

# **Jaypee Institute of Information Technology, Noida**

## **LAB MANUAL** **DIGITAL SYSTEMS LAB (18B15EC213)**



## **Department of Electronics and Communication Engineering**

## **PREFACE**

Digital Systems Laboratory has been designed for students at the undergraduate level to familiarize them with the fundamentals of digital electronics, communication systems, and digital signal processing.

The laboratory manual contains laboratory exercises based on MATLAB. MATLAB, which is distributed by the MathWorks, Inc., is chosen as the platform for these exercises because it is widely used by the practitioners in the field and is capable of realizing interesting systems. In MATLAB, one can specify the sequence of steps that construct a signal or operate on a signal to produce the output signal.

The lab exercises are divided in three parts. The first part is based on experiments related to digital electronics, such as the realization of digital logic gates, adders, multiplexers, and decoders. The second part is related to experiments related to digital communication systems, such as sampling, quantization, and modulation. Finally, in the third part topics such as continuous time and discrete time signals, discrete Fourier transform, and filter design have been included, in order to familiarize the students with the concepts of digital signal processing.

At the end of the course, the students will be able to handle the different aspects of a digital system.

### **GUIDELINES FOR STUDENTS**

- Be punctual in coming to the lab and be sincere in doing your lab work.
- Before starting the lab experiment you must go through the lab manual of that experiment. Get your doubts clear and then start.
- No food, beverages, chewing tobacco and gums is allowed in the lab.
- At the end of each lab period, shut down the computer.
- Keep the work area neat and clean; arrange the chairs before leaving the laboratory.
- Don't misuse the IT facility available in the laboratory.
- Don't bring your laptop in the lab without permission of the lab incharge.



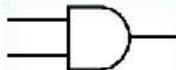

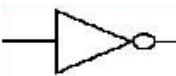

## EXPERIMENT – 1

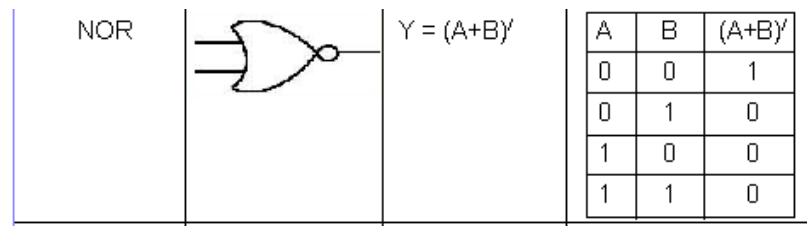
**AIM:** Write Matlab programs for the verification of truth tables of basic logic gates and their realization using universal logic gates.

The experiment is done in two parts as follows:

### PART 1: Verification of truth tables of basic logic gates

**THEORY:** Digital systems are said to be constructed by using logic gates. These gates are the AND, OR, NOT, NAND, and NOR gates. The basic operations are described with the aid of truth tables in Figure 1.1.

Gate Name	Symbol	Notation	Truth table															
AND		$Y = A.B$ OR $Y = AB$	<table><tr><th>A</th><th>B</th><th>A.B</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>	A	B	A.B	0	0	0	0	1	0	1	0	0	1	1	1
A	B	A.B																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR		$Y = A + B$	<table><tr><th>A</th><th>B</th><th>A+B</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>	A	B	A+B	0	0	0	0	1	1	1	0	1	1	1	1
A	B	A+B																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
NOT		$Y = A'$	<table><tr><th>A</th><th>Y</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	Y	0	1	1	0									
A	Y																	
0	1																	
1	0																	
NAND		$Y = (A.B)'$	<table><tr><th>A</th><th>B</th><th><math>(A.B)'</math></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>	A	B	$(A.B)'$	0	0	1	0	1	1	1	0	1	1	1	0
A	B	$(A.B)'$																
0	0	1																
0	1	1																
1	0	1																
1	1	0																

**Figure 1.1: Basic Logic Gates****MATLAB PROGRAM:**

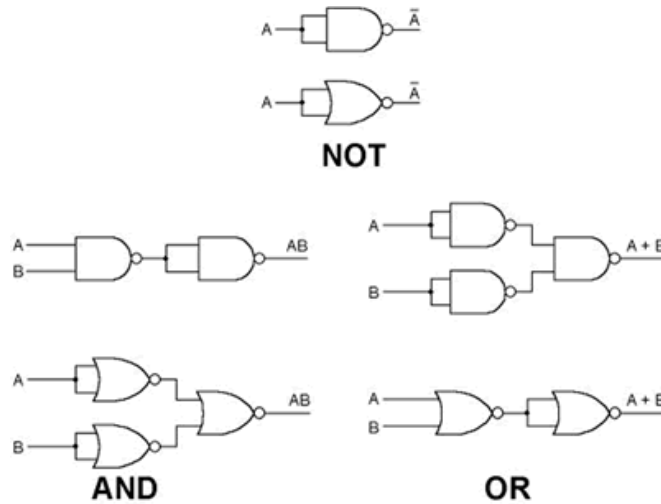
```
clearall
display'Program for verification of basic logic gates'
display'For verification of gates ';
a = input('Array of input A in terms of [0,1]'); %enter the logical values of the input
b = input('Array of input B in terms of [0,1]'); %enter the logical values of the input
c = a & b; %Logical AND
d = a | b; %%Logical OR
e = ~a; %Inverted A
%The 'a' & 'b' can be described as [0 1 0 1], or any other entry in matrix form. The variables
'c' and 'd' truth table entries of the output of AND and OR gates. The output 'e' is inverter
operation.
```

**OBSERVATIONS:** Note down the values obtained on command window.

**PART 2: Realization of basic gates using universal logic gates**

**THEORY:** A universal gate is a gate which can implement any Boolean function without the need to use any other gate type. The NAND and NOR gates are universal gates. In practice, this

is advantageous since NAND and NOR gates are economical and easier to fabricate and are the basic gates used in all IC digital logic families. The implementation of gates using universal gates can be seen in Figure 1.2.



**Figure 1.2: Implementation of basic gates using universal gates**

### **MATLAB PROGRAM:**

a) Create functions for NAND and NOR

```
function Z = nand(X,Y)
```

```
%This function has been created for implementation of logical NAND
```

```
%operation
```

```
k= X & Y;
```

```
Z= ~k;
```

```
end
```

```
function Z = nor(X,Y)
```

```
%This function has been created for implementation of logical NOR
```

```
%operation
```

```
k= X | Y;
```

$Z = \sim k$ ;

end

b) After writing the above functions, following script is written for implementation of basic gates with universal gates.

% Create basic gates from universal gates

a = input('Array of input A in terms of [0,1] with size 4 x 1'); %enter the logical values of the input A

b = input('Array of input b in terms of [0,1] with size 4 x 1'); %enter the logical values of the input A

k = nand(a,b);

v = nand(k,k); %AND via NAND

a\_bar = nand(a,a); %Inverter using NAND

b\_bar = nand(b,b);

x\_nand = nand(a\_bar,b\_bar); % OR gate via NAND

l = nor(a,b);

x\_nor\_or = nor(l,l);

a\_bar\_nor = nor(a,a); %Inverter using NAND

b\_bar = nor(b,b);

x\_nor\_and = nor(a\_bar,b\_bar); % OR gate via NAND

**OBSERVATIONS:** Note down the values obtained on command window.

**RESULTS:** The programs for basic logic gates and universal gates have been simulated in MATLAB and truth tables have been verified.

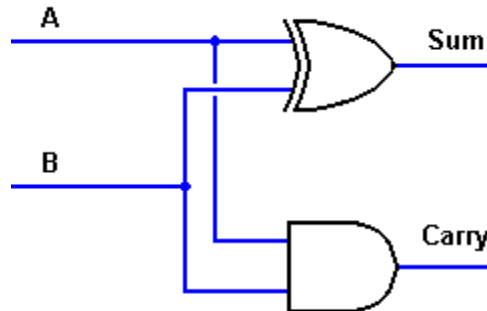
**EXPERIMENT – 2**

**AIM:** Write Matlab programs for half-adder, half-subtractor, full-adder, and full-subtractor.

**1. HALF-ADDER**

**THEORY:** *Half-adder* is a basic combinational digital circuit built from logic gates. The half adder adds two one-bit binary numbers (A, B). The outputs are the Sum (S) and Carry (C).

A half adder can be constructed from one Ex-OR gate and one AND gate. The logic diagram of half adder is as shown in Figure 2.1. As shown, half adder has two inputs, A and B, and two outputs, Sum (S) and Carry (C).



**Figure 2.1: Circuit Diagram of Half Adder**

**TRUTH TABLE:** The truth table of half adder is given as follows:

Input A	Input B	Output S	Output C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



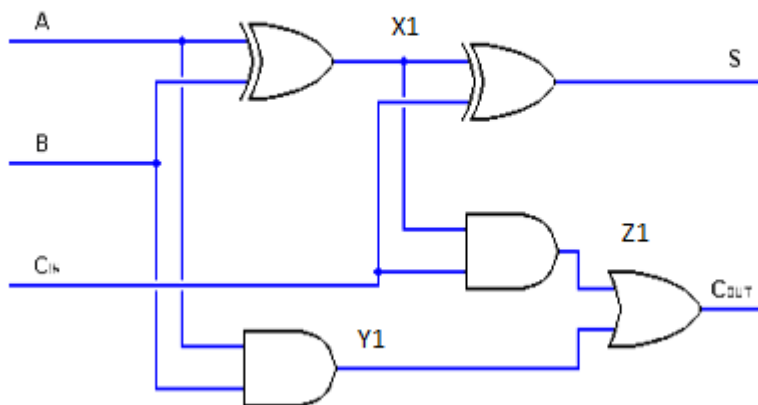
**MATLAB PROGRAM:**

```
clearall
display'half adder'
A = input('Array of input X'); %enter the logical values of the input A
B = input('Array of input Y'); %enter the logical values of the input B
Sum = xor(A,B);
Carry= A & B;
```

**OBSERVATIONS:** Note down the values obtained on command window.

**2. FULL-ADDER**

**THEORY:** *Full-adder* circuit adds three one-bit binary numbers. The circuit diagram of full adder is shown in the Figure 2.2. The output of Ex-OR gate is called SUM (S), while the output of the AND gate is the CARRY (Cout). The AND gate produces a high output only when both inputs are high. The Ex-OR gate produces a high output if either input, but not both, is high. The 1-bit full adder circuit has a provision to add the carry generated (Cin) from the lower order bits.



**Figure 2.2: Circuit Diagram of Full Adder**

**TRUTH TABLE:** The truth table of full adder is given as follows:

Input A	Input B	Input Cin	Output S	Output Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

**MATLAB PROGRAM:**

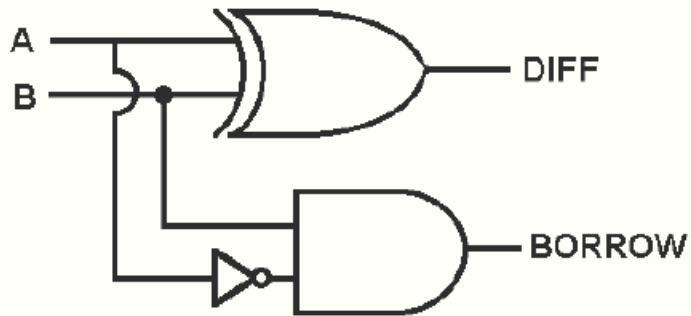
```
clearall
display'Full Adder'
A =input('Array of input A ');%enter the logical values of the input A
B = input('Array of input B');%enter the logical values of the input B
Cin =input('Array of input C');%enter the logical values of the input Cin
X1 = xor(A,B); % Intermediate signal
S = xor(X1,Cin);
Y1= A & B; % Intermediate signal
Z1= X1 &Cin; % Intermediate signal
Cout = Z1 | Y1;
```

**OBSERVATIONS:** Note down the values obtained on command window.

### **3. HALF-SUBTRACTOR**

**THEORY:** *Half-subtractor* is a combinational circuit which is used to perform subtraction of two bits. It has two inputs, X (minuend) and Y (subtrahend) and two outputs D (difference) and

B (borrow). A half subtractor can be realized from one Ex-OR gate and one AND gate. The logic diagram of half-subtractor is shown in Figure 2.3. The half-subtractor has two inputs, A and B, and two Outputs, DIFF and BORROW. This circuit performs the binary subtraction of B from A (A-B, recalling that in binary  $0 - 1 = 1$  borrow 1).



**Figure 2.3: Circuit diagram of Half-Subtractor**

**TRUTH TABLE:** The truth table of half-subtractor is given as follows:

Input A	Input B	Output DIFF	Output BORROW
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

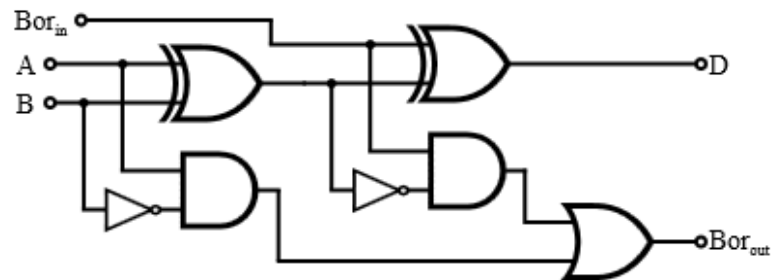
**MATLAB PROGRAM:**

```
clearall
display'Full Subtractor'
A = input('Array of input A '); %enter the logical values of the input A
B = input('Array of input B '); %enter the logical values of the input B
D = xor(A,B);
X= ~A; % Intermediate signal
Bor = B & X;
```

**OBSERVATIONS:** Note down the values obtained on command window.

#### 4. FULL SUBTRACTOR

**THEORY:** *Full-subtractor* is a combinational circuit which is used to perform subtraction of three bits. It has three inputs, X (minuend) and Y(subtrahend) and Bi (borrow in) and two outputs D (difference) and Bo (borrow out). The logic diagram of full-subtractor is shown in Figure 2.4.



**Figure 2.4: Circuit diagram of Full-Subtractor**

**TRUTH TABLE:** The truth table of half-subtractor is given as follows:

Input			Output	
A	B	Bin	D	Bout
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

#### **POST LAB EXERCISE:**

- Assignment for students to write code for full-subtractor.

**RESULTS:** The programs for half-adder, half-subtractor, full-adder, and full-subtractor have been simulated in MATLAB and truth tables have been verified.

### **EXPERIMENT – 3**

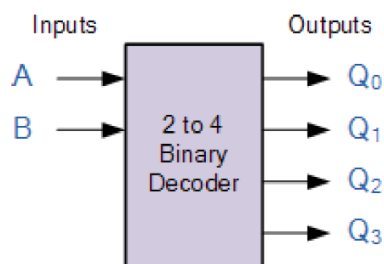
**AIM:** Write Matlab programs for the design of 2-to-4 decoder and 3-to-8 decoder.

**THEORY:** The name “*Decoder*” means to translate or decode coded information from one format into another, so a digital decoder transforms a set of digital input signals into an equivalent decimal code at its output. Binary decoder is a digital logic device that has  $N$ -bit input code and therefore it will be possible to represent  $2^N$  possible values. If a binary decoder receives  $N$  inputs (usually grouped as a single Binary or Boolean number) it activates one and only one of its  $2^N$  outputs based on corresponding input code with all other outputs deactivated.

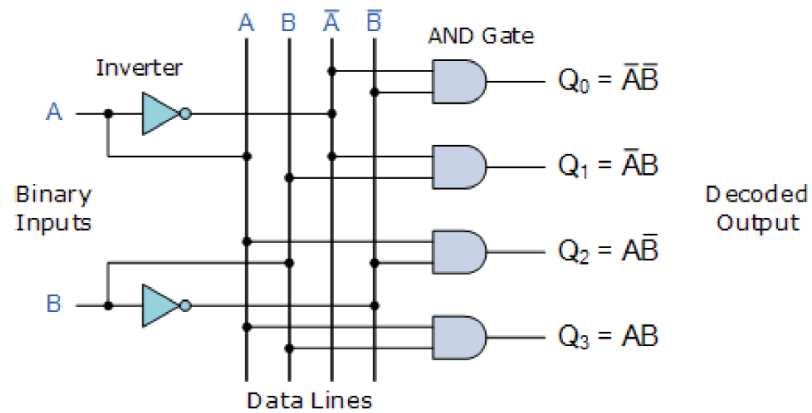
#### 1. 2-to-4 Decoder

**TRUTH TABLE:** The truth table of 2-to-4 decoder is given as follows:

A	B	$Q_0$	$Q_1$	$Q_2$	$Q_3$
0	0	<u>1</u>	0	0	0
0	1	0	<u>1</u>	0	0
1	0	0	0	<u>1</u>	0
1	1	0	0	0	<u>1</u>



**Figure 3.1: Block Diagram of 2-to-4 Decoder**



**Figure 3.2: Logic Circuit of 2-to-4 Decoder**

### **MATLAB PROGRAM:**

% 2x4 Decoder using Logic Gates

clc

closeall

clearall;

a1=input('input MSB input of 2x4 decoder a1=');

a0=input('input LSB input of 2x4 decoder a0=');

d0=(~a0)&(~a1);

d1=(a0)&(~a1);

d2=(~a0)&(a1);

d3=(a0)&(a1);

result=[d3 d2 d1 d0];

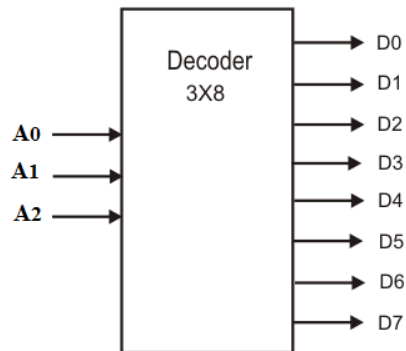
sprintf('decoder output is \n%d \n%d \n%d \n%d',result)

**OBSERVATIONS:** The decoder output is obtained for particular pair of inputs, A & B.

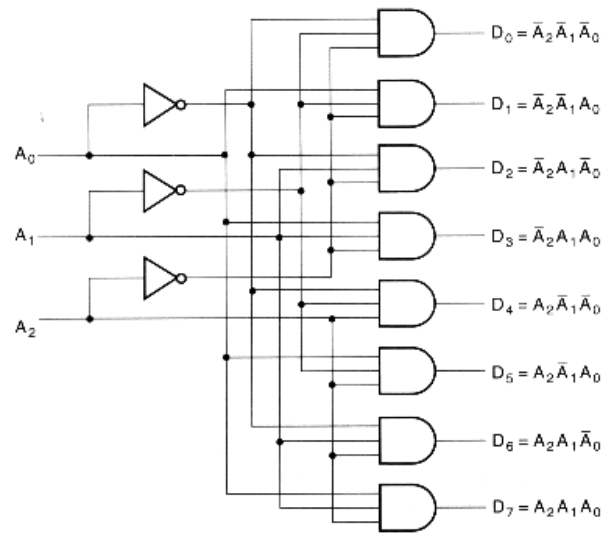
## **2. 3-to-8 Decoder**

**TRUTH TABLE:** The truth table of 3-to-8 decoder is given as follows:

$A_2$	$A_1$	$A_0$	$D_7$	$D_6$	$D_5$	$D_4$	$D_3$	$D_2$	$D_1$	$D_0$
0	0	0	0	0	0	0	0	0	0	<u>1</u>
0	0	1	0	0	0	0	0	0	<u>1</u>	0
0	1	0	0	0	0	0	0	<u>1</u>	0	0
0	1	1	0	0	0	0	<u>1</u>	0	0	0
1	0	0	0	0	0	<u>1</u>	0	0	0	0
1	0	1	0	0	<u>1</u>	0	0	0	0	0
1	1	0	0	<u>1</u>	0	0	0	0	0	0
1	1	1	<u>1</u>	0	0	0	0	0	0	0



**Figure 3.3: Block Diagram of 3-to-8 Decoder**



**Figure 3.4: Logic Circuit of 3-to-8 Decoder**

### MATLAB PROGRAM:

% 3x8 Decoder using Logic Gates

clc

closeall

clearall;

a2=input('Enter MSB of 3x8 decoder a2=');

a1=input('Enter 2nd bit of 3x8 decoder a1=');

a0=input('Enter LSB of 3x8 decoder a0=');

d0=(~a0)&(~a1)&(~a2);

d1=(a0)&(~a1)&(~a2);

d2=(~a0)&(a1)&(~a2);

d3=(a0)&(a1)&(~a2);

d4=(~a0)&(~a1)&(a2);

d5=(a0)&(~a1)&(a2);

d6=(~a0)&(a1)&(a2);

d7=(a0)&(a1)&(a2);

clc



```
display('decoder input is');  
[a2 a1 a0]  
display('decoder output is');  
[d7 d6 d5 d4 d3 d2 d1 d0]
```

**OBSERVATIONS:** The decoder output is obtained on command window for particular sequence of inputs,  $A_0$ ,  $A_1$ , and  $A_2$ .

**RESULTS:** The programs for 2-to-4 and 3-to-8 decoders have been simulated in MATLAB and truth tables have been verified.

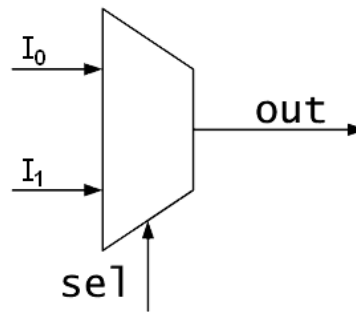


**EXPERIMENT – 4**

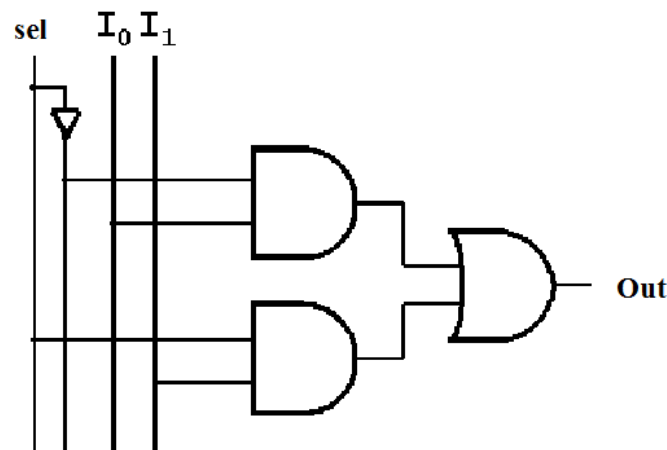
**AIM:** Write Matlab programs for the design of 2-to-1, 4-to-1, and 8-to-1 multiplexers.

**THEORY:** A multiplexer (or Mux) is a device that selects one of the several input signals and forwards the selected input into a single line. A multiplexer of  $2^n$  inputs has  $n$  select lines, which are used to select which input line to send to the output. Multiplexers are mainly used to increase the amount of data that can be sent over the network within a certain amount of time and bandwidth. A multiplexer is also called a data selector.

**2:1 MULTIPLEXER:** A 2-to-1 MUX has 2 input lines (namely  $I_0$ ,  $I_1$ ), one selection line (sel), and one output line (out). The schematic of a 2-to-1 Multiplexer is as shown in Figure 4.1. Further, the schematic can be represented using gates as shown in Figure 4.2.



**Figure 4.1: Schematic of a 2-to-1 Multiplexer**



**Figure 4.2: 2-to-1 Multiplexer using gates**

**TRUTH TABLE:** The truth table of 2-to-1 MUX is given as follows:

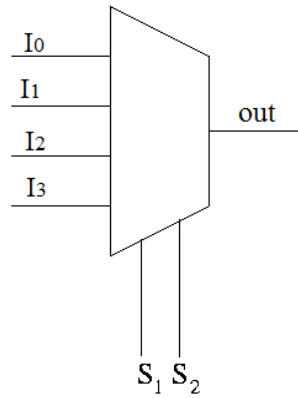
sel	Out
0	$I_0$
1	$I_1$

**MATLAB PROGRAM:**

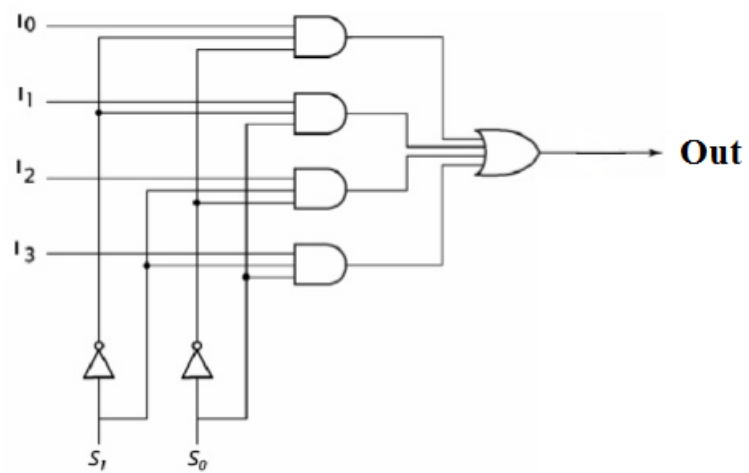
```
%2:1 Multiplexer
clearall;
display ('2x1 Multiplexer')
I0 = input('Enter I0');
I1 = input('Enter I1');
for i=1:3
sel = input('Enter sel');
if(sel==0)
out = I0;
elseif (sel==1)
out = I1;
end
end
```

**OBSERVATIONS:** Note down the values obtained on command window.

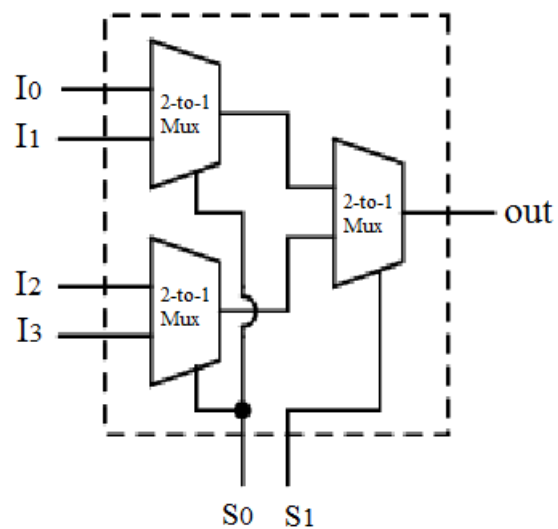
**4:1 MULTIPLEXER** : A 4-to-1 MUX has 4 input lines (namely  $I_0, I_1, I_2, I_3$ ), two selection lines ( $S_0, S_1$ ), and one output line (out). The schematic of a 4-to-1 multiplexer is as shown in Figure 4.3. Further, the schematic can be represented using gates as shown in Figure 4.4. A 4-to-1 MUX can be constructed using 2-to-1 MUX as shown in Figure 4.5.



**Figure 4.3: Schematic of a 4-to-1 Multiplexer**



**Figure 4.4: 4-to-1 Multiplexer using gates**



**Figure 4.5: 4-to-1 Multiplexer using 2-to-1 Multiplexers**

**TRUTH TABLE:** The truth table of 4-to-1 MUX is given as follows:

$S_1$	$S_0$	Out
0	0	$I_0$
0	1	$I_1$
1	0	$I_2$
1	1	$I_3$

**MATLAB PROGRAM:**

% First create function for 2-TO-1 Multiplexer

```
function f=mux( I0,I1,sel)
```

```
if(sel==0)
```

```
f= I0;
```

```
elseif (sel==1)
```

```
f= I1;
```

```
end
```

%After writing the above function, following script is written for implementation of 4: 1

```
closeall;
```

```
clearall;
```

```
I0 = input('Enter I0');
```

```
I1 = input('Enter I1');
```

```
I2 = input('Enter I2');
```

```
I3 = input('Enter I3');
```

```
for i=1:5
```

```
sel = input('Enter sel');
```

```
f02= mux(I0,I2,sel(1));
```

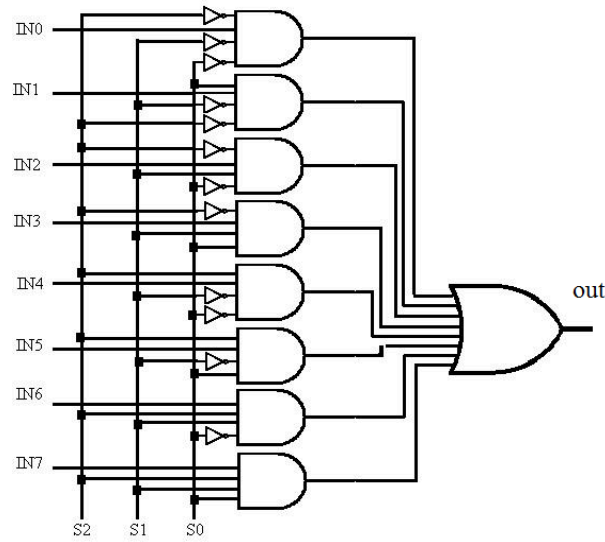
```
f13= mux(I1,I3,sel(1));
```

```
f= mux(f02,f13,sel(2))
```

```
end
```

**OBSERVATIONS:** Note down the values obtained on command window.

**8:1 MULTIPLEXER** : A 8-to-1 MUX has 8 input lines (namely  $I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7$ ), three selection lines ( $S_0, S_1, S_2$ ), and one output line (out). The logic diagram of 8-to-1 Mux is as shown in Figure 4.6.



**Figure 4.6: Logic diagram of 8-to-1 Multiplexer**

**TRUTH TABLE:** The truth table of 8-to-1 MUX is given as follows:

S2	S1	S0	out
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	$I_7$

**MATLAB PROGRAM:**

%8:1 Multiplexer: 8:1 Multiplexer can be implemented using seven 2:1 multiplexers.

```
closeall;
clearall;
I0 = input('Enter I0');
I1 = input('Enter I1');
I2 = input('Enter I2');
I3 = input('Enter I3');
I4 = input('Enter I4');
I5 = input('Enter I5');
I6 = input('Enter I6');
I7 = input('Enter I7');
for i=1:10
    sel = input('Enter sel');
    f01= mux(I0,I1,sel(3));
    f23= mux(I2,I3,sel(3));
    f45= mux(I4,I5,sel(3));
    f67= mux(I6,I7,sel(3));
    f0123= mux(f01,f23,sel(2));
    f4567= mux(f45,f67,sel(2));
    f= mux(f0123,f4567,sel(1))
end;
```

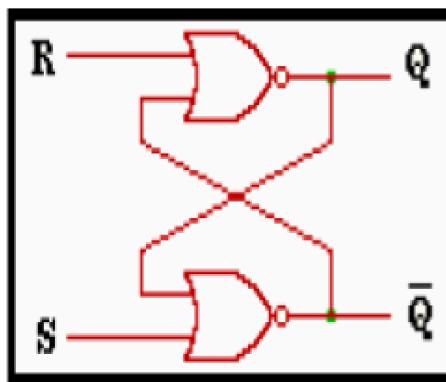
**OBSERVATIONS:** Note down the values obtained on command window.

**RESULTS:** The program for 2-to-1, 4-to-1, and 8-to-1 multiplexers have been simulated in MATLAB and truth tables have been verified.



**EXPERIMENT – 5(a)****AIM: Realization of SR Latch using MATLAB-Simulink.**

**THEORY:** The flip–flop, also known as a bistable multivibrator, varies from the other digital logic circuits that have been examined since the flip–flop has two stable states or memory. Since these devices have the ability to remember their current state, they are the basic building blocks for registers and Static RAM memory. The difference between the latch and the flip–flop lies in the method used to change its output state. The latch is a level sensitive circuit that will change its output state due to a change of states on its inputs. As a result, the latch is an asynchronous device. The logic symbol of the RS latch is shown in **Figure 5.1**. The RS latch has an invalid input condition. This invalid condition is not allowed since it forces both the Q output and its complement to go high at the same time and the final state cannot be predicted when the invalid input is removed.

**Fig 5.1 Logic circuit of SR Latch**

Input S R		Output Q Q'	Condition
0	0	Q <sub>0</sub> Q <sub>0</sub> '	No Change
0	1	0 1	Reset
1	0	1 0	Set
1	1	1 1	Invalid

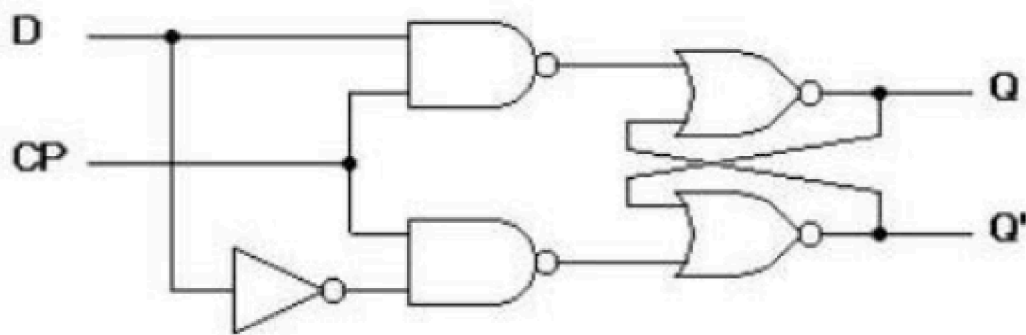
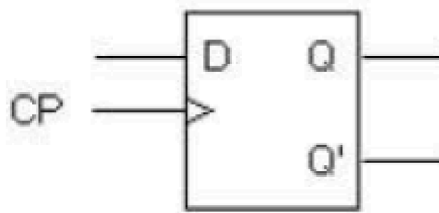
**Fig 5.2 Truth Table of SR latch**

**OBSERVATIONS:** The latch output is obtained on command window for particular sequence of inputs.

**RESULTS:** The programs for SR latch have been simulated in MATLAB and truth tables have been verified.

**EXPERIMENT – 5(b)****AIM-** Realization of D Flip-Flop using MATLAB-Simulink.

**THEORY-** D flip flop is actually a slight modification of the clocked SR flip-flop. From the figure you can see that the D input is connected to the S input and the complement of the D input is connected to the R input. The D input is passed on to the flip flop when the value of CP is „1“. When CP is HIGH, the flip flop moves to the SET state. If it is „0“, the flip flop switches to the CLEAR state.

**Fig 6.1 Logic circuit of D Flip Flop****Fig 6.2 Symbol of D Flip Flop**

Q	D	Q(t+1)
0	0	0
0	1	1
1	0	0
1	1	1

**Fig 6.3 Transition Table of D Flip Flop**

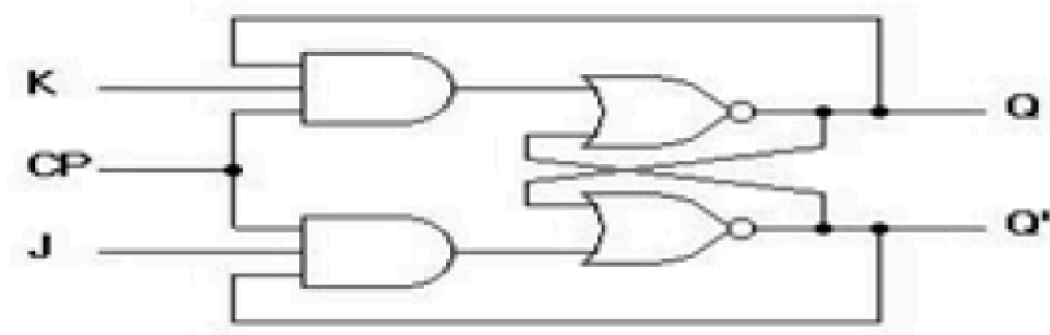
**OBSERVATIONS:** The D flip flop output is obtained on command window for particular sequence of inputs.

**RESULTS:** The program for D flip flop has been simulated in MATLAB and truth tables have been verified.

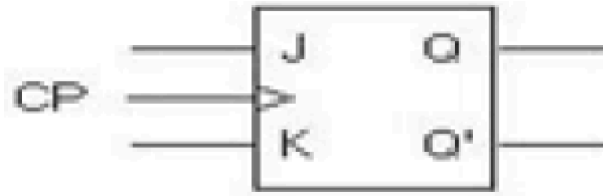
### **EXPERIMENT – 5(c)**

**AIM : - Realization of JK Flip-Flop using MATLAB-Simulink.**

**THEORY:-** A J-K flip flop can also be defined as a modification of the S-R flip flop. The only difference is that the intermediate state is more refined and precise than that of a S-R flip flop. The behavior of inputs J and K is same as the S and R inputs of the S-R flip flop. The letter J stands for SET and the letter K stands for CLEAR. When both the inputs J and K have a HIGH state, the flip-flop switch to the complement state. So, for a value of  $Q = 1$ , it switches to  $Q=0$  and for a value of  $Q = 0$ , it switches to  $Q=1$ . The circuit includes two 3-input AND gates. The output Q of the flip flop is returned back as a feedback to the input of the AND along with other inputs like K and clock pulse [CP]. So, if the value of CP is „1“, the flip flop gets a CLEAR signal and with the condition that the value of Q was earlier 1. Similarly output Q' of the flip flop is given as a feedback to the input of the AND along with other inputs like J and clock pulse [CP]. So the output becomes SET when the value of CP is 1 only if the value of Q' was earlier 1. The output may be repeated in transitions once they have been complimented for  $J=K=1$  because of the feedback connection in the JK flip-flop. This can be avoided by setting a time duration lesser than the propagation delay through the flip-flop. The restriction on the pulse width can be eliminated with a master-slave or edge-triggered construction.



**Fig 7.1 Logic circuit of JK Flip Flop**



**Fig 7.2 Symbol of JK Flip Flop**

Q	J	K	Q(t+1)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

**Fig 7.3 Transition Table of JK Flip Flop**

**OBSERVATIONS:** The JK flip flop output is obtained on command window for particular sequence of inputs.

**RESULTS:** The program for JK flip flop have been simulated in MATLAB and truth tables have been verified.

## **EXPERIMENT – 6**

**AIM:** Write Matlab programs for generation of elementary continuous time signals and discrete time signals.

**THEORY:** A continuous-time signal  $x(t)$  is a function of an independent variable  $t$ , that is time. A continuous -time signal  $x(t)$  is a mathematically continuous function, and is defined continuously in the time domain.

A discrete-time signal  $y(n)$  is a function of an independent variable  $n$ , that is an integer. It is important to note that a discrete-time signal is not defined at instants between two successive samples. Also it is incorrect to think that  $y(n)$  is not defined for non-integer values of  $n$ .  $y(n)$  can be obtained from sampling the continuous signal  $y(t)$ , then  $y(n) = y(nT)$ , where  $T$  is the sampling period (i.e. the time between successive samples).

### **PROCEDURE:**

- A. Steps for plotting continuous time signals are as follows:
  - i. Assign the value 'dt' very small.
  - ii. Vary the time 't' between two points with an increment of 'dt'.
- B. Steps for converting continuous time signal to the discrete time signal are as follows:
  - i. Assign the value of sampling time,  $T$ .
  - ii. Vary the time integer 'n' between two integer points with an increment of 1.
  - iii. The discrete version can be obtained as:  $y(n) = y(nT)$ .

### **Useful Commands in MATLAB**

**plot:** `plot(t, x(t));` plots the continuous function  $x(t)$  vs.  $t$ .

**stem:** `stem(n, x(n));` plots the discrete function  $x(n)$  vs.  $n$ .

**Stem:** `stem (n, Y(n));` plots the data sequence  $Y$  as stems from the  $x$  axis terminated with circles for the data value. If  $Y$  is a matrix then each column is plotted as a separate series.

**IN LAB EXERCISE:**

Plot the following elementary continuous time signals:

- 1) Unit stepfunction
- 2) Unit rampfunction
- 3) Sinusoidalfunction
- 4) Exponentialfunction
- 5) rectfunction
- 6) sincfunction

**MATLAB PROGRAM:**

```
figure(1);  
t1 = -2*pi:pi/360:2*pi;  
x1 = 1*(t1>=0)+0*(t1<0);  
plot(t1,x1)  
xlim([-2,2]);  
ylim([0,2]);  
xlabel('t');  
ylabel('Continuous Unit Step Function');
```

```
figure(2);  
t2 = -2*pi:pi/120:2*pi;  
x2 = t2.*(t2>=0)+0*(t2<0);  
plot(t2,x2)  
xlim([-2,5]);  
ylim([0,4]);  
xlabel('t');  
ylabel('Continuous Unit Ramp Function');
```

```
figure(3);  
t3 = linspace(-1,1);
```



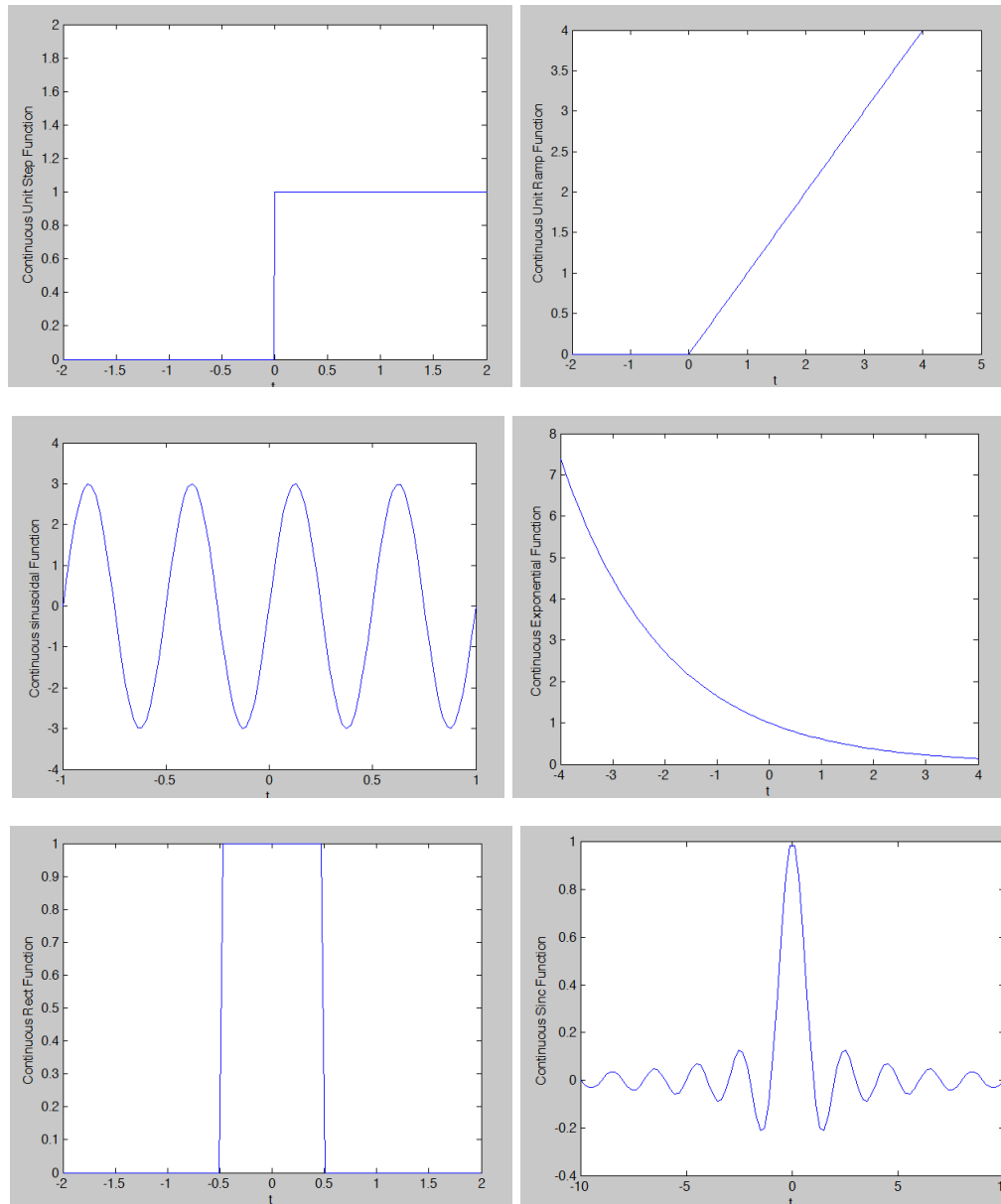
```
f = 200;  
x3 = 3*sin(2*pi*200*t3);  
plot(t3,x3)  
ylim([-4,4]);  
xlabel('t');  
ylabel('Continuous sinusoidal Function');
```

```
figure(4);  
t4 = linspace(-4,4);  
a = 0.5;  
x4 = exp(-a*t4);  
plot(t4,x4)  
xlabel('t');  
ylabel('Continuous Exponential Function');
```

```
figure(5)  
t5 = linspace(-2,2);  
x5 = 1*(t5<(1/2)& t5>(-1/2))+0*(t5>(1/2)&t5<(-1/2))+(1/2)*(t5==(1/2));  
plot(t5,x5)  
xlabel('t');  
ylabel('Continuous Rect Function');
```

```
figure(6)  
t6 = linspace(-10,10);  
x6 = sinc(t6);  
plot(t6,x6)  
xlabel('t');  
ylabel('Continuous Sinc Function');
```

**OBSERVATIONS:** The following plots are obtained:



**RESULTS:** The programs for generation of elementary continuous time signals have been simulated in MATLAB.

**POST LAB EXERCISE:** Plot the corresponding discrete time functions for all elementary signals mentioned in lab exercise.

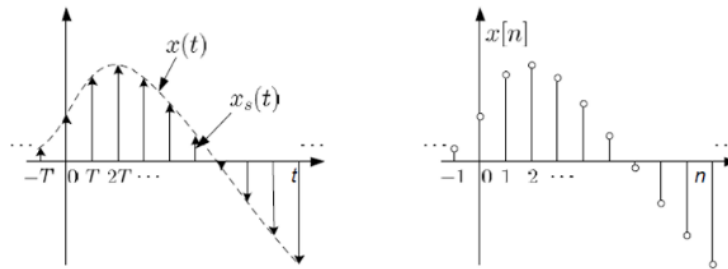
## EXPERIMENT – 7

**AIM:** Write Matlab program to study the sampling and reconstruction process.

The experiment is done in two parts as follows:

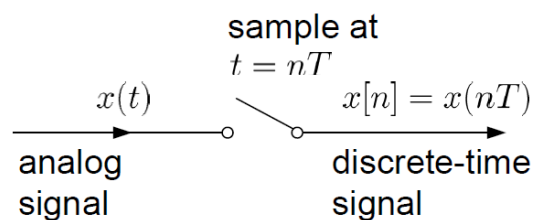
### PART 1: Sampling

**THEORY:** *Sampling* is a process of converting a continuous-time signal  $x(t)$  into a discrete time sequence  $x[n]$ , as shown in Figure 5.1.



**Figure 9.1: Continuous time  $x(t)$  and Discrete time  $x[n]$  signals**

$x[n]$  is obtained by extracting  $x(t)$  every  $T$  sec, where  $T$  is known as the sampling period or interval, as shown in Figure 7.2.



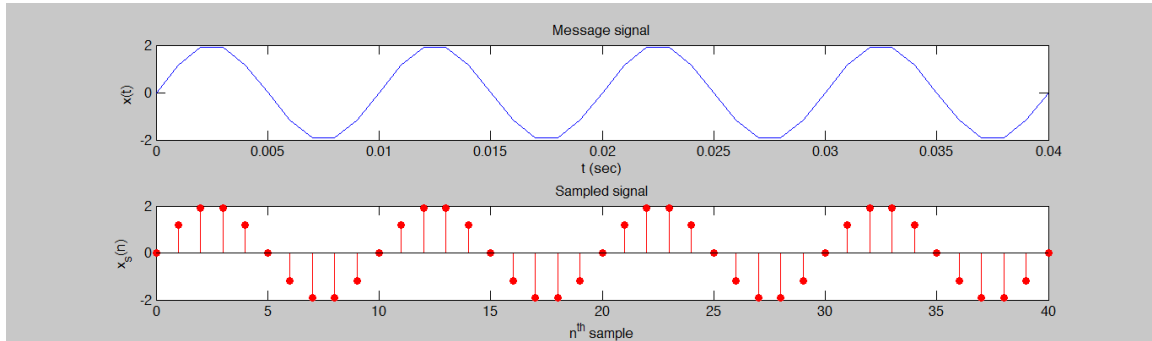
**Figure 9.2: Sampling Process**

**MATLAB PROGRAM:**

```
%Defining message signal  $x(t)=mag*\sin(2*\pi*fm*t)$ 
mag=2; %Magnitude of message signal
fm=100; %Frequency of message signal
tm=1/fm; %Time period of message signal in sec
n_cycles=4; %no. of cycles
t_time=n_cycles*tm; %Total duration of signal in sec
t=0:0.001:t_time; %Time interval for plotting
x=mag*sin(2*pi*fm*t); %message signal
subplot(3,1,1); %plotting message signal
plot(t,x);
title('Message signal')
xlabel('t (sec)');
ylabel('x(t)');

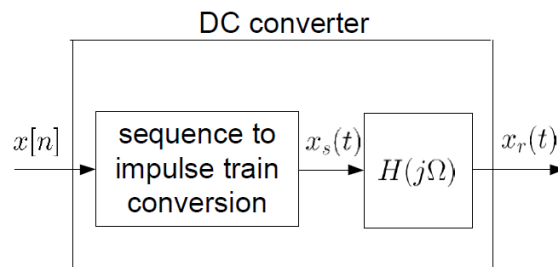
%Sampling
num_points=1000; %sampling rate i.e. samples/sec
T=1/num_points; %sampling period/interval T
ts = 0:T:t_time; %sampling instants for total duration of signal
ns = 0:length(ts)-1; %no. of samples for total duration of signal
x_samples = mag*sin(2*pi*fm/num_points*ns); %sample values
subplot(3,1,2); %Plotting the samples
axis=0:1:(length(x_samples)-1);
stem(axis,x_samples,'filled','r');
title('Sampled signal')
xlabel('n^{th} sample');
ylabel('x_s(n)');
```

**OBSERVATIONS:** Message signal and sampled signal plots are obtained as follows:



## **PART 2:Reconstruction**

**THEORY:***Reconstruction* is a process of transforming  $x[n]$  back to  $x(t)$  using a DC converter, as shown in Figure 9.3.



**Figure 9.3: Reconstruction of  $x[n]$  to obtain  $x(t)$**

In Figure 7.3,  $H(j\Omega)$  is the inverse Fourier transform of  $h(t) = \text{sinc}\left(\frac{t}{T}\right)$ . Therefore,  $x_r(t)$  is:

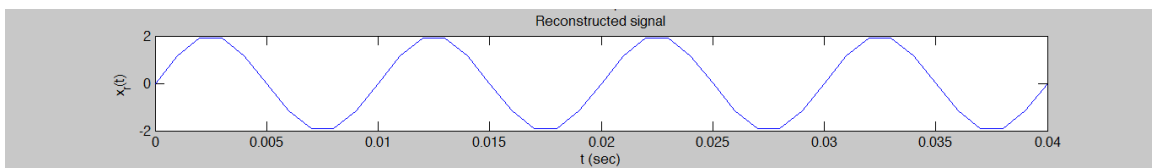
$$\begin{aligned}
 x_r(t) &= x_s(t) \otimes h(t) \\
 &= \left( \sum_{k=-\infty}^{\infty} x[k] \delta(t - kT) \right) \otimes h(t) \\
 &= \int_{-\infty}^{\infty} \sum_{k=-\infty}^{\infty} x[k] \delta(\tau - kT) h(t - \tau) d\tau \\
 &= \sum_{k=-\infty}^{\infty} x[k] h(t - kT) \\
 &= \sum_{k=-\infty}^{\infty} x[k] \text{sinc}\left(\frac{t - kT}{T}\right)
 \end{aligned}$$

which holds for all real values of  $t$ .

**MATLAB PROGRAM:**

```
%Reconstruction
x_recon=0; %initialising reconstructed signal
for k=0:length(x_samples)-1
    l=k:-1:-(length(t)-1)+k; %taking ns number of past and future samples
    x_recon=x_recon+x_samples(k+1)*sinc(l); %reconstruction using interpolation
end
subplot(3,1,3); %Plotting the reconstructed signal
plot(t,x_recon);
title('Reconstructed signal')
xlabel('t (sec)');
ylabel('x_r(t)');
```

**OBSERVATIONS:** The reconstructed signal is obtained as follows:



**RESULTS:** The programs for sampling and reconstruction have been simulated in MATLAB and necessary graphs are plotted.

## **EXPERIMENT – 8**

**AIM:** Write Matlab program to study the Quantization process of Sinusoid Signals.

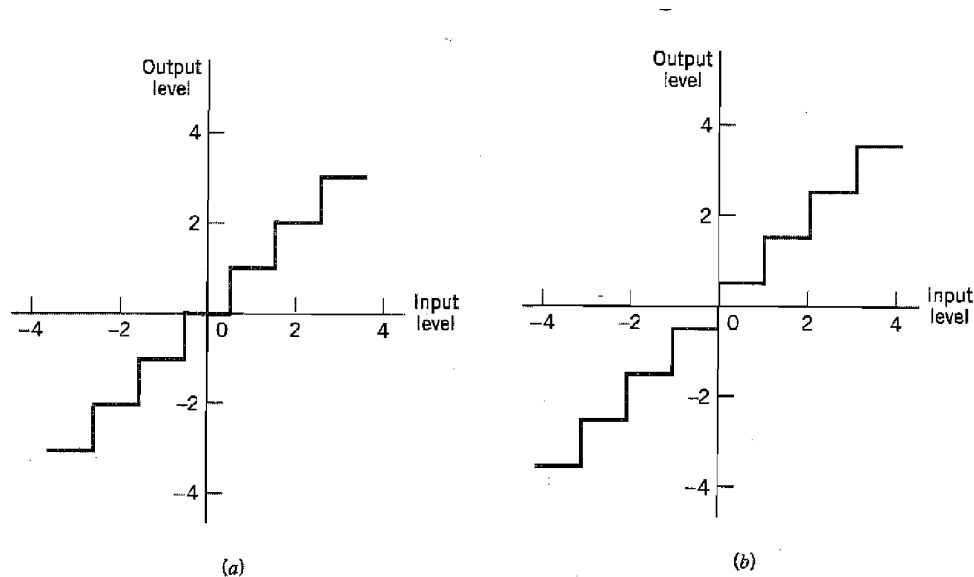
**THEORY:** Quantization is the process of approximating a continuous range of values (or a very large set of possible discrete values) by a relatively-small set of discrete symbols or integer values.

A common use of quantization is in the conversion of a discrete signal (a sampled continuous signal) into a digital signal by quantizing. Both of these steps (sampling and quantizing) are performed in analog-to-digital converters.

After the sampling we have a sequence of numbers which can theoretically still take on any value on a continuous range of values. Because this range is continuous, there are infinitely many possible values for each number. In order to be able to represent each number from such a continuous range, we would need an infinite number of digits - something we don't have. Instead, we must represent our numbers with a finite number of digits, that is: after discretizing the time-variable, we now have to discretize the amplitude-variable as well. This discretization of the amplitude values is called quantization. Assume, our sequence takes on values in the range between  $-1 \dots +1$ . Now assume that we must represent each number from this range with just two decimal digits: one before and one after the point. Our possible amplitude values are therefore:  $-1.0, -0.9, \dots, -0.1, 0.0, 0.1, \dots, 0.9, 1.0$ . These are exactly 21 distinct levels for the amplitude and we will denote this number of quantization levels with  $N_q$ . Each level is a step of 0.1 higher than its predecessor and we will denote this quantization step-size as  $q$ . Now we assign to each number from our continuous range that quantization level which is closest to our actual amplitude: the range  $-0.05 \dots +0.05$  maps to quantization level 0.0, the range  $0.05 \dots 0.15$  maps to 0.1 and so on. That mapping can be viewed as a piecewise constant function acting on our continuous amplitude of the signal.

Quantization can be of uniform or non-uniform type. In uniform quantization, the quantization levels are uniformly spaced; otherwise, the quantizer is non-uniform. The quantizer characteristics can be of midrise or mid-tread type. Figure 10.1 (a) shows input-output characteristics of a uniform quantizer of the mid-tread type. In mid-tread type quantizer, the origin lies in the middle of the tread of the staircase like graph. Further, in midrise type as shown in Figure 10.1 (b), the origin the graph lies in the middle of a rising part of the staircase

like graph. It can be noticed that, both mid-tread and midrise graphs are symmetric about the origin.



**Figure 10.1: Transfer characteristic of (a) mid-tread quantizer (b) mid-rise type quantizer**

In a mid-tread type quantizer, if input signal values lie in the interval  $-\Delta/2$  to  $+\Delta/2$  then quantized value of the signal is zero (i.e. the mean value of this interval). In general, quantized value of the input signal lies in the interval  $m\Delta/2$  to  $(m+2)\Delta/2$  is  $(m+1)\Delta/2$ . Similarly, mid-rise type quantizer, if input signal values lie in the interval  $0$  to  $\Delta$  then quantized value of the signal is  $+\Delta/2$  (i.e. the mean value of this interval). In general, quantized value of the input signal lies in the interval  $m\Delta$  to  $(m+1)\Delta$  is  $(2m+1)\Delta/2$ . In both of the quantizers, the quantization error at any point lies in the interval  $-\Delta/2$  to  $+\Delta/2$ .

### **PROCEDURE:**

- (i) Generate a sine wave of frequency 100Hz, and desired amplitude for duration of 0.02 seconds
- (ii) Sample it at sampling  $f_s \geq 2f_m$  frequency and plot it.
- (iii) Select the desired quantization levels and calculate the total number of quantization levels.
- (iv) Quantize each sample value using mid-rise type quantizer.
- (v) Plot the quantized signal
- (vi) Calculate quantization error and plot it.



- (vii) Also draw the plot of quantization error vs input signal. (For smoother graph increase the sampling frequency)
- (viii) Increase the number of quantization levels and observe the effect on quantization error.

### **MATLAB PROGRAM:**

```

clc;
closeall;
clearall;
disp('Generate 0.02-second sine wave of 100 Hz');
amp= input('Enter the amplitude of sinusoidal signal = ');
fs = input('Enter the Sampling Frequency = '); % Sampling rate
T = 1/fs; % Sampling interval
t = 0:T:0.02; % Duration of 0.02 second
sig1=amp* sin (2*pi*100.*t);
min_sig=min(sig1);
max_sig=max(sig1);
sig = amp* sin (2*pi*100.*t); % Generate the sinusoid
nbits= input('enter the number of bits quantizer = ');
quint_level=2^nbits; % midrise type quantizer
s=(max_sig-min_sig)/quint_level;
for jj=min_sig:s:max_sig
sig(sig<=jj+s& sig>=jj)=((2*jj)+s)/2;
end
figure
stem(sig);
title('Plot of Sampled Signal')
xlabel('Samples Number')
ylabel('Samples')
figure
qun_error=sig1-sig;
plot(qun_error);

```

```
title('Plot of Quantization Error')  
xlabel('Samples Number')  
ylabel('Quantization Error')
```

**OBSERVATIONS:** The following values are entered on command window:

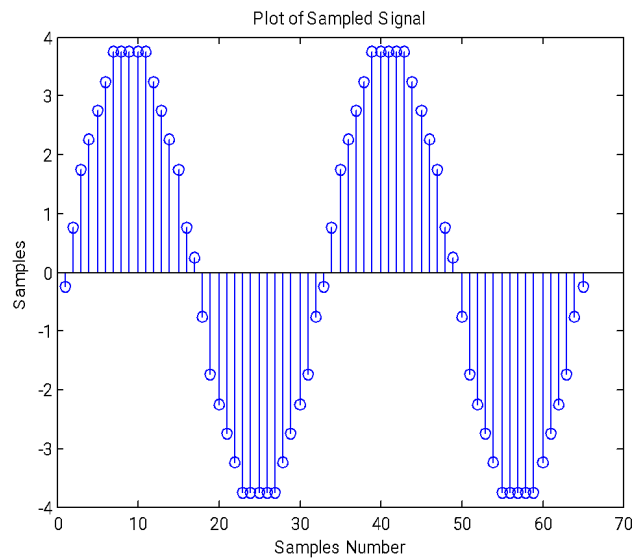
Generate 0.02-second sine wave of 100 Hz

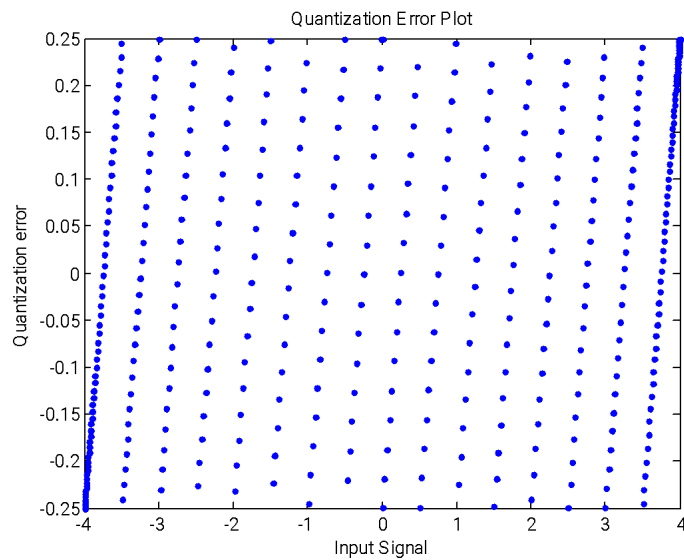
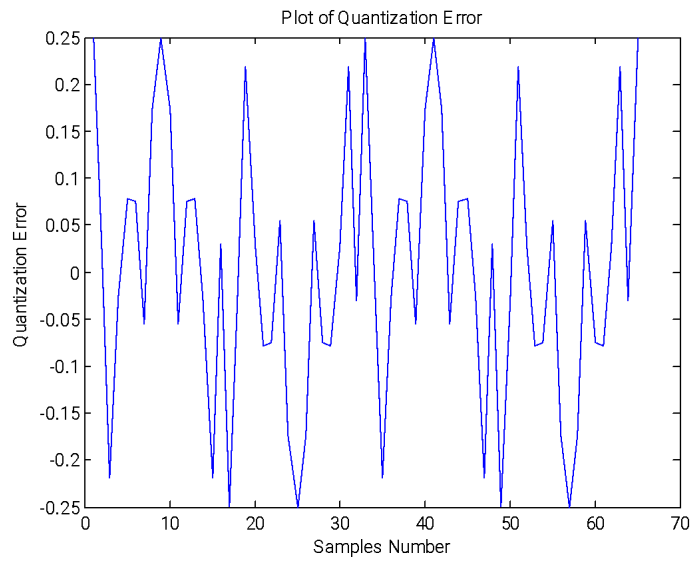
Enter the amplitude of sinusoidal signal = 4

Enter the Sampling Frequency = 3200

enter the number of bits quantizer = 4

The following plots are obtained:





**RESULTS:** The program for Quantization process of Sinusoid Signals has been simulated in MATLAB and necessary graphs are plotted.



## EXPERIMENT – 9

**AIM:** Write Matlab program to compute Discrete Fourier Transform (DFT) and Inverse Discrete Fourier Transform (IDFT) for the spectral analysis of signals.

**THEORY:** We know that aperiodic finite energy signals have continuous spectra, given by

$$X(\omega) = \sum_{n=-\infty}^{n=\infty} x(n) * e^{-j\omega n}$$

In case of a finite length sequence  $x(n)$ ,  $0 \leq n \leq L-1$ , only  $L$  values of  $X(\omega)$  over its period, called the frequency samples, are sufficient to determine  $x(n)$  and hence  $X(\omega)$ . This leads to the concept of discrete Fourier transform (DFT) which is obtained by periodic sampling of  $X(\omega)$ .

We often compute a higher point ( $N$  point) DFT where  $N > L$ . This is because padding the sequence  $x(n)$  with  $N-L$  zeros and computing an  $N$  point DFT results in a “better display” of the Fourier transform  $X(\omega)$ .

To summarize, the formulas are:

$$\text{DFT: } X(k) = \sum_{n=0}^{N-1} x(n) e^{-j \frac{2\pi kn}{N}}, \quad k = 0, 1, \dots, N-1$$

$$\text{IDFT: } x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{j \frac{2\pi kn}{N}}, \quad n = 0, 1, \dots, N-1$$

Further, the general IDFT matrix equations are:

$$\text{DFT: } X = [W_N]x$$

$$\text{IDFT: } x = \frac{1}{N} [W_N^*]X$$

$$\text{where } W_N = e^{-j2\pi/N}$$

$$x = [x(0)x(1)\dots\dots\dots x(N-1)]^T$$

$$X = [X(0)X(1)\dots\dots\dots X(N - 1)]^T$$

$$W_N = \begin{matrix} & \xrightarrow{\quad n \quad} \\ \begin{matrix} \downarrow k \\ \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & W_N & W_N^2 & \dots & W_N^{N-1} \\ 1 & W_N^2 & W_N^4 & \dots & W_N^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & W_N^{N-1} & W_N^{2(N-1)} & \dots & W_N^{(N-1)^2} \end{bmatrix} \end{matrix} \end{matrix}$$

### **LAB EXERCISE:**

1. Find the DFT-transform of the following signals

- $x(n) = \{1, 2, 3, 2\}$
- $x(n) = \{1, 1, 0, 0, 0, 0, 0\}$

2. Find the IDFT of the following signals

- $X(K) = \{4, 2, 0, 4\}$
- $X(K) = \{1, 0, 1, 0\}$

### **MATLAB PROGRAM:**

% To compute N pt. DFT of finite duration signal

N=input('Enter pt. of DFT = ');

x=input('Enter sequence = ');

L=length(x);

X=(1./N).\*fft(x,N);

subplot(3,1,1),stem(0:L-1,x)

title('x[n]')

xlabel('n')

ylabel('x[n]')

subplot(3,1,2),stem(0:N-1,abs(X));

title('Magnitude Spectrum')

xlabel('k')

```
ylabel('X[k]')
subplot(3,1,3),stem(0:N-1,phase(X))
title('Phase Spectrum')
xlabel('k')
ylabel('\angle X[k]')

% To compute N pt. IDFT
N=input('Enter pt. of IDFT =');
X=input('Enter X[k] =');
x=(N).*ifft(X,N);
subplot(3,1,1),stem(0:length(X)-1,X)
title('X[k]')
xlabel('k')
ylabel('X[k]')
subplot(3,1,2),stem(0:N-1,real(x));
title('Real of x[n]')
xlabel('n')
ylabel('Re {x[n]}')
subplot(3,1,3),stem(0:N-1,imag(x))
title('Phase Spectrum')
xlabel('n')
ylabel('Imag {x[n]}')
```

**OBSERVATIONS:** Plot the magnitude spectrum and phase spectrum of the signals.

**RESULTS:** The programs for computing N-point DFT and IDFT have been simulated in MATLAB.





## **EXPERIMENT – 10**

**AIM:** Write Matlab programs to study the binary phase shift keying (BPSK) and frequency shift keying modulation (FSK) process.

### **1. BINARY PHASE SHIFT KEYING (BPSK)**

**THEORY:** BPSK is a digital modulation scheme that conveys data by changing, or modulating, the phase of a reference signal (the carrier wave). PSK uses a finite number of phases, each assigned a unique pattern of binary digits. Usually, each phase encodes an equal number of bits. Each pattern of bits forms the symbol that is represented by the particular phase. The demodulator, which is designed specifically for the symbol-set used by the modulator, determines the phase of the received signal and maps it back to the symbol it represents, thus recovering the original data.

In a coherent binary PSK system, the pair of signal  $S_1(t)$  and  $S_2(t)$  used to represent binary symbols 1 & 0 are defined by :

$$S_1(t) = \sqrt{2E_b/T_b} \cos(2\pi f_c t),$$

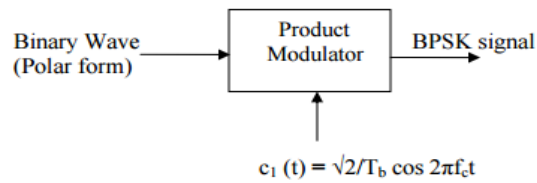
$$S_2(t) = \sqrt{2E_b/T_b} \cos(2\pi f_c t + \pi) = -\sqrt{2E_b/T_b} \cos(2\pi f_c t), \quad \text{where } 0 \leq t < T_b,$$

where  $E_b$  = Transmitted signed energy for bit

$T_b$  = Bit interval

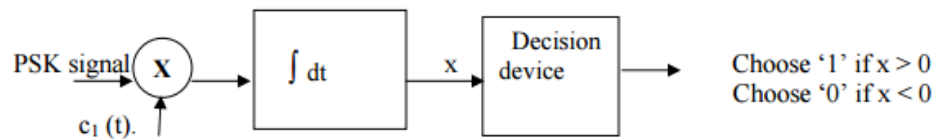
$f_c$  = Carrier frequency =  $n/T_b$  for some fixed integer  $n$ .

**BPSK Transmitter:** A BPSK transmitter is as shown in Figure 11.1 The input binary symbols are represented in polar form with symbols 1 & 0 which are represented by constant amplitude levels  $\sqrt{E_b}$  &  $-\sqrt{E_b}$ . This binary wave is multiplied by a sinusoidal carrier  $c_1(t)$  in a product modulator. The result is a BPSK signal.



**Figure 11.1: BPSK Transmitter**

**BPSK Receiver:** A BPSK receiver is as shown in Figure 11.2. The received BPSK signal is applied to a correlator which is also supplied with a locally generated reference signal  $c_1(t)$ . The correlated output is compared with a threshold of zero volts. If  $x > 0$ , the receiver decides in favour of symbol 1. If  $x < 0$ , it decides in favour of symbol 0.



**Figure 11.2: BPSK Receiver**

### **PROCEDURE:**

PSK modulation is carried out follows:

- i. Generate carrier signal.
- ii. Start FOR loop
- iii. Generate binary data, message signal in polar form
- iv. Generate PSK modulated signal.
- v. Plot message signal and PSK modulated signal.
- vi. End FOR loop.
- vii. Plot the binary data and carrier.

PSK demodulation is carried out follows:

- i. Start FOR loop Perform correlation of PSK signal with carrier to get decision variable
- ii. Make decision to get demodulated binary data. If  $x > 0$ , choose '1' else choose '0'
- iii. Plot the demodulated binary data.

**MATLAB PROGRAM:**

```
% PSK modulation
clc;
clearall;
closeall;
%generate carrier signal
Tb=1;
t=0:Tb/100:Tb;
fc=2;
c=sqrt(2/Tb)*sin(2*pi*fc*t);
%generate message signal
N=8;
m=rand(1,N);
t1=0;
t2=Tb
for i=1:N
    t=[t1:.01:t2]
    if m(i)>0.5
        m(i)=1;
        m_s=ones(1,length(t));
    else
        m(i)=0;
        m_s=-1*ones(1,length(t));
    end
end
```

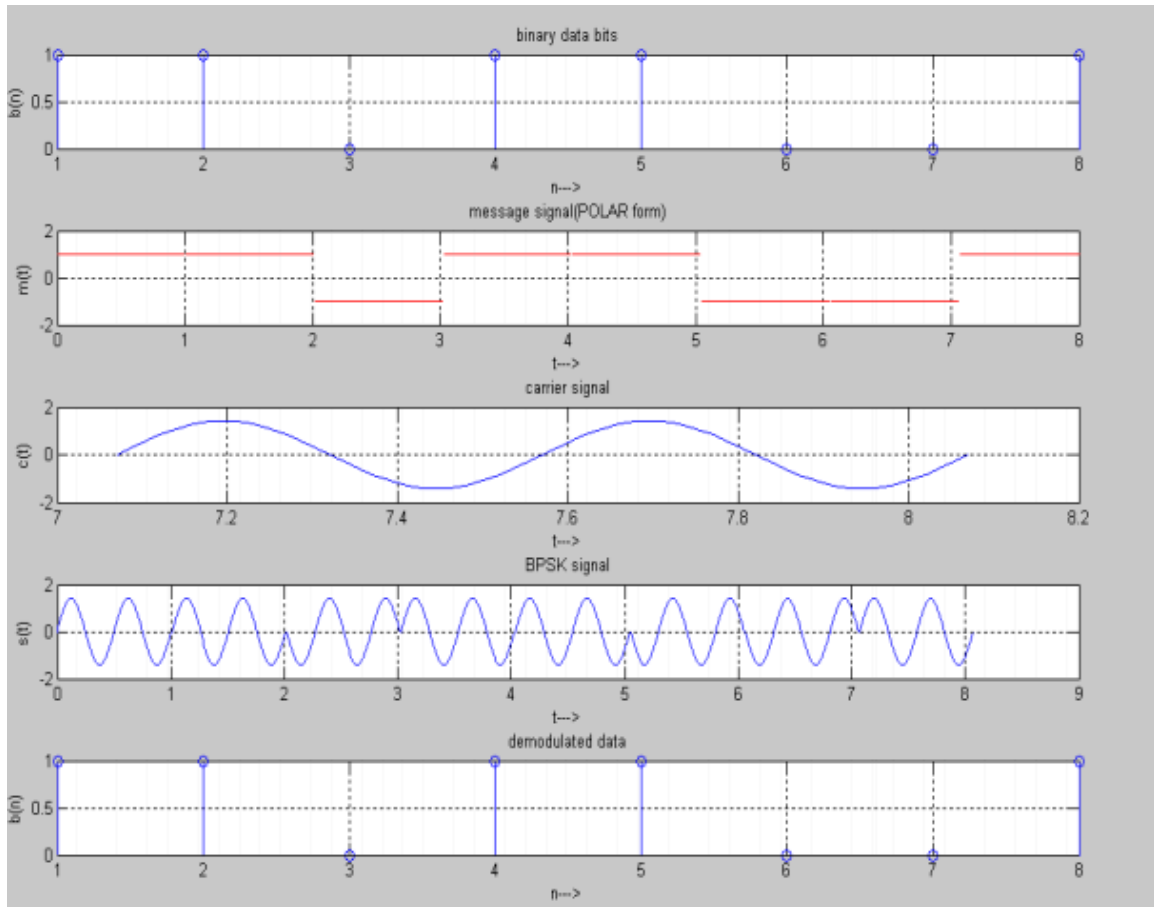
```

message(i,:)=m_s;
%product of carrier and message signal
bpsk_sig(i,:)=c.*m_s;
%Plot the message and BPSK modulated signal
subplot(5,1,2);
axis([0 N -2 2]);
plot(t,message(i,:), 'r');
title('message signal(POLAR form)');
xlabel('t--->');
ylabel('m(t)');
grid on;
hold on;
subplot(5,1,4); plot(t,bpsk_sig(i,:));
title('BPSK signal');
xlabel('t--->'); ylabel('s(t)');
grid on; hold on;
t1=t1+1.01; t2=t2+1.01;
end
hold off
%plot the input binary data and carrier signal
subplot(5,1,1); stem(m);
title('binary data bits');
xlabel('n--->'); ylabel('b(n)');
grid on;
subplot(5,1,3); plot(t,c);
title('carrier signal'); xlabel('t--->'); ylabel('c(t)');
grid on;
% PSK Demodulation
t1=0;
t2=Tb
for i=1:N
t=[t1:.01:t2]

```

```
%correlator
x=sum(c.*bpsk_sig(i,:));
%decision device
if x>0
    demod(i)=1;
else
    demod(i)=0;
end
t1=t1+1.01;
t2=t2+1.01;
end
%plot the demodulated data bits
subplot(5,1,5);
stem(demod);
title('demodulated data');
xlabel('n--->');
ylabel('b(n)');
grid on;
```

**OBSERVATIONS:** The following plots are obtained:



## 2. FREQUENCY SHIFT KEYING (FSK)

**THEORY:** FSK is a frequency modulation scheme in which digital information is transmitted through discrete frequency changes of a carrier wave. The simplest FSK is binary FSK (BFSK). BFSK uses a pair of discrete frequencies to transmit binary (0s and 1s) information. With this scheme, the "1" is called the mark frequency and the "0" is called the space frequency.

In binary FSK system, symbol 1 & 0 are distinguished from each other by transmitting one of the two sinusoidal waves that differ in frequency by a fixed amount as follows:

$$S_i(t) = \sqrt{2E_b/T_b} \cos(2\pi f_i t), \quad 0 \leq t \leq T_b,$$

$$S_i(t) = 0, \quad \text{elsewhere,}$$

where  $i = 1, 2$

$E_b$  = Transmitted energy/bit

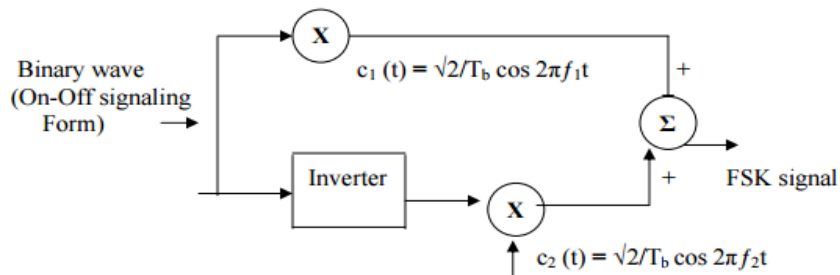
$f_i$  = Transmitted frequency =  $(nc + i)/T_b$

$n = \text{constant (integer)}$

$T_b = \text{bit interval}$

Symbol 1 is represented by  $S_1(t)$  and Symbol 0 is represented by  $S_0(t)$

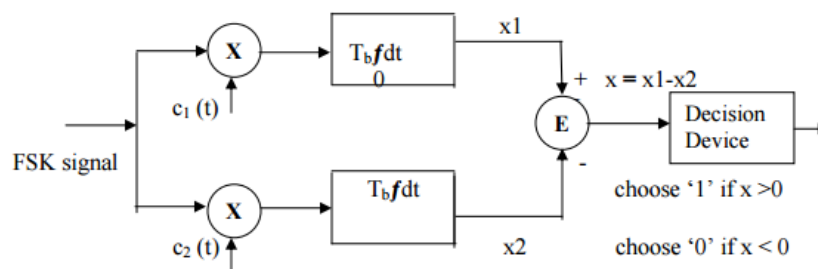
**BFSK Transmitter:** A BFSK transmitter is as shown in Figure 11.3. The input binary sequence is represented in its ON-OFF form, with symbol 1 represented by constant amplitude of  $\sqrt{E_b}$  volts & symbol 0 represented by zero volts. By using inverter in the lower channel, we in effect make sure that when symbol 1 is at the input, the two frequencies  $f_1$  &  $f_2$  are chosen to be equal integer multiples of the bit rate  $1/T_b$ . By summing the upper & lower channel outputs, we get BFSK signal.



**Figure 11.3: BFSK Transmitter**

**BFSK Receiver:** A BFSK receiver is as shown in Figure 11.4. The receiver consists of two correlators with common inputs which are supplied with locally generated coherent reference signals  $c_1(t)$  and  $c_2(t)$ .

The correlator outputs are then subtracted one from the other, and the resulting difference  $x$  is compared with a threshold of zero volts. If  $x > 0$ , the receiver decides in favour of symbol 1 and if  $x < 0$ , the receiver decides in favour of symbol 0.



**Figure 11.4: BFSK Receiver****PROCEDURE:**

FSK modulation is carried out follows:

- i. Generate two carriers signal.
- ii. Start FOR loop
- iii. Generate binary data, message signal and inverted message signal
- iv. Multiply carrier 1 with message signal and carrier 2 with inverted message signal
- v. Perform addition to get the FSK modulated signal
- vi. Plot message signal and FSK modulated signal.
- vii. End FOR loop.
- viii. Plot the binary data and carriers.

FSK demodulation is carried out follows:

- i. Start FOR loop
- ii. Perform correlation of FSK modulated signal with carrier 1 and carrier 2 to get two decision variables  $x_1$  and  $x_2$ .
- iii. Make decision on  $x = x_1 - x_2$  to get demodulated binary data. If  $x > 0$ , choose '1' else choose '0'.
- iv. Plot the demodulated binary data.

**MATLAB PROGRAM:**

% FSK Modulation

clc;

clearall;

closeall;

%generate carrier signal

Tb=1;

fc1=2;

fc2=5;

t=0:(Tb/100):Tb;



```

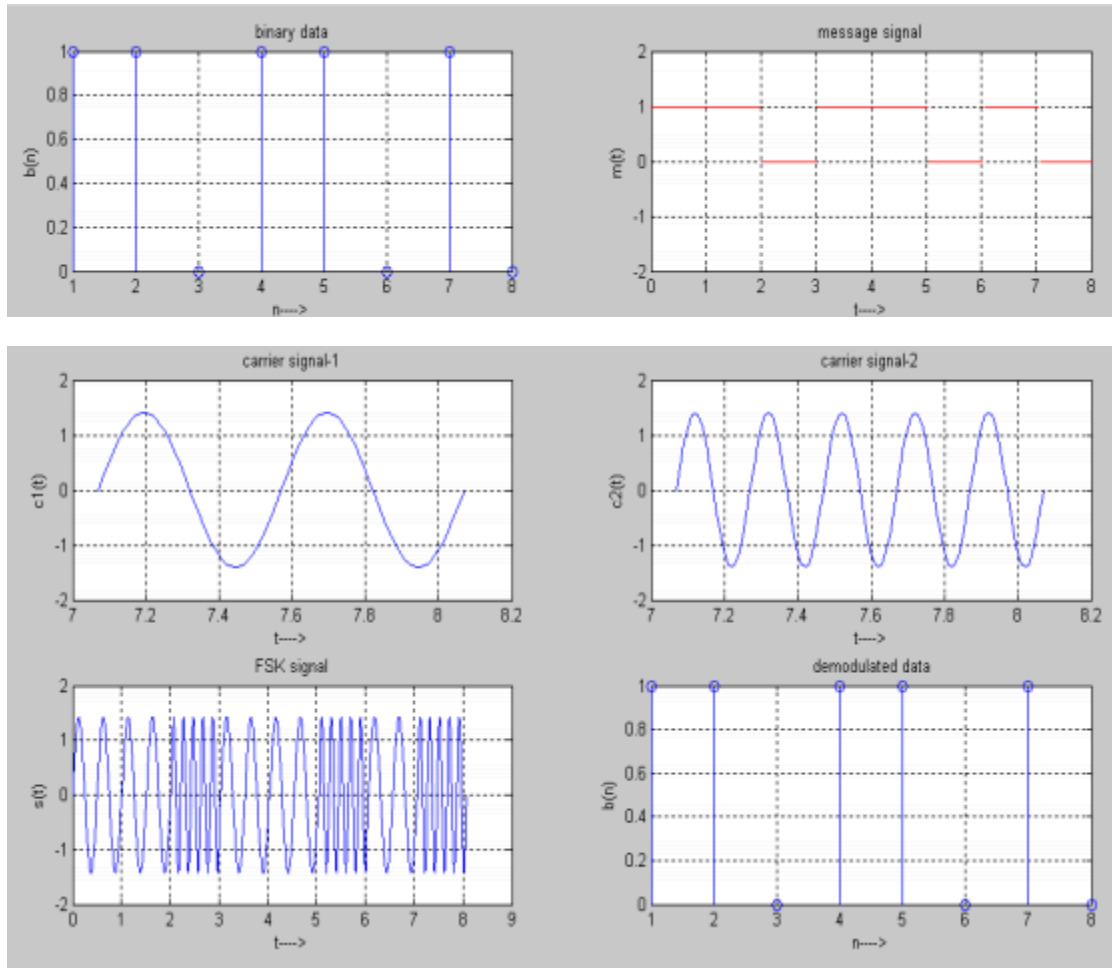
c1=sqrt(2/Tb)*sin(2*pi*fc1*t);
c2=sqrt(2/Tb)*sin(2*pi*fc2*t);
%generate message signal
N=8;
m=rand(1,N);
t1=0; t2=Tb
for i=1:N
t=[t1:(Tb/100):t2]
if m(i)>0.5
m(i)=1;
m_s=ones(1,length(t));
inv_m_s=zeros(1,length(t));
else
m(i)=0;
m_s=zeros(1,length(t));
inv_m_s=ones(1,length(t));
end
message(i,:)=m_s;
%Multiplier
fsk_sig1(i,:)=c1.*m_s;
fsk_sig2(i,:)=c2.*inv_m_s;
fsk=fsk_sig1+fsk_sig2;
%plotting the message signal and the modulated signal
subplot(3,2,2);
axis([0 N -2 2]);
plot(t,message(i,:), 'r');
title('message signal');
xlabel('t---->');
ylabel('m(t)');
grid on; hold on;
subplot(3,2,5); plot(t,fsk(i,:));
title('FSK signal'); xlabel('t---->'); ylabel('s(t)');

```

```
gridon; hold on;
t1=t1+(Tb+.01); t2=t2+(Tb+.01);
end
holdoff
%Plotting binary data bits and carrier signal
subplot(3,2,1); stem(m);
title('binary data'); xlabel('n---->'); ylabel('b(n)');
gridon;
subplot(3,2,3); plot(t,c1);
title('carrier signal-1'); xlabel('t---->'); ylabel('c1(t)');
gridon;
subplot(3,2,4); plot(t,c2);
title('carrier signal-2'); xlabel('t---->'); ylabel('c2(t)');
gridon;
% FSK Demodulation
t1=0; t2=Tb
for i=1:N
    t=[t1:(Tb/100):t2]
    % correlator
    x1=sum(c1.*fsk_sig1(i,:));
    x2=sum(c2.*fsk_sig2(i,:));
    x=x1-x2;
    %decision device
    if x>0
        demod(i)=1;
    else
        demod(i)=0;
    end
    t1=t1+(Tb+.01);
    t2=t2+(Tb+.01);
end
%Plotting the demodulated data bits
```

```
subplot(3,2,6);
stem(demod);
title(' demodulated data'); xlabel('n---->'); ylabel('b(n)');
grid on;
```

**OBSERVATIONS:** The following plots are obtained:



**RESULTS:** The programs for PSK and FSK modulation and demodulation have been simulated in MATLAB and necessary graphs are plotted.



**EXPERIMENT – 11**

**AIM – To write Matlab code for Binary to Gray and Gray to Binary Code Converter.**

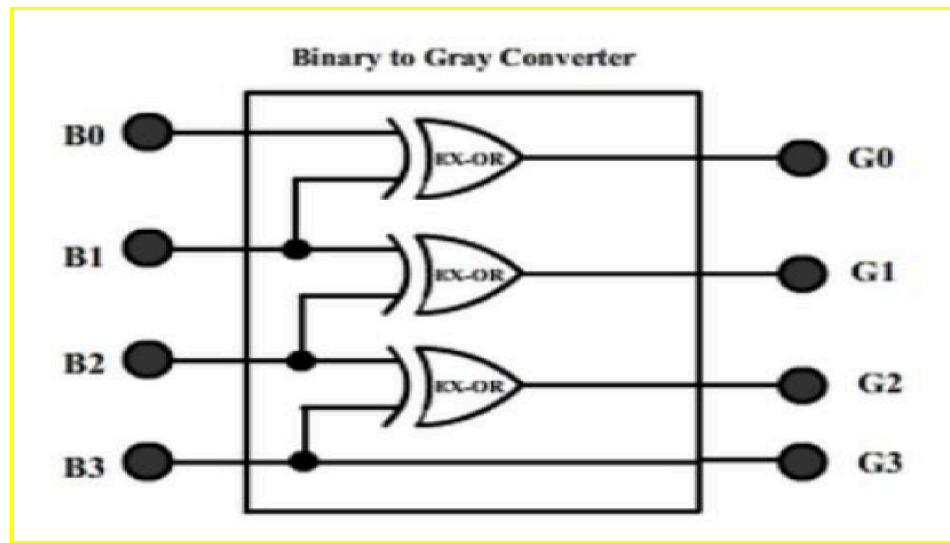
**THEORY- Gray Code** is one of the most important codes. It is a non-weighted code which belongs to a class of codes called minimum change codes. In these codes, while traversing from one step to another step only one bit in the code group changes. In case of **Gray Code** two adjacent code numbers differs from each other by only one bit. The idea of it can be cleared from the table given below. As this code it is not applicable in any types of arithmetical operations but it has some applications in analog to digital converters and in some input/output devices.

**BINARY TO GRAY CODE CONVERSION** is a very simple process. There are several steps to do this types of conversions. Steps given below elaborate on the idea on this type of conversion.

1. The M.S.B. of the gray code will be exactly equal to the first bit of the given binary number.
2. Now the second bit of the code will be exclusive-or of the first and second bit of the given binary number, i.e if both the bits are same the result will be 0 and if they are different the result will be 1.
3. The third bit of gray code will be equal to the exclusive-or of the second and third bit of the given binary number. Thus the **Binary to gray code conversion** goes on.

	<u>b[3:0]</u>				<u>g[3:0]</u>				
	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	
	1	0	1	0	1	1	1	1	
	1	0	1	1	1	1	1	0	
	1	1	0	0	1	0	1	0	
	1	1	0	1	1	0	1	1	
	1	1	1	0	1	0	0	1	
	1	1	1	1	1	0	0	0	

**Fig 14.1 Truth table of binary to gray converter**



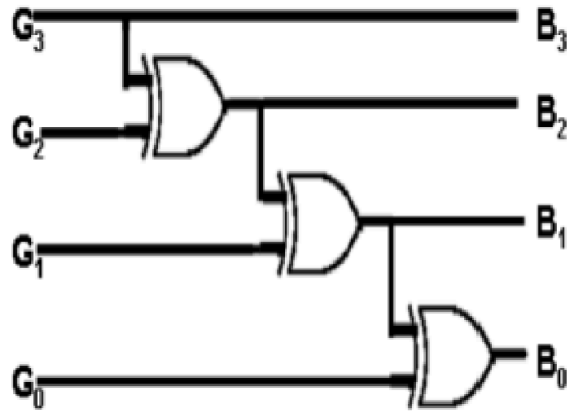
**Fig 14.2 Logic circuit of binary to gray converter**

**GRAY CODE TO BINARY CONVERSION** is again very simple and easy process. Following steps can make your idea clear on this type of conversions.

1. The M.S.B of the binary number will be equal to the M.S.B of the given gray code.
2. Now if the second gray bit is 0 the second binary bit will be same as the previous or the first bit. If the gray bit is 1 the second binary bit will alter. If it was 1 it will be 0 and if it was 0 it will be 1.
3. This step is continued for all the bits to do **Gray code to binary conversion**.

4 bit Gray Code	4 bit Binary Code
A B C D	B <sub>4</sub> B <sub>3</sub> B <sub>2</sub> B <sub>1</sub>
0 0 0 0	0 0 0 0
0 0 0 1	0 0 0 1
0 0 1 1	0 0 1 0
0 0 1 0	0 0 1 1
0 1 1 0	0 1 0 0
0 1 1 1	0 1 0 1
0 1 0 1	0 1 1 0
0 1 0 0	0 1 1 1
1 1 0 0	1 0 0 0
1 1 0 1	1 0 0 1
1 1 1 1	1 0 1 0
1 1 1 0	1 0 1 1
1 0 1 0	1 1 0 0
1 0 1 1	1 1 0 1
1 0 0 1	1 1 1 0
1 0 0 0	1 1 1 1

**Fig 14.3 Truth table of gray to binary converter**



**Fig 14.4 Logic circuit of gray to binary Converter**

**OBSERVATIONS:** The binary to gray converter and gray to binary converter output is obtained on command window for particular sequence of inputs.

**RESULTS:** The program for binary to gray converter and gray to binary converter has been simulated in MATLAB and truth tables have been verified.