multiplexer are used in various fields where multiple data need to be transmitted using a single line. Following are some of the applications of multiplexers -

1. **Communication system** – Communication system is a set of system that enable communication like transmission system, relay and tributary station, and communication network. The efficiency of communication system can be increased considerably using multiplexer. Multiplexer allow the process of transmitting different type of data such as audio, video at the same time using a single transmission line.
2. **Telephone network** – In telephone network, multiple audio signals are integrated on a single line for transmission with the help of multiplexers. In this way, multiple audio signals can be isolated and eventually, the desire audio signals reach the intended recipients.
3. **Computer memory** - Multiplexers are used to implement huge amount of memory into the computer, at the same time reduces the number of copper lines required to connect the memory to other parts of the computer circuit.
4. **Transmission from the computer system of a satellite** – Multiplexer can be used for the transmission of data signals from the computer system of a satellite or spacecraft to the ground system using the GPS (Global Positioning System) satellites.

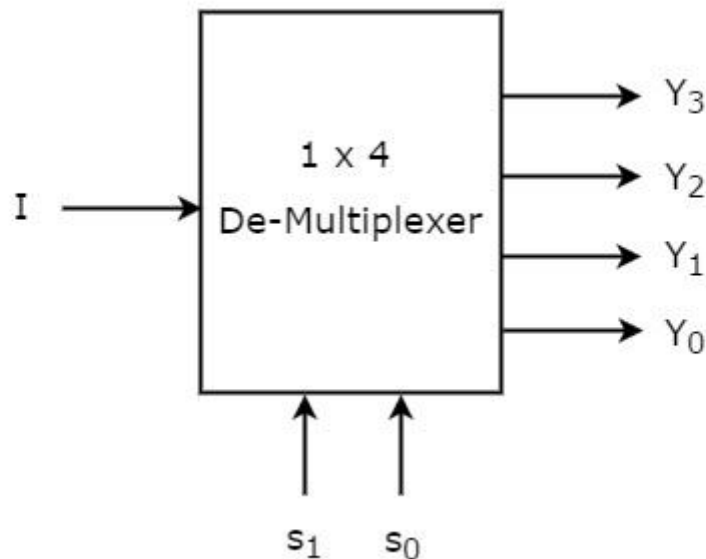
De-Multiplexer

De-Multiplexer is a combinational circuit that performs the reverse operation of Multiplexer. It has single input, 'n' selection lines and maximum of 2^n outputs. The input will be connected to one of these outputs based on the values of selection lines.

Since there are 'n' selection lines, there will be 2^n possible combinations of zeros and ones. So, each combination can select only one output. De-Multiplexer is also called as **De-Mux**.

1x4 De-Multiplexer

1x4 De-Multiplexer has one input I, two selection lines, s_1 & s_0 and four outputs Y_3 , Y_2 , Y_1 & Y_0 . The **block diagram** of 1x4 De-Multiplexer is shown in the following figure.



The single input 'I' will be connected to one of the four outputs, Y_3 to Y_0 based on the values of selection lines s_1 & s_0 . The **Truth table** of 1x4 De-Multiplexer is shown below.

From the above Truth table, we can directly write the **Boolean functions** for each output as

Selection Inputs		Outputs			
s_1	s_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	I
0	1	0	0	I	0
1	0	0	I	0	0
1	1	I	0	0	0

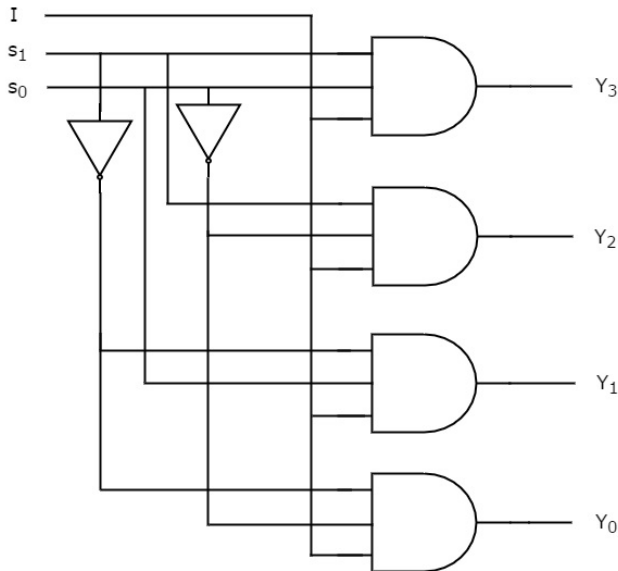
$$Y_3 = s_1 s_0 I$$

$$Y_2 = s_1 s_0 I'$$

$$Y_1 = s_1 s_0' I$$

$$Y_0 = s_1' s_0 I'$$

We can implement these Boolean functions using Inverters & 3-input AND gates. The **circuit diagram** of 1x4 De-Multiplexer is shown in the following figure.



We can easily understand the operation of the above circuit. Similarly, you can implement 1x8 De-Multiplexer and 1x16 De-Multiplexer by following the same procedure.

Implementation of Higher-order De-Multiplexers

Now, let us implement the following two higher-order De-Multiplexers using lower-order De-Multiplexers.

- 1x8 De-Multiplexer
- 1x16 De-Multiplexer

1x8 De-Multiplexer

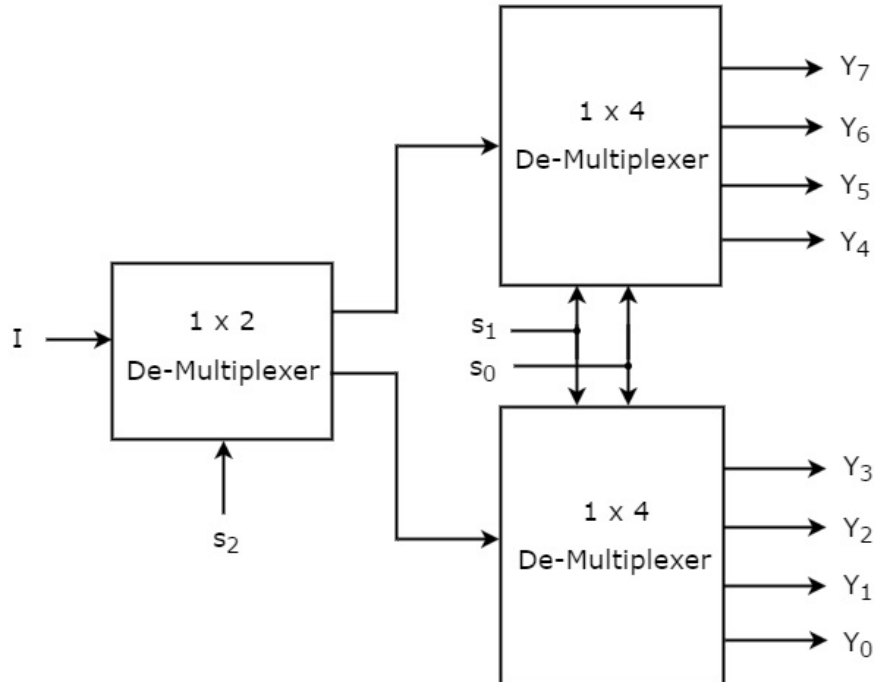
In this section, let us implement 1x8 De-Multiplexer using 1x4 De-Multiplexers and 1x2 De-Multiplexer. We know that 1x4 De-Multiplexer has single input, two selection lines and four outputs. Whereas, 1x8 De-Multiplexer has single input, three selection lines and eight outputs.

So, we require two **1x4 De-Multiplexers** in second stage in order to get the final eight outputs. Since, the number of inputs in second stage is two, we require **1x2 DeMultiplexer** in first stage so that the outputs of first stage will be the inputs of second stage. Input of this 1x2 De-Multiplexer will be the overall input of 1x8 De-Multiplexer.

Let the 1x8 De-Multiplexer has one input I , three selection lines s_2 , s_1 & s_0 and outputs Y_7 to Y_0 . The **Truth table** of 1x8 De-Multiplexer is shown below.

Selection Inputs			Outputs							
s_2	s_1	s_0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0	0	0	I
0	0	1	0	0	0	0	0	0	I	0
0	1	0	0	0	0	0	0	I	0	0
0	1	1	0	0	0	0	I	0	0	0
1	0	0	0	0	0	I	0	0	0	0
1	0	1	0	0	I	0	0	0	0	0
1	1	0	0	I	0	0	0	0	0	0
1	1	1	I	0	0	0	0	0	0	0

We can implement 1x8 De-Multiplexer using lower order Multiplexers easily by considering the above Truth table. The **block diagram** of 1x8 De-Multiplexer is shown in the following figure.



The common **selection lines**, s_1 & s_0 are applied to both 1x4 De-Multiplexers. The outputs of upper 1x4 De-Multiplexer are Y_7 to Y_4 and the outputs of lower 1x4 De-Multiplexer are Y_3 to Y_0 .

The other **selection line**, s_2 is applied to 1x2 De-Multiplexer. If s_2 is zero, then one of the four outputs of lower 1x4 De-Multiplexer will be equal to input, I based on the values of selection lines

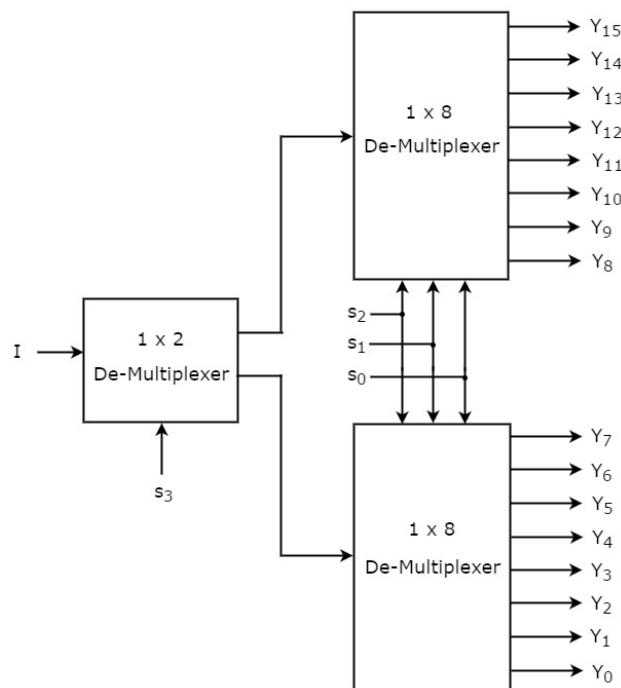
s_1 & s_0 . Similarly, if s_2 is one, then one of the four outputs of upper 1x4 DeMultiplexer will be equal to input, I based on the values of selection lines s_1 & s_0 .

1x16 De-Multiplexer

In this section, let us implement 1x16 De-Multiplexer using 1x8 De-Multiplexers and 1x2 De-Multiplexer. We know that 1x8 De-Multiplexer has single input, three selection lines and eight outputs. Whereas, 1x16 De-Multiplexer has single input, four selection lines and sixteen outputs.

So, we require two **1x8 De-Multiplexers** in second stage in order to get the final sixteen outputs. Since, the number of inputs in second stage is two, we require **1x2 DeMultiplexer** in first stage so that the outputs of first stage will be the inputs of second stage. Input of this 1x2 De-Multiplexer will be the overall input of **1x16 De-Multiplexer**.

Let the 1x16 De-Multiplexer has one input I, four selection lines s_3, s_2, s_1 & s_0 and outputs Y_{15} to Y_0 . The **block diagram** of 1x16 De-Multiplexer using lower order Multiplexers is shown in the following figure.



The common **selection lines** s_2, s_1 & s_0 are applied to both 1x8 De-Multiplexers. The outputs of upper 1x8 De-Multiplexer are Y_{15} to Y_8 and the outputs of lower 1x8 De-Multiplexer are Y_7 to Y_0 .

The other **selection line**, s_3 is applied to 1x2 De-Multiplexer. If s_3 is zero, then one of the eight outputs of lower 1x8 De-Multiplexer will be equal to input, I based on the values of selection lines s_2, s_1 & s_0 . Similarly, if s_3 is one, then one of the 8 outputs of upper 1x8 De-Multiplexer will be equal to input, I based on the values of selection lines s_2, s_1 & s_0 .

Applications of Demultiplexer:

1. Demultiplexer is used to connect a single source to multiple destinations. The main application area of demultiplexer is communication system where multiplexer are used. Most of the communication system are bidirectional i.e. they function in both ways (transmitting and receiving signals). Hence, for most of the applications, the multiplexer and demultiplexer work in sync. Demultiplexer is also used for reconstruction of parallel data and ALU circuits.
2. **Communication System** - Communication system use multiplexer to carry multiple data like audio, video and other form of data using a single line for transmission. This process makes the transmission easier. The demultiplexer receives the output signals of the multiplexer and converts them back to the original form of the data at the receiving end. The multiplexer and demultiplexer work together to carry out the process of transmission and reception of data in communication system.
3. **ALU (Arithmetic Logic Unit)** – In an ALU circuit, the output of ALU can be stored in multiple registers or storage units with the help of demultiplexer. The output of ALU is fed as the data input to the demultiplexer. Each output of demultiplexer is connected to multiple register which can be stored in the registers.
4. **Serial to parallel converter** - A serial to parallel converter is used for reconstructing parallel data from incoming serial data stream. In this technique, serial data from the incoming serial data stream is given as data input to the demultiplexer at the regular intervals. A counter is attaching to the control input of the demultiplexer. This counter directs the data signal to the output of the demultiplexer where these data signals are stored. When all data signals have been stored, the output of the demultiplexer can be retrieved and read out in parallel.

HAZARDS IN COMBINATIONAL LOGIC CIRCUITS:

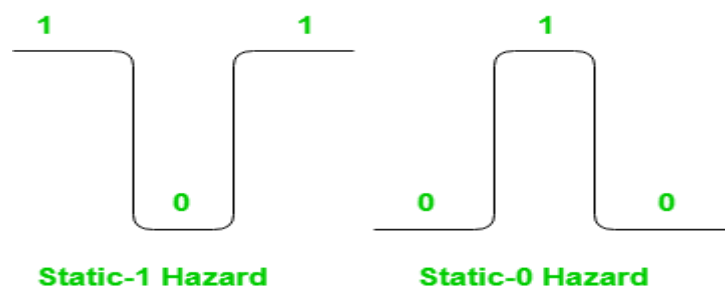
A hazard, if exists, in a digital circuit causes a temporary fluctuation in output of the circuit. In other words, a hazard in a digital circuit is a temporary disturbance in ideal operation of the circuit which if given some time, gets resolved itself. These disturbances or fluctuations occur when different paths from the input to output have different delays and due to this fact, changes in input variables do not change the output instantly but do appear at output after a small delay caused by the circuit building elements, i.e., logic gates. There are three different kinds of hazards found in digital circuits

1. Static hazard
2. Dynamic hazard
3. Functional hazard

We will discuss only static hazards here to understand it completely.

Formally, a static hazard takes place when change in an input causes the output to change momentarily before stabilizing to its correct value. Based on what is the correct value, there are two types of static hazards, as shown below in the image:

1. **Static-1 Hazard:** If the output is currently at logic state 1 and after the input changes its state, the output momentarily changes to 0 before settling on 1, then it is a Static-1 hazard.
2. **Static-0 Hazard:** If the output is currently at logic state 0 and after the input changes its state, the output momentarily changes to 1 before settling on 0, then it is a Static-0 hazard.



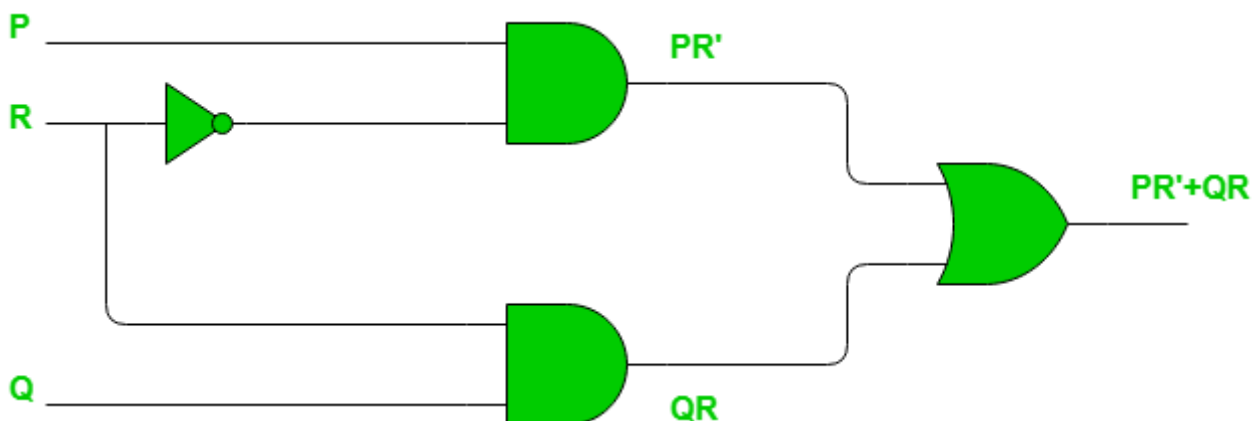
Detection of Static hazards using K-map:

Lets consider static-1 hazard first. To detect a static-1 hazard for a digital circuit following steps are used:

- **Step-1:** Write down the output of the digital circuit, say **Y**.
- **Step-2:** Draw the K-map for this function **Y** and note all adjacent 1's.
- **Step-3:** If there exists any pair of cells with 1's which do not occur to be in the same group (i.e. prime implicant), it indicates the presence of a static-1 hazard. Each such pair is a static-1 hazard.

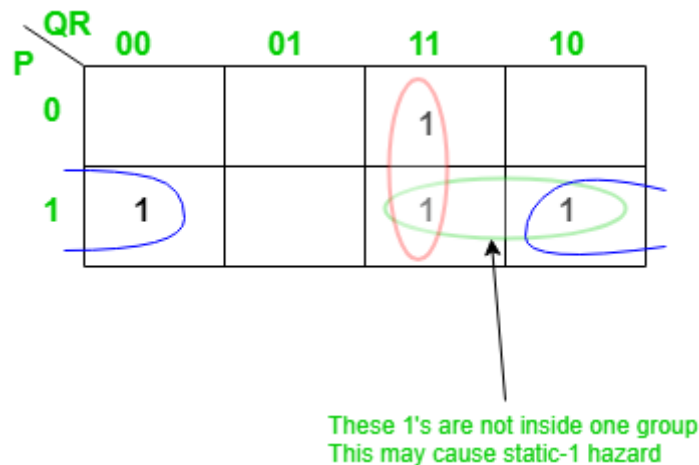
Lets have an example:

Example – Consider the circuit shown below.



We have output, say F, as:

Lets draw the K-map for this Boolean function as follows:

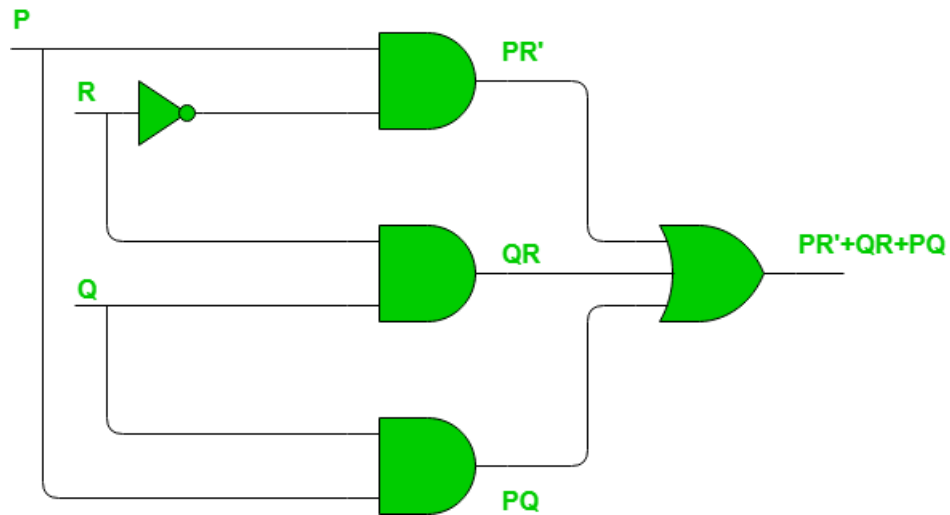


The pair of 1's encircled as green are not part of the grouping/pairing provided by the output of this Boolean function. This will cause a static-1 hazard in this circuit.

Removal of static-1 hazard:

Once detected, a static-1 hazard can be easily removed by introducing some more terms (logic gates) to the function (circuit). The most common idea is to add the missing group in the existing Boolean function, as adding this term would not affect the function by any mean but it will remove the hazard. Since in above example the pair of 1's encircled with blue color causes the static-1 hazard, we just add this as a prime implicant to the existing function as follows:

Note that there is no difference in number of minterms of this function. The reason is that the static-1 hazards are based on how we group 1's (or 0's for static-0 hazard) for a given set of 1's in K-map. Thus it does not make any difference in number of 1's in K-map. The circuit would look like as shown below with the change made for removal of static-1 hazard.



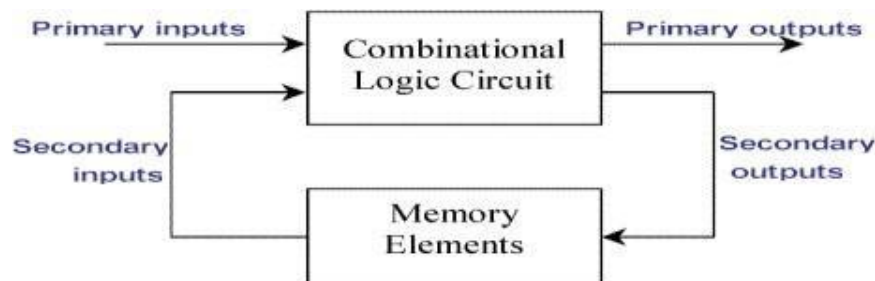
Similarly for **Static-0 Hazards** we need to consider 0's instead of 1's and if any adjacent 0's in K-map are not grouped into same group that may cause a static-0 hazard. The method to detect and resolve the static-0 hazard is completely same as the one we followed for static-1 hazard except that instead of SOP, POS will be used as we are dealing with 0's in this case.

MODULE-III
SEQUENTIAL CIRCUITS

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

1. Synchronous sequential circuits
2. Asynchronous sequential circuits

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:

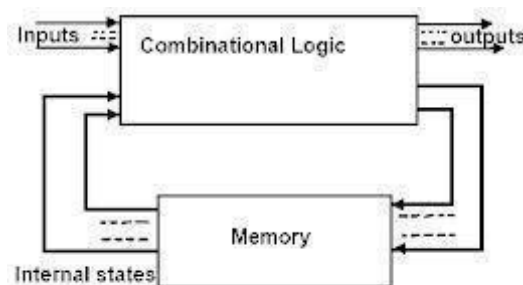
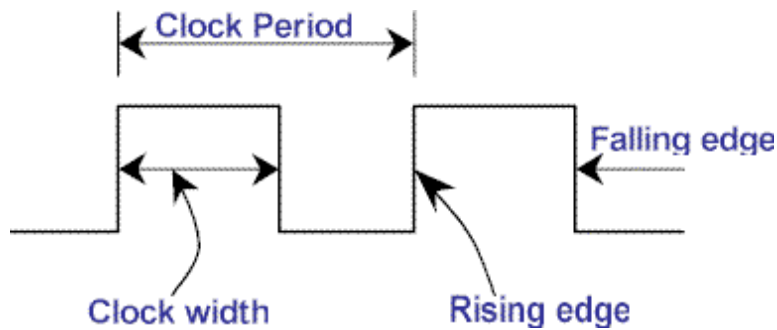


Figure 1: Asynchronous Sequential Circuit

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 theinput
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 a latch, their content changes immediately when their inputs change. In a flip-flop, their content change only either at the rising or falling edge of the clock signal. After the rising or falling edge of the clock, the flip-flop content remains constant even if the input changes.

S	R	Q	Q'	Function
0	0	Q ⁺	Q ⁺	Storage State
0	1	0	1	Reset
1	0	1	0	Set
1	1	0-?	0-?	Indeterminate State

ed, their content changes immediately when their inputs have their content change only either at the rising or falling 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.

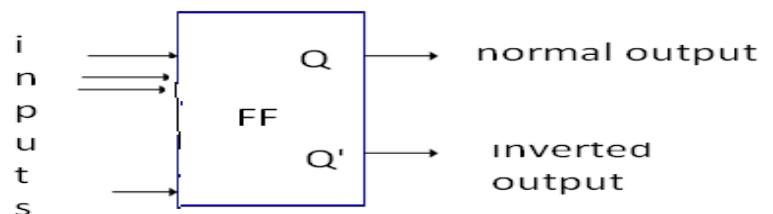


Figure: basic symbol of flipflop

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.

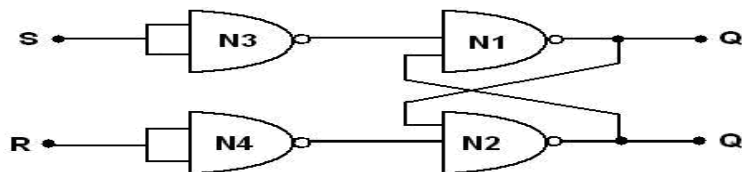
SR latch:

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

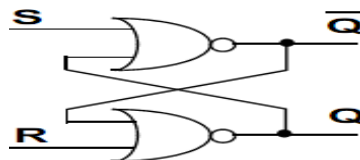
NAND latch

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 operation 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



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 state 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 state 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 $N1$ and $N2$ 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

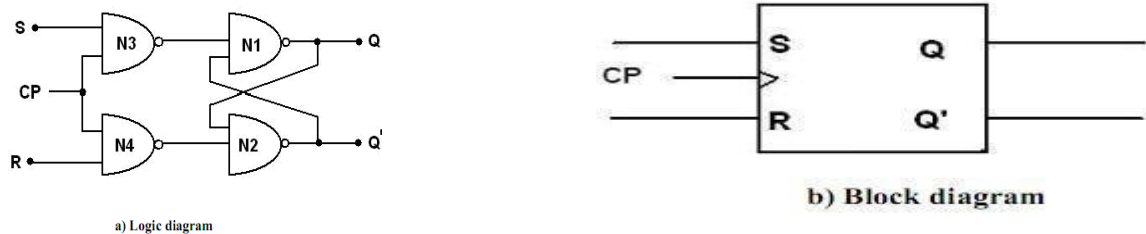


Figure: RS Flip-flop

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 of flip-flop.
6. $CP=1, S=1, R=1$ is not allowed, because it is not able to determine the next state. This condition 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.

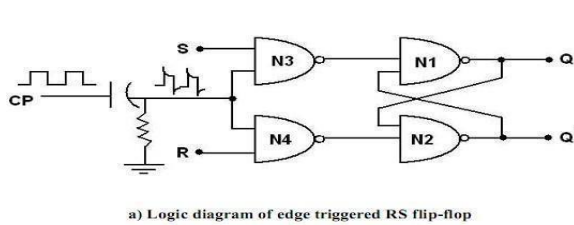
Truth table

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

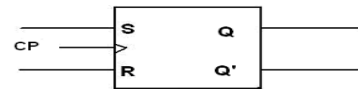
Edge triggered RS flip-flop:

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



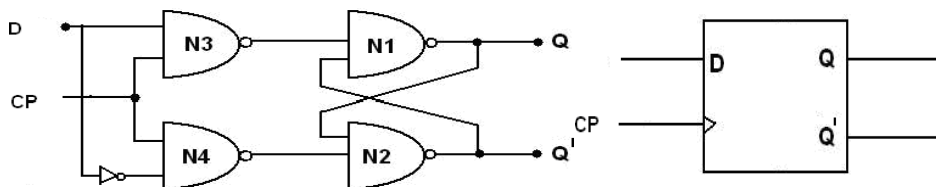
b) Block diagram of positive edge triggered flip-flop



c) Block diagram of negative edge triggered 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

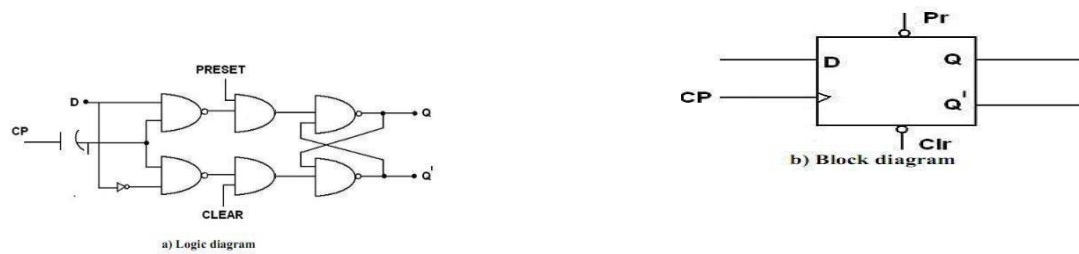
b) Block diagram

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.

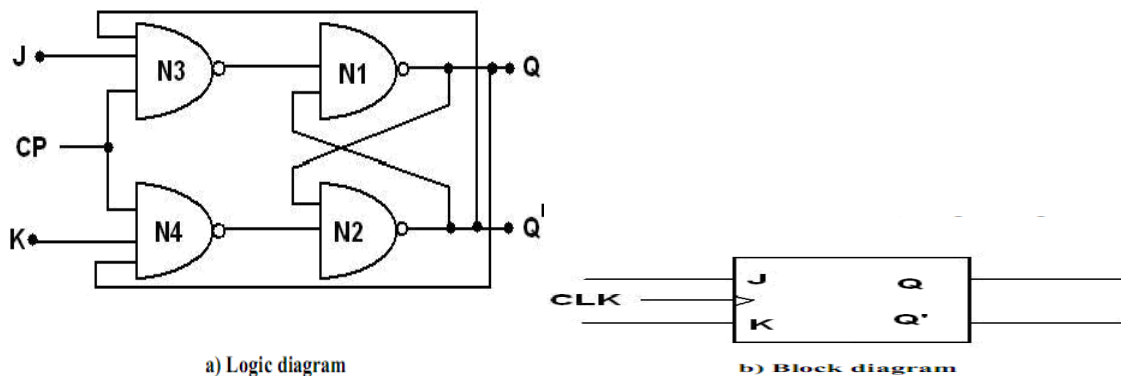


Figure: JK flip-flop

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.

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									
<u>Characteristic table</u>				<u>Excitation table</u>					
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

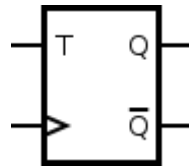


Figure : symbol for T flip flop

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

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 Previous is connected to the D input through an XOR gate).

T flip-flop operation ^[28]							
<u>Characteristic table</u>				<u>Excitation table</u>			
<i>T</i>	<i>Q</i>	<i>Q_{next}</i>	Comment	<i>Q</i>	<i>Q_{next}</i>	<i>T</i>	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**

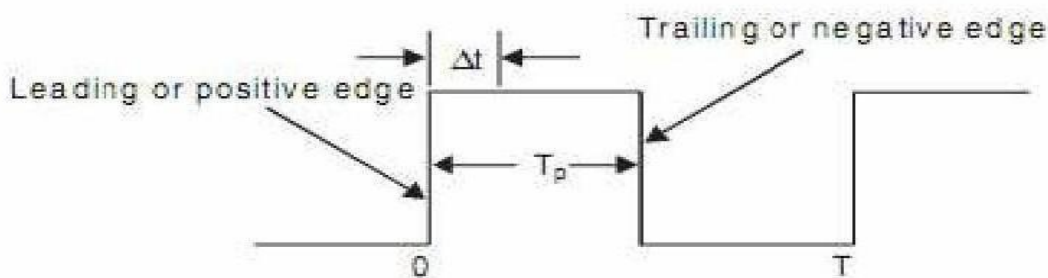
Dissipation: is the total power consumption of the device. It is equal to product of supply voltage (V_{cc}) and the current (I_{cc}).

$$P = V_{cc} \cdot I_{cc}$$

The power dissipation of a flip flop is usually in mW

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 $t_p < t < T$. 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:

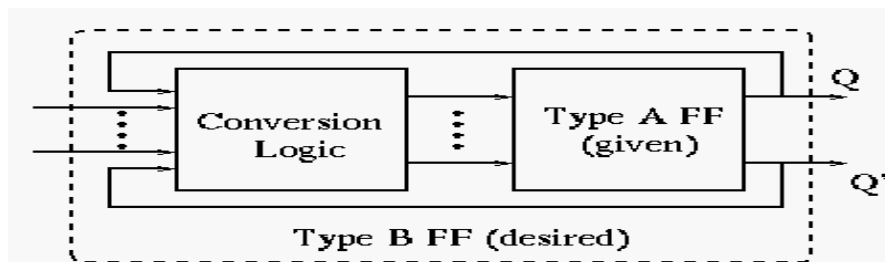
Previous State -> Present State	D
0 -> 0	0
0 -> 1	1
1 -> 0	0
1 -> 1	1

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

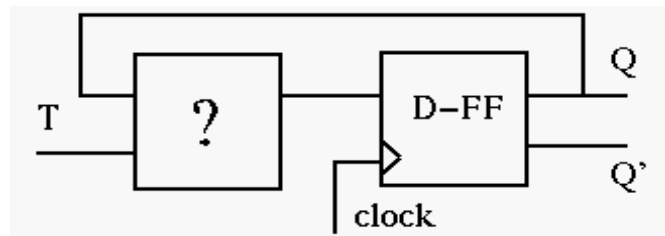
Conversions of flip-flops:



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:

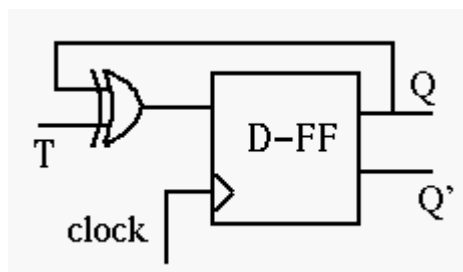


We need to design the circuit to generate the triggering signal D as a function of T and Q:
 . Consider the excitation table:

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

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

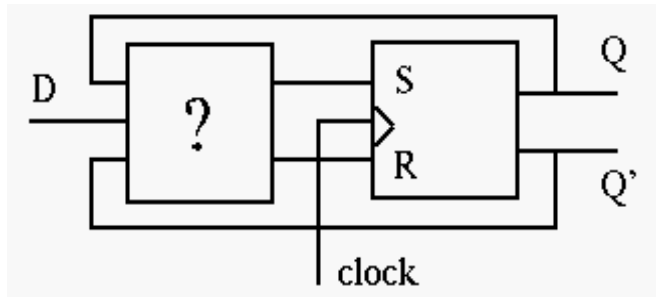
Treating as a function of and current FF state, we have



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

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 D 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 D and current FF state from the Karnaugh maps:

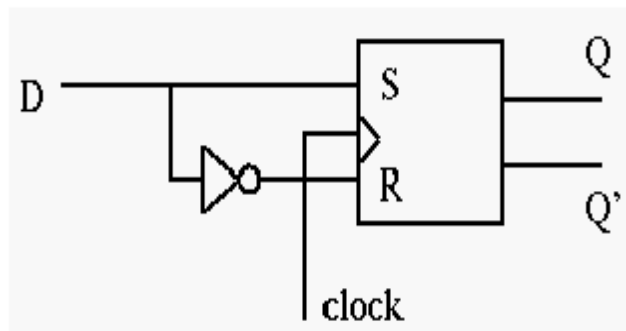
D \ Q	0	1
0	0	0
1	1	X

$$S = D$$

D \ Q	0	1
0	X	1
1	0	0

$$R = D'$$

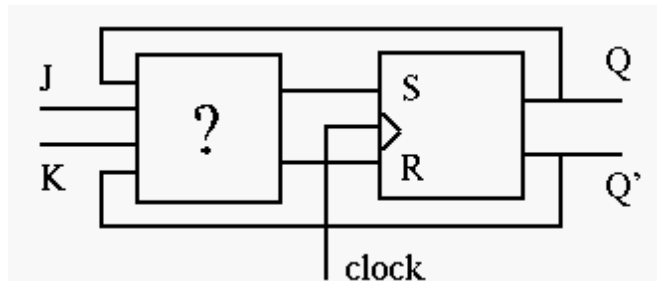
$$S = D, \quad R = D'$$



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:

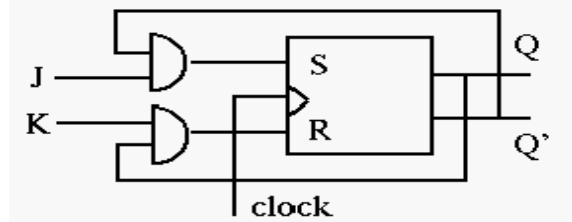
$Q \backslash J$	00	01	11	10
0	0	1	X	X
1	0	1	0	0

$S = Q'J$

$Q \backslash J$	00	01	11	10
0	X	0	0	0
1	X	0	1	1

$R = QK$

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



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" flip-flop 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 SequentialCircuit
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 bit-length 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

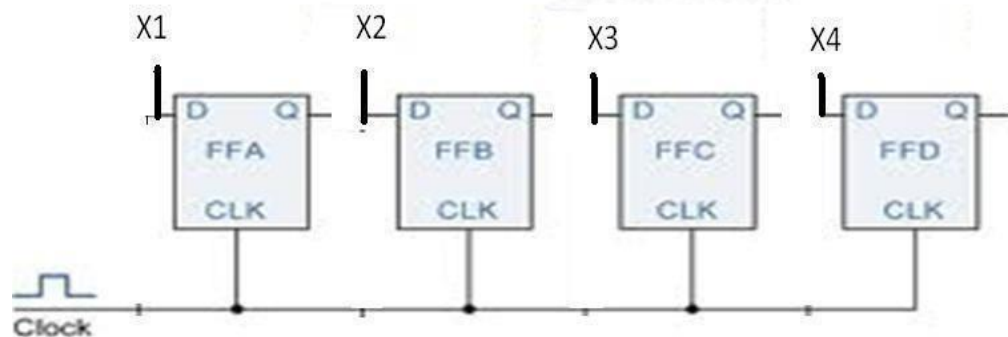


Figure: logic diagram of 4-bit buffer register

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 $\square\square\square$ goes LOW, all the FFs are RESET and the output becomes, $Q=0000$.

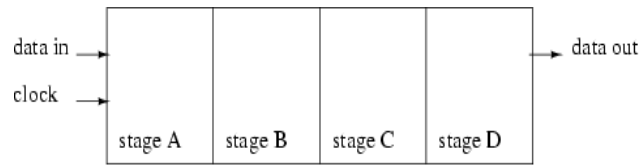
When $\square\square\square$ is HIGH, the register is ready for action. LOAD is the control input. When LOAD is HIGH, the data bits X can reach the D inputs of FF's.

$$Q_4Q_3Q_2Q_1 = X_4X_3X_2X_1$$

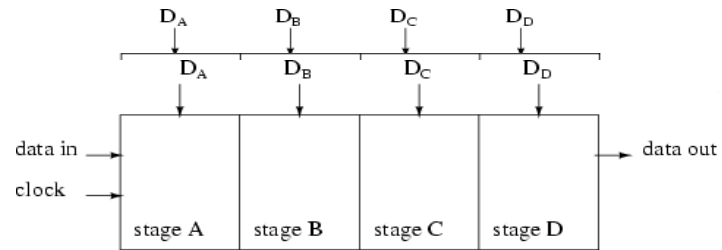
$$Q = X$$

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



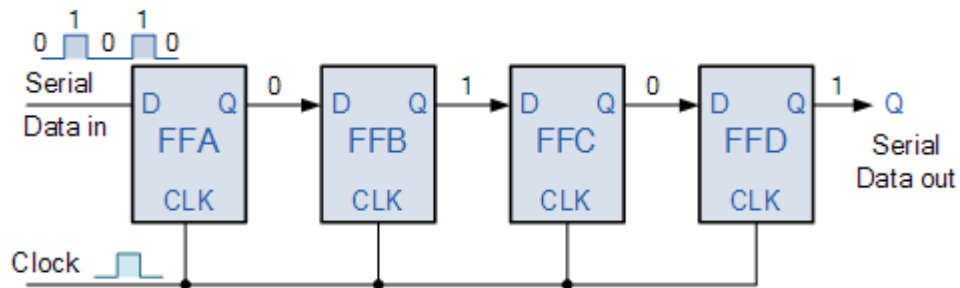
Parallel-in, serial-out shift register with 4-stages

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

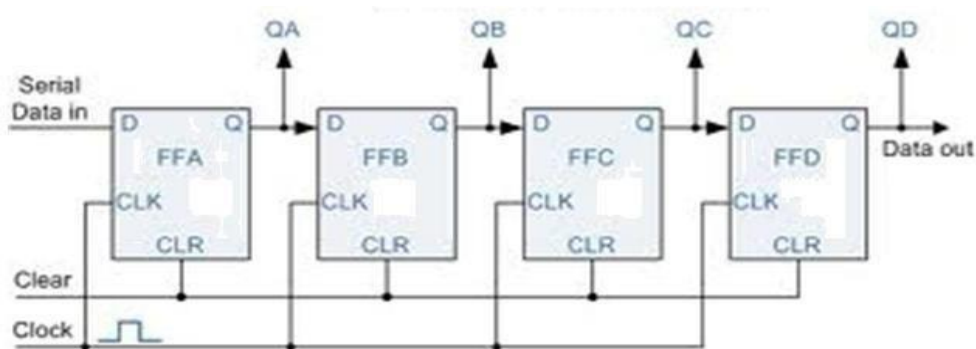
Serial IN, serial OUT, shift right, shift left register:

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:

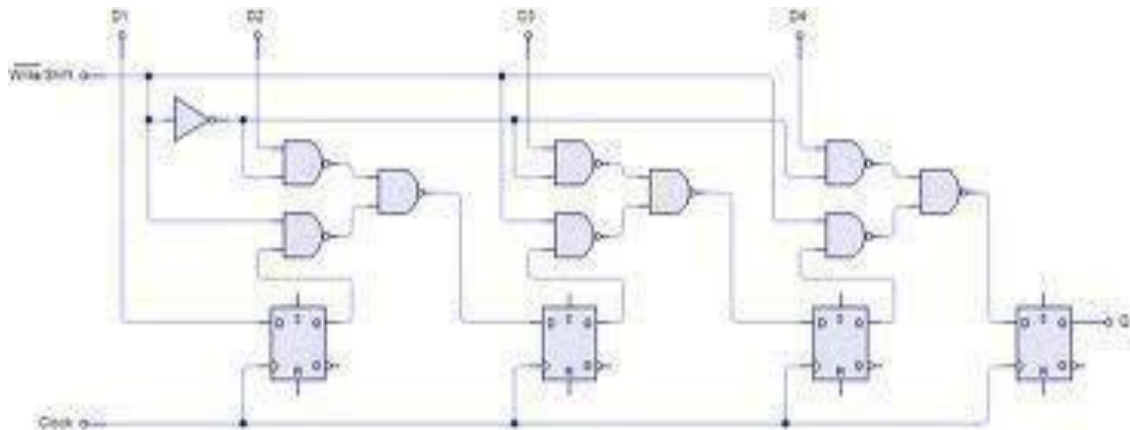


In this type of register, the data bits are entered into the register serially, but the data stored in the

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.

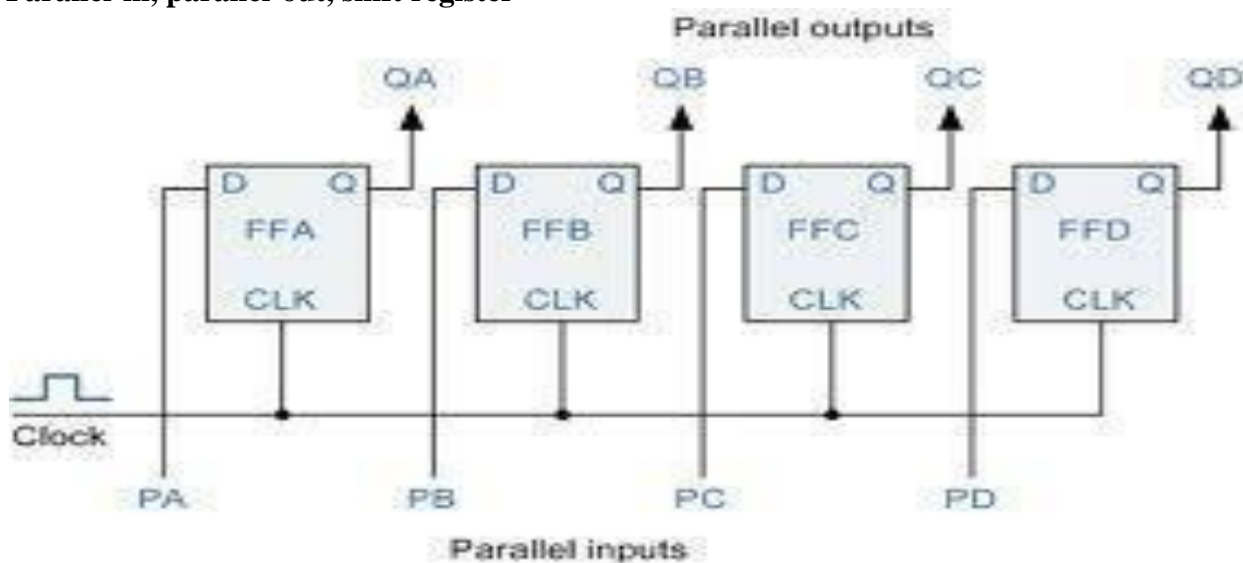
Parallel-in, serial-out, shift register:



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 terminal Q4

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 shift-register. 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

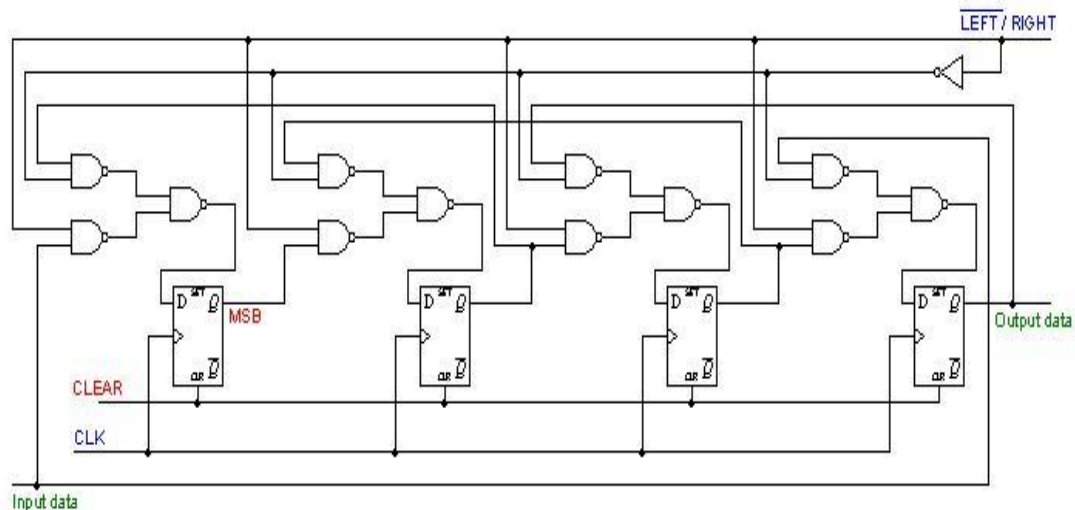


Figure: logic diagram of a 4-bit bidirectional shift register

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 the operations
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 the shift-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 s_1 and s_0 . Input 0 in each multiplexer is selected when $S_1S_0=00$, input 1 is selected when $S_1S_0=01$ and input 2 is selected when $S_1S_0=10$ and input 4 is selected when $S_1S_0=11$. The selection inputs control the mode of operation of the register according to the functions entries. When $S_1S_0=0$, 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 $S_1S_0=01$, terminal 1 of the multiplexer inputs have a path to the D inputs of the flip-flop. This causes a shift-right operation, with serial input transferred into flip-flop A_4 . When $S_1S_0=10$, a shift left operation results with the other serial input going into flip-flop A_1 . Finally when $S_1S_0=11$, the binary information on the parallel input lines is transferred into the register simultaneously during the next clock cycle

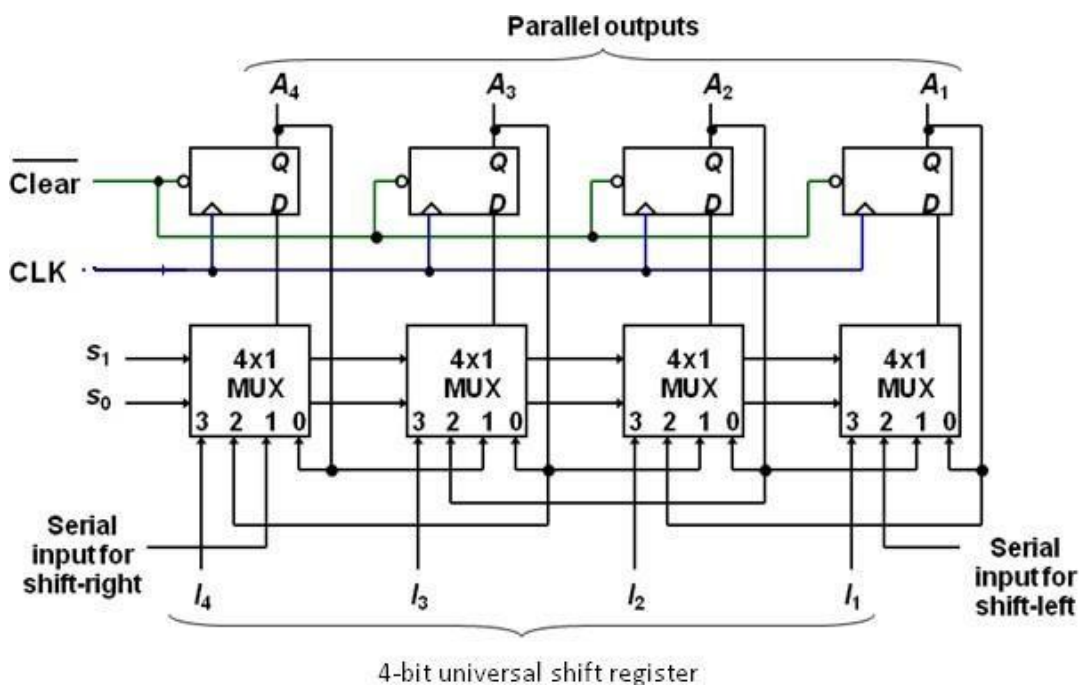


Figure: logic diagram 4-bit universal shift register

Function table for the register

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.

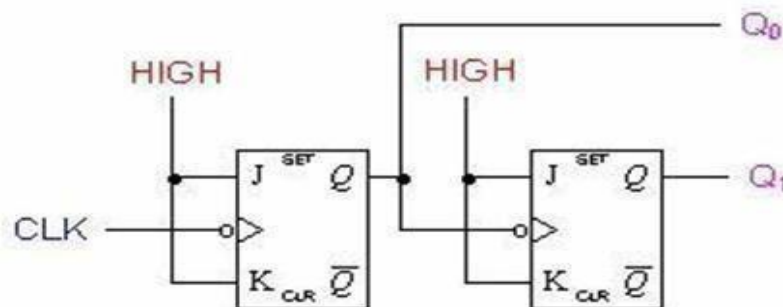
Asynchronous counters:

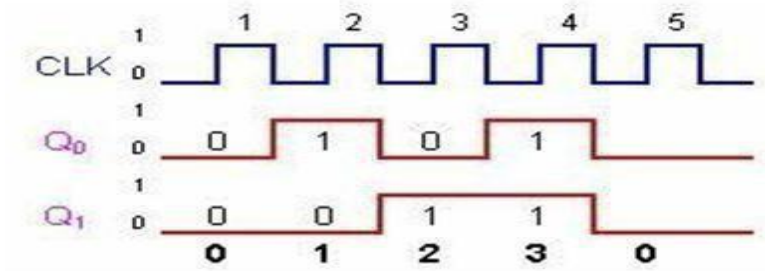
An asynchronous (ripple) counter is a single JK-type flip-flop, with its J (data) input fed from its own inverted output. This circuit can store one bit, and hence can count from zero to one before it overflows (starts over from 0). This counter will increment once for every clock cycle and takes two 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:

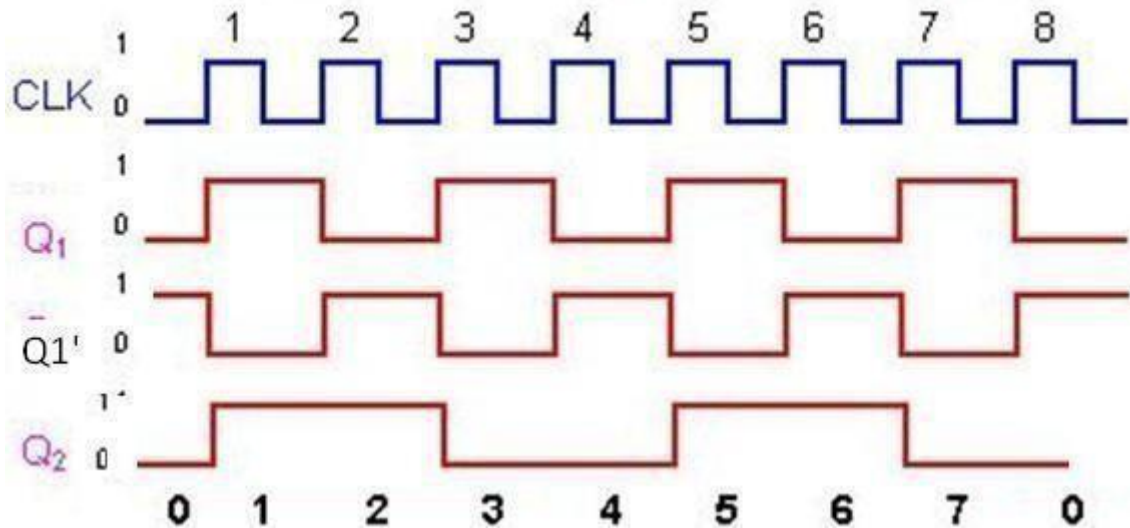
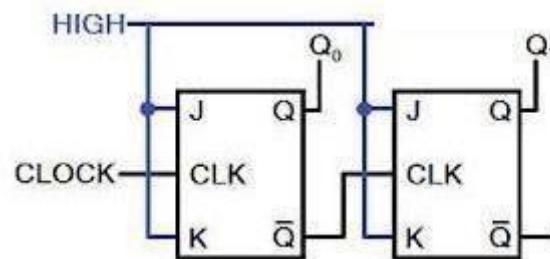
Two bit ripple counter used two flip-flops. There are four possible states from 2 – bit up- counting I.e. 00, 01, 10 and 11.

- The counter is initially assumed to be at a state 00 where the outputs of the tow flip-flops are noted as Q1Q0. Where Q1 forms the MSB and Q0 forms theLSB.
- For the negative edge of the first clock pulse, output of the first flip-flop FF1 toggles its state. Thus Q1 remains at 0 and Q0 toggles to 1 and the counter state are now read as 01.
- During the next negative edge of the input clock pulse FF1 toggles and Q0 = 0. The output Q0 being a clock signal for the second flip-flop FF2 and the present transition acts as a negative edge for FF2 thus toggles its state Q1 = 1. The counter state is now read as 10.
- For the next negative edge of the input clock to FF1 output Q0 toggles to 1. But this transition from 0 to 1 being a positive edge for FF2 output Q1 remains at 1. The counter state is now read as 11.
- For the next negative edge of the input clock, Q0 toggles to 0. This transition from 1 to 0 acts as a negative edge clock for FF2 and its output Q1 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, 00,11,10,01,00,11etc. 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 Q_1' is applied to the clock input of FF2, toggles FF2 and, therefore, Q_2 goes from a 0 to a 1. so, after one clock pulse $Q_2=1$ and $Q_1=1$, I.e., the state of the counter is 11.
- At the negative-going edge of the second clock pulse, Q_1 changes from a 1 to a 0 and Q_1' from a 0 to a 1.
- This positive-going signal at Q_1' does not affect FF2 and, therefore, Q_2 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 Q_1 , goes from a 0 to a 1 and Q_1' from 1 to 0. This negative going signal at Q_1' toggles FF2 and, so, Q_2 changes from 1 to 0, hence, the state of the counter after the third clock pulse is 01.
- At the negative going edge of the fourth clock pulse, FF1 toggles. So Q_1 , goes from a 1 to a 0 and Q_1' from 0 to 1. This positive going signal at Q_1' does not affect FF2 and, so, Q_2 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:

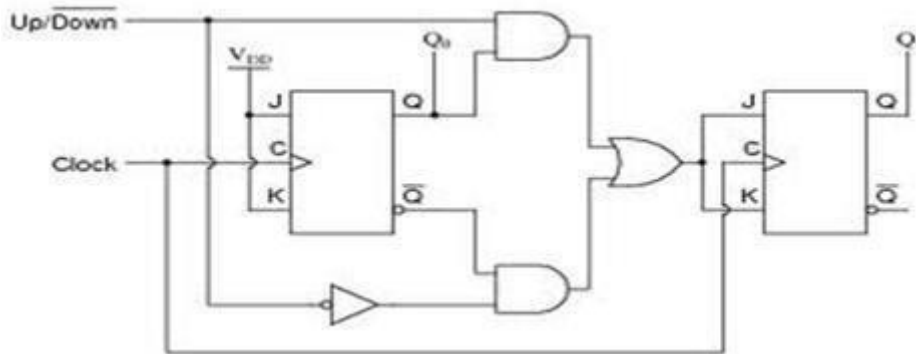


Figure: asynchronous 2-bit ripple up-down counter using negative edge triggered flip flop

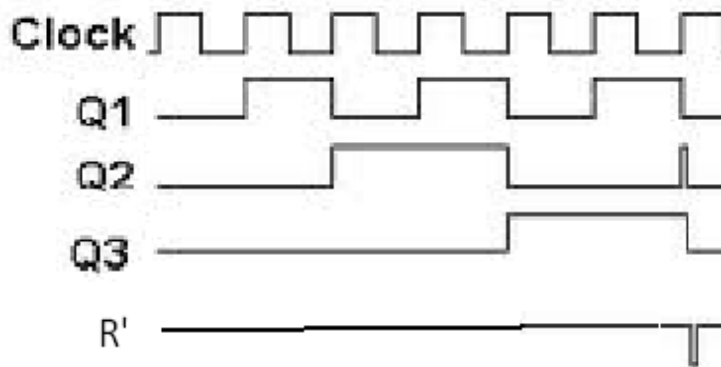
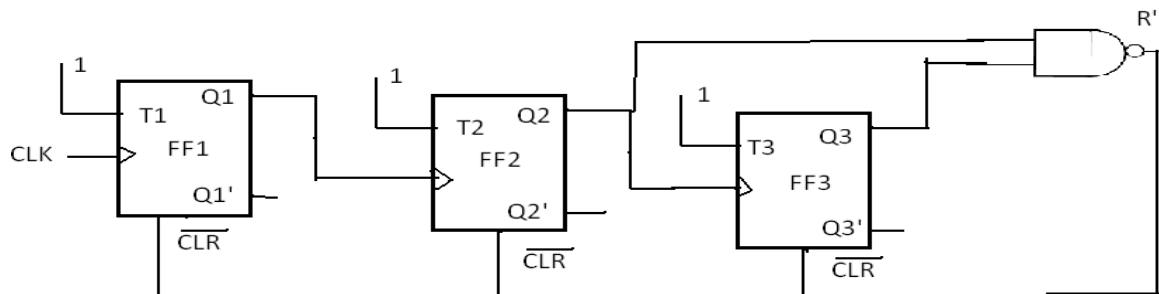
- As the name indicates an up-down counter is a counter which can count both in upward and downward directions. An up-down counter is also called a forward/backward counter or a bidirectional counter. So, a control signal or a mode signal M is required to choose the direction of count. When $M=1$ for up counting, Q_1 is transmitted to clock of FF2 and when $M=0$ for down counting, Q_1' is transmitted to clock of FF2. This is achieved by using two AND gates and one OR gate. The external clock signal is applied to FF1.
- Clock signal to FF2 = $(Q_1 \cdot \text{Up}) + (Q_1' \cdot \text{Down}) = Q_1 M + Q_1' M'$

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 a 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 110 and 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 Q_3 , Q_2 and Q_1 as the variables, and reset R as the output and obtain an expression for R in terms of Q_3 , Q_2 , and Q_1 that decides the feedback into be provided. From the truth table, $R = Q_3Q_2$. 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.

Therefore, $R=0$ for 000 to 101, $R=1$ for 110, and $R=X$ for 111

$$R = Q_3Q_2Q_1' + Q_3Q_2Q_1 = Q_3Q_2$$

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

The truth table is as shown in below.

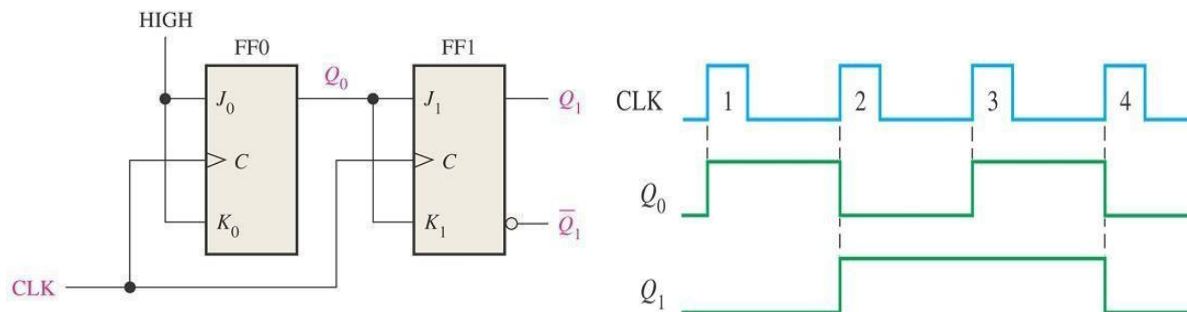
		Q2Q1			
		00	01	11	10
Q4Q3	00				
	01				
	11	X	X	X	X
	10		X	X	1

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 pulse 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 is also called an n th transition diagram, is a graphical means of depicting the sequence of states through which the counter progresses.

Step 2: 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.

Step 3: 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

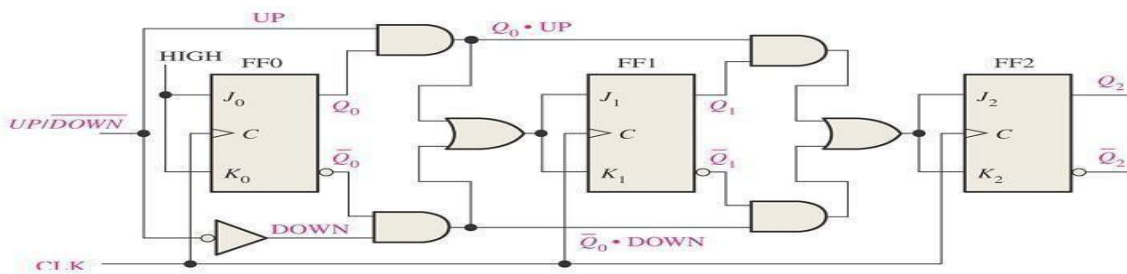
Step3: select the type of flip flop and draw the excitation table: JK flip-flops are selected and the excitation table of a 3-bit up-down counter using JK flip-flops is drawn as shown in fig.

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 J1 and 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.

	00	01	11	10
Q3Q2\Q1M				
1				
			1	
X	X	X	X	X
X	X	X	X	X

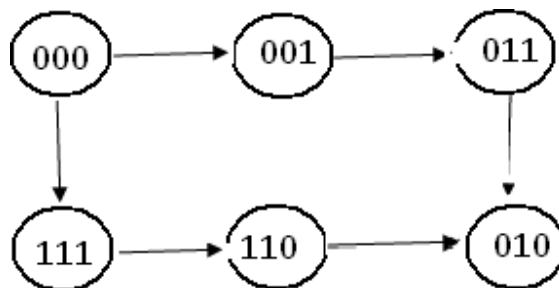
Step5: draw the logic diagram: a logic diagram using those minimal expressions can be drawn as shown in fig.



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 $n=3$ FFs ($N \leq 2^n$, 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.

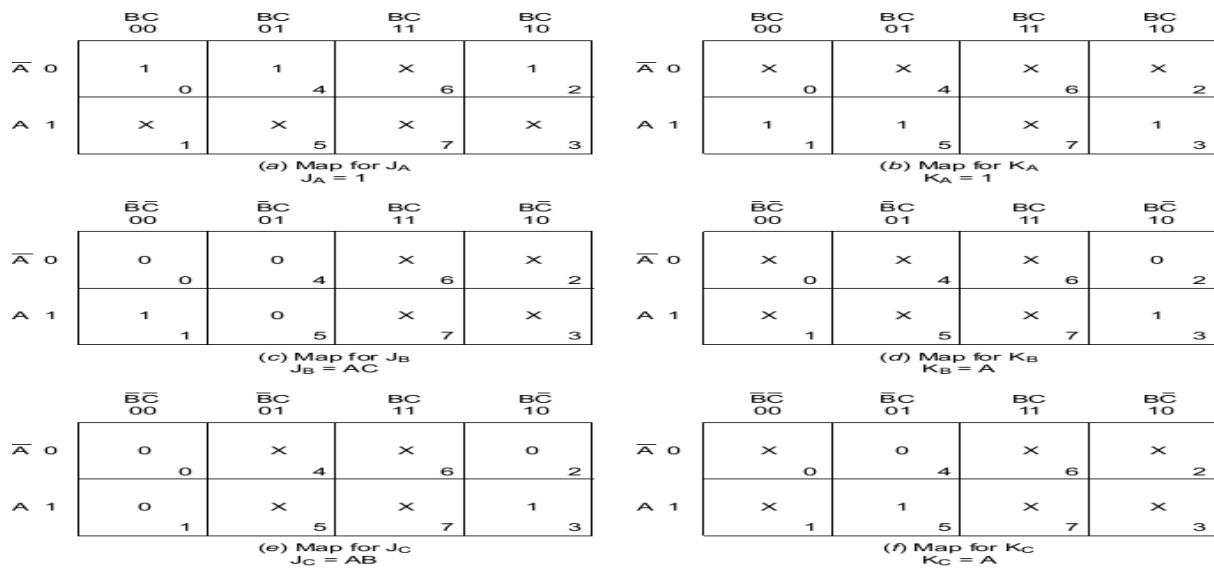
Step2: the state diagram: the state diagram of the mod-6 gray code converter is drawn as shown in fig.



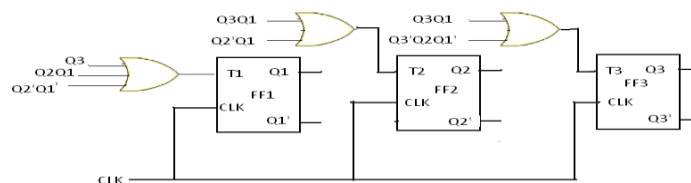
Step3: type of flip-flop and the excitation table: T flip-flops are selected and the excitation table of the mod-6 gray code counter using T-flip-flops is written as shown in fig.

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 $n=4\text{FFs}$ ($N \leq 2^n$, 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 ($\sum d(20,21,22,23,24,25,26,27,28,29,30,31)$) from the excitation table we can see that $T1=1$ and the expression for $T4, T3, T2$ are as follows.

$$T4 = \sum m(0,15,16,19) + d(20,21,22,23,24,25,26,27,28,29,30,31)$$

$$T3 = \sum m(7,15,16,8) + d(20,21,22,23,24,25,26,27,28,29,30,31)$$

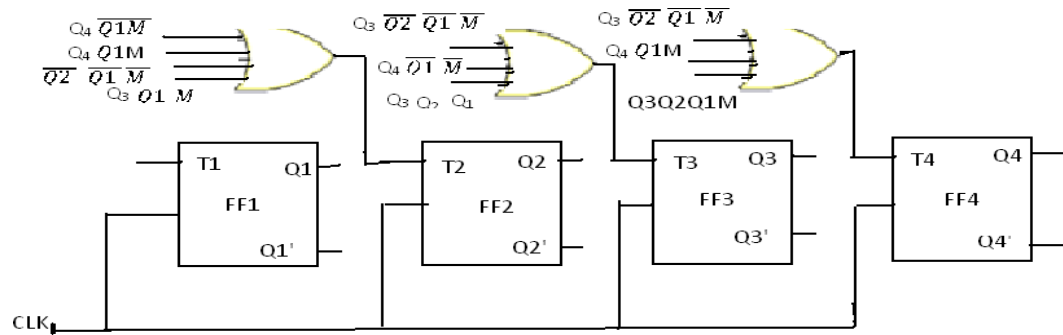
$$T2 = \sum m(3,4,7,8,11,12,15,16) + d(20,21,22,23,24,25,26,27,28,29,30,31)$$

PS					NS							
Q4	Q3	Q2	Q1	mode	Q4	Q3	Q2	Q1	required excitations			
				M					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 $Q_4Q_3Q_2Q_1M$ are valid and the remaining 12 combinations are don't cares. Minimizing K-maps for T_2 we get

$$T_2 = Q_4Q_1'M + Q_4'Q_1M + Q_2Q_1'M' + Q_3Q_1'M'$$

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.

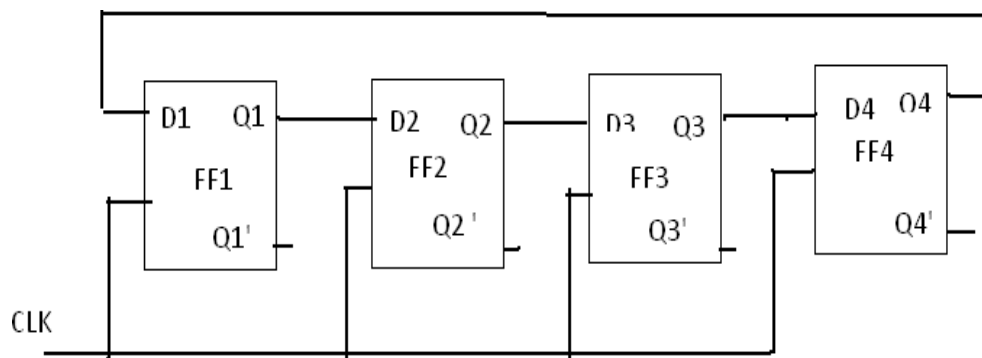


fig: logic diagram of 4-bit ring counter using D flip-flops

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., $Q_1=1, Q_2=0, Q_3=0, Q_4=0$. After each clock pulse, the contents of the register are shifted to the right by one bit and Q_4 is shifted back to Q_1 . The sequence repeats after four clock pulses. The 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, whereas 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:

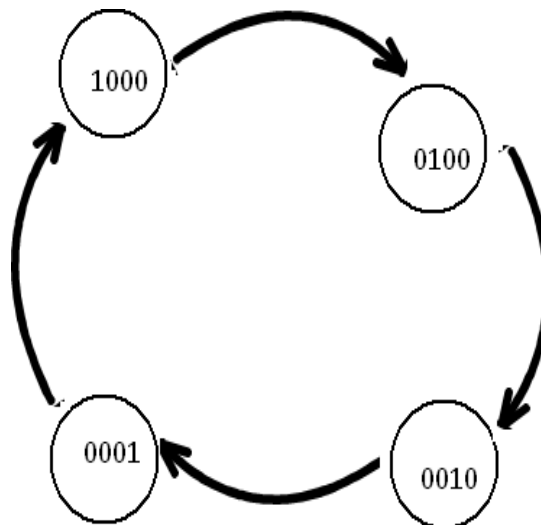
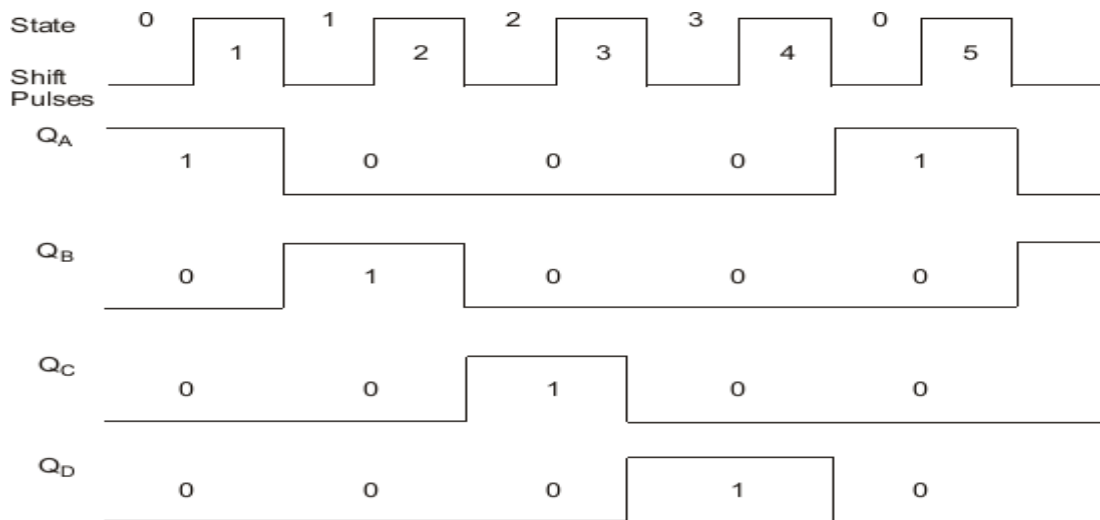


Figure: state diagram

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 in figure.

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 Q2 to Q3, Q3 to Q4 and the level of Q4' to Q1 and the sequences given in fig.

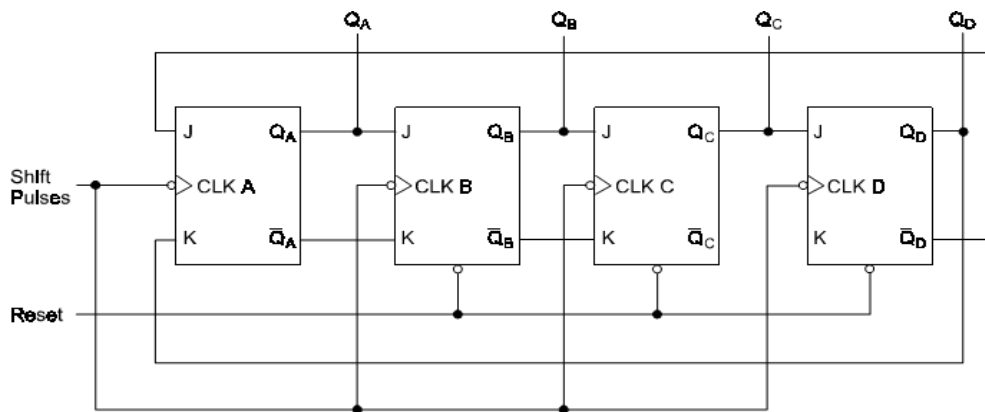


Figure: Johnson counter with JK flip-flops

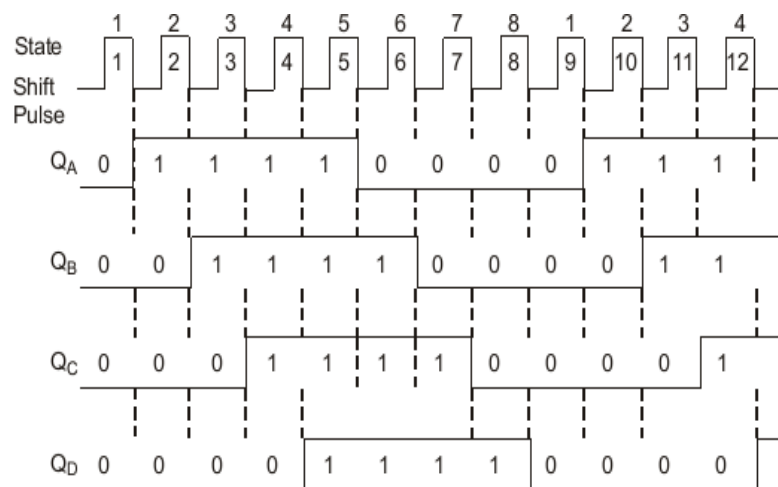
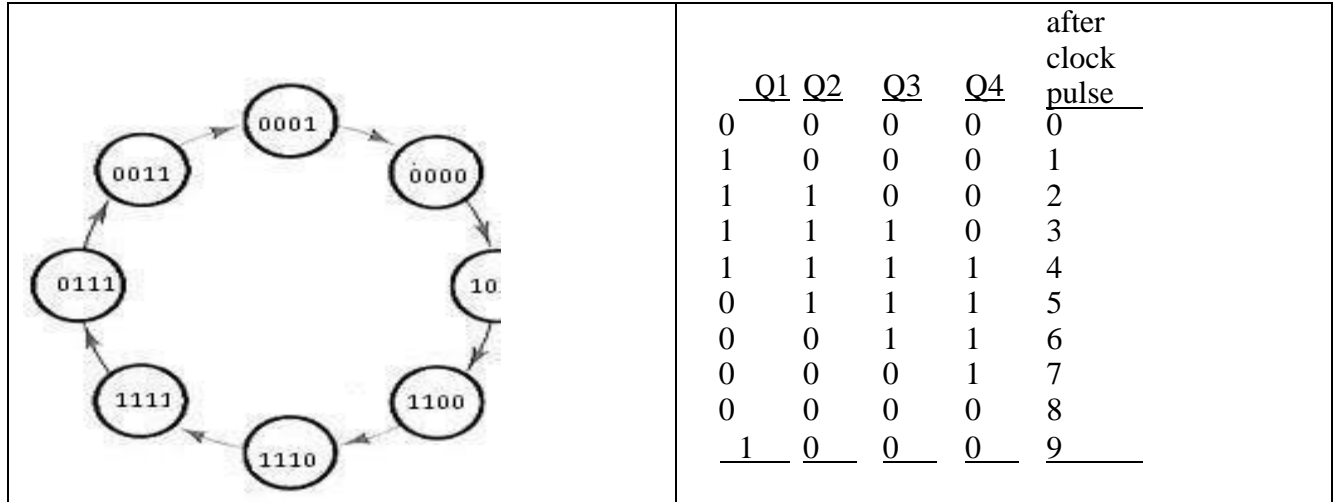


Figure: timing diagram

Statediagram:



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. Mealy 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 fact 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 circles. 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 directed 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.

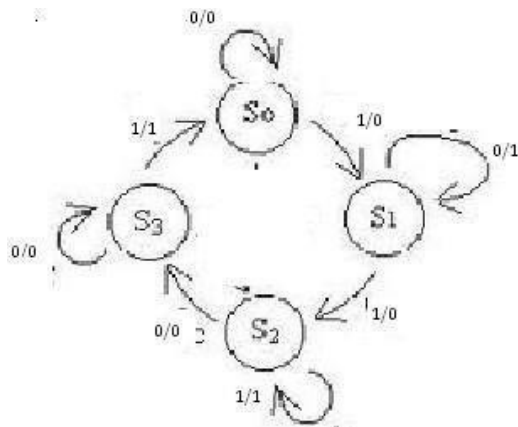


fig :a) state diagram (meelay circuit)

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: 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 in the circle below the present state, because the output depends only on the present state and not on the input.

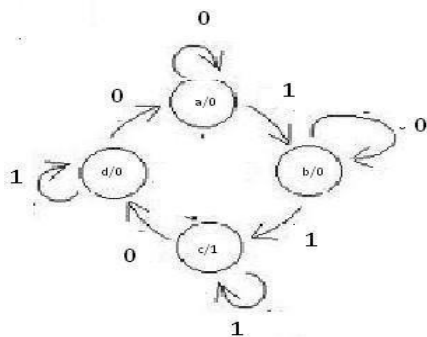


fig: a) state diagram (moore circuit)

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

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 X1 and 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 $X1(t_i)$ and $X2(t_i)$ at that time t_i-1 and of carry which had been generated at t_i-1 .

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

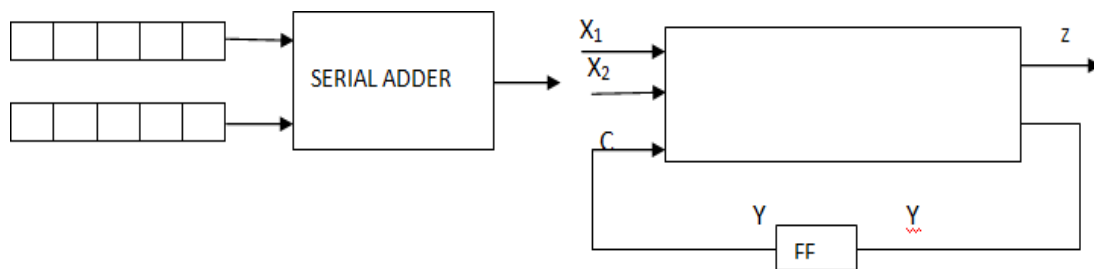
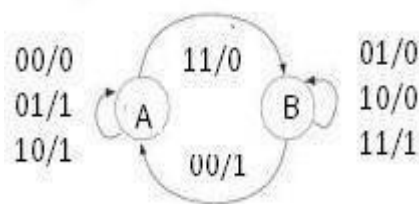


Figure: block diagram of serial binary adder

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	NS ,O/P			
	X1 X2			
	0	0	1	1
	0	1	0	1
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/F			
	0	0	1	1	0	0	1	1
	0	1	0	1	0	1	0	1
	0	0	0	1	0	1	1	1
	1	0	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	1	0	0	1
0	1	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

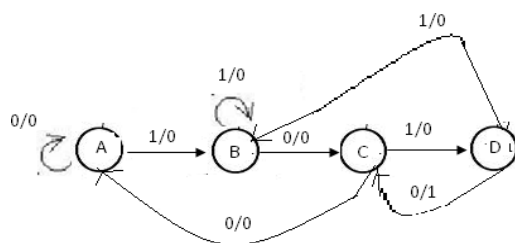
Sequence detector:

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 t1, 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 or 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.

PS(y1y2)	NS(Y1Y2)				O/P(z)	
	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.

PS		I/P	NS		INPUTS -		O/P
y1	Y2		Y1	Y2	FFS	D2	
		X			D1		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

MODULE-IV
ANALOG TO DIGITAL AND DIGITAL TO
ANALOG CONVERTER

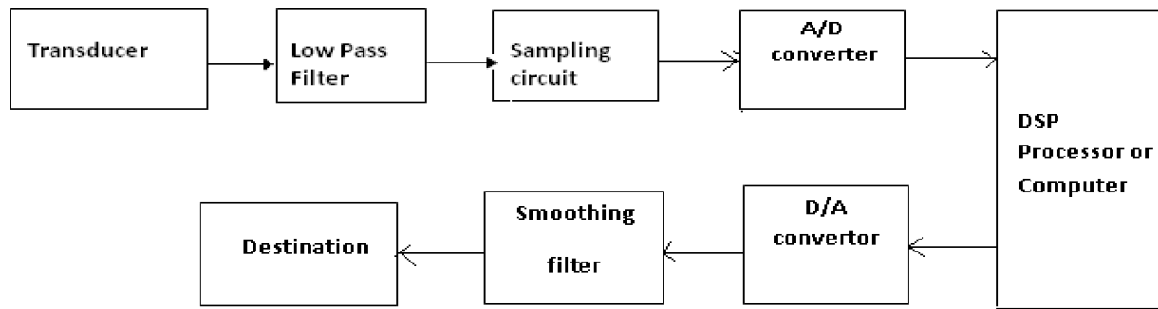


Fig 2.15: Application of A/D and D/A converters

Fig 2.15 shows the application of A/D and D/A converters. The transducer circuit will give an analog signal. This signal is transmitted through the LPF circuit to avoid higher components, and then the signal is sampled at twice the frequency of the signal to avoid the overlapping. The output of the sampling circuit is applied to A/D converter where the samples are converted into binary data i.e. 0's and 1's. Like this the analog data is converted into digital data.

The digital data is again reconverted back into analog by doing the exact opposite operation of the first half of the diagram. Then the output of the D/A converter is transmitted through the smoothing filter to avoid the ripples.

BASIC DAC TECHNIQUES

The input of the block diagram is binary data i.e. 0 and 1, it contains 'n' number of input bits designated as $d_1, d_2, d_3, \dots, d_n$. This input is combined with the reference voltage called V_{dd} to give an analog output.

Where d_1 is the MSB bit and d_n is the LSB bit

$$V_o = V_{dd}(d_1 \cdot 2^{-1} + d_2 \cdot 2^{-2} + d_3 \cdot 2^{-3} + \dots + d_n \cdot 2^{-n})$$

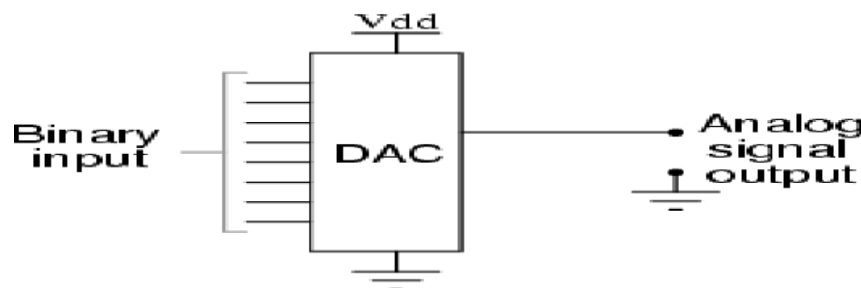


Fig .2.16: Basic DAC diagram

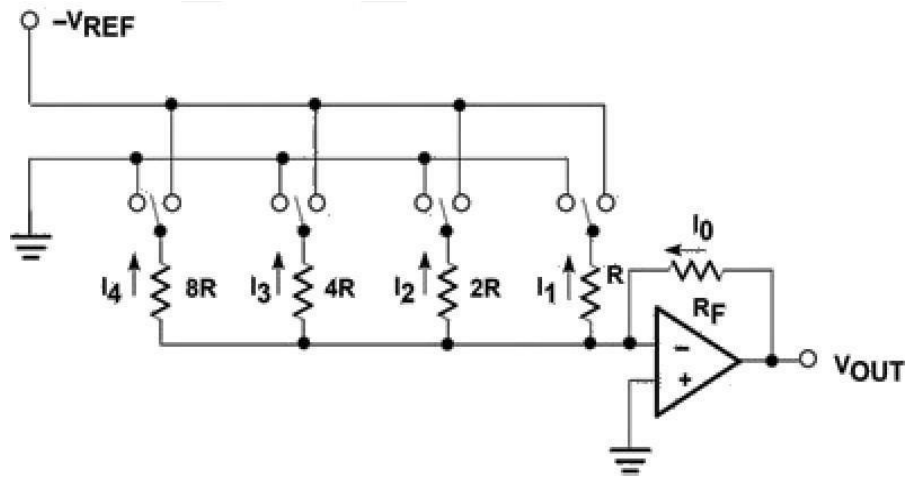


Fig. 2.17 simple 4-bit weighted resistor

Fig. 2.17 shows a simplest circuit of weighted resistor. It uses a summing inverting amplifier. It contains n - electronic switches (i.e. 4 switches) and these switches are controlled by binary input bits d_1, d_2, d_3, d_4 . If the binary input bit is 1 then the switch is connected to reference voltage $-V_{REF}$, if the binary input bit is 0 then the switch is connected to ground. The output current equation is $I_o = I_1 + I_2 + I_3 + I_4$

$$I_o = V_{REF} (d_1 \cdot 2^{-1} + d_2 \cdot 2^{-2} + d_3 \cdot 2^{-3} + d_4 \cdot 2^{-4})$$

The transfer characteristics are shown below (fig 2.13) for a 3-bit weighted resistor

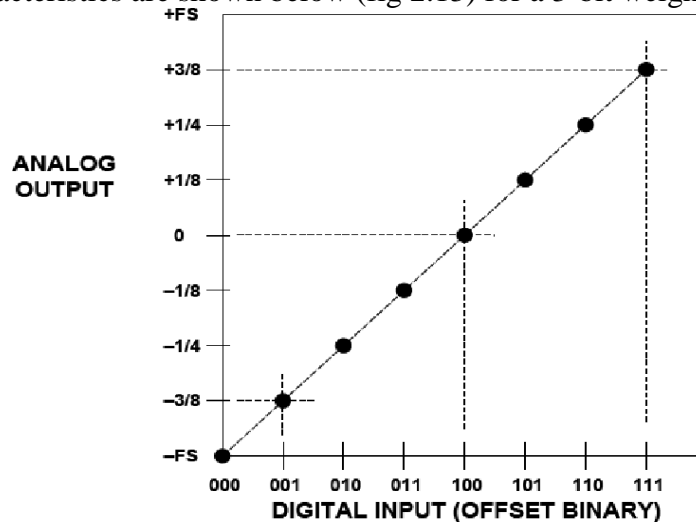


Fig 2.18 Transfer characteristics of 3-bit weighted resistor

Disadvantages of Weighted resistor D/A

Wide range of resistor's are required in this circuit and it is very difficult to fabricate such a wide range of resistance values in monolithic IC. This difficulty can be eliminated using R-2R ladder network.

R-2R LADDER DAC

Wide range of resistors required in binary weighted resistor type DAC. This can be avoided by using R-2R ladder type DAC. The circuit of R-2R ladder network is shown in fig

2.19. The basic theory of the R-2R ladder network is that current flowing through any input resistor ($2R$) encounters two possible paths at the far end. The effective resistances of both paths are the same (also $2R$), so the incoming current splits equally along both paths. The half-current that flows back towards lower orders of magnitude does not reach the op amp, and therefore has no effect on the output voltage. The half that takes the path towards the op amp along the ladder can affect the output. The inverting input of the op-amp is at virtual earth. Current flowing in the elements of the ladder network is therefore unaffected by switch positions.

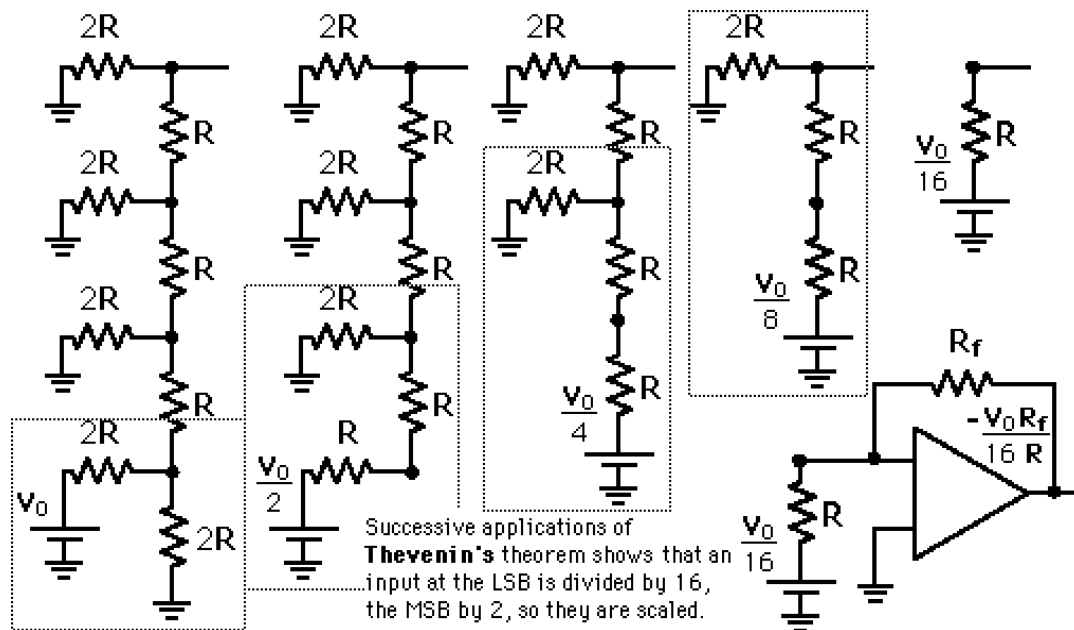


Fig 2.19: A 4-bit R-2R Ladder DAC

If we label the bits (or inputs) bit 1 to bit N the output voltage caused by connecting a particular bit to V_r with all other bits grounded is:

$$V_{out} = V_r/2^N$$

where N is the bit number. For bit 1, $V_{out} = V_r/2$, for bit 2, $V_{out} = V_r/4$ etc.

Since an R/2R ladder is a linear circuit, we can apply the principle of superposition to calculate V_{out} . The expected output voltage is calculated by summing the effect of all bits connected to V_r . For example, if bits 1 and 3 are connected to V_r with all other inputs grounded,

the output voltage is calculated by: $V_{out} = (V_r/2) + (V_r/8)$ which reduces to $V_{out} = 5V_r/8$.

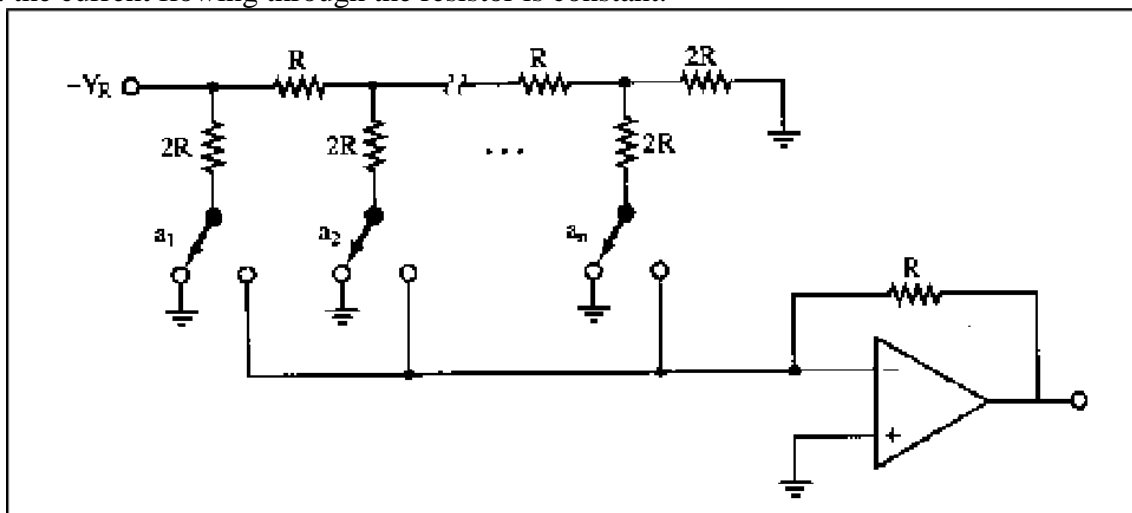
An R/2R ladder of 4 bits would have a full-scale output voltage of $1/2 + 1/4 + 1/8 + 1/16 = 15V_r/16$ or 0.9375 volts (if $V_r=1$ volt) while a 10bit R/2R ladder would have a full-scale output voltage of 0.99902 (if $V_r=1$ volt).

INVERTED R-2R LADDER DAC

In weighted resistor and R-2R ladder DAC the current flowing through the resistor is always changed because of the changing input binary bits 0 and 1. More power dissipation causes heating, which in turn creates non-linearity in DAC. This problem can be avoided by using INVERTED R-2R LADDER DAC (fig 2.20)

In this MSB and LSB is interchanged. Here each input binary word connects the corresponding switch either to ground or to the inverting input terminal of op-amp which is also at virtual ground. When the input binary is logic 1 then it is connected to the virtual ground, when input binary is logic 0 then it is connected to the ground

i.e. the current flowing through the resistor is constant.



DIFFERENT TYPES OF ADC'S

It provides the function just opposite to that of a DAC. It accepts an analog input voltage V_a and produces an output binary word $d_1, d_2, d_3, \dots, d_n$. Where d_1 is the most significant bit and d_n is the least significant bit.

ADCs are broadly classified into two groups according to their conversion techniques

- 1) Direct type
- 2) Integrating type

Direct type ADCs compares a given analog signal with the internally generated equivalent signal. This group includes

- i) Flash (Comparator) type converter

ii) Successive approximation type convertor

iii) Counter type

iv) Servo or Tracking type

Integrated type ADCs perform conversion in an indirect manner by first changing the analog input signal to linear function of time or frequency and then to a digital code. **FLASH**

(COMPARATOR) TYPE CONVERTER:

A direct-conversion ADC or flash ADC has a bank of comparators sampling the input signal in parallel, each firing for their decoded voltage range. The comparator bank feeds a logic circuit that generates a code for each voltage range. Direct conversion is very fast, capable of gigahertz sampling rates, but usually has only 8 bits of resolution or fewer, since the number of comparators needed, $2^N - 1$, doubles with each additional bit, requiring a large, expensive circuit. ADCs of this type have a large die size, a high input capacitance, high power dissipation, and are prone to produce

glitches at the output (by outputting an out-of- sequence code). Scaling to newer sub- micrometre technologies does not help as the device mismatch is the dominant design limitation. They are often used for video, wideband communications or other fast signals in optical storage.

A Flash ADC (also known as a direct conversion ADC) is a type of analog-to- digital converter that uses a linear voltage ladder with a comparator at each "rung" of the ladder to compare the input voltage to successive reference voltages. Often these reference ladders are constructed of many resistors; however modern implementations show that capacitive voltage division is also possible. The output of these comparators is generally fed into a digital encoder which converts the inputs into a binary value (the collected outputs from the comparators can be thought of as a unary value).

Also called the *parallel* A/D converter, this circuit is the simplest to understand. It is formed of a series of comparators, each one comparing the input signal to a unique reference voltage. The comparator outputs connect to the inputs of a priority encoder circuit, which then produces a binary output.

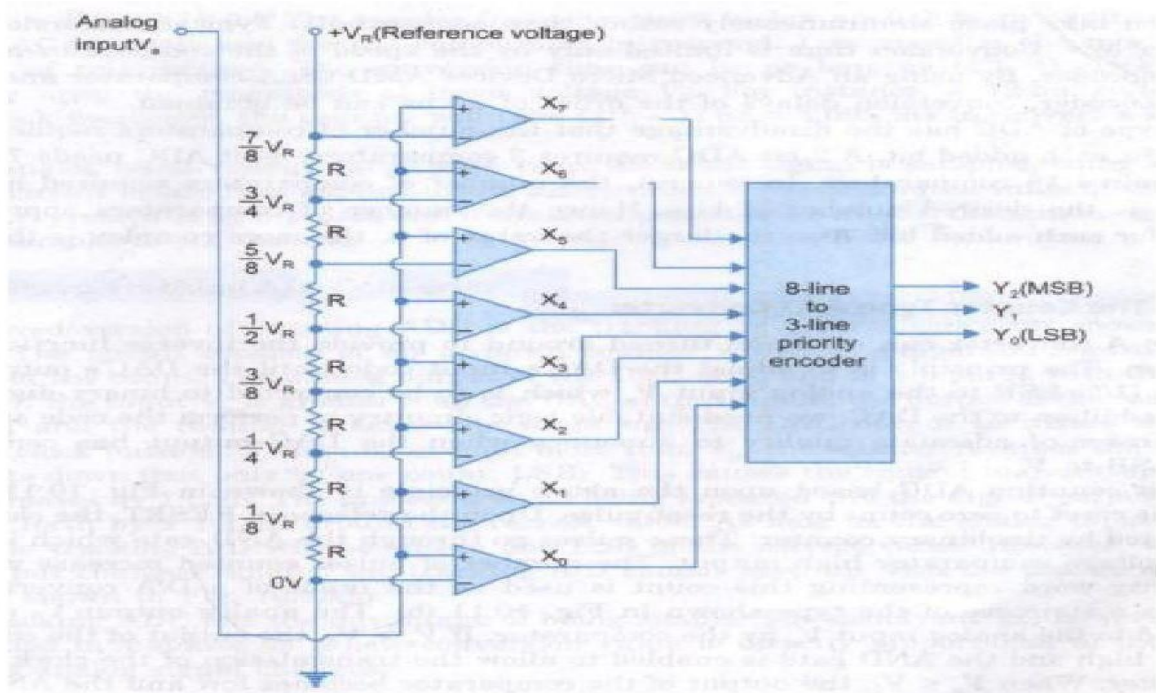


Fig 2.21: flash (parallel comparator) type ADC

V_R is a stable reference voltage provided by a precision voltage regulator as part of the converter circuit, not shown in the schematic. As the analog input voltage exceeds the reference voltage at each comparator, the comparator outputs will sequentially saturate to a high state. The priority encoder generates a binary number based on the highest-order active input, ignoring all other active inputs.

COUNTER TYPE A/D CONVERTER

In the fig 2.22 the counter is reset to zero count by reset pulse. After releasing the reset pulse the clock pulses are counted by the binary counter. These pulses go through the AND gate which is enabled by the voltage comparator high output. The number of pulses counted increase with time.

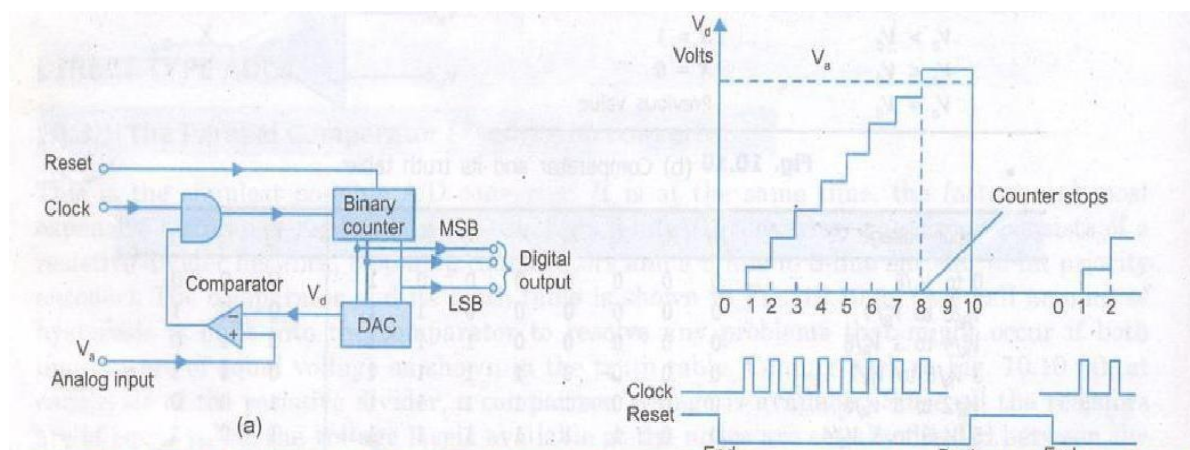


Fig 2.22: Countertype A/D converter

The binary word representing this count is used as the input of a D/A converter whose output is a stair case. The analog output V_d of DAC is compared to the analog input V_a by

the comparator. If $V_a > V_d$ the output of the comparator becomes high and the AND gate is enabled to allow the transmission of the clock pulses to the counter. When $V_a < V_d$ the output of the comparator becomes low and the AND gate is disabled. This stops the counting we can get the digital data.

SERVO TRACKING A/D CONVERTER :

An improved version of counting ADC is the tracking or servo converter shown in fig 2.23. The circuit consists of an up/down counter with the comparator controlling the direction of the count.

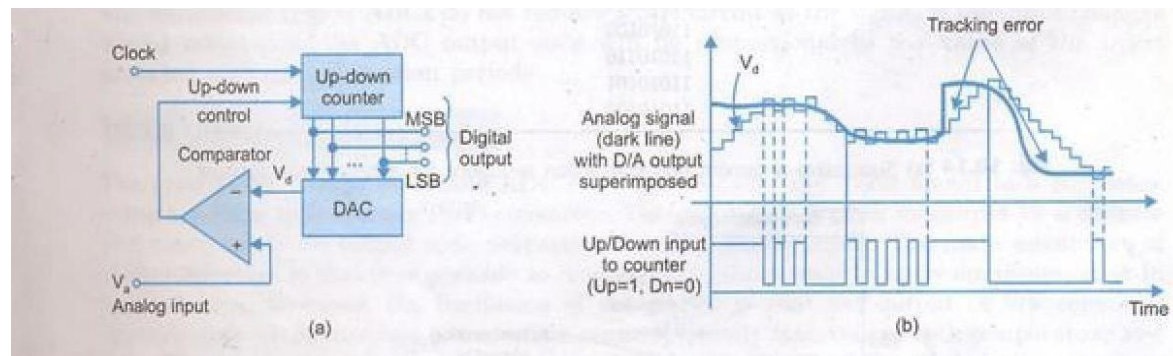


Fig: 2.23 (a) A tracking A/D converter (b) waveforms associated with a tracking A/D converter

The analog output of the DAC is V_d and is compared with the analog input V_a . If the input V_a is greater than the DAC output signal, the output of the comparator goes high and the counter is caused to count up. The DAC output increases with each incoming clock pulse when it becomes more than V_a the counter reverses the direction and counts down.

SUCCESSIVE-APPROXIMATION ADC:

One method of addressing the digital ramp ADC's shortcomings is the so-called successive-approximation ADC. The only change in this design as shown in the fig 2.19 is a very special counter circuit known as a successive-approximation register.

Instead of counting up in binary sequence, this register counts by trying all values of bits starting with the most-significant bit and finishing at the least-significant bit. Throughout the count process, the register monitors the comparator's output to see if the binary count is less than or greater than the analog signal input, adjusting the bit values accordingly. The way the register counts is identical to the "trial-and-fit" method of decimal-to-binary conversion, whereby different values of bits are tried from MSB to LSB to get a binary number that equals the original decimal number. The advantage to this counting strategy is much faster results: the DAC output converges on the analog signal input in much larger steps than with the 0-to-full count sequence of a regular counter.

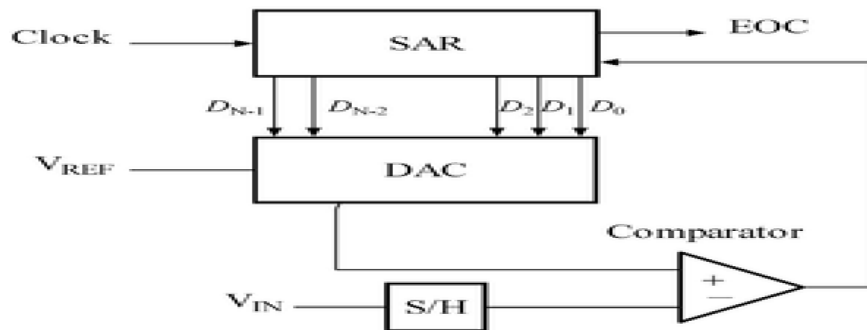


Fig: 2.24: Successive approximation ADC circuits

The successive approximation analog to digital converter circuit typically consists of four chief sub

1. A sample and hold circuit to acquire the input voltage (V_{in}).
2. An analog voltage comparator that compares V_{in} to the output of the internal DAC and outputs the result of the comparison to the successive approximation register (SAR).
3. A successive approximation register sub circuit designed to supply an approximate digital code of V_{in} to the internal DAC.
4. An internal reference DAC that supplies the comparator with an analog voltage equivalent of the digital code output of the SAR for comparison with V_{in} .

The successive approximation register is initialized so that the most significant bit (MSB) is equal to a digital 1. This code is fed into the DAC, which then supplies the analog equivalent of this digital code ($V_{ref}/2$) into the comparator circuit for comparison with the sampled input voltage. If this analog voltage exceeds V_{in} the comparator causes the SAR to reset this bit; otherwise, the bit is left a 1. Then the next bit is set to 1 and the same test is done, continuing this binary search until every bit in the SAR has been tested. The resulting code is the digital approximation of the sampled input voltage and is finally output by the DAC at the end of the conversion (EOC).

Mathematically, let $V_{in} = xV_{ref}$, so x in $[-1, 1]$ is the normalized input voltage. The objective is to approximately digitize x to an accuracy of $1/2^n$. The algorithm proceeds as follows:

1. Initial approximation $x_0 = 0$.
2. i th approximation $x_i = x_{i-1} - s(x_{i-1} - x)/2^i$.

where, $s(x)$ is the signum-function($\text{sgn}(x)$) (+1 for $x \geq 0$, -1 for $x < 0$). It follows using mathematical induction that $|x_n - x| \leq 1/2^n$.

As shown in the above algorithm, a SAR ADC requires:

1. An input voltage source V_{in} .
2. A reference voltage source V_{ref} to normalize the input.
3. A DAC to convert the i th approximation x_i to a voltage.
4. A Comparator to perform the function $s(x_i - x)$ by comparing the DAC's voltage with the input voltage.
5. A Register to store the output of the comparator and apply $x_{i-1} - s(x_{i-1} - x)/2^i$.

A successive-approximation ADC uses a comparator to reject ranges of voltages, eventually settling on a final voltage range. Successive approximation works by constantly comparing the input voltage to the output of an internal digital to analog converter (DAC, fed by the current value of the approximation) until the best approximation is achieved. At each step in this process, a binary value of the approximation is stored in a successive approximation register (SAR). The SAR uses a reference voltage (which is the largest signal the ADC is to convert) for comparisons.

For example if the input voltage is 60 V and the reference voltage is 100 V, in the 1st clock cycle, 60 V is compared to 50 V (the reference, divided by two. This is the voltage at the output of the internal DAC when the input is a '1' followed by zeros), and the voltage from the comparator is positive (or '1') (because 60 V is greater than 50 V). At this point the first binary digit (MSB) is set to a '1'. In the 2nd clock cycle the input voltage is compared to 75 V (being halfway between 100 and 50 V: This is the output of the internal DAC when its input is '11' followed by zeros) because 60 V is less than 75 V, the comparator output is now negative (or '0'). The second binary digit is therefore set to a '0'. In the 3rd clock cycle, the input voltage is compared with 62.5 V (halfway between 50 V and 75 V: This is the output of the internal DAC when its input is '101' followed by zeros). The output of the comparator is negative or '0' (because 60 V is less than 62.5 V) so the third binary digit is set to a 0. The fourth clock cycle similarly results in the fourth digit being a '1' (60 V is greater than 56.25 V, the DAC output for '1001' followed by zeros). The result of this would be in the binary form 1001. This is also called *bit-weighting conversion*, and is similar to a binary search.

The analogue value is rounded to the nearest binary value below, meaning

this converter type is mid-rise (see above). Because the approximations are successive (not simultaneous), the conversion takes one clock-cycle for each bit of resolution desired. The clock frequency must be equal to the sampling frequency multiplied by the number of bits of resolution desired. For example, to sample audio at 44.1 kHz with 32 bit resolution, a clock frequency of over 1.4 MHz would be required. ADCs of this type have good resolutions and quite wide ranges. They are more complex than some other designs.

DUAL-SLOPE ADC

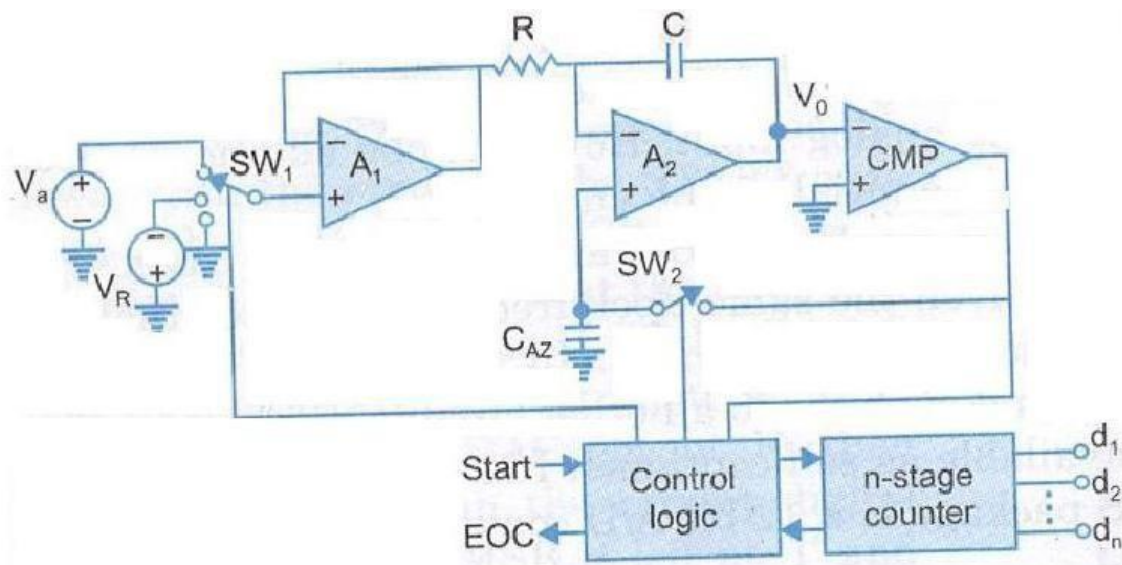


Fig 2.25 (a): Functional diagram of dual slope ADC

An integrating ADC (also **dual-slope** ADC) shown in fig 2.25 (a) applies the unknown input voltage to the input of an integrator and allows the voltage to ramp for a fixed time period (the run-up period). Then a known reference voltage of opposite polarity is applied to the integrator and is allowed to ramp until the integrator output returns to zero (the run-down period). The input voltage is computed as a function of the reference voltage, the constant run-up time period, and the measured run-down time period. The run-down time measurement is usually made in units of the converter's clock, so longer integration times allow for higher resolutions. Likewise, the speed of the converter can be improved by sacrificing resolution. Converters of this type (or variations on the concept) are used in most digital voltmeters for their linearity and flexibility.

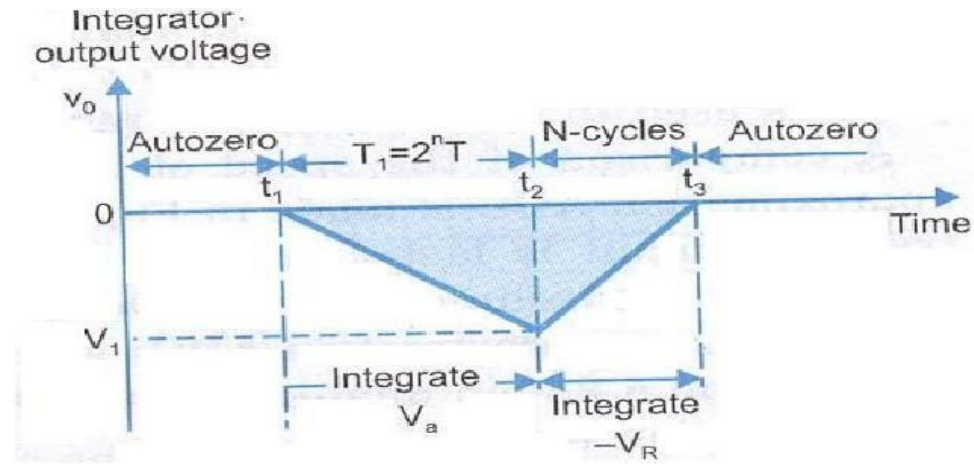


Fig 2.25 (b) o/p waveform of dual slope ADC

In operation the integrator is first zeroed (close SW2), then attached to the input (SW1 up) for a fixed time M counts of the clock (frequency $1/t$). At the end of that time it is attached to the reference voltage (SW1 down) and the number of counts N which accumulate before the integrator reaches zero volts output and the comparator output changes are determined. The waveform of dual slope ADC is shown in fig 2.25 (b).

The equations of operation are therefore:

$$T_1 = t_2 - t_1 = \frac{2^n \text{ counts}}{\text{clock rate}}$$

And

$$t_3 - t_2 = \frac{\text{digital count } N}{\text{clock rate}}$$

For an integrator,

The voltage V_o will be equal to V_1 at the instant t_2 and can be written as

$$V_1 = \frac{-1}{RC} V_a (t_2 - t_1) \quad \Delta V_o = \frac{-1}{RC} V (\Delta t)$$

The voltage V_1 is also given by

$$V_1 = \frac{-1}{RC} (-V_R) (t_2 - t_3)$$

So,

$$V_a (t_2 - t_1) = (V_R) (t_3 - t_2)$$

Putting the values of $(t_2 - t_1) = 2^n$ and

$$(t_3 - t_2) = N_{\text{get}}^{\text{we}}$$

$$V_a(2^n) = (V_R)N$$

Or,

$$V_a = (V_R) \left(\frac{N}{2^n} \right)$$

SPECIFICATIONS FOR DAC/ADC

1. **RESOLUTION:** The Resolution of a converter is the smallest change in voltage which may be produced at the output of the converter.

$$\text{Resolution (in volts)} = (\text{VFS}) / (2^n - 1) = 1 \text{ LSB increment}$$

Ex: An 8-bit D/A converter have $2^8 - 1 = 255$ equal intervals. Hence the smallest change in output voltage is $(1/255)$ of the full scale output range.

An 8-bit DAC is said to have: 8 bit resolution

: a resolution of 0.392 of full scale

: a resolution of 1 part in 255

Similarly the resolution of an A/D converter is defined as the smallest change in analog input for a one bit change at the output.

Ex: the input range of 8-bit A/D converter is divided into 255 intervals. So the resolution for a 10V input range is $39.22 \text{ mV} = (10\text{V}/255)$

2. **LINEARITY:** The linearity of an A/D or D/A converter is an important measure of its accuracy and tells us how close the converter output is to its ideal characteristics.

3. **GLITCHES (PARTICULARLY DAC):** In transition from one digital input to the next, like 0111 to 1000, it may effectively go through 1111 or 0000, which produces —unexpected voltage briefly. It can cause problems elsewhere.

4. **ACCURACY:** Absolute accuracy is the maximum deviation between the actual converter output and the ideal converter output.

5. **MONOTONIC:** A monotonic DAC is the one whose analog output increases for an increase in digital input. It is essential in control applications. If a DAC has to be monotonic, the error should be less than $\pm(1/2)$ LSB at each output level.

6. **SETTLING TIME:** The most important dynamic parameter is the settling time. It represents the time it takes for the output to settle within a specified band $\pm (1/2)$ LSB of its final value following a code change at the input. It depends upon the switching time of the logic circuitry due to internal parasitic capacitances and inductances. Its

ranges from 100ns to 10 μ s.

7. **STABILITY:** The performance of converter changes with temperature, age and power supply variations. So the stability is required.

Successive Approximation Converter

The successive approximation technique uses a very efficient code search strategy to complete n-bit conversion in just n-clock periods. An eight bit converter would require eight clock pulses to obtain a digital output. Figure 10.13 shows an eight bit converter. The circuit uses a successive approximation register (SAR) to find the required value of each bit by trial and error. The circuit operates as follows. With the arrival of the START command, the SAR sets the MSB $d_1 = 1$ with all other bits to zero so that the trial code is 10000000. The output V_d of the DAC is now compared with analog input V_a . If V is greater than the DAC output V_d then 10000000 is less than the correct digital representation. The MSB is left at '1' and the next lower significant bit is made '1' and further tested.

However, if V_a is less than the DAC output, then 10000000 is greater than the correct digital representation. So reset MSB to '0' and go on to the next lower significant bit. This procedure is repeated for all subsequent bits, one at a time, until all bit positions have been tested. Whenever the DAC output crosses V_a , the comparator changes state and this can be taken as the end of conversion (EOC) command. Figure 10.14 (a) shows a typical conversion sequence and Fig. 10.14 (b)

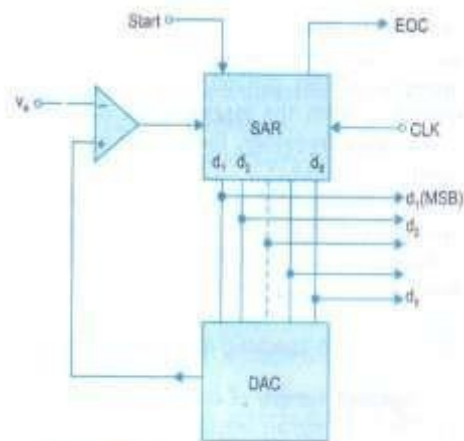


Fig. 10.13 Functional diagram of the successive approximation ADC

Correct digital representation	Successive approximation register output V_d at different stages in the conversion	Comparator output
11010100	10000000	1 (Initial output)
	11000000	1
	11100000	0
	11010000	1
	11011000	0
	11010100	1
	11010110	0
	11010101	0
	11010100	

Fig. 10.14 (a) Successive approximation conversion sequence for a typical analog input

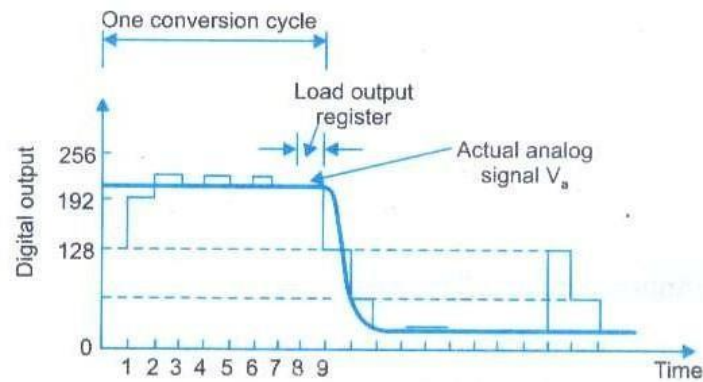


Fig. 10.14 (b) The D/A output voltage is seen to become successively closer to the actual analog input voltage

shows the associated wave forms. It can be seen that the D/A output voltage becomes successively closer to the actual analog input voltage. It requires eight pulses to establish the accurate output regardless of the value of the analog input. However, one additional clock pulse is used to load the output register and reinitialize the circuit.

A comparison of the speed of an eight bit tracking ADC and an eight bit successive approximation ADC is made in Fig. 10.15. Given the same clock frequency, we see that the tracking circuit is faster only for small changes in the input. In general, the successive approximation technique is more versatile and superior to all other circuits discussed so far. Successive approximation ADCs are available as self contained ICs. The AD7592 (Analog Devices Co.) a 28-pin dual-in-line CMOS package is a 12-bit A/D converter using successive approximation technique.

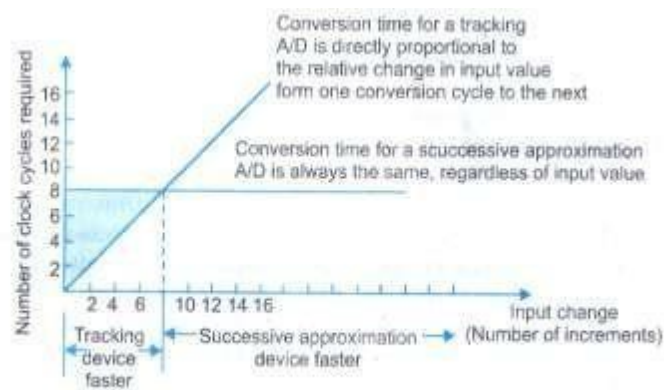


Fig. 10.15 Comparison of conversion times for tracking and successive approximation A/D devices

DAC/ADC SPECIFICATIONS

Both D/A and A/D converters are available with wide range of specifications. The various important specifications of converters generally specified by the manufacturers are analyzed. Resolution: The resolution of a converter is the smallest change in voltage which may be produced at the output (or input) of the converter. For example, an 8-bit D/A converter has $2^8 - 1 = 255$ equal intervals. Hence the smallest change in output voltage is $(1/255)$ of the full scale output range. In short, the resolution is the value of the LSB.

$$\text{Resolution (in volts)} = \frac{V_{FS}}{2^n - 1} = 1 \text{ LSB increment} \quad (10.8)$$

However, resolution is stated in a number of different ways. An 8-bit DAC is said to have

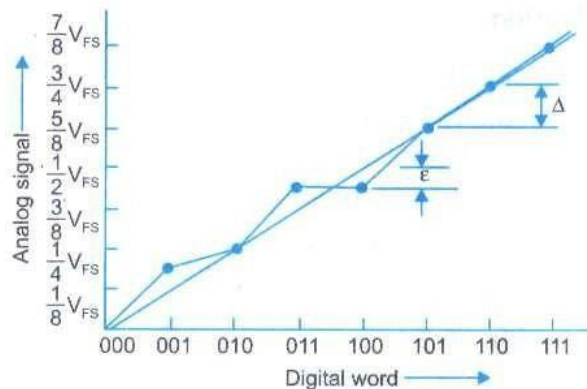
- :8 bit resolution
- : a resolution of 0.392 of full-scale
- : a resolution of 1 part in 255

Similarly, the resolution of an A/D converter is defined as the smallest change in analog input for a one bit change at the output. As an example, the input range of an 8-bit A/D converter is divided into 255 intervals. So the resolution for a 10 V input range is 39.22 mV ($= 10 \text{ V}/255$). Table 10.1 gives the resolution for 6-16 bit DACs.

Table 10.1 Resolution for 6–16 bit DACs

Bits	Intervals	LSB size (% of Full Scale)	LSB size (10 V Full Scale)
6	63	1.588%	158.8 mV
8	256	0.392%	39.2 mV
10	1023	0.0978%	9.78 mV
12	4095	0.0244%	2.44 mV
14	16383	0.0061%	0.61 mV
16	65535	0.0015%	0.15 mV

Linearity: The linearity of an A/D or D/A converter is an important measure of its accuracy and tells us how close the converter output is to its ideal transfer characteristics. In an ideal DAC, equal increment in the digital input should produce equal increment in the analog output and the transfer curve should be linear. However, in an actual DAC, output voltages do not fall on a straight line because of gain and offset errors as shown by the solid line curve in Fig. 10.17. The static performance of a DAC is determined by fitting a straight line through the measured output points. The linearity error measures the deviation of the actual output from the fitted line and is given by ϵ/Δ as shown in Fig. 10.17. The error is usually expressed as a fraction of LSB increment or percentage of full-scale voltage. A good converter exhibits a linearity error of less than $\pm (1/2)$ LSB.

**Fig. 10.17** Linearity error for 3-bit DAC

Accuracy: Absolute accuracy is the maximum deviation between the actual converter output and the ideal converter output. Relative accuracy is the maximum deviation after gain and offset errors have been removed. Data sheets normally specify relative accuracy rather than absolute accuracy. The accuracy of a converter is also specified in terms of LSB increments or percentage of full scale voltage.

Monotonicity: A monotonic DAC is the one whose analog output increases for an increase in digital input. Figure 10.18 represents the transfer curve for a non-monotonic DAC, since the output decreases when input code changes from 001 to 010. A monotonic characteristic is essential in control applications, otherwise oscillations can result. In successive approximation ADCs, a non-monotonic characteristic may lead to missing codes.

If a DAC has to be monotonic, the error should be less than $\pm (1/2)$ LSB at each output level. All the commercially available DACs are monotonic because the linearity error never exceeds $\pm (1/2)$ LSB at each output level.

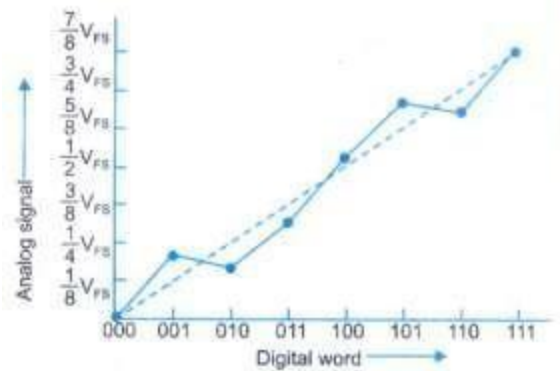


Fig. 10.18 A non-monotonic 3-bit DAC

Settling time: The most important dynamic parameter is the settling time. It represents the time it takes for the output to settle within a specified band $\pm (1/2)$ LSB of its final value following a code change at the input (usually a full scale change). It depends upon the switching time of the logic circuitry due to internal parasitic capacitances and inductances. Settling time ranges from 100 ns to 10 μ s depending on word length and type of circuit used.

Stability: The performance of converter changes with temperature, age and power supply variations. So all the relevant parameters such as offset, gain, linearity error and monotonicity must be specified over the full temperature and power supply ranges.

A brief overview of ADC and DAC selection guide is given below:

A/D converters:

AD 7520/AD 7530

10-bit binary multiplying type

AD 7521/AD 7531

12-bit binary multiplying type

ADC 0800/0801/0802

8-bit ADC

D/A converters:

DAC 0800/0801/0802

8-bit DAC

DAC 0830/0831/0832

microprocessor compatible 8-bit DAC

DAC 1200/1201

12-bit DAC

DAC 1208/1209/1210

12-bit microprocessor compatible DAC

MODULE-V
SEMICONDUCTOR MEMORIES AND
PROGRAMMABLE LOGIC DEVICES

BASIC COMPUTER ORGANIZATION:

Most of the computer systems found in automobiles and consumer appliances to personal computers and main frames have some basic organization. The basic computer organization has three main components:

- CPU
- Memory subsystem
- I/O subsystem.

The generic organization of these components is shown in the figure below.

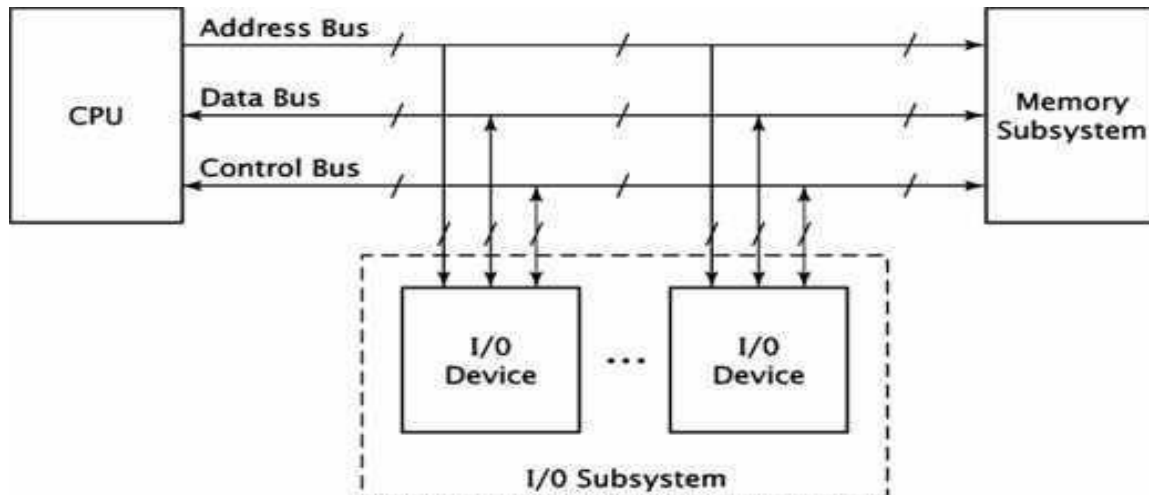


Fig 1.1 Generic computer Organization

System bus:

Physically the bus is a set of wires. The components of a computer are connected to the buses. To send information from one component to another the source component outputs data on to the bus. The destination component then inputs the data from bus.

The system has three buses

- Address bus
 - Data bus
 - Control bus
- The uppermost bus in this figure is the **address bus**. When the CPU reads data or instructions from or writes data to memory, it must specify the address of the memory location it wishes to access.

- Data is transferred via the **data bus**. When CPU fetches data from memory it first outputs the memory address on to its address bus. Then memory outputs the data onto the data bus. Memory then reads and stores the data at the proper locations.
- **Control bus** carries the control signal. Control signal is the collection of individual control signals. These signals indicate whether data is to be read into or written out of the CPU.

Instruction cycles:

- The instruction cycle is the procedure a microprocessor goes through to process an instruction.
- First the processor **fetches** or reads the instruction from memory. Then it decodes the instruction determining which instruction it has fetched. Finally, it performs the operations necessary to execute the instruction.
- After fetching it **decodes** the instruction and controls the execution procedure. It performs some Operation internally, and supplies the address, data & control signals needed by memory & I/O devices to **execute** the instruction.
- The READ signal is a signal on the control bus which the microprocessor asserts when it is ready to read data from memory or I/O device.
- When READ signal is asserted the memory subsystem places the instruction code be fetched on to the computer system's data bus. The microprocessor then inputs the data from the bus and stores its internal register.
- READ signal causes the memory to read the data, the WRITE operation causes the memory to store the data.

Below figure shows the memory read and memory write operations.

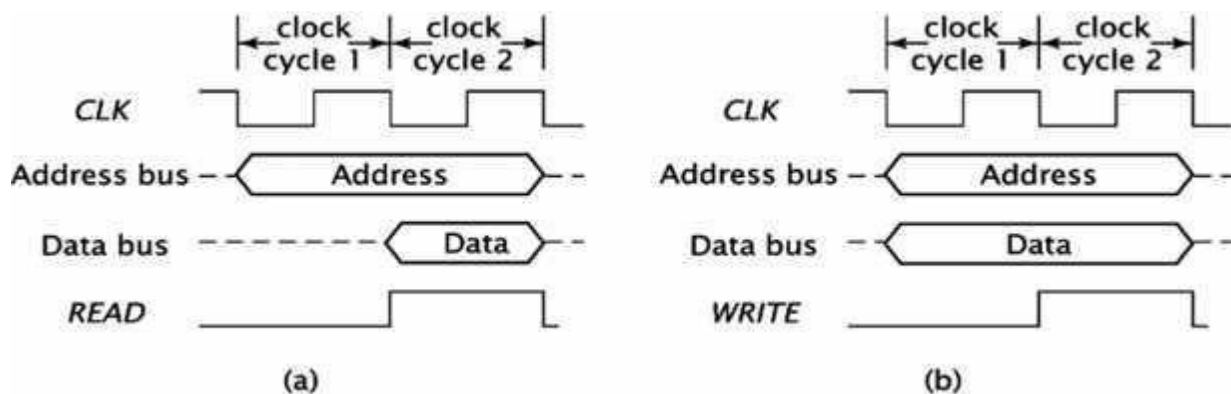


Fig 1.2: Timing diagram for memory read and memory write operations

- In the above figure the top symbol is CLK. This is the computer system clock. The processor uses the system clock to synchronize its operations.
- In fig (a) the microprocessor places the address on to the bus at the beginning of a clock cycle, a 0/1 sequence of clock. One clock cycle later, to allow for memory to decode the address and access its data, the microprocessor asserts the READ control signal. This causes the memory to place its data onto the system data bus. During this clock cycle, the microprocessor reads the data off the system bus and stores it in one of the registers. At the end of the clock cycle it removes the address from the address bus and deasserts the READ signal. Memory then removes the data from the data bus completing the memory read operation.
- In fig(b) the processor places the address and data onto the system bus during the first clock pulse. The microprocessor then asserts the WRITE control signal at the end of the second clock cycle. At the end of the second clock cycle the processor completes the memory write operation by removing the address and data from the system bus and deasserting the WRITE signal.
- I/O read and write operations are similar to the memory read and write operations. Basically the processor may use memory mapped I/O and isolated I/O.
- In memory mapped I/O it follows the same sequence of operations to input data as to read from or write data into memory.

In isolated I/O follow same process but have a second control signal to distinguish between I/O and memory accesses. For example in 8085 microprocessor has a control signal called $\overline{IO}/\overline{MEM}$. The processor set $\overline{IO}/\overline{MEM}$ to 1 for I/O read and write operations and 0 for memory read and write operations.

1.2 CPU ORGANIZATION:

Central processing unit (CPU) is the electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logical, control and input/output (I/O) operations specified by the instructions.

In the computer all the major components are connected with the help of the **system bus**. **Data bus** is used to shuffle data between the various components in a computer system.

To differentiate memory locations and I/O devices the system designer assigns a unique memory address to each memory element and I/O device. When the software wants to access some particular memory location or I/O device it places the corresponding address on the **address bus**. Circuitry associated with the memory or I/O device recognizes this address and instructs the memory or I/O device to read the data from or place data on the data bus. Only the device whose address matches the value on the address bus responds.

The **control bus** is an eclectic collection of signals that control how the processor communicates with the rest of the system. The **read** and **write** control lines control the direction of data on the data bus.

When both contain logic one the CPU and memory-I/O are not communicating with one another. If the read line is low (logic zero) the CPU is reading data from memory (that is the system is transferring data from memory to the CPU). If the write line is low the system transfers data from the CPU to memory.

The CPU controls the computer. It **fetches** instructions from memory, supply the address and control signals needed by the memory to access its data.

Internally, CPU has three sections as shown in the fig below

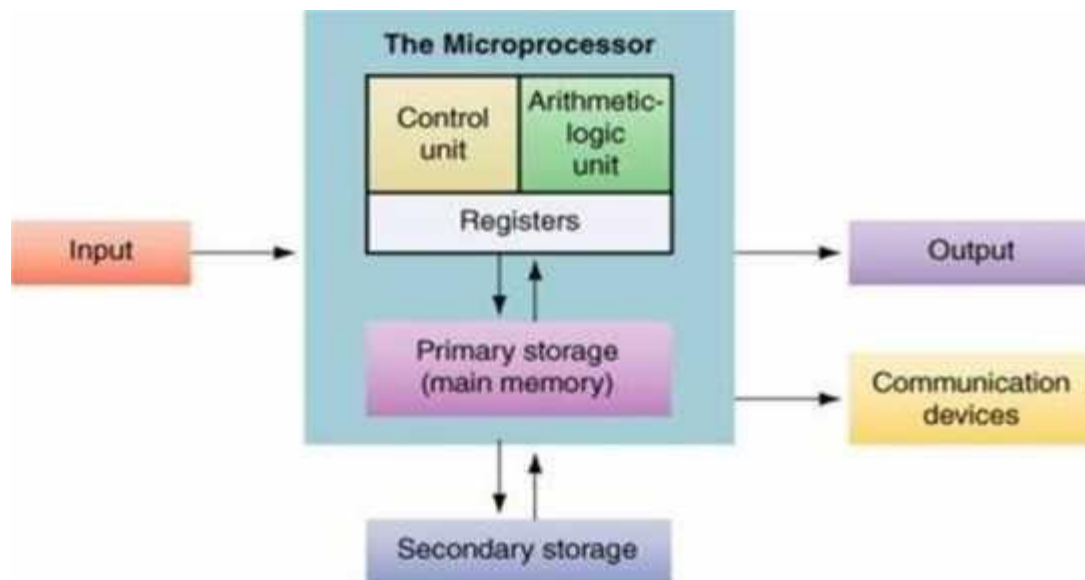


Fig 1.3: CPU Organization

- The register section, as its name implies, includes a set of registers and a bus or other communication mechanism.
- The register in a processor's instruction set architecture are found in the section of the CPU.
- The system address and data buses interact with this section of CPU. The register section also contains other registers that are not directly accessible by the programmer.
- The fetch portion of the instruction cycle, the processor first outputs the address of the instruction onto the address bus. The processor has a register called the "**program counter**".
- The CPU keeps the address of the next instruction to be fetched in this register. Before the CPU outputs the address on to the system bus, it retrieves the address from the program counter register.
- At the end of the instruction fetch, the CPU reads the instruction code from the system data bus.
- It stores this value in an internal register, usually called the "**instruction register**".
- The arithmetic / logic unit (or) ALU performs most arithmetic and logic operations such as adding and ANDing values. It receives its operands from the register section of the CPU and stores its result

back in the register section.

- Just as CPU controls the computer, the control unit controls the CPU. The control unit receives some data values from the register unit, which it used to generate the control signals. This code generates the instruction codes & the values of some flag registers.
- The control unit also generates the signals for the system control bus such as READ, WRITE, IO/M signals.

1.3 MEMORY SUBSYSTEM ORGANIZATION AND INTERFACING:

Memory is the group of circuits used to store data. Memory components have some number of memory locations, each word of which stores a binary value of some fixed length. The number of locations and the size of each location vary from memory chip to memory chip, but they are fixed within individual chip.

The size of the memory chip is denoted as the number of locations times the number of bits in each location. For example, a memory chip of size 512×8 has 512 memory locations, each of which has eight bits. The address inputs of a memory chip choose one of its locations. A memory chip with 2^n locations requires n address inputs.

- View the memory unit as a black box. Data transfer between the memory and the processor takes place through the use of two registers called MAR (Memory Address Register) and MDR (Memory data register).
- MAR is n-bits long and MDR is m-bits long, and data is transferred between the memory and the processor. This transfer takes place over the processor bus.

Internal organization of the memory chips:

- Memory is usually organized in the form of arrays, in which each cell is capable of storing one bit information.
- A possible organization is stored in the fig below...
- Each row of cell constitutes a memory word, and all cells of a row are connected to a common column called word line, which is driven by the address decoder on the chip.
- The cells in each column are connected to sense/write circuit by two bit lines.
- The sense /write circuits are connected to the data input/output lines of the chip.
- During read operation these circuits sense or read the information stored in cells selected by a word line and transmit the information to the output lines.
- During write operation the sense/write circuit receives the input information and store in the cell of selected word.

Types of Memory:

There are two types of memory chips

1. Read Only Memory (ROM)
2. Random Access Memory (RAM)

a) ROM Chips:

ROM chips are designed for applications in which data is read. These chips are programmed with data by an external programming unit before they are added to the computer system. Once it is done the data does not change. A ROM chip always retains its data, even when

Power to chip is turned off so ROM is called **nonvolatile** because of its property. There are several types of ROM chips which are differentiated by how often they are programmed.

- Masked ROM(or) simply ROM
- PROM(Programmed Read Only Memory)
- EPROM(Electrically Programmed Read Only Memory)
- EEPROM(Electrically Erasable PROM)
- Flash Memory
 - A masked ROM or simply ROM is programmed with data as chip is fabricated.
 - The mask is used to create the chip and chip is designed with the required data hardwired in it.

Once chip is designed the data will not change. Figure below shows the possible configuration of the ROM cell.

- Logic 0 is stored in the cell if the transistor is connected to ground at point P, other wise 1 stored.
- A sense circuit at the end of the bit line generates at the high voltage indicating a 1. Data are written into the ROM when it is manufactured.

PROM

- Some ROM designs allow the data to be loaded by the user, thus providing programmable ROM (PROM).
- Programmability is achieved by inserting a fuse at point P in the above fig. Before it is programmed, the memory contains all 0's.
- The user insert 1's at the required locations by burning out the fuse at these locations using high current pulse.
- The fuses in PROM cannot restore once they are blown, PROM's can only be programmed once.

2) EPROM

- EPROM is the another ROM chip allows the stored data to be erased and new data to be loaded. Such an erasable reprogrammable ROM is usually called an EPROM.

- Programming in EPROM is done by charging of capacitors. The charged and uncharged capacitors cause each word of memory to store the correct value.
- The chip is erased by being placed under UV light, which causes the capacitor to leak their charge.

3) EEPROM

- A significant disadvantage of the EPROM is the chip is physically removed from the circuit for reprogramming and that entire contents are erased by the UV light.
- Another version of EPROM is EEPROM that can be both programmed and erased electrically, such chips called EEPROM, do not have to remove for erasure.
- The only disadvantage of EEPROM is that different voltages are need for erasing, writing, reading and stored data.

4) Flash Memory

- A special type of EEPROM is called a flash memory is electrically erase data in blocks rather than individual locations.
- It is well suited for the applications that writes blocks of data and can be used as a solid state hard disk. It is also used for data storage in digital computers.

RAM Chips:

- RAM stands for Random access memory. This often referred to as read/writememory. Unlike the ROM it initially contains no data.
- The digital circuit in which it is used stores data at various locations in the RAMare retrieves data from these locations.
- The data pins are bidirectional unlike in ROM.
- A ROM chip loses its data once power is removed so it is a volatile memory.
- RAM chips are differentiated based on the data they maintain.
 - Dynamic RAM (DRAM)
 - Static RAM (SRAM)

1. Dynamic RAM:

- DRAM chips are like leaky capacitors. Initially data is stored in the DRAM chip, charging its memory cells to their maximum values.
- The charging slowly leaks out and would eventually go too low to represent valid data.
- Before this a refresher circuit reads the content of the DRAM and rewrites data to its original locations.
- DRAM is used to construct the RAM in personal computers.
- DRAM memory cell is shown in the figure below.

2. Static RAM:

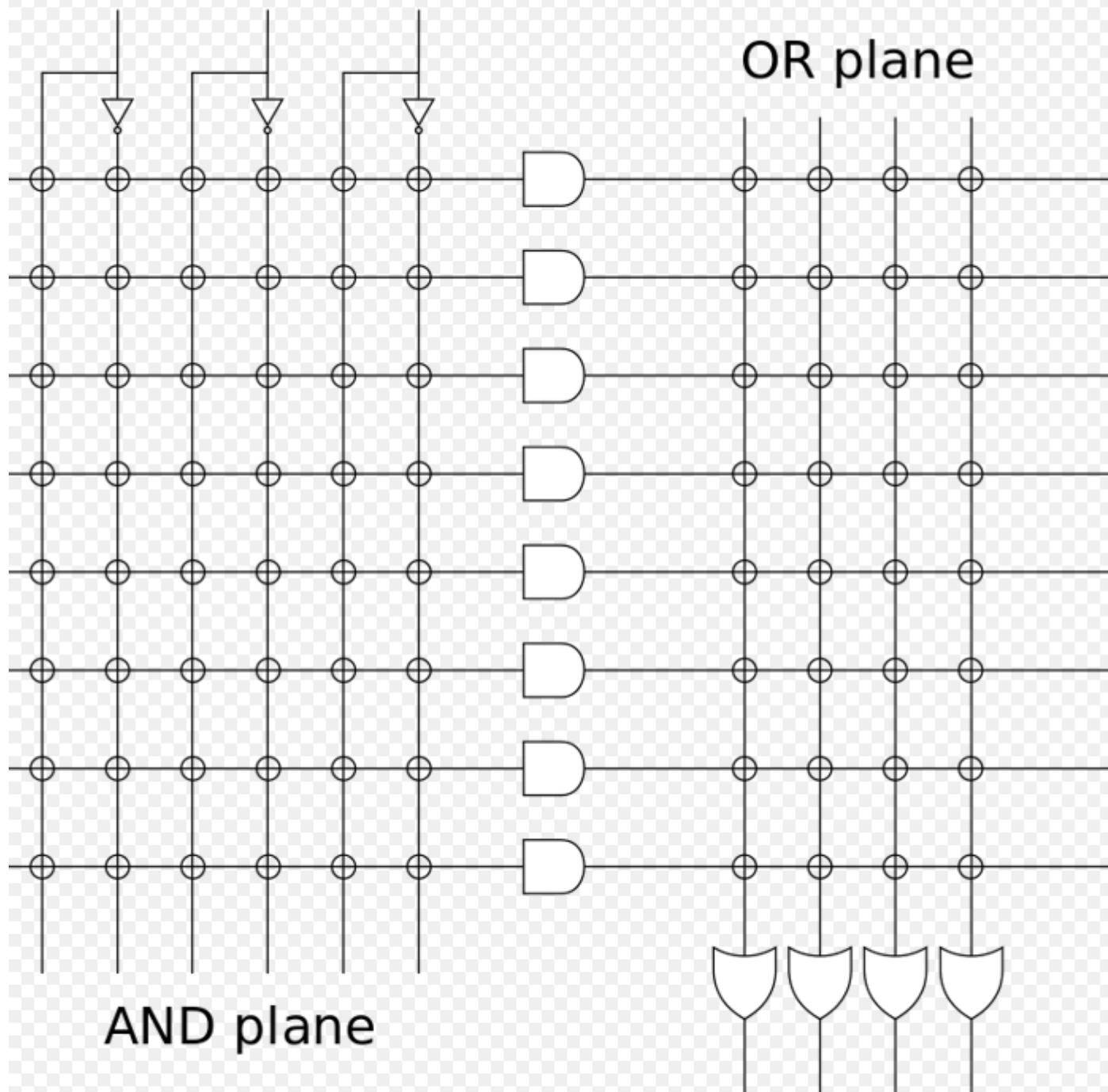
- Static RAM are more likely the register .Once the data is written to SRAM, its contents stay valid it does not have to be refreshed.
- Static RAM is faster than DRAM but it is also much more expensive. Cache memory in the personal computer is constructed from SRAM.
 - Various factors such as cost, speed, power consumption and size of the chip determine how a RAM is chosen for a given application
 - Static RAMs:
 - Chosen when speed is the primary concern.
 - Circuit implementing the basic cell is highly complex, so cost and size are affected.
 - Used mostly in cache memories.
 - Dynamic RAMs:
 - Predominantly used for implementing computer main memories.
 - High densities available in these chips.
 - Economically viable for implementing large memories

Programmable logic array:

A **programmable logic array (PLA)** is a kind of programmable logic device used to implement combinational logic circuits. The PLA has a set of programmable AND gate planes, which link to a set of programmable OR gate planes, which can then be conditionally complemented to produce an output. It has 2^N AND Gates for N input variables, and for M outputs from PLA, there should be M OR Gates, each with programmable inputs from all of the AND gates. This layout allows for a large number of logic functions to be synthesized in the sum of product.

input

OR plane



AND plane

output

Applications:

One application of a PLA is to implement the control over a data path. It defines various states in an instruction set, and produces the next state (by conditional branching). [e.g. if the machine is in state 2, and will go to state 4 if the instruction contains an immediate field; then the PLA should define the actions of the control in state 2, will set the next state to be 4 if the instruction contains an immediate field, and will define the actions of the control in state 4]. Programmable logic arrays should correspond to a state diagram for the system.

Other commonly used programmable logic devices are PAL, CPLD and FPGA.

Note that the use of the word "programmable" does not indicate that all PLAs are field-programmable in fact many are mask-programmed during manufacture in the same manner as a mask ROM. This is particularly true of PLAs that are embedded in more complex and numerous integrated circuits such as microprocessors. PLAs that can be programmed after manufacture are called FPGA (Field-programmable gate array), or less frequently FPLA (Field-programmable logic array).

Programmable Array Logic:

Programmable Array Logic (PAL) is a family of programmable logic device semiconductors used to implement logic functions in digital circuits introduced by Monolithic Memories, Inc. (MMI) in March 1978. MMI obtained a registered trademark on the term PAL for use in "Programmable Semiconductor Logic Circuits". The trademark is currently held by Lattice Semiconductor.

PAL devices consisted of a small PROM (programmable read-only memory) core and additional output logic used to implement particular desired logic functions with few components.

Using specialized machines, PAL devices were "field-programmable". PALs were available in several variants:

- "One-time programmable" (OTP) devices could not be updated and reused after initial programming (MMI also offered a similar family called HAL, or "hard array logic", which were like PAL devices except that they were mask-programmed at the factory.).
- UV erasable versions (e.g.: PALCxxxxx e.g.: PALC22V10) had a quartz window over the chip die and could be erased for re-use with an ultraviolet light source just like an EPROM.

In most applications, electrically-erasable GALs are now deployed as pin-compatible direct replacements for one-time programmable PALs.

Content-addressable memory:

Content-addressable memory (**CAM**) is a special type of computer memory used in certain very-high-speed searching applications. It is also known as associative memory **or associative storage** and compares input search data (tag) against a table of stored data, and returns the address of matching data (or in the case of associative memory, the matching data).

Field-Programmable Gate Array:

A **field-programmable gate array (FPGA)** is an integrated circuit designed to be configured by a customer or a designer after manufacturing – hence the term "field-programmable". The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an Application-Specific

Integrated Circuit (ASIC). Circuit diagrams were previously used to specify the configuration, but this is increasingly rare due to the advent of electronic design automation tools.

A **field-programmable gate array (FPGA)** is an integrated circuit designed to be configured by a customer or a designer after manufacturing – hence the term "field-programmable". The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an Application-Specific Integrated Circuit (ASIC). Circuit diagrams were previously used to specify the configuration, but this is increasingly rare due to the advent of electronic design automation tools.

FPGAs contain an array of programmable logic blocks, and a hierarchy of "reconfigurable interconnects" that allow the blocks to be "wired together", like many logic gates that can be inter-wired in different configurations. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory.¹ Many FPGAs can be reprogrammed to implement different logic functions, allowing flexible reconfigurable computing as performed in computer software.