Experiment 1

HDL CODE TO REALIZE ALL THE LOGIC GATES

Aim: To write VHDL code for all basic gates, simulate and verify functionality, synthesize .

Tools Required:

1. FPG Advantage
    i. Xilinx ISE 9.2

Theory :

AND:

The AND gate is an electronic circuit that gives a **high** output (1) only if **all** its inputs are high. A dot (.) is used to show the AND operation i.e. A.B.

OR:

The OR gate is an electronic circuit that gives a high output (1) if **one or more** of its inputs are high. A plus (+) is used to show the OR operation.

NOT:

The NOT gate is an electronic circuit that produces an inverted version of the input at its output. It is also known as an *inverter*. If the input variable is A, the inverted output is known as NOT A. This is also shown as A', or A with a bar over the top.

NAND:

This is a NOT-AND gate which is equal to an AND gate followed by a NOT gate. The outputs of all NAND gates are high if **any** of the inputs are low. The symbol is an AND gate with a small circle on the output. The small circle represents inversion.

NOR:

This is a NOT-OR gate which is equal to an OR gate followed by a NOT gate. The outputs of all NOR gates are low if **any** of the inputs are high. The symbol is an OR gate with a small circle on the output. The small circle represents inversion.

EX-OR:

The '**Exclusive-OR**' gate is a circuit which will give a high output if **either, but not both**, of its two inputs are high. An encircled plus sign ( ) is used to show the EXOR operation.

Logic gate Truth Tables:

2 Input AND gate		
A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

2 Input NAND gate		
A	B	$\overline{A.B}$
0	0	1
0	1	1
1	0	1
1	1	0

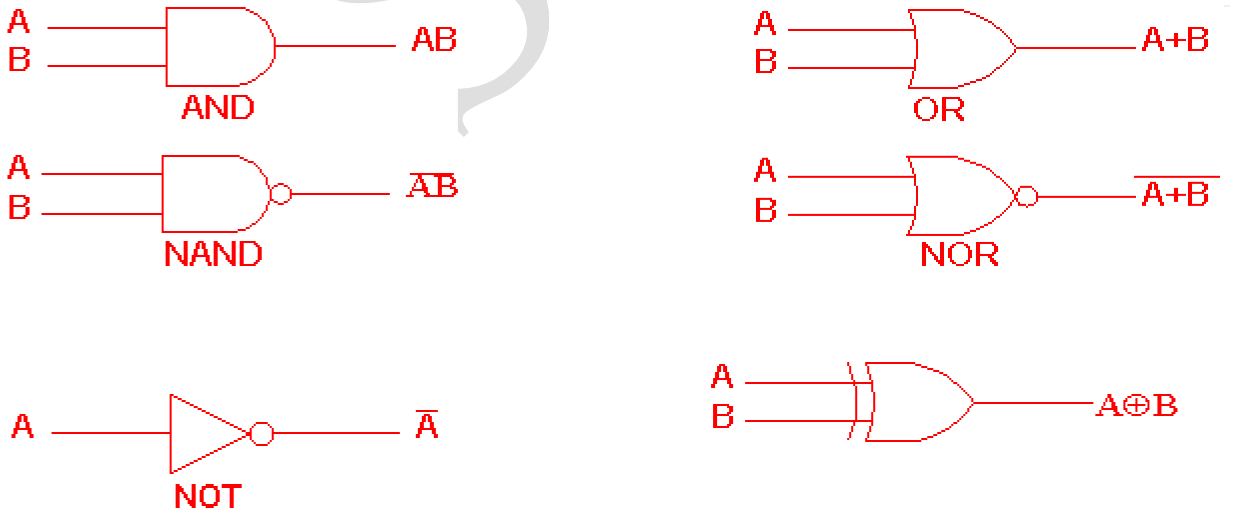
2 Input OR gate		
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

2 Input NOR gate		
A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

2 Input EXOR gate		
A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

NOT gate	
A	$\overline{A}$
0	1
1	0

Logic gate symbols:



Procedure:

1. Click on FPGA advantage icon on the desktop.
2. Click on file menu→new→project.
3. Create a new path for the project workspace.
4. Then, go to File→new→design content→VHDL file→entity.
5. Now, give the name of the entity & click next, then an editor window opens,
6. Declare the input, output ports in the entity and save it.
7. File→new→design content→VHDL file→architecture.
8. Now, give the name of the entity you gave before and a architecture name and click next, then a editor window opens, write the required style of code and save it.
9. Click the project file and verify the errors by CHECK button.
10. If no errors, click on simulate button, then modelsim gets started, select the ports and give them to "select to wave" option and type the force commands and run command ,then the graph is displayed.
11. After that, move to design manager window, select the project file and click on synthesize button, then Leonardo Spectrum windows gets opened, in that, click on view RTL schematic button, the required logic diagram is displayed.

VHDL code:

AND gate:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY and1 IS
  port(a,b:in std_logic;
       c:out std_logic);
END ENTITY and1;
ARCHITECTURE dataflow OF and1 IS
BEGIN
```

```
  c<=a and b;
END ARCHITECTURE dataflow;
```

OR gate:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY or1 IS
  port(a,b:in std_logic;
        c:out std_logic);
END ENTITY or1;
ARCHITECTURE dataflow OF or1 IS
BEGIN
  c<=a or b;
END ARCHITECTURE dataflow;
```

NOT gate:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY not1 IS
  port(a:in std_logic;
       o : out  std_logic);
END ENTITY not1;
ARCHITECTURE dataflow OF not1 IS
BEGIN
  o<=not a;
END ARCHITECTURE dataflow;
```

NAND gate:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY nand1 IS
port(a,b:in std_logic;
        c:out std_logic);

END ENTITY nand1;
ARCHITECTURE dataflow OF nand1 IS
BEGIN
  c<=a nand b;
```

```
END ARCHITECTURE dataflow;
```

NOR gate:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY nor1 IS
  port(a,b:in std_logic;
        c:out std_logic);
END ENTITY nor1;
ARCHITECTURE dataflow OF nor1 IS
BEGIN
  c<=a nor b;
END ARCHITECTURE dataflow;
```

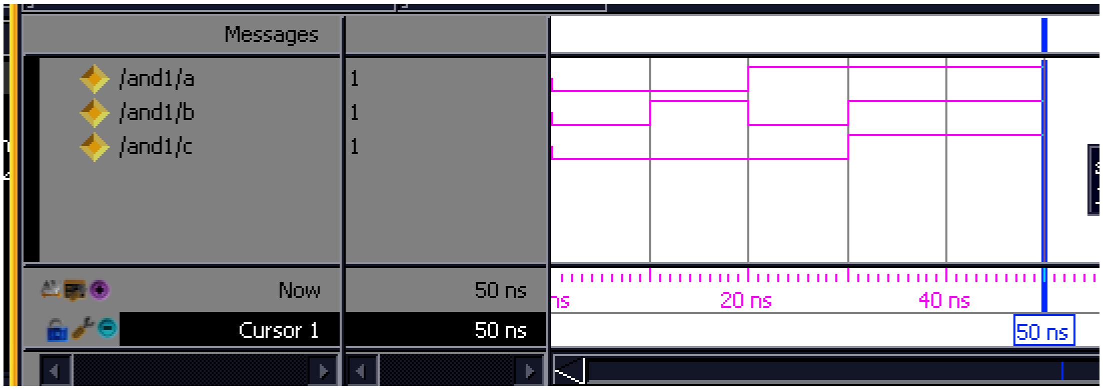
XOR gate:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY xor1 IS
  port(a,b:in std_logic;
        c:out std_logic);
END ENTITY xor1;
ARCHITECTURE dataflow OF xor1 IS
BEGIN
  c<=a xor b;
END ARCHITECTURE dataflow;
```

Simulations:

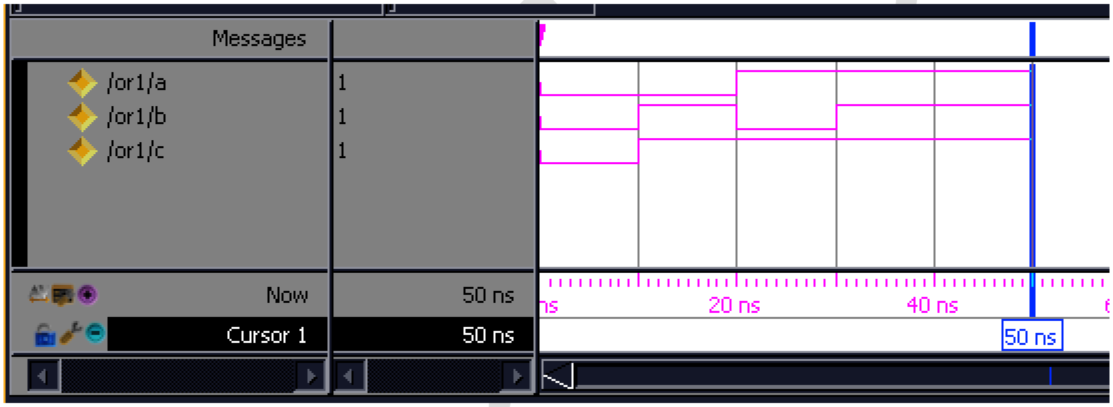
AND gate:

```
force a 0 0ns,0 10ns,1 20ns,1 30ns
force b 0 0ns,1 10ns,0 20ns,1 30ns
run 50ns
```



Messages	
/and1/a	1
/and1/b	1
/and1/c	1
Now	50 ns
Cursor 1	50 ns

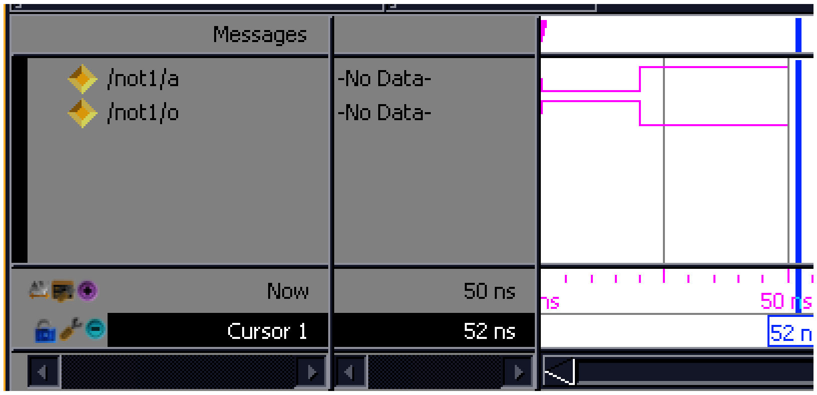
OR gate: force a 0 0ns,0 10ns,1 20ns,1 30ns
force b 0 0ns,1 10ns,0 20ns,1 30ns
run 50ns



Messages	
/or1/a	1
/or1/b	1
/or1/c	1
Now	50 ns
Cursor 1	50 ns

NOT gate:

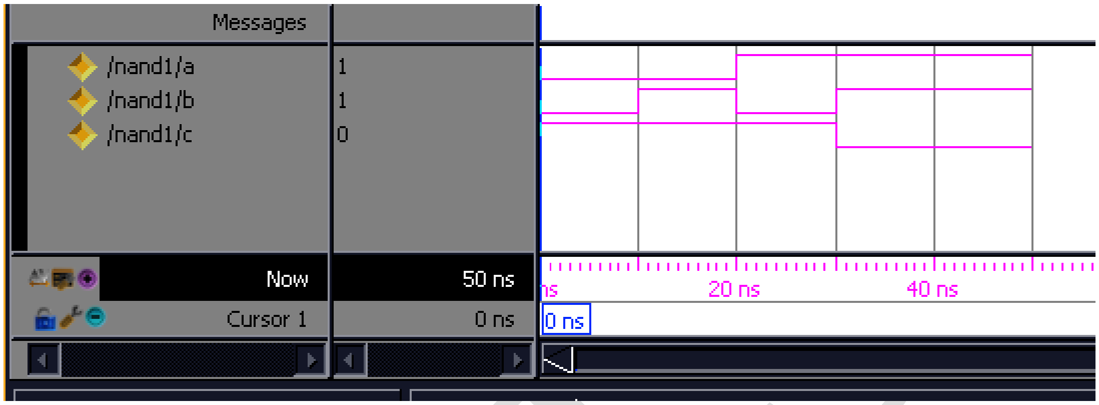
force a 0 0ns,0 10ns,1 20ns,1 30ns
force b 0 0ns,1 10ns,0 20ns,1 30ns
run 50ns



Messages	
/not1/a	-No Data-
/not1/b	-No Data-
Now	50 ns
Cursor 1	52 ns

NAND gate:

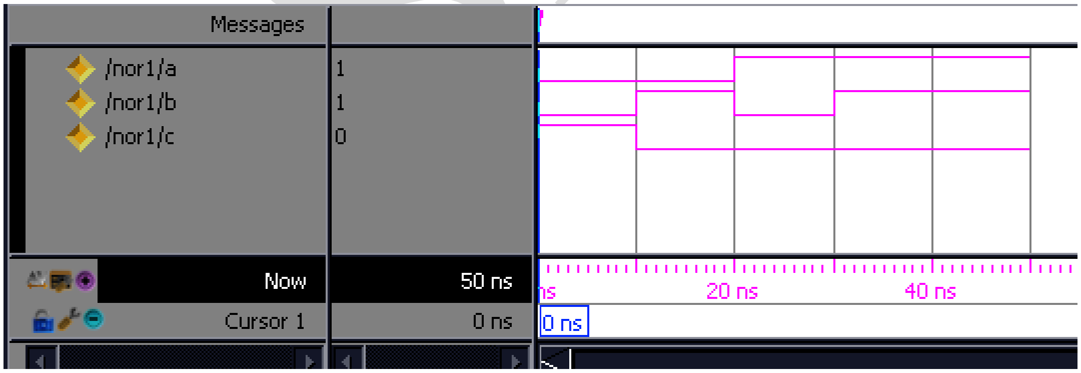
```
force a 0 0ns,0 10ns,1 20ns,1 30ns
force b 0 0ns,1 10ns,0 20ns,1 30ns
run 50ns
```



Messages	
/nand1/a	1
/nand1/b	1
/nand1/c	0
Now	50 ns
Cursor 1	0 ns

NOR gate:

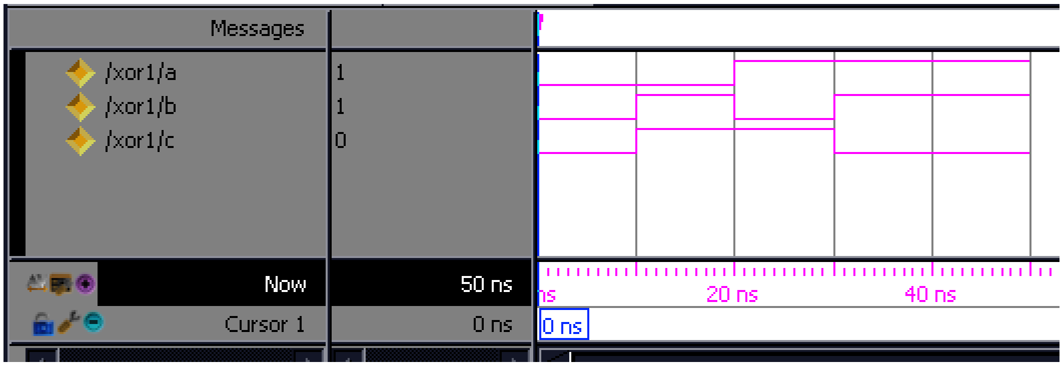
```
force a 0 0ns,0 10ns,1 20ns,1 30ns
force b 0 0ns,1 10ns,0 20ns,1 30ns
run 50ns
```



Messages	
/nor1/a	1
/nor1/b	1
/nor1/c	0
Now	50 ns
Cursor 1	0 ns

XOR gate:

```
force a 0 0ns,0 10ns,1 20ns,1 30ns
force b 0 0ns,1 10ns,0 20ns,1 30ns
run 50ns
```



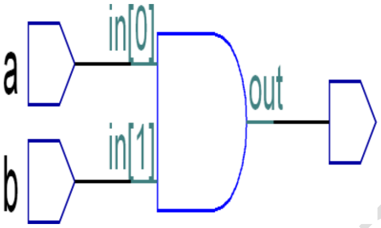
Messages

/xor1/a 1
/xor1/b 1
/xor1/c 0

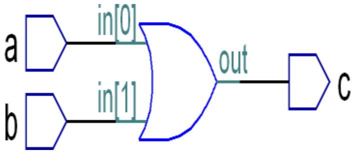
Now 50 ns
Cursor 1 0 ns

Synthesis Diagrams:

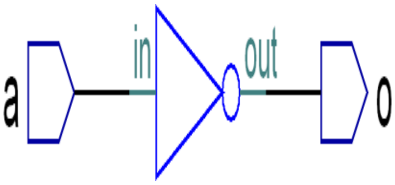
AND gate:



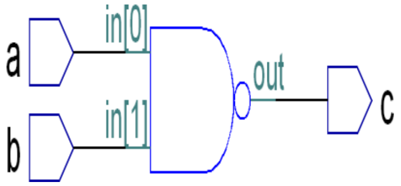
OR gate:



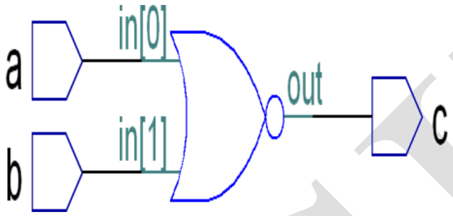
NOT gate:



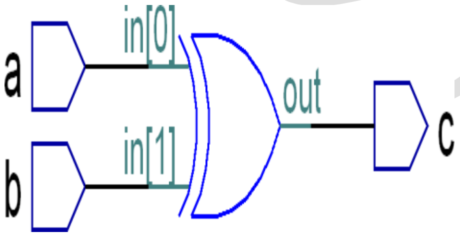
NAND gate:



NOR gate:



XOR gate:



Conclusion:

The VHDL code for all basic gates is written, simulated and synthesized.

Experiment 2

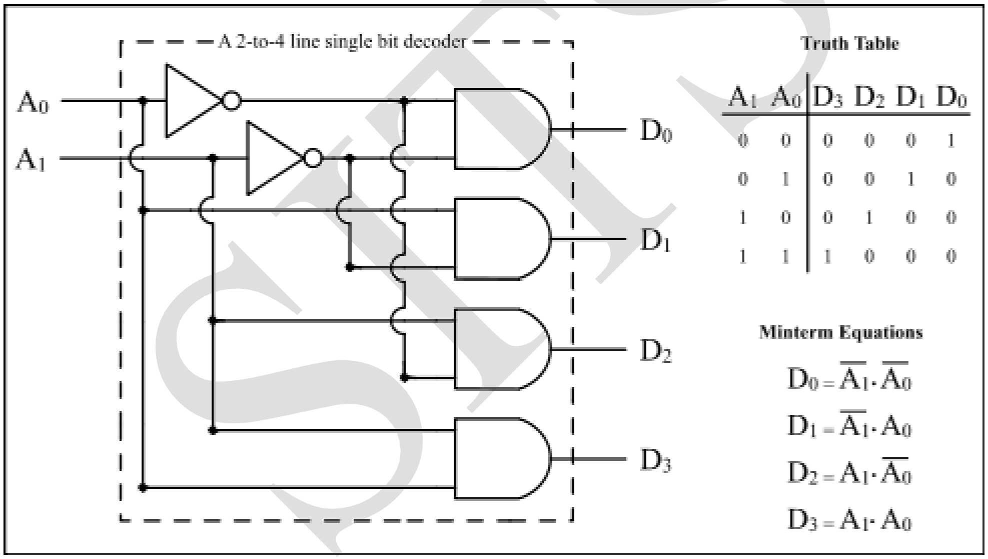
DESIGN OF 2-to-4 DECODER

Aim: To write VHDL code for 2-to-4 decoder in Behavioral modeling, Structural Modeling, simulate and synthesize

Tools Required:

1. FPG Advantage
   i. Xilinx ISE 9.2

Theory : A decoder can take the form of a multiple-input, multiple-output logic circuit that converts coded inputs into coded outputs, where the input and output codes are different e.g. n-to-2n , binary-coded decimal decoders. Decoding is necessary in applications such as data multiplexing, 7 segment display and memory address decoding.



Truth Table

A_1	A_0	D_3	D_2	D_1	D_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Minterm Equations

$D_0 = \overline{A_1} \cdot \overline{A_0}$

$D_1 = \overline{A_1} \cdot A_0$

$D_2 = A_1 \cdot \overline{A_0}$

$D_3 = A_1 \cdot A_0$

Procedure: Refer to page 3

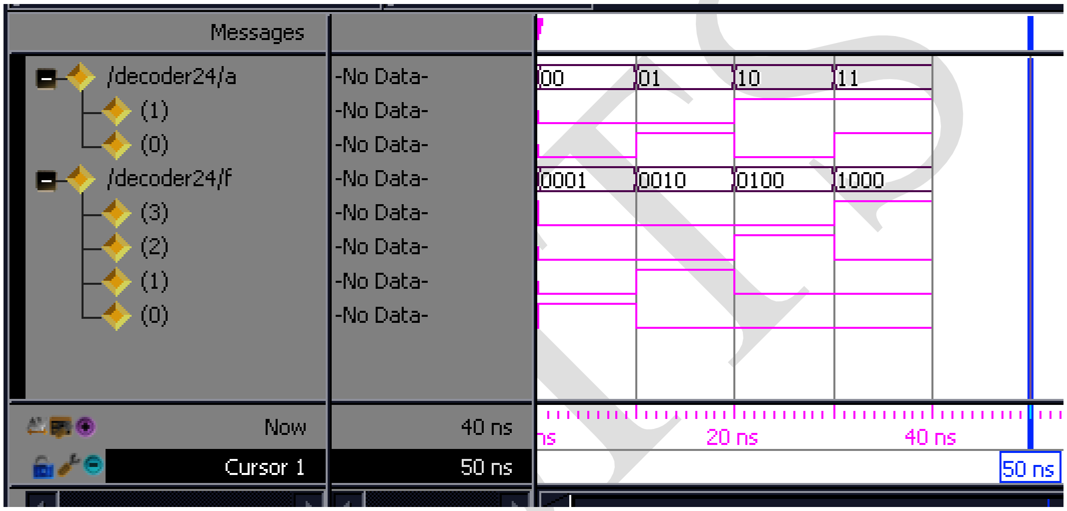
VHDL code (Behavioural Modelling using with-select):

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY decoder24 IS
  port(a:in std_logic_vector(1 downto 0);
       f:out std_logic_vector(3 downto 0));
END ENTITY decoder24;
```

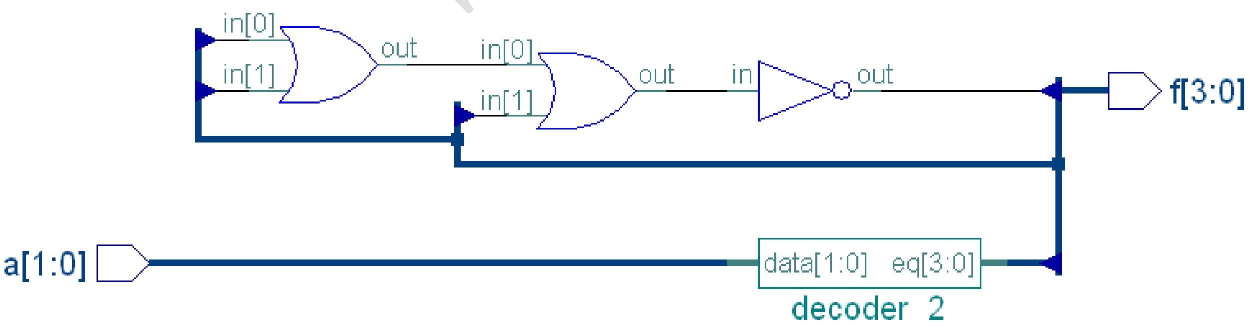
```
ARCHITECTURE behav_with_select OF decoder24 IS
BEGIN
  with a select
   f <="0001" when "00",
       "0010" when "01",
       "0100" when "10",
       "1000" when others;
END ARCHITECTURE behav_with_select;
```

Simulations: Force a 00 0ns,01 10ns, 10 20ns,11 30ns

Run 40ns



Synthesis Diagrams:



VHDL code (Behavioural using case):

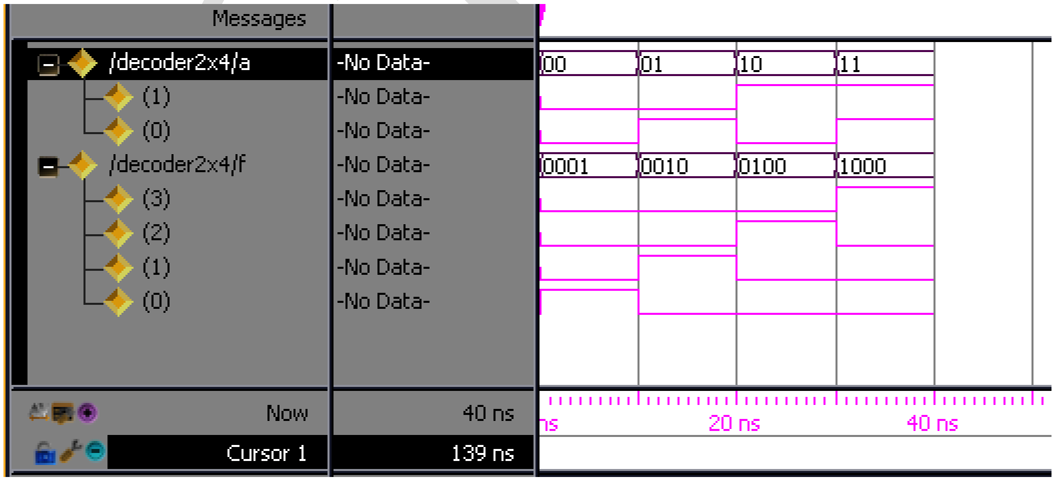
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
```

```
ENTITY decoder2x4 IS
  port(a:in std_logic_vector(1 downto 0);
     f:out std_logic_vector(3 downto 0));
END ENTITY decoder2x4;
ARCHITECTURE behav_when_case OF decoder2x4 IS
BEGIN
  process(a)
    begin
    case(a) is
       when "00" => f <= "0001";
       when "01" => f <= "0010";
       when "10" => f <= "0100";
      when "11" => f <= "1000";
       when others => f <= "0000";
       end case;
     end process;
END ARCHITECTURE behav_when_case;
```

Simulations:

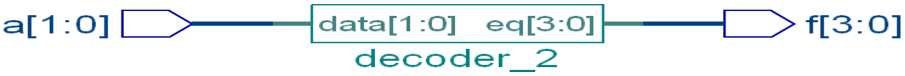
Force a 00 0ns,01 10ns, 10 20ns,11 30ns

Run 40ns



```
Messages
/decoder2x4/a    -No Data-   00    01    10    11
  (1)            -No Data-
  (0)            -No Data-
/decoder2x4/f    -No Data-   0001  0010  0100  1000
  (3)            -No Data-
  (2)            -No Data-
  (1)            -No Data-
  (0)            -No Data-
Now       40 ns
Cursor 1  139 ns
```

Synthesis Diagrams:



VHDL code(Structural modelling):

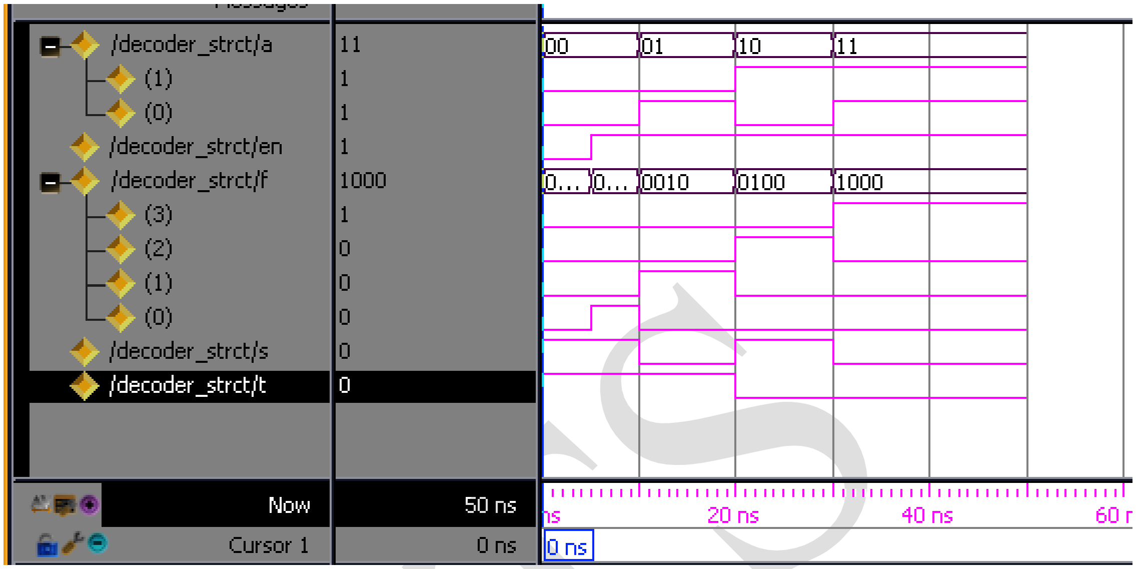
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY decoder_strct IS
  port(a:in std_logic_vector(1 downto 0);
       En:in std_logic;
     f:out std_logic_vector(3 downto 0));
END ENTITY decoder_strct;
ARCHITECTURE stuctural OF decoder_strct IS
  signal s,t: std_logic;
  component inv1 port(I :in std_logic;o:out std_logic);
  end component;
   component and3 port(I0,I1,I3: in std_logic;o:out std_logic);
   end component;
      begin
        u1:inv1 port map(a(0),s);
        u2:inv1 port map(a(1),t);
        u3:and3 port map(s,t,En,f(0));
        u4:and3 port map(a(0),t,En,f(1));
        u5:and3 port map(s,a(1),En,f(2));
        u6:and3 port map(a(0),a(1),En,f(3));
 END ARCHITECTURE stuctural;
```

Internal program and3 :

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE  ieee.std_logic_arith.all;
ENTITY and3 IS
  port(I0,I1,I3:in std_logic;
      o: out std_logic);
END ENTITY and3;
ARCHITECTURE dataflow OF and3 IS
BEGIN
  o <=I0 and I1 and I3;
END ARCHITECTURE dataflow;
```

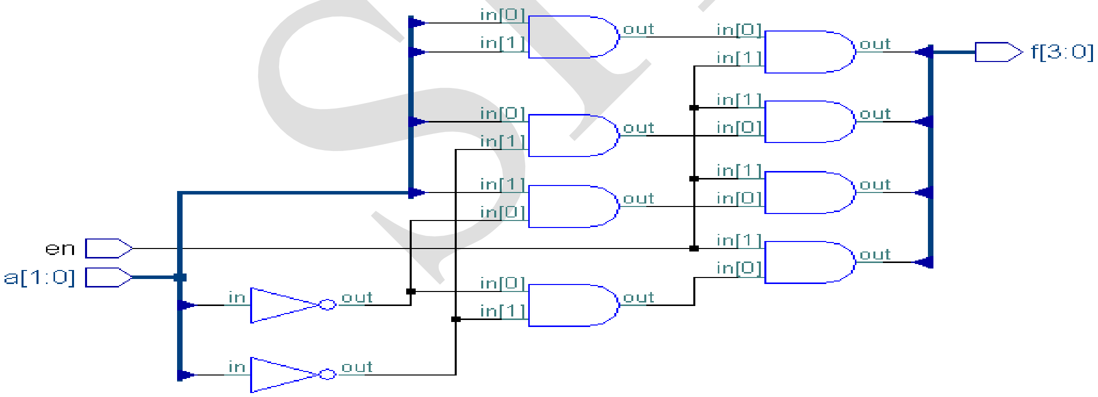
Simulations:

```
force a 00 0ns,01 10ns,10 20ns,11 30ns
force en 0 0ns,1 5ns
run 50ns
```



```
/decoder_strct/a     11
  (1)                1
  (0)                1
/decoder_strct/en    1
/decoder_strct/f     1000
  (3)                1
  (2)                0
  (1)                0
  (0)                0
/decoder_strct/s     0
/decoder_strct/t     0
Now                  50 ns
Cursor 1             0 ns
```

Synthesis Diagrams:



Conclusion:

The VHDL code for 2-to-4 decoder using behavioral (using with-select, when-else), Structural model using is written, simulated and synthesized.

Experiment 3

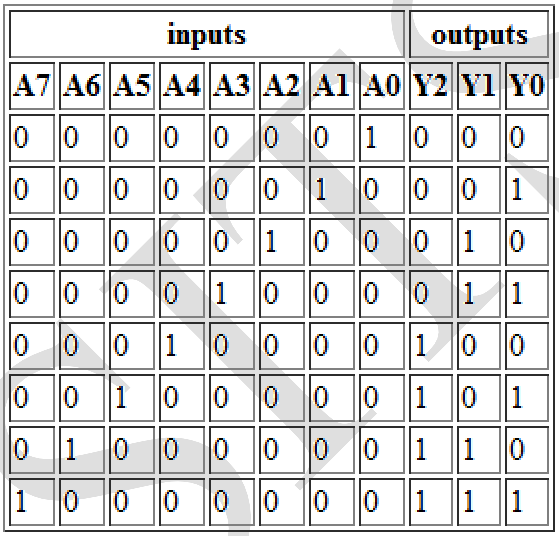
DESIGN OF 8-to-3 ENCODER

Aim: To write the VHDL code for 8-to-3 Encoder in Dataflow, Behavioral, Structural modeling simulate and synthesize.

Tools Required:

1. FPG Advantage
   i. Xilinx ISE 9.2

Theory : The truth table for an 8-3 binary encoder (8 inputs and 3 outputs) is shown in the following table. It is assumed that only one input has a value of 1 at any given time.



inputs								outputs		
A7	A6	A5	A4	A3	A2	A1	A0	Y2	Y1	Y0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

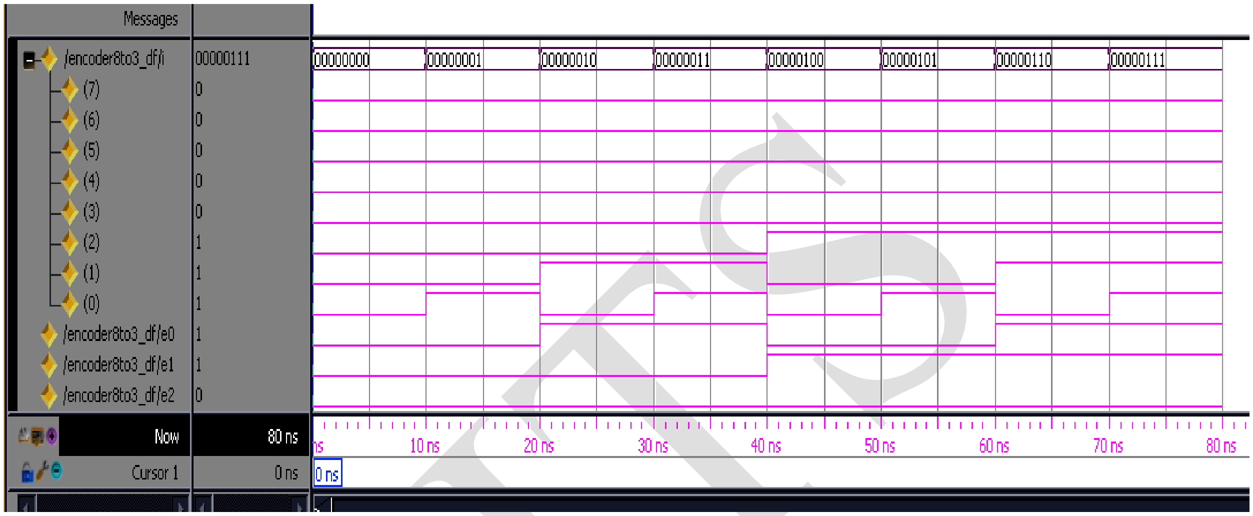
Procedure: Refer to page 3

VHDL Code (using Dataflow Modelling):

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY encoder8to3_df IS
  port(I:in std_logic_vector(7 downto 0);
          E0,E1,E2:out std_logic);
END ENTITY encoder8to3_df;
```

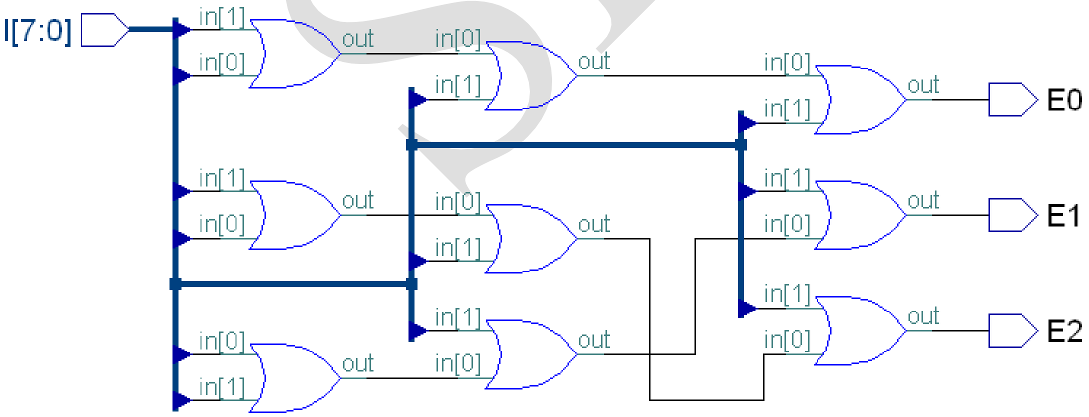
```
ARCHITECTURE dataflow OF encoder8to3_df IS
BEGIN
        E0 <=I(1) or I(3)or I(5) or I(7);
        E1 <=I(2) or I(3)or I(6) or I(7);
        E2 <=I(4) or I(5)or I(6) or I(7);
END ARCHITECTURE dataflow;
```

Simulation :



```
Messages
/encoder8to3_df/i    00000111
  (7)                0
  (6)                0
  (5)                0
  (4)                0
  (3)                0
  (2)                1
  (1)                1
  (0)                1
/encoder8to3_df/e0   1
/encoder8to3_df/e1   1
/encoder8to3_df/e2   0
Now                  80 ns
Cursor 1             0 ns
```

Synthesis diagram:



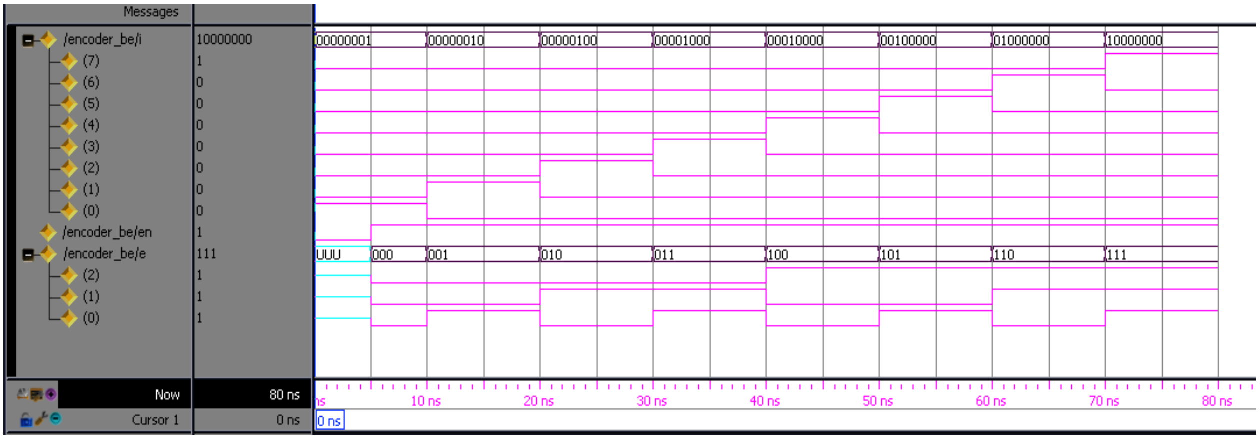
VHDL Code (in Behavioural Modelling using when-else Statement):

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
```

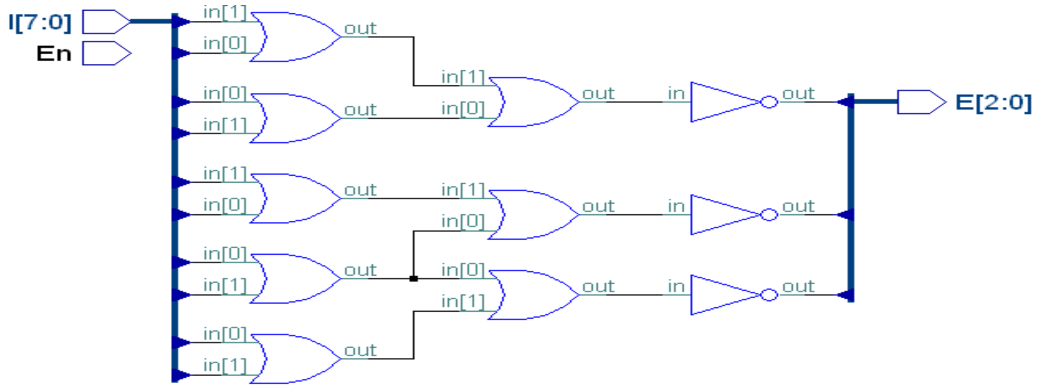
```
ENTITY encoder_be IS
  port(I : in std_logic_vector( 7 downto 0);
       En : in std_logic;
       E : out std_logic_vector(2 downto 0));
END ENTITY encoder_be;
ARCHITECTURE behav OF encoder_be IS
BEGIN
  process(I,En)
          begin
          if En ='1' then
          case I is
                     when "00000001"=> E <= "000";
                     when "00000010"=> E <= "001";
                     when "00000100"=> E <= "010";
                     when "00001000"=> E <= "011";
                     when "00010000"=> E <= "100";
                     when "00100000"=> E <= "101";
                     when "01000000"=> E <= "110";
                     when "10000000"=> E <= "111";
                     when others => E <= "UUU";
         end case;
                     else E <= "UUU";
      end if;
      end process;
END ARCHITECTURE behav;
```

Simulation **:**force I 00000001 0ns,00000010 10ns,00000100 20ns,00001000 30ns,00010000 40ns,00100000 50ns,01000000 60ns,10000000 70ns

force en 0 0ns,1 5ns run 80ns



Synthesis diagram:



VHDL Code(in Structural Modelling):

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY encoder_st IS
  port(I : in std_logic_vector(7 downto 0);
       E : out std_logic_vector(2 downto 0));
END ENTITY encoder_st;
ARCHITECTURE struct OF encoder_st IS
component or2 port(A,B,C,D : in std_logic;
                             M: out std_logic);
end component;
BEGIN
            u1 :or2 port map(I(1),I(3),I(5),I(7),E(0));
            u2 :or2 port map(I(2),I(3),I(6),I(7),E(1));
            u3: or2 port map(I(4),I(5),I(6),I(7),E(2)) ;
END ARCHITECTURE struct;
```

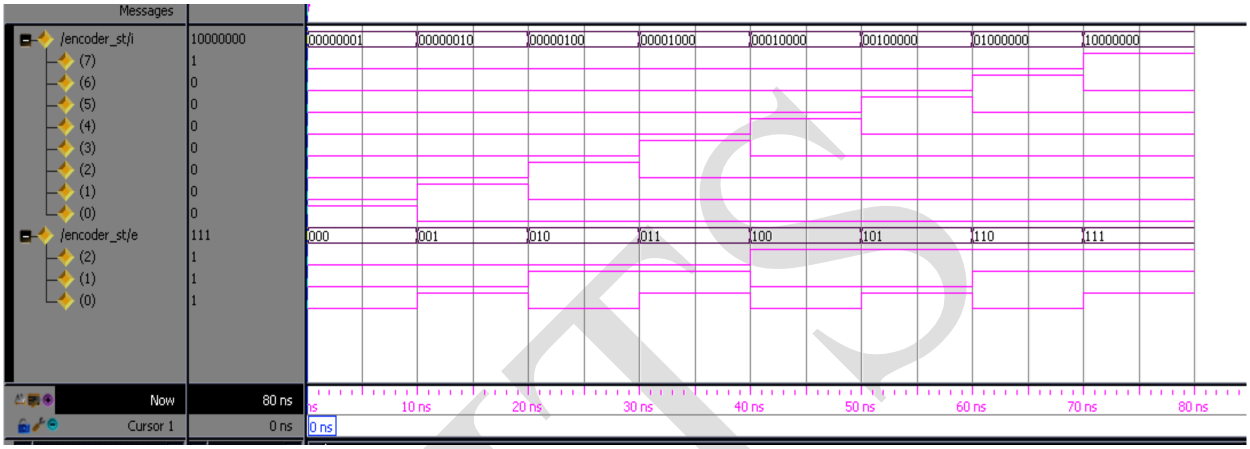
Internal program or2:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE  ieee.std_logic_arith.all;
ENTITY or2 IS
  port(A,B,C,D:in std_logic;
                 M:out std_logic);
END ENTITY or2;
ARCHITECTURE dataflow OF or2 IS
```

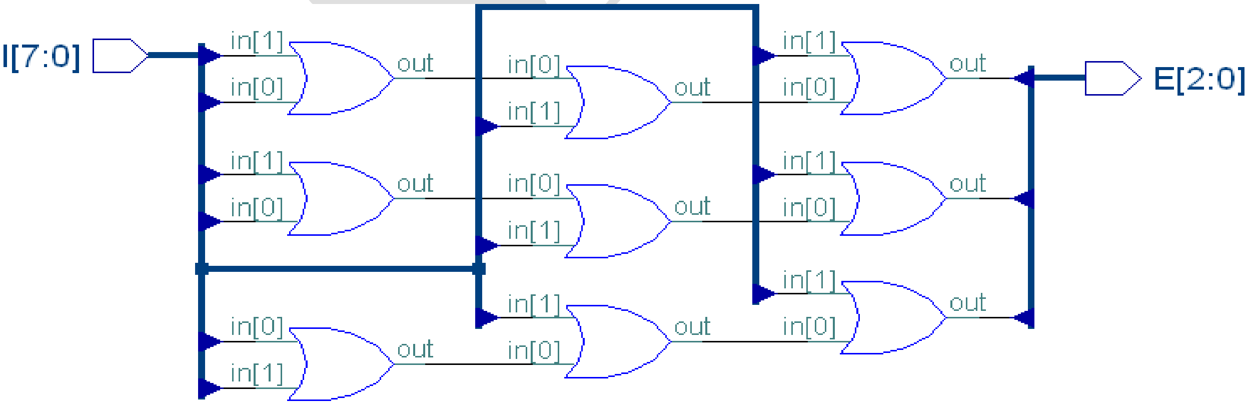
```
BEGIN
        M <= A or B or C or D;
END ARCHITECTURE dataflow;
```

Simulation :

```
force I 00000001 0ns,00000010 10ns,00000100 20ns,00001000 30ns,00010000
40ns,00100000 50ns,01000000 60ns,10000000 70ns
run 80ns
```



Synthesis diagram:



Conclusion:

The VHDL code for 8-to-3 encoder using Dataflow, Behavioural (using when-else Statement), Structural modelling is written, simulated and synthesized.

Experiment 4

DESIGN OF 8-To-1 MULTIPLEXER

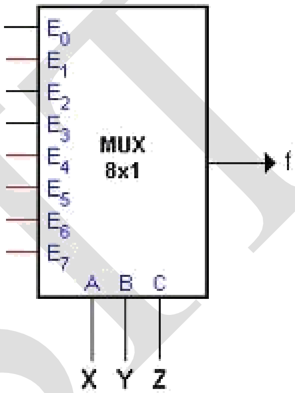
Aim: To write the VHDL code for 8-to-1 multiplexer, simulate and synthesize.

Tools Required:

1. FPG Advantage
   ii. Xilinx ISE 9.2

Theory:

A **multiplexer** (or **MUX**) is a device that selects one of several analog or digital 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.



Procedure: Refer to page 3

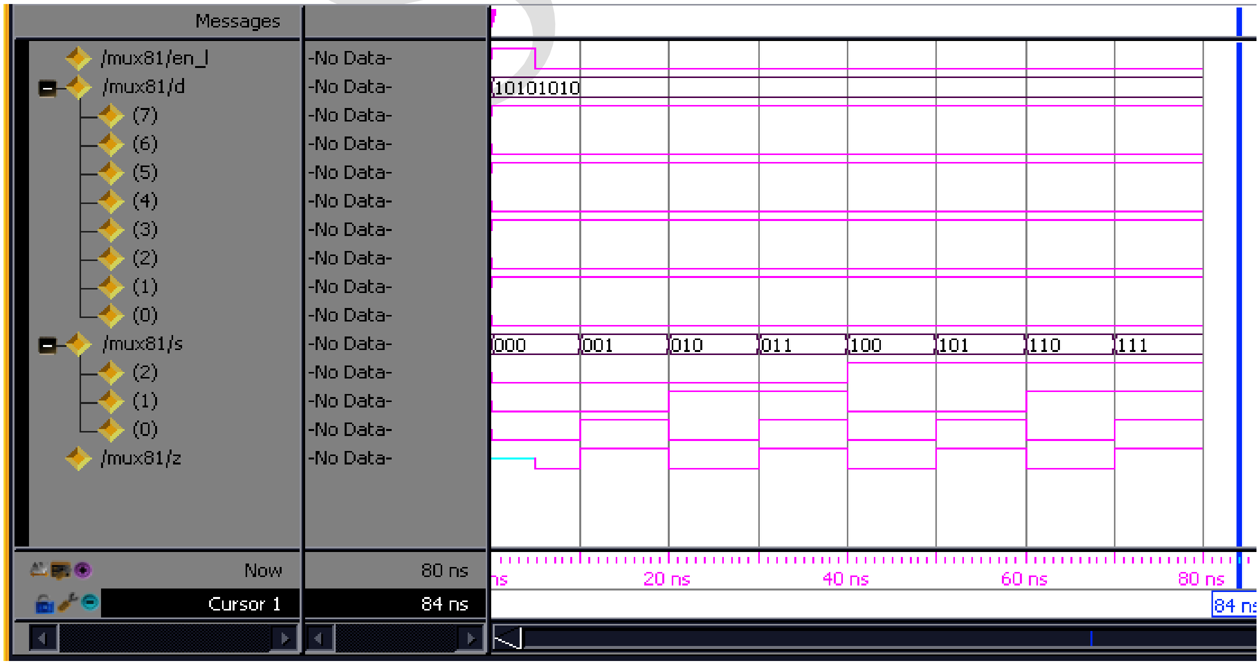
VHDL code:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE  ieee.std_logic_arith.all;
ENTITY mux81 IS
  port(EN_L: in std_logic;
       D: in std_logic_vector(7 downto 0);
       S: in std_logic_vector(2 downto 0);
       Z: out std_logic);
END ENTITY mux81;
ARCHITECTURE behav OF mux81 IS
```

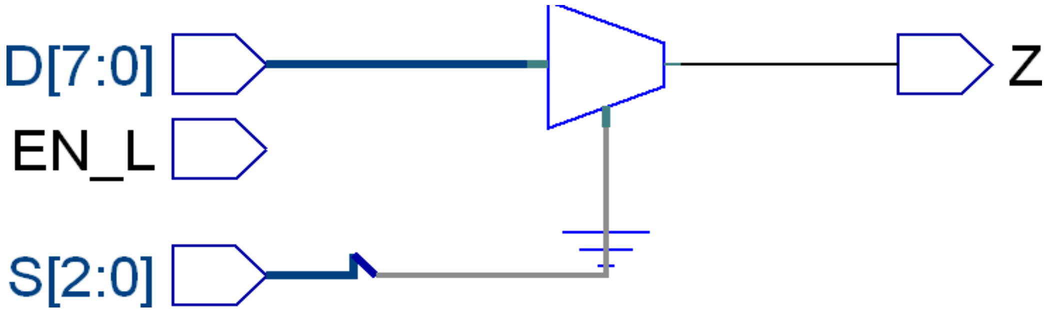
```
BEGIN
 process(S,D,EN_L)
  begin
   if EN_L ='0' then
    case(S) is
    when "000" => Z <= D(0);
   when "001" => Z <= D(1);
   when "010" => Z <= D(2);
   when "011" => Z <= D(3);
   when "100" => Z <= D(4);
   when "101" => Z <= D(5);
   when "110" => Z <= D(6);
   when "111" => Z<= D(7);
    when others => Z <= 'U';
      end case;
          else Z <='U';
      end if;
   end process;
END ARCHITECTURE behav;
```

Simulation :

```
force EN_L 1 0ns,0 5ns
force D 10101010 0ns
force S 000 0ns,001 10ns,010 20ns,011 30ns,100 40ns,101 50ns,110 60ns,111 70ns
run 80ns
```



Synthesis diagram:



Conclusion:

The VHDL code for 8-to-1 multiplexer is written, simulated and synthesized.

Experiment 5

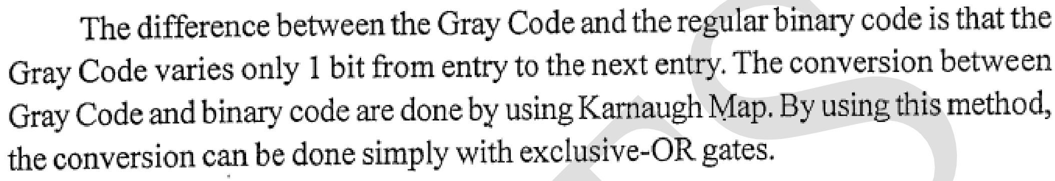
DESIGN OF 4 BIT BINARY TO GRAY CODE CONVERSION

Aim: To write the VHDL code for 4 Bit Binary to Gray code conversion, simulate and synthesize.

Tools Required:

1. FPG Advantage
   i. Xilinx ISE 9.2

Theory :



The difference between the Gray Code and the regular binary code is that the Gray Code varies only 1 bit from entry to the next entry. The conversion between Gray Code and binary code are done by using Karnaugh Map. By using this method, the conversion can be done simply with exclusive-OR gates.

Procedure: Refer to page 3

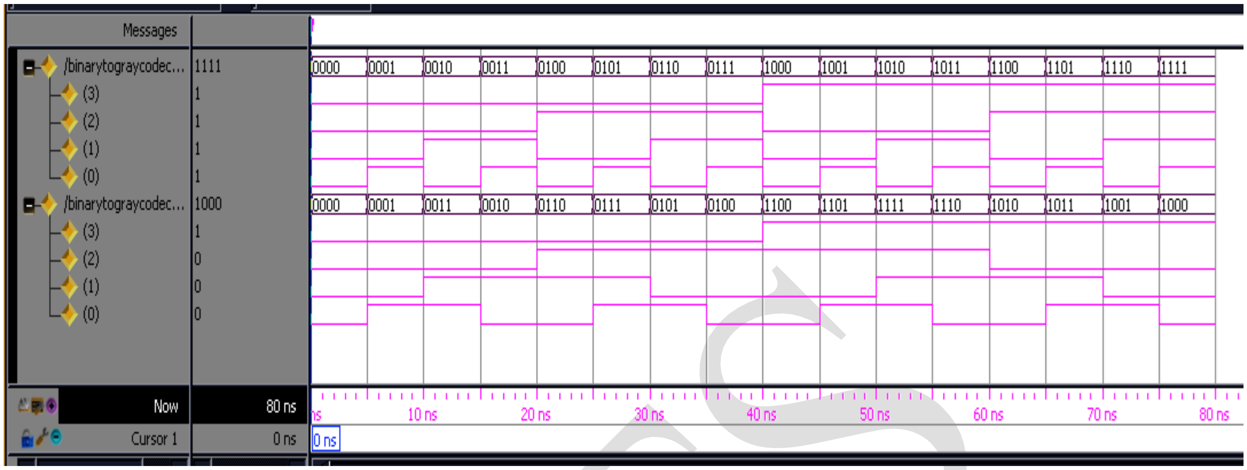
VHDL code :

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY binarytograycodeconverter IS
  port(b:in std_logic_vector(3 downto 0);
       g: out std_logic_vector(3 downto 0));
END ENTITY binarytograycodeconverter;
ARCHITECTURE dataflow OF binarytograycodeconverter IS
BEGIN
  g(3)<=b(3);
  g(2)<=b(3)xor b(2);
  g(1)<=b(2)xor b(1);
  g(0)<=b(1) xor b(0);
END ARCHITECTURE dataflow;
```

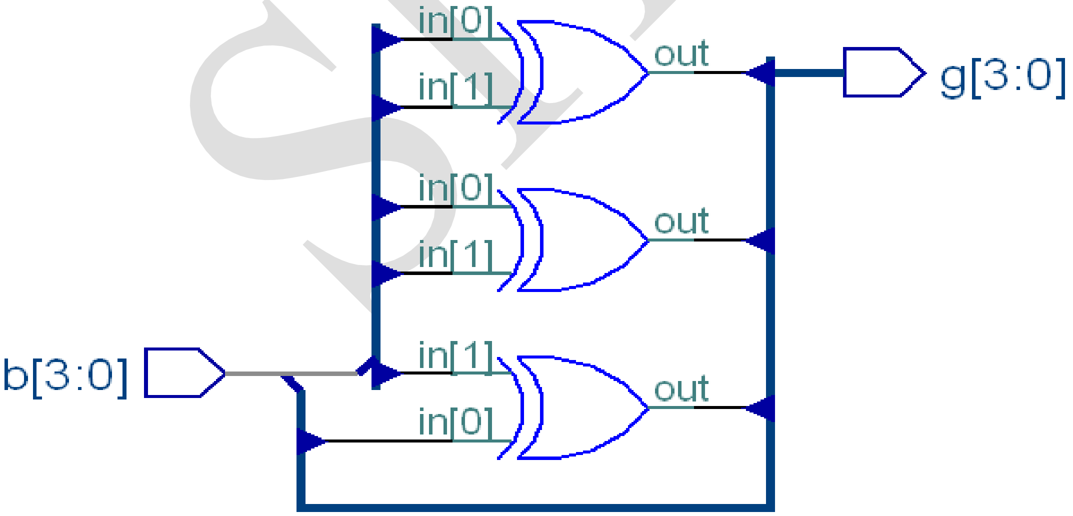
Simulations:

Force 0000 0ns,0001 5ns,0010 10ns,0011 15ns,0100 20ns,0101 25ns,0110 30ns,0111 35ns,1000 40ns,1001 45ns,1010 50ns,1011 55ns,1100 60ns,1101 65ns,1110 70ns,1111 75ns
Run 80ns



```
Messages
/binarytograycodec...  1111   0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
   (3)                 1
   (2)                 1
   (1)                 1
   (0)                 1
/binarytograycodec...  1000   0000 0001 0011 0010 0110 0111 0101 0100 1100 1101 1111 1110 1010 1011 1001 1000
   (3)                 1
   (2)                 0
   (1)                 0
   (0)                 0
Now                    80 ns   ns   10 ns   20 ns   30 ns   40 ns   50 ns   60 ns   70 ns   80 ns
Cursor 1               0 ns    0 ns
```

Synthesis Diagrams:



Conclusion:

The VHDL code for binary to gray code conversion is written , simulated and synthesized.

Experiment 6(a)

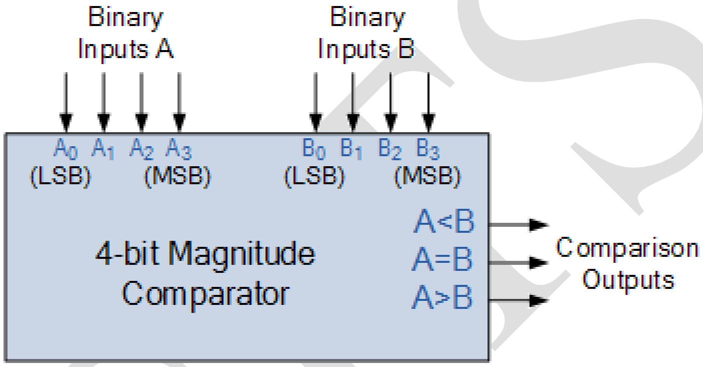
DESIGN OF 4-BIT COMPARATOR

Aim: To write the VHDL code for 4-BIT COMPARATOR, simulate and synthesize using dataflow model.

Tools Required:

1. FPG Advantage
   - i. Xilinx ISE 9.2

Theory :



The purpose of a **Digital Comparator** is to compare a set of variables or unknown numbers, for exampleA (A1, A2, A3, An, etc) against that of a constant or unknown value such as B (B1, B2, B3, ... Bn, etc) and produce an output condition or flag depending upon the result of the comparison. For example, a magnitude comparator of two 1-bits, (A and B) inputs would produce the following three output conditions when compared to each other.

$$A > B, \quad A = B, \quad A < B$$

Procedure: Refer to page 3

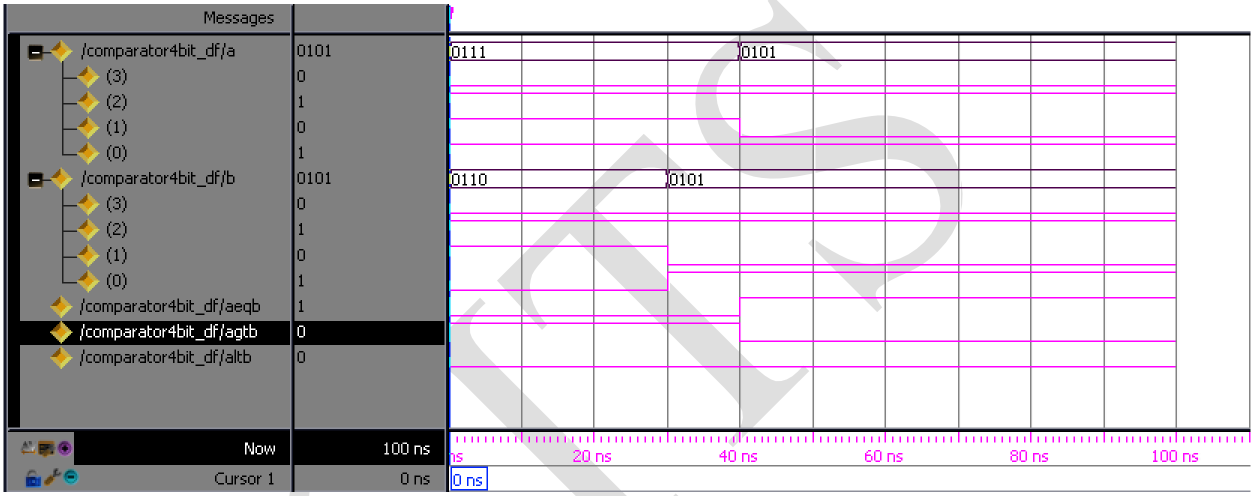
VHDL CODE:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY comparator4bit_df IS
  port(a,b: in std_logic_vector(3 downto 0);
          agtb,aeqb,altb : out std_logic);
```

```
END ENTITY comparator4bit_df;
ARCHITECTURE dataflow OF comparator4bit_df IS
BEGIN
        aeqb <= '1' when a=b else '0';
        agtb <= '1' when a>b else '0';
        altb <= '1' when a<b else '0';
END ARCHITECTURE dataflow;
```

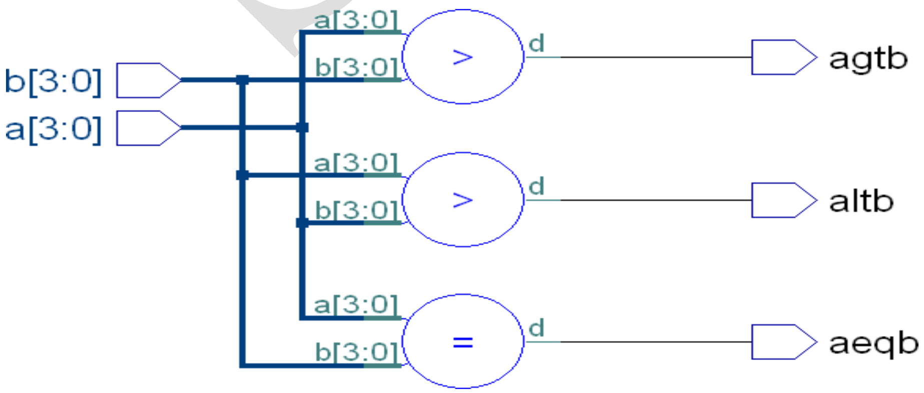
Simulation :

```
force a 0111 0ns,0101 40ns
force b 0110 0ns,0101 30ns
run 100ns
```



Messages
/comparator4bit_df/a 0101
(3) 0
(2) 1
(1) 0
(0) 1
/comparator4bit_df/b 0101
(3) 0
(2) 1
(1) 0
(0) 1
/comparator4bit_df/aeqb 1
/comparator4bit_df/agtb 0
/comparator4bit_df/altb 0
0111
0101
0110
0101
Now 100 ns
Cursor 1 0 ns
ns
20 ns
40 ns
60 ns
80 ns
100 ns
0 ns

Synthesis diagram:



Conclusion:

The VHDL code for 4-bit Comparator using dataflow model is written, simulated and synthesized.

Experiment 6(b)

DESIGN OF 4-BIT COMPARATOR

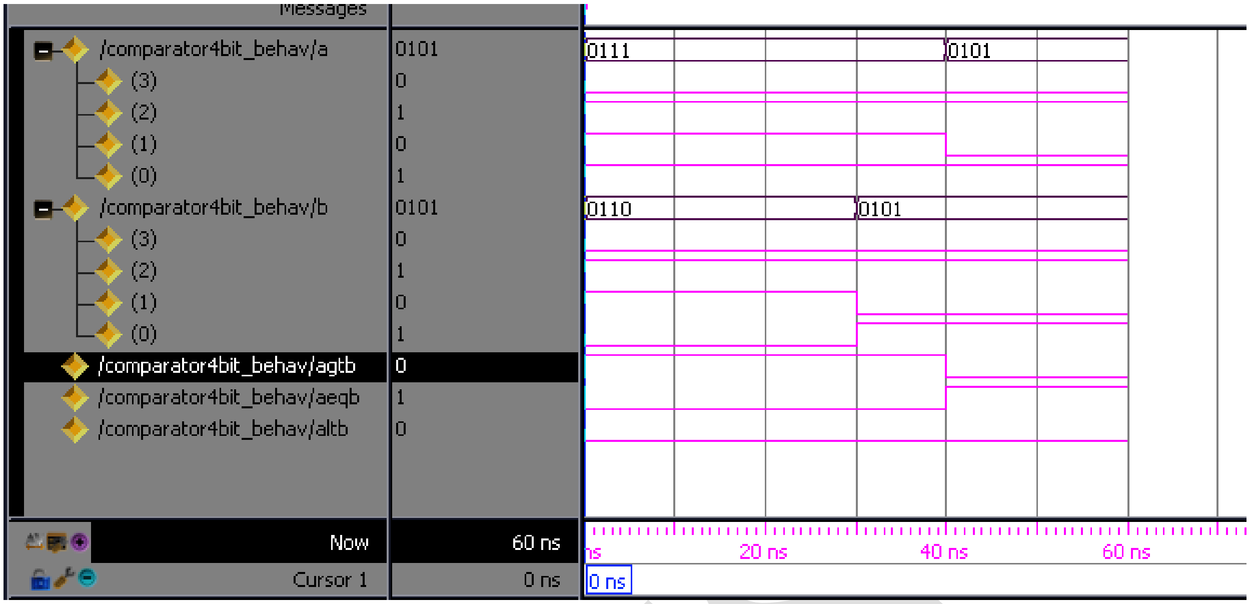
Aim: To write the VHDL code for 4-BIT COMPARATOR, simulate and synthesize using dataflow model

VHDL CODE(in Behavioural Modelling using if-elsif):

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY comparator4bit_behav IS
  port(a,b: in std_logic_vector(3 downto 0);
          agtb,aeqb,altb : out std_logic);
END ENTITY comparator4bit_behav;
ARCHITECTURE behavioural OF comparator4bit_behav IS
BEGIN
  process(a,b)
              begin
              if(a>b)then
                        agtb <='1';
                        aeqb <= '0';
                        altb <= '0';
            elsif(a<b) then
                         agtb <= '0';
                         aeqb <= '0';
                         altb <= '1';
            elsif(a=b) then
                        agtb <='0';
                        aeqb <= '1';
                        altb <= '0';
            else
                        agtb<='0';
                        aeqb <='0';
                        altb <= '0';
            end if;
         end process;
END ARCHITECTURE behavioural;
```

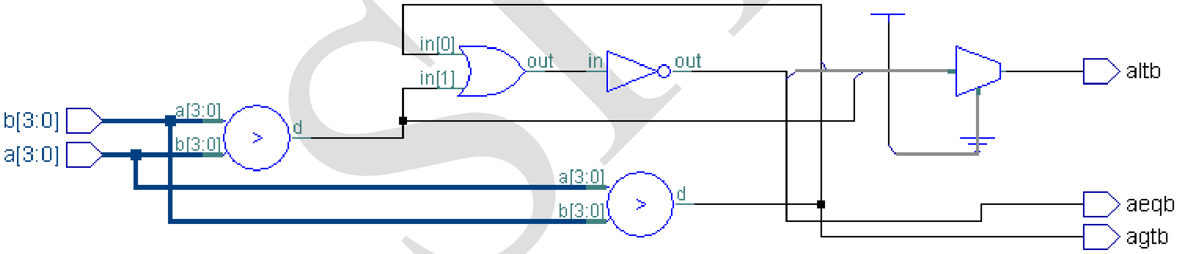
Simulation:

```
force a 0111 0ns,0101 40ns
force b 0110 0ns,0101 30ns
run 60ns
```



```
Messages
/comparator4bit_behav/a        0101
    (3)                        0
    (2)                        1
    (1)                        0
    (0)                        1
/comparator4bit_behav/b        0101
    (3)                        0
    (2)                        1
    (1)                        0
    (0)                        1
/comparator4bit_behav/agtb     0
/comparator4bit_behav/aeqb     1
/comparator4bit_behav/altb     0
Now                            60 ns
Cursor 1                       0 ns
```

Synthesis diagram:



Conclusion:

The VHDL code for 4-bit comparator using dataflow model is written, simulated and synthesized.

Experiment 7(a)

DESIGN OF HALF ADDER

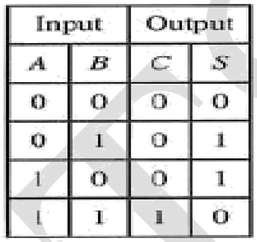
Aim: To write the VHDL code for Half adder, simulate and synthesize**.**

Tools Required:

1. FPG Advantage
   i. Xilinx ISE 9.2

Theory :

The **half adder** adds two one-bit binary numbers *A* and *B*. It has two outputs, *Sum S* and *Carry C* .



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

Truth table

Procedure: Refer to page 3

VHDL Code (using Dataflow Modelling):

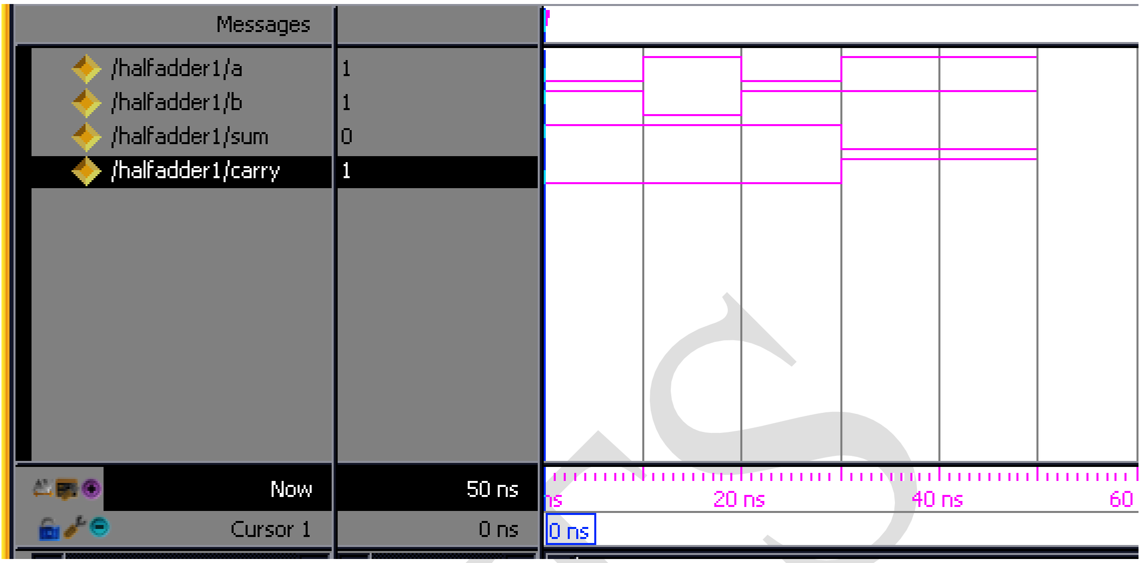
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY halfadder1 IS
port(a: in std_logic;
     b:in std_logic;
     carry:out std_logic;
     sum:out std_logic);
END ENTITY halfadder1;
ARCHITECTURE dataflow OF halfadder1 IS
BEGIN
  sum <=a xor b;
  carry <= a and b;
END ARCHITECTURE dataflow;
```

Simulations:

force a 0 0ns, 1 10ns,0 20ns,1 30ns

force b 1 0ns,0 10ns,1 20ns

run 50ns



Messages

/halfadder1/a 1

/halfadder1/b 1

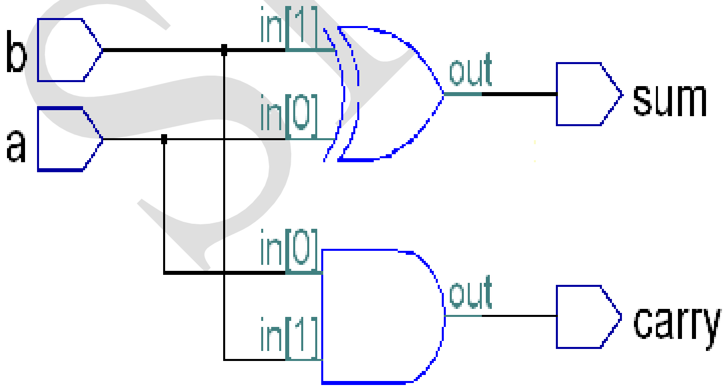
/halfadder1/sum 0

/halfadder1/carry 1

Now 50 ns

Cursor 1 0 ns

Synthesis Diagrams:



Conclusion:

The VHDL code for half adder is written and simulated and synthesized.

Experiment 7(b)

DESIGN OF FULL ADDER

Aim: To write the VHDL code for Full adder, simulate and synthesize using dataflow model.

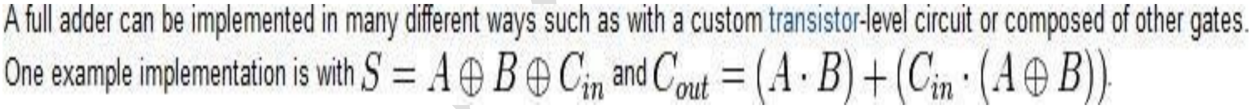
Tools Required:

1. FPG Advantage
   i. Xilinx ISE 9.2

Theory :

A **full adder** adds binary numbers and accounts for values carried in as well as out. A one-bit full adder adds three one-bit numbers, often written as *A*, *B*, and C_{in}; *A* and *B* are the operands, and C_{in} is a bit carried in from the next less significant stage.

Inputs			Outputs	
A	B	C_{in}	C_{out}	S
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
1	1	0	1	0
0	0	1	0	1
1	0	1	1	0
0	1	1	1	0
1	1	1	1	1



A full adder can be implemented in many different ways such as with a custom transistor-level circuit or composed of other gates. One example implementation is with $S = A \oplus B \oplus C_{in}$ and $C_{out} = (A \cdot B) + (C_{in} \cdot (A \oplus B))$.

Procedure: Refer to page 3

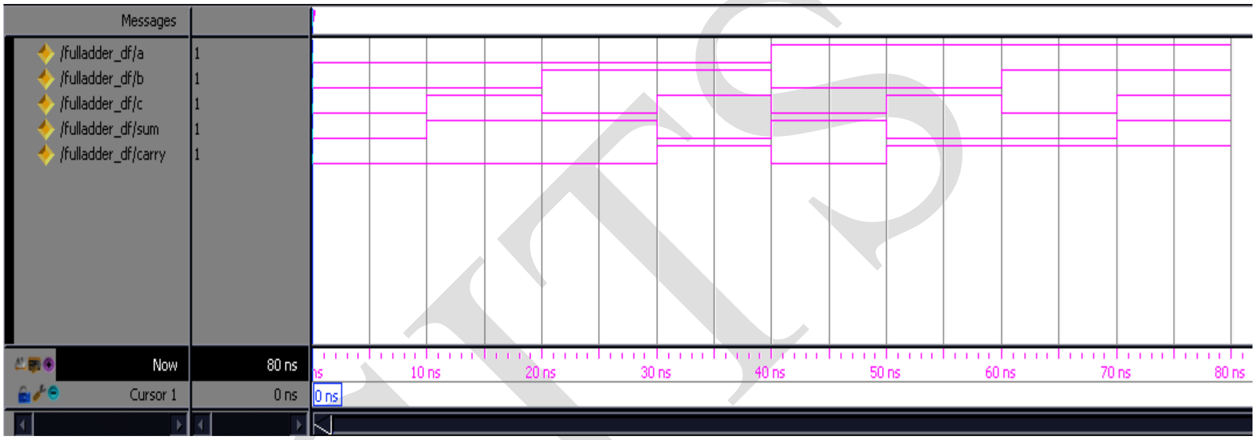
VHDL code (using dataflow Modelling):

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY fulladder_df IS
port(A,B,C : in std_logic;
     sum,carry : out std_logic);
```

```
END ENTITY fulladder_df;
ARCHITECTURE dataflow OF fulladder_df IS
BEGIN
  sum <= A xor B xor C;
  carry <= ((A and B) or (B and C) or (C and A));
END ARCHITECTURE dataflow;
```

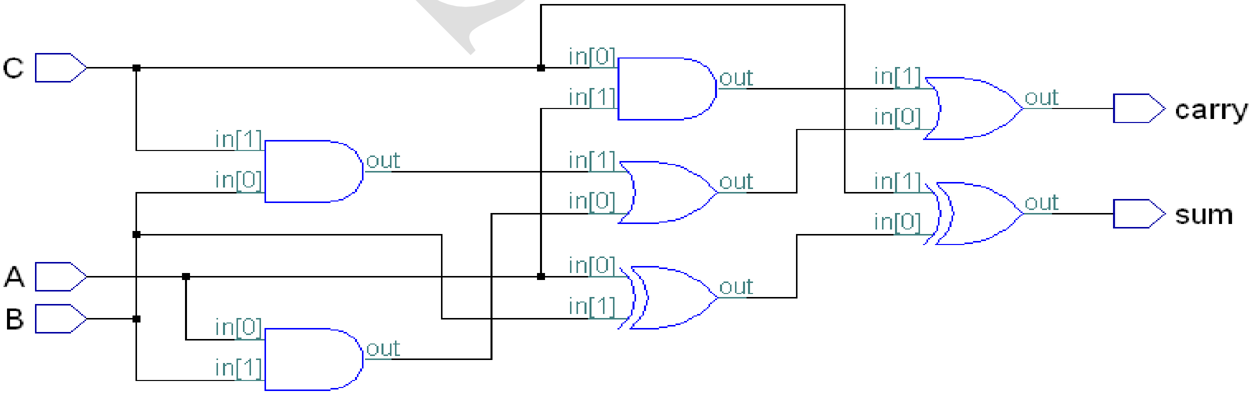
Simulations:

```
Force a 0 0ns,1 40ns
Force b 0 0ns,1 20ns,0 40ns,1 60ns
Force c 0 0ns,1 10ns,0 20ns,1 30ns,0 40ns,1 50ns,0 60ns,1 70ns
Run 80ns
```



```
Messages
/fulladder_df/a      1
/fulladder_df/b      1
/fulladder_df/c      1
/fulladder_df/sum    1
/fulladder_df/carry  1
Now                  80 ns
Cursor 1             0 ns
```

Synthesis diagrams:



Conclusion:

The VHDL code for full adder using Dataflow modeling is written, simulated and synthesized.

Experiment 7(c)

DESIGN OF FULL ADDER

Aim: To write the VHDL code for Full adder, simulate and synthesize using Structural model.

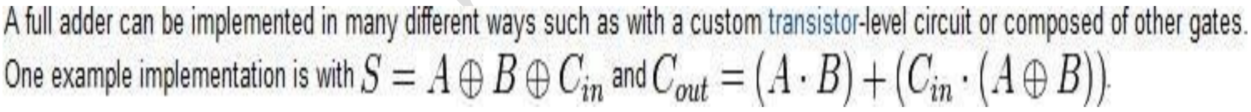
Tools Required:

1. FPG Advantage
   ii. Xilinx ISE 9.2

Theory :

A **full adder** adds binary numbers and accounts for values carried in as well as out. A one-bit full adder adds three one-bit numbers, often written as *A*, *B*, and C_{in}; *A* and *B* are the operands, and C_{in} is a bit carried in from the next less significant stage.

Inputs			Outputs	
A	B	C_{in}	C_{out}	S
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
1	1	0	1	0
0	0	1	0	1
1	0	1	1	0
0	1	1	1	0
1	1	1	1	1



A full adder can be implemented in many different ways such as with a custom transistor-level circuit or composed of other gates. One example implementation is with $S = A \oplus B \oplus C_{in}$ and $C_{out} = (A \cdot B) + (C_{in} \cdot (A \oplus B))$.

Procedure: Refer to page 3

VHDL Code(using Structural Modelling):

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
```

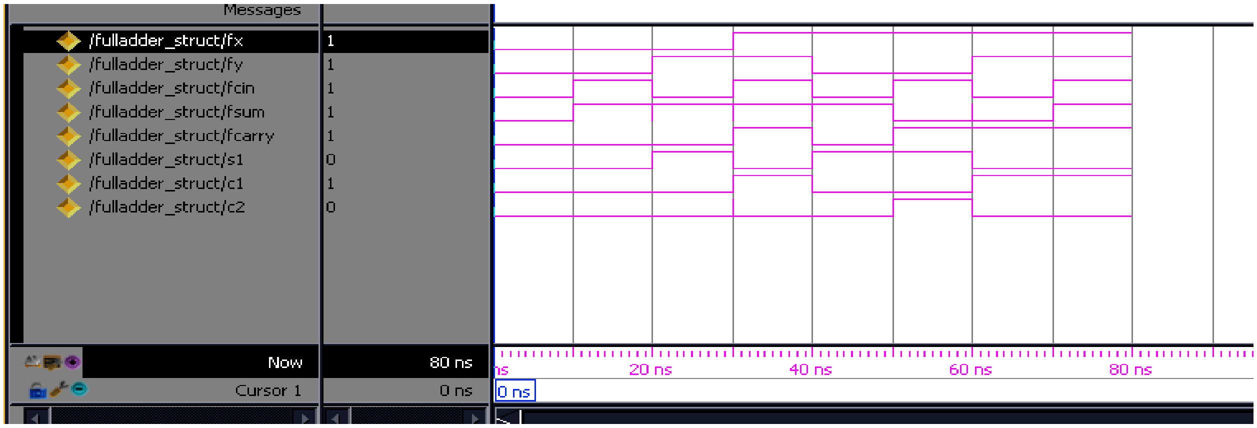
```
ENTITY fulladder_struct IS
  port(Fx,Fy,Fcin: in std_logic;
        Fsum,Fcarry : out std_logic);
END ENTITY fulladder_struct;
ARCHITECTURE struct OF fulladder_struct IS
signal s1,c1,c2:std_logic;
component HA port(A,B : in std_logic;
                 sum,carry : out std_logic)
end component;
BEGIN
            u1: HA port map(Fx,Fy,s1,c1);
            u2: HA port map(s1,Fcin,Fsum,c2);
            Fcarry <= c1 or c2;
END ARCHITECTURE struct;
```

Internal program Halfadder(HA):

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE  ieee.std_logic_arith.all;
ENTITY HA IS
  port(A,B : in std_logic;
          sum,carry: out std_logic);
END ENTITY HA;
ARCHITECTURE dataflow OF HA IS
BEGIN
            sum <= A xor B;
            carry <= A and B;
END ARCHITECTURE dataflow;
```

Simulation:

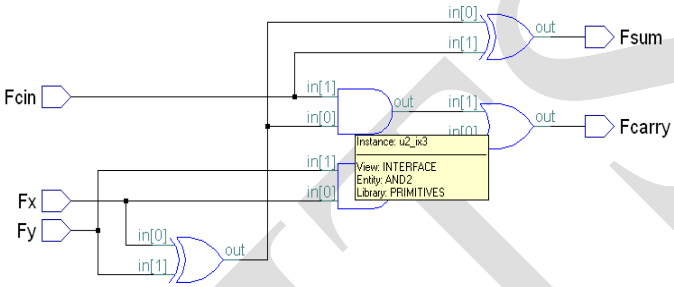
```
force Fx 0 0ns,1 30ns
  force Fy 0 0ns, 1 20ns,0 40ns, 1 60ns
force Fcin 0 0ns, 1 10ns,0 20ns,1 30ns,0 40ns,1 50ns,0 60ns,1 70ns
run 80ns
```



```
Messages
/fulladder_struct/fx      1
/fulladder_struct/fy      1
/fulladder_struct/fcin    1
/fulladder_struct/fsum    1
/fulladder_struct/fcarry  1
/fulladder_struct/s1      0
/fulladder_struct/c1      1
/fulladder_struct/c2      0

Now        80 ns
Cursor 1   0 ns
```

Synthesis diagram:



Conclusion:

The VHDL code for full adder using Structural modeling is written, simulated and synthesized.

Experiment 7(d)

DESIGN OF FULL ADDER

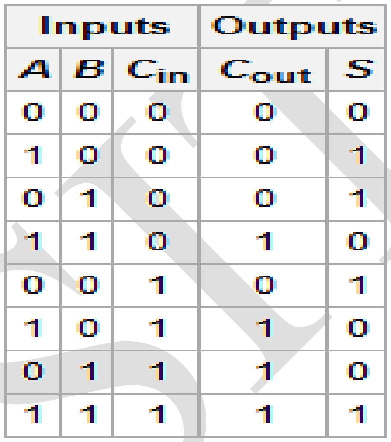
Aim: To write the VHDL code for Full adder, simulate and synthesize using Behavioral model.

Tools Required:

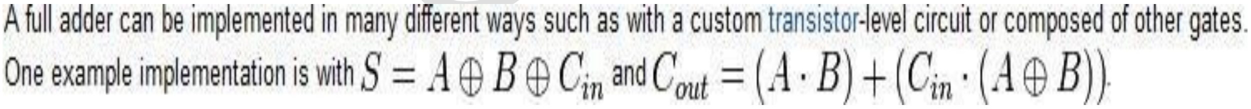
1. FPG Advantage
   i. Xilinx ISE 9.2

Theory :

A **full adder** adds binary numbers and accounts for values carried in as well as out. A one-bit full adder adds three one-bit numbers, often written as *A*, *B*, and C_{in}; *A* and *B* are the operands, and C_{in} is a bit carried in from the next less significant stage.



Inputs			Outputs	
A	*B*	C_{in}	C_{out}	*S*
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
1	1	0	1	0
0	0	1	0	1
1	0	1	1	0
0	1	1	1	0
1	1	1	1	1



A full adder can be implemented in many different ways such as with a custom transistor-level circuit or composed of other gates. One example implementation is with $S = A \oplus B \oplus C_{in}$ and $C_{out} = (A \cdot B) + (C_{in} \cdot (A \oplus B))$.

Procedure: Refer to page 3

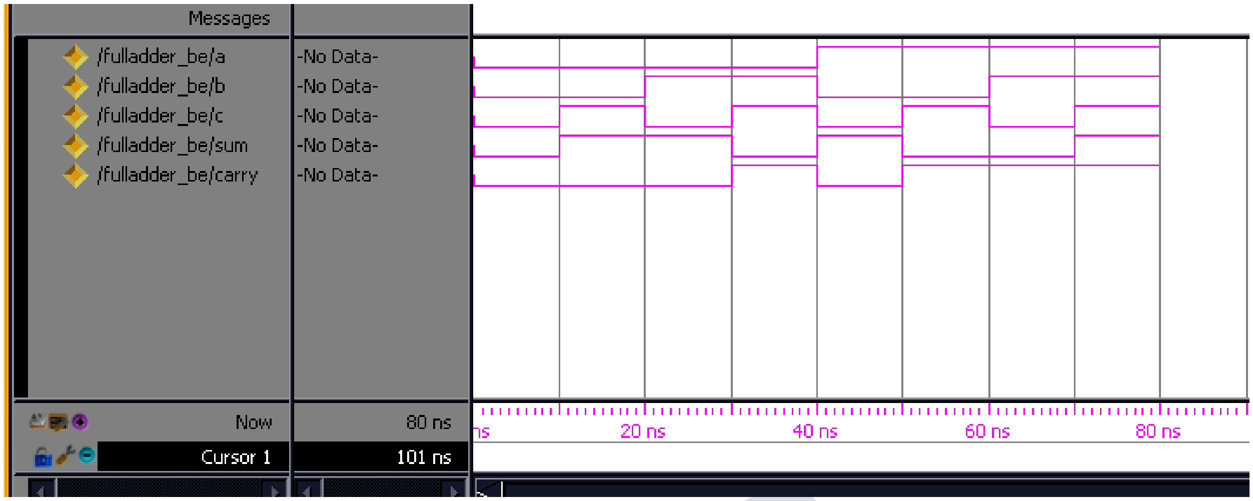
VHDL Code(in Behavioural Modelling using if-then-else Statement):

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY fulladder_be IS
port(a: in std_logic;
        b: in std_logic;
        c : in std_logic;
```

```
    sum : out std_logic;
   carry : out std_logic);
END ENTITY fulladder_be;
ARCHITECTURE behav OF fulladder_be IS
BEGIN
  process(a,b,c)
   begin
              if(a='0' and b= '0' and c='0')
                        then sum <= '0';carry <= '0';
     elsif(a='0' and b= '0' and c='1')
              then sum <= '1';carry <= '0';
              elsif(a='0' and b= '1' and c='0')
                        then sum <= '1';carry <= '0';
              elsif(a='0' and b= '1' and c='1')
                        then sum <= '0';carry <= '1';
          elsif(a='1' and b= '0' and c='0')
                        then sum <= '1';carry <= '0';
          elsif(a='1' and b= '0' and c='1')
                        then sum <= '0';carry <= '1';
          elsif(a='1' and b= '1' and c='0')
                        then sum <= '0';carry <= '1';
          elsif(a='1' and b= '1' and c='1')
                         then sum <= '1';carry <= '1';
          end if;
          end process;
END ARCHITECTURE behav;
```

Simulation :

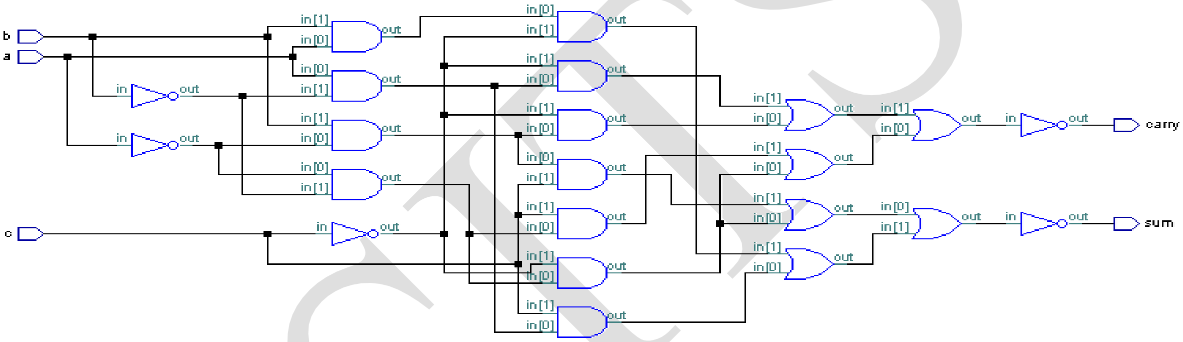
```
force a 0 0ns,1 40ns
force b 0 0ns,1 20ns,0 40ns,1 60ns
force c 0 0ns,1 10ns,0 20ns,1 30ns,0 40ns,1 50ns,0 60ns,1 70ns
run 80ns
```



Messages

/fulladder_be/a	-No Data-
/fulladder_be/b	-No Data-
/fulladder_be/c	-No Data-
/fulladder_be/sum	-No Data-
/fulladder_be/carry	-No Data-
Now	80 ns
Cursor 1	101 ns

Synthesis diagram:



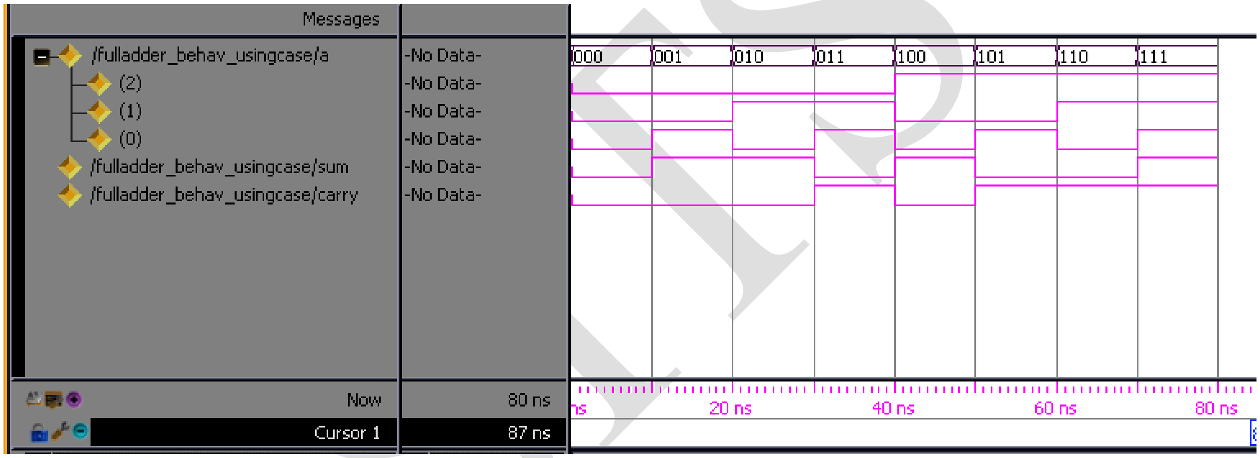
VHDL Code(in Behavioural Modelling style using Case Statement):

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY fulladder_behav_usingcase IS
  port(a: in std_logic_vector(2 downto 0);
      sum,carry :out std_logic);
 END ENTITY fulladder_behav_usingcase;
ARCHITECTURE behav OF fulladder_behav_usingcase IS
BEGIN
  process(a)
        begin
             case(a) is
             when "000" =>sum<='0';carry<='0';
             when "001" =>sum<='1';carry<='0';
```

```
                        when "010" =>sum<='1';carry<='0';
                        when "011" =>sum<='0';carry<='1';
                        when "100" =>sum<='1';carry<='0';
                         when "101" =>sum<='0';carry<='1';
                        when "110" =>sum<='0';carry<='1';
                        when "111" =>sum<='1';carry<='1';
                        when others =>sum<=Z';carry<='Z';
                end case;
            end process;
    END ARCHITECTURE behav;
```

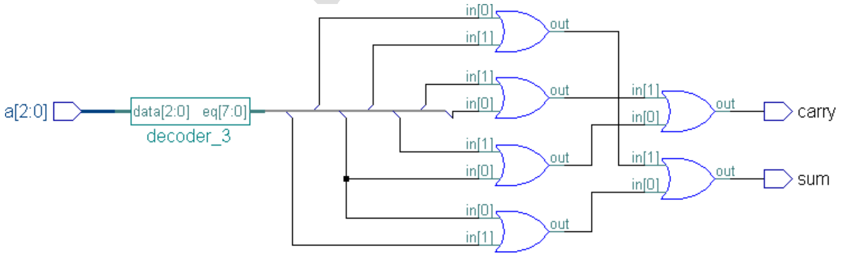
Simulation :

force a 000 0ns,001 10ns,010 20ns,011 30ns,100 40ns,101 50ns,110 60ns,111 70ns
run 80ns



```
Messages
/fulladder_behav_usingcase/a      -No Data-
    (2)                           -No Data-
    (1)                           -No Data-
    (0)                           -No Data-
/fulladder_behav_usingcase/sum    -No Data-
/fulladder_behav_usingcase/carry  -No Data-
Now                               80 ns
Cursor 1                          87 ns
```

Synthesis Diagram:



Conclusion:

The VHDL code for full adder using Behavioral (if-then-else, case statement) modeling is written, simulated and synthesized.

Experiment 8(a)

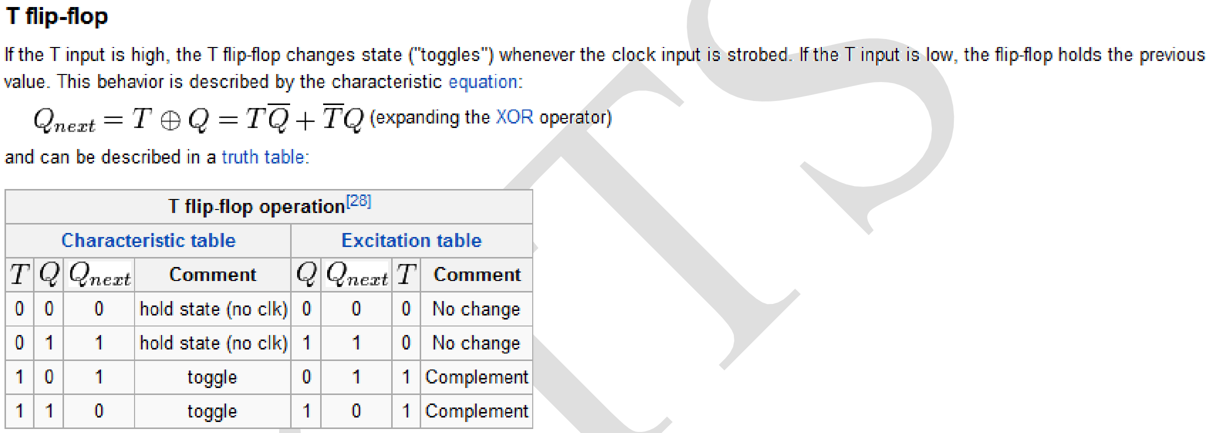
DESIGN OF T-FLIPFLOP

Aim: To write the VHDL code for T-FLIPFLOP, simulate and synthesize using structural model.

Tools Required:

1. FPG Advantage
   i. Xilinx ISE 9.2

Theory :



T flip-flop

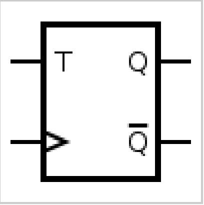
If the T input is high, the T flip-flop changes state ("toggles") whenever the clock input is strobed. If the T input is low, the flip-flop holds the previous value. This behavior is described by the characteristic equation:

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

and can be described in a truth table:

T flip-flop operation[28]

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



A circuit symbol for a T-type flip-flop

Procedure: Refer to page 3

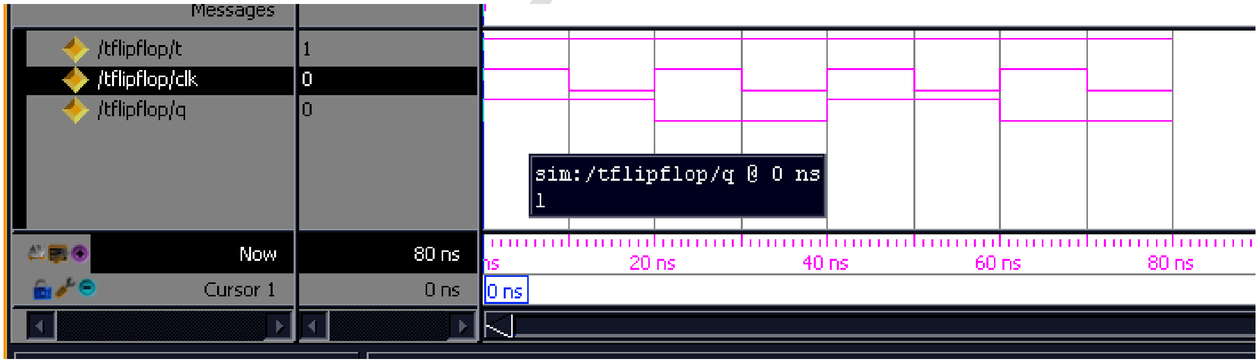
VHDL code(in Behavioural Modelling using if-else statement):

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
```

```
ENTITY tflipflop IS
  port(t,clk: in  std_logic;
              q : inout std_logic:='0');
END ENTITY tflipflop;
ARCHITECTURE behav OF tflipflop IS
BEGIN
  process(clk)
    begin
            q<='0';
             if(clk'event and clk='1')then
             if(t='1') then
            q <=not(q);
        else
                        q<=q;
                         end if;
                     end if;
            end process;
END ARCHITECTURE behav;
```

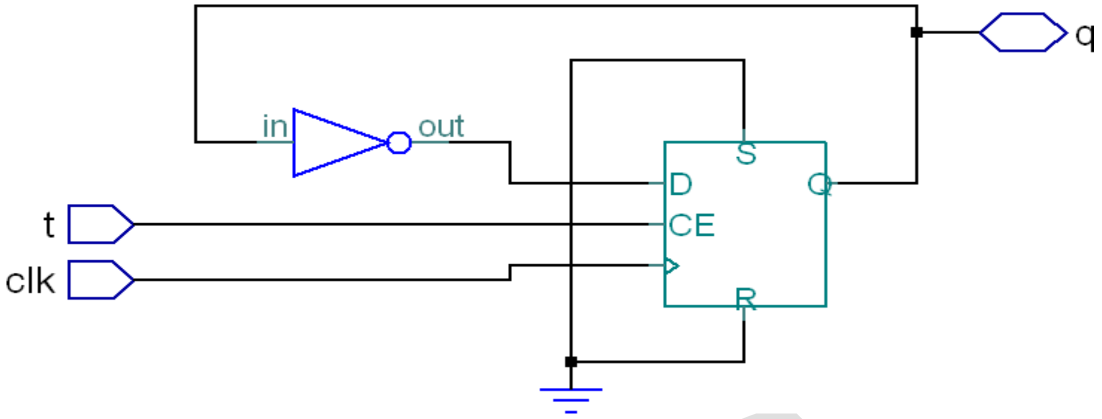
Simulation :

```
force t  1 0ns
force clk 1 0ns,0 10ns,1 20ns,0 30ns,1 40ns,0 50ns,1 60ns,0 70ns,1 80ns
run 80ns
```



```
sim:/tflipflop/q @ 0 ns
1
```

Synthesis diagram:



Conclusion:

The VHDL code for T-flipflop is written in Behavioural Modelling(using if-else statement), simulated and synthesized.

Experiment 8(b)

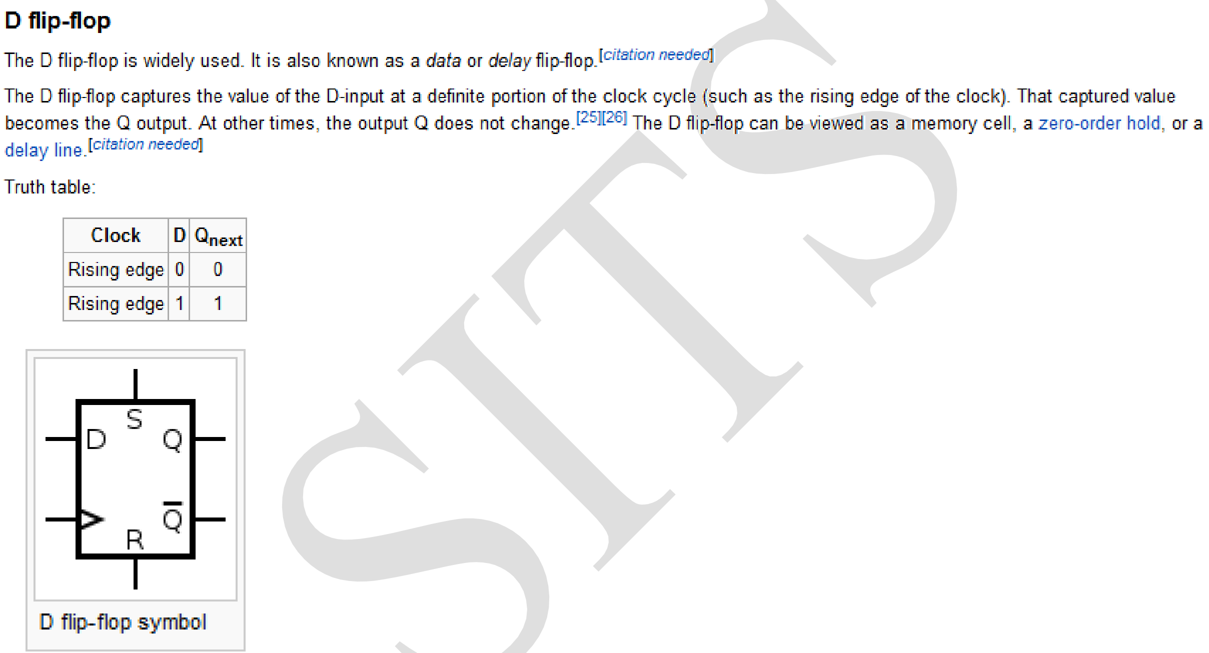
DESIGN OF D-FLIPFLOP

Aim: To write the VHDL code for D-FLIPFLOP,simulate and synthesize using behavioural model.

Tools Required:

1. FPG Advantage
    i. Xilinx ISE 9.2

Theory :



D flip-flop

The D flip-flop is widely used. It is also known as a *data* or *delay* flip-flop.[citation needed]

The D flip-flop captures the value of the D-input at a definite portion of the clock cycle (such as the rising edge of the clock). That captured value becomes the Q output. At other times, the output Q does not change.[25][26] The D flip-flop can be viewed as a memory cell, a zero-order hold, or a delay line.[citation needed]

Truth table:

Clock	D	Q_{next}
Rising edge	0	0
Rising edge	1	1

D flip-flop symbol

Procedure: Refer to page 3

VHDL code (in Behavioural Modelling using if-else Statement):

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY dflipflop IS
port(d,clk :in std_logic;
      q:out std_logic);
END ENTITY dflipflop;
ARCHITECTURE behav OF dflipflop IS
```

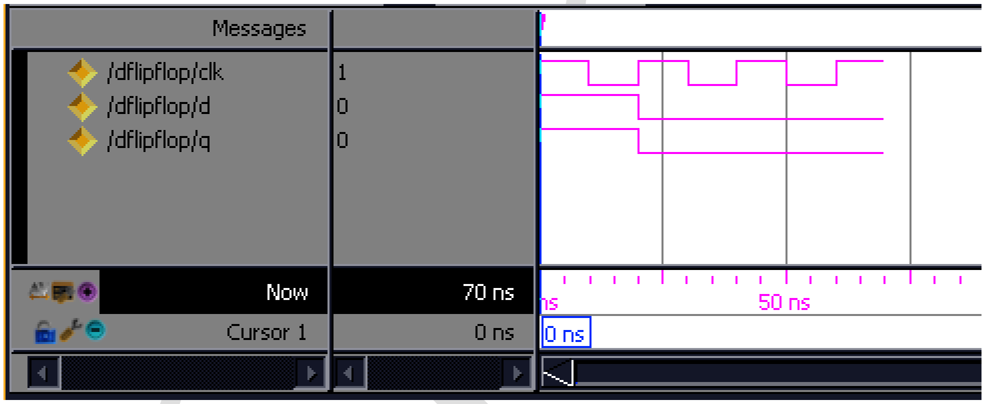
```
BEGIN
  process(clk)
         begin
                  if (clk'event and clk='1') then
                   q <= d;
         end if;
     end process;
END ARCHITECTURE behav;
```

Simulation :

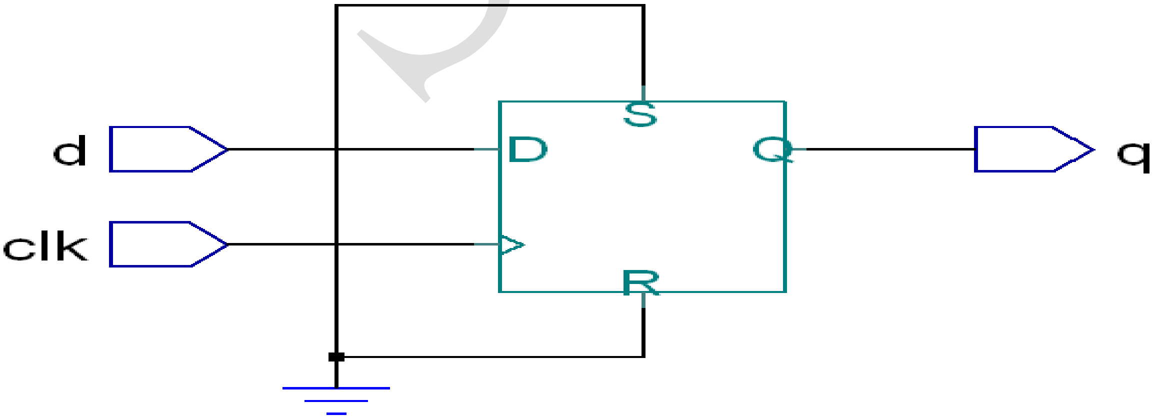
force d 1 0ns,0 20ns

force clk 1 0ns,0 10ns,1 20ns,0 30ns,1 40ns,0 50ns,1 60ns

run 70ns



Synthesis diagram:



Conclusion:

The VHDL code for T-flipflop is written in Behavioural Modelling(using if-else statement), simulated and synthesized.

Experiment 8(c)

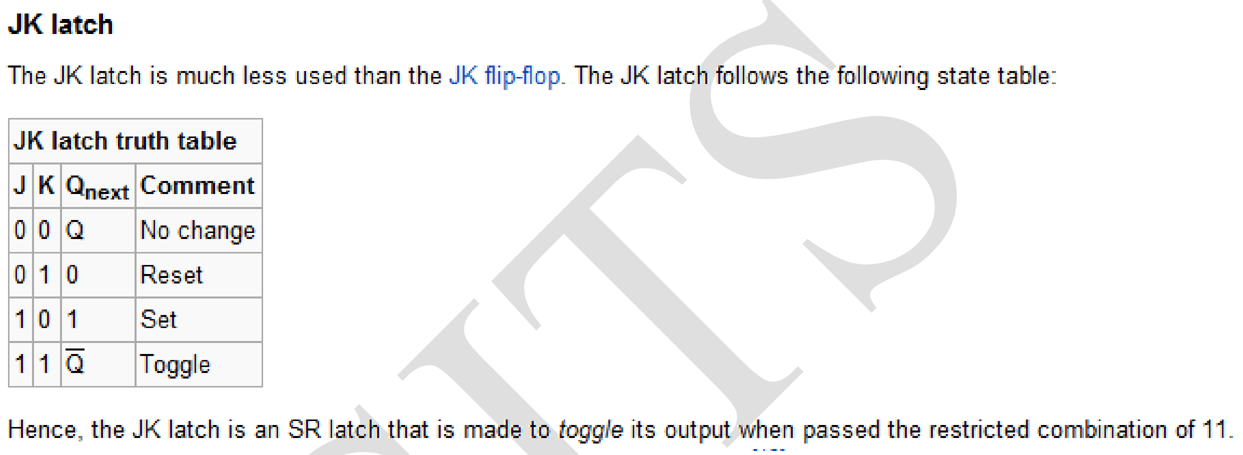
DESIGN OF JK-FLIPFLOP

Aim: To write the VHDL code for JK-FLIPFLOP,simulate and synthesize using behavioral model.

Tools Required:

1. FPG Advantage
   i. Xilinx ISE 9.2

Theory :



JK latch

The JK latch is much less used than the JK flip-flop. The JK latch follows the following state table:

JK latch truth table			
J	**K**	**Q_{next}**	**Comment**
0	0	Q	No change
0	1	0	Reset
1	0	1	Set
1	1	$\overline{Q}$	Toggle

Hence, the JK latch is an SR latch that is made to *toggle* its output when passed the restricted combination of 11.

Procedure: Refer to page 3

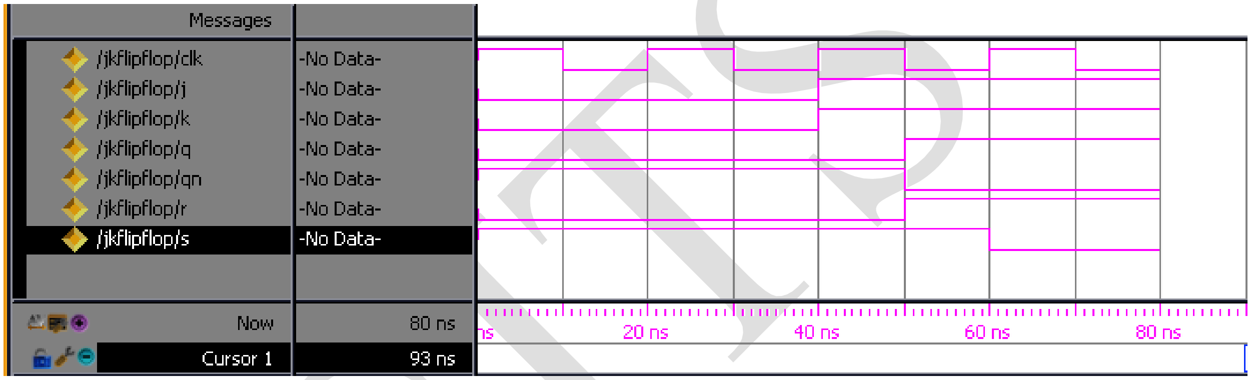
VHDL code:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY jkflipflop IS
port(s,r,j,k,clk : in std_logic;
                        q: inout std_logic;
                      qn: out std_logic:='1');
END ENTITY jkflipflop;
ARCHITECTURE behav OF jkflipflop IS
BEGIN
  process(s,r,clk)
            begin
                        if(r= '0' then q<= '0');
                        elsif s='0' then q<='1';
```

```
                elsif(clk='0' and clk'event) then
                q<=(j and (not q)) or ((not k)and q);
                end if;
        end process;
                qn <= not q;
END ARCHITECTURE behav;
```

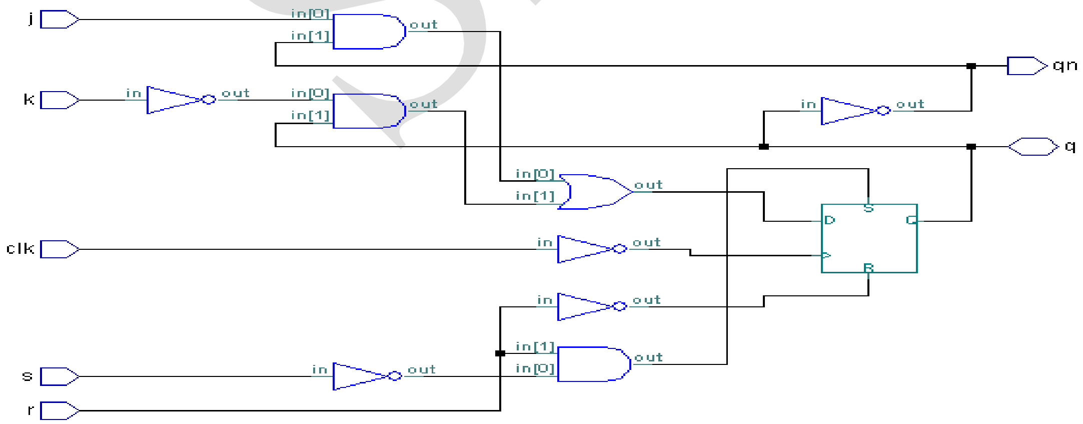
simulation:

```
Force clk 1 0ns,0 10ns,1 20ns,0 30ns,1 40ns,0 50ns,1 60ns,0 70ns
Force r 0 0ns,1 50ns
Force s 1 0ns, 0 60ns
Force j 0 0ns,1 40ns
Force k 0 0ns,1 40ns
Run  80ns
```



Messages	
/jkflipflop/clk	-No Data-
/jkflipflop/j	-No Data-
/jkflipflop/k	-No Data-
/jkflipflop/q	-No Data-
/jkflipflop/qn	-No Data-
/jkflipflop/r	-No Data-
/jkflipflop/s	-No Data-
Now	80 ns
Cursor 1	93 ns

Synthesis diagram:



Conclusion:

The VHDL code for JK Fliplop using behavioural modelling(using if-else) is written, simulated and synthesized.

CYCLE - 2

Experiment 1

DESIGN RULES

Aim: To study the design rules of CMOS

Design rules are the communication link between the designer specifying requirements and the fabricator who materializes them. Design rules are used to produce workable mask layouts from which the various layers in silicon will be formed or patterned.

The object of a set of design rules is to allow a ready translation of circuit design concepts, usually in stick diagram are symbolic form into actual geometry in silicon.

The first set of design rules are lambda based. These rules are straight forward and relatively simple to occupy. They are real and chips can be fabricated from mask layout using the lambda based rules set.

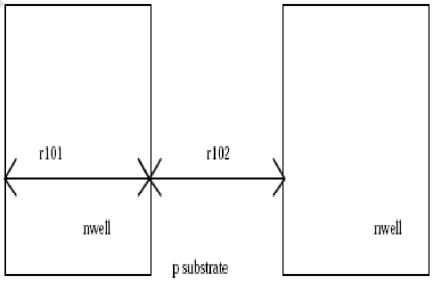
All paths in all layers will be dimensioned in lambda ‘λ’ and subsequently lambda can be allocated and appropriate value compatible with the feature size of the fabrication process.

N well Design Rules:

r101 Minimum well size: 12 λ

r102 Between wells: 12 λ

r110 Minimum surface: 144 λ^2



Diffusion Design Rules:

r201 Minimum N+ and P+ diffusion width : 4 λ

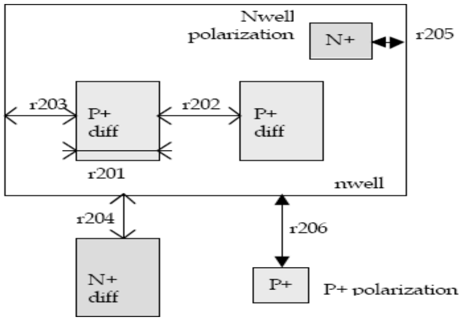
r202 Between two P+ and N+ diffusions : 4 λ

r203 Extra nwell after P+ diffusion : 6 λ

r204 Between N+ diffusion and nwell : 6 λ

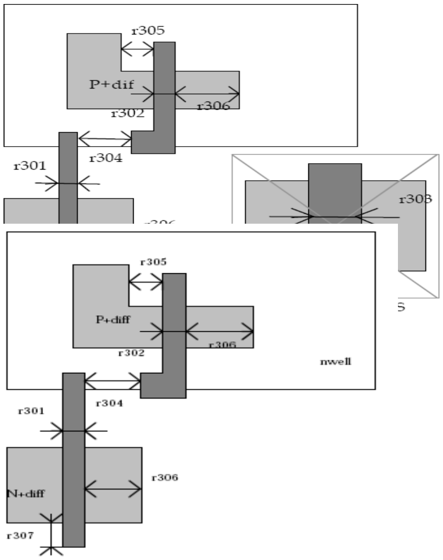
r205 Border of well after N+ polarization 2 λ

r206 Distance between Nwell and P+ polarization 6 λ



r210 Minimum surface : 24 λ^2

Polysilicon Design Rules:



r301 Polysilicon width : 2 λ

r302 Polysilicon gate on diffusion: 2 λ

r303 Polysilicon gate on diffusion for high voltage MOS: 4 λ

r304 Between two polysilicon boxes : 3 λ

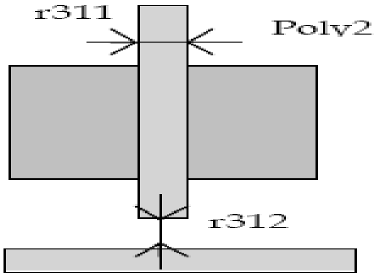
r305 Polysilicon vs. other diffusion : 2 λ

r306 Diffusion after polysilicon : 4 λ

r307 Extra gate after polysilicium : 3 λ

r310 Minimum surface : 8 λ^2

2nd Polysilicon Design Rules



r311 Polysilicon2 width : 2 λ

r312 Polysilicon2 gate on diffusion: 2 λ

Contact Design Rules

r401 Contact width : 2 λ

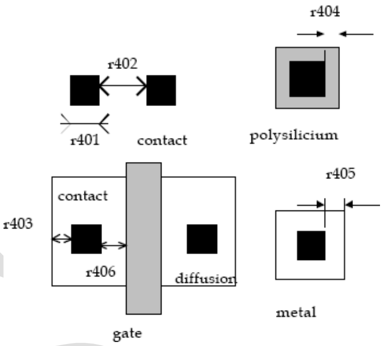
r402 Between two contacts : 5 λ

r403 Extra diffusion over contact: 2 λ

r404 Extra poly over contact: 2 λ

r405 Extra metal over contact: 2 λ

r406 Distance between contact and poly gate: 3 λ

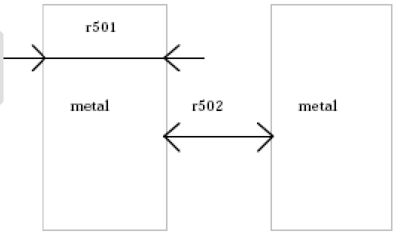


Metal & Via Design Rules

r501 Metal width : 4 λ

r502 Between two metals : 4 λ

r510 Minimum surface : 32 λ2



r601 Via width : 2 λ

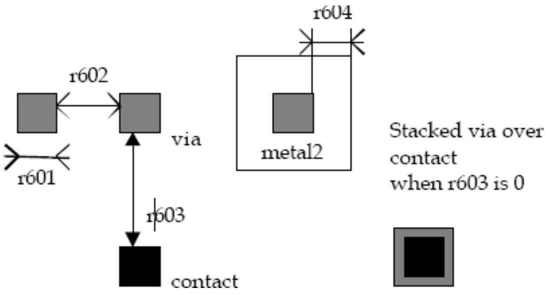
r602 Between two Via: 5 λ

r603 Between Via and contact:-0λ

r604 Extra metal over via: 2 λ

r605 Extra metal2 over via: 2 λ

When r603=0, stacked via over contact is allowed

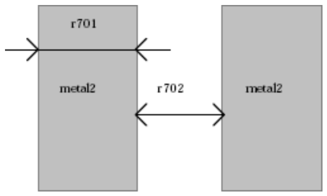


Metal2 & Via2 Design Rules

r701 Metal width: 4 λ

r702 Between two metal2 : 4 λ

r710 Minimum surface : 32 λ^2

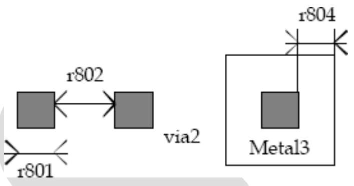


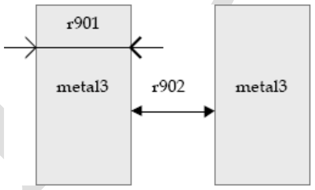
r801 Via2 width : 2 λ

r802 Between two Via2: 5 λ

r804 Extra metal2 over via2: 2 λ

r805 Extra metal3 over via2: 2 λ



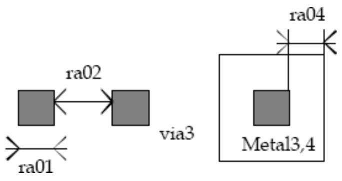


Metal 3 & Via 3 Design Rules

r901 Metal3 width: 4 λ

r902 Between two metal3 : 4 λ

r910 Minimum surface : 32 λ^2



ra01 Via3 width : 2 λ

ra02 Between two Via3: 5 λ

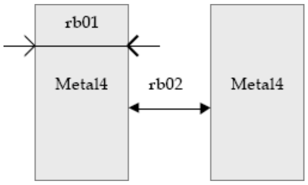
ra04 Extra metal3 over via3: 2 λ

ra05 Extra metal4 over via3: 2 λ

rb01 Metal4 width: 4 λ

rb02 Between two metal4 : 4 λ

rb10Minimum surface : 32 λ^2

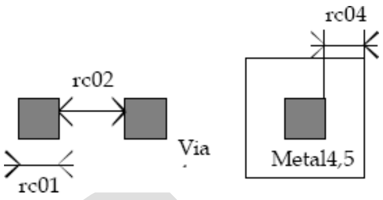


rc01 Via4 width : 2 λ

rc02 Between two Via4: 5 λ

rc04 Extra metal4 over via2: 3 λ

rc05Extra metal5 over via2: 3 λ

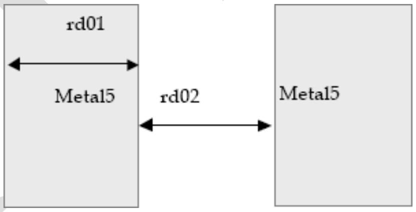


Metal 5 & Via 5 Design Rules

rd01 Metal5 width: 8 λ

rd02 Between two metal5 : 8 λ

rd10 Minimum surface : 100 λ2

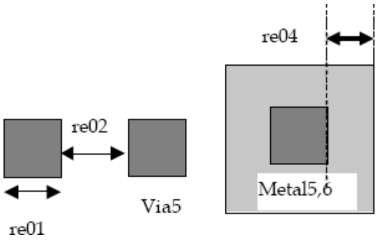


re01 Via5 width : 4 λ

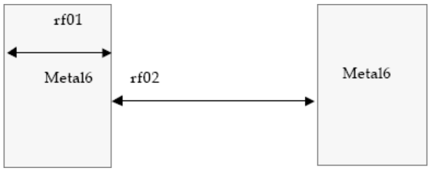
re02 Between two Via5: 6 λ

re04 Extra metal5 over via5: 3 λ

re05 Extra metal6 over via5: 3 λ



Metal 6 Design Rules

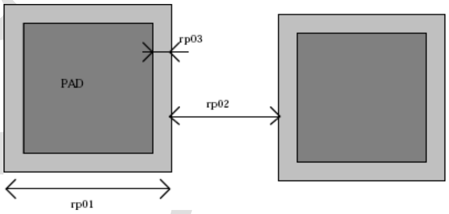


rf01 Metal6 width: 8 λ

rf02 Between two metal6 : 15 λ

rf10 Minimum surface : 300λ^2

Pad Design Rules



rp01 Pad width: 100 µm

rp02 Between two pads 100 µm

rp03 Opening in passivation v.s via : 5µm

rp04 Opening in passivation v.s metals: 5µm

rp05Between pad and unrelated active area : 20 µm

Experiment 2

BASIC LOGIC GATEs

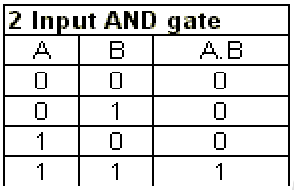
Aim: To design the digital schematics and corresponding layouts using **CMOS logic** for an **AND LOGIC** gate, **OR LOGIC** gate, **NOT LOGIC** gate and check the lambda based rules using DRC and verify its functionality.

Apparatus:

- DSCH2(logic editor & simulator)
- MICROWIND 3.1(layout editor & simulator)

Theory:

AND GateS: The AND gate is an electronic circuit that gives a **high** output (1) only if **all** its inputs are high. A dot (.) is used to show the AND operation i.e. A.B.

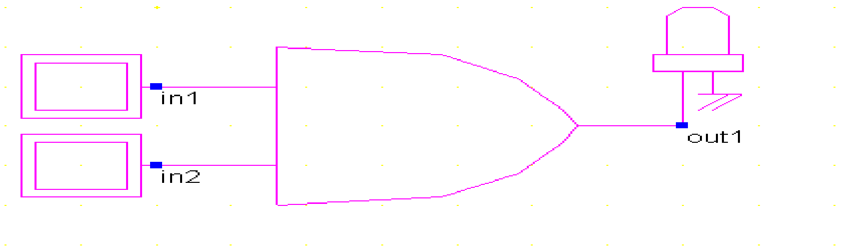


2 Input AND gate		
A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

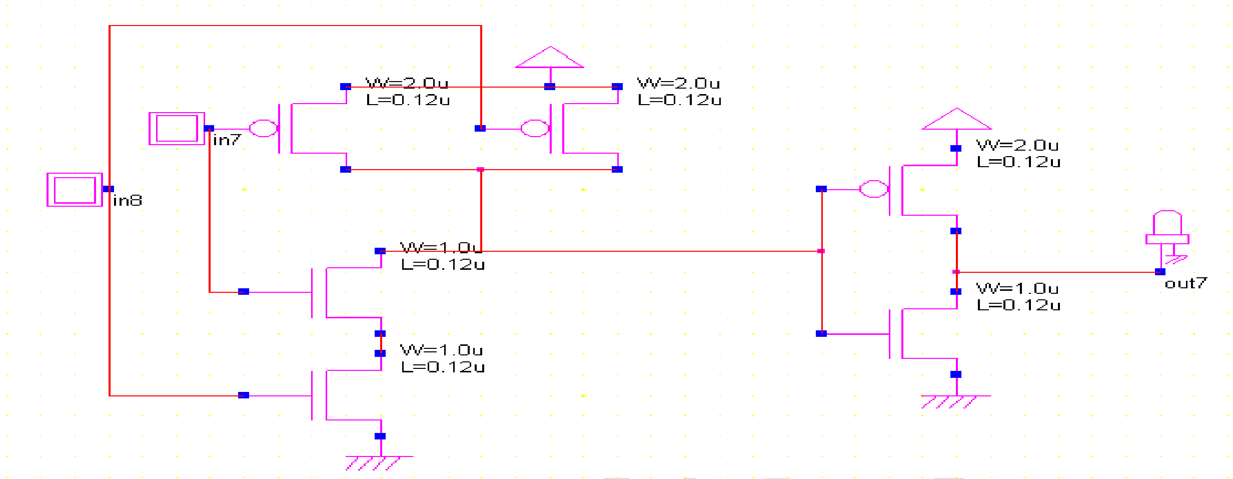
Procedure:

1. Open the DSch2 tool and draw the schematic diagram as per the circuit drawn.
2. Save the file and verify the functionality.
3. After that open Microwind 3.1 tool and draw the layout diagram as per the circuit drawn.
4. Save the file and verify the lambda rules by using DRC, then verify the functionality.

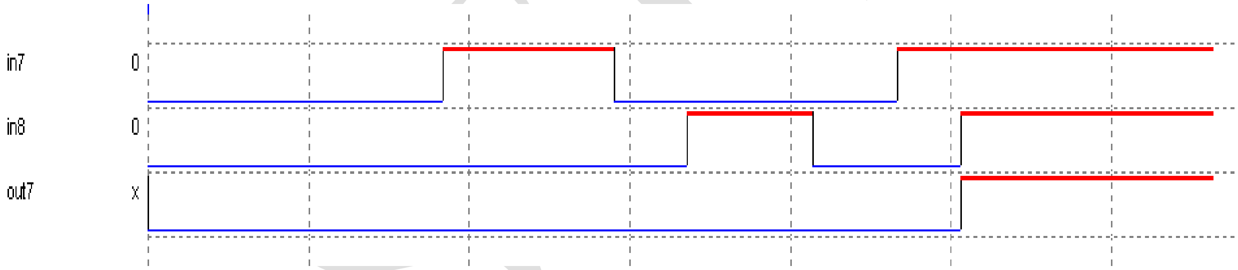
Digital Schematic Representation:



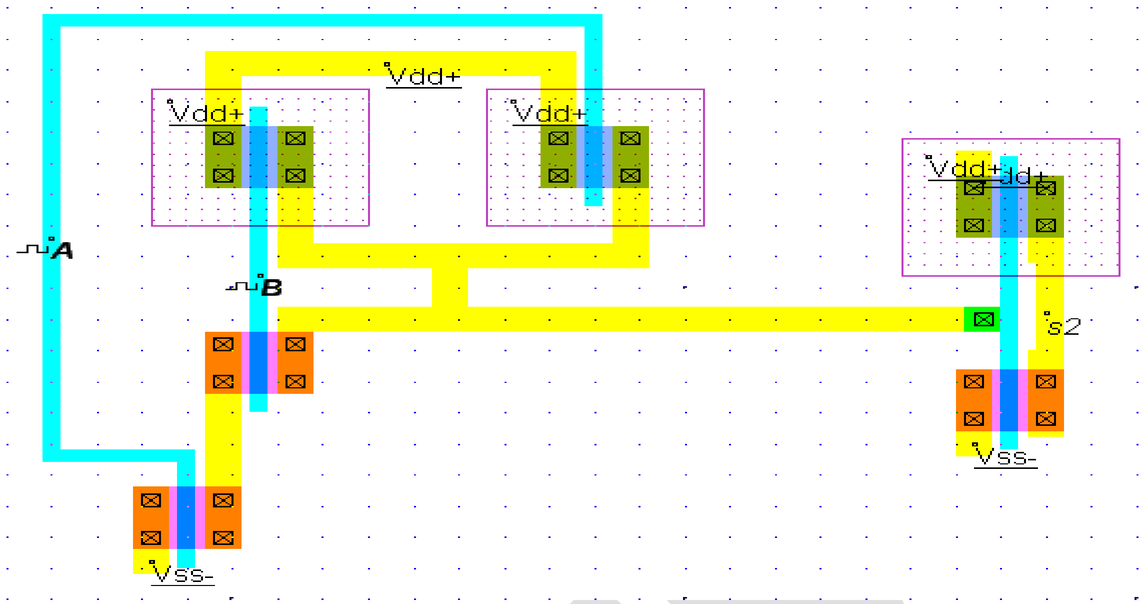
Transistor level design (CMOS logic):



Timing Diagram:

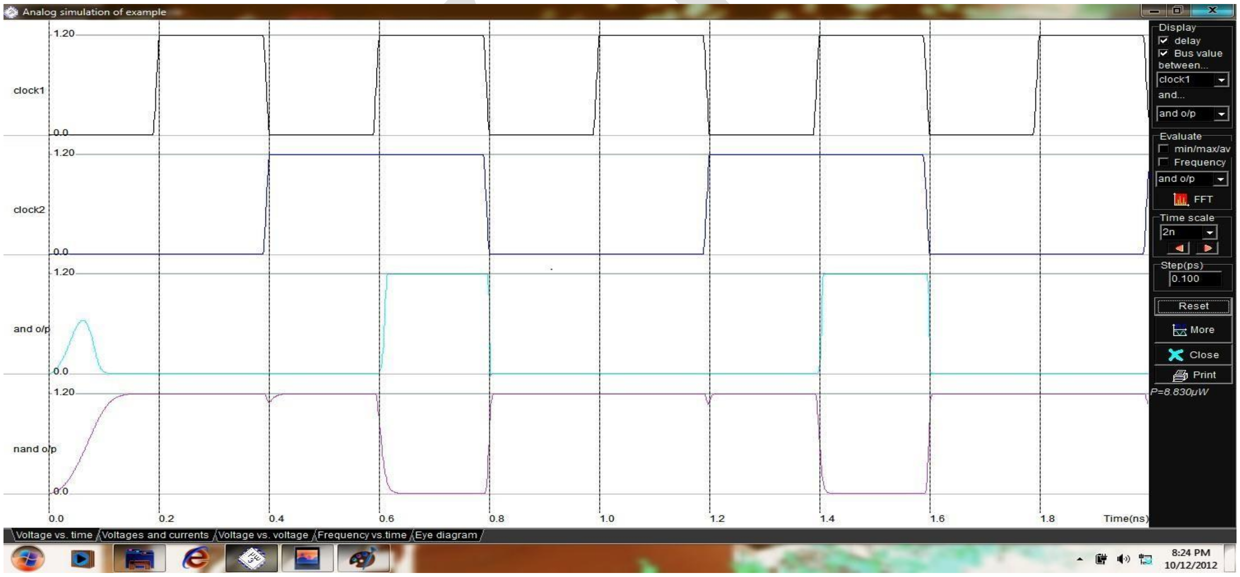


Semi-custom Layout:



Simulation:

Voltage –Time:



Display
☑ delay
☑ Bus value
between...
clock1
and...
and o/p

Evaluate
□ min/max/av
□ Frequency
and o/p

Time scale
2n

Step(ps)
0.100

P=8.830µW

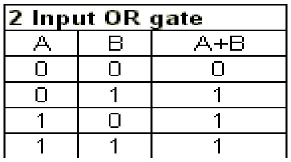
Conclusion:

The digital schematics and corresponding layouts using **CMOS logic** for an **AND LOGIC** gate are designed and the lambda based rules using DRC are checked and verified its functionality.

OR LOGIC GATE

OR Gate:

The OR gate is an electronic circuit that gives a high output (1) if **one or more** of its inputs are high. A plus (+) is used to show the OR operation.

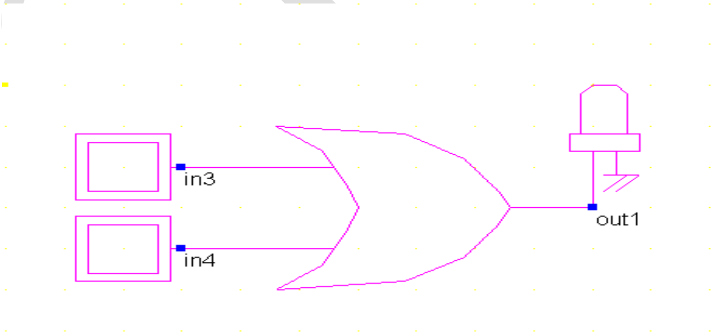


2 Input OR gate		
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

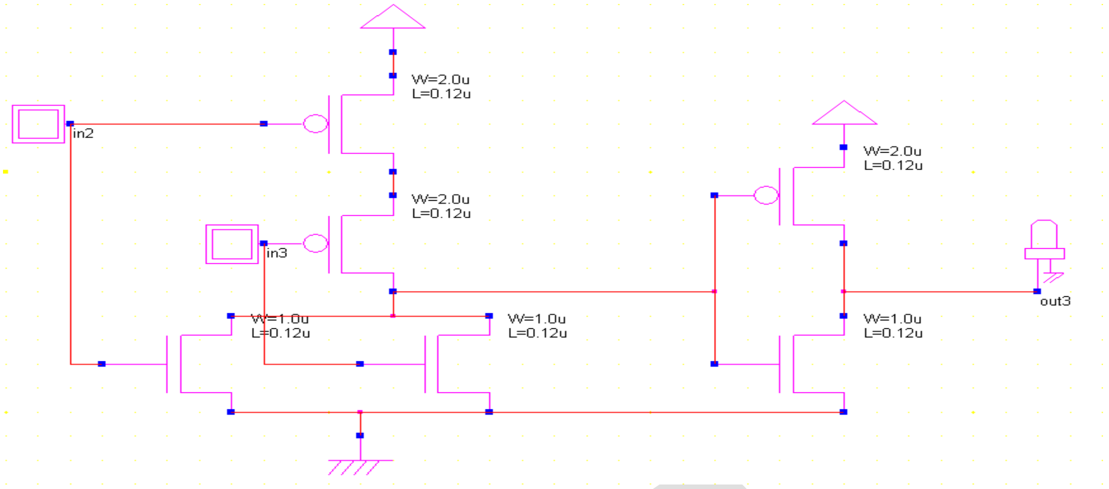
Procedure:

1. Open the DSch2 tool and draw the schematic diagram as per the circuit drawn.
2. Save the file and verify the functionality.
3. After that open Microwind 3.1 tool and draw the layout diagram as per the circuit drawn.
4. Save the file and verify the lambda rules by using DRC, then verify the functionality.

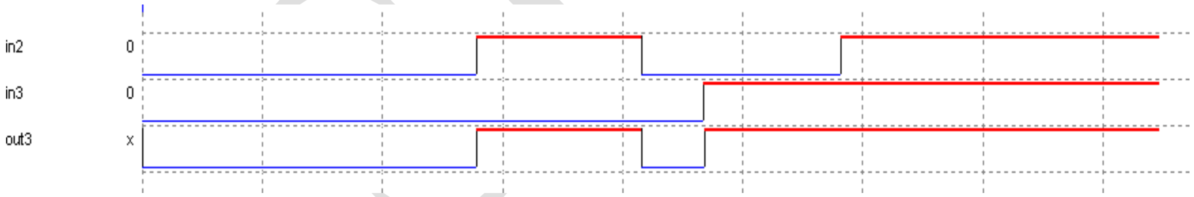
Digital Schematic Representation:



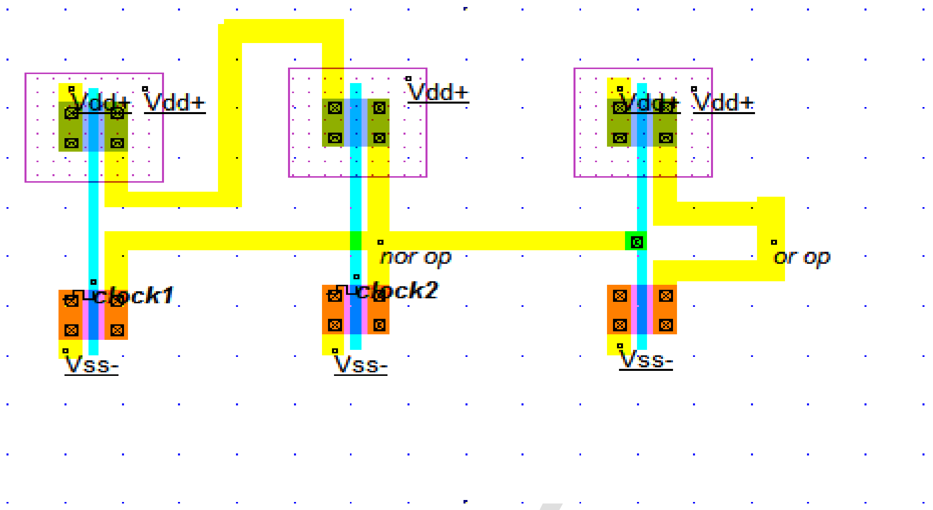
Transistor level design (CMOS logic):



Timing Diagram:

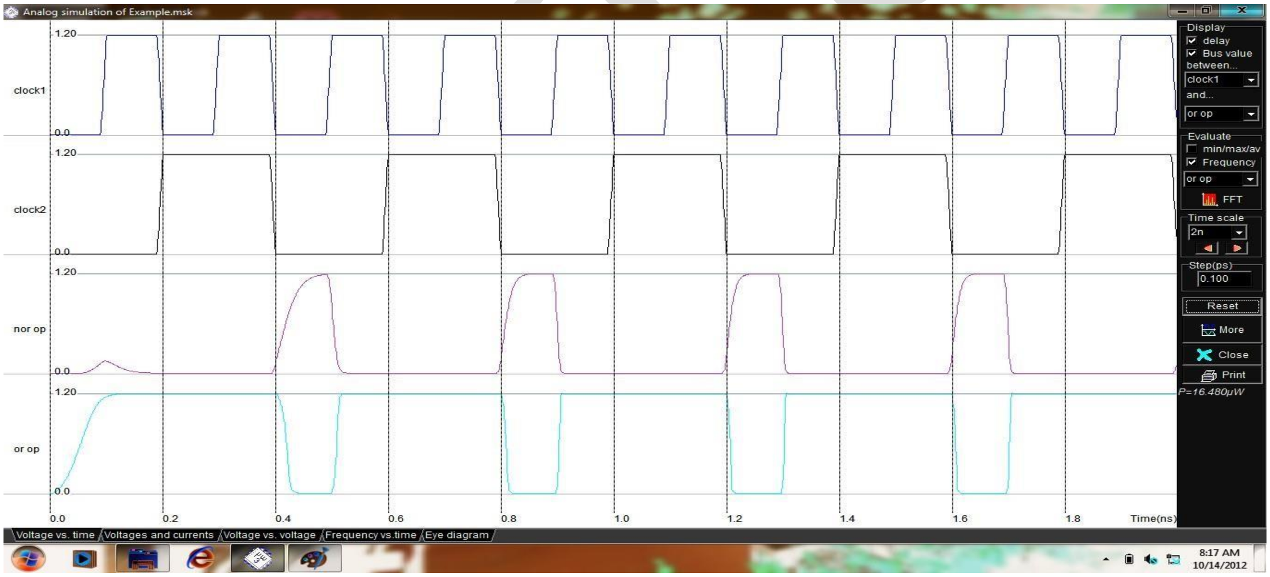


Semi-custom Layout:



Simulation:

Voltage –Time:


Analog simulation of Example.msk

Display
☑ delay
☑ Bus value between... clock1 and... or op

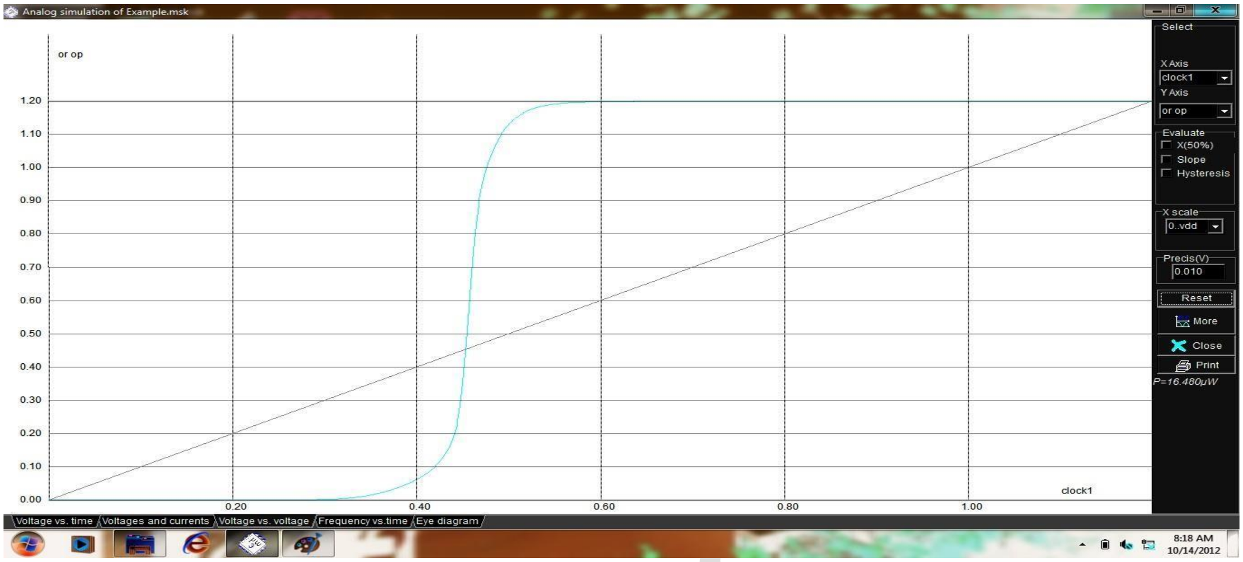
Evaluate
☐ min/max/av
☑ Frequency or op

Time scale 2n

Step(ps) 0.100

P=16.480µW

Voltage-voltage:



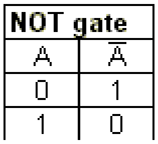
Conclusion:

The digital schematics and corresponding layouts using **CMOS logic** for an **OR LOGIC** gate are designed and the lambda based rules using DRC are checked and verified its functionality.

NOT LOGIC GATE

NOT Gate:

The NOT gate is an electronic circuit that produces an inverted version of the input at its output. It is also known as an *inverter*. If the input variable is A, the inverted output is known as NOT A. This is also shown as A', or A with a bar over the top.

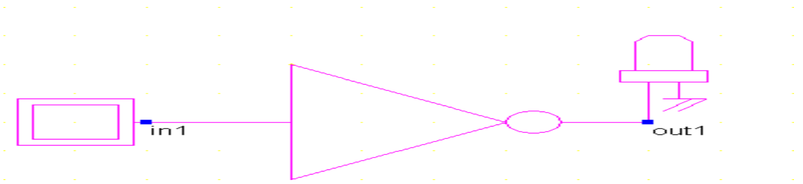


NOT gate	
A	$\overline{A}$
0	1
1	0

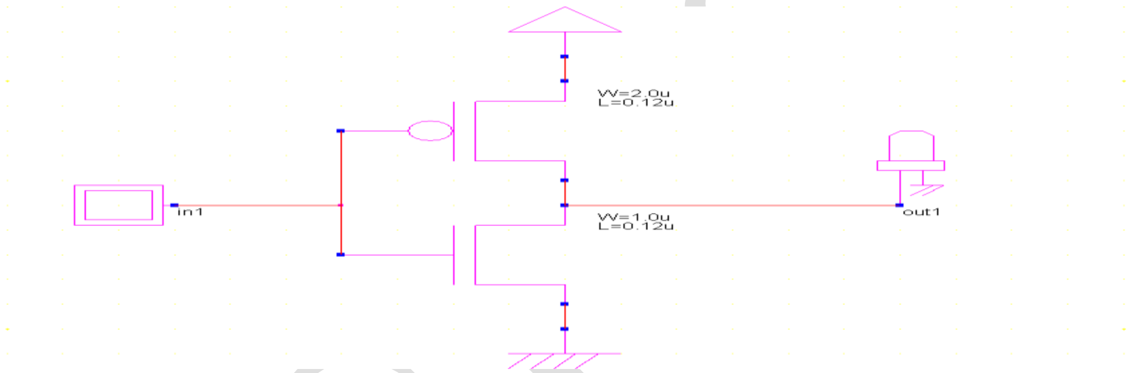
Procedure:

1. Open the DSch2 tool and draw the schematic diagram as per the circuit drawn.
2. Save the file and verify the functionality.
3. After that open Microwind 3.1 tool and draw the layout diagram as per the circuit drawn.
4. Save the file and verify the lambda rules by using DRC, then verify the functionality.

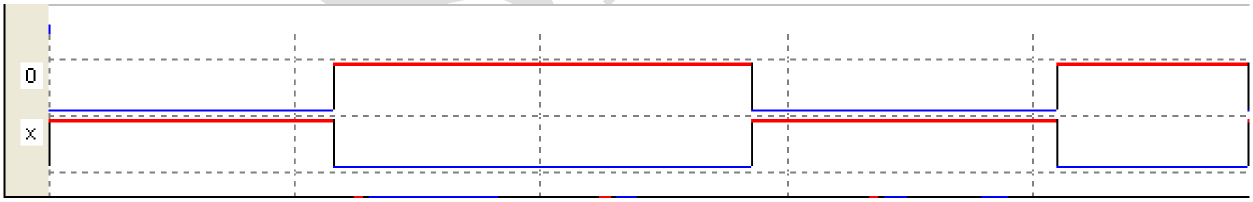
Digital Schematic Representation:



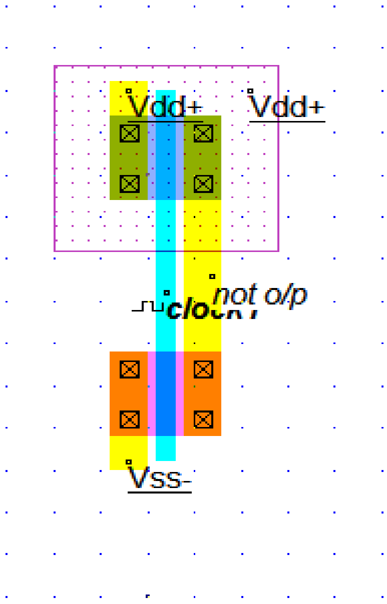
Transistor level design (CMOS logic):



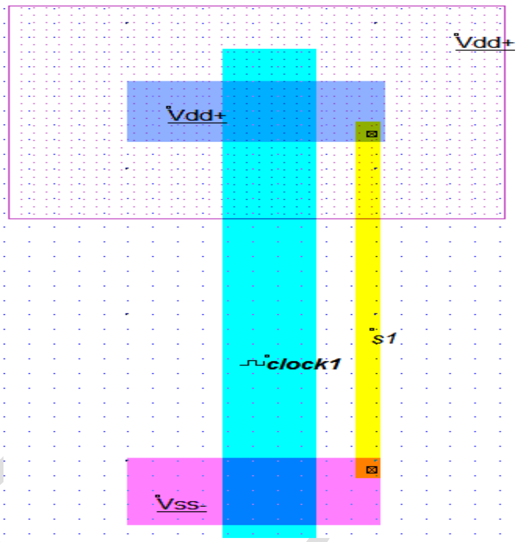
Timing Diagram:



Semi-custom Layout:

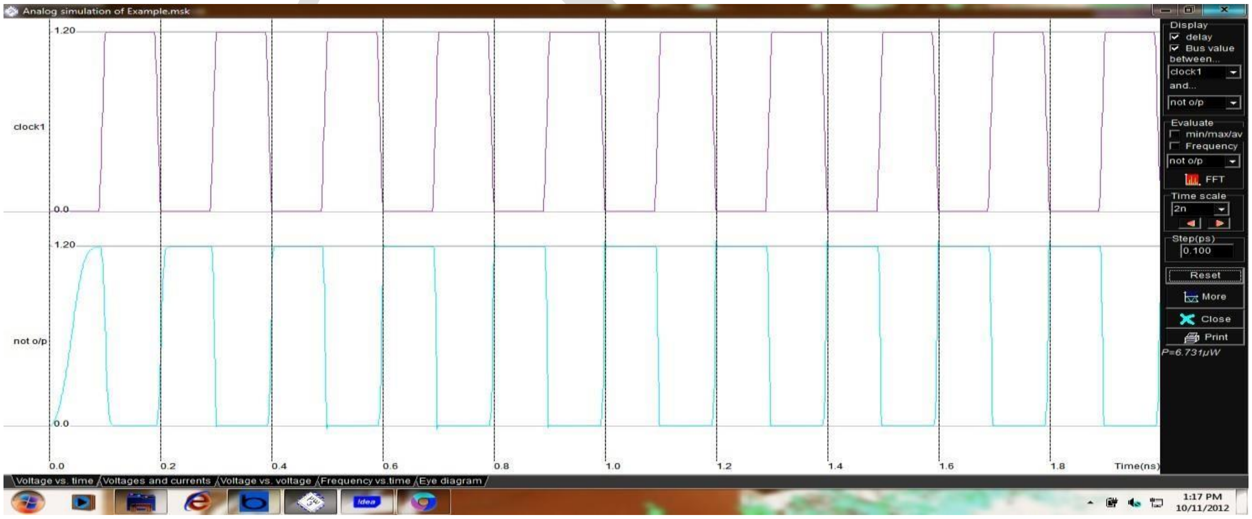


Full-custom Layout:



Simulation:

Voltage –Time:


Analog simulation of Example.msk

Display
☑ delay
☑ Bus value between...
clock1
and...
not o/p

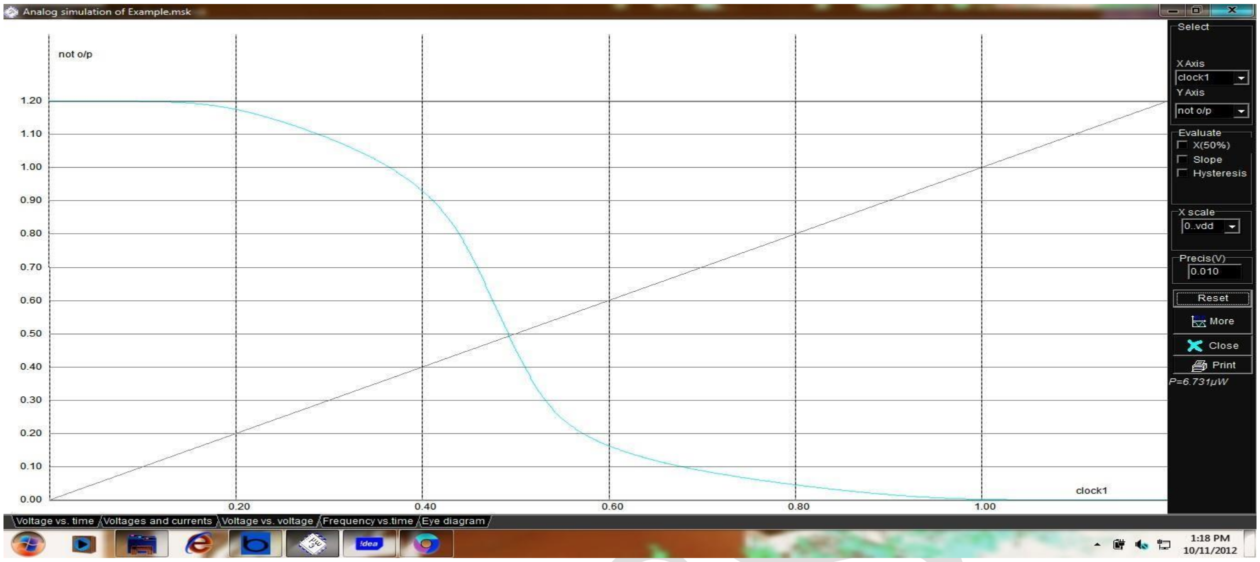
Evaluate
☐ min/max/av
☐ Frequency
not o/p

Time scale
2n

Step(ps)
0.100

P=6.731µW

Voltage-Voltage:



Conclusion:

The digital schematics and corresponding layouts using **CMOS logic** for an **NOT LOGIC** gate are designed and the lambda based rules using DRC are checked and verified its functionality.

Experiment 2(b)

NAND LOGIC GATE

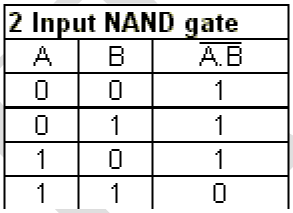
Aim: To design the digital schematics and corresponding layouts using **CMOS logic** for an **NAND LOGIC** gate and **NOR LOGIC** gate check the lambda based rules using DRC and verify its functionality.

Apparatus:

- DSCH2(logic editor & simulator)
- MICROWIND 3.1(layout editor & simulator)

THEORY:

NAND: This is a NOT-AND gate which is equal to an AND gate followed by a NOT gate. The outputs of all NAND gates are high if **any** of the inputs are low. The symbol is an AND gate with a small circle on the output. The small circle represents inversion

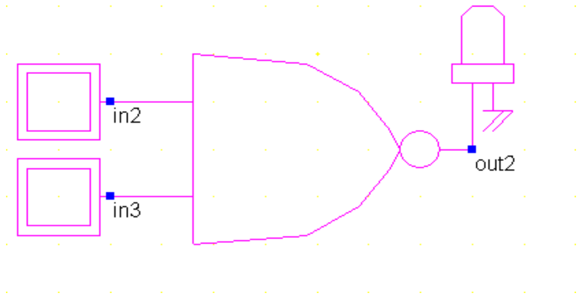


2 Input NAND gate		
A	B	$\overline{A.B}$
0	0	1
0	1	1
1	0	1
1	1	0

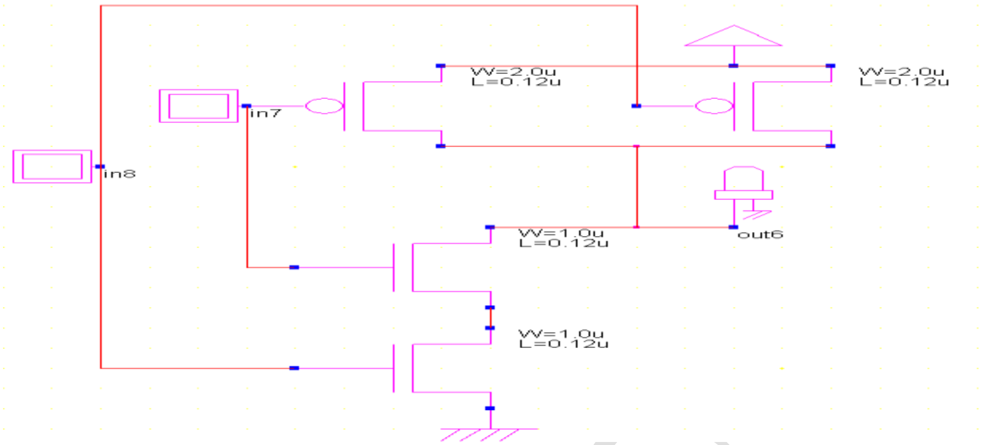
Procedure:

1. Open the DSch2 tool and draw the schematic diagram as per the circuit drawn.
2. Save the file and verify the functionality.
3. After that open Microwind 3.1 tool and draw the layout diagram as per the circuit drawn.
4. Save the file and verify the lambda rules by using DRC, then verify the functionality.

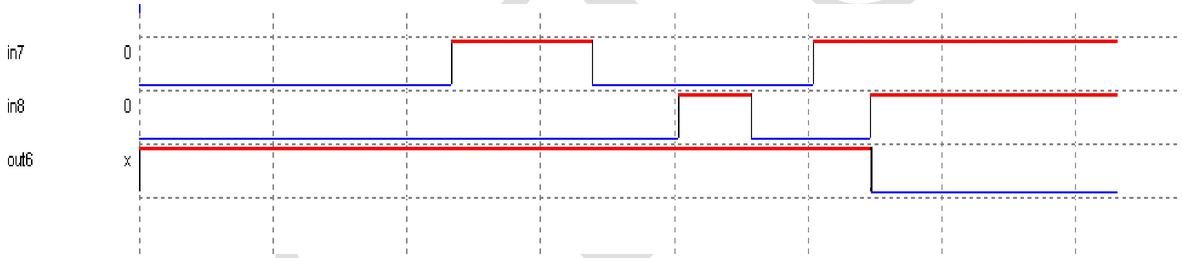
Digital Schematic Representation:



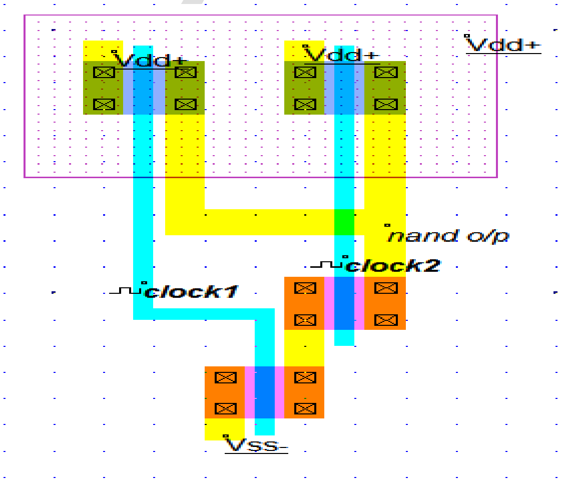
Transistor level design (CMOS logic):



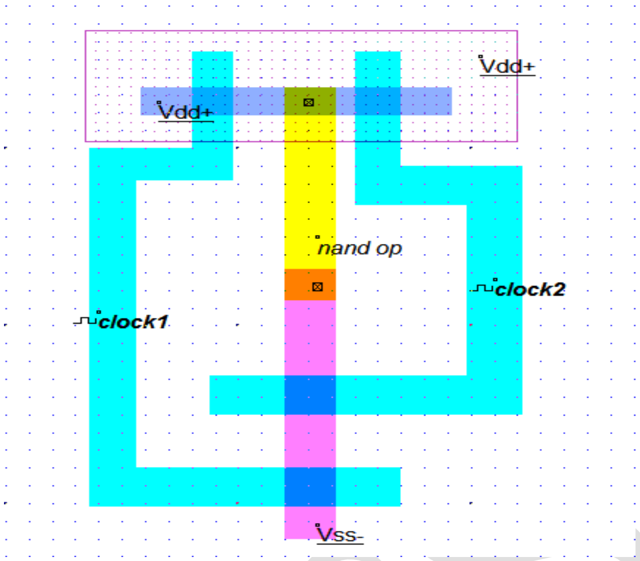
Timing Diagram:



Semi-custom Layout:

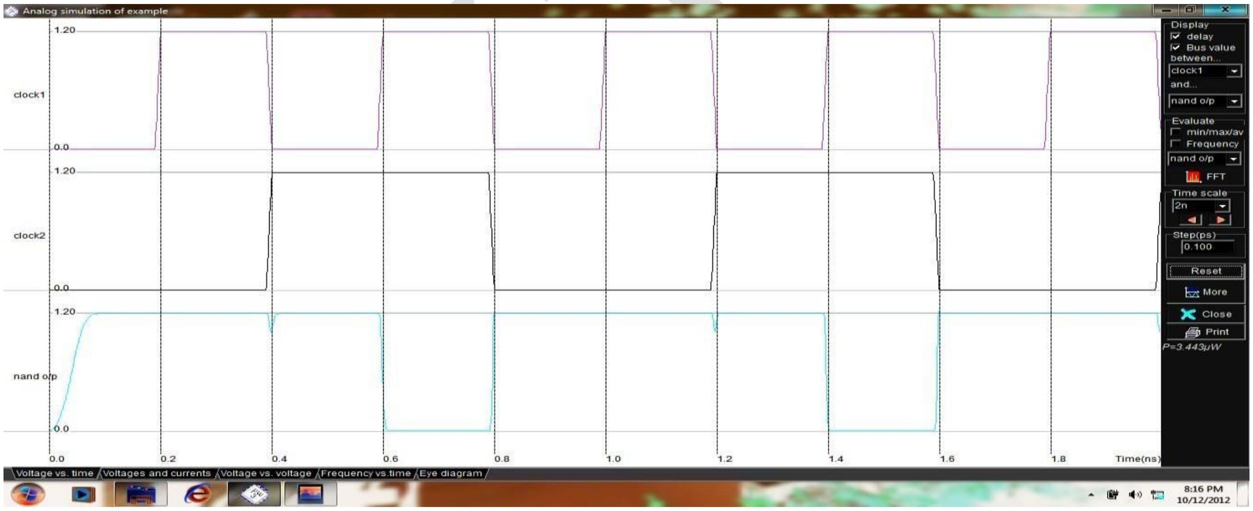


Full-Custom Layout:



Simulation:

Voltage –Time:



Display
☑ delay
☑ Bus value between...
clock1
and...
nand o/p

□ min/max/av
□ Frequency
nand o/p

Time scale 2n

Step(ps) 0.100

P=3.443µW

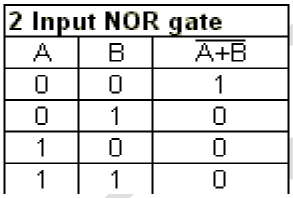
Conclusion:

The digital schematics and corresponding layouts using **CMOS logic** for an **NAND LOGIC** gate are designed and the lambda based rules using DRC are checked and verified its functionality.

NOR LOGIC GATE:

NOR Gate:

This is a NOT-OR gate which is equal to an OR gate followed by a NOT gate. The outputs of all NOR gates are low if **any** of the inputs are high.The symbol is an OR gate with a small circle on the output. The small circle represents inversion



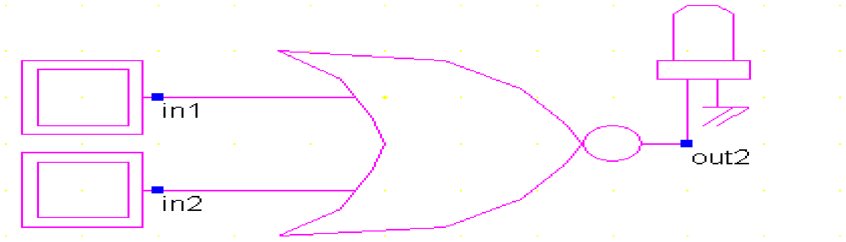
2 Input NOR gate

A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

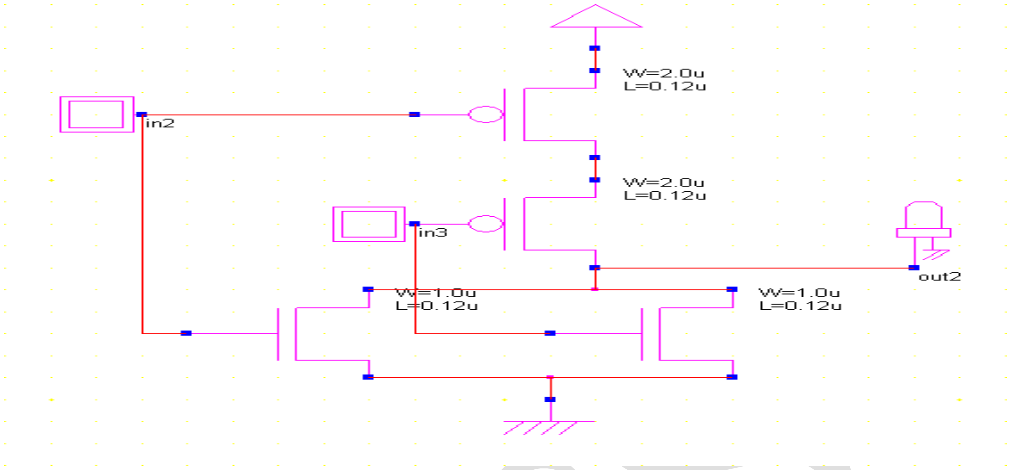
Procedure:

1. Open the DSch2 tool and draw the schematic diagram as per the circuit drawn.
2. Save the file and verify the functionality.
3. After that open Microwind 3.1 tool and draw the layout diagram as per the circuit drawn.
4. Save the file and verify the lambda rules by using DRC, then verify the functionality.

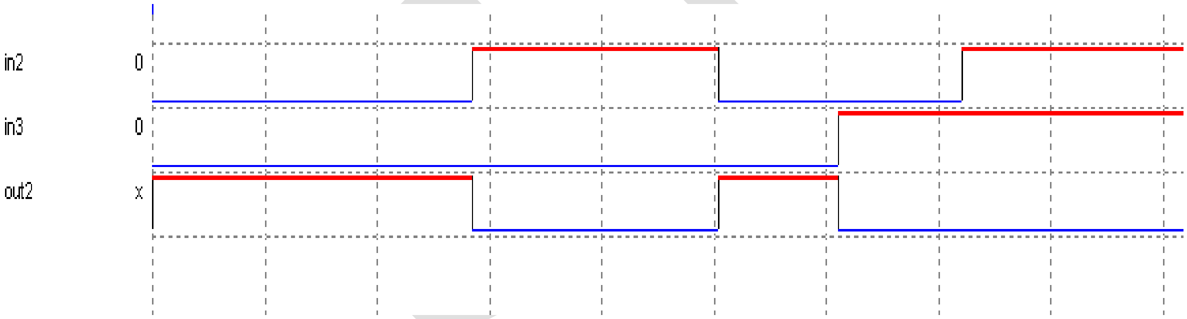
Digital Schematic Representation:



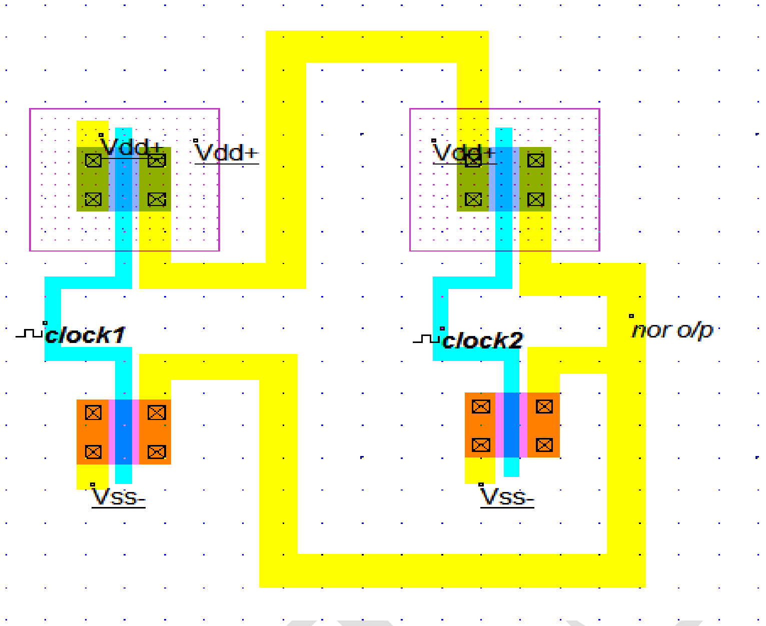
Transistor level design (CMOS logic):



Timing Diagram:

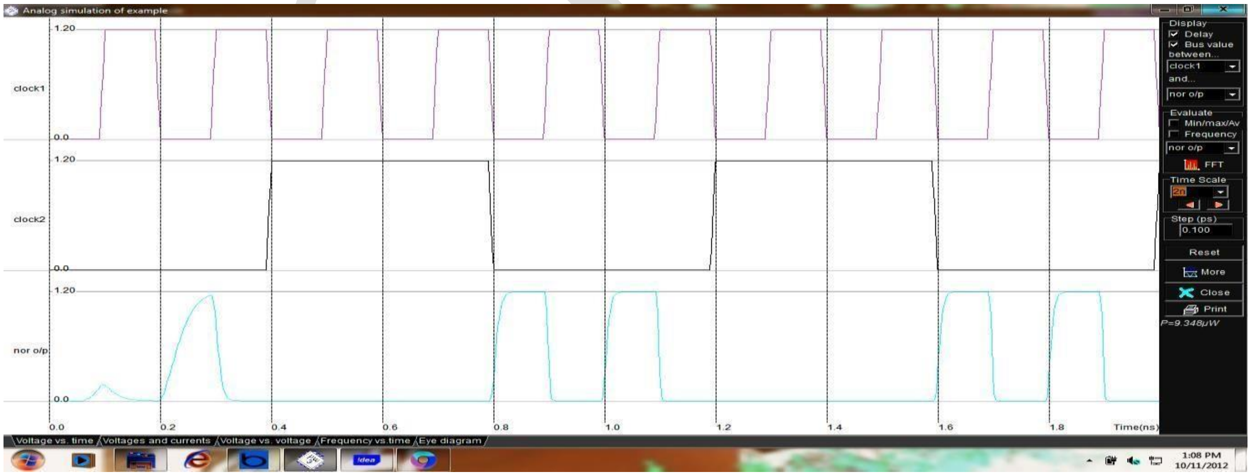


Semi-custom Layout:



Simulation:

Voltage –Time:



Display
☑ Delay
☑ Bus value
between... clock1
and... nor o/p

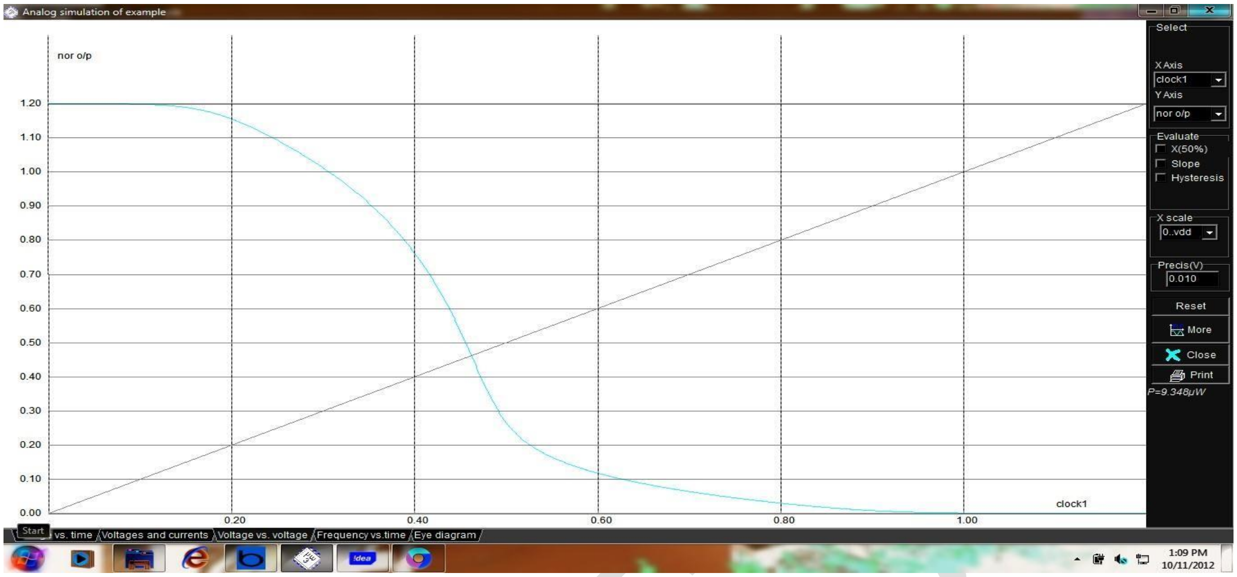
Evaluate
□ Min/max/Av
□ Frequency
nor o/p

Time Scale 2n

Step (ps) 0.100

P=9.348µW

Voltage-voltage:



X Axis: clock1
Y Axis: nor o/p
□ X(50%)
□ Slope
□ Hysteresis
X scale: 0..vdd
Precis(V): 0.010
P=9.348µW

Conclusion:

The digital schematics and corresponding layouts using **CMOS logic** for an **NOR LOGIC** gate are designed and the lambda based rules using DRC are checked and verified its functionality.

Experiment 2(c)

EX-OR LOGIC GATE

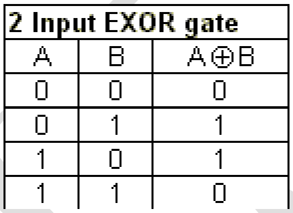
Aim: To design the digital schematics and corresponding layouts using **CMOS logic** for an **EX-OR LOGIC** gate, **EX-NOR LOGIC** gate and check the lambda based rules using DRC and verify its functionality.

Apparatus:

- DSCH2(logic editor & simulator)
- MICROWIND 3.1(layout editor & simulator)

THEORY:

EX-OR: The '**Exclusive-OR**' gate is a circuit which will give a high output if **either, but not both**, of its two inputs are high. An encircled plus sign ( ) is used to show the EXOR operation.

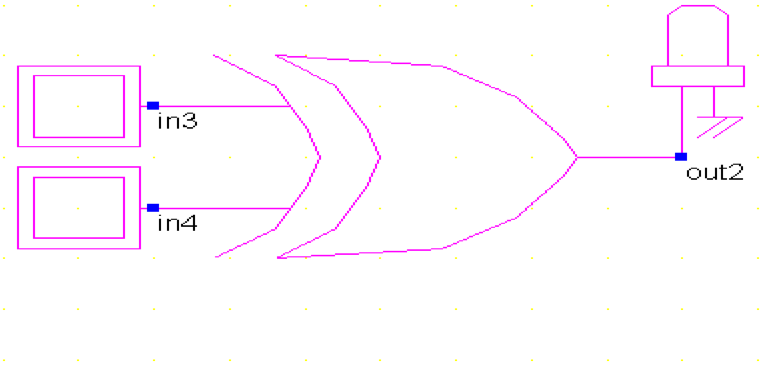


2 Input EXOR gate		
A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

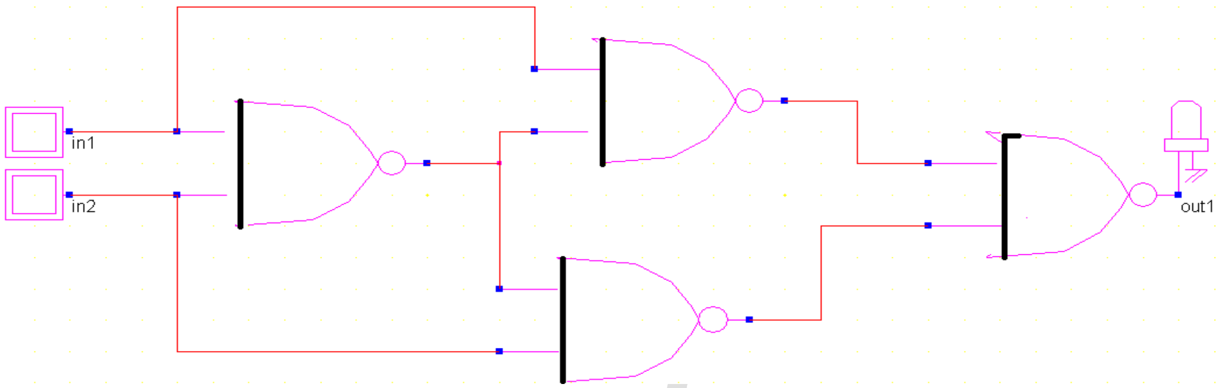
Procedure:

1. Open the DSch2 tool and draw the schematic diagram as per the circuit drawn.
2. Save the file and verify the functionality.
3. After that open Microwind 3.1 tool and draw the layout diagram as per the circuit drawn.
4. Save the file and verify the lambda rules by using DRC, then verify the functionality.

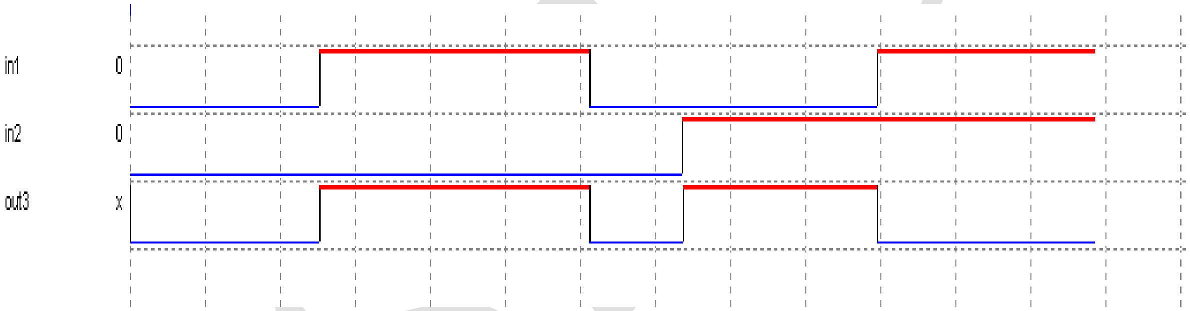
Digital Schematic Representation:



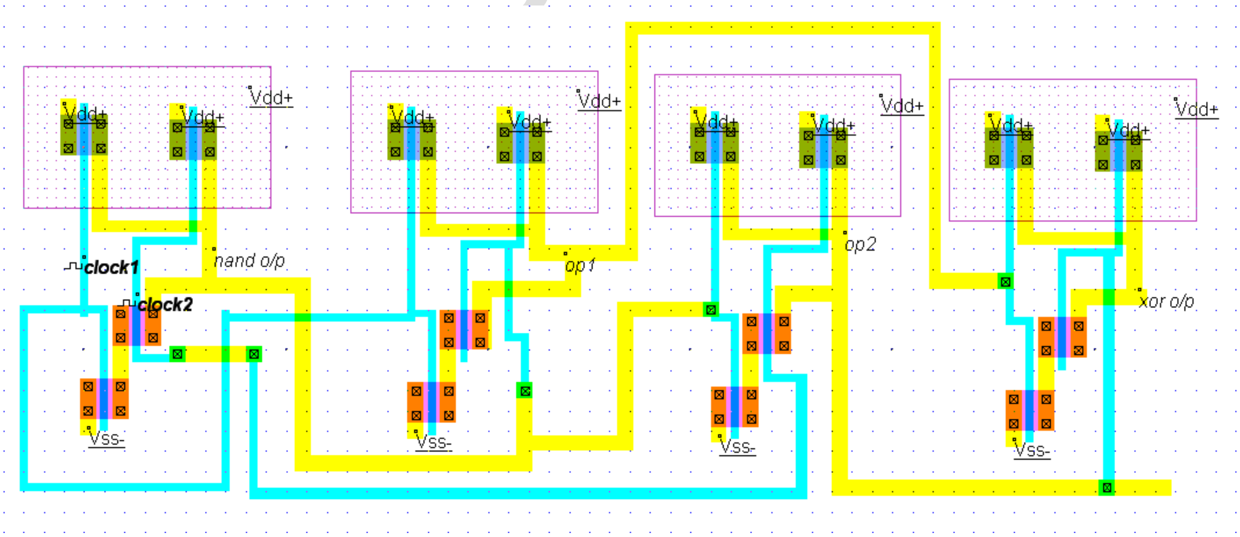
Transistor level design (CMOS logic):



Timing Diagram:

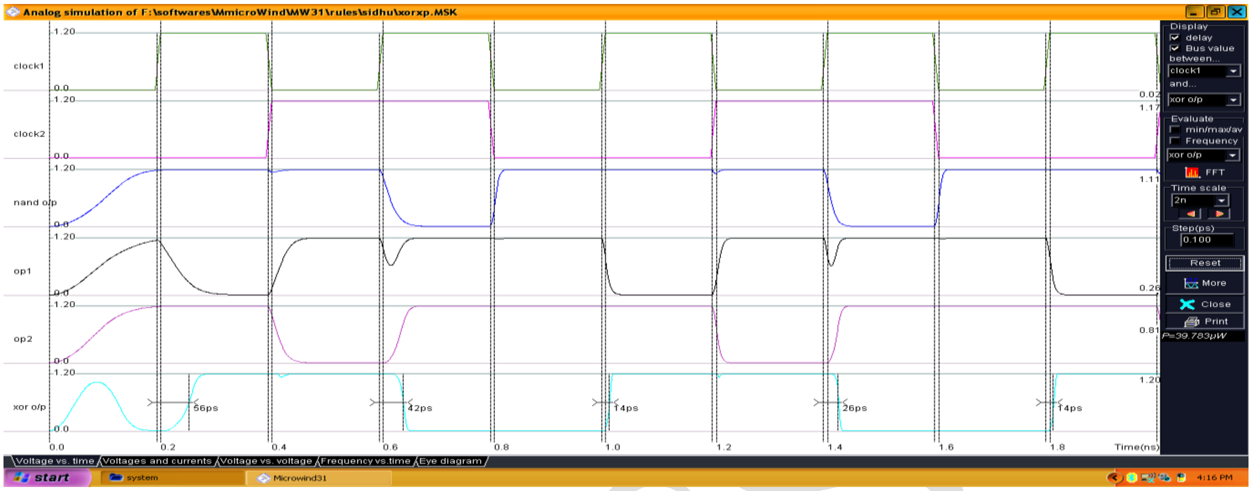


Semi-custom Layout:

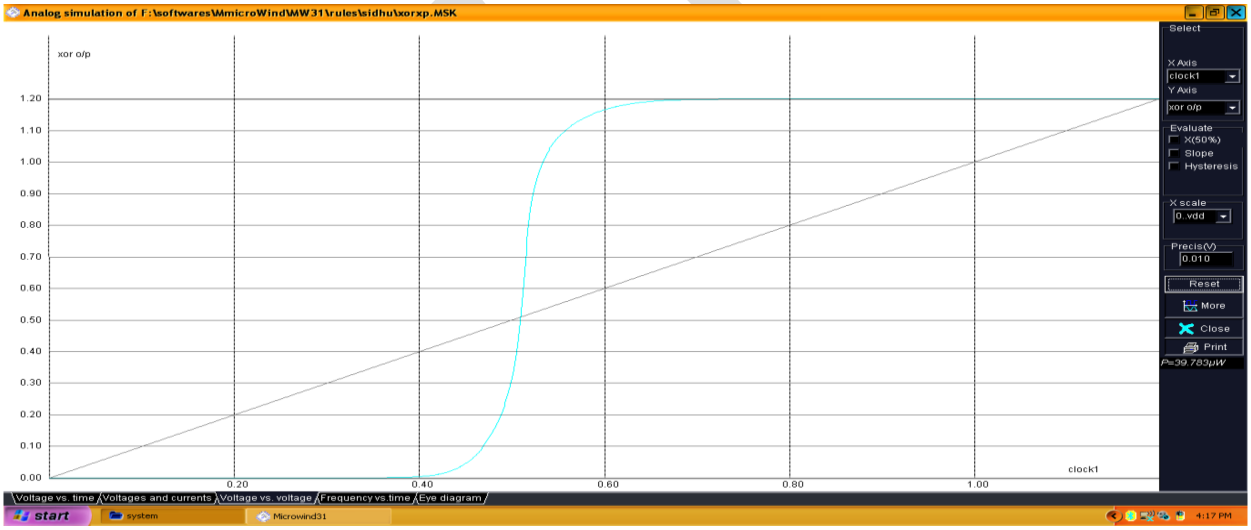


Simulation:

Voltage –Time:



Voltage-voltage:



Conclusion:

The digital schematics and corresponding layouts using **CMOS logic** for an **EX-OR LOGIC** gate are designed and the lambda based rules using DRC are checked and verified its functionality.

Ex-NOR LOGIC GATE:

Ex-NOR Gate:

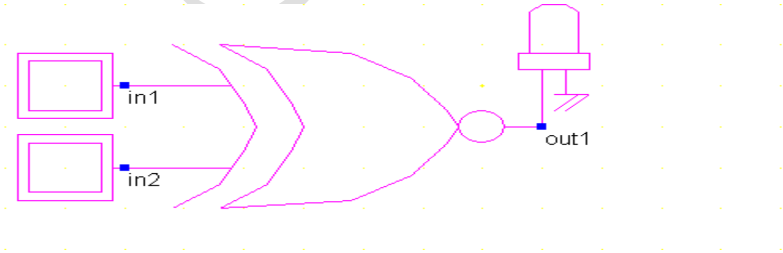
The '**Exclusive-NOR**' gate is a circuit which will give a high output if **both** of its inputs are high or low. An encircled dot sign (.) is used to show the EXNOR operation.

A	B	~ (a^b)
0	0	1
0	1	0
1	0	0
1	1	1

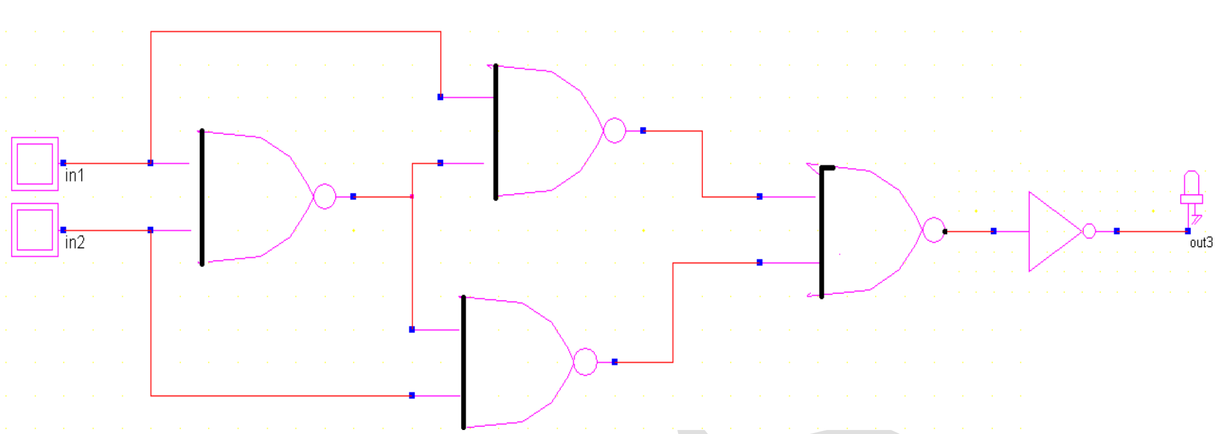
Procedure:

1. Open the DSch2 tool and draw the schematic diagram as per the circuit drawn.
2. Save the file and verify the functionality.
3. After that open Microwind 3.1 tool and draw the layout diagram as per the circuit drawn.
4. Save the file and verify the lambda rules by using DRC, then verify the functionality.

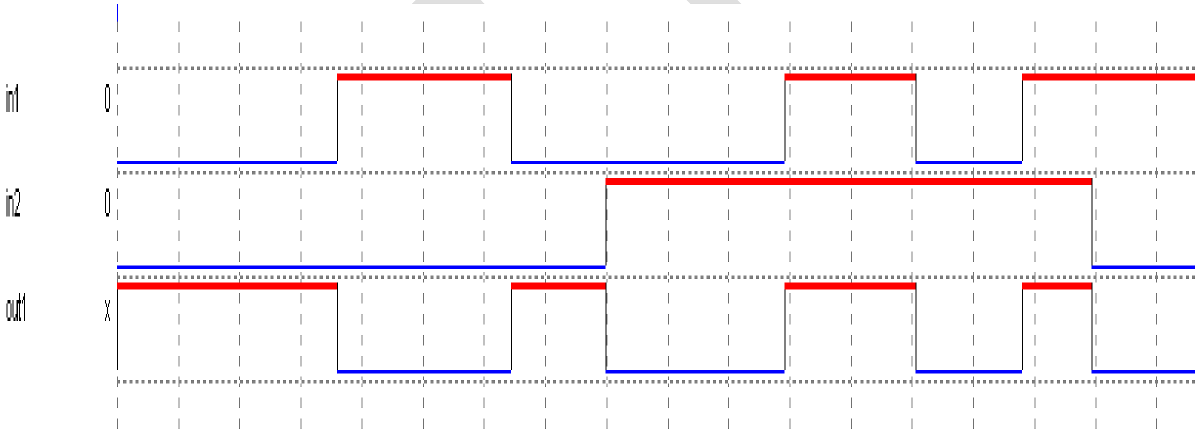
Digital Schematic Representation:



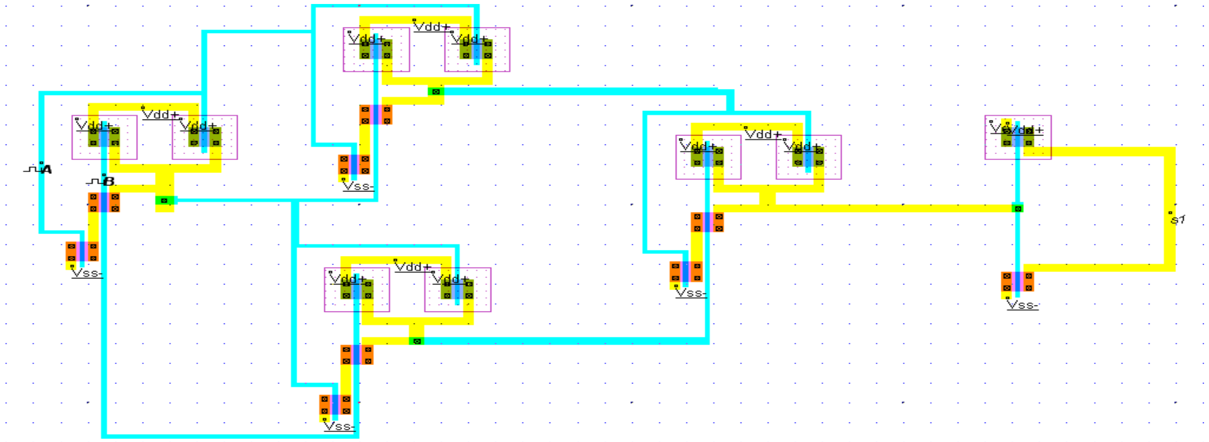
Transistor level design (CMOS logic):



Timing diagram:

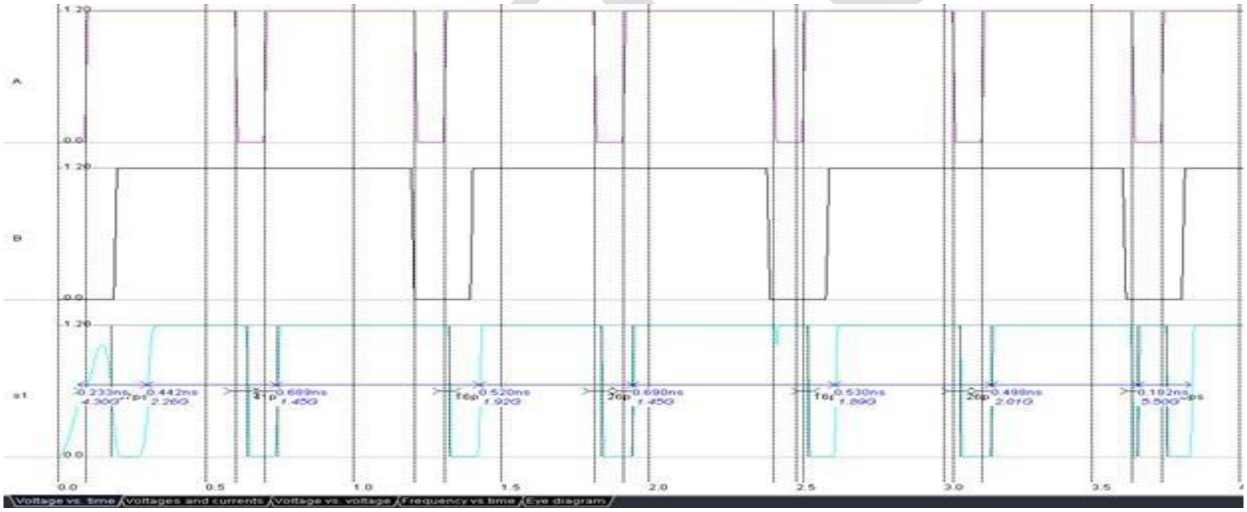


Semi-custom Layout:



Simulation:

Voltage –Time:



Conclusion:

The digital schematics and corresponding layouts using **CMOS logic** for an **EX-NOR LOGIC** gate are designed and the lambda based rules using DRC are checked and verified its functionality.

Experiment 3

HALF ADDER

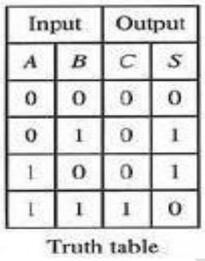
Aim: To design the digital schematics and corresponding layouts using **CMOS logic** for **HALF ADDER** and check the lambda based rules using DRC and verify its functionality.

Apparatus:

- DSCH2(logic editor & simulator)
- MICROWIND 3.1(layout editor & simulator)

Theory:

The **half adder** adds two one-bit binary numbers *A* and *B*. It has two outputs, *S* and *C*



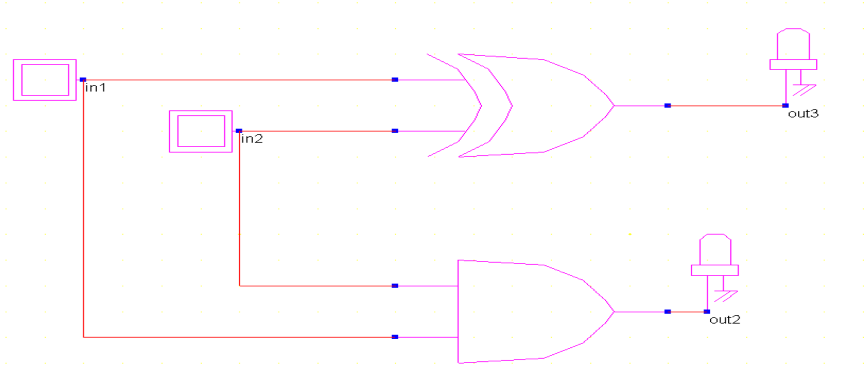
Input		Output	
A	*B*	*C*	*S*
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

. Truth table

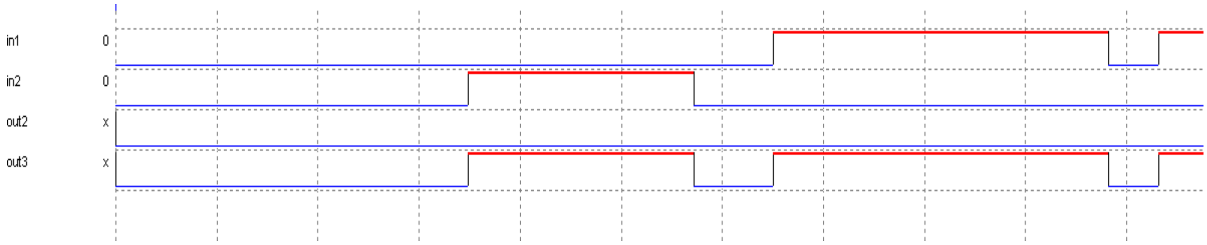
Procedure:

1. Open the DSch2 tool and draw the schematic diagram as per the circuit drawn.
2. Save the file and verify the functionality.
3. After that open Microwind 3.1 tool and draw the layout diagram as per the circuit drawn.
4. Save the file and verify the lambda rules by using DRC, then verify the functionality.

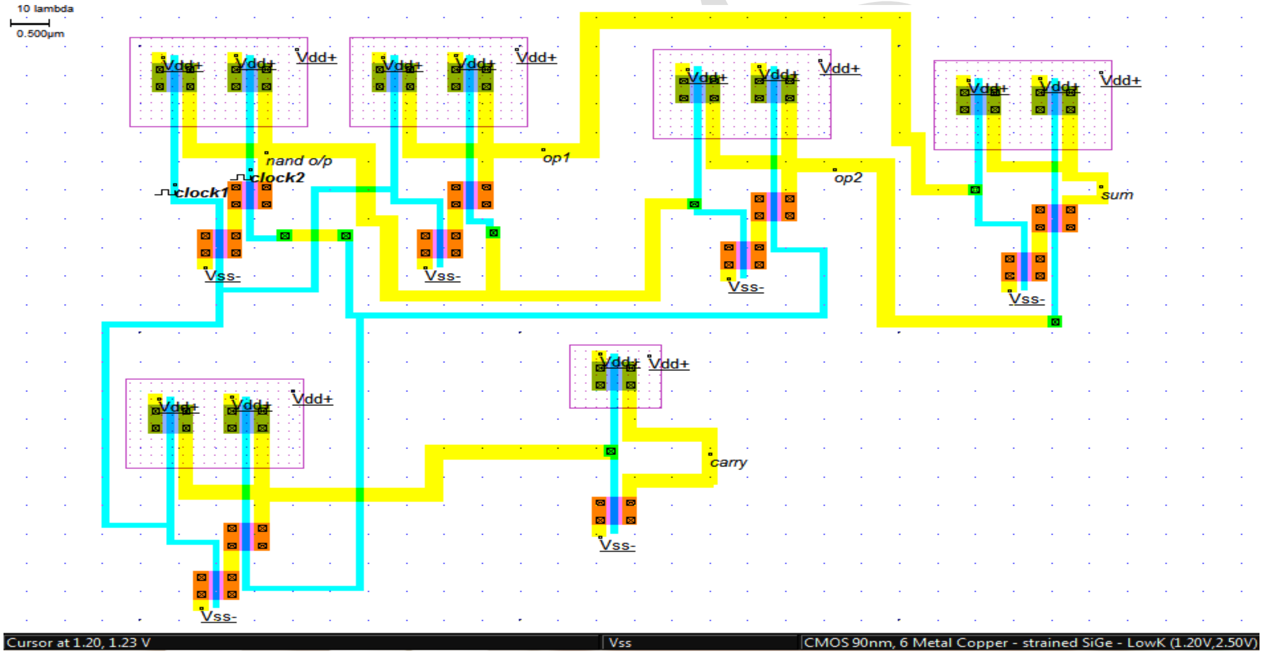
Transistor level design (CMOS logic):



Timing Diagram:

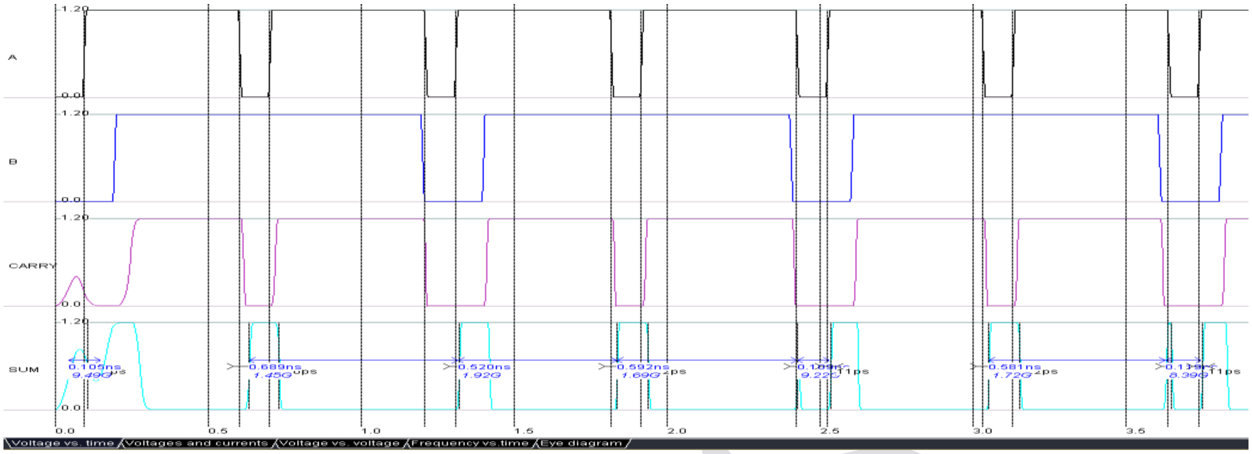


Semi-custom Layout:

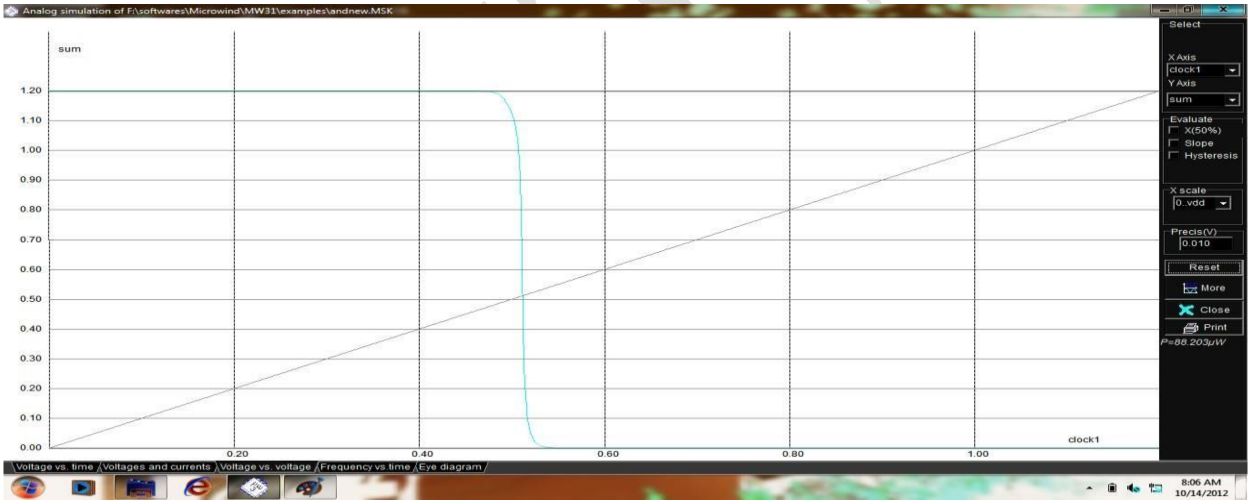


Simulation:

Voltage –Time:



Voltage-Voltage:


Analog simulation of F:\softwares\Microwind\MW31\examples\andnew.MSK

Select

X Axis: clock1

Y Axis: sum

Evaluate

□ X(50%)

□ Slope

□ Hysteresis

X scale: 0..vdd

Precis(V): 0.010

P=88.203µW

Conclusion:

The digital schematics and corresponding layouts using **CMOS logic** for an **HALF ADDER** are designed and the lambda based rules using DRC

Experiment 4

SPICE Simulation and a Coding of CMOS Inverter Circuit

Aim: To write SPICE code for CMOS Inverter Circuit, Simulate and verify functionality.

Apparatus:

1. PSPICE

   i. DesignLab Eval8

Theory:

In CMOS, both p and n-channel transistors are used. A schematic circuit representation of the CMOS inverter is shown in figure. The operation of the circuit on an inverter can be explained as follows. All voltages are referenced with respect to $V_{SS,}$ the ground potential. When the input voltage V_I is zero, the gate of the p-channel transistor is at V_{DD} below the source potential, that is, V_{GS}=V_{DD}. This turns on the transistor, which is turned off since V_{GS}=0 for this transistor. Now if the input voltage is raised to the threshold voltage level of the n-channel transistor raised to V_{DD}, the n-channel transistor will conduct while the p-channel transistor gets turned off, discharging the load capacitance C to ground potential.

Procedure:

1. Start → program →Design Lab Eval8 → select Design manager to get Design Manager window
2. Click on Run Text Edit window to get microsim text editor
3. Type the program, save it with experiment name.
4. Then run pspice AD, to get pspice AD window.
5. Then go to file, click on open to select the saved file
6. The selected file is simulated successfully.
7. Go to file, click Run Probe to get microsim probe window.
8. Click on Add Trace, Deselect Currents and Aliased names and click on OK to view the frequency response.

Circuit Diagram:

CMOS Inverter:

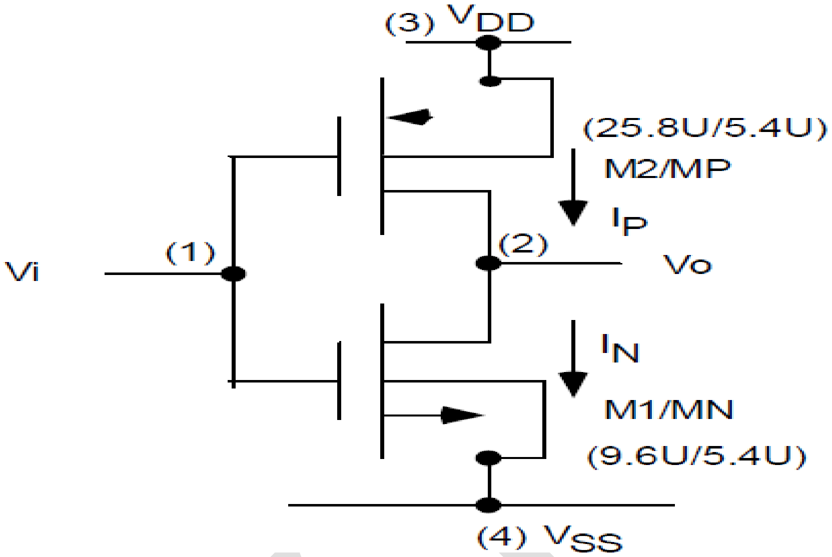


Figure: CMOS Inverter:

PSPICE Code for CMOS Inverter:

```
*Pspice file for CMOS Inverter
*Filename=”cmos.cir”
VIN 1 0 DC 0V AC 1VOLT
VDD 3 0 DC 2.5VOLT
VSS 4 0 DC -2.5VOLT
M1 2 1 4 4 NMOS1 W=9.6U L=5.4U
M2 2 1 3 3 PMOS1 W=25.8U L=5.4U
.MODEL NMOS1 NMOS VTO=1.0 KP=40U
+ GAMMA=1.0 LAMBDA=0.02 PHI=0.6
+ TOX=0.05U LD=0.5U CJ=5E-4 CJSW=10E-10
+ U0=550 MJ=0.5 MJSW=0.5 CGSO=0.4E-9 CGDO=0.4E-9
.MODEL PMOS1 PMOS VTO=-1.0 KP=15U
+ GAMMA=0.6 LAMBDA=0.02 PHI=0.6
+ TOX=0.05U LD=0.5U CJ=5E-4 CJSW=10E-10
+ U0=200 MJ=0.5 MJSW=0.5 CGSO=0.4E-9 CGDO=0.4E-9
.DC VIN -2.5 2.5 0.05
.TF V(2) VIN
.AC DEC 100 1HZ 100GHZ
.PROBE
.END
```

CMOS Inverter Transfer function:

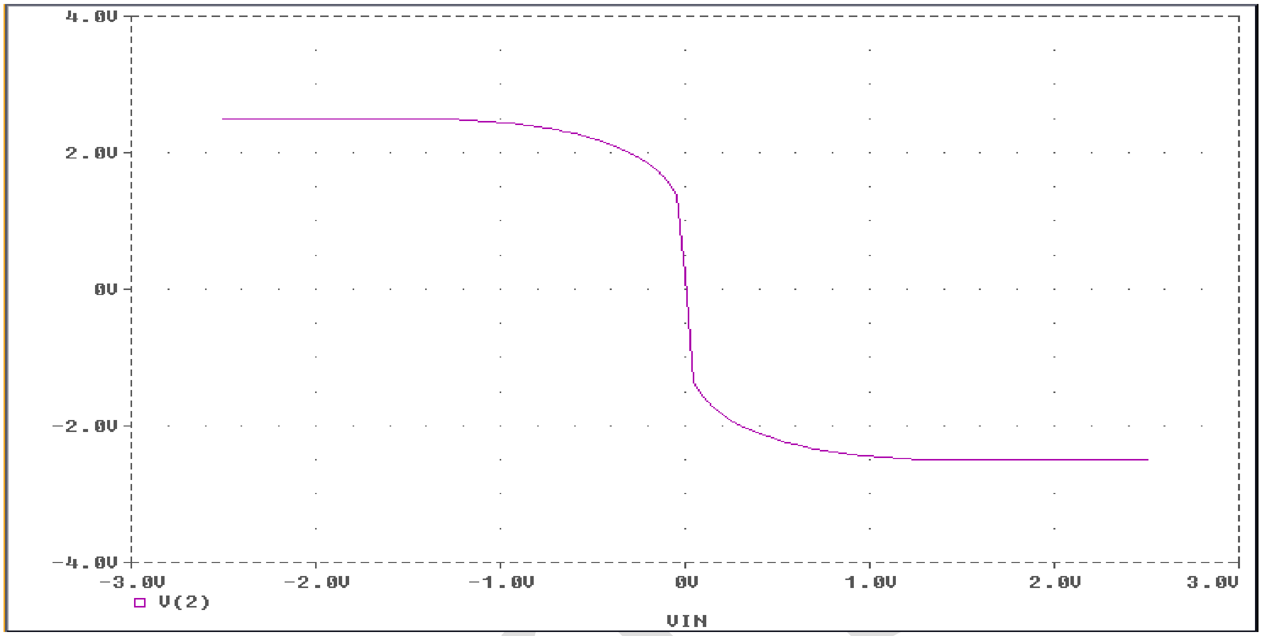


Figure: CMOS inverter transfer function

Conclusion:

The SPICE code for CMOS Inverter Circuit is written, simulated and the functionality is verified.

Experiment 5

SPICE Simulation of Basic Analog Circuit: Differential Amplifier

Aim: To write SPICE code for Differential Amplifier, Simulate and verifyfunctionality.

Apparatus:

1. PSPICE

    i. DesignLab Eval8

Theory:

Differential amplifiers are compatible with the matching properties of IC technology. The differential amplifier has two modes of signal operation:

i. Differential mode,
ii. Common mode.

Differential amplifiers are excellent input stages for voltage amplifiers Differential amplifiers can have different loads including:

- Current mirrors
- MOS diodes
- Current sources/sinks
- Resistors

The small signal performance of the differential amplifier is similar to the inverting amplifier in gain, output resistance and bandwidth. The large signal performance includes slew rate and the linearization of the transconductance. The design of CMOS analog circuits uses the relationships of the circuit to design the dc currents and the W/L ratios of each transistor.

A differential amplifier is an amplifier that amplifies the difference between two voltages and rejects the average or common mode value of the two voltages. Differential and common mode voltages: *v*1 and *v*2 are called *single-ended* voltages. They are voltages referenced to ac ground. The *differential-mode* input voltage, *vID*, is the voltage difference between *v*1 and *v*2. The *common-mode* input voltage, *vIC*, is the average value of *v*1 and *v*2 .

Procedure: refer to page 79

Circuit Diagram:

CMOS Differential Amplifier:

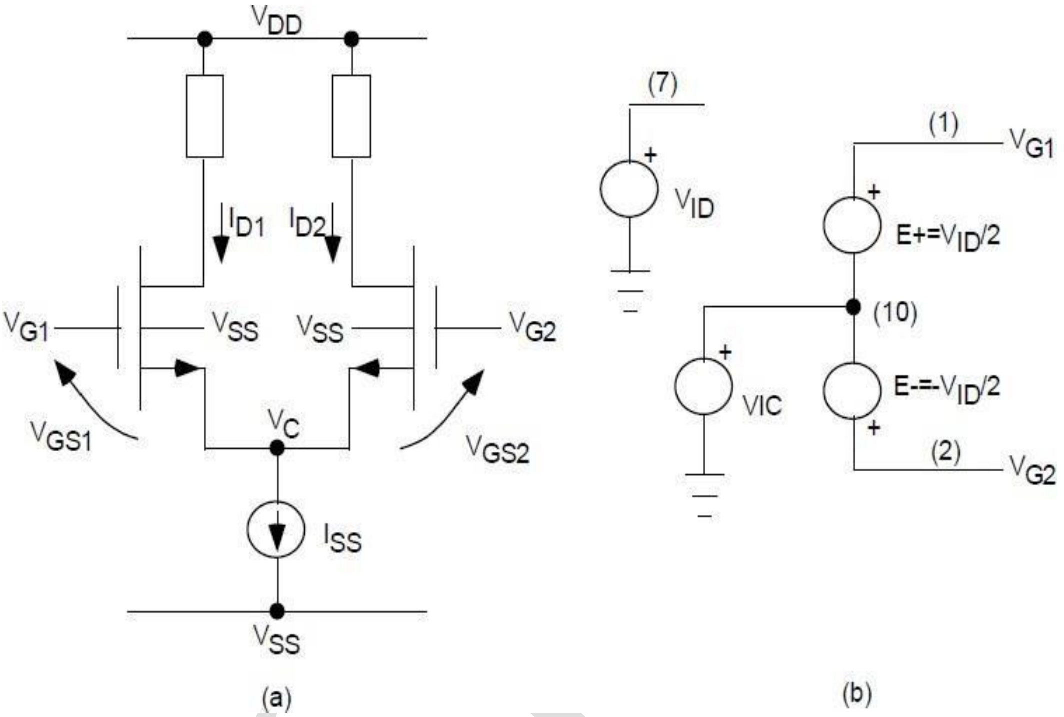


Figure: General MOS Differential Amplifier: (a) Schematic Diagram, (b) Input Gate Voltages Implementation.

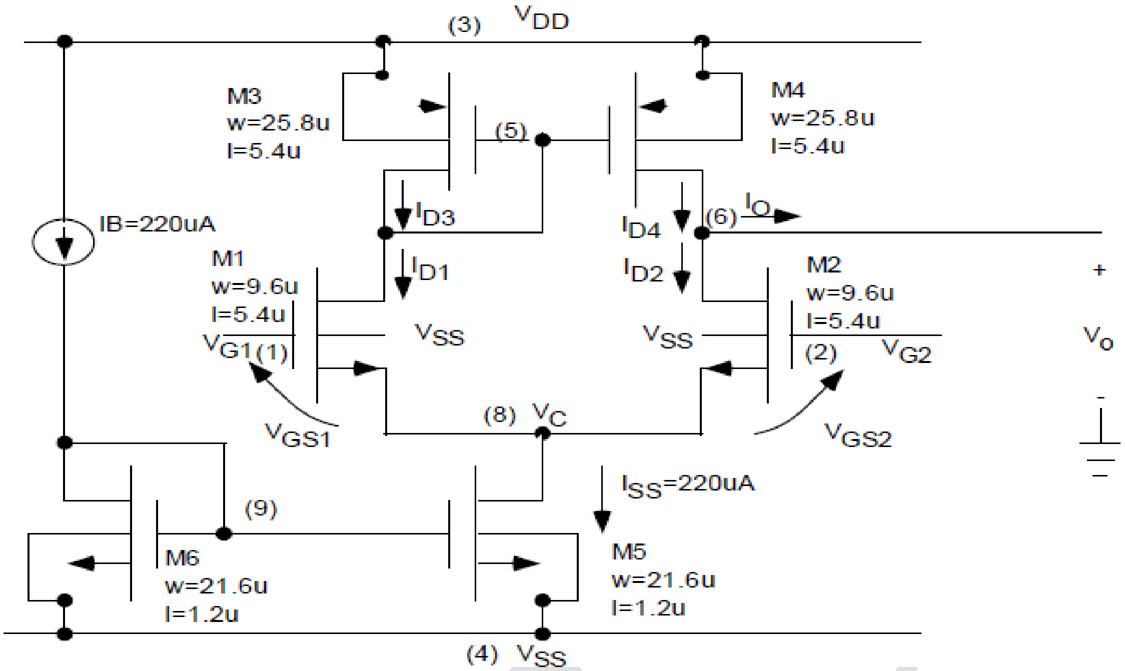


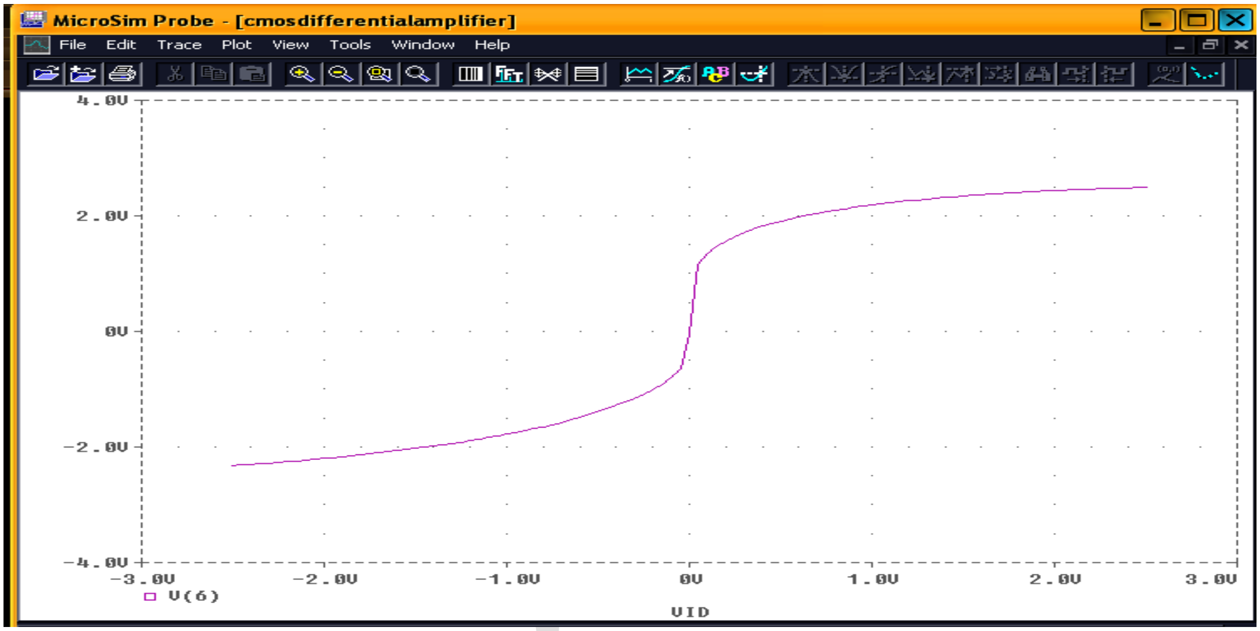
Figure: The Complete Differential Amplifier Schematic Diagram

PSPICE Code for CMOS Differential Amplifier:

```
* Filename="diffvid.cir"
* MOS Diff Amp with Current Mirror Load
*DC Transfer Characteristics vs VID
VID 7 0 DC 0V AC 1V
E+ 1 10 7 0 0.5
E- 2 10 7 0 -0.5
VIC 10 0 DC 0.65V
VDD 3 0 DC 2.5VOLT
VSS 4 0 DC -2.5VOLT
M1 5 1 8 8 NMOS1 W=9.6U L=5.4U
M2 6 2 8 8 NMOS1 W=9.6U L=5.4U
M3 5 5 3 3 PMOS1 W=25.8U L=5.4U
M4 6 5 3 3 PMOS1 W=25.8U L=5.4U
M5 8 9 4 4 NMOS1 W=21.6U L=1.2U
M6 9 9 4 4 NMOS1 W=21.6U L=1.2U
IB 3 9 220UA
.MODEL NMOS1 NMOS VTO=1 KP=40U
+ GAMMA=1.0 LAMBDA=0.02 PHI=0.6
+ TOX=0.05U LD=0.5U CJ=5E-4 CJSW=10E-10
+ U0=550 MJ=0.5 MJSW=0.5 CGSO=0.4E-9 CGDO=0.4E-9
```

```
.MODEL PMOS1 PMOS VTO=-1 KP=15U
+ GAMMA=0.6 LAMBDA=0.02 PHI=0.6
+ TOX=0.05U LD=0.5U CJ=5E-4 CJSW=10E-10
+ U0=200 MJ=0.5 MJSW=0.5 CGSO=0.4E-9 CGDO=0.4E-9
.DC VID -2.5 2.5 0.05V
.TF V(6) VID
.PROBE
.END
```

CMOS Differential Amplifier Transfer function:


MicroSim Probe - [cmosdifferentialamplifier]
File Edit Trace Plot View Tools Window Help
4.0V
2.0V
0V
-2.0V
-4.0V
-3.0V
-2.0V
-1.0V
0V
1.0V
2.0V
3.0V
□ V(6)
VID

Conclusion:

The SPICE code for CMOS Differential Amplifier is written, simulated and the functionality is verified.

Experiment 6(a)

Analog Circuit Simulation (AC analysis) – Common Source Amplifier

Aim: To write SPICE code for Common Source Amplifier, Simulate and verify the functionality.

Apparatus:

1. PSPICE

    i. DesignLab Eval8

Theory:

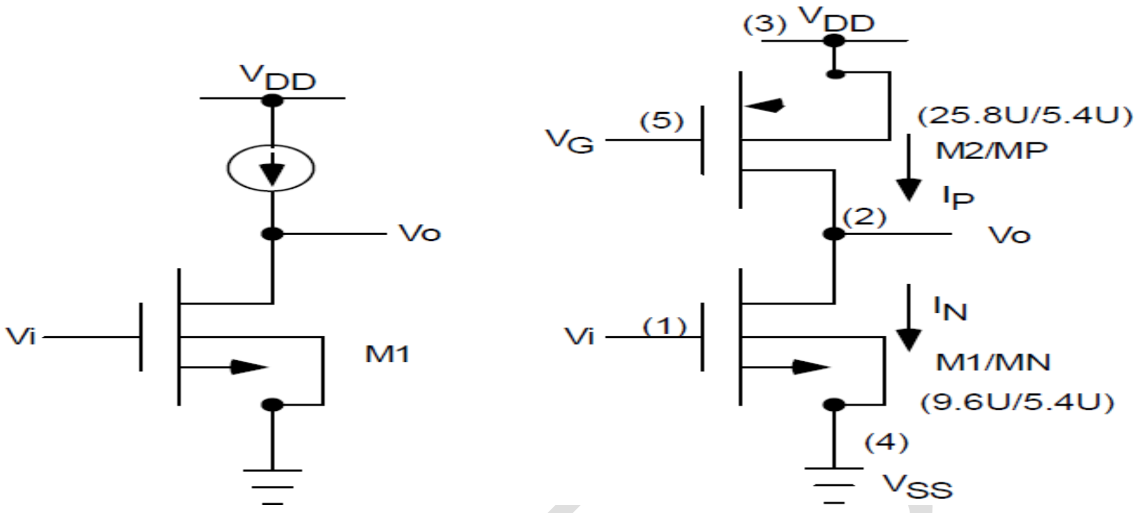
A common Source amplifier is one of three basic single-stage MOSFET amplifier topologies, typically used as a voltage or transconductance amplifier. The easiest way to tell if a MOSFET is common source, common drain, or common gate is to examine where the signal enters and leaves. The remaining terminal is what is known as “common”. The signal enters the gate, and exits the drain. The only terminal is the source. This is a common-source MOSFET. The analogous bipolar junction transistor circuit is the common-emitter amplifier.

The common-source (CS) amplifier may be viewed as a transconductance amplifier or as a voltage amplifier. As a transconductance amplifier, the input voltage is seen as modulating the current going to the load. As a voltage amplifier, input voltage modulates the amount of current flowing through the mosfet, changing the voltage across the output resistance according to Ohm’s law. However, the MOSFET device’s output resistance typically not high enough for a reasonable transconductance amplifier (ideally infinite), nor low enough for a decent voltage amplifier (ideally zero). Another major drawback is the amplifier’s limited high-frequency response. Therefore, in practice the output often is routed through either a voltage follower (common-drain stage), or a current follower (common-gate stage), or a current follower (common-gate stage) to obtainmore output and frequency characteristics. The CS-CG combination is called a cascade amplifier.

Procedure: refer to page 79

Circuit Diagram:

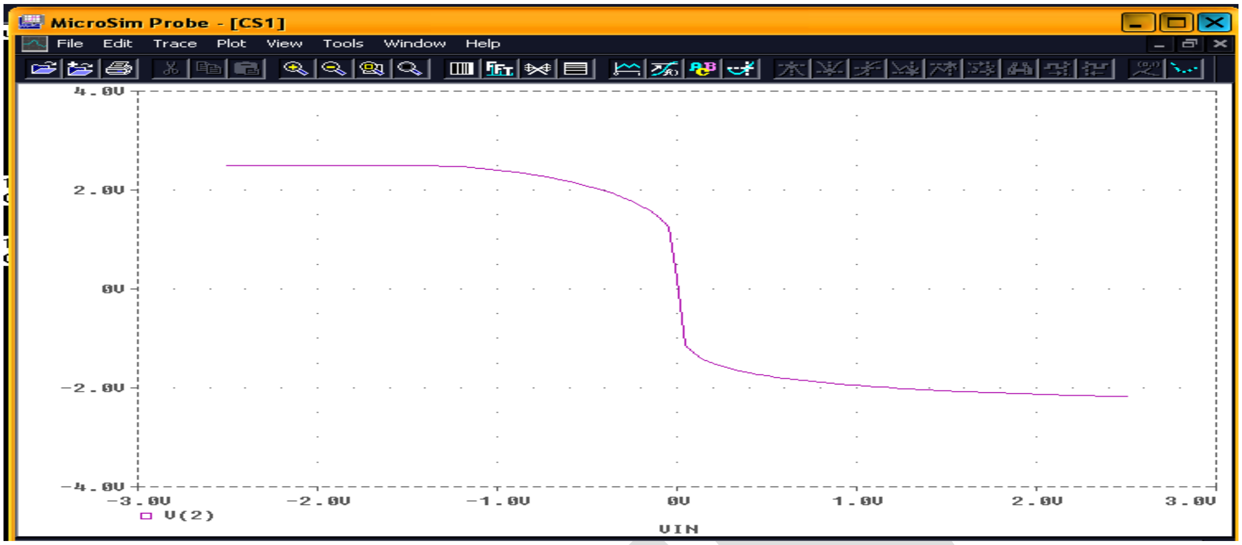
Common Source Amplifier:



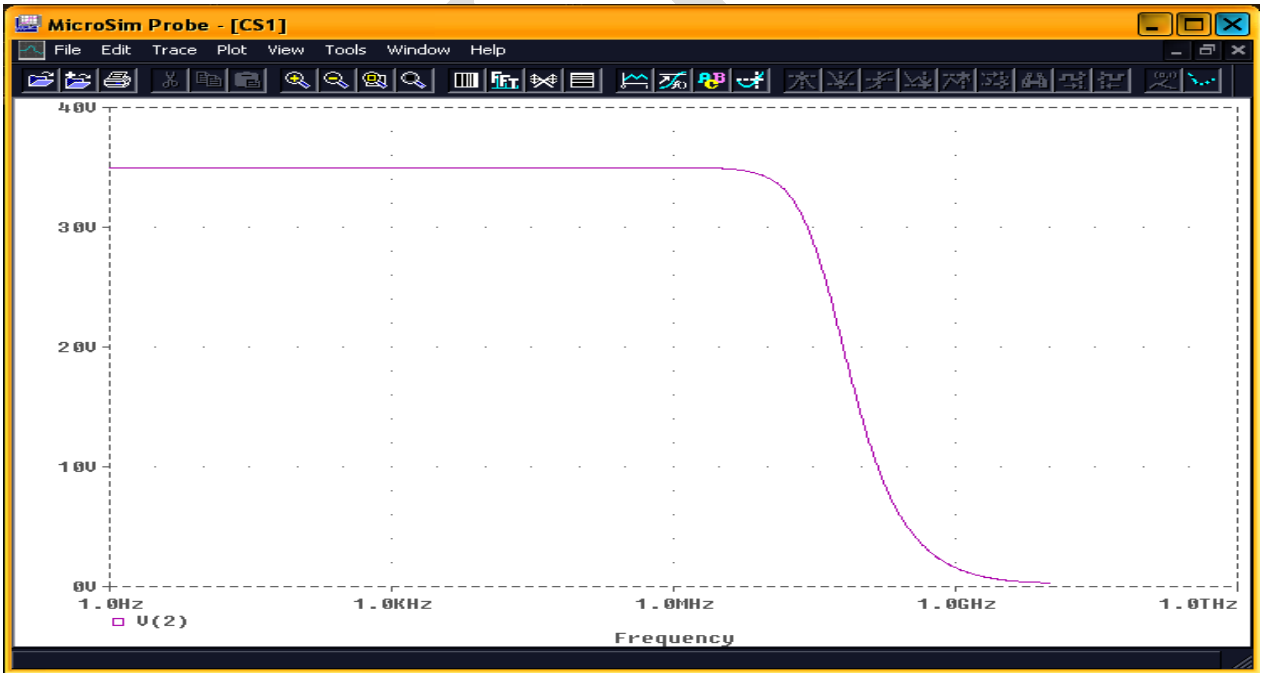
PSPICE Code for CMOS Common Source Amplifier:

```
*PSpice file for NMOS Inverter with PMOS Current Load
*Filename="Lab3.cir"
VIN 1 0 DC 0VOLT AC 1V
VDD 3 0 DC 2.5VOLT
VSS 4 0 DC -2.5VOLT
VG 5 0 DC 0VOLT
M1 2 1 4 4 MN W=9.6U L=5.4U
M2 2 5 3 3 MP W=25.8U L=5.4U
.MODEL MN NMOS VTO=1 KP=40U
+ GAMMA=1.0 LAMBDA=0.02 PHI=0.6
+ TOX=0.05U LD=0.5U CJ=5E-4 CJSW=10E-10
+ U0=550 MJ=0.5 MJSW=0.5 CGSO=0.4E-9 CGDO=0.4E-9
.MODEL MP PMOS VTO=-1 KP=15U
+ GAMMA=0.6 LAMBDA=0.02 PHI=0.6
+ TOX=0.05U LD=0.5U CJ=5E-4 CJSW=10E-10
+ U0=200 MJ=0.5 MJSW=0.5 CGSO=0.4E-9 CGDO=0.4E-9
*Analysis
.DC VIN -2.5 2.5 0.05
.TF V(2) VIN
.PROBE
.END
```

CMOS Common Source DC analysis:



CMOS Common Source AC analysis:



Conclusion:

The PSPICE code for CMOS Common Source Amplifier is written, simulated (AC analysis) and the functionality is verified.

Experiment 6(b)

Analog Circuit Simulation (AC analysis) – Common Drain Amplifier

Aim: To write SPICE code for Common Drain Amplifier, Simulate and verify the functionality.

Apparatus:

1. PSPICE

    ii. DesignLab Eval8

Theory:

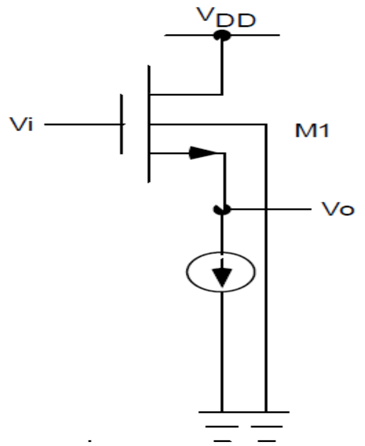
A common-drain amplifier, also known as a source follower, is one of three basic single-stage MOSFET amplifier topologies, typically used as a voltage buffer. In the circuit the gate terminal of the transistor serves as the input, the source is the output, and the drain is common to both (input and output), hence its name. The analogous bipolar junction transistor circuit is the common-collector amplifier.

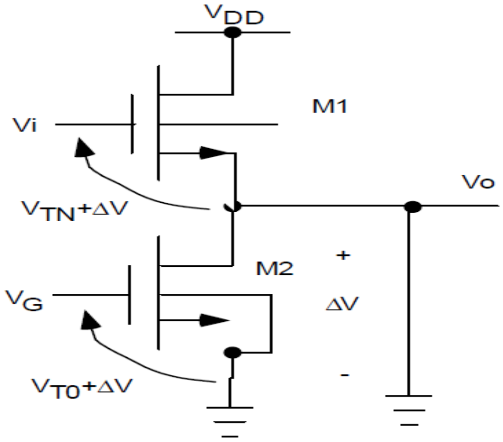
In addition, this circuit is used to transform impedances. For example, the Thévenin resistance of a combination of a voltage follower driven by a voltage source with high Thévenin resistance is reduced to only the output resistance of the voltage follower, a small resistance. That resistance reduction makes the combination a more ideal voltage source. Conversely, a voltage follower inserted between a driving stage and a high load (i.e. a low resistance) presents an infinite resistance (low load) to the driving stage, an advantage in coupling a voltage signal to a large load.

Procedure: refer to page 79

Circuit Diagram:

Common Drain Amplifier:

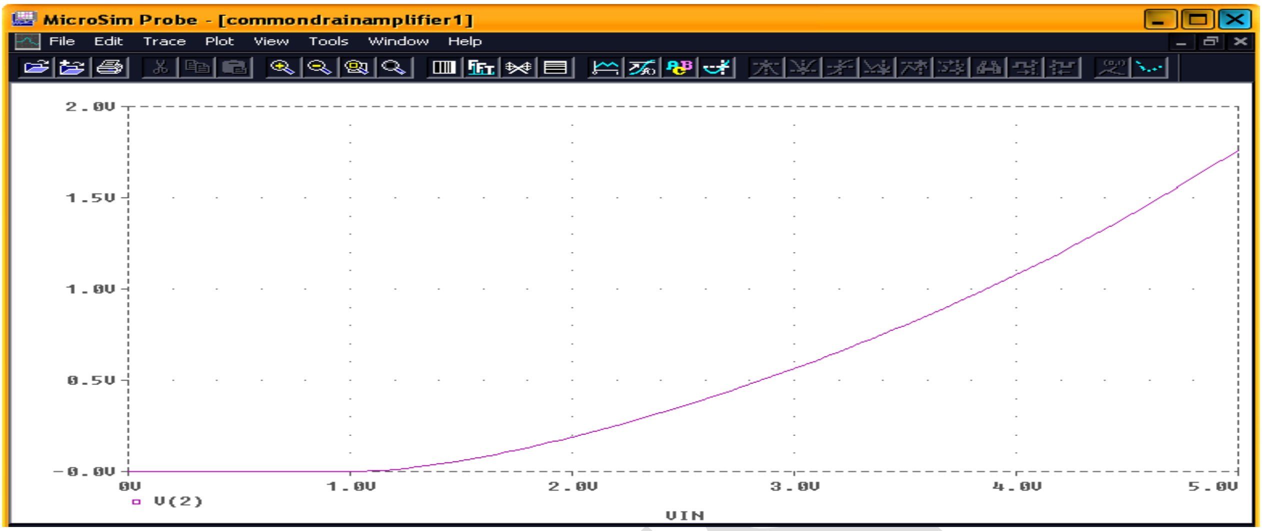




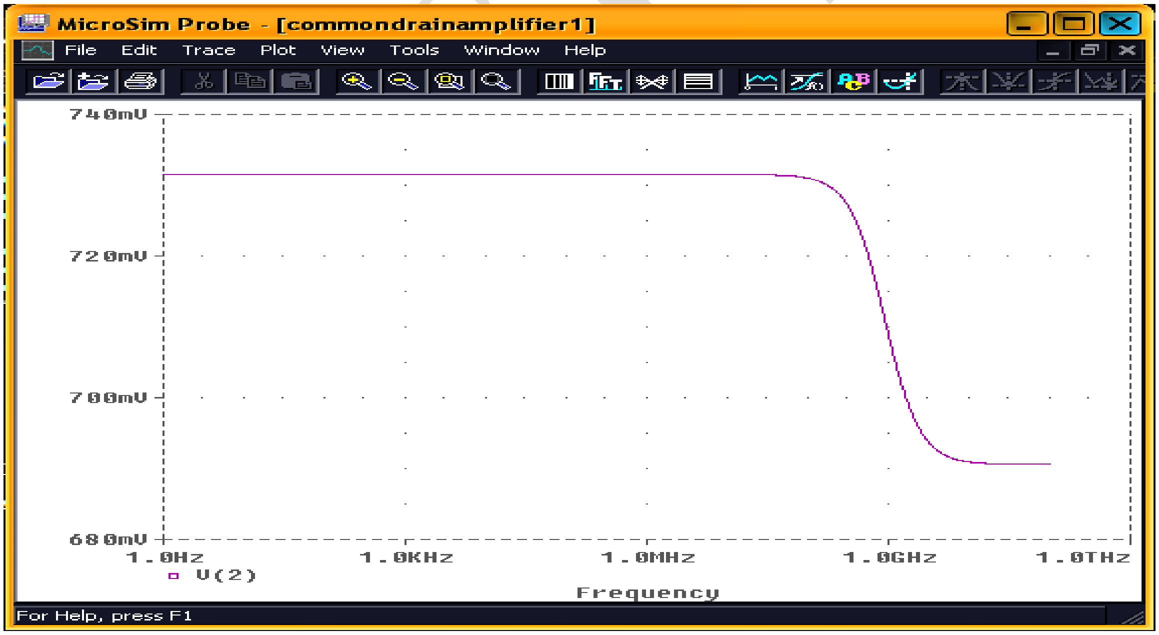
PSPICE Code for CMOS Differential Amplifier:

```
*PSpice file for NMOS Inverter with PMOS Current Load
*Filename="Lab3.cir"
VIN 1 0 DC 4.75VOLT AC 1V
VDD 3 0 DC 5VOLT
VSS 4 0 DC 0VOLT
VG2 5 0 DC 2.5VOLT
M1 3 1 2 4 MN W=9.6U L=5.4U
M2 2 5 4 4 MN W=9.6U L=5.4U
.MODEL MN NMOS VTO=1 KP=40U
+ GAMMA=1.0 LAMBDA=0.02 PHI=0.6
+ TOX=0.05U LD=0.5U CJ=5E-4 CJSW=10E-10
+ U0=550 MJ=0.5 MJSW=0.5 CGSO=0.4E-9 CGDO=0.4E-9
.MODEL MP PMOS VTO=-1 KP=15U
+ GAMMA=0.6 LAMBDA=0.02 PHI=0.6
+ TOX=0.05U LD=0.5U CJ=5E-4 CJSW=10E-10
+ U0=200 MJ=0.5 MJSW=0.5 CGSO=0.4E-9 CGDO=0.4E-9
*Analysis
.DC VIN 0 5 0.05
.TF V(2) VIN
.AC DEC 100 1HZ 100GHZ
.PROBE
.END
```

Common Drain Amplifier DC analysis:


MicroSim Probe - [commondrainamplifier1]
File Edit Trace Plot View Tools Window Help
2.0V
1.5V
1.0V
0.5V
-0.0V
0V
1.0V
2.0V
3.0V
4.0V
5.0V
□ V(2)
VIN

Common Drain Amplifier AC analysis:


MicroSim Probe - [commondrainamplifier1]
File Edit Trace Plot View Tools Window Help
740mV
720mV
700mV
680mV
1.0Hz
1.0KHz
1.0MHz
1.0GHz
1.0THz
□ V(2)
Frequency
For Help, press F1

Conclusion:

The PSPICE code for CMOS Common Drain Amplifier is written, simulated (AC analysis) and the functionality is verified.